Umgang mit technischen Schulden im Startup-Umfeld

Masterarbeit im Themenfeld der angewandten Informatik
mit Schwerpunkt Software Engineering

an der
Freien Universität Berlin

mit Unterstützung
des Hasso-Plattner-Instituts

Linus Philipp Ververs
Matrikelnummer: 5039477
ververs198@inf.fu-berlin.de

Erstgutachter: Prof. Dr. Lutz Prechelt
Zweitgutachter: Prof. Dr. Jürgen Döllner
Betreuer: Victor Brekenfeld, Daniel Atzberger

26. Juli 2021

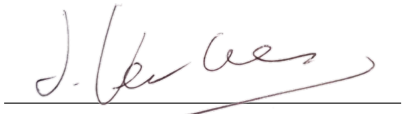
Zusammenfassung:

Diese Arbeit untersucht den Umgang dreier langlebiger Startup-Unternehmen mittels Interviews und in Anlehnung an die Grounded Theory mit dem Ziel mehr Erkenntnisse zum generellen Umgang mit technischen Schulden zu sammeln und vor allem Phänomene herauszustellen, die womöglich zum Erfolg der Unternehmen beitrugen. Dabei wurde insbesondere die Zufriedenheit der Entwickler, welche u.a. unter zu hohen technischen Schulden leidet, als potentiell kritischer Erfolgsfaktor für Software-Startups identifiziert. Das Verfolgen der Strategie der Vermeidung technischer Schulden im Alltag trägt im Zuge dessen dazu bei, die Zufriedenheit und Motivation der Entwickler nicht zu gefährden und wurde in allen untersuchten Unternehmen beobachtet.

Eidesstattliche Erklärung

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

Berlin, den 26. Juli 2021



Linus Philipp Ververs

Inhaltsverzeichnis

Tabellenverzeichnis

1 Einleitung

Der Umgang mit technischen Schulden ist immer an das Spannungsfeld zwischen kurzfristig gesparten Zeitaufwänden und einer langfristig behinderten Entwicklung, welche wiederum erhöhte Entwicklungsaufwände nach sich zieht, gekoppelt. Insbesondere im Startup-Umfeld, wo in der Regel unter hohem Zeitdruck gearbeitet wird, kommt dieses Spannungsfeld besonders stark zum tragen. So listet Tauber gutes „Engineering“ als einen der 5 wesentlichen Säulen eines Technologie-Startups und schreibt:

> „All products are expected to have some code that is outdated. However, technical debt is a trade-off between doing things the right way and taking a temporary shortcut — keyword temporary. If companies don't give time for engineers to clean up after themselves then they will eventually hit a wall.“[46]

Es gibt zwar keine gesicherten Erkenntnisse bei wie vielen Startups ein fehlerhafter Umgang mit technischen Schulden zum Misserfolg der Unternehmung führte, allerdings kann das Missmanagement von technischen Schulden die Entwicklungsaufwände so stark potenzieren, dass vor allem Unternehmen, die unter Zeitdruck operieren und auf begrenztes Fremdkapitel angewiesen sind, daran zugrunde gehen können.

Da es wenig gesicherte Erkenntnisse zum richtigen Management von technischen Schulden in Startups gibt, hat die hier durchgeführte Forschung das Ziel, Phänomene rund um den Umgang mit technischen Schulden in Software-Startups zu beobachten und zu vergleichen. Alle untersuchten Unternehmen sind langlebige und erfolgreiche Software-Startups, weshalb die Forschung mit der Hoffnung verbunden ist, Hinweise und Empfehlungen für das Management von technischen Schulden in anderen Startups abzuleiten. Im Sinne der Grounded Theory startet diese Forschung ohne feste Hypothesen, die es zu validieren gilt. Diese werden erst im Verlauf der Befragung und Auswertung aufgestellt.

Diese Arbeit beginnt in Kapitel 2 mit einem Umriss des wissenschaftlichen Hintergrunds von technischen Schulden gefolgt von Kapitel 3 mit einer kurzen Erläuterung zum Startup-Begriff. In Kapitel 4 werden verwandte Arbeiten vorgestellt und dabei insbesondere auf Forschungsarbeiten eingegangen, die auch zu technischen Schulden im Startup-Umfeld Forschung betrieben haben. Die Grounded Theory, auf der die Methodik dieser Arbeit fußt, wird in Kapitel 5 vorgestellt. Kapitel 6 gibt einen Überblick über die erhobenen Daten, die die Grundlage für die in Kapitel 7 beschriebenen beobachteten Phänomen bilden. In Kapitel 8 werden diese anschließend diskutiert und abstrahiert und Empfehlungen für Startups abgeleitet. Zum Ende wird die Validität der Forschung in Kapitel 9 bewertet und die Arbeit kurz in Kapitel 10 zusammengefasst.

2 Technische Schulden

2.1 Technische Schulden als Metapher

Der Begriff der technischen Schulden oder im Englischen „Technical Debt“ wurde zum ersten Mal von Ward Cunningham im Jahr 1992 verwendet[13]. Nach Kruchten et al. und Li et al. fand diese Metapher erst mit Beginn des neuen Jahrtausends breitere Verwendung in der Praxis und zog damit ein gesteigertes Interesse der Forschungsgemeinde ca. ab 2010 nach sich[36][39]. Bei der Terminologie handelt es sich um eine Metapher, die vor allem Entscheidungsträgern ohne tief gehende Kenntnisse der Informatik mithilfe des aus dem Finanzwesen entlehnten Begriffs der Schulden die Konsequenzen von technisch nicht einwandfreiem Code zu verbildlichen[35][3]. Gemeint ist damit Code, der zwar funktioniert, dessen Implementierung allerdings beispielsweise gegen Programmier- oder Architekturstandards verstößt[14]. Diese Verstöße führen dazu, dass Entwickler, die in Zukunft an dem Code arbeiten, sehr wahrscheinlich mehr Zeit als während der Arbeit an sauberem Code aufwenden müssen. Cunningham selbst beschreibt sein Konzepte mit folgenden Worten:

> „A little debt speeds development so long as it is paid back promptly with a rewrite. [...] The danger occurs when the debt is not repaid. Every minute spent on not-quite-right code counts as interest on that debt.“[13]

Hier werden die Überschneidungen zu den klassischen Geldschulden deutlich. Wie bei einem Kredit werden regelmäßig Zinsen fällig, immer dann, wenn an dem Code gearbeitet wird oder jemand mit dem System ineffizient interagiert und wie bei einem Kredit kann die Aufnahme von technischen Schulden dazu verwendet werden, einem Projekt mit begrenzten Ressourcen eine Anschubfinanzierung im übertragenen Sinne zu geben. Im Gegensatz zum Finanzwesen gibt es aber einen weniger direkten Zwang die technischen Schulden zurückzuzahlen, obwohl die Folgen von zu großen akkumulierten technischen Schulden ähnlich fatal sein können[2]. Zudem müssen diese selten vollständig zurückgezahlt werden, da ein geringes Maß an technischen Schulden in der Regel geringe oder gar keine negative Auswirkungen mit sich bringen.

Die Metapher der technischen Schulden wurde in den vergangenen Jahrzehnten durch die Theorien und Beobachtungen anderer angereichert. So unterteilt Martin Fowler in einem Blogeintrag technische Schulden weiter in vier Quadranten entlang der Dimensionen: rücksichtslos/umsichtig und überlegt/unüberlegt[19]. Laut dieser Theorie können technische Schulden bewusst bzw. überlegt aufgenommen werden, zum Beispiel um eine Deadline einzuhalten oder eben unbeabsichtigt bzw. unüberlegt in der Regel beispielsweise wenn die bestmögliche Herangehensweise erst zur Mitte eines Projekts klar und erst daraufhin angewandt wird. Während sich diese Dimension klar daran messen lässt, ob zu irgendeinem Zeitpunkt vor der Entstehung der technischen Schulden, also vor der Implementierung, ein Entwickler, ein Team oder ein Vorgesetzter bewusst die Entscheidung getroffen hat, diese Abkürzung zu nehmen oder ob diese Entscheidung nie getroffen wurde, fällt diese klare Einteilung bei der zweiten Dimension und damit der Unterscheidung zwischen rücksichtslos

oder umsichtig aufgenommenen technischen Schulden schwerer, da es sich um eine subjektivere und graduellere Einteilung handelt. Fowler führt als Beispiel für die rücksichtslose Aufnahme von technischen Schulden Zitate an, die nahelegen, dass den Verantwortlichen die Konsequenzen der Schulden egal wären, wohingegen eine umsichtige Aufnahme sich der Gefahren von zu hohen technischen Schulden bewusst wäre und wohl auch in der Zukunft mit dem Abbau dieser plant.

2.2 Technische Schulden definieren

Über die Jahre hinweg erfreut sich die Metapher der technischen Schulden großer Beliebtheit und wurde zunehmend in weiteren Kontexten rund um das Thema der Codequalität und -komplexität verwendet. Kruchten et al. kritisieren, dass die Metapher der technischen Schulden durch diese sehr breite Verwendung des Begriffs an Schärfe verloren hätte[35]. Auch Stochel et al. stellen in ihrer Metastudie zur Verwendung des Begriffs in der Forschung fest, dass es oft an einer klaren Definition mangelt, was sie auf den metaphorischen Ursprung der Terminologie zurückführen[44]. Dieser Umstand macht es um so wichtiger, klar zu definieren, was genau in welchem Kontext mit dem Begriff gemeint ist.
Nach der sehr breit gefassten Definition von Holvitie et al. sind technische Schulden als die Konsequenzen von bewussten oder unbewussten Entscheidungen während der Softwareentwicklung zu verstehen, bei denen das Implementieren von Funktionalität oder Projekt(-management-)anforderungen wie Liefertermine oder Budgetbeschränkungen priorisiert werden[29]. Diese Definition trifft zwar den Kern der Metapher, im Rahmen dieser und vermutlich auch anderer wissenschaftlicher Arbeiten ist allerdings eine Definition von Nöten, die technische Schulden stärker von System- und Programmierfehlern abgrenzt. Im Weiteren wird sich der Definition, welche im Rahmen des Dagstuhl Seminars 16162 „Managing Technical Debt in Software Engineering“ erarbeitet wurde, bedient:

> „In software-intensive systems, technical debt is a collection of design or implementation constructs that are expedient in the short term, but set up a technical context that can make future changes more costly or impossible. Technical debt presents an actual or contingent liability whose impact is limited to internal system qualities, primarily maintainability and evolvability.“[3]

Diese Definition macht deutlich, dass technische Schulden bewusst oder unbewusst aufgenommen werden können, allerdings nie durch pure Idiotie entstehen können, da die Aufnahme immer an einen Sinn gekoppelt ist. Man könnte also argumentieren, dass diese Definition nicht den rücksichtslos-unüberlegt Quadranten nach der Einteilung von Fowler umfasst. Des Weiteren hat technische Schuld hier nur direkte negative Auswirkungen auf interne Codequalität und kann damit zwar die grundlegende Ursache für Systemfehler sein, wird mit diesen allerdings nicht gleichgesetzt. Nichtsdestotrotz ist auch diese Definition nicht perfekt. Im Zentrum der Metapher nach Cunningham steht nämlich das Zahlen von Zinsen - also höheren Entwicklungsaufwänden - aufgrund von technischen Schulden. Dies verwässert die Definition allerdings durch den Halbsatz: „[...] but set up a technical context that

can make future changes more costly or impossible.“ Um die höheren Entwicklungsaufwände stärker in dieser Definition zu verankern, wird darauf plädiert das „can“ durch ein „are likely to“ zu ersetzen. Es ließe sich auch der Einsatz des noch strengeren „will“ verargumentieren, allerdings wird damit die Kategorisierung von neu entstandenen technischen Schulden erschwert, da man zum aktuellen Zeitpunkt evtl. noch gar nicht abschätzen kann, ob Zinszahlungen in der Zukunft garantiert fällig werden.
Entsprechend dieser Definition werden an dieser Stelle beispielsweise folgende Artefakte während der Softwareentwicklung als technische Schulden verstanden ohne Anspruch auf Vollständigkeit zu erheben[1]:

- Verstöße gegen fundamentale Design-Prinzipien (Englisch: „Code Smells“)
- Verstöße gegen Programmierkonventionen (z.B. Namenskonventionen)
- Hohe (vermeidbare) Code-Komplexität
- Fehlende, unzureichende oder fehlerhafte Dokumentation
- Fehlende, unzureichende oder fehlerhafte Tests
- Nicht-Einhalten der festgelegten Architektur- und Designkonventionen (Englisch: „design / architecure debt“[48])
- Suboptimale Architektur- oder Designkonventionen

Anzumerken ist hierbei, dass entgegen anderer Autoren unzureichende Anforderungsanalysen und die dazugehörige Dokumentation („requierments debt“) im Rahmen dieser Arbeit und der zugrunde gelegten Definition nicht unter den Begriff der technischen Schulden gefasst werden. Wie bereits oben angeführt haben Definitionen, die weiter gefasst sind und damit u.a. auch unzureichende Anforderungsanalysen miteinbeziehen, Abgrenzungsprobleme zum Beispiel gegenüber Softwarefehlern oder auch anderen Designdokumenten. Nichtsdestotrotz handelt es sich bei dem sogenannten „requierments debt“ um eine betrachtenswerte Dimension, die im Rahmen dieser Arbeit zwar nicht unter dem Oberbegriff der technischen Schulden subsumiert wird, allerdings auch nicht einfach aus der in Kapitel 5 beschriebenen Forschung und den durchgeführten Interviews ausgeklammert wird.
In Ergänzung betonen Kruchten et al., dass technische Schulden nicht nur mit qualitativ schlechtem Code oder anderen Artefakten der Softwareentwicklung gleichzusetzen sind. Eine Software, die beispielsweise nur eine Sprache in der Benutzeroberfläche unterstützt und damit die aktuellen Marktanforderungen vollständig bedient, kann auf einer einwandfreien Codebasis fußen, sobald man sich aber dazu entscheidet, in andere Märkte zu expandieren, in denen weitere Sprachen unterstützt werden müssen, kann sich die damals gewählte Architektur als suboptimal herausstellen und die Einführung zusätzlicher Übersetzungen Mehraufwände nach sich ziehen, die hätten vermieden werden können.[36]

[1] Diese Artefakte basieren auf der Auflistung von Kruchten et al.[35]. Da dort allerdings eine weiter gefasste Definition von technischen Schulden zugrunde gelegt wird, werden hier einige Artefakte nicht aufgeführt oder wurden enger gefasst.

Bereits an den zwei hier aufgeführten Definitionen und deren Implikationen, was genau unter den Begriff der technischen Schulden fällt, wird deutlich, dass es in der Forschungsgemeinde an einer einheitlichen Definition und der damit einhergehenden Abrenzung von technischen Schulden bisher fehlt. So schreiben Li et al. u.a. als Ergebnisse ihrer Metastudie:

> „The TD community has spent a lot of effort to investigate different types of TD, but spent little effort in distinguishing between TD and non-TD.“[39]

2.3 Entstehung technischer Schulden

Wie Cunningham schon bei der erstmaligen Einführung der Metapher anführt, ist das Aufnehmen von technischen Schulden kaum zu vermeiden. So schreibt er:

> „Shipping first time code is like going into debt.“[13]

Bereits 1980 also vor Entstehung des hier diskutierten Konzepts der technischen Verschuldung formulierte Lehmann als sein 2. Gesetz zur Software Evolution:

> „As an evolving program is continually changed, its complexity, reflecting deteriorating structure, increases unless work is done to maintain or reduce it.“[37]

Während hier nicht explizit von technischen Schulden gesprochen wird, so wird Code-Komplexität heutzutage als ein eine mögliche Ausprägungsform von technischen Schulden verstanden[2][9][14].
Wie bereits in 2.1 aufgeführt, ist bei der Entstehung von technischen Schulden klar zwischen der bewussten Entscheidung eine Abkürzung zu nehmen und der unbewussten Wahl der nicht optimalen Herangehensweise zu differenzieren. Die oberhalb angeführten Zitate beziehen sich dabei in erster Linie auf unbewusst aufgenommene technische Schulden während der Implementierung oder Überarbeitungen eines Abschnittes im Code. Hier ist allerdings anzuführen, dass diese unbewusst aufgenommenen technischen Schulden zu bewussten aufgenommenen technischen Schulden werden, wenn man sich aktiv, trotz der zu erwartenden, erhöhten, zukünftigen Entwicklungsaufwände, gegen die Überarbeitung des Codes entscheidet. Darüber hinaus können technische Schulden allerdings auch erst im Verlauf der Zeit ungeplant entstehen[9] beispielsweise durch Abhängigkeiten zu veralteten Bibliotheken oder anderer Dritt-Software[4].
Dem gegenüber stehen dann genau die Situationen während der Softwareentwicklung, bei denen man sich ganz bewusst für die Aufnahme von technischen Schulden entscheidet, um zum Beispiel ein Lieferdatum einzuhalten. So beschreiben Giardino et al. wie Startup-Unternehmen gezielt technische Schulden in Form von qualitativ weniger hochwertigem Code oder fehlender Testautomatisierung aufnehmen, um so den Entwicklungsprozess zu beschleunigen und schneller zu einem ersten marktfähigen Produkt zu gelangen[22].

2.4 Umgang mit technischen Schulden

Wie bereits oben bei der Einführung der Metapher angeführt, kann die Aufnahme technischer Schulden zwar einen kurzfristig wünschenswerten Effekte auf die Softwareentwicklung haben, langfristig jedoch führt die Anhäufung von zu vielen technischen Schulden zu schwerwiegenden Problemen, was Zazworka et al. in ihrer Untersuchung zu den Auswirkungen von „design debt“[48] wie folgt beschreiben:

> „Growing debt affects software development negatively by making the code less maintainable, less correct, more difficult to extend, less portable, and more complicated to understand.“

Cunningham greift in Bezug auf die Gefahren von technischen Schulden sogar zu noch drastischeren Worten:

> „Entire engineering organizations can be brought to a stand-still under the debt load of an unconsolidated implementation [...].“[13]

In Bezug darauf ist es unerlässlich, für eine professionelle Softwareentwicklung die Aufnahme von technischen Schulden zu kontrollieren und deren Abbau zu planen. So schlagen Guo et al. vor, technische Schulden, nachdem diese identifiziert wurden, durchgängig zu überwachen und konstant zwischen den zukünftigen Kosten, die diese jeweils geschätzt verursachen werden und den geschätzten Kosten die Schulden abzubezahlen, abzuwägen[27].
Li et al. listen in ihrer Metastudie, welche Publikationen bis 2013 berücksichtigt, folgende in wissenschaftlichen Arbeiten vorgeschlagene Maßnahmen zum Umgang mit technischen Schulden vor[39]:

- Identifizierung und
- Quantifizierung technischer Schulden
- Priorisierung beim Abbau
- Prävention der Anhäufung
- Überwachen der Kosten und Nutzen bestehender technischer Schulden

Insgesamt entstand bei der im Rahmen dieser Arbeit durchgeführten Literaturrecherche der Eindruck, dass es wenig wirklich konkrete und belastbare Empfehlungen in publizierten Forschungsarbeiten zum genauen Umgang mit technischen Schulden jenseits des Offensichtlichen gibt. Nach Avgeriou et al. gibt es auch in der Praxis wenig etablierte Vorgehensweisen:

> „Today many organizations do not have established practices to manage technical debt [...].“[3]

Auch Lenarduzzi et al. kommen in ihrer Meta-Studie zum Umgang mit technischen Schulden, welche auf Arbeiten basiert, die bis zum Dezember 2019 veröffentlicht wurden, zu dem Schluss, dass es keinen klaren Konsens in der Forschung zum Umgang mit technischen Schulden gibt. Sie schreiben:

> „[...] there is no unified approach regarding how the TD prioritization process should becarried out, nor is there a consensus on which aspects to focus on when performing the TD prioritization process.“[38]

2.5 Automatische Identifizierung von technischen Schulden

Aufgrund der großen Popularität der Metapher der technischen Schulden in Wissenschaft und Praxis existiert mittlerweile eine große Zahl an Forschungsarbeiten, aber auch Werkzeugen, die versuchen oder damit werben, technische Schulden innerhalb einer Codebasis zu erkennen oder sogar die damit verbundenen Kosten zu berechnen. Die Vergleiche der Funktionsweise der Berechnung von technischen Schulden in bekannten Werkzeugen von Fontana et al.[18] und Avgeriou[4] machen deutlich, dass große Unterschiede darin herrschen, welche Arten von technischen Schulden überhaupt in die Berechnung mit einfließen, was letztere Autoren in ihrem Fazit wie folgt beschreiben:

> „[...] there is no commonly-agreed and validated set of rules and metrics to measure Technical Debt. Instead, each tool uses its own set of rules and metrics without detailed explanation or motivation."

Sie betonen zudem, dass sich die Werkzeuge auf die einfach zu detektierenden Ausprägungsformen von technischen Schulden konzentrieren, während Schulden, die beispielsweise aus Verstößen gegen die definierten Architekturprinzipien entstehen, vernachlässigt werden. Vor der Gefahr, die damit verbunden ist, dass lediglich das Detektierbare unter dem Begriff der technischen Schulden subsumiert wird, während alles nicht automatisch klassifizierbare ignoriert wird, warnen auch Kruchten et al.:

> „Furthermore, once we identify tools such as static code analyzers to assist us in identifying technical debt, there's a danger of equating it with whatever our tools can detect. This approach leads to leaving aside large amounts of potential technical debt that's undetectable by tools, such as structural or architectural debt or technological gaps."[35]

So betonen auch Curtis et al. in ihrer Arbeit zur automatisierten Berechnung von technischen Schulden, dass die Kosten zur Abzahlung der technischen Schulden durch die verwendeten Analytics-Methoden lediglich auf Grundlage der detektierbaren Mängel einer Software geschätzt werden können[14].

Zudem schreiben Avgeriou et al., dass eine Reihe von Arbeiten existieren, die die Codequalität u.a. durch das Identifizieren von „Code Smells" bemessen und damit auf das Level an technischen Schulden schließen, sie stellen aber fest, dass die technischen Schulden mit besonders hoher Tragweite im Design der Software lägen und folglich nicht berücksichtigt würden:

> „A number of studies have examined the relationship between software code quality and technical debt. This work has applied detection of "code smells" (low internal code quality), coupling and cohesion, and dependency analysis to identify technical debt. However, empirical examples collected from industry all point out that the most significant technical debt is caused by design trade-offs, which are not detectable by measuring code quality."[3]

Im Gegensatz zu den bis jetzt diskutierten Arbeiten nähern sich Zazworka et al. der Problematik in ihrer Fallstudie nicht nur von der Seite der automatischen Klassifizierung von technischen Schulden, sondern lassen unabhängig davon auch das Entwicklerteam Artefakte mit technischen Schulden identifizieren[49]. Sie verwenden dabei Werkzeuge, um „code smells“ zu identifizieren und wenden zudem statische Codemetriken zur Bestimmung der Code-Komplexität an mit dem Ergebnis, dass die verwendeten Werkzeuge und Metriken zwar gut sogenannte „defect debt“ erkennen konnten, welche im Rahmen dieser Arbeit nicht unter die zugrunde gelegte Definition der technischen Schulden fallen, und bei allen anderen Formen der technischen Schulden deutlich hinter den manuellen Einschätzungen des Entwicklerteams zurückbleiben.

