TECHNISCHE
UNIVERSITÄT
DRESDEN

Fakultät Informatik Institut für Systemarchitektur, Professur für Rechnernetze

Diplomarbeit

Bewertung der Kodequalität in einer testgetriebenen Crowdsourcing-Plattform

Jakob Blume
Geboren am: 09.10.1991 in Halle (Saale)
Matrikelnummer: 3754765
Immatrikulationsjahr: 2011

15. Oktober 2017

Betreuer
Dr. Tenshi Hara
Dr. Iris Braun
Betreuender Hochschullehrer
Prof. Dr. rer. nat. habil. Dr. h. c. Alexander Schill

Selbstständigkeitserklärung

Hiermit versichere ich, dass ich die vorliegende Diplomarbeit mit dem Titel **„Bewertung der Kodequalität in einer testgetriebenen Crowdsourcing-Plattform“** selbstständig verfasst, keine anderen als die angegebenen Quellen und Hilfsmittel benutzt, sowie alle wörtlich oder sinngemäß übernommenen Gedanken und Zitate als solche kenntlich gemacht habe. Ich erkläre ferner, dass ich die vorliegende Arbeit an keiner anderen Stelle als Prüfungsarbeit eingereicht habe oder einreichen werde.

Jakob Blume
Dresden, 29.09.2017

Vorwort zur Arbeit

Für ein konsistentes Verständnis von auftretenden nicht standardisierten Formatierungen folgt an dieser Stelle eine Anmerkung zu den verwendeten Formatierungen und deren Bedeutung in der vorliegenden Arbeit:

kursiv	Ein Wort oder Textabschnitt ist *kursiv* formatiert, wenn Bezug zu einem Fachwort genommen wird, welches im Rahmen dieser Arbeit definiert wurde oder in einer Abbildung verwendet wird. Hinzu kommt, dass nur die erste Verwendung des Wortes in jedem Abschnirr kursiv gedruckt wird und alle weiteren normal dargestellt werden.
`monospace`	Ein Wort oder eine Wortgruppe ist in `monospace` formatiert, wenn ein Entitätenname, Komponentenname, Variablenname oder ein ähnlicher Name referenziert ist, der in Bezug zum eigenen Konzept oder eigenen Implementierung steht.
fett	Ein Wort oder Textabschnitt ist **fett** formatiert, wenn das Fettgedruckte sprachlich verhorgehoben werden soll.

Inhaltsverzeichnis

1. Einleitung

1.1. Motivation

Crowdsourcing Plattformen machen es möglich, dass sehr viele Menschen an der Lösung eines Problems parallel arbeiten. Ein Beispiel dafür ist die Suche nach der Boing-777 des Malaysia-Airlines-Fluges 370 nach ihrem Absturz im März 2014. Nachdem das Flugzeug von keinem Überwachungssystem aufgespührt werden konnte wurde eine Kampange auf der Plattform Tomnod[1] gestartet, bei der jeder Mensch mit Internetzugang bei der Suche helfen konnte, indem er Satellitenbilder meldet, die möglicherweise Trümmerteile des Flugzeugs zeigen. Trotz der Kampange konnte das Flugzeug nicht gefunden, aber dennoch eine große Menge an Bildern in kürzester Zeit ausgewertet werden. Ungefähr 1.007.750 km^2 Land und Meer wurden durch 8 Millionen Menschen ausgewertet.[2] Das Beispiel zeigt das Potential dieses Vorgehens.
Mit steigender Komplexität und Verantwortung der Aufgaben, die die sogenannten Crowd-Worker[3] übernehmen, steigt auch der Aufwand bei der Abnahme der eingereichten Lösungen. So ist es mitunter schwieriger zu bewerten, ob ein Quellcodeabschnitt die Aufgabenstellung erfüllt, als zu bewerten, ob ein Flugzeugtrümmerteil auf einem Foto zu sehen ist. Eine testgetriebene Crowdsourcingplattform setzt daher den Grad der ausgelagerten Aufgaben bei der Implementierung von Aufgaben mit vorhandenen Softwaretests. Diese können von einem Rechner ausgeführt werden und überprüfen, ob der Quellcode das tut, was die Tests verlangen. Die Tests beschränken sich dennoch oft auf die Funktion des Quellcodes und lassen die Qualität der eingereichten Lösung außen vor. Deshalb ist es notwendig, dass manuell überprüft wird ob er den Qualitätsansprüchen des Auftraggebers entspricht (Latoza u. a. 2013). Dieser Prozess ist oft sehr ressourcenintensiv und birgt großes Automatisierungspotential durch die Anwendung von intelligenten Algorithmen zur Bewertung der Codequalität.

1.2. Problemstellung

Bei der Bewertung von Codequalität werden Softwaremetriken benutzt. Ein Problem der Metriken liegt nicht unbedingt in ihrer Berechnung, sondern vielmehr in ihrer Auswertung. Oftmals

[1] `https://www.tomnod.com/` (Letzter Aufruf: 10.10.2017)
[2] `http://blog.digitalglobe.com/crowd/crowdsourcingmalaysianflightthankyou/` (Letzter Aufruf: 10.10.2017)
[3] Nutzer die auf Aufgaben auf einer Crowdsourcing-Plattform übernehmen

reicht ein Grenzwert nicht aus, um die Qualität präzise zu erfassen (**XXX**). Auch Entscheidungsbäume sind oft nur auf ein bestimmtes Projekt anwendbar und verlieren ihre Gültigkeit bei der Übertragung auf ein neues (**XXX**). Die Erstellung dieser Bäume unter Einbezug von ausreichend vielen Metriken ist dafür oft zu aufwendig. Daher werden Metriken vor allem dazu verwendet, um deren Verlauf zu betrachten und sie in Bezug zu einer bestimmten Qualitätsmetrik (z. B. Anzahl aufgetretener Fehler) zu stellen. Anhand von abgeleiteten Korrelation zwischen diesen beiden Daten kann der Entwicklungsprozess gesteuert werden.
In einer testgetriebenen Crowdsourcing-Plattform ist dieser Workflow allerdings nicht anwendbar, da die Qualität möglichst schnell nach der Abgabe analysiert sein muss, damit entsprechende Maßnahmen eingeleitet oder ausgelassen werden können. Die Betrachtung in dieser Arbeit geht von einer Plattform aus, in der auch die Bearbeitung der Aufgabe in einem Editor der Plattform durchgeführt wird. Dabei besteht die Möglichkeit mehr Informationen über den Entwicklungsprozess aufzuzeichnen, als das bei einem heterogenen Entwicklerteam, in dem viele verschiedene Werkzeuge bzw. IDEs verwendet werden, möglich wäre. Diese Daten werden heute noch nicht genügend für die Bewertung des Prozesses an sich und auch das Ergebnis benutzt.

1.3. Zielsetzung

Das Ziel dieser Arbeit ist zweigeteilt. In einem ersten Schritt soll ein System entworfen werden, welches die Aufzeichnung des Crowd-Workers bei der Lösung der Aufgabe ermöglicht. Dadurch soll auf der einen Seite eine Datengrundlage geschaffen werden, die bei der Auswertung mehr Informationen über den Prozess des Entwicklers bereitstellt. Die Plattform macht durch den geführten Prozess starke Einschränkungen in Bezug auf die Aufgabenvielfalt. So handelt es sich immer um abgegrenzte Entwicklungsaufgaben mit vorhandenen Softwaretests, die Feedback über den Aufgabenfortschritt geben. Unter Umständen wird eine Aufgabe sogar mehrfach vergeben und gelöst. Dadurch soll es am ermöglich werden, aufbauend auf den aufgezeichneten Daten, den durchgeführten Lösungsprozess und die eingereichte Lösung zu bewerten zu bewerten. Der Fokus liegt allerdings eher auf dem Sammeln der Daten und nicht auf ihrer Auswertung selbst.
Damit trotzdem ein schnelles Prototyping zur Evaluation der neu gewonnenen Daten möglich ist, soll ein System die Qualität der Abgabe bewertet umgesetzt indem herkömmliche[4] und neue[5] Metriken mit verschiedenen Qualitätsaspekten korreliert werden. Im Fokus der Arbeit steht die Umsetzung eines Systems, welches die Datenanalyse technisch möglich macht. Die Erarbeitung von besonders gut korrelierenden Metriken die präzise Analysen ermöglichen ist nicht Fokus dieser Arbeit, sondern vielmehr aufbauend mit dem umgesetzten System möglich.

1.4. Struktur dieser Arbeit

In Kapitel 2 wird ein Überblick über verwandte Arbeiten gegeben, die für diese Arbeit wichtig sind. Im Fokus steht die Vorstellung von qualitätsbezogenen Themen, da sie wegweisend für das Thema und die Ziele dieser Arbeit sind. In Kapitel 3 wird der Stand der Technik der Werkzeuge und Systeme vorgestellt, die zur Zeit eingesetzt werden, um die erläuterte Problemstellung zu lösen. Die Anforderungsanalyse schließt dieses Kapitel ab. Im Kapitel 4 wird die polylith-Plattform als Kontext dieser Arbeit beschrieben, da sich Konzept und Umsetzung

[4] klassische Softwaremetriken, die aus dem Quellcode berechnet werden
[5] Metriken, die aus den neuen aufgezeichneten Daten berechnet werden

darin einordnen. Anschließend wird in Kapitel 5 das Konzept für die Umsetzung des *Developer Monitoring Systems* und des *Quality Rating Systems* beschrieben. Diese beiden Systeme sollen die beiden Probleme der detaillierten Aufzeichnung des Bearbeitungsvorgangs einer Aufgabe und die anschließende Bewertung lösen, indem sie die definierten Anforderungen erfüllen. In Kapitel 6 erfolgt die Beschreibung der durchgeführten Umsetzung bzw. Implementierung des Konzepts. Die implementierten Systeme wurden im Anschluss technisch evaluiert, was in Kapitel 7 beschrieben ist. Das Kapitel 8 zieht als letztes Kapitel ein Fazit und gibt einen Ausblick auf weitere Arbeiten und Themenfelder, die an diese anschließen könnten.

2. Related Work

Dieses Kapitel beschreibt die verwandten Arbeiten zum Thema der Arbeit "Bewertung der Kodequalität in einer testgetriebenen Crowdsourcing-Plattform", um es in den Kontext der Forschung einzuordnen. Dafür wird an erster Stelle Bezug zum breiten Feld der Softwaremetriken & Softwarequalität genommen. Anschließend wird ein Überblick über unterstütztes maschinelles Lernen im Allgemeinen gegeben, um abschließend die Arbeiten zu betrachten, die Lernalgorithmen zur Qualitätsanalyse benutzen.

2.1. Softwaremetriken & Softwarequalität

Softwaremetriken spielen schon seit längerer Zeit bei der Evaluation der Qualität von Softwareprojekten eine Rolle (Li und Cheung 1987). Dabei dienen sie nach dem "Goal, Question and Metric Approach" aus Basili u. a. (1994) der **Quantifizierung** eines Sachverhalts, welcher anschließend durch **Interpretation der Metriken** qualitativ bewertet werden kann. Softwaremetriken lassen sich in 4 Kategorien unterteilen (Fenton und Bieman 2014):

1. Product
2. Process
3. Change and Evolution
4. Ressources

Innerhalb der Klassen kann wiederum zwischen *internen* und *externen* Metriken differenziert werden. Interne Metriken können direkt während jeder Phase des Software-Entwicklungszyklus' gemessen werden, während externe von der Ausführung und Umgebung abhängen (Fenton und Bieman 2014). Ein klassisches Beispiel für eine interne Metrik ist die Anzahl der Zeilen eines Softwaresystems, wohingegen die gemessene Anzahl an Fehlern pro Minute eine externe Metrik ist.

2.1.1. Softwarequalität

Softwarequalität kann viele Ausprägungen haben und wird je nach Kontext unterschiedlich definiert. Zum Beispiel kann sich die Qualität auf das Produkt oder den Enwicklungsprozess

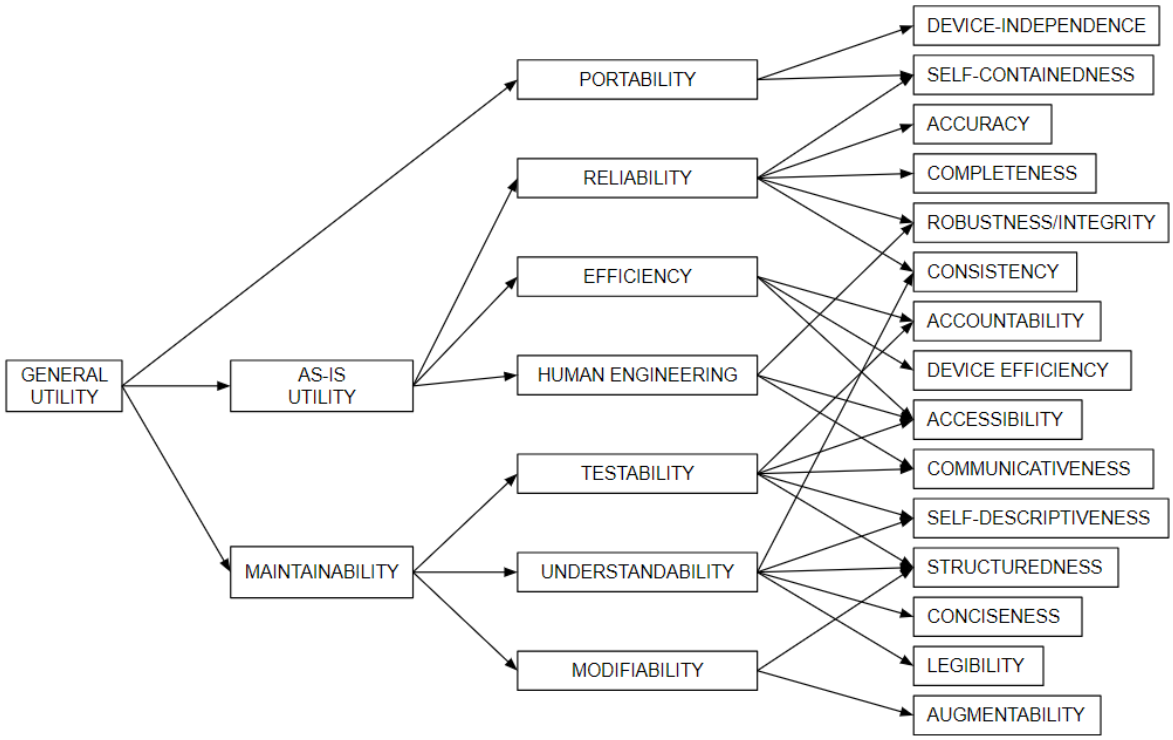

Abbildung 2.1.: Baum der Softwarequalität Charakteristiken (Boehm u. a. 1976)

beziehen. Im Vordergrund dieser Arbeit steht die Qualitätsbewertung des Produkts bzw. Ergebnisses einer Aufgabe auf einer testgetriebenen Crowdsourcing-Plattform. Deshalb nimmt die folgende Qualitätsdefinition ausschließlich Bezug zur Produktqualität. Für die Definition wird die Arbeit von Boehm u. a. (1976) herangezogen. Dort wurden in einem ersten Schritt eine Liste mit 11 Charakteristiken von Softwarequalität herausgearbeitet. Bekanntlich überschneiden sich einige Aspekte und andere wiederum nicht.

Daher überführt Boehm u. a. (1976) diese Liste in eine hierarchische Baumstruktur, welche die Beziehungen der Charakteristiken untereinander beinhaltet (vgl. Abbildung 2.1). Die drei Knoten *General Utility*, *As-Is Utility* und *Maintainability* stellt Boehm u. a. (1976) als den übergeordneten Nutzen auf. Darunter ordnen sich die Qualitäts Charakteristiken die dem jeweiligem Nutzen dienen an. Besitzt eine Software also beispielsweise eine verständliche Struktur bzw. verständlichen Quellcode (*Understandability*), ist testbar (*Testability*) und einfach modifizierbar (*Modifiable*), dann ist sie auch wartbar. Unter den einzelnen Qualitäts Charakteristiken ordnen sich wiederum die einzelnen Kriterien an, welche die Charakteristiken umsetzen. Auch hier gilt wieder, wenn alle Kriterien erfüllt sind, dann ist auch die übergeordnete Charakteristik erfüllt. Den Unterschied zwischen Kriterien und Charakteristiken benennt Fenton und Bieman (2014) mit dem direkten Bezug der Kriterien zu Metriken, welche die Messung eines Kriteriums ermöglichen. Charakteristiken sind wiederum nicht direkt, sondern nur mithilfe der Kriterien messbar.

Die resultierende Qualitätsdefinition ist also weniger eine textuelle Definition, wie man sie üblicherweise vornimmt. Vielmehr ist die Definition als Modell umgesetzt, welches einzelne Aspekte von Qualität in Beziehung zueinander stellt.

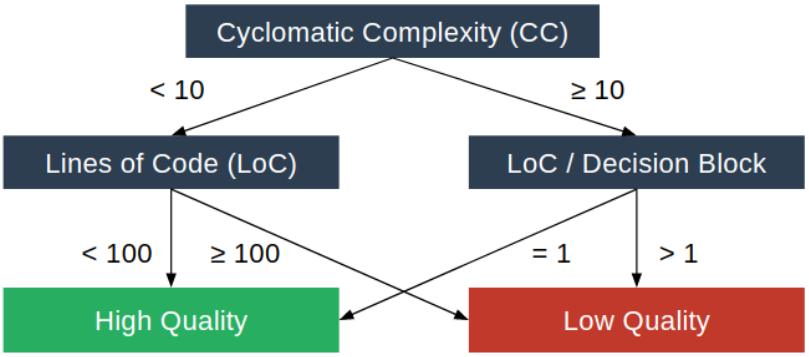

Abbildung 2.2.: Entscheidungsbaum mit drei Softwaremetriken

2.1.2. Probleme bei der Metrik-Analyse

> »*It is not difficult to propose some quantitative measures for software. It is not easy, however, to validate them.* «
>
> — Wei Li

Das Zitat von Wei Li beschreibt die Problematik bei der Analyse von Softwaremetriken sehr gut. Auf der einen Seite ist es nicht schwer viele Metriken zu berechnen, denn in den vergangenen Jahrzehnten haben sich viele Metriken zu traditionellen und häufig eingesetzten Metriken etabliert (Fenton und Bieman 2014). Auf der anderen Seite sind sie nach der Berechnung nur einzelne Werte, welche für die Beschreibung der Qualität weiter analysiert werden müssen. Dafür werden häufig ein oder mehrere Grenzwert definiert welche die Metrik einzelnen Qualitätsstufen zuordnet. Beispielsweise gilt im Allgemeinen die Annahme *Wenn die zyklomatische Komplexität über 10 liegt, ist die untersuchte Funktion zu komplex und sollte aufgespalten werden* (McCabe 1976). Diese Regel ist allerdings mehr als eine obere Grenze zu verstehen, da auch Funktionen mit wesentlich kleinerer zyklomatischer Komplexität schon zu komplex sein können.

Ein vielfach verwendeter Ansatz ist die Kombination mehrerer Metriken zu einem Entscheidungsbaum. Abbildung 2.2 zeigt einen solchen Entscheidungsbaum. Im verwendeten Beispiel wird die zyklomatische Komplexität mit den beiden Metriken *Lines of Code* und *Lines of Code per Decision Block* kombiniert. Beispielsweise ist nach dem Entscheidungsbaum auch eine Funktion mit einer zyklomatischen Komplexität kleiner als 10 und einer Anzahl von Codezeilen größer als bzw. gleich 100 von niedriger Qualität. Der Entscheidungsbaum hilft also bei der besseren Bewertung der Metrikwerte, aber stößt ebenfalls an Grenzen. Erstens lassen sich auch aus Entscheidungsbäumen wieder Grenzfälle generieren, in denen eine falsche Entscheidung getroffen wird, da oft zu wenige Metriken miteinander korreliert werden. Außerdem ist die manuelle Erstellung großer Entscheidungsbäume sehr aufwendig und lässt sich trotzdem häufig nicht auf andere Projekte ertragen (Al-Jamimi und Ahmed 2013). Außerdem können komplexe Zusammenhänge der Metriken häufig nicht von Hand erkannt werden. Aus diesem Grund werden zur Bewertung der Qualität Algorithmen eingesetzt, welche die automatische Korrelation vieler Metriken übernehmen.

