Bachelorarbeit

Priorisierung von Test-Cases im Test Driven Development

Felix Eschey

Universität Augsburg
Institut für Informatik
Lehrstuhl für Softwaremethodik für verteilte Systeme

Verantw. Professor: Prof. Dr. Bernhard Bauer
Betreuer: Dr. Thomas Eisenbarth
Bearbeitungszeitraum: 09.03.2023 - 09.06.2023

Zusammenfassung. Im Test-Driven-Development werden Regressionstests durchgeführt, um eine Fehlerfreiheit bezüglich aktueller Code-Modifikationen garantieren zu können. Da während der Überarbeitung einer Software dieser Ansatz aufgrund häufig notwendiger Wartung und Erweiterung der Testsuite, mehrmaliger Testsuite-Ausführungen und zusätzlich langen Testsuite-Laufzeiten mit hohen Kosten einhergeht, ist Ziel dieser Arbeit, Test-Case-Priorisierung als mögliche Lösung zur Kostensenkung zu untersuchen. Dabei wird zuerst die web-basierte Problemdomäne der Softwareentwicklung im Arbeitsfeld von makandra dargestellt, anhand deren Kriterien verschiedene Priorisierungsansätze bewertet werden. Auf Basis der Bewertungen wird ein Ansatz ausgesucht, prototypisch implementiert, ausgewertet und auf dessen praktische Limitierungen in Bezug auf die Problemdomäne analysiert.

Eidesstattliche Erklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, alle Ausführungen, die anderen Schriften wörtlich oder sinngemäß entnommen wurden, kenntlich gemacht sind und die Arbeit in gleicher oder ähnlicher Fassung noch nicht Bestandteil einer Studien- oder Prüfungsleistung war.

Augsburg, den 31. Mai 2023

Name

Inhaltsverzeichnis

1 Einführung und Motivation

Im Test Driven Development (TDD) werden Anforderungen an die Software mithilfe der notwendigen Tests definiert und bieten damit eine Grundlage, auf der die jeweilig zugehörigen Anforderung iterativ implementiert werden. Bei der Umsetzung wird somit direkt ersichtlich, ob bereits alle Anforderungen erfüllt wurden, indem die jeweiligen Tests ausgeführt werden. Dies motiviert den Entwickler objektorientierter Sprachen, den Programmcode der benötigten Objekte, deren zugehörige Klassen und die darin zur Verfügung gestellten Funktionen im Hinblick auf die Testbarkeit möglichst modular zu konzipieren, was wiederum in geringer Kopplung einzelner Klassen zueinander resultiert. Folglich lassen sich dadurch die jeweiligen Objekte mit ihrem einhergehenden Aufgabenbereich in einem Unit-Test für jede Funktion einzeln isoliert testen. Dies liegt daran, dass für jeden Test eine minimale Abhängigkeit der dafür notwendigen Objekten und darauf gesetzten Attributen erzeugt werden muss, damit dann nur noch jeweils die einzelne zu testende Funktionalität aufgerufen und mit dem gewünschten Resultat abgeglichen werden muss. Neben erhöhter Testbarkeit führt eine bestmögliche Abkapselung der Softwarebestandteile dazu, dass Code für andere Entwickler verständlicher ist, was neben der Modularität einen wichtigen Bestandteil der Erweiterbarkeit des Quelltexts darstellt.

Da diese Herangehensweise durch Testbarkeit auch eine höhere Testabdeckung und Erweiterbarkeit fördert, sind die Vorteile der testgetriebenen Entwicklung neben der bereits angedeuteten Erhöhung der Softwarequalität [50,55] eine gesteigerte und anhaltende Produktivität der Softwareentwicklung. Die Testabdeckung ist für die Produktivität relevant, da die Tests ein sicheres und zugleich effektives Indiz darstellen, ob die Software nach Änderungen weiterhin funktionsfähig ist. Ein weiterer Aspekt der Produktivitätssteigerung ist, dass die vorausschauende Konzeption des Codes im Hinblick auf die Testbarkeit eine zielgerichtete Umsetzung gewährleistet. All diese Zusammenhänge wurden sowohl empirisch [35, 41, 46] als auch qualitativ untersucht [23] und stellen damit eine wichtige theoretische Grundlage der testgetriebenen Entwicklung dar.

Um beispielsweise Bugs zu fixen oder neue Features umzusetzen, sind Modifikationen am vorhandenen Code im Entwicklungsprozess immer wieder notwendig. Jedoch führen diese Modifikationen häufig dazu, dass vorhandene Tests fehlschlagen und eine Fehler auslösende Änderung indizieren. Somit ist zur Wahrung der Integrität und Funktionalität der bereits implementierten Software eine Neu-Ausführung der Testsuite nach jeder Umsetzung von zusammenhängenden Modifikationen notwendig. Dieser Vorgang der Neu-Ausführung bei Modifikationen wird auch als Regression-Testing (RT) bezeichnet [25]. Allerdings gibt es neben den Vorzügen einer extensiven Test-Strategie auch einige Einschränkungen und Nachteile. So werden 50% der Entwicklungskosten für das Testen benötigt [47], wobei Regressionstesten schon einen Anteil von 80% der Testkosten ausmacht [37]. Diese Kosten entstehen vor allem dadurch, dass vorhandene Tests

häufig nach neuen Änderungen angepasst werden müssen und für das Schreiben neuer Tests meist dieselbe Zeit und Sorgfalt wie das Umsetzen der zugehörigen Code-Überarbeitungen erfordern. Die naheliegendste Methode beim Regression-Testing ist der *retest-all* Ansatz, bei der die komplette Testsuite neu ausgeführt wird. Jedoch hat dies zur Folge, dass erst bis zum Ende der Testsuite ein sicheres Feedback bezüglich möglicherweise verursachter Fehler vorliegt und somit auch durch die Wartezeit Kosten entstehen. Dies bekräftigend haben auch qualitative Studien gezeigt, dass ein Großteil der befragten Unternehmen mit der benötigten Ausführungszeiten des Regressionstestens und einer Kosten-Nutzen-Analyse dieser Praktik unzufrieden sind [34].

Das ist auch der Hintergrund, aus dem das Thema zu dieser Bachelor-Arbeit von der makandra GmbH, einer Firma im Bereich der Web-Softwareentwicklung, herausgegeben wurde. Dabei hat sich das Unternehmen vor allem auf Webentwicklung mit dem Framework Ruby on Rails (RoR) [18] spezialisiert. Als grundlegender Baustein zur Qualitätssicherung steht die angedeutete rigorose Teststrategie im Sinne der testgetriebenen Entwicklung im Vordergrund. Je nach Projekt werden dafür bis zu vier Frameworks für das Testen eingesetzt, nämlich Jasmine [7] für Unit-Tests der mit JavaScript (JS) [8] geschriebenen User-Interface-Komponenten, RSpec [16] für Unit-Tests des serverseitigen Backends in Ruby [19], Cucumber [2] oder neuerdings auch nur RSpec für Integrationstest zur Ansteuerung eines Browsers. Darüber hinaus ist die Anzahl der notwendigen Tests durch die zugenommene Komplexität des Webs [31] über die Jahre angestiegen, was im Fall von makandra überwiegend an dem vermehrten Einsatz von JavaScript-lastigen Frontend zurückzuführen ist. Neben den entstehenden Wartungs- und Entwicklungskosten einer hohen Testabdeckung durch eine umfangreiche Testsuite haben die zur Ausführung der Testsuites erforderlichen Laufzeiten zugenommen. Dies liegt neben der erhöhten Anzahl an Tests durch abzudeckende Komplexität der Anwendung, insbesondere an der Ausführung von Integrationstests, bei denen in vielen Fällen ein Browser mit JavaScript von dem Test-Framework gesteuert werden muss. Hierbei wird bei der Ausführung der Tests ein weiterer Prozess für den Browser gestartet und im Prozess des Test-Frameworks dieser Browser kontrolliert, damit dessen Reaktionen auf das gewünschte Verhalten abgefragt werden können.

Um den Ausführungszeiten wiederum Einhalt zu gebieten, wurde über die Jahre vermehrt Continuous-Integration eingeführt und die Ausführung der Testsuite auf mehrere Prozessorkerne parallelisiert. Da trotz all dieser Maßnahmen die Kosten der vorhandenen Teststrategie immer noch sehr hoch sind, wurde nach weiteren Methoden zur Erhöhung der Effizienz durch die Reduzierung der Testkosten Ausschau gehalten. In der Literatur wurden dafür neben prozessbasierten Untersuchungen [30, 37, 39, 47, 54], drei Techniken hierfür unterschieden, nämlich Test-Case-Selektion, -Priorisierung und -Minimierung. Da die Prozesse bei makandra dafür schon größtenteils optimiert sind, wurde eines der als letzteres genannten Verfahren als weitere Option in Erwägung gezogen. Die Testfall-Minimierung versucht, eine repräsentative minimale Testauswahl zu fin-

den, die alle Anforderungen abdeckt. Ziel ist es, die Anzahl der Tests, deren Ausführung zur Erfüllung der Anforderungen notwendig ist, permanent zu reduzieren. Bei der Testfall-Auswahl wird eine Teilmenge von Tests ausgewählt, um nur die Teile der Software zu testen, die von den aktuellen Modifikationen betroffen sind. Die Test-Case-Priorisierung sortiert die Testfälle nach einer aus vorgegebenen Kriterien berechneten Priorität.

Bezüglich einer Minimierung wird bei der Projektentwicklung bereits durch Durchsicht von Modifikationen eines erfahrenen Entwicklers und Schulung der Entwickler Wert darauf gelegt, dass eine Testsuite minimaler Menge geschrieben wird. So wird beispielsweise bei einer Kombination von zwei Dropdowns nur getestet, dass zwei Werte in Abhängigkeit zueinander gewählt werden können, anstatt alle möglichen Kombinationen zu testen. Zusätzlich würden die dafür notwendigen formalen Definitionen von Anforderungen, deren Vollständigkeit und Eindeutigkeit erst zweifelsfrei bewertbar sein müssten, mit kontinuierlichem Aufwand einhergehen und erst für ältere Projekte nachträglich hinzugefügt werden müssen. Auch wenn die Test-Case-Selektion die notwendigen Kosten reduzieren könnte, gibt es keine absolute Sicherheit, dass jeweils nur betroffene Tests ausgewählt wurden. Auch flackernde Tests, deren Zustand von Ausführung zu Ausführung variiert, müssen nach wie vor sicher erkannt werden können. Dadurch würde eine Neuausführung der Testsuite trotzdem weiterhin notwendig bleiben. Die Test-Case-Priorisierung hat den Vorteil, dass beim Laufen der gesamten Testsuite schneller eine Rückmeldung bezüglich einer fehlerfreien Software-Integrität vorliegt und zugleich alle Tests weiterhin ausgeführt werden können. Ein häufig erforschter Ansatz ist, nach potentiell fehlerhaften Tests bezüglich der aktuellen Änderungen zu priorisieren, was für die Intention des Regressionstestens von besonderer Bedeutung ist [56]. Daraus folgend ist das Ziel dieser Arbeit zu untersuchen, welche Verfahren der Test-Case-Priorisierung sich im Rahmen der eingesetzten Technologien und der Praxis der testgetriebenen Entwicklung eignen. Damit dies möglich ist, muss die Problemdomäne charakterisiert werden, um davon ausgehend Kriterien abzuleiten, welche von einer Software zur Priorisierung im Arbeitsumfeld von makandra erfüllt werden müssen. Auf Basis der herausgearbeiteten Kriterien können die verschiedenen Ansätze dahingehend bewertet werden, ob sie diese erfüllen. Im Hinblick auf die Frage der Praktikabilität und Effizienz wurde dann ein Ansatz ausgesucht, prototypisch implementiert und mithilfe von historischen Fehlerdaten aus einem Projekt von makandra exemplarisch ausgewertet.

2 Hintergrund

In diesem Abschnitt werden relevante und grundlegende Begriffe des Themengebiets eingeführt und formal definiert. Dabei spielen sowohl grundlegende Themen der Softwareentwicklung im Web sowie Aspekte und Techniken des Regressionstestens eine Rolle.

2.1 Regressionstesten

Unter Regressionstesten versteht man die Praxis, dass nach jeder Veränderung der Codebasis die vorhandene Testsuite laufen gelassen werden muss, damit vorhandene Funktionalitäten der Software erhalten bleiben. Im Folgenden werden kurz die wichtigsten Ansätze zur Veränderung der für Regressionstests benötigten Testsuite eingeführt.

2.1.1 Test-Case-Minimierung Bei der Test-Case-Minimierung wird versucht, eine Repräsentation der Testsuite zu finden, welche eine Reihe von Anforderungen an die Software erfüllt. Die Menge der dafür ausgewählten Tests soll so beschaffen sein, dass ein enthaltener Test mindestens eine der Anforderungen abdeckt und eine Anforderung höchstens durch einen Test abgedeckt wird. Anders gesagt ist die Absicht, die minimale Menge aller erfüllenden Testfälle für eine Reihe von Anforderungen zu finden. Da dieses Problem dem NP-vollständigen Hitting-Set-Problem entspricht, müssen Verfahren zum Finden einer minimalen Testmenge auf eine heuristische Herangehensweise zurückgreifen. Oft wird eine Minimierung verwendet, um redundante Testfälle zu erkennen und zur Minimierung der Testsuite dauerhaft zu entfernen [43].

2.1.2 Test-Case-Selektion Unter Test-Case-Selektion versteht man, dass eine Teilmenge an Tests aus der Testsuite ausgewählt wird. Meistens ist dabei die Absicht, nur Teile einer Software, welche von den aktuellen Modifikationen betroffen sind, zu testen [43].

2.1.3 Test-Case-Priorisierung Bei der Tast-Case-Priorisierung werden die Tests der Testsuite so sortiert, dass Tests mit höherer Priorität früher innerhalb der Testsuite ausgeführt werden. Ziel der Priorisierung ist jeweils, dass die Verwendung dieser zu einer verbesserten Fehlererkennungsrate bezüglich bestimmter Kriterien führt. Um eine mögliche Priorisierung mit einer anderen zu vergleichen, werden unterschiedliche Metriken zur Quantifizierung und Evaluierung der Performanz einer Testsuite-Sortierung verwendet [43]. Formell ist eine Test-Case-Priorisierung nach Yoo und Harman [56] wie folgt definiert:

Definition 1 (Test-Case-Priorisierung). *Gegeben sei eine Testsuite, T, sei PT die Menge der Permutationen von T und sei f definiert als Funktion $f : PT \to \mathbb{R}$ Dann entspricht $f(T'') \geq f(T')$ einer Test-Case-Priorisierung einer Testsuite für $T', T'' \in PT$ mit $T' \neq T''$.*

2.2 Ansätze der Test-Case-Priorisierung

In der Test-Case-Priorisierung haben sich verschiedene Herangehensweisen etabliert, wobei die größten Vertreter jeweils Coverage-, Anforderungs-, Such-, Fehler- und Geschichtsbasiert sind. Andere seltener verwendete Ansätzen basieren auf Modellen, Kosteneffizienz, kombinieren mehrere Priorisierungskriterien, beispielsweise mithilfe von genetischen Algorithmen oder Machine-Learning, bayesschen Netzen oder stellen eine Kombination von Ansätzen und Verfahren dar [38, 51]. Machine-Learning basierte Verfahren können darüber hinaus mehrere Parameter miteinbeziehen und fügen oft weitere Parameter wie wie textuelle Daten des Codes, Code-Komplexität oder Benutzereingaben mit ein [45].

2.2.1 Coverage-basierte Test-Case-Priorisierung Unter Coverage versteht man eine Metrik, die in Prozenten ausdrückt, wie viele Anteile eines Programmcodes durch eine Testsuite ausgeführt werden, also abgedeckt sind. Die häufigsten Zählweisen der Abdeckung sind dabei Funktions-, Statement-, Branch- und Condition-Coverage. Dabei soll durch die Coverage Rückschluss gezogen werden können, ob eine Testsuite zu einer hohen Fehlererkennungsrate führt. Analog dazu ist bei der coverage-basierten Priorisierung die Annahme, dass eine frühe Gesamtabdeckung der Testsuite-Ausführung zu einer schnelleren Fehlererkennungsrate führt [56]. Entsprechend einer gewünschten Priorisierbarkeit muss die Abdeckung dann für jeden Test einzeln aufgezeichnet werden. Diese Methoden stellen mit 15% nach wie vor einen der am meisten untersuchten Ansätze dar [43].

Geläufige Unterscheidungsmerkmale der einzelnen Algorithmen sind neben der Granularität der Coverage-Metrik auch, ob die Testsuite nach *total* oder *additional* Coverage priorisiert wird. Bei der *total* Coverage wird nach den absoluten Coverage Werten zu den betroffenen Zeilen unterschieden, während bei der *additional* Coverage jeweils iterativ versucht wird, maximale Coverage über das gesamte Programm möglichst schnell zu erreichen. Diese Algorithmen werten zur Berechnung der Priorisierung entweder die erreichte Coverage der aktuellen Modifikationen an Dateien oder anhand des gesamten Codes aus [51]. Es wird im Rest der Arbeit von Priorisierung nach absoluter oder additionaler Coverage gesprochen, wenn *total* respektive *additional* Coverage-Verfahren gemeint sind.

Ein großer Vorteil ist, dass Abhängigkeiten direkt durch den vorliegenden Code und dessen Ausführung analysiert werden können. Eine Schwachstelle ist, dass es nur die Abdeckung der Testfälle mit einbezieht, womit es zwar ein hinreichendes Merkmal zur Fehlererkennung ist, jedoch damit auch kein derat spezifisches Wissen wie die Fehleranfälligkeit eines Testfalls oder den logischen Inhalt der aktuellen Änderungen zur Priorisierung beitragen kann [33, 51].

2.2.2 Fehlerbasierte Test-Case-Priorisierung Bei diesem Ansatz wird jedem Test ein Fehlerentdeckungspotenzial (FEP) zugeordnet. Das heißt, dass für eine

Menge von Fehlern jeder Test eine gewisse Wahrscheinlichkeit hat, einen Fehler zu erkennen. Meist werden dafür gezielt sogenannte Mutanten, also eine syntaktische Modifikation des Programm-Codes, mit der Absicht, einen Fehler zu erzeugen, im Programmcode eingebaut. Daraufhin kann für jeden Test berechnet werden, welches Potenzial dieser Test hat, einen Fehler zu erkennen. Dabei muss eine bekannte Menge an Fehlern vorausgesetzt werden und Fehlerkennungsverhalten oft ergänzend auf Merkmale ähnlicher Fehlergruppen durch linguistische Datenverarbeitung zurückgeführt werden. Es wird auch wiederum zwischen FEP-*total* und FEP-*additional* unterschieden. Ebenso kann der FEP-Wert für verschiedene Granularitäten der Programmabdeckung berechnet werden. Die Schwierigkeit liegt hierbei vor allem darin, dass eine relevante und repräsentative Menge an Fehlern vorliegen oder erzeugt werden muss [43, 56].

