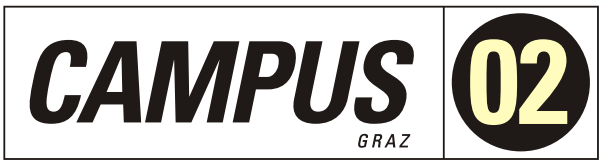
# Masterarbeit

# Senkung von Wartungs- und Folgeaufwänden von Software durch konstruktive Qualitätsgestaltung

ausgeführt am




Studiengang
IT & Wirtschaftsinformatik

Von: Jakob Aumüller
Pers. Kennz. 1610320025

Graz, am 13. Dezember 2017

.....
Jakob Aumüller

# Ehrenwörtliche Erklärung

Ich erkläre ehrenwörtlich, dass ich die vorliegende Arbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen nicht benützt und die benutzten Quellen wörtlich zitiert sowie inhaltlich entnommene Stellen als solche kenntlich gemacht habe.

.....
Jakob Aumüller

# Danksagung

An dieser Stelle möchte ich mich bei allen bedanken, die mich beim Verfassen dieser Arbeit und im bisherigen Studienverlauf aktiv oder auch passiv unterstützt haben.

Meinen größten Dank möchte ich meinem kürzlich verstorbenen Vater, Anton Aumüller, zum Ausdruck bringen und ihm diese Arbeit widmen. Er war mir und meiner Familie der beste, liebste und hingebungsvollste Mensch, den wir uns erträumen konnten. Seine Liebe und Fürsorge hat mich stets durch mein Leben und auch durch dieses Studium getragen und geleitet, sodass Vieles einfach erschien.

In besonderer Weise möchte ich mich auch bei meinem ausgezeichneten Betreuer, Dipl.-Ing. Dr. Johannes Pusterhofer, für die wertvolle Unterstützung bedanken. Er war mir als Ansprechpartner und wunderbarer Gesprächspartner immer eine große Hilfe und hat mit seiner umfassenden Expertise und guten Impulsen zu spannenden Diskussionen beigetragen, was der Arbeit sehr zugute kam.

Ein großer Dank geht an meine Mutter und meine Geschwister sowie deren Familien, welche auch in zuletzt schwierigen, unruhigen Zeiten mich stets unterstützt und mir Mut zugesprochen haben.

Weiters möchte ich mich bei meinen Vorgesetzten und Kollegen bedanken, welche mir diese Zeit des Studiums ermöglicht und mich immer in dieser Entscheidung bestärkt haben. Besonders meine Vorgesetzten haben gerade auch in schwierigen Phasen stets mit Rücksicht auf meine Zusatzbelastung durch das Studium gehandelt und dafür bin ich sehr dankbar.

Nicht zuletzt möchte ich mich bei allen Experiment-Teilnehmern, Interviewpartnern und Korrekturlesern bedanken, welche durch das Geschenk ihrer wertvollen Zeit einen großen Teil zu dieser Arbeit beigetragen haben.

Jakob Aumüller

Graz, am 13. Dezember 2017

# Kurzfassung

Codequalität sowie deren Sicherung und Überprüfung sind wichtige Aspekte heutiger Softwareentwicklungsprozesse und werden von zahlreichen Experten aus Literatur und Praxis als essentieller Faktor für ein langfristiges Bestehen von Softwareprojekten angesehen. Um diese Theorie in der Praxis deduktiv zu prüfen, war es Ziel dieser Arbeit, tatsächliche Auswirkungen ausgewählter Maßnahmen zur Steigerung von Codequalität auf den Wartungs- und Folgeaufwand von Software zu untersuchen. Hierfür wurden zuerst wesentliche Aspekte von Software- und Codequalität explorativ, argumentativ-deduktiv analysiert. Als Ergebnis dieser Analyse wurden grundlegende Merkmale und Kriterien von Softwarequalität sowie Prinzipien, Standards, Methoden und Muster für eine strukturierte Verbesserung von Codequalität vorgestellt. Weiters wurden Metriken zur Messung und Bewertung von Sourcecode sowie Werkzeuge, welche Metriken und andere Aspekte von Codequalität messen, skizziert. Anschließend wurden im Zuge eines Experiments vier funktional idente Programmversionen unterschiedlicher Codequalität Entwicklern zur Bearbeitung vorgelegt und Ergebnisse sowie die benötigte Zeit erhoben und ausgewertet. Hierbei wurde allen Teilnehmern dieselbe Aufgabenstellung gegeben, welche das Beheben von Bugs und das Implementieren einer Erweiterung vorsah. Zudem wurden die Ergebnisse in Expertengesprächen auf Gültigkeit und Plausibilität geprüft. Das Ergebnis konnte – entgegen angenommener Erwartungen – keine eindeutigen Auswirkungen belegen, aber diese auch nicht ausschließen, da die erhaltenen Ergebnisse unter Berücksichtigung üblicher Streuung für alle Programmversionen auf einem ähnlichen Zeitniveau waren. Lediglich in der Qualität der Resultate gab es einige Unterschiede. Anschließende Expertengespräche und die Diskussion der Ergebnisse kamen zu dem Schluss, dass die beobachteten Resultate für die Größe der Untersuchung durchaus plausibel erscheinen, jedoch für beobachtbare Auswirkungen eine deutlich größere Untersuchung notwendig wäre.

# Abstract

Assurance and validation of code quality are important aspects of today's software development process and are considered by many experts in literature and practice as an essential factor for the long-term existence of software projects. In order to deductively test this theory in practice, the aim of this paper was to examine the actual impact of selected code quality enhancement methods on the maintenance and follow-up expenditure of software. For this purpose, essential aspects of software and code quality were first analyzed by investigative and argumentative deduction. As a result of this analysis, basic characteristics and criteria of software quality as well as principles, standards, methods and patterns for a structured improvement of code quality were presented. It also outlined metrics for measuring and evaluating source code, as well as tools that measure metrics and other aspects of code quality. Subsequently, in the course of an experiment, four functionally identical program versions of different code quality were presented to developers for editing and applicable results as well as the required time were collected and evaluated. All participants were given the same scope of tasks, which was to fix bugs and implement an enhancement. In addition, the results were checked for validity and plausibility in expert discussions. Contrary to expectations, the result did not conclusively verify an impact but also cannot negate any effect, since the results obtained were at a similar time level for all program versions, taking into account the usual spread. Only in the quality of the results, there could be observed some differences. Subsequent discussions with experts and the discussion of the results concluded that the observed results seem quite plausible for the size of the study, but that a much larger study would be necessary for observable effects.

# Inhaltsverzeichnis

# 1. Einleitung

## 1.1. Motivation und Aufgabenstellung

Softwarequalität ist von essentieller Wichtigkeit in der Entwicklung von Softwareprojekten. Sie hat Auswirkungen auf nahezu jeden Aspekt eines Softwareproduktes wie beispielsweise Wartbarkeit, Zuverlässigkeit, Funktionalität oder Sicherheit. Jede Entwicklerin und jeder Entwickler sind tagtäglich mit der Wichtigkeit ebendieser Aspekte wie beispielsweise Lesbarkeit, Verständlichkeit und Wartbarkeit von Code konfrontiert, jedoch fehlt in vielen Softwareprojekten ein umfassendes Verständis beziehungsweise der Wille, diesem Thema mehr Bedeutung zukommen zu lassen. Zudem veranlassen hohe Prüf- und Bebehungskosten viele Unternehmen dazu, den Qualitätserfüllungsgrad von Qualitätsmanagement (QM)-Maßnahmen einzuschränken. Aus Kostengründen werden deshalb stets Kompromisse bei der Umsetzung von QM-Maßnahmen eingegangen. In der Softwareentwicklung kann diesem Umstand durch den Einsatz präventiver, statischer Prüfmethoden und einem umfassenden Einsatz von Best Practices aus der Softwareentwicklung weitgehend entgegengewirkt werden. Hierzu gehören beispielsweise Pair-Programming, statische Codeanalyse oder die formale Programmverifikation.

Aus dieser Motivation heraus entwickelte sich die Aufgabenstellung dieser Arbeit, welche darin besteht, sich umfassend mit dem Einsatz von qualitätsfördernden Maßnahmen und Best Practices aus der Softwareentwicklung zu befassen und diese in der Paxis auf Relevanz zu validieren.

## 1.2. Zielsetzung der Arbeit

Wie bereits in Abschnitt 1.1 angedeutet, fehlt in vielen Softwareprojekten das nötige Wissen und vor allem das Verständnis für die Wichtigkeit von Software- und Codequalität. Aus diesem Umstand folgernd ist es Ziel dieser Arbeit, eine mögliche Abhängigkeit von Produktivität in der Softwareentwicklung und Qualität von Sourcecode darzustellen und dafür folgende Forschungsfrage F1 zu beantworten: „Welchen Einfluss haben ausgewählte Maßnahmen zur Steigerung von Codequalität so-

wie der Einsatz statischer Prüfmethoden in der Softwareentwicklung auf die Reduktion des Wartungs- und Folgeaufwands von Software?“ Um diese Frage beantworten zu können wird bestehende Literatur zu den Themen Software- bzw. Codequalität analysiert und reflektiert. Weiters werden Methoden, Richtlinien, Prinzipien und Best Practices aus der Softwareentwicklung für eine gezielte Messung und Verbesserung von Codequalität erhoben und präsentiert. Schließlich werden ausgewählte Verbesserungsmethoden in der Praxis auf deren Wirksamkeit geprüft.

Daraus folgernd sind die Ziele dieser Arbeit, bestehende Vorschläge aus Literatur qualitativ mit Zahlen zu belegen, um eine Tendenz der in der Forschungsfrage gestellten These herauszuarbeiten und punktuelle Erkenntnisse zu bieten.

Es ist ausdrücklich nicht Ziel dieser Arbeit, eine vollständige Analyse aller qualitätsbezogenen Verbesserungsmaßnahmen von Software zu erheben und diese in der Praxis zu validieren. Zudem ist es nicht das Ziel, Ergebnisse mit statistischer Relevanz zu belegen, da dies den Umfang dieser Arbeit aufgrund der hohen Anzahl benötigter Studienteilnehmer deutlich sprengen würde.

Aufbauend auf der definierten Forschungsfrage F1 wurde folgende These gebildet, welche es mit detaillierten Arbeitshypothesen zu belegen gilt. „Der Einsatz ausgewählter Maßnahmen zur Qualitätssteigerung in der Softwareentwicklung hat Auswirkungen auf den Wartungs- und Folgeaufwand von Software.“

Aus dieser These wurde folgende Arbeitshypothese – sowie zugehörige Nullhypothese – definiert, welche später in Kapitel 6 durch weitere Subhypothesen präzisiert wird.

Arbeitshypothese H1: Je umfangreicher der Einsatz ausgewählter Maßnahmen zur Qualitätssteigerung in der Softwareentwicklung erfolgt, desto mehr kann eine Reduktion von Wartungs- und Folgeaufwänden beobachtet werden.

Nullhypothese $H1_0$: Der Einsatz ausgewählter Maßnahmen zur Qualitätssteigerung in der Softwareentwicklung bewirkt nur eine marginale Reduktion von Wartungs- und Folgeaufwänden.

Zur detaillierten Begriffsabgrenzung der Hypothesen wurden folgende Teilbegriffe wie folgt definiert:

- *ausgewählte Maßnahmen zur Qualitätssteigerung*: Im Zuge dieser Arbeit werden verschiedene Ansätze und Maßnahmen für eine Verbesserung der Softwarequalität vorgestellt. Im folgenden Experiment werden dann ausgewählte Maßnahmen, welche nach einer definierten Kriterienliste herangezogen wurden, in der Praxis evaluiert.

- *Reduktion von Wartungs- und Folgeaufwand*: Für eine Reduktion der Aufwände sind in dieser Arbeit Personenstunden (PS) für anfallende Wartungen und Folgeimplementierungen definiert.

- *marginale Reduktion*: Bezeichnet eine nicht relevante, sehr geringe Reduktion der Personenstunden.

## 1.3. Vorgehen und Methodik

Zur gezielten Bearbeitung und Beantwortung des in Abschnitt 1.2 genannten Forschungsthemas werden die Methoden einer argumentativ-deduktiven Analyse und ein darauffolgendes Experiment herangezogen.

### 1.3.1. Argumentativ-deduktive Analyse

In einer explorativen, argumentativ-deduktiven Analyse werden Auswirkungen von statischen Software-Prüfmethoden, Metriken sowie Best Practices der Softwareentwicklung auf Softwarequalität und den damit verbundenen Wartungs- und Folgeaufwänden basierend auf aktuellen Beiträgen aus Literatur und Forschung erarbeitet, um eine theoretische Basis für bestehende Erkenntnisse auf diesem Gebiet zu erhalten. Das Ergebnis dieser Analyse soll Theorien und aktuelle Erkenntnisse in dem Gebiet der Softwarequalität liefern und so die wissenschaftliche Basis für die weitere geplante Forschungsmethodik im Rahmen der Masterarbeit bilden. Die gewonnenen Erkenntnisse sollen anschließend in einer praktischen Umsetzung in einem Experiment unter definierten Rahmenbedingungen auf deren Relevanz überprüft werden.

### 1.3.2. Experiment

Im Rahmen eines Experiments sollen gewonnene Erkenntnisse aus der deduktiven Analyse unter definierten Rahmenbedingungen umgesetzt und auf Relevanz im praktischen Einsatz bewertet werden. Dazu sollen Auswirkungen einer steigenden Konformität von Softwarequalität auf die tatsächlichen und erwarteten Wartungs- und Folgeaufwände in einem Experiment mit mehreren Probanden aus dem Softwareentwicklungsumfeld evaluiert werden.

Hierfür werden zuerst qualitative Ergebnisse erhoben, indem den Probanden unterschiedlich qualitative Teile von Quellcode unter Angabe dezidierter Aufgaben (Ausbessern von Fehlern, Implementierung neuer Features) vorgelegt werden. Die unterschiedlichen Codefragmente bestehen einerseits aus qualitativ sehr hochwertigem

Code und andererseits aus demselben Code, jedoch unter Einsatz ausgewählter Codeverschlechterungen. Weiters werden quantitative Ergebnisse in einer Bewertung der einzelnen Codeversionen anhand der beschriebenen Analyse-Werkzeuge erhoben, um die qualitativen Ergebnisse des Experiments zu unterstützen. Weiters werden Einschätzungen zu den Ergebnissen von Experten aus dem Softwareentwicklungsumfeld eingeholt, um die erhobenen Ergebnisse zu validieren.

### 1.3.3. Experteninterview

Anschließend an das durchgeführte Experiment sollen die erhaltenen Ergebnisse durch offene Gespräche mit Experten aus dem betrachteten Umfeld auf deren Validität und Plausibilität geprüft werden. Hierfür werden zwei bis drei Experten nach einer klar definierten Kriterienliste ausgewählt und offene Interviews nach Vorlage von Mayring (2003) definierter Methodik geführt.

## 1.4. Rahmenbedingungen

Alle gezeigten und verwendeten Beispiele und Code-Listings in dieser Arbeit beziehen sich auf die Programmiersprache Java. Der Grund hierfür ist, dass Java eine sehr weit verbreitete Programmiersprache ist, und in diesem Bereich eine Vielzahl an Werkzeugen und Frameworks existiert.

Es wurde für eine verbesserte Verständlichkeit besonders nicht fach-versierter Leser davon abgesehen, unnötig komplexen Sourcecode in den verwendeten Beispielen zu verwenden, um so die essentiellen Aspekte der Arbeit besser hervorheben zu können.

Weiters wurde auf eine Durchmischung verschiedener Programmiersprachen verzichtet, um die Vorbereitung und Durchführung sowie Evaluierung der Arbeit nicht unnötig kompliziert zu gestalten. Dennoch soll hier erwähnt werden, dass die beschriebenen Methoden, Best Practices, Codemetriken sowie Werkzeuge nicht nur für die Programmiersprache Java eingesetzt werden können, weil diese meist nur geringe Abweichungen im Einsatz aufweisen.

## 1.5. Aufbau der Arbeit

Als Einführung in das Thema Softwarequalität werden im zweiten Kapitel die Grundlagen sowie wichtige Begriffe im Zusammenhang mit Softwarequalität sowie Code-

qualität charakterisiert, um ein grundlegendes Verständnis der Materie zu gewährleisten.

Im dritten Kapitel werden verschiedene wichtige Aspekte und Maßnahmen zur Verbesserung von Codequalität dargestellt. Hierzu gehören diverse Prinzipien, Konventionen, Methoden zu Ausgestaltung und Architektur sowie Best Practices und Anti-Muster, welche in ihrer Gesamtheit dazu beitragen, die Gestaltung und den Aufbau von wartbarem Code zu fördern.

In Kapitel vier wird ein Überblick über die Messung und Analyse von Sourcecode durch Metriken gegeben. Nach einer Übersicht über Kriterien für Metriken werden gängige traditionelle sowie auch objekt-orientierte Metriken dargelegt.

Kapitel fünf befasst sich mit bestehenden Analyse-Werkzeugen im Javaumfeld, welche für eine automatische Ausführung der beschriebenen Metriken sowie zahlreicher anderer Analysen für Codeverbesserung eingesetzt werden.

Im sechsten Kapitel wird ausführlich auf den praktischen Teil der Arbeit, das Experiment, eingegangen. In diesem Kapitel werden alle relevanten Informationen zum Aufbau, sowie zur Vorbereitung und der Durchführung des Experiments ausgeführt.

Kapitel sieben beschreibt alle notwendigen Informationen zur Durchführung des Experiments, sowie die Ergebnisse sowohl der qualitativen Analyse durch die Probanden, als auch der quantitativen der Analyse-Tools, sowie abschließend das Expertenfeedback.

In Kapitel acht werden die gewonnenen Ergebnisse analysiert, die aufgestellten Hypothesen geprüft und mögliche Schlüsse gezogen.

Zuletzt werden in Kapitel neun die Ergebnisse in einer Conclusio zusammengefasst und reflektiert, abschließend ein Ausblick für weitere Forschungsarbeit zu diesem und verwandten Themengebieten gegeben.

# 2. Grundlagen der Softwarequalität

## 2.1. Einleitung

Qualität ist ein für viele Menschen schwer greifbarer, abstrakter Begriff. Qualität ist immer eine subjektive Erfahrung einer Person und weist daher für unterschiedliche Menschen verschiedene Ausprägungen auf. Im Bereich heutiger Computerprogramme und verschiedenartiger Softwareanwendungen hängt der Qualitätsaspekt maßgeblich mit den Personen zusammen – im digitalen Sprachgebrauch meist Benutzer genannt – , welche eine Software verwenden und mit dieser interagieren. Benutzer sind zufrieden, wenn Software exakt das tut, was sie tun soll. Hierbei treten Softwarearchitektur, Codequalität und interne Strukturen in den Hintergrund, da diese Eigenschaften für Kunden eines Softwareproduktes nicht sichtbar sind und daher ebenso wenig interessieren wie beispielsweise der interne Aufbau einer Maschine oder eines elektronischen Gerätes. Wichtig ist einzig und allein, dass die gewünschten Funktionen ordnungsgemäß funktionieren. Dies macht deutlich, dass der Qualitätsbegriff gerade in der Softwarebranche schwer zu operationalisieren ist und daher sehr facettenreiche Auffassungen von Qualität existieren. Wie Liggesmeyer (2009) betont gibt es nicht den Begriff der *besten* Qualität, sondern man sollte immer von *richtiger* Qualität sprechen, da eine Softwarelösung nie alle Qualitätsanforderungen in gleichem Maße erfüllen kann. Das liegt vor allem daran, dass verschiedene Qualitätsaspekte von Software in direktem Konflikt zueinander stehen. Dies wird deutlich, wenn man beispielsweise die beiden Qualitätsanforderungen Portabilität und Effizienz von Software betrachtet. Diese beiden Anforderungen können nicht gleichermaßen optimal erfüllt werden, da gute Portabilität durch die Verwendung allgemeiner Sprachkonstrukte gekennzeichnet ist, wohingegen Effizienz nur durch spezielle Anpassung an eine bestimmte Plattform gesteigert werden kann. (Hoffmann, 2008; Liggesmeyer, 2009)

Um den Begriff Softwarequalität grundlegend definieren und detaillieren zu können, existieren verschiedenste Modelle, welche meist auf dem Modell des ISO/IEC 25010:2011 (2011) Standards basieren. Dieser Standard definiert Softwarequalität wie folgt:

> „Software-Qualität ist die Gesamtheit der Merkmale und Merkmalswerte eines Software-Produkts, die sich auf dessen Eignung beziehen, festgelegte Erfordernisse zu erfüllen.“ (ISO/IEC 25010:2011, 2011)

In dieser Definition wird deutlich, dass Softwarequalität nicht an der Erfüllung bestimmter Kriterien gemessen werden kann, sondern immer eine Gesamtheit verschiedener Merkmale darstellt. (Hoffmann, 2008)

## 2.2. Qualitätsmerkmale und Qualitätskriterien von Software

Wie bereits in Abschnitt 2.1 angeführt, beschreibt Softwarequalität ein Spektrum an verschiedenen Qualitätsmerkmalen und Kriterien. Für Liggesmeyer (2009) umfasst die Norm ISO/IEC 25010:2011 (2011) die Gesamtheit an Anforderungen, welche an eine Software gestellt werden. Dabei unterteilt er diese in drei Sichtweisen, jene der Benutzer, der Entwicklungsteams und der Managerinnen und Manager. Während für Benutzer die externen Merkmale von Software Priorität haben, ist ein Entwicklungsteam in stärkerem Maße daran interessiert, die interne Qualität zu steigern, wohingegen das Management eine Software hinsichtlich interner und externen Kriterien betrachtet und dafür verantwortlich ist, dass der Gesamteindruck einer Software und die Erfüllung aller Anforderungen gewährleistet wird. (Liggesmeyer, 2009)

Nachfolgend werden die Hauptsäulen, welche die Norm ISO/IEC 25010:2011 (2011) definiert, dargestellt und erläutert.

- Funktionalität

  Unter dem Merkmal Funktionalität versteht man den Grad der Erfüllung der geforderten und gewünschten Funktionen einer Software. Funktionale Fehler gibt es in vielfältiger Art und Weise. Diese reichen von Anforderungen an eine Software, welche unklar formuliert sind oder schlicht falsch verstanden werden, über schlichtes Fehlen von Funktionalität aus Zeitgründen oder weil sie technische Probleme verursachen bis hin zu der häufigsten Art von Fehlern – sogenannten Bugs –, welche eine fehlerhafte Implementierung der Algorithmen beschreiben. Letztere können meist durch präventive Maßnahmen in der Qualitätssicherung gefunden und eliminiert werden. (Sneed, Seidl & Baumgartner, 2010)

- Zuverlässigkeit

  Zuverlässigkeit von Software gewinnt in einer fortschreitend digitalisierten Welt stark an Bedeutung. Gerade in sicherheitskritischen Bereichen – wie beispielsweise in Medizin, Mobilität oder Avionik –, wo das kleinste Softwareversagen erhebliche Auswirkungen haben würde, ist die Wichtigkeit dieses Faktors

offensichtlich. Bereits Boehm (1980) hat sich vermehrt mit diesem Thema beschäftigt und versteht Zuverlässigkeit von Software als das möglichst unterbrechungsfreie Laufen eines Programms ohne Fehler sowie eine richtige Funktionsweise und Robustheit bei fehlerhaften oder unvollständigen Dateneingaben. Vereinfacht gesagt soll eine Software nur das tun, was sie tun soll. (Boehm, 1980; Liggesmeyer, 2009)

- Benutzbarkeit

  Benutzbarkeit beschreibt alle Eigenschaften einer Software hinsichtlich der Benutzerinteraktion an der Schnittstelle von Maschine und Mensch. In diesem Bereich hat sich die Softwarebranche in den letzten Jahren immer stärker fokussiert und ausgeprägt, da eine einfache und intuitive Benutzerinteraktion immer wichtiger für den Erfolg einer Software wird. Dennoch muss hier auch differenziert werden, denn Benutzerfreundlichkeit hat nicht für jede Software den selben Stellenwert. Schaut man beispielsweise auf Spezialsoftware in einer bestimmten Domäne, welche von wenigen Experten benutzt wird, so wird hier auf Benutzerfreundlichkeit nicht derselbe Fokus liegen, wie bei einem Softwareprodukt für die breite Masse. (Sneed et al., 2010; Hoffmann, 2008)

- Effizienz

  Eine Software wird als effizient bezeichnet, wenn diese zur Erfüllung ihrer Aufgaben ein Mindestmaß an Ressourcen benötigt. Ressourcen sind in diesem Kontext vordergründig Speicherplatz und CPU-Leistung. Zudem sollte eine Software zu keiner Zeit die Grenzen der vorhandenen Ressourcen überschreiten und ein angemessenes Laufzeitverhalten aufweisen, wobei diese Anforderung meist nicht explizit in Systemspezifikationen vermerkt wird. Besondere Relevanz hat Effizienz im Bereich von Echtzeitsystemen, welche besonderen Anforderungen hinsichtlich Rechenzeit genügen müssen. (Boehm, 1980; Sneed et al., 2010)

- Wartbarkeit

  Wartbarkeit beschreibt im wesentlichen, wie aufwändig es ist, Korrekturen, Änderungen und neue Funktionen in ein bestehendes Softwareprodukt zu integrieren. Diese Eigenschaft wird in realen Softwareprojekten oft vernachlässigt, da die Priorität meist in der schnellen Fertigstellung von Software liegt und dadurch nicht an spätere Wartbarkeit gedacht wird. Jedoch ist gerade diese Eigenschaft ein sehr wichtiges Merkmal von Software, da sie die wichtigste Voraussetzung für eine langlebige Software ist und dadurch den langfristigen Erfolg am Markt sichert. (Hoffmann, 2008)

- Portabilität

Portabilität oder auch Übertragbarkeit von Software beschreibt die Fähigkeit, mit welchem Aufwand eine bestehende Software von einer Umgebung in eine Andere portiert werden kann. Beispiele hierfür wären die Überführung einer Software für eine 32-bit Architektur in eine 64-bit Architektur oder der Wechsel des zugrundeliegenden Betriebssystems. Wird dieses Kriterium bei der Planung und Umsetzung einer Software nicht berücksichtigt, kann dies hohe nachträgliche Kosten verursachen. (Sneed et al., 2010; Hoffmann, 2008)

Die beschriebenen Kriterien werden in der Literatur häufig in externe und interne Merkmale eingeteilt. Während Funktionalität, Benutzbarkeit, Zuverlässigkeit und Effizienz Merkmale sind, welche nach außen sichtbar sind und dadurch direkten Einfluss auf Kunden haben, sind Wartbarkeit, Übertragbarkeit und Testbarkeit für Kunden sehr schwer greifbar. Dennoch sind gerade die inneren Kriterien für den langfristigen Erfolg von Software entscheidend. (Hoffmann, 2008)

Kitchenham und Pfleeger (1996) hinterfragen in ihrer Studie zu dem Thema Qualitätsmodelle der Softwarequalität die Norm ISO/IEC 25010:2011 (2011) kritisch, da diese zwar genaue Vorgaben über die Einteilung von Kategorien und Subkategorien macht, jedoch nicht darauf eingeht, welche Merkmale am besten in welcher Form kombiniert werden können, und zudem die Messung dieser Eigenschaften eher unpräzise erläutert wird. (Kitchenham & Pfleeger, 1996)

Wo die zuvor beschriebenen Qualitätsmerkmale für alle Softwareprodukte gelten, so existieren speziell für den Quellcode einer Software ebenfalls Qualitätskriterien. Guter Quellcode muss diesen Kriterien genügen, um ein Softwareprodukt dauerhaft erfolgreich zu machen und viele der genannten Qualitätsmerkmale einer Software zu erfüllen. Suryanarayana, Samarthyam und Sharma (2015) haben sich mit dieser Thematik auseinandergesetzt und folgende Kriterien erarbeitet, welche guten Code auszeichnen sollten:

- Änderbarkeit: Wie einfach kann bestehender Code geändert werden, ohne negative Seiteneffekte zu verursachen.
- Verständlichkeit: Wie gut kann Code gelesen und verstanden werden.
- Wiederverwendbarkeit: Wie gut können bestehende Codeteile für andere Problemstellungen herangezogen und eingesetzt werden.
- Erweiterbarkeit: Wie einfach kann ein Teil des Codes erweitert werden, um neue Funktionen zu integrieren, ohne dass es zu negativen Seiteneffekten kommt.
- Zuverlässigkeit: In welchem Maß fördert vorhandener Sourcecode die richtige Umsetzung neuer Funktionen und in welchem Maß schützt er vor unterschiedlichen Laufzeitproblemen.

- Testbarkeit: Inwieweit unterstützt Sourcecode das Entdecken von Fehlern durch Tests.

## 2.3. Software-Erosion

Im Gegensatz zu vielen physischen Artefakten altert eine Software nicht im herkömmlichen Sinne, im Prinzip könnte Software ewig so funktionieren wie am ersten Tag. Doch das Problem im Softwarebereich liegt an Änderungen des Hardware- und Softwareumfeldes sowie sich ständig ändernden Anforderungen und Funktionen einer Software, was oftmals dann zu sogenannter Software-Erosion führen kann. Unter Software-Erosion versteht man einen schleichenden Zerfall von Software hinsichtlich der ursprünglichen Qualität. Bandi, Williams und Allen (2013) begründen einen solchen Zerfall durch Verletzung von Softwarearchitekturen, Designregeln, Richtlinien und Code Standards. Die Folgen von Softwareerosion sind meist ein Anstieg von Komplexität innerhalb der Software, schwierigere Integration von Änderungen sowie insgesamt fehleranfälligere Systeme. (Eick, Graves, Karr, Marron & Mockus, 2001; Bandi et al., 2013)

Als mögliche Ursachen für das Entstehen von Software-Erosion definieren Eick et al. (2001) folgende Punkte.

- Eine Architektur, welche fortschreitende Änderungen an der Software nicht unterstützt.
- Missachtung der definierten Design Standards
- Unpräzise definierte und formulierte Anforderungen
- Zeitlicher Druck
- Ungenügende Entwicklungsumgebungen
- Organisationsumfeld, welches negative Auswirkungen auf Arbeitsmoral oder Kommunikation hat.
- Verschiedenartige Begabung innerhalb des Entwicklungsteams
- Fehlendes Änderungsmanagement, wie beispielsweise Versionskontrolle

Für Martin (2003) ist Softwareerosion ein schleichender Prozess, welcher durch die Summierung kleiner Missachtungen am Software Design nach und nach zu einem großen Problem für eine Software werden kann. (Martin, 2003)

## 2.4. Technische Schuld

Technische Schuld beschreibt die angehäuften Altlasten einer Software. Sie ist eine Metapher, welche erstmals durch Cunningham (1992) gebraucht wurde.

> „Technical debt is the debt that accrues when you knowingly or unknowingly make wrong or non-optimal design decisions." (Cunningham, 1992)

Technische Schuld beschreibt dabei eine angehäufte Sammlung von Fehlern und Missständen in der internen Codequalität, welche ein Resultat schlechter Programmiermethoden und unpassender Design-Abläufe ist, und ein hohes Risiko für die Zukunft einer Software darstellt. Meist geht es bei technischer Schuld nicht um reine interne Codequalität, sondern vielmehr um strukturelle und architekturelle Probleme und Missstände sowie veraltete Technologien. (Alzaghoul & Bahsoon, 2013; Quezada Sarmiento, Guaman, Barba Guamán, Quispe & Cabrera, 2017)

Technische Schuld lässt sich anschaulich mit Finanzschuld vergleichen und beschreiben. Zahlt ein Unternehmen die monatlichen Raten fristgerecht, gibt es keine weiteren Schwierigkeiten, werden diese jedoch ausgesetzt, kommt es zu Mahnungen und Strafzahlungen, welche die Gesamtschuld weiter in die Höhe treiben und letztendlich so erdrückend wird, dass eine Insolvenz angemeldet werden muss. In Analogie dazu entsteht in einer Software jedes Mal, wenn das Entwicklungsteam schnelle, ungenaue Veränderungen vornimmt, anstatt sich eine gute Lösung zu überlegen, technische Schuld. Wenn diese Schuld zeitnah beglichen wird, ist alles in Ordnung. Wenn diese jedoch vergessen oder versäumt wird, wird es auf Dauer sehr schwierig, Änderungen an der Software vorzunehmen. Aus diesem Grund ist es von hoher Bedeutung, dass Entwicklerinnen und Entwickler sich der Konsequenzen der täglichen Arbeit bewusst sind und stets die technische Schuld und deren schnelle Tilgung vor Augen haben. (Suryanarayana et al., 2015)

Die Hauptursachen für technische Schuld liegen nach Suryanarayana et al. (2015) in folgenden Punkten.

- Code-Schuld: Code Konventionen und Ergebnisse sowie Empfehlungen von statischen Analyse-Werkzeugen werden missachtet.
- Design-Schuld: Missachten von Design-Richtlinien und Design Smells
- Test-Schuld: Fehlende Tests, unpassendes Test-Design und mangelhafte Testabdeckung
- Dokumentations-Schuld: Fehlende, schlechte oder veraltete Dokumentation

Bei Vorhandensein von technischer Schuld gibt es meist zwei Optionen:

1. Den Zustand einer Software hinsichtlich der gestiegenen Aufwände für zukünftige Änderungen und Wartung beibehalten.

2. Versuchen, die technische Schuld nach und nach zu verringern und für zukünftige Versionen sogenannte Quality-Gates (siehe Abschnitt 5.7) einzuführen, um einen weiteren Anstieg von technischer Schuld zu verhindern.

(Marinescu, 2012)

## 2.5. Zusammenfassung

In diesem Kapitel wurden die wichtigsten Themen und Begriffe aus dem Umfeld der Qualitätsgestaltung von Software aufgezeigt und beschrieben, um einen Einblick in grundlegende Aspekte von Code- und Softwarequalität zu erhalten. Im folgenden Kapitel wird dieser Themenblock weiter vertieft, indem aus der Theorie vorgeschlagene und vielfach eingesetzte Methoden zur Strukturierung und Verbesserung von Programmcode vorgestellt werden.

# 3. Prinzipien und Maßnahmen zur Verbesserung von Codequalität

## 3.1. Einleitung

In der Evolution der Softwareentwicklung entstanden über viele Jahre Prinzipien, Methoden, Richtlinien und Muster, welche eine Hilfestellung zur Erreichung von Software-Qualitätsstandards bieten. Diese bieten geeignete Lösungsmöglichkeiten für wiederkehrende Problemstellungen durch bewährte Ansätze, welche sich in der Praxis etabliert haben. Wichtige Kriterien in puncto Softwarequalität sind Lesbarkeit, Verständlichkeit und Wartbarkeit, welche über einen langen Entwicklungszeitraum stets durch qualitätssichernde Schritte gewährleistet werden sollen. Häufig ist der wichtigste Schritt hierbei, laufend Refactoring im Sinne von stetiger Verbesserung der Codebasis zu betreiben. (Singh & Kahlon, 2011) In diesem Abschnitt werden wichtige Methoden, Prinzipien und Muster für eine erfolgreiche und konstruktive Qualitätsgestaltung angeführt.

## 3.2. SOLID Software-Prinzipien

Als Vorreiter seines Faches beschäftigte sich Martin (2003) in seinem Buch Agile Software Development mit grundlegenden Prinzipien von Software Design und präsentierte darin unter dem Akronym SOLID fünf wichtige Prinzipien, welche als Hilfestellung zur Erreichung von wartbarem und einfach erweiterbarem Sourcecode dienen sollten. Diese Prinzipien besitzen bis heute einen hohen Stellenwert in der Softwareentwicklung – und darüber hinaus – und sollen nachfolgend kurz beschrieben werden. Die wesentlichen Inhalte der nachfolgenden Darlegungen wurden weitgehend aus Martin (2003) entnommen.

### 3.2.1. Single Responsibility Principle

Obwohl Martin (2003) dieses Prinzip erstmals eingeführt hatte, betont der Autor, dass deren Inhalte und Überlegungen größtenteils auf den Arbeiten von Tom DeMarco und Meilir Page-Jones fundieren. Gerade diese beiden Autoren beschäftigten sich in DeMarco (1979) erstmals mit Kohäsion, worauf sich das hier beschriebene Prinzip im Wesentlichen stützt. Das Single responsibility principle (SRP) besagt im Wesentlichen, dass eine Klasse oder ein Modul nicht mehr als genau eine Zuständigkeit besitzen, und diese eine Zuständigkeit durch diese Klasse oder dieses Modul allein gekapselt sein soll. Somit wird sichergestellt, dass eine bestimmte Aufgabe oder Funktionalität in genau einem bestimmtem Codeteil implementiert ist und nirgendwo sonst. Martin (2003) versteht eine Responsibility als einen Grund für Änderung und begründet den Nutzen dieser Technik dadurch, aufgrund der klaren Trennung von Zuständigkeiten Klassen und Module stabiler gegen Fehler machen. Auch wenn dieses Prinzip auf den ersten Blick einfach umzusetzen erscheint, so betont Martin (2003) die teils schwierige Umsetzung in der Praxis, welche viel Erfahrung und Übung voraussetzt. (Martin, 2003; DeMarco, 1979)

### 3.2.2. Open Closed Principle

Das Open/closed principle (OCP) gilt als elementarer Grundsatz von objekt-orientienter Software. Es beschreibt den richtigen Umgang mit der Erweiterung von verschiedenen Codekonstrukten. Im wesentlichen empfiehlt OCP, dass Code (Module, Klassen, Funktionen) offen für Erweiterung, jedoch geschlossen für Modifikation sein sollte. Obwohl dieses Prinzip bereits zuvor von Meyer (1995) beschrieben wurde, so hatte Martin (2003) es erstmals auch im Zusammenhang von Polymorphie eingesetzt. Um die beschriebenen Erweiterungen von Klassen zu erreichen, verwendet man Abstraktion, beispielsweise durch Vererbung von abstrakten Klassen oder Implementierung von Interfaces. Dies dient dem Zweck, neue Funktionalität zu vorhandenen Klassen hinzuzufügen, ohne diese verändern zu müssen. Geschlossenheit für Veränderung wird dadurch erreicht, indem man Klassen nur von fixierten Abstraktionen abhängig macht. So ist es durch neue Implementierungen möglich, zusätzliche Funktionalität zu generieren, ohne den Code vorhandener Konstrukte zu verändern. Im Übrigen definiert Martin (2003) dieses Prinzip als überaus wichtig in objekt-orientiertem Design, da es Software wartbar und flexibel macht. (Meyer, 1995; Martin, 2003)

### 3.2.3. Liskov Substitution Principle

Das Liskov substitution principle (LSP), welches in Liskov und Wing (1999) definiert wurde, beschreibt Regeln für Klassen, welche von einem Supertyp ableiten. Im We-

sentlichen definiert dieses Prinzip, dass Klassen und insbesondere deren Methoden, welche auf einen Typ angewendet werden, ebenso beim Anwenden auf Subtypen richtig funktionieren müssen. Dafür stellten Liskov und Wing (1999) folgende Definition auf:

> „Sei q(x) eine Eigenschaft des Objektes x vom Typ T, dann sollte q(y) für alle Objekte y des Typs S gelten, wobei S ein Subtyp von T ist.“ (Liskov & Wing, 1999)

Veranschaulicht kann dieses Prinzip durch das sogenannte Kreis-Ellipse-Problem werden. Wenn eine Klasse Kreis von der Klasse Ellipse ableiten würde, welche zum Setzen der Halbachsen zwei Methoden anbietet, wäre LSP verletzt, da ein Kreis keine unterschiedlichen Halbachsen setzen kann. Martin (2003) sieht den Einsatz von LSP als essentielle Voraussetzung, um OCP zu ermöglichen, da eine Erweiterung von Klassen ohne die Austauschbarkeit von Subtypen nicht möglich wäre. (Martin, 2003; Liskov & Wing, 1999)

### 3.2.4. Interface Segregation Principle

Interface segregation principle (ISP) definiert Regeln für Schnittstellendefinitionen, sogenannte Interfaces. Kern-Aussage dieses Prinzips ist es, Schnittstellen möglichst klein zu halten, um Klassen nicht zu nicht benötigten Implementierungen zu zwingen. Die Empfehlung von Martin (2003) ist hier, große Schnittstellendefinitionen in kleinere Gruppen mit zusammengehörigem Verhalten zu unterteilen. Er nennt diese Gruppierungen Role-Interfaces, welche von den einzelnen Klassen separat je nach Bedarf verwendet bzw. implementiert werden können. Vorteile durch ISP sind eine bessere Entkoppelung von Klassen und Modulen und damit einhergehend bessere Maintainability. (Martin, 2003)

### 3.2.5. Dependency Inversion Principle

Das Dependency inversion principle (DIP) beschreibt einen Weg zur Entkoppelung von Klassen und ganzen Systemen. Martin (2003) definiert dieses Prinzip wie folgt:

> „High-level modules should not depend on low-level modules. Both should depend on abstraktions. Abstraktions should not depend on details. Details should depend on abstractions.“ Martin (2003)

Oft kommt es vor, dass Klassen höherer Ebenen Klassen niedrigerer Ebenen konsumieren und sich dadurch von diesen Detail-Implementierungen abhängig machen.

