Konzeption, Implementation und quantitative Evaluation einer statischen Clean-Code-Bewertungsapplikation

Masterarbeit

für den Masterstudiengang

Informatik für Geistes- und Sozialwissenschaftler

Fakultät für Informatik

Professur für Medieninformatik

Eingereicht von: Maurice Eichenseer
Matrikel-Nr.: [redacted]
Datum der Einreichung: 30.10.2023
Erstgutachter: [redacted]
Zweitgutachter: [redacted]

Abstract

Refactoring wird angewandt, wenn eine Software-Inspektion Defekte im Programmcode festellt. Code-Smells sind Beispiele solcher Defekte im Programmcode. Clean-Code ist ein neuerer Ansatz, der genauso wie das Code-Smell-Konzept festlegt, wann Defekte im Programmcode vorliegen. Für Code-Smells gibt es bereits zahlreiche Code-Analyse-Tools, die die automatische Erkennung solcher Defekte ermöglicht. Die vorliegende Arbeit implementiert ein Clean-Code-Analyse-Tool für Programmcode mithilfe von statischer Code-Analyse, das Refactoring-Hinweise ausgibt. Für diesen Zweck werden ein Lexer und ein Parser zur syntaktischen Analyse eines Subsets der Programmiersprache C++ implementiert. Die Evaluation durch quantitative Datenanalyse zeigt, wie nützlich die automatische Erkennung von Clean-Code mithilfe eines statischen Code-Analyse-Tools bei der Erstellung von Programmcode mit höherer Lesbarkeit für Entwicklerinnen und Entwickler ist.

Schlüsselwörter:
Clean-Code, Code-Smells, Refactoring,
Software-Inspektion, statische Code-Analyse

Danksagung:

Dank gilt an dieser Stelle allen Korrekturleserinnen und Korrekturlesern. Darunter Arsene Eduard, Diana, Esther, Julian, Leandra, Lisa und Patrick. Ein besonderer Dank gilt insbesondere meinem Betreuer [redacted], der mich durch die lange Zeit der Ausarbeitung der vorliegenden Arbeit fortlaufend unterstützt hat.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

Akronyme

AOI	Area of Interest
AST	abstrakter Syntaxbaum
AV	abhängige Variable
BNF	Backus-Naur-Form
CC	Clean-Code
EBNF	erweiterte Backus-Naur-Form
IR	interne Repräsentation
JDK	Java Development Kit
JDT	Java Entwicklungswerkzeuge
LALR	Lookahead-LR
ML	maschinelles Lernen
OOP	objektorientierte Programmierung
SQL	Structured Query Language
TDD	testgetriebene Entwicklung
UV	unabhängige Variable
VCS	Versionskontrollsystem
VSC	Visual Studio Code

1. Einleitung

Guten Programmcode zu schreiben ist ein Anspruch, den Entwicklerinnen und Entwickler von Software seit langer Zeit haben. Was „gut“ jedoch konkret heißt, hängt von vielen unterschiedlichen Aspekten ab. Um diesem Problem zu begegnen, gibt es immer wieder neue Ansätze, dieses „gut“ zu standardisieren. Eine Perspektive auf die Anforderungen von gutem Code ist es technisch besonders eleganten Code zu schreiben (vgl. Goodliffe [2007] Vorwort xxviii ff.). Ein anderer Ansatz betrachtet Programmiercode als guten Code, welcher im Sinne der Maschine geschrieben ist, die den Code ausführt (vgl. Hyde [2006]: 1). Dabei müssen schwierige Themen wie Assemblersprache und Compilertheorie beim Coding-Prozess einbezogen werden, um überhaupt guten Code schreiben zu können (vgl. Hyde [2006]: 1 f.). Gegenüber diesen eher technischen Perspektiven besitzt Clean-Code-Ansatz den Aspekt der Lesbarkeit für andere Entwicklerinnen und Entwickler (vgl. Martin [2009]: 42). Im Vergleich zu den früheren Ansätzen von gutem Code lässt sich durch den Fokus auf die Lesbarkeit für Menschen statt Maschinen die Zugänglichkeit von Programmcode durch Clean-Code erhöhen.
Mittlerweile hat sich der Clean-Code-Ansatz etabliert. Zum einen hat die Industrie die Relevanz des Themas für sich entdeckt, da der Programmcode von Software aus der Industrie teils über einige Jahrzehnte genutzt wird (vgl. Böttcher et al. [2011]: 34). Zum anderen ist durch dieses gewachsene Interesse der Industrie Clean-Code mittlerweile vermehrt ein Teil der Ausbildung im Bereich des Software-Engineerings geworden (vgl. Böttcher et al. [2011]: 34).
Zur Verwendung von Clean-Code in der Praxis wurde eine empirische Studie unter 39 Entwicklerinnen und Entwicklern durchgeführt, die seit bis zu 20 Jahren in der Industrie arbeiten (vgl. Ljung and Gonzalez-Huerta [2022]: 303). Die Studie zeigt wie die Entwicklerinnen und Entwickler zwar mit den Clean-Code Prinzipien übereinstimmen, jedoch trotzdem chaotischen Code schreiben, auf den zu einem späteren Zeitpunkt Refactoring angewendet wird (vgl. Ljung and Gonzalez-Huerta [2022]: 298). Das heißt, der Endzustand des Clean-Codes ist nicht zwingend im

Fokus beim Programmieren und wird auf einen späteren Zeitpunkt nach dem Refactoring verschoben. Als Grund wird von den Entwicklerinnen und Entwickler die hohe Schwierigkeit des Verfassens von Clean-Code angegeben (vgl. Ljung and Gonzalez-Huerta [2022]: 312).
Da die vorliegende Arbeit zeitgleich mit einer Lehrtätigkeit als Tutor an einer universitären Fakultät für Informatik entstanden ist, ist eine große Motivation entstanden, sich mit dem Thema des guten Codes auseinanderzusetzen. Zwar ist Clean-Code Teil des Curriculums des Informatikstudiums an der genannten Fakultät, jedoch wird aus persönlicher Erfahrung als Tutor von den Studierenden kaum Clean-Code geschrieben. Das Interesse an einem späteren Refactoring ist gegenüber den Entwicklerinnen und Entwicklern in der Industrie für Studierende gering. Da nach einer Abgabe des Codes für eine Prüfungsleistung der Code üblicherweise nicht weiter verwendet wird, werden die Kapazitäten in andere Prüfungsleistungen investiert. Der Refactoring-Vorgang wird dann von den Studierenden nicht geübt. Das Refactoring während des Bearbeitens der Prüfungsleistung wäre demgegenüber auch keine gute Option. Eine empirische Studie hat diesen Ansatz mit 22 Studierenden durch einen Gruppenvergleich evaluiert (vgl. Abid et al. [2015]: 228). Die Ergebnisse zeigen, wie eine Gruppe Studierender durch Refactoring nach dem Verfassen eines Programms saubereren Code erzeugt als eine Vergleichsgruppe, die stetiges Refactoring über den gesamten Prozess des Programmierens hinweg anwendet (vgl. Abid et al. [2015]: 230). Obwohl Clean-Code ein zugänglicherer Standard als frühere Ansätze des guten Codes ist, ist es in der Lehre sehr schwierig diesen Zustand zu erreichen. Es ist für Studierende weder während des Erstellens einer Prüfungsleistung rational, ein stetiges Refactoring hin zum Clean-Code durchzuführen, noch im Anschluss der Abgabe.
Aus dieser Ausgangslage ist die Idee entstanden ein Refactoring- und Bewertungs-Tool für Clean-Code zu realisieren. Dieses Tool kann in der Lehre eingesetzt werden, um die Zugänglichkeit zu gutem Code weiter zu erhöhen. Clean-Code benötigt weniger technisches Wissen als frühere Ansätze. Mit den Hinweisen des Refactoring-Tools kann das benötigte Wissen über die Umsetzung von Clean-Code reduziert werden, was den Schwierigkeitsgrad und die Komplexität verringert. Ein stetiges Refactoring, was Entwicklerinnen und Entwicklern zu schwierig erscheint, könnte erleichtert und das Thema guter Code weiter aus dem Nischendasein befreit werden. Dazu soll die vorliegende Arbeit einen kleinen Beitrag leisten, um insbesondere die Informatik zugänglicher zu machen, indem Code öfter besser gelesen

werden kann. Durch Refactoring-Hinweise, die den Ort und die Art der Probleme beschreiben, könnte es auch für Studierende rationaler werden, ein stetiges Refactoring durchzuführen. Es kann den weiteren Programmiervorgang bis zur Abgabe der Prüfungsleistung erleichtern und ist mithilfe der Hinweise einfach durchzuführen.
Darüber hinaus ist während der Recherche zur Konzeption des Clean-Code-Analyse-Tools eine große Forschungslücke in diesem Bereich aufgefallen. Obwohl der Clean-Code-Ansatz von Martin [2009] mittlerweile viele Jahre alt ist, gibt es keine Code-Analyse-Tools zu diesem Konzept in der Literatur. Die vorliegende Arbeit möchte daher mit der Realisierung eines statischen Clean-Code-Analyse-Tools dabei helfen, einen ersten Anlauf in diese Richtung zu wagen und weitere Forschung in dem Bereich anzuregen. Dabei gilt es die Forschungsfrage klären, inwiefern ein solches Tool nützlich bei der Erkennung von unsauberem Code ist.
Der Aufbau der vorliegenden Ausarbeitung ergibt sich aus dem Titel der Arbeit. Um sich der Forschungsfrage zu widmen, müssen zuerst wichtige theoretische Grundlagen erläutert werden, was in Kapitel 2 geschieht. Daraufhin wird der aktuelle Forschungsstand in Kapitel 3 unter die Lupe genommen. In Kapitel 4 wird ein Konzept entwickelt, welches zur Umsetzung und Evaluation des statischen Clean-Code-Analyse-Tools benötigt wird. Auf Grundlage dieses Konzepts werden im gleichen Kapitel die Hypothesen zur Beantwortung der Forschungsfrage formuliert. Im Anschluss wird in Kapitel 5 die Methodik der Datenerhebung zur Überprüfung der Hypothesen beschrieben. Die Methodik umfasst sowohl die Implementation des statischen Clean-Code-Analyse-Tools, als auch die Studie zur Evaluation desselben. Im folgenden Kapitel 6 werden die Ergebnisse der Studie unter Zunahme der Daten aus dem implementierten Tool vorgestellt. Die Ergebnisse werden dann in Kapitel 7 diskutiert. Schlussendlich wird in Kapitel 8 ein Fazit zur vorliegenden Ausarbeitung gezogen.

2. Theoretische Grundlagen

Das Kapitel beginnt in Abschnitt 2.1 mit den wichtigsten Begriffen und Definitionen im Bereich der Überarbeitung von Programmcode, die für die vorliegende Arbeit von Bedeutung sind. Anschließend werden in Abschnitt 2.2 die Code-Smells konkretisiert, um das Konzept mit dem Clean-Code-Konzept in Abschnitt 4.1 im Detail vergleichen zu können. In Abschnitt 2.3 findet eine Einführung in den Compilerbau statt, die für die praktische Umsetzung in Abschnitt 5.1 von Bedeutung ist.

2.1. Wichtige Begriffe und Definitionen

Um sich eines statischen Clean-Code-Analyse-Tools zu nähern, werden einige Definition benötigt. In diesem Unterkapitel soll es daher darum gehen, wichtige Begriffe und Definitionen aus dem Gebiet der Überarbeitung von Programmcode kennenzulernen. Dabei wird zuerst der Begriff der Software-Inspektion (Unterabschnitt 2.1.1), dann der Begriff der Fagan-Inspektion (siehe Unterabschnitt 2.1.2), im Anschluss der Begriff des checklistenbasierten Code-Reviews (Unterabschnitt 2.1.3) und anschließend die statische beziehungsweise dynamische Code-Analyse (Unterabschnitt 2.1.4) erläutert. Zum Ende des Kapitels werden der Begriff des Refactorings (Unterabschnitt 2.1.5), danach der Begriff des Code-Smells (Unterabschnitt 2.1.6 und schlussendlich der Begriff des Clean-Codes (Unterabschnitt 2.1.7) ausführlich beleuchtet.

2.1.1. Software-Inspektion

Ein älterer Begriff aus dem Gebiet der Überarbeitung von Programmcode ist der Begriff der Code-Inspektion, der von Fagan [1976] eingeführt wurde. Code-Inspektionen werden als eine formale, effiziente und ökonomische Methode zur Suche nach Defekten im Software-Design und im Software-Code definiert (vgl.

(Fagan [1976]: 584). Ein Defekt ist dabei als ein Umstand definiert, bei dem ein Arbeitsprodukt nicht die Abschlusskriterien erfüllt (vgl. Fagan [2002]: 571). Zusätzlich hat die Code-Inspektion das Ziel, gefundene Defekte aus einem gegebenen Programmcode zu eliminieren (vgl. Ebenau and Strauss [1994]: 37). Damit sind sowohl Probleme im Software-Design gemeint als auch Probleme im Entwicklungsprozess, welche die Wahrscheinlichkeit zukünftiger Probleme im Design erhöhen (vgl. Fagan [2002]: 567). Mittlerweile hat sich davon ausgehend der Begriff der Software-Inspektion mit gleicher Bedeutung durchgesetzt (vgl. Ebenau and Strauss [1994]: 5). Die folgende Definition von Fagan [1976] zielt auf die geringen Kosten im Vergleich zum Nutzen einer Inspektion ab. Eine Fehlerbehandlung von Programmcodes wird teurer, je später im Entwicklungsprozess sie stattfindet (vgl. Fagan [1976]: 579). Das bedeutet, jede Suche nach Defekten beziehungsweise deren Behebung sollte so früh wie möglich im Entwicklungsprozess durchlaufen werden (vgl. Fagan [1976]: 579). Die während der Inspektion gefundenen Defekte beziehungsweise deren Überarbeitung zu einem frühen Zeitpunkt des Entwicklungsprozesses reduzieren den Zeitraum des gesamten Prozesses, wodurch die Produktivität erhöht wird (vgl. Fagan [1976]: 580). Diese Zeitersparnis gilt bereits für ein frühes Entwicklungsstadium, wird aber über den gesamten Prozess umso gravierender (vgl. Fagan [1976]: 580). Darüber hinaus haben empirische Analysen gezeigt, wie im frühen Stadium des Entwicklungsprozesses Defekte gefunden werden, die in späteren Stadien gar nicht mehr auffindbar sind (vgl. Aurum et al. [2002]: 136). Des Weiteren ist die Software-Inspektion vom Testen abzugrenzen, wobei das Testen vor oder nach der Software-Inspektion angewandt wird (vgl. Gomes et al. [2009]: 2).

2.1.2. Fagan-Inspektion

Fagan [1976] beschreibt mit der Software-Inspektion neben der formalen Definition auch einen formalisierten Prozess, bei dem eine Gruppe von Entwickelnden mit der Erstellerin beziehungsweise dem Ersteller des Programmcodes über Fehler und Probleme im Design des Codes diskutieren (vgl. Fagan [1976]: 602). Das Review-Team besteht dabei aus einer moderierenden Person, die den Inspektionsprozess koordiniert, dem Autor oder der Autorin des Code-Designs, dem Autor oder der Autorin des Codes sowie einer Person, die Tests für die Software schreibt (vgl. Fagan [1976]: 585). Diese Methode, die in der Literatur auch Fagan-Inspektion genannt wird, hat sich mittlerweile als erste formale Methode des Code-Reviews eta-

bliert (vgl. Sadowski et al. [2018]: 181). Demgegenüber beschreibt Fagan [2002] die Software-Inspektion als Methode, die unabhängig von Code-Reviews angewandt wird und sich, anders als Code-Reviews, lediglich auf die Suche nach Defekten im Design beschränkt (vgl. Fagan [2002]: 571 f.). Das heißt, in weniger standardisierten Code-Reviews könnte also beispielsweise auch über Funktionalitäten des Programmcodes diskutiert werden. Dadurch entsteht eine Unschärfe des Begriffs der Software-Inspektion, da die Software-Inspektion sowohl eine konkrete Form des Code-Reviews darstellt, als auch synonym für den Oberbegriff des Code-Reviews verwendet wird. In der vorliegenden Arbeit wird im Folgenden für die konkrete Methode der Software-Inspektion der Begriff Fagan-Inspektion verwendet, während Software-Inspektion den Oberbegriff von Inspektionen meint und synonym zum Begriff Code-Review verwendet wird. Trotz dieser Unschärfe hat sich die Software-Inspektion als effektive Methode zur Qualitätssteigerung von Programmcode etabliert, die sogar das Testen von Software mithilfe von Unit-Tests teilweise ersetzen kann (vgl. Fagan [2002]: 569). Mit Unit-Tests sind dabei Einheiten eines Programmcodes gemeint, die andere Einheiten desselben Codes aufrufen und ein erwartetes Ergebnis der Ausführung mit dem tatsächlichen Ergebnis vergleichen (vgl. Osherove [2010]: 4).

2.1.3. Checklistenbasiertes Code-Review

Mittlerweile haben sich neben der Fagan-Inspektion viele weitere Arten von Software-Inspektionen etabliert, die meistens lediglich eine Teilmenge der Operationen der Fagan-Inspektion darstellen beziehungsweise umsetzen (vgl. Aurum et al. [2002]: 140). Ein Nachteil des Code-Reviews ist die Voraussetzung eines erfahrenen und trainierten Teams mit ausprägten Lese- und analytischen Fähigkeiten (vgl. Chong et al. [2021]: 22). Diese Fähigkeiten müssen zusammenkommen, um die Defekte im Programmcode zu erkennen (vgl. Louridas [2006]: 58).
Eine alternative Variante des Code-Reviews, die dieses Problem adressiert, ist das checklistenbasierte Code-Review (vgl. Aurum et al. [2002]: 140), welches zur Evaluation des Clean-Code-Tools in der vorliegenden Arbeit angewendet wird (siehe Kapitel 5). Diese Form des Code-Reviews ist abgeleitet von der Vorbereitungsphase der Fagan-Inspektion. Bei dieser Reviewform bereitet eine am Review teilnehmende Person Checklisten äquivalent zu denen der Fagan-Inspektion vor (vgl. Chong et al. [2021]: 22). In dieser Phase studieren die Teilnehmenden bereits im Vorfeld Checklisten, auf denen Hinweise beziehungsweise Fragen zu finden sind, nach wel-

chen Defekten sie im Programmcode Ausschau halten sollen (vgl. Fagan [1976]: 587). Dieses Verfahren unterscheidet sich nicht gegenüber dem checklistenbasierten Code-Review, außer hinsichtlich Herauslösung aus den sonstigen Operationen der Fagan-Inspektion und einer damit einhergehenden weiteren Formalisierung. Das checklistenbasierte Code-Review gehört zu den strukturierten Code-Review-Techniken, was im Gegensatz zu leichtgewichtigen Code-Review-Methoden steht, wie beispielsweise dem ad-hoc-Code-Review, das ohne Checklisten auskommt (vgl. Chong et al. [2021]: 22). „Strukturiert“ bezeichnet hier die klare Definition der spezifischen Schritte des Reviewprozesses (vgl. Shull et al. [2000]: 74).
Eine Checkliste wird als Menge von Fragen, die zu den potenziellen Defekten in Verbindung stehen, die bei einem Code-Review auftauchen könnten, definiert (vgl. Chong et al. [2021]: 21). Diese ist eine Auflistung von Interpretationen von Regeln, welche die Reviewenden beim Erkennen von Defekten unterstützt (vgl. Gilb et al. [1993]: 44). Regeln sind in diesem Kontext die Instruktionen an Autorinnen und Autoren eines Programmcodes über notwendige Praktiken, sodass sie eine Zusammenfassung von Best-Practices beim Programmieren darstellen (vgl. Gilb et al. [1993]: 61). Checklisten sollten dabei Fragen enthalten, welche die im Vorfeld formulierten Coding-Regeln in weniger formalisierte Fragen übersetzen (vgl. Gilb et al. [1993]: 60), wobei keine neuen Regeln geschaffen werden sollten (vgl. Gilb et al. [1993]: 56). Das heißt, es dürfen keine Fragen gestellt werden, für die im Vorfeld keine Regel aufgestellt worden ist, sodass jede Frage direkt auf eine Regel zurückzuführen ist (vgl. Gilb et al. [1993]: 56 f.). Dadurch sind Checklisten eine Möglichkeit Wissen über diese Regeln an die reviewende Person weiterzugeben (vgl. Gilb et al. [1993]: 60). Der Sinn dieser Methode ist die Verantwortung der Personen, die das Review durchführen, mithilfe der Checkliste festzulegen (vgl. Aurum et al. [2002]: 144) und so den Inspektionsprozess klarer zu machen (vgl. Shull et al. [2000]: 74). Infolgedessen wird der Ablauf des Code-Reviews stärker formalisiert (vgl. Gilb et al. [1993]: 58). Gute Checklistenfragen sind dabei nicht nur auf eine Coding-Regel zurückzuführen, sondern deren negative Beantwortung bedeutet unmittelbar einen Defekt gefunden zu haben (vgl. Gilb et al. [1993]: 56). Für ein Programmcodedokument sollten außerdem während des Code-Reviews niemals mehr als 25 Fragen gestellt werden (vgl. Gilb et al. [1993]: 56).
Die Stärke dieser Methode ist einen Fokus auf die wichtigen Defekte beim Review-Prozess legen zu können (vgl. Chong et al. [2021]: 22), auf welche die Reviewenden ohne die Checklistenfragen eventuell nicht geschaut hätten (vgl. Gilb et al. [1993]:

58). Darüber hinaus ist die Methode leicht an ein bestimmtes Projekt anpassbar (vgl. Shull et al. [2000]: 74). Eine einfache Kopie der Checkliste eines anderen Projekts führt bei dieser Methode nicht zu guten Ergebnissen, wobei Checklisten empirisch die Anzahl der gefundenen Probleme eines Teams bei einem Code-Review erhöhen (vgl. Gilb et al. [1993]: 58). Des Weiteren kann die Erstellung von Checklisten zur besseren Ausarbeitung der Regeln führen, auf denen sie basieren (vgl. Gilb et al. [1993]: 58). Eine Schwäche der Methode ist demgegenüber die höhere Wahrscheinlichkeit von übersehenen Defekten bei unerfahrenen Reviewenden, sollten die Defekte nicht auf der Checkliste vorkommen (vgl. Chong et al. [2021]: 22), deren Platz begrenzt ist (vgl. Aurum et al. [2002]: 144). McMeekin et al. [2009] haben in ihrer Studie gezeigt, wie eine Checkliste und die damit einhergehende klare Struktur des Code-Reviews Studierenden mit wenig Review-Erfahrung dabei hilft möglichst viele Defekte im Code zu finden (vgl. McMeekin et al. [2009]: 236 f.). Studierende sind dabei Entwicklerinnen und Entwickler mit tendenziell sowohl wenig Erfahrung im Programmieren als auch in der Inspektion von Code (vgl. McMeekin et al. [2009]: 236 f.), wobei die Checkliste dabei hilft, eine kognitive Überlastung der Person, die das Review durchführt, während des Code-Reviews zu verhindern (vgl. Chong et al. [2021]: 20).

2.1.4. Statische vs. dynamische Code-Analyse-Tools

Code-Reviews haben sich als die beste Variante herausgestellt Defekte zu eliminieren (vgl. Bardas [2010]: 99). Code-Reviews benötigen jedoch sehr erfahrene Reviewende, um alle Defekte zu finden (vgl. Bardas [2010]: 99). Dabei ist es fast unmöglich, die gesamte Code-Basis größerer Projekte zu reviewen (vgl. Bardas [2010]: 99). Bisher sind das checklistenbasierte Code-Review und das asynchron durchgeführte moderne Code-Review genannt worden, um diesem Problem zu begegnen. Ein weiterer Ansatz sind Tools, die eine automatische Software-Inspektion durchführen. Dabei wird die Suche nach Defekten mithilfe von automatischen Softwaretools vollzogen, sodass Menschen dieser Vorgang vollständig abgenommen wird (vgl. Gomes et al. [2009]: 2). Neben dem häufigeren Begriff des statischen Code-Analyse-Tools werden solche Tools in der Literatur auch als Software-Inspektions-Tools bezeichnet (vgl. Van Emden and Moonen [2002]: 100). In der vorliegenden Arbeit wird jedoch ausschließlich der Begriff des statischen Code-Analyse-Tool verwendet. Dabei bleibt jede statische Code-Analyse prinzipiell komplett ohne Computerunterstützung durchführbar (vgl. Liggesmeyer [2002]:

40). Durch den automatischen Einsatz bieten statische Code-Analyse-Tools jedoch den Vorteil der Früherkennung vieler häufiger Defekte, was bereits vor dem Testen oder einem späteren Code-Review einfach umzusetzen ist, wodurch die Defekte in einem frühen Entwicklungsstadium entfernt werden können (vgl. Louridas [2006]: 58). Der Begriff der statischen Code-Analyse meint dabei die Analyse der statischen Struktur und der Elemente eines Programms, ohne es dabei auszuführen (vgl. Prahofer et al. [2012]: 1). Sie erstellen ein Modell des Programmcodes, welches sie auf bestimmte Eigenschaften hin untersuchen (vgl. Bardas [2010]: 100). Auf der Grundlage der Strukturinformationen aus dem Modell analysieren sie die häufigsten Fehler und Defekte, wobei sie sich die Tendenz von Entwickelnden zunutze machen, die gleichen Fehler und Defekte immer wieder zu reproduzieren (vgl. Louridas [2006]: 58). Üblicherweise basiert die Extraktion der Strukturinformationen dabei direkt auf dem Programmcode oder einer einfachen Repräsentation davon (vgl. Prahofer et al. [2012]: 1). Dabei wird in der Regel ein sogenannter Parser eingesetzt, der einen Quelltext als Input bekommt, wobei als Output ein abstrakter Syntaxbaum (AST) zur Abstraktion des Quellcodes erstellt wird (vgl. Gosain and Sharma [2015]: 583). Ein Parser ist dabei derjenige Teil des Programmübersetzers, der die Grammatik des Quellcodes übersetzt, sodass der AST ein Nebenprodukt dieser Syntaxanalyse ist (vgl. Aho et al. [2008]: 233). Statische Code-Analyse-Tools können keine exakten Aussagen über die Korrektheit von Code tätigen (vgl. Liggesmeyer [2002]: 40). Sie können jedoch Hinweise auf das fehlerhafte Verhalten von Code bereitstellen (vgl. Or-Meir et al. [2020]: 2). Nach ihrem Aufkommen wurden statische Code-Analyse-Tools eher zur Fehlererkennung eingesetzt (vgl. Prahofer et al. [2012]: 1). Demgegenüber haben sie sich heutzutage als eine Qualitätsbeurteilungs- und Verbesserungstechnologie etabliert, wobei sie auch vor oder parallel zu einem Code-Review eingesetzt werden können (vgl. Prahofer et al. [2012]: 1). Ein großes Problem statischer Code-Analyse-Tools sind sogenannte falsch Positive, sowie falsch Negative (vgl. Bardas [2010]: 100). Das Modell, welches aus dem Programmcode abgeleitet wird, stellt lediglich eine Annäherung dar, sodass vom Tool Fehler und Defekte im Programmcode übersehen werden können (falsch Negative) oder unproblematischer Code als problematisch interpretiert werden kann (falsch Positive) (vgl. Bardas [2010]: 100). Folglich sollte beim Einsatz statischer Code-Analyse-Tools das Wissen um diese Problematik in möglichen späteren Code-Reviews bedacht werden.

Im Unterschied zu statischen Code-Analyse-Tools führen dynamische Code-Analyse-

Tools den Programmcode aus und suchen nach Inkonsistenzen in den Ergebnissen des Kompilierungsprozesses (vgl. Gomes et al. [2009]: 2), wodurch sie weitere Probleme der statischen Code-Analyse aufgreifen. Die statische Code-Analyse unternimmt den Versuch Eigenschaften des Programmcodes für alle möglichen Ausführungen des Programms abzuleiten und zu generalisieren (vgl. Ball [1999]: 216). Infolge der Extraktion von Programmcode findet jedoch von Beginn der Analyse an ein Informationsverlust statt (vgl. Ball [1999]: 217). Werden Eigenschaften des Codes während des Ausführens verändert, kann sie die statische Code-Analyse nicht beobachten (vgl. Or-Meir et al. [2020]: 3). Problematisch ist hier insbesondere die sogenannte dynamische Bindung, die einen Bezeichner beziehungsweise eine Variable mit ihrem aktuellen Ausführungskontext assoziiert, wobei beispielsweise eine dynamische Variable nur während des Ausführens des Programms mit einem Wert verknüpft ist (vgl. Kiselyov et al. [2006]: 491). Neben der dynamischen Bindung hat die statische Code-Analyse ebenfalls Probleme bei der Erkennung von Vererbung sowie von ungenutztem Programmcode (vgl. Arisholm et al. [2004]: 491). Die genannten Probleme werden demgegenüber von der dynamischen Code-Analyse adressiert, wobei sie im Unterschied zur statischen Code-Analyse nicht den Programmcode selbst analysiert (vgl. Or-Meir et al. [2020]: 3). Dabei leitet die dynamische Code-Analyse Eigenschaften eines laufenden Programms ab, die lediglich für eine (oder mehrere) Ausführungen des Programms gültig sind (vgl. Ball [1999]: 216), das heißt sich nicht auf alle möglichen Ausführungen des Programms beziehen müssen. Des Weiteren kann die dynamische Code-Analyse die Verletzung bestimmter Eigenschaften nachweisen, was jedoch nicht mit dem Beweis für die Erfüllung bestimmter Eigenschaften zu verwechseln ist (vgl. Ball [1999]: 216). Darüber hinaus hat die dynamische Code-Analyse Zugriff auf Informationen, die erst zur Laufzeit zur Verfügung stehen, wodurch mehr Informationen zur Verfügung stehen, die zur Lösung eines bestimmten Problems genutzt werden können (vgl. Ball [1999]: 216). Dadurch sind die erstellten Strukturinformationen für die Menge von Programmausführungen präziser (vgl. Ball [1999]: 217). Die Abstraktion der statischen Code-Analyse wird bei der dynamischen Code-Analyse nicht zwingend benötigt (vgl. Ball [1999]: 217), wobei sie trotzdem eine bessere Genauigkeit aufweist (vgl. Arisholm et al. [2004]: 491). Darüber hinaus kann die dynamische Code-Analyse semantische Abhängigkeiten zwischen Programmentitäten im Code analysieren, da sie Ausführungspfade im Code bis zum Ende geht, während eine hohe Distanz im Code zwischen Entitäten für statische Code-Analyse-Tools pro-

blematisch ist (vgl. Ball [1999]: 217). Ein entscheidender Nachteil der dynamischen Code-Analyse ist jedoch der hohe Aufwand durch die Notwendigkeit der Erstellung vieler verschiedener Programmabläufe, die verglichen werden (vgl. Rohatgi et al. [2008]: 236). Dabei müssen viele unterschiedliche Merkmale beobachtet werden, um lediglich ein einziges Merkmal von Interesse zu identifizieren (vgl. Rohatgi et al. [2008]: 236).

2.1.5. Refactoring

Ein weiterer wichtiger Begriff im Kontext der Software-Inspektion ist der Begriff des Refactorings. William [1992] definiert Refactorings als Menge von Programmrestrukturierungsoperationen, die das Design, die Erweiterbarkeit und die Wiederverwendbarkeit von objektorientierten Anwendungsframeworks verbessern, die dabei jedoch gleichzeitig das Verhalten des Programms vor den Refactorings erhalten (vgl. William [1992]: Vorwort iii). Refactoring ist also eine Art der Behebung von Programmdefekten und somit zeitlich nach der Software-Inspektion angesiedelt. Brown [1998] erweitert die Definition um eine Verbesserung der Wartbarkeit des Codes, die durch Refactoring erzielt wird, wobei die Korrektheit des Codes intakt bleibt. Probleme im Code-Design werden also im Sinne der Software-Inspekton als Defekte angesehen, auch wenn der Softwarecode formal korrekt ist. Daraus folgt die Zielsetzung des Refactorings, für Menschen lesbaren Code zu erstellen, statt sich auf die Maschinenlesbarkeit zu beschränken (vgl. Fowler [2019]: 10). Das Verhalten der Software ändert sich beim Refactoring nicht, wobei die Qualität der internen Struktur der Software ansteigt (vgl. Fowler [2019]: Vorwort XIV). Das heißt, beim Refactoring geht es darum, die interne Struktur von Programmcode zu verbessern, nachdem er geschrieben wurde (vgl. Fowler [2019]: Vorwort XIV).

2.1.6. Code-Smells

Der Begriff des Refactorings zeigt zwar Mechanismen auf, wie im Anschluss einer Software-Inspektion Probleme im Cod-Design eliminiert werden können, sagt aber nicht, wann diese genau angewendet werden sollten. Um diese Frage zu beantworten haben Beck et al. [1999] den Begriff des Code-Smells (deutsch: übel riechender Code) eingeführt (vgl. Beck et al. [1999]: 75). Ein Code-Smell beschreibt dabei problematische Strukturen im Programmcode, die auf die Möglichkeit des Refactorings hindeuten (vgl. Beck et al. [1999]: 71). Das heißt, es handelt sich um

Defekte, die während der Software-Inspektion erkannt werden können. Dabei geben Sie jedoch keine präzisen Kriterien an, wann genau ein Refactoring verwendet werden sollte, sondern beschreiben lediglich Indikatoren, wann höchstwahrscheinlich Schwierigkeiten im Code bestehen, die durch Refactorings gelöst werden könnten (vgl. Beck et al. [1999]: 75). Das heißt, Code-Smells sind nicht präzise, sondern subjektiv (vgl. Van Emden and Moonen [2002]: 2 f.). Wichtig zu erwähnen ist der Fokus des Begriffs auf Fehler im Design von Programmcode objektorientierter Programmiersprachen (vgl Beck et al. [1999]: 77). Beck et al. [1999] haben einen Katalog mit den existierenden Code-Smells veröffentlicht (siehe Kapitel 2.2). Dabei geben Fowler and Beck [2007] an, welche Refactorings bei welchen Code-Smells infrage kommen. Code-Smells werden in der Literatur als Symptome für die Präsenz von Design-Smells beschrieben (vgl. Moha et al. [2010]: 20). Mit Design-Smells sind sogenannte AntiPatterns gemeint. AntiPattern ist ein Begriff, den Koenig [1998] eingeführt hat und als etwas definiert, was wie eine Lösung eines Problems aussieht, jedoch keine Lösung darstellt (vgl. Koenig [1998]: 387). Brown [1998] hat den Begriff bekannt gemacht und AntiPattern als eine Methode zur effizienten Abbildung einer allgemeinen Situation zu einer spezifischen Klasse von Lösungen definiert (vgl. Brown [1998]: 8). Der Begriff des AntiPatterns ähnelt dem Begriff des Code-Smells sehr, da sein Ziel ebenfalls ist, Situationen und Ansätze zu beschreiben, wie und wann Software-Refactoring angewendet werden sollte (vgl. Brown [1998]: 68 f.). Da der Begriff des AntiPatterns jedoch nicht so stark mit dem Begriff des Clean-Codes verwoben ist, wird in der vorliegenden Arbeit nicht weiter darauf eingegangen.

2.1.7. Clean-Code

Der mit dem Begriff des Code-Smells eng verwobene Begriff ist der Begriff des Clean-Codes (deutsch: sauberer Code). Der Begriff wurde von Martin [2009] eingeführt, der sich ebenfalls stark auf den Begriff der Code-Smells von Beck et al. [1999] bezieht (vgl. Martin [2009]: 337). Es gibt dabei keine strikte Definition des Begriffs (vgl. Anaya [2018]: Kap. 1.1). Bei dem Begriff geht es darum, einen Zustand des Programmcodes zu beschreiben, bei dem während eines Code-Reviews möglichst wenig Probleme beim Verständnis des Codes auftreten (vgl. Martin [2009]: 21). Um sich einer Definition von Clean-Code zu nähern, fragt Martin [2009] sieben verschiedene populäre Entwickler nach ihren Vorstellungen von sauberem Code (vgl. Martin [2009]: 32 ff.). Dabei werden, wie beim Code-Smell-

Konzept, keine eindeutigen Kriterien genannt, wann Clean-Code genau vorliegt. Im Sinne der Software-Inspektion geht es bei dem Begriff um die sinkende Produktivität über einen langen Zeitraum eines Softwareprojekts, insofern unsauberer Code geschrieben wird, da die Erweiterung von unsauberem Code zu mehr unsauberem Code führt (vgl. Martin [2009]: 29 f.). Die Anforderungen für Clean-Code, die sich aus den Zitaten der populären Entwickler ableiten lassen, sind Lesbarkeit, Testbarkeit und Erweiterbarkeit (beziehungsweise Wartbarkeit) (vgl. Martin [2009]: 32 ff.). Wie bei den Zielen des Refactorings meint die Lesbarkeit beim Clean-Code dabei nicht die Lesbarkeit des Codes für Maschinen, die bei kompilierbarem Code bereits sichergestellt ist, sondern die Lesbarkeit für andere Entwickelnde (vgl. Anaya [2018]: Kap. 1.1). Dabei geht es vor allem darum, sehr simplen und wenig komplexen Code zu schreiben (vgl. Mayer [2022]: Kap. 4.1). Zu beachten ist außerdem der Umstand der kontextabhängigen Definitionen von Lesbarkeit, da diese abhängig von Coding-Konventionen, beispielsweise in der gegebenen Programmiersprache ist (vgl. Sadowski et al. [2018]: 188). Testbarkeit meint im Kontext von Clean-Code einen Programmcode, der mithilfe von Unit-Tests im Sinne von testgetriebene Entwicklung (TDD) testbar ist (vgl. Martin [2009]: 159). Das Ziel von TDD ist dabei sauberer Code, der funktioniert (vgl. Beck [2003]: Vorwort ix). TDD ist ein Programmierstil, bei dem neuer, nicht-duplizierter Code nur infolge des Scheiterns zuvor selbst erstellter, automatisierter Tests geschrieben wird, bis die Tests erfolgreich abgeschlossen werden (vgl. Beck [2003]: Vorwort ix). Die Erweiterbarkeit beziehungsweise Wartbarkeit von Code bezieht sich auf die einfache Änderung von Konfigurationen, das Hinzufügen neuer Funktionalität und der Korrektur von Fehlern (vgl. Post [2021]: 84). Das Ziel des Konzepts ist es außerdem, die Erstellung und die Weiterentwicklung von Software besser zu verzahnen (vgl. Lampe [2020]: 55). Der Begriff Clean-Code wird in der Industrie häufig mit der Abstinenz von Code-Smells gleichgesetzt, wobei der Code nach dem Refactoring der Code-Smells zu Clean-Code geworden ist (vgl. Alls [2020]: Kap. 13). Dieser Umstand trägt neben der fehlenden exakten Definition weiter zur Unschärfe des Begriffs bei, da die beiden Begriffe nicht deckungsgleich sind, wie in Abschnitt 4.1 zu sehen sein wird. Trotzdem kann Clean-Code durch das Refactoring von unsauberem Code erreicht werden (vgl. Martin [2009]: 196). Implizit gibt der Begriff also ähnlich zu den Code-Smells an, wann ein Refactoring nötig ist. Martin [2009] definiert eine umfassende Auflistung von Heuristiken, die dabei helfen sollen das Ideal des Clean-Codes zu erreichen, auf die ebenfalls in 4.1 detailliert eingegan-

gen wird. Auch wenn Clean-Code nicht an eine Programmiersprache oder eine Programmiermethode gebunden ist (vgl. Post [2021]: 56), so liegt der Fokus in den genannten Heuristiken, wie bei den Code-Smells, auf der objektorientierten Programmierung.

2.2. Code-Smell-Heuristiken im Detail

Wie bereits in Unterabschnitt 2.1.6 dargelegt, beschreiben Code-Smells Indizien, die auf ein nötiges Refactoring hinweisen. Um den aktuellen Forschungsstand von Code-Analyse-Tools zur Analyse von Defekten im Programmcode nachvollziehen zu können, ist Wissen über wichtige Code Smells nötig, die in vielen modernen Tools den Fokus der Analyse von Defekten darstellen. In diesem Abschnitt werden dabei zuerst variablenspezifische, dann methodenspezifische, gefolgt von klassenspezifischen Code-Smells erläutert. Im Anschluss daran werden Code-Smells, die mehrere Arten von Entitäten betreffen, sowie schlussendlich allgemeinere Code-Smells, beschrieben. In Tabelle 2.1 findet eine übersichtliche Erläuterung einiger wichtiger Code-Smells statt. Dabei wird links der Name des konkreten Code-Smells angegeben, während in der mittleren Spalte eine kurze Erläuterung folgt, die in der rechten Spalte um die Referenz im jeweiligen Werk (siehe Abbildungsunterschrift) ergänzt wird. Die weiteren Tabellen, die im Folgenden genannt werden, besitzen den gleichen Aufbau.