Khomyakov et al. resümieren in ihrer 2020 publizierten Meta-Studie über die in der Forschung verwendeten Methoden zur automatisierten Bestimmung technischer Schulden, dass es sich um ein sehr junges Forschungsfeld handele, in dem die Forschungsergebnisse fast ausschließlich nur durch die Hand der beteiligten Autoren evaluiert werden und wenig davon replizierbar wäre[32]. Folglich beschrieben neue Arbeiten in der Regel immer neue Ansätze, bauten nicht auf anderen Forschungsarbeiten aus dem Feld auf und erschweren es damit erheblich geeignete Werkzeuge für den Gebrauch in der Praxis zu identifizieren.

Zusammenfassend und mit dem Stand von Januar 2021 steckt die automatisierte Berechnung von technischen Schulden noch in den Kinderschuhen und es entsteht z.T. der Eindruck der Begriff der technische Schulden findet seinen Weg manchmal nur aufgrund seiner Popularität in solche Forschungsarbeiten. Es ist zudem durchaus diskutabel, dass das automatische Identifizieren von technischen Schulden so lange ein Irrweg bleiben wird, wie man versucht exakte und unumstößliche Ergebnisse zu produzieren. Da bei der Metapher der technischen Schulden die zu zahlenden Zinsen in Form von Mehraufwänden bei zukünftiger Entwicklungsarbeit im Vordergrund stehen, lassen sich diese nie nur auf Grundlage der Codebasis bestimmen, sondern sind immer im Kontext der gesamten Organisation zu betrachten. Daran scheitern die aktuellen Verfahren, weshalb das exakte Identifizieren von technischen Schulden nicht der Anspruch sein sollte.

3 Der Startup Begriff und Motivation dieser Arbeit

Nach Prof. Dr. Dr. Ann-Kristin Achleitner versteht man ein Startup-Unternehmen als:

> „junge, noch nicht etablierte Unternehmen, die zur Verwirklichung einer innovativen Geschäftsidee [...] mit geringem Startkapital gegründet werden und i.d.R. sehr früh zur Ausweitung ihrer Geschäfte und Stärkung ihrer Kapitalbasis entweder auf den Erhalt von Venture-Capital bzw. Seed Capital (evtl. auch durch Business Angels) angewiesen sind.“[1]

Ergänzend definieren Wang et al. Software-Startups als Unternehmen, die (hauptsächlich) auf Software gestützte Produkte innerhalb begrenzter Zeiträume und mit wenigen Ressourcen entwickeln und versuchen nachhaltige und skalierbare Geschäftsmodelle aufzubauen[47]. Situation in Software-Startups ist oftmals extrem dynamisch, unvorhersehbar und sogar chaotisch[21]. Die dort durchgeführte Softwareentwicklung ist aufgrund der fehlenden Prozesse einer neugebildeten Organisation[31] sowie der üblicherweise herrschenden Ressourcenknappheit[12] und der Arbeit mit modernsten Technologien erschwert. Giardino et al. empfehlen das schnelle Entwickeln von vielen Prototypen, eine agile und für Veränderungen offene Entwicklung, das Designen einer entsprechend einfach anpassbaren Software und das regelmäßige Refactorn für die Softwareentwicklung in Startups[23].

Der Umgang mit technischen Schulden ist immer durch die Abwägung zwischen kurzfristigen Vorteilen gegenüber langfristigen Nachteilen charakterisiert. Der betriebswirtschaftliche Druck, unter dem Software-Startups stehen, macht diese Abwägung umso spannender. Zum aktuellen Zeitpunkt gibt es verhältnismäßig wenig Forschung und Erkenntnisse zum richtigen Umgang mit technischen Schulden in Software-Startups. Im Rahmen dieser Arbeit werden langlebige Software-Startups befragt, um im Sinne einer qualitativen, explorativen Forschung erste Erkenntnisse zum erfolgreichen Management von technischen Schulden im besonders umkämpften Markt der Software-Startups zu gewinnen.

4 Verwandte Arbeiten

Im Folgenden werden verwandte wissenschaftliche Arbeiten aufgeführt. Zuerst werden dabei vier Arbeiten, die auch technische Schulden im Startup-Umfeld untersuchen, im Detail und in chronologischer Ordnung nach Erscheinungsdatum vorgestellt. Des Weiteren wird auf weitere wichtige Literatur zur Thematik verwiesen.

4.1 Forschung zu technischen Schulden in Startups

Managing technical debt in IT start-ups –an industrial survey [16]
Die Arbeit von Devos et al. beschreibt die frühste Forschung zu technischen Schulden in Startups, die im Rahmen der hier durchgeführten Literaturrecherche gefunden wurden. Die Autoren untersuchen mittels Codeanalyse und Befragungen 14 Startups und versuchen so Rückschlüsse auf unterschiedliche Dimensionen des Entwicklungsprozesses zu ziehen. Ergebnis der Analyse ist eine Tabelle der Startups mit prozentualen Bewertungen in den Kategorien (Code-)Qualität, Effizienz und Wartbarkeit sowie Reifegrad der Prozesse und Lösung. Den ersten 3 Dimensionen sind zwar interessante Definitionen zugrunde gelegt, allerdings ist vollkommen unklar wie diese mittels Codeanalyse genau berechnet wurden und zumindest zweifelhaft, ob überhaupt eine akkurate Erfassung möglich ist, wenn jedes Startup - wie die Autoren beschreiben - nur zu einem einzelnen Zeitpunkt untersucht wurde. So soll der Effizienz-Wert erfassen, wie viele technische Schulden in Relation zum gesamten Produktionsaufwand aufgenommen wurden. Wie bereits in 2.5 diskutiert wurde, ist

die automatische Erfassung von technischen Schulden mittels statischer Codeanalyse bis heute umstritten. Die Effizienz der Softwareentwicklung auf Grundlage dessen als einzelner Prozentwert kondensiert, der zudem den Vergleich zu anderen Startups ermöglichen soll, ist mit großer Vorsicht zu genießen. Auch die Berechnung der Dimensionen Qualität und Wartbarkeit ist nicht ganz klar ist, weshalb eine umfassende Beurteilung der Validität nicht möglich ist, obwohl das Erfassen dieser mittels statischer Codeanalyse als kredibler einzuschätzen ist. Die beiden letzteren Dimensionen wurden im Gegensatz dazu durch eine Befragung der für den Entwicklungsprozess verantwortlichen Person zur Prozessqualität auf Basis des ISO29110 ermittelt. Auch hier wäre eine transparentere Darstellung der verwendeten Fragen und erhaltenen Antworten wünschenswert gewesen, anstelle des schlichten, abgeleiteten Prozentwertes.

Der Vergleich der unterschiedlichen Werte in diesen Dimensionen über die untersuchten Startups hinweg liefert kaum Mehrwert oder Erkenntnisse. Zusammenfassend hat diese Arbeit - anders als es ihr Titel vermuten lassen würde - wenig Überschneidungen mit der hier beschriebenen Forschung.

Exploration of Technical Debt in Start-Ups [34]
In ihrem 2018 erschienenen Artikel untersuchen Klotins et al. mittels einer auf einem Fragebogen[2] basierenden Umfrage unter 86 Startups das Verhältnis dieser zu technischen Schulden. Befragt wurden dabei zum Großteil aktive Unternehmen aus dem europäischen und lateinamerikanischen Raum, die zwischen einem und fünf Jahren alt waren. Die Befragten[3] sollten in der Umfrage u.a. schätzen, wie viel technische Schulden sie in unterschiedlichen Bereichen aufgenommen hatten, welche Gründe sie dafür verantwortlich machten und wie technische Schulden innerhalb des Startups geschätzt wurden. Darüber hinaus wurden Fragen zur Charakterisierung des Startups, der beteiligten Entwickler und deren Einstellung zu bestimmten Aspekten der Softwareentwicklung (z.B. Einhaltung von Programmierkonventionen, Spannungsfeld zwischen Funktionalität und interner Qualität) gestellt.
Klotins et al. stellen dabei folgende Ergebnisse ihrer Arbeit heraus:

- Befragte aus aktiv operierenden Startups schätzen ihre technischen Schulden in allen Bereichen geringer ein als Befragte aus geschlossenen und verkauften Startups.
- „Testing debt“ ist der Bereich dem Startups den höchsten Anteil an technischen Schulden zuschreiben, was entsprechend der Umfrage auf fehlende Testautomatisierung zurückzuführen ist. So seien manuelle, explorative Softwaretests der Standard bei den befragten Startups und das unabhängig von deren Alter, Erfahrung, Teamgröße und Phase.
- Die Teamgröße und das Niveau der technischen Fähigkeiten im Unternehmen haben einen großen Einfluss darauf, wie schwerwiegend die anderen Ur-

[2]Der vollständige Fragebogen ist online einsehbar unter: https://eriksklotins.lv/uploads/TD-in-start-ups-questions.pdf (aufgerufen am 02.12.2021)
[3]Hauptsächlich wurden die Fragebögen durch die Gründer, aber auch Entwicklern innerhalb des Unternehmens, beantwortet.

sachen für technische Schulden (Einstellungen, Sachlichkeit, Kommunikation, verfügbare Ressourcen, Prozesse) bewertet wurden. Startups mit größeren Teams (≥ 9), haben laut der Umfrage mehr technische Schulden.

- Technische Schulden in Form von nicht sauber geschriebenem Code („code debt“) beeinflussen die Qualität des Produktes und die Produktivität der Entwicklung negativer als die anderen Ausprägungsformen. Laut der Umfrage haben technische Schulden innerhalb der Dokumentation nur Einfluss auf die Produktivität, wohingegen fehlende Testautomatisierung im Durchschnitt keine negativen Effekte zugeschrieben wurden.

Darüber hinaus merken sie an, dass der Grad der Auswirkungen von technischen Schulden basierend auf der Phase, in der sich das Startup befindet, variieren kann. So werden die Folgen von technischen Schulden besonders sichtbar, wenn das Produkt beispielsweise an eine große Zahl von Kunden ausgeliefert, an ein anderes Team zur Weiterentwicklung übergeben wird oder das Team sich signifikant vergrößert.

Natürlich kann ein Fragebogen nie die Wirklichkeit sondern nur die Wahrnehmung der Befragten von der Wirklichkeit erfassen. Besonders da es sich hier um ein sehr junges Forschungsfeld handelt, wäre der Einsatz von Methoden, die stärker auf die Exploration ausgerichtet sind, zu favorisieren. Hinzu kommt, dass besonders in Bezug auf das Thema der technischen Schulden - wie in Abschnitt 2.2 diskutiert - unterschiedlichste Definitionen und Abgrenzungen existieren. Hier birgt die Befragung per Fragebogen ganz besonders die Gefahr, dass die Fragen unterschiedlich interpretiert und folglich auch beantwortet werden. Klotins et al. umschiffen dieses Risiko, in dem sie nie explizit nach einer Einschätzung der Teilnehmer zu technischen Schulden fragen, sondern lediglich Aussagen zu der Softwarequalität, dem Entwicklungsprozess und der Firmenkultur durch die Befragten bewerten (wahre Aussage / falsche Aussage) lassen[4]. Insgesamt liefert diese Forschung zwar wichtige erste Hinweise durch die erfassten Korrelationen, man muss allerdings vorsichtig sein, daraus direkt Kausalzusammenhänge zu folgern. Es handelt sich hierbei um eine gute Grundlage, auf der weitere Forschung aufgebaut werden sollte.

Embracing Technical Debt, from a Startup Company Perspective [7]
Besker et al. nähern sich der Problematik in ihrem 2018 erschienenen Artikel durch 16 Interviews mit Angehörigen von 7 unterschiedlichen Startups. Ziel der Forschung war es zum einen organisatorischen Faktoren innerhalb der Unternehmen herauszufinden, die einen Einfluss auf die Akkumulation von technischen Schulden hatten, und zum anderen die Herausforderungen im Umgang mit, aber auch die Vorteile von technischen Schulden innerhalb der Startups zu ergründen.
Die Interviewten hatten dabei zum Großteil die Rolle eines Entwicklers inne, darüber hinaus wurden Gründer, Architekten und weitere Entscheidungsträger befragt. Vor den ein- bis zweistündigen Interviews mit den Teilnehmern führten die Autoren einen Workshop durch, in dem zuerst der Begriff der technischen Schulden von den

[4]Solche Aussagen sind zum Beispiel: „Test cases do not fully cover the product/service functionality“; „Product quality (Performance, scalability, maintainability, security, robustness etc.) suffers due toissues in the code“

Autoren eingeführt und anschließend darüber in der Gruppe diskutiert wurde, um eine gemeinsame Wissensgrundlage und Verständnis zu schaffen.
In ihrer Forschung identifizieren sie 6 unterschiedliche, organisatorische Faktoren, die einen Einfluss auf die Aufnahme technischer Schulden haben können: Erfahrung der Entwickler, technisches Wissen der Gründer, Größe des Entwicklerteams im Laufe der Zeit, Unsicherheit[5], fehlende Entwicklungsprozesse, Grad der Autonomie der Entwickler. Hervorzuheben ist dabei die Erkenntnis der Autoren, dass Startups mit technisch unerfahrenen Gründern dazu neigen höheren Wert auf technisch sauberen Code zu legen, während Gründer mit einem tieferen Verständnis für Softwareentwicklungsprozesse eher dazu tendieren, bewusst technische Schulden aufzunehmen, da sie Vorteile mit den Nachteilen abwiegen können. Daneben ist auch das Ergebnis überraschend, dass ein Ansteigen der Teamgröße zu einem Abbau von technischen Schulden führe, da die mit der Codebasis vertrauten Entwickler diese zum Erleichtern des Einstieges „aufräumten“ und neue Entwickler dazu neigten schlecht verständlichen oder lesbaren Code umzuschreiben.
In Bezug auf ihre 2. Fragestellung führen die Autoren zahlreiche Vorteile von technischen Schulden, wie zum Beispiel: verkürzte Entwicklungszeiten, geringerer Kapitaleinsatz oder größere Flexibilität, die vom Code unabhängige Entscheidungen ermöglicht. Dem entgegen stehen natürlich die Nachteile, die mit der Aufnahme von technischen Schulden verbunden sind. Dazu zählen die Autoren u.a.: mögliche Fehler in der Software, eingeschränkte Skalierbarkeit, Erhöhung des Zeit- und Ressourceneinsatzes oder höhere zukünftige Risiken für das Unternehmen. Subsumierend kommen Besker et al. zu dem Ergebnis, dass es für ein Startup wichtig ist eine gute Balance zwischen den Vor- und Nachteilen aufrecht zu erhalten. So macht es für ein Startup zu Beginn viel Sinn bewusst technische Schulden aufzunehmen. Mit der Weiterentwicklung und Reifung des Produktes werden technische Schulden allerdings auch zu einem immer größeren Risikofaktor für die Entwicklung.

Ausgehend von ihren Forschungsergebnissen formulieren die Autoren folgende Empfehlungen für Startups:

- Eine Mischung aus Senior und Junior Entwicklern führt zu einer gesunden Balance zwischen technischen Schulden und interner Qualität.
- Technisch unerfahrene Gründer brauchen einen technisch erfahrenen und unabhängigen Berater.
- Teile der technischen Schulden können Entwickler dazu verleiten ihren eigenen Programmierstil daran zu orientieren, führen deshalb zu weiteren technischen Schulden und sollten deshalb bei einem Abbau priorisiert werden.
- Fördere die Autonomie der Entwickler aber sorge für abstrahierte Richtlinien für den Umgang mit technischen Schulden.

Die Ergebnisse von Besker et al. sind von hoher Relevanz für die Praxis und weitere Forschung. Obwohl die Interviewpartner aus Startups mit diversen Charakteristika (Teamgröße, Bestehen, Domäne, Phase, etc.) stammen, ist die externe Validität

[5]Die Autoren meinen hiermit das Fehlen von realistischen Zukunftsprognosen, was zum Beispiel typisch für Startups ohne gesicherte Finanzierung ist.

aufgrund der geringen Teilnehmerzahl eher gering zu bewerten. Auf der anderen Seiten ist die interne Validität durch die gewählte Methodik und die ausführlichen Interviews hoch. Als Kritikpunkt ist hier lediglich anzuführen, dass jeweils nur ein Interview pro Person stattgefunden hat und folglich die Ergebnisse sich nur auf die Wahrnehmung der Befragten zu einem festen Zeitpunkt beziehen. Eine Befragung über einen längeren Zeitpunkt birgt das Potential die Erkenntnisse zu vertiefen.

Startups transitioning from early to growth phase - A pilot study of technical debt perception [10]
In dem im November 2020 erschienenen Artikel untersuchen Cico et al. den Umgang mit technischen Schulden in wachsenden Startups. Dafür wurden 8 CTOs oder CEOs von 7 Startups, die sich laut Angaben der Autoren gerade in der Wachstumsphase oder kurz davor befanden, interviewt. Alle Interviewten hatten dabei eine aktive Rolle bezogen auf die Softwareentwicklung inne. Die Transkripte der Interviews wurden mithilfe von thematischer Codierung analysiert.
Ähnlich wie in den zuvor vorgestellten Forschungen kommen sie dabei zu dem Schluss, dass die Aufnahme von technischen Schulden zu Beginn einer Startup-Unternehmung von Vorteil sein kann. Sie unterschieden dabei allerdings zwischen dem Ignorieren von technischen Schulden und dem Akzeptieren der Aufnahme. Letzteres beschreibt eine bewusste Aufnahme von technischen Schulden beispielsweise in Form mehrerer Prototypen, um Anforderungen zu validieren. Der Abbau von technischen Schulden spielt noch keine Rolle und wird nicht weiter geplant. Im Gegensatz dazu ist das Ignorieren von technischen Schulden eher mit einer unbewussten Aufnahme dieser zu vergleichen, was in erster Linie auf die Unerfahrenheit der Entwickler oder fehlender Expertise im Umgang mit technischen Schulden zurückzuführen ist. Mit dem Wachstum eines Software-Startups rücken technische Schulden mehr in den Fokus, da diese einen kritischen Faktor in Bezug auf den weiteren Unternehmenserfolg darstellen. Cico et al. unterschieden auch hier zwischen zwei möglichen Ausprägungsformen. Das Management von technischen Schulden („managing TD“) auf der einen Seite und der Vermeidung dieser („avoiding TD“) auf der anderen Seite. Beide Herangehensweisen verbindet der gezielte und umsichtige Umgang mit der Thematik. „Managing TD“ legt dabei den Fokus auf den bewussten Umgang mit technischen Schulden und ermöglicht damit weiterhin die zusätzliche Aufnahme, dies allerdings stets verbunden mit einem Ziel und dem konkreten Plan diese wieder abzubauen. Im Gegensatz dazu führt die Strategie der Vermeidung technischer Schulden bewusst Maßnahmen ein, um gar keine oder so wenig wie möglich aufzunehmen. Der Unterschied dieser beiden Herangehensweisen wird an dem Beispiel eines drohenden Abgabedatums oder Meilensteinziels deutlich. Während „managing TD“ zur bewussten Aufnahme von technischen Schulden tendieren würde, um das Lieferdatum einzuhalten, würde die „avoiding TD“ Herangehensweise wohl eher versuchen, den Termin nach hinten zu verschieben oder den Umfang der zu liefernden Funktionalität zu beschneiden.

Neben dieser Unterscheidung der Herangehensweisen für Startups im Umgang mit technischen Schulden und der Erkenntnis, dass unterschiedliche Startup-Phasen eine unterschiedliche Betrachtung dieser erfordern, ziehen die Autoren folgende Schlüsse:

- Das Zurückstellen von Funktionalität in der Software kann dabei helfen, trotz Drucks von Außen, die Aufnahme von technischen Schulden zu vermeiden.
- Zu Beginn einer Startup-Unternehmung sollten technische Schulden akzeptiert werden und so viele Prototypen (oder MVPs) entwickelt werden bis Produkt- und Marktanforderungen klar definiert und evaluiert sind.
- Bei fehlender Expertise im Umgang mit technischen Schulden sollten Startups früh in zusätzliche fachliche Unterstützung investieren.

Als mögliche Bedenken in Bezug auf die Validität der Forschung lässt sich auch hier die geringe Teilnehmeranzahl anführen, was die Übertragbarkeit gefährdet. Die interne Gültigkeit ist dahingehend und von außen betrachtet kaum eingeschränkt.

4.2 Qualitative Forschung zu technischen Schulden in anderen Kontexten

Im Folgenden werden einige weitere Arbeiten zum Umgang mit technischen Schulden vorgestellt, die sich der Thematik ähnlich wie hier durch qualitative Forschungsmethoden nähern.

Klinger et al. beschreiben in dem 2011 erschienen Artikel die Ergebnisse aus Interviews mit 4 Softwarearchitekten bei IBM. Sie heben hervor, dass viele der Entscheidungen zum Umgang mit technischen Schulden ad hoc und informell getroffen wurden und die Aufnahme technischer Schulden oftmals durch die Entscheidungen nicht-technischer Interessenvertreter verursacht wird[33]. Zu ähnlichen Ergebnissen kommen auch Lim et al. in ihrer Studie basierend auf 35 Experteninterviews, welche 2012 veröffentlicht wurde. Sie stellen zudem heraus, dass eine offene Kommunikation mit Interessenvertretern zu technischen Schulden sowie innerhalb des Entwicklerteams wichtig ist und eine darauf basierende Dokumentation helfen kann technische Schulden sichtbar und beherrschbar zu machen[40].
In ihrem 2013 erschienen Artikel untersuchen Codabux und Byron mittels Observation, halbstrukturierter Interviews und einem Fragebogen im Umfeld einer größeren agilen Softwareentwicklung den Umgang der Organisation mit technischen Schulden. Nach ihren Beobachtungen ist es sinnvoll, ein separates Sprint-Team dediziert auf den Abbau von technischen Schulden anzusetzen, sowie dem Team / den Teams, welche mit dem Entwickeln der Funktionalität betraut sind, ca. 20% ihrer Zeit zur Reduzierung von technischen Schulden zur Verfügung zu stellen[11].
Ernst et al. nähern sich der Thematik mittels einer Umfrage mit 1800 Teilnehmern und anschließenden Interviews. Sie kommen u.a. zu den Ergebnissen, dass die meisten technischen Schulden durch Architekturentscheidungen verursacht werden und dass das Management oft keinen klaren Überblick über die mit den technischen Schulden einhergehenden Problemen hat[17].
Freire et al. identifizieren mittels eines Fragebogens bestehend aus offenen und geschlossenen Fragen, welcher von 432 Entwicklern und weiteren an der Softwareentwicklung beteiligten Personen beantwortet wurde, mangelndes organisatorisches Interesse, geringe Priorität der Schulden, Fokus auf kurzfristige Ziele, Kosten und

Zeitmangel als die Hauptursachen für das nicht Zurückzahlen eingegangener technischer Schulden[20]. Almeida et al. schlagen in ihrem 2020 erschienen Artikel ein Verfahren zur besseren Priorisierung von technischen Schulden vor, mit dem Ziel die unterschiedlichen Interessen von technischen und und nicht-technischen Prozessbeteiligten besser aufeinander abzustimmen, evaluieren dieses durch eine fünfmonatige Fallstudie und identifizieren 8 Faktoren, die die Priorisierung von technischen Schulden beeinflussen[15].

