SONDERDRUCK FÜR CQSE

Dr. Sven Amann und Dr. Elmar Jürgens

Change-Driven-Testing

Change-Driven-Testing

Dr. Sven Amann und Dr. Elmar Jürgens – CQSE GmbH

***Zusammenfassung*—Software-Tester stehen zunehmend vor der Herausforderung, immer mehr Software in immer kürzerer Zeit zu testen. Daher ist es unmöglich, einfach die gesamte Test-Suite für jede Änderung der Software auszuführen. Und es ist ebenso nicht länger möglich manuell sicherzustellen, dass die durchgeführten Tests alle Änderungen abdecken. Unsere Antwort auf diese Herausforderung heißt Change-Driven-Testing. Change-Driven-Testing setzt Test-Impact-Analyse ein, um die relevanten Tests für eine gegebene Code-Änderung automatisiert zu identifizieren und so zu sortieren, dass Fehler frühstmöglich entdeckt werden. Dies macht Testen effizienter, sodass wir über 90% aller Fehler in nur 2% der Testausführungszeit abfangen können. Zudem setzt Change-Driven-Testing Test-Gap-Analyse ein, um Code-Änderungen, die nicht getestet wurden, sogenannte Test-Gaps, automatisiert zu identifizieren. Dies macht Testen effektiver, indem es uns Fakten zur strategischen Steuerung unserer begrenzten Test-Ressourcen an die Hand gibt und fehlende Regressionstests aufzeigt.**

I. Ein Teufelskreis

In der heutigen Software-Entwicklung müssen Tester immer mehr Software in immer kürzerer Zeit testen. Diese Herausforderung wächst, nicht nur weil Software-Systeme immer größer und komplexer werden, sondern auch, weil sich die Entwicklungsprozesse grundlegend verändert haben. Vor zehn Jahren war es weit verbreitet, dass Software in Release-Zyklen von sechs bis zwölf Monaten entwickelt wurde und vor dem Release eine ausführliche Testphase des Gesamtsystems erfolgte. Heute dagegen beobachten wir immer häufiger Feature-getriebene Releases innerhalb weniger Monate, Wochen oder sogar Tage. Um dies zu ermöglichen, erfolgt die Entwicklung auf parallelen Feature-Branches, und Testmaßnahmen müssen, entsprechend, sowohl auf jedem einzelnen dieser Branches als auch auf dem Gesamtsystem erfolgen.

Um den Test-Prozess entsprechend dieser wachsenden Anforderungen zu beschleunigen, wurde viel in Testautomatisierung und Continuous Integration (CI) investiert. Wir beobachten jedoch zunehmend, dass auch automatisierte Test-Suites mehrere Stunden oder sogar Tage laufen, insbesondere auf großen Systemen. In Konsequenz werden diese Test-Suites typischerweise aus der CI herausgenommen und nur in der Nacht, am Wochenende oder sogar noch seltener ausgeführt. Dadurch wächst die Zeitspanne zwischen einem Programmierfehler und der entsprechenden Rückmeldung, mit schwerwiegenden Folgen:

- Zwischen zwei Testläufen sammeln sich viele Code-Änderungen an, sodass es oft schwierig ist, fehlgeschlagene Tests auf eine konkrete Änderung zurückzuführen.
- Testergebnisse erreichen die Entwickler erst lange nach deren entsprechenden Änderungen, was die Identifikation von Fehlern zusätzlich erschwert.
- Die Effekte von mehreren Fehlern können in Wechselwirkung treten, sodass manche Fehler überhaupt erst entdeckt werden können, nachdem andere behoben wurden.

Diese Situation wird noch dadurch verschärft, dass Testautomatisierung nur die Hälfte des Problems angeht: Sie verbessert zwar die Effizienz der Testausführung, kann aber in keiner Weise sicherstellen, dass die ausgeführten Tests auch effektiv sind. In der Praxis haben wir beobachtet, dass selbst sehr strukturierten Test-Prozessen etwa die Hälfte der Änderungen entgehen können [2], [3]. Und wenn Tester versuchen, die Test-Suites effektiver zu machen, indem sie weitere Tests hinzufügen, verlängert sich wiederum die gesamte Testlaufzeit, und die Vorteile von Testautomatisierung und CI schwinden.

Wie können wir also aus diesem Teufelskreis ausbrechen und gleichzeitig effizient und effektiv Testen? Die Antwort ist überraschend einfach: *Wir passen unsere Testaufwände an die Code-Änderungen an.* Mit der wachsenden Zahl von zu testenden Änderungen wird es praktisch unmöglich, einfach alle Tests für jede Änderung auszuführen und manuell sicherzustellen, dass alle Änderungen adäquat getestet wurden. Anstatt nun aber auf selteneres Testen zurückzufallen, sollten wir weiterhin häufig testen, allerdings mit Fokus auf die Änderungen, also auf den Code, der neue Fehler enthalten kann. Wir nennen diese Idee *Change-Driven-Testing*. Change-Driven-Testing identifiziert 90% aller Fehler, die von der gesamten Test-Suite identifiziert werden können in nur 2% der gesamten Testlaufzeit [4] und informiert uns kontinuierlich über Code-Änderungen, die nicht getestet wurden [2], [3].

II. Test-Intelligence

In einem hochwertigen Test-Prozess stellen sich häufig Fragen wie „Welche Tests müssen ausgeführt werden?“, „Was müssen wir noch testen?“ oder „Enthält unsere Test-Suite redundante Tests?“. Da es sehr schwierig bis unmöglich ist, solche Fragen manuell korrekt zu beantworten, ist unser Ziel sie automatisiert zu beantworten, indem wir bestehende Daten aus dem Software-Entwicklungsprozess auswerten. Dieser Ansatz ist vergleichbar zur Idee von Business Intelligence, die vorhandene Daten analysiert, um etwas über Geschäftsprozesse zu lernen. Daher sprechen wir von *Test-Intelligence*. Abbildung 1 illustriert dieses Konzept.

Fragen, die wir mittels Test-Intelligence beantworten wollen, betreffen sowohl unsere Tests als auch die zu testenden Code-Änderungen. Entsprechend sammeln wir für deren Beantwortung Daten über die Tests und die Änderungen. Solche Daten liegen in typischen Entwicklungsumgebungen in der Regel bereits vor und müssen nur eingesammelt werden.

