Master in Informatik

Umgang mit *technischen Schulden* in Großunternehmen

Maic Siemering
Matrikelnummer: 2529980

Erst Prüferin: Dr. Daniela Keller
Zweit Prüfer: Prof. Dr. Lutz Prechelt
Betreuer: Linus Ververs
Fachbereich: Informatik
Lehrgebiet: Programmiersysteme

Kurzfassung

Technische Schulden sind ein wesentlicher Aspekt in der Softwareentwicklung und machen einen beachtlichen Anteil der Entwicklungszeit aus. Diese Forschungsarbeit konzentriert sich auf die interne Softwareentwicklung in einem deutschen Großunternehmen. Neben Interviews wurden vor allem Beobachtungen aus dem Alltag von Softwareentwicklern genutzt, um eine fundierte Theorie über den Umgang mit technischen Schulden zu bilden.

Die Ergebnisse weisen darauf hin, dass Entscheidungen, Kosten- und Nutzenabschätzungen bezüglich technischer Schulden zumeist auf Bauchgefühlen basieren. Metriken werden nur in den seltensten Fällen verwendet. Spikes bzw. Prototypen können helfen, Nutzen und Aufwand einer Umsetzung abzuschätzen. Diese bergen jedoch das Risiko, bei der direkten Übernahme in produktive Codebasen ein hohes Maß an technischen Schulden aufzunehmen.

Inhaltsverzeichnis

1 Einleitung **1**
1.1 Motivation . . . 1
1.2 Ziel der Arbeit . . . 1
1.3 Überblick . . . 2

2 Technische Schulden **3**
2.1 Die Metapher . . . 3
2.2 Herleitung . . . 3
2.3 Definition . . . 6
2.4 Entstehung . . . 8
2.5 Auswirkung . . . 9
2.6 Umgang . . . 9
2.7 Arten . . . 11

3 Untersuchtes Umfeld **14**
3.1 Untersuchtes Unternehmen . . . 14
3.2 Organisation . . . 15
3.3 Teams . . . 15
3.4 Besprechungen . . . 16

4 Verwandte Arbeiten **18**
4.1 Forschung in Großunternehmen . . . 18
4.2 Forschung in anderen Umgebungen . . . 30
4.3 Geschäftsgetriebene Priorisierung . . . 34
4.4 InsighTD . . . 35
4.5 Forschung in Startups . . . 36
4.6 Weitere Forschung . . . 37
4.7 Schwerpunkte der gefundenen Arbeiten und Lücken in der Forschung 37

5 Methodik **39**
5.1 Ziel der Forschung . . . 40
5.2 Vorgehen . . . 40
5.3 Grounded Theory Methodology . . . 40

6 Erhobene Daten **45**
6.1 Datenquellen . . . 45
6.2 Untersuchte Teams . . . 50

7 Ergebnisse **56**
7.1 Bewusstsein in der Organisation 56
7.2 Begriff „*technische Schulden*" in Softwareentwicklungsteams 57
7.3 Identifikation 59
7.4 Spikes: Vor- und Nachteile 71
7.5 Unzufriedenheit in Folge *technischer Schulden* 76
7.6 Dokumentation 79
7.7 Priorisierung und Tilgung 86

8 Diskussion **87**
8.1 Abgleich mit Ergebnissen aus „An enterprise perspective on technical debt" 87
8.2 Abgleich mit Ergebnissen aus „Umgang mit technischen Schulden im Startup-Umfeld" 95
8.3 Diskussion der Ergebnisse 105
8.4 Abgeleitete Empfehlungen/Grounded Theory 114

9 Gültigkeit **118**
9.1 Interne Validität 118
9.2 Externe Validität 118

10 Zusammenfassung/Fazit/Ausblick **120**

Danksagung

An dieser Stelle möchte ich mich bei meiner Frau für ihren selbstlosen Einsatz bedanken. Du hast über viele Wochen das Familienleben auf deine Schultern genommen und mir all die Zeit freigeräumt, die ich brauchte, um diese Arbeit fertigzustellen.

Ich danke dir!

Gender-Hinweis

Die in dieser Arbeit verwendeten Personenbezeichnungen beziehen sich immer gleichermaßen auf weibliche und männliche Personen. Auf eine Doppelnennung und gegenderte Bezeichnungen wird zugunsten einer besseren Lesbarkeit verzichtet.

1 Einleitung

1.1 Motivation

Die tägliche Priorisierung der richtigen Aufgaben ist entscheidend für den Erfolg eines Softwareentwicklungsteams. Dabei richtet sich die moderne Softwareentwicklung immer häufiger an agilen Methoden aus, um den sich stetig wechselnden Anforderungen der schnelllebigen digitalen Welt gerecht zu werden. Umsetzungen, die zunächst ihrem Zweck dienlich erschienen, veralten oder stellen sich angesichts neuer Anforderungen als suboptimal heraus. In dieser volatilen Umgebung ist der Umgang mit *technischen Schulden* ein wesentlicher Bestandteil der Softwareentwicklung und ein wichtiges Kriterium für die Priorisierung von Aufgaben.

Bestehende Forschungen nutzen zumeist Interviews mit Softwarepraktikern oder öffentliche Umfragen, um Daten über den Umgang mit *technischen Schulden* zu sammeln. Phänomene aus dem Alltag von Entwicklern sind nur in Einzelfällen zu finden, somit ist die Gewinnung dieser selten vorhandenen Daten ein großer Mehrwert für die Forschung.

Zudem ist der Einfluss des organisatorischen Umfeldes von Großunternehmen auf den Umgang mit *technischen Schulden* besonders interessant. Es ermöglicht Rückschlüsse, ob Empfehlungen für den Umgang mit technischen Schulden aus anderen Umgebungen (z.B. Open Source Entwicklung oder Startups) übernommen werden können.

Großunternehmen besitzen zumeist die Ressourcen, um umfangreiche Softwareorganisationen aufzubauen und eine beträchtliche Menge an Praktikern und Wissen zu bündeln. Demzufolge birgt die Forschung in einem Großunternehmen die Möglichkeit das Verhalten von diversen Softwareentwicklungsteams zu beobachten.

1.2 Ziel der Arbeit

Das Ziel der Arbeit ist es, anhand eines explorativen Ansatzes die Praxis im Umgang mit technischen Schulden in Großunternehmen zu untersuchen. Um dies zu erreichen, soll der Entwickleralltag als Datenquelle dienen. Dies soll einen tieferen Einblick in den Umgang mit technischen Schulden gewähren, als Umfragen oder Interviews ermöglichen, die vorwiegend in der existierenden Literatur verwendet werden.

Diese Beobachtungen sollen anschließend über einen längeren Zeitraum ausgewertet und qualitativ analysiert werden. Dadurch sollen Phänomene zum Umgang mit technischen Schulden identifiziert und die Ergebnisse mit der bisherigen Forschung abgeglichen werden.

1.3 Überblick

In den Kapiteln 2, 3 und 4 werden die Grundlagen erläutert, welche im thematischen Kontext der Arbeit existieren und wichtig für den allgemeinen Forschungsrahmen sind. Dabei erörtert Kapitel 2 den Begriff der *„technischen Schulden"*. Im Einzelnen wird auf die Metapher an sich, die Herleitung, Definition, Entstehung, Auswirkung, Umgang und die verschiedenen Arten von *technischen Schulden* eingegangen. Zentraler Bestandteil ist die eigene Definition des Begriffs der *„technischen Schulden"*, welcher in dieser Forschungsarbeit verwendet wird. Kapitel 3 beschreibt das untersuchte Umfeld, welches dieser Forschungsarbeit zugrunde liegt. Hierzu gehören eine kurze Beschreibung des Unternehmens, der Unternehmensorganisation, der untersuchten Softwareentwicklungsteams und etablierten Besprechungen. Verwandte Arbeiten, welche den derzeitigen Forschungsstand über den Umgang mit technischen Schulden in ähnlichen thematischen Kontexten darstellen, werden in Kapitel 4 zusammengefasst.

Anschließend wird in den Kapiteln 5 und 6 die Auswertungsmethode und erhobene Forschungsdaten dieser Forschungsarbeit erläutert. Es wird detailliert darauf eingegangen, welche Daten und Beobachtungen aus dem Umfeld des Großunternehmens erhoben und analysiert werden konnten.

Darauf werden die Ergebnisse vorgestellt, diskutiert und mit existierenden Forschungsergebnissen abgeglichen. Das Kapitel 7 ist anhand der gefundenen Konzepte unterteilt. Kapitel 8 gleicht die Ergebnisse von zwei verwandten Arbeiten anhand der eigenen Beobachtungen ab, diskutiert die Schlussfolgerungen der eigenen Ergebnisse und fügt diese zu einer fundierten Theorie zusammen.

Den Schluss bilden die Kapitel 9 und 10 in denen über die Gültigkeit der Forschungsergebnisse reflektiert und anschließend diese zusammengefasst werden.

2 Technische Schulden

2.1 Die Metapher

Der Begriff technische Schulden oder auch „Technical Debt" ist eine aus der Finanzwelt entlehnte Metapher, welche von Ward Cunningham im Jahr 1992 erstmals verwendet wurde.

> „A little debt speeds development so long as it is paid back promptly with a rewrite. Objects make the cost of this transaction tolerable." [25]

Die Metapher diente der Kommunikation zwischen den Entwicklern und dem Vorgesetzten. Cunningham erläuterte in einem Video-Interview den genaueren Kontext [133]:

Das Ziel der Metapher war es, dem Vorgesetzten in einer fachnahen Sprache, und somit für ihn leicht verständlich, zu erklären, warum das Team Zeit darauf verwendet, den Code von bestehenden Funktionen nachträglich zu überarbeiten, ohne die Funktionalität selbst zu ändern („Refactoring"). Aus Cunninghams Sicht war es eine gute Idee, Schulden aufzunehmen und die Software möglichst schnell Kunden bereitzustellen. Dies birgt die Möglichkeit zu lernen und nachträglich Anpassungen vorzunehmen, so dass die Software letztendlich so aussieht, als ob sie von vornherein schon gewusst hätten, „was sie tun". [133]

Falls sie jedoch diese Anpassungen nicht machen würden, würden sie Zinsen zahlen, bis zu dem Punkt, dass kein Fortschritt mehr möglich wäre. [122] Unter Zinsen wird ein Mehraufwand an Entwicklungsarbeit verstanden, der durch die aufgenommenen Schulden verursacht wird und nicht anfallen würde, würden diese vorher getilgt werden.

Cunningham weist strikt darauf hin, dass im Verlauf der Zeit seine Metapher, unter anderem von Bloggern, falsch verstanden wurde [133]. Es ginge niemals darum, schlechteren Code wider besseren Wissens zu schreiben, um diesen dann später aufzuräumen.

Eine gute Methode sei, Code zu schreiben, der sauber genug ist, um ihn später mit dem neuen fachlichen und technischen Verständnis anpassen zu können [133].

2.2 Herleitung

Bereits 1980 drückte Lehman, in seinem zweiten Gesetz der Programm Evolution, aus, dass bei fortschreitender Entwicklung einer Software die Komplexität steigt. Dagegen müsse aktiv gearbeitet werden.

> „As an evolving program is continually changed, its complexity, reflecting deteriorating structure, increases unless work is done to maintain or reduce it.” [70]

Diese Aussage unterstützt die Metapher der technischen Schulden dahingehend, dass während der Entwicklung die Software dauerhaften Veränderungen ausgesetzt ist, was sich in steigender Komplexität niederschlägt. Dies ist analog zu technischen Schulden, die aufgenommen werden und ohne aktive Tilgung nicht weniger werden.

Die Metapher wurde für lange Zeit nicht in der Forschung verfolgt, jedoch 2008 durch McConnell aufgegriffen [84]. Er leitete aus der Metapher zwei Arten von technischen Schulden ab, unbeabsichtigte (Type I) und beabsichtigte (Type II) technische Schulden.

Nach seiner Beschreibung kann es dazu kommen, dass erst im Nachhinein auffällt, dass zum Beispiel ein Design fehleranfällig ist und damit ungünstig für die derzeitige Implementierung. Dies seien unbeabsichtigte technische Schulden.

Wissentlich getroffene Entscheidungen für ein schnelleres Veröffentlichen zählt zu beabsichtigten technischen Schulden. Im weiteren Verlauf seines Whitepaper fokussiert er sich allein auf Empfehlungen für beabsichtigte technische Schulden, unbeabsichtigte lässt er außen vor.

Angeregt durch die Aufmerksamkeit in Blogs bezüglich der Metapher der technischen Schuld wirft M. Fowler die Frage in seinem Blog auf, ob es wichtig ist, was eine technische Schuld ist und was nicht. Die Metapher sei dazu gedacht, die Kommunikation mit nicht-technischen Personen zu erleichtern. Solange die nicht-technische Person die Metapher versteht und sie der Kommunikation dienlich ist, kann sie verwendet werden.

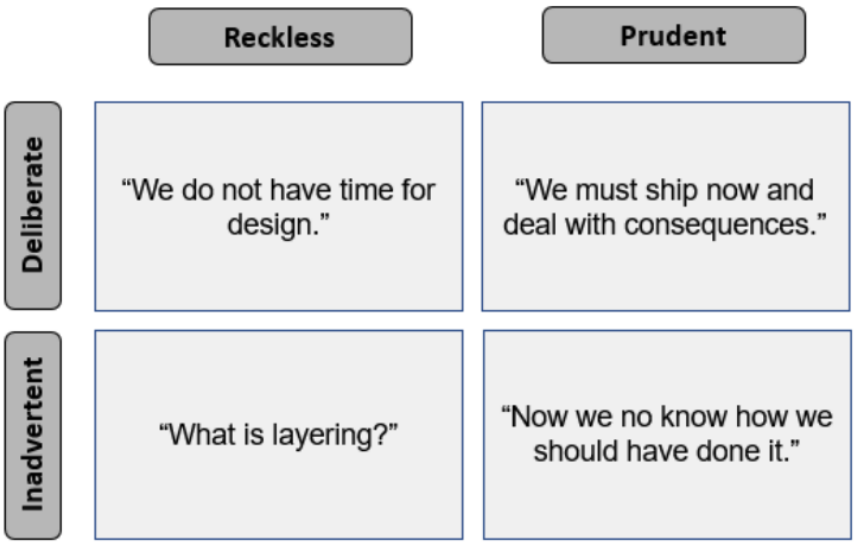

Abbildung 1: Vier Quadranten nach M. Fowler [117] adaptiert von [35]

Er teilt zudem technische Schulden in vier Quadranten ein, welche abgeleitet sind anhand der Dimensionen leichtsinnig („Reckless")/ vorsichtig („Prudent") und bewusster („Deliberate")/ unbewusster („Inadvertent") Aufnahme von technischen Schulden [35].

Aus diesen entstehen vier Quadranten, welche er mit Aussagen verdeutlicht:

leichtsinnig/bewusst „Wir haben keine Zeit für Design"

vorsichtig/bewusst „Wir müssen jetzt liefern und mit den Konsequenzen umgehen"

leichtsinnig/unbewusst „Was sind Schichten?"

vorsichtig/unbewusst „Nun wissen wir, wie wir es hätten machen sollen"

Durch das gesteigerte Interesse der Industrie am Umgang mit technischen Schulden erarbeiten Kruchten et al. 2013 eine Annäherung an eine Definition, was technische Schulden sind und leiten zudem Forschungsfragen ab [67].

In ihrer Ausarbeitung beschreiben sie technische Schulden als abstraktes Konzept, welches besonders in der Kommunikation zwischen technischen Personen, welche „Short time to market" nicht schätzen, und Business Personen, welche nicht immer ein Verständnis für frühe Design-Entscheidungen und deren langfristige Folgen haben [67].

Zudem gehen sie darauf ein, welche Vorkommnisse nicht unter dem Konzept verstanden werden sollten, um es auf direkte Systemeigenschaften zu begrenzen. Aus der wissenschaftlichen Sichtweise ist eine klare Definition eines Phänomens essentiell,

um dieses genauer zu untersuchen. Zudem verliert der Begriff technische Schulden an Bedeutung, desto unklarer er umrissen wird, da besonders spezifische Eigenschaften kaum erkennbar sind.

Die Agile Alliance veröffentlichte 2016 eine von Cunningham überprüfte Einleitung für das Konzept der technischen Schulden [74] samt Handlungsempfehlungen [73]. Als Zusammenfassung der Metapher wird folgende Beschreibung angeführt:

> „When taking short cuts and delivering code that is not quite right for the programming task of the moment, a development team incurs Technical Debt. This debt decreases productivity. This loss of productivity is the interest of the Technical Debt." [74]

Diese Spezifikation der Metapher beinhaltet keine unbeabsichtigten technischen Schulden, da explizit das Nehmen von Abkürzungen als Grund für technische Schulden genannt ist. Unklar ist jedoch, ob unter dieser Zusammenfassung auch später zu bereinigender schlechter Code wider besseren Wissens zählt, der von Cunningham in einem Interview eigentlich explizit ausgeschlossen wird [133].

Zur gleichen Zeit wird im Dagstuhl Seminar 16162 aus dem abstrakten Konzept eine klare Definition erarbeitet. Ein wichtiger Hinweis in dem veröffentlichten Bericht ist, dass die Gemeinschaft unter dem Begriff technische Schulden neben der Metapher für die Kommunikation auch ein Artefakt der Softwareentwicklung sieht. Diese Mehrdeutigkeit führt zu Verwirrung und birgt das Risiko, alles Schädliche für Softwaresysteme und den Entwicklungsprozess als technische Schulden zu bezeichnen. [7]

2.3 Definition

Im Dagstuhl Seminar 16162 wird eine klare Definition für den Begriff der technischen Schulden als Artefakt der Softwareentwicklung erarbeitet.

> „In software-intensive systems, technical debt consists of design or implementation constructs that are expedient in the short term, but set up a technical context that can make a future change more costly or impossible. Technical debt is a contingent liability whose impact is limited to internal system qualities, primarily maintainability and evolvability." [7]

Entgegen der Metapher schließt diese Definition sowohl bewusste als auch unbewusste technische Schulden mit ein. Systeme können auch ohne das Wissen der

Entwickler Design- oder Implementierungskontexte beinhalten, die Änderungen kostspieliger oder unmöglich machen.

Diese Definition soll als Basis für die weitere Verwendung des Begriffes der technischen Schulden dienen.

Zu bemerken ist, dass der Argumentation der verwandten Arbeit „Umgang mit technischen Schulden im Startup-Umfeld" [129] gefolgt wird und das „can" durch ein „are likely" zu ersetzen ist, um die höheren Entwicklungskosten stärker in der Definition zu verankern.

Darüber hinaus wird dazu plädiert, die explizite Limitierung auf interne Systemqualitäten mit einem „centered around" abzuschwächen. Dies erlaubt im Besonderen Randfälle technischer Schulden zu inkludieren, die auch einen Effekt auf externe Systemqualität haben, jedoch die gleiche Entscheidung über Tilgung oder Aufnahme mit sich bringen. Ein Beispiel ist eine aufgeschobene Überarbeitung des API Design, welche einerseits dazu führt, dass jede Entwicklung vor der Überarbeitung zusätzlichen Aufwand erzeugt, darüber hinaus jedoch auch Einfluss auf die externe Systemqualität in Form einer für den Kunden ungünstigen API haben kann. Eine Alternative wäre die Streichung des Satzes, wie es zum Beispiel in den Präsentationen von Ipek Ozkaya[97] der Fall ist.

An dieser Stelle soll zudem explizit darauf hingewiesen werden, dass technische Schulden zwar unbewusst als Design- oder Implementierungskonstrukte in Systemen existieren können, sie jedoch zu bewussten technischen Schulden werden, sobald sie entdeckt werden. Somit werden unbewusste technische Schulden nach der Definition in dem Moment zu bewussten technischen Schulden, sobald sie auffallen und entschieden wird, sie trotz ihrer wahrscheinlichen mittel- bis langfristigen negativen Folgen nicht zu tilgen.

Die für diese Arbeit verwendete Definition von technischen Schulden wird aus der angepassten englischen Variante abgeleitet. Da ein großer Anteil der verwendeten Quellen in englischer Sprache verfasst sind und untersuchte Teams unter anderem auch Englisch sprechen, werden beide Sprachen aufgeführt:

> In software-intensive systems, technical debt consists of design or implementation constructs that are expedient in the short term, but set up a technical context that are likely to make a future change more costly or impossible. Technical debt is a contingent liability whose impact is centered around internal system qualities, primarily maintainability and evolvability.

> In software-intensiven Systemen bestehen technische Schulden aus Entwurfs- oder Implementierungskonstruktionen, die kurzfristig zweckmäßig sind, aber einen technischen Kontext schaffen, der eine künftige Änderung wahrscheinlich kostspieliger oder unmöglich macht. Technische Schulden sind eine Eventualverbindlichkeit[1], deren Auswirkungen sich auf die internen Systemeigenschaften, vor allem auf die Wartbarkeit und die Weiterentwicklungsfähigkeit, konzentrieren.

In dieser Arbeit wird der Begriff „technische Schulden" immer dann kursiv dargestellt, wenn technische Schulden nach dieser Definition gemeint sind.

Wichtig zu bemerken ist, dass die gewählten Anpassungen die Definition von technischen Schulden eher ausweiten. Beispiele und Erkenntnisse aus Quellen, die sich auf die originale Definition beziehen, sollten somit im Kontext der eigenen Definition weiterhin gültig sein.

2.4 Entstehung

In den Ergebnissen des Dagstuhl Seminars [7] sind vier Ursachen für technische Schulden aufgelistet:

- Entscheidungen
- Zeitdruck
- Prozesse
- Ereignisse

Diese vier sehr abstrakten Ursachen zeigen, dass die Entstehung von technischen Schulden sehr vielseitig ist. In den Ergebnissen des Dagstuhl Seminars sind neben anderen Beispiele auch Gründe wie die Abwesenheit einer wichtigen Person oder das generelle Fehlen von Informationen über technische Funktionen aufgeführt.

Cunningham selbst beschrieb schon in der ersten Erwähnung, dass technische Schulden mit der ersten Zeile Code entstehen.

> „Shipping first time code is like going into debt." [25]

Folgend der gesetzten Definition aus 2.3 kann dies sowohl bewusst als auch unbewusst sein. Darüber hinaus entstehen unbewusste technische Schulden, im Besonders über Zeit zum Beispiel durch technologische Evolutionen, neue Versionen von Abhängigkeiten oder Änderung von Anforderungen. [82]

[1] Begriff entlehnt aus der Finanzwelt: Verbindlichkeit (z. B. aus Bürgschaft oder Gewährleistungsvertrag), bei der noch nicht feststeht, ob sie verwirklicht wird (Oxford Languages)

2.5 Auswirkung

Die Auswirkungen von technischen Schulden werden bereits durch Cunningham umrissen, das mittel- bis langfristige Anfallen von Zinsen in Form von Aufwand und damit Kosten. Nach der Definition aus 2.3 sind die Auswirkungen zumeist auf interne Softwarequalitäten beschränkt und beeinflussen sowohl die Wartbarkeit als auch die Weiterentwicklung eines Systems. Bei bewussten technischen Schulden ist zudem der kurzfristige Gewinn an Geschwindigkeit ein zentraler Aspekt der Auswirkung von technischen Schulden.

Die Anhäufung technischer Schulden soll gemäß Cunningham und Kruchten et al. dazu führen, dass eine Weiterentwicklung oder Wartung des Systems immer schwerfälliger wird, bis zum Stillstand. Dieser Zustand tritt ein, sobald die Entwicklungskapazität den zu zahlenden Zinsen der technischen Schulden entspricht.

In ihrer Studie untersuchten Besker et al. 2017 den anteiligen Aufwand für den Umgang mit technischen Schulden. Anhand einer Online-Umfrage von über 200 Teilnehmern schlussfolgerten die Autoren, dass bis zu 36% der gesamten Entwicklungszeit auf technische Schulden verwendet wird. Hierbei sind alle beteiligten Rollen betroffen, jedoch unterschiedlich stark. Die meiste verschwendete Zeit fällt hierbei auf das Verstehen und Messen von technischen Schulden. [13]

In ihrem systematischen Literaturreview erstellten Lenarduzzi et al. eine Übersicht über die Auswirkungen von technischen Schulden, die in der Literatur beschrieben werden. Diese fasst die Auswirkungen in folgenden Kategorien zusammen: Business, Kunde, Weiterentwicklung, Wartung, Systemqualität, Qualitätsschulden, Produktivität, Projektfaktoren, Sozialfaktoren, andere Faktoren und nicht spezifizierte. [71]

2.6 Umgang

In ihrem Buch „Managing Technical Debt" von Kruchten et al. [65] gehen die Autoren tiefer auf den Umgang mit technischen Schulden ein, hierbei ist eine ihrer Grundaussagen, dass jedes System technische Schulden beinhaltet.

> „Every system has technical debt" ([65])

Aufgrund der von Cunningham beschriebenen Folgen des Bankrotts ist es unerlässlich, technische Schulden in einem System zu beachten, um das Risiko zu minimieren, dass die Menge an Schulden jegliche Weiterentwicklung verhindert oder das System in Gänze nicht mehr wartbar wird.

In dem Literaturreview von Jeronimo et al. [59] wurden folgend Aktivitäten unter dem Umgang mit technische Schulden aufgeführt:

- Vermeidung
  Die Vermeidung technischer Schulden ist eine grundlegende Aufgabe, welche darauf abzielt, nicht unbedacht technische Schulden aufzunehmen, sondern diese vorher gegen ihren Nutzen abzuwägen.

- Identifikation
  Trotz der Vermeidung kommt es immer wieder zu technischen Schulden, diese sind zudem oft nicht bewusst aufgenommen worden. Das Erkennen von unbewussten technischen Schulden, um diese in bewusste Schulden umzuwandeln oder direkt zu tilgen, ist eine zentrale Handlung im Umgang mit technischen Schulden.

- Messen
  Technische Schulden sollten einzeln geschätzt werden. Hierbei können Verfahren unter anderem aus den Agilen Schätzmethoden [126], wie Planning Poker [2], oder eine Kosten/Nutzen Analyse genutzt werden.

- Priorisieren
  Die einzelnen technischen Schulden sollten anhand ihrer Eigenschaften priorisiert werden, um den größten Nutzen aus der Rückzahlung zu erhalten.[3]

- Rückzahlung/Tilgung
  Die Tilgung von technischen Schulden ist ein Vorgehen, welches zum Ziel hat, die technischen Schulden abzubauen. Abhängig von der Art der technischen Schulden kann dies in verschiedenen Formen geschehen. Beispiele hierfür sind Refactorings von schlecht lesbarem Code, Aktualisierung von Abhängigkeiten oder die Adaption neuer Technologien.

- Überwachen
  Die Überwachung von technischen Schulden bezieht sich auf quantitative Metriken, wie den zu erwartenden Aufwand der Tilgung oder die qualitativen

[2] Planning Poker bezeichnet eine Schätzmethode, welche anhand von unabhängigen Schätzungen jedes beteiligten Entwicklers versucht einen Konsens über die Größe oder Komplexität einer Aufgabe zu erreichen.[78]

[3] 2020 fasste Alfayez et al. die bisherigen Forschungen zur Priorisierung in einer systematischen Literaturrecherche zusammen. [2].

Auswirkungen auf die interne Systemqualität [8]. Diese können über den Lebenszyklus eines Systems stark schwanken. Hierbei sind historische Daten besonders wertvoll.

- Dokumentation
  Dokumentation bezeichnet das Aufzeichnen einzelner technischer Schulden in Form eines Backlogs oder ähnlichen Artefakten. Zu dokumentieren sind sowohl die Beschreibung der technischen Schuld, als auch die Priorität, Auswirkung, Entscheidungen und mögliche Hilfen bei der Tilgung. Dokumentation hilft sowohl beim nachhaltigen Verwalten von technischen Schulden als auch in der Kommunikation mit Stakeholdern.

2.7 Arten

Über die Jahre wurden verschiedene Ansätze verfolgt, um unterschiedliche Arten von technischen Schulden zu identifizieren. Rios et al. identifizierte in ihrer tertiären Studie 15 Arten technischer Schulden: [112]

Design-Schulden
Bezieht sich auf Schulden, die durch die Analyse des Quellcodes und die Identifizierung von Verstößen gegen die Grundsätze eines guten objektorientierten Designs entdeckt werden können.

Code-Schulden
Bezieht sich auf Probleme im Quellcode (schlecht geschriebener Code, der gegen bewährte Kodierungspraktiken oder Kodierungsregeln verstößt), die die Lesbarkeit des Codes beeinträchtigen und seine Wartung erschweren können.

Architektur-Schulden
Bezieht sich auf die in der Produktarchitektur auftretenden Probleme, die sich auf die Architekturanforderungen auswirken können. Gewöhnlich sind Architektur-Schulden das Ergebnis von suboptimalen Lösungen im Vorfeld oder von Lösungen, die suboptimal werden. Dieses kommt zustande, wenn Technologien und Muster ersetzt werden. Architektur-Schulden beeinträchtigen einige interne Qualitätsaspekte, wie etwa die Wartbarkeit.

Test-Schulden
Bezieht sich auf Probleme, die bei Testaktivitäten gefunden werden und die die Qualität dieser Aktivitäten beeinträchtigen können.

Dokumentations-Schulden

Bezieht sich auf die Probleme, die in der Dokumentation von Softwareprojekten auftreten.

Defekt-Schulden

Bezieht sich auf bekannte Fehler, die in der Regel durch Testaktivitäten oder durch Benutzer identifiziert und in Fehlerverfolgungssystemen gemeldet werden, die nach Meinung des Entwicklungsteams behoben werden sollten, aber aufgrund konkurrierender Prioritäten und begrenzter Ressourcen auf einen späteren Zeitpunkt verschoben werden müssen.

Infrastruktur-Schulden

Bezieht sich auf Infrastrukturprobleme, die, wenn sie in der Software Organisation vorhanden sind, einige Entwicklungsaktivitäten verzögern oder behindern können. Solche Probleme wirken sich negativ auf die Fähigkeit des Teams aus, ein hochwertiges Produkt zu produzieren.

Anforderungs-Schulden

Bezieht sich auf Kompromisse in Bezug auf die Anforderungen, die das Entwicklungsteam umsetzen muss oder wie es sie umsetzen soll. Mit anderen Worten, es handelt sich um den Abstand zwischen der optimalen Anforderungsspezifikation und der tatsächlichen Systemimplementierung.

Personen-Schulden

Bezieht sich auf menschliche Probleme, die, wenn sie in der Software-Organisation vorhanden sind, einige Entwicklungsaktivitäten verzögern oder behindern können.

Build-Schulden

Bezieht sich auf Probleme, die den Build-Prozess[4] erschweren und unnötig zeitaufwändig machen.

Prozess-Schulden

Bezieht sich auf ineffiziente Prozesse, z. B. wenn das, wofür der Prozess entwickelt wurde, möglicherweise nicht mehr angemessen ist.

Test-Automatisierungs-Schulden

Bezieht sich auf die Arbeit, die mit der Automatisierung von Tests zusammenhängt. Diese Tests sichern zuvor entwickelte Funktionen ab und dienen zur Unterstützung der kontinuierlichen Integration und Entwicklungszyklen.

[4] Prozess, der ausgeführt wird, um aus Quellcode ein verwendbares Software Artefakt zu erstellen

Benutzerfreundlichkeit-Schulden

Bezieht sich auf unangemessene Entscheidungen zur Benutzerfreundlichkeit, die später angepasst werden müssen.

Service-Schulden

Bezieht sich auf die unangemessene Auswahl und Substitution von Web-Diensten, die zu einer Diskrepanz zwischen den Dienstmerkmalen und Anforderungen der Anwendungen führen. Diese Art von Schuld ist relevant für Systeme mit dienstorientierten Architekturen.

Versionierungs-Schulden

Bezieht sich auf Probleme bei der Versionierung von Quellcode, wie z. B. unnötige Code-Forks [5].

Neuere Arbeiten fügen weitere Arten hinzu, darunter „Enterprise Architektur Schulden", welche in der Arbeit von Hacks et al. eine Definition erhalten hat:

> „Enterprise Architecture Debt is a metric that depicts the deviation of the currently present state of an enterprise from a hypothetical ideal state." [45]

In dieser werden Enterprise Architektur Schulden als Metrik für die Abweichung zwischen dem Ist- und einem hypothetischen Ideal-Zustand definiert.

2020 fügen Monaghan und Bass die neue Art „Ökosystem-Schulden" (Ecosystem Debt) hinzu. Dies resultierte aus einer Grounded Theory Methodology (detailliert beschrieben in Kapitel 5), welche sowohl Daten aus Startups als auch aus Großunternehmen nutzte. Die Forschung betrachtet vor allem die Systemlandschaft der Unternehmen und verzichtet auf tiefere Details. Die Autoren führen an, dass ihre Daten auf Unternehmen des Vereinigten Königreiches limitiert sind und die Ergebnisse gegebenenfalls nicht ohne Weiteres übertragbar sein könnten.

Die Autoren definierten Ökosystem-Schulden wie folgt:

> „Softwaresysteme und die Systeme, die sie unterstützen, bilden ein Software-Ökosystem. Wie in jedem Ökosystem können sich die Auswirkungen auf einen Aspekt auf andere auswirken. Wenn die Abhängigkeiten zwischen Softwaresystemen nicht verwaltet werden, können künftige Änderungen an den einzelnen Softwaresystemen, die dieses Ökosystem bilden, kostspielig, wenn nicht gar unmöglich sein" (Übersetzt aus dem Englischen) [89]

[5] Abzweigung in der Codeversionierung Git, welche den Code zu einem Zeitpunkt dupliziert

3 Untersuchtes Umfeld

Nach der Definition der Europäischen Union gelten Unternehmen mit mehr als 250 Mitarbeitern bereits als Großunternehmen. Somit erfüllt allein die Interne IT des Unternehmens die Kriterien eines Großunternehmen. [32]

Besondere Charakteristika von Großunternehmen sind die fortgeschrittene Bürokratisierung und etablierte Prozess und Richtlinien. [49] Großunternehmen setzen häufig auf große Frameworks wie ITIL oder SAFe und streben eine Zertifizierung ihrer Prozesslandschaft an. [46, 124]

Zudem besitzen diese häufig hierarchische Organisationsstrukturen, diese trennen Mitarbeiter anhand von Prozessen in Abteilungen auf, welche über definierte Schnittstellen miteinander in Kontakt treten. Dies isoliert einzelne Bereiche und Mitarbeiter voneinander. Benötigte Kommunikation, welche nicht bereits als Prozess aufgenommen wurde, verstößt gegen bestehende Prozesse und ist daher zumeist unerwünscht. [77] Hierbei ist zu beobachten, dass Vorgesetzte mehr Verantwortung tragen, Detailwissen jedoch entlang der hierarchischen Ordnung nach oben abnimmt. [77, 6]

Es ist darauf hinzuweisen, dass die Anzahl der Mitarbeiter keine Aussage dazu trifft, wie viele dieser an einem Softwareentwicklungsprozess beteiligt sind. Besonders bei produzierende Firmen kann hier eine signifikante Differenz liegen. Entscheidend sind eher die Charakteristiken bezüglich ausgeprägter Prozesse. Diese werden in den Studien jedoch nur selten beleuchtet. Aus diesem Grund sind die Unterteilungen in den verwandten Arbeiten (Kapitel 4) nicht trennscharf.

3.1 Untersuchtes Unternehmen

Die erhobenen Daten stammen von einem deutschen Großunternehmen mit mehr als einer halben Million Mitarbeiter. Das Unternehmen setzt historisch auf externe Entwicklungskapazitäten, hat jedoch vor einigen Jahren größere Bereiche aufgebaut, die sich vor allem auf interne Softwareentwicklung konzentrieren. Die Organisationsstruktur umfasst über 4000 Angestellte.

Die untersuchten Teams sind Teil dieser Organisationsstruktur und agieren insbesondere im Bereich der internen IT-Systeme. Trotz des stetigen Wachstums ist die interne Softwareentwicklung, im Vergleich zur externen Entwicklung, kleiner (gemessen am Umsatz).

Aufgrund der gesetzlichen Regelungen unterscheidet sich die interne und externe Softwareentwicklung und der Lebenszyklus der entwickelten Software. [18] Zudem ist die Untersuchung externer Teams nicht mit internen Regelungen vereinbar.

Folglich konzentriert sich diese Arbeit ausschließlich auf Daten aus der internen Softwareentwicklung.

3.2 Organisation

Die Mitglieder der beobachteten Teams sind den Abteilungen zugeordnet, die vor allem standort- und rollenbezogen sind. Teams werden von anderen Abteilungen mit fachlichem Hintergrund beauftragt, die die Abläufe des Unternehmens verantworten. Diese Abteilungen stammen aus separaten Organisationsstrukturen, die parallel angeordnet sind.

3.3 Teams

Die Teams orientieren sich an agilen Methoden der Softwareentwicklung. Die verwendeten Methoden sind an Scrum oder Extreme Programming (XP) angelehnt.[11, 10]

Alle Teams pflegen eine Aufgabenliste (Backlog) mit Aufgabenbeschreibungen (User Stories oder Stories).

Die Teams arbeiten zumeist nach "Pair Programming", dies bedeutet, dass zwei Entwickler an derselben Aufgabe arbeiten.[10] Der Anteil zwischen der Entwicklung alleine (Solo) und im Pair unterscheidet sich zwischen den Teams und der anstehenden Aufgabe.

Das Arbeiten im Pair wird auch „Pairing Session" oder kurz „Pairing" genannt.

Die beobachteten Teams decken den gesamten Softwareentwicklungszyklus ab, dazu gehören Entwicklung, Tests, Integration, Bereitstellung (Deployment), Betrieb, Fehlerbehebung, Infrastrukturverwaltung und Abschaltung. Die Teams bestehen aus mehreren Personen, die unterschiedliche Rollen einnehmen.

Product Owner (PO)

Der PO vertritt den beauftragenden Fachbereich und übernimmt die Abstimmung mit Gremien und Stakeholdern.

Product Manager (PM)

Der PM übernimmt die Pflege des Backlogs, beschreibt Aufgabenpakete (Stories) und übernimmt organisatorische Aufgaben.

Entwickler (Dev)

Die Entwickler schreiben Applikations-, Test- und Automatisierungscode.

Designer (Des)

Designer unterstützen PM und Entwickler bezüglich der Nutzererfahrung, Interviews und grafischen Gestaltung der Software.[6]

3.4 Besprechungen

Im Folgenden wird näher auf Besprechungen und Rituale eingegangen, die in den meisten untersuchten Teams etabliert sind.

Pre Iterational Planning Meeting - PreIPM

Das PreIPM befasst sich mit der Vorbereitung von Backlog Einträgen (auch Story genannt). Hierbei bespricht ein Teil des Teams neue Stories und prüft, ob sie genügend Informationen beinhalten, um vom Team eingeplant zu werden. Teams haben zumeist eine „Definition of Ready" vereinbart, die beschreibt, welche Kriterien erfüllt sein müssen, damit eine Story gestartet werden kann.

Iteration Planning Meeting - IPM

In diesem Meeting bespricht das gesamte Team die Stories für den neu beginnenden Zyklus. Das IPM liegt im XP häufig am Montag und leitet die Arbeitswoche ein. Das gesamte Team schätzt Stories in sogenannten „Story Points", welche zumeist relative Komplexität darstellen [60, 56].

Review

Teams besprechen Änderungen an der Codebasis. Dies dient vor allem dem Wissensaustausch und der Erhaltung einer hohen Codequalität.

Retrospektive - Retro

„Retrospektiven sind Teamtreffen, deren Ziel es ist, aus der Vergangenheit zu lernen. ‚Retrospektiv' bedeutet ‚rückblickend'. Die Teammitglieder schauen also gemeinsam zurück und bewerten, was gut und was schlecht gelaufen ist. Sie analysieren, warum Dinge gut liefen oder von Erwartungen abwichen, um Maßnahmen zur Verbesserung zu formulieren und anzugehen."

[1]

[6] Die Rolle des Designers hat in den Beobachtungen keinen größeren Einfluss auf den Umgang mit technischen Schulden und ist nur der Vollständigkeit halber beschrieben.

Technische Retrospektive - Techtro

Die technische Retrospektive ist ein regelmäßiges Zusammenkommen der Entwickler, um über die bestehende Codebasis zu sprechen und rückblickend zu bewerten, welche Teile gut und welche schlecht sind. Hierzu bereiten sich meist erfahrene Teilnehmer vor, um mit dem Team darüber zu sprechen, welche Stellen im Code gegebenenfalls anders umgesetzt werden sollten. Teilweise zeigen sie aber auch, wie eine Softwarebibliothek in spezifischen Situationen verwendet werden sollte. Das Ziel der technischen Retrospektive ist der Wissensaustausch und die Verbesserung bezüglich technischer Umsetzungen.

4 Verwandte Arbeiten

Im Folgenden werden verwandte Arbeiten vorgestellt, die entweder eine vergleichbare Methodik aufweisen oder ähnlich thematisch einzuordnen sind. Anschließend werden daraus resultierende Lücken der Forschung aufgezeigt, welche die Grundlage für diese Forschungsarbeit liefern.