Neben den hier aufgelisteten Forschungsarbeiten wird an dieser Stelle außerdem nochmal auf alle in den Kapiteln 2 und 8 zitierten Arbeiten verwiesen.

5 Methodik

Wie durch die Diskussion verwandter, wissenschaftlicher Arbeiten in Kapitel 4 deutlich wird, handelt es sich bei der Forschung zum Umgang mit technischen Schulden in Startup-Unternehmungen um ein recht neues Feld mit wenig gesicherten Erkenntnissen. Folglich hat auch diese Arbeit explorativer Natur mit dem Ziel qualitative Forschungsergebnisse in Form von Thesen und Beobachtungen aufzustellen und so den Grundstein für weitere Forschungen ein wenig zu verbreitern und zu stabilisieren. Die hier betriebene Forschung orientiert sich dabei an der durch die Soziologen Glaser und Strauss beschriebenen „Grounded Theory Methodology“(z.T. als GT oder GTM abgekürzt)[24]. Dieser Forschungsansatz vereint die Planung, Durchführung und Auswertung und ist besonders für die explorative Sozialforschung geeignet, wie Bohm beschreibt:

> „Die GT eignet sich dann, wenn das Verständnis größerer Textmengen bzw. ein vertieftes Verständnis angebracht ist, wenn aus den Texten neue Überlegungen, Zusammenhänge, Konsequenzen und Handlungsempfehlungen für einen Gegenstandsbereich abgeleitet werden sollen.“[8]

Zu beachten ist dabei, das anders als bei einer konkreten Methode die GTM einen weniger strengen Rahmen vorgibt. So schreibt Strauss:

> „Methodologisch gesehen ist die Analyse qualitativer Daten nach der Grounded Theory [...] keine spezifische Methode oder Technik. Sie ist vielmehr als ein Stil zu verstehen, nach dem man Daten qualitativ analysiert und der auf eine Reihe von charakteristischen Merkmalen hinweist [...], um die Entwicklung und Verdichtung von Konzepten sicherzustellen.“[45]

Nach Przyborski und Wohlrab-Sah setzt sich die Grounded Theory aus 5 Grundprinzipien zusammen:

> „1. dem Theoretischen Sampling und – darauf basierend – dem ständigen Wechselprozess von Datenerhebung und Auswertung; 2. dem theorieorientierten Kodieren und – darauf basierend – der Verknüpfung und theoretischen Integration von Konzepten und Kategorien; 3. der Orientierung am permanenten Vergleich; 4. dem Schreiben theoretischer Memos, das

> den gesamten Forschungsprozess begleitet, sowie 5. der den Forschungsprozess strukturierenden und die Theorieentwicklung vorantreibenden Relationierung von Erhebung, Kodieren und Memoschreiben.“[43][6]

Die hier durchgeführte Forschung orientiert sich an diesen Grundprinzipien. So wurde der ständige Wechsels zwischen Datenerhebung und Auswertung praktiziert und immer zuerst die geführten Interviews (1 bis 2) ausgewertet bevor neue Interviews geplant wurden. Das theoretische Sampling empfiehlt nicht blindlings so viele Daten wie möglich zu erheben, sondern gezielt anhand der in der Auswertung aufgeworfenen Fragen nach Möglichkeiten zu suchen, diese zu beantworten. Erst wenn alle offenen Fragen beantwortet sind - also eine Sättigung der Konzepte erreicht ist - wird die Datenerhebung abgeschlossen. Dieses Prinzip fand Anwendung durch das Führen von Folgeinterviews, falls sich im Zuge der Auswertung Unklarheiten ergaben. Des Weiteren wurde versucht, noch weitere Unternehmen, die sich in wesentlichen Charakteristika (insbesondere Größe des Entwicklerteam) von den untersuchten Unternehmen unterschieden, für die Forschung zu gewinnen, um so die entwickelten Konzepte und Thesen weiter anzureichern.

Durch das theorieorientierten Kodieren, das permanente Vergleichen und das Schreiben theoretischer Memos werden die Rohdaten - hier die Transkripte der Interviews - ausgewertet und die Theoriebildung vorangetrieben. Das Kodieren, das auch im Rahmen dieser Arbeit angewendet wurde, ordnet im ersten Schritt - dem offenen Kodieren - den Rohdaten also beispielsweise einzelnen Aussagen der Interviewpartnern Konzepte zu. Mithilfe dieser Konzepte lassen sich dann Aussagen vergleichen, um Unterschiede oder Parallelen abzuleiten. Konzepte lassen sich zudem zu Kategorien zusammenfassen um weiter Richtung Theoriebildung zu schreiten. So wurden im Rahmen dieser Arbeit unterschiedliche Gründe für die bewusste Aufnahme von technischen Schulden, die von den Interviewpartnern genannt wurden, als Konzepte kodiert und in der Kategorie „Bewusste Aufnahme von technischen Schulden“ zusammengefasst. Darauf aufbauend wird dann mithilfe des achsialen Codierens eine Kategorie ins Zentrum der Betrachtung gestellt, um so die Beziehungen zu den anderen Kategorien zu ergründen. Dies fand auch hier Anwendung und mündete in der Herausbildung der Schlüsselkategorie dieser Arbeit, welche in Kapitel 8.2 beschrieben wird. Nach Herausbilden der Schlüsselkategorie werden die Daten mithilfe des selektiven Kodierens ausgerichtet auf die Schlüsselkategorie neu ausgewertet und kodiert, um die Theorie vollständig herauszuarbeiten und in die Daten zu integrieren. Die geringe Datengrundlage war ursächlich dafür, dass dieser Schritt keine gesonderte Anwendung während der Auswertung der hier beschriebenen Forschung fand.

Das Schreiben theoretischer Memos, dessen Zweck das Festhalten von Erkenntnissen und theoretischen Überlegungen ist und welches gesondert von der Datenauswertung gehandhabt wird, um diese nicht zu beeinflussen, fand hier aufgrund der relativ überschaubaren Forschung und Datengrundlage weniger formale Anwendung.

Der 5. von Przyborski und Wohlrab-Sah angeführte Punkt - der den Forschungspro-

[6] Alle weiteren Erklärungen zur Grounded Theory stammen aus dem hier zitierten Werk von Przyborski und Wohlrab-Sah

zess strukturierenden und die Theorieentwicklung vorantreibenden Relationierung von Erhebung, Kodieren und Memoschreiben - verbindet die 4 vorangegangenen. Wichtig zu betonen ist, dass diese gesamte Auswertung hin zur Theoriebildung kein linearer Prozess ist, sondern vom ständigen Wechselspiel zwischen Datenerhebunng, Kodieren und Memos Schreiben zur Entwicklung von Hypothesen geprägt ist.

6 Erhobene Daten

6.1 Vorgehen zur Datenerhebung

Während zu Beginn geplant wurde, Unternehmen längerfristig vor Ort zu begleiten, so den Alltag, die Entwicklungsarbeit, Diskussionen und Entscheidungsprozesse zu observieren und mithilfe von Interviews weiter zu kontextualisieren, musste aufgrund der Covid-19-Pandemie umdisponiert werden. Anstelle dessen wurden ausführliche Interviews mit Entwicklern und Gründern von Startup-Unternehmungen geführt. Der Fragenkatalog für die ersten Interviews wurde dabei inspiriert durch die Forschungsergebnisse der verwandten Arbeiten und mit dem Ziel erstellt, die dort aufgeführten Thesen zu validieren. Das geht ein wenig entgegen der Grounded Theory, wonach man möglichst unbedarft mit der Forschung starten sollte, da allerdings die eigentlich geplante Observation wegfiel, wurde diese Herangehensweise gewählt, um schnellstmöglich viele relevante Daten zu erheben und in der Hoffnung möglichst frühzeitig auf interessante und näher zu untersuchende Phänomene zu stoßen. Neben dem Erheben der Charakteristika der Unternehmen und der befragten Personen sowie der Entwicklungsgeschichte wurden u.a. folgende Fragen den Erfahrungen mit technischen Schulden gestellt:

- Sind Sie mit dem Konzept der technischen Schulden vertraut? Was verstehen Sie darunter?
- Sind technische Schulden ein diskutiertes Thema während der Entwicklung?
- Welche Teile aus eurem Code enthalten deiner Meinung nach besonders hohe technische Schulden? Wie sind diese entstanden und haben sie einen Einfluss auf die Entwicklung?
- Gibt es eine langfristige Strategie, was technische Schulden angeht? Wird mit zukünftigem Abbau oder Verschuldung geplant?
- In welchen Bereichen habt ihr die höchsten technischen Schulden (Einteilung: Dokumentation, Code, Architektur, Tests)?
- Wie autonom können Entwickler Entscheidungen zur Aufnahme bzw. zum Abbau von technische Schulden treffen?
- Wie sehen Diskussionen und Entscheidungsprozesse rund um die Thematik der technischen Schulden aus?

Mit Voranschreiten der Forschung, dem Observieren von interessanten Aspekten und dem Aufstellen erster Theorien entfernten sich die Interviews von dem ursprünglichen Fragenkatalog und es wurde gezielter nach einzelnen Anekdoten und Erfahrungen gefragt, um die aufgestellten Theorien zu validieren und anzureichern. Die Interviews wurden transkribiert und einzelne Aussagen der Interviewpartner kodiert, um Gemeinsamkeiten und Unterschiede herauszuarbeiten.

6.2 Untersuchte Unternehmen

Im Rahmen der Befragung wurden Entwickler*innen und Gründer*innen aus 3 verschiedenen Unternehmen befragt. Die wesentlichen Characteristika sind in Tabelle 1 aufgelistet. Zur Wahrung der Anonymität der befragten Personen sind auch die Unternehmen anonymisiert. Im Weiteren werden den Unternehmen die Buchstaben von A bis C zugeordnet. Die Reihenfolge entspricht dabei der chronologischen Reihenfolge der geführten Interviews.

Name	Technologie	Geschäftsbeziehung	Gründung	#Entwickler
A	Machine Learning	b2b	2016	$5-10$
B	Data Visualisation	b2b	2015	$5-10$
C	Bildverarbeitung	b2c (überwiegend)	2013	< 5

Tabelle 1: Charakteristika untersuchter Startups

Alle in dieser Arbeit diskutierten Unternehmen wurden durch einen Absolventen eines Informatik-Masterstudiengangs [7] oder einem Doktor der Informatik gegründet bzw. co-gegründet. Bei den Unternehmen B und C handelt es sich dabei um direkte Ausgründungen aus der Universität heraus. Die technologische Grundlage ist dabei an der Universität im Rahmen von Seminaren oder einer Dissertation entstanden. Bei B wurden die Arbeiten durch weitere Abschlussarbeiten und Promotionen noch vor der eigentlichen Gründung des Unternehmens vorangetrieben. B und C sind bis zum Zeitpunkt der Befragung eng mit der Universität verbunden. Unternehmen A hingegen ist zwar Teil einer Service-Einrichtung für die Förderung von Unternehmensgründungen und Innovationen einer Universität, pflegt darüber hinaus allerdings keine engen Bindungen zur Forschung und Lehre an dieser.
Bei den befragten Unternehmen handelt sich sich um etablierte und erfolgreiche Software-Startups, deren Geschäftsmodell auf der Arbeit mit innovativen Technologien fußt. Ob die befragten Unternehmen zum Zeitpunkt der Datenerhebung, aufgrund des verhältnismäßig langen Bestehens, formal unter den Begriff des Startups fallen, ist eine eher nebensächliche Diskussion. Fakt ist, dass die Unternehmen aufgrund ihrer Größe, des Innovationsgrads und der Unternehmenskultur viele Überschneidungen zu klassischen Startups haben und definitiv zu einem gegebenen Zeitpunkt in der Vergangenheit unter die klassische Definition des Begriffs fielen. Ob und zu welchem Zeitpunkt sich das geändert hat, ist eine eher akademische und wirtschaftswissenschaftliche Diskussion, die an dieser Stelle nicht fortgeführt wird.

[7] oder eines vergleichbaren Studiengangs

6.3 Geführte Interviews

In Tabelle 2 sind die interviewten Personen aus den Unternehmen aufgelistet. Die Befragung wurde im Zeitraum von Januar bis Juni 2021 durchgeführt. Nach Möglichkeit wurde zudem mit jedem Interviewpartner versucht ein zweites Gespräch zu arrangieren, um Nachfragen stellen zu können, Annahmen zu validieren und über Erfahrungen zu sprechen, die gemacht wurden, nachdem durch das erste Interview auf das Thema der technischen Schulden etwas sensibilisiert wurde. Das erste Interview war dabei immer angelegt auf ca. 1 Stunde und das zweite auf ca. 30 Minuten. Die Befragung der Entwickler*innen von Unternehmen C wurden nicht einzeln, sondern in Form eines Gruppeninterviews durchgeführt. Dies hatte den Vorteil, dass das Gespräch weniger stur durch die Fragen des Interviewers gelenkt werden musste. Auf der anderen Seite kann eine Befragung innerhalb der Gruppe dazu führen, dass bestimmte negative Erfahrungen nicht so offen geteilt werden.
Die Interviews wurden aufgrund der Covid-19 Pandemie ausschließlich online über das Video-Konferenzwerkzeug WebEx geführt.

Kürzel	Unternehmen	Position	Jahre im Startup	Datum
A1	A	CEO / Gründer	5	08.01.2021 & 09.04.2021
A2	A	Entwickler	4	13.01.2021 & 25.05.2021
B1	B	CEO / Gründer	7	24.02.2021
C1	C	CEO / Gründer	8	08.04.2021 & 02.06.2021
C2	C	CTO	7	12.04.2021
C3	C	Entwickler	6	12.04.2021
C4	C	Entwickler	1	12.04.2021
C5	C	Technischer Berater	-	01.03.2021

Tabelle 2: Befragte Personen

7 Ergebnisse

In diesem Abschnitt werden die einzelnen beobachteten Phänomene rund um den Umgang mit technischen Schulden in den untersuchten Unternehmen nach Themengebieten gruppiert und Gemeinsamkeiten und Unterschiede über die geführten Interviews hinweg diskutiert.

7.1 Ansammlung technischen Schulden zu Beginn einer Start-up-Unternehmung

Das untersuchte Unternehmen C steht beispielhaft für alle Startups, die aufgrund von knappen Ressourcen um den Zeitpunkt der Gründung herum unter einem hohen Druck stehen. So beschreibt der Gründer die finanzielle Situation zur damaligen Zeit mit den Worten:

> „Wir hatten in den ersten Monaten [...] nicht genug, um unserem einen Entwickler einen Döner jeden Tag zu kaufen. Also ich komme aus Zeiten, da habe ich in Döner gerechnet tatsächlich. Da war das sehr sehr eng.“ - C1

Im Falle des Unternehmens C resultierte dieser finanzielle Druck unmittelbar in dem Zwang schnell Funktionalität zu schaffen und einer damit verbundenen bewussten Aufnahme von technischen Schulden:

> „Da war es wichtiger [...] Features umzusetzen oder die Monetarisierung anzuschieben, als die Software-Basis sauber zu haben.“ - C1

Vor allem zu Beginn der Entwicklung birgt die Aufnahme von technischen Schulden natürlich ein besonders hohes Risiko, da der dort konzipierte und geschriebene Code die Grundlage für alle folgenden Entwicklungen bildet. Die Applikationen bei C basieren teilweise immer noch auf dem Code, der ganz zu Beginn geschrieben, aber über die Jahre auch weiterentwickelt wurde. Über technische Schulden am Kern der Software sagt zum Beispiel Gründer A:

> „Je näher ich an die Architektur einer Softwarelösung dran komme, desto aufwändiger ist es dann, irgendetwas, das schnell gemacht wurde, nachher zu korrigieren. Das heißt der Zinssatz ist höher.“ - A1

Hier wird das Spannungsfeld, in dem Software-Startups zu Beginn stehen, besonders deutlich. Zum einen muss schnell Funktionalität geschaffen werden, um Investoren oder Kunden zu gewinnen, auf der anderen Seite beeinflusst der architekturelle Grundstein, der zu Beginn der Softwareentwicklung gelegt wird, die darauf aufbauende Softwareentwicklung erheblich, was besonders bei Unternehmen C deutlich zu beobachten war. Der Aufwand, der alleine mit der Implementierung eines gut erweiterbaren Software-Kerns verbunden ist, wird an folgender Aussage deutlich:

> „Wenn das Programm von vornherein so geschrieben ist, dass die Funktion oder [der] Funktionsumfang relativ wartungsarm erweitert werden kann, ist es dann natürlich auch teilweise [mit] sehr viel mehr Aufwand [verbunden], als wenn man einen relativ feststehenden kleineren Funktionsumfang implementiert.“ - A2

Dieses Spannungsfeld beschreibt auch Gründer A. Fehlende Funktionalität oder eine „schäbig aussehende GUI“ sind mit direkten negativen Konsequenzen für das Startup und die Kundenakquisition verbunden, wohingegen die Aufnahme von technischen Schulden im ersten Schritt für den Kunden unsichtbar ist, später aber sehr teuer zu beheben sein kann.

> „Es ist wie bei jedem Kredit die Frage: Wann wird er fällig gestellt? Und Sie können sich vorstellen, dass Sachen die in der GUI [schlecht] sind, bei jeder Demo fällig gestellt werden. Und Sachen, die in der Architektur [schlecht] sind, die werden dann fällig gestellt, wenn man schon eine stabile Geschäftsbeziehung hat. Daraus könnte man ableiten:≪Ich kann sehr sehr viel architekturelle Schulden aufnehmen, weil der Kunde es

> de facto nie merkt.» Aber gleichzeitig wissen wir als Entwickler, dass wenn ich das überreize, dass es einem irgendwann um die Ohren fliegt. Und genau das [ist], was diese unheimlich fragile Balance ausmacht. Dass es eigentlich da einfacher ist, Schulden aufzunehmen, wo der Kunde es nicht merkt, dass gleichzeitig die Sachen, die der Kunde nicht bemerkt, meistens genau diejenigen sind, wo die Schulden einen höheren Zinssatz haben.“ - A1

Während Unternehmen C also bewusst ein Risiko eingegangen ist, um den Aufwand zu verringern, konnten die technischen Schulden am Softwarekern nach Einschätzung des Gründers u.a. durch die eigene technische Expertise abgepuffert werden:

> „Wir haben dann auch angefangen andere Rendering-Techniken zu implementieren. Da kamen neue Anforderungen rein [...] und da war dann die Frage: Hält der Rendering-Kern an der Stelle? Dadurch dass [...] wir was Software Engineering anging recht gut ausgebildet, recht gut erfahren waren, hat die Architektur den neuen Anforderungen standgehalten.“ - C1

Die Unternehmen A und B standen hingegen zu Beginn nicht unter diesem enormen finanziellen Druck. Während bei C aufgrund der finanziellen Lage und des daraus resultierenden zeitlichen Zugzwangs auch besonders teure, architekturelle, technische Schulden am Software-Kern aufgenommen wurden, die mehrere Refactorings nach sich zogen, wurde hingegen bei A die Aufnahme von technischen Schulden insbesondere bezogen auf die Architektur vermieden.

> „Was technische Schulden angeht, hat sich eher wenig geändert, weil [der Gründer] - insbesondere ganz von Anfang an - drauf geachtet hat, [...] defensiv zu programmieren. Also da legen wir schon ein Augenmerk drauf, dass wir da lieber ein bisschen mehr Zeit investieren.“ - A2

Was alle Unternehmen in ihrer Gründungsphase verbindet, ist der Umgang mit neuen Technologien und die damit verbundene unbewusste Aufnahme von technischen Schulden, die sich auch nach Cunnigham gar nicht vermeiden lässt. So resümiert der Gründer von Unternehmen B, der auch den initialen Prototypen konzipiert und implementiert hatte:

> „Ich glaube am Anfang waren 90% der technischen Schuld unbewusst. Also es war einfach in dem System drin. Von Natur aus.“ - B1

Von Beginn an wurde mit hohen Qualitätsstandards und dem „rapid iterativ prototyping“ in Unternehmen A versucht dieser unbewussten Aufnahme so weit wie möglich vorzubeugen. Aber besonders in Bezug auf die Auswahl von zugrundeliegenden Frameworks, Programmiersprachen und Bibliotheken, wurden auch dort Entscheidungen getroffen, die sich rückblickend als suboptimal herausstellten und Refactorings nach sich zogen. Selbst als promovierter Informatiker zieht der Gründer hier das Fazit:

> „Eigentlich kann man sich für solche Technologie [...] nur entscheiden, wenn man wirklich schon mehrjährige Erfahrung mitgebracht hat.“ - A1

Darüber hinaus sind die Anforderungen bei der Entwicklung neuer Technologien nicht ganz klar bzw. können sich auch mit der Zeit ändern. Dies kann dazu führen, das bestimmte Teile der Codebasis besonders am Anfang oft überarbeitet werden müssen. Daraus ergab sich bei Unternehmen C, dass man auf die Softwaretests u.a. aus Frustration verzichtete:

> „Jetzt hast du das Problem du schreibst so was wie einen [Software-] Kern. Der besteht aus 20-50 Klassen und du entwickelst ihn gerade. [...] Idealerweise schreibst du erst den Test und dann die Klasse. Und beim Entwickeln stellst du fest: ≪Das, was ich mir da jetzt überlegt hatte, ist doch Käse.≫ [Und] dann kommt noch [eine neue] Anforderung hinzu. D.h. du fängst wieder von vorne an [und] du schmeißt auch alle Tests wieder weg. [...] [Das ist das Problem] mit den Tests: es potenzieren sich dann einfach auch die Zeit und Nerven, die man da verliert. Und ich glaube deswegen ist es so was, was man oft ignoriert.“ - C3

Während diese Überlegungen wohl typisch für frühe Softwareentwicklung sein kann, kam bei C erschwerend hinzu, dass die Softwaretests nicht trivial sind und für die Entwicklung dieser mindestens der gleiche Aufwand wie für die eigentliche Entwicklung von Nöten war.