2.2.3 Geschichtsbasierte Test-Case-Priorisierung Die historischen Ausführungsdaten einer Testsuite werden zur Priorisierung auf relevante Daten wie Kosten, vergangene Fehler, Häufigkeit der Fehlererkennung analysiert, um als Eingabe zur Priorisierung verwendet werden zu können. Viele Verfahren verwenden das als Grundlage, um das Fehlerentdeckungspotenzial und Ausfallwahrscheinlichkeiten für bestimmte Änderungen zu bestimmen oder um eine Cluster-Analyse für Eingabevektoren aus diesen Daten durchzuführen. Die größte Limitierung dieses Ansatzes ist, dass genügend aussagekräftige historische Daten vorliegen müssen [38, 43].

2.2.4 Anforderungsbasierte Test-Case-Priorisierung Einer oder eine Reihe von Tests können als Stellvertreter für Anforderungen an die Software aufgefasst werden, da die Funktionalität einer Anforderung in der aktuellen Software-Version über den Test abgefragt und auf Korrektheit überprüft wird. Dies dient dazu, fehlerhafte Zustände und damit auch nicht mehr erfüllte Anforderungen der Software optimalerweise bei Durchführung der Regressiontests zu erkennen. Dieser Ansatz der Priorisierung hat die Absicht, die Test-Cases den jeweiligen Anforderungen zuzuordnen, welche Sie abdecken und nach der Relevanz der Anforderungen zu priorisieren. Die Anforderungen können etwa kunden-, sicherheits-, funktions- oder implementationsspezifische Aspekte umfassen.

Demnach muss jeder Anforderungen auch eine unterschiedliche Relevanz zugeordnet werden. Wenn mehrere Anforderungsklassen in Erwägung gezogen werden, so muss eine eindeutige Gewichtung innerhalb der Klassen und der Klassen zueinander definiert werden. Da es kaum möglich ist, die Vollständigkeit einer Anzahl von Anforderungen und deren Bedeutung zu bestimmen, liegt hierin auch die größte Schwachstelle. Dies hat zur Folge, dass die notwendigen Anforderungen und deren Abstimmung zueinander oft subjektiv geschätzt werden und damit leicht einer Fehleinschätzung unterliegen oder sogar je nach Anliegen der Ausführung eine andere Einschätzung notwendig ist [43, 51].

2.2.5 Suchbasierte Test-Case-Priorisierung Die Klasse von Algorithmen verwendet meist eine der bisher definierten Kriterien, wie Coverage, Kosten oder FEP, oder kombiniert diese mithilfe von Heuristiken. Der am meisten untersuchte Ansatz sind dabei gierige Suchverfahren, welche oft in Zusammenhang mit additionalen Coverage-Verfahren kombiniert wurden. Weitere Möglichkeiten sind Ameisenkolonien, String-Distance oder genetische Algorithmen.

String-Distance-Verfahren können verwendet werden, um die textuelle Ähnlichkeit zwischen zwei Texten zu berechnen. Die Ähnlichkeit ist bei diesen Verfahren dadurch definiert, wie viele Änderungen einzelner Zeichen minimal notwendig wären, um einen Text in einen anderen zu überführen.

Bei diesem Verfahren ist es naheliegend, dass die Ähnlichkeit des modifizierten Programm-Codes zu Testfällen berechnet wird, wobei eine höhere Ähnlichkeit einer höheren Priorität entspricht. Ameisenkolonien sind eine Optimierungsmethode für Graphen. Dies ist analog zu dem Vorgang in der Natur zu verstehen, in dem Ameisen versuchen, für sie attraktive Wege, um zum Beispiel Nahrung zu beschaffen, zu finden und diese durch Pheromon-Botenstoffe für andere Ameisen zu markieren. Dementsprechend werden bei diesem Verfahren virtuelle „Ameisen“ parallel durch den Graphen geschickt und die Gewichte mit jeder Iteration nach ihrer aktuell festgestellten Attraktivität aktualisiert. Die Graphenstruktur können beispielsweise Pfade zwischen Anforderungen und den einzelnen Testfällen sein. Dabei muss festgelegt werden, nach welchen Kriterien die dem Pfad zu gehörige Attraktivität berechnet wird und wie diese miteinander kombiniert werden.

Genetische Algorithmen sind ebenfalls Optimierungsverfahren, die jedoch versuchen, die Evolution natürlicher Organismen als algorithmisches Verfahren nachzubilden. Dabei werden mehrere Populationen gebildet, die dann nach ihrer berechneten Fitnessfunktion selektiert werden. Im nächsten Schritt können durch Rekombination der Übriggebliebenen neue Populationen gebildet und mit Hilfe von Mutationen zufällig verändert werden. Dieser Vorgang wird solange wiederholt, bis ein Abbruchkriterium erreicht ist. Bei der Test-Case-Priorisierung entspricht eine Population einer Testsortierung, wobei die Güte der einzelnen Priorisierung durch die Fitnessfunktion, welche verschiedene Kriterien der Tests mit einbeziehen kann, bestimmt wird [38, 43].

2.2.6 Modellbasierte Test-Case-Priorisierung Das Ziel dieses Ansatzes ist es, eine symbolische Ausführung des Programms zu bewerkstelligen und zur Priorisierung zu nutzen. Für viele dieser Verfahren stellt statische Code-Analyse eine wichtige Methode dar, um ein Ausführungsmodell der Software zu erlangen. Als Grundlage dieser Modelle werden häufig Datenflussanalyse und Kontrollflussgraphen wie ein Aufrufgraph hergenommen. Es ist auch möglich, dass Diagramme mit weiteren Informationen über das Programm und dessen Ausführung in komplexeren Zustandsgraphen dargestellt werden.

Der große Vorteil ist, dass der Code in einfachen Varianten nur analysiert und nicht ausgeführt werden muss. Dasselbe gilt für die Priorisierung, die anhand des Modells über mögliche Ausführungspfade der Testfälle eine Priorisierung berechnet. Dieser Pfad kann beispielsweise von aktuellen Modifikationen ausgehend berechnet werden. Ein einfacheres Verfahren zur modellbasierten Priorisierung ist davon ausgehend, die Testfälle danach zu priorisieren, wie oft sich die Pfade von den modifizierten Dateien zu den Testfällen schneiden [38, 43, 56].

2.3 Webentwicklung

2.3.1 Revisionskontrolle mit Git Die eingesetzte Revisionskontrolle, oder zweideutig auch manchmal Versionskontrolle genannt, gibt vor, wie und zu welchen Zeitpunkten eine partielle Aktualisierung des verwendeten Ansatzes zur Priorisierung und der dafür zugrunde liegenden Daten notwendig ist. Im historischen Kontext betrachtet wird in der Softwareentwicklung heutzutage vor allem Wert auf Revisionskontrolle im Gegensatz zu einer Versionskontrolle gelegt.

Dabei wird bei einer Versionskontrolle der Fokus hauptsächlich auf die aktuelle Version einer Software und den zugehörigen Stand der Dateien gelegt, während es bei der Revisionskontrolle einzelne zusammenhängende Überarbeitungen sind. Die Unterscheidung kommt darin zum Ausdruck, dass bei der Revisionskontrolle historische Momentaufnahmen des Programmcodes in einem direkten azyklischen Graphen angeordnet werden und bei der Versionskontrolle unterschiedliche Versionen immer in einem linearen Graph aufeinander folgen. Der große Vorteil der Revisionskontrolle ist, dass die Veränderungen der Software in feingliedrigen Schritten zurückverfolgt werden und mehr als eine Version gleichzeitig existieren kann. Dadurch können Entwickler getrennt voneinander auf unterschiedlichen Versionen der Software arbeiten, ihre Revisionen trotzdem eindeutig zusammenführen oder sogar selektiv Änderungen aus einer anderen Version verwenden [52].

Ein dafür besonders häufig verwendetes Tool, welches auch von makandra eingesetzt wird, ist das Kommandozeilen-Tool Git, für welches aber auch grafische Interface-Lösungen existieren. Eine Reihe von zusammenhängenden Veränderungen wird als ein Knoten im Graphen dargestellt und dieser wird in Git auch als Commit bezeichnet. Jeder neue Commit wird an den vorherigen angehängt. Abzweigungen im Graphen, oder auch Branches genannt, umfassen einen Pfad von direkt aufeinander folgenden Commits bis zum ersten Commit, an dem dieser Pfad sich von einem anderen abgezweigt hat. Branches können wiederum eindeutig in der Historie zusammengeführt und nach Belieben gewechselt werden.

Dementsprechend wird für die Umsetzung einer Funktionalität der Software oder einer fest definierten Folge von Aufgaben, was auch das Beheben eines Fehlers oder ein Versions-Upgrade eines verwendeten Frameworks sein kann, jeweils ein eigener Branch verwendet. Der Stand der lokalen Versionierung entspricht immer dem letzten Commit des aktuellen Branches. Bei der Programmierung werden

lokale Veränderungen des Arbeitsverzeichnisses von Git erkannt und können nach Bedarf vom Entwickler dem Index hinzugefügt werden. Der Index ist ein Zwischenspeicher, der von Git genutzt wird, die aktuellen Modifikationen als Differenz zum letzten Commit für den aktuellen Commit zu indizieren. Alle aktuellen Änderungen des Indexs werden schließlich über einen Befehl zu einem Commit zusammengefasst, der an den zum Branch zugehörigen letzten Commit angehängt wird.

Damit derartige Vorgänge funktionieren, benötigt Git eine komplexe Datenstruktur, dem Repository, auf deren Basis eine Vielfalt von Metadaten, wie etwa eine Graphenstruktur, die Differenz zwischen Code-Versionen oder die aktuelle Position im Graphen, gepflegt werden müssen. Das Repository wird von Git mitsamt allen Versionen der Programmdateien dabei sowohl lokal als auch zur Absicherung auf einem Server hinterlegt, was für den Entwicklungsprozess vielfältige Vorteile und Flexibilität entstehen lässt. Diese sind unter anderem, dass der Entwickler immer wieder den aktuellen Stand vom Server abfragen und herunterladen, selbst seine Änderungen hochladen, Änderungen verwerfen, für spätere Wiederverwendung in einen Zwischenspeicher ablegen oder selektiv einen Commit aus einem anderen Branch anwenden kann. Der Vorgang des Abrufs und Herunterladens wird auch Auschecken bezeichnet, während der Vorgang des Hochladens als Einchecken bezeichnet wird.

2.3.2 Continuous-Integration und automatisierte Auslieferung Bei diesem Verfahren der Softwareentwicklung steht die stetige automatisierte Integration neuer Modifikationen der Software im Vordergrund. Grundlage dafür ist, dass eine Versionsverwaltung verwendet wird, in dem der Entwickler regelmäßig seinen Commit eincheckt. Der Vorgang des Eincheckens triggert dann die Continuous-Integration (CI), welche neben Regressionstests meist auch weitere Schritte wie automatisiertes Kompilieren, Aufsetzen und Migrieren der Datenbank, statische Code-Analyse im Hinblick auf den Code-Style, -Design oder Sicherheitslücken umfasst. Unter der Integration versteht man einerseits, rechtzeitig fehlerhaftes Verhalten zu erkennen und andererseits die Qualität der Software zu verbessern und zu wahren. Ein weiterer Vorteil dieses Verfahrens ist, dass Abhängigkeiten der Software und der Tests isoliert definiert werden müssen, so dass durch plattform- oder systemspezifische Details verursachte Probleme schneller erkannt werden können. Dadurch, dass Regressionstests seltener lokal ausgeführt werden müssen, kann hier durch bessere Skalierbarkeit der Server eine schnellere Rückmeldung über das Resultat des Regressionstests vorliegen.

CI wird auch als erster grundlegender Schritt einer automatisierten Auslieferungsstrategie aufgefasst. In der Webentwicklung ist es daher üblich, dass bis zu drei Abzweigungen als Hauptstränge gepflegt werden. In einer einfacheren Variante repräsentiert ein Strang die Entwicklungsumgebung und in einem anderen Strang die Produktivumgebung. Man spricht dabei von einer Umgebung, da sich beispielsweise der verwendete Server, die verwendete Datenbank und

die darin hinterlegten Daten je nach Umgebung unterscheiden. Ein wichtiger Vorteil diesbezüglich ist, dass damit beim Testen einer Anwendung in der Entwicklungsumgebung keine Inkonsistenzen für den Endnutzer der Produktivumgebung entstehen.

Dies impliziert, dass die Arbeitsweise so angepasst wird, dass Entwickler abgeschlossene Commits dem Strang der Entwicklungsumgebung anhängen, wobei der Strang der Produktivumgebung in regelmäßigem Abstand auf den Stand der Entwicklungsstrang aktualisiert wird. Continuous-Deployment (CD) ist die darauffolgende Phase einer automatisierten Auslieferungsstrategie und ermöglicht es Entwicklern, Änderungen im Quellcode analog zu CI automatisch zu überprüfen. Sofern alle Kriterien der CI und der CD erfüllt sind, werden die Änderungen dann für die Produktivumgebung freigegeben. Die beiden Phasen werden zusammen kurz als CI/CD bezeichnet. Die neuen Stände des Produktivstangs werden mit den neuen Commits ausgeliefert, so dass diese dann für den Endnutzer im Browser verfügbar sind. Technisch bedeutet dies, dass der Server der Webanwendung jeweils auf dem Stand des letzten Commits des ihm zugehörigen Strangs ausgeführt wird [42].

2.3.3 Flackernde Tests führen bei der Ausführung desselben Programmcodes zu unterschiedlichen Ergebnissen. Damit kann für diese Tests nicht mehr mit Sicherheit garantiert werden, dass die zugrunde liegende Funktionalität noch intakt ist. Da dies in der Praxis in vielen Fällen mit einer hohen Wartungsintensität einhergeht, sind flackernde Tests ein weiterer Flaschenhals für Regressionstests und die testgetriebene Entwicklung. Dies liegt mitunter daran, dass die Ursachen für flackernde Testergebnisse vielfältig, kontextgebunden und somit nicht immer eindeutig auffindbar sein können. Die häufigst gefundenen Gründe für flackernde Testzustände sind Nebenläufigkeit, asynchrone Warteaufrufe, die Verwendung von Zufälligkeit, Testreihenfolge- und Plattformabhängigkeiten [32, 40].

Bei makandra sind flackernde Testzustände vor allem in Zusammenhang mit Integrationstests eine wiederkehrende Problematik. Bei diesen Integrationstests wird die Software mithilfe eines Browsers mit oder ohne Javascript so getestet, wie sie vom Endnutzer verwendet wird. Da die Verwendung von JavaScript im Browser höhere Ausführungskosten mit sich bringt, wird JavaScript generell nur für Integrationstests aktiviert, für deren abgefragtes Verhalten die Ausführung von JavaScript-Code notwendig ist. Dies liegt mitunter auch daran, dass diese Tests in der Praxis mit am häufigsten flackernde Testzustände erzeugen. Die Hauptgründe dafür sind sind in diesem Zusammenhang Nebenläufigkeit und asynchrone Warteaufrufe und sind darauf zurückzuführen, dass der über Selenium [21] ausgeführte Chrome-Browser in einem eigenen Prozess gesteuert wird, während das Testskript und der Server in zwei Threads innerhalb eines weiteren Prozesses kontrolliert werden. Hierbei kann es beim Laden des Dokument-Objekt-Modells (DOM) eines HTML-Dokuments, einer Baumstruktur, die die Anordnung aller

HTML-Elemente einer Seite im Web definieren, zu Synchronisierungsproblemen kommen.

Dies liegt daran, dass beim Aufbau des DOM im Browser unterschiedliche Ladezeiten auftreten können, während das Testskript bereits versucht, auf diese Elemente zuzugreifen. Dies kann häufig auch passieren, wenn der als Testskript simulierte Endnutzer durch Interaktion mit einer Seite weitere JavaScript-Funktionen triggert, welche partielle Aktualisierungen der Seite nach sich ziehen. Insbesondere kann das dann auftreten, wenn durch die Interaktion zusätzlich asynchrone Warteaufrufe an den Server für benötigte Daten ausgelöst werden. Wenn auch die Anfälligkeit für flackernde Tests durch interne Dokumentation der Ursachen und das eigens geschriebene Gem Capybara-Lockstep [5] zur Erhöhung der Synchronisierung bereits auf ein Minimum reduziert werden, so kommt in den Projekten dennoch regelmäßig vor, dass Testsuite-Durchläufe flackernde Tests enthalten und Wartungsarbeiten dafür notwendig sind. Ein Gem ist eine in Ruby geschriebene Bibliothek, welche sich auf eine Reihe von zusammenhängenden Funktionalitäten beschränkt, damit diese in Projekte eingebunden werden können.

2.3.4 Arbeitsweise zur Umsetzung von Code-Modifikationen Der gewählte Ansatz der Test-Case-Priorisierung sollte mit der Art, wie Modifikationen mit Hilfe von Git und CI bei makandra umgesetzt werden, kompatibel sein. Da daraus folgend weitere Kriterien für den Ansatz der Priorisierung noch abgeleitet werden, soll der typische Vorgang einer Softwaremodifikation hier dargestellt werden.

Der Entwickler schreibt nach optionaler Einarbeitung in die aktuelle Aufgabe, zuerst die notwendigen Tests und beginnt dann die Funktionalität im Code umzusetzen. Zur Überprüfung der Funktionalität werden während der Umsetzung entweder die neu geschriebenen Tests oder spezifische Tests, welche die Modifikationen betreffen, erneut ausgeführt. Ebenso wird bei der Entwicklung mithilfe eines lokalen Servers das neu umgesetzte Feature der Software manuell getestet. Die Verwendung dieses Testservers liefert dem Entwickler dabei schon erste Indizien, ob die Anwendung an sich instand bleibt, für den Vorgang notwendige User-Interface-Elemente weiterhin funktionieren und die Umsetzung der aktuellen Code-Anpassungen korrekt sind.

Sobald sich die Umsetzung der Aufgabe dem Ende nähert, wird der Entwickler im Sinne des Regressionstestens die gesamte Testsuite lokal oder mithilfe von Continuous-Integration so oft durchführen, wie zum Beheben aller Fehler notwendig ist. Wenn das Projekt CI verwendet, wurden schon durch das Einchecken inkrementeller Commits erste Regressionstests getriggert. Dies kann auch lokal vor allem dann der Fall sein, wenn die Aufgabe durch aktuelle Modifikationen große Teile der Anwendung betrifft und damit vorher die Durchführung der Regressionstests angeraten ist.

