OTTO VON GUERICKE
UNIVERSITÄT
MAGDEBURG

FAKULTÄT FÜR
INFORMATIK


# Masterarbeit

im Studiengang Informatik

## Einfluss unterschiedlicher Kommentararten auf die Lesbarkeit des Quellcodes


**Dariusz Krolikowski**


---

| | | |
|---|---|---|
| Betreuer | : | Sebastian Nielebock M. Sc. |
| Erstgutachter | : | Prof. Dr. Frank Ortmeier |
| Zweitgutachter | : | Dr.-Ing. Eike Schallehn |
| Eingereicht am | : | 19. August 2016 |

# Zusammenfassung

Softwareentwickler verbringen sehr viel Zeit mit dem Lesen von Quellcode. Ein lesbarer Code verbessert die Wartbarkeit und Verständlichkeit des Quellcodes und senkt die Entwicklungskosten. Diese Arbeit untersucht, welchen Einfluss dabei Kommentare auf die Quellcodelesbarkeit haben. Hierfür wurde eine Nutzerstudie durchgeführt, bei der sowohl Studenten als auch professionelle Entwickler neun Programmieraufgaben lösen sollten. Die Studie ergab, dass es für die Korrektheit der Lösungen unerheblich zu sein scheint, ob ein Quellcode kommentiert ist. Bei erfahrenen Entwicklern wurde außerdem kein signifikanter Unterschied in den Bearbeitungszeiten bei unterschiedlichen Kommentararten festgestellt. Unerfahrene Teilnehmer waren dagegen mit Dokumentationskommentaren insbesondere beim Erweitern vom bestehenden Code signifikant langsamer als mit Implementationskommentaren oder unkommentiertem Code.

# Danksagung

An dieser Stelle möchte ich mich bei all denjenigen bedanken, die mich auf der langen Reise begleitet und motiviert haben.

Ich danke besonders allen Teilnehmern, die freiwillig ein Stück ihrer Freizeit für diese Nutzerstudie opferten. Insbesondere gilt der Dank den Mitgliedern der Softwerkskammer, den Kollegen von Eudemonia Solutions sowie zahlreichen anonymen Freiwilligen. Ich bedanke mich bei Sebastian für eine hervorragende Betreuung und die zahlreichen, wertvollen Tipps. Ich danke natürlich auch Christian, Dirk und Josephin, die mich mathematisch, wissenschaftlich und grammatisch bei der Anfertigung dieser Masterarbeit unterstützten.

# Inhaltsverzeichnis

**Zusammenfassung** I

**Inhaltsverzeichnis** III

**Abbildungsverzeichnis** V

**Tabellenverzeichnis** VI

**Quellcodeverzeichnis** VII

**1 Einleitung** 1
- 1.1 Notwendigkeit der Quellcodelesbarkeit . . . 1
- 1.2 Zielsetzung der Studie . . . 1
- 1.3 Rahmenbedingungen der Studie . . . 3
- 1.4 Aufbau der Arbeit . . . 4

**2 Stand der Forschung** 5
- 2.1 Softwarewartung . . . 5
  - 2.1.1 Definition . . . 5
  - 2.1.2 Wartungstätigkeiten . . . 6
- 2.2 Lesbarkeit des Quellcodes . . . 7
  - 2.2.1 Definition . . . 8
  - 2.2.2 Metriken zur Messung der Quellcodelesbarkeit . . . 8
  - 2.2.3 Faktoren für die Beeinflussung der Quellcodelesbarkeit . . . 10
- 2.3 Quellcodekommentare . . . 11
  - 2.3.1 Kommentararten . . . 11
    - 2.3.1.1 Implementationskommentare . . . 11
    - 2.3.1.2 Dokumentationskommentare . . . 12
    - 2.3.1.3 Weitere Einteilungen der Kommentare . . . 14
  - 2.3.2 Einfluss der Kommentare auf die Quellcodelesbarkeit . . . 15
- 2.4 Messung der Programmiererfahrung . . . 16

**3 Experimentdefinition** 18
- 3.1 Motivation einer Nutzerstudie . . . 18
- 3.2 Umgang mit Störfaktoren . . . 19
  - 3.2.1 Programmiererfahrung . . . 19
    - 3.2.1.1 Fragebogen zur Bestimmung der Programmiererfahrung . . . 19
    - 3.2.1.2 Bestimmung eines Erfahrungswertes . . . 20

3.2.2 Bezeichnernamen . . . . . 22
3.2.3 Sonstige Störfaktoren . . . . . 22
3.3 Auswahl der Aufgaben . . . . . 22
3.4 Feedback-Fragebogen . . . . . 24
3.5 Technische Umsetzung der Studie . . . . . 24

**4 Ergebnisse** **27**
4.1 Datenaufbereitung . . . . . 27
4.1.1 Umgang mit Abbrechern und fehlenden Werten . . . . . 27
4.1.2 Umgang mit Ausreißern . . . . . 28
4.1.3 Darstellung der empirischen Daten . . . . . 30
4.1.4 Gruppierung der Teilnehmer nach Erfahrung . . . . . 32
4.2 Prüfung der Bearbeitungszeiten auf signifikante Unterschiede . . . . . 32
4.2.1 Definition eines Signifikanztests . . . . . 32
4.2.2 Test auf Normalverteilung . . . . . 34
4.2.2.1 Quantile-Quantile Plot (QQ-Plot) . . . . . 35
4.2.2.2 Anpassungstests auf Normalverteilung . . . . . 35
4.2.3 Test auf Varianzhomogenität . . . . . 36
4.2.4 Kruskal-Wallis-Test . . . . . 37
4.2.4.1 Alle Teilnehmer . . . . . 38
4.2.4.2 Nach Erfahrung . . . . . 39
4.2.5 Subjektive Einschätzung der Teilnehmer . . . . . 40
4.3 Prüfung der Korrektheit der Antworten auf signifikante Unterschiede . . . . . 42
4.4 Diskussion der Ergebnisse . . . . . 44
4.5 Gefährdungen der Validität . . . . . 46

**5 Fazit und Ausblick** **47**

**A Tabellen** **49**

**B Experimentaufgaben** **50**
B.1 Aufgabe 1 . . . . . 50
B.2 Aufgabe 2 . . . . . 51
B.3 Aufgabe 3 . . . . . 51
B.4 Aufgabe 4 . . . . . 53
B.5 Aufgabe 5 . . . . . 54
B.6 Aufgabe 6 . . . . . 55
B.7 Aufgabe 7 . . . . . 56
B.8 Aufgabe 8 . . . . . 57
B.9 Aufgabe 9 . . . . . 57

**Literaturverzeichnis** **59**

**Selbständigkeitserklärung** **65**

# Abbildungsverzeichnis


2.1 Kategorien der Softwarewartung [Ins06] (Übersetzung) . . . . . . . . . . . . 6
2.2 Methoden zur Messung der Programmiererfahrung in der Literatur [FKL+12] 17

3.1 Zuteilung der Teilnehmer in Gruppen . . . . . . . . . . . . . . . . . . . . . . 26

4.1 Boxplot für die Antwortzeiten (in Sekunden): Gruppe A, Aufgabe 1 . . . . . 29
4.2 Angaben der Teilnehmer (y-Achse: Anzahl der Teilnehmer) . . . . . . . . . . 30
4.3 Anzahl der gewerteten Antworten pro Aufgabe . . . . . . . . . . . . . . . . . 31
4.4 Bearbeitungszeiten aller Teilnehmer (in Sekunden) . . . . . . . . . . . . . . 33
4.5 QQ-Plot für Gruppe A, Aufgabe 1 . . . . . . . . . . . . . . . . . . . . . . . 35
4.6 Bearbeitungszeiten erfahrener Teilnehmer (in Sekunden) . . . . . . . . . . . 39
4.7 Bearbeitungszeiten unerfahrener Teilnehmer (in Sekunden) . . . . . . . . . . 40
4.8 Einschätzung zur Auswirkung von Kommentaren auf die Bearbeitungszeit (alle Teilnehmer) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 41
4.9 Einschätzung zur Auswirkung von Kommentaren auf die Bearbeitungszeit (nach Erfahrung) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 42
4.10 Anzahl richtiger und falscher Antworten (alle Teilnehmer) . . . . . . . . . . 43

# Tabellenverzeichnis

2.1 Kommentararten nach Steidl [SJ13] . . . . . . . . . . . . . . . . . . . . . . 14

3.1 Fragebogen . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 20
3.2 Zuordnung der Antwortmöglichkeiten zu einer Punktanzahl für die Berechnung des Erfahrungswertes . . . . . . . . . . . . . . . . . . . . . . . . . 21
3.3 Kommentararten in den drei Versionen der Studie . . . . . . . . . . . . . . 23
3.4 Fragebogen zur Auswirkung von Kommentaren auf die Bearbeitungszeit . . 24

4.1 $p$-Werte für den Shapiro-Wilk-Test . . . . . . . . . . . . . . . . . . . . . . . 36
4.2 $p$-Werte für den Levene-Test (statistisch signifikante Werte sind hervorgehoben) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 37
4.3 $p$-Werte für den Kruskal-Wallis-Test . . . . . . . . . . . . . . . . . . . . . 38
4.4 $p$-Werte des Dunn-Tests für alle Teilnehmer . . . . . . . . . . . . . . . . . . 38
4.5 $p$-Werte des Dunn-Tests für unerfahrene Teilnehmer . . . . . . . . . . . . . 40
4.6 Signifikante Unterschiede in den Bearbeitungszeiten zwischen Kommentararten bei unerfahrenen Teilnehmern . . . . . . . . . . . . . . . . . . . . . . . 41
4.7 $p$-Werte für die Unabhängigkeitstests . . . . . . . . . . . . . . . . . . . . . . 44

A.1 $p$-Werte für die Varianzanalyse der Bearbeitungszeiten . . . . . . . . . . . . 49

# Quellcodeverzeichnis

2.1 Beispiel für einen Blockkommentar . . . 12
2.2 Beispiele für Inline-Kommentare . . . 12
2.3 Beispiel für einen Javadoc-Kommentar . . . 13

B.1 Aufgabe 1 (Javadoc) . . . 50
B.2 Aufgabe 1 (Inline) . . . 50
B.3 Aufgabe 1 - Musterlösung . . . 50
B.4 Aufgabe 2 (Javadoc) . . . 51
B.5 Aufgabe 2 (Inline) . . . 51
B.6 Aufgabe 2 - Musterlösungen . . . 51
B.7 Aufgabe 3 (Javadoc) . . . 52
B.8 Aufgabe 3 (Inline) . . . 52
B.9 Aufgabe 3 - Musterlösung . . . 52
B.10 Aufgabe 4 (Javadoc) . . . 53
B.11 Aufgabe 4 (Inline) . . . 53
B.12 Aufgabe 4 - Musterlösung . . . 53
B.13 Aufgabe 5 (Javadoc) . . . 54
B.14 Aufgabe 5 (Inline) . . . 54
B.15 Aufgabe 5 - Musterlösung . . . 54
B.16 Aufgabe 6 (Javadoc) . . . 55
B.17 Aufgabe 6 (Inline) . . . 55
B.18 Aufgabe 6 - Musterlösung . . . 55
B.19 Aufgabe 7 (Javadoc) . . . 56
B.20 Aufgabe 7 (Inline) . . . 56
B.21 Aufgabe 7 - Musterlösung . . . 56
B.22 Aufgabe 8 (Javadoc) . . . 57
B.23 Aufgabe 8 (Inline) . . . 57
B.24 Aufgabe 8 - Musterlösung . . . 57
B.25 Aufgabe 9 (Javadoc) . . . 58
B.26 Aufgabe 9 (Inline) . . . 58
B.27 Aufgabe 9 - Musterlösung . . . 58

# 1 Einleitung

## 1.1 Notwendigkeit der Quellcodelesbarkeit

Der Entwicklungsprozess einer Software endet nicht mit deren Auslieferung. Eine Anwendung muss kontinuierlich gewartet werden, um Programmfehler zu beheben oder neue Anforderungen der Benutzer oder Kunden zu erfüllen. Eine wichtige Rolle für die Softwarewartung spielt dabei die Quellcodelesbarkeit. Programmierer verbringen etwa 30-90% der Entwicklungszeit mit dem Lesen von Quellcode [Fos93b]. Je schneller Softwareentwickler einen ihnen unbekannten Quelltext verstehen, desto schneller können sie daran auch Wartungstätigkeiten durchführen. Eine schlechte Codelesbarkeit erhöht außerdem die Anzahl der Programmfehler [BW08, KWR10, Dor12]. Ein besseres Verständnis vom Quellcode führt somit zu einer kürzeren Entwicklungsdauer und damit zu niedrigeren Entwicklungskosten der Software.

In der Industrie setzten sich zahlreiche Praktiken zur Verbesserung der Codelesbarkeit durch [Mar08, HT99, Pre10, Fow99]. Diese basieren jedoch oft auf persönlichen Erfahrungen und Präferenzen. Forscher sind daher bemüht, diese Methoden wissenschaftlich zu untersuchen. So wurde etwa die Verwendung von aussagekräftigen Bezeichnernamen als wichtiger Faktor für die Verbesserung der Quellcodelesbarkeit umfassend erforscht und anerkannt [BWYS11, DP06, LBS06, Rei05, BLMM09, SS14]. Der Einfluss von Kommentaren auf die Lesbarkeit des Codes wird in der Literatur dagegen unterschiedlich stark gewertet. Außerdem werden viele Studien ausschließlich mit Studenten durchgeführt, wodurch die Relevanz der Ergebnisse für die Industrie umstritten ist.

## 1.2 Zielsetzung der Studie

Das Ziel dieser Arbeit ist die Einschätzung, ob bestimmte Kommentararten zu einer besseren Verständlichkeit des Quellcodes und zu einer schnelleren Bewältigung von quellcodebezogenen Aufgaben beitragen können.

Die wissenschaftliche Frage lautet:

> Wie sollte Quellcode kommentiert werden, damit er signifikant schneller und besser verstanden wird als unkommentierter Code?

Da es keine zuverlässigen Metriken zur Messung des Codeverständnisses gibt, wird für die Beantwortung dieser Frage eine Nutzerstudie durchgeführt. Für die wissenschaftliche Frage werden folgende Unterfragen gestellt:

**Forschungsfrage 1:** Welche Art von Kommentaren hilft, Programmieraufgaben schneller zu lösen?

In der Softwareentwicklung ist es von großer Bedeutung, Aufgaben möglichst zeiteffizient durchzuführen. Ein großer Teil der Zeit wird dabei mit dem Lesen des Quellcodes verbracht. Daher lautet die erste Frage, ob Codekommentare diese Zeit verkürzen können. Die Kommentare werden in dieser Arbeit in zwei Arten unterteilt: Dokumentationskommentare (Beschreibung der Schnittstelle) und Implementationskommentare (Implementierungsdetails). Es gilt herauszufinden, ob die Kommentararten einen unterschiedlichen Einfluss auf die Schnelligkeit der Antworten haben.

**Forschungsfrage 2:** Welche Kommentarart bewirkt, dass Programmieraufgaben häufiger korrekt gelöst werden?

Eine schnellere Lösung ist unbrauchbar, wenn sie nicht korrekt ist. Daher wird in dieser Arbeit zusätzlich untersucht, welchen Einfluss die Kommentararten auf die Korrektheit der Lösungen haben.

**Forschungsfrage 3:** Gibt es einen signifikanten Unterschied zwischen erfahrenen und unerfahrenen Programmierern bei der Schnelligkeit und Richtigkeit der Aufgabenbearbeitung?

Viele Nutzerstudien werden ausschließlich mit Studenten durchgeführt [JHDE11]. Dadurch verlieren die Ergebnisse an Praxisrelevanz, da es unklar ist, inwiefern sie auf die professionellen Entwickler angewendet werden können. Daher ist ein Ziel dieser Arbeit, möglichst viele Teilnehmer mit unterschiedlichem Erfahrungsstand für die Studie zu gewinnen. Somit können Unterschiede zwischen erfahrenen und unerfahrenen Entwicklern hinsichtlich des Einflusses der Kommentare auf die Codelesbarkeit näher untersucht werden. Eine Studie berichtet etwa, dass erfahrene Teilnehmer Kommentare eher meiden oder sogar ignorieren, diese also als störend empfinden [SS14]. Daher wird bei dieser Forschungsfrage vermutet, dass Kommentare unerfahrenen Entwicklern mehr helfen als erfahrenen.

**Forschungsfrage 4:** Unterscheidet sich der Einfluss der Kommentare auf die Schnelligkeit und Richtigkeit der Aufgabenbearbeitung bei unterschiedlichen Wartungstätigkeiten?

Verschiedene Wartungstätigkeiten in der Softwareentwicklung erfordern unterschiedliche Herangehensweisen und Informationen. Bei einer Fehlerbehebung wird zum Beispiel die Fehlerursache im Quellcode gesucht, während eine neue Anforderung an die Software zu Erweiterungen im Quellcode führt. Es gilt herauszufinden, ob die Kommentararten bei verschiedenen Wartungstätigkeiten einen unterschiedlichen Einfluss auf die Lesbarkeit des Quellcodes haben. Dieser Aspekt wurde bisher noch nicht wissenschaftlich untersucht.

Die ersten beiden Forschungsfragen konzentrieren sich auf den allgemeinen Einfluss von Kommentaren auf die Quellcodelesbarkeit. Die Forschungsfragen 3 und 4 untersuchen zusätzlich, ob der Einfluss der Kommentare von der Programmiererfahrung oder den Wartungstätigkeiten abhängt.

## 1.3 Rahmenbedingungen der Studie

Die Kommentare werden in dieser Arbeit in Implementations- und Dokumentationskommentare unterteilt. In der Literatur werden auch andere Einteilungen vorgestellt (siehe Kapitel 2.3.1.3). Diese werden bei der Beantwortung der Forschungsfragen jedoch nicht untersucht.

Aussagekräftige Bezeichner machen in der Praxis viele Kommentare überflüssig [Mar08]. In dieser Studie soll jedoch ausschließlich der Einfluss der Kommentare untersucht werden. Um den Einfluss der Bezeichnernamen auszuschließen, werden diese deshalb für das Experiment anonymisiert. Das hat zur Folge, dass die Bedeutung der Kommentare gegenüber der Praxis etwas überbewertet wird.

Für die Bearbeitung der Nutzerstudie wird etwa eine Stunde eingeplant. Um in dieser Zeit mehrere Aufgaben lösen zu können, sind die ausgewählten Codebeispiele kurz. Die Ergebnisse unterscheiden sich somit möglicherweise von komplexeren Praxisanwendungen.

Alle Quellcodebeispiele in der Studie sind in Java geschrieben, einer der verbreitetsten Programmiersprachen (siehe Tiobe-Index[1], RedMonk[2], PYPL[3]). Die hier untersuchten Kommentararten sind zwar auf andere Programmiersprachen übertragbar (vgl. Kapitel 2.3.1), es soll dennoch keine Aussage über eine allgemein beste Kommentarart in allen Programmiersprachen getroffen werden.

[1] http://www.tiobe.com/tiobe_index
[2] http://redmonk.com/sogrady/category/programming-languages/
[3] http://pypl.github.io/PYPL.html

## 1.4 Aufbau der Arbeit

In Kapitel 2 werden die für die Beantwortung der Forschungsfragen verwendeten Wartungstätigkeiten und Kommentararten definiert. Außerdem werden bisherige Arbeiten auf dem Gebiet der Quellcodelesbarkeit vorgestellt sowie Störfaktoren für die Studie beschrieben.

Kapitel 3 fasst die Vorüberlegungen sowie den Aufbau des Experiments zusammen und beschreibt, wie dabei mit den Störfaktoren umgegangen wurde.

Anschließend werden in Kapitel 4 die erhobenen Daten ausgewertet, die Ergebnisse der Studie vorgestellt sowie die Forschungsfragen beantwortet.

Kapitel 5 fasst schließlich die Arbeit zusammen und gibt einen Ausblick für einen möglichen Forschungsbedarf.

# 2 Stand der Forschung

## 2.1 Softwarewartung

Diese Arbeit untersucht unter anderem die Unterschiede in der Quellcodelesbarkeit bei verschiedenen Wartungstätigkeiten. Hierfür wird in diesem Kapitel die Softwarewartung definiert, um daraus anschließend die Wartungstätigkeiten für die Studie abzuleiten.

Das Ziel der Softwareentwicklung ist es, eine Software zu schreiben, welche gewissen Nutzeranforderungen erfüllt. Eine Software unterliegt jedoch, ähnlich wie Menschen, einem Alterungsprozess [Par94]. Eine Wartung ist erforderlich, damit das Softwareprodukt kontinuierlich bestehende sowie neue Nutzeranforderungen erfüllt. So müssen etwa Programmfehler behoben oder neue Kundenwünsche umgesetzt werden.

Die Wartung nimmt, je nach Studie, etwa 40-75% der Entwicklungskosten einer Software ein [Fos93b]. Daher ist der Bereich der Softwarewartung und speziell die Bemühungen, deren Zeit und Kosten zu senken, ein wichtiges Forschungsgebiet.

### 2.1.1 Definition

IEEE Standard for Software Maintenance (IEEE 1219) definiert Softwarewartung wie folgt:

> „Modification of a software product after delivery to correct faults, to improve performance or other attributes, or to adapt the product to a modified environment.“ [IEE98]

Laut International Standard for Software Maintenance (ISO/IEC 14764) bedeutet Softwarewartung:

> „The totality of activities required to provide cost-effective support to a software system. Activities are performed during the pre-delivery stage as well as the post-delivery stage.“ [Ins06]

Es handelt sich dabei also um alle Aktivitäten, welche die dauerhafte Verwendbarkeit der Software garantieren. Einige Aktivitäten finden bereits vor der Auslieferung der Software statt, beispielsweise die Planung der Wartungsphase. Das Ziel der Softwarewartung ist es, bestehende Software zu verändern, ohne deren Integrität zu gefährden [Ins06, BF14].

Aus den genannten Definitionen wird der Begriff *Wartbarkeit* abgeleitet. Sie gibt an, wie einfach ein Softwareprodukt hinsichtlich der Wartungsarbeiten verändert werden kann [Ins06].

