Freie Universität Berlin

Bachelorarbeit am Institut für Informatik der Freien Universität Berlin

Arbeitsgruppe Software Engineering

Qualitative Analyse öffentlicher Refactoringdiskussionen in Github Issue Trackern

Jeannine Darakci
Matrikelnummer: 4917809
jeannine.darakci@inf.fu-berlin.de

Betreuer/in: Prof. Dr. Lutz Prechelt
Eingereicht bei: Prof. Dr. Lutz Prechelt
Zweitgutachter/in: Prof. Dr. Claudia Müller-Birn

Berlin, 10. Februar 2022

Zusammenfassung

Die folgende Bachelorarbeit beschäftigt sich mit der Frage, welche Elemente des mentalen Modells von Entwickler:innen ihre Bereitschaft für Refactoring beeinflussen. Was muss gelten, damit Menschen bereit sind Refactoring zu betreiben und was hindert sie aktuell daran? Bei dem Material handelt es sich um Issues, die mittels Github Issue Tracker beschafft werden. Zur Analyse wird eine qualitative Datenanalyse anhand der Grounded Theory Methodology durchgeführt. Als Ergebnis ergibt sich, dass die Bereitschaft Refactoring zu betreiben von mehreren Faktoren abhängt. Zum einen von der Frage nach dem Aufwand und wie dieser abgeschätzt wird. Es entsteht hierbei der Eindruck, dass der Aufwand nicht konkret abgeschätzt und somit nicht abgeglichen wird. Zum anderen von Kriterien, wie Ausgangssituation und den zeitlichen Möglichkeiten, wie beispielsweise Deadlines. Ein weiterer entscheidender Faktor ist das Kosten-Nutzen-Verhältnis, welches mehrheitlich gesehen negativ wahrgenommen wird. Als Lösung könnte man die Aufwandsschätzung als festen Bestandteil einer solchen Diskussion machen und die Relevanz des Refactorings durch beispielsweise Vorträge hervorheben.

Eidesstattliche Erklärung

Ich versichere hiermit an Eides Statt, dass diese Arbeit von niemand anderem als meiner Person verfasst worden ist. Alle verwendeten Hilfsmittel wie Berichte, Bücher, Internetseiten oder ähnliches sind im Literaturverzeichnis angegeben, Zitate aus fremden Arbeiten sind als solche kenntlich gemacht. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungskommission vorgelegt und auch nicht veröffentlicht.

10. Februar 2022

Jeannine Darakci

Inhaltsverzeichnis

1 Einleitung **9**
1.1 Forschungsfrage . . . 9
1.2 Relevanz der Forschungsfrage . . . 9
1.3 Motivation der Arbeit . . . 9
1.4 Vergleichbare Arbeiten . . . 9
1.4.1 Aufbau der Arbeit . . . 10

2 Einführung in die Grounded Theory **11**
2.1 Kodierung . . . 11
2.1.1 Offenes Kodieren . . . 11
2.1.2 Axiales Kodieren . . . 12
2.1.3 Selektives Kodieren . . . 12
2.1.4 Memos . . . 12
2.1.5 Theoretische Sättigung . . . 13
2.2 Technische Schulden . . . 13
2.3 Refactoring . . . 14

3 Methodik **15**
3.1 Datensammlung . . . 15
3.1.1 GitHub . . . 15
3.1.2 QIF-Tool . . . 15
3.1.3 MAXQDA . . . 15
3.2 Analyse . . . 15
3.2.1 Datensammlung . . . 15
3.2.2 Datenauswertung . . . 16
3.2.3 Theoriekonstruktion . . . 20
3.2.4 Verworfene Issues . . . 21

4 Ergebnisse **23**
4.1 Erkenntnis 1: Die Aufwandseinschätzung ist relevant für die Bereitschaft Refactoring zu betreiben . . . 24
4.1.1 Issue 1: Testüberarbeitung . . . 24
4.1.2 Issue 2: Gründe gegen das Refactoring . . . 25
4.1.3 Issue 3: Linuxüberschneidungen . . . 25
4.1.4 Issue 4: Mono vs. Multi Repo . . . 25
4.1.5 Issue 5: Wie viel sollte man kommentieren? . . . 25
4.1.6 Issue 6: Hart-kodieren vermeiden? . . . 26
4.2 Erkenntnis 2: Ausschlaggebende Faktoren für die Bereitschaft Zeit in Refactoring zu investieren . . . 26
4.2.1 Faktor 1: Ausgangssituation der Diskutierenden . . . 26
4.2.2 Faktor 2: Haltung . . . 27
4.2.3 Faktor 3: Aufwand . . . 28
4.2.4 Faktor 4: Lust . . . 28
4.3 Erkenntnis 3: Das Kosten-Nutzen-Verhältnis ist entscheidend für die Bereitschaft Refactoring zu betreiben . . . 28

4.3.1 Issue 1: Testüberarbeitung 29
4.3.2 Issue 2: Gründe gegen das Refactoring 29
4.3.3 Issue 3: Linuxüberschneidungen 29
4.3.4 Issue 5: Wie viel sollte man kommentieren? 29
4.3.5 Issue 6: Hart-kodieren vermeiden? 30
4.4 Handlungsempfehlungen 30
4.4.1 Bewusstsein für die Menge des Aufwandes gewinnen 30
4.4.2 Schulung zu dem Herauskristallisieren von Vorteilen des Refactorings 31

5 Diskussion **33**
5.1 Reflexion der genutzten Quelle 33
5.2 Limitierungen 33
5.2.1 Anzahl Material 33
5.2.2 Verständnis des Materials & der Gedankengänge 33
5.2.3 Interpretationsspielraum 34
5.2.4 Kontakt zu Diskutierenden 34
5.2.5 Zeit 34
5.3 Ausblick 34

6 Zusammenfassung **36**

Literaturverzeichnis **37**

Abbildungsverzeichnis

1 Tabelle mit den Kategorien zum Issue 1 (Eigene Darstellung) 17
2 Tabelle mit Konzepten entstanden aus den Kategorien zum Issue 1 (Eigene Darstellung) 18
3 Mindmap mit Konzepten vom Issue 1 (Eigene Darstellung) 19
4 Baum mit Verwurzelungen als Metapher für Verzweigungen von sonst funktionierenden Code [15] 31
5 Zeitverteilung bezüglich der Arbeit mit Code [15] 32

1 Einleitung

1.1 Forschungsfrage

Das Thema „Qualitative Analyse öffentlicher Refactoringdiskussionen in Github Issue Trackern“ leitet sich aus der folgenden Forschungsfrage ab: Welche Faktoren des mentalen Modells von Entwickler:innen beeinflussen die Bereitschaft Refactoring zu betreiben?

1.2 Relevanz der Forschungsfrage

Der Begriff des Refactorings ist den meisten Menschen vor allem in der Softwarearchitektur bereits ein Begriff. Wir kennen den Zustand der Codebasis und die Vorteile, die durch ein Refactoring entstehen können. Es gibt Literatur über die Funktionsweise von Refactoring. Dennoch wird es nicht ansatzweise so häufig durchgeführt, wie vielleicht angenommen. In der Theorie ist die Relevanz bekannt, in der Praxis ist das Refactoring an vielen Stellen jedoch noch nicht weit genug verbreitet. Wieso gibt es einen solchen Unterschied zwischen der Theorie und der tatsächlichen Durchführung? Wie sieht die Motivation und die Bereitschaft von Softwareentwickler:innen gegenüber dem Refactoring aus? Was hindert sie daran, Refactoring zu betreiben? Diese Bachelorarbeit soll ein erster Schritt in Richtung Beantwortung dieser Fragen sein.

1.3 Motivation der Arbeit

Ich habe im Sommersemester 2020 das Modul „Frauen in der Geschichte der Mathematik“ belegt und einen Vortrag über Carola Lilienthal gehalten. Lilienthal ist eine Softwarearchitektin mit dem Schwerpunkt „Technische Schulden.“ Während meiner Recherche zu dem Thema wurde mir die Wichtigkeit dieses Gebietes immer bewusster und ich verstand nicht, wieso es nicht weit verbreitet ist. Aufgrund dessen wollte ich mir angucken, wie Menschen zu technischen Schulden stehen, womit mein Weg letztendlich auch begann. Erst im Laufe der Arbeit spezialisierte ich mich auf den Part des Refactorings.

1.4 Vergleichbare Arbeiten

Wie bereits in der Relevanz der Forschungsfrage erläutert, gibt es sehr viel Literatur dazu, wie Refactoring betrieben werden sollte, wie beispielsweise das Buch von Martin Fowler.[1] Außerdem findet man viele Analysen zur Verbesserung der Softwarequalität nach dem Refactoring.[2] In der Literaturarbeit „Technical Debt Prioritization: State of the Art. A Systematic Literature Review“ [3] gehen die Autor:innen auf die Möglichkeiten der Priorisierung von technischen Schulden ein. Zu den konkreten Beweggründen von Personen Refactoring zu betreiben, habe ich während meiner Recherche keine gefunden und nehme deshalb an, dass es keine gibt. Zumindest existieren keine mir zu diesem Zeitpunkt bekannten.

1.4.1 Aufbau der Arbeit

Dieses Kapitel stellt den Aufbau dieser Bachelorarbeit dar. Zunächst erfolgt in Kapitel zwei eine Reihe von Definitionen, um das Verständnis für das Thema zu fördern. Anschließend gehe ich in Kapitel drei auf meine Vorgehensweise ein. In Kapitel vier folgen dann die Ergebnisse und Handlungsempfehlungen. Kapitel fünf stellt den Diskussionspart dar, in welchem die Quelle reflektiert, Limitierungen dargelegt und ein Ausblick gegeben wird. Abschließend erfolgt in Kapitel sechs die Zusammenfassung der Bachelorarbeit.