Dies erhöht hochgradig die Komplexität von Software. Durch die Verwendung des DIP wird diese Kopplung aufgelöst, indem Klassen höherer Ebene ihre Abhängigkeiten auf Abstraktionen anstatt auf Detail-Implementierungen festlegen. (Martin, 2003)

## 3.3. Coding Standards

Ein wichtiger Faktor beim kollaborativen Erstellen von Software in Teams ist das Definieren und Deklarieren von Regeln zur Gestaltung von Sourcecode. Mittels sogenannter Code-Konventionen werden ebensolche Regeln meist unternehmensweit definiert und dokumentiert, sodass für jeden Softwareentwickler klare Richtlinien und Standards existieren, sodann jeder den Code anderer verstehen und damit arbeiten kann. Wenn Sourcecode nicht nach Code-Richtlinien geschrieben und organisiert ist, wird es zunehmend schwerer für Entwicklungsteams, daran zu arbeiten, neue Inhalte zu integrieren oder Teile der Software zu warten. Besondere Relevanz hat dieser Aspekt bei geographischer Distanz zwischen verschiedenen Entwicklungsteams. (Van Emden & Moonen, 2002; LEE, LEE & IN, 2015)

Code Konventionen sind meist eine Sammlung von Regeln, Richtlinien und Best Practices für die syntaktische und lexikalische Form von Code. Vorrangiges Ziel dieser Standards ist eine Erhöhung von Lesbarkeit, Verständlichkeit und Wartbarkeit sowie eine Verbesserung der Zusammenarbeit. Im Detail beschäftigen sich Code Konventionen meist mit den Themen Einrückungen, Kommentare, Dateiorganisation, Namenskonventionen und Programmierpraktiken. (Smit, Gergel, Hoover & Stroulia, 2011)

Auch wenn Code Konventionen in der Theorie viele Vorteile bergen, so werden sie in der Realität meist als erstes vernachlässigt, wenn der erste Zeitdruck in einem Projekt aufkommt. Um diesem Problem zu begegnen, existieren automatisierte Werkzeuge – im Java-Umfeld beispielsweise das Tool Checkstyle (*Checkstyle Project Website*, o. J.) – und Methoden aus Entwicklungsvorgehensmodellen wie beispielsweise Code Reviews und Pair Programming. Wesentlicher Bestandteil dieser beiden Methoden ist das Vier-Augen-Prinzip, womit Flüchtigkeitsfehler, aber auch gravierende Probleme, durch eine weitere Person meist sehr effizient aufgedeckt werden können. (Smit et al., 2011; LEE et al., 2015)

## 3.4. Design Patterns

Design Patterns sind Dokumentationen von festgelegten Lösungsvorschlägen für häufig wiederkehrende Design-Probleme in der Softwareentwicklung. Design Patterns

sind offen zugänglich und bieten bewährte Lösungsschablonen für weitverbreitete Probleme im Entwurf von Softwarestrukturen. (Alur, Malks, Booch, Crupi & Fowler, 2003) Einen gelungenen Definitionsansatz lieferte Fowler (2010) wie folgt: „A pattern is an idea that has been useful in one practical context and will probably be useful in others.“ (Fowler, 2010)

Weite Verbreitung fanden Design Patterns durch Gamma, Helm, Johnson und Vlissides (1994), welche auch unter dem Namen Gang of Four (GoF) bei vielen Entwicklern und Systemarchitekten bekannt sind. Auch wenn sie die Entwurfsmuster nicht primär als erste definiert haben, so sind sie deshalb bekannt, weil sie die gebräuchlichsten Muster in ihrem Werk übersichtlich dokumentiert und publiziert haben. Sie definieren für jedes Pattern vier wesentliche Aspekte:

- Name des Patterns: hat zum Ziel, eine einheitliche Kommunikation und Verwendung zu garantieren.
- Problem: Problembeschreibung sowie Kontextabgrenzung des anzuwendenden Lösungsmusters.
- Lösungsansatz: abstrakte Lösungsbeschreibung.
- Fazit: Darstellung von eventuellen Vor- und Nachteilen sowie Hilfestellung bei der Entscheidung, wann das Muster eingesetzt werden sollte und auch wann auch nicht.

Zudem haben Gamma et al. (1994) Design Patterns in drei Bereiche nach Einsatzgebiet und -Zweck aufgeteilt:

- Creational Design Patterns: Befassen sich mit der Erzeugung von Objekten und haben das Ziel, die Erzeugung eines Objektes von der Objektdeklaration zu trennen und so eine Entkoppelung von den konkreten Implementierungen zu gewährleisten.
- Structural Design Patterns: Ziel ist eine Vereinfachung des Software-Design durch eine einfache Art, Beziehungen zwischen Objekten abzubilden.
- Behavioral Design Patterns: Definieren verbreitete Kommunikations-Muster und modellieren komplexes Verhalten zwischen Objekten. Ziel ist eine Erhöhung der Flexibilität von Software hinsichtlich ihrer Algorithmen.

Um einen Überblick über die gängigen Design Patterns nach Gamma et al. (1994) zu erhalten, werden diese in Tabelle 3.1 nach den genannten Kategorien getrennt dargestellt.

| Creational Patterns | Structural Patterns | Behavioral Patterns |
|---|---|---|
| Abstract Factory | Adapter | Chain of Responsibility |
| Builder | Bridge | Command |
| Factory Method | Composite | Interpreter |
| Prototype | Decorator | Iterator |
| Singleton | Facade | Mediator |
| | Flyweight | Memento |
| | Proxy | Observer |
| | | State |
| | | Strategy |
| | | Template Method |
| | | Visitor |

**Tabelle 3.1.:** Design Patterns (Gamma et al., 1994)

Wie Gamma et al. (1994) betonen, stehen Design Patterns oft in engem Zusammenhang zu den in Abschnitt 3.2 beschriebenen SOLID Prinzipien, da die korrekte Anwendung von Design Patterns oft für die Erfüllung der SOLID Prinzipien notwendig sind. (Gamma et al., 1994)

Aufbauend auf den Design Patterns von Gamma et al. (1994) haben sich viele Frameworks und Erweiterungen entwickelt, welche konkretere Anwendung für bestimmte Programmiersprachen beschreiben, wohingegen die Design Pattern der GoF eher abstrakte Muster sind, welche sich nicht auf eine spezielle Programmiersprache festlegen. Dennoch basieren die meisten heutigen Muster auf ebendiesen Design Patterns der GoF. (Wolfgang, 1994)

## 3.5. Anti-Pattern

Anti-Pattern sind, wie der Name schon vermuten lässt, das genaue Gegenteil von Pattern. Im Gegensatz zu herkömmlichen Pattern, welche erprobte Lösungsansätze und Best Practices zu bekannten Problemen bereitstellen, sind Anti-Pattern Anschauungsbeispiele für weniger gute Praktiken bei der Problemlösung in der Softwareentwicklung. Anti-Pattern existieren in drei Perspektiven, der Architekturperspektive, der Projektperspektive und der Perspektive des Softwareentwicklungsprozesses. Nachfolgend liegt der Schwerpunkt auf Anti-Pattern der Softwareentwicklung. Palomba, de Lucia, Bavota und Oliveto (2015) sehen die Ursachen von Anti-Pattern darin, dass viele Entwicklerinnen und Entwickler meist unter enormem Zeitdruck ein Stück Software fertigstellen müssen, und auf diesem Wege oft unbeabsichtigt Anti-Pattern in den Code implementieren. Brown (1998) ist der Meinung, dass die Ursachen in der meist dürftigen Ausbildung oder den fehlenden Kenntnissen von Entwicklerinnen

und Entwicklern zu finden sind. Zudem betont er, dass häufig falsche Implementierungen von Design Patterns zu Anti-Pattern führen. Aus diesem Grund sah Brown (1998) die Notwendigkeit, für eine Sammlung und Aufbereitung von Anti-Pattern, da es für ihn meist einfacher ist, falsche Muster in Sourcecode zu sehen, als fehlende Design Patterns zu erkennen. (Palomba et al., 2015; Brown, 1998)

In wissenschaftlichen Kreisen wird in diesem Bereich intensiv geforscht, seitdem festgestellt wurde, dass die Existenz von Anti-Pattern negative Folgen für Lesbarkeit und Wartbarkeit von Sourcecode hat. Khomh, Di Penta, Guéhéneuc und Antoniol (2012) fanden beispielsweise heraus, dass Software mit Anti-Pattern eine höhere Fehleranfälligkeit aufweist als Software ohne Anti-Pattern. Yamashita und Moonen (2013) zeigen in ihrer Studie, dass ein Anti-Pattern allein noch nicht die Lesbarkeit verschlechtert, aber eine Kombination aus mehreren Anti-Pattern durchaus diesen Effekt aufweisen kann.

Brown (1998) befasste sich eingehend mit dieser Problematik in seinem Werk und stellt darin eine Sammlung von wichtigen Anti-Pattern vor, wobei er nicht nur Anti-Pattern aufzeigt und erklärt, sondern auch geeignete Lösungsvorschläge gibt. Nachfolgend werden einige wichtige Anti-Pattern aus diesem Werk kurz vorgestellt.

- Copy-and-Paste

  Unter diesem Anti-Pattern versteht man das offensichtliche Kopieren von Sourcecode-Teilen. Oft ist die Motivation von Entwicklerinnen und Entwicklern jene, schneller neue Funktionen zu implementieren, jedoch führt diese Praxis langfristig zu vielen Schwierigkeiten und einer Verschlechterung der Wartbarkeit. Um diesem Problem entgegenzutreten empfehlen sich einerseits Werkzeuge wie PMD, welche Codeduplikate automatisiert erkennen können und andererseits können durch Abstraktionen in Form wiederverwendbarer Klassen-Konstrukte und Templates Duplikate vermieden und die Wartbarkeit enorm verbessert werden. (Brown, 1998; *PMD Project Website*, o. J.) ()

- Spaghetti Code

  Hiermit ist Code gemeint, welcher mit der Zeit immer unstrukturierter und dadurch undurchsichtiger wurde und letztlich von niemandem mehr richtig interpretiert werden kann. Ein Indiz dafür sind Klassen mit wenigen, aber dafür sehr langen Methoden, welche eine hohe Komplexität durch viele Verzweigungen aufweisen. Als Lösung wird von Brown (1998) Code-Cleanup durch fortlaufendes Refactoring empfohlen.

- Lava Flow

  Dieses Anti-Pattern beschreibt Software mit viel totem Code, welcher nicht gelöscht wurde, sondern stattdessen um diesen herum ständig neue Verzweigun-

gen implementiert werden. Brown (1998) beschreibt toten Code als Code, welcher nie verwendet und daher nie ausgeführt wird. Er empfiehlt solche Code-Leichen durch Refactoring aufzudecken und zu löschen.

- Golden Hammer

  Mit Golden Hammer bezeichnet Brown (1998) Methoden und Lösungen, welche von Entwicklerinnen und Entwicklern für unterschiedlichste Problemstellungen angewendet werden, ohne zu differenzieren. Abhilfe können laut Brown (1998) Schulungen und Trainings bieten, wo neue Methoden und Ansätze gelehrt und weitergegeben werden.

- The Blob

  Unter Blob versteht Brown (1998) eine Gott-Klasse, also eine Klasse, welche zu viele Abhängigkeiten und Verantwortlichkeiten besitzt. Dieses Anti-Pattern steht im Widerspruch zu strukturierter Programmierung, welche zum Ziel hat, große Problemstellungen in möglichst kleine, übersichtliche Einheiten aufzuteilen. Für Abhilfe kann hier das Vorgehen nach SRP sorgen, um Klassen mit hoher Kohäsion zu bilden.

- Poltergeists

  Als Geisterklassen werden Klassen mit kurzer Lebensdauer und oft beschränkten Befugnissen beschrieben, welche ihre Anfragen hauptsächlich auf langlebigere Klassen weiterreichen. Als Abhilfe wird geraten, die Verantwortlichkeiten auf langlebigere Klassen zu verteilen und Geisterklassen zu löschen. (Brown, 1998)

- Functional Decomposition

  Functional Decomposition ist ein Anti-Pattern, welches den Einsatz von nicht objekt-orientiertem Sourcecode in einer objekt-orientierten Sprache beschreibt. Meist sind es erfahrene Entwicklerinnen und Entwickler, welche lange Zeit in nicht objekt-orientierten Programmiersprachen entwickelt haben und sich mit einer Umstellung auf eine objekt-orientierte Sprache schwer tun oder diese nicht akzeptieren. Indizien hierfür sind oft Klassen, welche nicht als eigenständige Klassen strukturiert sind, sondern lediglich als Sub-Routinen für andere Methoden dienen. (Brown, 1998)

## 3.6. Pair Programming

Pair Programming ist als wesentliches Element von Extreme Programming eine Methode in der agilen Softwareentwicklung, in welcher anstelle eines einzigen Programmierenden jeweils zwei Programmierende zusammen Software entwickeln. Der Ablauf sieht meist wie folgt aus: Einer der beiden Programmierenden schreibt aktiv neuen Code, Algorithmen oder Tests und begründet bei Bedarf seine Intentionen. Die andere Person beobachtet und analysiert das Geschriebene auf Fehler oder Probleme und kommuniziert Missstände und Verbesserungsvorschläge unmittelbar, um diese direkt zu zweit lösen zu können. Diese Rollen werden mehrmals täglich getauscht, um eine gleichmäßige Auslastung und Verteilung der Programmierenden zu gewährleisten. (Swamidurai, Dennis & Kannan, 2014; J. Cohen, Teleki, Brown & DuRette, 2006)

Ein entscheidender Vorteil dieser Methode ist das permanente Überprüfen und Infragestellen des geschriebenen Code ähnlich einem Code Review, jedoch intensiver. Auf diesem Weg lässt sich die Qualität von Software erheblich verbessern, da jedes Stück Code und jeder Algorithmus nach dem Vieraugenprinzip kritisch hinterfragt und geprüft wird. Zudem ist es mit Pair Programming möglich, einen Wissenstransfer zwischen verschiedenen Entwicklerinnen und Entwicklern besonders in Hinblick auf verschiedene Erfahrungsstufen zu realisieren. Aus diesem Grund ist es ratsam, Teams aus weniger erfahrenen und sehr erfahrenen Entwicklerinnen und Entwicklern zu bilden, um diesen Wissenstransfer bestmöglich zu forcieren. Als Nachteile von Pair Programming werden oft erhöhte Ressourcen und damit hohe Kosten genannt, sowie Probleme und Spannungen in den zwischenmenschlichen Beziehungen aus verschiedenen Gründen. Eine weitere Einschränkung ist die zwingend benötigte Ortsabhängigkeit, da diese Methode nur in physikalischer Nähe erfolgreich eingesetzt werden kann. (D. Cohen, Lindvall & Costa, 2004)

Ein interessantes Ergebnis haben Swamidurai et al. (2014) in einer empirischen Studie ermittelt, indem sie die beiden Techniken Pair Programming und Code Review miteinander verglichen haben und zu der Erkenntnis gekommen sind, dass beide Techniken in etwa dieselbe Qualitätsverbesserung leisten können, jedoch Code Reviews deutlich wirtschaftlicher umzusetzen sind.

## 3.7. Code Reviews

Ein Code Review ist eine Technik aus der agilen Softwareentwicklung mit dem Ziel, bereits geschriebenen Code von dritten Personen zu begutachten, indem fertige Code-Abschnitte auf Fehler und Inkonsistenzen hin überprüft werden, um die Qualität der Software zu verbessern. Eine spezielle Form des Code Review stellt hierbei das Peer

Code Review dar, wo sich die Autorin oder der Autor zusammen mit der prüfenden Person zusammensetzen und die Autorin oder der Autor den betrachteten Codeteil dem Prüfenden erläutert. Hierbei werden Unklarheiten diskutiert und und Verbesserungen vorgeschlagen. (J. Cohen et al., 2006)

Somit tragen Code Reviews zu einer gesteigerten Qualität von Software bei, indem Probleme früh aufgedeckt werden, das Einhalten von Standards und Konventionen gefördert und schlechtes Software-Design vermieden wird. Zusätzlich wird der Wissensaustausch zwischen den Mitarbeitern unterstützt. (Swamidurai et al., 2014)

Peer Code Reviews sind am effektivsten, wenn beide Parteien physisch zusammen sitzen, da auch ein Screen Sharing grundsätzlich möglich ist, jedoch Planungen auf diese Weise umständlicher sind, und Kommunikationsprobleme auftreten können. Nachteile von Code Reviews – insbesondere Peer Code Reviews – sind oft fehlendes Messen von Zielen, da der Erfolg eines Code Review nicht gut gemessen und überprüft werden kann. (J. Cohen et al., 2006)

In der modernen Softwareentwicklung wird zunehmend auf leichtgewichtigere Code Review tool-unterstützte Methoden gesetzt, anstatt Peer Code Reviews in dieser Form zu praktizieren. Mittlerweile existieren viele gute Werkzeuge, welche die Arbeit der Review Organisation um ein Vielfaches erleichtern und Entwicklerinnen und Entwicklern die Möglichkeit bieten, separat in gewünschtem Zeitraum die einzelnen Änderungen einer Prüfung zu unterziehen, über Kommentare zu kommunizieren und Verbesserungen aufzuzeigen. Diese moderne tool-basierte Art des Code Review wird bereits von führenden IT-Unternehmen – wie beispielsweise Microsoft, Google und Facebook – und zahlreichen Open-Source-Projekten verwendet. Bacchelli und Bird (2013) haben in ihrer Studie festgestellt, dass der Fokus dieser neuen Methoden durchaus nicht nur dem Finden von Fehlern gilt, sondern Wissenstransfer, Teambewusstsein und verbesserte Lösungsansätze wichtige Aspekte sind und bleiben. (Bacchelli & Bird, 2013)

## 3.8. Refactoring

Refactoring ist ein entscheidender Bestandteil des Software-Entwicklungsprozesses und essentiell für die Lebensdauer und Wiederverwendbarkeit moderner Software und Frameworks. Dabei reicht Refactoring von vermeintlich einfachen Aufgaben – wie dem Umbenennen von Klassen oder Methoden – bis hin zu tiefgreifenden Änderungen – wie dem nachträglichen Einführen eines Design Patterns in bestehende Softwarestrukturen. (Roberts, Brant & Johnson, 1997)

Fowler und Beck (2013) beschreiben Refactoring als eine Tätigkeit, bei der die interne Struktur eines Softwareabschnitts modifiziert und dadurch verbessert wird, ohne das

Verhalten nach außen zu beeinträchtigen. Ziel dieses Prozesses ist es, das Design bestehender Software durch Bereinigen und Verbessern des Quellcodes zu verbessern und dadurch Lesbarkeit und Wartbarkeit zu erhöhen sowie zukünftige Fehlerpotentiale zu minimieren. Grundsätzlich wird der Begriff in zweierlei Definitionen unterschieden. Während Refactoring als Hauptwort das Vornehmen einer bestimmten Änderung am Code bezeichnet wie beispielsweise die Umbenennung einer Methode, so steht Refactoring als Verb für den allgemeinen Prozess, Software neu zu strukturieren, indem ein bis mehrere Refactoring Methoden angewendet werden. (Fowler & Beck, 2013)

Nach der Meinung von Arcelli, Cortellessa und Trubiani (2015) ist der Prozess des Findens von Problemen in Software und die angemessene Auswahl geeigneter Refactoring Maßnahmen sowohl in kleineren als auch in großen Softwareprojekten eine durchaus herausfordernde, komplexe Aufgabe.

Wichtig ist laut Fowler und Beck (2013) der Aspekt, dass Refactoring Maßnahmen für den Kompiler nicht von Bedeutung sind, sondern einzig allein für die Menschen, welche an dem Quellcode einer Software arbeiten. Ein Kompiler kann sowohl weniger guten Code als auch guten Code gleichermaßen bearbeiten, wohingegen für Menschen Verständlichkeit und Lesbarkeit von Quellcode entscheidend ist, da eine Veränderung oder Erweiterung von Code stets ein vorheriges Verständnis bedingt. Daher ist es wichtig, dass Refactoring kontinuierlich betrieben wird, da ohne Refactoring eine kontinuierliche Verschlechterung der Softwarestruktur und damit erhöhter zukünftiger Wartungsaufwand unvermeidlich ist. Refactoring sollte keine eigenständige, periodisch geplante Aufgabe sein, sondern als Teil der täglichen Entwicklungsarbeit verstanden werden. Eine gute Gelegenheit für Refactoring sind auch Code Reviews (Abschnitt 3.7), da dort oft Ideen für Refactoring Maßnahmen entstehen und zudem weniger erfahrene Entwicklerinnen und Entwickler von den Erfahrenen den richtigen Einsatz von Refactoring lernen können. (Roberts et al., 1997; Fowler & Beck, 2013)

Im Folgenden werden die wichtigsten Empfehlungen für erfolgreiches Refactoring nach Fowler und Beck (2013) angeführt.

- Eingriffe am Sourcecode sollten immer in kleinen Schritten passieren, um potentielle Fehler schnellstmöglich aufzudecken.

- Das Naming von Software Entitäten ist sehr wichtig, da diese die Bedeutung des Code kommunizieren und damit die Verständlichkeit verbessern. Hierfür existieren in den gängigen Entwicklungsumgebungen geeignete Werkzeuge zum Renaming zur Verfügung.

- Vor dem Refactoring sollte für ausreichende Tests der bearbeiteten Funktionalität gesorgt werden, damit sichergestellt werden kann, dass die vorgenommenen

Refactoring Maßnahmen die Funktionalität nicht verändern oder neue Fehler verursachen.

- Große Methoden oder Klassen sollten möglichst in kleine, übersichtliche Pakete unterteilt werden, um besser verstanden und einfach verschoben werden zu können.

- Ein fortlaufender Rhythmus aus Anwenden kleiner Änderungen mit anschließendem Testen sorgt für ein schnelles und sicheres Refactoring.

Für die Durchführung von Refactoring existieren verschiedene Werkzeuge und Methoden. Fowler und Beck (2013) benennen vier essentielle Methoden, welche zu den häufig verwendeten Refactoring Methoden gehören.

- Move Method

  Ziel dieser Refactoring Methode ist es, eine Methode an eine andere Stelle – also in eine andere Klasse – zu verschieben. Ein Indiz hierfür sind vor allem große Klassen, welche eventuell auch eng mit anderen Klassen verwoben sind. Eine Methode ist in jene Klasse zu verschieben, welche die meisten Verwendungen aufweist. (Fowler & Beck, 2013)

- Extract Method

  Bei Code-Abschnitten, welche logisch zusammengehören und gut gruppiert werden können, empfiehlt sich die Refactoring Methode Extract Method. Hierbei erzeugt man für solche Code-Abschnitte eine eigene Methode, welche durch einen sprechenden Namen ihre Aufgabe klar kommuniziert und Software dadurch einfacher und verständlicher macht. Bei der Namensgebung der neuen Methoden ist darauf zu achten, dass zum Ausdruck kommt, was eine Methode tut und nicht wie sie es tut. (Fowler & Beck, 2013)

- Replace Temp with Query

  Ziel dieser Methode ist es, lokale, temporäre Variablen – welche Ergebnisse von Methoden speichern – zu entfernen, um dadurch weniger komplexen und verständlicheren Code zu erhalten. Dort, wo lokale Variablen verwendet werden, sollen diese durch den Methodenaufruf selbst ersetzt werden. (Fowler & Beck, 2013)

- Replace Conditional with Polymorphism

  Wenn im Code Bedingungen existieren, welche verschiedenes Verhalten aufgrund des Objekt-Typs definieren, kommt diese Refactoring Methode zum Ein-

satz. Häufige Indikatoren hierfür sind *if-then-else* Anweisungen oder *switch* Statements. In solchen Fällen sollten Abstraktionen und Sub-Klassen für die einzelnen Objekt-Typen erzeugt werden, welche dann durch Polymorphismus das gewünschte Verhalten implementieren. Vorteile dieser Methode sind eine einfache Integration neuer Typen, sowie eine Eliminierung von unnötigen Verzweigungen, welche üblicherweise – zudem an mehreren Stellen – im Sourcecode vorkommen. (Fowler & Beck, 2013)

## 3.9. Code Smells

Code Smells beschreiben in der Programmierung auffällige Muster und Strukturen in Quellcode, welche Indikatoren für mangelhaftes Design und schlechte Programmierpraktiken sind und eine Überarbeitung in Form von Refactoring nahelegen, um die Qualität in den Bereichen Wartbarkeit und Lesbarkeit zu verbessern. Grundsätzlich handelt es sich bei solchen Smells nicht unmittelbar um Fehler in einem Programm, sondern um schlecht strukturierten Programmcode, welcher schlecht lesbar und verständlich ist und dadurch weitere Arbeiten an der Software erheblich erschwert und Fehlerpotentiale beinhaltet. (Van Emden & Moonen, 2002)

Fowler und Beck (2013) haben mit detaillierten Beschreibungen definierter Code Smells weite Verbreitung und Anerkennung erlangt und erläutern in ihrer Publikation eine Vielzahl verschiedener Code Smells sowie geeignete Maßnahmen für Refactoring. Diese Smells reichen von einfachen Mustern, die jeder leicht erkennt – wie beispielsweise *Code Duplizierung* oder *lange Methode* – bis hin zu komplexeren Mustern aus der Objektorientierung – wie *Parallele Vererbungshierarchien* oder *Nachrichtenketten* (auch bekannt als *Law of Demeter*), wo Objekte auf interne Strukturen von anderen Objekten zugreifen. Weitere Beispiele sind *Große Klasse*, also Klassen mit zu vielen Feldern und Methoden, *Feature Envy*, wenn Klassen mehr an Methoden und Feldern anderer Klassen interessiert sind als an ihren eigenen, *Switch Anweisungen*, welche durch Vererbung und Polymorphismus besser lösbar wären, *Datenklassen*, also Klassen, welche nur Daten aber keine Funktionalität enthalten, *Spekulative Allgemeinheit*, wenn bereits im Vorfeld alle möglichen Spezialfälle vorbereitet werden, jedoch nie zum Einsatz kommen, *Datenklumpen*, wo Gruppen von Daten oft zusammen in Klassen als Felder vorkommen, jedoch nicht in einer eigenen Klasse gruppiert sind oder *Kommentare*, welche per se nicht schlecht sind, jedoch meist durch Methoden mit geeigneter Namensgebung ersetzt werden können. (Van Emden & Moonen, 2002; Fowler & Beck, 2013)

Eine Liste definierter Code Smells hat nie Anspruch auf Vollständigkeit, sie wird ständig erweitert und hat je nach Anwendungsgebiet eine andere Zusammenstellung. Zudem ist zu beachten, dass Code Smells immer auf subjektiven Aspekten wie Erfahrungen und persönlichen Meinungen beruhen und keine genaue Anleitung oder Vorgehensweise für Refactoring darstellen, wie Fowler und Beck (2013) betonen: „One

thing we won't try to do here is guve precise criteria for when a refactoring is overdue. In our experience no set of metrics rivals informed human intuition". Dennoch stellen Code Smells ein wichtiges Werkzeug in der Praxis dar, welches je nach Einsatzgebiet überlegt eingesetzt werden sollte. (Van Emden & Moonen, 2002)

Wie Khomh, Di Penta und Gueheneuc (2009) in ihrer Studie zu diesem Thema deutlich feststellen und belegen konnten, existiert ein Zusammenhang zwischen dem Vorhandensein von Code Smells und einer erhöhten Fehleranfälligkeit bei Änderungen und Erweiterungen der Codebasis. Aus diesem Grund liegt hier eine klare Empfehlung vor, den Fokus auf qualitätssichernde Maßnahmen und erhöhte Testaktivitäten zu legen, um solche Probleme frühzeitig aufzudecken. (Khomh et al., 2009)

## 3.10. Design Smells

Design Smells beschreiben – ähnlich wie Code Smells – schlechte Lösungen für gebräuchliche Implementierungs-Problemstellungen in der Softwareentwicklung und behindern die Entwicklung eines Systems langfristig dadurch, dass es für Softwareentwicklerinnen und Softwareentwickler zunehmend schwieriger wird, Änderungen in ein System einzupflegen. (Moha, Gueheneuc, Duchien & Le Meur, 2010)

Design Smells sind Code Smells sehr ähnlich, jedoch mit dem Unterschied, dass sie mehr bezogen auf die allgemeine Architektur von Software und damit einer höheren Ebene angesiedelt sind. Suryanarayana et al. (2015) definieren Design Smells treffend wie folgt: „Design smells are certain structures in the design that indicate violation of fundamental design principles and negatively impact design quality." Sinnbildlich kann man beispielsweise die Symptome eines Patienten mit Smells vergleichen, wohingegen die zugrunde liegende Krankheit das konkrete Design Problem darstellt. Auch hier ist es – um bei dem Beispiel zu bleiben – wichtig, nach Feststellung der Symptome und Ursachen eines Problems eine Diagnose für die richtige Behandlung – in der Softwareentwicklung das passende Refactoring – zu finden. Suryanarayana et al. (2015)

Ein wichtiger Vertreter und Begründer des Begriffs Design Smells ist Martin (2003), welcher sich in seiner Abhandlung intensiv mit Anzeichen von schlechtem Design in Programmen beschäftigt hat. Er sieht Design Smells vielmehr die Gesamtstruktur von Software betreffend, währenddessen Code Smells detaillierte Implementierungsdetails in Quellcode beschreiben. Häufige Ursache von Design Smells sind das Unwissen beziehungsweise Ignorieren von SOLID Prinzipien und führen langfristig zu einer Erosion von Software. (Martin, 2003)

Nachfolgend werden wichtige Design Smells nach Martin (2003) beispielhaft aufgelistet und erläutert.

- Fragility

  Mit Fragility werden Systeme beschrieben, welche bei einfachen Änderungen vergleichsweise anfällig für neue Probleme und Fehler sind. Häufig entstehen neue Fehler an Stellen, welche gar keine logische Verbindung zu der getätigten Änderung aufweisen, was eine Problemlösung schwerer macht.

- Rigididy

  Rigididy beschreibt die Starrheit eines Softwaresystems hinsichtlich Veränderung. Dies äußert sich meist durch überproportional hohe Zeitaufwände für vergleichsweise einfache Änderungsanforderungen. Die Ursache dafür liegt häufig an zu vielen, zu starken Abhängigkeiten verschiedener Module.

- Viscosity

  Ist das Design einer Software oder die verwendete Entwicklungsumgebung so beschaffen, dass es Entwickelnden schwer fällt, das Software Design zu erhalten, spricht Martin (2003) von dem Begriff Viskosität des Systems. Ziel soll es immer sein, Software Design so zu gestalten, dass es einfach fällt, gut strukturierten Code zu schreiben. Ebenso sollten verwendete Entwicklungsumgebungen und Werkzeuge einfach zu bedienen sein und die Entwicklungsarbeit nicht verlangsamen.

- Opacity

  Undurchsichtigkeit beschreibt das Maß an Verständlichkeit und Lesbarkeit von Sourcecode. In der Praxis lassen sich unverständliche Code-Abschnitte kaum vermeiden, weshalb Martin (2003) empfiehlt, nach gewisser zeitlicher Distanz den geschriebenen Code von den Autoren prüfen zu lassen, um die Lesbarkeit zu verbessern. Es kommt häufig vor, dass Entwickelnde während der Softwareerstellung undurchsichtige Muster und Deklarationen nicht als unverständlich wahrnehmen, doch wenn nach zeitlichem Abstand derselbe Code erneut gelesen wird, wird er oft von den Autoren selbst nicht wiedererkannt und verstanden, da Erinnerungen und Gedankengänge nicht mehr präsent sind.

- Needless Complexity

  Als unnötig komplex wird Software Design dann angesehen, wenn Infrastrukturen und Konstrukte ohne unmittelbaren Mehrwert existieren. Solche Konstrukte finden ihren Weg in Software oft durch unnötiges Vorausplanen von Entwicklerinnen und Entwicklern, indem komplexe Strukturen für mögliche spätere Einsätze geschaffen werden, welche jedoch in der Realität oft nicht eintreten. Grundsätzlich wird dieses Verhalten von Martin (2003) positiv gesehen,

wenn Software Design auf Zukunft ausgerichtet ist, jedoch wird dadurch oft der gegenteilige Effekt erreicht. Hier hilft es meistens, sich stets das Prinzip *Keep it simple and stupid (KISS)* vor Augen zu halten.

## 3.11. Zusammenfassung

Kernaussage dieses Kapitels ist, dass es zahlreiche Methoden, Prinzipien und sogenannte Antimuster gibt, welche für eine konstruktive Qualitätsgestaltung von Programmcode herangezogen werden können und auch sollen. Hierfür wurden in diesem Kapitel sowohl wichtige Prinzipien objekt-orientierter Softwareentwicklung – wie die SOLID-Prinzipien – erläutert, als auch viele Entwicklungsmethoden und Praktiken zur systematischen Verbesserung der Codequalität gezeigt. Im nachfolgenden Kapitel werden – darauf aufbauend – einige wichtige Metriken zur Messung und Beschreibung von Codequalität vorgestellt.

# 4. Codemetriken

## 4.1. Einleitung

In diesem Kapitel werden Lösungsansätze aufgezeigt, wie man Codequalität von Software objektiv messen und bewerten kann. Hierfür verwendet die Softwareentwicklung sogenannte Code- beziehungsweise Softwaremetriken, mit deren Hilfe Qualitätsaspekte von Software systematisch durch Erhebung bestimmter Kenngrößen ermittelt werden können. Hierbei können sowohl externe Aspekte wie Effizienz oder Funktionalität sowie auch interne Aspekte wie Änderbarkeit oder Testbarkeit erhoben werden. Das Ziel im Einsatz solcher Metriken ist in den meisten Fällen, Hersteller von Software darin zu unterstützen, ihre Produkte hinsichtlich verschiedener Qualitätsmerkmale zu prüfen und den Erfolg durch Verbesserungen objektiv festzuhalten. Weiters ist es das Ziel, Kosten und Terminplanung sowie den investierten Aufwand in die Entwicklung transparent zu halten. (Hoffmann, 2008; O'Regan, 2012)

Metriken fallen unter den Begriff der statischen Codeanalyse, da sie direkt auf Sourcecode angewendet werden, ohne diesen zuvor kompilieren oder laufen lassen zu müssen. Sie können automatisch oder manuell ausgeführt werden, wobei es ein guter Stil ist, solche Analysen automatisiert und kontinuierlich ausführen zu lassen, um permanente Transparenz über ein Softwareprojekt zu gewährleisten. (Sneed et al., 2010; Hoffmann, 2008)

Da die Anzahl unterschiedlicher Metriken sehr groß ist und es nicht das primäre Ziel dieser Arbeit ist, einen vollständigen Überblick über existierende Metriken zu geben, beschränkt sich dieses Kapitel auf die wichtigsten Metriken, welche in den gängigen, später noch vorgestellten Werkzeugen zur Durchführung statischer Codeanalyse eingesetzt werden. Ziel ist es, einen Einblick in die Grundlagen der Messung und Bewertung von Softwaresystemen zu gewähren.

## 4.2. Metriken für Größe und Komplexität

Eine der weit verbreiteten Metriken und Grundlage vieler Messungen ist die Metrik *Lines of code (LOC)*, welche eine Maßgröße darstellt, um die Größe der Codebasis ei-

ner Software auszudrücken. Als eine sehr einfache Metrik kann LOC bereits in groben Zügen als Maß von Codecomplexität herangezogen werden und wird meist ohne externe Tool-Unterstützung von den gängigen Entwicklungsumgebungen unterstützt. Aufbauend auf LOC existiert eine Erweiterung dieser Metrik unter der Bezeichnung *Non commented source statements (NCSS)*, welche sich mehr auf den eigentlichen Sourcecode, also Programmanweisungen, fokussiert und daher Kommentarzeilen aus der Messung ausnimmt. NCSS wird daher auch häufig eingesetzt, um die Qualität der Dokumentation einer Software zu eruieren. (Hoffmann, 2008)

Im wissenschaftlichen Diskurs wird die Aussagekraft dieser Metriken als eher gering eingeschätzt, da verschiedene Faktoren wie beispielsweise Programmierstil, Formatierung oder Code Standards erheblichen Einfluss auf die Resultate haben und daher nur mäßig für einen Vergleich herangezogen werden können. Um dieser Lücke zu begegnen, wurden andere Techniken mit dem Ziel entwickelt, reinen ausführbaren Code besser messen zu können, beispielsweise durch das Zählen von Anweisungsseparatoren. (Albrecht & Gaffney, 1983)

Um selbst verschiedene Programmiersprachen besser vergleichbar zu machen, entwickelte man in empirischer Forschung sogenannte Sprachfaktoren, welche eine Gewichtung für LOC und NCSS darstellen und auf diese Weise zu vergleichbaren Ergebnissen führen. Hierfür wurde Assembler als Referenzsprache ausgewählt und mit der Gewichtung von eins belegt. Die meisten Gewichtungen sind daher größer als der Referenzwert, etwa in Hochsprachen, welche deutlich mehr Codezeilen benötigen, um den selben Maschinencode zu generieren. (Thaller, 2000)

Doch selbst bei präzisem Abbilden der beiden Metriken durch erweiterte Techniken sehen Expertinnen und Experten diese Metriken als wenig geeignet an, um Software-Komplexität zu beschreiben. Dennoch sind sie bis heute in vielen modernen Berechnungsmodellen ein wichtiger Bestandteil, da sie eine wichtige Basisgröße für aufbauende Metriken darstellen und zudem auch gerne für Aufwandschätzungen in der Softwareentwicklung herangezogen werden, da der Aufwand in Wartung und Erweiterung durch einen Anstieg von Codezeilen ebenfalls erhöht wird. (Khoshgoftaar & Munson, 1990; Hoffmann, 2008)

## 4.3. Halstead Metriken

Bei den Halstead-Metriken handelt es sich um eine Sammlung aufeinander aufbauender Metriken zur Bestimmung von Code-Komplexität, welche 1977 von Maurice Howard Halstead erstmals der Öffentlichkeit vorgestellt wurden. Das Grundlegende seines Ansatzes war es, Quellcode in die wesentlichen Elemente einer jeden Programmiersprache, Operanden und Operatoren, zu untergliedern. Als Teil einer vorbereitenden Analyse werden daher Methoden, Variablen und Konstanten zu den Operan-

den gezählt, während logische Operatoren und Schlüsselwörter einzelner Programmiersprachen als Operatoren gewertet werden. Es soll jedoch erwähnt werden, dass die Unterscheidung zwischen Operanden und Operatoren in der Literatur durchaus kontrovers diskutiert wird und kein eindeutiger Konsens über eine Definition vorliegt, weshalb die Entscheidung in der Verantwortung des jeweiligen Anwenders liegt. (Halstead, 1979; Laird & Brennan, 2006)

Halstead definiert folgende aufeinander aufbauende Metriken, um letztendlich die Komplexität von Quellcode bestimmen zu können.