Code-Smell	**Beschreibung/Anweisung**	**Ref.**
Duplizierter Code	Dieselbe Code-Struktur existiert an mehreren Orten im Programmcode.	Beck et al. [1999]: 76
Lange Methode	Es gibt keine Möglichkeit, nur mithilfe eines Namens und ohne Kommentare, genau zu beschreiben, was die Funktion tut.	Beck et al. [1999]: 76f
Divergierende Änderungen	Wird eine Klasse regelmäßig verändert, führt eine Veränderung immer zu notwendigen Anpassungen an den gleichen Entitäten.	Beck et al. [1999]: 79
Schrotkugeln herausoperieren	Immer wenn eine kleine Änderung durchgeführt wird, müssen viele kleine Veränderungen bei vielen Klassen durchgeführt werden.	Beck et al. [1999]: 80
Neid	Eine Methode ruft mehr Methoden und Daten von anderen Klassen ab, als von der eigenen Klasse.	Beck et al. [1999]: 80f

Code-Smell	Beschreibung/Anweisung	Ref.
Parallele Verwaltungs-hierarchien	Immer wenn eine neue Kindklasse einer Klasse gebildet wird, muss auch eine neue Kindklasse einer anderen Klasse gebildet werden.	Beck et al. [1999]: 83
Datenklassen	Eine Klasse besitzt ausschließlich Instanzvariablen, Konstruktoren, sowie Getter- und Settermethoden.	Beck et al. [1999]: 86f
Kommentare	Werden Kommentare benötigt, um zu erklären, wie ein Code-Abschnitt funktioniert, sollte er refaktorisiert werden.	Beck et al. [1999]: 87f
Gottklasse	Wie "Große Klasse", die zu viele Funktionalitäten selbst bereitstellt, anstatt sie an andere Klassen zu delegieren, aber deren Daten exzessiv nutzt.	Lanza and Marinescu [2006]: 80
Toter Code	Code, der aktuell nicht ausgeführt wird, sollte entfernt werden.	Mäntylä and Lassenius [2007]: 407

Tabelle 2.1.: Code-Smells Auswahl

Beck et al. [1999] beschreiben 22 Code-Smells (siehe Tabelle A.1), die mittlerweile um weitere Code-Smells aus der Literatur ergänzt worden sind (siehe Tabelle A.2, Tabelle A.3, Tabelle A.4, Tabelle A.5 und Tabelle A.6). Dabei gibt es einige wenige variablenspezifische Code-Smells. Ein Beispiel für solch einen Code-Smell ist die magische Zahl, bei der numerische Literale nicht in Konstanten mit einem aussagekräftigen Namen gespeichert werden (vgl. Drozdz et al. [2006]: 387). Ein weiterer variablenbezogener Code-Smell ist der Typecast, der beim Kompilieren nicht erkannt wird und deswegen häufig zu Fehlern führt, weswegen er vermieden werden sollte (vgl. Van Emden and Moonen [2002]: 101). Ein Typecast kann dabei durch implizite Typkonvertierung vonstatten gehen, beispielsweise wenn ein kleinerer Zahlenbereich in einen größeren Zahlenbereich abgebildet wird (vgl. Ratz et al. [2018]: 64). Es gibt jedoch auch die explizite Typkonvertierung, bei dem der beabsichtigte Zieldatentyp in runden Klammern vor die jeweilige Variable oder den Ausdruck geschrieben wird (vgl. Ratz et al. [2018]: 65). Hierbei beziehen sich Van Emden and Moonen [2002] mit dem Code-Smell des Typecasts auf die explizite Typkonvertierung.
Des Weiteren gibt es Code-Smells, die sich auf Methoden beziehen. Ein solcher Code-Smell ist die lange Methode (vgl. Beck et al. [1999]: 76). Eine längere Methode sollte dabei in ihre Funktionalitäten, das heißt auf mehrere kleinere Methoden,

aufgeteilt werden (vgl. Beck et al. [1999]: 76 f.). Es lässt sich für die Methode kein passender Name finden, der die Intention der Methode zusammenfasst (vgl. Beck et al. [1999]: 77). Dann ist die Methode nach diesem Code-Smell zu groß (vgl. Beck et al. [1999]: 77). Ein weiterer methodenbezogener Code-Smell ist die lange Parameterliste (vgl. Beck et al. [1999]: 78). Dieser Code-Smell beschreibt den Nachteil von vielen Übergabeparametern bei Methoden (vgl. Beck et al. [1999]: 78 f.). Bei zu vielen Parametern sollten einige Argumente in Klassen eingekapselt sowie Flag-Argumente generell weggelassen werden (vgl. Beck et al. [1999]: 78 f.). Des Weiteren gibt es einen Code-Smell namens Neid (vgl. Beck et al. [1999]: 80). Neid bezeichnet die fehlende Schaffung sogenannter Zonen im Programmcode, bei denen innerhalb der Zonen eine große Interaktion zwischen den Entitäten der jeweiligen Zone stattfinden soll, während für die Zonen untereinander möglichst wenig Kohäsion bevorzugt wird (vgl. Beck et al. [1999]: 80 f.). Methoden einer Klasse, die mehr Interaktion mit anderen Klassen besitzen als mit der eigenen Klasse, erfüllen diesen Anspruch nicht (vgl. Beck et al. [1999]: 80 f.). Ähnlich dazu verhält sich die intensive Kopplung, die vorliegt, wenn eine Methode äußerst viele Methodenauf- oder Datenabrufe verwendet (vgl. Lanza and Marinescu [2006]: 120 f.). Diese Methodenauf- oder Datenabrufe gehören dabei zu sehr wenigen Klassen außerhalb der eigenen Klasse (vgl. Lanza and Marinescu [2006]: 120 f.). Im Unterschied dazu gibt es die verstreute Kopplung, bei der eine Methode sehr viele Methoden auf-, oder Daten abruft, die dabei auf sehr viele verschiedene Klassen außerhalb der eigenen Klasse verteilt sind (vgl. Lanza and Marinescu [2006]: 127). Ein weiterer methodenbezogener Code-Smell sind die Switch-Befehle (vgl. Beck et al. [1999]: 82). Dieser Code-Smell tritt auf, insofern Case-Anweisungen nicht lediglich aus dem Aufruf einer geerbten Methode bestehen (vgl. Beck et al. [1999]: 82). Um Switch-Anweisungen übersichtlicher zu gestalten, besteht die Idee darin, sie mithilfe von geerbten Methoden aus der Elternklasse zu minimieren (vgl. Beck et al. [1999]: 82). Ein weiterer methodenbezogener Code-Smells ist die Gehirnmethode (vgl. Lanza and Marinescu [2006]: 92 ff.). Eine Gehirnmethode ist dabei eine Methode, die zuerst als kleinere Methode implementiert wird, zu der aber im Laufe der Zeit immer neue Funktionalitäten hinzugefügt werden, sodass sie schlussendlich die Funktionalitäten einer ganzen Klasse zentralisieren (vgl. Lanza and Marinescu [2006]: 92).

Neben den methodenbezogenen Code-Smells gibt es ebenfalls Code-Smells, die sich auf Klassen beziehen. Ein solcher Code-Smell ist beispielsweise der Vermittler (vgl.

Beck et al. [1999]: 85). Bei diesem Code-Smell ruft eine Klasse in der überwiegenden Anzahl der eigenen Methoden eine konkrete andere Klasse auf und sollte folglich in diese andere Klasse integriert werden, statt als eigenständige Klasse fortzubestehen (vgl. Beck et al. [1999]: 85). Ein weiterer Code-Smell in Bezug auf Klassen ist der Datenklumpen, bei dem Daten immer wieder gemeinsam im Programmcode verwendet werden, ohne die Daten durch Einkapselung in einer Klasse in Beziehung zueinander zu setzen (vgl. Beck et al. [1999]: 81). Hinzukommt der Code-Smell der Datenklassen, wobei Klassen lediglich Attribute sowie Getter- und Settermethoden bieten, ohne weitere Funktionalitäten bereitzustellen (vgl. Beck et al. [1999]: 86f.).
Ein weiterer Code-Smell ist die große Klasse, bei der die Klasse zu viele Daten in Form von Attributen besitzt, was duplizierten Code impliziert (vgl. Beck et al. [1999]: 78). Eine Erweiterung der großen Klasse ist die Gottklasse, bei der zusätzlich zu vielen Attributen ein starker Zugriff auf Daten von weniger komplexen Klassen sowie wenig Kohäsion zwischen den Methoden der Klasse existiert (vgl. Lanza and Marinescu [2006]: 80). Ähnlich zur Gottklasse verhält sich die Gehirnklasse, die neben vielen Attributen auch eine oder mehrere Gehirnmethoden besitzt, wobei noch weniger Kohäsion zwischen den Methoden der Klasse besteht und weniger exzessiv Daten simplerer Klassen genutzt werden (vgl. Lanza and Marinescu [2006]: 97). Ein wichtiger Code-Smell bezüglich Vererbung heißt ausgeschlagenes Erbe, bei der eine Klasse fast keine Methoden beziehungsweise Daten der Elternklasse nutzt, sodass die Hierarchie falsch angelegt ist (vgl. Beck et al. [1999]: 87). Ähnlich dazu verhält sich der Traditionsbrecher, bei dem eine Kindklasse viele Funktionalitäten anbietet, die mit den Funktionalitäten der Elternklasse nichts zu tun haben (vgl. Lanza and Marinescu [2006]: 152). Ein weiter Code-Smell sind die parallelen Verwaltungshierarchien, bei denen immer wenn eine Kindklasse einer Klasse angelegt wird, ebenfalls die Kindklasse einer anderen Klasse angelegt werden muss (vgl. Beck et al. [1999]: 83).
Neben diesen klassenbezogenen Code-Smells gibt es noch Code-Smells, die mehrere Arten von Entitäten betreffen. Ein Beispiel ist der Code-Smell divergierende Änderungen (vgl. Beck et al. [1999]: 79). Dieser Code-Smell liegt vor, wenn jedes Mal, sofern eine kleine Änderung an einer Methode oder einer Klasse vollzogen wird, eine Reihe weiterer Entitäten verändert werden muss, was auf eine weitere nötige Aufspaltung der Methode oder Klasse hindeutet (vgl. Beck et al. [1999]: 79). Das Gegenteil von divergierenden Änderungen ist der Code-Smell Schrot-

kugeln herausoperieren, bei dem jede kleine implementierte Änderung sehr viele kleine Änderungen bei vielen verschiedenen Klassen nach sich zieht (vgl. Beck et al. [1999]: 80). Ein weiterer Code-Smell dieser Kategorie ist das faule Element, das gelöscht werden sollte (vgl. Beck et al. [1999]: 83). Hier gibt es keine klare Definition, wobei beispielsweise Klassen gemeint sind, die nach dem Refactoring irrelevant geworden sind oder Methoden, die ausschließlich Aufgaben an andere Methoden delegieren (vgl. Beck et al. [1999]: 83).
Ein Code-Smell der sowohl Variablen, als auch Methoden betrifft, ist die Neigung zu elementaren Typen (vgl. Beck et al. [1999]: 81 f.). Dabei sollen primitive Datentypen, wann immer es sinnvoll ist, eingekapselt und durch Objekte ersetzt werden, um sie in Datentypen mit einem aussagekräftigen Namen zu halten (vgl. Beck et al. [1999]: 81 f.). Ein weiterer entitätenübergreifender Code-Smell heißt Nachrichtenketten, wobei Methodenaufrufe und Zugriffe auf Attribute aneinandergekettet werden, um eine bestimmte Operation auszuführen (vgl. Beck et al. [1999]: 84). Ein neuerer Code-Smell bezüglich unterschiedlicher Entitäten ist der mysteriöse Name, der die Verwendung von Namen für Methode, Module, Variablen und Klassen bezeichnet, die die Funktionalität und die Art der Verwendung der Entität nicht kommunizieren (vgl. Fowler [2019]: 72).
Ein allgemeinerer Code-Smell ist der duplizierte Code (vgl. Beck et al. [1999]: 76). Er beschreibt das wiederholte Vorkommen der gleichen Codestruktur, was vermieden und in eine eigene Entität extrahiert werden sollte (vgl. Beck et al. [1999]: 76). Ebenfalls eher allgemein ist der Code-Smell des toten Codes, bei dem der Code aktuell nicht ausgeführt wird und aus dem Programmcode entfernt werden sollte (vgl. Mäntylä and Lassenius [2007]: 407). Darüber hinaus gibt es den Code-Smell Instanceof, der die wiederholte Nutzung von instanceof-Operatoren nah beieinander im Programmcode beschreibt (vgl. Van Emden and Moonen [2002]: 101). Der instanceof-Operator ist dabei ein Operator in Java, der ein Objekt darauf prüft, ob es eine Instanz einer spezifischen Klasse ist (vgl. Ratz et al. [2018]: 271). Ein letztes Beispiel ist der Code-Smell namens Kommentare, der den nötigen Gebrauch von Kommentaren zum Verständnis des Codes beschreibt, was zu einem Refactoring führen sollte, der die Kommentare überflüssig macht (vgl. Beck et al. [1999]: 87 f.).

2.3. Einführung in den Compilerbau

Um ein eigenes statisches Code-Analyse-Tool zu implementieren werden sowohl Teile der Sprachtheorie, als auch Teile der Compiler-Theorie benötigt. Abbildung 2.1 zeigt den typischen Ablauf eines Compilers, der verwendet wird, um einen Quellcode in einen Zielcode zu übersetzen (vgl. Fraser and Hanson [1995]: 1). Wie in Unterabschnitt 2.1.4 dargelegt, führen statische Code-Analyse-Tools den Eingabecode nicht aus und verwenden lediglich einen Parser. Daher wird in der vorliegenden Arbeit auf den Teil des Compilers eingegangen, der vor der Code-Generierung beziehungsweise der semantischen Analyse einer gegebenen Sprache stattfindet. Die Code-Generierung beziehungsweise die semantische Analyse spielen demgegenüber nur bei dynamischen Code-Analyse-Tools eine Rolle.

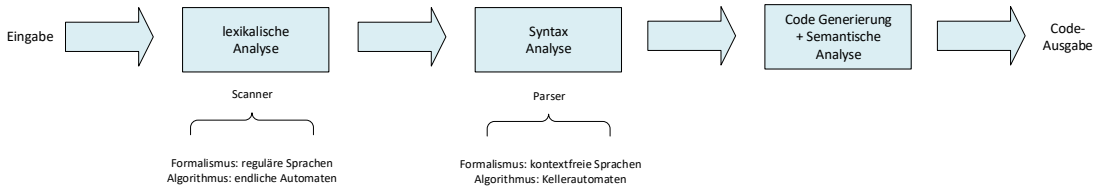

Abbildung 2.1.: Ablauf Compiler

Zuerst werden in Unterabschnitt 2.3.1 die benötigten Konzepte aus der Sprachtheorie erschlossen. Darauf folgt in Unterabschnitt 2.3.2 die Compilertheorie, die mit der lexikalischen Analyse beginnt und schlussendlich in Unterabschnitt 2.3.3 mit der syntaktischen Analyse fortgeführt wird.

2.3.1. Kurze Einführung in die Sprachtheorie

Um einen Parser für ein statisches Code-Analyse-Tool zu konstruieren, werden einige Begriffe aus der Sprachtheorie benötigt. Ein wichtiger Begriff ist das Alphabet, welches als endliche und nichtleere Menge von Zeichen definiert ist (vgl. Hedtstück [2012]: 6). Eine endliche Folge dieser Zeichen mit einer bestimmten Länge aus einem Alphabet ist ein Wort über diesem Alphabet (vgl. Hedtstück [2012]: 6). Aus diesem Alphabet lässt sich dabei eine endliche Menge an Wörtern erstellen (vgl. Hedtstück [2012]: 6). Ebenfalls ein Wort ist das sogenannte Leerwort, bestehend aus 0 Zeichen, wobei es mit ϵ bezeichnet wird (vgl. Hedtstück [2012]: 6). Eine Sprache ist dabei eine Teilmenge der endlichen Menge an Wörtern

eines Alphabets (vgl. Hedtstück [2012]: 6). Mit den Wörtern einer Sprache können dabei Operationen ausgeführt werden, wobei beispielsweise Wörter einer Sprache zu neuen Wörtern der Sprache verkettet werden können (vgl. Hedtstück [2012]: 7). Darüber hinaus sind Wörter einer Sprache selbst Zeichen eines Alphabets auf einem höheren Level, welches die Menge der syntaktisch und semantisch einwandfreien Sätze einer Sprache enthält (vgl. Hedtstück [2012]: 7). Zusätzlich ist der Begriff der Grammatik ein weiterer wichtiger Begriff. Eine Grammatik ist eine Menge von Umformungsregeln, die die erlaubten Operationen zur Erzeugung von Wörtern einer Sprache beschreiben (vgl. Priese and Erk [2018]: 54). Eine Sprache besteht dabei sowohl aus Nichtterminalen, die weiter ersetzt werden sowie aus Terminalen, die atomare Zeichen eines Alphabets einer spezifischen Sprache sind (vgl. Priese and Erk [2018]: 54). Auf der linken Seite einer Grammatik- oder Produktionsregel steht dabei eine Menge von Terminalen und Nichtterminalen ohne das Leerwort ϵ, während auf der rechten Seite eine Menge von Terminalen und Nichtterminalen angegeben wird (vgl. Hedtstück [2012]: 25 f.). Das Wort auf der linken Seite einer Produktionsregel produziert das Wort auf der rechten Seite der Produktionsregel, was durch einen Pfeil mit der Spitze nach rechts symbolisiert wird (vgl. Priese and Erk [2018]: 54). Dabei kann ein Wort wie beschrieben auch als Satz aufgefasst werden. Sprachen im Sinne der theoretischen Informatik bestehen dabei aus einem nichtterminalen Startsymbol, das ein neues Wort produziert, wobei das neue Wort aus Nichtterminalen und Terminalen bestehen kann (vgl. Priese and Erk [2018]: 54). Nichtterminale des neuen Wortes werden dann mithilfe der Produktionsregeln weiter ersetzt, bis ein Wort herauskommt, welches lediglich aus Terminalen besteht (vgl. Priese and Erk [2018]: 54).
Die Sprachen, die sich durch solche Produktionsregeln beschreiben lassen, werden formale Sprachen genannt (vgl. Priese and Erk [2018]: 54). Zusätzlich gibt es den Begriff der kontextfreien Sprache. Der Begriff beschreibt diejenigen Sprachen, bei deren Grammatiken immer nur lediglich ein Nichtterminal auf der linken Seite der Produktionsregel zu finden ist (vgl. Hedtstück [2012]: 32). Zusätzlich darf dasjenige Wort, welches das Leerwort ϵ produziert, nicht auf der rechten Seite der Produktionsregeln vorkommen (vgl. Hedtstück [2012]: 32). Zur Beschreibung von Programmiersprachen werden in der Regel solche kontextfreien Grammatiken verwendet (vgl. Aho et al. [2008]: 53). Eine Programmiersprache ist dann die Menge aller endlichen Folgen von Nichtterminalen und Terminalen, die ein syntaktisch und semantisch einwandfreies Programm beschreiben, wobei die Eingaben eines

einwandfreien Programms durch Grammatiken bestimmt werden (vgl. Hedtstück [2012]: 9). Die Syntax beschreibt dabei die richtige Form eines Programms einer konkreten Programmiersprache (vgl. Aho et al. [2008]: 51). Demgegenüber werden mit der Semantik die Bedeutungen solcher Programme definiert (vgl. Aho et al. [2008]: 51).
Um kontextfreie Sprachen in der Informatik zu beschreiben wird in der Regel die erweiterte Backus-Naur-Form (EBNF) verwendet, da sie maschinenlesbar ist (vgl. Hedtstück [2012]: 66). In der Backus-Naur-Form (BNF) werden lassen sich kontextfreie Grammatiken darstellen, wobei Nichtterminale mit „<...>“ umschlossen und Terminale nicht speziell markiert werden (vgl. Hedtstück [2012]: 66). Darüber hinaus wird die linke und die rechte Seite der Produktionsregel beibehalten sowie der Pfeil, der die beiden Seiten trennt, durch ein „::=“ ersetzt (vgl. Hedtstück [2012]: 38). Die EBNF erweitert die BNF, indem Optionalität durch „[...]“, Wiederholungen durch „{...}“und das Symbol für Strukturierungshilfe „(...)“ der Notation hinzugefügt werden (vgl. Hedtstück [2012]: 39 f.).
Um die Produktionen der einzelnen Wörter ausgehend von einem Startsymbol hierarchisch darzustellen, eignet sich ein sogenannter Parse-Baum (vgl. Aho et al. [2008]: 57). Das Startsymbol befindet sich dabei in der Wurzel des Baumes, während sich in jedem Blatt ein Terminal (oder das Leerwort ϵ) sowie in jedem inneren Knoten ein Nichtterminal befindet (vgl. Aho et al. [2008]: 57).

2.3.2. Die lexikalische Analyse

Nachdem in Unterabschnitt 2.3.1 wichtige Begriffe aus der Sprachtheorie besprochen wurden, wird im Folgenden in die lexikalische Analyse eingeführt. Wie in Abbildung 2.1 dargestellt, erhält der Lexer beziehungsweise der lexikalische Scanner die Zeichen eines Quellprogramms als Input, wobei er versucht daraus lexikalisch bedeutungsvolle Einheiten zu formen (vgl. Aho et al. [2008]: 54). Diese Zeichen stammen üblicherweise aus einer Datei (vgl. Wilhelm et al. [2012]: 3). Die bedeutungsvollen Einheiten werden als Lexeme bezeichnet, die durch sogenannte Tokens dargestellt werden (vgl. Aho et al. [2008]: 54). Jeder Token enthält einen Tokennamen sowie einen Attributwert, wobei der Tokenname als Terminale in Grammatiken dargestellt werden, sodass die Begriffe meistens synonym verwendet werden (vgl. Aho et al. [2008]: 54). Der Attributwert ist demgegenüber eine Referenz auf eine Symboltabelle, in der weitere Daten zu dem Token gespeichert sind (vgl. Aho et al. [2008]: 54). Symboltabellen sind Datenstrukturen, mit deren Hilfe

Compiler Informationen über Konstrukte des Quellprogramms speichern (vgl. Aho et al. [2008]: 105). Dabei werden Leerzeichen und Kommentare in der Regel eliminiert, wobei sie auch in die Syntax aufgenommen werden können (vgl. Aho et al. [2008]: 96). Des Weiteren benötigt ein Lexer in der Regel einen sogenannten Lookahead (deutsch: Vorhersage), um einige Zeichen vorausschauen zu können, sodass Lexeme wie „<“ und „<=“ unterschieden werden können (vgl. Aho et al. [2008]: 96). Darüber hinaus ist es meistens die Aufgabe des Lexers, die Zeichen einer Zahl zu einem gemeinsamen Nummern-Token zusammenzufassen (vgl. Aho et al. [2008]: 97). Des Weiteren erkennt der Lexer Schlüsselwörter und Bezeichner, indem gefundene Zeichenketten mit den in der Symboltabelle gespeicherten reservierten Wörtern abgeglichen werden (vgl. Aho et al. [2008]: 98 f.). Alle Zeichenketten, die mit keinem Schlüsselwort übereinstimmen, sind demnach Bezeichner (vgl. Aho et al. [2008]: 98).
Das Prinzip des Lexers lässt sich mit einem endlichen Automaten illustrieren, wobei sich der Lexer zu Beginn in einem Anfangszusstand befindet (vgl. Hedtstück [2012]: 52). Im Anschluss daran läuft eine Steuereinheit (aktuelles Zeichen) über das Eingabeband (Zeichenstream), wobei der Automat bei jedem gelesenen Zeichen seinen Zustand ändern kann (vgl. Hedtstück [2012]: 52). Ein Lexem wird erkannt, wenn der Lexer sich am Ende des Scannvorgangs in einem Endzustand befindet (vgl. Hedtstück [2012]: 52).

2.3.3. Die syntaktische Analyse

Zusammen mit dem Lexer erkennt der Parser die syntaktische Struktur sowie syntaktische Fehler eines Programms aus einer Quelldatei (vgl. Wilhelm et al. [2012]: 5 f.). Wie in Abbildung 2.1 dargestellt, erhält die syntaktische Analyse einen Strom von Tokens des Lexers (vgl. Aho et al. [2008]: 51). Jedes Programm besitzt eine hierarchische Struktur, die angibt, wie die einzelnen Konstrukte einer Programmiersprache in einem Programm zueinander angeordnet sind (vgl. Wilhelm et al. [2012]: 6). Der Parser liefert dabei einen AST als Ergebnis zurück (vgl. Hedtstück [2012]: 110). Gegenüber einem Parse-Baum, dessen innere Knoten aus Nichtterminalen bestehen, bestehen die inneren Knoten des AST aus Konstrukten der Programmiersprache, dessen Syntax der Parser analysiert (vgl. Aho et al. [2008]: 87). Zum Parser korrespondiert der Automatentyp des Kellerautomaten, der Parse-Bäume mithilfe eines Stacks, der das Last-In-First-Out-Prinzip nutzt traversiert (vgl. Hedtstück [2012]: 110). Außerdem gibt es das Problem

der Mehrdeutigkeit, bei dem die gleiche Grammatik unterschiedliche Parse-Bäume produzieren kann, was mithilfe der Assoziativität von Operatoren und der Operatorenpräzedenz gelöst wird (vgl. Aho et al. [2008]: 59 ff.). Die Assoziativität gibt dabei Auskunft darüber, auf welchen Operanden ein Operator angewandt wird, wobei Links- und Rechtsassoziativität unterschieden werden (vgl. Aho et al. [2008]: 60). Die Operatorenpräzedenz löst die Mehrdeutigkeit, indem eine Prioritätenrangliste von Operatoren angibt, welcher Operator bei Mehrdeutigkeiten zuerst angewandt wird (vgl. Aho et al. [2008]: 61). Durch diese Maßnahmen erzeugt die kontextfreie Grammatik immer nur einen Parse-Baum (vgl. Aho et al. [2008]: 59). Weiterhin kann die Konstruktion eines Parse-Baumes präzisiert werden, indem Produktionen als Ersetzungsregeln beziehungsweise Ableitungen behandelt werden (vgl. Aho et al. [2008]: 242). Dabei geht von jedem inneren Knoten beziehungweise Nichtterminal ein Ast zu einem seiner Ableitungen ab, wobei zwischen Rechts- und Linksableitungen unterschieden wird (vgl. Aho et al. [2008]: 242). Bei einer Linksableitung wird bei jeder Ableitung im Parsing-Algorithmus zuerst das sich am weitesten links befindende Nichtterminal gewählt, während bei der Rechtsableitung das sich am weitesten rechts befindende Nichtterminal zuerst abgeleitet wird (vgl. Aho et al. [2008]: 243).
Des Weiteren werden in der Literatur handgeschriebene rekursive Abstiegsparser sowie von Parser-Generatoren mithilfe kontextfreier Grammatiken generierte Parser unterschieden (vgl. Fischer et al. [2010]: 175). Ein Parser-Generator ist dabei selbst ein Compiler, der aus der übergebenen kontextfreien Grammatik ein übersetztes Programm, den Parser der von der übergebenen Grammatik erzeugten Sprache generiert (vgl. Fischer et al. [2010]: 176). Bei einem rekursiven Abstiegsparser wird jedes Nichtterminal mit einer implementierten Parsing-Funktionalität gehandhabt, wobei der Parser die richtige Produktion während der Inspektion der nächsten Tokens eines festgelegten Lookaheads von Tokens vorhersagt (vgl. Fischer et al. [2010]: 177 f.). Die Vorhersage des Parsers hängt dabei vor allem von der rechten Seite der Produktionsregel ab, wobei Vorhersageregeln aufgestellt werden (vgl. Fischer et al. [2010]: 177 ff.), was jedoch ausschließlich beim Top-Down-Parsing angewendet wird (vgl. Fraser and Hanson [1995]: 132). Top-Down-Parser erstellen ihren AST von der Wurzel aus, wobei der Baum nach unten wächst (vgl. Gumm et al. [2011]: 723). Beispiele für Top-Down-Parser sind der handgeschriebene rekursive Abstiegs-Parser sowie der generierte LL(1)- beziehungsweise LL(k)-Parser, der Parsing-Tabellen zur Auswahl der nächsten Produk-

tion auswählt (vgl. Wilhelm et al. [2012]: 103 f.). Dabei drückt „LL“ sowohl die Abtastung von links nach rechts, als auch das Erstellen einer Linksableitung aus (vgl. Aho et al. [2008]: 269). Gegenüber dem Top-Down-Parser beginnen Bottom-Up-Parser bei den Blättern des Parse-Baumes (den Terminalen), wobei mehrere kleine Bäume nach und nach zu einem einzigen AST zusammengesetzt werden (vgl. Gumm et al. [2011]: 726). Bottom-Up-Parser besitzen gegenüber Top-Down-Parsern den Vorteil, auch die Parse-Bäume linksrekursiver Grammatiken parsen zu können (vgl. Aho et al. [2008]: 257). Bei einer linksrekursiven Grammatik ist das am weitesten links befindende Nichtterminal auf der rechten Seite der Produktionsregel äquivalent zu dem sich am weitesten rechts befindenden Nichtterminal auf der linken Seite (vgl. Aho et al. [2008]: 257).
LL(k)-Parsing-Techniken, wie sie der LL(k) oder der rekursive Abstiegsparser verwenden, müssen die aktuelle Produktionsregel anhand der nächsten k Tokens Lookahead vorhersagen (vgl. Appel and Ginsburg [1998]: 56). Die Vorhersageregeln des rekursiven Abstiegsparsers können dabei mithilfe von Parsing-Tabellen in Form von First- und Follow-Mengen festgelegt werden (vgl. Fischer et al. [2010]: 178 f.). Die First-Menge eines Nichtterminals beschreibt alle Terminale, die zu Beginn einer Produktion des Nichtterminals auftauchen (vgl. Fraser and Hanson [1995]: 134). Die Follow-Menge eines Nichtterminals gibt demgegenüber an, welche Tokens auf das in der First-Menge angegebene Terminal folgen können (vgl. Fraser and Hanson [1995]: 136). Des Weiteren können Grammatiken danach unterschieden werden, wie viele Tokens ein Top-Down-Parser zur Syntaxanalyse benötigt, sodass eine LL(k)-Grammatik k Tokens Lookahead für den Parsing-Vorgang benötigt (vgl. Fischer et al. [2010]: 177 ff.). Gegenüber diesen LL(k)-Parsing-Techniken gibt es LR-Parsing-Techniken, die von links nach rechts parsen (vgl. Appel and Ginsburg [1998]: 56). Dabei können sie jedoch die Entscheidung für eine Produktionsregel aufschieben, bis sie die gesamte rechte Seite der Produktionsregel gesehen haben (vgl. Appel and Ginsburg [1998]: 56). Dabei handelt es sich um einen Bottom-Up-Parser (vgl. Wilhelm et al. [2012]: 113 f.). Ein Beispiel für einen LR-Parser ist der Lookahead-LR (LALR)-Parser, der mit speziellen Parsing-Tabellen für diesen Typ arbeitet (vgl. Aho et al. [2008]: 319 ff.).

3. Forschungsstand

In Kapitel 2 konnten die benötigten theoretischen Grundlagen aus dem Feld der Software-Inspektion sowie den Definitionen von Defekten, die dabei gesucht werden, ausführlich beschrieben werden. Nun wird im Folgenden auf den Forschungsstand zum Thema Code-Analyse-Tools zur Erkennung von Defekten eingegangen. Dabei werden solche Tools genannt, einen Bezug zum Clean-Code-Konzept besitzen.
Dabei gibt es im besten Wissen der vorliegenden Arbeit zum Zeitpunkt der Erstellung ebendieser Arbeit kein Code-Analyse-Tool in der Literatur, welches Clean-Code analysiert. Der Urheber des Clean-Code-Ansatzes gibt in seiner Hilfsliteratur Martin [2011] Tipps zur Erstellung von Clean-Code, wobei unter den empfohlenen Tools kein Clean-Code-Analyse-Tool vorhanden ist (vgl. Martin [2011]: 221 ff). Auch andere, ebenfalls programmiersprachenunabhängige Hilfsliteratur zur Erstellung von Clean-Code, nennen in ihren Tool-Empfehlungen keine Code-Analyse-Tools zur Analyse von Clean-Code, wie beispielsweise Post [2021] (vgl. Post [2021]: 257 ff) und Lampe [2020] (vgl. Lampe [2020]: 292). Auch bei Hilfsliteratur zur Erstellung von Clean-Code in einer konkreten objektorientierten Programmiersprache werden in ihren Empfehlungen keine Code-Analyse-Tools zur Erkennung von Clean-Code genannt, wobei sie jedoch Code-Analyse-Tools zur Erkennung von Code-Smells nennen. Hierzu gibt es die Beispiele Windler and Daubois [2022] bezüglich PHP (vgl. Windler and Daubois [2022]: Kap. 7.3), Alls [2020] bezüglich C# (vgl. Alls [2020]: Kap. 12.4), Anaya [2018] bezüglich Python (vgl. Anaya [2018]: Kap. 1) und Padolsey [2020] bezüglich JavaScript (vgl. Padolsey [2020] Kap. 15). Außerhalb von Clean-Code-Hilfsliteratur empfehlen Latte et al. [2019] in ihrer Ausarbeitung zur Erstellung von Clean-Code ebenfalls die Verwendung von Code-Smell-Analyse-Tools (vgl. Latte et al. [2019]: 97 f). Aufgrund der Empfehlungen der Code-Smell-Analyse-Tools in der genannten Literatur sowie der in Unterabschnitt 2.1.7 beschriebenen Nähe der Konzepte des Clean-Code und des Code-Smell-Konzepts, werden im Folgenden Code-Smell-Analyse-Tools aus der Li-

teratur beschrieben. Eine genauere Einordnung über die Schnittmengen der beiden Konzepte findet darüber hinaus in Abschnitt 4.1 statt.
Die in der Literatur bekannten Code-Smell-Analyse-Tools lassen sich, abhängig vom Ursprung der jeweiligen analysierten Informationen über den Programmcode, in vier verschiedene Kategorien einteilen. Diese Kategorien sind die Analyse von Strukturinformationen über den Programmcode, die Informationsgewinnung durch maschinelles Lernen (ML), die Analyse von Daten über Änderungen des Programmcodes, sowie die Textanalyse, die unmittelbar auf dem Programmcode stattfindet.
Eine wichtige Unterscheidung ist die, ob zur Messung von Code-Smells etablierte Metriken oder sogenannte ad-hoc-Metriken verwendet werden (vgl. Fontana et al. [2012]: 2). Da Code-Smells oft nicht eindeutig definiert sind (siehe Abschnitt 2.2), werden Schwellenwerte bezüglich Metriken zur Erkennung der Code-Smells festgelegt, die entweder auf bereits in der objektorientierte Programmierung (OOP) etablierten Metriken beruhen oder spontan erstellt werden, was Ad-hoc-Metrik genannt wird (vgl. Fontana et al. [2012]: 2). Dabei können jedoch nicht alle Code-Smells aus der Literatur durch die Verwendung von quantitativer Messung mittels Software-Metriken erkannt werden (vgl. Danphitsanuphan and Suwantada [2012]: 1). Die in diesem Kapitel beschriebenen Tools aus der Literatur sind dabei Laborprototypen, die nicht frei verfügbar sind (vgl. Fontana et al. [2015]: 217) und wahrscheinlich nicht mehr gewartet werden. Aktuell verfügbare Tools sind beispielsweise das Open-Source-Tool Checkstyle, oder das proprietäre Tool inFusion (vgl. Fontana et al. [2015]: 217). Diese Tools werden jedoch nicht in der Literatur beschrieben und daher auch in diesem Kapitel nicht weiter beleuchtet.
Tabelle 3.1 zeigt die verschiedenen Tools aus der Literatur. Die erste Spalte gibt dabei den Namen des jeweiligen Tools an, während die zweite Spalte die analysierten Programmiersprachen angibt. In einer dritten Spalte wird die Erkennungsmethode des konkreten Tools angegeben. Welche Code-Smells genau analysiert werden, wird in der vierten Spalte angegeben, wobei die Zahlen in der Legende unter der Tabelle einem Code-Smell zugeordnet werden. Die fünfte Spalte gibt an, woher die Informationen über den Programmcode kommen, die zur Erkennung der Code-Smells genutzt werden. Die letzte Spalte hingegen gibt die Referenz der Literatur an. Die Tools der ersten Kategorie, die am meisten verbreitet sind, sind solche Code-Analyse-Tools, die wie in Unterabschnitt 2.1.4 beschrieben abstrahierte Strukturinformationen aus dem Programmcode verwenden. Solche Struk-

turinformationen umfassen beispielsweise Informationen über Datentypen, Variablen, Operationen, die Auf- und Abrufe der Variablen und Operationen sowie Vererbungsbeziehungen zwischen Klassen (vgl. Lanza and Marinescu [2006]: 176). Üblicherweise liegen diese Strukturinformationen in einem AST vor, der das Resultat einer syntaktischen Analyse ist. Das heißt zu dieser Kategorie gehören die Tools, die einen eigens erstellten Parser oder einen Parser von Dritten nutzen. Sie werden in Abschnitt 3.1 dargestellt. In Abschnitt 3.2 werden Code-Smell-Analyse-Tools inspiziert, die Code-Smells durch ML aufspüren. Des Weiteren werden in Abschnitt 3.3 Tools dargestellt, die mit Änderungsdaten aus dem Versionskontrollsystem (VCS) bezüglich des Programmcodes arbeiten. Dabei beobachten sie die Änderungen des Programmcodes mittels des VCS. Schlussendlich werden in Abschnitt 3.4 diejenigen Tools erläutert, die eine Textanalyse direkt auf dem Programmcode durchführen.