2 Einführung in die Grounded Theory

Bei der Grounded Theory handelt es sich um eine Art der Theoriebildung in der qualitativen Datenanalyse. Sie wird vor allem verwendet, um bislang unerforschte Themen zu untersuchen. Die Analyse beruht vollständig auf qualitativen Daten, wie beispielsweise Interviews, Beobachtungen oder schriftlichen Diskussionen. Es werden einzelne Fälle analysiert und aus diesen gemeinsame Muster entwickelt, die zu konzeptuellen Kategorien führen. [4] Die Datenerhebung und -analyse geschieht zeitgleich und durchgehend, bis die sogenannte theoretische Sättigung (siehe 2.1.5) erreicht ist. [5]

2.1 Kodierung

Die Kodierung beginnt wie oben angedeutet parallel mit der Datenerhebung. Bei der Grounded Theory gibt es drei Arten der Kodierung, die nun erklärt werden.

2.1.1 Offenes Kodieren

Das offene Kodieren beschreibt den ersten Schritt des Analyseprozesses. Der Forscher beginnt erstmals das Material sorgfältig durchzuarbeiten und entwickelt hierbei erste Konzepte. Konzepte definieren Strauss und Robin wie folgt: „Konzeptuelle Bezeichnungen oder Etiketten, die einzelnen Ereignissen, Vorkommnissen oder anderen Beispielen für Phänomene zugeordnet werden können.“[5] Es gibt verschiedene Arten des offenen Kodierens. Zum einen kann man Wort-für-Wort kodieren. Hierbei fokussiert man sich auf jedes einzelne Wort und untersucht, ob dieses relevant sein könnte. [6] Die nächste Möglichkeit ist die Satz-für-Satz Kodierung [6], welche ich überwiegend verwendet habe. Bei dieser Variante guckt man sich Sätze im Ganzen an und versucht diese, falls relevant, mit einem Code zu versehen. Ein Code kann auch einer ganzen Textpassage oder sogar dem gesamten Text zugeordnet werden, beispielsweise wenn man mehrere Texte miteinander vergleicht. [6] Es ist wichtig, uneingenommen und offen für Interpretationen zu sein. Codes sind nicht vorher definiert und werden dann Stellen im Text zugeordnet, sondern der Text selbst definiert die Codes. Textpassagen, die auf dasselbe Phänomen hinweisen, werden mit demselben Code versehen. Am Ende entsteht eine Liste mit sämtlichen Codes, die eventuell in unterschiedlichen Texten zu finden sind. Die Forscher verwenden zwei verschiedene Arten von Codes. Die sogenannten *Conceptual Codes*, welche den Inhalt des Konzeptes wiedergeben und die *In-Vivo-Codes*, die eins zu eins den Wortlaut in der jeweiligen Textstelle wiedergeben. [6] Bei dem offenen Kodieren gibt es keine strengen Regeln, sondern viel mehr Leitfragen an bestimmte Phänomene, die man entdeckt.[4] Diese Fragen sind unter anderem: [6]

- Welches Phänomen tritt auf?
- Welche Personen spielen welche Rolle?
- Wann tritt das Phänomen auf?
- Wie häufig tritt das Phänomen auf?
- Wie stark tritt das Phänomen auf?

- Welche Begründungen sind vorzufinden?

Das Ziel ist es, Phänomene zu finden und zu untersuchen. Parallel zu dieser „Phänomenssuche“ werden immer weiter Daten erhoben und analysiert, solange bis neue Daten keine neuen Erkenntnisse mehr hervorbringen und somit eine theoretische Sättigung (siehe 2.1.5) erreicht ist. Bei der offenen Kodierung ist das Vergleichen das A und O. Es werden konstant Texte mit Texten, Texte mit Codes und Codes mit Codes verglichen.

2.1.2 Axiales Kodieren

Das axiale Kodieren erfolgt nach dem offenen Kodieren. Man kann sich das Ganze als Mindmap vorstellen, die dazu dient, Verbindungen und Beziehungen zwischen den einzelnen Konzepten zu finden und herzustellen. Hierbei wird beispielsweise untersucht, welche Konzepte aus welchen folgen und welche einander gegenüberstehen.

2.1.3 Selektives Kodieren

In diesem Schritt wird eine Kernkategorie bestimmt. Diese entsteht nach dem In-Verbindung-Setzen der einzelnen Konzepte. Es wird die Kategorie ausgewählt, welche mit all den anderen Konzepten in Verbindung steht und somit die Mitte von allem ist. In diesem Schritt beginnt die Interpretation und somit zum ersten Mal die Theorieentwicklung. Man betrachtet die bis dato gesammelten Daten und versucht ein Muster zu finden, welches wiederkehrt. Die Vorgehensweise ist dabei wie folgt: Die Codes werden zu sogenannte Kernkategorien zusammengefasst, sodass es anstelle von vielen einzelnen präzisen Codes am Ende wenige grobe Kernkategorien gibt, die dabei helfen, Muster auszuarbeiten. Hierfür geht man folgendermaßen vor: Zunächst sortiert und vergleicht man die zuvor erarbeiteten Codes. Anschließend führt man diese zu einer begrenzten Anzahl von Kernkategorien zusammen, mit derer die folgende Analyse fortgeführt wird.

2.1.4 Memos

In seinem Buch „Einführung in die computergestützte Analyse qualitativer Daten“ definiert Udo Kuckartz Memos als „Form der Fixierung der eigenen Ideen und theoretischen Gedanken“. [4] Gemeint ist, dass Memos als Hilfsmittel für Forscher:innen zu verstehen sind, welche dazu dienen, die eigenen Ideen, Hypothesen und Gedanken auf Papier zu bringen. Memos werden von Beginn an in den Prozess integriert und dauern bis zum Ende an. Die anfangs banalen und chaotischen Memos werden mit der Zeit immer konkreter und komplexer. Sie eignen sich unter anderem auch bei der Zusammenarbeit mehrerer Personen, da man durch diese „in den Kopf“ der anderen gucken und ihre Gedanken nachvollziehen kann.[4] Neben dem Niederschreiben von eigenen Ideen, können Memos auch genutzt werden, um Fragen zu notieren, die man im späteren Verlauf der Analyse untersuchen möchte.

2.1.5 Theoretische Sättigung

Eine theoretische Sättigung liegt vor, wenn das Erheben neuer Daten keine neuen Ergebnisse aufweist und unsere Analyse somit ein Ende hat.[6] [7] Dies ist in der Grounded Theory besonders wichtig, da die Datenerhebung nicht, wie in anderen Analysetechniken, nach einer bestimmten Phase endet, sondern ein ständig wiederkehrender Prozess ist und nur durch das Eintreffen der theoretischen Sättigung ein Ende findet.

2.2 Technische Schulden

„Technische Schulden entstehen, wenn bewusst oder unbewusst falsche oder suboptimale technische Entscheidungen getroffen werden. Diese falschen oder suboptimalen Entscheidungen führen zu einem späteren Zeitpunkt zu Mehraufwand, der Wartung oder Erweiterungen verzögert.“ [7] Mit diesen beiden Sätzen erklärt Carola Lilienthal den Begriff der technischen Schulden. Der Begriff „technische Schulden“ wurde im Jahr 1992 von Ward Cunningham eingeführt und diente primär dazu, das Phänomen der technischen Schulden auf nicht technische Weise den Finanzexpert:innen und Manager:innen greifbar zu machen. Im Wesentlichen gibt es vier Arten von technischen Schulden, welche ich im Folgenden genauer definieren werde.

- 1.Implementierungsschulden

  Bei den Implementierungsschulden handelt es sich, wie der Begriff schon verrät, um Schulden, welche im Code vorzufinden sind. Hierzu gehören beispielsweise zu lange oder komplexe Methoden und Codeduplikate. Man spricht hierbei von *Code-Smells*.

- 2.Design- und Architekturschulden

  Design- und Architekturschulden liegen vor, wenn im Code uneinheitliche oder komplexe Designs für Klassen, Pakete oder Module vorliegen. Kurz gesagt, wenn die Ist-Architektur nicht der Soll-Architektur entspricht.

- 3.Testschulden

  Man redet von Testschulden, wenn nicht ausreichend Tests vorhanden sind. Auch durch das Fehlen von automatisierten Tests kann man Testschulden anhäufen. Dies führt vor allem dazu, dass nicht alle Bereiche genügend getestet und somit einige Fehler nicht gefunden werden.

- 4.Dokumentationsschulden

  Unter den Dokumentationsschulden verstehen sich unvollständige oder veraltete Dokumentationen, die es Nachfolgern erschweren, Code zu verstehen und zu verändern.

Zusammenfassend entstehen technische Schulden während des Programmierens und müssen zu einem späteren Zeitpunkt abgezahlt werden. Sie entstehen unter anderem durch Zeitmangel, Geldmangel oder durch das Wachsen des Codes. Ein Ausweg aus den technischen Schulden ist das Refactoring.

2.3 Refactoring

Bei dem Refactoring handelt es sich um einen Prozess, in dem eine Software komplett oder zum Teil neu strukturiert wird. Die Funktionsweisen für den Endnutzer bleiben dieselben. Lediglich die Codestruktur wird verändert, um „aufzuräumen“. Dies passiert unter anderem, wenn ein Code immer komplexer und unübersichtlicher wird und, wie oben erklärt, technische Schulden abgebaut werden müssen. Dies ist vor allem notwendig, da immer mehr technische Schulden zu einer stärker werdenden Verzögerung der Arbeit führen.

3 Methodik

3.1 Datensammlung

3.1.1 GitHub

Als Quelle für meine Datenerhebung habe ich GitHub verwendet. GitHub ist ein Dienst, der es Teams unter anderem ermöglicht, gemeinsam eine Software zu entwickeln. Hierbei müssen die Teilnehmenden nicht zusammen an einem Ort sein, sondern Personen von überall können auf die gemeinsame Codebasis blicken und an dieser arbeiten. Neben dem Hochladen von Code, bietet die Plattform auch Features, die Diskussionen ermöglichen. Um mir diese Diskussionen zu beschaffen, habe ich das Tool QIF-Tool, welches ich im folgenden Kapitel genauer erklären werde, verwendet.