Die Softwarewartung setzt sich aus Fehlerbehebungen und Verbesserungen zusammen, die jeweils in zwei Unterklassen unterteilt werden (Abbildung 2.1):

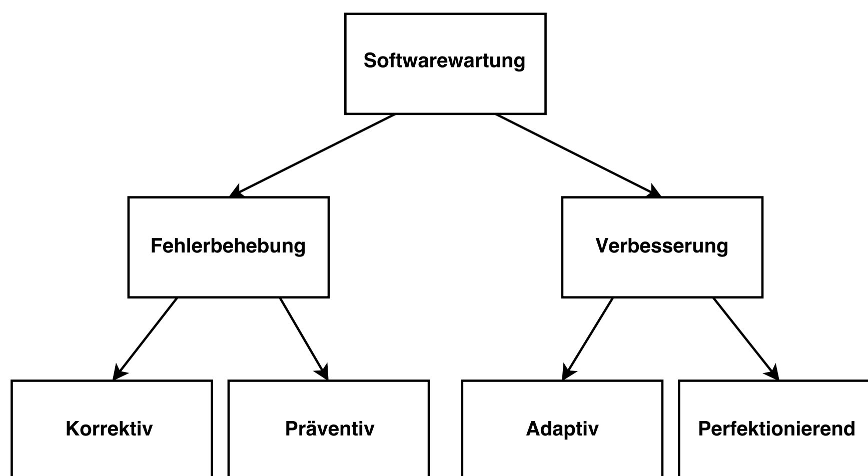


Abbildung 2.1: Kategorien der Softwarewartung [Ins06] (Übersetzung)

- **Korrektive Wartung**. Bereits entdeckte Fehler beheben.
- **Präventive Wartung**. Zukünftigen Problemen vorbeugen, indem beispielsweise noch unbekannte Fehler gefunden und behoben werden.
- **Adaptive Wartung**. Die Software an eine neue Umgebung anpassen, etwa nach einem Betriebssystemupdate.
- **Perfektionierende Wartung**. Performanz oder Wartbarkeit der Software verbessern.

### 2.1.2 Wartungstätigkeiten

Aus den genannten Kategorien werden für diese Arbeit folgende Tätigkeiten abgeleitet, die ein Softwareentwickler am Quellcode im Zuge der Wartung durchführen muss.

**Quellcode lesen.** Um einen Quelltext entsprechend der Anforderungen anpassen zu können, muss dieser stets gelesen und soweit verstanden werden, dass die Änderung keine neuen Fehler verursacht.

**Quellcode erweitern.** Um einen Änderungswunsch umzusetzen, muss der vorhandene Quellcode entsprechend um die neue Anforderung erweitert werden.

**Quellcode verwenden.** Die Wiederverwendung von vorhandenem Quellcode ist in der Softwareentwicklung sehr erwünscht. Das DRY[1]-Prinzip [HT99, Mar08] wird eingehalten, um Redundanzen wie Codeduplikate zu verhindern. Daher wird bestehender Quellcode oft verwendet, etwa wenn andere Methoden aufgerufen werden. Zu diesem Punkt gehört auch die Verwendung von Schnittstellen, zum Beispiel bei externen Bibliotheken.

**Fehler im Quellcode beheben.** Die Ursache von Programmfehlern ist oft nicht sofort ersichtlich. Daher kommt bei der Fehlerbehebung im Gegensatz zur Codeerweiterung ein weiterer Punkt hinzu: die Fehlerursache ermitteln.

**Refactoring durchführen.** Um die Wartbarkeit zu verbessern, werden während der perfektionierenden Wartung unter anderem Refactorings durchgeführt. Unter *Refactoring* versteht man Änderungen an der Software, welche die Codestruktur verbessern, ohne das Programmverhalten zu verändern. Das Ziel ist es, den Quellcode „aufzuräumen“, ohne neue Programmfehler einzubauen [Fow99]. Da bei reinen Strukturumformungen die Richtigkeit der Lösung schwer zu definieren ist, wird das Refactoring als Wartungstätigkeit in dieser Studie nicht untersucht (vgl. Kapitel 3.3).

Es ist naheliegend, dass Entwickler in den jeweiligen Tätigkeiten unterschiedliche Grade von Codeverständnis benötigen. So ist das erforderliche Verständnis von einer Methode bei deren Verwendung in der Regel geringer als bei einer Fehlerbehebung. Diese Vermutung gilt es in dieser Arbeit zu überprüfen.

## 2.2 Lesbarkeit des Quellcodes

Ein für die Entwickler unverständlicher Quellcode ist von geringem Wert [LBS06]. Ein lesbarer Code verbessert nicht nur die Wartbarkeit [BW08, Dor12, ASC02], er erhöht außerdem die Wiederverwendbarkeit [CV06] sowie die Verständlichkeit [Dor12, WPVS14, PHD11] des Quellcodes. Gleichzeitig senkt dieser die Entwicklungskosten [CV06, EM82], die Komplexität [TOAY13], sowie die Anzahl der Programmfehler [BW08, KWR10, Dor12].

[1] Don't Repeat Yourself

Viele Praktiken zur Verbesserung der Codelesbarkeit basieren auf Erfahrungswerten der Softwareentwickler. Es ist daher von großem Interesse, diese Methoden auch wissenschaftlich zu erforschen.

### 2.2.1 Definition

In der Literatur gibt es keine einheitliche Definition der Quellcodelesbarkeit. Nach Buse und Weimer bedeutet Codelesbarkeit die menschliche Beurteilung, wie einfach ein Quellcode zu verstehen ist [BW08]. Sedano definiert die Codelesbarkeit als benötigten mentalen Aufwand, um einen Code zu verstehen [Sed16]. Für Posnett et al. ist es der subjektive Eindruck über die Schwierigkeit, den Quellcode zu verstehen [PHD11]. Nach Tenny [Ten88] wird die Lesbarkeit durch die Anzahl richtiger Antworten zu einer Menge von Verständnisfragen über ein Programm in einer vorgegebenen Zeit ausgedrückt. Dagegen definieren Aggarwal et al. Codelesbarkeit als Anteil von Kommentarzeilen (LOC) im gesamten Code [ASC02].

Das Verständnis steht bei den meisten Definitionen im Vordergrund. Wie Posnett anmerkte, gibt es einen Unterschied, ob etwas soweit verstanden wird, um den Inhalt wiederzugeben, oder ob dieses Wissen auch angewendet werden kann. Es ist somit von großer Bedeutung, einen Quelltext so zu verstehen, dass er entsprechend der Anforderungen des Kunden oder der Nutzer angepasst werden kann. Daher wird für diese Arbeit eine angepasste Definition von Sedano verwendet:

> Die Quellcodelesbarkeit ist der benötigte mentale Aufwand, um einen Code so zu verstehen, dass daran erfolgreich Wartungstätigkeiten durchgeführt werden können.

Der Aufwand wird dabei mithilfe der Antwortzeit auf eine Frage zu einem Quelltext ausgedrückt [SGR+15]. *Erfolgreich* beschreibt die Korrektheit der Lösung, welche von der jeweiligen Wartungstätigkeit abhängt.

### 2.2.2 Metriken zur Messung der Quellcodelesbarkeit

In den in Kapitel 2.2.1 genannten Definitionen wird die Subjektivität der Quellcodelesbarkeit betont. Das liegt nahe, denn ein für einen Entwickler verständlicher Quelltext kann einen anderen Entwickler verwirren [Sed16, TOAY13, Dor12]. Das macht eine automatisierte Messung der Codelesbarkeit besonders schwierig.

Es existieren mehrere gängige Verfahren zur Bestimmung der Lesbarkeit von Texten, etwa das Flesch-Kincaid-Grade-Level [Fle48] oder der Gunning-Fog-Index [Gun52]. Diese Metriken nutzen einfache Messwerte, wie die Länge von Wörtern, Silben und Sätzen. Trotz ihrer Einfachheit sind sie in der Praxis sehr nützlich. Flesch-Kincaid wird beispielsweise bei Textverarbeitungsprogrammen wie Microsoft Word eingesetzt und ist zugleich ein Standard für die Dokumente der Regierung der Vereinigten Staaten [BW08]. Diese Verfahren sind jedoch nicht zur Messung der Codelesbarkeit geeignet.

Buse und Weimer [BW08] stellten als Erstes ein Modell zur Messung der Codelesbarkeit vor. Sie führten zunächst eine Studie mit 120 Informatikstudenten durch, welche die Lesbarkeit von 100 Codestücken subjektiv bewertet haben. Die Forscher stellten dann 25 rein syntaktische Quelltexteigenschaften zusammen, wie die Zeilenlänge oder die Anzahl von Variablen. Schließlich untersuchten sie die Korrelation zwischen diesen Eigenschaften und den Teilnehmerbewertungen. Das daraus entwickelte Modell war im Stande, etwa 80% der Codebeispiele richtig zu klassifizieren. Die Anzahl von Variablen zeigte die höchste Korrelation mit der Codelesbarkeit auf. Kommentare korrelierten dagegen nur mittelmäßig mit der Lesbarkeit. Die vermutete Ursache lag bei schlecht lesbarem Quelltext, der die Kommentare erst nötig machte.

Buse und Weimer betonten jedoch, dass dies kein allgemein gültiges Modell für die Quellcodelesbarkeit sei. Das Modell bot zwar eine erste Grundlage für weitere Forschungen, zeigte jedoch einige Schwächen auf. Buse merkte selbst an, dass sich die von ihm verwendeten Quellcodeeigenschaften teilweise überschnitten. Beispielsweise beschreiben „Anzahl der Wörter“ und „Anzahl der Leerzeichen“ in der Regel das gleiche Merkmal. Die gewählten Codebeispiele wurden außerdem bewusst sehr klein gehalten (maximal 7 Zeilen). Posnett et al. zeigten, dass sich das Modell auf größere Methoden oder Klassen nicht immer übertragen lässt [PHD11]. Schließlich gibt es einige Kritikpunkte an der Studie selbst. Da nur Studenten daran teilgenommen haben, lässt dies keine Aussage über die gegebenenfalls abweichende Bewertung von erfahreneren Entwicklern zu. Es kann außerdem nicht sichergestellt werden, dass die Teilnehmer die vorgestellten Beispiele soweit verstanden haben, um daran Wartungstätigkeiten durchführen zu können.

Posnett et al. [PHD11] bauten auf der Arbeit von Buse und Weimer auf und stellten ein vereinfachtes Modell vor, das nur von der Quelltextlänge und Entropie (mittlerer Informationsgehalt pro Zeichen) abhängt. Sie verwendeten dabei die Teilnehmerbewertungen von Buse, ohne eine neue Studie durchzuführen. Das vorgestellte Modell erzielte im Vergleich zu Buse eine höhere Genauigkeit.

Die Modelle von Buse und Posnett nutzten rein syntaktische Codeeigenschaften. Dorn [Dor12]

erweiterte diese um semantische Merkmale, welche die Codelesbarkeit verbessern. Dazu gehören unter anderem Strukturmuster, wie etwa die Verwendung von Zeileneinrückungen und Leerzeichen. Ein weiterer Faktor ist die visuelle Wahrnehmung. Beispielsweise ist ein Quelltext lesbarer, wenn eine farbige Syntaxhervorhebung verwendet wird. Die Verwendung der natürlichen Sprache trägt ebenfalls zur verbesserten Codelesbarkeit bei, etwa wenn Variablennamen aus Wörtern einer dem Leser vertrauten Sprache bestehen. Um sein Modell aufzustellen, führte Dorn die bisher größte Studie auf dem Gebiet der Codelesbarkeit durch. 5000 Teilnehmer, darunter sowohl Studenten als auch erfahrene Entwickler, bewerteten bis zu 360 Codebeispiele mit bis zu 50 Zeilen Länge. Dabei stellte Dorn unter anderem fest, dass die Bewertung der Teilnehmer nicht mit der Metrik von Buse und Weimer übereinstimmte.

Dieses Kapitel stellte die ersten Arbeiten zur automatischen Messung der Quellcodelesbarkeit vor. Eine subjektive Bewertung bleibt in der Wissenschaft jedoch weiterhin unerlässlich [Sed16]. Daher wird für die Beantwortung der Forschungsfragen eine Nutzerstudie durchgeführt, anstatt die vorgestellten Metriken zu verwenden.

### 2.2.3 Faktoren für die Beeinflussung der Quellcodelesbarkeit

Dieser Abschnitt beschreibt Faktoren, welche die Quellcodelesbarkeit beeinflussen. Diese müssen bei der Experimentdefinition berücksichtigt werden, damit sie die Ergebnisse nicht als Störfaktoren verfälschen. In Kapitel 2.3.2

Einer der wichtigsten Faktoren für eine gute Codelesbarkeit ist die Wahl von aussagekräftigen Bezeichnernamen, etwa für Variablen, Methoden oder Klassen [BWYS11, DP06, LBS06, Rei05, BLMM09, SS14]. Diese Bedeutsamkeit ist verständlich, denn Bezeichner nehmen etwa 70% des Quelltextes ein [DP06]. Das Ziel ist selbstbeschreibender Code, der viele Kommentare überflüssig macht. Beispielsweise verrät die Variable

```
int i;
```

nichts über deren Zweck. Dagegen ist

```
int customerId;
```

für den Leser sofort verständlich.

Andere Faktoren, welche sich positiv auf die Codelesbarkeit auswirken, sind beispielsweise die Verwendung von Leerzeilen [WPVS14] oder Syntaxhervorhebung [Dor12]. Die Codelesbarkeit hängt außerdem von der verfügbaren Dokumentation [SS14], der Erfahrung der Entwickler [SGR+15], sowie deren Wissen über die Domäne der Anwendung ab [SS14, EM82].

Die Verwendung von Kommentaren als Faktor für die Quellcodelesbarkeit wird in Kapitel 2.3.2 behandelt.

## 2.3 Quellcodekommentare

Dieses Kapitel beschreibt zunächst, welche Arten von Kommentaren in dieser Arbeit untersucht werden. Anschließend werden die Ergebnisse bisheriger Studien präsentiert, welche den Einfluss der Kommentare auf die Codelesbarkeit erforschen.

Quellcode wird in erster Linie für einen Computer geschrieben, der die darin enthaltenen Anweisungen ausführt. Es ist jedoch wichtiger, dass auch andere Entwickler verstehen, welche Aufgabe ein Quelltext erfüllt [Knu92]. Die Schwierigkeit, Quellcode zu lesen und zu verändern, kann durch sinnvolle Kommentare verringert werden [EM82]. Dennoch wird neuer Code kaum kommentiert [FWG07]. Ein weiteres Problem besteht darin, dass Kommentare bei Codeänderungen oft nicht angepasst werden [JH06, SS14]. Das führt dazu, dass Widersprüche zwischen dem Code und den dazugehörigen Kommentaren entstehen. Mögliche Ursachen hierfür sind etwa die fehlende Zeit aufgrund bestimmter Fertigstellungsfristen oder die fehlende Motivation der Entwickler.

### 2.3.1 Kommentararten

In dieser Studie wird ausschließlich Java-Quellcode verwendet (vgl. Kapitel 1.3). Gemäß der Java Code Conventions [Sun97] werden Kommentare in zwei Gruppen unterteilt: Implementationskommentare und Dokumentationskommentare. Robillard [Rob89] nimmt eine analoge Einteilung vor, nennt diese jedoch erzählerisch („was passiert“) und operativ („wie passiert etwas“).

#### 2.3.1.1 Implementationskommentare

Implementationskommentare beschreiben unter anderem den Zweck einer Variable oder eines Ausdrucks, bestimmte Entscheidungen zur Implementierung oder bekannte Probleme [Ver00].

Implementationskommentare können rein syntaktisch nicht eindeutig einem Ausdruck oder einer Gruppe von Ausdrücken zugeordnet werden [FWG07]. Sie können sich beispielsweise auf vorangehende oder nachfolgende Zeilen beziehen. Dies stellt eine Herausforderung beim

Refactoring dar [SZCF08]. Daher schlug Kaelbling vor, stattdessen *scoped comments* in den Programmiersprachen einzuführen, also Kommentare, die einzelnen Ausdrücken zugewiesen werden [Kae88]. Dieses Konzept setzte sich jedoch in keiner der verbreiteten Sprachen durch.

Reddy [Red00] unterteilt Implementationskommentare wie folgt.

**Blockkommentare** wurden von der Programmiersprache *C* übernommen. Sie werden von /* und */ umschlossen und können mehrere Zeilen lang sein. Quelltext 2.1 ist ein Beispiel für einen Blockkommentar. Reddy schlägt vor, Blockkommentare nur für das Copyright und zum temporären Auskommentieren von Codezeilen zu verwenden.

```
/*
 * Wir verwenden hier eine Queue, um die
 * Anfragen nacheinander abzuarbeiten.
 */
```

Quelltext 2.1: Beispiel für einen Blockkommentar

**Einzeilige (*Single-Line* oder *Inline*) Kommentare** beginnen mit //, gefolgt von einem Kommentartext. Solche Kommentare dürfen sowohl in einer neuen Zeile als auch am Ende einer Codezeile stehen. Quelltext 2.2 zeigt diese beiden Varianten. Mehrere aufeinanderfolgende einzeilige Kommentare können für größere Anmerkungen verwendet werden.

```
String ip = "127.0.0.1";    // Localhost IP-Adresse

if (status == 404) {

  // Seite nicht gefunden
  ...
}
```

Quelltext 2.2: Beispiele für Inline-Kommentare

#### 2.3.1.2 Dokumentationskommentare

Dokumentationskommentare beschreiben hauptsächlich die Schnittstelle, beispielsweise bei Programmbibliotheken. Sie sollten alle Informationen bereitstellen, die der Aufrufer benötigt, ohne auf die Implementation einzugehen [Ver00]. Beispielsweise sollte ein Dokumentationskommentar darauf hinweisen, falls eine Methode mit *null*-Werten umgehen oder diese zurückgeben kann. Diese Information ist für den Aufruf der Methode und die darin erlaubten Parameter wichtig.

**Javadoc** ist ein Werkzeug zur Erstellung von API-Dokumentation aus Kommentaren in Java [Kra99]. Das Ziel von Javadoc ist es, die Spezifikation der Schnittstelle mit dem Quellcode zu verknüpfen, um daraus automatisch eine Dokumentation im HTML-Format zu erstellen. Ein großer Vorteil besteht darin, dass eine generierte Dokumentation immer die aktuelle Schnittstelle beschreibt, selbst wenn die Kommentare veraltet sind [Kra99].

Die Popularität von Javadoc führte zu zahlreichen Umsetzungen in anderen Programmiersprachen, etwa JSDoc[2] in JavaScript oder phpDocumentator[3] in PHP. Doxygen[4], ein anderes Tool zur Dokumentationserstellung, erweitert die Javadoc-Syntax und unterstützt eine Vielzahl von Programmiersprachen. Daher wird vermutet, dass sich die Ergebnisse dieser Arbeit (Kapitel 4) auf andere Sprachen übertragen lassen.

Javadoc-Kommentare beginnen mit */*** und enden mit **/*. Sie sind syntaktisch eine Erweiterung der Blockkommentare, beschreiben jedoch keine Implementierung. Vermeulen schlägt daher vor, auf Blockkommentare nach Möglichkeit zu verzichten, um einer Verwechslungsgefahr gegenüber Javadoc-Kommentaren vorzubeugen [Ver00].

```
/**
 * Gibt das erste Element eines Arrays zurück. Das Array darf nicht
 * null sein. Falls das Array leer ist, wird -1 zurückgegeben.
 *
 * @param numbers  ein Array mit int Werten
 * @return         das erste Element des Arrays, -1 falls das Array leer ist
 * @see            ArrayUtil#getLastElement(int[])
 */
public int getFirstElement(int[] numbers) {
  ...
}
```

Quelltext 2.3: Beispiel für einen Javadoc-Kommentar

Oracle veröffentlichte einen Leitfaden zum Verfassen von Javadoc-Kommentaren [Ora12]. Ein Javadoc-Kommentar besteht aus einer Beschreibung gefolgt von Tags. Der erste Satz eines Javadoc-Kommentars sollte das kommentierte Element (etwa eine Klasse oder Methode) knapp aber vollständig zusammenfassen. Die Beschreibung soll unabhängig von der Implementierung sein, jedoch alle Randbedingungen und Sonderfälle beinhalten. Dies ist wichtig, um daraus aussagekräftige Modultests oder Neuimplementierungen schreiben zu können.

Nach der Beschreibung folgen Tags, die weiterführende Informationen beinhalten. Das Tag *@param* beschreibt die Parameter einer Methode, eines Konstruktors oder einer Klasse. *@return* enthält Informationen zum Rückgabewert einer Methode. Mittels *@see* werden Verweise

[2] http://usejsdoc.org
[3] https://www.phpdoc.org/
[4] http://www.stack.nl/~dimitri/doxygen/

zu anderen Klassen oder externen Dokumenten angegeben. Andere Tags werden im Rahmen dieser Arbeit nicht verwendet.

Quelltext 2.3 zeigt an einem Beispiel, wie ein Javadoc-Kommentar aussieht.

#### 2.3.1.3 Weitere Einteilungen der Kommentare

In der Literatur werden weitere Möglichkeiten zur Einteilung von Kommentaren aufgeführt.

Steidl [SJ13] stellte eine Einteilung der Kommentare nach dem Verwendungszweck vor (Tabelle 2.1). Robert C. Martin nennt in seinem Buch „Clean Code“ [Mar08] unter anderem auch diese Kommentararten.

| Art | Beschreibung |
|---|---|
| Copyright | Urheber und Lizenz |
| Header | Dokumentationskommentare für Klassen |
| Member | Dokumentationskommentare für Methoden und Felder |
| Inline | Implementationskommentare |
| Section | Markierungen für bestimmte Programmabschnitte |
| Code | Auskommentierter Code |
| Task | Ausstehende Aufgaben (TODO) |

Tabelle 2.1: Kommentararten nach Steidl [SJ13]

Sommerlad gruppiert Kommentare nach der relativen Position zum dazugehörigen Codeausdruck (davor, danach, freistehend) [SZCF08].

