Robert Brčina

Zielorientierte Erkennung und Behebung von Qualitätsdefiziten in Software-Systemen am Beispiel der Weiterentwicklungsfähigkeit

Zielorientierte Erkennung und Behebung von Qualitätsdefiziten in Software-Systemen am Beispiel der Weiterentwicklungsfähigkeit

Robert Brčina

Universitätsverlag Ilmenau
2012

Impressum

Bibliografische Information der Deutschen Nationalbibliothek
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Angaben sind im Internet über http://dnb.d-nb.de abrufbar.

Diese Arbeit hat der Fakultät für Informatik und Automatisierung der Technischen Universität Ilmenau als Dissertation vorgelegen.

Tag der Einreichung:	26. April 2011
1. Gutachter:	Priv.-Doz. Dr.-Ing. habil. Matthias Riebisch (Technische Universität Ilmenau)
2. Gutachter:	Prof. Dr.-Ing. habil. Armin Zimmermann (Technische Universität Ilmenau)
3. Gutachter:	Prof. Dr. rar. nat. Ulrich Bröckl (Hochschule Karlsruhe – Technik und Wirtschaft)
Tag der Verteidigung:	13. Oktober 2011

Technische Universität Ilmenau/Universitätsbibliothek
Universitätsverlag Ilmenau
Postfach 10 05 65
98684 Ilmenau
www.tu-ilmenau.de/universitaetsverlag

Herstellung und Auslieferung
Verlagshaus Monsenstein und Vannerdat OHG
Am Hawerkamp 31
48155 Münster
www.mv-verlag.de

ISBN 978-3-86360-019-8 (Druckausgabe)
URN urn:nbn:de:gbv:ilm1-2012100050

Titelfoto: photocase.com

Danksagung

Mein erster und ganz besonderer Dank richtet sich an meinen Doktorvater Priv.-Doz. Dr.-Ing. habil. Matthias Riebisch. Ich danke für seine scheinbar unendliche Geduld und die vielen wertvollen Diskussionen, die mich immer weitergebracht haben. Ich hatte nie das Gefühl zu stören und er hatte immer ein offenes Ohr für meine Sorgen und Nöte. Durch seine direkte Art und die Fähigkeit die richtigen Fragen zu stellen, hat er mich dazu gebracht, das Gedachte mit Hilfe der richtigen Worte zu formulieren. Danke für die jahrelange Begleitung auf meinem Weg zum Wissenschaftler.

Univ.-Prof. Dr.-Ing. habil. Armin Zimmermann danke ich dafür, dass er das Zweitgutachten übernommen hat. Insbesondere danke ich für die detaillierten Korrekturvorschläge und die vielen Hinweise zur Verbesserung meiner Arbeit.

Prof. Dr. Ulrich Bröckl danke ich für die Übernahme des Drittgutachtens und insbesondere für seine jahrelange Begleitung auf meinem Weg. Er hat vor ca. 10 Jahren bereits meine erste Studienarbeit und im Anschluss meine Diplomarbeit betreut. Ich danke für die vielen konstruktiven Kritiken und Vorschläge zur Verbesserung meiner Arbeit. Zudem habe ich sehr von seinen persönlichen Ratschlägen und von seiner Fähigkeit profitiert, mich als Mensch mit eigenen Ideen und Ansätzen anzusehen.

Prof. Dr. Manfred Dausmann danke ich für die vielen Korrekturvorschläge und Diskussionen im schwierigen letzten Abschnitt meiner Promotion. Insbesondere hat er mir durch seine direkte und freundliche Art neue Hoffnung und Zuversicht gegeben können.

Ich danke der Firma IT-Designers GmbH für die Unterstützung bei der Projektarbeit und bei der Fallstudie. Zudem danke ich insbesondere Prof. Dr. Joachim Goll für seine Unterstützung während der letzten Jahre und die zahlreichen Diskussionen innerhalb der Doktoranden-Workshops.

Insbesondere danke ich Dr. Markus Prechtel für die vielen Diskussionen und Ratschläge bezüglich jeder Art von formalen Formulierungen und seine positiven und motivierenden Worte während der letzten Jahre.

Zum Schluss danke ich meiner Frau Helma für ihre Unterstützung und ihr Einfühlvermögen während der letzten Jahre. Ohne sie wäre das ganze Unterfangen nicht möglich gewesen. Sie hat mich immer ermuntert und mir Mut gemacht, gerade dann, als ich es am Nötigsten hatte. Zudem danke ich meinen Eltern Cvija und Petar, und meinem Bruder Marijan für die Unterstützung während der ganzen Jahre; sie haben im Gegensatz zu mir nie an meinem Erfolg gezweifelt.

Stuttgart, im Oktober 2011 Robert Brčina

Kurzfassung

Für unternehmenskritische Software-Systeme, die langlebig und erweiterbar sein sollen, ist das Qualitätsziel Weiterentwicklungsfähigkeit essentiell. Kontinuierliche Änderungen und Erweiterungen sind unabdingbar, um solche Software-Systeme an neue oder veränderte Anforderungen anzupassen. Diese Maßnahmen verursachen aber auch oft Qualitätsdefizite, die zu einem Anstieg der Komplexität oder einem Verfall der Architektur führen können. Gerade Qualitätsdefizite in der Spezifikation oder Architektur können Software-Systeme stark beeinträchtigen.

Um dem entgegenzuwirken, wird in dieser Arbeit eine Methode entwickelt, welche die Einhaltung von Qualitätszielen bewerten kann. Dadurch wird sowohl das Erkennen als auch das Beheben von Qualitätsdefiziten in der Software-Entwicklung ermöglicht. Qualitätsdefizite werden anhand einer am Qualitätsziel orientierten und regelbasierten Analyse erkannt und durch zugeordnete Reengineering-Aktivitäten behoben. Als Beispiel für ein Qualitätsziel wird die Weiterentwicklungsfähigkeit von Software-Systemen betrachtet. Es wird gezeigt, wie dieses Qualitätsziel anhand von strukturellen Abhängigkeiten in Software-Systemen bewertet und durch gezielte Reengineering-Aktivitäten verbessert werden kann.

Um die Methode zu validieren, wurde eine industrielle Fallstudie durchgeführt. Durch den Einsatz der Methode konnten eine Vielzahl von Qualitätsdefiziten erkannt und behoben werden. Die Weiterentwicklungsfähigkeit des untersuchten Software-Systems wurde durch die vorgeschlagenen Reengineering-Aktivitäten entscheidend verbessert.

Abstract

The evolvability of software systems is one of the key issues when considering their long term quality. Continuous changes and extensions of these systems are neccessary to adjust them to new or changing requirements. But the changes often cause quality deficiencies, which lead to an increase in complexity or an architectural decay. Especially quality deficiencies within the specification or the architecture of a software system can heavily impair a software system.

To counteract this, a method is developed in this work to support the analysis of a quality goal in order to identify the quality deficiencies which hinder the achievement of the quality goal. Both the detection and the removal of quality deficiencies are accomplished in a systematic way. The method integrates detection of these quality deficiencies and their removal by reengineering activities based on rules. The detection of quality deficiencies is performed by means of measurable quality attributes which are derived from a quality goal, such as evolvability.

In order to demonstrate the practicability of the method, the quality goal evolvability is taken as an example. This work shows how a software system can be evaluated with regard to evolvability based on structural dependencies and which reengineering activities will improve the system in the direction of this quality goal.

To evaluate the method, it was applied within an industrial case study. By analyzing the given software system a large number of different quality deficiencies were detected. Afterwards the system's evolvability was improved substantially by reengineering activities proposed by the method.

Abkürzungs- und Symbolverzeichnis

Abkürzungen

FCM	Factor Criteria Metrics
FDD	Feature Driven Development
DSL	Domain Specific Language
GQM	Goal Question Metric
KLOC	Thousands of Lines of Code
SLOC	Source Lines of Code
SIG	Softgoal Interpendency Graph
OML	OPEN Modeling Language
OOSE	Object Oriented Software Engineering
OPEN	Object Oriented Process, Environment, and Notation
QMOOD	Quality Model of Object Oriented Design
RUP	Rational Unified Prozess
UML	Unified Modeling Language

Symbole

$\xrightarrow{typ}$	Beziehung mit Beziehungstyp
$\xrightarrow{i}$	Implementiert-Beziehung
$\xrightarrow{b}$	Benutzt-Beziehung (engl. Usage-Beziehung)
$\xrightarrow{t}$	Teil-von-Beziehung
$\xrightarrow{r}$	Realisiert-durch-Beziehung
$\xrightarrow{e}$	Erweitert-Beziehung

Lateinische Buchstaben

E	*Entitäten* eines Software-Systems
F	Entitätstyp *Feature* eines Software-Systems
A	Entitätstyp *Architekturkomponente* eines Software-Systems
S	Entitätstyp *Schnittstelle* eines Software-Systems
K	Entitätstyp *Klasse* eines Software-Systems
D	Entitätstyp *Datendatei* (zur Konfiguration) eines Software-Systems

Inhaltsverzeichnis

Danksagung v

Kurzfassung vii

Abstract ix

Abkürzungs- und Symbolverzeichnis xi

1 Einleitung 1

1.1 Motivation . . . 3
1.2 Ziele der Arbeit . . . 4
1.3 Gliederung der Arbeit . . . 5

2 Bewertung des Stands der Technik 7

2.1 Analyse von Qualitätszielen . . . 8
2.2 Erkennung von Qualitätsdefiziten . . . 10
2.3 Beheben von Qualitätsdefiziten . . . 13
2.4 Von der Erkennung zur Behebung von Qualitätsdefiziten . . . 14
2.5 Analyse des Qualitätsziels Weiterentwicklungsfähigkeit . . . 14
2.6 Abhängigkeitsanalyse und Verfolgbarkeit . . . 16
2.7 Zusammenfassung und präzisierte Zielstellung . . . 17

3 Überblick über die Methode 21

3.1 Aufbauprozess . . . 23
3.1.1 Analyse eines Qualitätsziels . . . 24
3.1.2 Erstellen von Reengineering-Rezepten . . . 25
3.1.3 Der Zuordnungsgraph – ein Ergebnis des Aufbauprozesses . . . 27
3.2 Anwendungsprozess . . . 31

3.2.1 Priorisieren der Qualitätsteilziele, Qualitätsmerkmale und Qualitätsteilmerkmale 32
3.2.2 Auswählen und Anwenden der Reengineering-Rezepte 32
3.2.3 Prüfung der Erfüllung der Qualitätsziele 34
3.3 Zusammenfassung 35

4 Analyse eines Qualitätsziels 37
4.1 Identifizieren der Qualitätsteilziele ausgehend von einem Qualitätsziel 38
4.1.1 Ziel und Vorgehen der Aktivität 38
4.1.2 Ergebnisse des Identifizierens von Qualitätsteilzielen am Beispiel der Weiterentwicklungsfähigkeit 39
4.2 Verfeinern zu Qualitätsmerkmalen 42
4.2.1 Ziel und Vorgehen der Aktivität 42
4.2.2 Ergebnisse des Verfeinerns von Qualitätsteilzielen am Beispiel der Weiterentwicklungsfähigkeit 42
4.3 Verfeinern zu Qualitätsteilmerkmalen 51
4.3.1 Ziel und Vorgehen der Aktivität 51
4.3.2 Ergebnisse des Verfeinerns von Qualitätsmerkmalen am Beispiel der Weiterentwicklungsfähigkeit 51
4.4 Zusammenfassung 58

5 Erstellen der Reengineering-Rezepte 61
5.1 Ermitteln der Qualitätsdefizite 62
5.1.1 Ziel und Vorgehen der Aktivität 62
5.1.2 Ergebnisse des Ermittelns von Qualitätsdefiziten am Beispiel der Weiterentwicklungsfähigkeit 64
5.2 Auffinden der Metriken und Aufstellen der Erkennungsregeln 66
5.2.1 Ziel und Vorgehen der Aktivität 66
5.2.2 Auffinden von Metriken 66
5.2.3 Aufstellen von Erkennungsregeln 69
5.2.4 Ergebnisse der aufgestellten Erkennungsregeln am Beispiel der Weiterentwicklungsfähigkeit 71
5.3 Bestimmen der Reengineering-Regeln 75
5.3.1 Ziel und Vorgehen der Aktivität 75
5.3.2 Struktur von Reengineering-Regeln 76
5.3.3 Bestimmen von Reengineering-Aktivitäten 78

5.3.4 Beispiel einer Reengineering-Regel 83
5.3.5 Ergebnisse der bestimmten Reengineering-Regeln am Beispiel der Weiterentwicklungsfähigkeit 84
5.4 Zusammenfassen zum Reengineering Rezept 91
5.4.1 Struktur von Reengineering-Rezepten 92
5.4.2 Beispiel eines Reengineering-Rezepts 93
5.4.3 Reengineering-Rezepte am Beispiel der Weiterentwicklungsfähigkeit 95

6 Automatisierung der Methode 101
6.1 Übersicht über das TraceTool 101
6.2 Architektur des Werkzeugs 102
6.3 Automatische Erkennung von Qualitätsdefiziten 105
6.4 Aufbauen von Abhängigkeitsdaten 106
6.5 Zusammenfassung 108

7 Validierung 111
7.1 Überblick über die Fallstudie 112
7.2 Priorisieren der Qualitätsteilziele, -merkmale und -teilmerkmale . . . 114
7.3 Auswählen und Anwenden der Reengineering-Rezepte 116
7.3.1 Anwenden des Reengineering-Rezepts Verbindliche Spezifikation 118
7.3.2 Anwenden des Reengineering-Rezepts Spezifizierte Implementierung 120
7.3.3 Anwenden des Reengineering-Rezepts Feature-Entschränkung 122
7.3.4 Anwenden des Reengineering-Rezepts Zyklenfreie Struktur 129
7.4 Zusammenfassung der Ergebnisse 132

8 Fazit und Ausblick 137
8.1 Ergebnisse 137
8.2 Ausblick 139

A Reengineering-Rezepte 143
A.1 Qualitätsdefizite 143
A.2 Metriken 148

A.3 Reengineering-Aktivitäten 149
A.4 Reengineering-Rezepte 152

B Abhängigkeitsbeziehungen im Werkzeug TraceTool 161
B.1 Struktur von Abhängigkeitsbeziehungen 161
B.2 Beziehungen im Abhängigkeitsmodell 165
B.2.1 Benutzt-Beziehung 166
B.2.2 Teil-von-Beziehung 172
B.2.3 Realisiert-durch-Beziehung 172
B.2.4 Zusammenfassung 174

C Erarbeitete Metriken 177
C.1 Definition von Metriken 177
C.2 Vorstellung der erarbeiteten Metriken 178
C.2.1 Isolierte Entitäten 178
C.2.2 Feature Scattering 180
C.2.3 Feature Tangling 181
C.2.4 Feature-Größe 182

D Tabellen der Messergebnisse 185

Literaturverzeichnis 195

Erklärung 209

Thesen 211

Kapitel 1

Einleitung

Tom DeMarco unterstreicht mit seiner Aussage *You can't control what you can't measure* [DeM86] die Notwendigkeit der Kontrolle und des Steuerns von Qualitätszielen innerhalb eines Software-Entwicklungsprozesses durch geeignete Messmethoden. Für moderne Software-Systeme, die langlebig und erweiterbar sein sollen, ist das Qualitätsziel Weiterentwicklungsfähigkeit essentiell. Insbesondere unternehmenskritische Systeme werden unter anderem an ihrer Weiterentwicklungsfähigkeit gemessen: Nur wenn sie sich flexibel und kontinuierlich erweitern lassen, sind sie konkurrenzfähig und genügen den heutigen Ansprüchen.

In dieser Arbeit wird Weiterentwicklungsfähigkeit gemäß Breivold et al. [BCE07] verstanden als

[...] the ability of a software system to adjust to change stimuli, i.e. changes in requirements and technologies that may have impact on the software system in terms of software structural and/or functional enhancements, while still taking the architectural integrity into consideration.

Pressman [Pre97] argumentierte, dass Aufwand und Kosten einer Änderung überproportional steigen, je später diese im Entwicklungsprozess stattfindet. In der Wartungsphase werden überwiegend funktionale Änderungen durchgeführt, wie eine Studie belegt, die von Sommerville [Som01] beschrieben wird: wie Abbildung 1.1 zeigt, wurde ermittelt, dass 65% des Wartungsaufwandes auf das Hinzufügen und Ändern von Funktionen entfallen, nur 17% auf die Fehlerbehebung selbst und 18% für die Anpassung des Systems an eine geänderte Betriebsumgebung. Die Änderbarkeit des Software-Systems ist somit ein wichtiges Qualitätsziel der Wartung.

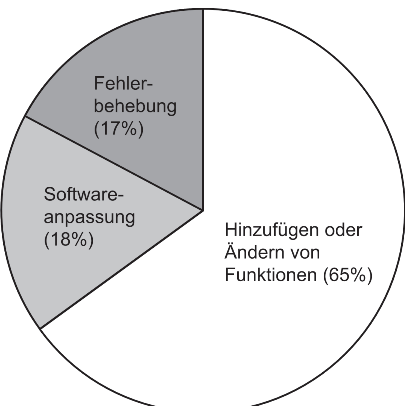


Abbildung 1.1: Verteilung des Aufwands für Wartung aus [Som01].

Neben des oft untersuchten Qualitätsziels Wartbarkeit wird daher mittlerweile in der Literatur die Bedeutung der Weiterentwicklungsfähigkeit als wichtiger Faktor der Software-Qualität differenziert [RB09]. Es geht bei der Verbesserung der Weiterentwicklungsfähigkeit zunächst auch um die Frage, durch welche Faktoren die Weiterentwicklungsfähigkeit beeinflusst werden kann. Beispielsweise sind kontinuierliche Änderungen und Erweiterungen unabdingbar, um ein System an neue oder veränderte Anforderungen anzupassen. Diese Maßnahmen führen aber auch oft zu Qualitätsdefiziten, wie beispielsweise zu einem Anstieg der Komplexität, einem Verfall der Architektur und zu mangelhafter, nicht mehr konsistenter Systemdokumentation: eine Untersuchung von Software-Systemen durch Eick [EGK+98] zeigte, dass der Qualitätsverlust von Software-Systemen oft durch Änderungen am Software-System hervorgerufen wird. Dies reduziert die Weiterentwicklungsfähigkeit und damit den Geschäftswert des Systems massiv. Gerade Qualitätsdefizite in der Spezifikation oder Architektur können Software-Projekte stark beeinträchtigen [SH10]. Werden die Qualitätsdefizite im Software-System zu spät erkannt, führt dies oft auch zum Scheitern des Software-Projekts [SH10].

Um dem entgegenwirken zu können, müssen Methoden entwickelt werden, welche die Einhaltung von Qualitätszielen bewerten können, um dadurch sowohl das Erkennen als auch das Beheben von Qualitätsdefiziten in der Software-Entwicklung zu ermöglichen. Diese Arbeit beschäftigt sich damit, Qualitätsdefizite anhand einer an

Qualitätszielen orientierten und regelbasierten Analyse zu erkennen und zu beheben. Dabei wird die Weiterentwicklungsfähigkeit von Systemen anhand ihrer strukturellen Abhängigkeiten gemessen, bewertet und durch gezielte Maßnahmen verbessert.

1.1 Motivation

Charakteristisch bei der Weiterentwicklung von Software-Systemen ist, dass sich kontinuierlich über einen längeren Zeitraum neue Anforderungen und Änderungen ergeben, die in bereits existierende Teile des Gesamtsystems integriert werden müssen. Dadurch werden sowohl die interne Struktur des Systems als auch die nach außen sichtbaren Schnittstellen stark verändert. Die Herausforderung dabei ist es, das System so aufzubauen, dass die Änderungen keinen Architekturverfall des Systems verursachen und die Weiterentwicklungsfähigkeit des Systems nicht eingeschränkt wird [MM98].

Insbesondere für unternehmenskritische Systeme, wie zum Beispiel sicherheitsrelevante Informationssysteme, wird neben der Wartbarkeit besonders die Weiterentwicklungsfähigkeit gefordert, welche die langfristige Erweiterbarkeit des Systems zum Ziel hat [RB09]. Die Herausforderung dabei ist, Qualitätsmerkmale, die für eine gute Weiterentwicklungsfähigkeit des Software-Systems ausschlaggebend sind, schon während der Entwicklung bewerten zu können. Denn je früher Qualitätsdefizite erkannt werden, desto früher können auch Maßnahmen zu deren Behebung durchgeführt werden. Dies erfordert den Einsatz von Bewertungsverfahren, welche Qualitätsdefizite der Weiterentwicklungsfähigkeit bereits während der Entwicklung objektiv erkennen und beheben können. Diese Bewertungsverfahren sollten iterativ einsetzbar sein, sodass wiederholt nach jedem Entwicklungszyklus das Software-System auf die Existenz von Qualitätsdefiziten untersucht werden kann. Solch eine Regressionsprüfung sollte durch Werkzeuge unterstützt werden können, um so den Aufwand für die Prüfung während der Entwicklung möglichst gering halten zu können.

Problematisch ist allerdings, dass die Weiterentwicklungsfähigkeit, wie auch andere Qualitätsziele von Software-Systemen, wie z.B. die Stabilität oder Testbarkeit, schwer zu messen und zu bewerten sind. Für die Bewertung von Qualitätszielen wurden Qualitätsmodelle erarbeitet. Eines der bekanntesten Qualitätsmodelle für die Softwarequalität ist in der DIN ISO 9126 [ISO01] angegeben. Dort werden aus Sicht der Produkt-Qualität wichtige Qualitätsziele festgelegt.

Neben der quellcodebasierten Restrukturierung wird mit einer Architektur-Restrukturierung der Ansatz verfolgt, die Architektur durch Rekonstruktionen zu verbessern [Sta08a]. Beide Techniken werden innerhalb des Reengineering verwendet, um die Qualität von Software-Systemen zu verbessern. Die Herausforderung beim Einsatz dieser Techniken ist allerdings deren Steuerung, sodass es effektiv und qualitätszielorientiert möglich ist, Qualitätsdefizite zu beheben.

1.2 Ziele der Arbeit

Ein Ziel dieser Arbeit ist es, die Analyse von Qualitätsdefiziten und deren Behebung miteinander in Verbindung zu bringen, so dass eine Qualitätsverbesserung von Software-Systemen erreicht werden kann. Während der Entwicklung von langfristig ausgelegten und komplexen Software-Systemen sollte möglichst früh im Entwicklungsprozess die Weiterentwicklungsfähigkeit bewertet und verbessert werden können. Daher ist es ein weiteres Ziel, Qualitätsdefizite der Weiterentwicklungsfähigkeit möglichst frühzeitig und damit bereits während des Entwicklungsprozesses erkennen zu können. Das frühzeitige Erkennen verspricht das Einsparen von Entwicklungsaufwänden, da die oft langwierige Suche in späteren Phasen vermieden werden kann.

Um die Objektivität der Bewertung zu erhöhen, sollen Metriken explizit im Sinne eines zielorientierten Vorgehens nach den vorgelagerten Qualitätszielen eingesetzt werden können. Das zielorientierte Vorgehen soll dabei eine systematische Analyse eines Qualitätsziels als Grundlage haben. Zudem sollen Qualitätsdefizite möglichst objektiv erkannt und quantifiziert werden können, sodass die Erkennung von Qualitätsdefiziten automatisierbar und auch in der industriellen Praxis effektiv möglich ist.

Beim Beheben von Qualitätsdefiziten sollten Entwickler durch konkrete Verbesserungsvorschläge unterstützt werden, um den Aufwand der Suche nach möglichen Verbesserungsmaßnahmen zu reduzieren. Jede Iteration, in der Qualitätsdefizite behoben wurden, soll effektiv im Sinne einer Regressionsprüfung kontrolliert werden können. Durch eine objektive und wiederholbare Erkennung von Qualitätsdefiziten könnte dieses Ziel erfüllt werden.

Sowohl das Erkennen von Qualitätsdefiziten als auch konkrete Verbesserungsvorschläge zu deren Behebung müssen zusammengefasst und dokumentiert werden, damit das

einmal erworbene Know-How auch in anderen Projekten gewinnbringend eingesetzt werden kann. Hierzu soll die Arbeit geeignete Vorschläge ausarbeiten.

Die Arbeit verknüpft somit zwei Teilgebiete der Informatik, nämlich Software-Qualitätsmanagement und Software-Engineering, indem Qualitätsdefizite, die ein Qualitätsziel beeinflussen, identifiziert und dann erkannt werden müssen und darauf aufbauend (Re-)Engineeringmaßnahmen vorgeschlagen werden, die das Software-Produkt in Richtung des Qualitätsziels weiterentwickeln.

1.3 Gliederung der Arbeit

In Kapitel 2 folgt eine Bewertung bisher veröffentlichter Arbeiten hinsichtlich der oben genannten Ziele. In Abschnitt 2.7 werden auf Grundlage der durchgeführten Bewertung des Stands der Technik die Ziele der Arbeit präzisiert.

Daraufhin führt Kapitel 3 in die in dieser Arbeit entwickelte Methode zur qualitätszielorientieren Erkennung und Behebung von Qualitätsdefiziten ein. Die Methode ist in zwei Prozesse unterteilt: beide, der Aufbauprozess und der Anwendungsprozess, werden in Kapitel 3 vorgestellt.

Den Hauptteil der Arbeit bilden Kapitel 4 und Kapitel 5, da sie die Aktivitäten des Aufbauprozesses im Detail beschreiben. Zudem werden dort die Ergebnisse vorgestellt, die während der Durchführung des Aufbauprozesses am Beispiel des Qualitätsziels Weiterentwicklungsfähigkeit erzielt wurden. Die Ergebnisse aus dem Aufbauprozess und dem Anwendungsprozess wurden anhand einer umfangreichen industriellen Fallstudie validiert. Um die Fallstudie effizient durchführen zu können, wurde ein Werkzeug entwickelt, das in Kapitel 6 beschrieben wird. Damit wurde auch gezeigt, dass große Teile des Anwendungsprozesses automatisierbar sind. Die Ergebnisse aus der Fallstudie werden dann in Kapitel 7 vorgestellt.

Schließlich fasst das Kapitel 8 die Ergebnisse der vorliegenden Arbeit insgesamt zusammen und gibt auch einen Ausblick auf mögliche zukünftige Arbeiten.

Kapitel 2

Bewertung des Stands der Technik

Für das Verbessern der Software-Qualität sind vor allem drei Schritte notwendig: Das Aufstellen von Qualitätszielen, das Erkennen von Qualitätsdefiziten und das Beheben von Qualitätsdefiziten. Um die Qualitätsziele für ein Projekt aufstellen zu können, ist es erforderlich, die Begriffe und deren Bedeutung zu analysieren und zu verstehen. Arbeiten, die sich damit beschäftigen, werden in Abschnitt 2.1 vorgestellt.

Der Abschnitt 2.2 befasst sich dann mit dem Erkennen von Qualitätsdefiziten und Abschnitt 2.3 mit deren Behebung. Anschließend werden in Abschnitt 2.4 Arbeiten untersucht, die beide Teilgebiete verknüpfend betrachten.

Wie bereits in der Einleitung erwähnt, steht das Qualitätsziel Weiterentwicklungsfähigkeit im Fokus dieser Arbeit. Die Literatur, die sich mit diesem Qualitätsziel beschäftigt, wird in Abschnitt 2.5 vorgestellt. Um für das Qualitätsziel Weiterentwicklungsfähigkeit Qualitätsdefizite zu erkennen und zu beheben, ist es notwendig, Abhängigkeitsbeziehungen zu analysieren und Änderungen zu verfolgen. Daher werden Ansätze für die Abhängigkeitsanalyse und Verfolgbarkeit in Abschnitt 2.6 betrachtet.

Abschließend folgt in Abschnitt 2.7 eine Bewertung des Stands der Technik auf den genannten Gebieten. Die dabei identifizierten Probleme, sollen im weiteren Verlauf dieser Arbeit angegangen und behoben werden.

2.1 Analyse von Qualitätszielen

Für die Analyse von Qualitätszielen gibt es den Ansatz von Qualitätsmodellen. Unter einem Qualitätsziel wird in dieser Arbeit die angestrebte Ausprägung einer Qualitätseigenschaft eines Software-Produkts verstanden. Ein Qualitätsmodell hat im Allgemeinen die Aufgabe, Qualitätsziele soweit in Qualitätsmerkmale zu verfeinern, dass diese durch Metriken bewertbar sind. Ein Qualitätsmodell hat daher einen hierarchischen Aufbau, wobei die Anzahl der Zerlegungsebenen nicht festgelegt ist [Lig02].

Üblicherweise werden Qualitätsmodelle mit Hilfe von an Qualitätszielen orientierten Zerlegungs- und Verfeinerungsmethoden aufgebaut. Bekannte Beispiele sind die Methoden Goal Question Metric (GQM) [Bas92], Softgoal Interdependency Graph (SIG) [CNYM00] und Factor Criteria Metrics (FCM) [MRW77]. Der FCM-Ansatz ist weitgehend akzeptiert und wird daher in vielen Qualitätsmodellen [EL96, BD02] und Standards [ISO01] verwendet.

Das Ziel von FCM ist die systematische Analyse und Verfeinerung von Qualitätszielen für die Bewertung durch Metriken. Dabei werden die noch abstrakten Qualitätsziele[1] (Verlässlichkeit, Wartbarkeit, etc.) im nächsten Schritt durch konkretere Qualitätsmerkmale (Fehlertoleranz, Änderbarkeit, etc.) verfeinert.

Zuletzt werden den Qualitätsmerkmalen Metriken oder aber auch andere Prüftechniken, wie etwa Reviews, zugeordnet. Der dadurch resultierende hierarchische Aufbau soll das Verstehen des Qualitätsziels erleichtern. Beziehungen zwischen den aufgestellten Qualitätszielen und Qualitätsmerkmalen können damit besser nachvollzogen werden.

Erni und Lewerentz [EL96], Gillibrand und Liu [GL98] sowie Bansiya und Davis [BD02] erstellten Qualitätsmodelle nach dem FCM-Ansatz. Beispielhaft wird das Qualitätsmodell QMOOD (Quality Model of Object Oriented Design) nach Bansiya und Davis [BD02] betrachtet, da es einen klaren hierarchischen Aufbau hat. Es besteht aus drei Ebenen: Qualitätszielen, Qualitätsmerkmalen und Metriken, wie auch Abbildung 2.1 zeigt. Die Metriken sind Qualitätsmerkmalen zugeordnet, da sie diese bewerten können. Auffällig ist, dass immer eine Metrik genau ein Qualitätsmerk-

[1] Um die zielorientierte Vorgehensweise hervorzuheben, wird hier der Begriff Qualitätsziel verwendet, statt Qualitätsfaktor (engl. quality factor), wie in der Arbeit von McCall [MRW77]. Ebenso wird der Begriff Kriterien (engl. criteria) aus der Arbeit von McCall ersetzt durch den Begriff Qualitätsmerkmale.

mal bewertet. Beispielsweise bewertet die Metrik „Number of Methods (NOM)“ das Qualitätsmerkmal Komplexität. Allerdings wird die Komplexität von Klassen oder Schnittstellen nicht nur durch die Anzahl der Methoden bestimmt. Vielmehr wirkt sich z.B. die Struktur des Codes innerhalb der Methoden oder deren Größe auch auf dieses Qualitätsmerkmal aus. Ein Nachteil dieses Qualitätsmodells ist es daher, dass nicht mehrere Metriken kombiniert werden können. Hinzu kommt ebenso, wie auch für die anderen oben genannten Qualitätsmodelle, dass keine Lösungen für die Interpretation der Metriken zur Qualitätsbewertung integriert sind.

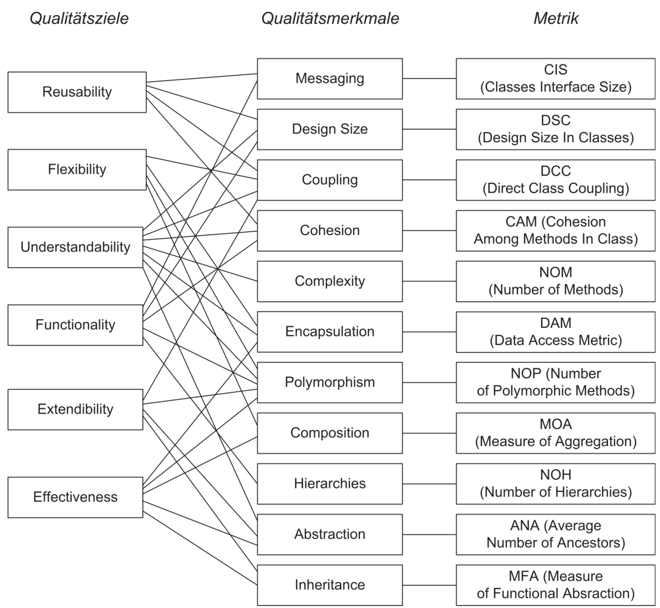


Abbildung 2.1: Qualitätsmodell nach Bansiya und Davis.

Oft ist bei den existierenden Qualitätsmodellen erkennbar, dass die FCM-Methode nicht konsequent angewendet wird. Beispielsweise werden im Qualitätsmodell nach Bansiya und Davis bekannte Qualitätsmerkmale aus Entwurfsmethoden in Beziehung zu Metriken gesetzt, ohne zu prüfen, ob diese Qualitätsmerkmale nicht weiter verfeinert werden können. Die Einführung von weiteren Verfeinerungsebenen, die die FCM-Methode ermöglichen würde, wird hier nicht genutzt. Daher sind diese Qualitätsmodelle nicht so fein-granular aufgebaut, dass die Zuordnungen zwischen Quali-

tätszielen über Qualitätsmerkmale bis hin zu den Metriken klar und nachvollziehbar sind. Zudem fehlen oft auch anwendbare Verbesserungsmaßnahmen, um auf Grundlage der durchgeführten Analyse der Qualitätsziele erkannte Qualitätsprobleme zu beheben [WBD+10].

2.2 Erkennung von Qualitätsdefiziten

Für die Analyse von Qualitätsdefiziten wurde durch Brown et al. [BMM98] ein prozessbasierter Ansatz eingeführt. Antipattern nach Brown [BMM98] sind eine Sammlung von Negativ-Beispielen bezüglich der Architektur oder bezüglich des Entwicklungsprozesses, die immer wieder auftreten und sich negativ auf die Qualität des Software-Systems auswirken. Ein Beispiel für ein Antipattern aus [BMM98] ist „Lavafluss“: Statt toten Code zu entfernen, wird über die Zeit mehr und mehr Code um ihn herum gebaut.

Alle Antipattern sind nach dem gleichen Muster beschrieben, was deren Anwendbarkeit erhöht. Durch manuelle Prüfungen werden durch einen Experten Arbeitsergebnisse wie etwa Code, Architektur-Modelle oder Entwurfsdokumente mit Hilfe der Antipattern auf Mängel hin untersucht. Im Gegensatz zu Entwurfsmustern [GHJV95], die konkrete Entwurfslösungen beschreiben, beziehen sich Antipattern vor allem auf schlechte Praktiken. Daher ist der Ansatz stark von der Ausbildung und der Subjektivität der Person abhängig.

Der Arbeit von Garcia [GPEM09a] folgend werden Architekturmängel (oder Architecture-Smells) verstanden als

[...] a commonly (although not always intentionally) used architectural decision that negatively impacts system quality.

Für das Erkennen von Architekturmängeln werden Entitäten der Architektur wie etwa Schichten, Komponenten oder Klassen betrachtet, aber keine Entitäten der Spezifikation [GPEM09a]. Ein Beispiel für ein solches Qualitätsdefizit ist „Zyklen zwischen Komponenten“ [Roo04], welches auch für das Bewerten des Qualitätsziels Weiterentwicklungsfähigkeit relevant ist.

Für die Erkennung von Qualitätsdefiziten sind Metrik-basierte Ansätze geeignet, da sie es ermöglichen, Qualitätsmerkmale anhand von Zahlenwerten abzubilden und so

Abweichungen festzustellen. Existierende Metriken sind meist einfach anzuwenden und lassen Rückschlüsse auf Qualitätsdefizite zu. Metriken wie etwa „Coupling Between Objects (CBO)“ [CK94] basieren auf dem Zählen von Abhängigkeitsbeziehungen zwischen Klassen. Es bleibt aber dem Experten überlassen, einzuschätzen, welches Qualitätsdefizit vorliegen könnte. Ein Hauptproblem ist daher, dass zu den Metriken nur selten passende Interpretationshilfen geliefert werden, damit klar wird, wie die Metriken eingesetzt werden sollen [WBD+10]. Auch Ford [For09] beschreibt, dass die Bedeutung von Metriken meist unklar ist, da die berechnete Zahl für sich alleine gesehen wenig Aussagekraft hat. Nützliche Ergebnisse bekommt der Qualitätsingenieur nur durch das Vergleichen von Werten, durch die Kombination von Metriken oder durch die Betrachtung der Ergebnisse über die Zeit.

Basili [DB85] forderte bereits früh Regeln zur Interpretation von Metriken, die dann auch benutzt werden können, um Qualitätsdefizite zu erkennen. Er schlägt in seiner Arbeit die Erstellung einer Liste vor, die jede Metrik auflistet und aussagt, wie diese interpretiert werden soll. Dies ist eine wichtige Vorarbeit für die Erkennung von Qualitätsdefiziten. Die Idee von Basili, Regeln zur Interpretation von Metriken einzuführen, hat Marinecsu im Ansatz von Erkennungsstrategien (engl. „Detection Strategies“) fortgeführt [MR04]. Sie ermöglichen durch den Einsatz von Regeln Metriken zu kombinieren, um Qualitätsdefizite im Code, sogenannte „Code-Smells“, zu erkennen. Ein Code-Smell ist ein Symptom in der Implementierung, welches zur Verbesserung auffordert und daher überarbeitet werden sollte [Fow99]. Bei Code-Smells liegt in der Regel kein Fehlverhalten vor, wie etwa bei einem Fehler[2]. Ein Beispiel für einen Code-Smell ist die „Hohe Verwendungskopplung [dBDV04]“.

Der Ansatz von Marinescu [MR04] berücksichtigt Qualitätsziele für den Einsatz von Metriken. Allerdings unterstützt er keine Zerlegung von Qualitätszielen in Qualitätsmerkmale im Sinne eines zielorientierten Vorgehens, sodass die Zusammenhänge zwischen Qualitätszielen und Qualitätsmerkmalen kaum nachvollziehbar sind. Zudem berücksichtigt der Ansatz innerhalb der vorgeschlagenen Erkennungsstrategien nur Code-Metriken und auch nur solche mit Bezug zum Qualitätsziel Wartbarkeit.

Ein weiterer auf Metriken basierender Ansatz wurde von Munro [Mun05] eingeführt. Dabei werden Code-Smells in Java Code zuerst analysiert und anhand ihrer Eigenschaften beschrieben, sodass sie besser identifiziert werden können. Dann werden

[2] Liggesmeyer [Lig02] folgend wird Fehler als *[...] die statisch im Programmcode vorhandene Ursache eines Fehlverhaltens oder Ausfalls* verstanden.

ähnlich zur Arbeit von Marinescu Metriken und Regeln zu deren Interpretation aufgestellt, um eine objektivere Erkennung von Qualitätsdefiziten zu ermöglichen. Der Ansatz ist allerdings auf Code-Smells limitiert und liefert keine Möglichkeit, Metriken zielorientiert und damit ausgehend von einem Qualitätsziel herzuleiten.

Für die systematische Erkennung von Code-Smells wurde die Methode DECOR (DETection and CORrection) durch Moha et al. [MGDM10] erstellt. Die Grundlage für die Erkennung von Qualitätsdefiziten im Code ist deren detaillierte Spezifikation durch DSL (Domain Specific Language) mit einem entsprechenden Vokabular. Ziel ist es, Algorithmen zu generieren, die unabhängig von der Programmiersprache für die Erkennung von Qualitätsdefiziten im Code verwendet werden können. Durch diesen Ansatz wird der Algorithmus für die Erkennung der Qualitätsdefizite im Code explizit generiert. Allerdings unterstützt die Methode kein zielorientiertes Vorgehen, sodass die Auswahl der Algorithmen nicht gesteuert werden kann. Obwohl der Begriff „CORrection“ Teil des Namens ist, wurden Ansätze zum Beheben der erkannten Qualitätsdefizite noch nicht angegangen, sondern nur in Ausblick gestellt.

