**Gottfried Wilhelm**
**Leibniz Universität Hannover**
**Fakultät für Elektrotechnik und Informatik**
**Institut für Praktische Informatik**
**Fachgebiet Software Engineering**

# Vermeidung von technischen Schulden in Startups bei der Entwicklung von Software

# Avoiding technical debt in startups when developing software

**Masterarbeit**

im Studiengang Informatik

von

**Florian Heintz**

**Erstprüfender: Prof. Dr. rer. nat. Kurt Schneider**
**Zweitprüferin: Dr. rer. nat. Jil Ann-Christin Klünder**
**Betreuerin: Dr. rer. nat. Jil Ann-Christin Klünder**

**Hannover, 19. April 2024**

# Erklärung der Selbständigkeit

Hiermit versichere ich, dass ich die vorliegende Masterarbeit selbständig und ohne fremde Hilfe verfasst und keine anderen als die in der Arbeit angegebenen Quellen und Hilfsmittel verwendet habe. Die Arbeit hat in gleicher oder ähnlicher Form noch keinem anderen Prüfungsamt vorgelegen.

Hannover, den 19.04.2024

________________________________

Florian Heintz

## Danksagung

Als erstes möchte ich mich bei meiner Betreuerin Frau Dr. Jil Ann-Christin Klünder bedanken, welche mir stets bei Fragen geholfen und mir wertvolle Ratschläge gegeben hat. Diese Unterstützung hat sich als wichtiger Mehrwert im Laufe der Planung sowie Verschriftlichung der Arbeit erwiesen.

Bedanken möchte ich mich ebenfalls bei Herrn Prof. Dr. Kurt Schneider für sein wichtiges Feedback und bei den Forschenden, welche mir großzügigerweise deren Paper zur Verfügung gestellt haben.

## Zusammenfassung

Im Laufe der letzten Jahre hat sich die Anzahl an Software-Startups vervielfacht, da diese eine Möglichkeit bieten innovative Ideen, in der Hoffnung auf Erfolg, auf den Markt zu bringen. Durch das kompetitive Umfeld und die begrenzten zeitlichen sowie finanziellen Ressourcen ist dies allerdings kein einfaches Unterfangen, weshalb die meisten Startups scheitern. Um sich einen Vorteil zu erschaffen und die Software schnell zu veröffentlichen, sparen Startups nämlich oft an unterschiedlichen Aspekten des Produkts. Dies führt kurzfristig zwar zu mehr Effizienz, verursacht aber das Erscheinen von technischen Schulden, welche softwarebezogene Probleme darstellen, die entstehen, wenn Entwicklungsaufgaben nicht angemessen durchgeführt werden. Das Ansammeln dieser Schulden führt zu einer Verlangsamung des Wachstums und oft zu einem frühzeitigen Ende von Projekten. Startups unterschätzen technische Schulden, das sie deren Folgen nicht kennen oder nicht wissen, wie diese vermieden werden können, sodass sie nur eingeschränkt dagegen agieren. Aus diesem Grund wird ein strukturierter Überblick über bestehendes Wissen in Form einer Checkliste erstellt. Diese erklärt, welche Arten an technischen Schulden es gibt, welche Ursachen und Folgen diese haben und wie sie erkannt und vermieden werden können. Ebenfalls wird, aufbauend auf diesen Erkenntnissen, ein bestehendes Framework so erweitert, dass es beschreibt, wie die Entwicklung eines ersten Produkts in Startups, möglichst wenig technische Schulden verursacht. Die Ergebnisse liefern Informationen, die Startups, ohne den Einsatz einer höheren Ressourcennutzung, einen Vorteil verschaffen könnten.

## Abstract

In recent years, the number of software startups has multiplied, as they offer an opportunity to bring innovative ideas to the market in hopes of success. However, due to the competitive environment, limited time and financial resources, this is not an easy undertaking, which is why most startups fail. In order to gain an advantage and release software quickly, startups often cut corners on various aspects of the product. While this leads to more efficiency in the short term, it causes the appearance of technical debt, which represents software-related problems. These appear when development tasks are not performed appropriately. The accumulation of this debt leads to a slowdown in growth and, often, a premature project end. Startups underestimate technical debt because they do not know its consequences or how to avoid it, leading them to taking only limited measures against it. For this reason, a structured overview of existing knowledge is created in the form of a checklist. This list explains what types of technical debt exist, what causes and consequences they have, and how they can be recognized and avoided. Based on these findings, an existing framework will also be expanded to describe how an initial product can be developed, while minimizing technical debt. The results provide information that could give startups an advantage without an increased resource utilization.

# Inhaltsverzeichnis

1. Einleitung ........ 1

1.1 Motivation ........ 1
1.2 Problemstellung ........ 2
1.3 Lösungsansatz ........ 3
1.4 Struktur der Arbeit ........ 4

2. Grundlagen ........ 5

2.1 Startups ........ 5
2.2 Technische Schulden ........ 7
2.3 Technical Debt Management ........ 9
2.4 Der Softwareentwicklungs-Lebenszyklus ........ 11
2.5 Das Entrepreneurial Software Engineering Modell ........ 12

3. Verwandte Arbeiten ........ 19

3.1 Grundlegende Werke ........ 19
3.2 Erweiternde Studien ........ 19
3.3 Checkliste und ESEM-Erweiterung ........ 20

4. Methodik: Literaturstudie ........ 21

4.1 Ziel der Studie ........ 21
4.2 Vorgehen ........ 21

5. Ergebnisse: Checkliste zur Vermeidung von technischen Schulden ........ 29

5.1 Startups auf TD-Management vorbereiten ........ 29
5.2 Typen an technischen Schulden ........ 30
5.3 Checkliste ........ 31
5.4 Zusammenhänge zwischen TD-Typen ........ 55
5.5 Automatisierung ........ 56
5.6 Zusammengefasste Checkliste ........ 59

6. Erweiterung des Entrepreneurial Software Engineering Modells ........ 61

6.1 Zusammenfassung des ursprünglichen ESEM ........ 61
6.2 Notation ........ 62
6.3 Feature Requirements Generation & Evaluation – Technical Debt Prevention ........ 63
6.4 Design & User Experience Evaluation - Technical Debt Prevention ........ 66
6.5 Pre-Launch Testing - Technical Debt Identification ........ 67
6.6 Technical Debt Categorization & Prioritization ........ 69
6.7 Development Cycle ........ 72
6.8 Zusammengefasstes ESEM ........ 75

7. Diskussion ..... 77
7.1 Antworten auf die Forschungsfragen ..... 77
7.2 Neuheiten der Arbeit ..... 79
7.3 Einschränkungen der Gültigkeit der Ergebnisse ..... 80
7.4 Risiko VS Aufwand ..... 80
7.5 Mögliche Erweiterungen ..... 82

8. Zusammenfassung und Ausblick ..... 83
8.1 Zusammenfassung ..... 83
8.2 Ausblick ..... 84

Anhang ..... 85
Verzeichnisse ..... 123

# 1. Einleitung

## 1.1 Motivation

### 1.1.1 Kontext

Im Laufe der letzten Jahre hat sich die Zahl an Software-Startups vervielfacht. Bei diesen handelt es sich um neu gegründete Unternehmen ohne Betriebsgeschichte deren Ziel es ist, innovative Technologien zu entwickeln [1]. Allein in Deutschland wurden im ersten Halbjahr 2023, 239 neue Software-Startups gegründet. Das ist 23 Prozent höher als im zweiten Halbjahr 2022 [2] und mehr als in allen anderen Branchen, sodass nun insgesamt 31.9% aller Startups in Deutschland sich mit Software befassen [3]. Diese Unternehmen haben als Ziel Marktlücken zu füllen, indem sie innovative Produkte entwickeln, die zum Beispiel Privatpersonen in ihrem Alltag oder Unternehmen in ihrem Betrieb unterstützen. Vor allem künstliche Intelligenz und Cloud Computing zählen zu den Technologien, die momentan am meisten aufgegriffen werden [3]. Dies klingt äußerst vielversprechend, doch zum heutigen Zeitpunkt schaffen es leider mehr als zwei Drittel aller Startups nie ihren Investoren eine positive Rendite zu liefern [4] und im Durchschnitt scheitern 60% aller gegründeten Startups innerhalb der fünf ersten Jahre [5]. Das bedeutet, dass zwar die Idee eines Startups vielversprechend sein kann und jährlich eine hohe Anzahl an solchen Unternehmen entstehen, es aber nur die wenigsten schaffen, sich durchzusetzen.

### 1.1.2 Herausforderungen

Software-Startups haben nämlich mit einer Bandbreite an Herausforderungen zu kämpfen, wie etwa enge Deadlines und daher wenig Zeit für Entwicklung, begrenzte Ressourcen in Form von finanziellen Mitteln oder Mitarbeitern und das konstante Bedürfnis, sich vor potenziellen Kunden und Investoren zu beweisen [6]. Es ist für Startups oft überlebenswichtig, das erste Unternehmen zu sein, das eine gewisse Lösung auf den Markt bringt [7]. Um letzteres zu ermöglichen, müssen oft so schnell wie möglich Prototypen präsentiert werden, die demonstrieren sollen, dass ihre Idee vielversprechend ist. Deshalb kann es für Startups sehr verlockend sein, den einfachen Weg zu wählen und an verschiedensten Stellen zu sparen, um, zumindest kurzfristig, so effizient wie möglich zu sein.

Die Probleme, die dadurch allerdings entstehen können, sind vergleichbar mit finanziellen Schulden beim Kauf eines Hauses: Eine Person vereinbart ein Darlehen, um in der Lage zu sein ein Haus zu kaufen. Dieses Darlehen muss innerhalb eines gewissen Zeitraumes mit Zinsen zurückgezahlt werden. Sollte dies nicht rechtzeitig geschehen, verliert die Person ihr Haus [8].

Bei Startups entstehen diese Schulden zum Beispiel dadurch, dass Tests nicht ausgeführt werden, mangelndes Refactoring oder inexistente Dokumentation, um den Entwicklungsprozess zu beschleunigen [9].

### 1.1.3 Technische Schulden

All dies führt zu sogenannten technischen Schulden (TD). Diese Schulden stellen Probleme dar, die in Unternehmen entstehen, wenn diese Softwareentwicklungsaufgaben nicht angemessen durchführen [9]. Der Begriff „Technische Schulden“ beschreibt seit einigen Jahren nicht mehr ausschließlich Code-bezogene Probleme, sondern auch Komplikationen in allen anderen Phasen der Softwareentwicklung [10]. Darunter befinden sich die Definierung von Anforderungen, das Design des Produkts und zahlreiche andere Kriterien, die zur Entstehung unterschiedlich wichtiger Typen an technischen Schulden geführt haben (TD-Typen).

Wenn sich diese technischen Schulden in einem Softwareprojekt unbeaufsichtigt weiterentwickeln, kann dies zu erhöhten Kostenüberschreitungen führen, zu erheblichen Qualitätsmängeln in allen Bereichen sowie zur Unfähigkeit, neue Features hinzuzufügen [11]. Die Zinsen für technische Schulden, die das Startups zahlen müsste, erhöhen sich also konstant. Dies kann den gesamten Entwicklungsprozess immer weiter verlangsamen. Das liegt daran, dass die unterschiedlichen TD-Typen, wie zum Beispiel Code Debt, unter anderem Code vollkommen unübersichtlich machen können und zu einem Mangel an Zusammenarbeit führen. Diese Probleme führen letztendlich dazu, dass die Instandhaltung des Projekts mehr Aufwand benötigt als die eigentliche Entwicklung.
All dies kann im schlimmsten Fall zum kompletten Stillstand des Startups führen. Da Startups noch sehr kleine Unternehmen sind, die in einem ungewissen Kontext handeln, können diese technischen Schulden, sollten sie die Überhand gewinnen, somit durchaus das Scheitern des Unternehmens bedeuten [12].

#### 1.1.4 Mangel an Übersicht und Kontrolle

Doch nicht nur Entwicklern kann entgehen, wie viele technische Schulden sich in einem Projekt aufbauen. Das Management sowie die Kunden und Aktionäre haben oft kein klares Bild über das Ausmaß der technischen Schulden im Startup. Sie wissen dementsprechend nicht, wie viele Ressourcen erspart werden könnten, wenn diese technischen Schulden abgebaut werden würden. Wie es der Forscher Watts Humphrey beschreibt: „Software is not released, it escapes!“ [13]. Was dazu führt, dass zum Beispiel in jeder zehnten Linie Code, ein Defekt eingebaut wird [13]. Dieses Phänomen allein führt schon zu einer hohen Menge an technischen Schulden. Sogar in einem normalen Betrieb ist das Einbauen von Fehlern normal. In Startups fügt sich dem aber noch hinzu, dass Defekte oft freiwillig in Kauf genommen werden, wodurch die negativen Auswirkungen nur noch vergrößert werden. Entwickler verbringen im Schnitt nämlich 33% ihrer Arbeitszeit mit technischen Schulden, was weltweit jährlich zu einem ungefähren Verlust von 85 Milliarden US-Dollar führt [14].

Wenn in Unternehmen versucht wird, technische Schulden zu minimieren, dann wird dies oft nur mit spontanen und informellen Entscheidungen gemacht, was dazu führt, dass diese oft nicht angemessen sind [15]. Um mit den genannten Problemen umzugehen, bedarf es dementsprechend einer strukturierten Lösung. Zu den Erfolgsfaktoren von Startups gehört die Erstellung eines funktionierenden Software-Prototypen, der dazu dient, die Möglichkeiten und Funktionalitäten des Produkts zu präsentieren [16]. Sehr oft wird auf diesen Prototypen dann aufgebaut, um das fertige Produkt zu implementieren, sodass das Vermeiden von technischen Schulden dabei unbedingt beachtet werden muss.

### 1.2 Problemstellung

Es gibt also eine Bandbreite an Problemen, die dazu führen, dass in Startups technische Schulden eingegangen werden, die zum Scheitern des Unternehmens führen können. Sei dies ein Mangel an zeitlichen und finanziellen Ressourcen oder die Unwissenheit über Ursachen und Konsequenzen von technischen Schulden. Oft ist Startups auch nicht bewusst, wie viele Schulden unterschiedlicher Typen sie überhaupt haben. Zudem existiert bis jetzt nur wenig Literatur zum Management von technischen Schulden in Unternehmen und vor allem in Startups [7]. Es wäre daher sehr wichtig, Startups einen Überblick über die Entstehung und Effekte von technischen Schulden zu verleihen sowie über Methoden und Praktiken, die eingesetzt werden könnten, um diese Schulden zu managen.

## 1.3 Lösungsansatz

Um Lösungen für die genannten Probleme zu finden, werden daher Forschungsfragen formuliert, an denen sich die Recherche orientiert und die es somit zu beantworten gilt.

### 1.3.1 Forschungsfragen

Die Hauptforschungsfrage, mit der sich befasst wird, um die in Abschnitt 1.2 genannte Problemstellung zu lösen, lautet:
„Wie können unterschiedliche Typen an technischen Schulden in Startups in verschiedenen Phasen der Softwareentwicklung vermieden werden?“

Um diese Frage zu beantworten, wird sich mit einer Reihe an Teilfragen befasst:

- „Welche TD-Typen treten in Unternehmen am häufigsten auf?“. Dies ist wichtig, da Startups wissen müssen, welche Arten an technischen Schulden am wahrscheinlichsten deren Unternehmen gefährden könnten und somit vorrangig behandelt werden müssen.
- „Was verursacht das Erscheinen dieser TD-Typen und welche Folgen zieht dies mit sich?“. Hierdurch können Startups erkennen, auf welche Aspekte sie bei der Entwicklung besonders Acht geben müssen und was passieren könnte, sollten sie dies nicht machen.
- „In welcher Phase der Softwareentwicklung sind diese TD-Typen besonders relevant?“. Somit können die Startups einschätzen, wie viel Aufwand pro Entwicklungsphase benötigt wird, um TD-Management zu betreiben und entsprechende Ressourcen einplanen.
- „Welche Möglichkeiten gibt es, um diese technischen Schulden zu vermeiden?“. Das Beantworten dieser Forschungsfrage wird Startups einen Überblick über Handlungsmöglichkeiten geben, die sie einsetzen können, um technische Schulden zu managen.
- „Inwiefern ermöglichen es die gewonnenen Ergebnisse einen, für Startups etablierten, Entwicklungsprozess zu erweitern, um technische Schulden zu vermeiden?“. Hierdurch könnte ein Vorgehensmodell um Aspekte erweitert werden, die den optimalen Umgang mit technischen Schulden in Startups graphisch beschreiben.
- „Ist es für Startups immer von Vorteil technische Schulden zu vermeiden?“. Da Startups über begrenzte Ressourcen verfügen, muss analysiert werden, ob sich das Management von technischen Schulden für sie auch immer lohnt.

### 1.3.2 Zielsetzung

Das angestrebte Ergebnis wäre eine Zusammenfassung der gewonnen Erkenntnisse in Form einer, an den TD-Typen orientierten, Checkliste. Diese würde die Ursachen und Effekte von TD-Typen, Indikatoren, die auf diese hinweisen und Methoden für das TD-Management in Startups beinhalten.
In die Liste würde sich ein erweitertes Vorgehensmodell hinzufügen, das TD-Management Prozesse in die Aufgaben des Startups integriert.
Derartige Beiträge würden es Startups also ermöglichen, ihre Produktivität zu erhöhen und dabei zugleich die Qualität der entwickelten Software sehr positiv beeinflussen. Die Management-Ebene und Kunden sollten einen besseren Überblick über Risiken und Möglichkeiten von technischen Schulden haben. Sie könnten somit, anhand der präsentierten Methoden, optimalere Entscheidungen treffen. Doch auch Entwickler, Administratoren, Tester und alle anderen Teammitglieder sollten das Ausmaß an technischen Schulden verstehen, um ihr Verhalten entsprechend anpassen zu können. Dadurch

wäre das Team auch in der Lage, Entscheidungen des Managements besser nachzuvollziehen und dies würde zu einer allgemein besseren Arbeitsstimmung und letztendlich zu einem qualitativ hochwertigeren Produkt führen.

## 1.4 Struktur der Arbeit

Um zu beschreiben, wie diese Ziele erreicht werden können, folgt diese Arbeit einer definierten Struktur.
Als erstes wurde im Kapitel 1 der Kontext der Arbeit präsentiert, sodass im Anschluss die vorhandene Problematik erläutert werden konnte und welche Forschungsfragen definiert wurden, um diese zu lösen. Im anschließenden Kapitel 2 werden die nötigen Grundlagen erklärt, auf denen die Arbeit aufbaut und somit für das Verständnis dieser notwendig sind. Hierfür wird aufgezeigt, weshalb Startups besonders anfällig auf technische Schulden sind und worum es sich bei diesen technischen Schulden handelt. Ebenfalls wird verdeutlicht, welche Ansätze es gibt, um diese technischen Schulden zu managen und das Entrepreneurial Software Engineering Model (ESEM) wird präsentiert. Im Anschluss werden in Kapitel 3 verwandte Arbeiten vorgestellt, die ähnliche Themen aus dem Forschungsbereich der technischen Schulden aufgreifen. Daraufhin wird in Kapitel 4 erklärt, wie vorgegangen wurde, um relevante Informationen aus der wissenschaftlichen Literatur zu extrahieren und wie diese Daten eingesetzt wurden, um einen Lösungsvorschlag zu formulieren. Durch dieses Vorgehen wurde die, in Kapitel 5 enthaltene, Checkliste erstellt. Diese definiert und priorisiert die unterschiedlichen TD-Typen, nennt Ursachen, Effekte und Indikatoren für jeden sowie beispielhafte Methoden, die eingesetzt werden können, um sie entweder vorzubeugen, zu messen oder zu reduzieren. Darauf folgt eine Erweiterung des ESEM im Kapitel 6, bei der das Modell um die fehlenden Aspekte erweitert wird, die es ermöglichen, technische Schulden in Startups unter Kontrolle zu halten. In der anschließenden Diskussion wird auf die Forschungsfragen geantwortet und Neuheiten der Arbeit erklärt. Ebenfalls werden darin eventuelle Einschränkungen beschrieben sowie weiterführende Fragen beim Umgang mit technischen Schulden in Startups. Letztendlich werden die Ergebnisse der Arbeit im abschließenden Kapitel 8 zusammengefasst.

# 2. Grundlagen

## 2.1 Startups

Der Begriff Startup ist oft nicht klar definiert, er beschreibt nicht genau, welche Unternehmen nun als solches betrachtet werden können und wieso diese wahrscheinlich Kandidaten für die Entstehung von technischen Schulden sind.

### 2.1.1 Was ist ein Startup?

Der Deutsche Startup Monitor (DSM) ist eine landesweite Studie und stellte 2023 die Entwicklungen in 1.825 Startups deutschlandweit dar [3]. Dieser definiert Startups als Unternehmen, die jünger als zehn Jahre sind, ein hohes Mitarbeitenden- und/oder Umsatzwachstum anstreben und hoch innovative Technologien und/oder Geschäftsmodelle einsetzen [3]. Die zwei wichtigsten Merkmale, die Startups somit von anderen Unternehmensgründungen unterscheiden, sind Skalierbarkeit und Innovativität [17]. Beide Faktoren führen zu Schwierigkeiten, die bei größeren, bereits bestehenden oder kleinen Unternehmen zwar existieren, aber bei Weitem nicht so schnell ausgeprägt sind wie bei Startups.

Eines der ersten Vorkommen des Begriffs Startup in der SE-Literatur, kann bei Carmel [18] im Jahre 1994 vorgefunden werden. Dieser hatte sich damit beschäftigt, mehrere kleine Softwareunternehmen zu analysieren und äußerte schon damals Beschreibungen, die noch heute eine Vielzahl aller Startups betreffen. Die Time-to-Completion, also die Zeit, die benötigt wird, um gewisse Aufgaben durchzuführen, sollte möglichst gering sein, sodass das Unternehmen sich einen Namen schaffen kann und konkurrenzfähig bleibt. Allerdings werden in solchen kleinen Unternehmen oft gewisse Prinzipien vernachlässigt, die für einen strukturierten Ablauf essenziell sind. Es werden oft keine konkreten Entwicklungsprozesse eingesetzt oder Software-Tools, die gewisse Aufgaben vereinfachen oder automatisieren könnten. Risikoanalysen werden so gut wie keine durchgeführt und ein organisiertes Projektmanagement ist auch nur selten vorhanden.

### 2.1.2 Herausforderungen von Startups

Eine weitere Charakterisierung von Software-Startups kann durch die Herausforderungen gewonnen werden, denen sie in ihrem Alltag begegnen [1]. Sutton hat hierfür vier Hautprobleme genannt [19], die den Startups Schwierigkeiten bereiten können:

- **Begrenzte Ressourcen**: Startups steht nur eine begrenzte Menge an Ressourcen zur Verfügung, sodass sie sich meistens nur auf essenzielle Aufgaben beschränken, da der Aufwand sonst den Fortschritt vom Startup gefährden könnte.
- **Vielseitige Einflüsse**: Aus unterschiedlichen Richtungen kommen Wünsche an das Produkt, sei dies von Kunden, Investoren oder Partnern. Diese Akteure verüben Druck auf das Unternehmen, da all deren Ziele erfüllt werden müssen, diese aber oft gegenseitig im Widerspruch stehen.
- **Wenig Betriebserfahrung**: Startups haben meistens nur sehr wenig Erfahrung, daher kann es, durch Carmel [18] beschriebenen, zu Mangel an konkreten Prozessen, Organisation und Management führen.

- **Sich ständig ändernde Technologien und Märkte**: Um eine Erfolgschance zu haben, muss oft mit neuartigen und somit noch wenig bekannten Technologien gearbeitet werden. Dies verspricht ein hohes Potenzial, birgt aber auch zahlreiche Probleme.

Diesen vier Hauptproblemen fügen sich noch spezifischere Komplikationen hinzu, die jedem Startup eigen sind und darin dementsprechend mehr oder weniger Ausprägung aufzeigen.

Wie bei den unterschiedlichen TD-Typen im Detail gesehen wird, wird der menschliche Aspekt oft unterschätzt. Oft werden Software-Startups durch junge Unternehmer gegründet und schnell Teams zusammengestellt, um so rasch wie möglich mit der Entwicklung beginnen zu können. Dies zieht allerdings oft mit sich, dass im Unternehmen dann ein Mangel an Qualifikation und Kompetenz herrscht, der weitreichende Folgen auf die technischen Schulden im Unternehmen haben kann [20]. Diese Umstände können zu suboptimal entwickelter Software führen sowie zu einem Zeit- und Ressourcenverlust [20].

Startups nehmen in ihrer Anfangsphase diese Risiken oft trotzdem in Kauf, da sie sich direkte Ergebnisse erhoffen und eine mangelhafte Vorstellung davon haben, welche Auswirkungen technische Schulden auf ihr Unternehmen haben könnten [21]. Sollte sich das Produkt allerdings etabliert haben und somit die Wachstumsphase des Unternehmens beginnen, kann die Gründlichkeit, mit der technische Schulden minimiert wurden, hohe Auswirkungen auf die Zukunft des Startups haben. Die Anhäufung von technischen Schulden, vor allem in einem jungen Unternehmen, nicht ernst zu nehmen, kann nämlich eine Entwicklungskrise verursachen, die die Wachstumsrate verlangsamen oder gar anhalten könnte und im schlimmsten Fall ein Aufgeben des Projektes verursacht [21].

### 2.1.3 Definition und Potenzial von Startups

Es gibt also mehrere Möglichkeiten Startups und deren Gefahren zu beschreiben, die es ermöglichen verschiedene Aspekte zu betrachten und sich dadurch ein Gesamtbild zu verschaffen. In dieser Arbeit wird sich, zur Übersicht, auf die Definition von Ries [22] geeinigt. Diese besagt, dass ein Startup eine menschliche Einrichtung ist, die das Ziel hat, ein neues Produkt oder eine Dienstleistung unter Bedingungen von extremer Unsicherheit zu liefern, was die Wahrscheinlichkeit technische Schulden anzuhäufen, deutlich erhöht.

Startups sind Unternehmen, die ein enormes Potenzial bieten, sei es für die Mitarbeitenden selbst oder für die Wirtschaft. Allein von 2017 bis 2023 ist unter anderem die Anzahl an sogenannten „Unicorns“, also Startups mit einer Unternehmensbewertung über einer Milliarde US-Dollar, in Deutschland von 2 auf 33 gestiegen [3]. Solch ein Erfolg, wie bei den deutschen Startups Flix oder N26, bleibt allerdings äußerst selten, da die meisten Startups leider schon nach zwei Jahren scheitern [23]. Die Resultate hängen unter anderem davon ab, wie gut Startups Schulden vermeiden oder einsetzen können, ob finanziell oder eben technisch.

Startups haben einige Merkmale mit kleinen Unternehmen gemeinsam und weisen eine Kombination verschiedener Faktoren auf, die das Entwicklungsumfeld von etablierten Unternehmen deutlich unterscheiden. Das dynamische Umfeld, in dem sich Startups befinden und unterschiedliche Kontexte je nach Unternehmen, führen dazu, dass es bis jetzt nur sehr wenige Studien gibt, die das Thema von technischen Schulden in Startups beschreiben [21]. Notwendige Maßnahmen, um technische Schulden in diesen Unternehmen zu vermeiden, werden daher nur sehr selten ergriffen.

## 2.2 Technische Schulden

1970 wurde durch Lehman [24], in einem seiner Gesetze zur Evolution von Software, eines der wichtigsten Probleme bezüglich eines Mangels an strukturierten Code genannt. Er beschrieb, dass wenn „sich ein System weiterentwickelt, seine Komplexität zunimmt, es sei denn es wird etwas dagegen getan“ [24]. Diese Komplexität reduziert im Laufe der Zeit die Leistung des entwickelten Produkts und des Unternehmens. Zahlreiche andere Studien befassten sich mit Richtlinien für optimale Softwareentwicklung, wodurch ein neues Konzept entstand, das bis heute immer ausführlicher behandelt wird.

### 2.2.1 Entstehung des Begriffs

Das erste konkrete Erwähnen des Begriffs Technical Debt geschah im Jahre 1992 durch Ward Cunningham, als er Verstöße gegen gute Architektur und Coderichtlinien, als Schulden bezeichnete [25]. Dieser beschrieb, dass Code zum ersten Mal ausliefern, ähnlich ist wie sich zu verschulden, da geringe Schulden die Entwicklung beschleunigen, solange sie schnell wieder zurückgezahlt werden. Er erwähnt allerdings, dass die Gefahr darin besteht, die Schulden nicht zurückzuzahlen. Die gesamte Zeit, die damit verbracht wird, nicht optimalen Code zu produzieren oder ihn zu korrigieren, da er zum Beispiel schlecht strukturiert wurde, verursacht sogenannte Zinsen [26], die ebenfalls zurückgezahlt werden müssen. Die Zinsen von technischen Schulden nicht zurückzuzahlen, und die Komplexität der Software unkontrolliert ansteigen lassen, erhöht also drastisch die Wahrscheinlichkeit des Scheiterns des Projekts oder dessen Stillstand.

Ein einfaches Beispiel hierfür, wird durch Power [27] beschrieben. Dieser hat die Effekte von technischen Schulden auf die Geschwindigkeit mit der Features entwickelt werden beobachtet. Um dies zu bewerkstelligen hat er in einem großen Softwareunternehmen analysiert, welchen Einfluss der Mangel an TD-Management auf die Produktivität eines Unternehmens haben kann. Wenn kein Aufwand betrieben wird, um technische Schulden zu kontrollieren und man sich nur auf das Entwickeln konzentriert, erhöht sich die Menge an technischen Schulden in jedem neuen Release. Die technischen Schulden erreichen dann den Punkt, an dem sie höher sind als die Anzahl an neuen Features pro Release und somit dringendst abgebaut werden müssen.

### 2.2.2 Klassifizierung der TD-Arten

Es gab mehrere unterschiedliche Ansätze, um zu versuchen, die verschiedenen Arten an technischen Schulden zu klassifizieren. Einer wurde durch Fowler [28] beschrieben, der technische Schulden nach vier Charakteristiken klassifiziert hat, je nachdem ob die Schulden bewusst oder unbewusst und mit Vorsicht oder durch Rücksichtslosigkeit verursacht wurden. Dies stellt den sogenannten TD-Quadranten dar.

Ein anderer Ansatz ist der von McConnel [29]. Dieser schlägt vor, dass technische Schulden entweder nur eine unbeabsichtigte Folge eines Mangels an Erfahrung oder falscher Annahmen sind oder, dass sie, wie durch Cunningham beschrieben, absichtlich eingegangen wurden, um schneller Ergebnisse zu erreichen [26].

Was diese beiden Ansätze und zahlreiche andere allerdings nicht beachten, ist die Natur der technischen Schulden [30]. Die Natur stellt dar, welche Aktivität des Entwicklungsprozesses zur Entstehung dieser Schulden geführt hat. Daraus entstehen sogenannte TD-Typen, von denen nun einige, der durch Alves et al. [30] definierten, beschrieben werden:

- **Design Debt**: Praktiken, die gegen die Grundsätze eines guten, objektorientierten Designs verstoßen. Ein Beispiel wären das Schreiben von sogenannten Gottklassen, welche zu viel Funktionalität beinhalten und somit unübersichtlich werden.
- **Architecture Debt**: Probleme, die in der Projektarchitektur auftreten, beispielsweise eine Verletzung der Modularität, welche Auswirkungen auf die Leistung und Sicherheit haben kann.
- **Test Debt**: Mangelhafte Anzahl an Tests oder eine suboptimale Testdurchführung, wie zum Beispiel das spontane, satt systematische Ausführen von Tests.
- **Process Debt**: Einsatz von ineffizienten Prozessen, wie zum Beispiel ein älterer Prozess der nach vielen Jahren nicht mehr angebracht ist.
- **People Debt**: Probleme, die mit den Teammitgliedern verbunden sind, so wie etwa ein Mangel an Kommunikation zwischen Entwicklern und Managern.

Diesen TD-Typen fügen sich noch zahlreiche andere hinzu, die in Kapitel 5 detailliert beschrieben werden.

### 2.2.3 Konzeptuelles Modell der technischen Schulden

Um aus diesen Informationen ein einheitliches Bild von technischen Schulden zu schaffen, haben sich 33 Forscher aus dem Bereich der technischen Schulden zusammengesetzt. Sie haben, wie es in Abbildung 1 zu sehen ist, ein konzeptuelles Modell erstellt, welches das Zusammenspiel von Artefakten, Aktivitäten und Elementen im Rahmen der Entstehung von technischen Schulden in einem Unternehmen darstellt [12].

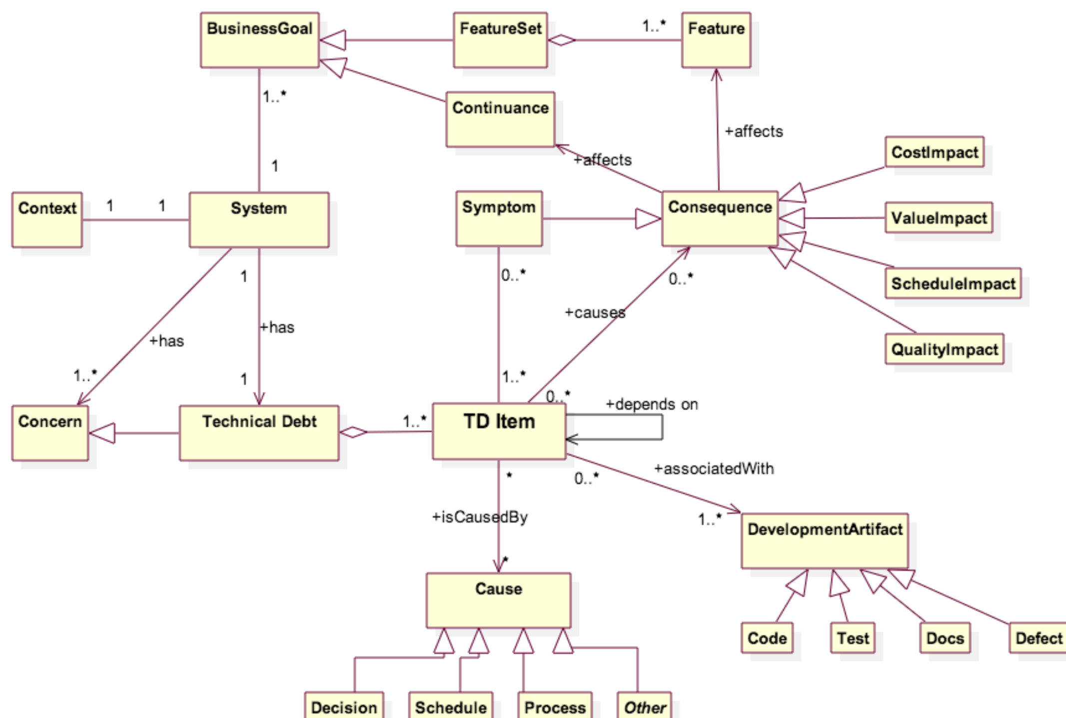


*Abbildung 1: Konzeptuelles Modell für TD [12]*

Das Modell von Avgeriou et al. [12] stellt dar, dass ein System viele unterschiedliche Probleme (Concerns) haben kann und weshalb technische Schulden darin eine zentrale Rolle spielen. Technischen Schulden bestehen aus sogenannten TD-Items, die direkt von konkreten Artefakten des Softwareentwicklungsprozesses abhängen. Diese können zum Beispiel Code-bezogen sein, wie die

zuvor beschriebenen Gott-Klassen. Diese Items haben verschiedene Ursachen wie unangemessene Entscheidungen, Zeitpläne, Prozesse oder andere Mängel, die in Kapitel 5 ausführlicher erklärt werden. Die Konsequenzen eines TD-Items haben Auswirkungen auf die Features und können vielfältig sein und sowohl die Kosten von zukünftigen Änderungen, den Wert des Systems, den Zeitplan oder die Qualität des Systems beeinflussen. Diese Effekte können dazu führen, dass die Ziele des Unternehmens verfehlt werden und die Aufrechterhaltung des Systembetriebs, die sogenannte Continuance, gefährdet wird. Dieses Modell ist somit äußerst vollständig, doch es fehlen die nötigen TD-Management Aktivitäten, die durchgeführt werden können, um die technischen Schulden zu kontrollieren [12], sodass sich mit diesen befasst werden muss.

## 2.3 Technical Debt Management

Heutzutage verfügen viele Unternehmen über keinerlei konkrete Praktiken, um technische Schulden zu managen und Manager sowie Entwickler und andere betroffene Teammitglieder, sehnen sich nach Methoden, um ihre Schulden abzubauen [12]. Um die zuvor genannten Probleme zu vermeiden, bedarf es dementsprechend an Methoden und Techniken, die das Auftauchen von technischen Schulden minimieren oder die Auswirkungen von bereits entstandenen Schulden reduzieren. Hierfür kann das sogenannte TD-Management eingesetzt werden.

### 2.3.1 Was ist TD-Management?

Auch für dieses gibt es keine einheitliche Definition, sondern mehrere verschiedene Sichtweisen, die es ermöglichen, sich ein Gesamtbild zu verschaffen. Yli-Huumo et al. [15] beschreiben, dass TD-Management durchgeführt wird, um technischen Schulden während der Softwareentwicklung vorzubeugen, zu messen und zu senken, indem Prozesse, Techniken und eventuell Werkzeuge verwendet werden. Shull et al. [31] erwähnen, dass, um TD-Management durchführen zu können, ein Unternehmen sich darüber einig sein soll, welche Softwarequalitäten am wichtigsten sind, wie zum Beispiel Sicherheit oder Usability. Ebenfalls muss das Unternehmen wissen, wo genau die größten Probleme in der Software vorhanden sind.

### 2.3.2 Hauptprobleme beim TD-Management

Dem TD-Management stellen sich zudem zahlreiche Schwierigkeiten in den Weg, was bis heute dazu führt, dass viele verschiedene Methoden durch Forscher vorgeschlagen werden, die auf unterschiedliche Weisen, ähnliche Probleme lösen wollen. Li et al. [32] beschreiben, dass die größten Schwierigkeiten beim TD-Management darin liegen, abzuschätzen wie viele technische Schulden im aktuellen System vorhanden sind, wie sie sich entwickeln werden und welche Effekte sie in Zukunft haben werden. Um dies zu vervollständigen werden durch Power [33] 7 Hauptprobleme beim Managen von technischen Schulden genannt: (1) sich darauf einigen was als TD betrachtet werden kann; (2) TD quantifizieren; (3) TD visuell darstellen; (4) TD im Laufe der Zeit beobachten; (5) die Auswirkungen der Vernachlässigung von TD über mehrere Versionen hinweg messen; (6) erkennen zu welchen Defekten TD führt und letztendlich; (7) verstehen zu welchen Kosten das Verschieben von TD-Management führen kann.

### 2.3.3 Management-Kategorien der technischen Schulden

Einer der Gründe, weshalb die Metapher der technischen Schulden angenommen wurde, ist, dass sie genau dies ermöglicht, sie zudem einfach zu verstehen ist und weitgehend akzeptiert wird. Probleme, die bei der Entwicklung auftauchen sowie deren Konsequenzen, werden wie finanzielle Schwierigkeiten betrachtet.
Daraus ergeben sich zwei wichtige Kernkonzepte, um technische Schulden zu kategorisieren:

- **Das Schuldenkapital** (Principal): Dieses stellt die Kosten dar, die notwendig sind, um technische Schulden zu entfernen. Die Kosten sind der Aufwand, der dafür betrieben werden muss, wie zu Beispiel durch Aktivitäten wie neue Testfälle schreiben oder Code Refactoring [9]. Das Kapital können daher ebenfalls die Kosten sein, die gespart werden, sollten die technischen Schulden nicht behoben werden [12].
- **Die Zinsen** (Interest): Wie auch durch Cunningham [25] erwähnt, sind die Zinsen die Kosten, die sich im Laufe der Zeit ansammeln, je länger die Schulden nicht zurückgezahlt werden [9]. Die Kosten sind hier der erhöhte Aufwand, um die technischen Schulden zu beseitigen und die reduzierte Produktivität als Folge der Schulden [9].

Diese zwei Konzepte ermöglichen es TD-Management Ansätze, wie die im Abschnitt 5.3.8 beschriebene Einschätzung der TD-Kosten, durchzuführen und erlauben es somit dem Management, einen Überblick über das Ausmaß der technischen Schulden im Startup zu gewinnen.

### 2.3.4 TD-Management Aktivitäten

Um technische Schulden zu vermeiden, werden durch Li et al. [34] und Yli-Huumo et al. [15] ebenfalls acht verschiedene TD-Management Aktivitäten definiert. Jede beschreibt einen konkreten Aspekt des Management-Prozesses, bei dem entweder die Entstehung neuer Schulden verhindert wird oder vermieden wird, dass Schulden durch bereits existierende entstehen:

- **TD-Identification:** Bei der Identifizierung von technischen Schulden sollen Schulden erkannt werden, die sowohl durch bewusste als auch unbewusste technische Entscheidungen verursacht wurden.
- **TD-Measurement:** Hierbei geht es darum, die Kosten und den Nutzen von bekannten, technischen Schulden oder die Menge an Schulden im gesamten System zu schätzen.
- **TD-Prioritization:** Diese dient dazu, je nach gewissen definierten Regeln TD-Items zu priorisieren, um bei der Entscheidung zu helfen, welche als erstes behandelt werden sollten.
- **TD-Prevention:** Wie der Name es sagt, geht es bei TD-Prevention darum, das Erscheinen von TD-Items vorzubeugen.
- **TD-Monitoring:** Beim Monitoring wird sich damit befasst, die Veränderungen der Kosten und des Nutzens von ungelösten technischen Schulden im Laufe der Zeit zu beobachten.
- **TD-Repayment:** Dieses hat als Ziel, TD-Items zu beseitigen oder zu entschärfen, um die technischen Schulden zurückzuzahlen.
- **TD-Documentation:** Die Dokumentation von technischen Schulden bietet eine Möglichkeit letztere auf einheitlicher Weise darzustellen, um somit die Anliegen der Stakeholder adressieren zu können.

- **TD-Communication:** Bei der Kommunikation von technischen Schulden geht es darum, diese den Stakeholdern, wie zum Beispiel Investoren sichtbar zu machen, damit sie besprochen und gemanagt werden können.

All diese Aktivitäten ermöglichen es daher, technische Schulden auf unterschiedliche Arten zu vermeiden, sei dies nun präventiv oder korrektiv, um den Aufbau von Schulden auf bereits Existierende zu vermeiden.

### 2.3.5 Optimales TD-Management

Wenn ein Team TD-Management also effizient einsetzt, soll es in der Lage sein, triviale Probleme, wie zum Beispiel mit der Codequalität, von wirklichen technischen Schulden zu unterscheiden [35]. Es soll informierte Entscheidungen treffen und technische Schulden priorisieren sowie Kompromisse eingehen können [35], um nicht den einfachsten Weg zu nehmen, sondern den, der die technischen Schulden unter dem definierten Grenzwert hält. Doch wie in Kapitel 5 zu sehen sein wird, ist eine einheitliche Herangehensweise für TD-Management, vor allem für Startups, nicht möglich, sodass aus einer Bandbreite an Methoden, unternehmensspezifische Auswahlen getroffen werden müssen. Geeignete Entwicklungsprozesse können das TD-Management für Startups jedoch deutlich vereinfachen.
Doch auch graphische Vorgehensmodelle können es Startups, vor allem in ihrer Anfangsphase bei der Entwicklung von Prototypen, ermöglichen, einen Überblick über gute Methoden und Praktiken zur Vermeidung von technischen Schulden zu bekommen.

## 2.4 Der Softwareentwicklungs-Lebenszyklus

Der Softwareentwicklungs-Lebenszyklus beschreibt die unterschiedlichen Phasen, die bei der Entwicklung eines Softwareprojektes durchlaufen werden [36]. Diese Phasen sind zwar, je nach Projekt, leicht verschieden, lassen sich aber im Generellen folgendermaßen zusammenfassen:

- **Planung**: Hierbei geht es darum, die Anforderungen der Stakeholder und Kunden zu sammeln, um eine Spezifikation der Softwareanforderungen zu erstellen und benötigte Ressourcen zu planen [36].
- **Design**: Es werden danach die Anforderungen analysiert, um die besten Lösungen zur Erstellung der Software zu identifizieren [36].
- **Implementierung**: Die Anforderungen werden umgesetzt, indem das Entwicklungsteam das Softwareprodukt entwickelt [36].
- **Testen**: Es werden sowohl automatisierte als auch manuelle Tests durchgeführt, um die Software auf Störungen, Fehler, dessen Qualität und andere projektspezifische Aspekte zu testen [36].
- **Bereitstellung**: In dieser Phase wird die Software den Kunden zur Verfügung gestellt, indem sie von der Entwicklungs- in die Produktionsumgebung verschoben wird [36].
- **Wartung**: Die letzte Phase dient dazu, eventuelle Störungen zu beheben, Kundenanliegen zu lösen, Softwareänderungen zu verwalten und die Software zu überwachen, um Verbesserungsmöglichkeiten zu identifizieren [36].

Diesem Zyklus werden in Kapitel 5 die entsprechenden TD-Typen zugeordnet.

## 2.5 Das Entrepreneurial Software Engineering Modell

Software-Startups erhoffen sich Erfolg durch innovative Ideen und haben dementsprechend mehrere Annahmen, die zum Beispiel den Nutzen ihres Produkts betreffen [37]. Eine Annahme könnte zum Beispiel sein, dass eine App ein Problem löst, das auch wirklich gelöst werden muss und Kunden einen Mehrwert bringen wird. Um zu überprüfen, ob diese Annahmen auch gerechtfertigt sind, müssen sie dementsprechend durch das Startup getestet werden, bevor dieses sein Produkt veröffentlicht. Da sonst ein hohes Risiko besteht, dass die angebotene Idee keine Kunden findet und das Startup scheitert. Um diese Frage zu beantworten, haben Brunner et al. [38] das sogenannte Entrepreneurial Software Engineering Modell (ESEM) entwickelt und darüber eine Arbeit [38] sowie ein Paper [37] verfasst, an denen sich die folgende Beschreibung im Wesentlichen orientiert.

### 2.5.1 Ziel des Entrepreneurial Software Engineering Modells

Wie es der Name verrät, basiert das Modell auf dem sogenannten Entrepreneurial Software Engineering. Dessen Ziel ist es herauszufinden, wie es möglich ist, effizient und effektiv software-intensive Systeme als Teil eines unternehmerischen oder innovativen Prozesses zu entwickeln. Um das zu bewerkstelligen, muss ab dem Anfang des Startup-Projektes sichergestellt werden, dass ein Prozess befolgt wird, der es ermöglicht, ein robustes Produkt aus dem SE-Blickwinkel zu entwickeln.

Das ESEM zielt darauf ab dies zu ermöglichen, indem es eine leichtgewichtige hybride Entwicklungsmethode vorschlägt, die Startups ein Framework zur Verfügung stellt [37]. Dieses Framework unterstützt Startups dabei, ein sogenanntes Single-Feature Minimum Viable Product zu entwickeln. Hybrid bedeutet in diesem Fall, dass die Entwicklungsmethode sowohl agile Aspekte als auch klassische SE-Methoden miteinander vereint, um eine, für Startups optimale, Herangehensweise zu gewährleisten. Das Single-Feature MVP ist ein Prototyp, der es ermöglicht, so viele Infos wie möglich über ein Produkt und die Kunden mit minimalem Aufwand zu sammeln. Single Feature soll bedeuten, dass nur eine einzige wichtige Funktionalität implementiert wird, die es zu evaluieren gilt. Als zum Beispiel der Prototyp der App Spotify getestet wurde, war das einzige vorhandene Feature, einzelne Lieder abspielen zu können [39]. Sich auf eine einzige Funktionalität zu beschränken hat den Vorteil, dass das MVP sehr schnell entwickelt werden kann und Feedback sowie Innovationszyklen schnell erstellt werden können.

### 2.5.2 Hauptphasen des ESEM

Abbildung 2 bietet einen Überblick über das gesamte Modell von Brunner at al. [37], das aus drei Haupt-Stages besteht:

1. **Feature Requirements Generation and Evaluation** (FRGE)
2. **Design und User Experience Evaluation** (DUXE)
3. **Pre-Launch Testing** (PLT)

Diese Stages beschreiben den Ablauf des Prozesses pro Feature und werden nacheinander, also inkrementell, durchgeführt, wie es das Evolutionary Prototyping von Floyd [40] beschreibt. Für jedes Teil-Feature, das nötig ist, um das Hauptfeature zu vervollständigen, wird der Zyklus erneut durchgegangen, sodass der Prozess ebenfalls iterativ ist.

Der Ablauf besteht aus unterschiedlichen Artefakten, welche als Input und Output für Stages dienen und die durch verschiedene Entwicklungsmethoden verbunden sind.

Der Ablauf beginnt mit einem sogenannten validierten Problem, also ein Problem das für die Kunden als wichtig bezeichnet wurde. Ein Beispiel wäre, dass es für potenzielle Kunden zu schwer ist, Produkte zu bestellen.

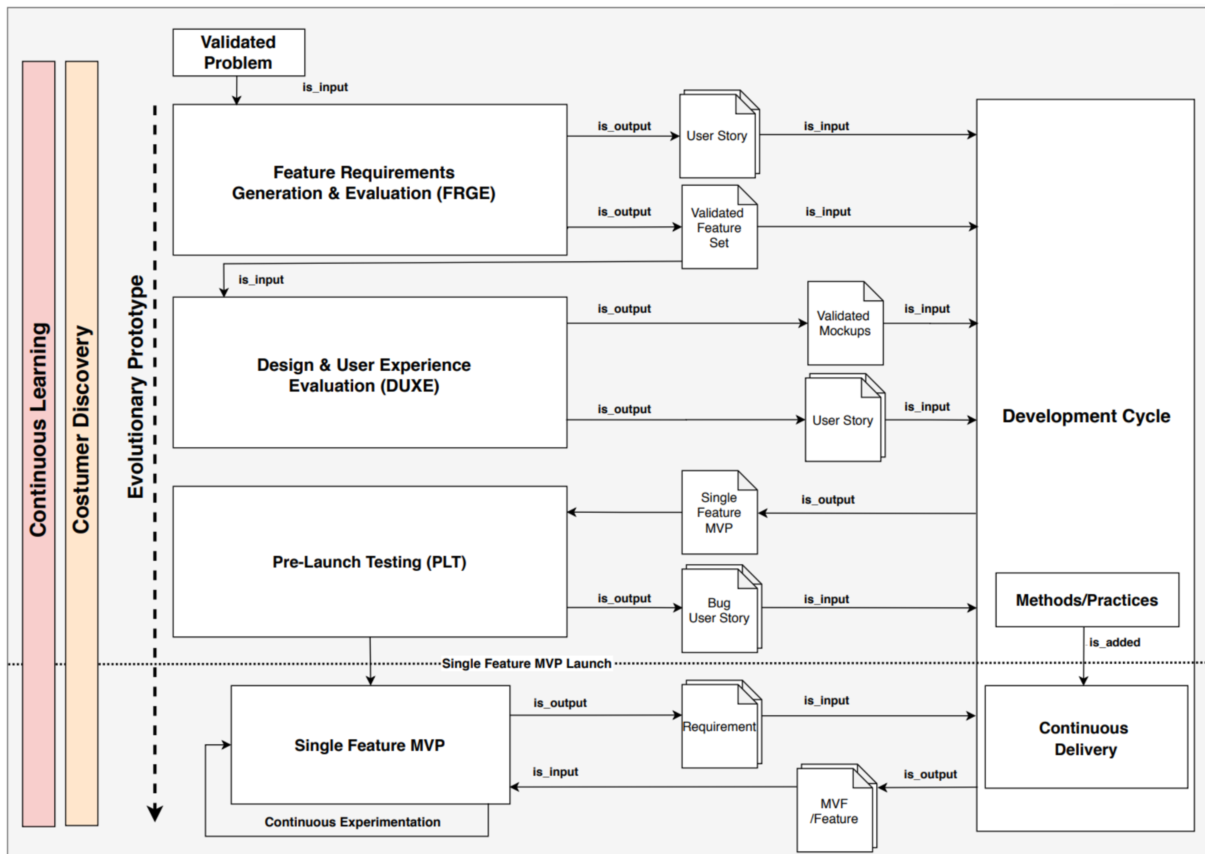


*Abbildung 2: Überblick über das hybride Entwicklungsmodell für Startups [38]*

Mit dem Ziel, sich ein Überblick über das ESEM zu verschaffen, werden nun kurz dessen einzelne Phasen beschrieben.

### 2.5.3 Feature Requirements Generation & Evaluation

Um ein Feature, wie die einfache Bestellung, für das MVP zu entwickeln, ohne eine unnötig hohe Menge an Aufwand zu betreiben, müssen die Teil-Features identifiziert werden, die auch wirklich notwendig und gewünscht sind, um dieses Feature zu vervollständigen. Mit diesem Aspekt wird sich in der Phase Feature Requirements Generation & Evaluation (FRGE) befasst, welche in Abbildung 3 dargestellt ist. Diese setzt sich aus einem Bild-Measure-Learn Zyklus zusammen, der selbst aus unterschiedlichen Phasen besteht.

Zuerst wird die, zuvor definierte, Problemstellung in einer kreativen Etappe analysiert. Hierbei geht es darum, zum Beispiel Brainstorming durchzuführen und diese Etappe dient dazu, eine konzeptuelle Lösung zu entwickeln. Im Falle unseres Beispiels, wäre dies beispielsweise eine Website mit Artikeln und der Möglichkeit diese zu bestellen. Anschließend wird aus dieser Idee eine gewisse Menge an Features durch Feature Mapping gewonnen. Hierbei geht es darum, in mehreren Schritten unterschiedliche Aufgaben der User und Bedingungen durchzugehen, um die Features zu erkennen, die sie abdecken würden. Anhand der gefundenen Features werden dann Wireframes erstellt. Diese stellen das Skelet und Layout des Systems dar, um eine Übersicht darüber zu bekommen, welche Elemente im Interface vorhanden sein werden [41]. Design, Farben und andere UX-Elemente, werden auf dem Minimum gehalten, da es hauptsächlich um den Inhalt geht [41]. Diese Wireframes werden in der folgenden Phase eingesetzt, um zum Beispiel durch sogenannte Early Evangelists, getestet zu

werden. Dies sind Personen, die schon sehr früh ein Interesse für das System zeigen und es während der experimentellen Phase bereits verwenden [42]. Durch die Experimentierphase kann herausgefunden werden, ob das Feature den Wünschen der Kunden entspricht. Um dies zu bewerkstelligen, werden die Ergebnisse analysiert und es ergeben sich zwei Möglichkeiten: Entweder erfüllt das Feature die nötigen Voraussetzungen und wird dementsprechend so beibehalten oder es entspricht nicht ganz dem Zielsystem und es wird ein neuer Build-Measure-Learn Zyklus gestartet. Dabei ermöglichen es die zuvor gewonnenen Ergebnisse, die sogenannten Validated Learnings, den folgenden Zyklus schneller und besser durchzuführen. Sobald das System angemessen ist, werden die validierten Features, aus denen es bestehen soll, ebenfalls in User Stories runtergebrochen, damit klare Arbeitsanweisungen für den Development Cycle vorhanden sind. Somit kann bereits mit der vertikalen Entwicklung gestartet werden.

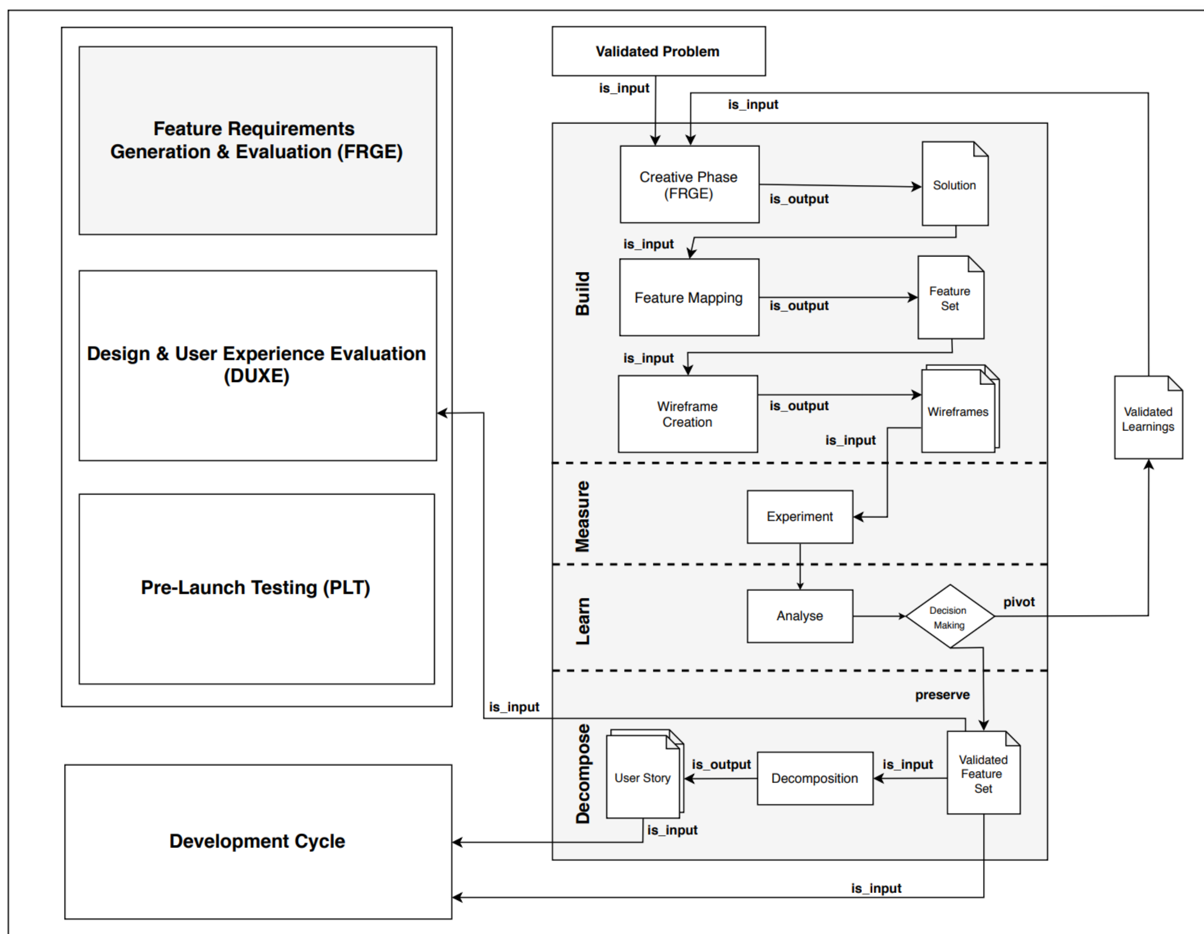


Abbildung 3: Detaillierte Ansicht des Stages FRGE [38]

Es ist wichtig, dass bereits das MVP eine gute User Experience gewährleistet, da der erste Eindruck für potenzielle Kunden von hoher Wichtigkeit ist. Dies kann die Erfolgschancen erhöhen, Marktvorteile einbringen und die Marketingkosten senken.

### 2.5.4 Design & User Experience Evaluation

Genau damit befasst sich die zweite Stage des ESEM; Design & User Experience Evaluation (DUXE), welche durch Brunner [38] in Abbildung 4 dargestellt wird. Diese ist ebenfalls als Build-Measure-Learn Zyklus strukturiert. Als Input bekommt DUXE die validierten Features aus FRGE, also die Features, die implementiert werden müssen, um das Haupt-Feature des Zyklus zu vervollständigen. Es folgt eine kreative Phase, die der aus FRGE ähnelt, nur, dass es hier darum geht, ein initiales Design zu entwickeln, um dem Feature ein gewisses Aussehen und Gefühl zu verleihen. Hier kann sich an bereits bestehenden Produkten inspiriert werden. In unserem Beispiel könnte sich zum Beispiel an bewährten Webseiten wie Amazon orientiert werden. Daraufhin wird erneut eine Wireframe Phase gestartet, nur, dass diese diesmal nicht statisch sind, sondern klickbare Mockups. Letztere ermöglichen es erneut Experimente mit Early Evangelists durchzuführen, um Feedback zu sammeln, indem sie mit den Mockups interagieren. Diese Experimente und die aus der Phase FRGE, führen dazu, dass während der gesamten Entwicklung des MVPs, immer dazugelernt wird, sei dies bezüglich Kunden oder Business Annahmen. Diese begleitende Phase namens Continuous Learning, wird in Abbildung 2 dargestellt. Zudem entdecken die Nicht-Entwickler durch die Early Adopters Probleme und eventuelle Lösungen, was durch die begleitende Phase Customer Discovery dargestellt wird.
Sollte die User Experience angemessen genug sein, wird in der Analyse entschieden die Mockups beizubehalten. Ist dies nicht der Fall, wird, wie bei FRGE, Dank der Validated Learnings ein neuer DUXE-Zyklus gestartet, bis die Tester mit dem Ergebnis zufrieden sind. Da die Mockups nur schwer als Anweisungen für die Entwicklung dienen können, werden diese erneut in User Stories heruntergebrochen, um übersichtliche Entwicklungsaufgaben zu liefern.

