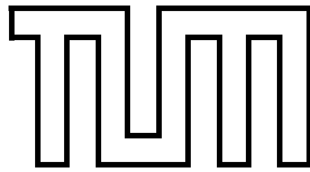


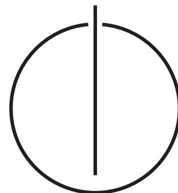
FAKULTÄT FÜR INFORMATIK

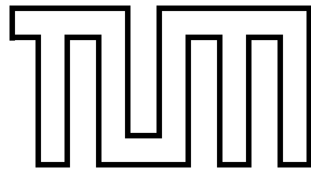
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatik

Untersuchung von Change-Request-Coverage als Metrik zur Qualitätssicherung von Software-Tests

Jakob Rott





FAKULTÄT FÜR INFORMATIK

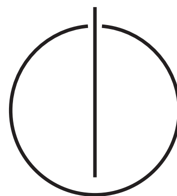
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatik

Untersuchung von Change-Request-Coverage als Metrik zur Qualitätssicherung von Software-Tests

Change request coverage as a metric of quality assurance for software tests

Bearbeiter:	Jakob Rott
Aufgabensteller:	Prof. Dr. Dr. h.c. Manfred Broy
Betreuer:	Dr. Elmar Jürgens
	Dr. Dennis Pagano
Abgabedatum:	15. März 2016



Ich versichere, dass ich diese Bachelor's Thesis in Informatik selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

München, den 15. März 2016 Jakob Rott

Zusammenfassung

Bei langlebigen Systemen treten vor allem dort Fehler auf, wo seit dem letzten Release Änderungen vorgenommen wurden. Test-Manager versuchen daher, diese Änderungen besonders ausführlich testen zu lassen. Studien haben jedoch gezeigt, dass das ohne gezielte Werkzeugunterstützung in der Praxis nicht gut gelingt. Um Test-Manager besser zu unterstützen, bietet die Test-Gap-Analyse ein Analyseverfahren, das ungetestete Änderungen während der Testphase ermittelt.

Test-Gap-Analyse zeigt die Test-Gaps in Form von geänderten und ungetesteten Methoden im Quelltext an. Da Tester den Quelltext häufig nicht kennen, wäre es jedoch wünschenswert, die Test-Gaps auf eine Art und Weise darzustellen, die auch Testern zugänglich ist.

Change-Requests dokumentieren die durchgeführten Änderungen an einem Software-System. Sie sind natürlichsprachlich formuliert und damit auch Nicht-Programmierern zugänglich. Außerdem gibt es in vielen Systemen eine zuverlässige Verlinkung zwischen Change-Requests und Code-Änderungen, auf deren Basis der Zusammenhang zu den Test-Gaps hergestellt werden könnte.

Diese Arbeit stellt die Konzeption einer neuen Metrik, der Change Request Coverage (CRC), vor und untersucht deren Nützlichkeit in einer empirischen Studie am Beispielprojekt Teamscale. Es zeigte sich, dass die CRC testabhängige Resultate liefert und dass es Möglichkeiten gibt uninteressante Test Gaps systematisch auszuschließen. Durch eine Entwicklerbefragung wurde bestätigt, dass die CRC wichtige Testlücken aufdeckt.

Inhaltsverzeichnis

Abstract **i**

1 Einleitung **1**

1.1 Berechnung von Test-Gaps 2
1.2 Problemstellung 3
1.3 Beitrag der Arbeit zur Test-Gap-Analyse 4

2 Grundlagen und Verwandtschaften **5**

2.1 Grundlagenbegriffe 5
2.2 Verwandte Arbeiten zur Change Request Coverage in der Literatur 10

3 Change Request Coverage als Metrik **11**

3.1 Zusammenhang von Test-Gap-Analyse und Change Request Coverage 11
3.2 Berechnung der Change Request Coverage 12

4 Entwickeltes Werkzeug zur Studie **15**

4.1 Bestimmung der Change Request Coverage zu einem CR 15
4.2 Vorteile und Nachteile bei der Implementation als Teamscale Service 18

5 Empirische Studie zur Change Request Coverage am Beispiel von Teamscale **19**

5.1 Forschungsfragen 19
5.1.1 RQ1: Testunabhängiger Anteil in der Change Request Coverage 19
5.1.2 RQ2: Uninteressante Test-Gaps in der Change Request Coverage und deren systematischer Ausschluss 20
5.1.3 RQ3: Auffindung relevanter Test-Gaps mit Hilfe der Change Request Coverage 21
5.2 Aufbau der Studie 21
5.2.1 Studienobjekt 21
Untersuchtes Projekt 21
Untersuchte Change Requests 22
5.2.2 Design der Studie 24
5.2.3 Methodik 25
Bestimmung des Changesets eines CRs 25
Testfälle 26
Allgemeine Vorbereitung des Testsetups 26
Erzeugung der Basis Coverage 27
Erzeugung der Testfall Coverage 27
Erstellung eines Bewertungs-Fragebogens zur CRC . 27
5.3 Ergebnisse 29

5.3.1 Gewichtung mit Methodenlänge 31
5.3.2 RQ1 32
Betrachtung einzelner CRs und Beispielfälle 32
Betrachtung über alle getesteten CRs 33
5.3.3 RQ2 34
5.3.4 RQ3 39
5.4 Interpretation / Beantwortung der Forschungsfragen 45
5.4.1 Gewichtung mit Methodenlänge 45
5.4.2 RQ1 46
5.4.3 RQ2 46
5.4.4 RQ3 47
5.5 Gefährdungen der Validität 49

6 Fazit / Zusammenfassung **50**

7 Ausblick **51**

Anhang **54**

A Beispiel: CRC Bogen zu CR#8737 **55**

B Besonderheiten im Testverfahren **58**

C Relation von Zeitdifferenz zwischen letztem Commit und Schließung eines CRs zu „intermediate“ Methoden **60**

Literatur **61**

1 Einleitung

Program testing can be used to show the presence of bugs, but never to show their absence!

EDSGER WYBE DIJKSTRA, 1972 [DIJ72]

With information about change coverage, testing efforts can be assessed and redirected if necessary, because the probability of bugs is increased in changed-untested methods.

EDER ET AL., 2013 [EDE+13]

Der Hauptgrund für Software Evolution sind wachsende und sich wandelnde Anforderungen aus der Fachlichkeit. Im Prozess dieser Fortentwicklung werden zum Beispiel aufgedeckte Fehler behoben und Funktionalitäten des Programms den Bedürfnissen der Benutzer angepasst. Entwickler von Softwaresystemen sind stets bemüht eine möglichst fehlerfreie Arbeit zu leisten und um dies sicherzustellen werden Tests verschiedener Natur entworfen und ausgeführt. Diese können beispielsweise automatisch ablaufende Unit-Tests sein oder Tests, die durch Fachpersonal nach festgelegten Spezifikationen ausgeführt werden. Die Tests überprüfen, ob entwickelte Komponenten so oder immer noch so arbeiten, wie beabsichtigt.

Außerdem können Tests auch die Entwicklung erleichtern, zum Beispiel indem sie eine Sicherheit bieten für Entwickler, die den Code anderer Entwickler verändern [Mar09]. Die Hemmschwelle zur Änderung an fremden Code sinkt intuitiv, wenn nach der Änderung ein Test signalisieren kann, dass durch die vorhergehende Änderung eine bisher funktionierende Komponente nun falsche Ergebnisse zurückliefert.

Der Softwaretest einer neuen Software unterscheidet sich von dem einer veränderten Software. Im zweiten Fall soll besonders der geänderte Teil des Programms betrachtet werden, denn die meisten Feldfehler sind in geänderten, aber ungetesteten Softwareteilen zu erwarten [Ede+13]. Test-Gap-Analyse hilft Test-Managern früh zu erkennen, wo sich ungetesteter, aber veränderter Code im System befindet. In diesem Kapitel wird zunächst beschrieben, was Test-Gap-Analyse ausmacht und wie sie heute eingesetzt wird. Im Anschluss wird geschildert, wie die vorliegende Arbeit eine Lücke schließen kann, die es bisher in der Nutzung von Test-Gap-Analyse durch Personen gab, die den Quelltext nicht kennen.

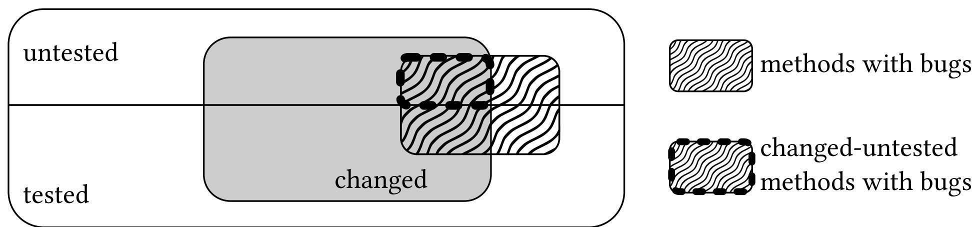


Abbildung 1.1: Eder et al. machen deutlich, dass der geänderte, aber nicht getestete Bereich einer Software am stärksten fehlergefährdet ist. (Abbildung aus [Ede+13])

1.1 Berechnung von Test-Gaps

Fehler in Softwaresystemen sind unerwünscht. Aus diesem Grunde wird zu verschiedenen Zeitpunkten im Entwicklungsprozess getestet, um eventuell auftretende Fehler vor dem Benutzer/Kunden zu finden und sie noch vor der Verbreitung zu korrigieren. Betrachtet man zwei Versionen eines Softwareprodukts, das von einer zur nächsten Version weiterentwickelt (also nicht vollständig neu aufgebaut) wird, können diese Auswirkungen auf die Methodenmenge des Systems beobachtet werden (vgl. auch Abb. 1.1):

- Einen Anteil der Methoden, die es in der *vorhergehenden Version* des Produktes schon gab, gibt es in **ungeändert**er Art und Weise auch wieder in der Folgeversion.
- Ein Anteil der Methoden, die es in der *vorhergehenden Version* des Produktes schon gab, wird im Entwicklungsprozess **geändert** und existiert in der Folgeversion also in geänderter Form.
- Ein Anteil der Methoden der *Folgeversion* ist **neu**, entstand also erst im zwischenzeitlichen Entwicklungsprozess.
- Ein Anteil der Methoden, die es in der *vorhergehenden Version* des Produktes gab, wurde im Entwicklungsprozess **gelöscht**. Die gelöschten Methoden existieren in der Folgeversion nicht mehr, können also weder in einer Test- noch in einer Produktivumgebung ausgeführt werden.

Fehler treten häufiger an den Stellen auf, die zwar geändert oder neu sind, aber nicht getestet wurden [Ede+13]. In Abbildung 1.1 wird veranschaulicht in welchen Teilen eines geänderten Programms häufiger Fehler zu vermuten sind.

Bei der Test-Gap-Analyse wird die Menge an geänderten und hinzugefügten Methoden (Punkte 2 und 3 der Aufzählung) betrachtet. Die Methoden in der Menge sind entsprechend ihres Status als „geändert“ oder „neu“ markiert. Testcoverage aus verschiedenen Umgebungen – genannt seien Unit und manuelle Tests (wiederum aus möglicherweise unterschiedlichen Testumgebungen) – kann über diese Menge gelegt werden. Methoden, die Testcoverage erfahren haben, werden als „getestet“ markiert, wohingegen die anderen Methoden ihre Markierung „geändert“ oder „neu“ behalten. Das Softwarequalitäts-Analysetool Teamscale unterstützt die Analyse von Test-Gaps und stellt diese übersichtlich dar (vgl. Abb. 1.2). Jedes Rechteck steht hier für eine Methode. Die Methoden, die nicht geändert wurden, werden grau angezeigt. Neue oder geänderte getestete Methoden

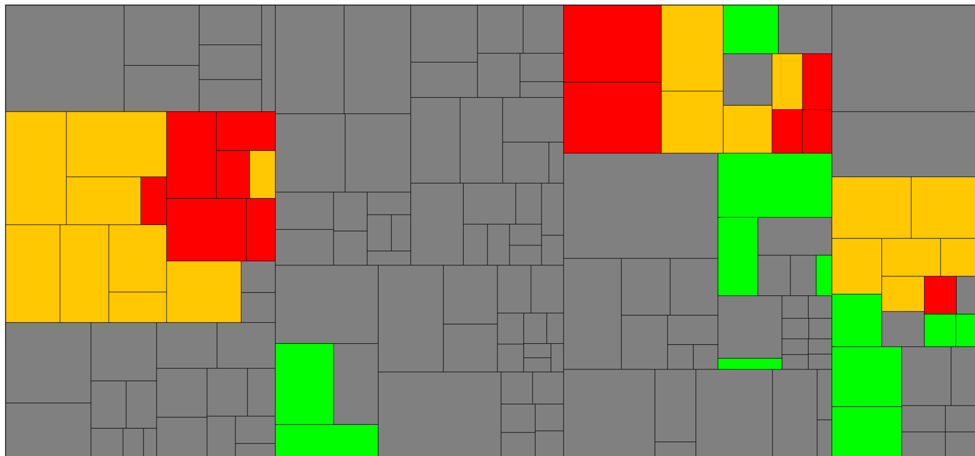


Abbildung 1.2: Test-Gap Treemap in Teamscale; Jedes Rechteck stellt eine Methode dar; grau: nicht geändert, grün: neu/geändert und getestet, orange: geänderte Methode (nicht getestet), rot: hinzugefügte Methode (nicht getestet)

werden durch grüne Rechtecke visualisiert. Nicht getestete, also Test-Gap Methoden werden entweder in orange (geänderte Methode) oder in rot (hinzugefügte Methode) abgebildet.

Dies ermöglicht eine einfache Diagnose, welche Teile des Systems noch ungetestet sind, ein Test aber empfehlenswert ist.

1.2 Problemstellung

Solange sich diejenigen, die sich mit den Test-Gaps eines System beschäftigen, auch in der Codebasis gut auskennen, ist die Präsentation auf Methodenbasis der Ergebnisse von Test-Gap-Analysen sehr hilfreich. Allerdings ist dies häufig nicht der Fall: In sogenannten Black-Box Tests, die in der Industrie sehr üblich sind, ist den Testern (gesamte Gruppe, der an der Softwareprodukttestung beteiligten Personen) der Code fremd. Manche Unternehmen haben eine komplett ausgelagerte Abteilung für Softwaretests und manuelle Testfälle werden lediglich nach der Testspezifikation ausgeführt. Diese Tester haben keine Kenntnis über den Quelltext und dürfen diesen in manchen Fällen auch gar nicht einsehen. Dann ist der direkte Mehrwert eines rot aufleuchtenden Rechtecks, welches für eine ungetestete hinzugefügte Methode steht, oder die Anzeige, wie viel Prozent des geänderten Systemsteils insgesamt abgedeckt wurden, gering.

Ein Test-Gap mag durch die Nichtausführung passender Testfälle bedingt sein und kann durch die Aufdeckung und das Hinzufügen weiterer auszuführender Testfälle bereinigt werden, bevor ein neues Release veröffentlicht wird.

Unvollständige Testfälle sind weitere Gründe für Testdefizite. Diese zu beheben, erfordert es im schlimmsten Fall Rücksprache mit dem Entwickler zu halten, der Auskunft darüber geben kann, warum spezifische Methoden nicht ausgeführt wurden. Um den „richtigen“ Entwickler zu finden und zu ermitteln, in welchem Zusammenhang die Methode editiert wurde, müssen aufwendig Daten manuell

aus dem Kontrollsystem gelesen werden. Der Entwickler kann dann Aufschluss darüber geben, warum diese Methode nicht ausgeführt wurde und der unvollständige Testfall kann angepasst/verbessert werden.

Das Schließen von Test-Gaps ist ein sehr aufwendiger Prozess, da Tester keine Aussagen über die Herkunft von Test-Gaps machen können. Entwickler und Tester sitzen in großen Unternehmen oftmals an verschiedenen Orten, gegebenenfalls sogar in unterschiedlichen Ländern und mit oraganisatorisch wirkenden Personen (Managern) dazwischen. Es ist also angebracht, an geeigneter Stelle (Kommunikations)Wege zu sparen und eine andere Metrik zu erstellen, die Testern und Test-Managern, die wenig oder kein Wissen über den zu testenden Code haben, eine effektive Möglichkeit gibt, Test-Gaps und deren Herkunft aufzudecken.

1.3 Beitrag der Arbeit zur Test-Gap-Analyse

Die Arbeit untersucht die Möglichkeit, mit der Change Request Coverage (CRC) die Lücke in der Anwendbarkeit der Test-Gap-Analyse zu schließen. Die Change Request Coverage bringt den Vorteil mit sich, dass nun für jeden einzelnen in der Entwicklungsphase fertiggestellten Change Request (CR) ein Wert verfügbar ist, der als *Key Performance Indicator* dienen kann. Kritische (ungetestete) CRs sind durch diese schnell zu identifizieren. Für Test-Leiter und Test-Manager ist dies hilfreich, da die bestehende Test-Gap-Analyse die Ergebnisse lediglich pro Methode anzeigt oder als aggregierten Wert über alle Änderungen. Bei der Change Request Coverage ist nicht nur zu erkennen, dass Test-Gaps existieren, sondern es wird auch ihr Ursprung herausgestellt.

Bisherige Darstellungen von Test-Gaps erfordern Kenntnis über den Quelltext und über die Tätigkeiten der Methoden. Der hier erstmalig vorgestellte Ansatz bietet einen Zugang auf höherem Abstraktionslevel zu den Ergebnissen der Test-Gap-Analyse. Test-Leiter und Test-Manager bekommen eine ebenso eindeutige wie gleichermaßen einfach zu verstehende Rückmeldung darüber, ob ein Change Request ausreichend getestet ist, ohne selbst den Quelltext kennen, gar verstehen oder gehäufte Nachfragen an Entwickler stellen zu müssen.

Eine empirische Überprüfung des Ansatzes ergab, dass die Change Request Coverage testabhängig ist, sich uninteressante Gruppen von Test-Gaps ausschließen lassen und durch die CRC nach Aussagen der Entwickler testenswerte Test-Gaps aufgedeckt werden.

2 Grundlagen und Verwandtschaften

Fehlervorhersage (*defect prediction*) kann als Rahmenthema zu dieser Forschung angesehen werden. Eine der Prämissen dabei ist, dass eine hohe Abdeckung ($\hat{=}$ Ausführungsdichte) der Methoden durch Tests und insbesonde derer, die seit dem letzten Release verändert wurden, zu einem stabileren Softwaresystem führt.

Die Punkte in diesem Kapitel erläutern Grundlagen auf denen das Modell der CRC aufbaut. Ähnliche Modelle zur Change Request Coverage finden sich derzeit nicht in der Literatur; so können am Schluss des Kapitels lediglich etwaige Beziehungen zwischen Ansätzen angesprochen werden.

2.1 Grundlagenbegriffe

Die folgenden Absätze beschreiben Begrifflichkeiten, die zum Verständnis der Arbeit beitragen.

Erneuter Test modifizierter Softwareteile Dass die Wartung eines langlebigen Softwaresystems innerhalb dessen Lebenszeit den größten Auswand ausmacht, ist schon lange bekannt und unumstritten. Basil Sherlund stellte 1995 auf der Konferenz „Defect Prevention - 12th International Conference and Exposition on Testing Computer Software“ in einer Präsentation [She95] seinen Ansatz vor, der (wie diese hier vorliegende Arbeit) den Fokus darauf legt, Modifikationen im Programm zu testen. Sherlund baut auf einem existierenden Verfahren auf und versucht dabei zwei Defizite zu beheben: (1) Die Befriedigung von Testkriterien darf nicht geschehen, ohne die modifizierten Stellen im Code ausgeführt zu haben. (2) Auch Modifikationen, die den Datenfluss nicht beeinflussen, sollten getestet werden. Sherlund stellt einen Protypen vor, der dieses Testsystem implementiert. Er nutzt dabei einen ähnlichen Aufbau zu dem, der in dieser Arbeit verwendet wird. So stellt sein Prototyp die modifizierten Teile eines Systems heraus, die anschließend getestet werden sollen. Reichen die durchgeführten Tests nicht aus – das heißt, überdeckt die Test Coverage nicht alle identifizierten Änderungen – so werden weitere Tests angewendet, um die Änderungen adäquat abzudecken.

Noch zehn Jahre vor Sherlunds erster Veröffentlichung zu diesem Vorgehen beschäftigten sich Fischer et al. im Jahr 1981 mit Methoden, um modfizierte Software erneut zu testen [FRC81]. In ihrem Papier beschränken sie sich auf Tests in der Produktiv- und Wartungsphase, da diese kostenintensiver als die Testphase seien. Ihr Ziel ist es, ein Tool zu entwickeln, das einerseits die hohen Testkosten reduzieren kann und andererseits die Zuverlässigkeit von gerade modifizierten Systemanteilen sichert. Unter anderem schlagen sie als eine Indikatormetrik zur

Wartbarkeit die Anzahl an Testfällen vor, die nötig sind, um eine Software erneut ausreichend zu testen.

Obwohl sich seit 1995 viel und noch mehr seit 1981 im Feld des Software Engineering getan hat und die Programmiersprachen heute noch komplexere Möglichkeiten von Abhängigkeiten mit sich bringen, bleiben die wesentlichen Gedanken von Sherlund und Fischer et al. aktuell. Zur Kostenreduzierung der Tests und gleichzeitigen Validierung gerade der modifizierten Softwareteile muss analysiert werden, wie abgesichert werden kann, welche Auswahl an Testfällen Modifikationen ausreichend abdeckt.

Testabdeckung Die Abdeckung ($\hat{=}$ Ausführung) von Methoden kann für unterschiedliche Programmiersprachen mit verschiedenen Tools festgestellt werden. So sind hierfür für `Java` zum Beispiel gängige Tools: *Cobertura*[1], *EMMA*[2] oder *Jacoco*[3]. Oder für `C`: *gcov*[4] oder *Cantata++*[5] oder *Tessy*[6]. Die Ausgabe dieser Tools reicht von reinen Ausführungsdaten bis hin zur Auswertung dieser Daten in Form von Reports. Die festgestelle Coverage kann auf Basis verschiedener Entitäten durchgeführt werden. Üblicher Weise wird die Abdeckung von Funktionen/Methoden, Statements oder Konditionen (wurde jeder boolische Ausdruck mit wahr und falsch ausgewertet?) berechnet oder die Vollständigkeit in der Ausführung von Zweigen/Pfaden überprüft. Einige Tools, die zur Messung von Coverage dienen, bieten noch breitere Spektren aus verschiedenen Funktionen, die beim Entwicklungsprozess hilfreich sein können [YLW09].

Zusammenhang von Qualität/Verlässlichkeit und Testabdeckung Isoliert betrachtete Test Coverage ist keine Metrik, mit der eine Aussage über die Zuverlässigkeit eines Softwaresystems zu machen ist. Malaiya et al. erstellten ein mathematisches Modell, um unter Zuhilfenahme der Testcoverage die Wahrscheinlichkeit von Fehlerauftritten in Systemen bestimmen zu können. Unter anderem untersuchten sie ein Projekt der NASA zum Sensor-Management. Die dazugehörigen Tests wurden gestaffelt ausgewertet und es ließ sich feststellen, dass dort mit steigender Branch Coverage die relative Fehlerdichte abnimmt [Mal+02].

Selektion manueller Regressionstests Evaluationen zu selektivem Regissionstesten sind nach [Jue+11] fokusiert auf automatisierte Tests. Dieses 2011 veröffentliche Papier beschäftigt sich unter anderem mit der Frage, ob Coverage über die Entwicklung des Systems hin stabil ist. Also ob modifizierte Anteile durch Tests, die vor der Modifikation entworfen wurden, abgedeckt werden oder nicht (vgl. beschriebene Indikatormetrik weiter oben aus [FRC81]).