Für die Erkennung von Qualitätsdefiziten im Entwurf werden Visualisierungstechniken eingesetzt; Dhambri nennt diese „Design-Anomalien“ [DSP08]. Dabei wird eine manuelle Prüfung des Software-Produkts durch Visualisierungen unterstützt, woraufhin übergreifende Entwurfsprobleme, die sich beispielsweise über mehrere Komponenten erstrecken, leichter erkannt werden können. Dieser Ansatz ist vor allem effektiv bei schwer erkennbaren Design-Anomalien. Die typischen Qualitätsdefizite im Code können mit Hilfe von Metriken meist effizienter erkannt werden.

Die Betrachtung der in diesem Abschnitt vorgestellten Ansätzen zeigt, dass die existierenden Ansätze bereits einige Typen von Qualitätsdefiziten betrachten. Sie zeigen, dass ein Software-System durch ein breites Spektrum von Qualitätsdefiziten beeinträchtigt werden kann. Allerdings werden durch die bisher genannten Typen von Qualitätsdefiziten, wie Code-Smell, Architekturmangel oder Antipattern, keine Qualitätsdefizite berücksichtigt, welche sich über mehrere Ebenen eines Software-Systems erstrecken, wie etwa Spezifikation, Architektur oder Implementierung. Gerade aber Abhängigkeitsbeziehungen zwischen Entitäten der Spezifikation und der Architektur oder zwischen der Spezifikation und der Implementierung eines Software-Systems können zu Qualitätsdefiziten führen [BBR09].

2.3 Beheben von Qualitätsdefiziten

Techniken zur Behebung von Qualitätsdefiziten lassen sich meist dem Bereich des Reengineering zuordnen. Reengineering ist ein wichtiger Bestandteil in der Softwareentwicklung geworden, da es zum Ziel hat, Software-Systeme in einem iterativen Prozess zu verbessern. Dazu gehört die Restrukturierung, welche die interne Struktur von Software-Systemen verbessern soll, ohne dabei die Funktionalität zu beeinflussen [dBDV04]. Die Restrukturierungen können sich auf die Dokumentation der Architektur oder auch auf den Code beziehen.

Eine Technik für das Restrukturieren von Code ist das Refactoring nach Fowler [Fow99], welche mittlerweile in der Software-Entwicklung etabliert ist. Das Ziel dieser Technik ist die Verbesserung der Code-Qualität, indem durch eine Sequenz von einfachen Änderungen Code-Smells behoben werden. Ein Beispiel für ein solches Qualitätsdefizit ist „Zu Große Klassen“ [Fow99]. Allerdings hängt die erfolgreiche Anwendung sehr stark von der Expertise des Entwicklers ab.

Unter dem Begriff „Architekturmängel“ wird eine Architektur-Entscheidung verstanden, die sich negativ auf die Qualität der Architektur auswirkt [GPEM09b]. Ein bekanntes Beispiel für solch ein Qualitätsdefizit ist die „Fehlende Schnittstelle zur Abstraktion“. Dieses Qualitätsdefizit bezieht sich auf das Qualitätsmerkmal der Abstraktion und nimmt Bezug zum Prinzip der „Dependency Inversion“ [Mar00]: Eine Architekturkomponente sollte eine Schnittstelle haben, die unabhängig von Änderungen der Implementierung (z.B. innerhalb der Klassen) ist. Der Ansatz nach Garcia et al. [GPEM09a] stellt eine textuelle und eine schematische Beschreibung von Architekturmängeln zusammen mit nachvollziehbaren Beispielen vor. Diese Informationen sollen dann einem Experten als Grundlage dienen und ihn beim Beheben von Qualitätsdefiziten unterstützen. Dadurch lassen sich Qualitätsdefizite in der Architektur einfacher beheben. Jedoch hängt der Erfolg sehr von der Erfahrung des Experten ab, der Qualitätsdefizite zu erkennen und die richtigen Maßnahmen zur Behebung auszusuchen hat, da objektive Mechanismen, wie etwa Regeln, nicht Teil des Ansatzes sind.

Ähnlich einzuordnen ist der Ansatz „Object Oriented Reengineering Patterns“ von Demeyer et al. [DDN02]. Er liefert bewährte Muster für das Reengineering von Software-Systemen. In diesem Ansatz werden nicht nur Lösungen für den Code, sondern auch für die Architektur, bereitgestellt. Leider beziehen sich die meisten Muster auf gene-

relle Reengineering-Probleme (wie beispielsweise “Beteilige den Benutzer”), statt auf konkrete und quantifizierbare Qualitätsdefizite.

2.4 Von der Erkennung zur Behebung von Qualitätsdefiziten

Kerievsky [Ker04] ermöglicht es, mit dem Ansatz „Refactoring by Patterns“ Maßnahmen zur Erkennung von Code-Smells stärker mit deren Behebung durch Refactoring gemäß wohl definierter Muster zu verbinden. Dazu werden als erstes eine Menge von Code-Smells beschrieben, die verbessert werden sollten. Zusätzlich werden dann zu jedem Code-Smell mögliche Maßnahmen für das Beheben der Qualitätsdefizite beschrieben. Ein Nachteil des Ansatzes von Kerievsky ist, dass das Vorgehen nicht qualitätszielorientiert ist, daher bleibt unklar, welche Qualitätsziele verbessert werden sollen. Zudem wird die objektive Bewertung der durchgeführten Verbesserungsmaßnahmen am Software-Produkt kaum unterstützt.

Bei der Methode „Softgoal Independency Graph“ von Ivkovic [IK06] werden auf Grundlage von Qualitätszielen, die dort „Softgoals“ genannt werden, Metriken für die objektive Erkennung von Qualitätsdefiziten hergeleitet. Zudem werden Refactoring-Maßnahmen integriert und sind durch Regeln zu „Softgoals“ zugeordnet. Da Softgoals auf der gleichen Ebene in Beziehung gesetzt werden können, ist keine Hierarchie ersichtlich. Dies beeinträchtigt die Verständlichkeit des Modells. Zudem wird die praktische Anwendbarkeit dadurch eingeschränkt, dass Regeln für die Interpretation der eingesetzten Metriken nicht klar definiert werden. Auch werden vor allem nur Qualitätsdefizite bezüglich des Codes betrachtet. Dadurch können viele Qualitätsziele, wie etwa die Weiterentwicklungsfähigkeit, nur ungenügend berücksichtigt werden.

2.5 Analyse des Qualitätsziels Weiterentwicklungsfähigkeit

Im Gegensatz zu anderen Qualitätszielen, wie etwa der Zuverlässigkeit oder der Wartbarkeit, die bereits durch McCall [MRW77] untersucht und in der Norm ISO/IEC

9126 [ISO01] standardisiert wurden, existieren nur wenige Arbeiten zur Weiterentwicklungsfähigkeit von Software-Systemen.

Die Problematik ergibt sich aus der Tatsache, dass in den meisten bisherigen Arbeiten die Qualitätsziele Wartbarkeit und Weiterentwicklungsfähigkeit gleichgesetzt wurden [BCE07] oder schlichtweg die Weiterentwicklungsfähigkeit nicht als eigenständiges Qualitätsziel berücksichtigt wurde. Es gibt aber auch Arbeiten, welche besonders die Unterscheidung der beiden Qualitätsziele herausstellen, wie etwa die Arbeit von Ciraci und van den Broek [CvdB06] oder auch die Arbeit von Cook et al. [CJH01].

Weiderman et al. [WST97] stellen fest, dass größere und strukturelle Änderungen nicht zur Phase der Wartung, sondern zur Weiterentwicklung gehören: Ein Software-System, welches weiterentwicklungsfähig sein soll, muss in der Regel größere und strukturelle Erweiterungen ermöglichen. Maßnahmen, welche zur Verbesserung der Weiterentwicklungsfähigkeit führen, verbessern auch die Wartbarkeit, dies gilt aber nicht zwangsläufig vice versa [RB09]. Beispielsweise führt das Verbessern von Code allein kaum zu einer nennenswerten Verbesserung des Qualitätsziels Weiterentwicklungsfähigkeit [RB09].

Chung et al. [CNYM00] stellen in ihrem NFR-Rahmenwerk („Non Functional Requirement“) durch Anwendung des Softgoal Interdependency Graph „Softgoals“ auf. Ähnlich wie bei einem Qualitätsmodell werden Softgoals in Unterziele verfeinert. Subramanian et al. [SC03] nutzen das NFR-Rahmenwerk, um das Qualitätsziel Weiterentwicklungsfähigkeit zu analysieren, mit dem Ziel, Metriken zur Bewertung anwenden zu können. Allerdings werden keine Qualitätsmerkmale betrachtet, welche die Software-Architektur betreffen. Zudem wird das Qualitätsziel Traceability und dazugehörende Qualitätsmerkmale, wie etwa die Korrektheit, nicht betrachtet, obwohl dieses Qualitätsziel einen starken Einfluss auf die Weiterentwicklung von Software-Systemen hat [DG02].

Breivold et al. [BCE07, BCLL08] diskutieren Qualitätsmerkmale der Weiterentwicklungsfähigkeit detaillierter als Chung et al. [CNYM00]. Sie tun dieses aus der Sicht der Software-Entwicklung und auch aus der Sicht einer kosteneffizienten Weiterentwicklung von Software-Systemen. Es werden auch Vorschläge gemacht, wie eine Bewertung der Qualitätsmerkmale der Weiterentwicklungsfähigkeit durchgeführt werden könnte. Dies ist ein wichtiger Schritt für die Bewertung der Weiterentwicklungsfähigkeit. Jedoch sind die angegeben Maßnahmen zur Quantifizierung der Qualitätsmerkmale eher allgemein gehalten, weshalb es schwer möglich ist, anwendbare Metriken abzuleiten.

2.6 Abhängigkeitsanalyse und Verfolgbarkeit

Die Analyse von Abhängigkeitsbeziehungen zwischen Entwicklungsartefakten eines Software-Systems, wie etwa Klassen, ermöglicht die Bewertung von Qualitätsmerkmalen für die Erkennung von Qualitätsdefiziten. Hierfür werden die Abhängigkeiten zwischen Entwicklungsartefakten erst mit Hilfe von Modellen als Abhängigkeitsbeziehungen beschrieben und dann im nächsten Schritt ausgewertet. Für das Beschreiben und Auswerten von Abhängigkeitsbeziehungen bieten sich Arbeiten an, die sich mit der Verfolgbarkeit (engl. Traceability) von Entwicklungstätigkeiten beschäftigen, wie etwa die Arbeiten von Gotel et al. [GF94] oder Jarke et al. [RJ01]. Traceability wird in dieser Arbeit der IEEE-Terminologie [IEE90] folgend verstanden als

[...] the degree to which a relationship can be established between two or more products of the development process. [...] for example, the degree to which the requirements and design of a given software component match.

In der Literatur herrscht weitgehend Konsens darüber, dass für das Beschreiben und Auswerten von Abhängigkeitsbeziehungen während der Entwicklung ein Abhängigkeitsmodell benötigt wird [PG96, Poh96, RJ01], da es die Struktur von Abhängigkeitsbeziehungen definiert. Für Abhängigkeitsbeziehungen, die von Anforderungen oder Features ausgehen, wird oft der Begriff Traceability-Link verwendet [GF94]. Dieser Begriff wurde bereits durch Sametinger [SR02] definiert:

[...] Traceability-Links designate dependencies between requirements, design, and source code.

Das Abhängigkeitsmodell nach von Knethen [vK02] betrachtet Abhängigkeitsbeziehungen in eingebetteten Systemen. Mit Hilfe von Abhängigkeitsbeziehungen können Anforderungsänderungen hinsichtlich ihrer Auswirkung bewertet werden. Dabei können auch Aussagen über die Qualität der Änderungen gemacht werden, beispielsweise, inwiefern Dokumente, die eine Abhängigkeitsbeziehung zur geänderten Anforderung haben, angepasst wurden. Das Abhängigkeitsmodell nach von Knethen ist speziell auf dieses Ziel ausgerichtet und kann daher nicht ohne weiteres für andere Ziele verwendet werden, wie etwa für die Bewertung eines Software-Systems bezüglich der Weiterentwicklungsfähigkeit.

Zur Bewertung der Weiterentwicklungsfähigkeit werden relevante Abhängigkeitsbeziehungen durch die Architektur des Systems festgelegt. Hofmeister argumentiert,

dass sogenannte „konzeptuelle Abstraktionen“ die Software-Architektur sehr stark beeinflussen [HNS00]. Beispiele sind Anforderungen, Features oder Geschäftsprozesse. Daher müssen nicht nur Abhängigkeitsbeziehungen im Code betrachtet werden können. Egyed hat dieses Problem erkannt: in seinen Arbeiten [Egy01, Egy02, EG04] werden gezielt auch Abhängigkeitsbeziehungen ausgehend von Anforderungen zum Code betrachtet. Die Abhängigkeitsbeziehungen werden verwendet, um daraus Modelle zu erstellen und die Software-Dokumentation zu verbessern. Eine objektive Bewertung der Qualitätsmerkmale durch Metriken wird durch die Arbeiten nicht explizit angestrebt.

In der betrachteten Literatur wurden keine Abhängigkeitsmodelle gefunden, welche für die Auswertung von Abhängigkeitsbeziehungen zwischen Features und Entwicklungsartefakten, wie etwa Architekturkomponenten, verwendet werden können. Features bilden die in vielen Fällen oftmals noch ungenau formulierten Anforderungen ab [Pre09]. Ein Feature ist eine Zusammenfassung von geforderten funktionalen und nicht-funktionalen Eigenschaften, welche sich zu einer Gesamteigenschaft bzw. Funktion aggregieren [ZK04]. Diese dienen dann zum einen als Vertragsgrundlage mit den Kunden und sind dementsprechend unmissverständlich zu verfassen. Zum anderen bilden sie die Grundlage für die Architektur, die Entwicklung und den Test des Software-Produkts [Pre09]. Die Auswertung der Abhängigkeitsbeziehungen zwischen Features und Entwicklungsartefakten ist aber für die Bewertung von Qualitätsmerkmalen der Architektur oder für die Spezifikation eines Software-Systems notwendig, da sie beispielsweise aufzeigen können, inwiefern spezifizierte Features auch wirklich umgesetzt wurden.

2.7 Zusammenfassung und präzisierte Zielstellung

In diesem Kapitel wurden verschiedene Ansätze vorgestellt, deren Ziel und Zweck es ist, die Qualität eines Software-Systems zu verbessern. Abbildung 2.2 zeigt schematisch eine Übersicht der einzelnen Vorgehensweisen.

Beim Ad-hoc-Vorgehen steht der Experte im Vordergrund, der auf Grund seiner persönlichen Erfahrung entscheidet, welche Verbesserungsmaßnahmen die Qualität des Software-Systems erhöhen. Häufig wird dabei aber ein bestehendes Qualitätsdefizit nicht klar erkannt. Strukturierter sind Ansätze, die das Ziel verfolgen, ein Software-System zu analysieren, um Qualitätsdefizite zu erkennen. Das Erkennen ist ein wich-

tiger Schritt, denn es hilft auch bei der Entscheidung über mögliche Verbesserungsmaßnahmen.

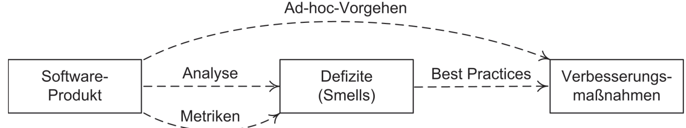


Abbildung 2.2: Übersicht über existierende Ansätze.

Die Analyse von Qualitätszielen ist in Qualitätsmodellen etabliert. Qualitätsmodelle ermöglichen ein zielorientiertes Vorgehen, indem der Einsatz von Metriken entsprechend der vorher festgelegten Qualitätsziele angewendet wird. In diesen Qualitätsmodellen wird aber der Interpretation der Metriken kaum Beachtung geschenkt. Daher ist die Erkennung von Qualitätsdefiziten meist subjektiv, da die Interpretation der Ergebnisse einer Metrik von der Expertise des Qualitätsingenieurs abhängt.

Zudem fehlen bei existierenden Arbeiten, wie etwa Qualitätsmodellen, anwendbare, im Qualitätsmodell integrierte Verbesserungsmaßnahmen, um Qualitätsdefizite zu beheben [WBD+10]. Es gibt zwar Methoden für das Beheben von Qualitätsdefiziten, allerdings ist deren Vorgehen in der Regel nicht zielorientiert, sodass unklar bleibt, welche Qualitätsziele durch die vorgeschlagenen Maßnahmen verbessert werden sollen. Vielmehr sind Verbesserungsmaßnahmen in Mustern fixiert und werden anhand von Beispielen erklärt (Best Practices). Das Beheben von Qualitätsdefiziten beruht daher meist auf der Erfahrung von Experten.

Die Erkennung und Behebung von Qualitätsdefiziten sollte objektiv nachvollziehbar sein und in einer Methode betrachtet werden, woraufhin auch die Umsetzung in Werkzeugen erleichtert würde. Bisher vorgeschlagene Maßnahmen zur objektiven Erkennung von Qualitätsdefiziten beziehen sich vor allem auf den Code. Dadurch können viele Qualitätsziele nur ungenügend berücksichtigt werden, wie beispielsweise das Qualitätsziel Weiterentwicklungsfähigkeit. Das Qualitätsziel Weiterentwicklungsfähigkeit ist besonders für komplexere und auf eine lange Lebensdauer ausgerichtete Software-System essentiell, da solche Systeme langfristig erweiterbar sein sollen. Trotzdem wurde das Qualitätsziel Weiterentwicklungsfähigkeit bisher nicht ausführlich durch Qualitätsmodelle untersucht.

Aus der Bewertung des Stands der Technik lassen sich schließlich die Ziele dieser

Arbeit folgendermaßen zusammenfassen:

1. Erarbeiten einer Methode, mit der ausgehend von einem Qualitätsziel die Analyse des Qualitätsziels, die objektive Erkennung von Qualitätsdefiziten und deren Behebung ermöglicht wird.

2. Anwenden der Methode am Beispiel des Qualitätsziels Weiterentwicklungsfähigkeit durch das Verfeinern des Qualitätsziels und das Zuordnen von anwendbaren Lösungsansätzen.

3. Sowohl das Erkennen als auch das Beheben von Qualitätsdefiziten soll möglichst soweit objektiviert werden, sodass eine Umsetzung in Werkzeugen ermöglicht wird und jede Iteration, in der Qualitätsdefizite behoben wurden, effektiv kontrolliert werden kann. Der Prototyp eines solchen Werkzeugs soll erstellt werden.

4. In einer industriellen Fallstudie soll die Methode unter Einsatz eines solchen Werkzeugs angewandt und dabei validiert werden.

Damit sollen die nach dem bisherigen Stand der Technik im Bereich des Erkennens und Behebens von Qualitätsdefiziten existierenden Mängel behoben werden und für das Qualitätsziel Weiterentwicklungsfähigkeit praktikable Verfahren entwickelt werden. Die Arbeit soll hierbei Software-Systeme aus dem Bereich der betrieblichen Informationssysteme betrachten, welche anhand des Objekt-Orientierten Paradigmas entwickelt werden.

Kapitel 3

Überblick über die Methode

In dieser Arbeit wird eine Methode vorgestellt, mit deren Hilfe Qualitätsziele mehrstufig analysiert und soweit verfeinert werden können, dass damit ein zielorientiertes Erkennen und Beheben von Qualitätsdefiziten möglich wird. In dieser Methode werden Metriken explizit den vorgelagerten Qualitätszielen zugeordnet, zudem erfolgt das Erkennen der Qualitätsdefizite als auch deren Behebung regelbasiert. Dadurch wird die Bewertung objektiviert und die Möglichkeit geschaffen, diese zu automatisieren. Zudem kann so jede Iteration, in der Qualitätsdefizite behoben wurden, effektiv kontrolliert werden.

Die Methode besteht aus zwei Prozessen: dem Aufbau- und dem Anwendungsprozess.

Das Ziel des Aufbauprozesses ist die Erstellung von sogenannten *Reengineering-Rezepten*. Ein Reengineering-Rezept dokumentiert für ein Qualitätsteilmerkmal Metriken, Regeln zur Erkennung von Qualitätsdefiziten im Software-System und Regeln zur Behebung der Qualitätsdefizite durch Reengineering-Aktivitäten. Ein weiteres Ziel des Aufbauprozesses ist der Aufbau eines Graphen, der die Ergebnisse des Aufbauprozesses und ihre Beziehungen dokumentiert, dieser heißt *Zuordnungsgraph zwischen Qualitätsziel und Reengineering-Aktivitäten*. Der Zuordnungsgraph zwischen Qualitätsziel und Reengineering-Aktivitäten ermöglicht einen Überblick über die Ergebnisse des Aufbauprozesses und hilft während der Anwendung der Reengineering-Rezepte. Die Schritte des Aufbauprozesses, der in Abschnitt 3.1 erläutert wird, werden von einem Experten durchgeführt, der idealerweise viel Erfahrung im Bereich Software-Qualität und im Reengineering hat.

Das Ziel des Anwendungsprozesses (siehe Abschnitt 3.2) ist die Verbesserung der

Qualität eines Software-Systems hinsichtlich der für ein Entwicklungsprojekt definierten Qualitätsziele, wie z.B. der *Weiterentwicklungsfähigkeit*. Dabei werden die für ein Qualitätsziel in einem Aufbauprozess erstellten Reengineering-Rezepte durch einen Qualitätsingenieur iterativ angewendet. Auf Grund der bereits genannten Eigenschaften des Zuordnungsgraphen und der Reengineering-Rezepte können die einzelnen Schritte des Anwendungsprozesses effektiv gesteuert und kontrolliert werden. Es wird zwischen den Rollen Experte und Qualitätsingenieur explizit unterschieden, da im Vergleich zum Aufbauprozess, während der Durchführung des Anwendungsprozesses keine besondere Expertise erforderlich ist. Stattdessen profitiert der Qualitätsingenieur von den Erfahrungen des Experten, da die zuvor im Aufbauprozess erstellten Reengineering-Rezepte im Anwendungsprozess nur angewendet werden müssen.

Dass die vorgeschlagene Methode auch in der Praxis einsetzbar ist, wird am Beispiel des Qualitätsziels *Weiterentwicklungsfähigkeit* gezeigt.

3.1 Aufbauprozess

Abbildung 3.1 zeigt den Aufbauprozess in Form eines Aktivitätsdiagramms nach UML.

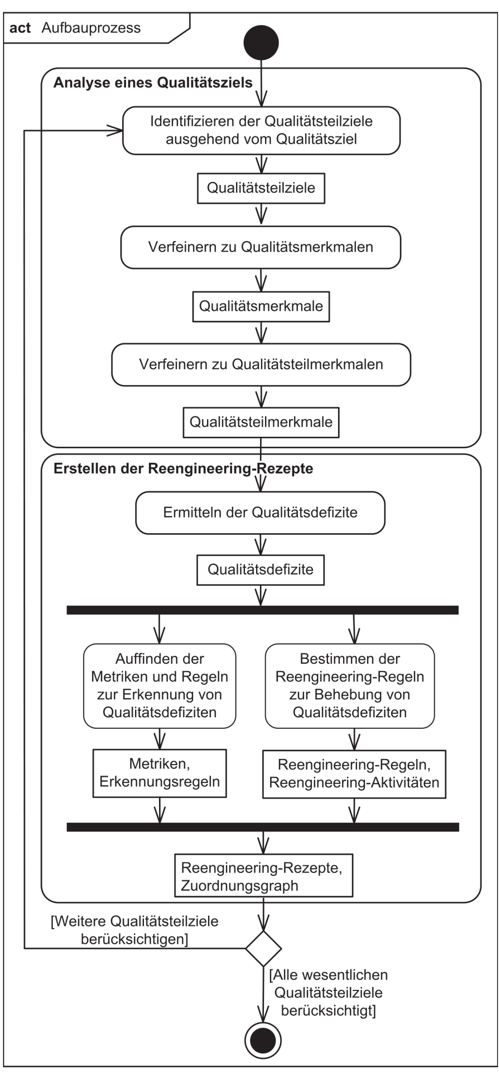


Abbildung 3.1: Aufbauprozess.

Der Aufbauprozess ist in die Schritte „Analyse eines Qualitätsziels“ und „Erstellen der Reengineering-Rezepte“ aufgeteilt. Die beiden Schritte und ihre Anwendung auf das Qualitätsziel *Weiterentwicklungsfähigkeit* werden in den Kapitel 4 und 5 beschrieben.

3.1.1 Analyse eines Qualitätsziels

Die Analyse eines Qualitätsziels besteht wie in Abbildung 3.1 zu sehen aus drei Teilaktivitäten:

1. **Identifizieren der Qualitätsteilziele ausgehend vom Qualitätsziel**

   Ein Qualitätsziel ist der Arbeit [IST10] folgend definiert als *Etwas bezüglich Qualität Angestrebtes oder Erreichendes.* Für die angestrebte Verbesserung des Software-Produkts soll das betrachtete Qualitätsziel erst analysiert und dann in Qualitätsteilziele verfeinert werden. Dabei geht der Experte so vor, dass er Qualitätsteilziele identifiziert, welche das Qualitätsziel konkretisieren. Eine Vorlage bieten hierfür existierende Arbeiten, wie etwa [ISO01]. Fehlen dort wichtige Qualitätsziele oder verfeinerte Qualitätsteilziele, müssen diese argumentativ entwickelt werden. Für das Qualitätsziel *Weiterentwicklungsfähigkeit* ist die *Analysierbarkeit*[1] ein Beispiel für ein Qualitätsteilziel: denn für das Weiterentwickeln des Software-Produkts muss in der Regel zuerst eine Analyse der änderungsbedürftigen Teile gemacht werden. Die Ergebnisse des ersten Analyseschritts sind verfeinerte Qualitätsteilziele, welche dann verwendet werden können, um Qualitätsmerkmale abzuleiten.

2. **Verfeinern zu Qualitätsmerkmalen**

   Ein Qualitätsmerkmal ist eine Eigenschaft eines Software-Systems, welche ein Qualitätsteilziel positiv oder negativ beeinflusst [IST10]. Ein Beispiel für ein Qualitätsmerkmal, das einen Einfluss auf das Qualitätsteilziel *Analysierbarkeit* hat, ist die *Modularität*. Für eine objektivere Bewertung werden ausgehend vom Qualitätsteilziel Qualitätsmerkmale identifiziert. Dies ist notwendig, da Qualitätsziele nicht direkt durch Metriken bewerten werden können.

[1] Gemäß dem Standard ISO/IEC 9126-1 ist die Analysierbarkeit definiert als *[...] Aufwand, der benötigt wird, um Ursachen von Versagen oder Mängel zu diagnostizieren oder um änderungsbedürftige Teile zu bestimmen* [ISO01].

3. **Verfeinern zu Qualitätsteilmerkmalen**

   Für eine effektive Ermittlung von Qualitätsdefiziten ist eine verständliche und fein-granulare Verfeinerung notwendig. Daher wird eine weitere Verfeinerungsebene von Qualitätsmerkmalen zu Qualitätsteilmerkmalen eingeführt. Ein Qualitätsteilmerkmal ist ein Ergebnis einer Verfeinerung von einem Qualitätsmerkmal [IST10], welches durch eine Metrik direkt bewertet werden kann. Zum Beispiel ist die *Lose Kopplung* von Klassen ein mögliches Qualitätsteilmerkmal des Qualitätsmerkmals *Modularität*.

3.1.2 Erstellen von Reengineering-Rezepten

Das Erstellen von Reengineering-Rezepten besteht gemäß Abbildung 3.1 aus drei Teilaktivitäten:

1. **Ermitteln der Qualitätsdefizite**

   Qualitätsdefizite und Metriken hängen stark voneinander ab, da sie beide auf Grundlage von Qualitätsteilmerkmalen identifiziert werden können. Oft wird der Experte eine Menge von Qualitätsdefiziten aus seiner Erfahrung aufstellen können. Es ist aber auch möglich, aus der Literatur bekannte Qualitätsdefizite zu ermitteln. Qualitätsdefizite können zwischen Projekten variieren. Dies ist beispielsweise dann der Fall, wenn Projekte Software-Systeme anhand unterschiedlicher Paradigmen entwickeln. Ein Beispiel für ein Qualitätsdefizit, das in einem Software-System zu finden ist, welches anhand des Objekt-Orientierten Paradigmas entwickelt wurde, stellt Zyklen zwischen Klassen dar.

2. **Auffinden der Metriken und Regeln zur Erkennung von Qualitätsdefiziten**

   Für eine objektivere Bewertung von Qualitätsmerkmalen werden Metriken eingesetzt. Deren Anwendung kann automatisiert werden, daher sind sie auch in größeren Software-Systemen anwendbar. Der Experte geht von den ermittelten Qualitätsdefiziten und Qualitätsteilmerkmalen aus und entwickelt eine dazu passende Metrik. Dabei kann er entweder auf existierende Metriken aus der Literatur zurückgreifen oder neue Metriken herleiten. Beispielsweise gibt es Metriken, welche Abhängigkeitsbeziehungen eines Software-Systems auswerten, wie etwa Abhängigkeitsbeziehungen zwischen Entitäten der Implementierung: zum

Beispiel Klasse `Anzeige` `benutzt` Klasse `Adresse` für die Anzeige einer Adresse. Hier und im weiteren Verlauf dieser Arbeit wird, den Autoren [RGP86, vKP02] folgend, der Begriff *Entität* synonym zu den Begriffen *Artefakt* [RJB04] oder *Entwicklungsartefakt* verwendet.

Eine grundsätzliche Herausforderung bei der Verwendung von Metriken ist allerdings die richtige Interpretation der Messwerte, beispielsweise durch Schwellwerte oder auf Grund der Tendenz der Messwerte. Existierende Ansätze, wie etwa Qualitätsmodelle, berücksichtigen in der Regel keine Strategien für die Interpretation von Metriken. Dies limitiert deren Effektivität, da ein Einsatz in Werkzeugen nicht ohne Weiteres möglich ist. In der hier dargestellten Methode werden Erkennungsregeln aufgestellt, deren Sinn und Zweck es ist, die Erkennung von Qualitätsdefiziten zu objektivieren und damit zu automatisieren. Die Erkennungsregeln definieren, in welchen Fällen ein Qualitätsdefizit vorliegt. Ein Beispiel für eine Erkennungsregel ist: `Zyklen zwischen Klassen := KLIZ > 0`. Der Experte stellt durch das Aufstellen der Erkennungsregeln die Anwendbarkeit der Metriken sicher.

3. **Bestimmen der Reengineering-Regeln zur Behebung von Qualitätsdefiziten**

Eine Reengineering-Regel definiert eine Auswahl und Kombination von Reengineering-Aktivitäten auf Grundlage von erkannten Qualitätsdefiziten. Zum Beispiel definiert eine Regel, dass im Fall des Qualitätsdefizits Geringe Kohäsion von Klassen eine Klasse durch Refactoring restrukturiert werden soll. Das zugehörige Beispiel einer Reengineering-Aktivität ist daher das Restrukturieren der Klasse durch Refactoring nach Fowler [Fow99]. Eine Reengineering-Aktivität definiert, welche Entitäten auf welche Art und Weise geändert werden sollen. Der Experte bestimmt durch die Vorgabe der Reengineering-Regel, welche Reengineering-Aktivitäten für das Beheben eines erkannten Qualitätsdefizits angewendet werden können. Existierende Ansätze wie beispielsweise Qualitätsmodelle berücksichtigen dies kaum [WBD+10]. Oft wird auch nur ein sogenanntes „Trial and Error“ Vorgehen vorgeschlagen, das so lange durchgeführt wird, bis das Software-Produkt verbessert wurde. In dieser Arbeit wird dagegen ein systematisches Vorgehen definiert, indem ausgehend vom erkannten Qualitätsdefizit passende Reengineering-Aktivitäten mit Hilfe von Reengineering-Regeln durch den Experten definiert werden.

Am Ende des Aufbauprozesses (siehe Abbildung 3.1) werden Erkennungsregeln und Reengineering-Regeln zu sogenannten Reengineering-Rezepten zusammengefasst. Beispiele von Reengineering-Rezepten werden in Kapitel 5.4 vorgestellt.

Durch Reengineering-Rezepte können Verbesserungsmaßnahmen besser verglichen und abgeschätzt werden, da Lösungen für das Erkennen und Beheben von Qualitätsdefiziten leicht gegenübergestellt werden können. Einmal durch einen Experten erstellte Reengineering-Rezepte können durch Qualitätsingenieure wiederverwendet werden. Dabei profitieren vor allem auch Qualitätsingenieure, die weniger Erfahrung und Wissen im Reengineering haben.

Zudem sind durch die Analysephase die Beziehungen zwischen Qualitätsziel und Reengineering-Aktivitäten am Ende des Aufbauprozesses bekannt und werden durch den *Zuordnungsgraph zwischen Qualitätsziel und Reengineering-Aktivitäten* dokumentiert. Ein am Qualitätsziel orientierter Einsatz von Reengineering-Aktivitäten ist dadurch möglich. Der Aufbau und der Nutzen eines Zuordnungsgraphen werden in Abschnitt 3.1.3 beschrieben.

3.1.3 Der Zuordnungsgraph – ein Ergebnis des Aufbauprozesses

Der Zuordnungsgraph zwischen Qualitätsziel und Reengineering-Aktivitäten dokumentiert inkrementell die Ergebnisse der einzelnen Aktivitäten des Aufbauprozesses.

Der in dieser Arbeit aufgebaute Zuordnungsgraph für das Qualitätsziel *Weiterentwicklungsfähigkeit* ist in Abbildung 3.2 zu finden. Wie in Abbildung 3.2 zu sehen ist, dokumentiert der Zuordnungsgraph zu einem Qualitätsziel die zugeordneten Qualitätsteilziele, Qualitätsmerkmale, Qualitätsteilmerkmale, Metriken, Qualitätsdefizite und Reengineering-Aktivitäten und deren Beziehungen. Ein Nutzen des Zuordnungsgraphen ist die einfache und übersichtliche Darstellung der Zusammenhänge. Weitere Details, wie zum Beispiel die Herleitung der einzelnen Beziehungen, sind in den Kapiteln 4 und 5 zu finden.

Die während der Analyse eines Qualitätsziels verfeinerten Qualitätsteilziele, Qualitätsmerkmale und Qualitätsteilmerkmale werden anhand des linken Teils des Zuordnungsgraphen dargestellt. Der rechte Teil des Zuordnungsgraphen dokumentiert die Metriken, Qualitätsdefizite und Reengineering-Aktivitäten, welche die Basis der

Reengineering-Rezepte bilden. Zu jedem Qualitätsteilmerkmal wird mindestens ein Reengineering-Rezept erstellt. Die Erkennungs- und Reengineering-Regeln eines Reengineering-Rezepts sind auf den im Zuordnungsgraphen dokumentierten Beziehungen aufgebaut. Der rechte Teil des Zuordnungsgraphen bildet daher auch die Grundlage der Reengineering-Rezepte und somit der regelbasierten Erkennung und Behebung von Qualitätsdefiziten. Aus Gründen der Übersichtlichkeit sind die Reengineering-Rezepte jedoch nicht im Zuordnungsgraphen direkt zu sehen.

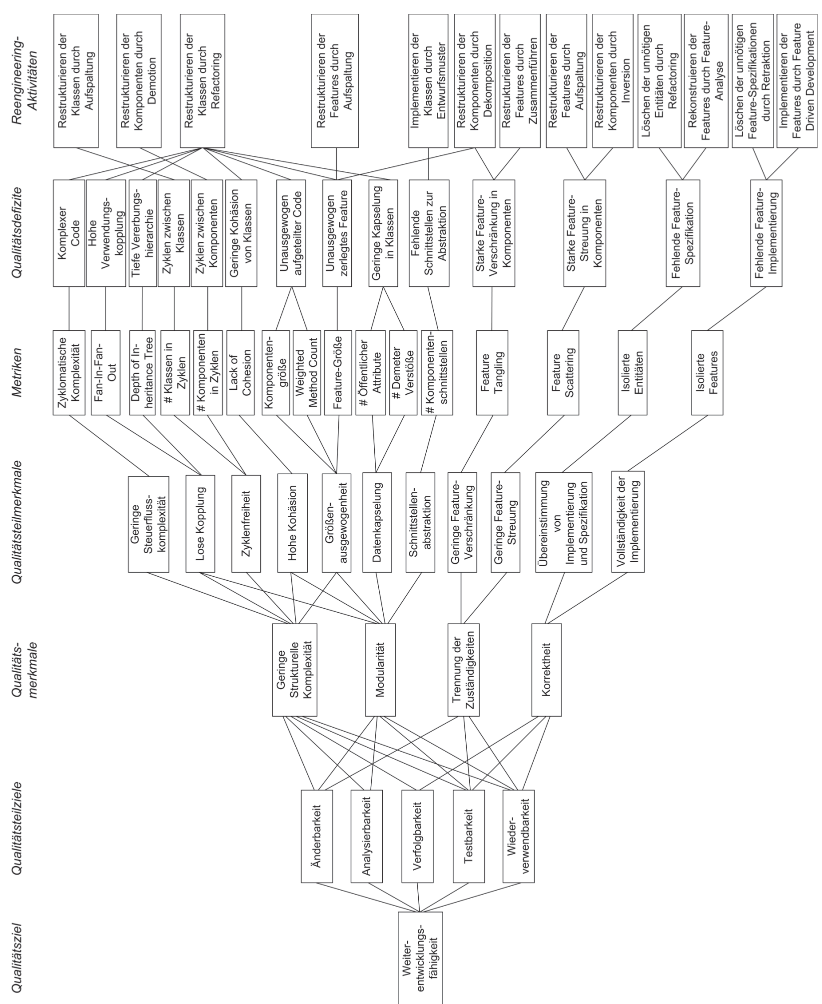


Abbildung 3.2: Zuordnungsgraph zwischen dem Qualitätsziel Weiterentwicklungsfähigkeit und Reengineering-Aktivitäten.

Neben der schrittweisen Dokumentation der Ergebnisse im Aufbauprozess, dient der Zuordnungsgraph außerdem zur Kommunikation der Zusammenhänge zwischen beispielsweise den verfeinerten Qualitätsteilzielen, Qualitätsmerkmalen und Qualitätsteilmerkmalen im Projekt. Darauf aufbauend wird der Zuordnungsgraph auch für die Priorisierung der Reengineering-Rezepte genutzt, da er die Gewichtung von Qualitätsteilzielen, Qualitätsmerkmalen, Qualitätsteilmerkmalen und Reengineering-Rezepten ermöglicht, indem die einzelnen Beziehungen gewichtet und für die Priorisierung ausgewertet werden können. Die Priorisierung von Reengineering-Rezepten wird durch den Anwendungsprozess in Abschnitt 3.2 erläutert.

Einmal für ein Qualitätsziel aufgebaut, kann ein Zuordnungsgraph in anderen Projekten wiederverwendet werden. Allerdings muss dabei natürlich der Anwendungsbereich berücksichtigt werden: Beispielsweise ist ein Zuordnungsgraph, der bezüglich der Bewertung von Informationssystemen aufgebaut wurde, nicht ohne weiteres auch für ein Software-System gültig, das einen technischen Prozess steuern soll, wie etwa ein Antiblockiersystem in einem Auto.

Schließlich können Teile des Zuordnungsgraphen in einem Werkzeug umgesetzt werden (siehe dazu Kapitel 6), sodass er als Ausgangspunkt für die Entwicklung eines Werkzeugs zur automatischen Erkennung und Behebung von Qualitätsdefiziten genutzt werden kann.