Dieser Effekt kann zudem durch die geringe Anzahl an Entwicklern zu Beginn einer Startup-Unternehmung verstärkt werden. So sind die Entwicklungen der Unternehmen B und C als 1-Mann-Projekt gestartet. Diese geringe Anzahl an Entwicklern kann dazu führen, dass keine oder ungenügende Qualitätsstandards und Kontrollstrukturen eingeführt werden. Noch vor der Gründung des Unternehmens B wurde die Software dazu im Rahmen einer Masterarbeit und späteren Promotion von einer Person geschrieben. Diese Entwicklungsphase vergleicht der Gründer im Gespräch rückblickend mit der chaotischen Entwicklung mancher Open-Source-Projekte:

> „Da merkt man, [viele Open-Source-Projekte] sind da, wo wir vor 10 Jahren waren. [...] Wenn man alleine programmiert und einen Prototypen zusammen baut, dann geht man immer genau diesen einen Entwicklungsweg lang und baut was und testet nur diesen einen Fall [...] und guckt nicht nach links und rechts. Und das sieht man halt an den ganzen Systemen. Die auch nicht dokumentiert sind und da findet man sich so ein bisschen wieder.“ - B1

Während diese Art des Programmierens zu Beginn zu Zeitersparnissen führt und sich die negativen Effekte von technischen Schulden bei einem kleinen Entwicklerteam und einer überschaubaren Code-Basis in Grenzen halten, werden die Probleme sichtbar sobald sich neue Entwickler und Funktionalitäten hinzukommen. So besonders ausgeprägt zu beobachten bei Unternehmen B:

> „Und so sind halt immer mehr Leute dazu gekommen, die sich immer mehr oder immer besser mit dem Framework ausgekannt haben. [...] Und dann ist das Framework immer größer geworden. Dann sind andere Leute dazu gekommen, die auch in dem Bereich promoviert haben, die ganz

> andere Aspekte dazu gebracht haben. Dann sind wieder neue Module hinzu gekommen und dann war es irgendwann so, wo wir gemerkt haben: ≪Okay, es ist nicht mehr so richtig gut wartbar.≫ Da sind dann Fehler reingekommen, dann hat einer was geändert, dann ist es uns an einer anderen Ecke [...] auseinander geflogen. Und das war halt ein Problem.“ - B1

In diesem Fall ist allerdings anzuführen, dass die Entwicklung bei Unternehmen B zum damaligen Zeitpunkt ausschließlich an der Universität stattfand und im Rahmen von Abschlussarbeiten oder Promotionen von Studenten vorangetrieben wurde. Diese im Vergleich zu den anderen Startups deutlich erhöhte Fluktuation an Entwicklern kann als Katalysator verstanden werden, der die Entwicklungsaufwände aufgrund von fehlender Dokumentation, Qualitätskontrolle und Testing in die Höhe trieb.

Bei Unternehmen C stellte vor allem das Wissensmonopol des ersten Entwicklers, der den Software-Kern als 1-Mann-Projekt geschrieben hatte, ein größeres Problem da. So sagt ein anderer Entwickler:

> „Wenn es ein Feature gab, dass irgendwie mit dem Kern zu tun hatte, hatte ich von vornherein schon weniger Bock mich da dran zu setzen, weil da viel mehr Aufwand mit verbunden war das dann anzupassen, weil ich mich da erst einmal rein arbeiten musste und eine andere Sprache nutzen [musste]. Und dich (1. Entwickler) dann erst fragen, wie du dir das gedacht hast damals und so.“ - C4

In dieser Aussage werden die erhöhten Zeitaufwände und auch möglichen Frustrationen, die aus den technischen Schulden resultieren, ganz stark deutlich. In der aktiven Entwicklung führte das dazu, dass in der Regel bzw. nach Möglichkeit nur der 1. Entwickler mit dem Software-Kern arbeitete. Ein solches Wissensmonopol in einem bestimmten Codeabschnitt lässt sich bei kleinen Entwicklungsteams natürlich nicht gänzlich vermeiden, allerdings werden die damit verbundenen zeitlichen Mehraufwände durch technische Schulden - wie schlecht lesbaren Code oder unzureichende Dokumentation weiter in die Höhe getrieben. Gründer C sagt über die Dokumentation während der frühen Entwicklung:

> „Weil wenn eine Person Code schreibt, dokumentiert sie möglicherweise parallel, hat aber definitiv nicht die Qualität, als wenn du ein 100-Mann-Team hast [...], die alle auf der Code-Basis arbeiten und unterschiedliche Sichten darauf haben. Also da sind wir, was die Qualität angeht nicht da, wo wir sein wollten.“ - C1

Diese unbewusst eingegangenen technischen Schulden, die aus der Entwicklungsarbeit einer Person bei den Unternehmen B und C resultierten, wurden mit der Zeit aber immer weiter abgebaut, was nach Einschätzung des technischen Beraters bei C u.a. auf langsames Wachstum des Entwicklerteams zurückzuführen ist:

> „Mein Eindruck ist auf jeden Fall, dass [das Level an technischen Schulden] geringer geworden ist und dass es an der Anzahl der Mitarbeiter hängt. An der Anzahl der qualifizierten Mitarbeiter. “ - C5

Ähnlich resümiert auch Gründer B:

> „Aber diese Schuld, die im System unbewusst drin ist, nimmt immer weiter ab. “ - B1

Und führt das u.a. auf die eingeführten Qualitätsstandards und Kontrollstrukturen zurück, nachdem die Probleme mit technischen Schulden deutlich zum Vorschein getreten sind:

> „Die Codequalität hat sich dann verbessert, wo wir gesagt haben: ≪Wir brauchen die Merge-Request.≫“ - B1

Zusammenfassend werden aus den geführten Interviews folgende Thesen zur Ansammlung von technischen Schulden zu Beginn einer Startup-Unternehmung abgeleitet:

1. Software-Startups, die zu Beginn unter enormen Druck aufgrund von Ressourcenknappheit stehen, müssen technische Schulden aufnehmen, um schnell Funktionalität schaffen zu können.

2. Technische Schulden, die in den frühen Phasen der Softwareentwicklung aufgenommen werden, bergen die Gefahr, vor allem wenn sie architektureller Natur sind, die Entwicklung langfristig, negativ zu beeinflussen.

3. Die Arbeit mit innovativen Software-Technologien führt zwangsläufig zur unbewussten Aufnahme von technischen Schulden aufgrund von unklaren Anforderungen und daraus resultierenden suboptimalen Designentscheidungen.

4. Kleinen, neu gebildeten Entwicklungsteams kann es zu Beginn an klaren Qualitätsrichtlinien und Kontrollstrukturen fehlen. Dies gilt besonders für 1-Mann-Projekte. Dadurch werden zusätzliche unbewusste technische Schulden aufgenommen. Kommen neue Entwickler hinzu, werden diese Schulden und evtl. daraus resultierende Probleme oder zeitliche Mehraufwände deutlich sichtbar.

7.2 Refactoring als wirtschaftlich bedeutende Entscheidung

Wie oben beschrieben nahmen vor allem die Unternehmen B und C in der ersten Phase der Entwicklung viele technische Schulden auf. Gleichzeitig wuchsen die Softwaresysteme weiter, da immer neue Anforderungen hinzu kamen. So beschließt Gründer C eine längere Aufzählung von (Rendering-)Technologien, Monetarisierungskonzepten und KI-gestützten Verfahren mit den Worten:

> „[Und] das sind alles Sachen, Anforderungen, technische Neuerungen oder konzeptionelle Neuerungen, die über die Jahre hinzu kamen. Wo das Softwaresystem standhalten und mitwachsen musste.“ - C1

Diese neu hinzukommenden Anforderungen führen unter Zeitdruck fast zwangsläufig dazu, dass sich die Änderungen nicht ideal in die Architektur des bestehenden Systems einpflegen lassen, selbst wenn versucht wurde, dieses so offen wie möglich gegenüber potentiellen Änderungen zu gestalten:

> „Wir haben ein Konzept entwickelt auf den Anforderungen, die wir bis dahin hatten und versucht die, die so ein bisschen im Kopf waren, gleich mit einzupflegen. Dann ist es natürlich wie immer im Leben so, dann kommen Sachen nochmal dazu, die man halt nicht wusste und dann werden immer so 1,2,3 vielleicht auch mal 5 Sachen ran gebastelt. [...] Es gibt bestimmte Sachen, wo sich klar die Anforderungen entwickeln über die Zeit. Dann fängt man halt an Sachen dazu zu bauen und die müssen halt schnell erledigt werden. “ - C3

Während diese Aufnahme zu Beginn dazu führt, dass schneller neue Funktionalität geschaffen werden konnte, wachsen aber mit der Zeit auch die Probleme, die damit einhergehen. Bei A wurden diese Schulden zum Beispiel dadurch deutlich, dass nur der ursprüngliche Entwickler (auch CTO) effizient mit dem Software-Kern arbeiten konnte:

> „Da war dann immer die Sache: ≪Also wir können das jetzt entweder [mich] machen lassen, dann dauert das 3 Tage oder [der CTO] macht das jetzt in einer Stunde. Na gut, dann macht das [der CTO] jetzt in einer Stunde und [ich] mache in den 3 Tagen lieber was anderes Sinnvolles.≫ Da war das dann eigentlich soweit, dass wir wussten: ≪Okay das muss neu geschrieben werden.≫ Das war dann auch der Punkt: ≪Naja es macht keinen Sinn, dass [ich mich] da jetzt einarbeite, weil wir schreiben das ja eh neu.≫“ - C3

Er fügt dann allerdings noch an:

> „Gut, dass das jetzt 5 Jahre dauert, hat da jetzt auch keiner gewusst an der Stelle.“ - C3

Dieser Satz verdeutlicht, dass man sich bei C schon sehr lange den technischen Schulden innerhalb des Software-Kerns bewusst war, das nötige Refactoring, um diese Schulden abzubauen, allerdings sehr lange aufschob. Hinzu kam, dass die Technologie (eine plattform- und programmiersprachenübergreifende Programmierschnittstelle), auf der das Softwaresystem basierte, auf absehbare Zeit veraltet sein würde und durch eine neue ersetzt werden müsste. Ein Refactoring war also gar nicht zu vermeiden:

> „Also sprich, wenn wir expandieren wollen, geht’s nicht mit dem alten Rendering-Kern. Trotzdem haben wir Ewigkeiten dran festgehalten , weil wir halt das Verfahren noch rein bringen wollten, die App loungen wollten, damit mehr Umsatz generiert werden kann. [...] Um dann ausreichend finanzielle Ressourcen, Zeit Ressourcen zu haben, um dann das Mammut-Projekt Rendering-Kern anzugehen.“ - C1

Hier waren also ganz eindeutig wirtschaftliche - insbesondere finanzielle Überlegungen - die Ursache für das lange Hinauszögern des Refactorings. Der CTO begründet diese Entscheidung zudem mit dem hohen Druck neue Funktionalität liefern zu müssen, da einige der entwickelten Apps durch ein Abo-System monetarisiert sind:

> „Hauptproblem, warum wir es nicht genutzt haben, ist, dass das - wie der Name schon sagt - eine absolute Kernkomponente ist, die mittlerweile auch weiß ich nicht wie viele Tausend Zeilen Code hat, und das zu portieren dauert verdammt lange. Vor allem weil wir auch ganz viele Features haben, die [wir] dann alle mit portieren müssen. [...] Im Idealfall [wenn das Refactoring abgeschlossen ist] sehen die Apps ganz genauso aus, können aber nicht mehr und wenn du gerade Apps hast, die z.B. ein Subscription Modell haben, wo die Nutzer erwarten, dass da auch ab und zu was passiert. Da kannst du nicht sagen: ≪Sorry Leute wir sind jetzt mal für ein Jahr weg, wir müssen mal die Rendering-Technologie neu schreiben.≫“ - C2

Auch der Entwickler C3 betont im Gespräch, wie schwierig es ist, den richtigen Zeitpunkt für ein Refactoring zu finden, weißt allerdings auch auf die immer größer werdenden Probleme hin, je weiter der Abbau der technischen Schulden hinauszögert wird, da man alleine dadurch gezwungen sei, noch weitere Schulden aufzunehmen. Und wie der CTO in dem Satz davor anmerkte, dass das weitere Implementieren von Funktionalität auf Basis des alten Software-Kerns das Refactoring nur noch aufwändiger machen würde.

> „Ich glaube die Zeit [für ein Refactoring], die hat man eigentlich nie. Man muss sich irgendwie zwingen. Wenn du einfach ein paar mal was dran gebastelt hast, wo du eigentlich merkst: ≪Okay, das ist eigentlich ganz schrecklich.≫ Das will man eigentlich nicht mehr machen. Wird halt in der Zukunft auch nur schlimmer. Wenn du an so etwas Rangefaffertes nochmal was ran gaffern[8] musst, wird's halt auch nicht besser.“ - C3

Bei C war also auch die schiere Größe des Software-Kerns und der enorme damit verbundene Aufwand, diesen von Grund auf neu zu implementieren, der Grund für das lange Aufschieben. Besonders für kleine Software-Startups erfordert ein solch umfangreiches Refactoring die Aufmerksamkeit des gesamten Entwicklerteams, was auf der anderen Seite bedeutet, dass keine neue Funktionalität geschaffen werden kann. D.h. es kann mit unter schwer sein Entscheider und Investoren von der Notwendigkeit eines Refactorings zu überzeugen.

> „Wir sind ein finanziertes Startup, d.h. wir haben Geldgeber und die haben natürlich auch Wünsche und Agenden, die sie halt umgesetzt haben wollen. Und gerade gegenüber denen ist es halt schwierig zu sagen: ≪Ne sorry, wir müssen jetzt erst mal eine Menge Geld verbrennen, um die Basis neu zu machen.≫ Wenn die sehen: Die Basis funktioniert ja. [...]
> Und jetzt haben wir uns halt wirklich seit Mitte letzten Jahres gesagt: ≪Okay sorry das [Refactoring] müssen wir jetzt machen. Alle coolen neuen Ideen, die wir jetzt haben, wenn wir die jetzt noch auf den alten Kern immer weiter drauf packen und gleichzeitig versuchen was Neues zu machen. Das ist total sinnfrei. Wir müssen jetzt einfach quasi einen Feature-Freeze machen. Und möglichst viele gleichzeitig hinsetzen, daran arbeiten und das jetzt durchziehen.≫“ - C2

[8] Mit „Gaffern“ ist hier das Implementieren gegen die Architektur gemeint.

Als Lösungsstrategie wurde bei A Funktionalität geschaffen, die es auch den Mitarbeitern außerhalb des Entwicklungsteams ermöglichte, relativ einfach bestimmte neue Funktionalität in den betriebenen Apps hinzuzufügen, um so auch während der Zeit des Refactorings neue Updates veröffentlichen zu können.
Dass das gesamte Entwicklungsteam bei C mit dem Refactoring betraut wurde, hatte nicht nur den Grund, dass man die Arbeiten daran so schnell wie möglich abschließen wollte, sondern das Refactoring hatte neben dem Umrüsten auf die neue Technologie und dem Abbau von technischen Schulden auch das erklärte Ziel, die Code-Ownership bezogen auf den Software-Kern auf das gesamte Team auszuweiten und so das oben beschriebene Wissensmonopol des ersten Entwicklers aufzulösen.

> „Ein Refactoring dient ja nicht nur dem Zweck des Refactorings, sondern sollte vorzugsweise auch dazu dienen, dass das Kern-Team Code-Ownership bekommt. Und dann sollte das Kern-Team auch gemeinsam da sein. Es macht keinen Sinn eine komplett neue Technologie zu schreiben, wieder durch ein, zwei Personen, die dritte Person außen vor zu lassen.“ - C1

Während man also klar festhalten muss, dass das Aufschieben eines großen Refactorings die Aufnahme weiterer technischer Schulden zu Folge haben kann und durch das Arbeiten gegen die suboptimale Architektur höhere Zeitaufwände entstehen, bietet das Hinauszögern - neben der Möglichkeit sich ein größeres finanzielles Polster anlegen zu können - auch den Vorteil, dass mehr Anforderungen zum Zeitpunkt des Refactorings bedacht werden können und die Entwickler durch das Sammeln von mehr Erfahrung mit den verwendeten Technologien diese besser einbinden können. Dies wird vor allem deutlich bei der Aussage des CTOs:

> „Jetzt wissen wir ja schon eine ganze Menge, was wir jetzt brauchen - auch vor allem an Basisfunktionalitäten. Und da ist jetzt gerade mein Ziel da wirklich Zeit zu investieren, die vernünftig zu machen und erweiterbar, das es nicht gerade ganz viel dupliziert und hingehackt ist, wie es halt vorher war. Und deswegen dauert es halt entsprechend auch länger, weil man auch grundsätzlich neu drüber nachdenken möchte.“ - C2

Auch ein anderer Entwickler betont, wie wertvoll die gemachten Erfahrungen und auch Fehler der Vergangenheit für das aktuelle große Refactoring sind:

> „Das [Refactoring] funktioniert auch deshalb so gut, weil der Core konzeptionell nicht komplett neu gemacht werden muss, weil wir einfach schon 7 Jahre Erfahrung mit einem anderen Core gemacht haben. Da gibt es schon Work-Flows oder eine Architektur, wo wir wissen, dass ist eine gute bzw. wir wissen, wo Schwachstellen sind. Und die können wir jetzt wahrscheinlich beheben. [...] Und das ist auch so eine Sache gerade wenn man das Thema Startup betrachtet. Es wird halt viel entwickelt und bestimmte Sachen werden halt erst [gut], wenn man ein paar mal damit auf die Nase gefallen ist oder Erfahrungen sammeln konnte. Und dann kann man halt auch besser Work-Flows oder Verhalten implementieren, was es einem hilft, technische Schuld zu vermeiden oder

> wahrscheinlich auch besser abschätzen und kommunizieren zu können. “ - C3

Zusammenfassend kann ein späteres Refactoring also durchaus von Vorteil sein, wenn man eine möglichst gute Grundlage für die zukünftige Entwicklung schaffen möchte.

Dass das Refactoring bei Unternehmen A so große Dimensionen annahm und einen so großen Stellenwert einnahm, ist der Tatsache geschuldet, dass die aufgenommenen technischen Schulden den funktionalen Kern der Software betrafen, welcher die Grundlage für alle weiteren Funktionalitäten darstellte. Bei Unternehmen A und B gab es nie die Notwendigkeit für so ein großes Refactoring, da hier technische Schulden am Software-Kern eher vermieden wurden. Nichtsdestotrotz spielt Refactoring vor allem durch den Umgang mit neuen Technologien eine große Rolle, konnte allerdings mehr in das Tagesgeschäft integriert werden, da die betroffenen Komponenten deutlich kleiner waren. So sagt Gründer B:

> „Wir haben sozusagen immer permanent refactort. Also wir haben nie sozusagen das ganze Ding weggeschmissen und gesagt, jetzt bauen wir es neu. Das haben wir für einzelne Sachen mal probiert, aber das war dann auch nicht viel besser. Und wir haben meistens sozusagen am bestehenden System refactort.“ - B1

Und auch in Unternehmen A werden kleinere Refactorings relativ regelmäßig durchgeführt, die aber nicht zwangsläufig den Abbau von technischen Schulden zum Ziel haben, sondern auch dazu dienen nicht funktionale Anforderungen wie Performance zu verbessern:

> „Bei uns ist es definitiv so, dass wir in regelmäßigen Abständen, die jetzt aber nicht wohldefiniert sind, tatsächlich ein Refactoring machen, um Sachen performanter oder stabiler zu machen.“ - A2

Dass ähnlich wie bei C aber auch wirtschaftliche Überlegungen und Feature-Druck bei der Planung eines Refactorings eine Rolle spielen wird u.a. bei dieser Antwort auf die Frage deutlich, warum zum Zeitpunkt des Gesprächs (Januar 2021) das Hauptaugenmerk auf Refactoring und dem Abbau von technischen Schulden lag:

> „Je nachdem wie zeitkritisch man Projekt liefern muss, hat man dann dementsprechend mehr Zeit um technische Schulden abzubauen. Tatsächlich ist es so, dass wir in der Corona-Krise nicht mehr so schnell liefern müssen wie wir das sonst gewohnt sind. Das heißt wir haben jetzt nicht so einen starken Stechschritt wie sonst immer. Von daher liegt der Augenmerk eher darauf nicht neue Features raus zu hauen oder Ergebnisse zu liefern sondern auch potentiell die Software zu verbessern. Das hängt damit zusammen. Wenn wir jetzt die ganze Zeit durch liefern müssten, dann würden wir keine technischen Schulden abbauen können.“ - A2

Zusammenfassend werden aus den geführten Interviews folgende Thesen zu Refactorings und dem Abbau von technischen Schulden in Software-Startup-Unternehmungen abgeleitet:

1. Technische Schulden vor allem architektureller Natur am Software-Kern führen zu höheren Entwicklungsaufwänden, Frustration und zur Aufnahme weiteren technischen Schulen und sollten durch ein Refactoring abgebaut werden.

2. Da durch ein Refactoring keine neue Funktionalität geschaffen wird und die Startups oft unter Druck von Seiten der Kunden oder Investoren stehen, ist es schwierig, einen geeigneten Zeitpunkt für ein Refactoring zu finden und muss sich mitunter durch Vorarbeiten in Abhängigkeit vom Umfang des zu überarbeitenden Codes erkämpft werden.

3. Durch das Aufschieben eines Refactorings können in der Zwischenzeit aber auch neue Anforderungen bekannt werden oder ins Bewusstsein rücken und es erlaubt mehr Erfahrungen mit den eingesetzten Technologien zu sammeln, was teilweise die unbewusste Aufnahme technischer Schulden durch das Refactoring vermeiden kann.

4. Der Abbau von technischen Schulden steht in einem Spannungsfeld zwischen dem Druck nach Funktionalität durch Projekte, Kunden und Investoren und dem stetigen Zuwachs des Wissens über Anforderungen und die Technologie auf der einen Seite und Zufriedenheit der Entwickler*innen, höhere Entwicklungsaufwände und weiterer Anhäufung technischer Schulden auf der anderen Seite.

7.3 Vermeidung technischer Schulden

Alle untersuchten Unternehmen versuchen im Tagesgeschäft möglichst wenig technische Schulden aufzunehmen. So beschreibt Entwickler A, dass bereits von Beginn an von Seiten der Gründer sehr stark darauf geachtet wurde defensiv zu programmieren und möglichst keine technischen Schulden anzusammeln, selbst wenn dafür ein höheres Zeitinvestment von Nöten war:

> „Was technische Schulden angeht, hat sich eher wenig geändert, weil [der Gründer] - insbesondere ganz von Anfang an - drauf geachtet hat [...] defensiv zu programmieren. Also da legen wir schon ein Augenmerk drauf, dass wir da lieber ein bisschen mehr Zeit investieren. Und wir haben auch ausdrücklich beschlossen mehr Zeit zu investieren, mehr zu testen und sicher zu sein, dass es funktioniert, als etwas schnell nieder zu schreiben. [...] Das heißt wir [...] achten tatsächlich ganz stark darauf, möglichst keine technischen Schulden anzusammeln und [...] dass der Code wartbar bleibt und eine hohe Qualität hat.“ - A2

Auch bei Unternehmen B wurde in der Regel gegen eine Aufnahme von technischen Schulden entschieden. Der Gründer führt dabei die Vermeidung von langfristigem Stress bei den Entwicklern als Grund an:

> „Wir nehmen nicht viel bewusste technische Schuld auf. Dass wir sagen: ≪Das muss man jetzt schnell machen.≫ Sondern wir nehmen uns eigentlich auch die Zeit. Und das ist auch das, was ich gelernt habe, dass es nichts bringt, langfristig die Entwickler unter Stress zu setzen und man muss denen auch einfach auch die Zeit geben zum Aufräumen.“ - B1

Sehr ähnlich dazu resümiert auch Gründer C zu Entscheidungen im Team, wenn es um das Verschieben von Releases geht, um technische Schulden zu bereinigen:

> „Die Verzögerung [bei den Releases] ergibt sich daraus, dass man technische Schulden [...] entdeckt. Und sich sagt: ≪Okay, machen wir es jetzt richtig [und] nehmen keine technische Schulden auf [oder doch?].≫ Und da ist die Entscheidung bei uns im Team sehr oft gewesen: ≪Wir nehmen keine technischen Schulden auf.≫ [Mit] Ausnahme [des] Rendering-Kern[s]. Wir haben immer probiert möglichst wenig technische Schulden aufzunehmen.“ - C1

Interessant ist bei diesem Zitat neben der Untermauerung der oben beschriebenen Beobachtung auch folgende Einschränkung, die der Gründer macht: „[Mit] Ausnahme [des] Rendering-Kern[s].“ Hier wird deutlich, dass trotz der allgemeinen Strategie möglichst keine technischen Schulden im Alltag aufzunehmen, die die Unternehmen verfolgen, es Unterschiede zum Grad der technischen Schulden in den Codebasen gibt. Entweder wie hier nach Komponenten oder auch nach Art der technischen Schulden, wie Gründer A erklärt:

> „Wir achten am meisten auf coding debt, da ist die technische Schuld auch am geringsten. Testing debt ist ein bisschen größer und documentation debt ist am größten. Ich glaube da sind wir auch sehr typisch. Alle Sachen, die dem business value zuträglich sind, auf die achtet man mehr als die anderen Sachen.“ - A1

Dass bei Unternehmen A Dokumentation von weniger großem Belang ist, führt Entwickler A auf die überschaubare Größe des Entwicklerteams zurück.