Das Problem ist nicht unbekannt und wird auch in [RH94] behandelt. Die Autoren schreiben hier, dass ihre Methode, die auf Abhängikeitsgraphen basiert, dieje-

1 `http://cobertura.sourceforge.net`
2 `http://emma.sourceforge.net`
3 `http://eclemma.org/jacoco/`
4 `https://gcc.gnu.org/onlinedocs/gcc/Gcov.html`
5 `http://www.qa-systems.com/cantata.html`
6 `http://www.razorcat.eu/tessy.html`

nigen existieren Testfälle herausfindet, die unterschiedliche Ausgaben erzeugen könnten.

Unvollständige Testfälle Unvollständige Testfälle sind ein Problem, das in der empirischen Studie (Kap. 5) dieser Arbeit auftritt. In [Jue+11] wurde eine Fallstudie in der Industrie durchgeführt, in der für drei Testfälle überprüft wurde, wie hoch die Wahrscheinlichkeit für eine Methode – die *manchmal* durch einen Testfall ausgeführt wird – ist in jedem Test ausgeführt zu werden. Dazu wurden die drei Testfälle spezifikationsgerecht ausgeführt, wobei in diesen keine exakten Schritte festgehalten waren, die zum Beispiel die Navigation durch das Nutzerinterface betreffen. Die herausgefundenen Wahrscheinlichkeiten lagen bei 92%, 93% und 33% [Jue+11]. Das bedeutet für die ersten beiden Testfälle, dass sich eine niedrige Varianz in der Anzahl der ausgeführten Methoden je Testfallausführung ergab. Im dritten Fall zeigt sich allerdings, dass viele Methoden je nach Durchführung des Testfalls nicht ausgeführt wurden. Seine Definition bietet gegebenenfalls mehr Spielraum für Tester und der Testfall determiniert somit schlechter die Menge an Methoden, die durch ihn ausgeführt wird.

Für manche Testfälle lässt sich also gut, für andere weniger gut vorhersagen, welche Methoden durch sie ausgeführt werden. Dies sollte generell bedacht werden, wenn Analysen anhand von Test Coverage stattfinden.

Test-Gap-Analyse und Test-Gap Die Test-Gap-Analyse deckt ungetestete Änderungen an einem Softwaresystem auf. In einer großen Fallstudie von Eder et al. stellte sich heraus, dass ein Fehler in einem veränderten, aber nicht getesteten (*changed-untested*) Systemteil wahrscheinlicher ist als in anderen Subsystemen ([Ede+13], vgl. dazu auch Abb. 1.1).

Ein **Test-Gap** ist der veränderte, aber ungeteste Anteil einer neuen Softwarerevision. Methoden im Test-Gap werden als Test-Gap Methoden oder im geigneten Kontext kurz ebenfalls als Test-Gap bezeichnet.

Test-Manager Sie sind zuständig dafür, dass wesentliche Teile eines Softwaresystems getestet sind, bevor das Release (besipielsweise im Kundenkontext) zum Einsatz kommt. Dafür müssen Tests ausgewählt werden, die alle essentiellen Teile des Systems testen. Diese Hauptaufgabe des Test-Managers ist in der ITIL Publikation zur Serviceinbetriebnahme (engl. Service Transition) festgehalten [Off11].

Auswirkungsanalyse im Software Engineering Dieses Gebiet beschäftigt sich mit der Frage, welche Konsequenzen mit der Änderung an einer Stelle im Quelltext einhergehen. Ändert man nur eine lokale Variable, so sind damit keine Änderungen anderswo im Code verbunden. Die Situation verhält sich anders, wenn man eine häufig genutzte Utility-Methode ändert und sie beispielsweise fortan keine Exception mehr auslösen wird. Diese Änderung schlägt sich auf alle Methoden nieder, die die geänderte Utility-Methode aufrufen. Entweder entfällt dadurch eine `throws`-Deklaration, oder ein `try ... catch ...`-Block. Speziell in großen Systemen spielt die Auswirkungsanalyse eine Rolle [ZSK14; Boh96].

Zuordnung von Funktionalitäten zu Quelltext In [WS95] wird ein Problem beschrieben, das Entwicklern begegnet, die Code bearbeiten, der in der Vergangenheit und gegebenenfalls nicht von ihnen selbst geschrieben wurde: Das Wissen, in welchem Teil des Quelltextes eine Funktionalität implementiert ist, fehlt. Soll etwas daran geändert werden, muss der richtige Angriffspunkt im Quelltext erst gefunden werden. Die beschriebene Vorgehensweise ist unkompliziert und hilft diese Kenntnis zu erlangen. Dazu benötigt wird lediglich ein Coverage Monitor, einige einfache Tools und eine niedrige Anzahl von Testfällen [WS95].

Im Ablauf unterscheidet sich das Verfahren im abstrahierten Sinne wenig von dem einer funktionellen Magnetresonanztomographie. Dort wird der untersuchten Person eine Aufgabe gestellt und die Hirnaktivitäten während dieser aufgenommen. Auf einem Bildschirm erscheinen aktive Hirnregionen farbig, während die anderen in Graustufen repräsentiert werden.

Bei der Zuordnung von Funktionalitäten übernimmt die Rolle des Gehirns der Quelltext; die auszuführende Aufgabe muss das Feature ausmachen, dessen Implementation im Code lokalisiert werden soll; Hirnregionen sind Methoden. [WS95] schlägt auf anderen Entitäten basierende individuelle Einteilungen vor. Die Darstellung kann auch mit Farbmarkierungen auf einem Bildschirm erfolgen, zum Beispiel in einer Treemap, vgl. Abb. 4.1.

Die Methode zum Erstellen des Mappings (vgl. [Wil+03]) sowie der Nutzen desselben sind vielfältig. Die in dieser Arbeit vorgestellte Change Request Coverage berechnet den Ort der Funktionalität intern. In der Ausgabe beschränkt sie sich dann allerdings auf die Orte, die innerhalb einer Funktionalität im Rahmen eines CRs einer Änderung unterzogen wurden.

Versionskontrollsysteme Programmquelltext ist ein wertvolles Gut in der Softwareentwicklung. Änderungen an einer Codebasis sollen nachvollziehbar und wenn nötig wieder reversibel und alte Versionen wiederherstellbar sein. Gängige Versionskontrollsysteme in der Softwareentwicklung sind beispielsweise Subversion (SVN) und Git. Die Systeme überwachen einen Ordner auf einem Speichermedium und bieten nach außen hin einen Serverdienst an, um Änderungen zu tätigen (Ordner/Dateien hinzufügen, löschen oder verändern). Jede Änderung heißt *Commit*, kann von einem möglicherweise authentifizierten Nutzer (Softwareentwickler) getätigt werden und hat im System einen eindeutigen Identifikationsschlüssel. Automatisch wird vom Versionskontrollsystem festgehalten, von wem der Commit getätigt wurde, welche Nachricht der Benutzer damit verknüpft hat, was sich im Rahmen dessen verändert hat etc. Diese Informationen werden in ein Logfile geschrieben und sind für die Benutzer auslesbar. Ein Beispiel für einen Ausschnitt aus einer Versionshistorie sieht wie folgt aus:

```
------------------------------------------------------------------------
r22103 | streitel | 2016-02-03 11:35:06 +0100 (Mi, 03. Feb 2016) | 3 Zeilen
Geänderte Pfade:
   M /trunk/engine/com.teamscale.index/src/com/teamscale/index/testgap/TestGapInfo.java
   M /trunk/engine/com.teamscale.index/src/com/teamscale/index/testgap/
       TestGapTreeMapService.java
   M /trunk/engine/com.teamscale.index/src/com/teamscale/index/treemap/TreeMapNodeBase.
       java

CR#9783:
YELLOW
aggregating BW objects to the class level in the TGA treemap
```

```
------------------------------------------------------------------------
r22013 | streitel | 2016-02-01 11:20:28 +0100 (Mo, 01. Feb 2016) | 3 Zeilen
Geänderte Pfade:
   M /trunk/engine/com.teamscale.index/src/com/teamscale/index/testgap/tfs/
       TfsBuildNotificationService.java

CR#8652:
YELLOW
removed last open comment (is moot)
```

Es ist möglich aus dem Versionskontrollsystem den Stand des Quelltextes zu einem beliebigen Commitzeitpunkt zu beziehen.

Projektmanagementsoftware („Issue Tracker") / Change Request (CR) Geschieht Software Evolution in einem organisierten Umfeld, wie es in größeren Projekten der Fall sein sollte, so wird meist eine Projektmanagementsoftware (PMS) eingesetzt, die es bei der Organisation von Softwareentwicklungen erleichtert, den Überblick zu behalten. PMS selbst gibt es in verschiedenen Ausführungen, sie kann in einem Desktopprogramm oder webbasiert realisiert sein und bietet Möglichkeiten, das Projekt zeitlich zu planen und Informationen über dessen Entwicklung zu beziehen. Betrachtet man eine neue Versionsentwicklung, so kann in einer PMS also festgehalten werden, welche Meilensteine bis zum Release der neuen Version erfüllt sein sollen, bis wann diese erledigt sein sollen, in welche Unterprozesse sie einzuteilen sind und wer diese erledigt. Meilensteine sowie Unterprozesse können und sollen als „Change Request" (CR; Änderungsanforderung) formuliert werden, sodass nachvollziehbar ist, welchen Stand die Entwicklung in diesem Belang (also Fehlerbehebung, Funktionalitätserweiterung etc.) hat, wer sich damit beschäftigt, auf welchem Branch entwickelt wird, bis zu welcher Iteration oder welchem Release die Bearbeitung beendet sein soll. Change Requests sind nummeriert und gibt man in einem Commit im Versionskontrollsystem eine CR Nummer an, so kann eine Verknüpfung zwischen den Änderungen im Code und dem CR hergestellt werden.

Teamscale Teamscale[7] ist ein Softwareprodukt der CQSE GmbH (Firmensitz: Garching bei München) und dient zur kontinuierlichen Qualitätsanalyse von in Entwicklung befindlicher Software. Teamscale hilft bei der Detektion von Quelltextduplikaten, kann Architekturkonformitätsanalysen durchführen, untersucht inkrementell die von den Entwicklern getätigten Commits, warnt bei potentiell problematischen Stellen im Quelltext und kann Aspekte der Softwarequalität unterschiedlicher Revisionen miteinander vergleichen. Außerdem – und für das Thema dieser Arbeit besonders relevant – unterstützt Teamscale den Import von Coveragedaten und kann Test-Gaps berechnen und in Treemaps darstellen. Es unterstützt mit diesen und vielen weiteren Funktionalitäten all jene, die bei der Entwicklung und Qualitätssicherung eines Programms mitwirken.

Das Teamscale Backend ist in Java implementiert, das Frontend ist webbasiert. Teamscale beruht auf insgesamt etwa 285 000 Zeilen Quelltext, wovon der Javaanteil etwa 75% ausmacht. Nach ersten Prototypen im Jahr 2010 startete die Entwicklung von Teamscale im Oktober 2012 und steht derzeit unter dem Einfluss etwa 30 aktiver Entwickler.

7 https://www.teamscale.com

Changeset eines CRs Das „Changeset“ bezeichnet die Menge der Methoden, die innerhalb eines CRs geändert oder hinzugefügt wurden.

Change Request Coverage (CRC) „Change Request Coverage“ (CRC) ist der Name des hier vorgestellten Ansatzes, der Daten aus Projektmanagementsoftware, Versionkontrollsystemen und Tests integriert. Für jeden CR kann mit der CRC festgestellt werden, welcher Anteil der geänderten Methoden durch einen Test abgedeckt ist.

2.2 Verwandte Arbeiten zur Change Request Coverage in der Literatur

Literatur zu Metriken mit Daten aus manuellen Softwaretests ist rar. Die Nutzung von Coverage Informationen, der Zusammenhang von diesen und der Qualität von Software, das Mapping von Programmfeatures zu Orten im Quelltext, die Auswahl von Testfällen auf Basis von geänderten/hinzugefügen Methoden wird in der Literatur vielfach besprochen und in Kapitel 2 mit entsprechenden Verweisen vorgestellt. Die Integration zur Change Request Coverage und diese als Metrik einzusetzen, ist neu. Eine Parallele kann jedoch zum Prototypen aus [She95] (auch im Grundlagenkapitel beschrieben) gezogen werden. Es gibt hier zweierlei Unterschiede: Zum einen wirken sich die Coverageinformationen aus dem Test wieder *direkt* verbessernd/erweiternd auf den Test aus (dieser Vorgang wird so lange wiederholt, bis die Coverage im gewünschten Maß erreicht wurde). Zum anderen ist der Input hier nicht ein Change Request und die dazugehörige Spezifikation, sondern die Gesamtheit der gemachten Änderungen seit dem letzten Referenzpunkt. Wird ein System zur Verwaltung von Change Requests verwendet und alle Änderungen im Code müssen auf einen CR aus diesem System referenziert werden, so kann die Gesamtmenge der Änderungen seit dem letzten Referenzpunkt auf die Menge aller seit dem letzten Referenzpunkt implementierten CRs zurückgeführt werden.

3 Change Request Coverage als Metrik

Die Change Request Coverage soll Testern, die keine Kenntnis des Quelltexts haben, helfen schnell erkennen zu können, ob der Test, den sie ausgeführt haben auch die Änderungen im Sourcecode des Change Request abdeckt. In diesem Kapitel wird zunächst beschrieben, wie die Test-Gap-Analyse mit der Change Request Coverage zusammenhängt. Anschließend werden die notwendigen Schritte zur Berechnung knapp dargestellt und Darstellungsmöglichkeiten aufgezeigt, wie die CRC präsentiert werden kann.

3.1 Zusammenhang von Test-Gap-Analyse und Change Request Coverage

Das Ergebnis einer Test-Gap-Analyse wird bisher im abgedeckten Anteil von Quelltextzeilen oder Methoden ausgegeben. Für jede neue oder geänderte Quelltextzeile/Methode wird geprüft, ob diese durch einen entsprechenden Test abgedeckt wurde.

Die Change Request Coverage bestimmt hingegen für einen ganzen Change Request die Coverage. Das bedeutet, dass zunächst von einem Change Request auf ein Changeset an Methoden geschlossen werden muss. Dieses besteht aus allen Methoden, die im Rahmen des CRs hinzugefügt oder geändert wurden. Ergebnisse einer Test-Gap-Analyse liefern Informationen zur Abdeckung einzelner Methoden. Diese werden mit dem Changeset des CRs abgeglichen und so die Ergebnisse für den CR aggregiert.

Der Zusammenhang von Change Request zu einer Änderungsmenge von Methoden und deren Tests ist in Abbildung 3.1 dargestellt.

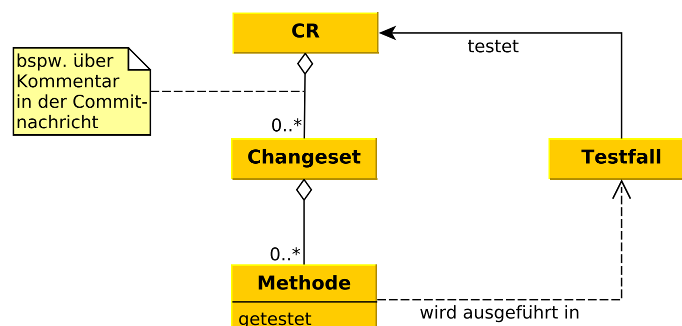


Abbildung 3.1: Ausgehend vom Change Request kann mit Daten aus Kontrollsystemen sowohl ein Methoden Changeset, als auch ein Testfall bestimmt werden.

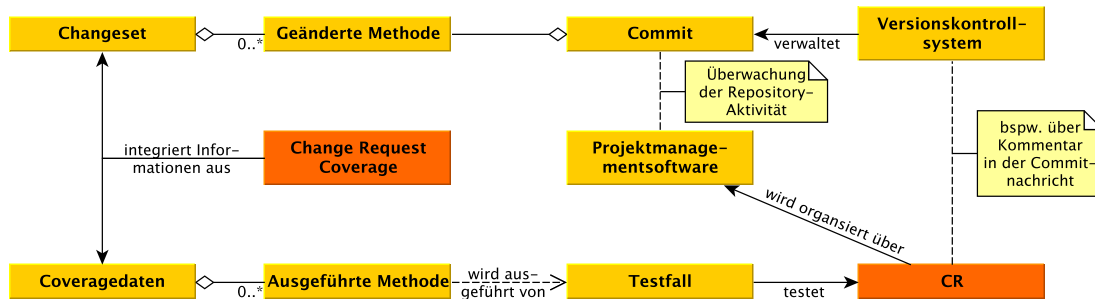


Abbildung 3.2: Modell zur Berechnung der CRC mit Darstellung der Beziehung zwischen den teilhabenden Systemen.

3.2 Berechnung der Change Request Coverage

Kontrollsysteme erlauben es, auch über große Softwareentwicklungen einen Überblick zu behalten. In diesem Abschnitt wird dargelegt, wie man aus Projektmanagementsoftware und Versionskontrollsystemen die Change Request Coverage bestimmen kann.

Zur Berechnung der Change Request Coverage soll man sich der Informationen bedienen, die die vom Projekt benutzte Projektmanagementsoftware und die Versionskontrollsysteme bereithalten. Die Berechnung für die hier eingeführte Change Request Coverage für einen bestimmten Change Request soll folgendermaßen erfolgen:

Aus der Projektmanagementsoftware wird abgerufen, welche Commits im Versionskontrollsystem dem Change Request zuordenbar sind. Das ist dann möglich, wenn die Projektmanagementsoftware die Commits im Versionskontrollsystem überwacht und im Commit festgehalten ist, zu welchem Change Request die Änderung gehört. Eine geeignete und gängige Möglichkeit, diese Information einzufügen, ist eine Commitnachricht immer mit der dazugehörigen CR-Nummer zu beginnen. Zum Beispiel: `CR#2840: Improved authentification mechanism.`

Aus dem Versionskontrollsystem kann nun für jeden Commit überprüft werden, welche Änderungen an der Programmbasis vorgenommen wurden. Betrachtet man nun die Unterschiede in den geänderten Dateien, so können Methoden identifiziert werden, die im Commit geändert oder hinzugefügt wurden. Hat man dies für jeden Commit eines Change Requests erledigt, so resultiert daraus das Changeset, die Menge der im CR geänderten/hinzugefügten Methoden.

In der Projektmanagementsoftware ist festgehalten, welche funktionalen Änderungen ein bearbeiteter Change Request mit sich bringen soll. Ein Testfall, der diese funktionalen Änderungen testet, wird nun auf einem Testsystem ausgeführt. Der Testlauf wird von einem Profiler überwacht, der aufzeichnet, welche Methoden (oder auch genauer, welche Quelltextzeilen/Statements) im Programmlauf ausgeführt wurden und dies in einer Datei festhält. Zur Berechnung der Change Request Coverage müssen dann noch die Daten aus dem Versionskontrollsystem (welche Methoden wurden hinzugefügt/geändert) und der Ausführungsanalyse (welche Methoden wurden im Testfall ausgeführt) abgeglichen werden. Eine schematische Darstellung, wie in der Berechnung in der zur Arbeit

gehörenden Studie vorgegangen wurde und weiteres dazu findet sich in Kapitel 5.

Die Aussagekraft der Change Request Coverage ist am besten, wenn die Methoden aus dem Changeset exklusiv durch den Testfall abgedeckt werden und nicht etwa schon „zufällig" beim Start des Systems. Dazu werden zusätzlich Informationen über die Coverage erhoben, die durch den Start (genauer: durch ein festgelegtes Startprozedere) des Systems entstand, um Einheitlichkeit sicherzustellen. Es wird im Folgenden unterschieden zwischen dieser Coverage, die als „Base"- oder „Initialisations"-Coverage geführt wird und der „Testfall"-Coverage, die Coverage bezeichnet, die ausschließlich durch den Test gewonnen wurde.

Die Zusammenhänge im Berechnungsprozess sind auch in Abbildung 3.2 darstellt.

Input Benötigt wird für einen gegebenen CR:

- die Spezifikation mindestens eines Testfalls, der die funktionalen Änderungen des CRs abdeckt;
- eine festgelegte Startprozedur, um zwischen „Base"- und „Testfall"-Coverage unterscheiden zu können;
- ein Programm, um die Ausführungsinformationen während des Testvorgangs aufzuzeichnen;
- die Möglichkeit, von CR auf Commits und deren Inhalt zu schließen. Im vorhergehenden Abschnitt wird beschrieben, wie dies durch die Zusammenarbeit von Projektmanagementsoftware und Versionskontrollsystemen möglich ist.

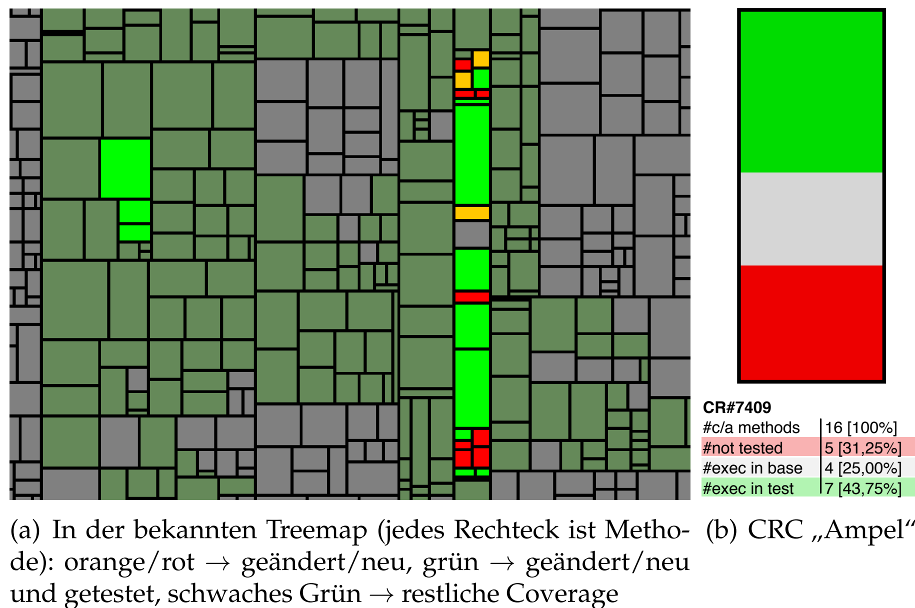
Nach Testausführung Nun ist verfügbar:

- das Changeset des CRs, also die Menge der Methoden, die im Rahmen des CRs geändert oder hinzugefügt wurden;
- die Menge der Methoden, die im ausgeführt wurde (Coverage).

Output Die Ergebnisse zur Change Request Coverage können

- tabellarisch ausgegeben werden: Die Tabelle enthält für einen CR Informationen darüber, wie hoch der Anteil an ausgeführten Methoden (exklusiv durch den Test oder beim Programmstart) ist. Siehe Tabelle 3.1.
- grafisch ausgegeben werden:
  - Durch Farbgebungen in der methodenbasierten Treemap (bekannt aus Abb. 1.2). Siehe Abbildung 3.3(a): Ungeänderte/geänderte/hinzugefügte Methoden sind weiterhin grau/orange/rot. In deutlichem Grün sind nun geänderte/hinzugefügte Methoden dargestellt, die durch den Test abgedeckt sind. Die Methoden in schwachem Grün sind ausgeführt, aber nicht verändert worden. Hier findet keine Unterscheidung der Coverage Mengen (Startszenario/Testfall) statt.
  - Durch eine Change Request Coverage „Ampel", die Aufschluss gibt, wie groß die getesteten/ungetesteten Anteile des CRs sind. Man beachte, dass dies zwar eine gute Übersicht (gerade über mehrere Change

Request Coverages) gibt, allerdings noch eine Aufschlüsselung der Orte beispielsweise der Test-Gaps nötig macht. Siehe dazu Abbildung 3.3(b) oder auch die Abbildungen der Studie in Kaptiel 5.



