Test-Impact-Analyse: Trotz großer, langlaufender Test-Suites Fehler früh erkennen

Elmar Jürgens
CQSE GmbH
juergens@cqse.eu

Dennis Pagano
CQSE GmbH
pagano@cqse.eu

***Zusammenfassung*—Große Test-Suites haben oft eine lange Laufzeit. Daher werden sie in der Praxis meistens nicht als Teil der Continuous Integration (CI) ausgeführt, sondern erst in späteren Testphasen. Dadurch bleiben viele Fehler während der CI unerkannt und werden erst spät gefunden, was hohen Aufwand verursacht.**

Test-Impact-Analyse erlaubt es, nur die Tests auszuführen, die von den Code-Änderungen seit dem letzten Testlauf betroffen sind. Dadurch kann im Rahmen der CI immer der Ausschnitt einer großen Test-Suite ausgeführt werden, der am wahrscheinlichsten neue Fehler findet. In unseren Studien konnten wir so in 2% der Testausführungszeit 90% der fehlerhaften Builds identifizieren. Dadurch werden schnelle CI-Zeiten mit hoher Fehlererkennungsrate unabhängig von der Größe und Laufzeit der gesamten Test-Suite möglich.

I. Motivation

Wenn Software wächst, muss auch ihre Test-Suite wachsen, wenn Fehler zuverlässig gefunden werden sollen. Damit steigt aber auch die Ausführungsdauer aller Tests. Wir sehen in der Praxis immer öfter Test-Suites, die mehrere Stunden oder sogar Tage laufen.

Nach unserer Erfahrung werden aber langlaufende Suites typischerweise aus der Continuous Integration herausgenommen. Oft werden sie dann nur noch in nachgelagerten Testphasen ausgeführt, die bspw. vor jedem Release stattfinden. Durch die seltenere Ausführung der Test-Suite verlängert sich jedoch der Zeitraum zwischen Fehlerentstehung und -aufdeckung. Das hat umfassende negative Konsequenzen:

- Fehler können sich gegenseitig überlagern und so erst erkannt werden, wenn andere Fehler behoben sind
- Debugging von Regressionsfehlern wird aufwändiger, da eine immer größere Anzahl von Code-Änderungen den Fehler verursacht haben kann
- Fehlerkorrekturen dauern länger, da sich Entwickler in ihren eigenen Code neu einarbeiten müssen, wenn ein Fehler erst Wochen nach seinem Einbau aufgedeckt wird

Diese Effekte nehmen zu, je stärker System und Test-Suite wachsen. Ironischerweise gefährdet das die Effektivität von Continuous Integration gerade bei großen Systemen, die oft einen besonders hohen strategischen Wert haben und bei denen kurze Feedback-Zyklen daher besonders wichtig wären. Wie können wir das verhindern und trotz steigender Ausführungsdauer unserer Test-Suites in der Continuous Integration schnell neue Fehler finden?

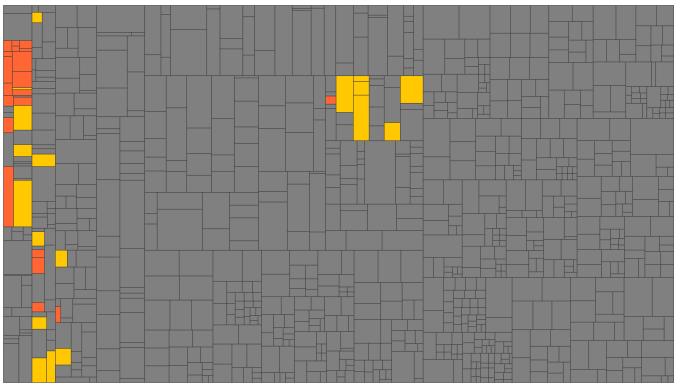

Abbildung 1. Änderungen durch Implementierung eines Features: Neuer Code ist rot, geänderter Code gelb dargestellt. Grauer Code ist unverändert.

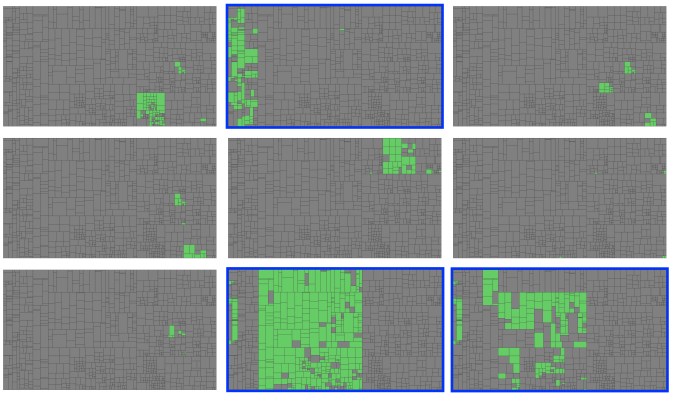

Abbildung 2. Test-Coverage von neun automatisierten Tests. Wenn ein Test eine Methode ausführt ist sie grün dargestellt, sonst grau.

Unser Ansatz für Test-Impact-Analyse (TIA) adressiert dieses Problem, indem Testfälle auf Basis von Code-Änderungen selektiert und priorisiert werden. Dazu werden die Code-Änderungen analysiert, die seit dem letzten erfolgreichen Testlauf durchgeführt worden sind. Dann werden nur die Testfälle ausgewählt, die auch geänderten Code durchlaufen. Diese Tests werden dann so sortiert, dass sie Fehler möglichst früh im Testlauf finden. Je häufiger TIA durchgeführt wird, desto weniger Änderungen liegen zwischen zwei Testläufen, und desto größer ist die zu erwartende Laufzeitersparnis.