3.2 Anwendungsprozess

Für die zielorientierte Erkennung und Behebung von Qualitätsdefiziten werden die aufgebauten Reengineering-Rezepte auf ein Software-System innerhalb eines Projekts angewendet. Hierbei folgt der Qualitätsingenieur dem Anwendungsprozess, der in Abbildung 3.3 dargestellt ist.

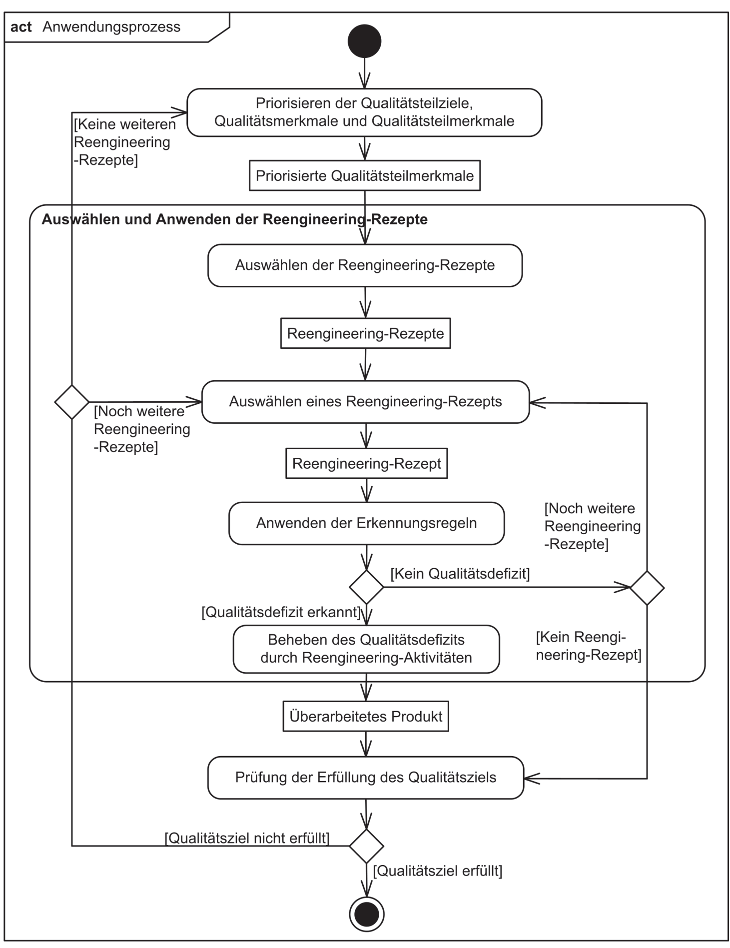


Abbildung 3.3: Anwendungsprozess.

Der Anwendungsprozess ist in die drei Schritte „Priorisieren der Qualitätsteilziele, Qualitätsmerkmale und Qualitätsteilmerkmale“, „Auswählen und Anwenden der Reengineering-Rezepte“ und „Prüfung der Erfüllung der Qualitätsziele“ aufgeteilt. Im Folgenden werden diese Schritte und das Vorgehen dargestellt. Details zu den Bestandteilen des Anwendungsprozesses am Beispiel der Fallstudie finden sich dann im Kapitel 7 der Arbeit wieder.

Um den Anwendungsprozess durchführen zu können, muss die folgende Vorbedingung gewährleistet sein: Das aus Sicht des Projekts geforderte Qualitätsziel wurde bereits anhand des Aufbauprozesses betrachtet, sodass sowohl Reengineering-Rezepte als auch ein Zuordnungsgraph zwischen Qualitätsziel und Reengineering-Aktivitäten bezüglich des geforderten Qualitätsziels erstellt wurden. Ein Beispiel für einen Zuordnungsgraph zwischen Qualitätsziel und Reengineering-Aktivitäten ist in Abbildung 3.2 dargestellt.

3.2.1 Priorisieren der Qualitätsteilziele, Qualitätsmerkmale und Qualitätsteilmerkmale

Ausgehend vom Qualitätsziel werden die Qualitätsteilziele über eine Gewichtung priorisiert. Das Gewicht eines Qualitätsteilziels wird verteilt auf die dazugehörigen Qualitätsmerkmale und weiter auf die Qualitätsteilmerkmale. Dadurch werden die zur Verfügung stehenden Qualitätsteilziele, Qualitätsmerkmale und Qualitätsteilmerkmale stufenweise gewichtet, woraufhin am Ende ein Priorisieren der Reengineering-Rezepte anhand der zugeordneten und bereits gewichteten Qualitätsteilmerkmale möglich ist. Der Zuordnungsgraph zwischen Qualitätsziel und Reengineering-Aktivitäten ist für das Priorisieren sehr hilfreich, denn er stellt alle Beziehungen zwischen den Qualitätsteilzielen, den Qualitätsmerkmalen und den Qualitätsteilmerkmalen dar.

3.2.2 Auswählen und Anwenden der Reengineering-Rezepte

Das Auswählen und das Anwenden der Reengineering-Rezepte wird in vier Schritte unterteilt. Diese sind bereits in Abbildung 3.3 dargestellt und werden in den folgenden Abschnitten beschrieben:

1. **Auswählen der Reengineering-Rezepte**

   Da zu jedem Qualitätsteilmerkmal mindestens ein Reengineering-Rezept zugeordnet ist, wählt der Qualitätsingenieur in diesem Schritt eine Menge von Reengineering-Rezepten aus, die angewendet werden sollen. Dabei wird auf Grundlage der Prioritäten der ausgewählten Qualitätsteilmerkmale die Anwendungsreihenfolge der Reengineering-Rezepte festgelegt.

2. **Auswählen eines Reengineering-Rezepts**

   In der Regel wählt der Qualitätsingenieur aus der im vorherigen Schritt bestimmten Menge von Reengineering-Rezepten das am höchsten priorisierte Reengineering-Rezept aus und wendet es als erstes an.

3. **Anwenden der Erkennungsregeln**

   Das Ziel des Qualitätsingenieurs ist zunächst das Erkennen eines Qualitätsdefizits mit Hilfe der Erkennungsregeln aus dem ausgewählten Reengineering-Rezept. In den Erkennungsregeln ist festgehalten, welche Metriken anzuwenden sind und wie deren Ergebnisse zu interpretieren sind. Das Erkennen von Qualitätsdefiziten erfolgt auf einer objektiven Basis und kann also leicht automatisiert werden, da die dafür notwendigen Metriken und Erkennungsregeln in Werkzeuge integriert werden können.

   Konnte kein Qualitätsdefizit mit Hilfe des ausgewählten Reengineering-Rezepts erkannt werden, wird der Qualitätsingenieur in der Regel das nächste Reengineering-Rezept zur Anwendung auswählen. Wurden bereits alle Reengineering-Rezepte aus der vorher festgelegten Menge ausgewählt, wird er im Schritt „Prüfung der Erfüllung der Qualitätsziele“ über sein weiteres Vorgehen entscheiden.

4. **Beheben der Qualitätsdefizite durch Reengineering-Aktivitäten**

   Die erkannten Qualitätsdefizite werden durch die im Reengineering-Rezept dokumentierten Reengineering-Aktivitäten behoben. Entsprechend des Qualitätsdefizits existieren oft mehrere Reengineering-Aktivitäten, daher wird deren Auswahl durch Reengineering-Regeln gesteuert. Die Durchführung der vorgeschlagenen Reengineering-Aktivitäten durch eine Person aus dem Entwicklungsteam, wie etwa einem Entwickler, führt zu einem überarbeiteten Produkt.

3.2.3 Prüfung der Erfüllung der Qualitätsziele

Durch nochmaliges Anwenden der Metriken und Erkennungsregeln wird geprüft, ob das überarbeitete Produkt das Qualitätsziel nun erfüllt. Sind alle erkannten Qualitätsdefizite wie beabsichtigt behoben, konnte die Qualität des Software-Produkts bezüglich des Qualitätsziels verbessert werden.

Wie in Abbildung 3.3 dargestellt, gibt es an dieser Stelle im Anwendungsprozess eine Entscheidung über das weitere Vorgehen. Diese hängt von den Ergebnissen der Prüfung ab (dargestellt durch den unteren Kontrollknoten):

1. Qualitätsziel erfüllt: Der Qualitätsingenieur entscheidet, ob das Qualitätsziel, wie etwa die *Weiterentwicklungsfähigkeit*, durch die Anwendung der Reengineering-Rezepte erreicht wurde. Wenn alle erkannten Qualitätsdefizite behoben wurden, ist der Anwendungsprozess beendet.

   Sofern ein weiteres Qualitätsziel betrachtet werden soll, wie etwa die Usability, wird der Anwendungsprozess von vorne begonnen (Voraussetzung für die Betrachtung eines weiteren Qualitätsziels ist aber, dass der Aufbauprozess bereits durchgeführt wurde).

2. Qualitätsziel nicht erfüllt: Der Qualitätsingenieur entscheidet, ob das Qualitätsziel, wie etwa die *Weiterentwicklungsfähigkeit*, durch die Anwendung der Reengineering-Rezepte erreicht wurde. Ist dies nicht der Fall, gibt es zwei Möglichkeiten:

   2.1 Noch weitere Reengineering-Rezepte: Wenn noch nicht alle geplanten Reengineering-Rezepte betrachtet wurden, kann nun ein weiteres ausgewählt und angewendet werden.

   2.2 Keine weiteren Reengineering-Rezepte: Als ultima ratio bleibt eine Neupriorisierung, sodass vorher höher priorisierte Qualitätsteilmerkmale dann niedriger priorisiert werden. Daraus resultiert in der Regel eine veränderte Menge und/oder eine veränderte Reihenfolge von anzuwendenden Reengineering-Rezepten. Mit dieser Entscheidung wird der Anwendungsprozess neu durchlaufen.

3.3 Zusammenfassung

Die in dieser Arbeit vorgeschlagene Methode ermöglicht das zielorientierte und regelbasierte Erkennen und Beheben von Qualitätsdefiziten. Mit Hilfe des vorgestellten Aufbauprozesses werden Reengineering-Rezepte qualitätszielorientiert aufgebaut. Dies hat den Vorteil, dass die aufgebauten Reengineering-Rezepte dann im Anwendungsprozess gemäß den Qualitätszielen des Projekts erst priorisiert und anschließend angewendet werden können. Dadurch ist die Anwendung der Reengineering-Rezepte steuerbar. Zudem ist die objektive Erkennung von Qualitätsdefiziten besser automatisierbar.

Konflikte zwischen Qualitätszielen können durch die durchgeführte Zerlegung genauer benannt und durch die Möglichkeit der Priorisierung flexibler gelöst werden. Zum Beispiel kann ein Konflikt zwischen den Qualitätsteilmerkmalen *Lose Kopplung* und *Hohe Kohäsion* einer Klasse existieren: durch das Zusammenlegen zweier stark gekoppelten Klassen wird eine Verbesserung der Kopplung nur durch eine Minderung der Kohäsion erreicht. Neben der Benennung des Konflikts kann in dieser Methode zur Lösung dieses Konflikts ein Qualitätsteilmerkmal höher priorisiert werden (z.B. *Lose Kopplung*), was zur Folge hat, dass nur damit verbundene Reengineering-Aktivitäten angewendet werden.

Ein weiterer Vorteil dieser Methode ist deren Anpassbarkeit. Es können beispielsweise neue Qualitätsteilziele integriert werden oder Reengineering-Rezepte angepasst werden, indem die im Reengineering-Rezept dargestellten Erkennungs- und Reengineering-Regeln erweitert werden. Zudem ist die Struktur der Reengineering-Rezepte so aufgebaut, dass neue Metriken oder Reengineering-Aktivitäten integriert werden können.

Dass die hier vorgestellte Methode auch in der Praxis anwendbar ist, wird im Verlauf dieser Arbeit gezeigt:

1. Der Aufbauprozess wurde exemplarisch für das Qualitätsziel *Weiterentwicklungsfähigkeit* durchgeführt.

2. Mit Hilfe der so gewonnen Reengineering-Rezepte und des Zuordnungsgraphen wurde der Anwendungsprozess anschließend innerhalb einer industriellen Fallstudie durchgeführt.

Die dabei erzielten Ergebnisse werden in den folgenden Kapiteln dargestellt.

Kapitel 4

Analyse eines Qualitätsziels

Die Analyse eines Qualitätsziels im Hinblick auf eine Verfeinerung dieses Qualitätsziels bis hin zu Qualitätsteilmerkmalen ist die Grundlage für das Erstellen von Reengineering-Rezepten, welche für das Erkennen und Beheben von Qualitätsdefiziten verwendet werden. Die Analyse eines Qualitätsziels unterteilt sich gemäß dem Aufbauprozess, der in Kapitel 3 eingeführt wurde, in drei Aktivitäten, welche in den folgenden Abschnitten 4.1, 4.2 und 4.3 genauer beschrieben werden.

Um die Vorgehensweise zu verdeutlichen, werden die Aktivitäten an einem konkreten Beispiel, dem Qualitätsziel *Weiterentwicklungsfähigkeit*, durchgeführt und die Ergebnisse mit Hilfe des Zuordnungsgraphen zwischen dem Qualitätsziel *Weiterentwicklungsfähigkeit* und den Reengineering-Aktivitäten schrittweise dokumentiert. Zur Veranschaulichung wird dazu jeweils der Ausschnitt des Zuordnungsgraphen abgebildet, der in dem jeweiligen Schritt in den Zuordnungsgraphen aufgenommen wurde. Der komplette Zuordnungsgraph, der während der Anwendung des Aufbauprozesses am Beispiel der *Weiterentwicklungsfähigkeit* aufgebaut wurde, wurde bereits in Abschnitt 3.1.3 durch Abbildung 3.2 dargestellt.

4.1 Identifizieren der Qualitätsteilziele ausgehend von einem Qualitätsziel

4.1.1 Ziel und Vorgehen der Aktivität

Um Qualitätsziele genauer zu beschreiben und damit deren Bewertung zu erleichtern, wird ein Qualitätsziel erst analysiert und dann in Qualitätsteilziele verfeinert. Dabei geht der Experte so vor, dass er Qualitätsteilziele identifiziert, welche das Qualitätsziel beeinflussen. Ein Qualitätsteilziel ist daher ein konkretes Ergebnis einer Verfeinerung. Im Allgemeinen gibt es zwei mögliche Vorgehensweisen: die Verfeinerung des Qualitätsziels kann entweder aus der Literatur entnommen werden oder nach Ermessen des Experten durchgeführt werden. Um weitgehende Akzeptanz zu erreichen, wird empfohlen, existierende Verfeinerungen wiederzuverwenden und diese nur zu erweitern, sofern dies aus Sicht einer erfolgreichen Bewertung eines Qualitätsziels notwendig ist.

Ein Beispiel für die zweite genannte Vorgehensweise, also das Identifizieren eines Qualitätsteilziels nach Ermessen des Experten, soll im Folgenden dargestellt werden: Wenn ein Qualitätsteilziel durch den Aufwand definiert wird, der zur Ausführung bestimmter Tätigkeiten im Software-Entwicklungsprozess benötigt wird, kann die Auswahl eines Qualitätsteilziels auch davon abhängen, ob die in der Definition des Qualitätsteilziels aufgeführten Tätigkeiten auch eine Beziehung zum betrachteten Qualitätsziel haben. Eine solche Beziehung ist immer dann vorhanden, wenn eine Reduzierung des Aufwands für diese Tätigkeiten nicht nur eine Verbesserung des Qualitätsteilziels, sondern auch des betrachteten Qualitätsziels nach sich ziehen. Diese Betrachtungsweise soll anhand eines Beispiels verdeutlicht werden:

Die *Analysierbarkeit* ist ein Qualitätsteilziel des Qualitätsziels *Weiterentwicklungsfähigkeit*, wie bereits Breivold et al. [BCE07] festgestellt haben. Gemäß dem Standard ISO/IEC 9126-1 ist die Analysierbarkeit definiert als

Aufwand, der benötigt wird, um Ursachen von Versagen oder Mängel zu diagnostizieren oder um änderungsbedürftige Teile zu bestimmen [ISO01].

Die in der Definition beschriebenen Tätigkeiten sind für die Weiterentwicklung essentiell, da vor jeder Weiterentwicklung in der Regel eine Analyse des Software-Systems durchgeführt werden muss, um beispielsweise die änderungsbedürftigen Teile zu identifizieren. Eine Verbesserung der *Analysierbarkeit* verbessert damit auch die *Weiter-*

entwicklungsfähigkeit.

Neben dem Qualitätsteilziel *Analysierbarkeit* wurden weitere Qualitätsteilziele für das Qualitätsziel *Weiterentwicklungsfähigkeit* nach diesem Vorgehen identifiziert und in den Zuordnungsgraphen aufgenommen, diese Teilziele werden im folgenden Abschnitt dargestellt.

4.1.2 Ergebnisse des Identifizierens von Qualitätsteilzielen am Beispiel der Weiterentwicklungsfähigkeit

Die Zuordnung zwischen dem Qualitätsziel *Weiterentwicklungsfähigkeit* und dem Qualitätsteilziel *Analysierbarkeit* wurde im vorhergehenden Abschnitt bereits erläutert. Abbildung 4.1 zeigt den Teilbereich des erstellten Zuordnungsgraphen, der die Qualitätsteilziele dokumentiert. Im Folgenden werden die Zuordnungen zwischen dem Qualitätsziel und den weiteren Qualitätsteilzielen begründet.

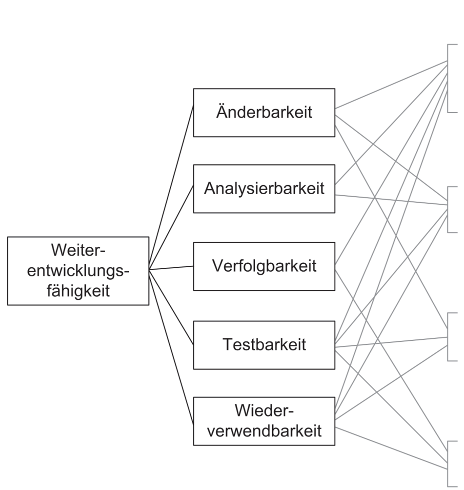


Abbildung 4.1: Identifizierte Qualitätsteilziele der Weiterentwicklungsfähigkeit

4.1.2.1 Änderbarkeit

Der Arbeit von Breivold et al. [BCLL08] folgend wird das Qualitätsziel *Weiterentwicklungsfähigkeit* durch das Qualitätsteilziel *Änderbarkeit* verfeinert. Unter dem Qualitätsteilziel *Änderbarkeit* versteht man den

[...] Aufwand zur Ausführung von Verbesserungen, zur Fehlerbeseitigung oder Anpassung an Umgebungsänderungen [DIN94, ISO01].

Aus dem Standard ISO/IEC 9126 [ISO01] wird das Qualitätsziel *Änderbarkeit* aus dem Qualitätsziel Wartbarkeit abgeleitet. Die *Weiterentwicklungsfähigkeit* und die Wartbarkeit sind bezüglich dieses Qualitätsteilziels verwandte Qualitätsziele [RB09]. Daher lässt sich das im Standard ISO/IEC 9126 [ISO01] definierte Qualitätsteilziel *Änderbarkeit* auch für das Qualitätsziel *Weiterentwicklungsfähigkeit* übernehmen.

4.1.2.2 Verfolgbarkeit

Die *Weiterentwicklungsfähigkeit* wird weiterhin durch das Qualitätsteilziel *Verfolgbarkeit* verfeinert. Unter dem Qualitätsteilziel *Verfolgbarkeit* wird mit Bezug zum IEEE Standard für Begriffe des Software-Engineerings [IEE90] der mögliche Aufstand verstanden,

Entwurfsentscheidungen auf das zugehörige Feature zurückzuführen (und umgekehrt).

Die *Weiterentwicklungsfähigkeit* des Software-Systems hängt von der Fähigkeit ab, grundlegende Eigenschaften der Implementierung eines Features verfolgen zu können. Können diese verfolgt werden, kann die Weiterentwicklung des Features effizienter vorgenommen werden, da die betroffene Implementierung leichter zu identifizieren ist.

4.1.2.3 Testbarkeit

Entsprechend dem Standard ISO/IEC 9126-1 ist das Qualitätsteilziel *Testbarkeit* definiert als

[...] der zur Prüfung der geänderten Software benötigte Aufwand [ISO01].

Wie beim Qualitätsteilziel *Änderbarkeit* wird die *Testbarkeit* als Qualitätsteilziel für

das Qualitätsziel *Weiterentwicklungsfähigkeit* übernommen, da Wartbarkeit und *Weiterentwicklungsfähigkeit* verwandte Qualitätsteilziele sind.

4.1.2.4 Wiederverwendbarkeit

Das Qualitätsziel *Wiederverwendbarkeit* wird in dieser Arbeit in Anlehnung an die Definition von Wiederverwendung nach [Sam97] definiert als

[...] der Aufwand, der benötigt wird, um ein Software-System auf Grundlage von einer existierenden Implementierung weiterzuentwickeln.

Der Aufwand für die Weiterentwicklung eines Software-Systems, beispielsweise wenn neue Features implementiert werden sollen, ist geringer, wenn bereits existierende Komponenten verwendet werden können, welche die geforderte Funktionalität erfüllen, da sich die Entwicklungszeit durch das Wegfallen von überflüssigen Arbeiten verkürzt [Sam97]. Aus diesen Gründen wurde die *Wiederverwendbarkeit* als ein weiteres Qualitätsteilziel des Qualitätsziels *Weiterentwicklungsfähigkeit* eingeführt.

4.1.2.5 Weitere Qualitätsteilziele

Neben den bisher genannten Qualitätsteilzielen gibt es weitere mögliche Qualitätsteilziele der *Weiterentwicklungsfähigkeit*, deren Untersuchung aber den Rahmen dieser Arbeit überschreitet. Die Autoren Breivold et al. [BCLL08] nennen beispielsweise das Qualitätsteilziel Austauschbarkeit von Diensten. Die *Weiterentwicklungsfähigkeit* kann auch durch organisatorische Aspekte oder die Qualität des Software-Entwicklungsprozesses beeinflusst werden. Ein weiteres Beispiel ist zudem auch der „Faktor Mensch“, der für die Software-Entwicklung entscheidend ist und daher auch die Weiterentwicklung von Software-Systemen beeinflusst: konkrete Beispiele sind organisatorische und soziale Einflussfaktoren wie etwa Mitarbeiterführung, Teambildung, kulturelle Unterschiede oder auch Motivation [AJMM05].

4.2 Verfeinern zu Qualitätsmerkmalen

4.2.1 Ziel und Vorgehen der Aktivität

Da Qualitätsteilziele nicht direkt durch Metriken bewertet werden können, werden diese zu Qualitätsmerkmalen verfeinert. Diese Verfeinerung kann entweder aus der Literatur entnommen oder nach Ermessen des Experten durchgeführt werden. Für die Verfeinerung sucht der Experte nach Qualitätsmerkmalen, durch die das Qualitätsteilziel positiv beeinflusst wird. Denn wenn es möglich ist, diese Qualitätsmerkmale zu verbessern, indem Qualitätsdefizite behoben werden, wird das Qualitätsteilziel in der Regel auch verbessert.

Zum Beispiel wird das Qualitätsteilziel *Testbarkeit* durch das Qualitätsmerkmal *Korrektheit* positiv beeinflusst [BCE07]. Das Qualitätsmerkmal *Korrektheit* zeigt sich durch eine korrekte Dokumentation, wie etwa eine konsistente Feature-Spezifikation, oder auch durch korrekt erstellte Traceability-Links in Modellen. Auf dieser Grundlage können solide Tests entwickelt und ausgeführt werden. Zudem ist die *Übereinstimmung zwischen Implementierung und Spezifikation* für die *Testbarkeit* essentiell, da die Implementierung eines Features ohne dessen korrekte Spezifikation nur schwer getestet werden kann.

Neben dem Qualitätsteilziel *Testbarkeit* wurden alle weiteren im letzten Schritt identifizierten Qualitätsteilziele nach diesem Vorgehen verfeinert, woraufhin mehrere Qualitätsmerkmale zugeordnet werden konnten. Die Ergebnisse des Verfeinerns aller Qualitätsteilziele, die für das Qualitätsziel *Weiterentwicklungsfähigkeit* identifiziert wurden, werden im folgenden Abschnitt 4.2.2 dargestellt.

4.2.2 Ergebnisse des Verfeinerns von Qualitätsteilzielen am Beispiel der Weiterentwicklungsfähigkeit

Der gesamte Zuordnungsgraph, in dem die Qualitätsteilziele dokumentiert wurden, die das Qualitätsziel *Weiterentwicklungsfähigkeit* verfeinern, ist in Abbildung 3.2 zu sehen. In den folgenden Abschnitten werden für jedes Qualitätsteilziel die verfeinerten Qualitätsmerkmale und ihre Zuordnungen beschrieben und die durchgeführte Verfeinerung durch existierende Arbeiten begründet.

4.2.2.1 Verfeinern von Änderbarkeit

Das Qualitätsteilziel *Änderbarkeit* wird in dieser Arbeit durch die Qualitätsmerkmale *Geringe Strukturelle Komplexität*, *Modularität* und Trennung von Zuständigkeiten verfeinert. Diese Beziehungen wurden in den Zuordnungsgraphen aufgenommen, der entsprechende Teilbereich des Zuordnungsgraphen ist in Abbildung 4.2 dargestellt.

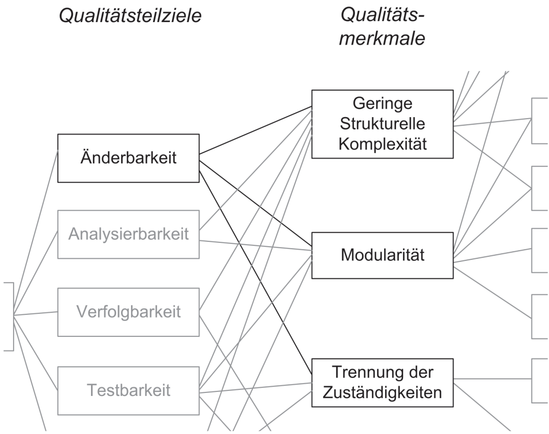


Abbildung 4.2: Qualitätsmerkmale der Änderbarkeit

Geringe Strukturelle Komplexität

Das Qualitätsteilziel *Änderbarkeit* wird durch eine *Geringe Strukturelle Komplexität* positiv beeinflusst [BCK03, FZL05]. Unter Struktureller Komplexität wird in dieser Arbeit Henderson-Sellers folgend [HS96]

[...] characteristic of software that requires effort to design, understand, or code

verstanden. Software-Systeme bestehen aus Entitäten (Komponente, Klasse, etc.) und Abhängigkeiten, welche die Strukturelle Komplexität ausmachen [Zus91, HS96]. Beispielsweise haben strukturell komplexe Komponenten meist viele Abhängigkeiten und erschweren dadurch Änderungen am System [BCK03, FZL05]. Folglich erhöht sich der Aufwand für Änderungen, da die Auswirkungen der geplanten Änderung erst bestimmt und mögliche Seiteneffekte geprüft werden müssen. Eine *Geringe Strukturelle*

Komplexität von Entitäten hat einen positiven Einfluss auf die *Änderbarkeit* und erleichtert daher die Weiterentwicklung eines Software-Systems [BCLL08].

Modularität

Unter *Modularität* wird in dieser Arbeit dem IEEE Standard für die Terminologie im Software-Engineering [IEE90] folgend verstanden als

[...] Grad der Zerlegung eines Software-Systems in einzelne Teile, sodass die Änderung eines Teils eine geringe Auswirkung auf die anderen Teile hat.

Das Qualitätsmerkmal *Modularität* hat einen positiven Einfluss auf die *Änderbarkeit* von Software-Systemen [BCLL08, BCK03, HC07], da einzelne Module unabhängig von anderen geändert werden können und so der Aufwand für die Weiterentwicklung verringert wird.

Trennung der Zuständigkeiten

Das Qualitätsmerkmal *Trennung der Zuständigkeiten* (engl. Separation of Concerns) ist auf die Arbeiten von Parnas [Par72] und Dijkstra [Dij76] zurückführbar. Nach Ossher und Tarr [OT00] können Belange (engl. Concerns) sowohl Features, Ziele, Konzepte als auch Aspekte sein. Komponenten eines Software-Systems sollten gemäß Parnas [Par72] und Dijkstra [Dij76] anhand ihrer Zuständigkeiten strukturiert werden. Wird im Entwurf beispielsweise von Features als „Zuständigkeit“ ausgegangen, sollte eine Komponente idealerweise für nur ein Feature die Verantwortung tragen. Eine gute Strukturierung der Komponenten hinsichtlich ihrer Zuständigkeiten verringert den Aufwand für Änderungen, wie etwa für die Weiterentwicklungen eines Features, da der Entwickler nicht die gesamte Struktur eines Software-Systems überschauen muss, sondern nur den abgegrenzten Teil [Ost04], in dem das Feature realisiert wurde.

4.2.2.2 Verfeinern von Analysierbarkeit

Das Ergebnis der durchgeführten Verfeinerung vom Qualitätsteilziel *Analysierbarkeit* sind die Qualitätsmerkmale *Geringe Strukturelle Komplexität* und *Modularität*. Die dabei entstandenen und in Zuordnungsgraphen aufgenommenen Beziehungen zwischen Qualitätsteilziel und Qualitätsmerkmalen sind in Abbildung 4.3 dargestellt und werden im Folgenden begründet.

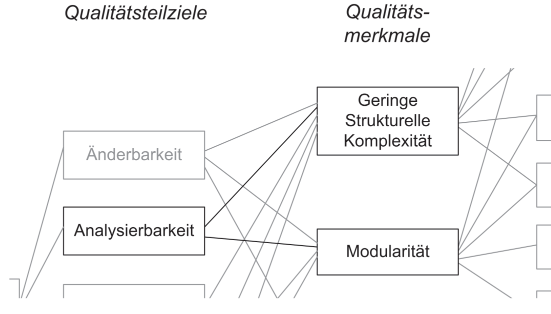


Abbildung 4.3: Qualitätsmerkmale der Analysierbarkeit

Geringe Strukturelle Komplexität

Das Qualitätsmerkmal *Geringe Strukturelle Komplexität* verbessert die Verständlichkeit der Architektur und der Implementierung und beeinflusst dadurch die *Analysierbarkeit* des Software-Systems positiv. Koschke [KP96] argumentierte sogar, dass das Verstehen des Software-Systems eine grundlegende Voraussetzung für die Analyse eines Software-Systems im Rahmen der Fehlersuche, Erweiterung oder Spezifikation ist.

Für die *Analysierbarkeit* des Software-Systems ist daher das Qualitätsmerkmal *Geringe Strukturelle Komplexität* erstrebenswert. Diese Beziehung wurde durch die Zuordnung zwischen dem Qualitätsziel *Analysierbarkeit* und dem Qualitätsmerkmal *Geringe Strukturelle Komplexität* in Abbildung 4.3 dargestellt. Beispielsweise können bei einer Geringen Strukturellen Komplexität Entitäten im Code leichter analysiert werden, wie etwa beim Diagnostizieren eines Fehlers.

Modularität

Das Qualitätsteilmerkmal *Modularität* hat einen positiven Einfluss auf die *Analysierbarkeit* von Software-Systemen [BCLL08], da einzelne Komponenten unabhängig von anderen Komponenten analysiert werden können und so der Aufwand für die Analyse verringert wird (die Analyse von Klassen schließt beispielsweise immer auch abhängige Klassen ein). Daher wird, wie bereits in Abbildung 4.3 dargestellt, das Qualitätsziel *Analysierbarkeit* durch das Qualitätsmerkmal *Modularität* verfeinert.

4.2.2.3 Verfeinern von Verfolgbarkeit

Anhand der Qualitätsmerkmale *Geringe Strukturelle Komplexität* und *Korrektheit* wird das Qualitätsteilziel *Verfolgbarkeit* verfeinert. Die erstellten und in den Zuordnungsgraphen eingeordneten Beziehungen zwischen Qualitätsteilziel und Qualitätsmerkmalen sind in Abbildung 4.4 dargestellt und werden in den folgenden Abschnitten begründet.

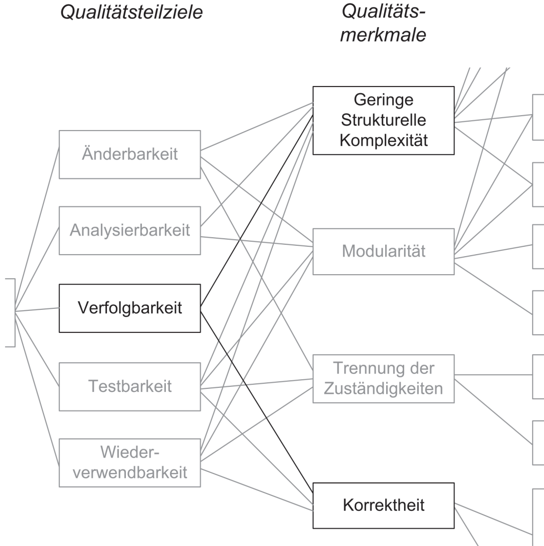


Abbildung 4.4: Qualitätsmerkmale der Verfolgbarkeit

Geringe Strukturelle Komplexität

Die *Geringe Strukturelle Komplexität* eines Software-Systems hat einen positiven Einfluss auf das Qualitätsziel *Verfolgbarkeit*. Denn der Aufwand für die *Verfolgbarkeit* sinkt, wenn beispielsweise eine geringere Anzahl von Features und deren Abhängigkeitsbeziehungen durch Traceability-Links verfolgt werden müssen. Zudem müssen bei der Weiterentwicklung des Software-Systems auch weniger Traceability-Links berücksichtigt und verwaltet werden.

Korrektheit

Die *Korrektheit* der Dokumentation, der Modelle oder auch der erstellten Traceability-Links zwischen Modellen hat ebenfalls einen positiven Einfluss auf das Qualitätsteilziel *Verfolgbarkeit*, da die Nachvollziehbarkeit und Verständlichkeit erhöht wird. Zudem sind Anforderungsänderungen einfacher durchführbar. Soll zum Beispiel durch eine Anforderungsänderung die Implementierung eines existierenden Features geändert werden, muss dessen Implementierung im Code zunächst identifiziert werden. Der zeitliche Aufwand dieser Tätigkeit wird reduziert, wenn der Entwickler korrekten Traceability-Links folgen kann, die es ermöglichen, das umgesetzte Feature zu lokalisieren[1].

4.2.2.4 Verfeinern von Testbarkeit

Das Qualitätsteilziel *Testbarkeit* wird durch die Qualitätsmerkmale *Geringe Strukturelle Komplexität*, *Modularität*, *Trennung von Zuständigkeiten* und *Korrektheit* verfeinert. Die in den Zuordnungsgraph aufgenommenen Beziehungen sind in Abbildung 4.5 dargestellt und werden in den folgenden Abschnitten begründet.

Geringe Strukturelle Komplexität

Hatton [Hat99] stellte fest, dass das Qualitätsmerkmal *Geringe Strukturelle Komplexität* einen positiven Einfluss auf die *Testbarkeit* eines Software-Systems hat. Denn je höher die Strukturelle Komplexität ist, desto höher ist auch der Aufwand für Aktivitäten im Test, da die Modelle komplexer sind oder mehr Abhängigkeiten zwischen Entitäten existieren, die vom Test-Ingenieur berücksichtigt werden müssen.

[1] Unter „Lokalisieren“ wird das Identifizieren von Entitäten verstanden, welche im Zuge der Realisierung eines Features erzeugt wurden und daher gegebenenfalls geändert werden müssen.

Schwer testbare Software-Systeme sind auch schwer zu erweitern [Fow01]. Folglich ist eine *Geringe Strukturelle Komplexität* von Entitäten eine wichtige Voraussetzung für die *Testbarkeit* eines Software-Systems und damit auch für dessen Weiterentwicklung.

Modularität

Der Aufwand von Test-Aktivitäten wird verbessert, wenn Entitäten, wie etwa Komponenten, unabhängig voneinander getestet werden können. *Modularität* hat einen positiven Einfluss auf die *Testbarkeit*, da durch modulare Komponenten die Parallelisierung von Test-Aktivitäten ermöglicht und so die Effizienz der Weiterentwicklung verbessert werden kann [BCLL08].

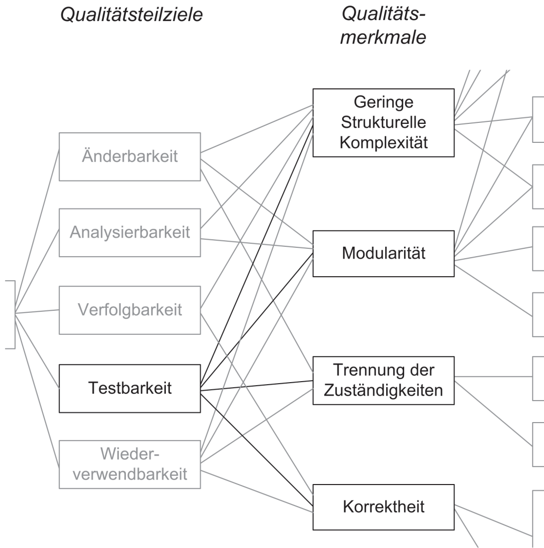


Abbildung 4.5: Qualitätsmerkmale der Testbarkeit

Trennung von Zuständigkeiten

Das Qualitätsmerkmal Trennung von Zuständigkeiten [Par72] hat ebenfalls einen positiven Einfluss auf das Qualitätsziel *Testbarkeit*, da besser nachvollziehbar ist, für welche Features eine Komponente im Software-System zuständig ist. Es reduziert sich dann in der Regel auch der Aufwand für den Test, da nur die Komponenten

geändert und damit auch getestet werden müssen, welche für ein geändertes Feature verantwortlich sind.

Korrektheit

Die Beziehung zwischen dem Qualitätsmerkmal *Korrektheit* und Qualitätsteilziel *Testbarkeit* wurde als Beispiel am Anfang des Abschnitts 4.2 bereits dargestellt.

4.2.2.5 Verfeinern von Wiederverwendbarkeit

Das Qualitätsteilziel *Wiederverwendbarkeit* wird durch die Qualitätsmerkmale *Geringe Strukturelle Komplexität*, *Modularität*, *Trennung von Zuständigkeiten* und *Korrektheit* verfeinert. Diese Beziehungen zwischen dem Qualitätsteilziel *Wiederverwendbarkeit* und den verfeinerten Qualitätsmerkmalen wurden in den Zuordnungsgraphen aufgenommen und sind als Ausschnitt des Zuordnungsgraphen in Abbildung 4.6 dargestellt. Jede dieser Zuordnungen wird in den folgenden Abschnitten begründet.

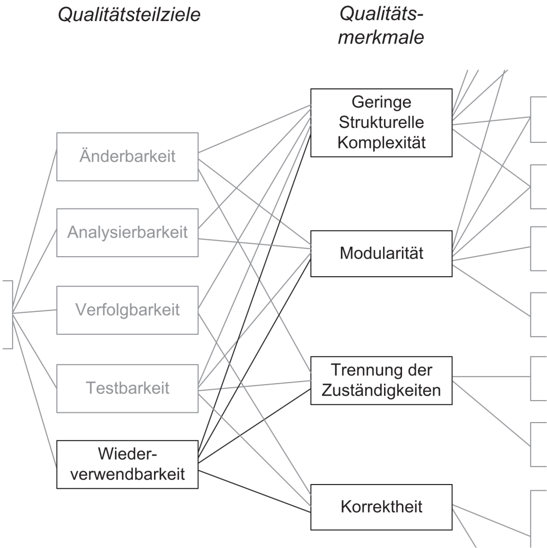


Abbildung 4.6: Qualitätsmerkmale der Wiederverwendbarkeit

Geringe Strukturelle Komplexität

Eine *Geringe Strukturelle Komplexität* von Entitäten eines Software-Systems hat einen positiven Einfluss auf deren *Wiederverwendbarkeit* [Zus91]: Denn je komplexer der Aufbau des Software-Systems, deren Spezifikation oder auch dessen Konfigurationsmöglichkeiten ist, desto schwerer kann eingeschätzt werden, ob und welche Funktion des Software-Systems wiederverwendet werden kann.