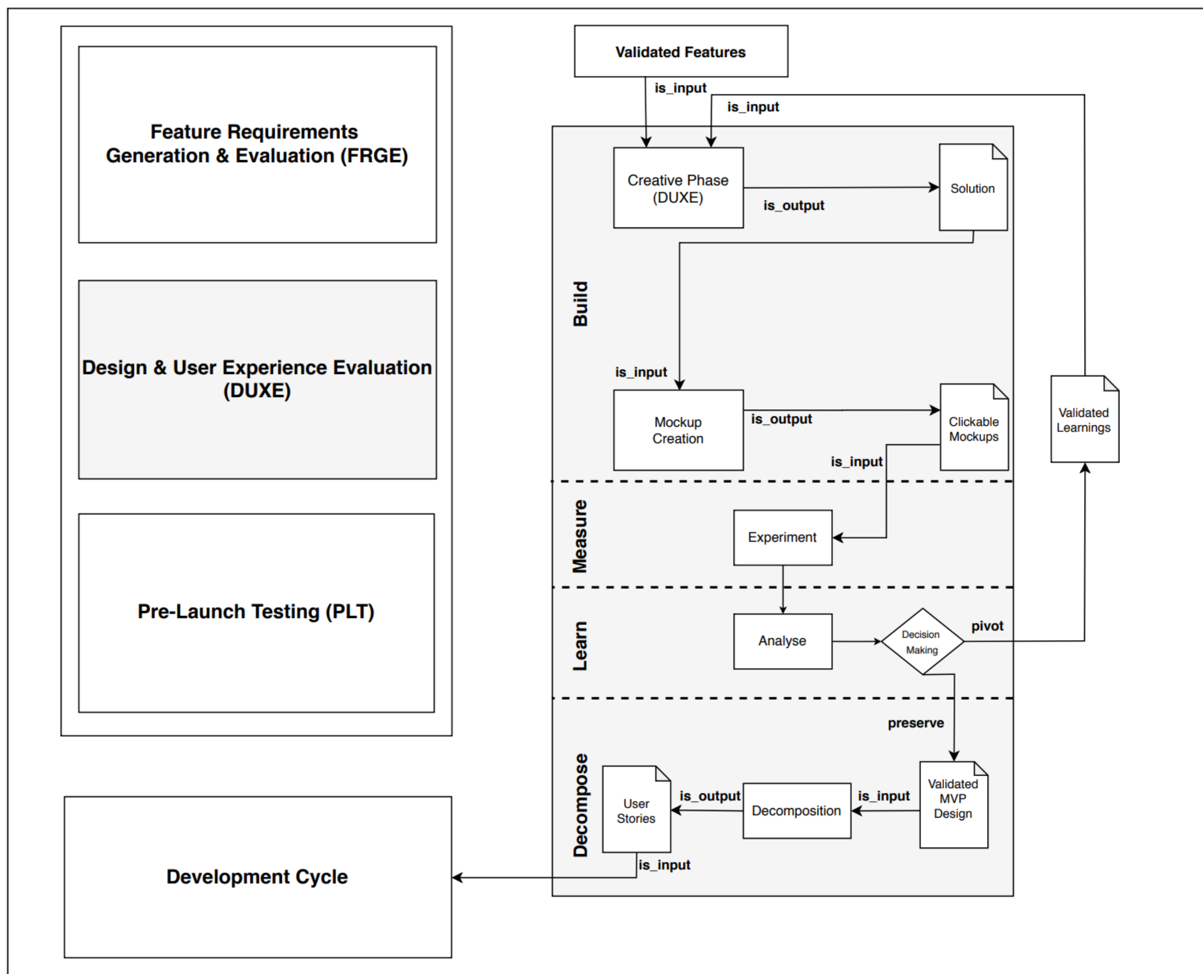


*Abbildung 4: Detaillierte Ansicht des zweiten Stages DUXE [38]*

### 2.5.5 Pre-Launch Testing

Während das Feature fertiggestellt wird, muss sichergestellt werden, dass dieses gewisse Kriterien erfüllt. Nicht alle Fehler müssen direkt korrigiert werden, da kleine Performanceprobleme und Bugs für einen ersten Release normal sind. Doch ist zum Beispiel die Usability bei der Veröffentlichung des Features nicht ausreichend oder ist die Software extrem langsam, kann dies erhebliche Konsequenzen haben. Die Meinung von sogenannten Early Adopters, also die ersten User des Produkts, kann dadurch nämlich negativ beeinträchtigt werden, sodass deren Kritik dem Ruf des Produkts unwiderruflich schaden könnte. Um dies möglichst zu vermeiden, werden beim Pre-Launch Testing (PLT), welches in Abbildung 5, dargestellt wird, eine Reihe an Tests verschiedener Arten durchgeführt. Hiervon einige Beispiele:

- **Security Testing**: Testen der Sicherheit des Systems durch automatisierte Penetrationstests.
- **Performance Testing**: Testen der Startdauer und der Ladezeiten innerhalb der Software.
- **Database Testing**: Testen der Datenintegrität und Performance der Datenbank sowie der Validität der Daten.

Sollte durch irgendeinen dieser Tests Probleme entdeckt werden, werden diese ebenfalls in Form von sogenannte Bug User Stories dokumentiert. Hierdurch ist es für die Entwicklung möglich, einen Überblick zu bekommen und die Probleme, während der Entwicklung zu beseitigen. Nachdem dies gemacht wurde, kann zudem durch die User Stories erneut geprüft werden, ob nun alles wie gewollt funktioniert.

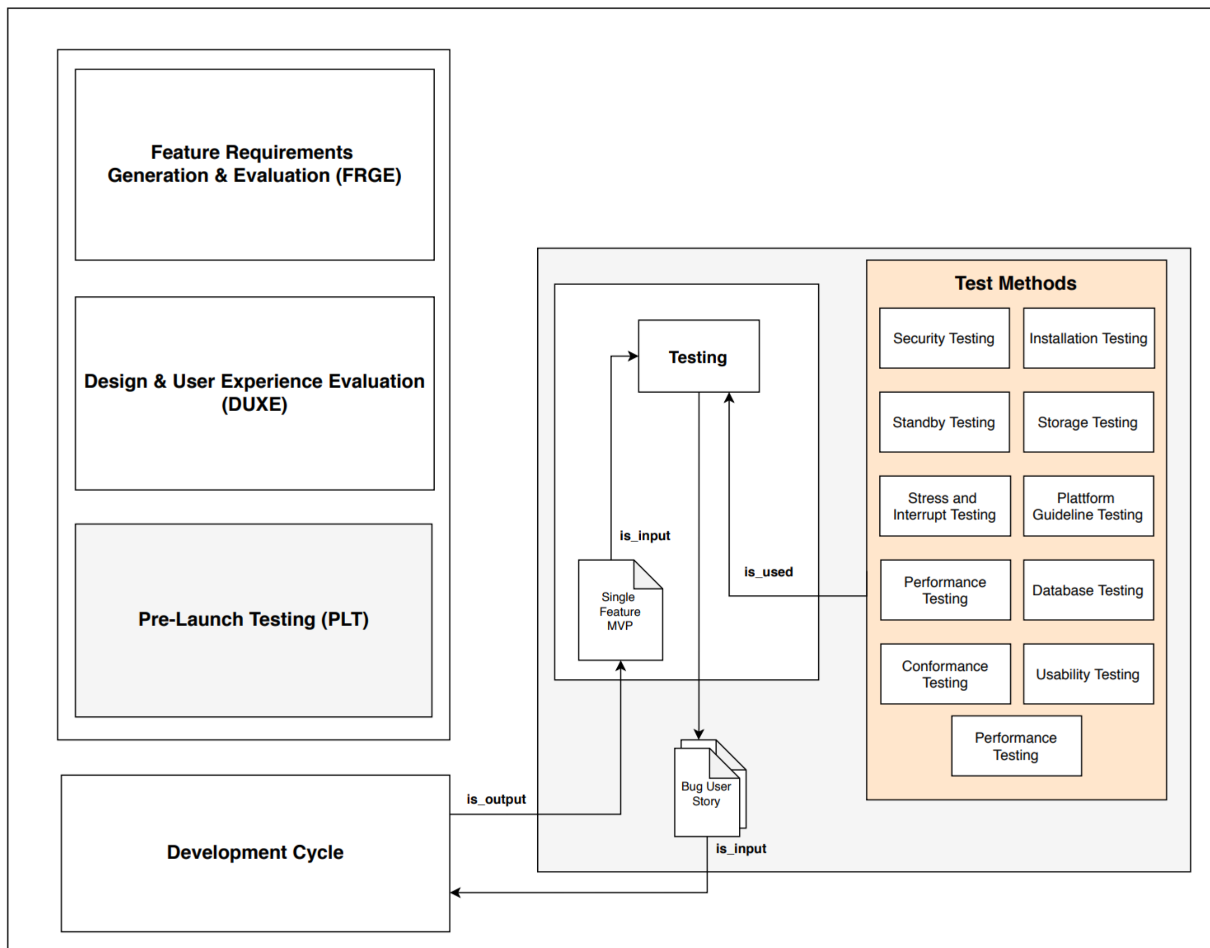


*Abbildung 5: Detaillierte Ansicht des dritten Stages PLT [38]*

### 2.5.6 Development Cycle

Die drei zuvor beschriebenen Phasen liefern die notwendigen Informationen für den begleitenden Development Cycle, welcher in Abbildung 6 dargestellt ist. In FRGE entstehen, wie dargestellt, eine Liste an benötigten Features und User Stories die diese beschreiben und durch DUXE werden User Stories erstellt die darlegen, wie die UX des Systems gestaltet werden soll. Somit entstehen Requirements an das Single Feature MVP, die, zusammen mit den Bug User Stories aus PLT, dem Feature Driven Development (FDD) übergeben werden. Das FDD besteht aus mehreren Schritten, wobei zuerst ein Modell des Systems erstellt wird, um eine Liste der nötigen Features zu erstellen, die dann gruppiert und priorisiert werden [43]. Der ganze Prozess fokussiert sich somit auf die benötigten Features und es entsteht eine Liste von zu implementierenden Features, die dazu dient, die Sprints zu planen. Pro Feature kann daraufhin, anhand derer Beschreibungen, eine User Story erstellt werden, die dieses beschreibt.

Als nächstes geht es darum die Features zu implementieren. Hierfür werden aus den User Stories Testfälle für das Test-Driven Development (TDD) abgeleitet. Dieses funktioniert so, dass für eine gewisse Funktionalität ein Test geschrieben wird, bevor der eigentliche Code geschrieben wird [44]. Der Code kann dann nach und nach verbessert und vervollständigt werden, indem darauf geachtet wird, dass die Testfälle erfolgreich bleiben. Dem TDD fügt sich Automated Testing hinzu. Hierfür wird eine Test-Software ausgewählt, welche eigenständig und parallel zur Entwicklung, selbständig Tests durchführt [45]. Dies vermeidet, dass Entwickler bei jeder Änderung dieselben Tests händisch durchführen müssen [45]. Durch das TDD und Automated Testing wird also das Single Feature MVP implementiert, welches somit die Requirements aus den User Stories erfüllt und den Beschreibungen der Features entspricht.

Die Implementierung geschieht, wie es weiterhin Brunner [38] beschreibt, mit dem Continuous Integration und Continuous Delivery Prinzip. Bei Continuous Integration geht es darum, dass jeder Entwickler seinen Code regelmäßig, zum Beispiel einmal am Tag, in das MVP integriert. Dieser wird durch das Automated Testing geprüft und jeder verfügt somit über den aktuellsten und gleichen Code. Nach dem Launch des MVPs wird der Development Cycle um die Continuous Delivery erweitert. Diese nutzt die kurzen Entwicklungszyklen, um zu garantieren, dass zu jedem Zeitpunkt zusätzliche Software, in Form von Updates, veröffentlicht werden kann. Dadurch kann das MVP konstant verbessert werden und konkurrenzfähig bleiben. Der Continuous Delivery fügt sich letztendlich zu der Continuous Experimentation hinzu. Durch Experimente, wie zum Beispiel mit Kunden, werden nach der Veröffentlichung des MVPs weitere, sogenannte Minimum Viable Features (MVF) definiert. Diese bestimmen, welche Anforderungen ein Feature mindestens erfüllen muss, damit es als funktionsfähig gilt.

Die MVFs können dann als Input für einen neuen Entwicklungszyklus in Form von neuen validierten Problemen eingesetzt werden. Aus Features entstehen also weitere Teil-Features und der inkrementell-iterative Zyklus wiederholt sich immer wieder, um das MVP zu vervollständigen. Zum Beispiel müssten Kunden in der Lage sein, ihre gekauften Produkte zu bewerten. Stages, die benötigt sind, werden also erneut durchgegangen, um das Produkt iterativ zu verbessern und Stages, die zum Beispiel dank der Validated Learnings unnötig sind, werden ausgelassen. Validated Learnings aus vorigen Iterationen, können also bei jeder neuen Iteration, eingesetzt und erweitert werden.

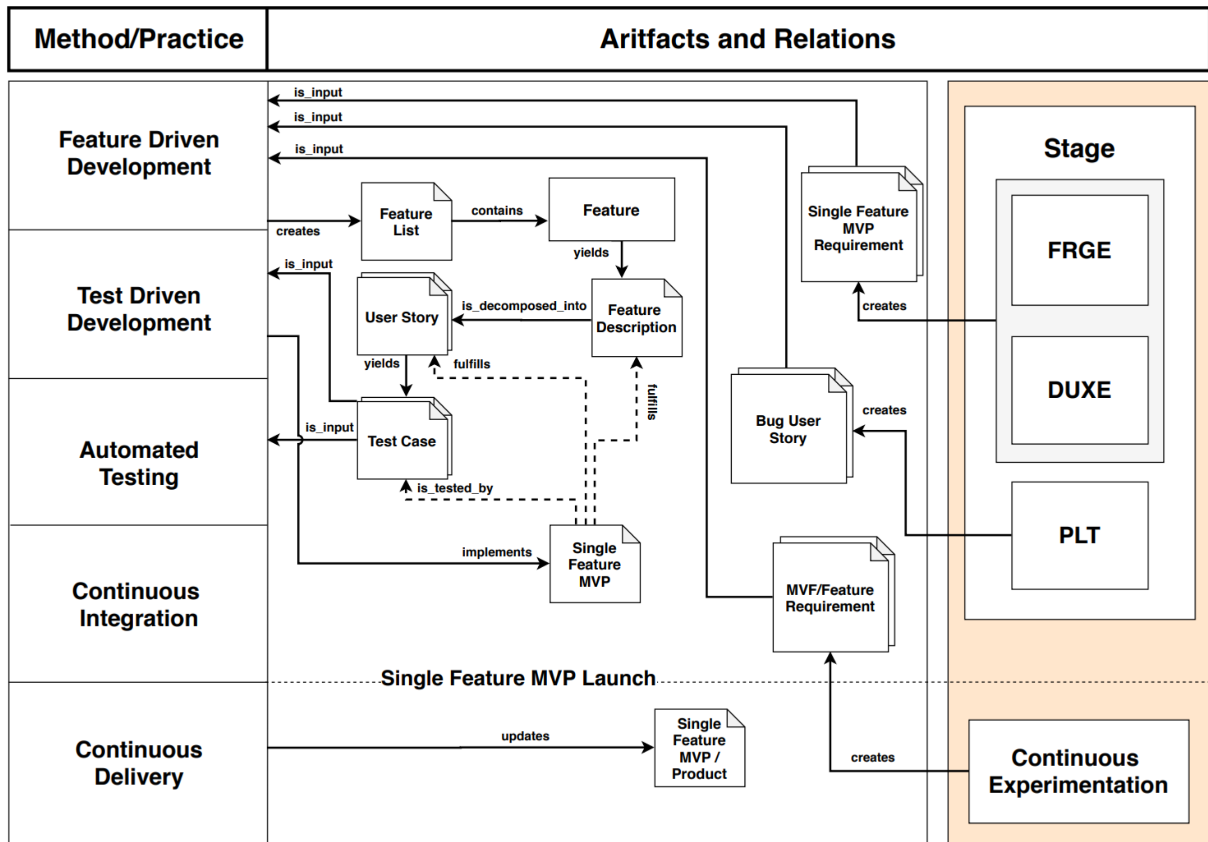


*Abbildung 6: Überblick über die Phasen des Startup-Modells, die Artefakte und den Fluss der Artefakte in Bezug auf die wichtigsten Entwicklungsmethoden [38]*

# 3. Verwandte Arbeiten

Einige Werke aus dem Forschungsbereich der technischen Schulden befassen sich mit ähnlichen Themen und wurden dementsprechend als Grundlagen für diese Arbeit verwendet.

## 3.1 Grundlegende Werke

Ab dem Jahre 2012 wurde die Forschung rund um technische Schulden aktiver, sodass immer mehr Studien sich damit befassten. Dies geschah allerdings, ohne ein gemeinsames Vokabular und Definitionen zu verwenden. Eine einheitliche Terminologie wurde somit 2014 durch Alves et al. [30] vorgeschlagen, welche den Grundstein dieser und voriger Forschungsarbeiten im Bereich der technischen Schulden darstellt. Diese haben eine systematische Literaturrecherche durchgeführt, um eine Ontologie der Begriffe rund um technische Schulden zu erstellen. Die wichtigste Erkenntnis aus dieser Ontologie war die erste Definition der, in Abschnitt 2.2 erwähnten, TD-Typen. Durch das Analysieren von unterschiedlichen Papern, wurde 13 Typen benannt und Definitionen für diese vorgeschlagen. Ebenfalls werden in diesem Paper bereits einige Indikatoren genannt, die charakteristisch für die jeweiligen TD-Typen sind.

Rios et al. [9] haben den aktuellen Stand der Forschung analysiert, indem sie 13 Studien bezüglich technischer Schulden in bestehenden Unternehmen aus den Jahren 2012 bis 2018 ausgewertet haben. Das Ziel war es, die bisher berücksichtigten Forschungsthemen zu identifizieren und Forschungsrichtungen sowie das bereits definierte praktische Wissen zu organisieren. Dadurch konnten Rios et al. [9] einen Überblick über die vorhandenen Studien verschaffen. Dank dieser Studie konnte ebenfalls ein konzeptuelles Modell für technische Schulden erstellt werden, welches in Abschnitt 2.2 vorgestellt wurde. Zudem wurden eigene Definitionen für 15 unterschiedliche TD-Typen vorgeschlagen sowie Situationen, in denen diese während der Softwareentwicklung, vorkommen könnten.

## 3.2 Erweiternde Studien

In den folgenden Jahren wurden diese Erkenntnisse durch eine breitgelegte, internationale Studie Namens InsighTD, erweitert [46]. Es handelt sich um eine Familie an weltweit verteilten Umfragen bezüglich technischer Schulden, die seit 2017 durchgeführt werden und an der Forscher von elf unterschiedlichen Ländern teilnehmen. Das Ziel der inkrementellen Studie ist es, eine offene und verallgemeinerbare Sammlung an empirischen Daten, über Ursachen und Effekte von technischen Schulden in Softwareprojekten, zu erstellen [47]. Um dies zu bewerkstelligen, wird die Studie kontinuierlich in jedem Land unabhängig repliziert, um somit immer mehr Informationen zu sammeln. Dank dieser Studie konnten sogenannte Cause-Effect-Diagramme erstellt werden, die in Kapitel 4 beschrieben werden und darstellen, welche Ursachen von technischen Schulden am häufigsten zu welchen Effekten führen.

Das, im Abschnitt 2.5 beschriebene, Entrepreneurial Software Engineering Model, stammt aus den Arbeiten von Brunner et al. [38] [37]. Grund für die Entstehung des Modells war, dass zwar zahlreiche Praktiken und Methoden für die Softwareentwicklung in Startups vorhanden waren, aber es an konkreten Richtlinien fehlte. Um das ESEM zu erstellen, wurde mit dem Startup Zippr zusammengearbeitet, welche eine App entwickelt hat, die die Gelegenheit bietet, Events und Aktivitäten mit Freunden zu organisieren. Die Erfahrungen des Startups sowie ein Expertenreview,

haben es ermöglicht, das vorgestellte iterative und inkrementelle Modell zu erstellen, welches Startups dabei unterstützen kann, ein MVP zu entwickeln.

Im Jahre 2023 wurde durch Abrahamsson et al. [48] eine Fallstudie in mehreren Tech-Start-ups in der Region Stockholm durchgeführt. Es ging darum herauszufinden, weshalb Startups technische Schulden anhäufen und wie damit umgegangen wird. Dies macht aus der Arbeit eines der wenigen Werke, welches sich mit technischen Schulden in Startups befasst. Die befragten Startups gaben oft nur wenig Acht auf technische Schulden, die sie erzeugen, was, wie es sich herausgestellt hatte, keine sofortigen Konsequenzen hatte, aber im Nachhinein zu sehr hohen Kosten führen konnte. Als Schlussfolgerung wurde daher gezogen, dass in Startups unbedingt TD-Management betrieben werden sollte. Ebenfalls wurde festgestellt, dass die Teamzusammensetzung sowie Persönlichkeitsmerkmale und die vorhandene Kompetenz innerhalb des Teams, eine zentrale Rolle beim Management von technischen Schulden spielen. Diese Faktoren müssen somit in Startups ebenfalls in Kauf genommen werden.

## 3.3 Checkliste und ESEM-Erweiterung

Diese Arbeit baut auf den gewonnenen Kenntnissen auf und erweitert sie, um die, in Abschnitt 1.3 gesetzten, Ziele zu erreichen.
Es wird eine Checkliste erstellt, die es Startups ermöglichen soll, einen Überblick über verschiedene Methoden zu bekommen, um TD-Typen zu managen. Hierfür werden die Typen priorisiert, je nachdem wie wichtig sie, nach der Analyse von Forschungstexten, geschätzt werden. Ebenfalls wird das ESEM um die Aspekte erweitert, die bis jetzt fehlen, um den Umgang mit technischen Schulden in Startups so optimal und übersichtlich wie möglich zu gestalten. Letztendlich wird sich damit befasst, wie wichtig das Managen von technischen Schulden in Startups ist und weshalb.
Das Ziel ist es somit, die existierende Literatur zu einer beispielhaften Anleitung für Unternehmen in ihrer Anfangsphase zusammenzufassen, damit diese möglichst wenig unter den Effekten von technischen Schulden leiden müssen.

# 4. Methodik: Literaturstudie

## 4.1 Ziel der Studie

Das Ziel dieser Arbeit ist es also, eine Literaturstudie durchzuführen, die einen Überblick über Forschungsarbeiten ermöglicht, welche sich mit technischen Schulden und Methoden, um diese zu bekämpfen, befassen. Dank der gewonnenen Erkenntnisse können dann beispielhafte Handlungsmöglichkeiten vorgeschlagen werden, um Startups zu zeigen wie vorgegangen werden kann. Doch es soll durch die Analyse der Literatur auch grafisch dargestellt werden, wie diese Erkenntnisse in einen Prozess integriert und umgesetzt werden können, um TD-Management durchzuführen.

## 4.2 Vorgehen

Ähnlich wie bei dem Management von technischen Schulden, geht es bei einer Literatursuche darum, eine gewisse Methodik anzuwenden. Somit kann nämlich verhindert werden, dass im Nachhinein unnötiger Aufwand entsteht, der, durch eine angemessene Vorbereitung, zu vermeiden gewesen wäre. Ebenso wurde das Vorgehen dem Thema spezifisch angepasst, da dies, durch die vielen TD-Typen, die behandelt werden, sehr breit ist. Somit muss sich auf gewisse Aspekte fokussiert werden.
Hierfür ist eine semi-systematische Literaturrecherche gut geeignet [49]. Bei dieser werden relevante Forschungstraditionen, die Auswirkungen auf das untersuchte Thema haben könnten, identifiziert [49]. Dieser Ansatz deckt ein breites Spektrum an Themen und Studien verschiedener Arten ab. Somit wird im Folgenden der Forschungsprozess beschrieben, um die Forschungsstrategie zu erklären, die die Auswahl der Methodik und gewisser Themen rechtfertig.
Die Forschungsarbeit bestand aus drei Hauptphasen, welche in Abbildung 7 dargestellt werden. Zusammenfassend wurde zuerst in der Vorstudie die relevante Literatur gesucht und selektiert, um sich dadurch einen Überblick zu verschaffen und die Forschungsfragen zu formulieren. Im Anschluss wurden, im Rahmen der Literaturanalyse, weitere Werke aus der gefundenen Literatur extrahiert sowie Informationen, die zur Lösung der Forschungsfragen beitragen können. Die Daten aus dieser Analyse wurden im Anschluss inhaltlich gegliedert. Letztendlich wurden die Erkenntnisse verschriftlicht und eingesetzt, um die TD-Management Checkliste zu erstellen, das ESEM zu erweitern und die Ergebnisse der Arbeit diskutiert.

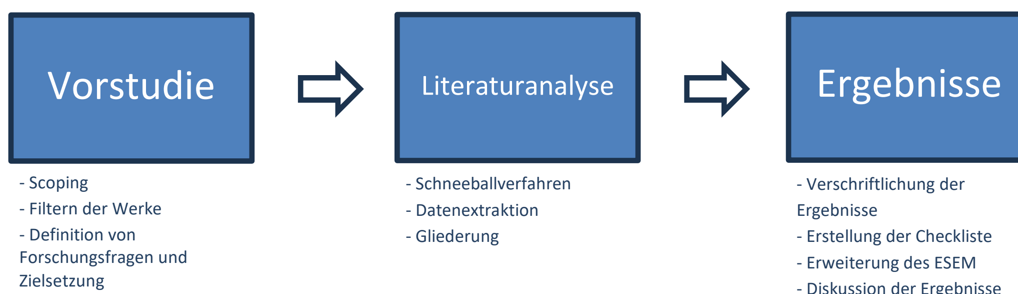


*Abbildung 7: Chronologischer Forschungsprozess*

### 4.2.1 Vorstudie

Beim ersten Schritt wurde sich damit befasst Scoping durchzuführen. Das Ziel des Scopings ist es, vor allem in einem heterogenen Umfeld, wie dem der technischen Schulden, sich einen Überblick über die existierenden Forschungsarbeiten und über die Themen, die im Bereich behandelt wurden [50], zu verschaffen. Dies gehört zu der, in Abbildung 7 dargestellten, Vorstudie.
Um die Suche durchzuführen, wurde Google Scholar verwendet, da dieses über eine hohe Anzahl an Forschungsarbeiten verfügt und auch in anderen Review-Papern bezüglich technischer Schulden, oft verwendet wird [9] [32].

#### Auswahlkriterien und Suchbegriffe

In dieser Phase wurden Paper ausgesucht die:

- peer-reviewt wurden,
- aus renommierten Fachbüchern oder Konferenzen stammen,
- oft in anderen Publikationen zitiert wurden,
- möglichst aktuell und somit repräsentativ für den aktuellen Stand der Forschung sind.

Hierfür wurden auf Google Scholar die entsprechenden Filter gesetzt und die, in Tabelle 1 dargestellten, Schlüsselwörter verwendet. Hierbei ging es darum, die Suche auf gewisse Themen zu fokussieren, da die Eingabe eines einzelnen Begriffes, zu viele irrelevante Ergebnisse geliefert hätte. Oft wurden etwas ältere Werke trotzdem beibehalten, da sie wichtige Erkenntnisse für die Arbeit mit sich bringen.

Veröffentlichungen wurden nicht ausgewählt sobald sie:

- kostenpflichtig waren,
- in keinerlei Hinsicht relevant waren, um die definierten Forschungsfragen zu beantworten,
- ältere Versionen von Dokumenten waren und neuere Werke vorhanden sind.

Ausnahmen für kostenpflichtige Paper wurden gemacht, sobald es eine sehr begrenzte Menge an Literatur bezüglich eines gewissen Themas gab. In diesen drei Fällen, wurden die Forschenden kontaktiert, die diese verfasst haben und nach einer Version des Dokuments gebeten. Alle drei Forschungsarbeiten wurden hierdurch erhalten.

*Tabelle 1: Eingesetzte Schlüsselwörter für die Literatursuche*

| Title | Thema | Operator |
|---|---|---|
| Technical Debt | Technical Debt | AND |
| - | Management | OR |
| - | Prevention | OR |
| - | Startups | OR |
| - | Types | OR |
| - | Tools | OR |
| Architecture Debt | Architecture Debt | - |
| Build Debt | Build Debt | - |
| Code Debt | Code Debt | - |
| Defect Debt | Defect Debt | - |
| Design Debt | Design Debt | - |
| Documentation Debt | Documentation Debt | - |
| Infrastructure Debt | Infrastructure Debt | - |
| People Debt | People Debt | - |
| Process Debt | Process Debt | - |
| Requirements Debt | Requirements Debt | - |

| | | |
|---|---|---|
| Security Debt | Security Debt | - |
| Service Debt | Service Debt | - |
| Test Automation Debt | Test Automation Debt | - |
| Test Debt | Test Debt | - |
| Usability Debt | Usability Debt | - |
| Versioning Debt | Versioning Debt | - |

#### Feinauswahl

Eine Schwierigkeit hierbei war, wie es Shull et al. [31] beschreiben, dass viele Paper dazu tendieren, den Begriff „Technical Debt“ in ihre Titel zu integrieren, selbst wenn die Verbindung zu TD sehr schwach ist. Da es bis zum heutigen Tag allerdings keine allgemein anerkannte Definition von technischen Schulden gibt, ist es schwer zu entscheiden, welche Werke nun als relevant bezeichnet werden können oder nicht. Um dem entgegenzuwirken, wurden die 74 ausgewählten Dokumente in dieser Phase kurz überflogen, indem deren Abstract und Zusammenfassung durchgelesen wurden. Somit konnten nur die relevanten Werke beibehalten und in einer organisierten Struktur klassifiziert werden. Jedes Werk wurde mit einem eindeutigen Titel und Tag gekennzeichnet und in einen Ordner mit ähnlichen Werken platziert, sodass diese sowohl durch ihre Kennzeichnung als auch thematisch, jederzeit wiedergefunden werden konnten.
Die Vorstudie hat es letztendlich ermöglicht die Forschungsfragen sowie eine konkrete Zielsetzung zu definieren, welche in Abschnitt 1.3 genannt werden.

### 4.2.2 Literaturanalyse

In der zweiten Phase wurde sich damit befasst, die vorausgewählten und klassifizierten Werke zu lesen, um relevante Informationen daraus extrahieren und strukturieren zu können.

#### Grundlagen

Die ersten Werke, welche in ihrer Gesamtheit durchgelesen wurden, waren die Arbeiten von Brunner et al. [38] [37] bezüglich des ESEM und die Arbeit von Abrahamsson und Holmqvist [48]. Diese verleihen nämlich einen Überblick über Themen, die von besonderer Relevanz sein könnten sowie Aspekte, die bereits bearbeitet wurden. Im Anschluss wurden Literaturreview Paper, wie das von Li et al. [34] oder Shull et al. [31], zum Thema technische Schulden gelesen. Da diese einen Überblick über den Stand der Forschung verleihen und eventuelle Lücken darin gut sichtbar machen.

#### Schneeballverfahren

Die ersten Informationen, die aus den anderen Werken gezogen wurden, waren Referenzen zu weiteren Werken, welche ähnliche Themen behandeln. Um dies zu bewerkstelligen, wurde das Schneeballverfahren [34] eingesetzt. Die Referenzen der vorselektierten Werke wurden hierfür, zum Teil oder vollkommen, auf andere relevante Arbeiten aus dem Forschungsbereich geprüft. Sobald dies gemacht wurde, wurden die Referenzen der neu gefundenen Werke ebenfalls auf relevante Werke geprüft. Dieser Prozess wurde wiederholt, bis immer öfters auf dieselben Werke gestoßen wurde. Dabei geschahen die Iterationen pro Thema. Zum Beispiel wurden durch die Vorstudie Dokumente bezüglich Design Debt gefunden, deren Referenzen dann auf andere Veröffentlichungen bezüglich Design Debt geprüft wurden. Sehr oft wurden durch dieses Vorgehen auch neue Themen entdeckt und Dokumente, die öfters vorkamen, konnten als zuverlässige Quelle angesehen werden. Ebenfalls hat es das Schneeballverfahren ermöglicht, auf Werke zu stoßen, die allein durch die präzisen Schlüsselwörter aus Tabelle 1 nicht aufgezeigt worden wären. Insgesamt wurden mit dem Schneeballverfahren 126 Forschungsarbeiten ausgewählt.
Sollten für ein gewisses Thema, mit den zuvor beschriebenen Methoden, trotzdem keine Werke mehr zu finden sein, wurden die gefundenen Werke auf Anhaltspunkte durchsucht. Ein Beispiel hierfür wäre

das Thema Versioning Debt, aus Abschnitt 5.3.7. Es wurde hierfür in der gefundenen Literatur nach Ursachen für diesen TD-Typ und im Anschluss nach Werken gesucht, die beschreiben, wie diese Auslöser vermieden werden können.

#### Gegliederte Zusammenfassung der Erkenntnisse

Anschließend wurden die Werke teilweise oder in ihrer Vollkommenheit studiert. Hierfür wurden die Forschungsfragen und die Titel der Abschnitte gelesen, um über deren Relevanz zu entscheiden.
Um die Inhalte der Dokumente klar aufzuteilen, wurden diese per Farbcode markiert. Hiermit war es möglich, pro Dokument, eine Übersicht über die behandelten Themen zu bekommen. Abschnitte mit einer höheren Komplexität wurden in den Werken selbst kommentiert, um die spätere Verschriftlichung der Ergebnisse, zu vereinfachen. Um sofort eine inhaltliche Übersicht über die, durch die Werke abgedeckten, Themen zu bekommen, wurde parallel zum Lesen eine Gliederung für die Arbeit erstellt. Sobald ein Dokument studiert wurde, wurde in dieser Gliederung aufgeschrieben, welcher Teil des Werks für welche Abschnitte der Arbeit relevant sein könnte. Um die in Kapitel 5 präsentierte Checkliste zu erstellen, wurden Referenzen zu den relevanten Teilen der Forschungsarbeiten, pro TD-Typ, in der Gliederung geordnet. Dieser Gliederung wurden ebenfalls erklärende Notizen beigefügt, um den Überblick über Struktur und Inhalt zu vereinfachen.

### 4.2.3 Ergebnisse

In der letzten Phase wurde sich damit befasst, aus dem Gelesenen Ergebnisse abzuleiten, um die definierten Forschungsfragen zu beantworten und das ESEM zu vervollständigen.

#### Auswahl der TD-Typen

Um die TD-Typen für die Checkliste auszuwählen, wurde teilweise ähnlich wie Rios et al. [9] vorgegangen und die Suche um einige Aspekte erweitert, die im Folgenden kurz beschrieben werden. Als erstes wurde nach TD-Typen gesucht die explizit als solches in den Forschungsarbeiten definiert wurden. Wenn Paper einen Typen nicht genau nannten, wurden diese nicht für die Auswahl der Typen in Betracht gezogen. Die Definition der Typen wurden verglichen und entweder die klarste ausgewählt oder eine zusammenfassende Definition geschrieben.

#### Ranking der TD-Typen

Nachdem sich für gewisse TD-Typen entschieden wurde, die im Anschluss vorgestellt werden, mussten diese in eine geordnete Reihenfolge gebracht werden. Die Priorisierung der TD-Typen hat als Ziel Startups zu zeigen, welche TD-Typen am wahrscheinlichsten in ihrem Projekt erscheinen könnten. Ebenfalls soll dies zeigen, welche technischen Schulden den größten Schaden anrichten könnten und somit als wichtiger betrachtet werden sollten.
Um die Priorisierung durchzuführen, wurden verschiedene Ansätze aus acht Papern kombiniert, um ein einheitliches Ranking über mehrere Werke hinweg zu erstellen. Hierfür wurden nur Forschungsarbeiten verwendet, die eine eigene Priorisierung der TD-Typen vorschlagen. Es wurde für alle TD-Typen, mittels eines Spreadsheets, ihre durchschnittliche Position in den Listen der Paper ausgerechnet. Da 16 TD-Typen ausgewählt wurden, ist der höchstmögliche Durchschnittswert dementsprechend 1 und der niedrigste 16. Diese Berechnung ermöglichte es herauszufinden, als wie wichtig die TD-Typen im Schnitt durch die Forschungsgemeinschaft angesehen werden.
In diesen Werken wurden die Typen nach unterschiedlichen Charakteristiken sortiert, wie zum Beispiel:

- wie oft sie in befragten Unternehmen im Durchschnitt vorkamen [51] [52],
- wie wichtig sie dem Management von Unternehmen erschienen [53],

- je nachdem wie oft sie durch Forschungsarbeiten studiert wurden [10] [9]
- je nachdem wie viele Management Methoden es für sie gibt [34],
- wie dringend sie beseitigt werden müssten [54],
- wie viele Ursachen und negative Effekte sie haben [55].

Das hierdurch errechnete Ranking wird in Abschnitt 5.3 verwendet, um die Checkliste an TD-Management Methoden zu sortieren.

#### Auswahl der Indikatoren von TD-Typen

Um eine Idee davon zu erhalten, welche TD-Items auf gewisse TD-Typen zurückführen, werden für jeden der TD-Typen aus der Checkliste beispielhafte Items, also Indikatoren, genannt. Dies hilft Startups beim Suchen und der Kategorisierung von gewissen Problemen. Um diese Indikatoren auszuwählen, wurden sie entweder aus Review-Papern mit entsprechenden Übersichten von Typen und ihren Items extrahiert oder aus Forschungsarbeiten, die einen TD-Typen und beispielhafte Erkennungsmerkmale von diesen beschreiben.

#### Kategorisierung der Ursachen und Folgen von technischen Schulden

Für jeden der TD-Typen gibt es eine Vielzahl an unterschiedlichen Ursachen und Effekten, sodass es Startups schwerfallen kann, einen Überblick darüber zu bekommen. Es musste dementsprechend ein Weg gefunden werden, diese zu kategorisieren und darzustellen.
Es wurden hierfür die Forschungsarbeiten von Rios et al. [47] eingesetzt, die an einer Lösung zu diesem Problem gearbeitet und dafür die Ergebnisse der InsighTD Studie verwendet haben. Wie in Kapitel 3 kurz erklärt, handelt es sich bei InsighTD um eine internationale Studie, bei der 2017 bis heute, weltweit Umfragen in 80 Softwareunternehmen rund um das Thema technische Schulden durchgeführt wurden. Die Antworten aus elf Ländern und von 513 professionellen Teilnehmern haben unter anderem zur Identifikation von 78 Ursachen und 66 Effekten von technischen Schulden geführt [46].
Rios et al. [47] haben die Ursachen und Folgen analysiert und, sobald diese auf ein ähnliches Problem zurückzuführen waren, wurden sie in eine entsprechende gemeinsame Kategorie eingeteilt. Die somit erhaltenen Kategorien sind: *Methodologie*, *Planung und Management*, *Entwicklungsprobleme*, *Qualitätsprobleme*, *Wartung*, *Menschen*, *Mangel an Fachwissen*, *externe Faktoren*, *Organisation* und *Infrastruktur*. Ursachen wie „Aufwand“, „Deadline“ und „Druck vom Projekt Manager“, können somit zum Beispiel der Kategorie *Planung und Management* zugewiesen werden.

### Identifizierung der wichtigsten Ursachen und Folgen

Um diese Ursachen und Probleme so übersichtlich wie möglich darzustellen, haben sich Rios et al. [47] entschieden, sogenannte probabilistische Ursache-Wirkungs-Diagramme einzusetzen, welche in dieser Arbeit eingesetzt werden. Das entsprechende Diagramm, für Design-Debt in einem kleinen Unternehmen, ist in Abbildung 8 dargestellt.

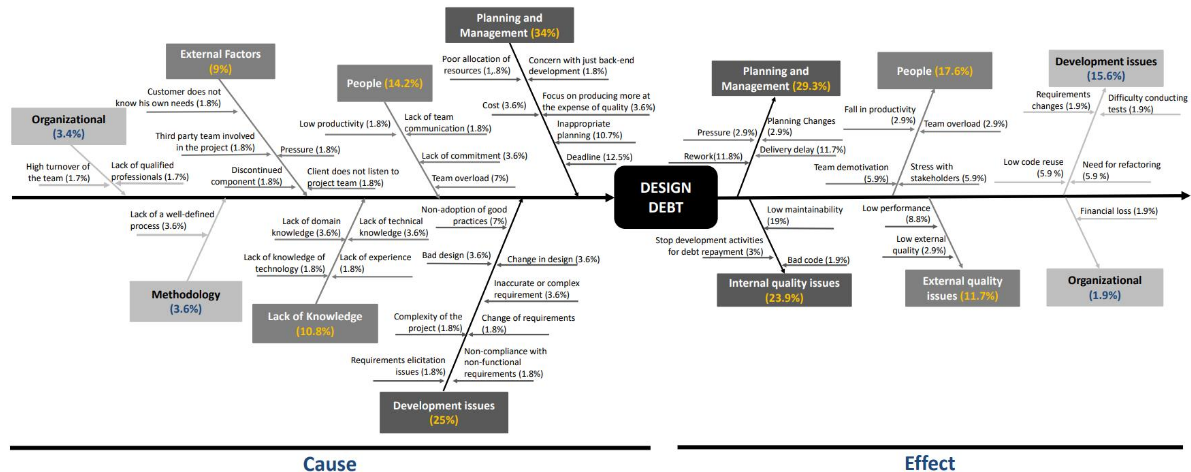


*Abbildung 8: Probabilistisches Ursache-Wirkungs-Diagramm für Design Debt [47]*

Diese Diagramme bieten eine visuelle Unterstützung bei der Analyse von Ursachen und Folgen, indem sie das, durch InsighTD gesammelte Wissen zu Problemen und deren Gründe und Effekte, darstellen. Die Prozentsätze wurden eingesetzt, um die wichtigsten Ursachen und Folgen für die Checkliste zu identifizieren. Sie zeigen an, wie wahrscheinlich jede einzelne Ursache zu Design Debt führen kann oder wie wahrscheinlich jeder Effekt auf das Team sein kann. Beispielsweise kann eine zu eng gesetzte *Deadline mit 12.5%* Wahrscheinlichkeit zu Design Debt im Projekt führen und diese Schulden können mit 29.3% Wahrscheinlichkeit die *Planung und das Management* des Softwareprodukts negativ beeinflussen. Die Prozentsätze für Ursachen und Effekte wurden erhalten, indem die Anzahl an Vorkommen bei denen zum Beispiel eine Ursache durch die Teilnehmer genannt wurde, durch die gesamte Anzahl an Vorkommen aller Ursachen dividiert wurde. Je näher sich die Kategorien der Ursachen und Effekte dem Zentrum des Diagramms befinden, desto wahrscheinlicher und dunkler dargestellt sind sie. Die wahrscheinlichsten Ursachen und Folgen wurden somit aus den Diagrammen extrahiert und in die Checkliste integriert, um eine verkürzte Übersicht darüber zu verschaffen.

### Auswahl der TD-Management Methoden

Um zu entscheiden welche Werke für das Management der einzelnen TD-Typen in Startups angebracht und relevant sind, mussten mehrere Faktoren beachtet werden. Wie bereits in Kapitel 1 und 2 erwähnt, verfügen Startups über nur wenig Ressourcen, seien diese zeitlich oder finanziell. Somit sollten die präsentierten Methoden übersichtlich gehalten werden, um Startups schnell eine Idee darüber verschaffen zu können, wie sie mit technischen Schulden umgehen können.

Zudem existieren, wie es sich durch die Literaturstudie herausgestellt hat, mehrere TD-Typen, für die es nur wenige oder keine konkreten Management Methoden gibt. Ebenfalls sind viele Methoden aus der Literatur deutlich zu komplex für einen direkten Einsatz in einem Startup. Allerdings existieren, wie in Abschnitt 5.5 erklärt wird, Tools, die sich teilweise mit dem Management dieser weniger studierten Typen befassen können. Es werden also für mehrere TD-Typen theoretische Methoden präsentiert, damit Startups verstehen können, was durch Tools ausgerechnet oder ermittelt wird. Somit können Startups für sich selbst entscheiden, ob der Lösungsansatz und dessen zugrundeliegenden Probleme auf sie zutreffen. Dadurch können Sie im Nachhinein, falls Interesse besteht, entscheiden, welche Methoden automatisiert und welche Methoden eher in House durchgeführt werden sollten. Um

Abwechslung in die Auswahl der Methoden zu bringen und zu zeigen, welche unterschiedlichen Ansätze existieren, wurden zum Beispiel rechnerische und graphische, aber auch Richtlinien und andere Ansätze ausgewählt.
Letztendlich muss, wie es Shull [56] und Zazworka et al. [57] beschreiben, jedes Startup für sich selbst entscheiden, welche technischen Schulden am ehesten vermieden werden sollten und wie, sodass es keine „one size fits all“ Lösung gibt. Die Checkliste dient somit als ein Überblick über mögliche theoretische und praktische Ansätze, die für das TD-Management eingesetzt werden können.

#### Erweiterung des ESEM

Das erweiterte ESEM wurde erstellt, indem die Erkenntnisse aus der Checkliste eingesetzt wurden, sodass sich diese zum Teil im Modell wiederfinden. Hierfür wurde zuerst, mit Hilfe des gewonnenen Wissens, analysiert, wo sich potenzielle Stellen der Entstehung von technischen Schulden befinden. Diese Stellen wurden dann für alle Phasen des Modells in der Software Lucidchart [58] markiert und kommentiert. Anschließend wurde das ESEM vervollständigt, indem Zusammenhänge zwischen dessen unterschiedlichen Teilen beachtet und Erweiterung sowohl graphisch als auch textuell beschrieben wurden.

# 5. Ergebnisse: Checkliste zur Vermeidung von technischen Schulden

Das Ziel des TD-Managements ist es, herauszufinden welche technischen Schulden zu welchem Zeitpunkt am besten entfernt werden sollten [59]. Um dies zu bewerkstelligen, besteht das TD-Management aus drei Hauptaktivitäten: dem Identifizieren, dem Überwachen und dem Zahlen von technischen Schulden [60]. Ein andere, Erkenntnis, die durch das TD-Management gewonnen werden kann, ist, dass nicht unbedingt alle technischen Schulden eine negative Auswirkung haben müssen [61]. In einigen Fällen kann es von Vorteil sein, gewisse Aspekte eines Projekts zu vernachlässigen, wenn diese unwichtig sind. Der Umgang mit technischen Schulden ist dementsprechend ein konstanter Balanceakt, bei dem kurzfristige Effizienz mit einer eventuellen, späteren Vergrößerung des Startups und des Produkts Hand in Hand gehen müssen [7]. Durch Management kann abgewogen werden, wie hoch die Risiken sind, die das Unternehmen eingehen kann, um die Entwicklung gefahrlos zu beschleunigen [62].
Die vorigen Kapitel haben gezeigt, dass es Startups nicht leicht fällt zu wissen, wie sie vorgehen müssen, um technische Schulden zu managen, da es ihnen nicht bekannt ist, welche Methoden und Praktiken dafür eingesetzt werden können. Aus diesem Grund geht es nun darum, anhand der studierten Literatur, den Startups einen Überblick über TD-Management in Form einer Checkliste zu geben. Diese beschreibt, welche Ursachen und Folgen unterschiedliche TD-Typen haben können und welche TD-Management Methoden dementsprechend eingesetzt werden könnten, um sie zu begrenzen. Um dies zu ermöglichen, müssen die Startups allerdings zuerst, als ersten Bestandteil der Checkliste, auf TD-Management vorbereitet werden.

## 5.1 Startups auf TD-Management vorbereiten

### 5.1.1 Maßnahmen

Martini et al. [63] beschreiben eine Reihe an vorbereitenden Maßnahmen, die in kleinen Unternehmen eingesetzt werden können, um ein Gelingen des TD-Managements wahrscheinlicher zu machen.

- Als erstes ist es wichtig eine Person zu nominieren, die als TD-Management „Champion“ agieren soll. Diese Person leitet das TD-Management und kontrolliert somit die Durchführung der Methoden. Da Startups aus sehr kleinen Teams bestehen, kann diese Aufgabe entweder durch Manager, erfahrene Entwickler oder eine andere höhere Rolle übernommen werden. Sollte genügend Personal vorhanden sein, um dies zu ermöglichen, kann pro Subsystem auch eine TD-verantwortliche Person ernannt werden.

- Ebenfalls muss durch Startups ein Zeitbudget für das TD-Management reserviert werden, da ansonsten die nötigen zeitlichen Ressourcen fehlen. In der Studie von Martini et al. [63] wurde ermittelt, dass dieses Budget 10 bis 30 Prozent der gesamten Arbeitsdauer eines Unternehmens entspricht. Am Anfang des Management-Prozesses kann das Budget nur grob geschätzt werden, doch es sollte lieber mehr Zeit eingeplant werden und der Bedarf im Laufe der Zeit angepasst werden.

- Als nächstes müssen entsprechende, finanzielle Mittel für das TD-Management freigeschaltet werden. Dies kann, ähnlich wie beim zeitlichen Budget, bewerkstelligt werden, indem am Anfang und kontinuierlich durch das gesamte Projekt hindurch, dem Management kommuniziert wird, weshalb TD-Management unentbehrlich ist. Eine Möglichkeit dies zu bewerkstelligen wäre, zum Beispiel dem Management die Ursachen und vor allem die Folgen von technischen Schulden vorzustellen, die in der Checkliste beschrieben werden.

- Letztendlich sollte auch ein Workshop organisiert werden, indem durch die, für das TD-Management verantwortliche Person, den Teammitgliedern erklärt wird, woraus technische Schulden bestehen, welche TD-Management Methoden ausgewählt wurden und wer diese ausführen soll. Ebenfalls sollten die Mitglieder eigene Bedenken und Verbesserungsvorschläge äußern. Somit sind die Verantwortlichkeiten und eingesetzten Verfahren bekannt und das Management kann angestoßen werden.

Andere Maßnahmen und Details zu dem eben Beschriebenen ergeben sich erst, indem mit dem Management begonnen wird und eventuelle Schwierigkeiten abgearbeitet werden.

### 5.1.2 Schwierigkeiten

Bei der Einführung von TD-Management kann es in kleinen Unternehmen, wie den Startups, zu Schwierigkeiten kommen, die es durch die Checkliste zu beheben gilt.

- Das Hautproblem besteht laut Martini et al. [63] darin, das Management davon zu überzeugen, dass es sich lohnt TD-Management durchzuführen. Vor allem für TD-Items mit hohen negativen Auswirkungen fällt es dem Management schwer, Ressourcen freizugeben, um sie zu beheben, da dafür mehr finanzielle und zeitliche Mittel benötigt werden. Die Mitarbeiter möchten Wege haben, um zu zeigen, dass es sich lohnt technische Schulden abzubauen, doch es fehlt ihnen an übersichtlichen Ressourcen, um dies zu erreichen.
- Den Unternehmen fällt es zudem schwer, die unterschiedlichen technischen Schulden zu priorisieren, um zu entscheiden, welche nun als erstes bearbeitet werden sollten [63]. Hierdurch werden TD-Typen abgearbeitet, die von niedriger Wichtigkeit sind und eigentlich schwerwiegendere Probleme unbewusst aufgeschoben.

Es muss also durch die existierende Literatur sowie durch Erfahrungen von erfolgreichen und gescheiterten Unternehmen, definiert werden, wie vorgegangen werden kann, um technische Schulden unter Kontrolle zu halten. Somit würden Startups auf diesen Kenntnissen ihr eigenes TD-Management aufbauen können.

## 5.2 Typen an technischen Schulden

Wie es Rios et al. [9] beschreiben, ist der erste Schritt zur Verwendung von TD-Management Methoden zu wissen, welche verschiedenen Typen von technischen Schulden es gibt und wo diese in einem Projekt erscheinen können.
Es gibt, wie in Kapitel 1 und 2 erwähnt, verschiedene Ansätze, um technische Schulden zu klassifizieren. Sei dies, zum Beispiel je nachdem ob die Schulden freiwillig oder unbeabsichtigt entstanden sind [29] oder ob sie rücksichtslos oder vorsichtig [28] erzeugt wurden. Viele verschiedene Werke nennen ihre eigenen TD-Typen, sodass es an Klarheit für Startups fehlen würde. Es muss sich also auf gewisse Typen beschränkt werden und eine Übersicht der TD-Typen mit einheitlichen Definitionen erstellt werden.
Es werden dementsprechend TD-Typen berücksichtigt, die den Phasen des Softwareentwicklungsprozesses entsprechen, in denen sie vorkommen, da diese am ausführlichsten im Bereich der TD-Forschung studiert wurden. Durch Klassifizierungen je nach der Natur der Schulden, wie zum Beispiel Design Debt und Test Debt, fällt es zudem allen Rollen einfacher, sich eine Idee über deren Form und Ursprung zu verschaffen [31]. Ebenfalls wird es dadurch klar, dass unterschiedliche Maßnahmen nötig sein werden, um sie zu behandeln.
Zusätzlich zu diesen Typen, welche nach Phasen aus dem Entwicklungszyklus benannt wurden, sind in den letzten Jahren auch immer mehr Typen erschienen, die Teilen des Objekts entsprechen, in dem

sie vorkommen, wie zu Beispiel das Team oder das Projekt. Einige davon werden, entsprechend der in Abschnitt 4.2.3 genannten Methodik, ebenfalls ausgewählt und deren Management in der Checkliste erklärt.

## 5.3 Checkliste

Dank der gewonnenen Erkenntnisse und der Auswahl wird nun die Checkliste erstellt, die Startups dabei helfen soll, ihre technischen Schulden zu managen. Hierfür werden die priorisierten TD-Typen jeweils definiert, um zu wissen, worin diese genau bestehen. Im Anschluss werden die Hauptursachen und Effekte der jeweiligen TD-Typen erklärt sowie Indikatoren genannt, durch welche sie identifiziert werden können. Anschließend werden beispielhafte Methoden erklärt, um diese Schulden zu erkennen, zu vermeiden oder zu reduzieren. Diese Checkliste hat also, wie in Abschnitt 4.1 erklärt, als Ziel, Startups einen Überblick über Praktiken aus dem Forschungsbereich des TD-Managements zu verleihen, um diese selbst anzupassen und anzuwenden. Die eventuelle Auswahl von TD-Management Tools wird durch die Erklärung einiger Methoden, die diese einsetzen, ebenfalls erleichtert.
Eine detailliertere Version der Checkliste, welche tiefer auf die Ursachen und Folgen sowie die genauen Funktionsweisen der Methoden eingeht, ist im Anhang A wiederzufinden. Besteht bei Startups Interesse einen gewissen TD-Typen zu managen, bietet diese ausführlichere Version zusätzliche Erklärungen.

Aus dem in Abschnitt 4.2.3 beschriebenen Ranking und der in Abschnitt 5.2 beschriebenen Auswahl, ergibt sich folgende Reihenfolge, in welcher die TD-Typen in der Checkliste bearbeitet werden:

1. Design Debt
2. Code Debt
3. Test Debt
4. Architecture Debt
5. Requirements Debt
6. Documentation Debt
7. Versioning Debt
8. Defect Debt
9. Build Debt
10. Process Debt
11. Infrastructure Debt
12. People Debt
13. Usability Debt
14. Test Automation Debt
15. Service Debt
16. Security Debt

### 5.3.1 Design Debt

#### Beschreibung

Beim TD-Typen, der insgesamt am wichtigsten bewertet wurde, handelt es sich um die sogenannte Design Debt. Design Debt bezeichnet Schulden, die durch die Analyse des Quellcodes erkannt werden können, indem Merkmale im Code entdeckt werden, die gegen die Grundsätze von gutem objektorientiertem Programmieren verstoßen [30] [64]. Diese Art an technischen Schulden erscheint während der Implementierungsphase der Softwareentwicklung.

#### Ursachen und Folgen

Wie durch Rios et al. [47] dank der InsighTD Studie beschrieben, handelt es sich bei den häufigsten Ursachen von Design Debt um eine zu enge Deadline, unangemessene Planung, die Nichtübernahme von bewährten Praktiken und ein überlastetes Team. Praktiken wären hierbei zum Beispiel Peer Programming oder Code Reviews. Bei den wichtigsten Folgen handelt es sich um eine niedrige

Wartbarkeit und Überarbeitungen der Software. Eine genauere Beschreibung der Ursachen und Folgen ist im Anhang A.1 wiederzufinden sowie durch das Ursache-Wirkungs-Diagramm in Abbildung 22.

### Indikatoren

Es gibt zahlreiche Indikatoren, die auf die Präsenz von Design Debt zurückführen. Davon sind einige:

- **Gottklassen**: Klassen, die zu viel Funktionalität beinhalten und dadurch unübersichtlich werden [30].
- **Code-Smells**: Bei Code Smells kann es sich um zahlreiche Probleme handeln, wie zum Beispiel zu stark verschachteltem oder unnötigem Code [65].
- **Grime**: Bezeichnet Code, der nicht dem verwendeten Design Pattern entspricht [66].
- **Komplexe Methoden**: Methoden, die unstrukturiert geschrieben wurden oder zu umfangreich sind [9].
- **Data Clumps**: Lange Parameterlisten, die immer wieder in verschiedenen Methoden im System auftauchen [67].

Im Falle von Design Debt wird sich damit befasst, Gottklassen zu vermeiden, da diese oft und schnell zu einem wiederkehrenden und zeitraubenden Problem werden können [68].

### Erste Methode: Ursachen- und Effekt-Analysemeetings

Die in Abschnitt 4.2.3 beschriebenen Ursache-Wirkungs-Diagramme ermöglichen es, wie es Rios et al. [47] beschreiben, sogenannte Ursachen- und Effekt-Analysemeetings durchzuführen. Diese Meetings sollen dazu dienen, im Softwareprojekt vorhandene TD-Items zu besprechen und deren Ursachen und Effekte zu identifizieren. Um dies zu bewerkstelligen, kann das in Abbildung 9 dargestellte Vorgehen befolgt werden: Drei bis fünf Teammitglieder, die mit technischen Schulden zu tun haben, sortieren die TD-Items, die ihnen Probleme bereiten, anhand der hier präsentierten Checkliste, je nach ihrem entsprechenden TD-Typen. Anschließend verwenden sie das Ursache-Wirkungs-Diagramm des entsprechenden TD-Typen, um damit die Ursachen und Folgen zu erkennen, die zu den TD-Items im Projekt geführt haben. Die Diagramme können dann durch die Teammitglieder entsprechend für das Startup angepasst werden.

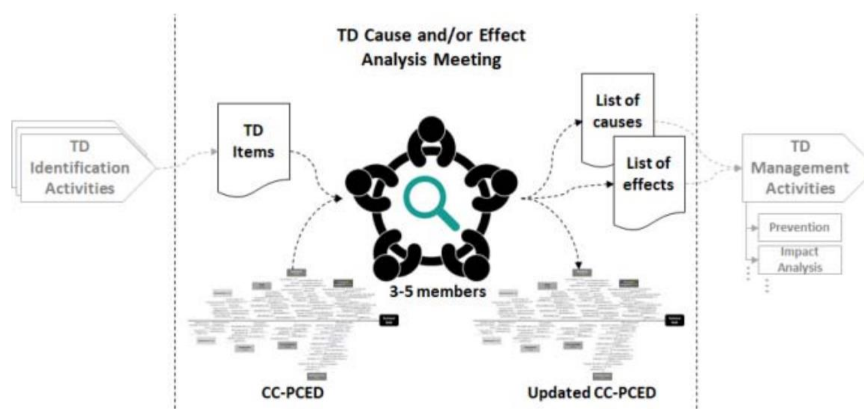


*Abbildung 9: Durchführung von Ursachen- und Effekt-Analysemeetings [47]*

Die, durch die Meetings erhaltenen, Listen an Ursachen und Effekten und das geupdatete Diagramm können im Anschluss verwendet werden, um sich für gewisse TD-Management Methoden aus der Checkliste zu entscheiden oder angemessene Tools auszuwählen. In der Studie von Rios et al. [47] behaupteten zudem 89% aller Teilnehmer, dass sie die Diagramme als hilfreich empfanden und dass sie dadurch zusätzliche und effizientere Ursachen und Effekte erkennen können.

Somit stellen die Diagramme nicht nur die Ursachen und Folgen von 14 der 16 präsentierten TD-Typen dar, sondern bieten mit den Ursachen- und Effekt-Analysemeetings eine bewehrte Methode, um diese Typen unternehmensspezifisch zu analysieren und zu managen.

### Zweite Methode: Cost-Benefit Analyse

Diese Methode befasst sich mit den Gottklassen, die detaillierte, beschriebene Klassen sind, die viele Verantwortlichkeiten in sich bündeln und viele Objekte steuern und überwachen [68]. Zudem befassen sie sich mehr mit Daten anderer Klassen als mit ihren eigenen und integrieren im Grunde die gesamte Funktionalität einer Anwendung, was sie sehr unübersichtlich macht [68].

Dementsprechend haben sich Zazworka et al. [68] eine Methode ausgedacht, die es ermöglicht das Refactoring von gewissen Gott-Klassen zu priorisieren, je nachdem wie viel Aufwand (Cost) dies benötigen würde und wie hoch der erbrachte Mehrwert (Benefit) sein würde. Dies ist also ein Beispiel einer Cost-Benefit Analyse. Um diese Werte zu berechnen, werden, wie im Anhang A.1 detaillierter erklärt, entsprechende Grenzwerte definiert und Metriken berechnet.

Dank dieser Metriken können die Klassen dann in eine sogenannte Cost-Benefit Matrix platziert werden, die durch ein kurzes Beispiel erklärt wird.

Durch die Berechnung der Werte für die Gottklassen in einem Projekt eines kleinen Unternehmens durch Zazworka et al. [68] ergab sich die, in Abbildung 10 dargestellte, Cost Benefit Matrix.

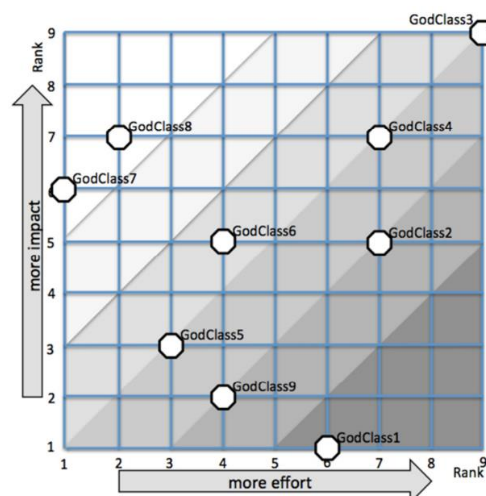


*Abbildung 10: Design Debt Cost Benefit Matrix für Gottklassen in einem Projekt [68]*

Um zu entscheiden welche Klassen nun am ehesten umgeschrieben werden sollten, wird geprüft, bei welchen dies möglichst einfach geschehen könnte und den höchsten Mehrwert erbringen würde. Gottklasse 1 wäre zum Beispiel ein schlechter Kandidat, da dessen Refactoring einen relativ hohen Aufwand benötigt und die Klasse nur wenig Probleme bereitet, also einen sehr niedrigen Impakt besitzt. Gottklasse 7 oder 8 hingegen sind hervorragende Kandidaten, da sie wenig Aufwand benötigen würden und hohe negative Auswirkungen auf das System haben. Somit kann durch die Cost Benefit Analyse, eine Priorisierung der Gottklassen vorgenommen werden.

