Karlsruher Institut für Technologie

Analyse von Entwurfsentscheidungen in natürlichsprachiger Softwarearchitekturdokumentation

Bachelorarbeit von

Bjarne Sauer

an der Fakultät für Informatik
Institut für Programmstrukturen und Datenorganisation (IPD)

Erstgutachterin:	Prof. Dr. Anne Koziolek
Zweitgutachter:	Prof. Dr. Ralf Reussner
Betreuender Mitarbeiter:	M.Sc. Jan Keim
Zweite betreuende Mitarbeiterin:	M.Sc. Sophie Schulz

07. Juni 2021 – 07. Oktober 2021

Zusammenfassung

Entwurfsentscheidungen bilden das Fundament zur Entwicklung qualitativ hochwertiger Softwaresysteme. Ihre Extraktion aus und Klassifikation in natürlichsprachiger Softwaredokumentation ermöglichen die Informationsgewinnung für Implementierungs- und Wartungsprozesse und die Erstellung konsistenter Dokumentationsartefakte.

Das in dieser Arbeit entwickelte Klassifikationsschema für Entwurfsentscheidungen erweitert bestehende Ansätze, die zwar in der Lage sind, Entwurfsentscheidungen in ihrer Breite zu klassifizieren, jedoch keine klar umrissenen Klassen für die Klassifikation in der hierarchischen Tiefe festlegen. Die hier entwickelten Klassen versuchen, die verschiedenen Arten von Entwurfsentscheidungen vollständig abzubilden. Die dazwischenliegenden, feingranulareren Trennlinien dienen dazu, die Entwurfsentscheidungen deutlicher voneinander abzugrenzen.

Nachdem zunächst ein initiales Klassifikationsschema entworfen wird, wird in einem iterativen Prozess die Passform des Klassifikationsschemas durch die Anwendung auf die reale Softwarearchitekturdokumentation von 17 Fallstudien validiert und verbessert, bis hin zur Konvergenz auf ein ausgereiftes Schema. Neben einer Übersicht, welche Entwurfsentscheidungen getroffen und dokumentiert werden, liefert die manuelle Analyse der Fallstudien einen mit Labeln versehenen Textkorpus.

In einem zweiten Teil wird eine Anwendungsmöglichkeit des entwickelten Klassifikationsschemas eröffnet, indem in einer Proof-of-Concept-Implementierung untersucht wird, mit welchen Ansätzen des maschinellen Lernens und der natürlichen Sprachverarbeitung Entwurfsentscheidungen in natürlichsprachiger Softwarearchitekturdokumentation identifiziert und klassifiziert werden können. Durch die Evaluation mit statistischen Maßen wird gezeigt, welche Methoden zur Textvorverarbeitung, zur Überführung in Vektorrepräsentationen und welche Lernalgorithmen besonders für diese Klassifikation geeignet sind. Die automatisierte Rückgewinnung von Entwurfsentscheidungen aus der Dokumentation und gleichzeitige Zuordnung zu Kategorien mit gemeinsamen Eigenschaften erleichtert Aspekte im Softwareentwicklungsprozess wie Verständlichkeit, Konsistenz und Wartbarkeit.

Inhaltsverzeichnis

Zusammenfassung **i**

1. Einleitung **1**
1.1. Motivation 1
1.2. Zielsetzung 2
1.3. Aufbau der Arbeit 3

2. Grundlagen und verwandte Arbeiten **4**
2.1. Klassifikationsschemata für Entwurfsentscheidungen 4
2.2. Automatisierte Verarbeitung von Entwurfsentscheidungen 6

3. Entwurf eines Klassifikationsschemas **9**
3.1. Hierarchische Gliederung des Klassifikationsschemas 9
3.2. Kruchtens Ontologie als Grundlage der Klassifikation 10
3.3. Initiales Klassifikationsschema 11
3.3.1. Aufbau und Grobstruktur 12
3.3.2. Schema-Erweiterung anhand weiterer Klassifikationsebenen . . 14

4. Iterative Weiterentwicklung des Klassifikationsschemas **19**
4.1. Konzeption 19
4.2. Auswahl der Fallstudien 21
4.3. Identifikation von Entwurfsentscheidungen 21
4.4. Untersuchung der Fallstudien 24
4.4.1. Iteration 1: Ausdifferenzierung der Verhaltensentscheidungen . 24
4.4.2. Iteration 2: Ausdifferenzierung der Strukturentscheidungen . . 26
4.4.3. Iteration 3: Festlegung der Funktionalität durch Entwurfsentscheidungen 28
4.4.4. Iteration 4: Ausdifferenzierung der Technologie-Entscheidungen 29
4.4.5. Iteration 5: Häufiges Vorkommen von Entwurfsentscheidungen zu Klassen, Methoden und Technologien 31
4.4.6. Iteration 6: Das Klassifikationsschema in seiner Breite 31

5. Auswertung des finalen Klassifikationsschemas **34**
5.1. Übersicht über die Klassen 34
5.1.1. Unterklassen der Existenzentscheidung 34
5.1.2. Unterklassen der Eigenschaftsentscheidung 38
5.1.3. Unterklassen der Ausführungsentscheidung 39
5.2. Klassenhäufigkeit in den Fallstudien 43
5.3. Validität des Klassifikationsschemas und dessen Anwendung in der manuellen Analyse 47
5.3.1. Evaluation der Struktur des Klassifikationsschemas 47
5.3.2. Evaluation der Klassifikation hinsichtlich Vollständigkeit 48
5.3.3. Evaluation der Anwendbarkeit und Handhabung des Schemas . 49
5.3.4. Vermeidung von Unsicherheiten bei der Anwendung des Klassifikationsschemas 50

6. Maschinelle Klassifikation der Entwurfsentscheidungen **52**
6.1. Automatisierte Analyse mit maschinellem Lernen 53
6.2. Ansätze für die Klassifikation von Sätzen aus natürlicher Sprache . . . 54
6.2.1. Vorverarbeitung und vektorielle Textrepräsentation 54
6.2.2. Verwendete Klassifikationsalgorithmen 55
6.2.3. Textklassifikation mit BERT 57
6.3. Implementierung eines Klassifikators für Entwurfsentscheidungen . . . 58
6.3.1. Vorverarbeitung von Text und Labeln 60
6.3.2. Training eines Klassifikators und k-fache Kreuzvalidierung . . 61
6.4. Evaluation der automatisierten Analyse 61
6.4.1. Erzielte Ergebnisse in der Klassifikation 62
6.4.2. Validität der erzielten Ergebnisse 69
6.4.3. Zukünftige Arbeiten 72

7. Fazit und Ausblick **74**

Literatur **76**

A. Anhang **81**

Abbildungsverzeichnis

3.1. Ontologie für architektonische Entwurfsentscheidungen nach Kruchten (nachgebildet und übersetzt) . . . 11
3.2. Klassifikation von Existenzentscheidungen unter Einbezug eines ausschließenden Charakters . . . 13
3.3. Erweiterung des initialen Klassifikationsschemas hinsichtlich Strukturentscheidungen . . . 15
3.4. Erweiterung des initialen Klassifikationsschemas hinsichtlich Verhaltens- und Anordnungsentscheidungen . . . 16
3.5. Erweiterung des initialen Klassifikationsschemas hinsichtlich technologischen Entscheidungen . . . 18

4.1. Verfeinerung der Klassifikation von Verhaltensentscheidungen . . . 25
4.2. Verfeinerung der Klassifikation von Strukturentscheidungen . . . 27
4.3. Verfeinerung der Klassifikation von technologischen Entscheidungen . 30

5.1. Finales Klassifikationsschema (Ausschnitt Existenzentscheidungen) . . 41
5.2. Finales Klassifikationsschema (Ausschnitt Eigenschafts- und Ausführungsentscheidungen) . . . 42

6.1. Schritte zur automatisierten Klassifikation von Entwurfsentscheidungen 59

Tabellenverzeichnis

4.1. Verwendete Projekte mit Softwarearchitekturdokumentation 22

5.1. Absolute Häufigkeit der Klassen von Entwurfsentscheidungen in den Fallstudien (Teil 1) 45
5.2. Absolute Häufigkeit der Klassen von Entwurfsentscheidungen in den Fallstudien (Teil 2) 46

6.1. Vergleich der durchschnittlichen und maximalen Korrektklassifizierungsrate (KKR) und F1-Werte erzielt unter Verwendung unterschiedlicher Textvorverarbeitung auf Ebene 0 (beste drei Werte jeweils grau hinterlegt) 64
6.2. Vergleich der Korrektklassifizierungsrate (KKR) und der F1-Werte für die verwendeten Vektorisierer und Klassifikationsalgorithmen für Ebene 0 65
6.3. Favorisierte Modellkombination für verschiedene Ebenen anhand der maximal möglichen Leistungfähigkeit hinsichtlich des F1-Wertes (alle Sätze) 66
6.4. Favorisierte Modellkombination für verschiedene Ebenen anhand der maximal möglichen Leistungfähigkeit hinsichtlich des F1-Wertes (nur Sätze mit Entwurfsentscheidungen) 67
6.5. Vergleich von BERT Transfer Learning mit der jeweils besten Modellkombination für eine Klassifikation auf den Ebenen 0 bis 3 68
6.6. Multi-Label-Klassifikation auf verschiedenen Ebenen, gegeben beste Modellkombination hinsichtlich gewichtetem F1-Wert sowie zugehörige Korrektklassifizierungsrate (KKR) 69

A.1. Vergleich KKR und F1-Werte für Vektorisierer-Lernalgorithmus-Kombinationen auf Ebene 0 für keine Textvorverarbeitung 81
A.2. Vergleich KKR und F1-Werte für Vektorisierer-Lernalgorithmus-Kombinationen auf Ebene 0 für Reduktion auf Kleinschreibung 82
A.3. Vergleich KKR und F1-Werte für Vektorisierer-Lernalgorithmus-Kombinationen auf Ebene 0 für Stammformreduktion/Lemmatisierung 82
A.4. Vergleich KKR und F1-Werte für Vektorisierer-Lernalgorithmus-Kombinationen auf Ebene 0 für Textaufbereitung hinsichtlich Sonderzeichen, Ziffern und Hyperlinks 83

A.5. Vergleich KKR und F1-Werte für Vektorisierer-Lernalgorithmus-Kombinationen auf Ebene 0 für Entfernung von Stoppwörtern 83
A.6. Vergleich KKR und F1-Werte für Vektorisierer-Lernalgorithmus-Kombinationen auf Ebene 0 für Reduktion auf Kleinschreibung & Stammformreduktion/Lemmatisierung 84
A.7. Vergleich KKR und F1-Werte für Vektorisierer-Lernalgorithmus-Kombinationen auf Ebene 1 (jeweils maximal erzielter Wert für verschiedene Methoden zur Textvorverarbeitung) 84
A.8. Vergleich KKR und F1-Werte für Vektorisierer-Lernalgorithmus-Kombinationen auf Ebene 2 (jeweils maximal erzielter Wert für verschiedene Methoden zur Textvorverarbeitung) 85
A.9. Vergleich KKR und F1-Werte für Vektorisierer-Lernalgorithmus-Kombinationen auf Ebene 3 (jeweils maximal erzielter Wert für verschiedene Methoden zur Textvorverarbeitung) 85
A.10. Vergleich KKR und F1-Werte für Vektorisierer-Lernalgorithmus-Kombinationen auf Ebene 1 unter Verwendung von Multi-Label-Klassifikation (Reduktion auf Kleinschreibung) 86
A.11. Vergleich KKR und F1-Werte für Vektorisierer-Lernalgorithmus-Kombinationen auf Ebene 2 unter Verwendung von Multi-Label-Klassifikation (Reduktion auf Kleinschreibung) 86
A.12. Vergleich KKR und F1-Werte für Vektorisierer-Lernalgorithmus-Kombinationen auf Ebene 3 unter Verwendung von Multi-Label-Klassifikation (Reduktion auf Kleinschreibung) 87

1. Einleitung

Dieses Kapitel motiviert zunächst in Abschnitt 1.1 die manuelle und automatisierte Analyse von Entwurfsentscheidungen in natürlichsprachiger Softwarearchitekturdokumentation, bevor in Abschnitt 1.2 die Zielsetzung dieser Bachelorarbeit präsentiert wird. Abschnitt 1.3 gibt einen Überblick über den Aufbau der Arbeit.

1.1. Motivation

Softwareprojekte lassen sich entlang von Entwicklungsphasen strukturieren, in denen Softwareentwickler neben ausführbaren und lieferbaren Softwareartefakten auch eine Dokumentation dieser anfertigen. Der Erfolg eines Projekts ist dabei maßgeblich von der Qualität der Dokumentation abhängig [36]. Um auch in großen und sich ändernden Entwicklungsteams hochwertige Softwaresysteme, unabhängig von ihrer Größe und Komplexität, entwickeln zu können, ist es essentiell, die getroffenen Entwurfsentscheidungen ausführlich und nachvollziehbar zu dokumentieren [45]. Entwurfsentscheidungen betreffen dabei insbesondere die Architektur der Software, aber auch den Einsatz bestimmter Technologien und Werkzeuge. Softwarearchitektur ist die grundlegende Organisation eines Systems, welche sich sowohl in seinen Komponenten, deren Beziehungen untereinander und zur Umgebung als auch den Entwurfs- und Weiterentwicklungsprinzipien offenbart [20]. Der Softwarearchitektur liegen dabei architektonische Entwurfsentscheidungen zugrunde [28].

Während die Softwarearchitektur textuell oder graphisch weitgehend dokumentiert wird, wird die explizite Dokumentation der zugrundeliegenden Entwurfsentscheidungen oft vernachlässigt [23]. Dadurch können diese Informationen während des Entwicklungs- und Wartungsprozesses verloren gehen, sodass (neu hinzukommende) Entwickler in späteren Entwicklungsphasen Schwierigkeiten haben, konkrete Einblicke in das Projekt zu erhalten und die getroffenen Entscheidungen nachzuvollziehen [45]. Dies liegt unter anderem daran, dass Entwurfsentscheidungen übersehen werden, nur noch veraltet dokumentiert sind oder die Entscheidungsbegründung nicht mehr zu rekonstruieren ist. Dieser Wissensverlust führt zu wiederkehrenden Problemen beim Entwurf komplexer Systeme, hohen Kosten für Veränderungen und einer Erosion der Softwarearchitektur [22].

Eine Möglichkeit, Zeit- und Budgeteinsparungen bei gleichzeitig hoher Dokumentationsqualität zu erreichen, ist der Einsatz automatisierter Verfahren zur Unterstützung

des Entwurfsprozesses sowie die automatisierte Rückgewinnung von Entwurfsentscheidungen aus Dokumentationen. Durch die automatisierte Rückgewinnung von Entwurfsentscheidungen lassen sich die für den Softwareentwickler in Implementation und Wartung relevanten Informationen aus der Dokumentation extrahieren. Darüber hinaus versucht diese Herangehensweise, die in [26] genannten Probleme aktuell auffindbarer Softwaredokumentation anzugehen: Die Dokumentation ist unzureichend hinsichtlich ihres Umfangs oder ihrer Aktualität und sie ist inkonsistent zu anderen Artefakten und Modellen.

1.2. Zielsetzung

Um die Erkennung und Analyse von Entwurfsentscheidungen zu unterstützen, entwickelt diese Bachelorarbeit ein Klassifikationsschema für Entwurfsentscheidungen, entlang dessen Softwarearchitekturdokumentationen analysiert werden. Um die Entwurfsentscheidungen in Dokumentationen zu erfassen und adäquat in die Entwicklung des Systems zu übersetzen, müssen Softwareentwickler den Anwendungsbereich der Entscheidungen und die intendierten Auswirkungen verstehen. Teil dieser Arbeit ist es daher auch, zu untersuchen, wie natürlichsprachige Softwaredokumentation strukturiert ist und wie Entwurfsentscheidungen darin beschrieben werden. Hierfür ist es sinnvoll, die Entwurfsentscheidungen durch eine Klassifikation voneinander abzugrenzen, sodass Informationen aus der Dokumentation als explizites Wissen aufbereitet werden. Die Klassifikation erlaubt es, Entwurfsentscheidungen eine Repräsentation erster Ordnung zu geben und dadurch einordnen zu können, wie sich Entwurfsentscheidungen auf die Gestaltung von Softwaresystemen auswirken. Um das Klassifikationsschema auf seine Passform hinsichtlich realer Dokumentationsartefakte zu prüfen, wird dieses iterativ auf Fallstudien angewandt. Die auffindbare Dokumentation der Softwarearchitektur in diesen Fallstudien wird dazu genutzt, verschiedene Entwurfsentscheidungen zu identifizieren und zu klassifizieren. Durch die Fallstudien erhält man zudem Einblick in reale Dokumentationen, um die Art und Weise, mit der Softwareentwickler ihre Systemarchitektur in natürlicher Sprache beschreiben, zu ergründen.

Das Klassifikationsschema und die Erkenntnisse aus den Fallstudien sollen darüber hinaus als Grundlage dienen, um Entwurfsentscheidungen automatisiert aus Dokumentationen zu extrahieren. Hierfür wird in einem zweiten Teil dieser Arbeit eine automatisierte Analyse für ausgewählte Klassen des Schemas vorgenommen. Die Implementation der Analyse dient als Proof of Concept, um zu bewerten, ob der gewählte Lösungsansatz geeignet ist, natürlichsprachige Softwarearchitekturdokumentation anhand des Schemas automatisiert zu analysieren.

Außerdem dient diese Bachelorarbeit dafür, auf ihr aufbauend Werkzeuge zu entwickeln, die eine Rückverfolgbarkeit und Konsistenzprüfung mit anderen Modellen ermöglichen. Dazu bedarf es einer Klassifikation der Entwurfsentscheidungen, da da-

von auszugehen ist, dass die Konsistenzprüfung für voneinander abgrenzbare Klassen von Entwurfsentscheidungen in unterschiedlicher Weise durchgeführt werden muss. Diese Annahme ist durch weitere Forschung noch zu überprüfen. Werkzeuge, die zum einen Entwurfsentscheidungen automatisch analysieren können und zum anderen Konsistenzprüfungen mit anderen Modellen ermöglichen, können ein Hilfsmittel für Softwareentwickler darstellen, um hochqualitative, gut dokumentierte Software zu entwickeln. Dies ist insbesondere dann der Fall, wenn der Nutzen von ausführlicher Dokumentation höher eingeschätzt wird als ihre Kosten.

1.3. Aufbau der Arbeit

Die vorliegende Bachelorarbeit gliedert sich wie folgt: Kapitel 2 führt in Grundlagen und verwandte Arbeiten zu Klassifikationsschemata für Entwurfsentscheidungen und die automatisierte Verarbeitung von Entwurfsentscheidungen ein. In Kapitel 3 wird ein initiales Klassifikationsschema anhand bestehender Arbeiten und eigener Erweiterungen entworfen, welches in Kapitel 4 iterativ auf Fallstudien angewandt wird und dabei kontinuierlich weiterentwickelt wird. Das daraus resultierende, finale Klassifikationsschema wird in Kapitel 5 in einer Übersicht dargestellt und die Klassenhäufigkeiten innerhalb der Fallstudien werden ausgewertet, bevor die Validität des Klassifikationsschemas diskutiert wird. Für ausgewählte Klassen des Schemas wird in Kapitel 6 eine maschinelle Klassifikation vorgenommen, indem verschiedene Klassifikationsalgorithmen aus dem Bereich des maschinellen Lernens in einer Implementierung verwendet werden und deren erzielte Ergebnisse unter Verwendung statistischer Maße untereinander verglichen werden. Kapitel 7 fasst die Erkenntnisse dieser Arbeit zusammen und gibt einen Ausblick auf mögliche zukünftige Arbeiten.

2. Grundlagen und verwandte Arbeiten

Bereits in anderen Arbeiten wurden Entwurfsentscheidungen in das Zentrum der Analyse von Dokumentationen gerückt. Hierfür wurden verschiedene Klassifikationsschemata entwickelt, die Entwurfsentscheidungen als eigene Entitäten modellieren. Für die Rückgewinnung von Informationen aus Dokumentationen wurden sowohl regelbasierte als auch auf maschinellem Lernen und Natural Language Processing beruhende Lösungen vorgeschlagen. Dieses Kapitel untersucht die bisher entwickelten Klassifikationsschemata für Entwurfsentscheidungen (Abschnitt 2.1) sowie bestehende Werkzeuge, die Entwurfsentscheidungen in Dokumentationen identifizieren (Abschnitt 2.2). Zudem beleuchtet es, inwiefern sich der Inhalt dieser Bachelorarbeit von bereits bestehenden Arbeiten abgrenzt.

2.1. Klassifikationsschemata für Entwurfsentscheidungen

Wie bereits in Kapitel 1 angesprochen, betreffen Entwurfsentscheidungen die Architektur eines Softwaresystems. Softwarearchitekten und Softwareentwickler legen hierin übergeordnete Strukturen fest, die das Softwaresystem erfüllen soll. Van der Ven et al. [50] definieren eine Entwurfsentscheidung als eine Beschreibung einer Entscheidung und der betrachteten Alternativen, um eine oder mehrere Anforderungen an ein Softwaresystem ganz oder teilweise umzusetzen. Hierbei setzen sich Entscheidungen und Alternativen aus einer Menge von architektonischen Ergänzungen, Verminderungen oder Anpassungen zusammen [50].

In [27] entwickelt Kruchten eine Ontologie für Entwurfsentscheidungen, die diese als selbstständige Entitäten modelliert. In der Ontologie werden Entwurfsentscheidungen ihrer Art nach als Existenzen, Eigenschaften oder die Ausführung betreffende Entscheidungen klassifiziert. Dabei wirken sich *Existenzentscheidungen* (*existence decision*) auf Softwareelemente aus, die im Design oder der Implementierung des Systems vorhanden sein werden. Sie können feiner unterschieden werden in Entscheidungen über die Struktur, das Verhalten oder einen Ausschluss. *Strukturentscheidungen* (*structural decision*) fordern die Erstellung von Subsystemen, Komponenten, Schichten oder Partitionen. *Verhaltensentscheidungen* (*behavioral decision*) betreffen die Interaktion der Softwareelemente untereinander, um eine bestimmte Funktionalität zu realisieren oder nicht-funktionale Anforderungen zu befriedigen. Ein *Ausschluss* (*ban*) kann gegensätzlich zu einer Existenzentscheidung getroffen werden, indem er die Existenz

eines Elements untersagt. Darüber hinaus können *Eigenschaftsentscheidungen* (*property decision*) getroffen werden, um Merkmale oder Qualitäten eines Systems durch Richtlinien oder Einschränkungen genauer zu spezifizieren. Betreffen Entscheidungen eher das Entwicklungsumfeld, beispielsweise die Organisationsstrukturen, Abläufe oder eingesetzten Technologien, sind sie der Klasse *Ausführungsentscheidung* (*executive decision*) zuzuordnen. Zudem können allen Entwurfsentscheidungen Begründungen zugewiesen werden und diese zu verwandten oder konträren Entwurfsentscheidungen in Beziehung gesetzt werden [27].

In einer Expertenbefragung stellten Miesbauer und Weinreich [33] fest, dass Softwareentwickler in der Mehrheit mit Existenzentscheidungen konfrontiert sind, während die Ausführung nur in etwa einem Viertel der Fälle im Zentrum steht und Eigenschaftsentscheidungen einen Anteil von unter 10% ausmachen. Die Befragten bekräftigten die Notwendigkeit der Dokumentation von Softwareentscheidungen, um Wissensverlust vorzubeugen. Außerdem ordneten die Befragten die Entwurfsentscheidungen unterschiedlichen Ebenen zu, die sich auf die vier Ebenen Implementierung, Architektur, Projekt und Organisation zusammenfassen ließen.

Jansen und Bosch [22] betrachten Softwarearchitektur als die Zusammensetzung zahlreicher architektonischer Entwurfsentscheidungen über die Zeit. Ihr Modell *Archium* modelliert die Beziehung zwischen Entwurfsentscheidungen und der Softwarearchitektur während des gesamten Entwicklungsprozesses der Software. Dabei ist die Architektur als eine Menge verwobener Entwurfsentscheidungen modelliert. Dadurch sollen Probleme bestehender Softwarearchitektur angegangen werden, wonach Entwurfsentscheidungen sich gegenseitig bedingen, Einschränkungen aus dem Entwurf im System verletzt werden und veraltete Entwurfsentscheidungen nicht aus der Dokumentation entfernt werden.

Das von Falessi et al. [12] entwickelte Rahmenwerk, welches Entwurfsentscheidungen mit ihrem Ziel und den betrachteten Alternativen (*Decision, Goal and Alternatives*) verknüpft, versucht die Wartbarkeit von Systemen durch die Nachvollziehbarkeit getroffener Entwurfsentscheidungen zu steigern. Zimmermann et al. [59] strukturieren darauf aufbauend die Entscheidungsfindung anhand der drei Stufen Entscheidungsidentifikation, Treffen der Entscheidung und Entscheidungsumsetzung. Hierbei sind ähnliche Entwurfsentscheidungen einem gemeinsamen Thema zugeordnet, welches sie betreffen. Ein Thema steht in einer Beziehung zu einem Abstraktionsniveau, auf dem eine Entscheidung getroffen wird. Es werden drei Abstraktionsniveaus unterschieden, nämlich solche Entscheidungen, die die Konzeption (*ConceptualLevel*), die Technologie (*TechnologyLevel*) oder Assets (*AssetLevel*) betreffen. Durch die Modellierung einer Entwurfsentscheidung als eigenständiges Objekt kann diese mit möglichen Alternativen verknüpft werden.

Ein Vergleich von neun existierenden Modellen zu architektonischen Entwurfsentscheidungen findet sich in [46]. Dieser stellt insbesondere 1) Entscheidung, 2) Einschrän-

kungen, 3) Lösung(sansatz) und 4) Begründung als zentrale, gemeinsame Elemente der Modelle fest. Dabei unterscheiden sich die Modelle in der benutzten Terminologie, die eingesetzt wird, um die gleichen Konzepte zu beschreiben.

Die bestehenden Modelle wurden überwiegend dazu entwickelt, Entwurfsentscheidungen in ihrer Breite zu klassifizieren. Zudem wurden Attribute herausgearbeitet, die die Entwurfsentscheidungen spezifizieren können, etwa wann, in welchem Kontext und von wem eine Entscheidung getroffen wurde. Außerdem zielen die Modelle darauf ab, die Begründung und Alternativen von Entwurfsentscheidungen zu erforschen. Sie wurden jedoch primär nicht dazu entwickelt, das architektonische Modell und die reale Architektur eines Softwaresystems mit den zugrundeliegenden Entwurfsentscheidungen zu vergleichen. Eine Ausnahme stellt hier das Modell Archium in [22] dar, wobei hier jedoch die Entwurfsentscheidungen nicht festgelegten Klassen entstammen müssen, sondern immer als Lösungsansatz eines zuvor identifizierten Problems modelliert werden.

Diese Arbeit untersucht darauf aufbauend, wie entsprechende Klassen für Entwurfsentscheidungen gewählt werden müssen, um diese auch in der hierarchischen Tiefe spezifisch voneinander abzugrenzen. Mit einem solchen Klassifikationsschema kann in zukünftigen Arbeiten eine Konsistenzprüfung zwischen der Dokumentation der Softwarearchitektur und ihrer Umsetzung in Modellen und Implementation angegangen werden. Die in [27] definierten Klassen Existenz-, Eigenschafts- und Ausführungsentscheidungen sind hierbei der Ausgangspunkt (vgl. Abschnitt 3.2), um eine in der hierarchischen Tiefe genauere Klassifikation zu entwickeln. Die Verfeinerung des Klassifikationsschemas behandelt zudem bisher nicht als eigene Klasse betrachtete Konzepte wie Architekturstile und Architekturmuster als mögliche Ausprägung von Entwurfsentscheidungen.

2.2. Automatisierte Verarbeitung von Entwurfsentscheidungen

Auch für die von Software unterstützte oder automatisierte Identifikation von Entwurfsentscheidungen lassen sich Forschungsarbeiten identifizieren. Diese können sich in Analysemodellen, algorithmischer Mustererkennung oder maschinellem Lernen ausprägen.

Basierend auf dem Modell *Archium* [22] wurde von Jansen et al. [23] ein Vorschlag (*Architectural Design Decision Recovery Approach, ADDRA*) zur manuellen Rückgewinnung von Entwurfsentscheidungen aus Dokumentationen anhand von Veränderungen zwischen zwei Releases gemacht. Der Vorschlag definiert fünf Phasen in einem iterativen Prozess, in denen ein Softwarearchitekt zwei Realeses auswählt und miteinander vergleicht, anschließend die Veränderungen an der Softwarearchitektur analysiert und

damit die zugrundeliegenden Entwurfsentscheidungen zurückgewinnt. Die Autoren führen insbesondere an, dass systematische Ansätze die Wahrscheinlichkeit reduzieren, wichtige Entwurfsentscheidungen zu übersehen und Wissen explizit machen können. Allerdings erfordern manuelle Ansätze die Mitarbeit von Softwarearchitekten und es bestehen generell Schwierigkeiten, nicht dokumentierte Entscheidungen oder Alternativen zurückzugewinnen.

Automatisierte Ansätze und die Entwicklung zugehöriger Softwarewerkzeuge erlauben es, Entwurfsentscheidungen mit geringerem Zeitaufwand und geringerem Projektwissen zurückzugewinnen. Shahbazian et al. [45] entwickelten ihre Technik *RecovAr*, um Entwurfsentscheidungen aus bestehenden Artefakten wie Issue-Tracking-System und Versionskontrolle zu regenerieren. Dabei unterteilt sich der Identifikationsprozess in drei Phasen: Zunächst wird die Veränderung der Softwarearchitektur anhand der Versionskontrolle rekonstruiert (*Change Analysis*). Anschließend werden identifizierte Probleme aus dem Issue-Tracker mit den Architekturelementen verknüpft, auf die sie sich laut Versionskontrolle ausgewirkt haben (*Mapping*). Abschließend werden die Ergebnisse der ersten beiden Phasen zu einem Graphen kombiniert, der eine Änderung mit dem damit adressierten Problem verknüpft (*Decision Extraction*).

Ebenfalls aus einem Issue-Tracking-System zweier großer Open-Source-Projekte haben Bhat et al. [6] Entwurfsentscheidungen mithilfe eines Machine-Learning-Ansatzes erkannt und anschließend klassifiziert. Dafür wurden ein 1500 Issues umfassender Datensatz manuell mit Labeln versehen und daraus Modelle für maschinelles Lernen generiert. Sie verwendeten hierfür das Klassifikationsschema von Kruchten [27] mit Fokus auf die Klasse der Existenzentscheidungen und ihre Unterkategorien Struktur, Verhalten und Ausschluss. Ihre Ergebnisse wurden für die Entwicklung des *Amelie-Decision Explorers (ADeX)* [5] genutzt. *ADeX* ist in der Lage, aus natürlichsprachigen Dokumentationstexten Entwurfsentscheidungen zu gewinnen und diese als kollaborative Webanwendung graphisch aufzubereiten. Durch Vorschläge für Lösungsansätze zu getroffenen Entwurfsentscheidungen, in Betracht zu ziehende Alternativen oder auch Verknüpfungen zu ähnlichen Entwurfsentscheidungen in der Vergangenheit unterstützt *ADeX* die Entscheidungsfindung über Softwarearchitektur.

Für die Erkennung von Entwurfsentscheidungen in Dokumentationen nutzen Li et al. [29] Mailinglisten, die sie mit maschinellem Lernen automatisiert verarbeiten. Hierfür wurde ein mit Labeln versehener, 1300 Sätze umfassender Datensatz aus der Hibernate Developer Mailing List generiert, welcher die Sätze nach *Entwurfsentscheidung* und *keine Entwurfsentscheidung* klassifiziert. Für verschiedene Methoden aus dem Bereich des Natural Language Processing (z.B. Stop Word Filter, Stemming) sowie für unterschiedliche Klassifikatoren wurden die Ergebnisse validiert. Hierbei wurde unter Verwendung einer Support Vector Machine ein F1-Wert von 0,759 erzielt.

Im Gegensatz zu den hier vorgestellten Arbeiten, die mit Issue-Tracking-Systemen, Versionskontrolle oder Mailinglisten arbeiten, steht die Dokumentation der Softwa-

rearchitektur selbst im Mittelpunkt der in dieser Arbeit durchgeführten Analyse. Aus der natürlichsprachigen Softwarearchitekturdokumentation werden die zugrundeliegenden Entwurfsentscheidungen extrahiert. Natürliche Sprache ist hierbei die meist genutzte Weise, um Softwarearchitektur zu spezifizieren [11]. Die Informationsquellen für die Analyse in dieser Arbeit sind somit nicht sekundär, sondern eine Aufbereitung der Softwarearchitekturdokumentation in Form von Identifikation und Klassifikation von Entwurfsentscheidungen. Das in den folgenden Kapiteln beschriebene Klassifikationsschema übersteigt dabei auch die in [29] angelegte binäre Klassifikation in *Entwurfsentscheidung* und *keine Entwurfsentscheidung*. Zudem soll die Klassifikation in ihrer hierarchischen Tiefe Entwurfsentscheidungen feingranularer unterscheiden, als dies mit der in [5] angelegten Klassifikation in die drei Unterkategorien der Existenzentscheidungen möglich ist. Dadurch entstehen feinere Trennlinien, derer es für die Konsistenzprüfung zwischen Architekturdokumentation und Softwaremodellen und -artefakten bedarf.