Modularität

Das Qualitätsteilmerkmal *Modularität* eines Software-Systems hat einen positiven Einfluss auf dessen *Wiederverwendbarkeit* [Mey00]. Eine Komponente des Software-Systems, die beispielsweise wenig Abhängigkeiten zu anderen Komponenten hat und allgemein nützliche Funktionen bietet, kann einfacher innerhalb eines weiteren Software-Systems integriert werden. Folglich ist der Aufwand für die *Wiederverwendbarkeit* und damit indirekt auch für die *Weiterentwicklungsfähigkeit* geringer.

Trennung von Zuständigkeiten

Die Trennung von Zuständigkeiten innerhalb eines Software-Systems wirkt sich positiv auf die *Wiederverwendbarkeit* aus. Diese Einschätzung folgt aus der Arbeit von Simao und Belchior [SB03]. Mit Bezug zur Arbeit von Sametinger [Sam97] beschreiben sie wiederverwendbare Komponenten als

[...] self-contained, clearly identifiable artifacts that describe or implement a specific function and that have clear interfaces in conformity with a given software architectural model, an appropriate documentation, and a defined degree of reuse.

Sie stellen fest, dass Komponenten von Software-Systemen, deren Aufgabe oder Zuständigkeit nicht klar identifizierbar ist, auch nicht zur Wiederverwendung ausgesucht werden können. Die Trennung von Zuständigkeiten nach Parnas [Par72] ist daher eine Voraussetzung, denn Ostermann [Ost04] folgend, sind Software-Komponenten umso wiederverwendbarer, je besser sie sich auf einzelne oder wenige Zuständigkeiten zurückführen lassen.

Korrektheit

Das Qualitätsteilmerkmal *Korrektheit* eines Software-Systems ist wichtig für dessen *Wiederverwendbarkeit* [CB91]. Ein Software-System oder eine Komponente davon ist nur dann nützlich für eine Wiederverwendung, wenn die Funktionalität weitgehend

fehlerfrei ist und die gemäß der Spezifikation versprochenen Features auch tatsächlich implementiert wurden und daher genutzt werden können. Fehler in der Implementierung oder Inkonsistenzen in der Schnittstelle einer Komponente eines Software-Systems erzeugen Aufwände, welche deren Wiederverwendung behindern.

4.3 Verfeinern zu Qualitätsteilmerkmalen

4.3.1 Ziel und Vorgehen der Aktivität

Um die Identifikation von Metriken zu erleichtern, werden Qualitätsmerkmale zu Qualitätsteilmerkmalen verfeinert. Ein Qualitätsteilmerkmal ist somit ein Ergebnis einer Verfeinerung von einem Qualitätsmerkmal. Dies ermöglicht eine fein-granulare Beschreibung eines Qualitätsmerkmals und erhöht damit die Verständlichkeit. Die Bedingung für die Verfeinerung durch den Experten ist, dass das verfeinerte Qualitätsteilmerkmal durch eine Metrik direkt bewertet werden kann. Der Experte hat dabei die Möglichkeit, die Verfeinerung eines Qualitätsmerkmals zu Qualitätsteilmerkmalen aus der Literatur zu entnehmen (hier gibt es einige verwandte Arbeiten, die dem FCM-Ansatz von McCall et al. [MRW77] folgend Qualitätsmerkmale verfeinert haben). Außerdem kann der Experte ein Qualitätsmerkmal nach eigener Erfahrung in Qualitätsteilmerkmale verfeinern.

Mit Hilfe der Arbeit von McCall et al. [MRW77] kann beispielsweise das Qualitätsmerkmal *Korrektheit* in die Qualitätsteilmerkmale *Übereinstimmung von Implementierung und Spezifikation* sowie *Vollständigkeit der Implementierung* unterteilt werden. Eine ausführliche Begründung für diese Verfeinerung folgt dann in Kapitel 4.3.2.4.

4.3.2 Ergebnisse des Verfeinerns von Qualitätsmerkmalen am Beispiel der Weiterentwicklungsfähigkeit

Ausgehend vom Qualitätsziel *Weiterentwicklungsfähigkeit* wurden für alle in der letzten Aktivität identifizierten Qualitätsmerkmale durch entsprechende Qualitätsteilmerkmale verfeinert. Die Ergebnisse der Verfeinerung werden in den folgenden Abschnitten dargestellt.

4.3.2.1 Geringe Strukturelle Komplexität

Das Ergebnis der Verfeinerung vom Qualitätsmerkmal *Geringe Strukturelle Komplexität* zu Qualitätsteilmerkmalen wurde mit Hilfe des Zuordnungsgraphen dokumentiert und ist in Abbildung 4.7 zu sehen. Die Qualitätsteilmerkmale *Geringe Steuerflusskomplexität*, *Lose Kopplung*, *Zyklenfreiheit*, *Hohe Kohäsion* und *Größenausgewogenheit* verfeinern das Qualitätsmerkmal *Geringe Strukturelle Komplexität*.

Geringe Steuerflusskomplexität

Die Steuerflusskomplexität ist ein Qualitätsteilmerkmal, welches nach der Arbeit von McCabe [McC76] vor allem das Verstehen und Überschauen des betroffenen Codes beeinflusst, sie sollte möglichst gering sein. Eine Erhöhung der Steuerflusskomplexität durch verschachtelte Algorithmen erhöht die Strukturelle Komplexität. Im Umkehrschluss ist daher das Qualitätsteilmerkmal *Geringe Steuerflusskomplexität* für das Qualitätsmerkmal *Geringe Strukturelle Komplexität* erstrebenswert.

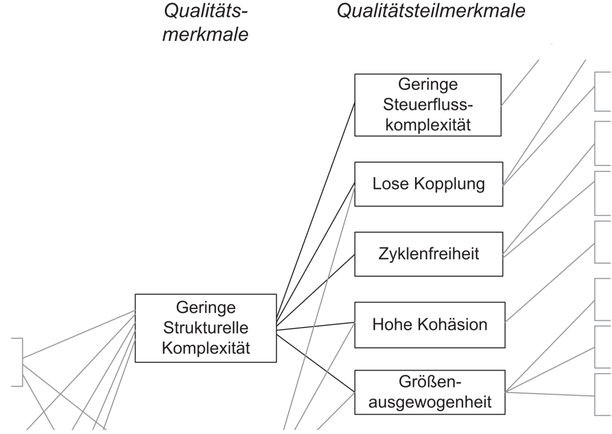


Abbildung 4.7: Qualitätsteilmerkmale der Geringen Strukturellen Komplexität

Lose Kopplung

Die Stärke der Kopplung beeinflusst die Strukturelle Komplexität [SMC79, CG90, Fen95]. Unter Kopplung wird die Stärke der Verbindung zwischen zwei Komponen-

ten verstanden [SMC79]. Eine hohe Kopplung von Klassen hat beispielsweise eine negative Auswirkung auf die Strukturelle Komplexität. Je höher die Kopplung ist, desto höher ist die Strukturelle Komplexität, daher ist eine *Lose Kopplung* erstrebenswert.

Zyklenfreiheit

Die Strukturelle Komplexität einer Komponente eines Software-Systems wird durch Zyklen in den Abhängigkeiten zwischen Entitäten wie etwa Klassen erhöht, da die beteiligten Komponenten schwerer zu verstehen sind [Lil08]. Es wurde bereits von Parnas [Par78] gezeigt, dass das Qualitätsteilmerkmal *Zyklenfreiheit* für die Qualität eines Software-Systems entscheidend ist [Par78], da sie die Strukturelle Komplexität verringert. Demnach ist die *Zyklenfreiheit* ein Qualitätsteilmerkmal des Qualitätsmerkmals *Geringe Strukturelle Komplexität.*

Hohe Kohäsion

Unter Kohäsion wird Stevens folgend

[...] der Zusammenhalt von Elementen einer Klasse [SMC79]

verstanden. Je höher die Kohäsion einer Klasse ist, desto höher ist auch der funktionale Zusammenhalt und desto niedriger die Strukturelle Komplexität, da alle relevanten Funktionen an einer Stelle zu finden sind [BDW98]. Für eine *Geringe Strukturelle Komplexität* ist daher die *Hohe Kohäsion* ein Qualitätsteilmerkmal.

Größenausgewogenheit

Lilienthal [Lil08] folgend ist die *Größenausgewogenheit* von Entitäten ein weiteres wichtiges Qualitätsteilmerkmal, welches die Strukturelle Komplexität beeinflusst: Entitäten sollten eine ähnliche Größe haben und auch aus vergleichbar vielen Entitäten bestehen, anderenfalls erhöhen sie die Strukturelle Komplexität.

4.3.2.2 Modularität

Das Qualitätsmerkmal *Modularität* lässt sich durch viele verschiedene Qualitätsteilmerkmale verfeinern; eine gute Schnittmenge aus der Literatur [MRW77, DS92, EL96, Lil08] sind *Lose Kopplung* und *Hohe Kohäsion*, *Größenausgewogenheit*, *Datenkapselung* (auch bekannt als Information Hiding nach Parnas [Par72]) und *Schnittstellen-*

abstraktion. Abbildung 4.8 zeigt einen Teil des erstellten Zuordnungsgraphen, der die Verfeinerung des Qualitätsmerkmals *Modularität* dokumentiert. Die in den Zuordnungsgraphen aufgenommenen Beziehungen zwischen dem Qualitätsmerkmal *Modularität* und den Qualitätsteilmerkmalen werden im Folgenden begründet.

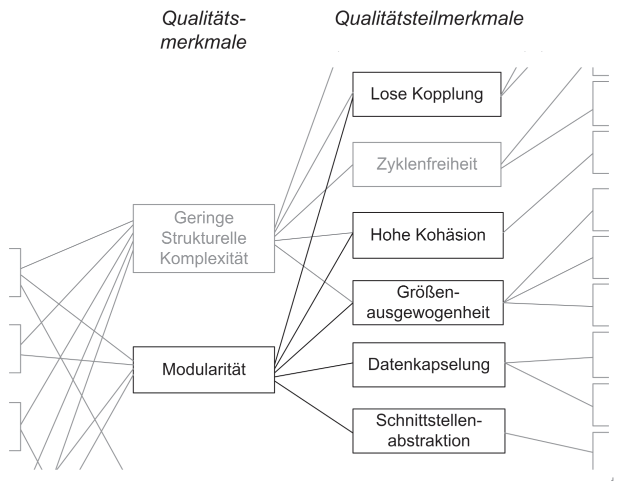


Abbildung 4.8: Qualitätsteilmerkmale der Modularität

Lose Kopplung und Hohe Kohäsion

Nach Fenton [Fen95] sind die Qualitätsteilmerkmale *Lose Kopplung* und *Hohe Kohäsion* als Grundlage für das Qualitätsmerkmal *Modularität* erstrebenswert. Dieser Aussage folgend wurden die Qualitätsteilmerkmale *Lose Kopplung* und Hohe *Kohäsion* als Verfeinerung vom Qualitätsmerkmal *Modularität* eingeführt.

Größenausgewogenheit

Das Kriterium *Größenausgewogenheit* ist nach Lilienthal [Lil08] essentiell für das Qualitätsteilmerkmal *Modularität*. Lilienthal geht davon aus, dass Software-Systeme eine ausgewogene *Modularität* haben, wenn Komponenten eine ähnliche Größe haben bzw. aus vergleichbar vielen Entitäten aufgebaut sind. Daher ist die *Größenausgewogenheit* ein weiteres Qualitätsteilmerkmal des Qualitätsmerkmals *Modularität*.

Datenkapselung

Die *Datenkapselung* verhindert den Zugriff auf Daten, die nicht öffentlich sind [Par72]. Damit sind nur die Teile der Implementierung einer Klasse sichtbar, welche für die Zusammenarbeit mit anderen Klassen notwendig sind. Dadurch wird die *Modularität* der Klassen verbessert. Denn folglich müssen bei Änderungen nur die sichtbaren Teile berücksichtigt werden, was den Aufwand für Änderungen verringert.

Schnittstellenabstraktion

Die *Schnittstellenabstraktion* erfordert den Einsatz von Schnittstellen zur Abstraktion der Implementierung [Sie05]. *Schnittstellenabstraktion* ist daher eine mögliche Art der Abstraktion, die in [Wik10] definiert ist als

Weglassen von Einzelheiten und des Überführens auf etwas Allgemeineres oder Wesentliches.

Speziell für die *Modularität* ist der Einsatz von Schnittstellen im Sinne einer *Schnittstellenabstraktion* wichtig: sie bilden die Abstraktion eines modularen Teils des Software-Systems ab und verbergen die Implementierung [DS92]. Dadurch können Wechselwirkungen zwischen Komponenten von Software-Systemen anhand ihrer Schnittstelle fixiert und so die *Modularität* verbessert werden.

4.3.2.3 Trennung der Zuständigkeiten

Das Qualitätsmerkmal *Trennung der Zuständigkeiten* wurde durch zwei Qualitätsteilmerkmale verfeinert. In Abbildung 4.9 ist die mit Hilfe des Zuordnungsgraphen dokumentierte Verfeinerung zu sehen. Sowohl eine *Geringe Feature-Streuung* als auch eine *Geringe Feature-Verschränkung* haben einen positiven Einfluss auf die *Trennung der Zuständigkeiten*, wie auch in [BR08a, BBR09] argumentiert wurde.

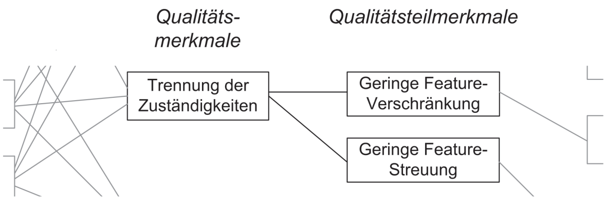


Abbildung 4.9: Qualitätsteilmerkmale der Trennung der Zuständigkeiten

In Software-Systemen lässt sich die Implementierung eines Features jedoch nicht immer in einer Komponente realisieren, obwohl dies aus Sicht der *Trennung der Zuständigkeiten* ideal wäre (siehe Abbildung 4.9). Vielmehr hat ein einzelnes Feature Einfluss auf mehrere Komponenten, dabei können im Allgemeinen zwei Abhängigkeitsstrukturen, Feature-Streuung (engl. Scattering of Feature) und Feature-Verschränkung (engl. Tangling of Features), unterschieden werden [Soc06, vdBCH06]. Für eine gute Weiterentwicklung sollten diese Abhängigkeitsstrukturen im Software-System möglichst gering ausgeprägt sein.

Geringe Feature-Streuung

Feature-Streuung beeinträchtigt das Qualitätsmerkmal *Trennung der Zuständigkeiten*, da ein Feature f_1 in vielen Komponenten a_1, a_2, ... a_n umgesetzt ist (siehe Abbildung 4.10). Dadurch wird das Prinzip einer einzigen Verantwortung verletzt [Par72]. Bei der Weiterentwicklung müssen alle abhängigen Komponenten und ihre Wechselwirkungen betrachtet werden, was den Aufwand für die Weiterentwicklung des Systems erhöht.

Geringe Feature-Verschränkung

Feature-Verschränkung beeinträchtigt das Qualitätsmerkmal Trennung von Zuständigkeiten, da eine Komponente für mehrere Features verantwortlich ist, beispielsweise ist sowohl Feature f_1 als auch Feature f_2 in der Komponente a_1 implementiert, wie in Abbildung 4.10 zu sehen ist (eine ausführliche Beschreibung der Entitäten finden sich im Anhang B.2). Erfüllt eine Komponente mehrere Verantwortungen, kann die Implementierung eines Features schwerer identifiziert werden, die Komponente ist schwerer verständlich, erweiterbar und wiederverwendbar. Letztlich erhöht sich der Aufwand für die Weiterentwicklung.

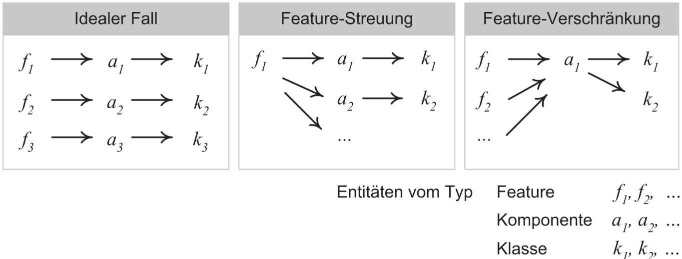


Abbildung 4.10: Idealer Fall, Feature-Streuung und Feature-Verschränkung.

4.3.2.4 Korrektheit

Auf Grundlage der Arbeit der Autoren McCall et al. [MRW77] wird die *Korrektheit* in die Qualitätsteilmerkmale *Übereinstimmung von Implementierung und Spezifikation* und *Vollständigkeit der Implementierung* verfeinert. Diese Verfeinerung wurde in den Zuordnungsgraphen aufgenommen, wie in Abbildung 4.11 zu sehen ist. Die Verfeinerung in diese Qualitätsteilmerkmale ist für das Erkennen von Qualitätsdefiziten bezüglich des Qualitätsmerkmals *Korrektheit* zielführend, was im Folgenden begründet wird.

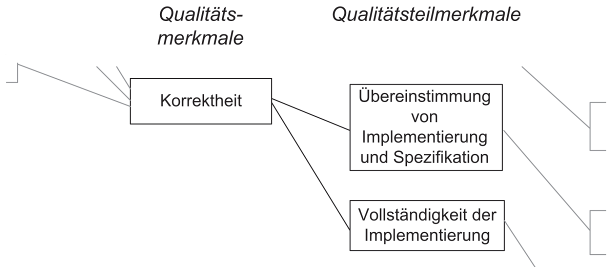


Abbildung 4.11: Verfeinerung des Qualitätsteilziels Korrektheit

Übereinstimmung von Implementierung und Spezifikation

McCall et al. stellen in ihrer Arbeit [MRW77] fest, dass die *Übereinstimmung von Implementierung und Spezifikation* ein wichtiges Kriterium zur Überprüfung des Qualitätsmerkmals *Korrektheit* ist.

Die existierende Implementierung soll daraufhin überprüft werden, ob sie zur Spezifikation passt. Beispielsweise wird beim Reengineering von einem existierenden Software-System ausgegangen, dessen Implementierung verbessert werden soll. Wenn die Spezifikation fehlt, kann die *Korrektheit* der Implementierung nicht überprüft werden [Lig02]. Daher wird angelehnt an die Arbeit von McCall et al. [MRW77] die *Übereinstimmung von Implementierung und Spezifikation* als ein Qualitätsteilmerkmal vom Qualitätsmerkmal *Korrektheit* eingeführt.

Vollständigkeit der Implementierung

Ausgehend von der Spezifikation, wie etwa eines Features, kann die *Vollständigkeit der Implementierung* geprüft werden. Vollständig bedeutet daher, dass die Implementierung eines Software-Systems alle geforderten Funktionalitäten bereitstellt [MRW77]. Jedes spezifizierte Feature sollte daher auch tatsächlich implementiert sein, anderenfalls ist die *Korrektheit* der Implementierung nicht gewährleistet. Daher wird angelehnt an die Arbeit von McCall et al. die *Vollständigkeit der Implementierung* als ein Qualitätsteilmerkmal des Qualitätsmerkmals *Korrektheit* eingeführt.

4.4 Zusammenfassung

Die Analyse des Qualitätsziels *Weiterentwicklungsfähigkeit* gemäß dem Aufbauprozess, der in Kapitel 3 eingeführt wurde, ist mit dem Verfeinern zu Qualitätsteilmerkmalen komplett. Während der Analyse des Qualitätsziels *Weiterentwicklungsfähigkeit* wurde schrittweise jedes verfeinerte Qualitätsteilziel, Qualitätsmerkmal und Qualitätsteilmerkmal mit Hilfe des Zuordnungsgraphen dokumentiert.

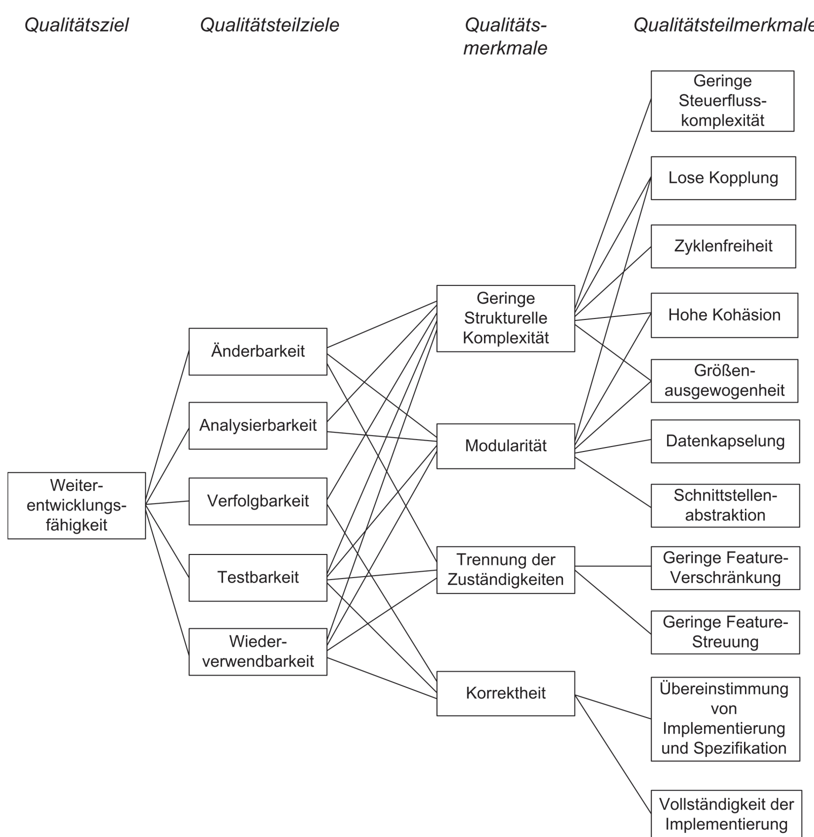


Abbildung 4.12: Der Zuordnungsgraph nach der Analyse des Qualitätsziels Weiterentwicklungsfähigkeit.

Der entsprechende Teil des dabei entstandenen Zuordnungsgraphen ist in Abbildung 4.12 dargestellt. Jedes Qualitätsteilziel ist dort durch mehrere Qualitätsmerkmale verfeinert worden, da sich die Qualitätsmerkmale auf verschiedene Qualitätsteilziele positiv auswirken.

Wie in den letzten Abschnitten beschrieben, konnten anhand existierender Arbeiten alle Qualitätsmerkmale durch mehrere Qualitätsteilmerkmale verfeinert werden. Der Zuordnungsgraph in Abbildung 4.12 zeigt das Ergebnis dieser Verfeinerung am Beispiel des Qualitätsziels *Weiterentwicklungsfähigkeit*. Damit sind die Zusammenhänge

zwischen den verfeinerten Qualitätsmerkmalen und Qualitätsteilmerkmalen anhand des Zuordnungsgraphs leichter nachvollziehbar. Zum Beispiel haben die Qualitätsmerkmale *Geringe Strukturelle Komplexität* und *Modularität* mehrere gemeinsame Qualitätsteilmerkmale. Folglich führt die Verbesserung solcher Qualitätsteilmerkmale, wie etwa *Lose Kopplung* und *Hohe Kohäsion*, zu einer Verbesserung beider Qualitätsmerkmale. Andere Qualitätsmerkmale, wie etwa *Trennung der Zuständigkeiten* und *Korrektheit*, haben keine gemeinsamen Qualitätsteilmerkmale.

Der Zuordnungsgraph in Abbildung 4.12 bildet weiter die Grundlage für eine gesonderte Gewichtung der erstellten Beziehungen zwischen dem Qualitätsziel *Weiterentwicklungsfähigkeit* und den verfeinerten Qualitätsteilzielen, Qualitätsmerkmalen und Qualitätsteilmerkmalen. Darauf aufbauend ist eine Priorisierung der Reengineering-Rezepte möglich, die einem Qualitätsteilmerkmal zugeordnet sind (siehe Kapitel 7). Die Erstellung der Reengineering-Rezepte wird in dem folgenden Kapitel 5 beschrieben.

Kapitel 5

Erstellen der Reengineering-Rezepte

Das Erstellen der Reengineering-Rezepte wird auf Grundlage von verfeinerten Qualitätsteilmerkmalen durchgeführt mit dem Ziel, Qualitätsdefizite zu erkennen und zu beheben. Zu jedem Qualitätsteilmerkmal sollte mindestens ein Reengineering-Rezept erstellt werden. Das Erstellen eines Reengineering-Rezepts unterteilt sich gemäß dem Aufbauprozess in Kapitel 3.1 in die Aktivitäten „Ermitteln der Qualitätsdefizite“, „Auffinden der Metriken und Regeln zur Erkennung von Qualitätsdefiziten“ und „Bestimmen der Reengineering-Regeln zur Behebung von Qualitätsdefiziten“, welche im Folgenden detailliert beschrieben werden. Die innerhalb dieser Aktivitäten gewonnenen Ergebnisse werden mit Hilfe des Zuordnungsgraphen zwischen Qualitätsziel und Reengineering-Aktivitäten schrittweise dokumentiert.

Neben dem allgemeinen Vorgehen in den Aktivitäten werden auch die Ergebnisse dargestellt, die während des Aufbauprozesses für das Qualitätsziel *Weiterentwicklungsfähigkeit* erzielt wurden.

5.1 Ermitteln der Qualitätsdefizite

5.1.1 Ziel und Vorgehen der Aktivität

Qualitätsdefizite sind Abweichungen zu einem gewünschten Qualitätsteilmerkmal und beeinträchtigen daher die Qualität eines Software-Systems. Die Erkennung und Behebung von Qualitätsdefiziten durch Reengineering-Rezepte führt damit zu einer Verbesserung der Software-Qualität. Für das Erstellen der Reengineering-Rezepte müssen also zunächst mögliche Qualitätsdefizite ermittelt werden.

Das Ermitteln der Qualitätsdefizite wird auf Grundlage der bereits bekannten Qualitätsteilmerkmale ermöglicht. Beispielsweise ist das Qualitätsdefizit *Zyklen zwischen Klassen* eine bekannte Abweichung zu dem Qualitätsteilmerkmal *Zyklenfreiheit*. Dieses Qualitätsdefizit wurde bereits gut erforscht, wie etwa durch Lilienthal [Lil08]. Lilienthal betrachtet in ihrer Arbeit ein weiteres Qualitätsdefizit, *Zyklen zwischen Komponenten*, welches auch das Qualitätsteilmerkmal *Zyklenfreiheit* beeinträchtigt und durch Abhängigkeitsbeziehungen hervorgerufen wird. Nach Lilienthal beeinträchtigen beide Arten von Zyklen die Architektur eines Software-Systems und folglich auch dessen Implementierung.

Es gibt verschiedene Qualitätsdefizite die ein Qualitätsteilmerkmal beeinträchtigen, da es unterschiedliche Ursachen für eine Abweichung zu einem Qualitätsteilmerkmal gibt, die somit auch gesondert als Qualitätsdefizite benannt und erkannt werden müssen. Dies zeigt sich nicht nur beim Qualitätsteilmerkmal *Zyklenfreiheit*, welches von Lilienthal untersucht wurde, sondern auch bei weiteren, wie etwa *Lose Kopplung* (siehe Abschnitt 5.4.3.1).

Für die Ermittlung von Qualitätsdefiziten kann der Experte entweder auf in der Literatur existierende Qualitätsdefizite zurückgreifen oder Qualitätsdefizite aus seiner Erfahrung benennen. Jedes Qualitätsdefizit wird zur Präzisierung in Tabellenform dokumentiert. Als Beispiel kann Tabelle 5.1 dienen: Es werden der Name des Qualitätsdefizits, eine kurze Beschreibung und das zugeordnete Qualitätsteilmerkmal angegeben.

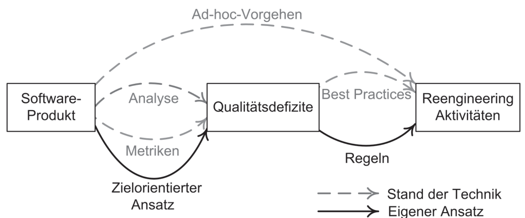


Abbildung 5.1: Zielorientierter Ansatz zum Erkennen und Beheben von Qualitätsdefiziten.

Durch das vorgelagerte zielorientierte Verfeinern von Qualitätszielen zu Qualitätsteilmerkmalen wird die Möglichkeit geschaffen, Abweichungen zu einem Qualitätsteilmerkmal möglichst objektiv als Qualitätsdefizit zu erkennen. Abbildung 5.1 zeigt den Unterschied zum Stand der Technik, da bei dem zielorientierten Ansatz der Arbeit Qualitätsdefizite eine zentrale Rolle einnehmen. Im Vergleich zu den existierenden Ansätzen, wie etwa dem Ad-hoc-Vorgehen oder einem rein auf Metriken basierenden Ansatz, werden Regeln aufgestellt, in denen definiert ist, wann ein Qualitätsdefizit vorliegt und mit welchen Reengineering-Aktivitäten es behoben werden kann.

Als Erweiterung von bisherigen Arbeiten werden in dieser Arbeit Qualitätsdefizite umfassender betrachtet. Denn Qualitätsdefizite berühren unterschiedliche Teile eines Software-Systems, wie etwa die Spezifikation, die Architektur und/oder die Implementierung. Die bisher in Kapitel 2 genannten Typen von Qualitätsdefiziten berücksichtigen keine Abhängigkeitsbeziehungen zwischen Entitäten der Spezifikation und der Architektur oder der Implementierung eines Software-Systems. Dafür wird in dieser Arbeit der Begriff Mehr-Ebenen-Mangel eingeführt:

Ein Mehr-Ebenen-Mangel ist ein über mehrere Ebenen eines Software-Systems auftretendes Qualitätsdefizit, beispielsweise zwischen Spezifikation und Implementierung, welches verbessert werden sollte.

Durch die Auswertung von Abhängigkeitsbeziehungen zwischen Entitäten der Spezifikation, wie etwa Features und deren Implementierung, wie etwa Komponenten, wird dieser Typ von Qualitätsdefizit erkennbar. Ähnlich zu Code-Smell oder Architekturmangel liegt bei einem Mehr-Ebenen-Mangel nicht unbedingt ein Fehlverhalten vor, vielmehr beeinträchtigen diese Qualitätsdefizite die Software-Qualität. Ein Beispiel für einen Mehr-Ebenen-Mangel, der die *Weiterentwicklungsfähigkeit* eines Software-

Systems erschwert, ist *Fehlende Feature-Spezifikation* (siehe Kapitel 5.4.2).

5.1.2 Ergebnisse des Ermittelns von Qualitätsdefiziten am Beispiel der Weiterentwicklungsfähigkeit

In dieser Arbeit wurden beispielhaft 14 Qualitätsdefizite ermittelt, welche das Qualitätsziel *Weiterentwicklungsfähigkeit* beeinträchtigen. Sie sind im Anhang in Abschnitt A.1 ausführlich beschrieben. Einige Beispiele sind in Tabelle 5.1 zu finden.

Dort wird jeweils ein Qualitätsdefizit vorgestellt, welches innerhalb der Implementierung (z.B. *Tiefe Vererbungshierarchie*) vorkommen kann, Auswirkungen auf die Architektur (z.B. *Zyklen zwischen Komponenten*) hat oder nur durch Betrachtung von Entitäten der Spezifikation (z.B. *Starke Feature-Verschränkung*) erkannt werden kann. Diese drei Beispiele von Qualitätsdefiziten werden in den folgenden Abschnitten immer wieder benutzt.

Wie bereits am Anfang des Kapitels erläutert wurde, stellt jedes Qualitätsdefizit eine Abweichung zu einem Qualitätsteilmerkmal dar. Die in diesem Schritt des Aufbauprozesses ermittelten Informationen wie Name, Beschreibung und Abweichung zu Qualitätsteilmerkmal reichen nicht für eine objektive Erkennung von Qualitätsdefiziten aus. Deswegen wird das Erkennen von Qualitätsdefiziten in dieser Arbeit durch Metriken und Regeln definiert, die dann auch innerhalb von Werkzeugen eingesetzt werden können. Der Zusammenhang zwischen Qualitätsdefiziten, Metriken und Regeln wird im folgenden Abschnitt 5.2 dargestellt.

Tabelle 5.1: Beispiele für ermittelte Qualitätsdefizite.

Name	Beschreibung	Abweichung zu Qualitäts-teilmerkmal
Tiefe Vererbungs-hierarchie [CK94]	Unter einer starken Vererbungshierarchie wird eine zu hohe Vererbungstiefe zwischen Klassen verstanden. Klassen, die tief in der Hierarchie angesiedelt sind, sind schwer zu verstehen, da sie viele Merkmale erben [CK94].	Lose Kopplung
Zyklen zwischen Komponenten [Lil08]	Zyklen zwischen Komponenten erhöhen die Strukturelle Komplexität der Architektur [Lil08]. Außerdem behindern Zyklen die Weiterentwicklung, da die beteiligten Komponenten fehleranfälliger sind und neue Komponenten meist an die vorhandenen Zyklen angebaut werden [Lil08].	Zyklen-freiheit
Starke Feature-Verschrän-kung in Komponenten	Bei starker Feature-Verschränkung existieren Abhängigkeitsstrukturen, welche das Verstehen der Architektur und des Codes hinsichtlich der spezifizierten Features verschlechtern. Nach Arndt et al. [AHKP09] steigt in diesem Fall auch die Gefahr von Programmfehlern, da die Struktur und Kontrollflusslogik schwer erschließbar sind.	Geringe Feature-Verschrän-kung

5.2 Auffinden der Metriken und Aufstellen der Erkennungsregeln

5.2.1 Ziel und Vorgehen der Aktivität

Für eine objektivere Bewertung von Qualitätsteilmerkmalen werden Metriken eingesetzt. Diese Beziehung zwischen Qualitätsteilmerkmalen und Metriken wird durch den Zuordnungsgraphen dokumentiert, wie in Abbildung 5.2 anhand des in dieser Aktivität erstellten Teils des Zuordnungsgraphen ersichtlich ist. Zusätzlich werden Regeln eingesetzt, um die im vorherigen Schritt des Aufbauprozesses ermittelten Qualitätsdefizite anhand der durch Metriken ermittelten Ergebnisse objektiv erkennen zu können. Diese Beziehung zwischen Metriken und Qualitätsdefiziten ist ebenfalls durch den Zuordnungsgraphen dokumentiert, wie auch in Abbildung 5.2 zu sehen.

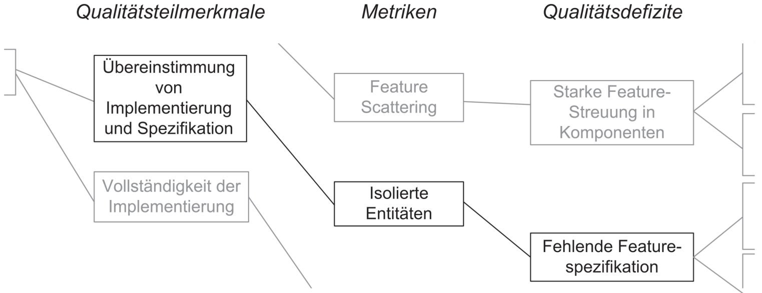


Abbildung 5.2: Beispiel von Zuordnungen zwischen Qualitätsteilmerkmal, Metrik und Qualitätsdefizit.

Das Auffinden der Metriken ist im Abschnitt 5.2.2 und das Aufstellen von Erkennungsregeln, in denen Metriken integriert sind, ist in Abschnitt 5.2.3 beschrieben.

5.2.2 Auffinden von Metriken

Um Metriken für eine objektive Bewertung von Qualitätsteilmerkmalen zu finden, gibt es zwei Möglichkeiten. Die eine Möglichkeit besteht darin, bewährte Metriken aus der Literatur wiederzuverwenden, sofern sie zur Bewertung der Qualitätsteilmerkmale

eingesetzt werden können. Die andere Möglichkeit ist, neue Metriken aufzubauen, wenn keine zum Qualitätsteilmerkmal passenden Metriken existieren. In beiden Fällen sind Metriken ein Mittel zum Zweck, um Qualitätsdefizite erkennen zu können.

Die Erkennung von Qualitätsdefiziten, welche das Qualitätsteilmerkmal *Übereinstimmung von Implementierung und Spezifikation* beeinträchtigen, soll im Folgenden exemplarisch vorgestellt werden: eine *Übereinstimmung von Implementierung und Spezifikation* ist dann vorhanden, wenn jeder Entität der Implementierung eine Entität der Spezifikation zugeordnet werden kann. Wenn dies nicht möglich ist, existierten Abweichungen zum betrachteten Qualitätsteilmerkmal. Beispiele für Entitäten der Implementierung sind Komponenten, Klassen oder auch Schnittstellen.

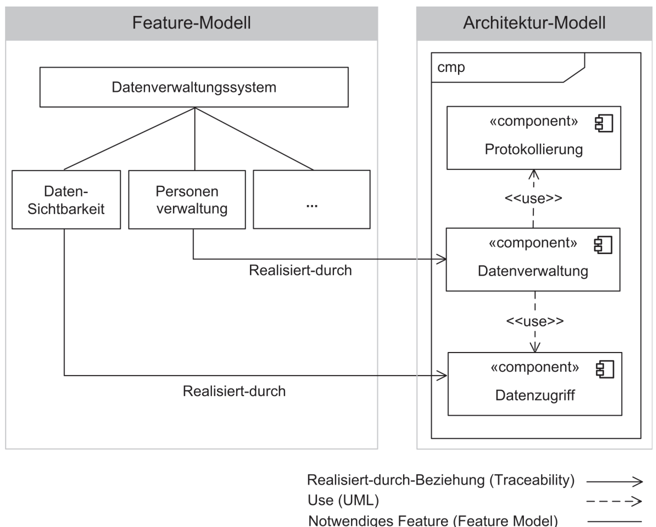


Abbildung 5.3: Ausschnitt von Features und Komponenten aus der Fallstudie.

Das Software-System in Abbildung 5.3 stellt einen vereinfachten Ausschnitt aus der Fallstudie dar, welche in Kapitel 7 eingeführt wird. Auf diesen Ausschnitt wird in den weiteren Beispielen, die das Erstellen eines Reengineering-Rezepts verdeutlichen, häufig Bezug genommen.

In dem Beispiel in Abbildung 5.3 sind Komponenten und ihre Abhängigkeiten zu sehen, wie etwa die Komponente Datenverwaltung, die die Komponenten Protokollierung

und Datenzugriff benutzt. Architekturmodelle werden häufig als UML-Diagramme dargestellt, wie es auch in Abbildung 5.3 der Fall ist.

Beispiele für Entitäten der Spezifikation sind Feature-Spezifikationen, die in Abbildung 5.3 als Features Datensichtbarkeit und Personenverwaltung durch ein Feature-Modell nach [KCH+90] visualisiert sind.

Zudem sind verschiedene Abhängigkeitsbeziehungen in Abbildung 5.3 durch die jeweiligen Modelle vorgegeben. UML stellt beispielsweise die Benutzt-Beziehung zur Verfügung, die mit Hilfe des Schlüsselworts use im Diagramm angezeigt wird. Eine Abhängigkeitsbeziehung zwischen den Entitäten Features und Komponenten wird durch einen Traceability-Link vom Typ Realisiert-durch aufgebaut, wie in Abbildung 5.3 zwischen dem Feature Personenverwaltung und der Komponente Datenverwaltung zu sehen ist. Der Traceability-Link Realisiert-Durch ist im Anhang in Abschnitt B.2.3 beschrieben.

Die im Rahmen der Arbeit neu erstellte Metrik *Isolierte Entitäten* $\mathit{IE}(X)$ ist im Anhang in Abschnitt C.2.1 definiert und wird auf das Software-System in Abbildung 5.3 beispielhaft angewandt. Bei dieser Metrik sind beliebige Entitäten (symbolisiert durch die Menge X) isoliert, wenn sie existieren aber nicht an ein Feature gebunden sind. Sie haben deswegen auch keine Berechtigung, umgesetzt zu werden.