Um die gesammelten Daten an Tester, Entwickler oder auch Manager zu vermitteln, visualisieren wir sie auf *Treemaps*. Die Treemap in Abbildung 2a beispielsweise stellt den Quellcode einer Komponente aus der Benutzeroberfläche des Softwaresystems Teamscale dar. Jede Box repräsentiert dabei eine einzelne Methode aus dem Code. Die Größe der Box ist proportional zur Anzahl der Code-Zeilen in der Methode. Wir heben einzelne Boxen farblich hervor, um bestimmte Eigenschaften des entsprechenden Codes aufzuzeigen.

A. Versionskontrollsystem

Versionskontrollsysteme (VCS), wie beispielsweise `git` oder `TFS`, sind heute De-Facto-Standard in der Softwareentwicklung. Ein VCS erfasst eine chronologische Historie aller Code-Änderungen an einer Software und hilft den Entwicklern, ihre Änderungen zu koordinieren. Innerhalb des VCS

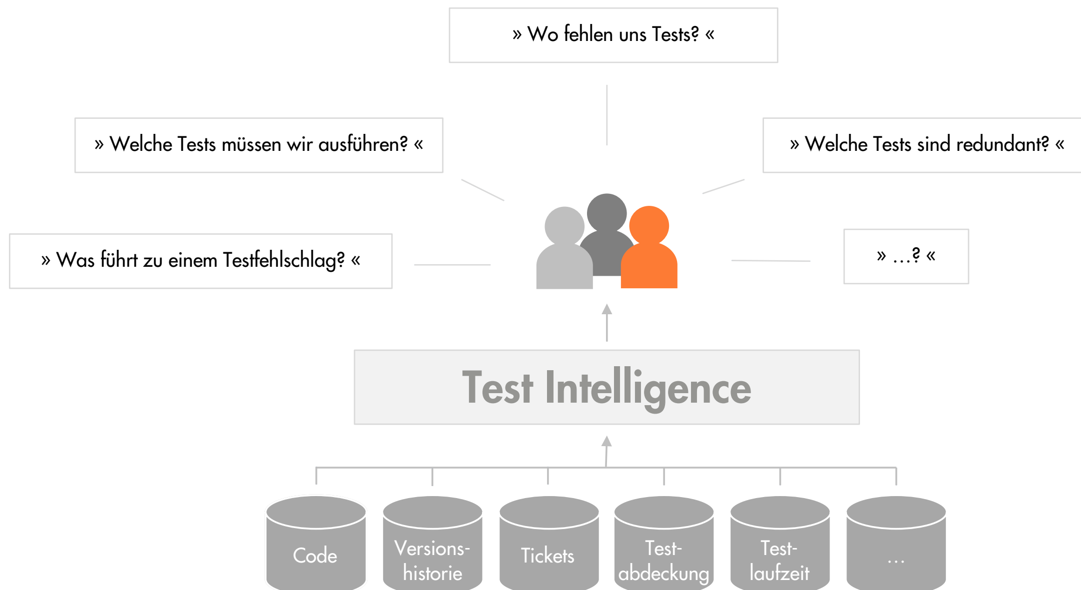


Abbildung 1. Test-Intelligence: Nutzt eine Kombination vorhandener Daten aus verschiedenen Quellen im Software-Entwicklungsprozess, um Test-Fragestellungen automatisiert zu beantworten.

können Änderungen in Branches organisiert werden, wobei ein *Branch* eine unabhängige Änderungslinie darstellt, die auf einer bestimmten Version des Codes basiert und in andere Versionen des Codes integriert werden kann. Wenn Entwickler einen separaten Branch für die Umsetzung eines einzelnen Features verwenden, sprechen wir von einem *Feature-Branch*.

Aus der Änderungshistorie in einem VCS können wir die Menge aller Änderungen seit einer Baseline bestimmen, wobei eine *Baseline* beispielsweise der Code-Stand zu einem bestimmten Release der Software, bei der letzten Testausführung, beim Start eines Feature-Branches, oder zu einem beliebigen anderen Zeitpunkt sein kann. Abbildung 2b zeigt Code-Änderungen auf einer Treemap. Methoden, die bei einer dieser Änderungen hinzukamen, sind in Rot, Methoden, die geändert wurden, in Gelb und Methoden, die unverändert blieben, in Grau hinterlegt.

B. Ticketsystem

Ticketsysteme, wie beispielsweise `Jira` oder `GitHub Issues`, werden in den meisten Software-Projekten eingesetzt, um Änderungsanfragen wie Fehlerberichte, Feature-Anfragen oder Wartungsaufgaben zu verwalten. In solchen Systemen wird jede Anfrage als ein Ticket abgebildet, das üblicherweise eine eindeutige Id, einen zuständigen Projektmitarbeiter, einen Bearbeitungsstatus und weitere Metadaten hat.

In vielen Software-Projekten annotieren Entwickler ihre Änderungen im VCS mit der Id des Tickets dem die Änderung zuzuschreiben sind. Hierzu wird meist die Id in der Änderungsbeschreibung (Commit-Nachricht) angegeben. Diese Annotation und die Metadaten aus dem Ticketsystem ermöglichen uns, alle Änderungen zu identifizieren, die bei der Umsetzung eines bestimmten Tickets vorgenommen wurden.

Abbildung 2c zeigt die Änderungen bei der Umsetzung eines einzelnen Tickets als Treemap. Methoden, die bei einer dieser Änderungen hinzukamen, sind in Rot und Methoden, die geändert wurden, in Gelb hinterlegt.

C. Profiler

Profiler, wie `Jacoco` oder `gcov`, zeichnen auf, welche Teile des Anwendungscodes ausgeführt werden, d.h. sie protokollieren *Code-Abdeckung*. Abhängig von der verwendeten Technologie kommen Profiler-Ansätze wie die Instrumentierung des Codes, die Überwachung der Ausführung in einer virtuellen Maschine oder Hardware-Profiler zum Einsatz. Technologieunabhängig ist Profiling immer möglich.