Es ist zu beachten, dass in dieser Arbeit eine Definition des Begriffes „Technische Schulden" von der 2016 veröffentlichten Definition abgeleitet wird. Arbeiten, die vor 2016 veröffentlicht wurden, nutzen andere Definitionen des Begriffes „Technische Schulden", die teilweise nicht explizit benannt werden [7]. Insbesondere Arbeiten aus Jahren vor 2016 beziehen sich zumeist auf McConnels Definition aus dem Jahr 2008 [85] oder auf die erstmalige Erwähnung von Cunningham [25].

4.1 Forschung in Großunternehmen

Im Folgenden werden Forschungsarbeiten beleuchtet, welche *technische Schulden* im Umfeld von Großunternehmen untersuchen. Hierbei ist darauf hinzuweisen, dass viele Arbeiten nicht die europäische Definition von Großunternehmen anwenden [32]. In den Vereinigten Staaten von Amerika (USA) gibt es keine offizielle Definition, in Brasilien hingegen, gibt es unter anderem seit 2015 eine Einteilung, die jedoch abhängig von der Art des Unternehmens ist [42]. Um einen Vergleich zu ermöglichen, werden die Literaturdaten auf Studien begrenzt, bei denen mindestens 30% der untersuchten Unternehmen die europäische Definition des Begriffs „Großunternehmen" erfüllen.

Eine der ersten Arbeiten, welche sich mit der Thematik von technischen Schulden im Kontext eines Großunternehmens befasst, ist die 2011 veröffentlichte Forschungsarbeit „An Enterprise Perspective on Technical Debt" von Klinger et al. [62]

In dieser wird die Sicht von vier technischen Architekten von der Firma IBM auf technische Schulden untersucht und ausgewertet. In den Ergebnissen sind grundlegende Prinzipien beschrieben. Dazu gehören die Abwägung zwischen Organisations- und Teamzielen, die Notwendigkeit, technische Entscheidungen zu dokumentieren, und Herausforderungen in der Kommunikation zwischen verschiedenen Beteiligten der Softwareentwicklung.

Aufgrund der allgemeinen Aussagen und der ähnlichen Umgebung werden in der Diskussion (Kapitel 8.1) die Ergebnisse der Forschungsarbeit von Klinger et al. detailliert anhand der eigenen Beobachtungen überprüft.

Eine weitere frühe Arbeit aus dem Jahr 2012 untersucht technische Build-Schulden bei dem Großunternehmen Google [90]. In der Forschungsarbeit werden spezielle Arten von technischen Schulden innerhalb der Firmen-eigenen Entwicklung identifiziert.

Bei der untersuchten Umgebung handelt es sich um eine spezifische Einrichtung, da firmenintern alle Produkte in einem zentralen Codeverzeichnis abgelegt und über ein proprietäres Build-Werkzeug gebaut werden. Über eine ähnliche Verfahrensweise und die daraus resultierende Umgebung ist in keiner anderen Forschungsarbeit berichtet worden. Aus diesem Grund ist nicht klar, ob die gefundenen Probleme in dieser Größenordnung in anderen Unternehmen wiederzufinden sind.

In ihrer Studie „Managing Technical Debt: An Industry Case Study" von 2013[22] untersuchten Codabux et al. ein Softwareunternehmen, welches nach der europäischen Definition als Großunternehmen eingeordnet werden kann. Die Forschungsdaten der Studie wurden während der Einführung agiler Methoden in einem Unternehmen gesammelt. Die gewonnenen Forschungsdaten wurden dazu verwendet, um eine Definition des Begriffs „technischer Schulden" und deren Charakteristiken im Umfeld von agiler Softwareentwicklung zu erarbeiten.

In ihren Beobachtungen finden sich einige Beispiele für technische Aufräumarbeiten, die Einfluss auf Kunden ausüben können. Aufgrund der Einflussnahme auf Kunden ist festzuhalten, dass die verwendete Definition von dem Begriff „Technische Schulden" abweicht und die Ergebnisse der Forschungsarbeit nur bedingt vergleichbar sind mit den Ergebnissen aus diesem Werk.

Jedoch soll hier ein spezifischer Fund aus den Beispiel der Forschungsdaten von Codabux et al. herausgegriffen werden: In dem beobachteten Entwicklungsbereich einer Firma wurden dedizierte Scrum Teams dazu eingesetzt, um „technische Schulden" (nach Codabux et al.) zu tilgen. Da die Definition von Codabux et al. von der eigenen abweicht, können die Ergebnisse nicht ohne weiteres übernommen werden, jedoch liegt die Vermutung nahe, dass diese Teams auch einen größeren Anteil an *technischen Schulden* (Definition aus 2.3) bearbeitet haben. Somit bleibt die grundlegende Aussage, dass Teams zur dedizierten Tilgung von technischen Schulden aufgestellt werden, verallgemeinerbar.

Eine weitere Studie von Daneva et al. aus dem Jahr 2013 untersucht die Anforderungspriorisierung in großen Systemen, welche durch Entwickler einer dritten Firma entwickelt werden (Outsourcing) [27]. Daneva et al. geht hierbei nach der Grounded Theory Methodology vor. In ihrer Datenanalyse bezüglich der Priorisierungskriterien stößt sie auf das Konzept der „technischen Schulden". Dies bezieht sich auf die Beschreibung eines Probanden, welcher angibt, dass Teams in der agilen Softwareentwicklung darauf fokussiert sind, die Anforderungen der derzeitigen Iteration umzusetzen. Aus dem daraus resultierenden Fokus auf kurzfristige Ziele leitet der Autor eine Situation ab, die eine Weiterentwicklung behindert, womit die Definition des Begriffs „technische Schulden" mit der in diesem Werk getroffenen

übereinstimmt.

2014 untersuchten die Forscher Holvitie et al. die Auswirkungen von agilen Entwicklungsmethoden auf technische Schulden[51]. In der Studie wurde eine anonyme Umfrage in Finnland durchgeführt. Das Ergebnis der Studie ist, dass Softwarepraktiker schätzen, dass agile Softwareentwicklungspraktiken sich positiv auf technische Schulden und die Wartung auswirken. Technische Schulden werden in diesem Werk anhand von Beschreibungen von McConnel[85] und Brown [17] erläutert, daher wäre eine Überprüfung dieser Forschungsergebnisse anhand neuerer Definitionen des Begriffes „Technische Schulden" interessant.

2015 befragten Ernst et al. über 1800 Personen, zumeist Softwareentwickler und Architekten aus drei großen Organisationen [31]. Im Anschluss führten sie sieben weitere Interviews, um einen tieferen Einblick zu erlangen. Neben der grundlegenden Forschung bezüglich des Verständnisses über das Konzept der technischen Schulden (nach McConnel [85]) untersuchten sie auch den derzeitigen Umgang mit technischen Schulden. Der Umgang mit technischen Schulden war zumeist unbewusst. Es wurde ein Bedarf nach Werkzeugen und Praktiken für den Umgang mit technischen Schulden festgestellt.

2016 untersuchen Ramasubbu et al. die Auswirkungen von technischen Schulden auf die Verlässlichkeit von Enterprise-Softwaresystemen [106]. Dazu wurde ein Längsschnittdatensatz[7] ausgewertet, der den zehnjährigen Lebenszyklus eines „Commercial off-the-shelf" (COTS)[8]-basierten Unternehmenssoftwaresystems umfasst, das bei 48 verschiedenen Kundenunternehmen eingesetzt wird. Unter dem Begriff „Technische Schulden" verstehen die Autoren „angestaute Wartungsverpflichtungen, die in Zukunft erfüllt werden müssen" [106].

Anhand der Beschreibungen des untersuchten Systems erscheint die Kategorisierung der Autoren jedoch falsch gewählt. Die Software bietet mehrere Konfigurationsmöglichkeiten, sowie die Möglichkeit, eigenen Code zu verfassen und auszuführen, daher wäre statt COTS eher MOTS (Modifiable off-the-shelf - modifizierbare Fertigsoftware) richtig. Die zentrale Schlussfolgerung der Forschungsarbeit ist, dass technische Schulden die Verlässlichkeit von Enterprise-Systemen beeinträchtigen.

Die nötigen Aufwände, für den Umgang mit technischen Schulden, untersuchten Guo et al. 2014 in ihrer Arbeit anhand einer Fallstudie in einer Dienstleistungsfirma aus Brasilien, welche Enterprise-Level Software, Beratung und Schulungen anbietet

[7] „Längsschnittdaten (auch: zeitabhängige Daten) sind das Ergebnis von Datenerhebungen, die im Zeitablauf immer wiederkehrend an der gleichen statistischen Einheit durchgeführt werden." [69]

[8] Commercial off-the-shelf: Fertige Softwareprodukte, die unverändert an mehrere Kunden gegeben werden z.B. Word

[43]. Das Fazit der Forschungsarbeit ist, dass bei Einführung der Nachverfolgung und Überwachung von technischen Schulden erhebliche Aufwände entstehen, diese jedoch nach und nach abnehmen und auf ein vertretbares Maß sinken.

In der 2016 veröffentlichten Arbeit „The Introduction of Technical Debt Tracking in Large Companies" untersuchten Martini et al. ebenfalls die Einführung der Nachverfolgung von technischen Schulden (nach McConnel[85] und Brown [17]) in Großunternehmen aus Skandinavien [80]. Dazu wurden 226 Teilnehmer aus 15 Organisationen befragt und drei spezifische Organisationen, in denen technische Schulden nachverfolgt werden, gesondert betrachtet. Aus den erhobenen Forschungsdaten geht hervor, dass den meisten Befragten zwar das Konzept der technischen Schulden bekannt ist, diese jedoch bei über der Hälfte der Befragten nicht nachverfolgt werden. In keinem der untersuchten Bereich wurde Automatisierung in jeglicher Form verwendet.

Der Versuch, mit technischen Schulden umzugehen, beansprucht laut der Umfrage 25% der Entwicklungszeit. Es ist jedoch nicht ganz klar definiert, ob diese 25% nur das Management oder auch die Behebung technischer Schulden beinhalten. Für die Tätigkeit des Schuldenmanagements alleine wäre dies ein signifikanter Anteil, der jedoch unter dem Wert von Besker et al. mit 37% liegt [13].

Aus den Daten der Umfragen, sowie dem theoretischen Stand zu der Thematik der technischen Schulden, erstellten Martini et al. ein strategisches Modell zur Übernahme systematischer Prozesse zum Umgang mit technischen Schulden:

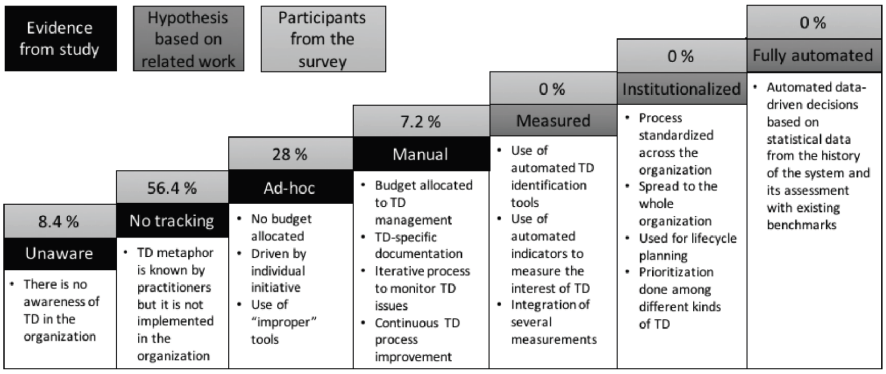

Abbildung 2: Strategic Adoption Model [80]

Das Modell beschreibt die verschiedenen beobachteten Phasen, die ein Unternehmen während der Einführung der Nachverfolgung von technischen Schulden durchlaufen kann. Hierbei extrapolieren sie nicht beobachtete Phasen anhand ihrer Erfahrungen (nachfolgend markiert durch *):

Unbewusst (unaware):

Kein Bewusstsein für „technische Schulden" in der Organisation vorhanden.

Keine Nachverfolgung (no tracking):
Die Metapher „technische Schulden" ist den Praktikern bekannt, wird aber in der Organisation nicht umgesetzt.

Ad-Hoc (ad-hoc):
Kein Budget zugewiesen, Nachverfolgung wird angetrieben durch individuelle Initiativen und Verwendung von „unsachgemäßen" Werkzeugen.

Manuell (manual):
Für die Verwaltung von technischen Schulden werden Mittel zugewiesen, spezifische Dokumentation, iterative Prozesse zur Überwachung von Problemen verursacht durch technische Schulden und kontinuierliche Verbesserung der Prozesse sind etabliert.

Gemessen (measured)*:
Einsatz von automatisierten Instrumenten zur Identifizierung von technischen Schulden, Verwendung automatischer Indikatoren zur Messung der Zinsen und Integration von mehreren Messungen sind etabliert.

Institutioniert (institutionalized)*:
Standardisierter Prozess in der gesamten Organisation ist etabliert und wird für die Lebenszyklusplanung verwendet. Priorisierung für verschiedene Arten von technischen Schulden ist etabliert.

Vollautomatisiert (fully automated)*:
Datengestützte Entscheidungen sind auf der Grundlage statistischer Daten aus der Historie des Systems und seiner Bewertung mit bestehenden Leistungstests automatisiert.

Das Modell stellt eine Vollautomatisierung als das höchste Ziel dar. Nach dem derzeitigen Forschungsstand ist es jedoch fraglich, ob eine Vollautomatisierung möglich bzw. sinnvoll ist. Technische Schulden werden vor allem durch den Kontext und im Besonderen durch die weitere Ausrichtung der Weiterentwicklung bestimmt. Dieser Kontext lässt sich (nach aktuellem technischen Stand) nicht automatisiert erfassen und ist zudem nirgends vollständig dokumentiert.

Die Daten der Veröffentlichung „The Introduction of Technical Debt Tracking in Large Companies"[80] wurden im Rahmen einer weiteren Forschungsarbeit 2018 vertieft betrachtet [79]. Bei der neuartigen Betrachtung wird dargestellt, in welcher Form die Teams technische Schulden dokumentieren. Ein Großteil der Teams nutzt das Backlog zur „Dokumentation" von technischen Schulden, wobei auch Werkzeuge

zur statischen Analyse von Code und „Lint”-Programme[9] zum Einsatz kommen. In ihrer Wirkung ähneln sich die drei Ansätze, zudem sind sie im Vergleich zu anderen Vorgehen ohne großen Mehraufwand anwendbar. Die Schlussfolgerung der Forschungsarbeit von Martini et al. ist, dass diese drei Ansätze als praktische Verfahren zur Verfolgung von technischen Schulden geeignet sind.

In einer vergleichenden Fallstudie aus dem Jahr 2016 von Martini und Bosch wurde die Thematik der Lücken in den Architekturpraktiken von agilen Großunternehmen untersucht. Diese Lücken stellen ein hohes Risiko für Unternehmen dar, da sie technische Architektur-Schulden erzeugen können. Dazu wurden Daten aus fünf großen Software Unternehmen gesammelt und mit der Grounded Theory Methodology analysiert. Im Zuge der Auswertung wurde ein Vorgehen entwickelt, welches auch den Umgang mit technischen Architektur-Schulden unterstützen kann, dies bedarf jedoch weiterer Forschung. [81]

Eine weitere Grounded Theory, bezüglich der Praktiken im Umgang mit technischen Schulden, stellten 2017 die Forscher Codabux et al. in ihrem Artikel „An empirical assessment of technical debt practices in industry” vor [23]. Die Studie sammelt Erkenntnisse von Praktikern über die Schlüsselkomponenten von technischen Schulden, unter anderem ihre Definition, Charakterisierung, Folgen, Vorteile und ihre Kommunikation. Sie werteten 67 Umfragen und 17 Interviews aus. Über 50% der Befragten arbeiten in Großunternehmen. Für die Definition des Begriffs technische Schulden werden die Ergebnisse aus dem Dagstuhl Seminar [7] verwendet. In dem Artikel wird ein „Defekt”[10] als eine Art von technischen Schulden definiert. Weiterhin wird auf die geringe Häufigkeit von formellen oder informellen Messungen von Entscheidungen über technische Schulden eingegangen, sowie die Art der Sachlagenaufnahme im Zuge einer Risikoanalyse. Aus den Daten wird weiterhin eine Taxonomie für technische Schulden und deren Management abgeleitet.

Aufbauend auf den Funden eines Literaturwerkes aus dem Jahr 2017 über das Ausmaß an verschwendeter Zeit durch technische Schulden und im Besonderen technischer Architektur-Schulden [13] wurde 2018 von Martini et al. die Identifikation und Priorisierung von technischen Architektur-Schulden in einem Großunternehmen [83] thematisiert. Dazu wurde eine anonyme Umfrage per Fragebogen, Interviews und Code-Reviews durchgeführt, welche durch quantitative und qualitative Methoden ausgewertet wurden. Durch Evaluierung der Daten wurde weiterhin ermittelt, dass

[9] Programm zur automatischen Analyse von Quellcode zur Erkennung von Bugs und (nach eigenen Angaben) „Seltsamkeiten” [58]

[10] Fehler in der Software

„Architecture Smells" (AS)[11] dazu geeignet sind, technische Architektur-Schulden zu identifizieren. In ihren Evaluierungen fanden sie heraus, dass ein Großteil der identifizierten Architecture Smells auf bisher unbekannte technische Architektur-Schulden hinweisen und zudem bekannte Schulden bestätigten.

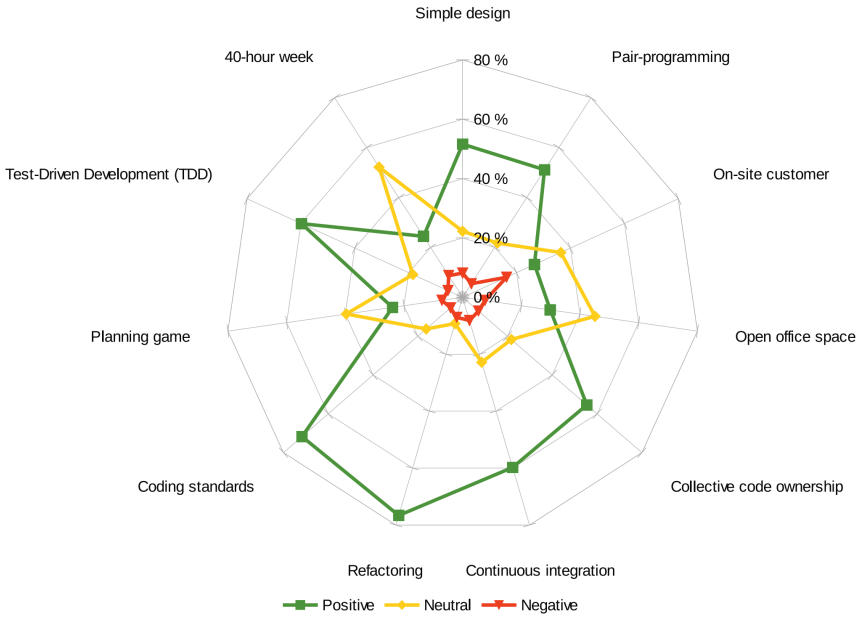

Abbildung 3: Wahrgenommene Auswirkungen agiler Methoden auf technische Schulden

Eine weitere Arbeit aus dem Jahr 2017 untersucht anhand einer multinationalen Umfrage[12] das Zusammenspiel zwischen technischen Schulden und agilen Softwareentwicklungsmethoden [52]. Ihre Ergebnisse weisen darauf hin, dass agile Methoden einen positiven Einfluss auf technische Schulden haben. Im Besonderen hatten Test-Driven-Development, Coding Standards und Refactoring den größten positiv wahrgenommenen Effekt (siehe 3). Die Arbeit bietet eine Grundlage, durch die besonders Teams den Umgang mit technischen Schulden verbessern können, indem sie bereits etablierte Prozesse und Praktiken erweitern, statt neue und gegebenenfalls störende Methoden für das technische Schuldenmanagement einzuführen. Es ist wichtig darauf hinzuweisen, dass die Ergebnisse allein auf der Meinung der befragten und interviewten Personen basieren. Diese wurden nicht durch Feldbeobachtungen überprüft. Anhand der großen Übereinstimmung der Befragten ist jedoch davon auszugehen, dass die grundlegenden Tendenzen stimmen.

[11] „Smell" zu deutsch „Geruch" bezeichnet Codestellen die aufgrund ihrer Qualität „schlecht riechen". Dies kann ein Hinweis sein, dass diese Stellen angepasst werden müssen [33]

[12] Brasilien, Finnland und Neuseeland

2019 untersuchten S. de Toledo et al. im Umfeld eines Großunternehmen die Risiken und Besonderheiten von technischen Architektur-Schulden, welche durch oder in einer Microservice[13] Architektur entstehen können [118].

Sie nutzten dazu qualitative Methoden, um Dokumente und Interviews zu analysieren. Durch ihre Untersuchungen fanden sie folgende Probleme, die mit Microservices in Verbindung stehen:

- Zu viele Punkt-zu-Punkt Verbindungen zwischen Systemen,
- Geschäftslogik in der Kommunikationsschicht,
- fehlende Standards für die Kommunikation zwischen Services,
- schwaches Quellcode- und Wissensmanagement,
- unnötig viele verschiedene Technologien.

Diese Ausprägungen von technischen Architektur-Schulden führten zu erheblichem Mehraufwand (Zinsen). Die Ergebnisse zeigen Probleme, Risiken und Lösungsansätze auf, im und für den Umgang mit technischen Schulden im Kontext von Microservices, und beleuchten damit ein größtenteils unerforschtes Themengebiet.

2019 untersuchten Albuquerque et al. Entscheidungen im Zusammenhang mit technischen Schulden, wie agile Teams sie während der Priorisierung des Backlogs fällen. Es handelt sich hierbei sehr wahrscheinlich um die erste veröffentlichte Arbeit, welche die Priorisierung von technischen Schulden in der Praxis untersucht. Dazu befragten die Forscher 17 Praktiker, aus vier Großunternehmen aus Schweden, zunächst durch eine Umfrage, anschließend durch tiefergehende Interviews. Aus einer vorgelagerten Literaturrecherche erarbeiteten sie ein Modell bezüglich der Priorisierung von technischen Schulden (4)

[13] Microservices sind kleine autonome Services, die zusammenarbeiten. [131, 92]

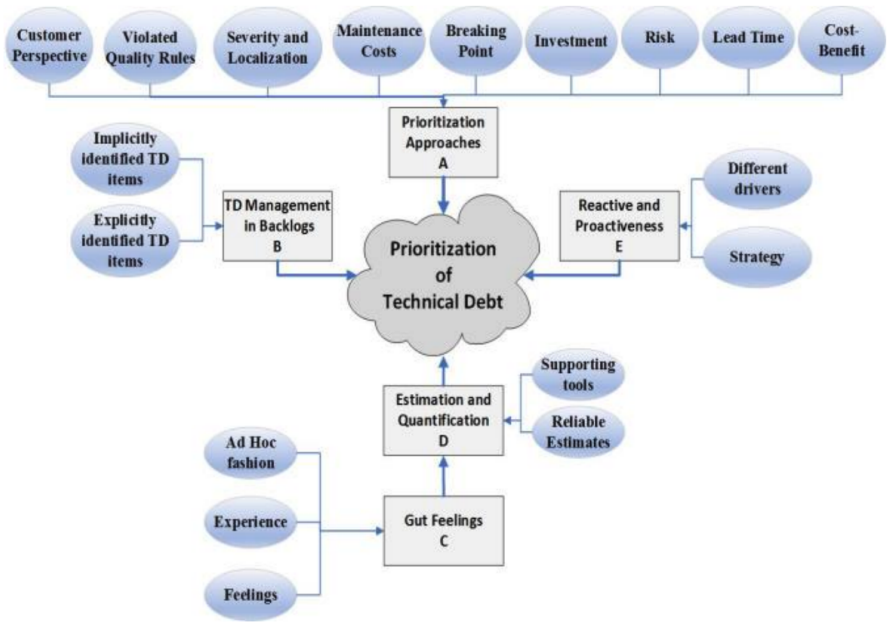

Abbildung 4: Konzeptuelles Modell - Priorisierung technischer Schulden [12]

In dem konzeptuellen Modell enthält der Bereich „Prioritization Approaches - A" die durch die Forschung vorgeschlagenen Herangehensweisen zur Priorisierung. Dem gegenüber steht der Bereich „Estimation and Quantification - D", der für die Abschätzung und Quantifikation von Parametern steht, die einen Einfluss auf die Priorisierung haben. Unterstützende Werkzeuge und verlässliche Schätzungen sind Teil dieses Bereiches. Ein wichtiger Punkt ist, dass „Gut Feelings - C" (das „Bauchgefühl") als markanter Einfluss auf die Abschätzung und Qualifikation dargestellt wird. Dieses Bauchgefühl wird zudem durch Ad-hoc Situationen, Erfahrung und Gefühle bestimmt. Dieses spiegelt die Ergebnisse von Van Vliet [130], Guo et al. [43] und Carriere et al. [20] wider.

Aus ihren qualitativen Untersuchungen der gesammelten Daten extrahierten sie verschiedene Erkenntnisse. Unter anderem nutzen viele Teams einen zweiten, nicht offiziellen Backlog, um technische Schulden nachzuverfolgen. Dies führte dazu, dass technische Schulden im Priorisierungsprozess übergangen wurden. Darüber hinaus wurde ihnen berichtet, dass das Bauchgefühl einen erheblichen Einfluss auf die Schätzungen und Priorisierungen der technischen Schulden hatte. Zu beachten ist, dass die befragten Praktiker nur selten Softwareentwickler waren, drei der vier Unternehmen waren ausschließlich durch andere Rollen, wie Manager oder Architekten, vertreten.

Ihre Erkenntnisse fassen sie im Schaubild 5 zusammen. Bezüglich der Validität der Ergebnisse bemängelten die Forscher die fehlende Detailtiefe durch die Art der Datensammlung, zudem beschränkte sich ihre Untersuchung auf eine kleine Anzahl

von Unternehmen.

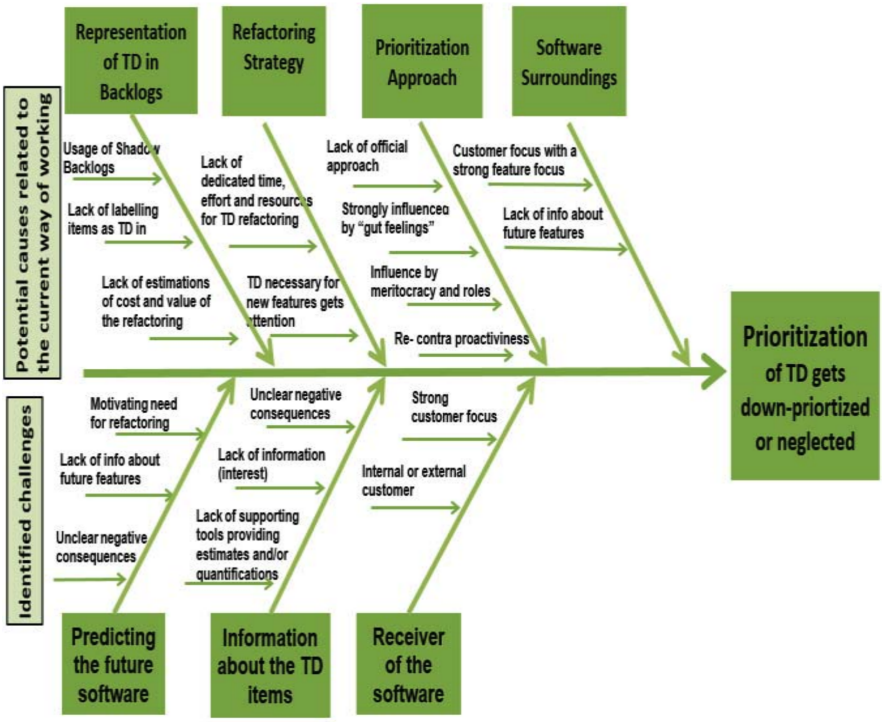

Abbildung 5: Identifizierte Ursachen und Herausforderungen im Zusammenhang mit dem Priorisierungsprozess von TS in Backlogs [12]

Im Jahr 2020 veröffentlichen Verdecchia et al. eine fundierte Theorie über technische Architektur-Schulden [128]. Dabei leiten sie unter anderem folgende Arten des Umgangs mit dieser spezifischen Art von technischen Schulden ab:

- Aktive Management-Strategien
  - Pfadfinder Regel - Eine Stelle sauberer (weniger TS) hinterlassen als sie vorgefunden wurde
  - Systematisch dedizierte Zeit - zur Tilgung von technischen Schulden
  - Technischer Kredit - Investition von Ressourcen zur Verbesserung, Wartbarkeit und Entwicklungsfähigkeit der Architektur vor dem Aufkommen von ATD
- Reaktive Management-Strategien
  - Opportunistisches Patching - kleine Verbesserungen über Zeit
  - Umfassendes Refactoring - benötigt dedizierte Zeit

- Passive Management-Strategien
  - Vernachlässigung - Aufwand zu groß für den Nutzen der Tilgung

Zudem weisen sie auf den menschlichen Faktor als Ursache für technische Architektur-Schulden hin, darunter fallen fehlende Expertise in Kombination mit kognitiven Verzerrungen (Dunning-Kruger-Effekt [68])

Die Forscher Apa et al. replizierten 2019 eine Befragung aus Brasilien [117] in Uruguay [5]. In ihren Auswertungen fanden sie heraus, dass die Teilnehmer ein deutliches Defizit bezüglich des Bewusstseins über technische Schulden haben. Aus ihren Daten sticht im Besonderen heraus, dass Messungen und Überwachung von technischen Schulden zwar als wichtig erachtet werden, jedoch kaum in der Praxis umgesetzt werden. Sie weisen darauf hin, dass die Antworten durch die unterschiedlichen Verständnisse, wie technische Schulden definiert sind, beeinträchtigt werden und einige Antworten bereits darauf hindeuten. So geben einige der Befragten an, in ihrem System gar keine technischen Schulden zu haben.

> „Trotzdem sind wir der Meinung, dass der Einsatz von Werkzeugen an sich die Probleme, die durch die Anhäufung von TD verursacht werden, nicht löst, wenn sie nicht durch die Annahme von TDM-Aktivitäten im Softwareentwicklungsprozess unterstützt werden." (Übersetzt aus dem Englischen) [5]

Ihre Auswertungen weisen darauf hin, dass die automatisierte Erfassung von technischen Schulden nicht die Anhäufung von technischen Schulden verhindert, wenn die Ergebnisse nicht in den Softwareentwicklungsprozess mit einbezogen werden.

In ihrem Artikel „An Analysis of Automated Technical Debt Measurement" von 2020 untersuchen Khomayakov et al. Werkzeuge zur automatisierten Erkennung und Messung von technischen Schulden [61]. Dazu untersuchten sie 835 Studien bis 2020 nach dem Vorgehen eines systematischen Literaturreviews. Dabei fällt ihre Zusammenfassung eher unvorteilhaft aus. Es gäbe eine konstante Menge an neuen Ansätzen, welche jedoch zumeist nicht unabhängig evaluiert werden, zudem basieren diese teilweise nicht auf Erkenntnissen vorheriger Studien.

Im Jahr 2021 veröffentlichten Sousa et al. ihre Ergebnisse über den Umgang mit technischen Schulden in groß angelegten verteilten Projekten. Dazu untersuchten sie Dokumente von 33 Projekten bei dem Großunternehmen Ericsson. Im Anschluss nutzten sie Interviews mit Senior Entwicklern, um ihre Funde zu interpretieren. Komplexe Aufgaben tendieren dazu, mehr technische Schulden anzusammeln, aus diesem Grund empfehlen sie Praktikern, große und komplexe Aufgaben zu vermeiden

und wenn möglich in kleinere Aufgaben zu zerlegen. Eine weitere Auffälligkeit der Studie ist, dass die Befragten ihre Vermutung äußerten, dass globale Distanz zwischen Teams einen enormen negativen Einfluss auf technische Schulden hätte. Diese Vermutungen konnten in den Daten jedoch nicht belegt werden.

Einen weiteren interessanten Ansatz, um den Umgang mit technischen Schulden in einem Großunternehmen zu stärken, präsentierten Haki et al. 2022 in ihrer Studie „Digital nudging for technical debt management at Credit Suisse" [47]. In dem Finanz-Großunternehmen Credit Suisse wurden für Projekte sogenannte „Nudges" eingeführt (eng. „nudge" zu deutsch: Stups/ Anstoß). Nudging ist eine Methodik aus der Verhaltensökonomie und beschreibt ein strategisches Vorgehen, welches menschliches Verhalten ohne Druck verändert [121]. Bei den eingeführten Nudges handelt es sich in diesem Fall um eine Art Punktekarte, welche verschiedene Werte über die Codebasis visualisiert (siehe Beispiel Nudge 6). Die unterschiedlichen Metriken wurden automatisiert aus Systemen wie SonarQube[14] gewonnen. Nach einem Jahr wurde die Nudges anhand von drei Teams evaluiert. Zwei der Teams empfanden die Nudge als Hilfe und nutzten sie als Indikator am Ende einer Iteration, ob sie sich mehr auf technische Schulden konzentrieren müssen. Das dritte Team ignorierte die dargebotenen Informationen auf Grund von anderen Prioritäten. Zudem wiesen ihre Metriken drastische Verschlechterungen auf.

[14] Werkzeug zur statischen Codeanalyse https://docs.sonarsource.com/sonarqube/latest/

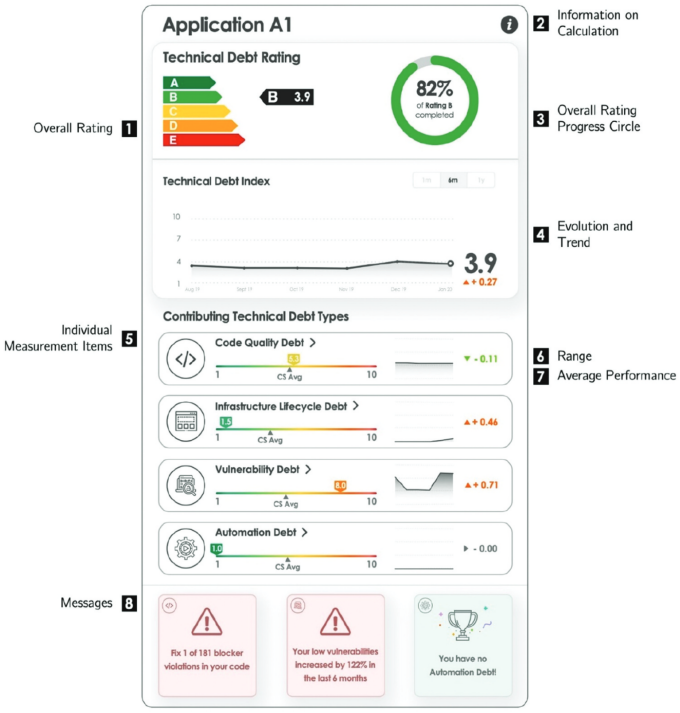

Abbildung 6: Exemplarische TDM Nudge [47]

Die Forscher haben gezeigt, dass Nudging ein Instrument sein kann, um das Bewusstsein von technischen Schulden in einem Unternehmen zu steigern, ohne Druck, durch die Einführung neuer Prozesse, zu benötigen. Offen bleibt jedoch, inwieweit sich Goodhart's Law auf die verwendeten Metriken auswirkt. Dies besagt:

> „Jede beobachtete statistische Regelmäßigkeit wird tendenziell zusammenbrechen, sobald zu Kontrollzwecken Druck auf sie ausgeübt wird." (Übersetzt aus dem Englischen) [119]

Wenn die verwendeten Metriken nicht mit Bedacht gewählt werden, kann es dazu kommen, dass Teams nur noch auf diese optimieren und dadurch ungewollte Nebeneffekte entstehen. Ein ähnlicher Zusammenhang wird im C2-Wiki[15] mit technischen Beispielen beschrieben.[19]

4.2 Forschung in anderen Umgebungen

Der Forscher Jean-Louis Letouzey präsentierte 2016 seine Erfahrungen mit Metriken, die über technische Schulden erhoben werden, um ein großes Portfolio von Applikationen zu verwalten [72]. Hierbei nutzt er Metriken, die in einer Methode

[15] https://wiki.c2.com/ das Ur-Wiki, welches von Ward Cunningham erstellt wurde

namens SQALE[16] beschrieben werden. Eines der verbreitetsten Tools für die statische Analyse von Quellcode, SonarQube[17], nutzt die auf SQALE basierenden Metriken zur Bestimmung von Kennzahlen bezüglich technischer Schulden. 2021 untersuchte Murillo et al. die von SonarQube erzeugten Metriken bezüglich der Validität für technische Schulden. Den Ergebnissen ihrer Literaturrecherche und einer Fallstudie nach gibt es einige Risiken bezüglich der Validität der Metriken, trotzdem können diese im Umgang mit technischen Schulden unterstützen. [91]

Ein spannendes Umfeld ist die Untersuchung von Open Source Software Projekten. Open Source bezeichnet hierbei Code der der Öffentlichkeit zugänglich ist und in der Regel durch freiwillige Helfer entwickelt oder unterstützt wird [109].

Die große Menge an verfügbaren Quellcode, Fehlerbeschreibungen (Issues) und versionierter Historie machen dieses Umfeld besonders interessant für Analysen, die sich den Quellcode und dessen Veränderungen genauer anschauen. Dies sind Informationen, die von Unternehmen nur eingeschränkt zur Verfügung gestellt werden. In der Arbeit „An Exploratory Study on Self-Admitted Technical Debt" [103] von Potar und Shihab untersuchten die Forscher in vier großen Open Source Projekten bewusste eingegangene technische Schulden („self-admitted technical debt" - SATD). Anhand von Kommentaren, welche mit Zeichenketten wie „TODO" oder „FIXME" beginnen, identifizierten sie Stellen im Quellcode, welche SATDs sein könnten. Die Analyse der Kommentare verwies teils tatsächlich auf eingegangene Umgehungen oder genommene Abkürzungen in der Umsetzung. Sie untersuchten daraufhin die Menge und Häufigkeit, den Grund der Entstehung und wann diese Stellen bereinigt werden. Unter anderem fanden sie heraus, dass Zeitdruck und Codekomplexität keinen Einfluss auf die Aufnahme dieser Art von technischen Schulden hat. Zudem tendierten besonders erfahrene Entwickler mehr dazu, diese Art von Schulden aufzunehmen.

Diese Ergebnisse könnten stark mit dem Umfeld der Open Source Entwicklung zusammenhängen. Freiwillige Entwickler, wie sie häufig in der Open Source Entwicklung vorkommen, teilen sich ihre Zeit selbst ein und können zu nichts gezwungen werden. Darüber hinaus kommt es vor, dass Entwickler über Open Source Arbeit ihre Reputation verbessern wollen, die Aufnahme technischer Schulden könnte dabei kontraproduktiv sein. Zudem wurde nicht untersucht, ob erfahrenere Entwickler ggf. transparenter mit technischen Schulden umgehen und anhand ihrer Erfahrung bereits die Fähigkeit entwickelt haben, diese an entsprechender Stelle zu dokumentieren.

Auf dieser Arbeit aufbauend gibt es ein wachsendes Forschungsfeld, welches besonders in Open Source Projekten Daten erhebt, unter anderem die Arbeit „A

[16] Software Quality Assessment Based on Lifecycle Expectations
[17] https://www.sonarsource.com/products/sonarqube/

large-scale empirical study on self-admitted technical debt" von 2016 [9], welche 159 Open Source Projekte untersuchte. Die Forscher Bavota et al. zeigen auf, dass über die Zeit die Menge an SATDs zunimmt und zudem einzelne SATD relativ lange existieren (über 1000 Code Änderungen). Diese Erkenntnisse zeigen die Wichtigkeit von Werkzeugen und Techniken, die einen effizienten Umgang mit technischen Schulden ermöglichten. Neuere Arbeiten untersuchen vor allem die automatisierte Erkennung von SADTs, teils mit Unterstützung von Machine-Learning-Methodiken [110, 132, 44, 115, 96, 30].

Eine andere Forschung im Umfeld von Open Source Projekten ist die Arbeit von Alfayez et al. aus dem Jahr 2018 [3]. Sie untersuchten über 19000 Codeänderungen aus 38 Apache Jave Systemen anhand verschiedener statischer Analysen. Aus ihren Daten geht hervor, dass Entwickler unterschiedlich viele technische Schulden verursachen. Darüber hinaus stellten sie fest, dass erfahrene Entwickler weniger technische Schulden ansammeln.

Eine sehr neue Veröffentlichung von Li et al. führte 2022 eine der ersten explorative Fallstudie über SATD im industriellen Umfeld aus [75]. Ihre Ergebnisse erstrecken sich über (übersetzt aus dem Englischen):

- Die Hauptmerkmale von SATD in Industrieprojekten;
- die Einstellung der Entwickler zu identifizierten SATDs und Statistiken;
- Auslöser für Praktiker, SATDs einzuführen und zurückzuzahlen;
- Beziehungen zwischen SATDs in verschiedenen Quellen;
- Praktiken zur Verwaltung von SATDs;
- Herausforderungen und Ideen für Werkzeuge für das SATD-Management.

Die Forscher Ampatzoglou et al. stellen 2018 in ihrem Paper ein Framework vor, welches den Umgang mit Zinsen, die durch technische Schulden entstehen, beschreibt [4]. Dieses validieren sie in einem kleinen bis mittleren Unternehmen aus Griechenland. Sie beschreiben ihre Absicht, den Fokus der Forschung auf den Umgang mit Zinsen zu erweitern und listen in ihrer Arbeit weitergehende Forschungsfragen auf. In ihrer Arbeit gehen die Forscher auf bisherige Forschungen zum Thema der Zinsen ein und leiten mehrere Formeln aus diesen ab, um Kennzahlen für unterschiedliche Arten von technischen Zinsen zu berechnen.