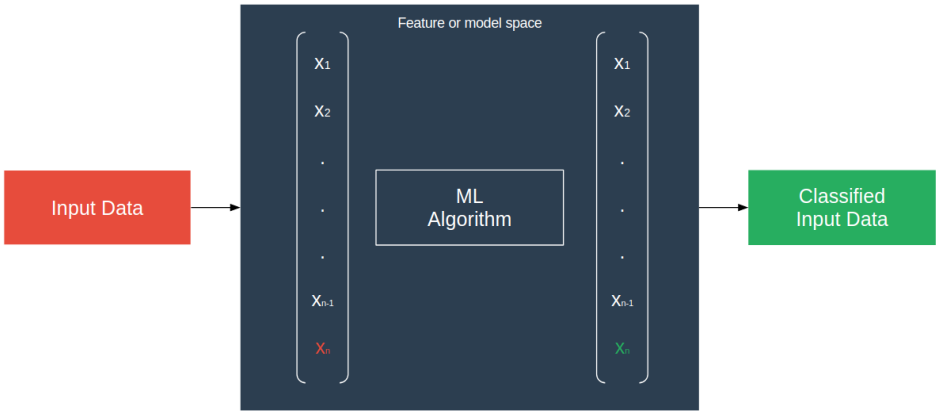

Abbildung 2.3.: Überblick über maschinelles Lernen

2.2. Unterstütztes maschinelles Lernen

An dieser Stelle folgt eine kurze Erläuterung der Grundlagen unterstützter maschineller Lernalgorithmen, um ein gemeinsames Vokabular für diese Arbeit zu definieren. Die Ausführungen haben nicht den Anspruch der Vollständigkeit, sondern dienen vor allem einer ersten Übersicht zum Thema "unterstütztes maschinelles Lernen". Sie wurden größtenteils aus der Arbeit von Razavi und Kurfess (2007) zusammengefasst.
Maschinelles Lernen (ML) dient dem Erkennen von Zusammenhängen zwischen Eingabedaten und einer Menge an Klassen durch einen Computer. Als möglicher Anwendungsfall ist das Thema dieser Arbeit zu nennen, wo Quellcodes verschiedenen Qualitätsstufen und Ausprägungen zugeordnet werden sollen, um sie zu bewerten. Dabei ist die Bewertung von **guter** und **schlechter Lesbarkeit** ein Beispiel für ein binäres Klassifizierungsproblem (Zuordnung von 2 Klassen), welches mithilfe von maschinellem Lernen gelöst werden kann. Die Eingabedaten sind die verschiedenen Quellcodes, welche Klassifiziert werden sollen. In Abbildung 2.3 wird der Datenfluss beim maschinellen Lernen dargestellt. Dabei wird ein Eingabedatum als *Instance* bezeichnet und zuerst in den *Feature* bzw. *Modell-Raum* überführt, damit der ML-Algorithmus mit den Daten arbeiten kann. Bei der Bewertung von Quellcodes ist der Quellcode, der bewertet werden soll, die Eingabe-Instanz und die Berechnung von Metriken die Überführung in den Modellraum. Der sogenannte *Feature-Vektor* beinhaltet alle Features, die zur Klassifikation benutzt werden. Dabei ist das letzte Element im Feature-Vektor (rotes x_n) das unbekannte Label, welches durch den Algorithmus zugeordnet werden soll. Die Berechnung des Labels übernimmt ein ML-Algorithmus. Anschließend beinhaltet der Feature-Vektor das zugeordnete Label, womit die Klassifizierung abgeschlossen ist.
Die Besonderheit beim unterstützten maschinellem Lernen (im Folgenden abgekürzt mit SML, stellvertretend für *Supervised Machine Learning*) ist, dass bereits gelabelte Instanzen vorliegen. Das heißt es existiert ein Datensatz von Feature-Vektoren für die das jeweilige Label bekannt ist. Im Fall des ununterstützten maschinellen Lernens sind diese vorklassifizierten Instanzen nicht vorhanden. Vielmehr ist dort das Ziel durch *Clusteranalyse* die Klassen eines Datensatzes zu berechnen.
Der Prozess der *Classifier* Erstellung anhand einer vorhandenen zu-lösenden Fragestellung oder eines Problems nennt man *induktives maschinelles Lernen* und ist in Abbildung 2.4 dargestellt. Im ersten Schritt gilt es durch einen Experten den relevanten Datensatz zu definieren, auf dem die Analyse stattfindet. Dieser kann unter Umständen Lücken, Rauschen, Ausreiser,

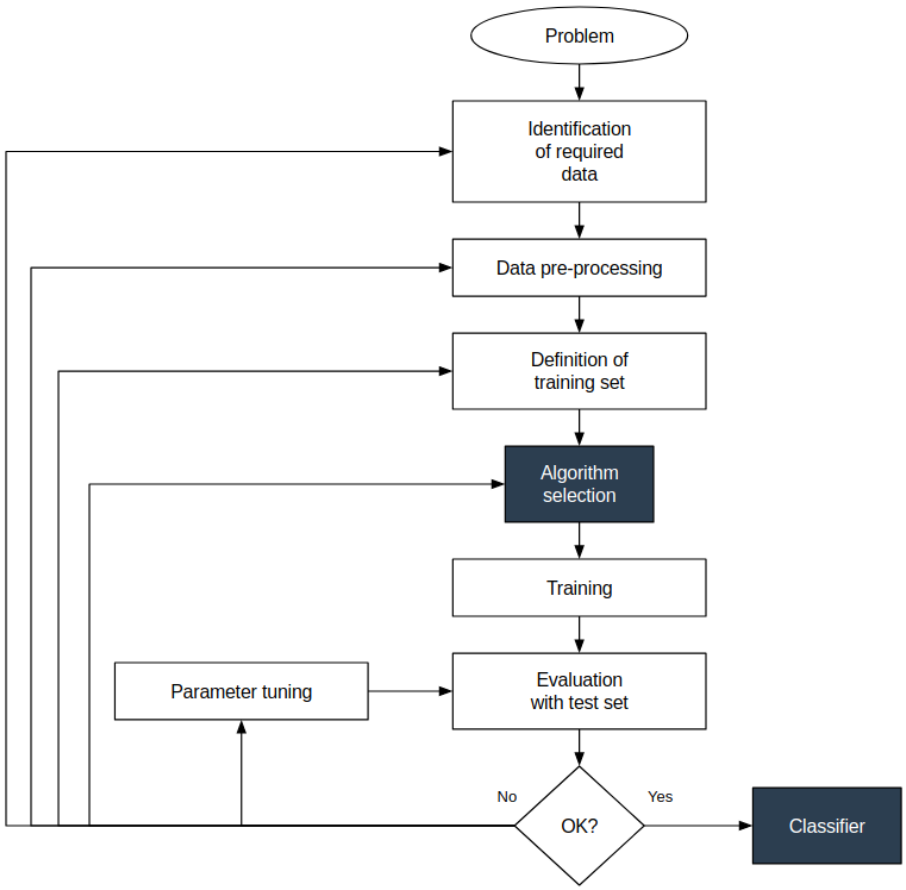

Abbildung 2.4.: Der Prozess des SML nach Razavi und Kurfess (2007)

o. ä. enthalten und muss in diesem Fall im Schritt 2 bereinigt werden. Anschließend muss der Trainingsdatensatz definiert werden, anhand dessen der Lernalgorithmus die wahrscheinlichen Zusammenhänge berechnet (3). Für diesen Schritt ist wichtig, dass nicht einfach alle gelabelten Instanzen verwendet werden können, da bei der späteren Evaluation ein Testdatensatz benötigt wird, der ebenfalls gelabelte Instanzen beinhaltet. Die Auswahl des Lernalgorithmus, muss auf die gewählten Features, Klassen usw. abgestimmt sein. Beispielsweise ist es mit einem *Linear Regression* Lernalgorithmus nur möglich diskrete und ungeordnete Werte zu behandeln. Oft liegen hier die Auswahlkriterien aber noch stärker im Detail, weshalb auch diese wichtige Entscheidung oft über mehrere Iterationen gehen muss (Domingos 2012). In der anschließenden Trainingsphase (auch als Lernphase bezeichnet) werden Testdatensatz und Lernalgorithmus zusammengebracht und durch für den jeweiligen Algorithmus spezifische Parameter ergänzt. Danach ist der sogenannte *Classifier* in der Lage einer neuen ungelabelten Instanz ein Label zuzuweisen. Die Wahrscheinlichkeit mit der diese Zuweisung stimmt wird in der *Evaluation* ermittelt. Dort wird mithilfe des bereits erwähnten Testdatensatzes überprüft, wie häufig der Classifier mit seiner Berechnung richtig oder falsch lag. Liegt dieser Wert unter einem gewünschten Grenzwert muss zu einem der Prozessschritte zurückgekehrt und entsprechende Änderungen vorgenommen werden.

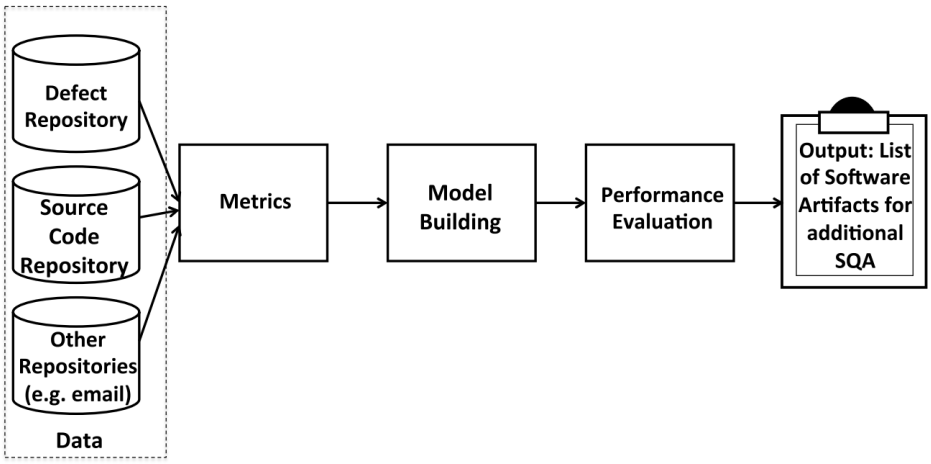

Abbildung 2.5.: Prozess der Software Fault Prediction (Kamei und Shihab 2016)

2.3. Qualitätsbewertung mit maschinellem Lernen

Im Abschnitt 2.1 wurde das Problem, der Interpretation der Metriken beschrieben, um sie zur qualitativen Bewertung zu benutzen. Wie beschrieben ist diese nicht immer einfach durchführbar, da oft erst mehrere Metriken zusammen eine Auskunft über die Qualität geben. Lernalgorithmen sind in der Lage solche komplexen Zusammenhänge anhand einer Menge von Trainingsdaten zu erkennen und diese auf neue Datensätze zu übertragen. Sie eignen sich daher für die Interpretation der Softwaremetriken, ohne Zusammenhänge und Grenzwerte explizit angeben zu müssen.
Die Interpretation von Softwaremetriken mithilfe von maschinellem Lernen ist ein Ansatz, der in der Literatur schon häufig betrachtet wurde. Die Codequalität kann dabei natürlich viele Ausprägungen haben, wie schon in Abschnitt 2.1 gezeigt wurde. Viele Forschungsgruppen beschäftigen sich mit der Korrelation von Metriken und der **Fehleranfälligkeit als Qualitätseigenschaft einer Software** (Rathore und Kumar 2017). In Abbildung 2.5 wird der Prozess bei der sogenannten *Software Fault Prediction (SFP)* dargestellt. Dabei beziehen sich diese Arbeiten meist auf ganze Projekte und analysieren dabei, wie ganz links im Bild zu sehen, verschiedene Projekt-Repositories. Die Konzepte für die Anwendung der Lernalgorithmen unterscheiden sich in den einzelnen Arbeiten häufig stark voneinander, aber aufgrund der Verwendung der Lernalgorithmen als Basis für die Korrelation decken alle die vier Schwerpunkte ab (Shepperd u. a. 2014):

1. Auswahl der Metriken als Features
2. Auswahl der Fehleranfälligkeits-Klassen als Labels
3. Auswahl einer Lernmethode/-technologie
4. Auswahl einer Methode zur Evaluation der Ergebnisse

Die Abbildung 2.5 fasst die Schritte (2) und (3) im *Model Building*-Prozessschritt zusammen. Als Ausgabe dieser Analyse steht eine Liste von Software-Artefakten, welche mit großer Wahrscheinlichkeit Fehleranfällig sind und bei der *Software Quality Assurance (SQA)* priorisiert werden sollten.

Die SFP stellt einen breit untersuchten Anwendungsfall von unterstütztem maschinellem Lernen zur Bewertung einer Ausprägung von Softwarequalität dar. Die Resultate und Erkenntnisse der vorangegangenen Arbeiten können für die vorliegende Arbeit verwendet werden.

Priorisierung der Prozessschritte Die Auswertung vieler Arbeiten durch Shepperd u. a. (2014) zeigt, dass die Auswahl der Lernmethode bzw. -technologie den geringsten Einfluss auf die Erfolgswahrscheinlichkeit einer Studie hat. Als größte Einflussfaktoren werden die durchführende Forschungsgruppe, der verwendete Datensatz und die Auswahl der Metriken aufgeführt. Diese Beobachtung deckt sich größtenteils mit den Herausforderungen, die Domingos (2012) für die Anwendung von Lernalgorithmen aufstellt. Dort wird u. a. die zunehmende Bedeutung der Konzepte für die Anwendung von Lernalgorithmen beschrieben. So ist zum Beispiel die Auswahl der Features und Labels sowie die Bedeutung der Datenmenge inzwischen wesentlich höher zu priorisieren als die Auswahl des Algorithmus' zur Klassifikation (Domingos 2012). Voraussetzung dafür ist der Einsatz eines Frameworks[1] oder Dienstes[2] der die Austauschbarkeit der Lernmethode einfacher möglich macht, als das noch vor einigen Jahren der Fall war.

Zugriff über Web-Schnittstellen Oftmals ist die Integration der Werkzeuge zur Qualitätsbewertung in andere Werkzeuge erwünscht. Damit dies möglich ist, müssen die Werkzeuge über Schnittstellen verfügen, welche möglichst variabel einsetzbar sind. Kamei und Shihab (2016) stellt daher die Bereitstellung von diesen Schnittstellen als Kriterium an zukünftige Werkzeuge, da aktuell noch wenige ihre Funktionen und Daten darüber bereitstellen.

Größere Datenvielfalt Nach Kamei und Shihab (2016) spielt die Vielfalt der vorhandenen Features bzw. Metriken eine große Rolle. Diese kann vor allem dann erreicht werden, wenn neue und detailliertere Informationen z. B. über den Entwickler und den Prozess aufgezeichnet werden (Rathore und Kumar 2017).

Lokale Anwendung vor Globaler Die Auswahl des Zielkontexts für die Anwendung des erarbeiteten Classifiers ist eine wichtige Entscheidung. Auf der einen Seite soll er möglichst Allgemeingültig eingesetzt werden können, verliert dadurch aber unter Umständen an Zuverlässigkeit bei der Klassifizierung der Instanzen (Kamei und Shihab 2016). Rathore und Kumar (2017) schlägt daher vor, dass der Einsatz zuerst lokal in Bezug auf einen abgegrenzten Kontext zugeschnitten und evaluiert werden sollte. Später kann dieser iterativ zunehmend globalisierter, also in weiteren Projekten oder Projektfelden, angewendet werden.

2.3.1. Abgrenzung zur vorliegenden Arbeit

Die betrachteten Arbeiten beschäftigen sich mit der Fehleranfälligkeit von Komponenten als Qualitäts-Charakteristik. Die vorliegende Arbeit zielt hingegen auf die Korrelation des Datensatzes mit mehreren Qualitäts-Charakteristiken ab, welche durch einen Qualitäts-Experten bewertet werden mit dem Ziel die **Qualität eines Quellcodeabschnitts** zu bewerten (vgl. Abschnitt 2.1). Weiterhin fokussieren sich die untersuchten Arbeiten auf ganze Projekte. Die Datengrundlage ihrer Analysen sind oftmals ganze Projektrepositories. In der vorliegenden

[1] TensorFlow: `https://www.tensorflow.org/` (Letzter Aufruf: 08.09.2017)
[2] AWS Machine Learning: `https://aws.amazon.com/de/machine-learning/` (Letzter Aufruf 08.09.2017)

Arbeit liegt der Fokus auf **einzelnen abgegrenzten Aufgaben** bzw. deren Lösungen, welche auf der Ebene einzelner Funktionen und Dateien angesiedelt sind. Abschließend steht das Analyseergebnis und die Evaluation der Methodik im Vordergrund der betrachteten Arbeiten. Außerdem setzt der Schwerpunkt dieser Arbeit an der Stelle einer **technischen Umsetzung eines Werkzeugs** an, welches die Durchführung solcher Analysen im Kontext einer tesgetriebenen Crowdsourcing-Plattform ermöglicht. Dementsprechend steht die Performance einzelner Bewertungen der Codequalität erst an zweiter Stelle, nachdem deren Durchführung technisch umgesetzt wurde.

2.4. Zusammenfassung

In diesem Kapitel wurden in einem ersten Schritt herausgearbeitet, das Softwaremetriken **numerische Werte oder Klassen** sind, welche aus den Quellcode bzw. den Entwicklungsprozessen berechnet werden können. Softwarequalität lässt sich für den Kontext dieser Arbeit am besten **strukturell** betrachten, wobei die verschiedenen Ausprägungen von Qualität, wie z. B. Wartbarkeit, in einem Baum zueinander in Bezug gebracht werden können. Dafür wurde das Modell von Boehm u. a. (1976) vorgestellt. Anschließend wurde der **grundlegende Datenfluss und Workflow** beim unterstützten maschinellen Lernen und wichtige Vokabeln, wie **Features**, **Labels** und **Instanzen** erklärt. Abschließend wurde **Software Fault Prediction** als ein Schwerpunktthema vorgestellt, was in der Forschung ein breit untersuchter Anwendungsfall der Vorhersage von Softwarequalität mit maschinellem Lernen ist.

3. State of the Art

In diesem Kapitel werden die Werkzeuge beschrieben, die aktuell eingesetzt werden, um die beiden betrachteten Problemstellungen dieser Arbeit, das Developer Monitoring und die Bewertung der Codequalität, umzusetzen. Für beide Prozessschritte werden separate Kriterien zur Bewertung der Werkzeuge vorgestellt. Anschließend wird jedes Werkzeug beschrieben und in einem Fazit mithilfe der Kriterien bewertet. In einer Zusammenfassung sind alle Werkzeuge und Kriterien in einer Tabelle aufgeführt und es wird ein Fazit bzgl. der Verwendbarkeit der Werkzeuge für die eigene Arbeit gezogen.

3.1. Aufzeichnung des Entwicklungsprozesses

Es existieren aktuell nicht sehr viele Werkzeuge, die sich der detaillierten Aufzeichnung der Entwicklungsaktivitäten des Entwicklers widmen. Zumindest fehlt es momentan noch an Werkzeugen, welche sich in den Entwickler-Alltag integrieren und detaillierte Metriken über die Performance eines Entwicklers geben. Dennoch wird an dieser Stelle ein Überblick über Werkzeuge gegeben, die aktuell das Mittel der Wahl sind, wenn man den Entwicklungsprozess eines Entwicklers aufzeichnen möchte. Um den Vergleich zu ermöglichen, werden folgende Kriterien benutzt:

Automatische und feingranulare Aufzeichnung Damit eine feingranulare Aufzeichnung des Entstehungsverlaufs des Ergebnisses möglich ist, muss diese automatisch, d. h. ohne Zutun des Entwicklers geschehen.

Getestete Versionsstände Um jeden erfassten Versionsstand in Bezug zu den Tests zu bringen, muss er einem automatischen Testlauf unterzogen werden.

Aktionserfassung Die Aktion oder Aktionen, welche zum neuen Versionsstand führten, müssen erfasst werden, damit nachvollzogen werden kann, wie die Änderung zustande kam.

Modellbasiert Damit eine Analyse erst möglich ist, müssen die aufgezeichneten Daten in einem strukturierten Format vorliegen.

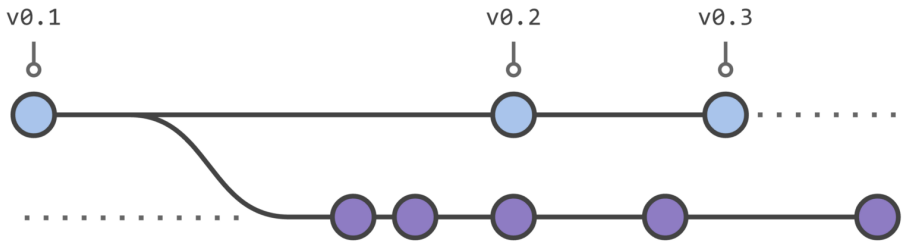

Abbildung 3.1.: Darstellung eines Versionsgraphen mit dem Werkzeug GitGraph (Bildquelle: `http://gitgraphjs.com/` (Letzter Aufruf: 12.08.2017))

Geeignete Lizenz Zur Verwendung des Werkzeugs im eigenen Projekt, muss es unter einer geeigneten Lizenz bereitstellt sein. Diese sollte möglichst die freie Verwendung ermöglichen, wie es z. B. die MIT Lizenz tut.

3.1.1. Versionskontrollsysteme

Versionskontrollsysteme (VCS) dienen schon seit geraumer Zeit der Persistierung des Verlaufs von Programmquellcode. Dabei ist hier nur die Rede von der Werkzeugart VCS, da das konkrete Werkzeug für die Betrachtung unerheblich ist. Die Arbeit mit einem VCS erfordert, dass der Entwickler seinen Quellcode *eincheckt* und so eine neue Version erstellt. Anschließend hat er u. a. die Möglichkeit zu alten Versionsständen zurückzukehren und die Unterschiede zum aktuellen Stand anzeigen zu lassen. Er ist dadurch in der Lage mit anderen Entwicklern an einem Projekt zu arbeiten, denn das VCS übernimmt die Anzeige von Konflikten zwischen Versionen und unterstützt bei der Auflösung dieser. Durch die Aufzeichnung entsteht ein Graph, welcher den Verlauf der einzelnen Versionsstände speichert. In Abbildung 3.1 ist das Werkzeug GitGraph[1] zu sehen, welches einen sehr kleinen Versionsgraphen visualisiert.

Fazit

Zwar haben sich die VCS für die Archivierung des Versionsverlaufs und für die Zusammenarbeit im Team etabliert, allerdings eignen sie sich nur bedingt für die detaillierte Aufzeichnung des Entwicklungsprozesses. Die Versionsstände werden strukturell in den für VCS übliche unveränderliche Commits gespeichert. Trotz alledem fehlt es ihnen an der automatischen Aufzeichnung von Versionsständen, da der Nutzer manuell committen muss. Weiterhin werden weder Tests automatisch zu jedem Testlauf ausgeführt, noch werden Aktionen die zu Änderungen führen erfasst. Die meisten VCS Werkzeuge sind frei einsetzbar, weshalb die Lizenz kein Grund für eine Ablehnung der Werkzeuge wäre.

3.1.2. HackerRank

HackerRank[2] ist eine Recruiting-Plattform, die es Unternehmen ermöglicht, die technischen Fähigkeiten der angemeldeten Entwickler anhand von Aufgaben auf der Plattform zu ermitteln. Entwickler lösen diese Aufgaben anhand von Tests und können anschließend durch das Unternehmen, welches die Aufgabe eingestellt hat, bewertet und eventuell zu einem Vorstellungsgespräch eingeladen werden. HackerRank ist also keine Plattform, die sich in den

[1] `http://gitgraphjs.com/` (Letzter Aufruf: 12.08.2017)
[2] `http://hackerrank.com/` (Letzter Aufruf: 04.09.2017)

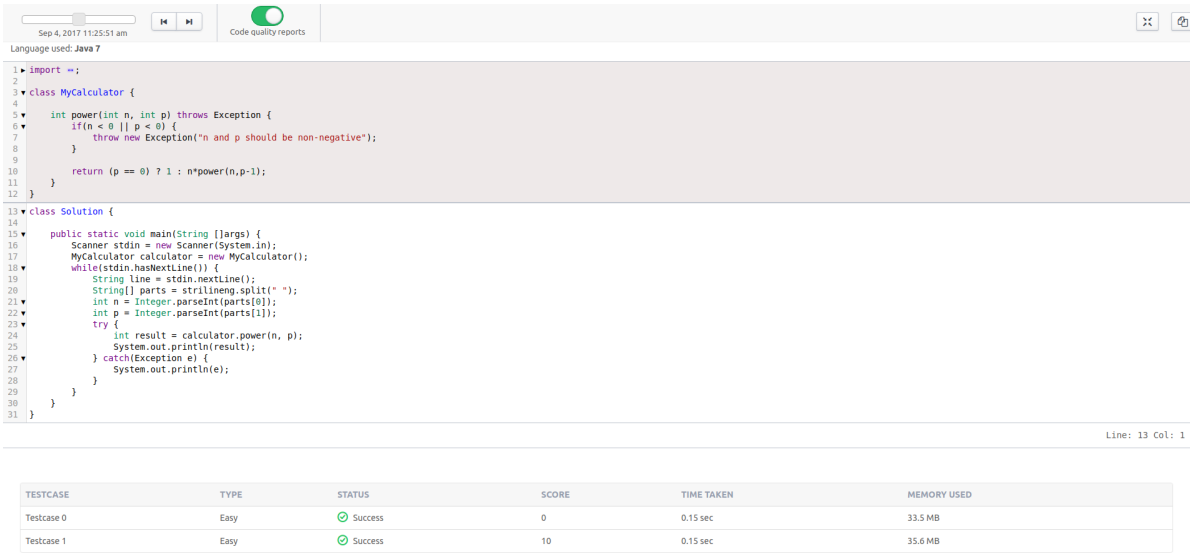

Abbildung 3.2.: Detaillierte Code-Review Ansicht mit HackerRank (Bildquelle: `https://www.hackerrank.com/` (Letzter Aufruf: 04.09.2017))

Entwicklungs-Alltag integriert, sondern vielmehr im Einstellungsprozess von Entwicklern angesiedelt ist. Trotzdem bietet HackerRank einen ähnlichen Ansatz des Developer Monitorings, der Entwickler während des Einstellungstests aufzeichnet und deren Vorgehen zum Teil bewertet. Beispielsweise wird aufgezeichnet, wie lange ein Entwickler auf speziellen Screens der Anwendung verweilt. Außerdem speichert das System jeden angestoßenen Testlauf des Entwicklers, dessen Testergebnis und den Versionsstand. Im Review ist es dann möglich diese einzelnen Stände zu laden und die Testergebnisse zu einem Stand anzuzeigen (vgl. Abbildung 3.2).

