

Institute of Computer Science, Software Engineering Group

Bachelorarbeit

Entwicklung einer Anwendung zur Bewertung der Dokumentation von Quellcode mithilfe von GitHub Actions

Timo Schoemaker

Immatrikulationsnummer: 978621

7. April 2022

Erstbetreuer: Prof. Dr.-Ing. Elke Pulvermüller

Zweitbetreuer: Nils Baumgartner, M. Sc.

Abstract

Deutsch Die Softwaredokumentation ist ein essenzieller Bestandteil der heutigen Softwareentwicklung geworden. Nichtsdestotrotz leidet die Qualität der Dokumentation häufig und viele Entwickler sind nicht motiviert genug, um eine gute Dokumentation zu schreiben. Das Ziel dieser Arbeit ist es, ein Tool zu entwickeln, dass exemplarisch die Dokumentationsqualität in Java-Programmen analysiert und mittels verschiedener Metriken (Anteil dokumentierter Komponenten an allen Komponenten, Flesch-Score, Kohärenz und Nichterwähnung von Randfällen) bewertet. Dieses Tool ist in GitHub Actions eingebunden, um den Entwickler bei einer sehr schlechten Dokumentationsqualität zu warnen und gegebenenfalls Mergevorgänge zu verhindern.

English The software documentation has become an integral part of software development. Nevertheless, the quality of the documentation is often poor and developers are often not motivated to write good documentation. The goal of this thesis is to develop a tool that can analyze the documentation quality of Java applications by applying different metrics (percentage of documented components in all components, Flesch score, coherence, not mentioning the handling of edge cases). This tool will be integrated in GitHub Actions to warn the developer about poor software documentation quality and to prevent a merge if the quality becomes too poor.

Inhaltsverzeichnis

Abkürzungsverzeichnis	v
1 Einführung	1
1.1 Zielsetzung	2
1.2 Gliederung	3
2 Grundlagen	3
2.1 Softwaredokumentation	4
2.2 Javadoc	5
2.3 Code-Smells	7
2.4 JavaScript, Node und TypeScript	8
2.5 ANTLR4	9
2.6 GitHub Actions	11
2.7 Verwandte Werkzeuge	15
2.8 Verwandte Arbeiten	17
3 Konzeption	19
3.1 Parsing von Dateien	19
3.2 Konzeption der Metriken	22
3.2.1 Implementation einer Metrik	22
3.2.2 Einzelergebnisse verarbeiten	25
4 Umsetzung	29
4.1 Ausführung des Programms	29
4.2 Traversierung aller relevanten Dateien und der Komponenten	31
4.3 Implementierung von ANTLR4 für Java	31
4.4 Parsen der strukturierten Kommentare	33
4.5 Konfiguration des Tools	33
4.6 Speicherung des letzten Ergebnisses	35
4.7 Einbindung in GitHub Actions	35
4.8 Implementierte Metriken	37
4.8.1 Metriken, welche die Abdeckung überprüfen	37
4.8.2 Metriken, welche die Semantik überprüfen	38
4.8.3 Metriken, welche nach Fehlern suchen	40
4.9 Algorithmen zur Bildung eines Gesamtergebnisses	41
4.9.1 Klassische Algorithmen	41
4.9.2 Algorithmen aus der Ökonomie	42

4.9.3	Squale	43
5	Evaluation	44
5.1	Wahl der zu analysierenden Projekte	45
5.2	Analyse der Qualität	46
5.2.1	Ergebnisse	48
5.2.2	Bewertung der Qualitätsevaluation	51
5.3	Analyse der Geschwindigkeit	52
5.3.1	Ergebnisse	53
5.3.2	Bewertung der Ergebnisse	55
5.4	Fazit der Evaluation	55
6	Fazit	56
	Literaturverzeichnis	58
	Anhang A Änderungen an der Parserdatei	64
	Anhang B UML-Diagramm: Parser	65
	Anhang C UML-Diagramm: Metriken	66
	Anhang D Konfiguration des Tools	67
	Anhang E Implementierte Metriken	68
	Anhang F Bilder des Tools	71

Abkürzungsverzeichnis

CI/CD	<i>Continuous Integration/Continuous Delivery</i>
HTML	<i>Hyper Text Markup Language</i>
IDE	<i>Integrated Development Environment</i>
JSON	<i>JavaScript Object Notation</i>
JDT	<i>Java Development Tools</i>
LOC	<i>Lines of Code</i>
NLP	<i>Natural Language Processing</i>

1 Einführung

Ein wichtiger Bestandteil der Softwareentwicklung von heute ist die Softwaredokumentation. Dies liegt unter anderem daran, dass die Größe von Softwareprojekten steigt, sodass die Entwickler schnell den Überblick über das Projekt verlieren können und daher zusätzliche Informationen neben dem Code benötigen [1, S. 1]. Nichtsdestotrotz wird die Softwaredokumentation von Entwicklern oft vernachlässigt [2, S. 83]. Die Gründe für schlechte Dokumentation sind vielfältig. Das Schreiben der Dokumentation wird oft als mühevoll empfunden und erfordert Fähigkeiten, die ein Programmierer nicht zwangsläufig besitzt [3, S. 70] [4, S. 593].

Weitere Studien verdeutlichen die Problematik der mangelhaften Softwaredokumentation. So belegt eine Umfrage aus dem Jahr 2002 mit 48 Teilnehmern beispielsweise, dass die Dokumentation bei Änderungen am System nur mit Verzögerung angepasst wird. Knapp 70 % der Teilnehmer stimmen der Aussage zu, dass die Dokumentation immer veraltet ist. [5, S. 28–29]

Eine weitere Studie [6, S. 1199–1208] aus dem Jahr 2019 verdeutlicht viele Aspekte aus der vorgenannten Umfrage. Es wurden dabei Daten aus Stack Overflow, GitHub-Issues, Pull-Requests und Mailing-Listen automatisiert heruntergeladen und dann von den Autoren analysiert, ob und inwieweit diese durch mangelhafte Softwaredokumentation verursacht wurden. Die Studie belegt, dass von 824 Problemen, die etwas mit dem Thema „Softwaredokumentation“ zu tun haben, 485 sich auf den Inhalt der Dokumentation beziehen (wie z. B. unvollständige, veraltete oder sogar inkorrekte Dokumentation). Bei 255 Einträgen gab es Probleme mit der Struktur der Dokumentation, sodass beispielsweise Informationen schlecht auffindbar sind oder nicht gut verständlich sind.

Eine andere Umfrage aus dem Jahr 2014 mit 88 Teilnehmern zeigt, dass eine automatisierte Überprüfung der Dokumentationsqualität von knapp der Hälfte der befragten Entwickler gewünscht wird. Die Autoren der Studie sehen dies als Zeichen dafür, dass ein grundsätzliches Bedürfnis zur automatisierten Bewertung von Dokumentationen besteht und daher weitere Studien notwendig sind. [7, S. 340]

Die mangelhafte Dokumentation führt dazu, dass nicht nur nachfolgende Entwickler Probleme mit dem Codeverständnis haben, sondern auch Entwickler eines Moduls nach einer längeren Pause Zeit aufbringen müssen, um den Code wieder zu verstehen [8, S. 511]. Auch für Kunden/Auftraggeber ist eine gute Dokumentation wichtig, da gut dokumentierte Software tendenziell besser wartbar ist und somit mehr Nutzen bringt [2, S. 83] [9, S. 1].

1.1 Zielsetzung

Aufgrund der Relevanz von gut dokumentierter Software ist eine regelmäßige Rückmeldung über die Dokumentation von hoher Bedeutung. Spezielle Metriken, die eine numerische Auskunft über die Qualität der Softwaredokumentation liefern, sind eine Möglichkeit, diese Rückmeldung zu geben. Diese Metriken verschaffen dem Programmierer eine Einschätzung darüber, ob die Softwaredokumentation ausreichend ist oder eine Verbesserung sinnvoll wäre. Die Qualität der Softwaredokumentation kann auf unterschiedliche Art und Weise bewertet werden. So kann beispielsweise die bloße Existenz einer Dokumentation geprüft werden oder aber auch die Verständlichkeit der Dokumentation bewertet werden, daher kann es sinnvoll sein, mehrere Metriken zu verwenden [10, S. 29]. Damit ein Entwickler einen Gesamtüberblick über die Dokumentationsqualität erhält, können diese Metriken kombiniert werden, um eine einzelne numerische Bewertung der Qualität der Dokumentation zu erhalten. Dabei ist es auch ratsam, die Metriken zu gewichten oder eine andere Methode zur Kombination der Metrikergebnisse zu benutzen, weil nicht jede Metrik die gleiche Zuverlässigkeit und Relevanz besitzt [11, S. 1117ff.].

Damit das Feedback über die Softwaredokumentation auch wahrgenommen wird, sollte die Qualität regelmäßig überprüft werden. Dies kann automatisiert im *Continuous Integration/Continuous Delivery* (CI/CD)-Prozess erfolgen, bei dem Software kontinuierlich getestet und für den Release (z. dt. Veröffentlichung) vorbereitet werden kann. Durch CI/CD können Unternehmen effizienter und besser Software entwickeln. So konnte das Unternehmen *ING NL* die gelieferten Function-Points vervierfachen und die Kosten für einen Function-Point auf einen Drittel reduzieren [12, S. 520].

Basierend auf diesen Überlegungen soll ein Tool (z. dt. Werkzeug) entwickelt werden. Dieses Tool (im Folgenden auch *DocEvaluator* soll ein gegebenes Software-Projekt analysieren und eine numerische Bewertung abgeben, die eine heuristische Aussage über die Qualität der Softwaredokumentation trifft. Dabei soll das Tool primär für Javadoc und Java bis Version 8 konzipiert werden, allerdings soll während der Entwicklung auch darauf geachtet werden, dass eine Portierung auf eine andere Programmiersprache ermöglicht wird und die Bewertung der Dokumentation unabhängig von der Programmiersprache funktioniert. Außerdem wird zur Vereinfachung nur englischsprachige Dokumentationen betrachtet. Komplexe *Natural Language Processing* (NLP)-Metriken sollen dabei außer Acht gelassen werden. Auch Verfahren, die den Quellcode mit der Dokumentation vergleichen, wie z. B. *iComment* in [13, S. 145ff.], sollen unberücksichtigt bleiben, da sie im Rahmen dieser Bachelorarbeit zu komplex sind.

Dabei sollte es nicht unbedingt das Ziel sein, dass jede Komponente dokumentiert ist, sondern dass die wichtigen Komponenten eine gute Dokumentationsqualität haben und somit die Wartung vereinfacht wird. Als Komponente im Sinne dieser Bachelorarbeit werden dabei Klassen, Schnittstellen, Methoden und Felder verstanden.

Dieses Tool soll anschließend in den CI/CD-Prozess eingebunden werden, sodass die Dokumentationsqualität kontinuierlich geprüft werden kann. Als CI/CD-Plattform soll dabei *GitHub Actions* [14] verwendet werden, da GitHub von der Mehrzahl der Entwickler und großen Unternehmen verwendet wird [15]. Mittels GitHub Actions soll das Tool

bei einer sehr schlechten Dokumentationsqualität den Entwickler auf diesen Umstand hinweisen, indem beispielsweise ein Merge (z. dt. Verschmelzung) in GitHub verhindert wird. Auch bei einer deutlichen inkrementellen Verschlechterung der Qualität soll der Entwickler informiert werden, um so eine ausreichende Qualität der Dokumentation sicherzustellen.

Ein Forschungsziel dieser Bachelorarbeit ist es zu prüfen, wie das Programm konzipiert werden muss, um mehrere Programmiersprachen zu unterstützen. Ein weiteres Ziel der Arbeit beschäftigt sich mit der Frage, wie die Ergebnisse der Metriken kombiniert werden können, um eine präzise Aussage über die Gesamtqualität der Dokumentation eines Softwareprojektes zu erhalten. Die Konzeption einer Architektur, mit der weitere Metriken hinzugefügt werden können und der Nutzer des Tools auswählen kann, welche Metriken bei der Bewertung der Dokumentationsqualität berücksichtigt werden sollen, soll ebenfalls als Forschungsziel untersucht werden. Zuletzt soll als Forschungsfrage diskutiert werden, welche Metriken eine heuristische Aussage über die Qualität der Dokumentation treffen können.

1.2 Gliederung

In Kapitel 2 werden die wichtigen Grundlagen über die Themen dieser Bachelorarbeit erläutert. Dazu wird zunächst der Begriff Softwaredokumentation definiert und ein Bezug zu Code-Smells hergestellt. Mittels Javadoc wird dann erläutert, wie Software dokumentiert werden kann. Anschließend wird eine Einführung in GitHub Actions gegeben. Zudem wird eine Einführung in ANTLR4 gegeben, das für das Parsing der Quelldateien in Java verwendet wird. Zuletzt werden einige wissenschaftliche Arbeiten mit vergleichbaren Zielsetzungen präsentiert und Tools vorgestellt, die ebenfalls die Qualität der Softwaredokumentation bewerten können.

In Kapitel 3 werden die Fragestellungen besprochen, die sich beim Design des Tools ergeben haben. Dazu gehören die notwendigen Objekte und ihre Interaktion untereinander und wie von einer losen Ansammlung von Dateien zu einer Bewertung der Softwaredokumentation gelangt werden kann.

In Kapitel 4 wird anschließend erläutert, wie aus dieser Konzeption ein vollständiges Programm entwickelt wird. Dazu wird erläutert, wie das Programm in GitHub Action eingebunden werden kann. Im Anschluss daran wird ein Überblick über die implementierten Metriken mit ihren Vor- und Nachteilen gegeben. Außerdem werden die Algorithmen bzw. Verfahren erläutert, um die Ergebnisse der einzelnen Metriken zu einem Gesamtergebnis zu aggregieren.

In Kapitel 5 wird das Programm dann mit ähnlichen Tools verglichen, indem beispielhafte Java-Projekte aus GitHub mit allen Programmen analysiert werden und die Geschwindigkeit und die Qualität jedes Programmes ermittelt wird.

Im abschließenden Kapitel wird der Inhalt der Arbeit zusammengefasst und ein Fazit gezogen. Es werden offengebliebene Fragen beleuchtet und ein Ausblick gegeben, welche Möglichkeiten zur Verbesserung des Tools sinnvoll wären.

2 Grundlagen

In diesem Kapitel werden die wichtigen Grundlagen erläutert, die nötig sind, um das Problem der mangelhaften Softwaredokumentation zu lösen. Dazu wird der Begriff „Softwaredokumentation“ genauer definiert (Kapitel 2.1). Anschließend gibt es eine Einführung in Javadoc, dessen Analyse der Schwerpunkt dieser Bachelorarbeit ist (Kapitel 2.2). In Kapitel 2.3 wird eine Einführung in Code-Smells gegeben. Anschließend erfolgt eine Einführung in TypeScript und Node.js, welche zur Entwicklung des Tools verwendet werden (Kapitel 2.4). Um später das Parsing von Quelltextdateien zu ermöglichen, wird zudem in Kapitel 2.5 die Bibliothek *ANTLR4* vorgestellt. Im darauffolgenden Kapitel 2.6 wird die CI/CD-Plattform *GitHub Actions* präsentiert, die genutzt werden soll, um die Dokumentationsqualität regelmäßig zu messen. In Kapitel 2.7 werden weitere Tools vorgestellt, die fehlerhafte Dokumentationen erkennen können. Zuletzt werden in Kapitel 2.8 wissenschaftliche Arbeiten zusammengefasst, die sich ebenfalls mit der Thematik „Softwaredokumentation“ befassen.

2.1 Softwaredokumentation

Um den Begriff „Softwaredokumentation“ zu definieren, sollte zunächst der Begriff „Dokumentation“ definiert werden. Das IEEE definiert diesen Begriff als jede textliche oder bildliche Information, welche Aktivitäten, Anforderungen, Abläufe oder Ergebnisse beschreibt, definiert, spezifiziert, berichtet oder zertifiziert [16, S. 28]. Somit beschreibt eine Dokumentation, wie eine Komponente aufgebaut ist oder wie sie sich verhält.

Diese abstrakte Definition lässt sich so auf Softwareentwicklung übertragen. In [17, S. 125] wird Softwaredokumentation als eine Sammlung von technischen Informationen dargestellt, die für Menschen lesbar sind und welche die Funktionen, Benutzung oder das Design eines Softwaresystems beschreiben. So beschreibt Donald E. Knuth in [18, S. 97], dass die Hauptaufgabe beim Programmieren nicht sein sollte, einem Computer zu erklären, was er machen sollte, sondern anderen Menschen zu erklären, was der Computer machen sollte. Beispiele für Softwaredokumentation sind Kommentare, UML-Diagramme, Readme-Dateien oder Handbücher.

Im Kontext dieser Bachelorarbeit sollen allerdings nur bestimmte Arten der Softwaredokumentation betrachtet werden, da eine umfassende Betrachtung innerhalb der vorgegebenen Zeit nicht möglich ist. Zwar wären Readme-Dateien oder UML-Diagramme für eine Bewertung auch interessant, allerdings sind diese nicht so stark mit dem dazugehörigen Code verknüpft, sondern befinden sich in externen Dateien, sodass eine Verarbeitung ungleich schwieriger wäre.

Aus diesem Grunde konzentriert sich diese Bachelorarbeit nur auf Kommentare. Studien zeigen, dass Kommentare eine hohe Relevanz in der Softwareentwicklung haben, sodass eine Analyse der Kommentare eine gute Annäherung an die tatsächliche Qualität der Dokumentation geben kann [19, S. 71]. Dabei sollen nicht alle Kommentare betrachtet werden, sondern nur eine bestimmte Kategorie von Kommentaren, die besonders

leicht verarbeitbar sind, sodass aus ihnen viele Informationen extrahiert werden können und die Bewertung dadurch genauer werden kann.

Für diese Bachelorarbeit werden daher nur bestimmte Inline-Kommentare im Quellcode betrachtet, die ein Spezialfall von mehrzeiligen Kommentaren sind. Diese Kommentare werden wie normale Kommentare erkannt und werden daher nicht Bestandteil des kompilierten Programms. Dennoch haben diese spezifischen Kommentare aber eine bestimmte Struktur, die eine leichtere Verarbeitung durch Programme ermöglicht und gleichzeitig trotzdem für Menschen lesbar bleibt. Diese Kommentare stehen als Präfix vor einer bestimmten Komponente und werden ihr so eindeutig zugeordnet. Im Folgenden werden diese Kommentare auch **strukturierte Kommentare** genannt. Ein Beispiel für solche strukturierte Kommentare ist Javadoc.

2.2 Javadoc

Javadoc [20] ist ein Tool zur Generierung von Dokumentationen, das sich als de facto Standard für Dokumentationen in der Programmiersprache Java etabliert hat [21, S. 249]. Javadoc verwendet spezielle Java-Kommentare, die an bestimmten Stellen im Quellcode eingefügt werden und daher bei der Kompilation nicht berücksichtigt werden. Ein Javadoc-Block beschreibt immer ein bestimmtes Modul (z. B. eine Klasse, Methode oder Feld). Es beginnt mit der Zeichenkette „/**“, wobei die ersten beiden Zeichen „/*“ den Beginn eines mehrzeiligen Kommentars in Java einläuten, und endet mit „*/“. Durch das zusätzliche „*“ unterscheidet sich ein Javadoc-Kommentar von einem normalen mehrzeiligen Kommentar, der zwar zur Kommentierung eines Blocks verwendet werden kann, aber vom Javadoc-Tool ignoriert wird und daher einen geringeren Mehrwert hat.

```

1  /**
2   * This method simulates a ternary operator
3   * @param condition The condition that decides the returned value
4   * @param ifTrue will be returned if {@code condition} is true
5   * @param ifFalse will be returned if {@code condition} is false
6   * @return either {@code ifTrue} or {@code ifFalse}
7   */
8  Object ternary(boolean condition, Object ifTrue, Object ifFalse){
9      return condition ? ifTrue : ifFalse;
10 }
```

Listing 2.1: Beispielhafter Javdoc-Block für einfache Methode

Listing 2.1 zeigt beispielhaft, wie eine Methode mittels Javadoc dokumentiert werden kann. Zunächst sollte am Anfang des Blocks eine generelle Zusammenfassung der Komponente stehen (Z. 2). Danach können sogenannte Block-Tags (-Tags z. dt. Auszeichner), die mit dem „@“-Zeichen beginnen, benutzt werden. Diese beschreiben wiederum einen bestimmten Teilbereich einer Komponente. In den Zeilen 3 bis 5 werden alle Parameter einer Methode beschrieben. Auch der Rückgabewert einer Methode kann so genauer

erläutert werden (Z. 6). Mittels sogenannter Inline-Tags können zudem Abschnitte in einer Dokumentation besonders gekennzeichnet werden. Hier wird mittels „{@code ...}“ deutlich gemacht, dass das Wort „condition“ nicht als Wort zu verstehen ist, sondern als Verweis auf den Parameter *condition* verstanden werden soll. Es ist zudem Konvention, dass jede Zeile in einem Javadoc-Block mit einem Asterisk beginnt.

In der folgenden Auflistung werden beispielhaft einige Javadoc-Tags aufgelistet und beschrieben. Inline-Tags werden mit geschweiften Klammern umschlossen:

@param Beschreibt einen Parameter, benötigt den Parameternamen als Argument

@return Beschreibt den Rückgabewert der Methode, sofern er existiert

@throws Beschreibt welche Ausnahmen diese Methode werfen kann und möglichst unter welchen Umständen dies passiert, benötigt den Namen der Ausnahme. Für mehrere Ausnahmen sollte dieser Tag entsprechend wiederholt werden

@deprecated Falls diese Methode veraltet ist und nicht mehr verwendet werden sollte, kann hier eine Alternative beschrieben werden

@see Ermöglicht einen Verweis auf einer anderen Komponente, um die Navierbarkeit zu erleichtern, beim Rendering werden diese Verweise in einem separaten Abschnitt angezeigt

{@code} Kann verwendet werden, um Quellcode in Dokumentationen einzubinden

{@link} Ermöglicht einen Verweis auf eine andere Komponente, um so Navierbarkeit zu erleichtern, wird an dieser Stelle in einem Link umgewandelt

Diese Javadoc-Blöcke können dann von dem gleichnamigen Tool in eine *Hyper Text Markup Language* (HTML)-Datei umgewandelt werden und ermöglichen den Entwicklern damit einen komfortablen Überblick über alle Komponenten eines Moduls. Zudem können Javadoc-Blöcke ebenfalls HTML-Inhalte besitzen, die dann von Javadoc in die HTML-Datei übernommen werden, sodass der Entwickler beispielsweise Tabellen zur übersichtlichen Präsentation von Informationen verwenden kann.

Eine *Integrated Development Environment* (IDE) kann die Informationen auslesen und dann beispielsweise bei der Autovervollständigung nutzen, um den Entwickler beim Schreiben des Programmcodes zu unterstützen. So kann sich ein Programmierer auch in einer fremden Programmbibliothek leichter zurechtfinden.

Ein Javadoc-Kommentar wird vererbt und muss daher für eine abgeleitete Klasse nicht neu geschrieben oder redundant geklont werden. Dies ist sinnvoll, da abgeleitete Klassen einen Vertrag erfüllen müssen, der bei einer guten Dokumentation auch schon in der Javadoc-Dokumentation beschrieben wird. Auch bei Methoden, die aufgrund einer Schnittstelle implementiert werden müssen, ist eine Neudefinition des Javadoc-Kommentars unnötig. Falls sinnvoll, kann aber dennoch ein eigener Javadoc-Kommentar erstellt werden, der allerdings den Kommentar der Quelle vollständig ersetzt. Mit „@inheritDoc“ kann der ursprüngliche Kommentar aber trotzdem eingefügt werden.

Nachfolgend ist ein Auszug von empfehlenswerten Tipps von der Oracle-Webseite, die beim Schreiben von Javadoc beachtet werden sollten [20]:

- Nicht in jedem Fall vollständige Sätze verwenden, aber klar formulieren, was die Aufgabe einer Komponente ist
- In der dritten und nicht in der zweiten Person schreiben
- Nicht repetitiv sein. Ein Kommentar, der im Wesentlichen nur den Namen einer Komponente wiedergibt, hat keinen Mehrwert
- Beschreibungen von Methoden sollten mit einem Verb beginnen
- Keine lateinischen Ausdrücke wie „e.g.“, „aka“ oder „i.e.“ verwenden
- Bei einem Verweis auf das aktuelle Objekt sollte das spezifische *this* statt des allgemeineren *the* verwendet werden, bspw. „opens the connection of **this** object“ statt „opens the connection of **the** object“
- Bezeichner sollten mit „<code></code>“ umschlossen werden, um deutlich zu machen, dass dies eine andere Komponente ist
- Der Kommentar sollte eventuelle Unterschiede unter verschiedene Plattformen erläutern
- Die Dokumentation sollte erläutern, wie sich die Komponente in Randfällen verhält

Für andere Programmiersprachen gibt es vergleichbare Werkzeuge, die ähnliche Funktionen anbieten und bei denen die Dokumentationen mit einer relativ ähnlichen Syntax erstellt werden. Dazu gehören beispielsweise *Doxygen* [22] für C++-Programme oder *Docstring* [23] für Python-Programme.

2.3 Code-Smells

Der Begriff „Code-Smell“ (z. dt. übel riechender Code) wurde von Kent Beck in [24, S. 71 ff.] vorgeschlagen, um Quellcode zu beschreiben, bei dem Refactoring (z. dt. Restrukturierung) sinnvoll ist. Bestimmte Strukturen im Quellcode deuten darauf hin, dass der Code verbessert werden sollte, da ansonsten die zukünftige Wartbarkeit des Codes verringert wird. Beispiele sind schlecht gewählte Namen für Variablen oder Methoden, duplizierter Code, lange Methoden oder globale Daten. In bestimmten Situationen werden Kommentare ebenfalls als Code-Smell klassifiziert, da sie auf komplizierten Code hindeuten können oder der Name einer Komponente schlecht gewählt wurde. Daher sollte geprüft werden, ob nach einem Refactoring die Kommentare noch notwendig sind.

