JOHANNES KEPLER
UNIVERSITÄT LINZ

Eingereicht von

Mag. Martin J. Schütz

Angefertigt am

Institut für Systemsoftware

Beurteiler / Beurteilerin

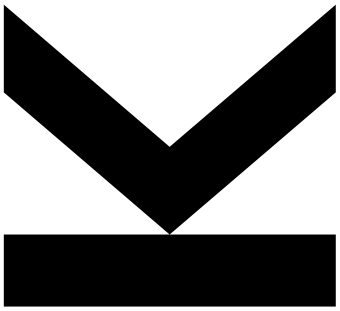
a. Univ.-Prof. Dipl.-Ing. Dr. Herbert Prähofer

Mitbetreuung

a. Univ.-Prof. Mag. Dr. Reinhold Plösch

März 2023

VORHERSAGE VON FEHLERWIRKUNGEN AUF BASIS STATISCHER ANALYSE



Masterarbeit

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Masterstudium

Computer Science

**JOHANNES KEPLER
UNIVERSITÄT LINZ**

Altenberger Straße 69
4040 Linz, Österreich
jku.at

JOHANNES KEPLER
UNIVERSITÄT LINZ

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Masterarbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt bzw. die wörtlich oder sinngemäß entnommenen Stellen als solche kenntlich gemacht habe.

Linz, März 2023

Danksagung

Ich möchte mich bei Herbert Prähofer und Reinhold Plösch für die kompetente und professionelle Betreuung dieser Arbeit bedanken. Es war für mich persönlich sehr wertvoll, in der Bearbeitung der Themenstellung einen engen Bezug zur Praxis herstellen zu können. Ganz besonders möchte ich Reinhold Plösch für die inspirierenden Diskussionen zur Themenfindung und die regelmäßig zeitlich überzogenen Abstimmungstermine während der Bearbeitung danken. Ich habe die konstruktiven Gespräche und wertvollen Denkanstöße sehr geschätzt.

Vielen Dank an Kurt Poxleitner und Hannes Scherb für die Möglichkeit, die praktische Ausarbeitung als Fallstudie in der Raiffeisen Software GmbH umzusetzen und dabei sowohl die technischen Werkzeuge als auch die notwendigen Daten produktiver Systeme nutzen zu können.

Ein besonderer Dank gilt meiner Familie, welche mich während des Studiums gestärkt und mir den notwendigen zeitlichen Freiraum eingeräumt hat. Allen voran meiner Frau Silke, welche mich zu dieser Arbeit ermutigt, motiviert und unterstützt hat.

Kurzfassung

Fehler sind in der Entwicklung von komplexen Software-Systemen unvermeidbar. Entwickler*innen sind daher unausweichlich mit Fehlern konfrontiert, deren Korrekturen zukünftige Ressourcen binden und daher in weiterer Folge notwendige Fehlerbehebungen mit Implementierungen neuer Funktionen konkurrieren. Zudem steigen die Korrekturkosten umso mehr, je später die Fehler im Projektfortschritt entdeckt werden. Um diesen Umstand entsprechend zu adressieren, wird die Software-Qualitätssicherung frühzeitig in den Software-Entwicklungsprozess eingebunden. Hierfür wird vor allem der Einsatz von statischer Analyse empfohlen, welche bereits in sehr frühen Konstruktionsphasen eingesetzt werden kann. Deren Ergebnisberichte sind jedoch typischerweise sehr umfangreich, zeigen viele Regelverletzungen auf und unterstützen durch mangelnde Priorisierung Entwicklungsteams nicht in geeignetem Ausmaß. Demzufolge wäre es von unmittelbarem Vorteil, wenn diejenigen Hinweise statischer Analyse frühzeitig identifiziert werden, welche in weiterer Folge zu Fehlerwirkungen bei der Anwendung der Software-Systeme führen, um resultierende Folgekosten für Korrekturen zu reduzieren.

Gegenstand dieser Arbeit ist es daher, Fehlerberichte von Releases und Hinweise statischer Analyse gegenüberzustellen. Auf Basis dieser Korrelationen werden geeignete Methoden der künstlichen Intelligenz eingesetzt, um praxistaugliche Modelle für die Anwendung im produktiven Umfeld zu trainieren und die Vorhersage von zukünftigen Fehlerwirkungen effektiv zu ermöglichen.

Abstract

Making errors during software development is unavoidable. Developers inevitably make errors that take additional time to fix later on. As a consequence efforts for bugfixing compete with implementing new features. Typically the later bugs are found, the higher the cost for remediation. To address this concern, software testing should start as early as possible in software development lifecycle. For this purpose, static analysis is proposed, but typically shows too many findings and hence do not support development teams appropriately. So it would be a benefit to premature detect those findings in static analysis that will result in failures to reduce subsequent efforts notably.

The purpose of the thesis is to analyse failure data from issue tracking systems that are correlated to findings from static analysis. Thereupon an artificial intelligence-based approach is used to train practicable models for business environment that enables effective prediction of software faults.

Inhaltsverzeichnis

1. Einleitung 1
1.1. Motivation 1
1.2. Rahmenbedingungen 2
1.3. Methodischer Ansatz 4
1.4. Struktur der Arbeit 5
2. Grundlagen 6
2.1. Vorhersage von Fehlerwirkungen auf Basis statischer Analyse 6
2.2. Prozess zur Entwicklung von Vorhersage-Modellen 7
2.3. Statische Analyse 9
2.4. Metriken in der Software-Entwicklung 12
2.4.1. Produkt-Metriken 13
2.4.2. Prozess-Metriken 14
2.5. Maschinelles Lernen 15
2.5.1. Überwachtes Lernen 16
2.5.2. Unüberwachtes Lernen 18
2.5.3. Validierung 18
3. Vorhersage-Modelle in der Literatur 20
4. Fallstudie 24
4.1. Aufbau der Fallstudie 24
4.2. Analyse von Fehlerkorrekturen 27
4.3. Berechnung von Metriken 35
4.4. Training von Modellen 42
4.5. Diskussion der Ergebnisse 48
5. Zusammenfassung und Ausblick 51
6. Abbildungsverzeichnis 55
7. Literaturverzeichnis 57
8. Anhangsverzeichnis 60

1. Einleitung

In diesem Kapitel werden zu Beginn die Motivation sowie die Zielsetzung dieser Arbeit vorgestellt. Weiters wird der Rahmen, welcher durch die Umsetzung als Fallstudie in einem Softwarehaus im Finanzdienstleistungsbereich vorgegeben ist, erläutert. Anschließend werden der methodische Ansatz sowie die weitere Struktur dieser Arbeit aufgezeigt.

> *„Im Gegensatz zu physikalischen Systemen entstehen Fehler in einem Softwaresystem nicht durch Alterung oder Verschleiß. Jeder Fehler oder Mangel ist seit dem Zeitpunkt der Entwicklung in der Software vorhanden. Er kommt jedoch erst bei der Ausführung der Software zum Tragen“* [4, p. 7]

1.1. Motivation

Die Software-Qualitätssicherung steht als Teil des Software-Entwicklungsprozesses im Zwiespalt einerseits die geforderten, qualitativen Anforderungen an Software-Produkte zu gewährleisten, andererseits aber auch kurze, durchaus geschäftskritische Durchlaufzeiten zur Markteinführung von Neuerungen zu ermöglichen. Technische Schuld ist dabei eine Möglichkeit diese Durchlaufzeiten zu beschleunigen, indem bestimmte Defizite im Quellcode vorübergehend geduldet werden [1]. Bestimmte Defizite im Quellcode können somit aus strategischen Überlegungen beabsichtigt sein, andererseits aber auch als Folge von Fehlerhandlungen im Zuge der Software-Entwicklung unbeabsichtigt erfolgen [2]. Daher ist es wichtig, technische Schulden im Software-Projekt aktiv zu erkennen, zu managen und deren Behebung entsprechend zu priorisieren, um zukünftige Aufwände von Entwicklungsressourcen zu optimieren [3]. Vor allem, wenn auf Grund von technischen Schulden Fehlerzustände im Software-System beim Kunden hervorgerufen werden, können vormals positiv eingeplante Aufwandseffekte in zeitkritischen Entwicklungsphasen nicht realisiert werden, da Korrekturaufwände und resultierende -kosten für Fehler mit steigender Konstruktionsphase zunehmen [4, p. 184f].

Hierfür bietet sich vor allem die statische Analyse als Werkzeug der Software-Qualitätssicherung an. Ohne lauffähige Software-Produkte vorauszusetzen, kann bereits in frühen Konstruktionsphasen Quellcode analysiert werden, um potenzielle Fehler im Quellcode zu finden [5, p. 115ff]. Da bei der statischen Analyse jedoch kein Code bzw. Programm ausgeführt wird, können daher auch keine konkreten Fehlerwirkungen

gefunden werden. Vielmehr werden meist qualitative Schwachstellen (c*ode smells*) aufgezeigt, welche kategorisiert und anschließend bewertet werden müssen. Die Ausführung der statischen Analyse selbst ist dabei automatisiert und kosteneffektiv möglich. Die Menge an gefundenen Schwachstellen, welche im Anschluss bewertet werden müssen, sind jedoch typischerweise sehr zahlreich. Eine Behebung aller Schwachstellen ist daher nicht realistisch. Darüber hinaus wird eine effektive Priorisierung der Schwachstellen durch zum Teil hohe Raten an Falschmeldungen (*false positives*) [6] als auch dem Umstand, dass nicht alle zukünftigen Fehler durch qualitative Schwachstellen abgebildet werden [7], erschwert.

Daher ist es von unmittelbarem Vorteil, jene qualitativen Schwachstellen aus der statischen Analyse zu erkennen und hervorzuheben, welche fehlerhaften Quellcode identifizieren und in weiterer Folge zu Fehlerwirkungen beim Kunden führen.

Das Ziel dieser Arbeit ist es, Fehlerberichte von bereits eingesetzten Software-Produkten und gefundene Schwachstellen aus statischer Analyse gegenüberzustellen und Hinweise auf eine Korrelation dieser Datenbestände zu suchen. In Folge soll auf Basis der gefundenen Zusammenhänge und weiteren Software-Metriken ein Modell zur Vorhersage von zukünftigen Fehlerwirkungen trainiert werden, welches wiederum auf Auswertungen statischer Analysen unbekannter Software-Projekte angewendet werden kann.

1.2. Rahmenbedingungen

Diese Arbeit wird als Fallstudie in der Raiffeisen Software GmbH umgesetzt. Die Raiffeisen Software GmbH ist das Softwarehaus der Raiffeisen Bankengruppe mit 980 Mitarbeiter*innen und ca. 400 Entwickler*innen. Mit mehr als 50 Jahren Erfahrung im Finanzdienstleistungsbereich werden aktuell Software-Produkte für ca. 2,8 Mio. Raiffeisen-Kund*innen entwickelt [8].

Die Software-Qualitätssicherung nimmt dabei einen festen Stellenwert im Unternehmensmodell ein und ist in den internen Prozessen integriert. Um positive Effekte aus einer frühen Qualitätssicherung zu erzielen, wird ergänzend zu diversen Testmethoden und -stufen auch die statische Analyse eingesetzt. Dabei wird neben Erweiterungen der Entwicklungswerkzeuge zur zeitnahen Überprüfung der Qualität am Entwicklerarbeitsplatz zusätzlich SonarQube [9] als unternehmensweit einheitliche Plattform für die statische Analyse verwendet. Zur Auslieferung von Software-Produkten werden sowohl konservative Release-Methoden eingesetzt und dabei neue Versionen nach Wasserfall-Methodik entwickelt als auch agile Release-Methoden mit weitaus kürzeren und häufigeren Release-Zyklen. In beiden Varianten werden die eingesetzten Programmiersprachen für Komponenten der serviceorientierten Architektur durch

SonarQube unterstützt und die Analysen voll automatisiert in die Entwicklungsprozesse integriert.

Statische Analysen werden anlassbezogen, aber auch kontinuierlich bzw. im Zuge der Erstellung von Releases angestoßen. Der Qualitätslevel von Software-Produkten ist hoch und in der Regel liefert die statische Analyse keine unmittelbaren Hinweise, welche kritische Fehlerzustände augenscheinlich identifizieren. Vielmehr werden auf Basis der eingesetzten Prüfkataloge eine Vielzahl von qualitativen Schwachstellen identifiziert. Die Analyse dieser Auswertungen sowie die Bewertung und Priorisierung der aufgezeigten Schwachstellen obliegen den Entwicklungsteams.

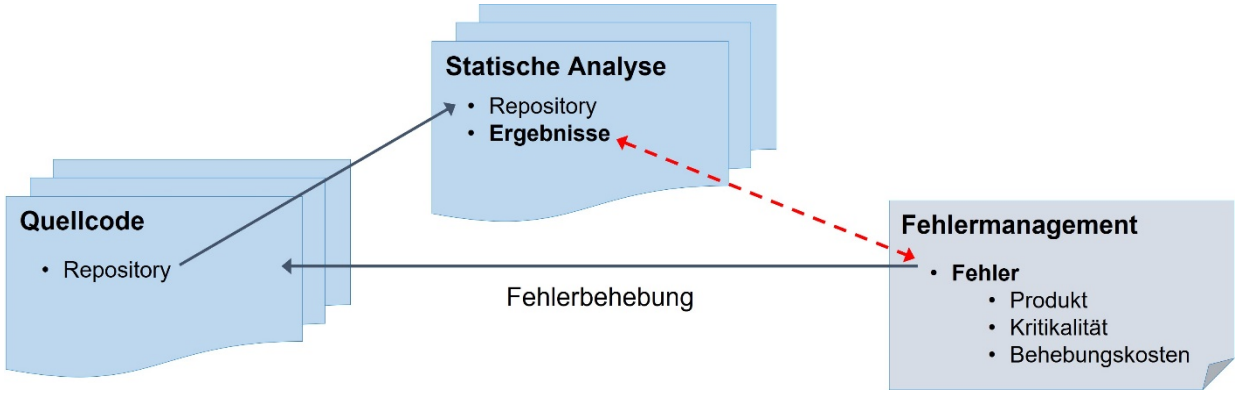


Abbildung 1 Ökosystem

Um die notwendigen Datenbestände für Analysen im Zuge dieser Arbeit zu erhalten ist der Zugriff auf unterschiedliche Werkzeuge notwendig. Die relevanten Systeme mit entsprechender Datenhaltung sind in Abbildung 1 Ökosystem abgebildet. Der Quellcode wird auf Basis von Repositories im Code-Managementsystem Git verwaltet [10]. Repositories bieten hierfür die Möglichkeit zur Versionskontrolle und unterstützen gleichsam das Arbeiten in verteilten Entwicklungsteams als auch die gleichzeitige Arbeit an unterschiedlichen Versionsständen im Software-Projekt. Ein Repository verwaltet dabei den Quellcode einer Komponente, welche im Betrieb ein eingeständiges Service als Teil eines Software-Produktes widerspiegelt. Im Zuge des Entwicklungsprozesses werden auf Ebene der einzelnen Repositories statische Analysen angestoßen. Die Ergebnisse der gefundenen Schwachstellen werden anschließend in SonarQube verwaltet. Dabei werden Trends für Art und Umfang der gefundenen Schwachstellen aufgezeigt, Detaildaten sind jedoch ausschließlich für neu durchgeführte Analysen vorgesehen und stehen historisch nicht zur Verfügung. Das Fehlermanagement erfolgt in Jira [11]. Im Zuge der Erfassung und Verwaltung von Fehlern werden neben Daten der Kritikalität und internen Behebungskosten auch eine Vielzahl weiterer Attribute erfasst, welche die Zuordnung zu Software-Produkten bzw. Repositories ermöglichen. Auf Basis von Kommentaren im Code-Managementsystem Git können Änderungen am Quellcode, welche im Zuge von Fehlerbehebungen erfolgen, direkt dem auslösenden Fehler im

Fehlermanagement zugewiesen werden. Eine Verknüpfung von historischen Ergebnissen der statischen Analyse zum Fehlermanagement und umgekehrt gibt es nicht.

1.3. Methodischer Ansatz

Der Entwicklungs- sowie Auslieferprozess von Komponenten ist hoch automatisiert. Im Zuge der kontinuierlichen Integration werden Änderungen am Quellcode im Code-Managementsystem verwaltet, regelmäßig gebaut sowie zeitnah getestet. Aufgabe der kontinuierlichen Auslieferung ist es, diese Änderungen über mehrere nicht-produktive Umgebungen mit unterschiedlichem Testfokus und Qualitätsanspruch bis in die produktiven Systeme zu heben und schließlich für die Nutzung durch den Kunden freizugeben (r*ollout*) [12].

In dieser Prozesskette können präzise Zuordnungen von „Fehlern“ zu den Prozessschritten getroffen werden: Eine Fehlhandlung (*error*) erfolgt bewusst oder unbewusst durch eine Person im Zuge der Entwicklungstätigkeiten. Diese Fehlhandlung führt unmittelbar zu einem Fehlerzustand (*defect oder bug*). Dieser Fehlerzustand kann, muss aber nicht, zu einer Fehlerwirkung (*failure*) in dem Software-Produkt führen, welche durch eine Abweichung vom erwarteten Sollverhalten bemerkbar ist [13, p. 4f].

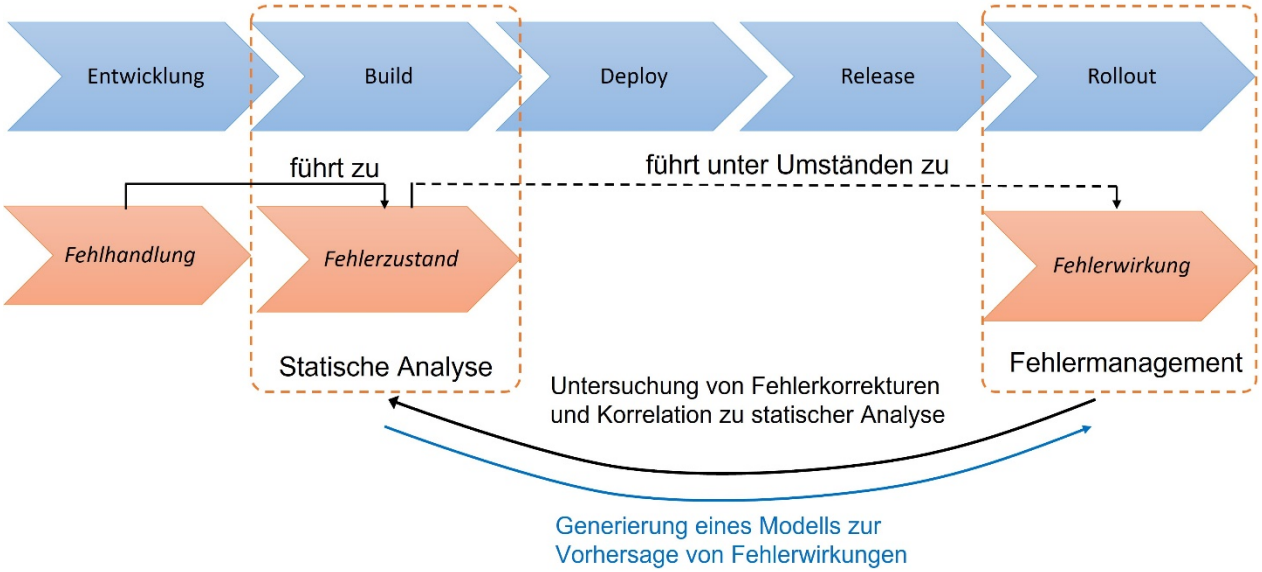


Abbildung 2 Methodik

Als Grundlagen für diese Fallstudie werden speziell eingerichtete Berichte im Fehlermanagement zu Fehlerwirkungen ausgewählter Software-Produkte über einen Zeitraum von einem Jahr überwacht. Führen Korrekturen dieser Fehler zu Änderungen am Quellcode, dann wird das betroffene Repository in den fehlerhaften Ausgangszustand versetzt und eine statische Analyse auf diesen Versionsstand ausgeführt. Die erhaltenen

Hinweise der statischen Analyse werden mit den Änderungen am Quellcode abgeglichen. Deutet ein Hinweis auf eine im Zuge der Fehlerbehebung geänderten Stelle im Quellcode hin, unabhängig davon, ob der Hinweis in der statischen Analyse durch die Änderung selbst behoben wurde oder nicht, werden Detaildaten zum Ausgangszustand des Repositories sowie die Daten betreffend den Hinweis der statischen Analyse aufgezeichnet (siehe Abbildung 2 Methodik). Durch diesen Ansatz der qualitativen Rückbetrachtung soll in einem ersten Schritt eine zuverlässige Datenbasis aufgebaut werden. Diese Datenmenge beinhaltet alle aufgetretenen sowie durch Anpassung am Quellcode behobenen Fehlerwirkungen, welche im fehlerhaften Ausgangszustand vor der Auslieferung der Software einen oder mehrere Hinweise auf qualitative Schwachstellen in den automatisierten Berichten enthielten. Zusätzlich werden auch alle Änderungen am Quellcode dokumentiert, welche ebenfalls im Zuge von Korrekturen erfolgt sind, aber keine Hinweise auf qualitative Schwachstellen in der Ausgangsversion aufzeigten.

Diese zusammenhängenden Datensätze von Fehlerwirkungen und qualitativen Schwachstellen dienen in einem nächsten Schritt dazu ein Modell zu trainieren. Ziel dieses Modells ist es zu einem frühen Zeitpunkt der Prozesskette Hinweise der statischen Analyse als Auslöser für die Vorhersage zukünftiger Fehlerwirkungen zu nutzen um diese zielgerichtet zu beheben.

1.4. Struktur der Arbeit

In Kapitel 2 werden die technischen Grundlagen für diese Arbeit beschrieben. Nach einer Konkretisierung der Zielsetzung der Vorhersage von Fehlerwirkungen werden mögliche Prozessmodelle und Vorgehensweisen vorgestellt. Anschließend werden die Möglichkeiten, aber auch Grenzen der statischen Analyse beschrieben und eine Gliederung von Metriken in der Software-Entwicklung aufgezeigt. In weiterer Folge wird eine Klassifikation für die Ansätze des maschinellen Lernens beschrieben. In Kapitel 3 werden aktuelle Ansätze aus der Literatur vorgestellt, welche unterschiedliche Vorgehensweisen des maschinellen Lernens zur Vorhersage von Fehlerwirkungen anwenden und deren Ergebnisse aufgezeigt. Weiters wird in Kapitel 4 im Detail auf die Fallstudie eingegangen. Es werden die drei Stufen der Fallstudie beschrieben und die Ergebnisse diskutiert. Abschließend werden in Kapitel 5 der Ablauf der Fallstudie zusammengefasst sowie ein Ausblick auf zukünftige Arbeiten gegeben.

2. Grundlagen

In diesem Kapitel wird zu Beginn die Vorhersage von Fehlerwirkungen inhaltlich konkretisiert und eine Abgrenzung zur Priorisierung von Hinweisen statischer Analysen gezogen. Dann werden Prozessmodelle für die Sammlung relevanter Daten und das Training von Vorhersage-Modellen dargestellt sowie die Grundlagen der statischen Analyse, dem Einsatz von Metriken in der Software-Entwicklung und dem maschinellen Lernen beschrieben.

2.1. Vorhersage von Fehlerwirkungen auf Basis statischer Analyse

Zahlreiche Studien untersuchen den Einsatz von statischer Analyse in der Software-Entwicklung. Wie in Kapitel 1.1 beschrieben macht allein die große Menge an gefundenen Hinweisen eine lückenlose Behebung dieser ökonomisch unmöglich. Resultierend aus dieser Problemstellung werden zahlreiche Ansätze zur Priorisierung von Hinweisen der statischen Analyse entwickelt und propagiert. Einheitliches Ziel ist es, Entwicklungsteams eine Hilfestellung in der Bewertung zu geben um eine Umsetzung ausgewählter, relevanter Hinweise kosteneffektiv und ressourcenschonend zu ermöglichen. In der Regel nutzen diese Ansätze weitere Daten, vor allem Metriken der Software-Entwicklung, um die gefundenen Hinweise der statischen Analyse zu priorisieren.

Boogerd et. al. [14] nutzen die Wahrscheinlichkeit, mit welcher eine Codestelle ausgeführt wird als Maßzahl für die Kategorisierung der Schwere eines gefundenen Hinweises anstelle der Kategorisierung der statischen Analyse selbst. Kim et. al. [6] trainieren ein Modell auf Basis der Änderungshistorie des Repositories, um damit die Hinweise der statischen Analyse zu priorisieren. Dabei wird eine Gewichtung für jede Kategorie der statischen Analyse erstellt, welche auf der Anzahl der Behebungen dieser Hinweise im Zuge der Änderungen im Quellcode basiert, wobei jene Änderungen im Zuge von Fehlerbehebung stärker auf die Gewichtung einwirken als jene der Entwicklung neuer Funktionalitäten. Malhotra et. al. [15] verwenden objektorientierte Metriken wie bspw. Vererbungstiefe und Kopplung um die Menge von Klassen, auf denen die Hinweise aus statischer Analyse angewendet werden, vorab einzuschränken. Sea-Lim et. al. [16] beschreiben einen kontextsensitiven Ansatz zur Priorisierung von Hinweisen. Dabei werden Ergebnisse der statischen Analyse auf die aktuellen Entwicklungsaufgaben und in weiterer Folge auf die dadurch zu ändernden Ressourcen angewendet. Dadurch wird der Fokus von einer Priorisierung rein auf Basis der Hinweise auf eine Priorisierung entsprechend der notwendigen Wartungstätigkeiten des Quellcodes vor der Bearbeitung weiterer Entwicklungsaufgaben gelegt.

Ebenso wie bei der Priorisierung von Hinweisen aus statischer Analyse ist auch bei der Vorhersage von Fehlerwirkungen das Ziel, Software kosteneffektiv zu entwickeln und Ressourcenaufwände einzusparen. Dafür werden Modelle trainiert, welche um Kontextinformationen und Metriken zum Software-Projekt erweitert werden, um die Entscheidungen des Modells zu beeinflussen und die Vorhersage zu bestimmen. Zusätzlich werden diese Modelle in der Regel um Informationen vergangener Fehlerwirkungen und Fehlerzuständen bekannter Software-Projekte erweitert und anschließend auf neue, unbekannte Projekte angewendet [17]. Im Hinblick auf die Nutzung von Daten aus statischer Analyse sind die beiden Ansätze ähnlich, unterscheiden sich vor allem aber darin, dass bei der Vorhersage von Fehlerwirkungen aufgrund statischer Analyse die gefundenen Schwachstellen nicht in einer Reihung gebracht werden, sondern mithilfe der statischen Analyse als Auslöser zu einer binären Entscheidung in Hinblick auf die Fehleranfälligkeit einer Ressource, meist auf Klassenebene, führt. Die Schwachstellen sind somit Grundlage für eine Klassifikation, ob Fehlerwirkungen zu erwarten sind oder nicht [18].