Die Metrik $\mathit{IE}(X)$ wird auf eine Menge von Entitäten angewandt. X steht für die Menge aller Entitäten (alle Entitätstypen). Soll die Metrik nur auf eine bestimmte Menge angewandt werden, werden folgende Mengenbezeichner verwendet: die im Beispiel in Abbildung 5.3 dargestellten Komponenten der Architektur, werden mit dem Mengenbezeichner A bezeichnet, Schnittstellen mit S, Klassen mit K und Datendateien mit D.

Wenn alle Komponenten mit einer Abhängigkeitsbeziehung zu Features in Verbindung stehen, soll das Ergebnis der Metrik $\mathit{IE}(A)$ gleich 0 sein. Anderenfalls soll das Ergebnis der Metrik größer als 0 sein und die Anzahl von sogenannten *Isolierten Entitäten* berechnet werden. Wird beispielsweise die Metrik *Isolierte Entitäten* $\mathit{IE}(A)$ auf das in Abbildung 5.3 dargestellte Software-System angewandt, kommt als Ergebnis von $\mathit{IE}(A)$ der Wert 1 heraus, da die Komponente Protokollierung isoliert ist.

5.2.3 Aufstellen von Erkennungsregeln

Eine Erkennungsregel definiert die Erkennung von Qualitätsdefiziten auf Grundlage von Metriken. Die Beziehung zwischen Metriken und Qualitätsdefiziten wurde bereits im vorhergehenden Schritt identifiziert und ist beispielhaft in Abbildung 5.2 zu sehen. Für das Aufstellen von Erkennungsregeln werden Metriken ausgewählt und z.B. Schwellwerte für die Interpretation der Ergebnisse einer Metrik festgelegt, um so die Eigenschaften des Software-Systems bestimmen zu können. Dazu können existierende Werte aus der Literatur wiederverwendet werden oder der Experte muss anhand seiner Erfahrung diese festlegen. In beiden Fällen ist es möglich, die im Aufbauprozess bestimmten Werte innerhalb des Anwendungsprozesses anzupassen, da in der Regel Eigenschaften eines Software-Systems im Voraus nicht vorhersehbar sind.

Beispielsweise kann ein Schwellwert für die Größe von Klassen, gemessen in ***SLOC*** (Source Lines of Code), in Software-Systemen aus empirischen Fallstudien ermittelt werden [BBM96]. Werden aber in einem konkreten Software-System Klassen beispielsweise durch externe Produkte generiert, kann die Größe der generierten Klassen meist nicht oder nur wenig beeinflusst werden. Daher sollte es möglich sein, den für die Größe der generierten Klassen festgelegten Schwellwert je nach Herkunft der Klassen innerhalb der Erkennungsregeln anzupassen. In einer Erkennungsregel sollten außerdem verschiedene Metriken verknüpft einsetzbar sein, da es Qualitätsdefizite gibt, die nur durch den Einsatz von mehreren Metriken erkannt werden können.

5.2.3.1 Struktur von Erkennungsregeln

Eine Erkennungsregel besteht aus einem Qualitätsdefizit und Bedingungen, welche die Erkennung des Qualitätsdefizits definieren. Mit Hilfe der Backus-Naur-Form (BNF) aus [BBG+97] wird in Tabelle 5.2 die Struktur einer Erkennungsregel genauer beschrieben. Beim Aufstellen einer Erkennungsregel können insbesondere die Bedingungen, wann ein Qualitätsdefizit vorliegt, im Detail formuliert werden.

Bedingungen: Bedingungen bestehen aus Metriken, Operatoren und Schwellwerten. Die Ergebnisse von Metriken können mit Hilfe von Relationalen Operatoren mit Schwellwerten verglichen werden, um beispielsweise zu überprüfen, ob das Ergebnis einer Metrik größer als 0 ist. Weitere Operatoren dienen dazu, eine Folge von Bedingungen zu definieren, wenn beispielsweise mehr als eine Metrik innerhalb der

Erkennungsregel berücksichtigt werden soll. Es können auch Vorbedingungen mit Hilfe des Schlüsselwortes `VORB` innerhalb der Erkennungsregeln festgehalten werden.

Operatoren: In Erkennungsregeln werden zwei Arten von Operatoren unterschieden. Ein Relationaler Operator wird für das Vergleichen von Ergebnissen einer Metrik verwendet. Ein Logischer Operator dient zur Auswahl von Metriken. Beispielsweise werden die Logischen Operatoren `UND` und `ODER` zur Auswahl von Metriken innerhalb von Bedingungen verwendet. Zudem werden Relationale Operatoren, wie etwa „größer als“, symbolisiert durch „>“, zum Vergleichen von Metrik-Ergebnissen mit Schwellwerten eingesetzt.

Tabelle 5.2: Beschreibung der Struktur von Erkennungsregeln.

```
<Erkennungsregel> ::= <Qualitätsdefizit> ':=' <Vorbedingung>
                      <Folge von Bedingungen> '.'

<Qualitätsdefizit> ::= 'Name eines Qualitätsdefizits'

<Vorbedingung> ::= 'VORB' '[' <Folge von Bedingungen> ']' |

<Folge von Bedingungen> ::= <Bedingung> | <Bedingung><Logischer
                             Operator><Folge von Bedingungen> |

<Bedingung> ::= <Metrik><Relationaler Operator><Schwellwert>

<Metrik> ::= 'Name einer Metrik'

<Schwellwert> ::= 'Zahl, die Vergleichen dient'

<Relationaler Operator> ::= '<' | '≤' | '=' | '≥' | '>' | '≠'

<Logischer Operator> ::= 'UND' | 'ODER'
```

Die „Detection Strategies“ nach Marinescu [MR04] haben einen ähnlichen Aufbau wie die hier vorgestellten Erkennungsregeln, berücksichtigen aber keine Vorbedingungen und zielen nur auf das Erkennen von Code-spezifischen Problemen ab (vergleiche dazu

auch Abschnitt 2.2).

5.2.3.2 Beispiel einer Erkennungsregel

Das Software-System in Abbildung 5.3 dient wieder als Beispiel, um eine Erkennungsregel zu verdeutlichen. Die in Abschnitt 5.2.2 eingeführte Metrik *Isolierte Entitäten* $\textit{IE}(X)$ wurde für folgende Erkennungsregel eingesetzt:

$$\texttt{'Fehlende Feature-Spezifikation' := IE(X) > 0 .} \tag{5.1}$$

Wenn das konkrete Software-System in Abbildung 5.3 betrachtet wird, ist die Komponente Protokollierung isoliert, da sie zu keinem Feature verfolgt werden kann. Durch die Anwendung der Erkennungsregel 5.1 aus obigen Beispiel wird das Qualitätsdefizit *Fehlende Feature-Spezifikation* erkannt, da die Anwendung der Metrik auf die Komponenten $\textit{IE}(A)$ den Wert 1 ergibt und dieser größer als 0 ist. Die Komponente, die das Qualitätsdefizit verursacht, ist Protokollierung.

Zusammenfassend lässt sich hervorheben, dass das Aufstellen von Erkennungsregeln einen hohen Nutzen hat, da eine Menge von Regeln sehr flexibel definiert werden kann. Zusätzlich sind Regeln leicht in Werkzeuge zu integrieren, da die notwendigen Instrumente, wie Operatoren, bereits in Programmiersprachen enthalten sind. Innerhalb des Werkzeugs lassen sich weitere nützliche Anwendungsfälle umsetzen, wie die automatische Benachrichtigung im Falle von erkannten Qualitätsdefiziten (siehe Kapitel 6).

5.2.4 Ergebnisse der aufgestellten Erkennungsregeln am Beispiel der Weiterentwicklungsfähigkeit

Gemäß dem Aufbauprozess in Kapitel 3 wurden Metriken und Erkennungsregeln auf Grundlage der Qualitätsdefizite in Bezug auf das Qualitätsziel *Weiterentwicklungsfähigkeit* aufgestellt. Für das Bewerten des Qualitätsziels *Weiterentwicklungsfähigkeit* konnten 11 bekannte Metriken aus der Literatur wiederverwendet werden. Im Rahmen der Arbeit wurden zusätzlich 5 neue Metriken erstellt und in der Fallstudie

validiert. Die neu erstellten Metriken und deren Struktur sind in Abschnitt C im Anhang dargestellt.

Die Qualitätsdefizite *Tiefe Vererbungshierarchie*, *Zyklen zwischen Klassen*, *Zyklen zwischen Komponenten* und *Starke Feature-Verschränkung* werden als repräsentative Beispiele für die Ergebnisse dieser Arbeit verwendet und zusammen mit den den Qualitätsdefiziten zugeordneten Metriken und Erkennungsregeln vorgestellt.

5.2.4.1 Erkennen des Qualitätsdefizits Tiefe Vererbungshierarchie

Das Qualitätsdefizit *Tiefe Vererbungshierarchie* stellt eine Abweichung zum Qualitätsteilmerkmal *Lose Kopplung* dar und kann durch die Metrik *Depth of Inheritance Tree* (*DIT*) nach Chidamber und Kemerer [CK94] bewertet werden. Die Zuordnungen zwischen Qualitätsteilmerkmal, Metrik und Qualitätsdefizit, die in den Zuordnungsgraphen aufgenommen wurden, sind in Abbildung 5.4 zu sehen.

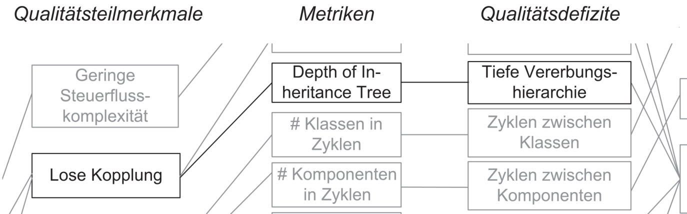


Abbildung 5.4: Zuordnungen des Qualitätsdefizits Tiefe Vererbungshierarchie zu Metrik und Qualitätsteilmerkmal.

Allerdings müssen die Ergebnisse der Metrik erst interpretiert werden, um das Qualitätsdefizit zu erkennen. Dazu wurde die Erkennungsregel 5.2 aufgestellt, die dokumentiert, unter welcher Bedingung das Qualitätsdefizit *Tiefe Vererbungshierarchie* vorliegt.

$$\texttt{'Tiefe Vererbungshierarchie' := DIT > 7 .} \tag{5.2}$$

Die Begründung für die Regel ist wie folgt: die Metrik ***DIT*** bewertet die Vererbung zwischen Klassen. Bei der Vererbung, die eine mögliche Ausprägung einer Kopplung

ist, erhält eine in der Vererbungshierarchie tiefer liegende Klasse alle Merkmale von der in der Vererbungshierarchie höher liegenden Klasse. Um das Qualitätsmerkmal *Lose Kopplung* zu erreichen, sollte die Vererbungshierarchie aus Sicht der *Weiterentwicklungsfähigkeit* so sein, dass eine gute *Wiederverwendbarkeit* möglich ist, ohne aber die Verständlichkeit der Implementierung zu stark zu beeinträchtigen. Um diese Eigenschaften zu erreichen, sollte nach Chidamber und Kemerer die Tiefe der Vererbungshierarchie nicht höher als sieben sein [CK94]. Ist der Wert höher, liegt das Qualitätsdefizit *Tiefe Vererbungshierarchie* vor.

5.2.4.2 Erkennen des Qualitätsdefizits Zyklen zwischen Klassen oder Zyklen zwischen Komponenten

Das Qualitätsteilmerkmal *Zyklenfreiheit* einer Architektur wird nach Lilienthal [Lil08] durch zwei Metriken bewertet: sowohl die Metrik *Anzahl der Klassen in Zyklen* (***KLIZ***) als auch die Metrik *Anzahl der Komponenten in Zyklen* (***KOIZ***) können Abweichungen zu dem Qualitätsteilmerkmal *Zyklenfreiheit* feststellen, woraufhin das Qualitätsdefizit *Zyklen zwischen Klassen* bzw. *Zyklen zwischen Komponenten* erkannt werden kann. Die Zuordnungen zwischen Qualitätsteilmerkmal, Metriken und Qualitätsdefiziten, die in den Zuordnungsgraphen aufgenommen wurden, sind in Abbildung 5.5 dargestellt.

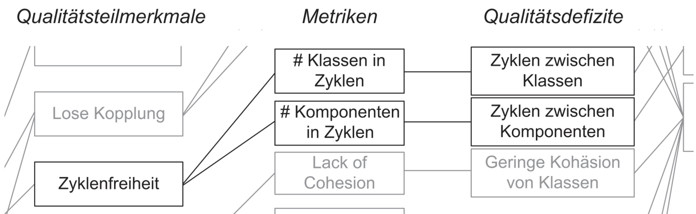


Abbildung 5.5: Zuordnungen der Qualitätsdefizite Zyklen zwischen Klassen und Komponenten zu Metriken und Qualitätsteilmerkmal.

Die Metrik *Anzahl der Klassen in Zyklen* betrachtet Abhängigkeitsbeziehungen zwischen Klassen innerhalb einer Komponente. Bilden diese Abhängigkeitsbeziehungen Zyklen, wird durch die Metrik gezählt, wie viele Klassen in Zyklen sind. Mit Hilfe dieser Metrik kann das Qualitätsdefizit *Zyklen zwischen Klassen* erkannt werden.

Zusätzlich betrachtet die Metrik *Anzahl der Komponenten in Zyklen* Abhängigkeitsbeziehungen zwischen Klassen, die aber in unterschiedlichen Komponenten enthalten sind. Es werden daher durch diese Metrik nur Zyklen betrachtet, welche durch Abhängigkeitsbeziehungen hervorgerufen werden, die aus unterschiedlichen Komponenten stammen. Mit Hilfe dieser Metrik kann das Qualitätsdefizit *Zyklen zwischen Komponenten* erkannt werden.

Jeder Zyklus behindert Weiterentwicklungen, da er die strukturelle Komplexität erhöht [Lil08]. Die Qualitätsdefizite *Zyklen zwischen Klassen* und *Zyklen zwischen Komponenten* existieren daher, sobald ein Zyklus erkannt wurde.

Daher werden die Erkennungsregeln für die beiden Qualitätsdefizite *Zyklen zwischen Klassen* bzw. *Zyklen zwischen Komponenten* folgendermaßen definiert:

```
'Zyklen zwischen Klassen' := KLIZ > 0 .
'Zyklen zwischen Komponenten' := KOIZ > 0.
```
(5.3)

5.2.4.3 Erkennen des Qualitätsdefizits Starke Feature-Verschränkung

Das Qualitätsdefizit *Starke Feature-Verschränkung in Komponenten* ist eine Abweichung zum Qualitätsteilmerkmal *Geringe Feature-Verschränkung*. Das Qualitätsteilmerkmal *Geringe Feature-Verschränkung* kann durch die Metrik *Feature Tangling* (*FTANG*) bewertet werden, somit kann das Qualitätsdefizit *Starke Feature-Verschränkung in Komponenten* erkannt werden. In Abbildung 5.6 ist ein Teil des erstellten Zuordnungsgraphen zu sehen. Dieser Teil zeigt die Zuordnungen zwischen Qualitätsteilmerkmal, Metrik und Qualitätsdefizit.

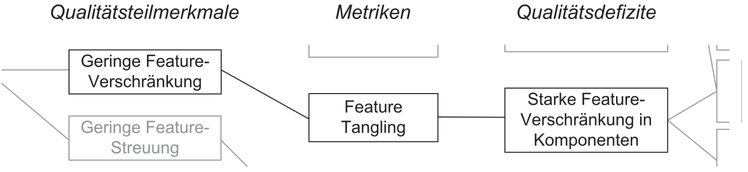


Abbildung 5.6: Zuordnungen des Qualitätsdefizits Starke Feature-Verschränkung zu Metrik und Qualitätsteilmerkmal.

Die Metrik *Feature Tangling* zählt die Abhängigkeitsbeziehungen zwischen Features

und Entitäten der Architektur oder der Implementierung, wie etwa Komponenten (in Abschnitt C.2.3 ist die Metrik im Detail dargestellt). Dabei sollte die Anzahl von Abhängigkeitsbeziehungen zwischen Features und Entitäten der Architektur oder Implementierung, also die Feature-Verschränkung, möglichst gering sein.

Die Metrik $\boldsymbol{FTANG}(A)$ nimmt Werte zwischen 0 (idealer Wert) und 1 (schlechtester Wert) an. Ziel sollte es daher sein, dass die Metrik bei einem Software-System das Ergebnis 0 liefert. Dies wird in der folgenden Regel festgehalten:

$$\texttt{'Starke Feature-Verschränkung in Komponenten' := FTANG(A) > 0 .} \quad (5.4)$$

In der Praxis wird dieser Wert aber selten erreicht. Beispielweise wurde in der Fallstudie (siehe Kapitel 7) bei einem Wert von $0,15$ keine weitere Reengineering-Aktivität mehr angewandt und der Wert durch das Entwicklungsteam als ausreichend klein akzeptiert, wobei die Feature-Verschränkung bezüglich der Komponenten des Software-Systems betrachtet wurde. Während dem Anwendungsprozess sollte der Wert in jeder Iteration durch das Projektteam überprüft werden. Im Anwendungsprozess wird eine Überprüfung durch den Schritt „Prüfung der Erfüllung der Qualitätsziele“ berücksichtigt (siehe Diskussion in Abschnitt 7.3.3 der Fallstudie).

5.3 Bestimmen der Reengineering-Regeln

5.3.1 Ziel und Vorgehen der Aktivität

Eine Reengineering-Regel definiert die Auswahl von Reengineering-Aktivitäten zum Beheben von erkannten Qualitätsdefiziten. In einer Reengineering-Aktivität wird festgelegt, welche Entitäten auf welche Art und Weise geändert werden sollen. Um ein Qualitätsdefizit zu beheben, können mehrere Reengineering-Aktivitäten notwendig sein. Abbildung 5.7 zeigt einen entsprechenden Ausschnitt des Zuordnungsgraphen. Die Beziehungen zwischen Qualitätsdefizit und Reengineering-Aktivitäten bedeuten, dass das Qualitätsdefizit *Fehlende Feature-Spezifikation* durch die Reengineering-Aktivitäten *Rekonstruieren der Features durch Feature-Analyse* [Gre07] oder *Löschen der unnötigen Entitäten durch Refactoring* [Fow99] behoben werden. Ob beide

Reengineering-Aktivitäten angewandt werden müssen oder eine der beiden Reengineering-Aktivitäten zur Behebung des Qualitätsdefizit *Fehlende Feature-Spezifikation* ausreicht, wird durch Reengineering-Regeln festgehalten.

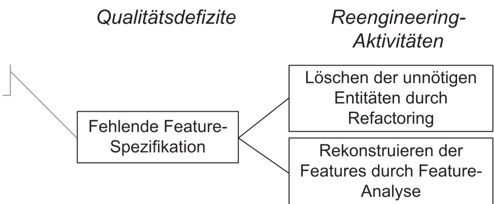


Abbildung 5.7: Beispiel von Zuordnungen zwischen Qualitätsdefizit und Reengineering-Aktivitäten.

Da nach Chikofsky [CC90] Reengineering-Aktivitäten sowohl Entitäten der Spezifikation, der Architektur als auch der Implementierung eines Software-Systems ändern, existieren viele verschiedene Ansätze, sodass der Aufbau von Fachwissen für das Reengineering von Software-Systemen aufwendig ist. Um den Aufwand für dieses Vorgehen zu reduzieren, sollte eine einmal erworbene Erfahrung durch den Experten dokumentiert werden und so für den Anwender zugänglich gemacht werden. Dafür werden in dieser Arbeit Reengineering-Regeln verwendet, die nach einer vorgegebenen Struktur beschrieben werden.

5.3.2 Struktur von Reengineering-Regeln

Eine Reengineering-Regel definiert eine Bedingung und eine Liste von Reengineering-Aktivitäten. Mit Hilfe der Backus-Naur-Form (BNF) aus [BBG+97] wird im Folgenden die Struktur einer Reengineering-Regel dargestellt, siehe Tabelle 5.3.

Tabelle 5.3: Beschreibung der Struktur von Reengineering-Regeln.

```
<Reengineering-Regel>::= <Bedingung> <Liste von Aktivitäten> '.'

<Bedingung>::= 'WENN' <Qualitätsdefizit>

<Liste von Aktivitäten>::= <Reengineering-Aktivität> |
                           <Reengineering-Aktivität>
                           <Logischer Operator>
                           <Liste von Aktivitäten> |

<Qualitätsdefizit>::= 'Name eines Qualitätsdefizits'

<Reengineering-Aktivität>::= 'NUTZE' 'Name einer
                                  Reengineering-Aktivität'

<Logischer Operator>::= 'UND' | 'ODER'
```

In der Bedingung wird nach dem Schlüsselwort `WENN` ein Qualitätsdefizit benannt. Die Bedingung gilt als erfüllt, wenn das Qualitätsdefizit erkannt wurde. Dann kann die Liste der Reengineering-Aktivitäten ausgeführt werden.

In der Liste der Reengineering-Aktivitäten werden einzelne Reengineering-Aktivitäten mit dem Schlüsselwort `NUTZE` eingeleitet, dem der Name einer Reengineering-Aktivität folgt. Die Logischen Operatoren `UND` und `ODER` werden verwendet, um mehrere Reengineering-Aktivitäten zu einer Liste zu verknüpfen. Die Operatoren sind notwendig, wenn mehrere Reengineering-Aktivitäten zusammen angewandt werden müssen (`UND`), oder aber auch eine Reengineering-Aktivität aus mehreren möglichen Reengineering-Aktivitäten verwendet werden kann (`ODER`). Die Entscheidung im `ODER`-Fall, welche der möglichen Reengineering-Aktivitäten angewandt werden, ist Aufgabe des Entwicklers.

5.3.3 Bestimmen von Reengineering-Aktivitäten

Das Bestimmen der Reengineering-Aktivitäten beruht entweder auf der Erfahrung des Experten oder wird durch konkrete Beispiele aus der Literatur unterstützt. Der Experte wählt daher aus einer Menge von Reengineering-Aktivitäten diejenigen aus, welche für das Beheben eines Qualitätsdefizits angewandt werden können.

Beispielsweise kann das Qualitätsdefizit *Starke Feature-Verschränkung in Komponenten* durch die Reengineering-Aktivität *Restrukturieren der Komponenten durch Dekomposition* oder auch durch die Reengineering-Aktivität *Restrukturieren der Features durch Zusammenführen* behoben werden. Diese Zusammenhänge werden in den Zuordnungsgraphen aufgenommen; der dazu gehörende Ausschnitt des erstellten Zuordnungsgraphen ist in Abbildung 5.8 zu finden.

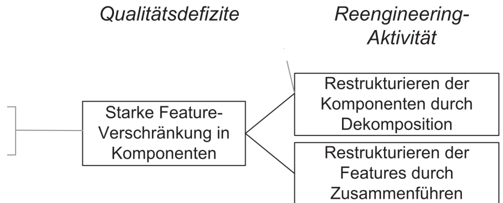


Abbildung 5.8: Beispiel von anwendbaren Reengineering-Aktivitäten.

Die Reengineering-Regel 5.5 stellt durch den Einsatz von Operatoren klar, ob eine der beiden Reengineering-Aktivitäten oder auch beide zusammen angewandt werden sollten. In der Reengineering-Regel 5.5 wurde der Operator `ODER` gewählt, da eine der dort angegeben Reengineering-Regeln ausreichen kann, um das Qualitätsdefizit *Starke Feature-Verschränkung in Komponenten* zu beheben. Somit kann der Qualitätsingenieur während der Anwendung entscheiden, welche Vorgehensweise für den konkreten Anwendungsfall besser geeignet ist. Kommt beispielsweise die Reengineering-Aktivität *Restrukturieren der Features durch Zusammenführen* [PT93] nicht in Frage, da die Features sinnvoll aufgeteilt wurden, bleibt als Alternative nur die Reengineering-

Aktivität *Restrukturieren der Komponenten durch Dekomposition* [Par72].

```
WENN 'Starke Feature-Verschränkung in Komponenten'
NUTZE 'Restrukturieren der Komponenten durch Dekomposition [Par72]'
ODER
NUTZE 'Restrukturieren der Features durch Zusammenführen [PT93]'.
```
(5.5)

5.3.3.1 Schablone für das Bestimmen von Reengineering-Aktivitäten

Um den Nutzen von Reengineering-Aktivitäten während der Anwendung zu erhöhen, wurde für die Definition der Reengineering-Aktivitäten ein schablonenbasierter Ansatz gewählt. Hierbei orientiert sich der Autor an der Arbeit von Rupp [Rup04], in der eine Schablone für die Definition von Anforderungen genutzt wird. Der schablonenbasierte Ansatz soll das Bestimmen von vollständigen und verständlichen Reengineering-Aktivitäten unterstützen, sodass die Reengineering-Aktivitäten in die Reengineering-Regeln integriert werden können.

Durch den schablonenbasierten Ansatz wird der Aufbau einer Reengineering-Aktivität vorgegeben. Die dadurch bestimmten Reengineering-Aktivitäten haben daher eine ähnliche Struktur und weisen alle von der Schablone geforderten Informationen auf. Die in dieser Arbeit verwendete Schablone ist in Abbildung 5.9 dargestellt. In Tabelle 5.4 sind vier Beispiele zu sehen, welche nach dieser Schablone erzeugt wurden.

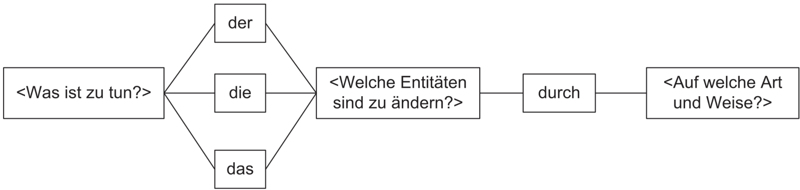


Abbildung 5.9: Schablone für das Beschreiben von Reengineering-Aktivitäten.

Für das Beschreiben einer Reengineering-Aktivität gemäß der Schablone in Abbildung 5.9 muss der Experte angeben, was zu tun ist, welche Entitäten auf welche Art und

Weise geändert werden sollen, um ein Qualitätsdefizit zu beheben. Ausgehend von einem Qualitätsdefizit entscheidet der Experte zunächst, was für eine Art von Aktivität durchzuführen ist, wie etwa das Restrukturieren. Daraufhin kann eingrenzt werden, welche Entitäten des Software-Systems geändert werden sollten. Schließlich muss der Experte noch entscheiden, auf welche Art und Weise das Software-System verbessert werden soll, beispielsweise durch die Angabe einer Technik. Eine Reengineering-Aktivität wird durch Artikel wie „der, die, das“ und das Füllwort „durch“ zu einem grammatikalisch richtigen Satz vervollständigt.

Tabelle 5.4: Beispiele von schablonenbasiert erstellten Reengineering-Aktivitäten.

Was ist zu tun?		Welche Entitäten sind zu ändern?		Auf welche Art und Weise?
Restrukturieren	der	Komponenten	durch	Demotion [Lak96]
Dokumentieren	der	Entitäten	mit	UML [RJB04]
Rekonstruieren	des	Features	durch	Feature-Analyse [Gre07]
Löschen	der	unnötigen Entitäten	durch	Refactoring [Fow99]

Insgesamt wurden beispielhaft 12 Reengineering-Aktivitäten im Rahmen dieser Arbeit bestimmt. Diese sind im Anhang in Abschnitt A.3 zu finden. Im Folgenden werden die durch die Schablone in Abbildung 5.9 geforderten Anteile einer Reengineering-Aktivität weiter erläutert.

Was ist zu tun?

Dieser Teil einer Reengineering-Aktivität drückt die grundlegende Art der Aktivität aus, die durchgeführt werden sollte, um das Qualitätsdefizit zu beheben. Dabei werden die bereits durch Chikofsky und Cross [CC90] definierten Begriffe verwendet. Die Autoren Chikofsky and Cross unterteilen das Reengineering in mehrere grundlegende und wiederkehrende Aktivitäten durch Betrachtung von Spezifikation, Architektur und Implementierung:

1. *Restrukturieren*: Es werden Entitäten aus der Spezifikation, der Architektur und der Implementierung restrukturiert.

   Der Begriff *Dokumentieren* (engl. Redocumentation) wird auch als Restrukturierung angesehen [CC90]. Dabei können neue Dokumente erzeugt werden, um eine existierende Implementierung zu beschreiben. Es kann aber auch sinnvoll sein, bereits existierende Dokumente so anzupassen, damit diese die jetzige Implementierung genauer beschreiben. Außerdem können existierende Dokumente so veraltet sein, dass sie vollständig ersetzt werden müssen.

   Zudem ist das *Löschen* von Entitäten der Spezifikation, Architektur oder Implementierung eine weitere Art des Restrukturierens.

2. *Forward Engineering*: Beim Forward Engineering werden zwei Phasen unterschieden:

   - *Umsetzen* einer Spezifikation in eine Architektur und
   - *Implementieren* einer Architektur in Code.

3. *Reverse Engineering*: Das Reverse Engineering wird auch in zwei Phasen unterteilt:

   - *Rekonstruieren*[1] der Architektur anhand der Implementierung und
   - *Rekonstruieren* der Spezifikation anhand der dokumentierten Architektur.

In den in Tabelle 5.4 dargestellten Reengineering-Aktivitäten wurden beispielsweise die Begriffe „Restrukturieren“, „Dokumentieren“ und „Rekonstruieren“ verwendet.

Welche Entitäten sind zu ändern?

Es gibt verschiedene Entitätstypen, welche mit einem Qualitätsdefizit in Verbindung stehen und während des Anwendungsprozesses durch den Qualitätsingenieur angepasst werden. In dem Aufbauprozess sind die konkreten Namen der Entitäten, wie etwa Klassennamen, nicht bekannt, da diese Informationen erst bei der Anwendung auf ein Software-Produkt bekannt sein können. Im folgenden Abschnitt 5.3.3.2 sind Beispiele zu sehen, die diese Konkretisierung zeigen.

Auf Grundlage eines Qualitätsdefizits kann der Experte den Entitätstyp soweit ein-

[1] Der Begriff Wiedergewinnen kann hier synonym verwendet werden.

schränken, wie es die gegebenen Informationen zum zugeordneten Qualitätsdefizit zulässt. In Tabelle 5.4 sind verschiedene Beispiele zu sehen. Die Reengineering-Aktivität in der ersten Zeile, *Restrukturieren der Komponenten durch Demotion* [Lak96], bezieht sich nur auf Komponenten. Im Gegensatz dazu ist die Reengineering-Aktivität in Zeile 2 allgemeiner formuliert, da verschiedene Entitäten, wie etwa Features oder Komponenten, mit UML [RJB04] (nach-) dokumentiert werden sollen.

Auf welche Art und Weise?

Es gibt bereits viele Methoden, Techniken oder Prozesse, die beschreiben, auf welche Art und Weise ein Software-System verbessert werden kann. Das Auswählen eines Lösungsansatzes für das Beheben von Qualitätsdefiziten wird durch unterschiedliche Eigenschaften beeinflusst, wie etwa durch das verwendete Programmiermodell. Dies muss bei der Bestimmung von Reengineering-Aktivitäten berücksichtigt werden. Beispielsweise wurde die Technik Refactoring nach Fowler [Fow99] hauptsächlich unter Berücksichtigung eines Objekt-Orientierten Programmiermodells erstellt. Die Beispiele in Tabelle 5.4 zeigen auch, dass unterschiedliche Lösungsansätze auf die selben Entitätstypen angewandt werden können.

Zusätzlich zu diesen Angaben kann eine Reengineering-Aktivität weiter detailliert werden. Beispielsweise können die Elementaroperationen von Potts und Takahashi [PT93] bzw. Baldwin [BC03] oder Operationen aus dem Refactoring nach Fowler [Fow99] verwendet werden, um die notwendigen Änderungen unabhängig von einer konkreten Implementierung eines Software-Produkts zu beschreiben.

5.3.3.2 Konkretisierung von Reengineering-Aktivitäten

Die Konkretisierung von Reengineering-Aktivitäten gehört in den Anwendungsprozess, sie wird aber an dieser Stelle der Arbeit aufgegriffen, um einen weiteren Vorteil der schablonenbasierten Vorgehensweise darzustellen. Als Nebenprodukt liefert eine Metrik die durch ein Qualitätsdefizit betroffenen Entitäten. Dadurch können die während dem Aufbauprozess erstellten Reengineering-Aktivitäten im Anwendungsprozess durch die Namen von Entitäten konkretisiert werden. Tabelle 5.5 zeigt Beispiele von Reengineering-Aktivitäten, in denen die von Metriken berechneten konkreten Entitäten eingesetzt wurden. Dabei beziehen sich diese Beispiele auf das bereits bekannte Software-System, das in Abbildung 5.3 dargestellt ist.

Tabelle 5.5: Beispiele konkretisierter Reengineering-Aktivitäten.

Was ist zu tun?		Welche Entitäten sind zu ändern?		Auf welche Art und Weise?
Restrukturieren	der	Komponente Protokollierung	durch	Demotion
Dokumentieren	der	Komponente Protokollierung	mit	UML
Rekonstruieren	des	Features Datensichtbarkeit	durch	Feature-Analyse
Löschen	der	Komponente Protokollierung	durch	Refactoring

Anhand von Traceability-Links können während des Anwendungsprozesses statt des Platzhalters „unnötige Entitäten" die konkreten Namen der Entitäten des Software-Systems ermittelt werden, wie im letzten Beispiel in Tabelle 5.5 die Komponente mit dem Namen Protokollierung.

Das Konkretisieren der Reengineering-Aktivitäten hilft dem Qualitätsingenieur während des Anwendungsprozesses, da er einen konkreten Ansatzpunkt hat, um mit der Durchführung der Reengineering-Aktivität zu beginnen. Es ist auch möglich, die Namen von Entitäten automatisch durch ein Werkzeug zu ermitteln, um konkretisierte Reengineering-Aktivitäten vorzuschlagen, wie in Kapitel 6 gezeigt wird.

5.3.4 Beispiel einer Reengineering-Regel

Die Grundlage für das Beispiel bildet weiterhin das Software-System, welches in Abbildung 5.3 dargestellt ist. Für das folgende Beispiel wird nun angenommen, dass alle benötigten Features spezifiziert wurden und die dafür notwendigen Traceability-Links zwischen Features (in Abbildung 5.3 dargestellt durch Feature-Modelle) und Komponenten (in Abbildung 5.3 dargestellt durch das UML-Komponentenmodell) gesetzt wurden.

Durch Anwenden der Erkennungsregel 5.1 aus Abschnitt 5.2.3.2 wird entdeckt, dass die Komponente Protokollierung nicht durch ein Feature spezifiziert wurde und so das Qualitätsdefizit *Fehlende Feature-Spezifikation* vorliegt. Um dieses Qualitätsdefizit

zu beheben, sollen die in der Komponente Protokollierung umgesetzten Features rekonstruiert und innerhalb von Feature-Spezifikationen dokumentiert werden. Die Rekonstruktion der noch unbekannten Features kann mit der Methode Feature-Analyse nach [Gre07] durchgeführt werden.

Das folgende Beispiel einer Reengineering-Regel legt die Reengineering-Aktivität *Rekonstruieren der Features durch Feature-Analyse [Gre07]* für diesen Fall fest:

```
WENN 'Fehlende Feature-Spezifikation'
NUTZE 'Rekonstruieren der Features durch Feature-Analyse [Gre07]'.
```
(5.6)

Zusammenfassend lässt sich festhalten, dass Reengineering-Regeln das Beheben von Qualitätsdefiziten insofern steuern, als dass der Experte für jedes Qualitätsdefizit eine Menge von Reengineering-Aktivitäten festlegen kann. Dadurch können Qualitätsingenieure mit weniger Erfahrung im Reengineering vom Wissen eines Experten profitieren. Dies verbessert die Entwicklung des Software-Produkts, da bei Qualitätsdefiziten frühzeitig eingegriffen werden kann. Ein weiterer Vorteil der Reengineering-Regel ist deren einfache Umsetzbarkeit in Werkzeugen.

5.3.5 Ergebnisse der bestimmten Reengineering-Regeln am Beispiel der Weiterentwicklungsfähigkeit

Der Aufbauprozess aus Kapitel 3 wurde für das Qualitätsziel *Weiterentwicklungsfähigkeit* angewandt. Bei der Durchführung wurden auch zugehörige Reengineering-Regeln bestimmt.

Als repräsentative Beispiele werden im Folgenden ausgehend von den Qualitätsdefiziten *Hohe Verwendungskopplung*, *Zyklen zwischen Klassen*, *Zyklen zwischen Komponenten* und *Starke Feature-Verschränkung in Komponenten* das Bestimmen der Reengineering-Regeln und der in den Regeln integrierten Reengineering-Aktivitäten näher erläutert.

5.3.5.1 Beheben des Qualitätsdefizits Hohe Verwendungskopplung

Für das Beheben des Qualitätsdefizits *Hohe Verwendungskopplung* kann die Reengineering-Aktivität *Restrukturieren der Klassen durch Refactoring* eingesetzt werden (wie in Abbildung 5.10 zu sehen). Bei dem Qualitätsdefizit *Hohe Verwendungskopplung* muss durch Restrukturieren der Klassen die Anzahl von Abhängigkeitsbeziehungen zwischen zwei Klassen (Kopplung) reduziert werden [Lar05]. Hierbei werden Abhängigkeitsbeziehungen vom Typ Benutzt-Beziehungen betrachtet. Die Benutzt-Beziehung wurde durch Parnas [Par78] eingeführt und durch Rumbaugh, Jacobson und Booch erweitert [RJB04]. Sie sagt aus, dass wenn eine Komponente eine andere benutzt, sie so von ihr abhängig ist.

Das Qualitätsdefizit *Hohe Verwendungskopplung* ist mittlerweile sehr gut erforscht. Daher gibt es bereits Arbeiten, wie die von Demeyer et al. [dBDV04], in der beschrieben ist, wie dieses Qualitätsdefizit zu verstehen ist und durch welche konkreten Maßnahmen es behoben werden kann. Demeyer et al. schlagen das Restrukturieren der Klassen durch konkretes Refactoring wie „Move Method“, „Extract Method“ oder auch „Extract Class“ vor.

Die in den Zuordnungsgraph aufgenommene Zuordnung des Qualitätsdefizits *Hohe Verwendungskopplung* zu dieser Reengineering-Aktivität zeigt Abbildung 5.10.

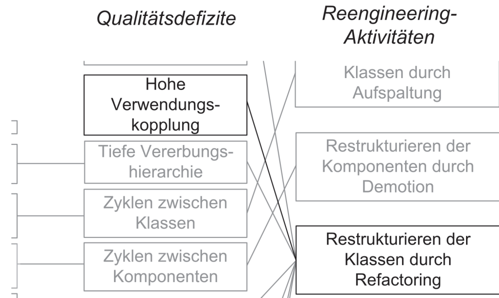


Abbildung 5.10: Anwendbare Reengineering-Aktivität zum Beheben des Qualitätsdefizits Hohe Verwendungskopplung.

Die folgende Reengineering-Regel präzisiert diesen Sachverhalt:

```
WENN 'Hohe Verwendungskopplung'
NUTZE 'Restrukturieren der Klassen durch Refactoring nach [dBDV04]'.
```
(5.7)

5.3.5.2 Beheben der Qualitätsdefizite Zyklen zwischen Klassen und Zyklen zwischen Komponenten

Für das Beheben der Qualitätsdefizite *Zyklen zwischen Klassen* und *Zyklen zwischen Komponenten* gibt es jeweils eine mögliche Reengineering-Aktivität. Die Beziehungen zwischen diesen Reengineering-Aktivitäten und den Qualitätsdefiziten zeigt der Ausschnitt des Zuordnungsgraphen in Abbildung 5.11. Beide Reengineering-Aktivitäten wurden in den Zuordnungsgraphen aufgenommen und werden im Folgenden kurz erläutert. Um das Qualitätsdefizit *Zyklen zwischen Klassen* zu beheben, kann die Restrukturierung der Klassen vorgenommen werden im Sinne einer Aufspaltung der Klassen. Diese Reengineering-Aktivität bezieht sich auf die Arbeit von Baldwin [BC03], in der Klassen zur Restrukturierung in separate Einheiten aufgespalten werden können. Die Aufspaltung einer Klasse kann durch unterschiedliche Maßnahmen erfolgen, wie etwa das Verschieben von Methoden oder Attributen der Klasse (die in [Fow99] als Refactoring bekannt sind).