Der Design Debt kann sich noch eine andere Art an technischen Schulden hinzufügen, die sich eher auf den Code der Klassen selbst bezieht.

### 5.3.2 Code Debt

#### Beschreibung

Bei dem zweitwichtigsten TD-Typen handelt es sich um Code Debt, weshalb dieser mit erhöhter Priorität behandelt werden sollte. Code Debt bezieht sich auf Probleme, die im Quellcode wiederzufinden sind und dessen Lesbarkeit negativ beeinflussen, wodurch die Wartbarkeit des Codes erschwert wird [9]. Code Debt erscheint während der Implementierungsphase der Softwareentwicklung. Oft kann diese Art von technischen Schulden identifiziert werden, indem im Code nach schlechten Coding-Praktiken gesucht wird [30] [69].

#### Ursachen und Folgen

Wie Rios et al. [55] es beschreiben, handelt es sich bei den wichtigsten Ursachen von Code Debt um eine unangebrachte Deadline sowie eine ungenaue Zeiteinschätzung, ein Mangel an Erfahrung, an Refactoring und an Training und eine unangemessene Planung. Die häufigsten, negativen Folgen von Code Debt sind eine geringe Wartbarkeit, finanzielle Verluste und häufige Überarbeitungen [55]. Code Debt führt zu schwer lesbarem Code, was es somit deutlich erschwert, ihn im Nachhinein zu verbessern. Da dies allerdings oft notwendig ist, müssen sowohl zeitliche als auch finanzielle Ressourcen eingesetzt werden.

#### Indikatoren

Wie auch bei Design Debt, gibt es für Code Debt zahlreiche Indikatoren, die teilweise denen des Design Debt sehr ähneln, da sie Code-bezogen sind.
Davon sind einige:

- **Code-Smells**: Wie bei Design Debt sind die sogenannten Code Smells ein wichtiger Indikator, wie zum Beispiel Methoden, die zu viel oder unnötigen Code implementieren [65].
- **Duplizierter Code**: Identischer Code, der sich an zwei unterschiedlichen Stellen in einem System befindet [34].
- **Suboptimaler Coding Styl**: Code, der auf unübersichtliche Weise geschrieben wurde, da keine Richtlinien befolgt wurden und er somit schwer wartbar ist [9].
- **Langsame Algorithmen**: Code, dessen Ausführung einen erhöhten Zeitraum benötigt und der auf eine ineffiziente Nutzung der Ressourcen bei dessen Erstellung hinweist [70].

#### Methode: Code Smell Intensitäts-Index

Um mit Code Debt in einem Startup umgehen zu können, wurde durch Fontana et al. [71] eine intuitive Methode zur Detektion und Priorisierung von Code Smells entwickelt. Es handelt sich um einen Intensitäts-Index, der anhand von mehreren Metriken berechnet wird und es ermöglichen soll, zu entscheiden, für welchen Code Ressourcen in dessen Refactoring investiert werden sollten.
Um dies zu bewerkstelligen, wurden durch Fontana et al. [71] sechs Code Smells Arten ausgesucht, die besonders oft vorkommen und schwerwiegende Folgen haben können. Dies sind:

- **Data Klassen**: Ein Data Klasse ist eine Klasse, die fast ausschließlich aus Attributen und aus keinen Methoden besteht [72].
- **Brain Methoden**: Eine Brain Methode ist eine Methode, die Funktionalitäten in sich integriert, die besser in mehrere Methoden hätte aufgeteilt werden sollen [73].
- **Shotgun Surgery**: Bei Shotgun Surgery geht es darum, dass mehrere Klassen modifiziert werden, um eine einzige Änderung durchzuführen [74].

- **Message Chains**: Message Chains bestehen darin, dass durch mehrere Klassen gegangen werden muss, um an Daten einer anderen Klasse zu gelangen [74].
- **Dispersed Couplings**: Letztendlich entsteht Dispersed Coupling, wenn eine Methode Methoden aus einer oder mehreren anderen Klassen aufruft [75].
- **Gottklassen**

Danach werden die Code Smells in Projekten identifiziert. Hierfür wurden durch Fontana et al. [71] mehrere Metriken eingesetzt. Eine Klasse ist zum Beispiel nur eine Data Klasse, wenn die Werte der Metriken für diese Klasse, durch Lanza et al. [76] definierte Grenzwerte über- oder unterschreiten.
Als nächstes werden die Werte der Intensitäts-Indexe der jeweiligen Code Smells berechnet, um sie mit den Code Smells ihrer Art vergleichen zu können. Durch die Intensitätsindexe können dann zum Beispiel tabellarische Übersichten aller Data Klassen erstellt werden und die mit den höchsten Werten als erstes refactored werden. Genauere Informationen zu sämtlichen Berechnungen können im Anhang A.2 gefunden werden.
Das Erkennen von Code Smells anhand der Prädikate und Metriken sowie das Vergleichen der Smells anhand der Werte der Intensitäts-Indexe, kann zudem mittels mehrerer Tools sowie mit dem, durch Fontana et al. [71] entwickelten Tool JCodeOdor, durchgeführt werden. Somit bietet der Intensitäts-Index eine angemessene Möglichkeit, um Code Smells zu managen.

### 5.3.3 Test Debt

#### Beschreibung

In vielen Software-Startups herrscht das Motto „gemacht ist besser als perfekt", was darauf hinweist, dass dort im Generellen ein Mangel an Testing und Qualitätssicherung vorhanden ist [7]. Es fällt Startups nämlich meistens schwer, die Wünsche des Kunden zu erfüllen und gleichzeitig ein hochwertiges Produkt zu liefern [7]. Aus diesem Grund entsteht die sogenannte Test Debt in der Testphase. Diese bezieht sich auf Probleme, die bei Testaktivitäten gefunden werden und die Qualität der Testaktivitäten beeinträchtigen können [9].

#### Ursachen und Folgen

Das entsprechende Diagramm von Rios et al. [55] im Anhang A.3 stellt dar, dass die wichtigsten Ursachen eine zu enge Deadline und somit eine ungenaue Zeiteinschätzung sind. Diesen fügt sich eine ungenaue Analyse der Auswirkungen und Risiken hinzu, eine zu hohe Projektkomplexität und ein schlechtes Design der Projektstruktur, welches in Abschnitt 5.3.4 genauer erklärt wird.
All diese Ursachen führen zu unterschiedlichen Mängeln, wie häufigere Überarbeitungen, finanzielle Verluste für das Startup und einer Verzögerung bei der Auslieferung des Produkts. Startups wissen also öfters nicht, ob es sich lohnt Testing durchzuführen, da sie über wenig Zeit verfügen und sie die Auswirkungen von ungeprüftem Code auf die Zukunft des Unternehmens stark unterschätzen.

#### Indikatoren

Es gibt eine Vielzahl an Indikatoren, die auf Test Debt zurückführen. Davon sind einige:

- **Mangel an automatisierten Tests**: Tests werden zum Großteil manuell durchgeführt, was zu einem erhöhtem Zeit- und Ressourcenaufwand führt [34].
- **Nicht ausgeführte geplante Tests**: Entweder Tests, die durch das Team geplant wurden oder bereits geschrieben wurden, aber nie ausgeführt worden sind [30].

- **Mangel an Test Case Documentation**: Es fehlt Dokumentation, die beschreibt welche Artefakte während oder nach der Ausführung von Tests entstanden sind [10].
- **Bekannte Fehler im Code**: Fehler, die im Code vorhanden sind, aber deren Auswirkungen nie getestet wurden [30].
- **Test Smells**: Bei Tests Smells handelt es sich um Tests, die schlecht geschrieben wurden und die Wartbarkeit vom Code somit negativ beeinträchtigen könnten [77].

### Erste Methode: Test Driven Development

Ein großer Fehler von vielen Startups ist, dass diese meistens sogenanntes Exploratory Testing durchführen. Hierbei handelt es sich um Testing, bei dem keine formellen Test-Cases definiert werden, sondern Tester ihrer Intuition und Wissen folgen, um zu entscheiden, wann Tests durchzuführen sind [78].

Um dieses Problem zu vermindern, kann als erstes Test Driven Development (TDD) eingesetzt werden. Das Ziel von TDD ist, wie es Beck [44] und Astels [79] beschreiben, einen sauberen Code zu erhalten, der zuverlässig funktioniert. Hierfür wird der sogenannte Rot-Grün-Refactor Zyklus durchgegangen, der in Abbildung 11 dargestellt wird.

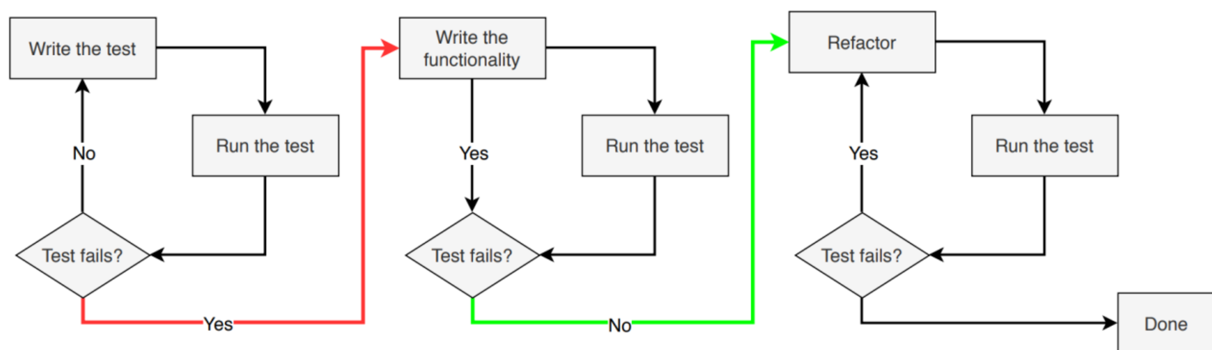


*Abbildung 11: Der Rot-Grün-Refactor Zyklus des Test Driven Developments [38]*

TDD funktioniert also so, dass zuerst ein Test implementiert wird, dann der entsprechende Code und letztendlich der Produktionscode verbessert wird, wobei jedes Mal der Erfolg des Tests geprüft wird [38] [79].

Durch Test Driven Development wird also, wie es Beck [44] beschreiben, der gesamte Code im Voraus auf eventuelle Probleme geprüft und der Zeitaufwand wird zusätzlich reduziert, da dieselbe Person für das Entwickeln und Schreiben der Tests zuständig ist.

### Zweite Methode: Automated Testing

Startups sind oft dazu verpflichtet Software in kurzen Zyklen zu veröffentlichen, um konkurrenzfähig zu bleiben. Deshalb ist es äußerst wichtig effiziente Tests zu ermöglichen, die nach ihrer Einführung nur einen minimalen Aufwand benötigen und Test Debt minimieren. Genau hierfür ist Automated Testing gut geeignet, welches, wie der Name es verrät, dazu dient, automatisch Tests für geschriebenen Code durzuführen [80]. Wie genau Automated Testing korrekt in einem kleinen Unternehmen eingeführt wird, um technische Schulden im Anschluss zu vermeiden, wird im Abschnitt 5.3.14 genauer beschrieben.

Jede Lösung zu Test Debt stellt, wie es Shah et al. [78] beschreiben, einen Punkt in einem Kontinuum dar. Auf der einen Seite bietet sich die Möglichkeit Exploratory Testing einzusetzen und somit geringe Vorlaufkosten einzugehen, aber auch erhöhte Schulden im Nachhinein. Auf der anderen Seite kann Automated Testing eingesetzt werden, was Anfangs ressourcenintensiver ist, aber im Anschluss Zeit

und Kosten erspart. Die Projektleitung des Startups muss also, je nach Projekt, entscheiden, welche Lösung für das Unterfangen angemessener ist.

### 5.3.4 Architecture Debt

#### Beschreibung

Sehr oft werden technische Schulden nicht nur durch Code verursacht, sondern auch durch strukturelle und architektonische Entscheidungen [62]. Wenn es um die Architektur eines Systems geht, kann es allerdings oft, während der Planungs-, Design- und Implementierungsphase, zu Architecture Debt kommen [30] [62]. Dieser Schuldentyp bezieht sich auf problematische Architekturentscheidungen, die sich auf die interne Qualität von Software auswirken [81].

#### Ursachen und Folgen

Zu den häufigsten Ursachen von Architecture Debt, welche durch Rios et al. [55] genannt werden, gehören eine unangemessen Deadline, ein Mangel an Fachwissen und an Wissen über die eingesetzte Technologie, sowie eine nicht ausreichende Infrastruktur des Unternehmens. Zeitdruck und ein Mangel an Qualifikationen sind also ein großer Verursacher von Architecture Debt. Die wahrscheinlichsten Folgen von Architecture Debt sind eine schlechte Performance, Auslieferungsverzögerungen und eine schlechte Wartbarkeit. Bei dem folgendem Lösungsansatz geht es vor allem darum, letztere zu verbessern.

#### Indikatoren

Eine Vielzahl an Indikatoren weist darauf hin, ob in einem System Architecture Debt vorhanden ist. Einige Beispiele sind:

- **Architectural Smells**: Hierbei handelt es sich um architektonische Entscheidungen, die die Qualität des Systems negativ beeinflussen. Wie zum Beispiel schlechte Designentscheidungen, die die Modularität gefährden [82].
- **Dependency Verstöße**: Wenn zum Beispiel durch eine Klasse auf Daten einer anderen Klasse zugegriffen wird, die diese laut Design Pattern nicht verwenden sollte [83].
- **Uneinheitlichkeit von Patterns und Policies**: Wenn in einem System mehrere unterschiedliche Architektur-Patterns, wie zum Beispiel Client-Server eingesetzt werden oder die eingesetzten Richtlinien nicht dem Pattern entsprechen [84].
- **Voneinander abhängige Ressourcen**: Dieses Problem erscheint, wenn zum Beispiel die Ressourcen eines Moduls von den Ressourcen eines anderen abhängen und somit zu großen Abhängigkeiten führen [85].
- **Mangel an Mechanismen für die Behandlung nicht-funktionaler Anforderungen**: Nicht-funktionale Anforderungen werden nicht oder nur wenig behandelt, sodass zum Beispiel die Wartbarkeit des Codes vernachlässigt wird [86].

#### Methode: Coupling-Between-Modules

Um Architecture Debt zu beheben, reicht es leider oft nicht aus, den Code nur stellenweise zu ändern, sodass umfangreichere Entwicklungsaktivitäten notwendig sind [30]. Es sei denn, die Architecture Debt werden kontinuierlich erkannt und entfernt, um deren Menge zu minimieren.

Um im Laufe der Projektdurchführung zu wissen, wie viel Architecture Debt vorhanden ist, haben Tvedt et al. [87] eine semi-automatisierte Methode entwickelt, die die tatsächliche Architektur mit der geplanten vergleicht. Aus diesem Grund haben Tvedt et al. [87] auch entschieden, als Metrik für die

Architekturqualität und die Wartbarkeit des Systems, die Anzahl an ungerichteten Verbindungen jedes Moduls mit anderen Komponenten zu verwenden. Diese Anzahl entspricht dem *Coupling-Between-Modules* (CBM).
Um diese Metrik für Module im Quellcode einfach auszurechnen, haben Tvedt et al. [87] ein Tool entwickelt, das den Code analysiert und dann eine Liste der Verbindungen für jede Komponente ausgibt. Vor dessen Einsatz sollte das Team eine geplante Architektur für das System definieren, zum Beispiel durch ein Whiteboard Gespräch unter zuständigen Teammitgliedern. Zu einem gewünschten Zeitpunkt kann dann die Architektur des Systems, mittels des Tools, mit der geplanten verglichen werden. Ein Beispiel eines Teils der, durch das Tool ausgegebenen, Liste, der nur die Anzahl an Verbindungen pro Komponenten nennt, ist in Tabelle 1 zu sehen.

*Tabelle 1: CBM-Werte für geplantes und tatsächliches Design, in Anlehnung an [87]*

| Components | Planned Design | Actual Design (VQI3) |
|---|---|---|
| Cache | 1 | 2 |
| Mediator | 10 | 8 |

Hier sind zwei Typen an Verstößen deutlich zu erkennen: Verbindungen im tatsächlichen Design, die nicht geplant waren und Verbindungen im geplanten Design, die nicht im tatsächlichen auftauchen. Nach dem Erkennen der Verstöße kann der Code angepasst und der Prozess wiederholt werden und genauere Angaben zu diesem Vorgehen im Anhang A.4 gelesen werden.
Somit bietet der Ansatz von Tvedt et al. [87] eine einfache, Tool-unterstützte Methode für Startups, um ihre Architecture Debt zu managen und die Wartbarkeit zu erhöhen.

### 5.3.5 Requirements Debt

#### Beschreibung

Einer der wichtigsten Typen an technischen Schulden ist die Requirements Debt, da sie den Grundstein für zahlreiche Entscheidungen in einem Projekt legt. Wie es der Name verrät, entsteht Requirements Debt meistens in der Planungsphase und bezieht sich auf Kompromisse, welche eigegangen werden, wenn es darum geht zu entscheiden, welche Anforderung und wie diese Anforderungen implementiert werden sollten [30] [62]. Es handelt sich also um den Abstand zwischen der optimalen Anforderungsspezifikation und der tatsächlichen Systemimplementierung [9].

#### Ursachen und Folgen

Wie bei den vorigen TD-Typen wird durch Rios et al. [55] beschrieben, dass eine der häufigsten Ursachen für Requirements Debt unter anderem eine zu enge Deadline ist. Dieser fügt sich ein Mangel an Fachwissen und ungenaue oder komplexe Anforderungen hinzu. Kleinen Unternehmen, wie Startups, fehlt es an Zeit und an, für das Team verständlichen, Anforderungen. Dies führt dann vor allem zu häufigen Planänderungen durch Verzögerungen und einer geringen Wartbarkeit des Produkts, sodass diese Schulden bei den Anforderungen unbedingt beglichen werden müssen.

#### Indikatoren

Auch für Requirements Debt gibt es mehrere Indikatoren, die auf dessen Präsenz in einem Projekt hinweisen. Davon sind einige:

- **Requirements Smells**: Bei Requirement Smells handelt es sich um unklare Anforderungen, die unter anderem durch den Einsatz von subjektiven und vagen Begriffen entstehen [88].
- **Veränderliche Anforderungen**: Die Anforderungen sind nicht von Anfang an klar definiert, sondern ändern sich im Laufe der Zeit, indem sie zum Beispiel erst später klarer werden [89].

- **Unnötige Merkmale**: Beim sogenannten Gold-Plating oder Over-engineering sind bereits Funktionalitäten entwickelt worden, die nicht verlangt waren, da die Anforderungen ungenau sind und falsch interpretiert wurden [89] [90].
- **Anforderungs-Backlog Liste**: Es besteht, während und nach der Entwicklung des Systems, eine lange Liste an unerfüllten Anforderungen [10].

#### Erste Methode: Vermeiden der Requirements Debt Typen

Wie es Rindell et al. [81] beschreiben, ist das Management von Requirements Debt am schwersten zu automatisieren, da es einer der TD-Typen ist, der am wenigsten technikbezogen ist. Lenarduzzi et al. [88] beschreiben daher drei Ansätze für kleine Unternehmen, um mit Requirements Debt umzugehen. Jeder Ansatz betrifft einen, durch die Forscher definierten, sogenannten Requirements Debt Typen.

- Beim ersten Typen handelt es sich um *unerfüllte Kundenwünsche* (Typ 0). Dieser entsteht, wenn gewisse Wünsche von Kunden oder Stakeholdern bezüglich des Systems, entweder absichtlich wegen einer Deadline oder unbeabsichtigt nicht beachtet wurden. Um zu messen, wie hoch der Typ 0 ist, kann einfach verglichen werden, wie hoch die Anzahl an implementierten Features, im Vergleich zur Anzahl an in Interviews oder Experimenten geäußerten Wünschen, ist [38]. Hierfür sind auch keine genauen Metriken nötig, aber es kann dem Team eine ungefähre Idee über das Ausmaß der Schulden verleihen.
- Beim Zweiten handelt es sich um die zuvor genannten *Requirement Smells* (Typ 1), wovon einige mit ihrer Definition und Erkennungsstrategie in Tabelle 2 wiederzufinden sind. Diese Smells können sowohl Anforderungen in natürlicher Sprache betreffen als auch in Darstellungen wie zum Beispiel UML. Sie stellen Verstöße gegen den ISO29148 Standard für die Qualität von Anforderungen dar [91]. Das, in Abschnitt 2.3.3 definierte, Schuldenkapital kann hier grob danach gemessen werden, wie viel Zeit und Ressourcen nötig sind, um die Requirement Smells zu beheben.

*Tabelle 2: Beispielhafte Requirement Smells und deren Erkennungsstrategien, in Anlehnung an [91]*

| Requirement Smells | Beschreibung | Erkennungsstrategie |
|---|---|---|
| **Subjektive Sprache** | Subjektive Sprache bezieht sich auf Wörter, deren Semantik nicht objektiv definiert ist, wie z. B. benutzerfreundlich, einfach zu verwenden, kosteneffizient. | Wörterbuch |
| **Zweideutige Adverbien und Adjektive** | Mehrdeutige Adverbien und Adjektive beziehen sich auf bestimmte Adverbien und Adjektive, die von Natur aus unspezifisch sind, wie z. B. fast immer, bedeutend und minimal. | Wörterbuch |
| **Schlupflöcher** | Schlupflöcher sind Formulierungen, die ausdrücken, dass die folgende Anforderung, nur in einem bestimmten, unpräzisen definierten Umfang, erfüllt werden muss. | Wörterbuch |

Wie Code Smells, können Requirement Smells durch Refactoring der Anforderungen beseitigt werden, indem die Ziele der Anforderungen beibehalten und die Requirement Smells entfernt werden.

- Beim letzten, durch Lenarduzzi et al. [88] beschriebenen Typen an Requirements Debt, handelt es sich um eine *nicht übereinstimmende Umsetzung* (Typ 2) mit den Wünschen der Stakeholder. Das Schuldenkapital entspricht in diesem Fall den Kosten, die nötig sind, um die tatsächliche Implementierung mit den eigentlichen Wünschen zu vergleichen.

#### Zweite Methode: Vision Videos

Eine andere, vielversprechende Methode, um Anforderungen zu erheben, zu klären und zu validieren, bieten die, durch Karras et al. [92] beschriebenen Vision Videos. Hierbei geht es darum, dass das Startup ein initiales Meeting mit den Stakeholdern durchführt, um zu wissen, was sich diese vom Produkt erwarten. Im Anschluss kann ein einfaches Video erstellt werden, dass das Problem beschreibt, welches gelöst werden muss, die entsprechende Lösung erklärt und welches Ergebnis sich dadurch für die Kunden ergibt. Wichtig ist hierbei, dass auf die Qualität des Videos und dessen Vision geachtet wird, sodass dieses technisch und inhaltlich angemessen ist und eine starke Wirkung auf dessen Zuschauer hinterlässt, indem es die Vision klar und vollständig darstellt [92].
Diese Videos sind kurz und bieten, durch die Vorführung und das Einholen von Feedback bei den Stakeholdern, einen guten Weg, um herauszufinden, ob das Team deren Wünsche verstanden hat. Durch diese wichtige Kommunikation können ebenfalls Mängel gefunden, korrigiert und deren Lösungen dokumentiert werden. Dadurch können schon im Voraus die drei, zuvor beschriebenen, Requirements Debt Typen teilweise vermieden werden.
Auch eine vollständige Dokumentation kann dabei helfen sicherzustellen, dass Anforderungen angemessen umgesetzt werden.

### 5.3.6 Documentation Debt

#### Beschreibung

Ein sehr häufiger und oft unterschätzter TD-Typ ist die sogenannte Documentation Debt, die in allen Entwicklungsphasen entsteht [93] [94]. Bei dieser handelt es sich um Projektdokumentation, die entweder unvollständig oder unangemessen ist. Dies führt dazu, dass sie zum Zeitpunkt der Entwicklung ausreichend sein mag, aber im Nachhinein Änderungen und Erweiterungen deutlich erschweren kann [30].

#### Ursachen und Folgen

Zu den häufigsten Ursachen von Documentation Debt gelten, wie es die InsighTD [55] Studie zeigt, eine zu frühe Deadline, gefolgt von der Tatsache, dass Unternehmen keinen Wert auf Dokumentation legen. Ein Mangel an guten Praktiken und Überlastung des Teams, führen ebenfalls häufig zu Documentation Debt. Unternehmen sind also die Auswirkungen von Documentation TD nicht bewusst und sie planen keine Zeit für das Verfassen davon ein. Zu den größten Auswirkungen von diesem TD-Typ zählen eine niedrige Wartbarkeit, geringe externe Qualität und Stress mit den Stakeholdern. Entwickler können also nur schwer Änderungswünschen nachkommen, was zu Spannungen und einer schlechteren Produktqualität für die Nutzer führen kann.

#### Indikatoren

Es gibt eine Bandbreite an Indikatoren, die auf Dokumentation Debt hinweisen. Einige davon sind:

- **Veraltete Dokumentation**: Wenn regelmäßig Änderungen am Code gemacht werden, aber Kommentare von der ersten Version des Projektes stammen und nie geändert wurden [34].
- **Keine oder unvollständige Dokumentation**: Die Dokumentation enthält Lücken, da zum Beispiel Methoden und Klassen nicht beschrieben sind oder keine Designdokumentation vorliegt [93].
- **Subjektive oder unklare Dokumentation**: Den Lesern ist nicht klar, was genau durch den Verfasser gemeint war, sodass es zu Fehlinterpretationen kommen kann, die schwerwiegende Folgen haben können [70].

#### Methode: Framework für gute Dokumentationsqualität

Wie es Treude et al. [95] beschreiben, existieren für jedes Unternehmen andere Ansichten bezüglich der Definition von guter Dokumentation. Dies liegt daran, dass Dokumentation sehr kontextabhängig ist, je nachdem um was für ein Projekt es sich handelt, auf welche Art es ausgeliefert wird, usw. Aus diesem Grund können Tools bis jetzt nur eingeschränkte Teile der Dokumentation bewerten und verbessern. Viele Entwickler scheuen sich auch davor, eine Dokumentation zu schreiben oder zu lesen [96]. Das weist darauf hin, dass sie sie als Zeitverlust sehen, da ihnen die Richtlinien fehlen, um zu wissen, wie Dokumentation kurz und klar geschrieben werden kann [95].

Aus diesem Grund haben Treude et al. [95] ein einfaches und somit für Startups ideales Qualitäts-Framework entwickelt, dass aus zehn Dimensionen besteht, die optimale Dokumentation beschreiben. Im Folgenden werden die fünf Ersten genannt, wobei die gesamte Liste im Anhang A.6 zu sehen ist:

- **Qualität**: Hier geht es darum, sich die Frage zu stellen, wie gut die Dokumentation, zum Beispiel die README oder der Kommentar, geschrieben ist, sei dies durch die Rechtschreibung oder Grammatik. Dies kann heutzutage einfach durch IDEs überprüft werden. Somit kann sichergestellt werden, dass das Lesen der Dokumentation nicht anstrengend wird oder Korrekturen benötigt [97].
- **Interesse**: Wird nun zum Beispiel ein erklärendes Dokument oder ein Tutorial veröffentlicht, sollte dieses möglichst interessant gestaltet werden, indem zum Beispiel Grafiken und Tabellen und kurze Sätze eingesetzt werden [97].
- **Lesbarkeit**: Es soll möglichst einfach sein, auf die relevanten Informationen zuzugreifen, indem diese logisch organisiert werden. Zudem sollen die Dokumentationselemente selbst übersichtlich und leserlich dargestellt werden und stehts verfügbar sein [98].
- **Verständlichkeit**: Die Person, die die Dokumentation verfasst, sollte sicherstellen, dass diese auch komplexe Sachverhalte möglichst einfach erklärt. Somit können zum Beispiel komplexe Methoden auch durch andere Entwickler erweitert werden [98].
- **Struktur**: Vor allem Dateien, welche nicht viel Design enthalten, wie zum Beispiel READMEs oder Kommentare, sollten so aufgebaut sein, dass der Leser sich schnell einen Überblick darüber verschaffen kann. Absätze, Titel und Listen können hierfür hilfreich sein [99].

Entwicklern können diese Dimensionen zum Beispiel in einer Schulung oder einem Dokument als Richtlinien vorgestellt werden. Dadurch wären sie besser dafür gewappnet, sich bei der Erstellung von Dokumentation die richtigen Fragen zu stellen und Informationen zu verschriftlichen, die den Ansprüchen des Startups entsprechen. So ist es möglich, das Erscheinen der genannten Indikatoren und Folgen im Voraus zu vermindern und die Skalierbarkeit des Produkts zu verbessern.
Eine korrekte Versionierung der Software kann die Skalierbarkeit des Produkts ebenfalls verbessern.

### 5.3.7 Versioning Debt

#### Beschreibung

Bei Versioning Debt handelt es sich um Schulden, die durch Probleme bei der Softwareversionierung entstehen [9]. Diese codebezogenen Schulden erscheinen daher eher von der Implementierung bis hin zur Wartung eines Softwareproduktes [100].

### Ursachen und Folgen

Für Versioning Debt wurden bei der InsighTD [55] Studie nur wenig Ursachen und Folgen genannt, da es sich um einen relativ neuen TD-Typen handelt. Zu den häufigsten Ursachen dieser Schulden gehören unter anderem, häufige Überarbeitungen der Software, Mangel an Fachwissen im Team und ein Mangel an angemessenen Prozessen und Planung. Wird das Produkt also oft geändert, ohne einem organsierten Ablauf zu folgen, kann es zu Konflikten zwischen Versionen kommen. Dies kann zu einer verspäteten Auslieferung führen sowie zu einer niedrigen, externen Qualität und vielen Überarbeitungen. Eine schlechte Versionsverwaltung kann bewirken, dass die Entwicklung immer komplexer und somit ineffizienter wird.

### Indikatoren

Mehrere Indikatoren weisen spezifisch auf die Entstehung oder Präsenz von Versioning Debt hin:

- **Unnötige Code Forks**: Diese entstehen, wenn Entwickler in Forks arbeiten, die überflüssig sind, da sie unterschiedliche Funktionalitäten bearbeiten als andere Entwickler. Somit wird der Code schnell verzweigt und unübersichtlich [34].
- **Merging Konflikte**: Das Abspalten von Code in einem Repository kann zwar das Entwickeln an separaten Funktionen erleichtern, aber beim Mergen kommt es dabei sehr oft zu Konflikten [80].
- **Mehrere unterstützte Produktversionen**: Wenn mehrere Versionen derselben Software durch Kunden verwendet werden und in einer neueren Version ein Bug entdeckt wird, müssen eventuell alle vorigen Versionen ebenfalls auf diesen geprüft werden [80].

### Methode: Versioning Tool

Eine Code Fork oder auch Abspaltung besteht darin, die Code Basis im Repository in mehrere Versionen zu spalten. Wie in den Indikatoren zu sehen, kann es dabei allerdings zu einer Vielzahl an Problemen kommen, die nur schwer durch einzelne Entwickler zu vermeiden sind. Aus diesem Grund existieren einige Tools, wie Git [101] oder Infox [102], die es ermöglichen, einen Überblick über die Forks zu behalten, um somit unnötigen Aufwand und Codekomplexität zu vermeiden. Durch Ren et al. [100] wurde ein Tool entwickelt und getestet, das Nutzer, wie in Abbildung 12 warnt, wenn deren Pull Requests ähnlich zu anderem Pull Requests sind. Das Tool vergleicht ebenfalls aktuelle Commits von Entwicklern mit anderen Forks um den Entwickler im Voraus, wie in Abbildung 13, zu warnen.

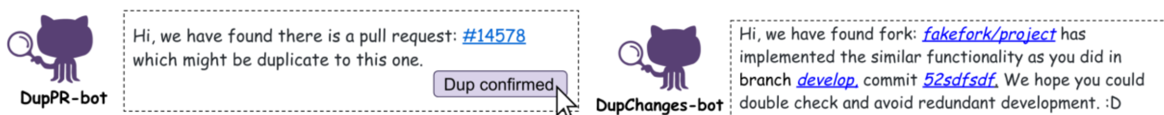


*Abbildung 12 (links): Warnung im Falle zweier ähnlicher Pull Requests [100]*
*Abbildung 13 (rechts): Warnung im Falle von ähnlichem Code zu einer anderen Fork, Pull Request, usw. [100]*

Um dies zu bewerkstelligen, haben Ren et al. [100] fünf Merkmale identifiziert, die es ermöglichen, die Inhalte einer Codeänderung zu charakterisieren und somit zwei Änderungen miteinander zu vergleichen. Für jedes Merkmal wurde eine Methode ausgewählt, um dieses messbar zu machen, welche alle im Detail im Anhang A.7 erklärt werden.

Das Tool gibt nach der Berechnung aller Merkmale einen Score aus, der beschreibt, wie wahrscheinlich es ist, dass zwei Änderungen dupliziert sind und die Entwickler können dann entscheiden, wie präzise sie bei der Erkennung sein wollen.

Für gutes Forking werden immer mehr unterstützende Tools entwickelt, sodass jedes Startup unbedingt eines in seinen Entwicklungsprozess integrieren sollte, um an Effizienz zu gewinnen.

### 5.3.8 Defect Debt

#### Beschreibung

In Softwareprojekten kommt es öfters vor, dass sich im Quellcode Defekte befinden, die dem Team entweder bewusst sind oder nicht [30] [103]. Die Defekte, die zu Defect Debt gehören, sind all jene, die dem Entwicklungsteam zum Beispiel durch Testing oder Bug Tracking bekannt sind, aber absichtlich nicht behoben wurden. Defekte erscheinen ab der Implementierungsphase und können bis in die Wartung vorgefunden werden [103].

#### Ursachen und Folgen

Zu den häufigsten Ursachen von Defect Debt gehören, wie es Rios et al. [55] erklären, ein Mangel an Testing, Fachwissen und angemessener Planung sowie an Rückverfolgbarkeit von Bugs. Defekte entstehen also dadurch, dass die Softwarequalitätsprüfung suboptimal und lückenhaft ist, da keine angemessenen Praktiken eingesetzt werden. Zu den häufigsten Folgen gehören eine verspätete Auslieferung und unzufriedene Kunden durch die schlechte externe Qualität, was letztendlich zu ungeduldigen Stakeholdern führt. Defekte können daher, wenn sie sich ansammeln, auf Dauer den Marktvorteil und das Image des Startups gefährden.

#### Indikatoren

Indikatoren, die auf Defect Debt hinweisen sind:

- **Funktionale Bugs**: Diese bedeuten, dass gewisse Funktionen sich nicht so verhalten, wie gewünscht. Ein einfaches Beispiel wäre ein Login Button, der den User nicht einloggt [104].
- **Logik Fehler**: Wenn ein Code falsch geschrieben wurde, also zum Beispiel ein Wert einer falschen Variable zugewiesen wurde, kann dies zu einem unerwünschten Verhalten der Software und Abstürzen führen [104].
- **System-Level Integrations-Bugs**: Hierbei handelt es sich um Probleme, die auftreten, wenn zum Beispiel verschiedene Module nicht korrekt zusammenarbeiten. Dies kann passieren, wenn Interfaces inkompatibel sind oder die Kommunikation zwischen den Modulen lückenhaft ist [105].

#### Methode: Beachten von Kosten- und Entscheidungsfaktoren

Ähnlich zu der Beschreibung aus Abschnitt 2.3.3, existieren beim Management von Defect Debt ein Schuldenkapital und Zinsen. Das Kapital entspricht der Geldsumme, die ausgegeben werden müsste, um einen Defekt sofort zu beheben, wenn er einem auffällt und die Zinsen sind die Kosten die zusätzlich gezahlt werden müssten, wenn dieser Defekt erst später korrigiert wird [103]. In einigen Fällen kann es sich allerdings lohnen, das Beheben von Defekten zu verschieben, da sie zum Beispiel nur geringe Risiken darstellen.

Um dementsprechend Unternehmen dabei zu unterstützen zu entscheiden, ob Defekte sofort oder erst später gelöst werden sollten, haben Snipes et al. [103] entsprechende Kosten- und noch wichtigere Entscheidungsfaktoren definiert. Die Kostenfaktoren für ein direktes Lösen des Defektes, also das Bezahlen des Schuldenkapitals und die Kostenfaktoren, die entstehen, wenn der Defekt nicht sofort behoben wird, also die Zinsen, werden im Anhang A.8 detailliert erklärt und deren Anteile gegeben.

In Anbetracht dieser zwei Kostenkategorien müssen Startups entscheiden, ob sie das Risiko eingehen wollen, einen Defekt nicht zu lösen und mit Glück das Schuldenkapital einzusparen und nur die Zinsen zu zahlen oder einen sicheren Weg gehen und das Schuldenkapital sofort zurückzuzahlen. Dies kann auf Dauer allerdings auch teuer werden. Bei dieser Entscheidung sollen die folgenden, durch Snipes et al. [103] definierten, Entscheidungsfaktoren helfen:

- **Schweregrad des Defekts**
- **Existenz eines Workarounds**
- **Dringlichkeit, des vom Kunden geforderten Fixes**
- **Aufwand, um den Fix zu implementieren**
- **Risiko des vorgeschlagenen Fixes**
- **Ausmaß an benötigtem Testing durch den Fix**

Diese Faktoren werden, sollte Interesse bestehen, ebenfalls ausführlich im Anhang A.8 definiert. Anhand der Kombination oder einer eigens erstellen Auswahl der Kosten- und vor allem Entscheidungsfaktoren müssen Startups entscheiden, welche Defekte sofort oder doch erst später behoben werden sollten. Tools und mathematische Methoden, um diese Aufgabe zu erleichtern, sind in Entwicklung [106] und zahlreiche Defekte können längst durch IDEs selbst behoben werden. Aber durch die hohe Anzahl an Faktoren und den komplexen Kontext, den das Feedback der Kunden darstellt, sind, vor allem bei größeren Defekten, kurze Teambesprechungen hilfreicher.

### 5.3.9 Build Debt

#### Beschreibung

Bei Build Debt handelt es sich um Probleme, die beim Bauen von Codes erscheinen und somit dazu führen, dass dieser Prozess zeit- und rechenintensiver wird [30] [107]. Die Entwicklung und Software können dadurch verlangsamt und frustrierender werden. Zu finden ist diese Art von Schulden somit von der Implementierung bis hin zur Wartung der Software [107].

#### Ursachen und Folgen

Wie durch die InsighTD Studie [47] gefunden, sind als Ursachen wieder eine zu enge Deadline und ein Mangel an Fachwissen im Team bezüglich optimierten Code Building vorhanden, sowie ein Mangel an Infrastruktur, wie zum Beispiel Software, um Build Debt zu testen und zu vermeiden. Build Debt kann zu, für den Usern sichtbaren, Bugs und Verlangsamungen führen und den zu implementierenden Code immer unübersichtlicher und ineffizienter gestalten.

#### Indikatoren

Nur einige von zahlreichen Indikatoren für Build Debt sind:

- **Under-declared Dependencies**: Bei diesen Abhängigkeiten ist ein Target transitiv abhängig von einem anderen. Das bedeutet, dass zum Beispiel die Library A von Library B abhängt, aber Library B von Library C abhängt. Somit ist A indirekt abhängig von C. Sollte Library C nun gelöscht werden, kann es bei Library A zu Build Problemen kommen [107].

- **Underutilized Dependencies**: Eine Klasse kann zum Beispiel von einer Library abhängen, die zu viel Funktionalitäten beinhaltet, welche nicht benötigt, aber trotzdem gebaut werden [107].

- **Over-declared Dependencies**: Diese bestehen, wenn in einem Target unnötige Abhängigkeiten zu anderen Targets genannt werden, die somit trotzdem gebaut werden müssen [107].

- **Fehlende Sichtbarkeitsregeln**: Targets sind nicht mit eindeutigen Sichtbarkeitsregeln wie public und private gekennzeichnet, sodass andere Targets auf deren Daten zugreifen können und ungewollte Abhängigkeiten entstehen [107].

- **Dead Flags**: Hierbei handelt es sich um Flags in der Kommandozeile, die ungenutzten Code aufrufen und somit eventuelle Refactorings deutlich erschweren [107].

#### Methode: Vermeiden der Build Debt Typen

Um das Erscheinen dieser Indikatoren zu vermeiden, haben Morgenthaler et al. [107] von Google, anhand des Beispiels der Softwareentwicklung in ihrem Unternehmen Build Debt in mehrere Untertypen geteilt. Zwei davon werden im Anschluss mit den, durch Morgenthaler et al. [107] definierten, Lösungen beschrieben. Mehr Details zu den Typen sowie ein weiterer Build Debt Typ, werden im Anhang A.9 beschrieben.

#### Dependency Debt

Dependency Debt entsteht durch die zuvor beschriebenen Under- und Over-Declared Dependencies. Um diese Art von Abhängigkeiten zu vermeiden, haben Morgenthaler et al. [107] einen sogenannten Fixit Day vorgeschlagen. Bei diesem sollen sich die Entwickler des Unternehmens zusammenschließen, um zu prüfen, in welchen ihrer Targets sich unnötige Dependencies befinden und wo direkte Dependencies fehlen. Dieser Prozess ist zwar nicht automatisiert, kann aber in einem kleinen Team, wie das eines Startups, deutlich einfacher durchgeführt werden als in einem multinationalen Unternehmen wie Google.
Für Underutilized Dependencies ist die Lösung nicht ganz so einfach, da die referenzierten Targets zum Teil eingesetzt werden und dementsprechend vorerst vollkommen gebaut werden müssen. Zudem kann ein Target zahlreiche transitive Dependencies haben. Aus diesem Grund hat Google einen Dependency Refactoring Assistenten namens Clipper entwickelt. Clipper priorisiert Dependencies, je nachdem wie einfach sie zu entfernen sind und verwendet dafür verschiedene Charakteristiken, welche im Anhang A.9 beschrieben werden.
Alle relevanten Informationen werden ebenfalls im Interface graphisch dargestellt, um die Entscheidung zu erleichtern, welche Dependencies nun gelöscht oder geändert werden können.

#### Visibility Debt

Fehlende Sichtbarkeitsregeln führen dazu, dass Entwickler Dependencies zu anderen Targets eingehen, auf die sie eigentlich nicht hätten zugreifen sollen. Dies kann dazu führen, dass die Entwickler der anderen Targets diese ändern oder löschen und somit unbewusst eine Kettenreaktion bei den unbekannten Clients auslösen. Aus diesem Grund wurde die Sichtbarkeit aller Targets durch Morgenthaler et al. [107] von public auf private gesetzt. Falls die Entwickler ausdrücklich wollen, dass auf ihre Targets zugegriffen werden kann, können sie diese wieder als teilweise sichtbar oder public kennzeichnen. Das Ziel ist es, Entwickler zu sensibilisieren, darauf zu achten, nicht ungewollt Dependencies einzugehen oder zu ermöglichen.
Optimierte Prozesse können ebenfalls dabei helfen, solche Sensibilisierungen zu vereinfachen.

### 5.3.10 Process Debt

#### Beschreibung

Bei Process Debt handelt es sich um Prozesse die ineffizient sind, da sie zum Beispiel für die Entwicklung eines Produkts nie angebracht waren oder nicht mehr ausreichend sind [30] [108]. Da dieser TD-Typ den Softwareentwicklungsprozess selbst betrifft, ist dieser somit phasenübergreifend und von der Planungs- bis hin zur Wartungsphase, wiederzufinden.

#### Ursachen und Folgen

Bei der InsighTD Studie [47] wurden zwei Hauptursachen für Process Debt genannt und zwar ein ineffektives Projektmanagement und falsche Zeiteinschätzungen. Das Management von Startups ist oft unerfahren und weiß somit nicht, welche Praktiken vorteilhaft wären und ist ebenfalls nicht in der Lage einzuschätzen, wie lange gewisse Aktivitäten benötigen werden. Daraus resultieren ein für Kunden, Stakeholder und das Startup unzufriedenstellendes Produkt mit schlechter Qualität sowie

häufige Überarbeitungen. Das Nicht-Einhalten gewisser Praktiken, wie zum Beispiel rechtzeitiges Testing, kann nämlich zu zahlreichen Mängeln und zusätzlicher Arbeit führen.

#### Indikatoren

Einen Großteil an Mängeln in Softwareprojekten kann man, zumindest teilweise, auf Process Debt zurückführen. Davon sind einige:

- **Reaktives Verhalten**: Statt, zum Beispiel, gewisse Maßnahmen oder Praktiken einzusetzen, um Bugs vorzubeugen, wird nur nach einer Lösung gesucht, nachdem ein Problem bereits entstanden ist [19].
- **Keine Verantwortlichkeiten**: Teammitgliedern werden keine klaren Rollen zugewiesen, sodass keine konkreten Zuständigkeiten vorliegen und Mitarbeitende zum Beispiel Aufgaben durchführen, für welche sie nicht qualifiziert sind [109].
- **Ineffizienz**: Langsame oder inexistente Prozesse führen zu mangelhafter Kommunikation, Organisation und somit einer begrenzten Leistung im Team [110].

#### Methode: Strengeres Definition of Done

Process Debt gehört zu den TD-Typen, die für Startups am schwierigsten zu bewältigen sind [23]. Startups lehnen die Idee von wiederholbaren und kontrollierten Prozessen meistens ab und bevorzugen es, unvorhersehbare, reaktive und ungenaue technische Verfahren einzusetzen [111]. Sie machen dies aus Angst davor, dass bürokratische Maßnahmen ihre Kreativität und Flexibilität negativ beeinflussen könnten [19].

Um in etablierten Unternehmen Prozesse zu verbessern und deren Qualität zu messen, kann das sogenannte Capability Maturity Model (CMM) eingesetzt werden [19]. Wie es im Anhang A.10 beschrieben wird, kann dieses Modell allerdings nicht für Startups eingesetzt werden, sodass ein anderer Weg gefunden werden muss, Prozesse zu verbessern.

In dem, durch Brunner et al. [38] erstellten, Softwareentwicklungsprozess, der im Abschnitt 2.5.2 beschrieben wird, wird Continuous Delivery eingesetzt, um die Software regelmäßig auszuliefern. Um dies zu bewerkstelligen, muss allerdings händisch geprüft werden, ob die Software auch gewisse Kriterien erfüllt. Wie dies systematisch und strukturiert möglich ist, wird durch Davis [13] erklärt.

Bei einem Mechanismus der agilen Softwareentwicklung, handelt es sich um das sogenannte Definition of Done (DoD) [35]. Dieses dient dazu zu prüfen, ob ein Softwareinkrement, wie zum Beispiel eine Story auch wirklich als fertig angesehen werden kann, indem geprüft wird, ob es gewisse Ziele und Qualitätsattribute erfüllt [112]. Davis [13] erklärt dementsprechend, wie er es in mehreren Unternehmen geschafft hat, ein DoD einzuführen, wenn keines vorhanden war oder ein strengeres DoD einzuführen und wie dies zu weniger technischen Schulden geführt hat.

Als erstes sollten die Teammitglieder geschult werden, um sicherzustellen, dass diese auch verstehen, worum es beim Konzept des DoD geht. Hierfür kann der Scrum Guide von Schwaber et al. [112] eingesetzt werden sowie das vereinfachte Beispiel aus der unteren Tabelle 3. Jede Reihe stellt eine, im DoD inkludierte, Aktivität dar und die Spalten stellen entweder dar, welche Inkremente betroffen sind oder was erreicht werden muss, damit die Aktivität für das Inkrement als erfüllt gilt.

Ein „X“ bedeutet, dass die Aktivität bzw. das Qualitätsattribut auf das Inkrement in dieser Spalte zutrifft. Die Reihenfolge, in der sich die Attribute hierbei befinden, spielt keine Rolle.

*Tabelle 3: Beispielhaftes Definition of Done eines Softwareprojektes, in Anlehnung an [13]*

| Qualitätsattribute | Story | Release | Bemerkungen |
|---|---|---|---|
| **Design** | | | Das Design ist fertiggestellt und entspricht den Kundenwünschen |
| **Code und Unit Testing** | X | | Es wurden Test Cases für wichtige Code-Elemente geschrieben und bestanden |
| **Performance Test** | X | X | Die Software läuft vergleichbar schnell zu Konkurrenz-Produkten |
| **Usability** | X | | Kunden sind größtenteils zufrieden mit der Usability der Software |

Als nächstes können die Teammitglieder aufgefordert werden, selbst DoD-Templates zu erstellen indem sie zum Beispiel Tools wie Rally [113] oder Agility [114] einsetzen. Es wird sich dann auf die besten Tools geeinigt und diese werden kontinuierlich ausgefüllt und verbessert.
Genauere Details zum strengeren DoD werden im Anhang A.10 gegeben und ein Beispielprozess, der beschreibt wie Startups vorgehen können, um einen Software-Prototypen zu entwickeln und die Definition of Done einzusetzen, wird in Kapitel 6 beschrieben.
Durch ein strengeres DoD kann auch sichergestellt werden, dass die Infrastruktur des Startups, gewissen Standards entspricht.

### 5.3.11 Infrastructure Debt

#### Beschreibung

Infrastructure Debt bezieht sich auf Probleme, die entweder in der Hardware- oder Softwareinfrastruktur eines Unternehmens auftreten und nicht rechtzeitig behandelt werden [115]. Infrastructure Debt ist phasenübergreifend und bezieht sich auf das Qualitätsmanagement des Unternehmens [116].

#### Ursachen und Folgen

Die wichtigste Ursache von Infrastructure Debt, ist, wies es sehr oft bei Startups der Fall ist, eine zu enge Deadline [55]. Es wird oft nicht die Zeit in Anspruch genommen, eine korrekte Infrastruktur für ein Projekt bereitzustellen. Das Problem ist hierbei, dass Infrastruktur Debt sehr schwere Folgen haben kann, denn falls die Infrastruktur zusammenbricht, kommt ein Projekt zum Stillstand [116]. Aus diesem Grund ist der größte Effekt von Infrastructure Debt eine schlechte Performance des Unternehmens, da dieses nicht mehr normal operieren kann [55].

#### Indikatoren

Indikatoren, die auf eine suboptimale Systeminfrastruktur hinweisen, sind vielseitig.

- **Verschobene Upgrades oder Infrastructure Fixes**: Sollten zum Beispiel neue Server oder ein Betriebssystem eingeführt werden, um Skalierbarkeit zu ermöglichen und dies wird nicht umgesetzt, kann das Wachstum des Startups gefährdet werden [30].
- **Veraltete Tools**: Egal ob es sich um Entwicklungstools oder TD-Management Tools handelt, führt das Einsetzen von älterer Software oft zu einer geringeren Effizienz und Produktivität, da es an Automatisierung fehlt [117].
- **Schlecht konfigurierte Infrastruktur**: Dies ist der Fall, wenn Maschinen oder zum Beispiel Software Services schlecht verbunden oder eingerichtet wurden, sodass das ganze System unübersichtlich und ineffizient ist [115].

- **Viele Ausfälle und Instabilität**: Wenn das System oder Tools oft ungeplant langsamer werden oder abstürzen, weist dies auf eine suboptimale Infrastruktur hin. Verschlimmert wird dieser Aspekt, wenn das ganze Monitoring für den Zustand der Infrastrukturkomponenten entweder gar nicht oder nur auf den einzelnen Komponenten selber stattfindet, statt einer zentralisierten Übersicht [116].
- **Sich wiederholende Aufgaben**: Wenn zum Beispiel täglich ein Backup von allen Geräten auf einen Server gemacht werden muss, verursacht dies einen vervielfachten Zeitverlust [116].

#### Methoden: DevOps und Infrastructure as Code

Startups beginnen meistens in kleinen Teams und haben somit anfangs oft nur mit kleineren Infrastrukturen zu schaffen. Sollte der präsentierte Prototyp allerdings erfolgreich sein, kann sich dies sehr schnell ändern. Es sollte für Startups dementsprechend möglich sein, eine skalierbare Infrastruktur einzurichten.

In ihrer Anfangsphase können Startups hierfür zuerst ein vereinfachtes Vorgehen befolgen, da nur wenige Geräte und Tools Teil vom System sind. Es sollte eine einfache Timeline erstellt werden und jedes Mal darauf aufgeschrieben werden, falls ein Vorfall im System auftritt [116]. Sollte zum Beispiel zu wenig Speicherplatz auf einer Festplatte oder einem Service vorhanden sein, ein Tool abstürzen, usw. Es sollten die Zeit sowie der Ort im System und eventuell die zuständige Person aufgeschrieben werden. Dies ermöglicht es, aus kleinen Anfangsfehlern zu lernen und somit Verbesserungskriterien für ein späteres System festzulegen. Ein anderes, oft unterschätztes Kriterium, um das zukünftige System zu verbessern, ist es, die im folgenden Abschnitt 5.3.12 präsentierte, People Debt zu reduzieren. Da, wie es Conways [118] Gesetzt beschreibt: „Unternehmen die Systeme designen… dazu gezwungen sind, Designs zu erstellen, die Kopien der Kommunikationsstrukturen dieser Unternehmen sind“. Wie in einem Unternehmen kommuniziert wird, reflektiert sich letztendlich in dessen Systeminfrastruktur.

Sollte der erste Infrastruktur-Prototyp sich bewiesen haben, kann das Startup als nächstes damit anfangen DevOps zu integrieren [115]. Hierbei geht es darum den Abstand, den es zwischen den Entwicklern (Dev) und dem operativen Teil (Ops) des Teams gibt, zu minimieren [119]. Dies wird erreicht, indem beide Abteilungen enger zusammenarbeiten und Infrastruktur mit ähnlichen Praktiken gehandhabt wird wie der Code, wodurch der Begriff *Infrastructure as Code* entstanden ist [120].

Um die, für das Startup notwendige, Infrastruktur zu erstellen, muss sich das Team zusammensetzen und aus den Erfahrungen der Timeline und Infrastrukturen von Konkurrenten für das Unternehmen spezifische Schlüsse ziehen. Wie dies im Detail im beispielhaften Falle eines Startups aussieht, dass Web-basierte Dienste einsetzt, und wie dessen Infrastruktur auf standardisierte Weise dargestellt werden kann, wird im Anhang A.11 erklärt.

### 5.3.12 People Debt

#### Beschreibung

Bei People Debt geht es darum, dass der menschliche Aspekt zu technischen Schulden führen kann, falls dieser vernachlässigt wird. Dies betrifft sowohl zwischenmenschliche Interaktionen als auch Qualifikationen der Teammitglieder [30] [121]. Menschen sind in allen Phasen des Entwicklungsprozesses involviert, sodass dieser TD-Typ das Projektmanagement betrifft.

#### Ursachen und Folgen

Zu den häufigsten Ursachen von People Debt gehört ein Mangel an Ressourcen, wie Zeit und Geld, sowie eine ungenügende Erfahrung der Teammitglieder [122]. Startups sehen den menschlichen

Aspekt daher als vorerst unwichtig an, was schnell zu einer mangelhaften Zusammenarbeit und somit einer verminderten Effizienz führt.

#### Indikatoren

In einem Projekt können zahlreiche Probleme auf People Debt hinweisen, doch bei den folgenden handelt es sich um eindeutige Indikatoren:

- **Schlechte Arbeitsstimmung**: In Teams können Spannungen entstehen, die zu einem Mangel an Vertrauen und Zusammenarbeit führen. Dadurch wird die Teamdynamik und somit auch die resultierende Software, negativ beeinflusst [122].
- **Mangelhafte Kommunikation**: Es kann in allen Unternehmen vorkommen, dass es einen Mangel an Kommunikation zwischen Mitarbeitenden gibt oder zwischen verschiedenen Managementebenen [12]. Dies führt dazu, dass wichtige Informationen nicht übermittelt werden, wodurch sich technische Schulden ansammeln.
- **Späte Einstellungen oder Training**: Wenn Mitarbeitende zu spät rekrutiert oder geschult werden, kann das zur Folge haben, dass zu wenig qualifiziertes Personal für das durchzuführende Projekt vorhanden ist. Dies bewirkt ebenfalls, dass das gesamte Wissen des Startups, sich in wenigen Personen konzentrieren muss und diese somit schnell überfordert sind [30].

#### Erste Methode: PANAS

People Debt gehört zu den TD-Typen, die am komplexesten zu studieren sind, da sie soziale, psychologische und organisationale Aspekte darstellt [12]. Es ist nur schwer möglich die technischen Schulden, welche durch gewisse soziotechnische Entscheidungen entstehen, zu identifizieren und zu messen [122]. Aber es gibt immer mehr Methoden, die es ermöglichen, soziotechnische Aspekte zu analysieren, was von hoher Wichtigkeit ist, da diese die Hauptursache von People Debt darstellen. Die dynamischen und miteinander verflochtenen Aktivitäten von Startups, erfordern nämlich eine enge Zusammenarbeit zwischen Teammitgliedern, aber auch Stakeholdern, Early Adopters und anderen involvierten Personen [7].

Ein wichtiger Aspekt, wenn es darum geht in einem Team zu arbeiten, stellt, wie erwähnt, die Stimmung dar [122], da Entwickler produktiver sind, wenn sie gut gelaunt sind [123]. Aus diesem Grund muss in einem Startup von Anfang an sichergestellt werden, dass diese angemessen ist. Eine Methode, wie dies sehr einfach und schnell und somit optimal für Startups durchgeführt werden kann, sind sogenannte Stimmungsbarometer. Ein Beispiel hierfür wäre das, aus der Psychologie stammende, PANAS (Positive and Negative Affect Schedule), bei dem Teammitglieder Fragebögen zur Selbsteinschätzung ausfüllen [124]. Den Mitarbeitenden werden in einer vereinfachten und somit kürzeren Version dieser Befragung, Namens PANAS-Short, jeweils sechs positive und sechs negative Affekte präsentiert [125]:

- Interessiert, wach, aktiv, glücklich, stark, freudig erregt
- Gereizt, wütend, nervös, besorgt, verwirrt, ängstlich

Im Anschluss müssen die Teammitglieder jeder dieser Affekte auf einer Skala von eins bis fünf bewerten, wobei fünf bedeutet, dass der Affekt sehr auf sie zutrifft [124]. Dies kann, je nach Wunsch, anonym durchgeführt werden, da dies ehrlichere Antworten produziert. Die Ergebnisse können dann eingesehen werden und Maßnahmen getroffen werden, wie zum Beispiel Privat- oder Teamgespräche, um die Ursachen von Problemen zu finden.

### Zweite Methode: SEnti-Analyzer

Ein für Startups noch einfacherer und somit optimaler Weg die Stimmung im Team zu analysieren, wird durch sogenannte Sentiment-Analyse Tools ermöglicht. Diese Tools haben als Ziel, Kommunikation zwischen Teammitgliedern zu analysieren und die Polarität ihrer Aussagen zu bewerten [126]. In Startups kann auf unterschiedliche Weisen kommuniziert werden, doch es finden vor allem in erfolgreichen Startups, immer Meetings statt [127]. Aus diesem Grund haben Herrmann und Klünder [128] ein Tool Namens SEnti-Analyzer entwickelt, das die Stimmung in Meetings analysieren kann, um falls nötig, entsprechende Maßnahmen zu treffen. Um dies zu bewerkstelligen, werden die Meetings entweder vor Ort oder durch die Mikrophone der Geräte der Teammitglieder aufgenommen und ein, im Anhang A.12 erklärtes Verfahren, eingesetzt [128].

Als Endergebnis bekommt zum Beispiel der Manager nach dem Meeting, einen Überblick über alle Aussagen und deren Polaritäten sowie über das Verhältnis der Polaritäten zueinander. Dank dieser Informationen ist es dann möglich, die Kommunikation und Stimmung im Meeting zu analysieren und eventuelle Probleme im Voraus zu erkennen, um zum Beispiel eine Software mit besserer Usability entwickeln zu können.

## 5.3.13 Usability Debt

### Beschreibung

Bei Usability Debt handelt es sich um unangemessene Entscheidungen, die die Benutzerfreundlichkeit betreffen und die somit dazu führen, dass im Nachhinein Anpassungen am System und Interface nötig sind [9]. Diese Probleme entstehen meistens während der Design- und Implementierungsphase der Softwareentwicklung, wenn suboptimale Mockups oder Wireframes entworfen und entwickelt werden.

### Ursachen und Folgen

Wie es Rios et al. [47] beschreiben, sind die häufigsten Ursachen, die bei InsighTD genannt wurden, ein Mangel an guten Praktiken und an Wissen bezüglich der einzusetzenden Technologien. Unternehmen wissen also oft nicht, was eine gute User Experience (UX) eines Systems überhaupt ausmacht und selbst wenn, wissen sie nicht, wie sie diese technisch umsetzen können. Dies führt dementsprechend zu einer suboptimalen Benutzerfreundlichkeit und einer schlechteren Leistung beim Einsatz der Software, da das Interface oft unübersichtlich gestaltet wurde.

### Indikatoren

Indikatoren für schlechtes UX-design sind vor allem für neue Nutzer sehr einfach zu erkennen, doch fallen Entwicklern nicht immer sofort auf. Davon sind, wie es Bandoly [129] beschreibt, einige:

- **Komplizierte Interfaces**: Oft wird versucht so viel Information wie möglich in einer einzigen Ansicht darzustellen, sodass, vor allem auf Startseiten, wirklich nützliche Informationen untergehen.
- **Erlauben von falschen Verhalten**: Interfaces, die es Usern ermöglichen zum Beispiel falsche Dateitypen hochzuladen oder Formulare auf inkorrekte Weise auszufüllen, verfehlen das Ziel guter UI, selbsterklärend und intuitiv zu sein.
- **Mangel an Feedback**: Wenn User Aktionen durchführen, wie einen Knopf zu drücken, um sich auszuloggen oder ein Dokument zu downloaden, erhalten diese kein Feedback. Somit wissen sie nicht, ob ihre Aktion nun überhaupt wahrgenommen wurde, was sehr schnell zu Frustration führen kann.