Unterschiedliche Profiler zeichnen Code-Abdeckung mit unterschiedlichem Detailgrad auf, also beispielsweise welche Methoden, Anweisungen oder Ausführungspfade ausgeführt wurden. Bei den meisten Profiler-Ansätzen bedeutet ein höherer Detailgrad automatisch, dass die Ausführung des Programms mit dem Profiler langsamer wird. Abdeckung auf Methodenlevel aufzuzeichnen ist, nach unserer Erfahrung, immer mit akzeptabler Laufzeiteinbuße möglich.

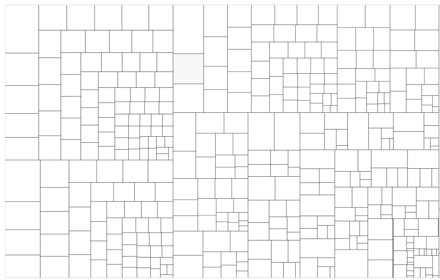
Wenn ein Profiler Code-Abdeckung während der Testausführung aufzeichnet, sprechen wir von *Testabdeckung*. Testabdeckung wird bereits häufig für automatisierte Tests in CI-Umgebungen aufgezeichnet, nur selten jedoch in anderen Testumgebungen. Vom technischen Standpunkt aus gesehen, können wir Profiler allerdings bei jeglicher Art von Ausführung einsetzen, sei es durch einen Unit-Test oder einen Ende-zu-Ende-Test, einen automatisierten Test oder einen manuellen. Daher können wir, mittels eines Profilers, Testabdeckung für jeden Test in unserer gesamten Test-Suite erfassen.

Heutige Profiler akkumulieren üblicherweise die Abdeckung von allen Tests in einem einzelnen Testlauf, da sie unabhängig von der Testausführung schlicht die Programmausführung von Start bis Ende der Laufzeit erfassen. Konzeptuell können wir allerdings auch *testweise Abdeckung* aufzeichnen, also die Testabdeckung jedes einzelnen Tests separat erfassen. Wenn wir dies tun, können wir trivialerweise auch die Laufzeit jedes einzelnen Tests erfassen.

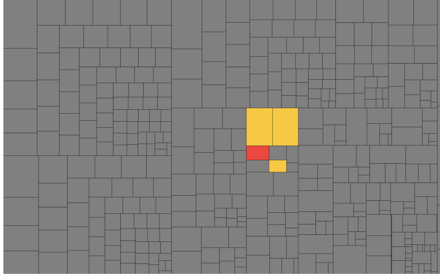
Abbildung 2d zeigt die akkumulierte Testabdeckung einer Test-Suite auf einer Treemap. Methoden, die ausgeführt wurden, sind in Grün und Methoden, die nicht ausgeführt wurden, in Grau hinterlegt.

III. Change-Driven-Testing

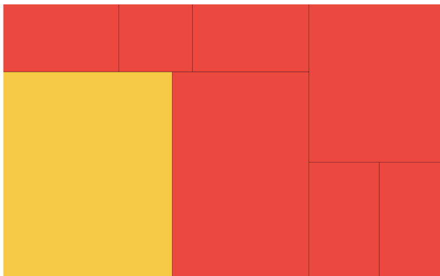
Change-Driven-Testing ist ein konkreter Anwendungsfall von Test-Intelligence, der es uns ermöglicht, unseren Test-Prozess gleichzeitig effizienter und effektiver zu gestalten.



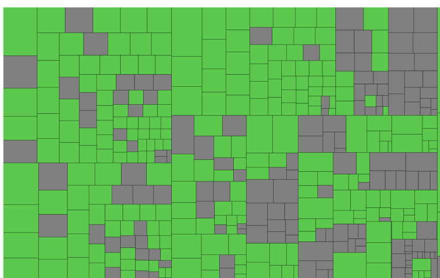
(a) Code als Treemap. Jede Box repräsentiert eine Methode. Die Größe der Box ist proportional zur Anzahl der Code-Zeilen in der Methode.



(b) Code-Änderungen seit einer Baseline. Neue Methoden werden in Rot, geänderte Methoden in Gelb und unveränderte Methoden in Grau dargestellt.



(c) Code-Änderungen für das Ticket `TS-15717`. Neue Methoden werden in Rot, geänderte Methoden in Gelb dargestellt.



(d) Testabdeckung auf Methodenlevel. Methoden, die ausgeführt wurden, werden in Grün, alle anderen Methoden in Grau dargestellt.

Abbildung 2. Treemaps, die Daten über das Softwaresystem `Teamscale` aus dem Versionskontrollsystem, dem Ticket-System und dem Profiling von Testausführung visualisieren.

Abbildung 3 veranschaulicht die Idee. Der Kerngedanke hinter Change-Driven-Testing ist, dass bestehende Tests nur fehlschlagen können, wenn sich neue Fehler eingeschlichen haben[1] und neue Fehler nur mit Änderungen hereingekommen sein können. Daher testen wir die (möglicherweise fehlerbehafteten) Änderungen zwischen zwei Versionen unseres Systems, also beispielsweise zwischen zwei aufeinanderfolgenden

[1]Wir vernachlässigen hier Gründe für sporadische Testfehlschläge, wie flackernde Tests oder Interaktionen mit Drittsystemen, die typischerweise eher auf ein Problem mit dem Test-Setup hindeuten als auf ein Problem mit dem getesteten System.

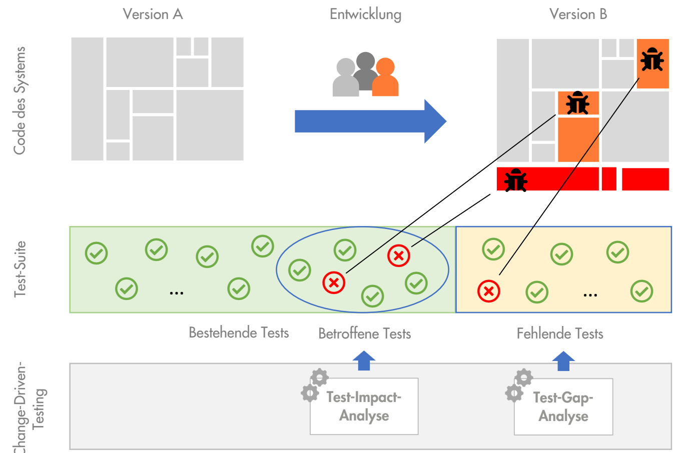


Abbildung 3. Change-Driven-Testing: Effizient testen, indem nur für die betreffenden Änderungen relevante Tests ausgeführt werden. Effektiv testen, indem alle Änderungen getestet werden.