Eine vollkommen andere Sichtweise auf technische Schulden veröffentlichten 2022 die Forscher Siavvas et al. [116]. Sie untersuchten die Zusammenhänge von technischen

Schulden und die Vorhersage von Sicherheitslücken. Hierzu nutzen sie die Daten von 210 Open Source Java Applikationen, um mit diesen verschiedene Machine Learning Models zu erzeugen. Diese evaluierten sie anhand eines Benchmark Datensatzes von OWASP[18]. Nach den ersten Ergebnissen kommen die Autoren zu dem Schluss, dass Indikatoren für technische Schulden auch ein Hinweis auf Sicherheitslücken geben können und dies eine potentielle Möglichkeit ist für Sicherheitsüberprüfungen von Software.

Einen Hinweis, dass die Erkennung von technischen Schulden anhand von SonarQube Funden nicht immer funktioniert, ist in der Arbeit von Pina et al. von 2022 zu finden [101]. Sie untersuchten die Priorisierung von technischen Schulden in Open Source Projekten. Dazu untersuchten sie Open Source Projekte mit mindestens drei Entwicklern. Zunächst wurden diese Projekte mit dem statische Codeanalysetool SonarQube untersucht und anhand der Funde Fragebögen an die Entwickler verschickt, um die Funde zu validieren. Die gesammelten Daten aus den Fragebögen wurden anschließend mit der Grounded Theory Methodology analysiert.

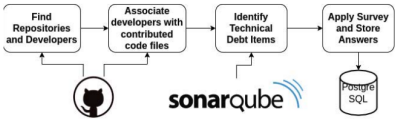

Abbildung 7: Datensammlung [101]

Die Aussagen der Befragten wurden in fünf Super-Kategorien zusammengefasst. Unter diesen Super-Kategorien befinden sich drei, die für das Nicht-Zurückzahlen von technischen Schulden stehen.

- PROJECT_SPECIFIC_DECISION
- PROBLEM_WITH_RULE
- UNUSED_CODE

Problematisch ist hier, dass diese drei Kategorien bereits darauf hindeuten, dass es sich bei den gefundenen Stellen nicht um technische Schulden handelt, entweder sind die Regeln nicht anwendbar, der Code nicht in Verwendung oder es gab eine explizite, projektspezifische Entscheidung, die Code-Änderung nicht vorzunehmen.

[18] https://owasp.org/www-project-benchmark/

4.3 Geschäftsgetriebene Priorisierung

2018 legten Rebouças de Almeida et al. den Grundstein für ein neues Framework zur geschäftsgetriebenen Priorisierung von technischen Schulden [108]. Sie untersuchten anhand einer Mehrfachfallstudie, wie zwei Großunternehmen mit technischen Schulden umgehen. Anhand ihrer Erkenntnisse empfehlen sie die Verwendung von Business Process Management (BPM)[19]. In dem darauf folgenden Jahr stellen sie das Framework „Tracy" vor [107]. Dieses sieht vor

1. wichtige Geschäftsschritte zu identifizieren,
2. Systeme und Codebereiche zu identifizieren, die diese umsetzen oder unterstützen,
3. diese Systeme und Codebereiche mit sie betreffenden technischen Schulden zu verknüpfen,
4. die Priorisierung eines Geschäftsschritts auf verknüpfte technische Schulden zu übertragen.

Dies fügt technischen Schulden eine weitere Priorisierungsdimension hinzu, die dabei helfen kann, den Umgang mit technischen Schulden mit Geschäftszielen in Einklang zu bringen. Dieses Framework wurde bereits in zwei Fallstudien evaluiert [28, 26].

Aus den gesammelten Daten der ersten Fallstudie kondensierten sie acht verschiedene Faktoren heraus, die für die Geschäftspriorisierung entscheidend waren:

- Kern Funktionalität,
- Nutzungsfrequenz,
- Risikowahrnehmung,
- zeitliche Verhandelbarkeit,
- Ausfallursachen,
- Lieferfrequenz,
- Eingebettete Systeme für Zertifizierungen,
- Kunden- und Marketingwahrnehmung gegen Funktionalität.

[19] Modellierung, Analyse und Optimierung von End-to-End-Geschäftsprozessen [48]

In ihren Beobachtungen stellte sich heraus, dass sich die Geschäftspriorität häufig stark von einer rein technischen Priorisierung unterscheidet. Daher sei die Priorisierung nach geschäftsspezifischen Kriterien eine geeignete ergänzende Dimension, welche die im Umgang mit technischen Schulden unterstützen kann.

Anzumerken ist, dass die Erkenntnisse der Studien einen interessante Sichtweise zur Priorisierung von technischen Schulden beitragen. Jedoch dient die Metapher der technischen Schulden insbesondere der Verdeutlichung technischer Notwendigkeiten gegenüber den Geschäftsbereichen und es bleibt zu erforschen, ob die Einführung einer geschäftsausgerichteten Priorisierungsdimension sich für den eigentlichen Zweck als dienlich erweist.

4.4 InsighTD

Ein besonders groß angelegtes Projekt war „InsighTD". Dies wurde von Rios et al. ins Leben gerufen, um technische Schulden über Landesgrenzen hinweg zu untersuchen. Der derzeitige Status des Projektes ist jedoch nach der Abschaltung der Seite im Jahr 2023 unklar.

> „InsighTD ist eine weltweit verbreitete Reihe von Industrieumfragen zum Thema technische Schulden. Ihr Ziel ist es, den Stand der Praxis und Branchentrends im Bereich TS zu untersuchen, einschließlich des Status quo, der Ursachen, die zum Auftreten von TS führen, der Auswirkungen ihrer Existenz, wie sich diese Probleme im Softwareentwicklungsprozess manifestieren und wie Softwareentwicklungsteams reagieren, wenn sie sich des Vorhandenseins von Schulden in ihren Projekten bewusst sind." (Übersetzt aus dem Englischen) [104]

Einige der Studien sind im Folgenden aufgelistet:

Die erste Studie mit Erkenntnissen wurde 2020 von Rios et al. veröffentlicht [113]. Sie identifizierten 78 Ursachen und 66 Auswirkungen von technischen Schulden und erweiterten damit das bestehende Wissen über technische Schulden.

Darauf folgte eine weitere Studie von Freire et al. [36], welche die Antworten von 432 Softwarepraktikern aus verschiedenen Ländern[20] auswertete. Aus den Umfragen geht hervor, dass der größte Anteil an Tilgungsaktivitäten im technischen Bereich liegt, während die Gründe technische Schulden nicht zu tilgen zumeist nicht-technisch sind. Den größten Einfluss hatte, den Befragten zu urteilen nach, fehlendes organisatorisches Interesse, sowie Fokussierung auf kurzfristige Ziele und fehlende Zeit.

[20] Brasilien, Chile, Kolumbien und die Vereinigten Staaten

Boris et al. nutzten die Daten dieser Umfrage [99]. Diese werteten sie sowohl qualitativ als auch quantitativ aus und untersuchten vor allem die Tilgung[21] von technischen Schulden. Dabei kamen sie zu dem Schluss, dass die Definition von Praktiken im Zusammenhang mit der Tilgung technischer Schulden eine wesentliche Aktivität für Softwareentwicklungsteams ist. Die Entwicklung gesunder Softwaresysteme, die auch in Zukunft gewartet werden können, setzt voraus, dass die Unternehmen die richtigen Ansätze für die Tilgung technischer Schulden finden.

In einer weiteren InsightTD Befragung mit dem Schwerpunkt auf die Industrie von Costa Rica wurden 148 Software Praktiker befragt [98]. Die Klassifikation der Unternehmen weicht von der europäischen Definition ab, nach dieser wären über die Hälfte der Antworten von Personen aus Großunternehmen gegeben worden.

Die ersten Ergebnisse deuten darauf hin, dass das Konzept technischer Schulden weitgehend bekannt ist, nur ein Drittel der Befragten gab zunächst an, den Begriff nicht zu kennen. Die Arten von technischen Schulden in den berichteten Beispielen waren das Ergebnis von Situationen, die hätten vermieden werden können. Darüber hinaus wurden technische Schulden in etwas mehr als der Hälfte der Fälle überwacht und in den meisten Fällen nicht getilgt. Aktives Schuldenmanagement war kaum verbreitet.

4.5 Forschung in Startups

Weitere Forschungen beziehen sich auf ein alternatives Umfeld wie andere Unternehmensgrößen. Dabei entspricht das Umfeld eines Startups, neu gegründete Unternehmen mit wenig Mitarbeitern und teils negativem Cashflow [105], dem absoluten Gegensatz zu dem eines Großunternehmens. Es gibt einige Arbeiten, welche technische Schulden in diesem Umfeld untersuchen [29, 63, 14, 21]. Hervorzuheben ist die Arbeit „Umgang mit technischen Schulden im Startup-Umfeld" von Ververs [129], welche ebenso wie die eigene die Grounded Theory Methodology verwendet, um den Umgang mit technischen Schulden in Startups zu erforschen. Aufgrund der Überschneidung in der Methodik und dem entgegengesetzten Umfeld eignet sich diese Arbeit besonders, um zu prüfen, welchen Einfluss das Unternehmensumfeld auf Erkenntnisse in Bezug auf den Umgang mit technischen Schulden haben könnte. Aus diesem Grund werden die Thesen und Ergebnisse der Arbeit in der Diskussion (8.2) anhand der eigenen Daten evaluiert.

[21] engl. „Payment" bzw. nach Li et al. „Repayment" [76]

4.6 Weitere Forschung

Es soll an dieser Stelle nochmal explizit auf die Quellen aus dem Kapitel 2 hingewiesen werden. Zudem ist 2019 von den Forschern Kruchten, Nord und Ozkaya mit dem Buch „Managing Technical Debt: Reducing Friction in Software Development" eine umfassende Zusammenfassung samt Referenzen auf Fallstudien veröffentlicht worden [65].

4.7 Schwerpunkte der gefundenen Arbeiten und Lücken in der Forschung

Die aufgeführten verwandten Arbeiten erwähnen durchgehend, dass die Definition von technischen Schulden aus der Forschung deutlich vom Verständnis der Praktiker abweicht. Zudem unterscheiden sich die Verständnisse der Praktiker selbst untereinander. Dieser Umstand erschwert die Erforschung des Themengebiets. Besonders stark davon betroffen sind Interviews und Umfragen. Antworten auf Fragen können erheblich variieren, je nachdem welche Definition die Befragten im Kopf haben. Zudem herrscht ein Ungleichgewicht zwischen den Forschenden und den Befragten, da sich der Forschende zumeist explizit mit der Definition, Bedeutung und Abgrenzung des Begriffes der technischen Schulden auseinandergesetzt hat. Eine auf Metriken oder Machine Learning basierende Definition von technischen Schulden ist auch problematisch, da diese häufig nicht die Probleme der Praktiker vollumfänglich widerspiegelt und somit nur eine Annäherung ist.

Darüber hinaus fällt auf, dass sich die bisherigen Forschungsarbeiten zumeist auf Umfragen, Interviews oder Code Analysen stützen. Zudem kommen in neueren Arbeiten tatsächlich Ansätze dazu, die maschinelles Lernen verwenden, um Kommentare oder Dokumentation zu analysieren. Nahezu vollkommen fehlt die Perspektive aus der Praxis, in dem ein Forscher in dem Umfeld arbeitet und so über einen längeren Zeitraum selbst Teil der Praxis wird. Dieses Vorgehen entspräche auch den Grundideen der Grounded Theory Methodology [41], welche explizit darauf eingehen, dass der Forschende den Alltag untersucht.

Diese Kritik üben auch die Forscher Nielsen et al. in ihrem systematischen Literaturreview zum Thema „Umgang mit technischen Schulden" von 2020. Nach ihrer Recherche nutzt kein einziges Paper Daten, die auf Beobachtungen basieren [93].

Es ist somit unklar, ob die Ergebnisse aus Interviews, Umfragen und Code Analysen tatsächlich die Entscheidungen eines Entwicklungsteams beschreiben. Aus diesem Grund ist ein regelmäßiger Abgleich beider Felder wichtig, es sollte eine tiefgehende

Datenerhebung genutzt werden, die Beobachtungen direkt im Entwicklungsalltag erhebt und somit weniger beeinflusst ist durch die verschiedenen Wahrnehmungen und Verständnisse von technischen Schulden.

Aus diesen Gründen kann diese Arbeit, durch den Zugang zu exakt diesen Daten, einen Mehrwert zur Forschung von technischen Schulden und dem Umgang mit diesen erbringen.

5 Methodik

Der „Umgang mit technischen Schulden” ist ein relativ neues Forschungsfeld, welches besonders seit 2008 in den Fokus gerückt ist. Mit dem Buch „Managing Technical Debt” schrieben Kruchten et al. 2019 [65], nach dem Vorworten von Grady Booch, „ein unglaublich weises und nützliches Buch” über den Umgang mit technischen Schulden.

Das Buch basiert im Besonderen auf den Erkenntnissen vieler qualitativer Studien - ein deutlicher Hinweis darauf, welchen Wert qualitative Studien im Bereich der Erforschung von technischen Schulden haben.

Eine Methodik, welche sich besonders dazu eignet, größere Textmengen (im Sinne der Sozialforschung) zu analysieren und Theorien abzuleiten, ist die „Grounded Theory Methodology”, auch „GTM” oder „GT” abgekürzt. Es ist an dieser Stelle darauf hinzuweisen, dass die „Grounded Theory Methodology” das Vorgehen beschreibt und die „Grounded Theory” (fundierte Theorie) das Ergebnis der Forschungsarbeit ist. Die Begriffe werden aber oft synonym verwendet.

In ihrem Artikel „Socio-Technical Grounded Theory for Software Engineering” geht Rochina Hoda [50] unter anderem auf das Verhältnis zwischen Softwareentwicklung und der „Grounded Theory Methodology” ein. Sie beschreibt, dass besonders der menschliche Faktor in der Softwareentwicklung immer mehr in den Fokus rückt und somit vermehrt die „GTM” eingesetzt wird.

Aus ihrer Sicht eignet sich die GTM vor allem, um tief in ein Interessengebiet einzusteigen, welches noch Lücken zwischen Theorie und Praxis aufweist.

> GT is a deep dive into an area of interest or phenomenon. It is particularly suited to understudied areas, phenomena with gaps between research and practice, and for addressing complex and deep questions of how and why in addition to the what of the phenomenon being studied. [50]

Wie in Kapitel 4 aufgezeigt, gibt es noch wenig Forschung zum Umgang mit technischen Schulden in Großunternehmen. Zudem greifen existierende Arbeiten nur auf wenige und meist Interview-basierte Daten zurück. Aus diesem Grund bedarf es weiterer qualitativer Forschung, die tiefergehende Datenquellen verwendet.

Vor dem beschriebenen Hintergrund, dass sich die GTM besonders gut für die qualitative Sozialforschung eignet und das Verhalten und die Entscheidungen der Beteiligten für den Umgang mit technischen Schulden von hohem Interesse sind, wird die Forschungsarbeit mit der GTM durchgeführt.

5.1 Ziel der Forschung

Die Forschungsarbeit soll einen weiteren Einblick in den Umgang mit technischen Schulden geben. Aufgrund der offenen Forschungslage und dem vorhandenen Kontext des Großunternehmens soll untersucht werden, welche Unterschiede zu anderen Umgebungen bestehen. Von besonderem Interesse ist, ob sich die prozessgetriebene Handlungsweise von Großunternehmen im Umgang mit technischen Schulden niederschlägt.

Die Erkenntnisse dieser Arbeit können zum einen helfen, Empfehlungen für den Umgang mit technischen Schulden in Großunternehmen zu geben. Zum anderen bietet sich die Möglichkeit, durch Vergleiche mit ähnlichen Arbeiten in anderen Kontexten Rückschlüsse auf den Einfluss der Unternehmensgröße und deren Charakteristika auf den Umgang mit technischen Schulden zu ziehen.

5.2 Vorgehen

Der Autor ist Software-Entwickler in einem Großunternehmen und hat daher die Gelegenheit, tiefe Einblicke in den Arbeitsalltag der Softwareentwicklung zu gewinnen.

Zunächst werden die möglichen Datenquellen untersucht und geprüft, welche sich aus Firmenperspektive nutzen lassen, ohne dass durch Anonymisieren die Nützlichkeit der Daten verloren geht.

Über einen längeren Zeitraum werden anschließend Daten gesammelt und aufbereitet. Anhand der beschriebenen Vorgehen der GTM werden die Daten analysiert. Rückschlüsse aus der Analyse und Theoriebildung werden genutzt, um weitere Daten zu sammeln. Neue gesammelte Daten sollen kontinuierlich mit den bisherigen Analyseergebnissen abgeglichen werden. Anhand dieses Vorgehens können die gebildeten Theorien angepasst oder verworfen werden, um im Verlauf der Forschungsarbeit eine „Grounded Theory" (fundierte Theorie) zu erarbeiten.

5.3 Grounded Theory Methodology

Die GTM existiert in unterschiedlichsten Formen bereits seit über 50 Jahren. Mit dem Buch „The Discovery of Grounded Theory: Strategies for qualitative research" verfassten Glaser, B.G. and Strauss, A.L. (1967) die historische Grundform der GTM. [41]

In ihr werden bereits alle grundlegenden Prinzipien der GTM etabliert. Im Verlauf mehrerer Jahrzehnte entwickelten sich verschiedene Formen der GTM, welche sich vor allem in ihren Schwerpunkten und Freiheitsgrade unterscheiden. Diese Arbeit

folgt den grundlegenden Prinzipien von Glaser und Strauss und nutzt Methodiken, welche von Strauss und Corbin näher beschrieben wurden. [24]

5.3.1 Datensammlung

Daten können in unterschiedlichsten Formaten vorliegen und werden in verschiedenen Formen gesammelt, darunter Gespräche und Diskussionen, Texte oder Diagramme.

Glaser et al. wiesen explizit darauf hin, dass alles als Daten verstanden werden sollte, „all is data" aus „The Discovery of Grounded Theory" [41]

Gesammelte Daten müssen vor der weiteren Verarbeitung darauf geprüft werden, ob ein grundlegendes Problem mit der Verwendung aus dem Unternehmenskontext entsteht, hierunter fallen zum Beispiel als „geheim" eingestufte Dokumente oder interne Dokumente, welche durch eine Anonymisierung ihre Aussagekraft verlieren würden.

Auf die möglichen Datenquellen und ihre Verwendbarkeit wird im Kapitel 6 tiefer eingegangen.

5.3.2 Aufbereitung

Die verwendete Software zur Datenanalyse [127] unterstützt ein breites Spektrum an Formaten, darunter Audioaufnahmen und Bilder wie Diagramme. Jedoch bietet die Analyse von Text erhebliche Vorteile. Audiodateien müssen bei jeder Analyse erneut angehört werden und können nur schwierig überflogen werden. Zudem können Textsegmente leichter zusammengestellt werden.

Audiomitschnitte von Interviews und Gesprächen haben einen deutlichen Mehrwert gegenüber Gedächtnisprotokollen oder Mitschriften (siehe Kapitel 6).

Aus diesem Grund wurden, wann immer möglich, Audiomitschnitte von Gesprächen aufgenommen und nachträglich manuell transkribiert. Hierbei ist zu beachten, dass der Aufwand je nach gewähltem Detailgrad stark schwankt. Eine grobe Zusammenfassung kann beim einmaligen Hören der Aufnahme erstellt werden. Jedoch kommt es immer wieder vor, dass gerade spezifische Worte oder Nebensätze den entscheidenden Hinweis geben, um ein neues Konzept zu finden. Der Aufwand für eine wörtliche Transkription benötigt leicht das drei bis vier-fache der Aufnahmelänge, dies steht dem letztendlichen Nutzen entgegen, da zumeist nur ein kleiner Teil dieser Transkripte als Beispiele in der Arbeit verwendet wird.

Um den Nutzen gegen den Aufwand abzuschätzen, wurden verschiedene Ansätze ausprobiert.

- Wörtliche Transkription ist die aufwändigste Form, bei der jedes Wort versucht

wird zu transkribieren, hierbei beeinflussen besonders Gesprächsdisziplin und Audioqualität wie genau jedes Wort extrahiert werden kann. Zudem leidet gesprochene Sprache häufig unter sprunghaften Gedanken oder nie zu Ende geführten Sätzen.

- Grobe Transkription erfasst nur die Bedeutung und gegebenenfalls Stichworte, hierbei wird zumeist nur der Sinn transkribiert.

- Grobe Zusammenfassung nutzt Stichpunkte, um das Gesagte in die besprochenen Themenbereiche einzuteilen. Hierbei wird versucht, besondere Schlagworte, die bei einer späteren Textsuche verwendet werden könnten, aufzunehmen.

- Adaptive Transkription benutzt eine Mischung aus den bisher beschriebenen Ansätzen, wobei der Autor zum Zeitpunkt der Transkription Vermutungen anstellt, welche Bereiche später besonders interessant sein könnten. Diese werden zum Teil wörtlich transkribiert. Passagen, die als uninteressant eingestuft werden, werden nur grob zusammengefasst. Die adaptive Transkription eignet sich auch dazu, Teile einer Audioaufnahme erneut zu transkribieren, um den Detailgrad zu erhöhen.

- Anreicherung von Mitschriften haben sich besonders zum Ende der Forschungsphase als sehr nützlich erwiesen. Durch die bis dahin gesammelte Erfahrung konnten während der Aufnahme stichpunktartige Mitschriften angefertigt werden, die dann grob mit Zeitmarken versehen wurden. Daraus entsteht eine ähnliche Detailtiefe, wie bei der groben Zusammenfassung, jedoch ohne zusätzlichen Zeitaufwand. Dafür wird zusätzliche Aufmerksamkeit bei der Aufnahme beansprucht.

Besonders zu Beginn der Datensammlung wurde ein hoher Detailgrad verwendet, um möglichst genaue und leicht zugreifbare Daten für die initialen Analysen zur Verfügung zu haben.

Besonders zum Ende der Forschung wurde die Detailtiefe stark reduziert, da sich durch die gebildeten Theorien und ausgearbeiteten Kategorien ein höheres Maß an theoretischer Sensitivität eingestellt hatte. Besonders Interviews wurden zuletzt explizit dafür eingesetzt, um zusätzliche Daten für erarbeitete Erkenntnisse zu sammeln und eine breitere Auswahl an Beispielen ins Feld führen zu können.

5.3.3 Kodieren

Um die gesammelten Daten zu analysieren, bringt die GTM einige Werkzeuge mit.

Die qualitative Datenanalyse beginnt mit dem Kodieren der ersten Daten. Dies beschreibt einen Vorgang, welcher aus Texten Codes, Kategorien und Konzepte abstrahiert, die anschließend zur Theoriebildung genutzt werden.

Open Coding (offenes Kodieren) bildet den Anfang der Datenanalyse, hierbei werden die Daten/Texte zum Teil Zeile für Zeile mit Etiketten, den „Codes", versehen. Dieses Vorgehen wird durch Glaser auch „line-by-line coding" genannt. [40] Die Codes können weiter zusammengefasst und/oder mit Informationen versehen werden, bis ein „Konzept" entsteht. Diese Konzepte werden später mit Eigenschaften ausgestattet und in Relation gesetzt, sodass aus ihnen Kategorien entstehen.

Axial Coding (axiales Kodieren) wird genutzt, um die kodierten Daten in Relation zu setzen. Der Forschende konzentriert sich dabei auf eine Kategorie und sucht nach Kontext, kausalen Zusammenhängen, behindernden Bedingungen, Verhaltensmustern und Konsequenzen. In diesem Zusammenhang nennt man das Auftreten des Geschehens einer Kategorie auch „Phänomen".[120]

Bei der Forschung entstehen meist mehrere Kategorien, deren Untersuchung nach Erfolgswahrscheinlichkeit priorisiert werden kann.

Selective Coding (selektive Kodierung) kann genutzt werden, sobald die ersten Kategorien bekannt sind. Der Forschende wählt zumeist eine Kern-Kategorie und kodiert weitere Daten nur anhand der existierenden Codes. Dies dient der Anreicherung von Beispielen und „sättigt" die Kategorie. [125] Wenn Kategorien nicht weiter angereichert werden können, ist das ein Hinweis, dass sie gesättigt sind, an falschen Stellen gesucht werden, oder die Kategorie verworfen werden kann. Das Ziel dieser Kodierung ist die „Grounded Theory" (fundierte Theorie).

An dieser Stelle ist darauf hinzuweisen, dass nach der ersten offenen Kodierung die Arbeitsmodi der Kodierungen sich in unbestimmter Reihenfolge abwechselnd und manchmal direkt ineinander übergehen. Dieses Vorgehen beansprucht die meiste Zeit der Forschung.

5.3.4 Memoing

Unter „Memoing" wird in der GTM verstanden, ausführliche Notizen („Memos") zu schreiben. Codes, Textstellen, Konzepte und Kategorien sollten mit Notizen versehen werden, um den aktuellen Wissensstand des Forschenden zu dokumentieren. Die den

Kategorien zugeordneten Memos werden meist so stark erweitert, dass diese sich als Grundlage für die schriftliche Arbeit eignen. [125]

> Unlike field and observational notes that are focused on documenting full accounts of observed practice in a single instance, memoing enables the researcher to explore emerging concepts, potential relationships between concepts compared across instances, and identify gaps in the emerging theoretical structures. [125]

5.3.5 Theoretical Sampling

„Theoretical Sampling", zu Deutsch „theoretisches Stichprobenziehen", betrifft das Sammeln weiterer Daten. Anhand der derzeitigen Erkenntnisse und benötigten Informationen werden Lücken identifiziert. Der Forschende trifft dann Annahmen, an welchen Stellen Daten zu finden sein müssten, die die Lücken füllen bzw. die Erkenntnisse weiter stützen. Anschließend werden an diesen Stellen Stichproben gezogen. Wenn die erwarteten Informationen vorhanden sind, können die Lücken gefüllt oder die bisherigen Ergebnisse bestätigt werden. Falls die Informationen nicht gefunden werden, gibt dies Rückschlüsse auf mögliche Fehler in den Annahmen.

5.3.6 Iterieren

Hauptaugenmerk der Methodik ist der iterative Ansatz. Daten werden gesammelt und kodiert. Anschließend werden die neuen Codes und Erkenntnisse mit den existierenden Daten abgeglichen. Hierbei können wiederkehrende Ereignisse, auch genannt „Phänomene", identifiziert werden. Basierend auf diesen Phänomenen werden Konzepte erarbeitet, welche auch anhand neuer Daten überprüft werden. Dieses Vorgehen wird in der GTM auch „kontinuierliches Vergleichen" genannt.

6 Erhobene Daten

6.1 Datenquellen

Um den Umgang mit technischen Schulden in Großunternehmen zu untersuchen, gibt es viele Bereiche, die als Datenquelle in Frage kommen.

6.1.1 Interviews mit Entwicklern mit Audioaufnahme

Die Audioaufnahme von Interviews mit Entwicklern eignet sich besonders, um gezielt an Informationen zu kommen. Durch vorbereitete Fragen kann genauer auf Themen eingegangen werden, zudem eignen sich Fragen häufig als Eisbrecher, um einen Redefluss zu ermöglichen, in dem die befragte Person Beispiele anführt, welche thematisch zu den erfragten Themen passen. Ein Nachteil bei dieser Art der Datenerhebung ist die Befangenheit der Befragten. Das Eingestehen von Fehlern in einem Interview mit Audioaufnahme fällt nicht immer leicht und wird häufig eher im Nebel gelassen. Generell eignen sich Interviews gut dazu, Gefühle und Blickwinkel der Befragten einzusammeln, jedoch dürfen diese nicht als Fakten verstanden werden. Die Einblicke spiegeln allein die Sichtweise der Befragten auf diese Daten wider. [95, 111]

6.1.2 Gedächtnisprotokolle von Gesprächen

Gedächtnisprotokolle von Gesprächen mit Entwicklern bergen vergleichbare Vor- und Nachteile wie Interviews mit Audioaufnahme. Jedoch können diese Art von Unterhaltungen zu jeder Zeit ohne große Aufwände durchgeführt werden. Beispiele hierfür sind die Kaffeeküche oder auf dem Weg zum Mittagstisch. Durch den Wegfall der Audioaufnahme entfällt gänzlich die angespannte Atmosphäre, in der die Befragten besonders auf ihre Wortwahl achten.

Beispiel aus einem der Interviews:

> „Du nimmst es jetzt zwar auf, aber es ist trotzdem unter uns, aber ich sag mal so, [...]"
> (Interviewpartner)

Ein entscheidender Nachteil an dieser Interview-Art ist, dass der Autor erst nachträglich ein Gedächtnisprotokoll erstellt. Somit werden Informationen in verkürzter Form zusammengefasst, zudem werden die Informationen durch den Protokollierenden gefiltert und teilweise vergessen.

Eine nachträgliche Suche nach zuvor kaum beachteten Aspekten ist höchst ungenau.

6.1.3 Pairing

Im Pairing arbeiten zwei Entwickler (das Pair) gemeinsam an einer Aufgabe. Aufgrund der Home-Office-Regelungen der Firma werden diese zumeist remote durchgeführt unter Verwendung einer entsprechenden Software.

Bei dieser Form der Datenerhebung kann es zwei mögliche Zusammenstellungen geben. Entweder der Autor ist Teil des Pairs und arbeitet somit selbst mit an der Aufgabe, oder er ist Zuschauer und nimmt als Unbeteiligter das Gespräch auf. Aufgrund der fachlichen Nähe des Autors zur Softwareentwicklung und der sonst beratenden Tätigkeiten wurde der Autor immer wieder von den beiden begleiteten Entwicklern eingebunden, um die zusätzliche Expertise zu nutzen. Dies entspricht eher einem Pairing mit drei Personen, welches „Mobbing" genannt wird und eher der ersten Zusammenstellung gleicht.

Über den Verlauf von mehreren Stunden arbeitet das Pair zumeist an Programmierproblemen, welche aus einer vorher definierten Aufgabe abgeleitet werden. Hierbei sind die interessanten Momente, welche den Umgang mit technischen Schulden betreffen, ein geringer Anteil des Prozesses.

Zur Aufbereitung von Audioaufnahmen benötigt das im Kapitel Methodik 5 beschriebene Vorgehen den ein bis dreifachen Aufwand an Zeit. Selbst wenn ein Großteil der Aufnahme uninteressant ist, so benötigt eine Aufnahme von drei Stunden also mindestens drei weitere Stunden, um die interessanten Passagen zu extrahieren. Dieses Vorgehen kann optimiert werden, indem während der Aufnahme Marker gesetzt werden. Dies bringt jedoch andere Nachteile mit sich. Die Aufmerksamkeit des Aufnehmenden ist beansprucht, zudem war aus organisatorischen Gründen die Wahl der Aufnahmesoftware begrenzt und bot keine Unterstützung für das einfache Setzen von Markern während der Aufnahme. Zudem ist die Audioaufnahme von Pairings eine zusätzliche Quelle von Unbehagen für alle Beteiligten, da sie sich „beobachtet" fühlen und sich nicht trauen frei, zu sprechen.

Aufgrund des schlechten Kosten-Nutzen Verhältnisses und der sozialen Nachteile, wurden selektive Aufnahmen bevorzugt. Diese wurden gestartet, sobald der Autor erkannte, dass ein Gespräch das Thema der *technische Schulden* streifte. Dieses Vorgehen verursachte zwar eine kurze Unterbrechung in der Diskussion, erwies sich aber selten als störend.

Darüber hinaus hatten die Teammitglieder die Möglichkeit, den Autor dieser

Arbeit hinzu zu rufen, sobald sie davon ausgingen, dass ihr Gespräch interessant für diese Arbeit sein könnte.

Die Aufnahmen wurden während regulärer Arbeitszeit des Autors erstellt, aus diesem Grund sind die selektiven Aufnahmen von Pairings auf das Team des Autors begrenzt.

6.1.4 Besprechungen

Die Aufnahme von Besprechungen bietet sich vor allem an, falls diese die Dokumentation und Priorisierung von Aufgaben betreffen. In dem Team des Autors sind diese das Pre-IPM und IPM. (3.4)

Aufgrund der theoretischen Dichte an interessantem Material wurden diese zu Beginn in voller Länge aufgenommen. Jedoch zeigte sich, dass gerade Aufgaben, die einen Wert für den Kunden beinhalten, den größeren Anteil an Zeit einnahmen, jedoch eher uninteressant für diese Arbeit sind. Aus diesem Grund ging der Autor dazu über, selektive Audioaufnahmen und Gedächtnisprotokolle anzufertigen.

Eine weitere Quelle von Informationen, die besonders die Stimmung im Team betrifft, ist die wöchentliche Retrospektive. Diese gilt als besonders schützenswerter Raum, welcher von den Team-Mitgliedern dazu genutzt wird, persönliche Gefühle zu teilen. Hierbei wird zumeist ein Format genutzt, welche grundlegend die Geschehnisse der Woche in die Kategorien „positiv", „neutral" und „negativ" aufteilt. Aus diesen werden Aufgaben, sogenannte „Action Items", abgeleitet, die das Ziel haben, positive Punkte zu verstärken und negative abzubauen.

Aufgrund der sensiblen Informationen sind sowohl Audioaufnahmen als auch Gedächtnisprotokolle nicht erlaubt, da sie dem Zweck der Retrospektive entgegenstehen. Aus diesem Grund gibt es keine Daten, die direkt aus einer Retrospektive stammen. Jedoch kommt es immer wieder vor, dass in Diskussionen und Interviews erwähnt wird, dass die Auswirkungen von technischen Schulden in der Retrospektive angesprochen wurden.

Einige Teams führen während oder am Ende einer Iteration Rückblicke („Reviews") durch. Dies sind Besprechungen, in denen aktuelle Codeänderungen mit dem gesamten Team durchgesprochen werden. Diese Änderungen können *technische Schulden* sowohl tilgen, als auch aufnehmen. Diese Reviews bieten den Kontext über gelösten Aufgaben, den Quellcode und die Abstimmung im Team.

6.1.5 Chatverläufe

Chatverläufe mit Kollegen können einen Ersatz für kurze Interviews darstellen. Für diese Arbeit werden sie zumeist genutzt, um eine konkrete Vermutung zu überprüfen oder eine spezifische zusätzliche Information einzuholen.

Aufgrund der asynchronen Natur von Chats haben die Partner zumeist die Möglichkeit länger nachzudenken und dann eine ausführliche Antwort zu senden. Jedoch kostet dies weitaus mehr Aufwand und Aufmerksamkeit. Aus diesem Grund werden Fragen teils schnell und kurz beantwortet, ohne weiter auszuholen.

Diese Art der Datenerfassung wurde vor allem am Ende der Arbeit eingesetzt, um Annahmen zu überprüfen oder eine explizite Aussage zu einer noch offenen Frage zu erhalten.

6.1.6 Begleitung des *Team S*

Der Autor ist Gründungsmitglieder in dem seit acht Jahre bestehenden *Team S*, welches andere Softwareentwicklungsteams berät und als Wissensmultiplikator sowohl direkte Expertise als auch formale Workshops zu spezifischen Themen anbietet.

Darüber hinaus stellt das Team eigene Services bereit, die grundlegende Bedürfnisse wie ein Versionsverwaltungssystem oder Systemüberwachung abdecken.

Der Autor ist in den Großteil der Entscheidungen im Team eingebunden.

6.1.7 Hospitation in Teams

Dem Autor war es möglich, weitere Teams tageweise zu begleiten. Dies bietet einen ähnlichen Einblick in fremde Teams wie in das Eigene, jedoch besitzt der Autor nur einen reduzierten Kontext und kann nur Ausschnitte aus dem Alltag der Teams einsehen.

Auf das Aufnehmen von Audiomitschnitten ganzer Arbeitstage ist aus den gleichen Gründen wie dem Mitschneiden von ganzen Pairings (3.3) verzichtet worden. Stattdessen wurden vom Autor Gedächtnisprotokolle angefertigt, welche durch niedergeschriebene Notizen unterstützt wurden.

Für die Hospitation wurden vor allem Tage gewählt, an denen Teams planungsintensive Meetings wie das Pre-IPM oder IPM durchführten. Zudem bot die Begleitung von Code Reviews eine Möglichkeit, in die Umsetzung von Aufgaben zu schauen und das Team zu beobachten, wie es mit Refactorings umgeht.

6.1.8 Firmeninterne Informationsseiten

Um existierende Prozesse und Regelungen zu finden, die den Umgang mit technischen Schulden betreffen, wurde das firmeninterne Wiki mit den Begriffen „technische Schulden" und „technical debt" durchsucht.

Diese Quelle eignet sich besonders, um Informationen über etablierte Prozesse oder Anweisungen zu finden, die sich mit technischen Schulden befassen. Zudem bietet sie eine Möglichkeit der Informationsgewinnung, über Prozesse oder Empfehlungen, die in den Teams zwar nicht bekannt sind, jedoch von der Organisation aufgestellt wurden.

6.1.9 Backlogs

Das Backlog ist eine dynamische Aufgabenliste, welche von den Teams verwaltet wird. Die Backlogs der untersuchten Teams sind zumeist aufgeteilt in die Kategorien „Backlog", „In Arbeit" und „Erledigt". Teilweise ist die Kategorie „Backlog" auch noch weiter unterteilt, je nach Planungsgrad und Detailtiefe.

Die Einträge in den Team spezifischen Backlogs unterscheiden sich anhand grundlegender Merkmale.

Kundenwert

Aufgabe betrifft eine Funktionalität, welche einen Wert für den Kunden hat.

Punkte

Die Punkte entsprechen einer geschätzten Größe der Aufgabe, hierbei werden zumeist Aufwand oder Komplexität als Referenz herangezogen.

Das Backlog kann Aufschluss darüber geben, wie Teams technische Schulden dokumentieren und nachverfolgen. Diese Quelle wurde zumeist genutzt, um Referenzen aus Interviews zu prüfen.

6.1.10 Versionsverwaltungssystem

Teams versionieren sowohl den erstellten Quelltext als auch einige technische Dokumente in einer Code-Ablage. In den untersuchten Teams wird flächendeckend die Versionsverwaltung Git eingesetzt. Git bietet die Möglichkeit, Änderungen zu bündeln, zusammenzufassen und kommentiert zentral zu speichern. In dem Talk „Prioritizing Technical Debt as if Time and Money Matters" von A. Tornhill wird darauf hingewiesen, dass Git eine hervorragende Quelle sein kann, um Bereiche mit häufigen Änderungen zu lokalisieren [123].

Die untersuchten Teams nutzen zwar Git, jedoch ist die Untersuchung der Änderungen ohne Wissen über die Codebasis und die zugehörige Aufgabe, welche zur Änderung geführt hat, sehr zeitaufwendig und mit ungewissem Nutzen. Zudem ist der Quellcode von firmeninterner Software besonders geschützt und die Nutzung als Datenquelle für diese Arbeit ist nicht ohne weiteres zulässig.

6.1.11 Teaminterne Dokumentation - Architecture Decision Records

Einige der Teams dokumentieren größere technische Entscheidungen in einer teaminternen Dokumentation. Diese werden unter anderem als „Architecture Decision Records" (ADR) oder nur „Decision Logs" bezeichnet, zu deutsch Entscheidungsprotokolle.

Diese sind neben dem Code im Versionsverwaltungssystem abgelegt und geben Hinweise auf vergangene Entscheidungen des Teams. Diese Dokumente beinhalten neben der Entscheidung auch die Entscheidungskriterien, Alternativen und Teilnehmer der Diskussion.

Die Protokolle sind zumeist Mitschriften, welche durch Diagramme und weitere Informationen nachträglich angereichert wurden. Zudem kann eine Dokumentenvorlage den Teams während der Diskussion als Leitfaden dienen.

Diese Dokumente wurden teils verwendet, um die Historie von Teams in Erfahrung zu bringen und darauf basierend Fragen für Interviews zu gestalten. Zudem bieten die Protokolle Hinweise auf geeignete Beispiele von *technischen Schulden*, welchen verstärkt nachgegangen werden kann.

6.2 Untersuchte Teams

Im Folgenden werden die Teams und Entwickler genannt und charakterisiert, in denen Daten erhoben wurden. Der Autor dieser Arbeit arbeitete während der Forschung in dem *Team S*, welches aus diesem Grund eine besondere Rolle zukommt, da dieses über den gesamten Zeitraum täglich begleitet wurde, zudem ist der gesamte Kontext des Teams bekannt.

6.2.1 Auswahl weiterer Teams

Um weitere Teams zu identifizieren, welche bereit sind, Informationen über ihren Umgang mit technischen Schulden zu teilen, wurde zunächst in verschiedenen Organisationseinheiten das Thema vorgestellt und nach Freiwilligen gefragt.

Auf diese initiale Anfrage meldeten sich 12 Teams mit jeweils einem Ansprechpartner (meist Entwickler), welche grundlegend bereit waren, Informationen zu teilen.

Jedoch wurden früh Bedenken bezüglich des Aufwandes angemeldet.

Unter diesen 12 Teams waren 4 Teams fachlich voneinander abhängig, zudem wurde das *Team E* speziell dafür gegründet, um teamübergreifende technische Belange zu bearbeiten.