#### Methode: Minimum Viable User Experience Framework (MVUX Framework)

Wenn sich viele Early Adopters für ein prototypisches Produkt interessieren, da sie dieses gerne verwenden, kann es dies ermöglichen, konstruktives Feedback zu erhalten [130]. Eine bereits ab dem Anfang gute UX anzubieten, kann zudem dazu führen, dass sich dies bei den Early Adopters schnell herumspricht, was zu einem guten Ruf des Produkts führt [131]. Da Startups, wie im Abschnitt 2.5 beschrieben, gerne sogenannte Minimum Viable Products (MVPs) entwickeln, um ihre Idee zu testen, kann es also von Vorteil sein bereits für dieses, eine angemessene UX anbieten zu können.

Aus diesem Grund haben Hokkanen et al. [132], zusammen mit zwölf Startups, ein sogenanntes Minimum Viable User Experience Framework entwickelt, das beschreibt, welche UX-Faktoren beim Entwickeln eines MVPs unbedingt beachtet werden sollten. Das Framework wird in der unteren Abbildung 14 dargestellt und besteht aus drei Ebenen.

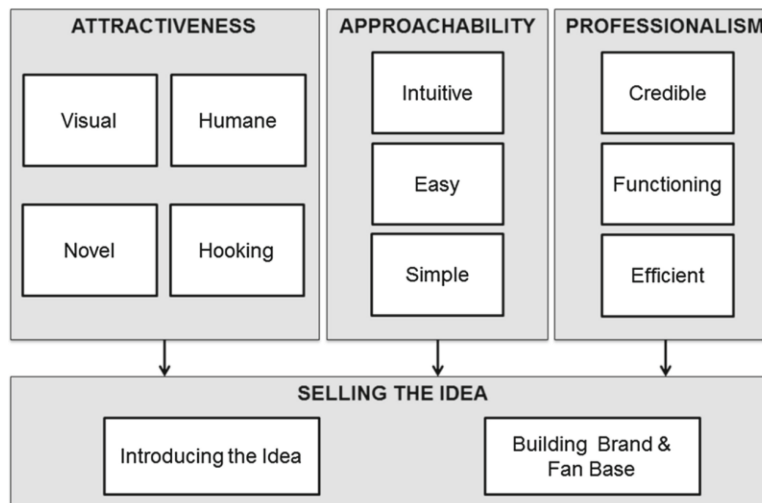


*Abbildung 14: MVUX-Framework zur Unterstützung der MVP-Entwicklung in Startups [132]*

Die höchste Ebene stellt Designziele wie Attraktivität dar, welche sich aus verschiedenen Faktoren zusammensetzen, und diese Faktoren werden durch das Umsetzen gewisser Praktiken erreicht.

Bei dem Designziel Attraktivität geht es zum Beispiel darum, dass der User einen positiven ersten Eindruck des Interfaces bekommen soll. Genauere Details zur Attraktivität und den anderen Designzielen sowie deren Faktoren, sind im Anhang A.13 wiederzufinden.

Die Designziele führen zum Hauptziel der Minimum Viable User Experience, nämlich die Idee des Startups auf überzeugende Weise vorzustellen. Die User sollen wissen wozu das Produkt dient und dessen Mehrwert erkennen, sodass sich die Produktidee herumspricht.

Die gesamte UX muss allerdings auch systematisch getestet werden, um sicherzustellen, dass das Interface möglichst fehlerfrei umgesetzt wurde.

### 5.3.14 Test Automation Debt

#### Beschreibung

Bei Test Automation Debt handelt es sich um technische Schulden, die entstehen, wenn automatisierte Tests erst eingeführt werden, nachdem Funktionalitäten bereits entwickelt wurden [30] [45]. Dieser TD-Typ erscheint somit erst ab der Testphase des Entwicklungszyklus und betrifft das Qualitätsmanagement des Startups.

### Ursachen und Folgen

Wie bei den meisten TD-Typen gehört zu den häufigsten Ursachen von Test Automation Debt, wie es die InsighTD Studie [47] zeigt, eine zu enge Deadline, sodass sich darauf fokussiert wird, mehr statt besser zu produzieren. Dem fügt sich hinzu, dass Unternehmen oft unterschätzen, wie wichtig Testing ist und nicht wissen, wie sie es automatisiert durchführen können. Dies hat zur Folge, dass das Testen der Software erschwert wird, da unter anderem, bei jedem größerem Inkrement, alle Tests wieder größtenteils händisch durchgeführt werden müssen.

### Indikatoren

Es weisen mehrere Aspekte auf die Existenz von Test Automation Debt hin:

- **Unbekannte oder nicht ausreichende Abdeckung durch automatisierte Tests**: Wenn das Startup keinen Überblick darüber hat, welcher Anteil seines Codes durch automatisierte Tests geprüft wird oder wenn es weiß, dass nur ein Bruchteil seiner Software ohne manuelles Eingreifen getestet wird [133].
- **Automatisiertes Testen ohne definierte Strategie**: Ebenfalls wird Test Automation Debt durch schlecht geplantes, automatisiertes Testen verursacht. Dies führt dazu, dass Entwickler den Überblick über den Test-Code verlieren und dadurch mehr Kosten verursacht werden als die, die potenziell gespart worden wären [134].
- **Unterbesetztes oder -qualifiziertes Tester-Team**: Es wird oft angenommen, dass automatisiertes Testen weniger Personal oder Qualifikationen benötigt, obwohl dies keineswegs der Fall ist [135].

### Methode: Sechs Aspekte für gutes Automated Testing

Wie es Wiklund et al. [45] beschreiben, wird beim Einführen von automatisierten Tests in Unternehmen meistens nicht dieselbe Rigorosität eingesetzt, wie bei der Entwicklung der Software. Dies führt zu einer erhöhten Menge an technischen Schulden.
Persson et al. [133] wollten durch eine Studie beweisen, dass das Befolgen eines präzisen Frameworks garantieren könnte, automatisiertes Testen optimal in einem Unternehmen zu implementieren. Nachdem sie allerdings automatisiertes Testing in zwei verschiedenen Projekten einführten, stellte sich heraus, dass ein informelles, projektspezifisches Vorgehen deutlich bessere Ergebnisse erbrachte. Dies liegt daran, dass automatisiertes Testing und entsprechende Tools je nach Projekt sehr unterschiedlich eingeführt und gewählt werden müssen. Ein konkretes Framework ist dementsprechend nicht nützlich, doch trotzdem hilft es, einige, durch Persson et al. [133] definierte, Aspekte zu beachten:

- Für jedes Projekt bei dem automatisiertes Testen eingesetzt wird, sollte durch das Team ein Wörterbuch oder „Wiki“ erstellt und im Laufe der Zeit vervollständigt werden. Dies liegt daran, dass Vokabular zum automatisierten Testen Begriffe aus verschiedenen Disziplinen vereint und eine Übersicht, mit der entsprechenden Terminologie, somit Unklarheiten bei Teammitgliedern beseitigen würde.
- Ebenfalls sollte ein Defect Tracking Tool, wie zum Beispiel Bugzilla [136] eingesetzt werden. Dies ermöglicht es, Defekte im Laufe der Zeit und nach Änderungen zu verfolgen und somit zu beobachten, wie sich diese in Anzahl und Form ändern. Dadurch können die Test Cases auf strukturierte Weise angepasst werden.

Die vier weiteren Schlüsselaspekte, die es ermöglichen automatisiertes Testen zu optimieren, werden im Anhang A.14 beschrieben.

Es müssen also einige Aspekte beachtet werden, um Tests in einem Startup zu automatisieren. Doch es muss bedacht werden, dass Automatisierung manuelles Testen nicht ersetzt, sondern beide komplementär sind [133].
Zusätzlich zum Testen, kann sich auch die Auswahl der Services auf die technischen Schulden des Startups auswirken.

### 5.3.15 Service Debt

#### Beschreibung

Unternehmen mieten oft sogenannte Web Services, um auf Ressourcen zuzugreifen, die sie selbst nicht haben aber benötigen. Hierbei kann es sich zum Beispiel um mehr Rechenleistung [137] oder Datenspeicher [138] handeln. Das Verwenden eines Web Services kann technische Schulden mit sich ziehen, wenn dieser zum Beispiel nicht den Anforderungen entspricht oder unterbenutzt ist [30]. Diese Art von Schulden kann in allen Phasen der Softwareentwicklung auftauchen, da sowohl unangemessene Anforderungen zur Auswahl von unangebrachten Web Services führen können, als auch erst festgestellt werden kann, dass die Web Services mangelhaft sind, sobald die Software ausgeliefert und gewartet wurde [137].

#### Ursachen und Folgen

Auch für Service Debt wurden, wie durch Rios et al. [55] beschrieben, bei Insight TD nur wenig Ursachen und Folgen genannt. Als Gründe für diese Art von Schulden wurden eine unangemessene Deadline und Entscheidungen des Managements genannt und zu hohe Kosten, um den passenden Service verwenden zu können. Ebenfalls war ein Mangel an Wissen bezüglich Services für Service Debt verantwortlich sowie Service Versionen, die mit der Eigenen inkompatibel waren. Diese Ursachen haben hauptsächlich dazu geführt, dass erneut Services ausgewählt werden mussten und dass sowohl finanzielle Verluste als auch Verzögerungen bei der Auslieferung der Software entstanden sind.

#### Indikatoren

- **Auswahl oder Ersatz eines Web Services** [70]: Die Tatsache, dass zu einem anderen Web Service gewechselt werden muss bedeutet, dass der Aktuelle nicht mehr angemessen ist, da er zum Beispiel zu langsam oder zu teuer ist [137].
- **Mangel an Ressourcen, Skalierbarkeit, Rechenleistung und anderen Quality of Service Attributen** [137]**.**

#### Methode: Real Options

Ein Ansatz, um die Menge an Service Debt in einem System bereits im Voraus zu minimieren, wird durch Alzaghoul et al. [137] vorgeschlagen. Dabei werden sogenannte Real Options besprochen, die eingesetzt werden sollen, um zwischen mehreren Web Services, den Besten auszuwählen. Real Options sind Optionen, also Handlungsmöglichkeiten, eines nicht finanziellen Kapitals, wie zum Beispiel eines Software Projektes [137]. Bei Web Services handelt es sich zum Beispiel um Dienste in der Cloud, die das Unternehmen mieten kann, um beispielsweise mehr Rechenleistung zu erhalten. Das Mieten dieser Web Services kann als Kredit angesehen werden, dem sich Zinsen hinzufügen, je nachdem wie effizient oder nicht dieser eingesetzt wurde. Diese Zinsen entsprechen den technischen Schulden, die es zu beseitigen gilt.
Um zu zeigen, wie die Methode funktioniert, wird durch Alzaghoul et al. [137] das Beispiel einer Softwarefirma verwendet, die sich mehr Skalierbarkeit wünscht, da deren Produkt gerade ein großes Wachstum erfährt. Es werden zwei sogenannte Call Options des Unternehmens verglichen, um entweder Methoden seines Systems durch einen Web Service 1 (WS1) oder einem Web Service 2

(WS2) ausführen zu lassen. Die Berechnungen ergaben, dass WS1 auf Dauer, im Gegenteil zu WS2, zu Profit führen wird, da WS1 eine zuverlässigere Verfügbarkeit hat und somit weniger Kunden verloren gehen. Ebenfalls werden Eigenschaften wie die monatlichen Nutzungskosten der Dienste beachtet. Genauere Details zu den Berechnungen der Werte der Optionen und den Entwicklungen dieser Werte im Laufe der Zeit, sind im Anhang A.15 zu finden.
Der Ansatz von Alzaghoul et al. [137] verbildlicht, wie wichtig es ist, auf die Eigenschaften von Web Services zu achten. Ein Startup könnte dank dieses Beispiels, auch wenn es die genaue Methode nicht vollkommen umsetzt, aus den Eigenschaften von Web Services herauslesen, welche in Zukunft einschränkend werden könnten und dementsprechend die beste Call Option auswählen.
Doch nicht nur auf die Services sollte Acht gegeben werden, sondern letztendlich auch auf die Sicherheit des Systems.

### 5.3.16 Security Debt

#### Beschreibung

Bei dem letzten TD-Typen, der Security Debt, handelt es sich um die Schulden, die entstehen, wenn an der Sicherheit der entwickelten Software gespart wird und somit Sicherheitslücken entstehen und ausgenutzt werden können [139]. Sicherheit ist kein Feature, das Software einfach hinzugefügt werden kann, sondern eher eine Eigenschaft, die daraus resultiert, wie die Software konzipiert wurde [139]. Aus diesem Grund kann die Entstehung von Security Debt nicht einer gewissen Phase der Softwareentwicklung zugewiesen werden und das Management dieses TD-Typen betrifft eher das Qualitäts- und Risikomanagement.

#### Ursachen und Folgen

Zu den häufigsten Ursachen für Security Debt gehören das Unterschätzen von dessen Gefahren, eine zu enge Deadline und ein Mangel an Fachwissen [81]. Da Sicherheitsprobleme, im Gegenteil zu Bugs, beispielsweise nicht unbedingt sofort sichtbar sind, werden diese mit einer niedrigeren Priorität bearbeitet. Die schlimmste Folge einer erhöhten Security Debt kann ein Vertrauensverlust bei den Kunden sein [139], was vor allem bei kleinen Unternehmen wie Startups, verheerende Konsequenzen haben kann.

#### Indikatoren

Durch die Common Weakness Enumeration [140], wurde sich auf Aspekte geeinigt, die beschreiben, welche Schwachstellen in einem System vorhanden sein können. Davon sind einige:

- **Unsachgemäße Eingabevalidierung**: In das System können durch Nutzer Daten eingegeben werden, doch es wird nicht geprüft, ob diese Daten die nötigen Eigenschaften erfüllen, um sicher verarbeitet zu werden. Hierdurch können Angreifer zum Beispiel das System lahmlegen oder bösartigen Code ausführen.
- **Fehlende Authentifizierung für kritische Funktionen**: Personen können auf Daten zugreifen, ohne ihre Identität bestätigen zu müssen. Vor allem in Zeiten, wo immer mehr Daten auf der Cloud gespeichert werden, kann dies Datendiebstahl deutlich vereinfachen.
- **Server-Side Request Forgery**: Ein Angreifer kann einem Server eine bestimmte, spezifisch erstellte, URL schicken [141]. Der Server wird durch diese URL getäuscht und fügt ihr eventuell sensitive Daten hinzu, auf die der Angreifer dann zugreifen kann.

#### Methode: Beachten der zehn Mängel der IEEE Computer Society

Es gibt Unmengen an Problemen in Systemen, die zu potenziellen Sicherheitsrisiken führen können. Aus diesem Grund haben sich Forscher des IEEE Computer Society Center for Secure Design [139] auf die wichtigsten Mängel geeinigt und kurze Beispiele genannt, wie diese umgangen werden können:

- Beim ersten Punkt wird erläutert, dass es möglichst vermieden werden sollte, sensible Daten des Systems und der User auf externen Geräten zu speichern. So sollte geprüft werden, ob es wirklich nötig ist, Daten zum Beispiel in Web Browsern oder Smartphones der Kunden zu speichern und falls dies unvermeidlich ist, sollte sichergestellt werden, dass die Daten auch entsprechend geschützt sind [142].
- Beim nächsten Aspekt geht es darum, Authentifizierungsmechanismen zu verwenden, die nicht umgangen oder manipuliert werden können. Um dies zu ermöglichen, sollten die Authentifizierungsmethoden mehrere Faktoren einsetzen, wie zum Beispiel Informationen, die der User kennt, biometrische Eigenschaften des Users oder etwas, was der User besitzt. Die sicherste Methode bietet immer noch ein gutes Passwort, doch dieses sicher zu speichern, ist nicht trivial. Aus diesem Grund empfehlen Arce et al. [139] hierfür einen Kryptographie-Experten zu beauftragen und ein bewährtes Passwort Management System einzusetzen, auch wenn dies für Startups Anfangs zusätzliche Kosten verursachen würde.

Mehr Details zu diesen Aspekten, sowie drei weitere Sicherheitsaspekte, werden im Anhang A.16 beschrieben.

Obwohl Security Debt durch die Forschenden als unwichtiger bewertet wurde, bleibt auch dieser TD-Typ von hoher Wichtigkeit, da, die Sicherheit betreffende, Zwischenfälle, je nach Schweregrad, das frühzeitige Ende eines Startups verursachen können. Deshalb sollten die hier genannten Aspekte beachtet werden, welche den Qualitätscharakteristiken des ISO/IEC 25010 [143] Standards ähneln, um die grundlegende Sicherheit des Systems zu gewährleisten. Doch es gibt auch zahlreiche TD-Typen, die stark mit Security Debt zusammenhängen, sodass auf diese ebenfalls geachtet werden muss.

## 5.4 Zusammenhänge zwischen TD-Typen

TD-Typen können anhand ihrer Merkmale erkannt und somit mit den entsprechenden Methoden gemanagt werden. Doch wenn ein TD-Typ identifiziert wird, bedeutet dies oft, dass noch andere im System vorhanden sind, da, wie es Li et al. [34] beschreiben: „ein TD-Item keine Insel ist, es kann andere Items beeinflussen oder von diesen beeinflusst werden."

Aus diesem Grund könnte es Startups helfen, die Zusammenhänge zwischen den TD-Typen zu kennen, um gezieltere Maßnahmen treffen zu können. Die Übersicht aus Tabelle 4 soll dies ermöglichen. Es wird für alle 16 TD-Typen gezeigt, ob es direkte Zusammenhänge zwischen diesen gibt. Ist dies der Fall, wird die Überschneidung mit einem blauen Feld markiert. Um dies zu bewerkstelligen, wurden genannte Verbindungen aus sechs Forschungsarbeiten kombiniert und mit Erkenntnissen aus der Checkliste vervollständigt, die auf direkte Verbindungen zurückführen.

Zum Beispiel existiert ein direkter Zusammenhang zwischen Test Debt und Test Automation Debt. Sollten nicht alle Unit Tests wegen eingeschränkten Ressourcen automatisiert werden können, fehlt es entweder an Testabdeckung oder Tests müssen aufwändig manuell durchgeführt werden [144].

Ein anderes Beispiel, ist das der Infrastructure Debt und People Debt. Da, wie im Abschnitt 5.3.11 erklärt, Conways [118] Gesetz beschreibt, dass Unternehmen dazu gezwungen sind, Systeme zu bauen, die dieselbe Struktur wie ihre Kommunikation haben.

Wie im vorigen Abschnitt erwähnt, ist ein TD-Typ mit sehr vielen Zusammenhängen zu anderen, die Security Debt. Durch Versioning Debt kann es beispielsweise vorkommen, dass ein gewisser Anteil der User immer noch eine veraltete Version des Produkts einsetzt, die mit niedrigeren Sicherheitsstandards entwickelt wurde als die Neueste [139]. Es besteht ebenfalls ein direkter Zusammenhang mit Usability Debt. Sollte nämlich das Einrichten von zusätzlichen Sicherheitsmaßnahmen zu kompliziert sein, wollen die User meistens keinen zusätzlichen Aufwand betreiben [139]. Somit bevorzugen sie die einfachsten, aber zugleich auch unsichersten Einstellungen. Es folgen noch zahlreiche andere TD-Typen, wie Service Debt, wenn Web Services für Authentifizierungen eingesetzt werden oder Documentation Debt, wenn es darum geht, sicherheitsbezogene Informationen zu dokumentieren [139].

*Tabelle 4: Zusammenhänge zwischen TD-Typen (dunkelblaue Felder kennzeichnen direkte Zusammenhänge). In Anlehnung an [30] [139] [12] [116] [144] [145]*

| | ArcTD | BldTD | CdTD | DefTD | DesTD | DocTD | InfTD | PplTD | PrcsTD | ReqTD | SecTD | SerTD | TATD | TesTD | UsaTD | VersTD |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Architecture Debt** (ArcTD) | ■ | | | | ■ | | | | | | ■ | | | | | |
| **Build Debt** (BldTD) | | ■ | ■ | | | | | | | | | | | | | |
| **Code Debt** (CdTD) | | ■ | ■ | | ■ | ■ | | | | | ■ | | | | | |
| **Defect Debt** (DefTD) | | | | ■ | | | | | | | ■ | | | | | |
| **Design Debt** (DesTD) | ■ | | ■ | | ■ | | | | | | ■ | | | | | |
| **Documentation Debt** (DocTD) | | | ■ | | | ■ | | | | | ■ | | | | | |
| **Infrastructure Debt** (InfTD) | | | | | | | ■ | ■ | | | ■ | | | | | |
| **People Debt** (PplTD) | | | | | | | ■ | ■ | | | | | | | | |
| **Process Debt** (PrcsTD) | | | | | | | | | ■ | | | | | | | |
| **Requirements** Debt (ReqTD) | | | | | | | | | | ■ | ■ | | | | | |
| **Security Debt** (SecTD) | ■ | | ■ | ■ | ■ | ■ | ■ | | | ■ | ■ | ■ | ■ | ■ | ■ | ■ |
| **Service Debt** (SerTD) | | | | | | | | | | | ■ | ■ | | | | |
| **Test Automation Debt** (TATD) | | | | | | | | | | | ■ | | ■ | ■ | | |
| **Test Debt** (TesTD) | | | | | | | | | | | ■ | | ■ | ■ | | |
| **Usability Debt** (UsaTD) | | | | | | | | | | | ■ | | | | ■ | |
| **Versioning Debt** (VersTD) | | | | | | | | | | | ■ | | | | | ■ |

Zellen, die nicht dunkelblau markiert sind, zeigen, dass es keinen direkten Zusammenhang zwischen den Typen gibt, aber es können durchaus indirekte Verbindungen existieren.
Zum Beispiel sind zwei Mitarbeitende, die nicht gut kommunizieren zwar ein People Debt Item, aber kein direktes Security Debt Item. Allerdings kann es durchaus sein, dass diese zwei Mitarbeitenden dadurch nicht genügend über Sicherheitsfeatures kommunizieren, wodurch wiederum Security Debt Items entstehen können.
Eine Bandbreite an TD-Items unterschiedlicher TD-Typen kann auch dadurch entstehen, dass gewisse Anforderungen unklar formuliert wurden, sodass eventuell nicht genug Wert auf die Usability der Software gelegt wurde.
Andere Zusammenhänge können auch durch den Einsatz von Tools entstehen, wie jene, die für die visuelle Modellierung von Code-Architektur eingesetzt werden [146]. Hierdurch können zum Beispiel Code Smells, wie leere Klassen oder „to-do“ Tags, entstehen.

## 5.5 Automatisierung

Vor allem für Startups, die oft nur über eingeschränkte finanzielle und zeitliche Ressourcen verfügen, können Tools zur Unterstützung des TD-Managements sehr hilfreich sein. Solche Werkzeuge können nämlich durch Mitarbeitende berechnete Ergebnisse überprüfen oder selbst die Software auf technische Schulden analysieren [147]. Erst in den letzten Jahren haben konkrete TD-Management Tools angefangen sich zu verbreiten [31], sodass sich viele noch etablieren müssen.

### 5.5.1 CAST Application Intelligence Platform (AIP)

Ein erstes Beispiel eines solchen Tools, wird durch Curtis et al. [26] beschrieben. Hierbei handelt es sich um die CAST Application Intelligence Platform (AIP), welche Software in mehr als 50 Programmiersprachen analysieren kann. Um dies zu ermöglichen, parst AIP zuerst den Code und sucht dann, mittels mehr als 1200 Regeln, nach Verstößen gegen gute Entwicklungspraktiken, gute Architektur und zahlreiche andere Kriterien. Anschließend berechnet die Software für jeden der fünf Gesundheitsfaktoren, Robustheit, Performance, Sicherheit, Übertragbarkeit des Wissens und Veränderbarkeit einen Score und einen gesamten Qualitätsindex für das Produkt. Ebenfalls können das Schuldenkapital und die Zinsen der technischen Schulden berechnet werden. Letztendlich können Dashboards und Berichte ausgegeben werden, die zum Beispiel Probleme in der Architektur visuell darstellen oder auf die Ursachen von Verstößen gegen gute Entwicklungspraktiken verweisen [148]. AIP ist somit ein etabliertes Tool, erfordert aber eine kommerzielle Lizenz, weshalb Startups die nötigen finanziellen Ressourcen zur Verfügung haben sollten.

### 5.5.2 VisminerTD

Ein anderes, vielversprechendes TD-Management Tool, ist das durch Mendes et al. [149] entwickelte VisminerTD. Dieses Web-Tool verwendet einen Repository Miner, um Daten wie Code und Kommentare, aus Git Repositories zu extrahieren. Im Anschluss berechnet es 19 Metriken, wie zum Beispiel die, im Abschnitt 5.3.2, identifizierten sechs Arten an Code Smells und erkennt Duplikate sowie Defekte. Anhand des Tools eXcomment werden ebenfalls, unter anderem durch Tokenisierung, wie im Abschnitt 5.3.7, die Kommentare analysiert. Dadurch wird erkannt, ob darin potenzielle Verbesserungsvorschläge oder Probleme genannt wurden, die somit auf technische Schulden hinweisen [150]. Die Ergebnisse des Tools können dann in mehreren sogenannten Views durch Visualisierung analysiert werden. Die Code Smell und FindBugs Views ermöglichen es anzuzeigen, wo welche Code Smells oder Bugs in Dateien vorhanden sind. Ebenso kann dargestellt werden, wie sich gewisse Metriken und technische Schulden im Laufe der Zeit und Produktversionen ändern. Im Gegenteil zu AIP handelt es sich bei VisminerTD um ein open Source Programm, wodurch es für Startups durchaus attraktiv sein könnte. Allerdings sollte bei der Auswahl beachtet werden, dass das Tool bis jetzt nur Java unterstützt und sich noch nicht so sehr wie AIP bewährt hat.

### 5.5.3 SonarQube

Ein anderes open Source Tool für das TD-Management ist SonarQube [151]. Dieses verwendet einen Datenbank-Server, auf dem sich mehr als 5000 Regeln für gute Entwicklung befinden [152]. Darunter befinden sich Metriken, die die Sicherheit der Software bewerten, die Wartbarkeit, die zum Beispiel durch Code Smells gefährdet wird und die Zuverlässigkeit, die unter anderem durch Defekte wie Bugs leiden kann [153]. Wenn ein Team nun dessen Code in einer IDE pusht und baut, muss dieser als nächstes sogenannte Quality Gates von Sonar [154] passieren. Diese Quality Gates werden durch das Startup konfiguriert, indem es zum Beispiel sagt, wie viele oder welche Code Smells erlaubt sind oder nicht. Hierbei wird durch den SonarQube Webserver anhand der Regeln in der Datenbank überprüft, ob der Code den gewünschten Richtlinien in den Quality Gates entspricht [152]. Nur wenn dies der Fall ist, wird der Code gemerged und für die Produktion freigegeben. Sollte dies nicht zutreffen, so werden die Probleme in sogenannten Tabs gespeichert [151]. In jedem Tab befindet sich ein Bericht, der dem Nutzer aufzeigt, wo sich das Problem befindet und sogar mit Beispielen erklärt wird, wieso es sich um ein Problem handelt. Durch das automatisierte Durchgehen der Quality Gates, wird der Code kontinuierlich auf dessen Qualität geprüft. SonarQube unterstützt mittlerweile mehr als 30

Programmiersprachen und ist kostenlos [151], sodass es eine durchaus interessante Option für Startups darstellt.

### 5.5.4 DebtFlag

Bei einem letzten TD-Management Tool, handelt es sich um das Eclipse Plugin DebtFlag [155]. Dieses ist ein leichtgewichtiges Dokumentationstool für technische Schulden, das sich in der Entwicklungsphase befindet und einen interessanten Ansatz bietet. Sobald ein Entwickler im Code auf ein Element stößt, das er als technische Schulden ansieht, wird dieses entsprechend „geflagt", also markiert. Wird zum Beispiel eine Brain Methode erkannt, wird diese in DebtFlag eingetragen und es werden automatisch die Uhrzeit und die Stelle der Methode im Code vorgefüllt. Der Entwickler definiert dann, um welchen TD-Typen es sich handelt und DebtFlag analysiert, wie sich das TD-Item dementsprechend auf andere Code-Elemente, wie Klassen und Methoden, auswirkt. Das Plugin kann dann zum Beispiel in Eclipse davor warnen, wenn zu viele Abhängigkeiten in einer Methode vorhanden sind. Eine entsprechende Web-Anwendung ermöglicht es auch, sich eine Liste der TD-Items und ihrer entsprechenden Informationen und Abhängigkeiten anzeigen zu lassen. Es kann dadurch ebenfalls vermerkt werden, wenn gewisse TD-Items modifiziert oder gelöst werden. Obwohl das DebtFlag Plugin sehr praktisch ist, muss beachtet werden, dass es bis jetzt nur mit der Eclipse IDE und Java funktioniert. Zudem hängt das Tool, im Gegenteil zu den Vorherigen, besonders von den manuellen Eingaben der Entwickler ab, sodass es nur einen geringen Automatisierungsgrad gewährleistet.

### 5.5.5 Weitere Möglichkeiten

Es gibt noch andere Tools, welche zum Beispiel Kommentare analysieren, um technische Schulden zu erkennen [156] oder sich nur auf präzise Code Smells beschränken. Diese sind allerdings ungenauer und unvollständiger, als die zuvor genannten Werkzeuge und zum Großteil noch Prototypen. Zum Teil könnten diese durch Startups aber bereits eingesetzt werden, um spezifische Bedürfnisse zu erfüllen. Letztendlich müssen Unternehmen, wie es Zazworka et al. [57] beschreiben, die Werkzeuge auswählen, die ihren Bedürfnissen entsprechen. Der Einsatz eines einzigen Tools wird nämlich nur selten auf alle relevanten TD-Items weisen, da jedes Tool gewisse andere Aspekte analysiert.

Es gibt auch leichtgewichtigere Methoden, als den Einsatz von Tools, um gewisse technische Schulden, zumindest teilweise, zu verhindern. Die meisten IDEs schlagen Usern standardmäßig Verbesserungen ihrer Schreibweise vor und warnen zum Beispiel vor Fehlern oder Problemen beim Bauen. Zudem gibt es zahlreiche Erweiterungen, wie zum Beispiel das Eclipse-Plugin JDoedorant, das automatisch gewisse Code Smells, wie Gott-Klassen oder duplizierten Code erkennen kann und dem Entwickler angemessene Refactorings vorschlägt [157].

Wichtig ist es letztendlich, darauf zu achten, dass immer mehr technische Schulden existieren werden, als durch Tools oder IDEs erkannt werden können [62]. Somit können letztere als Unterstützung für das TD-Management eingesetzt werden, ersetzen aber keineswegs die Aufgabe des Teams, technische Schulden aktiv zu vermeiden, zu erkennen und zu reduzieren.

## 5.6 Zusammengefasste Checkliste

Um die Ergebnisse aus Kapitel 5 möglichst übersichtlich darzustellen, fasst die Tabelle 5 diese zusammen. Als Erstes werden die Vorbereitungsmaßnahmen zum Einführen von TD-Management aus dem Abschnitt 5.1 genannt. Es folgen die 16 TD-Typen aus dem Abschnitt 5.3 und die jeweiligen beispielhaften Maßnahmen, um diese zu managen. Letztendlich werden die TD-Management Tools aus dem Abschnitt 5.5 genannt.

*Tabelle 5: Zusammengefasste Checkliste zum Management von technischen Schulden in Startups*

| **Vorbereitungsmaßnahmen** | |
|---|---|
| **TD-Management Champion** | |
| **Zeitbudget** | |
| **Finanzielle Mittel** | |
| **TD-Management Workshop** | |
| **TD-Typ** | **Management** |
| **Design Debt** | Ursachen- und Effekt-Analysemeetings |
| | Cost Benefit Analyse |
| **Code Debt** | Ursachen- und Effekt-Analysemeetings |
| | Intensitätsindexe, JCodeOdor |
| **Test Debt** | Ursachen- und Effekt-Analysemeetings |
| | Test Driven Development |
| | Automated Testing |
| **Architecture Debt** | Ursachen- und Effekt-Analysemeetings |
| | Coupling-Between-Modules Tool |
| **Requirements Debt** | Ursachen- und Effekt-Analysemeetings |
| | 3 Typen an Requirements Debt |
| | Vision Videos |
| **Documentation Debt** | Ursachen- und Effekt-Analysemeetings |
| | Framework für gute Dokumentationsqualität |
| **Versioning Debt** | Ursachen- und Effekt-Analysemeetings |
| | Git, Infox |
| | Bot: 5 Merkmale zum Vergleich von Codeänderungen |
| **Defect Debt** | Ursachen- und Effekt-Analysemeetings |
| | 6 Kostenfaktoren, 6 Entscheidungsfaktoren |
| **Build Debt** | Ursachen- und Effekt-Analysemeetings |
| | Dependency Debt, Zombie Targets, Visibility Debt |
| **Process Debt** | Ursachen- und Effekt-Analysemeetings |
| | Definition of Done |
| **Infrastructure Debt** | Ursachen- und Effekt-Analysemeetings |
| | DevOps |
| | Infrastructure as Code, TOSCA |
| **People Debt** | PANAS-Fragebogen |
| | Senti-Analyzer, Stimmungsanalyse-Tools |
| **Usability Debt** | Ursachen- und Effekt-Analysemeetings |
| | Minimum Viable User Experience Framework |
| **Test Automation Debt** | Ursachen- und Effekt-Analysemeetings |
| | 6 Aspekte für gutes Automated Testing |

| | |
|---|---|
| **Service Debt** | Ursachen- und Effekt-Analysemeetings |
| | Real Options |
| **Security Debt** | 10 Mängel der IEEE Computer Society |
| **TD-Management Tools** | |
| **CAST Application Intelligence Platform (AIP)** | |
| **VisminerTD** | |
| **SonarQube** | |
| **DebtFlag** | |
| **IDE-Korrektur, Plugins (zB. JDeodorant)** | |

# 6. Erweiterung des Entrepreneurial Software Engineering Modells

Anhand der durch die Checkliste gewonnenen Erkenntnisse, wird nun das, im Abschnitt 2.5 präsentierte Entrepreneurial Software Engineering Modell von Brunner et al. [37] [38], erweitert. Es fehlt nämlich im Generellen und somit auch für Startups ein Prozess, der beschreibt, wie technische Schulden in einem Entwicklungsprozess gemanagt werden können [63]. Die Erweiterung dient dazu, diesen zur Verfügung zu stellen und eine Übersicht darüber zu geben, wie technische Schulden bei der Entwicklung eines Minimum Viable Products (MVP) in einem Startup, durch einen leichtgewichtigen Prozess, gemanagt werden können. Um dies zu bewerkstelligen, wurden Schwachstellen im bestehenden Modell anhand der, sich in der Checkliste befindlichen, Beschreibungen, Ursachen und Folgen und der Indikatoren identifiziert. Daraufhin werden, mittels der erklärten Lösungsansätze und Forschungsarbeiten, sowohl das Modell als auch dessen Beschreibungen, erweitert. Das erweiterte ESEM bietet somit eine graphische Unterstützung zur strukturierten Checkliste und erweitert diese, indem es zeigt, wie dessen Erkenntnisse in einem Entwicklungsprozess eingesetzt werden können.

## 6.1 Zusammenfassung des ursprünglichen ESEM

Das zu erweiternde ESEM stellt, wie im Abschnitt 2.5 ausführlicher beschrieben, einen inkrementell-iterativen Entwicklungsprozess dar, in dem einzelne Stages, also Phasen, aufeinander aufbauen [37]. Für ein einzelnes Feature wird in einer ersten Phase (FRGE) geprüft, welche Teil-Features entwickelt werden müssen und mit der vertikalen Entwicklung begonnen. In der nächsten Phase (DUXE) wird geprüft, wie das Interface dieser Features aussehen soll und mit der horizontalen Entwicklung begonnen und die dritte Phase (PLT) dient dazu, diese Änderungen kontinuierlich und automatisiert zu testen. Am Ende eines jeweiligen Entwicklungszyklus werden die Features veröffentlicht. Für jedes weitere Teil-Feature das nötig ist, um das MVP zu vervollständigen, werden erneut solche Entwicklungszyklen durchgeführt und beliebig, benötigte Phasen durchgegangen.
Bei der Entwicklung des MVPs ist es jedoch wichtig, dass das Dringende nicht das Wichtige verdrängt [158]. Vor allem bei iterativer Entwicklung kommt es oft vor, dass trotz guter Entwicklungspraktiken, wie Feature- und Test Driven Development (FDD und TDD) oder auch automatisiertem Testen (AT), schnelle Lösungen den Eleganten vorgezogen werden und sich somit technische Schulden anhäufen [158]. Teams, die iterativ entwickeln, denken oft auch, dass sie dadurch „immun“ gegen technische Schulden seien, obwohl dies keineswegs der Fall ist, sodass sie noch weniger darauf achten [62]. Iteratives Entwickeln, sowie FDD, TDD und AT sind somit zwar gute Anfangspunkte, um einige technische Schulden zu vermeiden, doch es müssen erweiternde Praktiken und Aspekte beachtet werden, um technische Schulden effizient zu managen. Aus diesen Gründen wird nun die Erweiterung des Modells besprochen.

## 6.2 Notation

Für die Erweiterung des ESEMs wird folgende Notation eingesetzt:

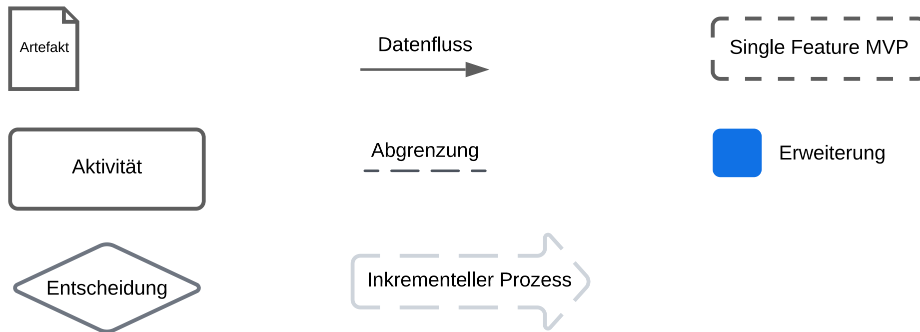


*Abbildung 15: Legende des erweiterten Entrepreneurial Software Engineering Modells*

Farblich blau markierte Elemente stellen graphische Erweiterungen des Modells dar, die im Folgenden besprochen werden.

## 6.3 Feature Requirements Generation & Evaluation – Technical Debt Prevention

Die *Feature Requirements Generation & Evaluation* befindet sich am Anfang des Entwicklungszyklus und ist somit, zusammen mit der *Design & User Experience Evaluation*, die optimale Phase, um TD-Prevention [34] durchzuführen. Diese beiden ersten Phasen müssen also, wie im Abschnitt 2.3.4 erklärt, als zusätzliches Ziel haben, das Erscheinen von technischen Schulden möglichst zu vermeiden.

### 6.3.1 Creative Phase

Bei der ersten Phase des ESEM, der *Feature Requirements Generation & Evaluation*, gibt es mehrere Möglichkeiten der Erscheinung von technischen Schulden. Deshalb wurde sie entsprechend erweitert und in die *Feature Requirements Generation & Evaluation - Technical Debt Prevention (FRGE-TDP)* umbenannt, welche in Abbildung 16 dargestellt ist und deren Funktionsweise nun erklärt wird.

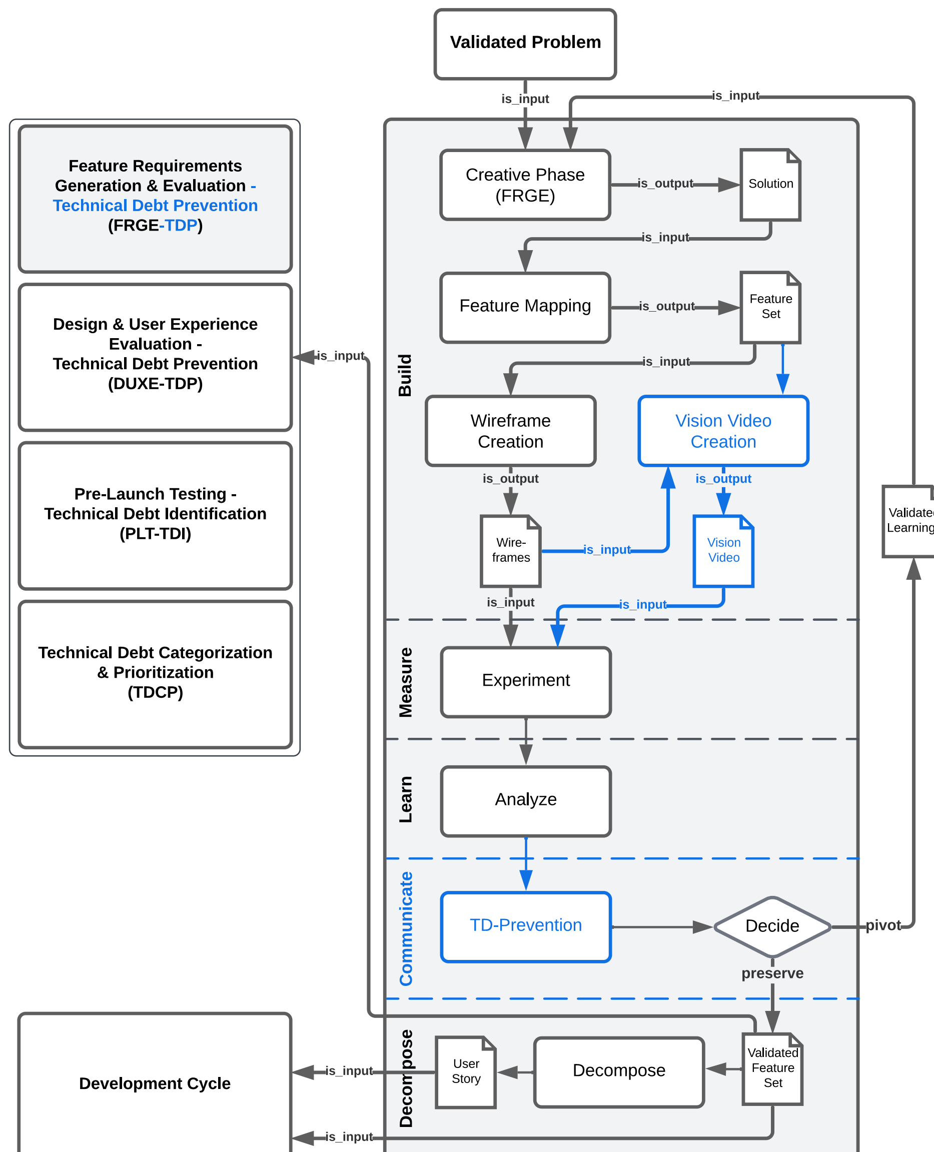


*Abbildung 16: Feature Requirements Generation & Evaluation - Technical Debt Avoidance (FRGE-TDA) Phase, aufbauend auf [38]*

Vor allem in der ersten kreativen Phase, kann es zu People Debt kommen, da verschiedene Teammitglieder unterschiedliche Mengen an Input haben können oder, da sie der Ansicht sind, dass ihre Ideen weniger oder gar nicht wahrgenommen werden. Vor allem bei unstrukturierten Aktivitäten, wie Brainstorming, ist dies wahrscheinlicher. Aus diesem Grund sollten ausschließlich, was die

Teilnahme betrifft, ausgewogenere Methoden, wie die 6-3-5 Methode [159] eingesetzt werden. Bei dieser schreiben 6 Teilnehmer 3 Ideen auf und diese Ideen werden 5-mal ausgetauscht, um entweder erweitert zu werden oder neue Ideen hinzuzufügen [159]
In dieser kreativen Phase und im Rahmen des kontinuierlichen TD-Managements somit in anderen Teambesprechungen, können auch Stimmungsanalysetools eingesetzt werden. Das Management kann dann Ergebnisse von zufälligen Terminen auswählen, diese begutachten und gegebenenfalls entsprechende Rücksprachen mit dem Team führen. Bei der Definition von Anforderungen in dieser Phase, sollten ebenfalls stets die Requirement Smells aus dem Abschnitt 5.3.5 vermieden werden.

### 6.3.2 Vision Video Creation

Es werden, dank der Feature Sets, Wireframes erstellt, um das Skelett des zukünftigen Interfaces darzustellen. Diese Wireframes sind allerdings statisch, dies kann bei der darauffolgenden Experimentierung mit den Early Adopters dazu führen, dass es schwierig ist, das konkrete Ziel, den Ablauf und die Vision der Software zu übermitteln. Aus diesem Grund können die, im Abschnitt 5.3.5 beschriebenen Vision Videos, komplementär zu den Wireframes eingesetzt werden. Durch die bekannten Features und die erstellten Wireframes, können einfache Vision Videos erstellt werden. Diese stellen dar, was das Problem ist, welche Features wie funktionieren würden, um das Problem zu lösen und welches Endergebnis es für die User ergeben würde [92]. Das Vorführen der Wireframes und der Vision Videos während der Experimentierphase stellt sicher, dass bei der folgenden Analyse- und TD-Vermeidungsphase, alle nötigen Informationen zur Verfügung stehen, um technische Schulden vorzubeugen.

### 6.3.3 Experiment

Durch die Experimentieraktivität, in welcher den Usern, dank der Wireframes und Videos, die grobe Funktionsweise der Software vorgeführt wird, können die User ein Gefühl für das Produkt bekommen. Daraufhin können sie im Gespräch oder durch Fragebögen ihren Eindruck und Verbesserungsvorschläge äußern. Die Videos bieten hierbei einen besonders guten Weg Feedback zu erzeugen, da sie sehr schnell klarstellen, ob die Vorstellung der Kunden mit der, des Startups übereinstimmt [92].

### 6.3.4 Analyze

In der Analyseaktivität geht es darum, die zwei Requirements Debt Typen aus dem Abschnitt 5.3.5, *unerfüllte Kundenwünsche* (Typ 0) und *nicht übereinstimmende Umsetzung* (Typ 2) [88] im Voraus zu minimieren. Hierfür werden die Ergebnisse der Experimente sowie die, im Abschnitt 5.3.5 beschriebenen, Methoden eingesetzt. Zusätzlich werden, die zwei durch Brunner et al. [37] und Bosch et al. [160] definierten Fragen gestellt, um die Ergebnisse der Experimente zu analysieren:

- "Reicht das derzeitige Angebot an Funktionen aus, um das Problem des Kunden zu lösen?"
- "Sind die Kunden bereit, das MVP zu testen?"

Die Ergebnisse dieser Fragen müssen ausgewertet werden, um die Durchführung der folgenden Etappe zu unterstützen.

### 6.3.5 TD-Prevention

Vor allem bei der Entwicklung eines Prototypen, der die Zukunft des Startups bestimmen wird, gilt der bekannte Spruch „Communication is key". Es wird zwar immer nach den Wünschen der Kunden, also zum Beispiel der Early Adopters und des Managements geschaut, doch es wird nur selten kommuniziert, inwiefern diese Wünsche mit einer begrenzten Menge an Ressourcen und Zeit

überhaupt machbar sind. Dies kann, vor allem bei Startups, schlimme Konsequenzen haben. Es dürfen also nicht zu viele Features für einen Entwicklungszyklus angenommen werden, damit die Qualität und Wartbarkeit des Produkts nicht darunter leiden, das Produkt rechtzeitig fertig ist und keine leeren Versprechungen gemacht werden [117]. Letztere können nämlich verheerende Auswirkungen auf den Ruf des Produktes haben. Das Ziel ist es also, entsprechend der TD-Prevention [34], das Erscheinen von TD-Items vorzubeugen.

Aus diesem Grund wurde der *Build-Measure-Learn* Zyklus von Ries [22] um eine *Communicate* Etappe erweitert, die ansonsten oft übersehen wird. Bei TD-Communication geht es darum, den Usern und Stakeholdern technische Schulden sichtbar zu machen, damit sie besprochen und gemanagt werden können [34]. Dadurch ist es ebenfalls möglich, potenzielle Probleme zu besprechen, die entstehen könnten, sollten gewisse Features implementiert werden [117]. Sollten also doch technische Schulden entstehen, gehören diese zu den geplanten Schulden [29], was somit keine „schlechten Überraschungen" verursachen würde. Doch auch dem Management müssen technische Schulden vorgestellt werden, damit dieses die Entscheidungen des Teams nachvollziehen kann.

Da beim ersten Durchgang durch das Modell noch wenig entwickelt wurde, können eventuelle zukünftige technische Schulden, zum Beispiel anhand der Informationen und Beispiele aus der Checkliste, geschätzt werden. In den Phasen *Pre-Launch Testing - Technical Debt Identification* und *Technical Debt Categorization & Prioritization*, welche in den Abschnitten 6.5 und 6.6 vorgestellt werden, werden die technischen Schulden bei folgenden Iterationen des Modells identifiziert, sodass dann diese Daten eingesetzt werden können.

Hierfür müssen diese Informationen aber angemessen dargestellt sein, um schnell ein klares Bild über den aktuellen Stand zu schaffen. Im Abschnitt 5.1.1 wurde erklärt, dass ein TD-Management Champion eingesetzt werden sollte, der den Mitarbeitenden, dank seiner Kenntnisse aus der Checkliste, erklärt, was als technische Schulden angesehen werden sollte. Durch diese Kenntnisse sowie die, in den nächsten Phasen, identifizierten Schulden können kontinuierlich TD-Übersichten erstellt werden. Wie diese Übersichten hergestellt werden, wird im Abschnitt 6.6.1 erklärt.

Anhand solcher Übersichten bekommen Stakeholder, Kunden und das Management ein Bild davon, welche und wie viele technische Schulden vorhanden sind oder sich erhöhen werden, sollten gewisse Features entwickelt werden. Sie dienen ebenfalls dazu zu zeigen, wo und weshalb sich welche TD-Items befinden und wie diese zusammenhängen. Die beteiligten Parteien müssen sich dank dieser Darstellungen einigen, welche Features, je nach ihrem Einfluss auf die Schulden, entwickelt werden sollten oder nicht.

Nach der TD-Communication, kann also den zwei Fragen von Bosch et al. [160] letztendlich die Frage hinzugefügt werden:

- „Konnte sich auf gewisse Features geeinigt werden, die möglichst wenig technische Schulden verursachen würden?"

Nur wenn alle drei gestellten Fragen mit „Ja" beantwortet werden können, wird der normale Verlauf des ESEM weiter befolgt und diese validierten Features werden zu User Stories und als Input für den Entwicklungszyklus eingesetzt. Sollte dies nicht der Fall sein, wird ein Pivot durchgeführt [38], bei dem die validierten Erkenntnisse aus der Analyse- und TD-Avoidance-Etappe eingesetzt werden. Hierbei können zum Beispiel die Anforderungen der Kunden und Stakeholder den Gesprächen entsprechend angepasst werden [81]. Dadurch kann eine neue Lösung vorgeschlagen werden, die einfacher zu realisieren wäre und somit weniger technische Schulden verursachen könnte und der Zyklus wird bis zur TD-Communication wiederholt.

Als nächstes geht es darum, einen ähnlichen Prozess zu befolgen, um das Design der Software zu entwickeln.

## 6.4 Design & User Experience Evaluation - Technical Debt Prevention

Ein guter erster Eindruck durch ein ausgereiftes Interface führt dazu, dass sich dies schnell herumspricht und immer mehr Early Adopters die Software verwenden werden [132]. Dies ist vor allem bei der Entwicklung eines MVPs sehr wichtig, da somit mehr Daten in der Experimentieretappe gesammelt werden können. Aus diesem Grund wird in der Phase *Design & User Experience Evaluation - Technical Debt Prevention (DUXE-TDP)*, welche in Abbildung 17 dargestellt ist, das Minimum Viable User Experience Framework von Hokkanen et al. [132] eingesetzt, um das MVP zu designen.

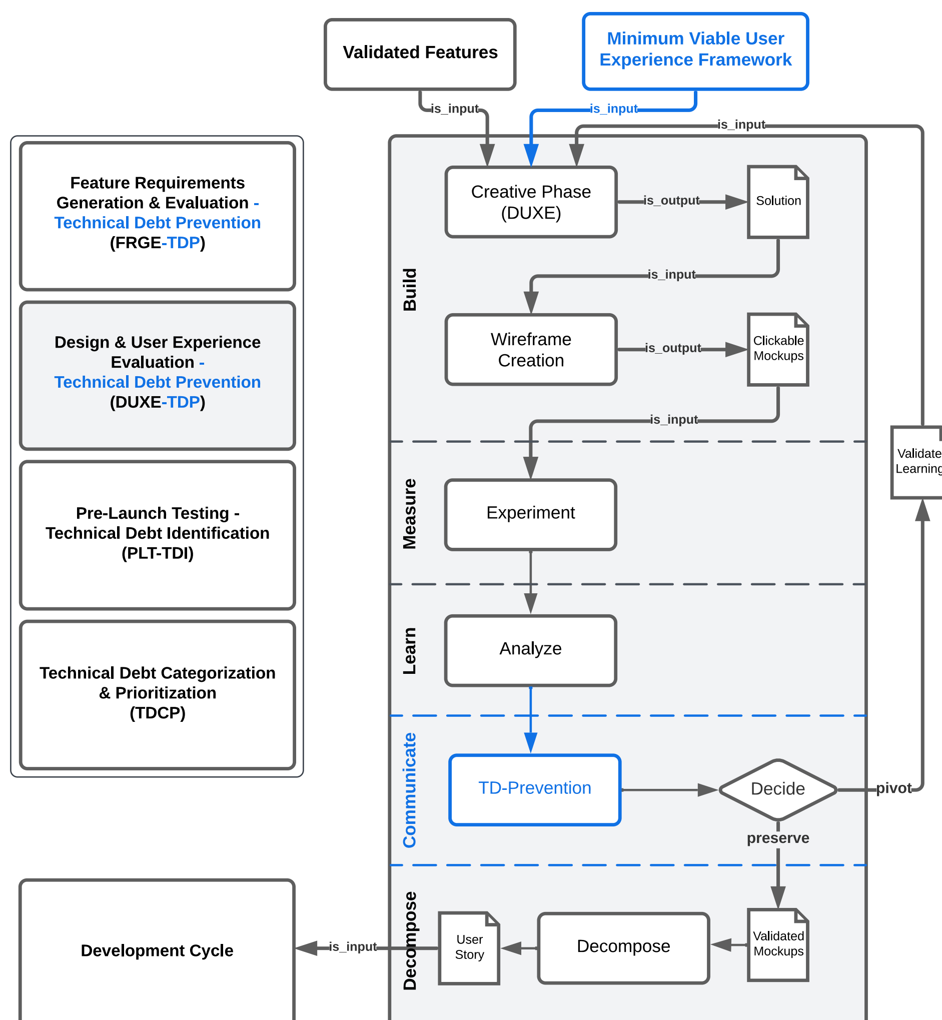


*Abbildung 17: Design & User Experience Evaluation - Technical Debt Prevention (DUXE-TDP), aufbauend auf [38]*

### 6.4.1 Creative Phase & Wireframe Creation

In der kreativen Phase, die, wie bei FRGE-TDP, möglichst allen Mitarbeitenden einen ähnlichen Input ermöglichen soll, kann das Framework unterstützende Richtlinien zur Verfügung stellen. Dadurch werden bereits bei der Entwicklung der konzeptuellen Lösung und danach der Mockups, wichtige Designziele, wie Attraktivität, Zugänglichkeit und Professionalismus beachtet, was zahlreiche Usability Schulden, bereits im Voraus vermeiden kann. Beim Designen ist es ebenfalls wichtig, sich darauf zu fokussieren, das Hauptziel des Frameworks zu erreichen, nämlich „die Idee zu verkaufen“. Dies stellt sicher, dass die Tester bei der folgenden Experimentieretappe genau wissen, worum es sich bei der Software handelt und diese somit präziser bewerten können [132]. Ebenfalls sollte sich bei der

Entwicklung der Lösung und der Mockups immer an Interfaces von Erfolgreichen Unternehmen aus demselben Sektor orientiert werden und nicht immer „das Rad neu erfunden werden“. Dies garantiert, dass das Interface zumindest bereits einigen etablierten Standards entspricht.

### 6.4.2 Experiment

Als nächstes werden die gewonnen Erkenntnisse aus dem Framework und den Mockups in den Experimenten mit den Early Adopters zur Probe gestellt. Hierfür werden die User zusätzlich gefragt, wie sehr die unterschiedlichen Aspekte, zum Beispiel Attraktivität, durch Faktoren wie Neuartigkeit oder visuellen Anspruch, ihrer Meinung nach, eingehalten wurden. Letztendlich können die User anonym auch ausdrücklich danach gefragt werden, wie sehr sie das Produkt weiterempfehlen würden. Die Ergebnisse ermöglichen es somit einerseits abzuwägen, welche Aspekte noch verbesserungswürdig sind und vor allem zu wissen, ob das Hauptziel des Frameworks und jedes Startups „die Idee zu verkaufen“, mit dem aktuellen Lösungsansatz, erreicht wäre.

### 6.4.3 Analyze

Die Analyse- und TD-Prevention-Etappen verlaufen sehr ähnlich zu den Etappen aus FRGE. Es werden jedoch hier folgende Fragen gestellt, um die Ergebnisse der Experimente zu analysieren [38]:

- „Ist die User Experience ausreichend für die ersten Kunden?“
- „Verstehen die ersten Kunden die Nutzung des Produktes vollständig?“

Nach der Beantwortung dieser Fragen müssen deren Antworten, wie bei FRGE, ausgewertet werden und in der nächsten Etappe unter allen Beteiligten besprochen werden.

### 6.4.4 TD-Prevention

Um diese Fragen im Kontext der TD-Prevention zu vervollständigen, wird sich zusätzlich die Frage gestellt:

- „Konnte sich auf ein Design geeinigt werden, das möglichst wenig technische Schulden verursachen würde?“

Die Beantwortung dieser Frage wird, ähnlich zur vorigen Phase, durch die Kenntnisse der Checkliste, sowie die in der Phase *Technical Debt Categorization & Prioritization* generierten TD-Übersichten, unterstützt. Sobald alle Fragen und deren Ergebnisse besprochen werden konnten und alle mit „Ja“ beantwortet werden konnten, wird der normale Verlauf des Modells befolgt. Ist dies nicht der Fall, wird ein Pivot durchgeführt und eine neue konzeptuelle Lösung sowie Mockups erstellt, die den Wünschen der Kunden entsprechen und technische Schulden möglichst vermeiden.
In der nächsten Phase geht es darum, die vertikale Entwicklung von FRGE-TDP und die horizontale Entwicklung von DUXE-TDP zu testen.

## 6.5 Pre-Launch Testing - Technical Debt Identification

### 6.5.1 TD-Identification

Das *Pre-Launch Testing – Technical Debt Identification (PLT-TDI)* welches in der Abbildung 18 dargestellt wird, ist ein besonders wichtiger Part des Modells, wenn es darum geht, technische Schulden, die im System vorhanden sind, zu erkennen, da es unterschiedliche Aspekte des Produkts testet. Auch wenn in den zwei vorigen Phasen so viel wie möglich versucht wurde technische Schulden zu vermeiden, werden trotzdem immer welche entstehen, die nun erkannt werden müssen. Somit ermöglicht es diese Phase TD-Identification durchzuführen, bei der bewusste und unbewusste

technische Schulden verschiedener TD-Typen erkannt werden [34]. Zusätzlich zu den, durch Brunner [38] genannten, Test-Methoden, wurden ergänzend einige hinzugefügt. Da, wie Brunner [38] es beschreibt, die Qualität der Software wichtiger ist als kleine Zeitgewinne.

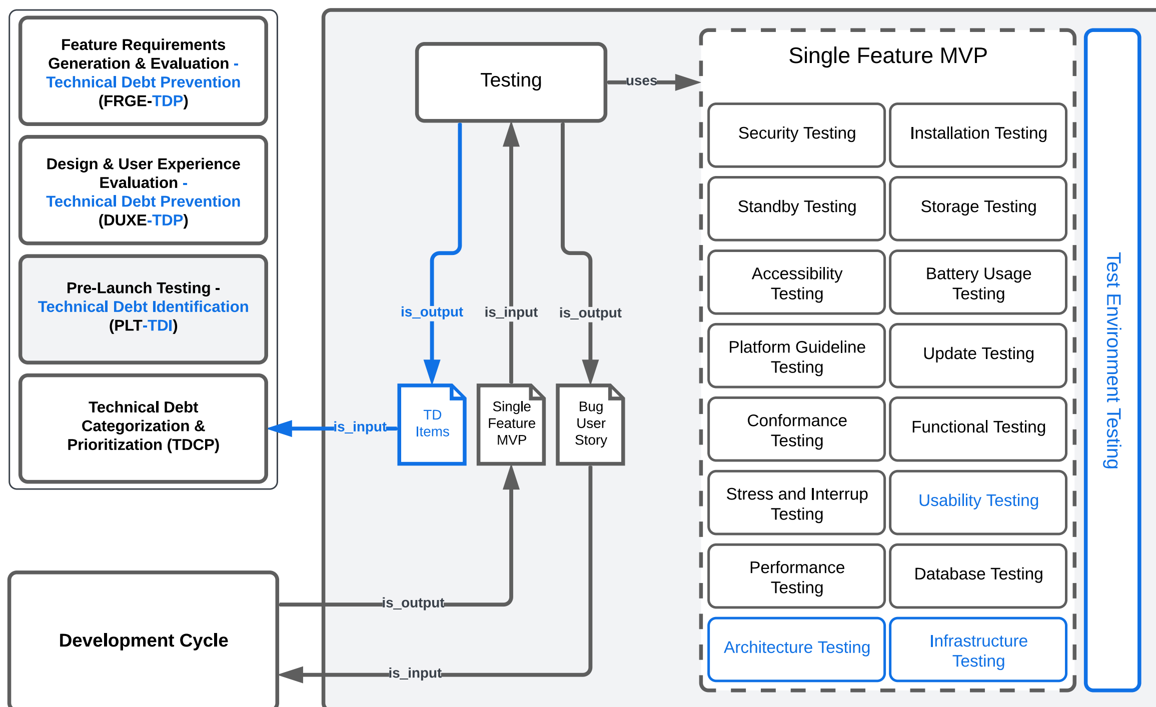


*Abbildung 18: Pre-Launch Testing – Technical Debt Identification (PLT-TDI), aufbauend auf [38]*

### 6.5.2 Erweiterte Testabdeckung

So wurde das Testen der Architektur der Software hinzugefügt, bei dem zum Beispiel, wie im Abschnitt 5.3.4 erklärt, durch Tools wie das von Tvedt et al. [87], die Verbindungen zwischen Komponenten und deren Anzahl analysiert werden können. Somit kann frühzeitig vorgebeugt werden, später ressourcenintensive Architekturanpassung vornehmen zu müssen.
Sollte die Software auf mehreren Geräten verfügbar sein, muss dessen Usability zum Beispiel auch für diese verschiedenen Formate getestet werden, um sicherzustellen, dass sie responsive ist, also, dass das Interface sich unterschiedlichen Bildschirmen anpasst [161]. Wie in DUXE-TDP zu sehen, sollte ebenfalls mit einfachen Befragungen durch Nutzer bewertet werden, inwiefern die Kriterien des Minimum Viable User Experience Frameworks [132] eingehalten wurden.
Da Startups meistens von einer begrenzten Hardwareinfrastruktur oder Webservices abhängen, kann es nützlich sein, regelmäßig automatisierte Tests laufen zu lassen, um diese zu testen. Hierbei kann es sich zum Beispiel um Stresstests handeln, die die Rechenleistung der Services und Infrastruktur zur Probe stellen oder auch der verbleibende Speicherplatz überprüft wird [116]. Im Rahmen von Infrastructure as Code sollte nämlich Infrastruktur genauso behandelt und getestet werden wie normale Software [115].
Ein anderer wichtiger Aspekt der, wie es Wiklund et al. [45] beschreiben, gerne vergessen wird, ist das Dokumentieren der Testfälle selbst. In der PLT-TDI Phase werden unterschiedliche Arten an Tests, wie zum Beispiel automatisierte Unit Tests, durchgeführt. Um spätere Wartungen zu vereinfachen und somit Test Automation- und Test Debt zu reduzieren, muss also stets dokumentiert werden, welche Tests wofür zuständig sind und welche Ergebnisse erwartet werden. Ebenso sollten die anderen, im Abschnitt 5.3.14 beschriebenen, Merkmale beachtet und umgesetzt werden.