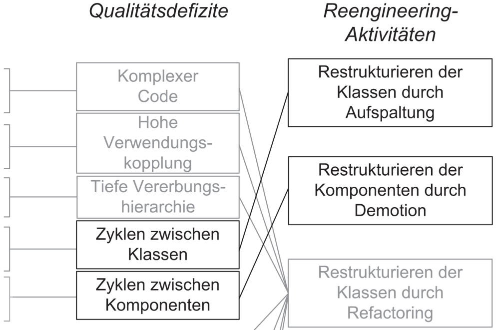


Abbildung 5.11: Anwendbare Reengineering-Aktivitäten zum Beheben der Qualitätsdefizite Zyklen zwischen Klassen und Zyklen zwischen Komponenten.

Abbildung 5.12 zeigt schematisch die Reengineering-Aktivität Restrukturieren der Klassen durch Aufspaltung.

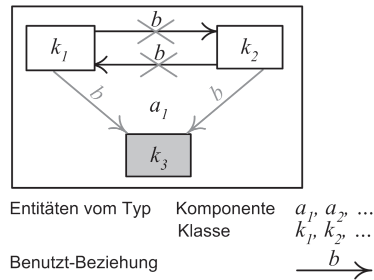


Abbildung 5.12: Schematische Darstellung der Reengineering-Aktivität Restrukturieren der Klassen durch Aufspaltung.

Abbildung 5.12 ist folgendermaßen zu verstehen: Bei zyklischen Abhängigkeitsbeziehungen zwischen zwei Klassen, kann eine neue Klasse k_3 erstellt werden, in welche dann der Teil der Implementierung verschoben wird, der aufgespalten wurde. Dadurch können dann beide Klassen (k_1 und k_2) die neue Klasse (k_3) benutzen, ohne einen Zyklus hervorzurufen. Am Ende der durchgeführten Reengineering-Aktivität sollen die durchgekreuzten Benutzt-Beziehungen in Abbildung 5.12 (dies gilt auch für die noch folgenden Abbildungen) nicht mehr vorhanden sein.

Das Qualitätsdefizit *Zyklen zwischen Komponenten* kann durch das Restrukturieren von Komponenten nach der von Lakos benannten Vorgehensweise „Demotion" behoben werden [Lak96]. Die Reengineering-Aktivität *Restrukturieren der Komponenten durch Demotion* [Lak96] ist in Abbildung 5.13 schematisch dargestellt.

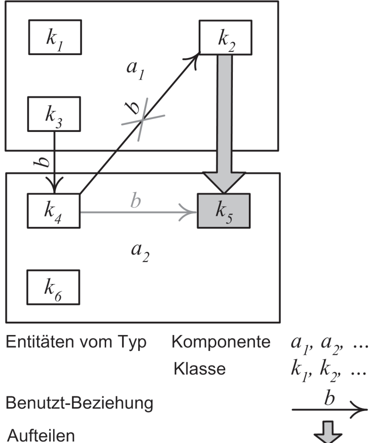


Abbildung 5.13: Schematische Darstellung der Reengineering-Aktivität Restrukturieren der Komponenten durch Demotion.

In der Situation von Abbildung 5.13 liegt eine zyklische Abhängigkeit zwischen den Komponenten a_1 und a_2 vor, da die Klasse k_3 die Klasse k_4 benutzt und diese die Klasse k_2. Bei zyklischen Abhängigkeitsbeziehungen zwischen zwei Komponenten, kann eine der Klassen (k_4 oder k_2), die im Zyklus ist, aufgespalten werden, sodass die Implementierung, die den Zyklus hervorruft, nur von einer Komponente verwendet wird und nicht durch beide Komponenten.

Zum Beheben der Qualitätsdefizite *Zyklen zwischen Klassen* und *Zyklen zwischen Komponenten* können nun die Reengineering-Regeln folgendermaßen formuliert werden:

```
WENN 'Zyklen zwischen Klassen'
NUTZE 'Restrukturieren der Klassen durch Aufspaltung'.
WENN 'Zyklen zwischen Komponenten'
NUTZE 'Restrukturieren der Komponenten durch Demotion [Lak96]'.
```
(5.8)

5.3.5.3 Beheben des Qualitätsdefizits Starke Feature-Verschränkung in Komponenten

Beim Qualitätsdefizit *Starke Feature-Verschränkung in Komponenten* existieren Abhängigkeitsstrukturen, welche das Verstehen der Architektur und des Codes hinsichtlich der spezifizierten Features verschlechtern. Durch die Reengineering-Aktivität *Restrukturieren der Komponenten durch Dekomposition [Par72]* kann das Qualitätsdefizit behoben werden.

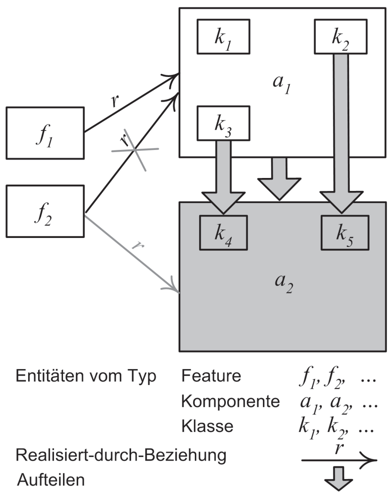


Abbildung 5.14: Schematische Darstellung der Reengineering-Aktivität Restrukturieren der Komponenten durch Dekomposition.

In Abbildung 5.14 wird die Vorgehensweise bei der Reengineering-Aktivität *Restrukturierung der Komponenten durch Dekomposition* mit Hilfe einer Grafik dargestellt. Ziel dieser Reengineering-Aktivität ist es, Abhängigkeitsstrukturen so aufzulösen, dass weniger Features in einer Entität (wie etwa einer Komponente) realisiert sind und die Entität daraufhin idealerweise nur für ein Feature die Verantwortung trägt. Dies geschieht folgendermaßen: Alle Entitäten, welche mehrere Features realisieren (a_1 realisiert f_1 und f_2), werden so aufgeteilt, dass die jeweiligen Entitäten möglichst nur ein Feature umsetzen. Die ursprünglichen Entitäten (a_1) bleiben unverändert erhalten. In der Abbildung wird beispielsweise die Komponente a_1 und deren Klassen

(k_1, k_2, etc.) so aufgeteilt, dass die Realisierung von Feature f_1 in der Komponente a_1 bleibt, aber die Realisierung des Features f_2 in der neuen Komponente a_2 vorgenommen wird.

Das Qualitätsdefizit *Starke Feature-Verschränkung in Komponenten* kann außerdem durch ein Restrukturieren der Features behoben werden. Die Reengineering-Aktivität *Restrukturieren der Features durch Zusammenführen* [PT93] ist in Abbildung 5.15 dargestellt. Das Zusammenführen von Features mit dem Ziel, die Anzahl der in einer Komponente (a_1) involvierten Features (f_1 und f_2) zu reduzieren, führt zu einem neuen Feature. Dessen geänderte Struktur und der angepasste Inhalt muss mit dem Kunden abgestimmt werden.

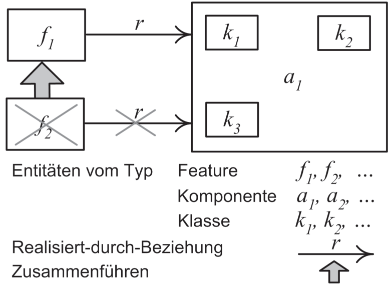


Abbildung 5.15: Schematische Darstellung der Reengineering-Aktivität Restrukturieren der Features durch Zusammenführen.

Diese Überlegungen wurden in der folgenden Reengineering-Regel festgehalten:

```
WENN 'Starke Feature-Verschränkung in Komponenten'
NUTZE 'Restrukturieren der Komponenten durch Dekomposition [Par72]'
ODER
NUTZE 'Restrukturieren der Features durch Zusammenführen [PT93]'.
```

(5.9)

5.4 Zusammenfassen zum Reengineering Rezept

Durch ein Reengineering-Rezept werden die für ein Qualitätsteilmerkmal erstellten Metriken, Regeln zum Erkennen und Regeln zum Beheben von Qualitätsdefiziten durch Reengineering-Aktivitäten zusammengefasst. Jedes Reengineering-Rezept stellt dadurch ein vorgefertigtes Wissen dar, das gezielt zur Erkennung und Behebung von Qualitätsdefiziten genutzt werden kann. Es werden dabei die Qualitätsdefizite erkannt und behoben, die das zum Reengineering-Rezept zugeordnete Qualitätsteilmerkmal beeinträchtigen.

Durch den regelbasierten Aufbau der Reengineering-Rezepte wird außerdem die Erkennung und Behebung von Qualitätsdefiziten objektiviert, sodass die Anwendung von Reengineering-Rezepten mittels Werkzeugen automatisiert werden kann (siehe Kapitel 6).

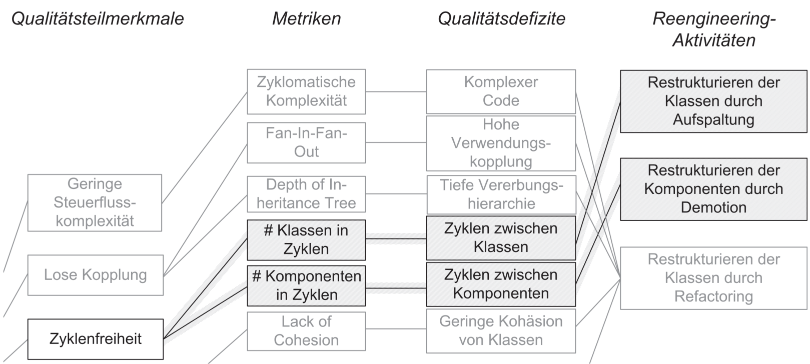


Abbildung 5.16: Zusammenfassen des Reengineering-Rezepts Zyklenfreie Strukturen.

Die Grundlage für das Zusammenfassen eines Reengineering-Rezepts bilden die im Zuordnungsgraphen schrittweise dokumentierten Beziehungen zwischen Qualitätsteilmerkmalen, Metriken, Qualitätsdefiziten und Reengineering-Aktivitäten. Abbildung 5.16 zeigt anhand eines Ausschnitts des erstellten Zuordnungsgraphen die Beziehungen, die für das Zusammenfassen des Reengineering-Rezepts *Zyklenfreie Strukturen* verwendet wurden. Ausgangspunkt für das Zusammenfassen eines Reengineering-Rezepts im Beispiel ist das Qualitätsteilmerkmal *Zyklenfreiheit*. Der Experte folgt

von diesem Ausgangspunkt ausgehend den Beziehungen und entnimmt alle zum Qualitätsteilmerkmal *Zyklenfreiheit* zugeordneten Metriken, Qualitätsdefizite und Reengineering-Aktivitäten, um daraus das Reengineering-Rezept *Zyklenfreie Strukturen* zusammenzufassen.

Für die bessere Kommunikation und Anwendbarkeit des Reengineering-Rezepts wird vom Experten ein eindeutiger Name vergeben und das Ziel des Reengineering-Rezepts kurz dargestellt. Um die Verständlichkeit eines Reengineering-Rezepts zu erhöhen und dessen spätere Nutzung zu erleichtern, wird ein Reengineering-Rezept nach einer einheitlichen und konsistenten Struktur aufgebaut, die im folgenden Abschnitt 5.4.1 vorgestellt wird.

5.4.1 Struktur von Reengineering-Rezepten

Für das Festlegen der Struktur eines Reengineering-Rezepts hat sich die Arbeit an bekannte musterbasierte Ansätze, wie etwa Entwurfsmuster [GHJV95] oder Analysemuster [Fow97], angelehnt. Dem Beispiel des Musters aus [GHJV95] folgend, besteht ein Reengineering-Rezept aus den gleichen Attributen und dient so als wiederverwendbare Vorlage. Die Struktur eines Reengineering-Rezepts besteht aus einem Namen, einem Ziel, einer Zuordnung zu einem Qualitätsteilmerkmal, Metriken und aus Erkennungs- und Reengineering-Regeln:

- **Name:** GOF [GHJV95] fordern für jedes Muster einen eindeutigen Namen zur Identifikation. Dem folgen auch die Reengineering-Rezepte, es muss für jedes erstellte Reengineering-Rezept ein eindeutiger Name durch den Experten vergeben werden. Damit können Reengineering-Rezepte besser kommuniziert und verwendet werden.

- **Ziel:** Für eine bessere Verständlichkeit des Reengineering-Rezepts wird dessen Ziel kurz beschrieben, welches durch das Anwenden des erstellten Reengineering-Rezepts erreicht werden soll. Dabei folgt der Autor der Arbeit von Gamma et al. [GHJV95], in der ebenfalls gefordert wird, dass das Ziel eines Musters beschrieben werden sollte.

- **Qualitätsteilmerkmal**: Jedes Reengineering-Rezept ist einem Qualitätsteilmerkmal zugeordnet. Das entsprechende Qualitätsteilmerkmal wird aus dem für ein Qualitätsziel aufgebauten Zuordnungsgraphen entnommen.

- **Metriken**: Ein Reengineering-Rezept besitzt Metriken, welche das Qualitätsteilmerkmal bewerten und die in den Erkennungsregeln verwendet werden. Auch die Metriken, die einem Qualitätsteilmerkmal zugeordnet sind, werden aus dem für ein Qualitätsziel aufgebauten Zuordnungsgraphen entnommen.
- **Erkennungsregeln:** Für die Erkennung von Qualitätsdefiziten wird mindestens eine Erkennungsregel definiert. Die Beziehungen aus dem Zuordnungsgraphen zwischen Metriken und Qualitätsdefiziten werden durch die Erkennungsregeln um Bedingungen erweitert. Die Struktur und die Bestandteile von Erkennungsregeln wurden in Abschnitt 5.2.3 bereits dargestellt.
- **Reengineering-Regeln:** Für das Beheben der Qualitätsdefizite wird mindestens eine Reengineering-Regel dokumentiert. Die Beziehungen aus dem Zuordnungsgraphen zwischen Qualitätsdefiziten und Reengineering-Aktivitäten werden innerhalb der Reengineering-Regeln ebenfalls mittels Bedingungen erweitert. In Abschnitt 5.3 wurden die Struktur und die Bestandteile von Reengineering-Regeln bereits erläutert.

  Innerhalb eines Reengineering-Rezepts können mehrere Erkennungs- und Reengineering-Regeln bestimmt werden. Die Nennung von mehreren Erkennungs- und Reengineering-Regeln in einem Reengineering-Rezept ermöglicht deren gemeinsame Anwendung im Sinne einer ODER-Verknüpfung.

5.4.2 Beispiel eines Reengineering-Rezepts

Die bereits vorgestellten Beispiele für Erkennungs- und Reengineering-Regeln werden nun in ein Reengineering-Rezept integriert. Das Reengineering-Rezept *Spezifizierte Implementierung* in Tabelle 5.6 ist dem Qualitätsteilmerkmal *Übereinstimmung von Implementierung und Spezifikation* zugeordnet und ermöglicht die Erkennung des Qualitätsdefizits *Fehlende Feature-Spezifikation*, welches eine Abweichung zu dem genannten Qualitätsteilmerkmal ist.

Zur Behebung des Qualitätsdefizits *Fehlende Feature-Spezifikation* dokumentiert die Reengineering-Regel in Tabelle 5.6 anwendbare Reengineering-Aktivitäten. Die dort vorgeschlagenen Reengineering-Aktivitäten unterscheiden sich dabei in dem voraussichtlich notwendigen Aufwand. Dadurch kann der Anwender die Auswirkung eines Qualitätsdefizits hinsichtlich der benötigten Änderungen initial abschätzen. Beispiels-

weise muss zum Beheben des Qualitätsdefizits *Fehlende Feature-Spezifikation* mehr Aufwand für das Beheben eingeplant werden, wenn eine Rekonstruktion eines Features durch Feature-Analyse durchgeführt wird, da diese aufwendiger ist, als das Löschen von Code.

Tabelle 5.6: Beispiel eines Reengineering-Rezepts.

Reengineering-Rezept	Spezifizierte Implementierung
Ziel	Das Qualitätsdefizit *Fehlende Feature-Spezifikation* soll erkannt und behoben werden. Dadurch sollen nur noch Entitäten der Implementierung existieren, deren Bezug zu spezifizierten Features verfolgbar sind.
Qualitätsteil-merkmal	Übereinstimmung von Implementierung und Spezifikation
Metriken	Isolierte Entitäten $IE(X)$
Erkennungs-regeln	`’Fehlende Feature-Spezifikation’ := IE(X) > 0 .`

Reengineering-Regeln	`WENN ’Fehlende Feature-Spezifikation’`
	`NUTZE ’Löschen der unnötigen Entitäten durch Refactoring [Ars08]’`
	`ODER`
	`NUTZE ’Rekonstruieren der Features durch Feature-Analyse [Gre07].’`

5.4.3 Reengineering-Rezepte am Beispiel der Weiterentwicklungsfähigkeit

Ausgehend von dem Qualitätsziel *Weiterentwicklungsfähigkeit* wurde der Aufbauprozess in Kapitel 3 angewandt, um Reengineering-Rezepte für die Erkennung und Behebung von Qualitätsdefiziten zu erstellen. Die während des Aufbauprozesses für das Qualitätsziel *Weiterentwicklungsfähigkeit* erstellten Zuordnungen zwischen identifizierten Qualitätsteilzielen, Qualitätsmerkmalen, Qualitätsteilmerkmalen, Qualitätsdefiziten, Metriken und Reengineering-Aktivitäten wurden durch einen Zuordnungsgraph zwischen Qualitätsziel und Reengineering-Aktivitäten dokumentiert. Dieser wurde bereits in Kapitel 3 gezeigt.

Die in diesem Zusammenhang ermittelten Reengineering-Rezepte sind im Anhang im Abschnitt A.1 aufgeführt. Die Ergebnisse können auf betriebliche Informationssysteme angewandt werden, welche anhand des Objekt-Orientierten Paradigmas entwickelt wurden. Dies wurde durch eine industrielle Fallstudie nachgewiesen, die in Kapitel 7 beschrieben wird.

Tabelle 5.7: Exemplarische Menge von Reengineering-Rezepten zur Verbesserung der Weiterentwicklungsfähigkeit.

Reengineering-Rezept	Qualitätsteilmerkmal	Bezugsebene
Einfache Klassenstruktur	Lose Kopplung	Implementierung
Zyklenfreie Struktur	Zyklenfreiheit	Architektur
Feature-Entschränkung	Geringe Feature-Verschränkung	Spezifikation/Architektur

Tabelle 5.7 zeigt die Auswahl der Reengineering-Rezepte, die im Folgenden im Detail vorgestellt werden. Dabei wurde jeweils ein Reengineering-Rezept mit Bezug zur Implementierung, zur Architektur und zur Spezifikation ausgewählt. Dadurch soll die Notwendigkeit hervorgehoben werden, dass nicht nur der Code eines Software-Produkts für die Bewertung des Qualitätsziels *Weiterentwicklungsfähigkeit* berücksichtigt werden sollte.

5.4.3.1 Reengineering-Rezept Einfache Klassenstruktur

Das Reengineering-Rezept *Einfache Klassenstruktur* hat das Ziel, Qualitätsdefizite im Code zu erkennen, welche durch schwer verstehbare Klassenstrukturen das Ändern der Klassen erschweren. Das Reengineering-Rezept ist in Tabelle 5.8 dargestellt. Das Reduzieren der Anzahl der Abhängigkeitsbeziehungen zwischen Klassen, wie etwa Benutzt-Beziehungen, durch die im Reengineering-Rezept vorgeschlagenen Reengineering-Aktivitäten, führt zu einfacheren Klassenstrukturen.

Tabelle 5.8: Reengineering-Rezept Einfache Klassenstruktur.

Reengineering-Rezept Einfache Klassenstruktur	
Ziel	Die Qualitätsdefizite *Tiefe Vererbungshierarchie* und *Hohe Verwendungskopplung* sollen erkannt und behoben werden. Dadurch werden Klassenstrukturen vereinfacht, sodass die Kopplung zwischen Klassen im Software-System reduziert wird.
Qualitätsteilmerkmal	Lose Kopplung
Metriken	Depth of Inheritance Tree DIT [CK94] Fan-In-Fan-Out FIFO [CG90]
Erkennungsregeln	`'Tiefe Vererbungshierarchie' := DIT > 7 .` `'Hohe Verwendungskopplung' := FIFO > 5 .`

Reengineering-Regeln	`WENN 'Tiefe Vererbungshierarchie'`
	`NUTZE 'Restrukturieren der Klassen durch Refactoring nach [dBDV04]'.`
	`WENN 'Hohe Verwendungskopplung'`
	`NUTZE 'Restrukturieren der Klassen durch Refactoring nach [dBDV04]'.`

5.4.3.2 Reengineering-Rezept Zyklenfreie Strukturen

Das zweite Beispiel eines Reengineering-Rezepts aus Tabelle 5.7 heißt *Zyklenfreie Struktur*. Es ermöglicht das Erkennen und das Beheben von Qualitätsdefiziten, welche die Architektur eines Software-Systems hinsichtlich des Qualitätsteilmerkmals *Zyklenfreiheit* beeinträchtigen. Das Reengineering-Rezept ist in Tabelle 5.9 dargestellt. Die im Reengineering-Rezept vorgeschlagenen Reengineering-Aktivitäten werden verwendet, um Zyklen zwischen Klassen zu entfernen und so eine zyklenfreie Struktur der Klassen im Software-System zu erreichen.

Tabelle 5.9: Reengineering-Rezept Zyklenfreie Struktur.

Reengineering-Rezept Zyklenfreie Struktur	
Ziel	Die Qualitätsdefizite *Zyklen zwischen Klassen* und *Zyklen zwischen Komponenten* sollen erkannt und behoben werden. Das Qualitätsdefizit *Zyklen zwischen Komponenten* wird durch Klassen aus unterschiedlichen Komponenten ausgelöst. Beide Qualitätsdefizite sollen entfernt werden, um zyklenfreie Strukturen zu erhalten.
Qualitätsteilmerkmale	Zyklenfreiheit
Metriken	Klassen in Zyklus ***KLIZ***, Komponenten in Zyklus ***KOIZ***
Erkennungsregeln	`'Zyklen zwischen Klassen' := KLIZ > 0 .` `'Zyklen zwischen Komponenten' := KOIZ > 0 .`
Reengineering-Regeln	`WENN 'Zyklen zwischen Klassen' NUTZE 'Restrukturieren der Klassen durch Aufspaltung [BC03]'.` `WENN 'Zyklen zwischen Komponenten' NUTZE 'Restrukturieren der Komponenten durch Demotion [Lak96]'.`

5.4.3.3 Reengineering-Rezept Feature-Entschränkung

Durch das Reengineering-Rezept *Feature-Entschränkung* wird das Erkennen und Beheben von Qualitätsdefiziten möglich, welche die Spezifikation von Features und deren Realisierung durch die Architektur oder deren Implementierung im Software-System beeinträchtigen. Das Reengineering-Rezept *Feature-Entschränkung* ist in Tabelle 5.10 dargestellt. Als Vorbedingung ist definiert, dass es keine isolierten Komponenten und isolierte Features geben darf. Das Reengineering-Rezept beinhaltet eine Auswahl von Reengineering-Aktivitäten für das Beheben des Qualitätsdefizits *Starke Feature-Verschränkung in Komponenten*.

Tabelle 5.10: Reengineering-Rezept Feature-Entschränkung.

Reengineering-Rezept Feature-Entschränkung	
Ziel	Das Qualitätsdefizit *Starke Feature-Verschränkung in Komponenten* soll erkannt und behoben werden. Abhängigkeitsbeziehungen zwischen mehreren Features zu einer Komponente werden dadurch reduziert, sodass eine Feature-Entschränkung resultiert.
Qualitätsteilmerkmal	Geringe Feature-Verschränkung
Metriken	Isolierte Entitäten $IE(A)$, Isolierte Features IF, Feature Tangling $FTANG(A)$
Erkennungsregeln	`'Starke Feature-Verschränkung in Komponenten' := VORB[(IE(A) = 0) UND (IF = 0)] FTANG(A) > 0 .`
Reengineering-Regeln	`WENN 'Starke Feature-Verschränkung in Komponenten' NUTZE 'Restrukturieren der Komponenten durch Dekomposition [Par72]' ODER NUTZE 'Restrukturieren der Features durch Zusammenführen [PT93]'.`

In den letzten Kapiteln wurden alle Schritte des Aufbauprozesses beschrieben. Mit

den Reengineering-Rezepten ist der Aufbauprozess abgeschlossen. Damit ist das Know-how für eine am Qualitätsziel orientierte Erkennung und Behebung von Qualitätsdefiziten in einem Software-System vorhanden und kann nun mit Hilfe des Anwendungsprozesses in Projekten angewandt werden.

Die Anwendung der Reengineering-Rezepte in Projekten kann wie bereits erwähnt durch den Einsatz von Werkzeugen unterstützt werden. Gerade wenn der Anwendungsprozess mehrfach angewandt werden soll, ist die Umsetzung der Methode durch ein Werkzeug nützlich. Im nächsten Kapitel 6 wird das Werkzeug TraceTool vorgestellt, in dem Automatisierung der Methode vorgestellt wird.

Kapitel 6

Automatisierung der Methode

Durch den regelbasierten Ansatz der in dieser Arbeit entwickelten Methode wird die Automatisierung von einzelnen Schritten des Anwendungsprozesses, wie etwa das Erkennen von Qualitätsdefiziten, erst möglich. Um dies zu demonstrieren und um die Fallstudie (siehe Kapitel 7) durchführen zu können, wurde der regelbasierte Ansatz prototypisch in dem Werkzeug TraceTool umgesetzt. In Abschnitt 6.1 wird eine Übersicht über das Werkzeug gegeben. Die gewählte Architektur des Werkzeugs wird dann in Abschnitt 6.2 erläutert. Im Anschluss werden in den Abschnitten 6.3 und 6.4 die wichtigsten Funktionalitäten des Werkzeugs bezüglich der Automatisierung der Methode und bezüglich des Aufbaus von auswertbaren Abhängigkeitsdaten dargestellt. Schließlich wird am Ende des Kapitels eine kurze Zusammenfassung gegeben.

Dieses Kapitel beschäftigt sich nicht mit der Implementierung des Werkzeugs, sondern zeigt, wie der regelbasierte Ansatz dieser Methode automatisiert werden kann. Die Implementierung des Werkzeugs erfolgte teilweise durch die Masterarbeiten [Vog07] und [Ble08], die vom Autor initiiert und betreut wurden. Mehr Informationen über die Implementierung des Werkzeugs finden sich daher in den genannten Arbeiten.

6.1 Übersicht über das TraceTool

Reengineering-Rezepte geben Erkennungsregeln und Reengineering-Regeln vor. Im Werkzeug TraceTool wurden die Erkennungsregeln zur Bewertung des Qualitätsziels *Weiterentwicklungsfähigkeit* prototypisch umgesetzt. Damit können Qualitätsdefizite auf Grundlage der Erkennungsregeln automatisch erkannt werden. Das Beheben

der erkannten Qualitätsdefizite wird durch Reengineering-Aktivitäten ermöglicht, die durch die Reengineering-Regeln vorgeschlagen wurden. Neue Regeln und Metriken können durch die Erweiterung des existierenden Codes flexibel eingeführt werden, da der grundsätzliche Zugriff auf die Abhängigkeitsdaten durch eine einheitliche Schnittstelle festgelegt ist.

Auf Code-Ebene bieten heutige Programmierwerkzeuge, wie etwa das Produkt *Eclipse*[1], die Unterstützung von automatischen Restrukturierungen an, die als Refactorings durch Fowler [Fow99] eingeführt worden sind. In der Fallstudie wurde vom Entwicklungsteam das Programmierwerkzeug Eclipse benutzt. Daher arbeitet das Werkzeug TraceTool mit Eclipse zusammen.

Das automatische Ausführen von vorgeschlagenen Reengineering-Aktivitäten ist durch die in Eclipse vorhandene Funktionalität möglich, wurde aber aus Gründen des Aufwands nicht umgesetzt. Auf der Architektur-Ebene gibt es noch keine ähnlich gute automatische Unterstützung. Daher müssen entsprechende Reengineering-Aktivitäten manuell durchgeführt werden.

Mit Hilfe des Werkzeugs TraceTool wird auch ein Teil der Dokumentationsaufgaben automatisiert: Tätigkeiten des Entwicklers, welche er mit Hilfe von Eclipse durchführt, können so automatisch durch Traceability-Links verfolgt werden. Beispielsweise können Restrukturierungen an Modellen oder Code verfolgt werden, wenn diese innerhalb von Eclipse durchgeführt werden. Ein Beispiel für ein solches Modell ist ein Feature-Modell, welches im XML-Format in Eclipse vorliegt.

6.2 Architektur des Werkzeugs

Abbildung 6.1 zeigt die Komponenten des Werkzeugs und mit welchen anderen Produkten es wie kommuniziert. Neben einem Block mit den Komponenten des Werkzeugs sind zwei weitere Blöcke zu sehen.

Ein Block umfasst das Werkzeug Eclipse, das vom Entwicklungsteam als Programmierwerkzeug benutzt wird, um Java Code und verschiedene Modelle zu erzeugen. Um bestimmte Aktivitäten, die in Eclipse durchgeführt werden, wie das Verfolgen von Restrukturierungen, wurde ein Eclipse-Plugin mit dem Namen TraceTool-Plugin

[1] www.eclipse.org

entwickelt. Wie in Abbildung 6.1 dargestellt, verwendet das TraceTool-Plugin aus dem Programmierwerkzeug Eclipse Komponenten des Werkzeugs TraceTool.

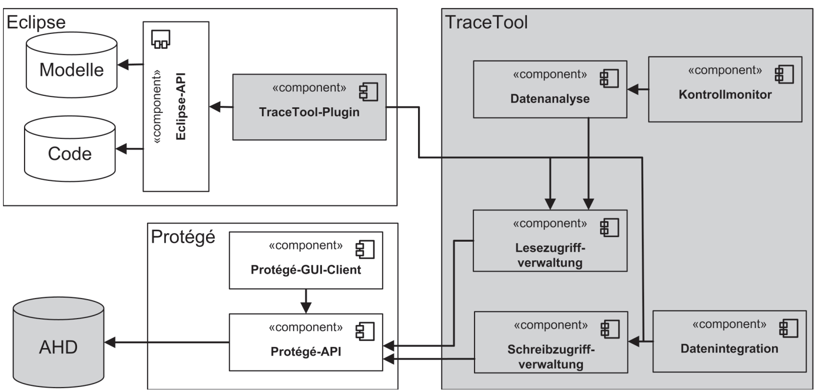


Abbildung 6.1: Architektur des Werkzeugs TraceTool.

Der andere Block enthält das Produkt *Protégé*[2]. Das Werkzeug TraceTool verwendet Protégé, um flexibel Beziehungen zwischen Modellen und Code verwalten zu können. Protégé ermöglicht es, die in dieser Arbeit fokussierten Abhängigkeitsbeziehungen strukturiert innerhalb eines Repositories zu verwalten. Außerdem können durch die vom Produkt angebotene Protégé-API effizient Abhängigkeitsbeziehungen in Form von Abhängigkeitsdaten (AHD) persistent gespeichert, geändert und durch selbst definierbare Integritätsbedingungen konsistent gehalten werden.

Abhängigkeitsdaten sind aus Modellen oder Code extrahierte Abhängigkeitsbeziehungen zwischen Entitäten, die im Repository des Werkzeugs TraceTool verwaltet und durch das Werkzeug ausgewertet werden können. Abhängigkeitsdaten bilden verschiedene Typen von Entitäten und Abhängigkeitsbeziehungen aus Software-Systemen ab, wie etwa eine Benutzt-Beziehung zwischen zwei Klassen oder einen Traceability-Link zwischen einem Feature und einer Komponente.

Es gibt zwei Oberflächen mit denen ein Anwender arbeiten kann, den Protégé-GUI-Client (siehe Abbildung 6.3) und die eigens entwickelte Komponente Kontrollmonitor (siehe Abbildung 6.2). Um die erstellten Abhängigkeitsdaten zu visualisieren und zu bearbeiten, bietet sich der Protégé-GUI-Client an, mit dem es möglich ist, Abhän-

[2] http://protege.stanford.edu

gigkeitsdaten direkt zu verwalten. Der Protégé-GUI-Client verwendet dabei auch die Protégé-API, wie das Architekturdiagramm in Abbildung 6.1 darstellt.

Abbildung 6.1 zeigt alle Komponenten des Werkzeugs TraceTool. Die grau hervorgehobenen Komponenten wurden im Rahmen der Arbeit entwickelt und werden im Folgenden beschrieben. Die anderen Komponenten gehören zu den bereits eingeführten Produkten Protégé und Eclipse.

Folgende Architekturkomponenten wurden zur Validierung der Methode entwickelt und eingesetzt:

- TraceTool-Plugin: Das TraceTool-Plugin ermöglicht die automatische Anpassung von Abhängigkeitsdaten, wenn Entitäten, wie etwa Komponenten oder Klassen, innerhalb des Programmierwerkzeugs erstellt, gelöscht oder geändert werden. Hierfür wird eine ereignisgesteuerte Implementierung im Werkzeug Eclipse verwendet: beim Speichern, Erzeugen, Löschen oder Verschieben von Code oder XML-Dateien werden Ereignisse ausgelöst, die vom TraceTool-Plugin über die Eclipse-API abgefangen und ausgewertet werden können.

- Schreibzugriffverwaltung: Diese Komponente ermöglicht das Speichern von Abhängigkeitsdaten. Die Speicherung der Abhängigkeitsdaten wird durch die Nutzung der Protégé-API ermöglicht. Die Abhängigkeitsdaten werden in einem Repository gespeichert, welches in Abbildung 6.1 mit AHD bezeichnet ist.

- Lesezugriffverwaltung: Durch diese Komponente wird das gezielte Auffinden von erstellten Abhängigkeitsbeziehungen innerhalb der Abhängigkeitsdaten im Repository ermöglicht. Dabei wird die Protégé-API von der Komponente Lesezugriffverwaltung benutzt.

- Datenanalyse: Die Komponente Datenanalyse enthält die Implementierung der Anwendungslogik zur Auswertung der Abhängigkeitsdaten. Die Anwendungslogik entspricht der Realisierung der im Aufbauprozess in Abschnitt 3.1 aufgebauten Reengineering-Rezepte. Die Komponente Datenanalyse benutzt die Komponente Lesezugriffverwaltung zur Auswertung der Abhängigkeitsdaten. Beispielsweise wurden Erkennungsregeln prototypisch umgesetzt, welche in dieser Arbeit am Beispiel des Qualitätsziels *Weiterentwicklungsfähigkeit* erstellt wurden.

- Kontrollmonitor: Zur Anzeige von Qualitätsdefiziten und den Ergebnissen der Metriken, die bei der Anwendung der Erkennungsregeln auf ein Software-System

entstehen, wird die Komponente Kontrollmonitor verwendet. Dabei greift die Komponente auf die Komponente Datenanalyse zu, um Berechnungsergebnisse der aktuell durchgeführten Metriken anzuzeigen.

- Datenintegration: Die Komponente Datenintegration ermöglicht die automatische Erstellung von Abhängigkeitsdaten, welche durch statische Code-Analysen von Java-Programmen erzeugt wurden. Die Komponente Datenintegration stellt dafür einen Parser zur Verfügung, welcher ein Software-System nach Abhängigkeitsbeziehungen durchsucht. Zur Speicherung der Abhängigkeitsdaten im Repository des Werkzeugs TraceTool benutzt die Komponente Datenintegration die Komponenten Lesezugriffverwaltung und Schreibzugriffverwaltung.

6.3 Automatische Erkennung von Qualitätsdefiziten

Zur Erkennung von Qualitätsdefiziten werden Metriken im Werkzeug TraceTool berechnet, sodass die Erkennungsregeln automatisch ausgewertet werden können. Sowohl die Ergebnisse der Berechnung der Metriken als auch die Auswertung der Erkennungsregeln werden dem Anwender mit Hilfe der Komponente Kontrollmonitor dargestellt. In Abbildung 6.2 ist die GUI der Komponente Kontrollmonitor zu sehen. Der Screenshot zeigt beispielsweise das Ergebnis der ausgewerteten Erkennungsregel des Reengineering-Rezepts *Spezifizierte Implementierung*, indem die dort definierten Ergebnisse und erkannten isolierten Entitäten dargestellt werden:

- Konnte beispielsweise anhand der Erkennungsregel ein Qualitätsdefizit erkannt werden, werden die für das Qualitätsdefizit relevanten Entitäten des Software-Systems im Anzeigefeld der Komponente Kontrollmonitor durch das Tool „rot“ eingefärbt, wie die isolierte Klasse *javax.xml.registry.SaveException*. In Abbildung 6.2 ist dieses Anzeigefeld mit (A) markiert. Diese „rot“ markierten Entitäten können außerdem zum Konkretisieren der Reengineering-Aktivitäten genutzt werden. Das Konkretisieren von Reengineering-Aktivitäten wurde in Abschnitt 5.3.3.2 beschrieben.

- Sind Entitäten nicht an einem Qualitätsdefizit beteiligt, werden sie im Werkzeug „grün“ hervorgehoben. Das mit (B) markierte Anzeigefeld in Abbildung 6.2 hebt diesen Zustand am Beispiel des Features Datenmonitor hervor.

- Der Screenshot im Teil (C) zeigt auch, dass die Ergebnisse von verschiedenen berechneten Metriken auch gezeigt werden können.

Der Kontrollmonitor eignet sich auch für die Regressionsprüfung von durchgeführten Reengineering-Aktivitäten. Um etwa das Qualitätsdefizit *Fehlende Feature-Spezifikation* zu beheben, muss beispielsweise die Rolle der isolierten Klasse *javax.xml.registry.-SaveException* überprüft werden. Nur wenn die vom Reengineering-Rezept vorgesehene Reengineering-Aktivität erfolgreich war, also das Qualitätsdefizit *Fehlende Feature-Spezifikation* behoben wurde, wird der Status der Klasse *javax.xml.registry.-SaveException* im Kontrollmonitor von „rot“ auf „grün“ durch das Werkzeug verändert.

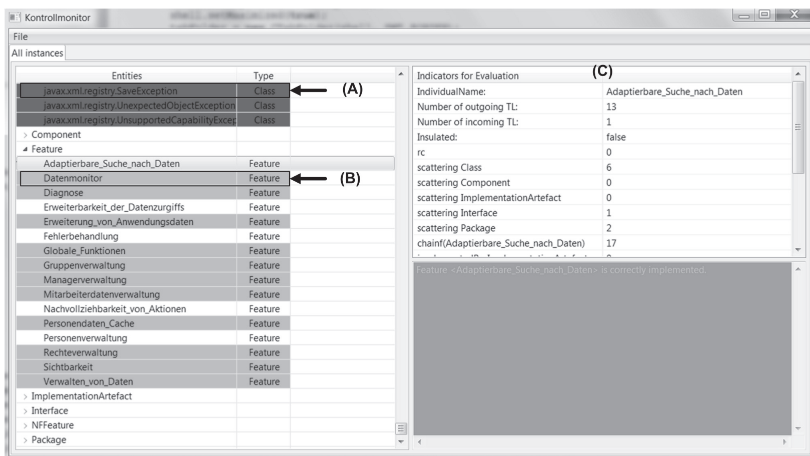


Abbildung 6.2: TraceTool GUI - Kontrollmonitor.