3.1.2 QIF-Tool

QIF-Tool oder Query Issue Finder-Tool ist ein Programm, das im Rahmen einer Bachelorarbeit von Robert Selack erstellt wurde. Das Tool dient dazu, nach verschiedenen Diskussionen auf der Plattform GitHub zu suchen (GitHub Issue Tracker). Hierfür muss man lediglich Schlagwörter und die Mindestanzahl der Diskutierenden und der Kommentare angeben. Das Tool wurde speziell für den Bereich der technischen Schulden geschrieben, kann jedoch durch die Wahl der Keywörter ebenso für andere Themengebiete verwendet werden.

3.1.3 MAXQDA

MAXQDA ist ein Dienst, der Forschern dabei helfen soll, qualitative Datenanalyse zu betreiben. Forscher:innen binden hierzu die gewünschten Texte ein und versehen sie in der Software beispielsweise mit Codes (siehe 2.1) und Memos (siehe 2.1.4). Des Weiteren hat man die Möglichkeit bei MAXQDA bestimmte Informationen zu den Codes zu erhalten, beispielsweise die Häufigkeit oder den Ort.

3.2 Analyse

3.2.1 Datensammlung

Um an meine Informationen zu gelangen, habe ich das oben erwähnte QIF-Tool verwendet. Ich wusste nicht genau, was mich erwartet und worauf ich mich spezialisieren sollte. Eine Sache, die für die Grounded Theory typisch ist. Man beginnt mit einer relativ offenen und großen Suche und passt diese durch die gewonnenen Informationen an. Ich wählte die Keywords „technical debt“, „code“ und „technische Schulden“. Das sind drei Begriffe, die zwar die Richtung der technischen Schulden angaben, aber dennoch genug Spielraum ließen, um mich zu leiten. Mein Ziel war es hierbei möglichst unvoreingenommen und ohne klare Erwartungen als Beobachterin herauszufinden, wie Personen in ihrer eigenen, neutralen Umgebung zu technischen Schulden stehen. Deshalb entschied ich mich für das Analysieren von GitHub Issues und gegen Interviews. In Interviews geben Menschen die Antworten, die sie geben

möchten. Menschen stellen sich selbst so dar, wie sie gerne wären. In hitzigen Diskussionen sieht das aufgrund der entstehenden Emotionen anders aus. Hinzu kommt, dass Menschen sich bei den Issues weniger beobachtet fühlen, wie wenn sie jemand beispielsweise interviewen würde. Das Analysieren von Issues bringt natürlich auch Schwierigkeiten mit sich, da man der stille Beobachter ist, der zu keinem Zeitpunkt eingreifen wird. Somit konnte ich keine konkreten Fragen stellen oder in ein Gespräch einsteigen. Ich musste mit dem Material arbeiten, welches ich vorfand und konnte es nicht beeinflussen. Umso wichtiger war es, auszuwählen, welche Issues ich betrachten würde. Durch die 12-Wochen-Arbeitszeit und die Tatsache, dass ich qualitative und nicht quantitative Forschung betreiben wollte, war mir relativ schnell bewusst, dass ich definieren muss, welche Issues einer weiteren Betrachtung wert sind und welche von Beginn an verworfen werden können. Einige der Kriterien, um Issues in die engere Auswahl zu nehmen, waren folgende:

- Passend zur Forschungsfrage
- Mehr als eine Person
- Nicht zu viele Personen, sodass keine richtige Diskussion entsteht
- Häufiges Auftreten des Begriffes „technischen Schulden“ beziehungsweise Synonyme davon
- Mehr als fünf Sätze

Diese Eigenschaften entstanden, nachdem ich mir den ersten Block an Issues (50 Issues) angeguckt hatte. Die Issues, die mir ausgegeben wurden, waren nicht relativ lang, sodass ich ziemlich schnell erkennen konnte, ob es relevant für meine Forschungsfrage war oder nicht. Ich speicherte die Issues in unterschiedlichen Ordnern je nach Relevanz ab. Die Issues, die es in die engere Auswahl geschafft hatten, kopierte ich raus. Ich erstellte eine PDF-Datei mit dem Text aus der Diskussion. Parallel errichtete ich ein Register, in welchem ich die Issues chronologisch nach ihrer Issue-Nummer sortierte, um im Laufe der Recherche keine doppelt zu lesen.

3.2.2 Datenauswertung

Im nächsten Schritt begann ich die Issues sorgfältig durchzuarbeiten. Hierfür druckte ich das Issue aus und markierte alle für mich relevant erscheinenden Stellen. Ich versuchte hierbei zunächst im Sinne der Grounded Theory Satz für Satz jeweils eine Erkenntnis herauszuziehen und diese mit einem Code zu versehen, welchen ich mit Hand an die jeweilige Stelle schrieb. Zu Beginn waren meine „Codes“ nichts anderes als reine Zusammenfassungen der Aussage. Dies wurde mit der Zeit jedoch immer besser und besser. Anschließend erstellte ich am Laptop ein separates Dokument, in welchem ich eine Tabelle anlegte. Die Tabelle sollte meine Codes darstellen. Sie bestand aus drei Spalten. Die erste Spalte nummerierte den Code. Die zweite Spalte stellte den Code dar, während die dritte die zum Code gehörenden Zeilenzahl im Text wiedergab, damit man zu jedem Zeitpunkt den genauen Wortlaut im Text finden konnte. Ich füllte die gesamte Tabelle, indem ich die zuvor per Hand aufgeschriebenen Codes nach und nach in die Tabelle eintrug.

1	Menge Arbeit alle Tests zu ändern	2
2	Nicht Teil unserer Leistungen	2-3
3	Neukodierung aller Tests	4
4	Verzögerung unserer Ziele	6
5	Haben bereits viel zu tun	6-7
6	Entspricht nicht den globalen Programmierstandards	8
7	Muss den Kodierungsstil ändern	12
8	Kodierungsstil in dem Test in den Repositories ist ähnlich und lesbar	14
9	Wenn ich Tests überarbeite, dann kodiere ich sie sowieso neu	17
10	Längere Definitionen -> Tests besser lesen -> Tests besser verstehen mit sehr geringen Kosten für ein zusätzliches Konzept	20-22
11	Reduziert Verwirrung	28
12	Reduziert Größe der Testdateien	30
13	Müssen alle Tests neu machen	32
14	Müssen alle neu lernen, wie man Dinge spottet	33
15	Jeder muss seine Arbeitsweise ändern	35
16	Großes Minus	36
17	Möchte nicht ewig an Refactoring von Tests arbeiten	36-37
18	Wird lange Zeit in Anspruch nehmen	37
19	Macht Dinge etwas klarer wenn man Code nicht, aber nicht so sehr, dass es die Zeit wert ist	38-39
20	Scheint eine Tonne Arbeit zu sein	41
21	Scheint eine Menge Arbeit zu sein	42
22	Scheint viele Leute zu verärgern	43
23	Nur ein kleiner Vorteil von besserer Lesbarkeit	43
24	Niemand war dagegen	44
25	Mitwirkende sollten Recht haben zu wählen	45

Abbildung 1: Tabelle mit den Kategorien zum Issue 1 (Eigene Darstellung)

Somit konnte ich auf einen Blick sinngemäß gleiche oder ähnliche Codes finden. Diese Codes gruppierte ich, indem ich sie mit derselben Farbe markierte. Im Anschluss daran habe ich eine zweite Tabelle für die Konzepte erstellt. Ich habe die zusammengehörenden Codes (selbe Farbe) unter einem zu allen Codes passenden Namen zusammenaddiert. Die Spalten unterschieden sich hierbei von denen der ersten Tabelle. Anstelle der Zeilenzahl des Textes stand in der dritten Spalte die Nummer der zu diesem Konzept gehörenden Codes (Kategorien - erste Spalte der vorherigen Tabelle), sodass man direkt sehen konnte, welche Codes zu dem Konzept gehörten. Hierbei ging es zunächst darum, jeden Fall einzeln zu betrachten und zu analysieren. Diesen Vorgang wiederholte ich für jedes Issue.

Nr.	Kategorie	Konzepte
1	Wird viel Arbeit sein	1,17,18,20,21
2	Nicht Teil unserer Leistungen	2
3	Neukodierung aller Tests	3,13
4	Verzögerung der Ziele	4
5	Haben bereits viel zu tun	5
6	Entspricht nicht den globalen Programmierstandards	6
7	Neuerlenen und Anpassen	12,14,15
8	Kodierungsstil in dem Test in den Repositories ist ähnlich und lesbar	8
9	Wenn ich Tests überarbeite, dann kodiere ich sie sowieso neu	9
10	Längere Definitionen -> Tests besser lesen -> Tests besser verstehen mit sehr geringen Kosten für ein zusätzliches Konzept	10
11	Reduziert Verwirrung & Größe der Testdateien	11.12
12	Großes Minus	16
13	Lohnt sich nicht	19,23
14	Verärgert viele Leute	22
15	Verärgerte keine Leute	24
16	Mitwirkende sollten Recht haben zu entscheiden	25

Abbildung 2: Tabelle mit Konzepten entstanden aus den Kategorien zum Issue 1 (Eigene Darstellung)

Im nächsten Schritt erfolgte das In-Verbindung-Setzen der einzelnen Konzepte. Hierfür druckte ich mir die einzelnen Codes als Papierschnipsel aus und legte sie mir nebeneinander auf den Tisch. Ich versuchte eine Verbindung zwischen den einzelnen Papierschnipseln zu finden und ihre Beziehung zueinander zu skizzieren, indem ich mir unter anderem folgende Fragen stellte:
Welcher Code folgte aus welchem?
Welcher Code setzt welchen voraus?
Welche Codes stehen im Gegensatz zueinander?
Ich entschied mich für eine Kernkategorie, von der alle anderen Codes abhingen und wählte sie als Zentrum der Mindmap. Hier als kleines Beispiel die Mindmap, die aus der obigen Tabelle entstand:

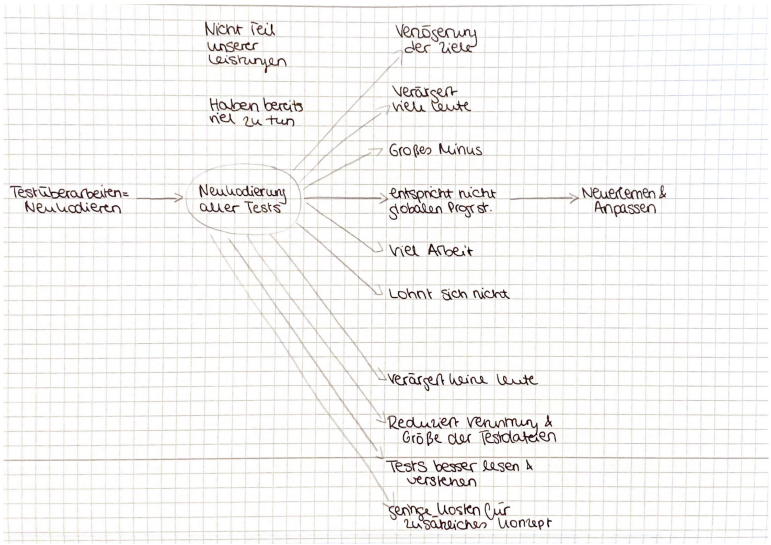

Abbildung 3: Mindmap mit Konzepten vom Issue 1 (Eigene Darstellung)

Ich hatte zu dem aktuellen Zeitpunkt drei Issues. Ich erstellte zu jedem Issue die entsprechenden Tabellen und Mindmaps, sodass ich nun einen ersten Eindruck über die einzelnen Issues gewann. Im nächsten Schritt wollte ich die einzelnen Issues und deren bis jetzt getroffenen Aussagen in Bezug zueinander betrachten. Hierfür eignete sich die Software MAXQDA. Das Arbeiten mit MAXQDA machte meine Arbeit effizienter, da ich nun auf einen Blick die Codes von allen Issues einsehen konnte, ohne ständig zwischen den einzelnen Dokumenten hin- und herzuwechseln. Außerdem erkannte ich, welche Codes wie häufig und an welchen Stellen auftraten, um wiederkehrende Muster zu erkennen. MAXQDA machte meine Arbeit somit einfacher, ohne die Ergebnisse zu verschlechtern. Ich fügte meine Dokumente in das MAXQDA ein, markierte die jeweiligen Textstellen und versah sie mit den dazugehörigen Codes. Hierbei habe ich darauf geachtet, dass ich ähnlichen Codes aus verschiedenen Dokumenten denselben Namen gab, damit später ein mehrfaches Auftauchen erkennbar ist. Ich fasste hierbei zunächst den Entschluss, das Ganze in drei große Kategorien zuzuordnen.

- „Gründe für technische Schulden“
- „Gründe gegen technische Schulden“
- „Folgen von technischen Schulden“

Je mehr ich in das Thema einstieg, desto häufiger bemerkte ich, dass das ganze Thema nicht so schwarz und weiß ist und ich erkannte, dass auch Faktoren, wie Position, Grundeinstellung, Arbeitsmoral und Ausgangssituation eine große Rolle spielten. In einem der Issues tauchte als Begründung für das Aufnehmen von technischen Schulden die Tatsache auf, dass die jeweilige Person als Freelancer häufig neben der Arbeit

zusätzlich Codes schreibt und somit nicht die Zeit hat einen schönen Code zu schreiben ("from the theoretical point or practical point in team projects I am absolutely with you! But as a freelancer currently I sometimes write some tools here and there for customers and others, besides other work.") [9]. Mit dieser neuen Erkenntnis begann ich, zusätzliche Codes hinzuzufügen, die die oben genannten Punkte abdeckten, wie beispielsweise „einzelner Programmierer“, „arbeitetet in größeren Teams“, da diese für die Einstellung der Personen eine große Rolle spielten. Parallel begann ich mit den Memos (siehe 2.1.4). Die Memos dienten dazu Fragen und Gedanken aufzuschreiben, die ich im Laufe der Analyse versuchen würde, zu beantworten. Anschließend begann der ganze Prozess außerhalb von MAXQDA von vorne. Ich verfeinerte basierend auf meinen bisherigen Ergebnissen meine Keywords, las mir den nächsten Issue-Block durch und analysierte diesen wie oben beschrieben. Im Laufe der Arbeit kam meine Analyse zu einem Wendepunkt. Ich stieß immer häufiger darauf, dass technische Schulden im Großteil der Fälle mit dem Wort Refactoring auftauchten, weshalb ich entschied, diesem Wort mehr Aufmerksamkeit entgegenzubringen. Aufgrund dessen beschloss ich nicht mehr nur noch darauf zu gucken, wann Personen bereit waren, technische Schulden einzugehen, sondern wann sie bereit waren, diese durch ein Refactoring zu beheben, wenn sie schon vorhanden waren. Ich betrachtete also nicht mehr nur noch den Weg in die technischen Schulden, sondern vor allem aus den technischen Schulden. Dadurch verschob sich der Fokus dieser Arbeit immer mehr in Richtung „Gründe und Anreize für beziehungsweise gegen das Refactoring“. Hierbei spielte auch die Tatsache, dass es zu diesem Thema viel mehr zu finden gab, als zu technischen Schulden allein, eine große Rolle. Mein Hauptfokus lag nun auf dem Thema Refactoring und nicht mehr speziell auf dem Thema „technische Schulden“. Ich begann nach den Worten „refactoring“, „reasons“, und „code“ zu suchen und neue Issues nach dem obigen Prinzip zu analysieren. Dies führte mich wiederum zu der Erkenntnis, dass Personen den Willen Refactoring zu betreiben von dem damit verbundenen Aufwand abhängig machten, sodass sich mein Fokus wieder verschob. Ich suchte ab dem Zeitpunkt nach den Wörtern „refactor“, „time“, „work“, „code“ und „effort“ und blieb bei diesen Wörtern bis zu dem Ende meines Analyseprozesses.

3.2.3 Theoriekonstruktion

Kommen wir nun zu dem Teil der Theoriekonstruktion. Der Abschnitt in meiner Bachelorarbeit, in dem aus den gesammelten Codes Erkenntnisse gewonnen werden. Man kann sich das so vorstellen: Man sitzt vor den Codes und guckt sie sich solange an, bis daraus eine schlaue Erkenntnis wird. Zumindest dachte ich das, bis ich realisiert habe, dass da mehr dazugehört. Ich begann zunächst nach den Codes zu suchen, die am häufigsten auftraten. Anschließend überprüfte ich, ob diese in einem bestimmten oder in verschiedenen Issues häufig auftraten. Ich sah mir die Personen genauer an und welche Meinung sie in dem Issue vertraten. Ob sie etwas über sich oder ihre Ausgangssituation verrieten. Ich versuchte nachzuvollziehen, um was für Projekte es sich handelte. Relativ schnell bemerkte ich beispielsweise, dass das Betrachten des Programmierteils in den Issues zu nichts führte, sodass ich diese von Beginn an ausblendete. Hier ist ein kleiner Einblick mithilfe eines Beispiel: Der Code, der am häufigsten auftrat, war "viel Arbeit". Dieser tauchte in fünf von sieben Issu-

es auf. Also beschloss ich diesen näher zu betrachten. Beim genauen Angucken der Stellen, in denen dieser Code auftrat, fiel mir dann auf, dass sich die Diskutierenden häufig über den zusätzlichen Aufwand beschwerten, diesen jedoch nie mit quantitativen Werten stützten. Stattdessen wurden Begriffe wie „HUGE“ [12] oder „colossal“ [11] verwendet. Ich erstellte also Memos mit Fragen wie: „Wie schätzt diese Person den Aufwand ein“ und „Unterschiedliche Einschätzung über den Aufwand?“. Aber dazu in den folgenden Kapiteln mehr.

3.2.4 Verworfene Issues

In diesem Kapitel gehe ich auf die verworfenen Issues ein. Im Laufe meines Analyseprozesses habe ich über 250 Issues gelesen, von denen ich am Ende 102 Issues näher betrachtet habe. In meine Analyse habe ich letztendlich sieben Diskussionen einbezogen. Aus diesen sieben Issues sind insgesamt 39 Codes entstanden, die zusammen an 115 Stellen auftauchten. Der Code, der am häufigsten auftauchte, war zwölfmal vorzufinden. Kommen wir zunächst zu den rund 150 Issues, die ich nicht näher betrachtet habe. Diese habe ich aufgrund von einigen Kriterien, welche ich nun näher beschreiben werde, direkt aussortiert.

- Nicht passend zur Forschungsfrage

Diesen Punkt zu erklären ist nicht einfach, da es sich hierbei um eine subjektive Einschätzung handelt. Es gab einige Issues, in denen die Wörter „technische Schulden“ oder „Refactoring“ beispielsweise nur einmal vorkam und sonst keine Relevanz spielte. Diese Issues wurden von vorneherein aussortiert.

- Zu kurz

Issues, die in meinen Augen zu kurz und nicht aussagekräftig waren, habe ich ebenfalls verworfen.

- Keine Diskussion entstanden

Während dem Durchgehen, stieß ich ebenfalls auf Issues mit einzelnen Aussagen über das Refactoring, die jedoch nicht diskutiert oder weiter betrachtet wurden, weshalb auch diese aussortiert wurden.

- Zu viele Diskutierende

In diesem Fall gab es zu viel Input von zu vielen unterschiedlichen Personen, sodass es zu keinem richtigen Austausch unter den Diskutierenden kam. Die Parteien haben ihre Aussage getätigt und sich danach nicht weiter dazu geäußert. Es konnte keine Debatte entstehen, da es zu viel unterschiedlichen Input gab.