- Länge: Die Grundgesamtheit verwendeter Operanden und Operatoren bildet die Länge eines Programms.

$$Länge = N(Operanden) + N(Operatoren) \tag{4.1}$$

- Vokabular: Die Anzahl an Merkmalsausprägungen von Operatoren und Operanden bildet das sogenannte Vokabular.

$$Vokabular = n(Operanden) + n(Operatoren) \tag{4.2}$$

- Größe: Die Größe eines bestehenden Codeabschnittes wird nach Halstead wie folgt berechnet.

$$Größe = Länge \times \log_2(Vokabular) \tag{4.3}$$

- Komplexität: Mittels des Verhältnisses von Merkmalsausprägungen zur Grundgesamtheit von Operanden und Operatoren, lässt sich die Komplexität ermitteln.

$$Komplexität = \frac{n(Operanden)}{N(Operanten)} \times \frac{n(Operatoren)}{N(Operatoren} \tag{4.4}$$

- Aufwand: Letztlich definiert Halstead eine Formel zur Berechnung des Aufwandes, welcher zum Erstellen und Verstehen eines Programmes benötigt wird.

$$Aufwand = \frac{Größe}{Komplexität} \tag{4.5}$$

(Halstead, 1979)

Auch wenn die Halstead-Metriken sehr gut automatisiert eingesetzt werden können, besitzen sie in der Praxis wenig Aussagekraft, da sie Komplexität rein textuell bewerten. Zudem existieren kritische Betrachtungen in vorhandener Literatur, welche darauf hinweisen, dass Halstead seine Erkenntnisse nie empirisch bewiesen hat. Dennoch besitzen Halsteads Metriken als eine der ältesten, heute noch verwendeten Metriken ihre Berechtigung und dienen als Basis für weitere Metriken wie dem Maintainability Index. (Sneed et al., 2010; Welker & Oman, 1995)

## 4.4. McCabe Metrik

Mit der zyklomatischen Komplexität oder auch McCabe Metrik definierte Thomas J. McCabe 1976 erstmals eine Möglichkeit, Komplexität von Software zu berechnen. Diese Metrik gehört bis heute zu den bekanntesten und verbreitetsten Metriken. Im Gegensatz zu vergleichbaren Metriken betrachtet McCabe nicht lexikalische Programmelemente, sondern beschreibt Quellcode in Anlehnung an die bekannte Graphentheorie von Euler als einen gerichteten Graphen mit Knoten und Kanten. Auf diese Weise analysiert McCabe's Metrik die Anzahl verschiedener Kontrollflüsse eines Moduls, wobei sich die Komplexität mit steigender Anzahl solcher Pfade erhöht. (McCabe, 1976; Sneed et al., 2010)

Auf diese Weise entwickelte er folgende Formel zur Berechnung zyklomatischer Komplexität.

$$V(s) = e - n + 2p \tag{4.6}$$

Hierbei steht *e* für die Anzahl der Kanten, *n* für die Anzahl der Knoten und *p* für die Anzahl an Teilgraphen beziehungsweise Programmsprüngen. (McCabe, 1976; Sneed et al., 2010)

Um die Vorgehensweise der McCabe Metrik etwas anschaulicher zu machen, wird in Listing 4.1 ein Java-Beispiel gezeigt und danach analysiert.

```
0    public void feed(List<Eatable> foodPackage) {
1        for(Eatable food : foodPackage) {
2            if(hungry)
3                calories += food.getCalories();
4        }
5        if(calories > 1000) {
6            drinkWhiskey();
7        }
8    }
```

**Listing 4.1:** Beispiel für zyklomatische Komplexität

In Abb. 4.1 sieht man die Darstellung der Beispielmethode aus Listing 4.1 als Flussgraph mit den jeweiligen Zeilennummern abgebildet.

Werden alle Knoten zusammengezählt, erhält man $n = 7$. Die Anzahl der verschiedenen Kanten in diesem Beispiel ist $e = 9$. In diesem Beispiel existiert nur ein Programmsprung in Zeile 6, weshalb $p = 1$ ist. Wendet man nun die Formel von McCabe

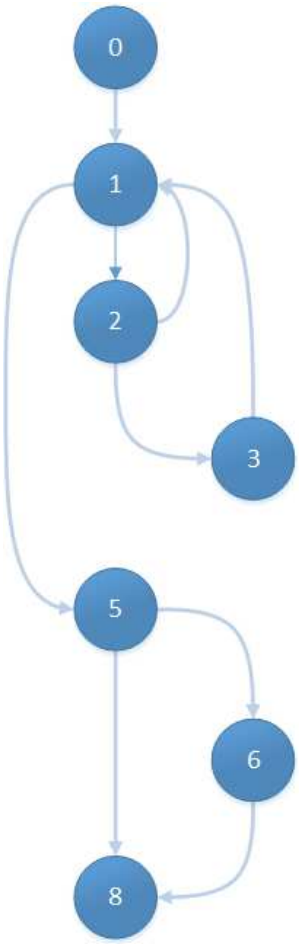



**Abbildung 4.1.:** Beispiel Flussgraph zyklomatische Komplexität (eigene Darstellung)

an, erhält man folgendes Ergebnis.

$$V(s) = 9 - 7 + 2 = 4 \tag{4.7}$$

Um die Ergebnisse dieser Metrik vernünftig einordnen und deuten zu können, hat McCabe den Wert 10 als eine sinnvolle obere Schranke für Komplexität vorgeschlagen, wobei dieser Wert als Empfehlung und nicht als festgeschrieben zu werten ist. Wie McCabe (1976) weiters betont, gibt diese Metrik nicht nur Auskunft über die objektive Komplexität von Software, sondern zeigt oft auch unterschiedliche Programmierstile von Entwicklerinnen und Entwicklern, wodurch sich Ergebnisse um den Faktor zehn verschieben können. (McCabe, 1976)

Zu ähnlichen Ergebnissen kommt auch andere Literatur. So definieren beispielsweise *Aivosto* (o. J.), wie in Tabelle 4.1 zu sehen, Empfehlungen für die Einordnung und Analyse dieser Metrik, indem Ergebnisse in verschiedene Kategorien und Risikostufen eingeteilt werden. Hierbei ist deutlich der Anstieg von Komplexität und dadurch gesteigerter Änderungsaufwand und Fehlerpotential zu erkennen.

| Zyklomatische Komplexität | Typisierung | Risikostufe |
|---|---|---|
| 1-4 | einfach | gering |
| 5-10 | gut strukturiert und stabil | gering |
| 11-20 | komplexer | moderat |
| 21-50 | besorgniserregend komplex | hoch |
| 50+ | instabil und fehleranfällig | sehr hoch |

**Tabelle 4.1.:** Einordnung der McCabe Metrik (*Aivosto*, o. J.)

Zudem erhöht sich mit steigender Komplexität auch die Wahrscheinlichkeit für sogenannte *Bad Fixes*, wie in Tabelle 4.2 zu sehen. Der Begriff Bad Fix beschreibt ein unwissentliches Erzeugen weiterer Bugs während der Behebung bestehender Bugs.

| Zyklomatische Komplexität | Bad-Fix-Wahrscheinlichkeit |
|---|---|
| 1-10 | 5 % |
| 20-30 | 20 % |
| 50+ | 40 % |
| 100 | 60 % |

**Tabelle 4.2.:** Bad-Fix-Wahrscheinlichkeit (*Aivosto*, o. J.)

## 4.5. Maintainability Index

Die Metrik *Maintainability Index (MI)* ist eine Berechnungsform, welche sich aus der Kombination anderer Metriken zusammensetzt. Ursprünglich wurde sie in ihrer Erstausführung von Oman, Hagemeister und Ash (1991) in zwei Varianten – einmal mit und einmal ohne Kommentare – definiert. Später wurde sie in weiteren Studien punktuell modifiziert. In den meisten Definitionen dieser Metrik werden die Metriken Lines of Code (LOC) (Abschnitt 4.2), Halstead (Abschnitt 4.3) sowie McCabe's zyklomatische Komplexität (Abschnitt 4.4) in einer Formel auf verschiedene Arten gewichtet und vereint. (Welker, 2001)

Nachfolgend ist eine Formel zur Berechnung des Maintainability Index nach Welker und Oman (1995) angeführt.

$$\begin{aligned} MI = {} & 171 - 5.2ln(aV) - 0.23 \times aG - 16.2ln(aLOC) \\ & + 50sin(\sqrt{2.4 \times perCM}) \end{aligned} \tag{4.8}$$

wobei

$aV =$ durchschnittliche Halstead Größe pro Modul
$aG =$ durchschnittlicher McCabe Wert pro Modul
$LOC =$ durchschnittliche Codezeilen pro Modul
$perCM =$ durchschnittlicher Prozentanteil an kommentierten Zeilen pro Modul

Hauptzweck dieser Metrik ist es, die Wartbarkeit von Quellcode im Sinne von Änderbarkeit und Erweiterbarkeit zu messen. Somit ist es möglich, den Trend der Wartbarkeit einer Software ständig im Auge zu behalten und mit vorherigen Softwareständen zu vergleichen. Diese Metrik findet sowohl in der Wirtschaft, als auch beim Militär, breite Anwendung. (Welker & Oman, 1995)

In der Literatur wird in einigen Studien lebhaft über die genaue Berechnung und Anwendung diskutiert. So sind beispielsweise Kommentarzeilen ein häufiges Thema für Diskussionen, da diese oft menschlicher Begutachtung bedürfen, um auf die Sinnhaftigkeit bewertet werden zu können. Aus den verschiedenen Diskussionen heraus haben sich daher weitere Verfeinerungen und Modifikationen des Maintainability Index gebildet. (Welker, 2001; Liso, 2001; Lowther, 1993)

## 4.6. Chapins Q-Complexity

Mit der Q-Komplexität verfolgte Chapin (1979) den Ansatz, die Komplexität eines Moduls aus der Gesamtheit der Komplexität seiner Daten zu verstehen. Hierfür unterscheidet er vier unterschiedliche Datenklassen, um diese differenziert zu gewichten.

- Eingaben
- Ausgaben
- Prädikate
- Transitdaten

(Sneed et al., 2010)

Eingaben beziehungsweise Argumente zur Prozessierung haben eine Gewichtung von *1*, Ausgaben (Ergebnisse) eine Gewichtung von *2*, Prädikate eine Gewichtung von *3* und Transitdaten eine Gewichtung von *0.5*. (Sneed et al., 2010; Chapin, 1979)

Um die Vorgehensweise für die Berechnung einfach nachvollziehen zu können, wird in Listing 4.2 ein einfaches Beispiels vorgestellt, welches nachfolgend analysiert wird.

```
0    public void doStuff (Object param1, Object param2, Object ↩
        param3, Object param4) {
1        if (param1 > 0)
2            param3 = param1 + x;
3        else
4            param4 = param1 - y;
5        doOtherStuff (param2);
6    }
```

**Listing 4.2:** Beispiel für Q-Komplexität

Betrachtet man dieses Beispiel, lassen sich folgende Datenkategorien zuteilen.

- param1, x, y sind Argumente: $3 \times 1 = 3$
- param3, param4 sind Ergebnisse: $2 \times 2 = 4$
- param1 ist zudem Bedingungsoperator (Prädikat): $1 \times 3 = 3$
- param2 ist ein Transitparameter: $1 \times 0.5 = 0.5$

Wenn nun alle Ergebnisse addiert werden, erhält man für dieses Beispiel eine Q-Komplexität von $10.5$. Schlussendlich wird die Anzahl der benutzten Daten mit der Gewichtung der Daten in Relation gestellt, um eine rationale Skala und dadurch die endgültige Datenkomplexität eines Modules zu erhalten.

$$Datenkomplexitaet = \frac{Anzahl der Daten}{GewichteteQ - Komplexitaet} \tag{4.9}$$

Für das Beispiel aus Listing 4.2 ergibt das eine berechnete Gesamtkomplexität von $\frac{6}{10.5} = 0.57$. (Sneed et al., 2010)

## 4.7. Metriken für objekt-orientierte Programmiersprachen

Viele der zuvor vorgestellten Metriken existieren bereits viele Jahre und wurden zu Zeiten entwickelt, wo Programme noch völlig andere Strukturen und Architekturen

aufwiesen als heutige Software. Mit dem Wandel zu objekt-orientierten Programmiersprachen entstand auch die Notwendigkeit, neue Metriken speziell für objektorientierte Programmstrukturen zu entwickeln, um diese besser messen und bewerten zu können. Wichtige Ansätze hierfür lieferten Chidamber und Kemerer (1994) und Lieberherr, Holland und Riel (1988), welche mit ihren Metriken breite Anwendung und Akzeptanz in der heutigen statischen Code Analyse genießen und in zahlreichen Werkzeugen eingesetzt werden. Diese Metriken wurden speziell für objektorientierte Konzepte wie Klassen, Vererbung, Kapselung oder Polymorphismus entwickelt und stellen einen wichtigen Anteil für die Prüfung objekt-orientierter Software dar. (Chidamber & Kemerer, 1994; Lieberherr et al., 1988; Li & Henry, 1993)

Nachfolgend werden die sechs Metriken nach Chidamber und Kemerer (1994) sowie das Law of Demeter von Lieberherr et al. (1988) vorgestellt.

- CBO - Coupling between Classes

  CBO analysiert Kopplung und Abhängigkeiten zwischen Klassen. Kopplungen entstehen durch Kommunikation mit anderen Klassen (Methodenaufrufe), welche nicht über Vererbung miteinander verbunden sind. Eine hohe Kopplung widerspricht dem Konzept objekt-orientierter Strukturen und hat negativen Einfluss auf die Komplexität einer Klasse. Zudem erschwert sie Wiederverwendbarkeit und Änderbarkeit vorhandener Code-Konstrukte und führt zu erhöhtem Test- und Verwaltungsaufwand. Oft ist eine hohe Kopplung auch Indikator für eine schlechte Aufteilung von Verantwortlichkeiten. (Chidamber & Kemerer, 1994)

- DIT - Depth in inheritance tree

  Diese Metrik betrachtet bestehende Vererbungshierarchien von Klassen. Hier gilt die Regel, dass eine erhöhte Vererbungstiefe zu einer starreren, komplexeren Struktur führt, da eine Klasse viele Elemente und Methoden von Superklassen erbt und dadurch fehleranfälliger wird. In Maßen ist Vererbung durchaus zu empfehlen und auch Indiz für gute Strukturierung und hohe Wiederverwendbarkeit, jedoch steigt bei großen Vererbungsbäumen die Gefahr, dass es zu Seiteneffekten durch Änderungen an Basisklassen kommt. In solchen Fällen empfiehlt es sich, Komposition anstelle von Vererbung zu verwenden.
  (Chidamber & Kemerer, 1994)

- LCOM - Lack of cohesion in methods

  Mit Hilfe dieser Metrik wird versucht, den Mangel an Kohäsion einer Klasse zu messen. Hierfür wird die Benutzung gemeinsam verwendeter Instanzvariablen durch verschiedene Methoden analysiert. Ein schlechter LCOM Wert ist ein Indiz dafür, dass ein Auftrennen in verschiedene Klassen sinnvoll wäre. (Chidamber & Kemerer, 1994)

- NOC - Number of children

  Diese Metrik bestimmt die Anzahl von Subklassen und bestimmt das Maß guter Wiederverwendbarkeit. Durch diese Messung wird aufgezeigt, wie wichtig eine Klasse ist und welche Auswirkungen durch die Änderung an einer solchen Klasse zu erwarten sind. Durch eine Vielzahl an Subklassen wird auch der Testaufwand bei Änderungen erhöht, da alle Subklassen ebenfalls getestet werden müssen. (Chidamber & Kemerer, 1994)

- RFC - Response for a class

  RFC ist eine Metrik, welche die Anzahl an Methodenaufrufen auf eine bestimmte Klasse inklusive der eigenen Methoden erhebt. Auf diese Weise zeigt diese Metrik die Auswirkungen eines Aufrufs einer Methode einer Klasse und bietet somit Aufschluss über die Komplexität einer Klasse. Auch hier wird der Testaufwand durch hohe Komplexität erhöht, sowie die Wiederverwendbarkeit beeinträchtigt. (Chidamber & Kemerer, 1994)

- WMC - Weighted methods per class

  Die Metric WMC bestimmt die gesamte Komplexität einer Klasse durch die Summierung der Komplexität einzelner Methoden. Zur Bestimmung der Komplexität wird meist die zyklomatische Komplexität von McCabe herangezogen. Mithilfe dieser Metrik lassen sich komplexe Methoden sowie komplexe oder zu große Klassen identifizieren. Zudem gibt diese Metrik Aufschluss über den Aufwand bei Wartung oder Erweiterung einer Klasse. Ein hoher Wert deudet überdies auf erhöhtes Fehlerrisiko hin, weshalb es oft sinnvoll ist, Klassen aufzuteilen. (Chidamber & Kemerer, 1994)

- Law of Demeter

  Das Law of Demeter beschreibt die Grundregel, dass Objekte nur mit anderen Objekten in unmittelbarer Nähe kommunizieren sollen und nicht auf die interne Struktur dieser zugreifen dürfen. Dadurch soll verhindert werden, dass Objekte die innere Struktur anderer Objekten kennen und diese verändern können. Diese Regel soll das Bilden langer Message-Ketten und die damit einhergehende starke Kopplung vermeiden. (Lieberherr et al., 1988)

Die erläuterten objekt-orientierten Metriken bieten eine gute Übersicht über den richtigen Umgang mit objekt-orientierten Strukturen und sind oft Indiz für den Aufwand von Änderungen und Wartung sowie die allgemeine Fehleranfälligkeit von Software. (Li & Henry, 1993)

## 4.8. Zusammenfassung

In diesem Kapitel wurden nun einige, verbreitete Codemetriken gezeigt, welche allesamt dafür eingesetzt werden, Code und dessen interne Strukturen zu messen und dadurch Auskunft über deren Zustand zu geben. Während in den ersten Abschnitten altbewährte, traditionelle Metriken vorgestellt wurden, so sind gerade auch neuere, objekt-orientierte Metriken besonders spannend, da der Großteil heutiger Softwarestrukturen nach objekt-orientierten Ansätzen aufgebaut ist. Anschließend an diese Informationen werden nun im folgenden Kapitel Werkzeuge vorgestellt, welche speziell im Java-Umfeld dafür eingesetzt werden, Codemetriken und andere qualitätsrelevante Faktoren automatisiert zu messen und diese Ergebnisse oftmals auch visuell aufzubereiten und darzustellen.

# 5. Statische Code-Analyse-Werkzeuge

## 5.1. Einleitung

In Kapitel 4 wurden verschiedene Möglichkeiten zur Messung einzelner Qualitätsaspekte dargestellt, welche ein sehr gutes Mittel bieten, um die einzelnen Qualitätsfaktoren von Sourcecode angemessen beurteilen und abbilden zu können. Oft sind diese Berechnungen jedoch sehr aufwändig und können besonders bei großen Mengen an Codezeilen nicht manuell berechnet werden. Um gerade große Projekte automatisiert und zeitsparend auswerten zu können, existieren zahlreiche Werkzeuge, welche einzelne bis hin zu ganzen Sammlungen von Codemetriken berechnen. Generell gibt es zwei Arten von Werkzeugen für die statische Codeanalyse. Die einen analysieren Programmcode, während die anderen auf der Basis von kompilierten Bytecode operieren. Dennoch arbeiten die meisten solcher Werkzeuge in einer ähnlichen Art und Weise, sie konstruieren – basierend auf Sourcecode – verschiedene Modelle, auf welchen dann verschiedene Muster entdeckt werden können. (Louridas, 2006)

In diesem Kapitel werden einige solcher Werkzeuge dargestellt und deren Funktionsweise anhand kurzer Beispiele erläutert. Hierbei erstreckt sich die Spanne von einfachen Werkzeugen, welche die berechneten Metriken in einer einfachen Konsolenausgabe darstellen, bis hin zu aufwändigen Programmen mit zahlreichen Einstellungsmöglichkeiten und aufwändigen Darstellungen in Form erstellter Reports oder Auswertungsmasken. An diesem Punkt gilt es nochmals zu betonen, dass in dieser Arbeit der Fokus auf der Programmiersprache Java liegt, weshalb die hier vorgestellten Werkzeuge allesamt aus dem Java-Umfeld stammen und zumindest Java-Code analysieren können.

## 5.2. Checkstyle

Das Werkzeug Checkstyle ist OpenSource Software, welche unter der LGPL Lizenz betrieben wird. Checkstyle unterscheidet sich gegenüber anderen vorgestellten Tools

dadurch, dass Reports und Analysen nicht manuell oder periodisch gestartet werden, sondern dass es dem Benutzer laufend während der Entwicklung Probleme und Verbesserungen aufzeigt. Hierfür existieren auch für dieses Werkzeug Plugins für gängige Entwicklungsumgebungen (IntelliJ, Eclipse, NetBeans, ...). In Abb. 5.1 ist beispielhaft die Integration von Checkstyle in die Integrierte Entwicklungsumgebung (IDE) Eclipse dargestellt. (*Checkstyle Project Website*, o. J.)

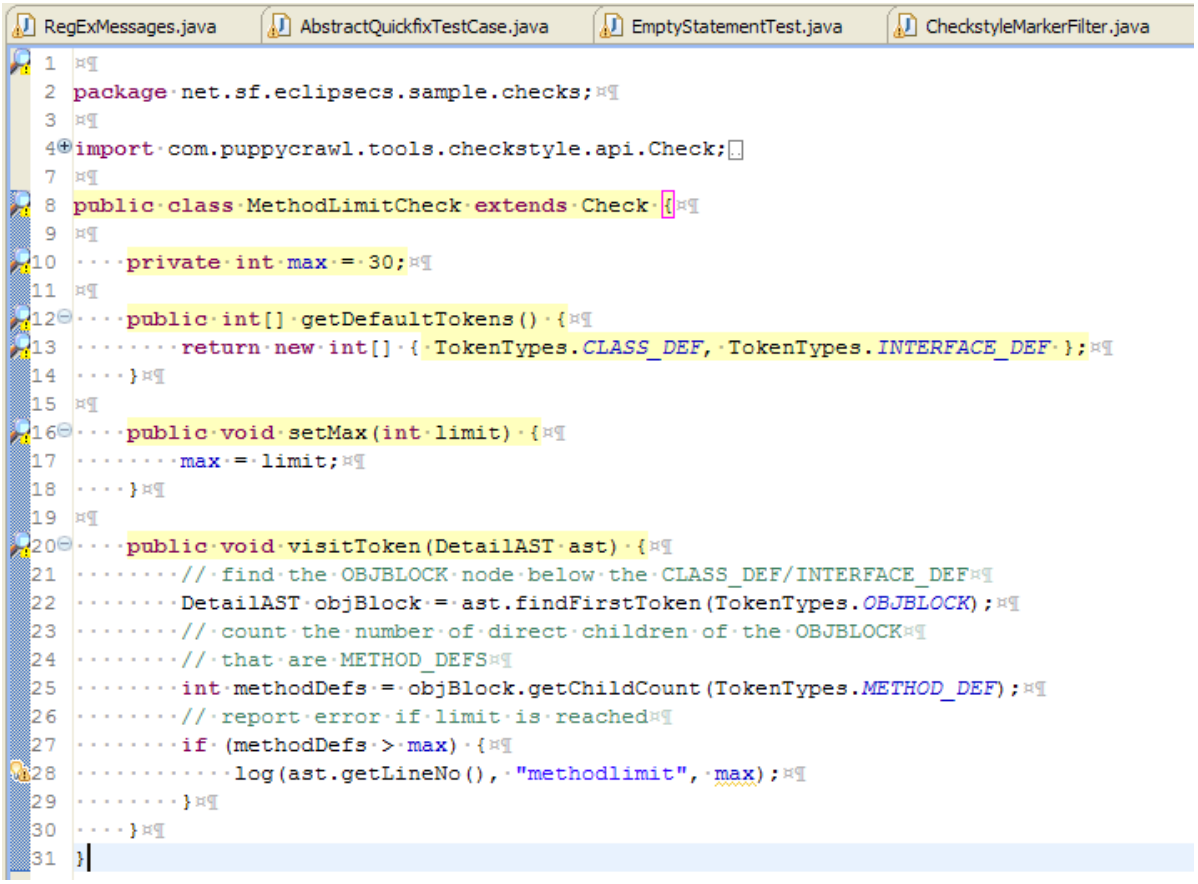



**Abbildung 5.1.:** Checkstyle Plugin Eclipse (*Checkstyle Project Website*, o. J.)

Checkstyle verwendet wie die meisten Werkzeuge ein Set an Regeln, welche je nach Einsatzwunsch frei konfiguriert sowie deaktiviert werden können. Zudem lassen sich für verschiedene Verwendungen mehrere Konfigurationssets erstellen, was es beispielsweise möglich macht, verschiedene Regeln für Java-Quellcode und JUnit Testklassen anzuwenden. Diese Konfigurationen liegen im XML-Format vor und können auf einfache Weise zentral innerhalb des Entwicklungsteams gewartet und verteilt werden, um einen einheitlichen Einsatz des Werkzeuges an jedem Entwicklungs-Rechner sicherzustellen. Werden Regeln verletzt, so erscheinen direkt Warnungen in der IDE – ähnlich Compiler-Warnungen – und so kann direkt reagiert werden. Besonderer Fokus von Checkstyle liegt in der Überprüfung von *weichen* Regeln, wie es in vielen Code-Konventionen definiert ist – wie beispielsweise syntaktische und lexikalische Auffälligkeiten. (*Checkstyle Project Website*, o. J.)

Checkstyle definiert folgende Unterteilungen der verschiedenen Regeln.

- Annotations: Überprüfung auf Korrektheit und der richtigen Platzierung von Annotationen.
- Block Checks: Sicherstellung einheitlicher Klammersetzung und Block-Formatierungen, sowie aufdecken nicht verwendeter Blöcke, welche meist Debug-Überreste sind.
- Class Design: Überprüfung auf Sichtbarkeiten von Variablen, sowie sinnvolle *final*-Deklarationen.
- Coding: Beispielsweise beidseitige Implementierung von *equals()* und *hashcode()*.
- Headers: Prüfung von beispielsweise Copyright Deklarationen zu Beginn einer Datei.
- Imports: Überprüfung von nicht verwendeten oder falsch deklarierten Import Statements.
- Javadoc Comments: Überprüfung von vorhandenen sowie validen Javadoc Kommentaren von Klassen und Methoden.
- Metrics: Analyse von Grenzwerten verschiedener Metriken für Klassen und Methoden.
- Miscellaneous: Verschiedene Überprüfungen wie beispielsweise nicht gelöschte TODO-Notizen.
- Modifiers: Prüfung auf Redundanz von Modifiern wie *public*, *protected*, *private*, *synchronized* und auf einheitliche Anordnung der Reihenfolgen innerhalb der Signaturen von Klassen und Methoden.
- Naming Conventions: Regeln für Bezeichnungen von Klassen, Methoden, Variablen und anderen Konstrukten, beispielsweise definierte Groß- und Kleinschreibung.
- Regexp: Verschiedene Prüfungen mittels definierter Regular Expressions.
- Size Violations: Definieren von Maximallängen für Textzeichen einer Zeile sowie Zeilenlängen von Klassen und Methoden.
- Whitespace: Aufzeigen von unnötigen Leerzeichen oder Abständen.

(*Checkstyle Project Website*, o. J.)

In der aktuellen Version von Checkstyle finden sich zahlreiche Regeln der verschiedenen Kategorien und diese werden von der Open-Source Community ständig weiter gewartet. Zusätzlich ist es jederzeit mit wenig Aufwand möglich, eigene Regeln zu definieren, um auf spezielle Richtlinien eingehen zu können. (*Checkstyle Project Website*, o. J.)

## 5.3. PMD

Das Programm PMD ist ein Werkzeug zur statischen Analyse von Javacode und Code in anderen Programmiersprachen und steht unter der Open-Source BSD-Lizenz. Über die genaue Bedeutung des Namens gibt es keine offiziellen Angaben, jedoch wird es in der Community oft als Project Mess Detector bezeichnet. (*PMD Project Website*, o. J.)

Im Gegensatz zu anderen Analysewerkzeugen berechnet PMD keine definierten Kennzahlen beziehungsweise Metriken, sondern PMD findet und deklariert ineffiziente Stellen in Sourcecode, sowie mögliche potentielle Probleme. Zusätzlich bietet PMD in den meisten Fällen auch praktische Beispiele und Lösungsmöglichkeiten an, um bestmöglich reagieren zu können. Im Kern verwendet PMD eine Vielzahl verschiedener Regelsammlungen für unterschiedliche Themen, welche durch praktische Erfahrungen aus der Softwareentwicklung gesammelt und weiterentwickelt werden. Dabei kann für jedes Projekt der Einsatz der zu verwendenden Regeln konfiguriert und genau an die Anforderungen angepasst werden. Zudem lassen sich auch neue, eigene Regeln definieren und hinzufügen. (*PMD Project Website*, o. J.)

Folgende Auffälligkeiten und Probleme können beispielsweise von PMD entdeckt und dargestellt werden.

- Duplicate Code
- Mögliche Fehler durch beispielsweise leere *try/catch/finally* oder *switch* Anweisungen
- Große Methoden oder Klassen, welche aufgeteilt werden sollten
- Falscher Einsatz von oft verwendeten Klassen wie beispielsweise String oder Stringbuffer
- Toter Code in Form ungenutzter Variablen und Methoden

- Unnötig komplizierte Anweisungen wie beispielsweise nutzlose *if* Ausdrücke oder *for* Schleifen, die stattdessen als *while* Schleifen verwendet werden könnten
- Potentielle Probleme durch mehrere Ausstiegspunkte aus Methoden
- Eine zu hohe zyklomatische Komplexität in Klassen
- Falscher Einsatz von Test-Implementierungen beispielsweise in JUnit Tests

(*PMD Project Website*, o. J.)

Die Einsatzmöglichkeiten von PMD reichen von einfachen Konsolenausführungen bis hin zu automatisierten Überprüfungen vorhandener Projekte. Hierfür existieren Plug-Ins für gängige Entwicklungsumgebungen wie beispielsweise Eclipse, Netbeans oder IntelliJ IDEA und Erweiterungen für weit verbreitete Build-Tools wie Apache Maven und Continuous-Integration-Server wie Jenkins. (*PMD Project Website*, o. J.)

## 5.4. FindBugs

FindBugs ist ein Programm, welches durch statische Analysepattern speziell Javacode auf mögliche Bugs hin überprüft. Es ist unter der freien GPL-Lizenz verfügbar und hat die Eigenheit, dass es nicht auf den Quellcode, sondern den resultierenden Java-Bytecode angewendet wird. Auch für dieses Werkzeug sind verschiedene Plugins für verschiedene Entwicklungsumgebungen verfügbar, welche durch eigene BugDetektoren erweitert werden können, und es existiert überdies eine eigenständige grafische Anwendung wie in Abb. 5.2 zu sehen. (*FindBugs Project Website*, o. J.)

In der Analyse teilt FindBugs gefundene Fehler in folgenden Kategorien ein:

- Correctness
- Performance
- Security
- Bad Practices
- Malicious code vulnerability
- Multithreaded Correctness

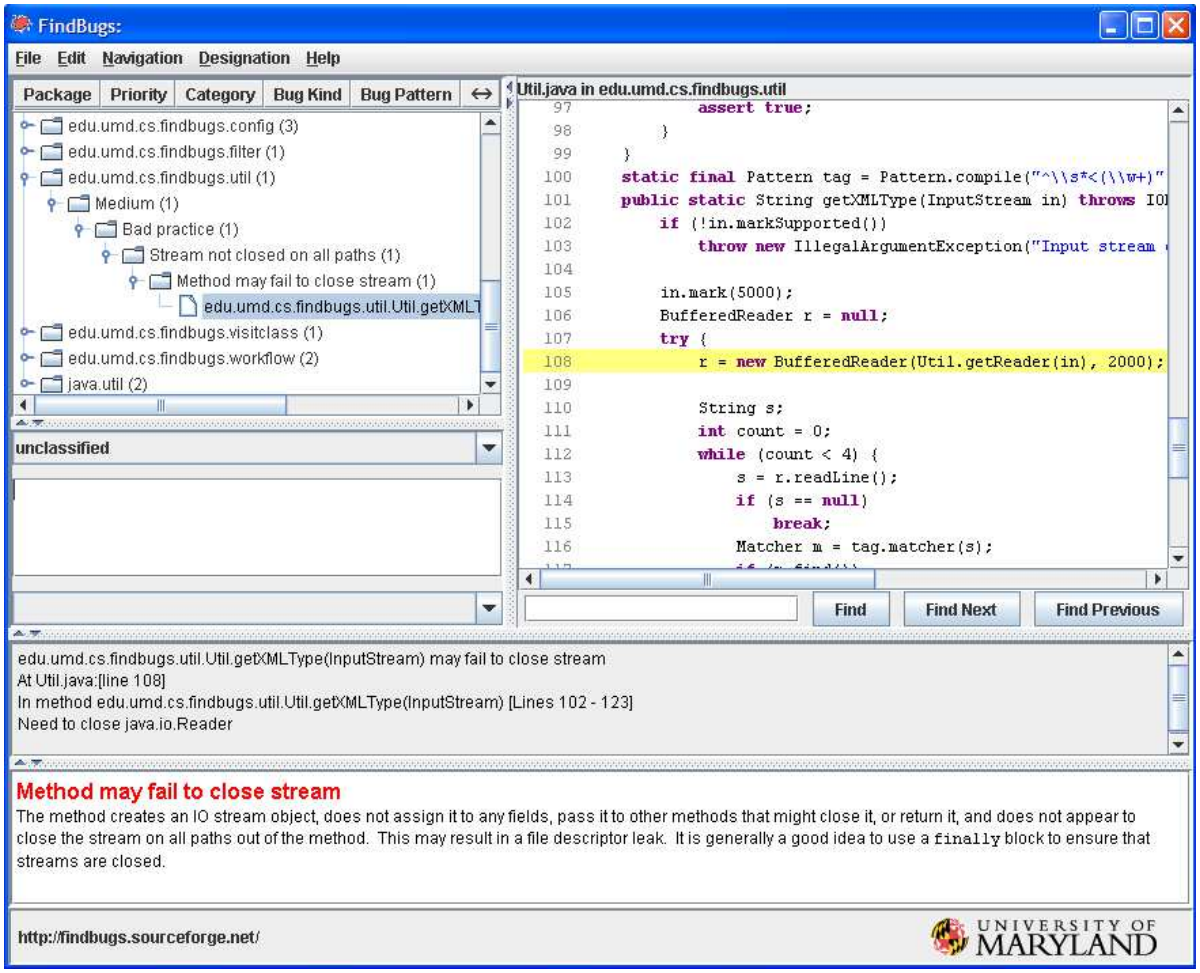



**Abbildung 5.2.:** FindBugs GUI (*FindBugs Project Website*, o. J.)

- Dodgy Code
- Experimental
- Internationalization

(*FindBugs Project Website*, o. J.)

Nachfolgend werden einige beispielhafte Bugs aufgezeigt, welche durch FindBugs aufgedeckt werden können.

- String Konkatenation mittels + innerhalb von Schleifen, wo bei jedem Durchlauf neue String Instanzen angelegt und Inhalte kopiert werden, was schlecht für die Performance ist. FingBugs empfiehlt in solchen Fällen *StringBuilder* oder *StringBuffer* zu verwenden.

- Strings werden mit == oder *!*= verglichen, wobei in solchen Fällen Java auf gleiche Referenz vergleicht und nicht auf den Inhalt. Dieses Muster zeigt in den meisten Fällen fehlerhaftes Verhalten an, da selten die Referenz eines String-Objektes, sondern vielmehr deren Inhalt verglichen werden möchte.

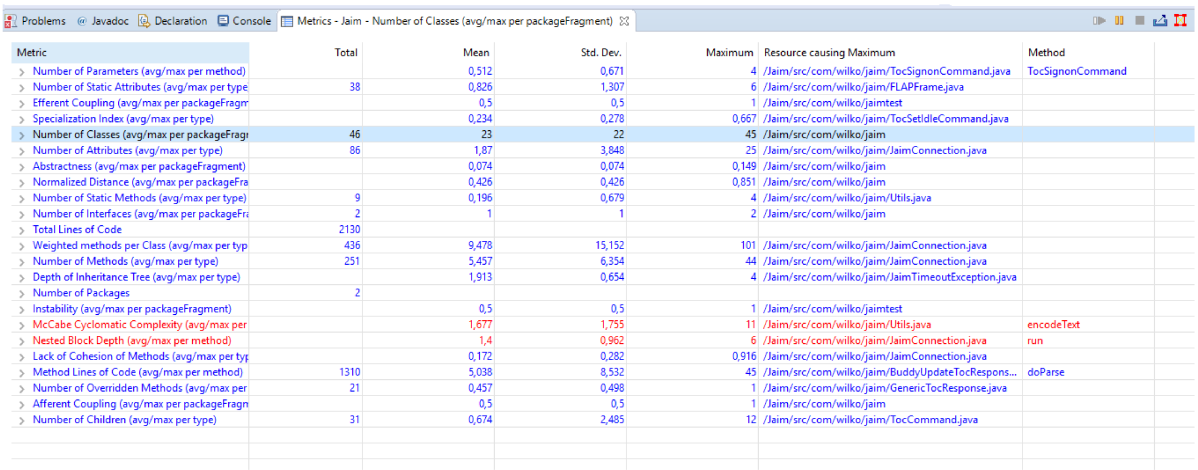
- Ignorieren von Exceptions, wo stattdessen in sauberem Code jede Exception angemessen durch Logging, Weiterwerfen oder Behandlung verarbeitet werden sollte.

- *Casten* von Integer-Divisionen auf Float oder Double, was zur Folge hat, dass Java automatisch rundet und so Berechnungspräzision verloren geht, was in kritischen Anwendungensfällen zu großen Problemen führen kann.

(*FindBugs Project Website*, o. J.)

## 5.5. Eclipse Metrics Plugin 1.3.6

Das Werkzeug Eclipse Metrics Plugin 1.3.6 ist, wie der Name schon verrät, ein freies Open-Source Plugin für die IDE Eclipse, welches zur Berechnung verschiedener Metriken und der Analyse von Modul- und Klassen-Abhängigkeiten verwendet werden kann. Das Sprectrum reicht dabei von Metriken zur Berechnung zyklomatischer Komplexität über Metriken zur Analyse von verschiedenen Abhängigkeiten bis hin zu graphischen Darstellungen von Abhängigkeitsnetzwerken. Dieses Werkzeug wurde als Plugin für die IDE Eclipse entwickelt und ist in Abb. 5.3 beispielhaft dargestellt. Zudem verfügt es über eine Unterstützung von Apache Ant, wodurch es auch einfach auf Buildservern automatisiert eingesetzt werden kann. (*Eclipse Metrics Plugin 1.3.6*, o. J.)


Problems Javadoc Declaration Console Metrics - Jaim - Number of Classes (avg/max per packageFragment)

| Metric | Total | Mean | Std. Dev. | Maximum | Resource causing Maximum | Method |
|---|---|---|---|---|---|---|
| Number of Parameters (avg/max per method) | | 0,512 | 0,671 | 4 | /Jaim/src/com/wilko/jaim/TocSignonCommand.java | TocSignonCommand |
| Number of Static Attributes (avg/max per type | 38 | 0,826 | 1,307 | 6 | /Jaim/src/com/wilko/jaim/FLAPFrame.java | |
| Efferent Coupling (avg/max per packageFragm | | 0,5 | 0,5 | 1 | /Jaim/src/com/wilko/jaimtest | |
| Specialization Index (avg/max per type) | | 0,234 | 0,278 | 0,667 | /Jaim/src/com/wilko/jaim/TocSetIdleCommand.java | |
| Number of Classes (avg/max per packageFragr | 46 | 23 | 22 | 45 | /Jaim/src/com/wilko/jaim | |
| Number of Attributes (avg/max per type) | 86 | 1,87 | 3,848 | 25 | /Jaim/src/com/wilko/jaim/JaimConnection.java | |
| Abstractness (avg/max per packageFragment) | | 0,074 | 0,074 | 0,149 | /Jaim/src/com/wilko/jaim | |
| Normalized Distance (avg/max per packageFra | | 0,426 | 0,426 | 0,851 | /Jaim/src/com/wilko/jaim | |
| Number of Static Methods (avg/max per type) | 9 | 0,196 | 0,679 | 4 | /Jaim/src/com/wilko/jaim/Utils.java | |
| Number of Interfaces (avg/max per packageFra | 2 | 1 | 1 | 2 | /Jaim/src/com/wilko/jaim | |
| Total Lines of Code | 2130 | | | | | |
| Weighted methods per Class (avg/max per typ | 436 | 9,478 | 15,152 | 101 | /Jaim/src/com/wilko/jaim/JaimConnection.java | |
| Number of Methods (avg/max per type) | 251 | 5,457 | 6,354 | 44 | /Jaim/src/com/wilko/jaim/JaimConnection.java | |
| Depth of Inheritance Tree (avg/max per type) | | 1,913 | 0,654 | 4 | /Jaim/src/com/wilko/jaim/JaimTimeoutException.java | |
| Number of Packages | 2 | | | | | |
| Instability (avg/max per packageFragment) | | 0,5 | 0,5 | 1 | /Jaim/src/com/wilko/jaimtest | |
| McCabe Cyclomatic Complexity (avg/max per | | 1,677 | 1,755 | 11 | /Jaim/src/com/wilko/jaim/Utils.java | encodeText |
| Nested Block Depth (avg/max per method) | | 1,4 | 0,962 | 6 | /Jaim/src/com/wilko/jaim/JaimConnection.java | run |
| Lack of Cohesion of Methods (avg/max per typ | | 0,172 | 0,282 | 0,916 | /Jaim/src/com/wilko/jaim/JaimConnection.java | |
| Method Lines of Code (avg/max per method) | 1310 | 5,038 | 8,532 | 45 | /Jaim/src/com/wilko/jaim/BuddyUpdateTocRespons... | doParse |
| Number of Overridden Methods (avg/max per | 21 | 0,457 | 0,498 | 1 | /Jaim/src/com/wilko/jaim/GenericTocResponse.java | |
| Afferent Coupling (avg/max per packageFragn | | 0,5 | 0,5 | 1 | /Jaim/src/com/wilko/jaim | |
| Number of Children (avg/max per type) | 31 | 0,674 | 2,485 | 12 | /Jaim/src/com/wilko/jaim/TocCommand.java | |

**Abbildung 5.3.:** Eclipse Metrics Plugin 1.3.6 (*Eclipse Metrics Plugin 1.3.6*, o. J.)

Nachfolgend sind jene Metriken angeführt, welche durch das Eclipse Metrics Plugin berechnet werden können.

