Institut für Softwaretechnologie

Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit

Verwalten der Sicherheitsanforderungen Technischer Schulden in der agilen Entwicklung

Rainer Huß

Studiengang:	Softwaretechnik
Prüfer/in:	Prof. Dr. Stefan Wagner
Betreuer/in:	Yang Wang
Beginn am:	16.05.17
Beendet am:	16.11.17
CR-Nummer:	D.2.1

Kurzfassung

In dieser Arbeit werden die Ursachen von Technischen Schulden und Requirement Technical Debts anhand einer systematischen Literaturrecherche aufgeführt. Darüber hinaus werden die Folgen von Requirement Technical Debts für die Sicherheit in der Softwaretechnik und für die agile Entwicklung dargestellt. Anschließend werden mögliche Maßnahmen für die Prävention von Technischen Schulden aufgezeigt. Das Ziel dieser Arbeit ist es, ein Verständnis gegenüber Technischen Schulden und Requirement Technical Debts zu vermitteln und die möglichen Gefahren aufzuzeigen.

Inhaltsverzeichnis

1 Einleitung **13**

2 Hintergrund **15**
2.1 Requirement Technical Debts 15
2.2 Agile Softwareentwicklung 17
2.3 Der Sicherheitsaspekt in der Softwareentwicklung 20

3 Forschungsmethode (Systematische Literaturrecherche) **23**

4 Research Questions **25**

5 Vorgehensweise der Recherche **27**
5.1 Protokollentwicklung 27
5.2 Ein- und Ausschlusskriterien 27
5.3 Datenquellen und Suchstrategie 28
5.4 Datenquellen und Suchstrategie 31
5.5 Datensammlung 32
5.6 Synthese der Funde 32

6 Qualitätsbewertung der einzelnen Quellen **35**

7 Ergebnisse **37**
7.1 Ursachen von Technischen Schulden 37
7.2 Ursachen von Requirement Technical Debts 42
7.3 Auswirkungen 45
7.3.1 Auswirkungen von Requirement Technical Debt auf die Agile Entwicklung 45
7.3.2 Auswirkungen von Requirement Technical Debts auf die Sicherheit . . . 51
7.4 Maßnahmen gegen Technische Schulden 52

8 Diskussion **61**

9 Zusammenfassung und Ausblick **63**

Abbildungsverzeichnis

2.1 Rubriken von Technischen Schulden 15
2.2 Scrum Verfahren 19
2.3 Faktoren der Sicherheit 21
2.4 Vier Elemente der Anwendungssicherheit 22

5.1 Vorgehen - Technische Schulden 33
5.2 Vorgehen - Requirement Technical Debt 34
5.3 Vorgehen - Maßnahmen 34

7.1 Wasserfallmodell 46
7.2 Technische Schulden und Sicherheit 52

Tabellenverzeichnis

5.1 Ursachen Technischer Schulden – Quellen und Akronym 1 28
5.2 Ursachen Technischer Schulden – Quellen und Akronym 2 29
5.3 Ursachen Technischer Schulden – Quellen und Akronym 3 30

6.1 Qualitätsbewertung der Quellen 1 35
6.2 Qualitätsbewertung der Quellen 2 36

7.1 Ursachen von Technischen Schulden 1 38
7.2 Ursachen von Technischen Schulden 2 39
7.3 Ursachen von Requirement Technical Debts 42
7.4 Gefahreneinstufung der RTD in der agilen Entwicklung 50
7.5 Maßnahmen zur Prävention von Technischen Schulden 53
7.6 Eine Zuordnung der Maßnahmen den Technischen Schulden 1 58
7.7 Eine Zuordnung der Maßnahmen den Technischen Schulden 2 59

Abkürzungsverzeichnis

RTD Requirement Technical Debt. 33

SLR systematische Literaturrecherche. 23

TD Technischen Schulden. 37

1 Einleitung

Technische Schulden spiegeln die technischen Kompromisse wieder, welche in der Softwareentwicklung eingegangen werden. Kompromisse bringen in der Regel einen kurzfristigen Nutzen, schaden der langfristigen Entwicklung [19]. Dabei beziehen sich Technische Schulden nicht nur auf den Programmcode, sondern auch auf die Architektur, die Dokumentation, den Testumfang und die Anforderungen [19]. Technische Schulden in der Anforderungsphase unterscheiden sich von den Schulden, die in der Implementation entstehen.

In der Implementation beziehen sich Technische Schulden auf Entscheidungen, absichtlicher oder unabsichtlicher Weise, welche der Qualität der Software schaden. Dies können Verletzungen des Coding-Standards oder andere Fehlerquellen sein, welche der Codequalität schaden [36].

Requirement Technical Debt beziehen sich auf die Anforderungen in der Softwareentwicklung. Schulden führen hier dazu, dass Kompromisse in der Implementation eingegangen werden müssen.

Technische Schulden sind im Allgemeinen ein Maß, an der sich die Qualität einer Software beschreiben lässt. Sind auch noch im Endprodukt viele Technische Schulden vorhanden, so entspricht die fertige Software nicht der gewünschten Qualität oder den Erwartungen des Kunden. Technische Schulden sollten in der Softwareentwicklung immer behandelt werden. Noch besser ist es, die Entstehung von Technischen Schulden von vorneherein einzuschränken.

Die agile Softwareentwicklung hat seit dem Jahr 2000 erhöhte Aufmerksamkeit erlangt [37]. Laut einer Studie von VersionOne, nutzten im Jahr 2014 bereits 88 Prozent der Softwareunternehmen die Methoden der agilen Entwicklungen [38]. Die agile Softwareentwicklung beschreibt eine Art der Softwareentwicklung, welche durch eine erhöhte Transparenz und Flexibilität charakterisiert ist [37]. Ziel der agilen Softwareentwicklung ist es, Änderungen schnell umsetzen zu können und einen konstanten Arbeitsfluss ohne zu viel Bürokratie zu ermöglichen [0].

Bei dieser Arbeit handelt es sich um eine systematische Literaturrecherche. Eine systematische Literaturrecherche fasst die Ergebnisse mehrerer Publikationen zusammen und bietet einen Übersicht über den Diskussionsstand zu einem bestimmten Forschungsthema. Die Recherche erfolgt hierbei in einer streng strukturierten und nachvollziehbaren Arbeitsweise. Die systematische Literaturreche kann für weitere Forschungen bezüglich des Themas verwendet werden. In dieser Arbeit wurden 43 Quellen auf die Ursachen und Prävention von Technischen Schulden untersucht. Es wurden nur Quellen verwendet welche sich explizit mit den Technischen Schulden befasst haben.

Zunächst werden die Ursachen von Technischen Schulden herrausgearbeitet. Anschließend wird auf die Ursachen von Requirement Technical Debt eingegangen. Die gefundenen Ursachen werden in ihrem Bezug zur agilen Entwicklung und zur Sicherheit in der Softwareentwicklung gewertet. Im Anschluss daran werden die aus den Quellen extrahierten Präventionsmaßnahmen vorgestellt. Dabei werden diese im Einzelnen erklärt und den einzelnen Ursachen zugeordnet.

In der abschließenden Diskussion wird auf die Möglichkeiten weiterführender Forschungen eingegangen. Die vorliegende Arbeit verfolgt damit folgende Ziele:

1. Ein Verständnis von Technischen Schulden und Requirement Technical Debt zu vermitteln
2. Eine Übersicht über die häufigsten Ursachen von Technischen Schulden und Requirement Technical Debts zu geben
3. Die Folgen von Requirement Technical Debts für die Sicherheit in der Softwareentwicklung und für die agile Entwicklung darzustellen
4. Möglichen Maßnahmen für die Prävention von Technischen Schulden aufzuzeigen

Diese Arbeit liefert keine wissenschaftlich gestützte Bewertung über die Häufigkeit und die Gefahr der einzelnen Ursachen in der Softwareentwicklung. Zudem können, aufgrund der hohen Anzahl nicht alle Ursachen von Technischen Schulden vorgestellt werden.

Die folgenden acht Kapitel sind wie folgt strukturiert: Kapitel 2 geht auf die Begriffe Requirement Technical Debts, agile Entwicklung und den Sicherheitsaspekt in der Softwareentwicklung ein. In Kapitel 3 wird die Forschungsmethode dieser Arbeit vorgestellt. Die Forschungsfragen wer-den in Kapitel 4 vorgestellt. Kapitel 5 zeigt die Vorgehensweise der systematischen Literatur-recherche und beschreibt die Qualitätskriterien der Suche. In 6 Kapitel wird die Qualität der einzelnen Quellen bewertet. Kapitel 7 präsentiert die Ergebnisse der extrahierten Daten aus der systematischen Literaturrecherche und beantwortet die Forschungsfragen. Im Kapitel 8 wird die Validität dieser Arbeit diskutiert und ein Vorschlag für weitere Forschungen geliefert. Zuletzt werden im 9. Kapitel die Ergebnisse zusammengefasst.

2 Hintergrund

2.1 Requirement Technical Debts

Schon im Jahr 1992 bezeichnete Ward Cunningham Technische Schulden als unfertigen oder fehlerhaften Programmcode, welcher in Zukunft noch verbessert werden muss [8] [9]. Technische Schulden entstehen durch bewusste oder unbewusste falsche oder suboptimale Entscheidungen [41]. Führt eine Entscheidung zu einem späteren Zeitpunkt zu Mehraufwand, so wird dabei von einer Technischen Schuld gesprochen [41]. Eine Technische Schuld ist die Konsequenz die aus einer schlechten technischen Umsetzung von Software entsteht. Während sich Technische Schulden zunächst nur auf den Programmcode bezogen, wird der Begriff heute für verschiedene Arten von Schulden verwendet [9]:

- Design-Schulden
- Architektur-Schulden
- Test-Schulden
- Wartbarkeits-Schulden

Eine einheitlich definierte Struktur Technischer Schulden gibt es zur Zeit nicht. Neil. A Ernst gliedert Technische Schulden in folgende drei Hauptbereiche ein [10]:

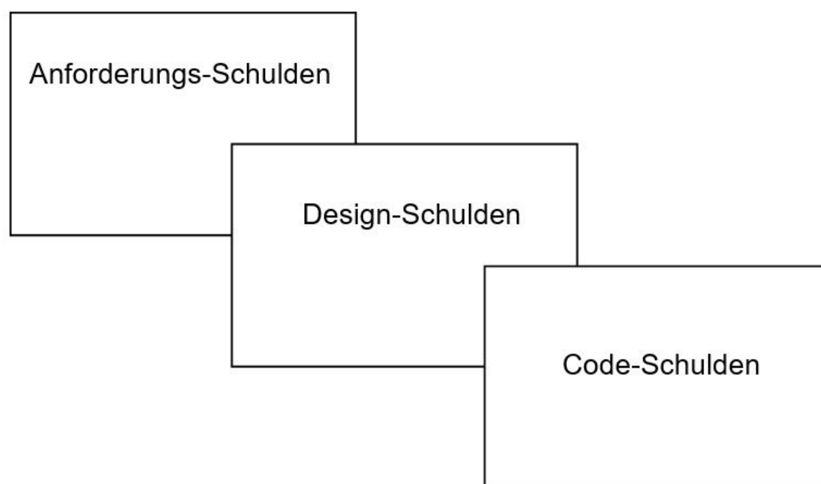

Abbildung 2.1: Rubriken von Technischen Schulden

Wie anhand der Abbildung zu sehen ist, kann es zu Überschneidungen zwischen den einzelnen Rubriken von Technischen Schulden geben. Grund dafür ist, dass die einzelnen Rubriken keine streng definierten Grenzen und Definitionen besitzen. Diese Arbeit befasst sich vorrangig mit Anforderungs-Schulden oder auch „Requirement Technical Debts“. Die Requirement Technical Debts sind eine Untergruppe der Technischen Schulden. Aufgrund der geringen Anzahl an Veröffentlichungen, welche sich mit den Requirement Technical Debts beschäftigen, ist man sich bis heute nicht einig, ob Requirements Debts überhaupt als eigene Rubrik Technischer Schulden angesehen werden können [10] [11]. Betrachten wir die folgenden zwei Definitionen des Requirement Technical Debt:

“tradeoffs made with respect to what requirements the development team need to implement or how to implement them” - N. Alves et al.[11]

“tradeoffs made with respect to what requirements the development team need to implement or how to implement them” - Neil Ernst [11]

Zusammengefasst sind Requirement Technical Debts die Kompromisse bezüglich der Anforderungsspezifikation. Diese haben Auswirkungen auf die spätere Systemimplementierung, beziehungsweise auf das fertige Produkt. Durch die Kompromisse entsteht ein Abstand zwischen dem aktuellen und dem optimalen Produkt. Doch kann auch auf der Grundlage dieser Definitionen die Zuordnung Requirement Technical Debts zu den Technischen Schulden nicht abschließend entschieden werden.

Die Anforderungsspezifikation steht am Anfang einer Entwicklung. Die Anforderungen beschreiben die Funktionen und Eigenschaften, die ein Kunde von einem System erwartet [42]. Es wird zwischen verschiedenen Arten von Anforderungen unterschieden.

1. Funktionale Anforderungen sind die Funktionen die ein System erfüllen muss [43].
2. Nicht-Funktionale Anforderungen geben an, wie ein System oder eine einzelne Funktion arbeitet. Dazu gehören Qualitätsanforderungen, wie der Performanz und der Zuverlässigkeit [42].
3. Eine Systemanforderung ist eine Systembeschreibung aus technischer Sicht. Systemanforderungen bestehen aus funktionalen, nicht funktionalen und internen Anforderungen [43]. Dazu gehören:
   a) Sicherheit
   b) Korrektheit
   c) Vertrauenswürdigkeit
   d) Funktionsfähigkeit
   e) Verfügbarkeit
   f) Testbarkeit
   g) Verständlichkeit
   h) Wartbarkeit
   i) Wiederverwendbarkeit

Der Prozess der Anforderungsanalyse besteht aus folgenden Punkten [42] [43]:

- Herausfindens
- Analysierens
- Dokumentierens
- Überprüfens
- Management

Anhand der Anforderungsspezifikation ist der Umfang der Requirement Technical Debt zu erkennen. Requirement Technical Debt umfassen neben einer reinen Anforderungssammlung auch die Planung der Entwicklung. Zu dieser Planung können die Erstellung einer Architektur und die sozialen Kompetenzen der Mitarbeiter miteingeschlossen werden. Anhand der Systemanforderungen ist zu erkennen, dass die Erstellung von Testfällen notwendig ist, um die einzelnen Anforderungen zu überprüfen. Die Durchführung von Tests sollte daher auch in den An-forderungen miteingeschlossen werden. Ursachen von Technischen Schulden, welche bereits in der Anforderungsphase behandelt werden können, werden den Requirement Technical Debts zugeordnet. Die Implementation des Systems gehört nicht zu den Requirement Technical Debts.

2.2 Agile Softwareentwicklung

Der Begriff „agile Softwareentwicklung“ entstand um das Jahr 2000 [1]. Erste Ansätze waren schon in den frühen 1990er-Jahren zu finden [37]. Im Jahr 1999 veröffentlichte der Informatiker Kent Beck sein Buch „Extreme Programming“, was als die Grundlage agiler Softwareentwicklung gesehen werden kann [1]. Von Kent Beck und seinem Forschungsteam stammt das sogenannte „Manifest für die agile Softwareentwicklung“, welches folgend zitiert ist:

“Wir erschließen bessere Wege, Software zu entwickeln, indem wir es selbst tun und anderen dabei helfen. Durch diese Tätigkeit haben wir diese Werte zu schätzen gelernt: Individuen und Interaktionen mehr als Prozesse und Werkzeuge Funktionierende Software mehr als umfassende Dokumentation Zusammenarbeit mit dem Kunden mehr als Vertragsverhandlung Reagieren auf Veränderung mehr als das Befolgen eines Plans Das heißt, obwohl wir die Werte auf der rechten Seite wichtig finden, schätzen wir die Werte auf der linken Seite höher ein. ” [2]

In der agilen Entwicklung werden Einzelpersonen und Interaktionen höher bewertet, als Prozesse und Werkzeuge [37]. Die Zusammenarbeit untereinander soll während der Entwicklungsphase immer vorhanden sein. Somit spielt die Kommunikation eine wichtige Rolle. In der agilen Entwicklung muss eindeutig definiert werden, was als funktionierende Software angesehen werden kann. Dem Kunden sollte in regelmäßigen Abständen Abschnitte einer funktionierenden Software geliefert werden [37]. Eine Zusammenarbeit mit dem Kunden ist aus diesem Grund nicht wegzudenken. Anforderungen können sich während der Entwicklung ändern. Um den Kunden zufrieden zu stellen, wird das Reagieren auf Änderungen höher als das Befolgen eines Plans gestellt.

Die Werte des Manifests zeigen einen Unterschied zu der herkömmlichen traditionellen Softwareentwicklung. Im Vordergrund stehen in der agilen Entwicklung die Funktionalität und der

Mensch [37]. Bürokratische Aspekte wie Verträge oder Pläne werden in den Hintergrund gerückt. Um diese Werte einzuhalten, wurden zwölf Prinzipien [2] formuliert, die bei der agilen Entwicklung befolgt werden sollen [37]. Um ein besseres Verständnis gegenüber der agilen Entwicklung zu bekommen, werden die Prinzipien im folgenden aufgelistet [2] [37]:

1. Stelle den Kunden durch eine frühzeitige und kontinuierliche Auslieferung zufrieden
2. Heiße Anforderungsänderungen jederzeit willkommen. Agile Prozesse nutzen Veränderungen zum Wettbewerbsvorteil des Kunden
3. Dem Kunden soll innerhalb kurzer Zeitspannen eine funktionierende Software ausgelie-fert werden
4. Eine Zusammenarbeit zwischen Entwicklern und Führungskräften sollte immer vorhan-den sein
5. Schaffe ein gutes und freundliches Arbeitsumfeld. Einzelne Personen sollen motiviert und unterstützt werden
6. Die effizienteste und effektivste Methode, Informationen an und innerhalb eines Entwicklungsteams zu übermitteln, ist im Gespräch von Angesicht zu Angesicht
7. Eine funktionierende Software ist das wichtigste Fortschrittsmaß
8. Agile Prozesse fördern nachhaltige Entwicklung. Eine Überarbeitung der Mitarbeiter soll verhindert werden. Die Auftraggeber, Entwickler und Benutzer sollten ein gleichmäßiges Tempo auf unbegrenzte Zeit halten können
9. Ständiges Augenmerk auf technische Exzellenz und gutes Design fördert Agilität.
10. Einfachheit die Kunst die Menge der Arbeit, die nicht getan wird, zu maximieren - ist ent-scheidend. Die besten Architekturen, Anforderungen und Entwürfe entstehen durch selbstorganisierte Teams
11. In regelmäßigen Abständen reflektiert das Team, wie es effektiver werden kann und passt sein Verhalten entsprechend an

Anhand der Prinzipien ist zu erkennen, dass es in der agilen Softwareentwicklung das höchste Ziel ist, die Transparenz und die Flexibilität zu erhöhen. Anforderungsänderungen sollen, selbst im späten Abschnitt der Entwicklung, zu keinem Problem führen [37]. Oft wird hierbei auf die iterative und parallele Entwicklung zurückgegriffen, wobei dem Kunden verschiedene Versionen der Software zur Verfügung gelegt werden können. Weitere Besonderheiten der agilen Entwicklung sind kleine selbstorganisierende Teams, kontinuierliche Designverbesserungen, eine testgetriebene Entwicklung und einer kontinuierlichen Integration [3]. Das Manifest bildet das Fundament der agilen Entwicklung. Die agilen Prinzipien bilden die Handlungsgrundsätze.