(a) In der bekannten Treemap (jedes Rechteck ist Methode): orange/rot → geändert/neu, grün → geändert/neu und getestet, schwaches Grün → restliche Coverage

(b) CRC „Ampel“

Abbildung 3.3: Ausgabe der Change Request Coverage in einer methodenbasierten Treemap durch Farbgebung (a) oder in einer Coverage Ampel (b) hier mit Legende.

Tabelle 3.1: Darstellung von Ergebnissen zur CRC hier für den CR#8737. Die Tabelle kann - wenn gewünscht - zum Beispiel auch den Ort der Methode anzeigen

CR	#8737
Titel	Display occurrence line number(s) for in coming and out going dependencies.
Geänderte Methoden	14
Methoden im Testfall ausgeführt	9
Methoden durch Startprozedere ausgeführt	0
„intermediate“ Methoden	1
Abdeckung	64,3%

4 Entwickeltes Werkzeug zur Studie

Für die empirische Studie in Kapitel 5 wurde ein Werkzeug in Java entwickelt, das die Berechnung der Change Request Coverage durchführt. In diesem Kapitel werden die implementieren Funktionalitäten des Werkzeugs dargelegt und auch betrachtet, welche Vor- und Nachteile die Implementation als Erweiterung von Teamscale böte.

4.1 Bestimmung der Change Request Coverage zu einem CR

Die Hauptaufgabe des entwickelten Werkzeugs ist die Bestimmung der Change Request Coverage.

Gegeben ist die Menge der geänderten Methoden in einem CR, das Changeset in Listenform[1] und außerdem eine Teamscale Instanz, die das Projekt unter Test analysiert. An die Teamscale Instanz wurden Coverageinformationen gesendet. Zum einen die Coverage, die beim Start bis zu einem definierten Zeitpunkt ausgeführt wurde; im Folgenden als Coverage `BASE` bezeichnet. Zum anderen die als im Folgenden Coverage `TEST` bezeichnete Coverage, die während des gesamten Testlaufs ermittelt wurde, also während der gesamten Ausführung des für den CR festgelegten Testfalls inklusive des Systemstarts.

Das Werkzeug lädt aus der Teamscale Instanz eine Treemap, welche für das gesamte Projekt für die Coverage `TEST` Coverageinformationen zu Methoden visualisiert. Eine solche Treemap wird auch Execution Treemap genannt. Ein Beispiel dazu findet sich in Abbildung 4.1. Die Treemap besteht aus grauen und grünen Rechtecken. Die grauen stehen für nicht ausgeführte Methoden und die grünen für ausgeführte Methoden. Orangefarbene und rote Rechtecke (wie aus der Test-Gap Treemap bekannt) tauchen nun nicht mehr auf. Das Werkzeug analysiert die in JSON geladene Treemap und erzeugt daraus eine Liste der Methoden, die während des gesamten Tests ausgeführt wurden.

Diese Information kann mit dem Changeset des CRs abgeglichen werden. Der Schlüssel zum Vergleich besteht aus Dateipfad, Methodenname und Parametertypen der Methode. Für jede Methode des Changesets wird nun überprüft, ob sie auch in der Liste der ausgeführten Methoden enthalten ist. Wenn ja, so wird die Methode der Liste `executedMethodList` hinzugefügt. Andernfalls der Liste `testGapMethodList`.

Nun sollen die Listen `executedMethodList` und `testGapMethodList` noch aufgeteilt werden. **(1)** Bei den ausgeführten Methoden soll unterschieden werden, ob sie durch die „Base"-Coverage abgedeckt wurden oder exklusiv durch den Test

1 Die Bestimmung des Changesets wird mit weniger programmatischem Hintergrund in Abschnitt 5.2.3 im Studienteil der Arbeit beschrieben.

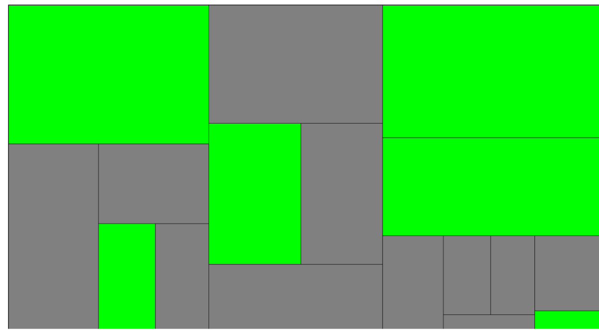


Abbildung 4.1: Eine Execution Treemap: Grüne Rechteckte zeigen Methoden, die in einer ausgewählten Coverage als ausgeführt markiert wurden. Graue Rechtecke zeigen nicht ausgeführte Methoden.

(„Test"-Coverage). **(2)** Jede Methode aus der Liste `testGapMethodList` soll noch auf ihre Existenz in der ausgeführten Version hin überprüft werden. Existiert eine Methode nicht mehr, so soll sie auch nicht als Test-Gap Methode angezeigt werden.

Zu (1): Es wird wieder eine Execution Treemap für das gesamte Projekt geladen. Nun allerdings mit der Coverage `BASE`. Wieder wird eine Liste der ausgeführten Methoden aus dem JSON erstellt. Methoden, die sich darin befinden und gleichzeitig in der Liste `executedMethodList`, werden verschoben in die Liste `executedMethodInBaseList`. Die Liste `executedMethodList` enthält nun nur noch Methoden, die exklusiv durch den definierten Testfall ausgeführt wurden und kann umbenannt werden in Liste `executedMethodInTestList`.

Zu (2): Zu jeder Methode der `testGapMethodList` wird der Quelltext (aus der ausgeführten Version) der Klasse abgerufen, in der sich die Methode befindet. Lässt sich die Methode in diesem nicht mehr finden (gleicher Name, gleiche Methodenparametertypen), so gibt es sie in der Form nicht mehr, wie sie im CR editiert wurde. Die Methode existiert zum Zeitpunkt des Tests also nicht mehr. Eine nicht existierende Methode ist für die Test-Gap-Analyse unerheblich (vgl. Abschnitt 1.1) und ist kein Test-Gap. Methoden, die nicht mehr existieren werden in die Liste `intermediateMethodList` verschoben. Man beachte, dass in dieser Studie keine Aussage darüber getroffen werden kann, ob die Methode im Rahmen des CRs gelöscht wurde oder nicht. Sie kann durch Entwicklungsarbeit in anderen CRs verändert, umgezogen oder gelöscht worden sein.

Listing 4.1: Pseudocode zur Bestimmung der 3 Listen `testGapMethodList`, `executedMethodInBaseList` und `executedMethodInTestList`. Diese enthalten die nötigen Methodeninformationen zur CRC-Berechnung zum beispielhaft gewählten CR#1234.

```
int cr = 1234;
List<MethodInfo> changeset = holeChangesetZuCR(cr);
List<MethodInfo> coverageTest = holeAusgefuehrteMethodenMenge(cr,
    "TEST");

foreach changedMethod in changeset {
    if (coverageTest.contains(changedMethod)) {
        executedMethodList.add(changedMethod);
    } else {
```

```
            testGapMethodList.add(changedMethod);
        }
    }

    coverageBase = holeAusgefuehrteMethodenMenge(cr, "BASE");

    foreach executedInTestOrBaseMethod in executedMethodList {
        if (coverageBase.contains(executedInTestOrBaseMethod)) {
            executedMethodInBaseList.add(executedInTestOrBaseMethod);
            executedMethodList.remove(executedInTestOrBaseMethod);
        }
    }

    executedMethodInTestList = executedMethodList;
```

Für alle verbleibenden Methoden im Test-Gap wird der Quelltext der Methode bereitgehalten, um eine spätere Bewertung (interessant/nicht interessant) zu erleichtern.

Im Anschluss werden noch drei verschiedene Prozentwerte zur Change Request Coverage errechnet, die die Methoden, die durch die „Base"-Coverage abgedeckt werden, unterschiedlich behandeln.

- In bisher üblichen Darstellungen zur Testüberdeckung kann nicht unterschieden werden, ob eine Methode exklusiv durch den Testfall abgedeckt wurde oder schon durch das Startprozedere. Die erste Berechnung lässt dieser Darstellung äquivalent die möglicherweise beiläufige Abdeckung durch die „Base"-Coverage ohne negative Bewertung zu.

$$\frac{\text{Anzahl ausgeführter neuer/geänderter Methoden}}{\text{Gesamtanzahl neuer/geänderter Methoden}}.$$

- Methoden, die in der Coverage `BASE` ausgeführt wurden, werden von der Change Request Prozentwertberechnung ausgeschlossen. In dieser Berechnung sollen Methoden, die durch die „Base"-Coverage abgedeckt werden keinen Einfluss auf die CRC nehmen.

$$\frac{\begin{array}{l}\text{Anzahl ausgeführter neuer/geänderter Methoden}\\ \qquad \text{abzüglich derer, die in } \texttt{BASE} \text{ ausgeführt wurden}\end{array}}{\begin{array}{l}\text{Gesamtanzahl neuer/geänderter Methoden}\\ \qquad \text{abzüglich derer, die in } \texttt{BASE} \text{ ausgeführt wurden}\end{array}}.$$

- Methoden, die in der Coverage `BASE` ausgeführt wurden, werden als ungetestet gewertet. Ihre Abdeckung geschah nur beiläufig; die Ausführung des festgelegten Testfalls konnte über sie keine weitere Information bringen, als es die Startprozedur schon tat.

$$\frac{\begin{array}{l}\text{Anzahl ausgeführter neuer/geänderter Methoden}\\ \qquad \text{abzüglich derer, die in } \texttt{BASE} \text{ ausgeführt wurden}\end{array}}{\text{Gesamtanzahl neuer/geänderter Methoden}}.$$

Zuletzt wird ein PDF-Dokument erstellt, das die abgerufenen und errechneten Informationen in aufbereiteter Form bereitstellt.

4.2 Vorteile und Nachteile bei der Implementation als Teamscale Service

Durch eine Kooperation mit der CQSE GmbH wäre es möglich gewesen, die Berechnung der Change Request Coverage direkt als Service von Teamscale zu implementieren. Teamscale erstellt intern eine Methodenhistorie, die es möglich macht, zu einer Methode - betrachtet zu einem bestimmten Zeitpunkt - deren Vorgänger (bis hin zur ersten Version der Methode) ausfindig zu machen. Die Tests in der Studie in Kapitel 5 wurden auf einer Teamscale-Revision getestet, die sehr zeitnah zur Schließung des Change Request liegt. Das ist notwendig, damit die Methoden, die im Rahmen des CRs geändert/hinzugefügt wurden, zum Zeitpunkt des Tests noch an dem Ort sind, an dem sie im Rahmen des CRs editiert wurden.

Bemerkung 1:
Würde die Datei umbenannt, so würde das entwicklte Werkzeug die Methoden nicht mehr finden und sie als intermediate Methoden ausgeben. Die Aussagekraft der Change Request Coverage nähme so ab.

Mit Hilfe der Teamscale Methodenhistorie könnte man die Methode auch nach einem Umzug noch finden und feststellen, ob sie vom Test abgedeckt wurde oder nicht.

Zum Zweck der Studie und damit für den Prototyp der Change Request Coverage Berechnung wurde sich dazu entschieden keinen Teamscale Service zu implementieren, sondern das weiter oben beschriebene Werkzeug. Teamscale bietet eine gut dokumentierte REST Schnittstelle, die es möglich macht, die benötigten Informationen zur Change Request Coverage abzurufen. Die Teamscale Instanz der Teamscaleentwickler analysiert Teamscale selbst und kennt die Historie des Projekts und ist an den Issue Tracker angebunden. Auf dieser Instanz hätte ein Prototyp zur CRC Berechnung nicht ohne weiteres eingesetzt werden können und Änderungen am Prototyp wären schwierig möglich. Eine eigene Instanz für die Studie setzte eine Analyse der Teamscalehistorie der letzten 20 Monate voraus, die etwa 4-5 Tage benötigte und deshalb unpraktikabler zu nutzen wäre als die bestehende Instanz.

Sollte sich die Change Request Coverage als nützliche Metrik herausstellen und man sich dazu entschließen die Metrik als Teamscale Service zu implementieren, so wäre das als Fortschritt zu bewerten, da die Methodenhistorie genutzt werden könnte. Vergleiche dazu auch Kapitel 7 („Ausblick“).

5 Empirische Studie zur Change Request Coverage am Beispiel von Teamscale

Die Change Request Coverage als theoretische Idee wurde in den vorangegangenen Kapiteln vorgestellt. Nach der konzeptuellen Betrachtung soll nun der Fokus darauf liegen, wie gut sich die Change Request Coverage in der Praxis berechnen lässt und wie nützlich sie dort ist.

Zu diesem Zweck wurde das Projekt Teamscale (vgl. auch Grundlagen-Kapitel) der CQSE GmbH als Studienobjekt gewählt. Durch die Untersuchung soll herausgefunden werden, ob sich die Change Request Coverage eignet, um relevante Test-Gaps zu finden.

Zunächst werden die Forschungsfragen vorgestellt, die es in der Studie zu beantworten gilt. Danach wird der Aufbau der Studie erläutert. Zuletzt werden die Ergebnisse der Studie präsentiert und diskutiert.

5.1 Forschungsfragen

In Kapitel 3 wurde vorgestellt, wie die Change Request Coverage als Metrik zur Qualitätssicherung von Softwaretests konstruierbar ist. Die nachstehenden Forschungsfragen sollen dazu dienen, Stärken und Schwächen dieses Konzepts zu finden, deren Herausstellung es ermöglicht das Konzept zu verbessern und die Aufmerksamkeit in Sonderfällen zu schärfen.

RQ1 Wie viel Change Request Coverage ist bei manuellen Tests unabhängig vom eigentlichen Test und damit aussagelos?

RQ2 Warum sind manche Test-Gaps für Tester uninteressant und können wir sie systematisch ausschließen?

RQ3 Finden wir aus Sicht des Entwicklers/des Testers relevante Test-Gaps?

Die Fragen werden im Folgenden einzeln besprochen und eine grobe Vorgehensweise zu deren Bearbeitung beschrieben.

5.1.1 RQ1: Testunabhängiger Anteil in der Change Request Coverage

Es soll hier explorativ herausgefunden werden, wie hoch der testunabhängige Anteil der Change Request Coverage ist. Würde festgestellt werden, dass in vielen/den meisten Fällen bereits alle geänderten Methoden durch den Start des System unter Tests ausgeführt werden, so wäre die Change Request Coverage testunabhängig. Das ist allerdings nicht gewollt. Die Aussagekraft der Change Request Coverage ist nur dann hoch, wenn der Anteil der Methoden, die bereits

beim Start ausgeführt werden, niedrig ist und der Großteil exklusiv durch den Test ausgeführt wird.

Zur Beantwortung dieser Frage ist es notwendig für jeden CR unter Test zwei Coverage-Dateien zu erstellen. Dabei beinhaltet die eine die „Base“-Coverage. Die andere Coverage-Datei beinhaltet die „Base“- und „Testfall“-Coverage.

Negativbeispiel: In der Praxis, ist die folgende Situation vorstellbar, wenn man den testunabhängigen Teil der CRC nicht beachtet: Ein Tester möchte CR#0012, CR#0123 und CR#1234 testen und hat dafür spezifizierte Testfälle vorliegen. Für jeden der CRs führt der Tester den passenden Testfall aus, lässt die Ausführungsinformationen mitschreiben und erfährt vom Test-Lead, dass die Change Request durch die Tests eine hohe Coverage haben. Später führt er das Programm unter Test ein weiteres Mal aus, ohne einen Test auszuführen und schneidet wiederum die Ausführungsinformationen mit. Aus Interesse berechnet der Tester mit diesen Daten die Change Request Coverage für die CR#0012, CR#0123 und CR#1234. Wieder ergibt sich eine hohe CRC. Scheinbar wurden viele der Methoden, die bei der ersten Programmausführung aufgerufen wurden auch bei der zweiten aufgerufen. Zwei Beispiele sind hier gegeben, wie es dazu kommen kann: Bei einem Programm mit grafischer Oberfläche wird beim Start immer der Teil des Systems ausgeführt, der die Oberflächendarstellung implementiert. Beim Start eines Server wird immer eine Routine ausgeführt, die den Dienst initialisiert, also zum Beispiel das Programm anweist, an einem bestimmten Port zu lauschen.

Die Coverage, die den Tester also durch den Test-Lead berichtet wurde, ist nicht das Ergebnis eines umfangreichen Tests. Das Ausführen der Testfälle hatte einen geringen oder im schlimmsten Fall gar keinen Einfluss auf die Change Request Coverage. Der Informationsgewinn aus den vorher errechneten Change Request Coverages ist deshalb schlecht.

Bemerkung 2:
Der Fall, dass in einem CR geänderte Methoden bereits durch die „Base“-Coverage abgedeckt sind, tritt dann ein, wenn der CR Änderungen an Teilen des Programms mit sich bringt, die im Startprozedere zur Ausführung kommen.

Bemerkung 3:
Eine CRC mit einem hohen Anteil an Methoden, die durch die „Base“-Coverage abgedeckt sind, ist niedrig, da – egal welcher Test nach dem Startprozedere ausgeführt wurde – dieser Anteil schon abgedeckt war. Die CRC ist in solchen Fällen ungewollt testunabhängig.

5.1.2 RQ2: Uninteressante Test-Gaps in der Change Request Coverage und deren systematischer Ausschluss

Uninteressante Test-Gaps in der Change Request Coverage verschlechtern die CRC ungewollt. Wird bei einem CR eine niedrige CRC festgestellt, so ist davon auszugehen, dass diesem CR Beachtung geschenkt wird, um ihn besser zu testen. Unwirtschaftlich ist diese Beachtung (die ja Zeit und damit in der Regel auch Geld kostet) dann, wenn die CRC nur wegen uninteressanter Test-Gaps niedrig ist.

In der Studie soll eine Menge von CRs untersucht und für den Tester uninteressante Test-Gaps identifiziert werden. Diese sollen dann kategorisiert und wenn möglich ein Vorgehen zum systematischen Ausschluss von Test-Gaps dieser Kategorie angegeben werden.

5.1.3 RQ3: Auffindung relevanter Test-Gaps mit Hilfe der Change Request Coverage

Während es in **RQ2** um generell uninteressante Test-Gaps geht, die systematisch von der Change Request Coverage ausgeschlossen werden sollen, geht es in Forschungsfrage **RQ3** um die Aufdeckung relevanter Test-Gaps mit Hilfe der CRC.

Gefundene Test-Gaps sollen in der Entwicklung, genauer in der Qualitätssicherung, einen Mehrwert bieten. Das ist dann der Fall, wenn durch die CRC relevante Test-Gaps aufgedeckt werden können. Aus diesem Grund sollten die Ergebnisse, die zu den Change Request Coverages einer CR-Menge errechnet werden, zur Bewertung kompetenten Personen, die den Quelltext der entsprechenden Funktionalität kennen, vorgelegt werden. Diese können dann beurteilen, ob es sich beim aufgedeckten Test-Gap um ein relevantes (testenswertes) handelt. Relevante Test-Gap-Funde können wichtige Testlücken aufdecken, sodass im risikobasierten Testbetrieb nachgetestet werden kann, oder möglicherweise zur Verbesserung eines Testfalls dienen.

5.2 Aufbau der Studie

Um die Studie nachvollziehbar zu machen, wird im Folgenden gezeigt, welche Schritte zu welchem Zeitpunkt an welchen Angriffsflächen getätigt wurden.

5.2.1 Studienobjekt

Untersucht wurden Change Requests aus dem Projekt *Teamscale* (vgl. Grundlagen-Kapitel).

Untersuchtes Projekt

Die Wahl des Studienobjekts fiel aus mehreren Gründen auf Teamscale (Details zu Teamscale im Grundlagen-Kapitel). Die Entwickler von Teamscale arbeiten für die CQSE GmbH, die andere Softwareentwickler bezüglich Qualität und Qualitätsbewahrung in deren Systemen berät. Die Entwickler von Teamscale haben einen hohen Anspruch an sich selbst und bemühen sich, durch einen geschickten Reviewprozess selbst eine höchstmögliche Softwarequalität beizubehalten. Dies ist für die Studie von Vorteil, da davon ausgegangen werden kann, dass Change Requests ausreichend detailliert formuliert sind und durch die vollständige und saubere Benutzung von Kontrollsystemen (vgl. Abschnitt 3.2) der einfache Schluss von CRs auf deren Commits möglich ist.

Des Weiteren kommt der Studie die räumliche Nähe des Firmensitzes der CQSE GmbH zur Fakultät für Informatik der Technischen Universität München zugute. In der **RQ3** werden Entwickler zu von ihnen bearbeiteten CRs befragt. Dank der geringen Entfernung kann diese Befragung persönlich erfolgen.

Untersuchte Change Requests

Die CRs sollten nicht zu alt sein, damit sich in **RQ3** der befragte Entwickler noch an den Inhalt des CRs erinnern kann. Es wurde festgelegt, dass ein ausgewählter CR seine letzte Änderung spätestens am 1. Juni 2014 haben sollte (Zeitpunkt der CR-Auswahl: 8. Februar 2016). Von 1. Juni 2014 bis 1. Februar 2016 wurden zur Entwicklung von Teamscale etwa 2620 Change Requests aktualisiert. Von diesen sind etwa 1460 als „GREEN“ markiert, was die erfolgreiche und abgeschlossene Bearbeitung eines CRs signalisiert.

Im Rahmen der Studie sollten die Ausführungsinformationen des Backends geloggt werden. Folglich wurden CRs vermieden, in denen zum Beispiel Änderungen am Teamscale Eclipse Plugin vorgenommen wurden.

Die Auswahl der zu untersuchenden CRs sollte ähnlich einer stratifizierten Stichprobe ablaufen. Für jeden Entwickler sollte es maximal drei zu untersuchende CRs geben. Die Auswahl der CRs von unterschiedlichen Entwicklern soll eine gefächerte Stichprobe liefern und es vermeiden, dass sich Tendenzen, die für Entwickler spezifisch sind, im Studienergebnis widerspiegeln. Dies könnte insbesondere dann der Fall sein, wenn einzelne Entwickler im Entwerfen von Testfällen besonders gut oder besonders schlecht sind.

Die Auswahl der zu testenden Change Requests erfolgte durch das entwickelte Werkzeug.

Um eine zufällige Auswahl von maximal 3 CRs pro Entwickler zu erhalten, ist eine Funktion in das Werkzeug implementiert, die eine solche Auswahl als Resultat liefert. Das Werkzeug greift auf das Projektmanagementsystem zu, das zur Entwicklung von Teamscale verwendet wird und sucht für jeden angegebenen Entwickler in einem spezifizierten Zeitraum nach grün markierten (also fertig gestellt und geschlossen) CRs. Aus dieser Liste werden nacheinander zufällig CRs ausgewählt und für jeden ausgewählten CR überprüft, ob dieser einen Commit enthält, der eine Java-Datei betrifft. Ist dies der Fall, so ist dieser CR Teil des Resultats. Wenn nicht, wird der CR verworfen und ein anderer an seiner Stelle getestet. Die Zufallsauswahl von CRs stoppt, wenn entweder für einen Entwickler 3 CRs gefunden wurden oder aus der Ursprungliste kein weiterer CR zur Verfügung steht. Doppelte Ausgaben von CRs werden vermieden. Das Resultat wird in kommaseparierten Werten (csv Format) in einer Datei gespeichert.