Abbildungen 1 und 2 zeigen ein Beispiel aus unserer eigenen Entwicklung. Jede der Abbildungen zeigt eine Treemap.

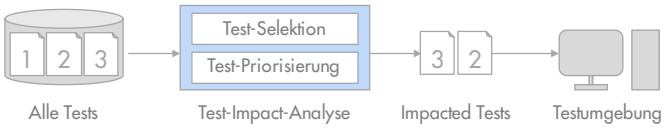

Abbildung 3. Phasen der Test-Impact-Analyse

Jedes kleine Rechteck in der Treemap repräsentiert dabei eine Methode im Quelltext des *System under Test*. In Abbildung 1 sind die Änderungen dargestellt, die im Rahmen der Entwicklung eines Features durchgeführt wurden. Unveränderter Code ist grau, angepasster orange und neuer Code rot dargestellt. Abbildung 2 zeigt die Code-Coverage von 9 automatisierten Tests. Wenn ein Testfall eine Methode ausführt ist sie grün dargestellt. Graue Methoden wurden vom Testfall nicht ausgeführt. Im Beispiel führen nur die drei Testfälle, deren Treemaps blau umrandet sind, überhaupt geänderten Code aus. Die verbleibenden sechs Testfälle können keinen Fehler finden, der durch die Implementierung des neuen Features entstanden ist. Ihre Laufzeit kann daher im Rahmen der CI eingespart werden.

Im Artikel stellen wir TIA vor, skizzieren verwandte Arbeiten und beschreiben dann unseren Ansatz und die empirischen Studien, die wir durchgeführt haben, um die Anwendbarkeit und den Nutzen von TIA in der Praxis zu quantifizieren. Für die untersuchten Studienobjekte konnten wir zeigen, dass TIA die Testzeiten im Rahmen der Continuous Integration maßgeblich verringern kann: Für unsere Studienobjekte kann die Testzeit um 98% reduziert werden, trotzdem werden noch 90% aller fehlerhaften Builds als solche erkannt. Der zentrale Beitrag dieses Artikels ist daher zu zeigen, dass das vorgeschlagene TIA-Verfahren (das zum großen Teil auf der Literatur aufbaut) unter den in diesem Artikel beschriebenen Bedingungen reif für den kontinuierlichen Einsatz in der Praxis ist.

II. Test-Impact-Analyse

Das Ziel von Test-Impact-Analyse ist es, für den aktuellen Code-Stand des *Systems under Test* eine Teilmenge der gesamten Test-Suite vorzuschlagen, mit der man möglichst schnell möglichst viele Fehler finden kann. Wie in Abbildung 3 dargestellt, setzt sich die Analyse dabei prinzipiell aus zwei Schritten zusammen:

- *Test-Selektion*: Auf Basis bestimmter Metriken wird eine Teilmenge der gesamten Test-Suite ausgewählt, die ausgeführt werden soll. Diese ausgewählte Teilmenge bezeichnen wir als »*impacted Tests*«.
- *Test-Priorisierung*: Die impacted Tests werden sortiert, so dass möglichst schnell Fehler gefunden werden. Hierbei können Daten aus vorhergehenden Testläufen zugrunde gelegt werden.

TIA kann sowohl nur mit Test-Selektion oder nur mit Test-Priorisierung als auch mit einer Kombination von beiden Schritten durchgeführt werden. Die konkrete algorithmische Umsetzung dieser beiden Schritte ist dabei das primäre Unterscheidungsmerkmal verschiedener Ansätze. Die in der Literatur beschriebenen Ansätze für Test-Selektion und Test-Priorisierung variieren oft stark hinsichtlich ihrer Komplexität und Effektivität. Beispielsweise gibt es randomisierte Ansätze oder solche, die auf Basis der testfallspezifischen Coverage selektieren oder priorisieren. Wir haben die wichtigsten Ansätze im Abschnitt IV zusammengefasst. Der von uns gewählte Ansatz verwendet eine Kombination aus Selektion und Priorisierung. Wir beschreiben diesen Ansatz genauer in Abschnitt V. Die resultierenden selektierten und sortierten Tests werden abschließend in der Testumgebung in der definierten Reihenfolge ausgeführt.

Da bei Verwendung von TIA nur ein Teil der gesamten Test-Suite ausgeführt wird, ist es in der Praxis notwendig, in regelmäßigen Abständen trotzdem alle Tests auszuführen. Wir beschreiben die Grenzen von TIA im folgenden Abschnitt.

III. Grenzen von Test-Impact-Analyse

Wie jedes Analyseverfahren hat auch TIA Grenzen. Ihre Kenntnis ist entscheidend, um sie sinnvoll einsetzen zu können.

Eine Grenze von TIA sind Änderungen, die auf Konfigurationsebene durchgeführt werden, ohne dass dabei Code verändert wird, da sie dadurch der Analyse verborgen bleiben.

Einige Ansätze führen Selektion oder Priorisierung auf Basis der Coverage durch, die in einem vergangenen Lauf aufgezeichnet wurde. Eine Einschränkung dieser Ansätze ist die Annahme einer gewissen Stabilität von Test-Coverage, da Änderungen am Code oder an Tests prinzipiell dazu führen können, dass im nächsten Lauf nicht mehr die aufgezeichneten Methoden durchlaufen werden. Gleiches gilt, wenn Tests nichtdeterministisch sind.