- McCabe's cyclomatic complexity
- Number of classes
- Number of methods
- Nunber of overridden methods
- Number of interfaces
- Number of children
- Number of fields
- Depth of inheritance tree
- Lines of code
- Lack of cohesion of methods (Henderson-Sellers)
- Weighted methods per class
- Afferent coupling
- Efferent coupling
- Normalized distance from main sequence
- Abstractness

(*Eclipse Metrics Plugin 1.3.6*, o. J.)

## 5.6. Chidamber & Kemerer Java Metrics

CKJM oder auch Chidamber & Kemerer Java Metrics ist ein Werkzeug zur Berechnung objekt-orientierter Metriken nach Chidamber und Kemerer. Es wurde von Diomidis Spinellis als frei verfügbare Open-Source Software aus Mangel an verfügbaren

Alternativen entwickelt, mit dem klaren Ziel, eine einfaches, effizientes Werkzeug zu erschaffen, welches alle wichtigen objekt-orientierten Metriken von Chidamber und Kemerer schnell berechnen kann. Dieses Tool arbeitet daher als einfache Konsolenanwendung auf der Basis von Bytecode kompilierter Java-Klassen wie in Abb. 5.4 beispielhaft in einer Konsolenausgabe zu sehen ist. (Spinellis, 2005; *CKJM*, o. J.; Chidamber & Kemerer, 1994)

The command's output will be a list of class names (prefixed by the
package they are defined in), \nfollowed by the corresponding
metrics for that class: WMC, DIT, NOC, CBO, RFC, LCOM, Ce, and NPM.

```
gr.spinellis.ckjm.ClassMetricsContainer 3 1 0 3 18 0 2 2
gr.spinellis.ckjm.MethodVisitor 11 1 0 21 40 0 1 8
gr.spinellis.ckjm.CkjmOutputHandler 1 1 0 1 1 0 3 1
gr.spinellis.ckjm.ClassMetrics 24 1 0 0 33 196 6 23
gr.spinellis.ckjm.MetricsFilter 7 1 0 6 30 11 2 5
gr.spinellis.ckjm.ClassVisitor 13 1 0 14 71 34 2 9
gr.spinellis.ckjm.ClassMap 3 1 0 1 21 0 0 2
gr.spinellis.ckjm.PrintPlainResults 2 1 0 2 8 0 1 2
```

**Abbildung 5.4.:** Beispielhafte Konsolenausgabe CKJM (*CKJM*, o. J.)

Im Wesentlichen beschränkt sich dieses Werkzeug – wie nachfolgend angelistet – auf die definierte Kernfunktionalität, das Berechnen objekt-orientierter Metriken nach Chidamber und Kemerer, sowie zwei zusätzliche objekt-orientierte Metriken.

- WMC: Weighted methods per class
- DIT: Depth of Inheritance Tree
- NOC: Number of Children
- CBO: Coupling between object classes
- RFC: Response for a Class
- LCOM: Lack of cohesion in methods
- Ca: Afferent couplings
- NPM: Number of public methods

(*CKJM*, o. J.)

## 5.7. SonarQube

SonarQube ist eine Open-Source Plattform mit der Aufgabe, für kontinuierliche Codequalität zu sorgen und diese laufend zu prüfen und zu steuern. Im Grundsatz vereint SonarQube die beiden vorgestellten Tools CheckStyle und PMD, um durch automatisierte Code-Inspektionen auf Basis statischer Code-Analysen Bugs, Code Smells und Sicherheits-Risiken aufzudecken. SonarQube kann auf einer Vielzahl gängiger Programmiersprachen operieren und setzt dabei auf ein breites Anwendungsfeld. (*SonarQube*, o. J.)

Nachfolgend sind die wesentlichen Kategorien, auf welchen SonarQube operiert, aufgelistet.

- Duplicated Code
- Coding Standards
- Unit Tests
- Code Coverage
- Code Komplexität
- Kommentare
- Bugs
- Security Schwachstellen

(*SonarQube*, o. J.)

Die wichtigsten Vorteile von SonarQube sind automatische Analysen auf Build-Servern, die Integration verschiedenster Build-Management-Tools wie Ant, Maven oder Gradle, sowie die Integration in Continous-Integration Werkzeuge wie Jenkins oder Hudson. Darüber hinaus existieren Plugins für gängige IDE's wie Eclipse, IntelliJ IDEA oder Visual Studio. SonarQube bietet auf diese Weise nicht nur viele Möglichkeiten, für eine gesunde Anwendung zu sorgen, sondern deckt zudem auch neu implementierte Fehlerquellen auf und sorgt durch ein sogenanntes Quality Gate für hohe Transparenz. (*SonarQube*, o. J.)

Abb. 5.5 zeigt eine Darstellung der SonarQube Projekt-Übersicht, wo die wichtigsten Kennzahlen zur Codequalität eines Projektes übersichtlich angezeigt werden.

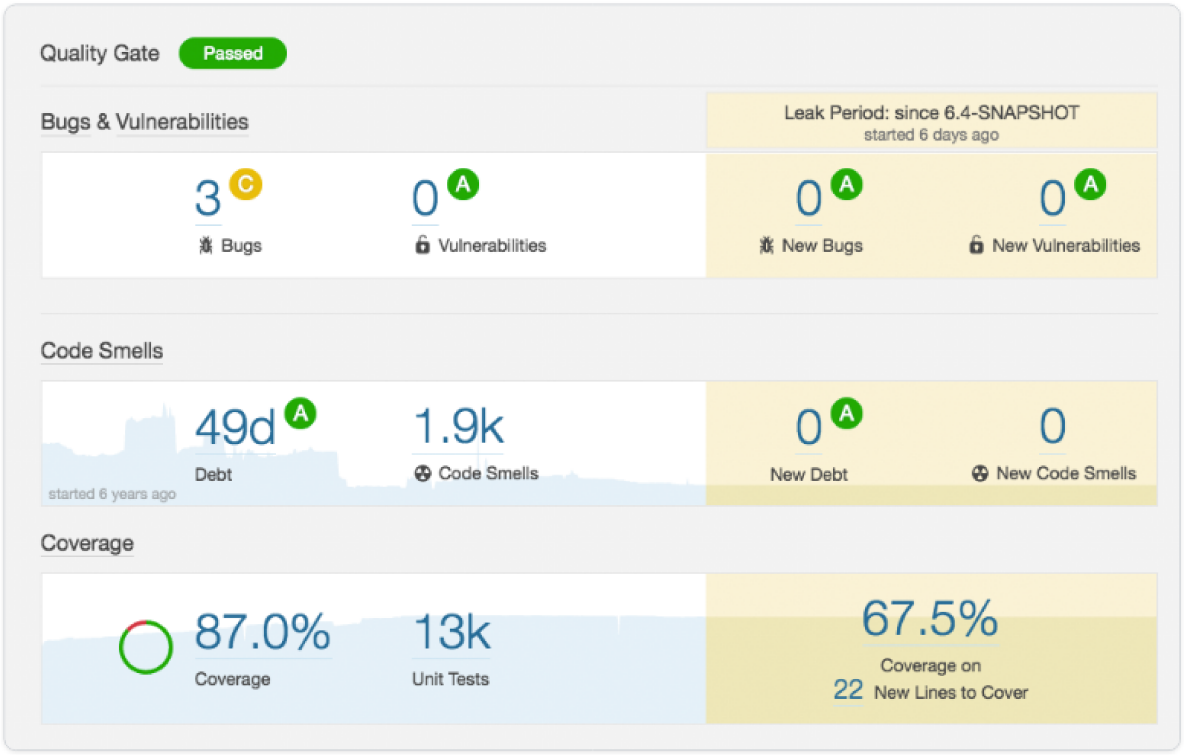



**Abbildung 5.5.:** SonarQube Quality Dashboard (*SonarQube*, o. J.)

## 5.8. Zusammenfassung

In diesem Kapitel wurden nun einige Werkzeuge zur Durchführung sogenannter statischer Analysemethoden und Erhebung von Metriken im Java-Umfeld vorgestellt. Diese Werkzeuge haben teils unterschiedliche Aufgabengebiete und Einsatzzwecke. Während die einen sich vermehrt auf qualitative, deskriptive Faktoren von Sourcecode – wie beispielsweise Naming oder Dokumentation von Quellcode – fokussieren, bestimmen andere Werkzeuge eine Vielzahl an Metriken aus unterschiedlichsten Kategorien und bieten darüber hinaus Grenzwerte, welche bei Überschreitung Fehler markieren. Einige Werkzeuge sind darüber hinaus in der Lage, voll-automatisiert auf eigenen Integrations-Servern zu laufen und so den aktuellen Qualitätsstandard einer Software ständig zu überprüfen. Mit diesem Kapitel endet die Aufbereitung vorhandener Theorien und Werkzeuge und nachfolgende Kapitel widmen sich dem praktischen Teil. Hierfür werden in nachfolgendem Kapitel der Aufbau und die Voraussetzungen des durchgeführten Experiments genauer beschrieben.

# 6. Experiment

## 6.1. Einleitung

Der bisherige theoretische Teil dieser Arbeit widmete sich den wichtigen Aspekten rund um die Thematik Code- und Softwarequalität, wie diese in der Praxis verbessert und mittels verschiedener Codemetriken und Analysewerkzeugen validiert und gemessen werden kann. Im praktischen Teil werden Auswirkungen ausgewählter Methoden zur Verbesserung der Codequalität untersucht und ob dies reale Auswirkungen auf den Wartungs- und Folgeaufwand – und damit auf die Produktivität von Softwareentwicklungs-Teams – hat.

Zu diesem Zweck wurden in Abschnitt 1.2 bereits die Forschungsfrage F1 sowie eine daraus abgeleitete Arbeitshypothese H1 aufgestellt, welche wie folgt lautet: *Je umfangreicher der Einsatz ausgewählter Maßnahmen zur Qualitätssteigerung in der Softwareentwicklung erfolgt, desto mehr kann eine Reduktion von Wartungs- und Folgeaufwänden beobachtet werden.* Diese Hypothese H1 soll nun im Zuge dieses Experiments nach Auswahl bestimmter Codeverschlechterungen, sogenannter Bad Smells, untersucht und mit weiteren Untersuchungsgegenständen verfeinert werden. Dafür wurden zwei weitere Subhypothesen inklusive zugehöriger Nullhypothesen gebildet, welche aufbauend auf die in Abschnitt 6.4 ausgewählten qualitätssteigernden Methoden erstellt wurden.

Subhypothese H1.1: Je umfangreicher der Einsatz von gutem Naming und Javadoc in Sourcecode erfolgt, desto mehr kann eine Reduktion von Wartungs- und Folgeaufwänden beobachtet werden.

Nullhypothese $H1.1_0$: Der Einsatz von gutem Naming und Javadoc in Sourcecode bewirkt nur eine marginale Reduktion von Wartungs-und Folgeaufwänden.

Subhypothese H1.2: Je umfangreicher eine Reduktion von Komplexität, Duplicate Code und Dead Code in Sourcecode erfolgt, desto mehr kann eine Reduktion von Wartungs- und Folgeaufwänden beobachtet werden.

Nullhypothese $H1.2_0$: Eine Reduktion von Komplexität, Duplicate Code und Dead Code in Sourcecode bewirkt nur eine marginale Reduktion von Wartungs-und Folgeaufwänden.

Zu diesem Zweck wurde im praktischem Teil dieser Arbeit ein Experiment zur Eruierung der Auswirkungen von Codequalität anhand praktischer Umsetzung durch Studienteilnehmer durchgeführt. Als Vorbereitung für die Durchführung wurde ein Java-Programm entwickelt, welches danach in unterschiedlichen Qualitätsstufen ausgearbeitet wurde. So existiert einerseits eine Version des Programms, welche durch Anwendung relevanter, im Theorieteil vorgestellter Best Practices und Software Prinzipien eine hohe Codequalität besitzt, und andererseits Versionen, wo ausgewählte Anti-Pattern beziehungsweise Code Smells integriert wurden, um schlechte Codequalität zu simulieren. Die vorhandene Funktionalität ist in allen Versionen des Progeamms ident, hat also für Benutzerinnen und Benutzer der Anwendung keinerlei sichtbare Unterschiede in der Programm-Ausführung, da lediglich der innere Aufbau und die Stuktur des Sourcecode verändert wurde.

Im vorliegenden Experiment wird jedem Teilnehmer eine dieser Programmversionen zur Bearbeitung vorgelegt. Die geforderten Aufgaben entsprechen realen Bedingungen aus der täglichen Entwicklungspraxis – wie dem Suchen und Entfernen von Fehlern, sowie dem Erweitern der Software durch neue Funktionaliät.

In diesem Kapitel wird der Aufbau des Experiments im Detail erörtert sowie Einblick in den Sourcecode der verwendeten Software gegeben, um dem Leser ein Verständnis über die Ausgangslage für dieses Experiment zu vermitteln. Weiters werden die Merkmale und Unterschiede der verschiedenen Anwendungs-Versionen erläutert und die dafür notwendige Integration von Anti Pattern und Code Smells schematisch dargelegt.

## 6.2. Experimentdesign

Um ein derartiges Experiment erfolgreich durchführen zu können, benötigt es eine möglichst große Menge an teilnehmenden Probanden. Da dieses Experiment fortgeschrittene Kenntnisse der Programmiersprache Java erfordert, wodurch das Spektrum möglicher Teilnehmer bereits stark begrenzt wird, wurde darauf geachtet, keine Frameworks oder spezielle Tools einzusetzen, um die mögliche Populationsgröße nicht weiter zu verkleinern.

Zudem wurde in Gesprächen mit potentiellen Teilnehmerinnen und Teilnehmern sehr schnell deutlich, dass nur wenige mehr als 60 Minuten ihrer privaten Zeit für ein solches Experiment zu opfern bereit sind, weshalb eine Limitierung der Experimentdauer je Kandidatin und je Kandidat auf 60 Minuten definiert wurde.

Als weiteres Kriterium für die Auswahl potentieller Probanden wurde eine Berufs- oder Praxiserfahrung mit der Programmiersprache Java von mindestens zwei Jahren festgelegt, um die Ergebnisse der Untersuchung durch fehlende Erfahrung der teilnehmenden Personen nicht zu beeinflussen.

Zusammenfassend werden die Anforderungen an die Teilnehmer dieses Experiments in nachfolgender Kriterienliste aufgezählt.

- Erfahrungen in objekt-orientierter Softwareentwicklung.
- Kenntnisse der Programmiersprache Java.
- Mindestens zwei Jahre Berufs- oder Praxiserfahrung in der Entwicklung Java-basierter Anwendungen.
- Die Berufserfahrung sollte nicht mehrere Jahre in der Vergangenheit liegen.

Wie später in Abschnitt 6.4 detaillierter erläutert wird, wurden insgesamt vier Versionen unterschiedlicher Codequalität einer fiktiven Java-Applikation erstellt, welche jedoch alle in der Kundenbetrachtung die exakt gleiche Funktionalität aufweisen. Um die Validität des Experiments und die Aussagekraft der Ergebnisse zu gewährleisten, wurde darauf geachtet, die am Experiment teilnehmenden Personen möglichst gleichmäßig und zufällig auf diese vier Programmversionen aufzuteilen. Damit soll sichergestellt werden, dass annähernd gleich viele Probanden an den vier verschiedenen Versionen arbeiten und dadurch einer möglichen Verfälschung der Ergebnisse durch ungleiche Gewichtung der Resultate vorgebeugt wird.

In der Durchführung des Experiments wurde jeder Teilnehmer separat eingeladen und die Durchführung im Rahmen der aufgestellten Rahmenbedingungen überprüft. Zudem wurde sichergestellt, dass jede Sitzung die vorgegebene Dauer von einer Stunde nicht überschreitet. Als Hilfestellung bei auftretenden Verständnisproblemen wurde der Autor als alleinige Ansprechperson definiert, die Verwendung von anderen Hilfestellungen wurde nicht gestattet.

Als Vorbereitung für die Teilnehmer und als Aufgabenbeschreibung der Untersuchung wurde eine strukturierte Anleitung erstellt, welche eine Einführung in die vorliegende Anwendung, bestehende Business Regeln und einen Einblick in die technische Umsetzung gibt. Zudem wurden die Anforderungen eines fiktiven Kunden in Form dreier Fehlerbeschreibungen und einer gewünschten Erweiterung angeführt. Die vollständige Aufgabenstellung der Teilnehmerinnen und Teilnehmer ist in Anhang A.5 angeführt.

Der allgemeine Ablauf des Experiments fand für jeden Teilnehmer wie folgt statt:

- Vorbereitung und Einrichtung der Entwicklungsumgebung
- Gründliches Studium der Aufgabenbeschreibung durch den Probanden
- Starten der Zeitmessung und Beginn der Bearbeitung durch den Probanden

- Bearbeitung der einzelnen Aufgaben in frei wählbarer Reihenfolge
- Zeitnahme nach Beendigung der Bearbeitung oder nach Überschreiten des definierten Grenzwertes von einer Stunde
- Überprüfung der Ergebnisse auf Korrektheit durch JUnit-Tests und statische Analyse-Tools

Hat ein Teilnehmer den Grenzwert von einer Stunde überschritten, wurde der Versuch an dieser Stelle beendet und lediglich die Herstellung eines kompilierenden Programmzustands gefordert, um später Teilergebnisse evaluieren zu können. Die genaue Auswertung und Evaluierung des Experiments findet sich in Kapitel 7.

## 6.3. Vorstellung Elektronische Bibliothek

Für das durchgeführte Experiment dieser Arbeit wurde ein Java-Programm mit der Bezeichnung *eLibrary* entwickelt. Hierbei handelt es sich um eine schlichte Anwendung zur elektronischen Beleihung von Büchern sowie zur Abrechnung dieser Leihen. Mit dieser Anwendung ist es möglich, Bücher in der Bibliothek anzulegen, diese an Kunden zu verleihen, retournierte Leihen abzurechnen sowie ein Bonuspunkte-System für Kunden zu führen. Für jede Kategorie eines Buches (Bestseller, Standardroman, Kinderbuch) existiert ein unterschiedliches Berechnungsmodell, sowohl für die Leihgebühr als auch für die Vergabe von Bonuspunkten. Mittels gesammelter Bonuspunkte ist es dem Kunden möglich, einen Rabatt auf zukünftige Buchleihen zu erhalten.

Die entwickelte Anwendung wurde angelehnt an das verwendete Beispielprogramm von Fowler und Beck (1999) in seinem Werk *Refactoring* und wurde durch weitere Funktionen erweitert. Zudem wurden beispielhafte Praxisszenarien in einer Konsolenanwendung integriert, um die Teilnehmer des Experiments bei der Überprüfung der korrekten Anforderungserfüllung zu unterstützen. Darüber hinaus wurde eine Vielzahl an JUnit Testfällen implementiert, um die Ergebnisse der Teilnehmer validieren zu können und festzustellen, ob während der Bearbeitung durch die Teilnehmer andere Probleme (sogenannte Bad Fixes) unbemerkt ihren Weg ins Programm gefunden haben.

Nachfolgend werden die wichtigsten Klassen dieser Anwendung kurz vorgestellt, um einen Einblick in den Aufbau und die Funktionsweise der Anwendung zu geben. Hierbei ist zu erwähnen, dass die nachfolgenden Codeausschnitte die qualitativ hochwertige Version der elektronischen Bibliothek zeigen. Die weiteren Versionen, welche für das Experiment herangezogen wurden, finden sich in vollständiger Ausführung in Anhang A. Ausschnitte der Beispielausführung mittels Konsolenausgabe,

sowie verwendete JUnit Tests werden hier nicht näher erläutert, da sie lediglich eine Hilfestellung für die Auswertung der Ergebnisse darstellen und somit nicht Teil des Experiments waren.

### 6.3.1. Buch.java

Die Klasse *Buch.java* definiert eine abstrakte Basisimplementierung eines Buches. Sie dient als Grundlage für konkrete Implementierungen verschiedener Buchklassen und stellt durch diesen Aufbau einen Best Practice (Polymorphismus, Open/closed principle) dar, da es auf diese Weise sehr einfach ist, neue Kategorien von Büchern in die Anwendung zu integrieren, ohne bestehenden Code ändern zu müssen.

Diese Basisklasse stellt allgemeine Datenstrukturen und Algorithmen bereit, welche nach Bedarf überschrieben werden können. Jedes Buch enthält den Autor, den Titel sowie die Anzahl verfügbarer Exemplare innerhalb der Bibliothek je nach Lagerstand. Somit kann ein Buch nur so lange ausgeliehen werden, wie Exemplare verfügbar sind oder bestehende Exemplare wieder retourniert werden. Weiters verfügt diese Klasse über eine Standardimplementierung der Vergabe von Bonuspunkten. Durch die Methoden *verringereExemplare()* und *erhoeheExemplare()* können Ausleihungen und Retouren von Büchern verwaltet werden.

```
1  package at.codequality.elibrary.business.buch;
2
3  import at.codequality.elibrary.business.↩
       KeineExemplareVerfuegbarException;
4  import at.codequality.elibrary.business.Leihe;
5
6  /**
7   * Abstrakte Klasse eines Buches. Konkrete Sub-Klassen ( z.B. ↩
        Bestseller,
8   * Standardroman, Kinderbuch) erweitern diese Klasse.
9   */
10 public abstract class Buch {
11
12   /** Autor des Buches. */
13   private String autor;
14
15   /** Titel des Buches. */
16   private String titel;
17
18   /** Verfuegbare Exemplare dieses Buches. */
19   private int exemplare;
20
21   /**
```

```
22      * Erstellt ein Buch mit den angegebenen Parametern.
23      *
24      * @param autor
25      * @param titel
26      * @param exemplare
27      */
28     public Buch(String autor, String titel, int exemplare) {
29       super();
30       this.autor = autor;
31       this.titel = titel;
32       this.exemplare = exemplare;
33     }
34
35     /**
36      * Liefert den Autor des Buches.
37      *
38      * @return Autor des Buches
39      */
40     public String getAuthor() {
41       return autor;
42     }
43
44     /**
45      * Liefert den Titel des Buches.
46      *
47      * @return Titel des Buches
48      */
49     public String getTitle() {
50       return titel;
51     }
52
53     /**
54      * Liefert die Anzahl der vorhandenen Exemplare.
55      *
56      * @return Anzahl der vorhandenen Exemplare
57      */
58     public int getExemplare() {
59       return exemplare;
60     }
61
62     /**
63      * Setzt die Anzahl verfuegbarer Exemplare.
64      *
65      * @param exemplare
66      *            Anzahl verfuegbarer Exemplare
67      */
68     public void setExemplare(int exemplare) {
69       this.exemplare = exemplare;
```

```
70  }
71
72  /**
73   * Erhoehen der Exemplare um den Wert 1.
74   */
75  public void erhoeheExemplare() {
76    exemplare++;
77  }
78
79  /**
80   * Verringern der Exemplare um den Wert eins.
81   *
82   * @throws KeineExemplareVerfuegbarException
83   *             Wenn keine Exemplare mehr verfuegbar sind
84   */
85  public void verringereExemplare() throws ↩
       KeineExemplareVerfuegbarException {
86    if (getExemplare() < 1) {
87      throw new KeineExemplareVerfuegbarException();
88    } else {
89      exemplare--;
90    }
91  }
92
93  /**
94   * Liefert die Leihgebuehr fuer das jeweilige Buch. Jedes Buch ↩
       muss hier ihren
95   * eigenen Algorithmus zur Berechnung der Leihgebuehren ↩
       implementieren.
96   *
97   * @param leihe
98   * @return Kosten der Leihe
99   */
100 public abstract double getLeihgebuehr(Leihe leihe);
101
102 /**
103  * Liefert die Bonuspunkte fuer die uebergebene Leihe abhangig ↩
       von den
104  * Leihkosten. Diese Methode kann bei Bedarf von ↩
       unterschiedlichen Buch-Typen
105  * ueberschrieben werden.
106  *
107  * @param leihe
108  * @return Bonuspunkte fuer die uebergebene Leihe
109  */
110 public int getBonusPunkte(Leihe leihe) {
111   int bonusPunkte = 1;
112
```

```
113     if (getLeihgebuehr(leihe) > 6) {
114       bonusPunkte = 3;
115     }
116     return bonusPunkte;
117   }
118 }
```

**Listing 6.1:** Buch.java

### 6.3.2. Standardroman.java

Die Klasse *Standardroman.java* ist eine konkrete Implementierung eines gewöhnlichen Romans. Im Wesentlichen beinhaltet diese Klasse – aufbauend auf der Basisklasse *Buch.java* – die Berechnung der Leihgebühr für einen Standardroman, wohingegen die Berechnung der Bonuspunkte aus der Basisklasse verwendet und nicht überschrieben wird.

```
1  package at.codequality.elibrary.business.buch;
2
3  import at.codequality.elibrary.business.Leihe;
4
5  /**
6   * Konkrete Implementierung eines Standardromanes.
7   */
8  public class Standardroman extends Buch {
9
10   /**
11    * Erstellt einen neuen Standardroman mit den angegebenen ↩
         Parametern.
12    *
13    * @param autor
14    *            Autor des Standardromanes
15    * @param titel
16    *            Titel des Standardromanes
17    * @param exemplare
18    *            Verfuegbare Exemplare
19    */
20   public Standardroman(String autor, String titel, int exemplare)↩
         {
21     super(autor, titel, exemplare);
22   }
23
24   /**
25    * @see Buch#getLeihgebuehr(Leihe)
```

```
26     */
27    @Override
28    public double getLeihgebuehr(Leihe leihe) {
29      double kosten = 3.5;
30
31      if (leihe.getLeihtage() > 6) {
32        kosten += (leihe.getLeihtage() - 6) * 1.5;
33      }
34      return kosten;
35    }
36
37  }
```

**Listing 6.2:** Standardroman.java

### 6.3.3. Bestseller.java

Die Klasse *Bestseller.java* definiert Bücher, welche neu erschienen sind beziehungsweise nach denen die Nachfrage besonders stark ist. Deshalb sind hier die Leihgebühren etwas höher angesetzt. Zudem werden für diesen Buchtyp mehr Bonuspunkte verteilt, als dies bei anderen Buchtypen der Fall ist. Aus diesem Grund implementiert diese Klasse sowohl die Berechnung der Leihgebühr als auch eine überschriebene Variante der Bonuspunkte-Berechnung.

```
1  package at.codequality.elibrary.business.buch;
2
3  import at.codequality.elibrary.business.Leihe;
4
5  /**
6   * Konkrete Implementierung eines Bestseller-Buches.
7   */
8  public class Bestseller extends Buch {
9
10   /**
11    * Erstellt ein neues Bestseller-Buch mit den angegebenen ↩
        Parametern.
12    *
13    * @param autor
14    *            Autor des Bestsellers
15    * @param titel
16    *            Titel des Bestsellers
17    * @param exemplare
18    *            Verfuegbare Exemplare
```

```
19     */
20    public Bestseller(String autor, String titel, int exemplare) {
21      super(autor, titel, exemplare);
22    }
23
24    /**
25     * @see Buch#getLeihgebuehr(Leihe)
26     */
27    @Override
28    public double getLeihgebuehr(Leihe leihe) {
29      double kosten = 5;
30
31      if (leihe.getLeihtage() > 3) {
32        kosten += (leihe.getLeihtage() - 3) * 2.7;
33      }
34      return kosten;
35    }
36
37    /**
38     * @see Buch#getBonusPunkte(Leihe)
39     */
40    @Override
41    public int getBonusPunkte(Leihe leihe) {
42      int bonusPunkte = super.getBonusPunkte(leihe);
43
44      if (leihe.getLeihtage() > 4) {
45        bonusPunkte++;
46      }
47      return bonusPunkte;
48    }
49  }
```

**Listing 6.3:** Bestseller.java

### 6.3.4. Kinderbuch.java

Die Klasse *Kinderbuch.java* bildet Bücher für Kinder ab. Diese besitzen ebenfalls eine eigene Berechnungslogik für die Leihgebühren, wohingegen die Ermittlung der Bonuspunkte von der Basisklasse *Buch.java* vererbt wird.

```
1  package at.codequality.elibrary.business.buch;
2
3  import at.codequality.elibrary.business.Leihe;
4
```

```
5  /**
6   * Konkrete Implementierung eines Kinderbuches.
7   */
8  public class Kinderbuch extends Buch {
9
10   /**
11    * Erstellt ein neues Kinderbuch mit den angegebenen Parametern↩
          .
12    *
13    * @param autor
14    *            Autor des Kinderbuches
15    * @param titel
16    *            Titel des Kindebuches
17    * @param exemplare
18    *            Verfuegbare Exemplare
19    */
20   public Kinderbuch(String autor, String titel, int exemplare) {
21     super(autor, titel, exemplare);
22   }
23
24   /**
25    * @see Buch#getLeihgebuehr(Leihe)
26    */
27   @Override
28   public double getLeihgebuehr(Leihe leihe) {
29     double kosten = 2.5;
30
31     if (leihe.getLeihtage() > 5) {
32       kosten += (leihe.getLeihtage() - 5) * 2.5;
33     }
34     return kosten;
35   }
36 }
```

**Listing 6.4:** Kinderbuch.java

### 6.3.5. Leihe.java

Die Klasse *Leihe.java* ist für die eigentliche Ausleihung von Büchern konzipiert und wird pro ausgeliehenem Buch eines Kunden angelegt. Bei Retournierung des Buches wird in dieser Klasse die Anzahl der Leihtage gesetzt, um später bei der Rechnungsstellung den erforderlichen Betrag, sowie die erhaltenen Bonuspunkte berechnen zu können. Um die Berechnung später durchführen zu können, besitzt jede Leihe eine

Referenz auf das zugrundeliegende Buch und leitet die Berechnung dann an diese weiter, da jedes Buch den Algorithmus für die spezielle Berechnung kennt.

```
1  package at.codequality.elibrary.business;
2
3  import at.codequality.elibrary.business.buch.Buch;
4
5  /**
6   * Diese Klasse bildet eine Buch-Ausleihe ab. Eine Leihe besteht ↩
       immer aus einem
7   * Buch mit einer bestimmten Ausleih-Zeit (Leihtage).
8   */
9  public class Leihe {
10
11   private Buch buch;
12
13   private int leihtage;
14
15   /**
16    * Erzeugt eine Leihe fuer ein bestimmtes Buch.
17    *
18    * @param buch
19    *            Buch, welches ausgeliehen wird
20    */
21   public Leihe(Buch buch) {
22     this.buch = buch;
23   }
24
25   /**
26    * Liefert die Anzahl der Tage, an welchen das Buch ausgeliehen↩
          war.
27    *
28    * @return Anzahl der Leihtage
29    */
30   public int getLeihtage() {
31     return leihtage;
32   }
33
34   /**
35    * Setzt die Anzahl der Tage, an welchen das Buch ausgeliehen ↩
          war.
36    *
37    * @param leihtage
38    *            Leihtage
39    */
40   public void setLeihtage(int leihtage) {
41     this.leihtage = leihtage;
```

```
42    }
43
44    /**
45     * Liefert das geliehene Buch.
46     *
47     * @return Das ausgeliehene Buch
48     */
49    public Buch getBuch() {
50      return buch;
51    }
52
53    /**
54     * Liefert die Leihgebuehr fuer die Leihe des Buches.
55     *
56     * @return Leihgebuehr
57     */
58    public double getLeihgebuehr() {
59      return getBuch().getLeihgebuehr(this);
60    }
61
62    /**
63     * Liefert die Bonuspunkte fuer die Leihe des Buches.
64     *
65     * @return Bonuspunkte
66     */
67    public int getBonusPunkte() {
68      return getBuch().getBonusPunkte(this);
69    }
70
71  }
```

**Listing 6.5:** Leihe.java

### 6.3.6. Kunde.java

Die Klasse *Kunde.java* beschreibt einen Kunden, welcher sich aus der Bibliothek Bücher ausleiht. Jeder Kunde hat einen Namen, eine Liste ausgeliehender Bücher (*Leihe.java*) sowie die aktuelle Anzahl an bereits gesammelten Bonuspunkten. Ein großer Teil der gesamten Anwendungslogik findet sich in dieser Klasse. So werden hier unter anderem Leihen verwaltet, Bonuspunkte zugewiesen und verbraucht und die Berechnung des gesamten Leihbetrages eines Kunden in der Methode *getGesamtLeihgebuehr()* ermittelt, wo retournierte Buchleihen verrechnet und Bonuspunkte durch Rabatte gutgeschrieben werden. Zudem wird in der Berechnung der Leihgebühren

ab einem gewissen Rechnungsbetrag das günstigste Buch ermittelt und vom Rechnungsbetrag abgezogen.

```
1  package at.codequality.elibrary.business;
2
3  import java.util.ArrayList;
4  import java.util.List;
5
6  /**
7   * Die Klasse Kunde repraesentiert einen Kunden aus dem echten ↩
       Leben. Ein Kunde
8   * hat einen Namen und kann mehrere Ausleihungen haben. Zudem ↩
       sammeln Kunden
9   * Bonuspunkte fuer ausgeliehene Buecher.
10  */
11 public class Kunde {
12
13   /** Name des Kunden. */
14   private String name;
15
16   /** Gesammelte Bonuspunkte des Kunden. */
17   private int bonuspunkte;
18
19   /** Liste der Leihen des Kunden. */
20   private List<Leihe> leihen = new ArrayList<>();
21
22   /**
23    * Erzeugt einen neuen Kunden mittels angegebenem Namen.
24    *
25    * @param name
26    *            Name des Kunden
27    */
28   public Kunde(String name) {
29     this.name = name;
30   }
31
32   /**
33    * Liefert den Namen des Kunden.
34    *
35    * @return Name des Kunden
36    */
37   public String getName() {
38     return name;
39   }
40
41   /**
```

```
42    * Liefert die Bonuspunkte, welche bis zu diesem Zeitpunkt ↩
          gesammelt wurden.
43    *
44    * @return Gesammelte Bonuspunkte
45    */
46   public int getBonuspunkte() {
47     return bonuspunkte;
48   }
49
50   /**
51    * Liefert die Liste der Ausleihungen des Kunden.
52    *
53    * @return Leihen des Kunden
54    */
55   public List<Leihe> getLeihen() {
56     return leihen;
57   }
58
59   /**
60    * Bonuspunkte beim retournieren ausgeliehener Buecher dem ↩
          Kunden zuweisen.
61    * Hierfuer werden die Bonuspunkte der einzelnen Leihen ↩
          akkumuliert.
62    *
63    * @param retournierteBuecher
64    */
65   public void bonuspunkteZuweisen(List<Leihe> retournierteBuecher↩
       ) {
66     for (Leihe leihe : retournierteBuecher) {
67       bonuspunkte += leihe.getBonusPunkte();
68     }
69   }
70
71   /**
72    * Verringert die gesamten Bonuspunkte des Kunden um die ↩
          angegebene Anzahl. Das
73    * ist zb. der Fall, wenn ein Kunde Bonuspunkte fuer ↩
          Ermaessigungen verbraucht.
74    *
75    * @param punkte
76    *            Punkte, welche von Punktekonto abgezogen werden ↩
          sollen.
77    */
78   public void bonuspunkteAbziehen(int punkte) {
79     if (!(punkte > this.bonuspunkte)) {
80       bonuspunkte -= punkte;
81     }
82   }
```

```
83
84  /**
85   * Liefert die retournierten Buecher eines Kunden. Hierfuer ↩
       werden Buecher
86   * gezaehlt, welche eine Ausleihdauer > 0 aufweisen. Die ↩
       Ausleihdauer wird beim
87   * Zurueckbringen von Buechern gesetzt.
88   *
89   * @return Retournierte Buecher
90   */
91  public List<Leihe> getRetournierteBuecher() {
92    ArrayList<Leihe> retournierteBuecher = new ArrayList<>();
93    for (Leihe leihe : getLeihen()) {
94      if (leihe.getLeihtage() > 0) {
95        retournierteBuecher.add(leihe);
96      }
97    }
98    return retournierteBuecher;
99  }
100
101 /**
102  * Liefert die Leihgebuehr des billigsten, ausgeliehenen Buches↩
       .
103  *
104  * @return Leihgebuehr des billigsten Buches
105  */
106 public double getKostenDesBilligstenBuches() {
107   double minKosten = 0;
108   for (Leihe leihe : getLeihen()) {
109     if (minKosten == 0) {
110       minKosten = leihe.getLeihgebuehr();
111     }
112     if (leihe.getLeihgebuehr() < minKosten) {
113       minKosten = leihe.getLeihgebuehr();
114     }
115   }
116   return minKosten;
117 }
118
119 /**
120  * Liefert die gesamte Leihgebuehr fuer alle zurueckgebrachten ↩
       Buecher. Zudem
121  * werden entsprechende Bonuspunkte verbraucht/gesetzt und die ↩
       Buecherleihen vom
122  * Kunden entfernt.
123  *
124  * @return Gesamte Leihgebuehr
125  */
```

```
126   public double getGesamtLeihgebuehr() {
127     List<Leihe> retournierteBuecher = getRetournierteBuecher();
128
129     double leihgebuehr = 0;
130     for (Leihe leihe : retournierteBuecher) {
131       leihgebuehr += leihe.getLeihgebuehr();
132     }
133
134     while (getBonuspunkte() > 9 && leihgebuehr > 1) {
135       leihgebuehr -= 1;
136       bonuspunkteAbziehen(10);
137     }
138
139     if (leihgebuehr > 20) {
140       leihgebuehr -= getKostenDesBilligstenBuches();
141     }
142
143     bonuspunkteZuweisen(retournierteBuecher);
144     getLeihen().removeAll(retournierteBuecher);
145
146     return leihgebuehr;
147   }
148 }
```

**Listing 6.6:** Kunde.java

### 6.3.7. KeineExemplareVerfuegbarException.java

Da jedes Buch eine gewisse Menge an verfügbaren Exemplaren aufweist, existiert in der Software eine Überprüfung, ob bei einer versuchten Ausleihung entsprechende Exemplare vorhanden sind. Dies geschieht in der Methode *verringereExemplare()* der Klasse *Buch.java*, wo in einem solchen Fall die Exception *KeineExemplareVerfuegbarException.java* geworfen wird. Diese Exception dient dazu, aufrufenden Methoden zu signalisieren, dass der Fall eintreten könnte, keine Exemplare mehr verfügbar zu haben, und darauf reagieren zu können.

```
1 package at.codequality.elibrary.business;
2
3 /**
4  * Diese Klasse repraesentiert eine Exception, welche geworfen ↩
      wird, wenn ein
5  * Buch versucht wird auszuborgen, von welchem es keine ↩
      verfuegbaren Exemplare
```

```
6   * mehr gibt.
7   */
8  public class KeineExemplareVerfuegbarException extends Exception ↩
       {
9
10 }
```

**Listing 6.7:** KeineExemplareVerfuegbarException.java

### 6.3.8. BibliothekManager.java

Der Bibliotheksmanager ist die zentrale Drehscheibe der elektronischen Bibliothek. Diese Klasse bietet die Schnittstelle zwischen User Interface und Geschäftslogik und leitet Anfragen an Instanzen der Klassen *Buch.java*, *Leihe.java* sowie *Kunde.java* weiter. Zu diesem Zweck werden in dieser Klasse alle Bücher und Kunden mit deren Leihen gespeichert. Da es insgesamt nur eine Instanz dieser Klasse geben kann wurde diese Klasse nach dem Singleton-Pattern implementiert.

```
1  package at.codequality.elibrary.business;
2
3  import java.util.ArrayList;
4  import java.util.List;
5
6  import at.codequality.elibrary.business.buch.Buch;
7
8  /**
9   * Bibliothek Manager, welcher als Zentrale der ↩
        Biliotheksverwaltung dient und
10  * jegliche Kunden und Buecher verwaltet.
11  */
12 public class BibliothekManager {
13
14   /** Manager Instanz fuer Singleton */
15   private static BibliothekManager manager;
16
17   /** Liste aller Buecher */
18   private List<Buch> buecherListe = new ArrayList<>();
19
20   /** Liste aller Kunden */
21   private List<Kunde> kundenListe = new ArrayList<>();
22
23   /**
24    * private Konstruktor fuer Singleton
```

```
25    */
26   private BibliothekManager() {
27
28   }
29
30   /**
31    * Singleton Implementierung.
32    *
33    * @return Instanz eines BibliothekManagers.
34    */
35   public static BibliothekManager getInstance() {
36     if (manager == null) {
37       manager = new BibliothekManager();
38     }
39     return manager;
40   }
41
42   /**
43    * Kunde dem Manager hinzufuegen.
44    *
45    * @param kunde
46    */
47   public void kundeHinzufuegen(Kunde kunde) {
48     kundenListe.add(kunde);
49   }
50
51   /**
52    * Buch dem Manager hinzufuegen.
53    *
54    * @param buch
55    */
56   public void buchHinzufuegen(Buch buch) {
57     buecherListe.add(buch);
58   }
59
60   /**
61    * Retourniert alle Kunden.
62    *
63    * @return Alle Kunden.
64    */
65   public List<Kunde> getAllKunden() {
66     return kundenListe;
67   }
68
69   /**
70    * Retourniert alle Buecher.
71    *
72    * @return Alle Buecher.
```

```
73    */
74   public List<Buch> getAllBuecher() {
75     return buecherListe;
76   }
77
78   /**
79    * Buch ausleihen. Methode erzeugt eine neue Leihe fuer das ↩
         Buch und fuegt diese
80    * dem Kunden zu.
81    *
82    * @param kunde
83    * @param buch
84    * @throws KeineExemplareVerfuegbarException
85    */
86   public void buchAusleihen(Kunde kunde, Buch buch) throws ↩
       KeineExemplareVerfuegbarException {
87     buch.verringereExemplare();
88     Leihe leihe = new Leihe(buch);
89     kunde.getLeihen().add(leihe);
90   }
91
92   /**
93    * Buch zurueckbringen. Methode vermerkt in der Leihe die ↩
         geliehenen Tage und
94    * erhoeht die Anzahl verfuegbarer Exemplare, da dieses Buch ↩
         nun wieder zu
95    * weiterer Ausleihe verfuegbar ist.
96    *
97    * @param leihe
98    * @param leihtage
99    */
100  public void buchZurueckgeben(Leihe leihe, int leihtage) {
101    leihe.setLeihtage(leihtage);
102    leihe.getBuch().erhoeheExemplare();
103  }
104
105  /**
106   * Liefert die gesamte Leihgebuehr des Kunden, nachdem dieser ↩
        einige Buecher
107   * zurueckgebracht hat. Zudem werden entsprechende Bonuspunkte
108   * verbraucht/gesetzt und die Buecherleihen vom Kunden entfernt↩
        .
109   *
110   * @param kunde
111   * @return Gesamte Leihgebuehr des Kunden
112   */
113  public double getGesamtLeihgebuehr(Kunde kunde) {
114    return kunde.getGesamtLeihgebuehr();
```

```
115     }
116
117 }
```

**Listing 6.8:** BibliothekManager.java

## 6.4. Systematische Verschlechterung des Sourcecode

Nachdem für die in Abschnitt 6.3 vorgestellte Beispielanwendung des durchgeführten Experiments darauf geachtet wurde, möglichst saubere Codequalität durch Anwenden zahlreicher Refactoringmaßnahmen zu erreichen, wurden an dieser Anwendung ausgewählte Qualitätsverschlechterungen des zugrundeliegenden Sourcecode vorgenommen und darauf drei weitere Versionen gebildet. Es gilt hier noch einmal zu betonen, dass die kohärente Funktionalität aller Versionen komplett ident ist und sich diese nur durch die innere Codequalität stark unterscheiden. Diese Versionen sollten später in der Evaluierung der Ergebnisse mit der qualitativ hochwertigen Version der Anwendung verglichen werden, um mögliche Auswirkungen auf Wartbarkeit und Erweiterbarkeit festzustellen. Zu diesem Zweck wurde bestehende Literatur auf besonders relevante Codesmells und Anti-Pattern durchsucht und daraus die beschriebenen drei verschlechterten Versionen der eLibrary je nach Einsatzmöglichkeit und Sinnhaftigkeit gebildet.