Eine solche Konsistenzprüfung adressiert bestehende Herausforderungen der Softwareentwicklung. Das ergab eine Befragung von Wohlrab et al. [56]. Inkonsistenzen treten demnach sowohl zwischen verschiedenen Architekturbeschreibungen als auch zwischen Architekturbeschreibung und Code auf. Sie stellten weiterhin vier verschiedene Typen von Inkonsistenzen fest: 1) Schnittstellenbeschreibung zu Schnittstellenimplementierung, 2) gebrochene Regeln und Einschränkungen, 3) nicht beachtete Muster und Richtlinien sowie 4) Inkonsistenzen in Wortwahl und Sprache, wobei die negativen Auswirkungen des letzten Punktes als eher gering erachtet werden.

3. Entwurf eines Klassifikationsschemas

Um Entwurfsentscheidungen in natürlichsprachiger Softwarearchitekturdokumentation systematisch analysieren zu können, werden die enthaltenen Entwurfsentscheidungen zunächst identifiziert und anschließend anhand eines Schemas klassifiziert. Die Entwicklung dieses Klassifikationsschemas lässt sich zweiteilen, indem zunächst ein initiales Klassifikationsschema entworfen wird und dieses anschließend in einem iterativen Prozess anhand von Fallstudien weiterentwickelt wird. Für den Entwurf des Klassifikationsschemas werden die bereits existierenden Betrachtungsweisen von Entwurfsentscheidungen, welche in Kapitel 2 beleuchtet wurden, einbezogen und erweitert, sodass für unser Anliegen sinnvolle Aspekte aufgegriffen werden sowie verschiedene Arbeiten kombiniert und erweitert werden. Das hieraus resultierende initiale Klassifikationsschema orientiert sich dabei an der Struktur von Kruchtens Ontologie für architektonische Entwurfsentscheidungen [27] und erweitert diese innerhalb einer hierarchischen Struktur um Unterklassen, sodass die Klassifikation die Entwurfsentscheidungen feingranularer voneinander abzugrenzen vermag. Die beim Entwurf des initialen Klassifikationsschemas eingeflossenen Überlegungen und Entscheidungen werden im Folgenden genauer dargelegt.

3.1. Hierarchische Gliederung des Klassifikationsschemas

Um Entwurfsentscheidungen zu identifizieren und gegeneinander abzugrenzen, bedarf es einer Repräsentation dieser als eigenständige Entitäten [22]. Allerdings kann diese Klassifikation auf unterschiedlichen Abstraktionsniveaus erfolgen. Den höchsten Abstraktionsgrad stellt eine binäre Klassifikation in *Entwurfsentscheidung* und *keine Entwurfsentscheidung* dar, wie sie in [29] angewandt wurde. Sie ist stets Ausgangspunkt der Analyse von natürlichsprachiger Softwarearchitekturdokumentation, die nicht ausschließlich Entwurfsentscheidungen enthält und dadurch auf den wesentlichen Kern der zu betrachtenden Dokumentation komprimiert werden kann.

Um zu untersuchen, wie Entwurfsentscheidungen in der auffindbaren Softwarearchitekturdokumentation beschrieben werden und um diese im Hinblick auf eine Konsistenzprüfung zwischen unterschiedlichen Entwurfsartefakten gegeneinander abzugrenzen, bedarf es eines Klassifikationsschemas, welches die Bandbreite möglicher Entwurfsentscheidungen angemessen abbildet. Mögliche Klassifikationsrichtungen sind in diesem

Fall eine hierarchische Gliederung des Klassifikationsschemas sowie die Klassifikation der Entwurfsentscheidungen entlang mehrerer Dimensionen.

Im Zentrum der hier durchgeführten Analyse steht eine Zuordnung von Entwurfsentscheidungen zu Klassen hinsichtlich ihrer Art. Entwurfsentscheidungen müssen durch das entworfene Klassifikationsschema insofern voneinander abgrenzbar sein, dass unterscheidbare Klassen von Entwurfsentscheidungen mit unterschiedlichen Auswirkungen auf die Softwarearchitektur einhergehen. Hierfür bietet sich eine hierarchische Gliederung des Klassifikationsschemas an, um die Auswirkungen auf die Softwarearchitektur entlang unterschiedlicher Abstraktionsniveaus zu untersuchen. Durch eine baumartige Struktur können Entwurfsentscheidungen entlang verschiedener Ebenen strukturiert werden. In einem iterativen Validierungsprozess kann daraufhin beurteilt werden, ob es sinnvoll ist, einzelne Klassen weiter aufzuspalten und somit das Schema zu verfeinern. Umgekehrt kann die Softwaredokumentation nur noch hinsichtlich der Oberklasse analysiert werden, falls sich entweder die Unterklassen nicht strikt voneinander abgrenzen lassen oder ihre Unterscheidung keinen Mehrwert liefert. Dies ist insbesondere dahingehend interessant, dass weitere Arbeiten noch zeigen müssen, wie eine Konsistenzprüfung für unterschiedliche Klassen von Entwurfsentscheidungen aussehen könnte. In der automatisierten Analyse der Entwurfsentscheidungen in dieser Arbeit können somit bereits Ergebnisse für ausgewählte Klassen erzielt werden, bevor zu einem späteren Zeitpunkt eine Verfeinerung der Mustererkennung vorgenommen wird.

Darauf aufsetzend kann die Detailliertheit der Klassifikation durch die Einführung zusätzlicher Dimensionen weiter erhöht werden. So wurden in der Expertenbefragung von Miesbauer und Weinreich [33] die Entwurfsentscheidungen zum einen den Klassen entsprechend der Ontologie von Kruchten [27] zugeordnet, zum anderen aber auch den Ebenen Implementierung, Architektur, Projekt und Organisation. Auch eine Klassifikation in unterschiedliche Formen der Repräsentation von Entwurfsentscheidungen, wie es Glinz für nicht-funktionale Anforderungen vorgeschlagen hat [14], ist denkbar. Mehrdimensionale Klassifikation geht jedoch einher mit erhöhtem Aufwand sowohl bei der manuellen als auch automatischen Analyse der Softwaredokumentationen, sodass abzuwägen bleibt, welchen Mehrwert eine zusätzliche Dimension einbringt.

3.2. Kruchtens Ontologie als Grundlage der Klassifikation

Mit Kruchtens Ontologie für architektonische Entwurfsentscheidungen [27] und Jansen und Boschs Sammlung architektonischer Entwurfsentscheidungen [22] bestehen zwei zentrale Arbeiten, die sich der Klassifikation widmen. Jedoch kann gerade in der Herangehensweise, mit der Entwurfsentscheidungen betrachtet werden, ein deutlicher Unterschied ausgemacht werden. Während Jansen und Bosch Entwurfsentscheidungen stets als die Lösung eines identifizierten Problems klassifizieren und sich somit der

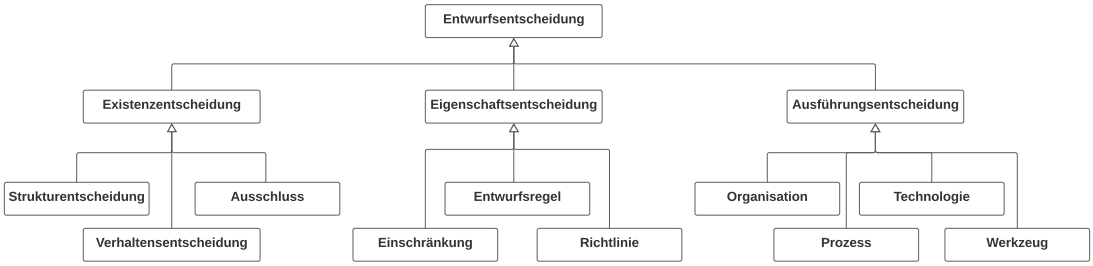

Abbildung 3.1.: Ontologie für architektonische Entwurfsentscheidungen nach Kruchten (nachgebildet und übersetzt)

Klassifikation ausgehend von der Problemstellung nähern, nimmt die Ontologie nach Kruchten die getroffene Entscheidung als solche in den Blick und versucht zu bewerten, welche Auswirkungen damit auf die Softwarearchitektur beabsichtigt wurden. Dies entspricht stärker der Vorgehensweise in dieser Arbeit, denn die Entwurfsentscheidungen werden aus der primären Softwarearchitekturdokumentation extrahiert, ohne eine Verknüpfung zum zugrundeliegenden Problem, welches sich etwa aus den Daten eines Issue-Tracking-Systems rekonstruieren lässt, herzustellen. Über das zugrundeliegende Problem lassen sich nur dann Rückschlüsse ziehen, falls die Softwarearchitekturdokumentation neben der Entscheidung als solche auch deren betrachtete Alternativen und eine argumentative Begründung für die Entscheidung bereithält.

Gemeinsam mit der Feststellung, dass Kruchtens Ontologie hierarchisch gegliedert ist, wurde daher die Entscheidung getroffen, diese als Grundlage der Klassifikation heranzuziehen (vgl. Abbildung 3.1). Das hieraus entwickelte initiale Klassifikationsschema weist dennoch einige Unterschiede auf, um die Passform für die sich anschließende Analyse der Fallstudien zu optimieren.

3.3. Initiales Klassifikationsschema

Anhand von verwandten Arbeiten und eigenen Überlegungen wurde ein initiales Klassifikationsschema entwickelt. Dieses dient als Ausgangspunkt für die Klassifikation innerhalb der Fallstudien. Im Vergleich zur in Abschnitt 3.2 beschriebenen Ontologie nach Kruchten weist das initiale Klassifikationsschema eine verfeinerte hierarchische Gliederung der Klassen auf. Diese Grobstruktur wird im Folgenden genauer erläutert, während die Beschreibung der Unterklassen im Detail sowie Anhaltspunkte, wann eine gegebene Entwurfsentscheidung entsprechend zugeordnet werden sollte, in Kapitel 4 geliefert werden.

3.3.1. Aufbau und Grobstruktur

Zunächst lassen sich entsprechend der Ontologie nach Kruchten [27] drei Oberklassen von Entwurfsentscheidungen identifizieren: *Existenzentscheidungen* (*existence decision*), *Eigenschaftsentscheidungen* (*property decision*) und *Ausführungsentscheidungen* (*executive decision*) (siehe Kapitel 2).

Existenzentscheidungen legen fest, ob ein Element oder Artefakt im Entwurf und der Implementierung des Softwaresystems existieren soll [27]. Insbesondere Existenzentscheidungen sind von besonderem Interesse für die Klassifikation, da sie zum einen direkte Auswirkungen auf die Gestaltung anderer Modelle und die Implementierung haben und zum anderen in Untersuchungen den Großteil aller in Softwarearchitekturdokumentation auffindbaren Entwurfsentscheidungen ausgemacht haben [33]. Aufgrund ihres häufigen Auftretens ist eine Aufspaltung in Unterklassen angebracht. In der Ontologie für architektonische Entwurfsentscheidungen wird eine Verfeinerung in *Struktur-* (*structural decision*), *Verhaltens-* (*behavioral decision*) und *Ausschlussentscheidungen* (*ban*) vorgeschlagen [27]. Während die Unterscheidung zwischen Struktur- und Verhaltensentscheidungen in das initiale Klassifikationsschema übernommen wurde, entfällt die Klasse der Ausschlussentscheidungen. Die Abgrenzung zwischen Struktur- und Verhaltensentscheidungen ist dahingehend sinnvoll, dass Strukturentscheidungen zur Erstellung von Softwareelementen führen, wohingegen Verhaltensentscheidungen deren Verbindung und Interaktion untereinander beschreiben sollen. Die Auswirkungen auf die Softwarearchitektur unterscheiden sich deutlich und dadurch auch eine angestrebte Konsistenzprüfung zwischen Entwurfs- und Implementationsartefakten. So kann eine Strukturentscheidung die Erstellung einer Komponente zur Folge haben, welche sich mit deren Existenz innerhalb eines Komponentenmodells abgleichen lässt. Handelt es sich jedoch um eine Verhaltensentscheidung, ist eine mögliche konsistente Übersetzung in ein Komponentenmodell eine Beziehung und Abhängigkeit zwischen zwei verschiedenen Komponenten.

Zwar lässt sich aus einer Ausschlussentscheidung ableiten, dass das entsprechende Softwareelement im Architekturmodell und in der Implementierung nicht vorhanden sein wird. Allerdings unterscheiden sich die Auswirkungen auf die Softwarearchitektur stark voneinander, je nachdem, ob der Ausschluss etwa die Verwendung einer bestimmten Bibliothek untersagt oder zwei Komponenten strikt voneinander entkoppelt werden sollen. Hinsichtlich einer Klassifikation nach Art der Entscheidung überwiegen die Gemeinsamkeiten zwischen ähnlichen Struktur- oder Verhaltensentscheidungen, ob beispielsweise ein und dieselbe Bibliothek verwendet oder nicht verwendet wird, den Gegensatz zwischen Existenz und Abstinenz. Vielmehr ist es eine grundsätzliche Eigenschaft einer Entwurfsentscheidung, ob sie die Verwendung eines Elementes vorschlägt oder untersagt. Die Klassifikation, ob es sich um einen Ausschluss handelt, kann daher als zusätzliches Attribut oder gar Dimension einer Entwurfsentscheidung eingeführt

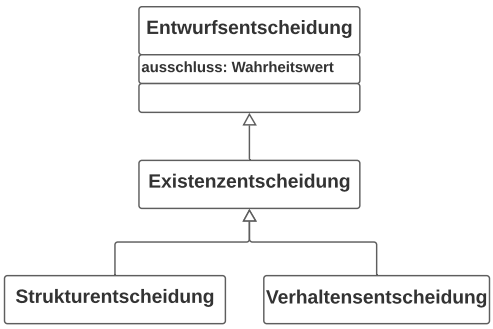

Abbildung 3.2.: Klassifikation von Existenzentscheidungen unter Einbezug eines ausschließenden Charakters

werden, taucht umgekehrt jedoch nicht als Unterklasse der Existenzentscheidungen auf (vgl. Abbildung 3.2).

Auch für die Analyse natürlichsprachiger Softwarearchitekturdokumentation lassen sich gleichartige Überlegungen treffen, inwiefern Existenzentscheidungen von Nichtexistenzentscheidungen zu unterscheiden sind. Die naheliegende Idee, diese Unterscheidung sprachlich auszudrücken, ist die Verneinung des Aussagesatzes. Der Erkenntnisgewinn einer Mustererkennung, die bejahende von verneinenden Sätzen trennt, ist jedoch hinsichtlich der Klassifikation nach Art einer Entwurfsentscheidung nicht ausreichend, solange nicht auch die Auswirkungen auf andere Modelle und die Implementierung Berücksichtigung finden.

Während Existenzentscheidungen einen Großteil dokumentierter Entwurfsentscheidungen ausmachen, wurden in einer vergangenen Expertenbefragung Entwurfsentscheidungen nur in 4% der Fälle zu den Eigenschaftsentscheidungen zugeordnet [33]. Da die Ontologie nach Kruchten die Grundlage für dieses initiale Klassifikationsschema bildet und der Anspruch besteht, alle identifizierten Entwurfsentscheidungen auch einer Klasse zuordnen zu können, weist das initiale Klassifikationsschema auch Eigenschaftsentscheidungen sowie deren Unterklassen *Einschränkung* (*constraint*), *Entwurfsregel* (*design rule*) und *Richtlinie* (*guideline*) [27] aus. Allein anhand der Fallstudien lässt sich beurteilen, ob diese Granularität bereits ausreichend ist oder auch für die Klasse der Eigenschaftsentscheidungen feinere Klassengrenzen eingeführt werden sollten.

Entwurfsentscheidungen, die sich der Oberklasse der Ausführungsentscheidungen zuordnen lassen, können genauer in Entscheidungen zur *Organisation* (*organization*), zum *Prozess* (*process*) sowie zu eingesetzten *Technologien* (*technology*) und eingesetzten *Werkzeugen* (*tool*) unterteilt werden [27]. Da insbesondere Entscheidungen zur Organisation und zum Prozess eher das Entwicklungsumfeld und den Entwicklungsprozess spezifizieren und damit nur indirekt auf die Gestaltung des Softwaresystems einwirken, bestehen an dieser Stelle im initialen Klassifikationsschema keine weiteren Verfeinerun-

gen. Auch für eine mögliche Konsistenzprüfung zwischen Architekturdokumentation und Entwurfs- und Softwareartefakten anhand des Klassifikationsschemas aus dieser Arbeit bleibt abzuwägen, ob entsprechende Entscheidungen im Umfeld der Entwicklung einen Einfluss auf die Softwarearchitektur bewirken können. Spezifikationen zu den eingesetzten Technologien und Werkzeugen sind hingegen aus dem Blickpunkt dieser Arbeit durchaus interessant, da der Einsatz unterschiedlicher Entwicklungsumgebungen Auswirkungen auf die Umsetzbarkeit architektonischer Entscheidungen mit sich bringt [21].

3.3.2. Schema-Erweiterung anhand weiterer Klassifikationsebenen

Mit den Vorüberlegungen aus Unterabschnitt 3.3.1 ergibt sich eine Bewertung, welche Klassen der Ontologie durch die Einführung neuer Klassifikationsebenen und damit neuer Unterklassen stärker ausdifferenziert werden. Dies betrifft im initialen Klassifikationsschema im Vergleich zur zugrunde gelegten Ontologie nach Kruchten die Struktur- und Verhaltensentscheidungen als Unterklasse der Existenzentscheidungen sowie die die Technologie betreffenden Ausführungsentscheidungen.

3.3.2.1. Strukturentscheidungen

Strukturentscheidungen als Unterklasse der Existenzentscheidungen können Entscheidungen über die Existenz ganz unterschiedlicher Artefakte ausdrücken. Betrachtet man die von Bhat et al. aufgestellten Regeln zur manuellen Klassifikation [6, S. 8], so werden stets zwei Gruppen von Entscheidungen getrennt voneinander genannt. Auf der einen Seite kann eine Strukturentscheidung zum Hinzufügen oder Aktualisieren eines Plug-ins, einer Bibliothek oder eines System eines Drittanbieters führen. Auf der anderen Seite haben Strukturentscheidungen Auswirkungen auf die Existenz von Modulen, Dateien oder Klassen inklusive der Deklaration ihrer Methoden.

Ergründet man den Hintergedanken dieser Gruppierung, so ist eine mögliche Erklärung, dass Entwurfsentscheidungen über *Plug-ins* (*plug-in*), *Bibliotheken* (*library*) und *Drittanbieter-Systeme* (*third-party system*) stets eine Auswirkung auf die Softwarearchitektur haben, die über die Systemgrenzen hinweg beobachtet werden kann. Dies geht mit entsprechenden Prinzipien nach Golden für die Architektur von Informationssystemen einher, welche unter anderem besagen, dass ein Informationssystem in einem abgrenzbaren Bereich existiert, aber sich stets in Interaktion mit anderen Systemen befindet [15]. Die zugehörigen Schnittstellen für eine solche Interaktion können der Einbau von Plug-ins, die Verwendung externer Bibliotheken oder der Anschluss bereits bestehender Softwaresysteme sein. Im Klassifikationsschema werden diese Entwurfsentscheidungen zu *intersystemischen Strukturentscheidungen* (*intersystem*) subsumiert.

Ihr Gegenstück sind die *intrasystemischen Strukturentscheidungen* (*intrasystem*) (vgl. Abbildung 3.3). Softwaresysteme können in eine Menge kleinerer Subsysteme herunter-

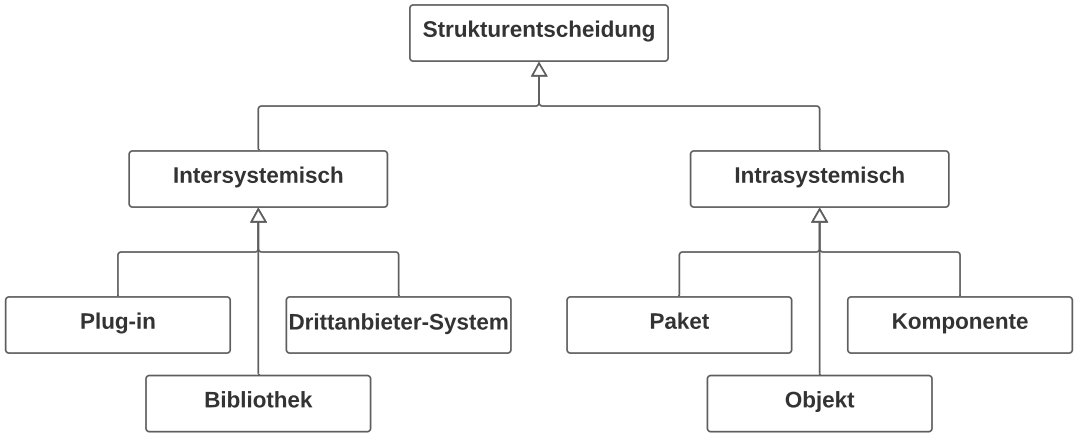

Abbildung 3.3.: Erweiterung des initialen Klassifikationsschemas hinsichtlich Strukturentscheidungen

gebrochen werden, die wiederum einzeln auf ihre architektonische Struktur analysiert werden können [15]. So strukturierte Softwaresysteme, die ihre Funktionalität hinter wohldefinierten Schnittstellen kapseln, ermöglichen die Entwicklung und Analyse der Softwarearchitektur entlang unterschiedlicher Abstraktionsniveaus [47]. Unter die intrasystemischen Strukturentscheidungen fallen insbesondere Entscheidungen, die wie genannt die Existenz von Modulen oder Klassen betreffen.

Das initiale Klassifikationsschema unterteilt die Klasse der intrasystemischen Strukturentscheidungen in Existenzentscheidungen zu erstens *Klassen* (*class*), zweitens *Komponenten* (*component*) und drittens *Paketen* (*package*). Aus Sicht der objektorientierten Softwareentwicklung werden Objekte anhand des in der Klasse festgelegten Bauplans instantiiert. Mehrere Klassen lassen sich zu modularen Teilen eines Systems zusammensetzen, sodass die Funktionsweise in einer Komponente gekapselt vorliegt und Komponenten austauschbar innerhalb ihrer Umgebung eingesetzt werden können [31, S. 222]. Dementsprechend definiert Szyperski Komponenten als zusammengesetzte Einheiten mit vertraglich festgelegten Schnittstellen und explizitem Kontextbezug, die voneinander unabhängig bereitgestellt und zusammengefügt werden können [48]. Auf einer dritten Strukturebene können Softwareelemente darüber hinaus in Paketen gruppiert werden, um der Gruppe einen Namensraum zuzuordnen [31]. Pakete enthalten eine Menge von zusammengehörigen Klassen und helfen beim Verpacken und Ausliefern fertiggestellter Softwareartefakte. Während die Bezeichnung als Pakete vor allem in Java Anwendung findet, werden in Python zusammengehörige Klassen in sogenannten Modulen gebündelt. Auf der Abstraktionsebene von Entwurfsentscheidungen kann diese Unterscheidung nicht mehr konsequent getroffen werden, sodass die Entscheidungen in der Klasse *Paket* zusammenfallen.

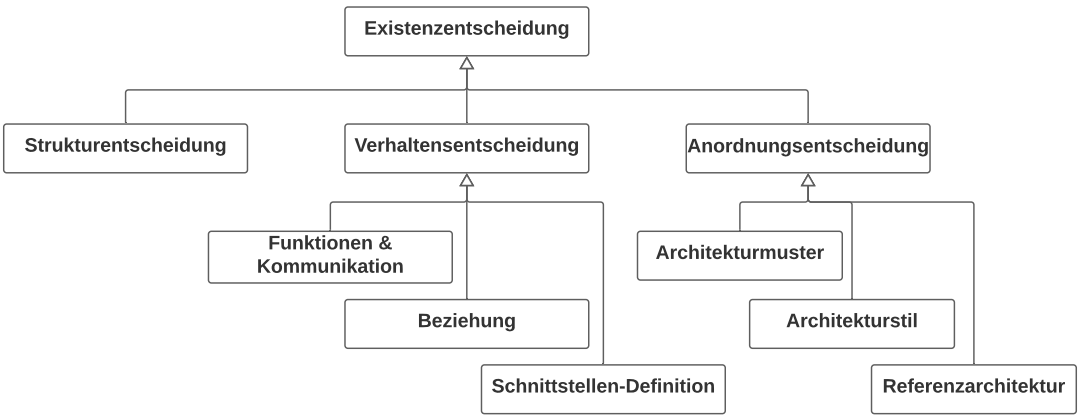

Abbildung 3.4.: Erweiterung des initialen Klassifikationsschemas hinsichtlich Verhaltens- und Anordnungsentscheidungen

3.3.2.2. Verhaltensentscheidungen

Neben den Strukturentscheidungen können auch Existenzentscheidungen getroffen werden, die das Verhalten eines Softwaresystems spezifizieren. Hierbei spielen die Funktionalität der Softwareelemente sowie deren Interaktion untereinander die zentrale Rolle. Verhaltensentscheidungen können das Hinzufügen von Funktionalität durch Methoden oder Funktionen betreffen, Prozessabläufe skizzieren oder Programmierschnittstellen neu einführen und verändern [6, S.8]. Gleiches gilt für negativ formulierte Verhaltensentscheidungen, die dazu führen, dass Funktionalität wieder ausgeschaltet wird und Methoden oder Schnittstellen entfernt werden.

Zur detaillierten Klassifikation unterteilt das initiale Klassifikationsschema Verhaltensentscheidungen in drei Unterklassen, namentlich Verhaltensentscheidungen zu erstens *Funktionen & Kommunikation* (*functions & communication*), zweitens *Beziehungen* (*relation*) und drittens *Schnittstellen-Definition* (*interface definition*) (vgl. Abbildung 3.4). Die Klasse *Funktionen & Kommunikation* umfasst dabei Entwurfsentscheidungen, mit denen Methoden hinzugefügt, verändert oder entfernt werden und ihre Parameter und Rückgabewerte festgelegt werden, aber auch wenn die Datenübermittlung und Prozessabläufe zwischen verschiedenen Softwarekomponenten durch Nachrichtenaustausch betroffen sind. Um das Verhalten zwischen verschiedenen Softwarekomponenten zu realisieren, werden Beziehungen und Abhängigkeiten eingesetzt. Eine solche Beziehung deutet an, dass ein Softwareelement oder eine Menge von Softwareelementen für seine vollständige Spezifikation oder Implementierung die Umsetzung anderer Softwareelemente benötigt [31, S. 42]. Hierbei kann es sich um Assoziationen, Aggregationen und Kompositionen handeln. Wie viele Softwareelemente an der Beziehung beteiligt sein sollen, wird durch Entscheidungen über die Multiplizitäten bestimmt. Um

Funktionalitäten und Abhängigkeiten gebündelt und verpflichtend festzulegen, können Softwareentwickler Schnittstellen definieren. Sie bilden eine Deklaration einer Menge von öffentlich einsehbaren Eigenschaften und Verpflichtungen, die implementiert einen kohärenten Service ergeben [31, S. 169]. Mit Verhaltensentscheidungen bezüglich einer *Schnittstellen-Definition* können Schnittstellen hinzugefügt, verändert und entfernt werden.

3.3.2.3. Anordnungsentscheidungen

Weitet man den Blick auf die Softwarearchitektur, kann häufig eine wiedererkennbare Anordnung der Softwareelemente beobachtet werden. Für die Architektur von Softwaresystemen, die eine bestimmte Funktionalität bereitstellen sollen, haben sich über die Zeit Anordnungen der Softwareelemente entwickelt, die die Implementierung und Wartung der Software vereinfachen. Es lassen sich *Architekturmuster* (*architectural pattern*), *Architekturstile* (*architectural style*) und *Referenzarchitekturen* (*reference architecture*) voneinander unterscheiden. Während ein Architekturmuster eine Lösung zu einem speziellen, wiederkehrenden Problem auf der Architekturebene darstellt, können Architekturstile als Lösungsprinzipien betrachtet werden, die unabhängig von der Applikation sind und durchgängig in der Softwarearchitektur angewandt werden müssen [43]. Referenzarchitekturen legen Komponenten und Subsysteme fest, die durch konkrete Instanzen genutzt werden können [43]. *Anordnungsentscheidungen* (*arrangement decision*) wirken sich dabei sowohl durch die Existenz von strukturellen Komponenten als auch durch die Prinzipien, die für deren Abhängigkeiten und Interaktionen untereinander gelten, auf das System aus. Daher sind sie auf einer Ebene mit den Struktur- und Verhaltensentscheidungen eingeordnet (vgl. Abbildung 3.4).

3.3.2.4. Technologische Entscheidungen

Während Ausführungsentscheidungen zu Organisation und Prozess eher die Projektarbeit der Entwicklerteams optimieren, haben Entscheidungen zu eingesetzten Technologien und Werkzeugen sehr konkrete Implikationen auf die Ausgestaltung des Softwaresystems. Während für eine initiale Klassifikation Entscheidungen über eingesetzte Werkzeuge eine homogene Gruppe an Entscheidungen erwarten lassen, ist der Begriff Technologie durchaus weit gefasst. Hierunter fällt die Entscheidung, sich auf eine Programmiersprache wie beispielsweise Java zu einigen, aber auch die Festlegung auf eine Entwicklungsplattform wie Jakarta EE (ehemals Java 2 Platform Enterprise Edition, J2EE) [27]. Daraus leiten sich für das initiale Klassifikationsschema die Unterklassen *Programmiersprache* (*programming language*) und *Plattform* (*platform*) ab. Die häufigsten Programmiersprachen sind unter anderem Java, Python oder C++. Dagegen kann eine Plattform sowohl die Software- als auch die System-Entwicklungsumgebung beschreiben und umfasst somit neben Softwarekomponenten auch die eingesetzte Hardware

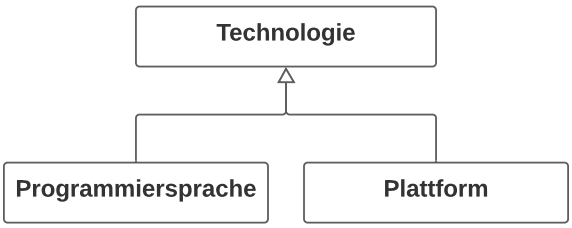

Abbildung 3.5.: Erweiterung des initialen Klassifikationsschemas hinsichtlich technologischen Entscheidungen

und das Betriebssystem [13]. Die sich anschließende Untersuchung der Fallstudien soll bestimmen, welche weiteren technologischen Entscheidungen in natürlichsprachiger Softwarearchitekturdokumentation getroffen werden.

4. Iterative Weiterentwicklung des Klassifikationsschemas

Um die Passform des Klassifikationsschemas auf natürlichsprachige Softwarearchitekturdokumentation zu überprüfen und zu verfeinern, findet eine iterative Anwendung verschiedener Versionen des Klassifikationsschemas auf ausgewählte Fallstudien statt. Diese manuelle Analyse anhand von Fallstudien gewährt zum einen Einblicke in Dokumentationsartefakte und lässt Aussagen zum Vorkommen und zur Häufigkeit bestimmter Entwurfsentscheidungen zu. Zum anderen verbessert sich die Qualität des Klassifikationsschemas, indem neben theoretischen Überlegungen auch ein Bezug zu realen Fallstudien hergestellt wird.

In diesem Kapitel gehe ich zunächst darauf ein, wie die die Analyse der Fallstudien konzipiert ist (Abschnitt 4.1), wie die Fallstudien ausgewählt wurden (Abschnitt 4.2) und wie sich Entwurfsentscheidungen darin identifizieren lassen (Abschnitt 4.3). Anschließend werden entlang der durchgeführten Iterationen die Erkenntnisse aus der Analyse der Fallstudien beleuchtet und in Veränderungen am Klassifikationsschema überführt (Abschnitt 4.4).