Nach [21, S. 249–250] kann eine mangelhafte Dokumentation als Code-Smell bezeichnet werden, da sie die Wartbarkeit der Software beeinträchtigen kann. Dies gilt besonders, wenn die Dokumentation vom Code abweicht und damit bei den Entwicklern

zu Verwirrung führt. Eine fehlerhaft dokumentierte Methode, bei der beispielsweise ein Parameter nicht dokumentiert ist, deutet stark auf Code-Smells hin, da der Code von der Dokumentation divergiert. Zudem kann eine IDE die Informationen von strukturierten Kommentaren nutzen, um beispielsweise dem Benutzer einer Methode Hinweise zur Benutzung zu geben. Dies ist bei normalen Kommentaren nicht möglich.

2.4 JavaScript, Node und TypeScript

Das Tool wird mittels der *Node.js*-Bibliothek und TypeScript entwickelt. In diesem Abschnitt werden daher die grundlegenden Programmiersprachen und Bibliotheken vorgestellt, die zur Entwicklung des Tools als Desktopanwendung notwendig sind. Außerdem wird begründet, warum TypeScript und „Node.js“ für die Entwicklung des Tools verwendet wird.

JavaScript

JavaScript [25] ist eine dynamisch typisierte Programmiersprache, die sich als de facto Standard für Browserskripte etabliert hat, die auf dem Client (z. dt. Klienten) ausgeführt werden. Zum Beispiel lassen sich mittels JavaScript Webseiten dynamisieren, Eingaben von Formularen validieren oder nachträglich Daten von anderen Servern laden.

Die Syntax von JavaScript basiert wie bei der Programmiersprache Java auf C, dennoch sollten Java und JavaScript nicht verwechselt werden. Die eingangs erwähnte dynamische Typisierung bedeutet im Gegensatz zur statischen Typisierung, dass eine Variable anders als in Java keinen fixen Datentyp hat, sondern der Datentyp sich stets nach dem aktuellen Wert richtet. Eine Variable *foo* kann mit dem Befehl *let foo = true* deklariert werden und ist dann vom Datentyp *Bool*. Daraufhin kann der gleichen Variable eine Zeichenkette zugewiesen werden, sodass die Variable nun vom Datentyp *String* ist. Durch die dynamische Typisierung ist der Programmierer flexibler, jedoch zeigt eine Studie, dass die Softwarequalität bei Projekten mit statischer Typisierung etwas besser ist als mit dynamischer Typisierung [26, S. 155ff.].

TypeScript

TypeScript [27] ist eine auf JavaScript aufbauende Programmiersprache, die JavaScript um statische Typisierung ergänzt. Jeder JavaScript-Code ist gültiger TypeScript-Code, sodass TypeScript als Obermenge von JavaScript angesehen werden kann. Der TypeScript-Code kann nicht direkt ausgeführt werden, sondern wird in gültiges JavaScript transkribiert, sodass er von einem Client mit JavaScript-Unterstützung ausgeführt werden kann.

In TypeScript können Typannotationen verwendet werden, die den Datentyp einer Variablen festlegen. Eine Variable *foo* kann in TypeScript dem Befehl *let foo:boolean=true* als *Boolean* deklariert werden. Durch die Angabe „:boolean“ ist der Datentyp fix und kann nicht geändert werden. Eine Zuweisung einer Zeichenkette zu *foo* würde TypeScript mit einer Fehlermeldung quittieren. Auch viele Funktionen, die in objektorien-

tierten Programmiersprachen verbreitet sind, wie z. B. Schnittstellen, generische Typen oder Namensräume werden von TypeScript unterstützt und helfen so dabei, komplexere Programme zu schreiben. Aus diesen Gründen wird TypeScript für die Entwicklung des Projektes verwendet, denn eine Verwendung von statischen Typen und anderen objektorientierten Funktionen ermöglicht eine bessere Abstraktion der einzelnen Schritte (Traversierung, Parsing, Anwendung der Metriken etc.).

Node.js

Die Bibliothek *Node.js* [28] ist eine beliebte Bibliothek, um JavaScript-Programme nicht nur im Browser, sondern auch als eigenständige Applikation zu entwickeln. Dadurch haben sie im Gegensatz zu den klassischen Browser-Skripten freien Zugriff auf das Dateisystem und Netzwerkfunktionen. Die Bibliothek läuft auf den meisten Systemen und wird von GitHub Actions nativ unterstützt. Außerdem gibt es über den Paketmanager *NPM* eine große Auswahl an kleineren Bibliotheken, die kleinere Probleme lösen können und damit Entwicklungsaufwand bei Aufgaben sparen, welche nicht im Kernbereich dieser Bachelorarbeit liegen. Folglich wurde diese Bibliothek als Fundament für die Entwicklung des Tools genutzt.

2.5 ANTLR4

Um die Java-Dateien zu parsen (siehe Kapitel 4.3), wird die Bibliothek ANTLR4 [29] verwendet, die kostenlos verfügbar ist. Diese Bibliothek kann nicht nur viele Programmiersprachen, sondern auch selbst geschriebene Sprachen parsen, sofern eine passende Grammatik verfügbar ist.

Eine andere Möglichkeit zum Parsen der Java-Dateien ist die NPM-Bibliothek „java-parser“ [30]. Dies hätte den Vorteil, dass das Parsen, welches nicht der Hauptfokus dieser Bachelorarbeit ist, ausgelagert wird und so Fehler vermieden werden können. Allerdings kann diese Bibliothek nicht die Verbindung zwischen einer Komponente und dem dazugehörigen Javadoc-Block herstellen, sodass eine Benutzung dieser Bibliothek die Arbeit deutlich erschwert hätte. Daher wird *ANTLR4* verwendet. Außerdem besteht die Möglichkeit, eine eigene Grammatik für Java zu schreiben, da die originale Grammatik für Java sämtliche Kommentare ignoriert. Da dies jedoch deutlich komplizierter war als erwartet, wird die von vielen Entwicklern geschriebene Grammatik verwendet.

Lexer und Parser

Eine Grammatik für Programmiersprachen besteht immer aus einer Lexer-Grammatik und einer Parser-Grammatik, die jeweils vom Lexer bzw. Parser in dieser Reihenfolge verarbeitet werden, um so eine Baumstruktur einer Quelldatei zu erhalten. Der Lexer erstellt aus einer Quelldatei eine Liste von Tokens. Dabei ist ein Token eine Gruppierung von einem oder mehreren Zeichen, die eine weitergehende Bedeutung haben. Beispielsweise können Schlüsselwörter einer Programmiersprache oder Operatoren als Token klassifiziert werden. Diese Tokens sind grundsätzlich kontextunabhängig, das

heißt für die gleiche Zeichenkette wird das gleiche Token verwendet. Jeder Token wird durch seine Zeichenkette und den Tokentyp klassifiziert.

Listing 2.2 zeigt zur Verdeutlichung der Syntax eine Zeile aus der Lexer-Grammatik.

```
1 JCOMMENT:  '/*' . * ? '*' ;
```

Listing 2.2: Beispielhafte Syntax vom Lexer

Eine Zeile in der Lexer-Grammatik beginnt mit dem Namen eines Tokens, gefolgt von einem Doppelpunkt und dann einem regulären Ausdruck, der dieses Token beschreibt. In der beispielhaften Lexer-Grammatik wird ein Javadoc-Kommentar definiert, das mit „/*“ beginnt, dann folgen beliebige Zeichen und mit „*/“ endet. Der Bezeichner eines Tokens muss mit einem Großbuchstaben beginnen [31, S. 3].

Der nächste Schritt ist das Parsen. Dabei werden die im vorherigen Schritt generierten Tokens, die als Liste vorliegen, genommen und in eine hierarchische Baumstruktur umgewandelt, wobei hier der Kontext die entscheidende Rolle spielt. ANTLR4 lädt ein Token nach dem anderen und prüft, ob es – basierend auf der aktuellen Position in der Baumstruktur – eine Regel gibt, die durch dieses Token erfüllt wird. Wenn es mehrere mögliche Pfade gibt, lädt ANTLR4 so viele Tokens, bis die nächste Regel eindeutig feststeht. Gibt es dennoch Uneindeutigkeiten, so wird die Regel genommen, die als Erstes in der Parser-Datei steht [32, S. 10ff.].

So kann eine vereinfachte Parser-Grammatik für Java einen Baumknoten definieren, der eine generelle Methode beschreibt. Eine Methode besteht aus Rückgabety, Bezeichner und Parameterliste. Die Parameterliste kann dann als eine Liste von Datentyp-Bezeichner-Paare aufgelöst werden. Der Rückgabety einer Methode ist allerdings anders zu verstehen als der Datentyp eines Parameters, da jeder Datentyp eines Parameters auch ein gültiger Rückgabety ist, was jedoch nicht umgekehrt der Fall ist. So ist *void* ein gültiger Rückgabety, aber kein Datentyp. Listing 2.3 zeigt einen typischen Ausschnitt aus der Parser-Grammatik von Java [33], wobei zur Übersichtlichkeit nicht alle Regeln und Tokens aufgelistet sind.

Listing 2.3 beschreibt die notwendigen Regeln, um Methodenparameter zu parsen. Ein Methodenparameter (Z. 1) kann keinen, einen oder mehrere Modifizierer (Z. 2) enthalten, die ihrerseits eine Java-Annotation oder das Schlüsselwort *final* sein können (Z. 6–7). Anschließend (Zeile 3) muss ein Datentyp (Z. 9) erfolgen, der wiederum aus einer Annotation bestehen kann (Z. 10), gefolgt von entweder einem primitiven Datentyp oder einem Klassen- bzw. Schnittstellentyp (Z.11). Auch die üblichen eckigen Klammern für Arrays in Java können nach diesem Datentyp folgen (Z. 12). Die letzte Voraussetzung für einen gültigen Methodenparameter ist ein valider Bezeichner (Z. 4), der in Zeile 14–16 genauer definiert wird.

Die Parser-Grammatik ist ähnlich wie die der Lexer-Grammatik aufgebaut. Auch hier wird eine Variante der regulären Ausdrücke verwendet, wobei hier die reguläre Grammatik auf die Tokens angewendet wird. Anstelle des Namens des Tokens kann alternativ auch die Zeichenkette des Tokens verwendet werden (bspw. „+“ statt „ADD“). Im Gegensatz zum Lexer muss der Bezeichner einer Parser-Regel mit einem Kleinbuchstaben

```

1  formalParameter:
2      variableModifier*
3      typeType
4      variableDeclaratorId;
5
6  variableModifier:
7      FINAL | annotation;
8
9  typeType:
10     annotation*
11     (classOrInterfaceType | primitiveType)
12     (annotation* '[' ' ' ' ')*;
13
14 variableDeclaratorId:
15     IDENTIFIER
16     ('[' ' ' ' ')*;

```

Listing 2.3: Beispielhafte Syntax vom Parser

beginnen [31, S. 3]. So ist eine Unterscheidung zwischen Tokens und den Parserregeln stets möglich.

2.6 GitHub Actions

GitHub Actions [14] ist eine von GitHub angebotene Plattform zur Vereinfachung des CI/CD's. Mithilfe von GitHub Actions wird Programmcode ausgeführt, wenn ein bestimmtes Ereignis stattfindet. Dieses Ereignis kann z. B. ein Push-Ereignis sein, bei dem neuer Quellcode in das GitHub-Repository hochgeladen wird oder eine neue Version des Programms zum Release freigegeben wird. Der Programmcode kann wahlweise auf einem von GitHub vorbereiteten System ausgeführt oder auf einem eigenen System ausgeführt werden. Dieses System wird auch als **Runner** bezeichnet.

Der gesamte Programmcode wird im Kontext von GitHub Actions auch als **Workflow (Ablauf)** bezeichnet und führt einen oder mehrere **Jobs** aus. Diese Jobs sind eine Ansammlung von **Steps (Schritte)**, die man als Befehle interpretieren kann. Die Jobs werden standardmäßig parallel ausgeführt, da keine Abhängigkeit zwischen den Jobs vermutet wird. Ein Step kann ein Befehl sein, der auf einer Kommandozeile ausgeführt werden kann, oder ein Verweis auf eine andere GitHub Action, die dann ausgeführt wird. [34, S. 5–7]

Abbildung 2.1 zeigt schematisch, wie ein Workflow in einem GitHub-Projekt ausgeführt werden kann. Ein Benutzer löst beispielsweise durch einen Push ein Ereignis aus. Daraufhin werden für jeden Job ein Runner initialisiert, die aus dem Repository das Projekt klonen, um so mit einer geklonten Kopie zu arbeiten. Das Klonen des Repositorys ist dabei nicht obligatorisch, ist aber notwendig, falls mit dem Quelltext des Projektes gearbeitet werden soll. Anschließend werden die Jobs gestartet, die ihrerseits wieder

2 Grundlagen

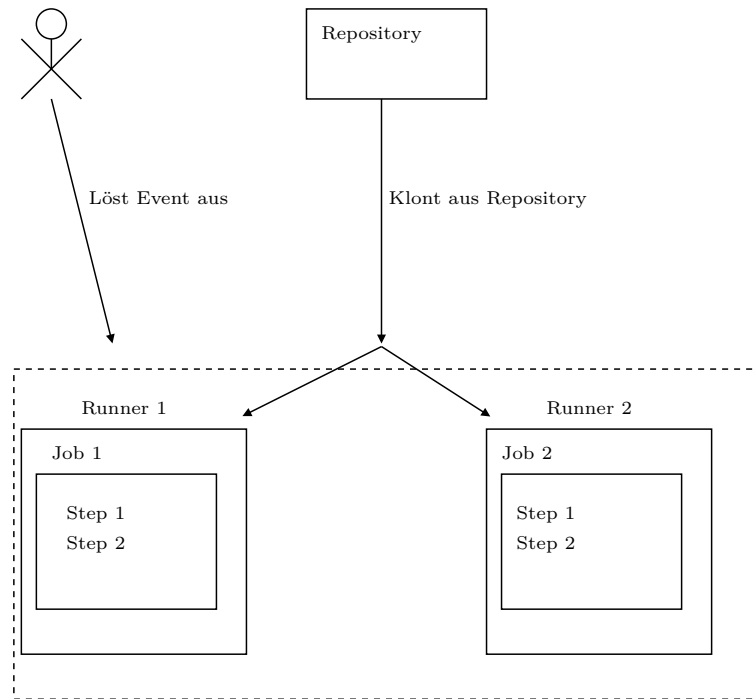


Abbildung 2.1: Schema: Workflows

die Steps ausführen. Jeder Job wird auf einem separaten Runner ausgeführt und läuft parallel, außer der Benutzer gibt explizit eine Abhängigkeit zwischen den Jobs an.

Schlägt der Programmcode fehl, kann der Nutzer den Grund des Fehlers über die „Actions“-Registerkarte herausfinden. Dort kann der Nutzer auch sämtliche Ausgaben des Programms ansehen, die auf der Konsole ausgegeben werden.

Verwendungsmöglichkeiten für GitHub Actions

Ein Anwendungsfall von GitHub Actions sind automatisierte Tests. Bei einem Push-Ereignis kann der aktuelle Programmcode mit einer geeigneten Testbibliothek getestet werden, sodass im Falle eines fehlgeschlagenen Tests der Programmierer informiert wird und die notwendigen Änderungen veranlassen kann. Des Weiteren kann das Deployment (z. dt. Verteilung) der Software mittels einer Action automatisiert erfolgen. So lässt sich z. B. bei Programmen, die auf verschiedenen Plattformen (NPM, Yarn, NuGet etc.) veröffentlicht werden, sicherstellen, dass stets die aktuelle Version auf jeder Plattform verfügbar ist. So können Aktualisierungen schnell verteilt werden, was auch zu einer höheren Softwarequalität und einem besseren Schutz vor Sicherheitslücken führen kann. Auch die Qualität des Codes kann mit geeigneten Actions geprüft werden, was im Kontext dieser Bachelorarbeit besonders interessant ist. [34, S. 1ff.]

Erstellung eines Workflows

Ein Workflow kann über die Registerkarte „Actions“ erstellt werden und wird intern in dem Verzeichnis „github/workflows“ gespeichert. Abbildung 2.4 illustriert eine typische Workflow-Datei, die standardmäßig erstellt wird.

```

1  name: CI
2  on:
3    push:
4      branches: [ main ]
5    pull_request:
6      branches: [ main ]
7    workflow_dispatch:
8  jobs:
9    build:
10     runs-on: ubuntu-latest
11     steps:
12       - uses: actions/checkout@v2
13       - name: Run a one-line script
14         run: echo Hello, world!
15
16       - name: Run a multi-line script
17         run: |
18           echo Add other actions to build,
19           echo test, and deploy your project.
```

Listing 2.4: Beispielhafte Workflow-Datei

In Zeile 1 wird der Name des Workflows festgelegt. Anschließend werden die Bedingungen festgelegt, die den Workflow starten. In diesem Beispiel wird dieser Workflow bei einem Push (Z. 3) oder ein Pull-Request (Z. 5) ausgeführt, wenn dabei die Main-Branch betroffen ist. Mit „workflow_dispatch“ wird zusätzlich ermöglicht, den Workflow auch manuell zu starten, was zu Debugging-Zwecken sinnvoll sein kann. In Zeile 8 werden dann die Jobs definiert, wobei hier nur ein Job namens „build“ (Z. 9) erstellt wird. Der Job benötigt einen Runner, der hier ein System mit einer aktuellen Ubuntu-Version ist (Z. 10). Als Nächstes werden die einzelnen Schritte deklariert. Jeder Schritt beginnt mit einem Bindestrich. Der erste Schritt (Z. 12) sorgt dafür, dass der Quellcode der entsprechenden Branch des Repositorys auf das System geklont wird. Dabei wird auch eine Umgebungsvariable namens „\$GITHUB_WORKSPACE“ bereitgestellt, die den Pfad des geklonten Repositorys enthält. Der nächste Schritt (Z. 13–14) gibt den Text „Hello, world!“ auf der Kommandozeile aus. Der letzte Schritt (Z. 17–19) zeigt exemplarisch, dass auch mehrere Kommandozeilenbefehle sequenziell ausgeführt werden können.

Ausführung von Workflows

Sobald ein Workflow durch ein Ereignis ausgeführt wird, kann ein Benutzer den Programmablauf live über die Registerkarte „Actions“ einsehen. Nachdem der Workflow abgeschlossen wurde, können hier auch Informationen über den Erfolg bzw. Misserfolg

abgerufen werden. Beispielsweise können die Ausgaben auf der Konsole angesehen werden, die bis zu 90 Tage gespeichert werden [34, S. 53]. Des Weiteren kann ein Benutzer auch erfahren, wie viel Zeit der Workflow benötigt hat, wobei hier auch der Zeitbedarf der einzelnen Schritte angezeigt wird.

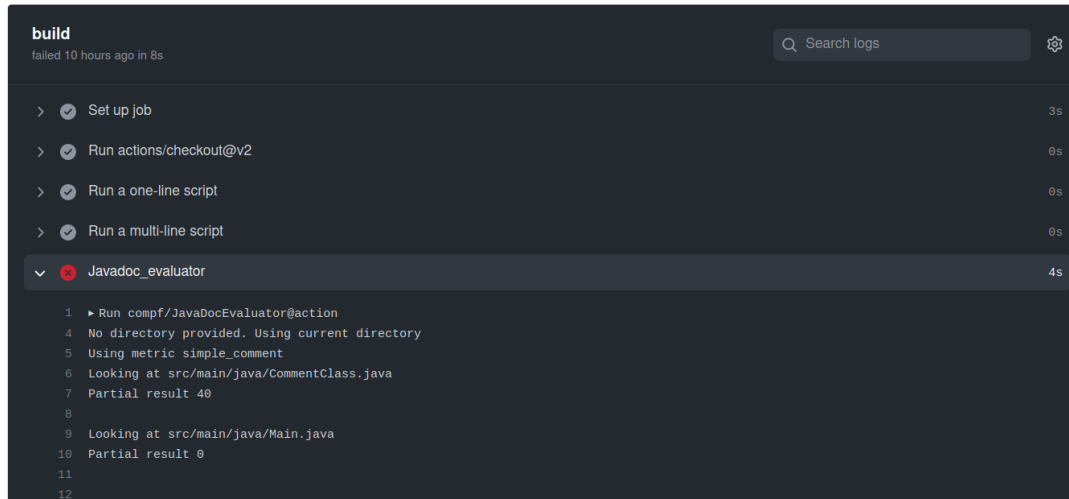


Abbildung 2.2: Ausgabe eines Workflows

Abbildung 2.2 zeigt, wie ein Benutzer den Ablauf eines Workflows einsehen kann. Der Nutzer kann durch einklappbare Gruppen die Ausgaben der einzelnen Schritte einsehen, sodass die Schritte für die Initialisierung etc. nicht die Übersicht stören. Zudem befindet sich auf der rechten Seite der Zeitbedarf eines Schrittes. Durch das rote Kreuz erfährt ein Nutzer, dass dieser Schritt fehlgeschlagen ist.

Erstellung einer eigenen Action

Um eine eigene GitHub Action zu erstellen, muss in dem Hauptverzeichnis des Repositories, in dem der Programmcode der Action liegt, eine Datei namens „action.yaml“ oder „action.yml“ erstellt werden. Listing 2.5 zeigt eine beispielhafte „action.yaml“. Zunächst wird der Name der Action und eine Beschreibung definiert (Z. 1–2). Anschließend können Eingabeparameter definiert werden, die später im Programm verwendet werden können (Z. 3–7). Zu jedem Parameter muss auch festgelegt werden, ob er zwingend erforderlich ist. Außerdem kann der Standardwert des Parameters definiert werden. Zusätzlich können Ausgabeparameter festgelegt werden, die spätere Actions als Eingabe nutzen können (Z. 8–10). Danach (Z. 11–13) wird festgelegt, wie diese Action ausgeführt wird. Eine Action kann in einer JavaScript-Umgebung, in einem Docker-Container oder als Liste von verschiedenen Schritten ausgeführt werden [34, S. 117ff.].

Verhindern von Merge

Eine fehlgeschlagene GitHub Action kann verwendet werden, um einen Merge zu verhindern. Dies geschieht allerdings nicht automatisch, sondern muss explizit aktiviert

```

1  name: 'Hello World'
2  description: 'Greet someone and record the time'
3  inputs:
4    who-to-greet: # id of input
5      description: 'Who to greet'
6      required: true
7      default: 'World'
8  outputs:
9    time: # id of output
10     description: 'The time we greeted you'
11  runs:
12    using: 'node12'
13    main: 'index.js'

```

Listing 2.5: Beispielhafte Action-Konfigurationsdatei

werden. Dazu muss in den Einstellungen des Repositorys, bei dem durch eine fehlgeschlagene Aktion ein Merge verhindert werden soll, eine „Branch protection rule (z. dt. Zweigschutzregel)“ aktiviert werden. In dieser Regel kann festgelegt werden, dass ein Merge abgelehnt wird, wenn ein bestimmter Job eines Workflows nicht erfolgreich war. Ein Merge kann trotz des Fehlschlages durch einen Administrator erzwungen werden.

2.7 Verwandte Werkzeuge

Um die Qualität der Softwaredokumentation in Javadoc zu bewerten, gibt es bereits einige Programme. In vielen Fällen handelt es sich dabei um Programme, die nicht nur auf die Bewertung der Softwaredokumentation beschränkt sind, sondern auch andere Fälle von unsauberem Code erkennen können. Da die Bewertung der Dokumentation nicht der primäre Zweck der Programme ist, sind die verwendeten Algorithmen recht allgemein und erkennen viele Problemfälle nicht. Gleichwohl sind diese Programme dadurch, dass sie viele „Code-Smells“ erkennen, geeignet, um sich ein Bild über die Qualität des Quellcodes zu verschaffen und den Entwickler zu Clean-Code zu motivieren.

Checkstyle

Ein bekanntes Programm ist *Checkstyle* [35]. *Checkstyle* ist ein Open-Source-Programm, das Verstöße gegen bestimmte Fehlern in Java-Quellcode erkennt und Warnungen dazu anzeigt. Mit *Checkstyle* lässt sich beispielsweise prüfen, ob alle Methoden überhaupt einen Javadoc-Block besitzen, der Javadoc-Block korrekt platziert ist oder ein Javadoc-Tag wie zum Beispiel „@param“ auch mit einer zusätzlichen Begründung versehen ist. Außerdem kann *Checkstyle* überprüfen, ob HTML-Tags korrekt geschlossen sind und jede Zeile im Javadoc-Block mit einem Asterisk beginnt.

Für jeden Verstoß gibt *Checkstyle* eine Zeile aus, welche dem Anwender bei der Behebung des Problems helfen soll. Listing 2.6 zeigt eine typische Zeile, die Checkstyle

ausgibt, wobei zur Übersichtlichkeit diese einzelne Zeile auf mehrere Zeilen verteilt wurde:

```
1 [ERROR]
2 FigCompartment.java:
3 466:5:
4 Missing a Javadoc comment.
5 [MissingJavadocMethod]
```

Listing 2.6: Beispielhafte Ausgabe von Checkstyle

Diese Zeile enthält die Schwere des Verstoßes (Z. 1), den absoluten Pfad der betroffenen Datei (Z. 2), welcher hier aufgrund der Länge gekürzt ist, die Zeilen- und Spaltennummer (Z. 3), in der das Problem auftritt, eine Beschreibung des Fehlers (Z. 4) und einen Fehlercode (Z. 5). Die Schwere eines Verstoßes kann für jeden Fehler konfiguriert werden. Eine vollständige Auflistung der Fehler, die *Checkstyle* bezüglich der Dokumentation finden kann, existiert in [36].

Checkstyle lässt sich mit einer Extensible-Markup-Language-Datei konfigurieren, die festlegt, welche Regeln von *Checkstyle* überprüft werden sollen. Dabei kann beispielsweise festgelegt werden, ob Kommentare bei privaten Komponenten nicht zwingend notwendig sind oder kurze Methoden nicht dokumentiert sein müssen. *Checkstyle* lässt sich zudem in IDEs wie *Eclipse* einbinden, um den Entwickler leicht auf Fehler hinzuweisen.