Fazit

Trotz des ähnlichen Ansatzes ist HackerRank bei der Aufzeichnung des Vorgehens des Entwicklers zu grobgranular. So speichert das System zwar Ergebnisse zu jedem gespeicherten Testlauf in einem strukturierten Format ab, allerdings werden nur manuell angestoßene Tests und deren Zwischenstände gespeichert. Eine automatische Persistierung der Versionsstände findet nicht statt. Auch die Aktionen die zu den Versionsständen führten werden nicht persistiert. Außerdem ist die komplette Plattform in sich geschlossen und ermöglicht keine Erweiterungen bzw. Integration über eine Schnittstelle zu anderen Werkzeugen oder Plattformen.

3.2. Bewertung von Codequalität

Softwareprojekte sind sehr vielfältig und die Unterschiede der Entwicklungsprozesse in verschiedenen Firmen sind sehr groß. Aus diesem Grund ist es sehr schwierig ein Werkzeug zu entwickeln, welches auf der einen Seite breite Einsatzmöglichkeiten bietet und trotzdem ausreichend Tiefe, um einen Mehrwert für die Bewertung der Qualität eigener Entwicklungsprozesse zu liefern. Im folgenden sind einige Werkzeuge aufgeführt, die aktuell für die Bewertung von Codequalität eingesetzt werden. Die folgenden Kriterien dienen der Vergleichbarkeit der Werkzeuge untereinander und zur Abgrenzung der eigenen Arbeit:

Mehrere Programmiersprachen Da die Plattform auf die Auswertung mehrere Programmiersprachen ausgelegt ist, muss auch das analysierende Tool darauf ausgelegt sein, sprachunabhängige Analysen durchzuführen oder zumindest eine Unterstützung für mehrere Sprachen zu bieten.

Analyse Web-API Da die Analysekomponente mit anderen Komponenten der Plattform zusammen arbeiten muss ist es notwendig, dass die Analyse über eine definierte Schnittstelle angestoßen werden kann. Dabei ist besonders wichtig, dass sowohl der Startbefehl einer Analyse sowie auch die Daten übermittelt werden können, welche analysiert werden sollen.

Ergebnis Web-API Ähnlich wie beim vorherigen Kriterium ist es auch für die Ergebnisse der durchgeführten Analyse wichtig, dass sie über eine Schnittstelle bereitgestellt werden, um von den anderen Komponenten der Plattform verwendet werden zu können.

Lernen von Qualität Das Werkzeug zur Bewertung der Codequalität muss in der Lage sein, durch Anwendung von Lernalgorithmen, die Bewertung der Codequalität vorzunehmen. Dadurch müssen Grenzwerte nicht manuell vom Benutzer festgelegt werden, sondern werden vom System mithilfe der Trainingsdaten gelernt.

Datensatz - Erweiterbarkeit Die Berechnung der Codequalität sollte auch auf eigene Datensätze erweiterbar sein. So sollte zum Beispiel nicht nur der abgegebene Quellcode analysiert werden können, sondern auch andere Datensätze, wie die des Developer Monitorings.

Geeignete Lizenz Zur Verwendung des Werkzeugs im eigenen Projekt, muss es unter einer geeigneten Lizenz bereitstellt sein. Diese sollte möglichst die freie Verwendung ermöglichen, wie es z. B. die MIT Lizenz tut.

3.2.1. Landscape

Landscape[3] ist eine Web-Plattform, die Qualitätsmetriken für Python Projekte berechnet und übersichtlich darstellt. In Abbildung 3.3 ist das Landscape-Dashboard dargestellt, welches für das untersuchte Projekt einen *Qualitäts-Score* von 66 % angibt. Landscape berechnet diesen Score aus einzelnen Metriken und anhand von konfigurierbaren Regeln für die einzelnen Versionen eines Projektes. Dazu baut Landscape auf die Integration in das GitHub-Ökosystem[4] auf. So ist es möglich, dass ein GitHub-Projekt sehr einfach untersucht werden kann, da der Link zum Repository reicht, um dieses zu analysieren. Der Fokus von Landscape bei der Analyse liegt auf der Bereitstellung von Qualitätsmetriken und aufbauend auf deren Verlauf zur Detektion von Trends.

Fazit

Landscape bietet eine strukturierte und moderne Übersicht über grundlegende Qualitätsmetriken sowie einer einfachen Integration eines Python GitHub-Projekts. Landscape ist über die einfache Integration des GitHub-Ökosystem hinaus leider sehr stark abgegrenzt und ermöglicht keine Integration anderer Dienste. So ist die Analyse auf die Programmiersprache Python

[3] `https://www.landscape.io/` (Letzter Aufruf: 14.08.2017)
[4] `https://github.com/` (Letzter Aufruf: 14.08.2017)

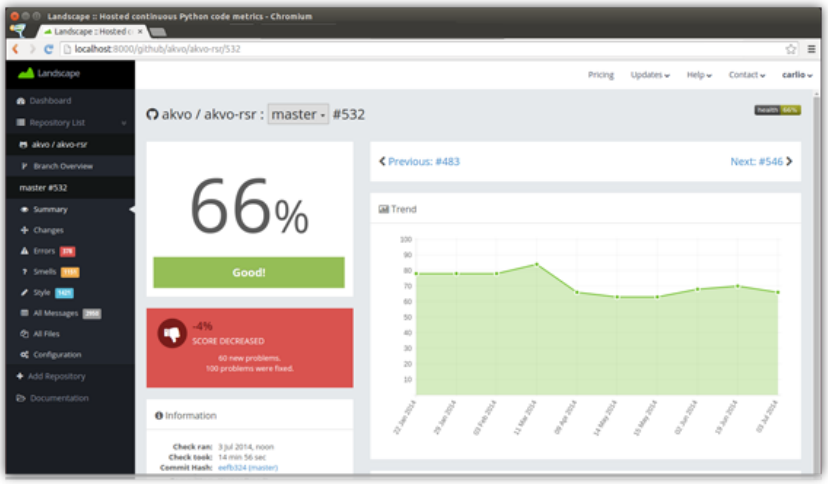

Abbildung 3.3.: Landscape Dashboard eines Projekts (Bildquelle: `https://www.landscape.io/` (Letzter Aufruf: 14.08.2017))

beschränkt und lässt sich ebenfalls nicht um eigene Datensätze erweitern. Auch an der Bereitstellung einer Web-API für die Analyse und Ergebnisse fehlt es. Zusätzlich findet innerhalb der Plattform nur eine rein regelbasierte Auswertung statt. Das heißt, anstelle von Lernalgorithmen zur Analyse kommen einfache Entscheidungsbäume zum Einsatz, welche vorher fest spezifiziert wurden.

3.2.2. SonarQube

SonarQube[5] ist ein Werkzeug zur statischen Codeanalyse und wird aktuell von vielen Unternehmen für die Qualitätskontrolle ihrer Softwareprojekte eingesetzt, da es einfach erweiterbar und integrierbar ist. SonarQube berechnet basierend auf einem Repository die gewünschten Qualitätsmetriken und stellt diese sowohl über eine Web-Oberfläche als auch über eine REST-Schnittstelle bereit. SonarQube berechnet anhand der Metriken und konfigurierbaren und erweiterbaren Regeln sogenannte *Issues*, welche dem Nutzer angezeigt werden und über den aktuellen Status des Projekts informieren (vgl. Abbildung 3.4). Es ist zudem in einer Vielzahl von Projekten anwendbar, da es sehr generisch aufgebaut ist. Das liegt auch daran, dass es als OpenSource Projekt auf GitHub[6] verfügbar 20.ist und als solches auch von den Nutzern weiterentwickelt wird.

Fazit

SonarQube ist stark erweiterbar und ermöglicht somit auch die Unterstützung mehrerer Programmiersprachen durch die Bereitstellung einer abstrakten Komponente (sogenannte *Scanner*). Implementierungen des Scanners sind für die verbreitetsten Programmiersprachen verfügbar. Außerdem stellt SonarQube REST-Schnittstellen bereit, weshalb die Integration in andere Werkzeuge möglich ist. Der Workflow von SonarQube für die Analyse ist, wie bei allen bisherigen Werkzeugen, Projekt & Repository basiert, was es erfordert jeglichen Quellcode der analysiert werden soll in ein Repository einzuchecken. Es ist zwar theoretisch möglich durch

[5] `https://www.sonarqube.org/` (Letzter Aufruf: 14.08.2017)
[6] `https://github.com/SonarSource/sonarqube` (Letzter Aufruf:

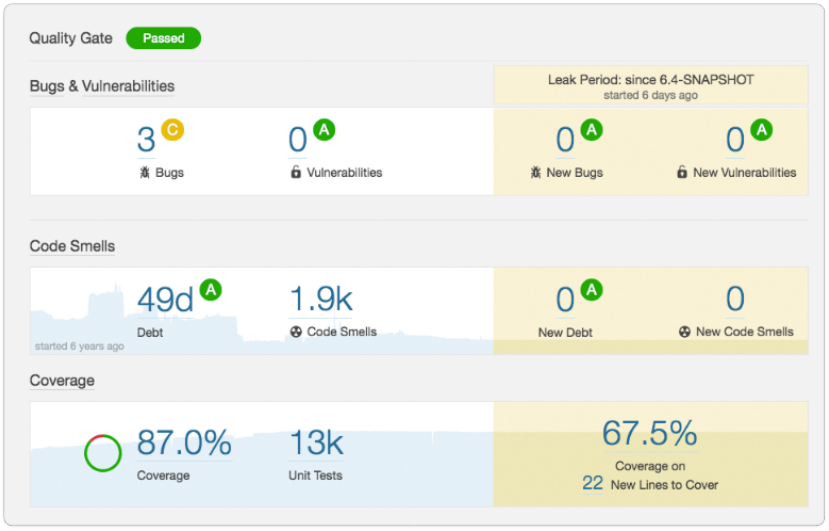

Abbildung 3.4.: SonarQube Dashboard eines Projekts (Bildquelle: `https://www.sonarqube.org/` (Letzter Aufruf: 14.08.2017))

die API des Scanners eine Schnittstelle zu implementieren, welche diesen Workflow aufbricht, allerdings würde man so gegen die Intension des Frameworks bzw. Werkzeugs arbeiten.
Ähnlich sieht es bei der Erweiterbarkeit der Berechnung der Metriken aus. So sieht die API des Scanners zwar vor, dass neue Metriken aus dem Quellcode berechnet werden können. Allerdings ist die Erweiterung der Datengrundlage über die Dateien des Projekt-Repositories hinaus nicht vorgesehen und deshalb ebenfalls nur über Umwege möglich.
Weiterhin setzt SonarQube anstelle von Lernalgorithmen zur Analyse auf den Einsatz von Entscheidungsbäumen. Diese sind zwar in SonarQube konfigurierbar aber nicht auf die Unterstützung von maschinellen Lernen ausgelegt.

3.3. Zusammenfassung

In diesem Kapitel wurden aktuelle Ansätze für die beiden Problemstellungen *Aufzeichnung des Entwicklungsprozesses* und *Bewertung von Codequalität* vorgestellt und auf Eignung bezüglich dieser überprüft. Abschließend wird diese Gegenüberstellung in Form zweier Tabellen zusammengefasst.
Anhand der Tabellen ist ersichtlich, dass keines der Werkzeuge die Mindestanforderungen erfüllt, welche die Problemstellung mit sich bringt. Die untersuchten Werkzeuge setzen zwar Teile des geforderten Prozesses um, allerdings fehlt es ihnen aufgrund eines anderen Einsatzkontextes an Funktionalitäten, um alle geforderten Funktionen abzudecken. Aus diesem Grund werden zur Lösung dieser beiden Problemstellungen zwei Systeme konzipiert und umgesetzt. Sie werden in den folgenden Ausführungen ***Developer Monitoring System*** und ***Quality Rating System*** genannt.

3.4. Anforderungsanalyse

In diesem Abschnitt werden die Anforderungen an die beiden Systeme zur Lösung der beiden Problemstellungen definiert. Dafür wird die ***MoSCoW****-Priorisierung* verwendet. Die ***Must-***

	Versionskontrollsysteme	**HackerRank**
Automatische Aufzeichnung	Nein	Nein
Getestete Versionsstände	Nein	Ja
Aktionserfassung	Nein	Nein
Modellbasiert	Ja	Ja
Geeignete Lizenz	Ja	Nein

Tabelle 3.1.: Vergleich etablierter Werkzeuge für die Aufzeichnung des Entwicklungsprozesses anhand der aufgestellten Kriterien

	Landscape	**SonarQube**
Mehrere Programmierspr.	Nein	Ja
Analyse Web-API	Nein	(Nein)
Ergebnis Web-API	Ja	Ja
Lernen von Qualität	Nein	Nein
Datensatz - Erweiterbarkeit	Nein	Nein
Geeignete Lizenz	Nein	Ja

Tabelle 3.2.: Vergleich etablierter Werkzeuge für die Bewertung von Codequalität anhand der aufgestellten Kriterien

Have Kriterien sind nicht verhandelbar und müssen zur Lösung des Problems umgesetzt werden. ***Should-Have*** Kriterien sind nicht erfolgskritisch, sollten allerdings hoch priorisiert werden. Im Gegensatz dazu haben die ***Could-Have*** Kriterien eine geringe Relevanz und sind gleichzusetzen mit "Nice to have" Funktionen. Abschließend sind die ***Won't-Have*** Kriterien für diese Arbeit nicht relevant, können aber später umgesetzt werden, weshalb sie an dieser Stelle erwähnt werden. Nicht-funktionalen Anforderungen sind zusätzlich aufgelistet, aber wurden aufgrund ihrer geringen Anzahl nicht priorisiert.

3.4.1. Developer Monitoring

In den folgenden Tabellen sind die Anforderungen für das Developer Monitoring aufgelistet und beschrieben. Für jede Stufe der MoSCoW-Priorisierung existiert eine Tabelle. Dabei wird für die Kriteriennummerierung der Interfix **DT** benutzt.

Tabelle 3.3.: Anforderungen an das Developer Monitoring - Must(M), Should(S), Could(C), Won't(W), Non-functional(NF)

ID	Name	Beschreibung
M-DM-10	Detaillierter Versionsverlauf	Für die vollständige Rekonstruktion des Versionsverlaufes müssen alle Aktionen, die der Nutzer durchführt zusammen mit dem entstehenden Versionsstand getrackt werden.
M-DM-20	Getestete Versionsstände	Damit die Versionsstände in Zusammenhang mit ihrem Testergebnis gebracht werden können muss jede gespeicherte Version einem Testlauf unterzogen werden, dessen Ergebnis gespeichert wird.
M-DM-30	Strukturierte Speicherung der Rohdaten	Die Daten des Enwicklertrackings müssen ohne Datenverlust gespeichert werden, damit die später analysiert werden können
M-DM-40	Web-Schnittstelle	Zur weiteren Verarbeitung der Daten und einfachen Integration in andere Systeme, müssen alle Daten des Developer Monitoring Datenmodells über eine Web-Schnittstelle abrufbar sein. Außerdem müssen auch Funktionen, wie das Speichern eines neuen Versionsstandes oder die Ausführung eines Testlaufs über die Schnittstelle angestoßen werden können.
S-DM-10	Mehrere Dateien	Es sollte möglich sein, dass das Developer Monitoring auch für Aufgaben mit mehreren Dateien funktioniert.
S-DM-20	Erkennen identischer Versionsstände	Es sollte möglich sein, dass das Developer Monitoring Versionsstände erkennt, die identisch sind. Für diese muss nicht erneut ein Testlauf durchgeführt werden.
S-DM-30	Erkennen syntaktisch korrekter Versionsstände	Um nur Versionsstände auszuführen die auch syntaktisch korrekt sind, sollte das Werkzeug überprüfen können, ob ein Versionsstand syntaktisch korrekt ist oder nicht.
C-DM-10	Review-Ansicht	Zur Visualisierung der aufgezeichneten Daten zu einer Aufgabe, wäre es von Vorteil eine Ansicht umzusetzen, die dem Reviewer eine Übersicht über die aufgezeichneten Daten gibt.

Fortgesetzt auf nächster Seite ...

...Fortgesetzt auf vorheriger Seite

ID	Name	Beschreibung
C-DM-20	Filterung der angezeigten Daten	Bei der Bearbeitung einer Aufgabe kann es zu Versionsständen kommen, deren Inhalt am Ende der Bearbeitung einer Aufgabe nicht mehr in der Lösung enthalten ist. Diese sind zwar für eine automatische Analyse interessant, sollten aber für die Visualisierung gefiltert werden.
W-DM-10	Weitere Auswertung der Daten	Durch die aufgezeichneten Versionsstände in Zusammenhang mit den Testergebnissen ist es möglich weitere Informationen über den Enwicklungsprozess zu gewinnen. Zum Beispiel wäre es möglich menschliches von unmenschlichem Verhalten beim Lösen einer Aufgabe zu erkennen oder Beziehungen zwischen Quellcode-Statements und Testfällen herzustellen.
W-DM-20	Absicherung vor *Denial of Service (DoS)* Angriffen	Die Absicherung vor Denial of Service (DoS) Angriffen ist für eine spätere Absicherung vor ungewollter Auslastung von Bedeutung und sollte vom System erkannt und abgefangen werden.
NF-DM-10	Anzahl der Zeichen einer Lösung	Das Developer Monitoring soll darauf ausgelegt sein eine Anzahl von bis zu **5000** Zeichen zu unterstützen.
NF-DM-20	Anzahl der Aktionen	Das Developer Monitoring soll darauf ausgelegt sein Aufgaben zu Unterstützen, die mit **10000** Aktionen bearbeitet wurden.
NF-DM-30	Tippgeschwindigkeit	Das Developer Monitoring soll Tippgeschwindigkeiten von bis zu 6 Zeichen pro Sekunde ermöglichen.
NF-DM-40	Anzahl an parallele arbeitenden Entwicklern	Das Developer Monitoring soll bis zu **5** parallel arbeitende Entwickler unterstützen.
NF-DM-50	Anzahl der Testfälle pro Aufgabe	Das Developer Monitoring soll bis zu **20** Testfälle pro Aufgabe unterstützen.

3.4.2. Quality Rating

In den folgenden Tabellen sind die Anforderungen für die Bewertung der Codequalität aufgelistet und beschrieben. Für jede Stufe der MoSCoW-Priorisierung existiert eine Tabelle. Dabei wird für die Kriteriennummerierung der Interfix **QR** benutzt.

Tabelle 3.4.: Anforderungen an das Quality Rating - Must(M), Should(S), Could(C), Won't(W), Non-functional(NF)

ID	Name	Beschreibung
M-QR-10	Extraktion der Qualitätsmetriken	Das System muss eine API bereitstellen, die es ermöglicht Metriken aus dem Endergebnis und den Daten des Developer Monitorings berechnen zu können. Die API muss auf der einen Seite eine Integration eines Werkzeugs, wie SonarQube, zur Auslagerung der Metrikenberechnung bereitstellen. Auf der anderen Seite muss die Berechnung von neuen Metriken aus dem Developer Monitoring implementiert werden können.
M-QR-20	Konfiguration der Features	Für den Nutzer muss es möglich sein, die Features zu konfigurieren die für den Lernschritt verwendet werden.
M-QR-30	Konfiguration der Klassifikationsmodelle	Für den Nutzer muss es möglich sein, verschiedene Klassifikationsmodelle und deren Labels zu konfigurieren die während des Anlernens durch den Bewertenden klassifiziert werden müssen.
M-QR-40	Anlernen durch Experten	Das Werkzeug muss ein Benutzer-Schnittstelle bereitstellen, welche das Anlernen durch einen Qualitäts-Experten ermöglicht. Der Experte muss anhand des angezeigten Ergebnis-Quellcodes eine Einordnung der verschiedenen konfigurierten Klassifikationsmodelle vornehmen können.
M-QR-50	Speicherung der Trainings- und Testdaten	Das System muss in der Lage sein die Trainings- und Testdaten zu persistieren.
M-QR-60	Web-Schnittstelle	Zur weiteren Verarbeitung der Daten und einfachen Integration in andere Systeme, müssen die konfigurierten und berechneten Daten des Quality Rating Systems über eine Web-Schnittstelle bereitgestellt werden. Weiterhin müssen Funktionen wie das Anstoßen einer Bewertung über die Schnittstelle aufgerufen werden können.
S-QR-10	Extraktion der Metriken mehrerer Spachen	Die Berechnung der Codemetriken sollte für mehrere Sprachen funktionieren, bzw. an ein Werkzeug weitergeleitet werden, welches mehrere Sprachen unterstützt.

Fortgesetzt auf nächster Seite ...

...Fortgesetzt auf vorheriger Seite

ID	Name	Beschreibung
S-QR-20	Datenbereinigung	Bevor die Daten für den Lernalgorithmus verwendet werden, könnten es sinnvoll sein, die Daten zu bereinigen, damit Ausreißer oder Rauschen das Bewertungsergebnis nicht verzerrt.
C-QR-10	GitHub Anbindung	Durch die geringe Anzahl an Nutzern am Anfang ist es schwierig ausreichend Daten für erste Bewertungen zu sammeln. daher sollte eine Anbindung an GitHub existieren, welche automatisch Code-Abschnitte aus ausgewählten Projekt-Repositories extrahiert und diese dann in der Trainings-Komponente benutzt werden können, um erste Trainingsdaten zu sammeln.
C-QR-20	Sprachabhängige Analyse	Die Analyse sollte die Möglichkeit bieten, dass ein Classifier nur für eine Sprache aufgebaut wird, damit Unterschiede der Sprachen bei der Bewertung berücksichtigt werden können.
W-QR-10	Optimierung des Classifier-Algorithmus	Zur Optimierung der Erfolgswahrscheinlichkeit bei der Klassifizierung von Quellcode, ist es möglich den Classifier-Algorithmus anzupassen oder autzutauschen.
W-QR-20	Bewertung des Enwicklers	Durch die aufgezeichneten Daten ist es auch möglich nicht nur den resultierenden Quellcode zu bewerten, sondern auch den Entwickler und dessen Performance zu bewerten.

4. Kontext

Die beiden Prozesse *Aufzeichnung des Entwicklungsprozesses* und *Bewertung der Codequalität* werden in dieser Arbeit für den Kontext einer *testgetriebenen Crowdsourcing-Plattform* konzipiert und umgesetzt. Dieses Kapitel gibt einen Überblick über *polylith*, welches solch eine Plattform ist. Anschließend an die Beschreibung werden die Teilprozesse in den Gesamtprozess einer solchen Plattform eingebettet.

4.1. Testgetriebene Crowdsourcing-Plattform

Die folgenden Ausführungen dieses Abschnitts erläutern den angestrebten Entwicklungsprozess der Plattform, in dem die beiden Teilprozesse dieser Arbeit *Developer-Monitoring* und *Codequality-Learning* aufgehangen sind. Sie wurden **vollständig** aus der Sektion "Crowdsouring-Plattform" der Arbeit von Hanspach (2017) übernommen:
Ziel der Plattform ist es, Softwareentwicklungs-Aufgaben mithilfe der Crowd parallel und damit sowohl zeit- als auch kosteneffizient lösen zu können. Um die Plattform dabei effizient und fair zu gestalten, sollen die Crowd-Entwickler grundsätzlich für, innerhalb eines Zeitlimits, korrekt gelöste Aufgaben bezahlt werden. Um diese Ziele erreichen zu können, wurde der folgende Prozess ausgearbeitet. Dieser ist an den Test-driven Development (TDD) Ansatz angelehnt. Abbildung 4.1 stellt diesen Workflow schematisch dar. Die Abbildung zeigt, wie der Softwareentwicklungsprozess durch Einsatz der Plattform, parallel zur internen Entwicklung genutzt werden kann. Schnittstelle zwischen interner und externer Entwicklung bildet dabei ein Branching-Modell des Versionsverwaltungs-Tools Git[1].