> „Wir könnten definitiv bei der Dokumentation mehr machen, da sehen wir aber alle keinen besonders großen Gewinn, da wir ein kleines Team sind und wenn neue Programmierer kommen, können wir die auch ein bisschen an die Hand nehmen bei Fragen und deswegen nehmen wir das dann in Kauf, dass wir keine aussagekräftige Dokumentation im eigentlichen Sinne haben. [...] Wir achten sehr stark darauf, dass der Code verständlich ist, dass wir gute Methodennamen haben, gute Klassennamen etc. Das wir so eine Art selbstdokumentierenden Code haben.“ - A2

Hervorzuheben ist allerdings, dass man auch bei Unternehmen A solche Aussagen über den Grad der technischen Schulden nicht über die gesamte Codebasis pauschalisieren kann. Der Gründer ganz getreu seiner vorherigen Analyse: „Alle Sachen, die dem business value zuträglich sind, auf die achtet man mehr, als die anderen Sachen“, beschreibt im Gespräch, dass auf den komplexen Codeabschnitten ein deutlich größeres Augenmerk bzgl. Qualität und technischen Schulden gelegt wird als auf den trivialeren.

> „Die schwierigen Sachen machen wir besser bzgl. coding, testing, documentation als die einfachen. Es bringt nichts, wenn [man] bei allen Getter- und Setter-Methoden ran schreibt was sie tun, weil es keinen

> Mehrwert bringt. Aber wenn ich eine komplizierte, nebenläufige Methode habe [...] - auch wenn sie sehr kurz ist - dann hat die auch bei uns eine sehr dicke Dokumentation dazu, warum [die] so ist.“ - A1

In einem anderen Teil des Gespräches betont er die leicht schwankende Qualität der Codebasis, die davon abhängig ist, wie stark die Teile die weitere Softwareentwicklung, aber auch Performance beeinflussen:

> „Wir können bei uns die Architektur relativ gut unterteilen, was die Fundamente sind, also solche Sachen wie das Datenmodell oder die Architektur-Elemente, die für die Regression Tests verwendet werden und die sind sauber. Genauso wie die Komponenten, die auf Nebenläufigkeit setzten. [...] Wenn man an den Stellen anfängt herum zu schrauben, dann kommt man in Teufels Küche. Gleichzeitig kann ich genau sagen, alles, was zum Beispiel mit Textausgabe zu tun hat, da ist [es] Kraut und Rüben, weil wir das dann erst glatt ziehen, wenn es zu der entsprechenden Stelle kommt, dass es gebraucht wird.“ - A1

Hier muss angemerkt werden, dass das Level an technischen Schulden in der Codebasis von Unternehmen A generell wohl sehr gering ist. Die Umschreibung „Kraut und Rüben“ sollte also nicht allzu wörtlich verstanden werden.

Zusammenfassend werden aus den geführten Interviews folgende Thesen zur Vermeidung technischer Schulden in Software-Startup-Unternehmungen abgeleitet:

1. Langlebige / erfolgreiche Software-Startup-Unternehmungen versuchen die Aufnahme technischer Schulden im Alltag zu vermeiden.

2. Es gibt Unterschiede im Grad der technischen Schulden nach Komponenten und Art. Diese Unterschiede sind darin begründet, dass Schulden in weniger komplexen Teilen der Codebasis, deutlich weniger Probleme nach sich ziehen und das vor allem Dokumenentation für kleine Entwicklungsteams weniger wichtig ist, da jeder Entwickler mit weiten Teilen der Codebasis vertraut ist.

7.4 Aufnahme technischer Schulden nach Projektanforderungen und Einfluss der Geschäftsbeziehungen

Während wie oben beschrieben bei den untersuchten Unternehmen die Aufnahme von technischen Schulden im Alltag vermieden wird, werden zum Teil technische Schulden in den Unternehmen A und C aufgrund von Projektanforderungen in der Regel in Form von Lieferterminen aufgenommen. Dies ist vergleichbar mit der in Kapitel 7.1 beschriebenen bewussten Aufnahme von technischen Schulden zu Beginn einer Startup-Unternehmung aufgrund von Ressourcenknappheit.
Auf die Frage im Januar, ob auch Unternehmen A in Zukunft technische Schulden aufnehmen werde, antwortet der Gründer:

> „Das wahrscheinlichste Szenario, das wir im Zeitraum von Ihrer Arbeit neue technische Schulden machen, ist, dass wir in der Zeit ein Kundenprojekt akquirieren, das wir mit unserer aktuellen Architektur nicht ganz

> optimal abbilden können, das aber so attraktiv ist ansonsten, dass wir es trotzdem angehen wollen. [...] Es ist so ein bisschen aus der Erfahrung heraus, wenn der Kunde mit viel Geld wedelt, dann ist man eher dazu geneigt von seiner Seite aus zu investieren, um das zu kriegen.“ - A1

Diese Prognose sollte sich bestätigen. Entwickler A beschrieb im Follow-up Interview im Mai ein Projekt, bei dem das Entwicklerteam aufgrund eines Liefertermins unter sehr starkem Zeitdruck stand:

> „Wir [hatten] auch letzte Woche ein Projekt geliefert, dass extrem zeitaufwändig war. Da arbeitest du schon mal eine Woche lang 13-14 Stunden pro Tag dran.“ - A2

Bei diesem Projekt ließen sich die Anforderungen des Kunden nicht optimal und generisch mit der aktuellen Architektur abbilden, weshalb man sich dazu entschied die Funktionalität über ein Skript umzusetzen. Da es sich um eine einmalige Analyse für den Kunden handelte, galten zudem nicht so strenge Anforderungen an die Qualität und man nahm bewusst technische Schulden auf.

> „Also wir haben manchmal Kundenanfragen [wie bei diesem Projekt], wo wir die Ergebnisse nicht so optimal mit unserer geschriebenen KI erstellen können. Das kommt selten vor, aber wenn es vor kommt, dann versuchen wir diese Analyse mit Skripts anzureichern. [...] Weil das nur ein Ein-Mal-Projekt war, machen wir uns sozusagen nicht die große Mühe und probieren diesen generischen Fall zu implementieren in unsere KI, sondern erlauben uns zu sagen: ≪Okay, wir schreiben das jetzt mal als Python Skript zusätzlich noch.≫ Und das ist dann tatsächlich so, dass wir da definitiv nicht so sehr auf die Codequalität achten. Sondern einfach das Ergebnis liefern müssen.“ - A2

Während hier aufgrund der Projektanforderungen technische Schulden aufgenommen wurden, wird direkt mit dem Abbau dieser Schulden nach Ende des Kundenprojekts und folglich gesunkenem Druck auf das Entwicklerteam geplant, indem die Funktionalität des implementierten Skripts für einen generischen Fall und mit deutlich höheren Qualitätsansprüchen in die Kernkomponente des Softwaresystems integriert wird.

> „Meine Aufgabe wird dann diese Woche auch sein, dieses schnell geschriebene Skript, was überhaupt gar keine gute Qualität hat, eben bei uns in die Kernkomponente einzupflegen und da dann natürlich auf die Sauberkeit des Codes zu achten.“ - A2

Ein ähnliches Vorgehen, um Zeitpläne einzuhalten, konnte auch bei Unternehmen C beobachtet werden. Im konkreten Fall ergaben sich die festen Liefertermine trotz Bedienung des Endverbrauchermarktes in Form von Smartphone-Applikationen aus der Zusammenarbeit mit dem Betreiber der Verkaufsplattform. Hier mussten die Veröffentlichungsdaten neuer Applikationen oder größerer Funktionsupdates mit einem Vorlauf von 6 Wochen an den Betreiber kommuniziert werden, um so eine bessere Sichtbarkeit auf der Plattform zu erhalten.

> „Das heißt spätestens 6 Wochen vor einem Release hat man eigentlich wirklich eine harte Deadline, dass man dann diesen Release auch rausbringt. [...] Und das heißt dann durchaus, dass da ein bisschen Crunch passiert. Gerade am Anfang wo wir neu mit diesem Zyklus waren, ist es öfter passiert, dass wir uns da verschätzt haben und dass das Feature da schnell schnell implementiert werden musste. Wo wir dann gesagt haben: ≪Nein, wir machen es jetzt erst mal so, dass es geht und in den nächsten paar Versionen, die dann hinterher kommen, machen wir es hübsch.≫ Was wir dann auch meistens wirklich gemacht haben.“ - C2

Wie bereits in Kapitel 7.2 umfänglich diskutiert, nahm Unternehmen C zudem technische Schulden durch das Aufschieben des Refactorings des Softwarekerns auf, um schneller neue Funktionalität schaffen zu können.

Durch den Vergleich der Aussagen der Interviewpartner von Unternehmen A, welches ausschließlich Geschäftsbeziehung zu anderen Unternehmen unterhält, und der von Unternehmen C, welches hauptsächlich Applikationen für den Endverbraucher entwickelt und vertreibt, wird deutlich, dass die Geschäftsbeziehung eines Software-Startups - also b2b bzw. b2c - einen Einfluss auf den Druck, der auf dem Entwicklungsteams lastet, und folglich auch auf die bewusste Aufnahme von technischen Schulden hat. Bei A wird die Entwicklung dabei vor allem von Kundenprojekten getrieben. Der damit einhergehende Druck wird von Entwickler A als sinuskurvenartig beschrieben, da es zum einen Pausen zwischen den einzelnen Projekten gibt und zum anderen Projekte beispielsweise in der Ferienzeit im Sommer oder zu Weihnachten von Seiten der Kunden heruntergefahren werden.

> „Da [ist] ein hoher Druck, der einhergeht auch mit Phasen mit weniger Druck. Also wie eine Sinuskurve. [...] es gibt Wochen oder Phasen bei uns, wo wir hohen Druck haben und liefern müssen und wo man dann potentiell technische Schulden auch ein Stück weit ansammeln muss, um liefern zu können - mit Überstunden - dass dann später wieder abgebaut wird.“ - A2

Dass diese starke Ausprägung der Sinuskurve nicht inhärent für jedes Software entwickelnde b2b-Unternehmen ist, wird im zweiten Gespräch mit dem Entwickler deutlich. Hier weist er darauf hin, dass dieser starke Kontrast der einzelnen Phasen der Teamgröße und der Tatsache geschuldet ist, dass nicht fortlaufend neue Kundenprojekte begonnen werden, sobald das vorherige abgeschlossen wurde.

> „[Der Druck auf dem Entwicklungsteam] ist bei uns nach wie vor sinuskurvenartig, weil wir auch nicht immer 24/7 Projekte in der Pipeline haben für Kunden. Also wir haben jetzt nicht so eine große Pipeline oder Warteschlage, wo wir immer Kunden haben, denen wir zeitnah was liefern müssen. Das kann sich ändern, wenn wir wachsen. Dann müssen wir neue Leute einstellen. Aber deswegen ist es bei uns gerade sinuskurvenartig. Wenn wir eine größere Pipeline hätten, wir dauerhaft liefern müssen, würde sich diese Kurve auch strecken.“ - A2

Auf der anderen Seite hat ein b2c Unternehmen nicht diese starken projektbasierten und vertraglich bindenden Liefertermine. Natürlich teilen sich Startups unabhängig der Geschäftsbeziehung interne Liefertermine beispielsweise durch eine vordefinierte Roadmap oder externe Liefertermine für Investoren. Besonders zu Beginn einer Startup-Unternehmung - wie in 7.1 diskutiert - ist der Druck für Startup-Unternehmungen, die unter Ressourcenknappheit leiden, sehr hoch. Anders als das durch Projekte vorangetriebene b2b-Geschäft ist bei Unternehmen C ein Rückgang des Drucks mit wachsender Kundenzahl und Einnahmen zu beobachten, wofür folgende Aussage des Gründers charakteristisch steht:

> „Wir sind jetzt aber definitiv entspannter was die Release-Daten angeht als wir früher waren. Das muss man schon sagen. Deshalb können wir jetzt mehr darauf achten, uns mehr Zeit für die Konzeption von zum Beispiel Features, Architekturen [zu] lassen, als dass wir sagen: «Okay pass auf, wir haben jetzt einen Monat Zeit, gibt Gas.» Wenn aus dem [einen] Monat, zwei Monate werden, ist es jetzt gerade nicht kritisch. Von daher würde ich schon sagen, dass wir weniger technische Schulden jetzt aufnehmen als früher.“ - C1

Natürlich ist anzumerken, dass es neben der Geschäftsbeziehung viele weitere Faktoren gibt, die den Druck auf die Softwareentwicklung innerhalb einer Startup-Unternehmung und damit auch die Aufnahme bzw. den Abbau von technischen Schulden beeinflussen. Betrachtet man allerdings nur diesen Einflussfaktor, losgelöst von allen anderen, so ist der Druck auf die Softwareentwicklung für Unternehmen im b2b Bereich wohl eher sinuskurvenartig geprägt, wohingegen im b2c Umfeld eher ein langsamer, linearer Rückgang des Druckes über die Zeit anzunehmen ist.
Bei A gab es also immer wieder trotz Phasen mit sehr hohem Druck „Verschnaufpausen“, um technische Schulden abzubauen, während bei Unternehmen C sich die Lage, aufgrund des deutlich langsameren, aber stetigen Wachstums an Kunden und Umsätzen, deutlich langsamer entspannte. Für diese Abstraktion spricht auch die Tatsache, dass bei Unternehmen C die technischen Schulden am Softwarekern erst nach sehr langer Zeit abgebaut wurden, was dafür spricht, dass der Druck auf die Softwareentwicklung viel länger viel höher war, als bei Unternehmen A.
Natürlich sind in der Praxis wohl eher Mischformen anzutreffen. So kann auch auch ein b2c-Unternehmen regelmäßige Einkünfte beispielsweise in Form von Software-as-a-Service-Verträgen haben und generell ist zu vermuten, dass der Druck auf die Entwicklung mit Wachsen der Funktionalität und der damit einhergehenden erfüllbaren Kundenanforderungen, abnimmt.

Zusammenfassend werden aus den geführten Interviews folgende Thesen zu der Aufnahme technischer Schulden nach Projektanforderungen und Einfluss der Geschäftsbeziehungen aufgestellt:

1. In Phasen, in denen ein hoher Druck auf der Entwicklung, aufgrund von Projektanforderungen wie Lieferterminen, lastet, nehmen erfolgreiche Software-Startups bewusst technische Schulden auf, um diese in Zeiten von weniger hohem Druck abzubauen.

2. Der Wechsel dieser Phasen wird dabei u.a. von der Geschäftsbeziehung des Unternehmens beeinflusst. Während die Entwicklung des Drucks bei b2b-Startups eher einer Sinuskurve gleichen kann, ist es im b2c von einem langsamen, linearen Rückgang geprägt.

7.5 Qualitätsansprüche der Entwickler*innen

Eine Aussage zieht sich durch alle Interviews: Die Entwickler*innen in den befragten Unternehmen wollen qualitativ hochwertige Arbeit leisten. So beschreibt Gründer B wie sich Entwickler dagegen sträuben technische Schulden, wenn auch kurzfristig, aufzunehmen:

> „Und dann gibt es immer die Leute bei uns, die sagen: «Nein das kommt jetzt nicht rein, weil muss erst noch umgebaut werden.» Oder manchmal müssen andere Sachen im Framework noch umgebaut werden, dass es halt schön ist und die die Architektur passt. [...] Das ist schon so, dass die Leute selbst motiviert sind, was schönes, wartbares zu schaffen [und] jeder guckt, dass er es möglichst sauber macht und wenig Schuld in das System rein kommt. “ - B1

Und fügt später im Gespräch über von der Universität übernommene Entwickler an:

> „[Die Entwickler sind] auch teilweise idealistisch und die wollen nicht irgendwas zusammen hacken, sondern die sagen: «Das muss schön sein».“ - B1

So betont auch Entwickler A trotz vereinzelter Aufnahme von technischen Schulden aufgrund von Projektanforderungen:

> „Wir probieren eigentlich durch die Bank weg schon sauber zu arbeiten. [...] Wir wollen trotzdem sehr, sehr gute Software liefern und uns nicht die Blöße geben, dass die Sache dann abschmiert.“ - A2

Auch Gründer C reflektiert im Gespräch über die Qualitätsansprüche des gesamten Teams:

> „Wir sind alle technikaffine Personen. Wir sind alle auch eher Leute, die sehr strebsam sind, sehr engagiert sind im Großen und Ganzen und eher Qualität gegenüber Quantität bevorzugen - einfach vom menschlichen her. Es ist aber auch ein Charakterzug, den jeder im Team hat.“ - C1

Zum einen kann dieser Wille qualitativ hochwertigen Code zu schaffen - also wenig vermeidbare technische Schulden aufzunehmen - auf einem intrinsischen Qualitätsanspruch beruhen, zum anderen ist aber auch anzuführen, dass die Arbeit mit qualitativ schlechtem Code aufwändiger und weniger befriedigend sein kann, was durch ein bereits in Abschnitt 7.1 verwendetes Zitat deutlich wird:

> „Wenn es ein Feature gab, dass irgendwie mit dem Kern zu tun hatte, hatte ich von vornherein schon weniger Bock mich da dran zu setzen, weil da viel mehr Aufwand mit verbunden war, das dann anzupassen, weil ich

> mich da erst einmal rein arbeiten musste und eine andere Sprache nutzen. Und dich (1. Entwickler) dann erst fragen, wie du dir das gedacht hast damals und so.“ - C4

Der in Unternehmen A interviewte Entwickler hat zwar nahezu direkt nach dem Studium bei A angefangen, wo - wie oben geschildert - die Aufnahme von technischen Schulden sehr stark vermieden wird, und kennt folglich die professionelle Arbeit mit qualitativ schlechterem Code nicht direkt aus eigener Erfahrung, reflektiert allerdings auch über das Level an technischen Schulden im Software-System:

> „Es ist schon gut, dass so wenig technische Schulden da sind, weil das System einfach extrem komplex ist und wenn das schlecht aufgebaut wäre, wäre es wirklich ein Graus, sich da durchzuarbeiten.“ - A2

Er schildert daraufhin eine Anekdote einer bekannten Entwicklerin eines anderen Unternehmen, deren Arbeit aufgrund schlechter Codequalität deutlich demoralisierender ist, und resümiert mit dem Satz:

> „Von daher weiß ich aus anderer Erfahrung, dass das echt frustrierend ist, wenn der Code nicht wartbar und unsauber ist.“ - A2

Bei Unternehmen C konnten die positiven Effekte dieser oben beschriebenen intrinsischen Motivation eines direkt von der Universität übernommenen Entwicklers besonders gut beobachtet werden:

> „Was [Entwickler C4] ganz viel gemacht hat, als er rein gekommen ist, ist natürlich sich einarbeiten und durch dieses Einarbeiten sind Stellen aufgefallen, wo man längst hätte nachbessern müssen. Und insofern ist [Entwickler C4] quasi [damit] eingestiegen, technische Schulden beseitigen.“ - C3

Oder wie der damals neu dazugekommene Entwickler C4 seine Motivation mit eigenen Worten beschreibt:

> „Und da ist mir aufgefallen, Moment mal, dass geht doch irgendwie noch anders. Und [...] weil ich mich eh gerade am Einarbeiten war, hatte ich gleich viel mehr Motivation das anzupassen, als wenn man schon eine ≪Never touch an runing system≫ Attitüde hat, dann fällt es halt schwer da nochmal dran zu gehen. Aber ich kannte das System halt eh noch nicht, deswegen fiel es mir in dem Fall leichter.“ - C4

Als Neuzugang habe man - anders als die anderen Entwickler*innen - zudem keine volle Liste an Aufgaben, die abgearbeitet werden müssen, und folglich auch die Zeit solche technischen Schulden, die man entdeckt, auszubessern. Damit dies in diesem Fall dann aber auch geschehen konnte, bedurfte es neben diesem Eigenengagement des Entwicklers auch einer offenen Kultur innerhalb des Entwicklerteams, wie es der CTO (auch Entwickler) bei C beschreibt:

> „Wenn ich in eine Firma kommen würde, erst recht wenn sie wahrscheinlich größer wäre und da erst mal vor eine bestehende Codebasis gesetzt werde, hätte ich Angst alles kaputt zu machen oder Leuten irgendwie rein zu grätschen. Aber ich glaube, da haben wir [Entwickler C4] auch möglichst den Eindruck gegeben, dass er das machen kann und dass wir nichts heilig finden, sondern eher im Gegenteil lieber schön aufgeräumten Code haben wollen.“ - C2

Auf der anderen Seite können technische Schulden allerdings auch eine negative Strahlkraft haben, die der intrinsischen Motivation der Entwickler*innen qualitativ hochwertigen Code zu schreiben entgegen stehen kann.

> „Wir machen relativ wenig Out-Sourcing, weil wir die Abwägung, die wir mit technischen Schulden eingehen, nicht gut nach Außen transportieren können. Da wäre der Kommunikations-Over-Head zu groß. Außerdem fordert es so zu arbeiten auch eine gewisse Stabilität im Team, um das machen zu können, dass jeder weißt was die Spielregeln sind. Es ist sehr sehr leicht, dass Missverständnisse entstehen, dass jemand sagt: ≪Hier bei der Firma kann ich mittelmäßigen Code abliefern, weil das okay ist oder weil Teile der Codebasis so aussehen.≫“ - A1

Zudem ist anzumerken, dass die intrinsischen Qualitätsansprüche zwischen den Entwickler*innen untereinander oder auch zu den Entscheider*innen variieren können, sodass gegebenenfalls auch bei zu hohen Qualitätsansprüchen von Seiten der Entscheider*innen Frustration bei einzelnen Entwickler*innen entstehen kann.

Zusammenfassend werden aus den geführten Interviews folgende Thesen zu den Qualitätsansprüchen der Entwickler*innen abgeleitet:

1. Gut ausgebildete Entwickler*innen haben in der Regel eine intrinsische Motivation qualitativ hochwertigen Code zu schreiben und wollen möglichst keine technischen Schulden ansammeln. (Es wird allerdings vermutet, dass dies vor allem für Code- und Architekturqualität gilt, evtl. etwas weniger stark bei Dokumentation und Softwaretests.)