Tool	**Sprachen**	**Erkennung**	**Code-Smells**	**Informations-ursprung**	**Ref.**
LINT	C	Verschärfte Grammatik	18, 25	Portable C Compiler	Johnson [1977]
jCosmo	Java	Userspez-ifische Parameter	1, 2, 3, 7, 9, 11, 12, 14, 15, 17, 18	LR-Parser	Van Emden and Moonen [2002]
iPlasma	Java, C++	Etablierte Metriken	1, 7, 8, 19, 20, 21, 22, 23, 26	RECODER (Java), McC (C++)	Lanza and Marinescu [2006]
JDeodorant	Java	Ad-hoc Metrik	7	Eclipse Java Entwicklungs-werkzeuge (JDT)	Fokaefs et al. [2007]
DCPP	Abhängig vom Graph	Quantitativ	5, 6	Artefakten-Graph	Rao and Reddy [2008]
DECOR	Java	BNF-Grammatik	BNF-Regelkarten	Mit JCUP generierter Parser	Moha et al. [2010]

Tool	Sprachen	Erkennung	Code-Smells	Informations-ursprung	Ref.
Stench Blossom	Java	Ad-hoc Metriken	1, 2, 3, 7, 8, 10, 11, 14, 18, 16, 24	Erkennungstool	Murphy-Hill and Black [2010]
BSDT	Java	Etablierte Metriken	2, 3, 4, 8, 11, 12, 13	Wahrscheinlich Eclipse JDT	Danphits-anuphan and Suwantada [2012]
JCode-Canine	Java	Ad-hoc Metriken	1, 7, 8, 11	Wahrscheinlich Eclipse JDT	Nongpong [2012]
Fica	C	Maschinelles Lernen	1	Outputs anderer Tools, User-Auswahl	Yang et al. [2012]
MLBA	Java	Maschinelles Lernen	2, 3, 7, 8, 19, 20	Outputs anderer Tools	Fontana et al. [2013]
HIST	Java	Analyse der Änderungs-historie	5, 6, 7, 12, 21	Code Analyzer, VCS	Palomba et al. [2013]
JCodeOdor	Java	Etablierte + ad-Hoc Metriken	6, 8, 14, 18, 19, 20, 23	Eclipse JDT	Fontana et al. [2015]
EDM	Java	Änderungs-historie	1, 5, 6	VCS	Fu and Shen [2015]
TACO	Java	Etablierte Metriken	2	Konzeptdaten	Palomba [2015]

Tabelle 3.1.: Code-Smell Analyse-Tools

Legende: [1] Duplizierter Code, [2] Lange Methode, [3] Große Klasse, [4] Lange Parameterliste, [5] Divergierende Änderungen, [6] Schrotkugeln herausoperieren, [7] Neid, [8] Datenklassen, [9] Datenklumpen, [10] Neigung zu elementaren Typen, [11] Switch-Befehle, [12] Parallele Verwaltungshierarchien, [13] Faule Klasse, [14] Nachrichtenketten, [15] Ausgeschlagenes Erbe, [16] Kommentare, [17] Instanceof, [18] Typecast, [19] Gottklasse, [20] Gehirnmethode, [21] Gehirnklasse, [22] Intensive Kopplung, [23] Verstreute Kopplung, [24] Magische Zahl, [25] Toter Code, [26] Traditionsbrecher

3.1. Code-Smell-Analyse-Tools auf Grundlage von Strukturinformationen

Das erste Code-Analyse-Tool in der Literatur stammt von Johnson [1977], der den Portable C Compiler verwendet (vgl. Johnson [1977]: 6). Da ein Compiler den Code ausführt, ist LINT als ein dynamisches Code-Analyse-Tool einzuordnen. In Johnson [1978] wird dabei erläutert, wie der Portable C Compiler mithilfe von Generatoren erstellt und angepasst wird (vgl. Johnson [1978]: 98). Lex ist ein bekanntes Tool zur Erstellung von Lexern, wobei es reguläre Ausdrücke erhält, um einen Lexer der Programmiersprache C zu erstellen (vgl. Johnson [1978]: 98). Des Weiteren wird der bekannte Parser-Generator Yacc mit regulären Ausdrücken gefüttert, um einen LALR-Parser zu generieren, wobei jedoch auch uneindeutige Grammatiken genutzt werden müssen, beispielsweise um Operatorenpräzedenz C-Operatoren zu beachten (vgl. Johnson [1978]: 98). Reguläre Ausdrücke sind Strings aus Zeichen und Metasequenzen, die beispielsweise die Lage, Art und Menge von Zeichen ausdrücken können und so zur Mustererkennung in Texten verwendet werden (vgl. Stubblebine et al. [2004]: 7). Darüber hinaus erstellt Yacc einen Bottom-Up-Parser, der aufgrund einiger Top-Down-Eigenschaften von C angepasst werden muss (vgl. Johnson [1978]: 98). Der dabei generierte LALR-Parser gibt dabei Zeilen von ASCII-Code aus, die beispielsweise Variablen aus der Symboltabelle oder Ausdrücke in C darstellen (vgl. Johnson [1977] 6). Um die Generierung des Maschinencodes beim Übersetzungsvorgang kümmert sich Johnson [1978] selbstständig (vgl. Johnson [1978]: 99 ff.). Dabei werden sowohl Typecasts (vgl. Johnson [1977]: 4), als auch nicht verwendete Variablen und Funktionen (vgl. Johnson [1977]: 1 f.) (toter Code) angemahnt, wobei letzterer auch Teil der Clean-Code-Heuristiken ist (vgl. Martin [2009]: 340). Das heißt, es wird lediglich binär geprüft, ob eine verschärfte Grammatik erfüllt wird, ohne konkrete Metriken zur Erkennung der Code-Smells anzuwenden. Eine Überprüfung der Validität der Ergebnisse findet dabei nicht statt.
Auch jCosmo verwendet Strukturinformationen aus dem Programmcode, die ein eigener LR-Parser generiert, wobei jCosmo ein statisches Code-Analyse-Tool ist (vgl. Van Emden and Moonen [2002]: 101). Das Tool kann unter anderem die Code-Smells duplizierter Code, lange Methode, Neid und Switch-Befehle erkennen (vgl. Van Emden and Moonen [2002]: 98), die sehr ähnlich zu einigen Clean-Code-Heuristiken sind (vgl. Martin [2009]: 64 ff.). Der LR-Parser erlaubt dabei

das Parsen aller kontextfreien Grammatiken (vgl. Van Emden and Moonen [2002]: 101). Um den LR-Parser zu generieren, werden die Grammatiken der Programmiersprache Java in einer speziellen Grammatik aufgeschrieben, die der Parser-Generator der ASF+SDF META-ENVIRONMENT Entwicklungsumgebung versteht (vgl. Van Emden and Moonen [2002]: 101), an deren Erstellung Van Emden and Moonen [2002] beteiligt sind. Die ASF+SDF META-ENVIRONMENT ist eine Entwicklungsumgebung zur Konstruktion von Sprachdefinitionen sowie zur Entwicklung weiterführender Tools bezüglich dieser Definitionen (vgl. Van Den Brand et al. [2001]: 3). Der Parser der Entwicklungsumgebung extrahiert aus der Grammatik Informationen und schreibt sie in eine Parsing-Tabelle (vgl. Van Den Brand et al. [2001]: 5 f.). Anschließend generiert der Parser-Generator der Entwicklungsumgebung aus der Parsing-Tabelle einen LR-Parser in der Programmiersprache C, der die in der Grammatik beschriebene Sprache syntaktisch analysieren kann (vgl. Van Den Brand et al. [2001]: 5 f.). Zusätzlich werden Module erstellt, die der Parser-Generator der Entwicklungsumgebung benötigt, damit der erstellte LR-Parser die richtigen Strukturinformationen bezüglich der Code-Smells extrahieren kann (vgl. Van Emden and Moonen [2002]: 101). Mithilfe der Strukturinformationen über den Programmcode erkennt jCosmo eine Vielzahl von Code-Smells, die entweder direkt im Programmcode beobachtbar sind oder indirekt aus den Strukturinformationen gefolgert werden können. (vgl. Van Emden and Moonen [2002]: 101). Zur Erkennung der Letzteren werden Metriken beziehungsweise Schwellenwerte verwendet, die vom User eingegeben werden können (vgl. Van Emden and Moonen [2002]: 99).

Ein weiteres Code-Analyse-Tool, das einen eigenen Parser verwendet, ist das Tool DECOR von Moha et al. [2010], wobei es die Replizierbarkeit der Ergebnisse in den Mittelpunkt stellt (vgl. Moha et al. [2010]: 20). Da Code-Smells nicht immer eindeutig sind und die Parameter der Code-Smell-Erkennungsalgorithmen oft nicht offengelegt werden sowie die Ergebnisse nicht auf einer breiten Basis getestet werden, ist die Replizierbarkeit und Vergleichbarkeit von Ergebnissen in der Regel gering (vgl. Moha et al. [2010]: 20). Um diese Punkte zu entkräften, nutzt DECOR eine eigene Spezifikationssprache, sodass Code-Smell-Spezifikationen in BNF-Regelkarten an das Tool übergeben werden müssen und theoretisch alle denkbaren Code-Smells spezifiziert werden können (vgl. Moha et al. [2010]: 22). Das Tool übersetzt die Regelkarten in konkrete Code-Smell-Erkennungsalgorithmen, die dadurch offengelegt werden (vgl. Moha et al. [2010]: 22). Da die BNF-Regelkarten

auch für Clean-Code-Heuristiken angepasst werden könnten, könnte DECOR theoretisch auch zu einem vollwertigen Clean-Code-Analyse-Tool werden. Zur Extraktion der Strukturinformationen aus dem Java-Programmcode kommt ein eigener Parser zum Einsatz (vgl. Moha et al. [2010]: 23). Der Lexer wurde dabei mit dem Java-Lexer-Generator JFLEX erstellt, während der Parser mit dem Java-Parser-Generator JCUP erstellt wurde (vgl. Moha et al. [2010]: 28). Die Code-Smell-Erkennungsalgorithmen aus den BNF-Regelkarten legen die Metriken und Schwellenwerte zur Erkennung von Code-Smells fest und werden mithilfe der Strukturinformationen des Parsers angewandt (vgl. Moha et al. [2010]: 23). Das heißt, es findet wie bei jCosmo eine Eingabe userspezifischer Metriken und Schwellenwerte statt. Um das Tool beziehungsweise die von Moha et al. [2010] erstellten BNF-Regelkarten zu testen, wurden drei Masterstudierende und zwei Softwareingenieure unabhängig voneinander gebeten, mithilfe der Beschreibungen von Beck et al. [1999] und Brown [1998] verschiedene Code-Smells und AntiPatterns im Programmcode von zehn Open-Source-Java-Projekten zu suchen (vgl. Moha et al. [2010]: 31). Die Präzision respektive der Recall des Tools berechnen sich anhand der Anzahl der während des Code-Reviews gefundenen Code-Smells beziehungsweise AntiPatterns (die Wahr-Positiven), geteilt durch die Anzahl der vom Tool gefundenen Code-Smells beziehungsweise AntiPatterns, die zuvor mit den nicht-gefundenen Code-Smells beziehungsweise AntiPatterns (Falsch-Positive) bei der Präzision respektive der Anzahl der während des Code-Reviews nicht-gefundenen Code-Smells beziehungsweise AntiPatterns beim Recall addiert wird (vgl. Moha et al. [2010]: 31). Die durch die BNF-Regelkarten erstellten Erkennungsalgorithmen haben dabei im Durchschnitt einen Recall von 100 % sowie eine Präzision von mehr als 50 % erreicht, wodurch die Nützlichkeit des Tools bei der Erkennung von Defekten nachgewiesen werden konnte (vgl. Moha et al. [2010]: 33).
Ein erstes Tool, welches einen Parser von Dritten verwendet, ist das statische Code-Analyse-Tool iPlasma von Lanza and Marinescu [2006] (vgl. Lanza and Marinescu [2006]: 176). Es kann die objektorientierten Programmiersprachen Java und C++ auf Code-Smells hin analysieren (vgl. Lanza and Marinescu [2006]: 175). Dabei erkennt es unter anderem die Code-Smells duplizierter Code und Neid, die für eine Clean-Code-Analyse ebenfalls von Relevanz sind. Um Strukturinformationen von Programmcodes der Programmiersprache Java zu extrahieren, wird der Open-Source-Parser RECODER verwendet (vgl. Lanza and Marinescu [2006]: 176). Des Weiteren nutzt iPlasma Model Capture for C++ (McC), um Strukturinformatio-

nen aus C++-Programmcode zu extrahieren, wobei die Informationen von McC in Form von ASCII-Tabellen bereitgestellt werden (vgl. Lanza and Marinescu [2006]: 176). Das Tool besitzt dabei das Merkmal, in vielen Fällen selbst bei fehlerhaftem Code ein Meta-Modell des Programmcodes zu erstellen, beispielsweise wenn die Einbindung von Bibliotheks-Deklarationen fehlt (vgl. Lanza and Marinescu [2006]: 175). Insgesamt verwendet das Tool über 80 etablierte Metriken, die verschiedene Arten des Softwaredesigns messen (vgl. Lanza and Marinescu [2006]: 177). Diese Metriken werden für komplexe Code-Smell Erkennungsalgorithmen genutzt, wobei zusätzlich zur Erkennung des Code-Smells duplizierter Code das Textvergleichstool DUDE verwendet wird (vgl. Lanza and Marinescu [2006]: 177). Dabei kann DUDE, zusätzlich zur Erkennung von Duplikaten, die direkt im Programmcode erkannt werden, die Strukturinformationen des Meta-Modells bezüglich Duplikaten annotieren, wodurch Duplikate mit ihrem Kontext untersucht werden können (vgl. Lanza and Marinescu [2006]: 177). Da duplizierter Code ein Code-Smell ist, der ebenfalls in den Clean-Code-Heuristiken vorkommt, ist diese Vorgehensweise auch für die Erkennung von Clean-Code relevant. Die Validierung der Ergebnisse wird durch die Nutzung in der Industrie begründet, da Großprojekte wie Eclipse (über 1.36 Millionen Zeilen Java-Code) und Mozilla (über 2.56 Millionen Zeilen C++-Code) iPlasma zur Analyse ihres Softwaredesigns verwendet haben (vgl. Lanza and Marinescu [2006]: 179 f.), wobei keine systematische Validierung angegeben ist.

Zusätzlich existiert das statische Code-Analyse-Tool JDeodorant von Fokaefs et al. [2007], welches den Code-Smell Neid im Java-Programmcode mithilfe von Strukturinformationen des ASTParsers aus den Eclipse JDT analysiert (vgl. Fokaefs et al. [2007]: 519). Es bietet die Besonderheit von Refactoring-Hinweisen für alle Methoden im Code, die nach dem Code-Smell Neid in einer falschen Klasse platziert sind (vgl. Fokaefs et al. [2007]: 519). Für jede Methode im Programmcode wird bezüglich einer Klasse berechnet, wie viele Daten abgerufen und Funktionalitäten aufgerufen werden (vgl. Fokaefs et al. [2007]: 519). Wird dieser Vorgang für eine Methode bezüglich jeder Klasse wiederholt, so lassen sich die Distanzen zwischen einer Methode und allen anderen Klassen vergleichen (vgl. Fokaefs et al. [2007]: 519). Diese ad-hoc-Metrik zeigt dann, ob eine Methode in der Klasse ist, zu der sie die niedrigste Distanz besitzt, wobei im gegenteiligen Fall der Code-Smell Neid gefunden und ein Refactoring-Hinweis angezeigt wird (vgl. Fokaefs et al. [2007]: 519 f.). Die Evaluation des Tools wird mit zwei bereits durch die

Autorinnen und Autoren der Codebeispiele annotierten Java-Programmcodes aus der Refactoring-Literatur vollzogen, wobei 15 von 16 möglichen Neid-Code-Smells durch JDeodorant entdeckt wurden (vgl. Fokaefs et al. [2007]: 520).
Ein weiteres statisches-Code-Analyse-Tool ist Stench Blossom, das unter anderem die für Clean-Code relevanten Code-Smells lange Methode, Neid, Switch-Befehle, Kommentare und duplizierter Code in der Programmiersprache Java analysiert (vgl. Murphy-Hill and Black [2010]: 6). Der Fokus des Tools liegt auf dem User-Interface, wobei drei unterschiedliche Views für den Anwendenden bereitgestellt werden (vgl. Murphy-Hill and Black [2010]: 6). Dem Anwendenden wird während des Programmierens ein sogenannter Ambient-View (deutsch: Umgebungsansicht) angeboten, der die Stärke von Code-Smells im aktuellen View-Kontext anzeigt (vgl. Murphy-Hill and Black [2010]: 6). Daneben gibt es einen sogenannten Active-View (deutsch: aktiv-Ansicht), die Code-Smells im Code direkt anzeigt (vgl. Murphy-Hill and Black [2010]: 6). Klickt der Anwendende im Active-View auf den markierten Code-Smell, öffnet sich der Explanation-View (deutsch: Erklärungsansicht), der Details zu dem angeklickten Code-Smell nennt (vgl. Murphy-Hill and Black [2010]: 6). In der gesamten Zeit läuft eine statische Analyse-Engine im Hintergrund, sodass die Informationen, die der Ambient-View benötigt, permanent verfügbar sind, wobei nur die Code-Smells im Programmcode des aktuellen Views analysiert werden (vgl. Murphy-Hill and Black [2010]: 7). Im Ambient-View werden blütenförmige Hinweise am Bildschirmrand angezeigt, deren Blütenfarbe auf den Code-Smell hinweist, sodass sie angeklickt werden können, um den Active-View mit farblichen Markierungen des Code-Smells im Programmcode zu aktivieren (vgl. Murphy-Hill and Black [2010]: 7 f.). Nach dem Anklicken des farblich markierten Codes werden im Explanation-View detaillierte Informationen zum erkannten und angeklickten Code-Smell angezeigt, wobei auch die Anzeige von Annotationen im Code bezüglich Code-Smells in diesem View aktiviert werden kann (vgl. Murphy-Hill and Black [2010]: 8 f.). Details zur statischen Code-Analyse sowie den Algorithmen zur Code-Smell-Erkennung werden nicht genannt. Eine Evaluation findet mithilfe von 12 Entwickelnden statt, wobei sechs kommerzielle Java-Entwickelnde und sechs Studierende eines Moduls über Datenbanksysteme ausgewählt wurden, die zumindest etwas vertraut mit der Programmiersprache Java sein mussten (vgl. Murphy-Hill and Black [2010]: 9). Mithilfe eines Fragebogens und quantitativer Datenanalyse wurde im Nachgang einer Trainingsphase mit beziehungsweise ohne die Nutzung des Tools eine verbesserte Sichtung von Code-Smells bei den Entwick-

lenden festgestellt, die das Tool verwendeten (vgl. Murphy-Hill and Black [2010]: 11 f.). Darüber hinaus konnten zur Hypothese der Subjektivität von Code-Smells lediglich Hinweise gefunden werden (vgl. Murphy-Hill and Black [2010]: 11). Des Weiteren konnte eine verbesserte Entscheidung bezüglich Refactoring-Maßnahmen bei den Entwickelnden, die das Tool nutzen, sowie die begrüßenswerte Unterstützung der Code-Smell-Erkennung durch das Design der Stench Blossom-Richtlinien gezeigt werden(vgl. Murphy-Hill and Black [2010]: 11 f.).
Danphitsanuphan and Suwantada [2012] haben das statische Code-Analyse-Tool BSDT entwickelt, welches als Eclipse Plug-In umgesetzt ist (vgl. Danphitsanuphan and Suwantada [2012]: 1). Daraus folgt die wahrscheinlich erneute Nutzung der Eclipse JDT, wobei jedoch die Herkunft der Strukturinformationen über den Java-Programmcode nicht näher spezifiziert ist. Dabei werden unter anderem die für Clean-Code relevanten Code-Smells der langen Methode, der langen Parameterliste und der Switch-Befehle erkannt (vgl. Danphitsanuphan and Suwantada [2012]: 2). OOP ist ein Forschungsfeld, in dem bereits etablierte Metriken zur quantitativen Messung von Software bestehen, wobei die von Danphitsanuphan and Suwantada [2012] genutzten Metriken auf die beiden Kategorien Klasse und Methode aufgeteilt werden (vgl. Danphitsanuphan and Suwantada [2012]: 2 f.). Wie die Metriken genau erhoben werden wird nicht angegeben (vgl. Danphitsanuphan and Suwantada [2012]: 2f.). Danphitsanuphan and Suwantada [2012] haben dabei selbstständig Schwellenwerte der verwendeten etablierten Metriken festgelegt, um bestimmte Code-Smells zu erfassen, wobei manche Code-Smells auch erst bei der Überschreitung mehrerer Schwellenwerte verschiedener Metriken erkannt werden (vgl. Danphitsanuphan and Suwantada [2012]: 2). Das Tool zeigt dem Anwendenden die Position des Code-Smells im Java-Programmcode, sowie diejenigen Werte der Software-Metriken, die die festgelegten Schwellenwerte überschreiten (vgl. Danphitsanuphan and Suwantada [2012]: 3). Dabei werden zusätzlich passende Refactoring-Hinweise gegeben und die Metriken außerdem in eine .csv-Datei gespeichert (vgl. Danphitsanuphan and Suwantada [2012]: 3). Zur Evaluation des Tools wurden die gleichen zwei annotierten Code-Beispiele wie bei Fokaefs et al. [2007] verwendet, wobei für die Code-Smells faule Klasse, lange Methode und Switch-Befehle von 14 Code-Smells 100 % gefunden wurden (vgl. Danphitsanuphan and Suwantada [2012]: 3). Die weiteren Code-Smells, die das Tool erkennt, wurden dabei nicht validiert.
Zusätzlich gibt es in dieser Kategorie das statische Code-Smell-Analyse-Tool JCo-

deCanine, das ein Eclipse-Plugin ist sowie die Strukturinformationen aus der Java-Programmanalyse von Eclipse nutzt (vgl. Nongpong [2012]: 91). Damit sind höchstwahrscheinlich erneut die Eclipse-JDT gemeint, die die Strukturinformationen bereitstellen, wobei es nicht exakt angegeben ist. JCodeCanine automatisiert den Refactoring-Prozess, indem bei jeder Änderung des Programmcodes eine Code-Smell-Erkennung durchgeführt wird, wobei ein gefundener Code-Smell durch eine vorgeschlagene Refactoring-Lösung automatisch behoben werden kann (vgl. Nongpong [2012]: 97). Das Tool zeigt unter anderem Warnungen, die Zeilennummer und den Typ (vgl. Nongpong [2012]: 97) für die Code-Smells duplizierter Code, Neid und Switch-Befehle an (vgl. Nongpong [2012]: 106). Mithilfe eines Java-Quell-Adapters wird der Eclipse-AST in einen AST des Open-Source Frameworks Fluid umgewandelt (vgl. Nongpong [2012]: 91). Fluid arbeitet intern mit einem AST, wobei es eine interne Repräsentation (IR) in Form von Knoten des AST, die Objekte im Programmcode repräsentieren, sowie in Form von sogenannten Slots ausgibt (vgl. Nongpong [2012]: 88). Slots können sowohl Werte, als auch Referenzen auf Knoten speichern (vgl. Nongpong [2012]: 88). Dabei können diese Slots zusammen in Containern gruppiert werden, wodurch beispielsweise ein Slot mit einer Referenz zu einem Knoten, der eine Methodendeklaration repräsentiert, zusammen mit weiteren Slots gruppiert werden können (vgl. Nongpong [2012]: 88). Die anderen Slots können dann beispielsweise Referenzen auf die Kindknoten des ersten Knotens halten, während ein weiterer Slot einen Wert annehmen könnte, welches den Methodenbezeichner hält (vgl. Nongpong [2012]: 88). Um von der Fluid-IR wieder auf den Programmcode schließen zu können, wird anschließend ein sogenannter Unparser verwendet, der benötigt wird, um ein automatisiertes Refactoring zu ermöglichen (vgl. Nongpong [2012]: 88 f.). Der Java-Quell-Adapter wird gebraucht, da der Eclipse-AST und der Fluid-AST inkompatibel sind, wobei insbesondere der Eclipse-AST kein Versioning enthält, sodass Änderungen am Programmcode nicht mit in die Analyse einbezogen werden können (vgl. Nongpong [2012]: 89 f.). Dieses Problem wird durch den Java-Quell-Adapter gelöst, sodass bei der Erstellung jedes neuen Fluid-ASTs eine neue Version erstellt wird (vgl. Nongpong [2012]: 91). Duplizierter Code wird dabei während des Parsing-Vorgangs im AST gefunden (vgl. Nongpong [2012]: 54 f.), während für den Code-Smell Neid eine ad-hoc-Kohäsionsmetrik erstellt wird (vgl. Nongpong [2012]: 57 f.). Switch-Befehle werden demgegenüber im AST erkannt (vgl. Nongpong [2012]: 60).
Zur Evaluation des Tools haben sie es mit den sieben annotierten Java-Testprogrammen

des fluid.util-Packages (über 50.0000 Programmzeilen), sowie des java.lang-Packages (über 10.000 Programmzeilen) und des java.util-Packages (über 13.000 Programmzeilen) getestet (Nongpong [2012]: 105). Des Weiteren haben sie es noch mit vier Programmen des UWM CS552 Homeworks, einer Reihe von Programmen, welche von Studierenden geschrieben wurden und jeweils zwischen 500 und 3.000 Programmzeilen umfassen, getestet (Nongpong [2012]: 105). Dabei konnte beim Code-Smell duplizierter Code eine Genauigkeit von 69 %, bei Neid eine Genauigkeit von 92 % und bei Switch-Befehlen eine Genauigkeit von lediglich 19 % erreicht werden (vgl. Nongpong [2012]: 106). Hierbei berechnet sich die Genauigkeit des Tools, indem die Anzahl der gefunden Code-Smells durch die Anzahl der gefundenen Code-Smells, addiert mit der Anzahl der nicht gefundenen Code-Smells, geteilt wird (vgl. Russell and Norvig [2012]: 1006).
Ein letztes Tool aus dieser Kategorie ist das statische Code-Analyse-Tool JCodeOdor von Fontana et al. [2015], wobei das Tool einen Intensitätsindex für Code-Smells verwendet (vgl. Fontana et al. [2015]: 16). Die Strukturinformationen aus dem Java-Programmcode werden abermals vom Parser der Eclipse-JDT extrahiert, wobei es nicht als Eclipse-Plugin umgesetzt ist (vgl. Fontana et al. [2015]: 17). Im Vergleich zu anderen Tools enthält das Tool zusätzlich unterschiedliche Intensitätslevel zur Anzeige der Schwere der Code-Smells, wobei sie von 1 (nicht sehr intensiv) bis 5 (sehr intensiv) reichen (vgl. Fontana et al. [2015]: 18). Dem Anwendenden wird diese Schwere der Code-Smells auf einer ganzzahligen Skala von 1 bis 10 angezeigt (vgl. Fontana et al. [2015]: 19). Die Höhe der Schwellenwerte können vom Anwendenden zwischen drei verschiedenen Stufen selbst ausgewählt werden, wobei eine Standardstufe, eine weniger strenge Stufe und eine strenge Stufe zur Auswahl stehen (vgl. Fontana et al. [2015]: 18). Mithilfe eines Modells aus den Strukturinformationen über den Java-Programmcode werden etablierte Metriken oder andere Informationen über bestimmte Entitäten oder Entitätengruppen abgefragt (vgl. Fontana et al. [2015]: 18 f.). Anschließend werden mithilfe dieser Informationen die Code-Smells erkannt und die Intensitätslevel für jeden Code-Smell anhand von Schwellenwerten, die abhängig von der Eingabe des Anwendenden sind, berechnet (vgl. Fontana et al. [2015]: 18). Eine Evaluation findet dabei lediglich bezüglich der verwendeten Schwellenwerte statt, ohne nach Falsch-Positiven oder Falsch-Negativen in den 74 getesteten Programmen zu suchen (vgl. Fontana et al. [2015]: 20 ff.).

3.2. Code-Smell-Analyse durch maschinelles Lernen

Neben statischen- und dynamischen Code-Smell Analyse-Tools, die die Strukturinformationen über den Programmcode aus einem AST verwenden, gibt es in der Literatur auch ML-Tools zur Erkennung von Code-Smells. Dabei kommt das sogenannte überwachte Lernen zum Einsatz. Beim überwachten Lernen lernt der Klassifizierer des neuronalen Netzwerks durch das Testen mit einer Trainingsmenge, die eine Anzahl von N Ein-/Ausgabe-Paaren enthält, eine Funktion h zu schätzen, die sich im Laufe des Trainings mithilfe der Beispiele in die Richtung einer wahren Funktion f bewegt (vgl. Russell and Norvig [2012]: 811). Lernen findet dabei statt, indem der Klassifizierer ein Input aus der Trainingsmenge erhält, das Output schätzt und durch den Lehrerenden Feedback erhält, ob die Schätzung richtig war, sodass die Parameter der Funktion angepasst werden können (vgl. Russell and Norvig [2012]: 811). Dabei wird eine Funktion h gesucht, die nicht nur für die N Beispiele aus der Trainingsmenge exakte Schätzungen des Outputs nach dem Erhalt des Inputs abgibt, sondern auch für Beispiele außerhalb der Trainingsmenge (vgl. Russell and Norvig [2012]: 812) und somit generalisiert werden kann. Ein Code-Smell Analyse-Tool, welches ML nutzt, ist Fica von Yang et al. [2012] (vgl. Yang et al. [2012]: 77). Es analysiert den Code-Smell duplizierter Code im Programmcode der Programmiersprache C (vgl. Yang et al. [2012]: 76). Dabei wird das Problem benannt, nach dem die Analyse von dupliziertem Code in einem gegebenen Programmcode in der Regel eine lange Liste von Duplikaten im Programmcode erstellt (vgl. Yang et al. [2012]: 76). Von dieser langen Liste an Duplikaten ist jedoch nur eine kleine Anzahl wirklich nützlich, um die Qualität des Programmcodes mithilfe von Refactoring zu verbessern (vgl. Yang et al. [2012]: 76). Bei der Nutzung von Fica wird ein Programmcode des Anwendenden an das Tool übergeben, wobei ein externes Duplikat-Erkennungs-Tool eine Menge von Duplikaten in dem Programmcode erkennt (vgl. Yang et al. [2012]: 77). Im Anschluss markiert der Anwendende die nützlichen Duplikate, woraufhin das ML-Tool die markierten Duplikate auf ihre Charakteristiken hin untersucht, sowie Duplikate mit ähnlichen Charakteristiken, die vom Anwendenden nicht markiert wurden, vorschlägt (vgl. Yang et al. [2012]: 77). Diese Vorschläge können dann vom Anwendenden bewertet werden, sodass die Möglichkeit besteht, neue Markierungen hinzuzufügen (vgl. Yang et al. [2012]: 77). Dabei werden die Bewertungen

des Anwendenden in einer Datenbank gespeichert (vgl. Yang et al. [2012]: 77). Aus den Beschreibungen geht hervor, wie Fica einen Klassifizierer im Sinne des überwachten Lernens trainiert, um ein Modell zur Erkennung von nützlichen Duplikaten zu erhalten. Dabei nimmt der Anwendende die Rolle des Lehrenden ein. Da die Entscheidung, ob ein Duplikat nützlich ist oder nicht, die Bedeutung der Wörter betrifft, sind ML-Tools in der Lage, die semantische Analyse von Sprache zu lernen.

Ein weiteres ML-Tool zur Erkennung von Code-Smells ist MSBA, das einen modifizierten Output anderer Code-Smell Analyse-Tools, wie beispielsweise von iPlasma, als Input für seinen Klassifizierer nutzt (vgl. Fontana et al. [2013]: 397). Um die Outputdaten zu erhalten wurden 76 unterschiedliche Java-Programme ausgewählt, wobei mithilfe der externen Code-Smell Analyse-Tools etablierte Metriken über den Programmcode und die gefundenen Code-Smells entnommen wurden. Im Anschluss wurden die gefundenen Code-Smells manuell gelabelt, indem ihnen ein Schweregrad zugeordnet wurde (vgl. Fontana et al. [2013]: 397). Dabei wurde ein Schweregrad zwischen 0 und 3 bestimmt, sodass das Modell mithilfe der zusätzlichen Informationen besser im Erkennen von Code-Smells wird, als es bei einer binären Variante der Fall wäre (vgl. Fontana et al. [2013]: 398). Außerdem können Code-Smells mit höheren Schweregraden beim Refactoring priorisiert werden (vgl. Fontana et al. [2013]: 398). Das Tool evaluiert unter anderem die für Clean-Code relevanten Code-Smells lange Methode und Neid (vgl. Fontana et al. [2013]: 397). Die gelabelten Daten wurden dann in die Form einer Matrix gebracht, wobei die Entität, auf die sich die Beobachtung bezieht, die Tatsache, ob es eine Code-Smell-Beobachtung ist, sowie ein zusätzliches Merkmal bezüglich einer Metrik über die Beobachtung in die Matrix eingefügt wurden (vgl. Fontana et al. [2013]: 398). Dabei wurden mithilfe dieser Daten sechs unterschiedliche vortrainierte neuronale Netzwerke trainiert, die gegeneinander getestet wurden, und je nach Modell eine Genauigkeit von 69 % bis 96,4 % für den Code-Smell Neid, sowie zwischen 69,2 % und 99,2 % für den Code-Smell lange Methode erreicht werden konnten (vgl. Fontana et al. [2013]: 398 f.).

3.3. Code-Smell-Suche mithilfe von Änderungsdaten

Neben den bereits genannten Ansätzen zur Identifizierung von Code-Smells im Programmcode gibt es ebenfalls den Ansatz Code-Smells mithilfe von Änderungsdaten bezüglich von Code-Dokumenten zu analysieren, da viele Code-Smells sich über längere Zeiträume manifestieren (vgl. Palomba et al. [2013]: 268). Beispielsweise kann der Code-Smell Neid über die Zeit entstehen, wenn immer mehr Zugriffe auf Daten beziehungsweise Abrufe von Methoden fremder Klassen bei einer Methode hinzugefügt werden (vgl. Palomba et al. [2013]: 268).
Ein erstes Tool dieser Kategorie ist DCPP, wobei die Code-Smells Schrotkugeln herausoperieren und divergierende Änderungen mithilfe quantitativer Methoden analysiert werden (vgl. Rao and Reddy [2008]: 1). Diese Code-Smells sind dabei stark mit der Wartbarkeit von Code verknüpft (vgl. Rao and Reddy [2008]: 1), weswegen sie auch für Clean-Code relevant sind. Dabei werden beide Code-Smells als analysierbar mithilfe der sogenannten Change Propagation (deutsch: Änderungsausbreitung) aufgefasst, die auf Stärke der Kopplung zwischen Artefakten basiert, wobei Artefakte hier Entitäten im Programmcode abbilden (vgl. Rao and Reddy [2008]: 1 f.). Dazu wurde das Konzept der DCPP-Matrix entworfen, die für VCS' entwickelt wurde, um Code-Smells im VCS direkt erkennen zu können (vgl. Rao and Reddy [2008]: 1). Dabei repräsentiert die DCPP-Matrix Änderungen im Programmcode der Ausbreitungswahrscheinlichkeiten von Artefakten (vgl. Rao and Reddy [2008]: 1). Besitzt ein Programmcode N Artefakte, besitzt die DCPP-Matrix eine Größe von N x N, sodass der Eintrag in der Zeile A, Spalte B, die Wahrscheinlichkeit repräsentiert, nach der eine Änderung des Artefakts A, eine Änderung in Artefakt B erforderlich macht, um die Funktionalität des Programms zu gewährleisten (vgl. Rao and Reddy [2008]: 2). Zur Bildung der DCPP-Matrix benötigt DCPP einen Input in Form einer konkreten gerichteten und gewichteten Artefakten-Graphenstruktur (vgl. Rao and Reddy [2008]: 2 f.), sodass theoretisch jede Programmiersprache abgebildet werden kann, insofern ein Modell des Programmcodes in die konkrete Graphenstruktur übersetzt wird. Mithilfe von ad-hoc-Metriken werden die Werte der DCPP-Matrix erstellt. Abhängig von festgelegten Schwellenwerten entscheidet das Tool bezüglich welchen Artefakts ein Code-Smell vorliegt (vgl. Rao and Reddy [2008]: 5). Eine Evaluation des Tools findet dabei nicht statt (vgl. Rao and Reddy [2008]: 7).

Ein weiteres Tool aus dieser Kategorie ist HIST, wobei das Tool die Methode der Extraktion von Änderungsdaten mithilfe des VCS' verwendet, wodurch die Änderungen bezüglich von Entitäten im Programmcode analysiert werden können (vgl. Palomba et al. [2013]: 268 f.). Ein Modul mit dem Namen Änderungsdatenextraktor nimmt Daten aus dem Systemlog des VCS', wobei dieses Log ausschließlich Änderungen im Programmcode übergibt (vgl. Palomba et al. [2013]: 269 f.). Da diese Änderungsdaten nicht genau genug sind, um die meisten Code-Smells zu entdecken, extrahiert ein Code-Analyzer von Markos European Project Strukturinformationen über den Programmcode (vgl. Palomba et al. [2013]: 269 f.), sodass die im VCS angezeigten Änderungen mithilfe etablierter Metriken präzisiert werden (vgl. Palomba et al. [2013]: 270). Dadurch können beispielsweise das Hinzufügen, Löschen, Umbenennen oder Verschieben einer Entität untersucht werden (vgl. Palomba et al. [2013]: 270). Da der Code-Analyzer zur Extraktion von Strukturinformationen verwendet wird, ist es erneut ein statisches Code-Analyse-Tool, wobei der Fokus auf die Änderungsdaten gelegt wird. HIST wurde mit den Java-Programmcodes von Apache Ant, Tomcat, jEdit und fünf Android APIs getestet (vgl. Palomba et al. [2013]: 268). Dabei hat es eine Präzision zwischen 61 % und 80 %, sowie einen Recall zwischen 61 % und 100 % erreicht, sodass es durch die Analyse der Änderungsdaten beim Recall bessere Ergebnisse als JDeodorant und DECOR erreicht (vgl. Palomba et al. [2013]: 268 f.).
Darüber hinaus zählt zu dieser Kategorie das Tool EDM, welches unter anderem den für Clean-Code relevanten Code-Smell duplizierter Code in Java-Programmcode analysiert (vgl. Fu and Shen [2015]: 41 f.). Erneut werden die Änderungslogs verschiedener VCS zur Erkennung von Code-Smells verwendet, wobei ein Präprozessor Transaktionsänderungsmengen erstellt (vgl. Fu and Shen [2015]: 43). Auf diesen Änderungsmengen wird ein etablierter Algorithmus, das Assoziationsregel-Mining angewandt, wodurch häufige Muster analysiert werden, woraus eine Menge von Regeln als Output entsteht, die die Kopplung zwischen verschiedenen Entitäten beschreiben (vgl. Fu and Shen [2015]: 43). Assoziationsregel-Mining verwendet das unüberwachte Lernen um Mengen von Merkmalen in einem Input zu finden, die in den Transaktionen einer Datenbank häufig genug auftauchen (vgl. Fu and Shen [2015]: 43). Beim unüberwachten Lernen gibt es im Unterschied zum überwachten Lernen keinen externen Lehrenden, der den Trainingsprozess überwacht, sondern das neuronale Netzwerk bekommt aufgabenunabhängig Daten, wobei er die statistischen Parameter der Verteilung der Daten lernt (vgl. Haykin [2016]: 37). Dadurch

legt das neuronale Netzwerk eine IR über die Merkmale des Inputs an, wodurch es lernt neue Einträge aus der Verteilung zu erstellen (vgl. Haykin [2016]: 37). Transaktionen sind in diesem Kontext eine Menge von Dateien, die im VCS von einer einzelnen Autorin beziehungsweise einem einzelnen Autor in einer Zeiteinheit verändert wurden (vgl. Fu and Shen [2015]: 43). In dem Algorithmus werden alle Änderungen extrahiert, bis keine Änderungen mehr zur früheren Zeiteinheit bestehen, um die Unterschiede zwischen Zeiteinheiten analysieren zu können (vgl. Fu and Shen [2015]: 43). Das Tool wird mithilfe von Änderungshistoriendaten fünf verschiedener großer Java-Projekte evaluiert, wobei diese Projekte Eclipse, jUnit, Guava, Closure Compiler und Maven sind (vgl. Fu and Shen [2015]: 44). Dabei umfassen die Daten, je nach Projekt, die Commits von 4 bis 13 Jahren (vgl. Fu and Shen [2015]: 44). Für die drei untersuchten Code-Smells wird bei der Evaluation eine Präzision, je nach Code-Smell und untersuchtem Java-Programm, zwischen 50 % und 100 % erreicht, wobei ebenfalls ein Recall zwischen 50 % und 100 % erreicht wird (vgl. Fu and Shen [2015]: 46 f.). Dabei werden insbesondere bessere Ergebnisse im Vergleich zu den Tools JDeodorant und DCPP erzielt (vgl. Fu and Shen [2015]: 47).

3.4. Code-Smell-Erkennung durch Textanalyse

Neben den vorigen Ansätzen zur Code-Smell-Erkennung gibt es darüber hinaus die Möglichkeit, Code-Smells direkt im Programmcode zu suchen. Ein Tool aus dieser Kategorie ist TACO, welches den Code-Smell lange Methode in Java-Programmcode erkennt (vgl. Palomba [2015]: 769). Dabei wird eine etablierte Technik der Literaturanalyse verwendet, um eine Methode in bedeutungsvolle Blöcke aufzuteilen, woraufhin TACO die Bezeichner und Kommentare aus den Code-Blöcken entfernt (vgl. Palomba [2015]: 769). Durch die Verwendung einer etablierten Methodik zur Analyse semantischer Ähnlichkeiten werden die einzelnen Codeblöcke evaluiert, wobei Änhlichkeitswerte für jeden Block berechnet und in einer Matrix gespeichert werden (vgl. Palomba et al. [2013]: 769). Jeder Eintrag gibt dabei die Ähnlichkeit zwischen zwei verschiedenen Codeblöcken an (vgl. Palomba et al. [2013]: 769). Wird ein empirisch-evaluierter Schwellenwert bei der Ähnlichkeit zwischen zwei Blöcken unterschritten, so wird eine lange Methode erkannt (vgl. Palomba [2015]: 769). Zur Evaluation des Tools werden die drei Java-Projekte Apache Cassandra, Apache Xerces und Eclipse Core verwendet, wo-

bei TACO beim Testen auf diesen Programmcodes eine Präzision von 65 %, sowie einen Recall von 61 % erreicht (vgl. Palomba et al. [2013]: 770). Im Test auf den gleichen Java-Projekten erreicht DECOR demgegenüber eine Präzision von 52 % und einen Recall von 74 %, sodass sich die Textanalyse als Methodik zur Code-Smell-Erkennung behauptet hat (vgl. Palomba [2015]: 770).
Der Teil des statischen Analyse-Tools iPlasma aus Abschnitt 3.1, der durch die Verwendung des Textvergleichstools DUDE duplizierten Code erkennt, würde ebenfalls zu dieser Kategorie passen (vgl. Lanza and Marinescu [2006]: 177). Das liegt an der Verwendung des Textvergleichstools DUDE unmittelbar auf dem Programmcode (vgl. Lanza and Marinescu [2006]: 177), sodass der Algorithmus auch unabhängig von den Strukturinformationen durch den Parser funktionieren sollte und iPlasma eine Mischform darstellt.

4. Forschungsfragen und Konzeptentwicklung

Nachdem der Forschungsstand bezüglich Code-Smell Analyse-Tools dargestellt wurde, da in der Literatur bisher keine Clean-Code-Analyse-Tools existieren, wird in Abschnitt 4.1 auf die Gemeinsamkeiten der beiden Konzepte eingegangen. Darüber hinaus werden die in einem statischen Code-Analyse-Tool methodisch umsetzbaren Clean-Code-Heuristiken erläutert und kategorisiert, um ein Konzept für die Implementierung zu erhalten. Im Anschluss werden in Kapitel 4 aus diesen Erkenntnissen Hypothesen zur Beantwortung der Forschungsfrage abgeleitet.