4.1.1. Softwareentwicklungsprozess innerhalb der Plattform

Der Softwareentwicklungsprozess beginnt, wie üblich, mit einem Auftrag. Dieser wird durch speziell geschulte Software-Architekten analysiert und auf Umsetzbarkeit geprüft. Wurde zusammen mit dem Auftraggeber (beispielsweise durch Agile Workshops) eine gemeinsame Lösung gefunden, beginnen die Architekten mit der Aufteilung der Aufgaben und dem Erstellen der automatischen Abnahmetests. Die Abnahmetests werden in Form natürlichsprachlicher Tests geschrieben und bilden gleichzeitig die Aufgabenstellung der Aufgabe. Dadurch sind Aufgabenstellung und Abnahmekriterien jederzeit konsistent und klar ersichtlich. Ist die Aufgabe hinreichend getestet, wird sie auf der Plattform zur Bearbeitung freigeschaltet. Eine Aufgabe

[1] `http://git-scm.com/` (Letzter Abruf: 03.07.2017)

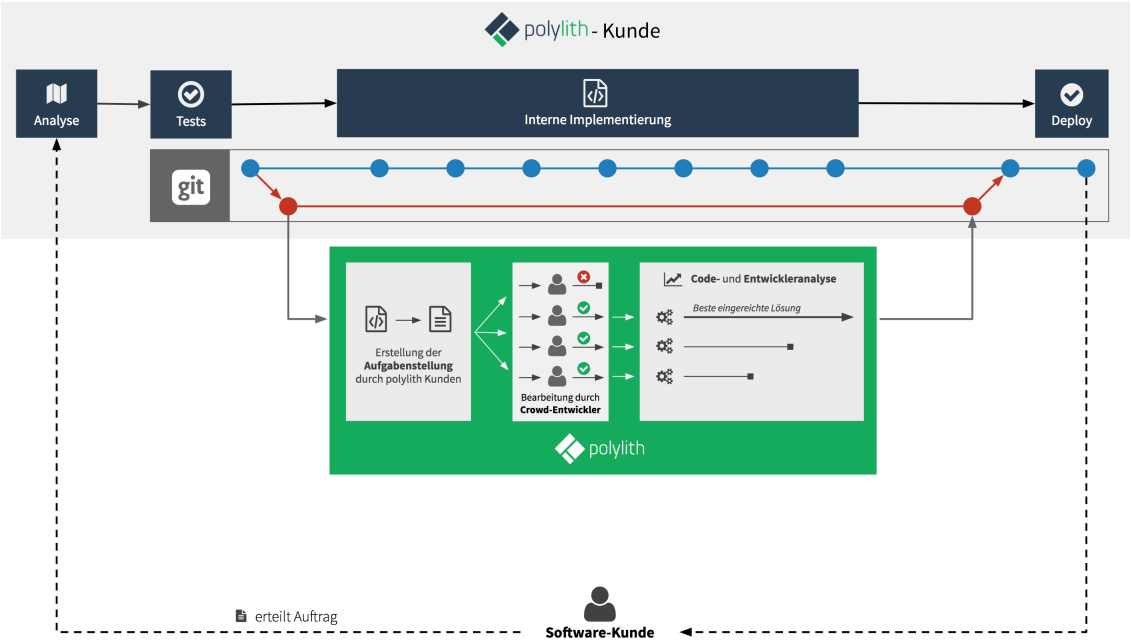

Abbildung 4.1.: Schematische Darstellung des test-getriebenen Crowdsourcing Ansatzes der polylith-Plattform als Ergänzung des Software-Entwicklungszyklus beim polylith-Kunden

ist hinreichend gut getestet, wenn sie im Rahmen des Plattformbetriebs die Testanforderungen erfüllt. Die Bestimmung, wann eine Aufgabe diese Bedingung erfüllt, ist nicht Teil dieser Arbeit.

Sobald eine Aufgabe veröffentlicht wurde, kann sie von einem oder mehreren Crowd-Entwicklern angenommen werden. Die Anzahl der Entwickler die gleichzeitig die Aufgabe bearbeiten dürfen, richtet sich nach den Vorgaben der Kunden. Mehr gleichzeitige Entwickler gewährleisten eine schnellere Bearbeitung und eine höhere Erfolgsquote, sind aber gleichzeitig teurer (da jede korrekt abgegebene Aufgabe auch bezahlt wird). Das Lösen der Crowdsourcing-Aufgabe findet direkt innerhalb der Plattform in einer bereitgestellten Web-IDE statt (Abbildung 4.2). Dadurch hat der Crowd-Entwickler keinerlei Aufwand beim Aufsetzen einer lokalen Entwicklungs- und Testumgebung. Weiterhin kann dadurch der Entwicklungszyklus jeder Aufgabe analysiert werden und beispielsweise auch Betrugsversuche (durch Kopieren einer korrekten Lösung) aufgedeckt werden. Zusätzlich wird der Entwickler durch den Editor unterstützt, indem unterstützende Grafiken (z. B. Klassendiagramme) eingeblendet werden oder Quellcode, der für die Lösung der Aufgabe nicht relevant ist, ausgeblendet wird. Die Web-IDE fügt den Quellcode des Projekts im Hintergrund wieder zusammen und ermöglicht dadurch die Bearbeitung eines kleineren Projekts für den Crowd-Entwickler.

Vollendet der Crowd-Entwickler die Aufgabe und erfüllt alle Tests, kann sie abgegeben werden. Bei der Abgabe wird der Quellcode einer erneuten Qualitätsüberprüfung unterzogen. Hierbei werden nicht-funktionale Qualitätsmerkmale wie Lesbarkeit, Doppelter Quellcode, Verschachtelungstiefe und weitere Code Smells (vgl. Fowler und Beck (1999)) untersucht. Der Nutzer soll bei besserer Erfüllung dieser Kriterien zusätzlich unentgeltlich, in Form eines Gamification-Ansatzes, belohnt werden. Dadurch sollen sich die Entwicklungsfähigkeiten der einzelnen Crowd-Entwickler verbessern. Durch die Qualitätsanalyse wird weiterhin die Notwendigkeit

eines zusätzlichen manuellen Reviews erkannt.

Sind alle Aufgaben durch die Crowd gelöst, können diese zu einem gemeinsamen Endergebnis zusammengefasst werden. Aufgrund des test-getriebenen Ansatzes erfüllt diese Lösung exakt die Anforderungen, die zuvor in den Tests aufgestellt wurden. Das fertige Ergebnis kann nach einem letzten Qualitätssicherungsschritt dem Auftraggeber übergeben werden.

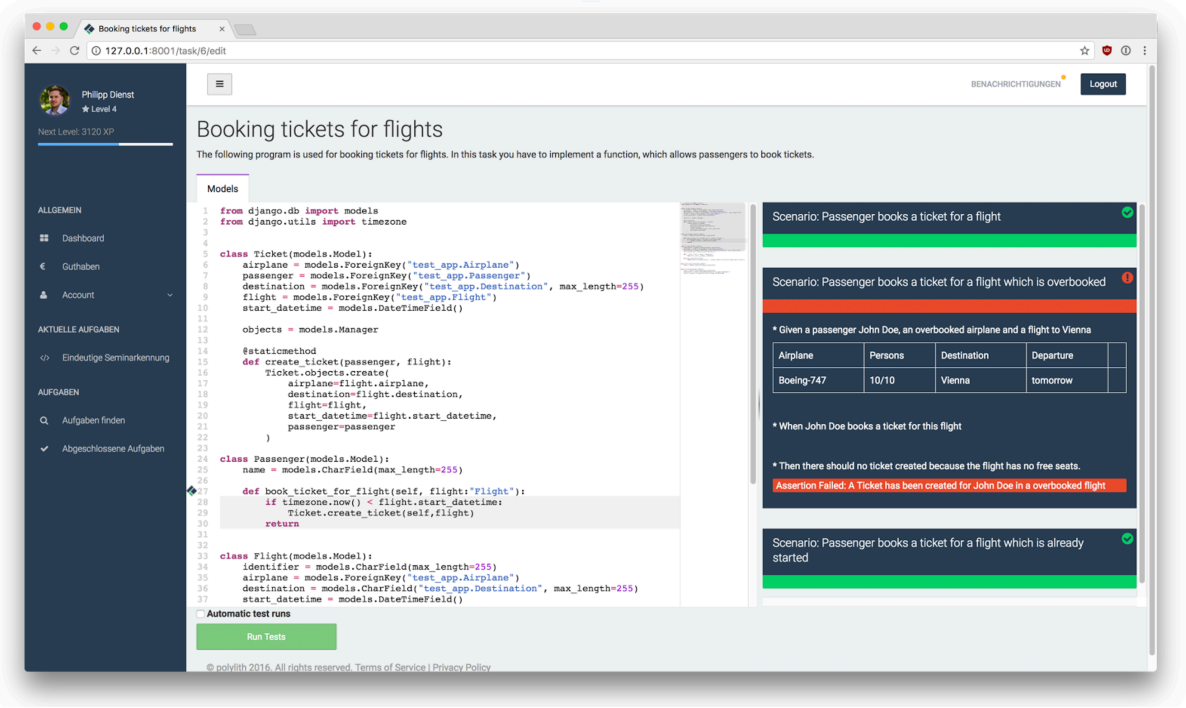

Abbildung 4.2.: Prototyp des Code-Editors der Plattform mit integrierten Hilfen, wie dem Ausblenden von (für diese Aufgabe) unnötigem Quellcode Bildquelle: Dienst (2017)

4.1.2. Crowd-Entwickler

Die Zielgruppe der Crowd-Entwickler der Plattform sind in erster Linie Informatikstudenten. Diese sind noch keine fertig ausgebildeten Fachkräfte, verfügen aber trotzdem schon über das nötige Fachwissen, um abgegrenzte Programmieraufgaben zu lösen. Der Code-Editor unterstützt diese bei der Lösung der Aufgabe, die Plattform verhindert gleichzeitig durch automatisierte Tests die Abgabe von fachlich falschem Quellcode.

Die Hauptzielgruppe der Plattform bildet somit eine Gruppe von Entwicklern, die nur wenig oder keine Praxiserfahrung haben. Dies erfordert ein Qualitätssicherungskonzept, durch das auch nicht-funktionale Qualität überprüft wird. Die Crowd-Entwickler brauchen weiterhin Feedback, um zu erfahren, an welchen Stellen Verbesserungsbedarf vorhanden ist. An dieser Stelle soll das Gamification-Konzept dieser Arbeit greifen. Einerseits soll dadurch der Spaß am Lösen der Programmieraufgaben erhöht werden, andererseits sollen die Fähigkeiten der Crowd-Entwickler spielerisch verbessert werden.

4.2. Einbettung der Arbeit in den Kontext

Zur Besserung Einordnung des Beitrags der Arbeit in den Kontext der polylith-Plattform erfolgt an dieser Stelle eine Erläuterung der Verbindungsstellen des *Developer Monitoring* und *Quality Rating Systems*. Sie wird durch die Abbildung 4.3 unterstützt.

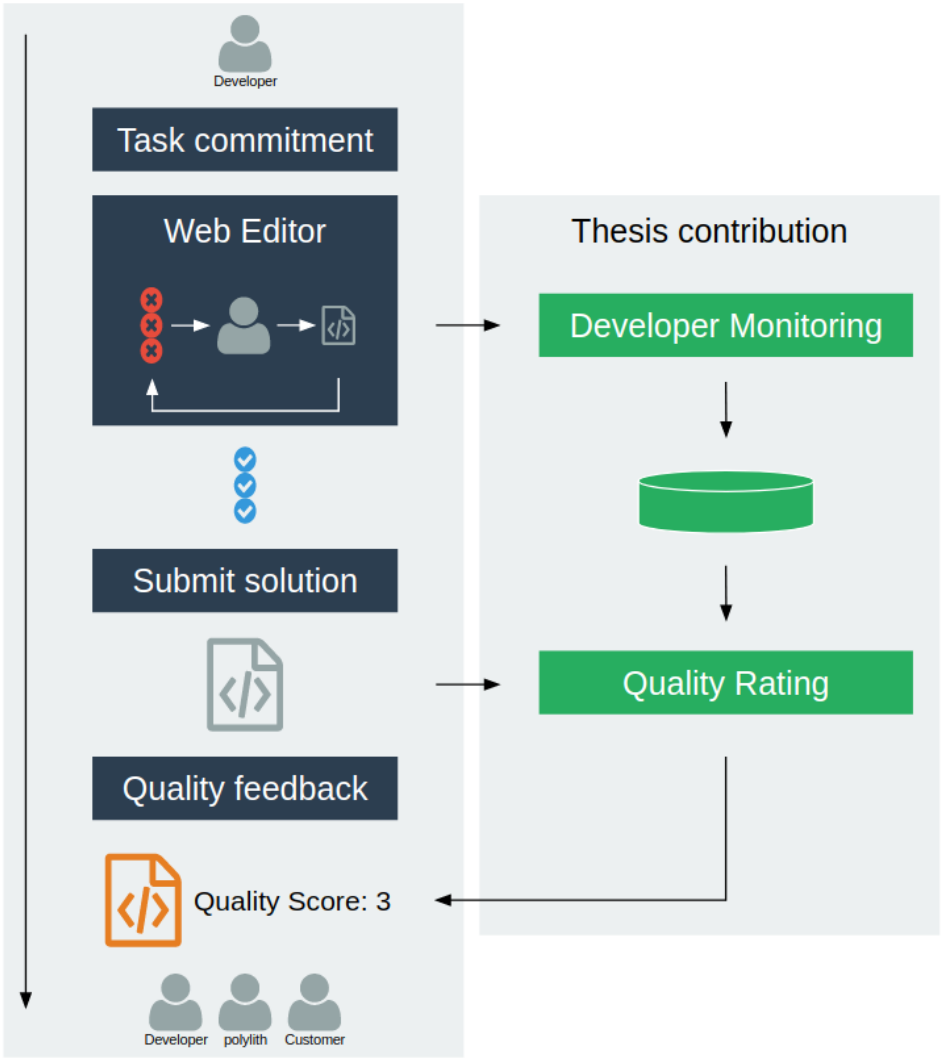

Abbildung 4.3.: Integration der in den Bearbeitungsprozess der Aufgabe - die Schritte des Plattformprozesses sind blau dargestellt, die zwei konzipierten und umgesetzten Systeme dieser Arbeit sind grün dargestellt

Nachdem ein Entwickler auf der Plattform eine Aufgabe angenommen hat, beginnt er diese in dem Web Editor der Plattform zu bearbeiten. Dabei arbeitet er nach einem auf die Plattform angepassten testgetriebenen Entwicklungsprozess, der im Unterschied zum Ursprünglichen schon alle Tests einer Aufgabe bereitstellt. Der Entwickler erfüllt die Tests in einem iterativen Prozess durch Bearbeitung des Quellcodes und Ausführen der Tests. Dies wiederholt er bis er alle Testfälle absolviert hat und kann anschließend die Aufgabe abgeben. Das Developer Monitoring System zeichnet jede aktive Interaktion, die der Entwickler von der Annahme bis zur Abgabe der Aufgabe durchführt, auf und erfasst somit den Prozess der

Implementierung der Aufgabe im Web Editor. Alle erfassten Daten werden in einem oder mehreren Datenbanksystemen gespeichert.
Nach der Abgabe des Ergebnisses soll der Entwickler, Plattformbetreiber und Kunde über die Qualität der Abgabe informiert werden. Diese ist nicht zwingend durch die fachlichen Tests erfüllt, da diese z. B. nicht die Lesbarkeit des Quellcodes erfassen. Dafür analysiert nach der Abgabe der Aufgabe das Quality Rating System die abgegebene Lösung und die aufgezeichneten Daten und bewertet die Qualität des abgegebenen Quellcodes.

5. Konzept

Im vorliegenden Kapitel werden die Konzepte für die Umsetzung der Anforderungen der beiden Problemstellungen *Developer Monitoring* und *Quality Rating* beschrieben. An erster Stelle werden die Konzepte vorgestellt, die für das Developer Monitoring erstellt wurden sind und an zweiter Stelle die für das Quality Rating.

5.1. Developer Monitoring

Das Developer Monitoring zeichnet den Prozess der Lösungsentstehung auf. Die aufgezeichneten Daten können für die Bewertung der Lösung, aber auch für die Bewertung des Entwicklers benutzt werden.
Abbildung 5.1 zeigt den konzipierten Aufzeichnungsprozess und die Informationen die dabei bezogen werden sollen. Wichtig dabei ist, dass der gesamte Entwicklungsprozess im Code Editor stattfindet (vgl. Kapitel 4). Dadurch ist es möglich alle Änderungen die am Quellcode gemacht werden zu erfassen. In erster Instanz sind das Tastatureingaben, die Zeichen zum Quellcode hinzufügen oder entfernen. Jedes eingegebene Zeichen wird aufgezeichnet und führt zu einem neuen Stand des Quellcodes. Außerdem werden besondere IDE-Aktionen aufgezeichnet. Darunter zählen Aktionen, wie das *Einfügen von Quellcode* oder von der IDE unterstützte *Refactorings*. Eine vollständige Liste der betrachteten Aktionen befindet sich im Anhang A.

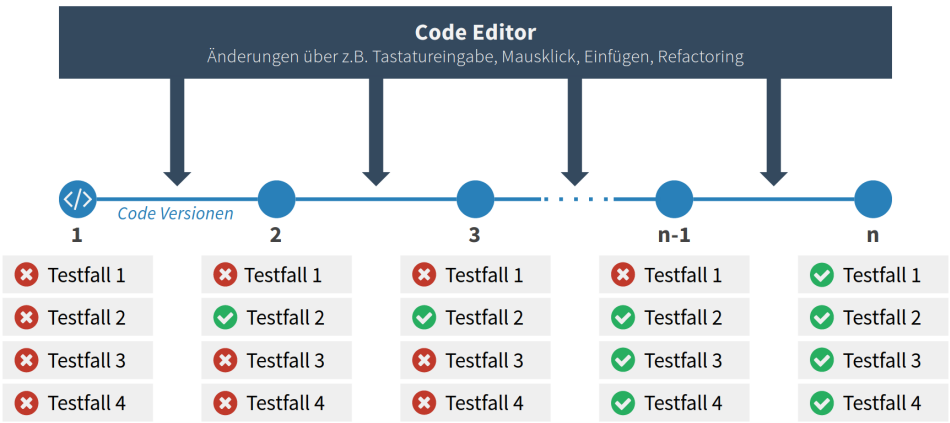

Abbildung 5.1.: Workflow des Developer Monitorings

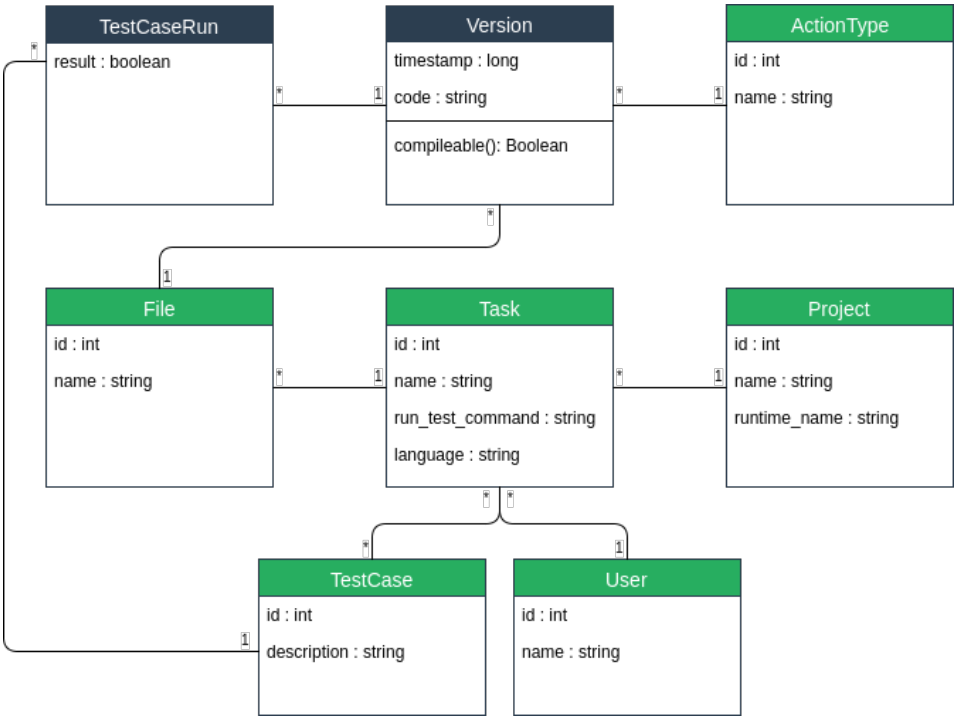

Abbildung 5.2.: Datenmodell des Developer Monitorings - die Rohdaten sind blau und die Kontextdaten grün hinterlegt

Führt der Entwickler also eine Aktion in der IDE durch, so führt diese auch direkt zu einem neuen Stand des Quellcodes im Sinne des Developer Monitorings. Dieser wird einem Testlauf unterzogen 5.1.3), sofern er kompilierbar und nicht identisch zum vorherigen Versionsstand ist. Zweiteres kann durch nicht-manipulierende Aktionen auftreten, die ebenfalls aufgezeichnet werden (vgl. nicht-manipulierende Aktionen im Anhang A). Dabei beinhaltet der Testlauf alle Tests-Szenarios, die zur Aufgabe gehören. Das Ergebnis der Tests wird zusätzlich zum aktuellen Stand des Quellcodes und zugehörigen Aktion gespeichert (vgl. Unterabschnitt 5.1.1). Der Vorgang startet, sobald der Nutzer anfängt die Aufgabe zu bearbeiten und endet, wenn der Nutzer seine Lösung einreicht.

5.1.1. Datenmodell

In diesem Unterabschnitt wird das dem Developer Monitoring zugrundeliegende konzeptionelle Datenmodell vorgestellt. Dabei werden Informationen, die gespeichert werden müssen, Entitäten und Attributen zugeordnet. Abbildung 5.2 visualisiert das konzeptionelle Datenmodell als UML-Klassendiagramm. Im Folgenden sind die Entitäten dieses Modells beschrieben:

Project Ein `Project` speichert die Informationen eines Projekts. Dabei definiert es die Laufzeitumgebung, die der Testausführung dient, über das `runtime_name`-Attribut. Außerdem hat jedes Projekt einen Namen (`name`) und einen Identifier (`id`). Zusätzlich wird ein Projekt mehreren Aufgaben (`Tasks`) zugeordnet.

Task Ein `Task` speichert die Informationen zu einer Aufgabe der Plattform. Neben dem Namen (`name`) und einem Identifier (`id`) speichert der Task ein Kommando im `run_test_command`-

Attribut, welches die Ausführung der Tests innerhalb der Laufzeitumgebung anstößt. Zusätzlich steht eine Aufgabe in Beziehung zu mehreren Dateien (`Files`), die innerhalb einer Aufgabe bearbeitet werden können, mehreren `TestCases`, die bestimmen wann eine Aufgabe abgeschlossen ist und einem `User`, der die Aufgabe bearbeitet.

User Die `User` Entität speichert den Identifier des Nutzers im Attribut `id` und seinen Namen im Attribut `name`.

TestCase Die `TestCase` Entität speichert den Identifier des Testfalls im Attribut `id` und eine Beschreibung im Attribut `description`.