2. Die Aufnahme von technischen Schulden kann zu Frust bei den Entwickler*innen und evtl. auch zu einer Absenkung der eigenen intrinsischen Qualitätsstandards führen.

7.6 Auswirkungen der technischen Expertise der Entscheider*innen

Dass die technische Expertise der Entscheider*innen vermutlich eine große Rolle im Umgang mit technischen Schulden bei den befragten Unternehmen spielt, geht aus den reflektierten Aussagen der Gründer und der wohlüberlegten Aufnahme von technischen Schulden, sowie der Vermeidung von technischen Schulden im Alltag der Unternehmen hervor. Dadurch, dass allerdings kein Startup befragt wurde, bei dem es den Gründer*innen und Entscheider*innen an der technischen Expertise mangelt,

fehlt im Rahmen dieser Arbeit die Gegenperspektive, die einen Vergleich und damit das Ableiten von fundierten Thesen ermöglichen würde. Nichtsdestotrotz werden im Folgenden eine Reihe von Aussagen und Beobachtungen zu der Thematik zusammengetragen.

So wird an folgendem Beispiel deutlich, dass technische Schulden - hier in der Architektur - zu zusätzlichen Aufwänden in der Entwicklung führen, die womöglich für technische Laien und anhand der Anforderungen nur sehr schwer vorhersehbar sind. Im konkreten Fall beschreiben die Entwickler von Unternehmen C, wie das Hinzufügen eines neuen Knopfes in der GUI einer Applikation sich als deutlich komplexer herausstellte als es in der Vergangenheit der Fall war, weil dieser Button, aufgrund der damit verbundenen Funktionalität, sich konzeptionell von den anderen unterschied. Einer der Entwickler beschreibt daraufhin das Gespräch mit der Product Ownerin über die Aufwandsschätzung.

> „Das ist auch so eine witzige Kurve [der Aufwandsschätzungen]: Man fügt immer [neue] Features hinzu und irgendwann muss man dann seinem Product Owner sagen: «Um den einen Knopf hier hinzu zu fügen, brauchen wir jetzt drei Wochen, weil wir müssen unter der Haube noch das und das und das machen.» Und dann merkt auch irgendwann der Product Owner, dass es total sinnvoll ist auch irgendwann eine Pause einzulegen und mal was zu refactorn. Es ist nicht so, als hätten wir diesen krassen Druck von oben, aber es ist halt ein bisschen [so], [dass] man merkt, dass das Verständnis teilweise nicht hundertprozentig da ist.“ - C2

Ein anderer Entwickler fügt direkt an, dass die Product Ownerin früher und nach erfolgreichem technischen Studium selber als Entwickler tätig war und dass dadurch die Kommunikation erheblich erleichtert wird.

> „Aber es hilft definitiv auch in der Kommunikation, wenn alle Seiten auch mal den Schmerz des Anderen gespürt haben. [...] Da es eben nicht mal alles schnell gemacht ist, sondern dass da ganz viel der Teufel im Detail steckt, und es hilft definitiv bei solchen Entscheidungsprozessen, wenn [man] dann eben weiß, dass das Einfügen eines Knopfes an der Stelle eben halt nicht nur mal: «If, dann Knopf hier ist», sondern dass da ganz viel dran hängt.“ - C3

Gründer C erklärt zudem, dass er im Vergleich zum zweiten Geschäftsführer, welchem der technische Hintergrund etwas fehle, einfacher verstünde, warum technische Schulden beispielsweise in der Architektur ein Risiko darstellen. In Konsequenz käme er deshalb den Entwicklern sehr oft in Entscheidungen entgegen und räume ihnen mehr Zeit für das Beseitigen von technischen Schulden ein.

> „Und bin ziemlich Informatik-freundlich. Informatik-freundlich, weil ich auch diesen technischen Hintergrund habe. [Ich] weiß was das Aufschieben von Refactoring kosten kann [und] wie hart es ist gegen Architekturen zu implementieren. Der zweite Geschäftsführer, der hat auch ausrei-

> chend Erfahrung in IT-Projekte, hat aber nicht den starken Informatik-Hintergrund. Also wenn du dem sagst: «Pass auf, [das] Refactoring brauchen wir, weil wir dann schneller sind.» Ist [das] bei ihm höchstwahrscheinlich nicht so ein großes Argument.“ - C1

Zusammenfassend hat in Unternehmen C nach den Aussagen der Befragten vor allem die Kommunikation rund um das Thema der technischen Schulden von der Expertise der Entscheider*innen profitiert. Durch das Verständnis der Problematik wurde dem Entwicklungsteam vor allem im späteren Verlauf der Unternehmung und vermutlich auch mit steigender (finanzieller) Sicherheit und Planbarkeit oftmals die Zeit für das Beseitigen von technischen Schulden eingeräumt. Hierfür ist die technische Expertise der Entscheider*innen allerdings nicht alleinig ausschlaggebend, sondern zudem die gute Firmenkultur und das gegenseitige Vertrauen u.a. in die fachliche Einschätzung der Entwickler, wann mehr Zeit für das Beseitigen von technischen Schulden eingeräumt werden sollte.

Bei Unternehmen A ist das Bild ein leicht anderes. Hier ist der interviewte Gründer aktiver in die Entwicklung eingebunden. Code-Anpassungen der Entwickler werden in der Regel von ihm persönlich freigegeben. Dies führt dazu, dass er einen direkteren Einfluss auf die Entwicklung hat und die Aufnahme technischer Schulden im Alltag ganz gezielt durch die aufgestellten Qualitätsstandards vermieden wird.

> „Beide [Gründer] haben promoviert in der Richtung Informatik. Insofern haben da beide Personen - beide Gründer - aufgrund dieser sehr sehr guten technischen Expertise ein sehr großes Augenmerk drauf. Das heißt wir sind uns da sehr sehr bewusst und achten tatsächlich ganz ganz stark darauf, möglichst keine technischen Schulden anzusammeln und achten sehr stark darauf, dass der Code wartbar bleibt und eine hohe Qualität hat.“ - A2

Bemerkenswert ist, dass während der befragte Entwickler ganz eindeutig davon berichtet, wie aufgrund der technischen Expertise der Gründer und deren direkten Einbindung in die Entwicklung sehr hohe Anforderungen an die Qualität des Codes gestellt werden, Gründer A hingegen resümiert, dass gerade durch die technische Expertise des Gründerteams die Aufnahme technischer Schulden beherrschbarer wäre.

> „Und wir sind ein technisches Gründerteam, das heißt für uns sind technische Schulden beherrschbarer als finanzielle Schulden. Wenn man dann seinen Verschuldungsgrad oder sein Fremdkapital-Grad managt, dann wird man sich die Frage stellen: «Inwiefern kann ich das?» Und wir sagen, wir können technische Schulden im Zweifelsfall besser managen als ein starkes BWL-Team. Und ein BWL-Team wird wahrscheinlich sagen, wir können unseren Fremdfinanzierungsgrad wahrscheinlich besser managen als die technischen Schulden. Daraus resultiert dann auch was man eher macht.“ - A1

Worin diese Dissonanz begründet liegt, ließ sich nicht abschließend klären. Es wird vermutet, dass der Gründer eine etwas breitere Definition von technischen Schulden

als der Entwickler zugrunde legt. Des Weiteren könnte der Blick des Gründers mehr auf strategische Entscheidungen gerichtet sein, wohingegen der Entwickler mehr über das Tagesgeschehen reflektiert.

Rekapitulierend werden folgende Beobachtungen - nicht Thesen - zu dem Einfluss der technischen Expertise der Entscheider*innen festgehalten:

1. Technische Expertise bei den Entscheider*innen kann die Kommunikation mit den Entwicklern rund um das Thema der technischen Schulden verbessern und dazu führen, dass man diesen mehr Zeit für das Beseitigen von technischen Schulden einräumt.

2. Gründer*innen mit technischer Expertise können eine aktivere Rolle in der Entwicklung einnehmen und so die eigenen Qualitätsansprüche oder strategischen Entscheidungen zum Management von technischen Schulden durchsetzen.

3. Entscheider*innen mit technischer Expertise haben mitunter hohe Qualitätsstandards.

4. Entscheider*innen mit technischer Expertise können die zukünftigen Kosten von technischen Schulden besser einschätzen und folglich kalkulierter technische Schulden in weniger kritischen Bereichen (unterschiedliche Teile der Codebasis oder unterschiedliche Arten von technischen Schulden) aufnehmen.

8 Diskussion

8.1 Diskussion der abgeleiteten Thesen

Im Folgenden wird die Validität der in Kapitel 7 aufgestellten Thesen anhand der erhobenen Daten diskutiert und mit den Ergebnissen der verwandten Forschungsarbeiten verglichen.

8.1.1 Ansammlung technischen Schulden zu Beginn einer Startup-Unternehmung

1.1 Software-Startups, die zu Beginn unter enormen Druck aufgrund von Ressourcenknappheit stehen, müssen technische Schulden aufnehmen, um schnell Funktionalität schaffen zu können.

Diese These beruht auf den Aussagen des Gründers und der Entwickler in Unternehmen C, wohingegen bei Unternehmen B - aufgrund der langen an der Universität betriebenen Forschung und Entwicklung - ein deutlich geringerer Druck nach neuer Funktionalität herrschte. Bei A wurden zu Beginn viele Prototypen entwickelt, die zuerst architekturell und später mehr in Bezug auf die Nutzerinteraktion geprägt waren. Hier herrschte im direkten Vergleich zu C also eine umsichtigere Aufnahme von technischen Schulden, vermutlich weil man sich - anders als bei C - eine längere

Entwicklung leisten konnte. Hier fehlen allerdings konkrete Aussagen von Unternehmen A, um diese Vermutung zu belegen.
Es wird vermutet, dass neben der Ressourcenknappheit bei C auch die Unsicherheit in Bezug auf den wirtschaftlichen Erfolg der Startup-Unternehmung - u.a. aufgrund der b2c-Geschäftsbeziehung - ein Einflussfaktor bei der bewussten Aufnahme von technischen Schulden war, wie sie auch Besker et al. beschreiben[7]. Die Ressourcenknappheit einer Startup-Unternehmung wird im Rahmen dieser Forschung allerdings mehr als notwendige Bedingung für die bewusste Aufnahme von technischen Schulden verstanden, wohingegen die Unsicherheit eher eine hinreichende Bedingung darstellt. Hierbei handelt es sich allerdings nur um eine Vermutung, die es in zukünftiger Forschung zu validieren gilt.
Darüber hinaus hat die aufgestellte These Überschneidungen mit den Ergebnissen von Cico et al., die „accepting TD“ und „ignoring TD“ als Strategien im Umgang mit technischen Schulden definieren, welche vor allem in den frühen Phasen einer Startup-Unternehmung beobachtetet wurden (siehe 4.1 Forschung zu technischen Schulden in Startups)[10].

1.2 Technische Schulden, die in den frühen Phasen der Softwareentwicklung aufgenommen werden, bergen die Gefahr, vor allem wenn sie architektureller Natur sind, die Entwicklung langfristig, negativ zu beeinflussen.

Auch diese These fußt zu weiten Teilen auf den Erfahrungen der Entwickler und Abwägungen bei Unternehmen C bzgl. der technischen Schulden am Softwarekern, der die Funktionen zum Rendering bereitstellt, auf der alle entwickelten Applikationen basieren. Auch Gründer A untermauert, dass Schulden in Kernkomponenten zu erheblich mehr Aufwänden führen können und dass deshalb vor allem von Anfang an ein großes Augenmerk darauf lag, keine technischen Schulden aufzunehmen u.a. durch den Einsatz des rapid iterative prototypings. Hinzu kommen die Erfahrungen von Gründer B, welcher feststellte, das das System aufgrund zu vieler angehäufter Schulden zu einem frühen Zeitpunkt der Entwicklung gar nicht mehr funktionierte. Die These ergibt sich zudem aus der Beobachtung, dass die frühen Phasen der Softwareentwicklung in Technologie-Startups vor allem von Design- bzw. Architekturentscheidungen geprägt sind und die dort gelegten Grundlagen - ob guter oder schlechter Natur - die Entwicklung aller weiteren darauf aufgebauten Komponenten beeinflusst. Folglich bewegt sich die Abwägung von Startups insbesondere jenen, die unter starker Ressourcenknappheit leiden, ob zu Beginn der Unternehmung technische Schulden aufgenommen werden sollten, in einem Spannungsfeld zwischen dem schnelleren Implementieren von Funktionalität zum Generieren von Einkommen (oder Investitionen) auf der einen Seite und der Gewährleistung einer sauberen, architekturellen Grundlage, die die weitere Softwareentwicklung erheblich vereinfachen kann, auf der anderen Seite.

1.3 Die Arbeit mit innovativen Software-Technologien führt zwangsläufig zur unbewussten Aufnahme von technischen Schulden aufgrund von unklaren Anforderungen und daraus resultierenden suboptimalen Designentscheidungen.

Hierbei handelt es sich um keine neue oder besonders überraschende These. Wie bereits im korrespondieren Ergebniskapitel 7.1 angerissen, beschreibt bereits Cun-

ningham bei Einführung der Methapher, dass sich die Aufnahme von technischen Schulden beim Entwickeln einer neuer Software gar nicht vermeiden ließe. Dass bei Software-Startups, die mit neuen Technologien auf dem aktuellen Stand der Forschung ein innovatives Produkt auf den Markt bringen wollen, diese unbewusst, aufgenommenen technischen Schulden vermutlich höher sind als bei einer eher klassischen Softwareentwicklung, ist eine Vermutung für deren Beleg im Rahmen dieser Arbeit zwar der Vergleich mit einer solchen klassischen Entwicklung fehlt, allerdings liefern die Entwicklungsgeschichten der befragten Unternehmen erste Indizien für diese These. So stellte sich die Auswahl der Programmiersprachen und Frameworks bei A trotz des rapid iterative prototyping Ansatzes später als suboptimal heraus. Der Fall bei C dient als Beispiel dafür, wie die Anforderungen an die Architektur und den Software-Kern, der für das Rendering zuständig war, zu Beginn noch deutlich wenig klar waren, als im Vergleich zu dem jetzigen Stand 8 Jahre später. Folglich wurden technische Schulden zu Beginn aufgenommen, die durch mehrere Refactorings über die Jahre abgebaut wurden. Auch Gründer B sprach davon, dass viel technische Schuld zu Beginn unbewusster Natur war. Diese überlappenden Erfahrungen in allen 3 Unternehmen sprechen für die aufgestellte These. In wie weit Software-Startups, die mit neuen Technologien arbeiten, davon mehr betroffen sind als die Softwareentwicklung in anderen Unternehmen muss durch weitere Forschung geklärt werden.

1.4 Kleinen, neu gebildeten Entwicklungsteams kann es zu Beginn an klaren Qualitätsrichtlinien und Kontrollstrukturen fehlen. Dies gilt besonders für 1-Mann-Projekte. Dadurch werden zusätzliche unbewusste technische Schulden aufgenommen. Kommen neue Entwickler hinzu, werden diese Schulden und evtl. daraus resultierende Probleme oder zeitliche Mehraufwände deutlich sichtbar.

Diese Aussage stützt sich auf den Erfahrungen der Unternehmen B und C, deren Entwicklung zu Beginn von einem Entwickler verantwortet wurde. Bei B wurden die damit verbundenen Probleme nach den Aussagen des Gründers besonders deutlich, als mehrere Parteien in Form von Abschlussarbeiten an Teilen der Codebasis weiterarbeiteten. Fehlende Entwicklungsprozesse zur Qualitätssicherung wie beispielsweise das Freigeben von „Merge Requests“ durch ein N-Augen-Prinzip, die bei B zu großen Problemen führten, wurden auch von Besker et al. als ein Einflussfaktor in Bezug auf das Level an technischen Schulden in Startup-Unternehmungen identifiziert und die ähnlich zu den Beobachtungen bei B und C feststellen, dass durch ein Wachstum des Entwicklerteams das Level an technischen Schulden absinkt, was sie darauf zurückführen, dass neue Entwickler deutlich schwerer von negativen Folgen von technischen Schulden betroffen seien[7]. Die Softwareentwicklung bei A wurde hingegen durch die Gründer bereits von Beginn stark kontrolliert, was zeigt, dass diese These nicht auf alle neu gegründeten Startups und Entwicklerteams übertragbar ist.
Das Sichtbarwerden der technischen Schulden durch das Anwachsen der Entwicklerteams in den Unternehmen B und C, welches im zweiten Teil der obigen These aufgegriffen wird, ist ein erster Indiz für die durch Klotins et al. formulierte Hypothese:

> „The number of people in a team amplifies precedents for technical debt.“[34]

Zusammenfassend zeigen die Beobachtungen und daraus abgeleiteten Thesen, dass die Aufnahme von technischen Schulden zu Beginn einer Startup-Unternehmung sehr typisch ist. Ressourcenknappheit und Unsicherheit konnten als Einflussfaktoren bei der bewussten Aufnahme von technischen Schulden identifiziert werden. Hingegen bleibt unklar, warum Unternehmen A die bewusste Aufnahme von technischen Schulden so stark vermeiden konnte. Mögliche Gründe könnten eine längerfristig gesicherte Entwicklung und die hohe technische und organisatorische Expertise des Gründerteams sein und das daraus resultierende Wissen über die Mehrkosten von technischen Schulden vor allem in der zugrunde liegenden Architektur.
Die 4 hier aufgestellten Thesen vertiefen die Ergebnisse verwandter Arbeiten. Es konnten keine Kontradiktionen innerhalb der im Rahmen dieser Forschung erhobenen Daten oder zu anderen Forschungsergebnissen festgestellt werden und insgesamt handelt es sich um Thesen und Beobachtungen, die sich gut in die Erkenntnisse rund um technische Schulden einfügen.

8.1.2 Refactoring als wirtschaftlich bedeutende Entscheidung

2.1 Technische Schulden vor allem architektureller Natur am Software-Kern führen zu höheren Entwicklungsaufwänden, Frustration und zur Aufnahme weiteren technischen Schulen und sollten durch ein Refactoring abgebaut werden.
2.2 Da durch ein Refactoring keine neue Funktionalität geschaffen wird und die Startups oft unter Druck von Seiten der Kunden oder Investoren stehen, ist es schwierig, einen geeigneten Zeitpunkt für ein Refactoring zu finden und muss sich mitunter durch Vorarbeiten in Abhängigkeit vom Umfang des zu überarbeitenden Codes erkämpft werden.
2.3 Durch das Aufschieben eines Refactorings können in der Zwischenzeit aber auch neue Anforderungen bekannt werden oder ins Bewusstsein rücken und es erlaubt mehr Erfahrungen mit den eingesetzten Technologien zu sammeln, was teilweise die unbewusste Aufnahme technischer Schulden durch das Refactoring vermeiden kann.
2.4 Der Abbau von technischen Schulden steht in einem Spannungsfeld zwischen dem Druck nach Funktionalität durch Projekte, Kunden und Investoren und dem stetigen Zuwachs des Wissens über Anforderungen und die Technologie auf der einen Seite und Zufriedenheit der Entwickler*innen, höhere Entwicklungsaufwände und weiterer Anhäufung technischer Schulden auf der anderen Seite.

Diese Thesen beruhen alle größtenteils auf den Beobachtungen bei Unternehmen C und werden deshalb zusammenfassend diskutiert. Hier wurden wie oben beschrieben zu Beginn der Entwicklung viele technische Schulden aufgenommen. Hinzu kam das Wissensmonopol des ersten Entwicklers. Dies in Kombination mit schlechter Dokumentation führte dazu, dass die anderen beiden Entwickler nicht effektiv an dem Code des Rendering-Kerns arbeiten konnten. Da dieser die Grundlage für alle entwickelten und kommerziell vertriebenen Applikationen war und das eigentlich nötige Refactoring sehr umfangreich sein müsste, entschied man sich stattdessen lange gegen die Architektur zu arbeiten also weitere technische Schulden aufzunehmen, um so weiter Funktionalität liefern zu können. Die Beobachtungen zu den Gefahren

von architekturellen Schulden decken sich außerdem mit den Ergebnissen von Ernst et al., die mittels Umfragen und Interviews zu dem Ergebnis kommen, dass die meisten technischen Schulden nach Einschätzung der Befragten durch architekturelle Entscheidungen entstehen[17]. Auf der anderen Seite berichten die Entwickler bei C davon, dass es ihnen durch die über Jahre gesammelte Erfahrung und das Bewusstsein über die Anforderungen an den Software-Kern leicht fiel eine für die Zukunft geeignete Architektur zu konzipieren und zu implementieren. Daraus wird abgeleitet, dass das Aufschieben eines Refactorings vor allem für Startups trotz der dadurch bedingten zusätzlichen Aufnahme von technischen Schulden, neben den betriebswirtschaftlichen Vorteilen auch mit technischen Vorteilen - begründet auf einem besseren Verständnis der (technischen) Anforderungen - einhergehen kann.
Insgesamt ist allerdings festzuhalten, dass diese Thesen alleinig auf der Entwicklungsgeschichte bei C fußen. Während zwar in der Auswertung der Gespräche mit A und B keine Evidenzen gefunden wurden, die den Thesen widersprüchlich entgegenstehen, so ist zu konstatieren, dass diese Thesen nicht auf alle Software-Startups übertragbar sind.

8.1.3 Vermeidung technischer Schulden

3.1 Langlebige / erfolgreiche Software-Startup-Unternehmungen versuchen die Aufnahme technischer Schulden im Alltag zu vermeiden.

Dies ist eine Beobachtung, die in allen drei untersuchten Unternehmen gemacht wurde. Die Aufnahme technischer Schulden im Alltag - also losgelöst von Projektanforderungen etc. - bringt wenig überraschend keinen kurzfristigen Mehrwert und ist folglich mit den langfristigen Kosten verbunden. Die Kosten von technischen Schulden sind zum einen die erhöhten Entwicklungsaufwände, zum anderen aber auch die leidende Zufriedenheit der Entwickler aufgrund von frustrierender Arbeit. Wie streng diese Vermeidung von technischen Schulden im Alltag verfolgt wird variiert aber. Während bei A der Gründer einen sehr hohen Standard an die Codequalität durch persönliche Reviews durchsetzt, wird bei Startup C eher ein Release-Termin verschoben, um den Entwicklern mehr Zeit einzuräumen.
Wie auch Klotins et al. schreiben stellen zu hohe technische Schulden ein Risiko für Startup-Unternehmen dar und und behindern zukünftige Erweiterungen der Software[34], weshalb es wenig überraschend ist, dass diese langlebigen Startups versuchen die Anhäufung technischer Schulden im Alltag zu vermeiden.

3.2 Es gibt Unterschiede im Grad der technischen Schulden nach Komponenten und Art. Diese Unterschiede sind darin begründet, dass Schulden in weniger komplexen Teilen der Codebasis, deutlich weniger Probleme nach sich ziehen und das vor allem Dokumenentation für kleine Entwicklungsteams weniger wichtig ist, da jeder Entwickler mit weiten Teilen der Codebasis vertraut ist.