Nach einer ersten Probeauswahl stellte sich heraus, dass sich nicht jeder Change Request für einen Test eignet. Tests, für die man sich extra in ein kompliziertes, breites Thema zeitintensiv einarbeiten müsste, wurden nicht durchgeführt. Um ersatzweise einen anderen CR für solche CRs zu finden, wurde das Werkzeug zur Change Request Findung insgesamt 3 Mal eingesetzt. CRs aus der ersten Tabelle, die sich unter vertretbarem Aufwand nicht testen ließen, wurden durch den nächstfolgenden CR-Eintrag zu diesem Entwickler aus der weiteren Tabelle ersetzt. Insgesamt wurden 18 CRs aus der ersten Tabelle nicht getestet und durch

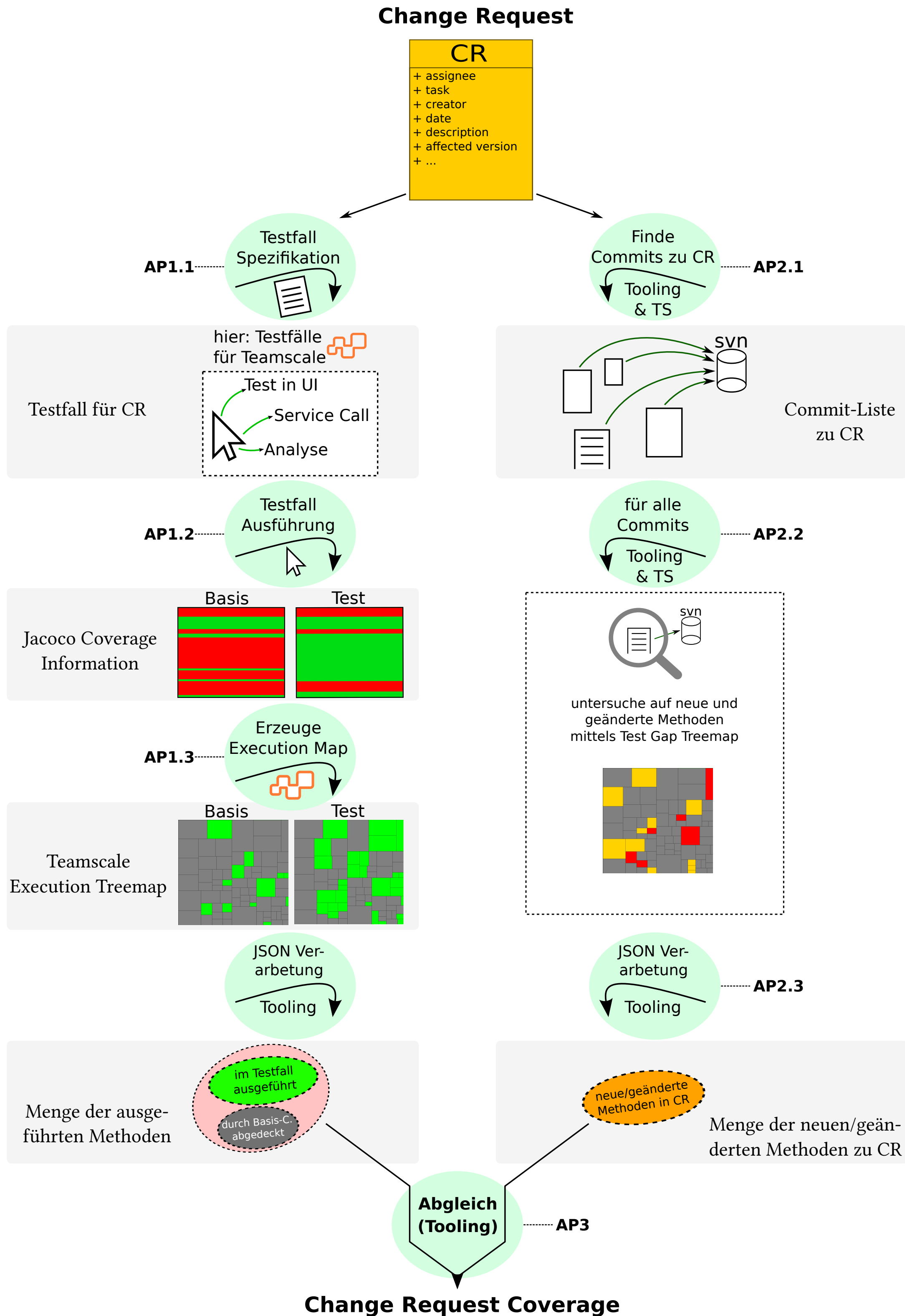


Abbildung 5.1: Schematische Darstellung des Ablaufs vom Change Request bis hin zur berechneten Change Request Coverage. Jeder Pfeil steht für eine Aktion. **AP**: Ankerpunkt, der zur Orientierung in der Grafik dient und auf den sich der Text bezieht.

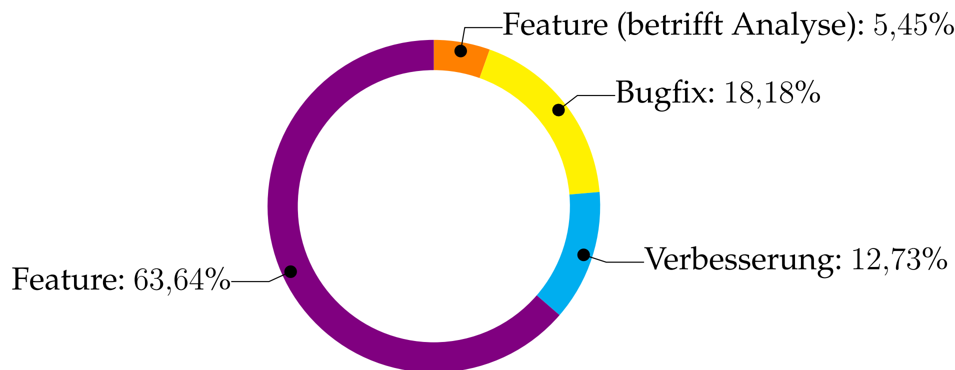


Abbildung 5.2: Die meisten CRs, die im Rahmen der Studie untersucht wurden, führten neue Funktionen in Teamscale ein. Andere CRs verbesserten bestehende Funktionen oder entfernten unerwünschtes Verhalten/Fehler.

andere nach dem obigen System ersetzt sofern noch weitere CRs vom betroffenen Entwickler zur Verfügung standen. Gründe für Ersetzungen waren „Keine Erfahrung mit Custom Checks in Teamscale“, „Noch nie mit ABAP gearbeitet“, „CR wurde vor 3 Jahren hinzugefügt“, „Kein TFS zum Testen verfügbar“, „Nur Plugin Code“, „Noch nie etwas mit DotNet gemacht und insbesondere nie mit Tracefiles dafür gearbeitet“. In Abbildung 5.2 ist eine grobe Übersicht gegeben, wie viele Change Requests mit welchen Arbeitsaufträgen untersucht wurden.

Die ausgewählten 54 CRs sind von verschiedenster Natur (Bugfix, neues Feature etc.) und verschiedenster Größe. In 5 Fällen wurde keine einzige Java-Methode (ausgeschlossen ist Testcode) verändert. Im CR mit den meisten neuen/geänderten Methoden betrug die Anzahl 109 Java Methoden. Im Durchschnitt wuren 9,48 Methoden pro CR verändert.

5.2.2 Design der Studie

In diesem Abschnitt wird vorgestellt, wie man von den aus der Studie gewonnenen Daten zu einer Beantwortung der Forschungsfragen gelangt.

RQ1 Für die erste Forschungsfrage „Wie viel Change Request Coverage ist bei manuellen Tests unabhängig vom eigentlichen Test und damit aussagelos?“ wird festgestellt, wie groß der Anteil in Prozent der Change Request Coverage ist, der testunabhängig festgestellt wurde. Testunabhängiger Anteil bedeutet, dass die Coverageinformation nicht vom definierten Testfall und dessen Ausführung rührt, sondern schon vom Start des Programms unter Test (Teamscale).

Die Beantwortung der Forschungsfrage **RQ1** ergibt sich aus der Analyse der Anteilswerte.

RQ2 Für die zweite Forschungsfrage „Warum sind manche Test-Gaps für Tester uninteressant und können wir sie systematisch ausschließen?“ werden die gefundenen Test-Gaps durch eine individuelle Inspektion durch den Autor der

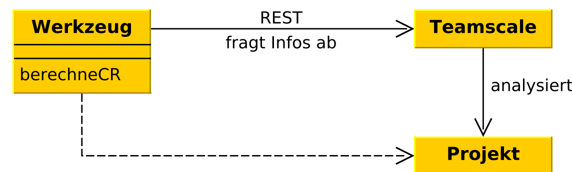


Abbildung 5.3: Die Beziehung von Werkzeug, Teamscale und Projekt unter Test im Modell dargestellt. REST (Representational State Transfer): Schnittstelle zum Teamscale Server.

Studie untersucht. Gibt es Anzeichen dafür, dass das Test-Gap nicht testenswert ist? Wenn ja, werden sie kategorisiert.

Die Beantwortung der Forschungsfrage **RQ2** besteht aus der Vorstellung der Kategorien, deren Anteil an der CRC und wenn möglich der Angabe dazu passender Verfahren zum Auschluss der darin enthaltenen Methoden.

RQ3 Für die dritte Forschungsfrage „Finden wir aus Sicht des Entwicklers/-des Testers relevante Test-Gaps?" werden dem jeweiligen Assignee eines CRs die Ergbnisse zu der zugehörigen Change Request Coverage gegeben. Der Entwickler beurteilt die gefundenen Test-Gaps als relevant (testenswert) oder als nicht relevant (nicht testenswert).

Zur Beantwortung der Forschungsfrage **RQ3** wird überprüft, wie hoch der Anteil interessanter Test-Gaps ist (pro CR und insgesamt). Außerdem werden aufgeführte Gründe kategorisiert und dargestellt. Es wird überprüft, ob die Einteilung aus RQ2 mit den hier durch den Entwickler gemachten Angaben übereinstimmt.

5.2.3 Methodik

In diesem Abschnitt wird zunächst beschrieben, wie das Setup allgemein vorbereitet wurde. In eigenen Absätzen zu jeder Forschungsfrage wird im Anschluss daran das weitere Vorgehen beschrieben.

Bestimmung des Changesets eines CRs Der in diesem Paragraphen beschriebene strukturelle Ablauf ist in Abbildung 5.1 dargestellt; die im Text erwähnten Ankerpunkte (AP) helfen, den richtigen Ort in der Grafik zuordnen zu können. Außerdem bildet das Modell in Abbildung 5.3 die Beziehung von Werkzeug, Teamscale und dem Projekt unter Test ab.

Zu einem angegebenen CR bestimmt das für die Studie entwickelte Werkzeug die Menge der im Rahmen des CRs geänderten/hinzugefügten Methoden. Es ruft dazu von einer Teamscale Instanz, die das zum CR zugehörige Projekt analysiert, Informationen zum CR ab. Diese Informationen beinhalten eine Liste der dazugehörigen Commits und deren Zeitstempel (in der Grafik: *AP2.1*). Für jeden Commit wird eine Test-Gap Treemap ohne Coverage Information erstellt (für eine Test-Gap Treemap vgl. auch Abb. 1.2), die die Unterschiede von einer Millisekunde vor dem Zeitstempel des CRs und einer Millisekunde danach berücksichtigt

(*AP2.2*). In der Treemap sind hinzugefügte Methoden rot und geänderte Methoden orange markiert. Das Werkzeug wertet das JSON (JavaScript Object Notation, Ausgabeformat von Teamscale) der Test-Gap Treemap aus und erstellt daraus eine Liste der geänderten Methoden, die auch Informationen enthält, wo sich die Methode im Quelltext befindet, wann sie zuletzt bearbeitet wurde, ob sie hinzugefügt oder von einer bestehenden Version verändert wurde (*AP2.3*). Ist dies für jeden Commit, der einem CR zugeordnet werden konnte durchgeführt, so integriert das Werkzeug die Informationen zu einer Liste, die nicht mehr spezifisch für einen Commit ist, sondern für einen Change Request gilt. Die resultierende Liste ist also das Changeset des CRs (vgl. auch Abschnitte 3.1 und 3.2).

Testfälle Für jeden der ausgewählten Change Requests wurde ein Testfall verfasst (in Abb. 5.1 Ankerpunkt *AP1.1*). Der Testfall soll auf die im Change Request funktionell spezifizierte Änderung ausgerichtet sein. Da es für einen Außenstehenden schwierig ist einen guten, angepassten Testfall zu schreiben, wurden die verfassten Testfälle an die jeweiligen Assignees der CRs verschickt, mit der Bitte die Testfälle zu validieren oder zu verbessern. In den Antworten machte sich bemerkbar, dass ohne diesen Schritt die Qualität der Tests weitaus geringer ausgefallen wäre.

Allgemeine Vorbereitung des Testsetups Wie bereits in Abschnitt 4.2 beschrieben, unterstützt das entwickelte Werkzeug kein „Methodentracking“. Würde man also für jeden CR die Berechung der CRC mit einem Test auf der HEAD-Revision durchführen, wären viele Methoden, die im CR unter Test geändert oder hinzugefügt wurden nicht mehr an dem Ort, der im Changeset des CRs festgehalten ist. Eine detaillierte Beschreibung des Problems findet sich auch im Anhang auf Seite 58. Um dieses Problem zu vermeiden, wurde sich dazu entschlossen, für jeden CR eine passende Revision von Teamscale auszuchecken und auf dieser zu testen. So ist die Wahrscheinlichkeit hoch, dass die Methoden noch am richtigen Ort sind. Als auszucheckende Revision wurde die letzte Revision des Tages gewählt, an dem der CR letztmalig aktualisiert wurde. Meist war die letzte einen CR betreffende Aktivität die grüne Markierung (CR abgeschlossen), die im Rahmen des Review-Prozesses vergeben wird. [1]

Die Teamscale Entwickler betreiben eine eigene Teamscale Instanz, die Teamscale selbst analysiert und die entsprechende Entwicklungshistorie bereithält. Aus dieser Teamscale Instanz wurden für jeden CR die passenden Revisionsnummern nachgeschlagen, um für den Testvorgang die richtigen Versionen auschecken zu können.

Die Aufnahme der Ausführungsinformation im Test erfolgte durch den Java-Profiler Jacoco. Dieser wurde im Server Modus ausgeführt und die benötigten Informationen wurden durch ein Java Programm abgeholt. Hintergrundinformationen, warum Jacoco im Server-Modus ausgeführt wurde, befinden sich im Anhang auf Seite 59.

1 Es wurde nach der Studie festgestellt, dass vom letzten Commit bis zur letzten Aktualisierung eines CR gelegentlich eine geraume Zeit vergeht. Der Graph in Abbildung C.1 (Seite 60) bildet diesen Zeitunterschied ab und verknüpft ihn mit der Anzahl der Methoden, die zum Testzeitpunkt nicht mehr existierten, aber im Rahmen des CRs hinzugefügt oder verändert wurden.

Für alle 54 CRs wurde die entsprechende Teamscaleversion kompiliert. Für jeden CR war im Testfall definiert, ob Teamscale von einem gespeicherten Store (Stand) zu öffnen ist oder Teamscale ohne vorangegangene Analyse oder ähnliches verwendet werden soll (zum Beispiel, wenn die Änderung des CRs eine Analyseeinheit oder eine Logeinheit betrifft). Im ersten Fall wurde, ohne Coverageinformationen aufzunehmen, der Store dem Testfall entsprechend vorbereitet. Dies beinhaltete in den meisten Fällen die Analyse eines Projekts oder aber auch das Setzen der Mailkonfiguration für Benachrichtigungs E-Mails. Im zweiten Fall wurde Teamscale auch einmal ohne das Aufzeichnen von Coverage gestartet, um einen Store zu erzeugen (allerdings ohne weiteren Inhalt). Danach wurde Teamscale wieder beendet.

Erzeugung der Basis Coverage Mit dem entsprechend der Testfallspezifikation vorbereiteten Store wurde Teamscale gestartet. Dieses Mal wurde Jacoco als Javaagent an die JVM übergeben, um so Ausführungsinformationen aufzuzeichnen. Die Basis Coverage enthält in jedem Fall den Start von Teamscale und den dreimaligen (Teamscale kompiliert beim ersten Mal die JavaScript-Dateien) Aufruf der Startseite von Teamscale.

Nachdem dieses Startprozedere ausgeführt wurde, kam ein Java Programm zum Einsatz, das die bisher aufgenommene Ausführungsinformation vom Jacoco-Server lud und auf der Festplatte ablegte.[2] Damit endete die Erzeugung der Basis Coverage. Teamscale wurde währenddessen nicht unterbrochen.

Hinweis: Die Erzeugung der Basis Coverage könnte im Falle von Teamscale automatisiert werden, sodass kein manuelles Eingreifen von Nöten ist. Folgende Schritte müssen dabei automatisiert werden: Start des Systems mit Jacoco, Ausführen des Startszenarios und das Holen der Coverageinformationen. In der Studie wurde darauf verzichtet, da der Aufwand der Automatisierung den Nutzen überstiegen hätte.

Erzeugung der Testfall Coverage Auf dem laufenden Teamscale wurde anschließend der definierte Testfall ausgeführt. Beispiele für Testfälle sind

- die Erzeugung eines Widgets auf einem Dashboard,
- die inkrementelle Analyse eines Projekts, sodass Benachrichtigungen an den User versendet werden oder
- der Userimport von einem LDAP-Server.

Nach erfolgreicher Ausführung des Testfalls wurde das Java Programm zum Abruf der Coverageinformation ausgeführt. Die Coverage, die nun vom Jacoco-Server geladen wurde, enthielt sowohl Basis- (oder Start-), als auch Testfallcoverage.[2] Das Teamscale unter Test konnte danach beendet werden.

Erstellung eines Bewertungs-Fragebogens zur CRC Die Coveragedateien, die vom Jacoco Server geladen wurden, haben ein binäres Format. Ein Skript, das man mit Jacoco herunterlädt, hält die Möglichkeit bereit, die Dateien in das

2 Die Ausführung des gesamten Tests (Startprozedere und Test) ist in in Abb. 5.1, Ankerpunkt *AP1.2* visualisiert.

XML-Format zu konvertieren. Mit diesem Format wiederum kann Teamscale umgehen. Einer lokalen Teamscale Instanz, die das eben ausgeführte Teamscale unter Test analysiert (vgl. Abb. 5.3), werden die XML-Coverage-Dateien übergeben, die gerade erzeugt wurden. Den Coveragedaten kann eine sogenannte „coverage source" zugewiesen werden. Die Ausführungsinformation, die die „Base"-Coverage enthält, bekam in der Regel die „coverage source" `CR1234BASE` und der „Testfall"-Coverage wird der „coverage source" `CR1234TEST` zugewiesen (wobei `1234` für die jeweilige Nummer des CRs steht, der gerade getestet wird). Beim Erstellen von Test-Gap und Execution Treemaps in Teamscale kann angeben werden, aus welcher „coverage source" (das können 0 bis alle verfügbaren Coverage-Quellen sein; im Fall der Studie war es aber immer nur ein einzelnes) die Coverageinformationen geladen werden sollen. Die lokale Teamscale Instanz kann nun Execution Treemaps erzeugen, die genau zeigen, welcher Anteil des Programms im Startszenario ausgeführt wurde und welcher Teil durch Startszenario und Testfall (in Abb. 5.1 Ankerpunkt *AP1.3*).

Damit sind die nötigen Eingangsdaten für das entwickelte Werkzeug (siehe Kapitel 4) komplett, um einen Fragebogen zur Change Request Coverage zu erstellen. Darauf wird festgehalten, welche Methoden im Testfall ausgeführt oder welche schon beim Startprozedere ausgeführt wurden, welche Methoden nicht ausgeführt wurden und welche Methoden im Verlauf des CRs geändert/hinzugefügt wurden, aber zum Zeitpunkt des Tests nicht mehr existieren. Außerdem wird auf dem Fragebogen der benutzte Testfall für den CR angegeben und die Frage gestellt, ob dieser korrekt und vollständig ist. Für jede gefundene Test-Gap-Methode gibt es darauf die Möglichkeit anzukreuzen, ob ein Test-Gap interessant (testenswert) ist oder nicht (nicht testenswert). Am Ende des Fragebogen werden außerdem Prozentwerte zur Change Request Coverage des CRs angegeben.

In Abbildung 5.1 (S. 23) ist die Erstellung des Fragebogens nicht ausdrücklich aufgeführt. Sie findet zeitlich in Ankerpunkt 3 statt. Ein Beispiel für einen solchen generierten Bogen findet sich im Anhang ab Seite 55.

RQ1 Das Werkzeug berechnet für jeden Change Request die Anzahl der Methoden, die im Rahmen des CRs hinzugefügt/geändert und im Startszenario (also nicht explizit durch den Test) aufgerufen wurden.

Aus der absoluten Anzahl wurde der relative Anteil berechnet. Die Ergebnisse werden sowohl für jeden einzelnen CR in Form von Coverage „Ampeln" (siehe Abb. 3.3(b), S. 14) dargestellt, als auch in einem Boxplot, der die Anteilsverteilung über alle CRs zeigt.

RQ2 Alle entdeckten Test-Gap-Methoden wurden in einer Tabelle gesammelt. Für jede Methode wurde durch den Autor beurteilt, ob es sich potentiell um ein wichtiges Test-Gap handelt. Wenn ja, wurde versucht der Methode eine Kategorie zuzuweisen.

Für jede Kategorie wurde anschließend wenn möglich eine Option zum Ausschluss angegeben.

RQ3 Für die **RQ2** wurde jeder Test-Gap im Rahmen der Studie als potentiell wichtig oder potentiell unwichtig eingestuft. Diese Angabe wurde durch entsprechendes Ausstreichen auf den generierten Fragebögen ergänzt. So stand vor jeder Bewertung eines Test-Gaps entweder „In another part of this study the method got already categorized as not worth testing.“ oder „In another part of this study the method got already categorized as ~~not~~ worth testing.“

Die Assignees der CRs bekamen die jeweiligen CRC Bögen per e-Mail zugesandt und wurden gebeten den ausgefüllten Bogen entweder ausgedruckt oder per e-Mail wieder zurückzusenden. Mit einigen Entwicklern wurde der Fragebogen persönlich besprochen.

Mit Hilfe einer Tabelle wurde ausgewertet, wie hoch der Anteil von als interessant bewerteten Testfällen ist (pro CR und insgesamt) und ob die von den Entwicklern gemachte Aussage sich mit der Bewertung aus **RQ2** deckt. Begründungen, die von den Entwicklern angegeben wurden, wurden kategorisiert und dargestellt.

5.3 Ergebnisse

Die Ergebnisse zu den festgestellen Ausführungsanteilen finden sich in Tabelle 5.1 sowie in den Abbildungen 5.5 und 5.7 dargestellt.

In der Tabelle wird für jeden CR aufgelistet, wie viele Methoden im CR geändert/hinzugefügt wurden, wie viele Methoden ausgeführt beziehungsweise nicht ausgeführt wurden und wie viele exklusiv durch den definierten Testfall und wie viele bereits durch das Startszenario abgedeckt wurden.