Bezüglich der Versionierung ist der Arbeitsablauf im Normalfall, dass der Entwickler einen eigenen Branch nach dem letzten aktuellen Stand des Hauptstrangs der Entwicklungsumgebung im Git-Repository erstellt und dann beginnt, lokal in seinem Arbeitsverzeichnis daran zu arbeiten. Für größere Änderungen kommt es bei der Umsetzung einer Aufgabe häufig vor, dass der Entwickler zur Absicherung mehrere Commits in das Repository eingecheckt hat. Um eine lineare Versionierung zu erreichen, werden Commits für eine Änderung nach Abschluss aller Modifikationen zu einem Commit zusammengefasst und als ein einziger Commit an den Hauptstrang der Entwicklungsumgebung angehängt. Dafür muss der Commit vor dem Anhängen erst auf den aktuellen Stand der Entwicklungsumgebung aktualisiert werden. Dies kann schon während oder auch zum Ende der Implementierung erfolgen.

3 Bewertung zur Auswahl eines Ansatzes

3.1 Problemdomäne

Webbasierte Software ist in den vergangenen Jahren immer mehr zu einer der wichtigsten Plattformen der Softwareentwicklung geworden. Dies liegt vor allem an der nativen Unterstützung vieler Betriebssysteme sowie moderner Cloud-Technologien, welche schnelle Entwicklungszyklen und simultane Wartung dank direkter Auslieferung durch CI/CD ermöglichen, wie in Abschnitt 2.3.2 ausgeführt wurde. Nicht zuletzt hat das dazu geführt, dass immer mehr Enterprise-Softwarelösungen in das Web ausgelagert wurden. Dies bedeutet auf der einen Seite, dass Software mit erhöhten Testanforderungen im Web entwickelt werden und gleichzeitig, dass Regressionstesten in besonders hoher Frequenz durch schnelle Auslieferungzyklen notwendig ist [57].

Dies ist auch das Umfeld in dem makandra tätig ist und es hat sich in der Praxis bestätigt, dass Regressionstesten mit der größte Kostenfaktor bei der inkrementellen Auslieferung ist. Bezüglich des in 2.3.4 dargestellten Arbeitsablaufs einer Code-Modifikation, entsteht für den Entwickler durch Regressionstests, welche bei Projekten oft eine Zeitspanne von 10 bis zu 30 Minuten füllen, ein Leerlauf, der einen Kontextwechsel erfordert oder für die Weiterentwicklung auf Basis der aktuellen Veränderungen eine Unsicherheit erzeugt, die damit auch den Prozess verlangsamt oder sogar kurzzeitig stoppt. Insbesondere ist das dann der Fall, wenn die Änderungen viele Zeilen des Programmcodes betreffen und die Regressionstests bei Fehlern dadurch tendenziell mehr als einmal ausgeführt werden müssen. Durch die Größe der Software und das Arbeiten an vielen Projekten gleichzeitig kann es nämlich für den Entwickler schwer abschätzbar sein, welche Funktionalitäten und die dafür geschriebenen Tests der Software durch aktuelle Änderungen potentiell fehlerhaft sein könnten.

Um flackernde Testergebnisse, die durch Testreihenfolgenabhängigkeiten verursacht werden, zu erkennen und beheben zu können, wird bei makandra aktuell eine zufällige Priorisierung der Testsuite eingesetzt. Dies kann beispielsweise auftreten, wenn in zwei Tests bestimmte Daten erwartet werden, die nicht unabhängig bei Ausführung der einzelnen Tests hinterlegt werden oder das Zurücksetzen der Datenbank zwischen den Tests nicht rechtzeitig geschieht. Da bei der zufälligen Priorisierung eine größere Streuung aller Tests über die Testsuite erreicht wird, verschafft eine zufällige Ausführung einen kleinen Vorteil gegenüber gar keiner Priorisierung [49].

3.2 Kriterien

Regressionstests werden ausgeführt, um Fehler verursachende Änderungen der Software ausfindig zu machen, was daran liegt, dass nicht immer alle Ausführungspfade der Software für den Entwickler direkt ersichtlich sind. Dies bedeutet in der Praxis, dass bei der Ausführung der Testsuite mindestens solange gewartet werden muss, bis der erste fehlerhafte Test mitsamt der auslösenden Programmzeile vorliegt. Sobald der Fehler vorliegt, müssen die nicht mehr funktionierenden Ausführungspfade der Software durch weitere Modifikationen des Codes wieder mit einbezogen werden. Diese Modifikationen bedeuten häufig, dass ergänzende Logik notwendig ist oder ein alternativer Lösungsansatz gewählt werden muss.

Folglich erlaubt eine Priorisierung, welche über fehlerhafte Testzustände früher eine Rückmeldung gibt als zufällige Priorisierung, dass Kosten aufgrund kürzerer Wartezeiten reduziert werden können. Dafür ist es notwendig, dass sich die Priorisierung auf aktuelle Modifikationen bezieht und den Tests eine höhere Priorität zuordnet, die von aktuellen Modifikationen fehlerhaft betroffen sein könnten.

Da gewöhnlicherweise Modifikationen des Codes mit Hinzufügen neuer Tests oder Überarbeitung vorhandener Tests einhergehen, ist es erforderlich, dass der eingesetzte Ansatz neu hinzugefügte oder überarbeitete Tests eindeutig in die Priorisierung mit einbeziehen kann, so dass diese eine im Kontext der aktuell vorgeschlagenen Priorisierungen eine sinnvolle und höhere Priorität zugewiesen bekommen.

Damit ein Priorisierungsverfahren eine Priorisierung angeben kann, muss eine Datenstruktur verwaltet werden, die für die Priorisierung relevante Daten enthält. Da sich durch Code-Modifikationen die Software stetig weiterentwickelt, muss es auch möglich sein, diese Datenstruktur effizient, selektiv und dynamisch auf die aktuelle Version der Software zu aktualisieren. Zusätzlich ist zu bedenken, dass die hinterlegten Daten mit der Verwendung der Versionsverwaltung kongruent sein müssen. Dafür müssen sich die Daten immer auf den Stand des aktuellen Branch beziehen, neue Modifikationen durch Hinzufügen neuer Commits oder das Verändern vorausgehender Commits korrekt erkannt und aktualisiert werden.

Darüber hinaus gilt, dass für das Aufzeichnen und Zusammenführen der Priorisierungs-Daten keine Testsuite-Durchläufe nur dafür erforderlich sein sollten. Dies liegt daran, dass der entstandene Mehraufwand vom Entwickler leicht vernachlässigt werden könnte, wenn sich die Kosten-Nutzen-Analyse durch zusätzliche Testsuite-Laufzeiten kaum von vorheriger Variante unterscheidet.

Um sich möglichst reibungslos in den vorhandenen Arbeitsablauf einzufügen, sollten die Information der während der Entwicklung laufenden Tests oder des lokalen Testservers zur Aktualisierung des eingesetzten Verfahrens inkorporiert werden, damit diese dann mitsamt den Modifikationen zu einer vorgeschlagenen Priorisierung verarbeitet werden können. Optimalerweise könnten auch Daten aus der CI in Erwägung gezogen werden, da für bestimmte Projekte bei makandra die Testsuite aufgrund hoher Ausführungszeit oftmals lokal nur noch selten ausgeführt wird.

Zusammenfassend sind die Kriterien folglicherweise:

- Es sollten aktuell fehlerhafte Tests früher als mit randomisierter Priorisierung ausfindig gemacht werden können.
- Tests, die von aktuellen Modifikationen betroffen sind, sollten höher priorisiert werden, so dass potenziell fehlerhafte Programmabschnitte schneller gefunden werden.
- Das Verfahren bezieht sich immer auf die aktuell ausgecheckte Version der Software, so dass bei Änderungen neue Tests und Funktionen mit einbezogen werden, indem die Priorisierungs-Daten effizient aktualisiert werden können.
- Neu hinzugefügte Tests sollten eindeutig und sinnvoll im Kontext der aktuellen Priorisierung höher priorisiert werden.
- Es sollte mit dem Entwicklungsprozess kompatibel sein, so dass es sich mit den daraus resultierenden Informationen im Hintergrund selbst aktualisieren kann und keine zusätzlichen Testsuite-Durchläufe erfordert.

3.3 Bewertung

Im Folgenden soll darauf eingegangen werden, welche der vorgestellten Ansätze nach den herausgearbeiteten Kriterien der Problemdomäne in Frage kommen und weshalb.

3.3.1 Coverage-basierte Test-Case-Priorisierung Die Abdeckung des Programmcodes erfüllt die wichtigsten Voraussetzungen, da diese ein sicheres Kriterium darstellt, Tests durch die darin implizierten Ausführungpfade zu selektieren und zu priorisieren. Zusätzlich hat Coverage den Vorteil, dass diese zu jederzeit für hinzugefügte Tests aufgezeichnet werden kann und sich so mit vergangenen Informationen kombinieren lässt. Das ist darauf zurückzuführen, dass für die

Aufzeichnung der Coverage lediglich eine eindeutige Zuordnung zwischen den jeweiligen Tests und dem abgedeckten Programmcode notwendig ist. Deswegen können Coverage-Daten einen ersten Eindruck geben, welche Dateien und welche Tests alles von den aktuellen Änderungen betroffen sein könnten und somit auch Implementations-Entscheidungen vereinfachen.

Jedoch müssen für das Aktualisieren der Abdeckung in Abhängigkeit von der gewählten Granularität sehr viele Details der Versionierung und des Auslesens des aktuellen Code-Stands mit einbezogen werden. Eine Schwierigkeit ist diesbezüglich, dass je nach eingesetzter Granularität der durch die Coverage aufgezeichnete Code sich verändern kann und damit gewährleistet sein muss, dass die Daten sich auf möglichst aktuelle Informationen beziehen. Das bedeutet erstmal, dass die aktuellen Daten auf einen festgelegten letzten vorausgegangenen Commit beziehen müssen. Wenn beispielsweise eine Statement-Coverage verwendet wird, können vergangene Zeilen sich durch Modifikationen verschieben. Solange noch an einem Commit gearbeitet wird, muss darauf geachtet werden, dass Informationen erst nach Abschluss des Commits in die Coverage-Daten aufgenommen werden. Wenn Informationen vor Abschluss eines Commits verwendet werden, muss sichergestellt sein, dass diese Informationen nicht zweimalig verarbeitet werden, was im schlimmsten Fall bedeuten würde, dass die schon aktualisierte Differenz der Daten mit abgespeichert werden muss.

Es wäre auch eine Coverage-Aufzeichnung des Testservers denkbar, wobei eine Zuordnung der Tests durch Browser-Navigation und -Steuerung des Entwicklers nicht direkt gegeben ist und erst definiert werden müsste. Eine einfache Option ist, Funktionsaufrufe der Tests und des Test-Browsers zu vergleichen und Coverage-Daten dann nach relevanten Dateien gefiltert auszuführen. Die Aktualisierung durch die Testläufe während der Entwicklung liefert allerdings schon ausreichend Daten und der Testserver müsste nur als weitere Option zur Optimierung in Erwägung gezogen werden.

3.3.2 Fehlerbasierte Test-Case-Priorisierung setzt als Grundlage eine aussagekräftige Menge an Fehlern voraus. Dazu wäre es möglich, das Projekt auf vergangene Fehler im Programmcode auszuwerten oder durch statische Analyse Fehler systematisch einzubauen. Daraufhin könnte jedem Test ein Fehlerentdeckungpotenzial für einen Fehler-Vektor, der Informationen wie textuellen Programmcode und Zeilennummer über den Fehler enthält, zugeordnet werden.

Eine praktische Herausforderung wäre, dass eine große Anzahl an Projekten zur Pflege von Fehlerdaten initial ausgewertet werden müsste. Zusätzlich müsste dieser Datensatz kontinuierlich für neu auftretende Fehlerdaten erweitert werden. Sofern durch dieses Verfahren Fehler in Test-Cases mit hoher Treffsicherheit effizient in der Testsuite priorisiert werden könnten, wäre dies ein sehr attraktives Merkmal zur Priorisierung der Testfälle. Speziell gilt das für Fehler, die auf ähnliche Muster zurückzuführen sind, wie beispielsweise auf Syntaktik oder

wiederkehrende Fehlverwendung einer Funktion aus einer für das Projekt eingesetzten Bibliothek. Hierfür würden sich auch syntaktische Mutanten besonders eignen, da diese eine systematische Generierung dieser Fehlerklassen ermöglichen.

Wenngleich dieser Ansatz in der Literatur oft als erfolgreiches Verfahren untersucht wurde, so ist die notwendige Menge an echten Fehlern, welche im Code entstehen können, sehr schwer durch vergangene Daten abzudecken [43], was sich besonders schwer im Rahmen der Problemdomäne herausstellen könnte. Die grundlegende Schwierigkeit historischer Daten im Zusammenhang der Problemdomäne wird dabei näher im Abschnitt 3.3.4 zum geschichtsbasierten Ansatz erläutert werden. Die Menge an möglichen Fehler durch Abhängigkeiten der eingesetzten Fremdsoftware, Programmiersprachen und deren Version ist durch syntaktische Modifikationen schwer einzugrenzen, da eine Vielzahl von Kombinationsmöglichkeiten mit Projekt-Anforderungen entsteht, die nur in einem bestimmten Kontext zueinander Fehler verursachen können.

Darüber hinaus impliziert ein Rückgriff auf historische Daten wiederum, dass sich das Verfahren immer nur auf frühere Versionen der Software beziehen kann und neuartige Fehler auch immer erst nach kompletter Ausführung der Testsuite vorliegen würden, was zusätzliche Testsuite-Ausführungen zur Erkennung dieser Fehler erfordern würde. Zudem ist die Frage an mindestens notwendigen Fehlerdaten und deren Generalisierbarkeit in den Veröffentlichungen zu diesem Themengebiet nicht hinreichend diskutiert und Fehler meist allein durch syntaktische Modifikationen erreicht worden [43, 56].

Im Gesamten ist dieses Verfahren mit einigen Hürden sowohl als primäres als auch als sekundäres Verfahren zur Priorisierung relevant. Allerdings wäre die Verwendung dieses Ansatzes auch davon abhängig zu machen, wie sicher und häufig fehlerhafte Test-Cases damit identifiziert werden könnten und wie sich die Kosten der Fehler-Datenpflege und -auswertung dazu verhalten.

3.3.3 Kostenbasierte Test-Case-Priorisierung Da sich auch die Kosten der einzelnen Tests durch Modifikationen verändern, sind die neuen Kosten erst nach einer kompletten Ausführung der Testsuite bekannt. Die einfachste Methode, diese Kosten festzustellen, wäre, die durchschnittlich benötigte Laufzeit für einen Test zu berechnen und abzuspeichern. Daraus folgend ist zu schließen, dass dieser Ansatz immer eine Neuausführung der Testsuite benötigt, da erst damit für die aktuellen Änderungen relevante Priorisierung berechnet werden kann. Somit würden auch kostenbasierte Verfahren im Rahmen dieser Arbeit ein zweitrangiges Merkmal darstellen.

Allerdings können den aktuellen Ausführungszeiten relevante Informationen zur Priorisierung entnommen werden. Größere Änderungen der Laufzeit eines Tests indizieren nämlich, dass an den Ausführungspfaden des Tests bedeutende Veränderungen vorliegen. Sofern einzelne Tests eine überdurchschnittlich

lange Laufzeit vorweisen, könnten deren getestetes Verhalten zur effizienteren Priorisierbarkeit der Testsuite in kleinere Tests aufgeteilt werden.

Dabei weisen vor allem Integrationstests durch die Browser-Ansteuerung und lange Interaktionsketten von Befehlen und Abfragen am häufigsten eine hohe Ausführungszeit auf. Da für die Priorisierung oft nur Abschnitte der Interaktionsketten relevant sind, könnte dies im Gesamten dazu führen, dass schneller eine Rückmeldung vorliegt. Der Nachteil jedoch ist, dass viele der enthaltenen Interaktionen einen gemeinsamen Kontext an benötigten Datenbankeinträgen und Autorisierung teilen, welcher dann mehrmals erstellt werden müsste, wodurch wiederum die Gesamtlaufzeit der Testsuite erhöht wird. Aktuelle und möglichst kurze Ausführungszeiten der Tests sind auch dann relevant, wenn die priorisierte Testsuite zur parallelen Ausführung in Gruppen mit möglichst gleich langer Laufzeit aufgespalten werden soll.

3.3.4 Geschichtsbasierte Test-Case-Priorisierung Dieser Ansatz ist sehr attraktiv, da durch CI-Ausführungen der Testsuites eine Vielzahl von Daten entsteht, welche direkt genutzt werden können. Relevante Daten zur Priorisierung können beispielsweise Kosten, Fehler, für Fehler besonders anfällige Programmabschnitte und Tests mit häufig flackernden Ergebnissen sein, womit dieses Verfahren zumindest als sekundäres Merkmal zur Priorisierung relevant ist. Interessant wäre auch, Fehler dahingehend auszuwerten, ob diese in einem bestimmten Muster immer wieder auftreten.

Dabei ist ein Nachteil für diesen Ansatz im Rahmen dieser Bewertung, dass bei makandra in vielen Projekten erst seit kurzem CI integriert wurde, womit diese Daten in großer Anzahl noch nicht vorliegen. Eine weitere Option wäre das in 5.2 vorgestellte Verfahren, um vergangene Daten wiederherzustellen und dadurch auszuwerten. Dies gibt zwar nicht den gleichen Einblick wie Daten einer CI-Historie, da hierbei aussagekräftigere Informationen bezüglich eines tatsächlich erstellten Commits und der dadurch getriggerten Testsuite ausgewertet werden können, kann jedoch durchaus Fehler durch zusammenhängende authentische Veränderungen wiederherstellen. Generell ist eine praktische Herausforderung bei diesem Verfahren, dass eine große Menge an Daten zentral abrufbar verwaltet werden müssten, so dass eine effiziente Auswertung für den aktuellen Commit möglich ist.

Eine weitere Einschränkung dieses Verfahrens ist, dass Daten, die sich auf tatsächlichen Quelltext beziehen, bezüglich der untersuchten Problemdomäne stark kontextgebunden sein können. Dies liegt vor allem daran, dass in der webbasierten Software immer wieder neue Trends auftreten und zum anderen das Web sich in den letzten Jahren stetig weiterentwickelt hat [24, 36]. Zusätzlich ist auch davon auszugehen, dass neben Weiterentwicklung des bereits bestehenden Technologiebereichs auch weitere Bereiche zunehmend im Web voranschreiten und inkorporiert werden [44, 53, 58]. Bezüglich des für die Anwendungslogik relevanten

Codes ist vor allem von Bedeutung, dass eingesetzte Frameworks, Bibliotheken und Sprachen wie JavaScript, HTML und CSS sich in der Vergangenheit stetig weiterentwickelt haben. Dies hat nicht zuletzt dazu geführt, dass für die im Web grundlegenden Sprachen einheitliche Standards definiert und dann in zeitlichen Abständen von den Browsern umgesetzt werden [22, 28]. Da bei Sprachen den CSS und JavaScript nach wie vor eine Weiterentwicklung stattfindet, existieren zur Frage, ob bereits eine Unterstützung neuer Features der Browser vorliegt, Open-Source Lösungen wie die Webseite *caniuse.com* [4].