File Ein `File` speichert die Informationen über eine Datei, die innerhalb einer Aufgabe bearbeitet werden kann. Sie hat einen Identifier (`id`) und einen Namen (`name`), welcher den gesamten Pfad innerhalb des zugehörigen Projektes beinhaltet. Ein File steht wiederum in Relation zu mehreren Versionen (`Versions`).

Version Die zentrale Entität des Developer Monitorings ist die `Version`. Sie fasst die Informationen eines Stands beim Entstehungsverlauf des Quellcodes zusammen. Der Zeitpunkt des Stands ist im `timestamp`-Attribut gefasst. Dieses Attribut ist gleichzeitig die eindeutige Versionsnummer eines Standes innerhalb einer Datei und Aufgabe, da es nicht vorgesehen ist, dass mehrere Aktionen innerhalb einer Millisekunde auftreten. Das `code`-Attribut speichert den Quellcode der Version. Wenn dieser syntaktisch korrekt ist, kann die Version einem Testlauf unterzogen werden. Die `compileable`-Methode bietet die Funktion, um genau dieses Kriterium zu überprüfen. Das Ergebnis des Testlaufs ist in der Referenz zu mehreren `TestCaseRuns` abgebildet. Die Referenz zur `ActionType`-Entität dient der Speicherung der Aktion, die zu diesem Versionsstand geführt hat.

ActionType Ein `ActionType` speichert den Identifier der Aktion im `id`-Attribut. Ein deskriptiver Name wird im `name`-Attribut gespeichert. Eine vollständige Liste der Aktionen die in dieser Arbeit aufgezeichnet werden befindet sich im Anhang A.

TestCaseRun Ein `TestCaseRun` speichert die Informationen eines ausgeführten Tests. Er hat eine Referenz zu genau einem Test, damit die Verbindung zum ausgeführten Testfall gespeichert ist. Das Testlaufergebnis wird als Boolean im `result`-Attribut gespeichert.

Die Entitäten sind in zwei Arten von Klassen eingeteilt. Auf der einen Seite die Rohdaten (blau), welche sich durch viele Instanzen einer Entität auszeichnen. Auf der anderen Seite die Kontextdaten (grün), welche den Kontext der Rohdaten widerspiegeln und Informationen zu ihrer Beschaffung beinhalten sowie wenige Instanzen einer Entität aufweisen. Geht man von einer Aktionsszahl von 10.000 Aktionen pro Aufgabe aus (vgl. *NF-DM-10*), dann kommen beispielsweise auf eine Instanz eines `Tasks` 10.000 `Version` Instanzen, da für jede Aktion ein Versionsstand gespeichert wird (die Verteilung der Aktionen auf die Dateien ist hier unerheblich).
Da vor allem bei den Rohdaten viele Instanzen der Entitäten generiert werden, ist es wichtig eine Betrachtung der auftretenden Datenmengen zu machen. Diese wird im folgendenden Kapitel beschrieben.

ID	Bezeichnung	Zeichen	Wert
NF-DM-10	Zeichenanzahl einer Lösung	solution_signs	5.000
NF-DM-20	Anzahl an Aktionen	actions	10.000
NF-DM-30	Tippgeschwindigkeit	actions_per_second	6
NF-DM-40	Anzahl paralleler Entwickler	parallel_developers	10

Tabelle 5.1.: Parameter aus den Anforderungen

5.1.2. Datenmengen und -durchsatz

Die Speicherfrequenz beim Developer Monitoring bestimmt die Datenmenge auf der die Analysealgorithmen arbeiten können und müssen. Das Konzept sieht vor jede IDE-Aktion aufzuzeichnen, die durch den Nutzer bei der Bearbeitung einer Aufgabe benutzt wird. So kann der Entstehungsverlauf des Quellcodes vollständig (in Bezug auf die ausgeführten Aktionen) rekonstruiert werden. Zwar hat nicht jede Aktion einen Einfluss auf die resultierende Lösung, allerdings sind sie trotzdem ein Abschnitt in der Entstehung und enthalten Informationen über die Arbeitsweise des Entwicklers.

Für die Wahl des Datenbanksystems sind vor allem die auftretenden Datenmengen wichtig, die aus dieser konzeptionellen Entscheidung resultieren. Aus diesem Grund wird in diesem Kapitel die entstehende Datenmenge der Rohdaten bei der Bearbeitung einer Aufgabe berechnet. Dazu muss die Datenmenge aller Instanzen aufsummiert werden, wozu wiederum die Summe der Datenmenge aller Attribute berechnet werden muss. Einige Parameter, die für diese Berechnung benötigt werden, können aus den Anforderungen der polylith-Plattform an das Developer Monitoring entnommen werden (vgl. Tabelle 5.1).

Das `code`-Attribut der Entität `Version` hat dabei den größten Anteil an der Datenmenge (vgl. Verhältnis zu den restlichen Attributen im Anhang B) und lässt sich unabhängig vom verwendeten Datenbanksystem berechnen. Durch den signifikanten Einfluss auf die entstehenden Datenmengen wird im Folgenden deren Berechnung erläutert.

Da für jede Aktion ein neuer Versionsstand gespeichert wird, existieren nach der Bearbeitung der Aufgabe so viele Versionsstände, wie Aktionen durchgeführt wurden. Die Summe der Datenmenge des Attributes `code` aller Versionsinstanzen ergibt die resultierende Datenmenge (vgl. Gleichung 5.1).

$$data = \sum_{n=0}^{actions} data\_of\_code_n \tag{5.1}$$

Die Größe des Quellcodes ändert sich mit jeder Aktion und wächst bis zur Zeichenanzahl `solution_signs` an. Im Diagramm aus Abbildung 5.3 sind drei mögliche Wachstumsarten des Quellcodes eingezeichnet. Im *best-case* (bzgl. der entstehenden Datenmenge) werden vom Entwickler nur Aktionen durchgeführt, die den Quellcode zwar verändern er aber nie mehr als 1 Zeichen beinhaltet. Mit der letzten Aktion wird dann der gesamte Lösunsquellcode einfügt. Im *worst-case* wird mit der ersten Aktion der gesamte Lösungsquellcode eingefügt und anschließend wird dieser nurnoch minimal bearbeitet. Der *average-case* ist als wahrscheinlichster einzuordnen und wird deshalb für die Berechnung des Wachstums verwendet. Das lineare Wachstum in diesem Fall ähnelt dem Schreiben von Quellcode am ehesten. Die Gleichung 5.1

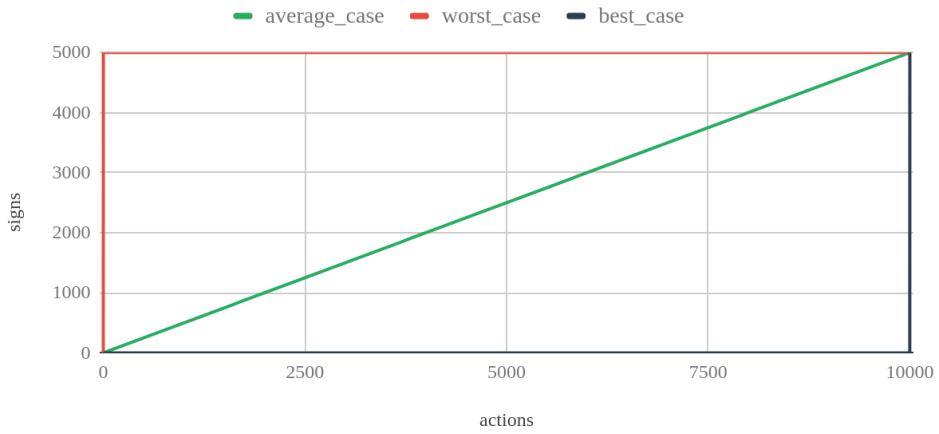

Abbildung 5.3.: Zeichenwachstum in Abhängigkeit zu den Aktionen (average-, worst- und best-case

lässt sich unter der Annahme des linearen Wachstums mit der Gleichung 5.2 ersetzen.

$$data = \sum_{n=0}^{actions} (n \cdot \frac{solution\_signs}{actions}) \tag{5.2}$$

Durch Ersetzen der Variablen mit den bekannten Werten aus den gestellten Anforderungen ergibt sich folgendes Ergebnis:

$$\begin{aligned} data &= \sum_{n=0}^{10.000} (n \cdot \frac{5.000}{10.000}) \\ &= 25.002.500B \\ &\approx 25MB \end{aligned} \tag{5.3}$$

Damit beträgt die Datenmenge des `code`-Attributs ca. **25 MB** pro aufgezeichneter Aufgabe. Diese Abschätzung ist unter der Annahme des linearen Zeichenwachstums eine Abschätzung nach oben, da die Werte aus den Anforderungen Maximalwerte darstellen mit denen das Developer Monitoring noch funktionieren soll.
Ebenfalls interessant für die Wahl des Datenbankystems ist die Menge an Daten die pro Sekunde gespeichert werden müssen. Dafür wird ebenfalls eine Abschätzung nach oben durchgeführt indem der Datendurchsatz kurz vor dem Abschluss einer Aufgabe betrachtet wird. Die gespeicherten Zeichen pro Aktion liegen ungefähr bei der maximalen Anzahl an Zeichen pro Aufgabe. Der Durchsatz ergibt sich weiterhin aus der Eingabegeschwindigkeit des Quelltextes.

$$throughput = actions\_per\_second \cdot solution\_signs \cdot 1B \tag{5.4}$$

Die Tippgeschwindigkeit kann an dieser Stelle mithilfe von best- und worst-case Werten aus der Studie Card u. a. (1980) abgeschätzt werden. In dieser Studie wurde u. a. die Eingabeperformance beim Tippen auf der Tastatur der Nutzer in verschiedenen Szenarien getestet. Unser best-case tritt demnach beim *Schreiben von komplexem Code* mit einer Tippgeschwindigkeit von ca. 1,3 Zeichen/s auf. Der worst-case tritt bei der *besten Schreibkraft* mit einer Tippgeschwindigkeit von ca. 12,5 Zeichen/s auf (Card u. a. 1980). Für die weitere Berechnung wird an dieser Stelle der aufgerundete Mittelwert dieser beiden Szenarien gewählt, da es sich beim

Developer Monitoring zwar um das Schreiben von Quellcode handelt, welches aber nicht nur richtige Tastenanschläge aufzeichnet (nur diese wurden in der Studie gezählt) und eine Abschätzung nach oben stattfinden soll, damit das Datenbanksystem nicht unterdimensioniert ist. Weiterhin ist das Auftreten des Worst-case im Durchschnitt als unwahrscheinlich Einzustufen, da es sich um das Abschreiben von Fließtext durch einen sehr schnell tippenden Nutzer handelt. Damit beträgt die angenommene Tippgeschwindigkeit ca. **6 Zeichen/s**. Mithilfe von Formel 5.4 wird der Wert des Datendurchsatzes `throughput` ermittelt. Nach Einsetzen der Daten erhält man folgendes Ergebnis:

$$\begin{aligned} throughput &= \frac{6}{1s} \cdot 10.000 \cdot 1B \\ &\approx 60KB/s \end{aligned} \tag{5.5}$$

Damit beträgt der Datendurchsatz **60 KB/s**. Diese Zahl wird durch das vorgesehene parallele Arbeiten von bis zu 10 Entwicklern nochmals multipliziert und erhöht sich auf ca. **0,6 MB/S**.

5.1.3. Architekturkonzept

Das Architekturkonzept des Developer Monitorings sieht die folgenden 4 großen Komponenten zur Umsetzung der geforderten Funktionalitäten vor.

Version Database

Die Version Database speichert die beschriebenen Rohdaten des Developer Monitorings. Sie ist dabei auf viele Instanzen der Entitäten hin optimiert, damit diese effizient gespeichert und durch die Analysealgorithmen effizient abgerufen werden können. Das gewählte Datenbanksystem sollte die auftretenden Datenmengen, die im Unterkapitel 5.1.2 berechnet wurden, bewältigen können. Für die Auswahl des Datenbanksystems ist ebenfalls wichtig, dass der *Test Runner Service* und *Developer Monitoring Service* verteilt betrieben werden und deshalb von verschiedenen Servern auf die Version Database zugreifen. Der Zugriff findet sowohl lesend, als auch schreibend statt.

Context Database

Die Context Database speichert alle Daten, welche den Kontext der Rohdaten bestimmen, aber für die eigentliche Aufzeichnung des Entwicklungsprozess' unerheblich sind. Zum Beispiel hat jede Aufgabe einen Namen und ist einer Sprache zugeordnet. Diese Attribute sind allerdings für den Aufzeichnungsprozess nicht von Bedeutung. Diese Daten folgen außerdem anderen Anforderungen als die Rohdaten. So existieren wesentlich weniger Instanzen einer Entität, als das bei den Daten der Version Database der Fall ist. Weiterhin werden die Daten der Context Database benutzt, um die eigentlichen Rohdaten aus der Version Database während und nach der Analyse in ihren Kontext zu setzen. Die Last liegt hier in den Lese-Zugriffen, die auch parallel ohne Konflikte durchgeführt werden können. Aus diesen beiden Gründen ist eine Speicherung in einem separaten Datenbanksystem vorgesehen.

Runtime Registry

In der Runtime Registry sind alle Laufzeitumgebungen der aktuellen Projekte als Artefakte gespeichert. Über eine Schnittstelle ist es möglich diese Laufzeitumgebungen auf den Server herunterzuladen und die Tests innerhalb der Laufzeitumgebung auszuführen.

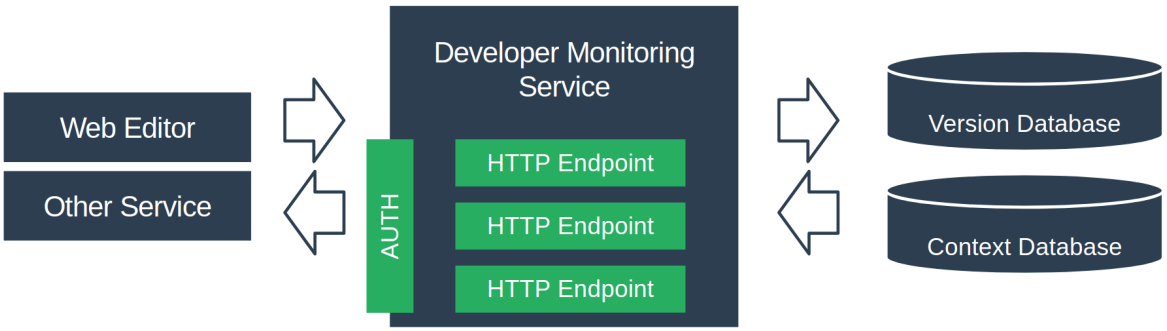

Abbildung 5.4.: Übersicht über den Developer Monitoring Service

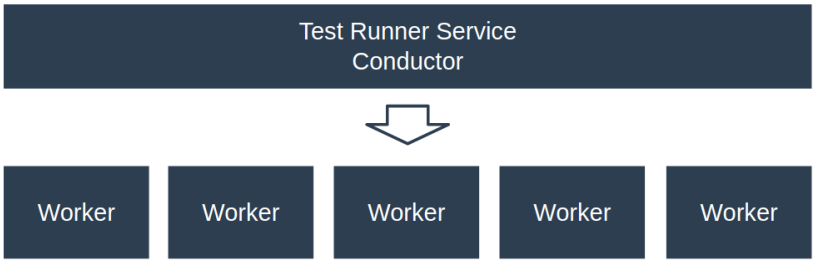

Abbildung 5.5.: Test Runner Service mit Conductor und Worker zur Lastenverteilung

Developer Monitoring Service

Der Developer Monitoring Service ist die Schnittstelle zwischen den gespeicherten Daten beim Developer Monitoring und anderen Systemen, die diese Daten für eine weitere Analyse benutzen (vgl. Abbildung 5.4). Da der Editor über HTTP kommuniziert, wird der Developer Monitoring Service als Webservice konzeptioniert. Er nimmt die serialisierten Daten (z. B. Versionsstand, Aktion, Testergebnis) entgegen, deserialisiert diese und speichert sie in der Version Database ab. Er entkoppelt dadurch das Schreiben und Lesen der Daten von der Datenbanktechnologie, was auch im Hinblick auf die Fusion der Kontext- und Rohdaten wichtig ist. Auch die Absicherung der Daten kann unabhängig vom Datenbanksystem geschehen.

Test Runner Service

Der Test Runner Service dient der verteilten Ausführung der Testläufe. Dafür teilt sich der Test Runner Service in einen Conductor und mehrere Worker auf. Der Conductor nimmt dabei die Anfrage an einen Testlauf für einen Versionsstand entgegen und verteilt ihn auf einen Worker mit freier Kapazität. Er spielt also die Rolle des Dirigenten in einem klassischen Orchestrierungs-Anwendungsfall, wobei die Ausfallsicherheit des Conductors für das Konzept zu vernachlässigen ist. Der Fokus liegt auf der effizienten Verteilung der Last bei der Testausführung, da diese für jede Aktion die zu einem syntaktisch korrekten Stand führt vom Developer Monitoring Service angefordert werden. Der Worker übernimmt die eigentliche Ausführung des Testlaufs. In Abbildung 5.5 ist dieser allgemeine Workflow abgebildet.
In Abbildung 5.6 ist dargestellt, wie der Datenfluss des Workers aussieht. Bevor der Worker Tests ausführen kann muss er initialisiert werden, was das Beziehen des Runtime Namens von der Context Database und das Herunterladen der Laufzeitumgebung von der Runtime

Abbildung 5.6.: Datenfluss eines Workers bei der Initialisierung und Testausführung

Registry beinhaltet. Anschließend wird bei einer Anweisung die Tests für einen Versionsstand auszuführen der aktuelle Inhalt der geänderten Datei von der Context Database bezogen. Danach kann der Testlauf gestartet, nach Abschluss das Testergebnis von der Standardausgabe geparst und zurück in die Version Database geschrieben werden.

5.2. Quality Rating System

Wie im Kapitel 2 herausgearbeitet, sind die verwandten Arbeiten zum maschinellen Lernen von Codequalität für die einmalige Analyse aufgebaut und zielen z. B. darauf ab einen Zusammenhang zwischen der Fehleranfälligkeit der Codebasis mit verschiedenen Metriken zu erkennen. Weiterhin finden Lernalgorithmen in aktuell verfügbaren Werkzeugen keine Anwendung, da Softwareprojekte in ihrer Ausprägung sehr verschiedenartig und dadurch wenig vergleichbar sind. Die Trainingsphase müsste daher für jedes Projekt neu durchgeführt werden, um wenigstens innerhalb dieses Projekts genaue Qualitätsbewertung durchführen zu können.
Das Konzept dieser Arbeit sieht vor, Lernalgorithmen in dem *Quality Rating System (QRS)* zur Qualitätsbewertung von Quellcode umzusetzen, da die zugrunde liegenden Quellcodes durch den Kontext der polylith-Plattform eingeschränkt sind. Im Gegensatz zu der Bewertung der Codebasis eines kompletten Projekts beschränkt sich die Bewertung innerhalb der Plattform auf kleine Aufgaben, deren Lösung vollständig getestet ist und ihr Entstehungsverlauf durch das Developer Monitoring nachvollzogen werden kann. Die Hypothese dieser Arbeit ist, dass dadurch die Vielfältigkeit der zu bewerteten Daten stark eingeschränkt wird und Trainingsdaten im gesteckten Rahmen besser wiederverwendet werden können. Damit diese Hypothese validiert werden kann ist es notwendig den QRS so umzusetzen, dass schnelles Prototyping von Quellcodebewertung im Kontext der polylith-Plattform möglich ist.
Abbildung 5.7 gibt eine Übersicht über den QRS und dessen Bewertungsvorgang des Quellcodes. Dabei werden die drei Ebenen *Processing*, *Data* und *Configuration* beschrieben. Als Eingabe für den QRS dienen im Allgemeinen die Daten des Developer Monitorings. Sie beinhalten sowohl die eingereichte Lösung, als auch ihren Entstehungsverlauf. Orthogonal zu den drei Ebenen sind die Prozessschritte *Training*, *Learning* und *Rating* dargestellt, welche für das unterstützte maschinelle Lernen obligatorisch sind.
Für den Trainingsschritt ist es notwendig, dass die Klassenmodelle und deren Klassen konfiguriert und selektiert werden. Das heißt, dass der Qualitätsbeauftragte definieren muss, mit welchen Qualitätsaspekten er die Lösung bewerten möchte. Im Bild wurden beispielsweise die

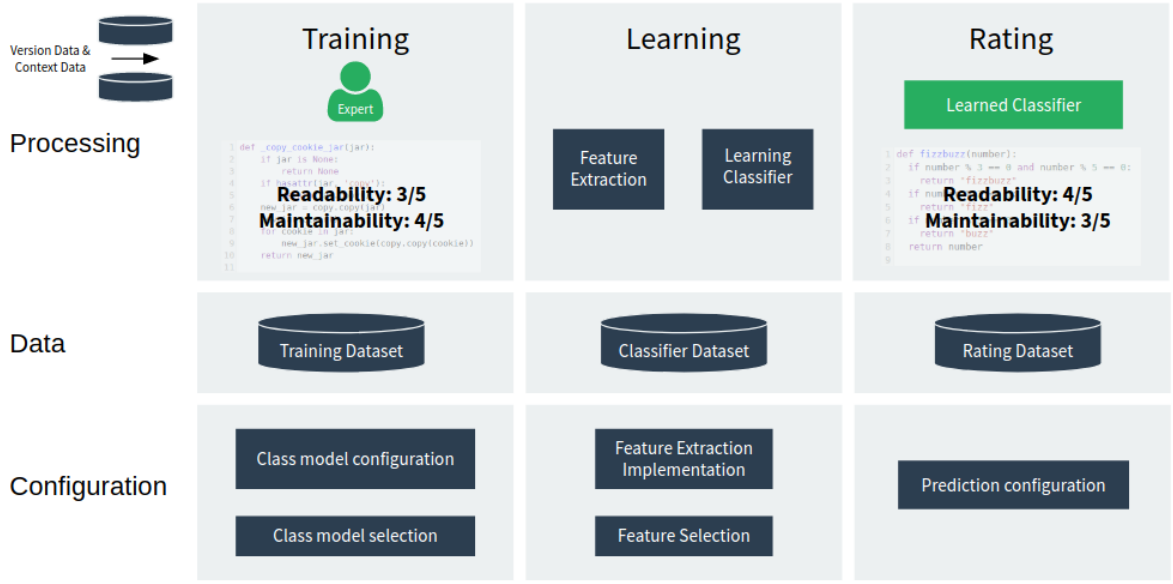

Abbildung 5.7.: Überblick über den Quality Rating System Prozess, die Datensätze und Konfigurationen

Readability und *Extendability*, jeweils mit den Klassen 1-5 ausgewählt. Mit steigender Klasse wurde der Qualitätsaspekt besser umgesetzt. Innerhalb des Trainingsschrittes übernimmt ein Experte die Rolle des Trainers und bewertet Codeabschnitte anhand der konfigurierten Klassenmodelle. Die Zuordnung von Trainingsinstanz zur Bewertung durch den Experten wird im Trainingsdatensatz gespeichert.

Für den zweiten bzw. Lernschritt muss konfiguriert werden, welche Metriken als Features benutzt werden und welcher Classifier benutzt wird. Die selektierten Features werden anschließend aus den Daten des Developer Monitorings berechnet und dienen zusammen mit dem Trainingsdatensatz aus dem vorherigen Prozessschritt als Eingabe für das Anlernen des ausgewählten Classifiers. Nachdem der Classifier angelernt wurde, wird das erstellte Bewertungsmodell im Classifier-Datensatz gespeichert. So kann der Classifier später wieder hergestellt und mit anderen verglichen werden.