Als theoretische Grundlage für die Auswahl eingesetzter Codesmells wurden Auflistungen definierter Codesmells von Fowler und Beck (1999) und zur Priorisierung zusätzlich Beiträge aus Yamashita und Moonen (2012) und Anda (2007) verwendet. Diese beschäftigen sich mit relevanten Wartungsfaktoren und bilden daraus zugehörige Codesmells ab. Weiters betonen sie, dass einige Smells nicht automatisiert überprüft (beispielsweise Naming, Design) und andere hingegen sehr gut getestet werden können (u.a. Komplexität). Aus diesen Vorschlägen wurden in der Folge einige Faktoren ausgewählt, wobei darauf geachtet wurde, dass diese sich auch in einer realistischen Form in diesem doch vom Umfang begrenzten Beispiel des Experiments integrieren lassen. So wurde beispielsweise darauf verzichtet, Wartungsfaktoren wie starke Kopplung verschiedener Klassen und andere Faktoren in das Experiment mitaufzunehmen, da diese erst bei einem weit höheren Programmumfang zum Tragen kommen.

Letztlich wurden folgende Codesmells ausgewählt, da diese nach aktueller Einschätzung auch in den Umfang des Experiments und der dazugehörigen Anwendung adäquat einsetzbar beziehungsweise teilweise kombinierbar waren.

- Naming und Javadoc

- Dead Code
- Complexity
- Duplicated Code
- Long Method

Bei der Anwendung der ausgewählten Verschlechterungsmaßnahmen wurde systematisch Schritt für Schritt vorgegangen und stets die bestehende Funktionalität mit JUnit Tests geprüft, um sicherzustellen, dass die Funktionalität ident blieb, da es aus Erfahrung und Literaturbeiträgen, wie auch Fowler und Beck (1999) anführen, bei Refactoring-Maßnahmen oft passiert, dass unbewusst neue Fehler entstehen.

Insgesamt wurden– wie bereits beschrieben – zusätzlich zu der originalen Version mit guter Codequalität, drei Versionen schlechterer Codequalität erstellt, wobei diese – wie nachfolgend beschrieben – mit unterschiedlichen Smells versehen wurden. Die Reihung der einzelnen Programmversionen wurde zufällig bestimmt, um mögliche Rückschlüsse der Teilnehmer – und dadurch eine etwaige Beeinflussung – aufgrund der Programmversion zu vermeiden.

- Version 1: Naming und fehlendes Javadoc
- Version 2: Good Quality
- Version 3: Komplexität, Dead und Duplicate Code und dadurch einhergehend Long Method
- Version 4: Generell schlechte Qualität durch verwenden aller angeführten Codesmells

Alle vier Versionen werden zugunsten der Übersichtlichkeit nur in Ausschnitten angeführt, können jedoch in voller Länge in Anhang A nachgeschlagen werden.

Nachfolgend werden alle im Experiment verwendeten Codesmells erläutert und beispielhaft beschrieben.

### 6.4.1. Naming und Javadoc

Bei dem Smell Naming handelt es sich um schlechte Benennung von Klassen, Methoden, Variablen und anderen Konstrukten. Dieser Smell ist ein offensichtliches Qualitätskriterium und im Grunde auch leicht zu beheben, dennoch findet sich in vielen

Anwendungen schlechtes Naming. Dies entsteht einerseits durch die Vergabe von kurzen, oft nichtssagenden Namen für temporäre Variablen – wie beispielsweise *i* oder *tmp*. Zusätzlich stellt der passende Name einer Methode oder auch einer Variablen eine häufig herausfordernde Aufgabe dar, da der Name möglichst präzise die Funktion der Methode oder Variable benennen soll, was nicht immer einfach ist. Nicht selten kommt es vor, dass Namen in abgekürzter Form verwendet werden, was ebenfalls zu späteren Verständnisproblemen führt. Besonders kritisch wird es, wenn Namen auf eine andere Funktionalität hinweisen, als in einer Methode tatsächlich vorherrscht, wodurch in zukünftigen Entwicklungen sehr schnell Fehlern entstehen können.

Als Beispiel wird die verwendete Methode *getKostenDesBilligstenBuches()* aus der Klasse *Kunde.java* im Vergleich dargestellt. In Listing 6.9 wird die schöne Variante mit gutem Naming der Methode zur Berechnung des billigsten Buches gezeigt.

```
1  public double getKostenDesBilligstenBuches() {
2    double minKosten = 0;
3    for (Leihe leihe : getLeihen()) {
4      if (minKosten == 0) {
5        minKosten = leihe.getLeihgebuehr();
6      }
7      if (leihe.getLeihgebuehr() < minKosten) {
8        minKosten = leihe.getLeihgebuehr();
9      }
10   }
11   return minKosten;
12 }
```

**Listing 6.9:** Kosten des billigsten Buches eines Kunden

Listing 6.10 zeigt dieselbe Methode, jedoch mit deutlich unpräziserem und dadurch schlechterem Naming. Bereits im Methodennamen ist nicht wirklich klar, was das Ziel der Methode ist, denn es wird nicht – wie impliziert – ein Buch retourniert, sondern eben die Kosten des billigsten Buches retourniert. Ebenso ist der Name *FuerVerringerung* nicht angemessen, denn es ist irrelevant, was der Aufrufer dieser Methode mit dem Ergebnis macht, dennoch wird hier die spätere Logik des Buchkostenerlasses angedeutet.

```
1  public double getBuchFuerVerringerung() {
2    double kosten = 0;
3    for (Leihe l : getBue()) {
4      if (kosten == 0) {
5        kosten = l.rechnen();
6      }
```

```
7       if (l.rechnen() < kosten) {
8         kosten = l.rechnen();
9       }
10    }
11    return kosten;
12  }
```

**Listing 6.10:** Kosten des billigsten Buches eines Kunden - Bad Naming

Weiters ist innerhalb der Methode bereits klar zu erkennen, dass man durch schlechtes Naming den Leser von Sourcecode stark irreführen kann, indem hier beispielsweise die Methode *getLeihgebuehr()* aus der Klasse *Leihe.java* mit der Bezeichnung *rechnen()* ersetzt wurde. Besonders gravierend sind falsche Andeutungen auf Objekttypen, wie hier der Methodeaufruf *getBuc()* zeigt. Denn vergleicht man den Code mit der ursprünglichen Version, steckt hinter dieser Methode nicht, wie angedeutet, das Erhalten eines Buches, sondern die Liste der Leihen eines Kunden.

Auf diese Weise wurde für die Version mit dem Smell *Naming* eine Vielzahl an Methoden und Variablennamen verändert (>85%), um diesen Smell stark zu repräsentieren.

In der originalen Version der Anwendung wurde auf allen Klassen und Methoden eine Dokumentation der Programmierschnittstelle (API) mittels Javadoc erstellt, um die verschiedenen Konstrukte und deren Aufgaben genau zu beschreiben. Bei Javadoc handelt es sich um ein Dokumentationswerkzeug für Java-Code und es ist das Standardtool, wenn es um Dokumentation im Java-Umfeld geht. (*Oracle Javadoc Tool Website*, o. J.)

```
1   /**
2    * Verringert die gesamten Bonuspunkte des Kunden um die ↩
        angegebene Anzahl. Das
3    * ist zb. der Fall, wenn ein Kunde Bonuspunkte fuer ↩
        Ermaessigungen verbraucht.
4    *
5    * @param punkte
6    *            Punkte, welche von Punktekonto abgezogen werden ↩
        sollen.
7    */
8   public void bonuspunkteAbziehen(int punkte) {
9     if (!(punkte > this.bonuspunkte)) {
10      bonuspunkte -= punkte;
11    }
12  }
```

**Listing 6.11:** bonuspunkteAbziehen() mit Javadoc

In Listing 6.11 ist beispielhaft die Methode *bonuspunkteAbziehen()* der Klasse *Kunde.java* mit vorhandenem Javadoc zu sehen, wohingegen in Listing 6.12 die selbe Methode ohne Javadoc dargestellt wird. In diesem Fall ist der Entwickler oder die Entwicklerin auf sich gestellt und muss den Inhalt des vorliegenden Konstrukts erst selbst nachvollziehen. Gerade in Kombination mit schlechtem Naming wird es umso schwieriger, die zugrundeliegende Programmlogik voll zu erfassen.

```
1  public void bonuspunkteAbziehen(int punkte) {
2    if (!(punkte > this.bonuspunkte)) {
3      bonuspunkte -= punkte;
4    }
5  }
```

**Listing 6.12:** bonuspunkteAbziehen() ohne Javadoc

Jedoch sollte Javadoc nicht mit dem Einsatz regulärer Inline-Kommentare verwechselt werden, denn einfache Kommentare inmitten von Sourcecode stellen laut Fowler und Beck (1999) ebenfalls einen Codesmell dar, da Kommentare sehr oft verwendet werden, um schlecht lesbare Algorithmen zu beschreiben, anstatt diese so zu strukturieren, dass sie einfach verständlich sind. Zudem sind Kommentare sehr schnell veraltet, wenn sich beispielsweise der beschriebene Algorithmus ändert. Gerade veraltete Kommentare tragen im Laufe der Zeit noch stärker zu Konfusionen beim Lesen von Sourcecode bei und sollten daher vermieden werden.

Für die im Experiment verwendete Version mit schlechtem Naming wurden daher sämtliche Javadoc Dokumentationen aus der originalen Version entfernt, um Auswirkungen dieses Smells in der Praxis zu erproben.

### 6.4.2. Dead Code

Unter Dead Code versteht man in der Softwareentwicklung Code, welcher nicht (mehr) ausgeführt wird und daher entfernt werden könnte. Oftmals entsteht Dead Code durch einen schleichenden Prozess, wenn beispielsweise neue Funktionen geschrieben werden, welche vorhandenen Code ersetzen, dieser jedoch nicht entfernt wird. Hierbei kann es sich um einzelne Zeilen, Methoden und oft sogar um ganze Klassen handeln. Durch die Zunahme von unbenutzem Code wird auch immer mehr die Lesbarkeit und Verständlichkeit des gesamten Konstruktes verschlechtert, da bestehende Algorithmen immer aufgeblähter und undurchsichtiger werden. Zudem erhöht Dead Code den Aufwand, sich in ein neues Programm einzulesen und es zu verstehen, da viel Zeit damit verschwendet wird, Code zu lesen, welcher gar keine Anwendung hat.

Um dieses Szenario zu simulieren wurde in einigen Methoden der Versionen 2 und 3 Dead Code eingesetzt, wie in Listing 6.13 beispielhaft mit der Methode der Klasse *Kunde.java* dargestellt wird.

```
1  public double getGesamtLeihgebuehr() {
2    List<Leihe> retournierteBuecher = getRetournierteBuecher();
3
4    double leihgebuehr = 0;
5    for (Leihe leihe : retournierteBuecher) {
6      leihgebuehr += leihe.getLeihgebuehr();
7    }
8
9    double tmpGebuehr = leihgebuehr;
10   boolean fuerBonusVerwenden = false;
11
12   for (Leihe leihe2 : retournierteBuecher) {
13     if (leihe2.getLeihgebuehr() > 4 && leihe2.getLeihtage() > 2↩
           && leihe2.getBuch().getTyp() == 2) {
14       fuerBonusVerwenden = true;
15     }
16     if (leihe2.getLeihgebuehr() > 1 && leihe2.getLeihtage() > 3↩
           && leihe2.getBuch().getTyp() == 1) {
17       fuerBonusVerwenden = true;
18     }
19     if (leihe2.getLeihgebuehr() > 5 && leihe2.getLeihtage() > 2↩
           && leihe2.getBuch().getTyp() == 4) {
20       fuerBonusVerwenden = true;
21     }
22     if (fuerBonusVerwenden) {
23       int punkteFuerBonuszuweisung = leihe2.getBonusPunkte();
24       BonusStufe bonusStufe = this.getBonusVerwaltung().↩
             getBonusLevel();
25       if (bonusStufe == BonusStufe.NEU) {
26         punkteFuerBonuszuweisung = punkteFuerBonuszuweisung / ↩
             3;
27       }
28       if (bonusStufe == BonusStufe.LEVEL1 || bonusStufe == ↩
           BonusStufe.LEVEL2
29           || bonusStufe == BonusStufe.LEVEL3) {
30         punkteFuerBonuszuweisung = punkteFuerBonuszuweisung / ↩
             1;
31       }
32       if (bonusStufe == BonusStufe.POWERUSER) {
33         punkteFuerBonuszuweisung = (int) Math.round(↩
             punkteFuerBonuszuweisung * 1.5);
34       }
35       if (bonusStufe == BonusStufe.PREMIUM) {
```

```
36          punkteFuerBonuszuweisung = (int) Math.round(↩
               punkteFuerBonuszuweisung * 2.5);
37        }
38        bonusVerwaltung.punkteErhoehen(punkteFuerBonuszuweisung);
39      }
40
41      int bonuspunkte = getBonusVerwaltung().getGesamtBonuspunkte↩
           ();
42
43      if (bonuspunkte < 40) {
44        // zu wenig Punkte gesammelt
45      }
46      if (bonuspunkte > 39 && bonuspunkte < 80) {
47        float abzugBetrag = (float) tmpGebuehr / 10;
48      }
49      if (bonuspunkte > 80) {
50        float abzugBetrag = (float) tmpGebuehr / 20;
51      }
52      fuerBonusVerwenden = false;
53    }
54
55    while (getBonuspunkte() > 9 && leihgebuehr > 1) {
56      leihgebuehr -= 1;
57      bonuspunkteAbziehen(10);
58      bonusVerbrauchen(10);
59    }
60
61    if (leihgebuehr > 20) {
62      leihgebuehr -= getKostenDesBilligstenBuches();
63      bonusVerbrauchen();
64    }
65
66    bonuspunkteZuweisen(retournierteBuecher);
67    getLeihen().removeAll(retournierteBuecher);
68
69    return leihgebuehr;
70  }
71
72 }
```

**Listing 6.13:** Berechnung Leihgebühr mit Dead Code

Wie in Listing 6.13 zu sehen ist, wurde hier von Zeile 11 bis 55 Dead Code simuliert, welcher darauf hindeuted, dass es früher eine andere Berechnung von Bonuspunkten gegeben hat, welche jedoch nicht aus dem System entfernt wurde. Bei genauerem Betrachten fällt auf, dass die besagten Zeilen keine relevante Verwendung innerhalb der Methode haben, da die resultierende Variable *abzugBetrag* später keine Verwen-

dung mehr findet. Aus diesem Grund könnten besagte Zeilen ohne weiteres entfernt werden, ohne die Funktionalität dieser Methode zu beeinflussen.

Bei genauerem Hinsehen fällt zudem auf, dass eine weitere Klasse *BonusStufe.java* verwendet wird, welche die verschiedenen Bonusstufen der alten Bonusberechnung repräsentiert. Auch diese Klasse ist Dead Code, da sie nur von anderem, nicht benutztem Code verwendet wird, und könnte entfernt werden.

### 6.4.3. Complexity

Der Begriff Komplexität ist in der Softwareentwicklung ein weit verbreiteter Begriff, welcher durch verschiedene – bereits in Kapitel 4 und Kapitel 5 vorgestellten – Metriken und Analysen verwendet wird. In der einfachsten Form bezieht sich komplexer Code auf eine Vielzahl verschiedener Entscheidungswege innerhalb einer Methode, wie beispielsweise Halstead (1979) und McCabe (1976) in ihren Definitionen von Codekomplexität anführen. Um diesen Aspekt in Version 3 der Beispielanwendung zu integrieren, wurde durch den Einsatz von Dead Code, Duplicate Code sowie den zusätzlichen Verzicht von Polymorphie – und stattdessen häufiger Einsatz von Verzweigungen und Switch-Statements – versucht, etwas Komplexität zu simulieren.

Dafür kann die Methode *getBonusPunkte()* der Klasse *Buch.java* aus Listing 6.14 beispielhaft zur Veranschaulichung von erhöhter Komplexität herangezogen werden.

```
1  public int getBonusPunkte(Leihe leihe) {
2    double leihK = 1;
3    if (buchTyp == 1) {
4      leihK = 5;
5      if (leihe.getLeihtage() > 3) {
6        leihK += (leihe.getLeihtage() - 3) * 2.7;
7      }
8    } else if (buchTyp == 2) {
9      leihK = 3.5;
10     if (leihe.getLeihtage() > 6) {
11       leihK += (leihe.getLeihtage() - 6) * 1.5;
12     }
13   } else {
14     leihK = 0;
15   }
16
17   if (buchTyp == 1) {
18     int punkte = 1;
19     if (leihK > 6) {
20       punkte = 3;
```

```
21        }
22        if (leihe.getLeihtage() > 4) {
23          punkte++;
24        }
25        return punkte;
26      }
27      if (getLeihgebuehr(leihe) > 6) {
28        return 3;
29      }
30      return 1;
31    }
```

**Listing 6.14:** Komplexer Code durch viele Verzweigungen

Wie in Listing 6.14 zu sehen ist, existieren in dieser Methode zahlreiche Verzweigungen aufgrund verschiedener Buchtypen und anderer Anforderungen des Algorithmus. Dadurch ist die Verständlichkeit dieser Methode beeinträchtigt, da man für die Vielzahl an Möglichkeiten nicht einfach nachvollziehen kann, welches Resultat diese Methode retourniert. Für den Fall, dass weitere Typen von Büchern in die Software integriert werden sollten, würde die Komplexität weiter steigen.

### 6.4.4. Duplicate Code

Der Smell Duplicate Code beschreibt Sourcecode, welche in identischer Form mehrfach in einem Programm vorkommt und ist laut Fowler und Beck (1999) einer der schlimmsten Codesmells, da durch Codeduplizierung sehr leicht Fehler bei zukünftigen Bearbeitungen passieren können. Das Problem ist hier meist, dass bei Änderung eines duplizierten Codeteiles sehr schnell übersehen wird, diese Änderung auch in den weiteren Duplikaten nachzuziehen, falls überhaupt Kenntnis über weitere Duplikate besteht, und auf diese Weise kommen in der Praxis häufig Fehlzustände und Inkonsistenzen in eine Software. Dieser Umstand kann sehr einfach durch Refactoring der duplizierten Codeteile in eine eigene Methode behoben werden, denn es muss immer das Ziel von Software sein, eine Funktionalität an genau einer Stelle ändern zu können.

In Listing 6.15 ist beispielhaft das Problem von Codeduplikaten zu sehen, indem idente kontitionale Zuweisungen von Bonusstufen in zwei unterschiedlichen Methoden verwendet werden.

```
1  public void punkteErhoehen(int punkte) {
2    gesamtBonuspunkte += punkte;
3
```

```
4     if (gesamtBonuspunkte < 10) {
5       bonusStufe = BonusStufe.NEU;
6     }
7     if (gesamtBonuspunkte > 9 && gesamtBonuspunkte < 20) {
8       bonusStufe = BonusStufe.LEVEL1;
9     }
10    if (gesamtBonuspunkte > 19 && gesamtBonuspunkte < 30) {
11      bonusStufe = BonusStufe.LEVEL2;
12    }
13    if (gesamtBonuspunkte > 29 && gesamtBonuspunkte < 30) {
14      bonusStufe = BonusStufe.LEVEL3;
15    }
16    if (gesamtBonuspunkte > 39 && gesamtBonuspunkte < 40) {
17      bonusStufe = BonusStufe.LEVEL4;
18    }
19    if (gesamtBonuspunkte > 39 && gesamtBonuspunkte < 80) {
20      bonusStufe = BonusStufe.POWERUSER;
21    }
22    if (gesamtBonuspunkte > 79) {
23      bonusStufe = BonusStufe.PREMIUM;
24    }
25  }
26
27  public void stufeAktualisieren() {
28    if (gesamtBonuspunkte < 10) {
29      bonusStufe = BonusStufe.NEU;
30    }
31    if (gesamtBonuspunkte > 9 && gesamtBonuspunkte < 20) {
32      bonusStufe = BonusStufe.LEVEL1;
33    }
34    if (gesamtBonuspunkte > 19 && gesamtBonuspunkte < 30) {
35      bonusStufe = BonusStufe.LEVEL2;
36    }
37    if (gesamtBonuspunkte > 29 && gesamtBonuspunkte < 30) {
38      bonusStufe = BonusStufe.LEVEL3;
39    }
40    if (gesamtBonuspunkte > 39 && gesamtBonuspunkte < 40) {
41      bonusStufe = BonusStufe.LEVEL4;
42    }
43    if (gesamtBonuspunkte > 39 && gesamtBonuspunkte < 80) {
44      bonusStufe = BonusStufe.POWERUSER;
45    }
46    if (gesamtBonuspunkte > 79) {
47      bonusStufe = BonusStufe.PREMIUM;
48    }
49  }
```

**Listing 6.15:** Duplicate Code in der Bonuspunkteverwaltung

Sowohl in der Methode *punkteErhoehen()* als auch in *stufeAktualisieren()* findet sich identer Sourcecode mit zweifacher Ausführung des exakt gleichen Algorithmus. Lediglich die erste Zeile der Methode *punkteErhoehen()* unterscheidet sich. Um die Problematik noch besser nachvollziehen zu können, wurde in beiden Methoden ein kleiner Fehler in den Zeilen 13 und 37 simuliert, wo der Schritt von jeweils 10 Bonuspunkten unterbrochen wurde, wodurch die dritte Bonusstufe nie erreicht werden kann. Würde man nun diesen Fehler suchen und entdecken, so müsste dieser in beiden Methoden behoben werden. Stattdessen könnte man als Refactoring-Maßnahme die Methode *stufeAktualisieren()* direkt aus der Methode *punkteErhoehen()* aufrufen und so das Codeduplikat eliminieren.

### 6.4.5. Long Method

Der Codesmell Long Method beschreibt sehr lange Methoden im Sinne von vielen Codezeilen pro Methode und wird in der Praxis sehr häufig angetroffen. Guter Codingstil ist es aber, Methoden nicht unnötig lang und unübersichtlich zu halten, sondern stattdessen einzelne Teilaufgaben einer langen Methode in eigene Methoden auszulagern. Dies hat neben besserer Lesbarkeit auch den Vorteil, dass gewisse Teile einfacher wiederverwendet werden können und Programme dadurch generell stabiler werden. Kurze Methoden mit einer guten Bezeichnung sind für den Entwickelnden einfach lesbar und veränderbar. Auf diese Weise hat jede Methode ein bestimmtes Aufgabengebiet und dadurch wird das Prinzip der Kohäsion unter Methoden gestärkt.

In Version 3 des vorgestellten Beispielprogramms befinden sich unter anderem durch das Hinzufügen von Duplicate und Dead Code einige solcher langen Methoden, welche die Verständlichkeit beeinträchtigen sollen, um diesen Smell zu simulieren. So kann man beispielsweise die Methoden *getGesamtLeihgebuehr()* aus Listing 6.6 mit der gleichnamigen, längeren Methode aus Listing 6.13 vergleichen, welche circa dreimal so lang ist, jedoch die komplett idente Funktionalität aufweist.

Durch kurze, kohärente Methoden lassen sich zudem auch Codeduplikate, wie in Listing 6.15 beschrieben, deutlich reduzieren. Einziger Nachteil von zu kleinen Methoden ist, dass manchmal die insgesamte Lesbarkeit eines Programmabschnitts erschwert wird, da sehr oft in tiefer geschachtelte Methoden gesprungen werden muss, um den eigentlichen Algorithmus zu verstehen und in manchen Fällen können zahlreiche verschachtelte Methodenaufrufe zu Konfusionen beim Lesenden führen. Diesem Umstand kann jedoch durch präzises Naming der einzelnen Methoden begegnet werden; denn gibt der Name genauen Aufschluss darüber, welchen Zweck diese innehat, so ist es nicht zwingend notwendig, sich den genauen Inhalt einer Methode anzusehen, es sei denn, darin müssen Änderungen aufgrund von Fehlern oder Erweiterungen vorgenommen werden.

## 6.5. Aufgabenstellung der Experiment-Teilnehmer

Ziel des durchgeführten Experiments war es, Unterschiede zwischen verschiedenen Stufen unterschiedlicher Codequalität in der Praxis zu überprüfen. Zu diesem Zweck wurden – wie schon in Kapitel 6 beschrieben – insgesamt vier Versionen ein und derselben Funktionalität jedoch mit unterschiedlicher Codequalität, entworfen. Als letzten Schritt galt es, Aufgabenstellungen für teilnehmende Personen zu kreieren.

Für diese Aufgaben lag besonderer Fokus darauf, häufig auftretende, realistische Szenarien aus der Praxis der Softwareentwicklung zu finden. In der Summe wurden dafür in jede der vier Versionen drei Fehler sowie ein Erweiterungswunsch definiert, welche somit die zu lösende Aufgabe für die Experimentteilnehmer darstellten. Gerade beim Finden und Beheben von Fehlern kommt es meist nicht darauf an, komplizierte Neuentwicklungen vorzunehmen, sondern oft lassen sich häufige Fehler sehr einfach durch wenige Zeilen oder einzelne Statements beheben, jedoch besteht die Schwierigkeit vor allem darin, diese Fehler – beziehungsweise deren Ursache – zu entdecken.

Zu diesem Zweck wurden vor der Integration der drei Fehler alle vier Versionen der eLibrary mit umfassenden JUnit Testfällen versehen, um durch die künstliche Platzierung von Fehlerquellen keine weiteren Inkonsistenzen unwissentlich einzubauen.

In nachfolgenden Abschnitten werden die einzelnen Fehlerbeschreibungen – sowie die Anforderung einer Erweiterung anhand der originalen, schönen Version – veranschaulicht und zur Wahrung der Übersichtlichkeit wird hier auf die Darstellung derselben Punkte in den anderen Versionen verzichtet. Zudem werden nur relevante Teile der jeweiligen Methoden gezeigt, welche nötig sind, um die Auswirkungen der fehlerhaften Codeteile darzustellen. Die vollständige Aufgabenbeschreibung, welche die Teilnehmerinnen und Teilnehmer im Zuge des Experiment in Papierform erhalten haben, befindet sich in Anhang A.5.

### 6.5.1. Fehler 1: Rundungsproblem

Häufig kommt es vor, dass bei Berechnungen vergessen wird, nicht nur auf ganze Zahlen sondern auch auf mögliche Fließkommazahlen zu reagieren. Hierfür sollten normalerweise Datentypen wie *long* oder *float* verwendet werden, jedoch kann es vorkommen, dass in der Hektik Datentypen zur Speicherung von ganzzahligen Werten verwendet werden, und dadurch, ohne Warnung des Kompilers, das Ergebnis abgeschnitten und verfälscht wird. In Listing 6.16 ist zu sehen, wie dieses Problem durch Ersetzen des Datentyps *double* durch *int* der Variable *leihgebuehr* simuliert wurde.

```
1  public double getGesamtLeihgebuehr() {
2    List<Leihe> retournierteBuecher = getRetournierteBuecher();
3
4    int leihgebuehr = 0;
5    for (Leihe leihe : retournierteBuecher) {
6      leihgebuehr += leihe.getLeihgebuehr();
7    }
8
9    // Bonuspunkte verrechnen
10
11   // Billigstes Buch abziehen
12
13   // Bonuspunkte zuweisen
14
15   // Leihen entfernen
16
17   return leihgebuehr;
18 }
```

**Listing 6.16:** Fehler 1: Rundungsproblem

Das Spezielle an diesem Fehler ist hier, dass dieser nicht zwingend immer auftritt. Wenn beispielsweise die Leihgebühr eines Buches immer ganzzahlig ist, wird diese Änderung keine Auswirkungen haben. Wenn allerdings die retournierte Leihgebühr einer Leihe nicht ganzzahlig ist, tritt dieser Fehler auf und der Kompiler schneidet jegliche Nachkommastellen einfach weg, um dem Datentyp *int* gerecht zu werden.

### 6.5.2. Fehler 2: Businesslogik

Durch den Fehler 2 wird ein Fehler im Ablauf der bestehenden Geschäftslogik simuliert. In der Berechnung der gesamten Leihgebühren werden richtigerweise nur diejenigen Bücher verrechnet, welche bereits vom Kunden zurückgegeben wurden. Schließlich ist es möglich, dass ein Kunde zwei von drei Büchern separat zurückbringt, das dritte jedoch noch weiter behält. Aus diesem Grund gibt es in der Fachlogik der Klasse *Kunde.java* die dafür vorgesehene Methode *getRetournierteBuecher()*, welche all jede Buchleihen retourniert, welche als zurückgegeben durch das Setzen der Leihtage vermerkt sind. Für den hier beschriebenen Fachfehler wurde nun ebendieser Aufruf durch die gesamte Liste an Leihen ersetzt und daher in der Abrechnung auch jene Leihen verrechnet, welche noch gar nicht zurückgegeben wurden. In Listing 6.17 ist dieser Fehler in Zeile fünf zu sehen.

```
1  public double getGesamtLeihgebuehr() {
2    List<Leihe> retournierteBuecher = getRetournierteBuecher();
```

```
3
4     int leihgebuehr = 0;
5     for (Leihe leihe : this.getLeihen()) {
6       leihgebuehr += leihe.getLeihgebuehr();
7     }
8
9     // Bonuspunkte verrechnen
10
11    // Billigstes Buch abziehen
12
13    // Bonuspunkte zuweisen
14
15    // Leihen entfernen
16
17    return leihgebuehr;
18  }
```

**Listing 6.17:** Fehler 2: Logikfehler in der Ermittlung von Leihgebühren

Auch dieser Fehler muss nicht direkt auffallen, denn wenn ein Kunde alle Bücher geschlossen retourniert, stimmt die Berechnung, jedoch stimmt diese nicht, falls – wie erläutert – nur einige Bücher retourniert werden.

### 6.5.3. Fehler 3: Businesslogik

Der dritte Fehler wurde in die Fachlogik des Retournierens von Büchern integriert. Wie in Listing 6.18 gezeigt wird, existiert eine Methode *buchAusleihen()* in der Klasse *BibliothekManager.java*, wo die jeweilige Leihe für ein Buch erstellt und dem Kunden zugeteilt wird. Zudem wird in dieser Methode die Menge verfügbarer Exemplare für das geliehene Buch verringert. Gibt es keine Exemplare mehr für dieses Buch, so wird eine Exception geworfen.

Im Gegensatz dazu wird richtigerweise in der Methode *buchZurueckgeben()* die Menge der Buchexemplare wieder erhöht, da ansonsten jedes Buch nur so oft verliehen werden könnte wie insgesamt Exemplare vorhanden sind, auch wenn Kunden ihre Bucher wieder zurückgebracht haben. Ebendieser Fehler wurde hier simuliert, indem das Erhöhen der Exemplare aus der Methode *buchZurueckgeben()* entfernt wurde.

```
1   public void buchAusleihen(Kunde kunde, Buch buch) throws ↩
       KeineExemplareVerfuegbarException {
2     buch.verringereExemplare();
3     Leihe leihe = new Leihe(buch);
4     kunde.getLeihen().add(leihe);
```

```
5   }
6
7   public void buchZurueckgeben(Leihe leihe, int leihtage) {
8     leihe.setLeihtage(leihtage);
9     leihe.getBuch().erhoeheExemplare();
10  }
```

**Listing 6.18:** Fehler 3: Fehler in der Buchretournierung

### 6.5.4. Erweiterung: Kinderbuch

Zusätzlich zu den oben beschrieben Fehlern, die es zu beheben galt, wurde eine zusätzliche Erweiterung gefordert. Hierfür wurde der in der originalen Version der eLibrary bereits vorhandene Buchtyp Kinderbuch entfernt, und die Aufgabe der Teilnehmer war es, ebendiesen Buchtyp in das System zu integrieren. Hierbei war die Vorgehensweise für die verschiedenen Versionen der eLibrary sehr unterschiedlich. In der originalen Version der Anwendung wurden verschiedene Buchtypen durch den Einsatz von Polymorphie gelöst und daher musste hier lediglich die Klasse *Kinderbuch.java* entfernt werden. In anderen Versionen waren Buchtypen durch eigene Verzweigungen innerhalb der Klasse *Buch.java* umgesetzt und hier mussten ebendiese Verzweigungen für den Buchtyp Kinderbuch entfernt werden.

## 6.6. Zusammenfassung

Dieses Kapitel widmete sich der detaillierten Beschreibung und Ausführung des durchgeführten Experiments. Hierfür wurden zuerst die Vorgangsweise sowie der Aufbau und die Rahmenbedingungen des Experiments erläutert. Weiters wurde ein Einblick in die verwendete Beispielapplikation und deren Eigenheiten gegeben und die definierte Aufgabenstellung der Experiment-Teilnehmer definiert. Schlussendlich wurde gezeigt, wie die unterschiedlich qualitativen Versionen der Beispielanwendung erstellt und mit unterschiedlichen Codesmells versehen wurden, um Problemstellungen aus der Praxis nachzustellen und deren Auswirkungen auf den Wartungsaufwand im Zuge des Experiments zu eruieren. Darauf folgend befasst sich das nächste Kapitel mit der Evaluierung und den Ergebnissen des durchgeführten Experiments.

# 7. Evaluierung

## 7.1. Einleitung

In diesem Kapitel werden die experimentell erhobenen Ergebnisse sowie Ergebnisse der statischer Code Analyse präsentiert und verschiedene Auswertungen dargestellt. Zudem werden die erhaltenen Ergebnisse durch Experten aus dem Java-Bereich mittels offener Interviews auf deren Plausibilität und Tragfähigkeit validiert.

An dem durchgeführten Experiment haben in Summe 20 Entwickler teilgenommen. Wie bereits in Abschnitt 6.2 erwähnt, wurde hierfür eine eigene Kriterienliste herangezogen, und grundsätzliche Java-Erfahrung von mindestens zwei Jahren gefordert, um die Ergebnisse nicht durch unerfahrene Entwickler zu verfälschen. Zudem wurde darauf geachtet, jedem Teilnehmer zufällig eine der vier Versionen der Anwendung eLibrary zuzuteilen, um maximale Durchmischung zu erhalten. Die genaue Verteilung der Teilnehmeranzahl zu den verschiedenen Programmversionen findet sich in Tabelle 7.1.

| **eLibrary-Version** | **Anzahl** |
|---|---|
| Version 1 (Naming, Javadoc) | 5 |
| Version 2 (Good Quality) | 5 |
| Version 3 (Complexity, Duplicate Code, ...) | 5 |
| Version 4 (Bad Quality, Smells Combined) | 5 |
| Summe | 20 |

**Tabelle 7.1.:** Verteilung der Teilnehmer des Experiments

## 7.2. Quantitative Evaluierung durch statische Code Analyse

Um zusätzlich zur qualitativen Evaluierung des Experiments eine quantitative Bewertung der einzelnen Programmversionen zu bekommen, wurden diese mittels einiger in Kapitel 5 beschriebener Werkzeuge einer statische Code-Analyse unterzogen.

Mit dieser Analyse sollte überprüft werden, ob es einen wahrnehmbaren Qualitätsunterschied aus Sicht der statischen Code-Analyse gibt und wie deutlich dieser zu beobachten ist. Dafür wurden die Werkzeuge PMD (siehe Abschnitt 5.3), Checkstyle (siehe Abschnitt 5.2) sowie SonarQube (siehe Abschnitt 5.7) ausgewählt und berechnete Ergebnisse nachfolgend in Tabelle 7.2 dargestellt.

| **Metrik** | **V1 (Naming)** | **V2 (Good)** | **V3 (Complex.)** | **V4 (Bad)** |
|---|---|---|---|---|
| LOC | 356 | 631 | 753 | 504 |
| CLOC | 4 | 278 | 253 | 5 |
| McCabe | 37 | 37 | 80 | 80 |
| Max Cylco. Compl. | 5 | 5 | 26 | 26 |
| Short Var. | 26 | 0 | 0 | 18 |

**Tabelle 7.2.:** Ergebnisse aus statischer Code-Analyse

Aus den Ergebnissen dieser statischen Codeanalyse geht deutlich hervor, dass sich die implementierten Codesmells deutlich in Zahlen belegen lassen. So sieht man beispielsweise an der McCabe Metrik – beziehungsweise der zyklomatischen Komplexität, dass diese Werte in den Versionen 3 und 4 deutlich höher sind. Zudem lassen sich Codeduplikate sowie Dead Code und aufwändige Codekonstrukte durch den Umfang der gemessenen Codezeilen darstellen. Durch die Messung kurzer Variablennamen kann auch die Verwendung von schlechtem Naming angedeutet werden, wobei dieser Codesmell generell nicht adäquat durch Tooleinsatz geprüft werden kann, da aktuelle Werkzeuge noch keinen Weg kennen, die Sinnhaftigkeit von Methoden- und Variablenbezeichnungen zu analysieren.