Um die Frage zu beantworten, wieso ich 250 und nicht mehr Issues betrachtet habe, muss man verstehen, wie lange die Analyse eines Issues dauert. Das erste Durchlesen und Aussortieren, geht relativ schnell. Das nähere Analysieren der potenziellen Issues steigt jedoch exponentiell vom Zeitaufwand her. Man steigt in das Issue ein und versucht Satz für Satz herauszufinden, ob brauchbare Informationen in diesem Issue sind. Je mehr Issues man betrachtet und in den Analyseprozess eingebunden hat, desto einfacher, aber auch zeitaufwändiger wird das Betrachten neuer Issues. Einfacher, da man mittlerweile ein Gefühl dafür hat, was reinpassen könnte. Zeitaufwändiger, da man beginnt Verbindungen zu den anderen Issues zu suchen und bei Befund zu skizzieren. Somit steigt der Zeitaufwand mit jedem betrachteten und vor allem eingebundenen Issue. Durch das kontinuierliche Finden neuer Issues und neuer interessanter Aspekte in diesen Issues konnte keine theoretische Sättigung erreicht werden. Um diese zu erreichen, wäre ein größerer Zeitaufwand nötig, welcher im Rahmen dieser Arbeit nicht gegeben war.

4 Ergebnisse

Bevor ich in die Phase der Ergebnisse einsteige, erfolgt zunächst eine kleine Zusammenfassung zu jedem Issue, um ein besseres Verständnis zu erzeugen.

Die Namen der jeweiligen Personen wurden aus Datenschutzgründen gehashed.

Issue 1: Testüberarbeitung [8]

Kurze Beschreibung:
Person 46fdf6063c0ce6f8156171e1c07a7bf9 bringt den Vorschlag, die aktuellen Tests effizienter zu gestalten, um ein leserlicheres und klareres Ergebnis zu erzielen. Er stößt dabei auf Gegenwind, weil die anderen beteiligten Personen der Meinung sind, dass sich der Aufwand nicht lohnt und sie ohnehin genug zu tun hätten.

Issue 2: Gründe für das Refactoring [9]

In diesem Fall haben wir zwei Seiten, die über das Thema Refactoring diskutieren. Die eine Seite ist fest davon überzeugt, dass das Verhindern von Refactoring in der Zukunft zu großen Problemen führen wird und es deshalb ein fester Bestandteil jedes Prozesses sein sollte. Die Gegenseite argumentiert damit, dass es nicht ihre Aufgabe sei den Code zu verschönern und dies auch nicht bezahlt werden würde.

Issue 3: Linuxüberschneidungen [10]

FreeBSD verhält sich bei dem vorliegenden Projekt in einigen Punkten exakt wie Linux, sodass viel Code doppelt vorhanden ist. Das Problem ist, dass es nicht komplett identisch ist und einige Teile sich unterscheiden.
Der Vorschlag lautet:

- Mehr Code in gemeinsamen Unix-Teil verschieben, wenn dieser über mehrere Plattformen hinweg gleich ist.
- Das Verhalten auf Plattformen überschreiben, die sich unterscheiden.

Issue 4: Mono-Repo vs. Multi Repo [11]

Es wird vorgeschlagen Multi-Repo in Mono-Repo umstrukturieren.
Die Argumente hierfür sind:

- Identifikation von Auswirkungen von Änderungen funktioniert besser.
- Hilft Entwicklerteam und Tooling-Team.
- Es wird möglich, mehr wiedererkennbaren Code zu refaktorisieren und abzuziehen.

Issue 5: Wie viel sollte man kommentieren? [12]

Person 457a911490ce834498170ea2112f13bc findet übermäßige Kommentare unnötig, da diese „verrotten“[12]
und eine Ausrede für schlechten Code darstellen. [12]
Dies führt zu der Diskussion darüber, wie viel man in dem Projekt kommentieren sollte. Hierbei stößt Person 457a911490ce834498170ea2112f13bc auf viel Gegenwind, da beispielsweise für andere Programmier:innen eine gute Dokumentation und somit Kommentierung fest mit dem Code einhergeht.

Issue 6: Hart-Kodieren vermeiden? [13]

Person 886fb948679927eddde145a61bcb636b schlägt vor, anstatt hart zu codieren, Bausteine zu bauen und somit das Problem zu lösen. Eine kurze Diskussion über den Aufwand entsteht.

Issue 7: Test- & überprüfungsgrad [14]

Das Problem, das hierbei angegangen wird, ist, dass in einigen PRs der Grad des Testens und der Überprüfung heuristisch festgelegt wird. Die Intention hinter dem Issue ist es nun in einem Wiki bestimmte Stufen festzulegen, um zu bestimmen, welche Stufen für das Testen und die Überprüfung, für neue Ergänzungen der mathematischen Bibliothek, notwendig sind.

4.1 Erkenntnis 1: Die Aufwandseinschätzung ist relevant für die Bereitschaft Refactoring zu betreiben

In sechs von sieben Issues war der Code „Viel Arbeit“ vorhanden.

4.1.1 Issue 1: Testüberarbeitung

In dem ersten Issue [8] hat die Gegenseite des Refactorings ihre Entscheidung damit begründet, ohnehin schon genug zu tun zu haben und ein solches Refactoring zu einem zusätzlichen Aufwand führen würde.

- „This is not part of our deliverables“ [8]
- „And we have lots on our plate already.“ [8]

Person 46fdf6063c0ce6f8156171e1c07a7bf9 hingegen sieht das Ganze nicht als zusätzlichen Aufwand, sondern als Prozedere, dass ohnehin schon geschieht, sobald dieser Tests verbessert. [8]

- „And when I fix tests I actually recode them anyways to match other standards“ [8]

4.1.2 Issue 2: Gründe gegen das Refactoring

Person d23bc6e894a445fe8eb50fe3b95ce27f gibt zu verstehen, Refactoring in einem kleinen Maße zu machen, da er noch andere Arbeit nebenbei und somit keine Zeit für diesen zusätzlichen Aufwand habe. [9]

- „I do refactor a lot. But not to the degree that is mentioned here sometimes“ [9]
- „But as a freelancer currently I sometimes write some tools here and there for customers and others, besides other work“ [9]

Zusätzlich bestärkt d23bc6e894a445fe8eb50fe3b95ce27f seine Sichtweise, indem er erklärt, dass Code fertig ist, sobald er funktioniert und somit keinen weiteren Aufwand erfordert. [9]

- „After a while there already is working code, maybe actually not even bad,...“ [9]

Als Gegenargument in Bezug auf den Aufwand wird hier genannt, dass man durch das Vermeiden von Refactoring einen immer schlechteren Code bekommt und somit immer weniger Zeit haben wird, einen sauberen Code zu schreiben, sodass dieser mit der Zeit noch schlechter wird und Person d23bc6e894a445fe8eb50fe3b95ce27f am Ende noch mehr Aufwand erzeugen muss, um das Ganze zu refactoren. [9]

- „...you will forever be fighting with bad code you wrote before and so will have even less time to make good the new code. And you will never escape that downward spiral“ [9]

4.1.3 Issue 3: Linuxüberschneidungen

In dem Issue 3 [10] wird das lediglich kurz angeschnitten.

- „Rather than trying to do some massive refactoring in advance, I think it makes sense to just tackle individual cases as they arise, and if there are cases that don't map to one of the above cases, we can figure it out as it comes up.“ [10]

4.1.4 Issue 4: Mono vs. Multi Repo

Person cab7458dddfab254d77dcb3a879be5d3 nennt den „kolossalen Aufwand“ als einen der Gründe für das bisher nicht getätigte Refactoring. [11]

- „But the colossal effort required to switch back to a monorepo [...] has prevented me from taking such drastic action, ...“ [11]

4.1.5 Issue 5: Wie viel sollte man kommentieren?

In diesem Issue [12] ging es vor allem darum, inwieweit es sich lohnt, viel bzw. wenig zu kommentieren. Eine der Argumente gegen das viele kommentieren war der Aufwand, der dabei entstehen würde.

- „Because i agree that comments should be treated as code (like tests), but that comes as a cost.“ [12]
- „There is a HUGE cost to documentation,...“ [12]

4.1.6 Issue 6: Hart-kodieren vermeiden?

In diesem Issue [13] haben wir ein Theorie-Praxis-Problem.
Person 6d274a1860c9a19e8143812cd32c5114 hat durchaus Verständnis für den Nutzen von Refactoring in der Theorie, ist aber nicht davon überzeugt, ob sich der Aufwand in der Praxis lohnt.

- „In theory I would love to get rid of the special-casing for NamedTuple, if possible. (Then again I'd also love to get rid of the special-casing for TypedDict.)" [13]
- „In practice I'm not sure that it's worth anybody's effort" [13]

Wenn auch wenig überraschend zeigt sich hier vor allem durch die Häufigkeit des Begriffs Aufwand bzw. dessen Synonyme, wie relevant die Frage des Aufwandes für die Abwägung eines Refactorings zu sein scheint.

4.2 Erkenntnis 2: Ausschlaggebende Faktoren für die Bereitschaft Zeit in Refactoring zu investieren

In meiner Analyse bin ich darauf gestoßen, dass mehrere unterschiedliche Faktoren eine Rolle spielen, wenn es darum geht, ob sich Menschen Zeit nehmen Refactoring zu betreiben. Auf diese vier Faktoren werde ich nun näher eingehen.

4.2.1 Faktor 1: Ausgangssituation der Diskutierenden

Ausgangssituationen beschreiben die mentalen Modelle der Diskutierenden innerhalb der GitHub-Issues. Diese sind durch mehrere Punkte bestimmt. Diese Punkte sind:

- Das Anstellungsverhältnis des Diskutierenden (Freelance, Vollzeit)
  - „like me there are thousands of thousands of programmers that do not work in a bigger team" [9]
  - „But as a freelancer currently I sometimes write some tools here and there for customers and others, besides other work" [9]

  An diesen Zitaten ist klar zu erkennen, dass die Tatsache, dass diese Person ein Freelancer:in ist, ausschlaggebend für seine/ihre Haltung gegenüber dem Refactoring ist. [9] Dies verdeutlicht diese Person, indem sie einräumt Refactoring zuzustimmen, wenn es sich um Teamarbeit handelt [9] :

  - „from the theoretical point or practical point in team projects I am absolutely with you!" [9]

- Erfahrung mit Folgen von angehäuften technischen Schulden
  - „Believe me, I've been in situations where I just wanted to ship and felt like I had no time to clean up and design better. I learned the hard way. [9]
  - „But also the other way round Guess, I've seen both sides of the coin." [9]

Hier haben wir zwei Parteien, die ihre Meinung auf ihre Erfahrung stützen. Das erste Zitat ist von einer Person, die sich für das Refactoring ausspricht, da sie schon negative Erfahrungen gemacht hat, als sie ebenfalls nur abliefern und kein zusätzliches Refactoring betreiben wollte. [9] Die zweite Partei hingegen behauptet ebenfalls mit beiden Varianten (Refactoring betreiben / belassen) Erfahrung zu haben. [9] Diese Person entscheidet sich dennoch bewusst gegen das Refactoring.