Aufgrund dieser Gegebenheiten wurden zunächst von *Team E* Daten über Interviews gesammelt. Im späteren Verlauf gelang es, weitere Teams zu akquirieren. Hierbei wurde darauf geachtet, Teams aufzunehmen, welche unterschiedliche Rahmenbedingungen hatten, um eine diverse Datenbasis zu generieren.

Zu den Diversitätskriterien zählen Organisationseinheit (OE), Projektdauer (Dauer), Anzahl Entwickler (Devs) und Verbindung zu anderen Teams.

Team	OE	Dauer	Devs	Besonderheit
S	O1	8 Jahre	3-8	kein Fachbereich, IT intern
V	O1	10 Jahre	3-8	kein Fachbereich, IT intern
E	O2	4 Jahre	5	Cluster aus 4 Teams, inklusive *F*
F	O2	4 Jahre	5	Cluster aus 4 Teams, inklusive *E*
D	O2	1 Jahre	6	
M	O2	6 Jahre	6	Entwicklerrotation E->M
W	O3	10 Jahre	5-15	4 Migrationen, vorher extern entwickelt
N	O3	2 Jahre	6	

Tabelle 1: Team Spezifika

Die Teams *S* und *V* stellen IT-interne Services bereit und besitzen keine direkte Abhängigkeit zu einem Fachbereich. Diese beiden Teams haben als Kunden andere Entwicklungsteams, welche zumeist für einen Fachbereich Software entwickeln. Dies stellt dahingehend eine Besonderheit dar, dass sie selbst über Zeit und Ressourcen verfügen, jedoch über die Nutzung ihrer Services finanzieren. Teams, die für einen Fachbereich Software entwickeln, werden durch diesen intern finanziert. Somit werden Zeit und Ressourcen vor allem mit dem Fachbereich ausgehandelt. Der Fachbereich wird vertreten durch den „Produkt Owner" (PO).

6.2.2 Begleitetes Team S

Im Verlauf mehrerer Monate wurde das *Team S* begleitet. Diskussionen und Besprechungen, die eine Nähe zum Umgang mit technischen Schulden aufwiesen, wurden durch den Autor dieser Arbeit aufgenommen oder als Gedächtnisprotokoll dokumentiert.

Zu Beginn der Datensammlung wurde zudem planungsintensive Besprechungen wie das Pre-IPM oder IPM aufgenommen.

Die gewonnenen Mitschnitte und Protokolle wurden durch den Zugriff auf das Backlog und den Quellcode angereichert, aufgrund der langen Zugehörigkeit des Autors zum Team ist der gesamte Projektkontext bekannt.

Die im Beobachtungszeitraum erhobene Daten aus *Team S* sind in Tabelle 2 aufgelistet. Diese wurden wie in Kapitel 5 beschrieben transkribiert und für die GTM verwendet.

Datum	Art	Dauer
Gedächtnisprotokoll	2023-07-11	
Audiomitschnitt	2023-06-27	30
Audiomitschnitt	2023-06-12	45
Audiomitschnitt	2023-03-28	60
Audiomitschnitt	2023-01-19	20
Audiomitschnitt	2023-01-18	10
Audiomitschnitt	2023-01-09	10
Audiomitschnitt	2022-11-03	20
Audiomitschnitt	2022-10-05	10
Audiomitschnitt	2022-09-19	30
Audiomitschnitt	2022-09-14	35
Audiomitschnitt	2022-09-13	45
Audiomitschnitt	2022-08-31	10
Gedächtnisprotokoll	2022-08-30	
Gedächtnisprotokoll	2022-08-29	
Audiomitschnitt	2022-07-04	10
Gedächtnisprotokoll	2022-06-29	
Audiomitschnitt	2022-06-27	20
Audiomitschnitt	2022-06-14	10
Audiomitschnitt	2022-06-08	15
Audiomitschnitt	2022-06-02	15
Audiomitschnitt	2022-06-01	5
Audiomitschnitt	2022-05-30	20
Audiomitschnitt	2022-05-18	25
Audiomitschnitt	2022-05-11	15
Audiomitschnitt	2022-05-02	35

Tabelle 2: Audiomitschnitte und Gedächtnisprotokolle aus *Team S*

Team	Anzahl	Dauer	Datum
D	1	20	2023-06-15
E	5	195	2023-02-06, 2022-11-11, 2022-11-11, 2022-07-23, 2022-07-05
F	1	50	2023-07-14
V	1	30	2023-06-30
M	2	60	2023-05-10, 2023-06-12
N	2	70	2023-05-15, 2023-06-15
W	2	80	2022-11-23, 2023-05-11

Tabelle 3: Befragte Teams

6.2.3 Interviews

Die weiteren Teams *D, E, F, V, M, N und W* wurden durch Interviews mit Entwicklern untersucht. Dabei wurden die Interviews zum Teil in größerer Runde mit mehreren Entwicklern durchgeführt. Auf diese Weise konnten sich die Befragten gegenseitig ergänzen und gegebenenfalls auch widersprechen.

Die Interviews beinhalten einen festen Teil, welcher einige Eckdaten über das Team und die Interviewpartner erfragt. Darüber hinaus wurde die Definition technischer Schulden mit dem Verständnis der Befragten abgeglichen. Um den Einstieg in das Thema zu erleichtern, wurde nach dem aktuellen Projektstand und technischen Herausforderungen gefragt. Anschließend öffneten sich die Interviews, um möglichst Raum zu geben für Anekdoten und konkrete Beispiele über den Umgang mit technischen Schulden.

Über den Zeitraum der Datensammlung wurden erneute Interviews durchgeführt. Diese nutzten meist Informationen aus den bisherigen Schilderungen, um den Einstieg zu gestalten.

Die befragten Teams sind in Tabelle 3 und die jeweiligen Interviewpartner in Tabelle 4 gelistet.

Team	Kürzel	Rolle	Jahre Erfahrung
S	*SL*	Entwickler	15
S	*SM*	Entwickler	5
S	*SH*	Entwickler	15
S	*ST*	Entwickler	10
S	*SK*	Entwickler	5
S	*SD*	Entwickler	13
S	*SA*	Entwickler	10
V	*VS*	Entwickler	25
V	*VF*	Entwickler	10
E	*EN**	Entwickler	4
E	*EF*	Entwickler	5
E	*ES*	Entwickler	15
F	*FF*	Entwickler	7
D	*DJ*	Entwickler	23
M	*MM*	Product Manager	23
M	*MX*	Entwickler	10
M	*EN**	Entwickler	4
W	*WS*	Entwickler	8
N	*NT*	Produkt Manager	7
N	*NS*	Entwickler	8

Tabelle 4: Teammitglieder

* Entwickler *EN* wechselte von *Team E* nach *Team M*

6.2.4 Hospitation

Im Verlauf der Datensammlung hospitierte der Autor jeweils für einen Tag in den Teams *Team E* (2023-04-19) und *Team W* (2023-05-16).

In diesen Tagen wurde sowohl an IPMs als auch an Reviews teilgenommen. Darüber hinaus bestand die Möglichkeit, während Pairing-Sitzungen tiefere Einblicke in den Umgang mit technischen Schulden zu bekommen. Hierzu boten besonders kleinere Pausengespräche und Exkurse in andere Codebereiche eine lohnenswerte Datenquelle.

6.2.5 Weitere Quellen

Die Teams *W und E* gewährten jeweils Zugriff auf ihren Backlog, zudem konnten die ADRs des Teams *E* eingesehen werden.

Diese Datenquellen wurden vor allem genutzt, um die Interviews vorzubereiten. Anhand von Auffälligkeiten konnten tiefergehende Fragen abgeleitet werden.

7 Ergebnisse

7.1 Bewusstsein in der Organisation

In verwandten Arbeiten wie „Umgang mit technischen Schulden im Startup-Umfeld" von Ververs [129] geben die Interviews Einsicht über ein breites Bewusstsein von technischen Schulden bis in die Führungsebene der untersuchten Startups. Zudem zeigt sich, dass die Metapher der technischen Schulden ein bekanntes Thema der Softwareentwicklung ist, welches auf Konferenzen und in der Literatur durchaus präsent ist. [34, 35, 65, 123, 135]

Aufgrund des Forschungsstands zu technischen Schulden und der Auswirkung auf die Softwareentwicklung könnte davon ausgegangen werden, dass ein Großunternehmen bereits formalisierte und dokumentierte Prozesse für den Umgang mit technischen Schulden erarbeitet und etabliert hat.[7]

In den Interviews hat sich jedoch gezeigt, dass Entwicklern keine offizielle Handlungsanweisung für den Umgang mit technischen Schulden bekannt ist. Bei der Recherche nach internen Dokumenten konnten jedoch folgende Informationen zu technischen Schulden gefunden werden.

- Software Organisations Wiki: zweiseitige Beschreibung über den empfohlenen Umgang mit technischen Schulden (existiert seit 2020)
- „Solution Architecture Documentation" ist ein verpflichtendes Dokument, welches unter dem Kapitel Risiko eine Auflistung von technischen Schulden vorsieht.
- Beschreibung eines „Technische Reife Modells", welches auch das Thema der technischen Schulden behandelt.

Es gibt also Dokumentationen und Handlungsempfehlungen für den Umgang mit technischen Schulden, dies ist nur kaum bekannt. Somit gibt es in dem betrachteten Bereich des Unternehmens keinen etablierten Prozess. Zudem zeigt sich in einer knappen Befragung der Hierarchie-Ebenen, dass bereits zwei Ebenen über den eigentlichen Entwicklern der Begriff *technische Schulden* unbekannt ist.

Eine weitere Beobachtung ist, dass alle befragten Teams einen Product Owner (PO) als Vertreter des Fachbereichs haben, welche oft keinen technischen Hintergrund besitzen und nur selten den Begriff *technische Schulden* kennen oder die Auswirkungen abschätzen können.
(Interview 2023-05-11 - WS, Interview 2023-05-15 - NT)

Schlussfolgerung

In einer Organisation von Nicht-Software zentrischen Großunternehmen ist auch in technischen Abteilungen und deren Hierarchie das Konzept der *technische Schulden* nicht durchgängig bekannt und beachtet. Dies hat zur Folge, dass kaum Richtlinien oder Empfehlungen zum Umgang mit technischen Schulden etabliert sind. Darüber hinaus sollte bei der Einführung von Handlungsempfehlungen großer Wert auf die Bekanntmachung gelegt werden.

7.2 Begriff „*technische Schulden*" in Softwareentwicklungsteams

In der Forschung ist in dem Dagstuhl Seminar [7] eine neue Definition für technische Schulden aufgestellt worden, diese ist seitdem in mehreren Studien verwendet worden [112, 129, 59] . Neben den beschriebenen Herausforderungen mit dieser Definition gilt sie zudem nicht als Standard, so führten Rios et al. 2020 eine Umfrage in Brasilien durch, die McConnells Definition nutzte [113, 84] Besonders die Unschärfe und irreführende Anwendung des Begriffes für jegliche Art von Problemen in der Softwareentwicklung wird immer wieder angeführt. Im Folgenden soll beleuchtet werden, ob sich dies innerhalb der letzten sieben Jahre seit Veröffentlichung der Definition von technischen Schulden in der Praxis geändert hat.

Nach der gewählten Definition (Kapitel 2.3) beziehen sich *technische Schulden* vor allem auf Konstrukte des Design oder der Implementierung, die einen kurzfristigen Nutzen versprechen, jedoch einen Kontext erzeugen, der Weiterentwicklung, Qualität und Wartbarkeit beeinträchtigt.

Um ein einheitliches Verständnis von technischen Schulden zu schaffen, wurden in allen Erstgesprächen zunächst die Personen befragt, was sie unter technischen Schulden verstehen und im Anschluss wurde die eigene Definition geteilt. Im Anschluss wurde gefragt, ob sie Unterschiede zwischen dem eigenen Verständnis und der geteilten Definition sehen.

Es war zu beobachten, dass das Verständnis in den meisten Fällen vorher weiter gefasst war, so beschreibt der *PM NT*:

> „Technische Schulden sind auch Bugs, die in der Plattform gefunden wurden oder Compliance Funde, ggf. nicht im klassischem Sinne, jedoch technische Aufgaben, die vom Team bewältigt werden müssen, die da drunter fallen." (Interview 2023-05-15 - NT)

Ein Entwickler aus dem gleichen Team sieht unter technischen Schulden vor allem

„technische Mängel, die sich über Zeit einschleifen" (Interview 2023-06-15 - NS).

Andere Entwickler sehen grundlegende Unterschiede:

> „Wobei ich sagen würde, dass ich nicht sicher bin, ob technische Schulden sich nur auf die internen Systemqualität beziehen. Aus dem einfachen Grund, wenn ich Kunde eines Systems bin und das hat technische Schulden, bedeutet das, dass ggf. die externe Systemqualität oder Weiterentwicklung beeinträchtigt ist. Wenn ich Bestandskunde bin, haben technische Schulden ggf. einen Einfluss auf mich als Kunden."
>
> (Interview 2022-11-11 - SD)

In diesem Kontext wird die Frequenz neuer Features als Kundenwunsch herangezogen, wodurch eine verlangsamte Weiterentwicklung einen direkten Bezug auf die Kundenzufriedenheit hat.

Die häufigste Umschreibung, was technische Schulden sind, bezieht sich auf den Unterschied zwischen dem Ist-Zustand und dem besten Weg. Hierbei bezieht sich der beste Weg auf die beste Umsetzung der nun bekannten Anforderungen und dem Wissensstand des Betrachtenden.

Der *Entwickler SL* sieht in technischen Schulden die „Differenz zwischen Ist-Zustand und idealer Ist-Zustand" (2023-06-12 1328 - SL). Dies entspricht einer älteren Definition nach Nugroho: „Der Begriff technische Schuld wird definiert als die Kosten für die Verbesserung der technischen Qualität bis zu einem Niveau, das als ideal angesehen wird" (Übersetzt aus dem Englischen) [94].

Eine Definition, die explizit von Cunningham nicht als technische Schuld angesehen werden sollte, wurde durch einen Entwickler aus dem *Team E* gegeben:

> „Ich verstehe darunter, man implementiert ein neues Feature oder eine Änderung in der Codebasis und entscheidet sich bewusst dazu etwas weniger schön zu lassen mit der Hoffnung, dass man jetzt schneller ist und es später aufräumt. Damit hat man eine Schuld, die man eingegangen ist, die man später aufräumt."
>
> (Interview 2022-07-05)

Cunningham weist explizit darauf hin, dass es nie darum ging, schlechten Code mit der Metapher zu rechtfertigen [133].

Die Beobachtungen lassen sich wie folgt zusammenfassen:

- Starke Abweichungen zwischen Definition der Befragten und der eigenen,

- jedoch auch Überschneidungen mit der Literatur,
- Technische Schulden werden, mit mäßigem Erfolg, an Gefühlen festgemacht,
- Übereinstimmung bei spezifischen Beispielen, wie Aktualisierungen von Abhängigkeiten, welche aufgeschoben werden, um Features zu entwickeln.

Schlussfolgerung
Eine einheitliche Definition von *technischen Schulden* ist in der Praxis nicht etabliert. Entwickler und PMs tendieren dazu, jegliche technische Aufgabe, die nicht explizit auf einen Kundennutzen einzahlt, als technische Schuld zu bezeichnen.

Wie in Kapitel 2.3 beschrieben, wird in dieser Arbeit explizit der Umgang mit Phänomenen betrachtet, die der gewählten Definition von *technische Schulden* entspricht. Zu beachten ist, dass jegliche technische Aufgabe zu einer *technischen Schuld* nach der gewählten Definition werden kann, sobald das Team einen langfristigen Mehrwert in der Umsetzung sieht und erwartet, dass die Nicht-Umsetzung eine *technische Schuld* samt Zinsen erzeugt.

7.3 Identifikation

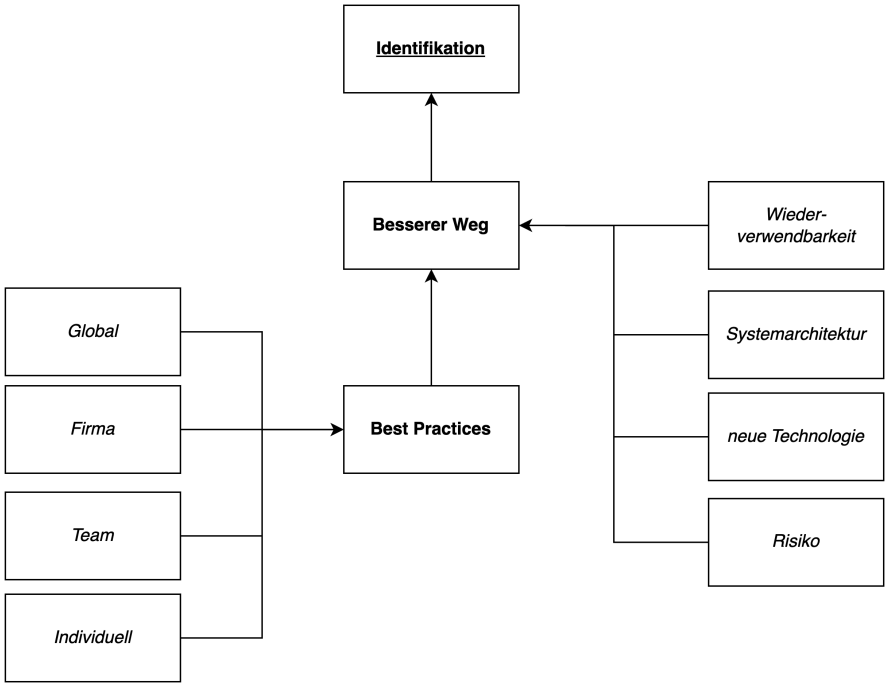

Abbildung 8: Qualitative Konzepte der GTM zur Identifikation von technischen Schulden

In der qualitativen Analyse sind die in Abbildung 8 gezeigten qualitativen Konzepte gefunden worden. Im Folgenden wird zunächst das Konzept des „Besseren Weg" erklärt und anschließend auf das Konzept „Best Practices" eingegangen.

7.3.1 Einfluss subjektiver Sichtweisen auf den Umgang mit *technischen Schulden*

Bewusste *technische Schulden* entstehen immer dann, wenn Teams sich gegen eine Änderung entscheiden, die längerfristig weniger Aufwand für das Team verursachen würde, als wenn man sie unterlässt.

Schaut man sich die Zusammenfassung der Agile Alliance an, so wird Bezug genommen auf „code that is not quite right for the programming task of the moment" (Code, der für die momentane Programmieraufgabe nicht geeignet ist).[74] Die Beurteilung, ob Code für die momentane Programmieraufgabe, also für die Anforderungen, geeignet ist oder nicht, liegt vor allem in den Augen des Betrachters.

Die gewählte Definition technischer Schulden (Kapitel 2.3) spricht von „make a future change more costly or impossible" (machen eine künftige Änderung kostspieliger oder unmöglich). Auch hier ergibt sich ein großer Spielraum für die Entwickler, eine eigene Sicht zu entwickeln, welche Änderungen in der Zukunft wahrscheinlich benötigt werden. Vorhersagen tendieren dazu, ungenauer zu werden, je weiter sie in die Zukunft reichen. [114]

Diese grundlegenden Probleme spiegeln sich auch in der Praxis wider. Entwickler haben unterschiedliche Sichten, welcher Code geeignet ist.

Ein Beispiel ist die Diskussion mit dem *Entwickler VF* - während dieser eine Applikation und Datenbank in einer Infrastrukturgruppe zusammen legen wollte, argumentierte der *Entwickler SA* für eine Trennung dieser beiden Komponenten. Diese unterschiedlichen Ansichten wurden vor allem durch unterschiedliche Bewertung von Risiken und deren Eintrittswahrscheinlichkeiten bedingt.

In den beobachteten Teams wurde vermehrt darauf hingewiesen, dass die Einschätzung der längerfristigen positiven Auswirkung von Änderungen häufig durch Gefühle beeinflusst wird, wenn nicht sogar gänzlich darauf beruht.

So berichtet der *Entwickler EN* im Rückblick auf seine Arbeit in dem *Team E*:

> „Ich habe ganz häufig das Gefühl, dass wir im Team einfach nur etwas empfinden, aber keine Metrik dafür haben"
>
> (Interview 2023-06-14 - EN)

Dies stimmt mit den Ausführungen anderer Teammitglieder überein. Ein anderes Beispiel zeigt, wie sehr die Gefühle der Entwickler von der Realität abweichen können.

Das *Team E* hatte das Gefühl, dass ihre Testabdeckung zu gering sei. Aufgrund der einfachen Möglichkeit dies zu überprüfen, konnten Metriken herangezogen werden. Die Testabdeckung lag bei über 90 Prozent. Daraufhin musste das Team seine Einschätzung korrigieren und ging anschließend davon aus, dass die Testqualität nicht ausreiche, ein Wert, der nicht mehr leicht überprüft werden konnte.

Im Folgenden werden einige der gefundenen Einflüsse auf die Ansichten der Entwickler aufgeführt. Diese beeinflussen die Sichtweise, welche Umsetzungen auf lange Sicht Vorteile bringen und welche eher *technische Schulden* aufbauen würden. Es konnte mehrfach beobachtet werden, dass Entwickler entgegengesetzter Meinung waren, was die Umsetzung des jeweils anderen an Mehraufwand kosten würde und welchen langfristigen Vorteil sie bieten könnte.

Es ist davon auszugehen, dass diese Liste bei weitem nicht vollständig ist, da es sich hierbei um einen komplexen Entscheidungsprozess handelt.

Wiederverwendbarkeit

Ein Einfluss, welcher der geeignete Weg zur Umsetzung einer Anforderung in Code ist, hat die Abwägung über die Wiederverwendbarkeit von Code. Code, der eine wiederkehrende Aufgabe löst, sollte entsprechend gestaltet werden, dass er leicht wiederzuverwenden ist. Grundlegende Anforderungen an wiederverwendbaren Code sind ein nützliches Maß an Abstraktion, stabile API und Unabhängigkeit von weiteren Komponenten („Decoupling")[53]. Wird dieser Code jedoch niemals wiederverwendet, bildet die Erhaltung der Wiederverwendbarkeit einen unnötigen Mehraufwand, eine technische Schuld. Diese entsteht dadurch, dass zum Beispiel eine spätere Erweiterungen oder Änderungen in der Funktionalität mit der Abstraktion vereinbar sein muss ist dies nicht der Fall, muss die Abstraktion angepasst oder in Gänze erneuert werden, dies entspricht den Zinsen der *technischen Schuld*, die für die Weiterentwicklung anfallen.

Ein konkretes Beispiel wurde in dem *Team S* gefunden, welches eine wiederverwendbare Komponente implementierte, in der Erwartung, das gelöste Problem erneut lösen zu müssen. Die Separierung führte im Verlauf des Projektes dazu, dass Mehraufwand bei der Entwicklung weiterer Funktionen entstand.

> Die *Entwickler SA und SH* diskutierten über den Mehraufwand, der durch die strikte Trennung zwischen den Datenbanken entstand. *SH* überlegte, ob es überhaupt sinnvoll wäre, zwei Datenbanken zu haben, wenn eine Softwarekomponente auf beide zugreife, es würde dann auch eine reichen.

> *SA* argumentiert, dass dies sicherlich eine offene Frage sei, er aber die Trennung gerne beibehalten würde, um eine leichte Wiederverwendung in einem anderen Kontext zu ermöglichen. Zudem wäre die Umsetzung, dass die Komponente auf beide Datenbanken zugreifen kann, kein großer Aufwand.
>
> (Pairing 2022-05-18 - SA, SH)

Rückblickend ist die Wiederverwendbarkeit zwar ein Vorteil, jedoch wurde dieser innerhalb eines Jahres noch nicht ausgeschöpft, dies ist auch weiterhin nicht in Aussicht. Interessant ist, dass je nach Auslegung beide genannten Umsetzungen sich als eine *technische Schuld* herausstellen können.

Wird die Trennung der Komponenten reduziert und die Wiederverwendbarkeit vernachlässigt, könnte das den Aufwand sowohl kurz- als auch langfristig einsparen. Solange in der Zukunft die Komponente niemals wiederverwendet wird, wäre die Erhaltung der Wiederverwendbarkeit ein erhöhter Aufwand und kommt der Zahlung von Zinsen gleich. Andersherum würde sich der Aufwand auf lange Sicht lohnen, wenn es tatsächlich zu einer Wiederverwendung kommt.

> „Even an architectural design that makes the system more flexible and adaptable than it really should be, can be a form of TD, if this additional flexibility hinders future development without actually being exploited." [66]

Auf diesen Sachverhalt weisen auch Kruchten et al. hin und nehmen als Beispiel ein Architekturdesign, welches mehr Flexibilität und Adaptierbarkeit bietet als eigentlich nötig, dabei jedoch zukünftige Weiterentwicklung behindert. [66]

Systemarchitektur

Im *Team S* wurde diskutiert, ob mehrere Komponenten auf eine Datenbank zugreifen sollten oder ob dies interne Abhängigkeiten schafft, die die Erweiterbarkeit beeinträchtigen könnten. Die Einschätzung, welche Systemarchitektur die meisten Vorteile birgt, hängt von den Umständen und der einzelnen Gewichtung der Faktoren ab. Für beide Umsetzungen sind Beschreibungen im Internet zu finden [87, 88].

In dem Gespräch zwischen *den Entwicklern VS und SA* vom 2022-05-17 bringt *SA* Bedenken bezüglich der Nutzung einer Datenbank durch verschiedene Komponenten auf. *VS* beschreibt seine Interpretation und argumentiert, dass die Komponenten Teil eines Systems sind, somit eine einzige Datenbank den Anforderungen angemessen ist.

Auch hier finden sich verschiedene Abwägungen, welche Architektur eine geeignete Umsetzung sein kann, in diesem Fall hängt die Interpretation vor allem davon ab, ob die Komponenten als ein System verstanden werden oder nicht. Davon leiten sich die erwarteten Vor- und Nachteile ab, die, nach Erfahrung eines Entwicklers, höchstwahrscheinlich eintreten.

Wissen über bisher unbekannte Technologie

Weiteren Einfluss hat das Wissen eines Entwicklers über bisher unbekannte Technologien. In dem *Team S* wurde eine Implementierung zunächst basierend auf den bereits verwendeten und bekannten Cloud Services gebaut. Der Austausch mit einem Entwickler aus einem anderen Team brachte das zusätzliche Wissen, welche Vorteile ein noch nicht verwendeter Cloud Service bringen würde.

Dieses neue Wissen führte dazu, dass die bisherige Implementierung als technische Schuld angesehen wurde. Der empfohlene Service beinhaltet Funktionalitäten, die zuvor selbst implementiert worden sind, darüber hinaus waren diese bereits als abstrakte Bausteine bereitgestellt. Um den Mehrwert nutzen zu können, musste das Team zusätzlichen Aufwand investieren, um sich in die Verwendung des neuen Service einzuarbeiten. Sobald dieser Aufwand geleistet wäre, könnte langfristig Aufwand in der Wartung gespart und zudem die Erweiterung mit neuen Funktionalitäten einfacher umgesetzt werden.

Deutlich wird hier, dass vor allem das neue Wissen dem Team ermöglichte eine geeignetere Umsetzung als die bestehende zu finden. Ohne dieses Wissen wäre die existierende Lösung dem Zweck angemessen und brächte keinen zusätzlichen Aufwand zu einer anderen Lösung, da sie diese nicht kennen.

Risikobewertung

Eine anderen Einfluss findet man in der unterschiedlichen Bewertung von Risiken. Entwickler bewerten Risiken zumeist durch ihre subjektive Wahrnehmung, die aufgrund individueller Erfahrungen voneinander abweichen.

Ein gefundenes Beispiel hierfür ist die Diskussion zwischen den *Entwicklern SA und VF* über die Aufteilung von „Infrastructure as Code" (IaC) Komponenten.

Die Entwickler SA und VF diskutierten über die Aufteilung, SA sieht es als unnötiges Risiko, persistente und nicht-persistente Komponenten zusammen zu deployen, ein Löschen und Neuerstellen bringt durch nicht gelöschte persistente Komponenten eine Komplexität in ein Szenario, welches während eines Produktionsausfall bereits

ausreichend problematisch sein würde. VF sieht die Eintrittswahrscheinlichkeit so gering, dass er eine Separation der Komponenten für unnötigen Mehraufwand ansieht. (Gedächtnisprotokoll - SA, VF)

Die Entwickler vertreten ihre Meinung in eigenen Teams, hier ist zu beobachten, dass im Team von *SM* die fehlende Trennung als technische Schuld gesehen wird, die früh getilgt werden sollte, da besonders Infrastrukturänderungen nach dem Veröffentlichen eines Services ein hohes Maß an Tilgungskosten mit sich bringen. Dies ist vor allem dadurch zu erklären, dass größere Infrastrukturänderungen häufig nur durch das Löschen und erneutes Erstellen aller Komponenten einer Gruppe durchgeführt werden können. Somit kommt der zusätzliche Aufwand für eine ausfallfreie Umstellung hinzu.

> SA: „Beim Redeploy einer Applikation kann es zur Neuerstellung der Datenbank kommen. Theoretisch müsste man das aufräumen."
> SH: „Gerade ist der Stack nicht aktualisierbar, und wir müssen ihn weg räumen, jetzt wäre es gut, das zu machen."
>
> (Gespräch 2022-05-18 - SA, SH)

In diesem Gespräch stimmt *SH SA* zu, *SH* vertritt ähnliche Ansichten wie SA. In diesem Fall gehen beide Entwickler davon aus, dass es sinnvoll ist, dies direkt anzupassen, statt bewusst technische Schulden aufzunehmen. *VF*, aus dem ersten Beispiel, hätte das sicherlich anders gesehen.

Schlussfolgerung

Die Einschätzung, welche Umsetzung für die Lösung eines Problems geeignet ist oder ob es doch einen besseren Weg gibt, hängt von vielen Einflussfaktoren ab, die sich auf die individuellen Sichten der Entwickler auswirken. Dies beeinflusst zudem, welche Umsetzungen als bewusste Aufnahme von technischen Schulden wahrgenommen werden.

Für einige Einflussfaktoren konnten Beispiele beobachtet werden, jedoch ist aufgrund der Komplexität menschlicher Entscheidungen davon auszugehen, dass weitaus mehr existieren. Eine genauere Untersuchung und Kategorisierung mag sinnvoll sein, ist jedoch nicht Bestandteil dieser Arbeit.

7.3.2 Einfluss von Best Practice auf den Umgang mit *technischen Schulden*

In der Softwareentwicklung gibt es unterschiedlichste Ansichten darüber, wie „richtiger" Code aussieht. Weinberg schreibt in seinem Buch „Quality Software Management" über die Relativität von Qualität: „Was für den einen angemessen ist, kann für den anderen von unzureichender Qualität sein." [134] Im Folgenden wird auf die individuelle Interpretation und Anwendung verschiedener Vorgehensweisen eingegangen.

„Best Practices" zu deutsch „beste Praxis" ist ein aus der Betriebswirtschaft entlehnter Begriff, welcher ein Vorgehen beschreibt, welches allgemein als besser akzeptiert wurde als andere bekannte Lösungen oder weil es die Standardlösung für ein bekanntes Problem ist.

In der Softwareentwicklung schlagen sich *Best Practices* zumeist in sogenannten Software „Pattern" (Entwurfsmuster) nieder, eine wohl bekannte Auflistung von Softwareentwicklungsmustern ist das Buch „Design Patterns: Elements of Reusable Object-Oriented Software" der Gang of Four [37].

Aus den Beobachtungen entwickelte sich während der Analyse aus mehreren Codes eine Kategorie *Best Practices*. Diese ist an die oben beschriebene Bedeutung angelehnt und trägt deswegen den gleichen Namen.

In den Interviews und weiteren Daten finden sich Referenzen auf erwähnte Vorgehensweisen, welche das Team ohne weitere Diskussion als *Best Practices* akzeptiert. Hierbei werden Abweichungen von *Best Practices* als Hindernis für die Weiterentwicklung oder Wartung des Systems angesehen. Die Abwägung, ob ein Codebereich umgebaut werden soll, um sich nach den *Best Practices* auszurichten, wird in dem Moment zu einer technischen Schuld nach der Definition, wenn das Team davon ausgeht, dass die Anpassung eine bessere Erweiterbarkeit oder Wartbarkeit verspricht und eine Unterlassung einer Aufnahme von technischen Schulden gleicht, welche wahrscheinlich Zinsen verursachen wird.

Während der Analyse wurde explizit darauf geachtet, dass nur Beispiele untersucht wurden, bei denen die Auswirkungen sich besonders auf die interne Software Qualität beziehen und kaum einen Einfluss auf den Kunden haben.

Im Folgenden werden zunächst vier Ausprägungen mit unterschiedlichen Eigenschaften beleuchtet, um anschließend ihre Wirkung auf den Umgang mit technischen Schulden zu erläutern.

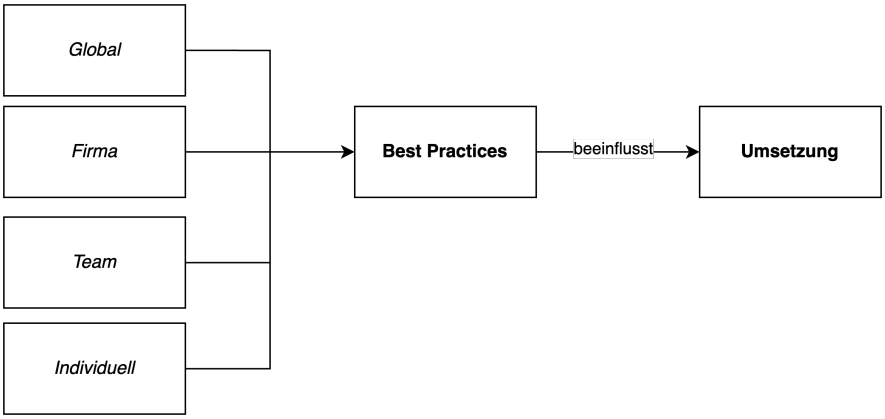

Abbildung 9: Best Practices

Von den Beobachtungen in den Teams konnten vier Ausprägungen von *Best Practices* abgeleitet werden.

Globale Best Practices

Diese Vorgehen stammen aus Büchern, Dokumentationen oder aus Blogs bekannter Entwickler. Beispiele sind die zuvor erwähnten Software-Entwurfsmuster oder auch die Dokumentation von Software Bibliotheken oder Werkzeugen.

Während sich Entwurfsmuster seltener ändern, kann sich die empfohlene Nutzung von Bibliotheken von Version zu Version ändern.

Ein Hauptmerkmal von *globalen Best Practices* ist, dass sie dem Entwickler bekannt sein müssen, diese müssen sich also zumeist aktiv informieren. Zudem ist die Nutzung dieser *globalen Best Practices* keine Pflicht. Zumeist sind es einzelne Entwickler, die eine Expertise mitbringen oder ausbilden und die neuen Informationen in das Team tragen.

Ein Beispiel für diese Art ist ein interner Workshop, der sich an einem Fachbuch orientiert und die dort vorgestellten Best Practices an Teams weitergibt (interner Workshop DynamoDB).

Firmen Best Practices

Bei *Firmen Best Practices* handelt es sich zumeist um feste Vorgaben oder interne Handlungsempfehlungen. Beispiele sind Compliance-, Dokumentations- oder Sicherheitsempfehlungen, die auf eine spezifische Weise umgesetzt werden sollen. Das Unternehmen besitzt teilweise Mechanismen, um die Umsetzung zu prüfen. Diese Prüfungen sind jedoch zumeist nur oberflächlich. Zudem sind die *Firmen Best Practices* nicht immer bekannt oder werden als nicht angepasst oder zu generell angesehen und deswegen ignoriert oder ihre Umsetzung wird

auf spätere Zeit verschoben. Es kommt immer wieder vor, dass die Organisation einen besonderen Fokus auf einzelne Anweisungen legt, in diesen Momenten durchlaufen Teams häufig eine Stresssituation, da sie gezwungen werden, die Anforderungen schnellstmöglich umzusetzen. Teams orientieren sich zumeist an den Umsetzungen anderer Teams, um Aufwand zu vermeiden.

Beispiele hierfür sind firmeninterne Vorlagen, die im Besonderen technische Umsetzungen von internen Regularien oder Sicherheitsregelungen beinhalten (Interview 2023-06-15 - NS).

Team Best Practices

Diese Art von *Best Practices* bezieht sich auf Vorgehen, welche im Team abgesprochen wurden, wie zum Beispiel der Einsatz von Pair Programming oder Code Reviews. Diese können auch *globale, firmen oder individuelle Best Practices* sein, welche das Team übernimmt. Beispiele hierfür sind Architecture Decision Records oder die Übereinkunft über die Art und Weise, wie Tests implementiert werden. *Team Best Practices* können sich im Besonderen durch den Wechsel von Personen ändern.

Einige Teams dokumentieren ihre *Best Practices* auch, um gerade neuen Entwicklern einen leichteren Einstieg ins Team zu ermöglichen. Weitere Beispiele sind zudem teaminterne Bibliotheken, welche Komponenten bereitstellen für wiederkehrende Herausforderungen. Ein Beispiel hierfür wäre eine Bibliothek aus dem *Team S*, welche Komponenten für Testumgebungen, Anmeldeabläufe und Validierungen bereitstellt.

Individuelle Best Practices

Individuelle Best Practices bezeichnet individuelle Vorgehen, welche durch Erfahrung einzelner Personen als persönliche *Best Practices* abgeleitet wurden. Diese können auch Interpretationen von zum Beispiel *globalen Best Practices* sein. Sie können sich auf die Arbeitsweise oder die Verwendung von Technologien beziehen. Sie können *Team Best Practices* beeinflussen oder diesen auch entgegengesetzt sein. In einem solchen Fall kann es zu häufig auftretenden Diskussionen kommen, in denen der Entwickler mehrfach versucht, das Team von seinen Ansichten zu überzeugen. Dies kann zu Unzufriedenheit bei dem Entwickler und/oder dem restlichen Team führen.

Eine dieser Situationen beschrieb der *Entwickler DJ*, welcher persönliche Best Practices für Architekturdokumentation mitbringt und auch darüber berichtet „Leute sind nicht glücklich wenn ich meckere" (Interview 2023-06-15 - DJ). Auch

im *Team S* kommt es im Alltag immer wieder zu unterschiedlichen Meinungen zu Best Practices. Ein Beispiel hierfür war eine Diskussion während einer Pairing Session der Entwickler *SA und SD*, welche unterschiedliche Ansichten vertraten, ob spezifische Filter als Absicherung in der Persistenzschicht eingebaut werden sollten oder nicht. Während *SD* sich strikt daran halten wollte keinen unnötigen Code einzubauen, der im Moment nicht benötigt, ggf. nicht einmal getestet werden würde, hielt der Entwickler *SA* es für besser, jegliche Anfragen mit Filtern abzusichern, da dies, seiner Erfahrung nach, erheblichen Nutzen im späteren Verlauf der Entwicklung erzeugt.

Best Practices haben einen markanten Einfluss auf die Sicht des Entwicklers auf das System und auf den besten Weg, wie die Anforderungen des Projektes umzusetzen sind, um die Erweiterbarkeit und Wartbarkeit des Systems zu gewährleisten.

Im Verlauf der Entwicklung lernen Teams dazu und entwickeln neue *Best Practices*. Wenn ein Team in acht Komponenten eine Systemüberwachung einbaut, werden nach den ersten Umsetzungen Ideen aufkommen, wie die Aufgabe besser gelöst werden könnte. Nun steht das Team vor der Entscheidung, ob sie die bereits umgesetzten Lösungen erneut anpassen, um einen Mehrwert aus dem Gelernten zu ziehen, alternativ können sie, um den Mehraufwand zu vermeiden und eine Konsistenz zu erzielen, die ausstehenden Umsetzungen auf die gleiche Art lösen.

Solange sich das Team für eines dieser Vorhaben entscheidet und als einen neuen *Team Best Practice* aufnimmt und keine Anpassung plant, nimmt das Team keine bewussten technischen Schulden auf. Erst wenn durch weiteres Wissen, äußere Umstände oder ein neuer Entwickler die Lösung und damit *Team Best Practices* in Frage gestellt wird, kann es sein, dass diese als unbewusste Schulden auftreten.

Ein Beispiel ist im *Team S* aufgetreten, hier beschäftigte sich das Team bereits mit einer übernommenen Komponente, welche aus ihrer Sicht einige technische Schulden mit sich brachte. Das Team arbeitete an der Bereinigung, jedoch waren ihnen einige *globale Best Practices* der verwendeten Programmiersprache nicht geläufig. Ein neuer Entwickler stieß zum Team dazu und brachte einiges an Erfahrung im Bereich der Programmiersprache und der idiomatischen Schreibweisen mit. Diese würden helfen, den Code später leichter erweiterbar zu machen. Code, der vor wenigen Tagen zur Bereinigung von technischen Schulden geschrieben wurde, erschien nun erneut als unbewusste technische Schulden. Das Team beschloss, diese direkt zu tilgen und nicht weitere technische Schulden bewusst aufzunehmen.

Gerade neue Technologien bergen das Risiko, dass Teams *globale Best Practices* erst spät lernen und damit unbewusst viele *technische Schulden* aufgenommen haben.

Dies kann an unterschiedlichen Gründen liegen.