4.1. Konzeption

Die manuelle Analyse erfolgt in aufeinanderfolgenden Iterationen, wobei jeweils für alle in einer Iteration betrachteten Fallstudien die gleiche Version des Klassifikationsschemas zur Klassifikation der auffindbaren Entwurfsentscheidungen verwendet wird. Zum Abschluss einer Iteration werden die Erkenntnisse gebündelt und in notwendige Veränderungen am Klassifikationsschema übersetzt. Dieses modifizierte Klassifikationsschema dient wiederum als Grundlage der Klassifikation in der nächsten Iteration. In der ersten Iteration kommt das in Abschnitt 3.3 beschriebene initiale Klassifikationsschema zur Anwendung. Die Veränderungen am Klassifikationsschema können sich in hinzugefügten (Unter-)Klassen oder entfernten Klassen, die wiederholt nicht gesehen wurden oder nun anderweitig erfasst werden, manifestieren. Außerdem ergibt sich eine detaillierte Beschreibung, welche Entwurfsentscheidungen in einer Klasse erfasst werden sollen. Um eine konsistente Klassifikation im für die automatisierte Analyse verwendeten Korpus vorweisen zu können, wird das resultierende Klassifikationsschema der letzten Iteration noch einmal auf alle zuvor gesehenen Fallstudien angewandt.

Zunächst wurden die Fallstudien einzeln betrachtet. In der ersten Iteration drängten sich bereits durch die eine gesehene Architekturdokumentation Veränderungen am initialen Klassifikationsschema auf, die auch unter theoretischen Überlegungen eine Gültigkeit über die Fallstudie hinweg vermuten lassen. Diese Besonderheit ist insbesondere darauf zurückzuführen, dass das initiale Klassifikationsschema zwangsläufig stärkeren Veränderungen unterliegt als fortgeschrittenere Versionen des Schemas. Dies liegt daran, dass das initiale Klassifikationsschema ohne den Bezug zu realer Softwarearchitekturdokumentation entwickelt wurde.

Für nachfolgende Iterationen ergibt sich ein immer weiter ausgereiftes Klassifikationsschema, sodass Veränderungen am Klassifikationsschema seltener werden. Daher kommen in den folgenden Iterationen jeweils drei Fallstudien zum Einsatz. Dies hat sich während der Bearbeitung innerhalb der Iterationen zwei und drei als sinnvoller Richtwert ergeben. Die Anzahl wurde entsprechend einer Abwägung zwischen einerseits häufiger Bewertung und Reflexion der Passform des Schemas und andererseits hinreichender Allgemeingültigkeit der Erkenntnisse gewählt. Für eine größere Anzahl an Fallstudien pro Iteration hätte sich eine Klassifikation ergeben, die bereits während der manuellen Analyse als gegensätzlich zu den bisherigen Erkenntnissen in Form des angewandten Klassifikationsschemas und den bereits analysierten Fallstudien dieser Iteration bewertet werden müsste. Falls sich aus den bereits analysierten Fallstudien dieser Iteration eine Änderung am Klassifikationsschema ergibt, wird die weitere Fallstudie dennoch mit dem noch nicht überarbeiteten Schema vorgenommen, da Änderungen erst am Ende einer Iteration gebündelt in das Schema überführt werden. Für eine größere Anzahl an Fallstudien pro Iteration müssten folglich für die Erstellung eines konsistenten Korpus zahlreiche Korrekturen vorgenommen werden, deren Ursprung bereits hätte vermieden werden können. Für eine kleinere Anzahl an Fallstudien ist zu befürchten, dass die Veränderungen am Klassifikationsschema sich zu sehr an einem Spezialfall orientieren, da davon auszugehen ist, dass die gesehenen Entwurfsentscheidungen sehr stark von der Art und Größe des dokumentierten Softwaresystems abhängig sind. Dies ließ sich anhand der Fallstudien bestätigen (vgl. Abschnitt 5.2).

Um die Anzahl der Iterationen zu bestimmen, werden die Veränderungen am Klassifikationsschema in den vorangegangenen Iterationen bewertet. Während in den ersten Iterationen die Veränderungen am Klassifikationsschema noch recht stark ausfallen, wirken sie sich in späteren Iterationen nur noch auf Unterklassen tief in der Baumstruktur des Schemas aus, bis schließlich alle identifizierten Entwurfsentscheidungen vollständig und nachvollziehbar klassifiziert werden können. Das Klassifikationsschema konvergiert also über die Iterationen zu einem stabilen Klassifikationsschema. Die Analyse der Fallstudien wird beendet, sobald sich nachhaltig keine Veränderungen mehr am Klassifikationsschema ergeben. In der hier durchgeführten Analyse der Fallstudien konnten in den Iterationen fünf und sechs alle Entwurfsentscheidungen vollständig und nachvollziehbar klassifiziert werden, sodass die iterative Weiterentwicklung mit der

sechsten Iteration finalisiert wurde. Insgesamt wurden, inklusive einer abschließenden Validierung auf einer sehr ausführlichen Softwarearchitekturdokumentation, in den Iterationen 17 Softwarearchitekturdokumentationen gesichtet und klassifiziert. Die klassifizierten Fallstudien dienen zum einen einer Auswertung des Schemas und zum anderen als Korpus für die automatisierte Analyse.

4.2. Auswahl der Fallstudien

Neben der manuellen und automatisierten Analyse der Entwurfsentscheidungen ist auch das Auffinden von Softwareprojekten mit ausreichend ausführlicher Softwarearchitekturdokumentation in natürlicher Sprache Teil dieser Arbeit. Insbesondere bei Open-Source-Projekten, die nicht von vornherein in Entwicklungsphasen aufgeteilt wurden und deren Softwareartefakten kein Entwurf vorausging, ist nur eine spärliche Softwarearchitekturdokumentation auffindbar. Außerdem handelt es sich bei den auffindbaren Dokumentationen häufig um Nutzerhandbücher, die den Umgang mit den Benutzerschnittstellen erklären, jedoch nicht deren zugrunde liegende Softwarearchitektur offenbaren. Zu guter Letzt werden häufig Diagramme und Modelle zur Visualisierung der Softwarearchitektur verwendet, ohne jedoch eine natürlichsprachige Beschreibung dieser mitzuliefern, die im Rahmen dieser Arbeit analysiert werden könnte.

Mithilfe einer an Prof. Kozioleks Lehrstuhl erstellten Liste von Softwareprojekten mit vorhandener Dokumentation konnten erste Dokumentationsartefakte identifiziert werden, die über eine ausreichend ausführliche Softwarearchitekturdokumentation verfügen. Darüber hinaus erwies es sich als äußerst fruchtbar, auf *GitHub* nach 'architecture' zu suchen und die Suche auf 'Wikis' zu verfeinern (`https://github.com/search?q=architecture&type=wikis`). Die jeweils vorliegende Architektudokumentation der Softwareprojekte wurde daraufhin anhand der Länge und der Natürlichsprachlichkeit, das heißt, dass vorwiegend Satzstrukturen statt Stichpunkten oder Code-Ausschnitten verwendet wurden, ausgewählt. Eine Übersicht über die verwendeten Softwareprojekte findet sich in Tabelle 4.1.

4.3. Identifikation von Entwurfsentscheidungen

Bevor die Entwurfsentscheidungen anhand des Schemas klassifiziert werden können, muss ihr Auftreten in der natürlichsprachigen Softwarearchitekturdokumentation identifiziert werden. Grundsätzlich ist natürlichsprachiger Text als Komposition zahlreicher Bausteine auf unterschiedlichen Ebenen zu sehen, etwa Buchstaben, Wörter und Sätze. Für eine sinnvolle Klassifikation von Entwurfsentscheidungen ist mindestens die sinntragende Wortebene erforderlich. Wie bereits erläutert, betreffen Entwurfsent-

Nr	Projektname	URL (Abrufdatum)
1	ZenGarden	https://github.com/mhroth/ZenGarden/wiki/Architecture (06.07.2021)
2	Spring XD	https://github.com/ilayaperumalg/spring-xd/wiki/Architecture (08.07.2021)
3	BIBINT	https://github.com/pebbie/BIBINT/wiki/Architecture (08.07.2021)
4	ROD	https://github.com/zywszy/rod/wiki/Architecture (08.07.2021)
5	tagm8vault	https://github.com/metafacets/tagm8-vault/wiki/Architecture (12.07.2021)
6	MunkeyIssues	https://github.com/seandgrimes/MunkeyIssues/wiki/Architecture (12.07.2021)
7	OnionRouting	https://github.com/mangei/onion-routing/wiki/Architecture (12.07.2021)
8	Calipso	https://github.com/arosboro/calipso/wiki/Architecture (14.07.2021)
9	IOSched	https://github.com/google/iosched/blob/main/README.md (23.07.2021)
10	MyTardis	https://github.com/mytardis/mytardis/blob/develop/docs/dev/architecture.rst (26.07.2021)
11	SCons	https://scons.org/doc/production/PDF/scons-design.pdf (26.07.2021)
12	OpenRefine	https://github.com/johnconnelly75/OpenRefine/wiki/Architecture (26.07.2021)
13	Beets	https://github.com/steinitzu/beets/wiki/Architecture (28.07.2021)
14	Teammates	https://github.com/TEAMMATES/teammates/blob/master/docs/design.md (30.07.2021)
15	QMiner	https://github.com/qminer/qminer/wiki/Architecture (28.07.2021)
16	Spacewalk	https://github.com/jdobes/spacewalk/wiki/Architecture (28.07.2021)
17	CoronaWarnApp	https://raw.githubusercontent.com/corona-warn-app/cwa-documentation/master/solution_architecture.md (28.07.2021)

Tabelle 4.1.: Verwendete Projekte mit Softwarearchitekturdokumentation

scheidungen die Softwarearchitektur eines Systems, also dessen Komponenten, deren Beziehungen untereinander und die Umgebung [20]. Während Komponenten gegebenenfalls noch auf Wortebene identifiziert werden können, ist für die Identifikation einer Beziehung zwischen Komponenten eine Satzstruktur erforderlich. Um die Klassifikation der Textbausteine einheitlich zu halten, werden daher grundsätzlich Sätze auf ihre enthaltenen Entwurfsentscheidungen untersucht.

Dabei ist zu beachten, dass nicht alle Sätze in einer Softwaredokumentation auch eine Entwurfsentscheidung beschreiben oder begründen, sodass die Analyse in einem ersten Schritt *Entwurfsentscheidungen* von *Nicht-Entwurfsentscheidungen* trennt. Hierbei folgt die Identifikation einer Entwurfsentscheidung der in Kapitel 2 genannten Definition von Entwurfsentscheidungen, welche dazu dienen, eine oder mehrere Anforderungen an ein Softwaresystem durch architektonische Ergänzungen, Verminderungen und Anpassungen ganz oder teilweise umzusetzen [50]. Da die Klassen *Existenz-*, *Verhaltens-* und *Ausführungsentscheidungen* [27] als Oberklassen im Klassifikationsschema dienen, sind insbesondere alle Sätze, die eine solche Entscheidung beschreiben oder begründen als Entwurfsentscheidung einzuordnen.

Auch Entwurfsentscheidungen, die nicht Kernaussage des Satzes sind, sich aber indirekt aus diesem ableiten lassen, sollen identifiziert werden. Während die Entwurfsentscheidung aus „The Corona-Warn-App uses a new framework provided by Apple and Google called Exposure Notification Framework“ (aus Dokumentation zu *Corona-WarnApp*, siehe Tabelle 4.1) offensichtlich hervorgeht, lässt sich im Satz „In beets, the ‘Query‘ abstract base class represents a criterion that matches items or albums in the database“ (*Beets*, Tabelle 4.1) zwar schnell eine Entscheidung zur Existenz einer Klasse ausmachen, jedoch weist „base class“ darüber hinaus auf eine Vererbungsstruktur hin. Auch die Existenz einer Datenbank wird zumindest erwähnt. Auch aus einzelnen Wörtern lässt sich eine übergeordnete Struktur erschließen, sodass man aus „Data Layer provides efficient access to the data by indexing the records“ (*QMiner*, Tabelle 4.1) eine Schichtenarchitektur und aus „Client-Side | Client Side Architecture: how the UI is built“ (*OpenRefine*, Tabelle 4.1) eine Client-Server-Architektur rekonstruieren kann. Andere Entscheidungen lassen sich auch nur im Kontext eindeutig klassifizieren, wie etwa, dass es sich bei „common.util: Contains utility classes“ (*Teammates*, Tabelle 4.1) um ein Paket handelt, denn die Abschnittsüberschrift in der Dokumentation lautet „Package overview“.

Da, wie in den Beispielen zuvor gesehen, Sätze auch mehrere Entwurfsentscheidungen enthalten können, können Sätze auch mit mehreren Labeln versehen werden. Dies betrifft beispielsweise Sätze der Art "Komponente A kommuniziert über Interface I mit Komponente B", der neben den Existenzentscheidungen der beiden Komponenten auch eine Verhaltensentscheidung über deren Interaktion untereinander enthält. Die primäre Klassifikation soll in einem solchen Fall möglichst die Kernaussage des Satzes widerspiegeln, in diesem Fall also das Verhalten der Komponenten untereinander, da

davon auszugehen ist, dass die Komponenten davor oder danach noch ausführlicher eingeführt und beschrieben werden. Die alternative Klassifikation umfasst daran anschließend die naheliegendste, noch nicht abgedeckte Entwurfsentscheidung. Diese Gewichtung ist dahingehend gewählt, dass in der automatisierten Analyse nicht immer eine Multi-Label-Klassifikation möglich oder erfolgreich ist und daher auch nur mit der primären Klassifikation gearbeitet werden kann. Um den Korpus nicht mit zu vielen Leerfeldern zu füllen, da häufig pro Satz nur eine Entwurfsentscheidung getroffen wird, und um die Klassifkation mit mehreren Labeln anhand von zweidimensionalen Kreuztabellen analysieren zu können, kann genau eine alternative Klasse benannt werden. Dieses Vorgehen erwies sich für nahezu alle Sätze in den analysierten Fallstudien als ausreichend.

4.4. Untersuchung der Fallstudien

Die nachfolgenden Unterabschnitte beschreiben die Erkenntnisse der jeweiligen Iteration im Weiterentwicklungsprozess des Klassifikationsschemas. Sie beinhalten zum einen eine Übersicht über Klassen, welche besonders häufig gesehen wurden, und zum anderen Auffälligkeiten, die zu einer Veränderung und Ausdifferenzierung des Klassifikationsschemas geführt haben.

4.4.1. Iteration 1: Ausdifferenzierung der Verhaltensentscheidungen

Grundlage der ersten Iteration ist die Softwarearchitekturdokumentation des GitHub-Projektes *ZenGarden* (siehe Tabelle 4.1). Hierbei handelt es sich um die Entwicklung einer eigenständigen Bibliothek für die Ausführung von Pure Data Patches. Für die Architektur dieser Bibliothek wurden insbesondere Existenzentscheidungen dokumentiert. Diese betreffen *Klassen* und die Festlegung der zugehörigen *Funktionen und Kommunikation*. Da Objekte konkrete Instanzen einer Klasse sind, werden Entwurfsentscheidungen über die Ausgestaltung von Objekten entsprechend im Klassifikationsschema zugeordnet. Für die Funktionen und Kommunikation des Softwaresystems zeigten sich Verhaltensentscheidungen, die in feinere Klassen der Klassifikationshierarchie untergliedert werden können. Entwurfsentscheidungen zu *Funktionen und Kommunikation* können *Methoden* (*method*), den *Nachrichtenaustausch* (*messaging*) zwischen Systemkomponenten sowie die Verwendung bestimmter *Algorithmen* (*algorithm*) betreffen (vgl. Abbildung 4.1).

Dabei betreffen Entwurfsentscheidungen zu *Methoden* deren Festlegung innerhalb einer Klasse inklusive Parametern und Rückgabewerten, aber auch die Beschreibung von Funktionalität, die Systemkomponenten ausführen können sollen und deren Umsetzung in Code später mittels Methoden erfolgt. Dagegen geschieht der Nachrichtenaustausch auf einer etwas höheren Abstraktionsebene. Er umfasst sowohl Funktionsaufrufe in

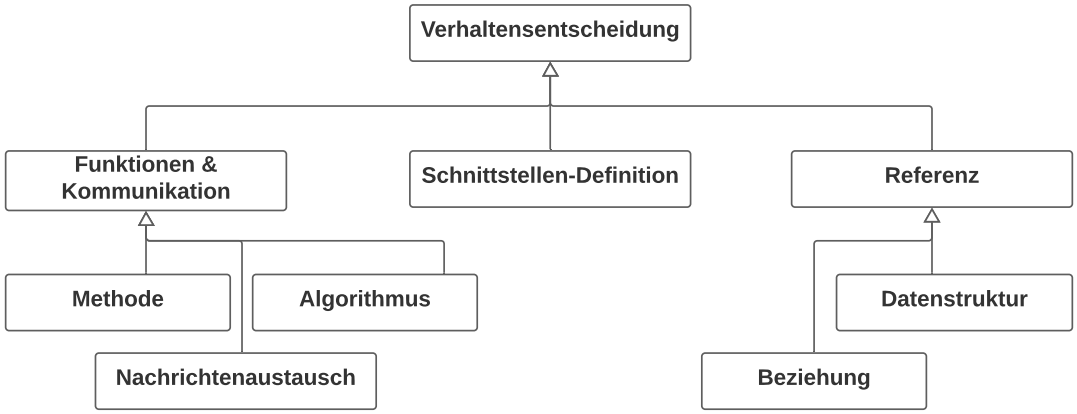

Abbildung 4.1.: Verfeinerung der Klassifikation von Verhaltensentscheidungen

einer bestimmten Abfolge als auch die Kommunikation zwischen Systemkomponenten durch das Versenden von Signalen und Datenpaketen [54]. Während Entwurfsentscheidungen zu *Methoden* also vorwiegend festlegen, über welche Funktionen ein Softwaresystem verfügt, konkretisieren Entwurfsentscheidungen zum *Nachrichtenaustausch*, in welcher Reihenfolge Funktionen aufgerufen werden und inwiefern Daten in Nachrichten verpackt und versendet werden. Das Auftreten einer Entscheidung über den *Nachrichtenaustausch* kann alleinstehend, wie etwa in „A Pd message may contain different kinds of information, primarily floats, symbols (character arrays), and bangs [...]“ (aus Dokumentation zu *ZenGarden*, siehe Tabelle 4.1), auftreten oder auch gemeinsam mit der Festlegung von Methoden: „When a message arrives at an object via the ’receiveMessage()’ [...] and ’processMessage()’ [...] functions, the object queries the message for the number and types of its elements [...]“ (*ZenGarden*). Teilweise gibt es für die Umsetzung bestimmter Funktionalität bereits bewährte Algorithmen. Wird auf die Verwendung dieser verwiesen, so wird die Entwurfsentscheidung mit *Algorithmus* klassifiziert. Denn anders als für selbst festgelegte Methoden werden für Algorithmen die einzelnen Ausführungsschritte nicht mehr angegebenen und für ihre Implementation können Methoden aus externen Bibliotheken herangezogen werden. Ein Beispiel aus *ZenGarden* hierfür lautet: „A breadth-first analysis of the DSP graph is used to order the tree such that children objects are evaluated after their parents.“

Auch für Verhaltensentscheidungen zu Objektverweisen (*Referenz, reference*) ergeben sich durch die manuelle Analyse der Fallstudie Verfeinerungen des Klassifikationsschemas: Objekte können zum einen in Beziehung zueinander stehen und zum anderen in Datenstrukturen abgelegt werden (vgl. Abbildung 4.1). Während *Beziehungen* (*relationship*) Assoziationen, Aggregationen und Kompositionen zwischen Klassen oder Komponenten umfassen, legen Entwurfsentscheidungen zu *Datenstrukturen* (*data struc-*

ture) bereits konkret fest, in welcher Form eine solche Beziehung im Code umgesetzt werden soll, beispielsweise durch Speicherung der Objekte in einer Liste, einem Feld oder einem Graphen. So legt der nachfolgende Satz eine solche Datenstruktur fest: „The list of 'MessageElement's is implemented as growable array and the 'PdMessage' object maintains the number of valid 'MessageElement's in the array [...]“ (*ZenGarden*).

4.4.2. Iteration 2: Ausdifferenzierung der Strukturentscheidungen

Für die zweite Iteration wurden die GitHub-Projekte *SpringXD*, *BIBINT* und *ROD* (siehe Tabelle 4.1) analysiert. Hier zeigte sich erneut die Häufung von Existenzentscheidungen gefolgt von Ausführungsentscheidungen, während die Fallstudien kaum Eigenschaftsentscheidungen aufwiesen. Viele Strukturentscheidungen zu *Klassen* finden sich in der Softwarearchitekturdokumentation von *ROD*. Während ein Satz wie „The central concept is one of a Message Handler class, which relies on simple coding conventions to Map incoming messages to processing methods“ (aus Dokumentation zu *SpringXD*, siehe Tabelle 4.1) die Existenz und Aufgabe einer Klasse „Message Handler“ festlegt, trifft der Satz „In order to be persistable, given object [has] to be an instance of a class that inherits from Rod::Model“ (*ROD*) eine Aussage über eine Vererbungsstruktur und damit über die Existenz von Ober- und Unterklassen. Zudem werden mit dem in der Dokumentation direkt folgenden Satz „That object is identified by its rod_id (ROD identifier) which is ascribed to it, when the object is stored“ (*ROD*) einem Objekt Attribute zugewiesen. Um diese Strukturentscheidungen voneinander abzugrenzen (vgl. Abbildung 4.2), ergibt sich im Klassifikationsschema die Klasse *Objekt* (*object*) im Sinne der objektorientierten Programmierung, also eine Entscheidung über die Möglichkeit einer Erzeugung einer Entität durch die Instanziierung einer Klasse [13]. Die Gesamtheit von Objekten mit gemeinsamen Eigenschaften wird in einer *Klasse* festgelegt [13], wobei diese Eigenschaften sowohl Attribute als auch Methoden sein können. Da die Entwurfsentscheidungen zu *Attributen* zunächst rein strukturell sind, werden Entwurfsentscheidungen zu diesen von der Klasse der *intrasystemischen Strukturentscheidungen* abgeleitet, während Entwurfsentscheidungen zu *Methoden* ein Verhalten bestimmen. Zudem gibt es im objektorientierten Entwurf die Möglichkeit, Unterklassen zu bilden, die die Attribute und Methoden der Oberklasse übernehmen und erweitern [13]. Dies wird als Entscheidung zu *Vererbung* (*inheritance*) klassifiziert.

Handelt es sich bei der gegebenen Softwarearchitekturdokumentation nicht ausschließlich um einen objektorientierten, sondern mitunter auch relationalen Entwurf, so werden die Konzepte des objektorientierten Entwurfs bestmöglich auf die Definition von Tabellen übertragen. Das Tabellenschema kann dabei als Bauplan für die Erzeugung konkreter Objekte, folglich der Tabelleneinträge gesehen werden, und die Spalten einer Tabelle bestimmen die Attribute, die solche Objekte haben müssen. Auch für nicht objektorientierte Sprachen wie C werden Entwurfsentscheidungen, falls möglich, in das für den objektorientierten Entwurf entwickelte Klassifikationsschema überführt,

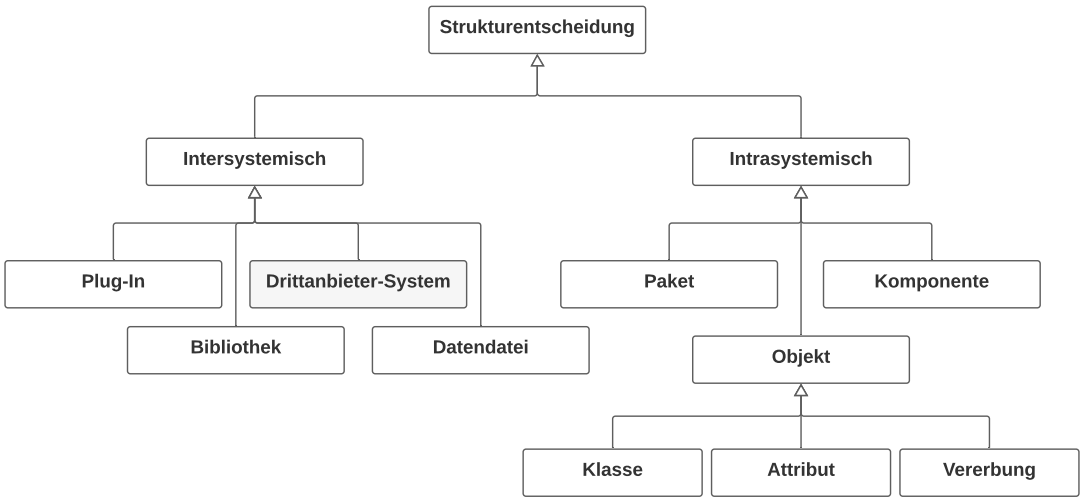

Abbildung 4.2.: Verfeinerung der Klassifikation von Strukturentscheidungen

sodass die Entscheidung „The Rod::JoinElement and Rod::PolymorphicJoinElement define their own C structs“ (*ROD*) zu *Klasse* zugeordnet werden kann, die als Pendant zu C-Strukturen dient.

Hinsichtlich der *intersystemischen Strukturentscheidungen* konnten vorwiegend Entscheidungen zur Verwendung von *Bibliotheken* beobachtet werden, während Entscheidungen zu *Plug-ins* und *Drittanbieter-Systemen* kaum bis gar nicht vorkamen. Allerdings konnte eine vierte Kategorie von intersystemischen Strukturentscheidungen ausgemacht werden, die zum Anlegen und Verändern von nicht-ausführbaren Dateien führen. Solche *Datendateien* (*data file*) enthalten Daten für die Applikation oder das System, unter anderem Eingabe- und Ausgabedaten [52], oder auch Metadaten und können in Form von XML, JSON, YAML oder weiteren Textdateiformaten vorliegen. Dadurch können nun Sätze wie „The database meta-data [...] is kept with it in a database.yml file“ (*ROD*) und „The definition of a module is stored in a Module Registry, which is a Spring XML configuration file“ (*SpringXD*) angemessen klassifiziert werden.

Da sich „The Ruby Object Database is implemented on the basis of Oracle Berkeley DB“ (*ROD*) und „Future releases will support backing the message channel with other transports such as RabbitMQ and JMS“ (*SpringXD*) als *Technologie*-Entscheidungen klassifizieren lassen, jedoch keiner der beiden Unterklassen *Programmiersprache* und *Plattform* zuzuordnen sind, wurde das Klassifikationsschema erweitert. Entwurfsentscheidungen, die die Verwendung von Datenbanken, Frameworks oder externen Schnittstellen beschließen, werden unter *Treiber & Rahmenstruktur* (*driver & framework*) aufgefasst. Dies entspricht dem äußeren Ring der Clean-Code-Architektur nach Robert

Martin [32], die dazu entworfen wurde, Abhängigkeiten in einem Softwaresystem nur in Richtung stabiler Komponenten zuzulassen.

4.4.3. Iteration 3: Festlegung der Funktionalität durch Entwurfsentscheidungen

Das resultierende Klassifikationsschema wurde auf die Softwarearchitekturdokumentation von *tagm8vault*, *MunkeyIssues* und *OnionRouting* (siehe Tabelle 4.1) angewandt. Die Einführung der Klasse *Treiber & Rahmenstruktur* konnte in dieser Iteration genutzt und ihre Berechtigung bestätigt werden: In der Dokumentation zu *MunkeyIssues* werden zahlreiche Technologien wie „Frontend: Aurelia + HTML5“, „API Gateway: ASP:NET Web API“ und „Data Persistence: Entity Framework“ benannt, die dem Softwaresystem als externe Schnittstellen dienen.

Es zeigte sich, dass die Grenzen zwischen den Klassen *Methode* und *Nachrichtenaustausch* mitunter verschwimmen. An folgenden Beispielen kann jedoch verdeutlicht werden, in welche Richtung die jeweiligen Entscheidungen abzielen: Der Satz „In order to maintain loose-coupling between the various services in the application, all communication between the various microservices is accomplished by sending messages across a service bus“ (*MunkeyIssues*) enthält, neben einer Entwurfsentscheidung für den Architekturstil Microservice, eine Festlegung auf die Art des Nachrichtenaustausches. Dagegen ist die Beschreibung einer Funktion, die das Softwaresystem ausführen können soll, eine Entwurfsentscheidung zu Methoden, etwa die Methoden „get-chain“ und „get-available-nodes“ (*OnionRouting*).

Häufig gesehen wurden in dieser Iteration darüber hinaus Entscheidungen zur *Anordnung* von Systemkomponenten. Mit „The Façade design pattern has been used to support an integrated command line server accessed via DRuby“ (*tagm8vault*) und „For cases where an immediate response isn´t neccessary, the more traditional pub/sub approach can be utilized“ (*MunkeyIssues*) werden Entwurfsentscheidungen zu den Architekturmustern Fassade und Publish&Subscribe getroffen. „The application architecture for MunkeyIssues is being implemented with a microservices approach“ (*MunkeyIssues*) bestimmt, dass die Software gemäß dem Architekturstil Microservice verteilt wird, wohingegen „The software implements a a classic 3 tiered architecture [...]“ (*tagm8vault*) ein in drei Ebenen unterteiltes System erwarten lässt.

Nachhaltige Änderungen am Klassifikationsschema ergaben sich nicht. Eine Überlegung war zunächst, die Klasse *Funktionen & Kommunikation* um eine Unterklasse zu *Protokollierung* (*logging*) zu erweitern, da sich zunächst durch „Berkeley DB also supports write ahead-logging, for catastrophic error recovery [...]“ (*ROD*) und "logging can also be done via hazelcast (=> centralized logging)“ (*OnionRouting*) zwei Hinweise ergaben, dass hierfür vermehrt Entwurfsentscheidungen in der Softwarearchitekturdokumentation getroffen werden. Dies ließ sich jedoch in keiner weiteren Fallstudie

bestätigen, sodass Protokollierung unter *Funktionen & Kommunikation* verbleibt und zudem natürlich noch Entwurfsentscheidungen zum einen zu der Berkeley-Datenbank und zum anderen der zu Plattform-Technologie Hazelcast identifiziert wurden.

4.4.4. Iteration 4: Ausdifferenzierung der Technologie-Entscheidungen

In den Untersuchungen der Softwarearchitekturdokumentation zu *Calipso*, *IOSched* und *MyTardis* (siehe Tabelle 4.1) verfestigt sich die Beobachtung, dass sich insbesondere Existenzentscheidungen (sowohl Struktur- als auch Verhaltensentscheidungen) und Technologie-Entscheidungen beobachten lassen. Dabei wurden einige Entwurfsentscheidungen der in Unterabschnitt 4.4.2 eingeführten Klasse *Treiber & Rahmenstruktur* zugeordnet.

Diese Gruppe ist jedoch nicht ausreichend homogen, sodass das Klassifikationsschema erneut aufgespalten wurde. So können Entwurfsentscheidungen zu Frameworks („MyTardis is built on the Django web framework“, *MyTardis*), Programmierschnittstellen („A RESTful API provides anonymous and authenticated access to most of the stored data“, *MyTardis*), Benutzeroberflächen („Module [...] Management: Additional commands for the Django CLI“, *MyTardis*) und Datenbanken („The backend storage for both content and sessions is stored in MongoDB, which can be sharded and scaled as required“, *Calipso*) getroffen werden. Dementsprechend lassen sich Technologie-Entscheidungen in *Programmiersprache*, *Plattform*, *Framework* und *Randschnittstellen* (*boundary interface*) unterteilen, wobei die Klasse *Randschnittstelle* erstens *Programmierschnittstellen* (*API*), zweitens *Benutzerschnittstellen* (*user interface*) und drittens *Datenbanken* (*database*) umfasst (vgl. Abbildung 4.3). Randschnittstellen befinden sich dabei an den Grenzen zwischen verschiedenen (Sub-)Systemen und können im gleichen Projekt entwickelt worden sein oder den Entwicklern anderweitig zur Verfügung stehen. Entscheidungen zur *Benutzerschnittstelle* können sowohl graphische als auch zeichenorientierte Benutzeroberflächen, etwa eine Kommandozeile, betreffen [13, S. 102f]. Unter *Datenbank* fällt sowohl der Aufbau eines Systems zur Datenorganisation als auch die Festlegung auf bestimmte Datenbanktechnologien.