4.1. Clean-Code: Welche Teile sind messbar? — Die Konzeptionalisierung hin zu einem maschinenlesbaren Ansatz

In Unterabschnitt 2.1.7 wurde bereits auf die Schwierigkeiten der fehlenden exakten Definition des Clean-Code-Begriffs hingewiesen. Im Rahmen der vorliegenden Arbeit wird der Versuch unternommen das Konzept mithilfe von Heuristiken aus Martin [2009] messbar zu machen, die in diesem Kapitel genannt werden. Dabei werden Bezüge zum verwandten Code-Smell-Begriff hergestellt, um die Überschneidungen der beiden Ansätze zu demonstrieren. In diesem Kapitel werden außerdem aufgrund der großen Menge der Clean-Code-Heuristiken ausschließlich diejenigen Heuristiken genannt, für die es Hinweise auf syntaktische beziehungsweise statische Analysierbarkeit in einer objektorientierten Programmiersprache gibt, ohne dabei einen mathematischen Beweis durchzuführen. Eine syntaktisch analysierbare Heuristik meint dabei eine Heuristik, deren Überprüfung mithilfe der Strukturinformationen, die sich aus der Syntaxanalyse des Parsers gewinnen lassen, möglich ist. Damit fungiert dieses Kapitel als Konzept für den Praxisteil der vorliegen-

den Arbeit (siehe Abschnitt 5.1, indem die Heuristiken innerhalb eines statischen Code-Analyse-Tools messbar gemacht werden. Bei der Einordnung bezüglich der syntaktischen Analysierbarkeit ist die Existenz des Konstrukts, auf das sich die jeweilige Heuristik bezieht, in der objektorientierten Programmiersprache der jeweiligen Clean-Code-Analyse vonnöten, wobei diese Prämisse nicht ausnahmslos für alle objektorientierten Programmiersprachen gelten kann. Diese Unterscheidung ist wichtig, da wie in Unterabschnitt 2.1.4 beschrieben, nur Strukturinformationen einer syntaktischen Analyse während einer statischen Code-Analyse abstrahiert und keine semantischen Beziehungen zwischen Programmteilen analysiert werden können. In diesem Kontext bezieht sich 'Semantik' auf die semantischen Aspekte der untersuchten Programmiersprache. Dabei wird von einer syntaktischen Analysierbarkeit der jeweiligen Heuristik ausgegangen, wenn sie keinen bedeutungsvollen Kontext besitzt, der eine semantische Analyse erfordern würde. Es lässt sich unterscheiden, ob die Semantik innerhalb der Programmiersprache gemeint ist, oder die Semantik innerhalb der natürlichen Sprache des Anwendenden, die beispielsweise bei Heuristiken in Bezug auf Kommentare oder Bezeichner eine Rolle spielt. In dem vorliegenden Kapitel ist dabei der letztere Fall gemeint, insofern nicht explizit darauf hingewiesen wird. Das Clean-Code-Konzept wird dabei in der vorliegenden Arbeit als Verschärfung der Syntax der jeweiligen Programmiersprache, die analysiert werden soll, durch die syntaktisch analysierbaren Clean-Code-Heuristiken begriffen. Dadurch wird dem Clean-Code-Begriff eine klarere Bedeutung zugewiesen. Darüber hinaus werden in diesem Kapitel Heuristiken genannt, deren Negationen sich äquivalent zu einem der in Abschnitt 2.2 genannten Code-Smells verhalten.

4.1.1. Größe von Entitäten im Clean-Code

Eine erste Gruppe von Clean-Code-Heuristiken lässt sich mit ihrem Bezug zu den Größen von Entitäten zusammenfassen. Die Heuristiken werden in Tabelle 4.1 beschrieben.

Nr.	Kat.	Beschreibung/Anweisung	Ref.	Code-Smell	Syntax-Analyse
1	M	Funktionen erfüllen nur eine Aufgabe gleichzeitig (außer die ausgeführten Aufgaben liegen genau eine Abstraktionsebene unter der durch den Methodennamen beschriebenen Ebene).	S. 65f	-	nein
2	M	Anweisungen innerhalb einer Funktion beziehen sich auf nur eine Abstraktionsebene.	S. 67	-	nein
3	M	Funktionen sind von oben nach unten lesbar, was durch Aufrufe von Methodennamen erreicht wird, die lediglich auf der nächsttieferen Abstraktionsebene angesiedelt sind.	S. 67	-	nein
4	M	Funktionen dürfen nur etwas tun, oder Informationen bereitstellen. Nicht beides.	S. 77	-	nein
5	K	Klassen dürfen nur eine Verantwortlichkeit besitzen (Single-Responsibility-Prinzip).	S. 176	-	nein
6	K	Klassen dürfen nur eine kleine Anzahl von Instanzvariablen verwenden.	S. 178	Große Klasse	zum Teil
7	K	Instanzvariablen müssen in möglichst vielen Methoden der Klasse vorkommen.	S. 178	Neid	zum Teil

Tabelle 4.1.: Clean-Code Größe aus Martin [2009]

Legende:[M] Methoden, [K] Klassen, [V] Variablen, [A] Allgemein, [G] Global

Dabei ist in der ersten Spalte eine Nummerierung (Nr.) angegeben, um die Heuristiken auseinanderzuhalten, da sie im Vergleich zu Code-Smells keine eigenen Bezeichnungen besitzen. In der zweiten Spalte ist die Kategorie (Kat.) der Heuristik dargestellt, die sich auf die Entität im Programmcode bezieht, die analysiert wird. Dabei kann die Kategorie einen oder mehrere der Werte Variablen, Methoden, Klassen und Allgemein/Global annehmen. In der dritten Spalte ist eine kurze Beschreibung der Heuristik zu finden, während in der vierten Spalte die Referenz (Ref.) aus Martin [2009] angegeben ist. In der fünften Spalte wird ein Code-Smell angegeben, falls eine Negation eines Code-Smells äquivalent zu einer Clean-Code-Heuristik anzusehen ist. Zusätzlich wird in der sechsten Spalte eine Einschätzung

angegeben, ob die jeweilige Heuristik in einer objektorientierten Programmiersprache syntaktisch analysierbar ist, da dieser Punkt entscheidend für die Messbarkeit in einem statischen Code-Analyse-Tool ist. Tabelle A.7, Tabelle A.8, Tabelle A.9, Tabelle A.10, Tabelle A.11, Tabelle A.12 und Tabelle A.13 aus dem Anhang besitzen einen äquivalenten Aufbau.
Bei den in Tabelle 4.1 dargestellten Heuristiken sind lediglich die beiden Heuristiken mit den Nummern 6 beziehungsweise 7 bezüglich Klassen syntaktisch analysierbar. Nach der Heuristik mit der Nummer 7 dürfen Klassen nur eine geringe Anzahl von Instanzvariablen verwenden, wobei die Instanzvariablen nach der Heuristik mit der Nummer 8 in möglichst vielen Methoden der Klasse vorkommen müssen (vgl. Martin [2009]: 178). Klassen existieren dabei in vielen objektorientierten Programmiersprachen, wobei sie als Verbünde von Variablen definiert sind, die oft auch Methoden implementieren (vgl. Poetzsch-Heffter [2000]: 44). Variablen können in (objektorientierten) Programmiersprachen einem Block zugeordnet werden (vgl. Kak [2003]: 303 ff.), wobei sie innerhalb von Klassen auch Attribute genannt werden (vgl. Poetzsch-Heffter [2000]: 37). Sie sind sozusagen lokale Variablen in einem Objektblock (vgl. Poetzsch-Heffter [2000]: 37). Ein Block ist dabei eine Folge von Deklarationen und Anweisungen, die beispielsweise in Java von geschweiften Klammern eingeschlossen wird (vgl. Poetzsch-Heffter [2000]: 29) und einen eigenen Geltungsbereich besitzt (vgl. Kak [2003]: 304). Somit lassen sich die Attribute einer Klasse zählen. Ein Problem bei der Syntax-Analyse der Heuristiken ist jedoch die Abwesenheit eindeutiger Schwellenwerte für die Menge von Attributen (Nr. 7) beziehungsweise Datenab- und Methodenaufrufe (Nr. 8), wodurch die Heuristiken nicht ohne Weiteres syntaktisch analysierbar sind. Die Negationen der Code-Smells große Klasse beziehungsweise Neid sind dabei sehr ähnlich zu den genannten Heuristiken.

4.1.2. Clean-Code und Zugriffsmodifikatoren

Tabelle A.7 behandelt diejenigen Heuristiken, die sich mit Zugriffsmodifikatoren befassen. Zugriffsmodifikatoren sind die Teile vieler objektorientierter Programmiersprachen, die die Zugriffskontrolle in Bezug auf Attribute und Methoden von außerhalb einer Klasse regeln (vgl. Poetzsch-Heffter [2000]: 86). Dabei können sie die Werte „public“, „private“ oder „protected“ annehmen (vgl. Kak [2003]: 71). Die Heuristik mit der Nummer 8 bezüglich Methoden fordert zum einen die Markierung von überladenen Konstruktoren mit dem Zugriffsmodifikator „priva-

te“, zum anderen die Verwendung der überladenen Konstruktoren durch als „public“ markierte, sogenannte Fabrik-Methoden (Martin [2009]: 55). Diese Fabrik-Methoden stellen dabei während ihres Aufrufs einen aussagekräftigeren Namen als gewöhnliche Konstruktoraufrufe zur Verfügung (vgl. Martin [2009]: 55). Die Fabrik-Methoden besitzen keine andere Aufgabe, als Konstruktoren aufzurufen (vgl. Ullenboom [2023]: 872). Wird ein Konstruktor überladen, so gibt es den Konstruktor mit dem gleichen Bezeichner bereits, wobei sich die Übergabeparameter zu anderen Konstruktoren unterscheiden (vgl. Poetzsch-Heffter [2000] 66). Somit ist der erste Teil der Heuristik mit der Nummer 8 syntaktisch analysierbar, da die Strukturinformationen nach bereits definierten Konstruktoren der Klasse untersucht werden können. Durch sie kann ein Überladen der Konstruktoren festgestellt werden, während der zweite Teil der Heuristik aufgrund des bedeutungsvollen Kontexts der Bezeichner der Fabrik-Methoden nicht syntaktisch analysierbar ist. Zwar gibt es keinen Code-Smell, dessen Negation äquivalent zu dieser Heuristik ist, doch nennt Fowler [2019] den gleichen Ansatz als Refactoring-Strategie (vgl. Fowler [2019]: 334), wodurch die genannte Heuristik zur Reduktion von Code-Smells beiträgt. Die Heuristiken mit den Nummern 9 beziehungsweise 10 sind äquivalent zum ersten Teil der Heuristik mit der Nummer 8 analysierbar, indem der Zugriffsmodifikator analysiert wird, insofern Zugriffsmodifikatoren ein Teil der Grammatik der analysierten Programmiersprache sind.

4.1.3. Bezeichner von Entitäten im Clean-Code

Tabelle A.8 zeigt alle Clean-Code-Heuristiken, die sich mit den Bezeichnern von Entitäten befassen, womit Variablen, Methoden und Klassen zugleich gemeint sind. Die Heuristik mit der Nummer 11 verlangt zweckbeschreibende Namen, wobei der Inhalt und die Intention der jeweiligen Entität durch den Bezeichner beschrieben werden müssen (vgl. Martin [2009]: 50). Diese Heuristik ist nur semantisch analysierbar, wobei sie große Ähnlichkeiten zur Negation des Code-Smells des mysteriösen Namens besitzt. Da dieser Code-Smell von Fowler [2019] neuer ist, als der Clean-Code-Ansatz, zeigt sich, wie die beiden Ansätze sich gegenseitig beeinflussen und ergänzen.
Eine teilweise syntaktisch analysierbare Heuristik ist die Heuristik mit der Nummer 13 bezüglich suchbarer Namen, wobei ein Bezeichner einer Entität möglichst gut im Code auffindbar ist, insofern längere Namen ohne Ziffern verwendet werden (vgl. Martin [2009]: 51). Diese Heuristik besitzt die wiederkehrende Schwierigkeit

keinen exakten Schwellenwert zu besitzen, ab wie vielen Buchstaben ein Bezeichner als suchbar gelten kann. Um Beschränkungen bei der Konstruktion von Bezeichnern zu formulieren, eignen sich besonders die regulären Ausdrücke besonders gut. Eine Untergrenze sollte die Verwendung von wenigstens mehr als ein Zeichen sein, die als Annäherung an die Heuristik angesehen wird.
Eine weitere Heuristik (Nr. 14) verbietet die Verwendung der ungarischen Notation für Bezeichner von Entitäten, bei der der Datentyp der Entität im Bezeichner selbst angegeben wird (vgl. Martin [2009]: 52f.). Diese Heuristik ist syntaktisch analysierbar, insofern der Datentyp der Variable, der Rückgabetyp der Funktion oder der Klassenname einen Substring des Bezeichners der jeweiligen Entität darstellt. Alle Heuristiken, die sich auf Substringprobleme beziehen, sind dabei syntaktisch analysierbar, was die Heuristiken mit der Nummer 15 und teilweise der Nummer 24 einschließt. Bei der Heuristik Nummer 26 lässt sich Konsistenz in Bezug auf Methodennamen auch als eine konsistente Nutzung einer einheitlichen Namenskonvention (beispielsweise CamelCase) auffassen, sodass sie so syntaktisch analysierbar wird.
Darüber hinaus betont eine weitere Heuristik mit der Nummer 17 die Konvention von Schleifenzählern, deren Variablenbezeichner aus einem einzelnen Buchstaben bestehen müssen, wobei der Name weder "l" noch "I" sein darf (vgl. Martin [2009]: 54), da sich diese Buchstaben in vielen Schriftarten sehr ähnlich sehen. Viele objektorientierte Programmiersprachen verwenden bei for-Schleifen einen sogenannten Init-Ausdruck als erste Schleifenbedingung, in dem die Zuweisung eines Werts an eine Zählervariable stattfindet (vgl. Poetzsch-Heffter [2000]: 30). Ein Ausdruck ist dabei so definiert entweder eine Zuweisung oder ein Funktionsaufruf zu sein (vgl. Poetzsch-Heffter [2000]: 28). Diese Heuristik ist innerhalb von for-Schleifen dieser Art syntaktisch analysierbar, da der Bezeichner des Schleifenzählers innerhalb des Init-Ausdrucks zu finden ist, wobei der Kontext eindeutig ist. Da while- und do-while-Schleifen üblicherweise lediglich einen booleschen Ausdruck innerhalb der Klammern besitzen (vgl. Poetzsch-Heffter [2000]: 30), ist hier der Kontext nicht eindeutig. Auch forEach-Schleifen verwenden keinen Init-Ausdruck (vgl. Poetzsch-Heffter [2000]: 30), sodass hier das Gleiche gilt.
Die Heuristik mit der Nummer 18 fordert die Verwendung von konkreten Substantiven für einen Klassenbezeichner (vgl. Martin [2009]: 54). Da die Validierung, ob ein Bezeichner ein Substantiv ist, eine semantische Bedeutung enthält, lässt sich die Heuristik nicht syntaktisch analysieren. Es könnte alternativ getestet werden,

ob der Klassenbezeichner mit einem Großbuchstaben beginnt, wie es bei Substantiven in vielen natürlichen Sprachen der Fall ist, um sich der Heuristik anzunähern. Äquivalent könnte eine Testung auf den Beginn mit einem Kleinbuchstaben bei der Heuristik mit der Nummer 19 erfolgen, die sich auf Methoden bezieht, wobei ein Funktionsbezeichner mit einem Verb beginnen muss (vgl. Martin [2009]: 54).

4.1.4. Formatierungskonventionen des Clean-Codes

In Tabelle A.9 werden Heuristiken in Bezug zur Formatierung erläutert. Bei der Heuristik mit der Nummer 28 sind Methoden beziehungsweise Funktionen eine Konvention, die zwischen 20 und 100 Zeilen lang sind, wobei sie in keinem Fall länger als 150 Zeilen lang sein dürfen (vgl. Martin [2009]: 64). In den allermeisten Programmiersprachen wird der Input aus der Quelltextdatei in Zeilen organisiert (vgl. Fraser and Hanson [1995]: 103). Compiler sind in der Lage die Zeilennummer aus Quelltextdateien numerisch zu lesen, wie beispielsweise übliche C++-Compiler (vgl. Erlenkötter [2018]: 279), was äquivalent für Interpreter gelten sollte. Diese Heuristik ist eine Konkretisierung des Code-Smells lange Methode, wobei klare Schwellenwerte angegeben werden, die sich daher syntaktisch analysieren lassen, indem die Zeilen von Beginn bis zum Ende einer Funktion gezählt werden. Ähnlich verhält es sich mit der Heuristik, wonach jede Zeile einer Funktion nicht mehr als 150 Zeichen breit sein darf (vgl. Martin [2009]: 64), wobei in jeder Zeile die Spalten gezählt werden müssen. Diese Vorgehensweisen lassen sich auf die Heuristiken mit den Nummern 34 und 35 bezüglich der allgemeinen/globalen Kategorie verallgemeinern, da sie sich lediglich auf die Quelltextdatei statt auf eine Funktion beziehen (vgl. Martin [2009]: 111 ff.).
Eine weitere Heuristik (Nr. 30) bezieht sich auf die Einrückungstiefe innerhalb von Funktionen, die 1 oder 2 Ebenen nicht überschreiten darf (vgl. Martin [2009]: 65). In C++ werden beispielsweise verschiedene Stile von Einrückungen verwendet, wobei der jeweilige Einrückungsstil nach Präferenz ausgewählt wird (vgl. Stroustrup [2015]: 259), was sich auf andere objektorientierte Programmiersprachen übertragen lassen sollte. Die Heuristik ist syntaktisch analysierbar, wenn die Konsistenz des Einrückungsstils in Form der Menge von verwendeten Leerzeichen überwacht und anschließend in diesem Stil die Einrückungstiefe innerhalb von Funktionen beziehungsweise Methoden gezählt wird. Da sich die Heuristiken mit den Nummern 39, 40, 41 und 42 bezüglich Methoden beziehungsweise Klassen ebenfalls auf Einrückungen beziehen, sind sie genauso syntaktisch analysierbar. Dabei wird

durch die Heuristiken für jede Entität eine Einrückungstiefe vorgeschrieben (vgl. Martin [2009]: 123).
Abseits von Einrückungen gibt es die Heuristik mit der Nummer 31 bezüglich Methoden, nach der Blöcke von Anweisungen nicht mehr als eine Zeile beanspruchen dürfen (vgl. Martin [2009]: 65). Werden die Zeilen innerhalb des Blocks der Funktion gezählt, so lässt sich diese Heuristik während einer Syntax-Analyse überprüfen. Ähnlich zur Negation des Code-Smells Switch-Befehle verhält sich die Heuristik mit der Nummer 32 in Bezug auf Methoden, die die Auslagerung von switch-Anweisungen in eine eigene Klasse fordert (vgl. Martin [2009]: 68 f.). Dabei sollen zusätzlich die in der switch-Anweisung verschachtelten case-Anweisungen jeweils nur aus einem Aufruf einer Methode bestehen, die die aktuelle Klasse von ihrer Elternklasse erbt (vgl. Martin [2009]: 68 f.). Da sowohl switch-Anweisungen (vgl. Poetzsch-Heffter [2000]: 31) als auch Vererbung, die es Kindklassen erlaubt auf die Methoden und Attribute der Elternklasse zuzugreifen (vgl. Kak [2003]: 31), in vielen objektorientierten Programmiersprachen unterstützt werden (vgl. Zeppenfeld [2004]: 26, 44), ist diese Heuristik syntaktisch analysierbar. Dabei muss neben der Nichtüberschreitung der Zeilengrenze der case-Anweisungen ebenfalls geprüft werden, ob die Ausdrücke in den case-Anweisungen Aufrufe der Funktionen aus der Elternklasse der Klasse sind, in der sich die switch-Anweisung befindet.
Eine weitere Heuristik (Nr. 36) bezüglich Methoden empfiehlt in den Funktionsköpfen den Verzicht auf ein Leerzeichen zwischen der öffnenden Klammer von Funktionen, die den Beginn der Übergabeparameterdeklaration darstellt, sowie dem Bezeichner der Funktion (vgl. Martin [2009]: 120). Da die Heuristik sich auf ein einzelnes Zeichen im Programmcode bezieht, ist die Heuristik syntaktisch analysierbar, indem die syntaktische Analyse die Funktion erkennt und die Anzahl der Leerzeichen zwischen Code-Entitäten zusätzlich speichert. Dabei dürfen Leerzeichen nicht während der lexikalischen Analyse verworfen werden. Diese Vorgehensweise lässt sich erneut auf ähnliche Heuristiken bezüglich Methoden beziehungsweise Variablen übertragen (Nr. 37 beziehungsweise 38), die Leerzeichen zwischen einzelnen Wörtern erfordern (vgl. Martin [2009]: 120).

4.1.5. Funktionsparameterübergabe im Clean-Code-Konzept

Tabelle A.10 beschreibt ausschließlich Heuristiken bezüglich Methoden, die über eine Verbindung zur Analyse von Funktionsparametern verfügen. Dabei gibt es die Heuristik mit der Nummer 43, die sich auf die Menge der Übergabeparameter

bezieht, wobei weniger Argumente als sauberer im Vergleich zu mehr Übergabeparametern eingeschätzt werden (vgl. Martin [2009]: 71). Da die Anzahl der Übergabeparameter zählbar ist, ist diese Heuristik syntaktisch analysierbar.
Beziehen sich die Heuristiken auf die Datentypen der Übergabeparameter, wie bei den Heuristiken mit der Nummer 45, die keine boolean-Argumente zulässt (vgl. Martin [2009]: 72), oder mit der Nummer 44, die Objekte als Übergabeparameter wünscht (vgl. Martin [2009]: 74), sind die Heuristiken tendenziell eher in typisierten Sprachen analysierbar, da bei ihnen Werte häufiger statisch statt dynamisch an einen Bezeichner gebunden werden müssen (vgl. Gantenbein [1991]: 31). Dabei muss jedoch im Einzelfall für jede konkrete Programmiersprache entschieden werden. Hier ist das in Unterabschnitt 2.1.4 genannte Problem der dynamischen Bindung entscheidend. Außerdem gibt es die sogenannte Typunsicherheit in nicht-typisierten Programmiersprachen, da in nicht-typisierten Sprachen jede Variable jeden Datentyp referenzieren kann (vgl. Poetzsch-Heffter [2000]: 76 f.). Durch die genannten Gründe kann für einen Funktionsparameter der Datentyp je nach Programmiersprache und Ausführungskontext statisch nicht ermittelbar sein. Darüber hinaus sind die genannten Heuristiken mit den Nummern 43, 44 und 45 dabei zusammengenommen sehr ähnlich zur Negation des Code-Smells lange Parameterliste, da sie jeweils einen Teil der Negation des Code-Smells ausmachen.
Die Heuristiken mit den Nummern 49 beziehungsweise 50, die die Nutzung von NULL-Pointern als Übergabeparameter an beziehungsweise als Rückgabetyp von einer Funktion (vgl. Martin [2009]: 147 f.) verbieten, sind ebenfalls (teilweise) syntaktisch analysierbar. NULL ist dabei ein besonderer Pointer, der auf kein Objekt zeigt (vgl. Poetzsch-Heffter [2000]: 24). Wird ein NULL-Pointer von einer Funktion zurückgegeben, ist die Heuristik syntaktisch analysierbar. Das gilt jedoch nur, insofern der NULL-Pointer nicht in einer Variable zwischengespeichert ist und von dem Variablenbezeichner verborgen wird, da die NULL-Referenz in Referenzvariablen gespeichert werden kann (vgl. Ullenboom [2023]: 254). Dabei ist es ein einfacher String-Vergleich zwischen dem Bezeichner für den NULL-Pointer, der von der Programmiersprache abhängt (vgl. Poetzsch-Heffter [2000]: 24), sowie dem zurückgegebenen Bezeichner der Funktion. Beispielsweise ist der Bezeichner für den NULL-Pointer in der Programmiersprache Java das Schlüsselwort „null“ (vgl. Zeppenfeld [2004]: 26). Das gleiche gilt ebenfalls für die Übergabe von NULL-Pointern an Funktionen.

4.1.6. Clean-Code-Kommentare

Die Tabelle A.11 enthält alle Clean-Code-Heuristiken, die sich mit Kommentaren im Programmcode befassen. Eine Heuristik (Nr. 52) bezüglich der allgemeinen/globalen Kategorie fordert saubereren Code, sodass keine informierenden Kommentare mehr für das Verständnis nötig sind (vgl. Martin [2009]: 52). Da die Definition von informierenden Kommentaren nicht eindeutig beziehungsweise ein semantischer Bezug ist, lässt sich die Heuristik nicht syntaktisch analysieren. Dabei verhält sich die Heuristik äquivalent zum Code-Smell namens Kommentare.
Eine weitere Heuristik (Nr. 62) bezüglich der allgemeinen/globalen Kategorie empfiehlt den Verzicht auf Kommentare, die sowohl in der gleichen Zeile, als auch unmittelbar hinter schließenden geschweiften Klammern von Blöcken platziert sind (vgl. Martin [2009]: 101). Diese Heuristik ist syntaktisch analysierbar, indem die Worte hinter den schließenden Klammern neben der Zeilenplatzierung darauf überprüft werden, ein Kommentar zu sein. Dabei darf die lexikalische Analyse Kommentare nicht verwerfen. Darüber hinaus ist die Heuristik nur syntaktisch analysierbar, insofern die Programmiersprache geschweifte Klammern zulässt, was beispielsweise in der objektorientierten Programmiersprache Python nicht der Fall ist, da die Einrückung die semantische Bedeutung der geschweiften Klammern ersetzt (vgl. Guttag [2016]: 16 f.).
Zusätzlich befasst sich die Heuristik (Nr. 64) bezüglich Methoden mit der Verwendung von Funktionsheader-Kommentaren, die nicht verwendet werden sollten (vgl. Martin [2009]: 102). Die Heuristik ist mit Einschränkungen syntaktisch analysierbar, da Funktionsheader-Kommentare bezüglich ihrer Platzierung unmittelbar vor einer Funktion erkannt werden können. Ob es sich dabei um einen Funktionsheader-Kommentar handelt, ist jedoch eine semantische Bedeutung. Java verwendet beispielsweise eine spezielle Syntax, um einen Dokumentationskommentar zu markieren (vgl. Schildt [2007]: 33). Annotationen, die standardmäßig in Funktionsheader-Kommentaren auftauchen, wie beispielsweise die Annotationsankündigung durch "@param", die im Kommentar die Methodenparameterdokumentation ankündigt (vgl. Schildt [2007]: 991), geben Indizien in Richtung eines Funktionsheader-Kommentars und können syntaktisch analysiert werden.

4.1.7. Clean-Code — Was nicht geht

Tabelle A.12 fasst Heuristiken zusammen, die bestimmte Stile von Formulierungen verbieten. Ein Beispiel hierfür sind die Heuristiken mit den Nummern 65 und 69 bezüglich Methoden, die die Verwendung von Duplikaten in Funktionen (vgl. Martin [2009]: 80) beziehungsweise Klassen (vgl. Martin [2009]: 342) untersagen. Wie bezüglich des Code-Smells duplizierter Code in Abschnitt 2.2 beschrieben, sollten öfter vorkommende Programmteile extrahiert werden. Beispielsweise sollte die Extraktion dabei in eigene kleinere Funktionen erfolgen. Diese Heuristik ist prinzipiell mit vielen String-Vergleichen syntaktisch analysierbar, wobei wiederkehrende Wörter aus der jeweiligen Programmiersprache für viele Duplikate sorgen, die von dieser Heuristik nicht gemeint sind. Wie bereits in Abschnitt 3.2 beleuchtet, besitzt die Erkennung des Code-Smells duplizierter Code einen semantischen Aspekt, da die Auswahl von nützlichen Duplikaten aus einer großen Menge von Duplikaten bedeutungsvoll ist, sodass statische Code-Analyse-Tools den Code-Smell nicht sinnvoll erkennen können.
Darüber hinaus empfiehlt die Heuristik mit der Nummer 66 bezüglich Methoden einen Verzicht auf sogenannte goto-Anweisungen (vgl. Martin [2009]: 81). Da goto-Anweisungen in vielen objektorientierten Programmiersprachen, wie beispielsweise C++ (vgl. Zeppenfeld [2004]: 114) oder Java (vgl. Schildt [2007]: 32), zu den Schlüsselwörtern gehören, ist die Heuristik syntaktisch analysierbar, falls das Schlüsselwort in der vorliegenden Programmiersprache existiert.
Des Weiteren untersagt die Heuristik mit der Nummer 67 bezüglich Klassen die Bereitstellung sogenannter Getter- und Settermethoden zum Zugriff auf die Attribute einer Klasse (vgl. Martin [2009]: 129f.). Getter- und Setter sind dabei Methoden, die Daten von einem Objekt abfragen (Getter) beziehungsweise Daten einem Attribut zuweisen (Setter) (vgl. Schiedermeier [2013]: 140). Dabei folgen die Bezeichner dieser Methoden bekannten Namenskonventionen (vgl. Schiedermeier [2013]: 140). Ist der Bezeichner der Klasse sowie die Namenskonvention der Programmiersprache bekannt, so ist die Heuristik mithilfe von String-Vergleichen syntaktisch analysierbar.
Außerdem fordert die Heuristik mit der Nummer 68 bezüglich Methoden den Verzicht auf Funktionen, die nicht aufgerufen werden (vgl. Martin [2009]: 340), was mit der Negation des Code-Smells toter Code übereinstimmt. Um alle Funktionen dahingehend zu überprüfen, ob sie genutzt werden, um ungenutzte Methoden beziehungsweise sogenannten Toten Code zu identifizieren, ist beispielsweise in

der objektorientierten Programmiersprache PHP eine dynamische Code-Analyse vonnöten, da Laufzeitinformationen benötigt werden (vgl. Boomsma et al. [2012]: 512). Wie in Unterabschnitt 2.1.4 beschrieben haben statische Code-Analyse-Tools große Probleme beim Erkennen von totem Code, sodass eine semantische Analyse bezogen auf die Semantik der Programmiersprache nötig ist. Ein dynamisches Code-Analyse-Tool hat daher bei der Analyse große Vorteile, indem es den alle Code-Pfade ausführt und nicht ausgeführte Code-Pfade erkennt. Jedoch kann in der statischen Code-Analyse eine Annäherung mithilfe von String-Vergleichen der aufgerufenen Funktionen und den definierten Funktionen gelingen. Gerade die Möglichkeit des Methodenüberladens in vielen Programmiersprachen macht diese Annäherung jedoch sehr aufwendig, da alle aktuellen Typen der Variablen bei einem Methodenaufruf geprüft werden müssen, um die aufgerufene Methode ausfindig zu machen.
Die Heuristiken mit den Nummern 70 und 71 bezüglich Methoden, die sich auf die Verwendungen von Vergleichen in dafür speziell erstellten booleschen Methoden beziehen (Nr. 70), beziehungsweise auf den Verzicht bestimmter Operatoren (Nr. 71) (vgl. Martin [2009]: 356) sind syntaktisch analysierbar. Vergleichsoperatoren sind ein Bestandteil vieler Programmiersprachen, wie beispielsweise von C++ (vgl. Stroustrup [2015]: 47). Kommt ein solcher Vergleich vor, muss zur Überprüfung der Heuristik mit der Nummer 70 getestet werden, ob der Rückgabetyp der aktuellen Funktion ein Boolean-Datentyp ist sowie ob der Vergleich die einzige Anweisung in der Funktion ist. Auch die Heuristik mit der Nummer 71 ist syntaktisch analysierbar, da der logische Operator, der der Negation dient, ein Bestandteil der meisten Programmiersprachen ist, wie beispielsweise in der Programmiersprache C (vgl. Wolf [2019]: 147), wodurch sein Vorkommen in diesen Sprachen syntaktisch überprüft werden kann.

4.1.8. Platzierung von Entitäten im Clean-Code

Die letzte Tabelle A.13 befasst sich mit den Platzierungen von Entitäten im Programmcode. Die Heuristik mit der Nummer 72 bezüglich Klassen fordert die Deklaration von Attributen ganz oben in einer Klasse, ohne dabei Leerzeilen oder Kommentare vor den Variablendeklarationen zu erlauben (vgl. Martin [2009]: 113). Da Informationen über die Platzierungen von Leerzeilen beziehungsweise Kommentaren sowie den Platzierungen von Entitäten im Programmcode aus den Strukturinformationen der syntaktischen Analyse entnommen werden können, ist die

Heuristik syntaktisch analysierbar. Dabei müssen die Platzierungen der Entitäten dahingehend geprüft werden, ob vor beziehungsweise zwischen den Variablendeklarationen in dem Block der jeweiligen Klasse keine anderen Entitäten deklariert beziehungsweise definiert werden. Außerdem muss darauf geprüft werden, ob Kommentare beziehungsweise Leerzeilen vor oder zwischen den Variablendeklarationen vorkommen.
Sehr ähnlich funktioniert die Überprüfung der Heuristik mit der Nummer 74 bezüglich Methoden, wobei nicht auf Leerzeilen und Kommentare geprüft werden muss und sich die Analyse auf den Block einer Funktion und nicht auf die Analyse einer Klasse bezieht (vgl. Martin [2009]: 114). Ähnlich verhält sich eine Heuristik (Nr. 78) bezüglich Klassen, die zuerst die Deklaration von public static-, dann von private static- und schlussendlich von private Variablen fordert (vgl. Martin [2009]: 173). Dabei müssen neben der Eigenschaft als Attribute noch die statischen Eigenschaften sowie die Zugriffsmodifikatoren überprüft werden, sodass geprüft werden kann, ob die genannten geordneten Folgen von Attributdeklarationen existieren. Außerdem fällt die Überprüfung der Leerzeilen beziehungsweise der Kommentare bei dieser Heuristik weg.
Eine weitere Heuristik mit der Nummer 73 bezüglich Klassen fordert die Definition von Methoden in einer Klasse mit jeweils einem Abstand von genau einer Zeile, wobei sie erst unter den Attributen (Heuristik Nr. 72), definiert werden dürfen (vgl. Martin [2009]: 113). Hier muss zuerst die erste Methodendefinition im Block einer Klasse gesucht werden. Sind vor der ersten gefundenen Methode nur Attribute deklariert worden, muss zwischen der aktuellen Methode und der jeweils nächsten Methode eine Leerzeile vorhanden sein. Dabei darf zwischen den Methoden keine andere Entität entdeckt werden. Zusätzlich müssen ebenfalls alle Entitäten bis zum Ende des Klassenblocks geprüft werden, da keine weiteren Methodendefinitionen außerhalb dieser Folge von Methodendefinitionen vorkommen darf.
Außerdem gibt es die Heuristik mit der Nummer 76 in Bezug auf Methoden, die die Definition von Funktionen unter dem Block fordert, der die Funktion aufruft (vgl. Martin [2009]: 116 f.), wobei sie ebenfalls syntaktisch analysierbar ist. Hierzu muss bei der Erkennung eines Funktionsaufrufs die Entität des aktuellen Blocks (oder der globale Block) identifiziert werden. Im Anschluss daran muss in dem Block, in dem sich die Anweisung bezüglich dieser Entität befindet, die nächste Anweisung untersucht werden. Diese Anweisung muss dahingehend geprüft wer-

den, ob sie die Funktionsdefinition derjenigen Funktion ist, die zuvor aufgerufen wurde.
Eine weitere Heuristik bezüglich Methoden fordert das Schreiben eines Funktionskopfs, inklusive des möglichen Datentyps, des Funktionsbezeichners, den Funktionsparametern und einer möglichen throw-Anweisung in einer Zeile (vgl. Martin [2009]: 121 f.). Hierzu müssen die entsprechenden Zeichen, die zur jeweiligen Funktionsdeklarationsanweisung gehören, daraufhin untersucht werden, in einer Zeile zu sein, wodurch die Heuristik syntaktisch analysierbar ist.
Eine letzte Heuristik bezüglich der Kategorie Allgemein/Global fordert die Platzierung von Import-Deklarationen zu Beginn des Programms, die hintereinander aufgeschrieben werden sollen (vgl. Martin [2009]: 362). Solche Import-Deklarationen machen dabei beispielsweise in Java den Compiler auf Typen in Paketen aufmerksam, die während der Import-Deklaration genannt werden, was beispielsweise Typen aus Bibliotheken oder Packages sind (vgl. Ullenboom [2023]: 246 ff.). Diese Heuristik ist ebenfalls syntaktisch analysierbar, indem die erste Entität, die in der Datei gefunden wird, eine Import-Deklaration sein muss, insofern eine Import-Deklaration existiert. Sobald eine Anweisung gefunden wird, die keine Import-Deklaration ist, darf keine weitere Import-Deklaration gefunden werden, um die Heuristik zu erfüllen.

4.2. Forschungsfragen und Hypothesen

In der vorliegenden Arbeit geht es um die Forschungsfrage, inwiefern statische Code-Analyse-Tools nützlich bei der Erstellung von Clean-Code sind. Die in Kapitel 3 beschriebenen Code-Smell Analyse-Tools sowie die Literatur zum Thema Software-Inspektion haben gezeigt, wie Code-Reviews und Analyse-Tools zur Erkennung von Defekten genutzt werden. Da das in der vorliegenden Arbeit zu implementierende statische Clean-Code-Analyse-Tool evaluiert werden soll, ergibt sich die Notwendigkeit, Hypothesen in einer ähnlichen Art und Weise aufzustellen. Dadurch wird die Forschungsfrage in Hypothesentests der quantitativen Datenanalyse beantwortbar. Die Erkennung von Programmcode, der nicht den Clean-Code-Konventionen entspricht, wird in der vorliegenden Arbeit als nützlich zur Erstellung von Clean-Code angesehen.
In Abschnitt 4.1 konnten die Clean-Code-Heuristiken den vier Kategorien Variablen, Methoden (bzw. Funktionen), Klassen, sowie Allgemein/Global zugeordnet

werden. Dadurch ergeben sich im Kontext des Themas der Arbeit vier relevante Hypothesen, die im Folgenden formuliert werden.

- H_1: Statische Code-Analyse-Tools können Variablen im Programmcode erkennen, die nicht den Clean-Code-Konventionen entsprechen.
- H_2: Statische Code-Analyse-Tools können Methoden bzw. Funktionen im Programmcode erkennen, die nicht den Clean-Code-Konventionen entsprechen.
- H_3: Statische Code-Analyse-Tools können Klassen im Programmcode erkennen, die nicht den Clean-Code-Konventionen entsprechen.
- H_4: Statische Code-Analyse-Tools können Programmcode erkennen, der nicht den Clean-Code-Konventionen entsprechen.

5. Methodik

Nachdem die Forschungsfrage sowie Hypothesen formuliert worden sind, werden in diesem Kapitel die Methoden der Datenerhebung beschrieben. Diese Daten wurden benötigt, um die in Kapitel 4 aufgestellten Hypothesen zu testen sowie die Forschungsfragen zu beantworten. Dazu wird in Abschnitt 5.1 die Implementierung des statischen Clean-Code Analyse-Tools beschrieben, welches Daten über die Sauberkeit von Code-Beispielen erhebt. In Abschnitt 5.2 wird dann das checklistenbasierte Code-Review beschrieben, welches zur Evaluation der Outputdaten des Clean-Code-Analyse-Tools verwendet wurde.

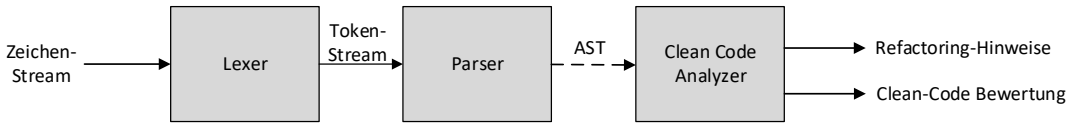

Abbildung 5.1.: Stellung des Parsers im Compilermodell, aus Aho et al. [2008] S.233

5.1. Implementierung des statischen Clean-Code-Analyse-Tools

In diesem Abschnitt wird die Realisierung des statischen Clean-Code-Analyse-Tool kurz beschrieben. Dazu werden zuerst in Unterabschnitt 5.1.1 die verwendeten Technologien genannt, wonach in Abschnitt 5.1 die eigentliche Implementierung beschrieben und schlussendlich in Unterabschnitt 5.1.3 die Daten, die aus dem Tool entstanden sind, Variablen für die spätere Analyse zugeordnet werden.