Bezüglich der verwendeten Funktionen eines Frameworks, einer Bibliothek oder Sprache sind hier vor allem so genannte *Breaking-Changes* des verwendeten Application-Programming-Interfaces (API) erwähnenswert, da damit häufig eine signifikant unterschiedliche Verwendung einzelner Funktionen notwendig ist, um dieselbe Funktionalität zu gewährleisten [27]. Für den Rahmen dieser Arbeit soll auch hervorgehoben werden, dass Ruby on Rails ein Web-Framework ist, das in besonderem Maße für stetige Veränderung und Weiterentwicklung bekannt ist [48]. Damit ist zu schließen, dass für eine Anwendung abhängig von den Anforderungen sehr unterschiedliche Frameworks und Bibliotheken eingesetzt sein können und je nach dem Zeitpunkt, zu dem diese geschrieben wurde, der Code durch die damals bereitgestellte API sehr unterschiedlich ausfallen kann und neuere Features der API dort noch gar nicht enthalten sein können.

Im Umkehrschluss ist zu folgern, dass damit entstehende Fehlermöglichkeiten signifikant von Projekt zu Projekt variieren können. Das gilt im Besonderen für Webanwendungen, welche häufig sehr viele als Open-Source verfasste Fremdsoftware verwenden, was auch bei makandra der Fall ist. Deswegen ist davon auszugehen, dass vor allem für neuere Features eingesetzter Bibliotheken oder erst seit kurzem genutzte Bibliotheken noch keine Daten vorliegen oder zu selten vorkommende Anwendungsfällen wenig bis kaum Daten vorhanden sein können. Damit gilt, dass eine Generalisierbarkeit für viele Anwendungsfälle nur schwer erreichbar sein könnte oder der Datensatz auf eine weitaus größere Anzahl an Projekten ausgeweitet werden müsste, als bei makandra selbst vorliegt.

3.3.5 Anforderungsbasierte Test-Case-Priorisierung Ein Nachteil vieler Veröffentlichungen in diesem Bereich ist, dass Sie die Anforderungen der Softwareentwicklung, welche stetige Veränderung des Programmcodes verlangen, nicht mit einbeziehen. So kommt auch die anforderungsbasierte Software als alleinige Herangehensweise nicht in Frage, da Priorisierung nach globalen Anforderungen für lokale Änderungen nicht zielführend ist und es zusätzlichen Aufwand bedeutet, Anforderungen für lokale Änderungen zu definieren und auszuwerten.

Des Weiteren müsste aus dem modifizierten Code zuerst eine betroffene Anforderung abgeleitet werden, was den Rückgriff auf Verfahren des Natural-Language-Processing oder String-Distance erfordern würde. Eine zusätzliche Priorisierung nach Anforderungen wäre denkbar, wenn auch der Nutzen abhängig

von der Größe der Testsuite und wie spezifisch die bereits zu Grunde legende Priorisierung ist. Allerdings gibt es hier wiederum insofern eine Problematik, dass historische Daten von Fehlern und Anforderungen notwendig sein müssten und das für eine Vielzahl von alten Projekten erst nachgezogen werden müsste.

3.3.6 Suchbasierte und kombinierte Ansätze Kombinierte Ansätze, welche mehrere Parameter mit einbeziehen können, sind besonders attraktiv, wenn Wissen verschiedener Daten vereint werden soll oder die beste Priorisierung möglicherweise von Zusammenhängen der einzelnen Ansätze abhängt. Das Hauptproblem, das mit diesen Ansätzen einhergeht, ist, dass die Aussagekraft der einzelnen Ansätze und Verfahren erst für die Kriterien und die eingesetzten Technologien evaluiert werden müssen, damit diese sinnvoll miteinander kombiniert werden können. Dasselbe gilt auch für genetische Algorithmen und maschinelles Lernen.

Da suchbasierte Verfahren in der Lage sind, mehrere Kriterien miteinzubeziehen, um beispielsweise gierig das Erstbeste zu wählen, könnten diese für die Kombination der Informationen eine geeignete Lösung sein oder zumindest einen ersten praktischen Anhaltspunkt liefern, welche Kombinationen die Performanz eines Ansatzes verbessern.

Maschinelles Lernen ist besonders dann attraktiv, wenn die Problemdomäne so komplex ist, dass Zusammenhänge zwischen einer Vielzahl von unterschiedlichen Daten nicht bekannt sind und kein anderes effizientes Verfahren gefunden werden kann. Aus diesem Grund ist für die Kombination vieler Kriterien dieser Ansatz attraktiv. Folglich wurde auch schon die Effizienz einer Vielzahl an Lernverfahren in diesem Gebiet untersucht. Gleichzeitig hat es jedoch auch inhärente Schwierigkeiten, da viele der vorgestellten Methoden nicht im Hinblick auf dynamische Veränderung der Software untersucht wurden. Dies liegt daran, dass diese Methoden oft voraussetzen, dass der gesamte Lerndatensatz initial zur Verfügung steht und nur einmal gelernt werden muss [45].

Vor allem textuelle Daten des Programmcodes können sich, wie in 3.3.4 ausgeführt wurde, abhängig von den eingesetzten Technologien und deren Version stark verändern. Manche der für das Lernen notwendigen Daten mögen zu Beginn eines neuen Projekts auch noch nicht im ausreichenden Umfang vorliegen, womit hier auf eine vorübergehende Kaltstart-Methode ausgewichen werden müsste.

3.3.7 Modellbasierte Test-Case-Priorisierung Ein modellbasiertes Verfahren wäre in der Lage, sich auf Modifikationen zu beziehen, sofern die Modifikationen auf vorhandene Graphenstrukturen angewandt werden und die Graphen für aktuelle Änderungen angepasst werden können. Da die meisten Graphenmodelle die Ergebnisse statischer Analyse als Grundlage dienen, könnten geänderte Zeilen in der Struktur ausfindig gemacht und über einen Pfad auf die betroffenen Testfälle zurückgeführt werden. Durch die Neuausführung der statischen Analyse beim

Erkennen von Änderungen oder durch Abfrage von Laufzeitinformationen des Programms könnten die ausfindig gemachten Pfade angepasst oder gegebenenfalls neue Pfade und Knoten hinzugefügt werden. Damit wäre es auch möglich, das Modell im Hintergrund für neue Änderungen anzupassen. Es könnten auch die ausgeführten Programmpfade von Tests und des Testservers identifiziert und in den Graphen eingearbeitet werden. Deswegen stellt ein modellbasiertes Verfahren eine mögliche Option zur Priorisierung dar.

Ein großer Nachteil jedoch ist, dass Ruby, die Sprache, auf der das von makandra eingesetzte Web-Framework Ruby on Rails basiert, eine dynamisch typisierte, zur Laufzeit interpretierte Programmiersprache ist. Dies hat zur Folge, dass bei Referenzen innerhalb der Code-Analyse nicht klar sein kann, auf welche Klasse sich diese beziehen und das oft erst zur Laufzeit bekannt ist. Das ist neben dynamischen Typen darauf zurückzuführen, dass in Ruby Metaprogramming häufig eingesetzt wird, d.h. dass Programmteile zur Programmlaufzeit umgeschrieben oder erweitert werden können, womit solche Teile des Codes nicht einfach durch statische Analyse identifizierbar sind. Dies sind auch die Gründe, warum es bisher wenige Tools in Ruby gibt, die versuchen, einen gesamten Graphen der Software zu erstellen.

Besonders häufig wird dafür das Parser [13] Gem verwendet. Dieses ist in der Lage, abstrakte Syntaxbäume, welche in diesem Zusammenhang eine hierarchische logische Zergliederung der verwendeten Operatoren darstellen, für einzelne Dateien zu erstellen. Viele Lösungen, die damit zum Beispiel Kontrollflussgraphen erstellen, sind jedoch meist auf die vorher genannten Limitierungen begrenzt und beziehen sich damit meist nur auf einzelne Dateien. Dass eine tiefer liegende Auswertung grundsätzlich möglich ist, zeigen jedoch auch Gems wie Brakeman [3] oder Reek [15]. Während Reek dasselbe für Code-Design bewerkstelligt, analysiert Brakeman statisch Ausführungspfade einer Rails-Anwendung im Hinblick auf Sicherheitslücken.

Als erster Anhaltspunkt zur Priorisierung könnte das auf Parser basierte Gem Rubrowser [17] dienen, welches eine statische Analyse aller Code-Dateien in einem Ordner durchführen kann, um damit Lade-Abhängigkeiten zwischen Dateien zu erkennen. Damit kann dann ein Abhängigkeitsgraph erstellt werden, in dem für jede Programmdatei eine gerichtete Kante darstellt, ob eine andere Datei zur Funktion des enthaltenen Codes geladen werden muss. Indem nachgesehen wird, welche Abhängigkeiten von geänderten Dateien benötigt werden, könnten Tests für diese Dateien mit höherer Priorität ausgeführt werden. Jedoch müsste das erst so erweitert werden, dass Rubrowser Abhängigkeiten aus den definierten RSpec-Tests erkennen kann, da die Abhängigkeiten dynamisch von RSpec nach den Test-Definitionen und jeweiligen Factory-Aufrufen geladen werden und damit in ersten Testversuchen des Gems nicht einsehbar waren. Eine Option zur Erweiterung eines Graphen mit Laufzeitinformationen wäre das in Ruby existierende Modul TracePoint [6], welches während der Laufzeit Events wie Methodenaufrufe und Klassendefinitionen aufzeichnen kann.

3.4 Auswahl

Da im Rahmen dieser Bachelorarbeit ein Einsatz prototypisch implementiert und auf seine Eignung ausgewertet werden soll, mussten alle bis auf einen der vorgestellten Ansätze ausgeschlossen werden. Nachdem bisher noch keines der genannten Verfahren in Ruby entwickelt wurde, wurden kombinierte Ansätze, für deren einzelne Aussagekraft noch keine Evidenz innerhalb der Problemdomäne vorliegt, ausgeschlossen. Dies hat den Grund, dass die Effizienz und Aussagekraft der einzelnen Ansätze erstmals für die definierten Kriterien evaluiert und getestet werden müssen. Außerdem wurden Ansätze, deren Beitrag zur Priorisierung sich als zweitrangig für die im Rahmen der Bewertung in Anbetracht der definierten Kriterien herausgestellt hat oder die sich überwiegend auf historische Daten beziehen, für die Implementierung nicht in Betracht gezogen.

Da der coverage-basierte Ansatz eines der am häufigsten untersuchten Verfahren der Priorisierung darstellt, nach früheren Experimenten zu guten Priorisierungs-Ergebnissen führt [56] und die in 3.2 definierten Kriterien erfüllt, ist das Ziel für den weiteren Verlauf dieser Arbeit diesen zu implementieren und auszuwerten. Zumal dieser Ansatz im Gegensatz zum Modellbasierten weniger anfängliche Hürden mit sich bringt.

Darüber hinaus sollen auch erste Varianten davon und die Einsatztauglichkeit dieses Ansatzes für den Entwickler untersucht werden. Dies ist auch für weitere Untersuchungen in diesen Bereichen relevant, da in vielen Auswertungen dieser Ansatz als Grundlage zum Vergleich verschiedener Priorisierungsverfahren hergenommen wurde. Dafür werden jeweils unterschiedliche Metriken berechnet und verglichen, welche ein Maß für die Güte einer Priorisierung darstellen sollen [29, 45]. Für zukünftige Arbeiten in diesem Bereich werden auch der modellbasierte Ansatz sowie verschiedene Kombinationen der bisher genannten Möglichkeiten in Zusammenhang mit zweitrangigen Merkmalen zur Priorisierung in Erwägung gezogen.

4 Prototypische Implementierung

Das von Ruby nativ bereitgestellte Modul Coverage [10] bietet eine solide Grundlage, auf welcher coverage-basierte Methoden implementiert werden können. Dabei verfügen die zur Verfügung gestellten Methoden bereits über eine Auswertung von Funktions-, Branch- und Statement-Coverage. Die genauen Zusammenhänge zur Konzeption, Funktionsweise und Umsetzung der Implementierung soll in den folgenden Abschnitten dargestellt und erläutert werden.

Als Vorlage für den initialen Entwurf diente ein Priorisierungsverfahren nach additionaler Coverage, wie es von Beena und Sarala [26] vorgestellt wurde. Es

wurde bei der Herangehensweise darauf geachtet, zuerst den einfachst möglichen Ansatz der Coverage-Auswertung als Grundlage zum Vergleich umzusetzen. Deswegen wurde auch die Vereinfachung getroffen, dass sich das implementierte Verfahren erstmals nur auf das Test-Framework RSpec bezieht.

Um damit auch einen ersten Einblick in die spätere Verwendbarkeit bekommen zu können, wurde der Prototyp als Kommandozeilen-Programm konzipiert und umgesetzt. Dieses führt bei Eingabe des Befehls *testsort prioritized* in die Kommandozeile die priorisierte Testsuite aus. Die Wahl eines Kommandozeilen-Programms hat einerseits den Vorteil, dass kein User-Interface benötigt wird und ist damit kongruent, dass ein Großteil der Entwickler bei makandra die Kommandozeile zur Ausführung der Testsuite bereits verwenden.

Intern werden bei Aufruf des Kommandos zur priorisierten Testsuite-Ausführung, die geänderten Dateien ausgelesen und darauf basierend die Priorisierung anhand der bereits aufgezeichneten Coverage-Daten berechnet. Als nächstes werden die Tests nach der Priorisierung geordnet laufen gelassen. Infolgedessen wurden Coverage-Daten bei jeder Ausführung einer Testsuite aufgezeichnet und zusammengeführt. Zur späteren Wiederverwendbarkeit bei erneuten Aufrufen des Kommandos wurden diese Daten als Dateien abgespeichert.

4.1 Verwendete Granularität

Zuerst wurde versucht eine Statement-Coverage zu verwenden, da in früheren Untersuchungen eine feinere Granularität mit höheren APFD-Werten (5.1.1) korreliert haben [49]. Nach ersten Entwürfen des Programms hat sich herausgestellt, dass eine Granularität, welche sich auf konkreten Inhalt des Codes bezieht, wie Zeilennummern oder Funktionsnamen, in der Praxis sehr oft aktualisiert werden muss. Das ist beispielsweise dann der Fall, wenn der Funktionsname sich ändert oder Zeilen gelöscht oder verschoben werden.

Das bedeutet, dass nicht nur der numerische Wert der Abdeckung, sondern auch die granulare Repräsentation des Codes immer so angepasst werden muss, dass diese weiterhin mit der Struktur des tatsächlichen Codes übereinstimmt. Dabei spielt hauptsächlich die verwendete Versionierung der Software eine Rolle und wie diese in der Praxis eingesetzt wird. Besonders kritisch ist hierbei, Inkonsistenzen der Coverage-Daten, die entstehen könnten, wenn Änderungen mehrfach selektiv mit einbezogen werden, zu vermeiden.

Die folgenden Abschnitte 4.1.1 und 4.1.2 gehen auf praktische Schwierigkeiten ein, die sich bei dem Versuch zur Verwendung einer feineren Granularität herausgestellt haben. Zugleich stellen diese die Grundlage dar, weshalb die in 4.1.3 vorgestellte Granularität schließlich konzipiert und zur prototypischen Implementierung gewählt wurde.

4.1.1 Aufzeichnung nach Abschluss eines Commits bei feiner Granularität

Die einfachste Lösungsidee wäre, Modifikationen erst nach Abschluss eines Commits als Aktualisierung der Coverage-Daten zu verwenden. Jedoch hat dies wiederum den Nachteil, dass Commits je nach Aufgabe und Entwickler zu unterschiedlichen Zeitpunkten erstellt werden. Dadurch würden die aktuellen Änderungen in der Priorisierung erst verspätet in der Datenstruktur zur Priorisierung aufgenommen werden. Zur Aufzeichnung wäre es notwendig, die Regressionstests automatisch nach Abschluss eines Commits zu triggern und nur in der getriggerten Ausführung die Coverage aufzuzeichnen.

Im Gesamten ist ein großer Nachteil an diesem Lösungsansatz, dass Daten immer erst einer getriggerten Testsuite ohne selektives Zusammenführen möglich wäre, was zugleich auch ihren Nutzen mindert, da viele durch den Entwickler verursachten Testsuite-Durchläufe nicht genutzt werden könnten.

4.1.2 Aufzeichnung vor Abschluss eines Commits bei feiner Granularität

Falls Modifikationen mit einbezogen werden, bevor der aktuelle Commit abgeschlossen ist, müsste erstmal der zuletzt abgeschlossene Commit hinterlegt werden, damit eindeutig bestimmt werden kann, auf welchen Stand des Codes sich die aktuellen Änderungen beziehen. Dieser muss dann nach Bedarf bei neuen Commits so angepasst werden, dass wiederum der letzte Commit hinterlegt ist, was aber nur dann geschehen darf, wenn die Coverage aller neuen Commits bereits aufgezeichnet wurde.

Wenn nach aktuellen Modifikationen eine Differenz zum zuletzt aufgezeichneten Commit besteht, stellt sich damit die Frage, ob für diese Modifikationen schon eine Aktualisierung durchgeführt wurde. Es müsste auch ein eindeutiges Verfahren dafür entwickelt werden, welches je nach Modifikationen einer Datei die den Zeilennummern zugehörige Abdeckung immer korrekt entfernt, verschiebt und gegebenenfalls neue Einträge hinzufügt.

Denkbar wäre, dass dafür immer die schon hinzugefügte Differenz für die aktuellen Modifikationen mit abgespeichert wird und dann nur in dieser Differenz nachgesehen werden muss, ob eine Modifikation schon in den Coverage-Daten aktualisiert wurde. Gesetzt dem Fall, dass einer der letzten Commits verändert wird, müsste hier immer eine Neuausführung der Testsuite zur Aufzeichnung der Coverage-Daten getriggert und zusammengeführt werden.

4.1.3 Verwendung einer Dateipfad-Granularität

Aus den in 4.1.1 und 4.1.2 vorgestellten Gründen wurde vorerst, damit weniger Wissen über die Veränderung an exakten Zeilennummern beachtet werden muss und das Verfahren trotzdem mit jeder Ausführung der Testsuite aktualisiert werden kann, als erste Vereinfachung angenommen, dass es genügt, eine Dateipfad-Granularität, als die Coverage von Testdateien auf Programmdateien, zu verwenden. Der Vorteil dieser Vereinfachung

ist, dass immer bei Ausführung einer ganzen Test-Datei die Ergebnisse verwendet werden können, ohne dass Inkonsistenzen erzeugt werden. Wenn fehlerhafte Testzustände der Tests einer Testdatei vorliegen, könnten die Ergebnisse für diese verworfen oder zwischengespeichert werden, bis ein fehlerfreier Durchlauf vorliegt. Dies stellt zugleich auch eine Einschränkung des Verfahrens dar, die im Abschnitt 5.5 über die Eignung und Limitierungen der Implementation nochmal näher betrachtet werden soll.