Ying [YWA05] beschäftigte sich mit TODO-Kommentaren, die hauptsächlich zur Kennzeichnung von ausstehenden Aufgaben verwendet werden. Er stellte auch einige andere Verwendungsmöglichkeiten fest, etwa Kommunikation mit Mitarbeitern, Verweise zum Issue Tracker, Lesezeichen oder Bedenken zum Code.

Die vorgestellten Kommentararten beschreiben eher die Verwendungsmöglichkeiten von Kommentaren oder reine Syntaxunterschiede. Dagegen zielt die Einteilung in Dokumentations- und Implementationskommentare auf unterschiedliches Wissen ab, welches zum Codeverständnis benötigt wird. Während Dokumentationskommentare lediglich die Spezifikation beschreiben, erläutern Implementationskommentare die genauen Implementierungsdetails. Die Einteilung in Implementations- und Dokumentationskommentare erscheint daher sinnvoll für

die in dieser Arbeit gestellten Forschungsfragen nach Unterschieden in der Codelesbarkeit bei der Verwendung verschiedener Kommentararten.

### 2.3.2 Einfluss der Kommentare auf die Quellcodelesbarkeit

Ein in der Literatur umstrittener Faktor für die Quellcodelesbarkeit ist die Verwendung von Kommentaren. Salviulo stellte in seiner Studie fest, dass erfahrene Entwickler für die Codelesbarkeit Bezeichner wichtiger als Kommentare ansehen [SS14]. Teilweise werden Kommentare sogar bewusst ignoriert. Dagegen sind Kommentare bei Studenten entscheidend für das Verständnis vom Quellcode. Aussagekräftige Bezeichnernamen erklären zwar den Inhalt, doch nicht den Grund für bestimmte Entscheidungen. Auch hierfür sind Kommentare weiterhin notwendig.

Woodfield et al. untersuchten den Einfluss von Kommentaren auf das Codeverständnis [WDS81]. 48 Studienteilnehmer wurden in Gruppen aufgeteilt und sollten 20 Fragen zu einem Quelltext (geschrieben in Fortran) beantworten, der entweder keine oder nur Dokumentationskommentare besaß. Die Teilnehmer mit kommentiertem Code konnten in der vorgegebenen Zeit mehr Fragen beantworten und erzielten im Schnitt eine höhere Punktzahl. Alle Teilnehmer haben den Code gleich gut verstanden, Kommentare halfen jedoch dabei, den Code schneller zu verstehen. Die Variablen wurden dabei anonymisiert, um diesen Einfluss auf die Codelesbarkeit zu minimieren. Dagegen verwendete Sheppard [SBCL78] in seiner Studie aussagekräftige Bezeichnernamen und stellte fest, dass Kommentare in diesem Fall der Codelesbarkeit nicht schaden, diese aber auch nicht signifikant verbessern.

Takang et al. [TGM96] untersuchten den wechselseitigen Einfluss von Kommentaren und Bezeichnernamen. In ihrer Studie beantworteten 89 Informatikstudenten, aufgeteilt in vier Gruppen, 15 Fragen zu einem Quelltext (geschrieben in Modula-2) mit vorgegebenen Antwortmöglichkeiten. Zusätzlich bewerteten die Teilnehmer die Quellcodelesbarkeit der Programme auf einer siebenstufigen Skala. Der Fragebogen zeigte, dass kommentierter Code lesbarer ist. Die subjektive Einschätzung bestätigte diese Erkenntnis jedoch nicht. Beide Methoden legten nahe, dass die gemeinsame Nutzung von Kommentaren und aussagekräftigen Bezeichnernamen keinen signifikanten Einfluss auf das Codeverständnis hatte.

Tenny [Ten88] erforschte den Einfluss von Kommentaren und drei Arten von Prozeduren auf die Lesbarkeit von PL/I-Quellcodes. An seiner Studie nahmen 157 Studierende teil, welche 12 Fragen zu einer von sechs Programmvarianten beantworten sollten. Dabei erzielten die Teilnehmer mit Kommentaren, unabhängig von der Art der Prozeduren, jeweils mehr Punkte

als die Teilnehmer ohne Kommentare. Ein signifikanter Unterschied wurde jedoch nur bei der Programmvariante ohne Prozeduren festgestellt.

Weitere Vorschläge für die Nutzung von Kommentaren basieren auf Erfahrungswerten. So wird etwa in den Java Code Conventions [Sun97] vermerkt, dass zu viele Kommentare auf eine schlechte Codequalität hindeuten können. Die besten Kommentare weisen auf bedeutende Details hin oder verschaffen einen größeren Überblick über die Vorgänge. Sie sollten weder wiederholen, was der Code bereits sagt, noch diesem widersprechen [KP99, Red00].

In den vorgestellten Arbeiten wurde der Einfluss von unterschiedlichen Kommentararten auf die Quellcodelesbarkeit überwiegend vernachlässigt. Zudem wurde die Auswirkung bei unterschiedlichen Wartungstätigkeiten bisher noch gar nicht erforscht. Die größtenteils mit Studenten durchgeführten Studien machen es außerdem schwer, die Erkenntnisse auf erfahrene Softwareentwickler zu übertragen. Schließlich ist es fraglich, inwiefern die Erkenntnisse der Studien hinsichtlich deren Alter (ca. 20 bis 40 Jahre alt) und der eingesetzten Programmiersprachen (etwa Fortran, PL/I, Modula-2) heute noch zutreffen. Das Ziel dieser Arbeit ist es, diese Fragen zu erforschen.

## 2.4 Messung der Programmiererfahrung

Die Programmiererfahrung beschreibt die Erfahrung, welche Softwareentwickler mit dem Schreiben und Verstehen von Quellcode gesammelt haben. Je mehr Erfahrung ein Entwickler erlangt hat, desto einfacher fällt es ihm, unbekannten Quellcode zu verstehen [FKL+12, SGR+15]. Deshalb muss dieser Einfluss berücksichtigt werden, wenn die Codelesbarkeit untersucht wird. Dazu muss zunächst die Programmiererfahrung von Studienteilnehmern bestimmt werden.

In der Literatur gibt es keine einheitliche Methode, um die Erfahrung von Programmierern zu messen. Feigenspan et al. [FKL+12] fassten 161 Paper zusammen, die eine Messung der Programmiererfahrung beinhalten (Abbildung 2.2). Etwa ein Viertel der untersuchten Studien gibt an, die Programmiererfahrung gemessen zu haben, nennt jedoch nicht die konkreten Methoden. Ein weiteres Viertel der Studien untersucht die Erfahrung überhaupt nicht, was die Ergebnisse verfälschen kann.

Die weitaus meistgenannte Methode zur Einschätzung der Programmiererfahrung ist die Anzahl der Jahre, die ein Entwickler mit dem Programmieren verbracht hat (entweder insgesamt, oder nur während seiner beruflichen Tätigkeit). Außerdem wird der Bildungsstand

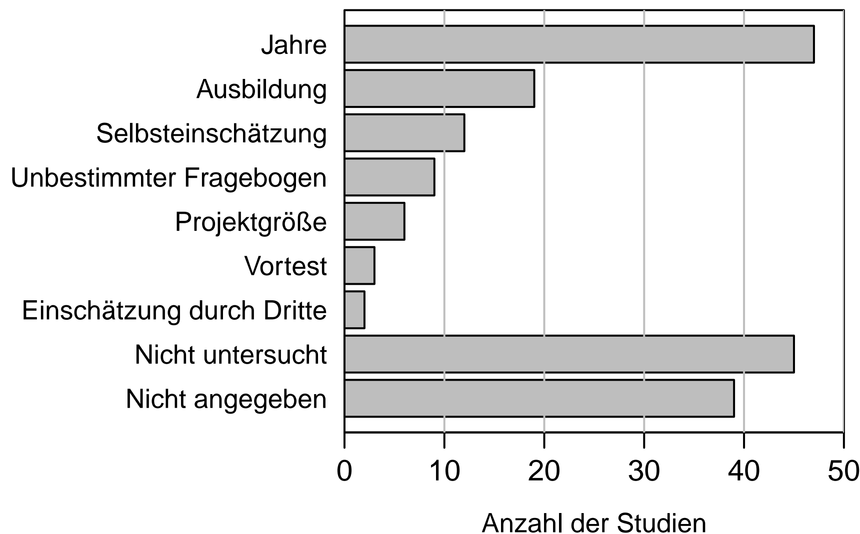


Abbildung 2.2: Methoden zur Messung der Programmiererfahrung in der Literatur [FKL+12]

als Kriterium verwendet, etwa ein abgeschlossenes Studium oder eine Ausbildung. Andere Forscher lassen die Studienteilnehmer eine Selbsteinschätzung zur Programmiererfahrung abgeben. Vereinzelt wird nach der Größe der Programme gefragt, welche die Entwickler geschrieben haben.

Feigenspan et al. führten selbst eine Nutzerstudie mit Studenten durch, um die Zuverlässigkeit dieser Methoden zu untersuchen. Sie stellten einen Fragebogen aus den genannten Kategorien zusammen, den die Teilnehmer vor dem Experiment beantworten sollten. Das Experiment bestand aus zehn Aufgaben mit jeweils einem Quellcodestück und einer dazugehörigen Frage, die das Verständnis überprüfte. Die Forscher suchten schließlich nach starken Korrelationen zwischen den Angaben zur Erfahrung und der Anzahl der richtigen Antworten. Mithilfe einer schrittweisen Regression ermittelten sie zwei Angaben, welche die größte Korrelation aufzeigten: die Erfahrung mit logischen Programmiersprachen sowie die geschätzte Erfahrung im Vergleich zu anderen Kommilitonen. Da an dieser Studie ausschließlich Studenten teilnahmen, unterschieden sich etwa die Angaben zur Programmiererfahrung oder zum Studienabschluss kaum. Daher empfehlen die Forscher, für weitere Studien Fragen aus unterschiedlichen Kategorien zu verwenden.

Kapitel 3.2.1 beschreibt, auf welche Weise die Programmiererfahrung der Studienteilnehmer bestimmt und als Störfaktor in der Studie berücksichtigt wird.

# 3 Experimentdefinition

## 3.1 Motivation einer Nutzerstudie

Die ersten Ansätze zur Berechnung der Quellcodelesbarkeit wurden in Kapitel 2.2.2 vorgestellt, doch eine subjektive Einschätzung durch Softwareentwickler besteht als Standard in der Wissenschaft fort [Sed16]. Daher wurde für die Beantwortung der wissenschaftlichen Fragen dieser Arbeit eine Nutzerstudie durchgeführt.

In der Literatur werden überwiegend die folgenden drei Methoden zur Bestimmung der Quellcodelesbarkeit verwendet.

**Bewertung.** In den Studien von Buse und Dorn (Kapitel 2.2.2) bewerteten die Probanden verschiedene Codebeispiele auf einer Skala von „unlesbar“ bis „sehr gut lesbar“. Wie bereits erwähnt, wird bei dieser Vorgehensweise nicht erfasst, inwieweit die Teilnehmer die Codebeispiele wirklich verstanden haben, was hinsichtlich der Definition der Codelesbarkeit zwingend erforderlich ist (vgl. Kapitel 2.2.1).

**Fragebogen.** Andere Forscher wie Woodfield [WDS81] oder Tenny [Ten88] verwenden Fragebögen, um das Quellcodeverständnis der Probanden zu überprüfen (vgl. Kapitel 2.3.2). Bei vorgegebenen Antwortmöglichkeiten besteht jedoch die Möglichkeit, die Lösungen zu raten und die Fragen durchzuklicken.

Die Studie von Takang [TGM96] zeigt jedoch, dass sich die Ergebnisse bei gleichzeitiger Verwendung von einer subjektiven Bewertungsskala und einem Fragebogen deutlich unterscheiden können (vgl. Kapitel 2.3.2).

**Programmieraufgaben.** Eine dritte Möglichkeit, die Codelesbarkeit experimentell zu bestimmen, besteht darin, den Teilnehmern Programmieraufgaben zu stellen und die Korrektheit der Lösungen sowie die Bearbeitungszeit zu untersuchen. Diesen Ansatz hat unter anderem Sheppard [SBCL78] gewählt.

Für die Nutzerstudie dieser Arbeit wird die letztgenannte Methode verwendet. Den größten Vorteil bietet sie hinsichtlich der Forschungsfrage 4, da nur bei dieser Methode Wartungstätigkeiten, wie in Kapitel 2.1 vorgestellt, simuliert werden können.

## 3.2 Umgang mit Störfaktoren

Bei dieser Nutzerstudie werden die Teilnehmer in drei Gruppen aufgeteilt, die jeweils mit einer anderen Kommentarart bzw. unkommentiertem Quellcode arbeiten. Damit können Unterschiede bei der Fragenbeantwortung zwischen den Gruppen festgestellt werden, die jedoch nicht zwingend auf die Kommentare zurückzuführen sind. Um möglichst nur den Einfluss der Kommentare auf die Quellcodelesbarkeit zu untersuchen, müssen zunächst alle Störfaktoren, welche in Kapitel 2.2.3 vorgestellt wurden, weitestgehend beseitigt werden.

### 3.2.1 Programmiererfahrung

#### 3.2.1.1 Fragebogen zur Bestimmung der Programmiererfahrung

Wie in Kapitel 2.4 beschrieben, ist die Programmiererfahrung der Teilnehmer ein wichtiger Störfaktor. Erfahrene Entwickler lösen die Aufgaben in der Regel schneller als beispielsweise Studenten [SS14]. Um diesen Faktor auszuschließen, muss sichergestellt werden, dass die Gruppen etwa gleiche Erfahrungsstände besitzen. Hierfür wird die Vorarbeit von Feigenspan [FKL+12] verwendet (vgl. Kapitel 2.4), um die Erfahrung der Studienteilnehmer mittels eines Fragebogens einzuschätzen. Der Vorschlag, unterschiedliche Kriterien zu verwenden, wird in dieser Arbeit umgesetzt. Die folgenden drei Angaben wurden für den Fragebogen ausgewählt.

**Selbsteinschätzung**. Die Studie von Feigenspan zeigte, dass die Selbsteinschätzung zumindest bei Studenten ein gutes Maß zur Bestimmung der Programmiererfahrung ist. Daher wurde diese Angabe in den Fragebogen aufgenommen. Die Skala reicht von Eins bis Sechs, wobei Eins der niedrigste Wert ist. Es wird bewusst eine geradzahlige Skala verwendet, damit Teilnehmer eine Tendenz hinsichtlich ihrer Erfahrung angeben müssen.

**Bildung**. In einigen Studien werden Studierende mit professionellen Entwicklern verglichen. Daher wurde für den Fragebogen zusätzlich die Frage nach einer abgeschlossenen Ausbildung ausgewählt. Diese Angabe wurde auf den „Bereich Informatik“ eingeschränkt, um einerseits

verwandte Studienfächer wie Computermathematik einzuschließen, andererseits sollten Abschlüsse aus nichttechnischen Bereichen keinen Einfluss auf die Programmiererfahrung haben.

**Jahre**. Die Anzahl der Jahre ist die in der Literatur am häufigsten verwendete Metrik zur Bestimmung der Programmiererfahrung. Oft wird dabei die Anzahl der *beruflichen* Jahre der Entwickler verwendet. Da auch Studenten zur Zielgruppe dieser Studie gehören, wurde stattdessen nach dem Beginn der Programmierung gefragt. Um unterschiedliche Interpretationen von „Beginn“ zu verhindern, wurde hinzugefügt, dass es sich um „z.B. die ersten 'Hello World'-Programme“ handelt.

Zusätzlich wurden demografische Angaben zum Geburtsjahr und Geschlecht aufgenommen, um bei der Auswertung einen Überblick über die Teilnehmer zu erhalten. Tabelle 3.1 fasst den Fragebogen zusammen, welchen die Probanden vor dem Experiment ausfüllen mussten.

| **Kategorie** | **Angabe** | **Skalen-einteilung** |
|---|---|---|
| Allgemein | Geburtsjahr | Jahr |
| | Geschlecht | m / w |
| Jahre | Anzahl Jahre in der Programmierung | Intervalle[1] |
| Bildung | Abgeschlossene Ausbildung im Bereich Informatik | ja / nein |
| Selbsteinschätzung | Programmiererfahrung | 1-6 |

Tabelle 3.1: Fragebogen

#### 3.2.1.2 Bestimmung eines Erfahrungswertes

Um die Teilnehmer bzgl. ihrer Erfahrung gleichmäßig in die Gruppen zu verteilen, muss aus den genannten Angaben zunächst ein Erfahrungswert bestimmt werden. Es gibt keine Anhaltspunkte, dass ein Kriterium einen größeren Einfluss auf die Erfahrung hat als ein anderes. Daher sollen die Angaben gleich gewichtet in die Berechnung des Erfahrungswertes einfließen. Ausgehend von der 6-stufigen Skala für die Selbsteinschätzung wird eine ebenfalls 6-stufige Skala für den Erfahrungswert verwendet. Bei einer ausreichend großen Anzahl der Teilnehmer ließen sich so genauere Erkenntnisse zwischen unterschiedlich erfahrenen Teilnehmern gewinnen. Bei dieser Nutzerstudie wurden die Probanden dagegen nachträglich in zwei Gruppen (Erfahrungswert $1-3$ sowie $4-6$) zusammengefasst (siehe Kapitel 4.1.4).

[1] Siehe Tabelle 3.2

Die Frage nach der Bildung kann nur mit *ja* oder *nein* beantwortet werden. Um die Gewichtung gleich zu halten, wird die Antwort „nein“ auf die erste Hälfte der Skala abgebildet (Werte 1-3 mit Durchschnittswert 2), während die Antwort „ja“ auf die zweite Hälfte abgebildet wird (Werte 4-6 mit Durchschnittswert 5). Diese Zuordnung stellt sicher, dass die Angabe zur Bildung den Erfahrungswert nicht unverhältnismäßig in eine Richtung lenkt.

Für die Anzahl der Jahre mussten Intervalle bestimmt werden, welche auf die sechs Erfahrungspunkte abgebildet werden kann. Als Hilfestellung wurde die bisher größte internationale Umfrage unter den Softwareentwicklern von Stack Overflow[2] verwendet. An dieser Umfrage nahmen 50.000 Entwickler aus 173 Ländern teil und gaben durchschnittlich 6,5 Jahre (nicht ausschließlich professioneller) Programmiererfahrung an. Dieser Wert wurde verwendet, um die Anzahl der Jahre für die mittleren Werte 3 und 4 zu bestimmen. Nun stellte sich die Frage, welche Anzahl von Jahren auf den höchsten Wert abgebildet werden soll. Es gibt unterschiedliche Angaben für die benötigte Anzahl von Jahren, um als *Experte* zu gelten [SRAM06]. Ericsson [EC95] spricht dabei von zehn Jahren Vollzeitvorbereitung. Fünf Jahre Ausbildung zusammen mit zehn Jahren professioneller Arbeit ergeben die für den höchsten Erfahrungswert geschätzte Anzahl von 15 Jahren.

Tabelle 3.2 fasst die gewählte Intervallskala für die Anzahl der Jahre, sowie alle Abbildungen auf entsprechende Punktwerte zusammen.

| **Punkte** | **Selbsteinschätzung** | **Jahre** | **Bildung** |
|---|---|---|---|
| 1 | 1 | $< 1$ | - |
| 2 | 2 | $1 - 3$ | nein |
| 3 | 3 | $3 - 6$ | - |
| 4 | 4 | $6 - 10$ | - |
| 5 | 5 | $10 - 15$ | ja |
| 6 | 6 | 15+ | - |

Tabelle 3.2: Zuordnung der Antwortmöglichkeiten zu einer Punktanzahl für die Berechnung des Erfahrungswertes

Der endgültige Erfahrungswert für alle drei Angaben kann nun berechnet, normalisiert und auf eine ganze Zahl gerundet werden:

$$\text{Erfahrungswert} = \text{runden}\left(\frac{\text{Selbsteinschätzung} + \text{Jahre} + \text{Bildung}}{3}\right)$$

[2] http://stackoverflow.com/research/developer-survey-2016#developer-profile-experience

### 3.2.2 Bezeichnernamen

Kapitel 2.2.3 beschreibt unter anderem den großen Einfluss von Bezeichnernamen auf die Quellcodelesbarkeit. In dieser Arbeit soll ausschließlich der Einfluss von Kommentaren untersucht werden, daher wurden alle Bezeichner in den Programmieraufgaben anonymisiert. Woodfield et al. sind bei ihrer Studie ähnlich vorgegangen [WDS81]. Diese Entscheidung wirkt sich jedoch auf die Interpretation der Studienergebnisse aus. In der Praxis müssen aussagekräftige Bezeichnernamen nicht zusätzlich kommentiert werden. Der Einfluss von Kommentaren wird in dieser Arbeit daher unter der Einschränkung untersucht, dass der Quellcode keine sprechenden Bezeichnernamen verwendet. Die Bezeichner für die Programmieraufgaben dieser Studie werden nach dem folgenden Muster benannt.

Variablen werden nach ihrem zugehörigen Typ benannt (String $string$, int $number$, boolean $bool$, char $ch$). Laufvariablen in Schleifen heißen $i$, $j$, $k$. Bei Arrays wird dem Bezeichner ein „s“ hinzugefügt (String[] $strings$, int[] $numbers$). Listen bekommen ein „list“-Suffix (List<String> $stringList$). Falls mehrere Variablen vom gleichen Typ in einem Beispiel verwendet werden, werden diese durchnummeriert (int $number1$, $number2$).

Methoden werden $foo()$, $bar()$ und $qux()$ benannt. Klassen heißen $Class1$, $Class2$, ....

### 3.2.3 Sonstige Störfaktoren

Sonstige in Kapitel 2.2.3 vorgestellten Störfaktoren werden kontrolliert, indem gleiche Bedingungen für alle Teilnehmer der Studie gelten. So wird etwa stets die gleiche Syntaxhervorhebung sowie Codestruktur (Leerzeilen, Einrückungen) verwendet. Die Aufgaben setzen außerdem kein fachspezifisches Wissen voraus. Zusätzlich sind alle Aufgaben sowie Kommentare auf Deutsch verfasst, um den Einfluss unterschiedlicher Fremdsprachenkenntnisse zu verringern.