Abbildung 5.7 gibt eine Übersicht, wie groß die Anteile an nicht ausgeführten, im Startszenario ausgeführten und im Testfall ausgeführten Methoden sind. Sortiert ist von links nach rechts und von oben nach unten. Die leeren Balken zu Anfang stehen für CRs, die keine Änderung einer Java Methode außerhalb des Testcodes erfahren haben (dieser kann durch manuelle Tests im Setup der Studie nicht abgedeckt werden). Dies ist auch nach der in 5.2.1 beschriebenen Auswahl an CRs möglich. Denn zur Auswahl der CRs wurde lediglich das Kriterium „CR beinhaltet Commit, der eine Java Datei betrifft“ verwendet, die Berechnung der Test-Gaps benötigt allerdings mindestens eine Änderung an einer Java Methode außerhalb des Testcodes. In einem Fall wurde eine Java Datei nur im Header (hier kein funktionaler Quelltext) verändert. In anderen Fällen gab es nur Commits zu Testcode. Dies kann bei Teamscale insbesondere dann der Fall sein, wenn Änderungen am JavaScript Teil der UI gemacht wurden, die von einem Java JUnit Test getestet werden sollen.

Nach den leeren Kästen ist die Sortierreihenfolge diese: Gesamtabdeckung (Basis und Test) aufsteigend, Abdeckung durch Basis absteigend und CR Nummer aufsteigend.

Für jeden CRC wurde ein Prozentwert errechnet, der beschreibt, welcher Anteil der Methoden ausgeführt wurde. Die Methoden, die durch die Base Coverage abdeckt waren, wurden bei der Berechnung unterschiedlich berücksichtigt. So ging es in der ersten Art der Berechnung nur darum, ob eine Methode gecovert wurde

Tabelle 5.1: Ergebnisse zu erhobenen Daten für die Change Requests

CR	changedOrAdded	notExecuted	executed	executedInTest	executedInBase	intermediate	executed%	executedBase%	executedTest%	notExecuted%
5517	7	0	7	7	0	3	100,0	0,0	100,0	0,0
5934	22	11	11	9	2	8	50,0	9,1	40,9	50,0
6141	2	0	2	0	2	0	100,0	100,0	0,0	0,0
6528	0	0	0	0	0	1	-	-	-	-
6552	5	0	9	9	0	2	180,0	0,0	180,0	0,0
6589	3	0	3	3	0	1	100,0	0,0	100,0	0,0
6643	2	0	2	2	0	0	100,0	0,0	100,0	0,0
6708	6	2	4	4	0	0	66,7	0,0	66,7	33,3
6719	7	2	5	5	0	0	71,4	0,0	71,4	28,6
6773	13	6	7	6	1	0	53,8	7,7	46,2	46,2
6856	0	0	0	0	0	0	-	-	-	-
6946	16	1	15	11	4	0	93,8	25,0	68,8	6,3
6981	4	0	4	4	0	0	100,0	0,0	100,0	0,0
7012	2	0	2	1	1	3	100,0	50,0	50,0	0,0
7026	20	3	17	15	2	8	85,0	10,0	75,0	15,0
7112	2	1	1	1	0	0	50,0	0,0	50,0	50,0
7182	6	1	5	5	0	0	83,3	0,0	83,3	16,7
7212	0	0	0	0	0	0	-	-	-	-
7258	1	0	1	0	1	0	100,0	100,0	0,0	0,0
7286	109	7	102	102	0	6	93,6	0,0	93,6	6,4
7328	9	1	8	8	0	1	88,9	0,0	88,9	11,1
7409	16	5	11	7	4	1	68,8	25,0	43,8	31,3
7426	14	7	7	5	2	30	50,0	14,3	35,7	50,0
7473	5	1	4	4	0	12	80,0	0,0	80,0	20,0
7511	3	2	1	1	0	1	33,3	0,0	33,3	66,7
7619	3	0	3	3	0	0	100,0	0,0	100,0	0,0
7713	10	1	9	9	0	1	90,0	0,0	90,0	10,0
7801	2	0	2	0	2	0	100,0	100,0	0,0	0,0
7873	4	0	4	4	0	0	100,0	0,0	100,0	0,0
8028	50	20	30	27	3	27	60,0	6,0	54,0	40,0
8030	4	1	3	3	0	0	75,0	0,0	75,0	25,0
8043	13	1	12	9	3	3	92,3	23,1	69,2	7,7
8090	16	5	11	8	3	14	68,8	18,8	50,0	31,3
8171	7	2	5	2	3	1	71,4	42,9	28,6	28,6
8202	3	0	3	2	1	0	100,0	33,3	66,7	0,0
8220	17	2	15	15	0	6	88,2	0,0	88,2	11,8
8345	2	0	2	2	0	0	100,0	0,0	100,0	0,0
8358	4	0	4	4	0	3	100,0	0,0	100,0	0,0
8449	0	0	0	0	0	13	-	-	-	-
8497	7	0	7	7	0	25	100,0	0,0	100,0	0,0
8579	1	0	1	1	0	3	100,0	0,0	100,0	0,0
8596	3	1	2	2	0	0	66,7	0,0	66,7	33,3
8630	14	1	13	13	0	1	92,9	0,0	92,9	7,1
8638	5	0	5	4	1	28	100,0	20,0	80,0	0,0
8641	6	0	6	6	0	0	100,0	0,0	100,0	0,0
8649	4	0	4	4	0	1	100,0	0,0	100,0	0,0
8719	4	1	3	3	0	4	75,0	0,0	75,0	25,0
8737	14	5	9	9	0	1	64,3	0,0	64,3	35,7
8816	2	0	2	1	1	0	100,0	50,0	50,0	0,0
8848	5	5	0	0	0	1	0,0	0,0	0,0	100,0
8852	4	0	4	4	0	1	100,0	0,0	100,0	0,0
8926	6	0	6	5	1	1	100,0	16,7	83,3	0,0
8964	27	15	12	12	0	0	44,4	0,0	44,4	55,6
9236	0	0	0	0	0	0	-	-	-	-

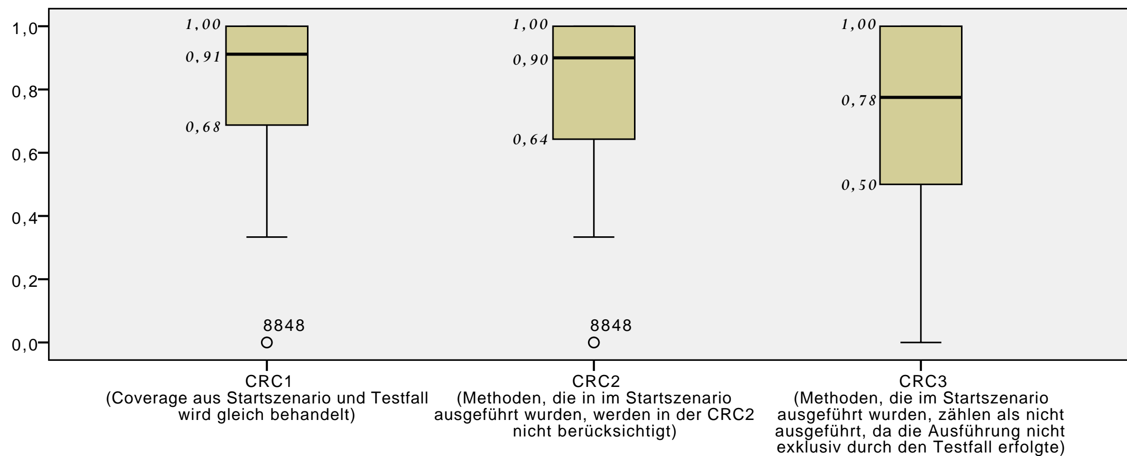


Abbildung 5.4: Für jeden CRC wurde der Prozentwert ermittelt, welcher Anteil des Changesets ausgeführt wurde. Dabei wurden, wie in Abschnitt 4.1 beschrieben, die Werte unter unterschiedlicher Berücksichtigung der Methoden, die durch das Startszenario ausgeführt wurden, berechnet. In dieser Abbildung geben Boxplots Aufschluss über die Verteilung der Werte der unterschiedlichen CRC Berechnungen.

oder nicht. Es spielte keine Rolle, ob dies durch den Testfall oder das Startprozedere geschah. In der zweiten Berechnung wurden die Methoden aus der Base Coverage von der Berechnung ausgeschlossen (sowohl aus dem Changeset als auch aus dem Set der ausgeführten Methoden). Bei der dritten und letzten Berechnung wirken sich Methoden, die schon im Startszenario ausgeführt werden, negativ auf die CRC aus. Die unterschiedlichen Arten der Berechnung wurden auch in Sektion 4.1 dargestellt und die Ergebnisse werden in Abbildung 5.4 präsentiert.

Auffällig hierbei ist CR#8848, der zu 0% abgedeckt ist. Dieser CR verbesserte die SVN Verbindungseigenschaften von Teamscale. Nach einer Verbindungsstörung, die zu einer unreinen SVN working copy führt, soll Teamscale diese Situation erkennen und ein entsprechendes Cleanup durchführen. Im Testfall wurde festgelegt, dass ein neues Projekt zur Analyse in Teamscale angelegt werden soll, das den SVN Connector verwendet. Die Verbindungsstörung sollte dadurch hervorgerufen werden, dass die Netzwerkverbindung des Rechners für einige Zeit getrennt wurde. Der Testfall war leider nicht ausreichend, um eine unreine SVN Kopie zu provozieren und den CR adäquat zu testen.

Nicht besonders auffällig in der Übersicht, jedoch auffällig in der Tabelle, in der auch die absoluten Zahlen gelistet sind, ist CR#7286. Er ist von allen getesteten CRs der mit der größten Menge an neu erstellten beziehungsweise geänderten Methoden. Das Changeset des CRs hat eine Mächtigkeit von 109 Methoden, von denen nur 7 nicht ausgeführt wurden.

5.3.1 Gewichtung mit Methodenlänge

Die bisher beschriebenen Anteile, wurden durch die ungewichtete Anzahl von Methoden errechnet. Geht man davon aus, dass die Fehlerwahrscheinlichkeit in

einer Methode mit ihrer Länge steigt, so ist es sinnvoll, die Anzahl nach den jeweiligen Methodenlängen zu gewichten. Das bedeutet, dass die fehlende Abdeckung einer kurzen Methode einen kleineren Einfluss auf die CRC hat als die fehlende Abdeckung einer langen Methode.

Neben den beschriebenen Darstellungen ohne Gewichtung sind auch Darstellungen mit Gewichtung abgedruckt. So können die Ausführungsanteile pro CR in Abbildung 5.8 und Informationen zu den Ausführungsanteilen über alle getesten CRs in Abbildung 5.6 betrachtet werden.

Im – in Abbildung 5.6 gezeigten – Boxplot, ist die höchste Abweichung in den betrachteten Quartilen leicht zu erkennen: Bei den nicht ausgeführten Methoden liegt das obere Quartil $Q_{0,75}$ wenn die Methoden gewichtet werden lediglich bei 21%, im Vergleich zu 31%, wenn die Methoden nicht gewichtet werden. Alle anderen berechneten Quartile der Ausführungsanteile unterschieden sich um maximal 5%, wie der Vergleich der Tabellen 5.2 und 5.3 ergibt.

5.3.2 RQ1

Für die Forschungsfrage **RQ1** sind besonders jene Methoden interessant, die durch die „Base"-Coverage abgedeckt wurden. Außerdem ist für die Beantwortung dieser Frage der Anteil der durch die „Base"-Coverage über alle getesteten CRs von Belang.

Die Ergebnisse zur **RQ1** werden zunächst für einzelne CRs und dann über alle CRs hinweg betrachtet.

Betrachtung einzelner CRs und Beispielfälle In der Übersicht (Abb. 5.7) der Ausführungsanteile sind die vollständig grau ausgefüllten Balken diejenigen mit der höchsten und höchstmöglichen (100%) Abdeckung durch die „Base"-Coverage. *Diese drei CRs CR#6141, CR#7258 und CR#7801, sowie die beiden CRs CR#7012 und CR#8816 mit 50% Abdeckung durch die „Base"-Coverage, werden nun betrachtet und der Hintergrund identifiziert, warum der „Base"-Coverage-Anteil so hoch ist.*

In **CR#6141** wurde der in JavaScript geschriebene Editor für Softwarearchitekturen verändert. Die zwei geänderten Java Methoden werden beide beim Start ausgeführt. Die Aufgabe der einen ist die Registrierung von JavaScript Modulen und die der anderen ist die Erzeugung von JavaScript Dateien. Der komplette JavaScript Anteil von Teamscale wird beim ersten Aufruf von Teamscale kompiliert, gespeichert und an den Browser übergeben. Aus diesem Grund ist CR#6141 komplett testunabhängig gecovert.

CR#7258 macht es einem Adminstrator von Teamscale möglich, auf die Dashboardkonfigurationen aller registrierten Teamscalenutzer zuzugreifen. Dafür wurde eine Java Methode verändert, die die Dashboard auflistet. Da das Startszenario so definiert ist, dass die Dashboardseite von Teamscale drei mal aufgerufen wird (vgl. dazu 5.2.3), ist dieser CR bereits vollständig gecovert, bevor der Testfall zur Ausführung kommt.

Vor der Erledigung des **CR#7801** wurde Teamscalebenutzern, für die kein Projekt konfiguriert war, angezeigt, dass sie zum Zweck einer Projektkonfiguration den

Teamscaleadministrator kontaktieren sollten. In CR#7801 wurde dies um Kontaktinformationen zum Administrator erweitert. Dafür wurden zwei Java Methoden verändert. Beide sind grundlegend für die Anzeige einer sogenannten Teamscale Perspektive. Jede Unterseite von Teamscale ist als eine Perspektive auffassbar, also auch die Dashboardseite, die im Startszenario geladen wird. Damit ist auch die vollständige Abdeckung durch die Basis-Coverage im Fall des CR#7801 geklärt.

In Teamscale gibt es einige Konfigurationsformulare. **CR#7012** war dafür zuständig, dass eine Liste, die einem Konfigurationsparameter übergeben werden soll, nicht mehr nur durch Kommata, sondern auch durch Zeilenumbrüche separiert sein kann. Zudem sollte es möglich sein, beide Seperatoren in beliebigem Wechselspiel zu verwenden. Dafür wurden zwei Java Methoden verändert. Eine davon wurde testunabhängig ausgeführt. Diese Methode wird unter anderem aufgerufen, um verschiedene Standardkonfigurationen zu erzeugen. Von welchem Ursprung die Methode aufgerufen wurde, ist aus den vorliegenden Daten nicht zu lesen. Es sei an dieser Stelle angemerkt, dass bei Methoden, die durch die Base Coverage abgedeckt sind, keine Aussage darüber getroffen werden kann, ob die Methoden *auch* im Testfall abgedeckt sind oder nicht.

Teamscale kann mit einem Issue Tracker verbunden werden, die vorhandenen Issues importieren und mit der Versionshistorie verknüpfen. In **CR#8816** wurde eine Funktion eingebaut, um alle bekannten Issue Nummern mit einem einzelnen Service Call abzurufen. Dafür wurden zwei Java Methoden geändert. Die eine Methode enthält die Servicehandlung an sich und die andere registriert den Service. Letztere wurde bereits durch die Base Coverage abgedeckt. Diese Registrierung erfolgt sehr früh im Programmablauf und ist nicht durch die Startszenariodefinition zu beeinflussen.

Bemerkung 4:
Bei der Untersuchung der fünf genannten CRs war auffällig, dass alle kein großes Changeset aufweisen. In CR#6141, CR#7801, CR#7012 und CR#8816 änderten sich zwei Methoden, in CR#7258 sogar nur eine.

Betrachtung über alle getesteten CRs Statistische Rahmendaten der Ausführungsaufteilung (Base Coverage, Testfall Coverage, nicht getestete Anteil) über alle CRs sind in der Tabelle 5.2 gelistet und im Boxplot in Abb. 5.5 dargestellt.

In der Hälfte der Fälle wurde eine Abdeckung des Changesets alleine durch den Test (also nicht schon im Startprozedere) von mindestens 75% erreicht. In 75% der Fälle ergab sich immerhin noch eine Abdeckung von 50% alleine durch den Testfall.

Die Abdeckung durch die „Base“-Coverage war hingegen in der Regel gering. 50% der Fälle weisen gar keine Abdeckung durch die „Base“-Coverage auf. In 75% der Fälle war für die Abdeckung durch die „Base“-Coverage kein größerer Wert als 18% feststellbar.

Die Tabelle 5.3 und die Abbildung 5.6 zeigen die Ergebnisse in einer Variation: Hier werden die Methoden mit ihren Längen gewichtet. Der Median für den Anteil des Changesets, der exklusiv durch den Test ausgeführt wurde, liegt nun bei

83% (ohne Gewichtung: 75%). Auch hier weisen 50% der Fälle gar keine Abweichung durch die „Base“-Coverage auf.

5.3.3 RQ2

Die nachfolgende Kategorisierung erfolgte durch den Autor der vorliegenden Studie. Bei der Untersuchung der getesteten CRs wurden folgende Beobachtungen gemacht.

- 13 Methoden stellten sich als einfache „Getter“ heraus. „Einfach“ bedeutet hier, dass in der Methode keine weitere Logik vorhanden ist. Die einfache getter-Methode mit dem Namen `getMember` bekommt also keine Parameter übergeben und besteht lediglich aus der Anweisung, den Wert der Klassenvariable `member` zurückzugeben. Beispiele für solche Methoden sind:

```
1 public String getUpdatedBy() {
2     return updatedBy;
3 }
```

```
1 public String getMessage() {
2     return message;
3 }
```

```
1 public int getDependencyLocation() {
2     return dependencyLocation;
3 }
```

- 12 Methoden wurden (wieder durch den Autor der Studie) als „too trivial to test“ bewertet. Hier war die Anzahl an Anweisungen ausschlaggebend, die in der Methode zur Ausführung kommen. Beispiele für „too trivial to test“ Methoden sind:

```
1 public ParseLogIndexEntry(ParseLogIndexEntry entry) {
2     this.message = entry.message;
3 }
```

```
1 public AnalysisTooComplicatedException(String message) {
2     super(message);
3 }
```

```
1 protected boolean shouldStripTrailingSlashFromUrl() {
2     return true;
3 }
```

```
1 public GitRepositoryConnectorDescriptor(String url) {
2     this();
3     this.repositoryUrl = url;
4 }
```

- 4 weitere Methoden waren Implementationen der toString-Methode

```
1 public String toString() {
2     return "State [valueState=" + valueState + ", feasibilityState="
3             + feasibilityState + ", predecessor=" + predecessor + "]";
4 }
```

```
1 public String toString() {
2     return dependencyUniformPath + ", " + dependencyLocationUniformPath
3             + ", " + dependencyLocations;
4 }
```

Die Tabelle 5.4 zeigt die Aufteilung der Methoden in die gefundenen Kategorien.

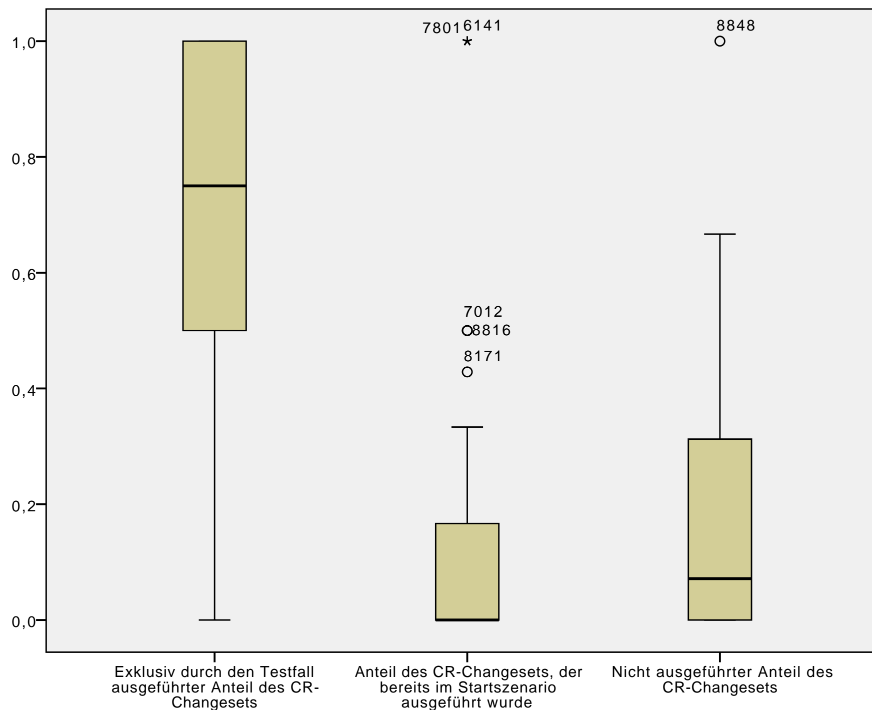


Abbildung 5.5: Anteile (Anzahlen von Methoden) der Changesets, die im Test oder im Initialisierungsvorgang durchlaufen oder gar nicht ausgeführt wurden.

Tabelle 5.2: Deskriptive Statistik zu den Ausführungsverteilungen in Base-Coverage, Testfall-Coverage und unterbliebene Ausführung.

Exklusiv durch den Test ausgeführt	Unteres Quartil $Q_{0,25}$	0,50
	Median	0,75
	Oberes Quartil $Q_{0,75}$	1
	Mittelwert	0,695
	Varianz / Standardabweichung	0,091 / 0,302
Abgedeckt durch die „Base“-Coverage	Unteres Quartil $Q_{0,25}$	0,00
	Median	0,00
	Oberes Quartil $Q_{0,75}$	0,18
	Mittelwert	0,133
	Varianz / Standardabweichung	0,068 / 0,260
Nicht ausgeführt	Unteres Quartil $Q_{0,25}$	0,00
	Median	0,07
	Oberes Quartil $Q_{0,75}$	0,31
	Mittelwert	0,180
	Varianz / Standardabweichung	0,050 / 0,224

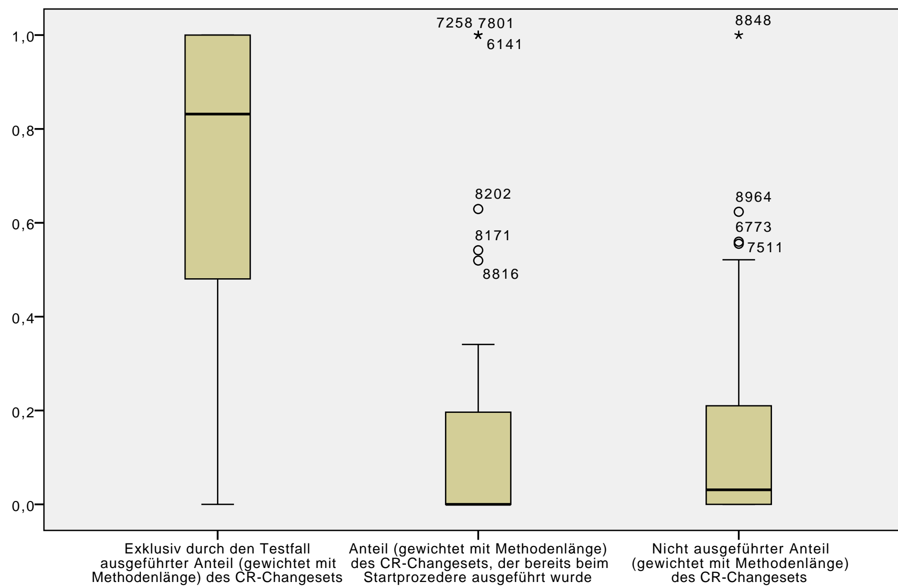


Abbildung 5.6: Anteile (Anzahlen von Methoden je gewichtet mit den Methodenlängen) der Changesets, die im Test oder im Initialisierungsvorgang durchlaufen oder gar nicht ausgeführt wurden.

Tabelle 5.3: Deskriptive Statistik zu den Ausführungsverteilungen (Methoden mit ihren Längen gewichtet) in Base-Coverage, Testfall-Coverage und unterbliebene Ausführung.