2.2. Prozess zur Entwicklung von Vorhersage-Modellen

Für die Erhebung notwendiger Daten sowie das Training von Vorhersage-Modellen werden unterschiedliche Prozesse angewendet. Diese gruppieren notwendige Arbeitsschritte im Prozessablauf unterschiedlich bzw. setzen daher andere Schwerpunkte.

Alikhashashneh et. al. [19] beschreiben einen Prozess in zwei Phasen.

1. Zuerst wird die notwendige Datenbasis aufgebaut. Diese Daten bestehen einerseits aus Software-Metriken wie Umfang und Komplexität des Projekts aber auch automatisiert erhobenen, objektorientierten Metriken. Zusätzlich werden diese Daten um Hinweise aus statischer Analyse ergänzt. Ergebnis dieser Phase ist eine numerische Datenmenge, welche in Form einer Matrix aufgebaut ist. Die Zeilen repräsentieren die Funktionen im Software-Projekt und die Spalten der Matrix sind die erhobenen Metriken. Die letzte Spalte enthält die Klassifizierung, ob es sich bei den dokumentierten Beispielen um Daten einer konkreten Fehlerwirkung handelt oder nicht.
2. Anschließend werden in der Lernphase unterschiedliche Ansätze des maschinellen Lernens angewendet, um ein Modell zu trainieren, welches in weiterer Folge auf unbekannten Quellcode angewendet werden kann. Dazu werden zuerst die bereits vorliegenden Daten vorverarbeitet und die Datensätze normalisiert. Auf Basis von Korrelationen der Beispiel-Daten zur erfolgten

Klassifikation werden relevanten Metriken erkannt und irrelevante verworfen. Anschließend wird das Modell mit Hilfe der relevanten Metriken trainiert, um anschließend auf unbekannten Quellcode ausgeführt zu werden.

Rathore et. al. [17] zeigen in ihrer Studie über unterschiedliche Techniken zur Vorhersage von Fehlerwirkungen einen flexiblen, dreistufigen Prozess, durch welchen die in der Studie analysierten Ansätze beschrieben werden können, siehe Abbildung 3 Prozessaufbau in Anlehnung an [17]. Dieser Prozess ist nicht zwingend auf Daten statischer Analyse angewiesen, kann diese aber in Form ergänzenden Metriken abbilden und flexibel integrieren.

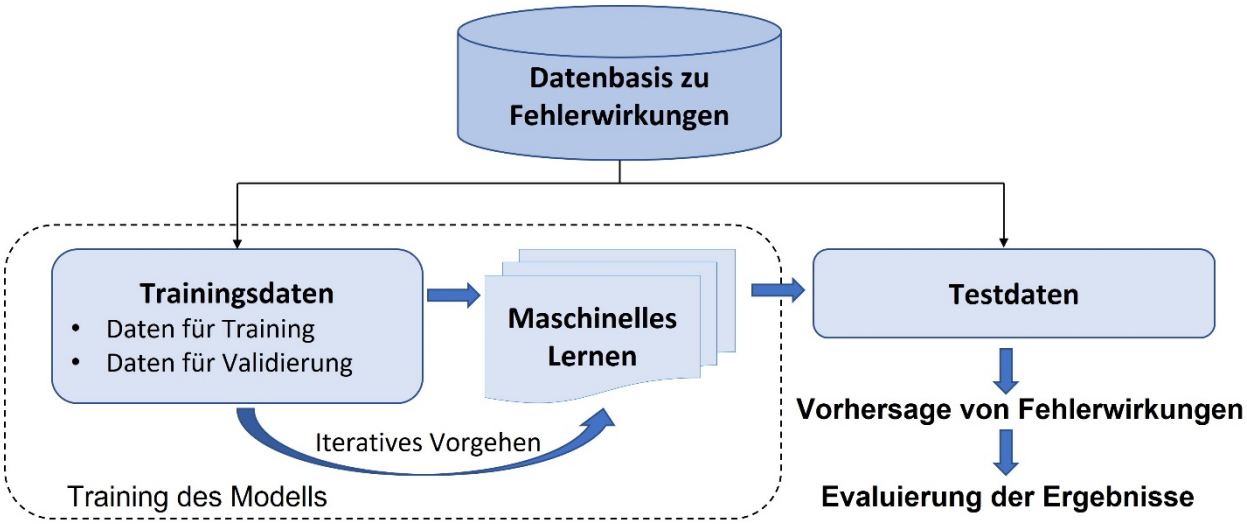


Abbildung 3 Prozessaufbau in Anlehnung an [17]

1. In der ersten Phase wird die Datenbasis erstellt. Diese kann aus Daten bzgl. Software-Metriken, historischen Informationen zu Fehlerwirkungen und Meta-Informationen zum Software-Projekt bestehend. Die Software-Metriken können dabei sowohl produktbezogene Metriken, welche die Eigenschaften eines Projektes beschreiben, als auch prozessbezogene Metriken, welche auf den Software-Entwicklungsprozess fokussieren, sein. Eine Taxonomie zu Metriken in den Software-Entwicklung wird in Kapitel 2.4 aufgezeigt. Die historischen Fehlerinformationen können im Detaillierungsgrad stark voneinander abweichen. Dieser ist abhängig von der Zugänglichkeit der Daten und unterscheidet sich in nicht einsehbare, teilweise einsehbare als auch öffentlich zugängliche Systeme. Die Meta-Informationen des Software-Projekts beschreiben in weiterer Folge unterschiedliche Charakteristika wie bspw. den Reifegrad oder die Domäne des Projekts.
2. In der zweiten Phase werden unterschiedliche Ansätze des maschinellen Lernens eingesetzt. Dabei kann aufgrund des Zusammenhanges der ausgewählten Daten für das Training des Modells zu den Daten für den Test bzw. die Vorhersage in drei

Varianten unterschieden werden. Es können Daten für Training und Test des Modells von identer Version bzw. Release des Projektes verwendet werden (*intra-release*), von unterschiedlichen Versionen von dem gleichen Projekt (*inter-release*), oder auch von anderen Projekten (*cross-project/company*).

3. In der abschließenden, dritten Prozessphase werden die Ergebnisse bewertet. Die Bewertung kann sowohl durch numerische als auch durch grafische Methoden bewertet werden, wobei die numerischen Ansätze weitere Verbreitung finden wie bspw. die Berechnung der Genauigkeit oder der Trefferquote von Vorhersagen. [17].

Der für diese Fallstudie eingesetzte Prozess basiert auf den beiden hier vorgestellten Prozessvarianten und wird in Kapitel 4.1 konkretisiert.

2.3. Statische Analyse

Statische Testmethoden sind eine Möglichkeit, um sehr früh im Software-Entwicklungsprozess die Qualität eines Software-Produktes sicherzustellen. Im Unterschied zu dynamischen Testverfahren wird die Software dabei nicht mit Testdaten versehen und ausgeführt, sondern einer Analyse unterzogen. Diese Analyse kann sowohl manuell als auch automatisiert erfolgen. Bei der manuellen Durchführung, sogenannten Reviews, wird die menschliche Analysefähigkeit genutzt. Hierzu gibt es unterschiedliche Ansätze, welche sich anhand des zugrundeliegenden Prüfobjektes, entweder des Software-Entwicklungsprozesses oder des Software-Produkts selbst, bzw. auch aufgrund des Formalismus und investierten Ressourcenaufwandes unterscheiden. Bekannte Ansätze sind Walkthrough, Technisches Review und die Inspektion. Ebenso wie bei den Reviews wird auch bei der automatisierten Ausführung, der sogenannten statischen Code-Analyse bzw. statischen Analyse, keine Software ausgeführt. Es können mehrere Analysetechniken der statischen Analyse unterschieden werden. Allen gemein ist, dass die zu untersuchenden Artefakte einem vorab bestimmten Formalismus unterliegen müssen. Die ursprüngliche Intention der manuellen als auch der automatisierten Verfahren des statischen Tests ist die Fehlerprävention [4, p. 81ff].

Da bei der statischen Analyse keine Ausführung der Software selbst erfolgt, können auch keine konkreten Fehlerwirkungen nachgewiesen werden. Vielmehr werden Stellen im Quellcode aufgezeigt, welche einem verdächtigen Fehlermuster ähneln bzw. von einem vorgegebenem Sollmuster abweichen. Diese Muster werden typischerweise in unterschiedliche Kategorien eingeteilt und können zielgerichtet zum Einsatz kommen bzw. auch deaktiviert werden. Die Anzahl der Ergebnisse hängt daher auch von Art und Umfang der aktivierten Kategorien und Prüfregeln für die statische Analyse ab. Die

gefundenen Ergebnisse können daher sehr zahlreich sein. Die Bewertung der Ergebnisse erfordert in der Regel entsprechendes Know-how um die Resultate entsprechend zu priorisieren [5, p. 117].

Im Folgenden werden die Analysetechniken Kontrollflussanalyse, Datenflussanalyse und die Einhaltung von Codierungsstandards vorgestellt. Zusätzlich werden im Zuge der statischen Analyse häufig auch vorab definierte Metriken erhoben, diese werden in Kapitel 2.4 gesondert beschrieben.

- Im Zuge der Kontrollflussanalyse wird die Struktur eines Programmes untersucht. Diese ist in der Regel durch Entscheidungsknoten und Schleifen geprägt und mit dem Ausmaß dieser Strukturen steigt sowohl die Komplexität des Quellcodes als auch die Fehleranfälligkeit. Neben der Bestimmung der Komplexität des Kontrollflusses sind das Auffinden von nicht ausführbarem bzw. „totem“ Quellcode und die Eintritts- und möglichen Austrittspunkte von Schleifen zur Identifikation von Endlosschleifen typische Ergebnisse dieser Kategorie [5, p. 118ff].
- Die Datenflussanalyse untersucht den Einsatz von Variablen im Quellcode. Von Interesse sind die Stellen sowohl der Definition als auch der Nutzung von Variablen. Typische Ergebnisse dieser Analysemethode sind das Auffinden undefinierter Variablen, von Variablen, welche zwar definiert sind, aber einen offensichtlich ungültigen Wert annehmen, oder auch Variablen die einen neuen Wert annehmen, bevor die vormals berechneten Werte gelesen und verwendet werden [5, p. 120f]
- Bei der Prüfung der Einhaltung von Codierungsstandards kommen Regelkataloge zum Einsatz, welche auf die eingesetzte Programmiersprache zugeschnitten sind aber auch auf spezielle Erfordernisse von Unternehmen oder Projekte angepasst werden können. Durch Standardisierung und die Steigerung der Wartbarkeit des Quellcodes steigt daher auch die Möglichkeit, dass unterschiedliche Entwickler*innen sich Programmiertätigkeiten aufteilen können. Quellcode ist somit auch besser übertragbar und wird in der Regel testbarer [5, p. 122f].

Für viele bekannte Programmiersprachen wird eine breite Palette an Werkzeugen für die statische Analyse angeboten. Die Regelkataloge sind typischerweise konfigurierbar und können meist auch um eigene Regeln erweitert werden. Viele dieser Werkzeuge können sowohl in der Entwicklungsumgebung als auch im Zuge der kontinuierlichen Integration in den Entwicklungs- und Auslieferungsprozess integriert werden. Ziel der Anwendung in der Entwicklungsumgebung ist es, zeitnahe Rückmeldungen im Zuge der Entwicklungstätigkeiten zu geben und frühestmöglich Qualitätsstandards zu forcieren. Ziel der Anwendung im Zuge der kontinuierlichen Integration ist eine umfassende Analyse

des Quellcodes, um vordefinierte Qualitätslevel sicherzustellen (*quality gate*) und Trends zu verfolgen.

Für Java stehen mehrere Werkzeuge inkl. eigener Regelkataloge mit unterschiedlichen Schwerpunkten zur Verfügung, bekannte Vertreter sind PMD[1], FindBugs[2] und Checkstyle[3]. Um umfassende Analysen mit unterschiedlichen Regelkatalogen effizient zu ermöglichen, werden darüber hinaus Meta-Werkzeuge, welche die Kataloge anderer Werkzeuge integrieren, angeboten. Ein weit verbreitetes Werkzeug, welches selbst in Java implementiert ist und weitere Regelkataloge externer Anbieter integrieren kann, ist SonarQube [9].

Regelkataloge können in SonarQube beliebig erweitert und verwaltet werden und finden Anwendung in einem entsprechendem Qualitätsprofil für eine bestimmte Programmiersprache. Im Zuge der statischen Analyse wird ein Qualitätsprofil auf ein Repository angewendet und bei jeder Regelverletzung wird ein Hinweis für den Bericht erstellt. Die gefundenen Hinweise werden in drei Typen unterteilt.

- Ein Hinweis vom Typ *Bug* zeigt einen Fehlerzustand, welcher zur Laufzeit zu einem Abweichen vom erwarteten Sollzustand führen kann.
- Ein Punkt vom Typ *Vulnerability* beschreibt eine sicherheitskritische Schwachstelle im Quellcode.
- Ein Hinweis vom Type *Code Smell* hebt Stellen im Quellcode hervor, welche höhere Komplexität oder Abweichungen von Codierungsrichtlinien darstellen und daher eine schlechtere Wartbarkeit aufweisen.

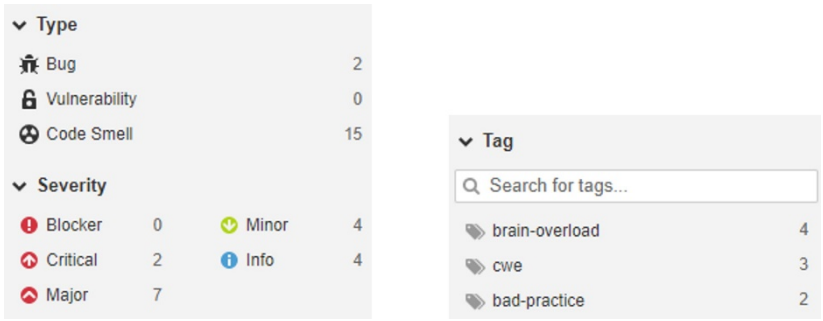


Abbildung 4 SonarQube: Typen von Hinweisen und Kategorisierung

Die drei Typen von Hinweisen werden aufgrund der Gewichtung der Verletzung in eine von fünf möglichen Schweregraden zugeordnet, welche einerseits zur automatisierten Bewertung bei der Berechnung von Qualitätslevel als auch zur Priorisierung der Hinweise

[1] https://pmd.github.io/

[2] https://findbugs.sourceforge.net/

[3] https://checkstyle.org/

durch das Entwicklungsteam verwendet werden. Die Gewichtung *Blocker* stellt den höchsten Schweregrad dar, *Info* den niedrigsten. Zusätzlich können die gefundenen Hinweise über Stichworte gegliedert werden, diese werden einerseits über die Regelkategorien mitgeliefert, können aber beliebig angepasst und erweitert werden, siehe Abbildung 4 SonarQube: Typen von Hinweisen und Kategorisierung.

2.4. Metriken in der Software-Entwicklung

Zur Überwachung und Steuerung von umfangreichen Software-Projekten ist es u.a. notwendig, kontinuierliche Messungen auf Basis des Quellcodes durchzuführen. Dadurch soll ermöglicht werden, Software-Projekte zu bewerten, Trends zu beobachten und vor allem Projekte miteinander zu vergleichen. Ziel dieser Aktivitäten ist es, die Software-Qualität aktiv zu beeinflussen und Risiken sowie Schwachstellen zu managen. Die hierfür angewendeten, in der Regel standardisierten, Kennzahlen werden als Metriken bezeichnet. Eine Metrik ist dabei "*die Eigenschaft eines Objekts deren Ausprägung mit einer geeigneten Messmethode ermittelt werden kann.*" [20, p. 430]

Um den erfolgreichen Einsatz von Metriken zu gewährleisten, müssen diese bestimmte Qualitätsmerkmale aufweisen [21, p. 248f].

- Die Objektivität einer Metrik stellt sicher, dass auf Basis von Formeln die Erhebung der Werte automatisiert, frei von subjektiven Einflüssen erfolgen kann.
- Die Robustheit garantiert, dass wiederholende Ausführungen der Berechnung jedes Mal das gleiche Ergebnis liefern und die erhobenen Werte somit belastbar sind.
- Die Vergleichbarkeit ermöglicht es, dass Messungen unterschiedlicher Datenbasen zueinander in Relation gesetzt und somit miteinander verglichen werden können. In der Folge sollen Steuerungsmechanismen ermöglicht werden.
- Die Ökonomie von Metriken zielt auf eine ressourcenschonende Berechnung der Werte ab. Im besten Fall können Metriken automatisiert erhoben werden.
- Die Korrelation einer Metrik beschreibt, wie sehr die berechneten Werte direkte Rückschlüsse auf die überwachten Daten ermöglichen.
- Die Verwertbarkeit einer Metrik beschreibt, wie sehr die erhobenen Werte das zukünftige Handeln beeinflussen. Im schlechtesten Fall werden Metriken zum Selbstzweck erhoben und es resultieren keine Steuerungsmaßnahmen aufgrund der Aussagen.

Primäres Ziel dieser Metriken ist der zielgerichtete Einsatz im gesamten Software-Entwicklungsprozess, um durch die getroffenen Entscheidungsprozesse durch die

Verringerung subjektiver Beurteilungen zu verbessern. Hierfür sollen sowohl Produkt- als auch Prozess-Metriken Einsatz finden [22].

Rathore et. al. [17] zeigen hierzu eine Taxonomie von Metriken, welche Produkt- und Prozess-Metriken beschreibt (siehe Abbildung 5 Klassifikation von Metriken, in Anlehnung an [17]). Eine Klassifizierung erfolgt dabei jedoch nicht wechselseitig exklusiv. Einzelne Metriken können sowohl Produkt- als auch Prozess-Metriken sein.

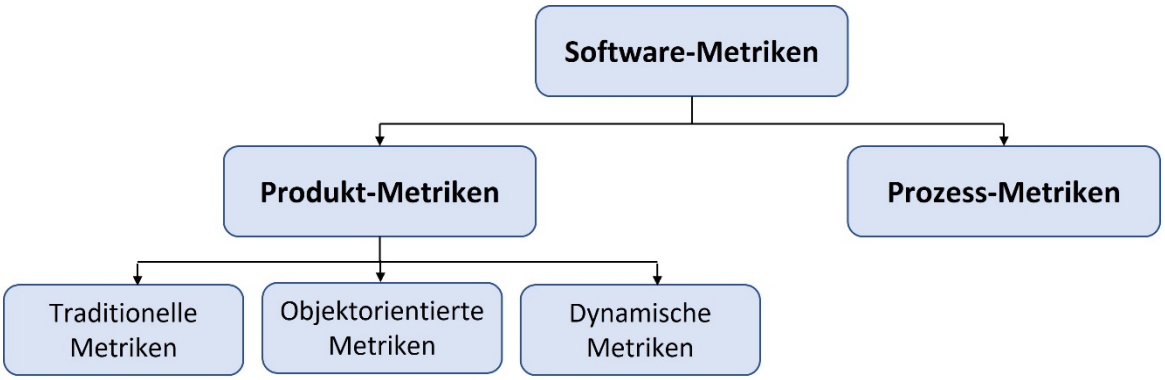


Abbildung 5 Klassifikation von Metriken, in Anlehnung an [17]

2.4.1. Produkt-Metriken

Produkt-Metriken werden auf der Basis fertiger Software-Produkte erhoben und zeigen, ob vordefinierte Anforderungen oder auch internationale Standards in der Software-Entwicklung eingehalten werden. Produkt-Metriken unterteilen sich in traditionelle, objektorientierte und dynamische Metriken [17]:

- Traditionelle Metriken geben Auskunft über den Umfang und die Komplexität von Software-Produkten. Häufig eingesetzte Metriken sind Lines of Code (*LOC*), welche die Anzahl der Zeilen Quellcode misst, als auch die zyklomatische Komplexität nach McCabe, welche zur Bestimmung der Komplexität des Kontrollflusses eingesetzt wird und dadurch eine wichtige Kennzahl in der Software-Qualitätssicherung ist.
- Objektorientierte Metriken beschreiben die Eigenschaften von Software-Projekten mit objektorientiertem Vorgehen und zielen auf die innere Struktur ab. Wichtige Vertreter dieser Kategorie sind Depth of Inheritance (*DIT*) zur Messung der Tiefe und Number of Children (*NOC*) zur Messung der Breite eines Vererbungsbaumes. Weitere Metriken sind Number of Children (*NOC*) zur Messung der direkten Vererbung zu einer Oberklasse und Coupling between Objects (*CBO*) als Maßzahl für die Beziehung zwischen zwei Klassen.

- Dynamische Metriken beschreiben das Verhalten von Software, welches zur Ausführungszeit gemessen wird. Ziel ist es die Interaktion von eindeutigen Objekten miteinander aufzuzeigen. Export Object Coupling (*EOC*) und Import Object Coupling (*IOC*) werden bspw. eingesetzt um die aus- bzw. eingehenden Nachrichten von Objekten in Relation zur Gesamtheit aller Nachrichten im Zuge eines konkreten Anwendungsfalles zu bestimmen. Somit werden häufige Kommunikationspfade aufgezeigt.

2.4.2. Prozess-Metriken

Prozess-Metriken definieren Messmethoden, welche auf den Software-Entwicklungsprozess selbst fokussieren. Der Einsatz von Metriken dieser Kategorie erfolgt, um langfristige Verbesserungsmaßnahmen zu setzen und den Entwicklungsprozess strategisch zu beeinflussen. Dadurch soll die Qualität des Software-Produktes nachhaltig positiv beeinflusst werden [17]. Dazu werden Parameter, welche Veränderungen am Software-Projekt auszeichnen, gemessen, bspw. wie viele Fehlerkorrekturen im Rahmen einer Zeitspanne oder für eine definierte Komponenten durchgeführt wurden. Ebenso kann die Anzahl unterschiedlicher Autoren, welche Änderungen an einer Komponente durchgeführt haben, aufgezeichnet werden, oder wie viele Zielen Quellcode hinzugefügt oder gelöscht wurden. Metriken können hierbei in unterschiedlichen Dimensionen betrachtet werden, bspw. wie viele Änderungen oder Reviews durch bestimmte Entwickler erfolgt sind im Unterschied zur Gesamtzahl der Änderungen.

Wichtig für den Projekterfolg ist es, eine aussagekräftige Menge von belastbaren Metriken zur Überwachung und Steuerung von Software-Projekten zu erstellen. Die Steuerung kann dabei auf Ebene der Projektmanagements oder auch der Produktentwicklung erfolgen [21, p. 248]. Sowohl Anzahl und Art von Metriken als auch die Kombination der eingesetzten Metriken kann sich unterscheiden und muss spezifisch an die Entscheidungsprozesse, welche dadurch unterstützt werden sollen, angepasst werden. Der Vorteil von weit verbreiteten Metriken liegt vor allem in der Standardisierung der Anwendung, da Messwerte im besten Fall selbsterklären sind und projektübergreifend angewendet werden können.

2.5. Maschinelles Lernen

Maschinelles Lernen ist ein Teilbereich der künstlichen Intelligenz und beschäftigt sich mit dem Lernen auf Basis von bereits vorab bekannten Daten [23, p. 177]. Diese historischen Beispiel- bzw. Lerndaten dienen zur Erzeugung statischer Modelle. Die Genauigkeit dieser Modelle nimmt in der Regel mit der zur Verfügung stehenden Menge an Lerndaten zu. Die Daten können dabei in unterschiedlichen Formen vorliegen und aus Bildern, Texten, numerischen Messwerten, akustischen Signalen usw. bestehen. Für alle Lerndaten sind vorab bekannte Zielwerte verfügbar, die in dem Modell während der Trainingsphase korreliert werden. Je Beispiel der Lerndaten gibt es bekannte Verbindung der Eigenschaften der Eingabedaten zum Zielwert. Im Zuge des Lernprozesses werden in den Datenbeständen allgemeingültige Muster gesucht, anstatt die Lerndaten auswendig zu lernen. Wie in Abbildung 6 Training von Vorhersage-Modellen, in Anlehnung an [23] dargestellt, soll es durch diese Vorgehensweise möglich sein, auf Basis erzeugter Modelle und neuer Eingabedaten bisher unbekannte Zielwerte vorherzusagen. [23, p. 177f]

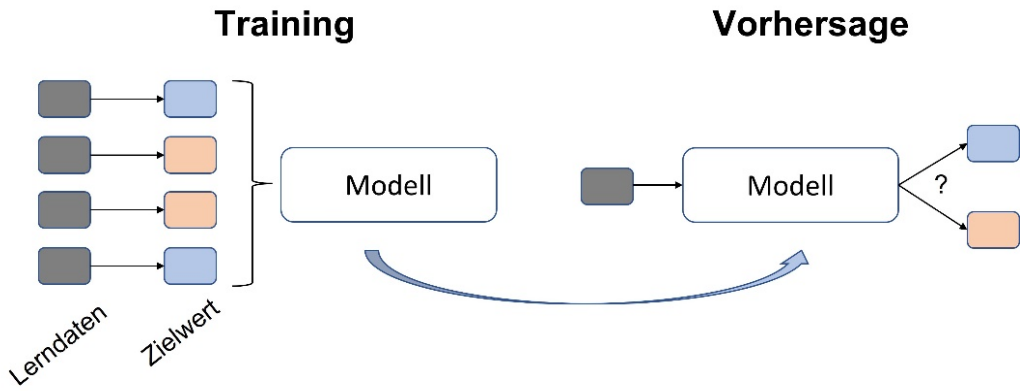


Abbildung 6 Training von Vorhersage-Modellen, in Anlehnung an [23]

Maschinelles Lernen gliedert sich wie in Abbildung 7 Klassifikation von maschinellem Lernen, in Anlehnung an [24] dargestellt in zwei Kategorien: Beim überwachten Lernen werden Modelle auf Basis bekannter Lerndaten und Zielwerte trainiert, um für zukünftige Eingabedaten eine Vorhersage zu treffen. Hier finden die Klassifikation und die Regression Anwendung. Ziel beim unüberwachten Lernen ist es neue, bisher unbekannte Muster in den Lerndaten zu erkennen. Hier kommt vor allem die Clusteranalyse zur Anwendung [24].