## 3.3 Auswahl der Aufgaben

Diese Arbeit untersucht, welchen Einfluss Kommentare auf die Quellcodelesbarkeit haben. Im Unterschied zu bisherigen Studien soll dabei zusätzlich untersucht werden, ob es einen Unterschied zwischen den in Kapitel 2.3.1 vorgestellten Implementationskommentaren (Inline) und Dokumentationskommentaren (Javadoc) gibt. Daher werden die Teilnehmer des Experiments in drei Gruppen eingeteilt, welche jeweils Aufgaben mit einer anderen Kommentarart lösen sollen.

| Aufgabe | Wartungstätigkeit | Gruppe A | Gruppe B | Gruppe C |
|---|---|---|---|---|
| 1 | Verwenden | Keine | Inline | Javadoc |
| 2 | | Inline | Javadoc | Keine |
| 3 | | Javadoc | Keine | Inline |
| 4 | Fehler beheben | Inline | Javadoc | Keine |
| 5 | | Javadoc | Keine | Inline |
| 6 | | Keine | Inline | Javadoc |
| 7 | Erweitern | Javadoc | Keine | Inline |
| 8 | | Keine | Inline | Javadoc |
| 9 | | Inline | Javadoc | Keine |

Tabelle 3.3: Kommentararten in den drei Versionen der Studie

In Kapitel 2.1 wurden fünf Wartungstätigkeiten vorgestellt, die Softwareentwickler in ihrer täglichen Arbeit durchführen. Es stellt sich die Frage, ob der Einfluss von Kommentararten auf die Quellcodelesbarkeit bei unterschiedlichen Tätigkeiten gleich groß ist. Das Lesen von Quellcode ist ein Teil von jeder anderen Wartungstätigkeit. Ein Quelltext kann beispielsweise nicht sinnvoll erweitert werden, wenn er nicht vorher gelesen und verstanden wird. Daher muss diese Wartungstätigkeit nicht getrennt betrachtet werden. Außerdem wird das Refactoring für diese Studie ausgeschlossen. In diesem Fall wird nur die Syntax umgeformt, um die Lesbarkeit des Codes zu verbessern. Da die Codelesbarkeit subjektiv ist, sind keine sinnvollen Aufgabenstellungen möglich sind, deren Lösungen auf Richtigkeit überprüft werden können. Somit werden für diese Arbeit drei Wartungstätigkeiten betrachtet: Code erweitern, Code verwenden und Fehler beheben.

Zusätzlich soll jeder Teilnehmer mit jeder Kommentarart bzw. keinen Kommentaren arbeiten. Die Aufgaben müssen hierbei jedoch unterschiedlich sein. Wenn ein Teilnehmer die gleiche Aufgabe mit und ohne Kommentare löst, werden die Ergebnisse bei der zweiten Variante von dem Wissen aus der ersten Variante beeinflusst [SGR+15]. Aus diesen drei Kommentararten sowie drei Wartungstätigkeiten ergeben sich somit insgesamt neun Aufgaben, die in der Tabelle 3.3 zusammengefasst sind.

Um möglichst viele Teilnehmer für das Experiment zu motivieren, wurde die Bearbeitungsdauer auf etwa eine Stunde ausgelegt. Dadurch sind die Quellcodestücke einfach gehalten (7 bis 23 Zeilen, unkommentiert) und benötigen kein fortgeschrittenes Java-Wissen.

Die verwendeten Javadoc-Kommentare beschreiben den Zweck der Methoden/Klassen und beinhalten alle Parameter einschließlich Einschränkungen (beispielsweise die Verwendung von null) sowie den Rückgabewert. Inline-Kommentare erklären die Semantik einzelner Code-

zeilen, beispielsweise den Zweck einer Variable oder Verzweigung. Triviale Kommentare, die keine zusätzlichen Informationen bereitstellen, werden nicht verwendet, etwa:

```
i = 0; // Variable i den Wert 0 zuweisen
```

Die Aufgaben mit unterschiedlichen Kommentararten sind im Anhang B zusammengefasst. Varianten ohne Kommentare werden nicht aufgeführt, da sie trivial zu erhalten sind. Mit Ausnahme der Aufgabe 6, welche einen absichtlichen Fehler enthält, kompilierten alle Programme und lieferten das erwartete Ergebnis.

## 3.4 Feedback-Fragebogen

Am Ende des Experiments wurde den Teilnehmern ein Feedback-Fragebogen präsentiert, um weitere Informationen für die Auswertung der Studie zu bekommen. Den Probanden wurde ermöglicht, Angaben zu den verwendeten Hilfsmitteln sowie sonstige Bemerkungen zu machen. Außerdem wurden die Teilnehmer nach einer subjektiven Einschätzung zum Einfluss der Kommentararten auf die Bearbeitungszeit gefragt. Für diese Frage wurde eine Likert-Skala mit fünf Antwortmöglichkeiten gewählt, von „deutlich langsamer“ bis „deutlich schneller“ (Tabelle 3.4). Eine ungeradzahlige Skala bietet sich hierfür an, da die Teilnehmer keine Auswirkung von Kommentaren festgestellt haben könnten.

Welche Auswirkung hatten Kommentare auf Deine Bearbeitungszeit im Vergleich zum unkommentierten Code?

| | deutlich langsamer | etwas langsamer | gleich schnell | etwas schneller | deutlich schneller |
|---|---|---|---|---|---|
| Zeilenkommentare (//) | ☐ | ☐ | ☐ | ☐ | ☐ |
| Dokumentationskommentare (Javadoc) (/** */) | ☐ | ☐ | ☐ | ☐ | ☐ |

Tabelle 3.4: Fragebogen zur Auswirkung von Kommentaren auf die Bearbeitungszeit

## 3.5 Technische Umsetzung der Studie

In den meisten verwandten Arbeiten findet eine kontrollierte Experimentdurchführung statt. Der Experimentleiter befindet sich dort in einem Raum mit den Teilnehmenden. Im wissen-

schaftlichen Umfeld bieten sich hierfür Studierende der Informatik an, um eine repräsentative Anzahl an Probanden zu bekommen. Dadurch lassen sich solche Ergebnisse jedoch nicht auf erfahrene Softwareentwickler übertragen [Sie12]. Es ist allerdings schwer, professionelle Entwickler für wissenschaftliche Studien zu motivieren [Fos93a].

Um möglichst viele Teilnehmer mit unterschiedlicher Erfahrung zu erhalten, wurde dieses Experiment daher online[3] durchgeführt. Online-Umfragen haben einige Vorteile gegenüber klassischen Befragungsmethoden. Die Erhebungskosten sind vergleichsweise gering, es werden viele Personen in einer kurzen Zeit erreicht, die Befragung ist zeit- und ortsunabhängig, die Anonymität der Teilnehmer wird gewährleistet und alle Daten liegen bereits digital vor, sodass sofort mit der Auswertung begonnen werden kann. Ein großer Nachteil besteht darin, dass keine Kontrolle über die Probanden und die Untersuchungsbedingungen möglich ist. Außerdem muss sicher gestellt werden, dass sich die Umfrage in allen gängigen Browsern gleich verhält [PS10, Gör03].

Die Werbung zur Teilnahme an dieser Studie erreichte sowohl Studenten unterschiedlicher Universitäten als auch professionelle Entwickler. Dazu gehörten insbesondere über 1600 Mitglieder der Softwerkskammer[4], einer Community, die sich unter anderem für die Softwarequalität interessiert.

Zur Umsetzung des Experiments wurde das quelloffene Tool LimeSurvey[5] verwendet. Dieses Tool unterstützt das Aufzeichnen von Bearbeitungszeiten einzelner Aufgaben, was für diese Studie wesentlich ist. Der Code von LimeSurvey wurde unter anderem um eine Syntaxhervorhebung sowie eine Schaltfläche zum Kopieren und Einfügen der Quelltexte erweitert, um die Aufgabenbearbeitung zu vereinfachen.

Für jede Gruppe wurde eine separate Umfrage erstellt, welche die entsprechenden Quellcodevarianten enthielt (vgl. Tabelle 3.3). Um die Teilnehmer ihrer Erfahrung nach gleichmäßig auf die Gruppen zu verteilen, wurde eine zusätzliche Umfrage erstellt, die ausschließlich den Fragebogen aus Tabelle 3.1 enthielt. Ein PHP-Skript wertete die Antworten aus und berechnete daraus einen Erfahrungswert (vgl. Kapitel 3.2.1.2). Die Teilnehmer wurden anschließend an diejenige Gruppe weitergeleitet, welche die wenigsten Teilnahmen mit dem berechneten Erfahrungswert enthielt (vgl. Abbildung 3.1). Es wurde außerdem sichergestellt, dass keine der drei Hauptumfragen gestartet werden kann, ohne vorher den Fragebogen ausgefüllt zu haben.

[3] `http://dev-ovgu.de`
[4] `https://www.softwerkskammer.org/`
[5] `https://www.limesurvey.org/`

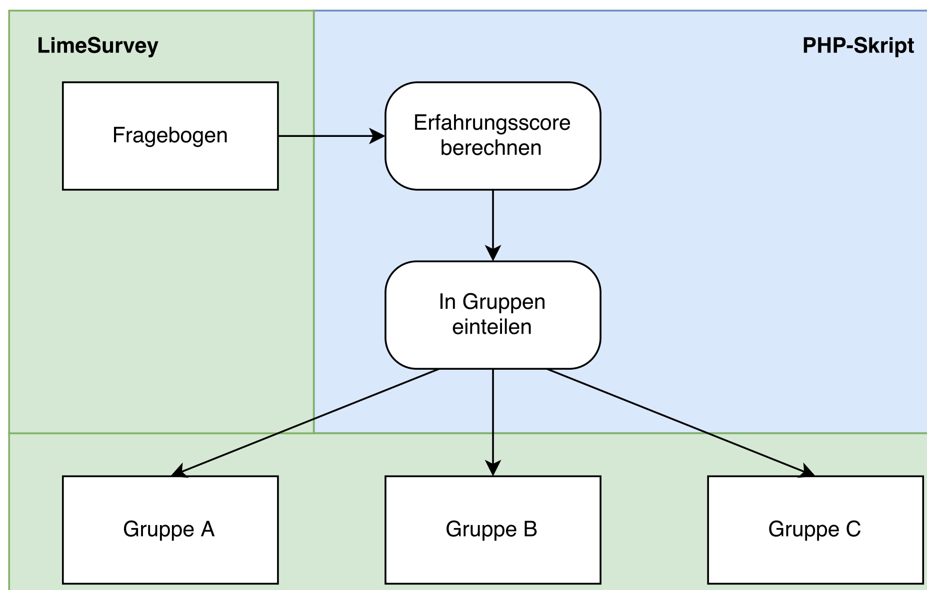


Abbildung 3.1: Zuteilung der Teilnehmer in Gruppen

Auf der Startseite wurde zunächst die Motivation der Studie mit „Beitrag zur Codelesbarkeit“ erklärt. Um die Teilnehmer nicht zu beeinflussen, wurden Kommentare erstmalig beim Feedback-Bogen erwähnt. Den Probanden wurde zugesichert, dass ihre Angaben anonym und vertraulich behandelt werden. Die geschätzte Dauer des Experiments wurde mit etwa einer Stunde angegeben. Die Teilnehmer wurden darum gebeten, das Experiment alleine und möglichst ohne Unterbrechung durchzuführen. Hilfsmittel, wie eine Entwicklungsumgebung oder Internetsuche, wurden jedoch zugelassen. Grundlegende Kenntnisse in Java war die einzige Voraussetzung für die Teilnahme an der Studie.

Vor der Veröffentlichung des Experiments wurde ein Vortest mit fünf Teilnehmern durchgeführt, um eventuelle Probleme festzustellen. Aufgrund des Feedbacks der Teilnehmer wurden kleine Fehler behoben sowie unklare Formulierungen berichtigt.

# 4 Ergebnisse

Dieses Kapitel präsentiert die Ergebnisse der Studie. Zunächst werden die erhobenen Daten aufbereitet und ausgewertet. Anschließend werden die Ergebnisse diskutiert und die Forschungsfragen beantwortet.

Das Experiment erfolgte im Zeitraum vom 01.06.2016 bis zum 11.07.2016. Die Studie wurde auf unterschiedlichen Wegen beworben: Entwickler- und Studierendenforen, E-Mail-Verteiler mehrerer Universitäten und Unternehmen, Softwerkskammer, soziale Netzwerke, sowie Aushänge. Somit erreichte die Werbung sowohl Studierende als auch professionelle Entwickler.

Die statistische Auswertung erfolgte mithilfe der Programmiersprache R[1].

## 4.1 Datenaufbereitung

Bevor mit der Auswertung der Studie begonnen werden kann, muss der Datensatz zunächst auf ungültige Werte überprüft und bereinigt werden. Untersucht werden hierbei sowohl die Bearbeitungszeiten als auch die Inhalte der Aufgabenlösungen. Anschließend wird der Datensatz für die Auswertung vorbereitet.

### 4.1.1 Umgang mit Abbrechern und fehlenden Werten

Insgesamt 416 Teilnehmer nahmen an der Studie teil. 131 Probanden füllten jedoch nur den Fragebogen zur Bestimmung der Programmiererfahrung aus, ohne eine einzige Aufgabe zu bearbeiten (vgl. Abbildung 3.1). Da hierbei keine Auswertung der Lösungen oder Bearbeitungszeiten möglich ist, wurden diese Fälle vollständig aus der Studie entfernt.

Ein Problem bei Onlineumfragen stellen die sog. „Durchklicker“ dar [FR07]. Diese Teilnehmer füllen den Fragebogen in einer so kurzen Zeit aus, dass nicht von einer sorgfältigen

[1] https://www.r-project.org/

Auseinandersetzung mit der Frage ausgegangen werden kann. Bei diesem Experiment werden Durchklicker am Inhalt ihrer Antworten, welche keine sinnvolle Lösung darstellen, sowie den ungewöhnlich niedrigen Bearbeitungszeiten (unter 10 Sekunden) sofort erkannt. Acht Fälle wurden von der Auswertung ausgeschlossen.

Nur 157 von den verbleibenden 277 Teilnehmern führten die Studie vollständig durch. Es gibt mehrere mögliche Gründe, warum die Studie von einigen vorzeitig abgebrochen wurde. Einige Probanden gaben an, das Experiment entspreche nicht ihrer Vorstellung. Die Aufgaben seien teilweise zu algorithmisch und praxisfern. Ein mangelndes Interesse könnte also zum Abbruch der Studie geführt haben. Andere Teilnehmer unterschätzten eventuell den zeitlichen Aufwand der Studie. Eine weitere Ursache könnte in Unterbrechungen liegen. O'Conaill und Frohlich stellten in ihrer Studie fest, dass die Teilnehmer in 40% der Unterbrechungen nicht die Arbeit fortsetzten, mit der sie sich vor der Unterbrechung beschäftigten [OF95].

Unabhängig von den Gründen muss als Erstes entschieden werden, wie die Ergebnisse von Teilnehmern behandelt werden, welche das Experiment nicht abgeschlossen haben. Da die Aufgaben unterschiedlich schwer und daher nicht zuverlässig miteinander vergleichbar sind, wird die Auswertung für jede Aufgabe einzeln durchgeführt. Somit stellen fehlende Lösungen kein Problem dar. Gleiches gilt für übersprungene Aufgaben. Hierbei wurden nur die betroffenen Antworten sowie Bearbeitungszeiten von der Auswertung ausgeschlossen.

### 4.1.2 Umgang mit Ausreißern

Ausreißer sind „Extrem hohe oder niedrige Werte innerhalb einer Reihe üblicher mäßig unterschiedlicher Messwerte, von denen fraglich ist, ob sie unter den vorgegebenen Bedingungen möglich sind“ [HS15]. Diese können aus dem Datensatz entfernt werden, sobald „zwingende sachlogische Begründungen dies rechtfertigen“.

Ausreißer können mithilfe von *Quartilen* bestimmt werden:

$$Q_1 - k(Q_3 - Q_1) \leq \textbf{kein Ausreißer} \leq Q_3 + k(Q_3 - Q_1) \qquad \text{[HS15]}$$

Die Quartile $Q_1$, $Q_2$ und $Q_3$ gliedern eine geordnete Datenreihe in vier gleich große Teile, d.h. ein Viertel der Werte ist kleiner als das (untere) Quartil $Q_1$ und ein Viertel der Werte ist größer als das (obere) Quartil $Q_3$ [Eck14].

Die Differenz $Q_3 - Q_1$ wird auch als *Interquartilsabstand* bezeichnet und entspricht der Spannweite der mittleren 50% der Werte. Ausreißer sind also Werte, die ein bestimmtes Vielfaches $k$ vom Interquartilsabstand vom unteren bzw. oberen Quartil entfernt sind.

Es gibt keine feste Vorgabe für den Parameter $k$. In der Literatur werden häufig Ausreißer für $k = 1.5$ als *milde Ausreißer* und für $k = 3$ als *extreme Ausreißer* (oder *Extremwerte*) bezeichnet [HS15, KRES13].

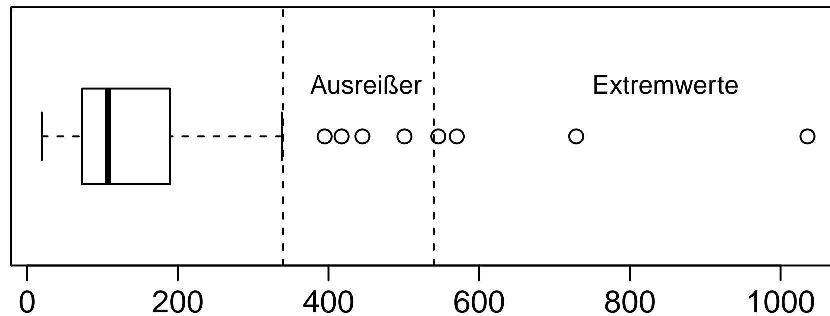


Abbildung 4.1: Boxplot für die Antwortzeiten (in Sekunden): Gruppe A, Aufgabe 1

Zur graphischen Darstellung von Ausreißern eignen sich Boxplots [KA14]. Abbildung 4.1 zeigt am Beispiel den Boxplot für die Beantwortungszeiten der Gruppe A für die erste Aufgabe. Diese Datenreihe enthält vier (milde) Ausreißer sowie vier Extremwerte. Der größte Wert beträgt hierbei 1035.75 Sekunden, also etwas über 17 Minuten. Der größte gemessene Wert in allen Gruppen beträgt 9874.58 Sekunden (ca. 164 Minuten).

Es gibt zwei mögliche Erklärungen für so ungewöhnlich hohe Bearbeitungszeiten. Erstens könnte das Experiment unterbrochen worden sein. Einige Teilnehmer gaben an, eine oder mehrere Pausen gemacht zu haben. Zweitens verwendeten manche Probanden eine Entwicklungsumgebung als Hilfsmittel zur Bearbeitung der Aufgaben. Die Entwicklungsumgebung bestätigte den Teilnehmern zwar die Korrektheit ihrer Lösungen, die Quellcodebeispiele zu übertragen könnte jedoch zu längeren Bearbeitungszeiten führen.

Unterbrechungen verzerren zwar die reinen Bearbeitungszeiten, sie gehören jedoch zum Alltag eines jeden Entwicklers. Durchschnittlich werden Programmierer bei ihrer Arbeit etwa vier Mal in einer Studie unterbrochen [OF95]. Für die Analyse sind Extremwerte dennoch ungünstig, da sie die Ergebnisse teilweise stark verfälschen können. Da ihre Ursache nachvollziehbar ist, werden Extremwerte aus dieser Studie daher entfernt. Diese Bearbeitungszeiten werden somit aus allen Berechnungen ausgeschlossen, die Lösungen können jedoch für die Prüfung auf Richtigkeit weiterhin verwendet werden. Milde Ausreißer werden dagegen vollständig beibehalten.

Da diese Zeiten sowohl von der Aufgabe als auch von der Gruppe abhängen, wurden die Extremwerte einzeln pro Aufgabe und Gruppe berechnet und aus dem jeweiligen Datensatz

entfernt. 66 Extremwerte von insgesamt 1685 Messwerten wurden auf diese Weise eliminiert (ca. 3.9% aller Messwerte). Zu niedrige Ausreißer wurden mit dem hier beschriebenen Verfahren nicht gefunden. Das liegt einerseits daran, dass Durchklicker bereits entfernt wurden und andererseits daran, dass der dreifache Interquartilsabstand vom unteren Quartil meist unter 0 Sekunden liegt.

Alle nachfolgenden Angaben und Berechnungen beziehen sich auf die in den Kapiteln 4.1.1 und 4.1.2 bereinigte Datenmenge.

### 4.1.3 Darstellung der empirischen Daten

Dieses Kapitel fasst die erhobenen, bereinigten Daten zusammen und bietet einen Überblick über die Teilnehmer dieser Studie (deskriptive Statistik).

Von den nach der Datenbereinigung verbleibenden 277 Probanden sind 21 weiblich und 243 männlich. 13 Teilnehmer gaben ihr Geschlecht nicht an. Die Anzahl der Teilnehmer ist als sehr positiv zu bewerten. Buse et al. fassten im Jahr 2011 aus über 3000 wissenschaftlichen Publikationen die wichtigsten Erkenntnisse zur Verwendung von Nutzerstudien zusammen [BSW11]. Sie beschreiben Studien mit mehr als 45 Teilnehmern als „sehr groß“ und fanden bis dahin keine Studie dieser Größe, die sowohl Studenten als auch professionelle Entwickler umfasst. Über die Hälfte der Nutzerstudien wird mit weniger als 30 Teilnehmern durchgeführt.