In der Praxis konnte beobachtet werden, dass eine neue Technologie zwar eingesetzt wird, jedoch nicht die Zeit investiert wurde, die Besonderheiten im Umgang mit der neuen Technologie zu lernen.

Im Folgenden beschreibt ein Mitglied aus *Team N* die Resultate eines Workshops, der dem Team *globale Best Practices* einer Technologie vermittelte. In diesem Fall konnte sogar einem produktiven Problem vorgebeugt werden (falls das Team die technischen Schulden früh genug behebt).

> „Kritisches Problem, von dem wir vorher nichts wussten, hat es [die Durchführung des Workshop] auch aufgedeckt."
>
> (Chat 2023-06-26 - NT)

Es kommt vor, dass sich Teams übernehmen und mehrere neue Technologien gleichzeitig einsetzen. In diesen Fällen unterschätzen Teams den zeitlichen Aufwand, den sie während der Entwicklung benötigen, um den Umgang mit allen zu erlernen. In dem gewählten Beispiel kam auf das Team sowohl eine neue Cloud Plattform als auch eine neu gewählte Datenbank Technologie zu. Um einer solchen Situation vorzubeugen, gibt es Ansätze in dem betrachteten Unternehmen in Form von Workshops. Ein passendes Beispiel ist ein Workshop für den Projektstart, er soll dazu dienen, Teams zu unterstützen, Fähigkeiten und Anforderungen abzugleichen und sich auf Technologien festzulegen, die ein bedachtes Maß an neuem Wissen benötigen. Dies soll vor allem der Aufnahme von unbewussten technischen Schulden entgegenwirken.

Die Beschreibung des Workshops lautet:

> „We start together with your team, to get an overview about the requirements of your product, skills within your team and recommend technical options. Furthermore, we inform your team about available support and guidance which are available for the chosen technologies."
>
> (Beschreibung interner Workshop: „Product Onboarding")

Andere Workshops konzentrieren sich darauf *globale Best Practices* von neueren Technologien in einer leicht konsumierbaren Form anzubieten. Im Folgenden wird das beworbene Ergebnis eines Technologie spezifischen Workshops angeführt:

> „Your database schema was validated against good practices from the book and the trainer shares additional experiences."
>
> (Beschreibung interner Workshop)

Hier wird explizit darauf eingegangen, dass spezifische Praktiken und Erfahrung mit dem Team geteilt werden sollen, dies dient unter anderem zur Vorbeugung und Identifikation von technischen Schulden.

Diese Art von Workshops werden jedoch teils so spät im Verlauf des Projektes gebucht, dass die technischen Schulden einer falschen Nutzung kaum noch zu bewältigen sind. So beschreibt *Entwickler NS*

> „Jetzt nach fast zwei Jahren und einer bereits gut gefüllten produktiv genutzten Datenbank lassen sich manche Themen (aus wirtschaftlichen Gründen) leider nicht mehr ändern."
>
> (Workshop-Feedback 2023-06-30 - NS)

Schlussfolgerung

Best Practices haben einen markanten Einfluss auf den Umgang mit technischen Schulden, da Teams zumeist ihre positive Wirkung auf die Erweiterbarkeit und Wartbarkeit nicht in Frage stellen. *Globale, firmen und individuelle Best Practices* werden im Team als *Team Best Practices* übernommen.

Anschließend identifizieren Teams unbewusst aufgenommene *technische Schulden* in Codebasen, welche den neu gelernten *Best Practices* entgegenstehen. Es konnte beobachtet werden, dass Teams die Tilgung dieser unbewussten technischen Schulden durchaus unter wirtschaftlichen Aspekten bewerten und diese als bewusste *technische Schulden* aufnehmen.

7.4 Spikes: Vor- und Nachteile

Teams haben teilweise Schwierigkeiten, Kosten und Nutzen eines Umbaus oder einer spezifischen Implementierung abzuschätzen. Um dies am Beispiel zu testen, werden teils sogenannte Spikes oder auch Proof-of-Concepts (PoC) gestartet.

Spikes oder PoCs werden von Entwicklern teils synonym verwendet, die klare Unterscheidung dieser beiden Begriffe ist nicht entscheidend für diese Arbeit, im Weiteren wird der Begriff Spike verwendet. Entscheidend ist, dass unter beiden Begriffen verstanden wird, grundlegende Coding Standards zu ignorieren. Hierzu zählt das Schreiben von Tests oder eine klare Strukturierung des Codes.

Das Ziel eines Spikes ist es, eine Hypothese so schnell wie möglich zu überprüfen, um daraus Erkenntnisse für das weitere Vorgehen abzuleiten. Der geschriebene Code kann als Referenz genutzt werden, jedoch ist die grundlegende Idee, den Code zu löschen und die Lösung von Grund auf neu zu implementieren.

Wird der Code jedoch übernommen, so nimmt das Team erhebliche *technische Schulden* auf, da der Code grundlegende Standards nicht erfüllt.

In den beobachteten Teams werden Spikes sowohl für die Abschätzung von Aufwand und Nutzen, als auch zur Machbarkeitsevaluation verwendet.

Spike als Aufwandsabschätzung

Eines der Teams hatte sich zu Beginn des Projektes für eine Datenbank-Technologie entschieden, jedoch unter Annahmen, die sich im Verlauf des Projektes als falsch erwiesen hatten.

> „Hat sich gezeigt, dass die Entitäten mehr verknüpft sind als gedacht."
>
> (Interview 2023-05-15 - NT)

Aus der Erfahrung der Teammitglieder aus vorherigen Projekten entstand die Vermutung, dass eine andere Art Datenbank Vorteile in der Entwicklung bieten könnte, jedoch war der Aufwand nur schwer abschätzbar. Aus diesem Grund strebte das Team einen Spike an, um den Aufwand besser abschätzen zu können.

> „Gegebenenfalls ist es in einer relationalen Datenbank einfacher. Tatsächlich planen wir einen PoC zu machen für eine Aufwandsabschätzung. Und ob es einen positiven Effekt auf die Weiterentwicklung haben könnte."
>
> (Interview 2023-05-15 - NT)

Diese Art des Vorgehens konnte auch im *Team S* beobachtet werden. Hier wurde eine Codebasis übernommen und vom Team überarbeitet. Jedoch ist auch nach mehreren Wochen der Überarbeitung kein zufriedenstellendes Ergebnis entstanden. Das Team hatte weiterhin Schwierigkeiten neue Funktionen in die bestehende Codebasis einzubauen. Um eine bessere Abschätzung machen zu können, wie schnell eine komplette Neuentwicklung voranschreiten würde und vertretbar wäre, starteten sie einen Spike.

Eine neue Funktion wurde als neue separate Codebasis neben die bestehende gestellt, nach einem initialen Aufwand für die Infrastruktur merkte das Team, dass sie deutliche Vereinfachungen vornehmen konnten, die in der alten Codebasis nur mit viel Aufwand einzubauen gewesen wären. Aufgrund mehrerer geplanter Funktionen und steigender Wichtigkeit der Komponente entschied das Team, die Investition zu machen und die alte Codebasis mit all den gefundenen technischen Schulden abzulösen.

Schlussfolgerung

Spikes oder PoCs können Teams helfen, ihre Aufwands- und Nutzenabschätzungen zu verbessern. Dies kann ein geeignetes Vorgehen sein, um sich positiven Effekten zu versichern oder die Tilgung von technischen Schulden gegen die Zinsen abzuwägen.

Teams sollten jedoch darauf achten, dass der investierte Aufwand dem Nutzen der Erkenntnisse angemessen ist. Deswegen wird empfohlen, im Vorfeld einen zeitlichen Rahmen festzulegen. Auch das Ergebnis, dass die gewünschten Erkenntnisse nicht innerhalb der verfügbaren Zeit erreicht werden können, mag bereits eine Hilfe bei der Entscheidung sein.

Gefahren eines Spike

Ein ähnlicher Sachverhalt zeigte sich im Team S. Das Team hatte eine fremde Codebasis übernommen und stieß immer wieder auf schwierig zu handhabende Probleme im bestehenden Code. Darunter waren irreführende Namen und ein inkonsistentes Verhalten der Speicher-Schicht, darüber hinaus waren einige Abstraktionen verwendet worden, die zwar eine Modularisierung des Codes erzielten, jedoch den Überblick für die neuen Entwicklern erschwerten.

Der *Entwickler SM* führte folgende Aspekte auf, die ihm in der Codebasis aufgefallen seien (Interview 2023-03-28 - SM):

- Sehr viele unterschiedliche Herangehensweisen
- Jede Funktionalität in einer eigenen Datei, dies erschwere den Überblick

- Sprechende Namen fehlen
- Datenobjekte, sind inkonsistent gefüllt
- Komponenten der Service-Schicht rufen sich gegenseitig auf
- Audit-Logs seien nicht korrekt, durch die Problematik in der Service-Schicht

Das Team entschied sich für einen Spike, der helfen sollte abzuschätzen, wie viel Aufwand es sein würde, das gesamte System neu zu schreiben. Alternativ müsse ein ausgedehntes Refactoring in der alten Codebasis durchgeführt werden. Die Grundannahme war, dass es einen erheblichen Gewinn an Geschwindigkeit gibt, sobald die neue Codebasis die Funktionen der alten Codebasis übernommen hat. Der Spike wurde inklusive Tests implementiert, jedoch ohne Dokumentation.

Dabei wurde schon im Vorfeld festgestellt, dass die Erkenntnisse anhand von Gefühlen der Entwickler abgeleitet werden würden, da es dem Team schwer fiel, Metriken im Vorfeld festzulegen.

> „Der Rewrite ist im Vergleich immer einfacher als das bisherige, allein weil es nicht so viel Code ist, am Anfang. Damit sind Kennzahlen schwierig zu finden. [...] [Ist] eher eine Gefühlssache."
>
> (Interview 2023-03-28 - SM)

Einen richtigen Vergleich könne man nur dann erzielen, wenn ein Feature in beiden Codebasen implementiert werden würde, diesen Mehraufwand wollte das Team jedoch nicht eingehen.

> „Man könnte noch ein kompliziertes Feature implementieren, um dann einen Vergleich zu haben. Neue Funktionen sind halt kaum zu vergleichen."
>
> (Interview 2023-03-28 - SM)

Im weiteren Verlauf des Interviews zeigt sich, dass die Bereitschaft den Spike anschließend wegzuwerfen eher gering ist. Dies scheint besonders an dem investierten Aufwand zu liegen und an den technischen Schulden, die in der alten Codebasis gänzlich entfallen könnten.

Auf die konkrete Frage, ob es noch eine mögliche Option sei, den Spike zu löschen, wies der *Entwickler SM* darauf hin, dass dies sicher möglich sei, die umgesetzten Funktionen dann jedoch erneut in der alten Codebasis umgesetzt werden müssten. Das würde jedoch größeres Refactoring voraussetzen. Über eine längere Zeit könnten

dann keine neuen Funktionen umgesetzt werden. Dies wäre eine Option, falls das Team sage: „Das [der Spike] wird uns nicht helfen". (Interview 2023-03-28 - SM)

Zu erkennen ist, dass die Bereitschaft einen Spike abzubrechen oder die erzeugte Codebasis zu verwerfen, insbesondere durch die Höhe des investierten Aufwandes/Zeit beeinflusst wird. Dies scheint besonders stark in Fällen zu sein, wenn die Umsetzung des Spike bereits die gewünschte Funktionalität bereitstellt. Wie anfangs beschrieben birgt das Übernehmen von Spikes jedoch besondere Risiken durch die aufgenommenen technischen Schulden, die bei der Übernahme des produzierten Codes entstehen.

Entwickler EN berichtet, dass sie manchmal Code aus Spikes übernehmen, diesem jedoch nachträglich Tests hinzufügen. Dies entspräche dann jedoch nicht dem Vorgehen nach TDD[22], welches sie sonst anwenden. Dies hätte jedoch bisher noch nicht zu Problemen geführt.

> „Ab und zu haben wir schon Code übernommen, aber auch dann getestet, wenn auch nicht nach TDD."
>
> (Chat 2023-07-12 - EN)

—

Ein weiteres Beispiel für die Aufnahme technischer Schulden durch die Verwendung eines Spikes findet sich in Team S. Das Team stellt eine interne Bibliothek bereit, welche über eine spezifische Authentifizierungsmethode bereitgestellt wird. Um diese Bibliothek automatisch aktualisieren zu können, wurde ein Spike gemacht, der die Verteilung von Zugriffs-Token ermöglichte. Entgegen teaminterner Standards wurde der Spike ohne automatisiertes Deployment oder Tests programmiert. Nach der erfolgreichen Erprobung wurde jedoch der Aufwand einer Neuimplementierung gespart, auch die automatisierte Pipeline oder Tests wurden nicht hinzugefügt.

Das Team hat hier *technische Schulden* aufgenommen, da die Implementierung des Spikes „gut genug" war. Änderungen wurden von den lokalen Rechnern der Entwickler deployed, jedoch waren dies seltene Einzelfälle im Abstand von Monaten.

Das manuelle Vorgehen führte schließlich dazu, dass ein Kunde keine Aktualisierungen mehr erhalten hat. Die Ursache war ein nicht veröffentlichter Eintrag in der Versionsverwaltung, der während einer Anpassung zwar deployed, jedoch nicht persistiert wurde. Dieser Fehler hat die Aufmerksamkeit des Teams wieder auf die aufgenommene *technische Schuld* gelenkt. Das Team entschied sich aufgrund des aufgetretenen Fehlers nun für eine Tilgung der Schuld.

[22] Testgetriebene Entwicklung, bei der Tests geschrieben werden, bevor produktiver Code geschrieben wird. https://www.it-agile.de/agiles-wissen/agile-entwicklung/was-ist-testgetriebene-entwicklung/

Einer der Hauptgründe des *Team S*, die Umsetzung des Spike nicht direkt wieder zu löschen, war, dass die Funktionalität benötigt wurde. Das Team war sich jedoch nicht sicher, ob die Nachteile der gefundenen technischen Lösung einer Verwendung nicht grundsätzlich im Wege stehen. Aus diesem Grund wurde die Umsetzung zunächst verwendet, um das akute Problem zu lösen. Die endgültige Entscheidung wurde vom Team, zu Gunsten anderer Funktionen, jedoch hinausgezögert. Anfragen anderer Teams führten dazu, dass die Umsetzung des Spike als einzige bekannte Lösung ausgeweitet wurde. Dies führte schließlich zu der beschriebenen Situation.

Der *Entwickler SH* sagte im Rückblick auf die Ereignisse „Wir waren uns nicht sicher, ob wir das so lassen wollten."

—

Auch der *Entwickler FF* wies in einem Interview darauf hin, dass er bereits einige schlechte Erfahrungen mit der Übernahme von Spikes gemacht habe. Ein häufiges Problem sei die Codequalität gewesen, in der Beschreibung eines konkreten Beispiels sagte er: „Es war alles in einer Datei, kaum erweiterbar. Es war viel Arbeit, es allen verfügbar zu machen. Es wurden alle Regeln gebrochen, die man sonst so anwenden würde." (Interview 2023-07-14 - FF)

Schlussfolgerung

Spikes können Teams helfen, Kosten und Nutzen von technischen Änderungen zu validieren, hierzu zählt auch die Tilgung technischer Schulden.

Code, welcher während Spikes entstanden ist, birgt häufig ein erhöhtes Maß an technischen Schulden, sollte er übernommen werden. Es sollte Zeit eingeplant werden, um die *technische Schulden*, die durch ein Übernahme der Codebasis entstehen, direkt zu tilgen.

Zudem ist es sinnvoll, Spikes klar zu definieren, bevor sie gestartet werden. Wichtig ist die zu überprüfende Hypothese und der geplante Aufwand, zudem sollte sich das Team vorher einigen, welches Ergebnis welche Konsequenzen hat. Anhand dieser Rahmenbedingungen sollte abgeleitet werden, welche Codestandards im Austausch gegen Schnelligkeit geopfert werden können.

7.5 Unzufriedenheit in Folge *technischer Schulden*

Einige beobachtete Situationen weisen darauf hin, dass *technische Schulden* in vielen Zusammenhängen mit Unzufriedenheit stehen. Dabei handelt es sich nicht allein um den Einfluss von existierenden *technischen Schulden*, sondern auch um deren Aufnahme bzw. Tilgung.

Unzufriedenheit in Teams ist eine eher vage Angelegenheit, die sich ausdrückt in Kommentaren und Gesprächen. Sie hat häufig zur Folge, dass die Motivation im Team sinkt.

Ein markantes Indiz für Unmut im Team ist in der Retrospektive zu finden. Aufgrund der unter Kapitel 6.1.4 genannten Gründe ist es jedoch nicht möglich, auf Daten aus der Retrospektive direkt zuzugreifen. Jedoch wird in Interviews immer wieder erwähnt, dass eine Thematik als negativer Punkt in einer Retrospektive vorkam.

Trotz des besprochenen Vorgehens kommen diese Punkte immer wieder in der Retrospektive vor. Es ist auch im begleiteten *Team S* zu erkennen, dass gerade die Wiederholung eines negativen Punktes, über mehrere Retrospektiven, einen markanten Einfluss auf die Stimmung im Team hat.

Im Folgenden werden Situationen angeführt, welche Hinweise über den Einfluss von technischen Schulden auf die Unzufriedenheit in Teams geben.

Aufnahme

Die Aufnahme von technischen Schulden kann dazu führen, dass die Qualitätsvorstellungen der Entwickler nicht mehr erfüllt werden. Zudem kommt es vor, dass die Erwähnungen anderer Entwickler nervt. So berichtet der *Entwickler DJ*: „Leute sind nicht glücklich, wenn ich [über die Aufnahme von technischen Schulden] meckere" (Interview 2023-06-15 - DJ)

Andersherum kann die bewusste Vermeidung von technischen Schulden Entwickler auch positiv beeinflussen. Nach einer Diskussion im *Team S*, in der sich gegen die Aufnahme von technischen Schulden entschieden wurde, sagte der *Entwickler SM* „Das macht mich sehr glücklich." (IPM 2022-05-11 - Team S)

Arbeit in Bereichen mit hohen technischen Schulden

Der *Entwickler EN und ES* berichten, dass in Retrospektiven es immer wieder dazu kam, dass Unmut über spezifische Bereiche im Code angeführt wurde. Dies hätte auch dazu geführt, dass man als Maßnahme die ein oder andere *technische Schuld* getilgt hätte. Sie berichten auch von der Arbeit in einem übernommenen Codebereich. Es sei schon entschieden, dass dieser später im Jahr neu geschrieben werden soll,

jedoch sei jetzt gerade keine Zeit. „Ist aber schon häufig in Retros aufgekommen. Dass es keinen Spaß macht daran zu arbeiten." (Interview 2022-07-23)

Ein ähnliches Beispiel gibt es aus dem *Team S*. Hier wurde besonders die Arbeit in einem übernommenen Codebereich als unangenehm empfunden. Nach dem *Entwickler SK* realisierten sie, wie lange die Umsetzung einer Funktion gebraucht habe, dies hätte dann sogar dazu geführt, dass sie überlegten, den gesamten Service neu zu schreiben. (Gespräch 2023-03-28 - SK)

In einem anderen Gespräch diskutiert das *Team S* über den Zustand, dass alle lieber in einen neueren Codebasis arbeiten, besonders weil dort die Umsetzung neuer Funktionen sehr schnell ginge.

Tilgung

Die Tilgung von *technischen Schulden* kann nach den Beobachtungen auch zu Unzufriedenheit führen. So berichtet *Entwickler FF* über die Entscheidung, ob eine Neuentwicklung stattfinden soll, oder die *technischen Schulden* in der bestehenden Codebasis zurückgezahlt werden sollten.

> „Als jemand, der drei Tage rein investiert hat. [...] Das kann ja nicht so schwer sein ... nach drei Tagen voller Komplexität und furchtbarer Entscheidungen, wir komplett frustriert waren, muss ich ehrlich sagen, war es für mich auch ein Faktor. Definitiv! Zwei andere Teammitglieder sahen das relativ nüchtern, [...] müsste man das halt machen. "
>
> (Gespräch 2023-07-14 - FF)

Deutlich wird, dass besonders das Refactoring im bestehenden Code, über drei Tage, bereits so frustrierend war, dass die Aussicht, dies längere Zeit machen zu müssen, dem Entwickler sehr widerstrebt. Interessant ist zudem, dass es deutliche Unterschiede zwischen den Entwicklern gab. Seine Arbeitskollegen schienen dies als zu erledigende Aufgabe anzunehmen.

Ähnliches ließ sich in *Team S* beobachten, hier rang das Team mehrere Monate abwechselnd mit der Tilgung *technischer Schulden* und der Entwicklung weiterer Funktionalität. Bis die Unzufriedenheit dazu führte, dass das Team einen Spike (siehe Kapitel 7.4) startete, um doch eine Neuentwicklung zu erwägen. Der Umstand, dass das Team sich nicht in der Lage sah, das Ausmaß der aufkommenden benötigten Tilgungen zu überschauen, bestärkte das Team, Alternativen zu prüfen.

Schlussfolgerung

Die bewusste Aufnahme von technischen Schulden kann zu Unzufriedenheit bei Entwicklern führen, jedoch wurden Beispiele gefunden, in denen Entwickler die Tilgung von technischen Schulden als ähnlich frustrierend empfanden. Generell wurde beobachtet, dass die Arbeit in Codebereichen mit hohen technischen Schulden zu Unzufriedenheit führen kann, unabhängig von den durchzuführenden Änderungen.

Dem entgegen stehen jedoch Beobachtungen, dass ein Entwickler genau dies als spannende Herausforderung betrachtete, es kann also empfohlen werden, diesbezügliche Veranlagungen bei Entwicklern in Erfahrung zu bringen und dieses Wissen gezielt einzusetzen.

7.6 Dokumentation

Sowohl das begleitete *Team S* als auch die weiteren interviewten Teams gaben Einsichten in die Dokumentation ihres Umgangs mit *technischen Schulden*. Im Folgenden sollen die gefundenen Dokumentationsstile und deren Einfluss auf den Umgang mit *technischen Schulden* beleuchtet werden.

7.6.1 Dokumentationsstile

Hierbei konnten vier verschiedene Dokumentationsstile beobachtet werden:

Task

Team S und W besitzen einen Backlog, der aus „Tasks" besteht, dies sind Aufgabenpakete, welche weder eine Aufteilung nach Art noch Aufwandsabschätzung haben. Beide Teams behandeln Tasks im Backlog als gleichberechtigte Aufgaben. Die Priorisierung der Tilgung von technischen Schulden leitet sich zumeist aus den nächsten umzusetzenden Anforderungen ab. Wenn das Team den Eindruck hat, dass es sinnvoll ist zunächst technische Schulden zu tilgen, bevor im gleichen Codebereich neue Anforderungen umgesetzt werden, dann wird dies zunächst gemacht. Beide Teams tendieren dazu, die Tilgung technischer Schulden in Tasks aufzunehmen, wenn diese eine Aktualität besitzen.

Chore

Die *Teams E, F, M* verwenden die gleiche Software zur Verwaltung ihrer Backlogs. Aufgaben werden in „Stories" festgehalten, diese haben einen Typen. Die Software bietet folgende Typen [102]:

„Features" sind der Standard Story Typ und sind gedacht für Aufgaben, welche einen Business-Wert für den Kunden haben. Features werden mit Punkten versehen, die Aufwand oder Komplexität anzeigen (teamabhängig, Stichwort agile Schätzung).

„Chores" sind Aufgaben, die zwar notwendig sind, jedoch keinen direkten Nutzerwert erzeugen. Als Beispiel wird der Austausch eines SSL-Zertifikats angeführt. Chores haben in der Standardeinstellung keine Punkte, sind daher nicht geschätzt, was ihren Aufwand oder Komplexität betrifft.

„Bugs" bezeichen ein unbeabsichtigtes Verhalten der Software und haben die gleichen Eigenschaften wie Features.

„Releases” sind Markierungen für Meilensteine, die in den Backlog eingereiht werden können.

Teams, welche diese Art der Backlogverwaltung nutzen, dokumentieren die Tilgung von technischen Schulden als Stories mit dem Typ Chore.

In den genannten Teams wurde die Standardeinstellung bezüglich der Schätzung von Chores nicht verändert. Dies hat zwei markante Effekte:

- Die Software berechnet Statistiken über die Entwicklungsgeschwindigkeit („Velocity”) des Teams, diese wird anhand der geschätzten Punkte einer Story berechnet. Da Chores jedoch keine Punkte haben, werden diese mit 0 berechnet. Somit werden in den Statistiken der Teams Aufgaben, die zwar notwendig sind, jedoch keinen direkten Nutzerwert erzeugen, übergangen. Die berechnete Geschwindigkeit einer Iteration sinkt, um so mehr Chores bearbeitet werden.

  Dies fällt besonders in Iterationen konzentrierter Schuldentilgung auf.

- Teams dokumentieren und bearbeiten Aufgaben wie *technische Schulden*, ohne dass es zuvor eine Abschätzung gibt, die dokumentiert wurde. In den Planungsbesprechungen mag es sein, dass Aussagen über den Umfang von Chores getroffen werden, jedoch fehlt das methodische Vorgehen, wie es bei der Schätzung von Features der Fall ist. [60]

Aus den Berichten des *PM MM* geht hervor, dass ihm dieser Umstand bewusst ist, er nutzt in Zeiten vieler Chores andere Metriken, welche sich nur auf die Anzahl der erledigten Stories beziehen.

> „Wenn die Arbeit von 3 Monaten genauso viel zählt wie die Arbeit von 2h für einen Bug oder einen Analyse-Chore, dann macht das jede Auswertung kaputt.Das Problem von Chores ist eher, dass man nie weiß, wieviel Arbeit das wirklich ist. Die Plattform Migration hatte bei uns 15 Monate gedauert. Wir hatten vorher 3 Monate angenommen. Die Gatewayumstellung sollte 1-2 Wochen dauern, hat aber wie gesagt 3 Monate gedauert. Echtes Refactoring (und da würde ich den Großteil der Arbeiten an technischen Schulden verorten) wäre da schon etwas einfacher zu schätzen.Dazu kommt, dass es mir als PM unmöglich erscheint den Value von Chores zu bewerten.”
>
> (Chat 2023-07-18 - MM)

Zudem geht er darauf ein, dass aus seiner Perspektive die Tilgung von technischen Schulden eher weniger ein Problem darstellt, andere technische Aufgaben und Herausforderungen brächten viel mehr Unruhe in die Planung. Diese Beobachtung ist zwar außerhalb des Rahmens dieser Arbeit, jedoch weist die Schilderung darauf hin, dass der Umgang mit technischen Schulden Teil eines größeren Problems mit dem Umgang mit technisch notwendigen Aufgaben ist.

Bezüglich der Priorisierung innerhalb des Backlogs werden Stories zumeist vom PM priorisiert mit dem Feedback der Entwickler und Designer, Chores werden jedoch von Entwicklern verschoben, wobei es durchaus zu Diskussionen zwischen Entwicklern und dem PM/PO kommt, ob dies notwendig ist. Der *Entwickler EF* beschreibt dies aus einem vorherigen Team, in dem er Software entwickelt hat.

> „Im vorherigen Team ist mir das so ein wenig aufgekommen, meistens nicht als generelle Ansage das Chores nicht gemacht werden sollen, sondern es wurde versucht aus der PO Sicht zu hinterfragen, ob das wirklich notwendig ist. [...] Die wurden dann auch mal geschoben."
>
> (Interview 2023-02-06 - EN)

Todo im Code

Manche der Teams, unter anderem das begleitete *Team S*, markieren Stellen im Code mit Kommentaren, welche auf eine noch ausstehende Änderung des Codes hindeuten (Todos).

Diese Art von Markierungen wird auch von Entwicklungsumgebungen unterstützt und explizit hervorgehoben. Zudem werden Entwickler beim Ablegen von Codeänderungen darauf hingewiesen, falls ihre Änderungen neue „Todos" beinhalten. [57]

Teams setzen diese ein, um unter anderem kleinere *technische Schulden* zu markieren. Ein Beispiel hierfür sind Markierungen im Code von *Team S*, welche Stellen markieren, die nur für einen Migrationszeitraum benötigt werden.

Entwickler SM beschreibt, dass diese Todos einerseits genutzt werden, um Stellen im Code zu markieren als Erinnerung, jedoch auch um weitere Hinweise zu geben, wann dieser Code wieder entfernt werden kann.

> „Es gibt noch Todos im Code, aber alle basieren darauf, dass wir noch Migrationscode haben, der nach der Migration entfernt werden kann. Da steht dann auch, welche Testdaten entfernt werden können."
>
> (Gespräch 2023-06-12 - SM)

Es wurden mehrere Beispiele hierfür gefunden. Der *Entwickler SM* weist darauf hin, dass in einigen Reviews aufgefallen ist, dass Teammitglieder sich teils an Codestellen orientiert haben, welche noch ausgebessert werden sollten, jedoch keine Markierung aufwiesen. Dadurch wurden unbewusst *technische Schulden* aufgenommen.

Neben dieser Beobachtung ist zudem aufgefallen, dass unter anderem im *Team D* zumeist diese Todos ohne Prefix (wie „TODO" oder „FIXME") erstellt wurden, dies erschwert deutlich die Erkennung, ob ein Kommentar nur eine Besonderheit der Umsetzung oder eine zu schließende Lücke beschreibt.

Entscheidungsprotokolle

Einige der beobachteten Teams führten Entscheidungsprotokolle. Diese wurden zumeist dazu genutzt, größere Entscheidungen festzuhalten. Das *Team E* verwendete Architecture Decision Records (ADR), um technische Diskussionen mit dem gesamten Team festzuhalten.

Entwickler EN berichtete, dass diese auch als Leitfaden dienen. So unterbrach das Team eine Diskussion, um zunächst ein neues ADR anzulegen und anschließend der Struktur des Dokuments zu folgen. Besonders häufig entstehen ADRs bei Diskussionen, die Teile der System- oder Softwarearchitektur betreffen. (Interview 2023-07-23 - EN)

Im *Team S* werden teilweise wichtige Entscheidungen oder Regeln, auf die sich das Team geeinigt hat, in Entscheidungsprotokollen festgehalten. Diese dienen zumeist der Dokumentation von Codingstandards. Die untersuchten Protokolle beinhalten keine bewusste Aufnahme von technischen Schulden. Jedoch befasst sich eines der Protokolle mit der Entscheidung zusätzlichen Aufwand zu betreiben, um die Entwicklung einer in der Zukunft liegenden Funktionalität zu erleichtern. Dies ist ein Hinweis, dass in diesem Fall die Vermeidung einer technischen Schuld dokumentiert wurde.

In allen Fällen wurden die Entscheidungsprotokolle nicht explizit für *technische Schulden* verwendet, jedoch sind unter den untersuchten Dokumenten Beschreibungen zu finden, die auf die Definition der technischen Schulden zutreffen.

7.6.2 Anhäufung von technischen Schulden

In den Teams kommt es unabhängig vom Dokumentationsstil immer wieder dazu, dass sich am Ende des Backlogs technische Aufgaben, die einen Mehrwert versprechen, ansammeln. Besonders in dem *Team E* wurde dieses Problem im Zeitraum der Untersuchung beschrieben.

> „[Es haben sich] viele Chores angesammelt, technische Schulden dementsprechend. Wir setzen uns jetzt Donnerstags zusammen und sprechen die Chores durch und priorisieren sie, packen sie in ein separates Backlog. Dass man sich ein paar Sachen heraus greifen kann, wenn man Leerlauf hat."
>
> (Interview 2023-02-06 - EF)

Der *Entwickler ES* beschreibt, dass sie viele Chores angesammelt hatten und diese nun nach und nach durchgehen, priorisieren und in einem separaten Backlog führen. In einem späteren Interview wurde berichtet, dass sich gerade bei der Durchsicht dieser Ansammlungen herausstellte, dass mehrere der Chores gelöscht wurden.

Es wurden verschiedene Gründe angegeben:

- Team konnte sich an den Kontext nicht mehr erinnern. Dies war häufig der Fall, wenn der Autor bereits das Team verlassen hatte, jedoch kam es auch vor, dass anwesende Autoren sich nicht mehr sicher waren, was sie vor einem halben Jahr damit gemeint hatten. Dies schien häufiger aufzutreten, wenn die Chores nur kurze Beschreibungen ohne weiteren Kontext hatten.
- Die vorgeschlagene Änderung hat sich im Verlauf der Zeit als gänzlich unwichtig herausgestellt. Dies konnte daran liegen, dass besonders die Zinsen der dokumentierten Schuld nicht anfielen.
- Änderungen waren nicht mehr notwendig. Über den Zeitraum der Beobachtung wurden viele der Services auf eine neue Cloud Plattform umgezogen. Hierbei sind besonders *technische Schulden* in der Infrastruktur gänzlich entfallen, da dieser Teil komplett ersetzt wurde.

Ähnliche Phänomene sind im *Team N* aufgefallen, als Reaktion darauf wurde die Tilgung von technischen Schulden nur noch dann dokumentiert, wenn dies im Verlauf der nächsten paar Wochen erfolgen sollte.

Andere technische Aufgaben, die bis zu einem gewissen Zeitpunkt umgesetzt werden mussten, jedoch nicht der Definition der *technische Schulden* entsprechen, wurden im Backlog entsprechend weit nach unten geschoben.

Die Erkenntnisse aus den Beobachtungen bezüglich der Dokumentation von *technischen Schulden* lässt sich wie folgt zusammenfassen:

Die Dokumentationsstile der Teams unterscheidet sich von Team zu Team leicht.

Aufgrund der Berichte von Entwicklern ist bekannt, dass große Tilgungen durchaus anhand ihres Aufwands/Komplexität und Nutzens diskutiert werden, jedoch ist

dies nicht für die Tilgung kleiner bis mittleren *technische Schulden* der Fall. Die Unterscheidung ergibt sich zumeist aus dem geschätzten zeitlichen Aufwand der Tilgung.

Die fehlende Schätzung und Dokumentation der Vorteile einer Tilgung führen immer wieder zu Diskussionen und Verschiebung der Umsetzung. Zudem füllen Teams ihre Backlogs mit Einträgen, welche dazu führen, dass Entwickler das Gefühl entwickeln, dass viele *technische Schulden* im System vorhanden sind, bei genauerer Betrachtung werden viele dieser Einträge jedoch gelöscht.

Diese Erfahrung führt in einigen Teams dazu, dass die Dokumentation von technischen Schulden weiter reduziert wird, so dass zumeist nur noch Tilgungen von technischen Schulden aufgenommen werden, die über den Verlauf weniger Wochen umgesetzt werden sollen.

Schlussfolgerung

Teams dokumentieren *technische Schulden* als technische Aufgaben, die die Tilgung betreffen. Fehlende Details, wie die Beschreibung der Auswirkungen, dem geschätzten Aufwand oder zu zahlenden Zinsen, führen dazu, dass sich *technische Schulden* ansammeln. Einträge im Backlog verlieren den Kontext und werden schließlich gelöscht.

Teams gehen dazu über nur noch die Tilgung von akut auftretenden *technischen Schulden* zu dokumentieren, wenn eine Umsetzung in wenigen Wochen möglich erscheint.

7.6.3 Markierung von Codestellen mit hoher Qualität

Neben den beschriebenen Todos wurden in den Codebasen mehrfach Beispiele gefunden, in denen vorbildliche Codestellen markiert wurden, um anzuzeigen, dass sie sich als passende Grundlage für ähnliche Umsetzungen eignen.

Der *Entwickler DJ* berichtet, dass sie Stellen im Code markieren, oder in einem separaten teaminternen Chat veröffentlichen, um Entwickler zu motivieren, sich bei der Entwicklung an diesen Codestellen zu orientieren.

Aus den Berichten des *Entwickler SM* geht hervor, dass eine Quelle von unbewussten technischen Schulden sein kann, dass Entwickler Codestellen kopieren, die zu einer technischen Schuld gehören. (Retrospektive Team S)

Schlussfolgerung

Sowohl die Markierung von noch ausstehenden Anpassungen (Todos) als auch die Markierung von vorbildlichem Code könnten helfen, *technische Schulden* zu vermeiden.

Ein Entwickler, welcher eine Codestelle mit einer ähnlichen Funktion sucht, kann hier sowohl positives als auch negatives Feedback erhalten, ob der Code sich als Vorlage eignet.

Ob sich dieses Vorgehen als sinnvoll herausstellt, benötigt jedoch weitere Forschung.

7.7 Priorisierung und Tilgung

In Teams konnte beobachtet werden, dass sich die Tilgung von technischen Schulden durchaus als eine motivierende Abwechslung im Alltag eignet. Während der Entwicklung von größeren Funktionsumfängen kann es dazu kommen, dass das Team über längere Zeit viele ähnliche Aufgaben im selben Codebereich bearbeiten muss.

Der Abbau von *technischen Schulden* kann dementsprechend genutzt werden, um einerseits Abwechslung in die Aufgaben einer Iteration zu bringen. So entschied sich das *Team S* dazu eine technische Schuld zu tilgen, da sie „eine spannendere Aufgabe für die, die nicht so gerne [...] machen", darstellt.

Darüber hinaus können Entwickler Zeiten nutzen, in denen sie normalerweise keine neuen Funktionen umsetzen würden. Beispiele hierfür sind Urlaubszeiten, wo große Teile des Teams fehlen, kurze Zeitabschnitte zwischen Besprechungen oder wenn durch eine ungerade Anzahl an Personen ein Entwickler alleine arbeitet.

Diese Möglichkeit wird unter anderem in *Team D* genutzt. So beschreibt *Entwickler DJ*:

> „Es ist auch [...] teilweise [...] [beim] Pair-Programming oder sowas, [...] da ist jetzt einer alleine, dann kannst du auch mal so ein Aufräum-Chore dazwischen schieben oder Urlaubszeit weniger los oder solche Sachen." (Interview 2023-06-15 1516 - DJ)

Ein ähnliches Vorkommen beschreibt *Entwickler EN*:

> „Wir hatten im Frühjahr, wo wir ungerade waren, da hat sich jemand alleine hingesetzt und hat die Low-Hanging Fruits raus gepickt. Und dann mal bisschen technische Schulden bearbeitet." (Interview 2022-07-23 EN)

„Low-Hangig Fruits" verweist hierbei auf leicht zu lösende Aufgaben, die ohne viel Aufwand erledigt werden können, so wie tief hängende Früchte leicht gepflückt werden können.

Schlussfolgerung

Während die Tilgung von technischen Schulden zumeist als hinderlicher Aufwand angesehen wird, können diese Aufgaben auch zum Vorteil eingesetzt werden. Abwechslung in den Aufgaben, Motivation der Entwickler, Schulungszwecke oder das Einbinden eines neuen Entwicklers sind durchaus Beispiele für eine gewinnbringende Verwendung *technischer Schulden*. Zudem wurde beobachtet, dass besonders in Teams, die Pair Programming praktizieren, technische Schulden häufig zurückgezahlt werden, wenn ein Entwickler alleine keine Story umsetzen möchte.

8 Diskussion

8.1 Abgleich mit Ergebnissen aus „An enterprise perspective on technical debt"

Wie in Kapitel 4 beschrieben, geht die Arbeit „An Enterprise Perspective on Technical Debt" von T. Klinger et al. [62] im Besonderen auf technische Schulden im Kontext eines Großunternehmens (IBM) ein. Ein markanter Unterschied zwischen den Unternehmen des Artikels und dieser Arbeit ist die Ausrichtung und Historie. Im Gegensatz zum untersuchten Unternehmen dieser Arbeit ist IBM ein zu großen Teilen softwarezentriertes Unternehmen. [54]

Der Artikel basiert auf vier Interviews mit Software Architekten, welche Erfahrungen aus jeweils verschiedenen Projekten beschreiben. Dies steht im Kontrast zu den Datenquellen dieser Arbeit. Die durchgeführten Interviews dieser Arbeit, welche sich besonders auf Entwickler in verschiedenen Projekten konzentriert, weisen eine andere Perspektive auf. Zudem wurden darüber hinaus weitere Datenquellen benutzt, wie der Backlog, ADRs, Pairings, Versionsverwaltung, Besprechungen sowie weitere interne Informationen und Leitfäden.

Im Folgenden werden die Kernaussagen des Artikels anhand der eigenen Daten und deren Interpretation überprüft. Dies gibt Aufschluss, inwieweit die eigenen Ergebnisse übertragbar sein könnten, im Besonderen auf softwarezentrierte Unternehmen.

Die Ergebnisse der Arbeit sind aufgeteilt in „Gemeinsame Funde bei allen Befragten" und „Schlussfolgerung".

8.1.1 Gemeinsame Funde bei allen Befragten

Der Artikel führt folgende Beobachtungen auf, die in allen vier Interviews aufgefallen waren:

„Induced and unintentional debt are challenges."
Die induzierte und unbewusste Verschuldung ist eine Herausforderung.

Nach den Aussagen der Befragten waren Schulden, die nicht bewusst eingegangen waren, also unbewusste Schulden, besonders problematisch. Diese wurden vor allem durch Stakeholder und externe Faktoren verursacht.

> „[...] involves unintentional debt that the architects and other stakeholders did not actively incur, but which was caused by situations such as acquisition, new alignment requirements, or changes in the market

> ecosystem. [...] In the view of the technical architects we interviewed, the nonintentional debt typically was much more problematic than the intentional debt." [62]

In den eigenen Daten finden sich zwei größere Beispiele von technischen Schulden, welche dazu geführt haben, dass eine Neuentwicklung notwendig war. Das erste Beispiel bezieht sich auf ein größeres Modul, welches einen Arbeitsablauf steuert. Dieses wurde vor über einem Jahr entwickelt und seitdem nicht mehr angepasst. Innerhalb dieser Zeit wurden jedoch durch äußere Umstände alle Entwickler des Teams rotiert.