2.5.1. Überwachtes Lernen

Lerndaten können aus unterschiedlich vielen Merkmalsausprägungen bestehen, welche in Summe zu einem einzelnen Zielwert korrelieren. Die Menge an zusammengehörigen Merkmalen wird als Lernvektor bezeichnet. Durch eine Vielzahl von strukturell gleichen Beispielen wird beim überwachten Lernen versucht eine Assoziation vom Lernvektor zum Zielwert zu erreichen.

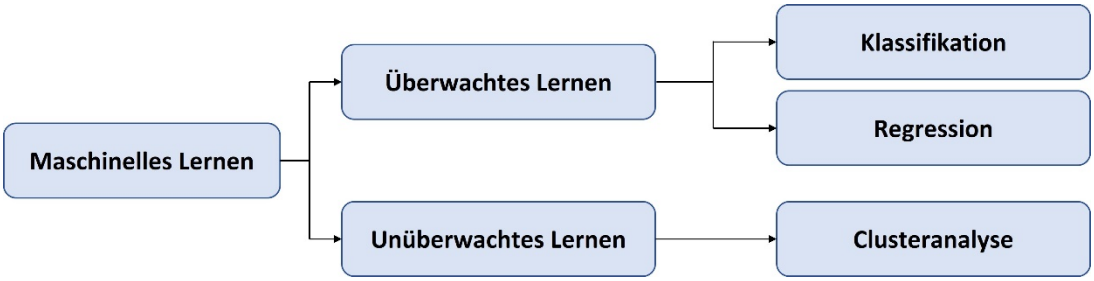


Abbildung 7 Klassifikation von maschinellem Lernen, in Anlehnung an [24]

Ist es Zielsetzung, dass aus einer Menge von vorab bekannten Kategorien eine konkrete Kategorie vorhergesagt werden soll, kommt die Klassifikation zum Einsatz. Je nach Aufgabenstellung sowie Art und Umfang verfügbarer Lerndaten können unterschiedliche Klassifikationsmodelle angewendet werden, siehe Abbildung 8 Klassifikationsmodelle [23, p. 182ff]. Der K-Nearest-Neighbor-Klassifikator stellt die Lerndaten über ihre Koordinaten in einem Merkmalsraum dar und speichert zu diesen Punkten die bekannten Zielwerte. Wird das trainierte Modell zur Vorhersage angewendet, werden die Koordinaten für die neuen Eingabedaten berechnet und die Zielwerte der nächstgelegenen, vorab bekannten Daten ermittelt. Die Anzahl der zu ermittelnden Nachbarpunkte (k) kann variieren und die Mehrheit derer Zielwerte ergeben den Vorhersagewert für die aktuelle Eingabe. Dieses Klassifikationsmodell ist sehr einfach anzuwenden, kann aber bei einer hohen Anzahl von Daten im Modell rechenintensiv und zeitaufwändig für Vorhersagen werden. Einen ähnlichen Ansatz verwenden Support-Vektor-Maschinen. Um jedoch nicht alle Koordination historischer Daten im Modell kennen zu müssen werden während der Trainingsphase jene Eingabedaten bzw. deren Lernvektoren bestimmt, welche die Grenzen zwischen den Klassen der Zielwerte aufzeigen, die sogenannten Support-Vektoren. Anhand der Koordinaten dieser Support-Vektoren und dem Kernel, einer Funktion, welche das Ausmaß des Einflussbereiches bestimmt, können zukünftige Vorhersagen mit einem kleineren Modell effektiv bestimmt werden. Vor allem die Auswahl der richtigen Daten im Lernvektor hat eine große Auswirkung auf die Genauigkeit dieses Ansatzes. Im Unterschied zu den koordinatenbasierten Klassifikationsmodellen nutzen Entscheidungsbäume einen anderen Ansatz. Während des Trainings werden die einzelnen Eigenschaftswerte der Lernvektoren separat behandelt. Das Modell versucht für jede Ausprägung einen Schwellwert zu definieren, der die weitere Verzweigung im

Baum bestimmt. Die Verzweigungen entsprechen somit den Merkmalen der Lerndaten und am Ende resultieren die Durchläufe in den Blattknoten, welche einen zugeordneten Zielwert enthalten. Wird ein trainiertes Modell mit neuen Eingabedaten abgefragt, wird der Baum von oben nach unten durchlaufen und die Eigenschaften des Eingabevektors bestimmen die Wege durch die Verzweigungen des Baumes. Der erreichte Blattknoten enthält die Klassifikation und ergibt damit die Vorhersage. Entscheidungsbäume können anhand großer Mengen an Lerndaten leicht übertrainiert werden. In diesem Fall werden die Lerndaten exakt abgebildet und eine gewünschte Generalisierung ist aufgrund des trainierten Modells nicht mehr möglich. Neue Eingabedaten durchlaufen in der Folge den

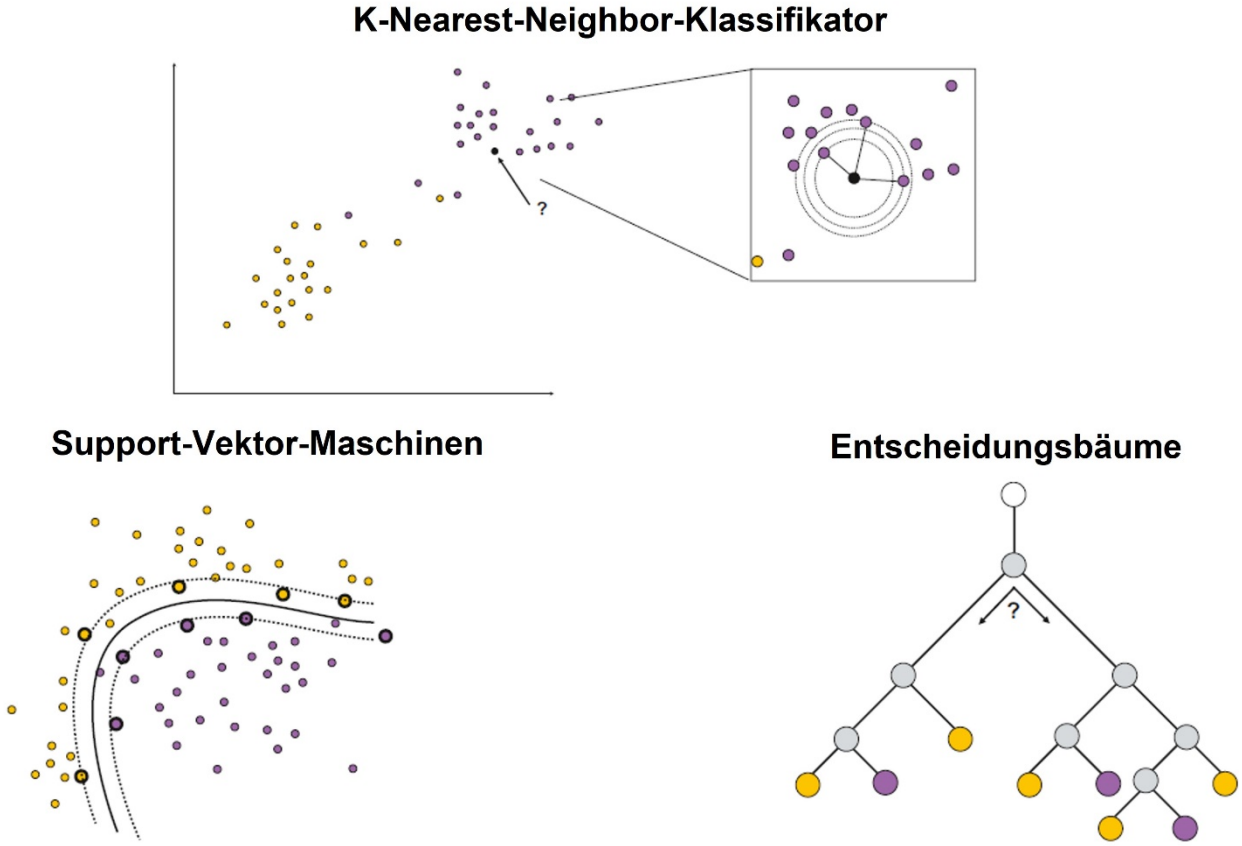


Abbildung 8 Klassifikationsmodelle [23, p. 182ff]

Baum nicht in der richtigen Abfolge bzw. ist die Qualität von getroffenen Vorhersagen nicht optimal. Damit Entscheidungsbäume nicht übertrainiert werden, nutzt man die zielgerichtete Reduktion von Verzweigungen – die max. Höhe als auch die max. Breite im Baum - in der Trainingsphase. [23, p. 182f].

Im Unterschied zur Klassifikation liegt bei der Regression keine vorab definierte Kategorisierung vor. Die Zielwerte bzw. Vorhersage-Ergebnisse sind keine konkreten Klassen bzw. bekannte Kategorien, sondern kontinuierlicher Werte.

2.5.2. Unüberwachtes Lernen

Unüberwachtes Lernen nutzt die Eingabedaten, um nach bisher unbekannten Mustern zu suchen. Im Unterschied zum überwachten Lernen sind die Zielwerte vorab nicht gekennzeichnet. Ziel ist es, Abhängigkeiten in den Strukturen und Ausprägungen der Daten zu finden, welche in Folge weitere Schlussfolgerungen zur Kategorisierung der Daten liefern und auf Basis dieses Modells Vorhersagen ermöglichen. Die Clusteranalyse ist hierfür eine weit verbreitete, explorative Methode, um bisher unbekannte Gruppierungen in den Eingabedaten zu finden. Neue Gruppierungen von ähnlichen Daten können erzeugt werden und in weitere Folge wiederum als Zielwerte für Klassifikationsmodelle eingesetzt werden [24].

2.5.3. Validierung

Um die Qualität der Vorhersage von Modellen zu messen, werden unterschiedliche Maßzahlen eingesetzt. Diese Maßzahlen beschreiben typsicherweise die Genauigkeit oder die Trefferquote von getroffenen Vorhersagen. Ein Modell zur Vorhersage, ob ein Code-Abschnitt zu einem Fehlerzustand führen wird, agiert im binären Raum. Die Klassifikation kann nur einen von vorab bekannten zwei Werten annehmen: Der Code-Abschnitt führt zu einer Fehlerwirkung oder zu keiner Fehlerwirkung. Für derartige Modelle der binären Klassifikation wird in der Regel die 2x2 Wahrheitsmatrix als Basis von Maßzahlen eingesetzt [25] [26].

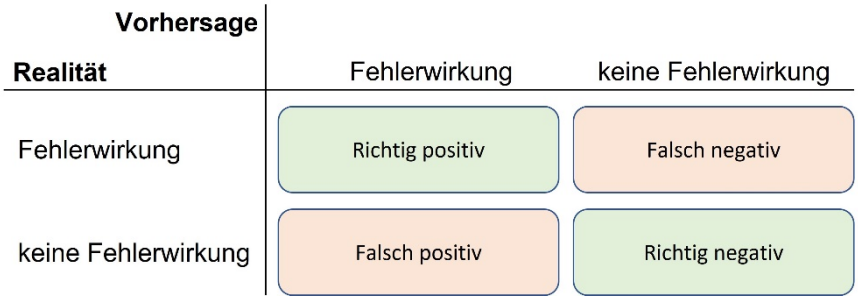


Abbildung 9 2x2 Wahrheitsmatrix, in Anlehnung an [25]

Die Wahrheitsmatrix stellt alle möglichen Ergebnisse zusammen und fasst diese in den zwei Kategorien der richtigen und der falschen Aussagen zusammen. Bei einer richtig positiven Vorhersage wird im Modell eine in der Realität eintretende Fehlerwirkung vorhergesagt. Bei einer richtig negativen Aussage wird im Modell korrekt vorhergesagt, dass keine Fehlerwirkung in der Realität auftreten wird. Diese beiden Ergebnistypen ergeben die Kategorie der richtigen Vorhersagen. Die Kategorie der beiden falschen Vorhersagen wird sowohl durch die Aussage einer Fehlerwirkung, welche in Folge nicht eintritt bzw. der Aussage, dass keine Fehlerwirkung eintreten wird, diese aber in Folge in

der Realität beobachtet werden kann, gebildet (siehe Abbildung 9 2x2 Wahrheitsmatrix, in Anlehnung an [25].

Die Accuracy berechnet die Genauigkeit der Vorhersage. Hierbei werden alle richtigen Ergebnisse, also die Summe der richtig positiven und richtig negativen, der Summe aller Vorhersagen gegenübergestellt. Wenn das Verhältnis der Lerndaten im Modell nicht ausgewogen ist und die Anzahl der Beispiele, die zu einem bestimmten Zielwert führen deutlich höher ist als die Anzahl der Beispiele des anderen Zielwertes, findet die Accuracy aufgrund fehlender Aussagekraft keine Anwendung. In diesem Fall werden in der Regel zwei weitere Maßzahlen, welche sich ebenfalls aus der Wahrheitsmatrix ableiten, eingesetzt. Die Precision beschreibt die Korrektheit der als positiv vorhergesagten Werte, in diesem Fall die prognostizierten Fehlerwirkungen. Die Precision berechnet sich aus der Anzahl der richtig positiven Vorhersagen, welche der Summe aus den richtig positiven und den falsch positiven gegenübergestellt wird. Die Maßzahl beschreibt somit, wie viel der vorhergesagten Fehlerwirkungen auch richtig waren. Der Recall ist im Gegensatz dazu die Trefferquote der richtigen Ergebnisse. Diese Maßzahl berechnet sich aus der Menge der richtig positiven Vorhersagen, welche in das Verhältnis zu der Summe der richtig positiven und falsch negativen gestellt wird. Der Recall beschreibt somit wie viele Fehlwirkungen richtig aus der gesamten Anzahl der Fehlerwirkungen vorhergesagt werden. Die beiden Maßzahlen Precision und Recall sind in der Regel konträr. Je höher die Precision eines Modells, desto geringer wird der Recall, und umgekehrt. Um ein optimales Qualitätsniveau an Vorhersagen zu ermöglichen, müssen diese beiden Werte entsprechend ausbalanciert werden. Hierfür wird das harmonische Mittel dieser beiden Werte, definiert als F-measure, berechnet. Die Berechnung erfolgt auf Basis des doppelten Produkts von Precision und Recall, welche in Verhältnis zu der Summe der beiden Werte gesetzt wird und somit diese Verhältniszahlen in gleicher Weise gewichtet [25] [26].

Die Qualität der Modelle wird auf Basis der ausgewählten Maßzahlen iterativ bewertet. Hierzu wird oft das Kreuzvalidierungsverfahren (*cross-validation*) eingesetzt. Die zugrunde liegenden Lerndaten werden dabei in mehrere, gleich große Teilmengen zerlegt. In Folge wird eine Teilmenge für die Validierung ausgewählt und ein Modell mit allen anderen Teilmengen trainiert. Die vorher beschriebenen Maßzahlen werden in den einzelnen Iterationen berechnet und liefern anschließend über alle Validierungen gemittelte Werte, welche das Qualitätsniveau des Ansatzes bestimmen [26].

3. Vorhersage-Modelle in der Literatur

In diesem Kapitel werden unterschiedliche Studien zur Entwicklung von Vorhersage-Modellen dargestellt, welche aufgrund ihrer Ansätze und dokumentierten Ergebnisse direkten Einfluss auf die Modell-Generierung in dieser Fallstudie gefunden haben. In der ersten Studie wird die Vorhersage-Qualität von Support-Vektor-Maschinen im Vergleich zu acht weiteren Klassifikationsmethoden bewertet. Anschließend werden die Ergebnisse einer Studie beschrieben, in welcher großer Wert auf die Auswahl relevanter Lerndaten gelegt wird und anstatt einer binären Klassifikation der Fehleranfälligkeit die Aussagekraft von Hinweisen aus statischer Analyse bewertet wird. In der dritten Studie wird die Eignung von Prozess- vs. Produkt-Metriken für das Training von Vorhersage-Modellen diskutiert. Zusätzlich werden fiktive Korrekturkosten für nicht erkannte, fehlerhafte Software in der Modell-Generierung mitberücksichtigt. Abschließend werden die Erkenntnisse einer umfangreichen Literaturstudie zur Bewertung von binären Klassifikationsmodellen zur Vorhersage von Fehlerwirkungen gezeigt.

Elish et. al. [26] beschreiben, dass typischerweise eine hohe Anzahl von Fehlerwirkungen aus einer geringen Anzahl an fehleranfälligen Modulen resultiert. Um die Fehleranfälligkeit von Modulen zu erheben, werden in dem beschriebenen Ansatz Produkt-Metriken erhoben und diese mit historischen Daten von Fehlerwirkungen kombiniert. Die Ebene der Korrelation der Daten wird auf Basis der Module durchgeführt. Die Module stellen Funktionen in prozeduralen bzw. Methoden in objektorientieren Programmiersprachen dar. Ziel der Studie ist es, die Vorhersage-Qualität von Modellen auf Basis von Support-Vektor-Maschinen im Vergleich zu acht weiteren Klassifikationsmethoden zu bewerten. Unter den Vergleichsmethoden befinden sich u.a. der K-Nearest-Neighbor-Klassifikator und Entscheidungsbäume.

Die Studie nutzt hierzu Lern- und Validierungsdaten von vier öffentlich zugänglichen Datensystemen von Software-Projekten der NASA. Zwei Datensets enthalten Quellcode der prozeduralen Sprache C. Die beiden weiteren Datensets enthalten objektorientierten Quellcode, je eines auf Basis von C++ bzw. Java. Der Lernvektor der Beispieldaten enthält 21 Produkt-Metriken auf Modulebene, darunter bspw. die zyklomatische Komplexität nach McCabe und die Anzahl an Zeilen Quellcode. Die Zielvariable hat einen boolschen Wert, ob ein Modul fehlerhaft ist oder nicht. Zur Bewertung der Vorhersage-Qualität der Modelle werden Accuracy, Precision und Recall und die aus den beiden letzten berechnete F-measure herangezogen. Die Validierung der Vorhersagen erfolgt anhand einer Kreuzvalidierung mit zehn Teilmengen und darauf aufbauen insgesamt 100 Testläufen.

Elish et. al. [26] kommen zu dem Ergebnis, dass die Vorhersage-Qualität von ihren Modellen, welche auf Basis von Support-Vektor-Maschinen trainiert wurden, in vielen Fällen besser, aber zumindest mit gleich guten Ergebnissen wie andere Klassifikationsmodelle arbeiten. Die Accuracy für alle vier Datensets lag in einem Bereich von 84,6% bis 93,3%. Die Precision lag in einem Bereich von 84,9% bis 93,6% und der Wert für Recall zwischen 99,4% und 100%. Die Precision wurde von ein paar Modellen moderat übertroffen, aber die Werte für Recall waren bei der Variante mit Support-Vektor-Maschinen unübertroffen. Der Wert der F-measure liegt zwischen 91,6% und 96,5% und damit im oberen Drittel der verglichenen Ansätze. Die Autoren kommen zu dem Schluss, dass diese Klassifikationsmethode einen praxistauglichen Nutzen bietet. Sie begründen dies vor allem damit, dass auf Basis des hohen Wertes für Recall das Risiko von nicht entdeckten, fehleranfälligen Modulen deutlich gesenkt werden kann.

Alikhashashneh et. al. [19] wählen eine andere Vorgehensweise und klassifizieren Hinweise aus statischer Analyse in die Kategorien richitg positiv, falsch positiv und falsch negativ. Hierfür werden Daten statischer Analyse unterschiedlicher Werkzeuge extrahiert und Produkt-Metriken für den analysierten Quellcode berechnet. Die Korrelation erfolgt auf Ebene von Methoden in C++ geschriebenem Quellcode. Ziel der Studie ist es mit dieser Datenbasis Modelle zu trainieren und zu vergleichen, welche in weiterer Folge eine Klassifizierung von unbekanntem Quellcode vornehmen. Dadurch soll es möglich sein vorherzusagen, ob in der Methode eine richtig positive, eine falsch positive oder eine falsch negative Ausage von einem Werkzeug für statische Analyse getroffen wird. Hierfür werden Support-Vektor-Maschinen, der K-Nearest-Neighbor-Klassifikator, der Random Forest- als auch der RIPPER-Ansatz als Traininsmethode verglichen.

Um Lern- und Validierungsdaten zu generieren werden Basisdaten der öffentlich zugänglichen Juliet Test-Suite[4] vom NSA Center for Assured Software mit zwei Werkzeugen für statische Analyse durchlaufen. Für dieses Datenset mit Quellcode in C++ werden 21 Produkt-Metriken berechnet. Die generierten Hinweise aus der statischen Analyse werden den Zielwerten richitg positiv, falsch positiv und falsch negativ zugeordnet. Aufgrund der ungleichen Anzahl an Beispielen der einzelnen Kategorien in den Lerndaten wir die Bewertung anstatt mit Accuracy auf Basis des Wertes F-measure, welche auf Precision und Recall basiert, durchgeführt. Die Validierung der Vorhersagen erfolgt anhand einer Kreuzvalidierung mit zehn Teilmengen und darauf aufbauen zehn Testläufen. Großer Wert wird auf die Auswahl relevanter bzw. das Entfernen irrelevanter Produkt-Metriken gelegt. Hierzu werden im Vorfeld der Modell-Generierung die Metriken

[4] https://samate.nist.gov/SARD/test-suites/112

bzw. Kombinationen von Metriken zu den Zielwerten korreliert und irrelevante Daten vorab aus der Basis entfernt.

Alikhashashneh et. al. [19] kommen zu dem Ergebnis, dass die Klassifikationmethode Random Forest mit einem Wert der F-measure von 90,4% die besten Vorhersagen trifft. Modelle auf Basis des Support-Vektor-Maschinen-Ansatzes trafen in dieser Studie die Vorhersagen am unzuverlässigsten mit einem Wert der F-measure von 75,4%.

Moser et. al. [27] untersuchen in ihrer Arbeit die Qualität von Vorhersage-Modellen für Fehlerwirkungen, welche auf Basis unterschiedlicher Lerndaten trainiert werden. Die erste Datenbasis enthält Produkt-Metriken, ein zweite Datenbasis Prozess-Metriken und die dritte Menge an Lerndaten enthält die Kombination aus Produkt- und Prozess-Metriken für das Training von Modellen. Für alle drei Varianten wird jeweils eine zweite Version eines Modells berechnet, welche die Klassifikationsmodelle auf Basis einer zusätzlichen Kostenfunktion erstellt. Diese Kostenfunktion enthält fiktive Mehrkosten zur Behebung von unerkannten Fehlerwirkungen in ausgelieferten Software-Produkte in Relation zu den Behebungskosten zur Entwicklungszeit und versucht somit den Ressourcenverbrauch bzw. die Gesamtkosten zu optimieren Ziel der Studie ist es herauszufinden, ob Produkt- oder Prozess-Metriken bzw. eine Kombination dieser beiden eine höhere Vorhersage-Qualität in den Modellen liefern. Zusätzlich wird untersucht, ob Modelle unter Berücksichtigung fiktiver Korrekturkosten und der gemessenen Fehlerrate in der Vorhersage kosteneffektiv eingesetzt werden können. Es werden drei Klassifikationsmethoden, u.a. Entscheidungsbäume, eingesetzt.

Die Lern- und Validierungsdaten werden aus dem PROMISE Repository[5] für drei Releases von Eclipse verwendet, für welche neben dem Quellcode auch Daten der Fehlerwirkungen verfügbar sind. Zur Anwendung kommen 31 Produkt-Metriken und 18 Prozess-Metriken. Die Prozess-Metriken werden aus dem jeweiligen Repository der Software-Projekte errechnet und beschreiben Änderungsvorgänge, bspw. wie oft eine Quelldatei im Zuge von Fehlerkorrekturen angepasst wurde, oder wie viele unterschiedliche Autoren jeweils Änderungen an den Quelldateien vorgenommen haben. Zusätzlich wird eine Prozess-Metrik vorgestellt, welche das Alter einer Ressource nach neu hinzugefügten Zeilen Quellcode gewichtet, wobei größere und vor allem spätere (kürzer vor einem Release durchgeführte) Änderungen diese Maßzahl (*weighted age*) prägen. Für die Berücksichtigung fiktiver Korrekturkosten wird eine experimentelle Kostenfunktion aufgestellt. Diese wird für die Gewichtung der als fehlerfrei eingestuften Vorhersagen verwendet, welche jedoch falsch klassifiziert wurden und eine tatsächliche, zukünftige Fehlerwirkung beschreiben. Die Gewichtung wird in der Studie mit dem Faktor

[5] http://promise.site.uottawa.ca/SERepository/datasets-page.html

fünf definiert. Die Validierung der Vorhersagen erfolgt anhand einer Kreuzvalidierung mit zehn Teilmengen und darauf aufbauen zehn Testläufen.

Moser et. al. [27] beschreiben, dass ohne Berücksichtigung des Kostenfaktors alle drei Klassifikationsmethoden auf Basis der Prozess-Metriken eine deutlich bessere Vorhersage-Qualität als auf Basis von Produkt-Metriken liefern. Die Variante der kombinierten Lerndaten schnitt gleich bzw. nur geringfügig besser als die Variante auf Basis der Prozess-Metriken ab und rechtfertig laut der Studie daher die aufwendige Erhebung der Produkt-Metriken für die Trainingsbasis der Modelle nicht. Unter Einbeziehung der Gewichtung des Kostenfaktors liefert vor allem die Klassifikationsmethode der Entscheidungsbäume gute Ergebnisse. Auch hier werden mit dem Modell auf Basis der Prozess-Metriken die besten Vorhersage-Ergebnisse erzeugt. Die Accuracy auf Basis von Prozess-Metriken als auch der kombinierten Modelle sind miteinander vergleichbar und entsprechend den Werten ohne Berücksichtigung der Kostenfunktion. Vor allem weist jedoch die Variante der Prozess-Metriken mit einem Wert für Recall von über 80% in Kombination mit einer Rate von falsch positiven Vorhersagen von unter 30% nicht nur deutlich bessere Werte als das Modell auf Basis der Produkt-Metriken dar, sondern liefert hier auch eine bessere Vorhersage-Qualität als das kombinierte Modell. Die Studienergebnisse bestätigen dadurch auch weitere Arbeiten auf diesem Gebiet, welche unter Einbeziehung von Änderungsdaten von Quellcode bessere Vorhersage-Ergebnisse aufweisen als bei der ausschließlichen Betrachtung von produktbezogenen Metriken wie bspw. den Umfang oder die Komplexität von Software-Projekten.