Diese Vereinfachung ist bei strikter Umsetzung der SOLID-Prinzipien gerechtfertigt, da hierbei jeweils eine Datei so geschrieben werden kann, dass diese eine Grundfunktionalität der Software abkapseln und in einer weiteren Datei dafür die jeweiligen Tests enthalten sind. Insbesondere ist dies auch für den Aufbau einer Rails-Anwendung und einer objektorientierten Programmiersprache wie Ruby geeignet. Einerseits, da die von Rails on Rails eingesetzte Model-View-Controller-Architektur einer Datei und dem zugehörigen Test einer der Komponenten View, Controller oder Model zugeordnet werden kann und somit auch in eine übersichtliche Ordnerstruktur gegliedert werden kann. Andererseits bietet Ruby viele Funktionalitäten durch Vererbung, Modularisierung und Metaprogrammierung weitere Funktionalitäten einer Klasse zu kapseln und wird in Rails in Arten wie Controller-Helpern und Service-Objekten umgesetzt.

Zusätzlich wird diese Praxis auch von makandra konsequent umgesetzt, so dass dafür sogar weitere Gems entwickelt wurden, und zwar Modularity [9] und ActiveType [1]. ActiveType fügt Ruby on Rails die Möglichkeit hinzu, große Models nach ihrer View-Funktionalität in kleinere Models durch Vererbung aufzuteilen. Modularity gibt einem Modul die Möglichkeit, mit verschiedenen Parametern aufgerufen zu werden. Sofern eine ganze Test-Datei ausgeführt wird, wie es meist von Entwicklern getätigt wird, kann dann immer die Coverage-Information der gesamten Datei anstelle einzelner Veränderungen aktualisiert werden. Damit kann also im Rahmen dieser Arbeit trotzdem bereits untersucht werden, ob genügend Indizien zur Umsetzbarkeit und Effizienz dieses Ansatzes zur Priorisierung vorliegen, um als Lösungsansatz für makandra in Erwägung gezogen zu werden.

4.2 Verwalten der Coverage-Daten

4.2.1 Aufzeichung Zur Auswertung der Coverage, war es zuerst notwendig, für jeden einzelnen Test die erhaltene Coverage aufzuzeichnen. Dafür musste vor jedem Aufruf einer Testdatei die Messung der Coverage begonnen und nach dem jeweiligen Test in eine dafür geeignete Datenstruktur hinterlegt werden. Mit der ersten Vereinfachung einer dateibasierten Zuordnung wurde dann die Coverage für die gesamte Testdatei berechnet, indem die Coverage der einzelnen Zeilen zusammen addiert wurde. Hierfür dient eine tabulare Datenstruktur als Vorlage, die in Tabelle 1 dargestellt ist, wobei T_i mit $1 \leq i \leq n$ und F_j mit

$1 \leq j \leq m$ jeweils für die ausgeführte Testdatei, beziehungsweise einer Datei des Programmcodes stehen.

Tabelle 1: Coverage-Datenstruktur der jeweiligen Test- und Programmcode-Dateien

	F1	**F2**	**F3**	**F4**	**F5**	**F6**
T1	20	0	0	5	6	0
T2	16	0	15	10	5	5
T3	5	0	0	29	0	21
T4	2	0	0	0	1	2
T5	1	4	5	0	4	0

Die tabulare Darstellung ist mithilfe einer zweidimensionalen Matrix aus zwei in sich geschachtelten Arrays umgesetzt worden. Um dafür eine vielseitig einsetzbare API bezüglich mehrdimensionaler Array-Operationen verwenden zu können, wurde das Gem Numo-Narray [11] verwendet, welches ein Ruby-Analogon der bekannten Bibliothek Numpy [12] für Python [14] darstellt. Dies hat den Vorteil, dass über die gegebenen Code- und/oder Test-Indizes alle betroffenen Testdateien performant abgefragt und weitere Funktionen wie Zählen, Sortieren oder Summieren effizient auf den abgefragten Teilbereich aufgerufen werden können.

4.2.2 Zuordnung Damit die Indizes eindeutig dem Dateipfad genauso wie umgekehrt zugeordnet werden können, wurde eine Klasse *FileToIndexMapping*, welche im Folgenden als Index-Datei-Zuordnung bezeichnet wird, erstellt. Dazu verwaltet diese Klasse zwei Hashtabellen.

Die erste Hash-Tabelle ordnet einen Index auf den Dateipfad und die zweite Hash-Tabelle ordnet einen Dateipfad einem Index zu. So kann für einen gegebenen Pfad der Index, respektive der Pfad für einen gegebenen Index, über Methodenaufrufe der Klasse effizient abgefragt werden.

4.2.3 Speicherung Nach Ende eines Testsuite-Durchlaufs wurde die Datenstruktur dann in mehreren Dateien abgelegt, um für spätere Aufrufe zu Beginn wieder ausgelesen werden zu können. Dafür wurden sowohl die Coverage-Matrix als auch die beiden Index-Datei-Zuordnungs-Objekte für die Testdatei- und Programmdatei-Zuordnung als Text- und JSON-Dateien gespeichert.

4.3 Auslesen der aktuellen Veränderungen

Da zur Revisionskontrolle bei makandra Git eingesetzt wird, mussten auch mithilfe von Git die aktuellen Änderungen ausgelesen werden. Deswegen wurde die Bibliothek Rugged [20] verwendet, die es erlaubt, die Kommandos von Git innerhalb des Programmcodes auszuführen, indem dafür abstrahierte Methoden für die Interaktion mit dem Git Repository zur Verfügung gestellt werden.

4.3.1 Wahl eines Git-Kommandos Grundsätzlich stehen in Git zwei Kommandos zur Verfügung, aktuelle Modifikationen auszulesen, und zwar *status* und *diff*. Der Befehl *status* gibt dabei an, ob Dateien im Arbeitsverzeichnis modifiziert wurden und ob diese schon zur Indizierung des aktuellen Commits hinzugefügt wurden. Im Gegensatz dazu kann *diff* verwendet werden, um genaue Änderungen auszulesen und die Differenz zu einer anderen Version des Codes anzuzeigen.

Für das weitere Verfahren wurde aufgrund der vorher beschriebenen Vereinfachung einer Datei-Granularität *status* verwendet, da hierfür direkt die notwendigen Dateiinformationen ausgegeben werden. Nachdem *diff* die inhaltlichen Modifikationen ausgeben kann, müsste, sofern eine andere Granularität eingesetzt werden würde, der Befehl *diff* eingesetzt werden.

4.4 Priorisierung

Zur Berechnung Priorisierung wurden bereits erste unterschiedliche Algorithmen als Varianten dieses Ansatzes implementiert, getestet und ausgewertet. In diesem Abschnitt werden die Verfahren und deren Unterschiede vorgestellt. Für alle Verfahren bildeten die aktuellen Veränderungen eine erste Selektion der zu priorisierenden Test-Fälle.

Durch die Pfade der von Modifikationen betroffenen Code-Dateien konnte eine Zuordnung zu den zugehörigen Indizes der Coverage-Matrix über die Index-Datei-Zuordnungs-Klasse der Coverage-Datenstruktur erfolgen. Mithilfe der Programmdatei-Indizes mussten dann nur noch alle Testdateien ausgewählt werden, welche für die betroffenen Änderungen überhaupt eine Coverage vorweisen. Damit konnte im nächsten Schritt für die Indizes der Testdateien eine Priorisierung berechnet werden, nach der diese dann sortiert werden konnten. Alle Testfälle, die in dieser Vorauswahl nicht enthalten gewesen sind, wurden an die Priorisierung in der Reihenfolge angehängt, wie Sie in den Coverage-Daten hinterlegt wurden.

4.4.1 Absolute Priorisierung Als erstes Verfahren wurde die Priorisierung nach absoluter Coverage implementiert. Dies bedeutet, dass die Tests, welche für

die aktuelle Änderung über die größte Abdeckung verfügen, höher priorisiert und damit zuerst ausgeführt werden. Dafür mussten lediglich die vorselektierten Testdateien absteigend nach ihren absoluten Coverage-Werten sortiert werden.

4.4.2 Additionale Priorisierung Bei diesen Verfahren wird versucht, möglichst schnell eine Gesamt-Coverage über alle Projektdateien zu erreichen, indem noch nicht getroffene Programmzeilen höher priorisiert werden. Um dies zu erreichen, wurden alle Indizes der Testdateien absteigend danach sortiert, wie viele der betroffenen Dateien Sie bei ihrer Ausführung abdecken. Da dieselbe Anzahl an Dateien mehrmals getroffen wurde, wurden im zweiten Schritt diese Dateien nach ihrer absoluten Coverage sortiert.

Daraufhin wurde der erste Index in dieser Sortierung gewählt und abgerufen, wie viele Code-Indizes von diesem getroffen wurden. Alle bereits getroffenen Indizes wurden vorübergehend gespeichert. Danach wurden nur darauf folgende Test-Indizes der Sortierung gewählt, die Programmcode-Abdeckung von Dateien enthielten, die bisher noch nicht von den gewählten Tests aufgerufen wurden, bis keine mehr gefunden wurden. Das wurde iterativ solange für die noch nicht gewählten Indizes wiederholt, bis alle Indizes aus der Vorsortierung gewählt waren, wobei die Zählung der bereits aufgerufenen Dateien mit jeder Iteration zurückgesetzt wurden.

4.4.3 Pseudo-additionale Priorisierung Diese Priorisierung verwendete die zur Einführung der additionalen Priorisierung beschriebene Vorsortierung der Testdateien absteigend nach der Anzahl an getroffenen Code-Dateien und ihrer absoluten Coverage. Sie wurde im Rahmen dieser Arbeit als pseudo-additional bezeichnet, da sie zwar auch eine frühe Coverage über das gesamte Projekt erreicht, jedoch dies nicht auf Basis bereits gewählter Tests erreicht, sondern nach einer einfacheren Sortierung.

4.5 Ausführung der Testsuite

Zur Ausführung der Testsuite mussten nun die nach der Priorisierung sortierten Indizes den Testdateien zugeordnet werden, wofür wiederum die Index-Datei-Zuordnungs-Klasse diente. Im letzten Schritt wurde einfach das zum Testen verwendete Tool RSpec mit den Testdateien als Parameter aufgerufen, was dann die Tests in der gegebenen Reihenfolge aufruft.

5 Auswertung und Beurteilung der Implementierung

In diesem Abschnitt werden die prototypisch implementierten Methoden im Hinblick darauf bewertet und verglichen, ob diese im Rahmen der makandra GmbH einsetzbar sind, sowie evaluiert, was mögliche Vor- beziehungsweise Nachteile sind. Die Auswertungen wurden basierend auf historischen Fehlerdaten vergangener Commits nach dem in 5.2 vorgestellten Verfahren durchgeführt.

Die zur Auswertung verwendeten Metriken und Diagramme werden dabei vorgestellt und deren Verwendung begründet. Die Ergebnisse werden zuerst einzeln vorgestellt und miteinander verglichen. Einzelne Ausreißer, bei denen das Verfahren zu besonders schlechten Ergebnissen geführt hat, wurden im Hinblick auf Ursachen und mögliche Optimierungen betrachtet. Zuletzt wurden Beschränkungen der Implementierung und mögliche Lösungsansätze dazu herausgearbeitet.

5.1 Verwendete Metriken und Grafiken

Als erstes wurde die Ausführungszeit für die Verwendung der prototypisch implementierten Priorisierungs-Strategien und der zufälligen Priorisierung mit und ohne Priorisierungs-Tool aufgezeichnet, damit hier mindestens eine genauso gute Effizienz wie bisher gewährleistet werden kann. Zur Auswertung der Güte eines Priorisierungsverfahren wurden für jede Strategie unterschiedliche Metriken berechnet und Grafiken erstellt.

Diese waren Average-Percentage-Fault-Detection (APFD) und die vergangene Zeit bis zum ersten fehlgeschlagenen Test. Es wurden auch die Durchschnittswerte dieser beiden Metriken für eine Strategie berechnet und angegeben. Da der Median eine größere Robustheit gegen Ausreißer aufweist, wurde dieser ebenso berechnet und angegeben. Für die Grafiken wurden Streudiagramme, Histogramme und Box-Plots gewählt. Im Weiteren soll kurz darauf eingegangen werden, weshalb die einzelnen Metriken und Grafiken verwendet wurden und was deren Aussagekraft im Kontext dieser Arbeit sind.

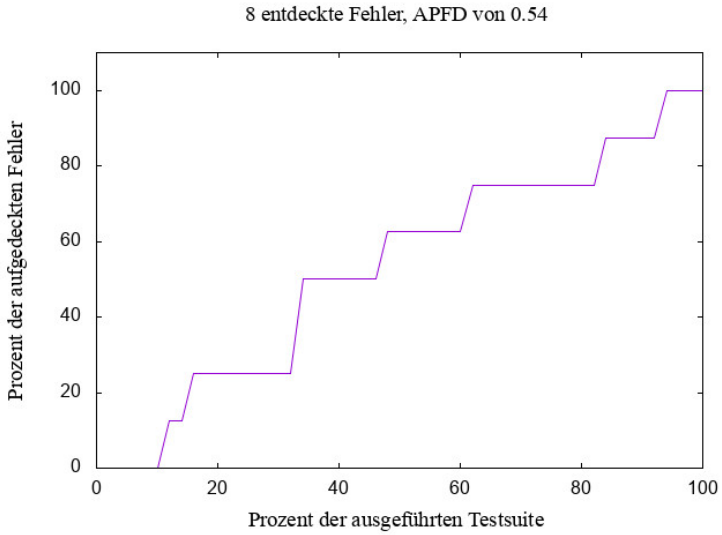
5.1.1 Average-Percentage-Fault-Detection Für eine gegebene Test-Case-Ordnung gibt APFD für Werte zwischen 0 und 1 an, wie schnell alle vorhandenen Fehler entdeckt werden. Das heißt, je schneller die Fehlerwerte entdeckt werden, desto höher sind die resultierenden Werte, kleiner für eine langsamere Fehlerabdeckung respektive. APFD ist für n Testfälle und m Fehler folgenderweise definiert:

$$AFDP = 1 - \frac{TF_1 + TF_2 + ... + TF_m}{nm} + \frac{1}{2n}$$

TF_i mit $1 \leq i \leq m$ gibt die Position innerhalb der Testsuite-Ausführung an, an der der i-te Fehler entdeckt wurde.

Visuell und formal betrachtet kann dies auch als Integral unter dem Graphen verstanden werden, welcher angibt, wie die Anzahl der entdeckten Fehler in Abhängigkeit der ausgeführten Testsuite ansteigt, was anhand Abbildung 1 für einen randomisierte Priorisierung als Beispiel dargestellt werden soll. Derartige Grafiken wurden zur exemplarischen Bewertung und Veranschaulichung der Güte einer Strategie für jeden einzelnen Commit erstellt.

Abb. 1: Beispiel eines APFD-Graphen einer zufälligen Priorisierung



Nachdem die randomisierte Priorisierung einer Gleichverteilung entspricht, liegt der APFD-Wert im Durchschnitt für diese Testsuite-Durchläufe bei $0,5$, was genau der Hälfte der Fläche unter dem Graphen entspricht. Dies liegt daran, dass jede Position für einen Test gleich wahrscheinlich ist, was auch an den Abbildungen 2 zur Auswertung dieser Strategien zu erkennen ist. Damit ist für Testsuite-Durchläufe mit geringer Fehleranzahl der Wert gleichverteilt in $(0,0; 1,0)$, während er für Commits mit vielen Fehlerfällen oft nahe $0,5$ aufzufinden ist.

5.1.2 Vergangene Zeit bis zum ersten fehlerhaften Test Die Zeit bis zur ersten fehlerhaften Rückmeldung ist deswegen von Bedeutung, da diese ein Maß dafür angibt, wie viel schneller der Entwickler beginnen kann, die erste fehlerhafte Rückmeldung der Testsuite-Ausführung umzusetzen. Solange die APFD-Werte keine zu große Streuung vorweisen, ist deswegen davon auszugehen, dass vor allem die Zeitangabe bis zur ersten Rückmeldung ausschlaggebend für den Nutzen einer Strategie ist.

Diese Angabe ist aber nur für den Fall relevant, dass tatsächlich noch Fehler durch die Testsuite aufgedeckt werden, da ansonsten keine frühere Garantie über die Fehlerfreiheit der Testsuite gegeben werden kann. Damit müsste die Zahl auch im Verhältnis der im Durchschnitt gesamt notwendigen Testsuite-Durchläufe betrachtet werden, was allerdings im Rahmen dieser Untersuchung nicht möglich ist.

5.1.3 Verwendete Diagramme Der Durchschnitt der beiden vorgestellten Metriken ist dabei ein gutes Richtmaß, ob eine Strategie im Gesamten zu besseren Werten geführt hat. Jedoch gibt dieser Wert keinen Einblick darüber, wie die einzelnen Werte verteilt sind und ob diese sich in einzelnen Bereichen häufen. Um auch Einblick über die Streuung APFD-Werte und die Zeitwerte bis zum ersten fehlerhaften Test zu erhalten, wurden auch Histogramme, Streudiagramme der einzelnen Strategien und Box-Plots zum Vergleich aller Strategien erstellt. Dabei bezieht sich eine dieser Grafiken immer auf eine Testsuite eines ausgewerteten Commits bezüglich der zur Priorisierung verwendeten Strategie. Die Histogramme und Streudiagramme geben einen groben Eindruck, in welchen Bereichen sich ein Großteil der Werte befindet und ob in diesen Bereichen Schwerpunkte der Verteilung vorliegen.

Box-Plot-Diagramme sind ein geläufiges Werkzeug zur Darstellung der Verteilung und Streuung statistisch erhobener Daten. Der Vorteil mehrerer Box-Plots in einer Grafik zum Vergleich ist, dass die Bereiche der größten Verteilung in einem Diagramm quantifiziert werden. Jeder Datensatz wird dabei als ein Rechteck dargestellt, das die Hälfte der Daten umfasst und durch das obere und untere Quartil abgegrenzt ist. Die Quartile können durch den Median, der Trennlinie innerhalb des Rechtecks, voneinander unterschieden werden. Der Bereich von einem Quartil bis zum Median markiert dabei den Bereich, in dem ein Viertel der Daten sich befindet. Die beiden Linien, welche senkrecht auf dem Rechteck stehen, werden als *Whisker* bezeichnet. Der senkrechte Strich dieser Linie markiert dabei jeweils das Minimum, beziehungsweise das Maximum des Datensatzes. Alle außerhalb davon liegenden Punkte bezeichnet man als Ausreißer und beginnen bei der am weitesten verbreiteten Definition ab einem Abstand, der größer ist als das $1,5$-fache des Interquartilsabstands, was die Spannweite des Rechtecks, in dem 50% der Daten liegen, charakterisiert [49].