Es folgt ein Auszug aus einem Entscheidungsprotokoll, welches den Kontext der Entscheidung beschreibt, welcher zu der Neu-Implementation geführt hat.

> [The] module is supposed to get many more functionalities in the future. Additionally, the main data source system [...] releases a new API with many changes in the workflow. Therefore, major changes in the existing [...] system are required to accomplish both [of] these goals.
>
> Unfortunately, the current team members joined the team only after [the module] was implemented so there is no inside knowledge of the existing [...] system.
>
> (ADR - Team F)

Dem derzeitigen Team war nicht bewusst, welche technischen Schulden sich in dem Modul angesammelt hatten. Durch die ausbleibenden Änderungen und die Rotation entwickelte sich eine problematische Situation, welche das Team mit einer großen Menge an technischen Schulden konfrontierte. Aus Sicht des Teams waren diese nur mit außerordentlichem Aufwand zu tilgen und nach Schätzungen des Teams überstieg der Aufwand dem einer Neuentwicklung. Dies kann als schlimmster Ausgang angesehen werden.

Ein weiteres Beispiel aus den eigenen Daten findet sich im *Team S*. Hier wurde ein Service übernommen und über einige Monate gepflegt. Nach der Übernahme hatte das Team bereits Teile der Codebasis angepasst, mit dem Ziel, die Erweiterbarkeit und innere Codequalität zu erhöhen. *Entwickler SK* beschreibt die Arbeit an dem Service wie folgt.

> „Es ist so ein bisschen die Büchse der Pandora: Man findet immer wieder etwas Neues, wo man sich denkt, „ähh" [Ausruf der Ratlosigkeit], das gibt einem immer wieder ein schlechtes Gefühl damit zu arbeiten."
>
> (Interview 2023-03-28 - SK)

Nach mehreren Wochen des Refactorings entschied sich das Team, einen Spike zu machen, um besser einschätzen zu können, wie viel Aufwand und welche Vorteile eine Neu-Implementierung haben würde. Dieses Vorgehen resultierte unter anderem aus den Gefühlen des Teams, dass immer weitere Probleme mit der bestehenden Codebasis auftauchen würden.

In diesem Beispiel ist besonders gut zu erkennen, welchen massiven Einfluss gerade die unbewussten Anteile der *technische Schulden* auf ein Team und die Bereitschaft, diese zu tilgen, haben können.

In beiden Fällen führten die hohe Menge bzw. ungewisse Menge an unbewussten technischen Schulden zu der extremsten Form der Tilgung. In den gesammelten Daten findet sich kein vergleichbares Extrem, welches durch bewusste *technische Schulden* ausgelöst wurde.

Schlussfolgerung

Wie im Artikel beschrieben, konnte auch in den eigenen Daten beobachtet werden, dass insbesondere injizierte oder unbewusste *technische Schulden* große Herausforderungen sind.

„Decisions are managed in an ad hoc manner"

Entscheidungen werden ad hoc getroffen

Die Autoren beschreiben, dass das Management und die Nachverfolgung von technischen Schulden zumeist informell und ad hoc war. Zudem wurden Entscheidungen oft ohne formelles Vorgehen getroffen, auch wurde nur selten beobachtet, dass die Auswirkungen quantitativ abgeschätzt wurden. Dokumentationen von Entscheidungen waren selten und die Gründe zumeist nur im „tribal memory" (Teamgedächtnis) vorhanden, welches sich als unzuverlässig erwies.

Diese Beobachtungen treffen auf die eigenen Daten zu, gerade kleinere-mittlere Entscheidungen werden gar nicht dokumentiert. Es gibt Teams, die auf ADRs setzen, wobei auch hier darauf hingewiesen wird, dass diese Art der Dokumentation zumeist erst erfolgt, wenn Diskussionen das gesamte Team in Anspruch nehmen, viele Entscheidungen werden zwischen zwei bis vier Entwicklern gefällt.

> „Sprechen erst im Pair, falls dann noch nötig sprechen wir mit anderen: „Könnt ihr uns noch einen Rat geben"; bei viel Impact [/Auswirkungen] [fragen wir] dann doch das ganze Team.""

(ADR 2022-07-23 - Team E)

Ähnliches Vorgehen wurde auch im *Team S* beobachtet.

Darüber hinaus ist aufgefallen, dass die Dokumentation immer wieder Lücken aufweist, wie ein Entwickler aus *Team E* berichtete:

> „Nein, [haben kein neues ADR], sollten wir mal machen. Haben neue Komponenten eingebaut, aber keine ADRs dafür. Sollten wir mal machen, das ist immer das Problem mit der Dokumentation.
>
> (Interview 2022-11-11 - ES)

Auch im späteren Verlauf wurde die hier erwähnte Dokumentation nicht nachgeholt.

Schlussfolgerung

Wie im Artikel beschrieben waren viele Entscheidungen ad-hoc und undokumentiert, jedoch wurde der Einsatz von Entscheidungsprotokollen beobachtet, welche jedoch nicht zuverlässig erstellt wurden.

„Stakeholders with different types of concerns lack effective ways to communicate and reason about technical debt"

Stakeholdern mit unterschiedlichen Anliegen fehlen effektive Wege, um über technische Schulden zu kommunizieren und zu argumentieren

Der Artikel beschreibt, dass nicht-technische Stakeholder oft nicht die Auswirkungen von technischen Schulden verstanden haben. Die befragten Architekten berichteten zudem, dass es keine etablierten Kommunikationskanäle oder gemeinsames Vokabular mit den nicht-technischen Stakeholdern gebe.

In den beobachteten Teams konnten Hinweise darauf gefunden werden, so beschrieb ein Entwickler in *Team W*, dass die Aufnahme von technischen Schulden transparent gemacht wurde, jedoch haben die Verantwortlichen deren Auswirkungen wahrscheinlich nicht verstanden.

> „Er [der Fachbereich] wusste es, dass wir technische Schulden aufgenommen hatten. Ob es ihm bewusst war, was das bedeutet, dass würde ich

erst einmal verneinen. Ich habe häufig gehört [vom Fachbereich]: Es wäre ihnen klar, dass technische Schulden aufgenommen werden, aber sie müssten liefern."

(Interview 2023-05-11 - WS)

Der PM des *Team N* beschreibt, dass sie Schwierigkeiten hatten, dem Fachbereich zu erklären, warum sie mehr Zeit bräuchten, um Code Smells zu bereinigen. Andere Aufgaben, wie die Behebung von Compliance-Verletzungen wären, durch die Sichtbarkeit in der Hierarchie, deutlich leichter zu verargumentieren.

Dies führte unter anderem dazu, dass das Team einen „U-Boot Spike" [23] plante, um dem Fachbereich nicht erklären zu müssen, warum sie etwas entwickeln wollen, wenn sie es danach vielleicht wegwerfen - ein klarer Hinweis darauf, dass durch eine schwierige Kommunikation Teams noch weniger motiviert sind, Transparenz über *technische Schulden* zu schaffen.

Ein weiterer Bericht des *Entwickler FF* zeigt die Wichtigkeit auf, die eine quantitative Beleuchtung von technischen Schulden hat. In einer Diskussion mit dem Product Owner (PO) über die Neu-Implementierung eines ganzen Moduls konnten gerade fundierte Aussagen über den Aufwand und das erwartete Resultat eine gemeinsame Basis für die Kommunikation schaffen.

Schlussfolgerung

In den Beobachtungen konnten sowohl Beispiele für die schwierige Kommunikation und das fehlende Verständnis von nicht-technischen Stakeholdern gefunden werden, als auch die Wirkung von quantifizierten Argumenten.

8.1.2 Schlussfolgerungen des Artikels

Über die diskutierten Funde konstatieren Klinger et al. folgende Schlussfolgerungen.

„Collectives matter, for both products and people"

Kollektive sind wichtig, sowohl für Produkte als auch für Menschen

Die Schlussfolgerung bezieht sich auf entgegengesetzte Effekte. Während eine globale Optimierung dazu führen kann, dass einzelne Projekte leiden, fehlt oft die Übersicht, um die Komplexität von technischen Schulden global zu handhaben.

[23] Das U-Boot ist eine Metapher, welche beschreiben soll, dass Aufgaben unsichtbar für eine gewisse Instanz ausgeführt werden. In diesem Fall unsichtbar für den Fachbereich, vorbei an den existierenden Prozessen.

Darüber hinaus können kompetitive Ziele von Stakeholdern dazu führen, dass ein Kompromiss aller Anforderungen umgesetzt wird, der rein technisch betrachtet suboptimal ist.

Für diese Effekte konnte ein prägnantes Beispiel gefunden werden. Die *Teams E, F* und zwei weiterer Teams wurden nacheinander in der gleichen Domäne aufgesetzt, jedoch mit disjunkten Aufgaben, d.h. dass die Teams unterschiedliche Aufgaben im gleichen Geschäftsbereich abbildeten und die gleichen Nutzer hatten, diese also keine Systemübergänge erkennen sollten. Die fehlende Kommunikation zwischen den isolierten Teams und unterschiedlichen Stakeholdern führte zum Aufbau technischer Schulden in allen Teams. Diese traten insbesondere durch die mehrfache Entwicklung gleicher Funktionalitäten in Erscheinung. Erst durch die Einführung eines neuen Kommunikationskanals, der alle Stakeholder und Teams zusammenführte, wurden diese koordiniert abgebaut.

Schlussfolgerung

Die beschriebene Thematik ließ sich in mindestens einem Fall beobachten, dies kann als Bestätigung der Schlussfolgerung gesehen werden.

„The ability to assess debt matters"

Die Fähigkeit, Schuldenfragen zu beurteilen

Klinger et al. beschreiben die Notwendigkeit quantitativer Werte für Rückschlüsse auf die Auswirkungen von technischen Schulden. Aufgrund fehlender Daten konnten keine weiteren Aussagen getroffen werden, jedoch wird vermerkt, dass sie die Schwierigkeit anerkennen, diese Daten in der Praxis zu sammeln.

In den eigenen Beobachtungen wurden ähnliche Phänomene gefunden, so schreibt der *PM MM*:

> „Die Gateway Umstellung sollte 1-2 Wochen dauern, hat aber wie gesagt 3 Monate gedauert. [...] Dazu kommt, dass es mir als PM unmöglich erscheint, den [Wert] von Chores zu bewerten. Die agilen Coaches scheinen mir dafür auch nicht die Patentlösung zu haben." (Chat 2023-07-18 - MM)

Er verweist darauf, dass es ihm schwer fällt, technische Aufgaben, wie auch technische Schulden, abzuschätzen. Auch interne beratende Rollen haben hierfür

derzeit keine geeignete Lösung anbieten können. Darüber hinaus beschreibt der *Entwickler EN* im Rückblick auf seine Entwicklungszeit im *Team E*:

> „[Wir haben die] Testabdeckung überprüft; mindestens 80%. [...] Wahrscheinlich stimmt die Qualität der Tests nicht. [...] Häufig haben wir, dass wir diskutieren und es ist alles sehr Bauchgefühl-mäßig. Das war beim Testen jetzt das Problem. Ok, Bauchgefühle, ... wir müssen jetzt mal messen. Hilft nichts, wenn wir in die falsche Richtung laufen. Das haben wir aber nicht permanent" (Interview 2023-05-10 - EN)

Er beschreibt, dass in wenigen Fällen, wie der Testabdeckung, leicht zu erhebende Metriken genutzt werden könnten, dies jedoch selten der Fall ist. Aus anderen Beschreibungen wird ersichtlich, dass Spikes genutzt werden können, um Aufwand und Nutzen abzuschätzen (siehe Kapitel 7.4.

Schlussfolgerung

Wie in Kapitel 7.6.2 berichtet, haben Teams in der Praxis erkennbare Schwierigkeiten passende Metriken zu finden, die eine Quantifizierung von technischen Schulden und deren Auswirkung möglich machen. Es gibt Situationen, in denen besonders Spikes eingesetzt werden, um eine Abschätzung beider Dimensionen zu machen, jedoch fehlen klare Metriken, um diese zu bewerten. Darüber hinaus fehlt die nachträgliche Betrachtung, um die Schätzungen langfristig zu verbessern.

„Bridges across enterprise gaps matter"

Brücken über Unternehmenslücken sind wichtig

Die Schlussfolgerung bezieht sich auf die „gaps" (Lücken) zwischen geschäftlichen, operativen und technischen Interessengruppen. Diese verursachen unsichtbare Probleme, die erst auffallen, sobald sich „verheerend auswirken" [24].

In einem der eigenen Interviews beschreibt der *Entwickler SD* die Lücken zwischen technischen Teams und Fachbereichen wie folgt:

> „Der damalige PO hat gesagt: Er hat kein Problem, zwei drei Sprints technische Schulden bewusst einzugehen, um ein Feature fertig zu bekommen. Er hat aber nicht gesagt, dass er bereit ist, das später zu bezahlen. Ihnen

[24] Die Autoren nutzen die Umschreibung „wreak havoc", welche eine drastische Wirkung impliziert

ist nicht klar, dass man sechs Monate später Schwierigkeiten haben kann, bis hin zu Produktionsausfällen. Ist manchmal sehr schwierig gewesen." (Interview 2022-11-11 - SD)

Hier ist ein deutliches „gap" zu erkennen. Bezüglich des „gap" zwischen Entwicklung und Betrieb konnte keine klaffende Lücke beobachtet werden. Die Teams verantworten sowohl die Entwicklung als auch den Betrieb der Software (DevOps), es gibt dementsprechend keine zwei Interessengruppen.

Es gibt Hinweise, dass unerfahrene Teams *technische Schulden* im Bereich der Operative aufnehmen und diese meist kurz vor der Live-Schaltung versuchen zu tilgen, jedoch ist dies unter einer anderen Perspektive zu betrachten.

Schlussfolgerung

Während die Beobachtung über die Defizite zwischen Entwicklung und Fachbereich die Folgerungen des Artikels bestätigen, führte die zusammengelegte Verantwortung von Entwicklung und Betrieb dazu, dass es keine zwei Parteien gab, zwischen denen sich eine Lücke zeigen konnte.

8.1.3 Zusammenfassung

Sowohl die Gemeinsamkeiten als auch Schlussfolgerungen von Klinger et al. wurden in den gesammelten Daten gefunden. Es ist davon auszugehen, dass die für diese Forschung gemachten Beobachtungen vergleichbar sind mit den zu Grunde liegenden Ergebnissen von „An Enterprise Perspective on Technical Debt". [62]

8.2 Abgleich mit Ergebnissen aus „Umgang mit technischen Schulden im Startup-Umfeld”

Die Masterarbeit „Umgang mit technischen Schulden im Startup-Umfeld” von Ververs [129] untersucht den Umgang mit technischen Schulden im Startup Umfeld, hierfür wird auch die GTM benutzt, somit stimmt sie mit der Methode dieser Arbeit überein, wobei unterschiedliche Datenquellen genutzt wurden.

Im Folgenden sollen die Aussagen der Arbeit mit den gesammelten Daten abgeglichen werden, um Rückschlüsse zu ziehen, wie viel Einfluss der Unternehmenskontext auf den Umgang mit technischen Schulden hat.

Anschließend werden die abgeleiteten Empfehlungen aus der Diskussion beleuchtet.

8.2.1 Ergebnisse

Die Ergebnisse der Arbeit sind in sechs Sektionen aufgeteilt und befassen sich mit den spezifischen Erkenntnissen, die aus den Interviews mit Gründern und Entwicklern aus drei verschiedenen Startups extrahiert werden konnten.

„Ansammlung technischen Schulden zu Beginn einer Startup- Unternehmung”

Ververs beschreibt, dass gerade der hohe finanzielle Druck Startups dazu treibt, zu Beginn technische Schulden aufzunehmen, obwohl diese, besonders im Bereich der Architektur, hohe Zinsen verursachen. Zudem birgt gerade die Arbeit mit neuen Technologien ein erhöhtes Risiko, unbewusst technische Schulden aufzunehmen. Neuen und kleinen Teams fehlen teils Qualitätsstandards, insbesondere bei Einmann-Projekten. Dies führe zu weiteren unbewussten technischen Schulden.

Das Umfeld zwischen dieser und der referenzierten Arbeit unterscheidet sich dahingehend, dass in Startups der Druck, technische Schulden aufzunehmen, dadurch entsteht, dass Kunden akquiriert werden müssen. Dies ist in den untersuchten Teams des Großunternehmens ein weniger ausgeprägter Faktor. Fachbereiche üben zwar Druck, in Form von Abgabefristen (Deadlines) für Funktionen, aus. Diese Abgabefristen sind jedoch nicht durch direkte Kundenakquiese bedingt, sondern werden vor allem in Gremien geplant.

Die wirtschaftliche Sicherheit der Entwicklungsteams bietet die Möglichkeit diesem Druck bis zu einem gewissen Maße entgegenzutreten, jedoch kommt es dabei

gerade zu Beginn zu „Eskalationen" [25]. Dies lässt einen Druck entstehen, der Teams trotz des unterschiedlichen Unternehmenskontexts dazu bringt, technische Schulden aufzunehmen, dies ist vor allem vor Meilensteinen, welche durch den Fachbereich eingefordert werden, zu beobachten.

> „War am Anfang schlimmer. Über Features, die versprochen wurden. Devs [Entwickler] haben dann oft gesagt, ohne uns, schaffen wir halt nicht. Es fällt was anderes dann runter. Das ist über die Jahre dann auch in den Gremien angekommen." (ADR 2022-07-23 - Team E)

Die Herausforderungen kleiner Teams lassen sich nicht in den Daten finden. Teams werden zumeist direkt mit mindestens sechs Personen (darunter vier Entwickler) aufgesetzt. Zudem werden diese aus einem Pool von Entwicklern zusammengestellt, die bereits Erfahrung in der Softwareentwicklung besitzen. Während der Datensammlung wurde kein Team gefunden, welches komplett aus neuen Personen zusammengesetzt wurde.

Schlussfolgerung

Die Aufnahme von technischen Schulden zu Beginn eines Projektes ist kaum ausgeprägt, zudem konnten keine Teams beobachtet werden, welche nur aus einer Person bestanden. Technische Schulden wurden zeitlich gesehen zumeist vor Abgabefristen und Meilensteinen aufgenommen statt zu Beginn von Projekten.

„Refactoring als wirtschaftlich bedeutende Entscheidung"

Ververs führt in seiner Arbeit aus, dass technische Schulden in der Architektur, die besonders den Software-Kern betreffen, ein erhebliches Maß an Zinsen verursachen. Zudem beschreibt er den enormen wirtschaftlichen Druck, durch Kunden und Investoren neue Funktionen zu liefern.

In den eigenen Beobachtungen wurden technische Schulden im Bereich der Software Architektur durchaus als problematisch angesehen. So litt das *Team N* besonders unter der Auswahl ihrer verwendeten Technologien im Software-Kern. Jedoch war zu beobachten, dass das Spannungsfeld durch die Begrenzung auf zumeist einen Kunden (den Fachbereich) weniger problematisch schien, als es in den von Ververs durchgeführten Interviews beschrieben wurde.

[25] Teams, die sich auf der Arbeitsebene nicht einigen können, „eskalieren" diese Themen zu ihren Vorgesetzten, in der Hoffnung, dass diese ihre Interessen durchsetzen.

Schlussfolgerung
In diesem Bereich der Ergebnisse und Beobachtungen tritt der wirtschaftliche Unterschied zwischen Startups und einem Großunternehmen durchaus hervor. Während in beiden Umfeldern finanzielle Aspekte eine Rolle spielen, unterscheiden sich die Konsequenzen erheblich. Längeres Ausbleiben von neuen Funktionen erschien den Befragten im Startup-Umfeld nahezu undenkbar, in den eigenen Beobachtungen berichteten Teams, dass dies durchaus schon vorgekommen ist. Am Ende sind die Konsequenzen eines vorzeitig beendeten Projekts weitaus harmloser als das Ende eines Startups.

„Vermeidung technischer Schulden"

Die Arbeit leitet anhand der geführten Interviews ab, dass erfolgreiche Startups versuchen im Alltag die Aufnahme von technischen Schulden zu vermeiden. Zudem gäbe es Unterschiede im Grad der technischen Schulden je nach Art und Komponente.

In einem der Interviews erwähnten zwei Entwickler aus dem *Team M*, dass, nach der ersten Live-Schaltung, der PO neue Funktionen zurückgestellt hatte, um eine Stabilisierung der Applikation zu ermöglichen. Dies weist indirekt darauf hin, dass technische Schulden in größerem Maße aufgenommen wurden. Aufgrund des fehlenden aussortierenden wirtschaftlichen Faktors scheint die Vermeidung technischer Schulden nicht zwingend notwendig zu sein, solange der Fachbereich bereit ist, schlechte Qualität entgegenzunehmen und anschließend die Funktionsentwicklung für längere Zeit der Tilgung von technischen Schulden unterzuordnen.

> MX: „Da haben wir ziemlich viel gemacht, haben viel im Backend refactored."
> EN: „Also die high traffic Stellen sind voll in Ordnung."
>
> (Interview 2023-05-10 - EN, MX)

Im Verlauf des Gespräches wurde weiter berichtet, dass der Fachbereich durchaus unzufrieden mit der Situation war.

Bezüglich der Verteilung technischer Schulden enthält das Beispiel einen weiteren Hinweis darauf, dass besonders in Codebereichen mit häufigen Änderungen („Hotspots") technische Schulden beseitigt wurden. Ähnliches wurde in anderen Teams beobachtet. Auch konnte beobachtet werden, dass gerade Dokumentation über den

Code hinaus eher vernachlässigt wurde. Diese wird häufig erst angefertigt, sobald ein Projekt an ein anderes Team übergeben werden soll.

Darüber hinaus wurde berichtet, dass die Qualität der Dokumentation auch zeitlich schwanken kann. Der *Entwickler EN* berichtet, dass sich nach seiner Erfahrung die Aufnahme und Tilgung von technischen Schulden bezüglich fehlender Dokumentation über die Zeit immer wieder abwechseln.

> „Nach der Migration nur auf Chores fokussiert, nicht nur technische Schulden, sondern alles mögliche auch Bugs. Danach Dokumentation hinterher ziehen. Ist die Erfahrung, dass man mal besser ist mit Dokumentation, dann mal wieder schlechter."
>
> (Interview 2022-11-11 - EN)

Schlussfolgerung

Abgesehen von technischen Schulden in Bereichen der Dokumentation vermeiden Teams zumeist die bewusste Aufnahme technischer Schulden. Falls, bedingt durch Abgabefristen, doch technische Schulden aufgenommen werden, planen Teams die aufgenommenen Schulden anschließend zurückzuzahlen. Die Ergebnisse treffen somit auf die eigenen Beobachtungen zu.

„Aufnahme technischer Schulden nach Projektanforderungen und Einfluss der Geschäftsbeziehungen"

Basierend auf den Erkenntnissen der Interviews hat Ververs abgeleitet, dass Teams in Situationen hohen Drucks technische Schulden aufnehmen, um diese in ruhigeren Zeiten wieder abzubauen.

Ein ähnlicher Effekt ist in den eigenen Daten beobachtet worden. Vor Deadlines nehmen die Teams technische Schulden eher auf und hoffen, diese anschließend zurückzahlen zu können, jedoch sind ruhigere Phasen selten absehbar. Durch interne Abhängigkeiten gibt es immer wieder Phasen mit wenigen fachlichen Anforderungen, diese werden manchmal für die Tilgung von technischen Schulden genutzt. Ein entsprechendes, klares Muster dieser Phasen konnte nicht abgeleitet werden.

Der *Entwickler DJ* beschreibt ruhigere Phasen eher als Chancen, technische Aufgaben (darunter die Bereinigung von technischen Schulden) abzuarbeiten, denn als planbare Phasen.

> Der Zeitpunkt ist entscheidend, es gibt Phasen, die ruhiger sind. Auch im Pairing, wenn da mal einer alleine ist, dann kann man einen Aufräum-Chor dazwischen schieben, oder auch in der Urlaubszeit, wenn wenig los ist.
>
> (Interview 2023-06-15 - DJ)

Schlussfolgerung
Die beschriebenen Beobachtungen bezüglich der Aufnahme von technischen Schulden unter Druck konnten auch in den eigenen Daten gefunden werden, anhand der eigenen Beobachtungen konnte jedoch keine Regelmäßigkeit von ruhigeren Phasen gefunden werden. Abgesehen davon sind die Rückschlüsse übertragbar.

„Qualitätsansprüche der Entwickler"

Die Qualitätsansprüche der Entwickler wurden als intrinsische Motivation identifiziert, welche gegebenenfalls bedingt, dass technische Schulden ungern aufgenommen werden. Die Aufnahme von technischen Schulden könnte hingegen zu Frust und einer Minderung der Qualitätsansprüche führen.

In den eigenen Daten konnte dies nicht vollumfänglich nachvollzogen werden. Während Entwickler durchaus den Anspruch ausdrückten, qualitative Software schreiben zu wollen, konnte nicht beobachtet werden, dass Entwickler durch die Aufnahme von technischen Schulden ihre Qualitätsansprüche senken. Jedoch konnten Beispiele von auftretendem Frust gefunden werden, wenn Entwickler in Bereichen mit hohen technischen Schulden arbeiteten, diese also nicht mit ihren Qualitätsanforderungen übereinstimmten. Diese wurden zumeist in Retrospektiven als negative Punkte angebracht.

Andererseits freuen sich Entwickler, wenn eine Möglichkeit gefunden wurde, keine technischen Schulden aufzunehmen. So freut sich *SM* aus *Team S* über die gefundene Lösung, welche zwar mehr Aufwand benötigt, jedoch eine leichtere und höhere Anpassbarkeit verspricht: „Macht mich sehr glücklich." (IPM 2022-05-11 - Team S)

Schlussfolgerung
Sowohl die Qualitätsansprüche der Entwickler als auch Frustration durch technische Schulden konnten beobachtet werden, jedoch nicht die Absenkung der Qualitätsansprüche durch technische Schulden. Somit sind die Thesen nur teilweise übertragbar.

„Auswirkungen der technischen Expertise der Entscheider"

In den Startups wurden mehrere Beispiele gefunden, welche darauf hindeuten, dass technische Expertise der Entscheider eine Auswirkung auf den Umgang mit technischen Schulden hat. Dies kann sich in höheren Qualitätsansprüchen oder einer merklichen Großzügigkeit mit Zeit zur Tilgung von technischen Schulden ausdrücken.

In der Struktur des Großunternehmens haben Teams, die für einen Fachbereich entwickeln, einerseits ihren eigenen Chef, der für die Softwareentwicklung verantwortlich ist, jedoch keine fachliche Verantwortung hat. Andererseits haben sie einen PO vom Fachbereich, welcher die fachliche Verantwortung trägt und in Bezug auf die Priorisierung den höchsten Einfluss hat.

In den Ausführungen der befragten Entwickler wurde häufig erwähnt, dass POs mit technischer Vorerfahrung mehr Verständnis für technische Belange wie die Tilgung technischer Schulden haben.

> „Haben wir mit unserem PO und PM gesprochen, dass wir das erstmal so machen, dass es funktioniert, dass wir dann aber auch ein paar Tage brauchen, um das dann vernünftig zu machen. [...] Feedback [vom PO] grundsätzlich sehr gut, weil der auch sehr technisch orientiert ist, er kommt aus der IT, der weiß auch, was es bedeutet, wenn die Wartung leidet. In anderen Projekten habe ich schon mitbekommen, dass es zu Unmut führt. Wo der PO dann gesagt hat „Ich bezahle euch doch nicht fürs Aufräumen.""
>
> (Interview 2023-06-15 DJ)

Aufgrund der organisatorischen Struktur müssen die Vorgesetzten der Entwickler nicht zwingend einen technischen Hintergrund haben, so ist in Gesprächen mit Vorgesetzten aufgefallen, dass das Wissen über das Konzept der technischen Schulden schnell abnimmt, je höher der Hierarchie nach oben gefolgt wird.

Entwickler schätzen dies jedoch nicht als problematisch ein, da ihre Vorgesetzten kaum Einfluss auf den Arbeitsalltag und die Priorisierung im Projekt haben.

Schlussfolgerung

Technische Expertise in den Fachbereichen, welche den größten Einfluss auf die Priorisierung haben, wurde mehrfach positiv erwähnt. Die grundlegenden Aussagen treffen zu, hierbei muss beachtet werden, dass die Vorgesetzten der Entwickler kaum

Einfluss auf das Projekt an sich haben und keine Auswirkung ihrer technischen Expertise gefunden wurde.

8.2.2 Anwendbarkeit der Empfehlungen nach Ververs

Im Folgenden werden die Empfehlungen aus der Arbeit von Ververs mit den eigenen Daten und Erkenntnissen abgeglichen:

> „Das langfristige Halten von Entwicklern durch das Sicherstellen einer hohen Zufriedenheit und Motivation ist kritisch für den Unternehmenserfolg und sollte deshalb immer als ein Faktor bei betriebswirtschaftlichen Entscheidungen berücksichtigt werden." (Ververs 1. Empfehlung)

Während der Beobachtungszeit hat kein Mitarbeiter aus den Teams das Unternehmen verlassen, jedoch kam es mehrfach zum Wechsel zwischen Teams. Von diesen wurden jedoch keine eigenen Erkenntnisse abgeleitet.

Es konnte beobachtet werden, dass fehlendes Wissen bei Übergaben von Komponenten von einem Team zu einem anderen Team durchaus zu Herausforderungen führte. In beiden beobachteten Fällen, in denen ganze Komponenten wegen zu hoher technischer Schulden neu implementiert wurden, waren diese Komponenten von einem anderen Team übernommen worden. Dies mag ein interessantes Phänomen sein, jedoch ist hier nicht das Halten von Entwicklern das grundlegende Problem, auch wenn die Auswirkungen Ähnlichkeiten aufweisen könnten.

> „Technische Schulden sollten generell vermieden werden, da das langfristige Arbeiten mit einer qualitativ schlechten Codebasis frustrierend für die Entwickler sein kann." (Ververs, 2. Empfehlung)

Nach den eigenen Beobachtungen bergen sowohl die Aufnahme als auch die Tilgung von technischen Schulden Frustpotential für Entwickler (siehe Kapitel 7.5). Diese Empfehlung stimmt mit der eigenen Beobachtung überein.

„Inwieweit technische Schulden den Arbeitsalltag der Entwickler negativ beeinflussen, ist abhängig von der Art der technischen Schulden, der Komponente und der Zusammensetzung des Entwicklerteams. So leidet ein kleines, eingespieltes Entwicklerteam weniger unter unzureichender Dokumentation, während dies für ein größeres Team und/oder mit viel Fluktuation kritischer wäre. Diese unterschiedliche Schwere in den Konsequenzen von technischen Schulden ist bei der Aufnahme zu berücksichtigen und kann vermutlich am besten vom Entwicklerteam beurteilt werden, weshalb dieses bei Entscheidungen zur Aufnahme und zum Abbau von technischen Schulden eingebunden werden sollte." (Ververs, 3. Empfehlung)

Wie in Kapitel 7.5 beschrieben, gehen Entwickler mit technischen Schulden durchaus unterschiedlich um. Zudem konnte beobachtet werden, dass Entwickler unterschiedliche Arten von technischen Schulden priorisieren, so achtete der *Entwickler DJ* stärker auf die Architektur-Dokumentation als der Rest des Teams. Daher wird der Vermutung zugestimmt, dass das Entwicklerteam dies selbst am besten beurteilen kann. In den eigenen Beobachtungen waren alle Teams an den Entscheidungen zur Aufnahme und zum Abbau von technischen Schulden beteiligt. Falls die Entscheidungen ein vertretbares Maß an Aufwand nicht übertreffen, wurden diese Entscheidungen teilweise autonom ohne weitere Kommunikation mit den Fachbereichen getroffen.

„Zu Beginn einer Startup-Unternehmung oder nach Projektanforderungen kann die Aufnahme von technischen Schulden dazu genutzt werden, den betriebswirtschaftlichen Druck abzufedern. Langfristig gesehen sollten diese allerdings wieder abgebaut werden, um die Entwicklung nicht zu stark negativ zu beeinflussen." (Ververs, 4. Empfehlung)

Im *Team M* wurden zu Beginn ein hohes Maß an technischen Schulden aufgenommen, was zu einer langen Stabilisierungs-Phase führte und die Zusammenarbeit mit dem Fachbereich nachhaltig beeinträchtigt hat. So beschreibt der *Entwickler EN*:

> „Ansonsten habe ich generell gemerkt, das liegt [...] an der Historie von der Anwendung, dass da so eine leichte Resistenz drin war, weil kurz bevor ich gekommen bin, wurde sehr viel Zeit in Refactoring reingesteckt [...], da wurde letztes Jahr echt viel refactored. Da war auch die Bereitschaft vom Fachbereich manchmal nicht so hoch, an [technischen Schulden] zu arbeiten [...], „wir haben doch gerade so viel refactored, warum wollen wir schon wieder was machen?" " (Interview 2023-06-12 1041 - EN)

In anderen *Teams (S und F)* führten die angesammelten technischen Schulden von Komponenten dazu, dass diese neu entwickelt wurden.

„Das Entwickeln vieler Prototypen kann dabei helfen unbewussten technischen Schulden beispielsweise bei der Gestaltung der Architektur oder der Auswahl von Bibliotheken und Frameworks vorzubeugen." (Ververs, 5. Empfehlung)

Die Nutzung von Prototypen bzw. Spikes (siehe Kapitel 7.4) wurde mehrfach nachgewiesen. Diese dienten sowohl der Validierung einer neuen Technologie als auch der Abschätzung von Aufwand und Mehrwert eines Umbaus. Dementsprechend kann diese Empfehlung sogar erweitert werden. Zu beachten ist, dass Prototypen bzw. Spikes neben den positiven Effekten auch Risiken beinhalten, ein großes Maß an technischen Schulden aufzunehmen, wenn diese ohne weitere Änderungen übernommen werden. Eine genauere Beleuchtung dieses Aspekts ist in dem Kapitel 7.4 bereits beschrieben.

„Das Anwachsen des Entwicklerteams kann dazu genutzt werden, technische Schulden sichtbarer zu machen. Gleichzeitig ist das Beseitigen von technischen Schulden eine gute Einstiegsaufgabe für neue Entwickler, um sich mit dem Code vertraut zu machen." (Ververs, 6. Empfehlung)

Aus den Interviews und in dem begleiteten Team sind während der Beobachtung keine nennenswerten Phänomene bekannt, die diese Empfehlung unterstützen. Jedoch bietet das *Team S* eine Möglichkeit an, dass andere Entwickler in dem eigenen Team hospitieren.

> „While not all devs have the posibility within a product, to learn about the necessary steps to migrate to [a new platform], we offer a 4-6 weeks hospitation program within our team. You will have the chance to work on infrastructure heavy stories and focus only on learning the required knowledge for the migration or usage of a new technology in a production environment." (Internes Hospitation Angebot)

In der Vergangenheit wurden mit den hospitierenden Entwickler zumeist Infrastrukturanpassungen durchgeführt, welche unter anderem auch technische Schulden tilgten. Dies bot eine besondere Chance die bestehenden Lösungen zu erkunden und diese anschließend zu verbessern.

8.2.3 Zusammenfassung

Die Thesen aus der Arbeit „Umgang mit technischen Schulden im Startup-Umfeld" von Ververs [129] sind in großen Teilen mit den eigenen Beobachtungen und Ergebnissen deckungsgleich. Unterschiede findet man zumeist nur in Thesen, die einen direkten

Bezug zu den wirtschaftlichen Zwängen einer Startup-Unternehmung haben. Die Annahme, dass dies aus der höheren Jobsicherheit im Großunternehmen resultiert, liegt nahe.

Die Empfehlungen widersprechen keinem der eigenen Ergebnisse, zudem konnten positive Effekte gefunden werden, wenn die Teams handelten, wie in den Empfehlungen beschrieben. Darüber hinaus konnte vor allem beobachtet werden, dass die Rotation von Entwicklern und deren Wissen negative Effekte auf die Weiterentwicklung eines Systems haben kann.

Die Empfehlung, viele Prototypen zu entwickeln, um die Aufnahme unbewusster technischer Schulden zu vermeiden, scheint aufgrund der eigenen Beobachtungen aus Kapitel 8.3.4 die Risiken zu vernachlässigen. Dies kann bei unbedachten Umgang mit der entstandenen Codebasis, des Spikes, zu einem massiven Wachstum technischer Schulden führen. Dies bringt bekannte Auswirkungen von technischen Schulden mit sich.

8.3 Diskussion der Ergebnisse

Im Folgenden werden die abgeleiteten Thesen und Schlussfolgerungen aus dem Kapitel 7 diskutiert.

8.3.1 Bewusstsein in der Organisation

In einer Organisation von Nicht-Software zentrischen Großunternehmen ist auch in technischen Abteilungen und deren Hierarchie das Konzept der *technische Schulden* nicht durchgängig bekannt und beachtet. Dies hat zur Folge, dass kaum Richtlinien oder Empfehlungen zum Umgang mit technischen Schulden etabliert sind.
Darüber hinaus sollte bei der Einführung von Handlungsempfehlungen großer Wert auf die Bekanntmachung gelegt werden.

Abgesehen von den Forschungen in Startups wurde keine Literatur gefunden, welche Angaben macht, ob sich das Bewusstsein über *technische Schulden* abhängig von der Hierarchie in einem Unternehmen ändert. In den Arbeiten über Startups liegt es jedoch eher daran, dass es keine tieferen Hierarchien gibt. Hier sind die Firmengründer meist mit im Team oder haben in softwarezentrierten Startups sogar selber Software beigesteuert.

Wie in Kapitel 4.7 berichtet, erhebt die bisherige Literatur (im Besonderen im Bereich Großunternehmen) ihre Daten meist über Umfragen oder Interviews, hierbei konnte keine Studie gefunden werden, die die organisatorischen Strukturen der Unternehmen untersuchte.

8.3.2 Begriff der *technische Schulden* in Softwareentwicklungteams

Eine einheitliche Definition von *technischen Schulden* ist in der Praxis nicht etabliert. Entwickler und PMs tendieren dazu, jegliche technische Aufgabe, die nicht explizit auf einen Kundennutzen einzahlt, als technische Schuld zu bezeichnen.

Dieses Ergebnis ist im Einklang mit der derzeitigen Forschung. Die Tertiärstudie von Helvio Jeronimo Junior et al. von 2022 beschreibt den Forschungsstand nach der Analyse von 19 Sekundärstudien und fasste damit über 532 Primärstudien zusammen:

> „Technical Debt (TD) kontextualisiert interne Qualitätssoftwareprobleme mit verschiedenen Arten, spezifischen Ursachen und Auswirkungen. Dennoch neigen Forscher und Software-Praktiker immer noch dazu, TD mit jedem Problem zu verwechseln, das während des Software-Lebenszyklus auftritt, was das richtige Verständnis und Management beeinträchtigt. Das mangelnde Verständnis von TD im Kontext von Software-Projekten

und die Frage, wie man damit umgeht, gefährden Software-Wartungs- und Entwicklungsaktivitäten." (Übersetzt aus dem Englischen) [59]

Es wird weiterhin an der Definition von technischen Schulden geforscht, zudem existieren neuere Forschungsarbeiten, die von dem Begriff „technische Schulden" Abstand nehmen, um stattdessen den Begriff „technische Kompromisse" zu verwenden [100].

Eine alternative Herangehensweise an die bisherigen Forschungsergebnisse stellen C. Jaspan und C. Green 2023 in ihrer Veröffentlichung „Defining, Measuring, and Managing Technical Debt" dar [55]. Sie arbeiten bei Google[26] als Forscher im Bereich der technischen Produktivitätsforschung und führen seit 2018 regelmäßige Entwicklerzufriedenheits-Befragungen durch. Ihr Ziel war es, zu verstehen, was Ingenieure unter dem Begriff „technische Schulden" verstehen, anstatt dies vorher zu definieren.

Im Verlauf mehrerer angepasster Umfragen erreichten sie, mit den folgenden zehn Kategorien, eine signifikante Abdeckung über das Verständnis *technischer Schulden*:

Migration ist benötigt oder wird durchgeführt
Dies kann motiviert sein durch die Notwendigkeit der Skalierung aufgrund von Auflagen, die Verringerung von Abhängigkeiten oder die Vermeidung veralteter Technologien.

Dokumentation zu Projekt- und Anwendungsprogrammierschnittstellen
Informationen darüber, wie ein Projekt funktioniert, sind schwer zu finden, fehlen oder sind unvollständig.

Tests
Schlechte Testqualität oder -abdeckung, wie fehlende Tests oder unzureichende Testdaten, führen zu Fragilität, fehlerhaften Tests oder vielen Rollbacks[27].

Code-Qualität
Die Produktarchitektur oder der Code innerhalb eines Projekts war nicht gut konzipiert. Möglicherweise war es übereilt erstellt oder ein Prototyp/Demo.

[26] Google stellt mit über 100000 Mitarbeitern ein Großunternehmen dar.

[27] Ein Rollback bezeichnet ein Zurückwechseln auf ein alten Version, aufgrund eines Fehlers in der aktuellen.