Hängen Selektions- oder Priorisierungsalgorithmus von der Test-Coverage ab, so muss diese außerdem testfallspezifisch aufgezeichnet werden. Hierfür ist es typischerweise erforderlich, die Tests isoliert voneinander auszuführen.

Eine weitere Grenze von TIA ist die Verwendung indirekter Code-Aufrufe. Wird bspw. in einem Testfall Reflection verwendet, um Klassen mit bestimmten Annotationen zu testen, so wird ein solcher Testfall typischerweise nicht selektiert, wenn eine weitere Klasse entsprechend annotiert wird.

Obwohl es in der Theorie Techniken gibt, eine Teilmenge der Tests zu bestimmen, die nachweislich alle Fehler finden (die die gesamte Test-Suite findet), sind diese in der Praxis oft nicht einsetzbar oder bringen keinen echten Vorteil. Praxistaugliche TIA kann hierfür also typischerweise keine Garantien geben. Daher müssen in regelmäßigen Abständen alle Tests ausgeführt werden. Die Verwendung von TIA führt jedoch dazu, dass Fehler im Regelfall deutlich früher gefunden werden als wenn ausschließlich in eigens anberaumten Testphasen getestet wird, was in der Praxis zumeist mit einer deutlichen Reduzierung der Kosten einhergeht.

IV. Existierende Arbeiten zu Test-Impact-Analyse

Es gibt in der Literatur einige Ansätze zu TIA, Test-Selektion und Test-Priorisierung. Wir stellen verschiedene

Gruppen von Ansätzen vor und ordnen unseren Ansatz ein. Da der Beitrag dieses Artikels primär die Untersuchung der Eignung von TIA in der Praxis ist, verzichten wir aus Platzgründen auf eine Abgrenzung der verschiedenen Ansätze im Detail.

A. Test-Selektion[1]

Engström et al. [2] teilen in einer Metastudie existierende Arbeiten zu Test-Selektion in verschiedene Kategorien auf. Wir folgen dieser Aufteilung und führen im Folgenden repräsentative existierende Arbeiten auf, mit denen TIA automatisiert durchgeführt werden kann.

1) DejaVu-basierte Selektion: DejaVu-basierte Techniken [7] leiten aus feingranularen Daten über die Testabdeckung ab, welche Tests geänderten Code durchlaufen. Diese Tests werden anschließend selektiert. Die zugrundeliegende Annahme hierbei ist, dass Fehler wahrscheinlich durch diese Änderungen hervorgerufen werden. Die meisten dieser Techniken verwenden Kontrollflussgraphen für jede einzelne Methode, um die minimale Menge an Tests zu selektieren, die den geänderten Code durchläuft. Der Nachteil dieser Technik liegt im hohen Berechnungsaufwand, der nötig ist, um die feingranularen Coverage-Daten zu verarbeiten.

Der von uns in der Praxis evaluierte Ansatz (siehe Abschnitt V) verwendet eine DejaVu-basierte Selektionsphase, die jedoch Coverage nur auf Methoden-Granularität aufzeichnet. Wir schränken den Berechnungsaufwand außerdem durch das Herausrechnen von einfachen Refactorings ein [1].

2) Firewall-basierte Selektion: Firewall-basierte Selektionsansätze wurden zuerst von Leung und White [4] beschrieben und beschränken sich auf Integrationstests. Sie bauen einen Abhängigkeitsgraphen zwischen den Modulen eines Systems auf und markieren dann alle Module, die geändert wurden oder eine direkte Abhängigkeit zu geänderten Modulen haben, als »innerhalb der Firewall«. Anschließend werden alle Integrationstests selektiert, die Code innerhalb der Firewall ausführen. Dieser Ansatz ist *sicher* [5], er garantiert also, dass mit den selektierten Tests die gleichen Fehler gefunden werden können wie mit allen Tests. Die Sicherheit hat jedoch ihren Preis, denn es werden im Allgemeinen sehr viele Testfälle selektiert, selbst wenn diese keinen geänderten Code durchlaufen. Existierende Studien haben gezeigt, dass dabei oft ein sehr großer Teil aller Tests ausgewählt wird, selbst wenn die Änderungen selbst sehr klein sind [3].

3) Abhängigkeitsbasierte Selektion: Abhängigkeitsbasierte Selektionsansätze [8] verwenden einen statischen Funktionsaufruf-Graphen, um die Abhängigkeiten zwischen Teilen der Software zu beschreiben. Es werden dann diejenigen Tests selektiert, die direkt oder transitiv geänderte Funktionen ausführen. Der Vorteil dieser Technik ist, dass sie Coverage-agnostisch ist. Ihr Nachteil liegt in der Erstellung des Funktionsaufruf-Graphen, die sprachspezifisch ist und sogar von verwendeten Frameworks (z.B. Dependency Injection) abhängt.

[1] In der Literatur auch *»Selektive Regressionstests«* genannt.