Releases, auf folgende Weise:

1) Um *effizient* zu testen, testen wir nur die Änderungen und ignorieren alle anderen Teile des Systems. Wir automatisieren die Identifikation von bestehenden Tests, die eine gegebene Änderung betreffen, mittels einer Test-Impact-Analyse.
2) Um *effektiv* zu testen, stellen wir sicher, dass wir alle Änderungen getestet haben. Um auf dieses Ziel zuzusteuern und zu erkennen, wann wir es erreicht haben, automatisierten wir die Identifikation von ungetesteten Änderungen mittels einer Test-Gap-Analyse.

Für beide Analysen interessieren uns nur Änderungen, die das Verhalten des Codes verändern, da nur solche Änderungen Fehler enthalten können. Entsprechend nutzen wir statische Code-Analysen, um Refactoring-Änderungen, wie beispielsweise Dokumentationsanpassungen und Umbenennung oder Verschiebung von Variablen oder Methoden, zu filtern.

A. *Test-Impact-Analyse*

Abbildung 4 stellt den Ablauf der Test-Impact-Analyse (TIA) dar. Test-Impact-Analyse kombiniert die relevanten Code-Änderungen (vgl. Abschnitt II-A) mit den Daten über Testlaufzeit und testweise Abdeckung (vgl. Abschnitt II-C), um eine Teilmenge von Testfällen aus der gesamten Test-Suite zu bestimmen, die so viele Fehler wie möglich so schnell wie möglich findet. Sie geht hierbei in zwei Schritten vor:

1) Bei der *Testauswahl* werden die *betroffenen Tests*, d.h. alle Tests, die nach der aufgezeichneten Testabdeckung wenigstens eine geänderte Methode ausführen, ausgewählt. Hinzugenommen werden alle Tests, die mit den Änderungen hinzugekommen sind oder selbst verändert wurden, weil unbekannt ist welchen Teil des Systems diese Tests betreffen.
2) Bei der *Testsortierung* werden die betroffenen Tests so sortiert, dass alle Änderungen bei der sequenziellen Ausführung möglichst schnell abgedeckt werden, um Fehler so früh wie möglich zu finden. Da die Berechnung einer optimalen Reihenfolge unmöglich ist, setzen wir eine Heuristik ein: Wir wählen als nächsten Test immer den Test, der die meiste zusätzliche Änderung pro Ausführungszeit abdeckt.

Eine Studie an zwölf Software-Systemen [4] zeigt, dass die von TIA ausgewählten betroffenen Tests 99.3% aller zufällig

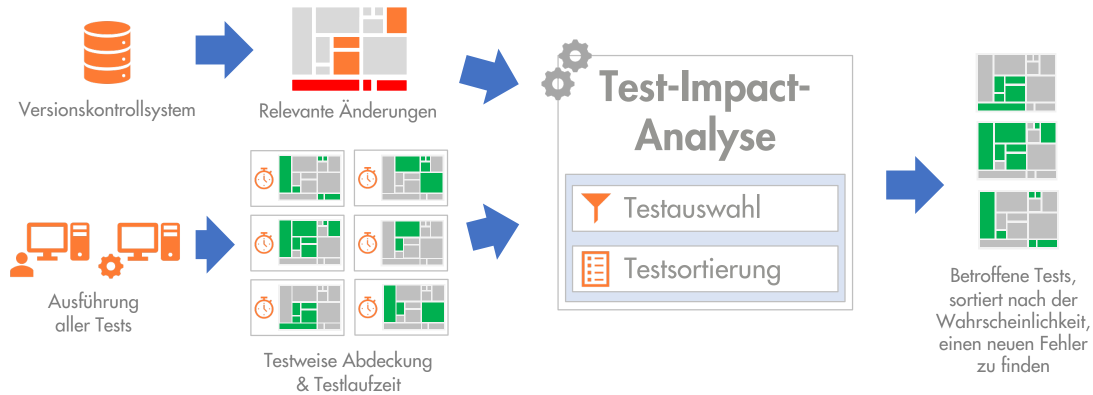


Abbildung 4. Ablauf der Test-Impact-Analyse (TIA). Für eine Menge von Änderungen wählt TIA die betroffen Testfälle aus und sortiert diese nach der Wahrscheinlich einen neuen Fehler zu finden.

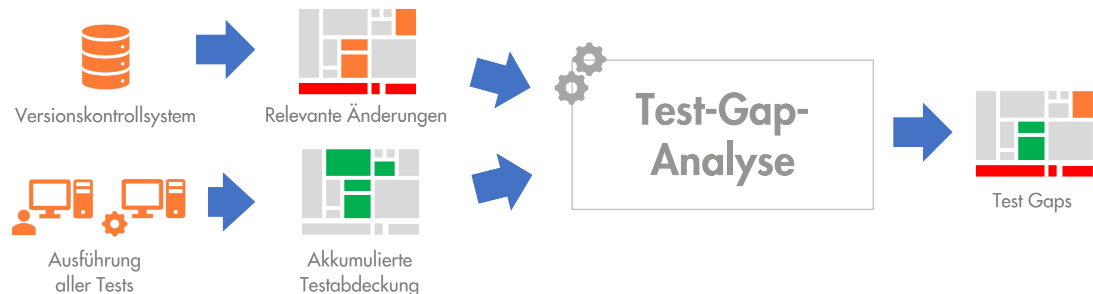


Abbildung 5. Ablauf der Test-Gap-Analyse (TGA). Für eine Menge von Änderungen identifiziert TGA diejenigen Änderungen, die noch nicht getestet wurden, also die Test-Gaps.

eingefügten Programmierfehler finden, die von der gesamten Test-Suite gefunden werden. In allen diesen Software-Systemen fanden die betroffenen Tests mehr als 90%, in sieben Systemen sogar 100% dieser Fehler.