Exklusiv durch den Test ausgeführt	Unteres Quartil $Q_{0,25}$	0,48
	Median	0,83
	Oberes Quartil $Q_{0,75}$	1
	Mittelwert	0,713
	Varianz / Standardabweichung	0,831 / 0,319
Abgedeckt durch die „Base“-Coverage	Unteres Quartil $Q_{0,25}$	0,00
	Median	0,00
	Oberes Quartil $Q_{0,75}$	0,22
	Mittelwert	0,147
	Varianz / Standardabweichung	0,074 / 0,271
Nicht ausgeführt	Unteres Quartil $Q_{0,25}$	0,00
	Median	0,03
	Oberes Quartil $Q_{0,75}$	0,21
	Mittelwert	0,141
	Varianz / Standardabweichung	0,046 / 0,215

Abbildung 5.7: Jedes Rechteck steht für einen CR. In Farben sind die relativen Anteile der Anzahl der Methoden des Changesets dargestellt, die im Testfall (grün) oder im Startprozedere (grau) ausgeführt bzw. nicht ausgeführt wurden (rot).

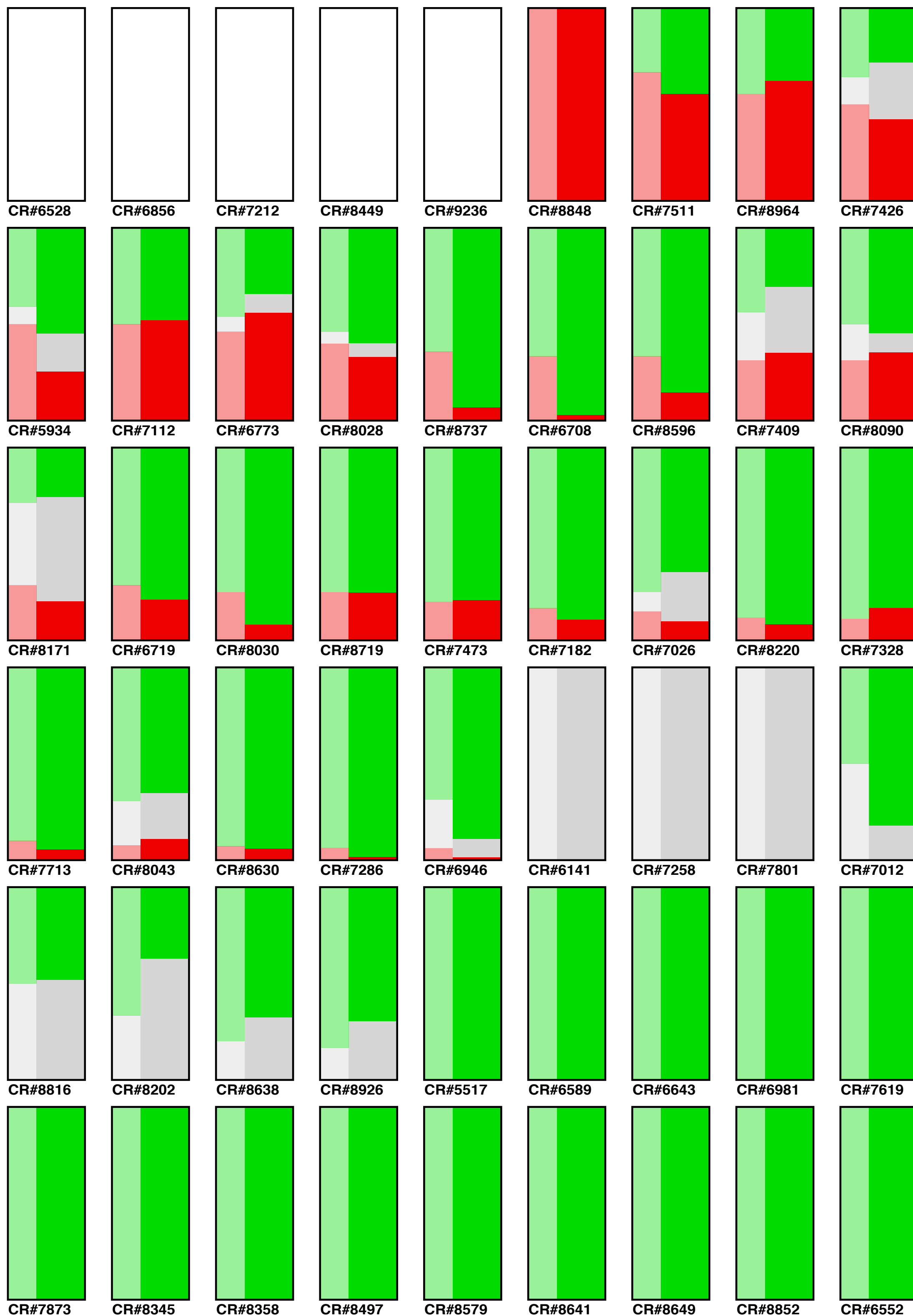


Abbildung 5.8: Jedes Rechteck steht für einen CR. In Farben sind die Anteile des Changesets dargestellt, die im Testfall (grün) oder im Startprozedere (grau) ausgeführt bzw. nicht ausgeführt wurden (rot). Links: ungewichtete Anzahl der Methoden, wie in Abb. 5.7. Rechts (breiter): Nach Methodenlänge gewichtete Anzahl der Methoden. Die Reihenfolge entspricht der in Abb. 5.7.

5.3.4 RQ3

Die Antworten der Entwickler ließen sich in insgesamt 11 Kategorien einteilen, die in Tabelle 5.5 dargestellt sind. Mit Abstand die größte Gruppe war die der Methoden, die durch ein Refactoring verändert wurden und nicht zwingend in direktem Zusammenhang mit der im CR entwickelten Funktionalität stehen.

Sollte ausgeführt werden Zu CR#8043 und CR#6773 wurde je einmal vom Entwickler angegeben, dass eine als Test-Gap enthaltene Methode ausgeführt worden sein sollte. Diese Methoden verdienen also Betrachtung.

Eine Test-Gap Methode des CR#8043 ist einer Funktion Teamscales zuordenbar, die bei Änderungen an einem Task eine Benachrichtigungs-E-Mail versendet. Der Entwickler bestätigt, dass seiner Meinung nach der Testfall hinsichtlich der Java-Methoden komplett sei. Allerdings wurde keine Nutzung von Tasks im Testfall spezifiziert. Die Nichtausführung der Methode ist also auf einen inkompletten Testfall zurückzuführen.

Die Test-Gap Methode aus CR#6773 wurde kommentiert mit „should be tested; main change in CR“. Die Funktionalitätsänderung in diesem CR betraf den Import einer Nutzergruppe aus einem Verzeichnisdienst. Ungetestet blieb eine Methode `setValues`, deren Funktion es ist, eine Liste von Nutzern in eine Datenstruktur einfügen und die sich also intuitiv gut dafür eignet in einem Test des CR#6773 eingesetzt zu werden. Es gilt also nachzuverfolgen, warum diese Methode nicht aufgerufen wurde. Eine Methode, die Rahmen des CRs verändert wurde, trägt den Namen `importGroup` und wird als zeitlich vorher ausgeführte Methode vermutet. Bei deren Analyse stellte sich heraus, dass darin in einer `for`-Schleife über die vom Server empfangene Liste von Nutzern iteriert wird und jeder Nutzer an die Methode `setValue` (für einen einzelnen Nutzer, vgl. `setValues` oben) übergeben wird:

```
1 for (String userUid : users) {
2     [...]
3     userIndex.setValue(user);
4 }
```

Tabelle 5.4: Die Methoden, die für Tester als unwichtig erkannt werden konnten, ließen sich in die drei Kategorien „einfacher Getter“, „too trivial to test“ und „einfache toString-Methode“ einordnen.

Kategorie	Häufigkeit	Fund in CR
Einfacher „Getter“	12 (10,9%)	CR#5934, CR#6708, CR#8028, CR#8030, CR#8737
Too trivial to test	12 (10,9%)	CR#5934, CR#7286, CR#8028, CR#8090
Einfache toString-Methode	4 (3,6%)	CR#7286, CR#7426, CR#8630, CR#8737
Potentiell wichtig	82 (78,2%)	
Test-Gaps	110 (100%)	

Tabelle 5.5: In dieser Tabelle sind die 11 verschiedenen Kategorien von Methoden dargestellt, die nach den Angaben der Entwicklern klassifiziert werden konnten.

kritisch: 20/110 (18,2%)	sollte ausgeführt werden	2	1,8%
	unzureichender Testfall	18	16,4%
eher unkritisch: 56/110 (50,9%)	CR-irrelevant oder Refactoring	54	49,1%
	Exception bei schweren Fehlern	1	0,9%
	Elternmethode überschrieben	1	0,9%
andere Coverageinfo nötig: 3/110 (2,7%)	Plugin-Code	1	0,9%
	Methode für Unit Tests	2	1,8%
uninteressant nach RQ2: 28/110 (25,5%)	einfache „Getter“	12	10,9%
	too trivial to test	12	10,9%
	toString-Methode	4	3,6%
	Antwort fehlt	3	2,7%
	Test-Gap-Methoden	110	100%

Die Methode `setValue` wurde im Übrigen auch im Rahmen des CRs verändert und in der Berechnung richtig als ausgeführt behandelt.

In diesem Fall hat sich der Entwickler geirrt und die Methode ist tatsächlich ein Test-Gap. Eine in Eclipse erstelle Ausführungshierarchie ergab, dass die Methode zum heutigen Zeitpunkt von keiner anderen Methode referenziert wird. Die Klasse, in der sich der Test-Gap befindet, erbt von einer Basisklasse und die betroffene Methode überschreibt die der Basisklasse. Um dies zu verdeutlichen, trägt sie den Kommentar `This method is reimplemented because the baseclass implementation will result in wrong results`. Die Methode wird zwar derzeit nicht verwendet, wurde aber für den Einsatz in künftiger Entwicklungsarbeit vorbereitet und soll das richtige Verfahren sichern. Ob sie das tut, ist nicht bekannt und kann nicht getestet werden. In der Interpretation wird die Kategorie dieser Methode zu „Methode aus Basisklasse überschrieben“ berichtigt.

Methode aus Basisklasse überschrieben Ein ähnlicher Fall wurde vom Entwickler des CR#8171 bemerkt. Auch hier stellte die Change Request Coverage einen Test-Gap heraus. Zwei Methoden aus der Elternklasse wurden überschrieben, wobei die eine eine detaillierte Anfrage beantwortet, die andere Methode eine einfache. Erstere wurde im Testfall ausgeführt. Der zugehörige Assignee bemerkte, dass es eventuell möglich sei durch einen anders definierten Testfall auch diese Methode abzudecken, aber er wisse nicht, unter welchen Bedingungen dies der Fall sei.

Exception für schlimme Fehler In der Studie wurde zu dieser Kategorie eine Methode in CR#6946 gefunden. In **RQ2** wurde sie als uninteressant in die Ka-

tegorie „too trivial to test“ eingeordnet. Der Entwickler stimmte zu, dass die Methode nicht testenswert sei und fügte im Kommentar hinzu, dass sie lediglich sehr schwerwiegende und seltene Fehler abfängt und eine entsprechende Exception erzeugt.

toString-Methode Alle Entwickler stimmten den Ergebnissen aus **RQ2** zu, was die Kategorisierung und Testwürdigkeit der toString-Methoden anbelangt. Einmal wurde angegeben, dass die toString-Methode lediglich zu Debug-Zwecken implementiert wurde.

einfache „Getter“ Alle Entwickler stimmten den Ergebnissen aus **RQ2** zu, was die Kategorisierung und Testwürdigkeit der getter-Methoden anbelangt. Ein Entwickler bezog sich auf die Serialisierung der Java-Klassen, die bei Teamscale stattfindet. Ein konkreter Grund für die Existenz der getter-Methoden wurde in diesem Zusammenhang nicht genannt.

Plugin Code In der Berechnung der CRC zu CR#7426 blieb eine Methode ungetestet. Nach Aussage des Entwicklers gehört sie zum Plugin Code und kann deshalb nicht im Teamscale UI-Test getestet werden. Der Testfall war also unvollständig.

Methode für JUnit Tests Testcode wurde zwar ausgeschlossen, allerdings hat sich auch außerhalb davon im Programmquelltext durch die CRC Berechnung Code gefunden, der ausschließlich für den Aufruf durch automatische Unit Tests geschrieben wurde. Dies betraf zwei gefundene Test-Gaps. In beiden Fällen handelte es sich um Konstruktoren, die zur Erzeugung eines für den Unit Test geeigneten Objekts dienen.

too trivial to test Die Kommentare zu den 7 von den Entwicklern als „zu einfach, um sie zu testen“ bewertet wurden, waren unterschiedlich. Insgesamt dreimal und damit am häufigsten als Zusatz zur Bewertung genannt wurde die Unsicherheit, warum die Methode nicht ausgeführt wurde. Es wurde aufgeführt, dass es bei der gefundenen Methode um unbenutzten Code handeln kann und es in jedem Fall interessant wäre herauszufinden, warum es hier nicht zur Ausführung kam. Zweimal betraf es das Auslösen einer Exception, was nur unter einer bestimmten großen Analyse geschieht. Hier kann eine Verbindung gezogen werden zur oben genannten Kategorie „Exception für schlimme Fehler“. Eine Methode dieser Kategorie regelt das Verhalten der Elternklasse.

Unzureichender Testfall Obwohl alle Testfälle zur Validierung den jeweiligen CR Assignees vorgelegt wurden, wurde zu 18 Methoden angegeben, dass der Testfall nicht ausreichend war, um den Test-Gap auszuführen. Hierzu zählen drei Methoden aus dem in 5.3 bereits vorgestellten CR#8848. Mit dem entworfenen Testfall wurde der Status der lokalen SVN-Kopie nicht so „verunreinigt“, um die Stabilitätsverbesserung, die Aufgabe des CRs war, in Anspruch zu nehmen.[3] Außerdem zählen sechs Methoden des CR#5934 hierzu. Die nicht gecoverten Methoden hätten einen Parsingfehler des zu analysierenden Projekts bedurft, der mit dem im Testfall beschriebenen Verfahren

3 Sowohl die Provokation eines unreinen SVN-Status in CR#8848 als auch die des Parsingfehlers in CR#5934 sollten durch einen Unit-Test abgedeckt sein. Zöge man auch die Coverageinformation aus dem Unit-Test zur Berechnung der CRC hinzu, sollten die entsprechenden Methoden abgedeckt sein. Ist dies nicht der Fall, liegt ein richtiger Test-Gap vor.

Tabelle 5.6: Von Entwicklern genannte Gründe für Refactorings/CR-irrelevante Änderungen.

Kategorie	Häufigkeit	Fund in CR
Verwendung Konstante	7 (13,0%)	CR#7112, CR#8719, CR#8964
`throws` Deklaration entfernt	15 (27,8%)	CR#8964
Variable/Methode umbenannt	8 (14,8%)	CR#7182, CR#8028
Funktionalität-Reuse	5 (9,3%)	CR#7409, CR#7511, CR#8028, CR#8171
Typadaption	2 (3,7%)	CR#7409
Nicht spezifiziert	17 (31,5%)	
Refactorings	54 (100%)	

nicht ausgelöst werden konnte.[3] Zu einer Methode aus dem CR#8090 gab der Entwickler an, dass er an den Spezialfall, in dem diese Methode aufgerufen wird, bei der Validierung/Verbesserung des Testfalls nicht gedacht habe. Mit CR#7436 wurde die Funktionalität entwickelt, mittels eines Serviceaufrufs von einer Datei (ggf. mit Pfadangabe) auf den Pfad zur Datei innerhalb eines Teamscale-Projekts zu gelangen. Hier wurde beim Festlegen des Testfalls der Fehler begangen, dass die Analyse des Projekts nicht Teil des Testfalls war, sondern der Store schon vor Aufnahme der „Base"-Coverage (und somit auch vor Aufnahme der Testfall Coverage, vgl. dazu auch den ab 5.2.3 beschriebenen Vorgang) erstellt wurde. Allerdings nahmen die Methoden, die in die Analyse eingreifen, einen wesentlichen Anteil am Changesets des CRs ein. So wurden zwei Methoden nicht getestet, weil keine Analyse stattfand; weitere drei hätten eine Analyse benötigt, für die aus einem verbundenen Repository mindestens je eine Datei gelöscht und hinzugefügt wurde. Eine Methode `validate` aus dem CR#8028 hätte eine weitere (optionale und im Testfall nicht spezifizierte) Konfiguration verlangt, um ausgeführt zu werden. Im CR#7328 wurde mit einem Filesystem Connector bearbeitet. Um die einzige Test-Gap Methode auch noch abzudecken, hätte mit noch einem anderen Connector getestet werden müssen.

CR-irrelevant oder Refactoring Diese Kategorie ist mit 54 zugeordneten Methoden mit Abstand die größte. Zu 31 Methoden wurde von den Entwicklern jeweils noch ein erläuternder Kommentar abgegeben, um welche Art Refactoring es sich handelt. Die genannten Gründe sind in Tabelle 5.6 zu überblicken. Die Subkategorien werden hier im einzelnen besprochen.

`throws` Deklaration entfernt Einer der untersuchten CRs veränderte eine in Teamscale häufig genutzte Serialisierungsfunktion. Die alte Variante davon löste in bestimmten Fällen eine Exception aus. Nach der Reimplementation war dies nicht mehr der Fall und von allen Methoden, die diese Funktion vorher genutzt hatten, konnte nun die `throws` Deklaration entfernt werden.

Variable in der Mutterklasse umbenannt Hier wurden Klassenvariablen in

```
User newUser = LDAPUtils.importUser(user.getUsername(), server,
                serverName);
```

```
User newUser = LDAPUtils.findUser(user.getUsername(), server,
                serverName);
```

Abbildung 5.9: Beispiel für einen Refactoring Fall. Verglichen wird ein Ausschnitt einer in CR#7182 bearbeiteten Methode vor den Änderungen durch den CR und danach. Die aufgerufene Methode `importUser` erfuhr im Laufe des CR drei Änderungen: in ihrer Sichtbarkeit, in einer Parameterbezeichnung und in ihrem Namen. Die Namensänderung wurde an die Methode `getUserDN` weitergegeben. Es handelt sich hier also um eine Refactoring bedingte Änderung.

```
if (user == null) {
        return serializeObjectToHttp("failure", query);
}
```

```
if (user == null) {
        return FAILURE;
}
```

Abbildung 5.10: Beispiel für einen Refactoring Fall. Im Falle eines Fehlers wurde zunächst die Meldung „failure“ im Methodenquelltext definiert und serialisiert. Im Laufe des CR#7112 wurde der entsprechende Abschnitt durch eine Konstante ersetzt, die eine allgemein geltende serialisierte Fehlermeldung hält. Diese Ersetzung wurde in der betroffenen Methode zwei Mal durchgeführt; sie waren die einzigen an der Methode vorgenommenen Änderungen.

einer Mutterklasse umbenannt. Entsprechend musste in allen von ihr erbenden Klassen dieser Identifier ausgetauscht werden. Die Umbennung betrifft also auch Methoden, die sonst eventuell gar nicht verändert worden sind.

Methode umbenannt Ein Beispiel hierfür zeigt Abbildung 5.9. Eine genutzte Methode erfuhr keine Änderung in der Logik, wohl aber in ihrer Benennung.

Verwendung einer Konstanten Werte, die vorher direkt in einer Methode verwendet wurden, wurden hier einer Konstante zugewiesen (sofern diese nicht schon vorher existierte) und in den Methoden entsprechend durch die Konstante ersetzt. Ein Beispiel hierfür zeigt Abbildung 5.10.

Wiedergebrauch von Funktionalitäten Diese Kategorie von Refactoringänderungen wird durch den Vorgang beschrieben, eine Methode aus einer vorhandenen zu extrahieren und die extrahierte Methode in der Implementation einer neuen Funtkionalität wiederzuverwenden. Die ursprüngliche Methode (aus der extrahiert wird) wird dabei verändert und deshalb bei der Berechnung der Change Request Coverage auch entsprechend behandelt. Getestet wird aber die neue Funktionalität, die die ursprüngliche Methode gegebenenfalls nicht verwendet.

Typenadaption In zwei Fällen wurde der Typ einer Variablen beziehungsweise der Typ einer Funktion verändert und die nutzende Methode

```
/** {@inheritDoc} */
@Override
protected IRepositoryConnection createConnection()
            throws RepositoryException {
        return new GitRepoRepositoryConnection(connectionIdentifier,
                    patternSupport, getRootDirectory(), contextIndex, startRevision);
}
```

```
/** {@inheritDoc} */
@Override
protected IRepositoryConnection createConnection()
            throws RepositoryException {
        return new GitRepoRepositoryConnection(connectionIdentifier,
                    patternSupport, new File(getRootDirectory().getFile()),
                    contextIndex, startRevision);
}
```

Abbildung 5.11: Beispiel für einen Refactoring Fall. Oben: Code am Anfang des CR#8028. Unten: Code am Ende des CR#8028. In diesem Fall wurde der Rückgabetyp der Methode `getRootDirectory` von `File` in `URL` geändert und die Konvertierung zog entsprechend um.

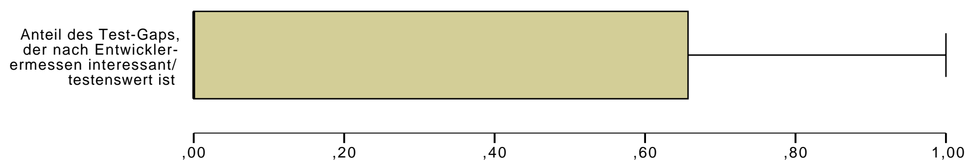


Abbildung 5.12: Verteilung der interessanten Test-Gap Anteile für die getesten CRs

entsprechend angepasst.

Ein Beispiel für ein Refactoring, das vom CR-Assignee nicht weiter spezifiziert wurde, ist in Abbildung 5.11 gezeigt. Der Rückgabetyp einer genutzten Methode wurde verändert, und entsprechend musste die nutzende Methode angepasst werden.

Auf jedem Fragebogen zur CRC wurde der Entwickler bei einem aufgetretenen Test-Gap hinterfragt, welche der unausgeführten Methoden ein interessanter/testenswerter Gap ist und welche nicht. Insgesamt wurde zu 37,6% der nicht ausgeführten Methoden angegeben, dass es sich hierbei um eine testenswerte Methode handle.

Der Boxplot in Abbildung 5.12 zeigt, wie sich die von den Entwicklern als interessant bewerteten Test-Gap-Anteile für einen CR verteilen. In 25% der Fälle lag der interessante Anteil (Methodenanzahl) des Test-Gaps zwischen 66% und 100%. In 25% der Fälle waren 0%-66% der gefundenen Test-Gap-Methoden für den Entwickler interessant und in der verbleibenden Hälfte der Fälle wurden keine für den Entwickler relevanten Test-Gap-Methoden gefunden. Die Abbildung 5.13 zeigt dazu, wie groß (relativ) die Test-Gap-Anteile sind, die von Entwicklern als interessant bewertet wurden und wie häufig sich diese Größen in der Studie feststellen ließen.