- Gegebenheiten
  - „I didn't realize that the whole Zowe community had a sort of standard for testing“ [8]
  - „technically I presented this idea in the very beginning and nobody was against it.“ [8]

  In diesen beiden Fällen beeinflussen die äußeren Umstände die Entscheidung der Diskutierenden. Die Person hinter dem ersten Zitat war zunächst für das Refactoring, verwarf diese Gedanken jedoch, als sie erfuhr, dass es für die Zowe Community bereits einen Standard gibt, der beibehalten werden sollte, da sich sonst viele Leute einen neuen Programmierstil aneignen müssten. [8] In dem zweiten Zitat geht es darum, dass diese Person die Idee bereits anderen Personen präsentiert hat und bisher nicht auf Gegenwind gestoßen ist, was sie in ihrer Meinung bestärkt. [8]

Diese Punkte bilden zusammen das Gerüst, aus dem sich die Ausgangssituation der Diskutierenden definieren lässt. Sie stellt die Grundlage der folgenden Meinungsbildung bezüglich des Zeitaufwandes und Refactorings dar.

4.2.2 Faktor 2: Haltung

Zum Teil von der Ausgangssituation beeinflusst, spielt die Haltung als Faktor ebenfalls eine Rolle im Zusammenhang mit der Frage, ob Menschen bereit sind, Zeit in Refactoring zu investieren. Die Haltung beschreibt in diesem Fall die Meinung der Diskutierenden zum Thema Refactoring. Hierbei fielen häufig Ansichten auf, die gefestigt schienen und indiskutabel waren.

- „Code I write is done when it works“ [9]

- „I don't really care too much about a bit of ugly code“ [13]
- „Also when being „in a flow“ most people probably don't „stop“ , but keep on writing“ [9]

Hierbei erkennen wir die Haltung „Ich habe das immer so gemacht “ oder „In der Praxis wird das so gemacht.“ . Zwei Punkte, die verdeutlichen, dass das Refactoring in der Praxis immer noch nicht verbreitet ist und bisher nicht als benötigte Teilleistung angesehen wird.

4.2.3 Faktor 3: Aufwand

Der Aufwand ist in Erkenntnis 1 bereits definiert.

4.2.4 Faktor 4: Lust

Der letzte Faktor, der bei der Analyse herausstach, war der Faktor der Lust der Diskutierenden Zeit in das Refactoring zu investieren. Hierbei fiel auf, dass die fehlende Bezahlung beispielsweise einen negativen Einfluss auf die Bereitschaft für Refactoring hatte. [9] Generell ist die Aufgabe von Programmierer:innen, ein Produkt oder Feature zu einem bestimmten Zeitpunkt zu liefern. Hierbei spielt es in erster Linie keine Rolle, wie der Code aufgebaut ist solange dieser funktioniert. Die Entscheidung, ob man einen guten und nachhaltigen Code schreiben will, liegt somit im Ermessen des Programmierers allein und wird in diesem Sinne nicht extra bezahlt. Dies kann dazu führen, dass Personen Refactoring als unbezahlte Arbeit ansehen und somit nicht durchführen wollen, wie in den folgenden Zitaten zu erkennen ist.

- „that time wouldn't get paid anyhow“ [9]
- „This is not part of our deliverables.“ [8]

Außerdem stellte sich heraus, dass die Personen wenig Lust hatten viel Zeit in das Refactoring zu stecken. [8]

- „I don't want to work on refactoring tests forever“ [8]

Dieser Faktor muss von dem Faktor Aufwand abgegrenzt werden. Sie unterscheiden sich darin, dass der Faktor Aufwand einen stärkeren Bezug auf potenzielle Konflikte in Bezug auf Deadlines und das generelle Arbeitsvolumen der Diskutierenden nimmt. Der Faktor der Lust wiederum bezieht sich stärker auf eine Abneigung gegenüber höherem Aufwand aus anderweitigen, zuvor bereits genannten Gründen.

4.3 Erkenntnis 3: Das Kosten-Nutzen-Verhältnis ist entscheidend für die Bereitschaft Refactoring zu betreiben

Diese Erkenntnis teilt sich in zwei Bereiche auf. Zum einen die Tatsache, dass sich der Aufwand in den Augen der Personen nicht lohnt. Das Interessante an dieser Stelle ist, dass der Nutzen nicht nur, wie zu erwarten wäre, hinterfragt wird, sondern explizit davon ausgegangen wird, dass es keinen Nutzen gibt. Dass die Diskutierenden den Mehrwert des Refactorings hinterfragen, ist wenig überraschend, da es im Voraus schwierig ist, diesen abzuschätzen. Dass man generell von keinem Nutzen ausgeht, würde voraussetzen, dass man diesen Nutzen voraussagen kann. Eine Tatsache, der ich skeptisch gegenüberstehe.

Zum anderen handelt es sich in diesem Abschnitt darum, dass das Refactoren laut einigen Personen zusätzlichen Aufwand aufbringt im Sinne von „Sachen komplizierter machen“ [10] beziehungsweise „Neuerlernen von beispielsweise Codestilen“. [8]

4.3.1 Issue 1: Testüberarbeitung

In diesem Issue [8] spricht sich die Gegenseite gegen das Refactoring aus, mit dem Argument, dass sich der Aufwand nicht lohnt. Dies unterstreicht sie, indem sie erwähnt, dass der Mehrwert, den man durch das Refactoring generieren würde, gering ist.

- „So it seems like a lot of work and rubbing a lot of people the wrong way just to get a small benefit of better readability.“ [8]
- „It does make things somewhat clearer when reading the code, but not so much so that it's worth the time spent learning how to do it for some special cases“ [8]

Außerdem wird als Gegenargument genannt, dass man etwas neu lernen muss und somit zusätzlicher Aufwand entsteht. [8]

- „...this is a new way of programming standard that I need follow.“ [8]
- „Rather than to contribute based on what I know, I have to change the coding style.“ [8]
- „We will all have to relearn how to mock some things (like the trouble I had today with mocking a static class function in the new way)“ [8]

4.3.2 Issue 2: Gründe gegen das Refactoring

In diesem Fall [9] begründet Person acc4ffd145c8c3448452340e9638346b damit, dass der Aufwand sich dahingehend nicht lohnt, dass es im Gegenzug keine Bezahlung gibt und sich somit für diese Person der zusätzliche Aufwand nicht lohnt.

- „I simply do not have more time to improve the coding, that time wouldn't get paid anyhow“ [9]

4.3.3 Issue 3: Linuxüberschneidungen

In diesem Issue [10] wird als Bedenken gegen das Refactoring die eventuell aus dem Refactoring resultierende Komplexität genannt.

- „Aside from the perf implications (which are a sufficient argument to me), I'd be a bit worried about how complex the interop areas were getting if they required that much class hierarchy.“ [10]

4.3.4 Issue 5: Wie viel sollte man kommentieren?

In diesem Issue [12] erwähnt die Gegenseite, dass sich gute Dokumentation nicht immer auszahlt und somit nicht in jedem Fall angewendet werden sollte.

- „There is a HUGE cost to documentation, and the benefits don't ALWAYS materialize.“ [12]

4.3.5 Issue 6: Hart-kodieren vermeiden?

In diesem Issue [13] wird das Bedenken geäußert, dass der Aufwand sich nicht lohnen könnte und somit kein Mehrwert entsteht.

- „In practice I'm not sure that it's worth anybody's effort“ [13]

Zusammenfassend aus den beiden ersten Erkenntnissen kann man sagen, dass einige Faktoren eine Rolle spielen, wenn es darum geht, ein Urteil über die Bereitschaft der Leute zu treffen, Refactoring zu betreiben. Es fällt dabei auf, dass zur Begründung Ihrer Positionen bezüglich Zeit und Aufwand nie quantitative Aussagen getätigt wurden. Dies könnte unter anderem dafür verantwortlich sein, dass die Diskutierenden ihre Einschätzung zu Aufwand und Zeit nur schlecht miteinander vergleichen können. Stattdessen werden primär Adjektive wie „Forever“ [8] oder „HUGE“ [12] verwendet, die aber bei unterschiedlichen Diskutierenden Unterschiedliches bedeuten können und damit die Erarbeitung eines Konsenses erschweren. Es ist schwierig im Voraus den Impact zu schätzen, sodass man dies kaum umgehen kann. Aber es wäre ein großer Schritt, überhaupt zu schätzen, wie lange das gesamte Prozedere dauern würde und ob es in den Zeitplan passt. Ohne eine solche Kosten-Nutzen-Analyse kann eine Entscheidung für oder gegen das Refactoring nicht effektiv getroffen werden.

Ziehen wir die dritte Erkenntnis mit heran, folgt, dass das Refactoring mehrheitlich als nicht wertschöpfend eingeschätzt wird und somit insgesamt auf Ablehnung stößt. Refactoring wird im Endeffekt nicht als Produkt und insbesondere nicht als Teil eines Produktes gesehen. Daher wird Refactoring als nicht bezahlte Arbeit ohne direkten finanziellen Mehrwert wahrgenommen.