Obwohl es derzeit schon viele verschiedene Versuche zur Umsetzung agiler Methoden gibt sind, ist deren Entwicklung nach wie vor nicht abgeschlossen [3]. Eine der weitverbreitetsten agilen Methoden ist Scrum [1].

Das Scrum-Verfahren kennt drei Rollen. Zum einen gibt es den „Product Owner“, welcher mit dem Kunden gleichzusetzen ist. Er stellt die Anforderungen und priorisiert diese. Der „Scrum Master“ verwaltet den Arbeitsprozess und beseitigt Hindernisse. Die dritte Rolle besteht aus dem

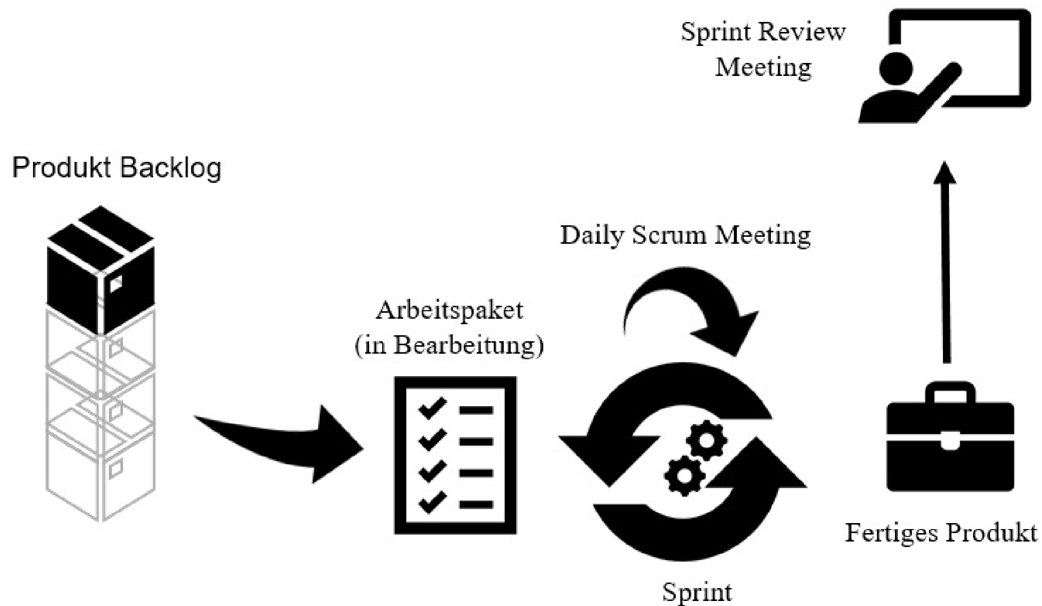

Abbildung 2.2: Scrum Verfahren

Programmierteam selber, die das Produkt entwickeln. [4] Neben diesen drei Rollen kann es noch Beobachter und Ratgeber, sogenannte „Stakeholder“ geben. [4]

Im Produkt Backlog werden die Anforderungen in einer Liste gesammelt und priorisiert. Das Produkt Backlog kann sich ständig ändern. Der Product Owner erstellt in der Zusammenarbeit mit dem Team Arbeitspakete. Die höher priorisierten Pakete werden oben abgelegt. Das Entwicklerteam nimmt sich das oberste Arbeitspaket und hat dann zum Beispiel einen Monat Zeit, die Anforderungen umzusetzen. Das Arbeitspaket wird während dem bearbeiten nicht verändert oder durch zusätzliche Anforderungen ergänzt. Das Backlog kann von dem Produkt Owner je-derzeit verändert oder neu priorisiert werden. [4]

Das in Bearbeitung stehende Arbeitspaket kann in kleinere Arbeitspakete aufgeteilt und an mehrere Entwickler verteilt werden. Die Entwickler haben die Pflicht, täglich den aktualisierten Restaufwand in einer Liste festzuhalten. Der Vorgang, in dem das Entwicklerteam vollkommen auf die Umsetzung seiner Aufgaben konzentriert ist, wird Sprint genannt. Ziel ist es, einen vollständig einsetzbaren Anwendungsteil zu produzieren. Um die geleistete Arbeit zu überblicken, wird jeden Tag ein sogenanntes Daily Scrum Meeting gehalten. In diesen Meetings teilt jeder Entwickler dem Team mit, an was er zuletzt gearbeitet hat, was er als nächstes erledigen wird und welche Probleme aufteten. Am Ende der Bearbeitung eines Arbeitspakets präsentiert das Team dem Produkt Owner und dem Stakeholder die Ergebnisse. Dieser Vorgang wird Sprint Review Meeting genannt. Das Feedback und die neuen Anforderungen fließen in den nächsten Sprint mit ein und der Prozess beginnt von vorne. [4]

Die Eigenschaften, welche nach der obigen Auflistung zu erkennen sind, geben ein Bild über das Vorgehen in der agilen Entwicklung. Die Prinzipien werden dazu genutzt um einen Bezug auf die Ursachen von Technischen Schulden zu nehmen.

2.3 Der Sicherheitsaspekt in der Softwareentwicklung

Vor allem Unternehmensanwendungen erfordern ein hohes Sicherheitsniveau. Webanwendungen müssen in der heutigen Zeit sehr gut gegen Webangriffe geschützt sein. Um einen Schutz zu gewährleisten, ist es wichtig, nicht erst am Ende, sondern während des gesamten Entwicklungsprozesses der Anwendung eine Sicherheit zu gewährleisten [40]. Schon in der Anwendungsarchitektur können Entscheidungen getroffen werden, welche Auswirkungen auf die spätere Sicherheit haben [40]. Aus diesem Grund ist es wichtig den Sicherheitsaspekt schon von Beginn an zu berücksichtigen.

Sicherheit wird oftmals verschieden aufgefasst. Die im folgenden verwendete Definition der Sicherheit bezieht sich auf das Journal „Engineering Safety Requirements, Safety Constraints, and Safety-Critical Requirements“ von Donald Firesmith des Software Institutes of Software Engine-ering (U.S.A.) [5]

Dort wird die Sicherheit als Teilaspekt der Abwehr beschrieben, welche wiederrum ein Teilaspekt der Abhängigkeit ist. Die Abhängigkeit ist der Grad, in dem der Nutzer von der Software abhängig sein kann. Diese besteht aus der Betriebsverfügbarkeit, Abwehr, Zuverlässigkeit und Robustheit. Die Abwehr beschreibt den Grad, in welcher Software vor Unfällen oder Attacken geschützt ist (Abbildung 2.3). [5] Die Sicherheit beschäftigt sich auch damit, Probleme richtig zu adressieren, zu identifizieren oder abzufangen.

Ein System gilt als unsicher, wenn es nicht akzeptierbare Verluste verursacht. Verluste entstehen durch Unfälle, durch unbeabsichtigte Ereignisketten, die zu Verlusten führen. Diese Verluste können Verletzungen, Schäden an Sachgütern und der Umwelt oder ein Verlust von Leben sein. [7] Die funktionale Sicherheit wird somit erheblich vom Entwicklungsprozess beeinflusst. Die Spezifikation, das Design, die Implementation, die Validierung, die Konfiguration und die Integration sind ausschlaggebend für eine sichere Software. [6] Jede Person, die an der Systementwicklung beteiligt ist, ist für die Sicherheit verantwortlich [7]. Ein System wird als sicher angesehen, wenn es keine Fehlerzustände erreichen kann [6]. Die Zunahme der Schwachstellen in der Software führt zu einem Anstieg der Risiken [39].

Um für das Endprodukt eine gewisse Sicherheit zu gewährleisten, müssen in der Softwareentwicklung entsprechende Maßnahmen durchgeführt werden. Bei diesen Maßnahmen kann es sich beispielsweise um Unit Tests, Entwurf von Sicherheitsanforderungen und der Sicherheitsarchitektur handeln. Auch gewisse Coding Standards können eine umfangreichere Sicherheit ge-währleisten [39]. Die ISO/IEC 13335-1:2004, Information technology – Security techniques – Management of information and communications technology security, bietet einen Ansatz für die Sicherheitsbewahrung in der Softwareentwicklung.

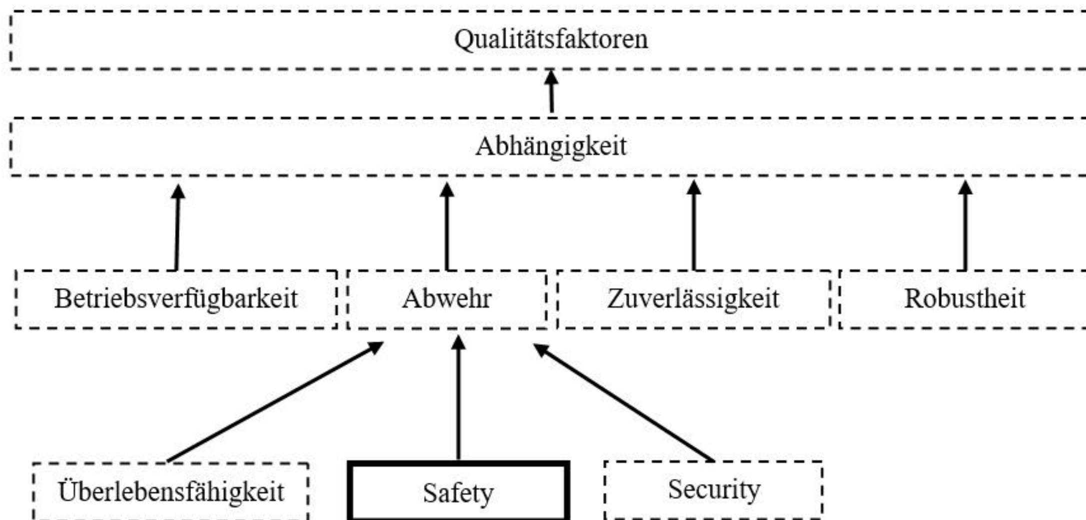

Abbildung 2.3: Faktoren der Sicherheit

Mögliche Gründe für die Entstehung von Schwachstellen in der Sicherheit sind [39]:

- Unklare Sicherheitsanforderungen
- Unvollständige Vorgehensmodelle
- Fehlende Lebenszyklusorientierung
- Unzureichende Standardisierung

Laut dem Diplom-Medieninformatiker und Experte für IT Sicherheit Matthias Rohr erfordert jede Maßnahme zur Verankerung von Sicherheit vier Ebenen (Abbildung 2.4) [40]. Das Management muss schon von Beginn an Pläne entwickeln um die Sicherheit in einem System zu prüfen. Dazu gehört die Einteilung von zuständigen Personen. Die Personen, aber auch die Entwickler müssen sich an Vorgaben wie Coding-Richtlinien halten. Gleichzeitig können Tools verwendet werden um die Sicherheit zu überprüfen. Dies funktioniert allerdings nur mit dem richtigen Know-How der Entwickler. Schulungen sollten ein Verständnis gegenüber der Sicherheit und der Prävention von Gefährdungen vermitteln.

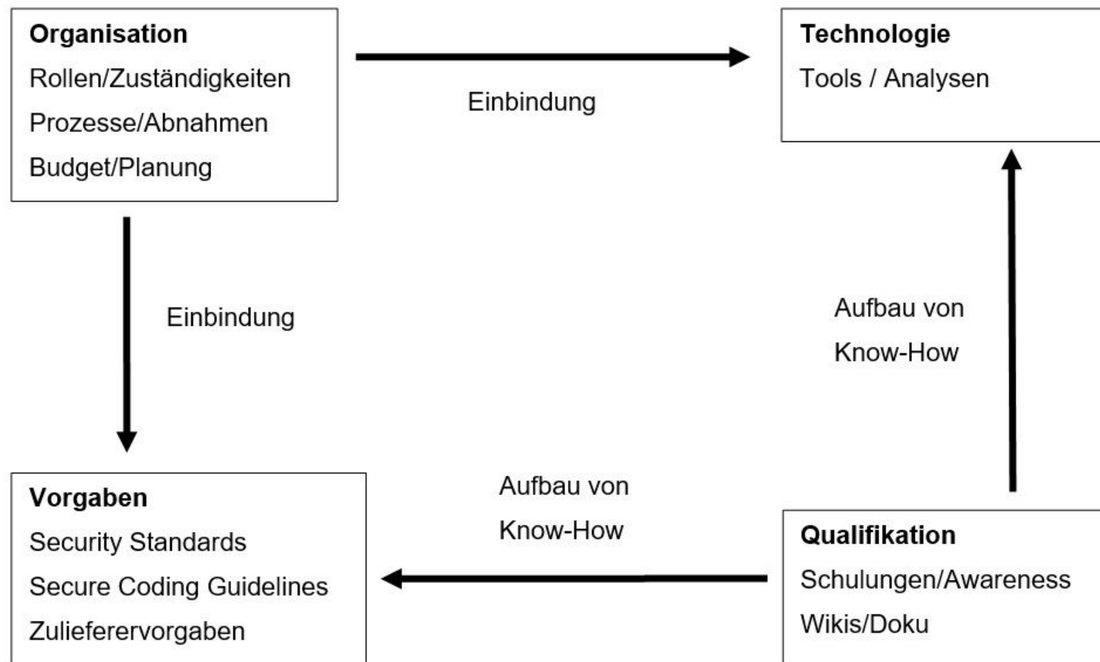

Abbildung 2.4: Vier Elemente der Anwendungssicherheit

3 Forschungsmethode (Systematische Literaturrecherche)

In dieser Arbeit handelt es sich um eine systematische Literaturrecherche. Eine systematische Literaturrecherche (SLR) ist ein Mittel zur Bewertung und Interpretation vieler verfügbaren Forschungen um eine bestimmte Fragestellung zu beantworten. Das Ziel einer SLR ist die Präsentation einer objektiven und fairen Bewertung eines Forschungsthemas unter Verwendung einer vertrauenswürdigen und überprüfbaren Methodik. [20] Diese Arbeit orientiert sich an das „Guidelines for Performing Systematic Literature Reviews in Software Engineering“ der Software Engineering Group der Keele University (UK) und Department of Computer Science der University of Durham (UK).

Eine SLR dient folgenden Zwecken [20]:

- Zusammenfassung vorhandener Belege bezüglich eines Themas
- Identifikation von Lücken in der aktuellen Forschung
- Schaffen eines Rahmens/Hintergrunds für neue Forschungstätigkeiten
- Stützung oder Widerlegung bestehender Hypothesen

Und besteht aus drei Phasen:

1. Planung des Reviews
2. Durchführung des Reviews
3. Berichterstattung

Die Planung des Reviews besteht aus der Erstellung von sogenannten „Research Questions“, den Forschungsfragen. Die Research Questions bilden den wichtigsten Schritt einer SLR. Sie zeigen, auf welche Publikationen sich diese Arbeit bezieht, welche Daten für die Fragestellung extrahiert werden und wie die Daten analysiert werden müssen um die Fragen zu beantworten.

Des Weiteren müssen die Ein- und Ausschlusskriterien geklärt werden. Ein- und Ausschlusskriterien bestimmen, ob Daten mitgezählt oder aus der Suche ausgeschlossen werden. Die Suche der Publikationen muss präzise geplant werden. So muss im Voraus geklärt werden, auf welche Suchbegriffe sich die Suche beschränken soll und ob damit die relevanten Publikationen auch wirklich abgedeckt werden. [20]

Für die Suche müssen Qualitätskriterien erstellt werden. Qualitätskriterien bestimmen ob eine gefundene Studie in die Arbeit aufgenommen oder ausgeschlossen wird. Es muss eine Strategie entwickelt werden um an bestimmte Daten zu gelangen. [20]

Während der Durchführung des Reviews, wird anhand einer Suchstrategie nach so vielen Quellen wie nötig gesucht. Eine größere Sammlung an Quellen bietet auch ein robusteres Ergebnis, ist allerdings mit einem erhöhten Aufwand verbunden. So werden SLR oftmals von mehreren Personen durchgeführt. Die einzelnen Quellen und deren Herkunft müssen angegeben werden und eine Qualitätskontrolle durchlaufen, in der nach bestimmten Kriterien geprüft wird, ob sich die aufgestellten Qualitätsstandard erfüllt. Die Datenextraktion/Datensynthese der gesammelten Quellen muss schließlich in Form eines Berichts zusammenfasst werden. In einem letzten Schritt werden die Ergebnisse des SLR in einer übersichtlichen Weise präsentiert. [20]

Anhand eines Review-Protokolls erfährt der Leser wie die SLR durchgeführt wurde. Ein Review Protokoll- spezifiziert die Methoden, welche genutzt werden, um eine SLR durchzuführen. Am Ende muss dem Leser verständlich sein, wie die Recherche durchgeführt wurde. [20]

Die Methodik der Zusammenfassung vieler Studien verringert zwar die Wahrscheinlichkeit fehlerhafter Ergebnisse, ist jedoch kein absoluter Schutz vor fehlerhaften Resultaten. Mit SLR können breit gefächerte Themenbereiche untersucht werden und robuste Ergebnisse präsentiert werden. Der Nachteil an SLR ist der erhöhte Aufwand gegenüber einer herkömmlichen Literaturrecherche.