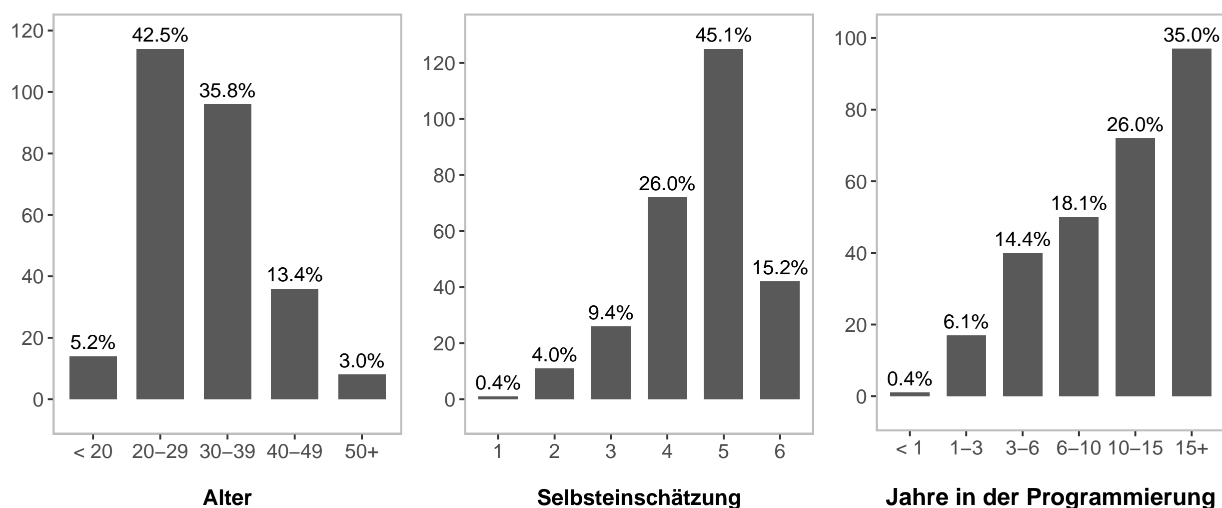


Abbildung 4.2: Angaben der Teilnehmer (y-Achse: Anzahl der Teilnehmer)

Abbildung 4.2 zeigt unter anderem die absolute Häufigkeitsverteilung über das Alter der Teilnehmer. Diese entspricht in etwa der Verteilung in der Umfrage von Stack Overflow mit über 50.000 Entwicklern. Bezüglich des Alters deutet die Verteilung also auf eine Repräsentativität der Studienteilnehmer hin.

188 Teilnehmer haben eine abgeschlossene Ausbildung im Bereich Informatik, 89 dagegen nicht. Zusammen mit den Angaben zur Selbsteinschätzung und über die Anzahl der Jahre in der Programmierung (Abbildung 4.2) wird deutlich, dass überwiegend erfahrene Entwickler an dem Experiment teilgenommen haben. Zwischen den Verteilungen der beiden Angaben gibt es dennoch Unterschiede. Über ein Drittel der Probanden programmiert seit über 15 Jahren, doch nur etwa 15% bewerten ihre Fähigkeiten mit sechs Punkten. Eine mögliche Ursache hierfür ist die Unterschätzung der eigenen Kenntnisse durch die Teilnehmer. Andererseits sind die in Kapitel 3.2.1.2 festgelegten 15 Jahre in der Programmierung möglicherweise nicht ausreichend, um als Experte zu gelten. Zukünftige Forschungen sollten bei einem ähnlichen Experimentaufbau daher einen größeren Maximalwert für die Anzahl der Jahre in der Programmierung wählen.

Im nächsten Kapitel werden die Teilnehmer entsprechend ihrer Programmiererfahrung für die Auswertung gruppiert.

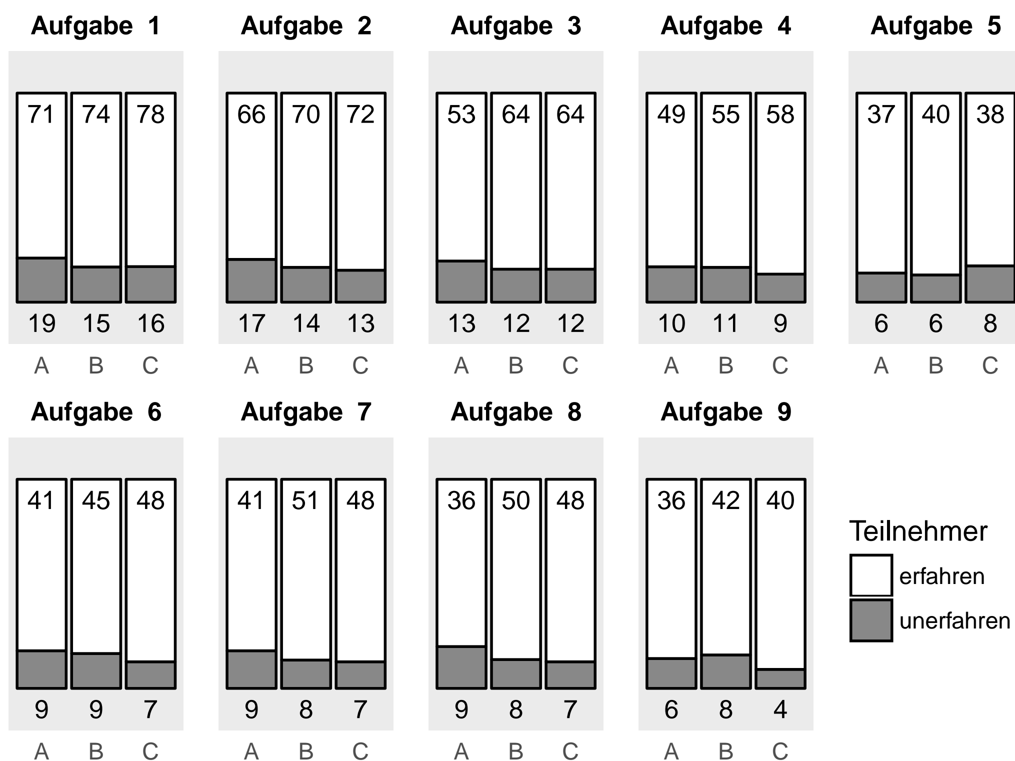


Abbildung 4.3: Anzahl der gewerteten Antworten pro Aufgabe

### 4.1.4 Gruppierung der Teilnehmer nach Erfahrung

Die Forschungsfrage 3 untersucht Unterschiede bei der Schnelligkeit und Richtigkeit der Fragenbeantwortung zwischen erfahrenen und unerfahrenen Programmierern. Um diese Frage später beantworten zu können, wurde den Teilnehmern vor dem Experiment mittels eines Fragebogens ein Erfahrungswert zwischen 1 und 6 zugewiesen (vgl. Kapitel 3.2.1). Um ausreichend viele Messwerte pro Stichprobe für die Analyse zu erhalten, werden die Probanden dem Erfahrungsstand entsprechend in nur zwei Erfahrungsgruppen zusammengefasst. Alle Teilnehmer mit einem Erfahrungswert zwischen 1 und 3 gelten als „unerfahren“, während die „erfahrenen“ Teilnehmer einen Erfahrungswert zwischen 4 und 6 haben. Abbildung 4.3 zeigt die Teilnehmerzahlen pro Aufgabe, Gruppe und Erfahrungsstand. Die Grafik bestätigt die Beobachtung in Kapitel 4.1.3, dass überwiegend erfahrene Probanden an der Studie teilnahmen.

## 4.2 Prüfung der Bearbeitungszeiten auf signifikante Unterschiede

Um den Einfluss der Kommentararten auf die Antwortzeiten zu überprüfen, werden in diesem Kapitel die Bearbeitungszeiten auf signifikante Unterschiede untersucht.

Einen ersten Überblick über die Streuung der Bearbeitungszeiten bieten die Boxplots in der Abbildung 4.4. Gruppen mit Javadoc-Kommentaren zeigen bei einigen Aufgaben eine längere Bearbeitungszeit auf als bei den anderen Gruppen. Zwischen den Medianen sind jedoch keine offensichtlich signifikanten Unterschiede erkennbar.

### 4.2.1 Definition eines Signifikanztests

Ein *statistischer Test* (auch *Hypothesentest* oder *Signifikanztest*) ist ein mathematisches Verfahren, welches überprüft, ob bestimmte Annahmen über eine Stichprobe auf die Grundgesamtheit zutreffen [HS15,FKPT07]. Hierfür werden zwei Hypothesen definiert, welche sich gegenseitig ausschließen. Die Nullhypothese $H_0$ drückt aus, dass die zu untersuchenden Unterschiede oder Zusammenhänge in der Grundgesamtheit nicht auftreten. Diese Hypothese wird solange angenommen, wie sie nicht mit einer hohen Wahrscheinlichkeit abgelehnt werden kann. Die Alternativhypothese $H_1$ ist die eigentliche Forschungsfrage. Sie kann mittels

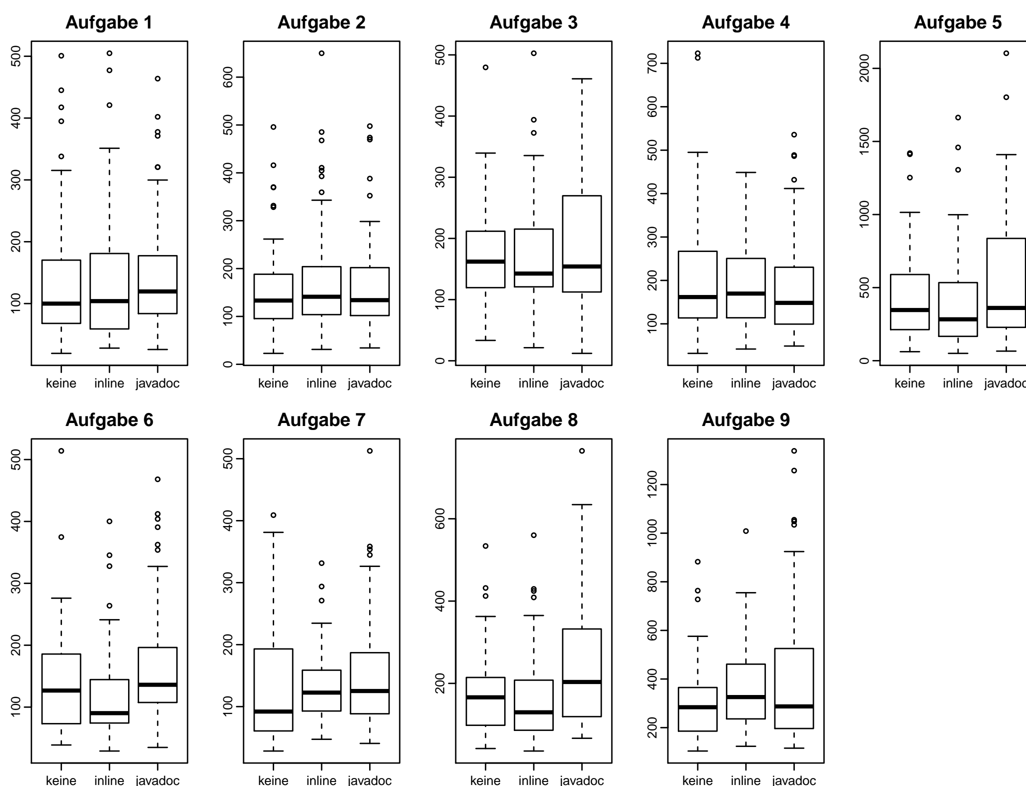


Abbildung 4.4: Bearbeitungszeiten aller Teilnehmer (in Sekunden)

eines Tests nicht bewiesen werden. Sie wird stattdessen angenommen, sobald die Nullhypothese abgelehnt wurde.

Bei einem Hypothesentest sind zwei Fehlentscheidungen möglich. Im ersten Fall wird die Nullhypothese abgelehnt, obwohl sie richtig ist (Fehler 1. Art). Die Wahrscheinlichkeit $\alpha$, dass dieser Fehler eintritt, wird als *Signifikanzniveau* bezeichnet. Ein Ergebnis wird allgemein als *signifikant* bezeichnet, wenn dessen Signifikanzniveau $\alpha$ kleiner als 5% ist [KRES13, JD06].

Eine weitere Fehlentscheidung kommt vor, wenn die Nullhypothese nicht abgelehnt wird, obwohl sie falsch ist (Fehler 2. Art). Die Wahrscheinlichkeit für diesen Fehler sinkt, wenn der Umfang der Stichprobe vergrößert oder ein Verfahren mit größerer Teststärke gewählt wird. Die *Teststärke* ist dabei „die Wahrscheinlichkeit, die [Alternativhypothese] $H_1$ anzunehmen, wenn sie auch in Wirklichkeit gilt“ [Ras06].

Statistische Tests werden in zwei Gruppen unterteilt. Parametrische Tests setzen die Kenntnis der Verteilung in der Grundgesamtheit voraus. Nichtparametrische Tests haben diese Voraussetzung nicht [Mit14]. Sie betrachten stattdessen die Rangplätze, welche den Messwerten zugeordnet werden [Ras10]. Nichtparametrische Tests zeichnen sich jedoch gegenüber

den parametrischen Tests durch eine geringere Teststärke (*Power*) aus [HS15]. Somit werden parametrische Tests in der Regel bevorzugt eingesetzt. Sofern deren mathematische Voraussetzungen jedoch verletzt oder die Stichproben sehr klein sind, werden stattdessen nichtparametrische Tests vorgezogen [Ras10].

Für die Wahl des passenden statistischen Verfahrens müssen zunächst die Testbedingungen geprüft werden. Die Messwerte der jeweiligen Gruppen stehen in keiner sich beeinflussenden Beziehung zueinander, daher sind die Stichproben *unabhängig*. Es werden drei Gruppen untersucht, somit wird ein *Mehrstichprobenverfahren* angewendet. Die Messwerte sind Zeitangaben in Sekunden mit zwei Nachkommastellen, sie können als *stetig* behandelt werden. Für diese Konstellation sind zwei Verfahren möglich, um die Bearbeitungszeiten der Gruppen auf signifikante Unterschiede zu prüfen. Sofern die Stichproben normalverteilt sind und bestimmte Bedingungen erfüllt sind (siehe unten), kann die Varianzanalyse (ANOVA) angewendet werden. Andernfalls wird der nichtparametrische Kruskal-Wallis-Test durchgeführt.

Aufgrund der größeren Teststärke der Varianzanalyse ist diese vorzuziehen. Hierfür müssen jedoch die Voraussetzungen geprüft werden [KRES13]:

**Intervallskalierte abhängige Variable.** Die Bearbeitungszeit – die abhängige Variable – ist intervallskaliert.

**Eindeutige Gruppierung der unabhängigen Variable.** Die unabhängige Variable ist die Gruppe, die aufgrund ihrer Nominalskala eindeutig gruppiert werden kann.

**Normalverteilung der Stichproben.** Kapitel 4.2.2 prüft diese Bedingung.

**Varianzhomogenität.** Die Varianzen der einzelnen Gruppen dürfen sich nicht signifikant voneinander unterscheiden. Diese Voraussetzung wird in Kapitel 4.2.3 geprüft.

### 4.2.2 Test auf Normalverteilung

Eine Normalverteilung der Bearbeitungszeiten wird für eine Varianzanalyse vorausgesetzt. Dies wird mithilfe von Anpassungstests geprüft [Mit14,HS15]. Eine vorangehende graphische Analyse wird jedoch empfohlen, da rechnerische Verfahren schon bei kleinen Unterschieden zur Normalverteilung eine starke Abweichung aufzeigen können. Daher erfolgt in diesem Kapitel zunächst eine graphische und anschließend eine rechnerische Überprüfung der Bearbeitungszeiten der Teilnehmer auf Normalverteilung.

#### 4.2.2.1 Quantile-Quantile Plot (QQ-Plot)

Bei einem QQ-Plot handelt es sich um ein Streudiagramm bestehend aus den Quantilen der Messwerte und den Quantilen der angenommenen Verteilung, hier der Normalverteilung. Diese kann für den Datensatz angenommen werden, sofern die Werte annähernd auf einer Diagonalen liegen [HS15].

Abbildung 4.5 zeigt den QQ-Plot für die Messwerte der ersten Gruppe und ersten Aufgabe. Wie zu sehen ist, weichen die Werte stark von der Diagonalen ab, sodass zunächst nicht von einer Normalverteilung des Datensatzes auszugehen ist. Die QQ-Plots für sämtliche Aufgaben und Gruppen sehen ähnlich aus, sodass bei keinem der Datensätze eine Normalverteilung vermutet wird. Dies wird im nächsten Kapitel rechnerisch überprüft.

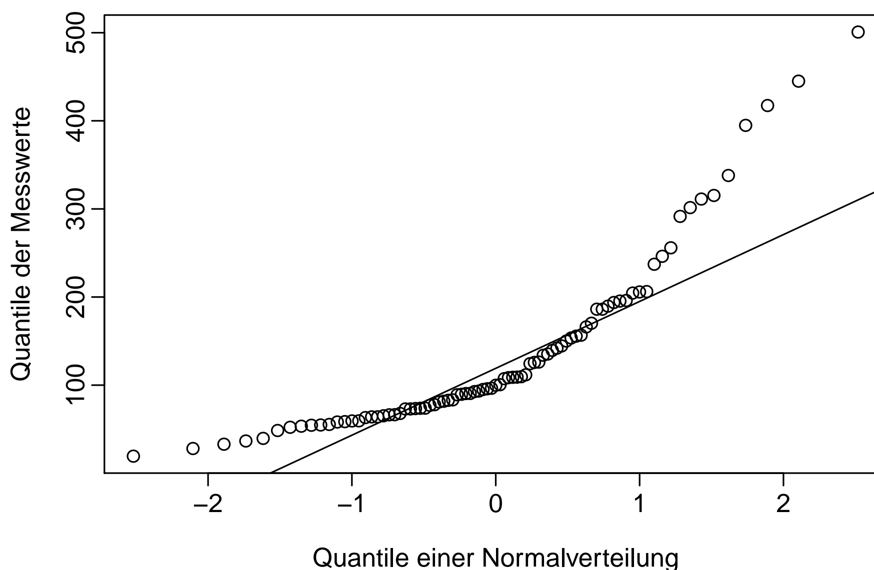


Abbildung 4.5: QQ-Plot für Gruppe A, Aufgabe 1

#### 4.2.2.2 Anpassungstests auf Normalverteilung

Ein Anpassungstest überprüft, ob die Messwerte einem bestimmten Verteilungstyp, hier der Normalverteilung, folgen [FKPT07]. Die Nullhypothese $H_0$ der folgenden Anpassungstests besagt jeweils, dass die Stichprobe normalverteilt ist. Die Alternativhypothese $H_1$ drückt dagegen aus, dass die Messwerte keiner Normalverteilung folgen. Unter dem Signifikanzniveau $\alpha = 5\%$ ist eine Stichprobe also dann nicht normalverteilt, wenn der Test einen Wahrscheinlichkeitswert $p$ kleiner als 0.05 ergibt.

Der **Shapiro-Wilk-Test** [SW65] besitzt die größte Teststärke der nichtparametrischen Anpassungstests, d.h. die geringste Wahrscheinlichkeit für einen Fehler 2. Art [HS15]. Für die Stichproben aller Gruppen und Aufgaben liefert der Test $p < 0.05$ (Tabelle 4.1). Die Nullhypothese wird demnach in allen Fällen abgelehnt, was bedeutet, dass keine Stichprobe normalverteilt ist.

| Aufgabe | Gruppe A | Gruppe B | Gruppe C |
|---|---|---|---|
| 1 | $9.8 \times 10^{-9}$ | $1.9 \times 10^{-8}$ | $4 \times 10^{-7}$ |
| 2 | $1.4 \times 10^{-7}$ | $4 \times 10^{-8}$ | $6.4 \times 10^{-7}$ |
| 3 | $4.7 \times 10^{-5}$ | 0.0036 | $3.9 \times 10^{-5}$ |
| 4 | 0.0082 | $5.7 \times 10^{-6}$ | $1.2 \times 10^{-7}$ |
| 5 | $1.5 \times 10^{-5}$ | $4.4 \times 10^{-5}$ | $1.1 \times 10^{-6}$ |
| 6 | $1.9 \times 10^{-5}$ | $6 \times 10^{-6}$ | $2.3 \times 10^{-5}$ |
| 7 | $3.9 \times 10^{-5}$ | $1.2 \times 10^{-5}$ | 0.00024 |
| 8 | 0.00021 | $8.2 \times 10^{-6}$ | $1 \times 10^{-4}$ |
| 9 | 0.0011 | $2.5 \times 10^{-6}$ | 0.00022 |

Tabelle 4.1: $p$-Werte für den Shapiro-Wilk-Test

Zwei weitere Anpassungstests, der **Lilliefors-Test** [Lil67] und der **Anderson-Darling-Test** [AD52], bestätigen, dass die Bearbeitungszeiten der Teilnehmer nicht normalverteilt sind. Somit ist die Voraussetzung der Varianzanalyse nicht erfüllt. Für ausschließlich erfahrene Teilnehmer liefern die Anpassungstests gleiche Ergebnisse. Bei unerfahrenen Teilnehmern sind einzelne Stichproben zwar normalverteilt, bei einer so kleinen Anzahl der Werte werden aber ohnehin nichtparametrische Tests bevorzugt [Ras10].

Bei ausreichend großen Stichproben ($n \geq 30$) kann der zentrale Grenzwertsatz angewendet werden. Dieser besagt, dass die Normalverteilungseigenschaft angenommen werden kann, selbst wenn die Stichproben nicht normalverteilt sind [SL15, KRES13]. Die Stichproben von sowohl allen als auch nur den erfahrenen Teilnehmern beinhalten mindestens 30 Messwerte. Daher wird mithilfe des zentralen Grenzwertsatzes für diese Stichproben eine Normalverteilung angenommen und die Voraussetzung der Varianzanalyse erfüllt. Die Stichproben der unerfahrenen Teilnehmer sind dagegen stets kleiner als 30, sodass eine Varianzanalyse für diese nicht durchgeführt werden kann.