Das Resultat der Lernphase ist der angelernte Classifier, welcher in der dritten bzw. Bewertungsphase verwendet wird, um Bewertungen von Quellcodeabschnitten ohne menschliche Hilfe vorzunehmen. Er ist in grün dargestellt, da er den Experten ersetzt, der in der Trainingsphase die Klassifizierung vorgenommen hat. Die Ergebnisse der Bewertung sind im *Rating Dataset* gespeichert. In dieser Phase ist es außerdem möglich, dass die Bewertungsgenauigkeit des Classifiers mithilfe eines Testdatensatzes evaluiert wird. Konfigurationen, wie die Klassenmodelle die bewertet werden sollen, können ebenfalls vorgenommen werden.

Im Folgenden werden die Komponenten konzeptionell beschrieben, die für die Umsetzung dieses Prozesses notwendig sind. Der *Training Service* wurde für den Trainingsschritt konzipiert. Der *Core Learning Service* und *Feature Extraction Service* setzten zusammen den Lern- und Bewertungsschritt um.

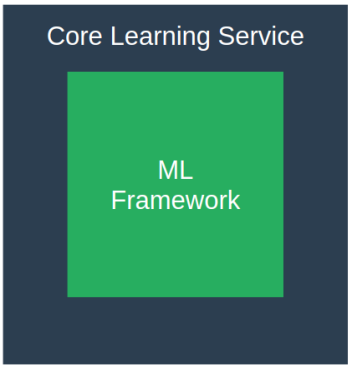

Abbildung 5.8.: Core Learning Service als Wrapper des Machine-Learning Frameworks

5.2.1. Core Learning Service

Im *Core Learning Service* werden die einzelnen Anfragen an ein *Machine-Learning Framework* weitergeleitet. Die Implementierung von Machine-Learning Algorithmen ist oft sehr komplex, weshalb Frameworks existieren, die die mathematischen Grundlagen und Berechnungen hinter einer API verstecken und dadurch eine einfache und schnelle Anwendung auf fachlicher Ebene ermöglichen.
Wie in Abbildung 5.8 dargestellt ist der Core Learning Service also ein Wrapper der Anfragen in unserem Kontext auf die API eines Machine-Learning Frameworks übersetzt. Beispielsweise ist es für das Antoßen des Lernschrittes aus den Trainingsdaten notwendig, diese in das Datenformat des Frameworks zu übersetzen, damit es diese vearbeiten kann.

5.2.2. Feature Extraction Service

Der *Feature Extraction Service (FES)* ist für die Berechnung der Metriken aus dem Quellcode zuständig. Für die Aufgabe der Berechnung von herkömmlichen Metriken[1] existieren allerdings schon Werkzeuge, wie SonarQube, welches im Abschnitt 3.2.2 beschrieben wurde.
Der FES ermöglicht einerseits die Auslagerung der Berechnung, indem er den Abgabe-Quellcode zur Berechnung an das zuständige Werkzeug weiterleitet und die berechnete Metrik als Rückgabewert entgegen nimmt. Dieser Prozessschritt ist an dieser Stelle bewusst sehr allgemein bzw. abstrakt konzipiert, da die Werkzeuge und deren Workflow sich stark voneinander unterscheiden. Auf der anderen Seite stellt der FES Funktionen für die Berechnung der Metriken aus dem Developer Monitoring bereit. Dieser Prozess ist wiederum unabhängig von dritten Werkzeugen und lässt sich in drei feste Prozessschritte unterteilen:

1. Beziehen der Kontextdaten für die Aufgabe
2. Beziehen der Rohdaten für die Aufgabe
3. Berechnen der Metrik aus den Daten

Nachdem diese Schritte durchgeführt wurden kann die Metrik an den Service übergeben werden der die Berechnung der Metrik angefordert hat. Die API für das Beziehen einer Metrik wird durch den FES somit vereinheitlicht und ist für die ausgelagerte und eigene Berechnung gleich (vgl. Abbildung 5.9).

[1] wie z. B. Cyclomatic Complexity, Cognitive Complexity, Halstead-Metrik, Lines of Code

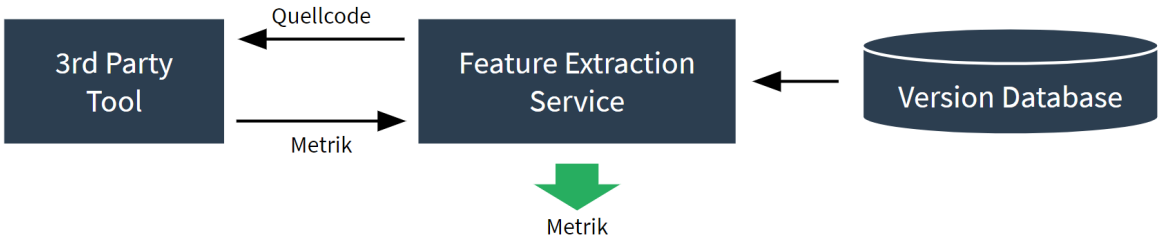

Abbildung 5.9.: Feature Extraction Service Datenfluss

Abbildung 5.10.: Struktur-Mockup der Trainings Oberfläche

Durch die Konfiguration der verwendeten Features ist es möglich nur eine Teilmenge der zur Verfügung stehenden in die Bewertung mit einfließen zu lassen. Dies bietet Vorteile bei dem Prototyping mit einzelnen Features, um deren Einfluss auf die Bewertung zu evaluieren. Im Abschnitt 2.1 wurde allerdings darauf eingegangen, dass sich Qualität strukturell definieren und in verschiedene Charakteristiken unterteilen lässt, die zusammen Qualität von Software und Quellcode ausmachen. Damit alle Charakteristiken in die Bewertung mit einfließen ist es von Vorteil, dass zu jeder dieser Metriken selektiert werden.

5.2.3. Training Service

Der *Training Service* ermöglicht die einfache Durchführung des Trainingsschrittes. Dafür stellt er eine Oberfläche zur Bewertung durch den Experten bereit. In Abbildung 5.10 ist ein Mock-Up dieser Oberfläche abgebildet. Die Bewertungsgrundlage die in einem Textfeld angezeigt wird, ist der Lösungsquellcode. Das Textfeld ermöglicht für die jeweilige dargestellte Sprache die Hervorhebung der Syntax. Links neben dem Textfeld werden Kontextinformationen zur Aufgabe angezeigt, welche für die Bewertung von Bedeutung sind. Die Klassenmodelle die der Experte bewerten muss sind rechts vom Textfeld platziert. Dabei sind die Klassenmodelle untereinander dargestellt und die Klassen eines Modells nebeneinander. Außerdem hat der Bewertende die Möglichkeit Kommentare in einem extra dafür vorgesehenen Feld unter dem angezeigten Quellcode anzugeben.

5.3. Zusammenfassung

Insgesamt wurden in diesem Kapitel jeweils Konzepte für die Umsetzung der Anforderungen an ein Developer Monitoring und das Quality Rating vorgestellt und nacheinander beschrieben. Das allgemeine Prozesskonzept des Developer Monitoring sieht vor die Implementierung aufzuzeichnen, indem für jede durchgeführte Aktion im Web Editor ein neuer Versionsstand angelegt wird. Sofern dieser nicht identisch mit dem vorherigen und syntaktisch korrekt ist wird er einem Testlauf unterzogen, der ebenfalls gespeichert wird. Die Aufzichnung erfolgt strukturiert und baut dem vorgestellten konzeptionellen Datenmodell auf. Dieses unterteilt sich in die beiden Klassenarten Roh- und Kontextdaten, wobei die Rohdaten den Großteil der entstehenden Datenmenge ausmachen. Diese kann mit ungefähr 27 MB pro getrackter Aufgabe abgeschätzt werden. Im Architekturkonzept wurden die fünf Komponenten, die den Zielprozess umsetzen vorgestellt. Sie bestehen aus drei Datenbanken (Context Database, Version Database, Runtime Registry) und zwei Services (Developer Monitoring, Test Runner).
Für das Quality Rating System wurde ebenfalls an erster Stelle der Zielprozess konzipiert, welcher die drei Schritte Training, Learning und Rating beinhaltet. Zur Umsetzung des ersten Schrittes dient der Training Service, welcher eine Oberfläche für das Training durch einen Experten bereitstellt. Die Learning und Rating Phase werden durch den Core Learning und Feature Extraction Service umgesetzt.

6. Umsetzung

In diesem Kapitel wird die Umsetzung der beiden definierten Prozesse aus dem Konzeptkapitel vorgestellt. An erster Stelle wird das allgemeine Vorgehen bei der Implementierung und die Auswahl der Technologien beschrieben. Anschließend werden wie schon in den vorangegangenen Kapiteln auch in diesem die beiden Prozesse *Developer Monitoring* und *Quality Rating* separat betrachtet.

6.1. Vorgehen und Programmiersprache

Für schnelles Feedback bei der Implementierung wurden einzelne Konzepte und Prozesse des *Extreme Programming* angewendet. Auf der einen Seite konnte für einzelne Problemstellungen durch sogenannte *Architectural Spikes* schnell ein Implementierungskonzept erarbeiten werden. Auf der anderen Seite ermöglichte der Fokus durch *vertikale Prototypen* einen Durchstich bei der Umsetzung. Vor allem durch die Anwendung des *Timeboxings*, konnten die konzipierten Features möglichst minimal und so schnell wie möglich umgesetzt werden. In der agilen Softwareentwicklung schreibt das Timeboxing vor ein Feature in einem festen Zeitfenster umzusetzen. Das ist in einer Forschungsarbeit wie dieser hilfreich, da so die Wahrscheinlichkeit für das Auftreten von *Over-Engineering* einzelner Komponenten minimiert wird.
Die einzelnen Funktionen wurden mithilfe der testgetriebenen Entwicklung umgesetzt. Anforderungen werden so in Form von Softwaretests definiert und die eigentliche Implementierung kann dann gegen die erstellten Tests stattfinden. Das Vorgehen ermöglicht den Fokus auf eine minimale Implementierung, welche die Tests erfüllt.
Eine weitere Designentscheidung war die Umsetzung der Services als eigenständige Services. Das heißt, dass sie den Prinzipien der in Fowler (2014) definierten Prinzipien der *Microservices* folgen. Sie sind danach u. a. entlang von Businessprozessen gekapselt, unabhängig voneinander installierbar, laufen in eigenständigen Prozessen und kommunizieren über "einfache" Pipelines. Durch die strikten Grenzen zwischen den Mircoservices, sind die einzelnen Businessprozesse horizontal skalierbar, was einen Vorteil bringt sobald die Grenze überschritten wird, dass ein Server zur skalierung des Systems ausreicht. Ein weiterer Vorteil tritt bei der Entwicklung der Microservices in Erscheinung. Große Teams können in kleine Teams auf die einzelnen Microserices aufgeteilt werden und größtenteils unabhängig voneinander entwickeln, was bei einer monolithischen Anwendung schwieriger ist.
Als Programmiersprache für die implementierten Services wurde Python in der Version 3.5 verwendet. Die Entscheidung für Python wurde vor allem aufgrund der Verfügbarkeit vieler

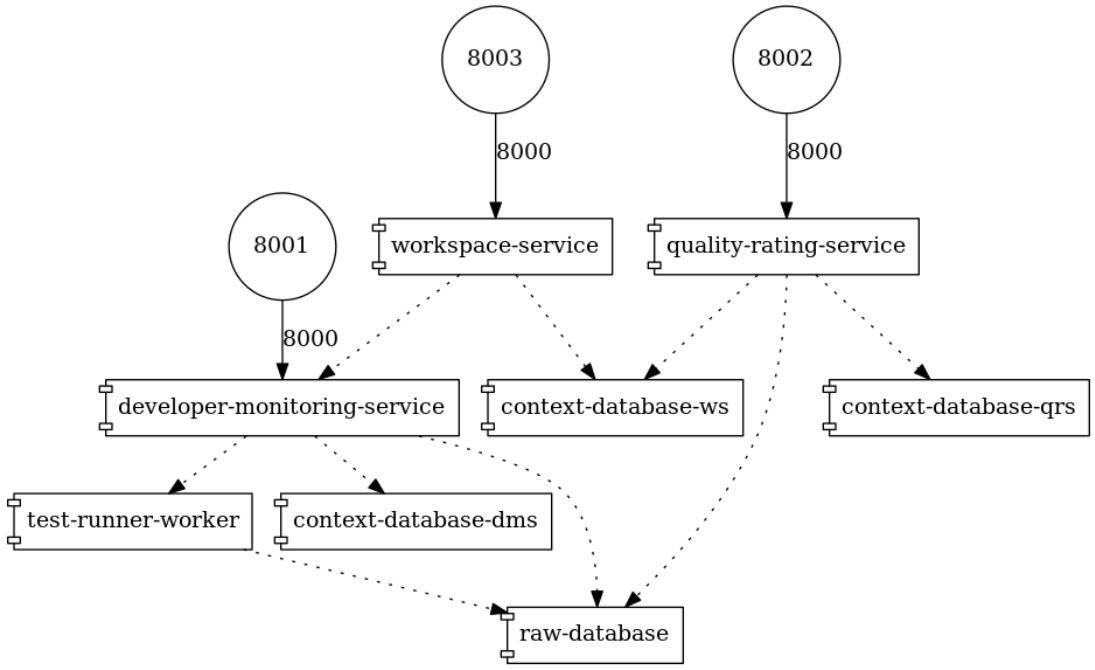

Abbildung 6.1.: Architekturübersicht aller vier Services und vier Datenbanken

Frameworks u. a. für Machine-Learning Anwendungen getroffen.

6.2. Service Architektur

Der folgende Abschnitt gibt einen Überblick über die umgesetzten Services der beiden Zielprozesse *Developer Monitoring* und *Quality Rating*. Die Kapselung der einzelnen Services wurde mit der Technologie *Docker* durchgeführt, da sie sich zur Bereitstellung einer Anwendung als Artefakt unabhängig von der verwendeten Programmiersprache und des Frameworks gut eignet. Sie hat sich mittlerweile zum Standard der Containertechnologien entwickelt. Darauf aufbauend wurde *Docker Compose* zur Definition der Startkonfiguration benutzt.
Abbildung 6.1 visualisiert die erstellte Docker Compose Konfigurationsdatei. Sie bildet die Container als Rechtecke ab, deren veröffentlichte Ports mithilfe der Kreise und ihre Beziehungen untereinander (definiert durch den Parameter `depends_on`) als gestrichelten Pfeil. Services und Datenbanken werden in dieser Abbildung nicht unterschieden, da sie für Docker Compose beides Services sind, die in Containern betrieben werden.
Die drei Services `developer-monitoring-service`, `test-runner-worker` sowie `quality-rating-service` wurden im Rahmen dieser Arbeit implementiert. Der `workspace-service` wurde nur ergänzt und ist ebenfalls abgebildet, um die Schnittstelle in die polylith-Plattform zu verdeutlichen. Der `workspace-service` ruft den `developer-monitoring-service` auf, welcher wiederum den `test-runner-worker` aufruft. Zusammen setzen die beiden letzteren den Developer Monitoring Prozess um. Der `quality-rating-service` setzt alleine den Quality Rating Prozess um. Außer dem `test-runner-worker` stellen alle Services ihre Web-Schnittstelle über den Port 8000 bereit,

Service o. Datenbank	Technologie	Version
`developer-monitoring-service`	Django	1.11
`quality-rating-service`	Django	1.11
`test-runner-worker`	Flask	0.12
`context-database-ws`	PostgreSQL	9.6
`context-database-dms`	PostgreSQL	9.6
`context-database-qrs`	PostgreSQL	9.6
`version-database`	CouchDB	1.6
`runtime-registry`	Docker Registry	2.6

Tabelle 6.1.: Verwendete Frameworks bzw. Technologien für die Services und Datenbanken

welcher für die Eindeutigkeit auf jeweils einen anderen Host-Port gemappt wird.
Alle drei implementierten Services arbeiten auf der `version-database`. Der `developer-monitoring-service` und `quality-rating-service` arbeiten zusätzlich auf den `context-databases`. Im Gegensatz zur `version-database` ist diese in voneinander getrennte Datenbanksysteme, `context-database-ws`, `context-database-dms` und `context-database-qrs`, aufgeteilt.

6.2.1. Service Technologien

Die Tabelle 6.1 gibt eine Übersicht über die verwendeten Frameworks zur Umsetzung der Services und Datenbanken.
Für die Services, die einen Teil der verarbeitenden Daten in einem relationalen Datenbanksystem speichern und eine Web-Schnittstelle bereitstellen, wird Django als Framework verwendet. Dies ist beim `developer-monitoring-service` und `quality-rating-service` der Fall. Die verwendete Version 1.11 ist die aktuellste stabile Version zum Zeitpunkt der Implementierung.
Flask wurde als Framework für den `test-runner-worker` verwendet, da er nur eine Web-Schnittstelle bereitstellen soll, ohne mit einem relationalen Datenbanksystem zu arbeiten. Es bietet daher weniger Optionen, benötigt aber weniger Ressourcen (z. B. Rechenleistung oder Arbeitsspeicher) während des Betriebs. Die Version von Flask ist die aktuellste, die zum Zeitpunkt der Implementierung verfügbar war.
Als relationale Datenbank wurde PostgreSQL in der aktuellsten Version benutzt, da PostgreSQL häufig als Standard in Zusammenhang mit Django verwendet wird. Darüber hinaus gab es keine Anforderungen an die relationale Datenbank, die zur Entscheidung beitrugen. Zur Speicherung der Laufzeitumgebungen bietet das Docker Ökosystem eine eigene Technologie - die Docker Registry - an. Sie ist der Standard, wenn es um die Speicherung und Bereitstellung von Docker Images geht.
Als nicht-relationale Datenbank wurde CouchDB verwendet. Für die Wahl von CouchDB sprachen vor allem die Speicherung der Daten im JSON Format, da JSON auch als Format bei der Kommunikation zwischen den einzelnen Services benutzt wird und so weniger Transformationen stattfinden müssen. Außerdem besitzt CouchDB eingebaute Unterstützung für *MapReduce*-Jobs, welche in nicht relationalen Datenbanken für die Analyse benutzt werden. Dadurch ist eine einfache Anwendung dieses Programmiermodells möglich. Ein verteilter Betrieb über mehrere Rechner wird ebenfalls von CouchDB unterstützt, was eine horizontale Skalierung bei einer großen Menge an Daten oder Anfragen ermöglicht. Die Version 1.6 wurde ausgewählt, da in dieser Version ein offiziell unterstütztes Docker Image bereitgestellt wird.

6.3. Developer Monitoring System

Der folgende Abschnitt widmet sich der Vorstellung der Implementierung des Developer Monitoring Systems. Er besteht aus den beiden Services `developer-monitoring-service` und `test-runner-worker` sowie aus den vier Datenbanken `version-database`, `context-database-ws`, `context-database-dms` und `runtime-registry`.

6.3.1. Kommunikationsübersicht

Der *Developer Monitoring System (DMS)* wird durch das Ziel der Prozessaufzeichnung stärker als der *Quality Rating System (QRS)* in die Plattform Prozesse eingebunden. Aus diesem Grund wird in diesem Abschnitt die Integrationsstelle in die Plattform und die anschließende Kommunikation innerhalb des DMS beschrieben.
Der `workspace-service` ist auf der Plattform das Backend für die `web-ide`. Das heißt sämtliche Kommunikation die bei der Bearbeitung einer Aufgabe durchgeführt wird geht über diesen Service. Das hat den Vorteil, dass die Daten die dabei entstehen nur einmal vom Client zum Server übertragen werden müssen und dort dann verteilt werden können. Für das DMS bedeutet das, dass es durch den `workspace-service` integriert werden muss. Die Umsetzung des `workspace-services` ist nicht Teil dieser Arbeit. Für das DMS ist daher nur aufgeführt, an welcher Stelle im `workspace-service` die Integration stattfindet.

Code Listing 6.1: Aufruf des DMS in der Methode `new_version`, des WS

```
1  def new_version(request, file_id):
2
3      ...
4
5      # pass data to developer monitoring service
6      dms_data = {
7          "file_id": file_id,
8          "timestamp": data.get('timestamp'),
9          "action": data.get('action'),
10         "content": data.get('content'),
11     }
12
13     try:
14         requests.post(DMS_URL, data=dms_data)
15     except requests.exceptions.ConnectionError:
16         print('DMS Offline')
17
18     ...
19
20     return JsonResponse(data=response_data)
```

In Listing 6.1 ist ein Teil des Methodenaufrufs dargestellt, welcher im `workspace-service` bei jeder Interaktion des Entwicklers mit der Web-IDE aufgerufen wird. Diese Methode eignet sich für die Integration, da sie alle Informationen mitbringt, die das DMS benötigt. Die Daten sind in der Variable `dms_data` gespeichert und werden mit einer POST Anfrage an den `developer-monitoring-service` (erreichbar unter der `DMS_URL`) weitergeleitet. Die weitere Verarbeitung der Daten findet im DMS selbst statt. Da ein Großteil der Komplexität des DMS in der Verteilung der Verarbeitung der Daten auf die zwei Services liegt, sind im Folgenden die

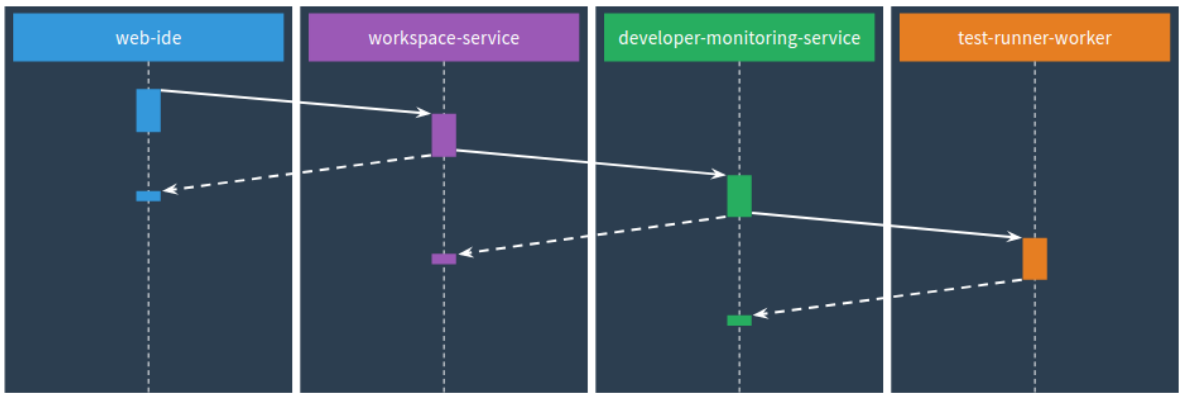

Abbildung 6.2.: Sequenzdiagramm mit den fünf Kommunikationspartnern des DMS - die nicht ausgefüllten Pfeile symbolisieren die asynchronen Aufrufe