5.2 Strategie der Auswertung

Anstelle von syntaktischen Mutationen wurde, um möglichst realistische Fehler zu erhalten, das implementierte Verfahren anhand von historischen Projektdaten ausgewertet. Dies hat auch vor allem den Vorteil, dass bezüglich einer Auswertung ein praxisnaher Eindruck entsteht, welche Art von Modifikationen der Software

effizient und eindeutig priorisiert werden können und wie sich deren Performanz bezüglich der ausgewerteten Metriken im statistischen Mittel verhält.

Dafür wurden die Projekte mithilfe von Git jeweils auf den Stand jedes früheren Commits gesetzt, während die Änderungen an den Tests für diesen Commit verworfen wurden, indem diese auf den Stand des vorausgehenden Commits gesetzt wurden. Dieses Vorgehen zur Auswertung wird in den folgenden Unterabschnitten näher beleuchtet. So wurden für ein Projekt von makandra names die-*radfahrausbildung.de* 94 von 420 Commits mit für die Auswertung relevanten Fehlerdaten ausfindig gemacht.

5.2.1 Vorbereiten des Repository- und Projektzustands Damit die Testsuite korrekt und effizient ausgewertet werden konnte, mussten das Repository und die Projektabhängigkeiten in dem jeweils aktuell zurückgesetzten Zustand für die Testsuite-Auswertung vorbereitet werden. Zuerst mussten einige Dateien automatisiert überschrieben werden, so dass die Konsolenausgabe der Testsuite kürzer ausfällt, das zur Priorisierung eingebundene Gem installiert wird und flackernde Tests neu ausgeführt werden.

Zu dem Zweck, dass die notwendigen Datenbanken- und Fremdsoftware-Abhängigkeiten und deren Einbindungen in die Web-Anwendung immer für den ausgecheckten Commit erfüllt sind, mussten deswegen für jeden Commit die gesamte Datenbank verworfen und neu migriert werden und alle Framework-Abhängigkeiten neu mit Hilfe des verwendeten Paket-Managers installiert werden. Der Paket-Manager dient dazu, die in einer zentralen Datei spezifizierten Versionen und Abhängigkeiten aller eingesetzter Fremdsoftwarepakete aufzulösen und zur Verwendung zu verlinken.

5.2.2 Auswertung eines Commits Um zuerst fehlerfreie Coverage-Daten zu erhalten, wurde das Repository auf den Commit vor dem auszuwertenden Commit zurückgesetzt, nach dem Vorgehen in 5.2.1 vorbereitet und die ganze Testsuite laufen gelassen. Als nächstes wurde das Projekt auf den auszuwertenden Commit gesetzt und das Projekt erneut analog zu 5.2.1 für diesen Zustand vorbereitet. Danach wurde die Testsuite für die zufällige und die umgesetzten coverage-basierten Priorisierungen ausgeführt. Zum Auslesen der aufgezeichneten fehlerfreien Coverage-Daten müssen für die Ausführung des implementierten Priorisierungs-Tools die Dateien in den dafür vorgesehenen Ordner hinterlegt werden.

Zur abschließenden Auswertbarkeit der Testsuite-Durchläufe musste für jeden einzelnen Tests eines Durchlaufs eine Vielzahl an Daten erhoben und gespeichert werden. Diese umfassten die Reihenfolge der ausgeführten Tests mit ihren jeweiligen Ausführungszeiten und Endergebnissen.

5.2.3 Verworfene Commits Commits, die aufgrund unterschiedlicher Ursachen zu keinem Fehler geführt haben, müssen verworfen werden. Eine Ursache war, dass Dateien umbenannt wurden und dadurch ihre zugehörigen Klassen- oder Modulreferenzen in den jeweiligen Tests nicht mehr gefunden werden konnten. Auch Commits, in denen keine Tests oder nur Tests zur Behebung flackernder Zustände modifiziert wurden, sind verworfen worden, da diese durch das Fehlen nennenswerter Modifikationen auch keine relevanten fehlerhaften Testzustände erzeugen können.

Ebenso wurden Veränderungen, welche vor allem das Frontend betroffen haben, verworfen, weil diese in großer Häufigkeit nur Modifikationen von Template-Views, Javascript- und CSS-Dateien enthalten haben. Unter einem Template-View ist zu verstehen, dass eine HTML-Datei mit Ruby-Code interpoliert wird, so dass diese dynamische Logik der Anwendung zur Anzeige enthalten kann. Wenn auch für die Auswertung Template-Coverage für Änderungen in Views aktiviert wurde, so konnten viele Test-Case-Abhängigkeiten der JS- und CSS-Dateien trotzdem nicht ausfindig gemacht werden, da diese auch nicht in der in Ruby aufgezeichneten Coverage enthalten waren. Dies hat auch dazu geführt, dass die berechnete Priorisierung für Modifikationen in diesen Dateien den anderen weit unterlegen waren.

In vielen dieser Commits hat dies dennoch einen ersten Anhaltspunkt geliefert, da die Code-Änderungen der Dateien oft derart zusammenhängen, dass Änderungen in den Views mit Änderungen der zugehörigen JavaScript- oder CSS-Dateien einhergehen oder umgekehrt. Dies liegt daran, dass HTML-Elemente im DOM über einen Selektor in CSS oder JavaScript referenziert werden und Änderungen in einer dieser Dateien auch die anderen betreffen kann. Ein Selektor kann den Typ, ein Attribut oder Kombinationen davon eines HTML-Elements spezifizieren und sich dadurch auf eines oder mehrere HTML-Elemente beziehen, die diesen Selektor erfüllen.

Da einige der ausgewerteten Commits flackernde Testergebnissen enthielten, wurden fehlerhafte Tests bis zu drei mal hintereinander durchgeführt. Wenn der aktuell ausgewertete Commit danach weiterhin flackernde Testergebnisse enthält, wurden diese Tests aus der Auswertung ausgeschlossen. Diese Tests konnten dadurch eindeutig identifiziert werden, dass sie entweder nur im zufälligen oder in einem der coverage-basierten Durchläufe fehlgeschlagen sind. Wenn es auch nicht ganz ausgeschlossen werden kann, so wurde dennoch zur Vereinfachung angenommen, dass ein Test, der in einem der auszuwertenden Durchläufe fehlgeschlagen ist, kein flackernder Test ist.

5.3 Ergebnisse der Auswertung

Zuerst sollen die Ergebnisse einzeln für die unterschiedlichen Strategien präsentiert und im darauf folgenden Unterkapitel 5.3.7 miteinander verglichen werden.

Für jede der ausgewerteten Strategien wurden auch die Histogramme und Streudiagramme hinzugefügt.

5.3.1 Zufällige Priorisierung Hier waren die Ergebnissen, wie in Abbildung 2a und 2b dargestellt, aufgrund der Gleichverteilung die APFD-Werte um den Wert $0,5$ gestreut, was auch an dem berechneten Durchschnitt von $0,503$ und dem Median von $0,52$ zu erkennen ist. Bei der Rückmeldung bis zum ersten fehlgeschlagenen Test konnte festgestellt werden, dass ein Großteil der Werte sich im Zeitraum von bis zu einer Minute befand, was auch in Abbildung 2c und 2d abzulesen ist. Darüber hinaus gab es eine recht große Verteilung mit dem höchsten Wert bei 07:30 Minuten. Das erste Feedback trat im Durchschnitt nach 01:42 Minuten auf. Der Median der Zeitwerte lag allerdings bei 01:10 Minuten.

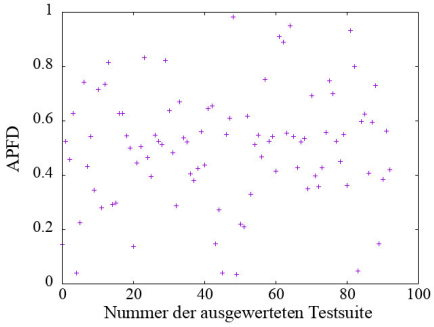
5.3.2 Absolute Priorisierung Bei den APFD-Werten ist anhand Abbildung 3a und 3b ersichtlich, dass diese weitaus weniger gestreut sind und sich vor allem zwischen $0,9$ und $1,0$ befinden und danach gleichmäßig abnehmen. Der Durchschnitt dieser Werte befand sich bei $0,89$ und der Median bei $0,94$. Bei der Zeit bis zur ersten Rückmeldung ist nach Abbildung 3c und 3d zu sehen, dass die meisten Werte auch im Bereich von bis zu einer Minute sind, während jedoch die Werte darüber weitaus seltener auftreffen und sich der maximale Wert bei 03:55 Minuten befindet. Dies war auch an der durchschnittlichen Rückmeldezeit von 00:48 Minuten und einem Median von 00:28 Minuten zu erkennen.

5.3.3 Additionale Priorisierung Ähnlich zur absoluten Priorisierung liegt ein Großteil der APFD-Werte zwischen $0,85$ und $1,0$. Nach Abbildung 4a und 4b ist zu erkennen, dass diese erstmal langsamer abnehmen, während eine weitere Häufung der Werte im Bereich 0.6 bis 0.7 vorliegt. Der Durchschnitt dieser Werte befand sich bei 0.85 und der Median bei 0.89. Bezüglich der Rückmeldezeit sind nach Abbildung 4c und 4d die meisten Werte unter 1 Minute zu finden, was sich auch im Durchschnitt von 00:31 Minuten und einem Median von 00:15 Minuten erkennen lässt. Der schlechteste Wert lag bei 02:35 Minuten.

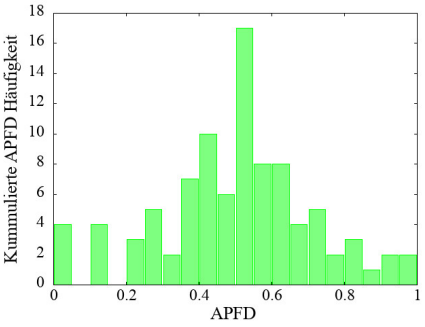
5.3.4 Pseudo-additionale Priorisierung Bei der pseudo-additionalen Testabdeckung gab es nach Abbildung 5a und 5b bei den APFD-Werten einen APFD-Durchschnitt von $0,89$ und Median von 0.93. Jedoch waren im Gesamten weniger APFD-Werte im Bereich $(0,9; 1,0)$ aufzufinden. Die durchschnittliche Rückmeldezeit nach Abbildung 5c und 5d lag bei 00:40 Minuten, während der schlechteste Wert auch bei 03:55 Minuten lag. Der Median der Zeitwerte war bei 00:19 Minuten anzutreffen.

Abb. 2: Auswertungsdiagramme der zufälligen Priorisierung

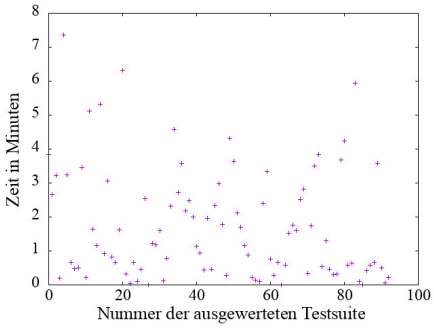
(a) Streudiagramm der APFD-Werte



(b) Histogramm der APFD-Werte



(c) Streudiagramm der vergangenen Zeit bis zum ersten fehlerhaften Test



(d) Histogramm der vergangenen Zeit bis zum ersten fehlerhaften Test

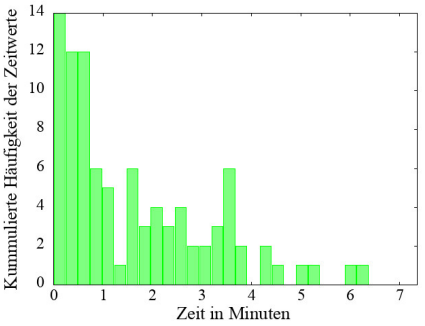
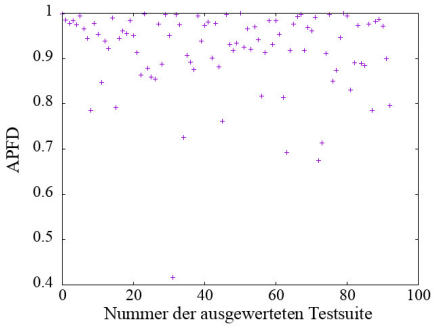
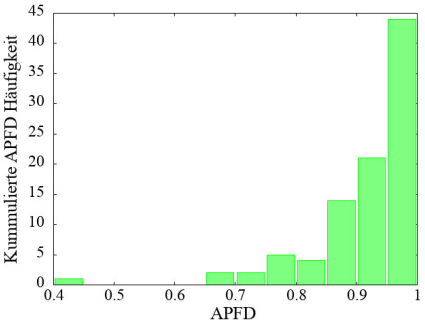


Abb. 3: Auswertungsdiagramme der Priorisierung nach absoluter Coverage

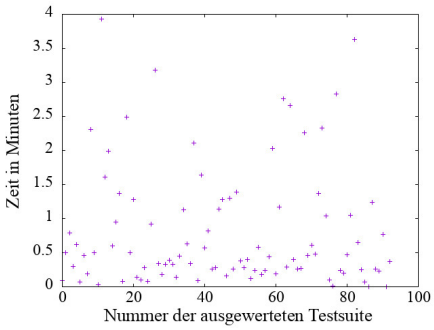
(a) Streudiagramm der APFD-Werte



(b) Histogramm der APFD-Werte



(c) Streudiagramm der vergangenen Zeit bis zum ersten fehlerhaften Test



(d) Histogramm der vergangenen Zeit bis zum ersten fehlerhaften Test

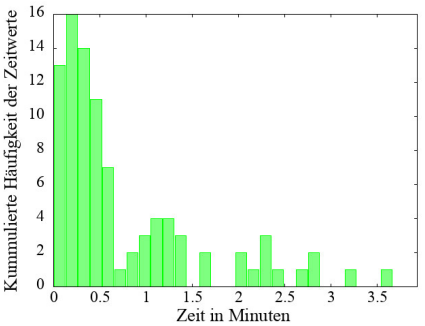
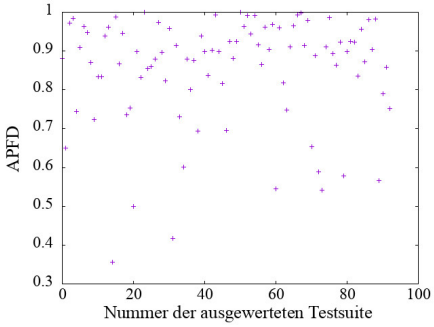
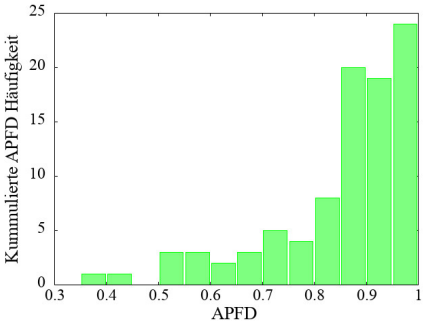


Abb. 4: Auswertungsdiagramme der Priorisierung nach additionaler Coverage

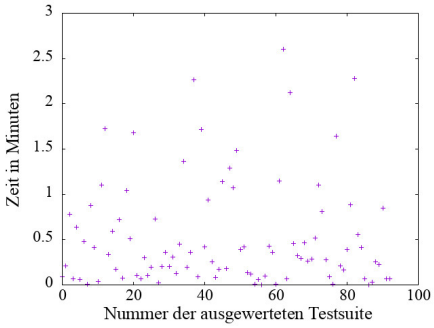
(a) Streudiagramm der APFD-Werte



(b) Histogramm der APFD-Werte



(c) Streudiagramm der vergangenen Zeit bis zum ersten fehlerhaften Test



(d) Histogramm der vergangenen Zeit bis zum ersten fehlerhaften Test

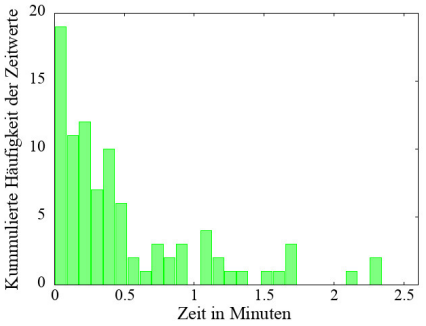
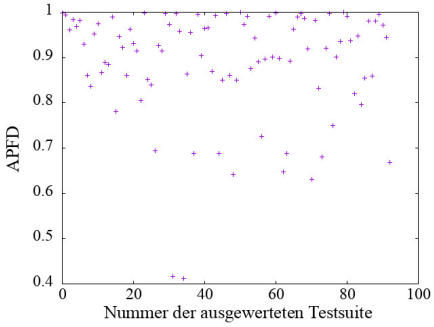
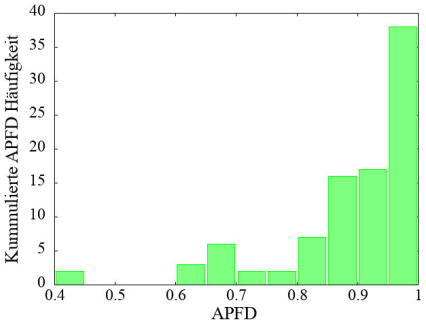


Abb. 5: Auswertungsdiagramme der Priorisierung nach pseudo-additionaler Coverage

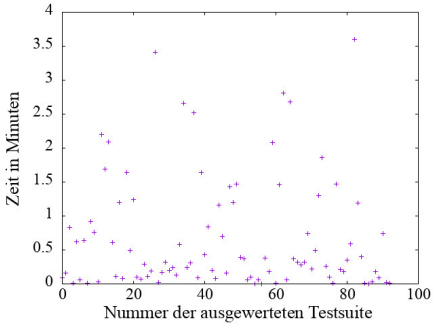
(a) Streudiagramm der APFD-Werte



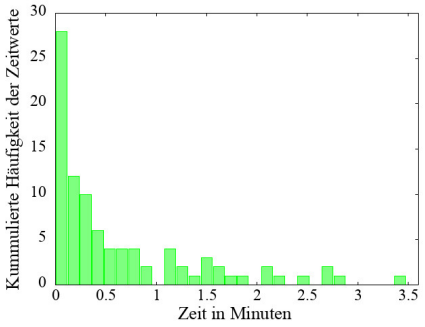
(b) Histogramm der APFD-Werte



(c) Streudiagramm der vergangenen Zeit bis zum ersten fehlerhaften Test



(d) Histogramm der vergangenen Zeit bis zum ersten fehlerhaften Test



5.3.5 Exemplarische Auswertung der Ausreißer Es wurden vereinzelt Commits, die im Box-Plot 6 als Ausreißer dargestellt wurden und damit besonders schlechte Werte vorweisen, anhand der APFD-Graphen, des geänderten Codes und der vorgeschlagenen Priorisierung ausgewertet. Viele dieser Commits hatten gemein, dass hier Dateien verändert wurden, die in sehr vielen Bereichen der Software aufgerufen werden. Im Zusammenhang mit der Coverage bedeutet dies, dass sehr viele nicht relevante Testdateien vorselektiert werden, womit eine hohe Wahrscheinlichkeit besteht, dass die fehlgeschlagenen Tests über einen Großteil der Testsuite verteilt sind.