### 4.2.3 Test auf Varianzhomogenität

Eine weitere Voraussetzung für die Varianzanalyse ist die Varianzhomogenität der Messwerte. Um diese Bedingung zu überprüfen, wird der Levene-Test [Lev60] durchgeführt. Da die

Normalverteilung für unerfahrene Probanden nicht gegeben ist, muss für diese keine Überprüfung auf Varianzhomogenität durchgeführt werden. Die Nullhypothese $H_0$ des Levene-Tests nimmt an, dass die Varianzen der Stichproben gleich sind. Die Alternativhypothese $H_1$ besagt dagegen, dass sich mindestens zwei Varianzen signifikant unterscheiden. Damit wird die Varianzhomogenität abgelehnt, sofern der Levene-Test $p < 0.05$ ergibt. Tabelle 4.2 fasst die $p$-Werte des Levene-Tests für die Bearbeitungszeiten zusammen. Sowohl bei allen als auch nur erfahrenen Teilnehmern ist die Varianzhomogenität für einige Aufgaben nicht gegeben und somit die Voraussetzung für eine Varianzanalyse nicht erfüllt (grau hervorgehoben). Daher wird einheitlich für die gesamte Auswertung der nichtparametrische Kruskal-Wallis-Test durchgeführt. Die Ergebnisse der Varianzanalyse für alle bzw. nur erfahrene Teilnehmer sind zum Vergleich in Tabelle A.1 im Anhang A aufgeführt und bestätigen die Ergebnisse des Kruskal-Wallis-Tests.

| Aufgabe | Alle Teilnehmer | Erfahrene Teilnehmer |
|---|---|---|
| 1 | 0.5835 | 0.6023 |
| 2 | 0.2626 | 0.1693 |
| 3 | **0.0294** | **0.0131** |
| 4 | 0.5919 | 0.7566 |
| 5 | 0.3576 | 0.1742 |
| 6 | 0.6046 | 0.6739 |
| 7 | 0.0976 | 0.0502 |
| 8 | **0.0270** | 0.2970 |
| 9 | **0.0362** | **0.0147** |

Tabelle 4.2: $p$-Werte für den Levene-Test (statistisch signifikante Werte sind hervorgehoben)

### 4.2.4 Kruskal-Wallis-Test

Der Kruskal-Wallis-Test [KW52] (auch *H-Test* oder *Rangvarianzanalyse*) ist eine nichtparametrische Alternative für die einfaktorielle Varianzanalyse [Ras10].

Die Hypothesen des Kruskal-Wallis-Tests besagen [HS15, Ras10]:

- **Nullhypothese $H_0$:** Alle Stichproben entstammen derselben Grundgesamtheit. Das bedeutet, dass die Verteilungen der untersuchten Gruppen gleich sind.
- **Alternativhypothese $H_1$:** Mindestens zwei Stichproben entstammen nicht derselben Grundgesamtheit. Mindestens eine Gruppe unterscheidet sich also von den anderen.

Der Kruskal-Wallis-Test stellt also fest, ob es mindestens einen statistisch signifikanten Unterschied zwischen den Stichproben gibt. Um zu bestimmen, zwischen welchen Gruppen genau der Unterschied besteht, wird ein Post-hoc-Test durchgeführt, welcher die Gruppen paarweise vergleicht. Hierfür eignet sich etwa der Dunn-Test [Din15].

Die Auswertung erfolgt im Folgenden zunächst für alle Teilnehmer und anschließend getrennt nach der Programmiererfahrung.

#### 4.2.4.1 Alle Teilnehmer

Tabelle 4.3 zeigt die $p$-Werte des Kruskal-Wallis-Tests, sowohl für alle Teilnehmer zusammen als auch getrennt nach der Programmiererfahrung.

| Aufgabe | Alle Teilnehmer | Unerfahren | Erfahren |
|---|---|---|---|
| 1 | 0.3512 | 0.2242 | 0.1074 |
| 2 | 0.5606 | 0.6421 | 0.7393 |
| 3 | 0.7047 | 0.2217 | 0.6865 |
| 4 | 0.4793 | 0.0797 | 0.7582 |
| 5 | 0.1677 | 0.5811 | 0.1651 |
| 6 | **0.0106** | 0.1074 | 0.0727 |
| 7 | 0.0728 | **0.0473** | 0.2611 |
| 8 | **0.0211** | **0.0080** | 0.2805 |
| 9 | 0.2393 | **0.0281** | 0.0564 |

Tabelle 4.3: $p$-Werte für den Kruskal-Wallis-Test

| Gruppen | Aufgabe 6 | Aufgabe 8 |
|---|---|---|
| A - B | 0.2656 | 0.9305 |
| A - C | 0.1664 | 0.0686 |
| B - C | **0.0039** | **0.012** |

Tabelle 4.4: $p$-Werte des Dunn-Tests für alle Teilnehmer

Bei allen Teilnehmern stellt der Test bei den Aufgaben 6 und 8 einen signifikanten Unterschied fest ($p < 0.05$). Anschließend wird für diese Aufgaben ein Dunn-Test durchgeführt, um den signifikanten Unterschied näher zu bestimmen. Tabelle 4.4 fasst die $p$-Werte des Dunn-Tests zusammen und zeigt jeweils einen signifikanten Unterschied zwischen den Gruppen B und C. In beiden Fällen ist die Gruppe mit Inline-Kommentaren signifikant schneller als

die Gruppe mit Javadoc-Kommentaren. Ein signifikanter Unterschied zu unkommentiertem Code wurde nicht festgestellt.

#### 4.2.4.2 Nach Erfahrung

Abbildung 4.6 und Abbildung 4.7 zeigen jeweils die Boxplots der Bearbeitungszeiten getrennt nach Programmiererfahrung der Teilnehmer. Bei Erfahrenen sind die Boxplots wie bei allen Teilnehmern eher gleichmäßig, während bei Unerfahrenen deutliche Unterschiede zu erkennen sind.

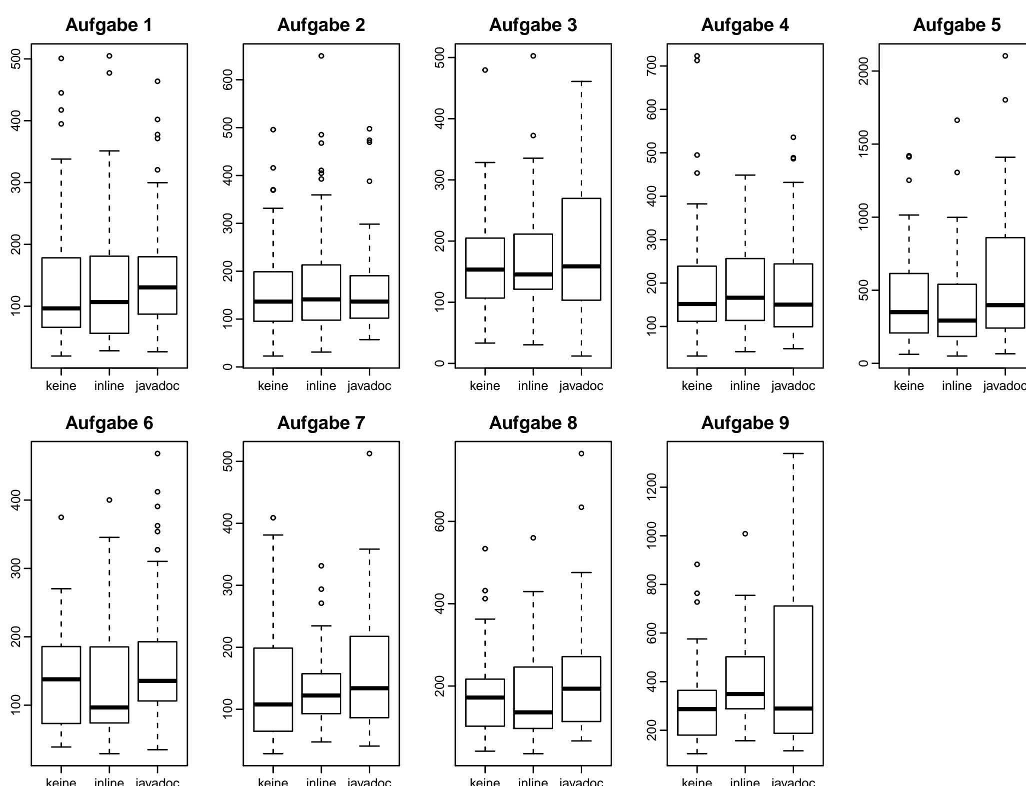


Abbildung 4.6: Bearbeitungszeiten erfahrener Teilnehmer (in Sekunden)

Der Kruskal-Wallis-Test bestätigt diese Beobachtung (Tabelle 4.3). Bei erfahrenen Teilnehmern konnte bei keiner Aufgabe ein signifikanter Unterschied festgestellt werden. Dagegen sind bei unerfahrenen Teilnehmern signifikante Unterschiede in den Aufgaben 7 bis 9 erkennbar. Ein anschließender Dunn-Test stellt fest, zwischen welchen Gruppen jeweils ein signifikanter Unterschied besteht (Tabelle 4.5).

Daraus ergeben sich die in Tabelle 4.6 dargestellten Zusammenhänge zwischen Bearbeitungszeiten und Kommentararten.

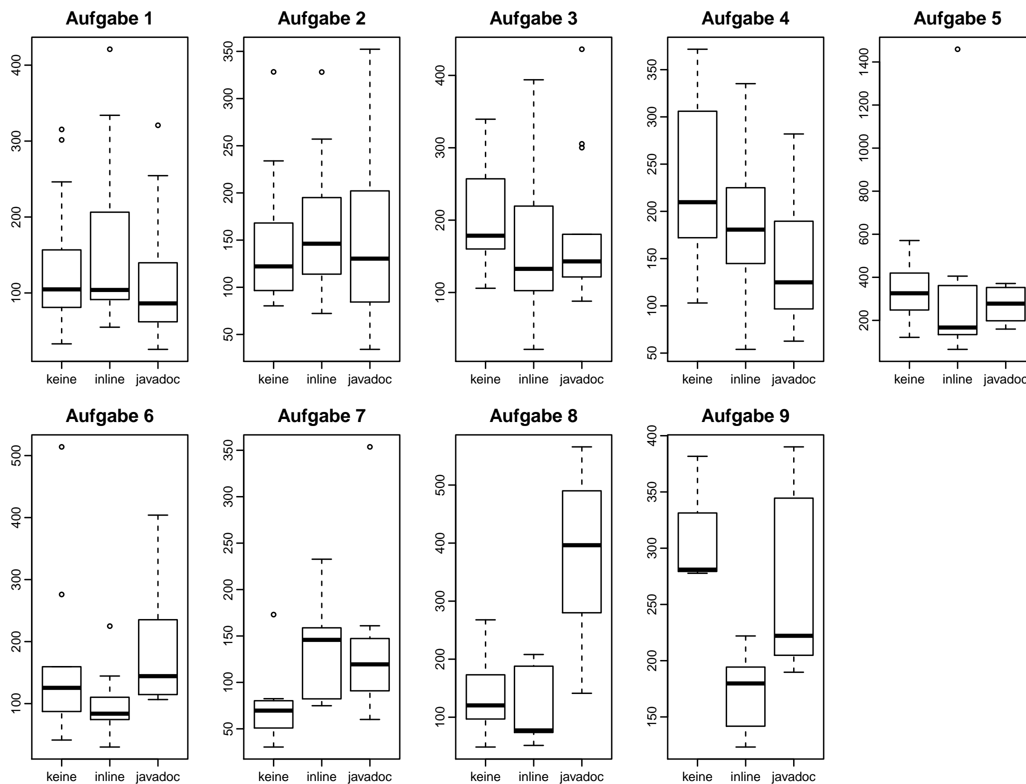


Abbildung 4.7: Bearbeitungszeiten unerfahrener Teilnehmer (in Sekunden)

| Gruppen | Aufgabe 7 | Aufgabe 8 | Aufgabe 9 |
|---|---|---|---|
| A - B | 0.0560 | 1.0000 | **0.0474** |
| A - C | 1.0000 | **0.0161** | **0.0240** |
| B - C | **0.0460** | **0.0059** | 0.7025 |

Tabelle 4.5: $p$-Werte des Dunn-Tests für unerfahrene Teilnehmer

### 4.2.5 Subjektive Einschätzung der Teilnehmer

Die Messung der Bearbeitungszeit ermöglichte eine genaue Analyse der Unterschiede bei der Verwendung von unterschiedlichen Kommentararten. Es ist dennoch interessant, wie der subjektive Eindruck der Entwickler aussieht. Hierzu werden die Antworten des Feedback-Fragebogens ausgewertet (vgl. Kapitel 3.4). Insgesamt 157 Teilnehmer füllten diesen Fragebogen aus.

Abbildung 4.8 zeigt die Häufigkeitsverteilung der ausgewählten Antwortmöglichkeiten für alle Teilnehmer. Es wird deutlich, dass die Probanden den Einfluss von Kommentaren auf die Bearbeitungszeit neutral bis leicht positiv bewerten. Die subjektive Bewertung der Kom-

| | *Schneller* | | *Langsamer* |
|---|---|---|---|
| Aufgabe 7: | Keine | $<$ | Inline |
| Aufgabe 8: | Keine | $<$ | Javadoc |
| | Inline | $<$ | Javadoc |
| Aufgabe 9: | Inline | $<$ | Javadoc |
| | Inline | $<$ | Keine |

Tabelle 4.6: Signifikante Unterschiede in den Bearbeitungszeiten zwischen Kommentararten bei unerfahrenen Teilnehmern

mentare weicht also von den Ergebnissen des Experiments ab. Eine ähnliche Beobachtung machte Takang in seiner Studie [TGM96].

Die Bewertungen der Kommentararten untereinander unterscheiden sich dagegen nur gering. Mehr als die Hälfte der Antworten bestand aus der gleichen Punktevergabe für Inline- und Javadoc-Kommentare. Ein möglicher Grund hierfür ist, dass die Teilnehmer nicht wussten, dass Quellcodekommentare das Thema der Studie waren. Einige Probanden gaben deswegen an, nicht auf die Kommentare geachtet zu haben.

Die Antworten der erfahrenen Teilnehmer zeigen die gleiche Verteilung (Abbildung 4.9). Bei unerfahrenen Teilnehmern wird der Einfluss der Javadoc-Kommentare dagegen neutral bis leicht negativ bewertet.

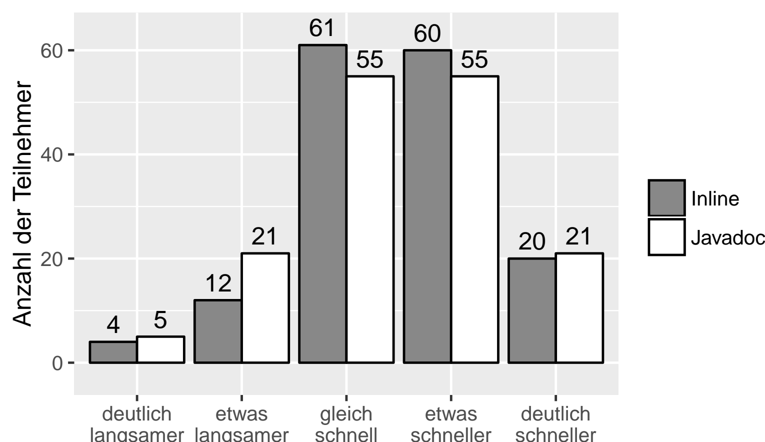


Abbildung 4.8: Einschätzung zur Auswirkung von Kommentaren auf die Bearbeitungszeit (alle Teilnehmer)

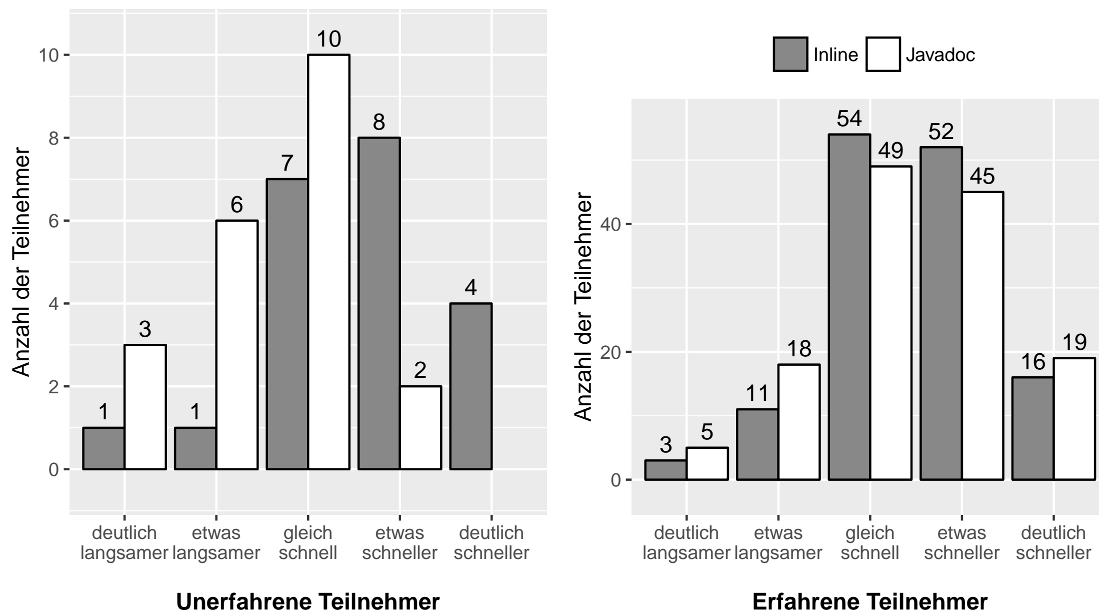


Abbildung 4.9: Einschätzung zur Auswirkung von Kommentaren auf die Bearbeitungszeit (nach Erfahrung)

## 4.3 Prüfung der Korrektheit der Antworten auf signifikante Unterschiede

Ein weiteres Ziel dieser Arbeit ist es, den Einfluss der Kommentararten auf die Korrektheit der Lösungen zu untersuchen. Dieses Kapitel prüft die Richtigkeit der Antworten auf signifikante Unterschiede zwischen den drei Teilnehmergruppen.

Die Korrektheit der Lösungen wurde in einer Entwicklungsumgebung mit vordefinierten Tests überprüft. Im Anhang B sind Musterlösungen aufgeführt, doch jede Lösung, welche die Aufgabenstellung erfüllt, wurde akzeptiert. Abbildung 4.10 fasst die Korrektheit der Lösungen aller Teilnehmer zusammen. Die Frage lautet, ob der Unterschied zwischen den Gruppen bei mindestens einer Aufgabe signifikant ist. Für die Beantwortung dieser Frage wird der Chi-Quadrat-Unabhängigkeitstest sowie der exakte Fisher-Test durchgeführt.

Der Chi-Quadrat-Test ($\chi^2$-Test) auf Unabhängigkeit überprüft, ob die Abhängigkeit zwischen zwei kategorialen[2] Merkmalen $X$ und $Y$ signifikant ist [Koh05]. Die Nullhypothese $H_0$ besagt, dass zwei Merkmale $X$ und $Y$ stochastisch unabhängig sind. Eine Alternative zum $\chi^2$-Test bietet der exakte Test nach Fisher, welcher bei kleinen Stichproben zuverlässigere Ergebnisse liefert [KRES13].

[2]Merkmale mit endlich vielen Ausprägungen, höchstens ordinalskaliert [FKPT07]

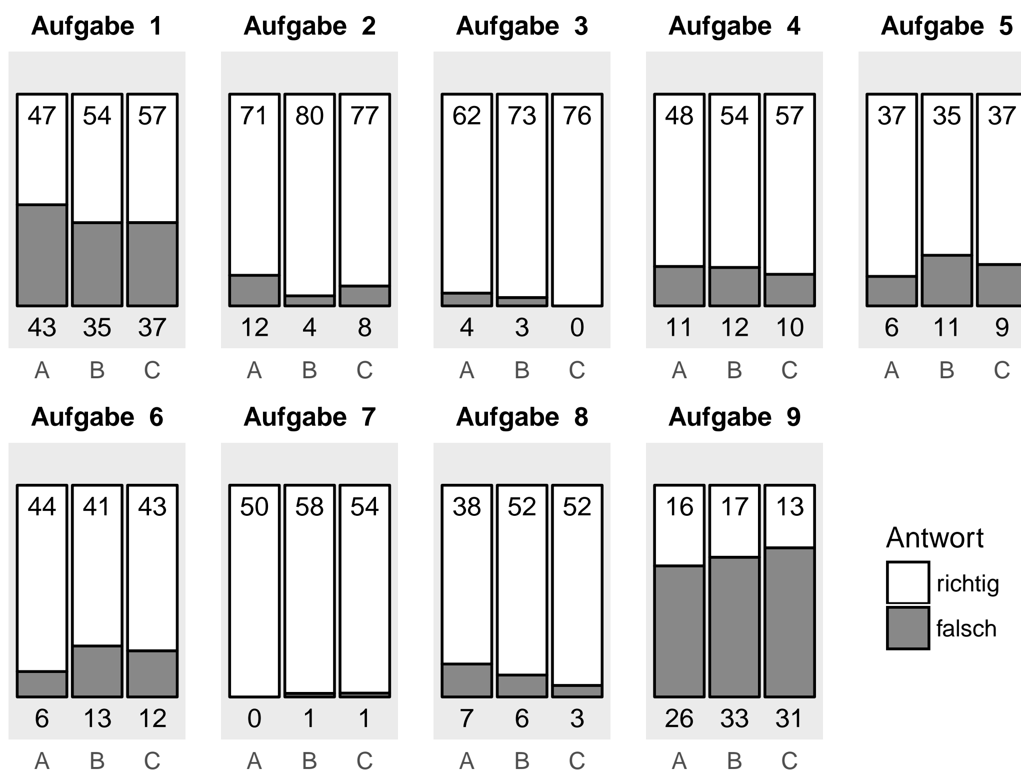


Abbildung 4.10: Anzahl richtiger und falscher Antworten (alle Teilnehmer)

Im Falle der Korrektheit der Antworten wird pro Aufgabe die Abhängigkeit zwischen der Gruppe (Ausprägungen: A, B, C) und der Korrektheit (Ausprägungen: richtig, falsch) überprüft. Sofern der Unabhängigkeitstest für eine Aufgabe einen $p$-Wert kleiner als 0.05 ergibt, wird die Nullhypothese für diese Aufgabe abgelehnt und eine Abhängigkeit zwischen mindestens zwei Gruppen angenommen.