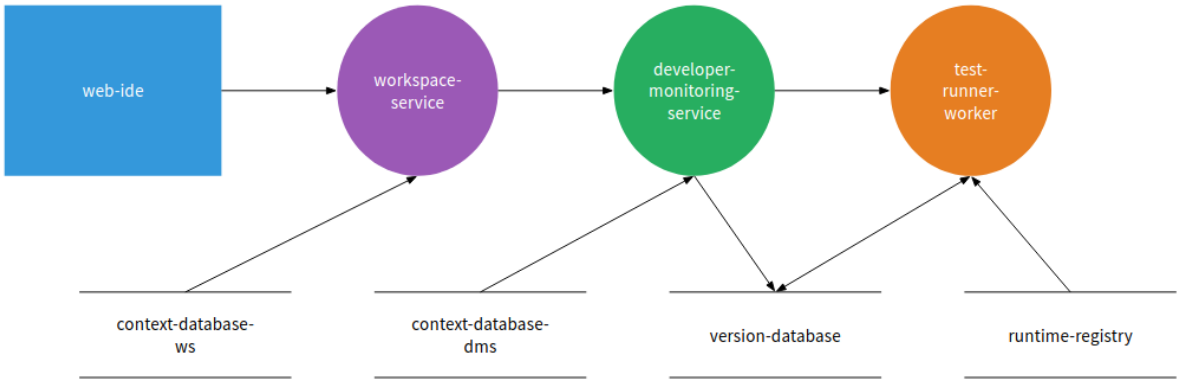

Abbildung 6.3.: Datenflussdiagramm mit den Verarbeitungsknoten der fünf Kommunikationspartner und vier Datenspeicher des DMS - der Pfeil zeigt die Richtung des Datenflusses an

Kommunikationswege aufgeführt, die auf den Aufruf im `workspace-service` folgen, in den Diagrammen aus Abbildung 6.2 und 6.3 visualisiert.
Beide Diagramme beschreiben das Szenario, in dem der Nutzer eine manipulierende Aktion in der Web IDE durchführt, die zu einem syntaktisch korrekten Stand führt. Beide Diagramme visualisieren den Ablauf der Kommunikation, beginnend bei der `web-ide`, über den `workspace-service` und anschließend den `developer-monitoring-service` bis hin zum `test-runner-worker`.
Das Sequenzdiagramm verdeutlicht, dass alle Aufrufe des jeweils nächsten Services ansynchron geschehen. Dadurch muss keiner der Services auf die Antwort des nächsten warten und wird dadurch nicht blockiert. Besonders beim Aufruf des `test-runner-workers` ist dies relevant, da die Ausführung der Tests häufig längere Zeit in Anspruch nimmt. Trotzdem wird nach dem Durchlauf ein Callback beim Aufrufenden ausgelöst, der auf die Antwort des Services reagiert. Zum Beispiel wird bei einem Fehler momentan geloggt, dass der Aufruf fehlgeschlagen ist. Es ist aber auch möglich diesen zu wiederholen, falls ein Service nur kurzzeitig nicht erreichbar war.
Im Datenflussdiagramm wird ersichtlich, woher die Services ihre Daten beziehen und wohin sie die verarbeiteten speichern. Mit der Interaktion des Nutzers und der Web IDE

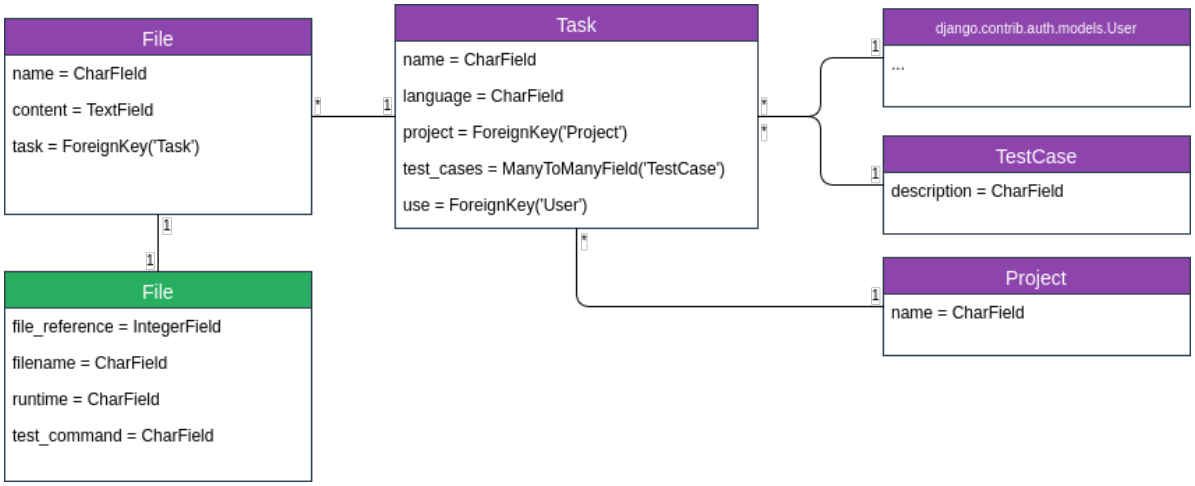

Abbildung 6.4.: Django Kontextdatenmodell des WS (lila) und DMS (grün)

gelangen über den *workspace-service* Daten in das System, weshalb die `web-ide` die Schnittstelle zur Umgebung ist. Im `workspace-service` werden Informationen über die `File` geladen, die sich geändert hat und zusammen mit aufgezeichneten Daten an den `developer-monitoring-service` weitergegeben. Dieser speichert die aufgezeichneten Daten in der `version-database` und lädt die Informationen die zur Ausführung der Tests notwendig sind, aus der `context-database-dms`. Diese leitet er dann an den `test-runner-worker` weiter, welcher sich wiederum den Dateiinhalt aus der `version-database` selbst lädt und das Ergebnis des Testlaufs wieder dort speichert. Zur Ausführung lädt er das Docker Image von der *runtime-registry* herunter. Deutlich wird, dass im betrachteten Szenario durch die Services des DMS nur lesend auf die Kontextdatenbanken zugegriffen wird, schreibender Zugriff erfolgt nur auf die Versionsdatenbank.

6.3.2. Developer Monitoring System

Der `developer-monitoring-service` ist mit Django umgesetzt und besteht aus der `core` App. Diese stellt die Kernfunktionalitäten des Services bereit. Das heißt, sie definiert die Modelle für die Kontextdaten und die Endpunkte für den Empfang der Versionsdaten.
Das konzipierte Modell aus dem Abschnitt 5.1.1 ist nicht alleine im `developer-monitoring-service` umgesetzt. Ein großer Teil der Entitäten bzw. Informationen ist im `workspace-service` gespeichert, da er schon diese Daten benötigt. Zum Beispiel ist er dafür zuständig, dass er die entsprechenden Dateien und deren Inhalte ausliefert, die der Nutzer in einer Aufgabe bearbeiten kann. In Abbildung 6.4 sind die Klassen dargestellt und wie sie sich auf die beiden Service aufteilen. Sie entsprechen zu einem großen Teil den Entitäten aus dem konzipierten Datenmodell. Die Attribute der `File`, `Task` und `Project` Entität wurde in zwei Klassen aufgespalten, da die Informationen nur jeweils auf einem der beiden Services verfügbar sein müssen. Zum Beispiel wird der Name des Projekts im `workspace-service` benötigt und die Laufzeitumgebung im `developer-monitoring-service`. Die zugehörigkeit einer `File` aus dem `developer-monitoring-service` zu der anderen ist über das Attribut `file-reference` abgebildet. Diese ist wie alle anderen kein `ForeignKey`-Attribut, da die Referenz über die Service bzw. Datenbanksystem Grenzen hinaus geht. Die Zugehörigkeiten der `Task` und `Project` Klassen zur jeweils anderen sind nicht zwingend notwendig, da es

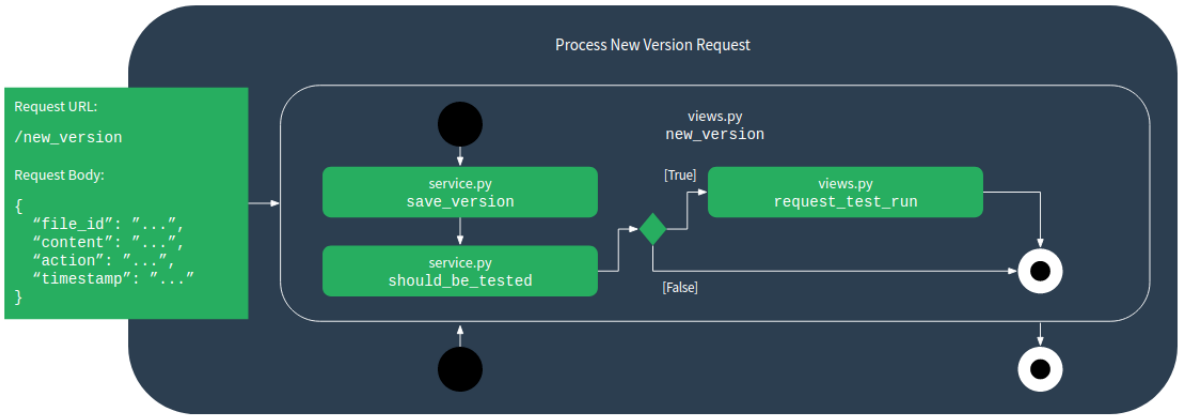

Abbildung 6.5.: Aktivitätsdiagramm des `developer-monitoring-services` bei der Anfrage nach der Speicherung einer neuern Version

sich mitunter um zwei verschiedene Instanzen handelt, die nicht miteinander in Verbindung stehen. Die `User` Klasse ist Django-spezifisch und ist vor allem für den `workspace-service` relevant.
Zur Beschreibung des Verhaltens des `developer-monitoring-services` ist in Abbildung 6.5 das Aktivitätsdiagramm zum Szenario *Speichern einer neuen Version* dargestellt. Als Eingabe dient der Request und dessen Daten an die URL `/new_version`. Die Daten sind im JSON Format und werden in der Methode `new_version` in der `views.py` entgegengenommen. Dort erfolgt der Aufruf der Methoden aus der `service.py`, die die Daten verarbeiten. In der `save_version` Methode wird der Inhalt und die zugehörige Aktion der Version gespeichert. In dem nächsten Schritt überprüft die Methode `should_be_tested`, ob der Versionsinhalt identisch mit der Vorgängerversion ist oder ob die aktuelle Version syntaktisch korrekt ist. Sollte diese Methode `True` zurückgeben wird vor dem Aktivitätsende in der Methode `request_test_run` ein Testlauf angefordert.
Die Anforderung der Tests läuft aktuell sehr einfach ab, da die Verwendung von Docker Compose diese Arbeit abnimmt. Mit Docker Compose ist es möglich mehrere Instanzen des TRW zu starten. Bei einem Aufruf des entsprechenden Hostnamens unter dem der Services erreichbar ist wird zufällig eines der Instanzen ausgewählt, an die der Request weitergeleitet wird. Für komplexere Orchestrierungsprozesse ist es möglich in der Methode `request_test_run` diese zu implementieren.

6.3.3. Version Database: CouchDB

Wie im Unterabschnitt 5.1.3 des Konzeptkapitels erläutert wurde, dient die `version-database` zur Speicherung der Entitäten `Version` und `TestCaseRun`, durch welche der größte Teil der Datenmenge entsteht. Die berechnete Datenmenge in 5.1.2 ist ein Grund für die Wahl eines nicht-relationalen Datenbanksystems. Diese sind für große Datenmengen und deren Analyse ausgelegt. Außerdem spielt der schreibende Zugriff von mehreren Services und damit möglicherweise auch von mehreren Servern eine Rolle. Bei vielen relationalen Datenbanksystemen blockieren Schreibzugriffe durch das *ACID-Prinzip* andere parallele Schreibzugriffe (). Durch den Verzicht auf dieses Prinzip bei nicht-relationalen Datenbanken kann garantiert werden, dass immer geschrieben werden, es aber zu Konsistenzproblem zwischen den Datensätzen in der Datenbank kommen kann.

Die beiden Entitäten mussten daher in ein Datenformat übertragen werden, wo diese Konsistenzprobleme nicht auftreten können. Dafür wurden sie in der CouchDB in einem Dokument zusammengefasst. In Listing 6.2 ist ein Beispiel es Datenmodells aufgeführt. Ersichtlich ist, dass neben den Versionsdaten `content` und `action` auch die Testlaufdaten als Liste im selben Dokument gespeichert werden.

Code Listing 6.2: Beispiel eines Dokuments als JSON in der CouchDB

```
{
  "_id":"1_1507401303839",
  "content":"def foo():\n return \"hello1\"",
  "action":"17",
  "test_result": [{
    "1":"True",
  },{
    "2":"False",
  },{
    "3":"True",
  }]
}
```

Eine weitere wichtige Entscheidung bei der Umsetzung von nicht-relationalen Datenbanken ist der Schlüssel des Dokuments, welcher dieses eindeutig referenziert. Eine `Version` ist eindeutig durch ihren `timestamp` und die `id` einer `File` referenzierbar. Aus diesem Grund setzt sich der Schlüssel eines Dokuments in der CouchDB folgendermaßen zusammen `<file_id>_<version_timestamp>`. Durch die gewählte Reihenfolge und das Trennzeichen ist es möglich die Versionen in ihrer zeitlichen Reihenfolge abzurufen, was für den Verlauf unbedingt notwendig ist. In Listing 6.2 beinhaltet das Attribut `_id` ein Beispiel für einen solchen Schlüssel.

6.3.4. Test Runner Worker

Der `test-runner-worker` übernimmt die Ausführung der Tests zu einem Versionsstand. Die beiden benötigten Funktionen *Initialisierung* und *Testausführung zu einer Version* werden über zwei Endpunkte einer Web-Schnittstelle bereitgestellt. Der Endpunkt für die Initialisierung muss einmalig vor der eigentlichen Ausführung der Tests aufgerufen werden und ist über die URL `/runtime/<runtime>/initialize` erreichbar. Er lädt ein fertiges Docker Images von der `runtime-registry` herunter, wodurch die Initialisierung bereits abgeschlossen ist. Die Verwendung der Technologie Docker reduziert die Komplexität dieses Prozessschrittes stark.
Die Umsetzung der Prozessschritte für die Anfrage nach einem Testlauf fällt komplexer aus, weshalb sie in Abbildung 6.6 in einem Aktivitätsdiagramm visualisiert ist. Der Request enthält alle Daten für die Anfrage als JSON. An erster Stells wird anhand des übermittelten Schlüssels (`key`) der Inhalt der Datei geladen, die getestet werden soll (`load_file_content`). Da die Tests im Docker Container ausgeführt werden, muss der Inhalt noch an die richtige Stelle im Container geschrieben werden. Dies geschieht über den Umweg einer temporären Datei die auf dem Host des TRW erstellt wird (`write_to_temp_file`). Diese Datei wiederum wird in den Container gemounted und dort dann ausgeführt (`run_test_container`). Für interprätierte Sprachen, wie Python, kann dies ohne einen Zwischenschritt passieren. Im Fall von kompilierten Sprachen, wie Java, muss zwischen dem Mount- und Ausführvorgang noch einmal im Container kompiliert werden. Nachdem die Tests im Container ausgeführt wurde wird das Resultat aus der Standardausgabe gelesen, da diese die gemeinsame Basis für viele

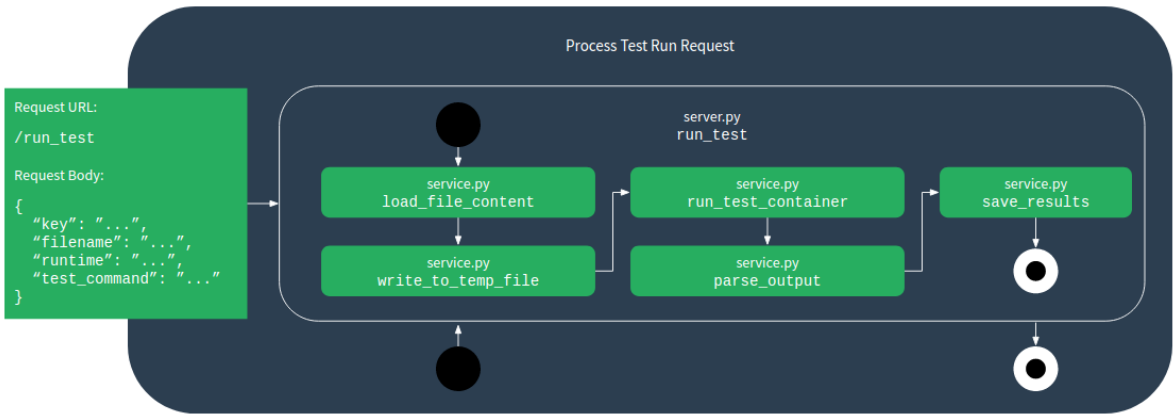

Abbildung 6.6.: Aktivitätsdiagramm des `test-runner-workers` bei der Anfrage nach einem Testlauf

der Testframeworks ist (`parse_output`). Die geparsten Resultate werden dann mithilfe des Schlüssels in das Versionsdokument in der CouchDB geschrieben (`save_results`).

6.4. Quality Rating System

Der folgende Abschnitt widmet sich der Vorstellung der Implementierung des QRS. Er besteht anders als der DMS nicht aus mehreren gekapselten Services, da der Fokus nicht auf der Lastenverteilung und Performanceoptimierung liegt. Die Kapselung wurde anhand der drei Django Apps `core`, `extraction` und `training` vorgenommen. Sie setzen die konzipierten Komponenten des QRS jeweils gekapselt in der App um. In den folgenden Abschnitten werden die einzelnen Apps beschrieben und wiederholt Bezug auf die Abbildung **??** genommen, welche die Datenmodelle der einzelnen Apps darstellt.

6.4.1. Core App mit TensorFlow

Die `core`-App setzt alle Funktionalitäten um, die in direktem Zusammenhang mit dem Aufruf des Machine-Learning Frameworks stehen. Dafür wurde das Framework *TensorFlow*[1] ausgewählt. TensorFlow wurde ursprünglich von Google entwickelt und für deren interne Produkte verwendet. Im Jahr 2015 wurde es unter der Apache 2.0 Open Source Lizenz veröffentlicht[2] und wird seitdem auch von der öffentlichen Community in einem GitHub Repository[3] weiterenwickelt.
In Abbildung 6.7 ist die vereinfachte Architektur von TensorFlow abgebildet. Dabei arbeitet der Benutzer mit der abstrakten API, welche u. a. von einem *Python client* bereitgestellt wird. Die API bietet auf der einen Seite fertig implementierte Classifier-Algorithmen (im Package `tf.estimators`), damit ein schnelles Prototyping möglich ist. Diese wiederum benutzen intern den grundlegenden Datentyp von TensorFlow, den `tf.Tensor`. Dadurch ist es möglich mit TensorFlow schnell eine Anwendung umzusetzen und später auch durch Vertiefung der eigentlichen Berechnungen zu optimieren. Die abstrakte API ist nur ein Wrapper der *C*

[1] `https://www.tensorflow.org/` (Letzter Aufruf: 06.09.2017)
[2] `https://www.heise.de/developer/meldung/Machine-Learning-TensorFlow-1-0-freigegeben-3627778.html` (Letzter Aufruf: 06.09.2017)
[3] `https://github.com/tensorflow` (Letzter Aufruf: 06.09.2017)

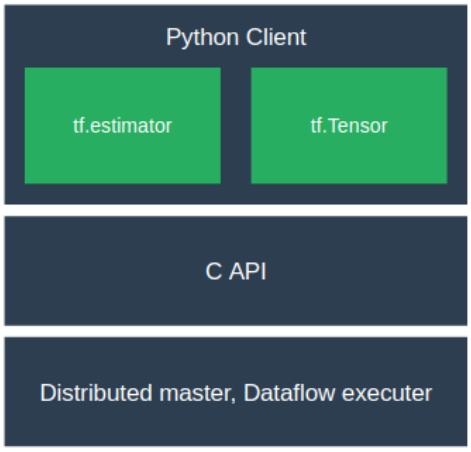

Abbildung 6.7.: Hierarchische Architektur des TensorFlow Frameworks nach (Abadi u. a. 2016)

API, welche die eigentlichen Funktionen von TensorFlow implementiert. Die C API wiederum benutzt die darunter liegenden Komponenten *Distributed master* und `Dataflow executor`, welche die verteilte Berechnung auf mehreren Servern und auch GPUs ermöglicht (Abadi u. a. 2016).

Das Datenmodell der `core`-App ist in Abbildung **??** in blau dargestellt. Es definiert die grundlegenden Entitäten beim maschinellen Lernen als Django Modellklassen. Ein Feature hat einen eindeutigen Namen (`name`) und eine deskriptive Beschreibung (`description`). Das `type`-Attribut definiert eine Auswahl an Datentypen, die zur Verfügung stehen. Abhängig vom gespeicherten Datentyp gibt die Property `dtype` den *NumPy*-Datentyp und `tf_feature_type` den TensorFlow-Datentyp zurück. Numpy wird intern von TensorFlows's Python client benutzt, um Vektoren etc. zu definieren. Die `ClassModel` und `Class` Klassen sind einfach gehalten und definieren eine einstufige Modell-Klassen Hierarchie, bei der ein Klassenmodell aus einer endlichen Anzahl an Klassen besteht. Die `Classifier` Klasse definiert alle Informationen, die benötigt werden um einen TensorFlow Classifier zu konfigurieren. Sein Typ bestimmt die Art des Classifier-Algorithmus'. Momentan werden von der `core`-App ein Linear Classifier und Deep Neural Network sowie eine Support Vector Machine unterstützt. Das `class_model` bestimmt die Qualitätscharakteristik, die der Classifier anhand der `features` lernen soll. Das Attribut `mode_directory` beschreibt den Pfad zum Ordner, wo die Brechnungsdateien und -ergebnisse gespeichert werden. Nachdem ein Classifier angelernt wurde kann er wieder geladen werden, ohne dass wiederholt der Lernschritt durchgeführt werden muss. Das `active`-Flag gibt an, ob ein Classifier für eine angeforderte Bewertung benutzt werden soll.

Im Aktivitätsdiagramm aus Abbildung 6.8 ist der Ablauf in der `core`-App beim *Anlernen des Classifiers* dargestellt. An erster Stelle wird der eintreffende Request in der `train_classifier`-Methode in der `views.py` entgegengenommen und entpackt. Dabei wird der einzige Parameter, die `classifier_id`, als URL Parameter übergeben. Dort wird der TensorFlow Classifier aus dem gespeicherten Django Modell des Classifiers generiert. Danach werden die Trainingsdaten für den Classifier in der Methode `load_data` geladen. Die Grundlage dafür ist dass sie im CSV Format vorliegen, welches von der `extraction`-App bereitgestellt wird. Anschließend kann der TensorFlow Classifier in der Methode `train_classifer` angelernt werden.

Das vorgestellte Aktivitätsdiagramm ist ebenfalls für die zwei weiteren umgesetzten Prozessschritte, der *Bewertung neuer Instanzen* und der *Evaluation des Classifiers*, symptomatisch.

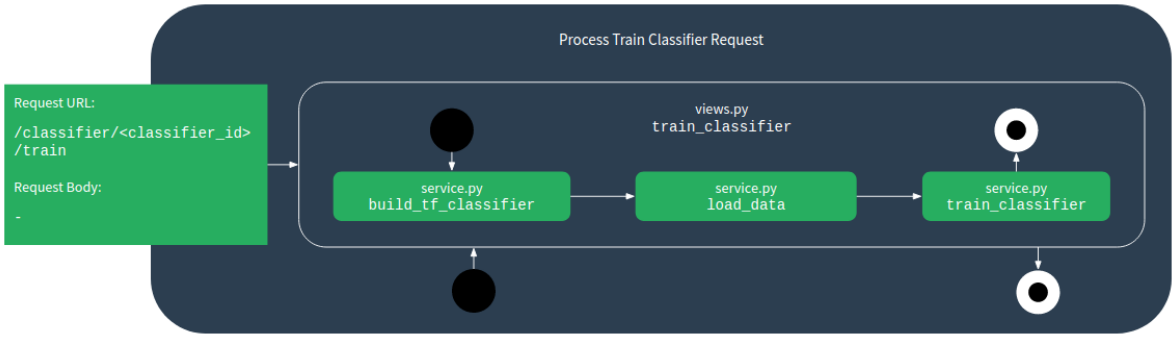

Abbildung 6.8.: Aktivitätsdiagramm der `core`-App bei der Anfrage nach dem Anlernen eines Classifiers

Dort ändert sich fast ausschließlich nur, dass eine andere TensorFlow-Funktion (`evaluate` bzw. `predict`) anstelle der Funktion `train` aufgerufen werden muss und im Bewertungsschritt die Labels nicht bekannt sind.