Rathore et. al. [17] analysieren in ihrer Literaturstudie nach eigenen Angaben alle relevanten Arbeiten in diesem Gebiet ab 1993. Für die binäre Klassifikation kommen sehr viele, unterschiedliche Klassifikationsmethoden zum Einsatz. Mit einer durchschnittlichen Accuracy von 70% bis 85% und typischerweise darunter liegenden Werten für Recall gibt es keine klar bevorzugten Klassifikationsmethoden. Ein Grund dafür ist vor allem, dass unterschiedliche Datenbasen für die Vorhersage-Modelle eingesetzt werden und die Anwendungsdomänen nicht vergleichbar sind. Einfachen Klassifikationsmethoden wie bspw. der logistischen Regression, werden tendenziell bessere Ergebnisse zugeschrieben. Im Gegensatz dazu liefert die Literatur Hinweise, dass mit komplexeren Techniken in der Regel eine schlechtere Vorhersage-Qualität erreicht wird. Rathore et. al. [17] begründen dies mit der Komplexität bzw. größerem Aufwand die optimalen Parameter für diese komplexeren Klassifikationsmodelle zu bestimmen. Abschließend wird erwähnt, dass der unterschiedliche Einsatz von Maßzahlen für die Qualität der trainierten Modelle eine direkte Vergleichbarkeit der Ansätze zusätzlich erschwert.

4. Fallstudie

In diesem Kapitel wird die durchgeführte Fallstudie beschrieben. Zu Beginn weden das gewählte Vorgehensmodell und die daraus resultierenden drei Stufen inkl. der Zielsetzung vorgestellt. Anschließend wird zuerst die Analyse von Fehlerkorrekturen zur Erstellung einer qualitätsgesicherten Datenbasis (siehe Kapitel 4.2), weiters die Analyse der Code-Repositories zur Berechnung aussagekräftiger Software-Metriken (siehe Kapitel 4.3) und in Folge das Training und die Validierung von Vorhersage-Modellen auf Basis dieser Datenkonstellationen aufgezeigt (siehe Kapitel 4.4). Abschließend werden die Ergebnisse vorgestellt und diskutiert.

Im Rahmen der Fallstudie wird das Verhalten des in Abbildung 1 Ökosystem vorgestellten Systems untersucht. Das Zusammenspiel der handelnden Personen, welche Defects erkennen, erzeugen aber auch lösen, der organisatorischen Strukturen und der eingesetzten Technologien werden von außerhalb des Systems betrachtet. Grundlegendes Untersuchungsobjekt ist der Zusammenhang von vorab bekannten Hinweisen statischer Analysen zu zukünftigen Meldungen von Fehlerwirkungen im Fehlermanagement. Nach Hevner et. al. [28] erfolgt eine ausführliche Beobachtung dieses Prozessablaufes im produktiven Geschäftsbetrieb ohne Beeinflussung bzw. Interaktion mit den handelnden Personen. Auf Basis der Beobachtungen werden in Folge Zusammenhänge hergestellt und mit Hilfe der erhaltenen Daten nachvollziehbar begründet sowie die Qualität dieser prognostizierten Korrelationen beurteilt. Der konkrete Aufbau im Geschäftsumfeld der Raiffeisen Software GmbH hierzu wird im folgenden Kapitel detailliert beschrieben.

4.1. Aufbau der Fallstudie

Die Bearbeitung dieser Fallstudie folgt einem sequenziellen Ablauf in drei Stufen, welcher in Hinblick auf die grundlegende Analyse historischer Daten von bereits durchgeführten Fehlerkorrekturen und dem Training von Modellen an den in Kapitel 2.2 vorgestellten Prozessmodellen angelehnt ist. Besonderer Wert wird in dieser Fallstudie vor allem auf die Berechnung zusätzlicher, aussagekräftiger Software-Metriken für das Training von Modellen gelegt. Daher wird die Berechnung von Metriken mittels einer separaten, zweiten Stufe explizit hervorgehoben. In der dritten Stufe erfolgen das Training sowie die Validierung von Modellen. Die drei Stufen werden nacheinander abgearbeitet und bestehen jeweils wiederum aus drei Arbeitsschritten, welche je nach Anforderung zum Teil iterativ bearbeitet werden, siehe Abbildung 10 Vorgehensmodell für die Fallstudie.

Die Fertigstellung einer Stufe im Vorgehensmodell stellt einen Meilenstein dar und die (Zwischen-)Ergebnisse werden jeweils qualitätsgesichert und bewertet.

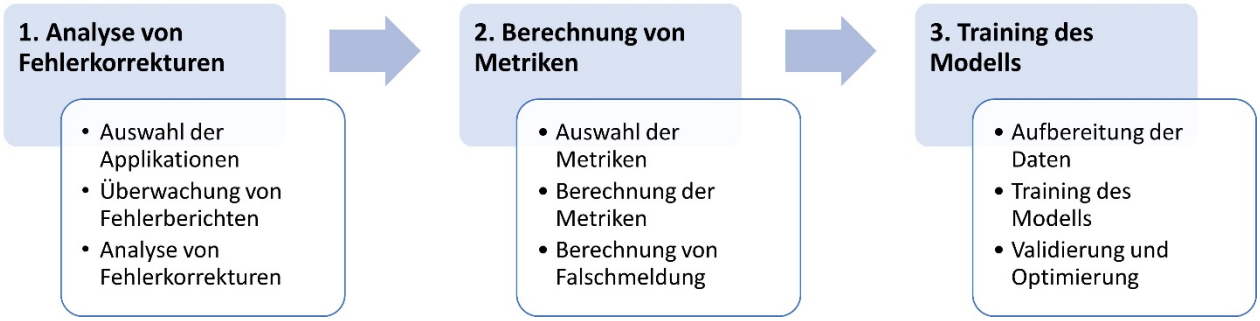


Abbildung 10 Vorgehensmodell für die Fallstudie

- In der ersten Stufe erfolgt die inhaltliche Analyse durchgeführter Fehlerkorrekturen. Ziel dieser Stufe ist es die Datenbasis für die weitere Berechnung von Metriken zu schaffen. Hierfür werden zuerst drei relevante Software-Produkte für die Fallstudie festgelegt und Fehlerberichte dieser für zwölf Monate im Fehlermanagement-Werkzeug Jira überwacht. Diese Software-Produkte bestehen jeweils aus wiederum mehreren Komponenten und zugehörigen Code-Repositories. Werden gemeldete Fehler durch Korrekturen an der Software, im Unterschied zu Anpassungen an der Infrastruktur, behoben, so wird eine statische Analyse der fehlerhaften Version der betroffenen Komponente durchgeführt. Hierfür kommen die zum Startzeitpunkt der Fallstudie aktuell eingesetzten Regelkataloge für die statische Analyse in SonarQube zum Einsatz. Wird im Zuge einer Fehlerkorrektur eine Zeile im Quellcode angepasst für die es einen gefundenen Hinweis in der statischen Analyse gibt und wird die durchgeführte Änderung durch ein Review mit der Korrektur in Verbindung gebracht, so werden allgemeine, statistische Daten des Repositories sowie Detaildaten der geänderten Datei sowohl seitens Fehlermanagement (Jira) und Versionsmanagement (Git) als auch der statischen Analyse (SonarQube) gespeichert. Bei den zur Verfügung stehenden Software-Projekten handelt es sich um Projekte im Bereich der Finanzdienstleistung. Die entsprechenden Quell-Systeme für die Erhebung der Daten sind streng geschützte Bereiche und es handelt sich in Folge um eine nicht öffentlich einsehbare Datenbasis. Um eine möglichst breite Wiederverwendung von generierten Modellen zu ermöglichen, wird der cross-project-Ansatz für die Auswahl der Software-Produkte verwendet (siehe Kapitel 2.2). Die drei Software-Produkte werden unabhängig voneinander entwickelt, weisen jedoch idente Entwicklungstechnologien und -methoden auf. Die verwendeten und in der statischen Analyse betrachteten Programmiersprachen sind Java, TypeScript und

HTML. Gemeinsam genutzte zentrale Services als auch zugelieferte Bibliotheken und Frameworks werden in der Fehleranalyse nicht berücksichtigt.

- In der zweiten Stufe des Vorgehensmodells erfolgt auf Basis der vorab erhobenen Daten die zielgerichtete Auswahl und Berechnung von Software-Metriken. Ziel ist es, eine qualitative, aussagekräftige und umfassende Datenbasis für das Training von Vorhersage-Modellen aufzubauen. Die kleinstmögliche, gut automatisierbare Ebene der Nachvollziehbarkeit von Änderungen am Quellcode welche in Jira, Git und SonarQube standardmäßig unterstützt wird, ist auf Ebene von einzelnen Ressourcen bzw. Dateien (vs. Paket- oder Methodenebene). Auf dieser Ebene werden aussagekräftige Software-Metriken eruiert, welche sowohl produkt- als auch prozessspezifische Maßzahlen enthalten. Vor allem bei den Prozess-Metriken wird großer Wert auf Betrachtung von Historien der entsprechenden Dateien gelegt. Hierzu werden unterschiedliche Werte berechnet, welche jeweils die letzten zwölf Monate ab der letzten Änderung an einer Ressource vor dem Release der Software, welche zur Meldung des Fehlers führte, durchgeführt wurde. Zusätzlich zu den Software-Metriken von Fehlerkorrekturen mit Hinweisen aus der statischen Analyse werden zusätzlich Metriken für Falschmeldungen (*false positives)* berechnet. Falschmeldungen sind in diesem Rahmen Hinweise der statischen Analyse, welche zum Zeitpunkt einer Korrektur im betroffenen Repository aufgezeichnet werden, jedoch im Betrachtungszeitraum, weder vorher noch nachher zu einem gemeldeten Fehler führen. Das Ergebnis dieser Stufe ist eine Liste von berechneten Software-Metriken unterschiedlicher Komplexität. Diese Werte sind für beide Kategorien, sowohl für echte Fehlermeldungen als auch für Falschmeldungen, vorhanden. Zu jedem Datensatz ist der Zielwert Defect bzw. Falschmeldung vorhanden.
- In der dritten Stufe werden die Software-Metriken technisch aufbereitet, um als Lernvektoren für das Training von Modellen Anwendung zu finden. Hier kommt der Ansatz der Klassifikation, als Teilbereich des überwachten Lernens, zum Einsatz. Dieses Vorgehen ist hier zielführend, da die Zielwerte der Lernvektoren in den Beispieldaten enthalten sind und im Zuge der Vorhersage jeweils eine konkrete Kategorie aus vorab bekannten gefunden werden soll (siehe Kapitel 2.5.1). Bzgl. der Klassifikationsmethode gibt es vorab keine Einschränkungen und es soll im Zuge der Fallstudie die passende Methodik erhoben werden. Besonderer Wert wird in dieser Stufe anschließend auf die Validierung der Ergebnisse und der Optimierung der Modelle gelegt. Hierfür kommen sowohl die Kombination unterschiedlicher Merkmalsausprägungen bzw. Software-Metriken der Lernvektoren als auch die Erprobung unterschiedlicher Parametrisierungen der eingesetzten Klassifikationsmethoden zum Einsatz. Ziel dieser Stufe ist das

Training von Modellen, welche in Bezug auf falschen bzw. korrekten Vorhersagen in der Anwendung bei vorab unbekannten Daten ein ausbalanciertes Verhältnis und in weiterer Folge Praxistauglichkeit aufweisen (siehe Kapitel 2.5.3).

Zielsetzung dieser Fallstudie ist die Beantwortung folgender Fragen:

- *F1: Gibt es Gemeinsamkeiten von Hinweisen statischer Analyse für welche Fehlerwirkungen nachgewiesen und Defects gemeldet wurden?*
- *F2: Wie können, ausgehend von relevanten Hinweisen statischer Analysen, praxistaugliche Modelle zur Vorhersage von Fehlerwirkungen generiert werden?*

4.2. Analyse von Fehlerkorrekturen

Als Grundlage für die folgenden Analysen werden drei Software-Produkte festgelegt. Die Auswahlkriterien umfassen sowohl die Architektur als auch die eingesetzten Programmiersprachen. Jedes Produkt besteht aus mehreren Komponenten, deren Quellcode in jeweils separaten Code-Repositories gepflegt wird und folgt dem Ansatz der mikroserviceorientierten Architektur. Jede Komponente wird separat versioniert und kann eigenständig released und installiert werden. Für die Entwicklung der Oberflächen kommt Angular und infolgedessen die Programmiersprache TypeScript zum Einsatz. Für die Programmierung der Services wird Java verwendet. Die betrachteten Software-Produkte und Komponenten stellen ausschließlich Eigenentwicklungen der Raiffeisen Software GmbH dar. Vom Marktzyklus und daher vom Reifegrad unterscheiden sich die Produkte voneinander (Einführung über Wachstum bis Reife), aber alle Produkte sind im produktiven Umfeld und bis zu den Endbenutzer*innen ausgerollt (Lebenszyklus Pilot bzw. Produktion). Als Release-Methode kommt für alle Produkte der agile Ansatz zum Zug, siehe Abbildung 11 Übersicht der Applikationen. Hierfür gibt es in der Raiffeisen Software GmbH u.a. die Vorgabe, dass Komponenten bzw. in weiterer Folge Produkte unabhängig voneinander released und beliebig ausgerollt werden können müssen. Um diese Anforderung zu erfüllen, werden Komponenten eigenständig als Mikroservices entwickelt und installiert. Für das Ausrollen einer neuen Produktversion kommt der Tranchen-basierte Ansatz zur Anwendung. Hierzu wird auf Basis verfügbarer Komponenten(-versionen) in einer Umgebung eine explizite Produkt-Releaseklammer erstellt, welche alle relevanten Komponenten, in der jeweils vorgesehenen Version inkl. deren transitiven Abhängigkeiten enthält. Diese Produkt-Releaseklammer wird ebenso versioniert und einer vorab definierten Tranche zugewiesen, welche neben Produkt-Version eine Zuordnung zu Gruppendefinitionen enthält und somit die vorgesehene Version für die jeweiligen Benutzer*innen bestimmt. Die Release-Zyklen der Entwicklungsteams folgen einem 14-tägigen Rhythmus, wobei in der Regel jedes Release

via Rollout für eine Gruppe von Fachexpert*innen zur Verfügung gestellt wird und jedes zweite Release bis zu den entsprechenden Endbenutzer*innen ausgerollt wird:

- SMART Beratung ist eine Beratersoftware, welche die Prozesse der Berater*innen in den Banken unterstützt und digital abbildet. Fehlerwirkungen, welche für SMART Beratung gemeldet werden, können sich auf 29 ausgewählte Komponenten beziehen, welche im Analysezeitraum mit unterschiedlicher Aktivität bearbeitet werden. Zu Beginn der Analysen umfassen die entsprechenden Code-Repositories dieser Komponenten in Summe ca. 284.000 Zeilen Quellcode.

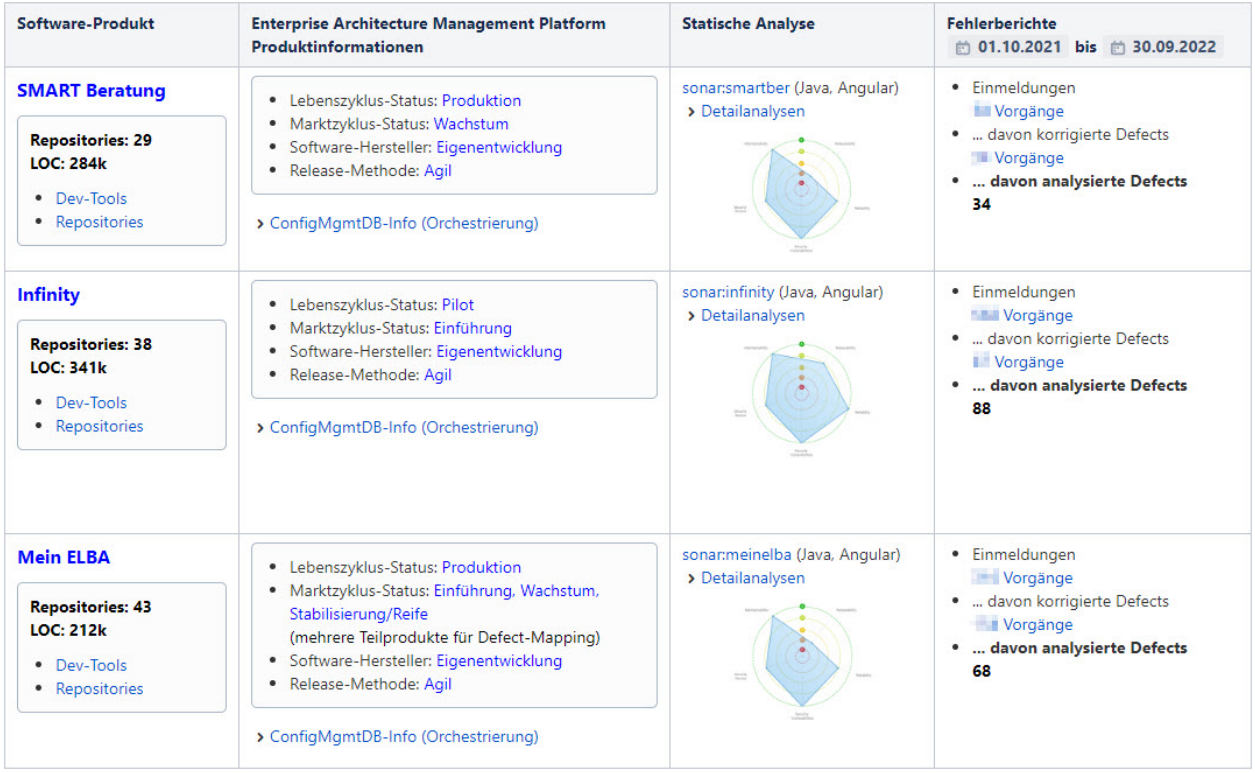


Abbildung 11 Übersicht der Applikationen

- Infinity ist das electronic banking-System für Geschäftskund*innen und ist aktuell in der Pilotierungsphase bei mehreren ausgewählten Unternehmen. Hierfür sind 38 Komponenten verknüpft und zu Beginn der Analysen werden in den Code-Repositories ca. 341.000 Zeilen Quellcode gehalten. Die Entwicklung für dieses Produkt findet in mehreren Entwicklungsteams statt.

- Das Pendant im electronic banking für Privatkund*innen ist Mein ELBA. Aufgrund der direkten Zugehörigkeit von 43 Komponenten ist die Entwicklung dieses Produktes ebenso auf mehrere Entwicklungsteams verteilt, aber auch fachlich in mehrere Teilprodukte gegliedert. Diese Produkte weisen einen Marktzyklus von Einführung über Wachstum bis Reife auf. Alle Teilprodukte gemeinsam halten dabei ca. 212.000 Zeilen Quellcode.

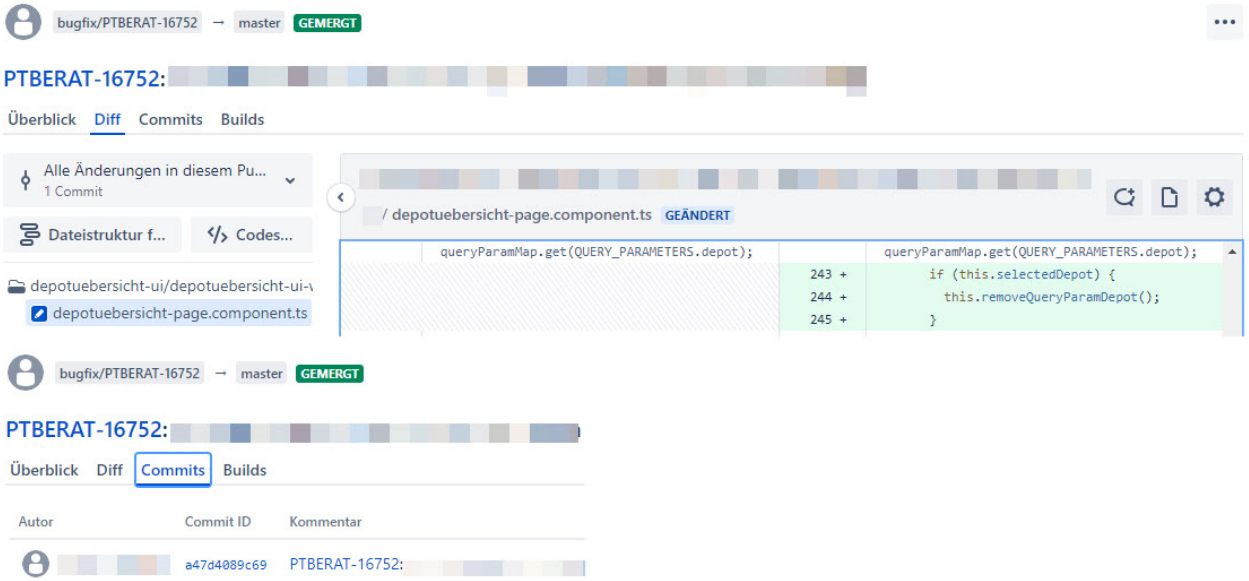


Abbildung 12 Änderungen am Quellcode

In der Applikationsübersicht sind jeweils die gemeldeten als auch die vom Entwicklungsteam korrigierten Fehlerwirkungen als Defects für den Betrachtungszeitraum der Fallstudie je Software-Produkt ersichtlich, siehe Abbildung 11 Übersicht der Applikationen. Erfolgt eine Einmeldung wird ein Defect in Jira erstellt. Wird die Bearbeitung dieses Defects qualitätsgesichert beendet, so wird der Defect geschlossen. Diese korrigierten Defects, welche durch Anpassungen an der Software behoben werden können, stellen die Basis für die weiteren Fehleranalysen dar.

Die korrigierten Defects enthalten eine Referenz zum Code-Managementsystem Git. Über diese Verknüpfungen können Daten zu inhaltlichen Änderungen eingesehen werden, bspw. welche Zeilen gelöscht, verändert oder hinzugefügt worden sind, siehe Abbildung 12 Änderungen am Quellcode. Zusätzlich sind die involvierten Personen, die geänderten Ressourcen als auch eine Commit-ID, welche eine Änderung in einem Repository eindeutig referenziert, enthalten. Über die Gesamtheit der Commit-IDs kann die inhaltliche Abfolge von Änderungen im Laufe der Zeit in einem Repository nachvollzogen werden. Für jede Version sind die Vorgängerversion als auch die Nachfolgeversion, ggf. auch mehrere, ersichtlich. Über diesen Mechanismus der Versionskontrolle in Git werden die Korrekturen vom Entwicklungsteam für die jeweils relevante Version der Komponente

erstellt. Bspw. wenn für frühere Versionen, welche noch über eine Tranche im produktiven Umfeld ausgerollt ist, eine nachträgliche Fehlerkorrektur erfolgen muss.

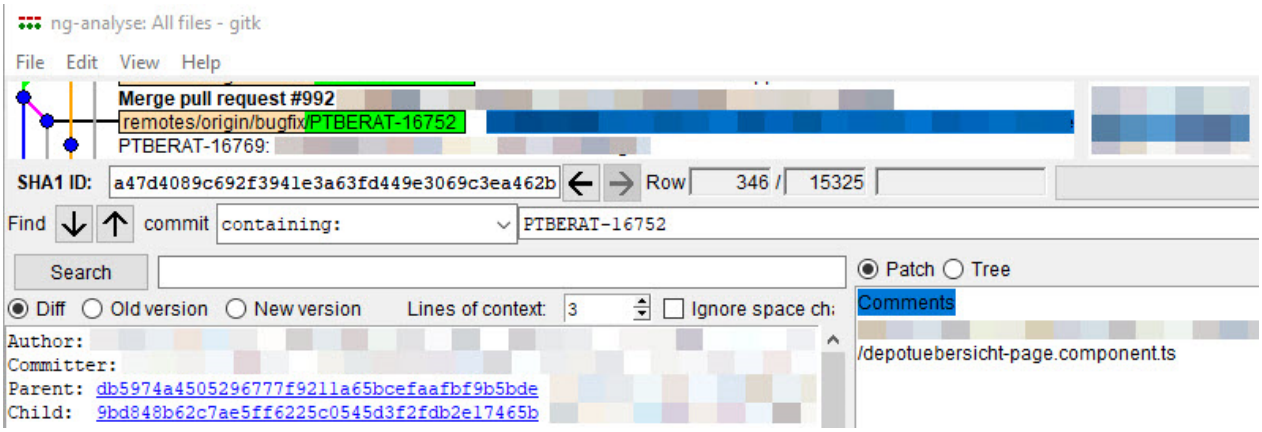


Abbildung 13 Nachverfolgbarkeit von Änderungen in Git

Derselbe Mechanismus wird ebenso für die Ermittlung der jeweils relevanten Ausgangsbasis für die Änderungsanalyse der Fehlerkorrektur herangezogen, um somit die fehlerbehaftete Version der Komponente wiederherzustellen. Über die eindeutige Commit-ID wird die entsprechende Ausgangsversion für die Korrektur (siehe *Parent* in Abbildung 13 Nachverfolgbarkeit von Änderungen in Git), ermittelt und diese wiederhergestellt. Für diesen Versionsstand wird in weiterer Folge über das Projektmanagement-Werkzeug Maven[6] per Kommandozeile der Befehl zur Ausführung der Phase *verify* erstellt, siehe Abbildung 14 Aufruf statischer Analyse über SonarScanner in Maven. Im Zuge dieses Befehles wir das Software-Projekt u.a. kompiliert, verpackt und über das Modul SonarScanner in Maven die statische Analyse in SonarQube angestoßen.