## 7.3. Ergebnisse Experiment

Nachdem alle Teilnehmer des Experiments ihre Bearbeitung abgeschlossen hatten, wurde die benötigte Zeit vermerkt und die bearbeiteten Beispielanwendungen ausgewertet. Dafür wurden unter anderem JUnit Tests zu Hilfe genommen, um sicherzustellen, dass die jeweilige Aufgabe erfüllt wurde und ob ungewollte Seiteneffekte, sogenannte Bad Fixes, durch eine falsche Analyse oder Bearbeitung der Teilnehmer eingebaut wurden. Wie bereits in Abschnitt 6.2 erwähnt wurde allen Teilnehmern ein Zeitrahmen von etwa einer Stunde zur Verfügung gestellt, bei Überschreitung dessen wurde die Bearbeitung abgebrochen und die bisherigen Ergebnisse ausgewertet. Sämtliche persönliche Daten der Teilnehmer wurden anonymisiert behandelt und dafür deren Namen durch generische Abkürzungen wie T1, T2 ersetzt.

In den nachfolgenden Tabellen – Tabelle 7.3,Tabelle 7.4, Tabelle 7.5 und Tabelle 7.6 – sind die Ergebnisse der erfolgreichen Umsetzungen der einzelnen Teilnehmer nach

deren Programmversionen dargestellt, wobei ein Haken die erfolgreiche Bearbeitung und ein X eine fehlerhafte Lösung darstellt. Zudem sind in den Spalten mit der Bezeichnung Bad Fix jene Bearbeitungen mit einem X und der Anzahl solcher Bad Fixes gekennzeichnet, welche ein oder mehrere Bad Fixes aufwiesen.

| Teilnehmer | Fehler 1 | Fehler 2 | Fehler 3 | Erweiterung | Bad Fix |
|---|---|---|---|---|---|
| T1 | ✓ | ✘ | ✓ | ✓ | ✘ (1) |
| T2 | ✓ | ✓ | ✓ | ✘ | ✘ (3) |
| T3 | ✓ | ✓ | ✓ | ✓ | ✘ (1) |
| T4 | ✓ | ✓ | ✓ | ✓ | |
| T5 | ✓ | ✓ | ✓ | ✓ | |

**Tabelle 7.3.:** Aufgabenerfüllung Version 1 (Naming)

| Teilnehmer | Fehler 1 | Fehler 2 | Fehler 3 | Erweiterung | Bad Fix |
|---|---|---|---|---|---|
| T1 | ✓ | ✓ | ✓ | ✓ | |
| T2 | ✓ | ✓ | ✓ | ✓ | |
| T3 | ✓ | ✓ | ✓ | ✓ | |
| T4 | ✓ | ✓ | ✓ | ✓ | |
| T5 | ✓ | ✓ | ✓ | ✓ | |

**Tabelle 7.4.:** Aufgabenerfüllung Version 2 (Good Quality)

| Teilnehmer | Fehler 1 | Fehler 2 | Fehler 3 | Erweiterung | Bad Fix |
|---|---|---|---|---|---|
| T1 | ✓ | ✓ | ✓ | ✓ | |
| T2 | ✓ | ✓ | ✓ | ✓ | |
| T3 | ✓ | ✓ | ✓ | ✓ | |
| T4 | ✓ | ✓ | ✓ | ✓ | |
| T5 | ✓ | ✓ | ✓ | ✓ | |

**Tabelle 7.5.:** Aufgabenerfüllung Version 3 (Complexity)

In den dargestellten Bearbeitungsergebnissen war es interessant zu sehen, dass – entgegen anderer Erwartungen – nahezu in allen Bearbeitungen sämtliche Aufgabenstellungen erfolgreich erfüllt wurden. Dennoch ist zu beobachten, dass in den Programmversionen 1 (schlechtes Naming) und 4 (Bad Quality) einige Bad Fixes zu beobachten waren. Diese reichten von zusätzlicher Integration von Codeduplikaten über falsche Berechnungen bis hin zu geänderter Businesslogik.

Weiters ist in der qualitativen Beurteilung der resultierenden Sourcecodedateien aufgefallen, dass gerade in den Versionen 1 und 4, welche schlechtes Naming aufwiesen, tendentiell unsaubere, wenn auch funktionierende Fehlerbehebungen implementiert

| Teilnehmer | Fehler 1 | Fehler 2 | Fehler 3 | Erweiterung | Bad Fix |
|---|---|---|---|---|---|
| T1 | ✓ | ✓ | ✓ | ✓ | ✘ (1) |
| T2 | ✓ | ✓ | ✓ | ✓ | |
| T3 | ✓ | ✓ | ✓ | ✓ | |
| T4 | ✓ | ✓ | ✓ | ✓ | |
| T5 | ✓ | ✓ | ✓ | ✓ | ✘ (1) |

**Tabelle 7.6.:** Aufgabenerfüllung Version 4 (Bad Quality)

wurden. So wurden die vermeintlichen Fehler beispielsweise an anderen Stellen im Sourcecode ausgebessert, als dies die Mehrheit der übrigen Teilnehmer getan hat. Zudem wurde schlechtes Naming in neu implementierte Codeabschnitte quasi mitübernommen, was den Verdacht erhärtet, dass schlechtes Naming meist zu noch mehr schlechtem Naming verleitet.

Neben der Auswertung der Bearbeitungsergebnisse qualitativer Art wurde zudem die Zeitmessung aller Teilnehmer ausgewertet und in Tabelle 7.7 dargestellt.

| V1 (Naming) | V2 (Good) | V3 (Complexity) | V4 (Bad) |
|---|---|---|---|
| 0:35:06 | 0:31:12 | 0:43:13 | 0:20:02 |
| 0:34:49 | 0:25:01 | 0:32:03 | 0:30:04 |
| 0:20:43 | 0:59:36 | 0:27:24 | 0:27:32 |
| 0:16:27 | 0:25:38 | 0:43:07 | 0:28:25 |
| 0:41:03 | 0:23:13 | 0:25:56 | 0:42:11 |
| **0:29:38** | **0:32:56** | **0:34:21** | **0:29:39** |

**Tabelle 7.7.:** Bearbeitungszeit der Teilnehmer

Um die Streuung und Unterscheidungen in der Bearbeitungszeit optisch vergleichbar darzustellen, wurden die Daten aus Tabelle 7.7 zusätzlich in übersichtlicher Form mittels Boxplot in Abb. 7.1 auf Sekundenbasis dargestellt.

Da in der Analyse der erhobenen Zeitmessungen einige Unregelmäßigkeiten beziehungsweise Abweichungen vorhanden waren, wurden diese durch den Mittelwert der jeweiligen Gruppe ersetzt und diese Auswertung in Abb. 7.2 dargestellt. Zum Beispiel kam in der Analyse eines Teilnehmers heraus, dass dieser Probleme mit der deutschen Sprache hatte, da dies nicht seine Muttersprache war, und daher eine überdurschnittlich lange Bearbeitungsdauer in der Programmversion Good zu erklären ist.

Es konnten – sowohl bei den originalen Ergebnissen der Zeitmessung als auch bei den bereinigten Ergebnissen – keine deutlichen Unterschiede beobachtet werden. Le-

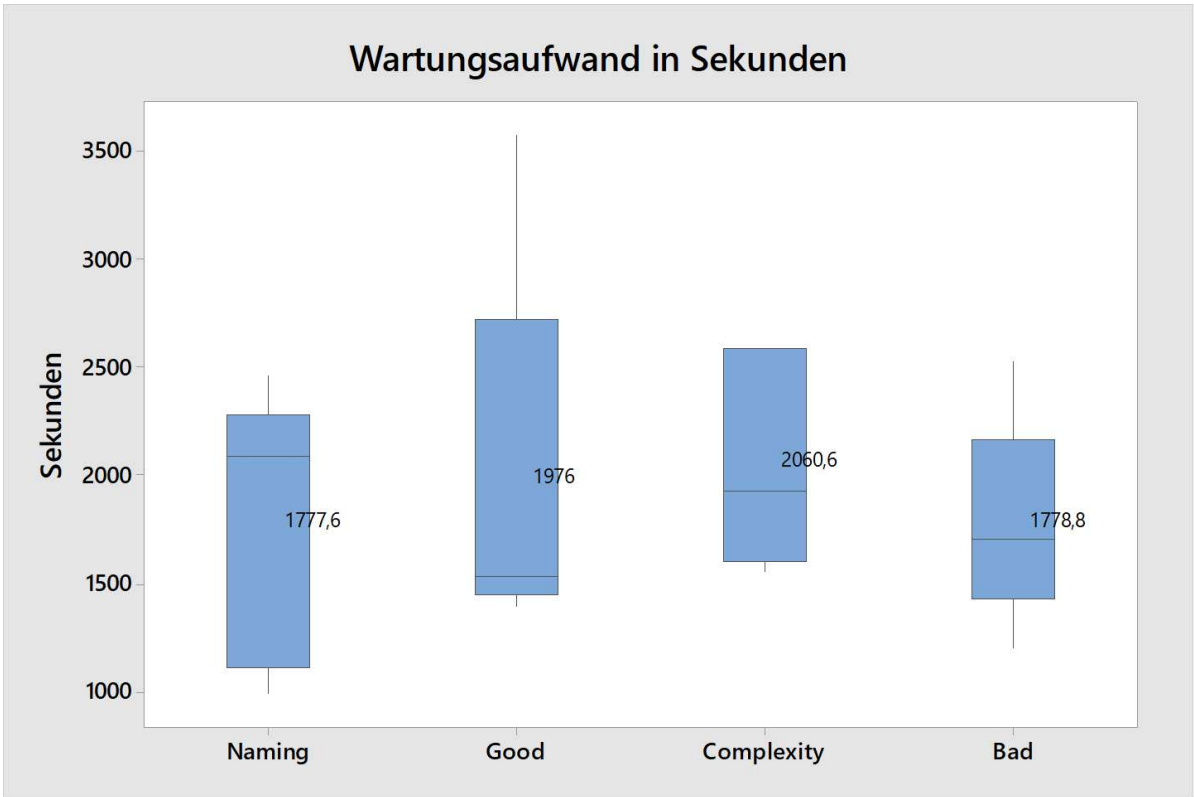



**Abbildung 7.1.:** Bearbeitungszeit in Sekunden

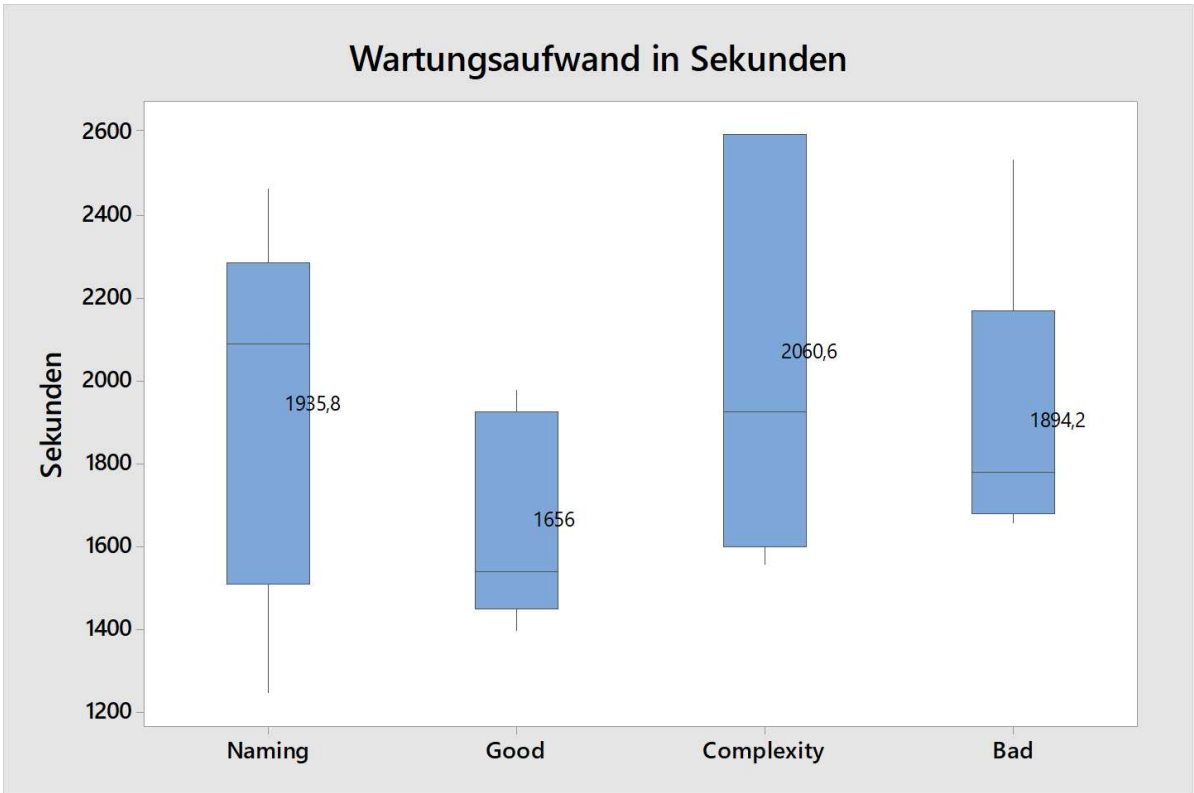



**Abbildung 7.2.:** Bearbeitungszeit in Sekunden (Abweichungen durch Mittelwerte ersetzt)

diglich in den bereinigten Ergebnissen in Abb. 7.2 lässt sich erahnen, dass die Version mit guter Qualität in der Gesamtbearbeitungzeit tendenziell etwas besser war, jedoch ist der Unterschied so gering, dass hier keine klare Aussage getroffen werden kann. So machte es – angesichts der benötigten Zeiten für die Bearbeitung – kaum einen Unterschied, welche der unterschiedlich qualitativen Versionen bearbeitet wurde. Über mögliche Hintergründe für dieses Ergebnis werden einige Szenarien in Kapitel 8 näher erläutert und kritisch diskutiert. Zudem werden diese Ergebnisse im folgenden Abschnitt durch die Einschätzungen von Experten bewertet und analysiert.

## 7.4. Validierung der Ergebnisse durch Experten

Um die erhaltenen Ergebnisse auf deren Plausibilität und Validität hin zu prüfen und so das Ergebnis auf eine breitere Basis zu stellen, wurden qualitative Einschätzungen von Experten aus dem Umfeld der Softwareentwicklung zu Einschätzungen hinsichtlich Durchführung und Ergebnisse der durchgeführten Untersuchung eingeholt. Diese Befragungen wurden unter Zuhilfenahme des Vorgehensmodells nach Mayring (2003) durch offenen Fragestellungen vollzogen. Als Befragungsform wurde eine mündliche Befragung im Sinne eines Interviews gewählt und diese mittels Gesprächsleitfaden nach Schnell, Hill und Esser (1999) durchgeführt. Hierbei wurden qualitative Einschätzungen – hinsichtlich der Untersuchung sowie der Ergebnisse – von Experten erhoben, sowie bei Bedarf in relevante Themenrichtungen hingeführt. Die Auswertung erfolgte nach (Mayring, 2002) und orientierte sich an dessen Ablaufmodell der zusammenfassenden Inhaltsanalyse. Hierfür wurde jedes Gespräch aufgezeichnet und relevante Aussagen in neutraler Form paraphrasiert. Danach wurden Kernaussagen im Sinne einer Generalisierung ermittelt. Die gesamte Auswertungstabelle findet sich im Anhang in Tabelle A.1.

Für die Auswahl geeigneter Experten, welche zur Validierung und Diskussion der erhaltenen Ergebnisse herangezogen wurden, waren folgende Kriterien festgelegt.

- Langjährige Berufserfahrung, speziell im Umfeld der Java-basierten Softwareentwicklung. Hierfür sollten mehr als acht Jahre Erfahrung in diesem Bereich erforderlich sein.

- Zusätzlich zu langjähriger Erfahrung sollten ausgewählte Experten auch über ein breiteres Hintergrundwissen von Systemarchitekturen und Software-Design besitzen. Hierfür sind Software-Architekten und Systementwicklerinnen und Systementwickler hervorragend geeignet.

Nachfolgend werden Kernaussagen und Interpretationen aus der Auswertung der Expertengespräche (Tabelle A.1) nach relevanten Themengebieten gruppiert dargestellt.

- Vorgangsweise der Teilnehmer

  Nach Einschätzung der Experten sowie einiger Rückmeldungen von Teilnehmern war die individuelle Vorgehensweise etwas unterschiedlich. Einige haben die Anwendung in Ruhe analysiert und wollten den Inhalt verstehen, andere haben sich zielgerichtet auf die Suche der Fehler und die Implementierung der Erweiterung fokussiert und durch gezieltes Debugging der Main Klasse die Fehler behoben.

- Plausibilität der Ergebnisse

  Nach Einschätzung aller Experten scheinen die Ergebnisse durchaus plausibel zu sein. Durch die einerseits vom Umgang der Codebasis vergleichsweise kleine Anwendung und andererseits den mehrheitlichen Einsatz von Debugging scheinen die Ergebnisse nicht unrealistisch. Dies wurde in den Gesprächen dadurch begründet, dass bei reinem zielgerichteten Debugging sowie der überschaubaren Größe der Anwendung die Qualität offensichtlich nicht maßgeblich entscheidend war. Zudem wurde von einem Experten die Meinung vertreten, dass Entwickler durchaus fähig sind, sich bis zu einem gewissen Grad an schlechten Code anzupassen. Dennoch kamen Vorschläge nach einer eventuell präziseren Kategorisierung der Teilnehmerkenntnisse und ihrer Erfahrung anhand geeigneter Gegenprüfungsmaßnahmen, wie später noch genauer erläutert wird.

- Begründung bzw. Interpretation der Ergebnisse

  Es war einheitliche Meinung der Experten, dass die Ergebnisse zum Einen durch die überschaubare Größe der Anwendung und zum Anderen durch die Vorgehensweise im Sinne von zielgerichtetem Debugging des Codes erklärbar seien.

  Eine weitere von den Experten vertretene Begründung war, dass möglicherweise gerade etwas weniger erfahrene Entwickler mit qualitativ minderwertigem Sourcecode besser umgehen können, als erfahrene Entwickler.

  In der Interpretation war es laut Experten spannend zu beobachten, dass sich schlechter Code offensichtlich durch noch mehr schlechten Code fortpflanzt, was sowohl an den Ergebnissen der Bad Fixes als auch an den Einschätzungen aus der Praxis zu erkennen sei.

  Abschließend waren sich alle Experten in der Argumentation einig, dass die durchgeführte Untersuchung die Tendenz erkennen lasse, dass Codequalität gerade in kleinen Anwendungen mit kurzer Lebensdauer keine entscheidenden Auswirkungen auf den Wartungsaufwand habe und sich daher in solchen Fällen der Aufwand nicht lohne.

- Einflussfaktoren auf Ergebnisse

  Hinsichtlich der Einflussfaktoren auf die vorliegenden Ergebnisse gab es durchaus unterschiedlich gewichtete, wenn auch thematisch ähnliche Meinungen. Während einerseits die Vorgangsweise durch Zuhilfenahme von Debugging als wesentlicher Faktor ausgemacht wurde, wurde andererseits vermehrt die Größe der Beispielanwendung, sowie die Größe der angelegten Untersuchung, als entscheidender Faktor angesehen.

  Konsens herrschte über den Faktor der individuellen Fähigkeiten der Teilnehmer, sowie deren Erfahrung und Fachwissen in der jeweiligen Programmiersprache. Zudem kam die Meinung auf, dass auch spezielle Kenntnisse der verwendeten IDE gerade für die Bearbeitung und Fehlersuche entscheidend seien.

- Validierung der Teilnehmerkenntnisse

  In der Frage, wie die Kenntnisse der einzelnen Teilnehmer und damit verbunden deren Einfluss auf das Ergebnis in einer zukünftigen Studie besser validiert und gewichtet werden könne, gab es im Wesentlichen zwei interessante Ansätze. Zum einen könnte man funktional wie auch strukturell unterschiedliche Anwendungen mit unterschiedlichen Qualitätsausprägungen gestalten und jedem Teilnehmer alle Versionen zur Bearbeitung vorlegen, was eine adäquate Bewertung der jeweiligen Kenntnisse fördern könnte. Ein anderer Ansatz wäre die Validierung durch eine zusätzliche Aufgabenstellung vor der jeweiligen Bearbeitung. Hierfür könnte man von den Teilnehmern die Erstellung eines Klassen- oder Sequenzdiagrammes fordern, wodurch die individuellen Analysefähigkeiten hervorgehoben würden und ein umfassendes Verständnis der vorliegenden Anwendung garantiert wäre.

- Sonstige Anregungen und Einschätzungen

  Die mehrheitliche Meinung zu künftiger Forschung war eindeutig – wie schon erwähnt – die Größe und Komplexität der Anwendungen sowie die Teilnehmerzahl zu erhöhen, um dadurch sichtbare Auswirkungen erkennen zu können. Eine weitere Meinung war, ob der Einfluss von Javadoc beziehungsweise dessen Abwesenheit großen Einfluss auf Java-Anwendungen hat, weil diese, wie sonstige Kommentare, sehr oft veraltet sind und daher meist nicht gelesen werden.

  Weiters wurde angeführt, dass gerade Duplicate und Dead Code in Kombination mit unübersichtlichen, komplexen Methoden sehr gut ausgewählt wurden, da diese in einem größeren Einsatz, über mehrere Klassen und Module verteilt, unweigerlich zu großen Problemen und Fehlerquellen führen.

Die abschließende Meinung der Experten war, dass umfassende Tests und ausführbarer Code sehr entscheidend für das Erlernen einer Anwendung sein können, generell zu besserem Programmverständnis beitragen und zu besserer Fehleranalyse führen.

Abschließend kann resümiert werden, dass bei den Befragungen der einzelnen Experten sehr interessante und teilweise auch deckungsgleiche Ansichten und Analysen erstellt wurden. Zudem wurden die Ergebnisse grundsätzlich als valide und plausibel eingeschätzt, sowie interessante Themenfelder für zukünftige Forschungsfragen gegeben, welche auch in Abschnitt 9.2 teilweise aufgegriffen wurden.

## 7.5. Zusammenfassung

Im Zuge dieses Kapitels wurden die Ergebnisse der Evaluierung innerhalb des Experiments präsentiert, welche keinen klaren Zusammenhang zwischen Codequalität und etwaigen Auswirkungen auf den Wartungsaufwand belegen konnten. Zusätzlich wurden quantitative Ergebnisse aus der statischen Codeanalyse sowie Einschätzungen von Experten zur Validierung der erhaltenen Ergebnisse festgehalten. Aufbauend auf diesen Ergebnissen werden in nachfolgendem Kapitel mögliche Schlüsse und Interpretationen diskutiert, sowie die aufgestellten Hypothesen geprüft.

# 8. Diskussion

In diesem Kapitel werden die Ergebnisse der experimentellen Untersuchung diskutiert und mögliche Rückschlüsse und Aussagen auf Basis der erhaltenen Ergebnisse getätigt. Zudem werden die in Abschnitt 1.2 aufgestellten und beschriebenen Hypothesen auf deren Gültigkeit geprüft.

## 8.1. Betrachtung der Ergebnisse

Im Zuge des durchgeführten Experiments wurde – entgegen vorheriger Annahmen und Erwartungen – festgestellt, dass die Ergebnisse für die unterschiedlichen Programmversionen alle in etwa auf einem ähnlichen Niveau liegen und keine zwingenden Tendenzen in die eine oder andere Richtung zu erkennen sind.

Nach einer ersten Analyse der Ergebnisse aus dem durchgeführten Experiment liegt die Vermutung nahe, dass die erhaltenen Ergebnisse für die durchgeführte Aufgabenstellung, nämlich hauptsächlich der zielgerichteten Fehlersuche, durchaus vertretbar sind, auch wenn – nach anfänglich gestellten Hypothesen zufolge – eine doch stärkere Abgrenzung hinsichtlich qualitativ schlechterer Codeversionen zu erwarten war. Nach Rückmeldungen vieler Teilnehmer, fanden diese die vorliegenden schlechten Programmversionen durchaus schlecht lesbar und schwer verständlich. Nichtsdestotrotz konnten die Fehlerquellen durch gezieltes Debugging ohne erwähnenswerten Zeitverlust gegenüber der guten Version gefunden und behoben werden. Laut Aufgabenstellung war ein umfassendes Verständnis der Anwendung nicht gefordert, sondern lediglich die Behebung der Fehler sowie die Implementierung einer Erweiterung; daher war es nur verständlich, dass sich die meisten Teilnehmer eben diesem Ziel untergeordnet haben.

Ein weiterer Grund für die ähnlichen Ergebnisse – sowohl in Bearbeitungszeit, als auch in der Qualität der Bearbeitung – könnte darin liegen, dass die im Zuge des Experiments eingesetzten Anwendungsversionen vom Umfang her zu klein waren, um den Effekt einer Aufwandsersparnis in der Wartung und Erweiterung festzustellen.

Ein interessanter Effekt konnte dennoch gerade in den Versionen 1 und 4 mit schlechtem Naming beobachtet werden. So haben sich die Teilnehmer großteils dem Um-

feld der bestehenden Codequalität angepasst und beispielsweise Variablen – entgegen besseren Wissens – mit einsilbigen Bezeichnungen abgekürzt und ähnliches. Dies birgt ein neues, spannendes Feld für weitere Untersuchungen, wo man beispielsweise diesen in kleinem Rahmen beobachteten Effekt – dass sich Entwickler oftmals an die Qualität des vorliegenden Sourcecode adaptieren, entgegen besseren Wissens – großflächig untersuchen könnte.

Als vorläufiges Ergebnis ist abschließend zu erkennen, dass für kleinere, einmalige Softwareprojekte durchaus auf hohe Codequalität verzichtet werden kann, da der Aufwand hierfür nicht aufzuwiegen ist mit dem Nutzen und diese kleine Untersuchung gezeigt hat, dass in einem kleinen Softwareprojekt keine nennenswerten Unterschiede in der Fehlersuche nachweisbar waren. Dennoch bleibt die Kernaussage dieser Untersuchung ganz klar, dass eine weitaus umfangreichere Erhebung mit deutlich mehr Daten von Nöten wäre, um eine mögliche Tendenz der Auswirkungen erkennen zu lassen.

## 8.2. Hypothesenprüfung

Im Zuge des durchgeführten Experiments wurde festgestellt, dass bei den Ergebnissen eher unerwartete Werte beobachtet wurden. Aus der mehrheitlichen Beobachtung lassen sich alle aufgestellten Hypothesen und Subhypothesen (H1, H1.1, H1.2) nicht bestätigen, sondern bedürfen im Zweifelsfall weiterer intensiverer Forschung. Dennoch sei erwähnt, dass die zugehörigen Nullhypothesen nicht angenommen werden können, da es trotz der erhaltenen Ergebnisse und zahlreicher Literatur nicht als sicher angesehen werden kann, dass beschriebene, qualitätssteigernde Maßnahmen keine Auswirkungen auf den nachfolgenden Wartungsaufwand haben. Zudem würde für eine Verwerfung der präsentierten Nullhypothesen ein wesentlich höherer Forschungsaufwand mit deutlich mehr Teilnehmern sowie umfangreicheren Aufgabenstellungen und Beispielanwendungen, welche stärkere Annäherung zu umfangreichen Softwareprojekten aus der täglichen Praxis aufweisen, erforderlich sein.

Die aus den Zielen abgeleitete Forschungsfrage – „*Welchen Einfluss haben ausgewählte Maßnahmen zur Steigerung von Codequalität sowie der Einsatz statischer Prüfmethoden in der Softwareentwicklung auf die Reduktion des Wartungs- und Folgeaufwands von Software?*“ – lässt sich auf Basis der beobachteten Ergebnisse nicht eindeutig beantworten. Den Ergebnissen zufolge könnte man annehmen, dass diese darauf hindeuten, dass Codequalität keine Auswirkungen auf den Wartungs- und Folgeaufwand hat, jedoch gibt es auch dafür keine eindeutigen Ergebnisse. Darüberhinaus wurde aus der Diskussion sowie der Einschätzungen der Theorie und der Expertenbefragung deutlich, dass Auswirkungen wahrscheinlich sind, dieser Effekt jedoch aufgrund des eingeschränkten Umfangs dieser Untersuchung nicht beobachtet werden konnte.

## 8.3. Zusammenfassung

Zusammenfassend kann in diesem Kapitel festgehalten werden, dass für eine aussagekräftige These noch intensiver und umfangreicher geforscht werden müsste und zudem eine deutlich größere Menge an Daten nötig wäre, um hier eine Tendenz erkennen zu können. Aus der mehrheitlichen Beobachtung ging ferner hervor, dass alle aufgestellten Arbeitshypothesen nicht bestätigt werden konnten. Das nachfolgende, abschließende Kapitel fasst die wichtigsten Aspekte dieser Arbeit noch einmal zusammen und gibt einen Ausblick auf weitere Forschungsthemen, welche sich aus den Erkenntnissen dieser Arbeit ableiten lassen.

# 9. Conclusio

In diesem Abschlusskapitel werden alle relevanten Inhalte der vorliegenden Arbeit zusammengefasst dargestellt und abschließend ein Ausblick über zukünftige, auf dieser Arbeit aufbauende Untersuchungsmöglichkeiten gegeben.

## 9.1. Zusammenfassung

Mit dieser Arbeit wurde das Ziel verfolgt, zu untersuchen, ob Codequalität positive Auswirkungen auf den Wartungs- beziehungsweise Folgeaufwand von Software hat. Zu diesem Zweck wurde zu Beginn dieser Arbeit folgende Forschungsfrage aus dieser Motivation heraus abgeleitet: *Welchen Einfluss haben ausgewählte Maßnahmen zur Steigerung von Codequalität sowie der Einsatz statischer Prüfmethoden in der Softwareentwicklung auf die Reduktion des Wartungs- und Folgeaufwands von Software?* Zur Operationalisierung dieser Frage wurden Arbeitshypothesen sowie die zugehörigen Nullhypothesen aufgestellt und später geprüft. Um die Forschungsfrage beziehungsweise die aufgestellten Hypothesen angemessen beantworten zu können, wurde zuerst in einer argumentativ-deduktiven Analyse vorhandene Literatur nach Lösungsansätzen und Praktiken aus der Theorie durchsucht und darauffolgend ein Experiment anhand eines Beispielprogrammes mit Java-versierten Entwicklern durchgeführt. Zudem wurden die Ergebnisse darauffolgend in Experteninterviews auf deren Plausibilität und Validität geprüft und mögliche Einflussfaktoren auf die Ergebnisse erhoben.

Im Zuge der argumentativ-deduktiven Analyse wurden zuerst Grundlagen hinsichtlich Begrifflichkeiten des Themas Codequalität und Qualitätsmerkmale sowie Qualitätsmodelle von Softwareprojekten vorgestellt. Weiters wurden relevante Inhalte aus der Theorie für eine konstruktive Qualitätsgestaltung erarbeitet. Hierfür wurden Prinzipien, Richtlinien, Standards und Best- sowie Bad-Practices aus der Softwareentwicklung vorgestellt. Besonderes Augenmerk galt hier dem Thema Codesmells, welche oft gesehene, schlechte Praktiken und Muster in Sourcecode beispielhaft beschreiben und damit ein Bewusstsein für potentielle Probleme schaffen. Zur Ermittlung und Messung relevanter Qualitätskennzahlen wurden bekannte Metriken – sowohl traditionelle Metriken, beispielsweise für Umfang und Komplexität von Software, als auch Metriken für objekt-orientierte Softwareprogramme – erörtert. Zuletzt

wurden darauf aufbauend Werkzeuge vorgestellt, welche Metriken und Codequalität automatisiert messen und bewerten können, und somit ein entscheidender Faktor für die Erstellung qualitativ hochwertiger Softwareprojekte sind.

Im Zuge eines Experiments wurde ein Beispielprogramm entwickelt und mit häufig vorkommenden Codesmells versehen, sodass in Folge vier Versionen derselben Anwendung mit unterschiedlichen Qualitätsstufen gebildet wurden. Nach Auswahl geeigneter Teilnehmer nach einer definierten Kriterienliste wurde jedem Teilnehmer eine dieser vier Versionen zugeteilt. Zudem wurde eine Aufgabenstellung zur Verfügung gestellt, welche das Suchen und Entfernen einiger Fehler sowie die Implementierung einer Erweiterung vorsah. Während der Durchführung dieser Aufgaben wurde für jeden Teilnehmer die benötigte Zeit gemessen und die Ergebnisse später ausgewertet. Entgegen aller Erwartungen und aufgestellter Hypothesen konnte in den Ergebnissen kein klares Indiz für eine Tendenz hinsichtlich der Auswirkungen verschiedener Qualitätsgrade festgestellt werden. Als Gründe hierfür wurden in der darauffolgenden Diskussion sowie in Expertengesprächen mehrere Aspekte analysiert. Eine der Kernaussagen ist, dass für belegbare Auswirkungen auf den Wartungs- und Folgeaufwand deutlich mehr Daten benötigt würden. Auch der Umfang der Untersuchung müsste deutlich erhöht werden, was im Zuge dieser Arbeit nicht möglich war, da dafür deutlich größerer Aufwand betrieben werden müsste. Dennoch ist das Ergebnis sehr interessant, da man durchaus annehmen kann, dass für die Fehlersuche einerseits, aber auch für vom Umfang her kleinere Softwareprojekte die interne Qualität des Quellcode eher zu vernachlässigen ist.

## 9.2. Ausblick

Zum Abschluss dieser Arbeit wird ein Ausblick auf weitere Forschungsmöglichkeiten – aufbauend auf den Ergebnissen dieser Arbeit – gegeben.

Mögliche weitere Forschungen auf diesem Gebiet könnten Untersuchungen innerhalb einer breiteren Anwendung der skizzierten Methodenanalyse sowie die Untersuchung von Auswirkungen weiterer Methoden im Sinne von Codesmells sein. Auch wäre es durchaus sinnvoll, das durchgeführte Experiment auf eine breitere Basis mit mehr Ressourcen – auch im Sinne einer umfassenderen Teilnehmerzahl – zu stellen, um eine durchaus erwartete Tendenz zu beobachten und darüber hinaus den statistischen Beweis anzutreten.

Nach den Ergebnissen wäre es auch interessant, die Größe der Anwendungen, sowie die Anzahl und möglicherweise die Schwierigkeit der Aufgaben und Erweiterungen in einem größeren, praxisnäheren Umfeld anzulegen und zu prüfen, ob hier Tendenzen abgelesen werden können.

Ein zusätzlich sehr spannendes Forschungsfeld, welches sich eher zufällig im Zuge der Untersuchungen ergeben hat, wäre es, den Effekt der bewusst oder unbewussten Adaption von Entwicklerinnen und Entwicklern an die Qualität des vorliegenden Sourcecodes – entgegen besseren Wissens – näher zu untersuchen.

Aus den Erkenntnissen der vorliegenden Untersuchung – wo zutage kam, dass offensichtlich die jeweiligen Eigenschaften einzelner Entwicklerinnen und Entwickler einen nicht unerheblichen Anteil an den erhaltenen Ergebnissen dieser qualitativen Analyse haben, könnte in weiteren Forschungsszenarien die Überlegung angestellt werden, ob es nicht sinnvoll wäre, eine Gegenprüfung der jeweiligen Teilnehmer durchzuführen. Hierzu wäre es denkbar, mehrere vergleichbare unterschiedliche Programme in unterschiedlichen Qualitätsstufen für die Evaluierung zu verwenden, wobei in einem solchen Experiment jeder Teilnehmer auch jede Programmversion bearbeiten würde und dadurch relative Unterschiede der einzelnen Teilnehmer in das Ergebnis miteinfließen könnten.