4.4 Handlungsempfehlungen

4.4.1 Bewusstsein für die Menge des Aufwandes gewinnen

Eines der Hauptprobleme, welches heraussticht, ist das fehlende Bewusstsein für die tatsächliche Dauer und den Aufwand des Refactorings. Hierfür wäre es bereits hilfreich, wenn die diskutierenden Personen ihre Aufwandsschätzung mit quantitativen Werten zum Ausdruck bringen, sodass alle Beteiligten wissen, was auf sie zukommt. Um dies umzusetzen könnte man beispielsweise vor dem eigentlichen Refactoring gemeinsam als Team abschätzen, wie viel Zeit welcher Refactoringschritt benötigt und somit dann als Gruppe entscheiden, ob es sich lohnt. Zum anderen könnte es dabei helfen, unterschiedliche Einschätzungen bewusst zu machen und somit Missverständnisse zu klären, die eventuell zu den jeweiligen Entscheidungen geführt haben. Zusätzlich könnte man Personen darauf schulen oder trainieren, bessere Schätzungen über Aufwand und Zeit hinzubekommen. Ich möchte hierbei klarstellen, dass ich in den über 250 Issues, die ich gelesen habe, keinen einzigen gefunden habe, indem tatsächlich mit quantitativen Werten der Aufwand abgeschätzt wurde, sodass meine Annahme, dass das Abschätzen die Entscheidungen beeinflussen könnte, auf meinem logischen Schlussfolgern beruht und nicht gegroundet ist.

4.4.2 Schulung zu dem Herauskristallisieren von Vorteilen des Refactorings

Das zweite große Problem, das bei der Analyse zum Vorschein gekommen ist, ist die Tatsache, dass das Refactoring mehrheitlich als nicht wertschöpfend betrachtet wird. Hierzu könnte man als Team Schulungen und Vorträge besuchen, um ein allgemeines Bewusstsein für dieses Thema zu schaffen. Carola Lilienthal hält beispielsweise seit Jahren Vorträge über technische Schulden und deren Folgen, falls man sie nicht beseitigt (Refactoring). Außerdem könnte man darüber nachdenken, wie man das Refactoring generell in den Augen der Menschen attraktiver machen könnte. Beispielsweise durch Zusatzleistungen oder Zeit, die von vornherein für dieses Prozedere eingeplant wird. Dieses Problem ist jedoch sehr tief verankert und beginnt laut Lilienthal in der Managementebene.

Abbildung 4: Baum mit Verwurzelungen als Metapher für Verzweigungen von sonst funktionierenden Code [15]

In einem ihrer Vorträge zeigt die Softwarearchitektin ein Bild von einem Baum, der von oben grün und schön scheint, aber unter der Erde stark verzweigt und verwurzelt ist. Sie kommentiert es in dem Vortrag mit den Wörtern „Das Management, also das guckt ja von oben und von oben ist ja alles schön grün.“ [15]. Hierbei nimmt sie die Position der Managementebene ein, welche lediglich das Endprodukt und somit die schönen grünen Blätter sieht. Den Code und vor allem die ganzen Verzweigungen, die mit dem Wachsen des Codes immer mehr werden, sehen diese Personen nicht. Dass das ganze Projekt immer komplexer und somit unsauberer wird, erkennen lediglich die Personen, die an dem Code arbeiten oder ihn reviewen. Dies hat zur Folge, dass bei der Managementebene kein Bewusstsein und somit kein Verständnis herrscht, wenn es darum geht, mehr Zeit zu bekommen, um den geschriebenen Code zu refactoren. Noch weiter ausgeholt führt der komplexer werdende Code dazu, dass das Programmieren durch seine Komplexität länger dauert und somit Zeitdruck entsteht, weshalb viele Programmierer:innen dazu neigen, einen noch unsaubereren Code zu

schreiben und noch weniger zu kommentieren, wodurch der Code noch komplexer zu verstehen wird. Das Ganze ist also ein Teufelskreis, aus dem man nur aussteigen kann, wenn man ein Bewusstsein dafür schafft, wie wichtig Refactoring ist und sich anschließend die Zeit nimmt, dieses Refactoring zu betreiben.

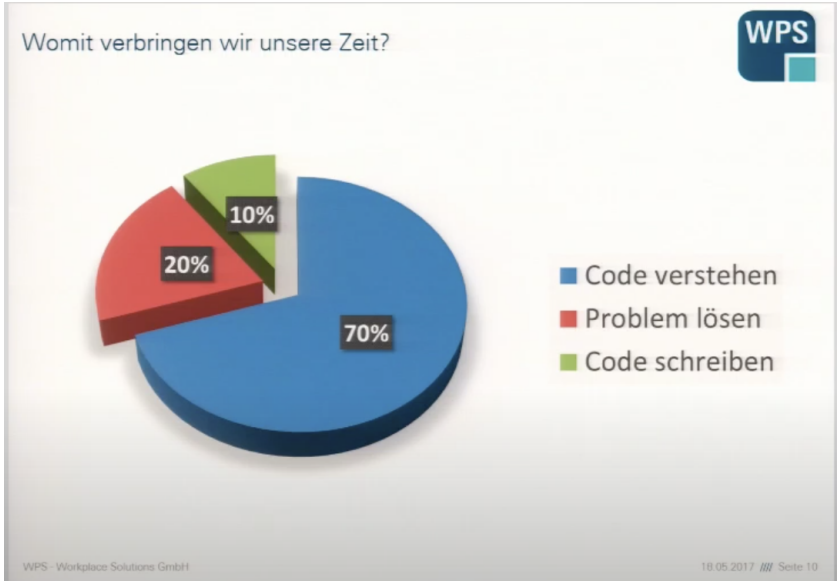

Abbildung 5: Zeitverteilung bezüglich der Arbeit mit Code [15]

Dieses Bild stammt ebenfalls aus diesem Vortrag von Lilienthal [15] und verdeutlicht die Wichtigkeit des Refactorings. Betrachten wir das Chart erkennen wir, dass der größte Teil der Zeit in „Code verstehen“ fließt. Lediglich 20 Prozent der Zeit verbringen Programmierer:innen mit dem Erarbeiten einer Lösung und zehn Prozent mit dem tatsächlichen Schreiben des Codes. Hierbei wird deutlich, dass der einzige Punkt, an dem wir Zeit sparen sollten, das Verstehen von Code ist. Eine Sache, die wir durch das Refactoring erreichen können und sollten.

5 Diskussion

5.1 Reflexion der genutzten Quelle

In diesem Kapitel würde ich gerne reflektieren, inwiefern sich die Arbeit mit dem GitHub Issue Tracker ausgezahlt hat und welche Limitierungen mir gesetzt wurden. Zunächst einmal einige Fakten: Ich habe in meiner gesamten Recherche über 250 GitHub Issues durchgelesen, von denen ich ungefähr 150 direkt beim Lesen verworfen habe. Die Gründe hierfür habe ich in dem Unterkapitel „verworfene Issues“ (3.2.4) bereits genannt. Die restlichen 102 Issues habe ich herausgezogen und nochmal genauer durchgelesen. Am Ende habe ich sieben Issues enger in meiner Analyse einbezogen. Hätte ich vorher stoppen können? Natürlich, aber dann hätte ich diese sieben wahrscheinlich nicht gefunden. Denn obwohl ich nur sieben Stück gefunden habe, habe ich dennoch konsistent Issues gefunden. In 30 betrachteten Issues war ein hilfreicher drin, sodass ich die Suche nicht aufgeben wollte und immer weitere Blocks an Issues analysiert habe. Der Großteil der Diskussionen auf GitHub sind kurze Texte, sodass das erste Durchgucken und Aussortieren nicht lange gedauert hat. Hierfür eignet sich GitHub besser als beispielsweise Forschungsliteratur, die teilweise über 20 Seiten lang ist. Der größte Vorteil, den ich durch das Nutzen der GitHub-Issues hatte, war die Tatsache, dass die Aussagen der Personen, die ich analysierte, in einem echten Umfeld stattfanden und nicht in einem erzeugten. Die Diskutierenden wussten nicht, dass ich sie analysieren würde, weshalb sie sich nicht verstellt haben, wie es bei einem Interview hätte sein können. Außerdem neigen Personen dazu, in Diskussionen emotional zu werden und sich somit von diesen Emotionen geleitet zu äußern, was die Authentizität der Aussagen bestärkt. Ein weiterer Punkt ist, dass die Personen nicht meinerseits beeinflusst wurden und nicht durch meine Fragen beispielsweise gelenkt wurden, sondern das gesagt haben, was sie in dem Moment äußern wollten. Dies alles führte dazu, dass ich davon ausgehen kann, dass die Aussagen, die diese Personen tätigen und die Meinungen, die sie vertreten, tatsächlich ihre sind.

5.2 Limitierungen

Natürlich bringt das Nutzen des GitHub Issue Trackers als Quelle für meine Analyse auch Nachteile mit sich. Nachteile, die mir bestimmte Limitierungen gesetzt haben.

5.2.1 Anzahl Material

Durch die bedingte Zeit konnte ich nur eine bestimmte Anzahl an Issues durcharbeiten, sodass ich die theoretische Sättigung nicht erreichen konnte und mehr Material gebraucht hätte.

5.2.2 Verständnis des Materials & der Gedankengänge

Außerdem war ich limitiert, was die Tiefe des vorhandenen Materials angeht. Durch das Arbeiten mit dem GitHub Issue Tracker war ich selbst kein Teil der Diskussion, sondern habe lediglich diese gelesen. Dadurch hatte ich nur den Inhalt dieser Issues

zu Verfügung und konnte diesen nicht durch zusätzliches Nachfragen erweitern, wie es bei Interviews beispielsweise der Fall gewesen wäre.

5.2.3 Interpretationsspielraum

Dies führt unweigerlich dazu, dass meine eigene Einschätzung und Interpretation der Aussagen einen stärkeren Einfluss auf meine Wahrnehmung und Ergebnisse hatten. Daraus ergibt sich eine weitere Schwäche der Nutzung des Github Issue Trackers im Vergleich zu Interviews.

5.2.4 Kontakt zu Diskutierenden

Ich konnte im Gegensatz zu Interviews nicht nachfragen, falls ich etwas nicht genau verstanden habe. Außerdem war es zu keinem Zeitpunkt möglich auf Antworten hinzuweisen, die mir zu ungenau erscheinen, wie beispielsweise das in Kapitel vier genannte Problem mit den nicht quantitativen Aussagen, um ausführlichere Antworten zu erhalten. Hinzukam, dass ich dem Gegenüber meine Auffassungen über das Gesagte nicht äußern und mich somit auch nicht absichern konnte, das Ganze richtig verstanden zu haben. All dies ist als stiller Leser der GitHub Issues nicht möglich. Hierfür könnte man versuchen, die einzelnen Akteure der jeweiligen Issues zu erreichen und zu befragen, aber das hätte meinen zeitlichen Rahmen gesprengt.