6.4 Aufbauen von Abhängigkeitsdaten

Es gibt mehrere Möglichkeiten, wie auf Grundlage der im Werkzeug TraceTool definierten Struktur Abhängigkeitsdaten erstellt werden können: die Erstellung der Abhängigkeitsdaten kann entweder manuell mit dem Protégé-GUI-Client, halb-automatisch mit der Komponente Datenintegration oder automatisch während der Entwicklungstätigkeit mit Hilfe der Komponente TraceTool-Plugin erfolgen.

In allen drei Fällen wird zuerst die Struktur der Abhängigkeitsdaten durch die Beschreibungssprachen Resource Description Framework (RDF) [LS99] und Web On-

tology Language (OWL) [DS04] festgelegt. Daher bestehen Abhängigkeitsdaten aus *OWLClasses* und *Properties*. Konkrete Instanzen können in Form von *Individuals* im Werkzeug verwaltet werden. Das Produkt Protégé bietet für das Festlegen der Struktur von Abhängigkeitsdaten den Protégé-GUI-Client an. Ein Screenshot dieser Oberfläche ist in Abbildung 6.3 wiedergegeben. Die markierten Inhalte werden im Folgenden erläutert.

1. *OWLClasses*: Durch *OWLClasses* werden Elemente von Abhängigkeitsdaten beschrieben. Dadurch können in dieser Arbeit Entitätstypen wie beispielsweise Feature, Komponente oder Klasse abgebildet werden. Dafür werden Attribute eines Elements, wie etwa dessen Name, definiert. In Abbildung 6.3 sind im Bedienfeld *Class Browser* alle Entitätstypen des in der Fallstudie untersuchten Software-Systems abgebildet (siehe Markierung (1)).

2. *Properties*: Die Struktur von Beziehungen zwischen Elementen der Abhängigkeitsdaten beschreiben *Properties*. Ein Beispiel dafür ist eine Benutzt-Beziehung zwischen zwei Klassen. Diese Beziehung ist in Abbildung 6.3 zu finden, sie trägt dort den Namen „Class use Class“ (siehe Markierung (2)). Weitere Beziehungen sind in diesem Teil des Screenshots ebenfalls zu sehen.

3. *Individuals*: Im Produkt Protégé sind *Individuals* konkrete Instanzen von *OWLClasses* oder *Properties*. *Individuals* repräsentieren daher die mit Hilfe des Werkzeugs TraceTool erstellten Abhängigkeitsdaten. In Abbildung 6.3 im Bedienfeld *Instance Browser* (siehe Markierung (3)) sind die Namen von Java-Klassen des Software-Systems aus der Fallstudie beispielhaft dargestellt.

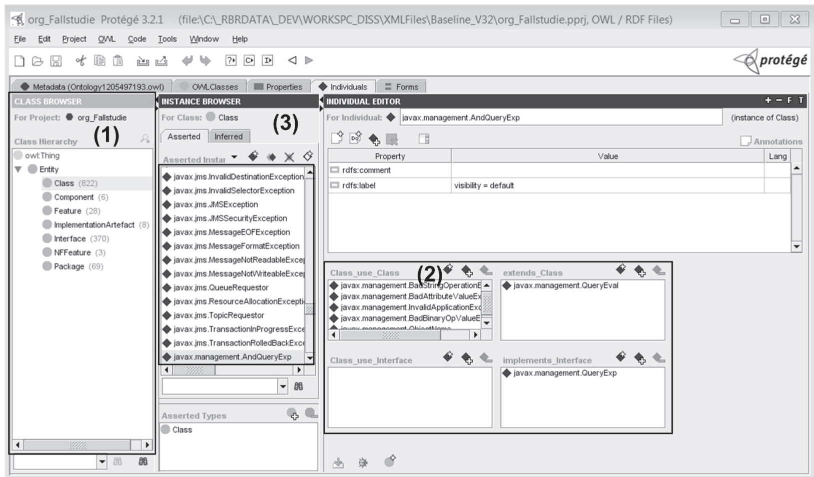


Abbildung 6.3: Definieren der Struktur von Abhängigkeitsdaten und deren Verwaltung mit dem Protégé-GUI-Client.

6.5 Zusammenfassung

Die Automatisierung von einzelnen Schritten des Anwendungsprozesses aus Abschnitt 3.2, wie etwa das Anwenden von Reengineering-Regeln zum Erkennen von Qualitätsdefiziten, wurde prototypisch im Werkzeug TraceTool umgesetzt. Der in den Reengineering-Rezepten gewählte regelbasierte Ansatz begünstigt die automatisierte Anwendung der Methode, da die vorgeschlagene Struktur von Regeln effektiv in Code realisiert werden kann. Beispielsweise werden die für die Regeln definierten Operatoren bereits durch die Programmiersprache Java angeboten.

Das durch eine GUI unterstützte Vorschlagen von Reengineering-Aktivitäten kann leicht ins Werkzeug TraceTool eingebaut werden, dies wurde aber nicht vollständig umgesetzt. Wie in Abschnitt 6.3 bereits vorgestellt, können schon jetzt die für ein Qualitätsdefizit relevanten Entitäten durch das Werkzeug ermittelt werden und so für das Konkretisieren von Reengineering-Aktivitäten verwendet werden. Beispielsweise zeigt der Kontrollmonitor im Fall des Qualitätsdefizits *Zyklen zwischen Klassen* die Liste der Klassen an, die im Zyklus enthalten sind.

Das Werkzeug unterstützt durch die in diesem Kapitel vorgestellten Funktionen den Anwendungsprozesses der Methode. Eine weitere Verbesserung des Werkzeugs TraceTool bezüglich des Anwendungsprozess könnte durch das Festlegen und Verwalten der Prioritäten für die Reengineering-Rezepte erreicht werden, bleibt aber zunächst einer künftigen Produktversion vorbehalten.

Innerhalb der Fallstudie wurde auch die Erweiterbarkeit des Werkzeugs durch weitere Entitäten und Abhängigkeitsbeziehungen erfolgreich getestet. Beispielsweise wurden Pakete, ein Strukturierungsmittel der Programmiersprache Java, als weiterer Entitätstyp im Werkzeug TraceTool eingeführt. Danach konnte auch dieser Entitätstyp und die dazu definierten Abhängigkeitsbeziehungen automatisch durch das TraceTool verwaltet werden.

Kapitel 7

Validierung

Um die Methode dieser Arbeit zu validieren, wurde der Anwendungsprozess in Abbildung 3.3 im Rahmen einer industriellen Fallstudie durchgeführt. Dafür wurden die Reengineering-Rezepte und der Zuordnungsgraph verwendet, die im Aufbauprozess exemplarisch für das Qualitätsziel *Weiterentwicklungsfähigkeit* aufgebaut wurden. Einen Überblick über das in der Fallstudie untersuchte Software-System gibt Abschnitt 7.1. Das Ziel der Fallstudie war die Verbesserung dieses Software-Systems. Es wurden Qualitätsdefizite, welche das Qualitätsziel *Weiterentwicklungsfähigkeit* beeinträchtigen, erkannt und behoben. Dabei wurden die erstellten Reengineering-Rezepte verwendet, welche anhand des Aufbauprozesses in Abbildung 3.1 für das Qualitätsziel *Weiterentwicklungsfähigkeit* erstellt und in den Kapiteln 4 und 5 als Ergebnisse vorgestellt wurden.

Die Anwendungsreihenfolge der Reengineering-Rezepte wurde durch das Entwicklungsteam des betrachteten Software-Systems festgelegt. Dazu wurden – dem Anwendungsprozess folgend – zuerst die Qualitätsteilziele, Qualitätsmerkmale und Qualitätsteilmerkmale priorisiert, wie in Abschnitt 7.2 beschrieben ist. Die priorisierten Reengineering-Rezepte wurden in mehreren Iterationen angewandt. Details dazu werden in Abschnitt 7.3 vorgestellt. Eine Zusammenfassung der Ergebnisse der Fallstudie ist in Abschnitt 7.4 zu finden.

7.1 Überblick über die Fallstudie

In der Fallstudie wird ein Datenverwaltungssystem aus dem industriellen Umfeld untersucht, welches bereits seit mehreren Jahren produktiv eingesetzt wird. Es wird regelmäßig durch die Realisierung neuer Features erweitert. Das Datenverwaltungssystem ist nur ein Teil eines Software-Systems, dessen Entwicklungsteam zur Zeit der Fallstudie aus mehreren Personen bestand. Davon waren zwei Mitarbeiter für die Entwicklung des Datenverwaltungssystems zuständig.

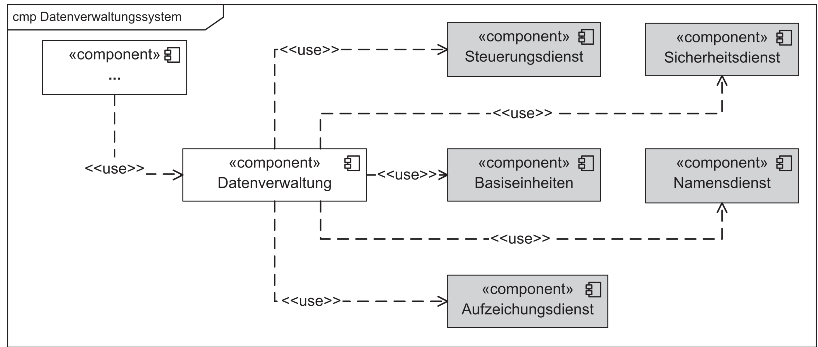


Abbildung 7.1: Komponentenmodell des Datenverwaltungssystems.

Das Gesamtsystem bildet neben anderen Systemen im Unternehmen die Grundlage für betriebliche Anwendungen und stellt daher verschiedene Dienste zur Verfügung. Dazu gehören die Verwaltung von organisatorischen und sicherheitsrelevanten Informationen des Unternehmens, die innerhalb einer Datenbank hochverfügbar und zentral gespeichert sind. In Abbildung 7.1 ist ein vereinfachtes Komponentenmodell des Software-Systems der Fallstudie dargestellt. Das Software-System nutzt weitere Komponenten, welche aber in der Fallstudie nicht weiter betrachtet werden (sie sind in Abbildung 7.1 grau hervorgehoben). Der Steuerungsdienst bietet notwendige Steuerungsdaten innerhalb des Lebenszyklus der Komponente Datenverwaltung an. Der Namensdienst ermöglicht die Ermittlung von weiteren Komponenten. Dabei kann über den Sicherheitsdienst der Zugriff auf Daten eingeschränkt werden. Die Komponenten Basiseinheiten und Aufzeichungsdienst liefern Umgebungsinformationen, welche während der Laufzeit der Komponente Datenverwaltung gebraucht werden.

In der Fallstudie wurde die Komponente Datenverwaltung anhand der hier vorgestellten Methode untersucht. Diese Komponente hat den Umfang 34744 SLOC (Source Lines of Code), 22402 MLOC (Method Lines of Code) und 2266 NOM (Number of Methods). Auf Grund einer Vertraulichkeitsvereinbarung können weitere Details über das Software-System und das Unternehmen nicht genannt werden. Daher wurden teilweise Systemnamen oder Komponentenbezeichnungen mittels der bisher in der Arbeit definierten Symbole und Buchstaben anonymisiert. Dies zeigt sich auch in der Übersicht der Ergebnisse der Fallstudie im Anhang in Tabelle D.2. Dort ist jeder Klassenname durch eine eindeutige Nummer ersetzt worden.

Ein besonders wichtiges Ziel für die Akzeptanz des Produkts ist die regelmäßige Auslieferung von neuen Produktversionen. Dieses Ziel ist an das generelle Ziel einer langfristigen Auslegung des Software-Systems gekoppelt. Da regelmäßig neue Kundenanforderungen gestellt werden, die anhand von Features im Software-System realisiert werden, muss das Software-System regelmäßig erweitert werden können. Um die Akzeptanz des Produkts im Unternehmen zu gewährleisten, muss es stabil und weiterentwickelbar sein. Schließlich ist für die Erhaltung der Wirtschaftlichkeit des Software-Systems die *Weiterentwicklungsfähigkeit* ein wichtiges Qualitätsziel.

Die Qualitätsziele, welche durch das Entwicklungsteam festgelegt wurden, sind in Tabelle 7.1 dargestellt. Auch wenn weitere wichtige Qualitätsziele durch das Entwicklungsteam genannt wurden, wurde das Qualitätsziel *Weiterentwicklungsfähigkeit* am höchsten gewichtet (mit 20 von 100 Punkten).

Tabelle 7.1: Gewichtung der Qualitätsziele durch das Entwicklungsteam.

Qualitätsziele	Gewicht
Ergonomie	**0**
Weiterentwicklungsfähigkeit	**20**
Wartbarkeit	**16**
Performance	**10**
Portabilität	**6**
Verlässlichkeit	**16**
Sicherheit	**16**
Usability	**6**
Funktionalität	**10**
Gesamt	**100**

7.2 Priorisieren der Qualitätsteilziele, -merkmale und -teilmerkmale

Die Priorisierung der Qualitätsteilziele, Qualitätsmerkmale und Qualitätsteilmerkmale wurden durch das Entwicklungsteam mit Hilfe des Zuordnungsgraphen in Abbildung 3.2 durchgeführt, da dort alle Zuordnungen zwischen dem Qualitätsziel *Weiterentwicklungsfähigkeit* und den Reengineering-Aktivitäten dargestellt sind.

Tabelle 7.2: Gewichtung der Qualitätsteilziele und Qualitätsmerkmale des Qualitätsziels Weiterentwicklungsfähigkeit.

Qualitätsteilziele (Gewicht)	Qualitätsmerkmale	Gewicht
Analysierbarkeit (2)		
	Geringe Strukturelle Komplexität	**1**
	Modularität	**1**
Änderbarkeit (3)		
	Geringe Strukturelle Komplexität	**1**
	Modularität	**1**
	Trennung der Zuständigkeiten	**1**
Wiederverwendbarkeit (3)		
	Geringe Strukturelle Komplexität	**0**
	Modularität	**1**
	Trennung der Zuständigkeiten	**1**
	Korrektheit	**1**
Testbarkeit (6)		
	Geringe Strukturelle Komplexität	**0**
	Modularität	**1**
	Trennung der Zuständigkeiten	**3**
	Korrektheit	**2**
Verfolgbarkeit (6)		
	Geringe Strukturelle Komplexität	**1**
	Korrektheit	**5**
Gesamt		**20**

Auf Grundlage dieser Verfeinerung wurde gemäß dem Anwendungsprozess die Gewichtung durchgeführt. Ausgangspunkt waren die 20 Punkte für das Qualitätsziel *Weiterentwicklungsfähigkeit*. Diese wurden auf die Qualitätsteilziele verteilt. Beispielweise erhielt das Qualitätsteilziel *Verfolgbarkeit* 6 Punkte. Die Punktzahl eines Qualitätsteilziels wurde weiter verteilt auf die Qualitätsmerkmale, wie in Tabelle 7.2 zu sehen ist. Dadurch, dass ein Qualitätsmerkmal mehreren Qualitätsteilzielen zugeordnet werden kann, kann es auch von mehreren Qualitätsteilzielen Punkte erhalten, wie am Beispiel des Qualitätsmerkmals *Korrektheit* zu sehen ist.

Am Ende war eine Priorisierung der Reengineering-Rezepte anhand der zugeordneten und bereits gewichteten Qualitätsteilmerkmale möglich. Dafür wurde der Wert der einzelnen Qualitätsteilmerkmale zuerst summiert und anschließend sortiert; dadurch ergibt sich die Priorisierung der Reengineering-Rezepte. Ein Beispielausschnitt aus dem Ergebnis der Priorisierung ist in Tabelle 7.3 dargestellt. Beispielsweise kam das Qualitätsmerkmal *Korrektheit* mit 8 Punkten auf den höchsten Wert. Diese Punkte wurden über die Qualitätsteilmerkmale auf die zugehörenden Reengineering-Rezepte verteilt. Im Falle des Qualitätsmerkmal *Korrektheit* erhielten die Reengineering-Rezepte *Spezifizierte Implementierung* und *Verbindliche Spezifikation* jeweils 4 Punkte. Dies waren die Reengineering-Rezepte mit der höchsten Punktzahl und erhielten damit die höchste Priorität. Andere Qualitätsteilmerkmale wie etwa die *Datenkapselung* wurden sehr niedrig bewertet und in Tabelle 7.3 nicht mehr aufgenommen.

Tabelle 7.3: Priorisierte Qualitätsmerkmale, Qualitätsteilmerkmale und zugehörige Reengineering-Rezepte.

Qualitätsmerkmal/Qualitätsteilmerkmal	Reengineering-Rezepte	Gewicht
Korrektheit (8)		
Übereinstimmung von Implementierung und Spezifikation	Spezifizierte Implementierung	**4**
Vollständigkeit der Implementierung	Verbindliche Spezifikation	**4**
Trennung der Zuständigkeiten (5)		
Geringe Feature-Verschränkung	Feature-Entschränkung	**3**
Geringe Feature-Streuung	Feature-Entstreuung	**2**
Modularität (4)		
Lose Kopplung	Einfache Klassenstruktur	**2**
Hohe Kohäsion	Kohäsive Klassen	**2**
Geringe Strukturelle Komplexität (3)		
Zyklenfreiheit	Zyklenfreie Struktur	**3**

7.3 Auswählen und Anwenden der Reengineering-Rezepte

Um die *Weiterentwicklungsfähigkeit* des Software-Systems zu verbessern, wurden in der Fallstudie entsprechend der Prioritäten verschiedene Reengineering-Rezepte auf

das Software-System angewandt. Dafür waren mehrere Iterationen notwendig, bei denen mehrere Qualitätsdefizite erkannt und behoben werden konnten. Eine Übersicht dieser Qualitätsdefizite ist am Ende des Kapitels in Tabelle 7.8 dargestellt. Die Baseline[1] mit der Version *V*32 des Datenverwaltungssystems ist Ausgangspunkt für das Anwenden der Reengineering-Rezepte in der Fallstudie. Für eine bessere Nachvollziehbarkeit wurde jedes zum Einsatz gebrachte Reengineering-Rezept durch eine eigene Baseline dokumentiert.

Im Folgenden werden exemplarisch die Ergebnisse der Anwendung von vier Reengineering-Rezepten im Detail erläutert, die während der Fallstudie ausgewählt wurden und zum Einsatz kamen.

- Die Anwendung des Reengineering-Rezepts *Verbindliche Spezifikation* wird in Abschnitt 7.3.1 beschrieben. Durch die Anwendung in der Fallstudie wurde das Qualitätsdefizit *Fehlende Feature-Implementierung* bei vier Entitäten festgestellt und behoben.

- Das Qualitätsdefizit *Fehlende Feature-Spezifikation* wurde mit Hilfe des Reengineering-Rezepts *Spezifizierte Implementierung* 24 mal erkannt und behoben. Im Vergleich zu anderen Qualitätsdefiziten trat dieses am häufigsten auf, Details dazu werden in Abschnitt 7.3.2 erläutert.

- Durch das Verwenden des Reengineering-Rezepts *Feature-Entschränkung* wurde das Qualitätsdefizit *Starke Feature-Verschränkung in Komponenten* fünf mal erkannt und behoben. Für das Beheben des Qualitätsdefizits wurde die Architektur des Software-Systems stark restrukturiert. Details der Verwendung dieses Reengineering-Rezepts werden in Abschnitt 7.3.3 beschrieben.

- Die Beschreibung der Anwendung des Reengineering-Rezepts *Zyklenfreie Struktur* erfolgt in Abschnitt 7.3.4. Mit Hilfe dieses Reengineering-Rezepts konnte zwei mal das Qualitätsdefizit *Zyklen zwischen Klassen* und einmal das Qualitätsdefizit *Zyklen zwischen Komponenten* erkannt und behoben werden.

---

[1] Der Begriff Baseline bezeichnet einen bestimmten Stand der Entwicklung eines Software-Produkts. D.h. alle Entitäten wie Quellcode, Dokumentation etc. werden unter einer eindeutigen Version gespeichert und sind damit referenzierbar und wiederherstellbar.

7.3.1 Anwenden des Reengineering-Rezepts Verbindliche Spezifikation

7.3.1.1 Auswählen eines Reengineering-Rezepts

Das Reengineering-Rezept *Verbindliche Spezifikation* wurde auf Grundlage der höchsten Priorität ausgewählt, um Abweichungen zum geforderten Qualitätsteilmerkmal *Vollständigkeit der Implementierung* zu erkennen und zu beheben. Damit die Implementierung vollständig ist, muss jedes spezifizierte Feature verbindlich umgesetzt sein. Das Reengineering-Rezept wurde bereits in Abschnitt 5.4.2 vorgestellt.

7.3.1.2 Anwenden der Erkennungsregeln

Die Erkennungsregel des Reengineering-Rezepts *Verbindliche Spezifikation* benutzt die Metrik *Isolierte Features IF*. Die Erkennungsregel wurde auf das Software-System mit der Baseline Version $V32$ angewandt, das Ergebnis der Metrik *Isolierte Features IF* ist in Tabelle[2] 7.4 zu finden.

Tabelle 7.4: Ergebnisse der Metrik Isolierte Features

Baseline	Anzahl Features NOF	Isolierte Features IF
$V32$	4	4
$V32.1$	4	0

Für die Auswertung der Erkennungsregel wurde das Werkzeug TraceTool eingesetzt (siehe Kapitel 6). Dafür wurden zuerst die existierende Implementierung und die Spezifikation von Features durch einen Experten untersucht. Ein Feature besteht aus einer eindeutigen Feature-Kennzeichnung und deren Spezifikation; die Feature-Kennzeichnungen wurden aus dem vom Entwicklungsteam benutzten Requirementswerkzeug exportiert und konnten dann mit dem Werkzeug TraceTool ausgewertet werden. Die Auswertung mit Hilfe der Metrik NOF ergab die Existenz von vier spezifizierten Features ($NOF = 4$, siehe Tabelle 7.4), die im Folgenden als f_1, f_2, f_3 und

[2] Die Metrik Number of Features NOF wird verwendet, um dem Leser weitere Details des Software-Systems zu verdeutlichen. Die Metrik NOF ist nicht Teil der Erkennungsregel.

f_4 bezeichnet werden. Da zu diesem Zeitpunkt geeignete Traceability-Links fehlten, mit denen die zum Feature gehörende Implementierung identifiziert werden konnte, wurden durch die Anwendung der Erkennungsregel das Qualitätsdefizit *Fehlende Feature-Implementierung* bei 4 Entitäten festgestellt (da $IF = 4$).

7.3.1.3 Beheben der Qualitätsdefizite durch Reengineering-Aktivitäten

Für das Beheben des Qualitätsdefizits *Fehlende Feature-Implementierung* wurde gemäß der Reengineering-Regel des Reengineering-Rezepts *Verbindliche Spezifikation* die Reengineering-Aktivität *Implementieren der Features* f_1, f_2, f_3, f_4 *durch Feature Driven Development* [PF01] angewandt. Die vier spezifizierten Features konnten bei der Durchführung der Reengineering-Aktivität in der Implementierung identifiziert werden. Daher wurden im nächsten Schritt Traceability-Links mit Hilfe des in Kapitel 6 beschriebenen Werkzeugs TraceTool aufgebaut, sodass zu jedem Feature auch die Entitäten der Implementierung verfolgbar sind.

7.3.1.4 Prüfung der Erfüllung der Qualitätsziele

Durch nochmaliges Anwenden der Metriken und Erkennungsregeln wurde geprüft, ob die überarbeitete Version (Baseline $V32.1$) des Software-Systems das Qualitätsziel *Weiterentwicklungsfähigkeit* ausreichend erfüllt. Ausreichend erfüllt bedeutet, dass keine weiteren Reengineering-Rezepte aus Sicht des Entwicklungsteams angewandt werden sollten. Wie in Tabelle 7.4 dargestellt, konnten die erkannten Qualitätsdefizite behoben werden ($IF = 0$) und es konnte kein weiteres Qualitätsdefizit mit Hilfe des Reengineering-Rezepts *Verbindliche Spezifikation* erkannt werden. Da es aber noch weitere anwendbare Reengineering-Rezepte gab, und damit weitere potentielle Qualitätsdefizite, welche das Qualitätsziel *Weiterentwicklungsfähigkeit* beeinträchtigen könnten, wurde mit dem Anwendungsprozess fortgefahren.

7.3.2 Anwenden des Reengineering-Rezepts Spezifizierte Implementierung

7.3.2.1 Auswählen eines Reengineering-Rezepts

Auf Grundlage der ermittelten Priorität wurde das Reengineering-Rezept *Spezifizierte Implementierung* angewandt, um Qualitätsdefizite zu erkennen und zu beheben, welche eine Abweichung zum geforderten Qualitätsteilmerkmal *Übereinstimmung von Implementierung und Spezifikation* darstellen. Jedes implementierte Feature hat nur dann eine Daseinsberechtigung, wenn es auch spezifiziert ist. Das Reengineering-Rezept wurde in Abschnitt 5.4.2 bereits vorgestellt.

7.3.2.2 Anwenden der Erkennungsregeln

Das Ergebnis der Anwendung der Erkennungsregeln des Reengineering-Rezepts *Spezifizierte Implementierung* ist in Tabelle 7.5 dargestellt (Baseline-Version $V32.1$). In den Erkennungsregeln dieses Reengineering-Rezepts wurde die Metrik *Isolierte Entitäten* $IE(X)$ benutzt.

Tabelle 7.5: Ergebnisse der Metrik Isolierte Entitäten.

Baseline	Anzahl Features NOF	Isolierte Schnittstellen $IE(S)$	Isolierte Klassen $IE(K)$	Isolierte Datendateien $IE(D)$
$V32.1$	4	46	131	9
$V32.2$	28	0	0	1

Durch die Erkennungsregeln wurden in der Baseline $V32.1$ folgendes Qualitätsdefizit erkannt, da $IE(S) > 0$, $IE(K) > 0$ und $IE(D) > 0$: *Fehlende Feature-Spezifikation.*

Als Nebenprodukt der Metriken wurden die Namen der Entitäten identifiziert, die isoliert sind. Da es eine Vielzahl von Entitäten sind (z.B. 131 Klassen), können nicht alle namentlich genannt werden. Es werden aber im Folgenden immer wieder Beispiele gegeben.

7.3.2.3 Beheben der Qualitätsdefizite durch Reengineering-Aktivitäten

Zur Behebung des Qualitätsdefizits *Fehlende Feature-Spezifikation* schlägt das Reengineering-Rezept die Reengineering-Aktivität *Rekonstruieren der Features durch Feature-Analyse* [Gre07] vor. Diese Reengineering-Aktivität wurde im Rahmen der Fallstudie mehrfach durchgeführt. Dabei wurde als erstes die Feature-Dokumentation rekonstruiert, indem die existierenden Strukturen mit Hilfe von Traceability-Links dokumentiert wurden (beispielsweise Abhängigkeitsbeziehungen zwischen Komponenten oder Klassen). Zudem wurden Feature-Modelle erzeugt, um Abhängigkeitsbeziehungen zwischen Features und Entitäten darstellen zu können. Das Vorgehen wurde durch das Werkzeug TraceTool unterstützt. Mit dem Werkzeug TraceTool wurden aus dem Quellcode sowohl Entitäten als auch Abhängigkeitsbeziehungen automatisch extrahiert und gespeichert. Dazu gehören neben den Entitäten wie etwa Komponenten A oder Klassen K auch die Abhängigkeitsbeziehungen vom Beziehungstyp Benutzt-Beziehung. Auch wurden manuell Traceability-Links vom Beziehungstyp Realisiert-Durch mit dem TraceTool erstellt, um die isolierten Entitäten zu reduzieren und ein besseres Verständnis für die existierende Implementierung zu erhalten.

Insgesamt wurden 24 Features auf Grundlage des Quellcodes rekonstruiert und jeweils innerhalb einer Spezifikation festgehalten. Beispiele von Features sind Mitarbeiterdatenverwaltung (Feature 7), Rechteverwaltung (Feature 16), Datenzugriffkapselung (Feature 21) oder Diagnosedienste (Feature 23).

7.3.2.4 Prüfung der Erfüllung der Qualitätsziele

Nach der Anwendung der Reengineering-Aktivität wurden zur Prüfung des Ergebnisses die Metriken und Erkennungsregeln erneut angewandt. In Tabelle 7.5 ist das Ergebnis dargestellt (Baseline $V32.2$). Es lassen sich die folgenden Veränderungen feststellen:

- Aus anfänglich vier dokumentierten Features wurden 24 weitere Features identifiziert: Baseline $V32.1$ mit $NOF = 4$ gegenüber Baseline $V32.2$ mit $NOF = 28$. Insgesamt ergeben sich damit 28 implementierte und spezifizierte Features des Software-Systems.

- Die isolierten Entitäten $IE(S) = 46$, $IE(A) = 131$ und $IE(D) = 9$ innerhalb der Baseline $V32.1$ konnten weitgehend behoben werden $IE(S) = 0$, $IE(K) = 0$ und

$IE(D) = 1.$

In einer weiteren Iteration wurde nun die alternative Reengineering-Aktivität verwendet, die im Reengineering-Rezept *Spezifizierte Implementierung* vorgeschlagen wird, um das noch bestehende Qualitätsdefizit zu beheben: wie der Wert von 1 der Metrik $IE(D)$ in Tabelle 7.5 zeigt, konnten in dieser Baseline nicht alle isolierten Entitäten behoben werden. In einer weiteren Iteration wurde mit Hilfe der Reengineering-Aktivität *Löschen der unnötigen Entitäten durch Refactoring* die Datendatei mit der Bezeichnung d_9 gelöscht. Dadurch entstand eine weitere Baseline $V32.2.1$, in der das Qualitätsdefizit behoben war. Da es aber noch weitere Reengineering-Rezepte gab, die noch nicht betrachtet wurden, wurde der Prozess fortgesetzt.

7.3.3 Anwenden des Reengineering-Rezepts Feature-Entschränkung

7.3.3.1 Auswählen eines Reengineering-Rezepts

Das Qualitätsteilmerkmal *Geringe Feature-Verschränkung* wird durch das Qualitätsdefizit *Starke Feature-Verschränkung in Komponenten* beeinträchtigt. Um dieses Qualitätsdefizit zu erkennen und zu beheben, wurde das Reengineering-Rezept *Feature-Entschränkung* auf das Software-System angewandt. Das Reengineering-Rezept wurde bereits in Tabelle 5.10 vorgestellt.

7.3.3.2 Anwenden der Erkennungsregeln

Die Erkennungsregel des Reengineering-Rezepts beinhaltet die Vorbedingung, dass keine isolierten Komponenten und keine isolierten Features existieren (`VORB[(IE(A) = 0) UND (IF = 0)]`). Diese Vorbedingung konnte durch die Anwendung der Reengineering-Rezepte *Verbindliche Spezifikation* und *Spezifizierte Implementierung* bereits erfüllt werden. Daher konnte durch die Metrik *Feature Tangling* $FTANG(A)$ das Software-System bewertet werden. Das Ergebnis der Anwendung der Metrik $FTANG(A)$ auf die Baseline $V32.3$ ist in Tabelle 7.6 zu sehen.

Tabelle 7.6: Ergebnis der Anwendung der Metrik $FTANG(A)$.

Baseline	Feature Tangling $FTANG(A)$
$V32.3$	$0,964$
$V32.4$	$0,482$
$V32.5$	$0,310$
$V32.6$	$0,223$
$V32.7$	$0,179$
$V32.8$	$0,149$

Wie bereits durch die Entwickler im Vorfeld der Anwendung des Reengineering-Rezepts vermutet, ist der Wert der Metrik $FTANG(A) = 0.964$ sehr hoch (der Wert 1 ist der schlechteste Wert, 0 der beste Wert), da alle 28 Features innerhalb der Komponente Datenverwaltung implementiert wurden. Im Vergleich zu den Erkennungsregeln der letzten Reengineering-Rezepte ist ein Schwellwert bei dieser Reengineering-Regel nicht absolut angebbar, sondern hängt davon ab, wie weit eine Aufteilung einer Komponente durch das Entwicklungsteam durchgeführt werden kann. Diese Entscheidung muss letztendlich das Entwicklungsteam treffen. Wie in Tabelle 7.6 zu sehen, hat sich der Wert von $0,149$ für dieses Software-System als ausreichend erwiesen.

7.3.3.3 Beheben des Qualitätsdefizits durch Reengineering-Aktivitäten

Das Reengineering-Rezept *Feature-Entschränkung* enthält zwei alternative Reengineering-Aktivitäten zum Beheben des erkannten Qualitätsdefizits *Starke Feature-Verschränkung in Komponenten*. In Abbildung 7.2 sind die beiden Reengineering-Aktivitäten, *Restrukturieren der Komponente* Datenverwaltung *durch Dekomposition* [Par72] oder *Restrukturieren der Features* $f_1 \ldots f_{28}$ *durch Zusammenführen* [PT93] schematisch dargestellt.

Durch die aufgebauten Traceability-Links vom Typ Realisiert-Durch zwischen den Features $f_1 \ldots f_{28}$ und der Komponente Datenverwaltung bzw. den Klassen dieser Komponente, war es möglich aufzuzeigen, welche Features durch welche Klassen realisiert wurden. In Abbildung 7.2 wird schematisch die Komponente Datenverwaltung mit einer exemplarischen Menge ihrer Klassen gezeigt. Durch die Schattierung der Klassen werden dabei unterschiedliche Klassen hervorgehoben, die jeweils gemeinsam

haben, dass sie ein bestimmtes Feature realisieren.

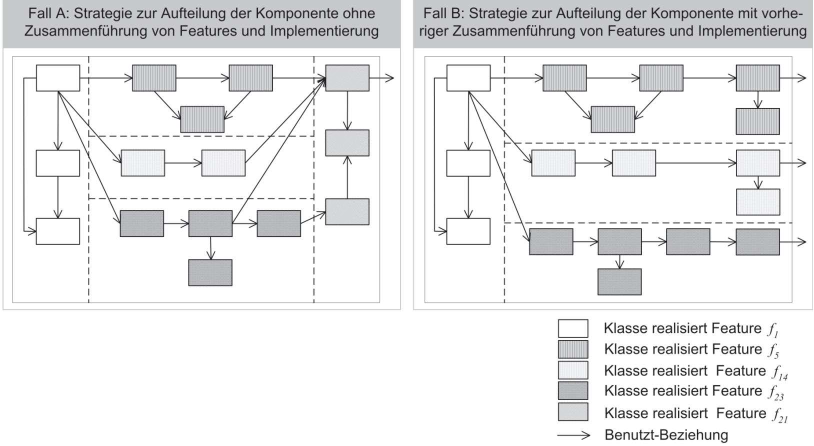


Abbildung 7.2: Zwei mögliche Strategien zur Aufteilung der Implementierung bezüglich der Features in der Komponente Datenverwaltung.

Die beiden alternativen Reengineering-Aktivitäten wurden durch das Entwicklungsteam diskutiert, da sie beide für das Beheben des Qualitätsdefizits *Starke Feature-Verschränkung in Komponenten* eingesetzt werden können:

1. Fall A sieht die Reengineering-Aktivität *Restrukturieren der Komponente* Datenverwaltung *durch Dekomposition* vor. Dabei wird die Komponente Datenverwaltung aufgeteilt, sodass idealerweise jeweils ein Feature in einer Komponente realisiert wird. Für die Entitäten, welche das Feature Datenzugriffkapselung (f_{21}) realisieren, wird eine neue Komponente erstellt, sodass Klassen, die das Feature f_{21} umsetzen, den Datenzugriff kapseln.

2. Fall B besteht in der Anwendung der Reengineering-Aktivität *Restrukturieren der Features* $f_1 \ \ldots \ f_{28}$ *durch Zusammenführen* [PT93]. Das Feature Datenzugriffkapselung (f_{21}) wird in kleinere Einheiten aufgeteilt und in die bestehenden Features integriert, wodurch das Feature f_{21} aufgelöst und mit anderen Features zusammengeführt wird. Dies ist in Abbildung 7.2 (Fall B) dadurch zu sehen, dass es keine Klasse mehr gibt, die das Feature f_{21} realisiert, sondern die Features werden durch die anderen Klassen der Komponente realisiert. Nachteil dieser

Strategie ist allerdings, dass der Zugriff auf – im Sinne einer Schichtenarchitektur – tiefer liegende Komponenten (wie etwa die Komponente Steuerungsdienst oder die Komponente Sicherheitsdienst in Abbildung 7.1) nicht mehr gekapselt wäre.

Da beide Strategien zu einer Verbesserung der *Weiterentwicklungsfähigkeit* beitragen, Fall A dem Entwicklungsteam aber weniger aufwendig erschien, wurde die Reengineering-Aktivität *Restrukturieren der Komponente* Datenverwaltung *durch Dekomposition* angewandt. Wie in Tabelle 7.6 dargestellt ist, waren dafür mehrere Iterationen notwendig. Die Komponente wurde so aufgeteilt, dass angelehnt an eine Schichtenarchitektur die Implementierung vom Feature Datenzugriffkapselung (f_{21}) in die neue Komponente Datenzugriff ausgelagert wurde.

Dies hat den Vorteil, dass eine Änderung der Datenhaltungsschicht keine Änderung der Anwendungslogik zur Folge hat und die jetzige Implementierung nicht bezüglich der weiteren existierenden Features, wie in Fall B beschrieben, aufgeteilt werden muss (siehe Abbildung 7.2 rechts). Der Nachteil der Lösung ist aber, dass sehr große Klassen erhalten bleiben (wie etwa Klasse k_{74}, siehe Tabelle D.2 im Anhang), die das Qualitätsmerkmal *Geringe Strukturelle Komplexität* beeinträchtigen.

Die Konsequenz dieser Entscheidung ist allerdings auch, dass das Feature Datenzugriffkapselung (f_{21}) als sehr großes Feature nicht aufgeteilt werden kann. Die Größe eines Features wird durch die Metrik *Feature-Größe* festgestellt, indem sie die Anzahl von Entitäten berechnet, die ein Feature realisieren. Die *Feature-Größe* wurde für alle Features berechnet, das Ergebnis dieser Berechnung ist in Abbildung 7.3 zu sehen. Beim Betrachten dieses Diagramms zeigt sich, dass Feature f_{21} im Vergleich zu allen anderen Features sehr groß ist. Deswegen wurde das Feature Datenzugriffkapselung (f_{21}), mit einer *Feature-Größe* von 68 Entitäten, vom Entwicklungsteam als Ausreißer akzeptiert, da man sich für Fall A entschieden hat.

Nach dieser Diskussion der Alternativen wird im Folgenden nun vorgestellt, wie das Durchführen der Reengineering-Aktivität abgelaufen ist.

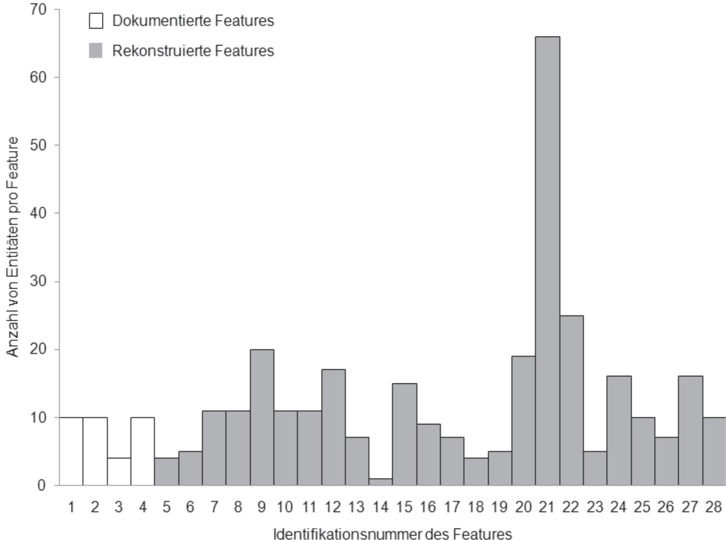


Abbildung 7.3: Übersicht über die rekonstruierten Features.