Als Folge dieser Veränderung des Schemas entfällt die Klasse *Drittanbieter-System* aus dem resultierenden Klassifikationsschema. Die Anbindung eines Drittanbieter-Systems erfolgt über eine externe Schnittstelle und wird nun über die Klasse *Randschnittstelle* erfasst. Anderweitig konnten in den bis hierhin gesehenen Fallstudien keine Entscheidungen zu Drittanbieter-Systemen identifiziert werden, die nicht auch durch den in Abbildung 4.3 dargestellten Ausschnitt des Schemas vollständig und nachvollziehbar klassifiziert werden können. Zu beleuchten ist an dieser Stelle noch, dass die Entwurfsentscheidungen nun hierarchisch unter den Technologie- und damit den Ausführungsentscheidungen eingeordnet werden und nicht unter den Strukturentscheidungen als Unterklasse der Existenzentscheidungen. Betrachtet man den Satz „Calipso is at its heart an Express application (expressjs.org) backed by MongoDB“ (*Calipso*) mit Ent-

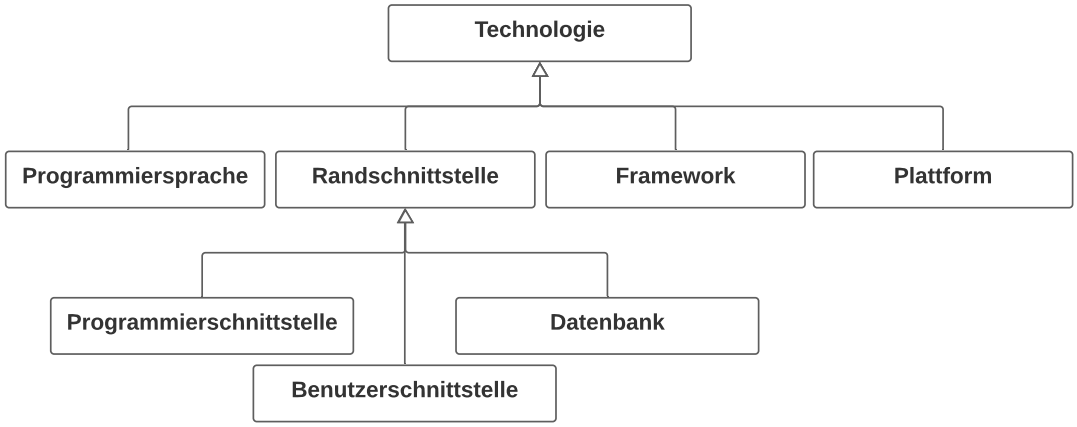

Abbildung 4.3.: Verfeinerung der Klassifikation von technologischen Entscheidungen

scheidungen zum Express.js-Framework und MongoDB als Datenbank, so betrifft dieser eher die Ausführung des Softwaresystems als die Erstellung neuer Systemkomponenten inklusive Quellcode, denn diese bestehen ja bereits extern und werden nun genutzt.

Nicht gänzlich unähnlich ist die Betrachtung für die Klasse *Bibliothek*, die neben im Softwareprojekt eigens erstellten Bibliotheken auch externe Bibliotheken erfasst. Jedoch wird die Verwendung von Bibliotheken durch Import-Aufrufe direkt im Quellcode adressiert. Außerdem stehen Entwurfsentscheidungen zu *Bibliotheken* den Entscheidungen zur Existenz von Paketen und bestimmten Funktionen näher als dies etwa Entscheidungen zu Frameworks und Datenbanken tun. Aufgrund dieser Überlegungen heraus und um nicht mit den Erkenntnissen aus Bhat et al. [6] zu brechen, die Grundlage der Einführung der Klasse *Bibliothek* waren, verbleibt die Klasse *Bibliothek* als Unterklasse der intersystemischen Strukturentscheidungen. Hierbei wird die Klassifikation *intersystemisch* auch der Beobachtung gerecht, dass solche Entscheidungen über die Grenzen eines Systems hinaus wirken.

Entscheidungen, die die Lizenzierung eines Softwareprojektes betreffen, werden der Klasse *Organisation* zugeordnet. Die Lizenzierung umgibt die Entwicklung eines Softwareprojektes und ist damit eine Ausführungsentscheidung. Während die Klasse *Prozess* den Entwicklunsgprozess als solchen umfasst, können innerhalb der Organisation finanzielle, rechtliche und organisatorische Rahmenbedingungen festgelegt werden. Eine solche rechtliche Rahmenbedingung ist die Festlegung auf eine Lizenz, so etwa „Licensed under the Apache License, Version 2.0“ (*IOSched*).

4.4.5. Iteration 5: Häufiges Vorkommen von Entwurfsentscheidungen zu Klassen, Methoden und Technologien

Für die Softwarearchitekturdokumentation der Projekte *SCons*, *OpenRefine* und *Beets* (siehe Tabelle 4.1) konnten in der fünften Iteration erstmalig alle identifizierten Entwurfsentscheidungen nicht nur vollständig, sondern auch zufriedenstellend und nachvollziehbar klassifiziert werden. Somit sind auch das angewandte und resultierende Klassifikationsschema miteinander identisch.

In der Softwarearchitekturdokumentation von *SCons* und *Beets* zeigt sich die besonders die Häufigkeit von Entwurfsentscheidungen aus den Klassen *Methode* und *Klasse*. Sie stellen den Kern von objektorientierter Software dar und dienen den Softwareentwicklern dazu, Funktionalität innerhalb festgelegter Objekte umzusetzen. Dabei werden diese jedoch unterschiedlich ausführlich dokumentiert, wie der Vergleich der Beispiele aus den Fallstudien zeigt: „The class method ‘Item.from_path(path)‘ conveniently constructs a new item with the metadata read from a file“ (*Beets*) bietet gegenüber der reinen stichwortartigen Auflistung „get-chain“ und „get-available-nodes“(*OnionRouting*) eine deutlich aussagekräftigere Architekturdokumentation. Die in dieser Iteration betrachteten Fallstudien zeichneten sich durch gut formulierte Softwaredokumentation aus, wie die folgenden Sätze noch einmal verdeutlichen: „The Node class is sub-classed to represent external objects of specific type: files, directories, database fields or records“ (klassifiziert als *Vererbung* und *Klasse*, aus Dokumentation von *SCons*) und „’add(item)’: Add an ’Item’ to the database [...]“ (klassifiziert als *Methode* und *Objekt*, aus *Beets*).

In der Dokumentation zu *OpenRefine* finden sich vor allem technologische Entscheidungen. So lässt sich der Satz „The server-side part of OpenRefine is implemented in Java as one single servlet which is executed by the Jetty [...] web server + servlet container“ (*OpenRefine*) aufgrund der Entscheidung für Java der Klasse *Programmiersprache* zuordnen und enthält mit Jetty eine Festlegung auf eine *Plattform* für die Ausführung.

4.4.6. Iteration 6: Das Klassifikationsschema in seiner Breite

Grundlage der sechsten Iteration sind die Softwarearchitekturdokumentationen zu *Teammates*, *QMiner* und *Spacewalk* (siehe Tabelle 4.1). Da sich erneut keine notwendigen Änderungen am Klassifikationsschema ergeben haben, kann das Klassifikationsschema als hinreichend ausgereift betrachtet werden. Zum Abschluss wurde daraufhin noch die Dokumentation der *CoronaWarnApp* herangezogen, die mit 480 analysierten Sätzen neben *Teammates* (342) eine sehr ausführliche Dokumentation bereithält und dadurch geeignet ist, das finale Schema in seiner Gänze anzuwenden.

Anhand der Dokumentation zu *Teammates* lässt sich die Zerlegung eines Softwaresystems entlang unterschiedlicher Abstraktionsniveaus in kleinere Einheiten nachvollziehen. Hierfür dienen Strukturentscheidungen. So lassen sich hier zahlreiche Entwurfsentscheidungen identifizieren, die den Klassen *Paket*, *Komponente* und *Objekt*

(vgl. Abbildung 4.2) zugeordnet werden können. *Teammates* verfügt über verschiedene Komponenten, also zusammengesetzte Einheiten mit festgelegten Schnittstellen und einem Kontextbezug, die voneinander unabhängig bereitgestellt und zusammengefügt werden können [48]. Beispielsweise wird in der Dokumentation die folgende Komponente festgelegt: „The 'Logic' component handles the business logic of TEAMMATES" (*Teammates*). Um die Entwicklung und Wartung des Systems zu vereinfachen, wurden Pakete eingeführt, die Klassen und Strukturen mit in sich geschlossener Funktionalität an einen Namensraum binden [13, S. 648]. So unterteilt sich die „Logic Component" in „'logic.api': Provides the API of the component to be accessed by the UI" und „'logic.core': Contains the core logic of the system" (*Teammates*). Hierhin finden sich wiederum Klassen wie beispielsweise „'Logic': A Facade class [...] which connects to the several 'Logic' classes to handle the logic related to various types of data" und „'EmailGenerator': Generates emails to be sent" (*Teammates*).

In *QMiner* sind insbesondere Verhaltensentscheidungen vorherrschend, sowohl zu *Methoden* und *Algorithmen*, aber auch zu *Architekturmustern*. So legt der Satz „Count – computes a distribution over a discrete field (e.g. string, integer)" (*QMiner*) eine Methode fest, die das Softwaresystem bereitstellen soll. Die entsprechende Funktionalität kann daraufhin durch die Verwendung der genannten Algorithmen implementiert werden, etwa „Hoeffding trees – Decision trees learning algorithm optimized for stream processing (under development)" (*QMiner*). Auch Architekturmuster helfen dabei, Architekturlösungen für wiederkehrende Probleme zu schaffen. Der Satz „ [The] Data Layer accesses the data through adapters, which must expose the data sources through a predefined interface" (*QMiner*) offenbart dabei das Architekturmuster Adapter.

Wie abhängig der Anteil getroffener Existenz-, Eigenschafts- und Ausführungsentscheidungen von der jeweiligen Softwarearchitekturdokumentation ist, zeigt sich zum Beispiel im Vergleich von *Spacewalk* mit den anderen Dokumentationen. Während Existenzentscheidungen in den meisten Dokumentationen und auch insgesamt den größten Anteil gesehener Entwurfsentscheidungen ausmachen, überwiegen in *Spacewalk* die Ausführungsentscheidungen, genauer die Technologie-Entscheidungen. Diese betreffen die verwendete Programmiersprache („Any new web UI development is being done in Java", *Spacewalk*), Benutzerschnittstelle und Ausführungsplattform („The web UI, which until recently consisted entirely of perl running through an Apache web server", *Spacewalk*) oder auch Datenbanktechnologien („[...] the data tier is backed by an Oracle database or by PostgreSQL database", *Spacewalk*).

In der Dokumentation zur *CoronaWarnApp* wird eine ganze Bandbreite an Entwurfsentscheidungen abgedeckt. Auch hier finden sich häufig Existenzentscheidungen, vor allem solche, die zur Klasse *Attribut* („Total Risk Score: calculated exposure risk level (with a range from 0-4096) according to the defined parameters", *CoronaWarnApp*) und zur Klasse *Methode* („Using the integrated functionality of the Corona-Warn-App to retrieve the results of a SARS-CoV-2 test from a verification server", *CoronaWarnApp*)

zugeordnet werden können. Die Ausführungsentscheidungen betreffen wiederholt das verwendete Framework („The Corona-Warn-App uses a new framework provided by Apple and Google called Exposure Notification Framework“).

Besonders auffällig ist jedoch in der Dokumentation der *CoronaWarnApp* das gehäufte Vorkommen von Eigenschaftsentscheidungen. Zuvor konnten nur in den Dokumentationen zu *Teammates* und *ROD* vermehrt solche Entscheidungen beobachtet werden, wenn auch nicht in der Anzahl wie in *CoronaWarnApp*. Eigenschaftsentscheidungen drücken dabei übergreifende Eigenschaften oder Qualitäten eines Softwaresystems aus [27]. Da sich für die Entwicklung der *CoronaWarnApp* intensive Gedanken über nicht-funktionale Anforderungen, allem voran den Datenschutz, gemacht wurden, lassen sich vermutlich auch vermehrt Eigenschaftsentscheidungen in der Dokumentation finden. Für andere Open-Source-Projekte spielen diese Überlegungen möglicherweise eine untergeordnete Rolle und werden insbesondere weniger dokumentiert als die zu implementierende Funktionalität. Die Eigenschaftsentscheidungen werden in *Einschränkung*, *Entwurfsregel* und *Richtline* unterteilt [27, S. 2]. Der Satz „For privacy reasons, it is not possible to track encounters with other individuals across multiple days“ (*CoronaWarnApp*) ist dabei der Klasse *Einschränkung* zuzuordnen, erkennbar an der negativen Konnotation des Satzes. Dagegen ist „The HTTP method POST is used instead of GET for added security, so data (e.g. the registration token) can be transferred in the body“ (*CoronaWarnApp*) eine positiv formulierte *Entwurfsregel* und „To ensure roaming qualities (protocol interoperability with servers in other geographical regions), it is planned to move to a single agreed protocol once finally defined“ (*CoronaWarnApp*) eine *Richtlinie* für die Weiterentwicklung des Systems. Die hier genannten Eigenschaftsentscheidungen betreffen Qualitätsanforderungen zu Datenschutz, Sicherheit und Interoperabilität. Somit ist auch das Vorkommen der Klassen *Einschränkung*, *Entwurfsregel* und *Richtlinie* im finalen Klassifikationsschema berechtigt. Eine weitere Verfeinerung in Unterklassen lässt keinen Mehrwert erwarten, da nicht genügend Fallbeispiele auffindbar wären.

5. Auswertung des finalen Klassifikationsschemas

Das finale Klassifikationsschema ist das Ergebnis theoretischer Überlegungen mithilfe von verwandten Arbeiten und einer praktischen Auswertung anhand der auffindbaren Softwarearchitekturdokumentation in den ausgewählten Fallstudien (siehe Tabelle 4.1). Besonders beeinflusst wurde die Struktur des Klassifikationsschemas dabei durch die Ontologie nach Kruchten [27] und die von Bhat et al. aufgestellten Regeln zur manuellen Klassifikation von Entwurfsentscheidungen [6, S. 8]. In diesem Kapitel findet sich ein Überblick über alle Klassen des finalen Klassifikationsschemas (Abschnitt 5.1) und eine Auswertung zu ihren Häufigkeiten in den analysierten Fallstudien (Abschnitt 5.2). Abschließend wird die Validität des Klassifikationsschemas beurteilt und bestehende Gefahren hierfür werden diskutiert (Abschnitt 5.3).

5.1. Übersicht über die Klassen

Im Folgenden werden die Klassen des Schemas zusammenfassend erläutert. In Kapitel 4 findet sich mehr über ihre Entstehung und die zugrunde liegenden Überlegungen. Für die Übersicht der Klassen wird das hierarchisch gegliederte, finale Schema (siehe Abbildung 5.1 und Abbildung 5.2) ähnlich einer Tiefensuche durchlaufen. Ausgangspunkt sind dabei die Klassen Existenz-, Eigenschafts- und Ausführungsentscheidungen. Aufgeführt ist zu jeder Klasse der Klassenname in Deutsch und Englisch sowie die hierarchische Tiefe und die Oberklasse (O.-Kl.).

5.1.1. Unterklassen der Existenzentscheidung

Existenzentscheidung (existence decision), Tiefe: 1, O.-Kl.: Entwurfsentscheidung
: Eine Existenzentscheidung bestimmt, ob ein Softwareelement oder -artefakt im Entwurf oder der Implementation vorhanden sein wird [27, S. 2].

Strukturentscheidung (structural decision), Tiefe: 2, O.-Kl.: Existenzentscheidung
: Strukturentscheidungen dienen zur Untergliederung des Softwaresystems in wiederverwendbare Subsysteme, Komponenten und Objekte. Diese können hierdurch hinzugefügt, angepasst oder entfernt werden [6, S. 8].

Intersystemisch (intersystem), Tiefe: 3, O.-KL.: Strukturentscheidung
Intersystemische Strukturentscheidungen wirken sich über die Systemgrenzen hinweg aus, indem die hiermit verbundene Erstellung von Softwareelementen stets mit dem Ziel verknüpft ist, die Softwareelemente in andere Systeme einzubauen oder Daten einzuspeisen.

Plug-in (plug-in), Tiefe: 4, O.-Kl.: Intersystemisch
Hiermit werden Entscheidungen zum Hinzufügen oder Entfernen von Plug-ins getroffen. Plug-ins sind Programmteile, die die einbettende Applikation um oft sehr spezialisierte Funktionen oder Funktionen aus einem anderen Anwendungsbereich erweitern [13, S. 681].

Bibliothek (library), Tiefe: 4, O.-Kl.: Intersystemisch
Die Verwendung von Bibliotheken erlaubt den Zugriff auf bereits für andere Systeme entwickelte Funktionalität. Diese Klasse umfasst dabei sowohl innerhalb des gleichen Projektes entwickelte als auch anderweitig hinzugezogene Bibliotheken. Bibliotheken stellen dabei eine „Sammlung häufig benutzter Systemroutinen“ [13, S. 110] dar. Einzeln beschriebene Funktionalität fällt unter die Klasse *Methode*.

Datendatei (data file), Tiefe: 4, O.-Kl.: Intersystemisch
Durch das Hinzufügen, Anpassen oder Entfernen von nicht-ausführbaren Datendateien können Daten oder Metadaten an die Applikation oder das System übergeben werden, unter anderem Eingabe- und Ausgabedaten [52]. Dateiformate, die auf Entscheidungen zu Datendateien hinweisen, sind XML, JSON, CSV etc.

Intrasystemisch (intrasystem), Tiefe: 3, O.-Kl.: Strukturentscheidung
Intrasystemische Strukturentscheidungen sind das Gegenstück zu den intersystemischen. Sie betreffen die Gliederung innerhalb des betrachteten Systems in Subsysteme und kleinere Einheiten.

Paket (package), Tiefe: 4, O.-Kl.: Intrasystemisch
Ob und in welcher Konstellation Softwareelemente gebündelt und mit einem gemeinsamen Namensraum versehen werden, entscheidet sich durch die Strukturierung in Pakete [31, S. 239]. Die Bezeichnung entstammt der in UML und Java verwendeten Terminologie; entsprechende Konzepte, beispielsweise Module in Python, sollen analog zugeordnet werden.

Komponente (component), Tiefe: 4, O.-Kl.: Intrasystemisch
Komponenten sind zusammengesetzte Einheiten mit vertraglich festgelegten Schnittstellen und explizitem Kontextbezug, die voneinander unabhängig bereitgestellt und zusammengefügt werden können [48]. Bei Entscheidungen zu

Subsystemen ist zu bewerten, ob das Subsystem hinsichtlich seiner Größe als Komponente gesehen werden kann.

Objekt (object), Tiefe: 4, O.-Kl.: Intrasystemisch

Entwurfsentscheidungen zu Objekten beschließen, ob ein Objekt als Entität durch die Instanziierung einer Klasse im Softwaresystem erzeugt werden soll [13, S. 625]. Objekte sind dabei Abbilder realer Gegenstände oder gedanklicher Konstrukte in ein Softwaresystem und verfügen über Eigenschaften und Funktionen.

Klasse (class), Tiefe: 5, O.-Kl.: Objekt

Um Objekte instantiieren zu können, werden Objekte mit gemeinsamen Merkmalen zu Klassen zusammengefasst, welche als eine Art Bauplan dienen. Entwurfsentscheidungen zu Klassen betreffen deren Existenz, Aufbau und Namen.

Attribut (attribute), Tiefe: 5, O.-Kl.: Objekt

Attribute sind Eigenschaften einer Menge von Objekten. Während Methoden das Verhalten eines Systems bestimmen, sind Attribute struktureller Natur. Auch das Vorhandensein von Markierungen (Flags/ Labels) für Objekte wird als diese Art der Existenzentscheidungen aufgefasst.

Vererbung (inheritance), Tiefe: 5, O.-Kl.: Objekt

Mittels Vererbung werden Unterklassen gebildet, die die Attribute und Methoden der übergeordneten Klasse (Oberklasse) übernehmen [13, S. 960]. Entscheidungen hierzu umfassen die Existenz und Ausgestaltung von Vererbungshierachien und die Festlegung von Klassen als (abstrakte) Ober- und Unterklassen.

Verhaltensentscheidung (behavioral decision), Tiefe: 2, O.-Kl.: Existenzentscheidung

Verhaltensentscheidungen bestimmen, wie Softwareelemente untereinander interagieren und verbunden sind, um Funktionalität bereitzustellen oder dem System bestimmte Qualitäten zu verleihen [27, S. 2]. Hierunter fallen das Hinzufügen und Überarbeiten von Funktionen, das Bündeln dieser als Schnittstellen und das Einführen oder Entfernen von Abhängigkeiten [6, S. 8].

Funktionen & Kommunikation (functions & communication), Tiefe: 3, O.-Kl.: Verhaltensentscheidung

Hierunter fallen Entwurfsentscheidungen, die in der Konsequenz dem Softwaresystem Funktionalität hinzufügen, Funktionalität verändern und entfernen. Dazu zählen die Festlegung auf Methoden, die Methodenaufrufe und Kommunikation zwischen Softwareelementen und die Entwicklung von Problemlösungsverfahren.

Methode (method), Tiefe: 4, O.-Kl.: Funktionen & Kommunikation

Methoden dienen der Implementation von Funktionalität in ein Softwaresystem (manchmal auch Feature genannt), indem durch Operationen der Zustand eines

Objekts verändert wird [13, S. 562]. Methoden können dabei über ihren Methodenkopf (Methodenname, Sichtbarkeit, Parameter, Rückgabewert) und ihren Methodenrumpf (Programmlogik) spezifiziert werden.

Nachrichtenaustausch (messaging), Tiefe: 4, O.-Kl.: Funktionen & Kommunikation
Entscheidungen zum Nachrichtenaustausch sind eng verwandt mit denen zu Methoden, beziehen sich jedoch nicht auf die Festlegung einzelner Methoden, sondern deren Aufruf in einer bestimmten Abfolge und auf das Versenden von Signalen und Datenpaketen zwischen Softwareelementen zur Kommunikation [54]. Eine Nachricht ist dabei als eine „Funktion, welche eine Information zwecks Weitergabe (Kommunikation) darstellt“ [13, S. 599], zu sehen.

Algorithmus (algorithm), Tiefe: 4, O.-Kl.: Funktionen & Kommunikation
Wenn auf bekannte Algorithmen und ihre Verwendung oder Nicht-Verwendung verwiesen wird, um Funktionalität im System umzusetzen, handelt es sich um eine Entwurfsentscheidung zu einem Algorithmus. Dabei muss die Abfolge des Problemlösungsverfahrens hinreichend allgemein sein und ein übergeordnetes Ziel benannt sein; sonst liegt nur eine Entscheidung zu einer Methode oder zum Nachrichtenaustausch vor.

Referenz (reference), Tiefe: 3, O.-Kl.: Verhaltensentscheidung
Mit Verweisen auf andere Objekte werden Abhängigkeiten in einem Softwaresystem geschaffen. Durch Entwurfsentscheidungen hierzu können solche Objektverweise eingeführt, verändert oder ausgeschlossen werden. Dadurch bestimmt sich, ob auf die Funktionalität des referenzierten Objekts zugegriffen werden kann.

Beziehung (relationship), Tiefe: 4, O.-Kl.: Referenz
Entwurfsentscheidungen bezüglich Beziehungen umfassen Assoziationen, Aggregationen und Kompositionen zwischen Objekten, inklusive der Festelgung von Multiplizitäten. Außerdem können sie Abhängigkeiten zwischen Systemkomponenten betreffen.

Datenstruktur (data structure), Tiefe: 4, O.-Kl.: Referenz
Datenstrukturen dienen der Strukturierung von Objektverweisen und stellen zusätzlich datenstrukturspezifische Funktionalität bereit. Hierunter fällt etwa die Verwendung von Feldern, Listen, Warteschlangen oder Stapeln.

Schnittstellen-Definition (interface definition), Tiefe: 3, O.-Kl.: Verhaltensentscheidung
Schnittstellen deklarieren eine Menge von öffentlich einsehbaren Eigenschaften und Verpflichtungen, die implementiert einen kohärenten Service ergeben [31, S. 169]. Durch Schnittstellen wird Verhalten gebündelt und Implementierungsdetails werden verborgen.

Anordnungsentscheidung (arrangement decision), Tiefe: 2, O.-Kl.: Existenzentscheidung
Durch die Entscheidung für die Anordnung der Softwareelemente in einer bekannten Weise und unter Anwendung der damit verbundenen Prinzipien für die Softwarearchitektur, wird die Implementierung und Wartung des Systems vereinfacht. Aus solchen Anordnungsentscheidungen folgt die Existenz zumeist mehrerer Softwareelemente sowie Abhängigkeiten dieser in vorgegebener Form.

Architekturmuster (architectural pattern), Tiefe: 3, O.-Kl.: Anordnungsentscheidung
Es handelt sich um ein Architekturmuster, wenn eine bekannte Lösung für ein wiederkehrendes Problem verwendet wird, bei der verschiedene Kräfte auf der Ebene der Architektur in Einklang gebracht werden müssen [43]. Beispiele hierfür sind etwa Model-View-Controller, Adapter oder Fassade.

Architekturstil (architectural style), Tiefe: 3, O.-Kl.: Anordnungsentscheidung
Architekturstile umfassen Lösungsprinzipien, die unabhängig von der spezifischen Applikation sind und für die gesamte Architektur eines Softwaresystems einheitlich verwendet werden sollten [43]. Hierunter fallen unter anderem die Schichtenarchitektur, Client-Server-Architektur und Microservice-Architektur.

Referenzarchitektur (reference architecture), Tiefe: 3, O.-Kl.: Anordnungsentscheidung
Mit Referenzarchitekturen werden Domänenkonzepte, Komponenten und Subsysteme definiert, die für genau einen Anwendungsfall durch konkrete Instanzen genutzt werden können [43]. Es bestehen Referenzarchitekturen für u.a. Portalsoftware, Data-Warehouse-Systeme und förderierte Datenbanksysteme [55].

5.1.2. Unterklassen der Eigenschaftsentscheidung

Eigenschaftsentscheidung (property decision), Tiefe: 1, O.-Kl.: Entwurfsentscheidung
Eigenschaftsentscheidungen drücken eine dauerhafte und systemübergreifende Eigenschaft oder Qualität eines Softwaresystems aus [27]. Darunter fällt die Festlegung auf nicht-funktionale Anforderungen.

Einschränkung (constraint), Tiefe: 2, O.-Kl.: Eigenschaftsentscheidung
Einschränkungen legen eine Eigenschaft oder Funktionalität fest, die das System nicht aufweisen wird [27, S. 2], um dadurch im Umkehrschluss bestimmte nicht-funktionale Anforderungen an das System zu erfüllen. Daher sind diese Entscheidungen auch negativ formuliert [27, S. 2].

Entwurfsregel (design rule), Tiefe: 2, O.-Kl.: Eigenschaftsentscheidung
Entwurfsregeln sind positiv formulierte Anforderungen an die Eigenschaften eines Softwaresystems [27, S. 2]. Sie legen fest, nach welchen Regeln das System entworfen werden muss.

Richtlinie (guideline), Tiefe: 2, O.-Kl.: Eigenschaftsentscheidung
Richtlinien sind ebenfalls positiv formuliert [27], jedoch fehlt der zwingende Charakter einer Entwurfsregel. Richtlinien sind bei der Entwicklung des Softwaresystems als Leitfaden zu verstehen.

5.1.3. Unterklassen der Ausführungsentscheidung

Ausführungsentscheidung (executive decision), Tiefe: 1, O.-Kl.: Entwurfsentscheidung
Ausführungsentscheidungen betreffen nicht direkt die Entwurfselemente des Systems, sondern die Umgebung, in der das System entwickelt wird und wie dieses ausgestaltet wird [27, S. 2]. Dieses Umfeld beeinflusst die Architektur der Software dahingehend, dass es Rahmenbedingungen setzt und indirekt Softwareelemente notwendig macht.

Organisation (organization), Tiefe: 2, O.-Kl.: Ausführungsentscheidung
Organisatorische Entscheidungen betreffen die Projektstruktur, Verantwortlichkeiten der Mitarbeiter und das Projektmanagement. Zudem werden hierzu Entscheidungen zur Lizenzierung der Software zugeordnet (vgl. Unterabschnitt 4.4.4).

Prozess (process), Tiefe: 2, O.-Kl.: Ausführungsentscheidung
Mit dieser Klasse sind Entscheidungen aus dem Entwicklungsprozess verknüpft, etwa bezüglich des methodischen Vorgehens oder des zeitlichen Ablaufs [27, S. 2]. Sie umfasst dabei das Release-Mangament sowie Genehmigungs- und Abnahmeverfahren.

Werkzeug (tool), Tiefe: 2, O.-Kl.: Ausführungsentscheidung
Werkzeuge sind „Software-Hilfsmittel [...] zur Erleichterung des Umgangs mit Hardware und Software bzw. zur Entwicklung von Systemen“ [13, S. 996]. Beispielsweise handelt es sich bei Gradle um ein Build-Management-Automatisierungswerkzeug [53].

Technologie (technology), Tiefe: 2, O.-Kl.: Ausführungsentscheidung
Diese Klasse von Entscheidungen bestimmt über den Einsatz bestimmter Technologien bei der Entwicklung von Softwaresystemen. Hierbei kann es sich etwa um Programmiersprachen und Plattformen [27, S. 2], aber auch um Technologien wie QR-Codes handeln.

Programmiersprache (programming language), Tiefe: 3, O.-Kl.: Technologie
Mit der Festlegung auf eine Programmiersprache für ausgewählte Systemkomponenten oder das gesamte Softwaresystem werden syntaktische und semantische Regeln zum Schreiben des Quellcodes festgelegt. Weit verbreitete Programmiersprachen sind Java, Python und C++. Auch Entscheidungen zur Verwendung bestimmter Protokolle, etwa HTTP und TCP, sind in dieser Klasse vereint.

Plattform (platform), Tiefe: 3, O.-Kl.: Technologie

Entwurfsentscheidungen zur verwendeten Plattform können sowohl die Software- als auch die System-Entwicklungsumgebung beschreiben und umfassen somit neben Softwarekomponenten auch die eingesetzte Hardware und das Betriebssystem [13, S. 679]. So ist eine Entscheidung für J2EE hier zuzuordnen, aber auch für welche Betriebssysteme entwickelt wurde (z.B. Android) und wie die Software eingesetzt wird (Deployment).

Framework (framework), Tiefe: 3, O.-Kl.: Technologie

Ein Framework bündelt eine Menge von Dokumenten, Editoren, Bibliotheken, Compilern und Werkzeugen zur Entwicklung umfangreicher Anwendungen [13, S. 345]. Dies gilt etwa für das Django Web Framework.

Randschnittstelle (boundary interface), Tiefe: 3, O.-Kl.: Technologie

Um bereits bestehende Systeme an das zu entwickelnde Softwaresystem anzuschließen, werden die Schnittstellen der bestehenden Systeme verwendet. Dabei ist diese Klasse abzugrenzen von Existenzentscheidungen zur *Schnittstellen-Definition*: Bei einer Schnittstellen-Definition werden eigene Methoden zu einer Schnittstelle gebündelt, während bei Randschnittstellen bereits bestehende Software über eine Schnittstelle benutzt wird.

Programmierschnittstelle (API), Tiefe: 4, O.-Kl.: Randschnittstelle

Programmierschnittstellen zur Anwendungsprogrammierung bieten eine Menge von Funktionen und Objekten, die verwendet werden können, um das Softwaresystem zu entwickeln. Dies kann zum Beispiel die REST (REpresentational State Transfer) API sein.

Benutzerschnittstelle (user interface), Tiefe: 4, O.-Kl.: Randschnittstelle

Die Benutzerschnittstelle befindet sich zwischen dem (menschlichen) Benutzer und dem technischen Softwaresystem (soziotechnisch) und erlaubt die Bedienung und Verwendung der Anwendung. Die Benutzungsoberfläche kann dabei graphisch oder zeichenorientiert sein [13, S. 102f]

Datenbank (data base), Tiefe: 4, O.-Kl.: Randschnittstelle

In diese Klasse fallen Entscheidungen, die für die Speicherung von Daten objektorientierte, relationale oder andere Datenbanken vorschlagen. Auch konkrete Technologien wie SQL als Abfragesprache für Datenbanken oder MongoDB als Datenbankmanagementsystem werden in dieser Klasse aufgefasst.