4 Research Questions

Die vorliegende Arbeit lässt sich grob in drei Abschnitte gliedern. Der erste Abschnitt dieser Arbeit beschäftigt sich mit der Frage, welche Ursachen zu Technischen Schulden in der Softwareentwicklung führen. Dabei wurde versucht, anhand der SLR möglichst viele Ursachen von Technischen Schulden aus verschiedenen Quellen zu extrahieren und tabellarisch aufzulisten. Der zweite Abschnitt dieser Arbeit besteht darin, aus den genutzten Quellen, die Ursachen von Requirement Technical Debts zu extrahieren. Der dritte Abschnitt dieser Arbeit besteht darin, anhand der Quellen, Möglichkeiten zu finden um mit den Ursachen von Technischen Schulden umzugehen. Zusammengefasst behandelt diese Arbeit folgende Forschungsfragen:

1. Was sind die Ursachen von Technischen Schulden in der Softwareentwicklung?
2. Was sind die Ursachen von Requirement Technical Debts in der Softwareentwicklung und in welchem Bezug stehen diese zur Sicherheit und der agilen Entwicklung?
3. Welche Präventionsmaßnahmen sind bezüglich Technischer Schulden möglich?

5 Vorgehensweise der Recherche

5.1 Protokollentwicklung

Der Aufbau des Protokolls erfolgt gemäß dem „Systematic literature review in software engineering – A systematic literature review“ von Barbara Kitchenham, O. Pearl Brereton, David Budgen, Mark Turner, John Bailey und Stephen Linkman.

5.2 Ein- und Ausschlusskriterien

In der Studie werden Quellen ab dem Jahr 2010, welche sich mit dem Thema der Ursachen von Technischen Schulden in der Softwareentwicklung beschäftigen, verwendet. Dazu gehören:

1. Systematische Literaturrecherchen mit Umfragen, definierten Forschungsfragen, einem definierten Suchvorgang
2. Fachliche qualifizierte Präsentationen
3. Metaanalysen
4. Universitätsvorträge von Professoren, welche sich auf eine wissenschaftliche Basis stützen
5. Literaturen bezüglich des Themas

Eingeschlossen wurden außerdem Artikel, welche nur Teile des hier betrachteten Gebiets abdecken.

Quellen, aus welchen sich nicht erschließen lässt, woher die Informationen stammen und wie gründlich sich mit der Thematik befasst wurde, wurden aus der Datensammlung ausgeschlossen.

Folgende Onlinedatenbanken wurden zur Suche verwendet:

- Google
- Google Scholar
- ACM Digital Library
- IEEE XPLORE
- Springer Link
- Elsevier Science Direct

Folgende Suchbegriffe wurden bei der Suche verwendet:

- Requirements Technical Debt AND Requirement Debt
- Requirements Technical Debt AND Softwaredevelopment
- Technical debt AND Agile Softwaredevelopment
- Safety AND Requirement AND Technical Debt
- Anforderungs-Technische Schulden AND Anforderungsschulden
- Anforderungs-Technische Schulden AND Softwareentwicklung
- Technische Schulden AND Agile Softwareentwicklung
- Sicherheit AND Anforderung AND Technische Schulden

5.3 Datenquellen und Suchstrategie

Der Suchprozess bestand aus einer manuellen Suche nach Quellen ab dem Jahr 2010. Als Suchmaschine wurde primär Google und Google Scholar verwendet. Es wurde sich auf die zuvor genannten Suchbegriffe beschränkt. Da das primäre Ziel dieser Arbeit ist, möglichst viele Ursachen Technischer Schulden herauszuarbeiten, wurde eine Quellenanzahl von über 40 Quellen festgelegt. Die Suchergebnisse wurden nacheinander von Beginn an abgearbeitet. Die ausgewählten Quellen sind in der Tabelle 5.1, 5.2 und 5.3 aufgelistet und wurden mit einem Akronym versehen.

Quellen, Literaturrecherchen, Beiträge	Akronym
Woubshet Nema Behutiye, Pilar Rodriguez, Markku Oivo, Ayse Tosun. "Analyzing the concept of technical debt in the context of agile software development: A systematic literature review". In: Information and Software Technology, Elsevier (2016)	S1
Girish Suryanarayana, Ganesh Samarthyam, Tushar Sharma. "Refactoring for Software Design Smells – Managing Technical Debt". In: Morgan Kaufmann (2014)	P1
Nicolli S.R. Alvesb, Thiago S. Mendesa, d, Manoel G. de Mendonçaa, Rodrigo O. Spínolaa, b, Forrest Shulle, Carolyn Seamanc. "Identification and management of technical debt: A systematic mapping study". In: Information and Software Technology, Elsevier (2015)	S2
Zengyang Li, Paris Avgeriou, Peng Liang. "A systematic mapping study on technical debt and its management". In: Journal of Systems and Software, ACM DigitalLibrary (2015),	S3
Neil A. Ernst, Stephany Bellomo, Ipek Ozkaya, Robert L. Nord, and Ian Gorton, "Measure It? Manage It? Ignore It? Software Practitioners and Technical Debt". 2015. In: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering Pages 50-60, ACM Digital Library	S4
Neil Ernst, University of British Columbia. "Technical Debt and Requirements". 2012. In: Carnegie Mellon University Libraries	P2
Alan MacCormack, Daniel J. Sturtevant. "Technical debt and system architecture: The impact of coupling on defect-related activity". 2016. In: The Journal of Systems and Software, Elsevier	S5
Bill Curtis, Jay Sappidi, Alexandra Szynkarski. "Estimating the Size, Cost, and Types of Technical Debt". 2012. In: Conference: Managing Technical Debt (MTD), 2012 Third International Workshop on	S6

Tabelle 5.1: Ursachen Technischer Schulden – Quellen und Akronym 1

Quelle	Akronym
Dr. Nico Zazworka, Fraunhofer Center for Experimental Software Engineering, Dr. Carolyn Seaman, University of Maryland Baltimore. “Identifying and Managing Technical Debt”. 2012. In: https://de.slideshare.net/zazworka/identifying-and-managing-technical-debt, (visited: 02.08.17)	P3
Philippe Kruchten, Robert L. Nord, Ipek Ozkaya, “Technical Debt: From Metaphor to Theory and Practice”. 2012. In: IEEE Software Volume: 29, Issue: 6, Nov.-Dec. 2012. IEEE	S7
Antonio Martini, Jan Bosch, Michel Chaudron. "Architecture Technical Debt: Understanding Causes and a Qualitative Model”. 2014. In: Software Engineering and Advanced Applications (SEAA), 2014 40th EUROMICRO Conference on. IEEE	S8
Erin Lim, Nitin Taksande, Carolyn Seaman. “A Balancing Act: What Software Practitioners Have to Say about Technical Debt”. 2012. In: IEEE Software Volume: 29, Issue: 6, Nov.-Dec. 2012	P4
Edith Tom, Aybüke Aurum, Richard Vidgen. “An exploration of technical debt”. 2013. In: Journal of Systems and Software Volume 86, Issue 6, June 2013, Pages 1498-1516. Elsevier	S9
Zadia Codabux, Byron Williams. „Managing technical debt: An industrial case study”. 2013. In: Managing Technical Debt (MTD), 2013 4th International Workshop on. IEEE	S10
Robert L. Nord, Ipek Ozkaya, Philippe Kruchten. “In Search of a Metric for Managing Architectural Technical Debt”. 2012. In: WICSA-ECSA '12 Proceedings of the 2012 Joint Working IEEE/IFIP Conference on Software Architecture. IEEE	P5
Carolyn Seaman, Yuepu Guo. „Measuring an Monitoring Technical Debt”. 2011. In: Advanced in Computer, Vol. 82, Elsevier	P6
Frank Buschmann. “To Pay or Not to Pay Technical Debt”. 2011. In: IEEE Software Volume: 28, Issue: 6, Nov.-Dec. 2011	P7
Jesse Yli-Huumo, Andrey Maglyas, Kari Smolander. “The Sources and Approaches to Management of Technical Debt: A Case Study of Two Product Lines in a Middle-Size Finnish Software Company”. 2014. In: 15th International Conference, PROFES 2014, Helsinki, Finland, December 10-12, 2014, At Helsinki, Finland, Volume: 8892. Research Gate	S11
Antonio Martini, Jan Bosch. "The Danger of Architectural Technical Debt: Contagious Debt and Vicious Circles”. 2015. In: WICSA '15 Proceedings of the 2015 12th Working IEEE/IFIP Conference on Software Architecture. ACM	S12
John D. McGregor, J. Yates Monteith, and Jie Zhang. “Technical Debt Aggregation in Ecosystems”. 2012. In: Managing Technical Debt (MTD), 2012 Third International Workshop on. IEEE	P8
Eric Allma. „Managing Technical Debt. Shortcuts that save money and time today can cost you down the road”. 2012. In: Magazine Communications of the ACM CACM Homepage archive Volume 55 Issue 5, May 2012 Pages 50-55. ACM	P9
Johannes Holvitie, Ville Leppänen, Sami Hyrynsalmi. “Technical Debt and the Effect of Agile Software Development Practices on It - An Industry Practitioner Survey”. 2014. In: Managing Technical Debt (MTD), 2014 Sixth International Workshop on. IEEE	S13
Will Snipes, Brian Robinson, Yuepu Guo, Carolyn Seaman. “Defining the decision factors for managing defects: A technical debt perspective”. 2012. In: MTD '12 Proceedings of the Third International Workshop on Managing Technical Debt Pages 54-60. ACM	S14
Nicolli S.R. Alves, Leilane F. Ribeiro, Vivyane Caires. “Towards an Ontology of Terms on Technical Debt”. 2014. In: Managing Technical Debt (MTD), 2014 Sixth International Workshop on. IEEE	S15
Rick Kazman, Yuanfang Cai, Ran Mo, Qiong Feng ,Lu Xiao, Serge Haziyev ,Volodymyr Fedak , Andriy Shapochka. "A case study in locating the architectural roots of technical debt”. 2015. In: ICSE '15 Proceedings of the 37th International Conference on Software Engineering - Volume 2 Pages 179-188. ACM	S16
Areti Ampatzoglou, Apostolos Ampatzoglou, Alexander Chatzigeorgiou, Paris Avgeriou. “The financial aspect of managing technical debt: A systematic literature review”. 2014. In: Information and Software Technology Volume 64, August 2015, Pages 52-73. Elsevier	S17

Tabelle 5.2: Ursachen Technischer Schulden – Quellen und Akronym 2

Quelle	Akronym
Areti Ampatzoglou, Apostolos Ampatzoglou, Alexander Chatzigeorgiou, Paris Avgeriou. "The financial aspect of managing technical debt: A systematic literature review". 2014. In: Information and Software Technology Volume 64, August 2015, Pages 52-73. Elsevier	S17
Robert J. Eisenberg. "A Threshold Based Approach to Technical Debt". 2012. In: ACM SIGSOFT Software Engineering Notes archive Volume 37 Issue 2, March 2012 Pages 1-6. ACM	S18
Md Abdullah Al Mamun, Christian Berger, Jörgen Hansson. „Explicating, Understanding, and Managing Technical Debt from Self-Driving Miniature Car Projects". 2014. In: Managing Technical Debt (MTD), 2014 Sixth International Workshop on. IEEE	S19
Antonio Martini, Jan Bosch, Michel Chaudron. "Investigating Architectural Technical Debt accumulation and refactoring over time: A multiple-case study". 2015. In: Journal Information and Software Technology archive Volume 67 Issue C, November 2015 Pages 237-253. Elsevier	S20
Georgios Skourletopoulos, Constandinos X. Mavromoustakis, Rami Bahsoon. "Predicting and quantifying the technical debt in cloud software engineering". 2014. In: Computer Aided Modeling and Design of Communication Links and Networks (CAMAD), 2014 IEEE 19th International Workshop on. IEEE	P10
Nico Zazworka, Michele A. Shaw, Forrest Shull, Carolyn Seaman. "Investigating the Impact of Design Debt on Software Quality". 2011. In: MTD '11 Proceedings of the 2nd Workshop on Managing Technical Debt Pages 17-23. ACM	S21
Israel Gat, John D. Heintz. „From Assessment to Reduction: How Cutter Consortium Helps Rein in Millions of Dollars in Technical Debt". 2011. In: MTD '11 Proceedings of the 2nd Workshop on Managing Technical Debt Pages 24-26. ACM	P11
Narayan Ramasubbu, Chris F. Kemerer. "Managing Technical Debt in Enterprise Software Packages". 2014. In: IEEE Transactions on Software Engineering Volume: 40, Issue: 8, Aug. 2014. IEEE	S22
Edith Tom, Aybuke Aurum, Richard Vidgen. "A CONSOLIDATED UNDERSTANDING OF TECHNICAL DEBT". 2012. In: Association for Information Systems AIS Electronic Library (AISeL).	S23
Trong Tan Ho, Guenther Ruhe. "When-to-Release Decisions in Consideration of Technical Debt". 2014. In: Managing Technical Debt (MTD), 2014 Sixth International Workshop on	S24
Paris Avgeriou, Philippe Kruchten, Ipek Ozkaya, and Carolyn Seaman. "Managing Technical Debt in Software Engineering". 2016. In: Dagstuhl Research Online Publication Server	P12
Vinay Krishna, Dr. Anirban Basu. "Software Engineering Practices for Minimizing Technical Debt". 2013. In: Conference: SERP13. ResearchGate	P13
Yuepu Guo, Rodrigo Oliveira Spínola, Carolyn Seaman. "Exploring the costs of technical debt management – a case study". 2016. In: Springer , 1. Auflage	S25
N. Ernst, "On the role of requirements in understanding and managing technical debt". 2012. In: 3rd International Workshop on Managing Technical Debt (MTD '12), IEEE Computer Society, pp. 61–64, Zurich, Switzerland. IEEE	P14
N. Ernst. "Technical debt and requirements". 2012. In: Proc. 3rd Int. Workshop Managing Tech. Debt, pp. 61-64	P15
Nanette Brown, Yuanfang Cai, Yuepu Guo, Rick Kazman, Miryung Kim, Philippe Kruchten, Erin Lim, Alan MacCormack, Robert Nord, Ipek Ozkaya, Raghvinder Sangwan, Carolyn Seaman, Kevin Sullivan, Nico Zazworka. "Managing Technical Debt in Software-Reliant Systems". 2010. In: FoSER '10 Proceedings of the FSE/SDP workshop on Future of software engineering research Pages 47-52. ACM	P16
Zahra Shakeri Hossein Abad, Guenther Ruhe. "Using real options to manage Technical Debt in Requirements Engineering". 2015. In: Requirements Engineering Conference (RE), 2015 IEEE 23rd International. IEEE	P17
Zahra Shakeri Hossein Abad, Guenther Ruhe. "Using Real Options to Manage Technical Debt in Requirements Engineering" 2015. In: Requirements Engineering Conference (RE), 2015 IEEE 23rd International. IEEE	P18

Tabelle 5.3: Ursachen Technischer Schulden – Quellen und Akronym 3

5.4 Datenquellen und Suchstrategie

Um eine entsprechende Qualität der einzelnen Studien und der systematischen Literaturrecherchen zu gewährleisten, wurden die gefundenen Publikationen, nach den DARE-Kriterien der „York University, Centre for Reviews and Dissemination (CDR) Database of Abstracts of Reviews of Effects“, bewertet. Folgende Qualitätskriterien wurden genutzt:

- Wurden in den Reviews angemessene Ein und Ausschlusskriterien definiert?
- Hat die Literatursuche alle relevanten Studien abgedeckt?
- Wurde die Qualität der Studien überprüft?
- Wurden die einzelnen Studien/Daten ausreichend beschrieben?

Die Bewertung der Fragen fand wie folgt statt:

Kriterium 1: J(Ja), es wurden angemessene Ein- und Ausschlusskriterien definiert; T(Teilweise), die Ein- und Ausschlusskriterien wurden ungenau definiert; N(Nein), es sind keine Ein- und Ausschlusskriterien vorhanden, die Literaturrecherche gilt als unzureichend.

Kriterium 2: J(Ja), die Autoren haben vier oder mehr Quellen/Literaturen genutzt und auf diese verwiesen; T (Teilweise), die Autoren haben drei oder vier Quellen/Literaturen genutzt und auf diese verwiesen; N (Nein), die Autoren haben nur 2 oder weniger Quellen/Literaturen genutzt.

Kriterium 3: J(Ja), die Autoren haben Qualitätskriterien für ihre Quellen definiert; T(Teilweise), es sind Qualitätsmängel zu erkennen; N(Nein), es wurden keine Qualitätseigenschaften festgelegt.

Kriterium 4: J(Ja), die Informationen jeder Studie wurden präsentiert; T(Teilweise), nur eine Zu-sammenfassung der Studien wurde präsentiert; N(Nein), die Ergebnisse der Studien wurden nicht präsentiert.

Für die Auswertung fand folgende Punkteverteilung statt:

- Y = 1
- T = 0,5
- N = 0

Studien, Vorträge und Publikationen zum Thema Technische Schulden enthielten oftmals keine Antworten zu den gestellten Fragen. Aufgrund des Zieles dieser Arbeit möglichst viele Ursachen von Technischen Schulden und Requirement Technical Debts zu finden, wurden Studien, Vorträge und Publikationen mitaufgenommen, welche zwar die oben aufgeführten Fragen nicht explizit beantworten, aber aufgrund ihrer wissenschaftlichen Qualität Hinweise dafür liefern. In diesen Literaturen wurde auf folgendes geachtet:

- Beschäftigt sich die Literatur oder Publikation mit dem Thema?
- Spezialisieren sich die Autoren auf das bestimmte Themengebiet?
- Handelt es sich hierbei um eine wissenschaftliche Arbeit?

Nur wenn diese Kriterien erfüllt waren, wurde die Quelle dennoch aufgenommen.