PMD

Des Weiteren gibt es das Programm PMD [37], das ebenfalls Fehler in Softwaredokumentationen finden kann. Anders als *Checkstyle* unterstützt PMD auch andere Sprachen als Java, sodass beispielsweise einige Fehler in JavaScript gefunden werden können. Dazu gehören aber keine Fehler bezüglich der Dokumentation.

Im Bereich Javadoc erkennt *PMD* wie *Checkstyle* nicht dokumentierte Komponenten. Zudem kann *PMD* bestimmte Wörter erkennen, die als anstößig oder schwieriger verständlich empfunden werden. Auch eine maximale Länge eines Kommentars in puncto maximale Anzahl an Zeichen pro Zeile und maximale Anzahl an Zeilen pro Kommentar kann festgelegt werden. Allerdings sollte beachtet werden, dass diese Regeln für alle Kommentare (einschließlich einzelzeilige und normale mehrzeilige) gelten und damit nicht auf Javadoc beschränkt sind.

Ähnlich wie *Checkstyle* gibt *PMD* gefundene Verstöße auf der Konsole aus. Die Struktur dieser Ausgabe ähnelt sehr der von *Checkstyle*, da sie ebenfalls den vollständigen Pfad, die Zeilennummer und einen Fehlercode mit einer Beschreibung enthält. Nur die Spaltennummer und einen Hinweis auf den Schweregrad des Verstoßes existiert nicht. Ähnlich wie *Checkstyle* kann *PMD* mittels einer Extensible-Markup-Language-Datei konfiguriert werden.

Javadoc

Das oben erwähnte Javadoc-Tool bietet ebenfalls die Möglichkeit, beim Generieren der HTML-Datei die Qualität des Javadocs zu prüfen. Dabei werden vor allem fehlende Tags für Parameter etc. bemängelt. Zudem erkennt Javadoc auch Tabellen mit fehlender Überschrift und andere Designmängel, die in der HTML-Seite später auffallen. Auch fehlerhaftes HTML kann erkannt werden.

2.8 Verwandte Arbeiten

Neben den praktischen Tools, die den Softwareentwickler bei der Bewertung der Softwaredokumentation unterstützen, wurden auch einige wissenschaftliche Arbeiten veröffentlicht, die sich mit Metriken über Softwaredokumentation auseinandersetzen. In diesen Arbeiten werden auch Programme vorgestellt, welche diese Metriken nutzen, allerdings sind die Quelltexte bzw. lauffähige Versionen der Programme ohne Weiteres nicht verfügbar, sodass ein Vergleich mit dem zu entwickelnden Tool nicht möglich ist.

Einige wissenschaftliche Arbeiten haben im weiteren Verlauf der Bachelorarbeit keine Relevanz und werden daher nur der Vollständigkeit halber erwähnt. Beispielsweise ermöglicht das Programm *iComment*, welches in [13, S. 145ff.] beschrieben wird, Inkonsistenzen zwischen Dokumentationen und Quellcode in C-Programmen zu finden. Die Autoren verwenden hierfür komplexere NLP-Techniken und führen eine statische Quellcodeanalyse durch. Da diese Themenblöcke den Rahmen dieser Bachelorarbeit sprengen würden, werden sie nicht weiter verfolgt.

Quasoledo

In [38, S. 4–10] geben die Autoren einige Metriken an, die sie für die Bewertung der Softwaredokumentation als nützlich erachten, und stellen ein Programm namens Quasoledo vor, das mittels dieser Metriken Softwaredokumentationen analysieren kann. Zu den Metriken gehören z. B. *ANYJ*, welches den Anteil der dokumentierten Deklarationen an allen Deklarationen beschreibt und *Flesch-Kindcaid*, welches die Lesbarkeit von Texten beschreibt.

Die Arbeit wertet anschließend die Javadoc-Dokumentation von Eclipse aus und findet heraus, dass öffentliche Methoden häufiger dokumentiert sind und ihre Dokumentation oft lesbarer ist. Außerdem wenden die Autoren das Tool auf die gesamte Versionshistorie von Eclipse bis zum 8. April 2006 an und plotten den Verlauf der Dokumentationsqualität über diesem Zeitraum. Dabei wird sichtbar, dass die Qualität am Anfang des Projektes noch sehr stark schwankt, aber nach einiger Zeit recht konstant bleibt.

JavaDocMiner

Eine weitere Arbeit, die eine der Grundlagen dieser Bachelorarbeit ist, ist der *JavaDocMiner* [3, S. 68–79]. In diesem Konferenzpapier wird ein Programm beschrieben, welches

mittels einfacher Heuristiken eine Einschätzung der Softwaredokumentation ermitteln kann.

Insgesamt verwendeten die Autoren viele Metriken aus dem Tool Quasoleo, führen aber auch neue Metriken ein. Dazu gehört eine Metrik, welche die Anzahl der Abkürzungen zählt, da diese laut den Autoren vermieden werden sollten. Auch die durchschnittliche Anzahl an Wörtern pro Javadoc-Kommentar wird als Metrik verwendet.

Anschließend wenden die Autoren das Tool auf den Quellcode von *ArgoUML* und *Eclipse* an und prüfen, ob es eine Korrelation zwischen der Anzahl an Bugs und der Qualität der Dokumentation nach den benannten Metriken gibt. Die Autoren finden dabei heraus, dass die *Kincaid*-Metrik nur wenig mit der Anzahl der Bugs korreliert, während es bei anderen Metriken wie z. B. *ANYJ* eine starke negative Korrelation gibt.

Javadoc-Fehler in Open-Source-Software

In [21, S. 249ff.] analysieren die Autoren 163 Open-Source-Projekte aus GitHub, die in Java geschrieben sind, und prüfen, wie oft Komponenten nicht mit Javadoc kommentiert sind oder die Dokumentation unvollständig ist. Dabei werden private Komponenten, Getter und Setter von der Analyse ausgeschlossen, da sie laut den Autoren nicht zwangsläufig dokumentiert sein müssen.

Das Ergebnis der Studie lautet, dass Methoden besonders oft Javadoc-Fehler aufweisen. Wird dies allerdings in Relation zur Häufigkeit der jeweiligen Komponente gesetzt, so haben Konstruktoren eine hohe Wahrscheinlichkeit, fehlerhaftes Javadoc zu besitzen. Dies könnte laut den Autoren daran liegen, dass Konstruktoren als selbsterklärend wahrgenommen werden. Außerdem ist der Grund für Javadoc-Fehler in den meisten Fällen, dass eine Komponente komplett undokumentiert ist. Auch undokumentierte „@throws“-Tags, welche die möglichen Ausnahmen einer Methode beschreiben, sind sehr häufig. Deutlich seltener als undokumentierte Ausnahmen sind undokumentierte Parameter und Rückgabewerte. Zudem stellt die Studie fest, dass solche Javadoc-Fehler im Mittel über zwei Jahre nicht behoben werden, sodass die Softwaredokumentation oft unvollständig ist.

@tComment

Ein anderer Ansatz, der ebenfalls mit Javadoc arbeitet, wird *@tComment* genannt [39, S. 1ff.]. Dabei wurde die Konsistenz der Javadoc-Dokumentation mit dem tatsächlichen Programmcode verglichen. Anders als bei den vorherigen Ansätzen wird hier das Programm dynamisch ausgeführt. Dabei werden die verschiedenen Methoden und ihr dazugehörige Javadoc-Dokumentation analysiert und daraus mittels einfacher Heuristiken ermittelt, ob ein Nullwert für einen Parameter eine Inkonsistenz zwischen Dokumentation und Quellcode offenbart.

Beispielsweise kann die Dokumentation einer Java-Methode beschreiben, dass ein Parameter nicht *null* sein darf. Daraus kann gefolgert werden, dass ein Nullwert für diesen Parameter zu einer Ausnahme führt. Wenn diese Ausnahme dann auch in der Dokumentation genauer spezifiziert ist, sollte genau diese Ausnahme bei einem Nullwert geworfen

werden. Falls Nullwerte explizit erlaubt sind, sollte die Methode sich dann stets korrekt verhalten und keine Ausnahmen werfen.

@tComment kann mit diesen einfachen Regeln jede Methode ausführen und prüfen, ob sich die Methode tatsächlich verhält, wie die Dokumentation es beschreibt. Dabei verwendet das Programm einfache semantische Heuristiken in der englischen Sprache. Zum Beispiel prüft das Programm, ob die kurz vor dem Wort „null“ ein negierendes Wort wie „not“ oder „never“ auftaucht und klassifiziert dann den dazugehörigen Parameter als Parameter, der nie *null* sein darf.

3 Konzeption

Im Folgenden wird ein Konzept für die Implementierung des Tools vorgestellt, das die Ziele der Bachelorarbeit erfüllen soll. Um diese Ziele zu erreichen, müssen zuerst die Dateien geparkt werden, was in Kapitel 3.1 besprochen wird. Im darauffolgenden Kapitel 3.2 wird erläutert, wie die extrahierten Informationen von den Metriken verarbeitet werden.

3.1 Parsing von Dateien

Eine Quellcodedatei besteht zunächst aus reinem Text und bietet daher nur indirekt Einblick in die innere Struktur. Damit eine Bewertung der Dokumentationsqualität möglich ist, müssen bestimmte Strukturen aus einer Datei gelesen werden, die für die Dokumentation wichtig sind. Andere Strukturen können dabei ignoriert werden. Für die Bewertung der Dokumentation sind nur wenige Bestandteile relevant. Beispielsweise sind alle Conditional-Branches (wie z. B. For-Schleifen und If-Verzweigungen) und viele andere Komponenten in Methodenrümpfen nicht relevant, da diese nur mit normalen Kommentaren und nicht mit Javadoc kommentiert werden (sie werden dennoch unstrukturiert als Zeichenkette gespeichert, damit Metriken diese Information eventuell nutzen können). Aus diesem Grund müssen die notwendigen Informationen extrahiert werden. Zudem ist es ein Ziel der Arbeit, eine Erweiterbarkeit auf andere objektorientierte Programmiersprachen zu ermöglichen. Daher müssen die Informationen in ein abstraktes Format gebracht werden, welches eine gute Annäherung für die meisten objektorientierten Programmiersprachen ist. Beispielsweise unterscheiden sich die Zugriffsmodifizierer vieler Programmiersprachen, sodass eine einheitliche Schnittstelle schwer umsetzbar ist. Daher enthält die abstrakte Repräsentation nur Informationen darüber, ob eine Komponente als öffentlich markiert ist. Dies ist sinnvoll, da öffentliche Komponenten als Teil der öffentlichen Schnittstelle eher dokumentiert werden sollten als nicht öffentliche und eine weiter gehende Differenzierung kaum Vorteile bietet. In einigen Programmiersprachen wie z. B. Python gibt es keine expliziten öffentlichen Komponenten, jedoch existieren de

facto Standards für Bezeichner, sodass beispielsweise private Komponenten zwei Unterstriche als Präfix haben.

Außerdem werden in der abstrakten Repräsentation die Vererbung etwas vereinfacht dargestellt, indem nicht zwischen Basisklassen und Schnittstellen unterschieden wird, da es auch hier Unterschiede zwischen Programmiersprachen gibt. Zudem werden Konstruktoren als Methoden mit den Namen „constructor“ und Schnittstellen als Klassen repräsentiert, da auch hier eine zu feine Spezifikation nicht notwendig sein wird.

In anderen Fällen gibt es jedoch viele Gemeinsamkeiten zwischen objektorientierten Programmiersprachen. So gibt es in allen relevanten Sprachen Klassen, Methoden und Felder, die alle einen Namen haben. Des Weiteren haben Methoden und Felder einen (Rückgabe-)Typen und Methoden besitzen Parameter, die ihrerseits durch einen Namen und einen Typen definiert sind. Einige Sprachen sind zwar nicht stark typisiert, jedoch kann für nicht bekannte Datentypen ein Alias wie „Any“ oder „Object“ verwendet werden. Zudem sind viele Komponenten hierarchisch. In den meisten Sprachen können beispielsweise Klassen andere Klassen enthalten, sodass diese abstrakte Struktur diese Tatsache berücksichtigen müsste.

Um dennoch sprachspezifische Funktionen anbieten zu können, besitzt jede Komponente ein Feld mit dem Typen *ComponentMetaInformation*, das wie oben erwähnt die Information enthält, ob eine Komponente als öffentlich angesehen werden soll. Dieser Typ, welches eine Schnittstelle ist, kann von einer Klasse implementiert werden, um Parser für andere Programmiersprachen die Möglichkeit zu geben, zusätzliche sprachspezifische Informationen zu speichern. Beim Java-Parser wird diese Funktion beispielsweise genutzt, um zu speichern, welche „checked“ Ausnahmen eine Methode werfen kann, sodass später ein Vergleich mit dem Javadoc-Block möglich ist. Die Schnittstelle enthält nur die Anforderung, eine *isPublic*-Methode anzubieten und kann daher für viele andere objektorientierte Programmiersprachen Informationen speichern, die für einige Metriken eventuell nützlich sind.

Abbildung 3.1 veranschaulicht, wie eine einfache Datei in die Objektstruktur umgewandelt werden kann. Dabei werden eine einfache Java-Datei und eine semantisch äquivalente Python-Datei als Beispiel verwendet, um zu zeigen, dass aus beiden Sprachen eine gleiche Objektstruktur erzeugt werden kann. Dabei wurden zur Übersichtlichkeit einige nicht relevanten Attribute entfernt.

Das Programm in beiden Sprachen besteht aus einer öffentlichen Klasse *Main* und einer privaten Methode *test*, die einen Parameter *a* als Ganzzahl erhält und eine Ganzzahl zurückgibt. Die höchste Hierarchieebene ist immer ein *FileComponent*. Diese Datei enthält hier genau ein Kind namens *classObj*, könnte aber in anderen Fällen auch mehrere Kinder (wie z. B. Klassen enthalten). Die Klasse besitzt zudem einen Verweis auf ihren Elternteil. Außerdem enthält das *classObj* eine Referenz auf Metainformationen, die hier nur angeben, dass die Klasse öffentlich ist. In diesen Metainformationen könnten auch andere relativ sprachspezifische Informationen definiert werden, falls sie für Metriken relevant sein könnten. Die Klasse enthält wiederum genau die Methode als einziges Kind. Die Methode hat ebenfalls einen Verweis auf die Metainformationen, welche die Methode als privat markieren. Außerdem hat die Methode einen *returnType* und einen Verweis auf die Liste der Parameter, die wiederum aus einem Namen und einem Datentyp beste-

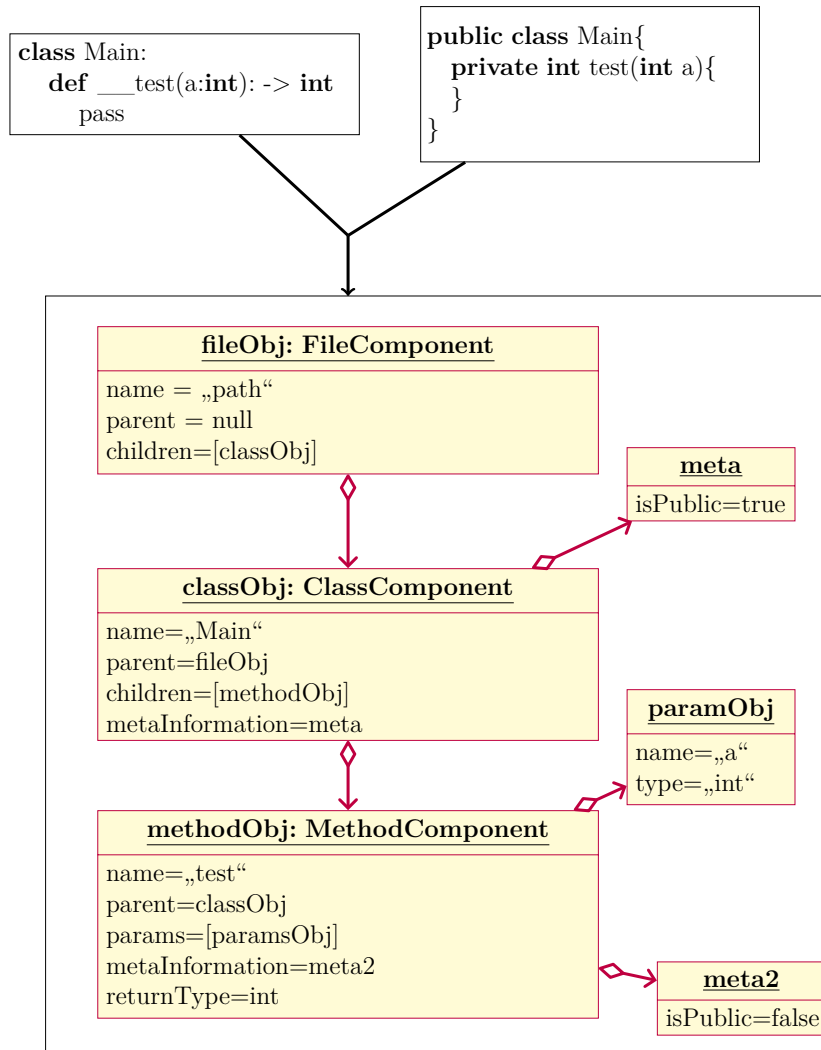


Abbildung 3.1: Objektdiagramm aus Java- und Python-Code

hen. Das Feld *comment*, welches einen Verweis auf den strukturierten Kommentar liefert, wird in dieser Abbildung nicht dargestellt, da es bei den Dateien keine Dokumentation gibt und es somit den Wert *null* hat.

Die Beziehungen der Klassen, die für das Parsing zuständig sind, werden zusätzlich in Abbildung B.1 im Anhang B. als UML-Diagramm illustriert.

Repräsentation der strukturierten Kommentare

Neben der hierarchischen Repräsentation der einzelnen Komponenten müssen auch die strukturierten Kommentare (wie z. B. Javadoc) geeignet in eine Datenstruktur umgewandelt werden. Wie in Kapitel 2.2 erläutert, besteht ein strukturierter Kommentar in vielen Fällen aus zwei Teilen. Der erste Teil ist eine allgemeine Beschreibung der Komponente. Im zweiten Teil werden bestimmte Strukturen genauer erläutert. So können einzelne Parameter erklärt werden oder der Rückgabewert beschrieben werden. Dieser

Aufbau findet sich auch in *Doxygen* [22] oder *Docstring* [23], sodass diese Struktur als Grundlage genommen wird.

Ein strukturierter Kommentar besteht also aus einer generellen Beschreibung, die auch weggelassen werden kann. Anschließend folgen null bis beliebig viele Tags. Jeder Tag besteht aus einem Typ (z. B. „@param“, „@return“ oder „@throws“), einem optionalen Parameter, welcher von einigen Tags benötigt wird und der Beschreibung des Tags. Die Namen der Tags sind generalisiert, dies bedeutet, dass unabhängig von der Programmiersprache der Tag zur Beschreibung eines Parameters immer „@param“ heißen muss. Dies muss bei der Entwicklung eines Parsers beachtet werden.

Ein strukturierter Kommentar kann ebenfalls spezielle Elemente (wie z. B. HTML-Elemente oder Inline-Tags) enthalten. Hier werden diese Elemente unverarbeitet gelassen, also unverändert mit dem Rest des Kommentars als Zeichenkette gespeichert. Metriken, die diese Informationen benötigen, müssen diese Elemente also selbstständig extrahieren. Andere Metriken (wie z. B. die Metriken in Kapitel 4.8.2) benötigen diese speziellen Elemente nicht, sodass eine einheitliche Schnittstelle, die sowohl die natürliche Sprache als auch die speziellen Elemente berücksichtigt und zudem relativ programmiersprachenunabhängig sein müsste, viele Herausforderungen mit sich bringt.

Wird kein strukturierter Kommentar angegeben, so liefert der entsprechende Getter *getComment* den Wert „null“ zurück.

3.2 Konzeption der Metriken

Nachdem eine Datei in ihre einzelnen Komponenten zerlegt wurde, kann die Qualität der Softwaredokumentation überprüft werden. Jede gefundene Komponente besitzt einen Verweis auf die dazugehörige Dokumentation, die bei Nichtvorhandensein auch null sein kann. Anhand dieser Referenz kann geprüft werden, ob die Softwaredokumentation der Komponente ausreichend ist. Zur Bewertung der Dokumentation gibt es verschiedene Möglichkeiten. Beispielsweise könnte überprüft werden, ob eine Komponente dokumentiert oder undokumentiert ist. Eine weitere Möglichkeit wäre es, die Verständlichkeit der Dokumentation zu prüfen. Alle diese Vorgehensweisen basieren auf Metriken, die auf wissenschaftliche Studien beruhen oder zumindest plausibel sind. In diesem Abschnitt wird ein Konzept erläutert, um eine Metrik zu implementieren. Anschließend wird beschrieben, wie die Ergebnisse jeder Metrik zusammengefasst werden, um ein Endresultat zu erhalten.

3.2.1 Implementation einer Metrik

Damit eine Metrik die Dokumentation bewerten kann, benötigt sie Zugriff auf die Komponente. Außerdem muss sie ihr Ergebnis irgendwie veröffentlichen bzw. zwischenspeichern, damit es später weiterverarbeitet werden kann. Des Weiteren ist nicht jede Metrik mit jeder Komponente kompatibel. Eine Metrik, die überprüft, ob jeder Methodenparameter dokumentiert ist, kann diese Aufgabe bei anderen Komponentenarten nicht erfüllen. Zudem sollte es die Möglichkeit geben, das Verhalten einer Metrik mittels Pa-

parameter anzupassen, damit die Metrik konfigurierbar bleibt. Außerdem sollte eine Metrik bei der Bewertung auch begründen, warum die Dokumentation einer Komponente nicht ausreichend ist. Zuletzt soll der Benutzer des Tools selbst auswählen können, welche Metriken angewendet werden sollen, da nicht jede Metrik immer sinnvoll ist.

Eine Möglichkeit, diese Anforderung für eine Metrik umzusetzen, ist die Verwendung einer Methode pro Metrik, welche die Komponente und die Parameter als Eingabe erhält und daraus die Bewertung und eventuelle Begründung ermittelt und zurückgibt. Allerdings ist dieser Ansatz sehr prozedural, denn es gibt beispielsweise keine Kapselung zwischen den Metriken.

Ein anderer Ansatz, der hier auch gewählt wird, ist es, jede gewünschte Metrik als Klasse zu implementieren. Jede implementierte Metrik kann somit die notwendigen Berechnungen abgekapselt von anderen Metriken erledigen, was die Wartbarkeit verbessert. Um trotzdem für eine einheitliche Schnittstelle zu sorgen, muss jede zu implementierende Metrik von einer abstrakten Basisklasse (*DocumentationAnalysisMetric*) erben. Wenn eine neue Metrik implementiert werden soll, muss eine neue Klasse erstellt werden, die von dieser abstrakten Basisklasse erbt. Eine Instanz dieser Klasse wird im Folgenden **Metrikobjekt** genannt und repräsentiert eine konkrete Implementierung der Metrik, die bei der Bewertung der Dokumentation berücksichtigt werden kann. Um diese Bewertung durchzuführen, muss geprüft werden, ob eine Metrik mit einer Komponente kompatibel ist. Dies wird durch die Methode *shallConsider* erledigt, welche dementsprechend einen Wahrheitswert zurückgibt. Die anschließende Analyse erfolgt durch die Methode *analyze*. Diese führt den metrikspezifischen Algorithmus aus und speichert das Ergebnis wie in Kapitel 3.2.2 beschrieben, damit es zu einem Gesamtergebnis verarbeitet werden kann.

Da eine Metrik auch parametrisierbar sein soll, müssen bei der Instanziierung eines Metrikobjektes Parameter übergeben werden, die von der Implementation der Metrik zur Modifikation des Verhaltens der Metrik verwendet werden können. Diese Parameter werden sehr abhängig von der Metrik sein, sodass eine einheitliche Schnittstelle nur schwer umsetzbar ist. Daher werden die Parameter als Datentyp *any* übergeben, sodass es keine Typüberprüfung gibt. Alternativ wäre eine assoziative Liste möglich, bei dem ein Parametername als Zeichenkette ein Wert zugeordnet wird, aber auch hier könnte keine Überprüfung eines Datentypes vorgenommen werden.

Eine weitere Voraussetzung für ein Metrikobjekt ist ein eindeutiger Name. Dadurch kann die gleiche Metrik mit unterschiedlichen Parametern verwendet werden. Außerdem wird so eine Zuordnung von Gewichten vereinfacht. Standardmäßig besteht dieser eindeutige Name aus dem Namen der implementierten Metrik, gefolgt von einem Unterstrich und einer fortlaufenden Nummer.

Ein UML-Diagramm der relevanten Klassen, die für die Berechnung der Dokumentationsqualität zuständig sind, befindet sich in Abbildung C.1 im Anhang C.

Bewertung der Dokumentation

Die Methode *analyze* muss eine Bewertung darüber abgeben, ob die Qualität der Dokumentation ausreichend ist. Für die Repräsentation dieser Bewertung gibt es viele Mög-

lichkeiten, allerdings ist eine numerische Bewertung mittels einer Intervallskala am sinnvollsten, da so der arithmetische Mittelwert, der Median etc. berechnet werden kann, was für die Bildung des Gesamtergebnisses wichtig ist. Die numerische Bewertung soll eine Aussage über die Dokumentationsqualität liefern. Eine Bewertung von 0 steht für eine sehr schlechte bis nicht existente Dokumentation und die Bewertung 100 steht für eine exzellente Dokumentation, sodass die Bewertung sich als Prozent lesen lassen kann. Das Ergebnis einer implementierten Metrik sollte diesen Wertebereich nicht verlassen, da eine Fehlerbehandlung nicht implementiert ist. Bei Metriken, die per Design schon einen prozentualen Wert zurückgeben, wird diese Vorgabe stets eingehalten. Bei anderen Metriken (z. B. die Flesch-Metrik in Kapitel 4.8.2) sollte eine mathematische Funktion gefunden werden, die das Ergebnis der Metrik auf den Wertebereich 0 bis 100 abbildet. Die genaue Umsetzung hängt von der Metrik ab. In jedem Falle sollte es für eine Metrik Ergebnisse geben, die auf eine gute bzw. schlechte Dokumentation hindeuten, damit diese auf 100 bzw. 0 abgebildet werden können. Nur durch diese Einschränkung auf einen fixen Wertebereich ist es möglich, den Mittelwert, Median etc. zu bilden und so eine Vergleichbarkeit zu ermöglichen.