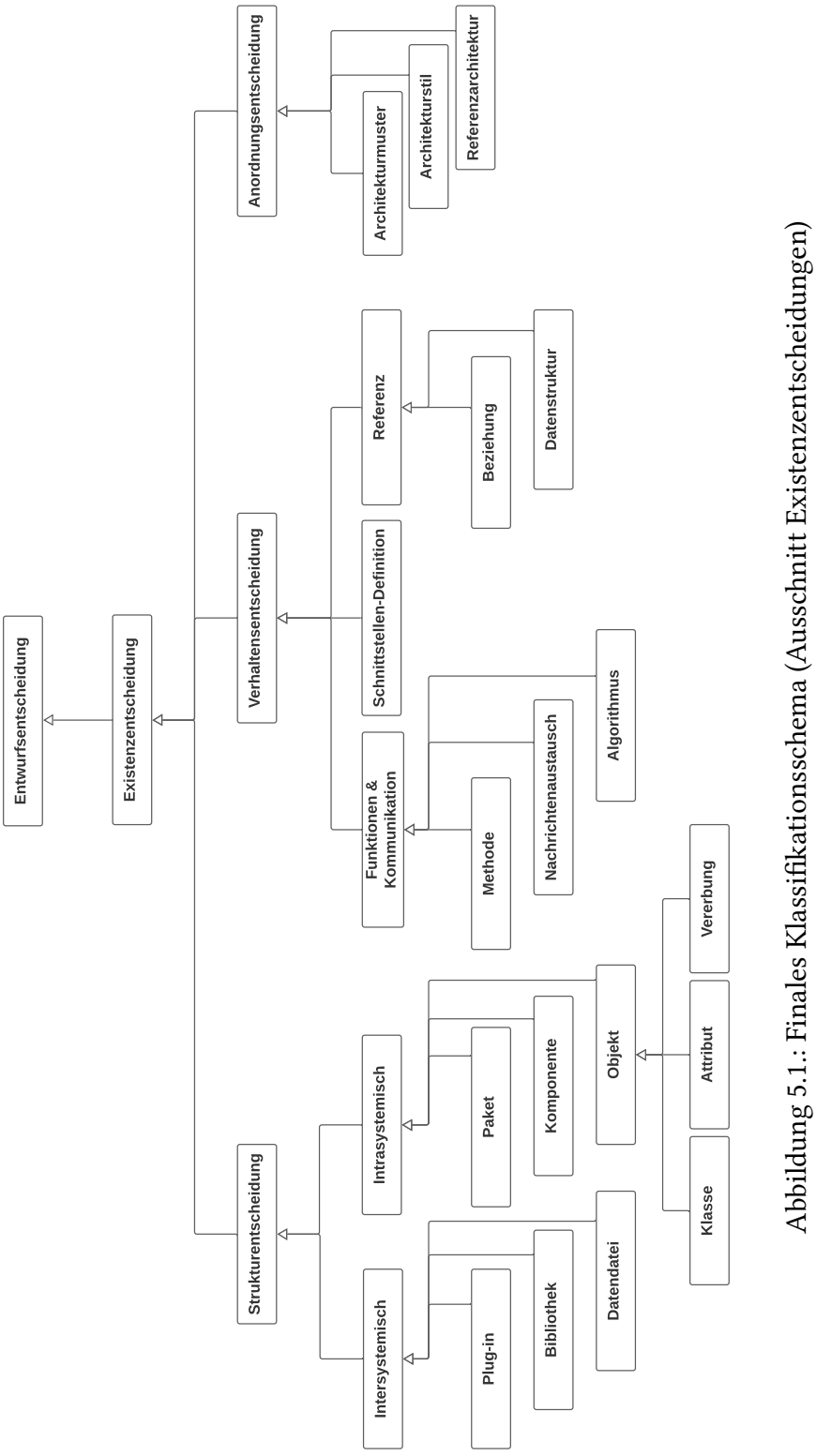

Abbildung 5.1.: Finales Klassifikationsschema (Ausschnitt Existenzentscheidungen)

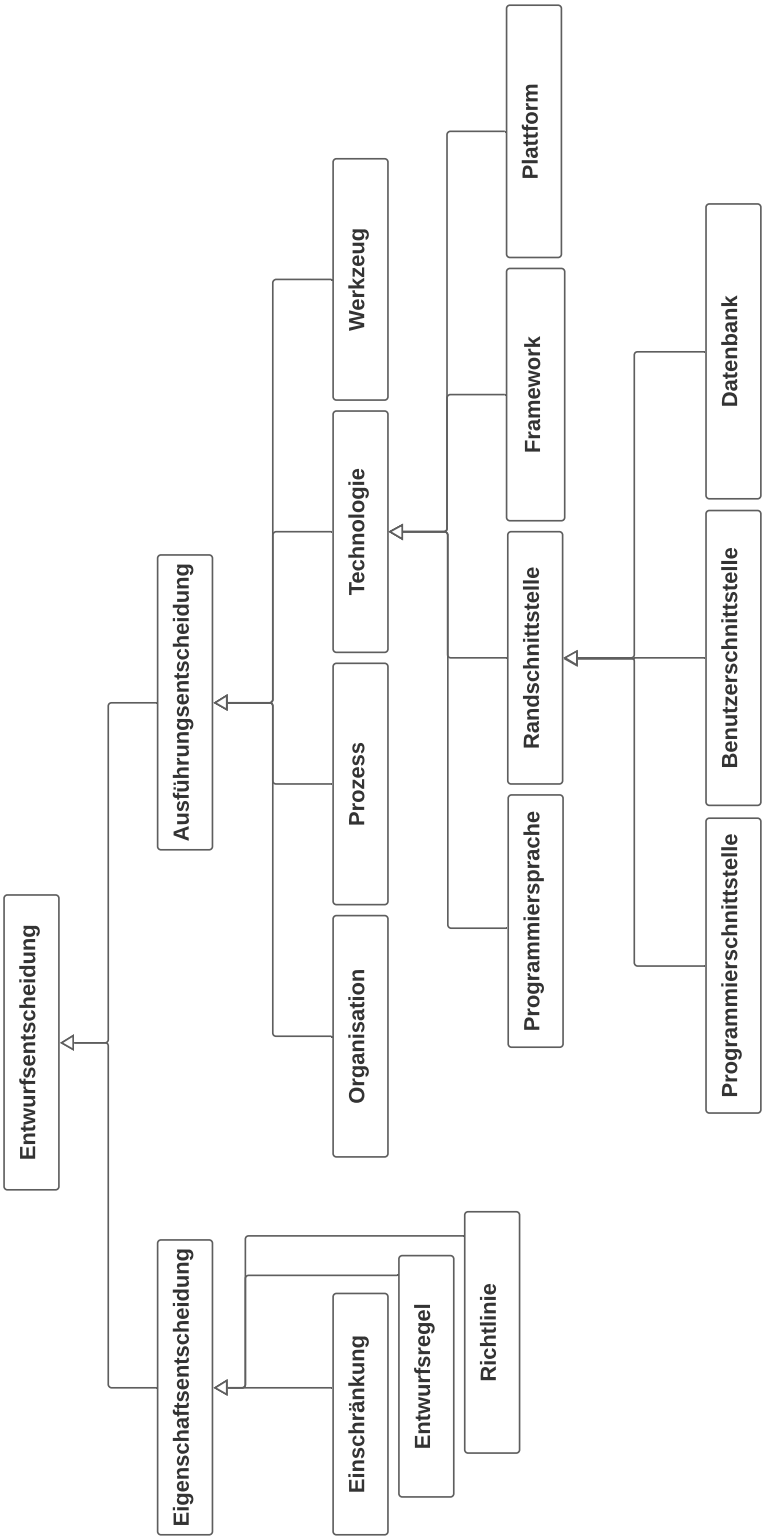

Abbildung 5.2.: Finales Klassifikationsschema (Ausschnitt Eigenschafts- und Ausführungsentscheidungen)

5.2. Klassenhäufigkeit in den Fallstudien

Wertet man die Klassifikation in den 17 Fallstudien anhand der zwei Label für die erste und zweite klassifizierte Entwurfsentscheidung pro Satz aus, konnten insgesamt 1427 Entwurfsentscheidungen identifiziert und klassifiziert werden (vgl. Tabelle 5.1). Dabei entstammen 492 Entwurfsentscheidungen Sätzen mit einem Multi-Label (246 Sätze) und 935 Entwurfsentscheidungen und damit etwa 2/3 lagen einzeln vor. Die umfangreichsten analysierten Architekturdokumentationen sind die zu *Teammates* (253 Entwurfsentscheidungen) und *CoronaWarnApp* (277 Entwurfsentscheidungen) (vgl. Tabelle 5.2).

Insgesamt konnten 1003 Existenzentscheidungen (~70%), 82 Eigenschaftsentscheidungen (~6%) und 342 Ausführungsentscheidungen (~24%) in den analysierten Fallstudien ausgemacht werden. Diese Ergebnisse decken sich mit der in [33] durchgeführten Expertenumfrage, bei der 73% der von den Experten genannten Kategorien Existenzentscheidungen zugeordnet werden konnten, 23% Ausführungsentscheidungen und 4% Eigenschaftsentscheidungen. Für gegebene Beispiele von Entwurfsentscheidungen wurden in der Umfrage 65% als Existenzentscheidung, 27% als Ausführungsentscheidung und 8% als Eigenschaftsentscheidung klassifiziert [33, S. 7].

Die Existenzentscheidungen sind mit 428 Strukturentscheidungen (~43% der Existenzentscheidungen) und 575 Verhaltensentscheidungen (~57% der Existenzentscheidungen) ähnlich verteilt. Dabei betreffen knapp die Hälfte (~47%) der Strukturentscheidungen Entscheidungen zu Objekten (203 Entwurfsentscheidungen, davon 101 zu Klasse) und weitere 26% betreffen Komponenten (110). Die Verhaltensentscheidungen konnten in der Mehrheit der Fälle genauer zu *Funktionen & Kommunikation* (~68%, absolut: 389) und dabei vor allem als *Methode* (284) klassifiziert werden. Kaum aufgetreten sind insbesondere Entscheidungen zu Plug-ins (2 Entwurfsentscheidungen) und Referenzarchitekturen (5). Die anderen Klassen weisen absolute Häufigkeiten zwischen 22 und 74 Entwurfsentscheidungen für die jeweils speziellste Klasse auf.

Die gefundenen Ausführungsentscheidungen betreffen ganz besonders *Technologie*. Mit 325 Technologie-Entscheidungen machen diese 95% der Ausführungsentscheidungen in den Fallstudien aus. Dabei zeigten alle Unterklassen der Technologie-Entscheidungen relevante Häufigkeiten. Ausführungsentscheidungen zur Organisation (4), zum Prozess (6) oder zu Werkzeugen (7) konnten kaum beobachtet werden. Bereits in der Expertenbefragung wurden keine Entwurfsentscheidungen, die sich dem Prozess oder den verwendeten Werkzeuge zuordnen lassen, genannt [33, S. 7f]. Weitere Untersuchungen ergaben, dass Entwurfsentscheidungen zum Prozess und zu Werkzeugen zwar getroffen werden, aber kaum dokumentiert werden [33, S. 8]. Für Entwurfsentscheidungen zur Organisation ist daher auch anzunehmen, dass diese insbesondere nicht innerhalb der Softwarearchitekturdokumentation dokumentiert werden. Dagegen gaben die befragten Experten an, dass Strukturentscheidungen und Technologie-Entscheidungen

immer (zumindest teilweise) dokumentiert werden, Verhaltensentscheidungen in der Mehrheit der Fälle [33, S. 8].

Eigenschaftsentscheidungen konnten in den Fallstudien nur in 5% der Fälle klassifiziert werden, wobei davon bereits über die Hälfte aus der Dokumentation der *CoronaWarnApp* entstammen. Dabei sind etwas mehr (49 von 82) positiv formulierte Entwurfsregeln oder Richtlinien im Vergleich zu negativ formulierten Einschränkungen (33). Auch dieses verminderte Auftreten in den Fallstudien lässt sich unter anderem mit der gängigen Handhabung von Softwareentwicklern erklären, Eigenschaftsentscheidungen seltener zu dokumentieren [33, S. 8].

Betrachtet man die Fallstudien einzeln, so können die Anteile von Existenzentscheidungen, Eigenschaftsentscheidungen und Ausführungsentscheidungen jedoch deutlich vom Mittel abweichen. So sind beispielsweise in der Dokumentation zu *ROD* Existenzentscheidungen mit 86% noch stärker vorherrschend, während dort Ausführungsentscheidungen nur noch einen Anteil von 10% ausmachen. Dagegen wurden in der Dokumentation zu *OnionRouting* 56% aller identifizierten Entwurfsentscheidungen einer Unterklasse der Ausführungsentscheidungen zugeordnet, 27% zu Existenzentscheidungen und 3% zu Eigenschaftsentscheidungen. Diese Schwankungen sind damit zu erklären, dass die Art der dokumentierten Entscheidungen abhängig ist von der Art des Softwareprojektes und der Handhabung durch die beteiligten Softwareentwickler, welche Entwurfsentscheidungen auch dokumentiert werden. Außerdem verstärkt sich dieser Effekt noch durch die Vorgehensweise bei der hier angewandten Klassifikation. Dadurch, dass die Sätze einzeln klassifiziert werden, betreffen aufeinanderfolgende Sätze häufig die gleiche Entwurfsentscheidung oder bereits erwähnte Entwurfsentscheidungen werden im Verlauf der Dokumentation noch einmal aufgegriffen und genauer erläutert. Dabei werden die Sätze wiederholt der gleichen Klasse zugeordnet.

Die auf den folgenden Seiten dargestellten Tabellen (siehe Tabelle 5.1 und Tabelle 5.2) zeigen die absoluten Häufigkeiten der jeweiligen Klasse von Entwurfsentscheidungen unter Anwendung des finalen Klassifikationsschemas auf die Fallstudien. In Tabelle 5.1 finden sich auch die Gesamtwerte als Summe aller Einzelwerte der Fallstudien. Dabei sind die Oberklassen Existenz- (blau), Eigenschafts- (orange) und Ausführungsentscheidungen (grün) farblich voneinander abgesetzt. Für die Darstellung der hierarchischen Tiefe verblasst die Farbgebung mit jeder Unterklasse. Die jeweilige Oberklasse befindet sich in der Tabelle oberhalb davon mit einer kräftigeren Färbung. Daher handelt es sich bei den angegebenen absoluten Klassenhäufigkeiten für Klassen, die über weitere Unterklassen verfügen, um die Summe aller Entwurfsentscheidungen, die zu einer der Unterklassen oder zur Klasse selbst zugeordnet wurden. Die Zuordnung nur zur Klasse selbst und keiner der Unterklassen erfolgte nur dann, falls keine Unterklassen existieren oder keine Unterklassen passend waren.

Tabelle 5.1.: Absolute Häufigkeit der Klassen von Entwurfsentscheidungen in den Fallstudien (Teil 1)

	Gesamt	ZenGarden	SpringXD	BIBINT	ROD	tagm8vault	MunkeyIssues	OnionRouting	Calipso
Textzeilen:	2080	134	113	27	135	16	24	55	33
Ident. Entwurfsentscheidungen:	1427	89	88	21	136	18	26	48	23
Zeilen mit Multi-Label:	246	13	14	0	31	4	8	8	9
Existenzentscheidung:	**1003**	**74**	**63**	**18**	**116**	**12**	**17**	**15**	**12**
Strukturentscheidung:	428	27	19	6	53	7	2	3	4
Intersystemisch:	72	3	2	1	12	0	1	3	1
Plug-in:	2	1	0	0	0	0	0	0	0
Bibliothek:	47	2	1	0	10	0	1	3	1
Datendatei:	23	0	1	1	2	0	0	0	0
Intrasystemisch:	356	24	17	5	41	7	1	0	3
Paket:	43	1	2	0	0	0	0	0	1
Komponente:	110	0	12	5	2	7	1	0	2
Objekt:	203	23	3	0	39	0	0	0	0
Klasse:	101	15	3	0	20	0	0	0	0
Attribut:	74	0	0	0	16	0	0	0	0
Vererbung:	22	8	0	0	3	0	0	0	0
Verhaltensentscheidung:	575	47	44	12	63	5	15	12	8
Funktionen & Kommunikation:	389	26	28	4	34	3	6	6	2
Methode:	284	10	16	2	20	3	0	5	2
Nachrichtenaustausch:	40	9	10	1	0	0	6	0	0
Algorithmus:	54	7	1	1	13	0	0	0	0
Referenz:	72	18	4	2	26	1	0	1	0
Beziehung:	34	8	0	0	16	1	0	0	0
Datenstruktur:	38	10	4	2	10	0	0	1	0
Anordnung:	64	0	7	2	1	0	9	4	5
Architekturstil:	36	0	1	2	0	0	7	4	4
Architekturmuster:	23	0	3	0	1	0	2	0	1
Refernzarchitektur:	5	0	3	0	0	0	0	0	0
Schnittstellen-Definition:	50	3	5	4	2	1	0	1	1
Eigenschaftsentscheidung:	**82**	**3**	**0**	**0**	**6**	**0**	**0**	**2**	**2**
Einschränkung:	33	0	0	0	2	0	0	0	0
Entwurfsregel:	39	1	0	0	3	0	0	2	1
Richtlinie:	10	2	0	0	1	0	0	0	1
Ausführungsentscheidung:	**342**	**12**	**25**	**3**	**14**	**6**	**9**	**31**	**9**
Organisation:	4	1	1	0	1	0	0	0	0
Prozess:	6	0	2	1	1	0	0	0	0
Werkzeug:	7	1	0	0	0	0	0	1	0
Technologie:	325	10	22	2	12	6	9	30	9
Programmiersprache:	78	4	6	0	3	3	0	8	1
Plattform:	85	5	14	0	1	0	1	14	6
Framework:	45	0	0	0	0	0	3	5	1
Randschnittstelle:	156	1	2	2	8	3	8	8	2
Programmierschnittstelle:	35	0	2	0	0	0	3	2	1
Benutzeroberfläche:	32	1	0	2	0	1	2	1	0
Datenbank:	42	0	0	0	8	2	0	0	0

Tabelle 5.2.: Absolute Häufigkeit der Klassen von Entwurfsentscheidungen in den Fallstudien (Teil 2)

	IOSched	MyTardis	SCons	OpenRefine	Beets	Teammates	QMiner	Spacewalk	CoronaWarnApp
Textzeilen:	109	159	92	26	170	342	120	45	480
Ident. Entwurfsentscheidungen:	52	65	68	24	116	253	88	35	277
Zeilen mit Multi-Label:	10	10	8	6	22	43	12	7	41
Existenzentscheidung:	**24**	**29**	**57**	**12**	**108**	**200**	**78**	**12**	**156**
Strukturentscheidung:	14	18	25	7	51	123	16	7	46
Intersystemisch:	12	1	5	4	8	9	0	2	8
Plug-in:	0	0	0	0	1	0	0	0	0
Bibliothek:	12	0	4	4	5	0	0	2	2
Datendatei:	0	1	1	0	2	9	0	0	6
Intrasystemisch:	2	17	20	3	43	114	16	5	38
Paket:	1	0	3	3	0	31	0	1	0
Komponente:	1	13	0	0	10	45	6	4	2
Objekt:	0	4	17	0	33	38	10	0	36
Klasse:	0	2	11	0	16	30	4	0	0
Attribut:	0	2	2	0	12	4	2	0	36
Vererbung:	0	0	4	0	4	2	1	0	0
Verhaltensentscheidung:	10	11	32	5	57	77	62	5	110
Funktionen & Kommunikation:	8	6	17	2	41	56	50	1	99
Methode:	7	6	12	1	38	51	28	1	82
Nachrichtenaustausch:	1	0	0	0	0	0	0	0	13
Algorithmus:	0	0	4	1	3	2	20	0	2
Referenz:	0	0	6	0	9	0	1	0	4
Beziehung:	0	0	3	0	6	0	0	0	0
Datenstruktur:	0	0	3	0	3	0	1	0	4
Anordnung:	2	4	0	3	6	7	8	4	2
Architekturstil:	2	3	0	3	0	1	5	2	2
Architekturmuster:	0	0	0	0	6	6	3	1	0
Refernzarchitektur:	0	1	0	0	0	0	0	1	0
Schnittstellen-Definition:	0	1	9	0	1	14	3	0	5
Eigenschaftsentscheidung:	**2**	**3**	**2**	**2**	**0**	**13**	**0**	**0**	**47**
Einschränkung:	0	0	0	0	0	4	0	0	27
Entwurfsregel:	1	0	2	1	0	9	0	0	19
Richtlinie:	1	3	0	1	0	0	0	0	1
Ausführungsentscheidung:	**26**	**33**	**9**	**10**	**8**	**40**	**10**	**23**	**74**
Organisation:	1	0	0	0	0	0	0	0	0
Prozess:	0	0	0	0	0	0	0	0	2
Werkzeug:	4	0	0	0	0	0	0	0	1
Technologie:	21	33	9	10	8	40	10	23	71
Programmiersprache:	6	6	7	4	1	13	3	9	4
Plattform:	4	9	0	3	0	1	1	4	22
Framework:	1	9	0	1	0	7	0	0	18
Randschnittstelle:	11	18	2	3	7	26	6	10	39
Programmierschnittstelle:	0	3	0	1	0	6	3	4	10
Benutzeroberfläche:	1	2	2	0	5	7	0	4	4
Datenbank:	9	4	0	1	2	5	3	2	6

5.3. Validität des Klassifikationsschemas und dessen Anwendung in der manuellen Analyse

Dieser Abschnitt wirft einen bewertenden Blick auf die iterative Entwicklung des Klassifikationsschemas und benennt Argumente für die Validität des finalen Klassifikationsschemas, aber auch Gefahren hierfür. Dies betrifft die Konzeption des Entwicklungsprozesses, dessen Durchführung und den Umgang mit realen Fallbeispielen in der manuellen Analyse. Für die Evaluation eines Klassifikationsschemas sind nach Bedford [4] die Prinzipien Konsistenz (*consistency*), Verwandtschaft (*affinity*), Aufgliederung (*differentiation*), Vollständigkeit (*exhaustiveness*), Ausschließlichkeit (*exclusiveness*), Erfassbarkeit (*ascertainability*) und Geltungsbereich (*currency*) ausschlaggebend. Die Anwendung des Klassifikationsschemas soll mit möglichst wenig Unsicherheiten behaftet sein [4, S. 4]. Anhand dieser Kriterien wird das entwickelte Klassifikationsschema im Folgenden bewertet.

5.3.1. Evaluation der Struktur des Klassifikationsschemas

Die *Konsistenz* zu bereits bestehenden Klassifikationsschemata für architektonische Entwurfsentscheidungen entsteht durch die Entwicklung entlang verwandter Arbeiten. Grundsätzlich stellt das Klassifikationsschema eine Weiterentwicklung und Verfeinerung der Ontologie für Entwurfsentscheidungen nach Kruchten [27] dar und ist in weiten Teilen konsistent zu dieser. Neben Jansen und Boschs Modell für architektonische Entwurfsentscheidungen [22] ist die Ontologie nach Kruchten eine der meistzitierten Grundlagenarbeiten für die Klassifikation von Entwurfsentscheidungen. Auch in anderen Arbeiten, die sich mit der manuellen und automatisierten Klassifikation von Entwurfsentscheidungen beschäftigen, stellt die Ontologie häufig die Grundlage dar [6][29][33]. Das in dieser Arbeit entwickelte Klassifikationsschema bricht strukturell nur einmal mit der Ontologie nach Kruchten. Kruchtens Ontologie untergliedert Existenzentscheidungen in Struktur-, Verhaltens- und Ausschlussentscheidungen [27]. Aus der Überlegung heraus, dass sich der überwiegende Teil der Struktur- und Verhaltensentscheidungen auch als Ausschluss formulieren lässt, müssten sich deren Unterklassen auch symmetrisch als Unterklassen der Ausschlussentscheidungen wiederfinden lassen und dadurch doppeln. Vielmehr ist die inkludierende oder eben exkludierende Formulierung eine Eigenschaft jeder Entwurfsentscheidung (vgl. Abbildung 3.2). Deshalb weist das hier entwickelte Klassifikationsschema nur noch die Struktur- und Verhaltensentscheidungen sowie die neu eingeführten Anordnungsentscheidungen als Unterklassen der Existenzentscheidungen auf. Es ließen sich in den Fallstudien auch Ausschlussentscheidungen hinsichtlich der Verwendung bestimmter Technologien finden („Architecture: [...] we decided NOT to use hazelcast“, aus Dokumentation zu *OnionRouting*), die als Unterklasse der Ausführungsentscheidungen ursprünglich

gar nicht über diese Klassifikationsrichtung verfügten. Abgesehen von dieser strukturellen Abweichung wurde das Klassifikationsschema nur in der hierarchischen Tiefe verfeinert. Dabei ist hervorzuheben, dass alle neu eingeführten Unterklassen auch in den Fallstudien vorgekommen sind und erst dadurch ihre Berechtigung finden. Auch die Regeln für die manuelle Identifikation von Entwurfsentscheidungen aus Bhat et al. [6, S. 8] wurden vollständig in Klassen und ihre Beschreibung überführt.

Für die *Verwandtschaft* innerhalb eines Klassifikationsschemas soll gelten, dass jede Klasse des Schemas in Anbetracht ihrer Oberklasse definiert wurde [4, S. 2]. Die iterative Weiterentwicklung verlief stets „top-down", da die Oberklassen bereits durch das initiale Klassifikationsschema bestanden und daraufhin feingranularer aufgespalten wurden. Da erst am Ende jeder Iteration Veränderungen am Klassifikationsschema vorgenommen wurden, musste die identifizierte Entwurfsentscheidung zuerst einer bestehenden Klasse zugeordnet werden und daraufhin konnte diese Klasse spezialisiert werden. Diese provisorische Zuordnung garantiert eine Verwandtschaft zwischen Ober- und Unterklasse.

Auch das Grundprinzip zur *Aufgliederung*, wonach eine Kategorie, wenn sie aufgeteilt wird, stets über mindestens zwei Unterkategorien verfügen muss [4, S. 2], wird für das hier entwickelte Klassifikationsschema eingehalten (vgl. Abbildung 5.1, Abbildung 5.2).

5.3.2. Evaluation der Klassifikation hinsichtlich Vollständigkeit

Hinsichtlich der *Vollständigkeit*, die das Schema für eine Klassifikation aller identifizierten Entwurfsentscheidungen in Softwaredokumentationen bieten soll, kann grundsätzlich nicht ausgeschlossen werden, dass in anderen Fallstudien Entwurfsentscheidungen auftreten, die nicht in das hier entwickelte Schema passen. Zumindest die Oberklassen des Schemas sind dabei jedoch entsprechend der Ontologie nach Kruchten [27], deren Anwendung in verwandten Arbeiten bisher noch nicht dadurch aufgefallen ist, bestimmte Entwurfsentscheidungen nicht klassifizieren zu können [6][33]. Außerdem entspricht der iterative Entwicklungsprozess dieser Arbeit dem empirischen Ansatz mit induktiver Methodik der amerikanischen National Information Standards Organization [2, S. 91ff]. Dabei werden neue Bezeichner, in diesem Fall neue Klassen, zunächst für eine potenzielle Aufnahme in das Schema gelistet, sobald sie sich in einem Inhaltsobjekt, hier in einer Fallstudie, hervorgetan haben [2, S. 92]. Die Aufnahme in die hierarchische Struktur erfolgt dann als Unterkategorie einer weiter gefassten Oberkategorie [2, S. 92]. In den letzten beiden Iterationen der Fallstudienanalyse (vgl. Unterabschnitt 4.4.5 und Unterabschnitt 4.4.6) wurden keine Fallbeispiele gefunden, zu deren Einordnung es potenziell einer Änderung am Klassifikationsschema bedarf. Dies spricht für die Ausgereiftheit und Stabilität des Schemas auch hinsichtlich neu hinzugefügter Klassen, da nachhaltig keine Änderungen erforderlich waren. Die Anwendung des Schemas hat dabei sowohl den ausführlichen Dokumentationen zu *Teammates* und *CoronaWarnApp* als auch kürzeren Dokumentationen wie *tagm8vault* standgehalten (siehe Tabelle 4.1).

Diese Softwareprojekte entstammen zudem verschiedenen Gebieten, unter anderem mobile Applikationen, verteilte Systeme und Web-Anwendungen. Bei den analysierten Softwareprojekten handelte es sich ausschließlich um Open-Source-Projekte, häufig auf GitHub. Für unternehmensinterne Architekturdokumentationen, die für diese Arbeit nicht zur Verfügung standen, lässt sich nicht ausschließen, dass diese sich von denen in Open-Source-Projekten unterscheiden und ihre Analyse zu einer weiteren Verfeinerung des Klassifikationsschemas führen könnte.

Das hier entwickelte und angewandte Klassifikationsschema klassifiziert Entwurfsentscheidungen eindimensional nach ihrer Art. Darüber hinaus können Entwurfsentscheidungen auch noch in weiteren Dimensionen analysiert werden, je nachdem welche Fragestellung eine entsprechende Arbeit verfolgt. Diese Arbeit beschränkt sich auf die Klassifikation von Entwurfsentscheidungen hinsichtlich ihrer Art, da hierdurch die entsprechenden Rückschlüsse für den Vergleich von Architekturdokumentation und der Umsetzung im Softwaresystem gezogen werden können, wie dies in [26] angestrebt wird. Verschiedenartige Entwurfsentscheidungen unterscheiden sich in der Konsequenz, die sie auf die Softwarearchitektur haben. Eine weitere mögliche Klassifikationsrichtung bilden die in [33, S. 141] vorgestellten Ebenen *Implementierung*, *Architektur*, *Projekt* und *Organisation*. In dieser Arbeit wurde sich auf Softwarearchitekturdokumentation fokussiert. Darin finden sich hauptsächlich Entwurfsentscheidungen auf der Ebene *Architektur*, die mitunter auch direkte Auswirkungen auf die *Implementierung* haben. Entscheidungen auf den Ebenen *Projekt* und *Organisation* sind wohl Ausführungsentscheidungen im Sinne der Ontologie nach Kruchten [27], welche jedoch selten aufgetreten sind (vgl. Abschnitt 5.2).

Eine weitere mögliche Klassifikationsrichtung besteht entlang der Ebenen *Entscheidung*, *Alternative* und *Argumentation*, wie die Analyse der Fallstudien gezeigt hat. Neben der primären Entscheidung, beispielsweise für die Existenz von Systemkomponenten oder der Verwendung einer Technologie, werden auch Alternativen betrachtet, die letztendlich nicht umgesetzt werden. Dies bedarf einer Argumentation für die Entscheidung und gegen die Alternative. Allerdings liegen in der Softwaredokumentation in der ganz großen Mehrheit nur die primären Entscheidungen vor, während die betrachteten Alternativen und die Argumentation nicht dokumentiert wurde. Dies erschwert eine Klassifikation entlang dieser Ebenen.

5.3.3. Evaluation der Anwendbarkeit und Handhabung des Schemas

Die *Ausschließlichkeit* eines Schemas besagt, dass keine zwei Klassen sich überlappen oder den gleichen Anwendungsbereich haben [4, S. 2]. Grundsätzlich wurde bei der Entwicklung des Klassifikationsschemas stets darauf geachtet, dass die Klassifikation eindeutig vorgenommen werden kann. Um dies zu gewährleisten, wurde beispielsweise ein Ausschluss als Eigenschaft jeder Entwurfsentscheidung definiert oder die Klasse *Drittanbieter-System* entfernt, nachdem die Klasse *Randschnittstelle* mit einem ähnlichen

Geltungsbereich geschaffen wurde. Durch die Vielfalt natürlicher Sprache fällt die eindeutige Zuordnung zu einer Klasse des Schemas und die klare Abgrenzung der Klassen untereinander mitunter dennoch schwer, da mehrere Entwurfsentscheidungen sprachlich miteinander vermischt werden, miteinander in Beziehung stehen oder sich gegenseitig bedingen [27, S. 4f]. So bestehen zwangsläufig Überschneidungen zwischen den Klassen des Schemas. Auch das Auftreten von mehreren Entwurfsentscheidungen in einem Satz der Softwarearchitekturdokumentation stellt keine Seltenheit dar, sodass bei zukünftigem Einsatz stets sorgfältig überprüft werden sollte, ob sich noch weitere Entwurfsentscheidungen in einem Satz identifizieren lassen.

Die *Erfassbarkeit* eines Klassifikationsschemas ist gegeben, falls jede Klasse durch den Klassennamen direkt verstanden werden kann [4, S. 2]. Die neu gewählten Klassennamen spiegeln feststehende Begriffe aus der Domäne der objektorientierten Softwareentwicklung wider. Um Missverständnissen dennoch vorzubeugen und Mehrdeutigkeiten aufzulösen, sind alle Klassen in Abschnitt 5.1 in einer Übersicht beschrieben, sodass die Zuordnung von Entwurfsentscheidungen zu den Klassen konsistent vorgenommen werden kann. Das Klassifikationsschema wurde bereits wiederholt auf die Fallstudien angewandt und dabei auf mögliche Mehrdeutigkeiten geprüft. Außerdem wurde die Klassifikation für identifizierte Entwurfsentscheidungen aus vorangegangenen Iterationen noch einmal mit dem finalen Klassifikationsschema überprüft.

Zu beachten ist abschließend noch bezüglich des *Geltungsbereiches* des Klassifikationsschemas, dass das Klassifikationsschema für den objektorientierten Softwareentwurf entwickelt wurde. Es ist zwar bedingt möglich, nicht-objektorientierte Konzepte, wie etwa Strukturen in C oder relationale Datenbankschemata, analog in das Klassifikationsschema zu überführen, jedoch bedarf dies Interpretation und Argumentation. So kann etwa eine Struktur in C als Klasse interpretiert werden, Tabelleneinträge sind Objekte und Spaltennamen einer Datenbanktabelle sind ähnlich zu Attributen. Insgesamt ist die Struktur des Klassifikationsschemas jedoch nicht daran angepasst.