Die statische Analyse erfolgt auf Basis vordefinierter Regelkataloge je Programmiersprache. Für die Fallstudie wird SonarQube in der Version 8.9.6 eingesetzt und wurde zusätzlich um weitere Plugins zur Code-Analyse erweitert: Checkstyle in Version 8.39, FindBugs in Version 4.0.2 und PMD in Version 3.3.1. Dadurch wird eine höhere Anzahl an Prüfregeln ermöglicht, welche für die Durchführung der statischen Analyse konfiguriert und eingesetzt werden können. Für Java sind 685 Regeln und für TypeScript 164 Regeln ausgewählt. Für jede Regelverletzung wird ein eigenständiger Hinweis erstellt. Jeder Hinweis hat in SonarQube einen von drei Typen zugewiesen: *Bug*, *Vulnerability* oder *Code Smell*. Ebenso wird ein Schweregrad der Regelverletzung festgelegt, welcher die Werte *Blocker*, *Critical*, *Major*, *Minor* oder *Info* annehmen kann. Zusätzlich werden die Regelverletzungen aufgrund einer Zuordnung der Regel selbst

[6] https://maven.apache.org/

einer Kategorie zugeordnet, welche die Hinweise klassifiziert. Beispiele für derartige Kategorien sind *pitfall*, *bad-practice*, *brain-overload* und *duplicate*.

3_verify-scan.ps1 M | 0_vars.ps1 M | settings.xml

SonarAnalyse > 3_verify-scan.ps1

```
1  . ./0_vars.ps1
2  Set-Location $repobase\$projectkey\$repokey
3
4  Write-Output "___ Run maven sonar scanner"
5  $sonarid = "PREDICTION_" + $projectkey + "-" + $repokey
6  $sonarProjectKey = "-Dsonar.projectKey="+$sonarid
7  $sonarProjectName = "-Dsonar.projectName="+$sonarid
8  $mvnSettings = $scriptbase + "\settings.xml"
9
10 mvn -gs $mvnSettings clean verify sonar:sonar -U -B "-Dorg.apache.maven.global-settings=$mvnSettings"
   $sonarargs "-Dorg.slf4j.simpleLogger.defaultLogLevel=info" $sonarProjectKey $sonarProjectName
11
```

Abbildung 14 Aufruf statischer Analyse über SonarScanner in Maven

Im Zuge der Analysen kann beobachtet werden, dass bei Fehlerkorrekturen tlw. Hinweise statischer Analysen, welche durch Erweiterungen zusätzlich auch in den Entwicklungsumgebungen angezeigt werden, mitberücksichtigt und sofort korrigiert werden. Da diese Änderungen am Quellcode jedoch keinen inhaltlichen Zusammenhang mit der Fehlerkorrektur aufweisen, müssen diese Anpassungen durch qualitative Reviews gefiltert werden, da sonst die Belastbarkeit der Datenbasis nicht gegeben ist.

```
212 -
213 -      const blz = this.berUtilsService.getBlz();
214 -      const kundennummer = this.berUtilsService.getKundennummer();
215 -
216
264
265 -      if (!session || !session[params.key]) {
266          this.logger.error('No session value for key ' + params.key + ':
    ' + value);
267          return;
268        }
```

```
212
260
261 +      if (!session || !session.hasOwnProperty(params.key)) {
262          this.logger.error('No session value for key ' + params.key + ':
    ' + value);
263          return;
264        }
```

Abbildung 15 Inhaltliche Analyse von Fehlerkorrekturen

Hierzu werden im Zuge einer Fehlerkorrektur alle Änderung im Quellcode auf deren Bezug zur ursprünglichen Fehlerkorrektur überprüft. Im Beispiel aus Abbildung 15 Inhaltliche Analyse von Fehlerkorrekturen, stellt die Anpassung in Zeile 265 die Fehlerkorrektur dar und das Entfernen der beiden Zuweisungen in den Zeilen 213 und 214 stellt eine Wartungstätigkeit dar, welche als Beiwerk der Korrektur durchgeführt wird, um die Code-Qualität zu erhöhen. Diese Wartungstätigkeit hat aber keine Relevanz für die Korrektur der gemeldeten Fehlerwirkung. Lediglich für die relevanten Code-Änderungen werden in weiterer Folge die Hinweise aus statischer Analyse der

fehlerhaften Basisversion erhoben. In der Analyse werden sowohl Hinweise auf Zeileneben als auch hierarchische Hinweise auf Ebene der Methoden berücksichtigt.

Abbildung 16 SonarQube-Analyse der fehlerhaften Basisversion ohne Relevanz

Wie in Abbildung 16 SonarQube-Analyse der fehlerhaften Basisversion ohne Relevanz dargestellt, hat die statische Analyse für die beiden für die Fehlerkorrektur irrelevanten Code-Änderungen jeweils einen Hinweis vom Typ *Code Smell* mit dem Schweregrad *Major* erstellt. Da das Entfernen dieser beiden Zeilen jedoch in keinem inhaltlichen Zusammenhang mit der Fehlerkorrektur steht, dürfen diese Hinweise auch nicht in der Datenbasis berücksichtigt werden. Die fachlich relevante Anpassung in Zeile 265 hat keinen Hinweis in der fehlerhaften Basisversion. Somit ist die Korrektur dieses Defects nicht weiter relevant für die Datenerhebung. Würde die statische Analyse einen entsprechenden Hinweis in Zeile 265 anzeigen, unabhängig davon, ob dieser Hinweis im Zuge der Fehlerkorrektur behoben wird oder nicht, würden zusätzliche Daten zu Defect, Fehlerkorrektur, Repository und statischer Analyse in die Datenbasis mit aufgenommen werden. In Anhang 1 ist ein Beispiel einer Fehlerkorrektur mit Relevanz dargestellt. Es wird ein hierarchischer Hinweis gezeigt, welcher auf Methodenebene existiert. Der maximal erlaubte Wert für die Metrik *cognitive compelxity* von 15 wurde für diese Methode überschritten und beträgt in diesem Analyse-Durchlauf 81. Der Hinweis ist vom Typ *Code Smell* und hat den Schweregrad *Critical*. Erstellte Hinweise auf Basis dieser Regel werden der Kategorie *brain overload* zugeordnet, welche anzeigt, dass der Komplexitätsgrad eines Code-Segmentes einen empfohlenen Maximalwert überschreitet und daher bei Anpassungen oder Erweiterungen tendenziell davon ausgegangen werden kann, dass Fehlhandlungen mit höherer Wahrscheinlichkeit auftreten.

Im Zuge dieser Fallstudie wurden für die drei ausgewählten Software-Produkte 190 Defects, welche im Beobachtungszeitraum durch die Entwicklungsteams bearbeitet und geschlossen wurden, einem qualitativen Review unterzogen. Die statistische Verteilung ist in Abbildung 17 Defect-Statistik ersichtlich, die Namen der Software-Produkte werden in den weiteren Analysen durch eine Nummerierung ersetzt. Für das erste der drei ausgewählten Software-Produkte wurden von 34 korrigierten Defects insgesamt zwölf mit einem relevanten Hinweis aus statischer Analyse gefunden. Von diese zwölf Defects wurden neun in der Programmiersprache Java behoben und 3 in TypeScript. Von den relevanten Hinweisen in der statischen Analyse sind elf auf Methodenebene und ein Hinweis direkt in der entsprechenden Code-Ziele erstellt.

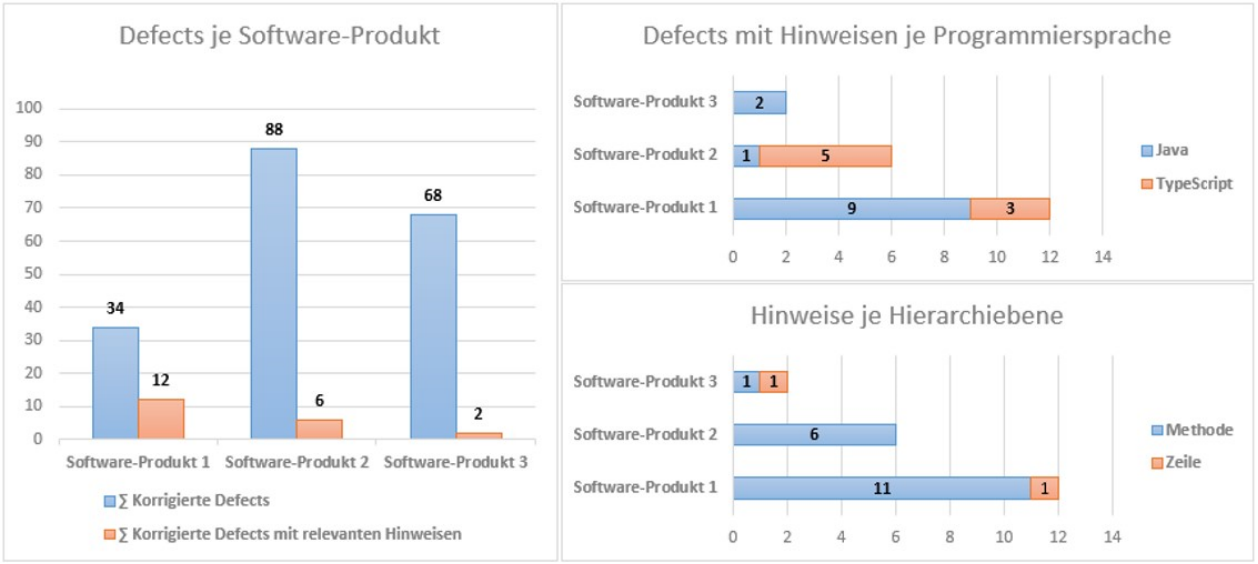


Abbildung 17 Defect-Statistik

Mit den in Summe 20 Defects wurde auf die Gesamtheit von 190 Defects eine Quote von 10,5% von relevanten Hinweisen statischer Analyse erreicht. Mit der Ausnahme von einem Hinweis, welcher das Entfernen von nicht erreichbarem Code aufzeigt, sind alle weiteren 19 Hinweise aufgrund des Überschreitens von festgelegten Komplexitätsmaßen erstellt und beschränken sich auf vier Prüfregeln: kognitive Komplexität[7] für sowohl Java (Prüfregel java:S3776) als auch TypeScript (Prüfregel typescript:S3776), die Anzahl der Parameter in der Methoden-Signatur für Java (Prüfregel java:S107) und die Komplexität von Regulären Ausdrücken in Java (Prüfregel java:S5843). Diese Prüfregeln werden in Stufe zwei weiterhin als Basis für die Suche nach Falschmeldungen genutzt. Ergebnis dieser Stufe ist eine Basis an Rohdaten, welche alle korrigierten Defects für die drei ausgewählten Software-Produkte im Betrachtungszeitraum enthält und, sofern es sich um einen Defect mit relevanten Hinweisen in statischer Analyse handelt, um zusätzliche

[7] https://www.sonarsource.com/docs/CognitiveComplexity.pdf

Daten zu Fundstelle und Detaildaten zum Hinweis als auch statistische Daten zum Repository. Ein Teil des Datenauszug für die Defects mit relevanten Hinweisen ist in Anhang 2 zu finden.

Erkenntnisse dieser ersten Stufe sind, dass bei Änderungen im Zuge von Defects, welche in einem frühen Stadium nach Entwicklung erkannt und korrigiert werden, tendenziell mehr Wartungsarbeiten, welche mit der Fehlerkorrektur in keinem inhaltlichen Zusammenhang stehen, durchgeführt werden, ebenso mehr Repositories und Dateien im Zuge der Korrekturen verändert werden und auch mehr Aufwände im Entwicklungsteam anfallen. Bei Korrekturen von Defects von Produkt-Versionen, welche bereits vollständig in der Fläche ausgerollt sind, fallen diese Merkmale jeweils geringer aus. Eine weitere Erkenntnis ist, dass natürlich die eingesetzten Prüfregeln Auswirkungen auf die Ergebnisse habe. Weitere Durchläufe statischer Analysen, im Zuge derer die Anzahl an Prüfregeln auf das Maximum angepasst wurde (1434 für Java und 211 für TypeScript), vor allem, um die Quote von 10,5% abgedeckter Defects zu erhöhen, führten zu einer enormen Anzahl von Hinweisen bei jedem einzelnen Defect. Auf Methodenebene zeigten sich mengenmäßig sehr viele Hinweise, welche regelrecht zu einem Überlauf der Datenbasis führen würden und in weiterer Folge daher auch an Aussagekraft verlieren. Ebenso ist interessant, dass fast ausschließlich Hinweise bzgl. Komplexitätsmaße auftraten, welche die Einfachheit mit der Quellcode gelesen bzw. verändert werden kann, messen. Weiters sind diese mit wenigen Ausnahmen bei der Analyse auf Methodenebene aufgetreten. Hier ist vor allem die Prüfregel der kognitiven Komplexität stark aufgefallen. Die Hinweise statischer Analyse, in diesem Fall Komplexitätsmäße, bleiben im Zuge von Fehlerkorrekturen in der Regel bestehen und werden nicht behoben. Ein erfolgskritischer Faktor ist vor allem die qualitative Analyse der Änderungen im Zuge von Fehlerkorrekturen. Da Entwicklungswerkzeuge im Standardfall mit Regelkatalogen zur statischen Analyse erweitert werden und die Hinweise in Echtzeit den Entwickler*innen angezeigt werden, werden diese im Zuge der Korrekturen beiläufig gelöst und würden bei einer vollautomatisierten Analyse in die Datenbasis mit aufgenommen werden, was die Qualität der Daten für folgende Korrelationen negativ beeinflussen würde.

4.3. Berechnung von Metriken

Auf Basis der Rohdaten aus Stufe 1 werden nun aussagekräftige Software-Metriken berechnet, welche mögliche Korrelationen der Daten aus statischer Analyse zu Fehlerwirkungen aufzeigen sollen.

Der Rahmen möglicher Software-Metriken wird durch Abbildung 5 Klassifikation von Metriken, in Anlehnung an [17] bestimmt. Einerseits werden grundlegende Daten bereits aus SonarQube bereitgestellt. Neben typischen Vertretern traditioneller Metriken wie Anzahl der Zeilen Quellcode auf Ebene von Repositories oder einzelnen Klassen werden Metriken zur Komplexität, bspw. kognitive Komplexität, über die erstellten Hinweise statischer Analyse definiert. Letztere Metriken stellen somit aus Sicht der Analyse den Auslöser für den Erhebungsprozess dar und werden nicht zusätzlich in die Datenbasis mit aufgenommen. Andererseits wird neben Vertretern der traditionellen Metriken vor allem auf Prozess-Metriken Wert gelegt. Auf Basis dieser Daten werden die Veränderungen, vor allem die Aktivität bzw. die Menge an Änderungen am Quellcode in den Repositories, berücksichtigt. Im Hinblick auf die in Moser et. al. [27] vorgestellten Ergebnisse, dass Vorhersage-Modelle auf Basis von Prozess-Metriken bessere Ergebnisse wie auf Basis von Produkt-Metriken und in der Regel ebenso gute Ergebnisse wie bei der Kombination von Produkt- und Prozess-Metriken liefern, wird im Zuge der Auswahl von Metriken ein Schwerpunkt daher auf Prozess-Metriken gelegt. Die für diese Arbeit ausgewählten Software-Produkte werden in 14-tägigen Zyklen entwickelt und released und in der Regel wird jedes zweite Produktinkrement in die Fläche ausgerollt. Daher kommt neuer Quellcode, im Vergleich zur konservativen Wasserfall-Releasemethode, vergleichsweise schnell ins produktive Geschäftsumfeld und wird durch Endkund*innen genutzt. Es wird somit die Annahme gestellt, dass neuer Code sowie neue Funktionen in Software-Produkten, welche noch nicht durch Endkund*innen genutzt wurden bzw. Code und Funktionen, welche im Vergleich zu anderen Teilen der Software neuer bzw. kürzer im produktiven Umfeld ausgerollt sind, tendenziell eine höhere Wahrscheinlichkeit haben unentdeckte Fehlerzustände zu enthalten und Fehlerwirkungen hervorzurufen. Daher wird in weiterer Folge vor allem auf Vertreter der Gruppe der Änderungsmaßzahlen aus der Kategorie der Prozess-Metriken Wert gelegt. In der weiteren Beschreibung wird vor allem auf die Erhebung und Berechnung dieser Maßzahlen eingegangen.

Für die Erhebung der Änderungsmaßzahlen werden die jeweiligen Repositories der betroffenen Komponenten im Versionsmanagement Git untersucht. Da eine Abfrage in Echtzeit auf die historischen Git-Strukturen umfangreicher Repositories

ressourcenintensiv und zeitaufwendig ist, wird hierzu über die Bibliothek JGit[8] jeweils das komplette Repository vorab ausgelesen und in einer flachen Objektstruktur persistiert. Diese Struktur orientiert sich anhand der chronologischen Abfolge aller Änderungen im Repository auf Basis von einzelnen Commits in einer Liste und zusätzlich drei HashMaps, welche aus fachlicher Sicht redundante Daten enthalten. Ziel dieser HashMaps ist es, dass performante Suchabfragen ermöglicht werden und über unterschiedliche Abfragedaten direkt auf die jeweils relevanten Indizes der vollständigen Liste aller Commits zugegriffen werden kann, siehe Abbildung 18 Struktur der Klasse CommitList.

```
/**
 * The class CommitList hosts a list object with the whole commit history of a
 * repository (based on git log) as a sorted list of <strong>CommitList</strong>
 * objects (last commit is first object in list and first commit is last object
 * in list) an provides serveral optimized methods for searching and calculation
 * based in this structure.
 */
public class CommitList implements Serializable {

    private static final long serialVersionUID = -2659144152073191l4L;

    /**
     * List of CommitInfo instances (id, timestamp, author, filepathnames). Sorted
     * by timestamp of commit. Empty pull requests commits are omitted.
     */
    private ArrayList<CommitInfo> list;

    /**
     * Mapping from a commitId to list index.
     */
    private Map<String, Integer> commitIdMap;

    /**
     * Mapping from a filepathname to the list index.
     */
    private Map<String, List<Integer>> filepathMap;

    /**
     * Mapping from a Jira Defect-ID to all included commitIds. (A Jira Defect-ID
     * has one or more pull request that in turn have one or more commitIds).
     */
    private Map<String, Set<String>> jiraIdMap;
```

Abbildung 18 Struktur der Klasse CommitList

Die erste HashMap (commitIdMap) ermöglicht den direkten Index-basierten Zugriff über eine eindeutige Commit-ID auf das relevante Segment der Liste und dadurch in weiterer Folge zu den zugehörigen Daten wie Zeitstempel, Autor, Jira Defect-ID als auch alle veränderten Dateien im Zuge des Commits. Die nächste Struktur (filepathMap) enthält alle Indizes der Liste, welche im Zusammenhang zu Änderungen einer bestimmten Datei stehen. Die dritte Struktur (jiraIdMap) enthält alle Indizes, welche zu Jira Defect-IDs gehören. Alle drei Strukturen werden im Zuge der initialen Befüllung der Liste angelegt und dienen ausschließlich dem performanten Zugriff auf relevante Segmente der

[8] https://www.eclipse.org/jgit/

gesamten Liste, ohne diese für zukünftige Berechnungen im gesamten durchlaufen zu müssen. Auf dieser Basis sind einfache Auswertung wie die Summe aller Commits für eine definierte Ressource während einer vorgegebenen Zeitspanne, oder die Menge aller Commits zur Behebung eines Defects auf Basis der Jira Defect-ID effizient möglich. Dies wird eingesetzt, um folgende Werte für alle Defects mit Hinweisen aus statischer Analyse zu berechnen:

- Die *Maintainability* definiert, wie häufig eine bestimmte Ressource in einem Repository verändert wird.
- Die *Correctness* stellt eine Spezialisierung der Maintainability dar und definiert, wie oft eine Ressource im Zuge von Fehlerbehebungen verändert wird.
- Der Wert *Author* gibt an wie viele unterschiedliche Autoren Änderungen an einer Ressource vorgenommen haben.

Für sowohl Maintainability als auch Correctness werden jeweils zwei Ausprägungen berechnet:

- Der *Index* stellt eine relative Maßzahl dar und stellt Änderungen an einer Ressource im Vergleich zu der gesamten Anzahl an Änderungen im Repository dar.
- Die Ausprägung *Value* stellt den konkreten, absoluten Wert der Veränderungen dar.

Somit ergeben sich für die angeführten Werte mit den möglichen Ausprägungen die Varianten MaintainabilityIndex, MaintainabilityValue, CorrectnessIndex, CorrectnessValue und AuthorValue. Der MaintainabilityIndex definiert dabei die Anzahl von Änderungen für eine bestimmte Ressource, bspw. eine Java- oder TypeScript-Datei, im Vergleich zur Menge aller Software-Änderungen im Repository für denselben Zeitraum, bspw. 4 Commits mit Änderungen einer Ressource von insgesamt 100 Commits im Repository ergeben einen MaintainabilityIndex von 0,04. Der CorrectnessValue beschreibt den absoluten Wert, wie oft eine Ressource im Zuge von Fehlerbehebungen in einem bestimmten Zeitraum geändert wurde, bspw. 2. Die Werte für Maintainability und Author können effizient auf Basis der Git-Repositories und der eingesetzten, flachen Datenstruktur berechnet werden. Für alle Ausprägungen des Wertes Correctness ist eine umfassende, den gesamten Auswertungszeitraum abdeckende Analyse aller Fehlermeldungen inkl. qualitativer Analyse der Ressourcenzuordnung notwendig, unabhängig davon, ob Hinweise aus statischer Analyse vorliegen oder nicht. Dies stellt einen zusätzlichen, vor allem kontinuierlich anfallenden Aufwand für die Bereitstellung notwendiger Berechnungsdaten dar.

Um darüber hinaus den jeweils zeitlichen Verlauf dieser Werte zu berücksichtigen, werden zusätzlich unterschiedliche statistische Rechenmethoden angewendet. Dazu wird die Historie an Änderungen von Ressourcen in Segmente unterteilt. Ein Segment repräsentiert den Zeitraum innerhalb dessen Änderungen an der Software in Produktion ausgerollt werden und ergibt sich für diese Arbeit aus der Dauer eines Entwicklungszyklus von 14 Tagen und dem Umstand, dass typischerweise jedes zweite Release ausgerollt wird. Ein Segment beschreibt daher den zeitlichen Verlauf von zwei Entwicklungszyklen und wird der Einfachheit halber mit der Zeitspanne von einem Monat berechnet. Für jeden Wert wird eine Historie von zwölf Segmenten in die Vergangenheit gerechnet und somit jeweils das letzte Jahr ab Ausgangsdatum berücksichtigt. Die Berechnung der Segmente auf Basis der Repositories (siehe Abbildung 18 Struktur der Klasse CommitList) bzw. für die Werte für Correctness zusätzlich auf Basis der gesamten Defect-Rohdaten aus Stufe 1 erfolgt in jeweils separat implementierten Berechnungsklassen, siehe Abbildung 19 Calculator-Klassen zur Berechnung von Segmenten.

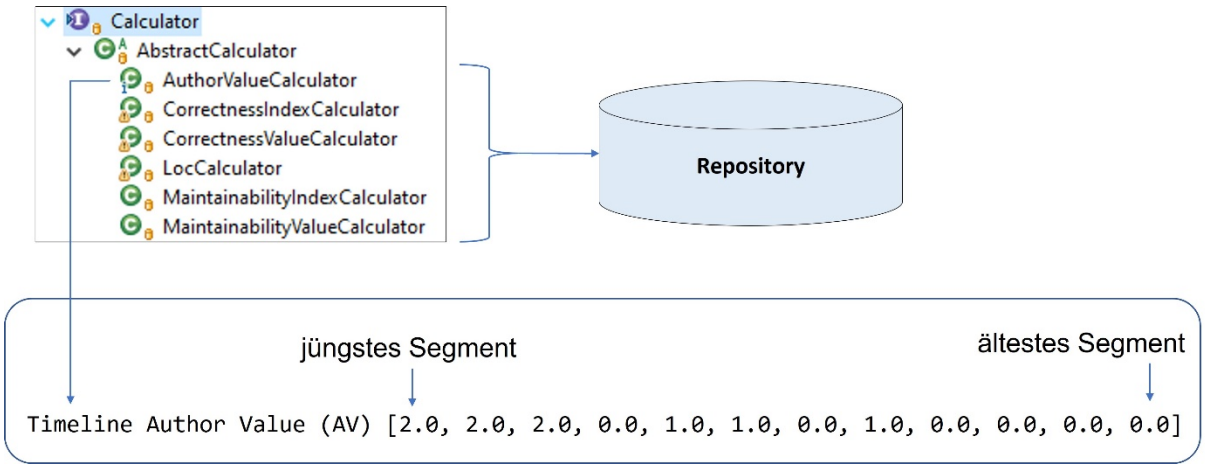


Abbildung 19 Calculator-Klassen zur Berechnung von Segmenten

Für jeden relevanten Wert liegt, sofern die zugrunde liegende Ressourcen das Mindestalter von einem Jahr erreicht, eine entsprechende, zeitlich sortierte Reihe an Segmenten vor. Erfüllt die Ressource dieses Mindestalter nicht, werden fehlende Segmente mit dem Wert „0“ aufgefüllt. Für diese Zeitreihen werden in einem ersten Schritt Durchschnittswerte berechnet.

- Der arithmetisch gleitende Durchschnitt (*simple moving average: SMA*) gewichtet die Werte aller Segmente gleich.
- Der linear gewichtete gleitende Durchschnitt (*linearly moving average: LWMA*) gewichtet die Werte jüngerer Segmente stärker als die Werte älterer Segmente, wobei der Einfluss der Gewichtung aufgrund der Historie linear ist.

- Der exponentiell gleitende Durchschnitt (*exponential moving average: EMA*) schenkt den Werten jüngerer Segmente höhere Bedeutung, wobei die Gewichtung aufgrund der Historie exponentiell ist.

Ziel dieser Durchschnittswerte ist es, aufgrund einer Wertereihe von zwölf Segmenten einen fiktiven nächsten Folgewert zu errechnen, wobei unterschiedliche Sensitivität auf das Alter der Werte der Segmente gelegt wird. Beim exponentiell gleitenden Durchschnitt wird eine weitaus stärkere Bedeutung auf jüngere Werte gelegt und die berechneten Kennzahlen sind daher weitaus empfänglicher für Änderungen im Trend der zwölf Segmente im Vergleich zum linear gewichteten Durchschnitt und vor allem zum arithmetisch gleitenden Durchschnitt, welcher die zeitliche Anordnung der Werte in der Berechnung nicht berücksichtigt.