5.5 Datensammlung

Daten, welche von jeder Quelle erfasst wurden:

- Die vollständige Quellenangabe und der Herausgeber
- Eine Klassifikation der Studie (Systematische Literaturrecherche, Metaanalyse, Zeitschrift etc.)
- Hauptthema der Literatur
- Die Autoren, die Institution und das Land der Studie/Literatur
- Die einzelnen Resultate oder eine Zusammenfassung dieser zum Thema Ursachen von Technischen Schulden oder nur die einzelnen Ursachen allein
- Die Ein- und Ausschlusskriterien für die Qualitätsprüfung
- Die Anzahl der genutzten Literaturen für die Qualitätssicherung
- Die eigenen Qualitätskriterien der Studie für die Qualitätssicherung

5.6 Synthese der Funde

Folgende Daten wurden anhand einer Tabelle zusammengefasst und ausgewertet:

- Die Quelle der einzelnen Literaturrecherchen
- Die Institution der einzelnen Literaturrecherchen
- Die Autoren der einzelnen Literaturrecherchen
- Das Jahr der Veröffentlichungen der einzelnen Literaturrecherchen
- Der Titel der einzelnen Literaturrecherchen
- Das Qualitätsergebnis zu den einzelnen Literaturen
- Die Ursachen von Technischen Schulden der einzelnen Literaturen

Abbildung 5.1 Zeigt die Vorgehensweise dieser Arbeit. Die Recherche und Analyse lässt sich in drei größere Bereiche gliedern. Der erste Bereich beschäftigt sich mit der Datenextraktion von Ursachen von Technischen Schulden. Dabei wird eine Qualitätskontrolle durchlaufen, welche die Quellen nach ihrer Vertrauenswürdigkeit prüft. Quellen welche den definierten Qualitätsmerkmalen entsprechen, werden in einer Ergebnistabelle gesammelt und sortiert.

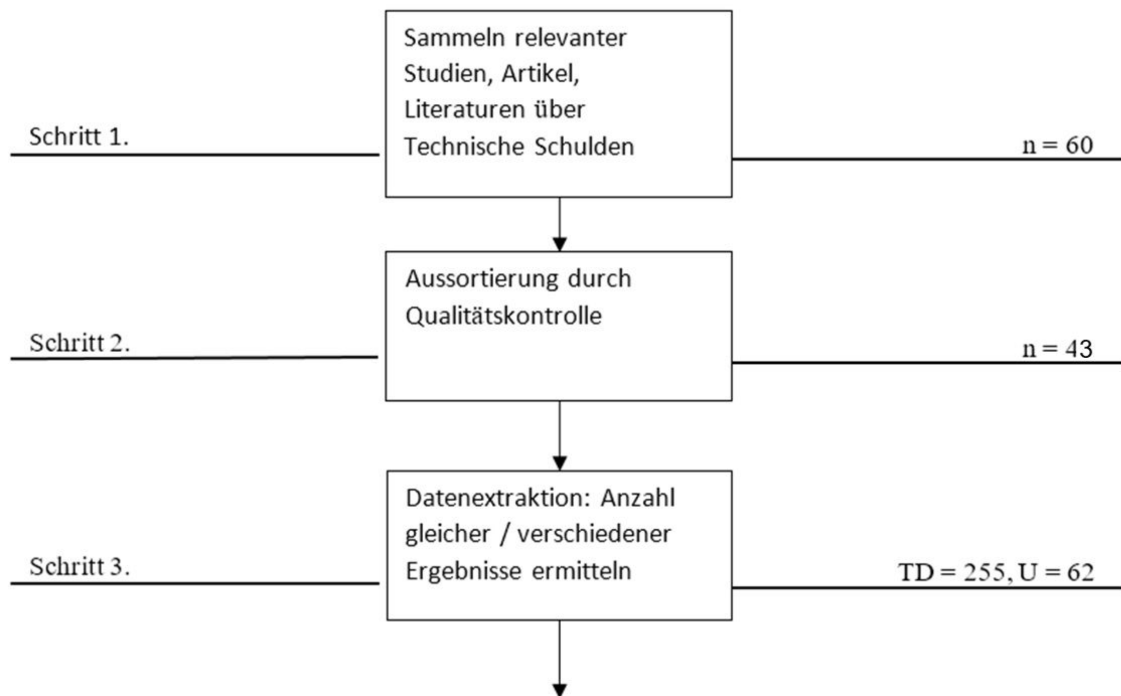

Abbildung 5.1: Vorgehen - Technische Schulden

Quellen welche sich mit Requirement Technical Debt (RTD) beschäftigt haben, wurden im zweiten Arbeitsschritt weiterverwendet. Hierbei wurden die Ursachen der RTD herausgefiltert und in einer weiteren Ergebnistabelle aufgelistet. Es gab 25 Quellen welche sich mit RTD beschäftigt haben. Insgesamt wurden 133 Ursachen gesammelt. Diese lassen sich in 28 unterschiedliche Ursachen von RTD unterteilen.

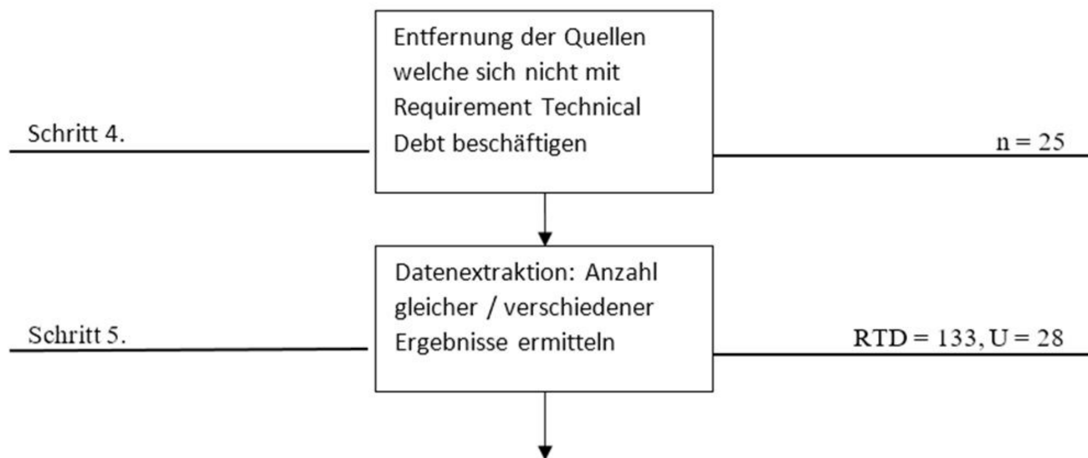

Abbildung 5.2: Vorgehen - Requirement Technical Debt

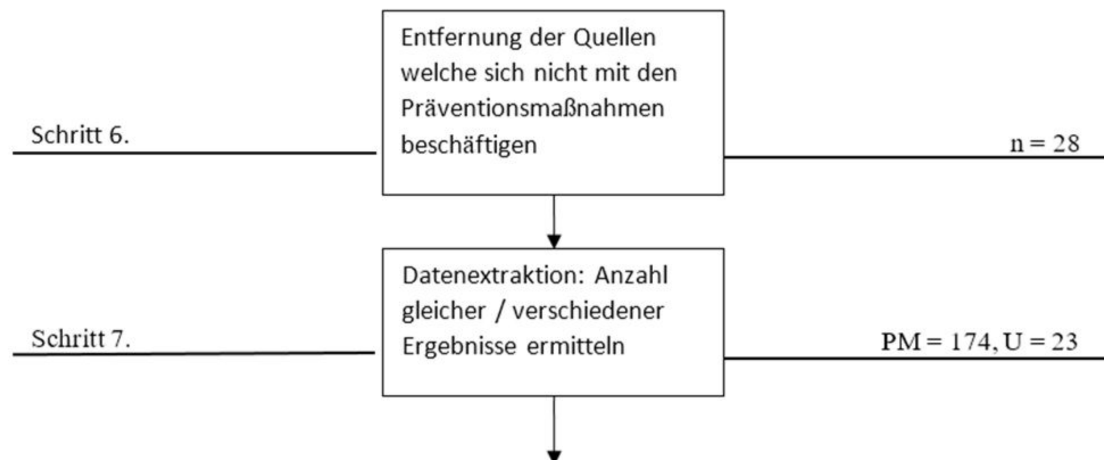

Abbildung 5.3: Vorgehen - Maßnahmen

Der dritte Teil dieser Arbeit beschäftigt sich schließlich mit den Präventionsmaßnahmen gegenüber Technischen Schulden. Dabei wurde auf die bereits vorhandenen Quellen zurückgegriffen und nach Maßnahmen gesucht. Es gab 28 Quellen welche Präventionsvorschläge enthielten. Insgesamt wurden 174 Vorschläge gesammelt. Diese lassen sich in 23 unterschiedliche Maßnahmen unterteilen.

6 Qualitätsbewertung der einzelnen Quellen

Akronym	Kriterium 1	Kriterium 2	Kriterium 3	Kriterium 4	Summe
S1	1	1	1	1	4
S2	1	1	0	0,5	2,5
S3	1	1	1	1	4
S4	0,5	1	0,5	0,5	2,5
S5	0	1	0	1	2
S6	0	1	0	0	1
S7	0	1	0	0	1
S8	1	1	1	1	4
S9	1	1	1	1	4
S10	0	1	0	0,5	1,5
S11	0,5	1	0,5	0,5	2,5
S12	0,5	1	0,5	1	3
S13	0	1	0	0,5	1,5
S14	1	1	1	0,5	3,5
S15	1	1	1	0,5	3,5
S16	1	1	1	0,5	3,5
S17	1	1	1	1	4
S18	0	1	0	1	2
S19	0,5	1	1	1	3,5
S20	1	1	1	1	4
S21	0,5	1	0	1	2,5
S22	0,5	1	0	1	2,5
S23	1	1	1	1	4
S24	1	1	0,5	0,5	3

Tabelle 6.1: Qualitätsbewertung der Quellen 1

S25	1	1	1	1	4
P1	-	-	-	-	-
P2	-	-	-	-	-
P3	-	-	-	-	-
P4	-	-	-	-	-
P5	-	-	-	-	-
P6	-	-	-	-	-
P7	-	-	-	-	-
P8	-	-	-	-	-
P9	-	-	-	-	-
P10	-	-	-	-	-
P11	-	-	-	-	-
P12	-	-	-	-	-
P13	-	-	-	-	-
P14	-	-	-	-	-
P15	-	-	-	-	-
P16	-	-	-	-	-
P17	-	-	-	-	-
P18	-	-	-	-	-

Tabelle 6.2: Qualitätsbewertung der Quellen 2

In der Tabelle 6.1/6.2 wurden alle Studien nach den Qualitätskriterien bewertet. Bücher, Vorlesungen und Publikationen, in welchen keine messbare Studie durchgeführt wurde, und keinen Aufschluss über die genannten Qualitätskriterien gaben, wurden nicht bewertet. In diesen Literaturen wurde auf folgendes geachtet:

1. Beschäftigt sich die Literatur oder Publikation mit dem Thema?
2. Spezialisieren sich die Autoren auf das bestimmte Themengebiet?
3. Handelt es sich hierbei um eine wissenschaftliche Arbeit?

Nur wenn diese Kriterien erfüllt waren, wurde die Quelle dennoch aufgenommen.

Bewertete Studien wurden mit einem S und der jeweiligen Nummerierung bezeichnet. Publikationen, welche anhand der Qualitätskriterien nicht bewertet werden konnten, wurden mit einem P bezeichnet. Die Gesamtpunktezahl aller bewerteten Studien ist 73,5. Der Mittelwert der bewerteten Studien ist 2,94. Der Mittelwert kann als Qualitätseigenschaft dieser Arbeit gesehen werden.

7 Ergebnisse

Die Ergebnisse wurden in drei Gruppen unterteilt: Die Ursachen von Technischen Schulden, Die Ursachen von Requirement Technical Debts und die Maßnahmen gegen Technische Schulden.

7.1 Ursachen von Technischen Schulden

In Tabelle 7.1/7.2 sind die aus den Quellen gefundenen Ursachen von Technischen Schulden aufgelistet. Aufgrund der großen Anzahl (62) unterschiedlicher Ursachen wird sich in dieser Arbeit auf die 37 häufigsten Ursachen beschränkt. Ursachen welche insgesamt nur einzeln vorkamen, wurden aus der Auflistung ausgeschlossen. Die Anzahl gibt in der Tabelle das Vorkommen der einzelnen Ursachen in allen Quellen an. In der Tabelle werden die Ursachen nach der Häufigkeit sortiert. Ähnliche Ursachen von Technischen Schulden (TD) wurden zu einem Punkt zusammengefasst.

	Ursachen	**Anzahl**
1	Mangelhaftes Design bzw. Architektur	33
2	Mangel an Achtsamkeit (Code Probleme, Defekte)	18
3	Ungenügende Testabdeckung	18
4	Fehlende oder mangelhafte Dokumentation	16
5	Fehlende Bildung / Schulung / Aufklärung	14
6	Enger Zeitplan / Druck durch Zeitplan	12
7	Mangelhaftes Verständnis was das zu bauende System und Anforderungen betrifft	10
8	Hohe Komplexität	7
9	Schwerpunkt auf schnelle Lieferung	6
10	Defekte wegen mangelnden Ressourcen oder falsch gesetzten Prioritäten	6
11	Schlechte Entwicklungsprozesse	5
12	Spontane Änderungen	5
13	Code Duplikate	5
14	Kunde kommt plötzlich mit neuen Anforderungen oder die Liste an Anforderungen wird zu lang	5
15	Mangelhaftes Wissen über den Markt, was die Menschen haben wollen und wo man in mehreren Jahren stehen wird	5
16	Schlechte Kommunikation zwischen Klienten und dem Software Unternehmen	4
17	Abkürzungen welche später zu Problemen führen	4

Tabelle 7.1: Ursachen von Technischen Schulden 1

18	Übersehende Lösungen / Probleme	4
19	Implementierte Features und Anpassungen führen zu neuen Anforderungen	4
20	Benutzung von „Third Party Software“ und Open Source Software ohne genaues Verständnis	4
21	Refactoring wird immer wieder verschoben	3
22	Schlechte Performance der Entwickler	3
23	Die Technik der Software veraltet im Verlauf des Entwicklungsprozesses. Technische Evolution.	3
25	Die To-Do Liste steigt ununterbrochen	3
26	Verletzung von Style Guides und Software Patterns	3
27	Trade-Offs von TB und Software Qualität	3
28	Gar kein Refactoring	2
29	Fehlendes Interesse am Projekt	2
30	Mangelhafte Anforderungsanalyse	2
31	Qualitätsanalyse ist nicht vorhanden	2
32	Schlechte Lesbarkeit des Codes	2
33	Keiner versteht den Code	2
34	Priorität der Features anstatt einem fertigen Produkt	2
35	Ignoranz/Desinteresse was Technische Schulden betrifft	2
36	Angst vor Änderungen, da sonst woanders Probleme auftreten	2
37	Strenge Kostengrenzen	2

Tabelle 7.2: Ursachen von Technischen Schulden 2

33 der 43 untersuchten Quellen setzen eine mangelhafte Architektur und mangelhaftes Design als Ursache von Technischen Schulden. Das Softwaredesign oder auch die Architektur werden in der Softwareentwicklung entworfen, um einen Entwurf des zu erstellenden Systems zu gewährleisten. Dabei werden mittels bestimmter Architekturbeschreibungssprachen die einzelnen Komponenten und deren Zusammenspiel beschrieben [12]. Auch die Oberfläche und die Zusammenhänge zwischen den Anforderungen und dem konstruierten System können erläutert werden [12]. Es zeigt sich, dass ein gutes Softwaredesign einen sehr großen Einfluss auf eine erfolgreiche Entwicklung hat. Durch ein mangelhaftes Design erhöht sich somit die Wahrscheinlichkeit Technischer Schulden. Ein mangelhaftes Design kann auf die Nichteinhaltung von Designprinzipien [13], schlechten Designern [13] oder einer schnellen und schlechten Darstellung der Struktur zurückgeführt werden.

Der zweite Punkt betrifft den Mangel an Aufmerksamkeit während der Entwicklung. Die Unachtsamkeit der Entwickler kann an Desinteresse am Projekt, an Überforderung oder anderen Faktoren liegen.

18 von 43 Quellen geben einen Mangel der Testabdeckung als Grund für die Technischen Schulden an. Dies kann auf fehlende Tests oder mangelhafte Testdurchläufe zurückgeführt werden [14].

Neben dem Mangel der Testabdeckung spielt auch die Dokumentation eine große Rolle bei den Technischen Schulden. Auch hierbei kann eine schlechte oder fehlende Dokumentation zur Unübersichtlichkeit führen und so Technische Schulden verursachen.

An fünfter Stelle steht die fehlende Bildung, beziehungsweise Schulung, der Angestellten. Softwareentwickler sind oftmals keine Experten im Portfolio Management [15]. Auch ein Mangel an Erfahrung ist in diesem Aspekt ausschlaggebend [16]. Durch das fehlende Wissen können ver-schieden Style-Guides und Prinzipien verletzt werden. Des Weiteren kann sich dadurch der Entwicklungsprozess verlangsamen.

Ein enger Zeitplan führt zu großem Druck unter den Entwicklern, was diese fehleranfälliger macht [17] [18]. Durch den Zeitmangel können Fehler oftmals nicht rechtzeitig und mit der gebotenen Gründlichkeit beseitigt werden.

Mangelhaftes Verständnis über das zu bauende System oder eine zu hohe Komplexität kann ebenfalls dazu führen, dass das System nicht vollständig implementiert wird [19].

Oftmals kommt es in Unternehmen vor, dass der Schwerpunkt auf die schnelle Lieferung gelegt wird. Dies kann zu den gleichen Problemen wie in ein enger Zeitplan führen.

An zehnter Stelle werden der Mangel an Ressourcen und falsch gesetzten Prioritäten aufgelistet. Die Ressourcen können sich hierbei auf verschiedene Aspekte beziehen. Ein Mangel an verfügbaren Budget, zu wenig Entwickler oder Zeit kann zu vielerlei Problemen führen und die ganze Entwicklung gefährden.

Ein stockender Entwicklungsprozess führt zu einer negativen Rückkopplung. Der Abstand zwischen dem aktuellen Stand und dem geplanten Ergebnis verringert sich nur langsam. Dadurch entsteht ein noch größerer Zeitdruck, womit auch Technische Schulden entstehen. In der Softwareentwicklung können sehr viele Faktoren zu Technischen Schulden führen. Spontane Änderungen, sowie neuen Anforderungen welche von dem Kunden eingebracht werden, können ganze Entwicklungspläne ruinieren.