B. Test-Priorisierung

Test-Priorisierung zielt darauf ab, eine Menge an Tests in eine bestimmte Reihenfolge zu bringen, so dass Fehler möglichst schnell gefunden werden. Hierfür gibt es in der Literatur verschiedene Ansätze, die eine solche Reihenfolge mit Hilfe unterschiedlicher Faktoren berechnen. In jedem Fall sollten Fehler jedoch schneller gefunden werden, als wenn die Tests in zufälliger Reihenfolge ausgeführt werden. Elbaum und Rothermel [6] beschreiben mehrere Herangehensweisen, die wir in vier Kategorien zusammenfassen:

- *Gesamtcoverage*: Testfälle werden anhand der durch sie hervorgerufenen Gesamtcoverage priorisiert, so dass zuerst diejenigen Testfälle ausgeführt werden, die die meiste Coverage erzeugen. Die Granularität der aufgezeichneten Coverage kann hierbei variieren (z.B. Branch-Coverage oder Methoden-Coverage).
- *Zusätzliche Coverage*: Testfälle werden anhand der durch sie *zusätzlich* hervorgerufenen Coverage angeordnet. Ausgehend von einem Coverage-Stand wird also immer der Testfall ausgewählt, der die meiste zusätzliche Coverage erzeugt. Auch hier kann die Granularität der Coverage variieren.
- *Potenzial zur Fehleraufdeckung*: Testfälle werden so angeordnet, dass diejenigen Testfälle zuerst ausgeführt werden, die das meiste Potenzial haben, Fehler tatsächlich aufzudecken. Um dieses Potenzial abzuschätzen, werden zuvor Mutation-Tests durchgeführt.
- *Gesamtfehler*: Es werden diejenigen Testfälle zuerst ausgeführt, die in der Vergangenheit am öftesten fehlgeschlagen sind.

In den Experimenten von Elbaum und Rothermel [6] stellte sich eine Kombination aus zusätzlicher Statement Coverage und Potenzial zur Fehleraufdeckung als beste Priorisierungstechnik heraus.

Der von uns in der Praxis evaluierte Ansatz (siehe nächster Abschnitt) verwendet eine Kombination aus zusätzlicher Zeilen-Coverage und erwarteter Testausführungszeit als Priorisierungstechnik.

V. Unser Test-Impact-Analyse Ansatz

Abbildung 4 gibt einen Überblick über den Aufbau der von uns eingesetzten TIA. Wie dargestellt, lässt sich die Analyse grob in zwei Phasen aufteilen.

Das Ziel von Phase 1 ist die Erhebung von *Coverage* für die einzelnen Testfälle sowie die Messung ihrer *Laufzeit*. Hierzu werden Testfälle einzeln auf der Testumgebung unter Verwendung eines Profilers ausgeführt. Dabei wird gemessen, welche Teile des Quelltexts pro Testfall durchlaufen werden und wie viel Zeit dies in Anspruch nimmt. Diese »Test-Impact-Daten« werden in einer Datenbank gespeichert.

Das Ziel von Phase 2 ist die eigentliche Generierung von Vorschlägen, welche Testfälle in welcher Reihenfolge ausgeführt werden sollen. Diese Phase wird immer dann durchgeführt, wenn Änderungen am Quelltext des *Systems under Test* vorgenommen wurden und in der Folge Tests zur Ausführung