Toter und/oder aufgegebener Code

Code/Features/Projekte wurden ersetzt oder verdrängt, aber nicht entfernt.

Verschlechterung des Codes

Die Codebasis hat sich verschlechtert oder ist nicht den sich verändernden Standards angepasst. Der Code kann sich im Wartungsmodus befinden und Refactoring oder Aktualisierungen benötigen.

Dem Team fehlt das nötige Fachwissen

Dies kann auf personelle Lücken und Fluktuation oder auf geerbte verwaiste Codebasen zurückzuführen sein.

Abhängigkeiten

Abhängigkeiten sind instabil, ändern sich schnell oder lösen Fehler aus.

Die Migration wurde unzureichend durchgeführt oder aufgegeben

Dies kann dazu geführt haben, dass zwei Versionen gewartet werden müssen.

Freigabeprozess

Der Rollout und die Überwachung der Produktion müssen aktualisiert, migriert oder gepflegt werden.

In der Arbeit wird deutlich, dass die Forscher (angestellt in einem softwarezentriertes Großunternehmen) Unterschiede zwischen der Definition von „technischen Schulden" in der Forschung und dem grundlegenden Verständnis in der Praxis wahrnimmt. Sie erwähnen zudem, Schwierigkeiten von ihnen definierte technische Schulden mittels Metriken zu erkennen. Sie weisen darauf hin, dass sie sich unsicher sind, ob es für alle ihrer Kategorien überhaupt Metriken gibt, da diese gegebenenfalls nicht einen Zustand beschreiben, sondern die Abweichung des Ist-Zustand mit einem nicht umgesetzten Ideal-Zustand.

> „Das Urteil eines Ingenieurs über die technische Schuld betrifft sowohl den gegenwärtigen Zustand als auch den möglichen Zustand [Ideal Zustand]". (Übersetzt aus dem Englischen) [55]

8.3.3 Identifikation von technischen Schulden

Schlussfolgerung
Die Einschätzung, welche Umsetzung für die Lösung eines Problems geeignet ist oder ob es doch einen besseren Weg gibt, hängt von vielen Einflussfaktoren ab, die sich auf die individuellen Sichten der Entwickler auswirken. Dies beeinflusst zudem, welche Umsetzungen als bewusste Aufnahme von technischen Schulden wahrgenommen werden.
Für einige Einflussfaktoren konnten Beispiele beobachtet werden, jedoch ist aufgrund der Komplexität menschlicher Entscheidungen davon auszugehen, dass weitaus mehr existieren. Eine genauere Untersuchung und Kategorisierung mag sinnvoll sein, ist jedoch nicht Bestandteil dieser Arbeit.

Der derzeitige Forschungsstand beleuchtet kaum den menschlichen Faktor und Einflüsse auf die Wahrnehmung von technischen Schulden. Es wurden zwei Arbeiten gefunden, welche einen Ideal-Zustand referenzieren, der im Zusammenhang mit technischen Schulden steht. Dies sind „An Empirical Model of Technical Debt and Interest" aus dem Jahr 2011 von Nugroho et al. [94] und die bereits erwähnte Arbeit aus dem Jahr 2023 „Defining, Measuring, and Managing Technical Debt"[55] von Jaspan et al.

Nugroho et al. definieren technische Schulden als „die Kosten für die Verbesserung, die technische Qualität auf ein Niveau zu bringen, das als ideal angesehen wird".

Während dies kaum Ähnlichkeit hat zu der eigens gewählten Definition von *technischen Schulden*, erscheint es durchaus erforschenswert, anhand welcher Kriterien Entwickler erkennen, dass es eine bessere Umsetzung gibt, die keine „Entwurfs- oder Implementierungskonstruktionen" erzeugen, welche interne Systemeigenschaften beeinträchtigen.

Jaspan et al. weisen in ihrer Arbeit darauf hin, dass sie Schwierigkeiten haben, Metriken für technische Schulden (nach ihrem Verständnis) zu finden. Zudem könnte es sein, dass Metriken für einige Arten gar nicht gefunden werden können, da diese Schulden keinen Ist-Zustand beschreiben, sondern eine Relation zwischen dem Ist-Zustand und einem Ideal-Zustand.

Die eigenen Schlussfolgerungen erscheinen im Kontext der spärlichen Forschung und einigen Randerwähnungen eines Ideal-Zustandes als durchaus plausibel und laden zu weiteren Forschungen ein.

Schlussfolgerung
Best Practices haben einen markanten Einfluss auf den Umgang mit technischen Schulden, da Teams zumeist ihre positive Wirkung auf die Erweiterbarkeit und Wartbarkeit nicht in Frage stellen. *Globale, firmen und individuelle Best Practices* werden im Team als *Team Best Practices* übernommen.
Anschließend identifizieren Teams unbewusst aufgenommene *technische Schulden* in Codebasen, welche den neu gelernten *Best Practices* entgegenstehen. Es konnte beobachtet werden, dass Teams die Tilgung dieser unbewussten technischen Schulden durchaus unter wirtschaftlichen Aspekten bewerten und diese als bewusste *technische Schulden* aufnehmen.

Wie bereits in Kapitel 4 zusammengefasst, verweist die Arbeit von Besker et al. [12] auf die Tendenz hin, dass Entscheidungen bezüglich technischer Schulden häufig durch das „Bauchgefühl" der Beteiligten beeinflusst wird [130, 43].

Gerald Weinberg beschrieb in seinem Buch „Quality Software Management (Vol. 1): Systems Thinking" die Relativität der Qualität und erörterte, dass Qualität abhängig vom Menschen ist, der sie betrachtet.

> „Qualität bedeutet, den Anforderungen einer Person zu entsprechen. Jede Aussage über Qualität ist eine Aussage über eine Person(en)." (Übersetzt aus dem Englischen) [134]

Die gefundenen Ergebnisse gliedern sich in die bestehende Literatur ein und beschreiben detailliertere Beobachtungen aus der Praxis.

8.3.4 Spikes: Vor- und Nachteile in Bezug auf technische Schulden

Schlussfolgerung
Spikes können Teams helfen, Kosten und Nutzen von technischen Änderungen zu validieren, hierzu zählt auch die Tilgung technischer Schulden.
Code, welcher während Spikes entstanden ist, birgt häufig ein erhöhtes Maß an technischen Schulden, sollte er übernommen werden. Es sollte Zeit eingeplant werden, um die *technische Schulden*, die durch ein Übernahme der Codebasis entstehen, direkt zu tilgen.
Zudem ist es sinnvoll, Spikes klar zu definieren, bevor sie gestartet werden. Wichtig ist die zu überprüfende Hypothese und der geplante Aufwand, zudem sollte sich das Team vorher einigen, welches Ergebnis welche Konsequenzen hat. Anhand dieser Rahmenbedingungen sollte abgeleitet werden, welche Codestandards im Austausch gegen Schnelligkeit geopfert werden können.

Die Empfehlung basiert auf den gesammelten Aussagen und Erfahrungen der beobachteten Teams. Spikes wurden vor und während der Beobachtungszeit genutzt, um neue Technologien zu validieren oder Abschätzungen über den Aufwand und Nutzen einer Änderung besser abschätzen zu können.

Spikes bzw. Prototypen werden in der Literatur zu technischen Schulden unter dem Aspekt der technischen Validierung und dem Einsatz in Startups erwähnt. Darüber hinaus berichten Borowa et al. in ihrer Arbeit „Living With Technical Debt— A Perspective From the Video Game Industry" über die Gefahren von Prototypen [16].

> „Die meisten dieser Prototypen werden aufgegeben und nicht in das endgültige Spielprodukt übernommen. Daher verwenden die Entwickler nicht allzu viel Mühe auf ihre Qualität. Infolgedessen enthalten Feature-Prototypen eine große Menge potenzieller technischer Schulden, die sich in tatsächliche technische Schulden verwandeln, wenn die Prototyp-Features weiterentwickelt werden oder ein neues Feature auf ihrer Grundlage erstellt wird." (Übersetzt aus dem Englischen) [16]

Die Autoren Gauthier et al. gehen zudem genauer auf die Abwägungen zwischen Aufnahme und Tilgung von technischen Schulden, die während der Entwicklung von Prototypen auftreten, ein [38].

Die erarbeitete Empfehlung fügt sich in das Gesamtbild über Prototypen ein, geht darüber hinaus auf die Nutzung von Spikes für die Kosten/Nutzen-Abschätzung ein.

8.3.5 Unzufriedenheit in Folge *technische Schulden*

Schlussfolgerung
Die bewusste Aufnahme von technischen Schulden kann zu Unzufriedenheit bei Entwicklern führen, jedoch wurden Beispiele gefunden, in denen Entwickler die Tilgung von technischen Schulden als ähnlich frustrierend empfanden. Generell wurde beobachtet, dass die Arbeit in Codebereichen mit hohen technischen Schulden zu Unzufriedenheit führen kann, unabhängig von den durchzuführenden Änderungen. Dem entgegen stehen jedoch Beobachtungen, dass ein Entwickler genau dies als spannende Herausforderung betrachtete, es kann also empfohlen werden, diesbezügliche Veranlagungen bei Entwicklern in Erfahrung zu bringen und dieses Wissen gezielt einzusetzen.

Es gibt bereits erste Studien über die Auswirkung von technischen Schulden auf die Teammoral [39, 15]. In diesen wird beschrieben, dass das Auftreten von technischen Schulden zumeist in fehlendem Fortschritt und Zeitverschwendung resultiert. Die Fallstudien zeigten, dass dies die Entwickler beeinflusst, dabei war die Aufnahme von technischen Schulden eher negativ behaftet.

Die Schlussfolgerung dieser Arbeit speist sich aus Beschreibungen von Entwicklern und spontanen Reaktionen in Gesprächen. Die gegebene Empfehlung ist eine Ableitung der gefundenen Beispiele.

8.3.6 Dokumentation von technischen Schulden

Schlussfolgerung
Teams dokumentieren *technische Schulden* als technische Aufgaben, die die Tilgung betreffen. Fehlende Details, wie die Beschreibung der Auswirkungen, dem geschätzten Aufwand oder zu zahlenden Zinsen, führen dazu, dass sich *technische Schulden* ansammeln. Einträge im Backlog verlieren den Kontext und werden schließlich gelöscht.
Teams gehen dazu über nur noch die Tilgung von akut auftretenden *technischen Schulden* zu dokumentieren, wenn eine Umsetzung in wenigen Wochen möglich erscheint.

Die Forschung beleuchtet nur selten, wie *technische Schulden* in Softwareorganisationen dokumentiert werden. Bisherige Ergebnisse darauf hin, dass es vielseitige Probleme mit der Dokumentation gibt [86, 76, 136].

Die eigenen Ergebnisse basieren auf den Aussagen von Entwicklern aus mehreren Teams und den Beobachtungen im *Team S*. Zudem wurden die beschriebenen Folgen in der Praxis beobachtet und passen zu der gefundenen Literatur. Es ist davon auszugehen, dass, insbesondere in Softwareentwicklungsteams ohne strukturierte Vorgaben bezüglich des Umgangs mit *technischen Schulden* oder einer expliziten Unterstützung durch Werkzeuge, ähnliche Phänomene auch in anderen Unternehmen und Kontexten auftreten.

Schlussfolgerung
Sowohl die Markierung von noch ausstehenden Anpassungen (Todos) als auch die Markierung von gutem Beispielcode können helfen, technische Schulden zu vermeiden. Ein Entwickler, welcher eine Codestelle mit einer ähnlichen Funktion sucht, kann hier sowohl positives als auch negatives Feedback erhalten, ob der Code sich als Vorlage eignet.

Die Verwendung von „Todos" im Code wird in der Literatur über technische Schulden als Indikator für akzeptierte technische Schulden (self admitted technical debt) verstanden. Besonders in den Jahren seit 2014 und der Arbeit von Potdar [103] wird aktiv an dieser Art der Schulden, aber auch an ihrer Erkennung geforscht.

Zudem wird diese Art von Vorgehen sogar empfohlen, Beispiel hierfür ist die Beschreibung von Aktivitäten von Kruchten [64].

Neu hingegen ist das Markieren von guten Codestellen, welches in der Forschung von technischen Schulden noch nicht beleuchtet wurden. Neben den positiven Effekten, dass Entwickler Codestellen erkennen, die vom Team als „so soll es sein/ das ist der beste Weg" angesehen werden, könnte es auch andere Effekte geben, die nicht betrachtet wurden.

So wie „Todos" veralten können durch eine Änderung des Kontext, so ist dies ggf. auch für positive Codestellen der Fall. Zudem ist nicht klar, ob die mehrfache ähnliche Verwendung von Code nicht darauf hindeutet, dass eine Abstraktionsschicht fehlt und dies selbst eine technische Schuld ist, bzw. dazu führt.

Die positiven Berichte über diese Technik stammen aus nur einem Team, andere Teams nutzten diese noch nicht, wobei durchaus Berichte aus *Team S* aufzeigen, dass die Orientierung an schlechten Codestellen zu Problemen geführt hat und die erwähnte Technik durchaus helfen könnte.

Zusammenfassend bestätigen die Erkenntnisse bereits etabliertes Wissen aus der Forschung, bringen jedoch ein weitere Technik zu Tage, die durchaus weiter untersucht werden sollte.

8.3.7 Priorisierung und Abbau von technischen Schulden

Schlussfolgerung
Während die Tilgung von technischen Schulden zumeist als hinderlicher Aufwand angesehen wird, können diese Aufgaben auch zum Vorteil eingesetzt werden. Abwechslung in den Aufgaben, Motivation der Entwickler, Schulungszwecke oder das Einbinden eines neuen Entwicklers sind durchaus Beispiele für eine gewinnbringende Verwendung *technischer Schulden*. Zudem wurde beobachtet, dass besonders in Teams, die Pair Programming praktizieren, technische Schulden häufig zurückgezahlt werden, wenn ein Entwickler alleine keine Story umsetzen möchte.

In der tertiären Studie von Artigo et al. werden unter anderem auch die Auswirkungen von technischen Schulden zusammengefasst. Neben den kurzfristigen positiven Effekten für eine schnellere Bereitstellung von Funktionen werden auch positive Auswirkungen auf der Teamebene aufgelistet. Dies kann zum Beispiel eine steigende Moral sein, durch die Möglichkeit, gewisse Aufgaben auf später verschieben zu können, um Lieferanforderungen zu erfüllen.

Darüber hinaus wird in der Arbeit von Ververs empfohlen das Beseitigen von technischen Schulden als Einstiegsaufgaben für neue Entwickler zu nutzen [129].

Es ist also durchaus denkbar, dass technische Schulden weitere Vorteile bieten können, die über den kurzzeitigen Geschwindigkeitsvorteil hinausgehen. Hierbei mag es sein, dass dies Gründe sind, die die Aufnahme von technischen Schulden nicht bestärken, jedoch könnten sie die negativen Folgen ausgleichen.

Ein hypothetisches Beispiel wäre, dass ein Team eine neue Automatisierung erstellen möchte, um lästige manuelle Schritte abzulösen. Eine solche Aufgabe würde sich ideal eignen, einem neuen Mitarbeiter die Möglichkeit zu geben, dem Team diese Arbeit abzunehmen und gleichzeitig benötigte Schritte zu verstehen und anschließend zu automatisieren. Zudem könnte dies das Ansehen des neuen Mitarbeiters steigern, weil seine ersten Ergebnisse dem Team direkt lästige Schritte ersparen.

Anhand der gefundenen Beispiele aus Teambesprechungen und dem Wissen über vergangene Hospitationen im *Team S*, kann geschlossen werden, dass durchaus weitere solcher positiven Auswirkungen in der Praxis zu finden sein könnten.

8.4 Abgeleitete Empfehlungen/Grounded Theory

Ein großer Teil der Funde lässt sich in der folgenden fundierten Theorie zusammenfassen.

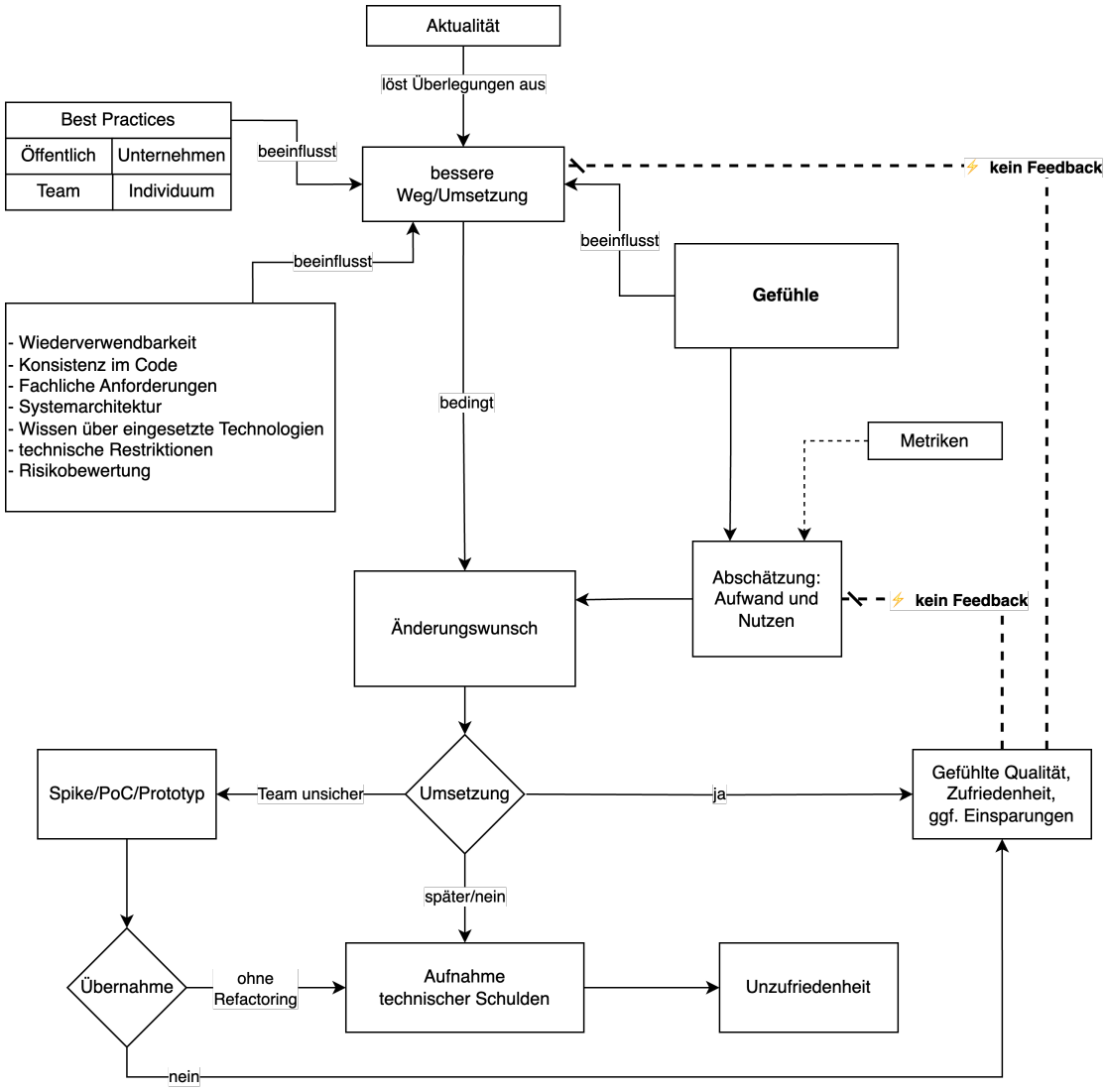

Abbildung 10: Grounded Theory

Softwareentwicklungsteams bestehen aus mehreren Individuen, die jeweils eine eigene Perspektive, Wissensstand und Erfahrung besitzen.

Die individuellen Ansichten aller Beteiligten bilden einen vom Team akzeptierten Kompromiss, welcher die „geeignete Umsetzung" für die zu lösenden Aufgaben beschreibt. (7.3.1)

Ausgelöst durch die Arbeit an Code, der nicht den Ansprüchen eines Entwicklers entspricht, beginnt dieser sich Gedanken über Anpassungen zu machen, welche in

einem Änderungswunsch resultieren können. Dieser basiert auf den Vorstellungen einer besseren Umsetzung des individuellen Entwicklers.

Die Vorstellung eines Entwicklers bezüglich einer geeigneten Umsetzung ist zu einem großen Anteil durch subjektive Faktoren beeinflusst. Zu diesen zählen unter anderem

- Wiederverwendbarkeit der Software,
- Konsistenz im Code,
- Wissen und Interpretation von fachlicher und nicht-fachlicher Anforderungen,
- Wissen über eingesetzte Technologien
- Erfahrung mit Systemarchitektur,
- technische Restriktionen und
- Risikobewertung.

Dazu kommen bekannte, vorgegebene oder selbst erarbeitete „Best Practices" (7.3.2). Die Beobachtungen weisen darauf hin, dass die Aktualität, also dass an einer Codestelle gearbeitet wird oder werden soll, einen starken Einfluss hat, in welchem Bereich Entwickler eine Änderung als relevant erachten.

Aus der Idee einer geeigneteren Umsetzung entstehen Änderungswünsche an der Software. Diese würden, so meist die Argumentation, die gewünschte Funktion schneller, besser, idiomatischer oder sauberer bereitstellen, der Code wäre dadurch anschließend leichter erweiterbar oder mit weniger Wartungsaufwand verbunden. Hierbei stützen sich die Entwickler bei der Schätzung des Aufwands und Nutzens auf ihre Gefühle, subjektive Empfindungen, die selten durch überprüfbare Zahlen validiert werden können. Es bedarf Vertrauen des PM/PO zum Team, zum Entwickler.

Metriken werden nur in seltenen Fällen herangezogen, dies wird dadurch begründet, dass es nur schwierig möglich sei, geeignete Metriken zu finden, zudem stehe der Aufwand, die benötigten Daten zu sammeln, nicht im Verhältnis zur Fragestellung.

Es konnte beobachtet werden, dass Entwickler durchaus dazu tendieren, einen Nutzen darin zu sehen, dass sie Code nach ihren eigenen Vorlieben gestalten, mit der Begründung, Änderungen könnten dann schneller umgesetzt werden. Es kommt in diesen Fällen dann häufig darauf an, welche Aktualität der Code besitzt und wie hoch der Aufwand einer Anpassung eingeschätzt wird. Liegen keine anderen Aufgaben im Backlog, die eine hohe Priorität haben, nehmen sich Entwickler „zwischendurch" die Zeit, um solche als „klein" wahrgenommenen Aufgaben zu erledigen. (7.7)

Entscheidet sich das Team für die Umsetzung der Änderungswünsche, wird häufig die wahrgenommene Qualität der Software verbessert, einzelne Entwickler oder das ganze Team sind zufriedener. Es konnte nur selten beobachtet werden, dass ein Rückblick gemacht wird, um zu prüfen, ob die Schätzungen des Aufwands zutreffen oder der erwartete Nutzen eingetreten ist. Zudem dokumentieren Teams ihre vorherigen Schätzungen, speziell bei technischen Aufgaben, nicht. Dies nimmt den Entwicklern entscheidende Informationen, um ihre Schätzungen zu reflektieren und zu verbessern. (7.6)

Entscheiden sich die Teams dazu, dass mehr Informationen gesammelt werden sollten, werden meist Spikes, PoC oder, wie es bei Ververs empfohlen wird [129], Prototypen eingesetzt (7.4). Diese eignen sich, um die Aufwands- und Nutzenabschätzungen zu verbessern oder erst möglich zu machen, bergen jedoch die Gefahr, dass diese Umsetzung aufgrund ihrer gezeigten Funktionalität übernommen wird. In diesen Momenten würden Teams ein enormes Maß an technischen Schulden aufnehmen, da das primäre Ziel die schnelle Gewinnung von Erkenntnissen ist und zumeist grundlegende Qualitätsansprüche an produktive Software, wie Tests oder Code Struktur, dem geopfert werden. Mit entsprechender Nachbearbeitung oder einer Neuentwicklung ließen sich jedoch oft die Erkenntnisse nutzen, um die unbewusste Aufnahme von technischen Schulden zu vermeiden.

Entscheidet sich das Team jedoch gegen eine Umsetzung oder verschiebt diese auf später, nimmt es dies als bewusste Aufnahme technischer Schulden auf, sofern nicht in der Diskussion erkannt wurde, dass der Nutzen falsch eingeschätzt wurde.

Die bewusste Aufnahme von technischen Schulden erzeugte bei Entwicklern häufig Unzufriedenheit, im Besonderen falls Entwickler ein stark abweichendes Bild der „geeigneten Umsetzung" im Vergleich zu dem derzeitigen Ist-Zustand des Systems hatten. (7.5)

Fazit

In den geschilderten Zusammenhängen verbirgt sich eine bisweilen in der Forschung nicht näher betrachtete Lücke. Die Forschung beschäftigt sich zurzeit mit der Definition, Identifikation, Messbarkeit und dem Umgang mit technischen Schulden. Die Entscheidungen zum Umgang mit technischen Schulden (ggf. sogar allen technischen Aufgaben) basieren nach den eigenen Beobachtungen in großen Teilen auf den individuellen Gefühlen und Ansichten der Entwickler und sind nur selten mit Fakten untermauert. Die Argumentation über die Schwierigkeiten diese Fakten zu sammeln ist nicht unbegründet. Jedoch ist die fehlende Reflexion über die tatsächlichen Aufwände und Nutzen einer Änderung ein erhebliches Hindernis für die beteiligten Personen, ihre Fähigkeiten bezüglich der Abschätzung zu verbessern.

Die Forschung empfiehlt keine Methodik, welche die Schätzungen der beteiligten Personen verbessert. In Anbetracht der Fortschritte von intelligenten Methoden, mag es durchaus möglich sein, in Zukunft manche Arten von technischen Schulden automatisiert erkennen zu können. Jedoch könnten Methoden zur Verbesserung der Aufwands- und Nutzenabschätzungen ein Verbesserung im derzeitigen Umgang erzielen.

9 Gültigkeit

9.1 Interne Validität

Eine verbreitete Herausforderung in der Erforschung technischer Schulden ist das unterschiedliche Verständnis über den Begriff der „technischen Schulden" und die Abgrenzung zu anderen Phänomenen. Um dieser Herausforderung Herr zu werden, wurde sowohl mit den interviewten Teams als auch mit den Mitgliedern des begleiteten Teams über die Definition von technischen Schulden gesprochen. Darüber hinaus wurde nach konkreten Beispielen gefragt, um Aussagen zu stützen. Die gegebenen Beispielen wurden entsprechend auf ihre Verwendbarkeit geprüft und notfalls verworfen.

Ein weiterer Aspekt ist der Hawthorne Effekt - „Die Tendenz des Menschen, seine Leistung vorübergehend zu verbessern, wenn er weiß, dass sie untersucht wird."[28] Besonders durch den expliziten Start von Audioaufnahmen war den beteiligten Personen bewusst, in welchem Zeitraum ihr Verhalten studiert wurde. Um diesem Effekt entgegenzuwirken, wurden verschiedene Techniken verwendet:

- Gedächtnisprotokolle: Gespräche ohne Audiomitschnitt im Alltag
- Einsicht in Dokumente und Backlog von Teams
- Abgleich mit dem Verhalten vor der Beobachtungszeit (möglich durch eine bestehende Arbeitsanstellung des Autors)

Darüber hinaus können weitere Ergebnisse in den Daten zu finden sein. Die stichwortartige Transkription von Interviews bzw. Gedächtsnisprotokolle erschweren das Auffinden von nachträglich entdeckten und unerwarteten Phänomenen in alten Daten. Aus diesem Grund wurden besonders frühe Interviews voll transkribiert. Im Falle von Gedächtnisprotokollen konnte dies zwar nicht helfen, jedoch konnten Personen erneut befragt und dabei auf explizite Phänomene geachtet werden.

9.2 Externe Validität

Die Daten wurden in einem einzigen deutschen Großunternehmen erhoben. Zudem wurden die Daten in einer relativ neuen Entwicklungsorganisation gesammelt (acht Jahre), welche getrennt ist von externer Entwicklung und der Wartung bestehender Software. Darüber hinaus wird die entwickelte Software zumeist intern oder von Firmenpartnern verwendet. Es besteht somit kein direkter Kontakt zu Endkunden.

[28] https://wiki.c2.com/?HawthorneEffect

Dieses spezifische Umfeld könnte die Anwendbarkeit auf andere Umfelder stark mindern, jedoch wurden bereits in Kapitel 8.2 die eigenen Ergebnisse mit denen aus einem Startup-Umfeld verglichen, wobei ein großer Anteil der Erkenntnisse im eigenen Umfeld wiederzufinden sind. Es ist daher durchaus anzunehmen, dass die eigenen Erkenntnisse generalisiert werden können, wenn auch in anders gearteten Ausprägungen.

In den Gesprächen mit dem *Entwickler DJ* berichtete dieser von seinen Erfahrungen mit externen Entwicklungen. Er weist darauf hin, dass eine externe Entwicklung durch Verträge vollkommen andere Rahmenbedingungen und Motivationen für die Entwickler birgt. Daher ist davon auszugehen, dass die eigenen Ergebnisse im Umfeld einer externen Softwareentwicklung kaum wiederzufinden sind. Dies bedarf jedoch weiterer Forschung.

10 Zusammenfassung/Fazit/Ausblick

Technische Schulden sind ein wesentlicher Aspekt in der Softwareentwicklung und machen einen signifikanten Anteil der Entwicklungszeit aus. Derzeit wird in verschiedenste Richtungen hinsichtlich der Thematik *technischer Schulden* geforscht, jedoch werden dabei wenig Daten aus der Praxis genutzt, die das Verhalten von Softwarepraktikern im Alltag untersuchen. Diese Forschungsarbeit nutzt neben Interviews und der Sichtung von Quellcode und Protokollen ebenfalls Beobachtungen aus dem Alltag von Softwareentwicklern, um eine praxisnahe Untersuchung zu ermöglichen und somit neue Einblicke in die Thematik zu finden. Hierzu wird in einem deutschen Großunternehmen ein Entwicklerteam begleitet und weitere Entwicklerteams vereinzelt interviewt oder über einen definierten Zeitraum begleitet.

Die gesammelten Daten werden anhand der Grounded Theory Methodology ausgewertet. Aus den Beobachtungen ist eine Tendenz erkennbar, dass sowohl Entscheidungen als auch Kosten- und Nutzenabschätzungen bezüglich technischer Schulden vorwiegend auf einem Bauchgefühl basieren. Datenbasierte Entscheidungen kommen nur selten vor. Spikes bzw. Prototypen können helfen, Nutzen und Aufwand einer Umsetzung abzuschätzen. Diese können Entwicklerteams jedoch belasten, sofern sie in die produktive Codebasis übernommen werden, da sie ein hohes Maß an technischen Schulden beinhalten.

Die Ergebnisse aus anderen Umgebungen, wie Startups, waren überwiegend auf das Umfeld eines Großunternehmens anwendbar, daher wird davon ausgegangen, dass die Ergebnisse dieser Arbeit in leicht veränderter Form übertragbar sind. Diese Forschungsarbeit untersuchte interne Softwareentwicklung, externe Softwareentwicklung wurde nicht betrachtet und unterliegt grundlegend anderen Rahmenbedingungen, welche für die betrachtete Umgebung untypische Phänomene hervorrufen können. Damit stellt die externe Softwareentwicklung ein bisher unerforschtes Gebiet dar, welches mit ähnlichen Methoden wie in dieser Forschungsarbeit untersucht werden könnte und somit ergänzende Einblicke ermöglicht.

In den Gesprächen mit den Softwarepraktikern ist aufgefallen, dass diese ihr Wissen zumeist aus Blogs oder der Dokumentation von Frameworks und Bibliotheken ziehen. Alternative Quellen wie wissenschaftliche Veröffentlichungen wurden nicht benannt. Somit bietet die Thematik der Einflussnahme von verschiedenen Medien auf den Umgang mit technischen Schulden ein weiteres Forschungsfeld, welches die hier gewonnenen Daten ergänzen könnte.

Literatur

[1] it-agile. *Was sind Retrospektiven?* it-agile GmbH. 2023. URL: https://www.it-agile.de/agiles-wissen/agile-teams/was-sind-retrospektiven/ (besucht am 20. 07. 2023).

[2] Reem Alfayez u. a. „A systematic literature review of technical debt prioritization“. In: *Proceedings of the 3rd International Conference on Technical Debt*. TechDebt '20. New York, NY, USA: Association for Computing Machinery, 25. Sep. 2020, S. 1–10. ISBN: 978-1-4503-7960-1. DOI: 10.1145/3387906.3388630. URL: https://dl.acm.org/doi/10.1145/3387906.3388630 (besucht am 25. 07. 2023).

[3] Reem Alfayez u. a. „An exploratory study on the influence of developers in technical debt“. In: *Proceedings of the 2018 International Conference on Technical Debt*. ICSE '18: 40th International Conference on Software Engineering. Gothenburg Sweden: ACM, 27. Mai 2018, S. 1–10. ISBN: 978-1-4503-5713-5. DOI: 10.1145/3194164.3194165. URL: https://dl.acm.org/doi/10.1145/3194164.3194165 (besucht am 06. 08. 2023).

[4] Areti Ampatzoglou u. a. „A framework for managing interest in technical debt: an industrial validation“. In: *Proceedings of the 2018 International Conference on Technical Debt*. ICSE '18: 40th International Conference on Software Engineering. Gothenburg Sweden: ACM, 27. Mai 2018, S. 115–124. ISBN: 978-1-4503-5713-5. DOI: 10.1145/3194164.3194175. URL: https://dl.acm.org/doi/10.1145/3194164.3194175 (besucht am 05. 08. 2023).

[5] Cecilia Apa u. a. „A Taste of the Software Industry Perception of Technical Debt and its Management in Uruguay: A survey in software industry“. In: *Proceedings of the 14th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. ESEM '20: ACM / IEEE International Symposium on Empirical Software Engineering and Measurement. Bari Italy: ACM, 5. Okt. 2020, S. 1–9. ISBN: 978-1-4503-7580-1. DOI: 10.1145/3382494.3421463. URL: https://dl.acm.org/doi/10.1145/3382494.3421463 (besucht am 02. 08. 2023).

[6] Jurgen Appelo. *Management 3.0: leading Agile developers, developing Agile leaders*. The Addison-Wesley signature series. OCLC: ocn670473587. Upper Saddle River, NJ: Addison-Wesley, 2011. 413 S. ISBN: 978-0-321-71247-9.

[7] Paris Avgeriou u. a. „Managing Technical Debt in Software Engineering“. In: *Report from Dagstuhl Seminar* (Apr. 2016), S. 16162. DOI: 10.4230/DagRep.6.4.110. URL: https://resources.sei.cmu.edu/asset_files/ConferencePaper/2016_021_001_484853.pdf (besucht am 29. 07. 2020).

[8] Paris C. Avgeriou u. a. „An Overview and Comparison of Technical Debt Measurement Tools“. In: *IEEE Software* 38.3 (Mai 2021), S. 61–71. ISSN: 0740-7459, 1937-4194. DOI: 10.1109/MS.2020.3024958. URL: https://ieeexplore.ieee.org/document/9200792/ (besucht am 06. 08. 2023).

[9] Gabriele Bavota und Barbara Russo. „A large-scale empirical study on self-admitted technical debt“. In: *Proceedings of the 13th International Conference on Mining Software Repositories*. ICSE '16: 38th International Conference on Software Engineering. Austin Texas: ACM, 14. Mai 2016, S. 315–326. ISBN: 978-1-4503-4186-8. DOI: 10.1145/2901739.2901742. URL: https://dl.acm.org/doi/10.1145/2901739.2901742 (besucht am 05. 08. 2023).

[10] Kent Beck und Cynthia Andres. *Extreme programming explained: embrace change*. 2nd ed. Boston, MA: Addison-Wesley, 2005. 189 S. ISBN: 978-0-321-27865-4.

[11] Kent Beck, Ward Cunningham und Martin Fowler. *Manifest für Agile Softwareentwicklung*. 2001. URL: https://agilemanifesto.org/iso/de/manifesto.html (besucht am 23. 07. 2023).

[12] Terese Besker, Antonio Martini und Jan Bosch. „Technical Debt Triage in Backlog Management“. In: *2019 IEEE/ACM International Conference on Technical Debt (TechDebt)*. 2019 IEEE/ACM International Conference on Technical Debt (TechDebt). Montreal, QC, Canada: IEEE, Mai 2019, S. 13–22. ISBN: 978-1-72813-371-3. DOI: 10.1109/TechDebt.2019.00010. URL: https://ieeexplore.ieee.org/document/8786030/ (besucht am 27. 07. 2023).

[13] Terese Besker, Antonio Martini und Jan Bosch. „The Pricey Bill of Technical Debt: When and by Whom will it be Paid?“ In: *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 2017 IEEE International Conference on Software Maintenance and Evolution (ICSME). Shanghai: IEEE, Sep. 2017, S. 13–23. ISBN: 978-1-5386-0992-7. DOI: 10.1109/ICSME.2017.42. URL: http://ieeexplore.ieee.org/document/8094405/ (besucht am 07. 08. 2023).

[14] Terese Besker u. a. „Embracing Technical Debt, from a Startup Company Perspective“. In: *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 2018 IEEE International Conference on Software Maintenance and Evolution (ICSME). Madrid: IEEE, Sep. 2018, S. 415–425. ISBN: 978-1-5386-7870-1. DOI: 10.1109/ICSME.2018.00051. URL: https://ieeexplore.ieee.org/document/8530048/ (besucht am 08. 08. 2023).

[15] Terese Besker u. a. „The influence of Technical Debt on software developer morale“. In: *Journal of Systems and Software* 167 (Sep. 2020), S. 110586. ISSN: 01641212. DOI: 10.1016/j.jss.2020.110586. URL: https://linkinghub.elsevier.com/retrieve/pii/S0164121220300674 (besucht am 11. 08. 2023).

[16] Klara Borowa, Andrzej Zalewski und Adam Saczko. „Living With Technical Debt—A Perspective From the Video Game Industry“. In: *IEEE Software* 38.6 (Nov. 2021), S. 65–70. ISSN: 0740-7459, 1937-4194. DOI: 10.1109/MS.2021.3103249. URL: https://ieeexplore.ieee.org/document/9508260/ (besucht am 13. 08. 2023).

[17] Nanette Brown u. a. „Managing technical debt in software-reliant systems“. In: *Proceedings of the FSE/SDP workshop on Future of software engineering research*. SIGSOFT/FSE'10: 18th ACM SIGSOFT Symposium on the Foundations of Software Engineering. Santa Fe New Mexico USA: ACM, 7. Nov. 2010, S. 47–52. ISBN: 978-1-4503-0427-6. DOI: 10.1145/1882362.1882373. URL: https://dl.acm.org/doi/10.1145/1882362.1882373 (besucht am 06. 08. 2023).

[18] Bundesministerium. *AÜG - nichtamtliches Inhaltsverzeichnis*. 2011. URL: https://www.gesetze-im-internet.de/a_g/ (besucht am 23. 07. 2023).

[19] c2. *Whatever Gets Measured Gets Optimized*. 2014. URL: https://wiki.c2.com/?WhateverGetsMeasuredGetsOptimized (besucht am 09. 08. 2023).

[20] Jeromy Carriere, Rick Kazman und Ipek Ozkaya. „A cost-benefit framework for making architectural decisions in a business context“. In: *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2*. ICSE '10: 32nd International Conference on Software Engineering. Cape Town South Africa: ACM, Mai 2010, S. 149–157. ISBN: 978-1-60558-719-6. DOI: 10.1145/1810295.1810317. URL: https://dl.acm.org/doi/10.1145/1810295.1810317 (besucht am 11. 08. 2023).

[21] Orges Cico u. a. „Startups transitioning from early to growth phase -A pilot study of technical debt perception“. In: Nov. 2020.

[22] Zadia Codabux und Byron Williams. „Managing technical debt: An industrial case study“. In: *2013 4th International Workshop on Managing Technical Debt (MTD)*. 2013 4th International Workshop on Managing Technical Debt (MTD). San Francisco, CA, USA: IEEE, Mai 2013, S. 8–15. ISBN: 978-1-4673-6443-0. DOI: 10.1109/MTD.2013.6608672. URL: https://ieeexplore.ieee.org/document/6608672 (besucht am 06. 08. 2023).

[23] Zadia Codabux u. a. „An empirical assessment of technical debt practices in industry“. In: *Journal of Software: Evolution and Process* 29.10 (Okt. 2017), e1894. ISSN: 20477473. DOI: 10.1002/smr.1894. URL: https://onlinelibrary.wiley.com/doi/10.1002/smr.1894 (besucht am 07. 08. 2023).

[24] Juliet M. Corbin und Anselm L. Strauss. *Basics of qualitative research: techniques and procedures for developing grounded theory*. Fourth edition. Los Angeles: SAGE, 2015. 431 S. ISBN: 978-1-4129-9746-1.

[25] Ward Cunningham. „The WyCash portfolio management system“. In: *ACM SIGPLAN OOPS Messenger* 4 (Apr. 1993), S. 29–30. DOI: 10.1145/157710.157715.