Auch die Codequalität spielt eine wichtige Rolle in der Entwicklung. Code Duplikate, Third-Party-Software, die Verletzung von Software Style Guides und das Aufschieben von Refactorings kann zu unüberschaubaren Programmcode führen. Das Refactoring wird dadurch sehr erschwert. Oftmals lohnt es sich mehr, den Code komplett neu zu schreiben.

Eine schlechte Kommunikation innerhalb des Teams oder auch zwischen dem Kunden und dem Softwareunternehmen kann dazu führen, dass die entstandene Software nicht den Anforderungen des Klienten entspricht. Die Kommunikation innerhalb des Teams ermöglicht das schnelle Entdecken und Lösen von Technischen Schulden.

Doch auch externe Faktoren können zu Technischen Schulden führen. So spielt zum Beispiel auch die Marktentwicklung eine sehr wichtige Rolle. Innerhalb einer kurzen Zeit kann sich das Interesse sehr schnell ändern, wodurch auch bestehende Software neu angepasst werden muss. Es ist oftmals, aufgrund der großen Anzahl, sehr schwer vorherzusagen, welche Software bereits von anderen Unternehmen entworfen wurde, die in ihrer Funktion der eigenen ähnelt. Je nach der zu erstellenden Software, muss darauf geachtet werden, wie sich der Markt entwickelt. Im schlimmsten Fall veraltet die Software schon während dem Entwicklungsprozess. Dies kann dann

der Fall sein, wenn andere Unternehmen bereits zu einer Lösung gekommen sind oder sich das Interesse der Nutzer ändert.

Es ist anhand der Tabelle zu erkennen, dass Technische Schulden durch verschiedener Faktoren entstehen können. Die 37 genannten Ursachen von TD sind noch lange nicht alle möglichen Gründe. Dennoch ist sehr gut zu erkennen, welche Probleme laut verschiedener Publikationen besonders auffällig erscheinen und worauf in der Softwareentwicklung speziell geachtet werden muss.

7.2 Ursachen von Requirement Technical Debts

In Tabelle 7.3 werden die aus den Quellen gefundenen Ursachen von Requirement Technical Debts aufgelistet. Unter den 43 Quellen haben sich 25 Quellen mit dem Begriff Requirement Technical Debts auseinandergesetzt. Insgesamt wurden 26 verschiedene Arten von RTD gefunden. Auch hier gibt die Anzahl das Vorkommen der einzelnen Ursachen in allen Quellen an. Die Ursachen werden absteigend nach ihrer Häufigkeit sortiert. Ursachen, welche sich sehr stark ähneln, wurden zu einem Punkt zusammengefasst.

Nr.	Ursachen	Anzahl
1	Fehlende oder Mangelhafte Spezifikation / Architektur	23
2	Vernachlässigung von Anforderungen	13
3	Mangelhaftes systematisches Testen	11
4	Entwicklung unter Zeitdruck	9
5	Mangelhafte Anforderungssammlung	7
6	Fehlendes Interesse am Projekt	7
7	Kunde kommt in der späten Entwicklungsphase mit neuen Anforderungen	7
8	Die Technik der Software veraltet im Verlauf des Entwicklungsprozesses. Technische Evolution.	6
9	Hohe Komplexität	6
10	Große Distanz zwischen aktuellem Stand und der optimalen Lösung	5
11	Fehlende Ressourcen	5
12	Fehlendes Wissen / Bildung	4
13	Durchführung einer parallelen Entwicklung	3
14	Unwissen über Patterns und Richtlinien	3
15	Entscheidungen ohne Voraussicht	3
16	Abkürzungen welche später zu Problemen führen	3
17	Arbeit ohne Use-Cases erschwert die Erstellung einer Architektur	3
18	Kunden sind sich unschlüssig über das was sie wollen	3
19	Schlechte / Langsame Entwicklung	3
20	Anforderungen die sich im Verlauf der Entwicklung ändern	2
21	Schlechte Kommunikation zwischen Klienten und Software Unternehmen	2
22	Projektmanager nur an Kosten und Gewinn interessiert	2
23	Unschlüssigkeit, wann eine Anforderung abgeschlossen/erledigt ist	1
24	Schlechtes System Design in der Planungsphase	1
25	Keine systematische Prüfung der Qualität	1
26	Sequentielles Arbeiten (Wasserfallmodell)	1

Tabelle 7.3: Ursachen von Requirement Technical Debts

Im Regelfall wurden Requirement Technical Debts in den verschiedenen Quellen nicht von den allgemeinen Technischen Schulden unterschieden. Aus diesem Grund wurden gezielt Publikationen ausgewählt, in deren Definition von Technischen Schulden auch die Requirement Technical Debts berücksichtigt wurden. Da für Requirement Technical Debts keine einheitliche Definition existiert und diese oftmals nicht als eigene Rubrik von Technischen Schulden angesehen werden, orientiert sich diese Arbeit an der hergeleiteten Definition von Requirement Technical Debts aus dem Kapitel 2.1. Es wurden die Schulden ausgewählt, welche sich mit der groben Definition nicht überschnitten. Technische Schulden, welche der Definition wiedersprachen oder sich nur auf die Codeschulden bezogen haben, wurden aus der Sammlung entfernt. Im Folgenden wird auf einige Ursachen kurz eingegangen und diese erklärt.

Anhand der Tabelle wird deutlich, dass vor allem die Spezifikation und Architektur eine wesentliche Rolle spielen. Dabei bleibt offen, ob die Architektur nun in die Rubrik Design Debt oder Requirement Debt gehört. Beide Rubriken überschneiden sich. Die Erstellung einer Spezifikation und Architektur ist Teil der Anforderungsanalyse, da in diesen Schritten geklärt wird, wie die zu entstehende Software auszusehen und welche Anforderungen sie erfüllt hat. Die Spezifikation und Architektur wurden deshalb zu einem Punkt vereinigt, da sie in vielen Quellen zusammengefasst werden. Insgesamt vertraten 23 von 25 Quellen die Auffassung, dass die Spezifikation und die Architektur Gründe für TD sind.

Laut 13 der 25 Quellen kommt es in der Softwareentwicklung vor, dass einzelne Anforderungen vernachlässigt werden. Dies kann in der Entwicklungsphase dazu führen, dass diese Anforderungen nicht rechtzeitig erfüllt werden oder ganz aus der Entwicklung herausfallen.

Das systematische Testen ist Teil der Anforderungen bei der Erstellung einer Software. Werden Tests gar nicht, nur selten oder in einer lückenhaften Weise durchgeführt, so können RTD entstehen, da so Fehler leichter übersehen werden. Auch diese Ursache hat laut der Anzahl in der Tabelle einen hohen Stellenwert.

In der Softwareentwicklung ist es wichtig, genügend Zeit für die einzelnen Entwicklungsschritte beziehungsweise Iterationen einzuplanen. Die benötigte Zeit ist somit auch eine Anforderung, die leider häufig vernachlässigt wird.

Eine mangelhafte Anforderungssammlung führt dazu, dass das fertig gestellte Softwaresystem nicht den Anforderungen des Kunden entspricht. Anforderungen, welche während des Systems beispielsweise durch neue Wünsche des Kunden hinzu kommen, können zu Zeit- und Ressourcenmangel mit ihren negativen Folgen führen. Die Kluft zwischen dem aktuellen System und dem optimalen Produkt wird dadurch immer größer.

Auch das Interesse ist eine Anforderung die in der Erstellung eines Produktes gegeben sein sollte. Desinteresse führt zu schlampiger Arbeit oder zu Unwissen über das zu entstehende Produkt.

Wie in den Technischen Schulden schon erklärt, kann es vorkommen, dass das System während der Entwicklung veraltet. Es ist somit eine Anforderung einen Überblick über den derzeitigen Markt und Technikstand zu haben, bevor mit der Entwicklung beginnen kann.

Eine hohe Komplexität kann in der Softwareentwicklung durch ungenügende Voraussicht und einer schlechten Planung entstehen. Es ist wichtig für die Entwickler, eine genaue Architektur und Spezifikation zu erstellen. Viele Quellen nennen die große Distanz zwischen dem aktuellen

Stand und der optimalen Lösung als Grund für ein RTD. Da vor allem am Anfang der Entwicklungsphase eine große Distanz besteht, ist diese Art von RTD in jedem Projekt zunächst immer vorhanden.

Die Ressourcen und auch die Ausbildung der Mitarbeiter sind ein weiterer Anforderungspunkt, welche eine wesentliche Rolle spielt.

Patterns und Richtlinien werden in der Softwareentwicklung benötigt, um ein einheitliches Qualitätsmaß zu gewährleisten [21]. Die Patterns und Richtlinien stehen den Entwickler zur Verfügung, um ein einheitliches System nach einem qualitätsgerechten Format zu entwickeln. Unwissenheit über verschiedene Patterns und Richtlinien können zu zwanghaften Refactorings führen.

Die parallele Entwicklung ist ein Faktor, welcher in fast jedem Softwareprojekt präsent ist. Die isolierte parallele Bearbeitung verschiedener Bereiche führt vor allem in der Zusammenführung häufig zu Problemen, zu deren Beseitigung ein hoher Arbeitsaufwand erforderlich ist. Man erkennt, dass alle Ursachen aus der Tabelle ganz oder teilweise zu den Anforderungen ei-ner Softwareentwicklung gehören. Der wichtigste Schritt besteht in der Voraussicht. Es muss versucht werden, jeden einzelnen Schritt zu planen.

Ein weiteres Problem ist die Unschlüssigkeit der Kunden. Oftmals ist diesen nicht wirklich klar, welche Anforderungen für ein System bestehen. Dies führt dazu, dass sich Entwickler teilweise ihre Informationen oder Anforderungen selber herleiten müssen.

Dass Kunden sich unschlüssig darüber sind, gehört eigentlich auch zum Punkt der mangelhaften Anforderungssammlung. In diesem Aspekt führen wir diese Art der Ursache getrennt auf, um auf diese nochmal speziell hinzuweisen. Es ist den Kunden oftmals nicht bewusst, aus welchen Aspekten eine Anforderungssammlung besteht. Je genauer die Spezifikation vorgegeben wird, desto einfacher fällt es den Entwicklern, ein System zu implementieren. Ein weiterer Punkt welcher mit diesem Thema sehr nah in Verbindung steht, sind die Anforderungen, welche sich im Verlauf der Entwicklung ändern. Dies ist entweder auf die Unschlüssigkeit der Entwickler oder auf eine mangelhafte Anforderungsanalyse zurückzuführen. Des Weiteren wissen Entwicklern oftmals nicht genau, ab wann eine Anforderung als abgeschlossen gelten kann und sie den Wünschen des Kunden entspricht. Dies führt uns zu einem weiteren Punkt, der schlechten Kommunikation zwischen Kunden und Softwareunternehmen. Es spielt in der Anforderungs-sammlung eine entscheidende Rolle, ob ein guter Kontakt zu den Kunden besteht, um falls nötig anfallende Fragen ohne Zeitverzug klären zu können. Auch wenn diese Art der Ursache in den RTD Quellen nur einmal erwähnt wurde, so gehört dieser Punkt dennoch zu den wichtigsten Ursachen. In der Softwareentwicklung kommt es allgemein oft vor, dass Kunden ein anderes Bild ihrer benötigten Software haben als die Entwickler.

Allerdings werden RTD nicht nur durch Kunden und Entwickler verursacht. Auch Manager, welche primär an Gewinn interessiert sind, haben ein Interesse daran, ein Projekt so schnell wie nur möglich abzuschließen. Darunter leidet häufig die Qualität.

7.3 Auswirkungen

7.3.1 Auswirkungen von Requirement Technical Debt auf die Agile Entwicklung

Auch die Arbeitsweise während der Entwicklung spielt eine wichtige Rolle. In der sequentiellen Arbeitsweise, können Technische Schulden zu sehr hohen Kosten führen. Ein Beispiel für eine sequentielle Arbeitsweise ist das Wasserfallmodell. Im Folgenden ist der grobe Ablauf einer Softwareentwicklung in einem Wasserfallmodell zu sehen. Die Skizze wurde aus den Schaubildern der Universität Stuttgart, der Universität Kiel und der Universität Leipzig erstellt.

Das Wasserfallmodell (Abbildung 7.1) ist eine weitverbreitete Variante der Softwareentwicklung. [22] Wie in der Abbildung zu sehen ist, besteht das Model aus mehreren Phasen. Dabei ist jede Phase abzu-schließen, bevor die nächste begonnen wird. Bei dem Wasserfallmodell handelt es sich um eine sequentielle Vorgehensweise. [22] Ein Rückschritt ist in einem Wasserfallmodell normalerweise nicht vorgesehen und zudem mit hohem Aufwand verbunden. Darüber hinaus ist das Wasserfallmodell mit folgenden Nachteilen verbunden [22]:

- Die Vollständige sequentielle Abwicklung der Entwicklungsschritte ist nicht immer möglich
- Es besteht die Gefahr falscher Prioritäten (z.B. Fokus auf Dokumentation)
- Es gibt keine Berücksichtigung von Risikofaktoren
- Änderungen / Anpassungen sind nur sehr schwer umsetzbar

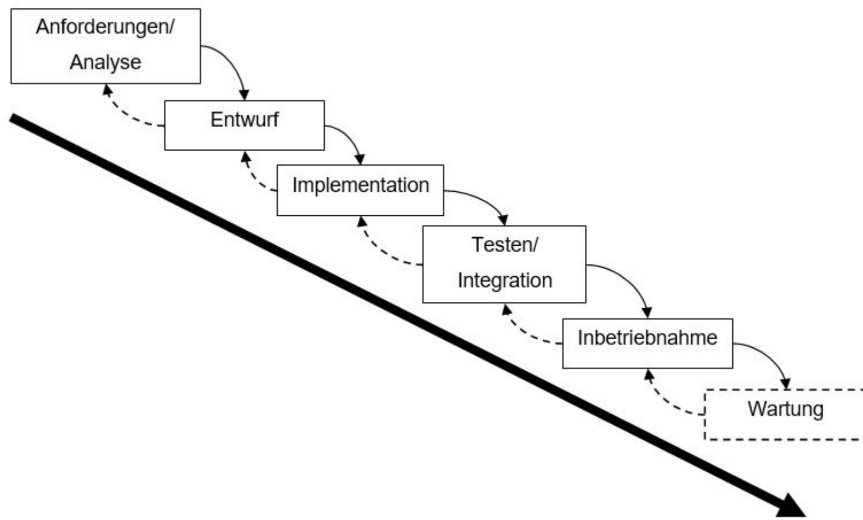

Abbildung 7.1: Wasserfallmodell

Wird das Modell mit der agilen Scrum-Entwicklung verglichen, so ist ein deutlicher Unterschied zu erkennen. Die agile Scrum-Methode ist kein sequentielles Verfahren. Wie in der Erklärung zu der agilen Entwicklung bereits beschrieben, besteht diese aus iterativen Schritten. Arbeitspakete werden aufgeteilt und solange bearbeitet bis sie den Anforderungen entsprechen. Dabei können theoretisch beliebig viele Iterationen durchlaufen werden. Anforderungen, welche zu Beginn der Planungsphase erstellt wurden, dürfen sich bei der Scrum-Entwicklung auch während der Produktentwicklung verändern. Es gibt also keine Phasen wie im Wasserfallmodell. Das agile Konzept beruht auf Anpassbarkeit und schließt daher die Phasenstruktur prinzipiell aus. Damit stellt sich die Frage inwiefern die gesammelten RTB auf die agile Entwicklung zutreffen.

Eine fehlende oder mangelhafte Spezifikation bzw. Architektur, betrifft auch die agile Entwicklung. In der agilen Entwicklung werden Anforderungen und Architektur im sogenannten Product Backlog erstellt. Auch wenn in der agilen Entwicklung nachträglich Änderungen bezüglich der Anforderungen und Architektur gemacht werden können, so entstehen dennoch sehr große Technische Schulden durch den großen Arbeitsaufwand. Somit ist diese Ursache des RTB in der agilen Entwicklung als hoch einzustufen.

Die Vernachlässigung von Anforderungen führt auch in der agilen Entwicklung zu großen Problemen. Werden falsche Prioritäten gesetzt, so kann es dazu kommen, dass wichtigen Anforderungen unerfüllt bleiben. Dies kann dann vorkommen, wenn ein wichtiges Arbeitspaket einer kleineren Gruppe gegeben wird, welche nicht auf die Umsetzung dieser Art von Funktion spezialisiert ist. Auch diese Art der Ursache wird aus diesem Grund als hoch eingestuft.

Das systematische Testen der Implementierung ist Teil eines Sprints. Finden diese Tests gar nicht oder nur teilweise statt, so kann es vorkommen, dass in dem abschließenden Review Fehler entdeckt werden, die im Nachhinein bearbeitet werden müssen. Die Erstellung von Testfällen gilt daher als grundlegendes Kriterium und wird deshalb auch als hoch eingestuft.

Die Entwicklung unter Zeitdruck stellt sich vor allem für die Scrum-Methode als Problem dar. Dadurch, dass ein Arbeitspaket oftmals mehrere Sprints durchläuft, kann ein hoher zeitlicher Aufwand entstehen. Die agile Entwicklung ist zudem deutlich anspruchsvoller und benötigt einen höheren Managementaufwand im Gegensatz zu dem Wasserfallmodell. Die Ursache Entwicklung unter Zeitdruck wird deshalb ebenfalls als hoch eingestuft.

Eine mangelhafte Anforderungssammlung kann in der agilen Entwicklung den Fertigungsprozess verlangsamen, da Anforderungen während der Entwicklung aufgenommen werden müssen. Im schlimmsten Fall ist der Kunde am Ende der Entwicklung nicht mit dem Produkt einverstanden, sodass dieses dann verbessert werden muss. Allerdings können Anforderungen jederzeit auch während der Entwicklung in das Product Backlog aufgenommen werden. Die Gefahr die von dieser Ursache ausgeht, wird deshalb als moderat eingestuft.

Fehlendes Interesse am Produkt kann in jeder Art von Entwicklung ein Problem sein. Es handelt sich hierbei nicht um eine Ursache, welche spezielle Auswirkungen auf die agile Entwicklung hat. Durch die täglichen Meetings werden diesbezüglich Defizite früh erkannt. Diese Ursache wird deshalb als niedrig eingestuft.