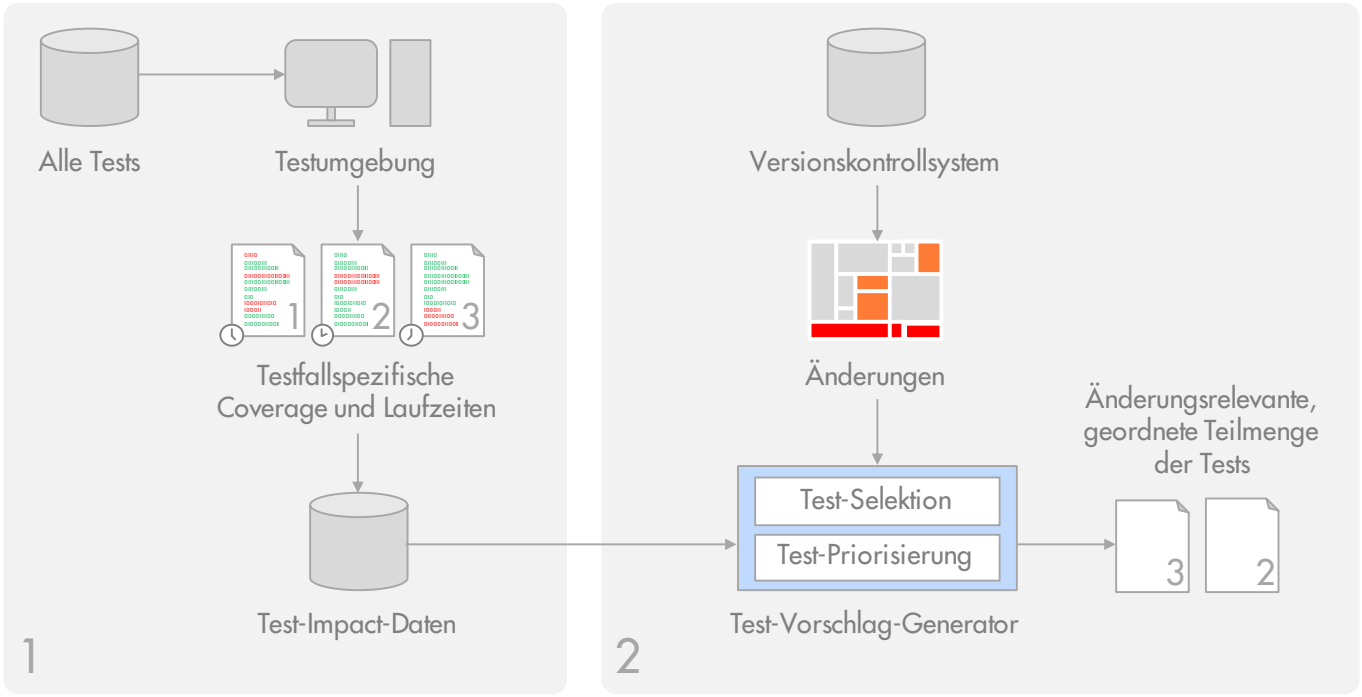

Abbildung 4. Aufbau der von uns eingesetzten Test-Impact-Analyse

kommen sollen. Ein solcher Vorschlag wird basierend auf diesen Änderungen und den in der vorherigen Phase gesammelten testfallspezifischen Test-Impact-Daten erzeugt:

- *Test-Selektion*: Es werden alle Tests ausgewählt, die die gegebenen *Änderungen* durchlaufen.
- *Test-Priorisierung*: Die ausgewählten Testfälle werden so sortiert, dass möglichst schnell alle Änderungen überdeckt werden. Dieses Problem kann jedoch auf das Mengenüberdeckungsproblem reduziert werden, welches $\mathcal{NP}$-vollständig ist. Daher verwenden wir einen heuristischen Greedy-Algorithmus, der Tests anhand ihrer zusätzlichen Coverage sortiert und in der Praxis auch bei großen Testmengen schnell genug ist.

VI. Empirische Studie: Anwendbarkeit von Test-Impact-Analyse

Die empirische Studie untersucht die Anwendbarkeit und den Nutzen von TIA in der Praxis. Erst beschreiben wir die Studienobjekte und das Studiendesign. Dann beschreiben wir die einzelnen Forschungsfragen und ihre Ergebnisse. Da die Durchführung der Berechnung der Selektion und Priorisierung für alle Studienobjekte nur Sekundenbruchteile erfordert, fokussiert die Studie auf den Nutzen der Ergebnisse und nicht auf die Berechnungszeit von TIA.

A. Studienobjekte

Die Studie wurde auf 12 Systemen durchgeführt, die alle in Java implementiert sind. Dabei wurden 11 Open Source Systeme ausgewählt, um die Studie reproduzierbar zu machen. Wir haben zusätzlich das kommerzielle Software-Analysewerkzeug Teamscale[2] hinzugenommen, da wir es als Entwickler von Teamscale sehr gut kennen und Studienergebnisse hier am besten plausibilisieren können. Die Studienobjekte unterschieden sich in Größe, Historienlänge und Anzahl der Testfälle. Die kleinsten Systeme umfassen 7k LOC (System- und Testcode), die größten über 300k, die Historienlänge geht von 44 bis 82.164 Commits. Durch die starken Unterschiede zwischen den Systemen ist es unwahrscheinlich, dass ein systemspezifischer Einzeleffekt die Studienergebnisse dominiert. Eine detaillierte Übersicht der Studienobjekte findet sich in Tabelle I.

Tabelle I
Übersicht der Studienobjekte

Studienobjekt	Σ	kLOC Source	Test	Commits
Apache Commons Collections	62	31	31	3.235
Apache Commons Lang	75	27	48	5.486
Apache Commons Math	178	87	91	7.156
Histone Template Engine 2	14	12	2	1.133
JabRef	122	94	27	10.645
Joda-Time	83	28	55	2.105
Lightweight-Stream-API	23	8	15	529
LittleProxy	9	4	5	1.037
OkHttp	52	26	26	3.548
RxJava	242	84	158	6.000
Symia Commons Math Parser	7	6	2	44
Teamscale	336	270	67	82.164

[2]www.teamscale.io

B. Studiendesign

TIA vergleicht immer zwei Versionen eines Systems miteinander. Wir nennen im Folgenden die erste Version die *baseline*, die zweite die *working copy*.

Für die Studie haben wir Mutation-Testing eingesetzt, um automatisiert Fehler in die Studienobjekte einzubauen. Dadurch kennen wir alle enthaltenen Fehler und können ermitteln, welcher Anteil davon in welcher Zeit von TIA gefunden werden kann.

Wir sind im Rahmen der Studie folgendermaßen vorgegangen, um unsere Studienobjekte zu erzeugen:

1) Als baseline haben wir jeweils ein offizielles Release eines Studienobjektes ausgewählt. Als Ausgangsversion der working copy haben wir dann einen Versionsstand genommen, der in der Versionshistorie auf die baseline folgt, und sichergestellt, dass eine substantielle Anzahl an Änderungen zwischen den Versionen stattgefunden hat. Dann haben wir über Mutation Testing in diese Version einen Fehler eingebaut, um so die working copy zu erzeugen.
2) Wir haben pro Studienobjekt zwischen 100 und 1.000 Paare von baseline und working copy erzeugt. (Wir haben die Anzahl so gewählt, dass alle Paare pro Studienobjekt in unter einem Tag Rechenzeit ausgewertet werden konnten). Die Stelle im Code, an der jeweils ein Fehler einmutiert wurde, wurde dabei zufällig gewählt.

Für jedes Studienobjekt haben wir dann auf jeder so erzeugten working copy alle automatisierten Tests ausgeführt. Wenn mindestens ein Test fehlschlägt, gilt der eingebaute Fehler als erkannt.