5.3.4. Vermeidung von Unsicherheiten bei der Anwendung des Klassifikationsschemas

Aufgrund der Umstände konnten die Softwarearchitekturdokumentationen nur durch eine Person gesichtet und mit Labeln versehen werden. Dadurch sind die Klassifikationsentscheidungen mit einer persönlichen Interpretation behaftet und werden gegebenenfalls von einem anderen, erfahreneren Softwareentwickler in Einzelfällen anders klassifiziert. Hinsichtlich der Klassifikationsentscheidungen können Unsicherheiten über das zu klassifizierende Objekt, die zugehörige Klasse sowie das Schema als Ganzes auftreten [4, S. 4]. Um die Unsicherheit über das Objekt (den zu klassifizierenden Satz) zu reduzieren, ist es erforderlich, die Absicht, die der Softwareentwickler mit dieser Entwurfsentscheidung und deren Dokumentation verfolgt, zu ergründen. Dies

ließ sich mitunter auch nur durch den Kontext erschließen, sodass für die manuelle Klassifikation stets eine Klassifikation der gesamten Architekturdokumentation durchgeführt werden sollte und auch ein kurzes Einlesen in das Softwareprojekt, etwa in Form eines Readme, sich als hilfreich herausgestellt hat. Um Unsicherheiten über die Klassen zu reduzieren, wurden diese definiert und ausgearbeitet, sodass neben dem Klassennamen auch die Klassenbeschreibung in Abschnitt 5.1 zur Verfügung steht. Für das Verständnis des gesamten Schemas in seinem Aufbau und seiner Struktur bietet der Entstehungsprozess in Kapitel 3 und Kapitel 4 Aufschluss.

6. Maschinelle Klassifikation der Entwurfsentscheidungen

Ziel der in diesem Kapitel beschriebenen Entwicklung ist es, die Analyse von Entwurfsentscheidungen in natürlichsprachiger Softwarearchitekturdokumentation automatisiert vornehmen zu können. Dadurch können Entwurfsentscheidungen aus bestehender Dokumentation im Projektverlauf rekonstruiert werden und Inkonsistenzen zwischen der Dokumentation und dem Quellcode aufgedeckt und vermieden werden. Dabei stellt diese Arbeit eine Proof-of-Concept-Implementierung dar, um eine Klassifikation von Entwurfsentscheidungen anhand des hier entwickelten Klassifikationsschemas vorzunehmen. Durch die hierarchische Gliederung des Klassifikationsschemas kann diese Klassifikation auf unterschiedlichen Ebenen des Schemas vorgenommen werden.

Die Implementierung beruht auf verschiedenen Ansätzen des maschinellen Lernens und der natürlichen Sprachverarbeitung (*Natural Language Processing, NLP*). Hierfür wurden die statistischen Modelle jeweils mit einem Textkorpus trainiert, wobei die Sätze den untersuchten Fallstudien (siehe Tabelle 4.1) entstammen und in diesem Zusammenhang auch mit Labeln versehen wurden. Die Klassifikatoren versuchen daraufhin, die Sätze aus der Softwarearchitekturdokumentation den zugehörigen Klassen des Klassifikationsschemas (vgl. Abschnitt 5.1) zuzuweisen. Um die Klassifikation anhand der gewählten statistischen Modelle sowie deren Genauigkeit in verschiedenen Tiefen des Klassifikationsschemas vergleichen zu können, wurden jeweils mittels Kreuzvalidierungsverfahren (*k-fold cross-validation*) statistische Evaluationsgrößen berechnet.

Das Kapitel stellt zunächst in Abschnitt 6.1 die zugrundeliegenden Überlegungen vor, mittels maschinellem Lernen die automatisierte Analyse von Entwurfsentscheidungen vornehmen zu können. In Abschnitt 6.2 werden verschiedene Ansätze und Modelle aus dem Bereich des maschinellen Lernens vorgestellt, mit denen die Klassifikation vorgenommen werden kann. Diese werden daraufhin in der in Abschnitt 6.3 beschriebenen Implementierung verwendet, um Entwurfsentscheidungen in natürlicher Sprache automatisiert zu identifizieren und zu klassifizieren. Abschnitt 6.4 vergleicht und evaluiert die verwendeten Ansätze und enthält einen Ausblick auf potenziell notwendige Erweiterungen der Implementierung und Analysen.

6.1. Automatisierte Analyse mit maschinellem Lernen

Um Sätze aus Softwarearchitekturdokumentationen anhand ihrer enthaltenen Entwurfsentscheidungen zu klassifizieren, können regelbasierte oder auf maschinellem Lernen beruhende Ansätze verwendet werden. In dieser Arbeit wurde die automatisierte Analyse schlussendlich mit Ansätzen des maschinellen Lernens und NLP implementiert und sich damit gegen einen regelbasierten Ansatz entschieden. Einen solchen regelbasierten Ansatz stellt etwa die Implementierung eines syntaktischen Erkenners für Schlüsselwörter dar. Für entsprechende Klassen aus dem Klassifikationsschema werden dafür Schlüsselwörter festgelegt, die bei entsprechendem Vorkommen in einem Satz dessen Zuordnung zu dieser Klasse nahelegen. Nachteilig an diesem Vorgehen ist allerdings, dass die entsprechenden Schlüsselwörter zunächst für jede Klasse festgelegt werden müssen und bei der Verwendung weiterer Klassen stets erweitert werden müssen. Die Analyse der Fallstudien vermittelt den Eindruck, dass dies für die Klasse *Programmiersprache* durch Eingabe der gängigen Programmiersprachen noch in einem überschaubaren Bereich möglich ist, während beispielsweise für *Plattform* bereits zahlreiche Technologien bestehen, deren vollständige Einpflegung in das System selbst mit Domänenwissen eine Herausforderung darstellt. Für Entwurfsentscheidungen zu *Funktionen und Kommunikation* ist sogar die Extraktion von Schlüsselwörtern nahezu unmöglich, da sich der Ausdruck über die gewünschte Funktionalität eines Softwaresystems in die Semantik des Satzes erstreckt und insbesondere nicht mehr auf einzelne Substantive reduziert werden kann.

Mithilfe von maschinellem Lernen können diese wiederkehrenden Wörter und Muster anhand von Trainingsdaten gelernt werden, ohne sie hart codiert festzulegen. Dafür müssen die Sprachbausteine zunächst in eine maschinenverständliche Repräsentation in Form von Vektoren gebracht werden. In diesem Schritt können die Zählung der Wortvorkommen und das sogenannte Tf-idf-Maß (*term frequency* und *inverse document frequency*) eingesetzt werden (vgl. Unterabschnitt 6.2.1). Diese übernehmen die eben für den regelbasierten Ansatz genannte Idee, dass bestimmte Schlüsselwörter, die charakteristisch für eine Klasse sind, Einfluss auf das Lernverfahren nehmen sollten [58].

Es handelt sich bei der hier vorgenommenen Klassifikation um eine Variante des überwachten Lernens. Als Grundlage dient der aus der manuellen Analyse der Fallstudien entstandene Korpus, in dem jeder Satz aus der jeweiligen Softwarearchitekturdokumentation mit bis zu zwei Labeln für die passende Klasse des Schemas versehen wurde. Anhand dieser Trainingsdaten wird daraufhin ein Modell trainiert, das anschließend möglichst in der Lage ist, für ungesehene Sätze die passende Klasse entsprechend der enthaltenen Entwurfsentscheidungen zuzuordnen.

6.2. Ansätze für die Klassifikation von Sätzen aus natürlicher Sprache

Die in dieser Arbeit vorgenommene Implementierung dient der prototypischen Anwendung verschiedener NLP-Ansätze, um anhand ausgewählter Klassen des Schemas zu bewerten, welche Ansätze erfolgversprechend sind. Für den Einsatz eines Klassifikators wurde dafür zunächst eine Vorverarbeitung des Textkorpus vorgenommen und dieser anschließend mit Bag-of-Words (BoW), Tf-idf oder N-Gramm in eine Vektorrepräsentation überführt. Diese Repräsentationen konnten anschließend genutzt werden, um die Modelle anhand von Logistischer Regression, Multinomialem Naive Bayes, Entscheidungsbaumverfahren, Random-Forest-Klassifikation oder einer Support Vector Machine zu trainieren. Diese eher klassischen Ansätze wurden zudem mit dem transformer-basierten Sprachmodell Bidirectional Encoder Representations from Transformers (BERT) verglichen.

6.2.1. Vorverarbeitung und vektorielle Textrepräsentation

Für die Vorverarbeitung natürlicher Sprache bieten sich verschiedene Methoden an, die Auswirkungen auf die Leistungsfähigkeit des Klassifikators haben können. Zunächst kann eine Textaufbereitung vorgenommen werden, die Zahlen und Sonderzeichen aus dem Text entfernt. Außerdem bestehen Möglichkeiten der Vorverarbeitung eines Textes für NLP darin, ausschließliche Kleinschreibung zu verwenden (*Lowercasing*), Stoppwörter zu entfernen (*Stop Words Removal*) und Wörter auf den gemeinsamen Wortstamm (Stammformreduktion, *Stemming*) sowie eine einheitliche Grundform (Lemmatisierung, *Lemmatization*) zurückzuführen [8][17]. Li et al. stellten bei ihrer Klassifikation von Entwurfsentscheidungen in der Hibernate Developer Mailing List fest, dass Stammformreduktion und Lemmatisierung für ihren Fall gar einen negativen Effekt auf die Leistungsfähigkeit des Klassifikators hatte [29].

Sonderzeichen, Zahlen sowie Hyperlinks tauchen in den Softwarearchitekturdokumentation immer wieder auf. Diese können etwa bei der in *GitHub* häufig verwendeten Auszeichnungssprache *Markdown* zum Definieren von Überschriften mittels '#' verwendet werden und damit eine reine Formatierung sein, aber beispielsweise auch für Stichwörter wie 'C++' eine Bedeutung tragen. Zudem vergrößern Sonderzeichen, Zahlen und Text-Sonderzeichen-Kombinationen die Größe des Wortschatzes, wodurch vektorielle Repräsentationen mehr Dimensionen benötigen. Neben der Entfernung von Zahlen und Sonderzeichen reduziert auch die Umformung aller Wörter in Kleinschreibung die Größe des Wortschatzes und seltenes Vorkommen einzelner Wörter kann stärker vermieden werden [8, S. 2]. Da in der englischen Sprache Groß- und Kleinschreibung (anders als im Deutschen) hauptsächlich nur für Satzanfänge und Eigennamen verwendet wird, ist ihre Bedeutung für die Informationsgewinnung aus

Text möglicherweise vernachlässigbar. Eine ähnliche Überlegung steht hinter dem Entfernen sogenannter Stoppwörter: Diese Wörter kommen besonders häufig in Texten vor, sind jedoch grundsätzlich nicht spezifisch für den Inhalt des Textes, sondern dienen dazu, andere Satzbausteine zu verbinden [17, S. 3]. Ihr Entfernen kann dazu führen, die Klassifikation zu verbessern, indem nur noch sinntragende Wörter auf diese Einfluss nehmen. Allerdings weisen beispielsweise Artikel auf die Verwendung von Substantiven und Eigennamen hin, die wiederum charakteristisch für eine Klasse sein können. Stammformreduktion und Lemmatisierung beschreibt den Prozess, dass Wörter zunächst auf eine gemeinsame Stammform gebracht werden, indem ihre Deklination und Konjugation aufgelöst wird, und anschließend das gegebene Wort durch das zugehörige Lemma ersetzt wird [8, S. 2]. Dies wirkt spärlichem Vorkommen im Trainingsdatensatz entgegen, da die abgeleiteten Formen eines Lemmas weniger häufig vorkommen, kann jedoch auch dazu führen, dass syntaktische und semantische Feinheiten nach der Lemmatisierung übersehen werden [8, S. 2].

Um den Klassifikator zu trainieren, bedarf es einer vektoriellen Repräsentation der Eingaben, in diesem Fall der Sätze aus der Softwarearchitekturdokumentation. Hierfür bieten sich mehrere Ansätze zur Merkmalsextraktion an, darunter gängige Varianten wie BoW, Tf-idf und N-Gramm. Für BoW wird zunächst ein Wörterbuch generiert, welches sich aus allen einzigartigen Satzbausteinen innerhalb des Datensatzes zusammensetzt [29, S. 5]. Daraufhin wird der Merkmalsvektor jedes Satzes erzeugt, indem die Vorkommenshäufigkeit (*term frequency*, tf) aller Begriffe des Wörterbuchs berechnet wird [29, S. 5]. Verwendet man das Tf-idf-Maß, dann wird die Vorkommenshäufigkeit anschließend noch mit der Inversen Dokumenthäufigkeit (*inverse document frequency*, idf) multipliziert, um sicherzustellen, dass das häufige Vorkommen charakteristisch für genau diesen Satz ist [41]. Wörter, die nur in wenigen Sätzen vorkommen, aber häufig in einem bestimmten Satz, werden durch das Tf-idf-Maß besonders stark gewichtet. BoW und Tf-idf sind zunächst sogenannte Unigramme, das heißt, sie betrachten nicht den Kontext, in dem ein Wort auftritt, nicht die Reihenfolge der Wörter und behandeln konjugierte und deklinierte Wörter sowie Rechtschreibfehler als eigenes Wort [37]. Mithilfe von N-Grammen wird das Vorkommen von N aufeinander folgenden Worten gezählt und dadurch ein Kontextbezug hergestellt [37]. N-Gramme stellen insbesondere eine effiziente Methode für das Erlernen präziser Wortvektoren von großen Mengen an Textdokumenten dar [29, S. 5].

6.2.2. Verwendete Klassifikationsalgorithmen

Die vektorielle Darstellung der Wörter aus dem Korpus kann nun genutzt werden, anhand von verschiedenen Klassifikationsalgorithmen ein Vorhersagemodell für ungesehene Sätze aus einer Softwarearchitekturdokumentation zu trainieren. Die Verwendung verschiedener Klassifikationsalgorithmen führt zumeist zu einer unterschiedlichen Genauigkeit des Vorhersagemodells. Verglichen werden daher statistische Maße für

die Klassifikationsalgorithmen Logistische Regression (LR), Multinomialer Naive Bayes (MNB), Entscheidungsbaumverfahren (*Decision Tree*, DT), Random-Forest-Klassifikation (RF) und Support Vector Machine (SVM).

Mithilfe von Logistischer Regression (LR) [35] werden Beobachtungen einer diskreten Menge von Klassen zugewiesen. Dabei wird anhand der Trainingsdaten eine Vorhersagefunktion gelernt, die die logarithmische Kostenfunktion (Fehlerhäufigkeit) minimiert. Bei Eingabe eines ungesehenen Merkmalsvektors in die Vorhersagefunktion wird eine Wahrscheinlichkeit zwischen 0 und 1 zurückgegeben, mit der die Klassenzuweisung erfolgen kann. Anders als bei Linearer Regression verfügt die Logistische Regression über nicht-lineare Funktionen, häufig die Sigmoid-Funktion, sodass komplexere Entscheidungsgrenzen gelernt werden können.

Ein Naiver Bayes-Klassifikator [42, S. 37f] beruht auf dem Satz von Bayes, mit dem die A-posteriori-Wahrscheinlichkeit bestimmt werden kann, indem die A-priori-Wahrscheinlichkeiten mithilfe von Trainingsdaten geschätzt werden. Anhand der Merkmalsvektoren der Daten wird eine Wahrscheinlichkeitstabelle aufgestellt, in der die Wahrscheinlichkeit einer Klasse abgelegt ist, um darauf basierend neue Beobachtungen zu klassifizieren. „Naiv“ an der Klassifikation ist die Annahme, dass Datenpunkte voneinander stochastisch unabhängig sind [42, S.38]. Multinomialer Naiver Bayes ist dabei geeignet, um eine Textklassifikation anhand von diskreten Merkmalsvektoren wie etwa BoW und grundsätzlich auch nicht-ganzzahligen Merkmalsausprägungen wie bei Tf-idf vorzunehmen [9].

Ein Entscheidungsbaum [25, S. 141ff] klassifiziert, indem ausgehend vom Wurzelknoten verschiedene Zweige in einer Baumstruktur genommen werden und an einem Blattknoten eine der vorher festgelegten Kategorien ausgegeben wird. Während der Trainingsphase wird dieser Entscheidungsbaum so aufgebaut, dass er möglichst viele Trainingsdaten richtig klassifiziert. Für Textklassifikation repräsentieren die Verzweigungsknoten im Entscheidungsbaum Wörter, für die jeweils anhand des gegebenen Merkmalsvektor entschieden wird, ob sie im gegebenen Satz enthalten sind oder nicht. Darauf beruhend wird entweder der linke oder rechte Teilbaum exploriert.

Eine Variante der Entscheidungsbaumverfahren ist das Klassifikationsverfahren Random Forest [25, S. 160]. Dabei wird anders als bei DT nicht nur ein Entscheidungsbaum gelernt, sondern der Trainingsdatensatz wird in mehrere Teilmengen aufgeteilt und anhand von diesen werden mehrere Entscheidungsbäume konstruiert. Die Zuordnung der Eingaben zu den Klassen wird vom Algorithmus dann durch eine Abstimmung zwischen den Ausgabewerten der verschiedenen Entscheidungsbäume vorgenommen. Die Annahme, dass die Teilmengen eine zufällige Partition des Trainingsdatensatzes darstellen, verleiht diesem Verfahren den Namen Random Forest.

Eine Support Vector Machine [25, S. 167] klassifiziert die Datenpunkte, indem die Klassen durch Hyperebenen voneinander abgetrennt werden. Dafür projiziert eine SVM die Datenpunkte vom ursprünglichen Merkmalsraum in einen Dualraum, sodass

die zuvor nur nicht-linear separierbaren Datenpunkte linear (durch eine Hyperebene) separierbar sind. Während der Trainingsphase konstruiert die SVM duale Hyperebenen, die die Trainingsdaten im Merkmalsraum maximal separieren.

Li et al. [29] haben die Leistungsfähigkeit der genannten Klassifikatoren bereits für die Unterscheidung von *Entwurfsentscheidungen* zu *Keine Entwurfsentscheidung* in Sätzen der Hibernate Developer Mailing List untersucht. Dabei konnte mit einer SVM der beste F1-Wert erzielt werden. Abualhaija et al. [1] haben verschiedene Klassifikationsalgorithmen, u.a. LR, DT, RF und SVM, angewandt, um Anforderungen in textuellen Softwarespezifikationen gegeneinander abzugrenzen. Den anderen Klassifikationsalgorithmen leicht überlegen war dabei RF. Speziell für nicht-funktionale Anfoderungen haben Jha et al. [24] NB und SVM verglichen und bessere Ergebnisse für die SVM erzielt. Pranckevicius et al. [39] haben die entsprechenden Klassifikationsalgorithmen genutzt, um Mehrklassen-Textklassifikation anhand von Amazon Produktrezensionen vorzunehmen. Für diese Aufgabe erzielte LR die höchste Klassifikationsgenauigkeit, während DT die niedrigste vorweisen konnte. Dieser Vergleich zeigt, dass die Leistungsfähigkeit eines Klassifikationsfähigkeit gegenüber anderen von der Problemstellung, dem Aufbau der Trainingsdaten und dem Umfang des Trainingsdatensatzes abhängt, sodass verschiedene Methoden miteinander verglichen werden sollten.

6.2.3. Textklassifikation mit BERT

Bidirectional Encoder Representations from Transformers (BERT) ist eine von Devlin et al. [10] vorgestellte Methode, Sprachmodelle mit großen Mengen von nichtklassifizierten Daten auf großen Netzwerken vorzutrainieren und die Feinabstimmung für den speziellen Anwendungsfall nur für eine einzige Ausgabeschicht vorzunehmen [10]. BERT vereint hierfür zuvor veröffentlichte Konzepte wie Worteinbettungen (*word embeddings*), Kontextualisierung, Transfer Learning und die Transformer-Architektur.

Mikolov et al. [34] präsentierten mit *word2vec* ein Sprachmodell, das es erlaubt, Wörter mit niedriger dimensionalen Vektoren zu repräsentieren und gleichzeitig den Kontext, in dem ein Wort aufgetreten ist, zu berücksichtigen. Die von Peters et al. [38] entwickelten *Embeddings from Language Models (ELMo)* erweitern die Vektoren um eine Kontextualisierung, sodass neben der syntaktischen Interpretation eines Wortes auch dessen semantische Interpretation in die vektorielle Repräsentation einfließt. Das *Universal Language Model Fine-tuned for Text Classification (ULMFiT)* [18] schuf die Möglichkeit, das Sprachmodell auf großen Textkorpora zu trainieren und anschließend für die spezielle Klassifikationsaufgabe nur eine Feinabstimmung vorzunehmen (*Transfer Learning*). BERT vereint diese Erkenntnisse zu einem neuen Sprachmodell. Zudem beruht BERTs Architektur auf einem mehrschichtigen, bidirektionalen Transformer, der insbesondere im Vergleich zu rekurrenten neuronalen Netzen für ganze Sätze und Textabschnitte besser in der Lage ist, sich den Kontextbezug zu merken [10]. Dabei nutzt BERT die Erkenntnisse aus [49], wonach Aufmerksamkeit (*Attention*) genutzt

werden kann, um die Information aus einer ganzen Sequenz zu extrahieren, indem eine gewichtete Summe über alle vorausgegangenen Zustände des Encoders berechnet wird (Transformer-Architektur). Die Bidirektionalität führt dazu, dass sowohl Wörter vor dem zu klassifizierenden Wort als auch die nachfolgenden Wörter in die vektorielle Repräsentation einfließen. Um dies umzusetzen, werden Teile der Eingaben mit einer Maske überzogen und das Sprachmodell darauf trainiert, diese Wörter vorherzusagen (*masked language model*) [10].

Insgesamt erzielte BERT zum Zeitpunkt der Veröffentlichung Ergebnisse auf dem höchsten Stand der Technik für elf NLP-Aufgabenstellungen, darunter *Sentence Classification* und *Question Answering*, und übertraf damit bisherige regelbasierte und statistische Ansätze [10]. In [16] zeigte sich die Überlegenheit von BERT gegenüber der Verwendung des Tf-idf-Maßes in allen durchgeführten Experimenten, darunter *Sentiment Analysis* und *Information Retrieval*. Auch für die automatische Extraktion von Softwareanforderungen aus App-Rezensionen übertraf BERT zuvor verwendete NLP-Ansätze, indem mit RE-BERT eine spezielle Feinabstimmung von BERT für die Extraktion von Softwareanforderungen gefunden wurde [3].

6.3. Implementierung eines Klassifikators für Entwurfsentscheidungen

Ein möglicher Anwendungsbereich für das in dieser Arbeit entwickelte Klassifikationsschema ist die automatisierte Klassifikation von Entwurfsentscheidungen. Um diesen zu untersuchen, wurde eine Proof-of-Concept-Implementierung eines Klassifikators unter Verwendung verschiedener Ansätze des maschinellen Lernens (vgl. Abschnitt 6.2) umgesetzt. Die Implementierung ist in Python geschrieben, der meist benutzten Sprache für maschinelles Lernen [51]. Python erfreut sich aus verschiedenen Gründen so großer Beliebtheit für die Implementierung von Ansätzen des maschinellen Lernens. Neben den recht niedrigen Eintrittsbarrieren aufgrund von recht einfacher Syntax und der Flexibilität der Sprache besticht Python insbesondere durch ein ausgeprägtes Umfeld von Bibliotheken und Programmierschnittstellen [30]. Mit *Scikit-learn*, *Pandas* und *NLTK* (*Natural Language Toolkit*) bestehen Bibliotheken, die die Implementierung aller gängigen Lernverfahren, Methoden zur Strukturierung und Analyse von großen Datenmengen beziehungsweise für die Verarbeitung von natürlicher Sprache zur Verfügung stellen [30]. Das von Pedregosa et al. entwickelte *Scikit-learn* [37] bietet eine große Auswahl an überwachten und unüberwachten Lernalgorithmen, eingebettet in eine konsistente und anwendungsorientierte Schnittstelle.

Die automatisierte Klassifikation von Entwurfsentscheidungen lässt sich dabei in sechs Schritte einteilen (vgl. Abbildung 6.1). Zunächst müssen aus den Rohdaten des Korpus die Sätze und Label extrahiert werden und so zueinander gebracht werden,

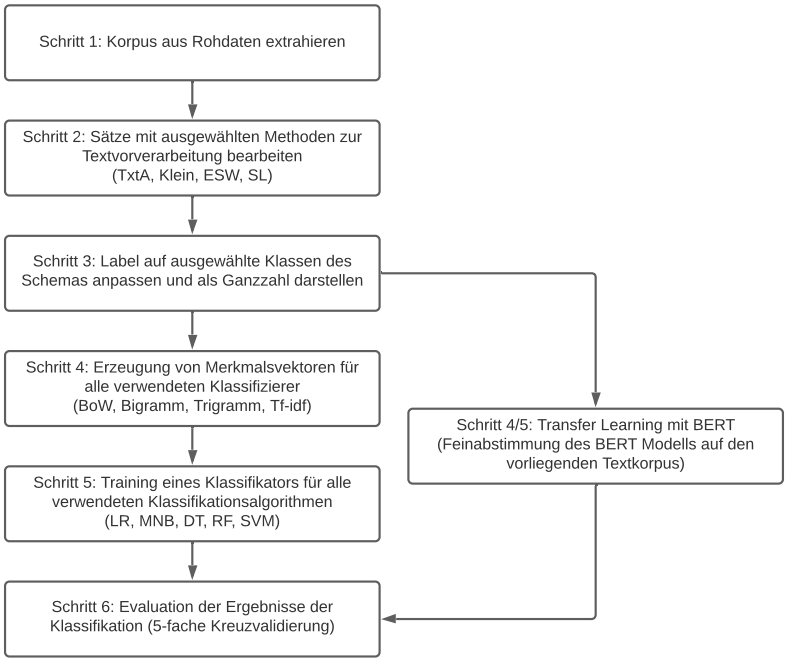

Abbildung 6.1.: Schritte zur automatisierten Klassifikation von Entwurfsentscheidungen

dass die Sätze mit dem zugehörigen Label vorliegen. Die Sätze werden anschließend den ausgewählten Methoden der Textvorverarbeitung unterzogen. Auch für die Label muss eine Vorverarbeitung vorgenommen werden, da nicht alle Klassen des Schemas zur Anwendung kommen und daher die Label auf eine gemeinsame Oberklasse vereinheitlicht werden müssen. Aus dem nun vorverarbeiteten Korpus können mithilfe von BoW, Bigramm, Trigramm oder Tf-idf Vektorrepärsentationen der einzelnen Sätze erzeugt werden, die als Eingabe in den Klassifikationsalgorithmus dienen. Für alle Klassifikationsalgorithmen (LR, MNB, DT, RF, SVM) wird so ein Klassifikator trainiert, der die Trainingsdaten möglichst gut klassifiziert. Die Leistungsfähigkeit dieses Klassifikators wird anschließend an ungesehenen Daten im Rahmen einer 5-fachen Kreuzvalidierung evaluiert. Statt den Varianten zur Vektorrepräsentation und den genannten Klassifikationsalgorithmen kann alternativ auch ein sogenanntes Transfer Learning des BERT-Modells vorgenommen werden. Dabei werden die Sätze in die BERT-eigene Vektorrepräsentation unter Verwendung von Worteinbettungen überführt und anschließend das vortrainierte *Bert-base-uncased*-Modell für eine Klassifikation von Sätzen (*BertForSequenceClassification*) anhand der Trainingsdaten verfeinert.

6.3.1. Vorverarbeitung von Text und Labeln

Für die Anwendung der Lernverfahren müssen sowohl die Eingabesätze als auch die Label des zugrunde liegenden Datensatzes vorverarbeitet werden. Für die Vorverarbeitung des Textes stehen folgende Methoden zur Verfügung (vgl. Unterabschnitt 6.2.1):

- Textaufbereitung hinsichtlich Sonderzeichen, Ziffern und Hyperlinks (TxtA)
- Reduktion auf Kleinschreibung (Klein)
- Entfernung von Stoppwörtern (ESW)
- Stammformreduktion und Lemmatisierung (SL)

Hinsichtlich der Label gilt folgendes: Aufgrund der Größe des Klassifikationsschemas und dem damit einhergehend Problem, dass für die tief in der Hierarchie des Schemas angesiedelten Klassen nur noch wenige Sätze zugeordnet werden können, können in der Proof-of-Concept-Implementierung nicht alle Klassen umgesetzt werden. Deshalb wird entlang der hierarchischen Schichten für ausgewählte Klassen analysiert. Dafür wurde die baumartige Struktur des Klassifikationsschemas in der Implementierung nachgebildet, sodass für einen gegebenen Wurzelknoten alle anhängenden Teilbäume traversiert werden können, um dadurch alle Unterklassen zu ermitteln. So können Unterklassen in Oberklassen zusammengefasst werden und mehr Datenpunkte pro Klasse verwendet werden. Es bestehen Implementierungen für die folgenden Ebenen im Klassifikationsschema:

- Ebene 0: *Entwurfsentscheidung* und *Keine Entwurfsentscheidung*
- Ebene 1: *Existenz-*, *Eigenschafts-* und *Ausführungsentscheidung*
- Ebene 2: *Struktur-*, *Verhaltens-* und *Anordnungsentscheidung*
- Ebene 3: *Komponente*, *Objekt*, *Referenz* und *Funktionen & Kommunikation*

Während die Klassifikation auf den Ebenen 0 und 1 vollständig in der Breite des Klassifikationsschemas ist, handelt es sich auf Ebene 2 und 3 um ausgewählte Klassen aus dem Schema. Die *Struktur-*, *Verhaltens-* und *Anordungsentscheidungen* sind Unterklassen der besonders häufig identifizierten *Existenzentscheidungen*. *Komponente* und *Objekt* sind wiederum Unterklassen der *Strukturentscheidungen*, *Referenz* und *Funktionen & Kommunikation* Unterklassen der *Verhaltensentscheidungen* (vgl. Abschnitt 5.1). Ihr gehäuftes Vorkommen in der analysierten Softwarearchitekturdokumentation (vgl. Abschnitt 5.2) erleichtert die Klassifikation aufgrund der größeren Menge an Trainingsdaten. Für die Ebenen 1 bis 3 ergibt sich zudem jeweils noch eine zusätzliche Klasse, die alle Sätze umfasst, die entweder keine Entwurfsentscheidung enthalten oder keiner der ausgewählten Klassen zugeordnet werden können.

Nachdem der Datensatz aus den txt- und csv-Dateien geladen wurde, wird dieser tabellarisch strukturiert in einem *DataFrame* der *Pandas*-Bibliothek gespeichert. Innerhalb dieser Datenstruktur können nun die Label angepasst werden, indem sie zunächst auf die ausgewählten Oberklassen abgestimmt werden und anschließend anhand einer Klasse-Index-Zuordnung in ganze Zahlen überführt werden. Für eine mögliche Multi-Label-Klassifikation werden die Label als Vektor dargestellt, wobei eine 1 an Index i besagt, dass der gegebene Satz Klasse i zugeordnet werden kann. Die Überführung der Labels in ganze Zahlen erleichtert die Nutzung der Schnittstellen von *Scikit-learn* und *BERT*, die nur teilweise auch textuelle Label als Eingabe akzeptieren.

6.3.2. Training eines Klassifikators und k-fache Kreuzvalidierung

Um die in Unterabschnitt 6.2.1 genannten Vektorrepräsentationen und die in Unterabschnitt 6.2.2 vorgestellten Klassifikationsalgorithmen zu vergleichen, wird die von *Scikit-learn* [7] zur Verfügung gestellte *Pipeline* verwendet. Dadurch können sequentiell nacheinander eine Liste von Transformationen in eine Vektordarstellung und anschließend ein Klassifikator angewendet werden. Entlang dieser *Pipeline* wird eine k-fache Kreuzvalidierung (*k-fold cross-validation*) durchgeführt. Häufig genutzt werden 5-fache und 10-fache Kreuzvalidierung. Da für größeres k die Anzahl der Datenpunkte im Evaluationsdatensatz abnimmt und die zur Verfügung stehenden Datenpunkte pro Klasse generell für diese Arbeit eher begrenzt sind, wird eine 5-fache Kreuzvalidierung angewandt. Dafür wird der Datensatz in fünf Teile zerteilt und in jedem Durchlauf des Kreuzvalidierungsverfahrens werden ein Teil zur Evaluation und vier Teile für das Training verwendet. Dadurch ergeben sich insgesamt fünf Durchläufe, über die ein durchschnittlicher Wert für die Genauigkeit (*accuracy*) sowie für *Precision*, *Recall* und *F1-Wert* berechnet werden kann. Die Verwendung einer k-fachen Kreuzvalidierung führt dazu, dass die statistischen Werte weniger durch die Auswahl von Trainings- und Testdatensatz verzerrt sind.