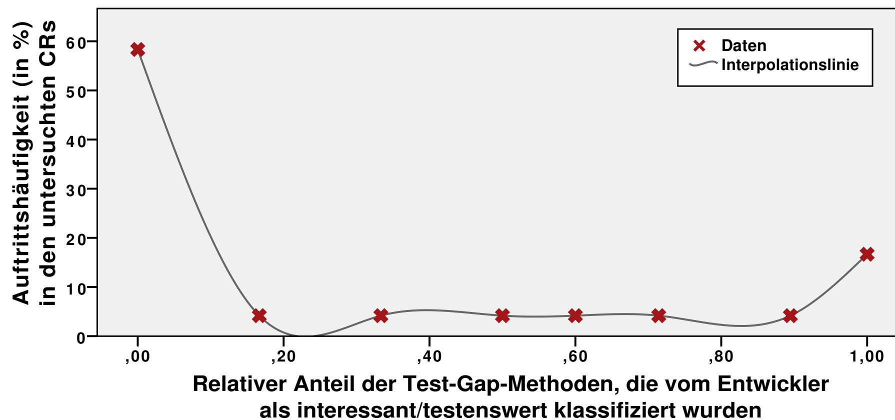


Abbildung 5.13: Auftrittshäufigkeiten von Größen des interessanten Test-Gap-Anteils, die im Rahmen der Studie festgestellt werden können. Sehr häufig finden die Entwickler keine Test-Gap-Methode testenswert. Am zweithäufigsten halten sie aber alle gefundenen nicht ausgeführten Methoden für testenswert.

5.4 Interpretation / Beantwortung der Forschungsfragen

Nachdem die erhobenen Daten im vorhergehenden Abschnitt vorgestellt wurden, werden – nach einem Einschub zur Gewichtung der Changeset Methoden mit ihren Längen – im Folgenden die Forschungsfragen beantwortet.

5.4.1 Gewichtung mit Methodenlänge

Geht man davon aus, dass in kürzeren Methoden auch tendenziell weniger Fehler auftreten, so ist die Überlegung, Methoden mit ihren Längen zu gewichten, sehr sinnvoll. Dieses Vorgehen erlaubt es, bei der Berechnung der Change Request Coverage präzisere Ergebnisse zu bekommen, als ohne Gewichtung, da sich die erwartete Fehlerwahrscheinlichkeit auf die Anteilswerte auswirkt.

Auch in den Quartilen der Anteilswerte macht sich die Gewichtung bemerkbar. So steigt der Median der Gruppe „exklusiv durch den Test ausgeführt“ leicht an; der Median der Gruppe „nicht ausgeführt“ sinkt etwas. Der größte Unterschied in den betrachteten Quartilen zeigt sich im oberen Quartil $Q_{0,75}$ der Anteilsgruppe „nicht ausgeführt“: In 75% der durchgeführten CR Testungen wurden bis zu 31% der Methoden aus dem Changeset (ohne Gewichtung) nicht ausgeführt. Mit Gewichtung beträgt dieser Quartilswert nur noch 21%. Das bedeutet, dass es sich bei den nicht ausgeführten Methoden vermehrt um kurze handelte. Gewichtet man die Methoden nicht mit ihrer Länge, so ist also der rote Anteil der Coverageampeln in Abbildung 5.7 „zu groß“ und die Change Request Coverage ergibt schlechtere Werte als gewünscht. Denn entsprechend ihrer kurzen Länge, sollen diese Methoden auch einen geringen Einfluss auf den Anteilswert haben, was nicht der Fall ist, wenn nur die Methodenanzahl je Gruppe beachtet wird.

In Abbildung 5.8 sind die ungewichteten und gewichteten Coverageampeln gegenübergestellt.

5.4.2 RQ1

Wie viel Change Request Coverage ist bei manuellen Tests unabhängig vom eigentlichen Test und damit aussagelos? Durch die untersuchten Change Request wurden insgesamt 511 Methoden hinzugefügt oder verändert. Davon wurden 401 Methoden abdeckt; lediglich 37 (9,2%) von diesen ausgeführten Methoden wurden durch Coverageinformationen abgedeckt, die durch das Startszenario erzeugt wurden. Betrachtet man den testunabhängigen Anteil pro CR, so liegt der Wert hierfür in 50% der Fälle bei 0% und in 75% der Fälle bei kleiner oder gleich 18%. Der Mittelwert dieses Anteils in der Change Request Coverage ist 13,3%.

Für sehr viele der getesteten Fälle ist die Change Request Coverage also komplett testabhängig. Die Forschungsfrage ist damit hinreichend beantwortet.

Interessant ist allerdings die Frage, in welchen Fällen sie nicht testabhängig ist. In der vorliegenden Studie tritt dieser Fall dann ein, wenn Änderungen an Systemteilen vorgenommen werden, die bereits bei der Initialisierung des Systems oder beim Ausführen des Startszenarios aufgerufen werden. Änderte also ein untersuchter CR etwas am JavaScript Quelltext und in einer Methode des JavaScript-Kompilers, so wurde letztere schon bei der Initialisierung von Teamscale aufgerufen. Erstellte man einen neuen Service, so wurde dieser beim Start registriert. Betraf eine Änderung die Anzeige der Startseite, so wirkte sich auch dies auf den testunabhängigen Teil aus.

Wie in Bemerkung 4 festgehalten, wiesen die fünf untersuchten CRs lediglich ein Changeset auf, das ein bis zwei Methoden enthielt. Im Umkehrschluss kann man annehmen, dass für CRs mit großen Changesets der Einfluss von Methoden, die im Startprozedere ausgeführt wurden, gering ist.

Fazit: Die Change Request Coverage weist unter bestimmten Bedingungen einen testunabhängigen Teil auf. Die Auswirkungen halten sich insbesondere bei CRs mit einem großen Changeset in Grenzen. Weiterhin können betroffene CRs oft schon vor der Testausführung identifiziert und der testunabhängige Teil durch ein geändertes Startprozedere verringert werden.

5.4.3 RQ2

Warum sind manche Test-Gaps für Tester uninteressant und können wir sie systematisch ausschließen? Es ist wichtig uninteressante Methoden aus der Change Request Coverage auszuschließen, um diese als effektive Metrik benutzen zu können. Die Verfahren, die in der Studie manuell angewandt wurden, um die vermeintlich unwichtigen Methoden zu identifizieren, können automatisiert werden. Mögliche Ausschlussverfahren für die gefundenen Kategorien sind im Anschluss dargelegt.

Einfache getter-Methoden Getter-Methoden werden im Allgemeinen einheitlich benannt. So starten all diese Methoden mit `get` und werden gefolgt vom Namen der Membervariable, die sie zurückgeben. Außerdem werden einer solchen Methode keine Parameter übergeben und sie sollte im auszuführenden Code einzig das `return`-Statement beinhalten, gefolgt von der dazugehörigen Membervariable. Für einen Ausschluss wird empfohlen, auch keine Bedingungen (zum Beispiel eine Überprüfung auf `null`) zuzulassen

beziehungsweise in einer Analyse eine Option anzubieten einfache Logiken zuzulassen.

Die Untersuchung einer Methode auf das Vorhandensein von Bedingungen oder anderen Logiken kann anhand eines abstrakten Syntaxbaums erfolgen.

Too trivial to test Eine Methode, die ihrer Natur nach zu trivial ist, um sie testen zu müssen, beinhaltet nur eine sehr begrenzte Anzahl von auszuführenden Statements. Als eine mögliche Konfigurationsoption der CRC ist es vorstellbar eine maximale Anzahl von Statements definieren zu können. Der systematische Ausschluss aufgrund der Statementanzahl ist ohne großen Aufwand möglich.

Einfache toString-Methode `toString`-Methoden überschreiben immer die Methode `toString` aus der `Object`-Klasse und heißen somit einheitlich. Sie können dadurch einfach erkannt werden. Der Grund für existierende, aber niemals im Programmkontext aufgerufene `toString`-Methoden wurde nicht untersucht.

Es konnten also Kategorien von für Tester uninteressanten Methoden heraus- und Ansätze zu deren Ausschluss vorgestellt werden. In der gemachten Studie wären 28 von 110 Methoden (25,4%) ausgeschlossen worden. Dieser Auschluss würde eine um die uninteressanten Methoden „bereinigte“ Change Request Coverage ergeben. Die Auswirkungen davon, die sich in dieser Studie ergäben, sind in Abbildung 5.14 dargestellt.

Ist ein System sehr kritisch und erfordet es eine äußerst konservative/vollständige Testung/Testabdeckung, so ist es gegebenfalls angebracht die ausgeschlossenen Test-Gap Methoden trotzdem in der Change Request Coverage aufzuführen oder in einer Grafik abzubilden. Code, der lediglich zu Debugging-Zwecken implementiert wurde (wie es in der Studie bei toString-Methoden auftrat) muss nicht auch in den Quelltext des Test- oder später Produktivsystems übernommen werden.

5.4.4 RQ3

Finden wir aus Sicht des Entwicklers/des Testers relevante Test-Gaps? Von den 11 gefundenen und in Tabelle 5.5 aufgelisteten Test-Gap-Kategorien sind etwa 18,5% für den Entwickler besonders interessant. Bei diesen Methoden handelt es sich um Test-Gaps, die entweder durch einen unzureichenden Testfall bedingt sind oder die nach dem Entwickler durch den Testfall hätten ausgeführt werden müssen.

Es werden also für den Entwickler interessante Test-Gaps gefunden. Es wäre jedoch wünschenswert, diese Methoden weiter zu isolieren, sodass der interessante Anteil der Change Request Coverage steigt. Dies ist durchaus möglich.

Bemerkung 5:
Die Test-Gap Kategorien „Plugin Code“ und „Code für den Unit Test“ können durch das Hinzufügen der relevanten Coverageinformationen eliminiert werden.

Die drei Gruppen uninteressanter Test-Gaps, die in **RQ2** identifiziert wurden, können durch das dort vorgeschlagene Verfahren die CRC im Einsatz effizienter

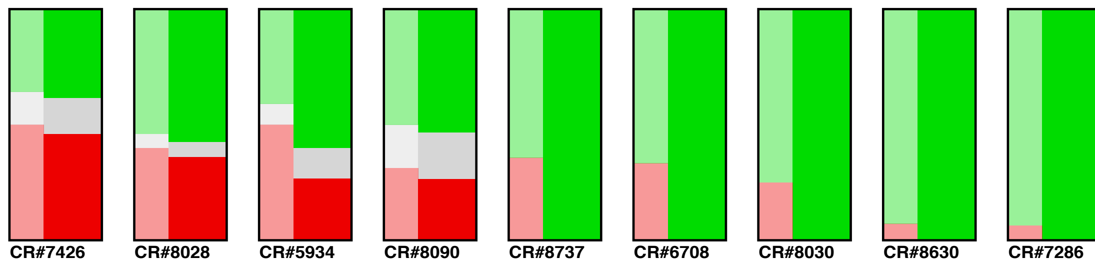


Abbildung 5.14: Die pro CR links dargestellte „Coverageampel“ zeigt die Ausführungsanteile für den jeweiligen CR ohne Ausschluss der in **RQ2** identifizierten „uninteressanten“ Methoden. In der rechten „Coverageampel“ sind diese Methoden ausgeschlossen.

machen. In Abbildung 5.14 wird die Auswirkung dieses Auschlusses für die CRs aufgezeigt, die „uninteressante“ Methoden enthielten.

Das Auftreten der Methodengruppen „Exception bei schweren Fehlern“ und „Methode aus der Superklasse überschrieben“ war in der Studie selten und mag auch im realen Einsatz der CRC eine untergeordnete Rolle spielen. Für die zweite der beiden Kategorien ist weiter unten aber noch eine mögliche Guideline angegeben, die Funde dieser Art verhindert und für die Gesamtqualität des Quelltextes einträglich sein kann. Die Hälfte der Test-Gap Methoden erfuhren ein Refactoring aufgrund einer Implementierung im untersuchten CR, sind aber nicht CR relevant. Diese Funde können durch eine integrierte Refactoring Detection eingedämmt werden. Ob und in wie weit sich die Subkategorien davon durch eine solche filtern lassen, ist eine interessante Fragestellung für zukünftige Studien.

Um in der CRC mit dem Fall umzugehen, dass eine Methode der Basisklasse überschrieben, aber nicht genutzt wird, könnte man vorschlagen: Die Methode wird entweder nicht implementiert oder sie wird mit einer entsprechenden Annotation vermerkt implementiert, dass sie bisher nicht in Gebrauch und ungetestet ist. Für beide Varianten gibt es Argumente:

- Auch eine Implementation, die nicht genutzt wird, muss gewartet werden, dafür verstanden und dafür wieder entsprechend Zeit aufgewendet werden. Die Implementation wirkte sich also negativ auf die Wartungskosten aus.
- Eine Nicht-Implementation bringt für denjenigen Entwickler, der die Methode als erstes nutzen möchte, einen hohen Mehraufwand. Je weniger er mit der Funktionalität, zu der die unimplementierte Methode gehört, vertraut ist, desto länger braucht er. Mit hoher Wahrscheinlichkeit hätte der Entwickler, der die Klasse erstellt hat, weniger Zeit gebraucht, da er den Teil des Systems ohnehin gerade bearbeitet.

  Eine entsprechend annotierte Methode kann bei der Berechnung der Change Request Coverage ausgeschlossen werden. Wenn gewünscht, ist dies auch bei der methodenbasierten Coverageübersicht möglich und nützlich. Eine überschreibende Methode, die sinnvollerweise (wie dargestellt) der Vollständigkeit halber implementiert wurde, die aber zum derzeitigen Zeitpunkt nicht genutzt wird, kann nicht aufgerufen werden und verschlechtert so die Testabdeckungs-Bewertung eines Systems.

Die Analyse der Test-Gap-Anteile, die von Entwicklern als testenswert eingeschätzt wurde, deutet auf eine hohe Dichte uninteressanter Funde hin. Allerdings beinhalten die Ergebnisse zum Beispiel ebenfalls die Methoden, die bereits in der **RQ2** als uninteressant identifiziert wurden. Schließt man diese von der CRC aus, steigt auch der Anteil interessanter Test-Gaps. Außerdem ist zu befürchten, dass nicht jeder Entwickler die Frage auf dem Fragebogen nach der Testenswertigkeit gleich interpretiert hat. So ließen sich bei manchen Entwicklern Tendenzen feststellen, alles testen zu wollen, da auch eine vermeintlich unwichtige Änderung Fehler in das System einschleusen könnte. Manche Entwickler wollten auch nur das testen, was funktionell zu ihrem CR gehört. Für sie sind Methoden, die zum Beispiel durch ein Refactoring geändert wurden, nicht testenswert. Andere Entwickler sehen das anders. Diese uneinheitliche Betrachtung erschwert eine genaue Interpretation.

RQ2 und **RQ3** haben in der Bewertung der uninteressanten Test-Gaps einen gemeinsamen Anteil. Bei den in **RQ2** als uninteressant und systematisch auszuschließenden Test-Gap-Methoden wurde auch erwartet, dass sie vom Entwickler als *nicht testenswert* beurteilt werden und so die Einteilung in **RQ2** bestätigt wird. Zwei Methoden, die in **RQ2** als „too trivial to test“ klassifiziert wurden, also sehr wenige Statements beinhalteten, wurden von den Entwicklern als Refactoring, gleichzeitig aber als *testenswert* eingestuft. Ansonsten stimmten die Entwickler den Einteilungen aus **RQ2** zu. Als Unterschied der beiden letzten Forschungsfragen ist hervorzuhaben, dass in **RQ2** die Einteilung durch den Autor erfolgte, der lediglich den Stand der Methode zum Endzeitpunkt des CRs, aber nicht die Änderungen an der Methode betrachtete. In **RQ3** erfolgte die Klassifizierung und Bewertung der Methoden stattdessen durch den Entwickler, der die gemachten Änderungen an der Methode berücksichtigen konnte.

5.5 Gefährdungen der Validität

Der Autor ist kein professioneller Tester und weist nicht in allen Bereichen Teamscales dieselbe Erfahrung auf. Aus diesem Grunde wurden einige CRs ausgeschlossen, sodass fehlerhafte Testungen keinen negativen Einflüsse auf die Untersuchung haben können.

Nach der ersten Probeberechnung stellte sich deutlich heraus, dass der Zeitraum zwischen CR-Schließung und Testung einen Einfluss auf die Change Request Coverage hat. Deswegen wurde die Testung jeweils auf einer für den CR aktuellen Version durchgeführt.

Es könnte Tendenzen von Entwicklern oder von Subsystemen geben, die die Studie verfälschten. Die CRs der Entwickler wurden ähnlich einer stratifizierten Suche ausgewählt, um solche Effekte zu vermeiden.

Diese Studie anhand von Teamscale ergab wichtige erste Untersuchungsdaten zur CRC. Die Ergebnisse dieser einzelnen Studie sollten allerdings nicht generalisiert werden. Wiederholte Testungen auch an anderen Projekten sind zu empfehlen mit dem Ziel die gewonnenen Erkenntnisse zu bestätigen.

6 Fazit / Zusammenfassung

Die Change Request Coverage liefert hilfreiche Ergebnisse und kann durch weitere Entwicklung und Forschung zu einer guten Metrik und zu einem sehr hilfreichen Werkzeug für Tester werden.

In der Studie stellte sich heraus, dass die Change Request Coverage nur unter bestimmten Bedingungen einen testunabhängigen Teil aufweist und dass dies bei CRs mit größerem Changeset eine kleinere Rolle spielt. Ferner wurden Kategorien von uninteressanten Test-Gap-Methoden gefunden und mögliche Verfahren zum systematischen Ausschluss derselben gegeben. Die Entwickler bestätigten mit ihren Angaben, dass die CRC interessante Test-Gaps aufdeckt – ein Potential, das sich weiter verbessert, wenn uninteressante Test-Gaps ausgeschlossen werden und sich der Anteil an bedeutenden Funden erhöht.

Einige Punkte wurden in dieser Studie festgestellt, die die CRC auszeichnen oder auf die bei deren Einsatz geachtet werden muss.

CRs altern Genau wie der Entwickler vergisst, welche Änderungen er genau durchgeführt hat, so entfernen sich die Änderungen im Code vom Stand des letzten Commit des CRs mit der Zeit. So ist es mit der `HEAD`-Revision ohne Methodenhistorie je nach Alter des CRs schwierig bis unmöglich adäquat die CRC zu berechnen. Siehe dazu auch die Beschreibung im Anhang, B.1.

Abweichungen von der CR Beschreibung Es sollte darauf geachtet werden, dass CRs nur solche Änderungen im Code mit sich bringen, die sich aus seiner textuellen Beschreibung beziehungsweise der Beschreibung des Features ergeben. Ansonsten werden die Änderungen falschen CRs/Features zugeordnet und können durch den featurespezifischen Test gar nicht abgedeckt werden.

Feature-/Codebasierte Testfälle Die Spezifikation des Testfalls ist zunächst featurebasiert. Durch Test-Gap Funde kann sich ergeben, dass dieser Testfall nachgebessert werden muss und dann gemischt feature- und codebasiert ist ([Jor13, Abschnitte 1.4 und 19.4.1]).

CR basiert Die kleingranulare Vorgehensweise auf Change Request Basis ist in großen Softwaresystemen zweckmäßig. Die Vorgehnsweise von Sherlund [She95] betrachtete alle Änderungen im Softwaresystem auf einmal, was bei großen Systemen unübersichtlich werden kann.

7 Ausblick

Die vorliegende Studie schließt zum aktuellen Zeitpunkt eine Lücke in der Forschung. Coverageinformationen auf die Basis von Change Requests zu aggregieren und das Ergebnis als Metrik zu verwenden, ist neu. Das Potential der CRC hat sich in dieser Studie gezeigt. Allerhand Möglichkeiten weiter in diese Richtung zu forschen stehen offen. Einige von ihnen sowie Möglichkeiten in folgenden Durchführungen der Studie das Design anzupassen, sind im Folgenden dargelegt.

Betrachtung der Methodenänderungen In **RQ2** wurde sich lediglich die Methode zu ihrem letzten Änderungszeitpunkt angesehen und in den Bögen zur CRC, die den Entwicklern in **RQ3** vorgelegt wurden, war auch nur dieser Methodenstand abgedruckt. In einigen Fällen wurde betrachtet, wie sich die Methoden im Verlauf des CRs verändert haben. Geht man davon aus, dass sich die Entwickler noch an die (im Rahmen des CRs) gemachten Änderungen der präsentierten Test-Gaps erinnern und dieses Wissen zur Beurteilung nutzen, dann spielt dieser Punkt keine Rolle hinsichtlich der Frage **RQ3**. Bei der Beantwortung, welche Test-Gaps für Tester interessant sind, mag diese Betrachtung aber einen erheblichen Mehrwert mit sich bringen. Die Forschung schlägt Verfahren zur Revalidierung von geänderten Quelltextanteilen vor (vgl. zum Beispiel [FRC81]). Stellte sich mit einem solchen Verfahren heraus, dass eine gefundene Änderung nicht testenswert ist, so kann sie ausgeschlossen werden und die Change Request Coverage noch gezieltere/effizientere Hinweise darauf liefern, wo sich Fehler im System verstecken.

Erweiterung unvollständiger Testfälle Die Ergebnisse aus der Change Request Coverage können auch dafür verwendet werden, Unvollständigkeiten in bestehenden Testfällen zu finden und die Testfälle entsprechend anzupassen/zu erweitern. Mangelhafte Softwaretests zu einem Change Request erreichen eine niedrige Change Request Coverage. Sind solche erkannt, können beim zuständigen Entwickler (Assignee des CRs) Verbesserungen eingeholt werden.

Testfallcoverage isoliert messen Wurde eine Methode bereits durch das Startszenario ausgeführt (Coverage `BASE`), so wurde sie hier in der CRC als „unabhängig vom Testfall ausgeführt“ markiert. Das ist zweckmäßig, allerdings lässt das Verfahren in dieser Studie keine Unterscheidung zu, ob eine Methode ausschließlich durch das Startszenario oder aber durch das Startszenario und auch durch den Testfall ausgeführt wird. Vorzuschlagen ist also, das Studiendesign so abzuändern (im Falle einer erneuten Nutzung von Jacoco im Serverbetrieb), dass beim Abholen der Base Coverage vom Server die Coverage am Server zurück-

gesetzt (Coverageaufzeichnung beginnt neu) wird. Diese Möglichkeit hält Jacoco bereit und es ist dann möglich die oben erwähnte Unterscheidung vorzunehmen.

Relation zwischen CRs und Methoden Zur Berechnung der CRC wird festgestellt, welche Methoden zu einem CR gehören. Es ist auch möglich umgekehrt zu mappen, sich also die Fragen zu stellen „Wie viele CRs betreffen eine Methode?“ und „Was bedeutet das für Methode, CR und Funktionalität?“.

Diese Fragen können in Zusammenhang gebracht werden mit der Studie von Hassan et al. [HH05], in der sich die Heuristik „Most Frequently Modified“ als guter Prädiktor für Fehler in einem Subsystem herausstellte.

Bugfix-CR vs. Feature-Request-CR In künftiger Forschung zum Thema sollte man sich der Frage annehmen, ob es eine Rollen spielt, welcher Natur ein CR ist, also ob es sich um einen Bugfix handelt oder im Rahmen des CRs neuer/präventiver Code in das System eingefügt wird. Macht es einen Unterschied, ob eine geänderte Methode Feature-Code oder Utility-Code zuzuordnen ist?

Weitere Anwendungsstudien Das in der Studie verwendete Projekt „Teamscale“ ist in Kontrollsystemen gut repräsentiert. Hier können bei der CRC Berechnung besonders dann Fehler auftreten, wenn Methodenänderungen keinem CR zuordenbar sind (Commit auf CR#0 als „Kleinigkeitenänderung“) oder wenn sie einem falschen CR zugeordnet werden (Vertippen bei der CR-Nummerneingabe beim Commit). Künftige Studien könnten die Fragestellung behandeln, ob die CRC in Systemen mit schlechterer Informationsqualität in Kontrollsystemen auch funktioniert.

Interessant wäre es auch, die CRC in einem System zu testen, das unter *Testgetriebener Entwicklung* entstanden ist. Funktionalitäten werden hier erst entwickelt, wenn ein passender Testfall existiert. Testfälle passen also herausragend zum System und so wird eine Schwierigkeit eliminiert, unter welcher die CRC sonst leidet: unzureichende oder unpassende Testfälle. Prinzipiell wäre ein erneuter Testlauf auf einem größeren System interessant.