Zuletzt haben wir auf jedem Paar aus baseline und working copy TIA durchgeführt, um die impacted Tests zu bestimmen. Auf Basis dieser Daten beantworten wir im Folgenden die Forschungsfragen.

C. Wie zuverlässig ist Test-Impact-Analyse?

TIA führt nur impacted Tests aus, also eine Teilmenge aller Tests eines Systems. Prinzipiell könnten dabei daher Fehler unerkannt bleiben, da sie durch die impacted Tests nicht erkannt werden, aber durch die restlichen Tests erkannt werden würden. Ziel dieser Forschungsfrage ist zu quantifizieren, wie groß der Anteil der Fehler ist, die durch die impacted Tests in der Praxis nicht erkannt werden.

Hierfür berechnen wir den Anteil der einmutierten Fehler, die die impacted Tests aufdecken in Relation zum Anteil der einmutierten Fehler, die alle Tests aufdecken.

Von den 6.661 einmutierten Fehlern wurden 4.102 durch die Ausführung aller Tests des jeweiligen Systems erkannt. (Da die verbleibenden 2.559 Fehler von keinem Test erkannt wurden, können sie auch durch TIA nicht erkannt werden). Von diesen 4.102 erkannten einmutierten Fehlern werden 4.073, also 99,29% auch durch die impacted Tests allein aufgedeckt. In 7 der 12 Studienobjekte wurden 100% der Fehler erkannt, in den verbleibenden 5 Studienobjekten zwischen 90,6% und 98,7%.[3] TIA erkennt in allen Projekten über 90% der Fehler. In mehr als der Hälfte der Projekte wurden jeweils alle Fehler erkannt.

Tabelle II
EINSPARUNG DURCH AUSFÜHRUNG DER IMPACTED TESTS

	Laufzeit (ms)		
Studienobjekt	Alle Tests	Impacted Tests	Ersparnis
Apache Commons Collections	25.277	610	97,59%
Apache Commons Lang	24.987	3.111	87,55%
Apache Commons Math	160.391	114.042	28,90%
Histone Template Engine 2	34.603	32.204	6,93%
JabRef	119.849	40.721	66,02%
Joda-Time	20.782	1.297	93,76%
Lightweight-Stream-API	1.523	480	68,48%
LittleProxy	155.334	150.406	3,17%
OkHttp	96.671	76.457	20,91%
RxJava	464.018	170.575	63,24%
Symia Commons Math Parser	528	528	0,00%
Teamscale	1.249.088	196.684	84,25%

D. Welche Zeiteinsparung ergibt die Beschränkung der Ausführung auf die impacted Tests?

Ziel dieser Forschungsfrage ist es, zu quantifizieren, wie stark die Dauer der Tests in der Praxis beschleunigt wird, wenn nur die impacted Tests ausgeführt werden.

Hierfür berechnen wir die Einsparung der Laufzeit der impacted Tests in Relation zur Ausführung aller Tests. Diese Einsparung tritt bei jedem Testlauf auf, auch wenn kein Fehler gefunden wird.

Die Einsparung reicht von 0% bis 97,6% und unterscheidet sich damit sehr stark zwischen den Studienobjekten. Im Durchschnitt spart die Selektion 52% der Testausführungszeit, im Median über alle Studienobjekte 64% der Zeit. Die detaillierten Ergebnisse sind in Tabelle II dargestellt.

Die starken Unterschiede bei der Einsparung liegen an der unterschiedlichen Menge an Code, die von den einzelnen Tests in den unterschiedlichen Studienobjekten durchlaufen werden. In *Apache Commons Collections* durchlaufen die meisten Tests nur sehr wenig Code, der sich zwischen unterschiedlichen Tests stark unterscheidet. Das kommt der Selektionsphase der TIA stark zugute. Im Fall von *Symia Commons Math Parser* hingegen starten viele Tests den gesamten Parser. Dadurch wird ein großer Teil des Codes von allen Tests ausgeführt, was die Nützlichkeit der Selektionsphase reduziert.

E. In welcher Zeit findet Test-Impact-Analyse den ersten fehlschlagenden Test?

Bereits wenn der erste Testfall fehlschlägt ist bekannt, dass die untersuchte Software nicht fehlerfrei ist (und bspw. ein Build nicht deployed werden kann), ohne auf die Ergebnisse

[3]Tiefergehende Analyse der 29 unerkannten Fehler hat dabei ergeben, dass die Mehrheit auf nichtdeterministische Testfälle (und damit instabile Test-Coverage) zurückzuführen ist. Eine angepasste Messung von Test-Coverage, die Testfälle mehrfach ausführt und nur die Methoden zur Coverage rechnet, die bei jedem Durchlauf ausgeführt werden, würde diese Probleme vermutlich reduzieren.