In einem weiteren Schritt wird für alle Werte der Zeitreihen die lineare Funktion ermittelt und die Steigung dieser Funktion (*slope: LRSlope*) berechnet. Ziel ist aufgrund dieser Kennzahl den Trend zu bewerten. Ist der berechnete Wert positiv, so ist die Steigung und in weitere Folge die Trendentwicklung positiv zu sehen. Je höher der Wert, desto steiler die Steigung bzw. die Trendentwicklung. Ist der berechnete Wert negativ, ist dementsprechend die Trendentwicklung negativ zu sehen.

Die Anwendung dieser Rechenmethoden wird für alle Segmente aller Calculatoren für alle berechneten Wert-Ausprägungs-Kombinationen für alle analysierten Defects mit relevanten Hinweisen aus statischer Analyse angewendet. Für die erhobene Wertereihe für AuthorValue der Ressource in Abbildung 19 Calculator-Klassen zur Berechnung von Segmenten werden über die Klasse Metrics die Werte SMA, LWMA, EMA und LRSlope berechnet. Im Verlauf des letzten Jahres ab Auswertungsdatum haben drei unterschiedliche Autoren Änderungen an dieser Ressource vorgenommen. Der arithmetisch gleitende Durchschnitt (SMA) hat einen Wert von 0,75 und der exponentiell gleitende Durchschnitt (EMA) hat durch die stärkere Gewichtung jüngerer Segmente einen höheren Wert von 1,9372, siehe Abbildung 20 Statistische Bewertungen von Segmenten.

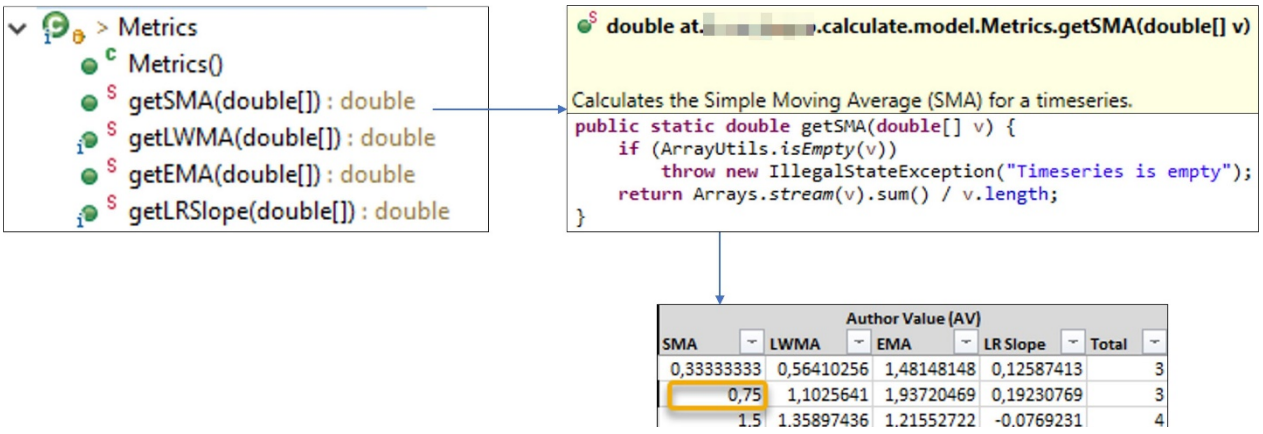


Abbildung 20 Statistische Bewertungen von Segmenten

Neben der Berechnung der Metriken für relevante Defects, werden auch Metriken für Falschmeldungen erhoben. Falschmeldungen sind Hinweise aus statischer Analyse, welche zum Zeitpunkt der Analyse der fehlerhaften Basisversion eines relevanten Defects im selben Repository vorhanden sind, jedoch im weiteren Beobachtungszeitraum nicht zu einer Fehlerwirkung bzw. zu einer Fehlermeldung in Jira führen. Die Ergebnisse der statischen Analyse werden automatisiert über die SonarQube-API abgefragt und verarbeitet. Für alle Instanzen von Hinweisen für Falschmeldungen werden ebenso alle Metriken wie für relevante Defects berechnet. Das grundlegende Vorgehen der Suche relevanter Falschmeldung zur Berechnung von Metriken wird in Abbildung 21 Suche nach relevanten Falschmeldungen skizziert.

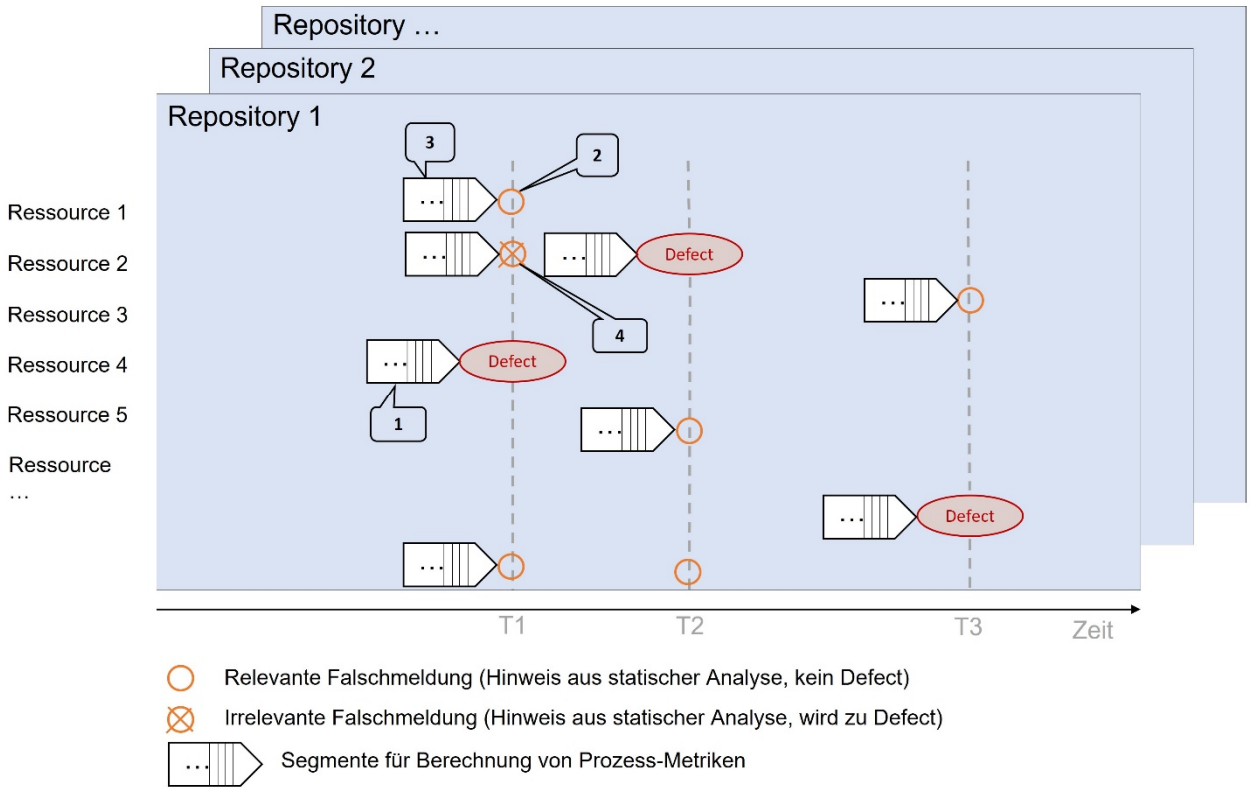


Abbildung 21 Suche nach relevanten Falschmeldungen

1. Erhebung der Historie und Berechnung der Segmente sowie darauf aufbauend aller Metriken für Defects mit relevanten Hinweisen aus statischer Analyse auf Basis der fehlerhaften Basisversion (der gemeldete Defect ist Auslöser der Erhebung zum Zeitpunkt T1), siehe Punkt 1 für Ressource 4.
2. Suche nach allen Hinweisen statischer Analyse in SonarQube, welche den erhobenen Regelkategorien entsprechen und in der fehlerhaften Basisversion (Zeitpunkt T1) des Defects enthalten sind, siehe Punkt 2 für Ressource 1.
3. Erhebung der Historie und Berechnung der Segmente sowie darauf aufbauend aller Metriken für Falschmeldungen, siehe Punkt 3 für Ressource 1.

4. Nach Erhebung aller Daten für sowohl relevante Defects als auch Falschmeldungen: Entfernung von Metriken für Falschmeldungen, welche im weiteren zeitlichen Verlauf des Beobachtungszeitraumes zu einem relevanten Defect führen, siehe Punkt 4 für Ressource 2 zu Zeitpunkt T1, welcher zum Analysezeitpunkt T2 zu einem relevanten Defect führt.

Das Ergebnis der zweiten Stufe ist die erweiterte Rohdatenbasis um die berechneten Software-Metriken. Neben traditionellen Maßzahlen wie der Anzahl an Zeilen Quellcode werden vor allem Änderungsmaßzahlen für sowohl alle gemeldeten Fehlerwirkungen mit relevanten Hinweisen in statischer Analyse als auch für alle relevanten Falschmeldungen, welche in einer fehlerhaften Basisversion im Zuge einer Analyse eines Defects aus SonarQube erhoben werden können, zur Verfügung gestellt. Ein exemplarischer Datenauszug, welcher als Erweiterung zusätzlich zu den in Stufe 1 erhobenen Rohdaten gespeichert wird, ist in Anhang 3 zu sehen.

Erkenntnisse dieser zweiten Stufe sind, dass die Standardfunktionalitäten in Java-Bibliotheken für Git zur Suche und Analyse von Datenbeständen für Repositories mit einer vergleichsmäßig geringen Anzahl an Commits gut sind, aber bei umfangreicheren Repositories mit vor allem einer höheren Anzahl an Commits nicht performant agieren (bspw. für Repositories > 15.000 Commits). Hierfür war die Implementierung einer passenden Datenstruktur zielführend, welche Redundanzen in der Datenhaltung aufwies, aber vor allem dem Anspruch performanter Suche und Navigation in der Bearbeitungshistorie von Ressourcen gerecht wird. Weiters wurden Pfad-Anpassungen oder Namensänderungen von Ressourcen in den Analysen explizit berücksichtigt, um auch in diesem Fall alle Änderungen an Ressourcen für die Berechnung der Metriken zur Verfügung stellen zu können. Ebenso war die Berechnung und in Folge die Gewichtung von neuem Quellcode bzw. neuen Funktionalitäten eine wichtige und wertvolle Erweiterung. Hier stellt die Berechnung von Trends und unterschiedlichen Gewichtungen des zeitlichen Verlaufes der Änderungen an Ressourcen die relevante Erweiterung dar. Zuletzt brachte die Suche von relevanten Falschmeldungen zur Korrelation die Erkenntnis, dass vor allem zwischen relevanten Falschmeldungen und irrelevanten Falschmeldungen (jene, für die im weiteren Zeitverlauf ein Defect gemeldet wird) unterschieden werden muss, da tlw. relevante Defects bereits vorab als Falschmeldungen über die SonarQube-Suche erfasst werden.

4.4. Training von Modellen

Auf Basis der um die in Stufe 2 berechneten Software-Metriken erweiterte Datenbasis von sowohl relevanten Defects als auch Falschmeldungen werden nun diese Daten in einem entsprechenden Format aufbereitet, um in weiterer Folge Modelle in einem iterativen Vorgehen zu trainieren und validieren, um praxistaugliche Ansätze zu erzeugen.

Zielsetzung ist es, Korrelationen der vorhandenen Rohdaten und Software-Metriken zu den Zielwerten, Defect bzw. Falschmeldung, zu finden. Hierzu wird der Ansatz des überwachten Lernens gewählt (siehe Kapitel 2.5.1). Als Vorbereitung werden die Rohdaten und berechneten Software-Metriken als maximal mögliche Anzahl an Merkmalsausprägungen des Lernvektors in das CSV-Format exportiert. Zusätzlich wird zu jedem Beispieldatensatz der vorab bekannte Zielwert ergänzt. Da mit einem Überhang von ca. 1:45 eine starke Diskrepanz an Beispieldaten zwischen relevanten Defects zu Falschmeldungen herrscht, werden im Zuge des Exports die Beispieldaten für relevante Defects in dem Verhältnis der Diskrepanz vervielfältigt (*oversampling,* siehe [29]). In Summe besteht die gesamte Datenmenge aus 1838 Beispieldaten, davon 923 Datensätze von relevanten Defects (durch Vervielfältigung der 20 konkreten Datensätze relevanter Defects) und 915 Datensätze von Falschmeldungen.

Für die Verarbeitung der Daten und Training sowie Validierung der Modelle wird die in Java implementierte Bibliothek Tribuo[9] von Oracle verwendet. Tribuo hat Schnittstellen zu vielen bekannten, nativen Implementierung für maschinelles Lernen und bietet Algorithmen für Klassifikation, Regression und die Clusteranalyse. Für Training und Validierung von Modellen wird in Tribuo standardmäßig das CSV-Datenformat verwendet. In einem ersten Schritt wird die Tauglichkeit der in den ersten beiden Stufen erhobenen Daten geprüft, vor allem, ob eine Korrelation der Lernvektoren zum jeweiligen Zielwert erkennbar ist. Hierfür werden jeweils die vier Implementierung LinearSGDTrainer, RandomForestTrainer, KernelSVMTrainer und XGBoostClassificationTrainer im Vergleich zu einem DummyClassifierTrainer trainert, welcher auf Basis einer 50:50-Verteilung zufällig einen Zielwert bestimmt. Die Durchführungen wurden sowohl mit dem maximalen Lernvektor als auch mit kleineren Untermengen durchgeführt. Im Unterschied zur maximalen Ausprägung des Lernvektors konnte für kleinere Teilmengen (*feature sets*) eine deutlich bessere Korrelation und Genauigkeit beobachtet werden. Vor allem der exponentiell gleitende Durchschnitt (EMA) als auch die Steigung der linearen Funktion der Zeitreihen (LRSlope) für jeweils MaintainabilityIndex, MaintainabilityValue und AuthorValue in Kombination mit Anzahl an Zeilen Quellcode (LOC) stellten sich als

[9] https://tribuo.org/

wertvolle Merkmalsausprägungen dar. Diese Teilmenge an sieben Merkmalsausprägungen wurde eingesetzt, um in weiterer Folge sowohl zusätzlich mehrere Trainer-Implementierungen von Tribuo einzusetzen als auch die teilweise sehr umfangreiche Parametrisierung der Algorithmen zu verfeinern. Zusätzlich zu den im Zuge der Datenprüfung eingesetzten Trainer-Varianten kamen KNNTrainer, LibLinearClassificationTrainer, LibSVMClassificationTrainer und MultinomialNaiveBayesTrainer zum Einsatz. Die Sensitivität der Parametrisierung als auch die Auswirkungen von Erweiterungen der Lernvektoren um zusätzliche Merkmalsausprägungen erfolgte vorab manuell. Wie in Abbildung 22 Modell-Training dargestellt, wurde hierzu der jeweilige Trainer (bspw. KernelSVM) parametrisiert erzeugt und in Kombination mit einem definierten Set an Merkmalsausprägungen des Lernvektors (fieldset 4) verwendet. Für die gesamten Datenbasis wurden auf Basis der gewählten Merkmalsausprägungen jeweils ein Datenset für das Training des Modells und ein weiteres für den Test erstellt. In einem weiteren Schritt wurde das Modell auf Basis des Lerndatensets erzeugt und auf Basis des Testdatensets evaluiert. Abschließend wird das erzeugte Modell gespeichert.

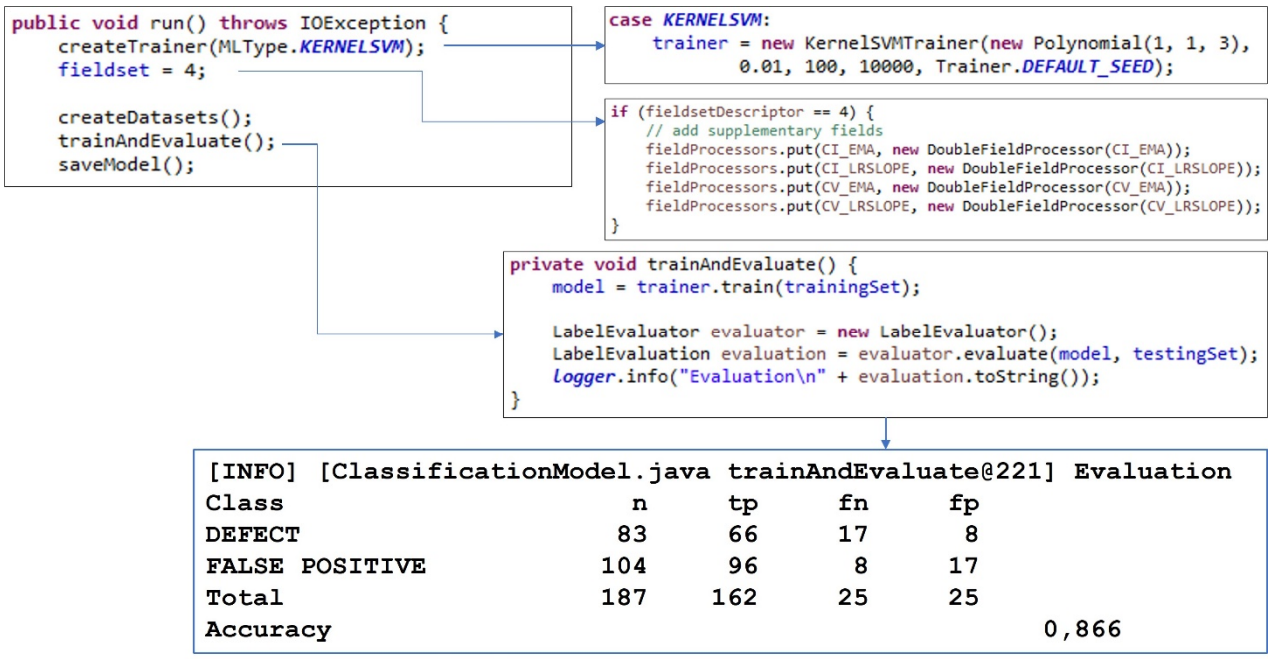


Abbildung 22 Modell-Training

Eine erste, grobe Bewertung der erzeugten Modelle wurde auf Basis der Accuracy durchgeführt, der Summer von richtig positiven und richtig negativen Vorhersagen im Vergleich zu der gesamten Menge von Beispieldaten. Wie in Abbildung 22 Modell-Training auf Basis eines kleines Datensets dargestellt, wurden 66 der 83 Defects und 96 der 104 Falschmeldungen korrekt erkannt und somit eine Accuracy von 86% erreicht. Durch das manuelle Vorgehen konnte eine Verbesserung der Genauigkeit bei sechs der insgesamt acht eingesetzt Trainer-Implementierungen beobachtet und für jeden Trainer

in einem ersten Ansatz die relevanten Parameter zu Erhöhung der Sensitivität festgestellt werden (LinearSGDTrainer, XGBoostClassificationTrainer, CARTClassificationTrainer, LibLinearClassificationTrainer, LibSVMClassificationTrainer und KernelSVMTrainer). Um in weiterer Folge die Qualität der erzeugten Modelle in einem umfassenderen Ansatz iterativ zu bewerten, wurde die Gesamtmenge der Beispieldaten in jeweils drei gleich große Datenmengen, sowohl für relevante Defects als auch für Falschmeldungen, eingeteilt, um diese für eine Kreuzvalidierung zu nutzen (siehe Kapitel 2.5.3). Obwohl die Bibliothek Tribuo bereits Funktionalitäten für den automatisierten Einsatz von Kreuzvalidierung anbietet, wurde die Einteilung der drei Teilmengen manuell durchgeführt und ist statisch. Dadurch wird einerseits sichergestellt, dass die drei Teilmengen für relevante Defects klar abgegrenzt sind und keine Duplikate eines Defects in mehreren Teilmengen vorkommen als auch andererseits die Teilmengen dauerhaft nachvollziehbar bestehen und darauf basierende Resultate reproduzierbar sind. In weiterer Folge wurde für jede Kombination einer konkreten Parametrisierung eines Trainers und eines Datensets drei Iterationen angewendet. Dabei wurde jeweils ein Modell auf Basis von zwei Datensets trainiert und gegen das jeweils dritte Datenset mit, aus Sicht des erstellen Modells, neuen und unbekannten Vektoren validiert. Die jeweils gemeinsame Bewertung dieser drei Modelle wurden für die Bewertung der Trainer-Testdatenset-Kombination herangezogen, siehe Abbildung 23 Abgegrenzte Teilmenge für Kreuzvalidierung.

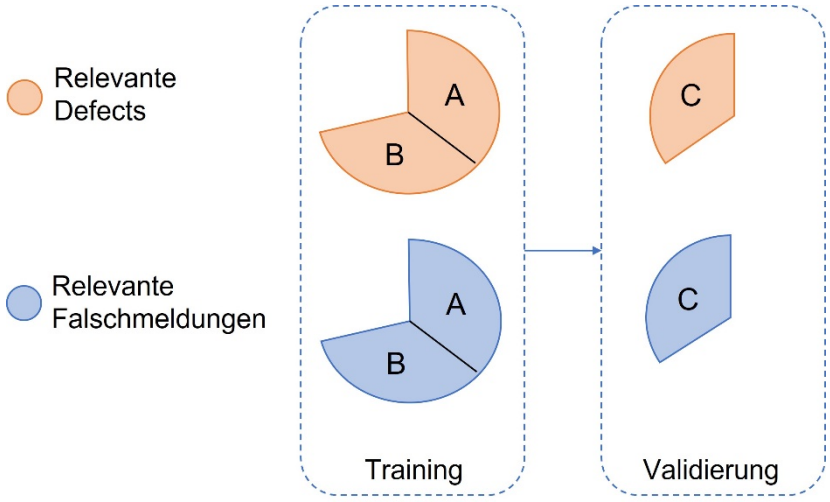


Abbildung 23 Abgegrenzte Teilmenge für Kreuzvalidierung

Auf Basis der vorhandenen Teildatenmengen für Training und Validierung als auch einer kleineren Anzahl von zielführenden Trainer-Implementierung und insgesamt acht unterschiedlichen Varianten von Merkmalsausprägungen für Lernvektoren, welche im Zuge der manuellen Überprüfung gute Ergebnisse lieferten, wurden durch Kombination alle möglichen Ausprägungen berechnet. Für die angeführten sechs Trainer-Implementierung wurden insgesamt 86 Parametrisierungen mit jeweils acht Varianten an Merkmalsausprägungen für den Lernvektor mit den statischen Teilmengen der

Beispieldaten einer 3-fach Kreuzvalidierung unterzogen. Für die resultierenden 2063 Kombinationsmöglichkeiten wurde jeweils ein Modell generiert und bewertet, siehe Abbildung 24 Kombination von Trainer, Beispieldaten und Lernvektoren. Die Bewertung der jeweils drei Modelle mit identen Trainer-Parametrisierungen und identen Lernvektoren wurde vereinheitlicht und mit den anderen Berechnungen verglichen.

```
public void run() {
    for (Trainer<Label> trainer : getTrainer()) {
        for (int fp = 0; fp <= FIELDPARAMS_VARIATIONS; fp++) {
            /**
             * Pool of 3 distinct datasets for defects and false positives.
             * Mutually train models based on 2 mixed datasets and evaluate to defects and
             * false positives of the third dataset.
             */
            Model<Label> modelAB = trainAndEvaluate(trainer, TRAINING_AB_MIXED, fp);
            predictList(modelAB, PREDICTION_C_DEFECTS, ClassificationModel.DEFECT, fp, "AB->C");
            predictList(modelAB, PREDICTION_C_FALSEPOSITIVES, ClassificationModel.FALSEPOSITIVE, fp, "AB->C");

            Model<Label> modelAC = trainAndEvaluate(trainer, TRAINING_AC_MIXED, fp);
            predictList(modelAC, PREDICTION_B_DEFECTS, ClassificationModel.DEFECT, fp, "AC->B");
            predictList(modelAC, PREDICTION_B_FALSEPOSITIVES, ClassificationModel.FALSEPOSITIVE, fp, "AC->B");

            Model<Label> modelBC = trainAndEvaluate(trainer, TRAINING_BC_MIXED, fp);
            predictList(modelBC, PREDICTION_A_DEFECTS, ClassificationModel.DEFECT, fp, "BC->A");
            predictList(modelBC, PREDICTION_A_FALSEPOSITIVES, ClassificationModel.FALSEPOSITIVE, fp, "BC->A");
        }
    }
}
```

Abbildung 24 Kombination von Trainer, Beispieldaten und Lernvektoren

Da das Verhältnis der Beispieldaten für das Training der Modelle nicht ausgewogen ist und die Anzahl der Beispiele für relevante Defects gegenüber den Falschmeldungen zwar mengenmäßig annähernd gleich, aber durch die Vervielfältigung der Defects inhaltlich unterlegen sind, kommt in diesem Fall die Accuracy für die Bewertung nicht zum Einsatz, siehe Kapitel 2.5.3. Vielmehr wir der Anteil der richtig positiven dem der falsch positiven gegenübergestellt um auf Basis der gesamten Beispieldaten eine Aussage darüber zu treffen, wie das Verhältnis, auf Basis der absoluten Zahlen, zwischen den falsch positiven Hinweisen zu richtig positiven ist. Ziel dieses Vergleiches ist es, den, unter Umständen aus Sicht nicht eintretender, zukünftiger Fehlerwirkungen durch nicht benötigten unnötigen Fehlerkorrekturaufwand, in ein Verhältnis zu den zielführenden Korrekturen zu stellen. Moser et. al. [27] verwenden hierzu eine Gewichtung mit dem Faktor fünf, siehe Kapitel 3 Im Zuge der Bewertung der Ergebnisse dieser Fallstudie soll vorab keine einseitige Gewichtung dieser beiden Werte erfolgen. Zusätzlich zur Verhältniszahl sollen die für deren Berechnung relevanten Daten dokumentiert werden, um im Nachhinein weiterführende Gewichtungen vornehmen zu können und somit das Nutzenverhältnis einzelner Modelle neu zu bewerten. In Abbildung 25 Nutzenverhältnis optimierter Modelle sind die beiden Modell-Varianten mit den besten Bewertungen bzgl. Nutzenverhältnis dargestellt. Das Nutzenverhältnis gibt den Anteil der aus Sicht zur Verhinderung zukünftiger Fehlerwirkungen nicht gerechtfertigten Aufwandes für Entwickler*innen an, also wie viele Hinweise aus einer von einem Vorhersage-Modell generierten Liste unter

Umständen die Wartbarkeit des Quellcodes verbessern, aber keine Fehlerkorrekturaufwände einsparen. Je geringer das Verhältnis, desto höher die Effizienz von eingesetzten Entwickler*innen-Ressourcen zur Abarbeitung von Hinweisen der Vorhersage.