[26] Filipe Tabosa Da Silva u. a. „Business-Driven Technical Debt Prioritization: A Replication Study“. In: *2022 17th Iberian Conference on Information Systems and Technologies (CISTI)*. 2022 17th Iberian Conference on Information Systems and Technologies (CISTI). Madrid, Spain: IEEE, 22. Juni 2022, S. 1–6. ISBN: 978-989-33-3436-2. DOI: 10.23919/CISTI54924.2022.9820390. URL: https://ieeexplore.ieee.org/document/9820390/ (besucht am 10. 08. 2023).

[27] Maya Daneva u. a. „Agile requirements prioritization in large-scale outsourced system projects: An empirical study“. In: *Journal of Systems and Software* 86.5 (Mai 2013), S. 1333–1353. ISSN: 01641212. DOI: 10.1016/j.jss.2012.12.046. URL: https://linkinghub.elsevier.com/retrieve/pii/S0164121212003536 (besucht am 06. 08. 2023).

[28] Rodrigo Reboucas De Almeida u. a. „Business-Driven Technical Debt Prioritization: An Industrial Case Study“. In: *2021 IEEE/ACM International Conference on Technical Debt (TechDebt)*. 2021 IEEE/ACM International Conference on Technical Debt (TechDebt). Madrid, Spain: IEEE, Mai 2021, S. 74–83. ISBN: 978-1-66541-405-0. DOI: 10.1109/TechDebt52882.2021.00017.

URL: https://ieeexplore.ieee.org/document/9462985/ (besucht am 06. 08. 2023).

[29] Nicolas Devos, Dimitri Durieux und Christophe Ponsard. „Managing technical debt in IT start-ups – an industrial survey“. In: Nov. 2013.

[30] Amleto Di Salle u. a. „PILOT: synergy between text processing and neural networks to detect self-admitted technical debt“. In: *Proceedings of the International Conference on Technical Debt*. TechDebt '22: International Conference on Technical Debt. Pittsburgh Pennsylvania: ACM, 16. Mai 2022, S. 41–45. ISBN: 978-1-4503-9304-1. DOI: 10.1145/3524843.3528093. URL: https://dl.acm.org/doi/10.1145/3524843.3528093 (besucht am 08. 08. 2023).

[31] Neil A. Ernst u. a. „Measure it? Manage it? Ignore it? software practitioners and technical debt“. In: *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ESEC/FSE'15: Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering. Bergamo Italy: ACM, 30. Aug. 2015, S. 50–60. ISBN: 978-1-4503-3675-8. DOI: 10.1145/2786805.2786848. URL: https://dl.acm.org/doi/10.1145/2786805.2786848 (besucht am 06. 08. 2023).

[32] eurostat. *Kleine und mittlere Unternehmen (KMU) - Strukturelle Unternehmensstatistiken - Eurostat*. 2023. URL: https://ec.europa.eu/eurostat/de/web/structural-business-statistics/information-on-data/small-and-medium-sized-enterprises (besucht am 20. 07. 2023).

[33] Martin Fowler. *Refactoring: improving the design of existing code*. Second edition. Addison-Wesley signature series. OCLC: on1064139838. Boston: Addison-Wesley, 2019. 418 S. ISBN: 978-0-13-475759-9.

[34] Martin Fowler. *TechnicalDebt*. martinfowler.com. 2019. URL: https://martinfowler.com/bliki/TechnicalDebt.html (besucht am 20. 07. 2023).

[35] Martin Fowler. *TechnicalDebtQuadrant*. Okt. 2009. URL: https://martinfowler.com/bliki/TechnicalDebtQuadrant.html.

[36] Sávio Freire u. a. „Surveying Software Practitioners on Technical Debt Payment Practices and Reasons for not Paying off Debt Items“. In: *Proceedings of the Evaluation and Assessment in Software Engineering*. EASE '20: Evaluation and Assessment in Software Engineering. Trondheim Norway: ACM, 15. Apr. 2020, S. 210–219. ISBN: 978-1-4503-7731-7. DOI: 10.1145/3383219.3383241.

URL: https://dl.acm.org/doi/10.1145/3383219.3383241 (besucht am 06. 08. 2023).

[37] Erich Gamma, Hrsg. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley professional computing series. Reading, Mass: Addison-Wesley, 1995. 395 S. ISBN: 978-0-201-63361-0.

[38] François Gauthier u. a. „Trade-offs in managing risk and technical debt in industrial research labs: an experience report“. In: *Proceedings of the 3rd International Conference on Technical Debt*. TechDebt '20: International Conference on Technical Debt. Seoul Republic of Korea: ACM, 28. Juni 2020, S. 98–102. ISBN: 978-1-4503-7960-1. DOI: 10.1145/3387906.3388623. URL: https://dl.acm.org/doi/10.1145/3387906.3388623 (besucht am 13. 08. 2023).

[39] Hadi Ghanbari u. a. „Looking for Peace of Mind? Manage Your (Technical) Debt: An Exploratory Field Study“. In: *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. 2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM). Toronto, ON: IEEE, Nov. 2017, S. 384–393. ISBN: 978-1-5090-4039-1. DOI: 10.1109/ESEM.2017.53. URL: http://ieeexplore.ieee.org/document/8170125/ (besucht am 06. 08. 2023).

[40] Barney G. Glaser. *Basics of grounded theory analysis*. Mill Valley, Calif: Sociology Press, 1992. 128 S. ISBN: 978-1-884156-00-7.

[41] Barney G. Glaser, Anselm L. Strauss und Elizabeth Strutzel. „The Discovery of Grounded Theory; Strategies for Qualitative Research:“ In: *Nursing Research* 17.4 (Juli 1968), S. 364. ISSN: 0029-6562. DOI: 10.1097/00006199-196807000-00014. URL: http://journals.lww.com/00006199-196807000-00014 (besucht am 19. 07. 2023).

[42] Giancarlo Gomes und Rafaele Matte Wojahn. „Organizational learning capability, innovation and performance: study in small and medium-sized enterprises (SMES)“. In: *Revista de Administração* 52.2 (Apr. 2017), S. 163–175. ISSN: 00802107. DOI: 10.1016/j.rausp.2016.12.003. URL: https://linkinghub.elsevier.com/retrieve/pii/S0080210716308287 (besucht am 06. 08. 2023).

[43] Yuepu Guo, Rodrigo Oliveira Spínola und Carolyn Seaman. „Exploring the costs of technical debt management – a case study“. In: *Empirical Software Engineering* 21.1 (Nov. 2014), S. 159–182. ISSN: 1382-3256, 1573-7616. DOI:

10.1007/s10664-014-9351-7. URL: http://link.springer.com/10.1007/s10664-014-9351-7 (besucht am 06. 08. 2023).

[44] Zhaoqiang Guo u. a. „How Far Have We Progressed in Identifying Self-admitted Technical Debts? A Comprehensive Empirical Study“. In: *ACM Transactions on Software Engineering and Methodology* 30.4 (31. Okt. 2021), S. 1–56. ISSN: 1049-331X, 1557-7392. DOI: 10.1145/3447247. URL: https://dl.acm.org/doi/10.1145/3447247 (besucht am 08. 08. 2023).

[45] Simon Hacks u. a. „Towards the Definition of Enterprise Architecture Debts“. In: *2019 IEEE 23rd International Enterprise Distributed Object Computing Workshop (EDOCW)*. 2019 IEEE 23rd International Enterprise Distributed Object Computing Workshop (EDOCW). Paris, France: IEEE, Okt. 2019, S. 9–16. ISBN: 978-1-72814-598-3. DOI: 10.1109/EDOCW.2019.00016. URL: https://ieeexplore.ieee.org/document/8907262/ (besucht am 01. 08. 2023).

[46] Nastaran Hajiheydari, Sarvenaz Mehravani und Manijeh Haghighinasab. „ITIL adoption model based on TAM“. In: (Jan. 2011).

[47] Kazem Haki u. a. „Digital nudging for technical debt management at Credit Suisse“. In: *European Journal of Information Systems* 32.1 (2. Jan. 2023), S. 64–80. ISSN: 0960-085X, 1476-9344. DOI: 10.1080/0960085X.2022.2088413. URL: https://www.tandfonline.com/doi/full/10.1080/0960085X.2022.2088413 (besucht am 06. 08. 2023).

[48] Red Hat. *Was ist Business Process Management?* 2023. URL: https://www.redhat.com/de/topics/automation/what-is-business-process-management (besucht am 09. 08. 2023).

[49] Copyright Haufe-Lexware GmbH \& Co KG- all rights Haufe. *BGM in Großunternehmen/Konzernen / 2.2 Qualitative Charakteristika von Konzernen | Haufe ...* Haufe.de News und Fachwissen. 2023. URL: https://www.haufe.de/personal/haufe-personal-office-platin/bgm-in-grossunternehmenkonzernen-22-qualitative-charakteristika-von-konzernen_idesk_PI42323_HI9363130.html (besucht am 20. 07. 2023).

[50] Rashina Hoda. „Socio-Technical Grounded Theory for Software Engineering“. In: *IEEE Transactions on Software Engineering* 48.10 (1. Okt. 2022), S. 3808–3832. ISSN: 0098-5589, 1939-3520, 2326-3881. DOI: 10.1109/TSE.2021.3106280. arXiv: 2103.14235[cs]. URL: http://arxiv.org/abs/2103.14235 (besucht am 19. 07. 2023).

[51] Johannes Holvitie, Ville Leppanen und Sami Hyrynsalmi. „Technical Debt and the Effect of Agile Software Development Practices on It - An Industry Practitioner Survey“. In: *2014 Sixth International Workshop on Managing Technical Debt*. 2014 6th International Workshop on Managing Technical Debt (MTD). Victoria, BC, Canada: IEEE, Sep. 2014, S. 35–42. ISBN: 978-1-4799-6791-9. DOI: 10.1109/MTD.2014.8. URL: https://ieeexplore.ieee.org/document/6974888 (besucht am 06. 08. 2023).

[52] Johannes Holvitie u. a. „Technical debt and agile software development practices and processes: An industry practitioner survey“. In: *Information and Software Technology* 96 (Apr. 2018), S. 141–160. ISSN: 09505849. DOI: 10.1016/j.infsof.2017.11.015. URL: https://linkinghub.elsevier.com/retrieve/pii/S0950584917305098 (besucht am 05. 08. 2023).

[53] Andrew Hunt und David Thomas. *The pragmatic programmer: from journeyman to master*. Reading, Mass: Addison-Wesley, 2000. 321 S. ISBN: 978-0-201-61622-4.

[54] IBM. *IBM About*. 6. Juni 2023. URL: https://www.ibm.com/de-de/about (besucht am 19. 07. 2023).

[55] Ciera Jaspan und Collin Green. „Defining, Measuring, and Managing Technical Debt“. In: *IEEE Software* 40.3 (Mai 2023), S. 15–19. ISSN: 0740-7459, 1937-4194. DOI: 10.1109/MS.2023.3242137. URL: https://ieeexplore.ieee.org/document/10109339/ (besucht am 01. 08. 2023).

[56] Ron Jeffries, Ann Anderson und Chet Hendrickson. *Extreme programming installed*. The XP series. Boston: Addison-Wesley, 2001. 265 S. ISBN: 978-0-201-70842-4.

[57] Jetbrains. *TODO comments | IntelliJ IDEA*. IntelliJ IDEA Help. 2023. URL: https://www.jetbrains.com/help/idea/using-todo.html (besucht am 20. 07. 2023).

[58] S. C. Johnson und Murray Hill. „Lint, a C program checker“. In: 1978. URL: https://api.semanticscholar.org/CorpusID:59749883.

[59] Helvio Jeronimo Junior und Guilherme Horta Travassos. „Consolidating a common perspective on Technical Debt and its Management through a Tertiary Study“. In: *Information and Software Technology* 149 (Sep. 2022), S. 106964. ISSN: 09505849. DOI: 10.1016/j.infsof.2022.106964. URL: https://linkinghub.elsevier.com/retrieve/pii/S0950584922001057 (besucht am 19. 07. 2023).

[60] Kanbanize. *Agile Estimation: Definition and Techniques*. Kanban Software for Agile Project Management. 2023. URL: https://kanbanize.com/agile/project-management/estimation (besucht am 23. 07. 2023).

[61] Ilya Khomyakov u. a. „An Analysis of Automated Technical Debt Measurement“. In: *Enterprise Information Systems*. Hrsg. von Joaquim Filipe u. a. Bd. 378. Series Title: Lecture Notes in Business Information Processing. Cham: Springer International Publishing, 2020, S. 250–273. ISBN: 978-3-030-40782-7 978-3-030-40783-4. DOI: 10.1007/978-3-030-40783-4_12. URL: http://link.springer.com/10.1007/978-3-030-40783-4_12 (besucht am 05. 08. 2023).

[62] Tim Klinger u. a. „An enterprise perspective on technical debt“. In: *Proceeding of the 2nd working on Managing technical debt - MTD '11* (2011). DOI: 10.1145/1985362.1985371.

[63] Eriks Klotins u. a. „Exploration of technical debt in start-ups“. In: *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice*. ICSE '18: 40th International Conference on Software Engineering. Gothenburg Sweden: ACM, 27. Mai 2018, S. 75–84. ISBN: 978-1-4503-5659-6. DOI: 10.1145/3183519.3183539. URL: https://dl.acm.org/doi/10.1145/3183519.3183539 (besucht am 08. 08. 2023).

[64] Philippe Kruchten. *Concrete things you can do about your technical debt*. Philippe Kruchten. 14. Feb. 2017. URL: https://philippe.kruchten.com/2017/02/14/concrete-things-you-can-do-about-your-technical-debt/ (besucht am 13. 08. 2023).

[65] Philippe Kruchten. *Managing technical debt: reducing friction in software development*. 1st edition. Hoboken, NJ: Pearson Education, Inc, 2019. ISBN: 978-0-13-564593-2.

[66] Philippe Kruchten, Robert L. Nord und Ipek Ozkaya. „Technical Debt: From Metaphor to Theory and Practice“. In: *IEEE Software* 29 (Nov. 2012), S. 18–21. DOI: 10.1109/ms.2012.167.

[67] Philippe Kruchten u. a. „Technical debt: Towards a Crisper Definition“. In: *ACM SIGSOFT Software Engineering Notes* 38 (Aug. 2013), S. 51. DOI: 10.1145/2507288.2507326.

[68] Justin Kruger und David Dunning. „Unskilled and unaware of it: How difficulties in recognizing one's own incompetence lead to inflated self-assessments.“ In: *Journal of Personality and Social Psychology* 77.6 (1999), S. 1121–1134. ISSN: 1939-1315, 0022-3514. DOI: 10.1037/0022-3514.77.6.1121. URL: http://doi.apa.org/getdoi.cfm?doi=10.1037/0022-3514.77.6.1121 (besucht am 06. 08. 2023).

[69] *Längsschnittdaten - Statista Definition*. Statista Lexikon. URL: https://de.statista.com/statistik/lexikon/definition/81/laengsschnittdaten/ (besucht am 27. 08. 2023).

[70] M.M. Lehman. „Programs, life cycles, and laws of software evolution“. In: *Proceedings of the IEEE* 68.9 (1980), S. 1060–1076. ISSN: 0018-9219. DOI: 10.1109/PROC.1980.11805. URL: http://ieeexplore.ieee.org/document/1456074/ (besucht am 19. 07. 2023).

[71] Valentina Lenarduzzi u. a. „A systematic literature review on Technical Debt prioritization: Strategies, processes, factors, and tools“. In: *Journal of Systems and Software* 171 (Jan. 2021), S. 110827. DOI: 10.1016/j.jss.2020.110827.

[72] Jean-Louis Letouzey. „Managing Large Application Portfolio with Technical Debt Related Measures“. In: *2016 Joint Conference of the International Workshop on Software Measurement and the International Conference on Software Process and Product Measurement (IWSM-MENSURA)*. 2016 Joint Conference of the International Workshop on Software Measurement and the International Conference on Software Process and Product Measurement (IWSM-MENSURA). Berlin, Germany: IEEE, Okt. 2016, S. 181–181. ISBN: 978-1-5090-4147-3. DOI: 10.1109/IWSM-Mensura.2016.035. URL: http://ieeexplore.ieee.org/document/7809607/ (besucht am 05. 08. 2023).

[73] Jean-Louis Letouzey. *Project Management and Technical Debt | Agile Alliance*. Agile Alliance |. 28. Feb. 2016. URL: https://www.agilealliance.org/project-management-and-technical-debt/ (besucht am 19. 07. 2023).

[74] Jean-Louis Letouzey u. a. „Introduction to the Technical Debt Concept“. In: ().

[75] Yikun Li u. a. „Self-Admitted Technical Debt in the Embedded Systems Industry: An Exploratory Case Study“. In: *IEEE Transactions on Software Engineering* 49.4 (1. Apr. 2023), S. 2545–2565. ISSN: 0098-5589, 1939-3520, 2326-3881. DOI: 10.1109/TSE.2022.3224378. URL: https://ieeexplore.ieee.org/document/9961946/ (besucht am 08. 08. 2023).

[76] Zengyang Li, Paris Avgeriou und Peng Liang. „A systematic mapping study on technical debt and its management“. In: *Journal of Systems and Software* 101 (März 2015), S. 193–220. ISSN: 01641212. DOI: 10.1016/j.jss.2014.12.027. URL: https://linkinghub.elsevier.com/retrieve/pii/S0164121214002854 (besucht am 03. 08. 2023).

[77] lucidchart. *7 Arten von Organisationsstrukturen*. Lucidchart. 17. Jan. 2017. URL: https://www.lucidchart.com/blog/types-of-organizational-structures (besucht am 20. 07. 2023).

[78] Robert C. Martin. *The clean coder: A code of conduct for professional programmers*. 1. Aufl. USA: Prentice Hall Press, 2011. ISBN: 0-13-708107-3.

[79] Antonio Martini, Terese Besker und Jan Bosch. „Technical Debt tracking: Current state of practice“. In: *Science of Computer Programming* 163 (Okt. 2018), S. 42–61. ISSN: 01676423. DOI: 10.1016/j.scico.2018.03.007. URL: https://linkinghub.elsevier.com/retrieve/pii/S0167642318301035 (besucht am 05. 08. 2023).

[80] Antonio Martini, Terese Besker und Jan Bosch. „The Introduction of Technical Debt Tracking in Large Companies“. In: *2016 23rd Asia-Pacific Software Engineering Conference (APSEC)*. 2016 23rd Asia-Pacific Software Engineering Conference (APSEC). Hamilton, New Zealand: IEEE, 2016, S. 161–168. ISBN: 978-1-5090-5575-3. DOI: 10.1109/APSEC.2016.032. URL: http://ieeexplore.ieee.org/document/7890584/ (besucht am 05. 08. 2023).

[81] Antonio Martini und Jan Bosch. „A Multiple Case Study of Continuous Architecting in Large Agile Companies: Current Gaps and the CAFFEA Framework“. In: *2016 13th Working IEEE/IFIP Conference on Software Architecture (WICSA)*. 2016 13th Working IEEE/IFIP Conference on Software Architecture (WICSA). Venice, Italy: IEEE, Apr. 2016, S. 1–10. ISBN: 978-1-5090-2131-4. DOI: 10.1109/WICSA.2016.31. URL: http://ieeexplore.ieee.org/document/7516806/ (besucht am 05. 08. 2023).

[82] Antonio Martini, Jan Bosch und Michel Chaudron. „Architecture Technical Debt: Understanding Causes and a Qualitative Model“. In: *2014 40th EUROMICRO Conference on Software Engineering and Advanced Applications*. 2014 40th EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA). Verona, Italy: IEEE, Aug. 2014, S. 85–92. ISBN: 978-1-4799-5795-8. DOI: 10.1109/SEAA.2014.65. URL: http://ieeexplore.ieee.org/document/6928795/ (besucht am 19. 07. 2023).

[83] Antonio Martini u. a. „Identifying and Prioritizing Architectural Debt Through Architectural Smells: A Case Study in a Large Software Company“. In: *Software Architecture*. Hrsg. von Carlos E. Cuesta, David Garlan und Jennifer Pérez. Bd. 11048. Series Title: Lecture Notes in Computer Science. Cham: Springer International Publishing, 2018, S. 320–335. ISBN: 978-3-030-00760-7 978-3-030-00761-4. DOI: 10.1007/978-3-030-00761-4_21. URL: http://link.springer.com/10.1007/978-3-030-00761-4_21 (besucht am 05. 08. 2023).

[84] Steve McConnell. „Managing Technical Debt“. Diss. Juni 2008. URL: https://www.construx.com/uploadedfiles/resources/whitepapers/Managing%20Technical%20Debt.pdf (besucht am 28. 06. 2023).

[85] Steve McConnell. *Technical Debt - 10x Software Development*. 19. Feb. 2008. URL: https://web.archive.org/web/20080219020452/http://blogs.construx.com/blogs/stevemcc/archive/2007/11/01/technical-debt-2.aspx (besucht am 06. 08. 2023).

[86] Leonardo Mendes, Cristina Cerdeiral und Gleison Santos. „Documentation Technical Debt: A Qualitative Study in a Software Development Organization“. In: *Proceedings of the XXXIII Brazilian Symposium on Software Engineering*. SBES 2019: XXXIII Brazilian Symposium on Software Engineering. Salvador Brazil: ACM, 23. Sep. 2019, S. 447–451. ISBN: 978-1-4503-7651-8. DOI: 10.1145/3350768.3350773. URL: https://dl.acm.org/doi/10.1145/3350768.3350773 (besucht am 07. 09. 2023).

[87] microservices.io. *Microservices Pattern: Database per service*. microservices.io. 2018. URL: http://microservices.io/patterns/data/database-per-service.html (besucht am 20. 07. 2023).

[88] microservices.io. *Microservices Pattern: Shared database*. microservices.io. 2019. URL: http://microservices.io/patterns/data/shared-database.html (besucht am 20. 07. 2023).

[89] Ben D. Monaghan und Julian M. Bass. „Redefining Legacy: A Technical Debt Perspective“. In: *Product-Focused Software Process Improvement*. Hrsg. von Maurizio Morisio, Marco Torchiano und Andreas Jedlitschka. Bd. 12562. Series Title: Lecture Notes in Computer Science. Cham: Springer International Publishing, 2020, S. 254–269. ISBN: 978-3-030-64147-4 978-3-030-64148-1. DOI: 10.1007/978-3-030-64148-1_16. URL: https://link.springer.com/10.1007/978-3-030-64148-1_16 (besucht am 08. 08. 2023).

[90] J. David Morgenthaler u. a. „Searching for build debt: Experiences managing technical debt at Google“. In: *2012 Third International Workshop on Managing Technical Debt (MTD)*. 2012 Third International Workshop on Managing Technical Debt (MTD). Zurich, Switzerland: IEEE, Juni 2012, S. 1–6. ISBN: 978-1-4673-1749-8 978-1-4673-1748-1. DOI: 10.1109/MTD.2012.6225994. URL: https://ieeexplore.ieee.org/document/6225994 (besucht am 06. 08. 2023).

[91] María Isabel Murillo und Marcelo Jenkins. „Technical Debt Measurement during Software Development using Sonarqube: Literature Review and a Case Study“. In: *2021 IEEE V Jornadas Costarricenses de Investigación en Computación e Informática (JoCICI)*. 2021 IEEE V Jornadas Costarricenses de Investigación en Computación e Informática (JoCICI). San José, Costa Rica: IEEE, 25. Okt. 2021, S. 1–6. ISBN: 978-1-66549-832-6. DOI: 10.1109/JoCICI54528.2021.9794341. URL: https://ieeexplore.ieee.org/document/9794341/ (besucht am 05. 08. 2023).

[92] Sam Newman. *Building microservices: designing fine-grained systems*. Second Edition. OCLC: on1263799754. Sebastopol, CA: O'Reilly Media, 2021. 586 S. ISBN: 978-1-4920-3402-5.

[93] Mille Edith Nielsen, Christian Østergaard Madsen und Mircea Filip Lungu. „Technical Debt Management: A Systematic Literature Review and Research Agenda for Digital Government“. In: *Lecture Notes in Computer Science* (2020), S. 121–137. DOI: 10.1007/978-3-030-57599-1_10. (Besucht am 13. 03. 2023).

[94] Ariadi Nugroho, Joost Visser und Tobias Kuipers. „An empirical model of technical debt and interest“. In: *Proceedings of the 2nd Workshop on Managing Technical Debt*. ICSE11: International Conference on Software Engineering. Waikiki, Honolulu HI USA: ACM, 23. Mai 2011, S. 1–8. ISBN: 978-1-4503-0586-0. DOI: 10.1145/1985362.1985364. URL: https://dl.acm.org/doi/10.1145/1985362.1985364 (besucht am 11. 08. 2023).

[95] Karl Nunkoosing. „The Problems With Interviews“. In: *Qualitative Health Research* 15.5 (Mai 2005), S. 698–706. ISSN: 1049-7323, 1552-7557. DOI: 10.1177/1049732304273903. URL: http://journals.sagepub.com/doi/10.1177/1049732304273903 (besucht am 19. 07. 2023).

[96] David OBrien u. a. „23 shades of self-admitted technical debt: an empirical study on machine learning software“. In: *Proceedings of the 30th ACM Joint*

European Software Engineering Conference and Symposium on the Foundations of Software Engineering. ESEC/FSE '22: 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering. Singapore Singapore: ACM, 7. Nov. 2022, S. 734–746. ISBN: 978-1-4503-9413-0. DOI: 10.1145/3540250.3549088. URL: https://dl.acm.org/doi/10.1145/3540250.3549088 (besucht am 08. 08. 2023).

[97] Ipek Ozkaya und Robert Nord. „10 Years of Research in Technical Debt and an Agenda for the Future". URL: https://apps.dtic.mil/sti/pdfs/AD1183607.pdf.

[98] Alexia Pacheco, Gabriela Marín-Raventós und Gustavo López. „Technical Debt in Costa Rica: An InsighTD Survey Replication". In: *Product-Focused Software Process Improvement*. Hrsg. von Xavier Franch, Tomi Männistö und Silverio Martínez-Fernández. Bd. 11915. Series Title: Lecture Notes in Computer Science. Cham: Springer International Publishing, 2019, S. 236–243. ISBN: 978-3-030-35332-2 978-3-030-35333-9. DOI: 10.1007/978-3-030-35333-9_17. URL: http://link.springer.com/10.1007/978-3-030-35333-9_17 (besucht am 10. 08. 2023).

[99] Boris Pérez u. a. „What are the practices used by software practitioners on technical debt payment: results from an international family of surveys". In: *Proceedings of the 3rd International Conference on Technical Debt*. TechDebt '20: International Conference on Technical Debt. Seoul Republic of Korea: ACM, 28. Juni 2020, S. 103–112. ISBN: 978-1-4503-7960-1. DOI: 10.1145/3387906.3388632. URL: https://dl.acm.org/doi/10.1145/3387906.3388632 (besucht am 03. 08. 2023).

[100] Rolf-Helge Pfeiffer. „Searching for Technical Debt - An Empirical, Exploratory, and Descriptive Case Study". In: *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER). Honolulu, HI, USA: IEEE, März 2022, S. 1018–1022. ISBN: 978-1-66543-786-8. DOI: 10.1109/SANER53432.2022.00119. URL: https://ieeexplore.ieee.org/document/9825694/ (besucht am 11. 08. 2023).

[101] Diogo Pina, Carolyn Seaman und Alfredo Goldman. „Technical debt prioritization: a developer's perspective". In: *Proceedings of the International Conference on Technical Debt*. TechDebt '22: International Conference on Technical Debt. Pittsburgh Pennsylvania: ACM, 16. Mai 2022, S. 46–55.

ISBN: 978-1-4503-9304-1. DOI: 10.1145/3524843.3528096. URL: https://dl.acm.org/doi/10.1145/3524843.3528096 (besucht am 09. 08. 2023).

[102] Pivotal. *Adding stories*. 2023. URL: https://www.pivotaltracker.com/help/articles/adding_stories/ (besucht am 20. 07. 2023).

[103] Aniket Potdar und Emad Shihab. „An Exploratory Study on Self-Admitted Technical Debt“. In: *2014 IEEE International Conference on Software Maintenance and Evolution*. 2014 IEEE International Conference on Software Maintenance and Evolution (ICSME). Victoria, BC, Canada: IEEE, Sep. 2014, S. 91–100. ISBN: 978-1-4799-6146-7. DOI: 10.1109/ICSME.2014.31. URL: http://ieeexplore.ieee.org/document/6976075/ (besucht am 05. 08. 2023).

[104] *Project Info – InsighTD Project*. URL: https://web.archive.org/web/20220627062046/http://www.td-survey.com/project-info/ (besucht am 08. 08. 2023).

[105] Qonto. *Was ist ein Start-up-Unternehmen? Einfach erklärt!* Qonto - Glossar. Section: FinTech. 12. Dez. 2022. URL: https://qonto.com/de/glossar/startup (besucht am 08. 08. 2023).

[106] Narayan Ramasubbu und Chris F. Kemerer. „Technical Debt and the Reliability of Enterprise Software Systems: A Competing Risks Analysis“. In: *Management Science* 62.5 (Mai 2016), S. 1487–1510. ISSN: 0025-1909, 1526-5501. DOI: 10.1287/mnsc.2015.2196. URL: https://pubsonline.informs.org/doi/10.1287/mnsc.2015.2196 (besucht am 06. 08. 2023).

[107] Rodrigo Reboucas De Almeida, Christoph Treude und Uira Kulesza. „Tracy: A Business-Driven Technical Debt Prioritization Framework“. In: *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 2019 IEEE International Conference on Software Maintenance and Evolution (ICSME). Cleveland, OH, USA: IEEE, Sep. 2019, S. 181–185. ISBN: 978-1-72813-094-1. DOI: 10.1109/ICSME.2019.00028. URL: https://ieeexplore.ieee.org/document/8919237/ (besucht am 09. 08. 2023).

[108] Rodrigo Reboucas De Almeida u. a. „Aligning Technical Debt Prioritization with Business Objectives: A Multiple-Case Study“. In: *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 2018 IEEE International Conference on Software Maintenance and Evolution (ICSME). Madrid: IEEE, Sep. 2018, S. 655–664. ISBN: 978-1-5386-7870-1. DOI:

10.1109/ICSME.2018.00075. URL: https://ieeexplore.ieee.org/document/8530078/ (besucht am 09. 08. 2023).

[109] RedHat. *Was ist Open Source? Definition, Funktionsweise und Vorteile*. 2023. URL: https://www.redhat.com/de/topics/open-source/what-is-open-source (besucht am 06. 09. 2023).

[110] Xiaoxue Ren u. a. „Neural Network-based Detection of Self-Admitted Technical Debt: From Performance to Explainability". In: *ACM Transactions on Software Engineering and Methodology* 28.3 (31. Juli 2019), S. 1–45. ISSN: 1049-331X, 1557-7392. DOI: 10.1145/3324916. URL: https://dl.acm.org/doi/10.1145/3324916 (besucht am 08. 08. 2023).

[111] Academic Research. *What are the advantages and challenges of using interviews in qualitative research?* URL: https://www.linkedin.com/advice/0/what-advantages-challenges-using-interviews-qualitative (besucht am 19. 07. 2023).

[112] Nicolli Rios, Manoel Gomes De Mendonça Neto und Rodrigo Oliveira Spínola. „A tertiary study on technical debt: Types, management strategies, research trends, and base information for practitioners". In: *Information and Software Technology* 102 (Okt. 2018), S. 117–145. ISSN: 09505849. DOI: 10.1016/j.infsof.2018.05.010. URL: https://linkinghub.elsevier.com/retrieve/pii/S0950584918300946 (besucht am 20. 07. 2023).

[113] Nicolli Rios u. a. „The practitioners' point of view on the concept of technical debt and its causes and consequences: a design for a global family of industrial surveys and its first results from Brazil". In: *Empirical Software Engineering* 25.5 (Sep. 2020), S. 3216–3287. ISSN: 1382-3256, 1573-7616. DOI: 10.1007/s10664-020-09832-9. URL: https://link.springer.com/10.1007/s10664-020-09832-9 (besucht am 20. 07. 2023).

[114] Bayerischer Rundfunk. „Edward Lorenz : Als der Schmetterlingseffekt ein Chaos anrichtete". In: (25. Juli 2022). URL: https://www.ardalpha.de/wissen/geschichte/historische-persoenlichkeiten/edward-lorenz-meteorologe-schmetterlingseffekt-chaostheorie-chaosforschung-100.html (besucht am 20. 07. 2023).

[115] Irene Sala, Antonela Tommasel und Francesca Arcelli Fontana. „DebtHunter: A Machine Learning-based Approach for Detecting Self-Admitted Technical Debt". In: *Evaluation and Assessment in Software Engineering*. EASE 2021: Evaluation and Assessment in Software Engineering. Trondheim Norway:

ACM, 21. Juni 2021, S. 278–283. ISBN: 978-1-4503-9053-8. DOI: 10.1145/3463274.3464455. URL: https://dl.acm.org/doi/10.1145/3463274.3464455 (besucht am 08. 08. 2023).

[116] Miltiadis Siavvas u. a. „Technical debt as an indicator of software security risk: a machine learning approach for software development enterprises“. In: *Enterprise Information Systems* 16.5 (4. Mai 2022), S. 1824017. ISSN: 1751-7575, 1751-7583. DOI: 10.1080/17517575.2020.1824017. URL: https://www.tandfonline.com/doi/full/10.1080/17517575.2020.1824017 (besucht am 06. 08. 2023).

[117] Victor Machado Silva, Helvio Jeronimo Junior und Guilherme Horta Travassos. „A Taste of the Software Industry Perception of Technical Debt and its Management in Brazil“. In: *Journal of Software Engineering Research and Development* 7 (15. Juli 2019), S. 1. ISSN: 2195-1721. DOI: 10.5753/jserd.2019.19. URL: https://sol.sbc.org.br/journals/index.php/jserd/article/view/19 (besucht am 06. 08. 2023).

[118] Saulo Soares De Toledo u. a. „Architectural Technical Debt in Microservices: A Case Study in a Large Company“. In: *2019 IEEE/ACM International Conference on Technical Debt (TechDebt)*. 2019 IEEE/ACM International Conference on Technical Debt (TechDebt). Montreal, QC, Canada: IEEE, Mai 2019, S. 78–87. ISBN: 978-1-72813-371-3. DOI: 10.1109/TechDebt.2019.00026. URL: https://ieeexplore.ieee.org/document/8786035/ (besucht am 06. 08. 2023).

[119] Marilyn Strathern. „‘Improving ratings’: audit in the British University system“. In: *European Review* 5.3 (Juli 1997), S. 305–321. ISSN: 10627987, 1234981X. DOI: 10.1002/(SICI)1234-981X(199707)5:3<305::AID-EURO184>3.0.CO;2-4. URL: https://www.cambridge.org/core/product/identifier/S1062798700002660/type/journal_article (besucht am 06. 08. 2023).

[120] Anselm L. Strauss, Juliet M. Corbin und Anselm L. Strauss. *Grounded theory: Grundlagen qualitativer Sozialforschung*. Unveränd. Nachdr. der letzten Aufl. Weinheim: Beltz, 2010. 227 S. ISBN: 978-3-621-27265-0.

[121] Richard H. Thaler u. a. *Nudge: wie man kluge Entscheidungen anstößt*. 3. Aufl. Berlin: Econ, 2009. 389 S. ISBN: 978-3-430-20081-3.

[122] Edith Tom, Aybüke Aurum und Richard Vidgen. „An exploration of technical debt“. In: *Journal of Systems and Software* 86 (Juni 2013), S. 1498–1516. DOI: 10.1016/j.jss.2012.12.052.

[123] Adam Tornhill. *Prioritizing Technical Debt as if Time and Money Matters*. Nov. 2019. URL: https://gotocph.com/2019/sessions/974/prioritizing-technical-debt-as-if-time-and-money-matters (besucht am 28. 06. 2023).

[124] Omer Uludag u. a. „Investigating the Adoption and Application of Large-Scale Scrum at a German Automobile Manufacturer“. In: *2019 ACM/IEEE 14th International Conference on Global Software Engineering (ICGSE)*. 2019 ACM/IEEE 14th International Conference on Global Software Engineering (ICGSE). Montreal, QC, Canada: IEEE, Mai 2019, S. 22–29. ISBN: 978-1-5386-9196-0. DOI: 10.1109/ICGSE.2019.00019. URL: https://ieeexplore.ieee.org/document/8807470/ (besucht am 23. 07. 2023).

[125] Cathy Urquhart. *Grounded theory for qualitative research: a practical guide*. OCLC: ocn823891293. Los Angeles, Calif. ; London: SAGE, 2013. 205 S. ISBN: 978-1-84787-053-7 978-1-84787-054-4.

[126] Muhammad Usman u. a. „Effort estimation in agile software development: a systematic literature review“. In: *Proceedings of the 10th International Conference on Predictive Models in Software Engineering*. PROMISE '14: The 10th International Conference on Predictive Models in Software Engineering. Turin Italy: ACM, 17. Sep. 2014, S. 82–91. ISBN: 978-1-4503-2898-2. DOI: 10.1145/2639490.2639503. URL: https://dl.acm.org/doi/10.1145/2639490.2639503 (besucht am 14. 08. 2023).

[127] Verbi. *Which media file formats can I analyze with MAXQDA?* MAXQDA. 2023. URL: https://www.maxqda.com/faq/media-files (besucht am 22. 07. 2023).

[128] Roberto Verdecchia, Philippe Kruchten und Patricia Lago. „Architectural Technical Debt: A Grounded Theory“. In: *Software Architecture*. Hrsg. von Anton Jansen u. a. Bd. 12292. Series Title: Lecture Notes in Computer Science. Cham: Springer International Publishing, 2020, S. 202–219. ISBN: 978-3-030-58922-6 978-3-030-58923-3. DOI: 10.1007/978-3-030-58923-3_14. URL: http://link.springer.com/10.1007/978-3-030-58923-3_14 (besucht am 06. 08. 2023).

[129] Linus Ververs. „Umgang mit technischen Schulden im Startup-Umfeld“. Diss. Apr. 2022. URL: https://www.inf.fu-berlin.de/inst/ag-se/theses/Ververs21-Umgang-techn-Schulden-Startups.pdf.

[130] Hans van Vliet. *Software engineering: principles and practice*. 2nd ed. Chichester [England] ; New York: John Wiley, 2000. 726 S. ISBN: 978-0-471-97508-3.

[131] Hulya Vural, Murat Koyuncu und Sinem Guney. „A Systematic Literature Review on Microservices“. In: *Computational Science and Its Applications – ICCSA 2017*. Hrsg. von Osvaldo Gervasi u. a. Bd. 10409. Series Title: Lecture Notes in Computer Science. Cham: Springer International Publishing, 2017, S. 203–217. ISBN: 978-3-319-62406-8 978-3-319-62407-5. DOI: 10.1007/978-3-319-62407-5_14. URL: http://link.springer.com/10.1007/978-3-319-62407-5_14 (besucht am 06. 08. 2023).

[132] Xin Wang u. a. „Detecting and explaining self-admitted technical debts with attention-based neural networks“. In: *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. ASE '20: 35th IEEE/ACM International Conference on Automated Software Engineering. Virtual Event Australia: ACM, 21. Dez. 2020, S. 871–882. ISBN: 978-1-4503-6768-4. DOI: 10.1145/3324884.3416583. URL: https://dl.acm.org/doi/10.1145/3324884.3416583 (besucht am 08. 08. 2023).

[133] Ward Cunningham. *Debt Metaphor*. 15. Feb. 2009. URL: https://www.youtube.com/watch?v=pqeJFYwnkjE (besucht am 19. 07. 2023).

[134] Gerald M Weinberg. *Quality software management*. New York: Dorset House Pub, 1997.

[135] Andrew Wilkinson. *Don't Call It Tech Debt · Andrew Wilkinson*. Feb. 2022. URL: https://www.theandrewwilkinson.com/2022/02/03/dont-call-it-tech-debt/ (besucht am 28. 06. 2023).

[136] Jesse Yli-Huumo, Andrey Maglyas und Kari Smolander. „How do software development teams manage technical debt? – An empirical study“. In: *Journal of Systems and Software* 120 (Okt. 2016), S. 195–218. ISSN: 01641212. DOI: 10.1016/j.jss.2016.05.018. URL: https://linkinghub.elsevier.com/retrieve/pii/S016412121630053X (besucht am 07. 09. 2023).

Sebstständigkeitserklärung

Name:	Maic Siemering
Matrikel-Nr.:	2529980
Fach:	Masterstudiengang Informatik
Modul:	Masterarbeit
Thema:	*Umgang mit technischen Schulden in Großunternehmen*

Ich erkläre, dass ich die Masterarbeit selbstständig und ohne unzulässige Inanspruchnahme Dritter verfasst habe. Ich habe dabei nur die angegebenen Quellen und Hilfsmittel verwendet und die aus diesen wörtlich, inhaltlich oder sinngemäß entnommenen Stellen als solche kenntlich gemacht. Die Versicherung selbstständiger Arbeit gilt auch für enthaltene Zeichnungen, Skizzen oder graphische Darstellungen. Die Arbeit wurde bisher in gleicher oder ähnlicher Form weder derselben noch einer anderen Prüfungsbehörde vorgelegt und auch noch nicht veröffentlicht. Mit der Abgabe der elektronischen Fassung der endgültigen Version der Arbeit nehme ich zur Kenntnis, dass diese mit Hilfe eines Plagiatserkennungsdienstes auf enthaltene Plagiate überprüft und ausschließlich für Prüfungszwecke gespeichert wird.

Hagen, den (06.09.2023)

Maic Siemering