5.1.1. Abhängigkeiten

Vor der eigentlichen Implementierung des statisches Clean-Code-Analyse-Tools ist es wichtig festzulegen, welche Technologien bzw. Abhängigkeiten verwendet werden sollen. Da die Ausgaben des Tools für die Evaluation weiterverwendet werden, macht es Sinn, die Daten zentral abzulegen, um sie später unkompliziert durch etwaige Statistik-Tools abrufbar zu machen. Es wurde sich zur Nutzung dieser Technologie entschieden, da das in Abschnitt 5.2 verwendete Umfrage-Tool ein MyStructured Query Language (SQL)-Skript ausgeben kann. MySQL ist dabei ein relationales Datenbanksystem, welches die Datenbanksprache SQL unterstützt (vgl. Kofler [2005]: 38). Durch die Verwendung des Open-Source Administrationstools phpMyAdmin wird die Nutzung des Datenbankservers mit einer graphischen Benutzeroberfläche möglich, wobei es noch einige weitere Funktionen bietet (vgl. Buchmann and Smolarek [2005]: 20). Es kommt die Open-Source Datenbank Maria-DB zum Einsatz, die in der Serverversion 10.3.32 sowie der Clientversion 5.5.68 betrieben wird. MariaDB kann dabei als ein 1:1-Ersatz für MySQL betrachtet werden.
Da Java als Programmiersprache zur Entwicklung des Tools verwendet wird, wird ein Java Development Kit (JDK) benötigt. Zur Entwicklung wird die Java-Standard-Edition der Firma Oracle in der Version 11.0.16.1 verwendet.
Eine weitere verwendete Technologie ist die Entwicklungsumgebung Eclipse, die in der Version 4.27.0 für Java-Entwicklende zum Einsatz kommt. Die Entwicklungsumgebung besteht unter anderem aus einer Plattform, die die allgemeine programmiersprachenunabhängige Infrastruktur umfasst, sowie den JDT, die eine eigene Java-Entwicklungsumgebung in Eclipse einfügen (vgl. Gamma et al. [2004]: 26). Es ist dabei eine Open-Source Entwicklungsumgebung der Eclipse Foundation (vgl. Künneth and Ullenboom [2009]: 13), die abhängig von einer Installation eines JDKs ist (vgl. Künneth and Ullenboom [2009]: 19).

5.1.2. Umsetzung des statischen Clean-Code-Analyse-Tools

In diesem Abschnitt soll das implementierte statische Clean-Code-Analyse-Tool kurz skizziert werden. Die komplette Umsetzung des Tools zu dokumentieren würde den Umfang der vorliegenden Arbeit deutlich überschreiten. Stattdessen wird in diesem Abschnitt kurz auf die verwendeten Techniken sowie die Struktur des Tools eingegangen. In der vorliegenden Arbeit wurden ein Lexer sowie ein Parser, der

die Clean-Code-Analyse beinhaltet für ein Subset der objektorientierten Programmiersprache C++ realisiert. Warum sich für die Sprache C++ entschieden wurde, wird in Unterabschnitt 5.2.2 erläutert. Bei der Implementierung wurde versucht zwei Formen von Komplexitätsreduktionen durchzuführen. Zum einen sollte der implementierte Parser lediglich ein Subset der Programmiersprache C++ parsen können, da das der Beginn vieler Compiler wie beispielsweise von lcc war (vgl. Fraser and Hanson [1995]: 11). Zum anderen sollte zuerst damit begonnen werden einen C-Parser zu realisieren, da die Grammatik von C++ auf der Grammatik von C aufbaut und hier große Überschneidungen bestehen (vgl. Stroustrup [2015]: 9).
Die Idee der Struktur des Clean-Code-Analyse-Tools wird in Abbildung 5.2 gezeigt. Aus Abschnitt 2.3 ist der Aufbau eines Compilers bereits bekannt, wobei in der Designidee nach dem Parsing-Vorgang ein AST zur weiteren Analyse an den Teil des Tools weitergegegeben werden sollte, der sich um die Clean-Code-Analyse kümmert. Dabei ist die gezeigte Struktur nicht exakt äquivalent zum später implementierten Tool, da die Clean-Code-Analyse zeitlich während des Parsing-Vorgangs stattfindet, wobei die syntaktische Analyse wie in der Graphik zeitlich nach der lexikalischen Analyse stattfindet. Des Weiteren wurde sich zur Realisation eines handgeschriebenen rekursiven Abstiegsparsers entschieden. Rekursive Abstiegsparser besitzen gegenüber generierten LL(k)-Parsern einige Vorteile, da sie nicht zwingend auf Parsing-Tabellen zurückgreifen müssen und so eine höhere Geschwindigkeit besitzen (vgl. Wilhelm et al. [2012]: 103). Dieser Punkt ist besonders wichtig, da die für einen LL(k)-Parser benötigten First- und Follow-Mengen bezüglich der C++ Standards nicht frei verfügbar sind. Darüber hinaus sind sie simpel genug, unmittelbar in der Programmiersprache des Compilers verfasst werden zu können (vgl. Wilhelm et al. [2012]: 103). Außerdem korrespondiert ein Parser zum Automatentyp des Kellerautomaten. Im Stack des Kellerautomatens befinden sich dabei die noch nicht abgearbeiteten Produktionsregeln, für die sich der Parser entscheiden kann (vgl. Gumm et al. [2011]: 739). Dabei beeinflussen sowohl der Stack, als auch der Tokenstream beziehungsweise das Lookahead von Tokens den Zustand des Automatens (vgl. Gumm et al. [2011]: 739). Ein rekursiver Abstiegsparser handhabt den Stack dabei mithilfe der Rekursion automatisch, sodass hier eine weitere Komplexitätsreduktion besteht. Des Weiteren wäre die Verwendung eines generierten Parsers ebenfalls eine komplexe Aufgabe gewesen, da die EBNFs aktuellerer C++-Standards nicht frei verfügbar sind und eigenstän-

dig hätten erstellt werden müssen. Ein handgeschriebener Parser besitzt außerdem gegenüber einem generierten Parser den Vorteil, keinen unleserlichen fremden Code des Parser-Generators verstehen zu müssen (vgl. Wilhelm et al. [2012]: 103).

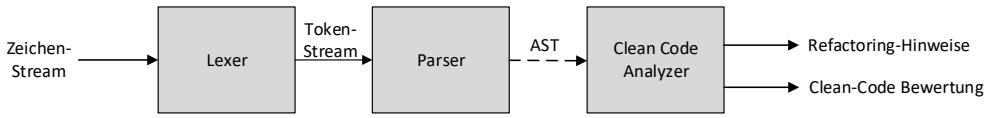

Abbildung 5.2.: Statisches Clean-Code-Analyse-Tool Aufbau

Der Lexer ist der einzige Teil des Compilers, der sich jedes Zeichen des Quellcodes ansieht (vgl. Fraser and Hanson [1995]: 102). Dabei besitzt der implementierte Lexer eine flexible Anzahl an Lookahead-Zeichen. Dadurch kann die Entscheidung für ein bestimmtes Token anhand einer flexiblen Anzahl von Lookahead-Zeichen (ausgehend vom aktuellen Zeichen) vorhergesagt werden. Dabei wird wie beispielsweise beim C-Compiler lcc von Fraser and Hanson [1995] ein Input-Buffer zur Zwischenspeicherung verwendet (vgl. Fraser and Hanson [1995]: 104). Ein Buffer (deutsch: Puffer) ist ein Container, der eine Folge von Werten eines primitiven Datentyps beinhaltet und von beziehungsweise auf den mithilfe von get- und put-Methoden gelesen beziehungsweise geschrieben werden kann (vgl. Ratz et al. [2018]: 605). Durch den Buffer werden dabei im implementierten Tool bis zu 2000 Zeichen gleichzeitig vorgehalten, um zusammengehörige Zeichen von Nummern oder Strings zu erkennen. Besitzt der Buffer keine neuen Zeichen mehr, wird er erneut aufgefüllt. Der endliche deterministische Automat Abbildung 5.3 zeigt die lexikalische Verarbeitung eines Bezeichners, wobei der (mögliche) Endzustand des gefundenen Bezeichners mit einer doppelten Umrandung markiert ist.
Anschließend wird überprüft, ob es sich bei dem gefundenen String um ein Schlüsselwort handelt. Alle Schlüsselwörter einer Sprache können am einfachsten als Liste aufgeführt werden (vgl. Hedtstück [2012]: 8). Die C++-Schlüsselwörter sind in Stroustrup [2015] aufgelistet (vgl. Stroustrup [2015]: 156) und wurden daher als solche in einer Boolean-Funktion implementiert, die mithilfe von String-Vergleichen entscheidet, ob ein String ein Schlüsselwort darstellt. Neben Bezeichnern und Schlüsselwörtern besitzt C noch Tokens vom Typ Konstante, String-Literal, Operator und Begrenzer (vgl. Fraser and Hanson [1995]: 108). Neben diesen Typen von Token wurden auch Kommentare als Tokentyp sowie ein Newline-Tokentyp umgesetzt, da sie für die spätere Analyse benötigt werden. Leerzeichen werden

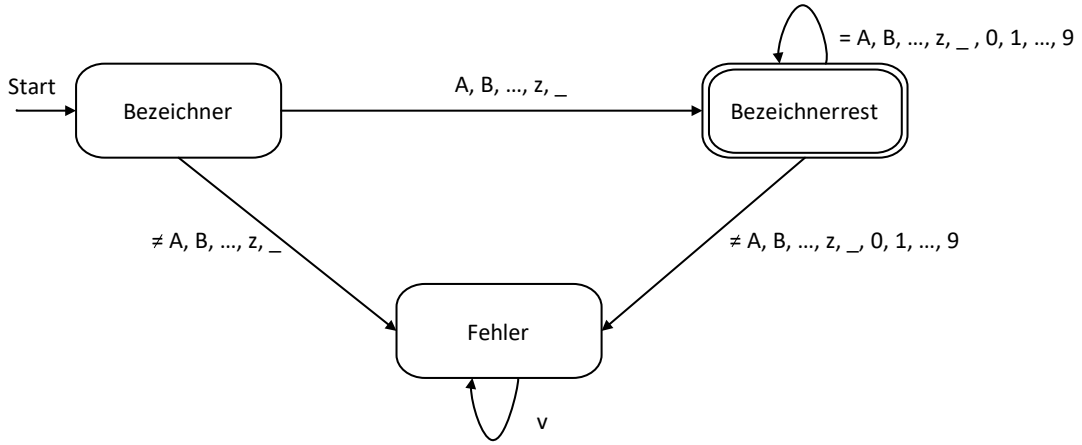

Abbildung 5.3.: Endlicher deterministischer Automat aus Hedtstück [2012], S. 58

nicht als Token-Typ gespeichert, sondern jedes Token-Objekt besitzt eine Integer-Variable, die die Anzahl der Leerzeichen rechts des Tokens enthält. Dabei gibt es im implementierten Tool die Besonderheit von öffnenden Klammern, die als Operator und schließenden Klammern, die als Begrenzer umgesetzt wurden. Darüber hinaus besitzt der Lexer bereits eine Fehlererkennung und kann wie in Abbildung 5.3 dargestellt beispielsweise in der C++-Grammatik ungültige Bezeichner oder Konstanten erkennen. Bezüglich Konstanten werden dabei Dezimal-, Okta- und Hexadezimalzahlen erkannt.
Der Lexer speichert jedes gefundene Token in einer Vector-Datenstruktur, die am Ende der lexikalischen Analyse an den Parser weitergegeben wird. Der Parser analysiert die Tokens, wobei er anhand der Syntax von C beispielsweise auf Ausdrücke stößt. Ausdrücke sind in C eine eigene Subsprache, die in einem AST als IR abgebildet werden. Dabei spielt die Assoziativität und die Operatorenpräzedenz eine große Rolle, da sie entscheiden welcher Parse-Baum der richtige ist und wie der Ausdruck am Ende aussieht. Die Operatorenpräzedenz sowie die Assoziativität der Operatoren von C werden in Kernighan and Ritchie [1988] beschrieben (vgl. Kernighan and Ritchie [1988]: 53). Beispiele für Ausdrücke sind Zuweisungen und bedingte Ausdrücke (vgl. Kernighan and Ritchie [1988]: 237 f.) Diese ASTs besitzen oftmals Operatoren, die in der analysierten Programmiersprache nicht vorkommen (vgl. Fraser and Hanson [1995]: 147 f.). Beispielsweise wird der Operator "()„ als Wurzelknoten eines Ausdrucks verwendet, der einen Funktionsaufruf beschreibt. Im linken Kind des Wurzelknotens befindet sich ein Knoten für den Funktions-

bezeichners, während auf der rechten Seite alle Übergabeparameter in den Subbäumen abgebildet werden. Nach der Erstellung des ASTs eines Funktionsaufrufs wird beispielsweise mit einer Tiefensuche nach der Clean-Code-Heuristik bezüglich der Übergabe von Nullpointern (siehe Tabelle A.10) gesucht. Außerdem gibt es in C Anweisungen, worunter Iterationen und Bedinungen fallen, wobei sie Listen von Deklarationen und Anweisungen in geschweiften Klammern verschachtelt enthalten (vgl. Kernighan and Ritchie [1988]: 236 f.). Deklarationen sind dabei beispielsweise Funktionsdeklarationen (vgl. Kernighan and Ritchie [1988]: 234), wobei Deklarationen im implementierten Tool genauso wie Anweisungen umgesetzt sind. Dabei gibt es einen Körperknoten, der Anweisungen und Ausdrücke aufnehmen kann. Für jede Anweisung gibt es dann eine eigene Knoten-Datenstruktur, die die Hierarchie der Daten beibehält. Das heißt der ein Strukturknoten ist beispielsweise der Elternknoten eines Körperknotens, der Elternknoten von im Körper deklarierten Variablenknoten ist. C++ bietet gegenüber C die Objektorientierung, sodass ab dem Zeitpunkt des Wechsels der Implementierung hin zu einem Parser eines C++-Subsets der Schwierigkeitsgrad deutlich zunahm. Während für den C-Parser noch ein Lookahead von einem Token ausgereicht hat, benötigte C++ ein flexibles Lookahead. Insbesondere Generics, die Templates von Klassen zulassen, deren Datentypen verzögert festegelegt werden (vgl. Stroustrup [2015]: 699), waren dabei eine große Herausforderung. Außerdem werden in C Deklarationen beziehungsweise Definitionen mit Schlüsselwörtern wie “struct„ eingeleitet, wobei in C++ jeder Bezeichner auch ein Datentyp einer Klasse sein kann, wobei große Symboltabellen verwaltet werden müssen. Des Weiteren ist hier auch das Methodenüberladen zu nennen, was ebenfalls die Errichtung von Symboltabellen mit der Speicherung der Bezeichner, der Reihenfolge der ÜBergabeparameter sowie der Gültigkeitsbereiche erfordert. Insgesamt wurden Funktionen für fast alle C sowie in der Folge für sehr viele C++-Konstrukte geschrieben, die die jeweiligen Produktionsregeln in Knoten eines ASTs parsen.

Tabelle 5.1 zeigt die ad-hoc-Metrik, wobei einige Clean-Code-Heuristiken aus Abschnitt 4.1 umgesetzt werden konnten. Der im Tool integrierte Clean-Code-Analyzer verwendet die ad-hoc-Metrik, um gefundene Defekte oder die Abwesenheit von Defekten bei gefundenen Instanzen zu bewerten. Dabei sind alle umgesetzten Heuristiken in der dritten Spalte zu sehen, wobei die erreichbare Punktzahl jeder Heuristik je gefundener Instanz gilt, auf die sich die jeweilige Heuristik bezieht. Darüber hinaus gibt das Tool Refactoring-Hinweise, wobei Listing A.6 eine Beispielausgabe

von Hinweisen für den Listing A.5 zeigt. Die Ergebnisse der Kategorien Variablen, Methoden, Klassen und Allgemein/Global werden in eine in eine CSV-Datei ausgegeben.

Kat.	**Erreichbare Punktzahl**	**CC-Heuristik-Nr.**	**Bemerkungen**
Variablen	1	13, 14, 15, 17, 24, 38	17: Je for-Anweisung, ohne forEach-Schleife; Nr. 38: Je Nutzung des Zuweisungsoperators
Methoden	-1	66	66: Je goto-Anweisung
Methoden	1	8, 13, 14, 15, 19, 24, 26, 29, 32, 33, 36, 37, 41, 44, 45, 49, 50, 64, 68, 71, 74	8: Je überladenen Konstruktor; 32: Je switch-Anweisung; 33: Je try-/catch-Anweisung; 49: Je Methode, die eine Referenz auf ein Objekt zurückgeben kann; 50, 71: Werden nur bei Funktionen außerhalb von Klassen angewandt und zählen sonst in die Klassenmetrik
Methoden	2	28, 31	31: Je Anweisung
Methoden	3	30	30: Konsistenz der Einrückungen wurde hinzugefügt
Methoden	4	43	
Klassen	1	9 + 10, 13, 14, 15, 18, 24, 35, 40, 50, 78	Nr. 9 und 10 geben nur gemeinsam 1 Punkt wenn beide eingehalten werden; Nr. 18: Es wird für die Nutzung von Pascal-Case statt nur Beginn mit Großbuchstaben geprüft
Klassen	2	67, 72	
Klassen	4	39	39: Es wird wegen Nr. 30 zusätzlich auf nicht mehr als 3 - 4 Einrückungen geprüft
Klassen	6	73	73: 2 Punkte für Vorwärts-Deklarationen, 2 Punkte für Methodendeklarationen, 2 Punkte für verschachtelte Strukturen
Allgemein/Global	2	34	

Kat.	Erreichbare Punktzahl	CC-Heuristik-Nr.	Bemerkungen
Allgemein/Global	4	72 + 73, 79	72 + 73 werden für den globalen Raum implementiert und durch Klassen erweitert, um dem globalen Raum mehr Struktur zu geben; 79: 2 Punkte für Bibliotheksdirektiven, 2 Punkte für Namensraumdefinitionen

Tabelle 5.1.: Ad-hoc-Metrik: Umgesetzte CC-Heuristiken im statischen Clean-Code Analyse-Tool und ihre Bewertungen

Die erreichbare Punktzahl gilt je gefundene Instanz und je CC-Heuristik in der Liste.
Die Nummern in der Tabelle beschreiben CC-Heuristiken und sind in Tabelle 4.1, Tabelle A.7, Tabelle A.8, Tabelle A.9, Tabelle A.10, Tabelle A.11, Tabelle A.12 und Tabelle A.13 zu finden.

5.1.3. Abhängige Variablen

Die abhängige Variable (AV) ist diejenige Variable, die in der Statistik anhand einer oder mehrere Prädiktoren (bzw. einer unabhängige Variable (UV)) vorhergesagt werden soll, wobei nicht zwingend ein Kausalzusammenhang bestehen muss (vgl. Bortz and Weber [2005]: 182).
Um die vier Hypothesen aus Abschnitt 4.2 testen zu können, werden die vier AVs VariablenAnalyzerScore, MethodenAnalyzerScore, KlassenAnalyzerScore und GesamtAnalyzerScore verwendet. Die AVs ergeben sich dabei aus den vier in Abschnitt 4.1 herausgearbeiteten Kategorien Variablen, Methoden, Klassen und Allgemein/Global. Auf Grundlage der der ad-hoc-Metrik bewertet das statische Clean-Code-Analyse-Tool die Sauberkeit von C++-Programmcodes, die die Inputdaten darstellen.
Es werden fünf verschiedene Code-Beispiele aus Martin [2009] getestet, wobei die Auswahl der Beispiele in Bezug zur Lesbarkeitsstudie in Abschnitt 5.2 genauer erläutert werden. Die AVs sind dabei metrische Variablen, die Werte zwischen 0,00 und 10,00 zur Bewertung der Sauberkeit von Programmcode annahmen können. Das Tool berechnet die AVs anhand der in Tabelle 5.1 beschrieben ad-hoc-Metrik, wobei jedes potenzielle Vorkommen einer Clean-Code-Heuristik aus Abschnitt 4.1, die syntaktisch analysierbar ist, je nach Komplexität der Heuristik Punkte für die jeweilige Kategorie erzielt. Das heißt, die erreichbare Punktzahl hängt vom analy-

sierten Programmcode und der Anzahl von verwendeten Instanzen der jeweiligen Kategorie ab. Der Anteil der erzielten Punkte für jede Kategorie wird am Ende berechnet und auf einen Wert zwischen von 0,00 bis 10,00 normalisiert. Da die Clean-Code-Heuristiken sich in Bezug zur Kategorie Allgemein/Global lediglich auf die Dateigröße sowie den globalen Scope beziehen, wird zuerst wie bei den anderen drei Kategorien eine Variable GlobalAnalyzerScore berechnet, die sich auf den Anteil der erreichten Punkte in der Kategorie Allgemein/Global bezieht. Sie ist ebenfalls metrisch skaliert und auf die Werte zwischen 0,00 und 10,00 normalisiert. Da die Heuristiken der Kategorie Allgemein/Global jedoch nicht den gesamten Programmcode betreffen, wird eine AV GesamtAnalyzerScore zu gleichen Anteilen aus den Werten der Variablen VariablenAnalyzerScore, MethodenAnalyzerScore, KlassenAnalyzerScore und GlobalAnalyzerScore berechnet, sodass der gesamte Programmcode einbezogen ist.
Gegenüber den Programmcodes, die bei der Lesbarkeitsstudie verwendet werden, gibt es kleinere Unterschiede. In den Java-Programmcodes von Martin [2009] fehlen die Deklarationen bezüglich importierter Bibliotheken und Packages im Dateikopf. Da diese Java-Programmcodes für die Lesbarkeitsstudie in die Programmiersprache C++ übersetzt wurden, die zeitlich vor der Realisierung des Tools durchgeführt wurde, wurden bei der Übersetzung äquivalent zu den fehlenden Bibliotheks- und Package-Importen sowohl die Definitionen von Namensräumen als auch die Einbindung von Bibliotheksdirektiven im Dateikopf weggelassen. Darüber hinaus werden in den C++-Programmcodes, die zur Erhebung der AVs an das Clean-Code-Tool übergeben werden, Klassen deklariert. Diese Klassen werden bei den Java-Code-Beispielen von Martin [2009] innerhalb des Programmcodes zwar verwendet, aber in den Programmcodes weder deklariert, noch eingebunden. Das statische Clean-Code-Analyse-Tool benötigt diese Klassen jedoch für die Analyse, sodass die Klassen bei den Übersetzungen in C++ nach den Bibliotheksdirektiven und den Namensraumdefinitionen im Dateikopf eingefügt wurden, wie beispielsweise im Dateikopf in Listing A.1 zu sehen ist. Die weiteren Input-Daten, die die Stichprobe der Erhebung durch das Tool bilden, werden in Listing A.2, Listing A.3, Listing A.4 und Listing A.5 dargestellt. Aus den genannten Gründen existiert ein kleiner Unterschied bei den Inputdaten des Tools und den Programmcodes, die bei der Lesbarkeitsstudie in Abschnitt 5.2 verwendet wurden.

5.2. Checklistenbasiertes Code-Review

Im Anschluss der Implementierung des Clean-Code-Tools, dessen Bewertungsdaten die AVs darstellen, wird der Output des Tools mithilfe von Daten aus einem checklistenbasierten Code-Review evaluiert. Dabei wird in Unterabschnitt 5.2.1 die Auswahl der Methodik der Studie zuerst begründet. In Unterabschnitt 5.2.2 wird daraufhin die Konzeption des Fragebogens beschrieben, während in Unterabschnitt 5.2.3 schlussendlich der Zusammenhang zwischen den in der Studie erhobenen Daten und den UV beschrieben wird.

5.2.1. Begründung der Methodenauswahl

Wie in Unterabschnitt 2.1.3 beschrieben, ist die Methode des checklistenbasierten Code-Reviews kostengünstig umzusetzen. Die vorliegende Arbeit wurde begleitend zu einer Lehrtätigkeit als Tutor an einer universitären Fakultät für Informatik umgesetzt, sodass Studierende mit moderaten Programmiererfahrung in ausreichendem Maße zur Verfügung standen. Aus diesem Grund wurde die Studie in einer quantitativen Form umgesetzt. Obwohl für Code-Reviews normalerweise erfahrene Reviewende benötigt werden, lassen sich mit der Unterstützung von Checklisten durch Studierende als Reviewende trotzdem viele Defekte während eines Code-Reviews finden, wie empirisch gezeigt wurde (vgl. McMeekin et al. [2009]: 236 f.). Bei der Konzeption der Checklisten soll dabei jedes Item aus einer Frage bestehen, deren negative Beantwortung einen Defekt offenbart (vgl. Gilb et al. [1993]: 56). Darüber hinaus sollen diese Fragen jeweils auf eine Regel zurückzuführen sein, die sich auf Best-Practices des Programmierens bezieht (vgl. Gilb et al. [1993]: 44). Daraus folgt ein weiterer Grund für die Nutzung des checklistenbasierten Code-Reviews. Da die Clean-Code-Heuristiken bereits als solche Best-Practice-Regeln aufgefasst werden können, ist es sehr einfach auf Basis dieser Heuristiken Fragen für die Checklisten des Code-Reviews zu generieren.
Das Clean-Code-Konzept besteht aus den drei Dimensionen Lesbarkeit, Testbarkeit und Wartbarkeit. Weil mit Studierenden der Informatik und den Ingenieurswissenschaften weniger erfahrene Reviewende eingesetzt werden konnten, wurde die Studie im Vorfeld lediglich auf die Dimension der Lesbarkeit reduziert. In der Studie wird von einer ausreichenden Korrelation der drei Dimensionen ausgegangen, wobei die Dimension der Lesbarkeit als die am einfachsten verständliche Dimension für Studierende eingeschätzt wurde. Darüber hinaus sind im checklistenbasierten

Code-Review lediglich 25 Items je Code-Dokument erlaubt (vgl. Gilb et al. [1993]: 56), sodass durch die Reduktion auf die Dimension der Lesbarkeit die Anzahl der Items je Code-Dokument reduziert werden konnte.

5.2.2. Fragebogen

Die Studie und der Fragebogen wurden mit SoSci Survey erstellt, einer Web-Applikation für wissenschaftliche Onlinebefragungen. SoSci-Survey bietet dabei den Vorteil die erhobenen Daten der Umfrage in ein SQL-Skript ausgeben zu können, welches für MySQL angepasst ist. So können die Daten der Online-Studie unkompliziert mit denen des Tools in der bestehenden MySQL-Datenbank verknüpft und für weitere Berechnungen zentral gespeichert werden.
Der Fragebogen beginnt mit den Kontrollvariablen, deren korrespondierende Items vor allem aus der Abfrage demografischer Daten bestehen. Abbildung A.1 zeigt den View der Seite des Fragebogens, die die Items zu den Kontrollvariablen enthält. Dabei wurde nach jedem Item die Möglichkeit gegeben, je nach Item entweder eine der vorgegebenen Antworten auszuwählen oder im Freifeld zu antworten.
Nach der Abfrage der Kontrollvariablen begann die eigentliche Studie. Dabei wurden fünf Code-Beispiele von den Teilnehmenden in den vier Kategorien Variablen, Methoden, Klassen und Allgemein/Global anhand von spezifischen Fragen im Sinne des checklistenbasierten Code-Reviews evaluiert. Diese Kategorien sind deckungsgleich mit den Kategorien der Clean-Code-Heuristiken.
Die Programmcode-Beispiele, die den Studienteilnehmenden gezeigt wurden, sind dabei in der Programmiersprache C++ verfasst. Da die vorliegende Arbeit im Rahmen einer Lehrtätigkeit als Tutor an einer universitären Fakultät für Informatik verfasst wird, ist die Wahl der Programmiersprache vom Wissen der Studienteilnehmenden abhängig. Die Befragten an der universitären Einrichtung hatten dabei entweder Kenntnisse in C oder in C++, sodass die Wahl aufgrund der Eigenschaften der OOP von C++ auf diese Programmiersprache fiel.
Die Kategorien wurden nacheinander abgehandelt, sodass jedes Code-Beispiel viermal innerhalb der Befragung evaluiert wurde. Der Aufbau für jede Kategorie war dabei jedes Mal gleich. Zunächst wurde eine Checkliste mit Fragen zu Defekten bezüglich der jeweiligen Kategorie präsentiert.
In Abbildung 5.4 wird beispielsweise eine Checkliste zur Kategorie der Klassen vorgestellt (siehe Abbildung A.3, Abbildung A.4 und Abbildung A.5 für alle weiteren Kategorien), wobei zu jeder der vier Kategorien eine Checkliste während der

Studie gezeigt wird. Die Konzeption der Checkliste orientiert sich an der Checkliste aus der Dokumentation der Google Engineering Practices, die diese Form der Checklisten während ihrer Code-Reviews verwenden (vgl. Chong et al. [2021]: 21). Dabei ist sowohl das Design, als auch die Struktur der Items an das dortige Konzept angelehnt. Diese Struktur beinhaltet im unteren grauen Bereich eine Liste von Items, die jeweils aus einem Spiegelstrich, gefolgt von einer Unterkategorie bestehen, auf die sich das jeweilige Item bezieht. Schlussendlich wird die Frage genannt, die sich auf einen Defekt im Programmcode in der genannten Unterkategorie bezieht. Durch diese Unterkategorien lässt sich die Oberkategorie, die im oberen schwarzen Bereich der Checkliste genannt wird und deckungsgleich mit einer der vier Clean-Code-Kategorien ist, weiter präzisieren. Die Checkliste soll dabei die Aufmerksamkeit der Studierenden auf die relevanten Defekte im nachfolgenden Programmcode lenken. Jede der Checklistenfragen lässt sich dabei unmittelbar auf eine der Clean-Code-Heuristiken aus Abschnitt 4.1 zurückführen.

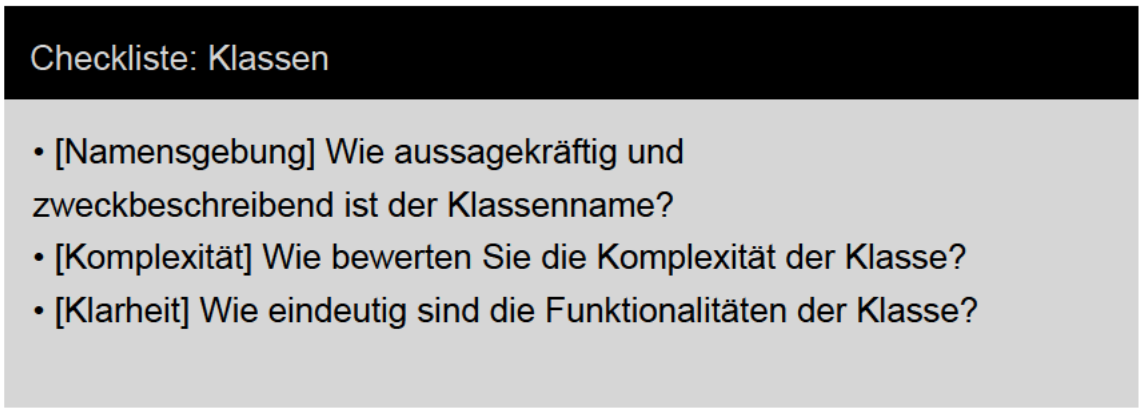

Abbildung 5.4.: Lesbarkeitsstudie Checkliste Klassen

Im Anschluss an die Checkliste wurde das Code-Beispiel mit den rot markierten Area of Interest (AOI)s gezeigt, die abhängig von der Kategorie der unmittelbar zuvor gezeigten Checkliste. Die Items, aus deren Antworten die UVs berechnet wurden, werden nach dem Programmcode gezeigt. Diese Items sind Aussagen, die auf Grundlage der unmittelbar vor dem Programmcode gezeigten Checklistenfragen formuliert worden sind.
Abbildung 5.5 zeigt das erste Beispiel, welches den Studienteilnehmenden gezeigt wurde. Dabei wurde ihnen ein Screenshot des Editors Visual Studio Code (VSC) im dunklen Theme präsentiert, in dem Syntax-Highlighting zu sehen war. Darüber hinaus wurden zum besseren Verständnis für die Studierenden die AOI (deutsch:

Interessensbereiche) mit roten Kästen markiert. Aufgrund der besseren Les- und Druckbarkeit ist dieser Screenshot in Abbildung 5.5 durch eine andere Form des identischen Programmcodes ersetzt worden, wobei die Markierungen der AOIs fehlen. Die in der Studie genutzten Beispiele sind ursprünglich Java-Programmcodes aus Martin [2009], die im Zuge der vorliegenden Arbeit in C++ übersetzt wurden. Die weiteren Code-Beispiele sind in Abbildung A.6 und Abbildung A.7 (zweites Code-Beispiel), Abbildung A.8 und Abbildung A.9 (drittes Code-Beispiel), Abbildung A.10 und Abbildung A.11 (viertes Code-Beispiel) sowie schlussendlich Abbildung A.12 (fünftes Code-Beispiel) dargestellt.

```
1  class UserValidator {
2    private:
3      Cryptographer cryptographer;
4
5    public:
6      bool checkPassword(string userName, string password);
7  };
8
9  bool UserValidator::checkPassword(string userName, string password) {
10   User *user = UserGateway::findByName(userName);
11   if (user != NULL) {
12     string codedPhrase = user -> getPhraseEncodedByPassword();
13     string phrase = cryptographer.decrypt(codedPhrase, password);
14     if (phrase.compare("Valid Password") == 0) {
15       Session::initialize();
16       return true;
17     }
18   }
19   return false;
20 }
```

Abbildung 5.5.: Lesbarkeitsstudie C++ Beispiel 1

Das erste Beispiel wird von Martin [2009] als relativ sauber beschrieben, wobei der Aufruf von Session.initialize() nicht zum Namen der checkPassword-Funktion passt, deren Bezeichner keine Initialisierung einer Sitzung ankündigt (vgl. Martin [2009]: 75 f.). Demgegenüber wird die Funktion im zweiten Beispiel als chaotisch in Form einer tief verschachtelten Struktur sowie seltsamen Variablen beschrieben (vgl. Martin [2009]: 180 f.). Das dritte Beispiel wird als Musterbeispiel der Formatierungsregeln von Martin [2009] genannt Martin [2009]: 125 ff.). Beim vierten Code-Beispiel wird von Martin [2009] kritisiert, wie die Funktion auf zu vielen unterschiedlichen Abstraktionsebenen anzusiedeln ist, wobei er im Nachgang eine deutlich kürzere Funktion mit identischer Funktionalität nach dem Refacto-

ringvorgang präsentiert (vgl. Martin [2009]: 62 f.). Beispiel fünf wird zuerst mit Leerzeichen zwischen den Entitäten im Programmcode gezeigt, wobei beschrieben wird, wie sich die Lesbarkeit durch das Weglassen der Leerzeichen deutlich verschlechtert (vgl. Martin [2009]: 112 f.). Aus diesem Grund wurde das Beispiel ohne Leerzeichen als Teil der Studie aufgenommen. Aufgrund der unterschiedlichen Beschreibungen der Qualität der Sauberkeit der Code-Beispiele von Martin [2009] wird von einer ausreichenden Unterschiedlichkeit der präsentierten Beispiele ausgegangen.
Die Items werden im Anschluss an die Code-Beispiele gezeigt, wobei sie sich genauso wie die Checklistenfragen unmittelbar aus einer der Clean-Code-Heuristiken aus Abschnitt 4.1 ableiten lassen, wie beispielhaft in Tabelle 5.2 dargestellt. Eine Ausnahme ist beim Herleiten von einer Regel das Item der Kategorie Allgemein/Global, da das Clean-Code-Konzept lediglich Heuristiken bezüglich von Variablen, Methoden, Klassen und des globalen Scopes besitzt, nicht jedoch explizit bezüglich des gesamten Programmcodes. Eine Regel zur allgemeinen Bewertung der Lesbarkeit von Programmcode müsste sich wie bei der Berechnung des Gesamtscores des Tools aus vielen kleinen Regeln zusammensetzen, wobei bei einer Studie auch eine allgemeinere Nachfrage möglich ist. Dadurch können kompliziertere Rechnungen vermieden und die Anzahl der Items reduziert werden. Eine ausführliche Übersicht der Herleitungen aller Items des Fragebogens findet sich in Tabelle A.14. Ein View der Items im Fragebogen wird in Abbildung A.2 dargestellt. Die Reviewenden müssen die Items in Bezug auf Defekte in der Lesbarkeit des Programmcodes auf einer fünfstufigen Likert-Skala (von 1 = „stimme überhaupt nicht zu“ bis 5 = „stimme voll und ganz zu“) beantworten. Diese Likert-Skala wird bei allen Items bezüglich Einschätzungen zu Defekten im Programmcode während der Lesbarkeitsstudie verwendet. Durch diese ausdifferenzierten Antwortmöglichkeiten lässt sich die Anzahl von Defekten im Programmcode genauer erheben, wobei wie im checklistenbasierten Code-Review gefordert, eine negative Antwort (mit einer geringeren Bewertung) auf einen Defekt im Programmcode schließen lässt.

Kat.	**CC-Heuristik-Nr.**	**Checklistenfrage**	**Item**
V	11	Wie aussagekräftig und zweckbeschreibend sind die Variablennamen?	Die Variablennamen sind aussagekräftig und zweckbeschreibend.

Kat.	CC-Heuristik-Nr.	Checklistenfrage	Item
M	31	Wie bewerten Sie die Komplexität der Blöcke (if/else, ..)?	Die Komplexität der Blöcke (if/else, ..) ist passend.
K	27	Wie eindeutig sind die Funktionalitäten der Klasse?	Die Funktionalitäten der Klassen sind eindeutig.
A/G	-	Lesbarkeit beschreibt den mentalen Aufwand, der benötigt wird, um den Code zu verstehen. Wie bewerten Sie die Lesbarkeit des Codeabschnitts?	Lesbarkeit kann so definiert werden, dass sie den mentalen Aufwand beschreibt, der benötigt wird, um den Code zu verstehen. Die Lesbarkeit des vorliegenden Codes ist angemessen.

Tabelle 5.2.: Beispiele Herleitung Checklisten und Items der Lesbarkeitsstudie

Die CC-Heuristiken und ihre Nummern sind in Tabelle 4.1, Tabelle A.7, Tabelle A.8, Tabelle A.9, Tabelle A.10, Tabelle A.11, Tabelle A.12 und Tabelle A.13 zu finden.

Insgesamt wurden 16 Items je Programmcode abgefragt, sodass die Studie deutlich unter der von Gilb et al. [1993] beschriebenen Obergrenze von 25 Items je Code-Dokument beim checklistenbasierten Code-Review bleibt. Die Studie endete mit einer Danksagung und einiger Witze bezüglich Themen der Informatik.

5.2.3. Unabhängige Variablen

Wie in Unterabschnitt 5.1.3 beschrieben, versuchen die UVs bzw. Prädiktoren die AV vorherzusagen, wobei die Auswirkungen der UVs auf die AVs untersucht werden (vgl. Bortz and Weber [2005]: 7). Um die Hypothesen aus Abschnitt 4.2 zu testen und die Forschungsfrage zu beantworten werden verschiedene UVs benötigt. Diese ergeben sich aus dem Design der Studie und der Einteilung der Lesbarkeit des Programmcodes in die vier Kategorien Variablen, Methoden, Klassen und Allgemein/Global, woraus die vier UVs VariablenReviewScore, MethodenReviewScore, KlassenReviewScore und AllgemeinerReviewScore entstehen. Bei den vier UVs handelt es sich um metrische Variablen, die zwischen 0,00 und 10,00 skaliert sind, wobei höhere Werte mehr Lesbarkeit ausdrücken. Zur Berechnung dieder vier UVs wurden die zu der Kategorie zugehörigen Antworten, die in Integerwerten zwischen 1 und 5 abgespeichert sind, durch SQL mit einem Cast in Dezimalwerte konver-

tiert. Anschließend wurden die Dezimalwerte aufsummiert, durch die Anzahl der Items der jeweiligen Kategorie geteilt und schlussendlich verdoppelt, sodass der normalisierte Wert zwischen 0,00 und 10,00 berechnet wird. Beim vierten Code-Beispiel sind die fehlenden Klassen zu beachten, sodass bei diesem Beispiel keine Items zur Lesbarkeit von Klassen abgefragt wurden.
Weitere UVs sind demgegenüber lediglich Kontrollvariablen. Kontrollvariablen sind diejenigen Variablen, die als weitere UVs mit in die Analyse einbezogen werden, wobei es bei ihnen darum geht, die Werte dieser Variablen bei der Analyse konstant zu halten (vgl. Bortz and Weber [2005]: 7). Die erhobenen Kontrollvariablen sind in Tabelle A.15 dargestellt.

5.2.4. Ablauf der Studie

Die Studie wurde im Sommersemester 2022 realisiert und war über einen Link auf die Webseite von SoSci Survey erreichbar. Zu Beginn der Onlinestudie fand zwischen dem 07. Juli 2022 und dem 10. Juli 2022 ein Pretest mit vier Teilnehmenden statt. Dabei ging es vor allem darum herauszufinden, ob die benötigte Bearbeitungszeit des Fragebogens die zuvor angedachten 15 Minuten deutlich überschreiten würde. Darüber hinaus wurden Rechtschreibfehler in der Datenschutzerklärung und in den Items des Fragebogens beseitigt.
Am 10. Juli 2022 begann dann die eigentliche Studie, die bis bis zum 17. August 2022 durchgeführt wurde. Die meisten Studienteilnehmenden konnten dabei während der Präsenzveranstaltung des Moduls „Betriebssysteme“ gewonnen werden. Darüber hinaus wurde die Studie in einem Online-Tutorium des Moduls „Algorithmen und Datenstrukturen“ durchgeführt. Diese Module werden hauptsächlich von Studierenden der Informatik belegt. Außerdem wurde die Studie im Modul „Grundlagen der Informatik II“ bzw. „Informatik II“ durchgeführt, die von Studierenden der Ingenieurs- und weiteren Wissenschaften besucht wird. Des Weiteren wurde die Studie über den Email-Verteiler des Fachschaftsrats Informatik der universitären Einrichtung geteilt. Insgesamt konnten so 60 Teilnehmende für die Studie gewonnen werden. Da jede teilnehmende Person die fünf genannten Code-Beispiele reviewed hat, konnten so 300 Beobachtungen bzw. Code-Reviews erreicht werden.