LibLinearTrainer (#62, Lernvektor-Variante: 1)

Defects			Falschmeldungen			Nutzenverhältnis
Daten	richtig positiv		Daten	falsch positiv		
20	**5**	**25,00%**	1448	**63**	**4,35%**	**12,6**

KernelSVMTrainer (#51, Lernvektor-Variante: 4)

Defects			Falschmeldungen			Nutzenverhältnis
Daten	richtig positiv		Daten	falsch positiv		
20	**4**	**20,00%**	1448	**26**	**1,80%**	**6,5**

Abbildung 25 Nutzenverhältnis optimierter Modelle

Eine Variante des LibLinearTrainer erkannte in Summe fünf Defects korrekt, wies jedoch zusätzlich 63 Falschmeldungen ebenfalls als Defects aus. Es wurden somit 25% der Defects korrekt erkannt (*recall*) und die Fehlerquote von falsch positiven Werten lag bei 4,35%. Von den insgesamt 68 als Defects Hinweisen der Vorhersage waren 7,35% echte Defects (*precision*), somit würde die Korrektur einer Liste mit 100 Hinweisen ca. sieben zukünftige Fehlerwirkungen erkennen und deren Korrekturen vorab im Entwicklungszyklus ermöglichen. Das Nutzenverhältnis liegt bei dieser Modell-Variante bei 12,6. Ein besseres Nutzenverhältnis wies eine Variante des KernelSVMTrainer auf. Hier wurden zwar mit vier Defects weniger richtig positive gefunden, aber mit 26 als Defects ausgewiesenen Falschmeldungen war diese Fehlerquote deutlich geringer. Diese lag in diesem Modell bei lediglich 1,8%. Von den insgesamt 30 als Defects erkannten Daten waren 13,3% echte Defects (*precision*) und das Nutzenverhältnis liegt in diesem Modell bei 6,5. Mit einer um fünf Prozentpunkte höheren Erkennungsquote von echten Defects, in diesem Fall genau ein Defect, steigt bei dem Modell auf Basis des LibLinearTrainers die Quote der falsch positiven um mehr als 100% von 26 auf 63. Während die erste Variante auf linearer Basis das Modell trainiert, wird für die Variante des KernelSVMTrainer ein Polynomial-Kernel verwendet. Die konträre Herangehensweise der Trainer ist vor allem auf die Unterschiedlichkeit der genutzten Merkmalsausprägungen der Lernvektoren zurückzuführen. Beide Lernvektoren basieren auf den sieben grundlegenden Merkmalsausprägungen. Die für das LibLinearTrainer-Modell eingesetzt Lernverktor-Variante 1 ergänzt diese sieben Ausprägungen um das Merkmal der Programmiersprache. Die für das KernelSVMTrainer-Modell verwendete Lernvektor-Variante 4 nutzt das Merkmal der Programmiersprache nicht, fügt dem Lernvektor aber vier andere Merkmalsausprägungen hinzu: Den exponentiell gleitenden Durchschnitt (EMA) als auch die Steigung der linearen Funktion der Zeitreihen (LRSlope)

für jeweils den CorrectnessIndex und den CorrectnessValue. Trotz deutlicher Überlegenheit des KernelSVMTrainer-Modells auf Basis des Nutzungsverhältnisses kommt auch dem ersten Modell aufgrund der weitaus geringeren Aufwände für die Datenerfassung im Vorfeld der Erstellung des Modells zu. Für die Betrachtung der Historie von Ressourcen und die Berechnung der dazu notwendigen Segmente als Grundlage für die Erstellung der Metriken in Bezug auf Correctness ist eine vollständige Analyse und Zuordnung von Fehlerkorrekturen zu den Ressourcen notwendig. Hier muss im Zuge der Berechnung die gesamte Historie von Fehlerkorrekturen mitbetrachtet werden. Können diese Prozesse nicht oder nur teilweise automatisiert werden, fallen hierfür zusätzliche Aufwände an, welche für eine Aktualisierung des Modells auf Basis des LibLinearTrainers nicht notwendig sind. Somit lassen sich für diesen Trainer bei Bedarf Modelle für bestimmte Repositories weitaus ressourcensparender berechnen und anwenden.

Ergebnis dieser dritten Stufe ist in erster Linie ein minimales Set von sieben Merkmalsausprägungen, welches die Erzeugung von ersten Modellen ermöglicht, die vor allem besser als der Zufall agieren. Hiermit konnte gezeigt werden, dass eine Korrelation der erhobenen Rohdaten sowie berechneten Software-Metriken auf Basis dieser Ausprägungen aufgezeigt werden kann. Durch weiterführende Anpassungen des Lernvektors als auch Parameter-Optimierungen der eingesetzten Trainer-Implementierungen wurden zwei Modelle mit einem ausbalancierten Verhältnis von korrekt erkannten Defects sowie irrtümlich als Defects identifizierten Falschmeldungen erstellt. Diese zeigten ein Nutzenverhältnis von 6,5 und 12,6, welches das Verhältnis von richtig positiven zu falsch positiven beschreibt. Für das Modell mit besserem Nutzenverhältnis kamen elf Ausprägungen im Lernvektor zum Einsatz, für das Modell mit einer schlechteren Quote kamen lediglich acht Ausprägungen zum Einsatz. Letzteres kann daher wesentlich ressourceneffizienter trainiert oder in Folge aktuell gehalten werden.

Erkenntnisse dieser Stufe sind, dass vor allem Änderungsmaßzahlen einen sehr deutlichen Mehrwert für die Korrelation lieferten. Hier besonders jene Varianten, welche jüngere Änderungen am Quellcode mit einer höheren Gewichtung berücksichtigen. Weiters zeigte sich, dass die Anzahl an Zeilen Quellcode als Vertreter der traditionellen Metriken essenziell ist. Aufgrund des unausgeglichenen Verhältnisses der Beispieldaten der unterschiedlichen Kategorien war eine Bewertung der Modelle auf Basis der Accuracy nicht zielführend. Hierfür wurde auf die beiden Werte für Precision und Recall gesetzt, um nicht einseitig zu optimieren und ein Gleichgewicht aus nicht gefundenen, echten Defects und als Defects vorhergesagten Falschmeldungen zu finden. Darüber hinaus wurde eine weitere Maßzahl für die finale Bewertung der Modelle erstellt. Das Nutzenverhältnis errechnet die Quote von richtig positiven zu falsch positiven Vorhersagen. Das Nutzenverhältnis ist ein Indikator für den zusätzlich notwendigen Mehraufwand für die

Bearbeitung der Vorhersage-Ergebnisse, um jene Hinweise zu korrigieren, welche aber in Folge zu keiner Fehlerwirkungen führen würden.

4.5. Diskussion der Ergebnisse

In Kapitel 4.1 wurden die relevanten Fragestellungen für diese Fallstudie definiert, welche in weiterer Folge auf Grundlage der erzielten Ergebnisse der einzelnen Stufen beantwortet werden.

- *F1: Gibt es Gemeinsamkeiten von Hinweisen statischer Analyse für welche Fehlerwirkungen nachgewiesen und Defects gemeldet wurden?*

Alle Meldungen im Fehlermanagement, für die im Zuge der Korrekturen Hinweise in den Berichten statischer Analyse in der fehlerhaften Basisversion nachgewiesen werden konnten, wurden in dieser Fallstudie als relevante Defects markiert und im Detail analysiert. Der Anteil dieser relevanten Defects an der Gesamtmenge betrachteter Defects betrug 10,5% bzw. 20 von 190 Defects. 95% bzw. 19 dieser 20 relevanten Defects wurden aufgrund von Regelverletzungen markiert, welche ausschließlich der Kategorie Komplexität zugeordnet sind. Das Überschreiten von vordefinierten Komplexitätsmaßen stellte jenes Merkmal dar, welche die meisten Defects zusammenfasste, wobei aufgrund dieser Kategorisierung allein kein Rückschluss auf zukünftige Fehlerwirkungen möglich war, da eine deutlich größere Anzahl von Hinweisen statischer Analyse dieser Kategorie zugeordnet wurde, als die der relevanten Defects. 90% bzw. 18 von 20 der relevanten Defects wurden aufgrund hierarchischer Betrachtung durch Regelverletzungen auf Methodenebene hervorgehoben, lediglich ein einziger Defect wurde nicht durch einen Hinweis bzgl. kognitiver Komplexität markiert, siehe Abbildung 26 Gruppierung der Hinweise relevanter Defects

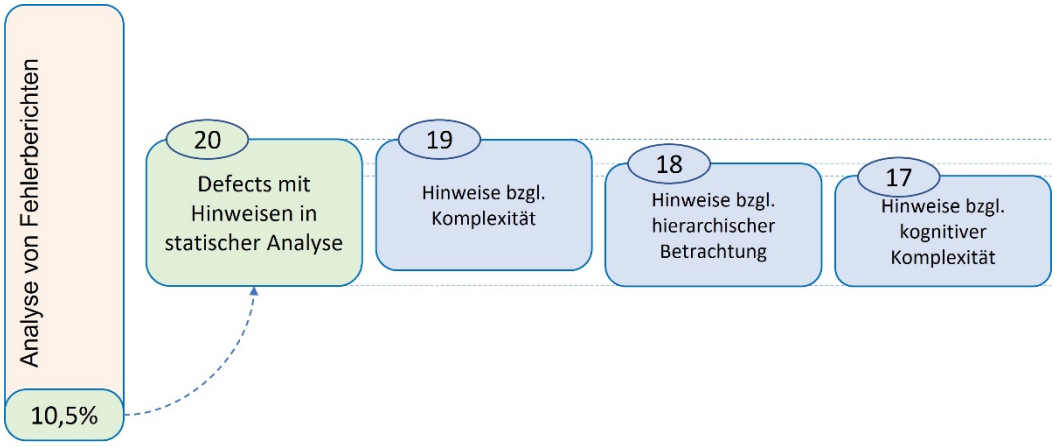


Abbildung 26 Gruppierung der Hinweise relevanter Defects

Jene Hinweise statischer Analyse, welche in Folge zu Fehlerwirkungen führen, zeigen nachweislich Gemeinsamkeiten in deren Merkmalsausprägungen auf. Diese Gemeinsamkeiten ermöglichen es allerdings nicht, dass allein darauf aufbauend eine Klassifizierung von zukünftigen Fehlerwirkung erfolgen kann, da ebenfalls eine sehr hohe Anzahl von Hinweisen, die in der späteren Betrachtung nicht zu Fehlerwirkungen führen, ebenso diese Merkmalsausprägungen aufweisen. Alle Hinweise von relevanten Defects waren qualitative Schwachstellen und somit vom Typ *Code Smell*. Drei Defects waren mit dem Schweregrad *Major* ausgezeichnet, die restlichen mit der Einstufung *Critical* versehen. Auch auf Basis dieser Merkmalsausprägungen war eine Korrelation aufgrund der hohen Anzahl von Hinweisen mit identen Merkmalen nicht möglich. Gemeinsamkeiten relevanter Defects in Berichten statischer Analyse sind durchaus gegeben, allein auf Basis von diesen Daten sind jedoch keine aussagekräftigen Vorhersage-Modelle zu erwarten.

- *F2: Wie können, ausgehend von relevanten Hinweisen statischer Analysen, praxistaugliche Modelle zur Vorhersage von Fehlerwirkungen generiert werden?*

Durch die Beantwortung der ersten Fragestellung geht hervor, dass eine Korrelation lediglich auf Basis der Hinweise statischer Analyse nicht zielführend ist. Für das Training praxistauglicher Modelle sind weitere entscheidungsrelevante Daten notwendig. Hierzu wurden Prozess-Metriken, vor allem änderungsbezogene Maßzahlen auf Basis der Ressourcen, eingesetzt. Durch Kombination unterschiedlicher Merkmalsausprägungen der Ressourcen konnten unterschiedliche Lernvektoren gebildet werden, mit deren Einsatz eine Vorhersage-Genauigkeit deutlich über dem Zufall erzielt werden konnte. Durch Anwendung unterschiedlicher Trainer-Implementierungen und optimierter Parametrisierungen konnte die Balance der Modelle zwischen korrekt identifizierten echten Defects und fälschlicherweise als Defect vorhergesagte Falschmeldungen verbessert werden. Diese Quote des Nutzenverhältnisses wurde zur Messung der Effektivität der generierten Modelle eingesetzt. Im Zuge der Rohdatenanalyse wurde hierfür für jeden Defect zusätzlich die vom Entwicklungsteam zur Behebung verbuchte Zeit erhoben. Ebenso wurde für jeden relevanten Defect der für die Korrektur des Hinweises statischer Analyse voraussichtliche Aufwand (*estimated effort*) in SonarQube dokumentiert. Das Verhältnis des geschätzten Aufwandes zur Behebung der Hinweise gegenüber der verbuchten Behebungszeit betrug 1 : 21,93. Wenn somit der prognostizierte Aufwand zur Behebung eines Hinweises in SonarQube drei Minuten betrug, dann stand dem gegenüber eine Behebungszeit des Entwicklungsteams, rein zur Korrektur der gemeldeten Fehlerwirkung von ca. einer Stunde. Demzufolge wäre ein Nutzenverhältnis eines Vorhersage-Modells dann vorteilhaft, wenn dieses unter dem Faktor 21,93 liegt. Je niedriger das Nutzenverhältnis, desto höher die Effektivität des eingesetzten Modells. Dies ist dadurch begründet, dass weniger Ressourcen zur

Bearbeitung derjenigen Hinweise eingesetzt werden, welche in weiterer Folge zu keinen Fehlerwirkungen führen würden. Die besten Modelle erreichten die Werte 12,6 und 6,5 für das Nutzenverhältnis. Diese können somit praxistauglich verwendet werden, da der eingesetzte Aufwand des Entwicklungsteams eine positive Investition darstellt, die sich durch höhere Ressourcen-Einsparungen aufgrund zukünftig entfallender Korrekturkosten bemerkbar macht. Die Modelle erheben jedoch keinen Anspruch darauf, alle zukünftigen Fehlerwirkungen vorhersagen zu können.

Die beiden favorisierten Modelle wurden in einer Anwendung integriert, welche auf Basis eines definierten Versionstandes (*Commit-ID*) eines Repositories und einer darauf ausgeführten statischen Analyse in SonarQube Daten abruft, Software-Metriken berechnet und eine Vorhersage bzgl. zukünftiger Feherwirkungen trifft. Abbildung 27 Vorhersage-Bericht zeigt eine Durchführung der Vorhersage für ein Repository mit 122758 Zeilen Quellcode und 1219 identifizierten Regelverletzungen in der statischen Analyse, welche als Grundlage für die weitere Berechnungen herangezogen wurden. Das Modell auf Basis LibLinearTrainer lieferte drei und das Modell auf Basis KernelSVMTrainer fünf Vorhersagen zu zukünftigen Fehlerwirkungen. Die Ergebnisse der Vorhersage werden als direkter Verweis zu den Hinweisen in SonarQube ausgegeben.

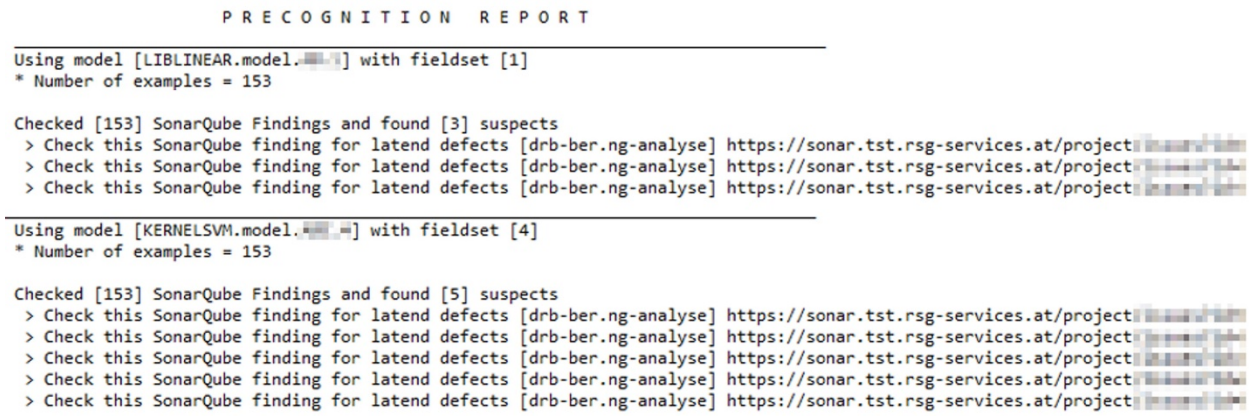


Abbildung 27 Vorhersage-Bericht

Interessant ist vor allem, dass alle drei Vorhersagen des LibLinearTrainer-Modells ebenfalls in dem Modell auf Basis KernelSVMTrainer vorhergesagt werden. Eine Prüfung der Korrektheit dieser Vorhersagen konnte zum Zeitpunkt des Abschlusses dieser Arbeit noch nicht durchgeführt werden.

5. Zusammenfassung und Ausblick

Das Ziel dieser Arbeit ist es, die Eignung der statischen Analyse als Basis für die Generierung von praxistauglichen Vorhersage-Modellen im Zuge einer Fallstudie aufzuzeigen. Dabei werden Fehlerberichte von bereits eingesetzten Software-Produkten und gefundenen Schwachstellen aus statischer Analyse gegenübergestellt und Hinweise auf eine Korrelation der Datenbestände gesucht. In weiterer Folge werden auf Basis der gefundenen Zusammenhänge und zusätzlich berechneter Software-Metriken Modelle zur Vorhersage von zukünftigen Fehlerwirkungen trainiert. Diese Modelle sollen für die praxistaugliche Anwendung auf bisher unbekannte Software-Projekte zur Verfügung gestellt werden.

Die Bearbeitung dieser Fallstudie erfolgte in drei Stufen. Diese wurden sequenziell bearbeitet, die Ergebnisse jeweils qualitätsgesichert und anschließend bewertet.

In der ersten Stufe wurde auf Basis ausgewählter Software-Produkte und zugehöriger Fehlerberichte über einen Zeitraum von zwölf Monaten eine Rohdatenbasis für Defects aufgebaut. Schwerpunkt lag hierbei vor allem auf der qualitativen Betrachtung durchgeführter Fehlerkorrekturen. Änderungen am Quellcode wurden daraufhin untersucht, ob Hinweise aus statischer Analyse an jenen Stellen im Quellcode in der fehlerhaften Basisversion vorhanden waren, welche einen inhaltlichen Zusammenhang mit der Korrektur der gemeldeten Fehlerwirkung aufwiesen. War dies der Fall so erfolgte die Kennzeichnung als relevanter Defect und es wurden weiterführende Detaildaten erhoben. War dieser Umstand nicht gegeben, so wurden allgemeine Daten zu Defect und Repository gesammelt und in der Datenbasis hinterlegt. Von 190 gemeldeten Defects wurden im Zuge der qualitativen Datenanalyse 20 als relevante Defects erkannt bzw. eine Quote von 10,5% erreicht, siehe Abbildung 28 Übersicht Training von Vorhersage-Modelle. 19 der 20 relevanten Defects wurden aufgrund von Regelverletzungen markiert, welche der Kategorie Komplexität zugeordnet sind. Hier wurden vorab definierte Schwellwerte überschritten. Den größten Anteil nahmen die Regeln bzgl. kognitiver Komplexität, sowohl für Java als auch TypeScript, ein. Eine Stärke der Berechnungsmethode dieser Maßzahl ist vor allem die Anwendung auf hierarchischer Ebene. 18 Defects wurden aufgrund hierarchischer Betrachtung durch Regelverletzungen auf Methodenebene als relevante Defects markiert, 17 davon aufgrund Verletzungen der Schwellwerte der Prüfregel bzgl. kognitiver Komplexität.

In der zweiten Stufe wurde die Rohdatenbasis um berechnete Software-Metriken erweitert. Hierzu wurden Daten aus dem Fehlermanagement Jira, der Versionsverwaltung Git und bzgl. der statischen Analyse aus SonarQube automatisiert aufbereitet. Schwerpunkt wurde auf Prozess-Metriken, im Besonderen auf Änderungsmaßzahlen der

Ressourcen, gelegt. Zusätzlich wurde die Historie in Segmente unterteilt, welche dem Release-Zyklus der Entwicklungsteams angepasst wurden, um darauf aufbauend unterschiedliche Gewichtungen zur Berücksichtigung von Trends in der Häufigkeit der Änderungen von Ressourcen in der Versionsverwaltung abzubilden. Zusätzlich zu relevanten Defects wurden auch Falschmeldungen (*false positives*) erhoben. Diese Hinweise aus statischer Analyse, welche aufgrund der identifizierten Prüfregeln relevanter Defects aufgezeigt wurden, aber zu keiner dokumentierten Fehlerwirkung (im Beobachtungszeitraum) führten, stellten den mengenmäßig umfassenderen Teil dar. Die Rohdatenbasis wurde sowohl für relevante Defects als auch für alle Falschmeldungen um die definierten Software-Metriken ergänzt.

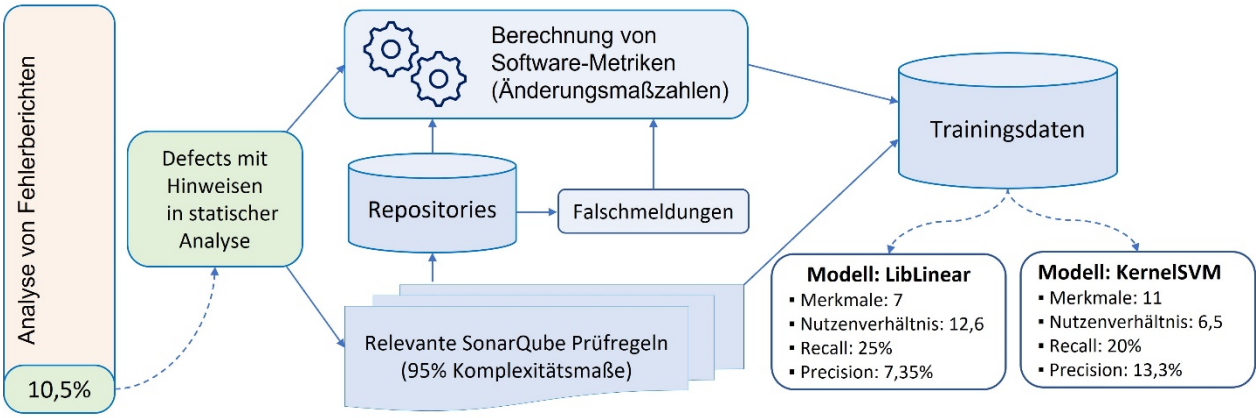


Abbildung 28 Übersicht Training von Vorhersage-Modellen

Die dritte Stufe baute auf die aus Stufe 2 erweiterte Rohdatenbasis auf, um belastbare Kombinationen von Merkmalsausprägungen für aussagekräftige Lernvektoren zu suchen. Für alle Beispieldaten wurde der Zielwert, welcher angibt, ob es sich bei einem Lernvektor als Summe aller Merkmalsausprägungen um eine beobachtete Fehlerwirkung handelt oder nicht, ergänzt und als Trainingsdaten zur Verfügung gestellt (siehe Abbildung 28 Übersicht Training von Vorhersage-Modelle). Auf einem minimalen Set von sieben vitalen Merkmalsausprägungen konnte bereits eine Korrelation zum jeweiligen Zielwert, welche deutlich über einer Quote von zufälligen Vorhersagen lag, nachgewiesen werden. Neben sechs Prozess-Metriken war vor allem die Maßzahl Anzahl Zeilen Quellcode (*LOC*) unerlässlich für das Training von Vorhersage-Modellen. Im Zuge von Optimierungen zeigten speziell die beiden in Abbildung 28 Übersicht Training von Vorhersage-Modelle dargestellten Trainer-Implementierungen LibLinear und KernelSVM praxistaugliche Ergebnisse. Zur Bewertung dieser Modelle wurde primär das Nutzenverhältnis herangezogen, welches die Quote von Vorhersagen für echte Defects zu Hinweisen bzgl. Falschmeldungen beschreibt. Hiermit wird der Anteil von ungerechtfertigtem Aufwand zur Korrektur von Falschmeldungen aufgezeigt. Je geringer das Nutzenverhältnis, desto höher die Effizienz des Modells. Ein Modell auf Basis LibLinearTrainer und einem

Nutzenverhältnis von 12,6 konnte mit vergleichsweise weniger Daten, einfach trainiert werden und lieferte eine Precision von 7,35%. Ein zweites Modell auf Basis KernelSVMTrainer mit polynomialen Kernel nutzte ein umfangreicheres Set von elf Merkmalsausprägungen und zeigte ein Nutzenverhältnis von 6,5 bzw. eine Precision von 13,3%.