Verwaltung der Metriken

Für die spätere Ausführung der Metriken ist es sinnvoll, eine zentrale Stelle zu haben, welche die Verwaltung der Metriken übernimmt. Diese zentrale Stelle entkoppelt die Verwaltung der Metriken von dem restlichen Programmcode und sorgt so für eine bessere Struktur des Programms. Diese zentrale Komponente ist der **Metrikmanager**. Der Metrikmanager ist eine statische Klasse, die von allen Modulen des Tools benutzt werden kann, welche mit den Metriken arbeiten müssen.

Eine wichtige Funktion des Metrikmanagers ist das Erzeugen neuer Metriken. Zwar wäre es möglich, die Metriken direkt zu instanziiieren, wenn sie benötigt werden, allerdings entsteht dadurch eine direkte Abhängigkeit zwischen der Metrik und dem Modul, welches eine Metrik erzeugen möchte. Zur Vermeidung dieser direkten Abhängigkeit können Fabrikmethode verwendet werden, bei dem ein Objekt nicht direkt erzeugt wird, sondern mittels einer Abfrage durch eine bestimmte Methode erzeugt wird und anschließend an das anfragende Objekt zurückgegeben wird [40, S. 149–161].

Basierend auf dieser Idee kann ein Modul, das ein konkretes Metrikobjekt benötigt, den Metrikmanager mit der Instanziierung beauftragen. Dabei benötigt der Metrikmanager eine Zeichenkette, um eine konkrete abgeleitete Klasse zu identifizieren, von der das neue Metrikobjekt instanziiert werden soll. Diese Zeichenkette wird eindeutig einer bestimmten abgeleiteten Klasse zugeordnet, sodass bei der Implementation einer neuen Metrik ein neuer Name spezifiziert werden muss. Alle Zeichenketten, die für eine bestimmte Metrik stehen, werden in einer Konstantensammlung (*Enum*) definiert, die ebenfalls bei der Definition einer neuen Metrik ergänzt werden muss. Das neue Metrikobjekt benötigt, wie in Kapitel 3.2.1 beschrieben, zudem einen eindeutigen Namen und Parameter, damit es eindeutig auffindbar ist und korrekt arbeiten kann. Dieses neue Metrikobjekt wird zudem vom Metrikmanager registriert und in einer Liste gespeichert, damit es später möglich sein wird, über alle erzeugten Metriken zu iterieren.

Der Metrikmanager bietet zudem eine Methode an, mit denen der Standardwerte für die Parameter einer Metrik abgerufen werden können, sodass diese an einer Stelle verwaltet werden können. Außerdem kann der Metrikmanager auch einen *MetricResultBuilder* erzeugen, um Teilergebnisse zu einem Gesamtergebnis zu aggregieren. Dies geschieht ebenfalls über eine Fabrikmethode, sodass auch hier eine Entkopplung stattfindet.

Sprachspezifische Informationen für Metriken

Da das Tool für möglichst viele objektorientierte Programmiersprachen konzipiert werden soll, muss eine Generalisierung erfolgen, da jede Sprache ihre Eigenheiten hat und möglicherweise besondere Funktionen anbietet, die nur schwer in einem abstrakten Format zu bringen sind.

Nichtsdestotrotz können solche sprachspezifischen Eigenheiten auch in der Dokumentation erwähnt werden. Daher ist es eine sinnvolle Idee, dass Metriken auch diese Besonderheiten benutzen, um ein genaueres Bild der Dokumentationsqualität zu erfahren, ohne jedoch zu wissen, welche Programmiersprache gerade analysiert wird. Beispielsweise können die Checked-Ausnahmen in Java mit den Informationen in der Javadoc verglichen werden. Auch können hierdurch überschriebene Methoden ignoriert werden, da diese oft nicht mehr dokumentiert werden müssen.

Um solche sprachspezifischen Analysen zu erlauben, besitzt jede Metrik Zugriff auf ein Objekt der Klasse *LanguageSpecificHelper*. Wenn eine neue Programmiersprache hinzugefügt werden soll, kann von dieser geerbt werden. In der Klasse *LanguageSpecificHelper* sind bereits einige Methoden definiert, die einigen Metriken bei der Bewertung helfen. So bewertet die Methode *rateDocumentationCompatibility*, ob die Dokumentation alle sprachspezifischen Informationen erläutert (z. B. „@throws“). Die Methode *shallConsider* kann genutzt werden, um überschriebene Methoden zu ignorieren.

Um eine eigene Methoden hinzuzufügen, muss diese in der Basisklasse definiert werden. Diese Methode sollte in der Basisklasse keine Aktionen durchführen, sondern entweder gar nichts tun oder Rückgabewerte haben, die keinen Einfluss auf Metriken haben. Anschließend kann diese Methode in einer sprachspezifischen abgeleiteten Klasse der Basisklasse korrekt implementiert werden. Die entsprechende Methode kann dann durch Änderung des Quellcodes der Metrik an den passenden Stellen von der Metrik verwendet werden. So kann beispielsweise das Resultat einer Metrik modifiziert werden oder weitere Ergebnisse mittels des *MetricResultBuilders* angefügt werden.

3.2.2 Einzelergebnisse verarbeiten

Das berechnete Ergebnis einer Komponente muss nun gespeichert werden, damit es später ausgewertet werden kann. Dazu wird ein *MetricResult*-Objekt erstellt, welches das im vorherigen Unterabschnitt berechnete Ergebnis enthält. Außerdem werden hier eventuelle Begründungen und Hinweise gespeichert, die den Anwender dabei unterstützen, die Qualität der Dokumentation zu verbessern. Jede Begründung enthält den Dateipfad der betroffenen Datei, den Namen der bemängelten Komponente und ein Zeilennummernintervall, sodass der Benutzer die problematische Stelle schnell finden kann. Zuletzt

werden noch Informationen gespeichert, die beschreiben, in welchem Kontext das Ergebnis produziert wurde. Dies ist für die Gewichtung der Einzelergebnisse notwendig und wird in den nächsten Unterabschnitten noch genauer erläutert.

Für die Speicherung des Objektes gibt es zwei Möglichkeiten. Die erste Möglichkeit wäre es, dass die *analyze*-Methode das *MetricResult*-Objekt zurückgibt, sodass der Aufrufer damit arbeiten kann. Bei der zweiten Möglichkeit wird das Ergebnis einem anderen Objekt übergeben, der dann die Weiterverarbeitung vornimmt. Dies hat den Vorteil, dass eine Metrik kein Ergebnis zurückliefern muss, wenn es kein sinnvolles Ergebnis berechnen kann. Bei einem Rückgabewert müsste ansonsten ein ungültiger Wert wie z. B. *null* vereinbart werden. Außerdem kann eine Metrik auch mehrere Resultate speichern, was bei komplexeren Komponenten in Betracht gezogen werden könnte. Dieses weitere Objekt ist ein *MetricResultBuilder*, der wie im nächsten Unterabschnitt beschrieben, die Softwaredokumentationsqualität jeder Komponente sammelt und daraus ein Gesamtergebnis berechnet.

Einzelergebnisse aggregieren

Da ein Softwareprojekt aus Tausenden von Dateien bestehen kann, die wiederum aus verschiedenen Komponenten bestehen, müssen die Einzelergebnisse aggregiert werden, um so am Ende ein Gesamtergebnis zu erhalten, das zur Einschätzung der Qualität der Softwaredokumentation genutzt werden kann. Dazu wird dem *MetricResultBuilder* jedes Ergebnis mittels der *processResult*-Methode mitgeteilt, welches das Ergebnis in einer Liste speichert. Wenn alle Metriken verarbeitet sind, wird daraus ein Gesamtergebnis gebildet. Dies geschieht durch die Methode *getAggregatedResult*. Dabei wird standardmäßig ein arithmetischer Mittelwert gebildet.

Neue Algorithmen (wie z. B. der Median oder der gewichtete Mittelwert) können implementiert werden, indem von dieser Klasse abgeleitet wird und die *getAggregatedResult*-Methode überschrieben wird.

Ein *ResultBuilder* basiert auf dem Vorbild des Design-Patterns „Builder“ aus [40, S. 139–149], da dieser aus einzelnen Metrikresultaten ein vollständiges Metrikergebnis baut.

Zuordnung der Gewichte

Für einige Algorithmen muss eine Gewichtung vorgenommen werden, um bestimmte Ergebnisse besser oder schlechter zu bewerten. Dazu muss jedes Teilergebnis ein Gewicht zugeordnet werden.

Insgesamt kann ein Teilergebnis anhand von drei Kategorien gewichtet werden. Durch die Gewichtung aufgrund der verwendeten Metrik können bestimmten Metriken einen größeren Einfluss auf das Gesamtergebnis haben, wenn diese als vertrauenswürdiger empfunden werden. Durch die Gewichtung von Dateien kann beispielsweise die öffentliche Schnittstelle einen größeren Einfluss auf die Bewertung nehmen, da diese Komponenten bzw. Dateien sehr kritisch sein können und daher gut verstanden werden müssen. Auch eine Gewichtung nach Komponenten kann in bestimmten Situationen sinnvoll sein. So kann beispielsweise argumentiert werden, dass Methoden, die durch ihre Parameter,

Rückgabewerte und geworfenen Ausnahmen komplexer als Felder sind, höher gewichtet werden sollen.

Die Zuordnung der Gewichte erfolgt über einen *WeightResolver*, welches eine Schnittstelle anbietet, um einen Bezeichner auf ein Gewicht abzubilden. Bei dem eindeutigen Namen eines Metrikobjektes kann hierfür eine assoziative Liste verwendet werden. Auch bei Komponenten, die durch ihren Klassennamen (wie z. B. *ClassComponent*) repräsentiert werden, ist eine solche assoziative Liste sinnvoll, da nur eine begrenzte Anzahl an Metriken bzw. Komponenten existieren kann. Für Dateipfade ist dies allerdings nicht praktikabel, da es eine Vielzahl an Dateien geben kann. Stattdessen können hier ähnlich wie bei der Filterung von Dateien in Kapitel 4.2 Wildcard-Patterns verwendet werden. Eine assoziative Liste kann dazu jedes Wildcard-Patterns und das dazugehörige Gewicht speichern. Bei einer Abfrage kann das Gewicht des ersten Eintrages zurückgegeben werden, bei dem der Dateipfad mit dem Wildcard-Pattern kompatibel ist. Dies ermöglicht es, ganze Verzeichnisse oder Dateien mit bestimmten Namen stärker zu gewichten.

Bei der Suche nach dem passenden Gewicht zu den Dateien und Komponenten kann es vorkommen, dass das passende Gewicht nicht gefunden wird. Um hierdurch entstehende Probleme zu vermeiden, wird ein Standardwert genommen (z. B. 1).

Jedes *MetricResult*-Objekt enthält ein Tupel von drei Namen, welche die Quelle dieses Ergebnisses repräsentieren (Dateipfad, Komponententyp und eindeutiger Metrikname). Der Dateipfad beschreibt, in welcher Datei die Komponente liegt, die durch dieses Teilergebnis analysiert wurde. Der Komponententyp enthält den Klassennamen der Komponente und gibt somit Aufschluss darüber, ob diese Komponente eine Klasse, Methode etc. ist. Bei einer Klasse würde beispielsweise die Zeichenkette „ClassComponent“ gespeichert werden. Durch den eindeutigen Metrikname (siehe Kapitel 3.2.1) wird festgelegt, welche Metrik dieses Teilergebnis produzierte. Alternativ könnte auch hier der Klassename der Metrik verwendet werden. Allerdings kann eine Metrik durch Verwendung verschiedener Parameter auch mehrfach angewendet werden, sodass eine unterschiedliche Gewichtung möglich wäre. Daher ist der eindeutige Name hier sinnvoller.

Durch dieses Tupel kann für jedes Teilergebnis die Gewichtung der Metrik, der Komponente und des Dateipfades abgerufen werden. Durch Multiplikation aller Gewichtungen entsteht ein Gesamtgewicht.

Abbildung 3.2 veranschaulicht anhand eines vereinfachten Objektdiagramms die Bildung eines Gesamtergebnisses mittels Gewichtung von verschiedenen Ergebnissen, indem der gewichtete Mittelwert aus Kapitel 4.9.1 verwendet wird. Hier wird ein hypothetisches Projekt mit zwei Dateien analysiert. Die erste Datei „Program.java“ enthält nur eine öffentliche Methode *method*, was in Java eigentlich nicht möglich ist, aber hier zur Vereinfachung zugelassen sein soll. Die zweite Datei „Main.java“ enthält eine nichtöffentliche Klasse *Main*. In diesem Beispiel werden zwei Metriken verwendet. Die erste Metrik prüft das Vorhandensein der Dokumentation bei allen Komponenten, während die zweite Metrik nur öffentliche Komponenten betrachtet (siehe Kapitel 4.8.1). In diesem Beispiel soll die Datei „Main.java“ mit dem Faktor 3 gewichtet werden. Methoden sollen mit dem Faktor 4 gewichtet werden. Die zweite Metrik *public_members* wird mit dem Faktor 2 gewichtet. Alle anderen Dateien, Komponenten und Metriken werden mit dem Standardfaktor 1 gewichtet.

3 Konzeption

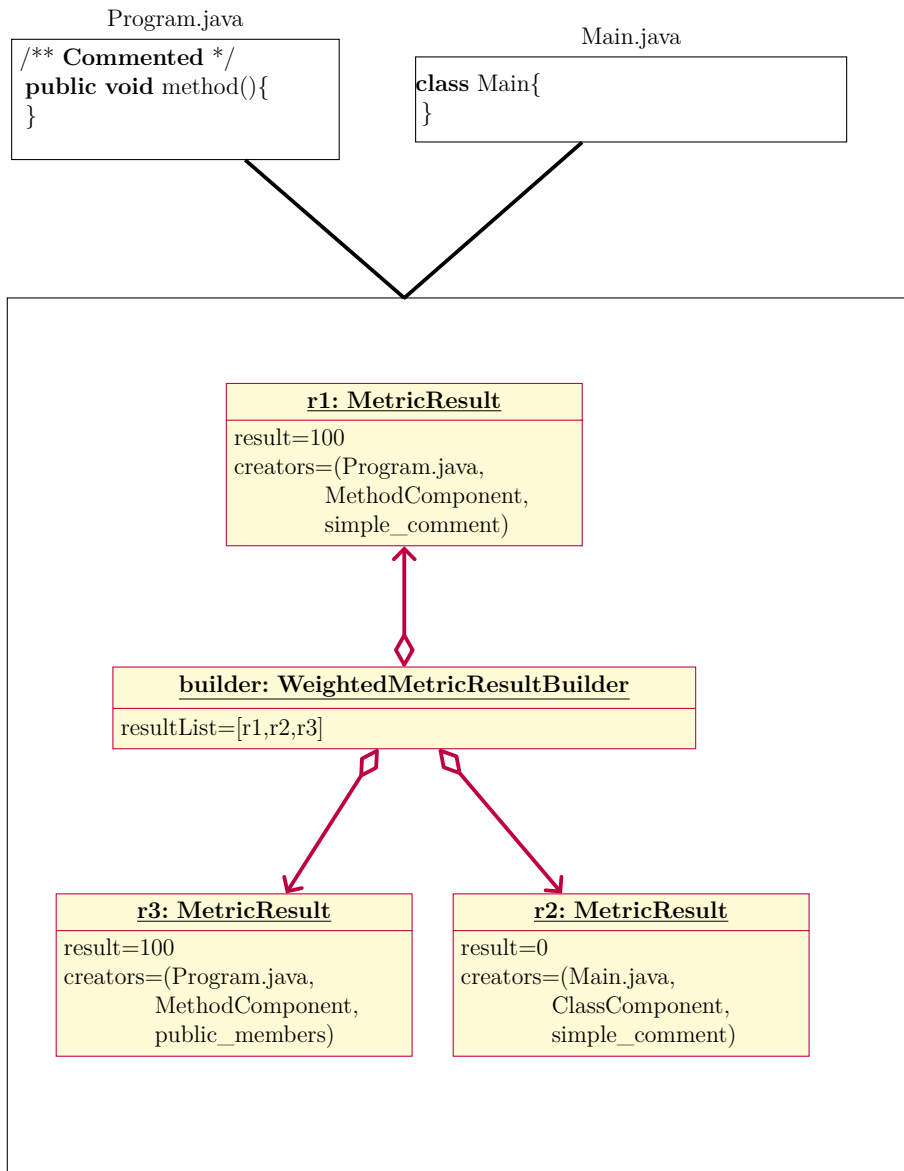


Abbildung 3.2: Veranschaulichung der Bildung eines Gesamtergebnisses

Das erste Teilergebnis *r1* ist 100, da es die Methode *method* beschreibt, welche dokumentiert ist. Auch das dritte Teilergebnis *r3* hat den Wert 100, da es die gleiche Komponente nur mit der zweiten Metrik beschreibt. Das zweite Teilergebnis *r2* hat den Wert 0, da es die undokumentierte Klasse *Main* beschreibt. Es gibt kein weiteres Teilergebnis von „Main.java“, da die zweite Metrik jegliche nichtöffentliche Komponenten ignoriert. Die Teilergebnisse werden durch den *WeightedMetricResultBuilder* intern gespeichert.

Das Teilergebnis jeder gefundenen Komponente enthält gemäß dem obigen Vorgehen ein Tupel aus Dateiname, Komponentennamen und Metriknamen. Beispielsweise enthält das erste Teilergebnis *r1* das Tupel „(Program.java, MethodComponent, simple_com-

ment)“, wobei die einzelnen Bestandteile des Tupels Zeichenketten sind. Basierend auf diesen Informationen kann der *WeightedResultBuilder* eine Gewichtung vornehmen. Das erste Teilergebnis würde mit dem Faktor $1 * 4 * 1 = 4$ gewichtet werden, da es von einer Methode stammt und ansonsten nicht besonders gewichtet wird. Das zweite Teilergebnis erhält das Gewicht $3 * 1 * 1 = 3$, da es von der Datei „Main.java“ stammt. Zuletzt besitzt das dritte Teilresultat die Gewichtung $1 * 4 * 2 = 2$, da es von der zweiten Metrik berechnet wurde und auch von einer Methode stammt.

Basierend auf diesen Teilresultaten kann ein Gesamtergebnis berechnet werden. Hierzu wird jedes Ergebnis mit dessen Gewicht multipliziert und anschließend wird durch die Summe der Gewichte geteilt. Bezogen auf das Beispiel würde die Rechnung folgendermaßen aussehen:

$$\frac{4 * 100 + 3 * 0 + 8 * 100}{4 + 3 + 8} = 80 \quad (3.1)$$

Dieses Ergebnis ist dann Maß für die Bewertung der Dokumentationsqualität dieses hypothetischen Projektes.

4 Umsetzung

In diesem Kapitel wird auf die Umsetzung der in Kapitel 3 beschriebenen Architektur eingegangen. In Kapitel 4.1 wird zunächst erläutert, wie das Programm konkret ausgeführt wird, um die Dokumentationsqualität zu ermitteln. Danach wird in Kapitel 4.2 beschrieben, wie das Programm die einzelnen Java-Dateien findet, damit diese weiterverarbeitet werden können. Außerdem wird in Kapitel 4.3 erläutert, wie ANTLR4 verwendet wird, um Java-Dateien zu parsen. Daraufhin wird erklärt, wie die strukturierten Kommentare geparkt werden (Kapitel 4.4). In Kapitel 4.5 wird die Konfiguration des Tools erläutert. Um auch einen Vergleich zwischen den aktuellen und den vorherigen Zustand der Dokumentationsqualität zu ermöglichen, wird in Kapitel 4.6 beschrieben, wie das letzte Ergebnis der Bewertung gespeichert werden kann. In Kapitel 4.7 wird erklärt, wie das Programm in GitHub Actions eingebunden wird und wie es so genutzt werden kann. Zum Abschluss werden in Kapitel 4.8 und 4.9 die implementierten Metriken und die implementierten Aggregationsalgorithmen erläutert.

4.1 Ausführung des Programms

In diesem Unterabschnitt wird beschrieben, wie das Programm die in Kapitel 3 beschriebenen Arbeitspakete nutzt, um die Qualität der Softwaredokumentation zu bewerten.

Die Koordination des Programms wird in der Datei „index.ts“ durchgeführt, die als Einstiegspunkt des Programms verstanden werden kann. In dieser Datei werden die einzelnen Module des Programms in der richtigen Reihenfolge aufgerufen und die Ergebnisse eines Moduls werden durch die „index.ts“-Datei an das folgende Modul/Arbeitspaket

übergeben, soweit sie dort benötigt werden. Dadurch sind die Module voneinander entkoppelt und greifen nicht direkt aufeinander zu.

Im ersten Schritt muss die Konfiguration des Programms geladen werden. Dazu wird das Arbeitsverzeichnis von der Kommandozeile gelesen. Basierend auf das Arbeitsverzeichnis kann dann die Konfiguration des Tools geladen werden, wie es in Kapitel 4.5 beschrieben ist.

Anschließend müssen einige Objekte initialisiert werden. Hierzu werden die Werte aus der Konfiguration (z. B. der Konfigurationsdatei) verwendet. Beispielsweise kann durch *builder* der Algorithmus festgelegt werden, der die Einzelergebnisse der einzelnen Metriken zu einem Gesamtergebnis kombiniert. Dazu wird eine Fabrikmethode verwendet, da damit die Konstruktion eines Objektes aus einer Zeichenkette möglich ist und somit der Anwender in der Konfiguration nur eine bestimmte Zeichenkette oder ID zur Konstruktion eines komplexeren Objektes angeben muss [40, S. 149–161]. Zudem werden die Metriken, die zur Analyse verwendet werden sollen, durch den Metrikmanager registriert.

Außerdem wird eine assoziative Liste für die Dateien und die Metriken erstellt, die einem eindeutigen Metriknamen bzw. einem Wildcard-Pattern einer Datei ein Gewicht zuordnet. Diese Liste kann einem *WeightResolver* übergeben werden, der wiederum dem *MetricResultBuilder* übergeben wird. Falls dieser keine Gewichtung benötigt, werden diese Informationen ignoriert.

Danach müssen die relevanten Dateien gefunden werden. Dazu werden dem Traversierer (siehe Kapitel 4.2) die Wildcard-Patterns der zu inkludierenden Dateien und der auszuschließenden Dateien übergeben. Mit der Methode *getRelevantFiles* werden dann alle relevanten Dateien zurückgegeben.

In nächsten Schritt muss jede Datei mit jeder Metrik geprüft werden und die Ergebnisse gesammelt werden. Hierzu wird eine verschachtelte For-Schleife verwendet. Dabei gibt es zwei Möglichkeiten zur Verschachtelung. Im ersten Fall könnte in der äußeren Schleife jede Datei und in der inneren jede Metrik durchlaufen werden. Alternativ könnte auch die innere und äußere Schleife vertauscht werden. Der erste Ansatz hat den Vorteil, dass jede Datei nur einmal geladen werden muss, was einen Geschwindigkeitsvorteil bringen kann, deshalb wird dieses Verfahren auch gewählt. Pro Iteration der inneren Schleife wird die aktuelle Datei von der jeweiligen Metrik analysiert und alle gefundenen Metrikresultate, die von den einzelnen Komponenten der Datei stammen, zu einem *MetricResultBuilder* hinzugefügt.

Nach Abschluss der beiden Schleifen steht das Ergebnis durch Aggregation der Resultate in dem *MetricResultBuilder* zur Verfügung und kann genutzt werden, um die Qualität der Dokumentation mit dem Grenzwert bzw. dem letzten Wert zu vergleichen. Wird bei diesem Vergleich festgestellt, dass die Dokumentationsqualität nicht ausreichend ist, wird eine Ausnahme geworfen und das Programm bricht ab. Wird das Programm mittels GitHub Actions ausgeführt, so kann durch diese Ausnahme ein Merge verhindert werden.

Die Abbildungen F.1 und F.2 im Anhang F visualisieren die Ausgabe des Programmes bei einer ausreichenden bzw. mangelhaften Dokumentationsqualität.

4.2 Traversierung aller relevanten Dateien und der Komponenten

Softwareprojekte bestehen aus Hunderten von Dateien, die nicht alle Quellcode enthalten. Beispielsweise gehören Konfigurationsdateien, Ressourcendateien wie Bilder oder binäre Dateien zu den Dateien, bei denen eine Analyse der Softwaredokumentation im Hinblick auf die begrenzte Zeit für die Bachelorarbeit nicht implementierbar ist. Daher ist es sinnvoll, bestimmte Dateien bei der Analyse auszuschließen beziehungsweise nur bestimmte Dateien zu betrachten. Bei einer Weiterentwicklung des Tools nach Abschluss der Bachelorarbeit kann das Tool auf andere Dateitypen ausgeweitet werden, um so ein besseres Gesamtbild über die Softwaredokumentation zu erhalten.

Um die relevanten Dateien zu finden, wird zunächst ein übergeordnetes Verzeichnis benötigt, was bei Softwareprojekten aber der Standard sein sollte. Dieses Verzeichnis kann dann rekursiv durchlaufen werden und somit die Liste aller darin gespeicherten Dateien abgerufen werden. Die relevanten Dateien können dann durch Überprüfung ihres Dateinamens mittels bestimmter Regeln ermittelt werden, die der Benutzer des Tools festlegen kann.

Beim DocEvaluator wird hierzu die NPM-Bibliothek *Minimatch* [41] verwendet, die es ermöglicht, Dateinamen mit Wildcard-Patterns zu vergleichen. Zum Beispiel könnte der Dateiname „test.txt“ mit der Wildcard „test.*“ verglichen werden und die Bibliothek würde eine Übereinstimmung melden.