Für die Textklassifikation mit BERT wurde die *Simpletransformers*-Bibliothek [40] verwendet. Sie stellt eine vereinfachte Programmierschnittstelle für die Verwendung zahlreicher Transformer-Modelle dar unter gleichzeitiger Verwendung der Methoden zur Berechnung von Evaluationsmetriken aus *Scikit-learn*. Die Transformer-Modelle entstammen der *Huggingface*-Bibliothek [57], die eine Sammlung von über 70 Transformer-Modellen bereithält, sodass auch Erweiterungen zu anderen Modellen (z.B. RoBERTa oder DistilBERT) innerhalb dieser Implementierung vorgenommen werden können.

6.4. Evaluation der automatisierten Analyse

Dieser Abschnitt evaluiert die automatisierte Identifikation und Klassifikation von Entwurfsentscheidungen für verschiedene Modellkombinationen der zuvor beschrie-

benen Ansätze und liefert die zugehörigen Auswertungstabellen. Dafür wurden für verschiedene Konfigurationen anhand der 5-fachen Kreuzvalidierung die statistischen Maße Korrektklassifizierungsrate und das F1-Maß (vgl. Unterabschnitt 6.4.1) berechnet. Die Evaluation versucht hierbei die folgenden Fragen zu beantworten:

Methoden der Textvorverarbeitung
Welche Methoden zur Textvorverarbeitung sind geeignet, um den hier verwendeten Textkorpus vor der Klassifikation aufzubereiten? (Unterabschnitt 6.4.1.1)

Modellkombination zur Identifikation von Entwurfsentscheidungen
Welche Modellkombination sollte gewählt werden, um Sätze mit Entwurfsentscheidungen von solchen ohne Entwurfsentscheidung abzugrenzen? (Unterabschnitt 6.4.1.2)

Vektorrepräsentation-Klassifikationsalgorithmus-Kombination
Welche Kombination aus einer Vektorrepräsentation für die Sätze (BoW, Bigramm, Trigramm, Tf-idf) und einem Klassifikationsalgorithmus (LR, MNB, DT, RF, SVM) liefert auf den unterschiedlichen Ebenen die besten Ergebnisse? (Unterabschnitt 6.4.1.3)

Textklassifikation mit BERT
Kann mit der Verwendung von BERT die Klassifikation auf den verschiedenen Ebenen noch verbessert werden? (Unterabschnitt 6.4.1.4)

Multi-Label-Klassifikation
Ist die Klassifikation von mehreren Entwurfsentscheidungen pro Satz mit einer Multi-Label-Klassifikation möglich und wenn ja, in welcher Modellkombination? (Unterabschnitt 6.4.1.5)

6.4.1. Erzielte Ergebnisse in der Klassifikation

Für den Vergleich der unterschiedlichen Methoden zur Textvorverarbeitung sowie der verwendeten Vektorisierer und Klassifikationsalgorithmen werden die statistischen Maße Korrektklassifizierungsrate (Treffergenauigkeit, *accuracy*) und F1-Wert berechnet. Die Korrektklassifizierungsrate berechnet sich als:

$$KKR = \frac{\text{\#Vorhersagen richtig}}{\text{\#Vorhersagen gesamt}} = \frac{r_p + r_n}{r_p + r_n + f_p + f_n} \tag{6.1}$$

wobei entsprechend einer Konfusionsmatrix gilt:

r_p = richtig positiv
r_n = richtig negativ
f_p = falsch positiv
f_n = falsch negativ

Der F1-Wert berechnet sich durch das harmonische Mittel zwischen Genauigkeit (*precision*, P) und Trefferquote (*recall*, R), also:

$$F1 = \frac{2 \times \mathrm{P} \times \mathrm{R}}{\mathrm{P} + \mathrm{R}} = \frac{2 \times r_p}{2 \times r_p + f_p + f_n} \tag{6.2}$$

Für den nicht-binären Fall, also wenn mehr als zwei Klassen vorliegen, denen die Datenpunkte zugeordnet werden können, bedarf es einer Anpassung der Berechnung des F1-Wertes. Für die Klassifikation auf den Ebenen 1 bis 3 wird daher ein gewichteter F1-Wert berechnet, um gleichzeitig der Ungleichverteilung der Daten in den verschiedenen Klassen gerecht zu werden. Dafür wird der F1-Wert zunächst für jede Klasse einzeln berechnet und anschließend eine Summe über alle F1-Werte gewichtet mit der relativen Häufigkeit von Datenpunkten (*weight*, W) in der Klasse berechnet:

$$F1_{gewichtet} = F1_{Klasse1} \times W_{Klasse1} + F1_{Klasse2} \times W_{Klasse2} + \ldots + F1_{KlasseN} \times W_{KlasseN} \tag{6.3}$$

Der F1-Wert ist insbesondere dann eine aussagekräftigere Metrik, wenn die Datenpunkte ungleich auf die Klassen verteilt sind und falsch positive und falsch negative Klassifikationen bedeutsam sind [19]. Die Ungleichverteilung der Daten ist im Falle des verwendeten Textkorpus gegebenen und falsch negative Klassifikationen wiegen besonders schwer, da in diesem Fall eine Klasse von Entwurfsentscheidungen in der Dokumentation übersehen wurde. Aber auch falsch positive Klassifikationen sollten vermieden werden, da eine Konsistenzprüfung sonst hier jeweils ein fehlendes Softwareartefakt bemängelt, welches jedoch nie spezifiziert wurde.

6.4.1.1. Vergleich für unterschiedliche Textvorverarbeitung

Um geeignete Methoden der Textvorverarbeitung für den gegebenen Anwendungsfall und den vorliegenden Datensatz zu bestimmen, wurde eine 5-fache Kreuzvalidierung unter Anwendung aller in Unterabschnitt 6.2.1 genannten Vektorisierer (BoW, Tf-idf, Bigramm, Trigramm) und aller in Unterabschnitt 6.2.2 genannten Klassifikationsalgorithmen (LR, MNB, DT, RF, SVM) für verschiedene Konfigurationen der Textvorverarbeitung durchgeführt. Tabelle 6.1 zeigt den jeweils durchschnittlichen und maximalen Wert für die statistischen Maße Korrektklassifizierungsrate (KKR) und F1 über alle Vektorisierer in Kombination mit allen Klassifikationsalgorithmen auf Ebene 0. Dabei zeigt sich zunächst, dass ohne jegliche Textvorverarbeitung gute Ergebnisse erzielt werden können (F1-⌀: 0,8730, F1-Max: 0,8968). Die besten Ergebnisse erzielen jedoch Klassifikatoren unter Verwendung der Reduktion auf Kleinschreibung, sowohl hinsichtlich durchschnittlicher (F1-⌀: 0,8735) als auch maximaler Leistung (F1-Max: 0,8978). Ähnlich gute Ergebnisse erzielt die Stammformreduktion/ Lemmatisierung (F1-Max: 0,8971) sowie die Verwendung der Kombination aus Reduktion auf Kleinschreibung

Tabelle 6.1.: Vergleich der durchschnittlichen und maximalen Korrektklassifizierungsrate (KKR) und F1-Werte erzielt unter Verwendung unterschiedlicher Textvorverarbeitung auf Ebene 0 (beste drei Werte jeweils grau hinterlegt)

Textvorverarbeitung	KKR		F1-Wert	
	Ø	**Max**	**Ø**	**Max**
Keine Vorverarbeitung	0,8036	0,8395	0,8730	0,8952
Textaufbereitung (TxtA)	0,7915	0,8335	0,8647	0,8924
Kleinschreibung (Klein)	0,8044	0,8395	0,8735	0,8978
Entf. Stoppwörter (ESW)	0,7988	0,8358	0,8705	0,8968
Stammform/Lemma (SL)	0,8030	0,8366	0,8726	0,8971
TxtA + Klein	0,7916	0,8340	0,8647	0,8925
Klein + SL	0,8031	0,8373	0,8726	0,8969
TxtA + Klein + ESW	0,7826	0,8202	0,8567	0,8876
TxtA + Klein + SL	0,7929	0,8340	0,8656	0,8950
Klein + ESW + SL	0,8021	0,8358	0,8712	0,8958
TxtA + Klein + ESW + SL	0,7878	0,8264	0,8611	0,8898

und Stammformreduktion/ Lemmatisierung (F1-Max: 0,8969). Die Textaufbereitung hinsichtlich Sonderzeichen, Ziffern und Hyperlinks hatte sowohl für sich genommen (F1-Max: 0,8928) als auch in Kombination mit den anderen Methoden der Textvorverarbeitung, etwa zusammen mit Reduktion auf Kleinschreibung und Stammformreduktion/ Lemmatisierung (F1-Max: 0,8950), einen (geringen) negativen Effekt auf die erzielten Ergebnisse des Klassifikators. Ebenso verhält es sich mit der Entfernung von Stoppwörtern als Einzelanwendung (F1-Max: 0,8953) sowie in Kombination mit Reduktion auf Kleinschreibung und Stammformreduktion/ Lemmatisierung (F1-Max: 0,8958). Für die Kombination aller hier aufgezählten Methoden zur Textvorverarbeitung kehrt sich dieser negative Effekt nicht um (F1-Max: 0,8898). Insbesondere scheint die Hinzunahme der Textaufbereitung hinsichtlich Sonderzeichen, Ziffern und Hyperlinks die Ergebnisse eher zu verschlechtern.

Insgesamt lässt sich aus den erzielten Ergebnissen ableiten, dass die Anwendung der Reduktion auf Kleinschreibung sowohl in der durchschnittlichen Leistungsfähigkeit über alle Vektorisierer-Klassifikationsalgorithmus-Kombinationen als auch in der maximalen Leistungsfähigkeit für die jeweils beste Kombination hier am vielversprechendsten funktioniert hat. Auch die Kombination aus Reduktion auf Kleinschreibung und Stammformreduktion/ Lemmatisierung erweist sich als sinnvolle Textvorverarbeitung für diesen Korpus. Von der Textaufbereitung hinsichtlich Sonderzeichen, Ziffern und Hyperlinks ist hingegen eher abzuraten. Die maximalen Werte für KKR und F1 entstammen dabei, unabhängig von der Textvorverarbeitung, speziell den Kombinationen aus BoW mit LR sowie RF mit Bigramm und Trigramm.

Tabelle 6.2.: Vergleich der Korrektklassifizierungsrate (KKR) und der F1-Werte für die verwendeten Vektorisierer und Klassifikationsalgorithmen für Ebene 0

KKR	**BoW**	**Bigramm**	**Trigramm**	**tf-idf**
LR	0,8395	0,8155	0,8362	0,7898
MNB	0,7982	0,8023	0,8237	0,7723
DT	0,7760	0,7575	0,7801	0,7698
RF	0,8241	0,8366	0,8381	0,8249
SVM	0,8165	0,8200	0,8276	0,8068

F1-Wert	**BoW**	**Bigramm**	**Trigramm**	**tf-idf**
LR	0,8937	0,8749	0,8896	0,8737
MNB	0,8760	0,8725	0,8869	0,8655
DT	0,8483	0,8358	0,8502	0,8489
RF	0,8864	0,8971	0,8978	0,8871
SVM	0,8833	0,8850	0,8907	0,8827

6.4.1.2. Vergleich für unterschiedliche Vektorisierer und Klassifikationsalgorithmen zur Identifikation von Sätzen mit Entwurfsentscheidungen

Für die Unterscheidung auf Ebene 0, also der Abtrennung von Sätzen, die Entwurfsentscheidungen enthalten und solchen, die keine enthalten, zeigte der Vergleich für verschiedene Vektorisierer-Klassifikationsalgorithmus-Kombinationen drei zu favorisierende Konfigurationen. Tabelle 6.2 zeigt den Vergleich für die Korrektklassifizierungsraten (KKR) und für die F1-Werte, wobei sich die angegebenen Werte jeweils als Maximum über die eingesetzten Methoden zur Textvorverarbeitung ergeben. Den besten F1-Wert erzielte für die Klassifikation in *Entwurfsentscheidung* und *Keine Entwurfsentscheidung* ein Random-Forest-Klassifikator (RF) mit Trigramm (0,8978), dicht gefolgt von der Kombination RF mit Bigramm (0,8971). Die beste KKR (0,8395) und den drittbesten F1-Wert (0,8937) erzielte die Kombination aus Logistische Regression (LR) und Bag-of-Words (BoW). Auch für die anderen in Unterabschnitt 6.4.1.1 analysierten Konfigurationen für die Textvorverarbeitung stachen diese drei Kombinationen mit guten Ergebnissen hervor, in Einzelfällen erweitert um LR mit Trigramm (F1-Max: 0,8896). Mit einer Support Vector Machine (SVM) konnte zusammen mit Überführung in Vektorrepärsentation mittels Trigramm zumindest noch annähernd ein solcher F1-Wert erreicht werden (0,8907), während Decision Tree (DT) und Multinomialer Naiver Bayes (MNB) für die automatisierte Identifikation von Sätzen mit Entwurfsentscheidungen deutlich gegenüber den anderen Verfahren zurückbleiben. Auch die Verwendung von Tf-idf gegenüber der reinen Berechnung der Vorkommenshäufigkeit der Wörter in Uni-, Bi- oder Trigrammen ist mit Abstrichen in den erzielten Ergebnissen verbunden. Dies

ist möglicherweise darauf zurückzuführen, dass die Eingaben für den Klassifikator Sätze und keine Dokumente sind und dadurch die Inverse Dokumentenhäufigkeit keinen Mehrwert liefert.

6.4.1.3. Klassifikation auf unterschiedlichen hierarchischen Ebenen des Klassifikationsschemas

Das in dieser Arbeit entwickelte Klassifikationsschema ist in der Lage, Entwurfsentscheidungen anhand einer hierarchischen Struktur zu klassifizieren. Für die in Unterabschnitt 6.3.1 eingeführten Ebenen der hier durchgeführten Klassifikation lassen sich Gemeinsamkeiten und Unterschiede hinsichtlich der besten Modellkombination feststellen (vgl. Tabelle 6.3). Für alle untersuchten hierarchischen Ebenen ist die Textvorverarbeitung mittels Reduktion auf Kleinschreibung (Klein) und die Verwendung eines Vektorisierers, der mit Trigrammen (Wortfolgen von jeweils drei Wörtern) arbeitet, die Modellkombination, die die besten F1-Werte liefert. Für die einzusetzenden Klassifikationsalgorithmen unterscheiden sich die verschiedenen Ebenen: Während auf Ebene 0 (*Entwurfsentscheidung* abgegrenzt von *Keine Entwurfsentscheidung*) ein Random-Forest-Klassifikator (RF) die besten Ergebnisse erzielt, können Struktur-, Verhaltens- und Anordnungsentscheidungen auf Ebene 1 am besten durch Logistische Regression (LR) voneinander abgegrenzt werden. Für Klassen, die tiefer im Klassifikationsschema angeordnet sind, lieferte Multinomialer Naiver Bayes (MNB) die besten Ergebnisse. Für die Ebenen 1 bis 3 konnten Korrektklassifizierungsraten (KKR) und gewichtete F1-Werte zwischen 60% und 70% erzielt werden.

Tabelle 6.3.: Favorisierte Modellkombination für verschiedene Ebenen anhand der maximal möglichen Leistungfähigkeit hinsichtlich des F1-Wertes (alle Sätze)

	Modellkombination			**KKR**	**F1**
	Textvorverarbeitung	**Vektorisierer**	**Klassifikationsalgorithmus**		
Ebene 0	Klein	Trigramm	RF	0,8381	0,8978
Ebene 1	Klein	Trigramm	LR	0,6675	0,6559
Ebene 2	Klein	Trigramm	MNB	0,6274	0,6228
Ebene 3	Klein	Trigramm	MNB	0,6810	0,6767

Dabei befinden sich auf den Ebenen 1 und 2 jeweils drei aus dem Schema ausgewählte Klassen sowie eine Klasse, der Sätze, die keiner der Klassen angehören, zugeordnet werden. Für die Ebene 3 sind es vier ausgewählte Klassen und die beschriebene „Auffangklasse“. Für Ebene 1 ergibt sich ein F1-Wert von 0,6559, für Ebene 2 von 0,6228 und für Ebene 3 von 0,6767. Für die tiefer liegenden Ebenen im Schema ergeben sich mehr Klassen, die voneinander unterschieden werden müssen, sodass auf Ebene 0 noch

ein F1-Wert von knapp 90% möglich ist, dieser auf Ebene 1 auf etwa 66% abfällt und auf Ebene 2 und 3 ebenfalls nur noch bei 62% beziehungsweise 68% liegt. Durch mehr Klassen (Ebene 0 ist binär, Ebene 1 und 2 haben vier Klassen, Ebene 3 sogar fünf) ist der F1-Wert erwartbar kleiner. Außerdem müssen die Sätze an immer feineren Unterscheidungsmerkmalen voneinander abgegrenzt werden. Interessant ist dabei allerdings, dass die Ergebnisse auf Ebene 3 besser sind als auf Ebene 1 und 2; also scheinen Sätze hinsichtlich *Komponente*, *Objekt*, *Referenz* und *Funktionen & Kommunikation* besser trennbar zu sein als hinsichtlich der darüber liegenden Klassen *Struktur-*, *Verhaltens-* und *Anordnungsentscheidung* beziehungsweise *Existenz-*, *Eigenschafts-* und *Ausführungsentscheidung*. Insgesamt zeigt sich aber auch, dass die Trennlinien zwischen den Klassen des Klassifikationsschemas in natürlichsprachiger Softwarearchitekturdokumentation durch NLP-Ansätze gelernt werden können, sodass eine automatisierte Klassifikation möglich erscheint (siehe auch Unterabschnitt 6.4.2).

Tabelle 6.4.: Favorisierte Modellkombination für verschiedene Ebenen anhand der maximal möglichen Leistungfähigkeit hinsichtlich des F1-Wertes (nur Sätze mit Entwurfsentscheidungen)

	Modellkombination			**KKR**	**F1**
	Textvorverarbeitung	**Vektorisierer**	**Klassifikationsalgorithmus**		
Ebene 0	-	-	-	-	-
Ebene 1	Klein	Trigramm	MNB	0,7569	0,7414
Ebene 2	Klein	Trigramm	MNB	0,6102	0,6004
Ebene 3	Klein	Trigramm	MNB	0,6351	0,6262

Es besteht zudem die Möglichkeit, die Klassifikation der Entwurfsentscheidungen nur anhand von Sätzen vorzunehmen, die auch eine Entwurfsentscheidung enthalten. Diese Abgrenzung kann etwa auf Ebene 0 mit einem Random-Forest-Klassifikator, der Trigramme verwendet, erfolgen. Trainiert man die Klassifikationsalgorithmen nur mit Sätzen, die auch eine Entwurfsentscheidung enthalten, ergibt sich auf Ebene 1 ein maximaler F1-Wert von 0,7414, auf Ebene 2 von 0,6004 und auf Ebene 3 von 0,6262 (vgl. Tabelle 6.4). Als beste Modellkombination ist nun für die Ebenen 1 bis 3 stets die Kombination aus Reduktion auf Kleinschreibung, Vektorrepräsentation mittels Trigrammen und Klasifikation durch Multinomialen Naiven Bayes zu wählen. Diese Werte sind weniger beeinflusst durch die Klassifikation von allen Sätzen ohne Entwurfsentscheidungen in die „Auffangklasse“, sondern drücken aus, inwieweit der Klassifikator in der Lage ist, die entsprechende Klasse des Schemas in einem Satz zu erkennen. Für Ebene 1 ist hinsichtlich der Verbesserung gegenüber der Klassifikation mit allen Sätzen zu beachten, dass nun keine Beispiele mehr in eine „Auffangklasse“ fallen

und die Klassifikation vollständig innerhalb der drei Klassen *Existenz-*, *Eigenschafts-* und *Ausführungsentscheidung* erfolgt.

6.4.1.4. Vergleich von BERT zu den anderen Klassifikationsalgorithmen

Statt den zuvor beschriebenen Vektorrepräsentationen und Klassifikationsalgorithmen kann auch eine Feinabstimmung des BERT-Modells eine Klassifikation für diesen Textkorpus liefern (vgl. Unterabschnitt 6.2.3). Der Vergleich der Ergebnisse hinsichtlich Korrektklassifizierungsrate (KKR) und (gewichtetem) F1-Wert zeigt, dass BERT für die Abgrenzung von Sätzen mit Entwurfsentscheidungen von Sätzen ohne Entwurfsentscheidung (Ebene 0) bessere Ergebnisse erzielt als alle anderen hier untersuchten Kombinationen aus Vektorisierern und Klassifikatoren (vgl. Tabelle 6.5). Mit BERT Transfer Learning konnte eine KKR von 0,8769 und ein F1-Wert von 0,9082 innerhalb einer 5-fachen Kreuzvalidierung erzielt werden. Für die ausgewählten Klassen auf den Ebenen 1, 2 und 3 konnte die Klassifikation mit BERT keine besseren Ergebnisse hervorbringen als die jeweils zuvor gefundene beste Modellkombination, sondern bleibt insbesondere hinsichtlich des gewichteten F1-Wertes hinter diesen zurück. Für Ebene 1 ergibt sich ein F1-Wert von 0,6185 (max.: 0,6675), für Ebene 2 von 0,5535 (max.: 0,6559) und für Ebene 3 von 0,6351 (max.: 0,6767). Insgesamt lässt sich daraus ableiten, dass die Verwendung von BERT für die Klassen *Entwurfsentscheidung* und *Keine Entwurfsentscheidung* einen Mehrwert liefert, während für Klassen innerhalb des entwickelten Klassifikationsschemas hier nicht gezeigt werden konnte, dass BERT eine bessere Alternative im Vergleich zu den zuvor untersuchten Modellkombinationen darstellt.

Tabelle 6.5.: Vergleich von BERT Transfer Learning mit der jeweils besten Modellkombination für eine Klassifikation auf den Ebenen 0 bis 3

	Klassifikationsalgorithmus			**BERT**	
	Modellkombination	**KKR**	**F1-Wert**	**KKR**	**F1-Wert**
Ebene 0	Klein + Trigramm + RF	0,8381	0,8978	0,8769	0,9082
Ebene 1	Klein + Trigramm + LR	0,6675	0,6559	0,6449	0,6185
Ebene 2	Klein + Trigramm + MNB	0,6274	0,6559	0,6227	0,5535
Ebene 3	Klein + Trigramm + MNB	0,6810	0,6767	0,6973	0,6351

6.4.1.5. Multi-Label-Klassifikation mit verschiedenen Modellkombinationen

Die zuvor beschriebenen Ergebnisse beruhen auf einer Klassifikation der Sätze nur hinsichtlich einer Klasse des Schemas, nämlich der Klasse von Entwurfsentscheidungen, die die Hauptaussage des Satzes am stärksten repräsentiert. Die Label, die die

Hauptaussage des Satzes am besten widerspiegeln, wurden während der Erstellung des Korpus in einer gemeinsamen Spalte abgelegt. Es bestand jedoch darüber hinaus noch die Möglichkeit, Sätzen eine zweite Klasse des Schemas zuzuordnen. Verwendet man diese zweite Spalte des Korpus ebenfalls und erstellt aus beiden Spalten für jeden Satz ein Multi-Label, dann ergeben sich leicht veränderte optimale Modellkombinationen sowie veränderte statistische Maße (vgl. Tabelle 6.6). Für Ebene 1 und 2 entsprechen die Modellkombinationen den bereits zuvor identifizierten Kombinationen aus Reduktion auf Kleinschreibung (Klein), Vektorrepräsentation mittels Trigramm und Klassifikation durch Logistische Regression (LR) für Ebene 1 beziehungsweise Multinomialer Naiver Bayes (MNB) für Ebene 2. Für Ebene 3 ist nun anstatt MNB die LR der Klassifikationsalgorithmus, der in Kombination mit Kleinschreibung und Trigramm den besten gewichteten F1-Wert liefert. Die F1-Werte belaufen sich für Ebene 1 auf 0,6279 (ohne Multi-Label: 0,6559), auf Ebene 2 auf 0,6255 (ohne Multi-Label: 0,6228) und auf Ebene 3 auf 0,5917 (ohne Multi-Label: 0,6767) (vgl. Tabelle 6.6, Tabelle 6.3). Diese fallen zwar eher etwas schlechter als die F1-Werte ohne Multi-Label aus, umfassen jedoch gleichzeitig mehr Kombinationsmöglichkeiten und versuchen bis zu zwei Klassen von Entwurfsentscheidungen in jedem Satz zu identifizieren. Die Ergebnisse zeigen, dass, zwar mit Abstrichen verbunden, eine Klassifikation auch anhand von Multi-Labeln möglich ist.

Tabelle 6.6.: Multi-Label-Klassifikation auf verschiedenen Ebenen, gegeben beste Modellkombination hinsichtlich gewichtetem F1-Wert sowie zugehörige Korrektklassifizierungsrate (KKR)

	Modellkombination			**KKR**	**F1**
	Textvorverarbeitung	**Vektorisierer**	**Klassifikationsalgorithmus**		
Ebene 1	Klein	Trigramm	LR	0,5113	0,6279
Ebene 2	Klein	Trigramm	MNB	0,4961	0,6255
Ebene 3	Klein	Trigramm	LR	0,6781	0,5917

6.4.2. Validität der erzielten Ergebnisse

Runeson und Höst [44] stellen in ihren Leitlinien zur Durchführung und zum Vortrag von auf Fallstudien basierter Forschung in der Softwaretechnik Kriterien für die Validität einer Datenanalyse auf (auch verwendet in [29]). Diese Kriterien umfassen die Validität der Konstruktion (*construct validity*), die interne (*internal validity*) und externe Validität (*external validity*) und die Verlässlichkeit (*reliability*).

Die *Validität der Konstruktion* besagt, inwiefern die angewandten Methoden und Auswertungen die mit der Forschungsfrage benannten Ziele abbilden können [44, S.153]. Da

es sich um eine Proof-of-Concept-Implementierung handelt, ist es das übergeordnete Ziel dieser Arbeit, zu untersuchen, ob mit dem entwickelten Klassifikationsschema eine automatisierte Analyse mittels maschinellem Lernen und NLP durchgeführt werden kann. Dafür müssen die Trennlinien der Klassen des Schemas anhand von Trainingsdaten gelernt werden können, um für ungesehene Eingaben Vorhersagen treffen zu können. Die grundsätzliche Eignung hierzu lässt sich eindeutig aus den erzielten Ergebnissen ableiten. Außerdem wurden einzelne verfeinerte Fragestellungen hinsichtlich der Kombination von Textvorverarbeitung, Vektorrepräsentation und Klassifikationsalgorithmen aufgeworfen (vgl. Forschungsfragen zu Beginn von Abschnitt 6.4). Für verschiedene Durchläufe konnten bessere gegen schlechtere Konfigurationen abgegrenzt werden und dabei auch Regelmäßigkeiten festgestellt werden, sodass sich bestimmte Konfigurationen unabhängig von der weiteren Modellkombination als überlegen für diesen Textkorpus herausgestellt haben. Dabei zu beachten ist jedoch, dass nicht alle Modellkombinationen getestet werden konnten, da ihre schiere Anzahl den Umfang dieser Arbeit übersteigt. Daher ist nicht auszuschließen, dass eine Modellkombination existiert, die auf diesem Korpus noch bessere Ergebnisse liefern kann. Gar nicht betrachtet werden konnte die Feinabstimmung der verwendeten Lernalgorithmen, etwa mit Parametern wie Lernrate, Epochenanzahl und Schwellwert. Eine verbesserte Abstimmung dieser Parameter auf das vorliegende Klassifikationsproblem kann die Ergebnisse noch verbessern. Die Validität der Konstruktion muss sich zudem daran messen lassen, ob der entwickelte Korpus dafür geeignet ist, allgemeine Rückschlüsse für die automatisierte Analyse von Entwurfsentscheidungen ziehen zu können. Dem begegnet wurde durch eine Auswahl an Fallstudien, die sich in der Art des entwickelten Softwaresystems sowie der Ausführlichkeit der Dokumentation deutlich voneinander unterscheiden. Dennoch ließen sich aus einem Korpus mit einer größeren Anzahl an Beispielen gesichertere Aussagen ableiten. Insbesondere können mit einem größeren Textkorpus die Ergebnisse zusätzlich noch anhand von Signifikanzniveaus analysiert werden, um die Aussagekraft geringer Abweichungen einzuordnen. Auch die Entwicklung von zwei voneinander unabhängigen Korpora anhand des gleichen Schemas, die zum einen als Trainings- und zum anderen als Evaluationsdatensatz dienen, stellt eine Möglichkeit zur weiteren Sicherung der Validität dar.

Zusätzlich besteht eine Gefahr für die Validität hier noch darin, dass der Korpus selbst möglicherweise bereits fehlerhafte Klassenzuordnungen aufweist, d.h. die Label während der Korpuserstellung falsch gesetzt wurden. Entgegengewirkt wurde dieser Gefahr dadurch, dass die Klassifikation nach der Weiterentwicklung des Schemas in einer abschließenden Anwendung noch einmal überarbeitet wurde. Alle Label wurden jedoch von einer einzigen Person gesetzt und haben dadurch keine Überprüfung nach dem „Vier-Augen-Prinzip“ erfahren. Um dieses Problem zu adressieren, müsste die Zuordnung der Label daher durch unterschiedliche Personen durchgeführt werden und anschließend der Grad der Übereinstimmung gemessen werden (Interrater-Reliabilität).

Die *interne Validität* der automatisierten Analyse ist dann gegeben, wenn die Ergebnisse der Untersuchung auch aus dem Datensatz folgen [44, S. 154]. Um dies sicherzustellen, wurde die Klassifikation mittels einer 5-fachen Kreuzvalidierung (*5-fold cross validation*) bewertet. Diese verhindert, dass die Ergebnisse ein Zufallsprodukt einer ganz speziellen Auswahl an Sätzen für entweder den Trainings- oder Evaluationsprozess sind. k-fache Kreuzvalidierung minimiert Schwankungen, die durch die Auswahl entstehen können. Die angegebenen Mittelwerte in dieser Arbeit sind daher Ausdruck davon, wie gut der Klassifikator nahezu beliebige ungesehene Sätze aus diesem Textkorpus klassifizieren kann. Aufgrund des noch recht kleinen Textkorpus besteht die Gefahr, dass die Trennlinien der Trainingsdaten überschätzt werden und der Klassifikator nicht ausreichend generalisiert (*overfitting*). Durch Anpassung von Lernrate und Epochenanzahl könnte man diese Gefahr zukünftig weiter minimieren.

Die *externe Validität* beschäftigt sich mit der Frage, inwiefern die Erkenntnisse verallgemeinert werden können und eine Relevanz für andere Anwendungsfälle haben [44, S. 154]. Da der Textkorpus bereits die Softwarearchitekturdokumentation aus 17 verschiedenen Softwareprojekten vereint, die sich auch in ihrer Herkunft, Ausführlichkeit und Schreibart voneinander unterscheiden, ist davon auszugehen, dass die automatisierte Klassifikation auch auf andere Open-Source-Projekte übertragen werden kann. Zudem kann vermutet werden, dass sich die Softwarearchitekturdokumentation von unternehmensinternen Projekten nicht gänzlich von den hier betrachteten Fallstudien unterscheidet, da bei der Fallstudienauswahl auf gute Dokumentationsqualität geachtet wurde. Die hier untersuchten Modellkombinationen geben Hinweise auf möglicherweise vielversprechende Ansätze, können sich aber für andere Datensätze auch unterscheiden. Für eine Verallgemeinerung auf alle Klassen des Schemas kann jedoch nicht sichergestellt werden, ob ähnlich gute Ergebnisse erzielt werden können. Für den bisher vorliegenden Korpus ist von vornherein eher von schlechteren Ergebnissen auszugehen, da die Anzahl der Fallbeispiele für andere Klassen deutlich geringer ist und dadurch womöglich zu wenig Trainingsdaten vorliegen. Bevor eine Erweiterung auf alle Klassen des Schemas vorgenommen wird, sollte daher der Korpus vergrößert werden. Auch eine Klassifikation mit maschinellem Lernen und NLP anhand dieses Schemas für andere Dokumentationsartefakte, beispielsweise Anforderungsspezifikationen oder Mailinglisten, ist grundsätzlich denkbar.