In weiterer Folge können die erzeugten Modelle auf bisher unbekannte Software-Produkte bzw. neue Versionen angewendet werden. Hierzu werden, initiiert durch Berichte statischer Analyse, die Hinweise relevanter Prüfregeln extrahiert. Für alle Hinweise bzw. deren zugeordneten Ressourcen werden Software-Metriken auf Grundlage der Historien des Repositories zur Bereitstellung der notwendigen Merkmalsausprägungen berechnet. Die resultierenden Vektoren werden im Modell verwendet, um für jeden Hinweise aus statischer Analyse eine Vorhersage zu treffen, ob es sich um eine prognostizierte Fehlerwirkung handelt oder nicht, siehe Abbildung 29 Übersicht Anwendung eines Vorhersage-Modells

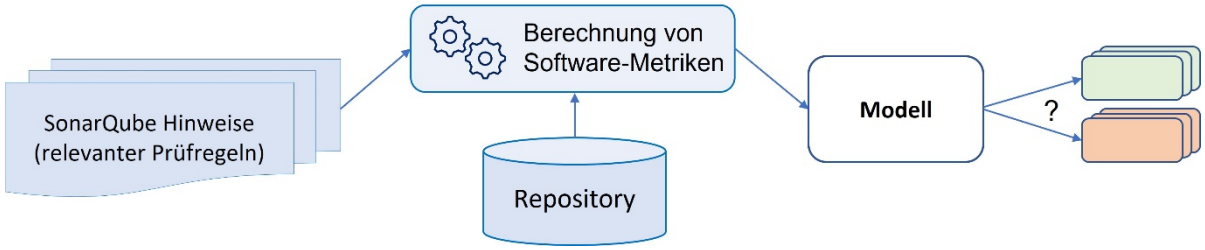


Abbildung 29 Übersicht Anwendung eines Vorhersage-Modells

Für alle Defects mit Hinweisen in statischer Analyse wurde sowohl der in SonarQube angegebene, voraussichtliche Aufwand zur Behebung des Hinweises zur Entwicklungszeit (*estimated effort*) als auch die vom Entwicklungsteam benötigte Zeit zur Fehlerbehebung erhoben. Das Verhältnis des geschätzten Aufwandes zur Behebung der Hinweise gegenüber der verbuchten Behebungszeit (inkl. Tätigkeiten für Release und Rollout der neuen Version) betrug 1 : 21,93. Beide ermittelten Modelle weisen ein deutlich besseres Nutzenverhältnis auf und können auf Basis der Ergebnisse praxistauglich Verwendung finden. Der eingesetzte Aufwand kann als positive Investition betrachtet werden, da die anfallenden Aufwände zur Behebung von Falschmeldungen den Einsparungen bzgl. zukünftiger Fehlerwirkungen positiv gegenüberstehen.

Die festgelegte Ebene für Analysen und Vorhersagen in dieser Fallstudie ist die Ressource, somit basieren Analysen und Vorhersagen auf einzelnen Dateien. Dies wurde bewusst definiert, da Daten zu Änderungen im Versionsmanagement auf dieser Ebene sehr effizient auszuwerten sind. Die statische Analyse bietet ebenfalls Maßzahlen auf Ebene der Ressourcen an, bietet aber in der Regel Hinweise auch deutlich granularer auf Ebene von Methoden oder einzelnen Zeilen an. Mögliche Verbesserung der Vorhersage-

Qualität auf diesen Ebenen wäre den aktuellen Ergebnissen gegenüberzustellen. Vor allem der Vergleich der gebuchten Korrekturkosten der Entwicklungsteams zu den prognostizierten Aufwänden in SonarQube wäre exakt möglich, da keine Vermischung von Datei vs. Methode bzw. Datei vs. Zeile im Quellcode auftreten kann. Weiters bietet eine feinere Granularität wertvolle Unterstützung bei der Analyse der Ressourcen, vor allem bei sehr umfangreichen Dateien oder komplizierten Inhalten.

Die Quote von 10,5% an relevanten Defects aller Meldungen im Beobachtungszeitraum spiegelt einen verhältnismäßig geringen Wert wider. Da eine Markierung mit einem Hinweis statischer Analyse einerseits auslösende Bedingung für die Berücksichtigung der Detaildaten als relevanter Defect für das weiterführenden Training der Modelle, andererseits aber auch relevant für die Suche nach Falschmeldungen für die Vorhersage ist, wäre eine Erweiterung der eingesetzten Regelkataloge vorteilhaft. Versuche, diese Regelkataloge mengenmäßig stark zu erhöhen, führten regelrecht zu einem Überlauf der Berichte in SonarQube und hätten in weiterer Folge auch keine weitere Aussagekraft im Zuge der Klassifikation. Hierfür wäre ein methodischer Ansatz zu wählen, um schrittweise einzelne Prüfregeln bzw. Regelkategorien zu aktivieren und deren Auswirkung unmittelbar sowohl auf die Genauigkeit der damit erzeugten Vorhersage-Modelle als auch auf die Menge unbekannter Hinweise, welche für Vorhersage aus SonarQube geladen werden, zu bewerten.

Zusätzlich könnte das auslösende Momente zur Markierung relevanter Defects neben den Hinweisen statischer Analyse um die gemessene Unittest-Abdeckung im Zuge der statischen Analyse erweitert werden, welche in SonarQube überwacht und auf Codezeilen-Ebene angegeben wird. Einerseits könnten Hinweise statischer Analyse durch eine fehlende Testabdeckung verstärkt bzw. durch das Vorhandensein von Test abgeschwächt werden.

Weiters kann die Kombination bestehender oder zusätzlicher Vorhersage-Modelle zur Erhöhung der Genauigkeit der Vorhersage eingesetzt werden (*ensemble learning*). Mehrere im Zuge der Fallstudie eingesetzte Trainer-Implementierung benutzen bereits derartige Ensemblemethoden, jedoch können diese generierten Modelle auf einer übergeordneten Programmebene bewertet und kombiniert werden, bspw. um die in Abbildung 27 Vorhersage-Bericht gezeigten drei Hinweise, welche in jedem Modell vorhergesagt werden, entsprechend zu gewichten.

Abschließend wird die Erweiterung der Datenbasis um neue Lerndaten und in Folge die zeitliche Weiterführung der Datenanalyse wertvoll gesehen. Vor allem neue Repositories, deren Komponenten dieselben Entwicklungsmodelle und Programmiersprachen einsetzen, können weitere entscheidungsrelevante Daten zur Berechnung besserer Vorhersage-Modelle bringen.

6. Abbildungsverzeichnis

Abbildung 1 Ökosystem....3
Abbildung 2 Methodik....4
Abbildung 3 Prozessaufbau in Anlehnung an [17]....8
Abbildung 4 SonarQube: Typen von Hinweisen und Kategorisierung....11
Abbildung 5 Klassifikation von Metriken, in Anlehnung an [17]....13
Abbildung 6 Training von Vorhersage-Modellen, in Anlehnung an [23]....15
Abbildung 7 Klassifikation von maschinellem Lernen, in Anlehnung an [24]....16
Abbildung 8 Klassifikationsmodelle [23, p. 182ff]....17
Abbildung 9 2x2 Wahrheitsmatrix, in Anlehnung an [25]....18
Abbildung 10 Vorgehensmodell für die Fallstudie....25
Abbildung 11 Übersicht der Applikationen....28
Abbildung 12 Änderungen am Quellcode....29
Abbildung 13 Nachverfolgbarkeit von Änderungen in Git....30
Abbildung 14 Aufruf statischer Analyse über SonarScanner in Maven....31
Abbildung 15 Inhaltliche Analyse von Fehlerkorrekturen....31
Abbildung 16 SonarQube-Analyse der fehlerhaften Basisversion ohne Relevanz....32
Abbildung 17 Defect-Statistik....33
Abbildung 18 Struktur der Klasse CommitList....36
Abbildung 19 Calculator-Klassen zur Berechnung von Segmenten....38
Abbildung 20 Statistische Bewertungen von Segmenten....39
Abbildung 21 Suche nach relevanten Falschmeldungen....40
Abbildung 22 Modell-Training....43
Abbildung 23 Abgegrenzte Teilmenge für Kreuzvalidierung....44
Abbildung 24 Kombination von Trainer, Beispieldaten und Lernvektoren....45
Abbildung 25 Nutzenverhältnis optimierter Modelle....46
Abbildung 26 Gruppierung der Hinweise relevanter Defects....48
Abbildung 27 Vorhersage-Bericht....50

Abbildung 28 Übersicht Training von Vorhersage-Modellen.....52
Abbildung 29 Übersicht Anwendung eines Vorhersage-Modells.....53

7. Literaturverzeichnis

[1] W. Cunningham, „The WyCash Portfolio Management System," in *Conference on Object-Oriented Programming Systems, Languages, and Applications*, Vancouver, British Columbia, 1992.

[2] S. McConnell, „Construx," 6 2008. [Online]. Available: https://www.construx.com/resources/whitepaper-managing-technical-debt/. [Zugriff am 8 1 2023].

[3] Z. Codabux und B. Wiliams, „Technical debt prioritization using predictive analytics," *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C),* 2016.

[4] A. Spillner und T. Linz, Basiswissen Softwaretest, dpunkt, 2012.

[5] G. Bath und J. McKay, Praxiswissen Softwaretest - Test Analyst und Technical Test Analyst, dpunkt, 2011.

[6] S. Kim und E. Michael D., „Which warnings should I fix first," in *ESEC-FSE '07*, 2007.

[7] Q. Hanam, L. Tan, R. Holmes und P. Lam, „Finding patterns in static analysis alerts: improving actionable alert ranking," *IEEE Working Conference on Mining Software Repositories,* 2014.

[8] R. S. GmbH, „Raiffeisen Software GmbH," [Online]. Available: https://www.r-software.at/. [Zugriff am 8 1 2023].

[9] Sonar, „Sonar," [Online]. Available: https://www.sonarsource.com/. [Zugriff am 8 1 2023].

[10] Git, „Git Project," [Online]. Available: https://git-scm.com/. [Zugriff am 8 1 2023].

[11] Atlassian, „Atlassian Jira," Atlassian, [Online]. Available: https://www.atlassian.com/software/jira. [Zugriff am 8 1 2023].

[12] RedHat, „What is CI/CD?," RedHat, [Online]. Available: https://www.redhat.com/en/topics/devops/what-is-ci-cd. [Zugriff am 10 1 2023].

[13] A. Spillner, T. Roßner, M. Winter und T. Linz, Praxiswissen Softwaretest - Testmanagement, dpunkt, 2011.

[14] C. Boogerd und L. Moonen, „Prioritizing Software Inspection Results using Static Profiling," in *Sixth IEEE International Workshop on Source Code Analysis and Manipulation*, 2006.

[15] R. Malhotra und P. Singh, „Exploiting bad-smells and object-oriented characteristics to prioritize classes for refactoring," in *International Journal of Systems Assurance Engineering and Management*, 2020.

[16] N. Sae-Lim, S. Hayashi und M. Saeki, „Context-based approach to prioritize code smells for prefactoring," in *Journal of Software: Evolution and Process*, 2018.

[17] S. Rathore und S. Kumar, „A study on software fault prediction techniques," *Artificial Intelligence Review,* Bd. 51, pp. 255-327, 15 2 2019.

[18] W. Ma, L. Chen, Y. Zhou und B. Xu, „Do We Have a Chance to Fix Bugs When Refactoring Code Smells," in *International Conference on Software Analysis, Testing and Evolution*, 2016.

[19] E. A. Alikhashashneh, R. R. Raje und J. H. Hill, „Using Machine Learning Techniques to Classify and Predict Static Code Analysis Tool Warnings," in *ACS/IEEE International Conference on Computer Systems and Applications*, 2018.

[20] L. J. Heinrich, A. Heinzl und F. Roithmayr, Wirtschaftsinformatik-Lexikon 7.Auflage, München: Oldenbourg, 2004.

[21] D. Hoffmann, Software-Qualität, Springer, 2012.

[22] I. S. 1061-1992, „IEEE Standard for a Software Quality Metrics Methodology," 1993.

[23] S. Papp, W. Weidinger, M. Meir-Huber, B. Ortner, G. Langs und R. Wazir, HANDBUCH DATA SCIENCE - Mit Datenanalyse und Machine Learning Wert aus Daten generieren, Hanser, 2019.

[24] MathWorks, „Machine Learning in MATLAB,“ MathWorks, [Online]. Available: https://www.mathworks.com/help/stats/machine-learning-in-matlab.html. [Zugriff am 20 1 2023].

[25] Google Developers, „Machine Learning,“ Google, 18 11 2022. [Online]. Available: https://developers.google.com/machine-learning/crash-course/classification/video-lecture?hl=de. [Zugriff am 26 1 2023].

[26] K. O. Elish und M. O. Elish, „Predicting defect-prone software modules using support vector machines,“ in *Journal of Systems and Software*, 2008.

[27] R. Moser, W. Pedrycz und G. Succi, „A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction,“ in *2008 ACM/IEEE 30th International Conference on Software Engineering*, 2008.

[28] A. R. Hevner, S. T. March, J.-S. Park, J. Park und S. Ram, „Design science in information systems research,“ in *Management Information Systems Quarterly*, 2004.

[29] T. Hall, S. Beecham, D. Bowes, D. Gray und S. Counsell, „A Systematic Literature Review on Fault Prediction Performance in Software Engineering,“ in *IEEE Transactions on Software Engineering*, 2011.

[30] M. D'Ambros, M. Lanza und R. Robbes, „An extensive comparison of bug prediction approaches,“ in *IEEE Working Conference on Mining Software Repositories*, 2010.

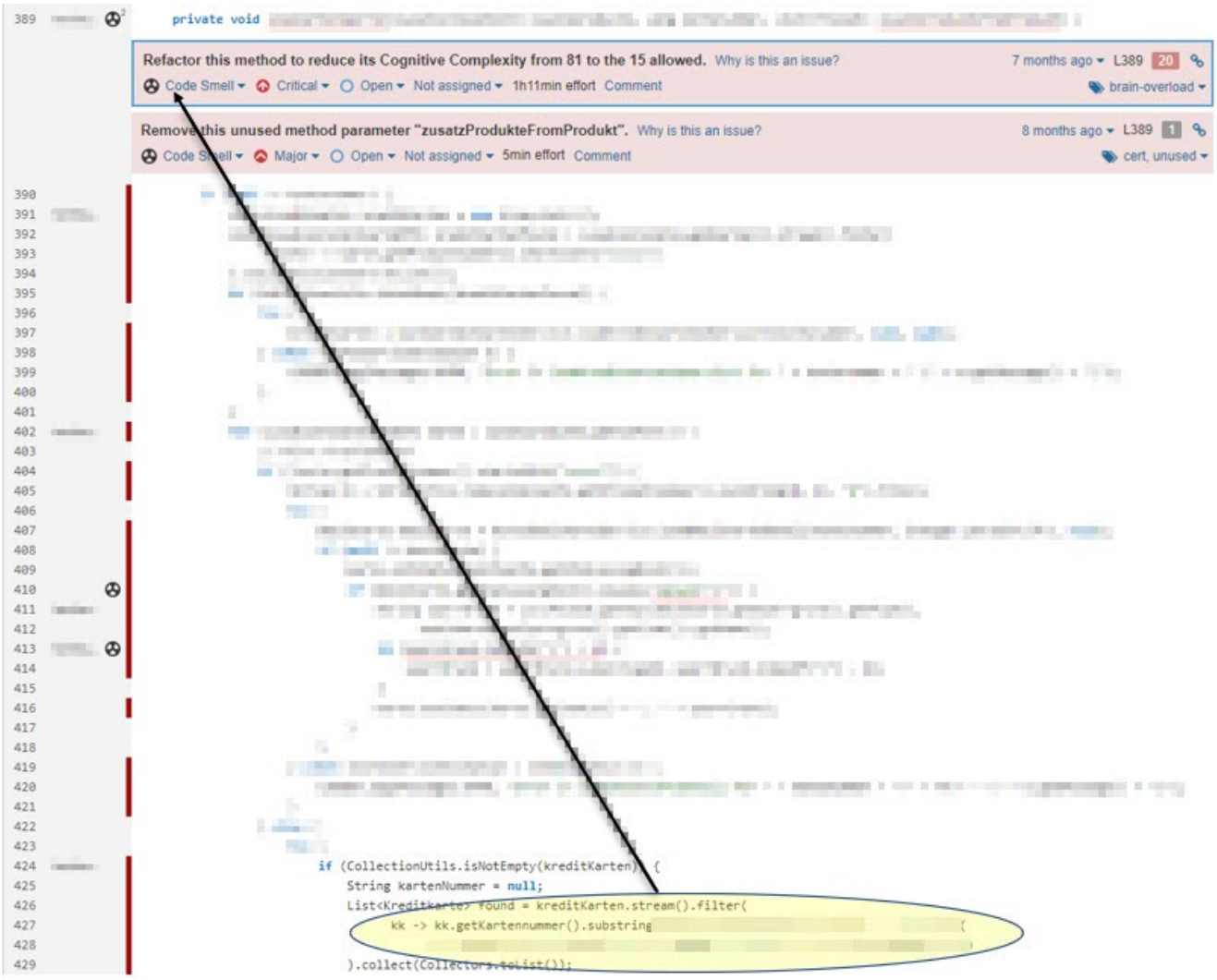
8. Anhangsverzeichnis

Anhang 1: SonarQube-Analyse der fehlerhaften Basisversion mit Relevanz.....61

Anhang 2: Auszug aus Rohdaten für Fehlerkorrekturen.....62

Anhang 3: Auszug von Metriken für Defects und relevanten Falschmeldungen.....63

Anhang 1: SonarQube-Analyse der fehlerhaften Basisversion mit Relevanz



Anhang 2: Auszug aus Rohdaten für Fehlerkorrekturen

			Git							Finding-Info RSG STD		
	Defect	Kriti	Pl	Comn	Autor	Repos	Files	Projekt	Repo	Git-Ref	Spache	Tag	Findings
	2319	M3	PR	2	1	1	1			88e82262586c778	TS	unused	PTBERAT-1
	2816	M3	PR	2	1	1	1			8e863a19b6a913e	J	brain-overload	PTBERAT-1
	2817	M3	PR	5	1	1	11			2bca819bef3f812e	TS, HTML	brain-overload	PTBERAT-1
	3040	M3	PR	3	1	1	3			901f26dbbd91ed9	TS, HTML, J	brain-overload	PTBERAT-1
	3603	M1	PR	6	1	1	6			75f8773bec21e13d	TS, HTML, J	brain-overload	PTBERAT-1
	3953	M1	PR	2	1	1	1			e337ee55c67ed30	J	brain-overload	PTBERAT-1
	3669	M3	PR	2	1	1	2			b21d18721c04c21	J	brain-overload	PTBERAT-1
	4168	M3	PR	6	1	1	1			d0fa5a70b63afdb6	J	brain-overload	PTBERAT-1
	5091	M2	PR	1	1	1	1			444ef7d978efdf104	J	brain-overload	PTBERAT-15
	4904	M3	799,	5	1	1	6			4043c05643860cc	TS	brain-overload	PTBERAT-14
	4936	M2	PR	1	1	1	1			7bb0f4438f7e508a	J	brain-overload	PTBERAT-14
	5721	M3	PR	3	1	1	1			eee217209d5c1ce	J	brain-overload	PTBERAT-1
	4592	M3	PR	2	1	1	1			e49bf75d8b76599	J	brain-overload	PTELBAIO-
	4797	M2	PR	2	1	1	2			ad94ecbb422c9bb	J	regex	PTELBAIO-
	2	M3	PR	12	1	2	3			320b3de7fe8f5cfcd	TS, HTML	brain-overload	PTBBB-420
	7	M2	PR	10	3	2	7			2e89beb159ab1ba	TS	brain-overload	PTBBB-470
	4	M2	PR	6	2	2	4			f8466afe7232e310	TS	brain-overload	PTDCB-722
	4	M2	5287	9	1	1	13			1498ddc9678f76f0	TS	brain-overload	PTDCB-721
	4772	M2	PR	7	1	1	20			4859db3d829e771	J	brain-overload	PTDCBMW
	4821	M2	PR	6	1	2	19			75ae7d8272ab8a5	TS	brain-overload	PTDCBMW

Findings RSG STD					Repo-Type			Project-Size				
File	Method	Method-Kommenta	Line	Line-Kommen	Bugs	Vul	Smells	LOC	States	Funcs	Classes	Comm%
_	_		x	typescript:S125	386	0	888	86299	22885	7110	478	2,50%
_	x	java:S3776 16	_		388	0	884	87113	23164	7181	479	2,50%
_	_	typescript:S3776 28	_		400	0	903	88504	23575	7318	480	2,50%
_	x	java:S3776 18	_		400	0	907	88859	23679	7352	480	2,50%
_	x	java:S3776 25	_		400	0	908	89269	23868	7393	481	2,50%
_	x	java:S3776 23	_		400	0	1122	94938	25710	8118	521	2,70%
_	x	java:S3776 19	_		11	0	596	22665	8981	1891	155	4,00%
_	x	java:S3776 18	_		11	0	564	22818	9087	1904	155	4,00%
_	x	java:S3776 81	_		413	0	1144	110750	26339	8297	524	2,40%
_	x	typescript:S3776 24	_		413	0	1232	111544	26579	8340	537	2,40%
_	x	java:S3776 37	_		12	0	555	22899	9188	1930	157	4,10%
_	x	java:S3776 30	_		10	0	1234	112463	26698	8374	540	2,40%
_	x	java:S107 8	_		2	0	28	3844	1175	430	40	6,50%
_	_		x	java:S5843 23	8	0	63	4779	1123	373	66	6,20%
_	x	typescript:S3776 17	_		9	2	112	9679	1225	450	97	1,80%
_	x	typescript:S3776 32	_		62	0	465	67961	12031	4309	344	3,60%
_	x	typescript:S3776 64	_		62	0	450	72524	12638	4600	356	3,50%
_	x	typescript:S3776 42	_		62	0	446	72401	12618	4590	355	3,50%
_	x	java:S3776 16	_		9	0	184	14995	3865	1175	184	8,80%
_	x	typescript:S3776 36	_		62	0	449	72545	12637	4601	354	3,50%

Anhang 3: Auszug von Metriken für Defects und relevanten Falschmeldungen

Maintainability Index (MI)				Maintainability Value (MV)					Correctness Index (CI)			
SMA	LWMA	EMA	LR Slope	SMA	LWMA	EMA	LR Slope	Total	SMA	LWMA	EMA	LR Slope
0,024616	0,041453	0,099196	0,009184	4,666667	7,782051	17,82716	1,699301	57	0	0	0	0
0	0	0	0	0	0	0	0	43	0	0	0	0
0,017334	0,031312	0,128706	0,007624	2	3,564103	14,2332	0,853147	25	0	0	0	0
0,000579	0,001068	0,00463	0,000267	0,083333	0,153846	0,666667	0,038462	44	0,002451	0,004525	0,019608	0,001131
0,015921	0,007675	0,008933	-0,0045	0,333333	0,435897	0,913592	0,055944	8	0	0	0	0
0,001007	0,001508	0,003547	0,000273	0,166667	0,25641	0,674897	0,048951	16	0	0	0	0
0,027778	0,047009	0,074074	0,01049	0,333333	0,564103	0,888889	0,125874	9	0	0	0	0
0,010013	0,016222	0,027813	0,003387	0,5	0,846154	2,222222	0,188811	12	0	0	0	0
0,012965	0,020902	0,054683	0,004329	1,75	2,846154	7,500991	0,597902	26	0,002451	0,003017	0,000242	0,000309
0,041867	0,033196	0,01522	-0,00473	7,083333	5,410256	2,197949	-0,91259	86	0,004902	0,003771	1,79E-05	-0,00062
0,003086	0,005223	0,00823	0,001166	0,083333	0,141026	0,222222	0,031469	2	0	0	0	0
0,017612	0,026331	0,043971	0,004756	2,333333	3,435897	5,158699	0,601399	33	0,004902	0,006787	0,006617	0,001028
0,090278	0,081197	0,222458	-0,00495	0,583333	0,487179	1,334327	-0,05245	178	0	0	0	0
0,013368	0,022623	0,045525	0,005048	0,5	0,833333	1,481481	0,181818	7	0	0	0	0
0,02186	0,026052	0,003509	0,002286	0,833333	1	0,1262	0,090909	68	0	0	0	0
0,008759	0,011126	0,021069	0,001291	2,083333	2,679487	6,747882	0,325175	64	0,000947	0,001748	0,007576	0,000437
0,021329	0,015521	0,020617	-0,00317	5,25	4,115385	5,592926	-0,61888	487	0	0	0	0
0,009277	0,010105	0,004349	0,000452	2,583333	2,987179	1,356342	0,22028	53	0,000947	0,001748	0,007576	0,000437
0,017201	0,026857	0,035867	0,005267	1,666667	2,705128	4,30727	0,566434	29	0	0	0	0
0,00387	0,004025	0,000512	8,46E-05	0,916667	1,025641	0,168273	0,059441	192	0	0	0	0

Correctness Value (CV)					Author Value (AV)					File attributes	
SMA	LWMA	EMA	LR Slope	Total	SMA	LWMA	EMA	LR Slope	Total	Type	LOC
0	0	0	0	0	0,75	1,230769	2,345679	0,262238	4	ts	2396
0	0	0	0	0	0	0	0	0	6	java	157
0	0	0	0	0	0,333333	0,487179	0,899863	0,083916	2	ts	645
0,083333	0,153846	0,666667	0,038462	1	0,083333	0,153846	0,666667	0,038462	6	java	161
0	0	0	0	0	0,333333	0,435897	0,913592	0,055944	2	java	482
0	0	0	0	0	0,166667	0,25641	0,674897	0,048951	2	java	512
0	0	0	0	0	0,166667	0,282051	0,444444	0,062937	3	java	340
0	0	0	0	0	0,333333	0,564103	1,481481	0,125874	3	java	152
0,083333	0,102564	0,00823	0,01049	1	0,75	1,102564	1,937205	0,192308	3	java	602
0,166667	0,128205	0,00061	-0,02098	2	1,5	1,358974	1,215527	-0,07692	4	ts	3107
0	0	0	0	0	0,083333	0,141026	0,222222	0,031469	2	java	311
0,166667	0,230769	0,224966	0,034965	2	1	1,346154	1,773239	0,188811	3	java	591
0	0	0	0	0	0,416667	0,294872	0,667626	-0,06643	5	java	898
0	0	0	0	0	0,25	0,423077	0,962963	0,094406	1	java	195
0	0	0	0	0	0,333333	0,410256	0,087791	0,041958	9	ts	271
0,083333	0,153846	0,666667	0,038462	1	1,333333	1,423077	3,041435	0,048951	15	ts	374
0	0	0	0	0	1,583333	1,371795	1,806771	-0,11538	20	ts	1418
0,083333	0,153846	0,666667	0,038462	1	1,25	1,448718	1,045713	0,108392	9	ts	379
0	0	0	0	0	0,5	0,769231	1,640604	0,146853	4	java	201
0	0	0	0	0	0,666667	0,730769	0,138094	0,034965	25	ts	213