Auch die Komponenten einer Datei müssen traversiert werden, damit bei jeder Komponente die Dokumentation überprüft werden kann. Da die Komponenten wie in Kapitel 3.1 beschrieben rekursiv aufgebaut sind, kann dies mittels einer Tiefensuche durchgeführt werden.

Ignorieren bestimmter Kommentare

Unter Umständen kann es sinnvoll sein, bestimmte Komponenten bei der Bewertung auszulassen, weil sie beispielsweise noch nicht vollständig implementiert sind, in einer nicht-englischen Sprache dokumentiert sind oder ein anderer gewichtiger Grund existiert. Für diesen Fall kann die allgemeine Beschreibung der Dokumentation einer Komponente den Begriff „%ignore_this%“ oder „%ignore_node%“ enthalten. Bei Ersterem wird nur diese Komponente ignoriert und als nicht existent betrachtet. Bei Zweiterem werden sowohl diese Komponente als auch alle Kinder dieser Komponente ignoriert, sofern sie existieren.

4.3 Implementierung von ANTLR4 für Java

Für die Programmiersprache Java steht eine ANTLR4-Grammatik, die auf GitHub unter der BSD-Lizenz angeboten wird, zur Verfügung [33], allerdings ignoriert diese Grammatik alle Kommentare. Daher müssen einige Änderungen sowohl am Lexer als auch am Parser vorgenommen werden. Im Lexer werden standardmäßig alle Tokens aus einem

Kommentar in einem versteckten Kanal gespeichert, was dazu führt, dass diese Tokens vom Parser ignoriert werden. Um dieses Problem zu lösen, wird das Verhalten durch Definition eines neuen Tokens so geändert, dass Javadoc-Kommentare auch vom Parser verarbeitet werden können, aber mehrzeilige und einzeilige Kommentare weiterhin ignoriert werden. Einzeilige Kommentare sind hier nicht relevant, da sie kein Javadoc enthalten.

Mehrzeilige Kommentare könnten theoretisch auch berücksichtigt werden, da einige Entwickler diese anstelle von Javadoc benutzen. Allerdings werden solche mehrzeiligen Kommentare von Komponenten nicht von Tools erkannt und haben daher einen geringeren, aber durchaus vorhandenen Nutzen [38, S. 4]. Deshalb werden Komponenten, die zwar mit mehrzeiligen Kommentaren, aber nicht mit Javadoc dokumentiert sind, wie undokumentierte Komponenten betrachtet. Für einen Entwickler sollte es so schnell möglich sein, solche nicht korrekt dokumentierten Komponenten zu identifizieren und deren mehrzeilige Kommentare in gültige Javadoc-Kommentare umzuwandeln und so die Qualität der Dokumentation zu erhöhen. Für andere Programmiersprachen können jedoch normale mehrzeilige wie strukturierte Kommentare betrachtet werden, wenn dies für sinnvoller erachtet wird.

Mehr Änderungen müssen an der entsprechenden Parser-Datei „JavaParser.g4“ durchgeführt werden. Da diese Änderungen für die eigentliche Thematik dieser Bachelorarbeit nur eine untergeordnete Rolle spielen, wird hier nicht jede Änderung genauer erklärt. Tabelle A.1 im Anhang A listet alle Änderungen an der Parserdatei auf. Die geänderte Parserdatei und das Original befinden sich auch im digitalen Anhang im Verzeichnis „parser_changes“.

Um die Informationen aus einer Java-Datei mittels ANTLR4 zu verarbeiten, kann das Visitor-Pattern verwendet werden [40, S. 400ff.]. Mit einem Visitor kann die Baumstruktur, die ANTLR4 erstellt hat, traversiert werden, damit so nur die notwendigen Informationen herausgefiltert werden. Andere Informationen (wie z. B. Conditional-Branches) können so ignoriert werden. Listing 4.1 zeigt einen Ausschnitt vom Visitor für Methoden-

```

1  private visitMethod(ctx:RuleContext) {
2      this.returnType = ctx.getChild(0).text
3      this.methodName = ctx.getChild(1).text;
4      let visitor = new MethodParamsAndThrowVisitor();
5      let paramsThrow = visitor.visit(ctx);
6      this.methodParams = paramsThrow.params;
7      this.thrownException = paramsThrow.thrownException;
8      this.methodName = new MethodBodyTextVisitor().visit(ctx)
9  }
```

Listing 4.1: Codeauschnitt aus Methoden-Visitor

deklarationen aus dem Quellcode des Tools. Hier ist die Baumstruktur leicht sichtbar. Alle Einzelbestandteile einer Methode wie z. B. Bezeichner, Rückgabetyt etc. sind Kindknoten des *RuleContext* und können über die Methode *getChild* abgerufen werden. So werden sowohl der Bezeichner als auch der Rückgabetyt direkt als Text abgerufen. Diese

beiden Bestandteile bestehen wiederum auch aus weiteren Kindknoten, doch eine weitergehende Betrachtung ist nicht nötig, da nur die Bezeichnung als Zeichenkette benötigt wird. Andere Bestandteile wie die Methodenparameter sind jedoch komplexer, deswegen werden sie von separaten Visitors betrachtet.

4.4 Parsen der strukturierten Kommentare

Um die strukturierten Kommentare in das Format nach Kapitel 3.1 zu bringen, wird eine simple Heuristik verwendet. Es werden so viele Zeilen als allgemeine Beschreibung betrachtet, bis eine Zeile auftaucht, die mit einem Tag wie z. B. „@param“ beginnt, der den allgemeinen Teil beendet.

Anschließend werden diese Tags verarbeitet. Benötigt ein Tag einen Parameter, so wird die Zeile in drei Teilen an den Leerzeichen aufgetrennt. Dabei ist der erste Teil der Typ des Tags, der zweite Teil der Parameter und der Rest (mit allen übrigen Leerzeichen) die Beschreibung des Tags. Bei einem Tag ohne Parameter wird die Zeile in zwei Teile getrennt, wobei hier der erste Teil der Typ des Tags und der letzte Teil die Beschreibung ist.

Diese Heuristik sollte die gängigsten Javadoc-Blöcke verarbeiten können. Alternativ könnte auch ANTLR4 Javadoc parsen. Allerdings ist dies aufgrund der Mischung von natürlicher Sprache und der relativen Flexibilität von Javadoc nicht trivial und wird daher nicht implementiert.

4.5 Konfiguration des Tools

Zur Nutzung des Tools werden bestimmte Informationen benötigt, die aus verschiedenen Quellen bezogen werden. Zunächst benötigt das Tool den Pfad, der die Quelldateien enthält, die nach Kapitel 4.2 traversiert werden sollen. Dieser wird als namenloser Parameter über die Kommandozeile übergeben. Er ist optional, da bei dessen Fehlen das aktuelle Arbeitsverzeichnis genommen wird. Die weiteren Informationen werden aus zwei Quellen bezogen. Wenn beide Quellen fehlen, werden Standardwerte genommen. Die erste Quelle ist eine *JavaScript Object Notation* (JSON)-Datei namens „comment_conf.json“, welche die notwendigen Daten für die Arbeit des Programms enthält. Listing 4.2 zeigt eine beispielhafte Konfigurationsdatei im JSON-Format.

In dieser Beispieldatei werden alle Dateien mit der Dateierweiterung „.java“ bei der Traversierung betrachtet (Z. 1). Außerdem werden dabei keine Dateien bei der Traversierung ausgeschlossen (Z. 2). Diese beiden Werte entsprechend dabei ihre Standardwerte. Sie könnten also bei dieser Konfigurationsdatei weggelassen werden und das Programmverhalten würde sich nicht ändern.

Anschließend (Z. 4–11) werden die zu verwendenden Metriken definiert. Jede Metrik besitzt einen *metric_name*, der den Typ der Metrik spezifiziert. In Zeile 6 wäre dies beispielhaft die Metrik „Anteil der dokumentierten Komponenten an allen Komponenten“ (vgl. Kapitel 4.8.1). Anhang E beschreibt alle implementierten Metriken mit ihren

```

1  {
2      "include": [ " *.java " ],
3      "exclude": [ ] ,
4      "metrics": [
5          {
6              "metric_name": " simple_comment " ,
7              "weight": 1 ,
8              "unique_name": "m1" ,
9              "params": { " ignore_getters_setters ": false }
10         } ,
11     ] ,
12     "absolute_threshold": 50 ,
13     "builder": " weighted_mean_builder " ,
14     "relative_threshold": 30
15 }

```

Listing 4.2: Beispielhafte Konfigurationsdatei für das Tool

Namen. Diese Namen werden vom Metrikmanager dazu genutzt, um die passende Klasse zu finden und so ein Metrikobjekt zu erzeugen. Außerdem erhält jede Metrik durch *unique_name* einen eindeutigen Namen (hier z. B. „m1“). Dieser kann auch weggelassen werden. Dann wird der eindeutige Name aus dem Namen der Metrik und einer fortlaufenden Nummerierung erzeugt. Zudem besitzt jede Metrik das Attribut *weight*, welches zur Bestimmung der Relevanz bzw. des Gewichts der Metrik dient und von einem *MetricResultBuilder* zur Bestimmung eines Gesamtergebnisses benutzt werden kann. Ein *MetricResultBuilder*, der keine Gewichtung der Metriken benötigt, wird diese Information ignorieren. Das Gewicht ist ebenfalls optional. Bei dessen Fehlen wird das Gewicht „1“ eingesetzt. Durch *params* werden der Metrik die Parameter übergeben, die sie benötigt. Die genaue Anzahl und Struktur der Parameter hängen von der jeweiligen Metrik ab. Fehlen diese Parameter, so werden standardmäßige Parameter verwendet.

Fehlt der Eintrag „metrics“, so werden alle implementierten Metriken mit ihren Standardwerten genommen.

Als Nächstes (Z. 12) wird der Schwellwert festgelegt. Dieser Wert legt fest, ob das Programm beim Unterschreiten dieses Wertes mit einer Fehlermeldung abbrechen soll. In Zeile 13 wird der *MetricResultBuilder* festgelegt, der bestimmt, wie die Einzelergebnisse aggregiert werden. In dem Beispiel werden alle Teilresultate mittels eines gewichteten Mittelwertes zu einem Gesamtergebnis aggregiert. In Zeile 14 wird durch *relative_threshold* festgelegt, um wie viel sich die Dokumentationsqualität verschlechtern muss, damit ebenfalls eine Fehlermeldung erscheint. Dies wird in Kapitel 4.6 genauer erläutert.

Die zweite Quelle für die Informationen sind die Eingabeparameter aus GitHub Actions. Dazu wird, wie in Kapitel 4.7 beschrieben, jeder Parameter aus der JSON-Datei auch in der „action.yml“-Datei übernommen. Bei der Ausführung des Programms stehen diese Eingabedaten über Umgebungsvariablen bereit. Jede Umgebungsvariable beginnt mit der Zeichenkette „INPUT_“, anschließend folgt der Name des entsprechenden Pa-

rameters (wie in der JSON-Datei), wobei der Name allerdings komplett in Großbuchstaben geschrieben ist. So steht „absolute_threshold“ als „INPUT_ABSOLUTE_THRESHOLD“ zur Verfügung.

Da es durchaus sein kann, dass sowohl eine Konfigurationsdatei existiert als auch die Umgebungsvariablen gesetzt sind, muss klar festgelegt werden, welcher Wert eines Parameters am Ende genommen wird. Bei dem Tool haben die von GitHub Actions erzeugten Umgebungsvariablen Vorrang, da das Tool für die Verwendung in GitHub Actions konzipiert wurde. Die Auflistung im Anhang D listet alle Parameter des Tools nochmals auf und erläutert sie zusätzlich.

4.6 Speicherung des letzten Ergebnisses

Neben der bereits erwähnten Möglichkeit, einen absoluten Grenzwert für die Dokumentationsqualität zu definieren, ist auch ein inkrementeller Vergleich interessant. Dabei wird das Ergebnis der Dokumentationsqualität zwischengespeichert. Bei einem neuen Start des Tools kann das alte Ergebnis mit dem neuen Ergebnis verglichen werden. Verschlechtert sich das Ergebnis über einen gewissen Schwellwert hinaus, so sollte der Entwickler ebenfalls gewarnt werden, selbst wenn die Dokumentationsqualität noch über der absoluten Grenze liegt. Schließlich kann dies ein Trend sein, der zum baldigen Unterschreiten des absoluten Grenzwertes führen kann.

Der Ort zur Speicherung des letzten Wertes ist dabei flexibel. Standardmäßig wird der Wert in einer Datei namens „evaluator_last_state.txt“ gespeichert. Falls das Programm im Kontext von GitHub Actions ausgeführt wird, sollte allerdings beachtet werden, dass diese Datei nach der Beendigung des Workflows gelöscht wird. Dieses Problem kann dadurch gelöst werden, dass die geänderte Datei im Repository des zu analysierenden Projektes hochgeladen wird. Dies kann beispielsweise mit dem Tool *Add & Commit* [42] erledigt werden. Nachteilhaft ist an diesem Vorgehen allerdings, dass hierdurch in dem Commit-Verlauf automatisierte Commits erscheinen, sodass der Überblick verloren gehen kann. Eine weitere Möglichkeit zur Speicherung des Wertes wäre es, den Wert an einen externen Server zu senden und bei einem erneuten Start diesen Wert abzurufen.

4.7 Einbindung in GitHub Actions

Um das Tool in GitHub Actions einzubinden, müssen einige Schritte erfolgen. Zunächst muss eine „action.yaml“ geschrieben werden, die das GitHub-Repository als Aktion markiert und die notwendigen Befehle für die Ausführung enthält. Listing 4.3 zeigt einen beispielhaften Code der Action. Zur Übersichtlichkeit wird in diesem Listing nur ein Eingabeparameter definiert. Die restlichen Eingabeparameter werden im Programm analog definiert.

In den ersten beiden Zeilen werden Attribute wie der Name und eine Beschreibung gesetzt. Danach (Z. 4–7) wird der Eingabeparameter für die minimal erlaubte Bewertung für die Dokumentationsqualität definiert, damit dieser von den Nutzern der Aktion


```

1  name: 'DocEvaluator'
2  description: 'A tool to check whether the documentation is sufficient'
3  author: 'Timo Schoemaker'
4  inputs:
5    absolute_threshold:
6      description: 'The global threshold when to fail'
7      required: false
8  runs:
9    using: 'node16'
10   main: 'build/index.js'

```

Listing 4.3: Beispielhafte Action-Datei für das Tool

verändert werden kann. In den Zeilen 8 bis 10 ist der wichtige Programmcode enthalten, in denen die Aktion als JavaScript-Aktion mit der Node-Version 16 festgelegt wird. Zudem enthält die letzte Zeile auch den Pfad zur Quellcodedatei, mit dem das Programm gestartet werden soll.

Eine JavaScript-Aktion in GitHub Actions benötigt JavaScript, sodass der TypeScript-Code des Tools erst in JavaScript umgewandelt werden muss. Damit das Programm bei der Veröffentlichung einer neuen Version in einen auslieferbaren Zustand gebracht werden kann, wird ein weiterer Workflow benötigt, der bei jedem Push in dem Main-Zweig folgende Schritte ausführt:

1. Klonen des Main-Branch des Repositorys (wie bei den meisten anderen Workflows)
2. Aufruf von TSC, Konvertierung des TypeScript-Codes in JavaScript
3. Aufruf und Benutzung von *NCC* [43]. Packen aller JavaScript-Dateien in einer einzigen Datei
4. Kopieren der generierten Datei, die den gesamten Quellcode enthält, und der „action.yml“, in eine (neue) Branch *action*. Dies wird mittels der Aktion *Branch-Push* [44] durchgeführt

Durch diese Schritte wird eine neue Branch erstellt, die nur die notwendige JavaScript-Datei und die *action.yml* enthält. Dadurch können Nutzer der Aktion diese schneller herunterladen und nutzen. Es wäre auch möglich, kein *NCC* zu verwenden, also alle Javascript-Dateien in die neue Branch zu kopieren, allerdings ist die hier gewählte Methode praktikabler, da dann nur ein Lesezugriff beim Starten des Programms erforderlich ist und so ein Geschwindigkeitsvorteil existiert.

Nutzung der Aktion

Die oben erstellte Aktion kann nun von jedem GitHub-Repository verwendet werden. Dazu kann das folgende Listing 4.4 als zusätzlicher Schritt in einem Workflow eingebunden werden.

```

1  uses: compf/DocEvaluator@action
2      with:
3      absolute_threshold: 20

```

Listing 4.4: Verwendung der Aktion in einem Workflow

Hier wird die aktuelle Version des DocEvaluators aus der Branch *action* heruntergeladen und automatisch ausgeführt. Als Parameter wird beispielsweise ein Grenzwert von 20 übergeben, der jedoch nach Belieben angepasst werden kann. Wenn das entsprechende Ereignis des Workflows eintritt (z. B. ein Push-Ereignis), wird der DocEvaluator mit diesem Parameter aufgerufen und zeigt unter der Registerkarte *Actions* eine Fehlermeldung an, wenn die Dokumentationsqualität den Grenzwert unterschreitet und somit nicht ausreichend ist.

4.8 Implementierte Metriken

In diesem Abschnitt wird ein Überblick über einige Metriken gegeben, die in der finalen Version des Tools implementiert sind. Dabei werden die Hintergründe der Metriken erläutert und Vor- und Nachteile der einzelnen Metriken genannt. Die implementierten Metriken lassen sich – abhängig von den berücksichtigten Aspekten der Dokumentation – grob in drei Kategorien einteilen.

4.8.1 Metriken, welche die Abdeckung überprüfen

Eine grundsätzliche Möglichkeit zur Bewertung der Dokumentationsqualität ist es, die Abdeckung der Dokumentation zu prüfen. Dabei werden aus einer Teilmenge aller Komponenten alle dokumentierten Komponenten mit dem maximalen Wert 100 bewertet und alle undokumentierten Komponenten mit 0 bewertet. Diese Metriken werden in der wissenschaftlichen Literatur auch als *ANYJ* oder *DIR* bezeichnet [38, S. 5].

Bei der Wahl der Teilmenge gibt es zwei naheliegenden Möglichkeiten. Bei der ersten Möglichkeit werden alle Komponenten überprüft, sodass die Teilmenge gleich der Gesamtmenge der Komponenten ist. Für eine gute Bewertung wird somit verlangt, dass möglichst jede Komponente dokumentiert wird. Bei der zweiten Möglichkeit werden nur die öffentlichen Komponenten untersucht und alle nicht öffentlichen Komponenten so behandelt, als ob sie nicht existieren würden. Dies hat den Vorteil, dass nur Komponenten untersucht werden, die wahrscheinlich von anderen Komponenten verwendet werden und somit besser verstanden werden müssen [21, S. 253].

Durch diese Metriken kann geprüft werden, ob ausreichend viele Komponenten dokumentiert sind. Allerdings überprüfen diese Metriken nur, ob die Komponente dokumentiert ist oder nicht. Ein leerer, sinnloser oder sachfremder Kommentar würde dennoch mit 100 Punkten bewertet werden. Falls alle Komponenten untersucht werden, kann außerdem das Problem auftreten, dass einige Komponenten nur in einem kleinen Bereich

des Programms verwendet werden, sodass sie für die Dokumentationsqualität weniger relevant sein können.

Als Erweiterung der Abdeckungsmetriken kann bei Methoden überprüft werden, ob auch ihre Parameter und der Rückgabewert dokumentiert sind. Dies ist sinnvoll, da diese auch Teil der Methode sind und zum Verständnis der Methode beitragen [38, S. 5]. Zudem kann es sinnvoll sein, triviale Getter und Setter, die den Wert eines privaten Feldes auslesen oder verändern, bei der Bewertung auszuschließen. Schließlich haben diese Methoden einen klaren Zweck, der oft keiner weiteren Kommentierung bedarf [21, S. 254].

Bei der Abdeckung der Dokumentation von Methoden könnte auch die Länge der Methode berücksichtigt werden, was aber in der Literatur nicht erwähnt wird. Dabei wird jede Methode mit ihrem *Lines of Code* (LOC) gewichtet und die Summe der LOC der dokumentierten Methoden innerhalb einer Klasse wird durch die Summe der LOC aller Methoden der Klasse geteilt. Dies hat den Vorteil, dass undokumentierte Getter und Setter, die nur wenige Codezeilen haben, das Ergebnis nicht so stark beeinflussen, aber trotzdem berücksichtigt werden.

4.8.2 Metriken, welche die Semantik überprüfen

Da nicht nur das Vorhandensein, sondern auch die Aussagekraft und Verständlichkeit der Dokumentation wichtig sind, bietet das Tool weitere Metriken an. Diese Metriken überprüfen die Semantik, also den Inhalt, der Dokumentation und können so Aufschluss darüber geben, ob die Dokumentation tatsächlich hilfreich ist.

Zur Bewertung der Verständlichkeit eines englischsprachigen Textes ist der **Flesch-Score** sehr bekannt [45, S. 21]. Dies ist eine Formel zur heuristischen Bewertung der Lesbarkeit eines Textes, welches die Anzahl der Sätze, Silben und Wörter berücksichtigt. Die Formel lautet:

$$206,835 - 1,015 * \frac{W}{S} - 84,6 * \frac{H}{W} \quad (4.1)$$

Dabei ist S die Anzahl der Sätze, W die Anzahl der Wörter und H die Anzahl der Silben. Die Formel liefert einen Wert von 0 bis 100 zurück, wobei 0 auf einen sehr komplizierten und 100 auf einen leichten Text hindeutet. Diese Formel und verwandte Formeln werden auch von verschiedenen US-Behörden verwendet, um die Lesbarkeit ihrer Dokumente zu verbessern [3, S. 72]. Die Implementation der Metrik nimmt an, dass ein Flesch-Score von 70 mit 100 Punkten bewertet werden sollte, da mehr als 80 % der US-Bevölkerung solche Texte verstehen können und diese Texte weder zu leicht noch zu schwer sein sollen [45, S. 22]. Für kleinere Flesch-Scores werden weniger Punkte vergeben. Für größere Flesch-Scores werden ebenfalls Punkte abgezogen, aber nicht mehr als 15, da sehr leichte Texte besser sind als schwierige Texte. Bei der Verwendung des Flesch-Scores sollte beachtet werden, dass die Bestimmung der Silbenzahl eines Wortes nicht immer trivial ist. Bei der Entwicklung des Tools musste bei einem Austausch einer Bibliothek die Testmethode angepasst werden, da das Wort „themselves“ plötzlich drei statt zwei Silben hatte. Dies hängt natürlich auch von der Aussprache und somit von

kulturellen Gegebenheiten ab. Dies gilt natürlich auch für die Bestimmung von Wörtern und Sätze.

Der **Gunning-Fog-Index** ist eine ähnliche Formel. Sie berücksichtigt nicht die Anzahl der Silben, sondern die Zahl der komplizierten Wörter. Ein Wort gilt dabei als kompliziert, wenn es mehr als zwei Silben hat [45, S. 24]. Dabei gibt der Gunning-Fog-Index die Anzahl an Schuljahren zurück, die ein Leser absolviert haben muss, um den Text gut verstehen zu können. Die Definition des komplizierten Wortes ist aber umstritten. So ist beispielsweise das Wort „vacation“ drei Silben lang, aber nicht unbedingt kompliziert [46, S. 10].

Eine weitere Möglichkeit zu Bewertung der Semantik der Dokumentation ist ein Vergleich der Dokumentation mit dem Namen der dokumentierten Komponente. Ein Kommentar, der einen Großteil des Namens der Komponente wiederholt, bietet keinen Mehrwert. Auch ein Kommentar, der keinen Zusammenhang mit der dokumentierten Komponente erkennen lässt, verliert an Nutzen [2, S. 86]. Um diese **Kohärenz** zu messen, kann die Anzahl der gemeinsamen Wörter zwischen Dokumentation und des Namens der dokumentierten Komponente ermittelt werden und dies durch die Zahl der Wörter der Dokumentation geteilt werden. Die Autoren in [2, S. 87] vertreten die Ansicht, dass dieser **Kohärenzoeffizient** von mehr als 0,5 oder gleich 0 auf eine schlechte Dokumentation hindeutet, da es im ersten Fall eine starke Ähnlichkeit zwischen dem Namen und der Dokumentation gibt und im zweiten Fall überhaupt keine Gemeinsamkeiten existiert. Die Implementation der Metrik bewertet in beiden Fällen die Dokumentation mit 0 Punkten, in allen anderen Fällen vergibt sie 100 Punkte.

```

1  /**
2   * Calculates the square root
3   * @param number a positive number
4   * @return the positive square root of the number
5   */
6  double calculateSquareRoot(double number){
7      // ...
8  }
9
10 /**
11  * Returns always a positive number
12  * @param number any number
13  * @return a positive number
14  */
15 double absoluteValue(double number){
16     // ...
17 }
```

Listing 4.5: Zwei Methoden mit mangelhafter Kohärenz

Listing 4.5 zeigt zwei Methoden, bei denen die Kohärenz nicht gut ist. Bei der ersten Methode ist der Kohärenzoeffizient 0,75, da alle Wörter der allgemeinen Beschreibung der Komponente (Z. 2) außer „the“ im Komponentennamen auftreten. Zur Verbesserung

der Kohärenz könnte beispielsweise erwähnt werden, wie sich die Methode bei negativen Werten verhält. Bei der zweiten Methode gibt es überhaupt keine Gemeinsamkeiten zwischen den Namen der Komponente und der Dokumentation (Z. 11), sodass der Kohärenzkoeffizient gleich 0 ist. Hier könnte z. B. zur Verbesserung das Wort „absolute“ in der Dokumentation erwähnt werden.

Die Kohärenzmetrik hilft dem Entwickler dabei, dass die Dokumentation zu dem Namen der dokumentierten Komponente passt und gleichzeitig einen Mehrwert bietet, sodass die Dokumentation nützlich ist. Allerdings können Synonyme den Wert verfälschen, da dann keine gemeinsamen Wörter gefunden werden. Auch durch Füllwörter (wie z. B. *the, of* etc.) kann die Bewertung unterschätzt werden.