Tabelle 4.7 fasst die $p$-Werte des $\chi^2$- und Fisher-Tests für alle Aufgaben zusammen. Für alle Ergebnisse gilt $p > 0.05$, also wird die Nullhypothese beibehalten. Das heißt, dass die Gruppen stochastisch unabhängig sind. Zwischen den Gruppen bestehen also keine signifikanten Unterschiede in der Korrektheit der Antworten, unabhängig von der Programmiererfahrung.

| | Alle Teilnehmer | | Unerfahren | | Erfahren | |
|---|---|---|---|---|---|---|
| Aufgabe | $\chi^2$-Test | Fisher-Test | $\chi^2$-Test | Fisher-Test | $\chi^2$-Test | Fisher-Test |
| 1 | 0.4148 | 0.4209 | 0.4361 | 0.4894 | 0.3282 | 0.3339 |
| 2 | 0.1025 | 0.1042 | 0.0711 | 0.0540 | 0.4347 | 0.4745 |
| 3 | 0.1064 | 0.0742 | 1.0000 | 1.0000 | 0.2419 | 0.1605 |
| 4 | 0.8292 | 0.8374 | 0.2491 | 0.3225 | 0.8601 | 0.8878 |
| 5 | 0.4912 | 0.5166 | 0.6859 | 0.8375 | 0.5538 | 0.5854 |
| 6 | 0.2592 | 0.2519 | 0.1316 | 0.1823 | 0.6201 | 0.6451 |
| 7 | 1.0000 | 1.0000 | 0.2994 | 0.2917 | 1.0000 | 1.0000 |
| 8 | 0.2674 | 0.2407 | 0.6472 | 0.6443 | 0.0790 | 0.0650 |
| 9 | 0.7037 | 0.7067 | 0.3119 | 0.4863 | 0.6571 | 0.6567 |

Tabelle 4.7: $p$-Werte für die Unabhängigkeitstests

## 4.4 Diskussion der Ergebnisse

Im Folgenden werden die Forschungsfragen beantwortet.

**Forschungsfrage 1**: Welche Art von Kommentaren hilft, Programmieraufgaben schneller zu lösen?

Um diese Frage zu beantworten, wurde der Kruskal-Wallis-Test in Verbindung mit dem Dunn-Test durchgeführt. Die Analyse zeigte, dass bei den Aufgaben 6 und 8 die Gruppe mit Javadoc-Kommentaren signifikant langsamer war als die Gruppe mit Inline-Kommentaren. Ein signifikanter Unterschied zu unkommentierten Varianten konnte bei keiner Aufgabe festgestellt werden. Das bedeutet, dass Kommentare allgemein keinen signifikanten Einfluss auf die Bearbeitungszeit zu haben scheinen. Sofern jedoch Kommentare eingesetzt werden, werden einige Aufgaben mit Inline-Kommentaren schneller gelöst als mit Javadoc-Kommentaren. Eine mögliche Ursache hierfür könnte darin bestehen, dass für die Lösung dieser Aufgaben Implementierungsdetails wichtiger sind als die Beschreibung der Schnittstelle.

Die Einschätzung der Teilnehmer macht deutlich, dass Kommentare subjektiv die Bearbeitungszeiten eher verkürzen. Von einigen wurden die Kommentare dennoch als störend empfunden. Einige Probanden gaben an, dass sie Kommentare absichtlich ignoriert haben oder davon ausgingen, dass sie „lügen“, also den Quellcode nicht wahrheitsgemäß beschreiben. Das bestätigt die Beobachtung von Salviulo, in dessen Studie erfahrene Teilnehmer die Kommentare ignorierten oder nur überflogen [SS14]. Tan et al. erkannten ebenfalls das Problem und entwickelten ein Tool, um automatisch Inkonsistenzen zwischen Kommentaren und Quellcode zu finden [TYKZ07].

**Forschungsfrage 2**: Welche Kommentarart bewirkt, dass Programmieraufgaben häufiger korrekt gelöst werden?

Hierfür wurden die Lösungen zunächst manuell auf Korrektheit überprüft. Anschließend wurden zwei Unabhängigkeitstests durchgeführt, welche jedoch bei keiner Aufgabe einen signifikanten Unterschied feststellen konnten. Das bedeutet, dass die Kommentare keinen Einfluss auf die Korrektheit der Lösungen zu haben scheinen. Die Studie von Woodfield et al. führte zu einer ähnlichen Erkenntnis [WDS81]. Dieses Ergebnis ist nicht überraschend, da für die Aufgabenbearbeitung keine Zeitbegrenzung bestand und die Quellcodestücke eher einfach gehalten sind. Somit war es den Teilnehmern möglich, alle Codebeispiele gleich gut zu verstehen, wenn auch eventuell mit einem unterschiedlichen Zeitaufwand.

**Forschungsfrage 3**: Gibt es einen signifikanten Unterschied zwischen erfahrenen und unerfahrenen Programmierern bei der Schnelligkeit und Richtigkeit der Aufgabenbearbeitung?

Hierfür wurde die Analyse getrennt nach Erfahrungsstand der Teilnehmer wiederholt. Bei der Korrektheit der Antworten wurde auch hier kein signifikanter Unterschied festgestellt. Kommentare scheinen somit keinen Einfluss auf die Richtigkeit der Fragenbeantwortung zu haben, unabhängig von der Programmiererfahrung der Entwickler.

Bei Erfahrenen unterscheiden sich die Bearbeitungszeiten bei keiner Aufgabe signifikant. Für unerfahrene Teilnehmer wurden bei den Aufgaben 7, 8 und 9 jeweils signifikante Unterschiede festgestellt. Bearbeitungszeiten bei Javadoc-Kommentaren sind hierbei größer als bei den entsprechenden Varianten ohne Kommentare bzw. mit Inline-Kommentaren. Dieses Ergebnis ist eher überraschend, da bisherige Studien keine schädlichen Wirkungen der Kommentare feststellten. Es bestätigt außerdem die in Kapitel 1.2 formulierte Vermutung nicht, dass Kommentare unerfahrenen Entwicklern mehr helfen als erfahrenen. Die Unterschiede zwischen keinen und Inline-Kommentaren sind bei der Aufgabe 7 und 9 jedoch abweichend, sodass abschließend kein eindeutig positiver oder negativer Einfluss von Inline-Kommentaren festgestellt wird.

Die subjektive Einschätzung der erfahrenen Teilnehmer deckt sich mit der Einschätzung aus der ersten Forschungsfrage. Unerfahrene Teilnehmer empfinden Javadoc-Kommentare dagegen eher als störend. Mögliche Ursachen hierfür wurden bei der ersten Forschungsfrage genannt.

**Forschungsfrage 4**: Unterscheidet sich der Einfluss der Kommentare auf die Schnelligkeit und Richtigkeit der Aufgabenbearbeitung bei unterschiedlichen Wartungstätigkeiten?

Bei der Untersuchung der Bearbeitungszeit aller Teilnehmer wurden jeweils bei einer von drei Aufgaben aus dem Bereich Fehlerbehebung und Codeerweiterung signifikante Unter-

schiede festgestellt. Bei den unerfahrenen Teilnehmern waren dagegen alle drei Aufgaben zur Codeerweiterung betroffen. Aufgaben zur Codeverwendung enthielten an keiner Stelle einen signifikanten Unterschied.

## 4.5 Gefährdungen der Validität

Im Folgenden werden Faktoren beschrieben, welche die Validität der Ergebnisse gefährden.

Ein Problem bei Online-Umfragen ist die Selbstselektion, d.h. die Teilnehmer entscheiden selbst, ob sie zu der Stichprobe dazugehören. Dies senkt die Repräsentativität der Studie, da die Probanden nicht zufällig vom Experimentleiter aus der Grundgesamtheit, also allen Softwareentwicklern, ausgewählt wurden. Das hat zur Folge, dass eine Verallgemeinerung der Studienergebnisse auf die Grundgesamtheit nicht möglich ist [Gör03].

277 Teilnehmer sind für eine sinnvolle Analyse ausreichend. Die Einteilung nach Gruppe und Erfahrungsstand führt jedoch bei den unerfahrenen Teilnehmern zu recht kleinen Stichproben (4-19 Personen). Dies kann bei den Unerfahrenen zu ungenaueren Ergebnissen führen.

Die Teilnehmer wurden darauf hingewiesen, das Experiment nach Möglichkeit ohne Unterbrechungen durchzuführen. Dennoch sind Ablenkungen oder kleinere Pausen bei einem Online-Experiment nicht auszuschließen. Diese Tatsache kann die Antwortzeiten verfälschen. Extreme Ausreißer wurden daher in der Auswertung nicht berücksichtigt. Außerdem waren im Gegensatz zu einem überwachten Experiment keine Rückfragen möglich. Eventuelle Unklarheiten konnten so ebenfalls die Antwortzeiten beeinflussen.

Die für die Codebeispiele gewählte Programmiersprache Java ist zwar sehr verbreitet, die Ergebnisse lassen sich aber nicht zwingend auf alle Programmiersprachen übertragen. Insbesondere ist die Syntax für Javadoc-Kommentare spezifisch für Java. Das Konzept von Dokumentationskommentaren ist jedoch in vielen anderen Programmiersprachen vertreten.

Die Codebeispiele sind sehr einfach gehalten und auf maximal zwei Klassen begrenzt. Außerdem benötigen die Aufgaben zum Verständnis kein spezifisches Domainwissen. Echte Anwendungen sind deutlich komplexer, sodass Kommentare hier erheblich an Bedeutung gewinnen könnten.

Das Experiment wurde auf Deutsch verfasst. Das ermöglicht, unterschiedliche Fremdsprachenkenntnisse als Störfaktor auszuschließen. Dadurch können jedoch eventuelle Unterschiede zu anderen Ländern und Kulturen nicht festgestellt werden.

# 5 Fazit und Ausblick

Die wissenschaftliche Frage dieser Arbeit lautet: Wie sollte Quellcode kommentiert werden, damit er signifikant schneller und besser verstanden wird als unkommentierter Code?

Um diese Frage zu beantworten, wurde eine Nutzerstudie, sowohl mit Studenten als auch professionellen Entwicklern, durchgeführt. Die Teilnehmer sollten dabei insgesamt neun Programmieraufgaben lösen. Die Bearbeitungszeiten sowie die Korrektheit der Lösungen wurden in Abhängigkeit von den Kommentararten, den Wartungstätigkeiten sowie der Programmiererfahrung der Teilnehmer umfassend untersucht.

In der Korrektheit der Lösungen wurden keine statistisch signifikanten Unterschiede zwischen den Gruppen festgestellt. Das bedeutet, dass es für die Richtigkeit keine Rolle zu spielen scheint, ob bzw. wie ein Quellcode kommentiert ist.

Laut einer subjektiven Einschätzung der Entwickler helfen die Kommentare dabei, die Aufgaben schneller zu lösen. Diese Erkenntnis konnte mit der Auswertung der Bearbeitungszeiten jedoch nicht bestätigt werden. Bei den erfahrenen Entwicklern wurden auch hier keine signifikanten Unterschiede festgestellt. Die Auswertung der unerfahrenen Entwickler zeigt dagegen, dass Kommentare, insbesondere Dokumentationskommentare, teilweise für signifikant langsamere Bearbeitungszeiten sorgen. Dieses Ergebnis widerspricht den bisherigen Studien, welche den Einfluss von Kommentaren als neutral bis positiv bewerteten. Es stimmt jedoch weitgehend mit Praktiken aus der Industrie überein, welche einen sparsamen Umgang mit Kommentaren empfehlen. Alle signifikanten Unterschiede bei den unerfahrenen Teilnehmern wurden bei der Codeerweiterung festgestellt. Dieses Ergebnis legt nahe, dass Kommentararten bei anderen Wartungstätigkeiten weniger wichtig sind.

Das Feedback der Teilnehmer bestätigt die Erkenntnis der Studie von Takang, dass erfahrene Entwickler Kommentaren oft nicht trauen. Dies beruht auf der Tatsache, dass bei einer Codeänderung die Kommentare nicht immer angepasst werden, was zu Widersprüchen zwischen dem Code und dem Kommentar führen kann. Eine mögliche Maßnahme, um diesem Problem entgegenzuwirken, ist die Softwareentwickler dahingehend zu sensibilisieren.

Die Ergebnisse dieser Studie zeigen, dass Kommentare keinen positiven Einfluss auf die Quellcodelesbarkeit zu haben scheinen. Die Antwort auf die wissenschaftliche Frage dieser Arbeit lautet daher, dass auf Kommentare verzichtet werden kann. Diese Erkenntnis basiert jedoch auf einigen genannten Annahmen, welche die Praxisrelevanz dieser Antwort in Frage stellen. Demnach besteht auf diesem Gebiet weiterer Forschungsbedarf. Einige Möglichkeiten für weitere Forschung werden im Folgenden diskutiert.

Um für diese Studie viele Teilnehmer mit einer unterschiedlichen Programmiererfahrung zu gewinnen, wurde das Experiment online durchgeführt. Dabei ist eine Kontrolle der Probanden nicht möglich, was die Ergebnisse verzerrt. Für weitere Studien wird daher ein kontrolliertes Experiment vorgeschlagen, bei dem sich die Teilnehmer zusammen mit den Forschern in einem Raum befinden. Diese Methode ist zwar deutlich zeit- und kostenaufwendiger, ermöglicht jedoch genauere Erkenntnisse.

Die für diese Studie ausgewählten Quellcodebeispiele sind eher kurz, was die Frage aufwirft, ob die Ergebnisse auf komplexere Anwendungen übertragbar sind. Zukünftige Studien sollten daher mehr Praxisnähe anstreben. So bietet sich etwa eine Studie mit Vollzeitprogrammierern an, bei der die Entwickler bei ihrer täglichen Arbeit beobachtet werden.

Diese Arbeit untersucht den Einfluss von Implementations- und Dokumentationskommentaren auf die Quellcodelesbarkeit. Eine weitere Studie könnte den Einfluss von weiteren, in Kapitel 2.3.1.3 vorgestellten Kommentararten untersuchen. So stellt sich beispielsweise die Frage, ob die relative Position eines Kommentars einen Einfluss auf die Codelesbarkeit hat.

Schließlich sollten weitere Forschungen zu den Widersprüchen zwischen Code und Kommentaren vorangetrieben werden, um hierfür zuverlässige Lösungen zu finden.

# A Tabellen

### Varianzanalyse der Bearbeitungszeiten

| Aufgabe | Alle Teilnehmer | Erfahrene Teilnehmer |
|---|---|---|
| 1 | 0.9182 | 0.6315 |
| 2 | 0.3035 | 0.3598 |
| 3 | 0.2321* | 0.1834* |
| 4 | 0.5704 | 0.9159 |
| 5 | 0.2024 | 0.1158 |
| 6 | **0.0306** | 0.0878 |
| 7 | 0.1172 | 0.1782 |
| 8 | **0.0061*** | 0.2035 |
| 9 | 0.1082* | 0.0545* |

Tabelle A.1: $p$-Werte für die Varianzanalyse der Bearbeitungszeiten

* Varianzhomogenität nicht gegeben, siehe Kapitel 4.2.3

# B Experimentaufgaben

## B.1 Aufgabe 1

Rufe die Methode *foo()* so auf, dass eine 7 zurückgegeben wird.

```
/**
 * Summiert die Werte der übergebenen Strings auf, wobei den Strings
 * die folgenden Zahlenwerte zugeordnet werden:
 *   "i" -> 1
 *   "v" -> 5
 *   "x" -> 10
 *
 * @param strings  ein String-Array
 * @return         Summe der Stringwerte
 */
public int foo(String[] strings) {
  int number = 0;

  for (int i = 0; i < strings.length; i++) {
    if (strings[i].equals("i")) {
      number = number + 1;
    } else if (strings[i].equals("v")) {
      number = number + 5;
    } else if (strings[i].equals("x")) {
      number = number + 10;
    }
  }
  return number;
}
```

Quelltext B.1: Aufgabe 1 (Javadoc)

```
public int foo(String[] strings) {
  int number = 0; // Summe der Stringwerte

  for (int i = 0; i < strings.length; i++) {
    if (strings[i].equals("i")) {
      number = number + 1;         // "i" = 1
    } else if (strings[i].equals("v")) {
      number = number + 5;         // "v" = 5
    } else if (strings[i].equals("x")) {
      number = number + 10;        // "x" = 10
    }
  }
  return number;
}
```

Quelltext B.2: Aufgabe 1 (Inline)

```
foo(new String[]{"v", "i", "i"});
```

Quelltext B.3: Aufgabe 1 - Musterlösung

## B.2 Aufgabe 2

Rufe die Methode *foo()* so auf, dass ”doremi” zurückgegeben wird.

```
/**
 * Liefert ein Suffix ab der Stelle number zurück. Falls bool true ist,
 * wird das entfernte Präfix am Ende angehängt.
 *
 * @param string1  ein String, darf nicht null sein
 * @param number   Startindex
 * @param bool     true, falls das entfernte Präfix des Strings am Ende
 *                 angehängt werden soll
 * @return         Suffix; gefolgt vom Präfix, falls bool true ist
 */
private String foo(String string1, int number, boolean bool) {

  String string2 = "";

  for (int i = number; i < string1.length(); i++) {
    string2 = string2 + string1.charAt(i);
  }

  if (bool) {
    for (int j = 0; j < number; j++) {
      string2 = string2 + string1.charAt(j);
    }
  }

  return string2;
}
```

Quelltext B.4: Aufgabe 2 (Javadoc)

```
private String foo(String string1, int number, boolean bool) {

  String string2 = "";

  for (int i = number; i < string1.length(); i++) {
    // Suffix ab der Stelle number
    string2 = string2 + string1.charAt(i);
  }

  if (bool) {
    // falls bool true ist: das entfernte Präfix des Strings anhängen
    for (int j = 0; j < number; j++) {
      string2 = string2 + string1.charAt(j);
    }
  }

  return string2;
}
```

Quelltext B.5: Aufgabe 2 (Inline)

```
foo("midore", 2, true);
foo("doremi", 0, false);
```

Quelltext B.6: Aufgabe 2 - Musterlösungen

## B.3 Aufgabe 3

Verändere (nur) die Liste *objectList* in der Methode *bar()* so, dass der Aufruf dieser Methode ”Amy” ausgibt.

```
public void bar() {
  List<Class1> objectList = new ArrayList<>();
  objectList.add(new Class1("Steve", 25));
  objectList.add(new Class1("John", 42));
  objectList.add(new Class1("Claudia", 19));

  Class1 object = foo(objectList);
  System.out.println(object.string);
}
```

```
/**
 * Enthält den Namen und das Alter einer Person.
 */
class Class1 {

  /** Name */
  String string;

  /** Alter */
  int number;

  public Class1(String string, int number) {
    this.string = string;
    this.number = number;
  }
}

/**
 * Gibt die älteste Person zurück.
 *
 * @param objectList  Liste von Personen, muss mindestens eine Person enthalten
 * @return            die älteste Person
 */
public Class1 foo(List<Class1> objectList) {
  Class1 object = objectList.get(0);

  for (int i = 1; i < objectList.size(); i++) {
    if (objectList.get(i).number > object.number) {
      object = objectList.get(i);
    }
  }
  return object;
}
```

Quelltext B.7: Aufgabe 3 (Javadoc)

```
class Class1 {

  String string; // Name
  int number; // Alter

  public Class1(String string, int number) {
    this.string = string;
    this.number = number;
  }
}

public Class1 foo(List<Class1> objectList) {
  Class1 object = objectList.get(0); // erste Person ist bisher auch die älteste

  for (int i = 1; i < objectList.size(); i++) {
    if (objectList.get(i).number > object.number) {
      // Alter vergleichen und die ältere Person zwischenspeichern
      object = objectList.get(i);
    }
  }
  return object;
}
```

Quelltext B.8: Aufgabe 3 (Inline)

```
objectList.add(new Person("Amy", 50));
```

Quelltext B.9: Aufgabe 3 - Musterlösung

## B.4 Aufgabe 4

Die Methode *foo()* wirft für diese Eingabe eine Laufzeitexception. Behebe den Fehler, sodass das erwartete Ergebnis [3, 8] zurückgegeben wird:

```
foo(new int[]{1, 3, 4, 5, 8, 11, 13},
    new int[]{2, 3, 5, 7, 8, 9});
  >> [3,8]
```

```java
/**
 * Gibt eine Liste von Zahlen zurück, die an den gleichen Stellen in den
 * übergebenen Arrays gleich sind.
 *
 * @param numbers1  erstes Array
 * @param numbers2  zweites Array
 * @return          Liste von gleichen Zahlen
 */
public List<Integer> foo(int[] numbers1, int[] numbers2) {

  List<Integer> numberList = new ArrayList<>();
  int max = numbers1.length;

  for (int i = 0; i < max; i++) {
    if (numbers1[i] == numbers2[i]) {
      numberList.add(numbers1[i]);
    }
  }
  return numberList;
}
```

Quelltext B.10: Aufgabe 4 (Javadoc)

```java
public List<Integer> foo(int[] numbers1, int[] numbers2) {

  List<Integer> numberList = new ArrayList<>(); // Ergebnisliste
  int max = numbers1.length; // Anzahl der zu durchlaufenden Elemente

  for (int i = 0; i < max; i++) {
    if (numbers1[i] == numbers2[i]) {
      // wenn der i-te Wert in beiden Arrays gleich ist, zur Ergebnisliste hinzufügen
      numberList.add(numbers1[i]);
    }
  }
  return numberList;
}
```

Quelltext B.11: Aufgabe 4 (Inline)

```java
  ...

  int max = Math.min(array1.length, array2.length);

  ...
```

Quelltext B.12: Aufgabe 4 - Musterlösung

## B.5 Aufgabe 5

Die Methode *foo()* enhält einen Fehler. Behebe diesen so, dass die erwarteten Ergebnisse zurückgegeben werden:

```
foo("abcd", "acbd")
  >> 1
foo("abcd", "badc")
  >> 2
foo("abcdef", "defabc")
  >> 3
```