Letztendlich wird, wie es Wiklund et al. [45] beschreiben, oft vergessen, die Testumgebung selbst zu testen, sodass diese Ergebnisse produziert, die verfälscht sind. Wird beispielsweise die Performance oder das Updaten einer App auf einem echten Smartphone getestet, muss geprüft werden, dass dieses gewisse Mindestanforderungen erfüllt, welche die Hardware oder das Betriebssystem betreffen sowie, dass es richtig eingestellt ist und eine normale Leistung aufweist [162].
Aus den unterschiedlichen Test-Methoden resultieren verschiedene Ergebnisse. Seien dies nun zum Beispiel Angaben zu Ladezeiten, mögliche Sicherheitsrisiken oder erkannte Bugs. Grobe TD-Items wie etwa „das Interface ist nicht responsive“ können händisch dokumentiert werden. Testing-Tools und TD-Management Tools, wie die aus dem Abschnitt 5.5, können ebenfalls einfache Übersichten der TD-Items für einige TD-Typen (zB. Code Debt, Design Debt, Defect Debt) darstellen. Dadurch können Listen an Arten identifizierter TD-Items, wie zum Beispiel Defekte oder Code Smells, komplementär zu den Bug User Stories erstellt oder automatisch generiert werden. Die TD-Items, die durch diese Phase generiert werden, dienen als Input für die folgende Phase des ESEM; die *Technical Debt Categorization & Prioritization* (TDCP).

## 6.6 Technical Debt Categorization & Prioritization

Diese neue Phase, welche in der Abbildung 19 dargestellt wird, dient dazu, einen Überblick über die technischen Schulden im Projekt zu gewinnen und zu entscheiden, welche als erstes behoben werden müssen.

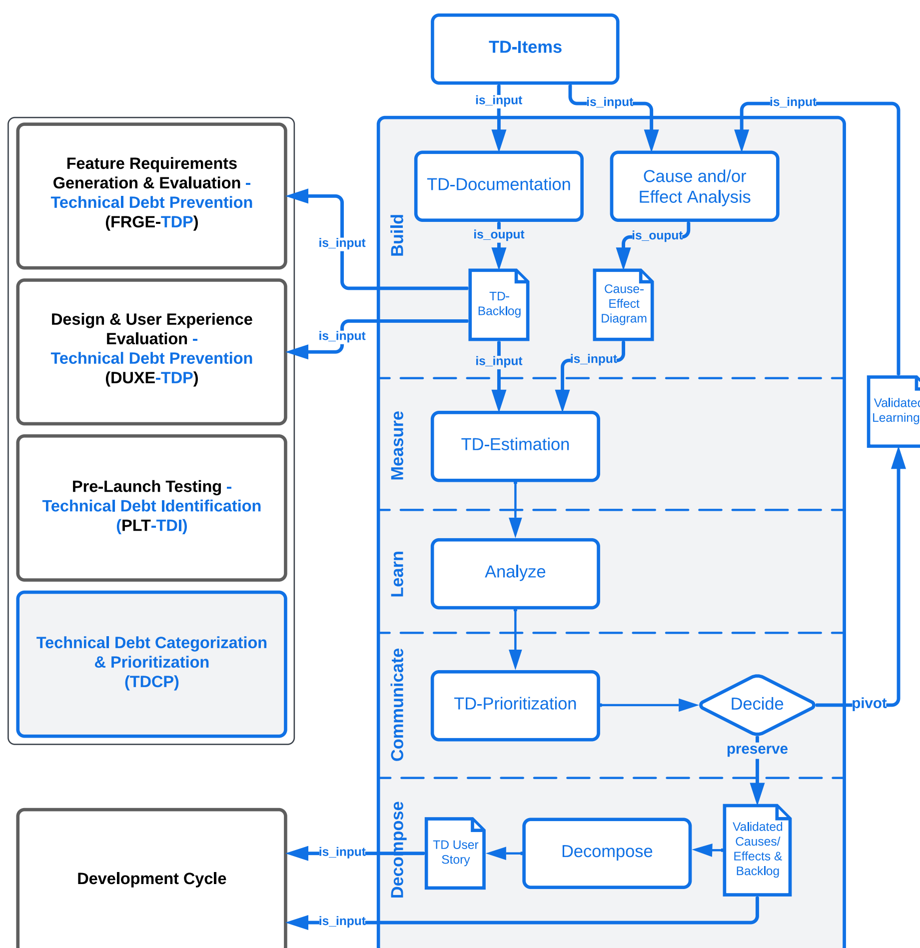

Abbildung 19: Technical Debt Categorization & Prioritization (TDCP), aufbauend auf [38]

### 6.6.1 TD-Documentation

Durch das Testing werden unterschiedliche TD-Items entdeckt, wie zum Beispiel eine schlechte Anpassung der Software an unterschiedliche Bildschirmformate, lange Ladezeiten, viele unverwendete Verbindungen zwischen Codekomponenten, usw. Um den Überblick über diese Items zu behalten, muss also die sogenannte TD-Documentation durchgeführt werden.
TD-Documentation dient dazu, die technischen Schulden auf einheitliche Weise darzustellen, um die Anliegen der Stakeholder adressieren zu können [34].
Hierfür gibt es zahlreiche Möglichkeiten, mit verschiedenen Detailgraden:

- Zum Beispiel könnte eine einfache Liste der Items erstellt werden, in der für jedes Item genannt wird, um welchen TD-Typen es sich handelt, weshalb das Item ein Problem darstellt [144], wo es sich im System befindet und wer dafür zuständig ist [163]. Hierfür würde schon ein einfaches Excel-Spreadsheet ausreichen.
- Eine andere, ausführlichere Methode, welche beispielhaft im Modell dargestellt ist, wäre es ein TD-Backlog zu erstellen, indem die TD-Items, genau wie geplante Features oder Bugs, dem Entwicklungsbacklog hinzugefügt werden [34]. Somit werden technische Schulden mit der gleichen Wichtigkeit, wie andere übliche Backlog-Elemente behandelt und nicht mehr, wie es bis jetzt oft der Fall war, vergessen [62]. Kleinere Bugs und ähnliche kleine TD-Items, müssen nicht händisch in den Backlog eingetragen werden. Sie können beispielsweise in den gewählten Defect Tracking Tools automatisch dokumentiert werden [144].
- Der detaillierteste Weg technische Schulden auf übersichtliche Weise darzustellen, sind sogenannte TD-Dashboards. Diese stellen, pro TD-Item, darüber hinaus auch noch dessen Anzahl dar [34] und können die Verteilung von technischen Schulden in der Software visuell präsentieren [31]. Vor allem sollten Startups keine zusätzliche Zeit damit verbringen, diesen Dashboards aktiv Daten zu liefern, sodass Tools wie SonarQube [151] oder auch Bug Tracker selbst in der Lage sind, Dashboards zu generieren. Diese können zum Beispiel graphisch darstellen, wie viel Code Smells und Bugs im Laufe der Zeit vorhanden sind und wo [164], um TD-Monitoring [34] zu erleichtern. Zudem können sie zeigen, wie gewisse TD-Items zusammenhängen [155] und Entwicklungen von Schulden und Code-Metriken verbildlichen. Sie können auch einen duplizierten Code erkennen, um zu verhindern, diesen unnötig zu bauen. Letztendlich können sie ebenfalls die aktuellen finanziellen Kosten einer gewissen Menge an technischen Schulden, zum Beispiel durch die Anzahl an Defekten, schätzen [35].

### 6.6.2 Cause and/or Effect Analysis

Anhand der TD-Items aus dem Testing können die, von Rios et al. [47] entwickelten, Ursachen- und Effekt-Analysemeetings durchgeführt werden. Drei bis fünf Mitarbeiter sortieren die Items anhand der Beschreibungen aus der Checkliste und ordnen sie ihren TD-Typen zu, wodurch die TD-Kategorisierung stattfindet. Dann verwenden sie das entsprechende Ursache-Wirkungs-Diagramm des TD-Typen, um die potenziellen Ursachen für ihre Items zu identifizieren sowie Folgen, die die Items mit sich ziehen können. Die Diagramme der TD-Typen können dann personalisiert werden, indem die Ursachen und Folgen, denen, im Startup bereits vorhandenen, angepasst werden.

### 6.6.3 TD-Estimation

Als nächstes geht es darum, das TD-Measurement durchzuführen, bei dem die Menge an technischen Schulden im ganzen System geschätzt wird [34]. Hierfür werden als Input die zuvor erstellte TD-Übersicht und die Ursache-Wirkungs-Diagramme eingesetzt. Wurde zum Beispiel eine TD-Liste oder

ein TD-Backlog erstellt, können die Entwickler, wie es Guo et al. [165] beschreiben, besprechen, wie hoch sie die Menge gewisser Items und Schulden schätzen. Diese Aufgabe wird präziser und einfacher, wenn zuvor TD-Dashboards erstellt wurden, da diese ja schon, durch den Einsatz von Tools und Metriken aus der Checkliste, die Anzahl vieler Items [34] graphisch darstellen [31].

### 6.6.4 Analyze

Es folgt die Analysephase, in der es nun darum geht, die erstellten Artefakte zu studieren, um im Anschluss zu definieren, wie wichtig gewisse TD-Items sind. Um alle Parteien bei der Analyse zu unterstützen, kann die TD-Dokumentation präsentiert werden, die Ursache-Effekt-Diagramme, die Ergebnisse der TD-Estimation, sowie präzise Ergebnisse von ausgewählten Tools aus der Checkliste. Dank dieser Tools können auch Informationen, wie die die Implementierungszeit und die priorisierten Schweregrade, als Hilfsmittel dienen. Tools können Items priorisieren, indem sie zum Beispiel automatisierte Cost-Benefit Analysen durchführen oder Intensitätsindexe berechnen und einige Kostenfaktoren, wie im Abschnitt 5.3.8 ermitteln.

### 6.6.5 TD-Prioritization

Das Ziel von TD-Prioritization ist es, zu entscheiden, welche TD-Items als erstes behandelt werden sollten [34]. Dadurch kann ein priorisierter TD-Backlog, zum Beispiel in Form eines Kanban Boards, erstellt werden [166]. Hierfür müssen sich die Teammitglieder, das Management und idealerweise Kunden und Stakeholder zusammensetzen. Diese Meetings müssen nicht bei jeder Iteration des Modells durchgeführt werden, doch sie sollten zumindest bei der Implementierung wichtiger Features stattfinden. Anhand der Häufigkeit der jeweiligen Items, deren Implementierungsaufwand, der Wichtigkeit für die Stakeholder, der Informationen der Tools und auch dem Bereich des Startups, muss zusammen entschieden werden, welche Items vorrangig sind [15]. Der Bereich ist daher wichtig, da ein Startup, das zum Beispiel eine Banking App anbietet, einen hohen Wert auf Security Debt-Items legen sollte. Die Wichtigkeit für die Stakeholder spielt ebenfalls eine große Rolle, da es etwa passieren kann, dass ein technisch wichtiges TD-Item erst später behandelt wird, da die Stakeholder darin keinen großen Mehrwert sehen [15]. Ebenso kann es vorkommen, dass ein technisch unwichtiges, aber sehr schwer zu behebendes TD-Item, für die Stakeholder einen hohen Wert hat, aber nicht für das Startup. Als nächstes müssen sich die Beteiligten folgende Fragen stellen:

- „Waren die präsentierten Informationen klar genug, um Stakeholder, Management und Kunden, die aktuellen Schuldenlage lückenfrei zu erklären?“
- „Konnte sich auf eine Priorisierung der TD-Items geeinigt werden?“

Können diese Fragen mit „Ja“ beantwortet werden, wird dem weiteren Verlauf des Modells gefolgt. Ist dies nicht der Fall, muss ein, mehr oder wenig starker, Pivot durchgeführt werden.
Wurde zum Beispiel durch die Kunden nicht genau verstanden, wie sich gewisse Items im Laufe der Zeit entwickeln oder weshalb gewisse Items in der Dokumentation nicht dargestellt wurden, muss diese, anhand der gewonnenen Kenntnisse, überarbeitet werden.
Durch das Gespräch neu erkannte Ursachen für Schulden und ihre Folgen, können ebenfalls die Ursache-Wirkungs-Diagramme erweitert werden. Es findet hier dementsprechend ebenfalls Continuous Learning statt, da der Umfang der Schulden immer besser verstanden wird und die Darstellungen kontinuierlich verbessert werden.

### 6.6.6 Decompose

Sobald positiv auf die Fragen geantwortet werden konnte, werden die validierten Ursachen und Effekte sowie der priorisierte TD-Item Backlog, in sogenannte TD User Stories heruntergebrochen. Diese sind ähnlich zu Bug User Stories, doch unterscheiden sie sich dadurch, dass sie nicht Bugs beschreiben, sondern Aspekte der Software, die entweder fehlen oder falsch umgesetzt wurden. Beispiele hierfür wären: „Ich möchte ein Gefühl von Sicherheit bekommen, wenn ich die App nutze“ oder „Ich möchte einfach erfahren können, wozu eine gewisse Methode im Code dient“. Diese Stories, sowie die ausgewählten Ursachen und Effekte, werden dem Development Cycle übergeben.

## 6.7 Development Cycle

Der Development Cycle, welcher in der Abbildung 20 dargestellt ist, stellt alle Methoden dar die eingesetzt werden, um aus den Artefakten der Phasen, ein funktionierendes Minimum Viable Product zu erstellen. Werden bei der Entwicklung technische Schulden allerdings nicht aktiv gemanagt, werden sich diese bei jedem Zyklus, potenziell exponentiell, vervielfachen [167]. Darum wurde dieser Zyklus, dank der Erzeugnisse der Phasen und zusätzlicher TD-Management-Methoden, entsprechend erweitert.

*Abbildung 20: Development Cycle, aufbauend auf [38]*

### 6.7.1 Feature- und Test Driven Development und Automated Testing

Als erstes werden, zusammen mit den Anforderungen an das MVP aus den Phasen FRGE-TDP und DUXE-TDP und den Bug User Stories aus PLT-TDI, die priorisierten TD User Stories aus TDCP an die Entwicklung übergeben. Dort werden diese Informationen eingesetzt, um sie, durch Feature- und Test Driven Development, wie normale User Stories in der Software umzusetzen. Das Ziel ist es, dank der TD User Stories, das TD-Repayment zu erleichtern, bei dem es darum geht, die technischen Schulden,

die nicht vermieden wurden, abzubauen, also sie „zurückzuzahlen“ [34]. Um bei der Entwicklung Versionierungs-Probleme zu vermeiden, kann auf Tools wie Git zurückgegriffen werden. Das Automated Testing überprüft im Anschluss, durch PLT-TDI, kontinuierlich und über alle Methoden, die entwickelte Software, wobei alle Richtlinien aus Abschnitt 5.3.14, beachtet und umgesetzt werden müssen.

### 6.7.2 TD-Management

Im Entwicklungszyklus wurde das TD-Management integriert, das, genau wie das Automated Testing, eine Querschnittsfunktion ist, die sich über alle Methoden des Zyklus hinaustreckt. Es stellt alle im Abschnitt 2.3.4 beschriebenen TD-Management-Aktivitäten dar, welche in den Phasen des ESEM vorkommen, sowie zusätzliche Praktiken, die während des Entwicklungszyklus eingesetzt werden können, um optimal mit technischen Schulden umzugehen.

Das TD-Management bekommt, zusätzlich zu den Informationen aus allen Phasen, die Ursachen und Effekte, sowie das priorisierte Backlog aus TDCP übergeben. Diese können, unterstützend zu den Bug User Stories eingesetzt werden, um den Management-Prozess dem Unternehmen anzupassen. Sind zum Beispiel eine zu enge Deadline und ein Mangel an Refactoring die Ursachen für viele TD-Typen, muss in den Entwicklungszyklen entsprechend das Beheben von technischen Schulden zeitlich miteinberechnet werden. Das Backlog ermöglicht es währenddessen, wenn nötig, präzise Informationen zu den TD-Items zu bekommen und den Fortschritt des TD-Managements im Zyklus mitzuverfolgen.

Ein anderer wichtiger Aspekt des TD-Managements im Entwicklungszyklus ist, dass wichtige Informationen konstant dokumentiert werden müssen. Um zum Beispiel People Debt bei neuen Mitarbeitenden oder Kündigungen [122] zu verhindern, muss ein Wiki gehalten werden, indem Ratschläge, Hinweise und gute Praktiken dokumentiert werden. Dadurch wird richtiges Verhalten vereinfacht, die Lernkurve flacher und wichtige Informationen gehen nicht verloren. Es sollte generell möglichst genügend und angemessen dokumentiert werden. Sei dies nun durch FRGE-TDP oder DUXE-TDP entwickelten Code oder für PLT-TDI erstellte Tests. Dies unterstützt das TD-Management deutlich, da Dokumentation die Übersicht vereinfacht und somit auch das Beheben und Vermeiden von technischen Schulden. Letztendlich sollten Entwickler auch kontinuierlich dokumentieren, wie viel Zeit sie mit TD-Management verbringen [168]. Dies kann entweder in einer Zeitverwaltungssoftware des Startups hinterlegt werden oder einfach während Meetings genannt werden. Diese Informationen können, zusätzlich zu den TD-Backlogs und den Ursache-Wirkungs-Diagrammen, in TDCP eingesetzt werden, um die Messung und Priorisierung der Schulden durchzuführen.

Es sollte auch die Stimmung im Startup regelmäßig gemessen werden, sei dies durch automatisierte Stimmungsanalysen oder durch das Beantworten von Panas Fragebögen. Letztere könnten zum Beispiel einmal die Woche durch die Teammitglieder beantwortet werden, damit das Management beobachten kann, wie sich die Stimmung, nach Änderungen am Prozess, im Laufe der Zeit verändert.

Ebenfalls dient die, im Abschnitt 6.3.5 erklärte, TD-Communication nicht nur dazu, den aktuellen Stand an technischen Schulden zu kommunizieren. Es muss auch zwischen Entwicklern über technische Schulden gesprochen werden, um voneinander zu lernen und somit andere TD-Typen und Items zu erkennen [169]. Um die Kommunikation und Zusammenarbeit zu garantieren, ist es also, wie es Yli-Huumo et al. [15] beschreiben, vor allem wichtig, dass es eine gute Einstellung dem TD-Management gegenüber gibt. Dies kann am besten bewerkstelligt werden, indem technische Schulden aktiv gemanagt werden und die positiven Ergebnisse dieser Maßnahmen erkannt werden.

### 6.7.3 Quality Gate

Bevor die Änderungen in jedem Zyklus per Continuous Integration der Software hinzugefügt werden, muss, wie es Davis [13] beschreibt, geprüft werden, ob diese auch gewisse Merkmale erfüllen, die bestätigen, dass nur wenig technische Schulden vorhanden sind. Es handelt sich bei dieser Etappe also um eine Qualitätskontrolle, welche zusammen mit dem automatisierten Testen aus der Continuous Integration überprüft, ob die Änderungen so überhaupt integriert werden dürfen.

Hierfür gibt es mehrere Möglichkeiten, wie zum Beispiel ein strengeres Definition of Done (DoD) [13] oder den Einsatz von Quality Gates [170], wie sie SonarQube einsetzt. Diese Quality Gates ermöglichen es, automatisiert gewisse Aspekte des DoD zu prüfen. Zum Beispiel kann eine obere Grenze für die Menge an duplizierten Code, Bugs oder Code Smells gesetzt werden. Es kann ebenfalls definiert werden, wie hoch die Testabdeckung sein soll oder die Wartbarkeit der Software oder wie mehrere andere Kriterien gemessen werden. Soll der Code im Entwicklungszyklus gepuscht werden, wird überprüft, ob diese Kriterien erfüllt sind. Ist dies nicht der Fall, kann zum Beispiel durch die Einsichten von SonarQube oder den TD-Dashboards beobachtet werden, wo sich wie viele und welche Probleme befinden [170]. Daraufhin können erneut die benötigten Teile des ESEM durchgegangen werden, bis die Software alle Bedingungen erfüllt. Somit kann sichergestellt werden, dass nur eine gewisse, bekannte Menge an technischen Schulden eingegangen wird und das Produkt den gewünschten Standards entspricht.

### 6.7.4 Continuous Integration & Improvement

Continuous Integration hat, durch das automatisierte Prüfen der Software aus PLT-TDI und der Quality Gates, einen positiven Einfluss auf technische Schulden [171]. Doch um sicherzustellen, dass auch bei folgenden Iterationen immer weniger technische Schulden produziert werden, müssen sowohl das Team als auch die Prozesse kontinuierlich verbessert werden.

Das Befolgen des ESEM kann dazu dienen, Process Debt zu reduzieren, da ein konkretes Verfahren, das technische Schulden reduziert, zur Verfügung steht. Es muss daher im Startup sichergestellt werden, dass dieser Prozess auch befolgt wird. Sollten Abweichungen entdeckt werden, kann sogenanntes „Reflective Improvement“ eingesetzt werden [144]. Bei diesem geht es darum, dass die Entwickler ihre Arbeit zum Beispiel einmal in der Woche unterbrechen und zusammen besprechen, was am Prozess funktioniert oder nicht [144]. Entsprechend können dann entweder die Aufgaben der Teammitglieder oder der Prozess angepasst werden. Auch die besprochenen Stimmungsanalysen sollen dazu dienen, die Teamdynamik nach und nach zu verbessern, indem Probleme frühzeitig erkannt und besprochen werden. Durch diese Maßnahmen kann also, parallel zur kontinuierlichen Entwicklung, das Team, durch sogenannte kontinuierliche Verbesserung, effizienter werden [172]. Das Ziel ist es, nicht nur das Produkt im Laufe der Zeit zu verbessern, sondern ebenfalls das Startup selbst, damit es im Falle eines Produkterfolgs, die nötige Leistung vorweisen kann.

### 6.7.5 Continuous Delivery & Maintenance

Continuous Delivery ermöglicht es, technische Schulden zu reduzieren, da zu jedem Zeitpunkt Software bereitsteht, die veröffentlicht werden kann, um zusätzliche Features hinzuzufügen [173]. Diese Continuous Delivery wird deutlich vereinfacht, wenn Versioning Debt vermieden wird, indem sichergestellt wird, dass alle User, falls möglich, dieselbe Version der Software verwenden, um Kompatibilitätsprobleme zu vermeiden [80]. Aus diesem Grund kann zum Beispiel „Software as a Service“ eingesetzt werden [80]. Bei dieser Art der Softwarebereitstellung ist dieselbe und neueste Software für alle User verfügbar, indem sie durch das Startup oder einen ausgewählten Service Provider zur Verfügung gestellt wird und kontinuierlich geupdatet wird [174]. Letztendlich sollte die

bereitgestellte Software nicht nur kontinuierlich erweitert, sondern auch gewartet werden, damit sie auf Dauer die bestmögliche Erfahrung für die Nutzer garantiert und Ausfälle, welche vor allem in der Anfangsphase kritisch sind, minimiert werden.

## 6.8 Zusammengefasstes ESEM

Abbildung 21 stellt das zusammengefasste, erweiterte ESEM dar. Es beinhaltet die erweiterten Phasen FRGE-TDP, DUXE-TDP und PLT-TDI, sowie die hinzugefügte TDCP-Phase.

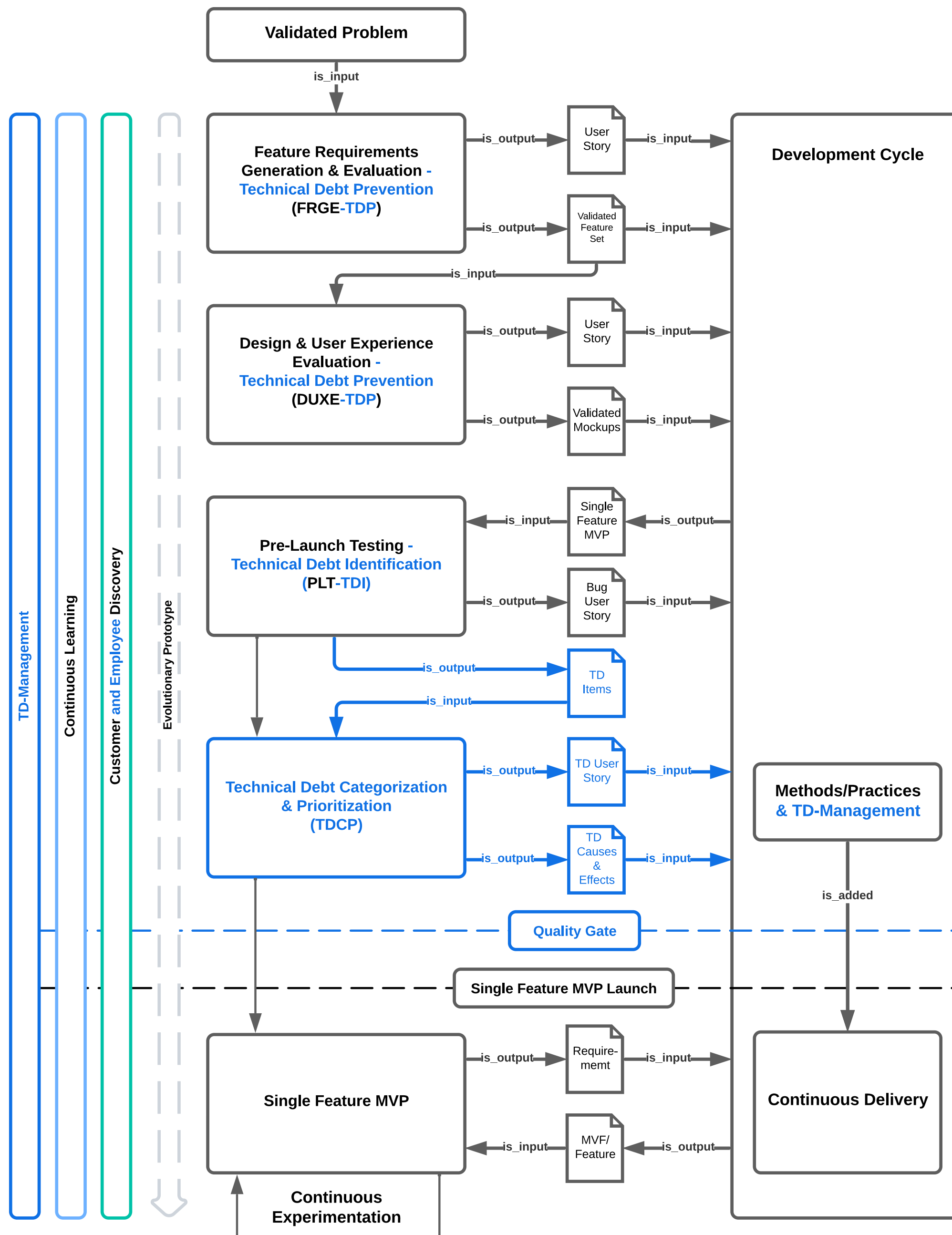


*Abbildung 21: Übersicht des erweiterten ESEMs, aufbauend auf [38]*

Wie in der Checkliste, durch die Zuordnung der TD-Typen zu den unterschiedlichen Entwicklungsphasen und durch die Beschreibungen der Phasen, zu sehen, ist das TD-Management keine einzelne Phase, sondern streckt sich über den ganzen Entwicklungszyklus hinaus. In allen Aspekten der Softwareentwicklung gibt es die Möglichkeit der Entstehung von technischen Schulden und somit der Bedarf für TD-Management.

Zusätzlich ist es, im Rahmen der Customer and Employee Discovery wichtig, nicht nur zu verstehen, was die Kunden sich wünschen. Es muss, wie in den Beschreibungen der Phasen erklärt, auch kontinuierlich dazugelernt werden, wie gut die eigenen Mitarbeiter zusammenarbeiten, welche Verbesserungsvorschläge sie haben und wie das Team effizienter operieren kann. Ebenfalls müssen die Mitarbeitenden, im Rahmen des Continuous Learnings, konstant weitergebildet werden, damit eventuelle Missverständnisse und ein Mangel an Knowhow beseitigt werden. Hierfür können zum Beispiel Experten um Unterstützung gebeten werden und die Mitarbeiter ab und zu, in benötigten Bereichen, geschult werden [144]. Dies kostet zwar anfangs an Ressourcen, doch es verhindert spätere, größere Komplikationen.

Das Zusammenspiel der Phasen und deren Aktivitäten, kann es ermöglichen, den iterativ-inkrementellen Entwicklungszyklus, dank der Erkenntnisse der Checkliste und der Literatur, so schuldenfrei wie möglich zu gestalten, um Startups die Entwicklung eines erfolgreichen MVPs zu erleichtern.

# 7. Diskussion

Es werden nun die Ergebnisse der Arbeit sowie deren Einschränkungen und Erweiterungen diskutiert.

## 7.1 Antworten auf die Forschungsfragen

Die in Abschnitt 1.3.1 definierten Forschungsfragen haben als Leitfaden gedient, um die Problemstellung zu lösen und werden nun, anhand der gewonnenen Erkenntnisse, beantwortet.

- Welche TD-Typen treten in Unternehmen am häufigsten auf?

In der Checkliste wurden 16 TD-Typen genannt, welche studiert wurden und deren Vorkommen in Unternehmen wissenschaftlich dokumentiert und gemessen wurde. Wie das Ranking aus Abschnitt 4.2.3 beweist, handelt es sich bei den häufigsten fünf TD-Typen, in absteigender Reihenfolge, um: Design Debt, Code Debt, Test Debt, Architecture Debt und Requirements Debt. Je nach Startup und dessen Aktivitätsbereich, können diese Ergebnisse aber natürlich variieren, doch es handelt sich um die Typen, die bei TD-Management am ehesten beachtet werden sollten.

- Was verursacht das Erscheinen dieser TD-Typen und welche Folgen zieht dies mit sich?

Wie durch die Checkliste und die verschiedenen Ursache-Wirkungs-Diagramme zu sehen, gibt es eine Vielzahl an verschiedenen Ursachen und Folgen für die Entstehung von technischen Schulden, welche je nach TD-Typ, unterschiedlich stark ausgeprägt sind. Generell ist allerding zu beobachten, dass ein Mangel an Zeit und Organisation, die wichtigsten Ursachen sind. Bei den häufigsten Folgen handelt es sich um eine geringe Wartbarkeit und viele Überarbeitungen der Software, was wiederum zu größeren Zeitverlusten führt. Einerseits wollen Startups also so viel wie möglich an Ressourcen sparen, doch dies führt zu nur noch mehr Arbeit und Verspätungen, solange der Zyklus nicht aufgebrochen wird.

- In welcher Phase der Softwareentwicklung sind diese TD-Typen besonders relevant?

Wie es die Checkliste und dessen Studien belegen, kommen TD-Typen in allen Phasen der Softwareentwicklung vor. Einige Typen, wie zum Beispiel Code Debt, können präzise einer Softwareentwicklungsphase zugeordnet werden, sodass sie darin am relevantesten sind. Andere hingegen, wie zum Beispiel People Debt, sind durch eine Vielzahl an involvierten Faktoren, wie dem menschlichen, phasenübergreifend. TD-Typen sind also, wie in der Checkliste für jeden Einzelfall beschrieben, in einer gewissen, mehreren oder allen Phasen eines Softwareentwicklungsprojektes von Bedeutung.

- Welche Möglichkeiten gibt es, um diese technischen Schulden zu vermeiden?

Je nach TD-Typ gibt es, wie in der Checkliste und dem erweiterten ESEM erklärt, verschiedene Ansätze, um diese zu vermeiden. Einerseits gibt es toolunterstützte, rechnerische Ansätze, die unter anderem gewisse Charakteristiken von Codedateien analysieren können. Ebenfalls gibt es Teamaktivitäten, wie spezifische Meetings, die es beispielsweise ermöglichen, einfacher über Themen wie Ursachen und Folgen zu entscheiden. Es gibt visuelle Methoden und Modelle, die TD-Management Methoden und Aktivitäten anschaulich erklären. Letztendlich gibt es ebenfalls graphische oder schriftliche Richtlinien und Anweisungen, die unter anderem richtiges Verhalten und Praktiken beschreiben. Es existieren somit eine Bandbreite an Möglichkeiten, technische Schulden zu vermeiden, doch es ist das Zusammenspiel dieser Ansätze, das angemessenes TD-Management ermöglicht.

- Inwiefern ermöglichen es die gewonnenen Ergebnisse, einen für Startups etablierten Entwicklungsprozess zu erweitern, um technische Schulden zu vermeiden?

Die unterschiedlichen Ansätze der Checkliste decken alle, durch Li et al. [35] und Yli-Huumo et al. [15] definierten, TD-Management Aktivitäten ab und wie sie für 16 unterschiedliche TD-Typen eingesetzt werden können. Aufbauend auf dem Entrepreneurial Software Engineering Modell und diesen gewonnenen Kenntnissen sowie den Zuordnungen der Entwicklungsphasen, konnten Schwachstellen im bestehenden Modell zuerst erkannt und dann entsprechend ausgebessert werden. Somit konnte ein inkrementell-iterativer Entwicklungsprozess erhalten werden, in dem TD-Management ein grundlegender Baustein ist.

- Ist es für Startups immer von Vorteil technische Schulden zu vermeiden?

Ob und welche technischen Schulden vermieden werden sollen, hängt von mehreren Faktoren ab, die in Abschnitt 7.4 genauer besprochen werden. Im Generellen gilt, dass technische Schulden manchmal bewusst eingegangen werden können, aber nach einer gewissen Zeit trotzdem immer zurückgezahlt werden sollten [61]. Hierfür sollte beachtet werden, welches Softwareprodukt entwickelt wird, da je nachdem, gewisse unwichtige TD-Typen vorerst vernachlässigt werden können [80]. Wie qualitativ hochwertig ein Produkt anfangs sein soll, hängt ebenfalls davon ab, wie viel Konkurrenz im betroffenen Sektor vorhanden ist [80].

- **Hauptforschungsfrage**: Wie können unterschiedliche Typen an technischen Schulden in Startups in verschiedenen Phasen der Softwareentwicklung vermieden werden?

Durch die Checkliste und das erweiterte Entrepreneurial Software Engineering Modell, werden zwei komplementäre Zusammenfassungen bereitgestellt, die erklären, wie technische Schulden in Startups, mit einer begrenzten Menge an Ressourcen, vermieden werden können. Indem die Startups die Checkliste einsetzen, können sie zuerst ihr Team auf das TD-Management vorbereiten. Dank der Erklärungen und Indikatoren können sie erkennen, welche TD-Typen in ihrem Unternehmen vorkommen oder vorkommen können. Dadurch und mittels der zusätzlichen Ursachen und Effekten, können sie dann entscheiden, welche TD-Typen für sie am relevantesten sind. Durch die Anwendung des ESEMs und der, den TD-Typen entsprechenden, Methoden aus der Checkliste können dann durch die Durchführung aller TD-Management Aktivitäten in den respektiven Entwicklungsphasen, Teile aller TD-Typen vermieden und kontrolliert werden.

## 7.2 Neuheiten der Arbeit

Das Beantworten der Forschungsfragen hat zu mehreren neuen Erkenntnissen, bezüglich technischer Schulden in Startups, geführt.
Eine zentrale Erkenntnis ist, dass Security Debt als der „vergessene TD-Typ“ in der Forschung zu technischen Schulden angesehen werden kann. Das Ranking aus Abschnitt 4.2.3 und die Checkliste zeigen, dass Security Debt im Vergleich zu allen anderen Typen als unwichtiger betrachtet wird und somit an letzter Stelle landet. An welcher Stelle Security Debt eher platziert sein sollte, bleibt offen, doch es ist sicher, dass eine höhere Position nötig wäre, da sie ein grundlegender Baustein jeder seriösen Software sein sollte. Wie es McGraw [175] beschreibt, ist Software Security nämlich von hoher Wichtigkeit, was vielen Entwicklern auch bewusst ist, doch meistens wissen sie nicht genau, wie sie ein sicheres System erstellen können. Es wird oft davon ausgegangen, dass das eigene Unternehmen keine Angriffe erleben wird, sodass die Gefahr unterschätzt wird und die entsprechende Verteidigung mangelhaft ist. Allein 2023 haben Angriffe durch Cyberkriminalität in Deutschland allerdings 206 Milliarden Euro Schaden angerichtet und verschiedenste Unternehmen betroffen [176], sodass keine Firma davor sicher ist. Zudem sollten vor allem Startups angemessene Sicherheitsmaßnahmen einrichten, da zum Beispiel ein Datendiebstahl bei den ersten Kunden einen Vertrauensbruch und somit wahrscheinlich das frühzeitige Ende des Unternehmens bedeuten würde.

Das Vermeiden von technischen Schulden bleibt ein relativ neues Feld, sodass dieses, vor allem im Bereich von Startups, nur sehr wenig erforscht wurde. Es wurde daher entsprechend recherchiert und erkannt, dass es trotzdem durchaus möglich sein sollte TD-Management, dessen Aktivitäten und Methoden in einem Startup zu integrieren, das über wenige Ressourcen verfügt. Trotz der Vielzahl an TD-Typen ist es möglich einen leichtgewichtigen Prozess einzusetzen, um Teile der technischen Schulden, im natürlichen Verlauf der Entwicklung, ohne großen Mehraufwand zu vermeiden. Dies wurde erzielt, indem auf einem bewiesenen Modell, dem ESEM, aufgebaut wurde und bewährte TD-Management Methoden und Aktivitäten darin integriert wurden.
Selbstverständlich können aber nicht sämtliche TD-Items vermieden werden, da dies selbst in Unternehmen mit dedizierten TD-Management Teams, nicht möglich ist [108]. Es sind nämlich, wie durch das Studieren von Forschungsarbeiten erkannt, viel zu komplexe Zusammenhänge zwischen den TD-Typen vorhanden, sodass jede neue Entwicklungsaktivität neue TD-Items in verschiedenen Bereichen des Projekts verursachen kann.

Das Ranking der TD-Typen durch vielseitige Aspekte über mehrere Forschungsarbeiten ist ebenfalls neu und weist darauf hin, dass einige TD-Typen deutlich öfter in Projekten vorkommen als andere oder zumindest als wichtiger angesehen werden. Die resultierende Checkliste, welche in entsprechenden Beschreibungen, Ursachen und Folgen, Indikatoren und Methoden strukturiert ist, bietet ebenfalls eine neue übersichtliche Sichtweise, wie in Startups mit technischen Schulden umgegangen werden kann.
Einige TD-Typen, wie die ersten zehn aus dem Ranking, wurden sehr ausführlich studiert, wobei die letzten, wie sich durch die Literaturrecherche ergab, nur wenig Aufmerksamkeit bekommen haben. Hieraus resultiert, dass es für einige Typen nur sehr wenig oder gar keine konkreten TD-Management Methoden gibt. Durch die Suche nach Methoden zur Vermeidung der respektiven Indikatoren in Forschungsarbeiten, konnten jedoch Methoden zur Vermeidung dieser Typen gefunden werden.
All die, im Rahmen der Arbeit präsentierten Ergebnisse unterliegen allerdings gewissen Einschränkungen, welche im Folgenden beschrieben werden.

## 7.3 Einschränkungen der Gültigkeit der Ergebnisse

Die Checkliste sowie das erweiterte ESEM müssen, um konkrete Aussagen über deren Anwendbarkeit machen zu können, noch auf deren Verständlichkeit und Validität evaluiert werden. Durch den großen Umfang der besprochenen Themen, ist es weder die Aufgabe noch Intention der Arbeit, einen Prozess zu konstruieren der ad hoc durch Startups nutzbar ist. Wie in der Zielsetzung beschrieben, geht es darum, einen ersten Überblick über die Möglichkeiten zur Vermeidung von technischen Schulden in Startups zu schaffen.
Dies wird bewerkstelligt, indem die Checkliste und das erweiterte ESEM erstellt wurden, welche nachgewiesene Fakten getreu synthetisieren und auf etablierten Kenntnissen aus der Literatur aufbauen. Somit sind die Liste und das Modell bereits teilweise evaluiert und die Gültigkeit der Schlussfolgerungen von zusammengefassten Informationen geprüft.

Die Arbeit erhebt keinen Anspruch auf Vollständigkeit, da, wie in Abschnitt 7.2 erwähnt, es sogar für die größten etablierten Unternehmen unmöglich ist sämtliche technische Schulden zu vermeiden. Zudem variieren die Typen und Mengen technischer Schulden von einem Projekt zum anderen und sind zu facettenreich [177], sodass ein vollkommen generalisierbarer Ansatz vorerst nicht denkbar ist. Selbst das Beachten aller genannten Aspekte kann es somit nicht ermöglichen, sämtliche technischen Schulden zu vermeiden.

Das Ranking der TD-Typen baut unter anderem darauf auf, wie gut erforscht die TD-Typen in der Literatur sind. Es wird also auf dem aktuellen Stand der Forschung aufgebaut, der aber nicht der Wahrheit entsprechen muss. Wie in Abschnitt 7.2 erwähnt, kann es also vorkommen, dass gewisse TD-Typen, wie zum Beispiel Security Debt, wenig studiert wurden und einen niedrigeren Rang bekommen, als deren Konsequenzen es rechtfertigen würden.

## 7.4 Risiko VS Aufwand

Eine neue grundlegende Frage, die durch diese Arbeit entsteht, ist, wie stark Startups gegen technische Schulden vorgehen sollten. Sämtliche Schulden zu vermeiden klingt nämlich verlockend, doch ist dies, wie in den vorigen Abschnitten besprochen, nicht möglich, sodass Kompromisse eingegangen werden müssen.

### 7.4.1 Healthy VS Unhealthy Technical Debt

Vor allem bei Startups, deren Ressourcen begrenzt sind, ist das komplette Vermeiden von technischen Schulden oft unmöglich, sodass, wie im Abschnitt 2.2.2 erklärt, Schulden oft freiwillig [29] eingegangen werden. Ktata und Levesque [61] beschreiben in diesem Sinne die Existenz von sogenannten „gesunden“ und „ungesunden“ technischen Schulden. Gesunde Schulden werden bewusst eingegangen und bestehen darin „das Einfachste zu machen, das funktioniert und der Versuchung zu widerstehen, die Zukunft vorherzusagen“. Es sollte also nicht versucht werden, alles im Voraus zu planen, da dies unmöglich ist und hierfür die nötigen Ressourcen fehlen.
Es sollte sich auf leichtgewichtige Prozesse fokussiert werden, die es dem Startup ermöglichen, mit wenig Zusatzaufwand, technische Schulden zu managen [132]. Ktata und Levesque [61] erklären auch, dass ungesunde technische Schulden entstehen, wenn Anforderungen implementiert werden, die der Qualität der Software schaden. Entweder weil zu viele Anforderungen angenommen werden, weil sie mit Absicht mangelhaft implementiert wurden oder weil sie einen direkten negativen Einfluss auf das Produkt haben. Sie erklären zudem, dass es wichtig ist, gesunde Schulden, trotz derer kurzeitigen

Vorteile, zu kontrollieren und kontinuierlich zurückzuzahlen, um deren negative Langzeiteffekte auf das Wachstum des Startups zu reduzieren. Sie sind also nur zeitweise gesund, doch nicht auf Dauer. Im Generellen gilt, wie es Yli-Huumo et al. [15] beschreiben: Aktionen, die zu technischen Schulden führen, können vorteilhaft sein, doch technische Schulden sind es nie.

### 7.4.2 Detaillierungsrad des TD-Managements

Durch die äußerst begrenzten Ressourcen von Startups muss auch beachtet werden, dass diese effizient genutzt werden und kein unnötig hoher Detaillierungsgrad beim TD-Management eingesetzt wird. Wie in Abschnitt 6.6.1 erwähnt, können zwar einige finanzielle Schulden einfach berechnet werden. Doch um die, durch alle TD-Typen verursachten Kosten, auszurechnen, benötigen sogar etablierte Unternehmen eigens darauf ausgerichtete Teams [108], wodurch dies für Startups, nicht in Frage kommt. Startups sollten sich dementsprechend, statt auf finanzielle eher auf technische Schulden fokussieren und die, für sie wichtigsten auswählen und managen. Nach der erfolgreichen Skalierung sollte sich das Startup überlegen, ein dediziertes TD-Management-Team zu erstellen [177].
Eine andere Art an technischen Schulden entsteht ebenfalls, wenn zu viel Aufwand in das Vorbeugen von TD gesteckt wird, statt zu wenig. Kruchten et al. [62] beschreiben in diesem Sinne das sogenannte „Gold Plating“, bei dem das Startup zusätzliche Features liefert, die durch die Stakeholder nicht explizit gewünscht waren. Das kann daraufhin dazu führen, dass diese Features nie benötigt werden und somit wertvolle Arbeitszeit, verloren gegangen ist. Das Ziel ist es also, die Anforderungen zu befolgen und weder zu viel zu versprechen und nicht genug zu liefern noch zu wenig zu versprechen und mehr als erwartet zu liefern, da beide Varianten, ihre eigenen Risiken bergen.

### 7.4.3 Kontextabhängigkeit

Wie es et Greening [80] beschreiben, hängt die Tatsache, ob Startups einen höheren Wert auf TD-Management setzen müssen auch davon ab, was für ein Produkt sie entwickeln. Handelt es sich um ein einzigartiges Produkt, das in ähnlicher Form noch nicht existiert, kann anfangs „Geschwindigkeit auf Kredit“ gekauft werden. Da wenig Konkurrenz vorhanden ist, kann das Produkt erstmal schnell entwickelt werden und die technischen Schulden erst später zurückgezahlt werden. Handelt es sich um einen bereits bewährten Markt, muss TD-Management sofort betrieben werden, da ansonsten besser organisierte Konkurrenten durch entsprechend mehr Skalierbarkeit, schnell die Überhand gewinnen werden. Sehr oft spielt die Qualität einer Produktidee nur eine Nebenrolle, denn wenn die Umsetzung mangelhaft ist, auch dem innovativsten Startup, die Ressourcen ausgehen werden. Ist ein Produkt also zum Beispiel UI-lastig, muss ein hoher Wert auf das Interface gelegt werden, um einen guten ersten Eindruck zu ermöglichen.
In Zukunft könnte dies, durch den Einsatz neuer Methoden und das Beachten neuer TD-Typen, noch einfacher geschehen.

## 7.5 Mögliche Erweiterungen

Die, in der Checkliste und dem ESEM, besprochenen Methoden entsprechen dem aktuellen Stand der Forschung bezüglich technischer Schulden. Doch es werden zahlreiche Konzepte entwickelt, die zwar noch nicht einsetzbar sind, aber Ideen über die Zukunft des TD-Managements verleihen können.

### 7.5.1 Entstehung weiterer TD-Typen

Die TD-Typen der Checkliste, wurden zum Beispiel entsprechend der, im Abschnitt 4.2.3, erklärten Methodik ausgewählt. Doch wie es Rios et al. [9] beschreiben, gibt es bereits viele strukturierte Studien für einige TD-Typen, wobei andere noch erforscht werden müssen. Es werden also nach und nach mehr Typen entdeckt, die sich auf ein Projekt auswirken können.

Bogner et al. [52] haben sich zum Beispiel damit befasst, neue TD-Typen zu identifizieren, die im Bereich vom KI-basierten Systemen vorkommen. Darunter befinden sich *Data Debt,* die sich auf Mängel bei dem Sammeln, Management und der Nutzung von Daten bezieht, *Model Debt,* die auf ein suboptimales Design und Training der Modelle zurückzuführen ist, *Configuration Debt,* die zum Beispiel durch eine mangelhafte Dokumentation der Konfigurationsdateien entsteht und *Ethics Debt,* die sich auf einen Mangel an Fairness und Transparenz der eingesetzten Algorithmen bezieht.

Vor allem der letztere TD-Typ gewinnt in den vergangenen Jahren, durch den oft unkontrollierten Aufmarsch von künstlicher Intelligenz, immer mehr an Wichtigkeit. Ein anderer TD-Typ, der durch Couto et al. [178] studiert wurde, ist *Energy Debt*. Dieser bezeichnet die Energie, die durch ein System verloren geht, wenn darin Code Smells wie zum Beispiel toter Code, zu viele Schleifen oder Rekursion vorhanden sind. Vor allem im Kontext des Klimawandels, ist dieser TD-Typ somit äußerst relevant, da viele kleinere und größere Systeme, die solche Smells beinhalten, zu enormen Energieverschwendungen führen können.

### 7.5.2 Das ideale TD-Management Tool

Das finale Ziel des TD-Managements wäre, über ein einziges Toolkit zu verfügen, das in der Lage ist, alle 16 TD-Typen aus der Checkliste, sowie die neu entdeckten, zu messen und deren Management zu ermöglichen. Es gibt zwar bereits einige TD-Management Tools, doch wie es Rios et al. [9] beschreiben, wäre es ideal, dass auch alle TD-Management Aktivitäten darin integriert werden, um zu verhindern, dass für verschiedene Aktivitäten unterschiedliche Tools eingesetzt werden müssen.

Da das ultimative Ziel des TD-Managements ist, wie es Snipes et al. [103] beschreiben, Entscheidungen zu erleichtern, müssten alle TD-Typen verglichen werden können. Um dies zu ermöglichen, sollte das Tool in der Lage sein, alle Typen einheitlich zu messen, indem es, wie im Abschnitt 2.3.3 angeführt, für jeden ein Kapital und Zinsen berechnen könnte. Ebenfalls könnte solch ein Tool, wie es Shull et al. [31] beschreiben, in der Lage sein, unterschiedliche Prognostizierungen und Simulationen durchzuführen. Manager könnten nämlich verschiedene TD-Management Strategien eingeben und das Tool könnte darstellen, wie sich gewisse Schulden und TD-Typen dadurch im Laufe der Zeit entwickeln könnten.

Durch Technologien, wie künstliche Intelligenz und Machine Learning, könnte diesem Ziel nähergekommen werden, doch es handelt sich um ein sehr kompliziertes und umfangreiches Unterfangen, weshalb bis jetzt nur Ansätze dafür existieren.

Letztendlich würde die Existenz umfassender Tools, wie es Avgeriou et al. [12] darlegen, mit sich ziehen, dass TD-bewusste Entwicklung eine akzeptierte Standartmethode wäre, Software zu produzieren. Da Entwickler und Manager über sämtliche Schulden informiert werden würden, würde ein Großteil der Schulden ebenfalls nur noch absichtlich eingegangen werden.

# 8. Zusammenfassung und Ausblick

## 8.1 Zusammenfassung

Startups sind beliebter denn je, sodass diese, vor allem im Sektor der Softwareentwicklung, immer zahlreicher werden. Sehr oft scheitern sie allerdings, da sie technische Schulden ansammeln, weil sie deren Folgen unterschätzen und nicht wissen, wie sie diese vermeiden können. In der Literatur sind Informationen zum Management von technischen Schulden zwar vorhanden, doch diese sind entweder stark verteilt oder beziehen sich nicht auf die Entwicklung in Startups. Diesem Mangel an Überblick fügt sich hinzu, dass viele Typen an technischen Schulden existieren, die an unterschiedlichen Stellen des Entwicklungszyklus auftreten, wodurch das Vermeiden der Schulden zusätzlich erschwert wird. Mit dem Ziel einen Lösungsansatz für diese Probleme zu finden, wurde sich dementsprechend gefragt, *„Wie können unterschiedliche Typen an technischen Schulden in Startups in verschiedenen Phasen der Softwareentwicklung vermieden werden?"*.

Um diese Frage und die daraus resultierenden Teilfragen zu beantworten, wurde eine strukturierte Übersicht des TD-Managements in Form einer Checkliste erstellt. Hierfür wurde bewährte Literatur zum Thema technische Schulden ausgewählt und Informationen so synthetisiert, dass sie Startups einen schnellen Überblick über die sehr vielfältigen Schulden geben. Die hierdurch erhaltene Checkliste priorisiert, anhand mehrerer Faktoren, die unterschiedlichen TD-Typen, die in Unternehmen vorkommen können und ordnet sie ihren respektiven Entwicklungsphasen zu. Jeder TD-Typ wird darin erklärt, die wichtigsten Ursachen und Folgen genannt, sowie beispielhafte Indikatoren beschrieben. Dies ermöglicht es Startups selber zu erkennen, durch welche technischen Schulden sie betroffen sind und welche sie am ehesten vermeiden sollten. Ebenfalls werden unterschiedliche Arten an Methoden beschrieben, die es ermöglichen, die jeweiligen TD-Typen zu vermeiden, indem verschiedene TD-Management-Ansätze umgesetzt werden. Ist ein Startup an einigen TD-Typen besonders interessiert, kann es sich ebenfalls, dank der beigefügten, detaillierteren Version der Checkliste, präziser über deren Ursachen und Folgen sowie Management-Methoden erkundigen.

Anhand dieser Kenntnisse wurde ein Framework erweitert, das beschreibt, wie Startups bei der Entwicklung ihres ersten Produktes vorgehen sollten. Hierfür wurden Schwachstellen im Modell erkannt und entsprechend der Checkliste und zusätzlicher Literatur ausgebessert und vervollständigt. Das erweiterte Modell integriert nun sämtliche TD-Management-Aktivitäten und Methoden zur Vermeidung von Teilen aller beschriebenen TD-Typen, sodass möglichst viele Risiken dadurch abgedeckt sind. Ebenfalls wurde es möglichst leichtgewichtig und komplementär zur Checkliste erstellt, sodass beide Ressourcen als eine gemeinsame Informationsquelle dienen.

Das erweiterte ESEM und die Checkliste bieten nicht nur einen Ansatz, um TD-Management bei der Entwicklung durchzuführen, sondern ermöglichen es Entwicklern ebenfalls sich zu rechtfertigen, sollten im Projekt Probleme entstehen. Somit kann sich mit dem Management und Stakeholdern geeinigt werden, welche Ziele realistisch sind und ein qualitativ hochwertiges erstes Produkt entwickelt werden, das für Startups überlebenswichtig ist.

Um sich zu etablieren, sind Startups dazu verpflichtet, das Produkt schnell auf den Markt zu bringen. Dank der Checkliste und des erweiterten Frameworks, muss dies allerdings nicht mehr auf Kosten der Qualität geschehen, da Startups nun wissen, wie und welche Schulden sie eventuell eingehen und wie und weshalb sie diese managen sollten.

## 8.2 Ausblick

Um die Ergebnisse auf deren Anwendbarkeit in Startups zu prüfen, müssen diese in einem nächsten Schritt auf deren Verständlichkeit und Validität evaluiert werden. Sämtliche Erkenntnisse basieren zwar auf bewährter Literatur, doch es fehlt eine praktische Anwendung im realen Startup-Umfeld, um diese zu bestätigen und zu verbessern. Aus diesem Grund wird im Folgenden ein mögliches, weiteres Vorgehen erläutert.

Als erstes muss eine Interviewstudie mit betroffenen Praktikern durchgeführt werden, wie zum Beispiel Startup-Gründern, Projektmanagern, Softwareentwicklern. Diesen wird dann die zusammengefasste Checkliste und das Framework präsentiert und erklärt. Im Anschluss erforscht der Einsatz eines Fragebogens, ob die Teilnehmer die Liste und den Prozess verstehen und inwiefern sie diese hilfreich und korrekt finden. Nach der Auswertung der Daten können dann entsprechend erste Verbesserungen vorgenommen werden.

Auch die Durchführung von Interviews an gescheiterten Startups bietet aufgrund der Analyse von Fehlern einen möglichen Erkenntnisgewinn. Die Befragung dieser Startups kann zu der Beantwortung der Frage führen, ob das Einhalten von Aspekten der Checkliste und des Frameworks ein frühzeitiges Ende vermieden hätte.

Letztendlich wäre es ideal ein oder mehrere Startups zu finden, die sich bereit erklären, die Checkliste und das Framework einzusetzen. Doch es wird schwerfallen, Startups mit limitierten Ressourcen davon zu überzeugen, neue Maßnahmen und Prozesse einzusetzen. Aus diesem Grund müssen die niedrigen Erfolgschancen von Startups und die Gründe deren Scheiterns aus der Checkliste vorgestellt werden, sodass der Mehrwert von TD-Management erkannt wird. Alle zuvor beschriebenen Evaluierungsschritte der Checkliste und des ESEM sollten ebenfalls durchgeführt worden sein.

Die Checkliste zur Durchführung von TD-Management und das erweiterte Entrepreneurial Software Engineering Modell bieten somit die Grundlage für zukünftige Recherchen, mit dem Ziel, das Vermeiden von technischen Schulden in Startups, zu systematisieren.

# Anhang

## A. Detaillierte Ergebnisse zu den technischen Schulden

### *A.1.Design Debt*

#### Beschreibung

Beim TD-Typen, der insgesamt am wichtigsten bewertet wurde, handelt es sich um die sogenannte Design Debt. Design Debt bezeichnet Schulden, die durch die Analyse des Quellcodes entdeckt werden können, indem Merkmale im Code erkannt werden, die gegen die Grundsätze von gutem objektorientiertem Programmieren verstoßen [30] [64]. Diese Art an technischen Schulden erscheinen während der Implementierungsphase der Softwareentwicklung.

#### Ursachen und Folgen

Um Ursachen für technische Schulden und die Probleme, die dadurch entstehen können, so übersichtlich wie möglich darzustellen, werden die probabilistischen Ursache-Wirkungs-Diagramme von Rios et al. [47] eingesetzt. Das entsprechende Diagramm für Design-Debt, in einem kleinen Unternehmen, ist in Abbildung 22 dargestellt.

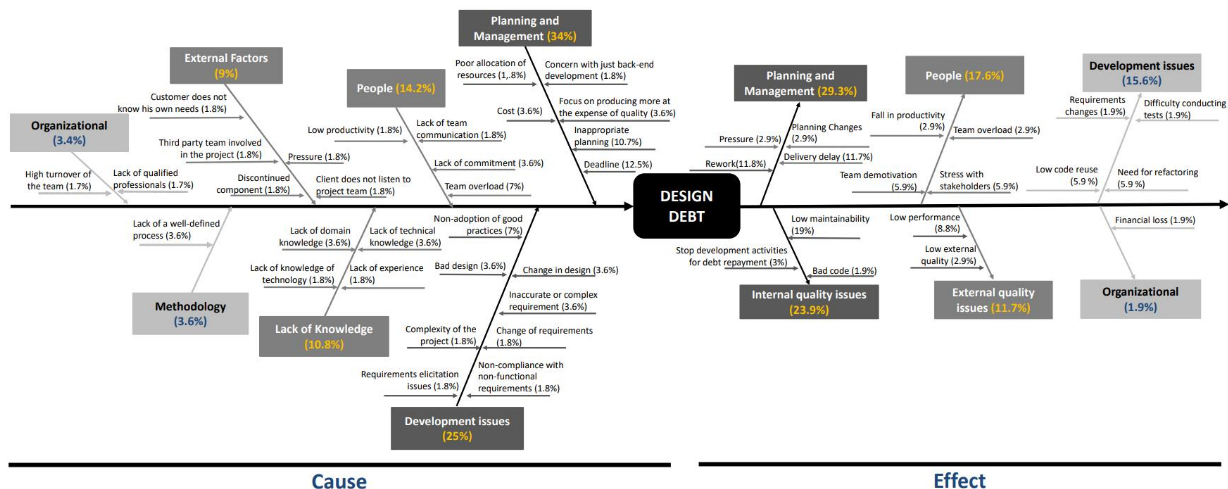


*Abbildung 22: Probabilistisches Ursache-Wirkungs-Diagramm für Design Debt [47]*

Wie durch Rios et al. [47] dank der InsighTD Studie beschrieben, handelt es sich bei den häufigsten Ursachen von Design Debt zuerst um eine zu enge Deadline und eine unangemessene Planung. Die Startups setzen also Termine, die es verhindern, ein möglichst schuldenfreies Produkt zu liefern. Zudem sind sie generell unorganisiert, was die Effizienz zusätzlich verringert. Dem fügen sich die Nichtübernahme von bewährten Praktiken und ein überlastetes Team hinzu, was beweist, dass Startups zum Beispiel Dokumentation, Richtlinien und bewährte Prozesse nicht einsetzen, sodass das Team dem Druck nicht nachkommt. Bei den wichtigsten Folgen handelt es sich um eine niedrige Wartbarkeit und Überarbeitungen. Das bedeutet, dass das Projekt durch hohe Design Debt unübersichtlich wird und oft Refactoring und andere Korrekturen benötigt.