Die agile Entwicklung ist vom Prinzip her auf Änderungen während der Entwicklung ausgelegt. So können Anforderungen auch im Nachhinein im Product Backlog aufgenommen werden. Diese Ursache wird daher als niedrig eingestuft.

Im Gegensatz zu einer sequentiellen Methode kann in der agilen Entwicklung viel besser auf die Technik der Software, welche im Verlauf des Entwicklungsprozesses veraltet, reagiert werden. Dadurch, dass der ganze Prozess aus mehreren Iterationen besteht, können Verbesserungen oder Aktualisierungen viel einfacher vorgenommen werden. Dennoch ist dies mit in jeder Art der Entwicklung mit viel Arbeit verbunden und wird deshalb als moderat eingestuft.

Eine hohe Komplexität ist bei allen Entwicklungsprozessen ein Problem. Durch tägliche Meetings können in Scrum Probleme besprochen und zusammen gelöst werden. In der agilen Entwicklung müssen Probleme, nicht wie in der sequentiellen Methode, sofort gelöst werden. Anhand von Iterationen können Probleme Schritt für Schritt behoben werden. Die Ursache wird deshalb als moderat eingestuft.

Eine große Distanz zwischen dem aktuellen Stand und der optimalen Lösung ist anfangs überall vorhanden. Eine große Distanz gegen Ende des Projektes ist allerdings eine Katastrophe. Die Technische Schuld ist somit abhängig von dem jeweiligen Zeitpunkt. Die Distanz spielt in der Softwareentwicklung eine wichtige Rolle. Das Problem wird somit als hoch eingestuft.

Fehlende Ressourcen können in der agilen Entwicklung schnell zu einem vorzeitigen Ende der Entwicklung führen. Dadurch, dass es in Scrum oftmals zu vielen Sprints und dadurch zu zusätzlicher Arbeit kommt, können die Kosten und der Aufwand häufig nicht genau abgeschätzt werden. Wenn beispielsweise das Budget aufgebraucht ist, können keine weiteren Sprints durchgeführt werden. Fehlende Ressourcen werden somit als ein hohes Problem eingestuft.

Fehlendes Wissen und Bildung betrifft selbstverständlich jede Softwareentwicklung. Doch können durch die täglichen Meetings und Besprechungen Probleme frühzeitig besprochen und Wissen ausgetauscht werden. Die Ursache wird deshalb als moderat eingestuft.

Die Durchführung einer parallelen Entwicklung ist eine wesentliche Eigenschaft von Scrum. Durch die täglichen Meetings und dem Review, werden alle Entwickler über den aktuellen Stand

informiert. In anderen agilen Entwicklungsmethoden, kann eine parallele Entwicklung allerdings zu größeren Problemen führen. Die Ursache wird als niedrig eingestuft, da sie in der agilen Entwicklung sogar vorgesehen ist.

Die Unwissenheit über Patterns und Richtlinien ist ein Problem, welches oftmals nicht sofort erkannt wird. Oft tritt das dadurch entstandene Problem erst in der Wartung auf. Diese Ursache wird als hoch eingestuft, da sie in allen gängigen Softwareentwicklungen als sehr wichtig angesehen wird.

Entscheidungen ohne ausreichende Voraussicht können dazu führen, dass selbst in der agilen Entwicklung Probleme entstehen. Dies kann beispielsweise dazu führen, dass nutzlose Sprints durchgeführt werden, was dann zu lange Schleifen zur Folge hat, die den ganzen Entwicklungsprozess verlangsamen. Entscheidungen ohne ausreichende Voraussicht werden somit als hoch eingestuft.

Durch die Iterative Methode können Abkürzungen, welche später zu Problemen führen, auch im Nachhinein nochmal bearbeitet werden. Allerdings hängt dies wiederum von der Art und Größe der Abkürzung ab. Eingestuft wird diese Ursache als moderat.

Use-Cases gehören zu dem Design der Software, welches am Anfang des Projekts erstellt werden muss. Die Arbeit ohne Use-Cases erschwert den Entwicklern die Implementation. Ohne die Erstellung der Use-Cases ist nicht klar definiert, wie ein System auszusehen und zu funktionieren hat. Diese Art von Ursache wird deshalb als hoch eingestuft.

Die Unschlüssigkeit der Kunden führt in jedem Projekt zu Problemen. Auch in der agilen Entwicklung führt die Unschlüssigkeit dazu, dass die Software am Ende anders aussieht, als sie sich der Kunde vorgestellt hat. Auch ist während der Entwicklung nicht bekannt was nun als abgeschlossen gelten kann, und was nicht. Die Ursache wird deshalb als hoch eingestuft.

Auch eine schlechte und langsame Entwicklung kann in jedem Entwicklungsprozess vorkommen. Durch die täglichen Meetings kann in Scrum teilweise dagegengewirkt werden. Es ist hier-bei Aufgabe des Managers solche Entwicklungen zu erkennen und abzufangen. Die Ursache wird nur als niedrig eingestuft, die diese vor allem in Scrum gut abgefangen werden kann.

Anforderungen, welche sich im Verlauf der Entwicklung ändern, stellen für einen agilen Entwicklungsprozess keine großen Probleme dar. Der agile Prozess wurde genau aus diesem Grund entwickelt. Es können sich außerplanmäßige Zeitverschiebungen ergeben, allerdings können Änderungen immer aufgenommen werden. Die Gefahr wird deshalb als niedrig eingestuft.

Die gute Kommunikation zwischen Kunden und dem Softwareunternehmen spielt in vielerlei Hinsicht eine sehr wichtige Rolle. Zum einen können Anforderungen an das System besser angepasst, aber auch der Kunde über aktuelle Entwicklungsstände informiert werden. Es ist somit sehr wichtig, dass eine gute Kommunikation besteht, was diese Ursache als hoch einstuft.

Ein Projektmanager welcher nur am Gewinn interessiert ist, wird in der agilen Entwicklung kaum Erfolg haben. Ein Manager muss sich somit an einem agilen Prozess pro aktiv beteiligen und diesen führen. Dieses Problem wird somit als hoch eingestuft.

Falls Kunden keine genaue Beschreibung ihrer Anforderungen liefern, kann es zu Unschlüssigkeit darüber kommen, wann eine Anforderung abgeschlossen ist. In der iterativen bzw. agilen Entwicklung ist es allerdings möglich Anforderungen und Funktionen neu anzupassen. Die Ursache wird als niedrig eingestuft.

Ein schlechtes Design in der Planungsphase, führt dazu, dass sich die Entwickler nicht im Klaren darüber sind, wann ein Arbeitspaket als abgeschlossen gelten kann. Designänderungen können allerdings auch während der Entwicklung angepasst oder ergänzt werden. Deshalb wird diese Ursache nur als moderat eingestuft.

Das Fehlen einer systematischen Qualitätskontrolle führt dazu, dass vor allem im Endprodukt und in der Wartung Fehler und Probleme entstehen können. Die Prüfung der Qualität sollte somit in jedem Fall vorhanden sein. Die Ursache wird deshalb als hoch eingestuft.

Da die sequentielle Arbeitsweise in der agilen Entwicklung ausgeschlossen ist, wird sie aus der Bewertung entfernt.

Die folgende Tabelle fasst die Bewertungen der einzelnen Ursachen zusammen und stellt diese dem Vorkommen in den Quellen gegenüber.

Ursache	Vorkommen	Gefahreneinstufung
Fehlende oder Mangelhafte Spezifikation / Architektur	23	Hoch
Vernachlässigung von Anforderungen	13	Hoch
Mangelhaftes systematisches Testen	11	Hoch
Entwicklung unter Zeitdruck	9	Hoch
Mangelhafte Anforderungssammlung	7	Moderat
Fehlendes Interesse am Projekt	7	Niedrig
Kunde kommt in der späten Entwicklungsphase mit neuen Anforderungen	7	Niedrig
Die Technik der Software veraltet im Verlauf des Entwicklungsprozesses. Technische Evolution.	6	Moderat
Hohe Komplexität	6	Moderat
Große Distanz zwischen aktuellem Stand und der optimalen Lösung	5	Hoch
Fehlende Ressourcen	5	Hoch
Fehlendes Wissen / Bildung	4	Moderat
Durchführung einer parallelen Entwicklung	3	Niedrig
Unwissen über Patterns und Richtlinien	3	Hoch
Entscheidungen ohne Voraussicht	3	Hoch
Abkürzungen welche später zu Problemen führen	3	Moderat
Arbeit ohne Use-Cases erschwert die Erstellung einer Architektur	3	Hoch
Kunden sind sich unschlüssig über das was sie wollen	3	Hoch
Schlechte / Langsame Entwicklung	3	Niedrig
Anforderungen die sich im Verlauf der Entwicklung ändern	2	Niedrig
Schlechte Kommunikation zwischen Klienten und dem Software Unternehmen	2	Hoch
Projektmanager nur an Kosten und Gewinn interessiert	2	Hoch
Unschlüssigkeit, wann eine Anforderung abgeschlossen/erledigt ist	1	Niedrig
Schlechtes System Design in der Planungsphase	1	Moderat
Keine systematische Prüfung der Qualität	1	Hoch
Sequentielles Arbeiten (Wasserfallmodell)	1	-

Tabelle 7.4: Gefahreneinstufung der RTD in der agilen Entwicklung

Anhand der Tabelle ist zu erkennen, dass das Vorkommen der einzelnen Ursachen keinen Aufschluss darüber gibt, wie gefährlich die einzelnen Ursachen in der agilen Entwicklung sind. Allerdings kann davon ausgegangen werden, dass anhand der Häufigkeit der Ursachen in den Quel-len ein Zusammenhang mit dem Vorkommen der Ursachen in der derzeitigen Softwareentwick-lung besteht. Um einen genaueren Aufschluss über das Vorkommen zu geben, sollten allerdings mehrere hundert Quellen begutachtet werden, und eine Befragung verschiedener Softwareunternehmen vorgenommen werden.

Wir können in der Tabelle sehen, dass nicht alle Ursachen, welche allgemein häufig in der Softwareentwicklung vorkommen, auch in der agilen Entwicklung eine große Gefahr darstellen. Die Einstufung der Gefahrenstufen, fand in einer rein interpretativen Form statt und bezieht sich auf eine persönliche Einschätzung

7.3.2 Auswirkungen von Requirement Technical Debts auf die Sicherheit

Im Folgenden sei noch einmal die Zusammenfassung unserer Definition zum Thema Sicherheit vor Augen geführt:

"…System als unsicher gilt, wenn es nicht akzeptierbare Verluste verursacht. Verluste entstehen durch Unfällen, einer unbeabsichtigten Kette von Ereignissen, die zu Verlusten führt. Diese Ver-luste können Verletzungen, Schäden an Sachgütern und der Umwelt … "

Wir können erkennen, dass Technische Schulden im allgemeinen eine große Gefahr für die Sicherheit in der Softwareentwicklung darstellen. Verluste können in der Softwareentwicklung verschiedenartig aufgefasst werden. Zum kann es sich um Probleme handeln, welche der ganzen Entwicklung schaden. Es kann der Fall sein, dass Funktionen nicht implementiert wurden. Weiterhin kann es der Fall seinm dass viel Zeit durch durch die Verbesserung einer Technischen Schuld verloren ging. Verluste können auch als Anforderungen verstanden werden, welche bis zum Schluss hin nicht erfüllt werden konnten. Laut der Definition umfasst ein Verlust, was dem Softwaresystem oder der Umwelt Schaden zufügt. Ereignisse, welche zu diesen Verlusten füh-ren, sind die Ursachen von Technischen Schulden. Es ist somit ein klarer Zusammenhang zwi-schen den Sicherheitsaspekten in der Softwareentwicklung und den Technischen Schulden zu erkennen. In Bezug auf die agile Entwicklung können wir die Gefahreneinstufung somit auch als Grad der Gefahr, dass es zu einer Verletzung der Sicherheit kommt, ansehen. Die Zunahme der Schwachstellen in der Software führt zu einem Anstieg der Risiken [39]. Technische Schulden können als Qualitätsmaß angesehen werden. Dieser Qualitätsmaß kann wiederrum dafür ein Indiz sein, dass Sicherheitsmängel in der Software bestehen.

Wir führen uns dies anhand einer Abbildung noch einmal besser vor Augen:

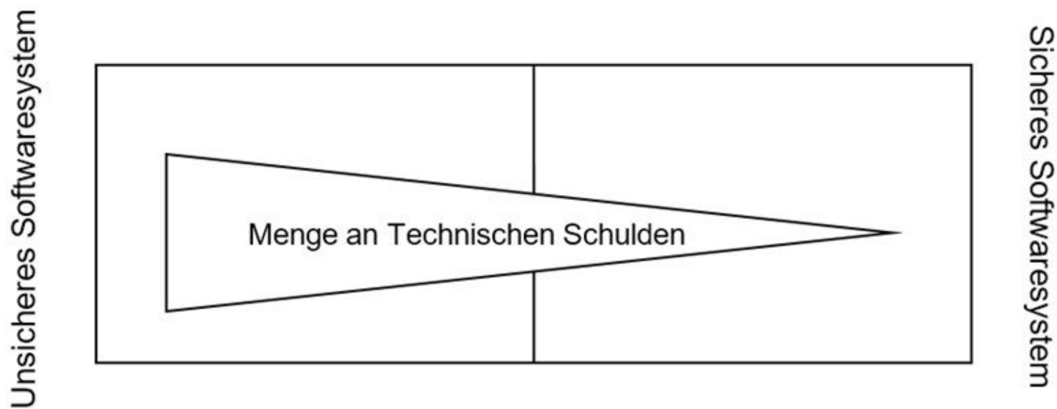

Abbildung 7.2: Technische Schulden und Sicherheit

Anhand der Abbildung 7.2 ist zu erkennen, welchen Einfluss Technische Schulden auf die Entwicklung eines Softwaresystems haben. Je nach Umfang an Technischen Schulden, befindet man sich weiter im Bereich des sicheren oder des unsicheren Softwaresystems.

7.4 Maßnahmen gegen Technische Schulden

In der Tabelle 7.5 werden die aus den Quellen gefundenen Maßnahmen gegen Technische Schulden aufgelistet. Unter den 43 Quellen, gab es 28, welche sich mit den Maßnahmen gegen Technische Schulden auseinandergesetzt und Vorschläge für eine Reduzierung der Technischen Schulden aufgezeigt haben. Insgesamt wurden 23 verschiedene Arten von Maßnahmen gefunden. Auch hier gibt die Anzahl die Häufigkeit der einzelnen Maßnahmen in den Quellen an. Die Maßnahmen wurden absteigend nach der Häufigkeit sortiert. Maßnahmen welche sich sehr stark ähneln wurden zu einem Punkt zusammengefasst.

Nr.	Maßnahme	Anzahl
1	Auflistung von Technischen Schulden mit Ranking	22
2	Regelmäßig geplante Refactorings	18
3	Verwendung von Tools zum Tracken von Technischen Schulden	13
4	Erstellung von Kostenanalysen/Modellen	10
5	Code Analysen	9
6	Weiterbilden der Mitarbeiter	9
7	Konstante Suche nach Technischen Schulden	9
8	Behandlungsphasen von Technischen Schulden	9
9	Verbesserung des Managements	7
10	Erstellung von detailreichen Architekturen und Prototypen	6
11	Durchführung von regelmäßigen Reviews	6
12	Verbesserte Testdurchführungen	5
13	Beachtung von Software Design Prinzipien	5
14	Verbesserung der Kommunikation	5
15	Erhöhte Wahrnehmung / Vorsicht bezüglich Technischen Schulden	4
16	Untersuchung auf Abweichung bezüglich der Architektur/Dokumentation	3
17	Untersuchungsteams Technischer Schulden	3
18	Flexibilität in der Softwareentwicklung erhöhen	2
19	Kontrolle der Anforderungen	2
20	Nutzung der freien Zeit für die Behandlung von Technischen Schulden	1
21	Behandlung von einer Technischen Schuld anstatt mehrere gleichzeitig	1
22	Erstellung von mehreren Definitionen von „Fertig“	1
23	Erstellung von Lösungspatterns	1

Tabelle 7.5: Maßnahmen zur Prävention von Technischen Schulden

Die Auflistung und Priorisierung der Technischen Schulden verschafft dem Entwickler einen Überblick über Gefährdungen, der sein Projekt möglicherweise ausgesetzt ist. Dieses sogenannte Portfolio der Technischen Schulden sollte ausfolgenden Punkten bestehen [19]:

- Standort der Technischen Schuld
- Zeitlicher Auftritt der technischen Schuld
- Verantwortliche Personen
- Grund, warum das Problem des TD einzustufen ist
- Schätzung des Ausmaßes durch den Manager oder Leiter

- Schätzung der Kosten
- Schätzung der Unsicherheiten der Kostenschätzung
- Ermittlung der Korrelation mit anderen TDs

Als wichtigste Maßnahme sollte während jeder Besprechung geklärt werden, was vorrangig behoben werden soll, und welche der Schulden später als nachrangig angesehen werden können. Die Liste der Technischen Schulden sollte jedem Entwickler und Stakeholder bekannt sein. In Scrum sollte ein spezielles, laufend aktualisierter, Backlog der Technischen Schulden angelegt werden [23]. Diese Methode eignet sich vor allem für Großprojekte, da deren Komplexität eine Übersicht über die verschiedenen Technischer Schulden erschwert. Des Weiteren wurde in den Quellen vorgeschlagen, die Software JIRA zu nutzen, um Tickets verschiedener Technischer Schulden anzulegen und diese allen zugänglich zu machen [24]. JIRA ist ein Entwicklungstool, welches sich vor allem für die agile Entwicklung eignet. Es handelt sich hierbei um eine Webanwendung für die Planung und Fehlerverwaltung/Problembehandlung von Softwareentwicklungen [25]. Nachdem ein TD aus der Liste beseitigt wurde, muss er aus der Liste der Technischen Schulden entfernt werden [13]. Eine regelmäßige Kontrolle und Aktualisierung der noch abzuarbeitenden Schulden muss gewährleistet sein.