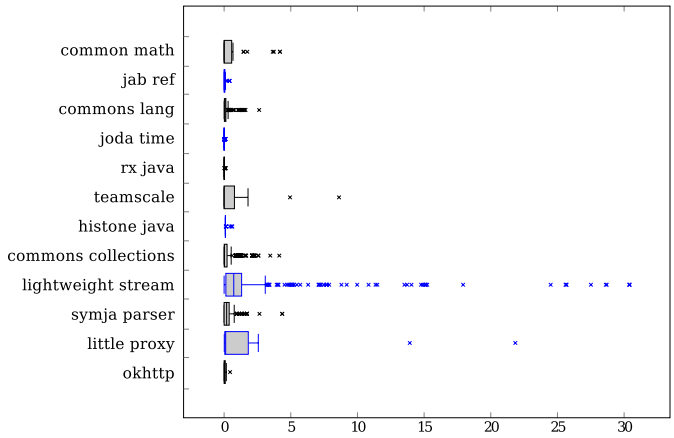

Abbildung 5. Prozentuale Verteilung der Zeiten bis zur Aufdeckung des ersten Fehlers

der restlichen Testfälle warten zu müssen. Diese Forschungsfrage quantifiziert, wie schnell TIA den ersten fehlschlagenden Test erkennt.

Hierfür berechnen wir, in welcher Zeit welcher Anteil der eingebauten Fehler gefunden wird, wenn wir die impacted Tests in der vorgeschlagenen Reihenfolge ausführen.

Hierfür haben wir für alle 4.073 Paare, bei denen die impacted Tests einen Fehler finden, ermittelt, nach welcher Ausführungszeit der erste Test fehlschlägt. Dann haben wir für jedes Paar die relative Dauer bis zur Erkennung des ersten Fehlers ermittelt. Als Beispiel: Falls der erste Fehler nach 3 Sekunden gefunden wird, die gesamte Suite aber 100 Sekunden läuft, ergibt sich hierfür ein Wert von 3%.

Die statistischen Verteilungen sind in Abbildung VI-E dargestellt. Für jedes System ist die Verteilung der Werte über einen Box-Plot[4] dargestellt. Da sich der Großteil der Werte im Bereich von unter 5% bewegt, ist nur der Bereich von 0% bis 30% der Laufzeiten dargestellt, um die Box-Plots nicht noch stärker zu stauchen. Jeder Box-Plot stellt in der Box die Werte des zweiten und dritten Quartils dar. Der Median wird als senkrechte Linie in der Box dargestellt. Bei einigen Systemen sind die Werte so dicht gedrängt (z.B. *JabRef*), dass die Box zu einem schmalen senkrechten Strich zusammenfällt. Der Box-Plot zeigt klar, dass die überwältigende Mehrheit der Werte zwischen 0% und 2% liegt.

F. Wie wahrscheinlich werden Fehler innerhalb eines begrenzten Zeitfensters erkannt?

Um zu garantieren, dass die Testausführung im Rahmen der Continuous Integration rechtzeitig zu Ende ist, reicht es nicht, die Testausführung bei Erkennung des ersten Fehlers anzuhalten. Denn falls kein Fehler enthalten ist, werden dann alle impacted Tests ausgeführt (was, wie oben gesehen, in Extremfällen die Ausführung der gesamten Test-Suite zur Folge hat. Stattdessen muss die Testausführung nach einer vorher definierten Zeitspanne abgebrochen werden. Diese Forschungsfrage quantifiziert daher, wie zuverlässig TIA innerhalb einer begrenzten Zeitspanne erkennt, dass ein System fehlerhaft ist.

[4]Die genaue Interpretation ist bspw. in https://de.wikipedia.org/wiki/Box-Plot beschrieben.

Tabelle III
PROZENTUALER ANTEIL ERKANNTER FEHLERHAFTER SYSTEMVERSIONEN NACH ANTEIL TESTZEIT

Studienobjekt	1%	2%	5%	10%
Apache Commons Collections	84,25	96,58	99,83	100,00
Apache Commons Lang	94,23	96,43	98,49	99,31
Apache Commons Math	80,20	85,92	91,55	92,96
Histone Template Engine 2	88,46	88,46	88,46	88,46
JabRef	95,71	95,71	95,71	95,71
Joda-Time	98,10	98,61	99,11	99,87
Lightweight-Stream-API	60,76	90,73	94,74	97,25
LittleProxy	74,29	77,14	80,00	80,00
OkHttp	90,70	90,70	90,70	96,51
RxJava	91,89	94,59	94,59	94,59
Symia Commons Math Parser	78,46	86,15	89,23	89,23
Teamscale	91,86	95,93	96,57	96,57

Hierfür haben wir ausgewertet, welcher Anteil der Fehler gefunden wird, wenn nur der Anteil der impacted Tests ausgeführt wird, der in 1%, 2%, 5% oder 10% der Dauer aller Tests ausgeführt werden kann. Die Ergebnisse sind in Tabelle III dargestellt. Bereits in 1% der Zeit werden zwischen 60,8% und 98,1% der Fehler gefunden, im Median 89,6%, im Mittel 85,7%. Bei 2% der Dauer bereits zwischen 77,1% und 98,6%, im Median 92,7% und im Mittel 91,4%.

VII. AUSBLICK

Wir planen, TIA in folgende Richtungen weiterzuentwickeln:

Teamscale-Entwicklungsprozess: Wir haben vor einiger Zeit TIA in unseren eigenen Entwicklungsprozess integriert. Entwickler können auf ihren Feature-Branches selbst entscheiden, ob alle Tests, oder nur die impacted Tests ausgeführt werden. Wir planen, TIA für alle unsere Branches als Default-Teststrategie einzuführen, wenn wir weitere Erfahrungen gesammelt haben.

Programmiersprachen: Aktuell setzen wir TIA nur für Java ein. Wir planen, sowohl die Werkzeugunterstützung als auch die Studien, auf weitere Programmiersprachen zu erweitern.