# A. Anhang A

## A.1. eLibrary - Version 1 (Naming)

### A.1.1. Buch.java

```
1  package at.codequality.elibrary.business.buch;
2
3  import at.codequality.elibrary.business.LeerException;
4  import at.codequality.elibrary.business.Leihe;
5
6  public abstract class Buch {
7
8    private String a;
9
10   private String t;
11
12   private int anz;
13
14   public Buch(String a, String t, int menge) {
15     super();
16     this.a = a;
17     this.t = t;
18     this.anz = menge;
19   }
20
21   public String getAuthor() {
22     return a;
23   }
24
25   public String getTitel() {
26     return t;
27   }
28
29   public int getAnz() {
30     return anz;
31   }
```

```
32
33    public void setAnz(int x) {
34      this.anz = x;
35    }
36
37    public void erhoehe() {
38      anz++;
39    }
40
41    public void abziehen() throws LeerException {
42      if (getAnz() < 1) {
43        throw new LeerException();
44      } else {
45        anz--;
46      }
47    }
48
49    public abstract double zahlung(Leihe l);
50
51    public int bonus(Leihe x) {
52      int punkt = 1;
53
54      if (zahlung(x) > 6) {
55        punkt = 3;
56      }
57      return punkt;
58    }
59  }
```

**Listing A.1:** Buch.java

### A.1.2. Standard.java

```
1  package at.codequality.elibrary.business.buch;
2
3  import at.codequality.elibrary.business.Leihe;
4
5  public class Standard extends Buch {
6
7    public Standard(String a, String t, int exemp) {
8      super(a, t, exemp);
9    }
10
11   @Override
12   public double zahlung(Leihe b) {
```

```
13      double betr = 3.5;
14
15      if (b.getLeihT() > 6) {
16        betr += (b.getLeihT() - 6) * 1.5;
17      }
18      return betr;
19    }
20
21  }
```

**Listing A.2:** Standard

### A.1.3. Bests.java

```
1  package at.codequality.elibrary.business.buch;
2
3  import at.codequality.elibrary.business.Leihe;
4
5  public class Bests extends Buch {
6
7    public Bests(String a, String t, int count) {
8      super(a, t, count);
9    }
10
11   @Override
12   public double zahlung(Leihe x) {
13     double menge = 5;
14
15     if (x.getLeihT() > 3) {
16       menge += (x.getLeihT() - 3) * 2.7;
17     }
18     return menge;
19   }
20
21   @Override
22   public int bonus(Leihe y) {
23     int p = super.bonus(y);
24
25     if (y.getLeihT() > 4) {
26       p++;
27     }
28     return p;
29   }
30 }
```

**Listing A.3:** Bests.java

### A.1.4. Leihe.java

```
1  package at.codequality.elibrary.business;
2
3  import at.codequality.elibrary.business.buch.Buch;
4
5  public class Leihe {
6
7    private Buch b;
8
9    private int leihTage;
10
11   public Leihe(Buch buch) {
12     this.b = buch;
13   }
14
15   public int getLeihT() {
16     return leihTage;
17   }
18
19   public void setLeihT(int lt) {
20     this.leihTage = lt;
21   }
22
23   public Buch getB() {
24     return b;
25   }
26
27   public double rechnen() {
28     return getB().zahlung(this);
29   }
30
31   public int checkBonus() {
32     return getB().bonus(this);
33   }
34
35 }
```

**Listing A.4:** Leihe.java

### A.1.5. Kunde.java

```
1  package at.codequality.elibrary.business;
2
```

```
3  import java.util.ArrayList;
4  import java.util.List;
5
6  public class Kunde {
7
8    private String name;
9
10   private int bonus;
11
12   private List<Leihe> lList = new ArrayList<>();
13
14   public Kunde(String n) {
15     this.name = n;
16   }
17
18   public String getName() {
19     return name;
20   }
21
22   public int bonus1() {
23     return bonus;
24   }
25
26   public List<Leihe> getBue() {
27     return lList;
28   }
29
30   public void checkBonus(List<Leihe> bList) {
31     for (Leihe l : bList) {
32       bonus += l.checkBonus();
33     }
34   }
35
36   public void verkleinereMenge(int x) {
37     if (!(x > this.bonus)) {
38       bonus -= x;
39     }
40   }
41
42   public List<Leihe> getBuecher() {
43     ArrayList<Leihe> bSet = new ArrayList<>();
44     for (Leihe l : getBue()) {
45       if (l.getLeihT() > 0) {
46         bSet.add(l);
47       }
48     }
49     return bSet;
50   }
```

```
51
52    public double getBuchFuerVerringerung() {
53      double kosten = 0;
54      for (Leihe l : getBue()) {
55        if (kosten == 0) {
56          kosten = l.rechnen();
57        }
58        if (l.rechnen() < kosten) {
59          kosten = l.rechnen();
60        }
61      }
62      return kosten;
63    }
64
65    public double findZahlung() {
66      List<Leihe> bue = getBuecher();
67
68      int kosten = 0;
69      for (Leihe le : getBue()) {
70        kosten += le.rechnen();
71      }
72
73      while (bonus1() > 9 && kosten > 1) {
74        kosten -= 1;
75        verkleinereMenge(10);
76      }
77
78      if (kosten > 20) {
79        kosten -= getBuchFuerVerringerung();
80      }
81
82      checkBonus(bue);
83      getBue().removeAll(bue);
84
85      return kosten;
86    }
87
88  }
```

**Listing A.5:** Kunde.java

### A.1.6. LeerException.java

```
1  package at.codequality.elibrary.business;
2
```

```
3  public class LeerException extends Exception {
4
5  }
```

**Listing A.6:** LeerException.java

### A.1.7. MainController.java

```
1  package at.codequality.elibrary.business;
2
3  import java.util.ArrayList;
4  import java.util.List;
5
6  import at.codequality.elibrary.business.buch.Buch;
7
8  public class MainController {
9
10   private static MainController mc;
11
12   private List<Buch> bList = new ArrayList<>();
13
14   private List<Kunde> kList = new ArrayList<>();
15
16   private MainController() {
17   }
18
19   public static MainController getInstance() {
20     if (mc == null) {
21       mc = new MainController();
22     }
23     return mc;
24   }
25
26   public void kundeErweitern(Kunde kunde) {
27     kList.add(kunde);
28   }
29
30   public void buchErweitern(Buch buch) {
31     bList.add(buch);
32   }
33
34   public List<Kunde> getKunde() {
35     return kList;
36   }
37
```

```
38   public List<Buch> getBuch() {
39     return bList;
40   }
41
42   public void bNehmen(Kunde x, Buch y) throws LeerException {
43     y.abziehen();
44     Leihe l = new Leihe(y);
45     x.getBue().add(l);
46   }
47
48   public void bGeben(Leihe l, int lt) {
49     l.setLeihT(lt);
50   }
51
52   public double findZahlung(Kunde k) {
53     return k.findZahlung();
54   }
55
56 }
```

**Listing A.7:** MainController.java

## A.2. eLibrary - Version 2 (Good Quality)

### A.2.1. Buch.java

```
1  package at.codequality.elibrary.business.buch;
2
3  import at.codequality.elibrary.business.↩
       KeineExemplareVerfuegbarException;
4  import at.codequality.elibrary.business.Leihe;
5
6  /**
7   * Abstrakte Klasse eines Buches. Konkrete Sub-Klassen ( z.B. ↩
        Bestseller,
8   * Standardroman, Kinderbuch) erweitern diese Klasse.
9   *
10  * @author Jakob
11  *
12  */
13 public abstract class Buch {
14
15   /** Autor des Buches. */
16   private String autor;
```

```
17
18  /** Titel des Buches. */
19  private String titel;
20
21  /** Verfuegbare Exemplare dieses Buches. */
22  private int exemplare;
23
24  /**
25   * Erstellt ein Buch mit den angegebenen Parametern.
26   *
27   * @param autor
28   * @param titel
29   * @param exemplare
30   */
31  public Buch(String autor, String titel, int exemplare) {
32    super();
33    this.autor = autor;
34    this.titel = titel;
35    this.exemplare = exemplare;
36  }
37
38  /**
39   * Liefert den Autor des Buches.
40   *
41   * @return Autor des Buches
42   */
43  public String getAuthor() {
44    return autor;
45  }
46
47  /**
48   * Liefert den Titel des Buches.
49   *
50   * @return Titel des Buches
51   */
52  public String getTitle() {
53    return titel;
54  }
55
56  /**
57   * Liefert die Anzahl der vorhandenen Exemplare.
58   *
59   * @return Anzahl der vorhandenen Exemplare
60   */
61  public int getExemplare() {
62    return exemplare;
63  }
64
```

```
65  /**
66   * Setzt die Anzahl verfuegbarer Exemplare.
67   *
68   * @param exemplare
69   *            Anzahl verfuegbarer Exemplare
70   */
71  public void setExemplare(int exemplare) {
72    this.exemplare = exemplare;
73  }
74
75  /**
76   * Erhoehen der Exemplare um den Wert 1.
77   */
78  public void erhoeheExemplare() {
79    exemplare++;
80  }
81
82  /**
83   * Verringern der Exemplare um den Wert eins.
84   *
85   * @throws KeineExemplareVerfuegbarException
86   *             Wenn keine Exemplare mehr verfuegbar sind
87   */
88  public void verringereExemplare() throws ↩
       KeineExemplareVerfuegbarException {
89    if (getExemplare() < 1) {
90      throw new KeineExemplareVerfuegbarException();
91    } else {
92      exemplare--;
93    }
94  }
95
96  /**
97   * Liefert die Leihgebuehr fuer das jeweilige Buch. Jedes Buch ↩
       muss hier ihren
98   * eigenen Algorithmus zur Berechnung der Leihgebuehren ↩
       implementieren.
99   *
100  * @param leihe
101  * @return Kosten der Leihe
102  */
103 public abstract double getLeihgebuehr(Leihe leihe);
104
105 /**
106  * Liefert die Bonuspunkte fuer die uebergebene Leihe abhaengig↩
        von den Leihkosten.
107  * Diese Methode kann bei Bedarf von unterschiedlichen Buch-↩
       Typen ueberschrieben
```

```
108     * werden.
109     *
110     * @param leihe
111     * @return Bonuspunkte fuer die uebergebene Leihe
112     */
113    public int getBonusPunkte(Leihe leihe) {
114      int bonusPunkte = 1;
115
116      if (getLeihgebuehr(leihe) > 6) {
117        bonusPunkte = 3;
118      }
119      return bonusPunkte;
120    }
121  }
```

**Listing A.8:** Buch.java

## A.2.2. Standardroman.java

```
1  package at.codequality.elibrary.business.buch;
2
3  import at.codequality.elibrary.business.Leihe;
4
5  /**
6   * Konkrete Implementierung eines Standardromanes.
7   *
8   * @author Jakob
9   *
10  */
11 public class Standardroman extends Buch {
12
13   /**
14    * Erstellt einen neuen Standardroman mit den angegebenen ↩
        Parametern.
15    *
16    * @param autor
17    *            Autor des Standardromanes
18    * @param titel
19    *            Titel des Standardromanes
20    * @param exemplare
21    *            Verfuegbare Exemplare
22    */
23   public Standardroman(String autor, String titel, int exemplare)↩
        {
24     super(autor, titel, exemplare);
```

```
25    }
26
27    /**
28     * (non-Javadoc)
29     *
30     * @see Buch#getLeihgebuehr(Leihe)
31     */
32    @Override
33    public double getLeihgebuehr(Leihe leihe) {
34      double kosten = 3.5;
35
36      if (leihe.getLeihtage() > 6) {
37        kosten += (leihe.getLeihtage() - 6) * 1.5;
38      }
39      return kosten;
40    }
41
42  }
```

**Listing A.9:** Standardroman.java

### A.2.3. Bestseller.java

```
1  package at.codequality.elibrary.business.buch;
2
3  import at.codequality.elibrary.business.Leihe;
4
5  /**
6   * Konkrete Implementierung eines Bestseller-Buches.
7   *
8   * @author Jakob
9   *
10  */
11 public class Bestseller extends Buch {
12
13   /**
14    * Erstellt ein neues Bestseller-Buch mit den angegebenen ↩
        Parametern.
15    *
16    * @param autor
17    *            Autor des Bestsellers
18    * @param titel
19    *            Titel des Bestsellers
20    * @param exemplare
21    *            Verfuegbare Exemplare
```

```
22     */
23    public Bestseller(String autor, String titel, int exemplare) {
24      super(autor, titel, exemplare);
25    }
26
27    /**
28     * (non-Javadoc)
29     *
30     * @see Buch#getLeihgebuehr(Leihe)
31     */
32    @Override
33    public double getLeihgebuehr(Leihe leihe) {
34      double kosten = 5;
35
36      if (leihe.getLeihtage() > 3) {
37        kosten += (leihe.getLeihtage() - 3) * 2.7;
38      }
39      return kosten;
40    }
41
42    /**
43     * (non-Javadoc)
44     *
45     * @see Buch#getBonusPunkte(Leihe)
46     */
47    @Override
48    public int getBonusPunkte(Leihe leihe) {
49      int bonusPunkte = super.getBonusPunkte(leihe);
50
51      if (leihe.getLeihtage() > 4) {
52        bonusPunkte++;
53      }
54      return bonusPunkte;
55    }
56  }
```

**Listing A.10:** Bestseller.java

### A.2.4. Leihe.java

```
1  package at.codequality.elibrary.business;
2
3  import at.codequality.elibrary.business.buch.Buch;
4
5  /**
```

```
6  * Diese Klasse bildet eine Buch-Ausleihe ab. Eine Leihe besteht ↩
       immer aus einem
7  * Buch mit einer bestimmten Ausleih-Zeit (Leihtage).
8  *
9  * @author Jakob
10 *
11 */
12 public class Leihe {
13
14   private Buch buch;
15
16   private int leihtage;
17
18   /**
19    * Erzeugt eine Leihe fuer ein bestimmtes Buch.
20    *
21    * @param buch
22    *            Buch, welches ausgeliehen wird
23    */
24   public Leihe(Buch buch) {
25     this.buch = buch;
26   }
27
28   /**
29    * Liefert die Anzahl der Tage, an welchen das Buch ausgeliehen↩
           war.
30    *
31    * @return Anzahl der Leihtage
32    */
33   public int getLeihtage() {
34     return leihtage;
35   }
36
37   /**
38    * Setzt die Anzahl der Tage, an welchen das Buch ausgeliehen ↩
         war.
39    *
40    * @param leihtage
41    *            Leihtage
42    */
43   public void setLeihtage(int leihtage) {
44     this.leihtage = leihtage;
45   }
46
47   /**
48    * Liefert das geliehene Buch.
49    *
50    * @return Das ausgeliehene Buch
```

```
51      */
52     public Buch getBuch() {
53       return buch;
54     }
55
56     /**
57      * Liefert die Leihgebuehr fuer die Leihe des Buches.
58      *
59      * @return Leihgebuehr
60      */
61     public double getLeihgebuehr() {
62       return getBuch().getLeihgebuehr(this);
63     }
64
65     /**
66      * Liefert die Bonuspunkte fuer die Leihe des Buches.
67      *
68      * @return Bonuspunkte
69      */
70     public int getBonusPunkte() {
71       return getBuch().getBonusPunkte(this);
72     }
73
74 }
```

**Listing A.11:** Leihe.java

## A.2.5. Kunde.java

```
1  package at.codequality.elibrary.business;
2
3  import java.util.ArrayList;
4  import java.util.List;
5
6  /**
7   * Die Klasse Kunde repraesentiert einen Kunden aus dem echten ↩
        Leben. Ein Kunde
8   * hat einen Namen und kann mehrere Ausleihungen haben. Zudem ↩
        sammeln Kunden
9   * Bonuspunkte fuer ausgeliehene Buecher.
10  *
11  * @author Jakob
12  *
13  */
14 public class Kunde {
```

```
15
16  /** Name des Kunden. */
17  private String name;
18
19  /** Gesammelte Bonuspunkte des Kunden. */
20  private int bonuspunkte;
21
22  /** Liste der Leihen des Kunden. */
23  private List<Leihe> leihen = new ArrayList<>();
24
25  /**
26   * Erzeugt einen neuen Kunden mittels angegebenem Namen.
27   *
28   * @param name
29   *            Name des Kunden
30   */
31  public Kunde(String name) {
32    this.name = name;
33  }
34
35  /**
36   * Liefert den Namen des Kunden.
37   *
38   * @return Name des Kunden
39   */
40  public String getName() {
41    return name;
42  }
43
44  /**
45   * Liefert die Bonuspunkte, welche bis zu diesem Zeitpunkt ↩
       gesammelt wurden.
46   *
47   * @return Gesammelte Bonuspunkte
48   */
49  public int getBonuspunkte() {
50    return bonuspunkte;
51  }
52
53  /**
54   * Liefert die Liste der Ausleihungen des Kunden.
55   *
56   * @return Leihen des Kunden
57   */
58  public List<Leihe> getLeihen() {
59    return leihen;
60  }
61
```

```
62  /**
63   * Bonuspunkte beim retournieren ausgeliehener Buecher dem ↩
       Kunden zuweisen.
64   * Hierfuer werden die Bonuspunkte der einzelnen Leihen ↩
       akkumuliert.
65   *
66   * @param retournierteBuecher
67   */
68  public void bonuspunkteZuweisen(List<Leihe> retournierteBuecher↩
       ) {
69    for (Leihe leihe : retournierteBuecher) {
70      bonuspunkte += leihe.getBonusPunkte();
71    }
72  }
73
74  /**
75   * Verringert die gesamten Bonuspunkte des Kunden um die ↩
       angegebene Anzahl. Das
76   * ist zb. der Fall, wenn ein Kunde Bonuspunkte fuer ↩
       Ermaessigungen verbraucht.
77   *
78   * @param punkte
79   *            Punkte, welche von Punktekonto abgezogen werden ↩
       sollen.
80   */
81  public void bonuspunkteAbziehen(int punkte) {
82    if (!(punkte > this.bonuspunkte)) {
83      bonuspunkte -= punkte;
84    }
85  }
86
87  /**
88   * Liefert die retournierten Buecher eines Kunden. Hierfuer ↩
       werden Buecher
89   * gezaehlt, welche eine Ausleihdauer > 0 aufweisen. Die ↩
       Ausleihdauer wird beim
90   * Zurueckbringen von Buechern gesetzt.
91   *
92   * @return Retournierte Buecher
93   */
94  public List<Leihe> getRetournierteBuecher() {
95    ArrayList<Leihe> retournierteBuecher = new ArrayList<>();
96    for (Leihe leihe : getLeihen()) {
97      if (leihe.getLeihtage() > 0) {
98        retournierteBuecher.add(leihe);
99      }
100   }
101   return retournierteBuecher;
```

```
102   }
103
104   /**
105    * Liefert die Leihgebuehr des billigsten, ausgeliehenen Buches↩
          .
106    *
107    * @return Leihgebuehr des billigsten Buches
108    */
109   public double getKostenDesBilligstenBuches() {
110     double minKosten = 0;
111     for (Leihe leihe : getLeihen()) {
112       if (minKosten == 0) {
113         minKosten = leihe.getLeihgebuehr();
114       }
115       if (leihe.getLeihgebuehr() < minKosten) {
116         minKosten = leihe.getLeihgebuehr();
117       }
118     }
119     return minKosten;
120   }
121
122   /**
123    * Liefert die gesamte Leihgebuehr fuer alle zurueckgebrachten ↩
          Buecher. Zudem
124    * werden entsprechende Bonuspunkte verbraucht/gesetzt und die ↩
          Buecherleihen vom
125    * Kunden entfernt.
126    *
127    * @return Gesamte Leihgebuehr
128    */
129   public double getGesamtLeihgebuehr() {
130     List<Leihe> retournierteBuecher = getRetournierteBuecher();
131
132     int leihgebuehr = 0;
133     for (Leihe leihe : getLeihen()) {
134       leihgebuehr += leihe.getLeihgebuehr();
135     }
136
137     while (getBonuspunkte() > 9 && leihgebuehr > 1) {
138       leihgebuehr -= 1;
139       bonuspunkteAbziehen(10);
140     }
141
142     if (leihgebuehr > 20) {
143       leihgebuehr -= getKostenDesBilligstenBuches();
144     }
145
146     bonuspunkteZuweisen(retournierteBuecher);
```

```
147     getLeihen().removeAll(retournierteBuecher);
148
149     return leihgebuehr;
150   }
151
152 }
```

**Listing A.12:** Kunde.java

## A.2.6. KeineExemplareVerfuegbarException.java

```
1  package at.codequality.elibrary.business;
2
3  public class KeineExemplareVerfuegbarException extends Exception ↩
       {
4
5  }
```

**Listing A.13:** KeineExemplareVerfuegbarException.java

## A.2.7. BibliothekManager.java

```
1  package at.codequality.elibrary.business;
2
3  import java.util.ArrayList;
4  import java.util.List;
5
6  import at.codequality.elibrary.business.buch.Buch;
7
8  /**
9   * Bibliothek Manager, welcher als Zentrale der ↩
        Biliotheksverwaltung dient und
10  * jegliche Kunden und Buecher verwaltet.
11  *
12  * @author Jakob
13  *
14  */
15 public class BibliothekManager {
16
17   /** Manager Instanz fuer Singleton */
18   private static BibliothekManager manager;
19
```

```
20  /** Liste aller Buecher */
21  private List<Buch> buecherListe = new ArrayList<>();
22
23  /** Liste aller Kunden */
24  private List<Kunde> kundenListe = new ArrayList<>();
25
26  /**
27   * private Konstruktor fuer Singleton
28   */
29  private BibliothekManager() {
30
31  }
32
33  /**
34   * Singleton Implementierung.
35   *
36   * @return Instanz eines BibliothekManagers.
37   */
38  public static BibliothekManager getInstance() {
39    if (manager == null) {
40      manager = new BibliothekManager();
41    }
42    return manager;
43  }
44
45  /**
46   * Kunde dem Manager hinzufuegen.
47   *
48   * @param kunde
49   */
50  public void kundeHinzufuegen(Kunde kunde) {
51    kundenListe.add(kunde);
52  }
53
54  /**
55   * Buch dem Manager hinzufuegen.
56   *
57   * @param buch
58   */
59  public void buchHinzufuegen(Buch buch) {
60    buecherListe.add(buch);
61  }
62
63  /**
64   * Retourniert alle Kunden.
65   *
66   * @return Alle Kunden.
67   */
```

```
68  public List<Kunde> getAllKunden() {
69    return kundenListe;
70  }
71
72  /**
73   * Retourniert alle Buecher.
74   *
75   * @return Alle Buecher.
76   */
77  public List<Buch> getAllBuecher() {
78    return buecherListe;
79  }
80
81  /**
82   * Buch ausleihen. Methode erzeugt eine neue Leihe fuer das ↩
       Buch und fuegt diese
83   * dem Kunden zu.
84   *
85   * @param kunde
86   * @param buch
87   * @throws KeineExemplareVerfuegbarException
88   */
89  public void buchAusleihen(Kunde kunde, Buch buch) throws ↩
     KeineExemplareVerfuegbarException {
90    buch.verringereExemplare();
91    Leihe leihe = new Leihe(buch);
92    kunde.getLeihen().add(leihe);
93  }
94
95  /**
96   * Buch zurueckbringen. Methode vermerkt in der Leihe die ↩
       geliehenen Tage und
97   * erhoeht die Anzahl verfuegbarer Exemplare, da dieses Buch ↩
       nun wieder zu
98   * weiterer Ausleihe verfuegbar ist.
99   *
100  * @param leihe
101  * @param leihtage
102  */
103 public void buchZurueckgeben(Leihe leihe, int leihtage) {
104   leihe.setLeihtage(leihtage);
105 }
106
107 /**
108  * Liefert die gesamte Leihgebuehr des Kunden, nachdem dieser ↩
       einige Buecher
109  * zurueckgebracht hat. Zudem werden entsprechende Bonuspunkte ↩
       verbraucht/gesetzt
```

```
110     * und die Buecherleihen vom Kunden entfernt.
111     *
112     * @param kunde
113     * @return Gesamte Leihgebuehr des Kunden
114     */
115    public double getGesamtLeihgebuehr(Kunde kunde) {
116      return kunde.getGesamtLeihgebuehr();
117    }
118
119  }
```

**Listing A.14:** BibliothekManager.java

## A.3. eLibrary - Version 3 (Complexity)

### A.3.1. Buch.java

```
1  package at.codequality.elibrary.business;
2
3  /**
4   * Klasse eines Buches. verschiedene Typen von Buechern sind z.B.↩
        Bestseller,
5   * Standardroman, Kinderbuch.
6   *
7   * @author Jakob
8   *
9   */
10 public class Buch {
11
12   /** 1 = Bestseller , 2 = Standardroman */
13   private int buchTyp;
14
15   /** Autor des Buches. */
16   private String autor;
17
18   /** Titel des Buches. */
19   private String titel;
20
21   /** Verfuegbare Exemplare dieses Buches. */
22   private int exemplare;
23
24   /**
25    * Erstellt ein Buch mit den angegebenen Parametern.
26    *
```

```
27      * @param autor
28      * @param titel
29      * @param exemplare
30      * @param buchTyp
31      */
32     public Buch(String autor, String titel, int exemplare, int ↩
           buchTyp) {
33       super();
34       this.autor = autor;
35       this.titel = titel;
36       this.exemplare = exemplare;
37       this.buchTyp = buchTyp;
38     }
39
40     /**
41      * Liefert den Autor des Buches.
42      *
43      * @return Autor des Buches
44      */
45     public String getAuthor() {
46       return autor;
47     }
48
49     /**
50      * Liefert den Titel des Buches.
51      *
52      * @return Titel des Buches
53      */
54     public String getTitle() {
55       return titel;
56     }
57
58     /**
59      * Liefert die Anzahl der vorhandenen Exemplare.
60      *
61      * @return Anzahl der vorhandenen Exemplare
62      */
63     public int getExemplare() {
64       return exemplare;
65     }
66
67     /**
68      * Liefert die den Typ des Buches.
69      *
70      * @return Typ des Buches.
71      */
72     public int getTyp() {
73       return buchTyp;
```

```
74    }
75
76    /**
77     * Setzt die Anzahl verfuegbarer Exemplare.
78     *
79     * @param exemplare
80     *            Anzahl verfuegbarer Exemplare
81     */
82    public void setExemplare(int exemplare) {
83      this.exemplare = exemplare;
84    }
85
86    /**
87     * Erhoehen der Exemplare um den Wert 1.
88     */
89    public void erhoeheExemplare() {
90      exemplare++;
91    }
92
93    /**
94     * Verringern der Exemplare um den Wert eins.
95     *
96     * @throws KeineExemplareVerfuegbarException
97     *             Wenn keine Exemplare mehr verfuegbar sind
98     */
99    public void verringereExemplare() throws ↩
        KeineExemplareVerfuegbarException {
100     if (getExemplare() < 1) {
101       throw new KeineExemplareVerfuegbarException();
102     } else {
103       exemplare--;
104     }
105   }
106
107   /**
108    * Liefert die Leihgebuehr fuer das jeweilige Buch. Jedes Buch ↩
         muss hier ihren
109    * eigenen Algorithmus zur Berechnung der Leihgebuehren ↩
         implementieren.
110    *
111    * @param leihe
112    * @return Kosten der Leihe
113    */
114   public double getLeihgebuehr(Leihe leihe) {
115     switch (buchTyp) {
116     case 1:
117       double kosten = 5;
118       if (leihe.getLeihtage() > 3) {
```

```
119         kosten += (leihe.getLeihtage() - 3) * 2.7;
120       }
121       return kosten;
122     case 2:
123       double kosten3 = 3.5;
124       if (leihe.getLeihtage() > 6) {
125         kosten3 += (leihe.getLeihtage() - 6) * 1.5;
126       }
127       return kosten3;
128     }
129     return 0.0;
130   }
131
132   /**
133    * Liefert die Bonuspunkte fuer die uebergebene Leihe abhaengig↩
          von den Leihkosten.
134    *
135    * @param leihe
136    * @return Bonuspunkte fuer die uebergebene Leihe
137    */
138   public int getBonusPunkte(Leihe leihe) {
139     double leihK = 1;
140     if (buchTyp == 1) {
141       leihK = 5;
142       if (leihe.getLeihtage() > 3) {
143         leihK += (leihe.getLeihtage() - 3) * 2.7;
144       }
145     } else if (buchTyp == 2) {
146       leihK = 3.5;
147       if (leihe.getLeihtage() > 6) {
148         leihK += (leihe.getLeihtage() - 6) * 1.5;
149       }
150     } else {
151       leihK = 0;
152     }
153
154     if (buchTyp == 1) {
155       int punkte = 1;
156       if (leihK > 6) {
157         punkte = 3;
158       }
159       if (leihe.getLeihtage() > 4) {
160         punkte++;
161       }
162       return punkte;
163     }
164     if (getLeihgebuehr(leihe) > 6) {
165       return 3;
```

```
166     }
167     return 1;
168   }
169 }
```

**Listing A.15:** Buch.java

## A.3.2. BonusPunkteVerwaltung.java

```
1  package at.codequality.elibrary.business;
2
3  public class BonusPunkteVerwaltung {
4    private int gesamtBonuspunkte;
5    private BonusStufe bonusStufe;
6
7    public BonusPunkteVerwaltung() {
8      bonusStufe = BonusStufe.NEU;
9      gesamtBonuspunkte = 0;
10   }
11
12   public void punkteErhoehen(int punkte) {
13     gesamtBonuspunkte += punkte;
14
15     if (gesamtBonuspunkte < 10) {
16       bonusStufe = BonusStufe.NEU;
17     }
18     if (gesamtBonuspunkte > 9 && gesamtBonuspunkte < 20) {
19       bonusStufe = BonusStufe.LEVEL1;
20     }
21     if (gesamtBonuspunkte > 19 && gesamtBonuspunkte < 30) {
22       bonusStufe = BonusStufe.LEVEL2;
23     }
24     if (gesamtBonuspunkte > 29 && gesamtBonuspunkte < 30) {
25       bonusStufe = BonusStufe.LEVEL3;
26     }
27     if (gesamtBonuspunkte > 39 && gesamtBonuspunkte < 40) {
28       bonusStufe = BonusStufe.LEVEL4;
29     }
30     if (gesamtBonuspunkte > 39 && gesamtBonuspunkte < 80) {
31       bonusStufe = BonusStufe.POWERUSER;
32     }
33     if (gesamtBonuspunkte > 79) {
34       bonusStufe = BonusStufe.PREMIUM;
35     }
36   }
```

```
37
38    public void punkteAbziehen(int punkte) {
39      gesamtBonuspunkte -= punkte;
40    }
41
42    public BonusStufe getBonusLevel() {
43      return bonusStufe;
44    }
45
46    public int getGesamtBonuspunkte() {
47      return gesamtBonuspunkte;
48    }
49
50    public void stufeAktualisieren() {
51      if (gesamtBonuspunkte < 10) {
52        bonusStufe = BonusStufe.NEU;
53      }
54      if (gesamtBonuspunkte > 9 && gesamtBonuspunkte < 20) {
55        bonusStufe = BonusStufe.LEVEL1;
56      }
57      if (gesamtBonuspunkte > 19 && gesamtBonuspunkte < 30) {
58        bonusStufe = BonusStufe.LEVEL2;
59      }
60      if (gesamtBonuspunkte > 29 && gesamtBonuspunkte < 30) {
61        bonusStufe = BonusStufe.LEVEL3;
62      }
63      if (gesamtBonuspunkte > 39 && gesamtBonuspunkte < 40) {
64        bonusStufe = BonusStufe.LEVEL4;
65      }
66      if (gesamtBonuspunkte > 39 && gesamtBonuspunkte < 80) {
67        bonusStufe = BonusStufe.POWERUSER;
68      }
69      if (gesamtBonuspunkte > 79) {
70        bonusStufe = BonusStufe.PREMIUM;
71      }
72    }
73  }
```

**Listing A.16:** BonusPunkteVerwaltung.java

### A.3.3. BonusStufe.java

```
1  package at.codequality.elibrary.business;
2
3  public enum BonusStufe {
```

```
4   NEU, LEVEL1, LEVEL2, LEVEL3, LEVEL4, POWERUSER, PREMIUM
5 }
```

**Listing A.17:** BonusStufe.java

## A.3.4. Leihe.java

```
1  package at.codequality.elibrary.business;
2
3  /**
4   * Diese Klasse bildet eine Buch-Ausleihe ab. Eine Leihe besteht ↩
       immer aus einem
5   * Buch mit einer bestimmten Ausleih-Zeit (Leihtage).
6   *
7   * @author Jakob
8   *
9   */
10 public class Leihe {
11
12   private Buch buch;
13
14   private int leihtage;
15
16   /**
17    * Erzeugt eine Leihe fuer ein bestimmtes Buch.
18    *
19    * @param buch
20    *            Buch, welches ausgeliehen wird
21    */
22   public Leihe(Buch buch) {
23     this.buch = buch;
24   }
25
26   /**
27    * Liefert die Anzahl der Tage, an welchen das Buch ausgeliehen↩
          war.
28    *
29    * @return Anzahl der Leihtage
30    */
31   public int getLeihtage() {
32     return leihtage;
33   }
34
35   /**
```

```
36     * Setzt die Anzahl der Tage, an welchen das Buch ausgeliehen ↩
           war.
37     *
38     * @param leihtage
39     *            Leihtage
40     */
41   public void setLeihtage(int leihtage) {
42     this.leihtage = leihtage;
43   }
44
45   /**
46    * Liefert das geliehene Buch.
47    *
48    * @return Das ausgeliehene Buch
49    */
50   public Buch getBuch() {
51     return buch;
52   }
53
54   /**
55    * Liefert die Leihgebuehr fuer die Leihe des Buches.
56    *
57    * @return Leihgebuehr
58    */
59   public double getLeihgebuehr() {
60     return getBuch().getLeihgebuehr(this);
61   }
62
63   /**
64    * Liefert die Bonuspunkte fuer die Leihe des Buches.
65    *
66    * @return Bonuspunkte
67    */
68   public int getBonusPunkte() {
69     return getBuch().getBonusPunkte(this);
70   }
71
72 }
```

**Listing A.18:** Leihe.java

## A.3.5. Kunde.java

```
1 package at.codequality.elibrary.business;
2
```

```
3  import java.util.ArrayList;
4  import java.util.List;
5
6  /**
7   * Die Klasse Kunde repraesentiert einen Kunden aus dem echten ↩
       Leben. Ein Kunde
8   * hat einen Namen und kann mehrere Ausleihungen haben. Zudem ↩
       sammeln Kunden
9   * Bonuspunkte fuer ausgeliehene Buecher.
10  *
11  * @author Jakob
12  *
13  */
14 public class Kunde {
15
16   /** Name des Kunden. */
17   private String name;
18
19   /** Gesammelte Bonuspunkte des Kunden. */
20   private int bonuspunkte;
21
22   /** Liste der Leihen des Kunden. */
23   private List<Leihe> leihen = new ArrayList<>();
24
25   /** Verwaltung der Bonuspunkte eines Kunden */
26   private BonusPunkteVerwaltung bonusVerwaltung = new ↩
       BonusPunkteVerwaltung();
27
28   /**
29    * Erzeugt einen neuen Kunden mittels angegebenem Namen.
30    *
31    * @param name
32    *            Name des Kunden
33    */
34   public Kunde(String name) {
35     this.name = name;
36   }
37
38   /**
39    * Liefert den Namen des Kunden.
40    *
41    * @return Name des Kunden
42    */
43   public String getName() {
44     return name;
45   }
46
47   /**
```

```
48    * Liefert die Bonuspunkte, welche bis zu diesem Zeitpunkt ↩
          gesammelt wurden.
49    *
50    * @return Gesammelte Bonuspunkte
51    */
52   public int getBonuspunkte() {
53     return bonuspunkte;
54   }
55
56   /**
57    * Liefert die Liste der Ausleihungen des Kunden.
58    *
59    * @return Leihen des Kunden
60    */
61   public List<Leihe> getLeihen() {
62     return leihen;
63   }
64
65   /**
66    * Liefert die Bonuspunkte Verwaltung des Kunden.
67    *
68    * @return Bonuspunkteverwaltung des Kunden
69    */
70   public BonusPunkteVerwaltung getBonusVerwaltung() {
71     if (bonusVerwaltung != null) {
72       bonusVerwaltung = new BonusPunkteVerwaltung();
73     }
74     return bonusVerwaltung;
75   }
76
77   /**
78    * Bonuspunkte beim retournieren ausgeliehener Buecher dem ↩
          Kunden zuweisen.
79    * Hierfuer werden die Bonuspunkte der einzelnen Leihen ↩
          akkumuliert.
80    *
81    * @param retournierteBuecher
82    */
83   public void bonuspunkteZuweisen(List<Leihe> retournierteBuecher↩
       ) {
84     for (Leihe leihe : retournierteBuecher) {
85       bonuspunkte += leihe.getBonusPunkte();
86     }
87   }
88
89   /**
90    * Verringert die gesamten Bonuspunkte des Kunden um die ↩
          angegebene Anzahl. Das
```

```
91    * ist zb. der Fall, wenn ein Kunde Bonuspunkte fuer ↩
          Ermaessigungen verbraucht.
92    *
93    * @param punkte
94    *            Punkte, welche von Punktekonto abgezogen werden ↩
          sollen.
95    */
96   public void bonuspunkteAbziehen(int punkte) {
97     if (!(punkte > this.bonuspunkte)) {
98       bonuspunkte -= punkte;
99     }
100  }
101
102  /**
103   * Liefert die retournierten Buecher eines Kunden. Hierfuer ↩
          werden Buecher gezaehlt,
104   * welche eine Ausleihdauer > 0 aufweisen. Die Ausleihdauer ↩
          wird beim
105   * Zurueckbringen von Buechern gesetzt.
106   *
107   * @return Retournierte Buecher
108   */
109  public List<Leihe> getRetournierteBuecher() {
110    ArrayList<Leihe> retournierteBuecher = new ArrayList<>();
111    for (Leihe leihe : getLeihen()) {
112      if (leihe.getLeihtage() > 0) {
113        retournierteBuecher.add(leihe);
114      }
115    }
116    return retournierteBuecher;
117  }
118
119  /**
120   * Bonuspunkte, welche bisher gesammelt wurden, werden in der ↩
          Bonusverwaltung
121   * verbraucht.
122   */
123  private void bonusVerbrauchen() {
124    int punkte = this.getBonusVerwaltung().getGesamtBonuspunkte()↩
          ;
125    int abzugpunkte;
126    if (punkte > 10) {
127      abzugpunkte = 10;
128    } else {
129      abzugpunkte = punkte;
130    }
131    this.getBonusVerwaltung().punkteAbziehen(abzugpunkte);
132  }
```

```
133
134   /**
135    * Bonuspunkte, welche bisher gesammelt wurden, werden in der ↩
          Bonusverwaltung
136    * verbraucht.
137    *
138    * @param punkte
139    */
140   private void bonusVerbrauchen(int punkte) {
141     int gesamtBonuspunkte = this.getBonusVerwaltung().↩
          getGesamtBonuspunkte();
142     int abzugpunkte;
143     if (gesamtBonuspunkte > punkte) {
144       abzugpunkte = punkte;
145     } else {
146       abzugpunkte = gesamtBonuspunkte;
147     }
148     this.getBonusVerwaltung().punkteAbziehen(abzugpunkte);
149   }
150
151   /**
152    * Liefert die Leihgebuehr des billigsten, ausgeliehenen Buches↩
          .
153    *
154    * @return Leihgebuehr des billigsten Buches
155    */
156   public double getKostenDesBilligstenBuches() {
157     double minKosten = 0;
158     for (Leihe leihe : getLeihen()) {
159       if (minKosten == 0) {
160         minKosten = leihe.getLeihgebuehr();
161       }
162       if (leihe.getLeihgebuehr() < minKosten) {
163         minKosten = leihe.getLeihgebuehr();
164       }
165     }
166     return minKosten;
167   }
168
169   /**
170    * Liefert die gesamte Leihgebuehr fuer alle zurueckgebrachten ↩
          Buecher. Zudem werden
171    * entsprechende Bonuspunkte verbraucht/gesetzt und die ↩
          Buecherleihen vom Kunden
172    * entfernt.
173    *
174    * @return Gesamte Leihgebuehr
175    */
```

```
176   public double getGesamtLeihgebuehr() {
177     List<Leihe> retournierteBuecher = getRetournierteBuecher();
178
179     int leihgebuehr = 0;
180     for (Leihe leihe : getLeihen()) {
181       leihgebuehr += leihe.getLeihgebuehr();
182     }
183
184     double tmpGebuehr = leihgebuehr;
185     boolean fuerBonusVerwenden = false;
186
187     for (Leihe leihe2 : retournierteBuecher) {
188       if (leihe2.getLeihgebuehr() > 4 && leihe2.getLeihtage() > 2↩
              && leihe2.getBuch().getTyp() == 2) {
189         fuerBonusVerwenden = true;
190       }
191       if (leihe2.getLeihgebuehr() > 1 && leihe2.getLeihtage() > 3↩
              && leihe2.getBuch().getTyp() == 1) {
192         fuerBonusVerwenden = true;
193       }
194       if (leihe2.getLeihgebuehr() > 5 && leihe2.getLeihtage() > 2↩
              && leihe2.getBuch().getTyp() == 4) {
195         fuerBonusVerwenden = true;
196       }
197       if (fuerBonusVerwenden) {
198         int punkteFuerBonuszuweisung = leihe2.getBonusPunkte();
199         BonusStufe bonusStufe = this.getBonusVerwaltung().↩
              getBonusLevel();
200         if (bonusStufe == BonusStufe.NEU) {
201           punkteFuerBonuszuweisung = punkteFuerBonuszuweisung / ↩
                3;
202         }
203         if (bonusStufe == BonusStufe.LEVEL1 || bonusStufe == ↩
              BonusStufe.LEVEL2
204             || bonusStufe == BonusStufe.LEVEL3) {
205           punkteFuerBonuszuweisung = punkteFuerBonuszuweisung / ↩
                1;
206         }
207         if (bonusStufe == BonusStufe.POWERUSER) {
208           punkteFuerBonuszuweisung = (int) Math.round(↩
                punkteFuerBonuszuweisung * 1.5);
209         }
210         if (bonusStufe == BonusStufe.PREMIUM) {
211           punkteFuerBonuszuweisung = (int) Math.round(↩
                punkteFuerBonuszuweisung * 2.5);
212         }
213         bonusVerwaltung.punkteErhoehen(punkteFuerBonuszuweisung);
214       }
```

```
215
216       int bonuspunkte = getBonusVerwaltung().getGesamtBonuspunkte↩
              ();
217
218       if (bonuspunkte < 40) {
219         // zu wenig Punkte gesammelt
220       }
221       if (bonuspunkte > 39 && bonuspunkte < 80) {
222         float abzugBetrag = (float) tmpGebuehr / 10;
223       }
224       if (bonuspunkte > 80) {
225         float abzugBetrag = (float) tmpGebuehr / 20;
226       }
227       fuerBonusVerwenden = false;
228     }
229
230     while (getBonuspunkte() > 9 && leihgebuehr > 1) {
231       leihgebuehr -= 1;
232       bonuspunkteAbziehen(10);
233       bonusVerbrauchen(10);
234     }
235
236     if (leihgebuehr > 20) {
237       leihgebuehr -= getKostenDesBilligstenBuches();
238       bonusVerbrauchen();
239     }
240
241     bonuspunkteZuweisen(retournierteBuecher);
242     getLeihen().removeAll(retournierteBuecher);
243
244     return leihgebuehr;
245   }
246
247 }
```

**Listing A.19:** Kunde.java

## A.3.6. KeineExemplareVerfuegbarException.java

```
1 package at.codequality.elibrary.business;
2
3 public class KeineExemplareVerfuegbarException extends Exception ↩
      {
4
```

```
5 }
```