6.4.2. Feature Extraction App

TODO

6.4.3. Training App

Die `training`-App bietet dem Experten eine Oberfläche zur Bewertung an. Diese kann über die URL `/training/<instance_id>` abgerufen werden. Die Oberfläche beinhaltet alle konzipierten Elemente, um die Bewertung einer Instanz (definiert durch `<instance_id>`) vorzunehmen. In Abbildung 6.9 befindet sich eine Beispielinstanz in der benannten Oberfläche. Alle Informationen werden über die `context`-Variable in Django geladen und anschließend dargestellt. Sobald der Nutzer alle Daten eingegeben hat, kann er über den blauen Button unten die klassifizierte Instanz abschicken. Danach wird er direkt zur nächsten Instanz weitergeleitet bis er alle Instanzen einmal klassifiziert hat. Die Klassifizierung wird benutzerspezifisch gespeichert, sodass überprüft werden kann, welche Instanzen dieser Nutzer bereits bewertet hat.
Das Datenmodell der Trainingsdaten, die der Oberfläche zugrunde liegen, ist in Abbildung **??** in der Lila dargestellt. Die `Instance` ist das zentrale Element der `training`-App. Es speichert den bewerteten Code im entsprechend benannten Attribut und ist einem `Training` zugeordnet, welches die `ClassModels` bestimmt, mit denen die Instanz bewertet werden soll. Anstelle der Referenz auf ein Codeausschnitt als Zeichenkette bestand die Möglichkeit eine `Task` oder `File` zu referenzieren, die dann bewertet wird. Die gewählte Variante ist allerdings generischer und lässt sich auch mit Codeausschnitten durchführen, die nicht über die Plattform implementiert wurden. Dann fehlen zwar die Daten des DMS, aber eine Bewertung der klassischen Metriken ist trotzdem möglich. Das `TrainedClassModel` speichert das trainierte Klassenmodell zu einer Instanz ab. Das heißt es referenziert ein `ClassModel` und eine `Instance` und speichert den trainierten Index des Klassenmodell im entsprechend benannten Attribut ab. Das Kommentar (`comment`), sowie der Nutzer der bewertet hat (`user`) sind ebenfalls gespeichert.

6.5. Zusammenfassung

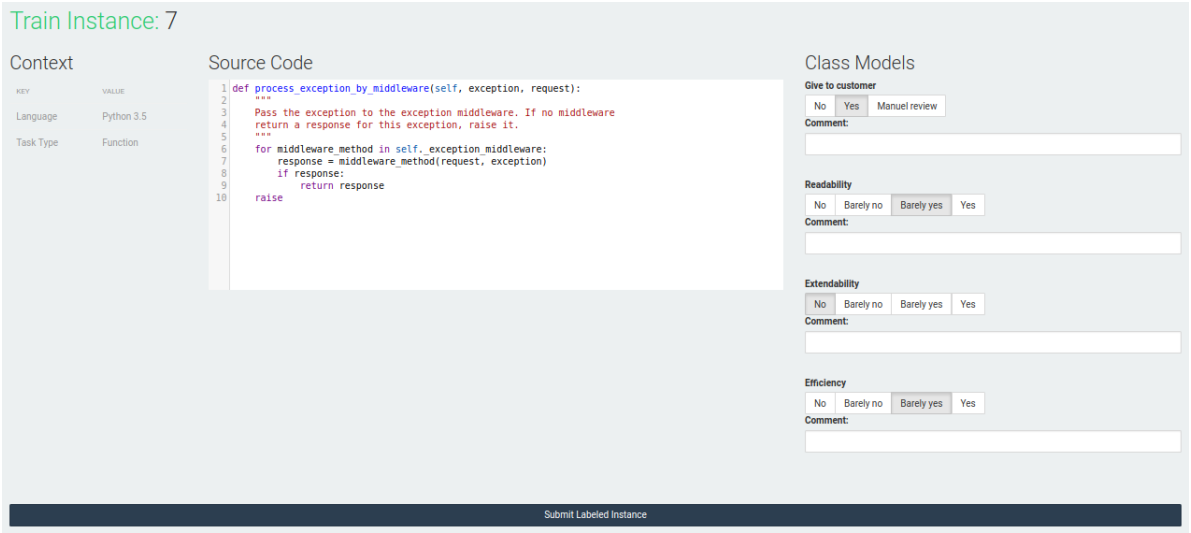

Abbildung 6.9.: Trainings Oberfläche der `training`-App mit vier Klassenmodellen

7. Evaluation

Im diesem Kapitel wird die durchgeführte Evaluation der zwei umgesetzten Systeme beschrieben. In einem ersten Schritt wird dafür das Vorgehen und die Art der Evaluation vorgestellt. Anschließend findet eine Erläuterung der Anforderungen und ihrer Qualitätsdiemensionen statt, anhand derer die Güte der Umsetzung evaluiert werden kann.

7.1. Vorgehen

Für die Evaluation der umgesetzten Systeme in dieser Arbeit ist maßgebend, wie die selbst gestellten Anforderungen qualitativ umgesetzt wurden. Zur Bestimmung wurden die Anforderungen *technisch evaluiert*, indem Kriterien für alle Anforderungen aufgestellt wurden, die die Güte der Umsetzung erfassen und anhand derer eine Bewertung möglich ist (vgl. Tabelle 7.1). Die technische Evaluation anhand der Kriterien wurde ohne andere Endnutzer durchgeführt. Die Ursache dafür liegt einerseits darin, dass es sich beim DMS um eine Backendtechnologie handelt. Das entsprechende Frontend bzw. der Web Editor sind noch in einem frühen Entwicklungstadium und unterstützen noch nicht alle Funktionalitäten, die notwendig wären, um das DMS anzubinden. Aus diesem Grund wurde die Evaluation mithilfe einer geskripteten Simulation des Editors durchgeführt (vgl. Abschnitt 7.2). Das QRS und dessen Qualitätsbewertung basiert andererseits zu einem großen Teil auf den aufgezeichneten Daten des Developer Monitorings. Für den QRS wäre eine Evaluation durch einen Nutzer aufgrund der Interaktion zwischen ihm und dem System sinnvoll, ist allerdings zum jetzigen Zeitpunkt der Entwicklung nicht möglich, da die notwendige Grundlage von echten Daten fehlt, um das maschinelle Lernen in vollem Umfang durchzuführen.
Für die Should-Have Anforderungen wurde jeweils ein Qualitätskriterium und für die Must-Have mind. 2 Kriterien evaluiert. Die Umsetzungsqualität der höher priorisierten Anforderungen ist so detallierter untersucht wurden. Die Could-Have Anforderungen wurden nur in Bezug auf deren Konzeption und Umsetzung untersucht und nicht näher durch Qualitätskriterien evaluiert. Die Won't-Have Kriterien sind im Kapitel 8 im Ausblick aufgeführt, da sie für weiterführende Arbeiten interessant sind, die sich mit der Erweiterung der beiden Systeme beschäftigen.

Tabelle 7.1.: Qualitätsdimension der Anforderungen

ID	Anforderungsname	Qualitätsdimension
M-DM-10	Detallierter Versionsverlauf	Versionsstände / Sekunde
		Durschn. Zeichenunterschiede zwischen Versionsständen
M-DM-20	Getestete Versionsstände	Testlaufzeit (max, min, avg.)
		Max. Testläufe eines TRW hintereinander
M-DM-30	Strukturierte Speicherung der Rohdaten	Anzahl betroffener Tabellen für Versionsstand
		Dauer für Sortierung nach Zeit (innerhalb Aufgabe)
		Besteht Möglichkeit für MapReduce-Analysen
M-DM-40	Web-Schnittstelle	Zeit für Speicherung eines neuen Versionsstands
		Zeit für Speicherung bei parallelen Anfragen eines neuen Versionsstands
S-DM-10	Mehrere Dateien	Anzahl der Dateien
S-DM-20	Erkennung identischer Versionstände	Formatierungs-sensitiv
S-DM-30	Erkennung syntaktisch korrekter Versionsstände	Anzahl der unterstützten Sprachen
M-QR-10	Extraktion der Qualitätsmetriken	Anzahl der Metriken
		Anzahl der Developer-Monitoring-Metriken
		Anzahl der klassischen Softwaremetriken
		Dauer der Extraktion
M-QR-20	Konfiguration der Features	Anzahl der auswählbaren Features die in die Auswertung einfließen
		Anzahl der Feature Types
M-QR-30	Konfiguration der Klassifikationsmodelle	Anzahl der Klassifikationsmodelle
		Anzahl der Klassen / Klassifikationsmodell
M-QR-40	Nutzer-Oberfläche zum Anlernen	Responsive Oberfläche
		Syntax-Highligthing des Quellcodes
M-QR-50	Speicherung der Trainings- und Testdaten	Anzahl der speicherbaren Instanzen
		Möglichkeit zur Gruppierung von Trainings
M-QR-60	Web-Schnittstelle	Dauer für Anfrage einer Bewertung
		Dauer für parallele Anfragen von Bewertungen
		Dauer für Anfrage eines Lernschrittes
S-QR-10	Extraktion der Metiken mehrerer Sprachen	Wieviele Sprachen
S-QR-20	Datenbereinigung	Anzahl bereinigter Metriken

7.2. Setup und Szenarien

7.3. Technische Evaluation

7.4. Zusammenfassung

8. Fazit & Ausblick

In diesem Kapitel wird ein Fazit gezogen, um die Beiträge der Arbeit abschließend zusammengefasst herauszustellen. Weiterhin sind aufbauende Themen und offene Punkte für weiterführende Arbeiten beschrieben.

8.1. Fazit

Die Qualitätssicherung auf Crowdsourcing-Plattformen ist ein Ressourcenproblem. Sie ist theoretisch möglich, muss aber so weit es geht automatisiert werden, damit Crowdsourcing auch wirklich Parallelität bietet und nicht ein Flaschenhals bei den Plattformbetreibern oder Auftraggebern entsteht. In dieser Arbeit wurde ein datengetriebener Ansatz verfolgt. Durch Aufzeichnung des Bearbeitungsprozesses der Aufgabe sollten neue Daten erfasst werden und in die Bewertung mit einfließen. Ein System für die schnellen Evaluation der Daten sollte ebenfalls entworfen werden.
Dafür wurden in einem ersten Schritt verwandte Arbeiten mit dem Fokus auf Qualitätsbewertung von Software im Allgemeinen beschrieben. Dabei wurden Softwaremetriken als zentrales Element herausgestellt und deren Probleme beschrieben, die vor allem in ihrer Auswertung liegen. Manuell erstellte Entscheidungsbäume bieten zwar eine Möglichkeit der Auswertung, sind aber nur bei der Verwendung von vielen Metriken sinnvoll, dann aber wiederum sehr aufwendig zu erstellen. Ein Ansatz bieten maschinelle Lernalgorithmen, die viele Metriken mit einer objektiven Qualitätscharakteristik (wie der Fehleranfälligkeit, z. B. definiert durch die Anzahl der auftretenden Bugs) korrelieren. Dieser Ansatz ist allerdings nur über einen längeren Zeitraum einsetzbar und bedingt, dass der zu überprüfende Quellcode bereits in den Betrieb geht.
Anschließend wurde der Stand der Technik bei der Aufzeichnung des Entwicklungsprozesses und der Qualitätsbewertung von Quellcode untersucht, indem Werkzeuge bzw. Systeme vorgestellt wurden, die aktuell dafür eingesetzt werden. Die beiden Prozessschritte wurden getrennt voneinander betrachtet. Keines der untersuchten Werkzeuge erfüllte die selbstgestellen groben Anforderungen an die Lösung der beiden Prozesse. Aus diesem Grund wurde in Abschnitt 3.4 eine Anforderungsanalyse an zwei Systeme zur Lösung der Problemstellung durchgeführt. Die Anforderungen wurden mit der MoSCoW-Priorisierung klassifiziert.
Im Kapitel 4 wurde der Kontext der testgetriebenen Crowdsourcing-Plattform *polylith* vorgestellt, um die Arbeit in diesen einzuordnen. Im Kapitel 5 wurde separat auf die Konzepte der beiden Systeme eingegangen. Beim DMS lag der Fokus auf dem Datenmodell und den entste-

henden Datenmengen, sowie auf der Durchführung der verteilten Ausführung der Testläufe. Beim QRS lag der Fokus auf dem allgemeinen Prozess des maschinellen Lernens im gegebenen Kontext, damit alle Datenverarbeitungsschritte von der Trainingsdatenerstellung bis zur Evaluierung der einzelnen berechneten Metriken abgedeckt ist.
Im Kapitel 6 wurde die Umsetzung der Konzepte für die beiden Systeme beschrieben. Dabei wurde das Vorgehen bei der Implementierung und die verwendeten Technologien vorgestellt. Vorgehen bei der Erstellung des DMS wurde durch das Microservice Paradigma geprägt. Docker, TensorFlow und Django sind die Technologien, die am meisten Einfluss auf die Umsetzung hatten beider Systeme hatten. Aufbauend auf der Beschreibung des Vorgehens folgte die Beschreibung der beiden Systeme entlang ihrer Services bzw. Django Apps.
XXX (Evaluation - Erfüllte/Offene Anforderungen)

Zusammenfassend wurde mit dieser Arbeit einerseits ein System zur Aufzeichnung des Entwicklungsprozesses entworfen und umgesetzt. Damit ist es möglich detaillierte Daten über die Entstehung des Quellcodes und die Arbeitsweise des Entwicklers zu sammeln. Die Interaktionen des Entwicklers mit dem Web Editor und Testergebnisse zu jedem Versionsstand können aufgezeichnet werden. Das DMS wurde als emergentes System umgesetzt. Jeder Service des DMS übernimmt einen Teilprozess und nur zusammen erfüllen sie die gestellte Anforderungen. Das System bleibt somit skalierbar, da die Services entsprechend ihrer individuellen Auslastung skaliert werden können. Die neuen Informationen können aufbauend für die Analysen verwendet werden, die in die verschiedensten Richtungen gehen können (vgl. Abschnitt **??**).
Eine Möglichkeit die Daten zu verwenden ist die Bewertung der Codequalität mit dem andererseits umgesetzten QRS. Es ermöglicht die Anwendung von maschinellen Lernalgorithmen mithilfe des Frameworks TensorFlow, auf die gespeicherten Daten. Trainingsdaten können durch einen Qualitätsexperten einfach ins System eingepflegt werden. Die Berechnung und Evaluation verschiedenster Metriken aus den gesammelten Daten kann im gegebenen Kontext schnell durchgeführt werden.

8.2. Ausblick

Die folgenden Abschnitt beschreiben eine Auswahl an weiteren Arbeiten die aufbauend auf dieser Arbeit durchgeführt werden könnten.

8.2.1. Testausführung mit Orchestrierungswerkzeug

8.2.2. Sicherheit

8.2.3. Gute Metriken und Qualitätsprofile

8.2.4. Entwickler Bewertung

Aufzeichnung birgt Motivationspotential Was nicht aufgezeichnet kann nicht gemessen werden

8.2.5. Testfall-Code Mapping

8.2.6. Detektion von Betrugsversuchen

8.2.7. Komplexere Aktionen unterstützen

StackoverFlow

Literatur

Abadi, M., P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, X. Zheng und G. Brain (2016). „TensorFlow: A System for Large-Scale Machine Learning TensorFlow: A system for large-scale machine learning“. In: *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI '16)*, S. 265–284. URL: https://www.usenix.org/conference/osdi16/technical-sessions/presentation/abadi.

Basili, V. R., G. Caldiera und H. D. Rombach (1994). „The goal question metric approach“. In: *Encyclopedia of Software Engineering* 2, S. 528–532. URL: http://maisqual.squoring.com/wiki/index.php/The%20Goal%20Question%20Metric%20Approach.

Boehm, B. W., J. R. Brown und M. Lipow (1976). „Quantitative evaluation of software quality“. In: *Proceedings of the 2nd international conference on Software engineering*, S. 592–605. URL: http://dl.acm.org/citation.cfm?id=800253.807736.

Card, S. K., T. P. Moran und A. Newell (1980). „The keystroke-level model for user performance time with interactive systems“. In: *Communications of the ACM* 23.7, S. 396–410.

Dienst, P. (2017). „Konzept zur Unterstützung von Entwicklern bei der Aufgabenbearbeitung in einer testgetriebenen Crowdsourcingplattform für Softwareentwicklung“. unveröffentlicht. TU Dresden.

Domingos, P. (2012). „A few useful things to know about machine learning“. In: *Communications of the ACM* 55.10, S. 78. URL: http://dl.acm.org/citation.cfm?doid=2347736.2347755.

Fenton, N. und J. Bieman (2014). *Software metrics: a rigorous and practical approach*. CRC Press.

Fowler, M. (2014). *Microservices*. URL: https://martinfowler.com/articles/microservices.html.

Fowler, M. und K. Beck (1999). *Refactoring: improving the design of existing code*. Addison-Wesley Professional.

Hanspach, F. (2017). „Entwicklung eines Gamification-Konzepts für crowd-basierte Softwareentwicklung“. unveröffentlicht. TU Dresden.

Al-Jamimi, H. A. und M. Ahmed (2013). „Machine Learning-Based Software Quality Prediction Models: State of the Art“. In: *2013 International Conference on Information Science and Applications (ICISA)*, S. 1–4. URL: http://ieeexplore.ieee.org/document/6579473/.

Kamei, Y. und E. Shihab (2016). „Defect Prediction: Accomplishments and Future Challenges“. In: *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)* 1, S. 33–45. URL: http://ieeexplore.ieee.org/document/7476771/.

Latoza, T. D., W. Ben Towne, A. Van Der Hoek und J. D. Herbsleb (2013). „Crowd development“. In: *2013 6th International Workshop on Cooperative and Human Aspects of Software Engineering, CHASE 2013 - Proceedings*, S. 85–88.

Li, H. F. und W. K. Cheung (1987). „An empirical study of software metrics“. In: *IEEE Transactions on Software Engineering* 6, S. 697–708.

McCabe, T. J. (1976). „A complexity measure“. In: *IEEE Transactions on software Engineering* 4, S. 308–320.

Rathore, S. S. und S. Kumar (2017). „A study on software fault prediction techniques“. In: *Artificial Intelligence Review*, S. 1–73.

Razavi, H. A. und T. R. Kurfess (2007). „Detection of Wheel and Workpiece Contact/Release in Reciprocating Surface Grinding“. In: *Journal of Manufacturing Science and Engineering* 125.2, S. 394. URL: http://manufacturingscience.asmedigitalcollection.asme.org/article.aspx?articleid=1447160.

Shepperd, M., D. Bowes und T. Hall (2014). „Researcher Bias : The Use of Machine Learning in Software Defect Prediction“. In: 40.6, S. 603–616.

Abbildungsverzeichnis

Tabellenverzeichnis

A. IDE-Aktionen beim Developer Monitoring (Liste)

Aktionen die den Quellcode manipulieren

- Tastatureingabe von einzelnen Zeichen
- Löschen des selektierten Abschnitts
- Einfügen (Ctrl - V)
- Umbenennen von Variablen, Konstanten und Methoden
- Extrahieren eines Codeblocks in eine Methode
- Auto-Completion Auswahl
- Quellcode Formatierung

Aktionen die den Quellcode nicht manipulieren

- Hover Funktion
- Sprunglink zu Klassendefinition
- Auto-Completion Anzeige
- Markieren

B. Weitere Rechnungen

Berechnung der Datenmengen der anderen Rohdaten Attribute

Die Berechnung der Datenmengen der anderen Attribute ist in den folgenden Gleichungen dargelegt. Andere Attribute beziehen sich in diesem Abschnitt auf die Attribute der Rohdaten-Entitäten `Version` und `TestCaseRun`, exklusive des `code`-Attrbutes. Die Datenmengen für die jeweiligen Datentypen sind in Tabelle B.1 aufgelistet. Die Berechnung ist nur eine grobe Näherung, da die exakte Größe stark vom verwendeten Datenbanksystem abhängt. Um die Daten nach oben abzuschätzen wurde hier eine speicherintensives Datenbankschema gewählt (Fremdschlüssel von `TestCaseRun` zu `Version`) gewählt.

Datentyp	**Datenmenge**
int	4 Byte
long	4 Byte
boolean	1 Byte
Fremdschlüssel Identifier	4 Byte

Tabelle B.1.: Datentypen und deren Größen

$$\begin{aligned} data\_attributes\_of\_version = {} & timestamp\_attribute\_data \\ & + action\_type\_id\_data \\ & + file\_id\_data \end{aligned} \tag{B.1}$$

Für die Datenmenge einer Versionsinstanz wird das Attribut `timestamp` mit den Datenmengen der 2 Fremdschlüssel Identifier (Relation zu den Entitäten `ActionType` und `File`) addiert (vgl. Gleichung B.1). Nach Einsetzen der Werte ergibt sich folgendes Ergebnis:

$$\begin{aligned} data\_attributes\_of\_version &= 4B + 4B + 4B \\ &= 12B \end{aligned} \tag{B.2}$$

Damit beträgt die Datenmenge für alle anderen Attribute einer Versionsinstanz **12 B**.

$$\begin{aligned} data\_attributes\_of\_testcaserun = {} & result\_data \\ & + test\_case\_id\_data \\ & + version\_timestamp\_data \end{aligned} \tag{B.3}$$

Für die Datenmenge einer Instanz eines Testlaufs wird das Attribut `result` zur Datenmenge des Fremdschlüssel Identifier (Relation zur Entität `Version`) addiert (vgl. Gleichung B.3). Nach Einsetzen der Werte ergibt sich folgendes Ergebnis:

$$\begin{aligned} data\_attributes\_of\_testcaserun &= 1B + 4B + 4B \\ &= 9B \end{aligned} \tag{B.4}$$

Damit beträgt die Datenmenge für eine Instanz eines Testlaufs **9 B**.

$$\begin{aligned} data\_other\_attributes = number\_of\_version\_instances \\ &\times (number\_of\_tests\_per\_task \\ &\times data\_attributes\_of\_testcaserun \\ &+ data\_attributes\_of\_version) \end{aligned} \tag{B.5}$$

Mit Gleichung B.5 lässt sich die Gesamtdatenmenge der anderen Attribute berechnen. Nach einsetzen der Werte aus den Anforderungen *NF-DM-20* und *NF-DM-50* ergibt sich folgendes Ergebnis:

$$\begin{aligned} data\_other\_attributes &= 10.000 \times (20 \times 9B + 12B) \\ &= 1.920.000B \\ &\approx 1,9MB \end{aligned} \tag{B.6}$$

Damit beträgt die Datenmenge aller anderen Attribute der Rohdaten **1,9 MB**. Im Verhältnis zu der im Unterabschnitt 5.1.2 berechneten Datenmenge des `code`-Attributes sind das 7,6 %. Da diese Datenmenge die Wahl des Datenbanksystems nicht beeinflusst beschränkt sich die Betrachtung im Konzept Kapitel auf die dort berechneten 25 MB.