Ebene der Coveragemessung Bei der Forschung zur CRC und besonders bei ihrer industriellen Verwendung sollte daran gedacht werden, was noch alles beachtet werden muss, wenn es um den Einsatz der Change Request Coverage geht. Sie basiert auf Testcoverage und unterliegt deswegen denselben Schwierigkeiten wie allgemein Tests: Wie tief soll die Ausführung gemessen werden? Im Beispiel der Studie wurde bis auf Methodenebene gemessen. Die Change Request Coverage ergibt in diesem Fall eine volle Abdeckung von 100%, wenn alle geänderten Methoden im Test ausgeführt wurden. Man bewegt sich also im Bereich der *Function Coverage*. Um eine genauere Abdeckungs-Messung vorzunehmen, könnte die Change Request Coverage auch auf Basis der *Statement Coverage*, der *Branch Coverage* oder der *Condition Coverage* berechnet werden. Dadurch ändert sich nicht das grundlegende in der Arbeit beschriebene Verfahren zur Berechnung, sondern

es ändern sich lediglich die Elementtypen in den zu vergleichenden Mengen (geändert/hinzugefügt, in der Basis ausgeführt, im Testfall ausgeführt) von Methoden zu Statements, Branches oder Konditionen.

Test-Gap-Methoden ohne Referenz auf eine geänderte Methode Im Falle eines Refactoring referenzieren die Methoden, die geändert, aber nicht getestet wurden, im Regelfall eine andere geänderte Methode. In zukünftigter Forschung könnte man sich die Frage stellen, was die Aussage von Test-Gap Methoden ist, die keine veränderte Methoden referenzieren? Gibt es diese überhaupt, oder sind es zwangsläufig False Positives?

Coveragequellen Change Request Coverage kann durch beliebige Coverage oder Coveragekombinationen berechnet werden. Interessant wäre eine Studie, in der die CRC mit Informationen aus einem Trampeltest oder aus einer explorativen Testung [Jor13, Kapt. 18] stammen.

Entwicklung als (Teamscale) Service Das für die Studie entwicklte Werkzeug eignet sich nicht für den produktiven Einsatz in einem Entwicklungsumfeld. Es ist lohnenswert einen Mechanismus zur Berechnung der CRC zu implementieren, der Teil eines größeren Systems ist. Ein Beispiel hierfür wäre die Integration in Teamscale. Unter anderem würde die Change Request Coverage von der bereits vorhandenen Refactoring Detection, und gleichermaßen von der vorhandenen Abbildung der Methodenhistorie profitieren. In diesem Zusammenhang sollte erforscht werden, inwieweit sich die Subkategorien der Refactoring Test-Gap Funde durch bisherige Ansätze der Refactoring Detection filtern lassen und ob eine Weiterentwicklung derselben für die CRC einträglich wäre. Außerdem sind in Teamscale Coverageinformationen von verschiedenen Testsystemen gesammelt verfügbar, sodass ein leichter Überblick darüber möglich ist, wie vollständig einzelne Testsysteme testen. In der kontinuierlichen Qualitätsanalyse ist dann noch interessant, wie sich die Change Request Coverage aggregiert über eine Menge von CRs (beispielsweise die Menge derer, die seit dem letzten Release dazugekommen sind) über die Zeit verändert. So kann bei gleichen Tests ein neuer CR eine Verschlechterung mit sich bringen oder die Qualität gleich lassen. Ein neu hinzugekommener Test bei gleicher CR Menge sollte die Qualität erhöhen. Über die Zeit betrachtet, sollte sich ein solcher aggregierter Qualitätsindikator nie im Wert verschlechtern.

Anhang

A Beispiel: CRC Bogen zu CR#8737

Change Request Coverage of CR#8737 (ashutoshb)

For my Bachelor's Thesis “Change request coverage as a metric of quality assurance for software tests” the CR#8737 (Display occurrence line number(s) for in coming and out going dependencies.) got analysed. According to redmine assignee was *ashutoshb*.

To quickly overview the matter of the change request it may be helpfull to look it up in redmine or Teamscale. Please also consider to view aggregated file changes on `build.cqse.eu/teamscale/issues.html#/teamscale/8737`. A test case was created:

> Use TS from store (analysed project e.g. JabRef). Navigate to Code Perspective and choose a file. View Dependencies by pressing 'D' or clicking on More → Show Dependencies. Line numbers for dependencies should be shown.

The test got executed and execution information got recorded and coverage information generated which methods got executed. Please judge now as developer of the CR whether this test case is correct and complete. So all added or changed methods within this CR should got executed.

☐ Yes ☐ No

If no, why?:

The following methods (changed or added within this CR) got executed through the test.

- `ElementDependencyService.java:DependencyLocationKey(String, String)`
- `ElementDependencyService.java:DependencyWithOccurrenceLocation(String, String, List)`
- `ElementDependencyService.java:DependencyWithOccurrenceLocation(String, String, int)`
- `ElementDependencyService.java:calculateIncomingDependenciesWithOccurrenceInfo(DependencyListIndex, TypeDependencyIndex, HistoryAccessOption, String, TypeIndex)`
- `ElementDependencyService.java:calculateOutgoingDependenciesWithOccurrenceInfo(DependencyListIndex, String, TypeIndex, List)`
- `ElementDependencyService.java:equals(Object)`
- `ElementDependencyService.java:hashCode()`
- `ElementDependencyService.java:populateDependenciesWithOccurrenceInfo(List, List, String, String)`
- `ElementDependencyService.java:processQuery(HttpQuery, DependencyListIndex, TypeDependencyIndex, HistoryAccessOption, boolean)`

The following methods (changed or added within this CR) get already executed when Teamscale starts. So this execution information was reached not exclusivly through the test case.

- No changed or added method executed.

The most interesting methods (changed or added within this CR) are these that get not executed within a well formulated test. For each of them you are asked: If the untested methods would be split up in two different categories “worth testing” and “not worth testing” (ok, if method is untested), which category would this method belong to?

- `org.conqat.engine.service/src/org/conqat/engine/service/architecture/ElementDependencyService.java, getDependencyLocation, 0, UNTESTED_ADDITION, 12909, 12978, Mon Dec 14 07:32:52 CET 2015`

```
1 public int getDependencyLocation() {
2             return dependencyLocation;
3         }
```

In another part of this study the method got already categorized as not worth testing. It is of course possible to disagree in this answer.

☐ worth testing ☐ not worth testing

1

In any case: why? In addition: Can you explain why it is untested (e.g. deficient test case; (refactoring of) not CR/feature related method)?

- org.conqat.engine.service/src/org/conqat/engine/service/architecture/ElementDependencyService.java, getDependencyUniformPath, 0, UNTESTED_ADDITION, 12648, 12726, Mon Dec 14 07:32:52 CET 2015

```
1 public String getDependencyUniformPath() {
2             return dependencyUniformPath;
3         }
```

In another part of this study the method got already categorized as not worth testing. It is of course possible to disagree in this answer.

☐ worth testing ☐ not worth testing

In any case: why? In addition: Can you explain why it is untested (e.g. deficient test case; (refactoring of) not CR/feature related method)?

- org.conqat.engine.service/src/org/conqat/engine/service/architecture/ElementDependencyService.java, toString, 0, UNTESTED_ADDITION, 12471, 12606, Mon Dec 14 07:32:52 CET 2015

```
1 public String toString() {
2             return dependencyUniformPath + ", " + dependencyLocationUniformPath
3                     + ", " + dependencyLocations;
4         }
```

In another part of this study the method got already categorized as not worth testing. It is of course possible to disagree in this answer.

☐ worth testing ☐ not worth testing

In any case: why? In addition: Can you explain why it is untested (e.g. deficient test case; (refactoring of) not CR/feature related method)?

- org.conqat.engine.service/src/org/conqat/engine/service/architecture/ElementDependencyService.java, getDependencyLocationUniformPath, 0, UNTESTED_ADDITION, 12776, 12870, Mon Dec 14 07:32:52 CET 2015

```
1 public String getDependencyLocationUniformPath() {
2             return dependencyLocationUniformPath;
3         }
```

In another part of this study the method got already categorized as not worth testing. It is of course possible to disagree in this answer.

☐ worth testing ☐ not worth testing

2

In any case: why? In addition: Can you explain why it is untested (e.g. deficient test case; (refactoring of) not CR/feature related method)?

- `org.conqat.engine.service/src/org/conqat/engine/service/architecture/ElementDependencyService.java, getDependencyLocations, 0, UNTESTED_ADDITION, 13018, 13099, Mon Dec 14 07:32:52 CET 2015`

```
1 public List<Integer> getDependencyLocations() {
2             return dependencyLocations;
3         }
```

In another part of this study the method got already categorized as not worth testing. It is of course possible to disagree in this answer.

☐ worth testing ☐ not worth testing

In any case: why? In addition: Can you explain why it is untested (e.g. deficient test case; (refactoring of) not CR/feature related method)?

The following methods are "intermediate methods" that got changed or created (and checked in) during the CR but do not exist anymore (maybe refactored in the review process) in the codebase on which the test was executed. The codebase for the test execution is the last revision of CQSE and ConQAT repository of the day the CR got closed.

- `ElementDependencyService.java:concise(List)`

Regarding this CR in total 14 Java methods got changed or added (excluded: intermediate methods that do not exists anymore and tests). 5 methods got not executed in the test case. 0 methods got executed but also get executed in the initialization set and therefore not exclusivly through the test case. We could imagine to calculate the Change Request Coverage in the following ways:

- the naive way: ignore that some methods got executed in the baseline:

$$\frac{\text{executed changed or added methods}}{\text{number of all changed or added methods}} = \frac{9}{14} = 64,286\%.$$

- methods in the baseline do not affect the Change Request Coverage in a negative way:

$$\frac{\text{executed changed or added methods without them executed in baseline}}{\text{number of changed or added methods without them executed in the baseline}} = \frac{9}{14} = 64,286\%.$$

- methods in the baseline do affect the Change Request Coverage in a negative way:

$$\frac{\text{executed changed or added methods without them executed in baseline}}{\text{number of all changed or added methods}} = \frac{9}{14} = 64,286\%.$$

Thank you! ☺

Return filled in sheets to Jakob - hardcopy or via email to rott@cqse.eu.

3

B Besonderheiten im Testverfahren

B.1 Zeitliche Differenz von CR-Schluss und Test

Beim Probedurchlauf einer CR Berechnung wurde eine sehr niedrige CR Coverage festgestellt. Der Sachverhalt wurde untersucht und die Gründe herausgestellt. Getestet wurde auf der `HEAD`-Revision. Die zeitliche Differenz von CR Abschluss (Markierung: grün) und der Testzeit war nicht unerheblich. Das entwickelte Tooling extrahiert die Methodenänderungen aus den einzelnen Commits, die dem CR zugeordnet sind. Unbeachtet bleibt aber, ob die Methoden zwischenzeitlich „umgezogen“ sind. Verlassen die Methoden ihren ursprünglichen Platz, so stimmt der Schlüssel (Pfad zur Klasse, Methodenname und Parametertypen) der Coverageinformation nicht mehr mit dem der geänderten Methoden überein und die Berechnung schlägt fehl.

Um auf der aktuellsten Revision testen zu können, wäre es notwendig das Methoden Changeset eines CRs laufend zu aktualisieren. Bildlich kann man sich das so vorstellen, als würde man zum Zeitpunkt des y. Commits in einem CR x einen Finger auf eine geänderte Methode richten und mit jedem Refactoring (Umbenennung, Umzug in eine andere (Basis)Klasse etc.) den Finger auf den danach aktuellen Standort der Methode verschieben. Würde anschließend auf der neuesten Revision getestet, so wäre der aktuelle Methodenname rückführbar auf den Namen und Ort, der sich im Methoden Changeset des CRs befindet.

Da sich im Tooling keine Methodenhistorie abbilden lässt, musste eine andere Möglichkeit gefunden werden, um dem Problem erfolgreich zu begegnen. Wenn die Rückführbarkeit auf alte Methodenstandorte nicht gegeben ist, dann kann so getestet werden, dass diese gar nicht benötigt wird. Es wurde also dazu entschieden nicht auf der aktuellsten Version getestet, sondern für jeden CR individuell auf einer dafür geeigneten Version. Als geeignete Revision wurde die letzte Revision (beziehungsweise der letzten Revisionen im Falle von Teamscale, das sich zweier unterschiedlicher SVNs bedient) des Tages gewählt, an dem der CR grün markiert wurde. Es wurde dabei angenommen, dass sich die Methoden - wurde der CR am Morgen eines Tages geschlossen - am Abend desselben Tages noch am gleichen Ort befinden. Dies bringt sowohl den Vorteil, dass eine „Umwandlung in das Präsens“ des Methoden Changesets nicht notwendig ist, als auch die Sicherheit, dass die Methoden exakt jene Funktionalitäten implementieren, die ihnen im Rahmen des CRs zugewiesen wurden. Würde man das Changeset nicht „umwandeln“ und eine Methode existierte zu einem späteren Zeitpunkt noch mit demselben Key, aber mit geänderter Funktionalität, so wäre sie doch als gecovert markiert werden, obwohl der Inhalt nicht mehr derselbe ist.

B.2 Speicherschutzverletzungen in Verbindung von Teamscale und Jacoco

Die Ausführungdaten sowohl für die „Base"-Coverage, als auch für die „Testfall"-Coverage, wurden mit Jacoco gewonnen. Die Bibliothek des so genannten Jacoco Agents und ein Ausgabeort für die Datenaufzeichnung können der JVM als Parameter übergeben werden. Jacoco kann direkt in eine Datei aufzeichnen, hält aber auch die Möglichkeit bereit, die Daten im Client-Server Modell abzurufen. Der Agent überwacht die Methodenausführung, schreibt diese aber nicht gleich in eine Datei, sondern bietet eine Serverschnittstelle, über die ein Client die Ausführungsinformation abfragen kann.

Da für den Test tendenziell die gesamte Lebenszeit des Programms von Interesse ist, wurde zunächst die erste Methode bevorzugt. Dafür ist der JVM also die Bibliothek des Jacoco Agenten zu übergeben und ein Pfad, an dem die binäre Ausgabedatei liegen soll. Es stellte sich heraus, dass es zwischen der Ausführung von Teamscale und der Aufzeichnung durch Jacoco Komplikationen gibt. Es kam sehr häufig zu Speicherschutzverletzungen. Sie traten nicht immer, wenn aber nur dann auf wenn Teamscale beendet werden sollte (kill -SIGTERM). Die Fehlermeldung bezog sich meist auf die Bibliothek, die die LevelDB Funktionalität bereithält, und trat vor allem dann auf, wenn Jacoco Coverage erhob. Die Speicherschutzverletzungen hatten zur Folge, dass die Ausführung der JVM noch vor Beendingung des Schreibprozesses von Jacoco angehalten wurde. Die Ausgabedatei war damit nicht vollständig und die Coverageinformation von Initialisierungsset und Test nicht brauchbar.

Es ist unbefriedigend, einen Test durchzuführen, daraus aber keine Coverageinformation erheben zu können. Die generelle Behebung des Fehlers hätte das Problem gelöst. Allerdings kann die Ursache des Fehlers nicht mit vertretbarem Aufwand festgestellt werden; die hätte Konversationen mit den Entwicklern von Jacoco, gegebenenfalls LevelDB und Java benötigt. Auch den Entwicklern von Teamscale ist das Problem bekannt; dies tritt jedoch in der Praxis sehr selten auf. Die Lösung, die das Problem unvollständiger Ausführungsinformationsdateien behebt, gestaltet sich wie folgt: Der Jacoco Agent wurde angewiesen die Ausführungsinformationen im Serverbetrieb bereit zu halten. Nach Durchführung des Tests wurde mit einem kleinen Java Programm die angebotene Information heruntergeladen und in einer `.exec`-Datei gespeichert. Selbst wenn bei der Beendigung von Teamscale dann eine Speicherschutzverletzung auftreten würde, hätte das keinen Einfluss mehr auf die Coverageinformationen. Dieses Verfahren hat allerdings zur Folge, dass (Methoden)Ausführungen, die der Beendigung des Programms zuzuschreiben sind, nicht mehr aufgezeichnet werden. Da keiner der zu untersuchten CRs Änderungen an einem solchen Programmteil beinhaltet, wurde dieser Abstrich in Kauf genommen.

C Relation von Zeitdifferenz zwischen letztem Commit und Schließung eines CRs zu „intermediate“ Methoden

Um die Frage zu klären, wie viel Zeit zwischen dem letzten Commit und der Schließung eines CRs vergeht, wurde die Möglichkeit für eine solche Berechnung in das Werkzeug implementiert. Ursprünglich erwartet wurde, dass die Schließung eines CRs in unmittelbarer zeitlichen Nähe zum letzten CR bezogenen Commit stattfindet. Wie dem Diagramm in Abb. C.1 zu entnehmen ist, vergehen oft einige Tage, manchmal Wochen, bis ein CR nach dem letzten dazugehörogen Commit geschlossen wird. Zu beachten ist, dass die „intermediate“ Methoden neben Methoden, die im Verlauf des CRs erstellt oder geändert und später gelöscht wurden, auch Methoden enthalten, die zwischen CR Schließung und Testung gelöscht wurden oder eine Änderung in ihrer Signatur erfuhren.

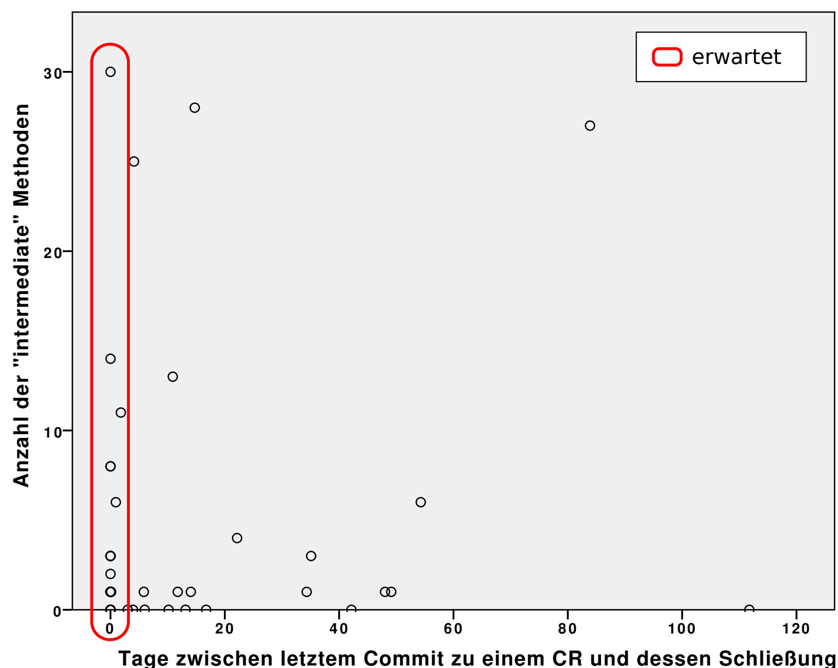


Abbildung C.1: Das Diagramm zeigt, dass nach dem letzten Commit zu einem CR noch eine lange Zeit bis zur Schließung (Markierung als erledigt oder ähnlich) vergehen kann. Die Länge dieses Zeitraums wird hier zusammen mit den „intermediate“ Methoden dargestellt. Beim dargestellten Zeitunterschied handelt es sich um den Betrag der Differenz; vereinzelt kam es bei den getesteten CRs vor, dass nach ihrer Schließung noch Commits getätigt wurden und sich somit eine negative Zeitdifferenz zwischen Schließung und letztem Commit ergäbe.

Literatur

[Boh96] Bohner, Shawn A. "Impact analysis in the software change process: A year 2000 perspective". In: *Software Maintenance 1996, Proceedings., International Conference on*. IEEE. 1996, S. 42–51.

[Dij72] Dijkstra, Edsger W. "Chapter I: Notes on structured programming". In: *Structured programming*. Academic Press Ltd. 1972, S. 1–82.

[Ede+13] Eder, Sebastian – Hauptmann, Benedikt – Junker, Maximilian – Juergens, Elmar – Vaas, Rudolf – Prommer, Karl-Heinz. "Did we test our changes?: assessing alignment between tests and development in practice". In: *Proceedings of the 8th International Workshop on Automation of Software Test*. IEEE Press. 2013, S. 107–110.

[FRC81] Fischer, Kurt – Raji, Farzad – Chruscicki, Andrew. "A methodology for retesting modified software". In: *Proceedings of the National Telecommunications Conference B-6-3*. 1981, S. 1–6.

[HH05] Hassan, Ahmed E – Holt, Richard C. "The top ten list: Dynamic fault prediction". In: *Software Maintenance, 2005. ICSM'05. Proceedings of the 21st IEEE International Conference on*. IEEE. 2005, S. 263–272.

[Jor13] Jorgensen, Paul C. *Software testing: a craftsman's approach*. CRC press, 2013.

[Jue+11] Juergens, Elmar – Hummel, Benjamin – Deissenboeck, Florian – Feilkas, Martin – Schlögel, Christian – Wübbeke, Andreas. "Regression test selection of manual system tests in practice". In: *Software Maintenance and Reengineering (CSMR), 2011 15th European Conference on*. IEEE. 2011, S. 309–312.

[Mal+02] Malaiya, Yashwant K – Li, Michael Naixin – Bieman, James M – Karcich, Rick. "Software reliability growth with test coverage". In: *Reliability, IEEE Transactions on* 51.4 (2002), S. 420–426.

[Mar09] Martin, Robert C. *Clean code: a handbook of agile software craftsmanship*. Pearson Education, 2009.

[MND09] Mockus, Audris – Nagappan, Nachiappan – Dinh-Trong, Trung T. "Test coverage and post-verification defects: A multiple case study". In: *Empirical Software Engineering and Measurement, 2009. ESEM 2009. 3rd International Symposium on*. IEEE. 2009, S. 291–301.

[MW00] Mockus, Audris – Weiss, David M. "Predicting risk of software changes". In: *Bell Labs Technical Journal* 5.2 (2000), S. 169–180.

[Off11] Office, Great Britain. Cabinet. *ITIL Service Transition*. Best management practice. TSO, 2011.

[RH94] Rothermel, Gregg – Harrold, Mary Jean. "Selecting tests and identifying test coverage requirements for modified software". In: *Proceedings of the 1994 ACM SIGSOFT international symposium on Software testing and analysis*. ACM. 1994, S. 169–184.

[She95] Sherlund, Basil Anton. "Logical Modification Oriented Regression Testing". In: *Defect prevention: presentations of the 12th International Conference and Exposition on Testing Computer Software*. 1995, S. 287–303.

[Wil+03] Wilde, Norman – Buckellew, Michelle – Page, Henry – Rajlich, Vaclav – Pounds, LaTreva. "A comparison of methods for locating features in legacy software". In: *Journal of Systems and Software* 65.2 (2003), S. 105–114.

[WS95] Wilde, Norman – Scully, Michael C. "Software reconnaissance: mapping program features to code". In: *Journal of Software Maintenance: Research and Practice* 7.1 (1995), S. 49–62.

[YLW09] Yang, Qian – Li, J Jenny – Weiss, David M. "A survey of coverage-based testing tools". In: *The Computer Journal* 52.5 (2009), S. 589–597.

[ZSK14] Zanjani, Motahareh Bahrami – Swartzendruber, George – Kagdi, Huzefa. "Impact analysis of change requests on source code based on interaction and commit histories". In: *Proceedings of the 11th Working Conference on Mining Software Repositories*. ACM. 2014, S. 162–171.