**Listing A.20:** KeineExemplareVerfuegbarException.java

## A.3.7. BibliothekManager.java

```
1  package at.codequality.elibrary.business;
2
3  import java.util.ArrayList;
4  import java.util.List;
5
6  /**
7   * Bibliothek Manager, welcher als Zentrale der ↩
       Biliotheksverwaltung dient und
8   * jegliche Kunden und Buecher verwaltet.
9   *
10  * @author Jakob
11  *
12  */
13 public class BibliothekManager {
14
15   /** Manager Instanz fuer Singleton */
16   private static BibliothekManager manager;
17
18   /** Liste aller Buecher */
19   private List<Buch> buecherListe = new ArrayList<>();
20
21   /** Liste aller Kunden */
22   private List<Kunde> kundenListe = new ArrayList<>();
23
24   /**
25    * private Konstruktor fuer Singleton
26    */
27   private BibliothekManager() {
28
29   }
30
31   /**
32    * Singleton Implementierung.
33    *
34    * @return Instanz eines BibliothekManagers.
35    */
36   public static BibliothekManager getInstance() {
37     if (manager == null) {
38       manager = new BibliothekManager();
```

```
39      }
40      return manager;
41    }
42
43    /**
44     * Kunde dem Manager hinzufuegen.
45     *
46     * @param kunde
47     */
48    public void kundeHinzufuegen(Kunde kunde) {
49      kundenListe.add(kunde);
50    }
51
52    /**
53     * Buch dem Manager hinzufuegen.
54     *
55     * @param buch
56     */
57    public void buchHinzufuegen(Buch buch) {
58      buecherListe.add(buch);
59    }
60
61    /**
62     * Retourniert alle Kunden.
63     *
64     * @return Alle Kunden.
65     */
66    public List<Kunde> getAllKunden() {
67      return kundenListe;
68    }
69
70    /**
71     * Retourniert alle Buecher.
72     *
73     * @return Alle Buecher.
74     */
75    public List<Buch> getAllBuecher() {
76      return buecherListe;
77    }
78
79    /**
80     * Buch ausleihen. Methode erzeugt eine neue Leihe fuer das ↩
         Buch und fuegt diese
81     * dem Kunden zu.
82     *
83     * @param kunde
84     * @param buch
85     * @throws KeineExemplareVerfuegbarException
```

```
86      */
87    public void buchAusleihen(Kunde kunde, Buch buch) throws ↩
          KeineExemplareVerfuegbarException {
88      if (buch.getExemplare() < 1) {
89        throw new KeineExemplareVerfuegbarException();
90      } else {
91        buch.setExemplare(buch.getExemplare() - 1);
92      }
93      Leihe leihe = new Leihe(buch);
94      kunde.getLeihen().add(leihe);
95    }
96
97    /**
98     * Buch zurueckbringen. Methode vermerkt in der Leihe die ↩
            geliehenen Tage und
99     * erhoeht die Anzahl verfuegbarer Exemplare, da dieses Buch ↩
            nun wieder zu
100    * weiterer Ausleihe verfuegbar ist.
101    *
102    * @param leihe
103    * @param leihtage
104    */
105   public void buchZurueckgeben(Leihe leihe, int leihtage) {
106     leihe.setLeihtage(leihtage);
107   }
108
109   /**
110    * Liefert die gesamte Leihgebuehr des Kunden, nachdem dieser ↩
           einige Buecher
111    * zurueckgebracht hat. Zudem werden entsprechende Bonuspunkte ↩
           verbraucht/gesetzt
112    * und die Buecherleihen vom Kunden entfernt.
113    *
114    * @param kunde
115    * @return Gesamte Leihgebuehr des Kunden
116    */
117   public double getGesamtLeihgebuehr(Kunde kunde) {
118     return kunde.getGesamtLeihgebuehr();
119   }
120
121 }
```

**Listing A.21:** BibliothekManager.java

# A.4. eLibrary - Version 4 (Bad Quality)

## A.4.1. Buch.java

```
1  package at.codequality.elibrary.business;
2
3  public class Buch {
4
5    private int buchTyp;
6
7    private String a;
8
9    private String t;
10
11   private int anz;
12
13   public Buch(String a, String t, int menge, int buchTyp) {
14     super();
15     this.a = a;
16     this.t = t;
17     this.anz = menge;
18     this.buchTyp = buchTyp;
19   }
20
21   public String getAuthor() {
22     return a;
23   }
24
25   public String getTitle() {
26     return t;
27   }
28
29   public int getAnz() {
30     return anz;
31   }
32
33   public int getTyp() {
34     return buchTyp;
35   }
36
37   public void setAnz(int x) {
38     this.anz = x;
39   }
40
41   public void erhoehe() {
```

```
42      anz++;
43    }
44
45    public void abziehen() throws LeerException {
46      if (getAnz() < 1) {
47        throw new LeerException();
48      } else {
49        anz--;
50      }
51    }
52
53    public double zahlung(Leihe leihe) {
54      switch (buchTyp) {
55      case 1:
56        double kost = 5;
57        if (leihe.getLeihT() > 3) {
58          kost += (leihe.getLeihT() - 3) * 2.7;
59        }
60        return kost;
61      case 2:
62        double kost3 = 3.5;
63        if (leihe.getLeihT() > 6) {
64          kost3 += (leihe.getLeihT() - 6) * 1.5;
65        }
66        return kost3;
67      }
68      return 0.0;
69    }
70
71    public int bonus(Leihe leihe) {
72      double leihK = 1;
73      if (buchTyp == 1) {
74        leihK = 5;
75        if (leihe.getLeihT() > 3) {
76          leihK += (leihe.getLeihT() - 3) * 2.7;
77        }
78      } else if (buchTyp == 2) {
79        leihK = 3.5;
80        if (leihe.getLeihT() > 6) {
81          leihK += (leihe.getLeihT() - 6) * 1.5;
82        }
83      } else {
84        leihK = 0;
85      }
86
87      if (buchTyp == 1) {
88        int punkte = 1;
89        if (leihK > 6) {
```

```
90          punkte = 3;
91        }
92        if (leihe.getLeihT() > 4) {
93          punkte++;
94        }
95        return punkte;
96      }
97      if (zahlung(leihe) > 6) {
98        return 3;
99      }
100     return 1;
101   }
102 }
```

**Listing A.22:** Buch.java

## A.4.2. BonusPunkteControl.java

```
1  package at.codequality.elibrary.business;
2
3  public class BonusPunkteControl {
4    private int gesamt;
5    private BonusStufe bonusS;
6
7    public BonusPunkteControl() {
8      bonusS = BonusStufe.NEU;
9      gesamt = 0;
10   }
11
12   public void checkPunkte(int p) {
13     gesamt += p;
14
15     if (gesamt < 10) {
16       bonusS = BonusStufe.NEU;
17     }
18     if (gesamt > 9 && gesamt < 20) {
19       bonusS = BonusStufe.LEVEL1;
20     }
21     if (gesamt > 19 && gesamt < 30) {
22       bonusS = BonusStufe.LEVEL2;
23     }
24     if (gesamt > 29 && gesamt < 30) {
25       bonusS = BonusStufe.LEVEL3;
26     }
27     if (gesamt > 39 && gesamt < 40) {
```

```
28        bonusS = BonusStufe.LEVEL4;
29      }
30      if (gesamt > 39 && gesamt < 80) {
31        bonusS = BonusStufe.POWERUSER;
32      }
33      if (gesamt > 79) {
34        bonusS = BonusStufe.PREMIUM;
35      }
36    }
37
38    public void checkPunkteToRemove(int punkte) {
39      gesamt -= punkte;
40    }
41
42    public BonusStufe getBonusLevel() {
43      return bonusS;
44    }
45
46    public int getGesamt() {
47      return gesamt;
48    }
49
50    public void checkStufe() {
51      if (gesamt < 10) {
52        bonusS = BonusStufe.NEU;
53      }
54      if (gesamt > 9 && gesamt < 20) {
55        bonusS = BonusStufe.LEVEL1;
56      }
57      if (gesamt > 19 && gesamt < 30) {
58        bonusS = BonusStufe.LEVEL2;
59      }
60      if (gesamt > 29 && gesamt < 30) {
61        bonusS = BonusStufe.LEVEL3;
62      }
63      if (gesamt > 39 && gesamt < 40) {
64        bonusS = BonusStufe.LEVEL4;
65      }
66      if (gesamt > 39 && gesamt < 80) {
67        bonusS = BonusStufe.POWERUSER;
68      }
69      if (gesamt > 79) {
70        bonusS = BonusStufe.PREMIUM;
71      }
72    }
73  }
```

**Listing A.23:** BonusPunkteControl.java

### A.4.3. BonusStufe.java

```
1  package at.codequality.elibrary.business;
2
3  public enum BonusStufe {
4    NEU, LEVEL1, LEVEL2, LEVEL3, LEVEL4, POWERUSER, PREMIUM
5  }
```

**Listing A.24:** BonusStufe.java

### A.4.4. Leihe.java

```
1  package at.codequality.elibrary.business;
2
3  public class Leihe {
4
5    private Buch b;
6
7    private int leihTage;
8
9    public Leihe(Buch buch) {
10     this.b = buch;
11   }
12
13   public int getLeihT() {
14     return leihTage;
15   }
16
17   public void setLeihT(int lt) {
18     this.leihTage = lt;
19   }
20
21   public Buch getBu() {
22     return b;
23   }
24
25   public double rechnen() {
26     return getBu().zahlung(this);
27   }
28
29   public int checkBonus() {
30     return getBu().bonus(this);
31   }
32
```

```
33 }
```

**Listing A.25:** Leihe.java

### A.4.5. Kunde.java

```
1  package at.codequality.elibrary.business;
2
3  import java.util.ArrayList;
4  import java.util.List;
5
6  public class Kunde {
7
8    private String name;
9
10   private int bonus;
11
12   private List<Leihe> lList = new ArrayList<>();
13
14   private BonusPunkteControl bonusCtr = new BonusPunkteControl();
15
16   public Kunde(String n) {
17     this.name = n;
18   }
19
20   public String getName() {
21     return name;
22   }
23
24   public int bonus1() {
25     return bonus;
26   }
27
28   public List<Leihe> getBue() {
29     return lList;
30   }
31
32   public BonusPunkteControl getBonusCtr() {
33     if (bonusCtr != null) {
34       bonusCtr = new BonusPunkteControl();
35     }
36     return bonusCtr;
37   }
38
39   public void checkBonus(List<Leihe> bList) {
```

```
40      for (Leihe l : bList) {
41        bonus += l.checkBonus();
42      }
43    }
44
45    public void verkleinereMenge(int x) {
46      if (!(x > this.bonus)) {
47        bonus -= x;
48      }
49    }
50
51    public List<Leihe> getBuecher() {
52      ArrayList<Leihe> bSet = new ArrayList<>();
53      for (Leihe l : getBue()) {
54        if (l.getLeihT() > 0) {
55          bSet.add(l);
56        }
57      }
58      return bSet;
59    }
60
61    private void bonusVerbrauchen() {
62      int punkte = this.getBonusCtr().getGesamt();
63      int pVerbrauch;
64      if (punkte > 10) {
65        pVerbrauch = 10;
66      } else {
67        pVerbrauch = punkte;
68      }
69      this.getBonusCtr().checkPunkteToRemove(pVerbrauch);
70    }
71
72    private void bonusVerbrauchen(int p) {
73      int gesamt = this.getBonusCtr().getGesamt();
74      int pVerbrauch;
75      if (gesamt > p) {
76        pVerbrauch = p;
77      } else {
78        pVerbrauch = gesamt;
79      }
80      this.getBonusCtr().checkPunkteToRemove(pVerbrauch);
81    }
82
83    public double getBuchFuerVerringerung() {
84      double kosten = 0;
85      for (Leihe l : getBue()) {
86        if (kosten == 0) {
87          kosten = l.rechnen();
```

```
88        }
89        if (l.rechnen() < kosten) {
90          kosten = l.rechnen();
91        }
92      }
93      return kosten;
94    }
95
96    public double findZahlung() {
97      List<Leihe> bue = getBuecher();
98
99      int kosten = 0;
100     for (Leihe le : getBue()) {
101       kosten += le.rechnen();
102     }
103
104     double tmpGebuehr = kosten;
105     boolean useBonus = false;
106
107     for (Leihe leihe2 : bue) {
108       if (leihe2.rechnen() > 4 && leihe2.getLeihT() > 2 && leihe2↩
            .getBu().getTyp() == 2) {
109         useBonus = true;
110       }
111       if (leihe2.rechnen() > 1 && leihe2.getLeihT() > 3 && leihe2↩
            .getBu().getTyp() == 1) {
112         useBonus = true;
113       }
114       if (leihe2.rechnen() > 5 && leihe2.getLeihT() > 2 && leihe2↩
            .getBu().getTyp() == 4) {
115         useBonus = true;
116       }
117       if (useBonus) {
118         int punkteFuerBonuszuweisung = leihe2.checkBonus();
119         BonusStufe bonusStufe = this.getBonusCtr().getBonusLevel↩
              ();
120         if (bonusStufe == BonusStufe.NEU) {
121           punkteFuerBonuszuweisung = punkteFuerBonuszuweisung / ↩
                3;
122         }
123         if (bonusStufe == BonusStufe.LEVEL1 || bonusStufe == ↩
              BonusStufe.LEVEL2
124             || bonusStufe == BonusStufe.LEVEL3) {
125           punkteFuerBonuszuweisung = punkteFuerBonuszuweisung / ↩
                1;
126         }
127         if (bonusStufe == BonusStufe.POWERUSER) {
```

```
128           punkteFuerBonuszuweisung = (int) Math.round(↩
                 punkteFuerBonuszuweisung * 1.5);
129         }
130         if (bonusStufe == BonusStufe.PREMIUM) {
131           punkteFuerBonuszuweisung = (int) Math.round(↩
                 punkteFuerBonuszuweisung * 2.5);
132         }
133         bonusCtr.checkPunkte(punkteFuerBonuszuweisung);
134       }
135
136       int bonuspunkte = getBonusCtr().getGesamt();
137
138       if (bonuspunkte < 40) {
139         // zu wenig Punkte gesammelt
140       }
141       if (bonuspunkte > 39 && bonuspunkte < 80) {
142         float abzugBetrag = (float) tmpGebuehr / 10;
143       }
144       if (bonuspunkte > 80) {
145         float abzugBetrag = (float) tmpGebuehr / 20;
146       }
147       useBonus = false;
148     }
149
150     while (bonus1() > 9 && kosten > 1) {
151       kosten -= 1;
152       verkleinereMenge(10);
153       bonusVerbrauchen(10);
154     }
155
156     if (kosten > 20) {
157       kosten -= getBuchFuerVerringerung();
158       bonusVerbrauchen();
159     }
160
161     checkBonus(bue);
162     getBue().removeAll(bue);
163
164     return kosten;
165   }
166
167 }
```

**Listing A.26:** Kunde.java

### A.4.6. LeerException.java

```
1  package at.codequality.elibrary.business;
2
3  public class LeerException extends Exception {
4
5  }
```

**Listing A.27:** LeerException.java

## A.4.7. MainController.java

```
1  package at.codequality.elibrary.business;
2
3  import java.util.ArrayList;
4  import java.util.List;
5
6  public class MainController {
7
8    private static MainController mc;
9
10   private List<Buch> bList = new ArrayList<>();
11
12   private List<Kunde> kList = new ArrayList<>();
13
14   private MainController() {
15   }
16
17   public static MainController getInstance() {
18     if (mc == null) {
19       mc = new MainController();
20     }
21     return mc;
22   }
23
24   public void kundeErweitern(Kunde kunde) {
25     kList.add(kunde);
26   }
27
28   public void buchErweitern(Buch buch) {
29     bList.add(buch);
30   }
31
32   public List<Kunde> getKunde() {
33     return kList;
34   }
35
```

```
36    public List<Buch> getBuech() {
37      return bList;
38    }
39
40    public void buchNehmen(Kunde x, Buch y) throws LeerException {
41      if (y.getAnz() < 1) {
42        throw new LeerException();
43      } else {
44        y.setAnz(y.getAnz() - 1);
45      }
46      Leihe l = new Leihe(y);
47      x.getBue().add(l);
48    }
49
50    public void buchGeben(Leihe l, int lt) {
51      l.setLeihT(lt);
52    }
53
54    public double findZahlung(Kunde k) {
55      return k.findZahlung();
56    }
57
58  }
```

**Listing A.28:** MainController.java

## A.5. Aufgabenstellung Teilnehmer

Studie über die Auswirkung unterschiedlicher Codequalität in Softwareprojekten

November 2017

# 1 E-Bibliothek

## 1.1 Einleitung

Herzlich Willkommen zu dieser Studie. Sie sind von nun an für die Wartung und Entwicklung der Anwendung *eLibrary* zuständig, da der bislang verantwortliche Mitarbeiter gekündigt hat. Zur Einarbeitung in die bestehende Software steht Ihnen dieses Dokument sowie der Java-Sourcecode zur Verfügung. Ein Kunde hat einige Fehler gefunden und einen Wunsch für eine neue Funktionalität geäußert. Aufgrund der Wichtigkeit dieser Anforderungen sollen Sie so rasch wie möglich Ihre Arbeit aufnehmen.

Gutes Gelingen!

## 1.2 Business Regeln

Bei der vorliegenden Anwendung *eLibrary* handelt es sich um eine elektronische Bibliotheksverwaltung für Bücher aller Art. Mit dieser Anwendung werden Kunden und Bücher sowie Buchleihen verwaltet und Abrechnungen sowie Bonuspunkte kalkuliert. In der finalen Version soll die Anwendung drei Buchtypen beinhalten (zwei Buchtypen sind bereits integriert).

1. Bestseller: Neue und stark nachgefragte Bücher
2. Standardroman: Bücher, welche schon länger am Markt sind
3. Kinderbuch: Bücher für Kinder (noch nicht implementiert)

Leihkosten und Bonuspunkte werden nach folgendem Berechnungsschema errechnet.

**Bestseller**

- die ersten 3 Tage pauschal *5€*, danach *2.7€* / Tag *zusätzlich*

1

- bei Leihkosten von mehr als *6€* bekommt man 3 Bonuspunkte, sonst 1 Bonuspunkt
- bei Verleihdauer von 5 Tagen oder mehr bekommt man 1 Bonuspunkt *zusätzlich*

**Standardroman**

- die ersten 6 Tage pauschal *3.5€*, danach *1.5€* / Tag *zusätzlich*
- bei Leihkosten von mehr als *6€* bekommt man 3 Bonuspunkte, sonst 1 Bonuspunkt

**Kinderbuch**

- die ersten 5 Tage pauschal*2.5€*, danach *2.5€* / Tag *zusätzlich*
- bei Leihkosten von mehr als *6€* bekommt man 3 Bonuspunkte, sonst 1 Bonuspunkt

**Ermäßigungen**

- Hat ein Kunde mehr als 10 Bonuspunkte auf seinem Konto gesammelt, wird pro 10 Bonuspunkten *1€* vom Rechnungsbetrag abgezogen.
- Ist der endgültige Leihbetrag größer als *20€*, werden zusätzlich die Gebühren für das günstigste Buch erlassen.

Von jedem Buch in der Bibliothek gibt es eine definierte Menge an verfügbaren Exemplaren. Folgerichtig kann ein Buch nur so oft verliehen werden, solange genügend Exemplare vorhanden sind. Werden Bücher von Kunden retourniert, wird die Anzahl an Leihtagen eingegeben und danach kann eine Rechnung (Rechnungsbetrag) aller retournierten Bücher ausgegeben werden. Ab dem Zeitpunkt der erfolgreichen Rechnungsstellung gelten die Bücher als retourniert.

### 1.3 Technische Rahmenbedingungen

Bei der Anwendung *eLibrary* handelt es sich im vorliegenden Beispiel um eine einfache Konsolenanwendung. In der Klasse *Main* finden Sie eine beispielhafte Ausführung der Anwendung mit Beispieldatensätzen und Szenarien.

Als zentrale Steuerung der Anwendung dient die Klasse *BibliotheksManager (bzw. MainController)*, welcher alle Kunden und Bücher hält und jegliche Anfragen aus dem User Interface entgegen nimmt. Für jedes ausgeliehene Buch wird ein Objekt der Klasse *Leihe* erstellt und dem jeweiligen Kunden hinzugefügt.

**ACHTUNG:** Das User Interface ist nicht Teil dieser Anwendung, da es von einem anderen Team erstellt und erweitert wird. Da diese Anwendung von unterschiedlichen Projekten verwendet wird, ist es wichtig, dass Klassen- und Methodennamen sowie Methodensignaturen **NICHT** geändert werden, es steht Ihnen jedoch frei den Inhalt bestehender Methoden zu verändern sowie zusätzliche Methoden zu erstellen.

### 1.4 Anforderung des Kunden

Nachfolgend sind die erwähnten Kundenwünsche (Erweiterung und Fehlerbehebung), welche eine rasche Bearbeitung benötigen, angeführt.

**Erweiterung**

- In der derzeitigen Version der Anwendung sind lediglich die Buch-Typen „*Bestseller*“ und „*Standardroman*“ implementiert. Nach erhöhter Nachfrage im Bereich von Kinderbüchern soll nun ein neuer Typ „*Kinderbuch*“ mit eigener Berechnung in der Software integriert werden.
- Für die Umsetzung soll das zuvor beschriebene Preis- und Bonuspunkteschema herangezogen werden.

**Fehler 1**

- Der Kunde *Franz* leiht sich die Bücher „*Tyll*“, „*The Lord of the Rings*“ und „*Der letzte Wunsch*“.
- Nach 3 Tagen bringt er alle Bücher wieder zurück.
- Bei der Rechnungsausstellung wird der Endbetrag von *13*€ ausgegeben, obwohl laut Preis- und Rabattschema ein Preis von *13.5*€ richtig wäre.

**Fehler 2**

- Der Kunde *Franz* leiht sich das Buch „*Der letzte Wunsch*“ aus.
- Der Kunde *Franz* hat das Buch noch nicht zurückgebracht doch als ein Mitarbeiter des Bücherverleihs unbeabsichtigt auf Rechnungsstellung für diesen Kunden klickt wird ein Betrag von *3.5*€ ausgegeben, obwohl zu diesem Zeitpunkt eigentlich der Betrag *0.0*€ richtig wäre.

**Fehler 3**

- Der Kunde *Franz* leiht sich die Bücher „*Honigtod*“ und „*Der Herr der Ringe*“ aus.
- *Johannes* leiht sich am selben Tag die Bücher „*Der Herr der Ringe*“ sowie „*Tyll*“ aus.
- Nach 3 Leihtagen bringt *Franz* das Buch „*Der Herr der Ringe*“ wieder zurück und bezahlt seine Rechnung.
- *Sandra* kommt in den Bücherladen und möchte das Buch „*Der Herr der Ringe*“ mitnehmen, jedoch meldet die elektronische Bibliotheksverwaltung den Fehler ”Keine Exemplare verfügbar”, obwohl sie das letzte Buch in den Händen hält. (Insgesamt hat die Bibliothek 2 Ausgaben dieses Buches)

Anmerkung: Bei den in Fehler 1-3 genannten Büchern handelt es sich bei „*The Lord of the Rings*“ und „*Tyll*“ um Bestseller und bei „*Honigtod*“ und „*Der letzte Wunsch*“ um Standardromane. Zusätzlich hatten alle Kunden eine Anzahl von 0 Bonuspunkten verbucht.

3

## A.6. Auswertungstabelle Experteninterviews

| Exp. | Nr. | Paraphrase | Generalisierung |
|---|---|---|---|
| A | 1 | Denke vorhandene Main Methode wurde zu Debugging eingesetzt | Einfluss der Ergebnisse durch Main-Debugging |
| A | 2 | Habe zuerst versucht den Code zu lesen, danach auf Debugging umgestiegen | Erklärung der Ergebnisse möglicherweise durch Debugging |
| A | 3 | Vermutlich ist das Ergebnis daher sehr ähnlich, da die meisten direkt Debugging verwendet haben, ohne den Code zu verstehen | Einfluss auf Ergebnisse durch Debugging |
| A | 4 | Ev. sollten die Kandidaten die Main Methode selbst schreiben, um den Code interpretieren zu müssen | Einfluss auf Ergebnisse durch Debugging |
| A | 5 | Javadoc nicht verwendet, da meist als falsch angenommen, weil es meist veraltet ist | Einfluss von Javadoc fraglich |
| A | 6 | Größe des Projektes ist entscheidend für die Auswirkungen von Codequalität auf die Wartung | Größeres Projekt wäre für Untersuchung wichtig |
| A | 7 | Unerfahrene Entwickler können eventuell mit schlechterem Code besser umgehen, weil sie Hacking und schlechtes Naming gewohnt sind | Anfänger können mit schlechtem Code besser zurecht? |
| A | 8 | Mehrere unterschiedliche Programmversionen wäre in einer umfangreicheren Untersuchung von Vorteil, weil dadurch die Skills der einzelnen Entwickler vergleichbar wären | Validierung der Teilnehmer durch mehrere Programmversionen |
| A | 9 | Die unterschiedliche Erfahrung der Teilnehmer wäre interessant herauszuarbeiten, ob es hier Unterschiede gibt | Erfahrung der Teilnehmer clustern |
| A | 10 | Die Erkenntnis von Bad Fixes finde ich sehr spannend und zutreffend und ich halte es für sehr realistisch | Bad Fix durch Bad Code |

| | | | |
|---|---|---|---|
| A | 11 | Ergebnisse zeigen, dass Tests und Continuous Integration sehr wertvoll sein können, für gezieltes Debugging und zusätzlich eine ausführbare Dokumentation darstellen | Umfassende Tests sind entscheidend für schnelles Debugging und Programmverständnis |
| B | 1 | Habe für die Fehlersuche hauptsächlich Debugging der Main Klasse verwendet, womit es dann relativ schnell zu finden ging, auch wenn der Code sehr schlecht war | Debugging war für Fehlersuche trotz schlechten Codes gut geeignet |
| B | 2 | Denke die Plausibilität ist durchaus gegeben, da sich Entwickler bis zu einem gewissen Maß an schlechten Code anpassen können und speziell durch Analyse-Methoden (Debugging) Fehler einfach finden | Ergebnisse plausibel, da Anpassung an schlechten Code gegeben und Codequalität hat wenig Auswirkungen auf gezieltes Debugging |
| B | 3 | Begründung der Plausibilität der Ergebnisse wie schon vorher erwähnt durch Debugging und Anpassungsfähigkeit | Debugging und Anpassungsfähigkeit begründen die Ergebnisse |
| B | 4 | Denke, dass es bei größerem Umfang der Anwendung und steigender Komplexität zu wesentlich mehr Aufwand und mehreren Bad Fixes kommen wird | Größe und Umfang sowie Komplexität einer Anwendung entscheident für Auswirkungen von Codequalität |
| B | 5 | Wenn beispielsweise Duplicate Code nicht nur in einer Methode sondern über die gesamte Anwendung hinweg vorkommt, wird es umso schwieriger neue Funktionalität zu implementieren bzw. etwas zu ändern | Zahlreicher Duplicate Code in großen Anwendungen führt zu zahlreichen Problemen |
| B | 6 | Komplexe Strukturen und auch toter Code führen bei größeren Anwendungen zu sehr unübersichtlichen Verhältnissen, wo es schwierig ist durchzublicken | Steigende Komplexität und Dead Code führen gerade bei umfangreicheren Anwendungen zu Problemen |
| B | 7 | Mögliche Einflussfaktoren sind die Erfahrung mit Umgang von Debugging Tools, da diese entscheidend sind für Fehlersuche | Debugging Kenntnisse haben Einfluss auf Ergebnis |

| | | | |
|---|---|---|---|
| B | 8 | Auch denke ich, dass das Fachwissen und die Eigenheiten einer Programmiersprache einfluss auf ein derartiges Ergebnis haben, da sich viele Fehler dadurch vermeiden lassen | Fachwissen über Programmiersprache entscheidend |
| B | 9 | Der wichtigste Einflussfaktor war höchstwahrscheinlich die unterschiedliche Herangehensweise der einzelnen Teilnehmer. Jemand der gesamten Code analysiert ist nicht so schnell, wie jemand der direkt debugging betreibt im konkreten Fall | Wichtigster Einflussfaktor ist Herangehensweise der Teilnehmer |
| B | 10 | Man könnte die unterschiedlichen Kenntnisse der Teilnehmer durch Erstellung von Klassendiagrammen oder Sequenzdiagrammen validieren, da dadurch jeder mit der gesammten Funktionalität der Anwendung vertraut sein müsste | Mögliche Validierung der Teilnehmerkenntnisse durch Klassen- und Sequenzdiagramme |
| C | 1 | Ich denke die Größe der Teilnehmeranzahl ist für eine solche Untersuchung sehr relevant. Hier müsste man mehr als 100 Teilnehmer untersuchen, um gute Ergebnisse zu bekommen | Größere Teilnehmeranzahl wäre für zukünftige Untersuchung notwendig |
| C | 2 | Main Methode war nicht entscheidend, aber vielleicht dennoch hilfreich, um den Ablauf der Anwendung zu verstehen | Debugging durch Main Methode nicht entscheidend aber hilfreich |
| C | 3 | Die Vorgangsweise des einzelnen Teilnehmers war sicher entscheidend für das Ergebnis | Vorgangsweise entscheidend für Ergebnis |
| C | 4 | Die tatsächliche Erfahrung sowie Kenntnis über potentielle Fehler sind wichtig zur Fehlersuche | Erfahrung und Fachwissen entscheidend für Fehlersuche |
| C | 5 | Ergebnisse sind dennoch plausibel, da offensichtlich die Codequalität für die reine Fehlersuche durch Debugging nicht so entscheidend ist | Ergebnisse plausibel, Qualität für Debugging nicht entscheidend |

| | | | |
|---|---|---|---|
| C | 6 | IDE Kenntnisse beeinflussen die Geschwindigkeit der Fehlersuche und Implementierung neuer Funktionalität | IDE Kenntnisse beeinflussen Ergebnisse |
| C | 7 | Generell war die Aufgabenstellung für diese Untersuchung nicht zu einfach und gerade die schlechten Versionen waren sehr unübersichtlich | Aufgabenstellung war dem Rahmen absolut angemessen, unterschiedliche Versionen gut ausgearbeitet |
| C | 8 | Interpretation der Ergebnisse dahingehend, dass sich der Aufwand für Qualität bei kleinen, kurzlebigen Anwendungen nicht lohnt | Qualität für kleinere Einmal-Anwendungen nicht entscheidend |
| C | 9 | Größter Einflussfaktor waren sicher individuelle Fähigkeiten in Programmierung und die alltägliche Erfahrung im Arbeitsumfeld | Individuelle Fähigkeiten und Erfahrung größter Einflussfaktor |
| C | 10 | Tendenz zu erkennen, dass sich schlechter Code weiterverplanzt (Bad Fixes) | Bad Fixes nehmen bei schlechtem Code stark zu |

**Tabelle A.1.:** Auswertungstabelle der Experteninterviews

# Akronyme

| | |
|---|---|
| API | Programmierschnittstelle |
| DIP | Dependency inversion principle |
| GoF | Gang of Four |
| IDE | Integrierte Entwicklungsumgebung |
| ISP | Interface segregation principle |
| LOC | Lines of code |
| LSP | Liskov substitution principle |
| NCSS | Non commented source statements |
| OCP | Open/closed principle |
| QM | Qualitätsmanagement |
| SRP | Single responsibility principle |

# Abbildungsverzeichnis

# Tabellenverzeichnis

# Listings

# Literaturverzeichnis


*Aivosto.* (o. J.). Zugriff am 23.08.2017 auf `http://www.aivosto.com/project/help/pm-complexity.html`

Albrecht, A. J. & Gaffney, J. E. (1983). Software function, source lines of code, and development effort prediction: a software science validation. *IEEE transactions on software engineering* (6), 639–648.

Alur, D., Malks, D., Booch, G., Crupi, J. & Fowler, M. (2003). *Core j2ee patterns: Best practices and design strategies* (2nd ed. Aufl.). Boston, MA: ProQuest Information and Learning Company.

Alzaghoul, E. & Bahsoon, R. (2013). Economics-driven approach for managing technical debt in cloud-based architectures. In *Utility and cloud computing (ucc), 2013 ieee/acm 6th international conference on* (S. 239–242).

Anda, B. (2007). Assessing software system maintainability using structural measures and expert assessments. In *Software maintenance, 2007. icsm 2007. ieee international conference on* (S. 204–213).

Arcelli, D., Cortellessa, V. & Trubiani, C. (2015). Performance-based software model refactoring in fuzzy contexts. In A. Egyed & I. Schaefer (Hrsg.), *Fundamental approaches to software engineering* (Bd. 9033, S. 149-164). Berlin: Springer.

Bacchelli, A. & Bird, C. (2013). Expectations, outcomes, and challenges of modern code review. In *Proceedings of the 2013 international conference on software engineering* (S. 712–721).

Bandi, A., Williams, B. J. & Allen, E. B. (2013). Empirical evidence of code decay: A systematic mapping study. In *Reverse engineering (wcre), 2013 20th working conference on* (S. 341–350).

Boehm, B. W. (1980). *Characteristics of software quality* (Bd. 1). Amsterdam: North-Holland Publ. Co.

Brown, W. J. (1998). *Antipatterns: Refactoring software, architectures, and projects in crisis*. New York: Wiley.

Chapin, N. (1979). A measure of software complexity. In *Proc. ncc* (Bd. 77, S. 995–1002).

*Checkstyle project website.* (o. J.). Zugriff am 17.08.2017 auf `http://checkstyle.sourceforge.net`

Chidamber, S. R. & Kemerer, C. F. (1994). A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, *20* (6), 476–493.

*Ckjm.* (o. J.). Zugriff am 15.07.2017 auf `http://www.spinellis.gr/sw/ckjm/`

Cohen, D., Lindvall, M. & Costa, P. (2004). An introduction to agile methods. *Advances in computers*, *62*, 1–66.

Cohen, J., Teleki, S., Brown, E. & DuRette, B. (2006). *Best kept secrets of peer code review*. Austin, Tex.: Smart Bear.

Cunningham, W. (1992). *The wycash portfolio management system, addendum to the proceedings on object-oriented programming systems, languages, and applications (addendum).* October.

DeMarco, T. (1979). *Structured analysis and system specification*. Englewood Cliffs, N.J.: Prentice-Hall.

*Eclipse metrics plugin 1.3.6.* (o. J.). Zugriff am 20.08.2017 auf `http://metrics.sourceforge.net/`

Eick, S. G., Graves, T. L., Karr, A. F., Marron, J. S. & Mockus, A. (2001). Does code decay? assessing the evidence from change management data. *IEEE Transactions on Software Engineering*, *27* (1), 1–12.

*Findbugs project website.* (o. J.). Zugriff am 16.08.2017 auf `http://findbugs.sourceforge.net/`

Fowler, M. (2010). *Analysis patterns: Reusable object models*. Boston: Addison-Wesley.

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

Fowler, M. & Beck, K. (Hrsg.). (2013). *Refactoring: Improving the design of existing code* (28. printing Aufl.). Boston: Addison-Wesley.

Gamma, E., Helm, R., Johnson, R. & Vlissides, J. (1994). *Design patterns: Elements of reusable object-oriented software* (42. printing Aufl.). Boston: Addison-Wesley.

Halstead, M. H. (1979). *Elements of software science* (3. pr Aufl., Bd. 2). New York: North Holland.

Hoffmann, D. W. (2008). *Software-qualität*. Berlin, Heidelberg: Springer-Verlag Berlin Heidelberg.

ISO/IEC 25010:2011. (2011). *ISO/IEC 25010:2011: Systems and software engineering – Systems and software product Quality Requirements and Evaluation (SQuaRE) – System and software quality models* (Standard). International Organization for Standardization.

Khomh, F., Di Penta, M., Guéhéneuc, Y.-G. & Antoniol, G. (2012). An exploratory study of the impact of antipatterns on class change- and fault-proneness. *Empirical Software Engineering*, *17* (3), 243–275.

Khomh, F., Di Penta, M. & Gueheneuc, Y.-G. (2009). An exploratory study of the impact of code smells on software change-proneness. In *Reverse engineering, 2009. wcre'09. 16th working conference on* (S. 75–84).

Khoshgoftaar, T. M. & Munson, J. C. (1990). The lines of code metric as a predictor of program faults: A critical analysis. In *Computer software and applications conference, 1990. compsac 90. proceedings., fourteenth annual international* (S. 408–413).

Kitchenham, B. & Pfleeger, S. L. (1996). Software quality: the elusive target [special issues section]. *IEEE software*, *13* (1), 12–21.

Laird, L. M. & Brennan, M. C. (2006). *Software measurement and estimation: a practical approach* (Bd. 2). John Wiley & Sons.

LEE, T., LEE, J.-B. & IN, H. P. (2015). Effect analysis of coding convention violations on readability of post-delivered code. *IEICE Transactions on Information and Systems*, *E98.D* (7), 1286–1296.

Li, W. & Henry, S. (1993). Object-oriented metrics that predict maintainability. *Journal of systems and software*, *23* (2), 111–122.

Lieberherr, K., Holland, I. & Riel, A. (1988). Object-oriented programming: An objective sense of style. *ACM SIGPLAN Notices*, *23* (11), 323–334.

Liggesmeyer, P. (2009). *Software-qualität: Testen, analysieren und verifizieren von software* (2. Aufl. Aufl.). s.l.: Spektrum Akademischer Verlag.

Liskov, B. H. & Wing, J. M. (1999). *Behavioral subtyping using invariants and constraints*. Defense Technical Information Center.

Liso, A. (2001). Software maintainability metrics model: An improvement in the coleman-oman model. *Crosstalk*, 15–17.

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

Lowther, B. W. (1993). *The application of software maintainability metric models on industrial software systems* (Unveröffentlichte Dissertation). University of Idaho.

Marinescu, R. (2012). Assessing technical debt by identifying design flaws in software systems. *IBM Journal of Research and Development*, *56* (5), 9–1.

Martin, R. C. (2003). *Agile software development: Principles, patterns, and practices*. Upper Saddle River, NJ: Pearson Education.

Mayring, P. (2002). Qualitative sozialforschung. *Eine Anleitung zu qualitativen Denken*, *5*.

Mayring, P. (2003). Qualitative inhaltsanalyse [qualitative content analysis]. *Qualitative Forschung Ein Handbuch (Qualitative Research: A Handbook)*, 468–475.

McCabe, T. J. (1976). A complexity measure. *IEEE Transactions on Software Engineering*, *SE-2* (4), 308–320.

Meyer, B. (1995). *Object-oriented software construction* ([19. Dr.] Aufl.). New York: Prentice Hall.

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

Oman, P., Hagemeister, J. & Ash, D. (1991). A definition and taxonomy for software maintainability, report setl report 91-08-tr. *University of Idaho*.

*Oracle javadoc tool website.* (o. J.). Zugriff am 09.11.2017 auf `http://www.oracle.com/technetwork/java/javase/documentation/index-jsp-135444.html`

O'Regan, G. (2012). *A practical approach to software quality*. Springer Science & Business Media.

Palomba, F., de Lucia, A., Bavota, G. & Oliveto, R. (2015). Anti-pattern detection: Methods, challenges, and open issues. *Advances in Computers*, *95*, 201–238.

*Pmd project website.* (o. J.). Zugriff am 16.08.2017 auf `https://pmd.github.io/`

Quezada Sarmiento, P., Guaman, D., Barba Guamán, L. R., Quispe, L. & Cabrera, P. (2017, 07). Sonarqube as a tool to identify software metrics and technical debt in the source code through static analysis.

Roberts, D., Brant, J. & Johnson, R. (1997). A refactoring tool for smalltalk. *Urbana*, *51*, 61801.

Schnell, R., Hill, P. B. & Esser, E. (1999). Methoden der empirischen sozialforschung.

Singh, S. & Kahlon, K. (2011). Effectiveness of encapsulation and object-oriented metrics to refactor code and identify error prone classes using bad smells. *ACM SIGSOFT Software Engineering Notes*, *36* (5), 1–10.

Smit, M., Gergel, B., Hoover, H. J. & Stroulia, E. (2011). Code convention adherence in evolving software. In *Ieee 27th international conference on software maintenance (icsm)* (S. 504–507).

Sneed, H. M., Seidl, R. & Baumgartner, M. (Hrsg.). (2010). *Software in zahlen: Die vermessung von applikationen*. München: Hanser.

*Sonarqube.* (o. J.). Zugriff am 17.09.2017 auf `https://www.sonarqube.org/`

Spinellis, D. (2005). Tool writing: A forgotten art? *IEEE Software*, *22* (4), 9–11.

Suryanarayana, G., Samarthyam, G. & Sharma, T. (2015). *Refactoring for software design smells: Managing technical debt*. Burlington: Elsevier, Morgan Kaufmann.

Swamidurai, R., Dennis, B. & Kannan, U. (2014). Investigating the impact of peer code review and pair programming on test-driven development. In *Southeastcon* (S. 1–5).

Thaller, G. E. (2000). *Software-metriken: einsetzen, bewerten, messen*. Verlag Technik.

Van Emden, E. & Moonen, L. (2002). Java quality assurance by detecting code smells. In *Reverse engineering, 2002. proceedings. ninth working conference on* (S. 97–106).

Welker, K. D. (2001). The software maintainability index revisited. *CrossTalk*, *14*, 18–21.

Welker, K. D. & Oman, P. W. (1995). Software maintainability metrics models in practice. *Crosstalk, Journal of Defense Software Engineering*, *8* (11), 19–23.

Wolfgang, P. (1994). Design patterns for object-oriented software development. *Reading Mass*.

Yamashita, A. & Moonen, L. (2012). Do code smells reflect important maintainability aspects? In *Software maintenance (icsm), 2012 28th ieee international conference on* (S. 306–315).

Yamashita, A. & Moonen, L. (2013). Exploring the impact of inter-smell relations on software maintainability: An empirical study. In *35th international conference on software engineering (icse)* (S. 682–691).