### Indikatoren

Es gibt zahlreiche Indikatoren, die auf die Präsenz von Design Debt zurückführen. Davon sind einige:

- **Gottklassen**: Klassen, die zu viel Funktionalität beinhalten und dadurch unübersichtlich werden [30].
- **Code-Smells**: Bei Code Smells kann es sich um zahlreiche Probleme handeln, wie zum Beispiel zu stark verschachtelter oder unnötiger Code [65].
- **Grime**: Bezeichnet Code, der nicht dem verwendeten Design Pattern entspricht [66].
- **Komplexe Methoden**: Methoden, die unstrukturiert geschrieben wurden oder zu umfangreich sind [9].
- **Data Clumps**: Lange Parameterlisten, die immer wieder in verschiedenen Methoden im System auftauchen [67].

Im Falle von Design Debt wird sich damit befasst, Gottklassen zu vermeiden, da diese oft und schnell zu einem wiederkehrenden und zeitraubenden Problem werden können [68].

### Erste Methode: Ursachen- und Effekt-Analysemeetings

Die, im Abschnitt 4.2.3, beschriebenen Ursache-Wirkungs-Diagramme ermöglichen es, wie es Rios et al. [47] erklären, sogenannte Ursachen- und Effekt-Analysemeetings durchzuführen. Diese Meetings sollen dazu dienen, im Softwareprojekt vorhandene TD-Items zu besprechen und deren Ursachen und Effekte zu identifizieren. Um dies zu bewerkstelligen, kann das, in Abbildung 23, dargestellte Vorgehen befolgt werden: Drei bis fünf Teammitglieder, die mit technischen Schulden zu tun haben, sortieren die TD-Items, die ihnen Probleme bereiten, anhand der hier präsentierten Checkliste, je nach ihrem entsprechenden TD-Typen. Anschließend verwenden sie das Ursache-Wirkungs-Diagramm des entsprechenden TD-Typen, um damit die Ursachen und Folgen zu erkennen, die zu ihren TD-Items geführt haben. Die Diagramme können dann durch die Teammitglieder entsprechend für das Startup angepasst werden.

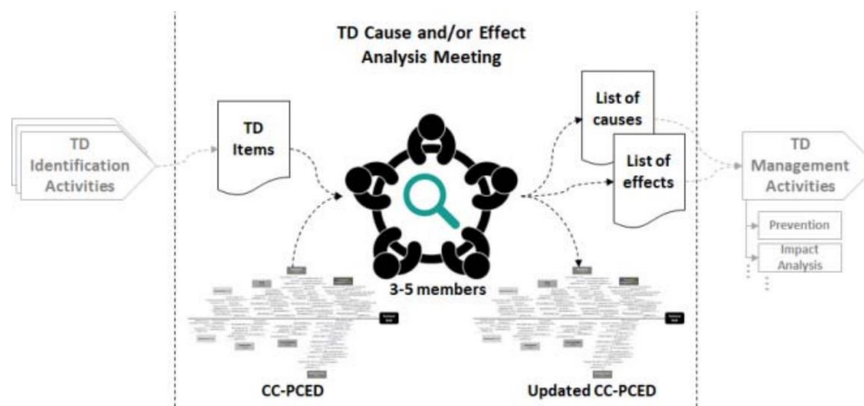


*Abbildung 23: Durchführung von Ursachen- und Effekt-Analysemeetings [47]*

Die, durch die Meetings, erhaltenen Listen von Ursachen und Effekten und das geupdatete Diagramm können im Anschluss verwendet werden, um sich für gewisse TD-Management Methoden aus der Checkliste zu entscheiden oder angemessene Tools auszuwählen. In der Studie von Rios et al. [47] behaupteten zudem 89% aller Teilnehmer, dass sie die Diagramme als hilfreich empfanden und, dass sie dadurch viel mehr und effizienter Ursachen und Effekte erkennen können.

Somit stellen die Diagramme nicht nur die Ursachen und Folgen von 14 der 16 präsentierten TD-Typen dar, sondern bieten mit den Ursachen- und Effekt-Analysemeetings eine bewehrte Methode, um diese Typen unternehmensspezifisch zu analysieren und zu managen.

#### Zweite Methode: Cost-Benefit Analyse

Detaillierter beschrieben sind Gottklassen Klassen, die viele Verantwortlichkeiten in sich bündeln, viele Objekte steuern und überwachen, sich mehr mit Daten anderer Klassen befassen als mit ihren eigenen und im Grunde die gesamte Funktionalität einer Anwendung integrieren [68]. Das ist daher gefährlich, da diese Klassen dann sehr oft geändert werden müssen, wenn irgendwas am System modifiziert werden soll. Gottklassen sind allerdings äußerst unübersichtlich, dies kann zu einer sehr schweren und langsamen Wartung des Systems führen.
Dementsprechend haben sich Zazworka et al. [68] eine Methode ausgedacht, die es ermöglicht, das Refactoring von gewissen Gottklassen zu priorisieren, je nachdem wie viel Aufwand (Cost) dies benötigen würde und wie hoch der erbrachte Mehrwert (Benefit) sein würde. Dies ist also ein Beispiel einer Cost-Benefit Analyse.
Um den benötigten Aufwand für das Refactoring zu ermitteln, werden drei Metriken und Grenzwerte verwendet, die heutzutage, wie im Abschnitt 5.5 besprochen wird, durch TD-Management Tools berechnet werden können. Hierbei handelt es sich um:

- den **Weighted Method Count** (WMC), der beschreibt wie viel Funktionalität in einer Klasse integriert ist und dessen obere Grenze durch Zazworka et al. [68] auf 46 gesetzt wurde,
- die **Access To Foreign Data Metric** (ATFD), die beschreibt, wie viel Daten aus anderen Klassen durch die Klasse verwendet werden und deren obere Grenze auf 5 gesetzt wurde,
- die **Tight Class Cohesion** (TCC), die darstellt wie hoch die interne Kohäsion der Klasse ist, also ob sie viele Verantwortlichkeiten hat und dessen Mindestwert auf 0.33 gesetzt wurde.

Je weiter sich die jeweiligen Werte der Metriken von Klassen zu diesen Grenzen befinden, desto weniger Aufwand wäre also nötig, um sie zu refactoren und es können entsprechende Ranks für die analysierten Klassen berechnet werden.
Der zweite Aspekt, der nun beachtet werden muss, ist wie viel Benefit es bringen würde, die Klassen zu refactoren. Hierfür wird zuerst der sogenannte Change Likelihood berechnet, also wie wahrscheinlich es ist, dass die Klasse geändert wird, wenn eine Änderung am System vorgenommen wird. Zum Beispiel würde ein Wert von 0.1 bedeuten, dass für jede zehnte Änderung die Klasse einmal modifiziert werden musste. Letztendlich kann der Defect Likelihood berechnet werden, der darstellt, wie oft die Gottklasse zu Problemen im System geführt hat. Ein Defect Likelihood von 0.5 würde zum Beispiel bedeuten, dass sich jedes zweite Problem in der betroffenen Klasse befand.
Sobald der Aufwand und der Mehrwert berechnet wurden, können die Klassen in eine sogenannte Cost-Benefit Matrix platziert werden, die durch ein kurzes Beispiel erklärt wird.
Um die Cost Benefit Analyse von Gottklassen zu prüfen, haben Zazworka et al. [68] diese nämlich in einem kleinem Unternehmen zur Probe gestellt. Durch die Berechnung der zuvor genannten Metriken für die Gottklassen im Projekt, ergab sich die, in Abbildung 24, dargestellte Cost Benefit Matrix.

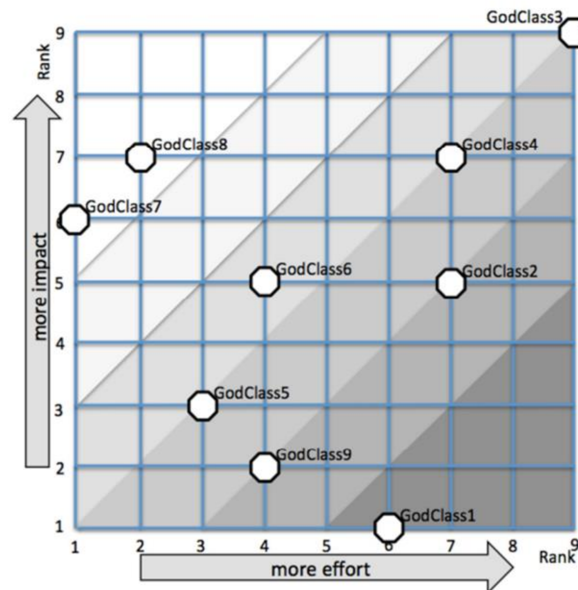


*Abbildung 24: Design Debt Cost Benefit Matrix für Gottklassen in einem Projekt [68]*

Klassen mit einem höheren Refactoring Aufwand (Effort oder Cost) werden in der Matrix mit einem höheren Effort Rank dargestellt und Klassen deren Refactoring einen höheren Mehrwert (Impact oder Benefit) erbringen würde, besitzen einen größeren Impact Rank. Um zu entscheiden, welche Klassen nun am ehesten umgeschrieben werden sollten, wird geprüft, bei welchen dies möglichst einfach geschehen könnte und den höchsten Mehrwert erbringen würde. Gottklasse 1 wäre zum Beispiel ein schlechter Kandidat, da deren Refactoring einen relativ hohen Aufwand benötigt und die Klasse nur wenig Probleme bereitet. Gottklasse 7 oder 8 hingegen sind hervorragende Kandidaten, da sie wenig Aufwand benötigen würden und hohe negative Auswirkungen auf das System haben. Somit kann durch die Cost Benefit Analyse eine Priorisierung der Gottklassen vorgenommen werden.
Der Design Debt kann sich noch eine andere Art an technischen Schulden hinzufügen, die sich eher auf den Code der Klassen selbst bezieht.

## *A.2.Code Debt*

### Beschreibung

Bei den zweitwichtigsten TD-Typen handelt es sich um Code Debt, weshalb dieser mit erhöhter Priorität behandelt werden sollte. Code Debt bezieht sich auf Probleme die im Quellcode wiederzufinden sind und dessen Lesbarkeit negativ beeinflussen, wodurch die Wartbarkeit des Codes erschwert wird [9]. Code Debt erscheint während der Implementierungsphase der Softwareentwicklung. Oft kann diese Art von technischen Schulden identifiziert werden, indem im Code nach schlechten Coding-Praktiken gesucht wird [30] [69].

### Ursachen und Folgen

Um die häufigsten Ursachen und Folgen von Code Debt zu erkennen, wird, wie bei Design Debt, das durch Rios et al. [47] erstelle Ursache-Wirkungs-Diagramm eingesetzt, welches in Abbildung 25 dargestellt ist. Durch das Diagramm ist zu erkennen, dass die wichtigsten Ursachen eine unangebrachte *Deadline* sind, sowie eine *ungenaue Zeiteinschätzung*, ein *Mangel an Erfahrung*, an *Refactoring* und an *Training* und eine *unangemessene Planung* [55]. Somit kommt es also öfters vor, dass ein zu enger Zeitplan für ein Projekt gesetzt wird und, dass Code dadurch so schnell wie möglich geliefert werden soll und nur selten verbessert wird. Ebenso führt ein Team mit *zu wenig Qualifikationen* oder *Ausbildung* zu suboptimalen Code.

Die häufigsten negativen Folgen von Code Debt sind eine *geringe Wartbarkeit*, *finanzielle Verluste* und *häufige Überarbeitungen* [55]. Code Debt führt zu schwer lesbaren Code, was es somit deutlich erschwert ihn im Nachhinein zu verbessern. Da dies allerdings oft notwendig ist, müssen sowohl zeitliche als auch finanzielle Ressourcen eingesetzt werden.

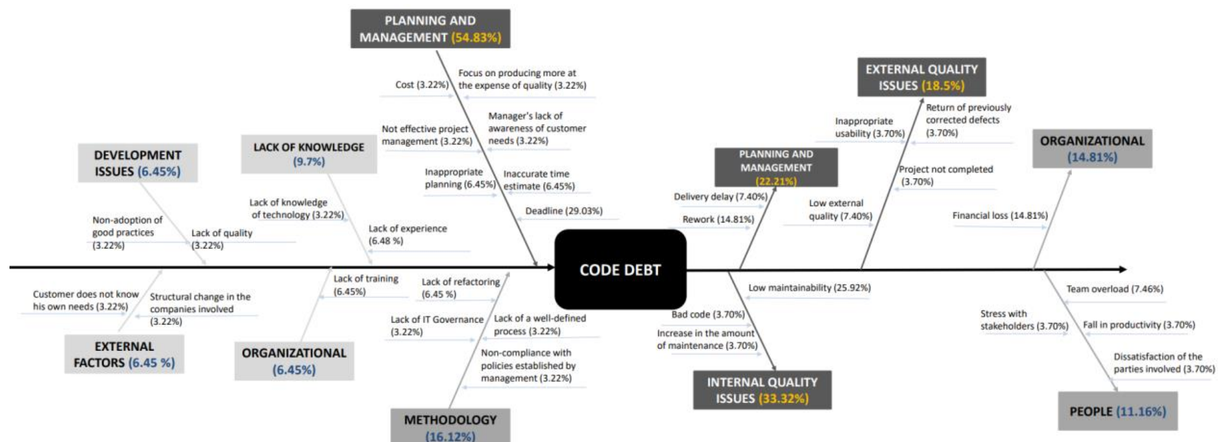

*Abbildung 25: Probabilistisches Ursache-Wirkungs-Diagramm für Code Debt [55]*

## Indikatoren

Wie auch bei Design Debt, gibt es für Code Debt zahlreiche Indikatoren, die teilweise denen von Design Debt sehr ähneln, da sie Code bezogen sind.
Davon sind einige:

- **Code-Smells**: Wie bei Design Debt, sind ein wichtiger Indikator die sogenannten Code Smells, wie zum Beispiel Methoden die zu viel oder unnötigen Code implementieren [65].
- **Duplizierter Code**: Identischer Code der sich an zwei unterschiedlichen Stellen in einem System befindet [34].
- **Suboptimaler Coding Styl**: Code der auf unübersichtliche Weise geschrieben wurde, da keine Richtlinien befolgt wurden und der somit schwer wartbar ist [9].
- **Langsame Algorithmen**: Code dessen Ausführung einen erhöhten Zeitraum benötigt, weist auf eine ineffiziente Nutzung der Ressourcen bei dessen Erstellung [70].
- **Komplexer Code**: Code der so geschrieben wurde, dass es schwierig ist nachzuvollziehen was dieser bewerkstelligt, wozu gewisse Funktionen dienen, usw.

## Methode: Code Smell Intensitäts-Index

Um mit Code Debt in einem Startup umgehen zu können, wurde durch Fontana et al. [71] eine intuitive Methode zur Detektion und Priorisierung von Code Smells entwickelt. Es handelt sich um einen Intensitäts-Index der anhand von mehreren Metriken berechnet wird und es ermöglichen soll, zu entscheiden für welchen Code Ressourcen in dessen Refactoring investiert werden sollten.
Um dies zu bewerkstelligen, wurden durch Fontana et al. [71] sechs Code Smells Arten ausgesucht die besonders oft vorkommen und schwerwiegende Folgen haben können. Dies sind:

- **Data Klassen**: Ein Data Klasse ist eine Klasse die fast ausschließlich aus Attributen und aus keinen Methoden besteht [72].
- **Brain Methoden**: Eine Brain Methode ist eine Methode die Funktionalitäten in sich integriert die besser in mehrere Methoden hätte aufgeteilt werden sollen [73].

- **Shotgun Surgery**: Bei Shotgun Surgery geht es darum, dass mehrere Klassen modifiziert werden um eine einzige Änderung durchzuführen [74].
- **Message Chains**: Message Chains bestehen darin, dass durch mehrere Klassen gegangen werden muss, um an Daten einer anderen Klasse zu gelangen [74].
- **Dispersed Couplings**: Letztendlich entsteht Dispersed Coupling, wenn eine Methode Methoden aus einer oder mehreren anderen Klassen aufruft [75].
- **Gottklassen**

Als nächstes geht es darum diese Code Smells in Projekten zu identifizieren. Hierfür wurden durch Fontana et al. [71] mehrere Metriken eingesetzt. Ein Beispiel für eine solche Metrik ist *Number Of Local Variables* (NOLV), die, wie der Name es sagt beschreibt, wie viel lokale Variablen in einer Klasse vorhanden sind. Um zu messen, ob eine Methode zum Beispiel als Code Smell anzusehen ist, werden pro Code Smell-Art Prädikate eingesetzt die Dank der Definitionen von Lanza et al. [76] erhalten wurden. Ein Beispiel eines solchen Prädikats ist im Folgenden für Data Klassen genannt:

$$\mathrm{WMCNAMM} \leq \mathrm{LOW}(14) \land \mathrm{WOC} \leq \mathrm{LOW}(0.33) \land \mathrm{NOAM} \geq \mathrm{MEAN}(4) \land \mathrm{NOPA} \geq \mathrm{MEAN}(3)$$

Eine Klasse ist somit nur eine Data Klasse, wenn die Werte der Metriken, die in den Klammern definierten Grenzwerte über- oder unterschreiten. Zusätzlich ist es Usern möglich einzustellen, ob sie Standardgrenzwerte einsetzen möchten oder „härtere“ und „weichere“. Ein Beispiel von härteren Grenzwerten wäre zum Beispiel den Grenzwert von NOLV von fünf auf acht zu erhöhen und somit würden nur Klassen mit sehr viel Lokalen Variablen als Code Smell betrachtet werden.
Als nächstes geht es darum, die Intensitäts-Indexe der jeweiligen Code Smells zu berechnen, um sie mit den Code Smells ihrer Art vergleichen zu können. Hierfür wurden durch Fontana et al. [71] fünf verschiedene Wertebereiche definiert, welche von *Very Low* [1, 3.25) bis zu *Very High* [10, 10] reichen. Die Intensitäts-Indexe von 1 bis 10 werden dann errechnet indem berechnet wird, wie weit die Werte der Code Smells sich von den Grenzwerten derer Metriken in den Prädikaten im Durchschnitt befinden. Je weiter die Werte sich von den entsprechenden erlaubten Grenzwerten befinden, also je höher die Verteilung der Werte pro Code Smell, desto höher ist der normalisierte Intensitäts-Index. Als Beispiel kann zum Beispiel das Ranking der Data Klassen genommen werden, was durch Fontana et al. [71] mit der eben beschriebenen Methode in einem von 74 Beispielssystemen durchgeführt wurde und in Abbildung 26 dargestellt ist. Die Klasse *MatchesFilePatternDecideRule* aus dem System Heritrix hat einen deutlich höheren Intensitätsindex (10.00) als die Klasse *DomainScope* (7.00), was bedeutet, dass diese viel eher inspiziert und refactored werden sollte. Zudem können sich die Werte der einzelnen Metriken und deren Grenzwerte angeschaut werden und somit, wenn nötig, die Intensitäts-Indexe nachvollzogen werden und personalisierte Entscheidungen getroffen werden.

| Class | Intensity | | ER | WOC $\leq 0.33$ | WMC' $\leq 14$ | NOPA $\geq 3$ | NOAM $\geq 4$ |
|---|---|---|---|---|---|---|---|
| **MatchesFilePatternDecideRule** | 10.00 | Very High | 13 | 0.08 | 9 | 12 | 0 |
| LowDiskPauseProcessor | 9.25 | High | 10 | 0.11 | 9 | 8 | 0 |
| RegexpLineIterator | 8.50 | High | 9 | 0.2 | 4 | 4 | 0 |
| StringIntPair | 7.75 | High | 20 | 0.2 | 1 | 0 | 4 |
| **DomainScope** | 7.00 | Mean | 6 | 0.25 | 12 | 3 | 0 |

*Legend* WMC': WMCNAMM.

*Abbildung 26: Data Klassen in einem Beispielssystem namens Heritrix [71]*

Das Erkennen von Code Smells anhand der Prädikate und Metriken sowie die Berechnung der Intensitäts-Indexe, kann zudem mittels mehrerer Tools sowie mit dem, durch Fontana et al. [71]

entwickelten Tool JCodeOdor, durchgeführt werden. Somit bietet der Intensitäts-Index eine angemessene Möglichkeit um Code Smells zu managen.

## A.3. Test Debt

### Beschreibung

In vielen Software Startups herrscht das Motto „gemacht ist besser als perfekt“, was darauf hinweist, dass dort im Generellen ein Mangel an Testing und Qualitätssicherung vorhanden ist [7]. Es fällt Startups nämlich meistens schwer die Wünsche des Kunden zu erfüllen und gleichzeitig ein hochwertiges Produkt zu liefern [7]. Aus diesem Grund entsteht die sogenannte Test Debt in der Testphase. Diese bezieht sich auf Probleme, die bei Testaktivitäten gefunden werden und die Qualität der Testaktivitäten beeinträchtigen können [9].

### Ursachen und Folgen

Auch hier wird das entsprechende Ursache-Wirkungs-Diagramm aus Abbildung 27 analysiert das es Startups ermöglicht, Ursachen- und Effekt-Analysemeetings für das Management von Test Debt durchzuführen. Das Diagramm von Rios et al. [55] stellt dar, dass die wichtigsten Ursachen eine zu enge *Deadline* und somit eine *ungenaue Zeiteinschätzung* sind. Diesen fügt sich eine *ungenaue Analyse der Auswirkungen und Risiken* hinzu, eine *zu hohe Projektkomplexität*, *schlechtes Design*, ein *Mangel an Fachwissen* im Team und letztendlich ein *zu hoher Fokus darauf mehr zu produzieren, statt die Qualität des Produkts zu beachten*. All diese Ursachen führen zu unterschiedlichen Mängeln, wie *häufigere Überarbeitungen, finanzielle Verluste* für das Startup und einer *Verzögerung bei der Auslieferung* des Produkts. Startups wissen also öfters nicht ob es sich lohnt Testing durchzuführen, da sie über wenig Zeit verfügen und sie die Auswirkungen von ungeprüftem Code auf die Zukunft des Unternehmens, stark unterschätzen.

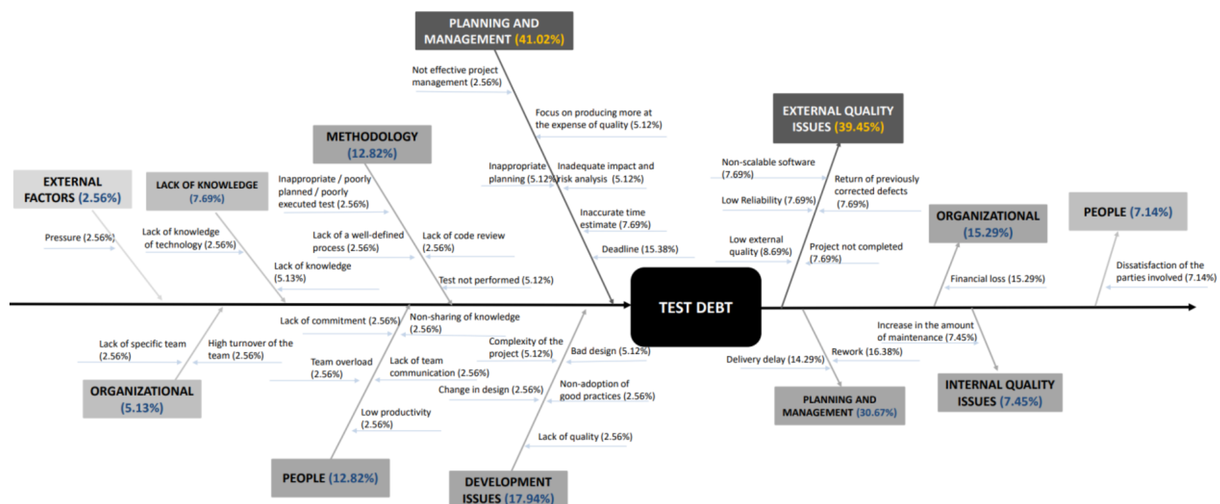


*Abbildung 27: Probabilistisches Ursache-Wirkungs-Diagramm für Test Debt [55]*

### Indikatoren

Es gibt eine Vielzahl an Indikatoren die auf Test Debt zurückführen. Davon sind einige:

- **Mangel an automatisierten Tests**: Tests werden zum Großteil manuell durchgeführt, was zu einem erhöhtem Zeit- und Ressourcenaufwand führt [34].
- **Nicht ausgeführte geplante Tests**: Entweder Tests die durch das Team geplant wurden oder bereits geschrieben wurden, aber nie ausgeführt worden sind [30].
- **Mangel an Test Case Documentation**: Es fehlt Dokumentation die beschreibt welche Artefakte während oder nach der Ausführung von Tests entstanden sind [10]

- **Bekannte Fehler im Code**: Fehler die im Code vorhanden sind aber deren Auswirkungen nie getestet wurden [30].
- **Test Smells**: Bei Tests Smells handelt es sich um Tests die schlecht geschrieben wurden und die Wartbarkeit vom Code somit negativ beeinträchtigen könnten [77].

### Erste Methode: Test Driven Development

Ein großer Fehler von vielen Startups ist, dass diese meistens sogenanntes Exploratory Testing durchführen. Hierbei handelt es sich um Testing, bei dem keine formellen Test Cases definiert werden, sondern Tester ihrer Intuition und Wissen folgen um zu entscheiden, wann Tests durchzuführen sind [78].
Um dieses Problem zu vermindern, kann als erstes Test Driven Development (TDD) eingesetzt werden. Das Ziel von TDD ist, wie es Beck [44] und Astels [79] beschreiben, sauberen Code zu erhalten der zuverlässig funktioniert. Um dies zu bewerkstelligen, werden die Tests geschrieben, bevor der eigentliche Produktionscode entwickelt wird. Hierfür wird der sogenannte Rot-Grün-Refactor Zyklus durchgegangen, der in Abbildung 28 dargestellt wird.

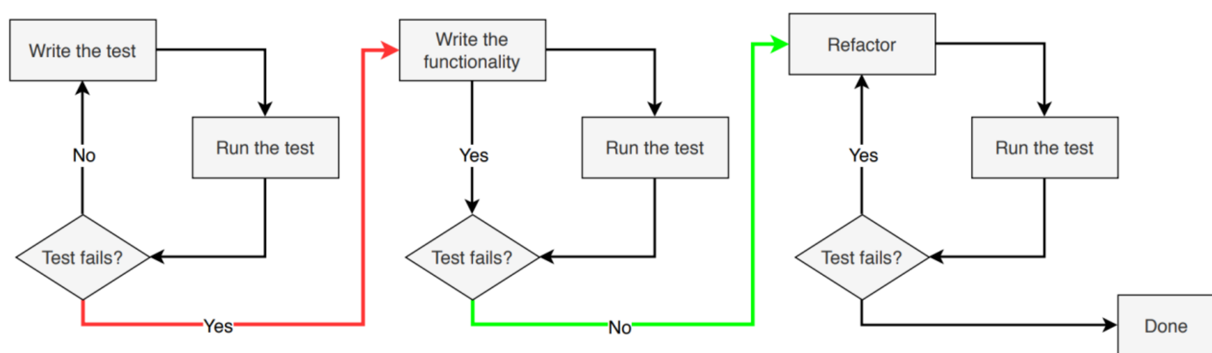


*Abbildung 28: Der Rot-Grün-Refactor Zyklus des Test Driven Developments [38]*

Als erstes geht es darum, in der roten Phase einen Unit Test zu implementieren der einem Test Case entspricht [38] [79]. Dieser Test sollte daraufhin beim Ausführen fehlschlagen, da noch kein auszuführender Code vorhanden ist. Daher die rote Farbe. Ist die nicht der Fall, wird der Unit Test neu geschrieben, bis er fehlschlägt. Anschließend wird der notwendige minimale Code geschrieben damit der Test erfolgreich ist, daher die grüne Farbe. Der Unit Test kann dann, entsprechend ausgewählter Designrichtlinien, refactored werden. Hierbei wird bei jeder Änderung automatisch erneut geprüft, dass der Test mit Erfolg durchgegangen wird, sodass Fehler nur selten vorkommen und qualitativer Code entsteht.
Durch Test Driven Development wird also, wie es Beck [44] beschreiben, der gesamte Code im Voraus auf eventuelle Probleme geprüft und der Zeitaufwand wird zusätzlich reduziert, da dieselbe Person für das Entwickeln und Schreiben der Tests zuständig ist.

### Zweite Methode: Automated Testing

Startups sind oft dazu verpflichtet Software in kurzen Zyklen zu veröffentlichen um konkurrenzfähig zu bleiben. Deshalb ist es äußerst wichtig effiziente Tests zu ermöglichen, die nach ihrer Einführung nur einen minimalen Aufwand benötigen und Test Debt minimieren. Genau hierfür ist Automated Testing gut geeignet, welches, wie der Name es verrät dazu dient, automatisch Tests für geschriebenen Code durzuführen [80]. Dafür wird zuerst definiert was getestet werden soll und mit welcher Software. Die Software führt dann eigenständig und parallel zur Entwicklung Tests durch, wie zum Beispiel auf einen gewissen Button im Interface zu drücken und die Auswirkungen zu prüfen. Wie genau Automated

Testing korrekt in einem kleinen Unternehmen eingeführt wird um technische Schulden im Anschluss zu vermeiden, wird im Abschnitt 5.3.14 genauer beschrieben.
Jede Lösung zu Test Debt stellt, wie es Shah et al. [78] beschreiben, einen Punkt in einem Kontinuum dar. Auf der einen Seite bietet sich die Möglichkeit Exploratory Testing einzusetzen und somit geringe Vorlaufkosten einzugehen, aber auch erhöhte Schulden im Nachhinein. Auf der anderen Seite kann Automated Testing eingesetzt werden, was Anfangs ressourcenintensiver ist, aber im Anschluss, Zeit und Kosten erspart. Die Projektleitung des Startups, muss also, je nach Projekt entscheiden, welche Lösung für das Projekt angemessener ist.

## *A.4.Architecture Debt*

### Beschreibung

Sehr oft werden technische Schulden nicht nur durch Code verursacht, sondern auch durch strukturelle und architektonische Entscheidungen [62]. Wenn es um die Architektur eines Systems geht, kann es allerdings oft, während der Planungs-, Design- und Implementierungsphase, zu Architecture Debt kommen [30] [62]. Dieser Schuldentyp bezieht sich auf problematische Architekturentscheidungen die sich auf die interne Qualität von Software auswirken [81].

### Ursachen und Folgen

Zu den häufigsten Ursachen von Architecture Debt, welche in Abbildung 29 dargestellt werden, gehören eine unangemessene *Deadline*, ein *Mangel an Fachwissen* und an *Wissen über die eingesetzte Technologie*, sowie eine *nicht ausreichende Infrastruktur* des Unternehmens. Zeitdruck und ein Mangel an Qualifikationen sind also ein großer Verursacher von Architecture Debt. Die wahrscheinlichsten Folgen von Architecture Debt sind eine *schlechte Performance*, *Auslieferungsverzögerungen* und eine *schlechte Wartbarkeit*. Bei dem folgendem Lösungsansatz geht es vor allem darum, letztere zu verbessern.

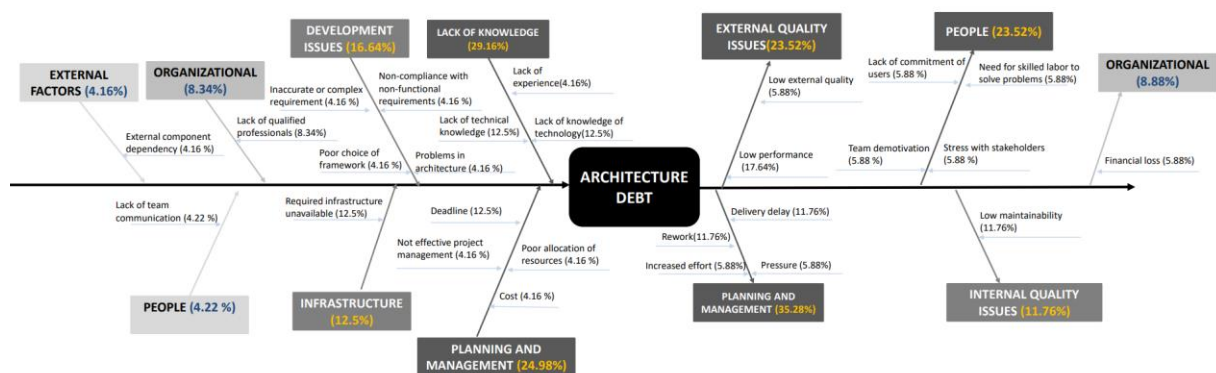

*Abbildung 29: Probabilistisches Ursache-Wirkungs-Diagramm für Architecture Debt [55]*

### Indikatoren

Ein Vielzahl an Indikatoren weist darauf hin, ob in einem System Architecture Debt vorhanden ist. Einige Beispiele sind:

- **Architectural Smells**: Hierbei handelt es sich um architektonische Entscheidungen, die die Qualität des Systems negativ beeinflussen. Wie zum Beispiel schlechte Designentscheidungen die die Modularität gefährden [82].
- **Dependency Verstöße**: Wenn zum Beispiel durch eine Klasse auf Daten einer anderen Klasse zugegriffen wird, die diese laut Design Pattern nicht verwenden sollte [83].

- **Uneinheitlichkeit von Patterns und Policies**: Wenn in einem System mehrere unterschiedliche Architektur-Patterns wie zum Beispiel Client-Server eingesetzt werden oder die eingesetzten Richtlinien nicht dem Pattern entsprechen [84].
- **Voneinander abhängige Ressourcen**: Dieses Problem erscheint, wenn zum Beispiel die Ressourcen eines Moduls von den Ressourcen eines anderen Abhängen und somit zu großen Abhängigkeiten führen [85].
- **Mangel an Mechanismen für die Behandlung nicht-funktionaler Anforderungen**: Nicht-funktionale Anforderungen werden nicht oder nur wenig behandelt, sodass zum Beispiel die Wartbarkeit des Codes vernachlässigt wird [86].

### Methode: Coupling-Between-Modules

Um Architecture Debt zu beheben, reicht es leider oft nicht aus den Code nur stellenweise zu ändern, sodass umfangreichere Entwicklungsaktivitäten notwendig sind [30]. Es sei denn, die Architecture Debt werden kontinuierlich erkannt und entfernt. Aus diesem Grund sollte Architecture Debt, wie einige andere technische Schulden, lieber bereits vor dessen Entstehung vermieden werden oder kurz danach.

Um im Laufe der Projektdurchführung zu wissen, wie viel Architecture Debt vorhanden ist, haben Tvedt et al. [87], eine semi-automatisierte Methode entwickelt, die die tatsächliche Architektur mit der geplanten vergleicht. Um deren Lösung zu prüfen, wurde sie auf einer eigenen Datenvisualisierungssoftware namens VQI getestet. Bereits bei der Entwicklung dieser Software, wurde das Mediator Design Pattern eingesetzt, das so funktioniert, dass Module nicht direkt miteinander verbunden sind, sondern über einen sogenannten Mediator. Dies reduziert die Anzahl an Verbindungen zwischen Modulen, was die Wartbarkeit des Codes somit erleichtert. Aus diesem Grund haben Tvedt et al. [87] auch entschieden, als Metrik für die Architekturqualität und die Wartbarkeit des Systems, die Anzahl an ungerichteten Verbindungen jedes Moduls zu verwenden. Diese Anzahl entspricht dem *Coupling-Between-Modules* (CBM). Ein einfaches Beispiel ist in Abbildung 30 dargestellt. Hier hätte Modul X ein CBM von 2, Y ein CBM von 1 und Z ein CBM von 1.

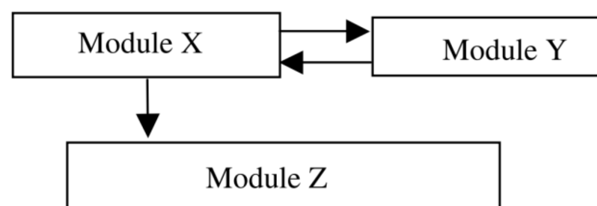


*Abbildung 30: Beispielhafte Systemarchitektur [87]*

Um diese Metrik für Module im Quellcode einfach auszurechnen, haben Tvedt et al. [87] ein Tool entwickelt, dass den Code analysiert und dann eine Liste der Verbindungen für jede Komponente ausgibt. Für jede Verbindung werden ebenfalls die involvierten Klassen, Pakete und Subsysteme genannt. Diese Liste kann dann in Excel importiert und mit der geplanten Architektur verglichen werden. Ein Beispiel eines Teils der Liste, der nur die Anzahl an Verbindungen pro Komponenten nennt, ist in Tabelle 6 zu sehen.

*Tabelle 6: CBM-Werte für geplantes und tatsächliches Design, in Anlehnung an [87]*

| Components | Planned Design | Actual Design (VQI3) |
|---|---|---|
| Cache | 1 | 2 |
| Mediator | 10 | 8 |

Hier sind zwei Typen a Verstößen deutlich zu erkennen: Verbindungen im tatsächlichen Design die nicht geplant waren und Verbindungen im geplanten Design die nicht im tatsächlichen auftauchen. Um ebenfalls zu bestätigen, dass das Tool wirksamer ist als ein Entwickler, wurde ein Entwickler gefragt, Verstöße im VQI3 zu erkennen und im Anschluss wurde das Tool eingesetzt. Der Entwickler hatte 7 Verstöße entdeckt und das Tool 15.

Um die Ergebnisse des Tools auszuwerten kann folgend vorgegangen werden [87]:

1. Eine geplante Architektur für das System definieren. Zum Beispiel durch ein Whiteboard Gespräch unter zuständigen Teammitgliedern.
2. Zu einem gewünschten Zeitpunkt, die Architektur des Systems, mittels des Tools, mit der geplanten vergleichen.
3. Verbindungen analysieren und auf Verstöße prüfen und nachschauen, ob diese Verbindungen gegen zusätzliche betriebseigene Designkriterien verstoßen.
4. Code anpassen und den Prozess wiederholen um sicherzustellen, dass die Architektur in Zukunft einfach wartbar sein wird.

Somit bietet der Ansatz von Tvedt et al. [87] eine einfache, Tool-unterstützte Methode für Startups, um ihre Architecture Debt zu managen und die Wartbarkeit zu erhöhen.

## A.5. Requirements Debt

### Beschreibung

Einer der wichtigsten Typen an technischen Schulden ist die Requirements Debt, da sie den Grundstein für zahlreiche Entscheidungen in einem Projekt legt. Wie es der Name verrät, entsteht Requirements Debt meistens in der Planungsphase und bezieht sich auf Kompromisse welche eigegangen werden, wenn es darum geht zu entscheiden, welche Anforderung und wie diese Anforderungen implementiert werden sollten [30] [62]. Es handelt sich also um den Abstand zwischen der optimalen Anforderungsspezifikation und der tatsächlichen Systemimplementierung [9].

### Ursachen und Folgen

Wie bei den vorigen TD-Typen, ist auch in Abbildung 31 zu sehen, dass eine der häufigsten Ursachen für Requirements Debt, eine zu enge *Deadline* ist. Dieser fügt sich ein *Mangel an Fachwissen* und *ungenaue oder komplexe Anforderungen* hinzu. Kleinen Unternehmen wie Startups fehlt es an Zeit und an für das Team verständliche Anforderungen. Dies führt dann vor allem zu *häufigen Planänderungen durch Verzögerungen* und einer *geringen Wartbarkeit* des Produkts, sodass diese Schulden bei den Anforderungen, unbedingt beglichen werden müssen.

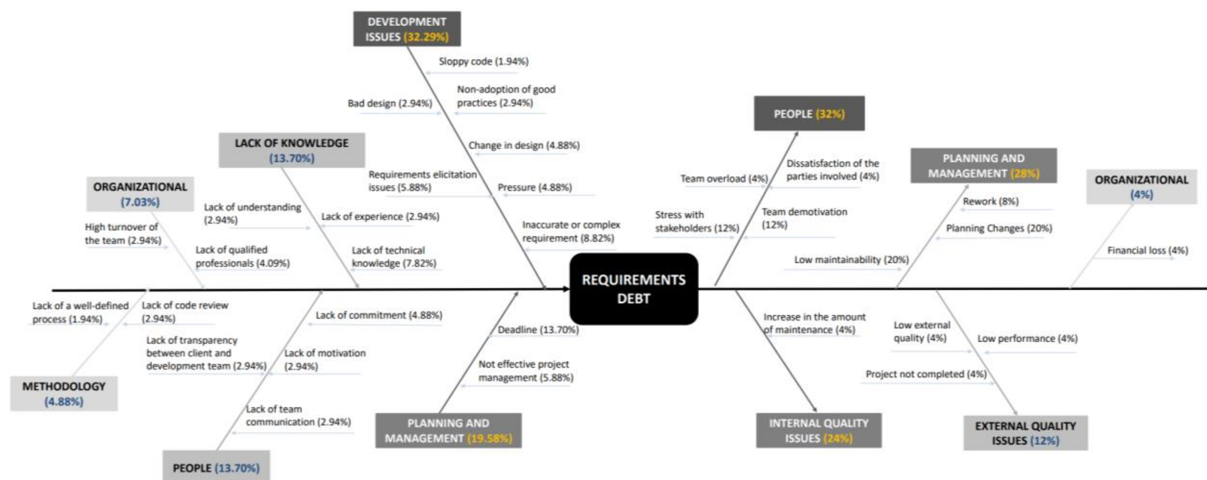


*Abbildung 31: Probabilistisches Ursache-Wirkungs-Diagramm für Requirements Debt [55]*

### Indikatoren

Auch für Requirements Debt gibt es mehrere Indikatoren die auf dessen Präsenz in einem Projekt hinweisen. Davon sind einige:

- **Requirements Smells**: Bei Requirement Smells handelt es sich um unklare Anforderungen die unter anderem durch den Einsatz von subjektiven und vagen Begriffen entstehen [88].
- **Veränderliche Anforderungen**: Die Anforderungen sind nicht ab dem Anfang klar definiert, sondern ändern sich im Laufe der Zeit, indem sie zum Beispiel erst später klarer werden [89].
- **Unnötige Merkmale**: Beim sogenannten Gold-plating oder Over-engineering, sind bereits Funktionalitäten entwickelt worden die nicht verlangt waren, da die Anforderungen ungenau sind und falsch interpretiert wurden [89] [90].
- **Anforderungs-Backlog Liste**: Es besteht während und nach der Entwicklung des Systems, eine lange Liste an unerfüllten Anforderungen [10].

### Erste Methode: Vermeiden der Requirements Debt Typen

Wie es Rindell et al. [81] beschreiben, ist das Management von Requirements Debt am schwersten zu automatisieren, da es einer der TD-Typen ist, der am wenigsten Technikbezogen ist. Lenarduzzi et al. [88] beschreiben daher drei Ansätze für kleine Unternehmen, um mit Requirements Debt umzugehen. Jeder Ansatz betrifft einen durch die Forscher definierten Requirements Debt Typen.
Beim ersten handelt es sich um *Unerfüllte Kundenwünsche* (Typ 0). Dieser entsteht, wenn gewisse Wünsche von Kunden oder Stakeholdern, bezüglich des Systems, nicht beachtet wurden. Diese Schulden können implizit sein, also ungeplante Kosten verursachen oder explizit, da sie zum Beispiel wegen einer Deadline eigegangen wurden. Kundenwünsche werden, wie durch Brunner [38] beschrieben, durch Kundenfeedback geäußert, sei dies in Interviews oder bei der Durchführung von Experimenten. Um zu messen, wie hoch der Typ 0 ist, kann einfach verglichen werden, wie hoch die Anzahl an implementierten Features ist, im Vergleich zur Anzahl an Wünschen. Hierfür sind auch keine genauen Metriken nötig, aber es kann dem Team eine ungefähre Idee über das Ausmaß der Schulden verleihen.
Beim zweiten Typen handelt es sich um die zuvor genannten *Requirement Smells* (Typ 1), wovon einige mit ihrer Definition und Erkennungsstrategie in Tabelle 7 wiederzufinden sind. Diese Smells können sowohl Anforderungen in natürlicher Sprache betreffen, als auch in Darstellungen wie zum Beispiel UML. Sie stellen Verstöße gegen den ISO29148 Standard für die Qualität von Anforderungen dar [91]. Das in Abschnitt 2.3.3 definierte Schuldenkapital, kann hier grob danach gemessen werden, wie viel

Zeit und Ressourcen nötig sind um die Requirement Smells zu beheben. Die Zinsen stellen das Risiko dar, dass Funktionalitäten, wegen der unklaren Anforderungen, falsch implementiert werden.

*Tabelle 7: Beispielhafte Requirement Smells und deren Erkennungsstrategien, in Anlehnung an [91]*

| Requirement Smells | Beschreibung | Erkennungsstrategie |
|---|---|---|
| **Subjektive Sprache** | Subjektive Sprache bezieht sich auf Wörter, deren Semantik nicht objektiv definiert ist, wie z. B. benutzerfreundlich, einfach zu verwenden, kosteneffizient. | Wörterbuch |
| **Zweideutige Adverbien und Adjektive** | Mehrdeutige Adverbien und Adjektive beziehen sich auf bestimmte Adverbien und Adjektive, die von Natur aus unspezifisch sind, wie z. B. fast immer, bedeutend und minimal. | Wörterbuch |
| **Schlupflöcher** | Schlupflöcher sind Formulierungen, die ausdrücken, dass die folgende Anforderung, nur in einem bestimmten, unpräzisen definierten Umfang, erfüllt werden muss. | Wörterbuch |

Wie Code Smells, können Requirement Smells durch Refactoring der Anforderungen beseitigt werden, indem die Ziele der Anforderungen beibehalten und die Requirement Smells entfernt werden.
Beim letzten, durch Lenarduzzi et al. [88] beschriebenen Typen an Requirements Debt, handelt es sich um eine *nicht übereinstimmende Umsetzung* (Typ 2) mit den Wünschen der Stakeholder. Das Schuldenkapital entspricht in diesem Fall den Kosten die nötig sind, um die tatsächliche Implementierung mit den eigentlichen Wünschen zu vergleichen. Die Zinsen sind die Kosten die entstehen, wenn die notwendigen Änderungen durchgeführt werden. Diese Änderungen dienen dazu, das Produkt den eigentlichen Anforderungen der Stakeholder anzupassen.

#### Zweite Methode: Vision Videos

Eine andere vielversprechenden Methode um Anforderungen zu erheben, zu klären und zu validieren, bieten die durch Karras et al. [92] beschriebenen Vision Videos. Hierbei geht es darum, dass das Startup ein initiales Meeting mit den Stakeholdern durchführt, um zu wissen, was sich diese vom Produkt erwarten. Im Anschluss kann ein einfaches Video erstellt werden, dass das Problem beschreibt welches gelöst werden muss, die entsprechende Lösung erklärt und welches Ergebnis sich dadurch für die Kunden ergibt. Wichtig ist hierbei, dass auf die Qualität des Videos und dessen Vision geachtet wird, sodass dieses technisch und inhaltlich angemessen ist und eine starke Wirkung auf dessen Zuschauer hinterlässt, indem es die Vision klar und vollständig darstellt [92].
Diese Videos sind kurz und bieten, durch die Vorführung und das Einholen von Feedback bei den Stakeholdern, einen guten Weg um herauszufinden, ob das Team deren Wünsche verstanden hat. Durch diese wichtige Kommunikation können ebenfalls Mängel gefunden, korrigiert und deren Lösungen dokumentiert werden. Dadurch können schon im Voraus, die drei zuvor beschriebenen Requirements Debt Typen, teilweise vermieden werden.

### A.6. Documentation Debt

#### Beschreibung

Ein sehr häufiger und oft unterschätzter TD-Typ, ist die sogenannte Documentation Debt, welche in allen Entwicklungsphasen entsteht [93] [94]. Bei dieser handelt es sich um Projektdokumentation die entweder unvollständig oder unangemessen ist. Dies führt dazu, dass sie zum Zeitpunkt der Entwicklung ausreichend sein mag, aber im Nachhinein Änderungen und Erweiterungen deutlich erschweren kann [30].

#### Ursachen und Folgen

Zu den häufigsten Ursachen von Documentation Debt gelten, wie in Abbildung 32 zu sehen, eine zu frühe *Deadline*, gefolgt von der Tatsache, dass *Unternehmen keinen Wert auf Dokumentation legen*. Ein *Mangel an guten Praktiken* und *Überlastung des Teams*, führen ebenfalls häufig zu Documentation

Debt. Unternehmen sind also die Auswirkungen von Documentation TD nicht bewusst und sie planen keine Zeit für das Verfassen davon ein. Zu den größten Auswirkungen von diesem TD-Typ zählen eine *niedrige Wartbarkeit*, *geringe externe Qualität* und *Stress mit den Stakeholdern*. Entwickler können also nur schwer Änderungswünschen nachkommen, was zu Spannungen und einer schlechteren Produktqualität für die Nutzer führen kann.

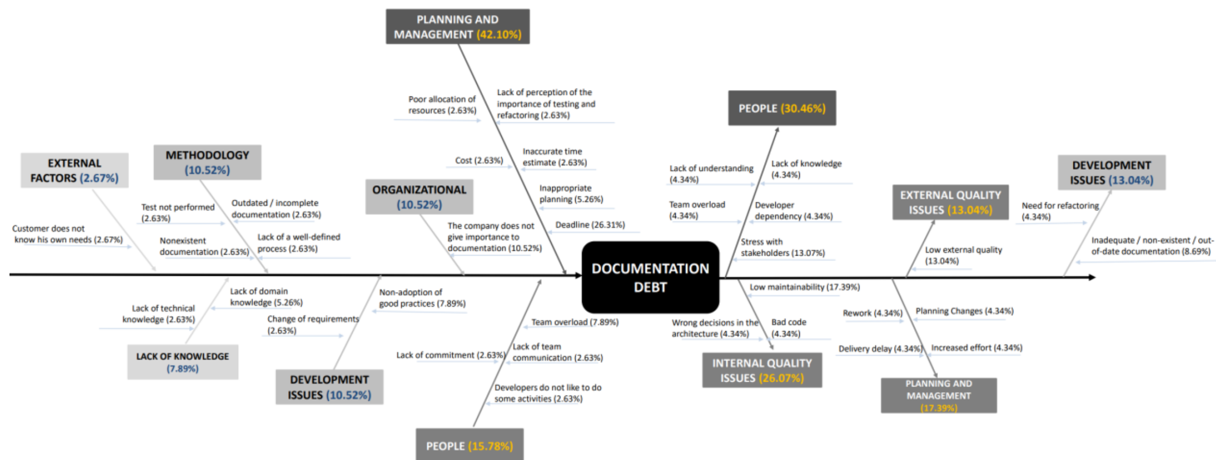


*Abbildung 32: Probabilistisches Ursache-Wirkungs-Diagramm für Documentation Debt [55]*

### Indikatoren

Es gibt eine Bandbreite an Indikatoren die auf Dokumentation Debt hinweisen. Davon sind einige:

- **Veraltete Dokumentation**: Wenn regelmäßig Änderungen am Code gemacht werden, aber Kommentare von der ersten Version des Projektes stammen und nie geändert wurden [34].
- **Keine oder Unvollständige Dokumentation**: Die Dokumentation enthält Lücken, da zum Beispiel Methoden und Klassen nicht beschrieben sind oder keine Designdokumentation vorliegt [93].
- **Subjektive oder unklare Dokumentation**: Den Lesern ist nicht klar, was genau durch den Verfasser gemeint war, sodass es zu Fehlinterpretationen kommen kann, die schwerwiegende Folgen haben können [70].

### Methode: Framework für gute Dokumentationsqualität

Wie es Treude et al. [95] beschreiben, existieren für jedes Unternehmen andere Ansichten bezüglich der Definition von guter Dokumentation. Dies liegt daran, dass Dokumentation sehr kontextabhängig ist, je nachdem um was für ein Projekt es sich handelt, auf welche Art es ausgeliefert wird, usw. Aus diesem Grund können Tools bis jetzt nur eingeschränkte Teile der Dokumentation bewerten und verbessern. Viele Entwickler scheuen sich auch davor Dokumentation zu schreiben oder zu lesen [96]. Das weist darauf hin, dass sie sie als Zeitverlust sehen, da ihnen die Richtlinien fehlen um zu wissen, wie Dokumentation kurz und klar geschrieben werden kann [95].

Aus diesem Grund haben Treude et al. [95] ein einfaches und somit für Startups ideales Qualitäts-Framework entwickelt, dass aus zehn Dimensionen besteht, die optimale Dokumentation beschreiben:

- **Qualität**: Hier geht es darum sich die Frage zu stellen, wie gut die Dokumentation, also zum Beispiel die README oder der Kommentar geschrieben ist. Sei dies durch die Rechtschreibung oder Grammatik. Dies kann heutzutage einfach durch IDEs überprüft werden. Somit kann sichergestellt werden, dass das Lesen der Dokumentation nicht anstrengend wird oder Korrekturen benötigt [97].

- **Interesse**: Wird nun zum Beispiel ein erklärendes Dokument oder ein Tutorial veröffentlicht, sollte dieses möglichst interessant gestaltet werden indem zum Beispiel Grafiken und Tabellen und kurze Sätze eingesetzt werden [97].
- **Lesbarkeit**: Es soll möglichst einfach sein auf die relevanten Informationen zuzugreifen, indem diese logisch organisiert werden. Zudem sollen die Dokumentationselemente selbst übersichtlich und leserlich dargestellt werden und stehts verfügbar sein [98].
- **Verständlichkeit**: Die Person welche die Dokumentation verfasst sollte sicherstellen, dass diese auch komplexe Sachverhalte möglichst einfach erklärt. Somit können zum Beispiel komplexe Methoden auch durch andere Entwickler erweitert werden [98].
- **Struktur**: Vor allem Dateien welche nicht viel Design enthalten, wie zum Beispiel READMEs oder Kommentare, sollten so aufgebaut sein, dass der Leser sich schnell einen Überblick darüber verschaffen kann. Absätze, Titel und Listen können hierfür hilfreich sein [99].
- **Kohäsion**: Die Dokumentation sollte so geschrieben werden, dass reichlich Pronomen und konkrete Artikel eingesetzt werden und Sätze oder Grafiken thematisch zusammenhängen. Dies führt zu einem intuitiveren Lesefluss [179].
- **Konzise**: Das Element der Dokumentation sollte keinerlei überflüssige oder irrelevante Information beinhalten, sodass das Lesen effizienter und einfacher möglich ist [180].
- **Effektivität**: Das Vokabular soll genügend technische Begriffe einsetzten, damit die gezielten Leser schnell verstehen können, welcher Sachverhalt erklärt wird. Hierbei muss allerdings darauf geachtet werden, dass das Vokabular nicht zu komplex ist, da sonst das sonst der umgekehrte Effekt entsteht.
- **Konsistenz**: Die Dokumentation, wie zum Beispiel Kommentare selber Art oder auch Grafiken, sollten ähnlich formatiert sein, damit sie einfach wiederzuerkennen und zu vergleichen sind [98].
- **Klarheit**: Nur zentrale Informationen sollen genannt und so eindeutig wie möglich beschrieben werden, damit es zu keinen Missverständnissen bezüglich Relevanz oder Bedeutung kommen kann [180]. So kann zum Beispiel die externe Qualität erhöht werden, da User zum Beispiel verstehen, wie die Software verwendet wird.

Entwicklern können diese zehn Dimensionen zum Beispiel in einer Schulung oder einem Dokument als Richtlinien vorgestellt werden. Dadurch wären sie besser dafür gewappnet, sich bei der Erstellung von Dokumentation die richtige Fragen zu stellen und Informationen zu verschriftlichen, die den Ansprüchen des Startups entsprechen. So ist es möglich, das Erscheinen der genannten Indikatoren und Folgen im Voraus zu vermindern, und die Skalierbarkeit des Produkts zu verbessern.

### *A.7. Versioning Debt*

#### Beschreibung

Bei Versioning Debt handelt es sich um Schulden, die durch Probleme bei der Softwareversionierung entstehen [9]. Diese Codebezogenen Schulden erscheinen daher eher von der Implementierung bis hin zur Wartung eines Softwareproduktes [100].

### Ursachen und Folgen

Für Versioning Debt wurden bei der InsighTD Studie, wie in Abbildung 33 zu sehen, nur wenig Ursachen und Folgen genannt, da es sich um einen relativ neuen TD-Typ handelt. Zu den häufigsten Ursachen dieser Schulden gehören, unter anderem, häufige *Überarbeitungen* der Software, *Mangel an Fachwissen* im Team und ein *Mangel an angemessenen Prozessen und Planung*. Wird das Produkt also oft geändert ohne einem organsiertem Ablauf zu folgen, kann es zu Konflikten zwischen Versionen kommen. Dies kann zu einer *verspäteten Auslieferung* führen, sowie zu einer *niedrigen externen Qualität* und vielen *Überarbeitungen*. Eine schlechte Versionsverwaltung kann also dazu führen, dass die Entwicklung immer komplexer und somit ineffizienter wird.

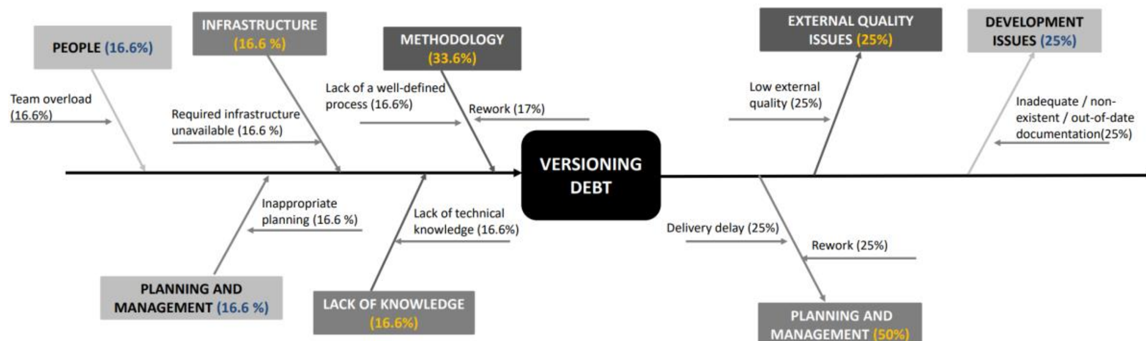


*Abbildung 33: Probabilistisches Ursache-Wirkungs-Diagramm für Versioning Debt [55]*

### Indikatoren

Mehrere Indikatoren weisen spezifisch auf die Entstehung oder Präsenz von Versioning Debt:

- **Unnötige Code Forks**: Diese entstehen, wenn Entwickler in Forks arbeiten, die überflüssig sind, da sie unterschiedliche Funktionalitäten bearbeiten als andere Entwickler. Somit wird der Code schnell verzweigt und unübersichtlich [34].
- **Merging Konflikte**: Das Abspalten von Code in einem Repository kann zwar das Entwickeln an separaten Funktionen erleichtern, aber beim Mergen kommt es dabei sehr oft zu Konflikten [80].
- **Mehrere unterstützte Produktversionen**: Wenn mehrere Versionen der selben Software durch Kunden verwendet werden, und in einer neueren Version ein Bug entdeckt wird, müssen eventuell alle vorigen Versionen ebenfalls auf diesen geprüft werden [80].

### Methode: Versioning Tool

Eine Code Fork oder auch Abspaltung besteht darin die Code Basis im Repository in mehrere Versionen zu spalten. Wie in den Indikatoren zu sehen, kann es dabei allerdings zu einer Vielzahl an Problemen kommen, die nur schwer durch einzelne Entwickler zu vermeiden sind. Aus diesem Grund existieren einige Tools, wie Git [101] oder Infox [102] die es ermöglichen, einen Überblick über die Forks zu behalten um somit unnötigen Aufwand und Codekomplexität zu vermeiden. Durch Ren et al. [100] wurde ein Tool entwickelt, das Nutzer, wie in Abbildung 34 warnt, wenn deren Pull Requests ähnlich zu anderem Pull Requests sind. Das Tool vergleicht ebenfalls aktuelle Commits von Entwicklern mit anderen Forks um den Entwickler im Voraus, wie in Abbildung 35, zu warnen.

*Abbildung 34 (links): Warnung im Falle zweier ähnlicher Pull Requests [100]*
*Abbildung 35 (rechts): Warnung im Falle von ähnlichem Code zu einer anderen Fork, Pull Request, usw. [100]*

Um dies zu bewerkstelligen, haben Ren et al. [100] fünf Merkmale identifiziert die es ermöglichen, die Inhalte einer Codeänderung zu charakterisieren und somit zwei Änderungen miteinander zu vergleichen. Für jedes Merkmal wurde eine Methode ausgewählt, um dieses messbar zu machen.