Mit den Befragten aus Unternehmen A und C wurde im Detail über gewisse Komponenten der Codebasis und dem korrespondierenden Grad an technischen Schulden gesprochen. Hier wurde bei beiden Unternehmen deutlich, dass große Unterschiede

sowohl zwischen den Komponenten herrscht. So war der Rendering-Kern von Unternehmen C nicht ausreichend genug dokumentiert und Funktionalität wurde entgegen der Architektur implementiert. Auch Gründer A bezeichnete beispielsweise den Code zur Textausgabe als „Kraut und Rüben“ und begründete die Unterschiede im Grad der technischen Schulden nach Komponente und Art mit den eindrücklichen Worten:

> „Alle Sachen, die dem business value zuträglich sind, auf die achtet man mehr als die anderen Sachen.“ - A1

Diese Aussage bildet den Kern der hier formulierten These. Im Kontext eines Software-Startup, das unter Ressourcenknappheit leidet, muss priorisiert werden. Vor allem Dokomentation ist für ein kleines, eingespieltes Entwicklerteam von geringem Interesse, trägt nicht zu verkürzten Entwicklungszeiten bei bzw. treibt diese auch nicht in die Höhe und wird in Konsequenz mit Blick auf neu zu entwickelnde Funktionalität oder den Abbau von anderen technischen Schulden vernachlässigt.

Da sich die hier forumlierten Thesen auf breiten Beobachtungen in den Unternehmen A, B und C stützen und sich gut in das generelle Verständnis von technischen Schulden einfügen, wird erwartet, dass diese auf andere Software-Startups übertragbar sind.

8.1.4 Aufnahme technischer Schulden nach Projektanforderungen und Einfluss der Geschäftsbeziehungen

4.1 In Phasen, in denen ein hoher Druck auf der Entwicklung, aufgrund von Projektanforderungen wie Lieferterminen, lastet, nehmen erfolgreiche Software-Startups bewusst technische Schulden auf, um diese in Zeiten von weniger hohem Druck abzubauen.

Die Aufnahme technischer Schulden wurde in den Unternehmen A und C (es fehlen konkrete Informationen zu B) nach Aussagen der Befragten dazu genutzt, um in der Vergangenheit schneller Funktionalität zu schaffen und so interne oder externe Liefertermine einzuhalten. Dies ist wenig überraschend und deckt sich mit den initialen von Cunningham gemachten Beobachtungen[13], weshalb diese These als gut übertragbar auf andere Startups eingeschätzt wird.

4.2 Der Wechsel dieser Phasen wird dabei u.a. von der Geschäftsbeziehung des Unternehmens beeinflusst. Während die Entwicklung des Drucks bei b2b-Startups eher einer Sinuskurve gleichen kann, ist es im b2c von einem langsamen, linearen Rückgang geprägt.

Die bewusste Aufnahme von technischen Schulden ist oftmals auf betriebswirtschaftliche Anforderungen zurückzuführen[40]. Folglich erscheint es logisch, dass auch die Art des Geschäftsmodells und die Kundenbeziehungen einen Einfluss auf die bewusste Aufnahmen von technischen Schulden haben könnten. Der Vergleich der Aussagen der Befragten in den Unternehmen A und C lässt darauf schließen, dass die Kundenbeziehung einen Einfluss auf deren Umgang mit technischen Schulden hat. Entwickler A beschreibt den Druck, der auf dem Entwicklerteam in dem b2b-Unternehmen

lastet und der zum Teil in der Aufnahme von technischen Schulden resultiert als sinuskurvenartig, in Abhängigkeit von Kundenprojekten und Lieferterminen. Dagegen wird aus den Aussagen bei C abgeleitet, dass mit der Zeit der Druck auf die Entwicklung abgenommen hat, weil sich das b2c-Startup zunehmend am Markt etabliert hat. All das deutet darauf hin, dass die Kundenbeziehung zwar mit Nichten der einzige oder größte Einflussfaktor in Bezug auf die bewusste Aufnahme von technischen Schulden in Startups ist, aber ein potentieller.
Auch Iuliia identifiziert in ihrer Fallstudie in 3 größeren Softwareunternehmen die Kundenbeziehung als Einflussfaktor im Umgang mit technischen Schulden und stellt fest:

> „Based on the company's business model, it was found out that [...] B2C companies (company A) are more willing to pay off technical debt than B2B and B2G software development companies (companies B and C).“[30]

Begründet wird das im Kontext ihrer Arbeit damit, dass b2b-Unternehmen die Implementierung von vom Kunden gewünschter Funktionalität dem Abbau von technischen Schulden vorzögen. Außerdem müssen diese Unternehmen oftmals die Zuverlässigkeit der von Ihnen entwickelten Systeme garantieren und seien deshalb sehr scheu Änderungen an der Software vorzunehmen.
Diese starke Priorisierung von Funktionalität in b2b-Unternehmen wurde auch im Rahmen dieser Arbeit beobachtet. So erklärt Gründer A (b2b):

> „Es ist so ein bisschen aus der Erfahrung heraus, wenn der Kunde mit viel Geld wedelt, dann ist man eher dazu geneigt von seiner Seite aus [durch die Aufnahme von technischen Schulden] zu investieren, um das zu kriegen.“ - A1

Im Kontrast wurde bei C vor allem mit voranschreitender Etablierung am Markt die Vermeidung technischer Schulden stärker priorisiert als das Festhalten von Lieferterminen. Eine mögliche Ursache dafür könnte sein, dass bei b2b-Startups das Einhalten von Lieferterminen viel direkter an den Umsatz geknüpft ist als das bei b2c-Startups der Fall ist.
Natürlich nehmen auch b2c-Startups - wie bei C beobachtet - technische Schulden auf um schneller Funktionalität zu schaffen. Wird dadurch allerdings nur der Endverbrauchermarkt bedient, so wird zwar auch hier die Funktionalität in der Regel geschaffen, um den Umsatz zu erhöhen, ist aber nicht direkt daran gekoppelt, wie bei einem b2b-Unternehmen, welches gemeinhin auf Grundlage eines Vertrages die Leistung erbringt.
Das projektbasierte Arbeiten eines b2b-Startups und die damit einhergehenden regelmäßigen Liefertermine können zur punktuellen Aufnahme von technischen Schulden führen. Um die Entwicklung nicht langfristig zu gefährden, sollten diese Schulden nach Erfüllung der Projektanforderungen wieder abgebaut werden. Aus diesem Grund wird argumentiert, dass das Level an technischen Schulden über die Zeit im Kontext eines b2b-Startups einer Sinuskurve ähnelt.
Durch die zunehmende Etablierung am Markt nimmt bei einem b2c-Startup der Druck auf die Entwicklung langsam ab, was es ermöglicht technische Schulden abzubauen, wie bei C beobachtet.

Anzumerken ist, dass es sich bei dieser These um eine Abstraktion - entkoppelt von anderen das Level an technischen Schulden beeinflussenden Faktoren - handelt. Obwohl die Grundlage für diese These lediglich auf der Gegenüberstellung zweier Unternehmen beruht und die darauf aufbauende Abstraktion zum Anwachsen / Abbaus an technischen Schulden durchaus diskutabel ist, so erscheint die Argumentation im betriebswirtschaftlichen Kontext schlüssig.

8.1.5 Qualitätsansprüche der Entwickler*innen

5.1 Gut ausgebildete Entwickler*innen haben in der Regel eine intrinsische Motivation qualitativ hochwertigen Code zu schreiben und wollen möglichst keine technischen Schulden ansammeln. (Es wird allerdings vermutet, dass dies vor allem für Code- und Architekturqualität gilt, evtl. etwas weniger stark bei Dokumentation und Softwaretests.)
5.2 Die Aufnahme von technischen Schulden kann zu Frust bei den Entwickler*innen und evtl. auch zu einer Absenkung der eigenen intrinsischen Qualitätsstandards führen.

Bei dieser These handelt es sich natürlich um eine Pauschalisierung. Die Tatsache, dass sich diese Aussage allerdings durch alle Interviews zieht, verdeutlicht, dass es zumindest in diesen langlebigen Startups kein Zufall ist, dass die dort arbeitenden Entwickler eine intrinsische Motivation haben qualitativ hochwertige Code zu schreiben. Die hier aufgestellte These deckt sich zum Teil mit den Ergebnissen der Meta-Studie von Beecham et al.[5]. Diese führen die Zufriedenheit mit der Arbeit als den in der Literatur am meisten genannten Motivator für Entwickler*innen an und listen als Teilaspekt dessen u.a., dass für die Entwickler der Zweck einer Aufgabe verständlich sein sollte und dass sie an einem qualitativ hochwertigen, klar identifizierbaren Teilergebnis arbeiten wollen. Während Letzteres eindeutige Parallelen zu den hier gemachten Beobachtungen aufweist, hat der erste Teilaspekt zwar auf den ersten Blick keine direkte Überschneidung zur hier diskutierten Thematik, allerdings lässt sich argumentieren, dass die Arbeit an einem qualitativ schlechten Code mit hohen technischen Schulden, der sehr wahrscheinlich sowieso überarbeitet werden sollte, auch das Verständnis für den Zweck einer Aufgabe einschränken könnte. Auch Besker et al. formulieren vorsichtig, als Ergebnis ihrer Forschung zu den negativen Effekten von technischen Schulden auf die Produktivität und Arbeitsmoral:

> „[...] software suffering from TD reduces developers' morale and thereby also their productivity.“[6]

Insgesamt wurde im Rahmen der hier durchgeführten Literaturrecherche zwar einige Arbeiten gefunden, die die positiven Effekte der Zufriedenheit der Entwickler*innen auf die Arbeitsergebnisse also die produzierte Software hervorheben (z.B. [26][25][41]), allerdings wurden bis auf die oben zitierten Forschungsergebnisse keine aussagekräftigen Arbeiten gefunden, die die entgegengesetzte Wechselwirkung - also den Einfluss von Softwarequalität und technischen Schulden auf die Zufriedenheit und Arbeitsmoral der Entwickler - betrachten. Die im Rahmen dieser Forschung über alle Unternehmen hinweg beobachteten Qualitätsansprüche der Entwickler,

die durch die Entwickler bei Unternehmen C beschriebene Frustration, sowie die Überschneidungen zu anderen Forschungsergebnissen geben den zwei hier formulierten Thesen starken Rückhalt.

8.1.6 Auswirkungen der technischen Expertise der Entscheider*innen

5.1 Technische Expertise bei den Entscheider*innen kann die Kommunikation mit den Entwicklern rund um das Thema der technischen Schulden verbessern und dazu führen, dass man diesen mehr Zeit für das Beseitigen von technischen Schulden einräumt.
5.2 Gründer*innen mit technischer Expertise können eine aktivere Rolle in der Entwicklung einnehmen und so die eigenen Qualitätsansprüche oder strategischen Entscheidungen zum Management von technischen Schulden durchsetzen.
5.3 Entscheider*innen mit technischer Expertise haben mitunter hohe Qualitätsstandards.
5.4 Entscheider*innen mit technischer Expertise können die zukünftigen Kosten von technischen Schulden besser einschätzen und folglich kalkulierter technische Schulden in weniger kritischen Bereichen (unterschiedliche Teile der Codebasis oder unterschiedliche Arten von technischen Schulden) aufnehmen.

Da im Rahmen der hier durchgeführten Forschung nur Gründer mit sehr hoher technischer Expertise befragt wurden, ist die Datengrundlage durch den fehlenden Vergleich mit Gründer*innen / Entscheider*innen mit wenig oder keiner technischen Expertise, verhältnismäßig dünn, weshalb auch die hier aufgestellten Thesen eher schwach und im Konjunktiv formuliert sind. These 1 beruht dabei auf den Aussagen des Gründers und der Entwickler bei C. Die 2. These auf den Aussagen aller drei Gründer und These 3 und 4 vor allem auf den Beobachtungen bei Unternehmen A. Auch Besker et al. identifzieren die technische Expertise der Gründer als Einflussfaktor in Bezug auf die Aufnahme von technischen Schulden in Startups[7]. Im Kontrast zu den hier gemachten Beobachtungen beschreiben die Autoren dort, dass Gründer*innen mit weniger technischer Expertise die Ansammlung technischer Schulden eher vermeiden würden, da ihnen die potentiellen Vorteile, die mit der Aufnahme einhergingen (u.a. Zeitersparnisse) nicht bewusst seien, wohingegen Gründer*innen mit technischer Expertise eher dazu geneigt wären technische Schulden aufzunehmen. In dieser Forschung wurde dagegen beobachtet, dass Gründer A mit der meisten technischen Expertise, aufgrund von langjähriger Berufserfahrung, am umsichtigsten mit der Aufnahme von technischen Schulden umging. Allerdings ist anzuführen, dass alle interviewten Gründer ein hohes Maß an technischer Expertise aufwiesen und folglich die Unterschiede in ihrem Umgang mit technischen Schulden vermutlich auf andere Faktoren zurückzuführen ist.

8.2 Entwickler*innenzufriedenheit als potentiell, fundamentaler Erfolgsfaktor eines Technologie-Startups

Im Sinne des axialen Codierens nach Strauss und Corbin wurde hier versucht, die beobachteten Phänomene weiter zu abstrahieren und in einen größeren Gesamtkontext zu rücken.

Während der Auswertung der Interviews stach dabei die Zufriedenheit der Entwickler in den untersuchten Unternehmen besonders hervor. Am stärksten dabei in dem Gespräch mit den Entwicklern aus Unternehmen C, aber auch in dem mit Entwickler A. So sagt beispielsweise Entwickler C2:

> „Und ich denke, wir haben so ein Glück mit der Flexibilität und Arbeitszeit und Arbeitsatmosphäre. Also es fühlt sich nicht wie so ein klassisches amerikanisches Klischee-Startup an, sondern sehr entspannt eigentlich."
> - C2

Zudem war die Zufriedenheit der Entwickler ein Thema in den Gesprächen mit allen Gründern. So berichtete Gründer A davon, dass einige Funktionalität primär fertiggestellt wurde, um Entwickler nicht durch den Abbruch der Entwicklung und der damit einhergehenden vertanen Arbeit zu frustrieren. Und dass - im betriebswirtschaftlichen Sinne - sich das Geld für eine aus diesem Grund motivierte Weiterentwicklung aus dem gleichen „Bespaßungs-Topf" speise wie beispielsweise eine Betriebsfeier oder ein Kicker-Tisch. Gründer B und C stellten heraus, dass Überlegungen zur Zufriedenheit der Entwickler eine Rolle bei Entscheidungen zur Aufnahme von technischen Schulden und dem Festhalten bzw. Verschieben von Deadlines hatte, was natürlich miteinander einhergeht. Auch die Aussagen der Entwickler von C lassen darauf schließen, dass die Einschätzungen des Entwicklungsteams sehr stark bei Entscheidungen berücksichtigt werden. Es handelt sich dabei also um eine sehr starke Gemeinsamkeit über alle befragten Unternehmen hinweg, dass zum einen eine - aus der Perspektive eines Außenstehenden - gute Unternehmskultur herrscht, was zum anderen mit einer hohen Zufriedenheit der Entwickler einhergeht.

Es muss natürlich konstatiert werden, dass alle untersuchten Unternehmen durch Menschen mit hoher technischer Expertise gegründet wurden, die alle Software entwickelt haben und sich damit gut in die Position der Softwareentwickler*innen in ihren Unternehmen hinein versetzen können. Man kann die daraus resultierende Empathie durchaus als Hauptursache für eine wohlwollende Haltung des Managements dem Entwicklerteam gegenüber ansehen, allerdings und beruhend auf den im Rahmen dieser Forschung gemachten Beobachtungen wird die Zufriedenheit der Entwickler*innen als potentiell kritischer Erfolgsfaktor für Technologie-Startups verstanden.

Grundsätzlich ist festzuhalten, dass es für Startups, die ein komplexes Softwaresystem einwickeln, von strategischer Bedeutung ist, Entwickler*innen langfristig zu halten. So schreibt Sutton bereits im Jahr 2000:

> „Software project managers have long recognized the importance of good people forsuccessful software development. One challenge for many start-up companies is to hire and retain such star developers. [...] General competence among developers, in all roles and at all stages of the life cycle, remains crucial."[31]

Der Hauptgrund dafür ist, dass das Softwaresystem kontinuierlich weiterentwickelt werden muss, um die Anforderungen des Marktes zu erfüllen wie auch Heitlager et al. betonen:

> „It is too optimistic to assume that market uncertainty is resolved with the first introduction of the product to the market. Therefore product innovations will continue to take place to meet market demand.“[28]

Sie führen darüber hinaus an, dass diese Unsicherheit über die Anforderungen des Marktes nur durch die Weiterentwicklung der Software verringert werden kann und auch der Gewinn neuer Kunden, zu neuen Anforderungen führen wird, was letzten Endes wieder Änderungen der Software nach sich zieht. Dies deckt sich mit den Beobachtungen im Rahmen dieser Forschung. Bei Unternehmen A entsteht neue Funktionalität in ersten Linie durch Kundenanforderungen im Rahmen neuer Projekte und bei C durch das Erweitern der betriebenen Applikationen, um die Endverbraucher (vor allem jene, die ein Abonnement abgeschlossen haben) weiter zufrieden zu stellen.

Dies macht die Wartbarkeit bzw. Erweiterbarkeit des Softwaresystems entscheidend für den Erfolg der Unternehmung. Aufgrund der kleinen Teamgröße von Startups - bedingt durch das kurze Bestehen und die Ressourcenknappheit - sowie der Arbeit mit neuen Technologien ist die Erweiterbarkeit der Software - nach eigener Einschätzung - keine Eigenschaft, die sich alleine auf der Codebasis und deren Dokumentation stützt, sondern ist zudem fundamental von dem Wissen und den Fähigkeiten der Entwickler abhängig, welches im Verlauf der Softwareentwicklung immer weiter zu nimmt, insbesondere bei der Arbeit mit innovativen Technologien. Das gilt zwar auch zu einem Teil für die Entwicklung in größeren Organisationen oder für Software mit einem weniger hohen Innovationsanteil, allerdings ist die Erweiterbarkeit in kleinen Teams viel stärker von dem Wissen einzelner Entwickler abhängig, wie das beispielsweise auch bei Unternehmen C beobachtet wurde. So sagt der erste Entwickler, welcher den Rendering-Kern von Unternehmen C damals geschrieben hat zum Zeitpunkt der Befragung im April 2021:

> „Selbst [Entwickler C3], der seit 2015 Festangestellter ist, hat kaum was mit dem Kern zu tun, gehabt, weil ich den wirklich alleine geschrieben habe. Also auch die neuste Version die wir haben. Das Wissen muss geteilt werden.“ - C2

Diese Wissensmonopole vor allem in Bezug auf neue Technologien, machen die Erweiterbarkeit der Software stark von einzelnen Entwicklern abhängig. In Konsequenz ist es für die Startups von höchster strategischer Bedeutung diese Fachkräfte möglichst lang für die Entwicklung zu halten. Eine Einschätzung, die auch der technische Berater von Startup C teilt:

> „Ich denke auch, dass Zufriedenheit in so einer hochspeziellen Sparte von Softwareentwicklung ein fundamentales Gut ist, die Leute zu halten und vielleicht auch zu akquirieren.“ - C5

Und auch Gründer A betont:

> „[...] Grundsätzlich versuchen wir die Entwickler lange an uns zu binden. Und das funktioniert auch ganz gut.“ - A1

Wie in den Kapiteln 7.5 und 8.1.5 vorgestellt und diskutiert, wurde bei den Entwicklern eine intrinsische Motivation, qualitativ hochwertigen Code zu produzieren und damit die Aufnahme von technischen Schulden weitestgehenden zu vermeiden, beobachtet. So kommen auch Proccaccino in ihrer Umfrage zur Motivation von Softwareentwicklern zu dem Schluss:

> „Our findings indicate that practitioners consider software projects successful if they provide intrinsic, internally motivating work to develop software systems that both meet customer/user needs and are easy touse.“[42]

In Konsequenz hat die in den Kapitel 7.3 und 8.1.3 beschriebene Vermeidung von technischen Schulden im Alltag für die untersuchten Unternehmen einen doppelt, positiven Effekt auf die Erweiterbarkeit der Software:

1. Durch die Vermeidung von technischen Schulden wird die Erweiterbarkeit der Software nicht beeinträchtigt.

2. Die Vermeidung technischer Schulden liegt im intrinsischen Interesse der meisten Entwickler*innen und trägt zu einer motivierenden Arbeit mit der Codebasis bei. Dadurch sinkt die Gefahr Mitarbeiter und deren Kenntnisse zu verlieren, was im Umkehrschluss eine einfachere, zukünftige Erweiterbarkeit garantiert.

Wie kritisch dieses über lange Jahre aufgebaute Fachwissen für die zukünftige Entwicklung sein kann, wird beispielsweise bei folgender Aussage von Entwickler C3 deutlich, der beschreibt, wie sehr das aktuelle Refactoring durch die in der Vergangenheit gesammelten Erfahrungen des Entwicklerteams profitiert:

> „Das [Refactoring] funktioniert auch deshalb so gut, weil der Core konzeptionell nicht komplett neu gemacht werden muss, weil wir einfach schon 7 Jahre Erfahrung mit einem anderen Core gemacht haben. Da gibt es schon Work-Flows oder eine Architektur, wo wir wissen, dass ist eine gute bzw. wir wissen wo Schwachstellen sind. Und die können wir jetzt wahrscheinlich beheben. [...] Und das ist auch so eine Sache gerade wenn man das Thema Startup betrachtet. Es wird halt viel entwickelt und bestimmte Sachen werden halt erst [gut], wenn man ein paar mal damit auf die Nase gefallen ist oder Erfahrungen sammeln konnte.“ - C3

Auf der anderen Seite ermöglicht ein stabiles Entwicklungsteam und das mit dem Team gewachsene Fachwissen die in den Kapiteln 7.4 und 8.1.4 beschriebene, kontrollierte Aufnahme von technischen Schulden zur kurzfristigen Beschleunigung der Entwicklung. Dies ermöglicht es dem Startup in Zeiten von beispielsweise erhöhtem, finanziellen Druck in Abstimmung mit den Entwickler*innen diesen Druck durch Aufnahme von technischen Schulden umzuleiten und so den betriebswirtschaftlichen Spielraum zu erhöhen, was letzten Endes das entscheidende Zünglein an der Waage sein kann, das über Erfolg oder Misserfolg der Unternehmung entscheidet.

Aus diesen Gründen ist auch das in den Kapiteln 7.2 und 8.1.2 dargelegte Refactoring

so entscheidend für den Erfolg eines Startups - sei es im Kleinen, nachdem technische Schulden aufgrund von Projektanforderungen aufgenommen wurden oder im Großen, beispielsweise nachdem sich bewusste oder unbewusste technische Schulden zu Beginn einer Startup-Unternehmung - wie in 7.1 und 8.1.1 vorgestellt - akkumuliert haben.