Als zweithäufigster Punkt, empfiehlt sich, regelmäßige Refactorings durchzuführen. Das Refactoring bezieht sich hauptsächlich auf die Codequalität. Es können aber auch Änderungen und Korrekturen bezüglich des Designs und der Architektur des Softwaresystems vorgenommen werden um die Qualität zu verbessern [23]. Dies betrifft auch die Behebung von Technischen Schulden [23]. Refactorings können dabei helfen, Probleme zu finden, welche zuvor übersehen wurden. Code Abschnitte können erneuert oder abgeändert werden. Die Durchführung von Refactorings führt zu einer Verbesserung der Lesbarkeit und zu einem besseren Verständnis des Codes. Refactorings ersetzen keine Testfälle, helfen aber dabei eine erfolgreiche Testdurchführung zu vollziehen. Refactorings sollten dabei in regelmäßigen Abständen vorgenommen werden. Auch ist es sinn-voll, das Refactoring auf bestimmte Bereiche mit Technischen Schulden zu beschränken [26]. Die Auswahl kann anhand des zuvor definierten Portfolios geschehen.

Als dritte Maßnahme wäre die Verwendung von Tools zum Tracken von Technischen Schulden zu nennen. Im Folgenden werden einige Tools der Quellen kurz vorgestellt:

DebtFlag Tool [27]: Das DebtFlag Tool wurde extra für Java erstellt und besteht aus zwei Bestandteilen, einem Eclipse-Plugin und einer Web-Applikation. Das Tool findet Technische Schulden nach bestimmten vorgegebenen Regeln und Abhängigkeiten. Des Weiteren ist es möglich mit dem Tool die Schulden zu managen.

COCOMO[28]: COCOMO steht für Constructive Cost Model und dient der Veranschaulichung von Kosten, Aufwand und einem Zeitplan. Anhand von Algorithmen erstellt COCOMO eine Abschätzung dieser Aspekte. Das Tool eignet sich nicht um Technische Schulden im Code zu finden, sondern nur um Schulden in Form von Ressourcen und Komplexität zu veranschaulichen.

Ndepend [29]: Ndepend ist eine Visual Studio extension für.NET, welches den Programmiercode auf Code Smells wie unbenutzten toten Code, API-Änderungen, schlechte OOP-Nutzung und verstrickten Code untersucht. Die Software bezieht sich dabei auf über 100 Regeln.

Neben diesen Tools werden unzählige weitere Tools angeboten, welche z.B. Java Code auf Code Smells untersuchen. Die Methode eignet sich besonders um frühzeitig grobe Fehler zu erkennen

und beheben zu können. Es kann hiermit eine Metrik zur Analyse der Softwarequalität geschaffen werden.

Eine weitere Maßnahme zur Prävention von Technischen Schulden ist die Erstellung von Kosten-Analyse-Modellen. Wie im vorherigen Punkt schon beschrieben wurde, können Tools verwendet werden, um solche Modelle zu erstellen. Des Weiteren können solche Modelle helfen eine Beurteilung gegenüber Technischen Schulden darzustellen. Es kann geklärt werden, ob es sich lohnt eine bestimmte Schuld zu beheben, und welche Kosten die Behebung oder das Ignorieren der Schuld mit sich bringt [19]. Die spezifische Planung der Ressourcenverteilung kann in der Entwicklung helfen, frühzeitig TD einzuschränken [23]. Es sollte eine gewisse Voraussicht bei der Planung bestehen, so können bestehende und zukünftige Kosten besser eingeplant werden. Das Management sollte die Kosten verschiedener Phasen vor allem in SCRUM schon frühzeitig erkennen.

Um TD einzuschränken sollten regelmäßige Codeanalysen durchgeführt werden. Diese können mit den schon beschriebenen Tools oder durch die Entwickler selbst erfolgen. Auf folgendes sollte dabei geachtet werden:

- Code Regeln / Patterns
- Code Smells
- „Spaghetti Code“
- Code Duplikate
- Toter Code
- Übersichtlichkeit der Abhängigkeiten

Ein weiterer wichtiger Punkt ist die Weiterbildung der Mitarbeiter. Durch Schulungen sollten Mitarbeiter auf verschiedene Arten von Technischen Schulden vorbereitet werden. Außerdem sollten Coding-Standards und Coding-Guides erläutert werden [24]. Es sollten Übungen zur Erkennung von Technischen Schulden und zum Refactorings durchgeführt werden [26]. Oftmals ist Mitarbeitern gar nicht bekannt was Technische Schulden sind. Die Weiterbildung sollte sich hierbei weniger auf den Code beziehen, sondern viel mehr auf die Designerstellung [18].

Es empfiehlt sich, eine ständig, projektbegleitende Suche nach Technischen Schulden vorzusehen, um größere Anhäufungen zu vermeiden. Jeder Entwickler sollte seine Aufgaben und Abschnitte selbst überprüfen. Nebenbei können Tools helfen um die TD Suche aktuell zu halten. Der Nachteil dieses Verfahrens ist der zusätzliche Aufwand, bei dem jedoch die Chance besteht, dass er sich am Ende auszahlt, indem er TD frühzeitig zu vermeiden hilft.

Eine weitere Methode ist die Einführung von Behandlungsphasen von Technischen Schulden. Hierbei werden regelmäßige Zeitabschnitte definiert, in denen TD beseitigt werden [18]. Diese Variante ist nicht so zeitaufwändig wie die vorherige Methode, bringt allerdings die Gefahr einer Anhäufung von Technischen Schulden.

Ob und wie Technische Schulden behandelt oder wie diese behandelt werden, hängt oft vom Management ab. Laut der Quelle [30] hatten 65 Prozent aller Beteiligten kein definiertes TD Manage-ment. 25 Prozent der Befragten löste TD auf der Team-Ebene. Auch liegt es in der Verantwortung des Managements, dass das Team in einer kontrollierten und organisierten Weise arbeitet

[19]. Dazu gehört, ein gutes Arbeitsklima zu begünstigen und die Kommunikation zwischen den Entwicklern zu fördern. Grundsätzlich bestimmt das Management mit welcher Methode ein System entwi-ckelt wird.

Die Erstellung einer guten und detailreichen Architektur oder eines geeigneten Prototyps, erleichtert Entwicklern die Erstellung eines Systems. Kunden können anhand des Prototyps schon früh entscheiden ob Änderungen vorgenommen werden müssen [31]. Des Weiteren kann durch die Architektur und den Prototyp frühzeitig erkannt werden, ob Technische Schulden vorhanden sind [18].

Die Durchführung von regelmäßigen Reviews haben in der Entwicklung einen positiven Effekt in Bezug auf Technische Schulden [32]. Durch die regelmäßigen Treffen wird die Kommunikation untereinander gefördert und auf die entscheidenden Themenkategorien gelenkt:

- Derzeitiger Stand des Entwicklungsteams /Entwicklers
- Priorität von Aufgaben
- Schätzung des zeitlichen Aufwands der Aufgabe
- Schätzung der Kosten (Manager)
- Probleme die während der Entwicklung auftraten
- Verantwortlichkeiten
- Austausch an Informationen / Tipps

Während der Entwicklung sollte ein System regelmäßig getestet werden. Die Durchführung der Tests, sollte so genau und umfassend wie möglich vollzogen und protokolliert werden, so dass die Entwicklung auch später noch nachvollzogen werden kann. Es erweist sich manchmal auch als sinnvoll, Systemtester zu benennen, welche sich nur mit dem Überprüfen des Systems aus-einandersetzen. Der Test sollte nie von dem Entwickler allein durchgeführt werden. Neben den üblichen Unit-Tests gibt es in Java und auch in anderen Programmiersprachen Tools die das Testen eines Systems erleichtern.

Die Einhaltung von Coding-Richtlinien ist nach Quelle [32] einer der positivsten Maßnahmen. Es ist somit wichtig die Entwickler auf die geltenden Richtlinien hinzuweisen und regelmäßige Überprüfungen durchzuführen. Die Verletzung von Richtlinien ist in der Entwicklung häufig mit Technische Schulden verbunden [33]. Anhand von Schulungen kann das Einhalten der Richtlinien trainiert werden.

Der Leiter des Softwareteams sollte immer darauf achten, dass ein freundlicher Umgang und ein gutes Arbeitsklima untereinander herrscht. Alle Mitglieder in einem Softwareteam sollten dazu ermutigt werden, Probleme offen zu benennen und mit den anderen zu besprechen [34]. Ein Problem ist, dass Entwickler Fehler oder Probleme nicht eingestehen wollen, da sie Angst um ihre Position haben. Dabei ist entscheidend, dass der sachliche Aspekt der Entwicklung im Vordergrund steht, und die Tendenzen zu persönlichen Profilierungen einzelner weitgehend vermieden wird. Die Schilderung von Problemen hilft, da andere Entwickler oftmals wissen wie das Problem gelöst werden kann [34]. In diesem Zusammenhang sollte klar sein, dass Fehler nicht nur unvermeidbare Begleiterscheinungen menschlicher Tätigkeiten sind, sondern oft sogar eine Notwendigkeit für Weitentwicklung und Fortschritt darstellen. Darüber hinaus sollte immer

eine gute Kommunikation zwischen dem Kunden und dem Entwickler bestehen, um frühzeitige Anpassungen vornehmen zu können.

Um das Auftreten Technischer Schulden zu verringern, müssen diese als Defizite wahrgenommen werden. Dies kann durch Aufklärung und Schulungen erreicht werden. Entwickler müssen darüber informiert sein, was Technische Schulden sind, und was sie bewirken können, wenn man sie ignoriert [13].

Das Softwaresystem sollte immer auf Abweichungen bezüglich der Dokumentation und der Architektur überprüft werden. Diese bilden nämlich eine Grundlage dafür da, wie das System am Ende auszusehen hat. Selbstverständlich besteht am Anfang noch eine große Abweichung, welche sich gegen Ende allerdings verringern sollte.

Die Einführung von sogenannten Untersuchungsteams oder auch „TD Swat-Teams“ dient dem Auffinden von Technischen Schulden. Die Gruppe sollte am besten von einem erfahrenen Softwarearchitekten geführt werden. Das Team konzentriert sich auf die Tests und den Aufbau der Software [35]. Gefundene Technische Schulden werden durch das TD Swat-Team behoben.

Um zeitliche Verschiebungen zu vermeiden, sollte die Softwareentwicklung flexibel gestaltet werden. Es sollte genug Zeit für die jeweiligen Phasen eingeplant und Zeitdruck vermieden werden. Die Möglichkeit Technischer Schulden sollten im Voraus eingeplant werden.

Bestehende Anforderungen sollten regelmäßig auf den aktuellen Stand geprüft werden. Es muss sichergestellt werden, dass keine der Anforderungen vergessen oder permanent aufgeschoben wird.

Während der Entwicklung sollte der ungenutzte Puffer dazu verwendet werden um Technische Schulden zu suchen. Technische Schulden sollten einzeln behandelt werden. Die Behandlung von mehreren TD gleichzeitig kann dazu führen, dass einzelne Ursachen nicht vollständig oder umfassend behoben werden.

Es sollte in der Anforderungsphase genau spezifiziert werden, ab wann eine Anforderung als abgeschlossen gilt. Es ist möglich mehrere Definitionen von „fertig“ zu erstellen, um aktuelle Stände genauer im Auge zu behalten.

Es können Protokolle bezüglich der gefundenen Technischen Schulden und deren Lösungen erstellt werden. Anhand dieser Protokolle können Patterns geschrieben werden, welche in den nächsten Projekten wieder hilfreich sein können.

Die folgende Tabelle weist die gefundenen Ursachen von Technischen Schulden und Requirement Technical Debts den gefundenen Maßnahmen zu. Dabei die Nummer der jeweiligen Ursache aus den vorherigen Tabellen einer Maßnahme zugeordnet.

Nr.	Maßnahme	TD Nr.	RTD Nr.
1	Auflistung von Technischen Schulden mit Ranking	allgemein	allgemein
2	Regelmäßig geplante Refactorings	1, 2, 3, 13, 18, 21, 23, 26, 27, 28, 32, 33	1, 3, 8, 25
3	Verwendung von Tools zum Tracken von Technischen Schulden	2, 3, 13, 18, 26, 27, 28, 31	3, 25
4	Erstellung von Kostenanalyse-Modellen	10, 37	11
5	Code Analysen	s.3	s.3
6	Weiterbilden der Mitarbeiter	1, 3, 4, 5, 18, 22, 26, 30, 31, 32, 35	1, 3, 5, 8, 12, 14, 24
7	Konstante Suche nach Technischen Schulden	allgemein	allgemein
8	Behandlungsphasen von Technischen Schulden	allgemein	allgemein
9	Verbesserung des Managements	1, 3, 4, 5, 11, 15, 16, 25, 30, 31, 35, 37	1, 3, 5, 12, 15, 17, 19, 21, 22, 25, 26
10	Erstellung von detailreichen Architekturen und Prototy-pen	1, 7, 8, 25, 30	1, 5, 18, 19, 21, 24

Tabelle 7.6: Eine Zuordnung der Maßnahmen den Technischen Schulden 1

11	Durchführung von regelmäßigen Reviews	allgemein	allgemein
12	Verbesserte und geregelte Testdurchführungen	3, 18, 31	3, 25
13	Beachtung von Software Design Prinzipien	26, 32, 33	9, 14
14	Verbesserung der Kommunikation	11, 14, 16, 22, 36	2, 5, 15, 18, 21, 23
15	Erhöhte Wahrnehmung / Vorsicht bezüglich Technischen Schulden	allgemein	allgemein
16	Untersuchung auf Abweichung bezüglich der Architektur/Dokumentation	1, 4, 18, 25, 34	2, 10, 19, 23
17	Untersuchungsteams Technischer Schulden	allgemein	allgemein
18	Flexibilität in der Softwareentwicklung erhöhen	6, 9, 14, 19,	4, 7, 11, 16, 20
19	Kontrolle der Anforderungen	7, 10, 34	2, 5, 20, 23
20	Nutzung der freien Zeit für die Behandlung von Technischen Schulden	allgemein	allgemein
21	Behandlung von einer Technischen Schuld anstatt mehrere gleichzeitig	allgemein	allgemein
22	Erstellung mehrerer Definitionen von „Fertig“	1, 4, 6, 7, 11, 30	1, 3, 5, 10, 23, 24
23	Erstellung von Lösungspatterns	2, 8,11, 36	9, 14, 19

Tabelle 7.7: Eine Zuordnung der Maßnahmen den Technischen Schulden 2

Es handelt sich in dieser Zuweisung nur um einen Vorschlag. Einzelne Maßnahmen können teilweise auch für andere Arten von Ursachen verwendet werden. In der obigen Tabelle werden Ursachen den Maßnahmen zugeordnet, welche sich speziell auf diese Arten von Ursachen beziehen. Maßnahmen, welche auf keine spezifischen Ursachen gerichtet sind, wurden mit „allgemein“ vermerkt.

Anhand der Tabelle ist zu erkennen, dass schon allein diese 23 vorgeführten Maßnahmen eine breite Gruppe von Ursachen behandeln oder einschränken. Die Tabelle liefert somit einen Ansatz zur Problemlösung in der Softwareentwicklung. Im Folgenden wird erklärt, warum Ursachen ausgerechnet diesen Maßnahmen zugeordnet wurden.

8 Diskussion

Die hier durchgeführte SLR mit 43 Quellen liefert eine Übersicht bezüglich der Technischen Schulden. Mit einer größeren Anzahl von Quellen, ließe sich auch ein detaillierteres Ergebnis erreichen. Doch kann man schon in dieser Arbeit erkennen, welche Ursachen häufig zu Technischen Schulden oder Requirement Technical Debts führen. Wie erwartet spielen vor allem die Architektur, Spezifikation, Dokumentation, Tests, Anforderungsanalysen und der Zeitdruck eine große Rolle. Diese Punkte legen in der jeder Softwareentwicklung ein Grundgerüst fest und sind immanenter Bestandteil größerer Projekte. Eine bei den genannten Punkten führt wie erwartet zu Technischen Schulden. Des Weiteren haben die Ergebnisse bestätigt, dass bei vielen Entwicklern und Managern ein unzureichendes Bewusstsein bezüglich Technischer Schulden besteht. Anhand der Tabellen ist zu erkennen, dass das Vorkommen der einzelnen Ursachen keinen Aufschluss darüber gibt, welche Folgen die einzelnen Ursachen in der Softwareentwicklung haben. Allerdings deutet die Häufigkeit der Nennungen der Ursachen in den Quellen, auf große Relevanz in der Softwareentwicklung hin. Die Ergebnisse zeigen, auf welche Ursachen ein besonderes Augenmerk gerichtet werden muss. Die Maßnahmen, welche der SLR entnommen wurden, bieten einen ersten Ansatz für die Prävention Technischer Schulden.

Anhand der Ergebnisse dieser Arbeit sowie einer Befragung verschiedener Softwareunternehmen bezüglich der Gefahr von Technischen Schulden, kann beurteilt werden, ob die Häufigkeit von TD in den Quellen in einer Beziehung zu ihrem Gefährlichkeitsgrad stehen.

Technische Schulden sind von Beginn an in jedem Softwareprojekt vorhanden. Die Distanz zwischen dem aktuellen Stand und dem fertigen Produkt kann als TD angesehen werden. Auch ist es unmöglich, eine Software zu entwickeln, ohne in irgendeinem Bereich Fehler zu machen. Es ist nur menschlich, dass während der Entwicklung Probleme aufgrund von Unwissen oder anderen Faktoren entstehen

Zu beachten ist dass auch diese Arbeit die Frage nach dem Umgang mit Technischen Schulden nach wie vor offen lässt. TD können aufgrund von unzählig verschiedenen Gründen entstehen. Es kann keine einzelne optimale Maßnahme für alle TD genannt werden.

Zudem kann in dieser Arbeit nicht auf alle Arten von Technischen Schulden eingegangen werden, da dies den Aufwand sprengen würde. In dieser Arbeit bleiben aus diesem Grund Ursachen und Maßnahmen unbeachtet, welche nur in einer der Quellen genannt wurden. Dabei sollte bewusst bleiben, dass auch einzelne hier nicht beachtete Technische Schulden die Sicherheit in einem Softwareprojekt gefährden.

Es ist zu beachten, dass diese Arbeit auf einer Systematic Literature Review besteht. Die Ergebnisse wurden somit nur aus den 43 Quellen extrahiert. Es wurde sich in dieser Arbeit nur auf die Technischen Schulden der Softwareentwicklung bezogen.