Die Merkmale sind:

- **Beschreibung der Änderung**: Commit-Nachrichten oder Titel und Beschreibungen von Pull Requests beinhalten Text der potenziell bei zwei Änderungen ähnlich ist. Es wird zuerst Tokenisierung eingesetzt um die Sätze in Wörter zu teilen, dann Stemming um die Wörter in deren Grundform zu bringen und der TF-IDF Score für die Wörter berechnet, um zu wissen, wie wichtig sie im Text sind. Im Anschluss wird die Natural Language Processing Technik Latent Semantic Indexing (LSI) eingesetzt um die Gruppen an Wörtern zu vergleichen und die Kosinus-Ähnlichkeit der durch das LSI erhaltenen Werte. Daraus entsteht ein Ähnlichkeits-Score der es ermöglicht, zum Beispiel Beschreibungen inhaltlich zu vergleichen.
- **Inhalt der Änderung**: Die Änderungen des Patches können in Git durch das Kommando „git diff" erhalten werden und können entweder Code oder auch Text sein. Diese Änderungen werden dann alle als Text angesehen und können somit mit dem selben Prozess wie für die Beschreibungen, verglichen werden. Statt LSI wird allerdings die Technik Vector Space Model eingesetzt, da sie besser genaue Übereinstimmung wie zum Beispiel für Variablennamen erkennt.
- **Geänderte Dateien**: Je mehr selbe Dateien durch zwei Änderungen modifiziert werden, desto wahrscheinlicher ist es, dass diese Änderungen ähnlich sind. Deshalb wird einfach der Jaccard-Koeffizient zwischen den zwei Dateigruppen berechnet um zu ermitteln, wie ähnlich sie sind.
- **Position der Änderungen**: Diese entspricht den Blöcken an Linien die in Dateien geändert wurden. Wird eine selbe Datei in zwei Änderungen von Entwicklern modifiziert, wird gemessen, wie groß jeweils die Überschneidung der Blöcke ist. Wenn zum Beispiel in Datei X in Änderung 1, Linien 23-45 geändert werden und in Änderung 2 Linien 23-48, dann ist es sehr wahrscheinlich, dass beide Entwickler die selbe Funktionalität bearbeitet haben.
- **Referenzen zu Issue Trackern**: Oft werden Code-Änderungen in Issue Trackern referenziert um sie zu melden, priorisieren oder als gelöst zu markieren. Indem die API von GitHub eingesetzt wird, können die in den Beschreibungen enthaltenen Links zu diesen Issues, verglichen werden. Dies ermöglicht es sehr einfach zu erkennen, ob durch die Änderung das selbe Problem bearbeitet werden soll.

Um die Merkmale der Änderungen auszuwägen und zu vergleichen, wurde die AdaBoost Machine Learning Methode eingesetzt um ein Modell zu trainieren. Diese Modell wurde getestet und gibt einen Score aus der beschreibt, wie wahrscheinlich es ist, dass zwei Änderungen dupliziert sind. Wie gut diese Methode funktioniert, wird in der unteren Abbildung 36 dargestellt. Bei einem Wahrscheinlichkeits-Score unter dem Threshold von 0.5925 bis 0.62, werden 57 bis 83% der, als Duplikate gelabelten, Pull Requests richtig als solches gelabelt und 10 bis 22% der Duplikate in den analysierten Pull Requests, wurden als Duplikate bezeichnet.

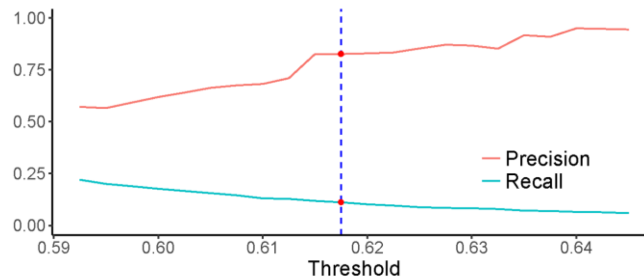


*Abbildung 36: Präzision und Recall bei verschiedenen Thresholds, wobei die gestrichelte Linie den Default Threshold darstellt. [100]*

Immer mehr unterstützenden Tools für gutes Forking werden entwickelt, sodass jedes Startup unbedingt eines in seinen Entwicklungsprozess integrieren sollte, um an Effizienz zu gewinnen.

## A.8. Defect Debt

### Beschreibung

In Software Projekten kommt es öfters vor, dass Defekte sich im Quellcode befinden, die dem Team entweder bewusst sind oder nicht [30] [103]. Die Defekte die zu Defect Debt gehören, sind all jene die dem Entwicklungsteam zum Beispiel durch Testing oder Bug Tracking bekannt sind, aber absichtlich nicht behoben wurden. Defekte erscheinen ab der Implementierungsphase und können bis in die Wartung vorgefunden werden [103].

### Ursachen und Folgen

Zu den häufigsten Ursachen von Defect Debt gehören, wie in Abbildung 37 zu sehen, ein *Mangel an Testing*, *Fachwissen* und *angemessener Planung* sowie an *Rückverfolgbarkeit von Bugs*. Defekte entstehen also, dadurch, dass die Softwarequalitätsprüfung suboptimal und Lückenhaft ist, da keine angemessenen Praktiken eingesetzt werden. Zu den häufigsten Folgen gehören eine *verspätete Auslieferung* und *unzufriedenen Kunden* durch die *schlechte externe Qualität*, was letztendlich zu *ungeduldigen Stakeholdern* führt. Defekte können also, wenn sie sich ansammeln, auf Dauer den Marktvorteil und das Image des Startups gefährden.

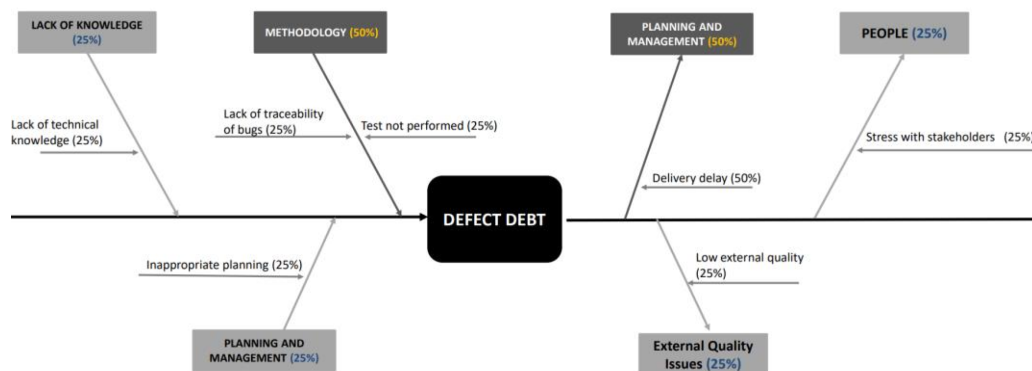


*Abbildung 37: Probabilistisches Ursache-Wirkungs-Diagramm für Defect Debt [55]*

### Indikatoren

Indikatoren, die auf Defect Debt hinweisen sind:

- **Funktionale Bugs**: Diese bedeuten, dass gewisse Funktionen sich nicht so verhalten wie gewünscht. Ein einfaches Beispiel wäre ein Login Button der den User nicht einloggt [104].
- **Logik Fehler**: Wenn Code falsch geschrieben wurde, also zum Beispiel ein Wert einer falschen Variable zugewiesen wurde, kann dies zu einem unerwünschten Verhalten der Software und Abstürzen führen [104].

- **System-Level Integrations-Bugs**: Hierbei handelt es sich um Probleme die auftreten, wenn zum Beispiel verschiedene Module nicht korrekt zusammenarbeiten. Dies kann passieren, wenn Interfaces inkompatibel sind oder die Kommunikation zwischen den Modulen lückenhaft ist [105].

#### Methode: Beachten von Kosten- und Entscheidungsfaktoren

Ähnlich zu der Beschreibung aus Abschnitt 2.3.3, existieren beim Management von Defect Debt ein Schuldenkapital und Zinsen. Das Kapital entspricht der Geldsumme die ausgegeben werden müsste um einen Defekt sofort zu beheben, wenn er einem auffällt und die Zinsen sind die Kosten die zusätzlich gezahlt werden müssten, wenn dieser Defekt erst später korrigiert wird [103]. In einigen Fällen kann es sich allerdings lohnen, das Beheben von Defekten zu verschieben, da sie zum Beispiel nur geringe Risiken darstellen.
Um dementsprechend Unternehmen dabei zu unterstützen zu entscheiden, ob Defekte sofort oder erst später gelöst werden sollten, haben Snipes et al. [103] entsprechende Kosten- und Entscheidungsfaktoren definiert.
Die Kostenfaktoren für ein direktes Lösen des Defektes, also das Bezahlen des Schuldenkapitals, sind:

- **Ermittlungskosten**: 50 bis 70% des Schuldenkapitals entstehen dadurch, dass geprüft werden muss, woher der Defekt stammt. Entwickler müssen analysieren wo der Ursprung des Defektes liegt und den Defekt eventuell replizieren sowie Lösungsmöglichkeiten definieren.
- **Änderungskosten**: 10 bis 15% des Kapitals stellen Kosten dar die dafür nötig sind, den Code entsprechend anzupassen und auf dessen Korrektheit zu inspizieren. Dem können sich auch noch die Kosten des Testings hinzufügen, um sicherzustellen, dass die Software keine weiteren Defekte enthält.
- **Validierungskosten**: 20 bis 30% des Schuldenkapitals sind die Kosten die notwendig sind um das komplette System nach der Änderung auf dessen Funktionsfähigkeit zu prüfen. Somit kann die externe Qualität des Produkts geprüft werden.

Die Kosten die entstehen, wenn der Defekt nicht sofort behoben wird, also die Zinsen, sind:

- **Workaround Kosten**: Workarounds können eingesetzt werden um einen Defekt nicht zu lösen, sondern zu umgehen. Diese Workarounds müssen dokumentiert und geprüft werden, damit Entwicklern und Kunden darüber Bescheid wissen.
- **Customer Support Kosten**: Wenn ein Nutzer auf den Defekt stoßt, muss diesem geholfen werden und ihm eventuell beigebracht werden, wie das Problem umgangen werden kann.
- **Patch Kosten**: Patches können eingesetzt werden um eine kurzfristige Lösung zu bieten, aber je nachdem in welchen System diese implementiert werden sollen, kann dies sehr zeit- und kostenintensiv werden.

In Anbetracht der zwei Kostenkategorien, müssen Startups entscheiden, ob sie das Risiko eingehen wollen einen Defekt nicht zu lösen und mit Glück das Schuldenkapital einzusparen und nur die Zinsen zu zahlen oder die sichere Route zu nehmen und das Schuldenkapital sofort zurückzuzahlen. Dies kann auf Dauer allerdings auch teuer werden. Bei dieser Entscheidung sollen die, durch Snipes et al. [103] definierten Entscheidungsfaktoren, helfen:

- **Schweregrad des Defekts**: Es muss beachtet werden, welchen Einfluss der Defekt auf die Leistungsfähigkeit des Systems haben könnte. Der Einfluss kann nämlich von einem kleinem Problem bis hin zu einem großen Hindernis reichen.

- **Existenz eines Workarounds**: Falls ein potenzieller Workaround existiert und wenig Aufwand benötigt, kann dieser in Betracht gezogen werden, bevor der Defekt komplett behoben wird.
- **Dringlichkeit des vom Kunden geforderten Fixes**: Wenn der Kunde zum Beispiel fordert, dass ein Defekt sofort behoben werden soll, muss dieser, insofern möglich, mit höchster Priorität behandelt werden.
- **Aufwand um den Fix zu implementieren**: Die Behebung des Defektes kann mehr oder weniger Zeit und Aufwand benötigen, je nachdem wie viel Ressourcen und Einschränkungen vorhanden sind.
- **Risiko des vorgeschlagenen Fixes**: Je nachdem wie umfangreich ein Fix ist, kann dieser ein gewisses Risiko mit sich ziehen, sodass geprüft werden muss, welche Funktionalität dadurch, potenziell negativ, beeinflusst werden könnten.
- **Ausmaß an benötigtem Testing**: Die betroffenen Anforderungen und Funktionalitäten bestimmen, ob neue Test Cases geschrieben werden sollten und eventuell sogar Regressions-Testing nötig ist.

Anhand der Kombination oder einer eigens erstellen Auswahl der beschriebenen Kosten- und Entscheidungsfaktoren, müssen Startups entscheiden, welche Defekte sofort oder doch erst später behoben werden sollten. Tools und mathematische Methoden um diese Aufgabe zu erleichtern sind in Entwicklung [106] und zahlreiche Defekte können längst durch IDEs selbst behoben werden. Aber durch die hohe Anzahl an Faktoren und den komplexen Kontext den das Feedback der Kunden darstellt, sind, vor allem bei größeren Defekten, kurze Teambesprechungen hilfreicher.

### *A.9.Build Debt*

#### Beschreibung

Bei Build Debt handelt es sich um Probleme die beim Bauen von Code erscheinen und somit dazu führen, dass dieser Prozess zeit- und rechenintensiver wird [30] [107]. Die Entwicklung und Software kann dadurch verlangsamt und frustrierender werden. Zu finden ist diese Art an Schulden somit von der Implementierung, bis hin zur Wartung der Software [107].

#### Ursachen und Folgen

Für Build Debt wurden durch Rios et al. [55] nur wenige Ursachen und Folgen genannt, welche in Abbildung 38 dargestellt sind. Wieder sind eine zu enge Deadline und ein Mangel an Fachwissen im Team bezüglich optimierten Code Building vorhanden, sowie ein Mangel an Infrastruktur wie zum Beispiel Software, um Build Debt zu testen und zu vermeiden. Build Debt kann zu, für den Usern sichtbaren, *Bugs und Verlangsamungen* führen und den zu implementierenden Code immer *unübersichtlicher und ineffizienter* gestalten.

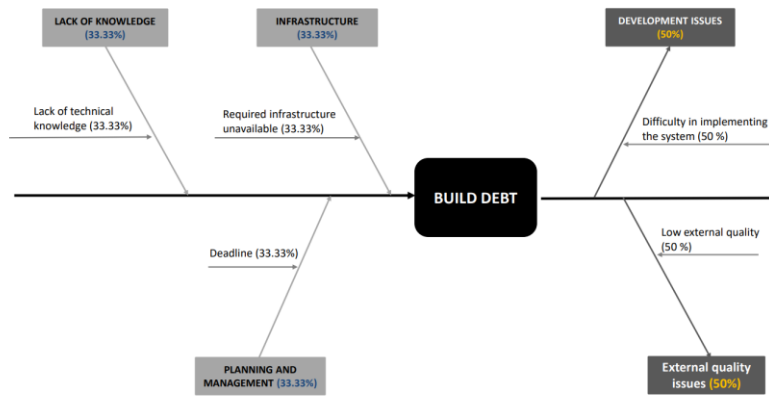


*Abbildung 38: Probabilistisches Ursache-Wirkungs-Diagramm für Build Debt [55]*

### Indikatoren

Nur einige von zahlreichen Indikatoren für Build Debt sind:

- **Under-declared Dependencies**: Bei diesen Abhängigkeiten ist ein Target transitiv abhängig von einem anderen. Das bedeutet, dass zum Beispiel die Library A von Library B abhängt, aber Library B von Library C abhängt. Somit ist A indirekt abhängig von C. Sollte Library C nun gelöscht werden, kann es bei der Library A zu Build Problemen kommen [107].
- **Underutilized Dependencies**: Eine Klasse kann zum Beispiel von einer Library abhängen die zu viel Funktionalitäten beinhaltet, welche nicht benötigt, aber trotzdem gebaut werden [107].
- **Over-declared Dependencies**: Diese bestehen, wenn in einem Target unnötige Abhängigkeiten zu anderen Targets genannt werden, die somit trotzdem gebaut werden müssen [107].
- **Fehlende Sichtbarkeitsregeln**: Targets sind nicht mit eindeutigen Sichtbarkeitsregeln wie public und private gekennzeichnet, sodass andere Targets auf deren Daten zugreifen können und ungewollte Abhängigkeiten entstehen [107].
- **Dead Flags**: Hierbei handelt es sich um Flags in der Kommandozeile die ungenutzten Code aufrufen und somit eventuelle Refactorings, deutlich erschweren [107].

### Methode: Vermeiden der Build Debt Typen

Um das Erscheinen dieser Indikatoren zu vermeiden, haben Morgenthaler et al. [107] von Google, anhand des Beispiels der Softwareentwicklung in ihrem Unternehmen, Build Debt in mehrere Untertypen geteilt. Pro Typ haben sie dann Ideen vorgeschlagen, wie diese vermieden werden können.

### Dependency Debt

Dependency Debt entsteht durch die zuvor beschriebenen Under- und Over-Declared Dependencies. Um diese Art an Abhängigkeiten zu vermeiden, haben Morgenthaler et al. [107] einen sogenannten Fixit Day vorgeschlagen. Bei diesem sollen sich die Entwickler des Unternehmens zusammenschließen um zu prüfen, in welchen ihrer Targets sich unnötige Dependencies befinden und wo direkte Dependencies fehlen. Dieser Prozess ist zwar nicht automatisiert, aber kann in einem kleinem Team wie das eines Startups deutlich einfacher durchgeführt werden, als in einem multinationalem Unternehmen wie Google. Sollte das Startup wachsen, könnte es, ähnlich zu Google, selbst ein Tool einsetzen oder entwickeln, dass Entwickler warnt sobald sie transitiv auf Code einer anderen Klassen referenzieren oder eine unnötige Referenz schreiben. Hierfür hat bei Google der Compiler dem Build System die Classpath von allen Klassen gegeben, wodurch das Build System alle Dependencies kennt und entsprechende Warnungen generieren kann.

Für Underutilized Dependencies ist die Lösung nicht ganz so einfach, da die referenzierten Targets zum Teil eingesetzt werden und dementsprechend vorerst vollkommen gebaut werden müssen. Zudem kann ein Target zahlreiche transitive Dependencies haben. Aus diesem Grund hat Google einen Dependency Refactoring Assistenten namens Clipper entwickelt. Clipper priorisiert Dependencies je nachdem wie einfach sie zu entfernen sind. Dafür verwendet es, wie in Abbildung 39 zu sehen ist, unter anderem die Tiefe der Dependency, also wie viele Sprünge gemacht werden müssen um vom Target dessen Dependency zu erreichen. Ebenfalls wird die Densität eingesetzt, welche darstellt, wie viel Pfade vom Target zur Dependency führen.

**Refactoring Candidates**

**Filter Target Names:**

| Target Name | Type | Alias | Depth | Density |
|---|---|---|---|---|
| //third_party/java/jung:jung_algorithms | java_library | yes | 2 | 1 |
| //third_party/java/jung:jung_api | java_library | yes | 2 | 1 |
| //third_party/java/jung:jung_graph_impl | java_library | yes | 2 | 1 |

*Abbildung 39: Durch Clipper erstellte Liste an Libraries welche einfache Kandidaten für Dependency Refactoring wären [107]*

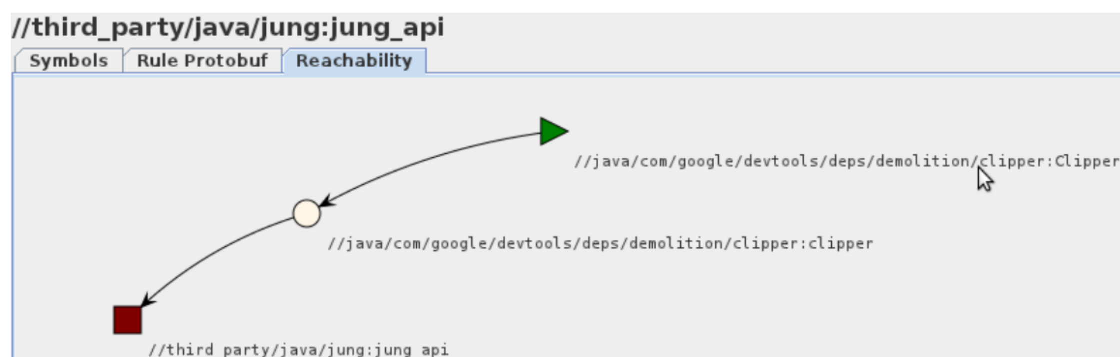


*Abbildung 40: Graphische Darstellung einer Dependency zwischen zwei Targets in der Clipper Software [107]*

Alle relevanten Informationen, werden ebenfalls, wie in Abbildung 40 für die jung_api Library, im Interface graphisch dargestellt um die Entscheidung zu erleichtern, welche Dependencies nun gelöscht oder geändert werden können.

### Zombie Targets

Ein ähnliches Problem stellt Code dar dessen Build zwar immer gestartet wird, aber seit einer längeren Zeitperiode fehlschlägt. Hierbei handelt es sich um sogenannte Zombie Targets, also toten Code der zwar laut Build Dateien noch ausgeführt werden soll, aber keinen Nutzen mehr hat, wodurch fehlgeschlagene Builds nicht auffallen oder ignoriert werden. Dieser tote Code erschwert eventuelle Refactorings deutlich, da Entwickler schnell den Überblick darüber verlieren können, welcher Code nun eingesetzt wird oder nicht. Um dies zu lösen könnte ein Startup, wie bei Google, ein Build Tool verwenden das es ermöglicht, bei jedem Upgrade des Build Systems, alle Targets zu bauen und die Ergebnisse zu speichern. Somit könnten Entwickler deren Dateien, wie bei Google zum Beispiel seit 90 Tagen nicht mehr bauen, gefragt werden, diese zu ändern oder zu löschen.

### Visibility Debt

Fehlende Sichtbarkeitsregeln führen dazu, dass Entwickler Dependencies zu anderen Targets eingehen, auf die sie eigentlich nicht hätten zugreifen sollen. Dies kann dazu führen, dass die Entwickler der anderen Targets diese ändern oder löschen und somit unbewusst eine Kettenreaktion bei den unbekannten Clients auslösen. Aus diesem Grund wurden die Sichtbarkeit aller Targets durch Morgenthaler et al. [107] von public auf private gesetzt. Falls die Entwickler ausdrücklich wollten, dass auf ihre Targets zugegriffen werden kann, konnten sie diese wieder als teilweise sichtbar oder public kennzeichnen. Das Ziel war es Entwickler zu sensibilisieren darauf zu achten, ungewollte Dependencies einzugehen oder zu ermöglichen.

### A.10. Process Debt

#### Beschreibung

Bei Process Debt handelt es sich um Prozesse die ineffizient sind, da sie zum Beispiel für die Entwicklung eines Produkts nie angebracht waren oder nicht mehr ausreichend sind [30] [108]. Da dieser TD-Typ den Softwareentwicklungsprozess selbst betrifft, ist dieser somit phasenübergreifend und von der Planungs-, bis hin zur Wartungsphase, wiederzufinden.

#### Ursachen und Folgen

Bei der InsighTD Studie wurden, wie in Abbildung 41 zu sehen, zwei Hauptursachen für Process Debt genannt und zwar ein *ineffektives Projektmanagement* und *falsche Zeiteinschätzungen*. Das Management von Startups ist oft unerfahren und weiß somit nicht, welche Praktiken vorteilhaft wären und ist ebenfalls nicht in der Lage einzuschätzen, wie lange gewisse Aktivitäten benötigen werden. Daraus resultieren ein für Kunden, Stakeholder und das Startup *unzufriedenstellendes Produkt* wegen dessen *schlechter Qualität* sowie häufige *Überarbeitungen*. Das Nicht-Einhalten gewisser Praktiken, wie zum Beispiel rechtzeitiges Testing, kann nämlich zu zahlreichen Mängeln und zusätzlicher Arbeit führen.

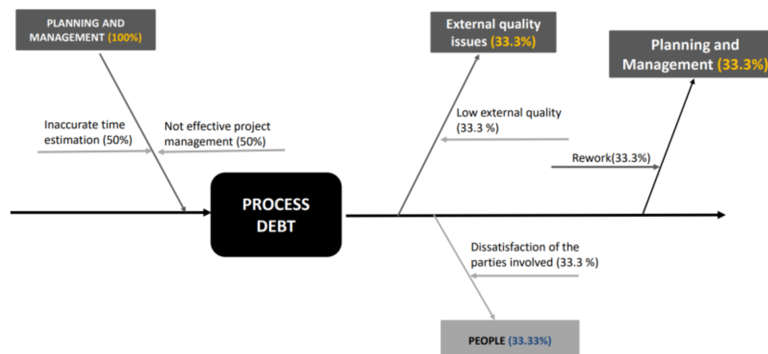


*Abbildung 41: Probabilistisches Ursache-Wirkungs-Diagramm für Process Debt [55]*

#### Indikatoren

Ein Großteil an Mängeln in Softwareprojekten kann, zumindest teilweise, auf Process Debt zurückführen. Davon sind einige:

- **Reaktives Verhalten**: Statt, zum Beispiel, gewisse Maßnahmen oder Praktiken einzusetzen um Bugs vorzubeugen, wird nur nach einer Lösung gesucht, nachdem ein Problem bereits entstanden ist [19].
- **Keine Verantwortlichkeiten**: Teammitgliedern werden keine klaren Rollen zugewiesen, sodass keine konkreten Zuständigkeiten vorliegen und Mitarbeitende zum Beispiel Aufgaben durchführen, für welche sie nicht qualifiziert sind [109].
- **Ineffizienz**: Langsame oder inexistente Prozesse, führen zu mangelhafter Kommunikation, Organisation und somit ein begrenzte Leistung im Team [110].

#### Methode: Strengeres Definition of Done

Process Debt gehört zu den TD-Typen die für Startups am schwierigsten zu bewältigen sind [23]. Startups lehnen die Idee von wiederholbaren und kontrollierten Prozessen meistens ab und bevorzugen es unvorhersehbare, reaktive und ungenaue technische Verfahren einzusetzen [111]. Sie machen dies aus Angst davor, dass bürokratische Maßnahmen ihre Kreativität und Flexibilität negativ beeinflussen könnten [19].

Um in etablierten Unternehmen Prozesse zu verbessern und deren Qualität zu messen, kann das sogenannte Capability Maturity Model (CMM) eingesetzt werden. Wie es Sutton [19] allerdings beschreibt, fehlt es Startups an nötigen Fundamenten um das CMM überhaupt einsetzen zu können, wie zum Beispiel Erfahrung, Infrastruktur zur Erstellung und Verbesserung von Prozessen oder wiederholbare und vorherseebare Praktiken. Trotz allem, müssen Startups gewisse Prozessaspekte und Technologien einsetzen, da sie nur so ein Niveau erreichen können, an dem das CMM für sie relevant wird [19]. Startups müssen sich dementsprechend eigene, leichtgewichtige Prozesse erstellen, um ein Minimum an Struktur und Skalierbarkeit zu garantieren.

Greening [80] bewies, anhand des Beispiels des Unternehmens Citrix Online, dass aus einem Startup ein Unternehmen werden kann, falls dieses an Scrum angelehnte, agile Prozesse einsetzt. Auch Brunner [38] beschreibt, wieso agile Prozesse in Startups von Vorteil sind. Manager behaupten oft, dass Startups von Natur aus agil sind, doch dies ist meistens nicht der Fall [80]. Agile Softwareentwicklung basiert auf iterativer und inkrementeller Entwicklung, sodass zum Beispiel eventuelle Defekte in kurzen Zyklen behoben werden und sich somit weniger technische Schulden ansammeln können [13].

In dem durch Brunner et al. erstellten Softwareentwicklungsprozess, welcher in Abschnitt 2.5.2 beschrieben wird, wird Continuous Delivery eingesetzt um die Software regelmäßig auszuliefern. Um dies zu bewerkstelligen muss allerdings händisch geprüft werden, ob die Software auch gewisse Kriterien erfüllt. Wie dies systematisch und strukturiert möglich ist, wird durch Davis [13] erklärt.

Bei einem Mechanismus der agilen Softwareentwicklung, handelt es sich um die sogenannten Definition of Done (DoD) [35]. Diese dient dazu zu prüfen, ob ein Softwareinkrement, wie zum Beispiel eine Story, auch wirklich als fertig angesehen werden kann, indem geprüft wird, ob es gewisse Ziele und Qualitätsattribute erfüllt [112]. Davis [13] erklärt dementsprechend, wie er es in mehreren Unternehmen geschafft hat, ein DoD einzuführen, wenn keines vorhanden war oder eine strengere DoD einzuführen und wie dies zu weniger technischen Schulden geführt hat.

Als erstes sollten die Teammitglieder geschult werden um sicherzustellen, dass diese auch verstehen worum es beim Konzept des DoD geht. Hierfür kann der Scrum Guide von Schwaber et al. [112] eingesetzt werden sowie das vereinfachte Beispiel aus Tabelle 8. Jede Reihe stellt eine, im DoD inkludierte Aktivität dar und die Spalten stellen entweder dar, welche Inkremente betroffen sind oder was erreicht werden muss, damit die Aktivität für das Inkrement als erfüllt gilt. Ein „X“ bedeutet, dass die Aktivität bzw. das Qualitätsattribut, auf das Inkrement in dieser Spalte zutrifft. Zum Beispiel ist ein Release nur vollständig, sobald dafür ein Performance Test durchgeführt und bestanden wurde. Die Reihenfolge in welcher sich die Attribute hierbei befinden, spielt keine Rolle.

*Tabelle 8: Beispielhaftes Definition of Done eines Softwareprojektes, in Anlehnung an [13]*

| Qualitätsattribute | Story | Release | Bemerkungen |
|---|---|---|---|
| **Design** | | | Das Design ist fertiggestellt und entspricht den Kundenwünschen |
| **Code und Unit Testing** | X | | Es wurden Test Cases für wichtige Code-Elemente geschrieben und bestanden |
| **Performance Test** | X | X | Die Software läuft vergleichbar schnell zu Konkurrenz-Produkten |
| **Usability** | X | | Kunden sind größtenteils zufrieden mit der Usability der Software |

Als nächstes können die Teammitglieder aufgefordert werden selbst DoD-Templates zu erstellen indem sie zum Beispiel Tools wie Rally [113] oder Agility [114] einsetzen. Wenn im Startup zum Beispiel keine konkreten Stories oder Sprints durchgeführt werden, sondern Software kontinuierlich ausgeliefert wird, dann können zum Beispiel DoD-Templates für Releases erstellt werden. So können sich die Mitglieder darauf einigen, welche Attribute für sie als wichtig angesehen werden. Bei der

Definierung der Attribute sollte allerdings ein hoher Wichtigkeitsgrad auf die Wünsche der Kunden gesetzt werden und das Produkt angepasst werden, falls es diesen nicht entspricht [181].
Ein Beispielprozess, der beschreibt wie Startups vorgehen können um einen Prototypen zu entwickeln und die Definition of Done einzusetzen, wird in Kapitel 6 beschrieben. Dieser stützt sich auf das in Abschnitt 2.5 präsentierte Entrepreneurial Software Engineering Model und erweitert es um Aspekte die dazu dienen sollen, die Entstehung von technischen Schulden zu minimieren.

## *A.11. Infrastructure Debt*

#### Beschreibung

Infrastructure Debt bezieht sich auf Probleme die entweder in der Hardware- oder Softwareinfrastruktur eines Unternehmens auftreten und nicht rechtezeitig behandelt werden [115]. Infrastructure Debt ist Phasenübergreifend und bezieht sich auf das Qualitätsmanagement des Unternehmens [116].

#### Ursachen und Folgen

Die wichtigste Ursache von Infrastructure Debt, ist, wies es sehr oft bei Startups der Fall ist, eine zu enge *Deadline* [55]. Es wird oft nicht die Zeit genommen eine korrekte Infrastruktur für ein Projekt bereitzustellen. Das Problem ist hierbei, dass Infrastruktur Debt, sehr schwere Folgen haben kann, da wenn die Infrastruktur zusammenbricht, ein Projekt zum Stillstand kommt [116]. Aus diesem Grund ist der größte Effekt von Infrastructure Debt, eine *schlechte Performance* des Unternehmens, da dieses nicht mehr normal operieren kann [55].

#### Indikatoren

Indikatoren die auf eine suboptimale Systeminfrastruktur hinweisen, sind vielseitig.

- **Verschobene Upgrades oder Infrastructure Fixes**: Sollten zum Beispiel neue Server oder ein Betriebssystem eingeführt werden, um Skalierbarkeit zu ermöglichen und dies wird nicht umgesetzt, kann das Wachstum des Startups gefährdet werden [30].
- **Veraltete Tools**: Ob es sich um Entwicklungstools handelt oder TD-Management Tools, führt das Einsetzen von älterer Software oft zu einer geringeren Effizienz und Produktivität, da es an Automatisierung fehlt [117].
- **Schlecht konfigurierte Infrastruktur**: Dies ist der Fall, wenn Maschinen oder zum Beispiel Software Services, schlecht verbunden oder eingerichtet wurden, sodass das ganze System unübersichtlich und ineffizient ist [115].
- **Viele Ausfälle und Instabilität**: Wenn das System oder Tools oft ungeplant langsamer werden oder abstürzen, weist dies auf eine suboptimale Infrastruktur hin. Verschlimmert wird dieser Aspekt, wenn das ganze Monitoring für den Zustand der Infrastrukturkomponenten entweder gar nicht oder nur auf den einzelnen Komponenten selber stattfindet, statt einer zentralisierten Übersicht [116].
- **Sich wiederholende Aufgaben**: Wenn zum Beispiel täglich ein Backup von allen Geräten auf einen Server gemacht werden muss, verursacht dies einen vervielfachten Zeitverlust [116].

### Methoden: DevOps und Infrastructure as Code

Startups beginnen meistens in kleinen Teams und haben somit Anfangs oft nur mit kleineren Infrastrukturen zu schaffen. Sollte der präsentierte Prototyp allerdings erfolgreich sein, kann sich dies sehr schnell ändern. Es sollte für Startups dementsprechend möglich sein, eine skalierbare Infrastruktur einzurichten.

In ihrer Anfangsphase können Startups hierfür zuerst ein vereinfachtes Vorgehen befolgen, da nur wenig Geräte und Tools Teil vom System sind. Im Grunde geht es darum eine Timeline zu erstellen und jedes Mal darauf aufzuschreiben, falls ein Vorfall im System auftritt [116]. Sollte zum Beispiel zu wenig Speicherplatz auf einer Festplatte oder einem Service vorhanden sein, ein Tool abstürzen, usw. Es sollten die Zeit, sowie der Ort im System und eventuell die zuständige Person aufgeschrieben werden. Dies ermöglicht es aus kleinen Anfangsfehlern zu lernen und somit Verbesserungskriterien für ein späteres System festzulegen. Ein anderes, oft unterschätztes Kriterium, um das zukünftige System zu verbessern, ist es die, in Abschnitt 5.3.12 präsentierte, People Debt zu reduzieren. Da, wie es Conways [118] Gesetzt beschreibt: „Unternehmen die Systeme designen… sind dazu gezwungen Designs zu erstellen, die Kopien der Kommunikationsstrukturen dieser Unternehmen sind“. Wie in einem Unternehmen kommuniziert wird, reflektiert sich letztendlich in dessen Systeminfrastruktur.

Sollte der Erste Prototyp sich bewiesen haben, kann das Startup als nächstes damit anfangen DevOps zu integrieren [115]. Hierbei geht es darum, den Abstand den es zwischen den Entwicklern (Dev) und dem operativen Teil (Ops) des Teams gibt, zu minimieren [119]. Dies wird erreicht, indem beide Abteilungen enger zusammenarbeiten und Infrastruktur mit ähnlichen Praktiken gehandhabt wird wie der Code, wodurch der Begriff *Infrastructure as Code* entstanden ist [120].

Um die, für das Startup notwendige, Infrastruktur zu erstellen, muss sich das Team zusammensetzen und aus den Erfahrungen der Timeline und Infrastrukturen von Konkurrenten, für das Unternehmen spezifische Schlüsse ziehen. Wenn das Unternehmen zum Beispiel viele Web-Basierte Dienste einsetzt, kann im Anschluss zum Beispiel das Tool DICER (Data-Intensive Continuous Engineering and Rollout) verwendet werden, um aus diesen Ideen ein konkretes Infrastrukturmodell zu erstellen [182]. In der Modellierungsumgebung des Tools, kann ein User direkt Nodes aus einer Eclipse IDE ziehen und sie mit entsprechenden Links verbinden, die gewünschte Bedingungen erfüllen [115]. Sobald das Team mit der Infrastruktur zufrieden ist, wird in DICER mit einem Klick ein sogenanntes TOSCA (Topology and Orchestration Specification for Cloud Applications) Blueprint erstellt, wie es in Abbildung 42 dargestellt ist. Bei TOSCA handelt es sich um einen industriellen Standard um Infrastructure as Code darzustellen [183].

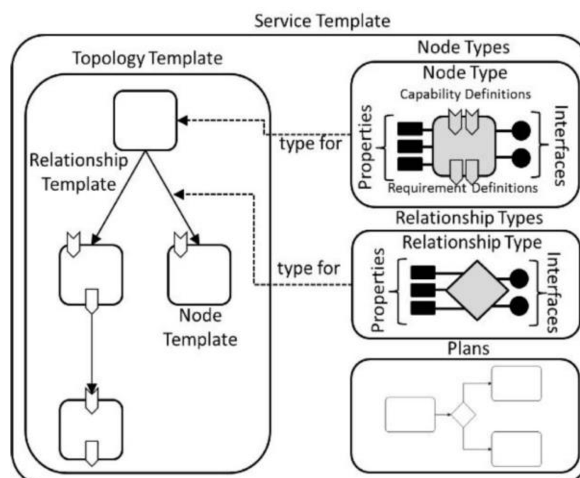


*Abbildung 42: TOSCA Service Template zur Darstellung einer Systeminfrastruktur [184]*

Das Blueprint stellt dar, welche Nodes im System involviert sind, wie zum Beispiel eine MySQL Datenbank oder ein WordPress Host und wie diese durch Relationships verbunden sind. Hieraus kann eine Topologie der Infrastruktur gewonnen werden [184]. In Plänen werden die Nodes und Relationships dann als Aufgaben dargestellt, um zu beschreiben, wie diese in Prozessen involviert sind. All diese Informationen stellen letztendlich das Service Template dar, das die Infrastruktur des Systems als Service und auf standardisierte Weise, darstellt. Möchte das Startup nun zum Beispiel Services mieten, fällt es durch die einheitliche Darstellung von System und Service viel einfacher, diese in die Infrastruktur zu integrieren.
Sobald die Hardware- und Softwareinfrastruktur des Startups eingerichtet ist, können dann APIs, wie zum Beispiel die durch Equinix angebotenen, eingesetzt werden [185]. Unter anderem ermöglichen es diese in Echtzeit, gewünschte Informationen, wie zum Beispiel Speicherplatz und Leistung zu monitoren und somit Vorfälle im Voraus zu verhindern.

### *A.12. People Debt*

#### Beschreibung

Bei People Debt geht es darum, dass der menschliche Aspekt zu technischen Schulden führen kann, falls dieser vernachlässigt wird. Dies betrifft sowohl zwischenmenschliche Interaktionen, als auch Qualifikationen der Teammitglieder [30] [121]. Menschen sind in allen Phasen des Entwicklungsprozesses involviert, sodass dieser TD-Typ das Projektmanagement betrifft.

#### Ursachen und Folgen

Zu den häufigsten Ursachen von People Debt, gehört ein Mangel an Ressourcen wie Zeit und Geld sowie eine ungenügende Erfahrung der Teammitglieder [122]. Startups sehen den menschlichen Aspekt daher als vorerst unwichtig an, was schnell zu einer mangelhaften Zusammenarbeit und somit einer verminderten Effizienz führt.

#### Indikatoren

In einem Projekt können zahlreiche Probleme auf People hinweisen, doch bei diesen handelt es sich um eindeutige Indikatoren:

- **Schlechte Arbeitsstimmung**: In Teams können Spannungen entstehen welche zu einem Mangel an Vertrauen und Zusammenarbeit führen. Dadurch wird die Teamdynamik und somit auch die resultierende Software, negativ beeinflusst [122].
- **Mangelhafte Kommunikation**: Es kann in allen Unternehmen vorkommen, dass es ein Mangel an Kommunikation zwischen Mitarbeitenden gibt oder zwischen verschiedenen Managementebenen [12]. Dies führt dazu, dass wichtige Informationen nicht übermittelt werden, wodurch sich technische Schulden ansammeln.
- **Späte Einstellungen oder Training**: Wenn Mitarbeitende zu spät rekrutiert oder geschult werden, führt dies dazu, dass zu wenig qualifiziertes Personal für das durchzuführende Projekt vorhanden ist. Dies zieht ebenfalls mit sich, dass das gesamte Wissen des Startups, sich in wenigen Personen konzentrieren muss und diese somit schnell überfordert sind [30].

#### Erste Methode: PANAS

People Debt gehört zu den TD-Typen welche am komplexesten zu studieren sind, da sie soziale, psychologische und organisationale Aspekte darstellt [12]. Es ist nur schwer möglich die technischen Schulden, welche durch gewisse soziotechnische Entscheidungen entstehen, zu identifizieren und zu messen [122]. Aber es gibt immer mehr Methoden die es ermöglichen, soziotechnische Aspekte zu

analysieren, was von hoher Wichtigkeit ist, da diese die Hauptursache von People Debt darstellen. Die dynamischen und miteinander verflochtenen Aktivitäten von Startups, erfordern nämlich eine enge Zusammenarbeit zwischen Teammitgliedern, aber auch Stakeholdern, Early Adopters und anderen involvierten Personen [7].

Ein wichtiger Aspekt, wenn es darum geht in einem Team zu arbeiten, stellt, wie erwähnt, die Stimmung dar [122], da Entwickler Produktiver sind, wenn sie gut gelaunt sind [123]. Aus diesem Grund muss in einem Startup von Anfang an sichergestellt werden, dass diese angemessen ist. Eine Methode, wie dies sehr einfach und schnell und somit optimal für Startups durchgeführt werden kann, sind sogenannte Stimmungsbarometer. Ein Beispiel hierfür wäre das, aus der Psychologie stammende PANAS (Positive and Negative Affect Schedule) bei dem Teammitglieder Fragebögen zur Selbsteinschätzung ausfüllen [124]. Den Mitarbeitenden werden, in einer vereinfachten und somit kürzeren Version dieser Befragung, Namens PANAS-Short, jeweils sechs positive und sechs negative Affekte präsentiert [125]:

- Interessiert, wach, aktiv, glücklich, stark, freudig erregt
- Gereizt, wütend, nervös, besorgt, verwirrt, ängstlich

Im Anschluss müssen die Teammitglieder jedes dieser Affekte auf einer Skala von eins bis fünf bewerten, wobei fünf bedeutet, dass das Affekt sehr auf sie zutrifft [124]. Dies kann, je nach Wunsch, anonym durchgeführt werden, da dies ehrlichere Antworten produziert. Der Manager kann letztendlich die Medianwerte der positiven und negativen Stimmungen für das ganze Team ausrechnen. Indem diese und die einzelnen Werte, ausgewertet werden, kann sich ein Überblick darüber verschafft werden, wie sich das Team fühlt. Entsprechend können Maßnahmen getroffen werden, wie zum Beispiel Privat- oder Teamgespräche, um die Ursache von Problemen zu finden.

#### Zweite Methode: SEnti-Analyzer

Ein für Startups noch einfacherer und somit optimaler Weg die Stimmung im Team zu analysieren, wird durch sogenannte Sentiment-Analyse Tools ermöglicht. Diese Tools haben als Ziel Kommunikation zwischen Teammitgliedern zu analysieren und die Polarität ihrer Aussagen zu bewerten [126]. In Startups kann auf unterschiedliche Weisen kommuniziert werden, doch es finden, vor allem in erfolgreichen Startups, immer Meetings statt [127]. Aus diesem Grund haben Herrmann und Klünder [128] ein Tool Namens SEnti-Analyzer entwickelt, das die Stimmung in Meetings analysieren kann, um, falls nötig, entsprechende Maßnahmen zu treffen.

Um dies zu bewerkstelligen, werden die Meetings entweder vor Ort oder durch die Mikrophone der Geräte der Teammitglieder aufgenommen [128]. Dabei wird Sprachaktivitätserkennung eingesetzt, um den Audiostream in einzelne Statements aufzuteilen. Als nächstes wird das Mozilla Deepspeech Framework [186] mit deutschen Sprachmodellen eingesetzt, um das entsprechende Transkript für die Aufzeichnung zu generieren. Sobald die Aufzeichnung beendet wurde, werden den Aussagen, durch Natural Language Processing, entsprechende Metriken zugewiesen. Diese Daten werden letztendlich dem Sentiment-Analyse Tool von Klünder et al. [187] übergeben, das ein trainiertes Klassifizierungsmodell einsetzt, um die Aussagen als „positiv", „negativ" oder „neutral" zu kategorisieren.

Als Endergebnis bekommt, zum Beispiel der Manager, einen Überblick über alle Aussagen und deren Polaritäten sowie über das Verhältnis der Polaritäten zueinander, im Verlauf des gesamten Meetings [128]. Dank dieser Informationen ist es dann möglich, die Kommunikation und Stimmung im Meeting zu analysieren und eventuelle Probleme im Voraus zu erkennen.

## A.13. Usability Debt

### Beschreibung

Bei Usability Debt handelt es sich um unangemessene Entscheidungen welche die Benutzerfreundlichkeit betreffen und die somit dazu führen, dass im Nachhinein Anpassungen am System und Interface nötig sind [9]. Diese Probleme entstehen meistens während der Design- und Implementierungsphase der Softwareentwicklung, wenn suboptimale Mockups oder Wireframes entworfen und entwickelt werden.

### Ursachen und Folgen

Wie in Abbildung 43 zu sehen, sind die häufigsten Ursachen die bei InsighTD genannt wurden, ein *Mangel an guten Praktiken* und an *Wissen bezüglich der einzusetzenden Technologien*. Unternehmen wissen also oft nicht was eine gute User Experience (UX) eines Systems überhaupt ausmacht und selbst wenn, wissen sie nicht, wie sie diese technisch umsetzen können. Dies führt dementsprechend zu einer *suboptimalen Benutzerfreundlichkeit* und einer *schlechteren Leistung* beim Einsatz der Software, da das Interface oft unübersichtlich gestaltet wurde.

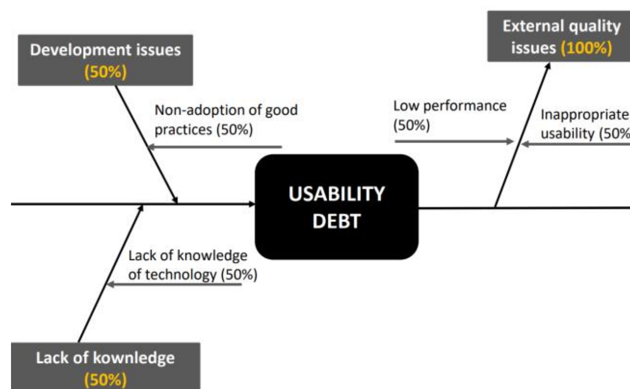


*Abbildung 43: Probabilistisches Ursache-Wirkungs-Diagramm für Usability Debt [55]*

### Indikatoren

Indikatoren für schlechtes UX-design sind vor allem für neue Nutzer sehr einfach zu erkennen, doch fallen Entwicklern nicht immer sofort auf. Davon sind, wie es Bandoly [129]beschreibt, einige:

- **Komplizierte Interfaces**: Oft wird versucht so viel Information wie möglich in einer einzigen Ansicht darzustellen, sodass, vor allem auf Startseiten, wirklich nützliche Informationen, untergehen.
- **Erlauben von falschen Verhalten**: Interfaces die es Usern ermöglichen, zum Beispiel falsche Dateitypen hochzuladen oder Formulare auf inkorrekte Weise auszufüllen, verfehlen das Ziel guter UI, selbsterklärend und intuitiv zu sein.
- **Mangel an Feedback**: Wenn User Aktionen durchführen, wie einen Knopf zu drücken um sich auszuloggen oder ein Dokument zu downloaden, erhalten diese kein Feedback. Somit wissen sie nicht, ob ihre Aktion nun überhaupt wahrgenommen wurde, was sehr schnell zu Frustration führen kann.

### Methode: Minimum Viable User Experience Framework (MVUX Framework)

Wenn sich viele Early Adopters für ein prototypisches Produkt interessieren, da sie dieses gerne verwenden, kann es dies ermöglichen, konstruktives Feedback zu erhalten [130]. Eine bereits ab dem Anfang gute UX anzubieten, kann zudem dazu führen, dass sich dies bei den Early Adopters schnell herumspricht, was zu einem guten Ruf des Produkts führt [131]. Da Startups, wie in Abschnitt 2.5

beschrieben, gerne sogenannte Minimum Viable Products (MVPs) entwickeln um ihre Idee zu testen, kann es also von Vorteil sein, bereits für dieses, eine angemessen UX anbieten zu können.
Aus diesem Grund haben Hokkanen et al. [132], zusammen mit zwölf Startups, ein sogenanntes Minimum Viable User Experience Framework entwickelt das beschreibt, welche UX-Faktoren beim Entwickeln eines MVPs unbedingt beachtet werden sollten. Das Framework wird in der unteren Abbildung 44 dargestellt und besteht aus drei Ebenen.

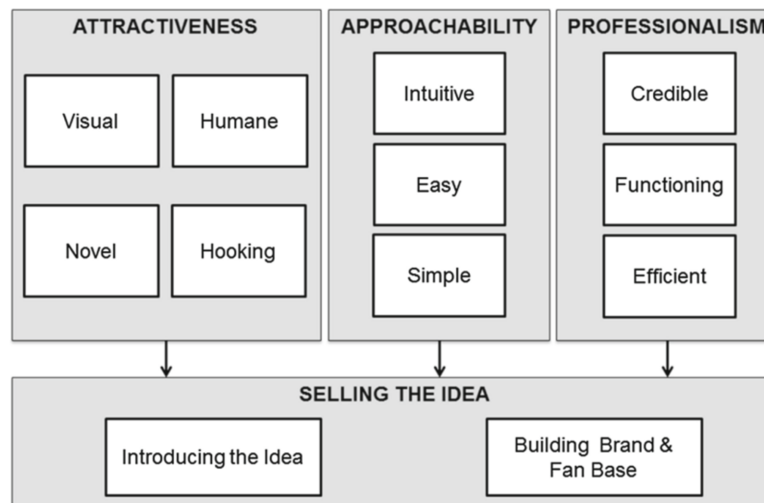


*Abbildung 44: MVUX-Framework zur Unterstützung der MVP-Entwicklung in Startups [132]*

Die höchste Ebene stellt Designziele wie Attraktivität dar, welche sich aus verschiedenen Faktoren zusammensetzen und diese Faktoren werden durch das Umsetzen gewisser Praktiken erreicht.
Bei dem Designziel Attraktivität geht es darum, dass der User einen positiven ersten Eindruck des Interfaces bekommt. Dieses muss also visuell ansprechend sein, das heißt das Design sollte, unter anderem, möglichst modern sein [188], indem zum Beispiel Googles Material Design [189] eingesetzt wird. Ebenfalls sollte das Interface human wirken, indem es sich zum Beispiel dem User anpasst oder ihn in Benachrichtigungen direkt anspricht [190]. Das Design sollte neuartig wirken, indem es sich, zum Beispiel durch ein gewisses Farbschema, von anderen unterscheidet [191]. Und letztendlich kann das Interesse der User beibehalten werden, indem sogenannte Gamification eingesetzt wird, bei der es darum geht, die Interaktion mit dem System, spielerisch zu gestalten, indem der User zum Beispiel, für gewisse Aktionen, belohnt wird [192].
Beim nächsten Designziel geht es darum, die Zugänglichkeit des Systems zu gewährleisten. Hierfür soll die UX möglichst intuitiv sein, indem sich zum Beispiel Elemente an gewohnten Positionen befinden und die UX, User bei Ihren Aktionen leitet [193]. Die UX sollte auch möglichst einfach verständlich und leichtgewichtig sein, indem zum Beispiel, nur die minimal notwendige Anzahl an Elementen angezeigt wird [194].
Letztendlich sollte das System möglichst professionell wirken. Hierfür muss es vor allem glaubwürdig erscheinen, indem zum Beispiel die Sicherheitsaspekte aus Abschnitt 5.3.16 beachtet und dem User bekannt gemacht werden. Ein professionelles Produkt muss möglichst fehlerfrei und effizient funktionieren, wofür vor allem die Aspekte aus Abschnitten 5.3.3 und 5.3.14 beachtet werden können. Auch wenn bei einem MVP, nicht kritische Fehler normal sind [38].
Diese drei Designziele führen zum Hauptziel der Minimum Viable User Experience, nämlich die Idee des Startups auf überzeugende Weise vorzustellen. Die User sollen wissen wozu das Produkt dient und dessen Mehrwert erkennen, sodass sich die Produktidee herumspricht.

## *A.14. Test Automation Debt*

### Beschreibung

Bei Test Automation Debt handelt es sich um technische Schulden die entstehen, wenn automatisierte Tests erst eingeführt werden, wenn Funktionalitäten bereits entwickelt wurden [30] [45]. Dieser TD-Typ erscheint somit erst ab der Testphase des Entwicklungszyklus und betrifft das Qualitätsmanagement des Startups.

### Ursachen und Folgen

Wie bei den meisten TD-Typen, gehört zu den häufigsten Ursachen von Test Automation Debt, wie es in Abbildung 45 zu sehen ist, eine zu enge *Deadline*, sodass sich darauf fokussiert wird *mehr, statt besser, zu produzieren*. Dem fügt sich hinzu, dass Unternehmen oft *unterschätzen wie wichtig Testing ist* und *nicht wissen, wie sie es automatisiert durchführen* können. Dies hat zur Folge, dass das *Testen der Software erschwert* wird, da unter anderem, bei jedem größerem Inkrement, alle *Tests wieder*, größtenteils händisch, *durchgeführt werden* müssen.

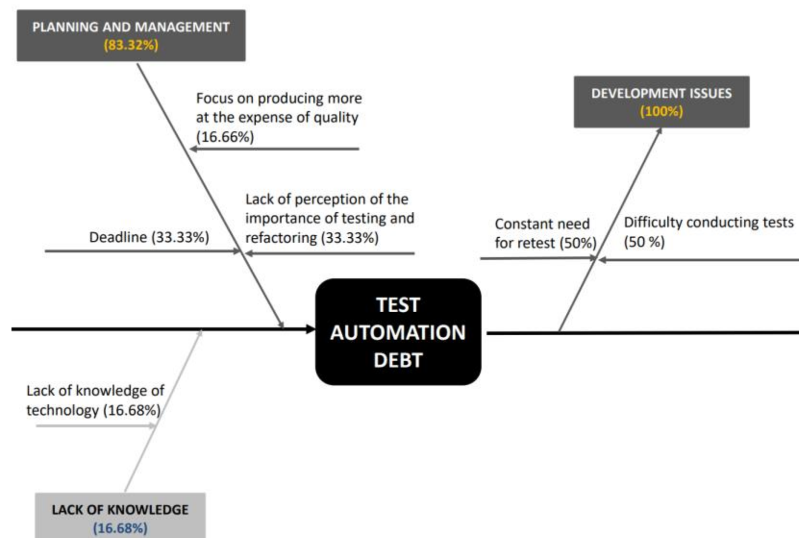


*Abbildung 45: Probabilistisches Ursache-Wirkungs-Diagramm für Test Automation Debt [55]*

### Indikatoren

Es weisen mehrere Aspekte auf die Existenz von Test Automation Debt hin:

- **Unbekannte oder nicht ausreichende Abdeckung durch automatisierte Tests**: Wenn das Startup keinen Überblick darüber hat, welcher Anteil seines Codes durch automatisierte Tests geprüft wird oder wenn es weiß, dass nur ein Bruchteil seiner Software, ohne manuelles Eingreifen getestet wird [133].

- **Automatisiertes Testen ohne definierte Strategie**: Ebenfalls wird Test Automation Debt durch schlecht geplantes automatisiertes Testen verursacht. Dies führt dazu, dass Entwickler den Überblick über den Test-Code verlieren und dadurch mehr Kosten verursacht werden, als die die potenziell gespart worden wären [134].

- **Unterbesetztes oder -qualifiziertes Tester-Team**: Es wird oft gedacht, dass automatisiertes Testen weniger Personal oder Qualifikationen benötigt, obwohl dies keineswegs der Fall ist [135].

### Methode: Sechs Aspekte für gutes Automated Testing

Wie es Wiklund et al. [45] beschreiben, wird beim Einführen von automatisierten Tests in Unternehmen meistens nicht dieselbe Rigorosität eingesetzt, wie bei der Entwicklung der Software. Dies führt zu einer erhöhten Menge an technischen Schulden.
Persson et al. [133] wollten durch eine Studie beweisen, dass das befolgen eines präzisen Frameworks garantieren könnte, automatisiertes Testen optimal in einem Unternehmen zu implementieren. Nachdem sie allerdings automatisiertes Testing in zwei verschiedenen Projekten einführten, stellte sich heraus, dass ein informelles, projektspezifisches Vorgehen, deutlich bessere Ergebnisse erbrachte. Dies liegt daran, dass automatisiertes Testing und entsprechenden Tools je nach Projekt sehr unterschiedlich eingeführt und gewählt werden müssen. Ein konkretes Framework ist dementsprechend nicht nützlich, doch trotzdem es hilft, einige, durch Persson et al. [133] definierte, Aspekte zu beachten:

- Für jedes Projekt bei dem automatisiertes Testen eingesetzt wird, sollte durch das Team ein Wörterbuch oder „Wiki“ erstellt und im Laufe der Zeit vervollständigt werden. Dies liegt daran, dass Vokabular zum automatisierten Testen Begriffe aus verschiedenen Disziplinen vereint und eine Übersicht, mit der entsprechenden Terminologie, somit Unklarheiten bei Teammitgliedern beseitigen würde.

- Das Team sollte entsprechend geschult und zusammengestellt werden. Um die Schulung zu ermöglichen, ergeben sich zwei Möglichkeiten. Ein Teammitglied sollte das Buch Automated Software Testing von Dustin et al. [135] studieren und im Anschluss seinen Kollegen einen Crash-Kurs über die, für das Projekt relevanten Inhalte, geben. Falls es, wie im nächsten Punkt beschrieben, die Ressourcen erlauben einen externen Dienstleister hierfür einzustellen, wäre dies umso besser, da dieser wichtige Erfahrung mit sich bringen kann. Durch solch einen Kurs, können die üblichen Probleme bei Test-Automatisierung besprochen und somit im Voraus vermieden werden.

- Vor allem das Planen der Einführung von automatisiertem testen benötigt viele menschliche, finanzielle und zeitliche Ressourcen. Oft mehr als für manuelles Testen ab dem Anfang nötig wäre, doch sobald die Automatismen in Kraft treten, werden die benötigten Ressourcen für die Wartung, überschaubarer. Es können einfache sogenannte Business-Impact-Analysen durchgeführt werden um abzuschätzen, wie hoch zum Beispiel die Kosten wären, einen Teil der Tests zu automatisieren [195]. Hierfür wird geprüft welche Mitarbeitende und Projektaspekte durch die Einführung wie lange betroffen wären.

- Zusätzlich zu Programmierern und Entwicklern, sollte ebenfalls ein ausgewogenes Verhältnis an Mitarbeitenden mit anderen Verantwortlichkeiten wie zum Beispiel Projektmanagement oder Datenbankenverwaltung vorhanden sein. Wissen sollte zudem nicht in externen Dienstleistern konzentriert werden, sondern durch das Startup angeeignet werden, sodass dieses nicht verloren geht.

- Es kann hilfreich sein einen Test Plan zu erstellen um zum Beispiel zu definieren welche Funktionalität geprüft werden soll, wie viel Defekte dabei akzeptiert werden, wann sie getestet werden soll und weitere Aspekte [196]. Somit kann ebenfalls sichergestellt werden, dass ein klares Ziel vorhanden ist und nicht unnötig viele Test Cases geschrieben werden.

- Letztendlich sollte ein Defect Tracking Tool, wie zum Beispiel Bugzilla [136] eingesetzt werden. Dies ermöglicht es, Defekte im Laufe der Zeit und nach Änderungen zu verfolgen und somit zu

beobachten, wie sich diese in Anzahl und Form ändern um die Test Cases entsprechend, auf strukturierte Weise, anzupassen.

Es müssen also einige Schlüsselaspekte beachtet werden um Tests in einem Startup zu automatisieren. Doch es muss beachtet werden, dass Automatisierung manuelles Testen nicht ersetzt, sondern beide komplementär zueinander sind [133].

### *A.15. Service Debt*

#### Beschreibung

Unternehmen mieten oft sogenannte Web Services um auf Ressourcen zuzugreifen die sie selber nicht haben, aber benötigen. Hierbei kann es sich zum Beispiel um mehr Rechenleistung [137] oder Datenspeicher [138] handeln. Das Verwenden eines Web Services kann technische Schulden mit sich ziehen, wenn dieser zum Beispiel nicht den Anforderungen entspricht oder unterbenutzt ist [30]. Diese Art von Schulden kann in allen Phasen der Softwareentwicklung auftauchen, da sowohl unangemessene Anforderungen zur Auswahl von unangebrachten Web Services führen kann, als auch erst festgestellt werden kann, dass die Web Services mangelhaft sind, sobald die Software ausgeliefert und gewartet wurde [137].

#### Ursachen und Folgen

Auch für Service Debt wurden, wie durch Rios et al. [55] in Abbildung 46 beschrieben, bei Insight TD nur wenig Ursachen und Folgen genannt. Als Gründe für diese Art an Schulden wurden eine unangemessene *Deadline* und *Entscheidungen des Managements* genannt, sowie zu hohe *Kosten* um den passenden Service verwenden zu können. Ebenfalls war ein *Mangel an Wissen bezüglich Services* für Service Debt verantwortlich, sowie *Service Versionen* die mit der eigenen *inkompatibel* waren. Diese Ursachen haben hauptsächlich dazu geführt, dass *erneut Services ausgewählt* werden mussten und, dass sowohl *finanzielle Verluste* als auch *Verzögerungen bei der Auslieferung* der Software entstanden.