6. Ergebnisse

Nachdem die Methodik der Erhebung umfassend beschrieben wurden, werden in diesem Kapitel die statistischen Verfahren angewendet, um die aufgestellten Hypothesen zu testen und die Forschungsfrage zu beantworten. Zur statistischen Analyse der Daten wird in diesem Kapitel die Entwicklungsumgebung RStudio in der Version 2023.06.2+561 mit der Statistik-Software R in der Version 4.3.1 verwendet. R ist eine domänenspezifische Programmiersprache zur Beschreibung statistischer Probleme (vgl. Parr [2014] 104). Die Daten der 60 Reviewenden aus der Umfrage wurden nach der Ausgabe des SQL-Skripts durch SoSci-Survey mithilfe von PHPMyAdmin in die MySQL-Datenbank importiert, die in Abschnitt 5.1 bereits angelegt wurde. Zuerst wird die Population der Studie anhand der Kontrollvariablen in Abschnitt 6.1 dargestellt. Anschließend werden auf die erhobenen Daten der Lesbarkeitsstudie in Abschnitt 6.2 sowie auf die Daten des statischen Clean-Code-Analyse-Tools in Abschnitt 6.3 die Verfahren der deskriptiven Statistik angewandt, um die die Daten weitergehend zu beschreiben. Schlussendlich werden die in Abschnitt 4.2 aufgestellten Hypothesen mithilfe von Regressionsanalysen in Abschnitt 6.4 getestet, um die Forschungsfrage zu beantworten.

6.1. Populationsbeschreibung

Die 60 Teilnehmenden der Studie haben zwischen 286,00 und 44.669,00 Sekunden (M = 3,80, SD = 6.744,37) für die Beantwortung des Fragebogens benötigt. M beschreibt hierbei den Mittelwert, wobei SD die Standardabweichung angibt. Die Teilnehmenden haben sich dabei sehr unterschiedlich viel Zeit für die Beantwortung des Fragebogens genommen. 18 Reviewende haben das Gender weiblich angegeben, während 42 das Gender männlich angegeben haben. Das Gender divers wurde von keiner befragten Person ausgewählt. Die Teilnehmenden sind zum Zeitpunkt der Studie zwischen 18 und 63 Jahre alt (M = 24,70, SD = 7,11). 39 der Teilnehmenden wurden über die Universität erreicht, 2 Teilnehmende über

die Firma, 4 Teilnehmende über ein Online-Forum sowie 15 Teilnehmende über sonstige Kanäle. Die Erfahrung im Programmieren wurde anhand der Anzahl von Jahren, die die teilnehmende Person bereits programmiert, abgefragt. Dabei besaßen die Teilnehmenden zwischen 0,50 und 41,00 Jahren (M = 3,65, SD = 5,66) Programmiererfahrung. acht der Reviewenden hatten noch keine Erfahrung mit der OOP, während 52 der Reviewenden bereits Erfahrung mit der OOP besaßen. Mit dem Konzept des Clean-Codes hatten demgegenüber acht Teilnehmender der Studie bereits Erfahrung, während 52 noch keinerlei Erfahrung mit Clean-Code gesammelt hatten. Schlussendlich wurde die Erfahrung mit der Programmiersprache C++ zwischen 0,00 und 32,00 Jahren angegeben (M = 1,71, SD = 4,33).

6.2. Deskriptive Werte der Evaluationsitems aus der Lesbarkeitsstudie

Nachdem die Kontrollvariablen deskriptiv beschrieben wurden, werden im Folgenden die übrigen UVs beschrieben, wobei es sich um die Bewertungen der Programmcodes aus der Lesbarkeitsstudie handelt. Aus den Antworten bezüglich der Lesbarkeit der vier Kategorien Variablen, Methoden, Klassen und Allgemein/Global lassen sich Bewertungen für jeden Programmcode bezüglich jeder der Kategorien berechnen, die die UVs darstellen.

Die Tabelle 6.1 zeigt die deskriptiven Daten zu den Bewertungen der Lesbarkeit für die vier Kategorien aufgeteilt auf die fünf Code-Beispiele.

Kat.	**Min.**	**Max.**	**M**	**SD**	**Code-Beispiel**
V	3,00	10,00	7,97	1,90	1
M	5,11	10,00	8,36	1,37	1
K	3,33	10,00	8,41	1,59	1
A/G	2,00	10,00	8,33	1,77	1
V	2,00	8,00	4,83	1,38	2
M	2,00	8,11	5,37	1,52	2
K	2,00	10,00	6,60	2,37	2
A/G	2,00	8,00	4,07	2,33	2

Kat.	Min.	Max.	M	SD	Code-Beispiel
V	4,50	10,00	8,01	1,38	3
M	4,67	10,00	8,23	1,40	3
K	2,00	10,00	7,31	2,04	3
A/G	2,00	10,00	6,83	2,25	3
V	3,50	10,00	7,53	1,49	4
M	3,67	9,22	5,86	1,48	4
A/G	2,00	10,00	5,00	2,00	4
V	2,50	10,00	6,46	1,79	5
M	2,78	10,00	6,37	1,69	5
K	2,00	10,00	6,01	2,10	5
A/G	2,00	10,00	6,00	2,05	5

Tabelle 6.1.: Mittelwerte, Kennwerte und Standardabweichungen je Code-Beispiel (N = 60)

Legende: [M] Methoden, [K] Klassen, [V] Variablen, [A] Allgemein, [G] Global

Die deskriptiven Werte für die Evaluierung der Lesbarkeit über alle Code-Beispiele hinweg sind in Tabelle 6.2 abgebildet.

Kat.	Min.	Max.	M	SD
V	2,00	10,00	6,97	2,00
M	2,00	10,00	6,84	1,93
K	2,00	10,00	7,08	2,22
A/G	2,00	10,00	6,05	2,55

Tabelle 6.2.: Mittelwerte, Kennwerte und Standardabweichungen der Lesbarkeitsstudie über alle fünf Code-Beispiele

Stichprobengrößen: [V, M, A / G] N = 300, [K] N = 250

6.3. Deskriptive Werte der Evaluation des statischen Clean-Code-Analyse-Tools

Nachdem die Daten der Lesbarkeitsstudie deskriptiv dargestellt wurden, werden nun die Beobachtungen des statischen Clean-Code-Analyse-Tools bezüglich der Sauberkeit von Programmcode deskriptiv beschrieben.
Die einzelnen Beobachtungen des statischen Clean-Code-Analyse-Tools sind in Tabelle 6.3 angegeben. In den weiteren Spalten sind die Werte der Kategorien für die einzelnen Code-Beispiele eingetragen. Obwohl im vierten Code-Beispiel keine Klassen vorkommen, gibt das Tool hierfür einen Wert von 10,00 aus. Da das Tool korrekten Code erwartet, wurden bei diesem Code-Beispiel zu Beginn einige Klassen hinzugefügt. Im Abschnitt 6.4 zur Inferenzstatistik werden diese Werte jedoch nicht mit in die Analyse einbezogen, da die Daten bei der Lesbarkeitsstudie nicht erhoben wurden, sodass hier kein zusätzlicher Bias entsteht.

Kat.	**Code-Beispiel 1**	**Code-Beispiel 2**	**Code-Beispiel 3**	**Code-Beispiel 4**	**Code-Beispiel 5**
V	8,64	9,84	8,53	8,02	8,81
M	8,33	4,38	8,96	3,25	8,82
K	10,00	9,23	9,55	10,00	9,59
A/G	8,83	8,36	9,13	7,40	8,47

Tabelle 6.3.: Einschätzung der Code-Beispiele in den vier Kategorien durch das statische Clean-Code-Analyse-Tool

Die deskriptiven Daten für die Evaluierung aller Code-Beispiele des statischen Clean-Code-Analyse-Tools sind in Tabelle 6.4 beschrieben.

Kat.	**Min.**	**Max.**	**M**	**SD**
V	8,02	9,84	8,77	0,60
M	3,25	8,96	6,75	2,43
K	9,23	10,00	9,67	0,29
A/G	7,40	9,13	8,44	0,59

Tabelle 6.4.: Mittelwerte, Kennwerte und Standardabweichungen des statischen Clean-Code-Analyse-Tools über alle fünf Code-Beispiele (N = 5)

6.4. Inferenzstatistik

Um die in Abschnitt 4.2 aufgestellten Hypothesen zu analysieren, wurden Modelle der multiplen linearen Regression erstellt, womit die korrespondierenden Nullhypothesen überprüft wurden. Mithilfe der linearen Regression können Merkmalszusammenhänge der AVs und UVs untersucht werden (vgl. Bortz and Weber [2005]: 201). Bei der multiplen linearen Regression wird ein Modell der Form

$$Y = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + ... + \beta_n X_n + \epsilon$$

erstellt, wobei Y die AV, β_0 den y-Achsenabschnitt, β_i die geschätzten Regressionskoeffizienten der UVs X_i und ϵ eine Fehlervariable darstellt (vgl. Fahrmeir et al. [2011]: 495). Durch das Einfügen der Kontrollvariablen als weitere UVs in das Modell können bei der multiplen linearen Regression Moderatorvariablen identifiziert werden, die den Zusammenhang zwischen zwei Merkmalen beeinflussen (vgl. Bortz and Weber [2005]: 463). Die Regressionskoeffizienten β_i geben an, um wie viele Einheiten der AV die AV steigt, insofern die jeweilige UV um eine Einheit in der Einheit der UV steigt (vgl. Imai [2017]: 148), wenn die Werte der übrigen UVs festgehalten werden (vgl. Fahrmeir et al. [2011]: 494). Die Fehlervariable ϵ umfasst die unbeobachteten Zufallsvariablen, für die wie bei der AV eine Normalverteilung angenommen wird (vgl. Fahrmeir et al. [2011]: 495). Regressionskoeffizienten β_i sind dabei statistisch signifikant, sobald der Wert 0 außerhalb des Konfidenzintervalls liegt (vgl. Bortz and Weber [2005]: 194). Das heißt, der Wert 0 ist zu einer Wahrscheinlichkeit p der wahre Wert des Regressionskoeffizientens (vgl. Bortz and Weber [2005]: 194). Beträgt p beispielsweise einen Wert von kleiner als 0,01, kann eine Nullhypothese, die dem statistischen Zusammenhang der beiden Merkmale widerspricht, auf dem 1 %-Niveau verworfen werden (vgl. Bortz and Weber [2005]: 217). Der Wert unter dem p liegen muss, um die Nullhypothese verwerfen zu können, wird mit dem Term α beschrieben (vgl. Imai [2017]: 350). In der vorliegenden Arbeit wird wie in den Hypothesentests von Imai [2017] ein α von 0,05 festgelegt (vgl. Imai [2017]: 350). Darüber hinaus gibt R^2 bei einer linearen Regression den Anteil der erklärten Streuung von der Gesamtstreuung an, sodass es ein Maß zur Evaluierung der Qualität eines Regressionsmodells ist (vgl. Fahrmeir et al. [2011]: 498). Außerdem gibt es den Standardfehler der Residuen, der nicht mit dem Fehlerterm ϵ zu verwechseln ist. Der Standardfehler der Residuen beschreibt

den Wert, den die geschätzten Regressionskoeffizienten im Modell im Durchschnitt um den wahren Wert der Verteilung streuen (vgl. Imai [2017]: 324). Darüber hinaus ist die F-Statistik ein wichtiger Wert. Die statistische Signifikanz Ergebnisses des Overall-F-Tests sagt aus, ob die Prädiktoren des Modells überhaupt bei der Erklärung der AV helfen (vgl. Fahrmeir et al. [2016]: 458).
Im Folgenden werden die Ergebnisse der Hypothesentests dargestellt. Dabei werden Testmodelle zur Überprüfung der Hypothesen H_{0i}, welche die Negationen der Hypothesen H_i darstellen, mithilfe der multiplen linearen Regression entworfen und getestet. Das Modell zur Testung der Hypothese H_{01} bezüglich der Kategorie Variablen besitzt den folgenden Aufbau:

$$VariablenAnalyzerScore = \beta_0 + \beta_1 VariablenReviewscore + \beta_2 Reviewdauer + \beta_3 Gender + \beta_4 Alter + \beta_5 Beworben + \beta_6 Programmiererfahrung + \beta_7 OOP + \beta_8 CleanCode + \beta_9 CppErfahrung + \epsilon$$

AV: **UV:**	**Variablen-Analyzer-Score**	**Methoden-Analyzer-Score**	**Klassen-Analyzer-Score**	**Gesamt-Analyzer-Score**
VariablenReviewScore	−0,164***	-	-	-
MethodenReviewScore	-	0,645***	-	-
KlassenReviewScore	-	-	0,039***	-
AllgemeinerReviewScore	-	-	-	0,085***
Reviewdauer	0,00000	-0,000001	0,00000	-0,00000
Gender	0,022	-0,084	-0,012	0,026
Alter	0,010	-0,018	-0,002	-0,003
Beworben	-0,005	-0,035	-0,004	0,002
Programmiererfahrung	0,011	-0,080	-0,005	-0,003
OOP	0,028	0,184	-0,019	-0,027
CleanCode	-0,075	-0,018	0,008	0,001
CppErfahrung	-0,015	0,111	0,007	0,002
Konstante	9,626***	2,846***	9,400***	8,009***
Beobachtungen	300	300	240	300
R^2	0,286	0,249	0,093	0,132
Korr. R^2	0,264	0,226	0,057	0,105

AV: **UV:**	**Variablen-Analyzer-Score**	**Methoden-Analyzer-Score**	**Klassen-Analyzer-Score**	**Gesamt-Analyzer-Score**
Std.-Fehler der Residuen	0,513 (df = 290)	2,142 (df = 290)	0,266 (df = 230)	0,555 (df = 290)
F-Statistik	12,923*** (df = 9; 290)	10,694*** (df = 9; 290)	2,611*** (df = 9; 230)	4,911*** (df = 9; 290)
Notiz: *p<0,1; **p<0,05; ***p<0,01				

Tabelle 6.5.: Regressionsergebnisse der vier Modelle zu Variablen, Methoden, Klassen und Allgemein/ Global

Tabelle 6.5 zeigt die Ergebnisse der multiplen linearen Regression. Dabei sind in der ersten Spalte die UVs X_i platziert, während rechts von den UVs die jeweiligen geschätzten Regressionskoeffizienten β_i der Stichprobe angegeben werden. Die Konstante ist gleichbedeutend mit dem y-Achsenabschnitt β_0. Die AV in der ersten Spalte markiert, um welches Modell der linearen Regression es sich handelt.
In dem Modell bezüglich Variablen (dritte Spalte) sinkt die AV VariablenAnalyzerScore um 0,16 Einheiten für jede Einheit, die die UV VariablenReviewScore steigt. Dabei besitzt die UV VariablenReviewScore einen p-Wert von kleiner als 0,01, sodass eine Wahrscheinlichkeit von weniger als 1 % für einen wahren Wert 0 des Regressionskoeffizienten besteht. Daraus folgt die Ablehnung der Hypothese H_{01}, sodass H_1 angenommen werden kann. Des Weiteren ist β_0 ebenfalls statistisch signifikant ($p < 0{,}01$), sodass VariablenAnalyzerScore einen Wert von 2,85 annimmt, wenn die UVs null sind. R^2 besitzt einen Wert von 0,29, sodass 29 % der Streuung der AV durch das Modell erklärt werden. Der Standardfehler beträgt 0,51, sodass die geschätzten Regressionskoeffizienten im Mittel um 0,51 Einheiten um den wahren Wert der Verteilung streuen. Darüber hinaus ist der Overall-F-Test statistisch signifikant ($p < 0{,}01$), sodass die Prädiktoren zur Erklärung der Varianz der AV beitragen.
Zur Testung der Hypothese H_{02} bezüglich der Kategorie Methoden wird erneut ein Modell der multiplen linearen Regression aufgestellt.

$$MethodenAnalyzerScore = \beta_0 + \beta_1 MethodenReviewScore + \beta_2 Reviewdauer + \beta_3 Gender + \beta_4 Alter + \beta_5 Beworben + \beta_6 Programmiererfahrung + \beta_7 OOP +$$

$$\beta_8 CleanCode + \beta_9 CppErfahrung + \epsilon$$

In den Ergebnissen des Modells (siehe Tabelle 6.5, vierte Spalte) steigt die AV MethodenAnalyzerScore um 0,65 Einheiten für jede Einheit, die die UV MethodenReviewScore ansteigt. Erneut ist der p-Wert kleiner als 0,01, sodass die Hypothese H_{02} verworfen und die Alternativhypothese H_2 angenommen werden kann. Darüber hinaus ist β_0 ebenfalls statistisch signifikant ($p < 0{,}01$), sodass wenn die UVs null sind, VariablenAnalyzerScore einen Wert von 2,85 annimmt. R^2 liegt bei 0,25 und der Standardfehler der Residuen bei 2,14. Der Overall-F-Test ist erneut statistisch signifikant ($p < 0{,}01$).
Bei der Testung der Hypothese H_{03} in Bezug auf die Kategorie der Klassen ergibt sich bei der multiplen linearen Regression ein weiteres Modell.

$$KlassenAnalyzerScore =$$
$$\beta_0 + \beta_1 KlassenReviewScore + \beta_2 Reviewdauer + \beta_3 Gender + \beta_4 Alter + \beta_5 Beworben +$$
$$\beta_6 Programmiererfahrung + \beta_7 OOP + \beta_8 CleanCode + \beta_9 CppErfahrung + \epsilon$$

Die Resultate der Analyse sind in der fünften Spalte von Tabelle 6.5 vermerkt. Dabei steigt die AV KlassenAnalyzerScore um 0,04 Einheiten, wenn die UV KlassenReviewScore um eine Einheit steigt, wobei eine statistische Signifikanz vorliegt ($p < 0{,}01$). Das heißt, die Hypothese H_{03} kann verworfen werden sowie die Alternativhypothese H_3 angenommen werden kann. Des Weiteren ist erneut der y-Achsenabschnitt β_0 statistisch signifikant ($p < 0{,}01$), sodass der Wert für KlassenAnalyzerScore bei 9,40 liegt, insofern die UVs bei null liegen. Die Güte des Modells wird mit einem R^2 von 0,10 angegeben, während der Standardfehler bei 0,27 liegt. Des Weiteren ist das Ergebnis des Overall-F-Tests statistisch signifikant ($p < 0{,}01$).
Schlussendlich wird zum Testen der Hypothese H_{04} bezüglich der Kategorie Allgemein/Global erneut ein Hypothesentest mithilfe der multiplen linearen Regression durchgeführt. Dabei ergibt sich ein weiteres Modell.

$$GesamtAnalyzerScore = \beta_0 + \beta_1 AllgemeinerReviewScore + \beta_2 Reviewdauer +$$
$$\beta_3 Gender + \beta_4 Alter + \beta_5 Beworben + \beta_6 Programmiererfahrung + \beta_7 OOP +$$
$$\beta_8 CleanCode + \beta_9 CppErfahrung + \epsilon$$

Die Ergebnisse der Regression (siehe Tabelle 6.5, fünfte Spalte) zeigen die statistische Signifikanz der UV AllgemeinerReviewScore ($p < 0{,}01$). Dabei steigt die

AV GesamtAnalyzerScore um 0,09 Einheiten für jede Einheit, die Allgemeiner-ReviewScore ansteigt. Daraus folgt die Ablehnung der Hypothese H_{04}, wobei die Alternativhypothese H_4 angenommen werden kann. Außerdem ist erneut der y-Achsenabschnitt β_0 statistisch signifikant ($p < 0{,}01$). Das bedeutet, wenn die UVs null sind, beträgt der Wert für die AV 8,00. Die Qualität des Regressionsmodells wird mit einem R^2 von 0,13 beziffert, wobei der Standardfehler 0,56 beträgt. Darüber hinaus ist das Ergebnis des Overall-F-Tests erneut statistisch signifikant ($p < 0{,}01$).

7. Diskussion

Nachdem das implementierte statische Clean-Code-Analyse-Tool evaluiert wurde, werden die Ergebnisse in diesem Kapitel diskutiert. Dazu wird in Abschnitt 7.1 zuerst die Forschungsfrage mithilfe der Ergebnisse beantwortet. Daraufhin werden in Abschnitt 7.2 die Bedrohungen der Validität der Ergebnisse beschrieben. Am Ende des Kapitels findet außerdem in Abschnitt 7.3 ein Vergleich mit ähnlichen Code-Analyse-Tools aus der Literatur statt.

7.1. Beantwortung der Forschungsfrage

Die Forschungsfrage, inwiefern statische Code-Analyse-Tools nützlich bei der Erstellung von Clean-Code sind, lässt sich nach der quantitativen Datenanalyse beantworten. Alle vier Hypothesen bezüglich der Kategorien Variablen, Methoden, Klassen und Allgemein/Global, die sich aus dem Clean-Code-Konzept ergeben, konnten infolge der Hypothesentests angenommen werden. Der Clean-Code-Ansatz ist eine starke Theorie, dessen Heuristiken sowohl vom statischen Clean-Code-Analyse-Tool, als auch von den Items der Lesbarkeitsstudie abgefragt wurden. Auf Grundlage der Gegebenheiten von statistisch signifikanten Zusammenhängen sowie starker Theorie, lassen sich Kausalzusammenhänge für die Grundgesamtheit annehmen (vgl. Bühner and Ziegler [2009]: 590). Diese Gegebenheiten sind in den Forschungsergebnissen gegeben, sodass ein Kausalzusammenhang angenommen werden kann. Das heißt, statische Clean-Code-Analyse-Tools erkennen Defekte bezüglich unsauberem Code, sodass sie nützlich bei dem Verfassen von Clean-Code sind.
Darüber hinaus zeigen die Streudiagramme der vier AVs in einem Streudiagramm bezüglich der vier korrespondierenden wichtigsten UVs wie dünn die Datenbasis von fünf Code-Beispielen ist. Diese Streudiagramme sind in Abbildung A.13, Abbildung A.14, Abbildung A.15 und Abbildung A.16 zu sehen. Die AVs würden hier mehr Datenpunkte benötigen, um ein klareres Bild zu zeichen, sodass die

Streudiagramme wenig Aussagekraft besitzen. Dieser Umstand gilt dabei für alle vier herausgearbeiteten Kategorien der Clean-Code-Heuristiken. Hier könnten zukünftige Arbeiten die Lücke schließen.
Des Weiteren fällt der negative Einfluss des VariblenReviewScores auf den VariablenAnalyzerScore in den Ergebnissen auf. Hier würde die Theorie eine positiven Einfluss erwarten lassen. Eine Erklärung könnte beinhalten, dass während der Studie ausschließlich Items zur Benennung von Variablen involviert waren. Solche Items machen den größten Anteil der variablenspezifischen Heuristiken aus, sind aber besonders schwierig syntaktisch zu analysieren. Wieso es dennoch zu statistischen Signifikanzen bei der Regressionsanalyse kommt, muss in weiteren Studien ergründet werden.

7.2. Bedrohungen der Validität

Ein weiteres wichtiges Thema ist sind potenzielle Bedrohungen der Validität der Ergebnisse. Die interne Validität der vorliegenden Arbeit wird verringert, indem die Reviewenden der Lesbarkeitsstudie nicht exakt der gleiche Programmcode wie das Tool bewertet haben. Da das Tool zeitlich nach der Studie realisiert wurde, konnten Zwänge bei der Entwicklung des Tools während der Lesbarkeitsstudie nicht einbezogen werden. Des Weiteren wurde die Studie online durchgeführt, wodurch sie weniger überwacht stattgefunden hat als kontrollierte Studien-Designs. Demgegenüber wurde versucht durch das Abfragen möglichst vieler und relevanter Kontrollvariablen, die während der statistischen Analyse konstant gehalten werden können, die interne Validität zu erhöhen. Insbesondere das Wissen über das Clean-Code-Konzept oder die Programmiererfahrung (in C++) wurden als potenzielle Verzerrungen erkannt und in den Fragebogen einbezogen. Außerdem wurde die Dauer des Code-Reviews als wichtige Kontrollvariable hinzugefügt, da bei einem Online-Fragebogen die Dauer der Beantwortung des Fragebogens stark variiert.
Die Reliabilität des Code-Reviews ist eingeschränkt. Durch das Corona-Semester haben einige Universitäts-Module, in denen die Studie durchgeführt wurde, online stattgefunden. Andere Module fanden im Hörsaal der Universität statt. Durch diese spezielle Situation könnten Teilnehmende der Lesbarkeitsstudie besonders aufgeregt oder besonders abgelenkt gewesen sein, wobei diese Situation nicht reproduzierbar ist.

Eine weitere wichtige Eigenschaft von Forschungsergebnissen ist die externe Validität. Durch den Fokus auf Studierende bei der Befragung könnte die externe Validität eingeschränkt sein. Durch eine breiter gestreute Zufallsstichprobe, die sich nicht nur eine bestimmte Gruppe von Entwicklerinnen und Entwicklern bezieht, könnte die externe Validität erhöht werden.

7.3. Ausblick und Vergleich mit ähnlichen Code-Analyse-Tools

Nach der Implementierung stellt sich die Frage, wie sich das Tool zu ähnlichen Tools aus der Literatur verhält. Ein Aspekt, der an der vorliegenden Arbeit neu ist, ist die Verwendung des Clean-Code-Ansatzes zur Erkennung von Defekten. In der Literatur spielt bisher die Erkennung von Code-Smells oder AntiPatterns die größte Rolle bei der Erkennung von Defekten. Clean-Code kommt hingegen als Analyse-Kriterium für Defekte im Programmcode bisher nicht vor. Darüber hinaus ist die Analyse der Programmiersprache C++ eine Seltenheit, die lediglich iPlasma ebenfalls beherrscht. Das bisherige Feld an Code-Analyse-Tools aus der Literatur ist demgegenüber sehr auf die Analyse von Code-Beispielen der Programmiersprache Java fokussiert. Des Weiteren nutzen lediglich LINT, jCosmo und DECOR aus der Literatur einen eigenen Parser, während die anderen Tools Parser von Dritten verwenden. Hier liegt ein großer Vorteil der Implementierung in der vorliegenden Arbeit, da die Anpassungsmöglichkeiten eines selbst-geschriebenen Parsers für unterschiedlichste Forschungsinteressen im Feld der Code-Analyse flexibler für zukünftige Forschung ist. Außerdem ist die Implementierung in der vorliegenden Arbeit neben einer Clean-Code Bewertungsapplikation ebenfalls ein Clean-Code Refactoring-Tool. Dieses Merkmal bietet im untersuchten Feld bisher nur das Code-Smell Analyse-Tool JDeodorant sowie Stench Blossom. Stench Blossom besitzt darüber hinaus eine graphische Oberfläche, die die erkannten Defekte je nach Art des Defekts in Views hervorheben (vgl. Fontana et al. [2015]: 17). Eine solche Funktionalität bietet das in der vorliegenden Arbeit implementierte Tool nicht, da es bisher lediglich als Kommandozeilen-Tool umgesetzt ist.
Des Weiteren verwenden Tools aus der Literatur in vielen Fällen etablierte Metriken, insbesondere aus der OOP, zur Erkennung von Defekten im Programmcode. Die vorliegende Arbeit verwendet demgegenüber eine ad-hoc-Metrik, die binär die Verletzung der Clean-Code-Heuristiken erkennt, ohne den Schweregrad zu bestim-

men. Hier sind insbesondere die Tools iPlasma, BSDT, JCodeOdor und TACO im Vorteil, da sie sich auf etablierte Metriken aus der Literatur zur Bestimmung von Defekten berufen. Außerdem gibt es dynamische Code-Analyse-Tools in der Literatur, wie beispielsweise LINT, die toten Code erkennen. Da toter Code für statische Code-Analyse sehr schwierig zu erkennen ist, versucht das in der vorliegenden Ausarbeitung implementierte Tool sich mit einer Annäherung, die jedoch nicht perfekt ist. Hier könnte durch die weitergehende Implementation hin zu einem vollwertigen Compiler noch viel erreicht werden.
Außerdem verwenden einige Tools Schwellenwerte zur Erkennung von Defekten, die abhängig von der Auswahl des Anwendenden sind oder nutzen überwachtes Lernen um die Präferenz des Anwendenden in die Erkennungsalgorithmen einzubeziehen. Diese Tools sind jCosmo, DECOR, Stench Blossom und Fica. Eine solche Implementierung ist in der vorliegenden Arbeit nicht realisiert worden. Zukünftige Forschungsarbeiten im Bereich der Clean-Code-Analyse sollten diesem Thema mehr Raum geben.
Darüber hinaus nutzen die Tools in der Literatur oft etablierte Metriken zur Evaluation des eigenen Tools, worunter Genauigkeit, Präzision und Recall fallen. Dazu werden annotierte Programmcodes verwendet, deren Code-Smells bereits bekannt sind, um so das eigene Code-Analyse-Tool kostengünstig zu evaluieren. Hier hat die vorliegende Arbeit mit der Lesbarkeitsstudie einen anderen Fokus gesetzt. Da das Code-Verständnis für andere Entwicklerinnen und Entwickler im Fokus des Clean-Code-Ansatzes steht, bot sich eine Evaluation mit Menschen an. Die Wahrnehmung von der Lesbarkeit von Code lässt sich mithilfe der quantitativen Datenanalyse und einer ausreichend großen Stichprobe verallgemeinern. Durch diese Verallgemeinerung ist der gewählte Ansatz genauer wie die Verwendung von annotiertem Programmcode. Bei diesen Annotationen muss eine Expertin oder ein Experte die Regel des jeweiligen Defekts im Programmcode interpretieren und im Code suchen sowie anmerken. Dadurch ist die Datenbasis an Defekten im Code deutlich subjektiver als bei einer quantitativen Datenanalyse mit einer ausreichend großen Stichprobe. Dafür entsteht jedoch der Vorteil, deutlich größere Mengen an Programmcode mit dem eigenen Code-Analyse-Tool testen zu können. In der vorliegenden Arbeit werden demgegenüber lediglich fünf kleinere Code-Beispiele aus der Literatur getestet, die aufwendig übersetzt werden mussten damit die Studienteilnehmenden die Programmiersprache der Beispiele kennen.

8. Fazit

Die statische Code-Analyse ist ein Ansatz, das Erkennen von Defekten im Programmcode kostengünstiger zu machen. Zwar ist die Software-Inspektion die beste Variante Defekte im Programmcode zu erkennen, doch erfordert sie erfahrene Reviewende. Die statische Code-Analyse ist hier ein Ansatz, die Kosten durch die automatische Erkennung von Defekten zu verringern.
Guter Code sollte nicht wie frühere Ansätze im Sinne der Maschine geschrieben sein, sondern im Sinne des Menschen. Diesen Ansatz verfolgt der Clean-Code-Ansatz. Die vorliegende Arbeit hatte das Ziel, einen Beitrag zu leisten guten Code zugänglicher zu machen. Dazu eignete sich insbesondere die Idee, ein Refactoring-Tool zum Verfassen von Clean-Code zu entwerfen. Mit einem solchen Tool können Studierende und andere Entwickerinnen und Entwicklern Feedback zur Qualität ihres Codes erhalten. Durch die vom Tool erhaltenen Hinweise könnte es in der Zukunft einfacher werden, den eigenen Code fortlaufend zu evaluieren und Clean-Code im Ergebnis zu erzielen. Dadurch könnte sich die Zugänglichkeit zum guten Code sowohl für diejenigen, die den Code schreiben, als auch für diejenigen, die ihn lesen (müssen), erhöhen.
Aus dieser Motivation heraus wurde ein statisches Clean-Code-Analyse-Tool anhand der von Maschinen lesbaren Regeln des Clean-Codes konzeptionalisiert, für die Programmiersprache C++ implementiert sowie hauptsächlich mithilfe von Studierenden der Informatik und der Ingenieurswissenschaften evaluiert. Dabei wurde die Forschungsfrage gestellt, ob statische Code-Analyse-Tools nützlich beim Verfassen von Clean-Code sind.
Die Forschungsfrage konnte bejaht werden, indem Hypothesen bezüglich der Erkennung der Sauberkeit der vier Kategorien Variablen, Methoden, Klassen und Allgemein/Global aufgestellt und getestet wurden. Das in der vorliegenden Arbeit implementierte statische Code-Analyse-Tool analysiert die Clean-Code-Heuristiken aus allen diesen vier Kategorien und gibt Bewertungen für die Sauberkeit des gegebenen C++-Codes ab. Das Tool wurde dabei mit fünf angepassten Code-

Beispielen, die aus dem Buch des Clean-Code-Erfinders stammen und von Java in C++ übersetzt wurden, getestet. Bei einer Evaluation bezüglich der Lesbarkeit nahezu deckungsgleicher C++-Programmcodes mithilfe eines checklistenbasierten Code-Reviews und 60 Studierenden konnten die Hypothesen angenommen werden. Da die vorliegende Arbeit mit einem Clean-Code-Analyse-Tool einen noch nicht in der Literatur umgesetzten Ansatz verfolgt, sollten die Ergebnisse in weiteren Studien repliziert werden. Darüber hinaus sollten zukünftige Studien die Probleme der vorliegenden Arbeit aufgreifen. Diese Probleme sind insbesondere der Fokus auf Studierende während der quantitativen Evaluation des Tools sowie der Verzicht auf die Nutzung etablierter Metriken. Clean-Code-Heuristiken könnten durch Verwendung etablierter Metriken aus der OOP deutlich genauer gemessen werden, als es in der vorliegenden Arbeit getan wurde, sodass hier noch großes Verbesserungspotenzial besteht. Darüber hinaus könnten ML-Ansätze die Menge der erkannten Defekte durch die Implementierung weiterer Clean-Code-Heuristiken erhöhen. Dazu müsste ein Klassifizierer auf die Erkennung semantischer Inhalte der Heuristiken hin trainiert werden. Solche Ansätze werden bereits bei Tools zur Erkennung von (relevantem) dupliziertem Code verwendet, den das in der vorliegenden Ausarbeitung realisierte Tool nicht erkennt. Außerdem könnte das statische Code-Analyse-Tool zu einem vollwertigen dynamischen Code-Analyse-Tool erweitert werden, um weitere Clean-Code-Heuristiken (insbesondere toter Code) erkennen zu können. Darüber hinaus gibt es ein Tool in der Literatur, welches verschiedene Views für unterschiedliche Arten von Defekten anzeigt. Zukünftige Arbeiten könnten sich darauf fokussieren, ein vollständiges Front-End für das in der vorliegenden Arbeit implementierte Tool zu konstruieren. Dabei würde die Zugänglichkeit des Tools und damit des Schreibens von gutem Code insgesamt weiter erhöht werden. Alles in allem trägt die vorliegende Arbeit durch die Exploration des neuen Forschungsgebiets der Clean-Code-Analyse-Tools einen kleinen Teil dazu bei, weitere Forschungsarbeiten in diesem Bereich anzuregen.

Literaturverzeichnis

Abid, S., Abdul Basit, H., and Arshad, N. (2015). Reflections on Teaching Refactoring: A Tale of Two Projects. In *Proceedings of the 2015 ACM Conference on Innovation and Technology in Computer Science Education*, pages 225–230, Vilnius Lithuania. ACM.

Aho, A. V., Lam, M. S., Sethi, R., Ullman, J. D., and Leuschel, M., editors (2008). *Compiler: Prinzipien, Techniken und Werkzeuge*. it informatik. Pearson Studium, München, 2., aktualisierte aufl. [der engl. ausg.] edition.

Alls, J. (2020). *Clean Code in C#*. Packt Publishing, Birmingham, 1st edition edition. OCLC: 1195920489.

Anaya, M. (2018). *Clean code in Python: refactor your legacy code base*. Packt Publishing, Birmingham.

Appel, A. W. and Ginsburg, M. (1998). *Modern compiler implementation in C*. Cambridge University Press, Cambridge ; New York, rev. and expanded ed edition.

Arisholm, E., Briand, L., and Foyen, A. (2004). Dynamic coupling measurement for object-oriented software. *IEEE Transactions on Software Engineering*, 30(8):491–506.

Aurum, A., Petersson, H., and Wohlin, C. (2002). State-of-the-art: software inspections after 25 years. *Software Testing, Verification and Reliability*, 12(3):133–154.

Ball, T. (1999). The concept of dynamic analysis. *ACM SIGSOFT Software Engineering Notes*, 24(6):216–234. Publisher: ACM New York, NY, USA.

Bardas, A. G. (2010). Static code analysis. *Journal of Information Systems & Operations Management*, 4(2):99–107. Publisher: Romanian-American University.

Beck, K. (2003). *Test-driven development: by example*. The Addison-Wesley signature series. Addison-Wesley, Boston.

Beck, K., Fowler, M., and Beck, G. (1999). Bad smells in code. *2017 IEEE 37th Central America and Panama Convention (CONCAPAN XXXVII)*, 1(1999):75–88.

Boomsma, H., Hostnet, B. V., and Gross, H.-G. (2012). Dead code elimination for web systems written in PHP: Lessons learned from an industry case. In *2012 28th IEEE International Conference on Software Maintenance (ICSM)*, pages 511–515, Trento, Italy. IEEE.

Bortz, J. and Weber, R. (2005). *Statistik für Human- und Sozialwissenschaftler: mit 242 Tabellen*. Springer-Lehrbuch. Springer Medizin, Heidelberg, 6., vollst. überarb. und aktualisierte aufl edition.

Brown, W. J., editor (1998). *AntiPatterns: refactoring software, architectures, and projects in crisis*. Wiley computer publishing. Wiley, New York.

Buchmann, A. and Smolarek, R. (2005). *SQL, MySQL 5 interaktiv*. Omnigena-Lehrbuch. dpunkt-Verl. [u.a.], Heidelberg, 1. aufl edition.

Böttcher, A., Thurner, V., and Müller, G. (2011). Kompetenzorientierte Lehre im Software Engineering. In *SEUH*, pages 33–39.

Bühner, M. and Ziegler, M. (2009). *Statistik für Psychologen und Sozialwissenschaftler*. PS Psychologie. Pearson, München, korr. nachdr. edition.

Chong, C. Y., Thongtanunam, P., and Tantithamthavorn, C. (2021). Assessing the Students' Understanding and their Mistakes in Code Review Checklists: An Experience Report of 1,791 Code Review Checklist Questions from 394 Students. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering Education and Training (ICSE-SEET)*, pages 20–29, Madrid, ES. IEEE.

Danphitsanuphan, P. and Suwantada, T. (2012). Code smell detecting tool and code smell-structure bug relationship. In *2012 Spring Congress on Engineering and Technology*, pages 1–5. IEEE.

Drozdz, M., Kourie, D. G., Watson, B. W., and Boake, A. (2006). Refactoring tools and complementary techniques. *IEEE International Conference on Computer Systems and Applications*, 6:685–688.

Ebenau, R. G. and Strauss, S. S. (1994). *Software inspection process*. McGraw-Hill systems design & implementation series. McGraw-Hill, New York.

Erlenkötter, H. (2018). *C++: objektorientiertes Programmieren von Anfang an*. Number 60077 in rororo rororo Computer. Rowohlt Taschenbuch Verlag, Reinbek bei Hamburg, 18. auflage edition.

Fagan, M. (2002). A History of Software Inspections. In Broy, M. and Denert, E., editors, *Software Pioneers*, pages 562–573. Springer Berlin Heidelberg, Berlin, Heidelberg.

Fagan, M. E. (1976). Design and code inspections to reduce errors in program development. *IEEE*, 15(3):182–211.

Fahrmeir, L., Heumann, C., Künstler, R., Pigeot, I., and Tutz, G. (2016). *Statistik: der Weg zur Datenanalyse*. Springer-Lehrbuch. Springer Spektrum, Berlin Heidelberg, 8., überarbeitete und ergänzte auflage edition.

Fahrmeir, L., Künstler, R., Pigeot, I., and Tutz, G. (2011). *Statistik: der Weg zur Datenanalyse*. Springer-Lehrbuch. Springer, Berlin Heidelberg, 7. auflage, korrigierter nachdruck edition.

Fischer, C. N., Cytron, R. K., and LeBlanc, R. J. (2010). *Crafting a compiler*. Pearson, Boston Munich, global ed edition.

Fokaefs, M., Tsantalis, N., and Chatzigeorgiou, A. (2007). JDeodorant: Identification and removal of feature envy bad smells. In *2007 IEEE International Conference on Software Maintenance*, pages 519–520. IEEE. ISSN: 1063-6773.

Fontana, F. A., Braione, P., and Zanoni, M. (2012). Automatic detection of bad smells in code: An experimental assessment. *The Journal of Object Technology*, 11(2):1–5.