5.2.5 Zeit

Wenn auch nicht abhängig von der Forschungsmethode selbst, sondern von der 12-Wochen-Dauer einer Bachelorarbeit, war ich limitiert, was die Zeit angeht, sodass meine Forschung nicht die erwünschte Tiefe widerspiegelt.

5.3 Ausblick

Rückblickend lässt sich sagen, dass es ab diesem Zeitpunkt nicht mehr sinnvoll gewesen wäre, mit dieser Methode weiterzuforschen. Es ist während der Analyse oft aufgefallen, dass die Diskutierenden ein unterschiedliches Verständnis von Refactoring gezeigt haben. Es gab Issues, in denen als Bedenken genannt wurde, dass sich die Benutzeroberfläche für die Nutzer:innen ändern könnte, aber genau das passiert bei dem Refactoring nicht. Bei dem Refactoring wird die innere Struktur verändert. Die Funktionsweise bleibt jedoch gleich. In anderen Issues kam es zu Missverständnissen, weil eine Person unter Refactoring das komplette Neuschreiben des Codes verstand, während die andere Person lediglich von Optimierungen sprach. All diese Umstände zeigen, dass man zunächst verstehen sollte, was Menschen genau unter Refactoring verstehen, ehe man die Motivation für das Refactoring untersucht. Man kann nicht versuchen zu verstehen, was Menschen motiviert „etwas“ zu machen, wenn diese unterschiedliche Auffassungen darüber haben, was dieses „etwas“ ist. Hierfür kann man beispielsweise nach Definitionen in der Forschungsliteratur suchen, um abschätzen zu können, wie zerrieben der Begriff des Refactorings in der Forschung selbst ist. Dies würde bereits Einblicke in den Ursprung des Problems geben. Man könnte dadurch unterscheiden, ob das Problem in der zur Verfügung stehenden

Literatur oder in den Köpfen der Menschen besteht. Ersteres wäre ein grundlegendes Problem in der Literatur und sollte weiter untersucht werden. Wenn man der Meinung ist, dass das Problem bei dem Verständnis der Menschen selbst und nicht in der Literatur liegt, könnte man in unterschiedlichen Softwareunternehmen Interviews mit Programmierer:innen führen und dabei diese nach ihrem Verständnis von Refactoring befragen. Zum einen, wie sie Refactoring definieren und zum anderen, wie sie sich dieses Wissen angeeignet haben. Eine falsche Definition von Refactoring, könnte entscheidend sein für die Bereitschaft, Refactoring zu betreiben, wie weiter oben bereits angeschnitten. Durch unterschiedliche Definitionen könnten Menschen den Aufwand beispielsweise unterschiedlich hoch einschätzen und somit ein Refactoring vermeiden wollen. Die Frage nach der Definition scheint also grundlegend für die Einschätzung der Bereitschaft Refactoring zu betreiben zu sein. Die Nachweislage für diese Annahme ist dünn, aber bereits die unterschiedlich wirkenden Annahmen über den Aufwand in Erkenntnis 1 (Kapitel 4.1.1) geben erste Hinweise darauf. Dies wäre bei meiner zeitlichen Begrenzung jedoch nicht sinnvoll gewesen. Mit mehr Zeit hätte ich außerdem den Kontakt zu den Diskutierenden aufgesucht, um meine Fragen, die aktuell in Memos in MAXQDA stehen, klären zu können. Dies würde den Interpretationsspielraum der Aussagen verringern. Hierfür würde ich kein richtiges Interview führen, sondern lediglich anhand eines Frageprotokolls meine Fragen klären wollen. Sollten sich daraus weitere Fragen ergeben und sich die Person als kommunikativ erweisen, könnte der Austausch mit einem Interview vertieft werden.

Während meiner Analyse sind noch weitere Fragen aufgetaucht, die ich aufgrund der zeitlichen Beschränkung nicht beantworten konnte, aber gerne näher untersucht hätte. Diese waren unter anderem:

1. Ist das Verzichten auf Refactoring bei sogenanntem Wegwerfcode gerechtfertigt?
2. Haben die Personen tatsächlich keine Zeit oder wollen sie sich keine Zeit nehmen?
3. Wären diese Personen zu einem anderen Zeitpunkt mit dem Refactoring einverstanden?
4. Entscheidet die Größe des Projektes über die Bereitschaft der Personen in Bezug auf das Refactoring?

6 Zusammenfassung

Diese Bachelorarbeit begann mit dem Wunsch herauszufinden, wieso Informatiker:innen bereit sind technische Schulden einzugehen und entwickelte sich zu der Forschungsfrage „Welche Faktoren des mentalen Modells von Entwickler:innen beeinflussen ihre Bereitschaft Refactoring zu betreiben?“ Für die Analyse wird mit einem GitHub Issue Tracker gearbeitet und für die Datenanalyse die Grounded Theory Methodology verwendet. Am Ende gelangt man bei der Beantwortung dieser Frage zu mehreren Faktoren, die entscheidend sind, wenn es um die Frage der Bereitschaft in Bezug auf das Refactoring geht. Zum einen legen die Diskutierenden einen hohen Wert auf den Grad des Aufwandes, ohne diesen zu irgendeinem Zeitpunkt mit quantitativen Aussagen abzuschätzen. Dies kann zu Missverständnissen und „aneinander vorbeireden“ führen. Zum anderen scheinen Kriterien, wie Ausgangssituation, Lust und die Haltung gegenüber Refactoring generell eine große Rolle zu spielen. Bei der Frage nach der Ausgangssituation fällt auf, dass sich die Einstellung gegenüber dem Refactoring ändert, wenn Personen allein und nicht in Teams an Projekten arbeiten. Als Freelancer beispielsweise scheint dies keine Option zu sein. Die Lust hängt laut den Diskutierenden mit der Frage der Bezahlung zusammen. Sie sind nicht bereit Refactoring zu betreiben, wenn Sie das Refactoring nicht als Teil ihrer entlohnten Arbeit oder Aufgaben wahrnehmen. Die Haltung gegenüber dem Refactoring ist rein subjektiv und von Person zu Person unterschiedlich, hängt jedoch auch von der herrschenden Ausgangssituation ab. Die letzte Erkenntnis, die auffällt, ist, dass die Diskutierenden den Nutzen des Refactorings nicht nur hinterfragen, sondern von Grund auf häufig keinen Nutzen durch das Refactoring sehen, weshalb sie dazu neigen dieses nicht durchzuführen. Um dem Ganzen entgegenzuwirken, sollten Teams von vorneherein die einzelnen Schritte des Refactorings mit quantitativen Werten abstimmen, um einen konkreten Eindruck von dem tatsächlichen Aufwand zu bekommen. Damit können Missverständnisse aus dem Weg geräumt werden, da alle über denselben Aufwand reden. Eine andere Handlungsempfehlung ist das Herauskristallisieren der Relevanz des Refactorings, da es mehrheitlich immer noch als nicht wertschöpfend wahrgenommen wird.

Literaturverzeichnis

[1] M.Fowler, *Refactoring: Improving the Design of Existing Code*. Boston, Mass. [u.a.] : Addison-Wesley, 2005

[2] R.Gh.Alsarraj und A.M.Altaie, *Refactoring for software maintenance: A Review of the literature*. In: *Journal of Education and Science*, Mosul University, 2021 (Vol 30), pp.89-102

[3] V.Lenarduzzia, T.Beskerb, D.Taibic, A.Martinid und F.A.Fontanae, *Technical Debt Prioritization: State of the Art. A Systematic Literature Review*. 2019.

[4] U.Flick, *The SAGE Handbook of Qualitative Data Analysis*. SAGE Publications Ltd, 2013

[5] A.L.Strauss; J.Corbin: *Grounded Theory: Grundlagen Qualitativer Sozialforschung*. BELTZ Psychologie Verlags Union, 1996

[6] U.Kuckartz, *Einführung in die computergestützte Analyse qualitativer Daten*. Springer Nature Switzerland AG, 2010

[7] Carola Lilienthal, *Langlebige Software-Architekturen : technische Schulden analysieren, begrenzen und abbauen*. dpunkt.verlag GmbH, 2016

[8] GitHub. (30.04.2020 - 02.06.2020). *Unit Tests - Mocking concepts #764*. [Online]. Verfügbar: https://github.com/zowe/vscode-extension-for-zowe/issues/764

[9] GitHub. (14.04.2017 - 23.02.2019). *Discosal: Null-conditional return #437*. [Online]. Verfügbar: https://github.com/dotnet/csharplang/issues/437

[10] GitHub. (14.06.2015 - 15.07.2015). *Code duplication between different Unixes #14715*. [Online]. Verfügbar: https://github.com/dotnet/runtime/issues/14715

[11] GitHub. (23.03.2018 - 25.03.2018). *[RFC]: rewrite in typescript and restructure to monorepo #3284*. [Online]. Verfügbar: https://github.com/d3/d3/issues/3284

[12] GitHub. (30.06.2017 - 07.08.2017). *Changing guidelines on Documentation #195*. [Online]. Verfügbar: https://github.com/10up/Engineering-Best-Practices/issues/195

[13] GitHub. (01.04.2017 - 03.04.2017). *Refactor NamedTuple to put it in a stub #3102* [Online]. Verfügbar: https://github.com/python/mypy/issues/3102

[14] GitHub. (26.08.2018 - 30.11.2019). *Level of Review Necessary for PRs #1321*. [Online]. Verfügbar: https://github.com/stan-dev/math/issues/1321

[15] rheinjug. "Carola Lilienthal - Langlebige Softwarearchitekturen – der Weg aus den technischen Schulden" *YouTube*, 24.07.2017 [Videofile]. Verfügbar: https://www.youtube.com/watch?v=Nfzuy4L9_UM [[Accessed: 09.02.2022].]