4.8.3 Metriken, welche nach Fehlern suchen

Die letzte Kategorie an Metriken analysiert die Dokumentation und sucht nach dem Vorkommen von bestimmten Fehlern, welche die Qualität der Dokumentation negativ beeinflussen können. Solche Fehler können beispielsweise Rechtschreibfehler oder fehlerhafte Formatierung (wie z. B. nicht konformes HTML) sein. In der offiziellen Javadoc-Dokumentation wird außerdem geraten, bestimmte Abkürzungen (wie z. B. *e. g., aka.* oder *i. e.*) zu vermeiden [20].

```

1  /**
2   * Assign all field values of the target to the source
3   * @param target the target object, may not be null
4   * @param source the source object
5   */
6  void assign(Object target, Object source){
7
8  }
```

Listing 4.6: Methode mit unvollständiger Erklärung der Behandlung von Nullwerten

Bei Methoden sollte zudem darauf eingegangen werden, ob die Methode *null* als Parameterwert akzeptiert oder *null* zurückgeben kann, damit der Benutzer die Methode korrekt verwenden kann [47]. Listing 4.6 zeigt eine Methode, bei denen die Behandlung von *null* nur teilweise erwähnt wird. Beim ersten Parameter (Z. 3) wird erwähnt, dass der Parameter niemals null sein kann und daher ein Nullwert zu einer Ausnahme führen kann. Beim zweiten Parameter (Z. 4) ist dies nicht klar und der Benutzer der Dokumentation wird im Unklaren gelassen, was zu einer fehlerhaften Verwendung der Methode führen kann.

In allen Fällen sollte ein Verstoß gegen diese Konventionen zu Punkteabzug führen. Dazu können pro Verstoß Fehlerpunkte vergeben werden. Diese Fehlerpunkte können dann der Funktion

$$B(l) = S - (S - B_0) * e^{-k*l} \quad (4.2)$$

übergeben werden, die eine beschränkt fallende Exponentialfunktion ist. Dabei ist S die untere Schranke für die Bewertung, B_0 die bestmögliche Bewertung, k eine Konstante

für die Wachstumsrate und l die Zahl der Fehlerpunkte. Durch diese Funktion erhalten Kommentare mit vielen Fehlerpunkten eine geringere Bewertung als Kommentare mit keinen oder wenigen Fehlerpunkten. Die Wachstumsrate k kann dabei parametrisiert sein und sollte auch von der jeweiligen Fehlerkategorie abhängen. So kann beispielsweise argumentiert werden, dass Rechtschreibfehler weniger gravierend sind als fehlerhaftes HTML.

Bei allen Metriken sollte allerdings beachtet werden, dass diese nur nach relativ einfachen Fehlern suchen. Bei einer Prüfung des HTML-Inhaltes werden beispielsweise nur nicht geschlossene Tags detektiert. Auch falsch-positive Ergebnisse können auftreten, wenn beispielsweise bei der Rechtschreibprüfung ein nicht im Wörterbuch vorkommendes Wort, das aber korrekt geschrieben ist, bemängelt wird.

Als weitere Möglichkeit, die in der Literatur nicht erwähnt wird, könnten lange Methoden, die bestimmte Fehler aufweisen, härter bestraft werden. So kann beispielsweise argumentiert werden, dass lange Methoden, die sowieso vermieden werden sollten [48, S. 34], eher dokumentiert werden sollten, da sie komplexer sind. Daher können längere undokumentierte Methoden schlechter bewertet werden, indem sie mit Fehlerpunkten, welche äquivalent zu ihren LOC sind, bestraft werden.

4.9 Algorithmen zur Bildung eines Gesamtergebnisses

Zur Bildung eines Gesamtergebnisses aus verschiedenen Teilresultaten gibt es verschiedene Möglichkeiten, die in diesen Abschnitt vorgestellt werden. Jeder vorgestellte Algorithmus hat bestimmte Voraussetzungen und kann eine große Auswirkung auf die Interpretation des Gesamtergebnisses haben, sodass die Wahl sorgsam überlegt werden muss.

4.9.1 Klassische Algorithmen

Die klassischen Algorithmen zur Berechnung eines Ergebnisses aus mehreren Ergebnissen sind hinlänglich bekannt. Sie alle basieren auf der Annahme, dass die dahinter liegende Verteilung symmetrisch ist. Im Kontext der besprochenen Metriken bedeutet dies, dass die Wahrscheinlichkeit für eine gute Bewertung genauso sein sollte wie eine schlechte Bewertung. Bei vielen Metriken zur Bewertung der Softwarequalität kann diese Annahme nicht getroffen werden [49, S. 313]. Auch bei den vorgestellten Metriken muss diese Annahme nicht zwangsläufig stimmen, sodass die Gesamtbewertung stets kritisch hinterfragt werden sollte.

Arithmetischer Mittelwert

Der arithmetische Mittelwert wird durch die Klasse *MetricResultBuilder* implementiert. Dieser Algorithmus berücksichtigt jedes Ergebnis gleichermaßen. Dies ist insbesondere dann sinnvoll, wenn keine guten Kriterien für die Gewichtung von Metriken, Dateien oder Komponenten gefunden werden können. Es sollte allerdings auch beachtet werden, dass

der Mittelwert extreme Ausreißer berücksichtigt. Dies kann hier sinnvoll sein, da eine in Teilen sehr schlechte Dokumentation besser berücksichtigt wird und so ein verlässliches Gesamtergebnis geliefert wird.

Median

Der Median der Einzelresultate wird von der Klasse *MedianResultBuilder* berechnet. Dabei werden die Einzelresultate nach dem bekannten Median-Algorithmus verarbeitet. Bei einer geraden Anzahl an Elementen wird der Median aus dem Mittelwert der zwei infrage kommenden Ergebnisse gebildet.

Der Median berücksichtigt einzelne Ausreißer nicht und kann daher interessant sein, wenn ein allgemeines Bild von der Dokumentationsqualität erhalten werden soll. Es sollte aber beachtet werden, dass die Anwendung des Medians ein Sortiervorgang benötigt, der in den meisten Fällen eine Komplexität von $O(n * \log(n))$ hat. Zudem kann sich der Median stark ändern, wenn einzelne Komponenten hinzugefügt oder entfernt werden.

Gewichteter Mittelwert

Der gewichtete Mittelwert ist in der Klasse *WeightedMetricResultBuilder* implementiert. Die Zuweisung der Gewichte erfolgt, wie in Kapitel 3.2.2 beschrieben, durch einen *WeightResolver*. Die Gewichte müssen nicht normiert werden, da dies während der Berechnung implizit erledigt wird. Die Resultate jeder Metrik werden multipliziert mit dessen Gewicht, dann aufsummiert und zuletzt durch die Summe aller Gewichte geteilt.

Dieser Algorithmus ermöglicht es zum Beispiel, bestimmte Metriken zu bevorzugen bzw. zu benachteiligen. Dies ist sinnvoll, da nicht jede Metrik immer ein aussagekräftiges Ergebnis liefert und bestimmte Metriken je nach Situation ein besseres Bild über die Dokumentationsqualität liefern. Allerdings ist auch zu beachten, dass die Wahl der Gewichte nicht trivial ist und ein Vergleich von Ergebnissen, die verschiedene Gewichte verwenden, nicht sinnvoll ist.

Gewichteter Median

Der gewichtete Median wurde leicht abweichend nach [50, S. 37] implementiert. Dabei wird zunächst die Summe der Gewichte berechnet und die Resultate nach ihrem Gewicht sortiert. Anschließend werden die sortierten Resultate und ihre Gewichte so lange aufsummiert, bis diese temporäre Summe die Hälfte der Gesamtsumme überschreitet. Das Metrikergebnis, bei der diese Bedingung zutrifft, ist das gesuchte Gesamtergebnis. Die Vor- und Nachteile dieses Algorithmus entsprechen den Vor- und Nachteilen des gewichteten Mittelwerts und des Medians. So muss auch hier eine Sortierung durchgeführt werden und die Wahl der richtigen Gewichte ist nicht trivial.

4.9.2 Algorithmen aus der Ökonomie

Als Alternative zu den klassischen Algorithmen wurden Verfahren aus den Wirtschaftswissenschaften vorgestellt, die eine bessere Aggregation der Einzelergebnisse ermöglicht.

chen sollen. Diese Verfahren werden in der Ökonomie benutzt, um die Ungleichheit von Einkommen mathematisch darzustellen. So besitzen in den meisten Staaten viele Menschen nur wenig Einkommen und nur einige Menschen ein sehr hohes Einkommen. Auch auf Softwareprojekten lässt sich diese Erfahrung übertragen, da es beispielsweise nur wenige Methoden mit sehr vielen Codezeilen geben wird, aber dafür sehr viele Methoden mit einer relativ kleinen Anzahl an Zeilen. Beispiele für diese Algorithmen sind Gini, Theil oder Atkinson. Einige implementierte Metriken basierend auf die Anzahl der LOC, sodass eine Anwendung der klassischen Algorithmen problematisch sein kann, wenn die Verteilung sehr asymmetrisch ist. [49, S. 314]

Allerdings beschreiben diese Algorithmen nur, ob die Ergebnisse der Metriken ungleich verteilt sind und nicht, wie gut die Ergebnisse sind. Ein Projekt, bei dem alle Metriken schlechte Resultate liefern, würde dann ein ähnliches Gesamtergebnis liefern als ein Projekt, welches überall gut bewertet wird [11, S. 1121]. Im Kontext der Bachelorarbeit könnte dies bedeuten, dass eine nicht vorhandene Dokumentation genauso gut bewertet wird wie eine exzessive Kommentierung aller Komponenten.

Daher werden diese Algorithmen vom Tool in der ausgelieferten Fassung nicht implementiert. Sie können aber in Verbindung mit anderen Aggregationsalgorithmen interessant sein, um mehr über eventuelle Ungleichheiten bei der Dokumentation zu erfahren.

4.9.3 Squale

Eine andere Möglichkeit zur Aggregation von Metrikergebnissen wird in [11, S. 1124ff.] vorgestellt. Das sogenannte Squale-Modell berechnet aus verschiedenen Teilergebnissen von Metriken ein Gesamtergebnis und kann dabei besonders schlechte Ergebnisse stärker gewichten, damit eventuelle Probleme schnell erkannt werden können.

Im ersten Schritt werden die Ergebnisse der einzelnen Metriken, die auf unterschiedliche Skalen basieren, auf einer Skala normalisiert. Die Autoren nutzen dafür das Intervall 0 bis 3, da so eine Einteilung in Ziel nicht erfüllt (0 bis 1), Ziel größtenteils erfüllt (1 bis 2) und Ziel erfüllt (2 bis 3) möglich ist. Diese normalisierten Ergebnisse werden von den Autoren als „Individual marks (IM)“ (z. dt. individuelle Bewertungen) bezeichnet [51, S. 142].

Im nächsten Schritt werden die einzelnen Ergebnisse aggregiert. Dazu wird jeder IM der Funktion

$$g(\text{IM}) = \lambda^{-\text{IM}} \quad (4.3)$$

übergeben. Dabei ist λ eine Konstante, welche bei der Gewichtung der Ergebnisse hilft. Sollen schlechte Ergebnisse sehr hart bestraft werden, sodass bereits einige Fehler zu einer schlechten Bewertung führen, so muss λ eine große Zahl sein. Durch die Wahl einer kleineren Zahl kann die Bewertung toleranter ausfallen. Aus den Ergebnissen der Funktion $g(\text{IM})$ wird anschließend der Mittelwert W_{avg} gebildet.

Im Anschluss daran muss der Mittelwert wieder in den ursprünglichen Wertebereich abgebildet werden. Dazu wird die Umkehrfunktion

$$g^{-1}(W_{\text{avg}}) = -\log_{\lambda}(W_{\text{avg}}) \quad (4.4)$$

angewendet. Dieses Ergebnis wird von den Autoren als „global mark (GM)“ (z. dt. globale Bewertung) bezeichnet und repräsentiert das endgültige Ergebnis.

Durch diese Aggregation werden Komponenten mit schlechten Ergebnissen stärker gewichtet und haben so einen größeren Einfluss auf das Gesamtergebnis. Allerdings kann dies bei der Dokumentation dazu führen, dass nur wenige schlecht dokumentierte Komponenten die Bewertung stark nach unten ziehen, sodass das Erreichen einer guten Bewertung einen hohen Aufwand erfordert.

Im Tool wird dieser Algorithmus ebenfalls unterstützt. Die Konstante λ kann über den Konfigurationsparameter *builder_params* festgelegt werden. Standardmäßig ist er 9, was für eine mittlere Bestrafung von schlechter Dokumentationsqualität führt [11, S. 1127]. Intern werden die Teilergebnisse in das Intervall 0 bis 3 konvertiert, da ein Intervall von 0 bis 100 zu einer hohen negativen Potenz führen kann und dementsprechend mit Rundungsungenauigkeiten behaftet ist. Am Ende wird das Ergebnis wieder in das originale Intervall von 0 bis 100 gebracht.

5 Evaluation

In diesem Kapitel soll das Programm evaluiert werden, um zu prüfen, ob der *DocEvaluator* eine gute heuristische Aussage über den Stand der Dokumentationsqualität treffen kann und in der Praxis auch einsetzbar ist. Dazu wird der *DocEvaluator* mit *Checkstyle* und *PMD* verglichen, indem drei exemplarische Open-Source-Projekte mit allen drei Programmen analysiert werden und die Treffergenauigkeit und Geschwindigkeit der Programme verglichen werden. Zunächst müssen diese drei Projekte ausgewählt werden, was in Kapitel 5.1 erläutert wird. Anschließend wird in Kapitel 5.2 beschrieben, wie die Evaluation der Treffergenauigkeit durchgeführt wird. In Kapitel 5.3 wird die Geschwindigkeitsevaluation durchgeführt. Zum Abschluss der Evaluation werden die Ergebnisse der Evaluation in Kapitel 5.4 resümiert.

5.1 Wahl der zu analysierenden Projekte

Zur Durchführung der Evaluation wurden verschiedene Softwareprojekte aus GitHub heruntergeladen. Grundsätzlich kann der Vergleich mit jedem Java-Projekt durchgeführt werden, dennoch wurden einige Bedingungen festgelegt, die bei der Auswahl der Projekte eine wichtige Rolle spielen. Diese Bedingungen werden in der folgenden Auflistung präsentiert:

1. Die Projekte müssen mindestens einen Umfang von 10 000 LOC haben
2. Die Projekte müssen bereits in einer in dieser Bachelorarbeit zitierten Quelle in puncto Dokumentationsqualität analysiert worden sein
3. Die Projekte sollen möglichst wenige Parsing-Fehler beim *DocEvaluator* produzieren. Bei mehr als zwei Fehlern wird ein Projekt nicht betrachtet. Können jedoch Fehler folgenlos behoben werden, so kann das Projekt dennoch betrachtet werden
4. Das größte Projekt sollte mindestens zehnmal so groß sein wie das kleinste Projekt

Die LOC sollen nur als sehr grobe Heuristik der Größe eines Softwareprojektes verstanden werden und enthalten nur den reinen Code ohne Kommentare. Dabei wird heuristisch vermutet, dass mit steigender LOC die Anzahl der Komponenten in einem Softwareprojekt steigt, wobei all diese Komponenten fehlerhaft (oder gar nicht) dokumentiert sein können. Somit dienen die LOC als approximative Schätzung der zu erwartenden Fehler.

Durch die Bedingung in Nr. 1 wird sichergestellt, dass eine ausreichende Anzahl an Fehlern in der Dokumentation zu erwarten ist, um eine gute Analyse der Dokumentationsqualität und eine aussagekräftige Bewertung der Geschwindigkeit zu ermöglichen. Durch die Bedingung in Nr. 2 werden nur Projekte in Betracht gezogen, die bereits in der wissenschaftlichen Literatur berücksichtigt wurden und daher (zumindest für den jeweiligen Autor der Quelle) geeignet für eine Analyse der Dokumentation sind. Mit der Bedingung in Nr. 3 wird eine Verzerrung zugunsten oder zuungunsten des *DocEvaluators* vermieden, denn durch Parsing-Fehler können Komponenten falsch analysiert werden, die von den anderen Tools richtig analysiert werden. Indirekt wird dadurch gefordert, dass ein Projekt keine neueren Java-Funktionen verwendet, da der *DocEvaluator* nur Java bis Version 8 unterstützt. Die letzte Bedingung ist besonders für die Bewertung der Geschwindigkeit relevant, da so geprüft werden kann, ob der *DocEvaluator* auch bei größeren Projekten noch in annehmbarer Zeit ein Ergebnis berechnen kann.

Analysierte Projekte

Die folgende Auflistung zeigt die gewählten Projekte für die Evaluation. Die Quellen verweisen auf das Verzeichnis in GitHub, das von den drei Tools analysiert werden soll. In runden Klammern dahinter befinden sich die (auf Zehntausenderstelle gerundeten) LOC.

- Log4J Version 1 [52] (20 000)

- ArgoUML [53] (17 000)
- Eclipse *Java Development Tools* (JDT) [54] (400 000)

ArgoUML und Eclipse wurden in [3, S. 74] bewertet, wobei hier nur Eclipse JDT betrachtet werden soll, da das gesamte Eclipse-Projekt zu umfassend ist. Log4J wurde in [39, S. 267] betrachtet.

Bei allen Projekten traten zunächst Parsing-Fehler auf. Dies lag allerdings in den meisten Fällen daran, dass auskommentierte Methoden noch ein Javadoc-Präfix hatten. Dies kann vom *DocEvaluator* nicht korrekt verarbeitet werden. Da diese Methoden offensichtlich nicht verwendet werden und keinen Einfluss auf die Qualität der Dokumentation haben können, wurden sie ersatzlos entfernt. Bei einem Fehler in Eclipse JDT konnte der *DocEvaluator* eine Datei mit mehr als 3000 Codezeilen nicht verarbeiten. Diese Datei wurde bei der Evaluation von allen Programmen ignoriert.

5.2 Analyse der Qualität

Durch die Evaluation der Qualität soll geprüft werden, ob der *DocEvaluator* trotz des in Kapitel 3 beschriebenen abstrakten Formates eine Java-Datei richtig parsen kann und alle für die Dokumentation relevanten Informationen korrekt extrahieren kann. Zur Durchführung der Evaluation muss zunächst definiert werden, welche Funktionen der einzelnen Programme miteinander verglichen werden können, da die Programme unterschiedliche Aspekte der Dokumentation überprüfen und die Darstellung der Ergebnisse im Vergleich zum *DocEvaluator* abweicht.

Checkstyle und *PMD* können als fehlersuchend bezeichnet werden. Sie prüfen die einzelnen Komponenten eines Programms und finden Abweichungen von vorher definierten Regeln. Eine solche Regel kann beispielsweise sein, dass jede öffentliche Methode dokumentiert sein muss, dass bestimmte Wörter nicht verwendet werden dürfen oder dass die Syntax der Dokumentation gültig sein muss. Damit sind sie vergleichbar mit den Metriken aus den Kapiteln 4.8.1 und 4.8.3, welche ebenfalls bestimmte Fehler suchen und bei einem Verstoß gegen die Regeln eine Warnmeldung ausgeben. Allerdings berechnen *Checkstyle* und *PMD* keine Metriken, sondern finden nur die besagten Verstöße gegen die definierten Regeln. Somit kann ein Entwickler sehen, dass ein Projekt beispielsweise 100 Verstöße gegen die Dokumentationsrichtlinien hat, erfährt aber nicht, ob die Anzahl der Verstöße unter Berücksichtigung der Projektgröße schwerwiegend ist und erhält keine normierte Bewertung, die dem Entwickler bei der Beurteilung der Dokumentationsqualität hilft.

Im Gegensatz dazu verwendet der *DocEvaluator* Metriken, die stets einen Wert von 0 bis 100 zurückgeben, sodass ein Entwickler weiß, dass ein hoher Wert für eine hohe Qualität steht. Außerdem kann der *DocEvaluator* auch die Semantik des Kommentars heuristisch prüfen, um zu erfahren, ob der Kommentar verständlich ist und nicht redundant ist (vgl. Kapitel 4.8.2). Nichtsdestotrotz gibt der *DocEvaluator* auch Warnmeldungen aus, wenn er bestimmte Komponenten schlechter bewerten muss.

Aus diesen Gründen wird die Evaluation nicht mit den Metriken an sich, sondern mit den Warnmeldungen der jeweiligen Tools durchgeführt. Jedes Programm wird mit einem Verweis auf das zu analysierende Projekt aufgerufen und die Ausgaben der Programme werden in separaten Dateien umgeleitet. Diese Dateien dienen dann als Grundlage für die spätere Evaluation.

Durchführung der Evaluation

Wie im vorherigen Absatz beschrieben, dienen die Logdateien der drei Programme als Datengrundlage für die Evaluation. Jede Zeile in diesen Logdateien enthält mindestens den Dateipfad des gefundenen Fehlers, die Zeilennummer des Fehlers und einen Fehlercode als Zeichenkette. Wenn die drei Programme einen übereinstimmenden Fehler finden, sollte es in allen drei Logdateien einen Eintrag geben, bei dem der Dateipfad identisch ist, die Zeilennummer innerhalb eines gewissen Intervalls identisch ist und die der Fehlercode identisch ist. Die Zeilennummer muss nicht zwingend identisch sein, da *Checkstyle* und *PMD* die exakte Zeile des Fehlers ausgeben, während der *DocEvaluator* nur ein Intervall ausgibt, in dem der Fehler liegt. Beispielsweise wird von *Checkstyle* und *PMD* bei einem unzulässigen Wort die exakte Zeile des Wortes genannt. Da die drei Tools unterschiedliche Fehlercodes ausgeben, werden diese so kategorisiert, dass Verstöße, welche von mehr als einem Tool erkannt werden, einen eigenen Fehlercode erhalten. Alle anderen Verstöße erhalten einen allgemeinen programmspezifischen Fehlercode, der somit keine näheren Informationen über die Art des Fehlers hergibt.

Basierend auf diesen Vorbereitungen kann nun für jeden Fehler jedes Programm gefunden werden, das diesen Fehler entdeckt hat. Beispielsweise kann ein Fehler von allen drei Programmen, nur von *Checkstyle* oder ausschließlich von *PMD* und *DocEvaluator* gefunden werden. Mathematisch gesehen kann die Potenzmenge der Menge $\{\textit{Checkstyle}, \textit{PMD}, \textit{DocEvaluator}\}$ genommen werden, wodurch alle möglichen Kombinationen an Programmen entstehen, die einen bestimmten Fehler erkennen können. Durch Zählen der Fehler pro Programmkombination lässt sich bewerten, welche Programme besonders viele Fehler finden, die andere Programme nicht erkennen und ob alle drei Programme viele gemeinsame Fehler finden.

Auswahl der Regeln

Zum Vergleich der drei Programme müssen zunächst Regeln festgelegt werden, bei denen die Programme eine Warnmeldung ausgeben. Bei *Checkstyle* und *PMD* erfolgt die Konfiguration über Extensible-Markup-Language-Dateien. Beim *DocEvaluator* erfolgt die Konfiguration über das in Kapitel 4.5 beschriebene JSON-Format. Bei der Auswahl der Regeln muss beachtet werden, dass *Checkstyle* und *PMD* auch andere Fehler wie z. B. komplexe Methoden finden können. Diese sind in diesem Kontext nicht relevant und werden ignoriert. Außerdem können die drei Programme zum Teil unterschiedliche Fehler finden, da beispielsweise *PMD* (anders als *Checkstyle*) den Inhalt der Dokumentation auf das Auftauchen bestimmter Begriffe prüfen kann. *Checkstyle* kann aber dafür (anders als *PMD*) prüfen, ob die Block-Tags in der richtigen Reihenfolge definiert werden.

$CS \cap DE$	$PMD \cap DE$	$PMD \cap DE \cap CS$
<ul style="list-style-type: none"> • Komponente dokumentiert • Methode vollständig dokumentiert • Fehler in Javadoc 	<ul style="list-style-type: none"> • Komponente dokumentiert • Bestimmte Wörter in Kommentar verbieten 	<ul style="list-style-type: none"> • Komponente dokumentiert

Tabelle 5.1: Überschneidungen der Regeln der drei Programme

Die letztere Regel wird vom *DocEvaluator* in der ausgelieferten Fassung nicht geprüft. Alle drei Programme können jedoch prüfen, ob eine Komponente dokumentiert ist oder nicht. Tabelle 5.1 vergleicht die Überschneidungen der Regeln, welche die drei Programme anwenden können. Dabei stehen (sowohl hier als auch im Rest des Kapitels 5) die Abkürzungen *CS* für *Checkstyle* und *DE* für *DocEvaluator*.

Aus der Tabelle lässt sich entnehmen, dass der *DocEvaluator* mit *Checkstyle* die meisten Übereinstimmungen hat. Beide Programme können die Vollständigkeit der Methodendokumentation und typische Fehler in Javadoc-Kommentaren (wie z. B. fehlerhaftes HTML) finden. PMD und der *DocEvaluator* können bestimmte Wörter in der Dokumentation bemängeln, die in einer Dokumentation vermieden werden sollten. Alle drei Programme können das Vorhandensein der Dokumentation überprüfen. Allerdings ignoriert *PMD* alle privaten Komponenten außer Felder. Vor allem bei *Checkstyle* gibt es zudem viele Regeln, die von den anderen beiden Programmen nicht gefunden werden und auf der Webseite [36] aufgelistet werden.

Da ein Vergleich der drei Tools somit nur eingeschränkt möglich ist, wird sich die qualitative Evaluation nur auf den Kernbereich beschränken. Es werden also nur die Regeln verwendet, die von mindestens zwei Tools unterstützt werden. Dies sind alle Regeln in Tabelle 5.1. Da somit nur relativ leichte Fehler gefunden werden, können die Ergebnisse der Evaluation dazu verwendet werden, um die Parsing-Qualität zu ermitteln, denn wenn solche grundlegenden Fehler (wie z. B. das Nichtvorhandensein der Dokumentation) nicht gefunden werden, besteht eine erhebliche Chance, dass eine Java-Datei falsch interpretiert wird.