Es wird vorausgesetzt, dass beide Zeichenketten gleich lang und ungleich null sind.

```
/**
 * Gibt die Anzahl der Transpositionen zwischen zwei Zeichenketten zurück.
 *
 * Eine Transposition ist die Vertauschung zweier Zeichen eines Strings.
 *
 * @param string1  erster String, ungleich null
 * @param string2  zweiter String, ungleich null
 * @return         Anzahl der Transpositionen zwischen zwei Strings
 */
public int foo(final String string1, final String string2) {
  int number = 0;
  for (int i = 0; i < string1.length(); i++) {
    if (string1.charAt(i) != string2.charAt(i)) {
      number++;
    }
  }
  return number;
}
```

Quelltext B.13: Aufgabe 5 (Javadoc)

```
public int foo(final String string1, final String string2) {
  int number = 0; // Anzahl unterschiedlicher Zeichen zwischenspeichern
  for (int i = 0; i < string1.length(); i++) {
    if (string1.charAt(i) != string2.charAt(i)) {
      // wenn sich zwei Zeichen unterscheiden, number erhöhen
      number++;
    }
  }
  return number; // Anzahl der Transpositionen zurückgeben
}
```

Quelltext B.14: Aufgabe 5 (Inline)

```
  ...
  return number / 2; // Anzahl der Transpositionen zurückgeben
}
```

Quelltext B.15: Aufgabe 5 - Musterlösung

## B.6 Aufgabe 6

Behebe den Compilefehler in der Methode *foo()*.

```
/**
 * Gibt ein Suffix ab dem ersten Auftreten des Zeichens ch zurück.
 *
 * @param string  Zeichenkette
 * @param ch      Zeichen zum Ermitteln des Suffixes
 * @return        Suffix ab dem ersten Zeichen ch,
 *                Leerstring, wenn das Zeichen im String nicht vorkommt
 */
public String foo(String string, char ch) {
  for (int i = 0; i < string.length(); i++) {
    if (string.charAt(i) == ch) {
      return string.substring(i + 1);
    }
  }
}
```

Quelltext B.16: Aufgabe 6 (Javadoc)

```
public String foo(String string, char ch) {
  for (int i = 0; i < string.length(); i++) {
    if (string.charAt(i) == ch) {
      // Suffix ab dem ersten Auftreten des Zeichens ch zurückgeben
      return string.substring(i + 1);
    }
  }
}
```

Quelltext B.17: Aufgabe 6 (Inline)

```
  ...
  return ""; // gibt einen leeren String zurück, wenn das Zeichen im String nicht vorkommt
}
```

Quelltext B.18: Aufgabe 6 - Musterlösung

## B.7 Aufgabe 7

Füge der Methode *foo()* einen int-Parameter hinzu, der zurückgegeben wird, falls *number* vor der Rückgabe gleich 0 ist. Beispiel:

```
foo(new int[]{5, 13, 31}, 7);
  >> 7
```

```java
/**
 * Summiert alle geraden Zahlen im Array auf.
 *
 * @param numbers  int-Array
 * @return return  Summe aller geraden Zahlen im Array
 */
public int foo(int[] numbers) {

  int number = 0;

  for (int i = 0; i < numbers.length; i++) {
    if (numbers[i] % 2 == 0) {
      number = number + numbers[i];
    }
  }

  return number;
}
```

Quelltext B.19: Aufgabe 7 (Javadoc)

```java
public int foo(int[] numbers) {

  int number = 0; // ergebnis zwischenspeichern

  for (int i = 0; i < numbers.length; i++) {
    if (numbers[i] % 2 == 0) {
      // wenn die i-te Zahl gerade ist, wird sie zum ergebnis addiert
      number = number + numbers[i];
    }
  }

  return number;
}
```

Quelltext B.20: Aufgabe 7 (Inline)

```java
public int foo(int[] numbers, int fallback) {

  ...

  if (result == 0)
    return fallback;

  return result;
}
```

Quelltext B.21: Aufgabe 7 - Musterlösung

# B.8 Aufgabe 8

Ändere die Methode *foo()* so, dass null-Strings bei der Ausgabe ignoriert werden. Beispiel:

```
foo(new String[]{"Tic", null, "Tac", "Toe"}, ',')
  >> Tic,Tac,Toe
```

```
/**
 * Verkettet alle Elemente eines String-Arrays zu einem String, getrennt
 * durch das übergebene Trennzeichen.
 *
 * @param strings  ein String-Array
 * @param ch       Trennzeichen
 * @return         Verkettung der Array-Elemente
 * @see            java.lang.StringBuilder
 */
public static String foo(final String[] strings, char ch) {

  final StringBuilder builder = new StringBuilder();

  for (int i = 0; i < strings.length; i++) {

    if (i > 0) {
      builder.append(ch);
    }

    builder.append(strings[i]);
  }
  return builder.toString();
}
```

Quelltext B.22: Aufgabe 8 (Javadoc)

```
public static String foo(final String[] strings, char ch) {

  // Endstring mit einem StringBuilder zusammenbauen
  final StringBuilder builder = new StringBuilder();

  for (int i = 0; i < strings.length; i++) {

    if (i > 0) {
      // vor jedem String das Trennzeichen einfügen - außer vor dem ersten
      builder.append(ch);
    }

    builder.append(strings[i]); // i-ten String zum Endstring hinzufügen
  }
  return builder.toString();
}
```

Quelltext B.23: Aufgabe 8 (Inline)

```
  ...

  for (int i = 0; i < strings.length; i++) {
    if (strings[i] == null)
      continue;

  ...
```

Quelltext B.24: Aufgabe 8 - Musterlösung

# B.9 Aufgabe 9

Erweitere die Klasse *Class2* um eine Methode *bar()* mit einem Rückgabewert vom Typ Integer, die die Operation von *foo()* rückgängig macht.

```
/**
 * Datenstruktur zum Speichern eines Verlaufs von Zahlen.
 */
class Class2 {

  LinkedList<Integer> numberList = new LinkedList<>();
  int number1 = 0;

  /**
   * Fügt eine Zahl an der aktuellen Position im Verlauf hinzu. Alle
   * eventuell nachfolgenden Zahlen werden gelöscht.
   *
   * @param number2  Zahl, die hinzugefügt werden soll
   */
  public void qux(Integer number2) {
    while (number1 < numberList.size()) {
      numberList.removeLast();
    }

    numberList.add(number2);
    number1++;
  }

  /**
   * Nimmt die letzte Zahl im Verlauf zurück und liefert die vorangehende Zahl.
   * Falls keine Zahl im Verlauf vorhanden ist, wird null zurückgegeben.
   *
   * @return  die im Verlauf vorangehende Zahl
   */
  public Integer foo() {
    if (number1 > 0) {
      number1--;
      return numberList.get(number1);
    } else {
      return null;
    }
  }
}
```

Quelltext B.25: Aufgabe 9 (Javadoc)

```
class Class2 {

  LinkedList<Integer> numberList = new LinkedList<>(); // Zahlen in einer Liste speichern
  int number1 = 0; // Zeiger auf die aktuelle Position in der Liste

  public void qux(Integer number2) {
    while (number1 < numberList.size()) {
      // alle Zahlen hinter dem Zeiger werden gelöscht
      numberList.removeLast();
    }

    // die neue Zahl wird an die letzte Stelle hinzugefügt und der Zeiger erhöht
    numberList.add(number2);
    number1++;
  }

  public Integer foo() {
    if (number1 > 0) {
      // Zeiger befindet sich nicht am Anfang der Liste - Zeiger um
      // 1 vermindern und aktuellen Wert zurückgeben
      number1--;
      return numberList.get(number1);
    } else {
      return null; // Zeiger am Anfang der Liste - null zurückgeben
    }
  }
}
```

Quelltext B.26: Aufgabe 9 (Inline)

```
public Integer bar() {
  if (number1 < numberList.size()) {
    number1++;
    return numberList.get(number1);
  } else {
    return null;
  }
}
```

Quelltext B.27: Aufgabe 9 - Musterlösung

# Literaturverzeichnis


[AD52] Theodore W. Anderson und Donald A. Darling. Asymptotic Theory of Certain 'Goodness of Fit' Criteria Based on Stochastic Processes. *Ann. Math. Statist.*, 23(2): S. 193–212, 1952.

[ASC02] Krishan K. Aggarwal, Yogesh Singh, und Jitender K. Chhabra. An integrated measure of software maintainability. *Reliability and Maintainability Symposium, 2002. Proceedings. Annual*, S. 235–241, 2002.

[BF14] Pierre Bourque und Richard E. Fairley. *Guide to the Software Engineering Body of Knowledge, Version 3.0.* IEEE, 2014.

[BLMM09] Dave Binkley, Dawn Lawrie, Steve Maex, und Christopher Morrell. Identifier length and limited programmer memory. *Science of Computer Programming*, 74(7): S. 430–445, 2009.

[BSW11] Raymond P.L. Buse, Caitlin Sadowski, und Westley Weimer. Benefits and barriers of user evaluation in software engineering research. *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications - OOPSLA '11*, S. 643, 2011.

[BW08] Raymond P.L. Buse und Westley R. Weimer. A metric for software readability. *Proceedings of the 2008 international symposium on Software testing and analysis - ISSTA '08*, S. 121–130, 2008.

[BWYS11] Simon Butler, Michel Wermelinger, Yijun Yu, und Helen Sharp. Exploring the influence of identifier names on code quality: An empirical study. *Proceedings of the European Conference on Software Maintenance and Reengineering, CSMR*, S. 156–165, 2011.

[CV06] Emilio Collar und Ricardo Valerdi. Role of Software Readability on Software Development Cost. *In Proceedings of the 21st Forum on COCOMO and Software Cost Modeling*, 2006.

[Din15] Alexis Dinno. Nonparametric pairwise multiple comparisons in independent groups using Dunn's test. *Stata Journal*, 15: S. 292–300, 2015.

[Dor12] Jonathan Dorn. A General Software Readability Model. *University of Virginia, Charlottesville, Virginia*, S. 1–62, 2012.

[DP06] Florian Deissenboeck und Markus Pizka. Concise and consistent naming. *Software Quality Journal*, 14(3): S. 261–282, 2006.

[EC95] K. Anders Ericsson und Neil Charness. Expert performance: Its structure and acquisition. *American Psychologist*, 50(9): S. 803–804, 1995.

[Eck14] Peter Eckstein. *Repetitorium Statistik Deskriptive Statistik - Stochastik - Induktive Statistik*. Springer Gabler, 8. Aufl., 2014.

[EM82] James L. Elshoff und Michael Marcotty. Improving computer program readability to aid modification. *Communications of the ACM*, 25(8): S. 512–521, 1982.

[FKL+12] Janet Feigenspan, Christian Kastner, Jörg Liebig, Sven Apel, und Stefan Hanenberg. Measuring Programming Experience. *IEEE 20th International Conference on Program Comprehension (ICPC), 2012*, 2005(of 2161): S. 73–82, 2012.

[FKPT07] Ludwig Fahrmeir, Rita Künstler, Iris Pigeot, und Gerhard Tutz. *Statistik : der Weg zur Datenanalyse*. Springer, 6. Aufl., 2007.

[Fle48] Rudolf Flesch. A New Readability Yardstick. *The Journal of Applied Psychology*, 32(3): S. 221–233, 1948.

[Fos93a] John Foster. An Industry View On Program Comprehension. In *IEEE Proceedings, 2nd Workshop on Program Comprehension*, S. 107. IEEE, 1993.

[Fos93b] John Foster. *Cost factors in software maintenance*. PhD thesis, Durham University, 1993.

[Fow99] Martin Fowler. *Refactoring : Improving the design of existing code*. Addison-Wesley, 1999.

[FR07] Frederik Funke und Ulf-Dietrich Reips. Datenerhebung im Netz: Messmethoden und Skalen. *Online-Forschung 2007: Grundlagen und Fallstudien*, S. 51–76, 2007.

[FWG07] Beat Fluri, Michael Würsch, und Harald C. Gall. Do code and comments co-evolve? On the relation between source code and comment changes. *Proceedings - Working Conference on Reverse Engineering, WCRE*, S. 70–79, 2007.

[Gör03] Anja S. Göritz. Online-Panels. In *Online-Marktforschung*, S. 227–240. Springer, 2003.

[Gun52] Robert Gunning. *The Technique of Clear Writing*. McGraw-Hill, 1952.

[HS15] Jürgen Hedderich und Lothar Sachs. *Angewandte Statistik: Methodensammlung mit R*. Springer Spektrum, 15. Aufl., 2015.

[HT99] Andrew Hunt und David Thomas. *The Pragmatic Programmer: From Journeyman to Master*. Addison-Wesley Longman Publishing Co., Inc., 1999.

[IEE98] IEEE Standards Board. *IEEE Standard for Software Maintenance*. IEEE, 1998.

[Ins06] Institute of Electrical and Electronics Engineers. *ISO 14764: Software Engineering - Software Life - Cycle Processes - Maintenance*. IEEE, 2006.

[JD06] Bortz Jürgen und Nicola Döring. *Forschungsmethoden und Evaluation für Human- und Sozialwissenschaftler*. Springer, 2006.

[JH06] Zhen Ming Jiang und Ahmed E. Hassan. Examining the evolution of code comments in PostgreSQL. *Proceedings of the 2006 international workshop on Mining software repositories - MSR '06*, S. 179, 2006.

[JHDE11] Rüdiger Jacob, Andreas Heinz, Jean Philippe Décieux, und Willy Eirmbter. *Umfrage: Einführung in die Methoden der Umfrageforschung*. Oldenbourg, 2011.

[KA14] Martin Krzywinski und Naomi Altman. Points of Significance: Visualizing samples with box plots. *Nature Methods*, 11(2): S. 119–120, 2014.

[Kae88] Michael J. Kaelbling. Programming Languages Should NOT Have Comment Statements. *SIGPLAN Not.*, 23(10): S. 59–60, 1988.

[Knu92] Donald E. Knuth. Literate programming. *CSLI Lecture Notes, Stanford, CA: Center for the Study of Language and Information (CSLI), 1992*, 1(1), 1992.

[Koh05] Wolfgang Kohn. *Statistik: Datenanalyse und Wahrscheinlichkeitsrechnung*. Springer-Verlag Berlin Heidelberg, 2005.

[KP99] Brian Kernighan und Rob Pike. *The Practice of Programming*. Addison-Wesley, 1999.

[Kra99] Douglas Kramer. API documentation from source code comments: a case study of Javadoc. *Proceedings of the 17th annual international conference on Computer documentation*, S. 147–153, 1999.

[KRES13] Udo Kuckartz, Stefan Rädiker, Thomas Ebert, und Julia Schehl. *Statistik: Eine verständliche Einführung*. VS Verlag für Sozialwissenschaften, 2. Aufl., 2013.

[KW52] William H Kruskal und W Allen Wallis. Use of Ranks in One-Criterion Variance Analysis. *Journal of the American Statistical Association*, 47(260): S. 583–621, 1952.

[KWR10] Ninus Khamis, René Witte, und Juergen Rilling. Automatic Quality Assessment of Source Code Comments: The JavadocMiner. In *Natural Language Processing and Information Systems*, S. 68–79. Springer, 2010.

[LBS06] Ben Liblit, Andrew Begel, und Eve Sweeser. Cognitive Perspectives on the Role of Naming in Computer Programs. *Proceedings of the 18th Annual Psychology of Programming Interest Group Workshop*, S. 53–67, 2006.

[Lev60] Howard Levene. Robust tests for equality of variances. *Contributions to probability and statistics: Essays in honor of Harold Hotelling*, 2: S. 278–292, 1960.

[Lil67] Hubert W. Lilliefors. On the Kolmogorov-Smirnov Test for Normality with Mean and Variance Unknown. *Journal of the American Statistical Association*, 62(318): S. 399–402, 1967.

[Mar08] Robert C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall PTR, 2008.

[Mit14] Hans-Joachim Mittag. *Statistik: Eine Einführung mit interaktiven Elementen*. Springer Spektrum, 3. Aufl., 2014.

[OF95] Brid O'Conaill und David Frohlich. Timespace in the Workplace: Dealing with Interruptions. In *Conference Companion on Human Factors in Computing Systems*, CHI '95, S. 262–263. ACM, 1995.

[Ora12] Oracle Corporation. How to Write Doc Comments for the Javadoc Tool, 2012.

[Par94] David L. Parnas. Software aging. *16th International Conference on Software Engineering*, S. 279–287, 1994.

[PHD11] Daryl Posnett, Abram Hindle, und Prem Devanbu. A Simpler Model of Software Readability Categories and Subject Descriptors. *Proceedings of the 8th Working Conference on Mining Software Repositories*, S. 73–82, 2011.

[Pre10] Roger Pressman. *Software engineering: a practitioner's approach*. McGraw-Hill Higher Education, 7. Aufl., 2010.

[PS10] Manuela Pötschke und Julia Simonson. Online-Erhebungen in der empirischen Sozialforschung : Erfahrungen mit einer Umfrage unter Sozial-, Markt- und Meinungsforschern. *ZA-Information / Zentralarchiv für Empirische Sozialforschung (2001)*, 49: S. 6–27, 2010.

[Ras06] Björn Rasch. *Quantitative Methoden 1 : Einführung in die Statistik*. Springer Medizin Verlag Heidelberg, 2. Aufl., 2006.

[Ras10] Björn Rasch. *Quantitative Methoden 2. Einführung in die Statistik für Psychologen und Sozialwissenschaftler*. Springer-Verlag Berlin Heidelberg, Berlin, Heidelberg, 2010.

[Red00] Achut Reddy. Java Coding Style Guide, 2000.

[Rei05] Phillip A. Reif. Tool assisted identifier naming for improved software readability: An empirical study. *2005 International Symposium on Empirical Software Engineering, ISESE 2005*, S. 53–62, 2005.

[Rob89] Pierre-N. Robillard. Automating comments. *ACM SIGPLAN Notices*, 24(5): S. 66–70, 1989.

[SBCL78] S. B. Sheppard, M. A. Borst, B. Curtis, und L. T. Love. Predicting Programmers' Ability to Modify Software. Technical report, General Electric Co Arlington VA, 1978.

[Sed16] Todd Sedano. Code Readability Testing, an Empirical Study. *2016 IEEE 29th International Conference on Software Engineering Education and Training (CSEET)*, S. 111–117, 2016.

[SGR+15] Giuseppe Scanniello, Carmine Gravino, Michele Risi, Genoveffa Tortora, und Gabriella Dodero. Documenting Design-Pattern Instances. *ACM Transactions on Software Engineering and Methodology*, 24(3): S. 1–35, 2015.

[Sie12] Janet Siegmund. *Framework for Measuring Program Comprehension*. PhD thesis, Otto-von-Guericke-Universität Magdeburg, 2012.

[SJ13] Daniela Steidl und Elmar Juergens. Quality Analysis of Source Code Comments. *Icpc*, S. 83–92, 2013.

[SL15] Philipp Sibbertsen und Hartmut Lehne. *Statistik Einführung für Wirtschafts- und Sozialwissenschaftler*. Gabler, 2. Aufl., 2015.

[SRAM06] Susan Sim, Sukanaya Ratanotayanon, Oluwatosin Aiyelokun, und Erin Morris. An initial study to develop an empirical test for software engineering expertise. Technical report, Institute for Software Research, University of California, 2006.

[SS14] Felice Salviulo und Giuseppe Scanniello. Dealing with identifiers and comments in source code comprehension and maintenance. *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering - EASE '14*, S. 1–10, 2014.

[Sun97] Sun Microsystems Inc. Java Code Conventions, 1997.

[SW65] Samuel S. Shapiro und Martin B. Wilk. An Analysis of Variance Test for Normality. *Biometrika*, 52(3/4): S. 591–611, 1965.

[SZCF08] Peter Sommerlad, Guido Zgraggen, Thomas Corbat, und Lukas Felber. Retaining Comments when Refactoring Code. *Companion to the 23rd ACM SIGPLAN Conference on Object-oriented Programming Systems Languages and Applications*, S. 653–662, 2008.

[Ten88] Ted Tenny. Program readability: Procedures versus comments. *IEEE Transactions on Software Engineering*, 14(9): S. 1271–1279, 1988.

[TGM96] Armstrong A. Takang, Penny A. Grubb, und Robert D. Macredie. The effects of comments and identifier names on program comprehensibility: An experimental investigation. *Journal of Programming Languages*, 4(3): S. 143–167, 1996.

[TOAY13] Yahya Tashtoush, Zeinab Odat, Izzat Alsmadi, und Maryan Yatim. Impact of programming features on code readability. *International Journal of Software Engineering and its Applications*, 7(6): S. 441–458, 2013.

[TYKZ07] Lin Tan, Ding Yuan, Gopal Krishna, und Yuanyuan Zhou. /*iComment: Bugs or Bad Comments?*/. *Proc. SOSP*, S. 145–158, 2007.

[Ver00] Al Vermeulen. The Elements of Java (TM) Style, 2000.

[WDS81] S. N. Woodfield, H. E. Dunsmore, und V. Y. Shen. The effect of modularization and comments on program comprehension. In *ICSE '81 Proceedings of the 5th international conference on Software engineering*, S. 215–223, 1981.

[WPVS14] Xiaoran Wang, Lori Pollock, und K. Vijay-Shanker. Automatic Segmentation of Method Code into Meaningful Blocks: Design and Evaluation. *Journal of Software: Evolution and Process*, 26(1): S. 27–49, 2014.

[YWA05] Annie T. T. Ying, James L. Wright, und Steven Abrams. Source code that talks: an exploration of Eclipse task comments and their implication to repository mining. *Proc. MSR*, S. 1–5, 2005.

# Selbständigkeitserklärung

Hiermit versichere ich, die vorliegende Arbeit selbständig und ohne fremde Hilfe angefertigt zu haben. Die verwendete Literatur und sonstige Hilfsmittel sind vollständig angegeben.

Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht.

Magdeburg, 19. August 2016 ................................................

Dariusz Krolikowski