Eine Empfehlung für die weitere Forschung ist daher, ähnliche Studien durchzuführen, um zu ermitteln, ob eine Beziehung besteht zwischen dem Vorkommen von Technischen Schulden in

einer SLR und dem in Softwareunternehmen. Dadurch könnte auch die von diesen Schulden ausgehende Gefährdung genauer ermittelt werden.

9 Zusammenfassung und Ausblick

Eine einheitliche Definition der Requirement Technical Debts existiert gegenwärtig nicht. Allgemein ist jedoch festzustellen, dass unter dem Begriff Technische Schulden Kompromisse zusammengefasst werden, die bezüglich der Anforderungsspezifikation eingegangen werden und welche Auswirkungen auf die spätere Systemimplementation oder das fertige Produkt haben können. Die agile Entwicklung zielt darauf ab, Software schnell und flexibel zu entwickeln. Änderungen die während der Entwicklung gemacht wurden, sollen keine Hindernisse für das Projekt darstellen. Die agile Entwicklung beruht daher auf weitestgehende Anpassbarkeit. In der agilen Ent-wicklung wird oft auf die iterative Methode zurückgegriffen. Scrum ist gegenwärtig einer der bekanntesten Verfahren, welches iterative Methode verwendet und mit kleinen selbstorganisierenden Teams arbeitet. Ein System gilt als unsicher, wenn es nicht akzeptierbare Verluste verursacht. Verluste entstehen durch Unfälle, das heißt durch eine unbeabsichtigte Kette von Ereignissen, die zu Verletzun-gen, Schäden an Sachgütern und der Umwelt oder ein Verlust von Leben führen. Technische Schulden gelten als einer der Ursachen, welche zu Verlusten führen können.

Die erste Frage der vorliegenden Arbeit beschäftigt sich mit der Identifizierung von Ursachen, die zu Technischen Schulden führen. Es konnte dabei herausgefunden werden, dass mangelhaftes Design oder Architektur, zu den häufigsten Ursachen gezählt wird. Bei der Identifizierung von Technischen Schulden konnte erkannt werden, dass 33 von 43 Quellen diese als Ursache benannten. Die zweithäufigste Ursache war der Mangel an Achtsamkeit. Viele Quellen nannten diesen Punkt als Ursache von Code Problemen und Defekten. Von den 43 Quellen klassifizierten 18 diese Ursache als TD. Auch ungenügende Testabdeckung wurde 18-mal genannt. Durch einen Mangel an Testfällen, können Fehler unentdeckt bleiben und später zu Problemen führen. Der vierthäufigste Punkt, ist eine fehlerhafte oder mangelhafte Dokumentation und der fünfthäufigste die fehlende Bildung und Schulungen bezüglich Technischen Schulden. Dies sind allerdings nur die fünf häufigsten Ursachen, die in den Quellen genannt wurden. Technische Schulden können darüber hinaus durch viele weitere Einflüsse entstehen. Dazu gehören der Zeitdruck, Änderungen während der Entwicklung, falsche Entwicklungsstrategien und eine mangelhafte Qualität des Quellcodes.

Der zweite Abschnitt der vorliegenden Arbeit beschäftigt sich mit der Frage, welche Technischen Schulden als Requirement Technical Debts identifiziert werden können. Es wurden dabei nur die Quellen analysiert, welche sich mit dem Begriff Requirement Technical Debts auseinandergesetzt haben. Auch hierbei stand die Architektur/Spezifikation mit 23 Nennungen in 25 Literaturen an obererster Stelle. An zweiter Stelle stand die Vernachlässigung von Anforderungen. Insbesondere durch falsche Priorisierung von Anforderungen. In den 25 Quellen wurde diese Ursache 13-mal genannt. Das mangelhafte Testen des Systems tritt auch hier wieder auf. Die Tests werden deshalb den Requirement Technical Debts zugeteilt, da die Durchführung von Testfällen eine Anforderung darstellen die zu Beginn des Projekts eingeplant werden muss. An vierter Stelle stand der zeitliche Druck, welcher dazu führt, dass TD nicht oder nur unvollständig behoben

werden können. Eine weitere wichtige Ursache stellt die mangelhafte Anforderungssammlung dar. Die Anforderungssammlung bietet ein Grundgerüst für die Softwareentwicklung. Dazu gehören nicht nur Wünsche der Kunden, sondern auch Architekturen, Designs und der Entwicklungsplan. Doch auch dies sind nur einige der Ursachen des RTD. Ressourcen, Bildungslücken, Komplexität, Kommunikationsprobleme und auch das Management selbst können Gründe für RTB sein.

In Bezug auf die agile Entwicklung haben vor allem die ersten vier Punkte einen hohen Einfluss auf eine Gefährdung der Sicherheit. Doch auch fehlende Ressourcen, die Arbeit ohne die Erstellung von Use-Cases oder dergleichen, Unwissenheit über Patterns, Entscheidungen ohne aus-reichende Voraussicht, Unschlüssigkeit der Kunden, schlechte Kommunikation und fehlende systematische Prüfung der Qualität stellen eine hohe Gefahr in der agilen Entwicklung dar.

Der dritte und letzte Bereich der Arbeit beschäftigt sich mit der Forschungsfrage, was gegen Technische Schulden unternommen werden kann, beziehungsweise inwieweit eine Prävention möglich ist. Nach der SLR umfassen die fünf am häufigsten genannten Vorschläge zur Vermeidung von TD die übersichtliche Auflistung von Technischen Schulden, die regelmäßigen Durchführung von Refactorings, die Verwendung von Tools zum Tracken von Technischen Schulden, das Erstellen von Kostenanalysemodellen und Code Analysen.

Es sollte bewusst bleiben, dass Technische Schulden in jedem Softwareprojekt vorhanden sind. Eine vollständige Prävention wird nie möglich sein. Es müssen jedoch Maßnahmen ergriffen werden, die das Ausmaß an Technischen Schulden bilden. Die genannten Maßnahmen gegen Technische Schulden bilden eine gute Ausgangsbasis und bieten eine Checkliste darüber, auf was in einer Entwicklung geachtet werden muss.

Literaturverzeichnis

[1] Stephan Augsten. „Was ist agile Softwareentwicklung?“. 04.01.17. Url: http://www.dev-insider.de/was-ist-agile-softwareentwicklung-a-569187/ (visited 14.08.17)

[2] Heitor Roriz Filho, Ilja Preuss, Marc Bless, Fabian Ehls, Stefan Roock, Jutta Eckstein, Stefan Hoehn, Bernd Schiffer. “Manifest für Agile Softwareentwicklung“. 2001. Url: http://agilemanifesto.org/iso/de/manifesto.html (visited 14.08.17)

[3] Des Greer, Yann Hamon. „Agile Software Development”. 7.Juli 2011. In: Journal of Software: Practice and Experience

[4] Alexander Kriegisch. „Agiles Projektmanagement“. Url: Scrum-master.de (visited 14.08.17)

[5] Donald Firesmith, Software Engineering Institute, U.S.A. “Engineering Safety Requirements, Safety Constraints, and Safety-Critical Requirements”. In: Journal of Object Technology, Vol. 3, No. 3, March-April 2004

[6] Menfred Broy.” Funktionale Sicherheit ISO 26262 Schwerpunkt Requirement Engineering“. 18.11.2009. Url: https://de.slideshare.net/ProjectSymphony/software-requirements-for-safetyrelated-systems (visited 15.08.17)

[7] Prof. Lars Grunske, Universität Stuttgart. “Vorlesung Sichere und Zuverlässige Softwaresysteme“. 16.03.15

[8] W. Cunningham, “The wycash portfolio management system,”.1992. In: Proceedings of the International Conference on Objectoriented Programming Systems, Languages, and Applica-tions (Addendum), ser. OOPSLA’92. ACM, pp. 29–30.

[9] Zahra Shakeri Hossein Abad, Guenther Ruhe, “Using Real Options to Manage Technical Debt in Requirements Engineering”. 2015. In: Requirements Engineering Conference (RE), 2015 IEEE 23rd International, IEEE

[10] K. Schmid, “On the limits of the technical debt metaphor some guidance on going beyond”. 2013. In: Proceedings of the 4th International Workshop on Managing Tech-nical Debt (MTD), pp. 63–66. ACM

[11] N. Alves, L. Ribeiro, V. Caires, T. Mendes, and R. Spinola, “Towards an ontology of terms on technical debt”. 2014. In: Proceedings of the 6th International Workshop on Managing Technical Debt (MTD), 2014, pp.1–7. IEEE

[12] Ralf Reussner, Wilhelm Hasselbring, KIT. „Arbeitskreis Handbuch“. Url: http://www.handbuch-softwarearchitektur.de/ . 2008. (visited 28.10.17)

[13] Girish Suryanarayana, Ganesh Samarthyam, Tushar Sharma. “Refactoring for Software Design Smells – Managing Technical Debt”. 2014. In: Morgan Kaufman, erste Auflage

[14] Yuepu Guo, Rodrigo Oliveira Spínola, Carolyn Seaman. “Exploring the costs of technical debt management – a case study”. 2016. In: Journal Empirical Software Engineering archive Volume 21 Issue 1, February 2016 Pages 159-182

[15] AretiAmpatzoglou, Apostolos Ampatzoglou, AlexanderChatzigeorgiou, ParisAvgeriou. “The financial aspect of managing technical debt: A systematic literature review”. 2014. In: Infor-mation and Software Technology Volume 64, August 2015, Pages 52-73. Elsevier

[16] Md Abdullah Al Mamun, Christian Berger, Jörgen Hansson . “Explicating, Understanding, and Managing Technical Debt from Self-Driving Miniature Car Projects”. 2014 In: Managing Technical Debt (MTD), 2014 Sixth International Workshop on. IEEE

[17] Edith Tom, Aybuke Aurum, Richard Vidgen .”A CONSOLIDATED UNDERSTANDING OF TECHNICAL DEBT”. 2012. In: European Conference on Information Systems (ECIS)

[18] Antonio Martini, Jan Bosch, Michel Chaudron. “Architecture Technical Debt: Understanding Causes and a Qualitative Model”. 2014. In: Software Engineering and Advanced Applications (SEAA), 2014 40th EUROMICRO Conference on, IEEE

[19] Nicolli S.R. Alves, Thiago S. Mendes, Manoel G. de Mendonça, Rodrigo O. Spínola, Forrest Shulle, Carolyn Seaman “Identification and management of technical debt: A systematic map-ping study”. 2016. In: Information and Software Technology Volume 70, February 2016, Pages 100-121, Elsevier

[20] Software Engineering Group (Keele University), Department of Computer Science (Univer-sity of Durham). “Guidelines for performing Systematic Literature Reviews in Software Engi-neering”. 9 July, 2007. In: EBSE Technical Report EBSE-2007-01. Elsevier

[21] Nico Zazworka, Michele A. Shaw, Forrest Shull, Carolyn Seaman. “Investigating the Impact of Design Debt on Software Quality”. 2011. In: MTD ’11 Proceedings of the 2nd Workshop on Managing Technical Debt Pages 17-23. ACM

[22] Prof. Dr. Klaus-Peter Fähnrich, Institut für Informatik Betriebliche Informationssysteme, Universität Leipzig. „Vorlesung Softwaretechnik , Vorgehensmodelle, V-Modell XT“, WS 2005/2006

[23] Zengyang Li, Paris Avgeriou, Peng Liang. “A systematic mapping study on technical debt and its management”. 2015. In: Journal of Systems and Software Volume 101, March 2015, Pages 193-220. Elsevier

[24] Jesse Yli-HuumoAndrey MaglyasKari Smolander. “The Sources and Approaches to Management of Technical Debt: A Case Study of Two Product Lines in a Middle-Size Finnish Soft-ware Company”. 2014. In: International Conference on Product-Focused Software Process Im-provement PROFES 2014: Product-Focused Software Process Improvement pp 93-107. Sprin-ger

[25] ATLASSIAN. ”Jira Software”. Url: https://de.atlassian.com/software/jira visited(14.11.2017)

[26] Antonio Martini, Jan Bosch, Michel Chaudron. “Investigating Architectural Technical Debt accumulation and refactoring over time: A multiple-case study”. 2015. In: Journal Information and Software Technology Volume 67 Issue C, November 2015 Pages 237-253. ACM.

[27] Johannes Holvitie,Ville Leppänen. “DebtFlag: Technical Debt Management with a Development Environment Integrated Tool”. 2013. In: MTD 2013, San Francisco, CA, USA, TUCS

[28] Dr. Barry W. Boehm,Mr. A. Winsor Brown, Dr. Brad Clark, Dr. Ray Madachy, Mr. Don Reifer, Dr. Bert Steece, Dr. Ye Yang, Mr. Ali Malik, Mr. Thomas Tan, Mr. Vu Nguyen. “COCO-MO II”. In: Center for Systems and Software Engineering, USC (visited 07.11.2017)

[29] ndepend. https://www.ndepend.com/, (visited 07.10.2017)

[30] Neil A. Ernst, Stephany Bellomo, Ipek Ozkaya, Robert L. Nord, and Ian Gorton. “Measure It? Manage It? Ignore It? Software Practitioners and Technical Debt”. 2015. In: Proceeding ESEC/FSE 2015 Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engi-neering Pages 50-60. ACM

[31] Eric Allma. "Managing Technical Debt. Shortcuts that save money and time today can cost you down the road”. 2012. In: acmqueue Volume 10, issue 3, March 23, 2012

[32] Johannes Holvitie, Ville Leppänen, Sami Hyrynsalmi. “Technical Debt and the Effect of Ag-ile Software Development Practices on It - An Industry Practitioner Survey”. 2014. In: Managing Technical Debt (MTD), 2014 Sixth International Workshop on

[33] Dr. Nico Zazworka, Dr. Carolyn Seaman. “Identifying and Managing Technical Debt” 2012. Url: https://de.slideshare.net/zazworka/identifying-and-managing-technical-debt (visited 20.10.17)

[34] Vinay Krishna, Dr. Anirban Basu. “Software Engineering Practices for Minimizing Technical Debt”. 2013. In: Conference: SERP13. ResearchGate

[35] Israel Gat, John D. Heintz . „From Assessment to Reduction: How Cutter Consortium Helps Rein in Millions of Dollars in Technical Debt”. 2011. In: Proceeding MTD ’11 Proceedings of the 2nd Workshop on Managing Technical Debt Pages 24-26

[36] N. Ernst, “On the role of requirements in understanding and managing technical debt”. 2012. In: 3rd International Workshop on Managing Technical Debt (MTD ‘12), IEEE Computer Society, pp. 61–64, Zurich, Switzerland, 2–9 December 2012.

[37] Marie Claire Dzukou, Natali Brozmann, HTW Aalen. „Grundprinzipien der agilen Software-entwicklung“. 2014/15. Url: http://image.informatik.htw-aalen.de/Thierauf/Seminar/Ausarbeitungen-14WS/agil.pdf (visited 10.10.2017)

[38] VersionOne. Url: https://www.versionone.com/pdf/2013-state-of-agile-survey.pdf (visited 10.11.17)

[39] Müller Klaus-Rainer. „IT-Sicherheit mit System – (Integratives IT-Sicherheits-, Kontinuitäts- und Risikomanagement - Sicherheitspyramide - Standards und Practices - SOA und Software-entwicklung)“. 2011. In: Springer

[40] Matthias Rohr. „Sicherheit im Software-Entwicklungsprozess“. 28.Juli 2015. Url: https://www.informatik-aktuell.de/betrieb/sicherheit/sicherheit-im-software-entwicklungsprozess.html. (visited 12.11.17)

[41] Carola Lilienthal. “Langlebige Software-Architekturen: Technische Schulden analysieren, begrenzen und abbauen“. 2017. In: dpunkt.verlag, 2.auflage

[42] Prof. Dr. Peter Knauber FH Mannheim. „Anforderungsanalyse, Requirements Engineering, Vorlesung Software Engineering“. Url: http://services.informatik.hs-mannheim.de/ knauber/BCSc-SE/06a-f.pdf (visited 12.11.17)

[43] Christian Löper, Universität Paderborn . „Anforderungsanalyse und Anforderungsdefinition für sicherheitskritische Systeme (inkl. formale Methoden)“. 2004. Url: http://www2.cs.uni-paderborn.de/cs/ag-schaefer-static/Lehre/Lehrveranstaltungen/Seminare/AEIzS/Abgaben/Ausarbeitung/CLoeper.pdf (visited: 12.11.17)

Abbildungsverzeichnis

Abbildung 2.1. [10] K. Schmid, “On the limits of the technical debt metaphor some guidance on going beyond”. 2013. In: Proceedings of the 4th International Workshop on Managing Tech-nical Debt (MTD), pp. 63–66. ACM

Abbildung 2.2. [4] Alexander Kriegisch. „Agiles Projektmanagement“. Url: scrum-master.de (visited 14.08.17)

Abbildung 2.3. [5] Donald Firesmith, Software Engineering Institute, U.S.A. “Engineering Safety Requirements, Safety Constraints, and Safety-Critical Requirements”. In: Journal of Object Technology, Vol. 3, No. 3, March-April 2004

Abbildung 2.4. [40] Matthias Rohr. „Sicherheit im Software-Entwicklungsprozess“. 28.Juli 2015. Url: https://www.informatik-aktuell.de/betrieb/sicherheit/sicherheit-im-software-entwicklungsprozess.html. (visited 12.11.17)

Abbildung 7.1. [44] Prof. Dr. Klaus-Peter Fähnrich, Institut für Informatik Betriebliche Informations-systeme, Universität Leipzig. „Vorlesung Softwaretechnik , Vorgehensmodelle, V-Modell XT“, WS 2005/2006

Dr. Wolfgang Goerigk , „Objektorientierte Programmierung in Java“ In: Vorlesung an der Fachhochschule Kiel Sommersemester 2009

Alle URLs wurden zuletzt am 10. 11. 2017 geprüft.

Erklärung

Ich versichere, diese Arbeit selbstständig verfasst zu haben. Ich habe keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommene Aussagen als solche gekennzeichnet. Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines anderen Prüfungsverfahrens. Ich habe diese Arbeit bisher weder teilweise noch vollständig veröffentlicht. Das elektronische Exemplar stimmt mit allen eingereichten Exemplaren überein.

Ort, Datum, Unterschrift