Fontana, F. A., Ferme, V., Zanoni, M., and Roveda, R. (2015). Towards a prioritization of code debt: A code smell intensity index. In *2015 IEEE 7th International Workshop on Managing Technical Debt (MTD)*, pages 16–24. IEEE.

Fontana, F. A., Zanoni, M., Marino, A., and Mantyla, M. V. (2013). Code smell detection: Towards a machine learning-based approach. In *2013 IEEE International Conference on Software Maintenance*, pages 396–399. IEEE.

Fowler, M. (2019). *Refactoring: improving the design of existing code*. Addison-Wesley signature series. Addison-Wesley, second edition edition. OCLC: on1064139838.

Fowler, M. and Beck, K. (2007). *Refactoring: [oder] wie Sie das Design vorhandener Software verbessern*. Programmer's choice. Addison Wesley, studentenausg.; repr. edition.

Fraser, C. W. and Hanson, D. R. (1995). *A retargetable C compiler: design and implementation*. Benjamin/Cummings Pub. Co, Redwood City, CA.

Fu, S. and Shen, B. (2015). Code bad smell detection through evolutionary data mining. In *2015 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 1–9. IEEE.

Gamma, E., Beck, K., and Gamma, E. (2004). *Eclipse erweitern: Prinzipien, Patterns und Plug-Ins*. Open source library. Addison-Wesley, München.

Gantenbein, R. E. (1991). Dynamic binding in strongly typed programming languages. *Journal of Systems and Software*, 14(1):31–38.

Gilb, T., Graham, D., and Finzi, S. (1993). *Software inspection*. Addison-Wesley, Wokingham, England ; Reading, Mass.

Gomes, I., Morgado, P., Gomes, T., and Moreira, R. (2009). An overview on the static code analysis approach in software development. *Faculdade de Engenharia da Universidade do Porto, Portugal*, pages 1–8.

Goodliffe, P. (2007). *Code Craft: the practice of writing excellent code*. No Starch Press, San Francisco, Calif.

Gosain, A. and Sharma, G. (2015). Static Analysis: A Survey of Techniques and Tools. In Mandal, D., Kar, R., Das, S., and Panigrahi, B. K., editors, *Intelligent Computing and Applications*, volume 343, pages 581–591. Springer India, New Delhi. Series Title: Advances in Intelligent Systems and Computing.

Gumm, H.-P., Sommer, M., and Hesse, W. (2011). *Einführung in die Informatik*. Oldenbourg, München, 9., vollst. überarb. aufl edition.

Guttag, J. (2016). *Introduction to computation and programming using Python: with application to understanding data*. The MIT Press, Cambridge, Massachusetts, second edition edition.

Haykin, S. S. (2016). *Neural networks and learning machines*. Pearson, Delhi Chennai, third edition, indian subcontinent adaptation edition.

Hedtstück, U. (2012). *Einführung in die theoretische Informatik: formale Sprachen und Automatentheorie*. Oldenbourg, München, 5., überarb. aufl edition.

Hyde, R. (2006). *Write great code. 2: Thinking low-level, writing high-level*. No Starch Press, San Francisco, Calif.

Imai, K. (2017). *Quantitative social science: an introduction*. Princeton University Press, Princeton. OCLC: ocn958799734.

Johnson, S. C. (1977). *Lint, a C program checker*. Bell Telephone Laboratories Murray Hill.

Johnson, S. C. (1978). A portable compiler: theory and practice. In *Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages - POPL '78*, pages 97–104, Tucson, Arizona. ACM Press.

Kak, A. C. (2003). *Programming with objects: a comparative presentation of object-oriented programming with C++ and Java*. Wiley, Hoboken.

Kernighan, B. W. and Ritchie, D. M. (1988). *The C programming language*. Prentice Hall, Englewood Cliffs, N.J, 2nd ed edition.

Kiselyov, O., Shan, C.-c., and Sabry, A. (2006). Delimited dynamic binding. *ACM SIGPLAN Notices*, 41(9):26–37.

Koenig, A. (1998). Patterns and antipatterns. In *The patterns handbooks: techniques, strategies, and applications*, pages 383–389. Cambridge University Press.

Kofler, M. (2005). *MySQL 5: Einführung, Programmierung, Referenz*. Open source library. Addison-Wesley, München, 3. aufl., [nachdr.] edition.

Künneth, T. and Ullenboom, C. (2009). *Einstieg in Eclipse 3.5: aktuell zu "Galileo"; effiziente Java-Entwicklung mit Eclipse ; Plug-ins, Web- und RCP-Anwendungen erstellen ; inkl. Refactoring, Debugging, Subversion, CVS u.v.m.* Galileo Computing. Galileo Press, Bonn, 3., aktualisierte aufl edition.

Lampe, J. (2020). *Clean Code für Dummies.* für Dummies. Wiley-VCH Verlag GmbH & Co. KGaA.

Lanza, M. and Marinescu, R. (2006). *Object-oriented metrics in practice: using software metrics to characterize, evaluate, and improve the design of object-oriented systems ; with 8 tables.* Springer.

Latte, B., Henning, S., and Wojcieszak, M. (2019). Clean code: On the use of practices and tools to produce maintainable code for long-living. In *Proceedings of the Workshops of the Software Engineering Conference 2019*, pages 96–99, Stuttgart.

Liggesmeyer, P. (2002). *Software-Qualität: Testen, Analysieren und Verifizieren von Software.* Spektrum, Akad. Verl, Heidelberg Berlin.

Ljung, K. and Gonzalez-Huerta, J. (2022). "To Clean Code or Not to Clean Code" A Survey Among Practitioners. In Taibi, D., Kuhrmann, M., Mikkonen, T., Klünder, J., and Abrahamsson, P., editors, *Product-Focused Software Process Improvement*, volume 13709, pages 298–315. Springer International Publishing, Cham. Series Title: Lecture Notes in Computer Science.

Louridas, P. (2006). Static code analysis. *IEEE Software*, 23(4):58–61.

Martin, R. C. (2009). *Clean Code: Refactoring, Patterns, Testen und Techniken für sauberen Code.* mitp, Frechen Hamburg, deutsche ausgabe, 1. auflage edition.

Martin, R. C. (2011). *Clean Coder: Verhaltensregeln für professionelle Programmierer.* Programmer's choice. Addison-Wesley.

Mayer, C. (2022). *The art of clean code: best practices to eliminate complexity and simplify your life.* No Starch Press, San Francisco.

McMeekin, D. A., Von Konsky, B. R., Chang, E., and Cooper, D. J. (2009). Evaluating Software Inspection Cognition Levels Using Bloom's Taxonomy. In *2009 22nd Conference on Software Engineering Education and Training*, pages 232–239, Hyderabad, India. IEEE.

Moha, N., Gueheneuc, Y.-G., Duchien, L., and Le Meur, A.-F. (2010). DECOR: A method for the specification and detection of code and design smells. *IEEE*, 36(1):20–36.

Murphy-Hill, E. and Black, A. P. (2010). An interactive ambient visualization for code smells. In *Proceedings of the 5th international symposium on Software visualization*, pages 5–14. ACM.

Mäntylä, M. V. and Lassenius, C. (2007). Subjective evaluation of software evolvability using code smells: An empirical study. *Elsevier Inc*, 11(3):395–431.

Nongpong, K. (2012). Integrating "code smells"detection with refactoring tool support. *Thesis*.

Or-Meir, O., Nissim, N., Elovici, Y., and Rokach, L. (2020). Dynamic Malware Analysis in the Modern Era—A State of the Art Survey. *ACM Computing Surveys*, 52(5):1–48.

Osherove, R. (2010). *The art of unit testing: with examples in .NET*. Manning, Greenwich, nachdr. edition.

Padolsey, J. (2020). *Clean code in JavaScript: develop reliable, maintainable, and robust JavaScript*. PACKT Publishing, Birmingham. OCLC: 1137845041.

Palomba, F. (2015). Textual Analysis for Code Smell Detection. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, pages 769–771, Florence, Italy. IEEE.

Palomba, F., Bavota, G., Di Penta, M., Oliveto, R., De Lucia, A., and Poshyvanyk, D. (2013). Detecting bad smells in source code using change history information. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 268–278. IEEE.

Parr, T. (2014). *The definitive ANTLR 4 reference*. The pragmatic programmers. The Pragmatic Bookshelf, Dallas, Texas Raleigh, North Carolina, book version: p 2.0 edition.

Poetzsch-Heffter, A. (2000). *Konzepte objektorientierter Programmierung: mit einer Einführung in Java*. Springer, Berlin Heidelberg.

Post, U. (2021). *Besser coden: best practices für clean code*. Rheinwerk Computing. Rheinwerk Verlag, Bonn, 2., aktualisierte und erweiterte auflage edition.

Prahofer, H., Angerer, F., Ramler, R., Lacheiner, H., and Grillenberger, F. (2012). Opportunities and challenges of static code analysis of IEC 61131-3 programs. In *Proceedings of 2012 IEEE 17th International Conference on Emerging Technologies & Factory Automation (ETFA 2012)*, pages 1–8, Krakow, Poland. IEEE.

Priese, L. and Erk, K. (2018). *Theoretische Informatik: eine umfassende Einführung*. Lehrbuch. Springer Vieweg, Berlin, 4., aktualisierte und erweiterte auflage edition.

Rao, A. A. and Reddy, K. N. (2008). Detecting bad smells in object oriented design using design change propagation probability matrix. In *Proceedings of the international multiconference of engineers and computer scientists*, volume 1, pages 1–7.

Ratz, D., Schulmeister-Zimolong, D., Seese, D. G., and Wiesenberger, J. (2018). *Grundkurs Programmieren in Java*. Hanser, München, 8., aktualisierte auflage edition.

Rohatgi, A., Hamou-Lhadj, A., and Rilling, J. (2008). An Approach for Mapping Features to Code Based on Static and Dynamic Analysis. In *2008 16th IEEE International Conference on Program Comprehension*, pages 236–241, Amsterdam. IEEE.

Russell, S. J. and Norvig, P. (2012). *Künstliche Intelligenz: ein moderner Ansatz*. it - Informatik. Pearson, Higher Education, München Harlow Amsterdam, 3., aktualisierte auflage edition.

Sadowski, C., Söderberg, E., Church, L., Sipko, M., and Bacchelli, A. (2018). Modern code review: a case study at google. In *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice*, pages 181–190, Gothenburg Sweden. ACM.

Schiedermeier, R. (2013). *Programmieren mit Java. 2*. Pearson Higher Education, München.

Schildt, H. (2007). *Java: the complete reference ; [... updated and expanded for Java SE 6]*. McGraw-Hill, New York, 7. ed edition.

Shull, F., Rus, I., and Basili, V. (2000). How perspective-based reading can improve requirements inspections. *Computer*, 33(7):73–79.

Stroustrup, B. (2015). *Die C++-Programmiersprache: aktuell zum C++ 11-Standard*. Hanser, München.

Stubblebine, T., Klicman, P., and Stubblebine, T. (2004). *Reguläre Ausdrücke - kurz & gut*. O'Reillys Taschenbibliothek. O'Reilly, Beijing Köln, dt. ausg., 1. aufl edition.

Ullenboom, C. (2023). *Java ist auch eine Insel: Einführung, Ausbildung, Praxis*. Rheinwerk Computing. Rheinwerk Verlag, Bonn, 16., aktualisierte und überarbeitete auflage, 1., korrigierter nachdruck 2023 edition.

Van Den Brand, M., Van Deursen, A., Heering, J., De Jong, H., De Jonge, M., Kuipers, T., Klint, P., Moonen, L., Olivier, P., Scheerder, J., Vinju, J., Visser, E., and Visser, J. (2001). The Asf+Sdf Meta-Environment. *Electronic Notes in Theoretical Computer Science*, 44(2):3–8.

Van Emden, E. and Moonen, L. (2002). Java quality assurance by detecting code smells. In *Ninth Working Conference on Reverse Engineering, 2002. Proceedings.*, pages 97–106. IEEE Comput. Soc.

Wilhelm, R., Seidl, H., and Hack, S., editors (2012). *Übersetzerbau. 2: Syntaktische und semantische Analyse / Reinhard Wilhelm; Helmut Seidl; Sebastian Hack*. eXamen.press. Springer Vieweg, Berlin Heidelberg.

William, F. (1992). Obdyke: Refactoring object-oriented frameworks. *Doktorarbeit, University of Illinois at Urbana-Champaign*, 3:4.

Windler, C. and Daubois, A. (2022). *Clean code in PHP: expert tips and practices to write beautiful, human friendly, and maintainable PHP*. Packt Publishing, Birmingham. OCLC: 1350639174.

Wolf, J. (2019). *C von A bis Z: das umfassende Handbuch*. Rheinwerk Computing. Rheinwerk Verlag, Bonn, 3., aktualisierte und erweiterte auflage 2009, 7. korrigierter nachdruck 2019 edition.

Yang, J., Hotta, K., Higo, Y., Igaki, H., and Kusumoto, S. (2012). Filtering clones for individual user based on machine learning analysis. In *2012 6th International Workshop on Software Clones (IWSC)*, pages 76–77. IEEE.

Zeppenfeld, K. (2004). *Objektorientierte Programmiersprachen: Einführung und Vergleich von Java, C++, C#, Ruby*. Spektrum Akademischer Verlag, Heidelberg Berlin, 1. auflage edition.

A. Anhang

A.1. Tabellen

Code-Smell	Beschreibung/Anweisung	Ref.
Duplizierter Code	Dieselbe Code-Struktur existiert an mehreren Orten im Programmcode.	S. 76
Lange Methode	Es gibt keine Möglichkeit, nur mithilfe eines Namens und ohne Kommentare, genau zu beschreiben, was die Funktion tut.	S. 76f
Große Klasse	Die Klasse besitzt zu viele Attribute, was zu dupliziertem Code führt, der vermieden werden sollte.	S. 78
Lange Parameterliste	Es werden zu lange Listen von Übergabeparametern an Methoden verwendet (inkl. Flag-Argumente und fehlende Einkapselung).	S. 78f.
Divergierende Änderungen	Wird eine Klasse regelmäßig verändert, führt eine Veränderung immer zu notwendigen Anpassungen an den gleichen Entitäten.	S. 79
Schrotkugeln herausoperieren	Immer wenn eine kleine Änderung durchgeführt wird, müssen viele kleine Veränderungen bei vielen Klassen durchgeführt werden.	S. 80
Neid	Eine Methode ruft mehr Methoden und Daten von anderen Klassen ab, als von der eigenen Klasse.	S. 80f
Datenklumpen	Daten, die immer nah zueinander im Code verwendet werden, sollten in eine eigene Klasse eingekapselt werden.	S. 81
Neigung zu elementaren Typen	Wann immer es möglich ist, sollten Objekte verwendet werden.	S. 81f
Switch-Befehle	Switch-Befehle sollten in die Methode einer Klasse extrahiert werden, bei der die Case-Befehle geerbte Methoden der Elternklasse aufrufen.	S. 82

Code-Smell	Beschreibung/Anweisung	Ref.
Parallele Verwaltungshierarchien	Immer wenn eine neue Kindklasse einer Klasse gebildet wird, muss auch eine neue Kindklasse einer anderen Klasse gebildet werden.	S. 83
Faule Klasse	Eine Klasse hat infolge des Refaktorisierens so viel Funktionalität eingebüßt, dass sie in eine andere Klasse integriert werden sollte.	S. 83
Spekulative Allgemeinheit	Wurden Vorbereitungen für Funktionalitäten getroffen, die noch nicht implementiert sind, sollten diese Vorbereitungen entfernt werden.	S. 83f
Temporäre Felder	Werden Instanzvariablen nur unter manchen Umständen initialisiert und sind ansonsten ein NULL-Wert, sollten sie immer initialisiert werden.	S. 84
Nachrichtenketten	Es sollten keine Ketten von Delegationen verwendet werden, um auf entfernte Funktionalitäten oder Daten zuzugreifen.	S. 84
Vermittler	Bestehen zu viele Funktionalitäten einer Klasse lediglich aus dem vermitteln an andere Klassen, sollte sie entfernt werden.	S. 85
Unangebrachte Intimität	Ist die Interaktion zwischen Klassen zu groß, muss sie reduziert werden, indem Methoden in die passende Klasse verschoben werden.	S. 85
Alternative Klassen mit verschiedenen Schnittstellen	Ähnliche Klassen sollten das gleiche Interface umsetzen, um die Austauschbarkeit der Klassen zu ermöglichen.	S. 85f
Unvollständige Bibliotheksklasse	Soll eine Bibliothek erweitert werden, sollten die Methoden der Bibliothek in eine eigene Klasse ausgelagert und dann erweitert werden.	S. 86
Datenklassen	Eine Klasse besitzt ausschließlich Instanzvariblen, Konstruktoren, sowie Getter- und Settermethoden.	S. 86f
Ausgeschlagenes Erbe	Kindklassen erben von ihren Elternklassen, nutzen jedoch die bereitgestellten Daten und Funktionalitäten nicht.	S. 87
Kommentare	Werden Kommentare benötigt, um zu erklären, wie ein Codeabschnitt funktioniert, sollte er refaktorisiert werden.	S. 87f

Tabelle A.1.: Code-Smells Beck et al. [1999], Übersetzungen aus Fowler and Beck [2007]

Code-Smell	Beschreibung/Anweisung	Ref.
Instanceof	Befinden sich mehrere instanceof-Operatoren nah zueinander, sind Vererbungshierarchie oder Methodenüberladen bessere Alternativen.	S. 101
Typecast	Fehler beim Typecasten werden beim Kompilieren nicht erkannt und führen zu Fehlern, weswegen Typecasten vermieden werden sollte.	S. 101

Tabelle A.2.: Code-Smells Van Emden and Moonen [2002]

Code-Smell	Beschreibung/Anweisung	Ref.
Magische Zahl	Eine numerisches Literal wird direkt verwendet, ohne in einer Konstante mit aussagekräftigem Namen gespeichert zu sein.	S. 387

Tabelle A.3.: Code-Smells Drozdz et al. [2006]

Code-Smell	Beschreibung/Anweisung	Ref.
Gottklasse	Wie "Große Klasse", die zu viele Funktionalitäten selbst bereitstellt, anstatt sie an andere Klassen zu delegieren, aber deren Daten exzessiv nutzt.	S. 80
Gehirnmethode	Methode, die wie "Gottklasse"die Funktionalitäten, anstatt eines Programms, von einer Klasse zentralisiert.	S. 92
Gehirnklasse	Wie "Gottklasse", besitzt jedoch weniger Kohäsion zwischen den Methoden der Klasse und ruft weniger Daten anderer Klassen ab. Nutzt Gehirnmethoden.	S. 97
Intensive Kopplung	Eine Methode ruft sehr viele Methoden weniger anderer Klassen ab, die nicht die eigene ist.	S. 120
Verstreute Kopplung	Eine Methode ruft sehr viele Methoden auf, die auf viele verschiedene Klassen verteilt sind.	S. 127
Traditionsbrecher	Eine Kindklasse bietet sehr viele Funktionalitäten an, die mit der Elternklasse nichts zu tun haben.	S. 152

Tabelle A.4.: Code-Smells Lanza and Marinescu [2006]

Code-Smell	Beschreibung/Anweisung	Ref.
Toter Code	Code, der aktuell nicht ausgeführt wird, sollte entfernt werden.	S. 407

Tabelle A.5.: Code-Smells Mäntylä and Lassenius [2007]

Code-Smell	Beschreibung/Anweisung	Ref.
Mysteriöser Name	Die Namen von Entitäten kommunizieren nicht, was sie tun und wie sie verwendet werden.	S. 72
Globale Daten	Keine globalen Daten verwenden, stattdessen Daten in Methoden einkapseln.	S. 74
Wandelbare Daten	Daten sollten nie verändert werden. Beim aktualisieren einer Datenstruktur sollte immer eine neue Instanz erstellt werden.	S. 75
Schleifen	Schleifen sollten durch Pipeline-Operationen ersetzt werden.	S. 79

Tabelle A.6.: Code-Smells Fowler [2019]

Nr.	Kat.	Beschreibung/Anweisung	Ref.	Code-Smell	Syntax-Analyse
8	M	Bei der Überladung von Konstruktoren werden diese als private markiert und von Funktionen aufgerufen, die mit ihrem Namen die übergebenen Argumente beschreiben.	S. 55	-	zum Teil
9	K	Keine Variablen mit dem Zugriffsmodifikator protected verwenden (nur in Notfällen).	S. 113, (S. 173)	-	ja
10	K	Klassenvariablen werden privat gehalten (außer sie sind statisch).	S. 129, (S. 173)	-	ja

Tabelle A.7.: Clean-Code Zugriffsmodifikatoren aus Martin [2009]

Nr.	Kat.	Beschreibung/Anweisung	Ref.	Code-Smell	Syntax-Analyse
11	V, M, K	Zweckbeschreibende Namen: Der Name beschreibt den Inhalt und Zweck möglichst genau.	S. 50	Mysteriöser Name	nein
12	V, M, K	Aussprechbare Namen: Name ist gut aussprechbar, um darüber diskutieren zu können. Wenig Ziffern und bestehend aus ganzen Wörtern.	S. 50	-	nein
13	V, M, K	Suchbare Namen: Der Name soll möglichst gut im Code auffindbar sein. Längere Namen ohne Ziffern sind besser suchbar.	S. 51	-	zum Teil
14	V, M, K	Keine Verwendung der Ungarischen Notation. Hier wird der Datentyp im Bezeichner angegeben.	S. 52f	-	ja
15	V, M, K	Keine Verwendung von Member-Präfixen. Variablen in Klassen sollten nicht mit dem Präfix "m_"versehen werden.	S. 53	-	ja
16	V, M, K	Keine Verwendung mentaler Mappings. Man sollte Namen nicht im Kopf in andere Namen übersetzen müssen (meint vor allem einbuchstabige Platzhalter außerhalb von Schleifen).	S. 54	-	nein
17	V	Schleifenzähler dürfen i oder j oder k heißen, aber niemals nur aus einem kleinen L oder einem großen l bestehen.	S. 54	-	zum Teil
18	K	Klassen und Objekte besitzen Namen, die aus einem Substantiv bestehen und nicht zu allgemein sind (und auf keinen Fall ein Verb).	S. 54	-	zum Teil
19	M	Funktionsnamen bestehen aus einem Verb, oder aus einem Ausdruck mit einem Verb.	S. 55	-	zum Teil
20	V, M, K	Es werden keine humorvollen Namen verwendet.	S. 55	-	nein
21	V, M, K	Es wird ein Wort pro Konzept verwendet. Z. B. es werden nicht fetch, retrieve und get als Namen für ähnliche Methoden verwendet.	S. 56	-	nein

Nr.	Kat.	Beschreibung/Anweisung	Ref.	Code-Smell	Syntax-Analyse
22	V, M, K	Es werden technische Namen aus der Informatik (Lösungsdomäne) verwendet, sonst aus der Problemdomäne (Fachbereich).	S. 57	-	nein
23	V, M, K	Bedeutungsvollen Kontext hinzufügen (beispielsweie für Vornamen-Variable außerhalb von Adressenklasse). Ein Präfix (Beispiel: addr) ist die Lösung.	S. 57f	-	nein
24	V, M, K	Keinen überflüssigen Kontext beifügen. Beispielsweise Präfixe für Namen innerhalb einer Klasse, dessen Klassenname den Kontext bereits beschreibt.	S. 60	-	zum Teil
25	M	Beschreibende Namen. Ein langer beschreibender Name ist besser als ein kurzer gehemnisvoller Name.	S. 70	-	nein
26	M	Funktionsnamen sollten konsistent vergeben werden und so eine Geschichte erzählen.	S. 70	-	zum Teil
27	K	Einen prägnanten Klassennamen verwenden, der auf die eine Verwantwortlichkeit der Klasse verweist.	S. 176	-	nein

Tabelle A.8.: Clean-Code Bezeichner aus Martin [2009]

Nr.	Kat.	Beschreibung/Anweisung	Ref.	Code-Smell	Syntax-Analyse
28	M	Funktionen sind bestenfalls zwischen 20 und 100 Zeilen lang. Niemals länger als 150 Zeilen.	S. 64	Lange Methode	ja
29	M	Jede Zeile einer Funktion ist nicht mehr als 150 Zeichen breit.	S. 64	-	ja
30	M	Einrückungstiefe innerhalb von Funktionen sind nicht tiefer als 1 oder 2 Ebenen.	S. 65	-	ja
31	M	Blöcke in if-, else-, while- und ähnlichen Anweisungen sind eine Zeile lang sein und beinhalten einen Funktionsaufruf mit einem aussagekräftigen Namen.	S. 65	-	zum Teil

Nr.	Kat.	Beschreibung/Anweisung	Ref.	Code-Smell	Syntax-Analyse
32	M	Switch-Anweisungen in eine kleine Klasse aulagern, die ihre Methoden erbt, die in den (einzeiligen) Cases aufgerufen werden.	S. 68f	Switch-Befehle	ja
33	M	Die Körper von Try/Catch-Blöcken sind eine Zeile lang und enthalten lediglich einen Funktionsaufruf.	S. 78	-	ja
34	A/G	Obegrenze von 500 Zeilen pro Datei. Kleinere Dateien sind leichter zu verstehen als größere. Präfiert werden maximal 200 Zeilen je Datei.	S. 111	-	ja
35	K	Zeilen dürfen nicht länger als 120 Zeichen breit sein.	S. 119	-	ja
36	M	Im Funktionskopf befindet sich kein Leerzeichen zwischen dem Funktionsnamen und der öffnenden Klammer für die Übergabeparameter.	S. 120	-	ja
37	M	Im Funktionskopf befindet sich zwischen den Kommata der einzelnen Übergabeparameter und dem Datentyp des nächsten Übergabeparameters ein Leerzeichen.	S. 120	-	ja
38	V	Bei der Initialisierung von Variablen befinden sich jeweils ein Leerzeichen links und rechts des Zuweisungsoperators.	S. 120	-	ja
39	K	(Nicht-verschachtelte) Klassen nicht einrücken.	S. 123	-	ja
40	K	Funktionen in Klassen genau einmal einrücken.	S. 123	-	ja
41	M	Implementierungen in Funktionen genau 1 mal mehr einrücken, wie die übergeordnete Funktion.	S. 123	-	ja
42	M	Implementierungen der Blöcke von Anweisungen genau 1 mal mehr einrücken, wie die übergeordnete Anweisung.	S. 123	-	ja

Tabelle A.9.: Clean-Code Formatierung aus Martin [2009]

Nr.	Kat.	Beschreibung/Anweisung	Ref.	Code-Smell	Syntax-Analyse
43	M	Niedrige Anzahl an Übergabeparametern: 0 (niladisch) ist am besten, gefolgt von 1 (monadisch), gefolgt von 2 (dyadisch). Mehr als 3 (polyadisch) Argumente vermeiden.	S. 71	Lange Parameterliste	ja
44	M	Bei zu vielen Übergabeparametern die Argumente in seperate Klassen einhüllen, um die Anzahl der Argumente zu verringern.	S. 74	Lange Parameterliste	ja
45	M	Keine Flag-Argumente (boolean) an Funktionen übergeben.	S. 72	Lange Parameterliste	ja
46	M	Output Argumente vermeiden, da „this“ diese Rolle einnimmt. Outputargumente sind übergebene Objekte, die in der Funktion verändert werden (!= Rückgabeargumente).	S. 76	-	nein
47	M	Exceptions verwenden, statt die Rückgabe von Fehlercodes in großen if-/else-Verkettungen.	S. 78	-	nein
48	M	Alle Fehler-Codes, die von einer Funktion zurückgegeben werden, in einer Klasse oder einem Enum definieren.	S. 79	-	nein
49	M	Eine Funktion darf keinen Nullpointer zurückgeben.	S. 147	-	zum Teil
50	M	Beim Funktionsaufruf darf nicht der Nullpointer an Funktionen übergeben werden.	S. 148	-	zum Teil

Tabelle A.10.: Abbildung Clean-Code Funktionparameter aus Martin [2009]

Nr.	Kat.	Beschreibung/Anweisung	Ref.	Code-Smell	Syntax-Analyse
51	A/G	Juristische Kommentare dürfen nicht zu lang werden (Z. B. Copyright des Autors oder der Autorin).	S. 87f	-	nein
52	A/G	Informierende Kommentare durch besseren Code obsolet machen.	S. 88	Kommentare	nein

Nr.	Kat.	Beschreibung/Anweisung	Ref.	Code-Smell	Syntax-Analyse
53	A/G	Bei verschiedenen Alternativen der Implementierung die Absicht des Autors bzw. der Autorin in einem Kommentar festhalten.	S. 88f	-	nein
54	A/G	Kommentare für Klarstellungen zum Verständnis nutzen, falls der Code nicht angepasst werden (z. B. bei Methoden aus einer Standardbibliothek).	S. 89f	-	nein
55	A/G	Warnungen vor Konsequenzen in Form von Kommentaren für andere Programmierende einfügen. Beispeilsweise wenn das Auführen einer Methode sehr lange dauert.	S. 90	-	nein
56	A/G	Keine TODO-Kommentare (Kommentare die beschreiben, was noch erledigt werden sollte) verwenden.	S. 91	-	nein
57	A/G	Kommentare nutzen, um die Bedeutung von einem bestimmten Codeabschnitt zu verstärken.	S. 91f	-	nein
58	A/G	Keine unverständlichen Kommentare verwenden.	S. 92	-	nein
59	A/G	Keine redundanten Kommentare verwenden.	S. 93	-	nein
60	A/G	Keine verpflichtenden Kommentare verwenden (z. B. Zwang widerstehen, Javadocs in Java über jeder Funktion einzufügen).	S. 96	-	nein
61	A/G	Keine Tagebuch-Kommentare verwenden.	S. 97	-	nein
62	A/G	In einer Zeile hinter eine schließende Klammer keinen Kommentar platzieren.	S. 101	-	ja
63	A/G	Keinen auskommentierten Code im Programm belassen.	S. 102	-	nein
64	M	Keine Funktionsheader-Kommentare verwenden.	S. 102	-	zum Teil

Tabelle A.11.: Clean-Code Kommentare aus Martin [2009]

Nr.	Kat.	Beschreibung/Anweisung	Ref.	Code-Smell	Syntax-Analyse
65	M	Keine Duplikate verwenden. Kommen Teile des Programmcodes häufiger vor, sollte er in eine eigene Funktion ausgelagert werden.	S. 80	Dupli-zierter Code	zum Teil
66	M	Keine goto-Anweisungen verwenden.	S. 81	-	ja
67	K	Keine Getter- und Settermethoden für den Zugriff auf die Klassenvariablen bereitstellen.	S. 129f	-	zum Teil
68	M	Sogenannte „tote Funktionen“ (Funktionen, die nie aufgerufen werden) aus dem Programm löschen.	S. 340	Toter Code	nein
69	K	Keine Duplikate (identischer Code, der mehrfach vorkommt) verwenden.	S. 342	Dupli-zierter Code	zum Teil
70	M	Bedingungen in boolesche Methoden mit einem aussagekräftigen Methodennamen einkapseln, statt viele komplizierte Vergleiche zu verwenden.	S. 356	-	ja
71	M	Positiv formulierte Bedingungen, statt negativ formulierten Bedingungen verwenden.	S. 356	-	ja

Tabelle A.12.: Clean-Code NoGos aus Martin [2009]

Nr.	Kat.	Beschreibung/Anweisung	Ref.	Code-Smell	Syntax-Analyse
72	K	Klassenvariablen ohne eine Zeile Abstand und ohne Kommentar zwischen den Variablen zu Beginn der Klasse deklarieren.	S. 113	-	ja
73	K	Methoden in einer Klasse nach den Klassenvariablen mit jeweils 1 Zeile Abstand definieren.	S. 113	-	ja
74	M	Lokale Variablen in Funktionen direkt am Beginn der Funktion deklarieren.	S. 114	-	ja

Nr.	Kat.	Beschreibung/Anweisung	Ref.	Code-Smell	Syntax-Analyse
75	M	Kontrollvariablen für Schleifen wenn möglich innerhalb von Schleifen, sonst direkt vor der Schleife oder am Anfang eines Blocks deklarieren.	S. 115	-	nein
76	M	Aufgerufene Funktion unter dem Block definieren, der die Funktion aufruft.	S. 116f	-	ja
77	M	Die Funktionsparameter (und eine mögliche throw-Anweisung) befinden sich in einer Zeile.	S. 121f	-	ja
78	K	In einer Klasse zu Beginn zuerst die public static Variablen, dann die private static Variablen und anschließend die private Variablen deklarieren.	S. 173	-	ja
79	A/G	Import-Deklarationen am Anfang des Programms hintereinander platzieren.	S. 362	-	ja

Tabelle A.13.: Clean-Code Entitäten-Platzierung aus Martin [2009]

Kat.	CC-Heuristik-Nr.	Checklistenfrage	Item
V	11	Wie aussagekräftig und zweckbeschreibend sind die Variablennamen?	Die Variablennamen sind aussagekräftig und zweckbeschreibend.
V	12	Wie gut aussprechbar sind die Variablennamen?	Die Variablennamen sind gut aussprechbar.
V	13	Wie gut suchbar sind die Variablennamen in einer Entwicklungsumgebung / einem Editor?	Die Variablennamen sind in einer Entwicklungsumgebung / einem Editor gut suchbar.
V	21	Wie konsistent ist die Namensgebung der Variablen?	Die Namensgebung der Variablennamen ist konsistent.
M	11	Wie aussagekräftig und zweckbeschreibend sind die Methodennamen?	Die Methodennamen sind aussagekräftig und zweckbeschreibend.

Kat.	CC-Heuristik-Nr.	Checklistenfrage	Item
M	21	Wie konsistent ist die Namensgebung der Methoden?	Die Namensgebung der Methoden ist konsistent.
M	12	Wie gut aussprechbar sind die Methodennamen?	Die Methodennamen sind gut aussprechbar.
M	40	Wie konsistent sind die Einrückungen in den Methoden?	Die Einrückungen in den Methoden sind konsistent.
M	26	Wie eindeutig sind die Funktionalitäten der Methoden?	Die Funktionalitäten der Methoden sind eindeutig.
M	4	Besitzt die Methode unerwartete Effekte abseits der im Methodennamen beschriebenen Funktionalitäten?	Die Methode besitzt keine unerwarteten Effekte abseits der im Methodennamen beschriebenen Funktionalitäten.
M	1	Wie bewerten Sie die Komplexität der Methoden?	Die Komplexität der Methoden ist passend.
M	31	Wie bewerten Sie die Komplexität der Blöcke (if/else, ..)?	Die Komplexität der Blöcke (if/else, ..) ist passend.
K	11	Wie aussagekräftig und zweckbeschreibend ist der Klassenname?	Der Klassenname ist aussagekräftig und zweckbeschreibend.
K	5	Wie bewerten Sie die Komplexität der Klasse?	Die Komplexität der Klasse ist passend.
K	27	Wie eindeutig sind die Funktionalitäten der Klasse?	Die Funktionalitäten der Klassen sind eindeutig.
A/G	-	Lesbarkeit beschreibt den mentalen Aufwand, der benötigt wird, um den Code zu verstehen. Wie bewerten Sie die Lesbarkeit des Codeabschnitts?	Lesbarkeit kann so definiert werden, dass sie den mentalen Aufwand beschreibt, der benötigt wird, um den Code zu verstehen. Die Lesbarkeit des vorliegenden Codes ist angemessen.

Tabelle A.14.: Herleitung Checklisten und Items der Lesbarkeitsstudie

Die CC-Heuristiken und ihre Nummern sind in Tabelle 4.1, Tabelle A.7, Tabelle A.8, Tabelle A.9, Tabelle A.10, Tabelle A.11, Tabelle A.12 und Tabelle A.13 zu finden.

Kontrollvariable	Beschreibung
Reviewdauer	Absolutskalierte Variable, die für eine Beobachtung die Dauer vom Beginn der Beantwortung des Fragebogens bis zum Abschließen des Fragebogens in Sekunden angibt.
Gender	Nominalskalierte Variable, die bei einer Beobachtung das Geschlecht des Teilnehmenden angibt. 1 bedeutet männlich und 0 bedeutet weiblich.
Alter	Absolutskalierte Variable, die für eine Beobachtung das Alter der befragten Person in Jahren angibt.
Beworben	Nominalskalierte Variable, die bei einer Beobachtung angibt, wie die teilnehmende Person auf die Studie gekommen ist. 0 bedeutet Universität, 1 Firma, 2 Online-Forum und 3 Sonstige.
Programmiererfahrung	Absolutskalierte Variable, die für eine Beobachtung die Programmiererfahrung in Jahren für die befragte Person angibt.
OOP	Nominalskalierte Variable, die bei einer Beobachtung angibt, ob die befragte Person bereits Erfahrung mit der OOP besitzt.
CleanCode	Nominalskalierte Variable, die bei einer Beobachtung angibt, ob die befragte Person bereits Erfahrung mit dem Clean-Code-Konzept besitzt.
CppErfahrung	Absolutskalierte Variable, die für eine Beobachtung die Programmiererfahrung in C++ in Jahren für die befragte Person angibt.