Benchmark: Wir setzen aktuell einen Mutation-Based-Benchmark zur Beantwortung der Forschungsfragen ein. Wir sind gerade dabei, einen Benchmark auf Basis von echten Fehlern aus Open-Source-Projekten aufzubauen, um die Forschungsfragen auf den Fehlern quantifizieren zu können, die historisch in diesen Projekten durch die CI gefunden wurden. In laufenden Arbeiten wurden hierzu 437 fehlerhafte Versionsstände aus 31 Projekten analysiert. Die Ergebnisse ähneln den Ergebnissen des Mutation-Based-Benchmarks in diesem Papier stark, sind aber noch nicht final.

Test-Arten: Wir arbeiten gerade an einem Forschungsprojekt, das TIA für Hardware-In-the-Loop-Tests der Software eines eingebetteten Steuergeräts evaluiert. Die ersten Ergebnisse

sind auch hier sehr vielversprechend, aber auch diese Studie ist noch nicht abgeschlossen.

TIA-Strategien: Wir planen, weitere Ansätze zu TIA auf Basis des Mutation-Based-Benchmarks und auf Basis der Fehlerdaten aus realen Projekten zu evaluieren. Nicht nur, um zu prüfen, ob es bessere Ansätze gibt als den von uns verfolgten, sondern auch, welche Ansätze geeignet sind, falls bspw. keine testfallspezifische Coverage erhoben werden kann.

VIII. Zusammenfassung

Unser Ansatz für Test-Impact-Analyse selektiert und priorisiert Testfälle auf Basis der Änderungen seit dem letzten Testlauf. Dadurch kann bei jedem Lauf der Continuous Integration ein (kleiner) Teil einer großen Test-Suite zur Ausführung ausgewählt werden. In unserer empirischen Studie haben wir die Zuverlässigkeit und den Nutzen auf 12 Java Systemen aus der Praxis quantifiziert.

TIA hat in der Praxis eine sehr hohe Zuverlässigkeit von 99,26% über alle Studienobjekte (im schlechtesten Einzelfall von 90,6%). Während die Einsparung bei Ausführung aller impacted Tests zwischen den Systemen stark schwankt, ist die Verkürzung der Testzeiten bis zur Aufdeckung des ersten Fehlers über alle Systeme hinweg sehr deutlich.

Das größte Potential von TIA ist daher die Verkürzung des Zeitraums zwischen Start eines Testlaufs und Aufdeckung des ersten Fehlers. Bei nur 1% Testlaufzeit deckt TIA im Mittel (und im Median) in über 80% der Fälle, bei 2% Testlaufzeit in über 90% der Fälle auf, dass die untersuchte Systemversion fehlerhaft ist.

Anders ausgedrückt kann die Testlaufzeit um 98% verkürzt werden und trotzdem werden nur in 10% der Fälle fehlerhafte Builds nicht als solche erkannt. Gerade für Systeme, deren Tests so lange laufen, dass sie überhaupt nicht im Rahmen der Continuous Integration ausgeführt werden, stellt das eine substantielle Verbesserung dar, da nun 90% aller Fehler im Rahmen der CI erkannt werden, und nicht erst in späten Testphasen aufgedeckt werden, die ggf. erst Monate später stattfinden.

Da TIA nicht alle Fehler aufdeckt, müssen in regelmäßigen Abständen nach wie vor alle Tests ausgeführt werden. Da die meisten Fehler allerdings bereits im Rahmen der CI aufgedeckt werden, sollten bei der Ausführung aller Tests vergleichsweise weniger Fehler auftreten. Dadurch sollte auch der zusätzliche Aufwand, der durch die Verzögerung und die Überlagerung von Fehlern auftritt, vergleichsweise geringer sein.

Danksagung

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

Literatur

[1] Florian Dreier, Elmar Juergens, and Andreas Goeb. Detection of refactorings. Bachelor's thesis, Technische Universität München, 2015.

[2] Emelie Engström, Per Runeson, and Mats Skoglund. A systematic review on regression test selection techniques. *Inf. Softw. Technol.*, 52(1):14–30, January 2010.

[3] Elmar Jürgens, Benjamin Hummel, Florian Deissenboeck, Martin Feilkas, Christian Schlögel, and Andreas Wübbeke. Regression test selection of manual system tests in practice. In *Proceedings of the 15th European Conference on Software Maintenance and Reengineering (CSMR'11)*, pages 309–312, 2011.

[4] H.K.N. Leung and L. White. A study of integration testing and software regression at the integration level. In *Proceedings of the 1990 Conference on Software Maintenance*, 1990.

[5] G. Rothermel and M. J. Harrold. Analyzing regression test selection techniques. *IEEE Transactions on Software Engineering*, 22(8):529–551, Aug 1996.

[6] G. Rothermel, R. H. Untch, Chengyun Chu, and M. J. Harrold. Prioritizing test cases for regression testing. *IEEE Transactions on Software Engineering*, 27(10):929–948, Oct 2001.

[7] Gregg Rothermel and Mary Jean Harrold. A safe, efficient regression test selection technique. *ACM Trans. Softw. Eng. Methodol.*, 6(2):173–210, April 1997.

[8] Ye Wu, Mei-Hwa Chen, and Howard M. Kao. Regression testing on object-oriented programs. In *Proceedings of the 10th International Symposium on Software Reliability Engineering*, ISSRE '99, pages 270–, Washington, DC, USA, 1999. IEEE Computer Society.