Verlässlichkeit beschreibt den Grad der Unabhängigkeit der Daten und ihrer Analyse von der Person, die die Untersuchung durchgeführt hat [44, S. 154]. Hinsichtlich der automatisierten Analyse der Entwurfsentscheidungen führt die Verwendung einer k-fachen Kreuzvalidierung dazu, dass die gemittelten Daten auch bei einer Durchführung durch andere Personen zu einem sehr ähnlichen Ergebnis führen. Für die Reproduktion der Ergebnisse wurde zudem der von *scikit-learn* [7] zur Verfügung gestellte Parameter *random_state* gesetzt, dessen Festlegung zu einer gleichen Aufspaltung des Korpus in Trainings- und Testdatensatz bei wiederholter Durchführung der k-fachen Kreuzvali-

dierung führt. Im Hinblick auf die Unabhängigkeit der Daten besteht, wie bereits oben erwähnt, die Gefahr, dass einzelne Sätze mit einem unpassenden Label versehen wurden und andere Softwareentwickler andere Klassen von Entwurfsentscheidungen zuordnen würden. Auch die Festlegung, welche Entwurfsentscheidung die Hauptaussage eines Satzes am stärksten widerspiegelt, ist mit einer subjektiven Bewertung behaftet. Möchte man diese Festlegung vermeiden, sollte auf eine Multi-Label-Klassifikation zurückgegriffen werden.

6.4.3. Zukünftige Arbeiten

Die Proof-of-Concept-Implementierung hat gezeigt, dass die automatisierte Identifikation von Entwurfsentscheidungen und ihre Klassifikation anhand des hier entwickelten Klassifikationsschemas einen Anwendungsfall für dieses darstellt. Ansätze des maschinellen Lernens und NLP-Ansätze sind in der Lage, die sprachlichen Trennlinien der Klassen anhand von Trainingsdaten zu lernen und ungesehene Sätze aus Softwarearchitekturdokumentation zu klassifizieren. Um die Leistungsfähigkeit der Klassifikatoren zukünftig zu verbessern und die Signifikanz der erzielten Ergebnisse zu bestimmen, bietet sich eine Erweiterung des Korpus an, sodass mehr Trainingsdaten vorliegen. Insbesondere für Klassen, die bisher eher selten in den Fallstudien identifiziert wurden, ist eine Korpuserweiterung vonnöten. Langfristig ist das Ziel, die Klassifikation anhand des gesamten Schemas durchzuführen und nicht nur ausgewählte Klassen zu betrachten. Dabei kann hierarchisch vorgegangen werden. Insbesondere die Abgrenzung von Sätzen, die Entwurfsentscheidungen enthalten und denen, die keine enthalten, hat sich als sinnvoller Ansatz herausgestellt. Auch für Klassen, die tiefer in der Hierarchie angesiedelt sind, ließen sich Klassifikationen vornehmen. Auch der Abfall der Ergebnisse setzt sich nicht kontinuierlich fort, wenn weiter in die Tiefe des Klassifikationsschemas vorgedrungen wird, sondern kehrt sich für Ebene 3 gegenüber Ebene 2 sogar um.

Neben der Korpuserweiterung ist es sinnvoll, für den bereits bestehenden Korpus die Label durch mindestens eine weitere Person zu setzen und die Interrater-Reliabilität zu bestimmen. Dadurch wird zum einen die Anwendbarkeit und Nachvollziehbarkeit des Schemas evaluiert und zum anderen wird der subjektive Einfluss auf den zugrunde liegenden Textkorpus verkleinert.

Diese Arbeit kann Ausgangspunkt zukünftiger Arbeiten sein, anhand der jeweils zwei bis drei vielversprechendsten Modellkombinationen weitere Verfeinerungen vorzunehmen. Um die Ergebnisse weiter zu verbessern, können insbesondere Modellparameter wie Lernrate, Epochenanzahl und Schwellwerte verfeinert werden. Auch die Erweiterung auf weitere Klassifikationsalgorithmen ist innerhalb der *Scikit-learn*-Bibliothek recht einfach möglich. Mithilfe der *Simpletransformers*-Bibliothek können außerdem andere Transformer-Modelle ausprobiert werden, die insbesondere für die Klassifikation in *Entwurfsentscheidung* und *Keine Entwurfsentscheidung* genutzt werden können.

Auf die Möglichkeit, durch die automatisierte Klassifikation von Entwurfsentscheidungen anhand des entwickelten Klassifikationsschemas eine Konsistenzprüfung zwischen Dokumentations- und Implementationsartefakten vorzunehmen, wird in Kapitel 7 abschließend eingegangen.

7. Fazit und Ausblick

In dieser Bachelorarbeit wurde ein Klassifikationsschema entwickelt, entlang dessen Entwurfsentscheidungen in natürlichsprachiger Softwarearchitekturdokumentation manuell und automatisiert analysiert werden können. Das Klassifikationsschema unterteilt ausgehend von der Ontologie nach Kruchten [27] Entwurfsentscheidungen in hierarchisch gegliederte Klassen. Dabei stellt es eine Erweiterung bisher bestehender Schemata dar, da es in der Lage ist, Entwurfsentscheidungen auch in der hierarchischen Tiefe feingranular zu klassifizieren. Seine Anwendbarkeit wurde durch eine iterative Weiterentwicklung entlang der Softwarearchitekturdokumentation von 17 Fallstudien verbessert, validiert und schlussendlich sichergestellt. Durch das Klassifikationsschema erhalten die Entwurfsentscheidungen eine Repräsentation erster Ordnung, die es ermöglicht, Entwurfsentscheidungen entsprechend ihrer Art und den damit einhergehenden Auswirkungen auf die Architektur eines Softwaresystems voneinander abzugrenzen. Die Identifikation und Klassifikation von Entwurfsentscheidungen ermöglicht es Softwareentwicklern, wichtige Informationen aus der Softwarearchitekturdokumentation zu gewinnen und in den Entwurf und die Implementierung eines Softwaresystems zu überführen. Dabei lieferte die Analyse der Fallstudien einen Einblick darin, welche Arten von Entwurfsentscheidungen wie häufig und in welcher Form dokumentiert werden und wie diese manuell innerhalb der Softwarearchitekturdokumentation identifiziert werden können. Die ausführliche Beschreibung des iterativen Prozesses und die Konkretisierung der Klassen des Schemas ermöglichen die Anwendung des Klassifikationsschemas durch andere Softwareentwickler und -architekten. Die Validität des Klassifikationsschemas wurde gemäß den Prinzipien nach Bedford [4] analysiert.

Die Arbeit hat zudem gezeigt, dass es möglich ist, Entwurfsentscheidungen entlang von ausgewählten Klassen dieses Schemas automatisiert zu klassifizieren. Dafür wurden verschiedene Ansätze des maschinellen Lernens und der natürlichen Sprachverarbeitung (NLP) auf den Korpus, der während der Analyse der Fallstudien erstellt wurde, angewandt. Es zeigte sich, dass für den gegebenen Korpus insbesondere die Textvorverarbeitung mittels Reduktion auf Kleinschreibung und gegebenenfalls auch Stammformreduktion und Lemmatisierung sinnvoll sind. Die Überführung in eine Vektorrepräsentation kann durch das Zählen von Wortvorkommen in Trigrammen zielführend umgesetzt werden. Für die Abgrenzung von Sätzen mit und ohne Entwurfsentscheidungen bietet sich ein Random-Forest-Klassifikator oder insbesondere BERT an, mit dem ein F1-Wert von 0,9082 in einer 5-fachen Kreuzvalidierung erzielt werden konnte. Für die Klassifikation innerhalb der Ebenen des Klassifikationssche-

mas erzielten Logistische Regression und Multinomialer Naiver Bayes mit F1-Werten zwischen 60% und 70% die vielversprechendsten Ergebnisse. Die Rückgewinnung und Klassifikation von Entwurfsentscheidungen ermöglicht es Softwareentwicklern, die für sie relevanten Informationen für die Implementation und Wartung von Sofwaresystemen aus der Dokumentation zu extrahieren. Durch automatisierte Analyse kann eine hohe Dokumentationsqualität bei gleichzeitigen Zeit- und Budgeteinsparungen erreicht werden. Dafür benötigt es einer weiterführenden Entwicklung der hier prototypisch angegangenen Implementierung eines Klassifikators für Entwurfsentscheidungen.

Das in dieser Arbeit entwickelte Klassifikationsschema leistet einen Beitrag dazu, natürliche Sprachverarbeitung auf Architekturdokumentationen anzuwenden. Darauf aufbauend können nun Werkzeuge entwickelt werden, die eine Konsistenzprüfung zwischen Dokumentations- und Sofwareartefakten ermöglichen. Durch seine feingranularen Trennlinien können die Klassen des Schemas in Auswirkungen auf die Softwarearchitektur übersetzt werden und deren konsistente Überführung in Softwareartefakte anschließend überprüft werden.

Literatur

[1] Sallam Abualhaija u. a. „A Machine Learning-Based Approach for Demarcating Requirements in Textual Specifications“. In: *2019 IEEE 27th International Requirements Engineering Conference (RE)*. 2019, S. 51–62. DOI: 10.1109/RE.2019.00017.

[2] ANSI/NISO. *Guidelines for the construction, format, and management of monolingual controlled vocabularies*. ANSI/NISO Z39.19-2005. National Information Standards Organization, Juli 2005.

[3] Adailton Ferreira de Araújo und Ricardo Marcondes Marcacini. „RE-BERT: Automatic Extraction of Software Requirements from App Reviews Using BERT Language Model“. In: *Proceedings of the 36th Annual ACM Symposium on Applied Computing*. SAC '21. Virtual Event, Republic of Korea: Association for Computing Machinery, 2021, S. 1321–1327. DOI: 10.1145/3412841.3442006.

[4] Denise Bedford. „Evaluating classification schema and classification decisions“. In: *Bulletin of the American Society for Information Science and Technology* 39 (Dez. 2013). DOI: 10.1002/bult.2013.1720390206.

[5] Manoj Bhat u. a. „ADeX: A Tool for Automatic Curation of Design Decision Knowledge for Architectural Decision Recommendations“. In: *2019 IEEE International Conference on Software Architecture Companion (ICSA-C)*. 2019, S. 158–161. DOI: 10.1109/ICSA-C.2019.00035.

[6] Manoj Bhat u. a. „Automatic Extraction of Design Decisions from Issue Management Systems: A Machine Learning Based Approach“. In: *Software Architecture*. Cham: Springer International Publishing, 2017, S. 138–154. DOI: 10.1007/978-3-319-65831-5_10.

[7] Lars Buitinck u. a. „API design for machine learning software: experiences from the scikit-learn project“. In: *ECML PKDD Workshop: Languages for Data Mining and Machine Learning*. 2013, S. 108–122.

[8] Jose Camacho-Collados und Mohammad Taher Pilehvar. *On the Role of Text Preprocessing in Neural Network Architectures: An Evaluation Study on Text Categorization and Sentiment Analysis*. 2018. arXiv: 1707.01780 [cs.CL].

[9] Scikit-learn contributors. *MultinomialNB*. Aufgerufen am 21.09.2021. URL: https://scikit-learn.org/stable/modules/generated/sklearn.naive_bayes.MultinomialNB.html.

[10] Jacob Devlin u. a. *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding*. 2019. arXiv: 1810.04805 [cs.CL].

[11] Wei Ding u. a. „How Do Open Source Communities Document Software Architecture: An Exploratory Survey“. In: Aug. 2014. DOI: 10.1109/ICECCS.2014.26.

[12] Davide Falessi, Martin Becker und Giovanni Cantone. „Design decision rationale: experiences and steps ahead towards systematic use“. In: *ACM SIGSOFT Software Engineering Notes* 31 (Sep. 2006). DOI: 10.1145/1163514.1178642.

[13] Peter Fischer und Peter Hofer. *Lexikon der Informatik*. Heidelberg: Springer, 2011. DOI: 10.1007/978-3-642-15126-2.

[14] Martin Glinz. „On Non-Functional Requirements“. In: Nov. 2007, S. 21–26. DOI: 10.1109/RE.2007.45.

[15] Boris Golden. „A unified formalism for complex systems architecture“. In: 2013.

[16] Santiago González-Carvajal und Eduardo C. Garrido-Merchán. *Comparing BERT against traditional machine learning text classification*. 2021. arXiv: 2005.13012 [cs.CL].

[17] Vairaprakash Gurusamy und Subbu Kannan. „Preprocessing Techniques for Text Mining“. In: Okt. 2014.

[18] Jeremy Howard und Sebastian Ruder. *Universal Language Model Fine-tuning for Text Classification*. 2018. arXiv: 1801.06146 [cs.CL].

[19] Purva Huilgol. *Accuracy vs. F1-Score*. Aufgerufen am 23.09.2021. 2019. URL: https://medium.com/analytics-vidhya/accuracy-vs-f1-score-6258237beca2.

[20] IEEE. „Recommended practice for architectural description of software intensive systems“. In: *IEEE Std. 1471:2000* (2000). DOI: 10.1109/IEEESTD.2000.91944.

[21] Anne Immonen und Eila Niemelä. „Survey of reliability and availability prediction methods from the viewpoint of software architecture“. In: *Software & Systems Modeling* 7.1 (Jan. 2007), S. 49. DOI: 10.1007/s10270-006-0040-x.

[22] A. Jansen und J. Bosch. „Software Architecture as a Set of Architectural Design Decisions“. In: *5th Working IEEE/IFIP Conference on Software Architecture (WICSA'05)*. 2005, S. 109–120. DOI: 10.1109/WICSA.2005.61.

[23] Anton Jansen, Jan Bosch und Paris Avgeriou. „Documenting after the fact: Recovering architectural design decisions“. In: *Journal of Systems and Software* 81.4 (2008), S. 536–557. DOI: 10.1016/j.jss.2007.08.025.

[24] Nishant Jha und Anas Mahmoud. „Mining non-functional requirements from App store reviews“. In: *Empirical Software Engineering* 24 (Dez. 2019), S. 1–37. DOI: 10.1007/s10664-019-09716-7.

[25] Taeho Jo. *Machine Learning Foundations: Supervised, Unsupervised, and Advanced Learning*. Springer International Publishing, 2021. ISBN: 9783030659004.

[26] Jan Keim, Yves Schneider und Anne Koziolek. „Towards Consistency Analysis between Formal and Informal Software Architecture Artefacts“. In: Mai 2019, S. 6–12. DOI: 10.1109/ECASE.2019.00010.

[27] Philippe Kruchten. „An Ontology of Architectural Design Decisions in Software-Intensive Systems“. In: *2nd Groningen Workshop on Software Variability* (2004), S. 54–61.

[28] Philippe Kruchten, Patricia Lago und Hans Vliet. „Building Up and Reasoning About Architectural Knowledge“. In: Bd. 4214. Dez. 2006, S. 43–58. DOI: 10.1007/11921998_8.

[29] Xueying Li, Peng Liang und Zengyang Li. „Automatic Identification of Decisions from the Hibernate Developer Mailing List“. In: Feb. 2020. DOI: 10.1145/3383219.3383225.

[30] Andrew Luashchuk. *Why I Think Python is Perfect for Machine Learning and Artificial Intelligence*. Aufgerufen am 23.09.2021. 2019. URL: https://towardsdatascience.com/8-reasons-why-python-is-good-for-artificial-intelligence-and-machine-learning-4a23f6bed2e6.

[31] Object Management Group. *OMG Unified Modeling Language – Version 2.5.1*. https://www.omg.org/spec/UML/2.5.1. Dez. 2017. URL: https://www.omg.org/spec/UML/2.5.1.

[32] Robert C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*. 1. Aufl. USA: Prentice Hall PTR, 2008. DOI: 10.5555/1388398.

[33] Cornelia Miesbauer und Rainer Weinreich. „Classification of Design Decisions – An Expert Survey in Practice“. In: *Software Architecture*. Hrsg. von Khalil Drira. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, S. 130–145.

[34] Tomas Mikolov u. a. *Efficient Estimation of Word Representations in Vector Space*. 2013. arXiv: 1301.3781 [cs.CL].

[35] Ayush Pant. *Introduction to Logistic Regression*. Aufgerufen am 21.09.2021. URL: https://towardsdatascience.com/introduction-to-logistic-regression-66248243c148.

[36] David Parnas. „Precise Documentation: The Key to Better Software“. In: *Nanz S. (ebs) The Future of Software Engineering*. Berlin, Heidelberg: Springer, 2011. DOI: 10.1007/978-3-642-151787-3_8.

[37] F. Pedregosa u. a. „Scikit-learn: Machine Learning in Python“. In: *Journal of Machine Learning Research* 12 (2011), S. 2825–2830.

[38] Matthew E. Peters u. a. „Deep Contextualized Word Representations“. In: *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers)*. New Orleans, Louisiana: Association for Computational Linguistics, Juni 2018, S. 2227–2237. DOI: 10.18653/v1/N18-1202. URL: https://aclanthology.org/N18-1202.

[39] Tomas Pranckevicius und Virginijus Marcinkevičius. „Comparison of Naive Bayes, Random Forest, Decision Tree, Support Vector Machines, and Logistic Regression Classifiers for Text Reviews Classification“. In: *Baltic Journal of Modern Computing* 5 (Jan. 2017). DOI: 10.22364/bjmc.2017.5.2.05.

[40] Thilina Rajapakse. *Simple Transformers Documentation - Classification Models*. Aufgerufen am 25.09.2021. URL: https://simpletransformers.ai/docs/classification-models/.

[41] Juan Enrique Ramos. „Using TF-IDF to Determine Word Relevance in Document Queries“. In: 2003.

[42] Susmita Ray. „A Quick Review of Machine Learning Algorithms“. In: *2019 International Conference on Machine Learning, Big Data, Cloud and Parallel Computing (COMITCon)*. 2019, S. 35–39. DOI: 10.1109/COMITCon.2019.8862451.

[43] Ralf Reussner und Wilhelm Hasselbring. *Handbuch der Software-Architektur*. 2. Aufl. Heidelberg: dpunkt, 2008.

[44] Per Runeson und Martin Höst. „Guidelines for conducting and reporting case study research in software engineering“. In: *Empirical Software Engineering* 14 (2008), S. 131–164.

[45] Arman Shahbazian u. a. „Recovering Architectural Design Decisions“. In: *Proceedings of the IEEE International Conference on Software Architecture (ICSA)*. 2018, S. 95–104. DOI: 10.1109/ICSA.2018.00019.

[46] Mojtaba Shahin, Peng Liang und Mohammad Khayyambashi. „Architectural Design Decision: Existing Models and Tools“. In: Sep. 2009, S. 293–296. DOI: 10.1109/WICSA.2009.5290823.

[47] Kevin J. Sullivan u. a. „The Structure and Value of Modularity in Software Design“. In: *SIGSOFT Softw. Eng. Notes* 26.5 (Sep. 2001), S. 99–108. DOI: 10.1145/503271.503224.

[48] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. 2nd. USA: Addison-Wesley Longman Publishing Co., Inc., 2002. ISBN: 0201745720.

[49] Ashish Vaswani u. a. *Attention Is All You Need*. 2017. arXiv: 1706.03762 [cs.CL].

[50] Jan Salvador van der Ven u. a. „Design Decisions: The Bridge between Rationale and Architecture“. In: *Rationale Management in Software Engineering*. Hrsg. von Allen H. Dutoit u. a. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, S. 329–348. DOI: 10.1007/978-3-540-30998-7_16.

[51] Christina Voskoglou. *What is the best programming language for Machine Learning?* Aufgerufen am 23.09.2021. 2017. URL: https://towardsdatascience.com/what-is-the-best-programming-language-for-machine-learning-a745c156d6b7.

[52] Wikipedia. *Data file*. Aufgerufen am: 04.08.2021. URL: https://en.wikipedia.org/wiki/Data_file.

[53] Wikipedia. *Gradle*. Aufgerufen am: 25.08.2021. URL: https://de.wikipedia.org/wiki/Gradle.

[54] Wikipedia. *Nachrichtenaustausch*. Aufgerufen am: 03.08.2021. URL: https://de.wikipedia.org/wiki/Nachrichtenaustausch.

[55] Wikipedia. *Referenzarchitektur*. Aufgerufen am: 25.08.2021. URL: https://de.wikipedia.org/wiki/Referenzarchitektur.

[56] Rebekka Wohlrab u. a. „Improving the Consistency and Usefulness of Architecture Descriptions: Guidelines for Architects“. In: *2019 IEEE International Conference on Software Architecture (ICSA)*. 2019, S. 151–160. DOI: 10.1109/ICSA.2019.00024.

[57] Thomas Wolf u. a. *HuggingFace's Transformers: State-of-the-art Natural Language Processing*. 2020. arXiv: 1910.03771 [cs.CL].

[58] Wen Zhang, Taketoshi Yoshida und Xijin Tang. „TFIDF, LSI and multi-word in information retrieval and text categorization“. In: *2008 IEEE International Conference on Systems, Man and Cybernetics*. 2008, S. 108–113. DOI: 10.1109/ICSMC.2008.4811259.

[59] Olaf Zimmermann u. a. „N.: Reusable Architectural Decision Models for Enterprise Application Development“. In: Juli 2007. DOI: 10.1007/978-3-540-77619-2_2.

A. Anhang

Die folgenden Auswertungstabellen zeigen die Einzelergebnisse für die in Kapitel 6 beschriebene Proof-of-Concept-Implementierung. Während innerhalb der Arbeit häufig auf maximale oder durchschnittliche Werte zurückgegriffen wurde, um Erkenntnisse zu verallgemeinern, lassen sich an dieser Stelle die Ergebnisse der 5-fachen Kreuzvalidierung für eine bestimmte Kombination aus Textvorverarbeitung, Überführung in Vektorrepräsentation und angewandter Klassifikationsalgorithmus auf den verschiedenen Ebenen einsehen (siehe Kapitel 6).

Tabelle A.1.: Vergleich KKR und F1-Werte für Vektorisierer-Lernalgorithmus-Kombinationen auf Ebene 0 für keine Textvorverarbeitung

KKR	**BoW**	**Bigramm**	**Trigramm**	**tf-idf**
LR	0,8395	0,7984	0,8348	0,7861
MNB	0,7982	0,7998	0,8233	0,7668
DT	0,7653	0,7571	0,7754	0,7575
RF	0,8177	0,8323	0,8340	0,8243
SVM	0,8161	0,8157	0,8276	0,8019

F1-Wert	**BoW**	**Bigramm**	**Trigramm**	**tf-idf**
LR	0,8937	0,8623	0,8886	0,8721
MNB	0,8760	0,8705	0,8867	0,8627
DT	0,8400	0,8358	0,8473	0,8333
RF	0,8819	0,8947	0,8952	0,8864
SVM	0,8809	0,8819	0,8907	0,8799

Tabelle A.2.: Vergleich KKR und F1-Werte für Vektorisierer-Lernalgorithmus-Kombinationen auf Ebene 0 für Reduktion auf Kleinschreibung

KKR	**BoW**	**Bigramm**	**Trigramm**	**tf-idf**
LR	0,8395	0,7976	0,8362	0,7861
MNB	0,7982	0,7994	0,8237	0,7668
DT	0,7694	0,7485	0,7801	0,7614
RF	0,8212	0,8352	0,8381	0,8245
SVM	0,8161	0,8157	0,8276	0,8019

F1-Wert	**BoW**	**Bigramm**	**Trigramm**	**tf-idf**
LR	0,8937	0,8618	0,8896	0,8721
MNB	0,8760	0,8702	0,8869	0,8627
DT	0,8432	0,8292	0,8502	0,8358
RF	0,8841	0,8963	0,8978	0,8865
SVM	0,8809	0,8818	0,8906	0,8799

Tabelle A.3.: Vergleich KKR und F1-Werte für Vektorisierer-Lernalgorithmus-Kombinationen auf Ebene 0 für Stammformreduktion/Lemmatisierung

KKR	**BoW**	**Bigramm**	**Trigramm**	**tf-idf**
LR	0,8335	0,7980	0,8313	0,7898
MNB	0,7914	0,7970	0,8181	0,7659
DT	0,7723	0,7503	0,7665	0,7620
RF	0,8223	0,8366	0,8362	0,8239
SVM	0,8155	0,8200	0,8233	0,8068

F1-Wert	**BoW**	**Bigramm**	**Trigramm**	**tf-idf**
LR	0,8893	0,8625	0,8865	0,8737
MNB	0,8712	0,8685	0,8836	0,8623
DT	0,8454	0,8298	0,8414	0,8371
RF	0,8854	0,8971	0,8965	0,8865
SVM	0,8798	0,8850	0,8878	0,8827

Tabelle A.4.: Vergleich KKR und F1-Werte für Vektorisierer-Lernalgorithmus-Kombinationen auf Ebene 0 für Textaufbereitung hinsichtlich Sonderzeichen, Ziffern und Hyperlinks

KKR	**BoW**	**Bigramm**	**Trigramm**	**tf-idf**
LR	0,8335	0,7762	0,8179	0,7842
MNB	0,7970	0,7368	0,7838	0,7659
DT	0,7684	0,7273	0,7507	0,7698
RF	0,8163	0,8253	0,8294	0,8196
SVM	0,8099	0,804	0,8114	0,8031

F1-Wert	**BoW**	**Bigramm**	**Trigramm**	**tf-idf**
LR	0,8894	0,8473	0,8768	0,871
MNB	0,8752	0,8300	0,8610	0,8623
DT	0,8419	0,8134	0,8291	0,8415
RF	0,8805	0,8905	0,8924	0,8831
SVM	0,8763	0,8744	0,8775	0,8804

Tabelle A.5.: Vergleich KKR und F1-Werte für Vektorisierer-Lernalgorithmus-Kombinationen auf Ebene 0 für Entfernung von Stoppwörtern

KKR	**BoW**	**Bigramm**	**Trigramm**	**tf-idf**
LR	0,8173	0,8155	0,8350	0,7787
MNB	0,7949	0,8023	0,8198	0,7723
DT	0,7483	0,7575	0,7783	0,7382
RF	0,7885	0,8358	0,8313	0,8085
SVM	0,8165	0,8167	0,8239	0,7957

F1-Wert	**BoW**	**Bigramm**	**Trigramm**	**tf-idf**
LR	0,8808	0,8749	0,8893	0,8686
MNB	0,8722	0,8725	0,8847	0,8655
DT	0,8225	0,8351	0,8489	0,8489
RF	0,8536	0,8968	0,8935	0,8703
SVM	0,8833	0,8832	0,8889	0,8765

Tabelle A.6.: Vergleich KKR und F1-Werte für Vektorisierer-Lernalgorithmus-Kombinationen auf Ebene 0 für Reduktion auf Kleinschreibung & Stammformreduktion/Lemmatisierung

KKR	**BoW**	**Bigramm**	**Trigramm**	**tf-idf**
LR	0,8344	0,7986	0,8319	0,7898
MNB	0,7914	0,7974	0,8194	0,7659
DT	0,7760	0,7421	0,7663	0,7631
RF	0,8241	0,8338	0,8373	0,8249
SVM	0,8157	0,8200	0,8239	0,8068

F1-Wert	**BoW**	**Bigramm**	**Trigramm**	**tf-idf**
LR	0,8899	0,863	0,8868	0,8737
MNB	0,8712	0,8689	0,8843	0,8623
DT	0,8483	0,8234	0,8415	0,8375
RF	0,8864	0,8954	0,8969	0,8871
SVM	0,8799	0,8850	0,8880	0,8827

Tabelle A.7.: Vergleich KKR und F1-Werte für Vektorisierer-Lernalgorithmus-Kombinationen auf Ebene 1 (jeweils maximal erzielter Wert für verschiedene Methoden zur Textvorverarbeitung)

KKR	**BoW**	**Bigramm**	**Trigramm**	**tf-idf**
LR	0,6704	0,6291	0,6675	0,6632
MNB	0,6375	0,6624	0,6661	0,5802
DT	0,5752	0,5436	0,5795	0,5670
RF	0,6422	0,6467	0,6554	0,6424
SVM	0,6260	0,6227	0,6389	0,6387

F1-Wert	**BoW**	**Bigramm**	**Trigramm**	**tf-idf**
LR	0,6532	0,6266	0,6559	0,6190
MNB	0,6045	0,6106	0,6474	0,4777
DT	0,5650	0,5412	0,5745	0,5607
RF	0,5924	0,5837	0,6025	0,5928
SVM	0,5669	0,5523	0,5785	0,5737

Tabelle A.8.: Vergleich KKR und F1-Werte für Vektorisierer-Lernalgorithmus-Kombinationen auf Ebene 2 (jeweils maximal erzielter Wert für verschiedene Methoden zur Textvorverarbeitung)

KKR	**BoW**	**Bigramm**	**Trigramm**	**tf-idf**
LR	0,6215	0,5949	0,6233	0,6249
MNB	0,6321	0,5532	0,6274	0,5859
DT	0,5497	0,4873	0,5489	0,5384
RF	0,6122	0,6069	0,6461	0,6056
SVM	0,5824	0,5921	0,6175	0,6040

F1-Wert	**BoW**	**Bigramm**	**Trigramm**	**tf-idf**
LR	0,6024	0,589	0,6154	0,5908
MNB	0,6172	0,5510	0,6228	0,5193
DT	0,5417	0,4856	0,5483	0,5328
RF	0,5750	0,5643	0,6193	0,5669
SVM	0,5224	0,5467	0,5761	0,5535

Tabelle A.9.: Vergleich KKR und F1-Werte für Vektorisierer-Lernalgorithmus-Kombinationen auf Ebene 3 (jeweils maximal erzielter Wert für verschiedene Methoden zur Textvorverarbeitung)

KKR	**BoW**	**Bigramm**	**Trigramm**	**tf-idf**
LR	0,6774	0,6519	0,6813	0,6558
MNB	0,6854	0,6058	0,6810	0,6243
DT	0,6083	0,5411	0,5810	0,5878
RF	0,6572	0,6479	0,6619	0,6535
SVM	0,6356	0,6449	0,6533	0,6461

F1-Wert	**BoW**	**Bigramm**	**Trigramm**	**tf-idf**
LR	0,6468	0,6451	0,6685	0,5816
MNB	0,6502	0,6169	0,6767	0,498
DT	0,5972	0,5419	0,5800	0,5799
RF	0,5916	0,5597	0,5997	0,5768
SVM	0,5246	0,5501	0,5684	0,549

Tabelle A.10.: Vergleich KKR und F1-Werte für Vektorisierer-Lernalgorithmus-Kombinationen auf Ebene 1 unter Verwendung von Multi-Label-Klassifikation (Reduktion auf Kleinschreibung)

KKR	**BoW**	**Bigramm**	**Trigramm**	**tf-idf**
LR	0,5170	0,4681	0,5113	0,4127
MNB	0,4844	0,4628	0,5060	0,4120
DT	0,4009	0,3331	0,3841	0,3825
RF	0,4870	0,4881	0,5148	0,4755
SVM	0,4496	0,4501	0,4698	0,4373

F1-Wert	**BoW**	**Bigramm**	**Trigramm**	**tf-idf**
LR	0,6169	0,5925	0,6279	0,4700
MNB	0,5637	0,5795	0,6193	0,4158
DT	0,5680	0,5241	0,5639	0,5636
RF	0,5769	0,5418	0,5876	0,5597
SVM	0,5034	0,4929	0,5262	0,4917

Tabelle A.11.: Vergleich KKR und F1-Werte für Vektorisierer-Lernalgorithmus-Kombinationen auf Ebene 2 unter Verwendung von Multi-Label-Klassifikation (Reduktion auf Kleinschreibung)

KKR	**BoW**	**Bigramm**	**Trigramm**	**tf-idf**
LR	0,5294	0,4799	0,5212	0,4486
MNB	0,5037	0,4338	0,4961	0,4116
DT	0,4001	0,3220	0,3691	0,3740
RF	0,4893	0,4262	0,4821	0,4673
SVM	0,4408	0,4252	0,4757	0,4569

F1-Wert	**BoW**	**Bigramm**	**Trigramm**	**tf-idf**
LR	0,5999	0,5879	0,6175	0,4847
MNB	0,6084	0,5791	0,6255	0,4444
DT	0,5581	0,5249	0,5549	0,5553
RF	0,5465	0,5018	0,5737	0,5288
SVM	0,4618	0,4555	0,5198	0,5011

Tabelle A.12.: Vergleich KKR und F1-Werte für Vektorisierer-Lernalgorithmus-Kombinationen auf Ebene 3 unter Verwendung von Multi-Label-Klassifikation (Reduktion auf Kleinschreibung)

KKR	**BoW**	**Bigramm**	**Trigramm**	**tf-idf**
LR	0,6112	0,5655	0,5917	0,6097
MNB	0,5863	0,4729	0,5573	0,6032
DT	0,4908	0,3981	0,4572	0,4566
RF	0,6038	0,6034	0,5914	0,6067
SVM	0,6171	0,5945	0,6138	0,6159

F1-Wert	**BoW**	**Bigramm**	**Trigramm**	**tf-idf**
LR	0,6551	0,6582	0,6781	0,5592
MNB	0,6455	0,6327	0,6730	0,5528
DT	0,6213	0,5788	0,6132	0,6106
RF	0,5868	0,5670	0,5997	0,5745
SVM	0,5517	0,5406	0,5771	0,5638