Eine zweite Studie an über 100 verschiedenen Systemen [5] zeigt, dass mit TIA im Durchschnitt über 90% aller Fehler, die von der gesamten Test-Suite gefunden werden können, in nur 2% der Ausführungszeit der gesamten Suite identifiziert werden. Der Einsatz von TIA ist besonders vorteilhaft, wenn die zu testenden Änderungen klein und lokal sind, weil dann eine entsprechend kleine Menge von Tests gewählt wird. Dies passt gut zu unserer Anforderung, jede eintreffende Änderung schnell zu testen, da einzelne Änderungen im Vergleich zum Gesamtsystem meistens klein sind.

B. Test-Gap-Analyse

Abbildung 5 stellt den Ablauf der Test-Gap-Analyse (TGA) dar. Test-Gap-Analyse kombiniert die relevanten Code-Änderungen (vgl. Abschnitt II-A) mit der aggregierten Testabdeckung (vgl. Abschnitt II-C), um diejenigen Änderungen zu identifizieren, die von keinem Test abgedeckt wurden. Wir nennen diese Änderungen *Test-Gaps*.

Bei der Identifikation von Test-Gaps beachtet die TGA die chronologische Reihenfolge von Code-Änderungen und Testausführung: Neue Änderungen invalidieren vorhergegangene Testabdeckung von Änderungen und öffnen neue Test-Gaps. In nachfolgenden Tests wird neue Testabdeckung aufgezeichnet, die entsprechende Test-Gaps aus vorangegangenen Änderungen schließt. Auf diese Weise kann TGA nach jeder Testausführung und jeder Code-Änderung eine Übersicht über die jeweils aktuellen Test-Gaps liefern. Und da die Analyse inkrementell erfolgt, kann sie die aktuelle Übersicht in wenigen Sekunden liefern, auch für sehr große Systeme.

In einer Fallstudie an einem großen industriellen Software-System [2] zeigte TGA, dass jeweils über 55% der Code-Änderungen in zwei aufeinanderfolgenden Releases ungetestet blieben. Trotz eines sehr strukturierten Testprozesses. Im Nachgang konnten über 70% aller gemeldeten Feldfehler auf solche ungetesteten Änderungen zurückgeführt werden.

In einer zweite Fallstudie an einem anderen industriellen Software-System [6] zeigte TGA über 110 Test-Gaps in den Änderungen aus 54 Tickets, was 21% der 511 geänderten Methoden entspricht. Für 35% dieser Test-Gaps befanden die Entwickler, dass Tests hätten geschrieben werden sollen. Darüber hinaus bemerkten sie, dass 49% der Test-Gaps durch Aufräumarbeiten im Code entstanden waren, die sich nicht in der Ticketbeschreibung widerspiegelten und daher ungetestet blieben, obwohl die im Ticket beschriebenen Änderungen gründlich getestet wurden.

C. Einschränkungen

Um die Analysen bestmöglich einzusetzen, ist es wichtig, ihre Grenzen zu kennen. Sowohl TIA also auch TGA können Änderungen an Konfiguration oder Daten nicht berücksichtigen, da diese nicht im Code sichtbar werden. In solchen Fällen kann TIA die betroffenen Tests nicht bestimmen, und TGA den betroffenen Code nicht als ungetestet aufzeigen.

Eine ähnliche Einschränkung besteht für TIA bei indirekten Methodenaufrufen. Wenn beispielsweise ein Tests alle Klassen mit einer bestimmten Annotation ausführt, wählt TIA diesen Test nicht, wenn die Annotation zu einer weiteren Klasse hinzugefügt wird, weil die zuvor aufgezeichnete Abdeckung des Tests die neu annotierte Klasse nicht beinhaltet und sich der Test selbst im Code nicht verändert hat.

TIA geht davon aus, dass die Abdeckung von Tests stabil ist, d.h., dass aus mehrfacher Ausführung desselben Tests immer

Abbildung 6. Ticket `TS-15717`: Löschen von Zugangsdaten ermöglichen.

dieselbe Abdeckung resultiert. Wenn das nicht der Fall ist, beispielsweise weil manuelle Tests jedesmal leicht unterschiedlich durchgeführt werden, kann TIA die Auswirkungen eines Tests nicht präzise erfassen und, entsprechend, die betroffenen Tests nicht immer richtig wählen. In der Praxis kann sich auch die Abdeckung von automatisierten Tests zwischen mehreren Testläufen unterscheiden, beispielsweise aufgrund von Garbage Collection. Obwohl solche Abweichungen in der Regel klein sind, kann TIA nicht garantieren, dass die betroffenen Tests immer alle Fehler finden, die von der gesamten Test-Suite gefunden werden können. Soweit wir wissen, gibt es keine Testauswahlstrategie, die unter diesen Bedingungen eine solche Garantie geben kann. Daher empfehlen wir, zusätzlich zur Testausführung mit TIA, die gesamte Test-Suite weiterhin regelmäßig auszuführen. Dabei werden dann gegebenenfalls übersehene Fehler gefunden und gleichzeitig die Daten bzgl. Testabdeckung und Testausführungszeit aktualisiert. Diese Empfehlung entspricht dem traditionellen Ansatz Test-Suites regelmäßig (z.B. jede Nacht) auszuführen. Allerdings haben wir nun zusätzlich schnelle Rückmeldung aus der direkten Testausführung mittels TIA bekommen, von der bereits die Mehrzahl der Fehler abgefangen wird.

Eine weitere Einschränkung der TGA ist, dass sie nicht berücksichtigt, wie gründlich eine Änderung getestet wurde, sondern nur, ob eine Methode, die geändert wurde, überhaupt in einem Test ausgeführt wurde. Wie jeder andere Testansatz kann TGA nicht die Abwesenheit von Fehlern beweisen. Daher konzentriert sich die Analyse darauf, diejenigen Änderungen zu identifizieren, die definitiv noch gar nicht getestet wurden, in denen also garantiert noch keine Fehler gefunden wurden. Die Wahrscheinlichkeit, dass sich Fehler in solchen ungetesteten Änderungen verbergen, ist fünfmal höher als in getesteten Änderungen [3]. Nach unserer Erfahrung identifiziert TGA substanzielle Test-Gaps und ermöglicht deutliche Verbesserungen des Testprozesses.