Dies ist beispielsweise dann der Fall, wenn häufig verwendete Funktionen der User-Klasse verändert werden. Da die User-Klasse für sehr viele Tests benötigt wird, die eine Autorisierung voraussetzen oder mit dem User assoziiert sind, muss in all diesen Tests ein Objekt des Users angelegt werden. Aufgrund dessen ist auch in den Coverage-Daten dieser Tests jeweils der ausgeführte Programmcode der User-Klasse enthalten. Ein anderes Beispiel sind View-Templates, die sehr in vielen Views der Anwendung aufgerufen werden, wie eine Navigationsleiste. Die grundlegende Problematik ist, dass die absolute Anzahl an Aufrufen in so einem Beispiel potentiell nicht relevanten Tests einerseits höhere Priorität zuordnet und andererseits eine große Anzahl an Tests ausgewählt wird.

Tatsächlich könnte als einfacherer Ansatz in Betracht gezogen werden, dass die Coverage das Treffen einer Code-Zeile nur einmalig zählt, damit die Signifikanz der absoluten Werte abgemildert wird. Falls diese Strategie im Gesamten zu schlechteren Ergebnissen führen würde, wäre eine Option, nur selektiv für Commits anzuwenden, bei denen eine überdurchschnittlich große Anzahl an Tests vorselektiert wird. Um Verfahren zu entwickeln, die mit diesen Schwachstellen effizienter umgehen, könnte eine systematische Analyse all dieser Commits in Betracht gezogen werden. Interessante Informationen wären, ob dies in Zusammenhang mit bestimmten Code-Modifikationen, Dateien oder Anforderungen besonders häufig auftritt und ob es eine Anzahl an ausgewählten Tests gibt, bei der dies mit bedeutend höherer Wahrscheinlichkeit auftritt.

5.3.6 Ausführungszeiten Zur Bestimmung der Laufzeit wurde jede Strategie exakt 5 Mal laufen gelassen und für jede Strategie dann der Durchschnitt aller festgestellten Werte berechnet. Es wurden keine Laufzeiten in Abhängigkeit zur Testsuite-Größe ausgewertet, da bei Auswertung der Strategien hierzu die Ausführungszeit linear zur Testsuite-Größe zugenommen hat.

Anhand der Werte in der Tabelle 2 zeigt sich, dass die zufällige Priorisierung ohne das eingesetzte Priorisierungs-Tool im Durchschnitt 8 Sekunden schneller ist als mit Einsatz davon. Bezüglich der Ausführungszeit der pseudo-additionalen, absoluten und der additionalen Strategie konnte nur ein geringer Unterschied von einer Sekunde zueinander festgestellt werden, während die absolute Priorisierung mit dem durchschnittlichen Wert von 05:40 zwischen 10 bis 20 Sekunden länger

Tabelle 2: Ausführungszeiten der Testsuite mit und ohne Priorisierung

Testpriorisierung	**Laufzeit in Minuten**					
	1	**2**	**3**	**4**	**5**	**Ø**
Zufällig ohne Priorisierungstool	05:22	05:18	05:20	05:21	05:25	05:21
Zufällig mit Priorisierungstool	5:27	05:35	05:30	05:26	05:29	05:29
Absolute Priorisierung	5:28	05:50	05:45	05:43	05:34	05:40
Additionale Priorisierung	5:26	05:29	05:29	05:27	05:26	05:27
Psueo-additionale Priorisierung	5:37	05:31	05:19	05:27	05:25	05:28

benötigt als alle anderen Strategien mit Priorisierungs-Tool. Da mit jeder der Strategien die erste Rückmeldung durchschnittlich mindestens 54 Sekunden früher vorliegt, stellt die Ausführungszeit nach bisher aufgenommenen Daten keine wichtige Kenngröße zur Wahl eines Priorisierungsverfahrens dar.

5.3.7 Vergleich der Ergebnisse Anhand Abbildung 6 ist zu erkennen, dass alle Strategien bezüglich der APFD-Metrik besser als die zufällige Strategie abgeschnitten haben, wobei die absolute Priorisierung die besten Werte geliefert hat und pseudo-additional und additional den zweiten und dritten Platz einnehmen. Dieser Zusammenhang ist auch an den jeweiligen Durchschnittswerten in Tabelle 3 und des Box-Plot-Diagramms 6 zu erkennen. Anzumerken ist auch, dass die additionale Priorisierung besonders viele Ausreißer vorzuweisen hat. Besonders soll hervorgehoben werden, dass die APDF-Werte der absoluten Priorisierung zu besseren Werten als die der additionalen Priorisierung geführt haben. Dies ist deswegen interessant, da dies im Gegensatz zu früheren Forschungsergebnissen steht, bei denen die additionale Priorisierung der Absoluten überlegen gewesen ist [49].

Tabelle 3: Durchschnittswerte der Strategien im Vergleich zueinander

Strategie der Priorisierung	**APFD**		**Zeit bis zum ersten Fehler**	
	Ø	**Median**	**Ø**	**Median**
Zufällig	0.50	0.52	01:42	01:10
Absolut	0.91	0.94	00:48	00:28
Additional	0.85	0.89	00:31	00:15
Pseudo-additional	0.89	0.93	00:40	00:19

Bei den Zeitwerten hat sich, wie in Abbildung 6 und zu erkennen ist, ergeben, dass die additionale Priorisierung bezogen auf diese Metrik wiederum am Besten abgeschnitten hat, gefolgt von pseudo-additionalen und absoluten Strategie. Dies ist auch anhand der Durchschnittswerte und Mediane in der Tabelle 3 zu erkennen.

Abb. 6: Box-Plots der APFD-Werte aller Strategien

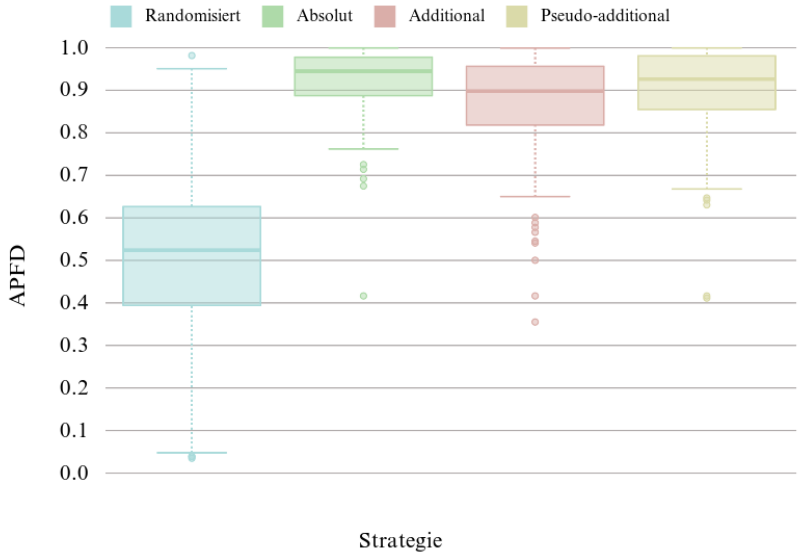
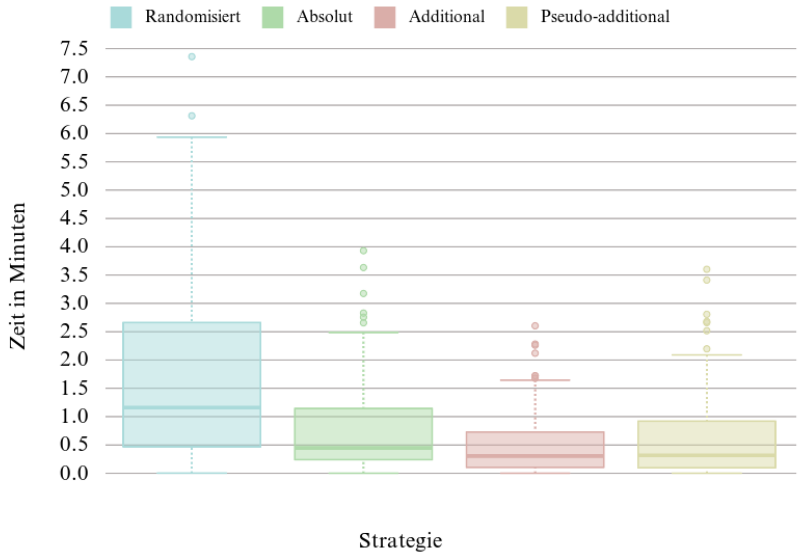


Abb. 7: Box-Plots der Zeitwerte bis zum ersten fehlerhaften Testergebnis aller Strategien



Da bei allen Methoden die Rückmeldung früher vorliegt als bei der Zufälligen, kann für alle Werte die Zeitersparnis berechnet werden. Bemerkenswert ist auch, dass mit der schlechtesten Methode bereits eine durchschnittliche Zeitersparnis von 54 Sekunden vorgelegen ist und bei der besten, der additionalen Priorisierung, sogar 01:11 Minuten. Mit jeder der behandelten Strategie konnte eine Wartezeit von mindestens 00:40 bis zu 01:11 Minuten eingespart werden, was bei einer Anzahl von 25 Commits pro Tag durchschnittlich eine Zeitersparnis bei 40 Sekunden eine Einsparung von circa 16:30 Minuten pro Tag bedeuten würde.

Im Gesamten lässt sich schlussfolgern, dass bezogen auf die die beiden Metriken sich die pseudo-additionale Strategie am robustesten erwiesen hat, während die absolute und additionale Strategie jeweils größere Schwachstellen in einer der beiden Metriken vorwiesen, jedoch in der anderen sich als bestes Verfahren herausgestellt haben. Bei der additionalen Priorisierung sticht noch besonders heraus, dass die APFD-Werte über viele Ausreißer und eine große Streuung aufweist. Welche der Coverage-Verfahren der Implementierung tatsächlich verwendet werden würden, müsste davon abhängig gemacht werden, ob eine Metrik für die Praxis eine höhere Bedeutung hat als die andere oder ob sie in beiden Metriken gleich gut performt werden sollte.

5.4 Einschränkungen

Bei den erstmal vielversprechenden Ergebnissen ist auch zu bedenken, dass im Rahmen dieser Untersuchung bisher nur eine Software ausgewertet wurde. Deswegen wurde in einer früheren Arbeit von Rothermel et al. [49] eine große Varianz innerhalb der erreichten Metriken einer Strategie bei verschiedenen Programmen gefunden, was auch der Grund ist, dass dafür dann eine Varianzanalyse zum Vergleich der Mittelwerte der Strategien durchgeführt wurde. Eine Varianzanalyse ist ein statistisches Verfahren, um herauszufinden, ob bei verschiedenen Gruppen, den sogenannten unabhängigen Variablen, im Mittelwert ein signifikanter Unterschied bezüglich einer untersuchten abhängigen Variable vorliegt.

Eine derartige Analyse wurde ausgeschlossen, da im Kontext dieser Arbeit und der prototypischen Implementierung nur erste Indizien geliefert werden konnten. Zumal, da zur zeitliche Einschränkung der Ausarbeitung das Priorisierungs-Tool nur für RSpec umgesetzt wurde und damit eine große Anzahl von Projekten bei makandra, die nach wie vor anstelle von RSpec Cucumber für Integrationstests einsetzen, nicht ausgewertet werden konnte.

Da mit jedem eingesetzten Tool eben eine weitere Komplexität hinzukommt und die Implementierung nach 5.5 Kosten verursacht, wäre hier eine Kosten-Nutzen-Analyse zur Amortisierung der Kosten notwendig. Für eine tatsächliche Kosten-Nutzen-Analyse der Verfahren müssten die Zeitwerte bis zum ersten fehlgeschlagenen Test noch prozentual auf die aktuelle Testsuite-Größe normiert

werden, da diese die gesamte Ausführungszeit bestimmt und die Zeitersparnis davon abhängig ist.

Im Rahmen der Auswertung bedeutet dies, dass die Zeit bis zur ersten Rückmeldung im Verhältnis zur gesamten Ausführungszeit berechnet und der Durchschnitt davon gebildet werden würde. Dies müsste dann noch im Verhältnis aller ausgeführten Testsuite-Durchläufe betrachtet werden, die überhaupt einen fehlerhaften Zustand enthalten, da nur für diese mit einer Zeitersparnis zu rechnen ist.

5.5 Eignung und Limitierungen der Implementierung

Allem voran spricht für die Umsetzung, dass die festgestellte Fehlererkennungsrate und frühere Erkennung des ersten Fehler der Zufälligen überlegen ist. Zusätzlich fällt die erste Analyse positiv zur Einsetzbarkeit aus, da die grundlegenden Kriterien der Problemdomäne praktisch gelöst werden können. Obwohl sich bei der prototypischen Implementierung herausgestellt hat, dass das Verfahren die grundlegenden Kriterien löst, sind neben diesen vielversprechenden Aspekten auch Limitierungen des Ansatzes aufgefallen, welche bei einer Umsetzung als Software zum dauerhaften Priorisierungsverfahren bei makandra beachtet werden sollten und deswegen im Folgenden behandelt werden.

Neben den Limitierungen werden bei einigen Punkten auch mögliche Lösungsansätze kurz skizziert und diskutiert, da sich viele davon bereits durch die Implementierung, die Auswertung und den manuellen Tests angedeutet haben. Die Bedeutung der dargestellten Einschränkungen für die Eignung des Verfahren ist, dass sie eine Anpassung oder Erweiterung des Verfahrens als Softwarelösung zum Einsatz bei makandra verlangen würden. In einigen Fällen wäre die Umsetzung dessen zwingend notwendig, während Sie in anderen zu einer effizienteren Einsetzbarkeit und Handhabung der Software führen würden. Viele der vorgestellten Beschränkungen zeigen auch Punkte auf, die besonders bei der Umsetzung eines coverage-basierten Ansatzes und auch für weitere Ansätze bedacht werden müssten.

5.5.1 Automatisches Triggern der Testsuite-Ausführung

Eine weitere praktische Einschränkung ist, dass immer erst manuell die Testsuite gestartet werden muss, anstatt dass nach Erkennen betroffener Änderungen, beispielsweise bei Abschluss eines Commits, direkt nach bestimmten Zeitintervallen die Testsuite gestartet wird. Dies könnte mit geeigneter Nutzer-Rückmeldung so umgesetzt werden, dass schneller anhand der fehlgeschlagenen Tests der modifizierte Code angepasst werden kann.

5.5.2 Integration in weitere Praktiken des Regressionstestens Die Software zur Priorisierung sollte auch mit gängigen Praktiken des Regressionstesten bei makandra kompatibel sein. Das umfasst vor allem die häufige Verwendung von CI, Parallelität der Testsuite-Ausführung und Unterstützung aller bei makandra eingesetzten Testtools, da im Rahmen dieser Arbeit eine Einschränkung auf RSpec getroffen wurde.

Die parallel priorisierte Ausführung sollte dabei lokal und in der CI möglich sein. Um mit parallelen Testsuite-Durchläufen verwendbar zu sein, müsste die Priorisierung in eine variable Anzahl von Gruppen aufgeteilt werden können. Für diese parallel laufenden Gruppen müsste garantiert sein, dass bei paralleler Ausführung keine Deadlocks durch simultanes Auslesen oder Schreiben der Coverage-Daten entstehen.

Es sollte auch geprüft und bedacht werden, ob mit Verwendung dieses Tools weiterhin Testreihenfolgen effizient genug erkannt werden können, wofür auch die Umsetzung von 5.5.3 eine wichtige Rolle spielen könnte.

5.5.3 Reihenfolge der nicht vorselektierten Tests Ein Schwachstelle des bisher implementierten Prototyps ist, dass er nach der ersten Selektion aller Tests, die Coverage für veränderte Dateien aufweisen, alle übrigen Tests in der Reihenfolge hinzugefügt, wie diese in den Coverage-Daten enthalten sind. Hier wäre eine Optimierung, diese Tests zumindest in zufälliger Reihenfolge hinzuzufügen. Eine nützliche Erweiterung könnte sein, nach den ausgewählten Tests potenziell flackernde Tests, die aus vergangenen Commits oder der CI-Historie ausgewertet werden könnten, hinzuzufügen, damit diese tendenziell früher gefunden werden. Allerdings würde dies immer noch nicht das Ausführen der kompletten Testsuite verhindern, da Fehler beispielsweise auch aufgrund eingesetzter Fremdsoftware ausgelöst werden oder neue flackernde Testzustände auftreten können.

5.5.4 Sonderfälle der Versionsverwaltung Es müssten auch einige Sonderfälle, die durch die Verwendung der Versionsverwaltung entstehen könnten, daraufhin geprüft werden, ob diese für die Aufzeichnung der Coverage-Informationen richtig erkannt werden. Dies betrifft verschiedene Kommandos von Git, die verwendet werden, um Änderungen einzelner Dateien auf frühere Commits zurückzusetzen, einen Commits aus einer anderen Abzweigung an den aktuellen Branch anzuhängen oder verschiedene Commits zusammenzuführen.

Beim Zusammenführen und Anwenden eines anderen Commits könnte es passieren, dass die Änderungen nicht mehr im Status der aktuellen Veränderungen angezeigt werden, wenn die Testsuite davor noch nicht ausgeführt wurde. Hier wäre es eventuell notwendig, die Modifikationen nur bezüglich der Differenz des aktuellen Standes und dem ersten Commit des zugehörigen Branches auszuwerten. Sicherheitshalber könnte auch bei jedem ausgeführten Commit eine Testsuite

zur Aufzeichnung der Daten getriggert werden. Zumal, da dieselbe Problematik entstehen könnte, wenn ein Commit abgeschlossen wird, bevor die Testsuite ausgeführt wurde und damit die Coverage-Informationen der aktuellen Änderungen nicht ausgewertet werden konnten.

5.5.5 Fehlerhafte Testzustände bei der Coverage-Aufzeichnung Hinsichtlich der Vereinbarkeit der verschiedenen Software-Versionen hat sich gezeigt, dass die als Dateien gespeicherten Datenstrukturen mit neuen vereinen lassen, indem einfach die Coverage-Daten des letzten Durchlaufs für eine Test-Datei-Zuordnung überschrieben werden. Dazu wurde auch ein vereinfachtes Verfahren implementiert. Hierbei hat sich jedoch herausgestellt, dass die Zusammenführung nur dann erfolgen darf, wenn eine komplette Test-Datei ausgeführt wird und keiner der ausgeführten Tests fehlgeschlagen ist.