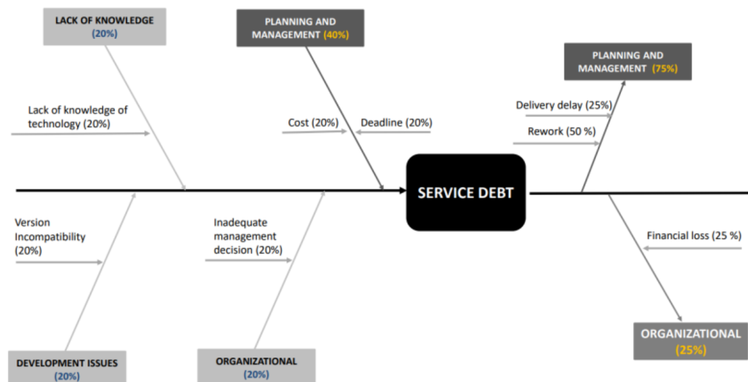


*Abbildung 46: Probabilistisches Ursache-Wirkungs-Diagramm für Service Debt [55]*

#### Indikatoren

- **Auswahl oder Ersatz eines Web Services** [70]: Die Tatsache, dass zu einem anderen Web Service gewechselt werden muss bedeutet, dass der aktuelle nicht mehr angemessen ist da er zum Beispiel zu langsam oder zu teuer ist [137].
- **Mangel an Ressourcen, Skalierbarkeit, Rechenleistung und anderen Quality of Service Attributen** [137]**.**

### Methode: Real Options

Ein Ansatz um die Menge an Service Debt in einem System bereits im Voraus zu minimieren, wird durch Alzaghoul et al. [137] vorgeschlagen. Dabei werden sogenannte Real Options besprochen, die eingesetzt werden sollen, um zwischen mehreren Web Services, den besten auszuwählen. Real Options sind Optionen, also Handlungsmöglichkeiten, eines nicht finanziellen Kapitals, wie zum Beispiel eines Software Projektes [137]. Bei Web Services handelt es sich zum Beispiel um Dienste in der Cloud die das Unternehmen mieten kann, um zum Beispiel mehr Rechenleistung zu erhalten. Das Mieten dieser Web Services kann als Kredit angesehen werden dem sich Zinsen hinzufügen, je nachdem wie effizient dieser eingesetzt wurde. Diese Zinsen entsprechen den technischen Schulden, die es zu beseitigen gilt.

Um zu zeigen wie die Methode funktioniert, wird durch Alzaghoul et al. [137] das Beispiel einer Softwarefirma verwendet, die sich mehr Skalierbarkeit wünscht, da dessen Produkt gerade ein großes Wachstum erfährt. Es werden zwei sogenannte Call Options verglichen die das Unternehmen hat, um entweder Methoden seines Systems durch einen Web Service 1 (WS1) oder einem Web Service 2 (WS2) ausführen zu lassen. Der Wert V des Systems im genannten Beispiel hängt nur davon ab, wie viel User das System zu einem Zeitpunkt nutzen und nutzen können. Definiert wird V durch folgende Formel:

$$V_t = V_s + V_{qos_1(t)} + V_{qos_2(t)} + \ldots + V_{qos_n(t)} \text{ [137]}$$

$V_t$ ist der Wert des Systems nachdem zu eine Webservice gewechselt wurde, $V_s$ der Wert des Systems vor dem Wechsel und $V_{qosn(t)}$ ist die Utility Value die erhalten wurde, indem die Quality of Service, also die Skalierbarkeit, durch den Wechsel verbessert wurde.

In Tabelle 9 werden die Eigenschaften der beiden Web Services genannt. Zu beobachten ist, dass nur die Verfügbarkeit sich bei beiden Diensten unterscheidet, sowie die Kosten die monatlich nötig sind, um zu den Diensten zu wechseln (Switching Costs).

*Tabelle 9: Eigenschaften der analysierten Web Services [137]*

| Attributes | Web Service1 | Web Service2 |
|---|---|---|
| Expiry Date | 6 months | 6 months |
| Switching Cost | £130.00 | £120.00 |
| Exercise Price | £100.00 | £80.00 |
| Capacity | 1140 | 1140 |
| Availability | 99.9 % | 90 % - 99.9 % |

Im durch Alzaghoul et al. [137] genannten Beispiel, wird ein Binomialmodell verwendet um die Werte des Systems und dadurch der Call Optionen, im Laufe der Zeit vorherzusagen. Daraus ergeben sich die in Abbildung 47dargestellten Werte.

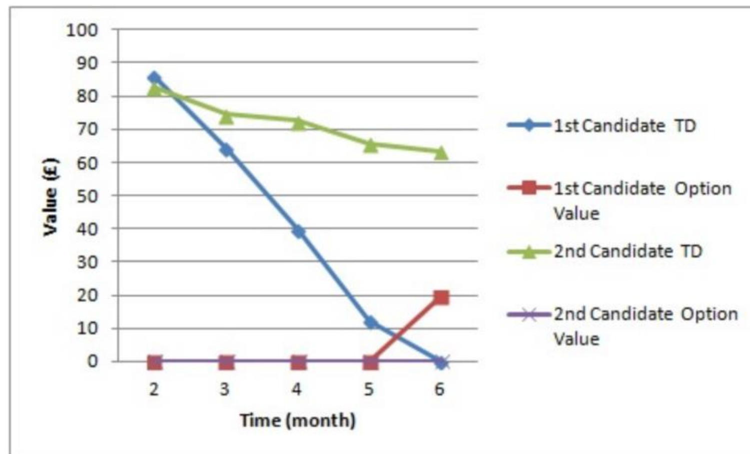


*Abbildung 47: Technische Schulden im Vergleich zum Wert der Optionen der Web Services im Laufe von sechs Monaten [137]*

Durch das Wechseln zu Cloud basierten Web Services, verursachen beide Dienste Anfangs ähnlich hohe technische Schulden. Dies liegt daran, dass die Wechselkosten monatlich bezahlt werden, aber noch nicht genügend User die Dienste verwenden und somit noch zu wenig Einkommen und zu viel überflüssige Kapazität der Web Services, vorhanden ist. Zu beobachten ist allerdings, dass bei WS1 die technischen Schulden sehr schnell sinken, bis sie im sechsten Monat einen Wert von null erreichen. Dies liegt daran, dass die nötige Skalierbarkeit vorliegt um 1140 Nutzer zu beherbergen und vor allem entsprechend viele Nutzer den WS1 nutzen.
Die technischen Schulden von WS2 hingegen sinken nur sehr langsam und die Call Option von WS2 fängt ebenfalls nie an, an Wert zu gewinnen. Dies liegt daran, dass WS2 eine geringere Verfügbarkeit hat und es somit weniger Nutzer verwenden würden als WS1. Das führt dazu, dass Kapazität für 1140 Nutzer zwar vorhanden wäre und dafür bezahlt wird, aber durch die unregelmäßige Verfügbarkeit nicht genügend User das System mit WS2 verwenden und dafür bezahlen würden. Optimal wäre es daher die Call Option für WS1 zu wählen, da diese Option, dem Graphen nach, nach einigen Monaten zu einem Profit führen könnte, sobald dessen Wert die monatlichen Wechselkosten von 130 £ überholt.
Das Vorgehen von Alzaghoul et al. [137] hat also verbildlicht wie wichtig es ist, auf die Eigenschaften von Web Services zu achten. Ein Startup könnte Dank dieses Beispiels, auch wenn es diese Methode nicht vollkommen umsetzt, aus den Eigenschaften von Web Services herauslesen, welche in Zukunft einschränkend werden könnten und dementsprechend die beste Call Option auswählen.

## A.16. Security Debt

### Beschreibung

Bei dem letzten TD-Typen, der Security Debt, handelt es sich um die Schulden die entstehen, wenn an der Sicherheit der entwickelten Software gespart wird und somit Sicherheitslücken entstehen und ausgenutzt werden können [139]. Sicherheit ist kein Feature das Software einfach hinzugefügt werden kann, sondern eher eine Eigenschaft die daraus resultiert, wie die Software konzipiert wurde [139]. Aus diesem Grund, kann die Entstehung von Security Debt nicht einer gewissen Phase der Softwareentwicklung zugewiesen werden und das Management dieses TD-Typen betrifft eher das Qualitäts- und Risikomanagement.

### Ursachen und Folgen

Zu den häufigsten Ursachen für Security Debt gehören das *Unterschätzen von dessen Gefahren*, eine *zu enge Deadline* und ein *Mangel an Fachwissen* [81]. Da Sicherheitsprobleme, im Gegenteil zu Bugs zum Beispiel, nicht unbedingt sofort sichtbar sind, werden diese mit einer niedrigeren Priorität

bearbeitet. Die schlimmste Folge einer erhöhten Security Debt, kann ein Vertrauensverlust bei den Kunden sein [139], was vor allem bei kleinen Unternehmen wie Startups, verheerende Konsequenzen haben kann.

#### Indikatoren

Durch die Common Weakness Enumeration [140], wurde sich auf Aspekte geeinigt die beschreiben, welche Schwachstellen in einem System vorhanden sein können. Davon sind einige:

- **Unsachgemäße Eingabevalidierung**: Es können in das System durch Nutzer Daten eingegeben werden, doch es wird nicht geprüft, ob diese daten die nötigen Eigenschaften erfüllen, um sicher verarbeitet zu werden. Hierdurch können Angreifer zum Beispiel das System lahmlegen oder bösartigen Code ausführen.
- **Fehlende Authentifizierung für kritische Funktionen**: Personen können auf Daten zugreifen ohne ihre Identität bestätigen zu müssen. Vor allem in Zeiten wo immer mehr Daten auf der Cloud gespeichert werden, kann dies Datendiebstahl deutlich vereinfachen.
- **Server-Side Request Forgery**: Ein Angreifer kann einem Server eine bestimmte, spezifisch erstellte, URL schicken [141]. Der Server wird durch diese URL getäuscht und fügt ihr, zum Beispiel sensitive Daten hinzu, auf die der Angreifer dann zugreifen kann.

#### Methode: Beachten der Zehn Mängel der IEEE Computer Society

Es gibt Unmengen an Problemen in Systemen, die zu potenziellen Sicherheitsrisiken führen können. Aus diesem Grund haben sich Forscher des IEEE Computer Society Center for Secure Design [139] auf die wichtigsten Mängel geeinigt und kurze Beispiele genannt, wie diese umgangen werden können.

- Beim ersten Punkt geht es darum, dass es möglichst vermieden werden sollte, sensible Daten des Systems und der User auf externen Geräten zu speichern. So sollte geprüft werden, ob es wirklich nötig ist, Daten zum Beispiel in Web Browsern oder Smartphones der Kunden zu speichern und falls dies unvermeidlich ist, sollte sichergestellt werden, dass die Daten auch entsprechend geschützt sind. Um letzteres zu erreichen, können Techniken wie Daten-Obfuskation eingesetzt werden, die Daten verfremdet und verschleiert, um sie für unbefugte Personen unbrauchbar zu machen [142]. Ebenfalls kann die externe Speicherung von Daten zeitlich begrenzt werden, sodass sie nach einer gewissen Dauer entfernt werden, falls sie nicht gebraucht werden.
- Beim nächsten Aspekt geht es darum, Authentifizierungsmechanismen zu verwenden die nicht umgangen oder manipuliert werden können. Um dies zu ermöglichen, sollten die Authentifizierungsmethoden mehrere Faktoren einsetzen, wie zum Beispiel etwas was der User weiß, biometrische Eigenschaften des Users oder etwas was der User besitzt. Die sicherste Methode bietet immer noch ein gutes Passwort, doch dieses sicher zu speichern, ist nicht trivial. Aus diesem Grund empfehlen Arce et al. [139] hierfür einen Kryptographie-Experten zu beauftragen und ein bewährtes Passwort Management System einzusetzen, auch wenn dies für Startups Anfangs zusätzliche Kosten verursachen würde.
- In gewissen Fällen reicht Authentifizierung alleine nicht aus und es müssen den Mitarbeitenden oder Usern gewisse Rechte zugewiesen oder entzogen werden. Wenn nun zum Beispiel ein Mitarbeiter ein Unternehmen verlässt, muss verhindert werden, dass dieser noch Zugriffsrechte auf zuvor autorisierte sensible Daten hat. Wenn ein Mitarbeiter oder Kunde auf vertrauliche Informationen zugreifen möchte, kann auch verlangt werden,

zusätzlich zur Authentifizierung, noch eine höhere Authentifizierung durchzuführen, um dessen Zugriffsrechte zu prüfen.

- Daten- und Steuerungsanweisungen sollten immer strikt getrennt und richtig verarbeitet werden. Wenn in einem System Daten eingegeben werden können, kann es zum Beispiel passieren, dass ein Angreifer in das Datenfeld eine SQL-Anfrage eingibt um somit unerlaubt an Daten zu gelangen. Der in das Datenfeld eingegebene Inhalt sollte dementsprechend stets als reine Daten und nicht als ausführbarer Code gespeichert werden. Ist es allerdings nötig, eingegebene Inhalte als Code auszuführen, kann zum Beispiel Datenvalidierung eingesetzt werden. Hierbei wird der Inhalt, zum Beispiel anhand von regulären Ausdrücken, auf eventuelle Steuerzeichen geprüft oder sichergestellt, dass der Inhalt dem erwarteten Format entspricht [197]. Um dies für Startups zu vereinfachen, gibt es bereits eine Vielzahl an Libraries für Input-Validierung.
- Letztendlich ermöglicht es der Einsatz von Kryptographie Daten zu verschlüsseln, den Zugriff auf diese zu regulieren und den Ursprung von Daten zu ermitteln. Einige Entwickler versuchen daher ihre eigenen kryptographischen Algorithmen zu entwickeln oder existierende Libraries einzusetzen, doch oft fehlt ihnen das Fachwissen um dies korrekt durchzuführen. Selbst wenn dies richtig gemacht wird, werden Kryptographische Schlüssel oft hart-kodiert oder schwache Schlüssel eingesetzt. Wie auch bei der Speicherung von Passwortdaten, empfehlen Arce et al. [139] dementsprechend hierbei nicht zu sparen, sondern einen Experten um Rat zu fragen, damit die Sicherheit der Software gewährleistet wird.

Obwohl Security Debt durch die Forschenden als unwichtiger bewertet wurde, bleibt auch dieser TD-Typ von hoher Wichtigkeit, da die Sicherheit betreffende Zwischenfälle, je nach Schweregrad, das frühzeitige Ende eines Startups verursachen können. Deshalb sollten die hier genannten Aspekte beachtet werden, welche den Qualitätscharakteristiken des ISO/IEC 25010 [143] Standards ebenfalls ähneln, um die grundlegende Sicherheit des Systems zu gewährleisten. Doch es gibt zahlreiche TD-Typen welche stark mit Security Debt zusammenhängen, sodass ebenfalls auf diese geachtet werden muss.

# Verzeichnisse

## Literaturverzeichnis


[1] N. Paternoster, C. Giardino, M. Unterkalmsteiner, T. Gorschek und P. Abrahamsson, „Software development in startup companies: A systematic mapping study,“ in *Information and Software Technology*, Science Direct, 2014, pp. 1200-1218.

[2] S. Álvarez, „In diesen Bereichen gibt es die meisten Neugründungen,“ 11 Juli 2023. [Online]. Available: https://www.wiwo.de/politik/deutschland/start-up-standort-deutschland-in-diesen-bereichen-gibt-es-die-meisten-neugruendungen/29252560.html. [Zugriff am 21 November 2023].

[3] A. Hirschfeld, J. Gilde und V. Walk, „Der Deutsche Startup Monitor 2023 ist veröffentlicht,“ PwC Deutschland, 25 September 2023. [Online]. Available: https://deutscherstartupmonitor.de/. [Zugriff am 21 11 2023].

[4] T. Eisenmann, Why Startups Fail: A New Roadmap for Entrepreneurial Success, P. R. H. LLC, Hrsg., Currency, 2021.

[5] C. Giardino, M. Unterkalmsteiner, N. Paternoster, T. Gorschek und P. Abrahamsson, „What Do We Know about Software Development in Startups?,“ *IEEE Software,* pp. 28-32, 2014.

[6] N. Brown, Y. Cai, Y. Guo, R. Kazman, M. Kim, P. Kruchten, E. Lim, A. MacCormack, R. Nord, I. Ozkaya, R. Sangwan, C. Seaman, K. Sullivan und N. Zazworka, „Managing technical debt in software-reliant systems,“ in *Proceedings of the FSE/SDP workshop on Future of software engineering research (FoSER '10)*, New York, 2010.

[7] M. Unterkalmsteiner, P. Abrahamsson, X. Wang, A. Nguyen-Duc, S. M. Ali Shah, S. Shahid 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 und A. Yagüe, „Software Startups -- A Research Agenda,“ *e-Informatica Softw. Eng.,* 2023.

[8] T. Huffine, „What is technical debt? And why does almost every startup have it?,“ 2017. [Online]. Available: https://www.freecodecamp.org/news/what-is-technical-debt-and-why-do-most-startups-have-it-9a54458daabf/. [Zugriff am 22 11 2023].

[9] N. Rios, M. G. de Mendonça Neto und R. O. Spínola, „A tertiary study on technical debt: Types, management strategies, research trends, and base information for practitioners,“ *Information and Software Technology,* Nr. Volume 102, pp. 117-145, 2018.

[10] N. S.R. Alves, T. S. Mendes, M. G. de Mendonça, R. O. Spínola, F. Shull und C. Seaman, „Identification and management of technical debt: A systematic mapping study,“ *Information and Software Technology,* Nr. 70, pp. 100-121, 2016.

[11] C. Seaman, G. Yuepu, N. Zazworka, F. Shull, C. Izurieta, Y. Cai und A. Vetrò, „Using technical debt data in decision making: Potential decision approaches,“ in *2012 Third International Workshop on Managing Technical Debt (MTD)*, 2012.

[12] P. Avgeriou, K. Philippe, I. Ozkaya und C. Seaman, „Managing Technical Debt in Software Engineering (Dagstuhl Seminar 16162),“ in *Dagstuhl Reports*, 2016.

[13] N. Davis, „Driving Quality Improvement and Reducing Technical Debt with the Definition of Done,“ in *2013 Agile Conference*, 2013.

[14] A. Omeyer, „The Cost of Technical Debt,“ 2020. [Online]. Available: https://stripe.com/files/reports/the-developer-coefficient.pdf. [Zugriff am 22 11 2023].

[15] J. Yli-Huumo, A. Maglyas und K. Smolander, „How do software development teams manage technical debt? – An empirical study,“ *Journal of Systems and Software,* 2016.

[16] Z. Block und I. C. MacMillan, „Milestones for Successful Venture Planning,“ 1985. [Online]. Available: https://hbr.org/1985/09/milestones-for-successful-venture-planning. [Zugriff am 22 11 2023].

[17] R. Sternberg, N. Gorynia-Pfeffer, F. Täube, L. Stolz, J. Schauer, A. Baharian und M. Wallisch, „Global Entrepreneurship Monitor 2022/2023 - Unternehmensgründungen im weltweiten Vergleich – Länderbericht Deutschland 2022/2023,“ 2023. [Online]. Available: https://www.rkw-kompetenzzentrum.de/publikationen/studie/global-entrepreneurship-monitor-2022-2023/. [Zugriff am 25 11 2023].

[18] Carmel, „Time-to-completion in software package startups,“ in *1994 Proceedings of the Twenty-Seventh Hawaii International Conference on System Sciences*, 1994.

[19] S. M. Sutton, „The role of process in software start-up,“ *Journal: IEEE Software,* pp. 33-39, 2000.

[20] M. Njima und S. Demeyer, „Value-based technical debt management: an exploratory case study in start-ups and scale-ups,“ in *IWSiB 2019: Proceedings of the 2nd ACM SIGSOFT International Workshop on Software-Intensive Business: Start-ups, Platforms, and Ecosystems*, 2019.

[21] O. Cico, R. Souza, L. Jaccheri, A. N. Duc und I. Machado, „Startups Transitioning from Early to Growth Phase - A Pilot Study of Technical Debt Perception,“ in *International Conference on Software Business*, 2021.

[22] E. Ries, The lean startup: How today’s entrepreneurs use continuous innovation to create radically successful businesses, Crown, 2011.

[23] M. Crowne, „Why software product startups fail and what to do about it. Evolution of software product development in startup companies,“ 2005.

[24] M. M. Lehman, „Programs, life cycles, and laws of software evolution,“ in *Proceedings of the IEEE*, 1980.

[25] W. Cunningham, „The WyCash portfolio management system,“ *SIGPLAN OOPS Messenger,* p. 29–30, 1992.

[26] B. Curtis, J. Sappidi und A. Szynkarski, „Estimating the size, cost, and types of Technical Debt,“ in *2012 Third International Workshop on Managing Technical Debt (MTD)*, 2012.

[27] P. Ken, „Understanding the impact of technical debt on the capacity and velocity of teams and organizations: Viewing team and organization capacity as a portfolio of real options,“ in *2013 4th International Workshop on Managing Technical Debt (MTD)*, 2013.

[28] W. Trumler und P. Frances, „How “Specification by Example” and Test-Driven Development Help to Avoid Technial Debt,“ in *2016 IEEE 8th International Workshop on Managing Technical Debt (MTD)*, 2016.

[29] S. McConnell, „Managing Technical Debt,“ in *Construx Software Development Best Practices*, Construx Software, 2008.

[30] N. S.R. Alves, L. F. Ribeiro, V. Caires, T. S. Mendes und R. O. Spínola, „Towards an Ontology of Terms on Technical Debt,“ in *2014 Sixth International Workshop on Managing Technical Debt*, 2014.

[31] F. Shull, D. Falessi, C. Seaman, M. Diep und L. Layman, „Technical Debt: Showing the Way for Better Transfer of Empirical Results,“ in *Perspectives on the Future of Software Engineering*, Springer, 2013, p. 179–190.

[32] L. Zengyang, L. Peng und A. Paris, „Architectural Technical Debt Identification Based on Architecture Decisions and Change Scenarios,“ in *2015 12th Working IEEE/IFIP Conference on Software Architecture*, 2015.

[33] K. Power, „Understanding the impact of technical debt on the capacity and velocity of teams and organizations: Viewing team and organization capacity as a portfolio of real options,“ in *2013 4th International Workshop on Managing Technical Debt (MTD)*, 2013.

[34] Z. Li, P. Avgeriou und P. Liang, „A systematic mapping study on technical debt and its management,“ *Journal of Systems and Software,* pp. 193-220, 2015.

[35] B. Raja, „Distributed Agile, Agile Testing, and Technical Debt,“ *IEEE Software,* pp. 28-33, 2012.

[36] Amazon Web Services, Inc., „Was ist SDLC (Software Development Lifecycle)?,“ 2023. [Online]. Available: https://aws.amazon.com/de/what-is/sdlc/#:~:text=The%20software%20development%20lifecycle%20(SDLC,expectations%20during%20production%20and%20beyond.. [Zugriff am 14 März 2024].

[37] D. Brunner, J. Münch und M. Kuhrmann, „Entrepreneurial Software Engineering: Towards a Hybrid Development Method for Early-Stage Startups,“ in *A Hybrid Development Method for Early-Stage Start-Ups*, 2021.

[38] D. Brunner, „A Hybrid Development Method for Early-Stage Start-Ups,“ 2020.

[39] M. R. Flowers, „Types of Minimum Viable Products [Series]: The Single Feature Application,“ 6 Oktober 2021. [Online]. Available: https://medium.com/@mdotflow22/types-of-minimum-viable-products-series-the-single-feature-application-e3c7e182a2bb. [Zugriff am 1 12 2023].

[40] C. Floyd, „A Systematic Look at Prototyping,“ in *Approaches to prototyping*, 1984.

[41] Lucidchart, „What Is a wireframe? Why you should start using this UX design tool,“ 2023. [Online]. Available: https://www.lucidchart.com/blog/what-is-a-wireframe#:~:text=A%20wireframe%20is%20a%20diagram,they%20can%20benefit%20your%20team.. [Zugriff am 4 Dezember 2023].

[42] S. Blank, The Four Steps to the Epiphany, K&S Ranch, 2013.

[43] J. De Luca, „Feature driven development,“ 2002. [Online]. Available: https://himolde.instructure.com/files/240682/download?download_frd=1.

[44] K. Beck, Test Driven Development: By Example, Addison-Wesley Professional, 2002.

[45] K. Wiklund, S. Eldh, D. Sundmark und K. Lundqvist, „Technical Debt in Test Automation,“ in *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, 2012.

[46] S. Freire, N. Rios, M. Mendonça, D. Falessi, C. Seaman, C. Izurieta und R. O. Spínola, „Actions and Impediments for Technical Debt Prevention: Results from a Global Family of Industrial Surveys,“ in *SAC '20: Proceedings of the 35th Annual ACM Symposium on Applied Computing*, 2020.

[47] N. Rios, R. Oliveira Spínola, M. Mendonça und C. Seaman, „Supporting Analysis of Technical Debt Causes and Effects with Cross-Company Probabilistic Cause-Effect Diagrams,“ in *IEEE/ACM International Conference on Technical Debt (TechDebt)*, 2019.

[48] V. Abrahamsson und V. Holmqvist, „Technical Debt in Swedish Tech Startups: Uncovering its Emergence, and Management Processes,“ 2023.

[49] H. Snyder, „Literature review as a research methodology: An overview and guidelines,“ *Journal of Business Research,* pp. 333-339, 2019.

[50] M. T. Pham, A. Rajić, J. D. Greig, J. M. Sargeant, A. Papadopoulos und S. A. McEwen, „A scoping review of scoping reviews: advancing the approach and enhancing the consistency,“ in *Research Synthesis Methods*, 2014, pp. 371-385.

[51] V. Mandić, N. Taušan und R. Ramač, „The prevalence of the technical debt concept in serbian IT industry: results of a national-wide survey,“ in *TechDebt '20: Proceedings of the 3rd International Conference on Technical Debt*, 2020.

[52] J. Bogner, R. Verdecchia und I. Gerostathopoulos, „Characterizing Technical Debt and Antipatterns in AI-Based Systems: A Systematic Mapping Study,“ in *2021 IEEE/ACM International Conference on Technical Debt (TechDebt)*, 2021.

[53] F. Gomes, E. Santos, S. Freire, T. Souto Mendes, M. Mendonça und R. Spínola, „Investigating the Point of View of Project Management Practitioners on Technical Debt - A Study on Stack Exchange,“ *Journal of Software Engineering Research and Development,* pp. 31-40, 2023.

[54] N. Kozanidis, R. Verdecchia und E. Guzman, „Asking about Technical Debt: Characteristics and Automatic Identification of Technical Debt Questions on Stack Overflow,“ in *ESEM '22: Proceedings of the 16th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement*, 2022.

[55] N. Rios, O. S. Rodrigo, M. Mendonça und C. Seaman, „The practitioners' point of view on the concept of technical debt and its causes and consequences: a design for a global family of industrial surveys and its first results from Brazil,“ in *Empirical Software Engineering*, 2020, p. 3216–3287.

[56] F. Shull, „Perfectionists in a World of Finite Resources,“ *IEEE Software,* pp. 4-6, 2011.

[57] N. Zazworka, A. Vetro', C. Izurieta, S. Wong, Y. Cai, C. Seaman und F. Shull, „Comparing four approaches for technical debt identification,“ *Software Quality Journal,* pp. 403-426, 2013.

[58] Lucid Software Inc., „Where seeing becomes doing.,“ 2024. [Online]. Available: https://www.lucidchart.com/pages/. [Zugriff am 14 März 2024].

[59] Y. Guo, R. O. Spínola und C. Seaman, „Exploring the costs of technical debt management – a case study,“ in *Empirical Software Engineering*, Springer, 2016.

[60] I. Griffith, H. Taffahi, C. Izurieta und D. Claudio, „A simulation study of practical methods for technical debt management in agile software development,“ in *Proceedings of the Winter Simulation Conference 2014*, 2014.

[61] O. Ktata und G. Levesque, „Designing and implementing a measurement program for Scrum teams: what do agile developers really need and want?,“ in *Proceedings of the Third C* Conference on Computer Science and Software Engineering*, 2010.

[62] P. Kruchten, R. L. Nord und O. Ozkaya, „Technical Debt: From Metaphor to Theory and Practice,“ *IEEE Software,* pp. 18-21, 2012.

[63] A. Martini, T. Besker und J. Bosch, „The Introduction of Technical Debt Tracking in Large Companies,“ in *2016 23rd Asia-Pacific Software Engineering Conference (APSEC)*.

[64] C. Izurieta, A. Vetrò, N. Zazworka, C. Yuanfang, C. Seaman und F. Shull, „Organizing the technical debt landscape,“ in *2012 Third International Workshop on Managing Technical Debt (MTD)*, 2012.

[65] A. Yamashita und L. Moonen, „Do code smells reflect important maintainability aspects?,“ in *2012 28th IEEE International Conference on Software Maintenance (ICSM)*, 2012.

[66] C. Izurieta , I. Ozkaya , C. Budinger Seaman, P. Kruchten, R. Nord, W. Snipes und P. Avgeriou, „Perspectives on Managing Technical Debt: A Transition Point and Roadmap from Dagstuhl,“ in *CEUR Workshop Proceedings*, 2016.

[67] R. Marinescu, „Assessing technical debt by identifying design flaws in software systems,“ *IBM Journal of Research and Development,* pp. 9:1-9:13, 2012.

[68] N. Zazworka, C. Seaman und F. Shull, „Prioritizing Design Debt Investment Opportunities,“ in *Proceedings of the 2nd Workshop on Managing Technical Debt*, 2011.

[69] J. Bohnet und J. Döllner, „Monitoring code quality and development activity by software maps,“ in *Proceedings of the 2nd Workshop on Managing Technical Debt*, 2011.

[70] N. S. R. Alves, R. Araújo und R. Spínola, „A Collaborative Computational Infrastructure for Supporting Technical Debt Knowledge Sharing and Evolution,“ in *Americas Conference on Information Systems*, 2015.

[71] F. A. Fontana, V. Ferme, M. Zanoni und R. Roveda, „Towards a prioritization of code debt: A code smell Intensity Index,“ in *2015 IEEE 7th International Workshop on Managing Technical Debt (MTD)*, 2015.

[72] O. Kalendin, „Code Smells: Dispensable — Data Class,“ 1 November 2023. [Online]. Available: https://o-kaledin.medium.com/code-smells-dispensable-data-class-723111ea80de. [Zugriff am 2 Januar 2024].

[73] D. Ananta Kumar, S. Yadav und D. Subhasish, „Detecting Code Smells using Deep Learning,“ in *IEEE Region 10 International Conference TENCON*, 2019.

[74] T. Guggulothu und S. Abdul Moiz, „Detection of Shotgun Surgery and Message Chain Code Smells using Machine Learning Techniques,“ in *Research Anthology on Machine Learning Techniques, Methods, and Applications*, Information Resources Management Association, 2022, pp. 800-816.

[75] J. Martins, C. Bezerra, A. Uchôa und A. Garcia, „Are Code Smell Co-Occurrences Harmful to Internal Quality Attributes? A Mixed-Method Study,“ in *Proceedings of the XXXIV Brazilian Symposium on Software Engineering*, 2020.

[76] M. Lanza, S. Ducasse, H. Gall und M. Pinzger, „CodeCrawler: An Information Visualization Tool for Program Comprehension,“ in *ICSE '05: Proceedings of the 27th international conference on Software engineering*, 2005.

[77] G. Bavota, A. Qusef, R. Oliveto, A. De Lucia und D. Binkley, „An empirical analysis of the distribution of unit test smells and their impact on software maintenance,“ in *2012 28th IEEE International Conference on Software Maintenance (ICSM)*, 2012.

[78] S. M. A. Shah, M. Torchiano, A. Vetro und M. Morisio, „Exploratory Testing as a Source of Technical Debt,“ *IT Professional,* pp. 44-51, 2014.

[79] D. Astels, Test Driven development: A Practical Guide, Prentice Hall Professional Technical Reference, 2003.

[80] D. R. Greening, „Release Duration and Enterprise Agility,“ in *2013 46th Hawaii International Conference on System Sciences*, 2013.

[81] K. Rindell, K. Bernsmed und M. G. Jaatun, „Managing Security in Software: Or: How I Learned to Stop Worrying and Manage the Security Technical Debt,“ in *ARES '19: Proceedings of the 14th International Conference on Availability, Reliability and Security*, 2019.

[82] J. Garcia, D. Popescu, G. Edwards und N. Medvidovic, „Identifying Architectural Bad Smells,“ in *2009 13th European Conference on Software Maintenance and Reengineering*, 2009.

[83] M. Eichberg, S. Kloppenburg, K. Klose und M. Mezini, „Defining and Continuous Checking of Structural Program Dependencies,“ in *ICSE '08: Proceedings of the 30th international conference on Software engineering*, 2008.

[84] C. Fernández-Sánchez, J. Díaz, J. Pérez und J. Garbajosa, „Guiding Flexibility Investment in Agile Architecting,“ in *2014 47th Hawaii International Conference on System Sciences*, 2014.

[85] B. Boehm, „Architecture-Based Quality Attribute Synergies and Conflicts,“ in *2015 IEEE/ACM 2nd International Workshop on Software Architecture and Metrics*, 2015.

[86] D. Ameller, C. Ayala, J. Cabot und X. Franch, „Non-functional Requirements in Architectural Decision Making,“ *IEEE Software,* pp. 61-67, 2013.

[87] R. T. Tvedt, P. Costa und M. Lindvall, „Does the code match the design? A process for architecture evaluation,“ in *International Conference on Software Maintenance, 2002. Proceedings*, 2002.

[88] V. Lenarduzzi und D. Fucci, „Towards a Holistic Definition of Requirements Debt,“ in *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 2019.

[89] C. Rupp, Requirements-Engineering und -Management: Professionelle, iterative Anforderungsanalyse für die Praxis., Carl Hanser Verlag GmbH & Co. KG, 2004.

[90] S. Klaus, „On the limits of the technical debt metaphor some guidance on going beyond,“ in *2013 4th International Workshop on Managing Technical Debt (MTD)*, 2013.

[91] H. Femmer, D. M. Fernández, S. Wagner und E. Sebastian, „Rapid quality assurance with Requirements Smells,“ *Journal of Systems and Software,* pp. 190-213, 2017.

[92] O. Karras, K. Schneider und S. A. Fricker, „Representing software project vision by means of video: A quality model for vision videos,“ *Journal of Systems and Software,* p. 29, 2020.

[93] H. F. Soares, N. S.R. Alves, T. S. Mendes, M. Mendonça und R. O. Spínola, „Investigating the Link between User Stories and Documentation Debt on Software Projects,“ in *2015 12th International Conference on Information Technology - New Generations*, 2015.

[94] C. Seaman und Y. Guo, „Measuring and Monitoring Technical Debt,“ *Advances in Computers,* pp. 25-46, 2011.

[95] C. Treude, J. Middleton und T. Atapattu, „Beyond Accuracy: Assessing Software Documentation Quality,“ in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020.

[96] H. Zhong, L. Zhang, X. Tao und H. Mei, „Inferring Resource Specifications from Natural Language API Documentation," in *2009 IEEE/ACM International Conference on Automated Software Engineering*, 2009.

[97] E. Pitler und A. Nenkova, „Revisiting Readability: A Unified Framework for Predicting Text Quality," in *In Proceedings of the 2008 conference on empirical methods in natural language processing*, 2008.

[98] D. M. Strong und R. Y. Wang, „Beyond Accuracy: What Data Quality Means to Data Consumers," *Journal of Management Information Systems,* pp. 5-33, 1996.

[99] S.-a. Knight und J. Burn, „Developing a Framework for Assessing Information Quality on the World Wide Web," *Informing Science The International Journal of an Emerging Transdiscipline,* pp. 159-172, 2005.

[100] L. Ren, S. Zhou, C. Kästner und A. Wąsowski, „Identifying Redundancies in Fork-based Development," in *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2019.

[101] Software Freedom Conservancy, „Git--local-branching-on-the-cheap," 2024. [Online]. Available: https://git-scm.com/. [Zugriff am 18 Januar 2024].

[102] INFOX, „INFOX beta Insights into Forks," 2024. [Online]. Available: http://www.forks-insight.com/. [Zugriff am 18 Januar 2024].

[103] W. Snipes, B. Robinson, Y. Guo und C. Seaman, „Defining the decision factors for managing defects: A technical debt perspective," in *2012 Third International Workshop on Managing Technical Debt (MTD)*, 2012.

[104] J. Unadkat, „7 Common Types of Software Bugs Every Tester Should Know," 2023. [Online]. Available: https://www.browserstack.com/guide/types-of-software-bugs. [Zugriff am 19 Januar 2024].

[105] K. Devaraj, „Common Types of Bugs in Software Testing," 2024. [Online]. Available: https://testsigma.com/blog/types-of-bugs-in-software-testing/#5_System-Level_Integration_Bugs. [Zugriff am 19 Januar 2024].

[106] M. Ruchika und B. Laavanye, „A defect tracking tool for open source software," in *2017 2nd International Conference for Convergence in Technology (I2CT)*, 2017.

[107] J. D. Morgenthaler, M. Gridnev, R. Sauciuc und S. Bhansali, „Searching for build debt: Experiences managing technical debt at Google," in *2012 Third International Workshop on Managing Technical Debt (MTD)*, 2012.

[108] C. Zadia und W. Byron, „Managing technical debt: An industrial case study," in *2013 4th International Workshop on Managing Technical Debt (MTD)*, 2013.

[109] J. R. Hackman, „The design of work teams," in *Handbook of organizational behaviour.*, Prentice Hall, 1987, p. 315–342.

[110] K. Schwaber, „SCRUM Development Process," in *Business Object Design and Implementation*, Springer, 1997, p. 117–134.

[111] A. Fuggetta, „Software process: a roadmap.," in *Proceedings of the Conference on the Future of Software Engineering*, 2000.

[112] K. Schwaber und J. Sutherland, „The 2020 Scrum Guide," 2020. [Online]. Available: https://scrumguides.org/scrum-guide.html. [Zugriff am 23 Januar 2024].

[113] Broadcom Inc., „Rally," 2024. [Online]. Available: https://techdocs.broadcom.com/us/en/ca-enterprise-software/valueops/rally/rally-help.html. [Zugriff am 23 Januar 2024].

[114] DIGITAL.AI, „Agility," 2024. [Online]. Available: https://digital.ai/products/agility/. [Zugriff am 23 Januar 2024].

[115] M. Artac, T. Borovssak, E. Di Nitto, M. Guerriero und D. A. Tamburri, „DevOps: Introducing Infrastructure-as-Code," in *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, 2017.

[116] A. Shafer, „Infrastructure Debt: Revisiting the Foundation," Cutter Consortium, 2010. [Online]. Available: https://www.cutter.com/article/infrastructure-debt-revisiting-foundation-416226. [Zugriff am 24 Januar 2024].

[117] D. Falessi, M. A. Shaw, F. Shull, K. Mullen und K. Mark Stein, „Practical considerations, challenges, and requirements of tool-support for managing technical debt," in *2013 4th International Workshop on Managing Technical Debt (MTD)*, 2013.

[118] M. E. Conway, „How Do Committees Invent?," *Datamation,* pp. 28-31, 1968.

[119] L. Bass, I. Weber und L. Zhu, DevOps: A Software Architect's Perspective (SEI Series in Software Engineering), Addison-Wesley, 2015.

[120] K. Morris, Infrastructure As Code: Managing Servers in the Cloud, Oreilly & Associates Incorporated, 2016.

[121] T. Edith, A. Aybüke und R. Vidgen, „An exploration of technical debt," *Journal of Systems and Software,* pp. 1498-1516, 2013.

[122] D. A. Tamburri, P. Kruchten, P. Lago und H. Vliet, „What is social debt in software engineering?," in *2013 6th International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE)*, 2013.

[123] D. Graziotin, X. Wang und P. Abrahamsson, „Happy software developers solve problems better: psychological measurements in empirical software engineering," *PeerJ,* p. e289, 2014.

[124] B. Breyer und M. Bluemke, „Deutsche Version der Positive and Negative Affect Schedule PANAS (GESIS Panel)," 2016. [Online]. Available: https://www.researchgate.net/publication/309210530_Deutsche_Version_der_Positive_and_Negative_Affect_Schedule_PANAS_GESIS_Panel. [Zugriff am 26 Januar 2024].

[125] F. Kortum, „Supporting the Understanding of Team Dynamics in Agile Software Development Through Computer-Aided Sprint Feedback," Logos Verlag Berlin GmbH, 2022.

[126] R. Kaur Bakshi, N. Kaur, R. Kaur und G. Kaur, „Opinion mining and sentiment analysis," *2016 3rd International Conference on Computing for Sustainable Global Development (INDIACom),* pp. 452-455, 2016.

[127] S. Delecourt, S. Hasan und R. Koning, „When does advice impact startup performance?," *Strategic Management Journal,* pp. 331-356, 2019.

[128] J. Klünder und M. Herrmann, „From Textual to Verbal Communication: Towards Applying Sentiment Analysis to a Software Project Meeting," in *2021 IEEE 29th International Requirements Engineering Conference Workshops (REW)*, 2021.

[129] A. Bandoly, „5 Signs of Poor User Experience Design," 2023. [Online]. Available: https://www.codesigned.com/blog/5-signs-of-poor-user-experience-design. [Zugriff am 6 Februar 2024].

[130] K. Väänänen-Vainio-Mattila und L. Hokkanen, „UX Work in Startups: Current Practices and Future Needs,“ in *Agile Processes in Software Engineering and Extreme Programming*, 2015.

[131] J. Füller, R. Schroll und E. Von Hippel, „User generated brands and their contribution to the diffusion of user innovations.,“ *Research Policy,* pp. 1197-1209, 2013.

[132] L. Hokkanen, K. Kuusinen und K. Väänänen, „Minimum Viable User EXperience: A Framework for Supporting Product Design in Startups,“ in *Agile Processes, in Software Engineering, and Extreme Programming*, 2016.

[133] C. Persson und N. Yilmazturk, „Establishment of automated regression testing at ABB: industrial experience report on 'avoiding the pitfalls',“ in *Proceedings. 19th International Conference on Automated Software Engineering, 2004.*, 2004.

[134] J. Bach, „Test Automation Snake Oil,“ in *Proceedings of the 14th International Conference and Exposition on Testing Computer Software*, 1999.

[135] E. Dustin, J. Rashka und J. Paul, Automated software testing: Introduction, management, and performance: Introduction, management, and performance., A. Professional, Hrsg., 1999.

[136] Zarro Boogs Corporation, „What is Bugzilla?,“ 2024. [Online]. Available: https://www.bugzilla.org/about/#:~:text=Bugzilla%20is%20a%20robust%2C%20featureful,requests%20in%20their%20products%20effectively.. [Zugriff am 29 Januar 2024].

[137] E. Alzaghoul und R. Bahsoon, „CloudMTD: Using real options to manage technical debt in cloud-based service selection,“ in *2013 4th International Workshop on Managing Technical Debt (MTD)*, 2013.

[138] Amazon Web Services, Inc., „Amazon S3 Objektspeicher für den Abruf beliebiger Datenmengen von jedem Ort aus,“ 2023. [Online]. Available: https://aws.amazon.com/de/s3/. [Zugriff am 11 Januar 2024].

[139] I. Arce, K. Clark-Fisher, N. Daswani, J. DelGrosso, D. Dhillon, C. Kern, T. Kohno, C. E. Landwehr, B. Schoenfield, M. I. Seltzer, D. D. Spinellis, I. Tarandach und J. West, „Avoiding the Top 10 Software Security Design Flaws.,“ 2014. [Online]. Available: https://ieeecs-media.computer.org/media/technical-activities/CYBSI/docs/Top-10-Flaws.pdf. [Zugriff am 2 Februar 2024].

[140] MITRE Corporation, „2023 CWE Top 10 KEV Weaknesses List Insights,“ 2023. [Online]. Available: https://cwe.mitre.org/top25/archive/2023/2023_kev_insights.html#. [Zugriff am 2 Februar 2024].

[141] Eoftedal, „Server Side Request Forgery,“ 2024. [Online]. Available: https://owasp.org/www-community/attacks/Server_Side_Request_Forgery. [Zugriff am 2 Februar 2024].

[142] Talend, Inc., „Obfuskation: Anonymisierung von Daten für mehr Schutz,“ 2024. [Online]. Available: https://www.talend.com/de/resources/data-obfuscation/#:~:text=Was%20ist%20Obfuskation%20von%20Daten,f%C3%BCr%20unbefugte%20Personen%20unbrauchbar%20sind.. [Zugriff am 3 Februar 2024].

[143] International Organization for Standardization, „ISO/IEC 25010,“ 2022. [Online]. Available: https://iso25000.com/index.php/en/iso-25000-standards/iso-25010. [Zugriff am 5 Februar 2024].

[144] Z. Codabux, B. J. Williams und N. Niu, „A Quality Assurance Approach to Technical Debt,“ in *Proceedings of the International Conference on Software Engineering Research and Practice (SERP)*, 2014.

[145] Equinix, Inc., „The Value of API-Driven Digital Infrastructure," 2024. [Online]. Available: https://blog.equinix.com/blog/2021/01/27/the-value-of-api-driven-digital-infrastructure/. [Zugriff am 11 April 2024].

[146] G. Rojas, C. Izurieta und I. Griffith, „Preemptive Management of Model Driven Technical Debt for," in *Proc. of the 11th Int'l ACM SIGSOFT Conf. on Quality of Software Architectures (QoSA'15)*, 2015.

[147] J. Schumacher, N. Zazworka, F. Shull, C. Seaman und M. Shaw, „Building empirical support for automated code smell detection," in *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, 2010.

[148] CAST Software, „Overview of CAST Application Intelligence Platform," 2024. [Online]. Available: https://doc-legacy.castsoftware.com/display/DOC83/Overview+of+CAST+Application+Intelligence+Platform. [Zugriff am 6 Februar 2024].

[149] T. S. Mendes, F. G. S. Gomes, D. P. Gonçalves, M. G. Mendonça, R. L. Novais und R. O. Spínola, „VisminerTD: a tool for automatic identification and interactive monitoring of the evolution of technical debt items," *Journal of the Brazilian Computer Society,* pp. 1-28, 2019.

[150] M. Farias, J. Santos, A. da Silva, M. Kalinowski, M. Mendonça und R. Spínola, „Investigating the Use of a Contextualized Vocabulary in the Identification of Technical Debt: A Controlled Experiment," in *Proceedings of the 18th International Conference on Enterprise Information Systems*, 2016.

[151] SonarSource SA., „clean code for teams and enterprises with {SonarQube}," 2024. [Online]. Available: https://www.sonarsource.com/products/sonarqube/. [Zugriff am 8 Februar 2024].

[152] SonarSource S.A, „Introduction to the server installation," 2024. [Online]. Available: https://docs.sonarsource.com/sonarqube/latest/setup-and-upgrade/install-the-server/introduction/. [Zugriff am 8 Februar 2024].

[153] SonarSource SA., „what is Clean Code?," 2024. [Online]. Available: https://www.sonarsource.com/solutions/clean-code/. [Zugriff am 8 Februar 2024].

[154] SonarSource SA., „SonarQube 10.4 Documentation," 2024. [Online]. Available: https://docs.sonarsource.com/sonarqube/latest/. [Zugriff am 8 Februar 2024].

[155] J. Holvitie und V. Leppänen, „DebtFlag: Technical debt management with a development environment integrated tool," in *2013 4th International Workshop on Managing Technical Debt (MTD)*, 2013.

[156] E. d. S. Maldonado und E. Shihab, „Detecting and quantifying different types of self-admitted technical Debt," in *2015 IEEE 7th International Workshop on Managing Technical Debt (MTD)*, 2015.

[157] N. Tsantalis, „JDeodorant," 2024. [Online]. Available: https://marketplace.eclipse.org/content/jdeodorant. [Zugriff am 7 Februar 2024].

[158] R. Torkar, P. Minoves und J. Garrigós, „Adopting Free/Libre/Open Source Software Practices, Techniques and Methods for Industrial Use," *Journal of the Association for Information Systems,* p. 11, 2011.

[159] C. Liu, „The 6-3-5 Technique Is the Brainstorming Replacement That Actually Works," 19 Juni 2020. [Online]. Available: https://www.themuse.com/advice/the-635-technique-is-the-brainstorming-replacement-that-actually-works. [Zugriff am 4 Dezember 2023].

[160] J. Bosch, „The early stage software startup development model: a framework for," in *International Conference on Lean Enterprise Software and Systems*, 2013.

[161] J.-P. Voutilainen, J. Salonen und T. Mikkonen, „On the Design of a Responsive User Interface for a Multi-device Web Service," in *2015 2nd ACM International Conference on Mobile Software Engineering and Systems*, 2015.

[162] Google for Developers, „Run apps on a hardware device," 2024. [Online]. Available: https://developer.android.com/studio/run/device. [Zugriff am 27 Februar 2024].

[163] Y. Guo, C. Seaman, R. Gomes, A. Cavalcanti, G. Tonin, F. Q. B. Da Silva, A. L. M. Santos und C. Siebra, „Tracking technical debt — An exploratory case study," in *2011 27th IEEE International Conference on Software Maintenance (ICSM)*, 2011.

[164] Wipro, „Analyzing a Project with SonarQube," Wipro, 2024. [Online]. Available: https://dx.appirio.com/quality-sonarqube/sonarqube-project-analysis/. [Zugriff am 26 Februar 2024].

[165] Y. Guo und C. Seaman, „A portfolio approach to technical debt management," in *Proceedings of the 2nd Workshop on Managing Technical Debt*, 2011.

[166] P. S. M. dos Santos, A. Varella, C. Ribeiro Dantas und D. Beltrão Borges, „Visualizing and Managing Technical Debt in Agile Development: An Experience Report," in *Agile Processes in Software Engineering and Extreme Programming*, 2013.

[167] A. Martini und J. Bosh, „On the interest of architectural technical debt: Uncovering the contagious debt phenomenon," *Journal of Software: Evolution and Process,* 2017.

[168] T. Besker, A. Martini und J. Bosch, „Technical Debt Cripples Software Developer Productivity: A Longitudinal Study on Developers' Daily Software Development Work," in *TechDebt '18: Proceedings of the 2018 International Conference on Technical Debt*, 2018.

[169] E. Lim, T. Nitin und C. Seaman, „A Balancing Act: What Software Practitioners Have to Say about Technical Debt," *IEEE Software,* pp. 22-27, 2012.

[170] Amplify DX, „Analyzing a Project with SonarQube," 2024. [Online]. Available: https://dx.appirio.com/quality-sonarqube/sonarqube-project-analysis/. [Zugriff am 5 März 2024].

[171] J. Holvitie, S. A. Licorish, R. O. Spínola, S. Hyrynsalmi, S. G. MacDonell, T. S. Mendes, J. Buchan und V. Leppänen, „Technical debt and agile software development practices and processes: An industry practitioner survey," *Information and Software Technology,* pp. 141-160, 2018.

[172] K. Eby, „Grundlagen und fortgeschrittene Kenntnisse zur kontinuierlichen Verbesserung: Modelle, Prozesse und Pläne," 2023. [Online]. Available: https://de.smartsheet.com/content/continuous-improvement. [Zugriff am 19 Februar 2024].

[173] L. Chen, „Continuous delivery: Huge benefits, but challenges too," *IEEE Software,* p. 50–54, 2015.

[174] Cloudflare, Inc., „Was ist SaaS? | SaaS-Definition," 2024. [Online]. Available: https://www.cloudflare.com/de-de/learning/cloud/what-is-saas/. [Zugriff am 6 März 2024].

[175] G. McGraw, „Software security," *IEEE Security & Privacy,* pp. 80-83, 2004.

[176] J. Andreae, „Bitkom legt Zahlen vor 206 Milliarden Euro Schaden durch Cyberkriminalität," 2023. [Online]. Available: https://www.tagesschau.de/wirtschaft/cybercrime-deutschland-100.html. [Zugriff am 19 März 2024].

[177] I. Gat und J. D. Heintz, „From assessment to reduction: how cutter consortium helps rein in millions of dollars in technical debt," in *Proceedings of the 2nd Workshop on Managing Technical Debt*, 2011.

[178] M. Couto, D. Maia, J. Saraiva und R. Pereira, „On energy debt: managing consumption on evolving software," in *Proceedings of the 3rd International Conference on Technical Debt*, 2020.

[179] H. M.A.K. und H. Ruqaiya, Cohesion in English, Longman, 1976.

[180] M. J. Eppler, „A Generic Framework for Information Quality in knowledge-intensive Processes," in *Proceedings of the MIT Information Quality 2001 Conference*, 2001.

[181] L. Williams und A. Cockburn, „Agile software development: it's about feedback and change," *Computer,* pp. 39-43, 2003.

[182] E. Di Nitto, P. Jamshidi, M. Guerriero, I. Spais und D. A. Tamburri, „A software architecture framework for quality-aware DevOps," in *QUDOS 2016: Proceedings of the 2nd International Workshop on Quality-Aware DevOps*, 2016.

[183] P. Lipton, D. Palma, M. Rutkowski und D. A. Tamburri, „TOSCA Solves Big Problems in the Cloud and Beyond!," *IEEE Cloud Computing,* pp. 37-47, 2018.

[184] Cloudify Platform Ltd., „What is TOSCA?," 2022. [Online]. Available: https://cloudify.co/what-is-tosca/. [Zugriff am 25 Januar 2024].

[185] Equinix Inc., „API Documentation," 2024. [Online]. Available: https://developer.equinix.com/docs. [Zugriff am 25 Januar 2024].

[186] GitHub, Inc., „DeepSpeech," 2024. [Online]. Available: https://github.com/mozilla/DeepSpeech. [Zugriff am 27 Januar 2024].

[187] J. Klünder, J. Horstmann und O. Karras , „Identifying the Mood of a Software Development Team by Analyzing Text-Based Communication in Chats with Machine Learning," in *International Conference on Human-Centered Software Engineering*, 2020.

[188] K. Väänänen-Vainio-Mattila, V. Roto und M. Hassenzahl, „Towards Practical User Experience Evaluation Methods," in *Meaningful measures: Valid useful user experience measurement (VUUM)*, 2008.

[189] Google, „Material Design," 2024. [Online]. Available: https://m3.material.io/. [Zugriff am 6 Februar 2024].

[190] I. Kangas, M. Schwoerer und L. J. Bernardi, „Recommender Systems for Personalized User Experience: Lessons learned at Booking.com," in *Proceedings of the 15th ACM Conference on Recommender Systems*, 2021.

[191] F. Van de Sand, A.-K. Frison, P. Zotz, A. Riener und K. Holl, „The Intersection of User Experience (UX), Customer Experience (CX), and Brand Experience (BX)," in *User Experience Is Brand Experience*, Springer, 2020, p. 71–93.

[192] S. Deterding, M. Sicart, L. Nacke, K. O'Hara und D. Dixon, „Gamification. using game-design elements in non-gaming contexts," in *CHI '11 Extended Abstracts on Human Factors in Computing Systems*, 2011.

[193] N. Karamanis, M. Pignatelli, D. Carvalho-Silva, F. Rowland, J. A. Cham und I. Dunham, „Designing an intuitive web application for drug discovery scientists," *Drug Discovery Today,* pp. 1169-1174, 2018.

[194] M. Fraculj, L. Lekaj und L. Kondić, „RESEARCH ON ATTITUDES TOWARD MINIMALISTIC DESIGN IN MARKETING COMMUNICATIONS,“ *International journal of multidisciplinarity in business and science,* pp. 5-14, 2023.

[195] Mind Tools Content Team, „Impact Analysis,“ 2023. [Online]. Available: https://www.mindtools.com/axt4kh3/impact-analysis. [Zugriff am 29 Januar 2024].

[196] H. Son, „How To Create A Test Plan (Steps, Examples, & Template),“ 2023. [Online]. Available: https://www.testrail.com/blog/create-a-test-plan/. [Zugriff am 29 Januar 2024].

[197] CheatSheets Series Team, „Input Validation Cheat Sheet,“ 2024. [Online]. Available: https://cheatsheetseries.owasp.org/cheatsheets/Input_Validation_Cheat_Sheet.html. [Zugriff am 4 Februar 2024].

## Abbildungsverzeichnis

*Abbildung 1: Konzeptuelles Modell für TD [12]* ........ *8*
*Abbildung 2: Überblick über das hybride Entwicklungsmodell für Startups [38]* ........ *13*
*Abbildung 3: Detaillierte Ansicht des Stages FRGE [38]* ........ *14*
*Abbildung 4: Detaillierte Ansicht des zweiten Stages DUXE [38]* ........ *15*
*Abbildung 5: Detaillierte Ansicht des dritten Stages PLT [38]* ........ *16*
*Abbildung 6: Überblick über die Phasen des Startup-Modells, die Artefakte und den Fluss der Artefakte in Bezug auf die wichtigsten Entwicklungsmethoden [38]* ........ *18*
*Abbildung 7: Chronologischer Forschungsprozess* ........ *21*
*Abbildung 8: Probabilistisches Ursache-Wirkungs-Diagramm für Design Debt [47]* ........ *26*
*Abbildung 9: Durchführung von Ursachen- und Effekt-Analysemeetings [47]* ........ *32*
*Abbildung 10: Design Debt Cost Benefit Matrix für Gottklassen in einem Projekt [68]* ........ *33*
*Abbildung 11: Der Rot-Grün-Refactor Zyklus des Test Driven Developments [38]* ........ *36*
*Abbildung 12 (links): Warnung im Falle zweier ähnlicher Pull Requests [100]* ........ *42*
*Abbildung 13 (rechts): Warnung im Falle von ähnlichem Code zu einer anderen Fork, Pull Request, usw. [100]* *42*
*Abbildung 14: MVUX-Framework zur Unterstützung der MVP-Entwicklung in Startups [132]* ........ *51*
*Abbildung 15: Legende des erweiterten Entrepreneurial Software Engineering Modells* ........ *62*
*Abbildung 16: Feature Requirements Generation & Evaluation - Technical Debt Avoidance (FRGE-TDA) Phase, aufbauend auf [38]* ........ *63*
*Abbildung 17: Design & User Experience Evaluation - Technical Debt Prevention (DUXE-TDP), aufbauend auf [38]* ........ *66*
*Abbildung 18: Pre-Launch Testing – Technical Debt Identification (PLT-TDI), aufbauend auf [38]* ........ *68*
*Abbildung 19: Technical Debt Categorization & Prioritization (TDCP), aufbauend auf [38]* ........ *69*
*Abbildung 20: Development Cycle, aufbauend auf [38]* ........ *72*
*Abbildung 21: Übersicht des erweiterten ESEMs, aufbauend auf [38]* ........ *75*
*Abbildung 22: Probabilistisches Ursache-Wirkungs-Diagramm für Design Debt [47]* ........ *85*
*Abbildung 23: Durchführung von Ursachen- und Effekt-Analysemeetings [47]* ........ *86*
*Abbildung 24: Design Debt Cost Benefit Matrix für Gottklassen in einem Projekt [68]* ........ *88*
*Abbildung 25: Probabilistisches Ursache-Wirkungs-Diagramm für Code Debt [55]* ........ *89*
*Abbildung 26: Data Klassen in einem Beispielssystem namens Heritrix [71]* ........ *90*
*Abbildung 27: Probabilistisches Ursache-Wirkungs-Diagramm für Test Debt [55]* ........ *91*
*Abbildung 28: Der Rot-Grün-Refactor Zyklus des Test Driven Developments [38]* ........ *92*
*Abbildung 29: Probabilistisches Ursache-Wirkungs-Diagramm für Architecture Debt [55]* ........ *93*
*Abbildung 30: Beispielhafte Systemarchitektur [87]* ........ *94*
*Abbildung 31: Probabilistisches Ursache-Wirkungs-Diagramm für Requirements Debt [55]* ........ *96*
*Abbildung 32: Probabilistisches Ursache-Wirkungs-Diagramm für Documentation Debt [55]* ........ *98*

*Abbildung 33: Probabilistisches Ursache-Wirkungs-Diagramm für Versioning Debt [55] ................................. 100*
*Abbildung 34 (links): Warnung im Falle zweier ähnlicher Pull Requests [100] .................................................. 100*
*Abbildung 35 (rechts): Warnung im Falle von ähnlichem Code zu einer anderen Fork, Pull Request, usw. [100] ........................................................................................................................................... 100*
*Abbildung 36: Präzision und Recall bei verschiedenen Thresholds, wobei die gestrichelte Linie den Default Threshold darstellt. [100] ........................................................................................................... 102*
*Abbildung 37: Probabilistisches Ursache-Wirkungs-Diagramm für Defect Debt [55] ....................................... 102*
*Abbildung 38: Probabilistisches Ursache-Wirkungs-Diagramm für Build Debt [55] .......................................... 105*
*Abbildung 39: Durch Clipper erstellte Liste an Libraries welche einfache Kandidaten für Dependency Refactoring wären [107] ........................................................................................................................... 106*
*Abbildung 40: Graphische Darstellung einer Dependency zwischen zwei Targets in der Clipper Software [107]106*
*Abbildung 41: Probabilistisches Ursache-Wirkungs-Diagramm für Process Debt [55] ...................................... 107*
*Abbildung 42: TOSCA Service Template zur Darstellung einer Systeminfrastruktur [184] ................................ 110*
*Abbildung 43: Probabilistisches Ursache-Wirkungs-Diagramm für Usability Debt [55] .................................... 113*
*Abbildung 44: MVUX-Framework zur Unterstützung der MVP-Entwicklung in Startups [132] .......................... 114*
*Abbildung 45: Probabilistisches Ursache-Wirkungs-Diagramm für Test Automation Debt [55] ........................ 115*
*Abbildung 46: Probabilistisches Ursache-Wirkungs-Diagramm für Service Debt [55] ...................................... 117*
*Abbildung 47: Technische Schulden im Vergleich zum Wert der Optionen der Web Services im Laufe von sechs Monaten [137] ........................................................................................................................... 119*

## Tabellenverzeichnis

*Tabelle 1: Eingesetzte Schlüsselwörter für die Literatursuche ............................................................................ 22*
*Tabelle 2: Beispielhafte Requirement Smells und deren Erkennungsstrategien, in Anlehnung an [91] ................ 39*
*Tabelle 3: Beispielhaftes Definition of Done eines Softwareprojektes, in Anlehnung an [13] .............................. 47*
*Tabelle 4: Zusammenhänge zwischen TD-Typen (dunkelblaue Felder kennzeichnen direkte Zusammenhänge). In Anlehnung an [30] [139] [12] [116] [144] [145] ..................................................................................... 56*
*Tabelle 5: Zusammengefasste Checkliste zum Management von technischen Schulden in Startups ................... 59*
*Tabelle 6: CBM-Werte für geplantes und tatsächliches Design, in Anlehnung an [87] ........................................ 94*
*Tabelle 7: Beispielhafte Requirement Smells und deren Erkennungsstrategien, in Anlehnung an [91] ................ 97*
*Tabelle 8: Beispielhaftes Definition of Done eines Softwareprojektes, in Anlehnung an [13] ............................ 108*
*Tabelle 9: Eigenschaften der analysierten Web Services [137] .......................................................................... 118*