IV. Change-Driven-Testing im Einsatz

Um den Einsatz von Change-Driven-Testing zu veranschaulichen, betrachten wir die Entwicklung eines Features in unserem eigenen Software-System `Teamscale`. `Teamscale` ist eine Plattform, die Software-Entwicklungsteams bei der Analyse, der Überwachung und der Optimierung von Code- und Testqualität unterstützt. Zu diesem Zweck verbindet es sich mit verschiedenen anderen Entwicklungssystemen, wie dem Versionskontrollsystem, dem Build-Server oder dem Ticketsystem. In `Teamscales` Nutzeroberfläche gibt es eine entsprechende Ansicht, um die Zugangsdaten zu solchen externen Systemen zu verwalten. Abbildung 6 zeigt Ticket `TS-15717`, mit dem die Möglichkeit hinzugefügt werden soll, solche Zugangsdaten aus `Teamscale` zu löschen.

Bevor unsere Entwicklerin mit der Arbeit an `TS-15717` beginnt, legt sie zunächst einen Feature-Branch im Versionskontrollsystem an. Dann implementiert sie die entsprechende Funktionalität, schreibt einen Test und stellt die Änderungen in das Versionskontrollsystem ein, mit der Beschreibung:

`TS-15717`: Allow external-credentials deletion.

Die Treemap in Abbildung 2c zeigt diese Änderungen.

A. *Testen mit TIA*

Nach dem Einstellen der Änderungen in das Versionskontrollsystem kompiliert und testet unsere CI-Umgebung die Änderungen automatisch. Vor der Testausführung fragt sie dazu die TIA nach den betroffenen Tests. Um diese Anfrage zu beantworten, verwendet TIA die testweise Testabdeckung und -laufzeit von allen etwa 6.500 Tests in unserer Test-Suite (Unit-Tests, Integrationstests, Systemtests und Oberflächentests). Um diese Daten zu erfassen, haben wir die Testausführung in unserer CI-Umgebung mit Profilern instrumentiert, sodass die testweise Abdeckung und Laufzeit sowohl für den Javascript-Code aus `Teamscales` Web-Oberfläche, als auch für den serverseitigen Java-Code aufgezeichnet wird. Anschließend haben wir die gesamte Test-Suite einmal ausgeführt, was etwa 45 Minuten dauerte. Anhand dieser daraus resultierenden Datenbasis berechnet TIA nun eine Liste von sechs betroffenen Tests für die Änderungen unserer Entwicklerin: fünf Regressionstests, die Teile des geänderten Codes abdecken und der neue Test aus ihren Änderungen. Der gesamte CI-Lauf mit diesen betroffenen Tests dauert nur etwa 1,5 Minuten, womit wir über 96% der Testausführungszeit gespart haben.

Dank TIA wird unsere Entwicklerin nur 1,5 Minuten nach dem Einstellen ihrer Änderungen in das Versionskontrollsystem benachrichtigt, dass ihr neuer Test (den die Testsortierung an zweiter Stelle einreihte) fehlschlägt. Die Änderungen noch frisch im Gedächtnis, behebt sie das Problem in zehn Minuten. Anschließend stellt sie die neuen Änderungen wiederum in das Versionskontrollsystem ein, mit der Beschreibung:

`TS-15717`: Fix deletion of external credentials.

Da die Korrektur sehr lokal war, wählt TIA nur einen einzelnen betroffenen Test, namentlich den Test, der zuvor

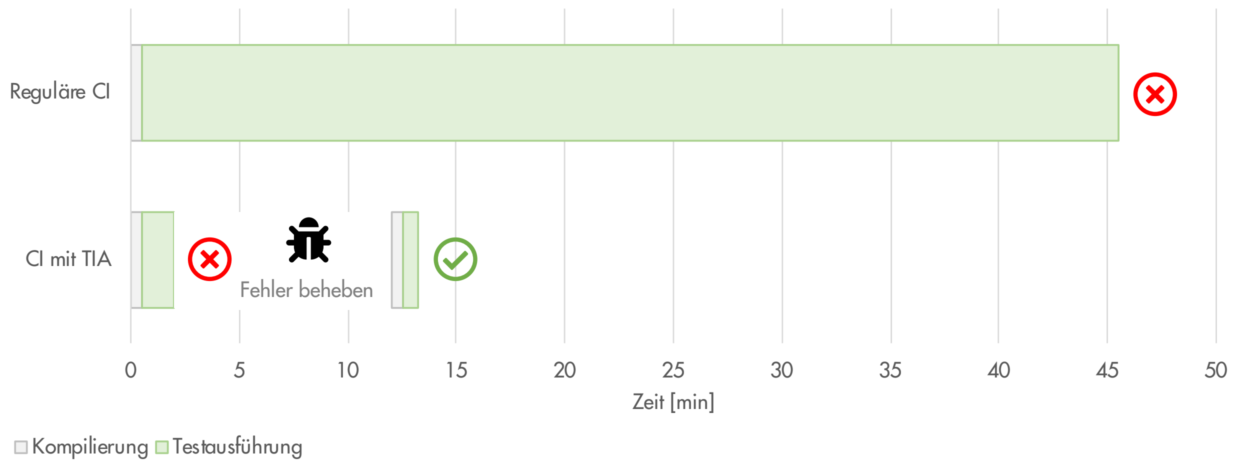


Abbildung 7. Zeitersparnis durch den Einsatz von Test-Impact Analysis: Zwei aufeinanderfolgende CI-Läufe mit TIA und das Beheben eines Fehlers benötigen weniger Zeit als das einmalige Ausführen der gesamten Test-Suite von `Teamscale`.

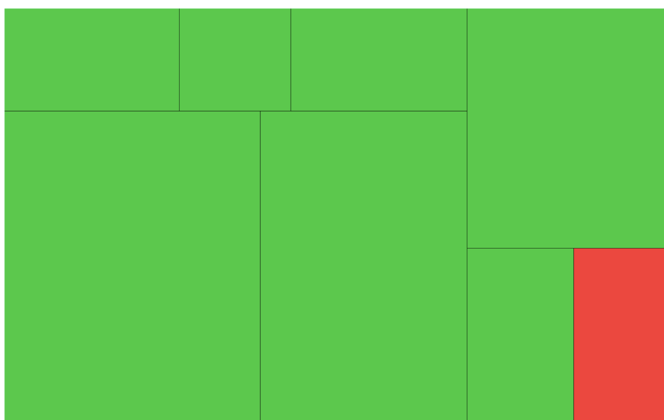


Abbildung 8. Issue-Test-Gaps für `TS-15717` nach der Ausführung der betroffenen Tests aus der TIA. TGA zeigt ein verbleibendes Test-Gap (rote Box rechts unten).

fehlschlug. Entsprechend dauert der CI-Lauf diesmal nur insgesamt 45 Sekunden. Alle Tests laufen erfolgreich.