Sofern ein Test fehlgeschlagen ist, kann es sein, dass ein Teil der notwendigen Coverage-Informationen für diesen Test fehlen, weswegen es zu einem besseren Ergebnis führen könnte, diesen Test nicht zu verwenden. Andererseits kann auch gerade der fehlgeschlagene Test Informationen über Modifikationen an Änderungen enthalten. Zur Verwendung dieser Daten wäre denkbar, die Coverage-Matrix um eine Dimension zu ergänzen, die nur die aktuelle Differenz für fehlgeschlagene Tests speichert und sofern für einen Test eine Änderung vorliegt, diesen mit höherer Gewichtung in die Priorisierung einbezieht. Ein weiterer Vorteil ist, dass Änderungen der Coverage zu einer vorherigen Version mehr Information zu potenziellen Fehlern enthalten, als nur ob überhaupt eine Coverage vorliegt.

Allerdings ist ein grundlegendes Problem für Methoden, in denen einzelne Tests der aktuell ausgeführten Testdatei aufgrund eines fehlerhaften Zustands ausgeschlossen werden, dass Informationen über die fehlerhaften Tests gespeichert werden müssen mithilfe derer der Wert der Abdeckung dann bei einem späteren erfolgreichen Zustand identifiziert und überschrieben werden können. Die Schwierigkeit hierbei ist, dass sich diese Informationen zur Identifikation ändern können und dann trotz Modifikationen am Code eine eindeutige Zuordbarkeit gewährleistet sein muss. Hier wäre es denkbar, entweder die Zeilennummer mitsamt dem Dateipfad oder den Namen des Tests zu speichern. Dafür müssten also wiederum dieselben Voraussetzungen, wie in 4.1.2 beschrieben wurde, erfüllt werden können.

5.5.6 Differenz der Coverage zur Priorisierung Ein Nachteil des vorgestellten Verfahrens ist, dass für die Priorisierung bisher nur verwendet wird, ob überhaupt eine Coverage für bestimmte Ausführungspfade vorliegt. Jedoch könnte alternativ die Priorisierung der Dateien auch danach erfolgen, ob zusätzlich eine Differenz der Coverage zu einem vorherigen Commit vorliegt. Diese Änderung würde indizieren, dass sich an den Ausführungspfaden der Tests etwas verändert hat und macht diese damit besonders anfällig für Fehler und damit auch zur Priorisierung.

5.5.7 Coverage für JavaScript- und CSS-Dateien Eine weitere größere Einschränkung ist, dass der vorgestellte Ansatz bisher nicht mit JavaScript und CSS kompatibel ist, wie in 5.2 ausgeführt wurde und weswegen $\frac{2}{5}$ aller Commits nicht zur Auswertung verwendet wurden. Allerdings ist hierbei auch wichtig in Betracht zu ziehen, dass viele dieser Commits nach wie vor zu einem besseren Ergebnis mithilfe der implementierten Priorisierung geführt haben, was in vielen Fällen eine schlechte Performanz aufwies. Zur besseren Einschätzung dieser Problematik könnten auch weitere Auswertungen zur Häufigkeit und Ursachen der vorkommenden Fehler anhand mehrerer Projekte in Betracht gezogen werden, da das in dieser Arbeit behandelte Projekte besonders viel JavaScript-Code vorweist.

Es wäre möglich, den Ansatz für JavaScript- und CSS-Coverage zu erweitern und in die vorhandene Datenstruktur diese Dateien miteinzubeziehen. Ein etwas einfacherer Ansatz wäre, die Dateien über ihre CSS-Selektoren, welche ein HTML-Element referenzieren, jeweils zuzuordnen. Das heißt, wenn beispielsweise der einem Selektor zugehörige Code in CSS- oder JS-Dateien geändert wurde, können jeweils die Tests ausgeführt werden, die Views betreffen, in denen der betroffene Selektor enthalten ist. Für JavaScript ist zusätzlich noch zu bedenken, dass es in JavaScript geschriebene Tests für Komponenten geben kann, die je nach dem Testsuite Umfang selbst auch wieder priorisiert werden müssen. Für diese Tests wäre hier eine eigens angefertigte Implementierung des Verfahrens in JavaScript notwendig.

5.5.8 Versionsverwaltung der Coverage-Dateien Eine praktische Problematik ist, dass die Coverage-Informationen immer erst vorliegen, sobald die Testsuite mindestens einmal gelaufen ist. Als Lösungsansatz können die Dateien jeweils in Git eingecheckt werden, damit gleich zu Beginn einer neuen Aufgabe die Coverage-Informationen bereits vorliegen. Eine Problematik ist, dass diese Dateien dann in jedem *Merge-Request* und in jeder angezeigten Differenz der aktuellen Änderungen bei der Verwendung des Git-Kommandos *diff* angezeigt werden. Unter einem Merge-Request versteht man das Zusammenführen (Merge) verschiedener Abzweigungen. Dabei ist es üblich, alle Differenzen zweier Branches sich in einem Merge-Request anzeigen zu lassen, um in einem *Review*, der Durchsicht der Modifikationen eines erfahrenenen Entwicklers, auf Fehlerfreiheit, Qualität und Konsistenz der Software-Anforderungen überprüft werden zu können.

Um das zu verhindern, gibt es in Git die Möglichkeit in einer globalen Konfigurationsdatei, Dateipfade beim Erstellen einer Differenz an Veränderungen auszuschließen. Eine größere Schwierigkeit ist jedoch, dass, sofern die Dateien per Versionsverwaltung eingecheckt werden, stets darauf zu achten ist, dass nur korrekte Versionen der Dateien hinzugefügt werden. Dies betrifft insbesondere fehlgeschlagene Tests, sowie gelöschte und hinzugefügte Test-, bzw. Code-Dateien. Folglich müsste hierfür ein Lösungsansatz eingesetzt werden, der sicherstellt, dass mit jeder Test-Ausführung darauf geachtet wird, dass bei Abschluss eines Commits nur vollständige Coverage-Daten vorliegen. Eine Möglichkeit zur Lösung

wäre, die Dateien von der CI in das Repository einzuchecken, sobald die Testsuite erfolgreich durchgelaufen ist.

5.5.9 Löschen von Coverage-Dateien Eine Schwierigkeit tritt auch dann auf, wenn Code-Dateien gelöscht werden. Wenn diese aus den Coverage-Daten damit direkt entfernt werden würden, wären die Informationen für die Priorisierung nicht mehr abrufbar. Allerdings muss gerade auch beim Löschen von Dateien darauf geachtet werden, dass Code, der Funktionen dieser Dateien aufruft, weiterhin intakt ist. Dementsprechend sind diese Ausführungspfade der aufgezeichneten Coverage-Informationen auch weiterhin zur Priorisierung relevant und sollten nicht direkt mit dem Entfernen der Datei gelöscht werden.

Sofern eine feinere Granularität als in der Implementierung eingesetzt werden würde, wäre diese Limitierung sogar besonders relevant, da Löschen von Funktionen und insbesondere von Zeilen weitaus häufiger auftrifft. Dabei müsste sogar optimalerweise noch unterschieden werden, ob ein Code nur entfernt wurde, um in einem anderen Kontext eingefügt zu werden. Eine ähnliche Problematik ergibt sich auch für Umbenennungen einer Datei oder eine Funktion bei Verwendung von Datei- bzw. Funktions-Coverage.

6 Schlusswort und Ausblick

Es hat sich gezeigt, dass die prototypische Implementierung des ausgewählten Verfahrens im Rahmen dieser Arbeit erste Indizien zu einer besseren Performanz der kritischen Metriken geführt hat. Um abzuschätzen, ob das Verfahren tatsächlich im Umfeld von makandra anwendbar wäre, wurden Limitierungen der Implementation im Hinblick auf den Anwendungsbereich untersucht. Daraus ließ sich schließen, dass das Verfahren mit zusätzlichen Implementationsaufwand einsetzbar wäre. Nach einigen der hier herausgearbeiteten Punkten könnte an vielen Stellen die Handhabbarkeit und Performanz in bestimmten Fällen sogar noch weiter optimiert werden.

Besonders hervorzuheben ist auch die Analyse der Kriterien im Bezug zur testgetriebenen Entwicklung und der Web-Entwicklung, sowie einer Auswertungsstrategie, die möglichst praxisnahe Ergebnisse für die gegebenen Anforderungen liefern konnte. Das hat sich auch für die Bewertung einer industrienahen Anwendbarkeit äußerst relevant erwiesen. Die realitätsnahen Fehlerdaten konnten Muster der Modifikationen in Commits identifizieren, in denen die Algorithmen zur Priorisierung besonders schlecht abgeschnitten hatten. Durch die prototypische Implementierung konnten vielfältige Limitierungen herausgearbeitet werden, die generell innerhalb der Problemdomäne und allem voran für coverage-basierte Verfahren zu bedenken sind.

Diese Bewertung im Hinblick auf die Kriterien war auch der Grund, weswegen ein Großteil der Verfahren als primäres Priorisierungsverfahren ausgeschlossen wurden, jedoch als ergänzendes sekundäres Merkmal zur Priorisierung relevante Informationen beitragen könnten. Diesbezüglich hat sich gezeigt, dass vor allem modell-, suchbasierte und kombinierte Verfahren für weitere Forschungen als vielversprechend eingeschätzt wurden. Des Weiteren sollte die vorgestellten Coverage-Verfahren noch für unterschiedlichen Granularitäten der Programmabdeckung erweitert und untersucht werden, da hervorzuheben ist, dass höhere Granularität oft mit höheren APFD-Werten korreliert hat [49].

Für zukünftige Untersuchungen ist zu bedenken, dass eine Auswertung mehrerer Programme und Zeitwerte im Verhältnis zur Testsuite-Größe betrachtet werden sollte, wie im Abschnitt 5.4 ausgeführt wurde. Zu einer Kosten-Nutzen-Analyse wurde in diesem Abschnitt auch angeraten, eine Amortisierung der Kosten bezüglich der durchschnittlichen Anzahl ausgeführter Testsuites zu berechnen.

Abbildungsverzeichnis

Tabellenverzeichnis

Literatur

1. Activetype: Make any ruby object quack like activerecord — rubygems.org. https://rubygems.org/gems/active_type. [Accessed 29-May-2023].
2. BDD Testing & Collaboration Tools for Teams | Cucumber — cucumber.io. https://cucumber.io/. [Accessed 11-May-2023].
3. brakeman: Brakeman detects security vulnerabilities in ruby on rails applications via static analysis. https://rubygems.org/gems/brakeman. [Accessed 11-May-2023].
4. Can I use... Support tables for HTML5, CSS3, etc — caniuse.com. https://caniuse.com/. [Accessed 23-May-2023].
5. capybara-lockstep: Synchronize capybara commands with client-side javascript and ajax requests — rubygems.org. https://rubygems.org/gems/capybara-lockstep. [Accessed 29-May-2023].
6. class TracePoint - RDoc Documentation — ruby-doc.org. https://ruby-doc.org/3.2.2/TracePoint.html. [Accessed 11-May-2023].
7. Jasmine Documentation: A testframework for javascript — github.com. https://jasmine.github.io/index.html. [Accessed 11-May-2023].
8. JavaScript — developer.mozilla.org. https://developer.mozilla.org/en-US/docs/Web/JavaScript. [Accessed 16-May-2023].
9. modularity: Traits and partial classes for ruby — rubygems.org. https://rubygems.org/gems/modularity. [Accessed 29-May-2023].
10. module Coverage - coverage: Ruby Standard Library Documentation — ruby-doc.org. https://ruby-doc.org/3.2.2/exts/coverage/Coverage.html. [Accessed 11-May-2023].
11. numo-narray: Narray class library in ruby/numo. https://rubygems.org/gems/numo-narray. [Accessed 17-May-2023].
12. NumPy: Numeric multidimensional array operations — numpy.org. https://numpy.org/. [Accessed 17-May-2023].
13. parser: A ruby parser written in pure ruby. https://rubygems.org/gems/parser. [Accessed 11-May-2023].
14. Python — python.org. https://www.python.org/. [Accessed 17-May-2023].
15. reek: A tool that examines ruby classes, modules and methods and reports any code smells it finds. https://rubygems.org/gems/reek. [Accessed 11-May-2023].
16. rspec: Bdd for ruby. https://rubygems.org/gems/rspec. [Accessed 11-May-2023].
17. rubrowser: A ruby interactive dependency graph visualizer. https://rubygems.org/gems/rubrowser. [Accessed 11-May-2023].
18. Ruby on Rails: Model View Controller framework for web development — rubyonrails.org. https://rubyonrails.org/. [Accessed 11-May-2023].
19. Ruby Programming Language — ruby-lang.org. https://www.ruby-lang.org/en/. [Accessed 11-May-2023].
20. rugged: Rugged is a ruby bindings to the libgit2 linkable c git library — rubygems.org. https://rubygems.org/gems/rugged. [Accessed 17-May-2023].
21. Selenium: Framework for browser automation — selenium.dev. https://www.selenium.dev/. [Accessed 29-May-2023].
22. Standards - W3C — w3.org. https://www.w3.org/standards/. [Accessed 23-May-2023].
23. M. Aniche and M. A. Gerosa. Does test-driven development improve class design? a qualitative study on developers' perceptions. *Journal of the Brazilian Computer Society*, 21:1–11, 2015.

24. M. Anttonen, A. Salminen, T. Mikkonen, and A. Taivalsaari. Transforming the web into a real application platform: new technologies, emerging trends and missing pieces. In *Proceedings of the 2011 ACM Symposium on Applied Computing*, pages 800–807, 2011.
25. K. Beck. *Test-driven development: by example*. Addison-Wesley Professional, 2003.
26. R. Beena and S. Sarala. Code coverage based test case selection and prioritization. *arXiv preprint arXiv:1312.2083*, 2013.
27. A. Brito, M. T. Valente, L. Xavier, and A. Hora. You broke my code: understanding the motivations for breaking changes in apis. *Empirical Software Engineering*, 25:1458–1492, 2020.
28. T. Brooks. World wide web consortium (w3c). In *Encyclopedia of library and information sciences*, pages 5695–5699. CRC Press, 2009.
29. C. Catal and D. Mishra. Test case prioritization: a systematic mapping study. *Software Quality Journal*, 21:445–478, 2013.
30. P. K. Chittimalli and M. J. Harrold. Recomputing coverage information to assist regression testing. *IEEE Transactions on Software Engineering*, 35(4):452–469, 2009.
31. R. Connolly. Facing backwards while stumbling forwards: The future of teaching web development. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*, pages 518–523, 2019.
32. M. Eck, F. Palomba, M. Castelluccio, and A. Bacchelli. Understanding flaky tests: The developer's perspective. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 830–840, 2019.
33. S. Elbaum, A. G. Malishevsky, and G. Rothermel. Test case prioritization: A family of empirical studies. *IEEE transactions on software engineering*, 28(2):159–182, 2002.
34. E. Engström and P. Runeson. A qualitative survey of regression testing practices. In *Product-Focused Software Process Improvement: 11th International Conference, PROFES 2010, Limerick, Ireland, June 21-23, 2010. Proceedings 11*, pages 3–16. Springer, 2010.
35. H. Erdogmus, M. Morisio, and M. Torchiano. On the effectiveness of the test-first approach to programming. *IEEE Transactions on software Engineering*, 31(3):226–237, 2005.
36. M. Jazayeri. Some trends in web application development. In *Future of Software Engineering (FOSE'07)*, pages 199–213. IEEE, 2007.
37. C. Kaner. Improving the maintainability of automated test suites. In *International Software Quality Week*, 1997.
38. M. Khatibsyarbini, M. A. Isa, D. N. Jawawi, and R. Tumeng. Test case prioritization approaches in regression testing: A systematic literature review. *Information and Software Technology*, 93:74–93, 2018.
39. H. K. Leung and L. White. Insights into regression testing (software testing). In *Proceedings. Conference on Software Maintenance-1989*, pages 60–69. IEEE, 1989.
40. Q. Luo, F. Hariri, L. Eloussi, and D. Marinov. An empirical analysis of flaky tests. In *Proceedings of the 22nd ACM SIGSOFT international symposium on foundations of software engineering*, pages 643–653, 2014.
41. L. Madeyski. The impact of test-first programming on branch coverage and mutation score indicator of unit tests: An experiment. *Information and Software Technology*, 52(2):169–184, 2010.
42. M. Meyer. Continuous integration and its tools. *IEEE software*, 31(3):14–16, 2014.

43. R. Mukherjee and K. S. Patnaik. A survey on different approaches for software test case prioritization. *Journal of King Saud University-Computer and Information Sciences*, 33(9):1041–1054, 2021.
44. R. Newman, V. Chang, R. J. Walters, and G. B. Wills. Web 2.0—the past and the future. *International Journal of Information Management*, 36(4):591–598, 2016.
45. R. Pan, M. Bagherzadeh, T. A. Ghaleb, and L. Briand. Test case selection and prioritization using machine learning: a systematic literature review. *Empirical Software Engineering*, 27(2):29, 2022.
46. B. Papis, K. Grochowski, K. Subzda, and K. Sijko. Experimental evaluation of test-driven development with interns working on a real industrial project. *IEEE Transactions on Software Engineering*, 48(5):1644–1664, 2020.
47. K. Petersen, J. Carlson, E. Papatheocharous, and K. Wnuk. Context checklist for industrial software engineering research and practice. *Computer Standards & Interfaces*, 78:103541, 2021.
48. J. Plekhanova. Evaluating web development frameworks: Django, ruby on rails and cakephp. *Institute for Business and Information Technology*, 20:2009, 2009.
49. G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold. Test case prioritization: An empirical study. In *Proceedings IEEE International Conference on Software Maintenance-1999 (ICSM'99).'Software Maintenance for Business Change'(Cat. No. 99CB36360)*, pages 179–188. IEEE, 1999.
50. T. Sharma, P. Mishra, and R. Tiwari. Designite: A software design quality assessment tool. In *Proceedings of the 1st International Workshop on Bringing Architectural Design Thinking into Developers' Daily Activities*, pages 1–4, 2016.
51. Y. Singh, A. Kaur, B. Suri, and S. Singhal. Systematic literature review on regression test prioritization techniques. *Informatica*, 36(4), 2012.
52. D. Spinellis. Git. *IEEE software*, 29(3):100–101, 2012.
53. M. Tyagi, M. Sharma, and P. Sharma. The future of the web. In *Proceedings of the International Conference on Innovative Computing & Communications (ICICC)*, 2020.
54. W. E. Wong, J. R. Horgan, S. London, and H. Agrawal. A study of effective regression testing in practice. In *PROCEEDINGS The Eighth International Symposium On Software Reliability Engineering*, pages 264–274. IEEE, 1997.
55. S. S. Yau and J. J.-P. Tsai. A survey of software design techniques. *IEEE Transactions on Software Engineering*, SE-12(6):713–721, 1986.
56. S. Yoo and M. Harman. Regression testing minimization, selection and prioritization: a survey. *Software testing, verification and reliability*, 22(2):67–120, 2012.
57. A. Zarrad. A systematic review on regression testing for web-based applications. *J. Softw.*, 10(8):971–990, 2015.
58. J. Zarrin, H. Wen Phang, L. Babu Saheer, and B. Zarrin. Blockchain for decentralization of internet: prospects, trends, and challenges. *Cluster Computing*, 24(4):2841–2866, 2021.