5.2.1 Ergebnisse

Tabelle 5.2 listet die Anzahl der gefundenen Fehler (gemäß Tabelle 5.1) auf. In den Spalten werden die Fehler nach dem Projekt gruppiert. Die ersten drei Zeilen beschreiben, wie viele Fehler die einzelnen Tools pro Projekt gefunden haben. So hat der *DocEvaluator* 1710 Fehler in *Log4J* gefunden. In den übrigen Zeilen wird aufgeführt, welche Kombinationen der Tools wie viele Fehler gefunden haben. Die Zeile mit der ersten Spalte „{PMD, DE}“ beschreibt beispielsweise, dass *PMD* und der *DocEvaluator* (aber nicht

5 Evaluation

	Log4J	ArgoUML	Eclipse JDT
DE	1 710	10 054	17 380
CS	1 590	9 961	17 638
PMD	1 008	9 051	12 702
{DE}	108	124	555
{CS}	26	285	273
{PMD}	86	377	298
{PMD, DE}	41	264	253
{CS, DE}	683	1 266	5 214
{PMD, CS}	3	10	793
{PMD, CS, DE}	878	8 400	11 358

Tabelle 5.2: Anzahl der gefundenen Fehler pro Projekt

Checkstyle) 41 Fehler bei *Log4J*, 264 Fehler bei *ArgoUML* und 253 Fehler bei *Eclipse JDT* gefunden haben.

Aus der Tabelle ist ersichtlich, dass stets über 50 % aller Fehler von allen drei Tools gefunden werden. Bei *Eclipse JDT* und *Log4J* werden mehr als ein Viertel der Fehler von der Kombination *DocEvaluator* und *Checkstyle* gefunden. Beim Projekt „ArgoUML“ liegt diese Quote bei weniger als 15 %. Weniger als 10 % der Fehler werden nur von einem Tool erkannt.

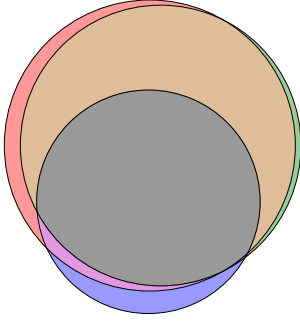
Die Überschneidungen der Fehler lassen sich auch mit Venn-Diagrammen darstellen. Die Abbildungen 5.1a, 5.1b und 5.1c zeigen für jedes Projekt die Überschneidungen der gefundenen Fehler. Die Abbildung 5.1d ist die Legende dieser drei Abbildungen:

Auch hier zeigt der graue Bereich visuell, dass die drei verglichenen Tools viele Fehler gemeinsam finden. Dies ist vor allem bei *ArgoUML* deutlich, da der graue Kreis fast alle anderen Kreise großflächig überdeckt. Bei den anderen beiden Projekten ist zudem eine große Überschneidung von *Checkstyle* und *DocEvaluator* (hellbraun) zu erkennen. Bei *Log4J* zeigt das Diagramm einen im Vergleich zu den anderen Projekten größeren Bereich (dunkelblau) an Fehlern, die nur von *PMD* gefunden werden. Auch der Bereich der exklusiv vom *DocEvaluator* gefundenen Fehler (rot) ist bei *Log4J* größer. Die nur vom *DocEvaluator* und *PMD* gefundenen Fehler (lila) sind bei *Eclipse JDT* nicht zu erkennen, bei den anderen Venn-Diagrammen allerdings schon. Dafür scheint es bei *Eclipse JDT* relativ viele Fehler zu geben, die nur von *Checkstyle* und *PMD* gefunden werden, da der hellblaue Abschnitt nur dort sichtbar ist. Außerdem gibt es bei *Eclipse JDT* mehr Fehler, die nur von *Checkstyle* gefunden werden, da der grüne Bereich größer ist.

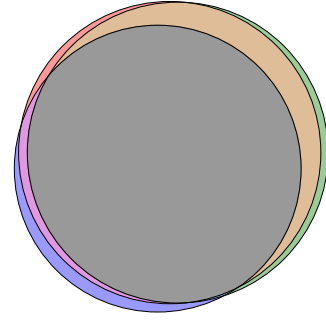
Um die Trefferrate mathematisch auszudrücken, kann die Formel

$$1 - \frac{|\{DE\}|}{|DE|} \quad (5.1)$$

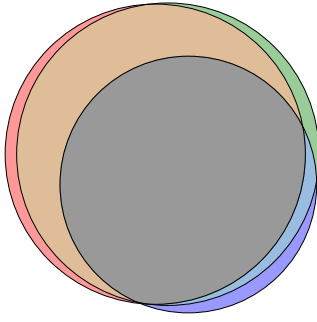
verwendet werden. Diese gibt in Prozent an, wie viele Fehler, die vom *DocEvaluator* gefunden werden, auch von den anderen beiden Tools gefunden werden. Demgegenüber



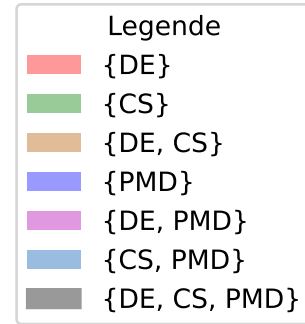
(a) Venn-Diagramm: Log4J



(b) Venn-Diagramm: ArgoUML



(c) Venn-Diagramm: Eclipse JDT



(d) Legende der Venn-Diagramme

kann auch ermittelt werden, wie viele Fehler von *Checkstyle* oder *PMD* gefunden wurden, die auch vom *DocEvaluator* erkannt wurden:

$$1 - \frac{|\{CS\} + \{PMD\} + \{CS, PMD\}|}{|PMD| + |CS|} \quad (5.2)$$

Tabelle 5.3 zeigt basierend auf den genannten Formeln (5.1 und 5.2) die Treffergenauigkeit des *DocEvaluators* für jedes analysierte Projekt: Es ist klar erkennbar, dass die Trefferrate unabhängig von der Formel und dem analysierten Projekt größer als 90 % ist, sodass die meisten Fehler, die vom *DocEvaluator* gefunden werden, von mindestens einem anderen Tool gefunden werden. Zudem werden die meisten Fehler, die von *Checkstyle* oder *PMD* gefunden werden, auch vom *DocEvaluator* gefunden.

Formel	Log4J	ArgoUML	Eclipse JDT
5.1	93,68 %	98,77 %	96,81 %
5.2	95,57 %	96,47 %	95,50 %

Tabelle 5.3: Trefferrate des *DocEvaluators* gemäß den Formeln 5.1 und 5.2

5.2.2 Bewertung der Qualitätsevaluation

Insgesamt zeigt die Evaluation der Qualität, dass der *DocEvaluator* eine hohe Abdeckung mit *Checkstyle* und *PMD* hat und somit die meisten von diesen Tools gefundenen Fehler auch findet. Somit ist das abstrakte Format zur Repräsentation einer Quellcodedatei geeignet, um die meisten Aspekte, welche für die Dokumentation relevant sind, zu beschreiben. Allerdings wurde diese Evaluation auf größere Projekte (über 10 000 LOC) beschränkt, sodass nicht geprüft wurde, ob der *DocEvaluator* auch bei kleineren Projekten eine genaue Einschätzung der Dokumentationsqualität gibt. Tendenziell wird die Trefferrate kleiner sein, da durch die geringere Größe jeder nicht gefundene Fehler ein höheres Gewicht hat.

Während der Evaluation wurde geprüft, warum einige Fehler nicht vom *DocEvaluator* gefunden wurden. In einigen Fällen waren dies einfache Fehler, die bereits behoben sind, sodass dadurch die Trefferrate erhöht wurde. In anderen Fällen gibt es größere strukturelle Probleme, die nicht mehr leicht behebbar sind. Einige dieser Fehler werden im Folgenden präsentiert:

Fehler durch verschiedene Zeilennummerierung

Einige Fehler sind keine Fehler des *DocEvaluators* an sich, sondern sind in der Methodik der Evaluation begründet. Um gemeinsame Fehler zu finden, müssen die gefundene Fehler pro Tool abgeglichen werden, wozu die Zeilennummer elementar ist. Allerdings gibt es Unterschiede bei der Festlegung der Zeilennummer. Der *DocEvaluator* verwendet in seiner Logausgabe ein Zeilennummerintervall, der mit der Zeile des Bezeichners der Komponente endet und dessen Anfang durch Subtraktion der Anzahl der Zeilen der Dokumentation von der Zeile des Bezeichners definiert wird. Die anderen Tools verwenden die exakte Zeile eines Fehlers. In den meisten Fällen ist dies kein Problem, da diese Zeilennummer von *Checkstyle* und *PMD* innerhalb des vom *DocEvaluator* beschriebenen Intervalls liegen muss. Listing 5.1 zeigt ein Beispiel, wo es problematisch wird.

```

1  /**
2   * this a test method
3   */
4   void test(int a,
5             int b,
6             int c) {}

```

Listing 5.1: Methode auf viele Zeilen verteilt

Besonders an dieser Methode ist, dass die Parameter auf verschiedenen Zeilen verteilt ist. Während *Checkstyle* bei einem undokumentierten Parameter die exakte Zeile eines nicht dokumentierten Parameters ausgibt (z. B. Z. 5), würde der *DocEvaluator* die Zeilen 1 bis 4 ausgeben, da die Dokumentation bei Zeile 1 beginnt und der Bezeichner der Komponente in Zeile 4 definiert ist. Ähnlich problematisch ist es, wenn zwischen der Dokumentation und dem Bezeichner noch Annotationen stehen, sodass der Bezeichner um eine Zeile nach unten rutscht.

Klassen in Methoden

In Java können Klassen in Methoden deklariert werden bzw. anonyme Klassen direkt instantiiert werden. Diese Klassen können ebenfalls Javadoc besitzen, werden allerdings vom *DocEvaluator* ignoriert, da der *DocEvaluator* jeglichen Code in Methoden nur unstrukturiert als Zeichenkette speichert und nicht weiterverarbeitet. Die anderen beiden Tools prüfen auch diese Klassen, sodass sie entsprechend einige Fehler finden, die der *DocEvaluator* nicht mehr finden kann.

Fehler in einzeiligen Kommentaren bei PMD

Anders als der *DocEvaluator* und *Checkstyle* berücksichtigt *PMD* auch einzeilige Kommentare. Wenn ein einzeiliger Kommentar ein unzulässiges Wort enthält, so würde *PMD* einen Fehler melden, aber der *DocEvaluator* nicht, da er nur Javadoc-Kommentare prüft. Dadurch kommt es zu einer Verzerrung und der Anteil der alleinig von *PMD* gefundenen Fehler wird überschätzt.

5.3 Analyse der Geschwindigkeit

In diesem Abschnitt wird der *DocEvaluator* mit *Checkstyle* und *PMD* bezüglich der Geschwindigkeit verglichen. Damit soll geprüft werden, ob das Tool nicht nur eine ausreichende Qualität besitzt, sondern auch in einer angemessenen Zeit ein Ergebnis liefert. Dies ist im CI/CD-Kontext wichtig, da bei einer langen Laufzeit des Tools die Bereitstellung eines geprüften Softwareprojektes verzögert wird und somit die Produktivität reduziert wird.

Durchführung der Geschwindigkeitsevaluation

Zur Durchführung der Evaluation der Geschwindigkeit analysieren die drei Tools die in Kapitel 5.1 genannten Projekte. Damit jedes Programm fair behandelt wird und eine ungefähr gleiche Menge an Analysen durchführen kann, werden die Regeln so beschränkt, dass nur noch das Vorhandensein von Dokumentation geprüft wird. So wird verhindert, dass beispielsweise der *DocEvaluator* und *Checkstyle* die Dokumentation von Methodenparameter überprüfen, während *PMD* dies ignoriert. Bei jeder Analyse wird die Zeit gemessen, die vom Start eines Tools bis zu dessen Beendigung vergehen.

Die Ausgabe jedes Tools wird auf „dev/null“ umgeleitet, sodass jegliche Ausgabe ignoriert wird. Dadurch können Schwankungen unberücksichtigt bleiben, die bei der Verwendung von Eingabe- und Ausgabegeräten auftreten. So sind die Ergebnisse näher an der tatsächlichen Verarbeitungsgeschwindigkeit. Nachteilhaft an diesem Vorgehen ist, dass die Tools im Praxiseinsatz eine Ausgabe produzieren müssen, um überhaupt dem Entwickler helfen zu können, sodass dieser wichtige Aspekt hier ignoriert wird.

Die Analyse jedes Projektes mit jedem Tool wird zehnmal durchgeführt, um Schwankungen durch Hintergrundprozesse oder andere Einflussfaktoren auszugleichen. Die Evaluation der Geschwindigkeit wird auf einem Laptop mit dem Prozessor „i7-1165G7“ mit 16 GB Arbeitsspeicher durchgeführt. Dabei wurden alle Programme auf dem Computer geschlossen und die Berechnungen wurden ohne grafische Benutzeroberfläche durchgeführt, um Schwankungen in der Laufzeit zu minimieren.

5.3.1 Ergebnisse

Die Tabellen 5.4 und 5.5 zeigen den Median und die Standardabweichung der benötigten Durchlaufzeit pro Tool und Projekt in Sekunden. Die Abbildungen 5.2a, 5.2b und 5.2c visualisieren den Inhalt der Tabellen als Boxplot.

Im Verzeichnis „speed_eval“ des digitalen Anhangs befinden sich die Rohdaten der Geschwindigkeitsmessung, gruppiert nach dem analysierten Projekt. Auch die Originaldateien der Boxplots befinden sich dort.

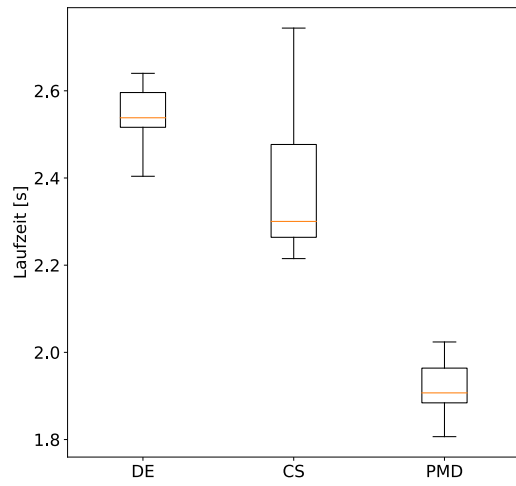
	DE	CS	PMD
Log4J	2,538	2,30	1,907
ArgoUML	16,965	9,301	7,917
Eclipse JDT	69,148	27,316	21,586

Tabelle 5.4: Median der Laufzeit in Sekunden

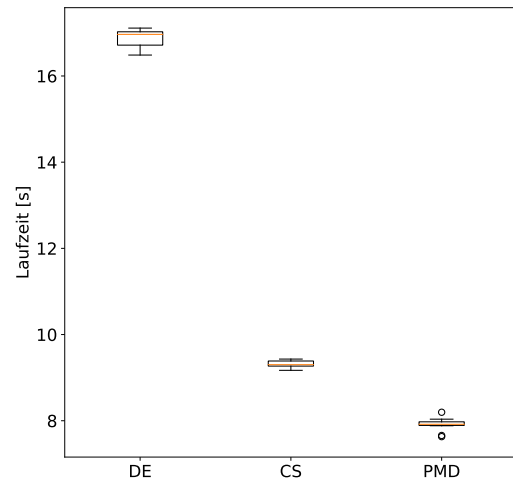
	DE	CS	PMD
Log4J	0,077	0,174	0,068
ArgoUML	0,197	0,083	0,157
Eclipse JDT	1,594	0,265	0,270

Tabelle 5.5: Standardabweichung der Laufzeit in Sekunden

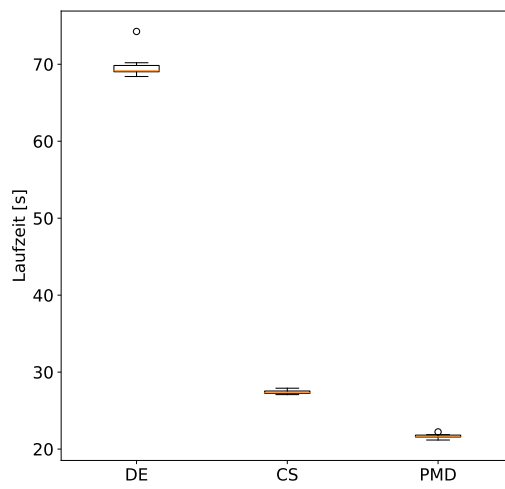
Aus den Tabellen und Boxplot-Diagrammen wird ersichtlich, dass der *DocEvaluator* im Durchschnitt länger benötigt, um die Projekte zu bewerten. Wie aus dem Boxplot-Diagramm 5.2a zu entnehmen ist, gab es nur bei Log4J einen Ausreißer, bei dem *Checkstyle* mehr Zeit benötigt hat. Ansonsten war *Checkstyle* stets schneller als der *DocEvaluator*. *PMD* analysiert am schnellsten ein Projekt. Der Unterschied in der Laufzeit zwischen dem *DocEvaluator* und den anderen Tools steigt stark mit wachsender Größe des analysierten Projektes. Bei dem kleinsten Projekt *Log4J* benötigt der *DocEvaluator*



(a) Boxplot: Log4J



(b) Boxplot: ArgoUML



(c) Boxplot: Eclipse JDT

durchschnittlich die 1,103-fache Zeit im Vergleich zu *Checkstyle*. Bei dem größten Projekt *Eclipse JDT* benötigt der *DocEvaluator* durchschnittlich 2,53-mal so viel Zeit wie *Checkstyle*. Beim *DocEvaluator* und *PMD* steigt die Standardabweichung mit der Größe des Projektes, während dieser Trend bei *Checkstyle* nicht so eindeutig ist.

Bei dem Boxplot-Diagramm 5.2a zu *Log4J* ist auch erkennbar, dass Ausreißer der Laufzeit nach oben häufiger sind als nach unten, da der Median sich stets im unteren Bereich des Boxplot-Quartils befindet. Bei den anderen Projekten ist dies aufgrund des größeren Zeitabstandes zwischen dem *DocEvaluator* und den anderen Tools und der daraus resultierenden Verkleinerung der einzelnen Boxplots nicht so klar im Boxplot ersichtlich, allerdings stimmt diese Aussage größtenteils auch dort. Nur bei der Analyse von *ArgoUML* durch den *DocEvaluator* (Abbildung 5.2b) scheinen Abweichungen nach unten häufiger zu sein.

5.3.2 Bewertung der Ergebnisse

Insgesamt zeigt die Evaluation der Geschwindigkeit, dass der *DocEvaluator* langsamer arbeitet als die anderen Programmen. Allerdings bleibt die Laufzeit auf einem angemessenen Niveau, da die Verarbeitung von dem größten Projekt *Eclipse JDT* mit 400 000 LOC im Durchschnitt nur etwas mehr als einer Minute benötigt. Nichtsdestotrotz ignoriert diese Analyse, dass die Ausgabe von Ergebnissen über die Konsole hier nicht berücksichtigt wurde und nur eine einfache Metrik angewendet wurde. Bei einem Experiment mit aktivierter Ausgabe wurden ähnliche Ergebnisse produziert, insbesondere ist der *DocEvaluator* weiterhin das langsamste Programm.

Für die schlechtere Laufzeit des *DocEvaluator* im Vergleich zu *Checkstyle* und *PMD* lassen sich zwei Hauptargumente finden. So soll *Node.js*, welches die Plattform des Tools ist, langsamer sein als Java, in dem *Checkstyle* und *PMD* programmiert sind [55]. Außerdem ist zu beachten, dass der *DocEvaluator* Metriken berechnen soll und daher die Zwischenergebnisse aller Komponenten speichern muss, um daraus ein Gesamtergebnis mittels eines arithmetischen Mittelwerts oder eines anderen Algorithmus' berechnen zu können. Auch wenn dieses Gesamtergebnis bei der Laufzeitevaluation uninteressant ist, wird es dennoch berechnet, was zusätzliche Laufzeit benötigt. Dies erklärt auch die starke Steigerung des Zeitaufwands bei größeren Projekten.

5.4 Fazit der Evaluation

Insgesamt zeigt die Evaluation sowohl bezüglich der Qualität als auch der Geschwindigkeit, dass der *DocEvaluator* mit den anderen Tools mithalten kann. Zwar wird nicht jeder Fehler gefunden, aber die Trefferrate ist hoch und durch die Verwendung eines abstrakten Formates, das für mehrere Programmiersprachen geeignet ist, sind solche Abstriche nicht vermeidbar.

Auch bei der Geschwindigkeit zeigt sich, dass der *DocEvaluator* zwar im Extremfall zweieinhalbmal langsamer ist als die anderen Tools, allerdings ist die Laufzeit mit knapp 70 Sekunden im Extremfall bei einem großen Projekt noch in einem (relativ) angemess-

senen Rahmen. Nichtsdestotrotz ist es eine Überlegung wert, den *DocEvaluator* weiter zu optimieren, damit die Laufzeit verbessert wird.

6 Fazit

Ziel dieser Bachelorarbeit war es, ein Tool zu Bewertung der Dokumentation zu entwickeln, das in einem CI/CD-Prozess eingebunden werden kann und nicht auf einer Programmiersprache beschränkt ist. Dabei beschränkt sich diese Bachelorarbeit auf strukturierte Kommentaren wie z. B. Javadoc. Dieses Ziel wurde im Großen und Ganzen erreicht.

Durch die allgemein gehaltene Klassenstruktur für das Parsing ist es möglich, Quellcode in anderen Programmiersprachen bewerten zu lassen. Allerdings muss dafür ein entsprechender Parser geschrieben werden, welcher den Quellcode in die vorgegebene Struktur transformiert. Dabei ist es natürlich nicht möglich, jedes Detail abzubilden, sondern es müssen Abstriche gemacht werden. Nichtsdestotrotz können auch sprachspezifische Eigenheiten berücksichtigt werden, indem eine entsprechende abgeleitete Klasse von *ComponentMetaInformation* gebildet wird und diese sprachspezifischen Informationen dort gespeichert werden. Diese Daten können von einem geeigneten *LanguageSpecificHelper* dazu verwendet werden, um sprachabhängige Details bei der Bewertung der Dokumentation zu berücksichtigen.

Auch das Parsen der strukturierten Kommentare erfolgt recht abstrakt, indem die Informationen in den Beschreibungstexten unstrukturiert als Zeichenketten gespeichert werden, sodass die einzelnen Metriken diese Informationen weiterverarbeiten müssen. Da es Metriken gibt, die mit den einzelnen Wörtern ein eines Kommentars arbeiten und auch Metriken, welche die interne Struktur des Kommentars analysieren, wäre es ein mögliches Forschungsthema, wie diese zwei Darstellungen besser repräsentiert werden können.

Um das Tool konfigurierbar zu halten, wurde ein Konzept für eine Konfigurationsdatei im JSON-Format vorgestellt, das alle wichtigen Informationen enthält. Die Konfiguration kann auch über GitHub Actions durchgeführt werden, indem passende Umgebungsvariablen gesetzt werden. So kann das Tool sowohl als reguläres Programm auf einem lokalen System verwendet werden als auch mittels GitHub Actions in den CI/CD-Prozess eingebunden werden. Durch die flexible Konfiguration kann ein Nutzer frei entscheiden, welche Metriken er für sinnvoll hält und wie er sie gewichten will. Dabei überschreibt die Konfiguration mittels GitHub Actions stets die Konfiguration in der JSON-Datei. Außerdem können die Metriken selbst begrenzt konfiguriert werden. Eine Nutzung des Tools auf anderen CI/CD-Plattformen, die mit GitHub Actions vergleichbar sind, ist prinzipiell ebenfalls möglich, da die Konfiguration von dem übrigen Programm entkoppelt ist.

Für das Tool wurden bereits einige Metriken entwickelt, welche verschiedene Bereiche der Dokumentation analysieren können. Leider war eine Evaluation der einzelnen Metriken nicht möglich, sodass weiterhin offenbleibt, welche Metriken in welchen Situationen

valide Ergebnisse liefern und wie eine sinnvolle Gewichtung der Metriken aussehen kann. Weitere Metriken lassen sich durch Einfügen einer neuen abgeleiteten Klasse von *DocumentationAnalysisMetric* bilden. Schwierig kann im konkreten Einzelfall allerdings das Finden einer geeigneten Funktion werden, welche die Werte der Metrik in das vorgegebene Intervall von 0 bis 100 transformiert. Mögliche Ideen für weitere Metriken lassen sich in [36] finden. Auch die wissenschaftliche Literatur liefert weitere Metriken, die fortschrittliche Techniken im Bereich des NLP nutzen und daher im Rahmen dieser Bachelorarbeit zu komplex waren. Beispielhaft sei hier das Tool „iComment“ aus [13, S. 145ff.] genannt, bei dem geprüft wurde, ob die Dokumentation mit dem Code auch konsistent ist und somit korrekt und aktuell ist. Diese Metriken können als Inspiration genommen werden, um die Dokumentationsqualität umfassender analysieren zu können, sodass Entwickler bei der Identifikation und Behebung von mangelhafter Softwaredokumentation besser unterstützt werden.