Insgesamt haben die zwei aufeinanderfolgenden CI-Läufe und das Beheben des Fehlers weniger Zeit in Anspruch genommen als eine einzelne Ausführung unserer vollständigen Test-Suite. Abbildung 7 veranschaulicht diese Zeitersparnis.

B. *Testen mit TGA*

Um sicherzustellen, dass alle ihre Änderungen getestet sind, sieht sich unsere Entwicklerin die Test-Gap-Treemap für ihre Änderungen im Kontext des Tickets `TS-15717` an. Die Treemap in Abbildung 8 zeigt die aggregierte Testabdeckung aus den zwei vorhergegangen CI-Läufen. Wir sprechen auch von den *Issue-Test-Gaps* für `TS-15717`: Die meisten geänderten Methoden wurden getestet (grüne Boxen), aber ein Test-Gap verbleibt (rote Box unten rechts).

Um zu entscheiden, ob das verbleibende Test-Gap geschlossen werden sollte, sieht sich unsere Entwicklerin den betreffenden Code an, zu dem sie mit einem Klick auf die Treemap gelangt. Sie entdeckt, dass der ungetestete Code eine zusätzliche Sicherheitsabfrage durchführt, wenn ein Nutzer versucht Zugangsdaten zu löschen, die von `Teamscale` noch verwendet werden. Da dieser Code recht einfach ist und zukünftige Änderungen unwahrscheinlich sind, entscheidet sich unsere Entwicklerin einen einmaligen manuellen Test durchzuführen. Sie startet ihre Entwicklungsversion von `Teamscale` lokal auf ihrer Maschine, über ein Startskript für manuelle Tests, dass wir mit unserem Code verwalten. Sie öffnet das Testsystem in ihrem Browser, navigiert zur entsprechenden Verwaltungsansicht und prüft ob die Sicherheitsabfragen erwartungsgemäß erscheinen und berücksichtigt werden. Dies ist der Fall. Daher fährt sie das Testsystem wieder herunter, wobei das Startskript die soeben aufgezeichnete Testabdeckung für TGA zur Verfügung stellt. Alle Änderungen für `TS-15717` wurden nun getestet, was nachvollziehbar dokumentiert ist durch eine vollständig grüne Test-Gap-Treemap.

Da sich unsere Entwicklerin für einen explorativen manuellen Test entschieden hat, existiert kein Regressionstest für die zusätzliche Sicherheitsafrage. Da TGA aber die chronologische Reihenfolge von Testabdeckung und Code-Änderungen berücksichtigt, würde erneut ein Test-Gap aufgezeigt, sollte sich die entsprechende Funktionalität nochmals ändern. Dank dieses Sicherheitsnetzes ist es absolut angemessen, einen schnellen manuellen Test der Entwicklung eines automatisierten Oberflächentests oder der Erstellung eines formalen manuellen Tests vorzuziehen, wenn Änderungen am betreffenden Code unwahrscheinlich sind.

C. *Änderungen integrieren*

Zu diesem Zeitpunkt ist unsere Entwicklerin mit ihren Änderungen zufrieden und sendet sie an einen Kollegen zum Review. Sobald sie und der Reviewer sich einig sind, schließt dieser den Feature-Branch und integriert die Änderungen. In Reaktion auf diese Aktion im Versionkontrollsystem führt unsere CI-Umgebung nun unsere gesamte Test-Suite aus. Dadurch wird sichergestellt, dass `Teamscale` fehlerfrei bleibt, auch wenn TIA relevante Tests ausgeschlossen haben sollte. Außerdem wird hierbei der Datenbestand über Testabdeckung und -ausführungszeit aktualisiert. Da die Integration von Feature-Branches deutlich seltener erfolgt als das Einstellen neuer Änderungen auf Feature-Branches, profitieren trotzdem die meisten CI-Läufe von den kürzeren Laufzeiten durch TIA.

Vor jedem `Teamscale`-Release (aktuell alle sechs Wochen) sieht sich ein Testarchitekt systemweit alle verbliebenen Test-Gaps aus der Entwicklung seit dem letzten Release an. Dies dient als zusätzliches Quality-Gate um sicherzustellen, dass keine kritische Funktionalität aus Versehen ungetestet bleibt. Für diese Betrachtung verwendet der Testarchitekt die selben Daten, die bereits während der Entwicklung für TGA eingesetzt wurden, betrachtet sie aber auf einer Treemap, die das

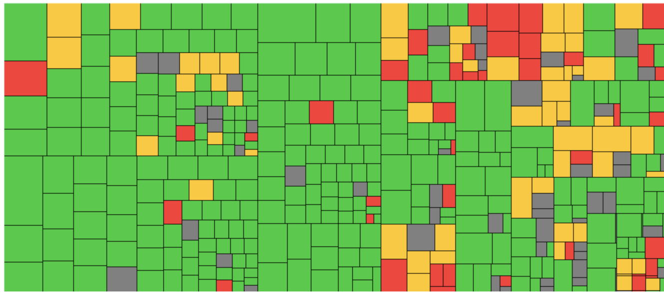


Abbildung 9. Test-Gap-Treemap für eine Komponente von `Teamscale`. Eine systemweite Analyse von verbliebenen Test-Gaps dient als zusätzliches Quality-Gate vor jedem Release.

gesamte System darstellt, anstatt nur Änderungen an einem einzelnen Feature. Abbildung 9 zeigt einen Ausschnitt aus dieser systemweiten Test-Gap-Treemap, die eine Komponente von `Teamscales` Web-Oberfläche darstellt.

V. Change-Driven-Testing anpassen

In der Praxis treffen wir auf sehr unterschiedliche Testumgebungen und -strategien, abhängig von den konkreten Anforderungen und der entsprechenden Projekthistorie. Daher muss die konkrete Umsetzung von Change-Driven-Testing oft angepasst werden, um den meisten Mehrwert zu bieten. Nachfolgend diskutieren wir einige Aspekte, die uns immer wieder begegnen, ohne Anspruch auf Vollständigkeit.