Damit ist das zentrale Ergebnis der in dieser Arbeit beschriebenen Forschung:

Ein stabiles Entwicklungsteam ist fundamental wichtig für den Balanceakt von Anhäufung und Abbau von technischen Schulden während der Entwicklung. Das Meistern dieses Balanceaktes ist eine entscheidende Fähigkeit für ein Software-Startup, um möglichst schnell Funktionalität zu schaffen, womit die Unsicherheit in Bezug auf die Anforderungen des Marktes verringert werden kann, und ist zum Teil unerlässlich aufgrund von begrenzten finanziellen Ressourcen. Die gezielte Aufnahme von technischen Schulden kann helfen dieses Ziel zu erreichen. Gleichzeitig beeinflusst die Anhäufung von technischen Schulden nicht nur die Qualität der Codebasis und führt damit zu höheren Entwicklungsaufwänden, sondern erschwert zudem die Arbeit der Entwickler und kann damit zu Frustration führen. Die Vermeidung von technischen Schulden trägt damit zur Aufrechterhaltung eines stabilen Entwicklungsteams bei, da die Zufriedenheit der Entwickler*innen, welche als potentiell kritisch angesehen wird, um Fluktuation vorzubeugen, nicht in Mitleidenschaft gezogen wird.
Während ein stabiles Entwicklungsteam also die Aufnahme von technischen Schulden erleichtern kann und auch deren negativen Effekte z.T. vermindert[9] müssen technische Schulden wenn möglich wieder abgebaut werden, um Frustration vorzubeugen, welche im schlimmsten Fall zur Abwanderung von für Startups besonders wichtigem Humankapital führen kann.

8.3 Abgeleitete Empfehlungen

In Anlehnung an die hier aufgestellten Thesen und weitere Beobachtungen aus den Interviews werden folgende Empfehlungen für Software-Startups aufgestellt. Diese sind bewusst etwas schärfer formuliert, wohl wissend, dass es weiterer Forschung bedarf, um die empirische Grundlage dieser Aussagen zu verbreitern.

1. 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.

2. Technische Schulden sollten generell vermieden werden, da das langfristige Arbeiten mit einer qualitativ schlechten Codebasis frustrierend für die Entwickler sein kann.

[9] Beispielsweise haben technische Schulden durch fehlende Dokumentation weniger negative Konsequenzen für ein eingespieltes Entwicklerteam.

3. In wie weit 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.

4. 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.

5. Das Entwickeln vieler Prototypen kann dabei helfen unbewussten technischen Schulden beispielsweise bei der Gestaltung der Architektur oder der Auswahl von Bibliotheken und Frameworks vorzubeugen.

6. 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.

9 Gültigkeit

9.1 Interne Validität

Die wohl größte Gefahr bei der qualitativen Forschung zu technischen Schulden mittels Interviews liegt in der schwammigen Definition des Begriffs aufgrund seines metaphorischen Ursprungs. Wie bereits in Kapitel 2 angeführt existieren viele unterschiedlich breit gefasste Definitionen in der Theorie und folglich auch in der Praxis, was auch im Rahmen dieser Forschung beobachtet wurde. So war eine Erkenntnis im Rahmen der Befragung, dass direkte Fragen zum Umgang mit technischen Schulden eher vermieden werden sollten, um sich stattdessen mehr auf konkrete Beispiele zu fokussieren. Bei der Auswertung der ersten Interviews wurde festgestellt, dass Teile der Antworten nicht weiterverwendet werden konnten, da unklar war in wie weit sich diese überhaupt auf die in dieser Forschung zugrunde gelegte Definition beziehen. Spätere Interviews wurden zudem mit einer kleinen Einführung in die Thematik von Seiten des Interviewführers begonnen. Während vor allem die frühen Interviews methodisch nicht optimal geführt wurden, wurde bei der Auswertung stark darauf geachtet, nur klare und eindeutig zuzuordnende Aussagen der Befragten zu berücksichtigen. Die geführten Follow-up Gespräche dienten auch dem Zweck, Unklarheiten aus den ersten Interviews auszuräumen.

Eine weitere Schwäche der hier betriebenen Untersuchung ist die fehlende Validierbarkeit der Aussagen der Interviewpartner. Forschung mittels Interviews ist immer mit dem Nachteil verbunden, dass man nicht Daten über die Realität erhebt, sondern Daten zu der Wahrnehmung der Realität durch die Befragten. Folglich ist eine subjektive Verzerrung der Realität immer enthalten. Um dem entgegen zu wirken wurde versucht möglichst viele Datenpunkte zu erheben, zum einen indem mindestens 2 Personen pro Unternehmen befragt wurden und zum anderen durch das Integrieren einer zeitlichen Dimension in Form von Follow-up Gesprächen. Während dies bei den Unternehmen A und C gelang, basieren die Beobachtungen bei Unternehmen B lediglich auf einem Interview mit dem Gründer. In Folge stützen sich die aufgestellten Thesen zu weiten Teilen auf den Aussagen der Befragten Gründer und Entwickler in den Unternehmen A und C. Beobachtungen bei B wurden in weiten Teilen nur unterstützend herangezogen. Des Weiteren würden die hier formulierten Erkenntnisse auf noch sicheren Beinen stehen, wenn die Aussagen der Befragten neben Interviews auch in anderer Form validiert worden wären. Hier würden sich beispielsweise Code-Reviews, um Aussagen zur Qualität der Codebasis und zum Level an technischen Schulden zu kontextualisieren und über die Unternehmen hinweg vergleichbar zu machen. Auch wäre das Durchführen von Observationen in den untersuchten Unternehmen denkbar, um Beobachtungen zur Unternehmenskultur, Arbeitsweise und Entscheidungsprozessen aus erster Hand zu sammeln. Letzteres an der zum Zeitpunkt der Forschung tobenden Covid-19-Pandemie scheiterte. Zudem waren die Unternehmen, deren Geschäftmodell die Arbeit mit innovativen Technologien ist und deren Wettbewerbsvorteile folglich zu weiten Teilen in ihrer Codebasis zu finden sind, sehr zurückhaltend bzw. ablehnend bzgl. Zugängen zur Codebasis. Dies stellt ein Problem dar, das auch zukünftige Forschung haben wird.

Trotz der hier diskutierten Schwächen der Forschung ist festzuhalten, dass es sich hier um eine sehr kleine explorative Forschung handelt, deren Ziel es nicht ist, Ergebnisse zu produzieren, die in Stein gemeißelt sind, sondern erste Beobachtungen zu diskutieren, darauf aufbauend Thesen zu formulieren und diese mit anderen Erkenntnissen vergleichbarer Arbeiten zusammenzuführen. Innerhalb dieses Kontextes wird die interne Validität als hoch bewertet.

9.2 Externe Validität

Die drei hier vorgestellten und untersuchten Unternehmen gleichen sich in sehr vielen Charakteristika. Insbesondere wurden alle drei Unternehmen durch studierte oder promovierte Informatiker gegründet, arbeiten mit innovativen Technologien und bestehen bereits eine lange Zeit am Markt. Die Ergebnisse dieser Arbeit sind damit am ehesten auf Unternehmen mit ähnlichen Eigenschaften übertragbar. Es ist denkbar, dass sich die Softwareentwicklungsprozesse in Startups, deren Alleinstellungsmerkmal nicht die entwickelte Software ist[10] und/oder deren Gründer*innen über keine technische Expertise verfügen, fundamental von den hier beobachteten unterscheiden. Nichtsdestotrotz macht gerade die Langlebigkeit der untersuchten Unternehmen

[10]Beispielsweise ein Startup, das einen klassischen Online-Shop betreibt, aber deren Alleinstellungsmerkmal in der Logistik liegt

und die technische Expertise der Gründer einen großen Vorteil dieser Arbeit aus. Die hier aufgestellten Thesen und abgeleiteten Empfehlungen basieren auf den Erfahrungen von erfolgreichen Startup-Unternehmungen, deren Umgang mit technischen Schulden zumindest einen Teil dieses Erfolgs ausgemacht hat. Diese Forschung trägt die Erfahrungen und gelernten Lektionen dieser Unternehmen in Bezug auf technische Schulden zusammen und schafft damit einen Mehrwert für viele unterschiedliche Unternehmen, die Software entwickeln. Obwohl vermutet wird, dass viele der hier formulierten Thesen und Empfehlungen - auch aufgrund der Überschneidungen zu anderen verwandten Arbeiten - auf die Softwareentwicklung in Software-Startups mit anderen Charakteristika übertragbar sind, ist der hier untersuchte Datensatz so klein und homogen, dass die externe Validität als gering zu bewerten ist. Nur weitere qualitative und quantitative Forschung in Startups mit heterogenen Eigenschaften kann die externe Validität erhöhen. Es ist allerdings zu konstatieren, dass solche Forschung wohl nie eine gewisse Verzerrung und damit auch eine eingeschränkte Übertragbarkeit vermeiden kann, da Daten in Unternehmen, die dem Thema abgeneigt sind oder deren Softwareentwicklung so chaotisch verläuft, dass sie keinem Externen Zugriff gewähren möchten, fast gar nicht erhoben werden können.

10 Zusammenfassung und Ausblick

In der hier beschriebenen Forschung wurden drei langlebige Software-Startup-Unternehmen zu ihrem Umgang mit technischen Schulden befragt, um durch Methoden der Grounded Theory mehr Erkenntnisse zum Management von technischen Schulden in Startup-Unternehmungen im Allgemeinen zu sammeln und um im Speziellen zu ergründen, ob es Gemeinsamkeiten im Umgang mit technischen Schulden über die drei befragten Unternehmen hinweg gibt, die zum Erfolg dieser beitrugen. Dazu wurden insgesamt 11 Interviews mit 8 Personen aus den Unternehmen geführt. Im Verlauf der Forschung bildete sich dabei neben abstrahierten Phänomenen, wie der Anhäufung von technischen Schulden zu Beginn einer Startup-Unternehmung oder dem Refactoring als wirtschaftlich bedeutende Entscheidung insbesondere heraus, dass die Zufriedenheit der Entwickler durch ihre eigenen intrinsischen Qualitätsansprüche durch den Umgang mit technischen Schulden des Unternehmens beeinflusst wird. Da Startups in der Regel nur ein kleines Entwicklerteam unterhalten (können) und unter Zeitdruck operieren (sei es aufgrund von Ressourcenknappheit oder des Marktumfelds), ist die Zufriedenheit und Motivation der einzelnen Entwickler ein sehr hohes Gut, um Fluktuation vorzubeugen und das potentiell kritisch für den Unternehmenserfolg sein kann.
Die Übertragbarkeit der hier durchgeführten Forschung ist aufgrund der geringen Anzahl an Befragten limitiert. Da die Forschung zu technischen Schulden in Startups relativ jung und überschaubar ist, leistet diese Arbeit trotzdem einen wichtigen Beitrag, auf dem in weiterer Forschung aufgebaut werden muss. Besonders wünschenswert wären dabei vor allem qualitative Forschung, die neben Aussagen aus den Interviews auch den Quellcode, auf die sich die Aussagen beziehen, untersucht, um so Aussagen vergleichbarer zu machen. Darüber hinaus würde auch weitere Forschung in Startups mit deutlich unterschiedlichen Charakteristika zu übertragbareren und gesicherteren Erkenntnissen beitragen.

Literatur

[1] A.-K. Achleitner. Start-up-unternehmen. https://https://wirtschaftslexikon.gabler.de/definition/start-unternehmen-42136/version-265490, 2018. Aufgerufen am 05.07.2021.

[2] E. Allman. Managing technical debt. *Commun. ACM*, 55(5):50–55, 2012.

[3] P. Avgeriou, P. Kruchten, I. Ozkaya, and C. B. Seaman. Managing technical debt in software engineering (dagstuhl seminar 16162). *Dagstuhl Reports*, 6(4):110–138, 2016.

[4] P. C. Avgeriou, D. Taibi, A. Ampatzoglou, F. A. Fontana, T. Besker, A. Chatzigeorgiou, V. Lenarduzzi, A. Martini, N. Moschou, I. Pigazzini, et al. An overview and comparison of technical debt measurement tools. *IEEE Software*, 2020.

[5] S. Beecham, N. Baddoo, T. Hall, H. Robinson, and H. Sharp. Motivation in software engineering: A systematic literature review. *Inf. Softw. Technol.*, 50(9-10):860–878, 2008.

[6] T. Besker, H. Ghanbari, A. Martini, and J. Bosch. The influence of technical debt on software developer morale. *J. Syst. Softw.*, 167:110586, 2020.

[7] T. Besker, A. Martini, R. E. Lokuge, K. Blincoe, and J. Bosch. Embracing technical debt, from a startup company perspective. In *2018 IEEE International Conference on Software Maintenance and Evolution, ICSME 2018, Madrid, Spain, September 23-29, 2018*, pages 415–425. IEEE Computer Society, 2018.

[8] A. Böhm. *Grounded Theory-wie aus Texten Modelle und Theorien gemacht werden*, volume 14. UVK Univ.-Verl. Konstanz, 1994.

[9] N. Brown, Y. Cai, Y. Guo, R. Kazman, M. Kim, P. Kruchten, E. Lim, A. MacCormack, R. L. Nord, I. Ozkaya, R. S. Sangwan, C. B. Seaman, K. J. Sullivan, and N. Zazworka. Managing technical debt in software-reliant systems. In G. Roman and K. J. Sullivan, editors, *Proceedings of the Workshop on Future of Software Engineering Research, FoSER 2010, at the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2010, Santa Fe, NM, USA, November 7-11, 2010*, pages 47–52. ACM, 2010.

[10] O. Cico, R. Souza, L. Jaccheri, A. Nguyen-Duc, and I. Machado. Startups transitioning from early to growth phase - A pilot study of technical debt perception. In E. Klotins and K. Wnuk, editors, *Software Business - 11th International Conference, ICSOB 2020, Karlskrona, Sweden, November 16-18, 2020, Proceedings*, volume 407 of *Lecture Notes in Business Information Processing*, pages 102–117. Springer, 2020.

[11] Z. Codabux and B. Williams. Managing technical debt: An industrial case study. In *Proceedings of the 4th International Workshop on Managing Technical Debt*, MTD '13, pages 8–15, Piscataway, NJ, USA, 2013. IEEE Press.

[12] M. Crowne. Why software product startups fail and what to do about it. evolution of software product development in startup companies. In *IEEE International Engineering Management Conference*, volume 1, pages 338–343. IEEE, 2002.

[13] W. Cunningham. The wycash portfolio management system. *ACM SIGPLAN OOPS Messenger*, 4(2):29–30, 1992.

[14] B. Curtis, J. Sappidi, and A. Szynkarski. Estimating the size, cost, and types of technical debt. In P. Kruchten, R. L. Nord, I. Ozkaya, and J. Visser, editors, *Proceedings of the Third International Workshop on Managing Technical Debt, MTD 2012, Zurich, Switzerland, June 5, 2012*, pages 49–53. IEEE/ACM, 2012.

[15] R. R. de Almeida, R. do Nascimento Ribeiro, C. Treude, and U. Kulesza. Business-driven technical debt prioritization: An industrial case study, 2020.

[16] N. Devos, D. Durieux, and C. Ponsard. Managing technical debt in it start-ups–an industrial survey. In *International Conference on Software and System Engineering and their Applications (ICSSEA)*, 2013.

[17] N. A. Ernst, S. Bellomo, I. Ozkaya, R. L. Nord, and I. Gorton. Measure it? manage it? ignore it? software practitioners and technical debt. In E. D. Nitto, M. Harman, and P. Heymans, editors, *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*, pages 50–60. ACM, 2015.

[18] F. A. Fontana, R. Roveda, and M. Zanoni. Technical debt indexes provided by tools: A preliminary discussion. In *8th IEEE International Workshop on Managing Technical Debt, MTD 2016, Raleigh, NC, USA, October 4, 2016*, pages 28–31. IEEE Computer Society, 2016.

[19] M. Fowler. Technicaldebtquadrant. https://martinfowler.com/bliki/TechnicalDebtQuadrant.html, 2009. Aufgerufen am 28.10.2020.

[20] S. Freire, N. Rios, B. Gutierrez, D. Torres, M. Mendonça, C. Izurieta, C. Seaman, and R. O. Spínola. 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, page 210–219, New York, NY, USA, 2020. Association for Computing Machinery.

[21] C. Giardino, S. S. Bajwa, X. Wang, and P. Abrahamsson. Key challenges in early-stage software startups. In C. Lassenius, T. Dingsøyr, and M. Paasivaara, editors, *Agile Processes, in Software Engineering, and Extreme Programming - 16th International Conference, XP 2015, Helsinki, Finland, May 25-29, 2015, Proceedings*, volume 212 of *Lecture Notes in Business Information Processing*, pages 52–63. Springer, 2015.

[22] C. Giardino, N. Paternoster, M. Unterkalmsteiner, T. Gorschek, and P. Abrahamsson. Software development in startup companies: The greenfield startup model. *IEEE Trans. Software Eng.*, 42(6):585–604, 2016.

[23] C. Giardino, M. Unterkalmsteiner, N. Paternoster, T. Gorschek, and P. Abrahamsson. What do we know about software development in startups? *IEEE Softw.*, 31(5):28–32, 2014.

[24] B. G. Glaser. *The discovery of grounded theory : strategies for qualitative research / Barney G. Glaser and Anselm L. Strauss.* Observations. Weidenfeld & Nicolson, London, 1968.

[25] D. Graziotin, X. Wang, and P. Abrahamsson. Happy software developers solve problems better: psychological measurements in empirical software engineering. *CoRR*, abs/1505.00922, 2015.

[26] D. Graziotin, X. Wang, and P. Abrahamsson. How do you feel, developer? an explanatory theory of the impact of affects on programming performance. *PeerJ Comput. Sci.*, 1:e18, 2015.

[27] Y. Guo, R. O. Spínola, and C. B. Seaman. Exploring the costs of technical debt management - a case study. *Empir. Softw. Eng.*, 21(1):159–182, 2016.

[28] I. Heitlager, S. Jansen, R. Helms, and S. Brinkkemper. Understanding the dynamics of product software development using the concept of coevolution. In *2006 Second International IEEE Workshop on Software Evolvability (SE'06)*, pages 16–22. IEEE, 2006.

[29] J. Holvitie, S. A. Licorish, R. O. Spínola, S. Hyrynsalmi, S. G. MacDonell, T. S. Mendes, J. Buchan, and V. Leppänen. Technical debt and agile software development practices and processes: An industry practitioner survey. *Inf. Softw. Technol.*, 96:141–160, 2018.

[30] G. Iuliia. Technical debt management in russian software development companies. 2017.

[31] S. M. S. Jr. The role of process in a software start-up. *IEEE Softw.*, 17(4):33–39, 2000.

[32] I. Khomyakov, Z. Makhmutov, R. Mirgalimova, and A. Sillitti. An analysis of automated technical debt measurement. In J. Filipe, M. Śmiałek, A. Brodsky, and S. Hammoudi, editors, *Enterprise Information Systems*, pages 250–273, Cham, 2020. Springer International Publishing.

[33] T. Klinger, P. L. Tarr, P. Wagstrom, and C. Williams. An enterprise perspective on technical debt. In I. Ozkaya, P. Kruchten, R. L. Nord, and N. Brown, editors, *Proceedings of the 2nd Workshop on Managing Technical Debt, MTD 2011, Waikiki, Honolulu, HI, USA, May 23, 2011*, pages 35–38. ACM, 2011.

[34] E. Klotins, M. Unterkalmsteiner, P. Chatzipetrou, T. Gorschek, R. Prikladnicki, N. Tripathi, and L. B. Pompermaier. Exploration of technical debt in startups. In F. Paulisch and J. Bosch, editors, *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice, ICSE (SEIP) 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, pages 75–84. ACM, 2018.

[35] P. Kruchten, R. L. Nord, and I. Ozkaya. Technical debt: From metaphor to theory and practice. *IEEE Softw.*, 29(6):18–21, 2012.

[36] P. Kruchten, R. L. Nord, I. Ozkaya, and D. Falessi. Technical debt: towards a crisper definition report on the 4th international workshop on managing technical debt. *ACM SIGSOFT Softw. Eng. Notes*, 38(5):51–54, 2013.

[37] M. M. Lehman. Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE*, 68(9):1060–1076, 1980.

[38] V. Lenarduzzi, T. Besker, D. Taibi, A. Martini, and F. A. Fontana. A systematic literature review on technical debt prioritization: Strategies, processes, factors, and tools. *J. Syst. Softw.*, 171:110827, 2021.

[39] Z. Li, P. Avgeriou, and P. Liang. A systematic mapping study on technical debt and its management. *J. Syst. Softw.*, 101:193–220, 2015.

[40] E. Lim, N. Taksande, and C. B. Seaman. A balancing act: What software practitioners have to say about technical debt. *IEEE Softw.*, 29(6):22–27, 2012.

[41] S. C. Müller and T. Fritz. Stuck and frustrated or in flow and happy: Sensing developers' emotions and progress. In A. Bertolino, G. Canfora, and S. G. Elbaum, editors, *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1*, pages 688–699. IEEE Computer Society, 2015.

[42] J. D. Procaccino, J. M. Verner, K. M. Shelfer, and D. Gefen. What do software practitioners really think about project success: an exploratory study. *J. Syst. Softw.*, 78(2):194–203, 2005.

[43] A. Przyborski and M. Wohlrab-Sahr. *Qualitative Sozialforschung*. Oldenbourg Wissenschaftsverlag, 2013.

[44] M. G. Stochel, P. Cholda, and M. R. Wawrowski. On coherence in technical debt research : Awareness of the risks stemming from the metaphorical origin and relevant remediation strategies. In *46th Euromicro Conference on Software Engineering and Advanced Applications, SEAA 2020, Portoroz, Slovenia, August 26-28, 2020*, pages 367–375. IEEE, 2020.

[45] A. L. Strauss. Grundlagen qualitativer sozialforschung : Datenanalyse und theoriebildung in der empirischen soziologischen forschung / anselm l. strauss., 1998.

[46] S. Tauber. Why tech startups fail - based on auditing 50+ saas startups. https://madewithlove.com/blog/this-is-madewithlove/why-tech-startups-fail-based-on-auditing-50-saas-startups/, 2021. Aufgerufen am 15.07.2021.

[47] M. Unterkalmsteiner, P. Abrahamsson, X. Wang, A. Nguyen-Duc, S. M. A. Shah, S. S. Bajwa, G. H. Baltes, K. Conboy, E. Cullina, D. Dennehy, H. Edison,

C. Fernández-Sánchez, J. Garbajosa, T. Gorschek, E. Klotins, L. Hokkanen, F. Kon, I. Lunesu, M. Marchesi, L. Morgan, M. Oivo, C. Selig, P. Seppänen, R. Sweetman, P. Tyrväinen, C. Ungerer, and A. Yagüe. Software startups - A research agenda. *e Informatica Softw. Eng. J.*, 10(1):89–124, 2016.

[48] N. Zazworka, M. A. Shaw, F. Shull, and C. Seaman. Investigating the impact of design debt on software quality. In *Proceedings of the 2Nd Workshop on Managing Technical Debt*, MTD '11, pages 17–23, New York, NY, USA, 2011. ACM.

[49] N. Zazworka, R. O. Spínola, A. Vetro, F. Shull, and C. B. Seaman. A case study on effectively identifying technical debt. In F. Q. B. da Silva, N. J. Juzgado, and G. H. Travassos, editors, *17th International Conference on Evaluation and Assessment in Software Engineering, EASE '13, Porto de Galinhas, Brazil, April 14-16, 2013*, pages 42–47. ACM, 2013.