Literaturverzeichnis

- [1] P. Thomson. „Static Analysis: An Introduction: The Fundamental Challenge of Software Engineering is One of Complexity.“ In: *Queue* 19.4 (Aug. 2021), S. 29–41. ISSN: 1542-7730. DOI: 10.1145/3487019.3487021.
- [2] D. Steidl, B. Hummel und E. Juergens. „Quality analysis of source code comments“. In: *2013 21st International Conference on Program Comprehension (ICPC)*. 2013, S. 83–92. DOI: 10.1109/ICPC.2013.6613836.
- [3] N. Khamis, R. Witte und J. Rilling. „Automatic Quality Assessment of Source Code Comments: The JavadocMiner“. In: *Natural Language Processing and Information Systems*. Hrsg. von C. J. Hopfe u. a. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, S. 68–79. ISBN: 978-3-642-13881-2. DOI: 10.1007/978-3-642-13881-2_7.
- [4] F. Young. „Software engineering and software documentation: a unified long course“. In: *Proceedings Frontiers in Education Twenty-First Annual Conference. Engineering Education in a New World Order*. 1991, S. 593–595. DOI: 10.1109/FIE.1991.187557.
- [5] A. Forward und T. C. Lethbridge. „The Relevance of Software Documentation, Tools and Technologies: A Survey“. In: *Proceedings of the 2002 ACM Symposium on Document Engineering*. DocEng ’02. McLean, Virginia, USA: Association for Computing Machinery, 2002, S. 26–33. ISBN: 978-1-58113-594-7. DOI: 10.1145/585058.585065.
- [6] E. Aghajani u. a. „Software Documentation Issues Unveiled“. In: *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. 2019, S. 1199–1210. DOI: 10.1109/ICSE.2019.00122.
- [7] R. Plösch, A. Dautovic und M. Saft. „The Value of Software Documentation Quality“. In: *2014 14th International Conference on Quality Software*. 2014, S. 333–342. DOI: 10.1109/QSIC.2014.22.
- [8] T. Vestdam. „Writing Internal Documentation.“ In: *EuroPLoP*. Jan. 2001, S. 511–534.
- [9] S. C J und A. Mahendran. „Software Documentation Management Issues and Practices: A Survey“. In: *Indian Journal of Science and Technology* 9 (Mai 2016). DOI: 10.17485/ijst/2016/v9i20/86869.
- [10] S. L. Pfleeger, J. C. Fitzgerald und D. A. Rippey. „Using multiple metrics for analysis of improvement“. In: *Software Quality Journal* 1.1 (1992), S. 27–36. DOI: <https://doi.org/10.1007/BF01720167>.

- [11] K. Mordal u. a. „Software quality metrics aggregation in industry“. In: *Journal of Software: Evolution and Process* 25.10 (2013), S. 1117–1135. DOI: <https://doi.org/10.1002/smr.1558>.
- [12] C. Vassallo u. a. „Continuous Delivery Practices in a Large Financial Organization“. In: *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 2016, S. 519–528. DOI: 10.1109/ICSME.2016.72.
- [13] L. Tan u. a. „/*icomment: Bugs or Bad Comments?*/“. In: *Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles*. SOSP ’07. Stevenson, Washington, USA: Association for Computing Machinery, 2007, S. 145–158. ISBN: 978-1-59593-591-5. DOI: 10.1145/1294261.1294276.
- [14] *Github Actions*. URL: <https://docs.github.com/en/actions> (besucht am 16. Dez. 2021).
- [15] M. Asay. *Microsoft CEO accidentally underplays GitHub’s pervasiveness*. 30. Juli 2021. URL: <https://www.techrepublic.com/article/microsoft-ceo-accidentally-underplays-githubs-pervasiveness/> (besucht am 24. Feb. 2022).
- [16] „IEEE Standard Glossary of Software Engineering Terminology“. In: *IEEE Std 610.12-1990* (1990), S. 1–84. DOI: 10.1109/IEEESTD.1990.101064.
- [17] J. M. Jose und T. Viswanathan. „Software documentation and standards“. In: *Annals of library science and documentation* 39 (1992), S. 123–133.
- [18] D. E. Knuth. „Literate Programming“. In: *The Computer Journal* 27.2 (Jan. 1984), S. 97–111. ISSN: 0010-4620. DOI: 10.1093/comjnl/27.2.97.
- [19] S. C. B. de Souza, N. Anquetil und K. M. de Oliveira. „A Study of the Documentation Essential to Software Maintenance“. In: *Proceedings of the 23rd Annual International Conference on Design of Communication: Documenting & Designing for Pervasive Information*. SIGDOC ’05. Coventry, United Kingdom: Association for Computing Machinery, 2005, S. 68–75. ISBN: 978-1-58113-594-7. DOI: 10.1145/1085313.1085331.
- [20] *How to Write Doc Comments for the Javadoc Tool*. URL: <https://www.oracle.com/de/technical-resources/articles/java/javadoc-tool.html> (besucht am 16. Dez. 2021).
- [21] M. Steinbeck und R. Koschke. „Javadoc Violations and Their Evolution in Open-Source Software“. In: *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 2021, S. 249–259. DOI: 10.1109/SANER50967.2021.00031.
- [22] *Doxygen*. URL: <https://www.doxygen.nl/manual/docblocks.html> (besucht am 22. Feb. 2022).
- [23] *Docstring*. URL: <https://www.python.org/dev/peps/pep-0257/> (besucht am 22. Feb. 2022).

- [24] M. Fowler und K. Beck. *Refactoring: Improving the Design of Existing Code*. A Martin Fowler signature book. Addison-Wesley, 2019. ISBN: 978-0-13-475759-9.
- [25] *JavaScript-Referenz*. URL: <https://developer.mozilla.org/en-US/docs/Web/JavaScript> (besucht am 28. März 2022).
- [26] B. Ray u. a. „A Large Scale Study of Programming Languages and Code Quality in Github“. In: *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. FSE 2014. Hong Kong, China: Association for Computing Machinery, 2014, S. 155–165. ISBN: 978-1-4503-3056-5. DOI: 10.1145/2635868.2635922.
- [27] *Typescript-Referenz*. URL: <https://www.typescriptlang.org/docs/> (besucht am 28. März 2022).
- [28] *Node.js Webseite*. (Besucht am 28. März 2022).
- [29] *ANTLR*. URL: <https://www.antlr.org/> (besucht am 17. Feb. 2022).
- [30] *Java-parser*. URL: <https://www.npmjs.com/package/java-parser> (besucht am 1. Dez. 2021).
- [31] T. J. Parr und R. W. Quong. „ANTLR: A predicated-LL(k) parser generator“. In: *Software: Practice and Experience* 25.7 (Juli 1995), S. 789–810. ISSN: 0038-0644. DOI: 10.1002/spe.4380250705.
- [32] T. Parr. *The Definitive ANTLR 4 Reference*. 2nd. Pragmatic Bookshelf, 2013. ISBN: 978-1-934356-99-9.
- [33] *ANTLR-Grammatik für Java*. URL: <https://github.com/antlr/grammars-v4/tree/master/java> (besucht am 17. Feb. 2022).
- [34] C. C. P. Herath. *Hands-on GitHub Actions. Implement CI/CD with GitHub Action Workflows for Your Applications*. 1. Aufl. Apress, Berkeley, CA, 2021. 162 S. ISBN: 978-1-4842-6464-5. DOI: <https://doi.org/10.1007/978-1-4842-6464-5>.
- [35] *Checkstyle*. URL: <https://checkstyle.sourceforge.io/> (besucht am 16. Dez. 2021).
- [36] *Checkstyle Javadoc-Metriken*. URL: https://checkstyle.sourceforge.io/config_javadoc.html (besucht am 22. Feb. 2022).
- [37] *PMD*. URL: <https://pmd.github.io/> (besucht am 17. Feb. 2022).
- [38] D. Schreck, V. Dallmeier und T. Zimmermann. „How Documentation Evolves over Time“. In: *Ninth International Workshop on Principles of Software Evolution: In Conjunction with the 6th ESEC/FSE Joint Meeting*. IWPSE '07. Dubrovnik, Croatia: Association for Computing Machinery, 2007, S. 4–10. ISBN: 978-1-59593-722-3. DOI: 10.1145/1294948.1294952.
- [39] S. H. Tan u. a. „@tComment: Testing Javadoc Comments to Detect Comment-Code Inconsistencies“. In: *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*. 2012, S. 260–269. DOI: 10.1109/ICST.2012.106.

- [40] E. Gamma u. a. *Design Patterns: Entwurfsmuster als Elemente wiederverwendbarer objektorientierter Software*. mitp Professional. MITP-Verlags GmbH & Co. KG, 2015. ISBN: 978-3-8266-9700-5.
- [41] *Minimatch*. URL: <https://github.com/isaacs/minimatch> (besucht am 16. Dez. 2021).
- [42] *Add and commit*. URL: <https://github.com/EndBug/add-and-commit> (besucht am 17. Feb. 2022).
- [43] *NCC*. URL: <https://github.com/vercel/ncc> (besucht am 16. Dez. 2021).
- [44] *Branch-Push*. URL: <https://github.com/ActionwareIO/branch-push-action> (besucht am 22. Feb. 2022).
- [45] W. Dubay. „The Principles of Readability“. In: *CA* 92627949 (Jan. 2004), S. 631–3309.
- [46] J. Bogert. „In defense of the Fog Index“. In: *The Bulletin of the Association for Business Communication* 48.2 (1985), S. 9–12.
- [47] S. Colebourne. *Javadoc coding standards*. URL: <https://blog.joda.org/2012/11/javadoc-coding-standards.html> (besucht am 22. Feb. 2022).
- [48] R. C. Martin. *Clean code: a handbook of agile software craftsmanship*. Pearson Education, 2009. ISBN: 978-0-1323-5088-4.
- [49] B. Vasilescu, A. Serebrenik und M. van den Brand. „You can’t control the unfamiliar: A study on the relations between aggregation techniques for software metrics“. In: *2011 27th IEEE International Conference on Software Maintenance (ICSM)*. 2011, S. 313–322. DOI: 10.1109/ICSM.2011.6080798.
- [50] R. R. Yager. „Fusion of ordinal information using weighted median aggregation“. In: *International Journal of Approximate Reasoning* 18.1 (1998), S. 35–52. ISSN: 0888-613X. URL: <https://www.sciencedirect.com/science/article/pii/S0888613X97100032>.
- [51] K. Mordal-Manet u. a. „An Empirical Model for Continuous and Weighted Metric Aggregation“. In: *2011 15th European Conference on Software Maintenance and Reengineering*. 2011, S. 141–150. DOI: 10.1109/CSMR.2011.20.
- [52] *Log4J V 1 Quellcode*. URL: <https://github.com/apache/logging-log4j1/tree/b7e9154128cd4ae1244c877a6fda8f834a0f2247/src> (besucht am 5. Apr. 2022).
- [53] *ArgoUML Quellcode*. URL: <https://github.com/argouml-tigris-org/argouml/tree/3ddc7f94d83c4e2f439ca6fc0fe787616a1900a9/src> (besucht am 5. Apr. 2022).
- [54] *Eclipse JDT Quellcode*. URL: <https://github.com/eclipse/aspectj.eclipse.jdt.core/tree/37759146f8c5e68e69b19258bc31a8c812e8afb7/org.eclipse.jdt.core> (besucht am 5. Apr. 2022).

- [55] A. Chikina und A. Sushevich. *Node.js vs Java: Why Compare?* 15. Juli 2019. URL: <https://intexsoft.com/blog/node-js-vs-java-why-compare/> (besucht am 9. März 2022).

Anhänge

A Änderungen an der Parserdatei

Zeile	Änderung	Begründung
116	Deklaration Kommentar	Hier wird ein mehrzeiliger Kommentar definiert, dies ist hier ein Alias für den Token <i>JCOMMENT</i>
127–128	<i>comment</i> als mögliches Präfix in Klassenmember	Hier wird dem Parser mitgeteilt, dass ein Bestandteil einer Klasse wie z. B. eine Methode einen Javadoc-Kommentar besitzen kann
47	<i>comment</i> als mögliches Präfix vor Datentyp	Hier wird dem Parser mitgeteilt, dass ein Datentyp (Klasse, Schnittstelle etc.) einen Javadoc-Kommentar haben kann
404	Zulassung von Javadoc in Methoden	Da Javadoc-Kommentare an beliebigen Stellen auftauchen können, auch wenn es nicht empfohlen wird und keinen Mehrwert bietet, wird hier sichergestellt, dass solche Kommentare nicht zu Warnungen oder Fehler von ANTLR4 führen. Diese Javadoc-Kommentare werden nichtsdestotrotz später ignoriert
34, 38	Zulassung von Kommentaren vor Paketdeklarationen und Imports	Hier werden Kommentare auch vor Paketdeklarationen und Import-Statements erlaubt, was vor allem bei Klassen mit Urheberrechtsangabe sinnvoll ist
105	Zulassung von Kommentaren bei Enumerationen	Zwar werden Javadoc-Kommentare in Enumerationen mit diesem Tool nicht betrachtet, sie führen aber dennoch zu Warnungen und Fehlermeldungen. Daher werden sie hier zugelassen, aber später ignoriert
82, 83	Erzeugung eines separaten Knotens für <i>Extends</i> - und <i>Implementations</i> -Deklarationen	In der originalen Version der Parserdatei wurde die Definition der Basisklasse bzw. der implementierten Schnittstellen direkt über die Tokens <i>EXTENDS</i> bzw. <i>IMPLEMENTS</i> gelöst. Dies wurde in einem neuen Knoten <i>extendClass</i> bzw. <i>implementInterfaces</i> ausgegliedert, um so das Parsing etwas zu vereinfachen

Tabelle A.1: Änderungen an der Parserdatei

B UML-Diagramm: Parser

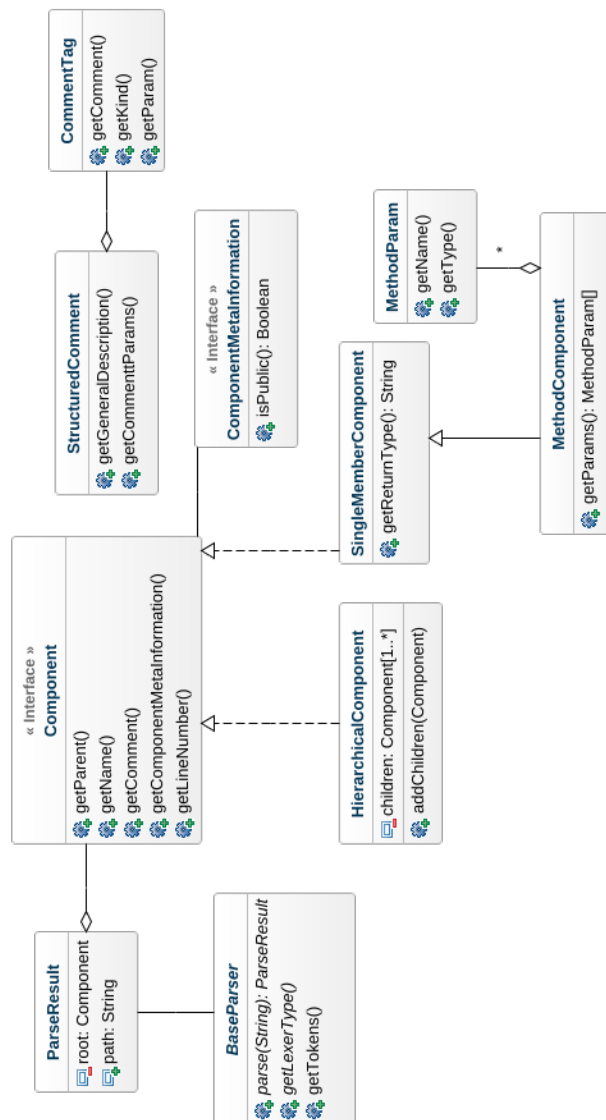


Abbildung B.1: UML-Diagramme aller Klassen, die relevant für das Parsen sind

C UML-Diagramm: Metriken

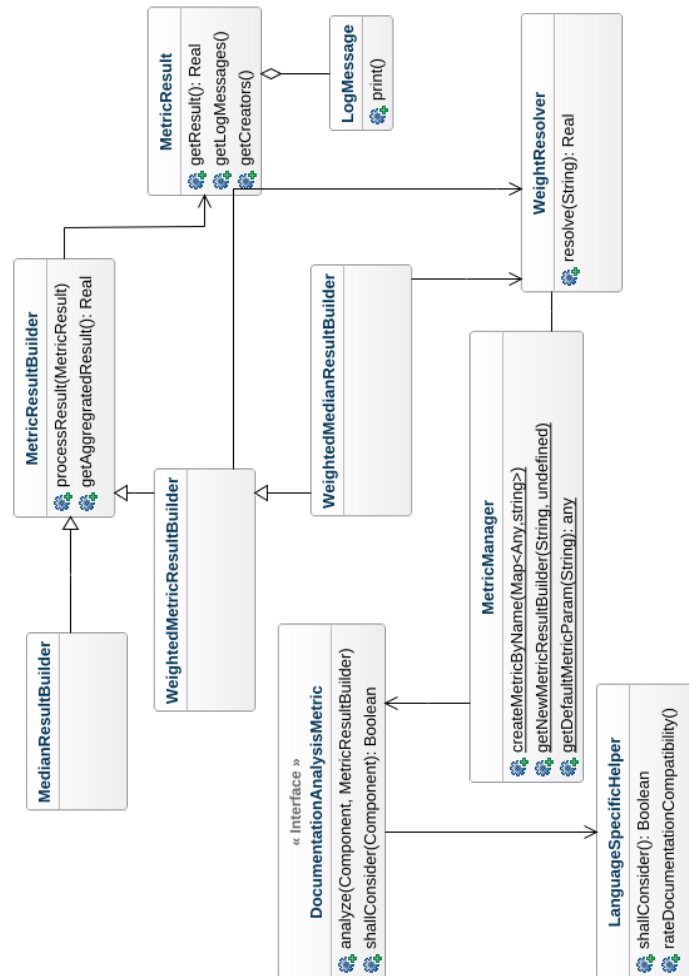


Abbildung C.1: UML-Diagramme aller Klassen, die relevant für die Metriken sind

D Konfiguration des Tools

include Alle Dateien, die bei der Bewertung der Dokumentationsqualität berücksichtigt werden müssen

exclude Teilmenge von include, enthält Dateien, die nicht weiter betrachtet werden müssen

metrics Alle Metriken, die das Tool verwenden soll. Dies ist ein Array von Objekten mit der Struktur „(name,weight, unique_name, params)“, wobei *weight* das Gewicht der jeweiligen Metrik ist (Bei Algorithmen ohne Relevanz des Gewichts wird es ignoriert), *name* der Name der Metrik und *params* ein Objekt mit den Parametern der Metrik

absolute_threshold Mindestwert der Bewertung, die erreicht werden muss, sonst wird die Dokumentationsqualität nicht akzeptiert

builder Der Algorithmus/*ResultBuilder*, der die einzelnen Ergebnisse verarbeitet.

parser Kann verwendet, um die zu parsende Programmiersprache zu wählen. Dazu muss *ParserFactory* angepasst werden

path_weights Ein Array von Objekten der Struktur „(path,weight)“. Wird verwendet, um einzelne Pfade höher oder niedriger zu gewichtet

component_weights Ein Array von Objekten der Struktur „(name,weight)“. Wird verwendet, um einzelne Komponenten höher oder niedriger zu gewichtet

default_path_weight Das Standardgewicht für eine Datei, wenn keine passende Gewichtung gefunden wurde

default_component_weight Das Standardgewicht einer Komponente, wenn keine passende Gewichtung gefunden wurde

state_manager Kann verwendet werden, um festzulegen, wie das letzte Ergebnis der Dokumentationsqualität gespeichert werden soll. Weitere Möglichkeiten können durch Erweiterung der *StateManagerFactory* hinzugefügt werden.

relative_threshold Der maximale relative Abstand zur letzten Dokumentationsqualität bevor eine Fehlermeldung geworfen wird.

builder_params Parameter für die *MetricResultBuilder*. Diese wird aktuell nur von dem Squal-Builder (Kapitel 4.9.3) genutzt

E Implementierte Metriken

Anteil dokumentierter Komponenten an allen Komponenten

Metrikname simple_comment

Klassenname SimpleCommentPresentMetric

Beschreibung Berechnet den Anteil der dokumentierten Komponenten an allen Komponenten, kann Getter und Setter ignorieren

Quellen [38, S. 5]

Anteil öffentlicher dokumentierter Komponenten

Metrikname public_members_only

Klassenname SimplePublicMembersOnlyMetric

Beschreibung Berechnet den Anteil der öffentlichen dokumentierten Komponenten an allen öffentlichen Komponenten, kann Getter und Setter ignorieren

Quellen [21, S. 253]

Bestrafung langer undokumentierter Methoden

Metrikname large_method_commented

Klassenname SimpleLargeMethodCommentedMetric

Beschreibung Bestraft undokumentierte Methoden je nach ihrer Länge

Quellen Eigene Idee

Vollständigkeit der Dokumentation von Methoden

Metrikname method_fully_documented

Klassenname SimpleMethodDocumentationMetric

Beschreibung Prüft, ob alle Methodenparameter und Rückgabewert dokumentiert sind

Quellen [38, S. 5]

Anteil dokumentierter Methoden unter Berücksichtigung der LOC

Metrikname commented_lines

Klassenname CommentedLinesRatioMetric

Beschreibung Berechnet den Anteil der LOC der dokumentierten Methoden an allen LOC aller Methoden

Quellen Eigene Idee

Flesch-Score

Metrikname flesch

Klassenname FleschMetric

Beschreibung Berechnet den Flesch-Score des Kommentars und bewertet so, ob der Kommentar verständlich ist

Quellen [3, S. 72]

Kohärenz zwischen Kommentar und Komponentename

Metrikname comment_name_coherence

Klassenname CommentNameCoherenceMetric

Beschreibung Prüft, ob der Kommentar und der Name der dokumentierten Komponente sehr ähnlich sind oder keine Ähnlichkeit haben, arbeitet nur mit Methoden

Quellen [2, S. 86ff]

Verwendung bestimmter Wörter bestrafen

Metrikname certain_terms

Klassenname CertainTermCountMetric

Beschreibung Bestraft das Vorkommen bestimmter Wörter (wie z. B. Abkürzungen)

Quellen Inspiriert von Verbot lateinischer Ausdrücke nach [20]

Bewertung der Formatierung

Metrikname formatting_good

Klassenname FormattingGoodMetric

Beschreibung Überprüft, ob korrekte Tags verwendet wurde, HTML-Tags geschlossen wurden und bei langen Methoden überhaupt eine Formatierung verwendet wurden

Quellen Inspiriert von Regel in Checkstyle [36]

Rechtschreibfehler bestrafen

Metrikname spelling

Klassenname SpellingMetric

Beschreibung Sucht nach Rechtschreibfehlern und bestraft sie

Quellen Eigene Idee

Erwähnung von Randfällen bei Methodenparameter und -rückgabewerte

Metrikname edge_case

Klassenname EdgeCaseMetric

Beschreibung Prüft, ob bei der Dokumentation von Parametern die Behandlung des Wertes *null* erwähnt wird

Quellen Inspiriert von Idee in [47]. In [39, S. 1ff.] wird ebenfalls auf einer ähnlichen Art und Weise die Erwähnung von Randfällen geprüft, dort aber auch, ob diese Angaben korrekt sind

Gunning-Fog-Index

Metrikname gunning_fog

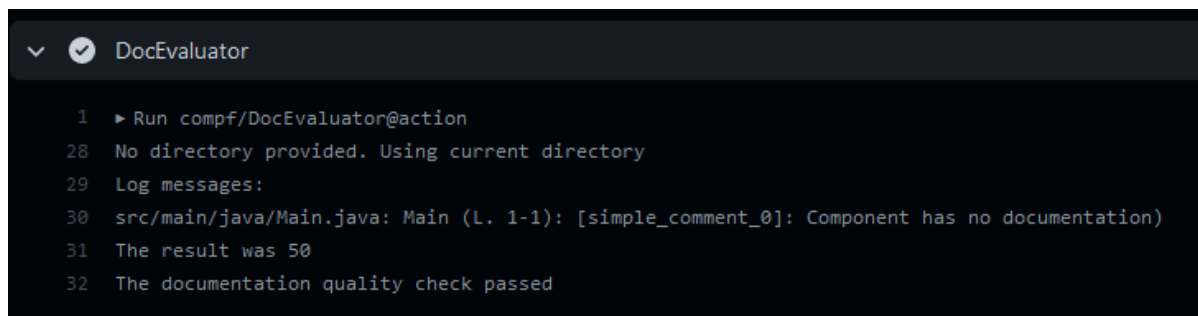
Klassenname GunningFogMetric

Beschreibung Berechnet den Gunning-Fog-Index des Kommentars und bewertet so, ob der Kommentar verständlich ist

Quellen [3, S. 71]

F Bilder des Tools

In diesem Kapitel sind zwei Bilder des *DoxEvaluators* abgedruckt, welche die zwei möglichen Ausgaben des Programms zeigen (Dokumentationsqualität ausreichend und nicht ausreichend):



```
▼ ✓ DocEvaluator

1 ▶ Run compf/DoxEvaluator@action
28 No directory provided. Using current directory
29 Log messages:
30 src/main/java/Main.java: Main (L. 1-1): [simple_comment_0]: Component has no documentation)
31 The result was 50
32 The documentation quality check passed
```

Abbildung F.1: Foto vom Tool: Dokumentationsqualität ausreichend



```
▼ ✗ DocEvaluator

1 ▶ Run compf/DoxEvaluator@action
28 No directory provided. Using current directory
29 Log messages:
30 src/main/java/Main.java: Main (L. 1-1): [simple_comment_0]: Component has no documentation)
31 src/main/java/Main.java: Main.main (L. 2-2): [simple_comment_0]: Component has no documentation)
32 The result was 0
33 /home/runner/work/_actions/compf/DoxEvaluator/action/build/index.js:35722
34     throw new Error("Threshold was not reached: The minimum value must be " + conf.absolute_threshold);
35     ^
36
37 Error: Threshold was not reached: The minimum value must be 50
```

Abbildung F.2: Foto vom Tool: Dokumentationsqualität zu schlecht

Erklärung zur selbstständigen Abfassung der Bachelorarbeit

Ich versichere, dass ich die eingereichte Bachelorarbeit selbstständig und ohne unerlaubte Hilfe verfasst habe. Anderer als der von mir angegebenen Hilfsmittel und Schriften habe ich mich nicht bedient. Alle wörtlich oder sinngemäß den Schriften anderer Autoren entnommenen Stellen habe ich kenntlich gemacht.

Osnabrück 7. April 2022

Timo Schoemaker