- Test-Impact-Analyse und Test-Gap-Analyse können Testabdeckung sowohl aus allen Teststufen als auch von manuellen und automatisierten Tests berücksichtigen. Die Testart macht keinen Unterschied für die Vorteile von Test-Gap-Analyse. Die Vorteile von Test-Impact-Analyse, andererseits, sind am größten, wenn die Tests langsam sind, weil dann das Ausschließen von Tests viel Zeit spart, oder wenn die Testzeit begrenzt ist, weil dann die Testsortierung die Wahrscheinlichkeit erhöht, trotzdem viele Fehler zu finden, auch wenn wir nicht alle betroffenen Tests ausführen können.
- Wenn die Entwicklung hauptsächlich auf Feature-Branches erfolgt, ist es sinnvoll, Test-Impact-Analyse zum Ausführen der Tests und Issue-Test-Gaps zur Vermeidung von Test-Gaps auf diesem Branch zu nutzen. Zusätzlich sollte die gesamte Test-Suite beim Integrieren von Feature-Branches ausgeführt werden um sicherzustellen, dass keine Fehler durchrutschen. Wenn die Entwicklung allerdings auf dem Haupt-Branch erfolgt, können wir Test-Impact-Analyse zum Testen von individuellen Änderungen einsetzen und die gesamte Test-Suite periodisch oder vor jedem Release ausführen.
- Es ist immer möglich, Test-Impact-Analyse mit anderen Testauswahlverfahren zu kombinieren. Beispielsweise wenn Tests für Kernfunktionalität immer ausgeführt werden sollen. Es werden dann sowohl die von Test-Impact-Analyse identifizierten Tests als auch die Tests aus der komplementären Auswahlstrategie ausgeführt.
- In unserer Erfahrung ist die Analyse von Issue-Test-Gaps am effizientesten, weil das Ticket einen Kontext gibt, um die (ungetesteten) Änderungen zu interpretieren. TGA kann allerdings auch dann Test-Gaps aufzeigen, wenn Code-Änderungen nicht mit Tickets verknüpft werden können. Eine solche systemweite TGA stellt für sich bereits einen Mehrwert dar, aber auch zusätzlich zu Issue-Test-Gaps als zweites Quality-Gate.
- In vielen Projekten werden die Test-Gaps nicht von den Entwicklern selbst analysiert, sondern von Testern oder Architekten. Derartige Arbeitsteilung erschwert manchmal die Interpretation von Test-Gaps, wenn dem Analyst Hintergrundinformationen fehlen. In solchen Fällen erweisen sich die Daten aus dem Versionskontrollsystem erneut als hilfreich, weil sie den für eine Änderung verantwortlichen Entwickler benennen, was dem Analysten erlaubt, diesen direkt zu kontaktieren.
- Eine vollständig grüne Test-Gap-Treemap ist zwar das theoretische Ideal, sollte aber niemals Selbstzweck sein. Oft können Test-Gaps vertretbar oder sogar vernünftig sein, beispielsweise wenn es sich um Code für ein zukünftiges Feature handelt, der noch nicht freigeschaltet ist, oder um Code, der nur selten und intern benutzt wird, sodass Testressourcen anderswo gewinnbringender eingesetzt werden können. Letztendlich müssen wir im Testprozess immer abwägen und priorisieren. TGA ermöglicht es uns hierbei faktenbasierte Entscheidungen.

VI. Zusammenfassung

In der heutigen Software-Entwicklung müssen Tester immer mehr Software in immer kürzerer Zeit testen. Daher ist es unmöglich, einfach die gesamte Test-Suite für jede Änderung der Software auszuführen. Und es ist ebenso nicht länger möglich (falls es das jemals war) manuell sicherzustellen, dass die durchgeführten Tests alle Änderungen abdecken. Daher müssen wir unsere Teststrategien überdenken und sowohl effizienter als auch effektiver werden.

In diesem Kapitel haben wir Change-Driven-Testing vorgestellt. In Change-Driven-Testing analysieren wir bestehende Daten aus dem Software-Entwicklungsprozess, um automatisiert Antworten auf Fragen zu liefern, die unseren Testprozess steuern. Wir nutzen Test-Impact-Analyse, um automatisiert die betroffenen Tests für eine gegebene Code-Änderung zu identifizieren und diese Tests so zu sortieren, dass wir Fehler möglichst früh identifizieren. Dies macht Testen effizienter, sodass wir über 90% der Fehler in nur 2% der Testausführungszeit abfangen können. Wir nutzen Test-Gap-Analyse, um automatisiert Test-Gaps, d.h. Lücken in unserer Testabdeckung, aufzudecken. Dies ermöglicht uns faktenbasierte Entscheidungen darüber, wo wir unsere begrenzten Testresources am besten einsetzen, um unser Testen effektiver zu machen.

VII. Kenntlichmachung

Dieser Text ist eine Übersetzung von „Change-Driven Testing“, ursprünglich veröffentlicht in „The Future of Software Quality Assurance“ [1], durch Springer Open, unter der Creative Commons Attribution 4.0 International License.

Literatur

[1] Amann, S., Jürgens, E.: Change-Driven Testing. Springer Open (2019)

[2] Eder, S., Hauptmann, B., Junker, M., Juergens, E., Vaas, R., Prommer, K.H.: 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] Juergens, E., Pagano, D.: Did We Test the Right Thing? Experiences with Test Gap Analysis in Practice. Whitepaper, CQSE GmbH (2016)

[4] Juergens, E., Pagano, D., Goeb, A.: Test Impact Analysis: Detecting Errors Early Despite Large, Long-Running Test Suites. Whitepaper, CQSE GmbH (2018)

[5] Rott, J.: Empirische Untersuchung der Effektivität von Testpriorisierungsverfahren in der Praxis. Master's thesis, Technische Universität München (2019)

[6] Rott, J., Niedermayr, R., Juergens, E., Pagano, D.: Ticket coverage: Putting test coverage into context. In: Proceedings of the 8th Workshop on Emerging Trends in Software Metrics (WETSoM'17) (2017)