Tabelle A.15.: Lesbarkeitsstudie Kontrollvariablen

A.2. Clean-Code Analyse-Tool Ein- und Ausgabe

```
using namespace std;

public class User {
    string getPhraseEncodedByPassword();
};
public class UserGateway {
```

```
    User findByName(string userName);
};
public class Cryptohgrapher {
    string getPhraseEncodedByPassword();
};
public class Session {
    void initialize();
};

class UserValidator {
  private:
    Cryptohgrapher cryptohgrapher;

  public:
    bool checkPassword(string userName, string password);
};

bool UserValidator::checkPassword(string userName, string password)
    {
  User *user = UserGateway::findByName(userName);
  if (user != NULL) {
    string codedPhrase = user -> getPhraseEncodedByPassword();
    string phrase = cryptographer.decrypt(codedPhrase, password);
    if (phrase.compare("Valid␣Password") == 0) {
      Session::initialize();
      return true;
    }
  }
  return false;
}
```

Listing A.1: C++ Clean-Code-Analyse-Tool Input 1

```
#include <iostream>
#include <iomanip>

using namespace std;

class PrintPrimes {
  public:
    friend int main(char* argv[]);
};
```

```
int main(char* argv[]) {
  const int M = 1000;
  const int RR = 50;
  const int CC = 4;
  const int ORDMAX = 30;
  int* P = new int[M + 1];
  int PAGENUMBER;
  int PAGEOFFSET;
  int ROWOFFSET;
  int C;
  int J;
  int K;
  bool JPRIME;
  int ORD;
  int SQUARE;
  int N;
  int* MULT = new int[ORDMAX + 1];

  J = 1;
  K = 1;
  P[1] = 2;
  ORD = 2;
  SQUARE = 9;
  while(K < M) {
    do {
      J = J + 2;
      if (J == ORDMAX) {
        ORD = ORD + 1;
        SQUARE = P[ORD] * P[ORD];
        MULT[ORD - 1] = J;
      }
      N = 2;
      JPRIME = true;
      while (N < ORD && JPRIME) {
        while (MULT[N] < J) {
          MULT[N] = MULT[N] + P[N] + P[N];
          if (MULT[N] == J)
            JPRIME = false;
        }
        N = N + 1;
      }
    } while (!JPRIME);
```

```
    K = K + 1;
    P[K] = J;
  }
  {
    PAGENUMBER = 1;
    PAGEOFFSET = 1;
    while (PAGEOFFSET <= M) {
      cout << "The␣First␣" << M <<
              "␣Prime␣Numbers␣----␣Page␣" << PAGENUMBER;
      cout << "";
      for (ROWOFFSET = PAGEOFFSET; ROWOFFSET < PAGEOFFSET + RR;
         ROWOFFSET++) {
        for (C = 3; C < CC;C++) {
          if (ROWOFFSET + C * RR <= M) {
            cout << setw(2) << P[ROWOFFSET + C * RR];
            cout << "␣";
          }
        }
        PAGENUMBER = PAGENUMBER + 1;
        PAGEOFFSET = PAGEOFFSET + RR * CC;
      }
    }
  }
}
```

Listing A.2: C++ Clean-Code-Analyse-Tool Input 2

```
using namespace std;
generic <typename T>

class File {};
class LineWidthHistogram {};
class list<T> {};
class vector<T> {};
class Integer {};
class FileReader {};
class BufferedReader {};

class CodeAnalyzer {
  private:
    int lineCount;
    int maxLineWidth;
    int widestLineNumber;
```

```
    LineWidthHistogram lineWidthHistogram;
    int totalChars;

    static void findJavaFiles(File& parentDirectory, list<File>
       files);
    void measureLine(string line);
    void recordWidestLine(int lineSize);
    int lineCountForWidth(int width);
    vector<Integer> getSortedWiths();
  public:
    CodeAnalyzer();
    ~CodeAnalyzer();
    static list<File> findJavaFiles(File& parentDirectory);
    void analyzeFile(File& javaFile) throw();
    int getLineCount();
    int getMaxLineWidth();
    int getWidestLineNumber();
    LineWidthHistogram getLineWithHistogram();
    double getMeanLineWidth();
    int getMedianLineWidth();
};

CodeAnalyzer::CodeAnalyzer() {
  LineWidthHistogram lineWidthHistogram = LineWidthHistogram();
}

CodeAnalyzer::~CodeAnalyzer() {
  delete lineWidthHistogram;
}

list<File> CodeAnalyzer::findJavaFiles(File& parentDirectory) {
  list<File> files;
  findJavaFiles(parentDirectory, files);
  return files;
}

void CodeAnalyzer::findJavaFiles(File& parentDirectory, list<File>
   files) {
  for (File file : parentDirectory.listFiles()) {
    if (file.getName().ends_with(".java"))
      files.insert(files.begin(), file);
    else if (file.isDirectory())
```

```
      findJavaFiles(file, files);
  }
}

void CodeAnalyzer::analyzeFile(File& javaFile) throw() {
  FileReader fr = FileReader(javaFile);
  BufferedReader br = BufferedReader(fr);
  string line;
    while (!(line = br.readLine()).empty()) {
      measureLine(line);
    }
}

void CodeAnalyzer::measureLine(string line) {
  lineCount++;
  int lineSize = line.length();
  totalChars += lineSize;
  lineWidthHistogram.addLine(lineSize, lineCount);
  rectordWidestLine(lineSize);
}

void CodeAnalyzer::recordWidestLine(int lineSize) {
  if (lineSize > maxLineWidth) {
    maxLineWidth = lineSize;
    widestLineNumber = lineCount;
  }
}

int CodeAnalyzer::getLineCount() {
  return lineCount;
}

int CodeAnalyzer::getMaxLineWidth() {
  return maxLineWidth;
}

int CodeAnalyzer::getWidestLineNumber() {
  return widestLineNumber;
}

LineWidthHistogram CodeAnalyzer::getLineWithHistogram() {
  return lineWidthHistogram;
```

```
}

double CodeAnalyzer::getMeanLineWidth() {
  return (double)totalChars/lineCount;
}

int CodeAnalyzer::getMedianLineWidth() {
  vector<Integer> sortedWidths = getSortedWidths();
  int cumulativeLineCount = 0;
  for(int width : sortedWidths) {
    cumulativeLineCount += lineCountForWidth(width);
    if (cumulativeLineCount > lineCount/2)
      return width;
  }
  throw Error("Cannot␣get␣here");
}

int CodeAnalyzer::lineCountForWidth(int width) {
  vector<Integer> widths = lineWidthHistogram.getWidths();
  vector<Integer> sortedWiths = widths;
  sort(sortedWiths, sortedWiths + widths.size());
  return sortedWiths;
}
```

Listing A.3: C++ Clean-Code-Analyse-Tool Input 3

```
using namespace std;

public class PageData {};
public class WikiPage {};
public class StringBuffer {};
public class WikiPagePath {};

static string testableHtml(
  PageData pageData,
  bool includeSuiteSetup
) {
  WikiPage *wikiPage = pageData.getWikiPage();
  StringBuffer *buffer = new StringBuffer();
  if (pageData.hasAttribute("Test")) {
    if (includeSuiteSetup) {
      WikiPage *includeSuiteSetup =
        PageCrawlerImpl::getInheritedPage(
```

```
        SuiteResponder::SUITE_SETUP_NAME, *wikipage
      );
    if (suiteSetup != NULL) {
      WikiPagePath *pagePath =
        suiteSetup.getPageCrawler().getFullPath(suiteSetup);
      string pagePathName = PathParser.render(pagePath);
      buffer -> append("!include␣-setup␣.")
             -> append(pagePathName)
             -> append("\n");
    }
  }
  WikiPage *setup =
    PageCrawlerImpl.getInheritedPage("SetUp", wikiPage);
  if (setup != NULL) {
    WikiPagePath *setupPath =
      wikiPage.getPageCrawler().getFullPath(setup);
    string setupPathName = PathParser.render(setupPath);
    buffer -> append("!include␣-setup␣.")
           -> append(setupPathName)
           -> append("\n");
  }
}
buffer.append(pageData.getContent());
if (pageData.hasAttribute("Test")) {
  WikiPage *teardown =
    PageCrawlerImpl.getInheritedPage("TearDown", wikiPage);
  if (teardown != NULL) {
    WikiPagePath tearDownPath =
      wikiPage.getPageCrawler().getFullPath(teardown);
    if (teardown != NULL) {
      WikiPagePath tearDownPath =
        wikiPage.getPageCrawler().getFullPath(teardown);
      string tearDownPathName = PathParser.render(tearDownPath);
      buffer -> append ("\n")
             -> append("!include␣-teardown␣.")
             -> append(tearDownPathName)
             -> append("\n");
    }
    if (includeSuiteSetup) {
      WikiPage *suiteTearDown =
        PageCrawlerImpl.getInheritedPage(
          SuiteResponder.SUITE_TEARDOWN_NAME,
```

```
            wikiPage
          );
      }
      if (suiteTearDown != NULL) {
        WikiPagePath *pagePath =
          suiteTearDown.getPageCrawler().getFullPath(suiteTearDown)
              ;
          buffer -> append("!include␣-teardown␣.")
                 -> append(pagePathName)
                 -> append("\n");
      }
    }
  }
  pageData.setContent(buffer.toString());
  return pageData.getHtml();
}
```

Listing A.4: C++ Clean-Code-Analyse-Tool Input 4

```
using namespace std;

public class ParentWidget {};
public class Pattern {};
public class StringBuffer {};

class BoldWidget : public ParentWidget {
  public:
    static const string REGEXP;
    static const Pattern& pattern;
    ParentWidget* parent;
    BoldWidget(ParentWidget& parent, string text);
    string render();
};
const Pattern& BoldWidget::pattern = Pattern::compile("'''(.+?)'''"
    ,
      Pattern::MULTILINE + Pattern::DOTALL
);
const string BoldWidget::REGEXP = "'''.+?'''";
BoldWidget::BoldWidget(ParentWidget& parent, string text) {
  Matcher* match = pattern.matcher(text);
  match -> find();
  addChildWidgets(match -> group(1));
}
```

```
string BoldWidget::render() {
  StringBuffer *html = new StringBuffer("<br>");
  html -> append(childHtml()).append("<br>");
  return html -> toString();
}
```

Listing A.5: C++ Clean-Code-Analyse-Tool Input 5

```
Es wurden globale Strukturen oder Klassen gefunden, hinter der sich
    keine Leerzeile befindet. Sauberer wäre es, eine Leerzeile vor
   der Definition von globalen Strukturen oder Klassen zu verwenden
   .
In Zeile 7, Spalte 17:
        Die Struktur oder Klasse "BoldWidget" besitzt public oder
           protected Variablen, die nach Möglichkeit vermieden
           werden sollten. Protected Variablen dürfen in Ausnahmefä
           llen verwendet werden.
        Die Struktur oder Klasse "BoldWidget" besitzt Methoden, die
            mehr oder weniger als eine konsistente Einrückung
           besitzen. Sauberer wäre es, wenn jede Methode genau eine
            Einrückung innerhalb der Struktur oder Klasse besitzt.
In Zeile 10, Spalte 34:
        Der Name der Variable, "pattern", enthält nicht nur Groß
           buchstaben und '_' zur Trennung von Wörtern. Dann wäre
           sie am einfachsten suchbar im Code.
In Zeile 11, Spalte 25:
        Der Name der Variable, "parent", enthält nicht nur Groß
           buchstaben und '_' zur Trennung von Wörtern. Dann wäre
           sie am einfachsten suchbar im Code.
In Zeile 12, Spalte 15:
        Die Funktion oder Methode "BoldWidget" verwendet eine
           Anzahl von 2 Übergabeparametern (dyadisch). 1 Parameter
           (monadisch), oder besser noch, 0 Paremeter (niladisch),
           wären noch sauberer.
In Zeile 12, Spalte 36:
        Der Name der Variable, "parent", enthält nicht nur Groß
           buchstaben und '_' zur Trennung von Wörtern. Dann wäre
           sie am einfachsten suchbar im Code.
In Zeile 12, Spalte 49:
        Der Name der Variable, "text", enthält nicht nur Groß
           buchstaben und '_' zur Trennung von Wörtern. Dann wäre
           sie am einfachsten suchbar im Code.
In Zeile 19, Spalte 11:
```

```
        Die Funktion oder Methode "BoldWidget" wird nicht
           aufgerufen und ist damit eine sogenannte "tote Funktion
           ". Methoden oder Funktionen, die nicht verwendet werden,
            sollten aus Platzgründen aus dem Code entfernt werden.
        Die global definierte Funktion oder Methode "BoldWidget"
           besitzt keine Leerzeile vor sich. Sauberer wäre es, vor
           jeder definierten Methode oder Funktion eine Leerzeile
           zur visuellen Abgrenzung zu verwenden.
In Zeile 19, Spalte 24:
        Die Funktion oder Methode "BoldWidget" verwendet eine
           Anzahl von 2 Übergabeparametern (dyadisch). 1 Parameter
           (monadisch), oder besser noch, 0 Paremeter (niladisch),
           wären noch sauberer.
In Zeile 19, Spalte 45:
        Der Name der Variable, "parent", enthält nicht nur Groß
           buchstaben und '_' zur Trennung von Wörtern. Dann wäre
           sie am einfachsten suchbar im Code.
In Zeile 19, Spalte 58:
        Der Name der Variable, "text", enthält nicht nur Groß
           buchstaben und '_' zur Trennung von Wörtern. Dann wäre
           sie am einfachsten suchbar im Code.
In Zeile 24, Spalte 7:
        Die Funktion oder Methode "render" wird nicht aufgerufen
           und ist damit eine sogenannte "tote Funktion". Methoden
           oder Funktionen, die nicht verwendet werden, sollten aus
            Platzgründen aus dem Code entfernt werden.
In Zeile 25, Spalte 21:
        Der Name der Variable, "html", enthält nicht nur Groß
           buchstaben und '_' zur Trennung von Wörtern. Dann wäre
           sie am einfachsten suchbar im Code.
```

Listing A.6: Clean-Code-Analyse-Tool Refactoring-Hinweise Code-Beispiel 5

A.3. Lesbarkeitsstudie

KV01

Was ist Ihr Gender (Geschlecht)?

männlich

weiblich

divers

KV02

Wie alt sind Sie?

Alter (in Jahren):

KV03

Wie haben Sie von der Studie erfahren?

Universität

Firma

Online-Forum

Sonstige

KV04

Wie viele Jahre Erfahrung in Programmierung besitzen Sie circa?

Programmiererfahrung (in Jahren):

KV05

Besitzen Sie bereits Erfahrung in Objektorientierter Programmierung?

Ja

Nein

KV06

Kenne Sie bereits das Clean Code Konzept von Robert C. Martin (Uncle Bob)?

ja

nein

Abbildung A.1.: Lesbarkeitsstudie Fragebogen Kontrollfragen

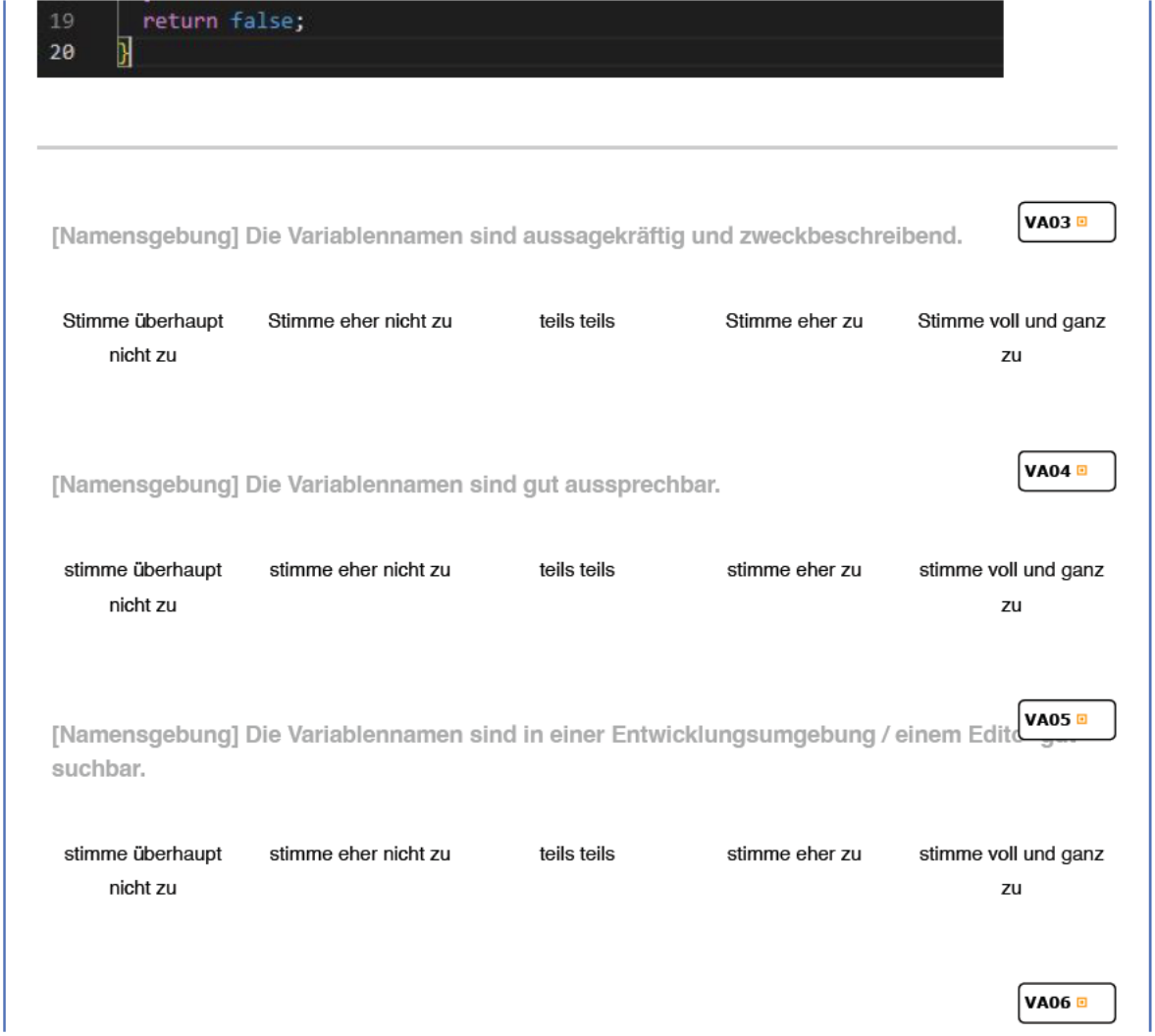

Abbildung A.2.: Fragebogen Beispielfragen

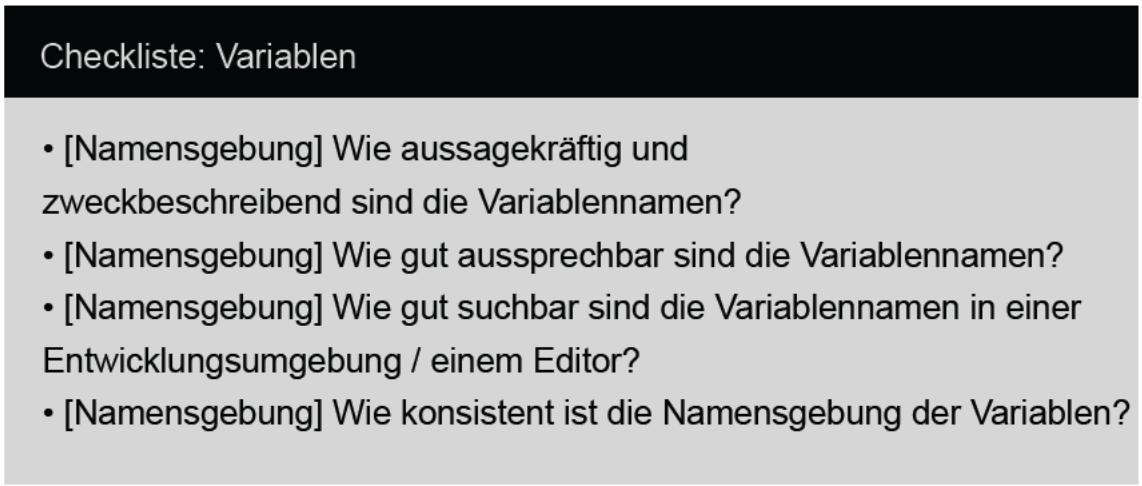

Abbildung A.3.: Lesbarkeitsstudie Checkliste Variablen

Checkliste: Funktionen / Methoden

- [Namensgebung] Wie aussagekräftig und zweckbeschreibend sind die Methodennamen?
- [Namensgebung] Wie konsistent ist die Namensgebung der Methoden?
- [Namensgebung] Wie gut aussprechbar sind die Methodennamen?
- [Formatierung] Wie konsistent sind die Einrückungen in den Methoden?
- [Klarheit] Wie eindeutig sind die Funktionalitäten der Methoden?
- [Komplexität] Besitzt die Methode unerwartete Effekte abseits der im Methodennamen beschriebenen Funktionalitäten?
- [Komplexität] Wie bewerten Sie die Komplexität der Methoden?
- [Komplexität] Wie bewerten Sie die Komplexität der Blöcke (if/else, ..)?

Abbildung A.4.: Lesbarkeitsstudie Checkliste Methoden

Checkliste: Gesamt

- [Klarheit] Lesbarkeit beschreibt den mentalen Aufwand, der benötigt wird, um den Code zu verstehen. Wie bewerten Sie die Lesbarkeit des Codeabschnitts?

Abbildung A.5.: Lesbarkeitsstudie Checkliste Allgemein

```
class PrintPrimes {
  public:
    friend int main(char* argv[]);
};

int main(char* argv[]) {
  const int M = 1000;
  const int RR = 50;
  const int CC = 4;
  const int WW = 10;
  const int ORDMAX = 30;
  int* P = new int[M + 1];
  int PAGENUMBER;
  int PAGEOFFSET;
  int ROWOFFSET;
  int C;
  int J;
  int K;
  bool JPRIME;
  int ORD;
  int SQUARE;
  int N;
  int* MULT = new int[ORDMAX + 1];

  J = 1;
  K = 1;
  P[1] = 2;
  ORD = 2;
  SQUARE = 9;
  while (K < M) {
    do {
      J = J + 2;
      if (J == SQUARE) {
        ORD = ORD + 1;
        SQUARE = P[ORD] * P[ORD];
        MULT[ORD - 1] = J;
      }
      N = 2;
      JPRIME = true;
      while (N < ORD && JPRIME) {
        while (MULT[N] < J) {
          MULT[N] = MULT[N] + P[N] + P[N];
          if (MULT[N] == J)
            JPRIME = false;
        }
        N = N + 1;
      }
    } while (!JPRIME);
    K = K + 1;
    P[K] = J;
  }
  {
    PAGENUMBER = 1;
    PAGEOFFSET = 1;
    while (PAGEOFFSET <= M) {
      cout << "The First " << M <<
```

Abbildung A.6.: Code-Beispiel 2 Teil 1

```
57                " Prime Numbers --- Page " << PAGENUMBER;
58        cout << "";
59        for (ROWOFFSET = PAGEOFFSET; ROWOFFSET < PAGEOFFSET + RR; ROWOFFSET++
60          for (C = 0; C < CC;C++) {
61            if (ROWOFFSET + C * RR <= M) {
62              cout << setw(2) << P[ROWOFFSET + C * RR];
63              cout << " ";
64            }
65          }
66          PAGENUMBER = PAGENUMBER + 1;
67          PAGEOFFSET = PAGEOFFSET + RR * CC;
68        }
69      }
70    }
71    return 0;
72  }
```

Abbildung A.7.: Code-Beispiel 2 Teil 2

```
class CodeAnalyzer {
  private:
    int lineCount;
    int maxLineWidth;
    int widestLineNumber;
    LineWidthHistogram lineWidthHistogram;
    int totalChars;

    static void findJavaFiles(File& parentDirectory, list<File>& files);
    void measureLine(string line);
    void recordWidestLine(int lineSize);
    int lineCountForWidth(int width);
    vector<Integer> getSortedWidths();
  public:
    CodeAnalyzer();
    ~CodeAnalyzer();
    static list<File> findJavaFiles(File& parentDirectory);
    void analyzeFile(File& javaFile) throw();
    int getLineCount();
    int getMaxLineWidth();
    int getWidestLineNumber();
    LineWidthHistogram getLineWidthHistogram();
    double getMeanLineWidth();
    int getMedianLineWidth();
};

CodeAnalyzer::CodeAnalyzer() {
  LineWidthHistogram lineWidthHistogram = LineWidthHistogram();
}

CodeAnalyzer::~CodeAnalyzer() {
  delete lineWidthHistogram;
}

list<File> CodeAnalyzer::findJavaFiles(File& parentDirectory) {
  list<File> files;
  findJavaFiles(parentDirectory, files);
  return files;
}

void CodeAnalyzer::findJavaFiles(File& parentDirectory, list<File>& files)
  for (File file : parentDirectory.listFiles()) {
    if (file.getName().ends_with(".java"))
      files.insert(files.begin(), file);
    else if (file.isDirectory())
      findJavaFiles(file, files);
  }
}

void CodeAnalyzer::analyzeFile(File& javaFile) throw() {
  FileReader fr = FileReader(javaFile);
  BufferedReader br = BufferedReader(fr);
  string line;
  while (!(line = br.readLine()).empty()) {
    measureLine(line);
  }
```

Abbildung A.8.: Code-Beispiel 3 Teil 1

```
57  }
58
59  void CodeAnalyzer::measureLine(string line) {
60    lineCount++;
61    int lineSize = line.length();
62    totalChars += lineSize;
63    lineWidthHistogram.addLine(lineSize, lineCount);
64    recordWidestLine(lineSize);
65  }
66
67  void CodeAnalyzer::recordWidestLine(int lineSize) {
68    if (lineSize > maxLineWidth) {
69      maxLineWidth = lineSize;
70      widestLineNumber = lineCount;
71    }
72  }
73
74  int CodeAnalyzer::getLineCount() {
75    return lineCount;
76  }
77
78  int CodeAnalyzer::getMaxLineWidth() {
79    return maxLineWidth;
80  }
81
82  int CodeAnalyzer::getWidestLineNumber() {
83    return widestLineNumber;
84  }
85
86  LineWidthHistogram CodeAnalyzer::getLineWidthHistogram() {
87    return lineWidthHistogram;
88  }
89
90  double CodeAnalyzer::getMeanLineWidth() {
91    return (double)totalChars/lineCount;
92  }
93
94  int CodeAnalyzer::getMedianLineWidth() {
95    vector<Integer> sortedWidths = getSortedWidths();
96    int cumulativeLineCount = 0;
97    for(int width : sortedWidths) {
98      cumulativeLineCount += lineCountForWidth(width);
99      if (cumulativeLineCount > lineCount/2)
100       return width;
101   }
102   throw Error("Cannot get here");
103 }
104
105 int CodeAnalyzer::lineCountForWidth(int width) {
106     return lineWidthHistogram.getLinesforWidth(width).size();
107 }
108
109 vector<Integer> CodeAnalyzer::getSortedWidths() {
110     vector<Integer> widths = lineWidthHistogram.getWidths();
111     vector<Integer> sortedWidths = widths;
112     sort(sortedWidths, sortedWidths + widths.size());
113     return sortedWidths;
114 }
```

Abbildung A.9.: Code-Beispiel 3 Teil 2

```
static string testableHtml(
  PageData pageData,
  bool includeSuiteSetup
) {
  WikiPage *wikiPage = pageData.getWikiPage();
  StringBuffer *buffer = new StringBuffer();
  if (pageData.hasAttribute("Test")) {
    if (includeSuiteSetup) {
      WikiPage *suiteSetup =
        PageCrawlerImpl::getInheritedPage(
            SuiteResponder::SUITE_SETUP_NAME, *wikiPage
        );
      if (suiteSetup != NULL) {
        WikiPagePath *pagePath =
          suiteSetup.getPageCrawler().getFullPath(suiteSetup);
        string pagePathName = PathParser.render(pagePath);
        buffer.append("!include -setup .")
             .append(pagePathName)
             .append("\n");
        }
      }
      WikiPage *setup =
        PageCrawlerImpl.getInheritedPage("SetUp", wikiPage);
      if (setup != NULL) {
        WikiPagePath *setupPath =
          wikiPAge.getPageCrawler().getFullPath(setup);
        string setupPathName = PAthPArser.render(setupPath);
        buffer -> append("!include -setup .")
             -> append(setupPathName)
             -> append("\n");
      }
  }
  buffer.append(pageData.getContent());
  if (pageData.hasAttribute("Test")) {
    WikiPage *teardown =
      PageCrawlerImpl.getInheritedPage("TearDown", wikiPage);
    if (teardown != NULL) {
        WikipagePath tearDownPath =
          wikiPage.getPageCrawler().getFullPath(teardown);
        string tearDownPathName = PathParser.render(tearDownPath);
        buffer -> append("\n")
             -> append("!include -teardown .")
             -> append(tearDownPathName)
             -> append("\n");
    }
    if (includeSuiteSetup) {
      WikiPage *suiteTeardown =
        PageCrawlerImpl.getInheritedPage(
          SuiteResponder.SUITE_TEARDOWN_NAME,
          wikiPage
      );
    if (suiteTeardown != NULL) {
      WikiPagePath *pagePath =
        suiteTeardown.getPageCrawler().getFullPath(suiteTearDown);
        buffer -> append("!include -teardown .")
             -> append(pagePathName)
```

Abbildung A.10.: Code-Beispiel 4 Teil 1

```
57              -> append("\n");
58
59        }
60      }
61    }
62    pageData.setContent(buffer.toString());
63    return pageData.getHtml();
64  }
```

Abbildung A.11.: Code-Beispiel 4 Teil 2

```
1  class BoldWidget : public ParentWidget {
2    public:
3      static const string REGEXP;
4      static const Pattern& pattern;
5      ParentWidget* parent;
6      BoldWidget(ParentWidget& parent, string text);
7      string render();
8  };
9  const Pattern& BoldWidget::pattern = Pattern::compile("'''(.+?)'''",
10       Pattern::MULTILINE + Pattern::DOTALL
11 );
12 const string BoldWidget::REGEXP = "'''.+?'''";
13 BoldWidget::BoldWidget(ParentWidget& parent, string text) {
14   Matcher *match = pattern.matcher(text);
15   match -> find();
16   addChildWidgets(match -> group(1));
17 }
18 string BoldWidget::render() {
19   StringBuffer *html = new StringBuffer("<br>");
20   html -> append(childHtml()).append("<br>");
21   return html -> toString();
22 }
```

Abbildung A.12.: Code-Beispiel 5

A.4. Regressionsergebnisse

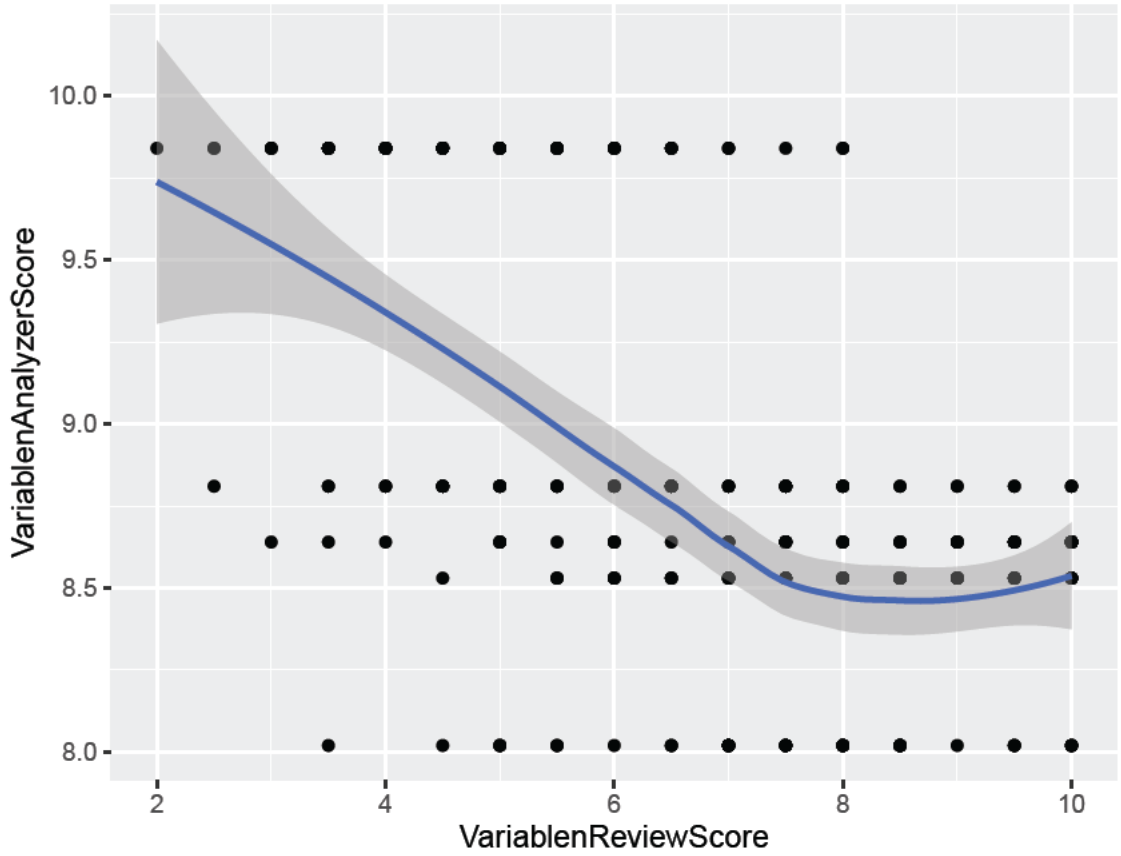

Abbildung A.13.: Streudiagramm Modell Variablen

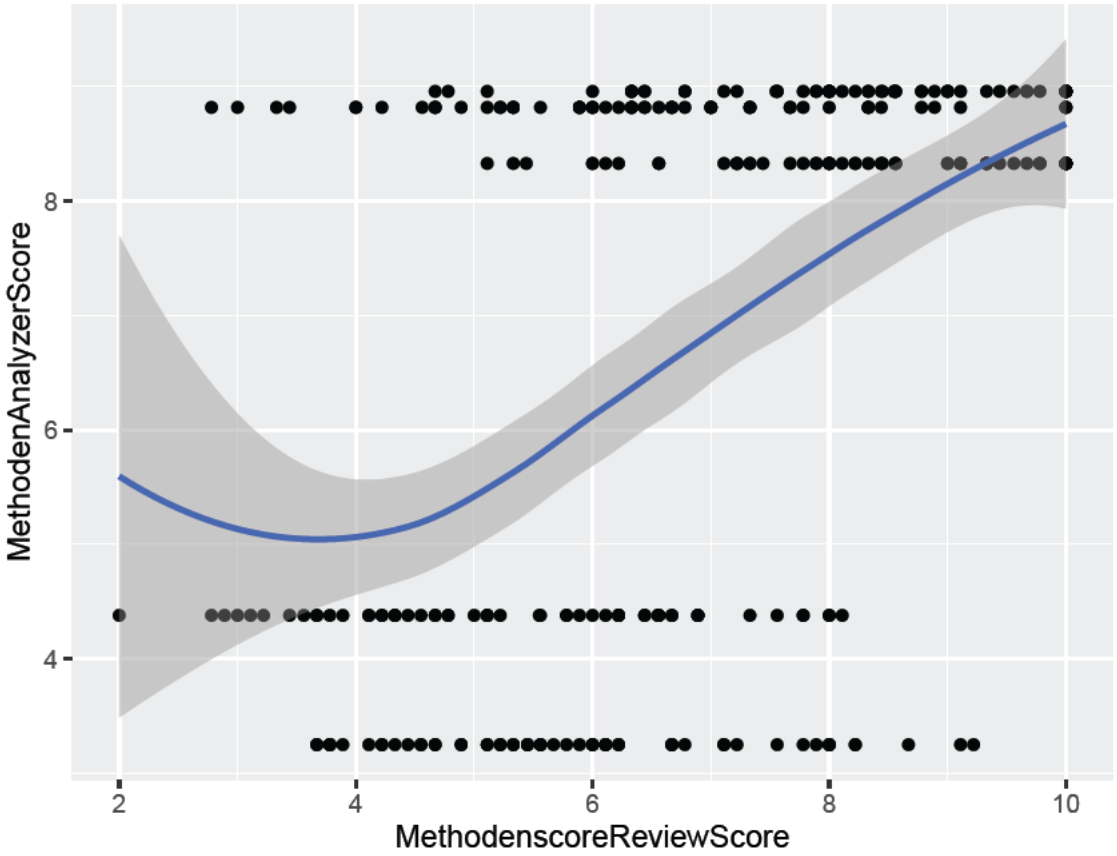

Abbildung A.14.: Streudiagramm Modell Methoden

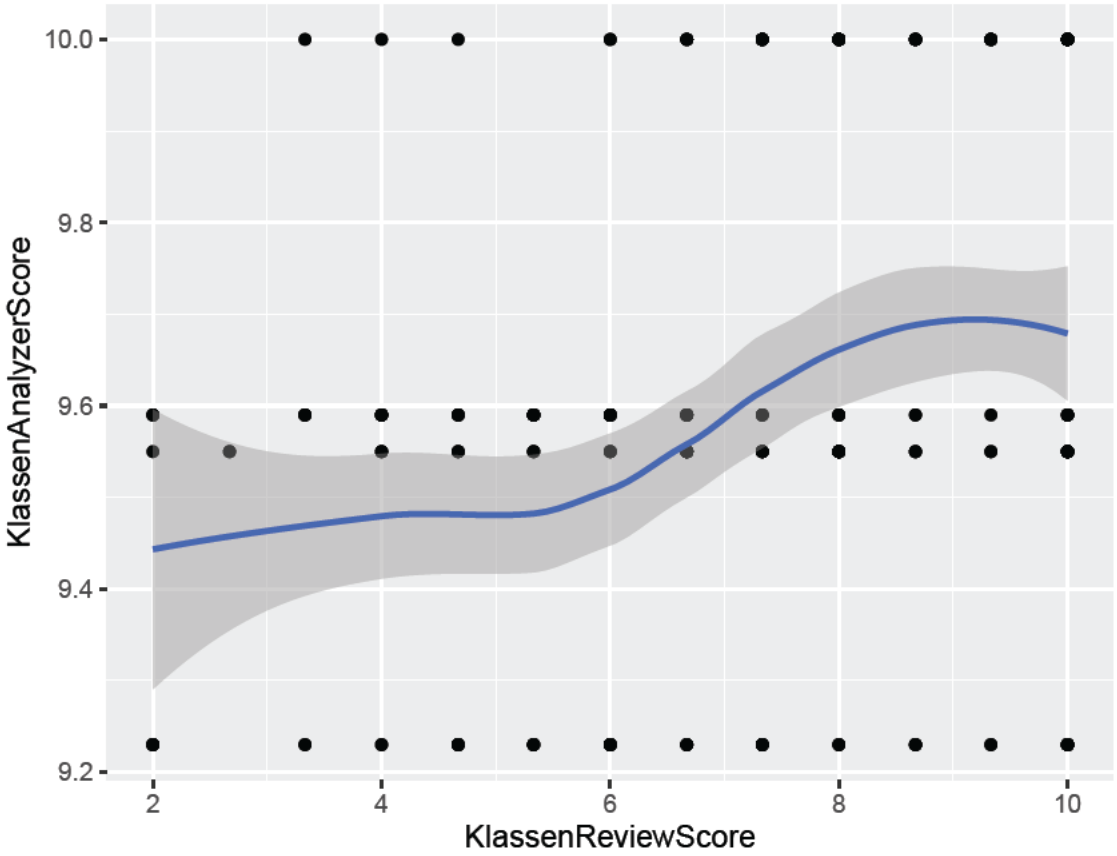

Abbildung A.15.: Streudiagramm Modell Klassen

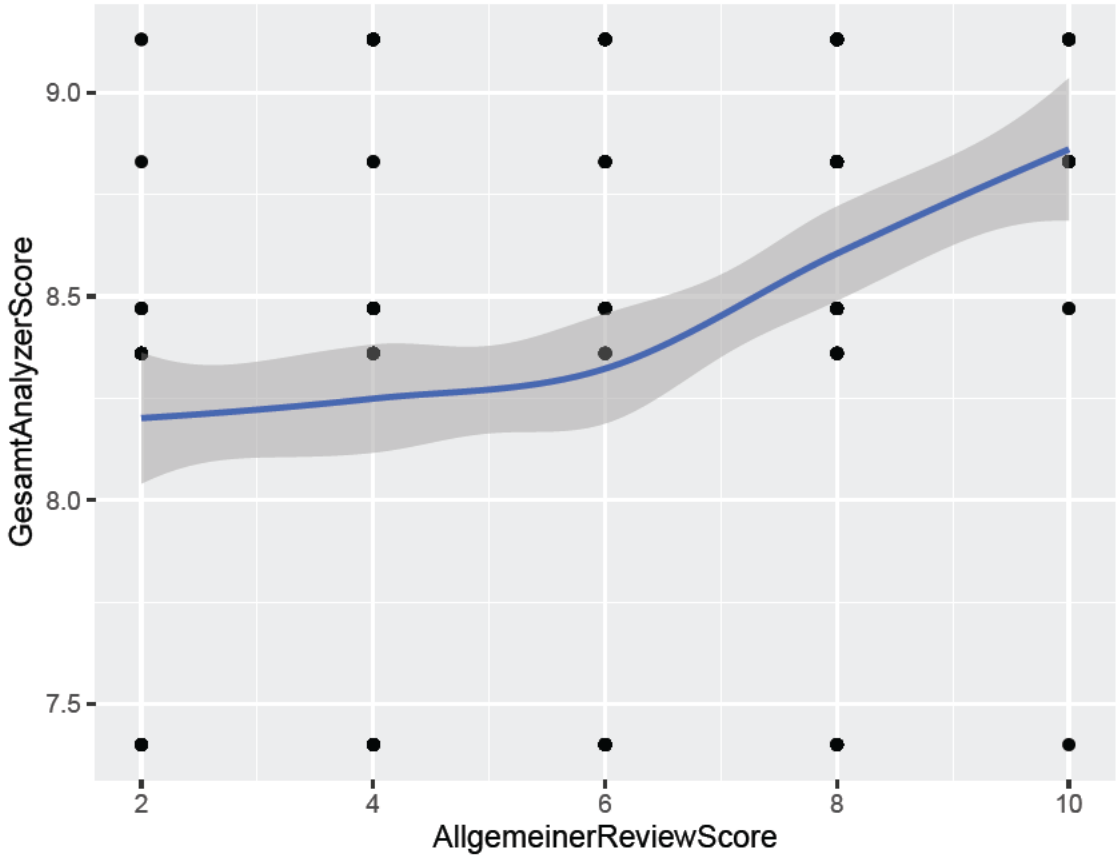

Abbildung A.16.: Streudiagramm Modell Allgemein/Global

B. CD

Der Quellcode des Tools für diese Abschlussarbeit wurde auf der Seite des VSC' GitLab abglegt zum Download.
Die Quelldateien sind unter dem folgendem Link zu finden:
`https://gitlab.hrz.tu-chemnitz.de/mauei--tu-chemnitz.de/cppcompiler`
Auf der CD sind die folgenden Dateien in der hier zu sehenden Struktur zu finden:
—-CD
———01-DigitaleAbschlussarbeit
———02-QuelltextTool
———03-AnleitungTool
———04-FragebogenStudie
———05-RohdatenStudie+Tool
———06-Replikationsfile

C. Selbstständigkeitserklärung

TECHNISCHE UNIVERSITÄT CHEMNITZ

Zentrales Prüfungsamt
Selbstständigkeitserklärung

Name: Vorname: geb. am: Matr.-Nr.:	**Bitte beachten:** 1. Bitte binden Sie dieses Blatt am Ende Ihrer Arbeit ein.

Selbstständigkeitserklärung*

Ich erkläre gegenüber der Technischen Universität Chemnitz, dass ich die vorliegende selbstständig und ohne Benutzung anderer als der angegebenen Quellen und Hilfsmittel angefertigt habe.

Die vorliegende Arbeit ist frei von Plagiaten. Alle Ausführungen, die wörtlich oder inhaltlich aus anderen Schriften entnommen sind, habe ich als solche kenntlich gemacht.

Diese Arbeit wurde in gleicher oder ähnlicher Form noch nicht als Prüfungsleistung eingereicht und ist auch noch nicht veröffentlicht.

Datum: ..... Unterschrift: .....

* Statement of Authorship

I hereby certify to the Technische Universität Chemnitz that this thesis is all my own work and uses no external material other than that acknowledged in the text.

This work contains no plagiarism and all sentences or passages directly quoted from other people's work or including content derived from such work have been specifically credited to the authors and sources.

This paper has neither been submitted in the same or a similar form to any other examiner nor for the award of any other degree, nor has it previously been published.