Für die Dekomposition wurde die neue Komponente Datenzugriff erstellt und alle Klassen und Schnittstellen, welche für die Implementierung des Features Datenzugriff-kapselung (f_{21}) verantwortlich sind, der neuen Komponente untergeordnet. Hierbei wurden die Restrukturierungen *Move Class*[3] und *Move Interfaces* angewandt.

Dabei wurde Quellcode entdeckt (Klasse k_{35}), der auf Daten zugreift, obwohl diese Daten überhaupt nicht für das Feature benötigt werden. Mittels der Restrukturierung *Create Class* wurde eine neue Klasse in der Komponente Datenzugriff erzeugt, welche nur den benötigten Code enthält. Außerdem wurden die Restrukturierungen *Move Method* und *Move Attribute* eingesetzt.

Zudem mussten Konstanten, welche zum Protokollieren von eindeutigen Nachrichten benötigt werden, in die dafür verantwortliche Klasse mittels *Move Attributtes* verschoben werden. Die Restrukturierungen *Create Package* und *Rename Package* wurden zum Anpassen und Erstellen der Paketstruktur eingesetzt, um den neuen Aufbau der Funktionalität zu verdeutlichen. Bei der Verbesserungsmaßnahme wurde

[3]Im Folgenden werden mehrfach Restrukturierungen angewandt, die Fowler [Fow99] als Refactoring-Operationen bezeichnet und eingeführt hat.

die Restrukturierung *Rename Package* angewandt, um nicht mehr sichtbare Klassen innerhalb der neuen Komponente sichtbar und dadurch nutzbar zu machen.

Schließlich wurde durch die Restrukturierung *Change Method Signature* bei einigen Methoden der Zugriffsmodifikator geändert, damit Zugriffe auf Klassen in den neuen Paketen möglich sind.

7.3.3.4 Prüfung der Erfüllung der Qualitätsziele

Nach der durchgeführten Reengineering-Aktivität wurde mit dem TraceTool die Erkennungsregel nochmals ausgeführt. Das Ergebnis, wurde bereits in Tabelle 7.6 dokumentiert. Zur Dokumentation wurde vorher eine neue Baseline mit der Version $V32.4$ im Repository erstellt. Das Ergebnis der Dekomposition ist in Abbildung 7.4 links im UML Komponentenmodell abgebildet.

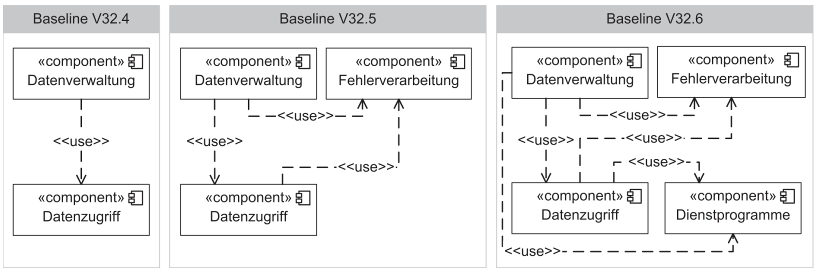


Abbildung 7.4: Komponentenmodelle der Baselines $V32.4$, $V32.5$ und $V32.6$.

Primäres Ziel der Reengineering-Aktivität war es, die Komponente so aufzuteilen, dass immer ein Feature in einer Komponente realisiert wird. Es konnte durch die durchgeführten Reengineering-Aktivitäten der Wert der Metrik *Feature Tangling* $FTANG(A)$ von $0,964$ auf $0,482$ reduziert werden. Es wurde ungefähr ein Viertel der Klassen des Software-Systems in eine eigene Komponente ausgelagert, da sie das Feature Datenzugriffkapselung (f_{21}) realisierten. Dieses Vorgehen hatte auch den Vorteil, dass eine weitere Strukturierung innerhalb der Komponente mittels Paketen durchgeführt werden konnte, die das Programmverstehen erhöhte[4].

[4]Die Einhaltung der 7 ± 2 Regel [Mil56] war beispielsweise einfach möglich.

Das Reengineering-Rezept *Feature-Entschränkung* wurde in mehreren Iterationen wiederholt angewandt, um das Qualitätsdefizit *Starke Feature-Verschränkung in Komponenten* zu beheben. Ziel war es dabei, das Software-System weiter aufzuteilen, sodass die Erkennungsregel des Reengineering-Rezepts erfüllt wird. In jeder Iteration wurde dabei die Reengineering-Aktivität *Restrukturieren der Komponenten durch Dekomposition* [Par72] angewandt und das Software-System überarbeitet. Das überarbeitete Produkt wurde jeweils durch eine eigene Baseline dokumentiert. Wie in Tabelle 7.6 zu sehen, entstanden so die Baselines $V32.5$, $V32.6$, $V32.7$ und $V32.8$. Durch die durchgeführten Reengineering-Aktivitäten wurde die Architektur des Systems grundlegend verändert, wie in Abbildung 7.5 erkennbar ist.

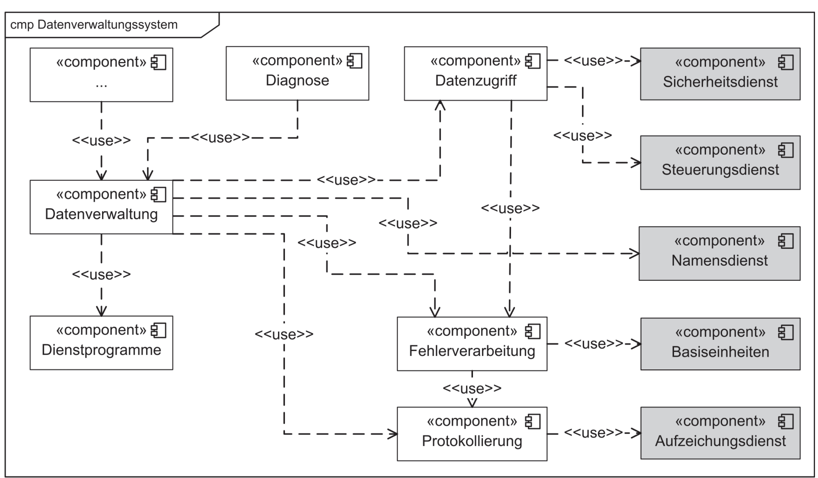


Abbildung 7.5: Datenverwaltungssystem nach der Rekonstruktion (Baseline $V32.8$).

Wie bereits erwähnt, wurde der Wert von $FTANG(A)$ als ausreichend niedrig akzeptiert, sodass das Qualitätsdefizit in Baseline $V32.8$ als behoben angesehen wurde. Da es aber noch weitere Reengineering-Rezepte gab, die noch nicht betrachtet wurden, wurde der Prozess fortgesetzt.

7.3.4 Anwenden des Reengineering-Rezepts Zyklenfreie Struktur

7.3.4.1 Auswählen eines Reengineering-Rezepts

Das Reengineering-Rezept *Zyklenfreie Struktur* wurde entsprechend der Priorität ausgewählt. Es dient dazu, Qualitätsdefizite zu erkennen, welche das Qualitätsteilmerkmal *Zyklenfreiheit* beeinträchtigen. Das Qualitätsdefizit *Zyklen zwischen Klassen* ist ein Beispiel dafür: Klassenstrukturen, die hierarchisch aufgebaut sein sollen, dürfen keine Zyklen enthalten. Das Reengineering-Rezept *Zyklenfreie Struktur* wurde innerhalb der Fallstudie angewandt und ist in Abschnitt 5.4.3.2 beschrieben.

7.3.4.2 Anwenden der Erkennungsregeln

Die Erkennungsregel des Reengineering-Rezepts *Zyklenfreie Struktur* benutzt die Metriken *Anzahl Klassen in Zyklen* (*KLIZ*) und *Anzahl Komponenten in Zyklen* (*KOIZ*), mit deren Hilfe die Qualitätsdefizite *Zyklen zwischen Klassen* und *Zyklen zwischen Komponenten* erkannt wurden. Wie in Tabelle 7.7 dargestellt, wurde innerhalb der Baseline *V*32.8 des Software-Systems das Qualitätsdefizit Zyklen zwischen Klassen zweimal erkannt (*KLIZ* mit dem Wert 2) und das Qualitätsdefizit *Zyklen zwischen Komponenten* einmal erkannt (*KOIZ* mit dem Wert 1).

Tabelle 7.7: Ergebnisse der Metriken *KOIZ* und *KLIZ*.

Baseline	Anzahl Komponenten in Zyklen *KOIZ*	Anzahl der Klassen in Zyklen *KLIZ*
*V*32.8	1	2
*V*32.9	0	0

Mit Hilfe der Metriken konnten die für die Zyklen verantwortlichen Klassen, wie etwa Klasse Sortierer und Klasse Ergebnis, identifiziert werden. Diese sind in Tabelle D.2 als k_7 bzw. k_9 benannt worden. Da die Metriken im Werkzeug TraceTool bei jeder Iteration auch automatisch ausgewertet werden und bei der Version *V*32.8 (im Vergleich zur ursprünglichen Version *V*32) bei der Metrik KOIZ der Wert 1 auftauchte, wurde dem

Entwicklungsteam klar, dass das für diese Veränderung verantwortliche Qualitätsdefizit *Zyklen zwischen Komponenten* durch die letzten Reengineering-Aktivitäten eingebaut wurde. Die letzten Reengineering-Aktivitäten hatten zum Ziel, das Qualitätsdefizit *Starke Feature-Verschränkung in Komponenten* zu beheben (vgl. Abschnitt 7.3.3):

Beim Verschieben der Klassen und Schnittstellen in die neue Komponente Datenzugriff wurde ein Zyklus zwischen den Komponenten Datenverwaltung und Datenzugriff erzeugt; eine spezifische Funktionalität vom Feature Datenzugriffkapselung (f_{21}) wurde übersehen, nämlich die Möglichkeit der Konvertierung von technischen Objekten zu Objekten der Anwendungslogik. Dies wurde durch die Metrik ***KOIZ*** mit dem Wert 1 erkannt. Der Zyklus ergibt sich, da die Komponente Datenverwaltung die Komponente Datenzugriff verwendet, die Komponente Datenzugriff aber auch die Komponente Datenverwaltung. Dies ist ein Beispiel für eine erfolgreiche Regressionsprüfung anhand der Reengineering-Rezepte.

7.3.4.3 Beheben der Qualitätsdefizite durch Reengineering-Aktivitäten

Zur Behebung des Qualitätsdefizits *Zyklen zwischen Klassen* wurde gemäß der Reengineering-Regel des Reengineering-Rezepts *Zyklenfreie Struktur* die Reengineering-Aktivität *Restrukturieren der Klassen durch Aufspaltung* [PT93] angewandt. Bei näherer Betrachtung des Quellcodes war der Zyklus zwischen den Klassen Sortierer und Ergebnis schnell erkannt. Dabei verwendet die Klasse Sortierer die Klasse Ergebnis und die Klasse Ergebnis aber auch die Klasse Sortierer (siehe Abbildung 7.6 und Listing 7.1).

Listing 7.1: Beispielausschnitt des Quellcodes mit Qualitätsdefizit

```
01 public class Sortierer { ...
02   Collection sortieren(Collection nichtsortiert, Comparator c){
03   if (nichtsortiert instanceof Ergebnis){
04     ...
05     return sortiert;
06   }
07   if (nichtsortiert instanceof Set){
08     ...
09     return sortiert;
10   } else if (nichtsortiert instanceof List)
```

```
11    { ...        }
12    else if ...
13    ...
```

Bei der Klasse Sortierer handelt es sich um eine Klasse zur Sortierung von verschiedenen Objekttypen. Zum Beheben des Qualitätsdefizits *Zyklen zwischen Klassen* wurde die Klasse Sortierer und die Klasse Ergebnis geändert. Da die Klasse Ergebnis immer sortierte Ergebnisse enthält (wird durch Funktionen der Vaterklasse ermöglicht) ist ein Aufruf der Klasse Sortierer nicht notwendig, deswegen wurde die Benutzt-Beziehung von der Klasse Ergebnis zur Klasse Sortierer aufgelöst.

Listing 7.2: Beispielausschnitt des Quellcodes ohne Qualitätsdefizit

```
01 public class Sortierer { ...
02   Collection sortieren(Collection nichtsortiert, Comparator c){
03   if (nichtsortiert instanceof Set){
04     ...
05     return sortiert;
06   } else if (nichtsortiert instanceof List)
07   { ... }
08   else if ...
09   ...
```

Wie im Listing 7.2 zu sehen, wurde zudem in der Klasse Sortierer vier Zeilen entfernt. Da wie bereits erwähnt die relevante Sortierer-Funktion in der Klasse Ergebnis durch die Vaterklasse Set übernommen wird ist keine zusätzliche Fallunterscheidung mehr für diesen Fall erforderlich. Das Löschen des relevanten Quellcodes in der Klasse Sortierer entspricht nicht vollständig der Aufspaltung nach [BC03], da der Quellcode nicht zu einer neue Klasse verschoben, sondern entfernt wurde.

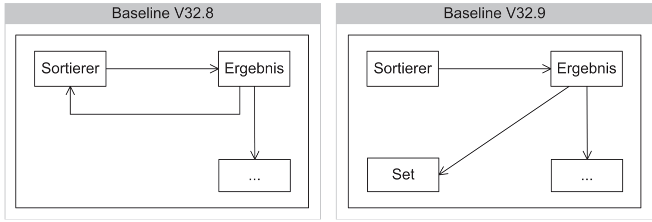


Abbildung 7.6: Behebung des Qualitätsdefizits Zyklen zwischen Klassen in der Komponente Dienstprogramme.

Um den Zyklus zwischen den beiden Komponenten Datenverwaltung und Datenzugriff zu beheben, wurde die Reengineering-Aktivität *Restrukturieren der Komponenten Datenverwaltung und Datenzugriff durch Demotion* [Lak96] angewandt. Dabei wurden beispielsweise die Teile der Klassen, welche für die Konvertierung zuständig sind, durch die Restrukturierung *Move Method* zurückgeführt.

7.3.4.4 Prüfung der Erfüllung der Qualitätsziele

Nach der Durchführung der Reengineering-Aktivität wurde die Baseline $V32.9$ erstellt, die Erkennungsregel dann nochmals angewandt und die Bedingung geprüft. In der neuen Baseline lässt sich anhand der Metriken (siehe Tabelle 7.7) feststellen, dass alle erkannten Zyklen behoben wurden: Baseline $V32.8$ mit $KLIZ = 2$, $KOIZ = 1$ gegenüber Baseline $V32.9$ mit $KLIZ = 0$, $KOIZ = 0$.

7.4 Zusammenfassung der Ergebnisse

Die Reengineering-Rezepte, welche bezüglich des Qualitätsziels *Weiterentwicklungsfähigkeit* mit Hilfe des vorgestellten Aufbauprozesses erstellt wurden, konnten erfolgreich auf das Datenverwaltungssystem angewandt werden. Dabei konnten mehrere verschiedene Qualitätsdefizite im Software-System erkannt und behoben werden und so die Qualität des Software-Systems bezüglich der *Weiterentwicklungsfähigkeit* deutlich verbessert werden. Es konnten dabei Qualitätsdefizite sowohl innerhalb der Spezifikation, als auch innerhalb der Architektur und der Implementierung aufgedeckt

und beseitigt werden. Tabelle 7.8 zeigt alle während der Fallstudie erkannten und behobenen Qualitätsdefizite.

Es hat sich beispielsweise gezeigt, dass die gefundenen isolierten Entitäten ein klares Indiz für fehlende Spezifikationen von Features sind (Qualitätsdefizit *Fehlende Feature-Spezifikation*), was die Qualitätsteilziele *Verfolgbarkeit*, *Testbarkeit* und *Wiederverwendbarkeit* stark beeinflusste. Ausgehend von ursprünglich 4 spezifizierten Features konnten am Ende weitere 24 Features identifiziert werden, sodass insgesamt 28 Features bei der Weiterentwicklung des Software-Systems berücksichtigt werden müssen.

Die praktische Anwendbarkeit der Reengineering-Rezepte innerhalb der Entwicklung und der große Nutzen der Metriken im Zusammenspiel mit den Erkennungsregeln zur Interpretation der Metriken wurde demonstriert. So war es möglich, die Erkennungsregeln innerhalb eines Werkzeugs TraceTool umzusetzen und damit die Erkennung von Qualitätsdefiziten zu automatisieren. Mit Hilfe des Werkzeugs TraceTool war eine einfache Regressionsprüfung nach Durchführung der Reengineering-Aktivitäten möglich. Damit wurden auch Qualitätsdefizite erkannt, die durch eine Reengineering-Aktivität entstanden sind, wie etwa das Qualitätsdefizit *Zyklen zwischen Komponenten*, das in Abschnitt 7.3.4 beschrieben wurde.

Tabelle 7.8: Erkannte und behobene Qualitätsdefizite der Fallstudie.

Qualitätsdefizite	erkannt	behoben
Fehlende Feature-Spezifikation	24	24
Fehlende Feature-Implementierung	4	4
Starke Feature-Verschränkung in Komponenten	5	5
Starke Feature-Streuung in Komponenten	0	0
Unausgewogen aufgeteilter Code	2	0
Unnötige Feature-Spezifikation	0	0
Zyklen zwischen Klassen	2	2
Zyklen zwischen Komponenten	1	1

Der Nutzen des Werkzeugs TraceTool im Zusammenspiel mit den Reengineering-Rezepten zeigte sich insbesondere bei der Behebung des Qualitätsdefizits *Starke Feature-Verschränkung in Komponenten*, das nur durch mehrere Iterationen beho-

ben werden konnte. Durch die Metrik *FTANG* wurde dieses Qualitätsdefizit in jeder Iteration erkannt. Durch die Anwendung der von den Reengineering-Regeln vorgeschlagenen Reengineering-Aktivitäten konnte das Qualitätsdefizit *Starke Feature-Verschränkung in Komponenten* in jeder Iteration schrittweise reduziert werden. Der Verlauf wird in Abbildung 7.7 veranschaulicht, in der zu sehen ist, dass der Wert der Metrik *FTANG* von 0,964 auf 0,149 sinkt. Dafür wurden fünf Reengineering-Iterationen benötigt, die jeweils in fünf unabhängigen Baselines resultierten (Baselines *V*32.4, *V*32.5, *V*32.6, *V*32.7 und *V*32.8). Zwischen jeder der fünf Baselines konnte relativ zur vorherigen Baseline eine Reduzierung des Ergebnisses der Metrik *Feature Tangling* (*FTANG*) erreicht werden; insgesamt konnte *Feature Tangling* um 85% reduziert wurden.

Durch Anwenden der Reengineering-Aktivitäten wurden primär Entitäten, die technische Features umsetzen, in eigene Komponenten ausgelagert, um so eine Trennung in Fach- und Anwendungslogik zu ermöglichen. Daher besteht das System nun aus insgesamt sechs Komponenten plus den anfangs bereits existierenden fünf Blackbox-Komponenten, wie in Abbildung 7.5 bereits zu sehen war. Die Features Diagnosedienste, Programmprotokollierung und Fehlerbehandlung konnten jeweils in eigene Komponenten ausgelagert werden, ohne eine weitere Feature-Verschränkung oder Feature-Zerstreuung zu verursachen.

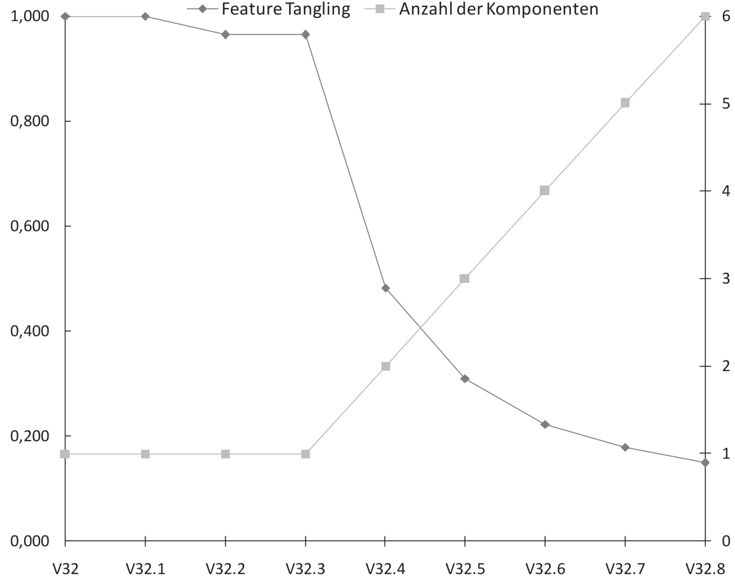


Abbildung 7.7: Reduzierung von Feature-Verschränkung pro Baseline mit Hilfe von durchgeführten Reengineering-Aktivitäten.

Zudem konnten durch die Anwendung der Reengineering-Aktivitäten auch ausgewogenere Komponenten erzielt werden: die durchschnittliche Anzahl der Entitäten ($\varnothing X/A$), die innerhalb einer Komponente gruppiert werden, konnte, wie Tabelle 7.9 zu sehen ist, von 185 auf 30 reduziert werden. Auch die durchschnittliche Anzahl von Programmzeilen pro Komponente (in der Tabelle als $\varnothing SLOC/A$ bezeichnet) ging von anfangs 34744 auf 5860 zurück, wodurch kompaktere Komponenten entstanden. Dass kompaktere Komponenten entstanden sind, zeigt sich auch durch die durchschnittliche Anzahl von Klassen pro Komponente, die von 131 auf 22 zurück ging und durch die durchschnittliche Anzahl der Schnittstellen pro Komponente, die von 46 auf 8 reduziert werden konnte (in der Tabelle als $\varnothing K/A$ und $\varnothing S/A$ bezeichnet).

Tabelle 7.9: Vergleich der Baselines vor und nach der Anwendung der Reengineering-Rezepte.

Baseline	$\varnothing K/A$	$\varnothing S/A$	$\varnothing X/A$	$\varnothing SLOC/A$
vorher ($V32$)	131	46	185	34744
nachher ($V32.9$)	22	8	31	5860

Insgesamt wurden also im Rahmen der Fallstudie 9 Baselines nach Anwendung von durch Reengineering-Rezepte vorgeschlagene Reengineering-Aktivitäten erstellt. Der Anwendungsprozess innerhalb der Fallstudie wurde mit der Baseline $V32.9$ erfolgreich beendet, da anhand der priorisierten Reengineering-Rezepte keine Qualitätsdefizite mehr erkannt werden konnten.

Kapitel 8

Fazit und Ausblick

Im folgendem Abschnitt 8.1 werden die Ergebnisse der Arbeit hinsichtlich der anfangs in Abschnitt 2.7 definierten Ziele dargestellt. Im darauf folgenden Abschnitt 8.2 werden mögliche zukünftige Arbeiten vorgeschlagen.

8.1 Ergebnisse

In dieser Arbeit wurde eine Methode erarbeitet, mit der ausgehend von einem Qualitätsziel die systematische Analyse des Qualitätsziels, die objektive Erkennung von Qualitätsdefiziten und deren Behebung ermöglicht wird. Die Methode ermöglicht das Erstellen von Reengineering-Rezepten auf Grundlage der systematischen Analyse eines Qualitätsziels. Durch den regelbasierten Aufbau der Reengineering-Rezepte ist deren Anwendung stärker automatisierbar, sodass sie frühzeitig und damit bereits während des Entwicklungsprozesses zur Erkennung und Behebung von Qualitätsdefiziten eingesetzt werden können.

Die Methode wurde am Beispiel des Qualitätsziels *Weiterentwicklungsfähigkeit* erfolgreich angewandt. Dabei wurden die Eigenschaften des Qualitätsziels *Weiterentwicklungsfähigkeit* innerhalb des Aufbauprozesses dieser Methode systematisch untersucht. Das Ergebnis dieser Untersuchung ist ein neu erarbeiteter Zuordnungsgraph, welcher die *Weiterentwicklungsfähigkeit* durch konkrete Qualitätsteilziele, Qualitätsmerkmale und Qualitätsteilmerkmale verfeinert. Darauf aufbauend konnten Reengineering-Rezepte erstellt werden, um Qualitätsdefizite zu erkennen und zu beheben,

welche die Qualitätsteilmerkmale des Qualitätsziels *Weiterentwicklungsfähigkeit* beeinträchtigen.

Das am Qualitätsziel orientierte Vorgehen hat auch den Vorteil, dass die aufgebauten Reengineering-Rezepte im Anwendungsprozess gemäß der Qualitätsziele des Projekts erst priorisiert und anschließend eingesetzt werden können. Dadurch ist die Anwendung der Reengineering-Rezepte steuerbar: Innerhalb der Fallstudie wurden beispielsweise die Qualitätsteilmerkmale wie *Übereinstimmung von Implementierung und Spezifikation* oder auch *Vollständigkeit der Implementierung* höher priorisiert als *Hohe Kohäsion* einer Klasse, was zur Folge hat, dass nur damit verbundene Reengineering-Aktivitäten angewandt werden. Dies ist auch ein Beispiel dafür, dass Konflikte zwischen Qualitätsteilzielen aufgrund der durchgeführten Verfeinerung genauer benannt und durch die Möglichkeit der Priorisierung gelöst werden können.

Dass die hier vorgestellte Methode auch in der Praxis anwendbar ist, wurde anhand einer industriellen Fallstudie gezeigt. In der Fallstudie wurde ein Ausschnitt eines industriellen Software-Produkts mit Hilfe der gewonnenen Reengineering-Rezepte und des Zuordnungsgraphen zwischen Qualitätsziel und Reengineering-Aktivitäten bewertet.

Es wurden zahlreiche Qualitätsdefizite im Bereich der Spezifikation, Architektur und Implementierung aufgedeckt, welche durch die in den Reengineering-Rezepten definierten Reengineering-Aktivitäten weitgehend behoben werden konnten. Ausgehend von nur wenigen spezifizierten Features konnten beispielsweise im Verlauf der Studie viele weitere Features identifiziert und dokumentiert werden. Zudem konnte die Feature-Verschränkung in Komponenten stark verbessert werden, wie durch die Metriken in Abbildung 7.7 zu sehen ist. Insbesondere konnten mit Hilfe des Werkzeugs TraceTool Traceability-Links erstellt werden, welche die Verfolgung von Features ermöglichten und damit auch das Verstehen des Software-Systems entscheidend erhöhten. Die *Weiterentwicklungsfähigkeit* des in der Fallstudie untersuchten Software-Systems konnte so bewertet und entscheidend verbessert werden. Es wurden dabei auch neue Metriken hergeleitet und veröffentlicht [BR08b, BBR09].

Die prototypische Umsetzung der Reengineering-Rezepte innerhalb des Werkzeugs TraceTool wurde vor allem durch den regelbasierten Ansatz der Methode ermöglicht. Während der Fallstudie konnte mit Hilfe des Werkzeugs in jeder Iteration, in der Qualitätsdefizite behoben wurden, auch effektiv eine Regressionsprüfung durchgeführt werden. Anhand der erzielten Ergebnisse beurteilte das Entwicklungsteam die

vorgestellte Methode dieser Arbeit als sehr nutzbringend. Beispielsweise wurde das Qualitätsdefizit *Zyklen zwischen Komponenten* durch eine Reengineering-Aktivität verursacht, die eigentlich das Ziel hatte, ein anderes existierendes Qualitätsdefizit zu beheben. Durch die durchgeführte Regressionsprüfung konnte das Qualitätsdefizit aber sofort erkannt und innerhalb einer weiteren Iteration behoben werden.

Ein weiterer Vorteil dieser Methode ist deren Anpassbarkeit. Es können beispielsweise neue Qualitätsteilziele integriert werden oder Reengineering-Rezepte angepasst werden, indem die im Reengineering-Rezept definierten Erkennungs- und Reengineering-Regeln erweitert werden. Beispielsweise können sehr leicht neue Metriken oder Reengineering-Aktivitäten integriert oder auch alte Metriken ausgeschlossen werden.

8.2 Ausblick

Obwohl alle gestellten Ziele erreicht wurden, kann der Ansatz noch sinnvoll erweitert werden.

Die für das Qualitätsziel *Weiterentwicklungsfähigkeit* erstellten Reengineering-Rezepte können auf betriebliche Informationssysteme angewandt werden, welche anhand des Objekt-Orientierten Paradigmas entwickelt wurden. Um solche Typen von Software-Systemen umfangreicher oder generell auch andere Software-Systeme auf existierende Qualitätsdefizite bewerten zu können, sollte der existierende Zuordnungsgraph zwischen Qualitätsziel *Weiterentwicklungsfähigkeit* und Reengineering-Aktivitäten erweitert und zusätzliche Reengineering-Rezepte erarbeitet werden.

Die Methode wurde anhand des Qualitätsziels *Weiterentwicklungsfähigkeit* erfolgreich angewandt. Es bietet sich daher an, die Methode hinsichtlich weiterer Qualitätsziele, wie etwa der Gebrauchstauglichkeit, anzuwenden und einen neuen Zuordnungsgraphen zwischen Qualitätsziel und Reengineering-Aktivitäten als auch neue Reengineering-Rezepte aufzustellen. Zur Aufstellung weiterer Reengineering-Rezepte könnten Arbeiten zur qualitätszielorientierten Bewertung von Architektur-Entscheidungen genutzt werden, wie beispielsweise die Arbeit von Bode [Bod11].

Die erarbeitete Methode unterstützt das Analysieren von Qualitätszielen, wobei im Aufbauprozess immer von einem Qualitätsziel ausgegangen wird. In diesem Zusammenhang wäre eine Erweiterung des Aufbauprozesses denkbar, sodass mehrere Qualitätsziele auf einmal berücksichtigt werden können und so Konflikte zwischen verschie-

denen Qualitätszielen aufgrund der durchgeführten Verfeinerung schon während des Aufbauprozesses benannt werden können. Falls es gemeinsame Qualitätsteilmerkmale gibt, erhalten die Qualitätsteilmerkmale durch dieses Verfahren automatisch eine höhere Gewichtung.

Der Anwendungsprozess könnte durch das Hinzufügen einer weiteren Aktivität verbessert werden, die zur Voreinstellung bzw. Parametrisierung der anzuwendenden Reengineering-Rezepte, wie etwa der Werte in den Erkennungsregeln, benutzt werden kann. Dadurch wird es möglich, auf projektspezifische Qualitätsstandards oder Unternehmensrichtlinien einzugehen und Expertenwissen zu konservieren.

Die Zielgruppe, für welche die Methode dieser Arbeit primär erstellt wurde, sind Architekten, Entwickler und Qualitätsingenieure. Allerdings könnten die Ergebnisse der Bewertungsmethode für das Management soweit aufbereitet und konsolidiert werden, dass sie als Grundlage für Entscheidungen hinsichtlich des Reengineering eines Software-Systems dienen können. Denkbar wäre es, eine Gesamtaussage zur Systemqualität abzuleiten und diese dann dem Geschäftswert des betrachteten Systems gegenüberzustellen. Ist beispielsweise die Systemqualität eines Software-Systems gering und der Geschäftswert hoch, würde dieser Zusammenhang für ein Reengineering des Software-Systems sprechen.

Es sind verschiedene Verbesserungen im Bereich der Werkzeugunterstützung denkbar, indem speziell das Verfolgen von Abhängigkeitsbeziehungen zwischen Entitäten schon während des Forward-Engineerings stärker automatisiert wird. Dabei sollte es eine bessere Integration von Entwicklungsmodellen geben. Beispielweise könnten mit Hilfe von speziellen Repositories Anforderungsmodelle über Feature-Modelle mit Architektur-Modellen stärker gekoppelt und hinsichtlich ihrer Qualität automatisch bewertet werden. Hierfür gibt es bereits erste Ansätze aus einer Forschungsgruppe der TU Ilmenau (siehe [BR10] und [BR11]).

Die automatische Durchführung von Reengineering-Aktivitäten, um durch Erkennungsregeln erkannte Qualitätsdefizite zu beheben, würde den Aufwand für diese Verbesserungsmaßnahmen reduzieren. Hierfür könnten die Reengineering-Regeln im Werkzeug umgesetzt und hinsichtlich dieser Funktionalität ausgebaut werden, da sie bereits die anzuwendenden Reengineering-Aktivitäten dokumentieren.

Weiterhin sollte es möglich sein, die Identifizierung von Features und ihrer Eigenschaften im Quellcode durch ein formales Verfahren zu automatisieren. Das Verfahren

Formal Concept Analysis könnte hierfür verwendet werden. Es wurde von Bojic und Valesevic [BV00] bereits eingesetzt, um Anwendungsfalldiagramme aus bestehendem Quellcode zu erzeugen.

Die bisherige Implementierung des Werkzeugs benutzt Abhängigkeitsbeziehungen auf der Ebene von Klassen. Um die Implementierung eines Features genauer bewerten zu können, müssten aber auch quellcodespezifische Entitäten, wie etwa Methoden und Attribute einer Klasse, berücksichtigt werden können. Durch mögliche Erweiterungen des Werkzeugs TraceTool, könnten diese Entitätstypen dann auch Gegenstand der Bewertung durch Metriken sein. Zudem könnten die genaueren Informationen über die quellcodespezifische Entitäten und ihrer Abhängigkeitsbeziehungen genutzt werden, um die Dokumentation einer Komponente und deren Schnittstellen zu verbessern. Besonders bei COTS-Komponenten (Commercial Off-The-Shelf) besteht nach [VMD+03] das Problem, dass die Komponenten-Lieferanten die Software-Komponenten bauen, ohne zu wissen, wie die Applikation des Komponenten-Nutzers aussieht. Der Komponenten-Nutzer hat oft auch keinen Zugriff auf den Quellcode, daher ist er auf eine korrekte und vollständige Dokumentation angewiesen.

Anhang A

Reengineering-Rezepte

Die für das Qualitätsziel *Weiterentwicklungsfähigkeit* erstellten Reengineering-Rezepte verwenden Metriken, Qualitätsdefizite und Reengineering-Aktivitäten aus dem im Aufbauprozess erstellten Zuordnungsgraphen. Eine Beschreibung der Qualitätsdefizite ist in Abschnitt A.1 zu finden. Die verwendeten Metriken werden in Abschnitt A.2 aufgeführt. Die in den Reengineering-Rezepten verwendeten Reengineering-Aktivitäten sind in Abschnitt A.3 beschrieben. Schließlich werden in Abschnitt A.4 die für das Qualitätsziel *Weiterentwicklungsfähigkeit* erstellten Reengineering-Rezepte vorgestellt.

A.1 Qualitätsdefizite

Im Folgenden werden alle Qualitätsdefizite beschrieben, die in dieser Arbeit ausgehend vom Qualitätsziel *Weiterentwicklungsfähigkeit* ermittelt wurden. Die von dem jeweiligen Qualitätsdefizit beeinflussten Qualitätsteilmerkmale werden angegeben.

1. **Qualitätsdefizit Komplexer Code.**

   Bei dem Qualitätsdefizit *Komplexer Code* weist der Code eine hohe algorithmische Komplexität auf [CK94]. Ein solcher Code ist schlechter verstehbar und daher schwerer änderbar [CK94].

   Abweichung zu Qualitätsteilmerkmal: Geringe Streuflusskomplexität.

2. **Qualitätsdefizit Hohe Verwendungskopplung.**

   Unter dem Qualitätsdefizit *Hohe Verwendungskopplung* wird, angelehnt an die Arbeit von Larman [Lar05], eine zu hohe Anzahl von Abhängigkeitsbeziehungen zwischen einer Klasse und davon abhängigen anderen Klassen verstanden; diese Abhängigkeitsstrukturen erschweren Änderungen an den entsprechenden Klassen.

   Abweichung zu Qualitätsteilmerkmal: Lose Kopplung.

3. **Qualitätsdefizit Tiefe Vererbungshierarchie.**

   Das Qualitätsdefizit *Tiefe Vererbungshierarchie* wird durch Klassen verursacht, die in der Vererbungshierarchie sehr tief eingeordnet wurden. Diese Klassen sind schwer zu verstehen, da sie viele Merkmale erben [CK94]. Änderungen an Klassen, die in der Vererbungshierarchie höher eingeordnet wurden, können sich auf alle tiefer eingeordneten Klassen auswirken.

   Abweichung zu Qualitätsteilmerkmal: Lose Kopplung.

4. **Qualitätsdefizit Zyklen zwischen Klassen.**

   Zyklen zwischen Klassen erhöhen die Strukturelle Komplexität der Implementierung und der Architektur [Lil08]. Sie behindern die Weiterentwicklung, da sie fehleranfällig sind und neue Klassen meist an die vorhandenen angebaut werden. Die Benennung des Qualitätsdefizits geht auf die Arbeit von Parnas [Par78] zurück.

   Abweichung zu Qualitätsteilmerkmal: Zyklenfreiheit.

5. **Qualitätsdefizit Zyklen zwischen Komponenten.**

   Zyklen zwischen Komponenten erhöhen die Strukturelle Komplexität der Architektur [Lil08]. Ähnlich wie bei dem Qualitätsdefizit *Zyklen zwischen Klassen* wird die Weiterentwicklung der Komponenten beeinträchtigt, da Änderungen an den Komponenten fehleranfälliger sind. Die Benennung des Qualitätsdefizits geht auf die Arbeit von Parnas [Par78] zurück.

Abweichung zu Qualitätsteilmerkmal: Zyklenfreiheit.

6. **Qualitätsdefizit Geringe Kohäsion von Klassen.**

   Das Qualitätsdefizit *Geringe Kohäsion von Klassen* besteht, wenn eine Klasse für viele Funktionalitäten zuständig ist und damit kein deutlich abgegrenztes Verhalten aufzeigt [Boo94]. Es enthält Methoden, die unterschiedliche funktionale Bereiche aufzeigen [Lar05]. Das Verstehen der Klasse während der Weiterentwicklung wird dadurch erschwert, sodass der Aufwand für die Weiterentwicklung erhöht ist.

   Abweichung zu Qualitätsteilmerkmal: Hohe Kohäsion

7. **Qualitätsdefizit Unausgewogen aufgeteilter Code.**

   Zur Bestimmung der Ausgewogenheit von Entitäten, wie etwa von Klassen oder Schnittstellen, wird untersucht, inwiefern sie ähnlich groß[1] sind [Lil08]. Beispielsweise erschwert *Unausgewogen aufgeteilter Code* die Weiterentwicklung [EL96], da die Klassen schwerer zu verstehen und anzupassen sind. Sie erfordern meist eine weitere Restrukturierung der Implementierung.

   Abweichung zu Qualitätsteilmerkmal: Größenausgewogenheit.

8. **Qualitätsdefizit Unausgewogen zerlegtes Feature.**

   Features sind ausgewogen zerlegt, wenn sie ähnlich groß sind [Lil08]. *Unausgewogen zerlegte Features* erschweren die Weiterentwicklung, da sie schwerer zu verstehen und umzusetzen sind. Sie erfordern meist eine weitere Zerlegung und erhöhen dadurch den Aufwand für die Weiterentwicklung.

   Abweichung zu Qualitätsteilmerkmal: Größenausgewogenheit.

[1] Siehe Definition C.2.4 der Metrik Feature-Größe.

9. **Qualitätsdefizit Geringe Kapselung in Klassen.**

   Im Fall des Qualitätsdefizits *Geringe Kapselung in Klassen* sind auch Teile der Implementierung einer Klasse sichtbar, die nicht für die Zusammenarbeit mit anderen Klassen notwendig sind. Beispielweise wird der Zugriff auf Daten ermöglicht, die nicht öffentlich sein sollten; eine Datenkapselung findet nicht oder nicht vollständig statt [Par72]. Bei Änderungen müssen folglich alle sichtbaren Teile berücksichtigt werden, auch wenn sie für die Kommunikation nicht erforderlich sind.

   Abweichung zu Qualitätsteilmerkmal: Datenkapselung.

10. **Qualitätsdefizit Fehlende Schnittstelle zur Abstraktion.**

    Das Qualitätsdefizit *Fehlende Schnittstelle zur Abstraktion* existiert, wenn es keine Schnittstelle innerhalb einer Komponente gibt, welche die innere Implementierung einer Komponente zum äußeren Kommunikationspartner abstrahiert. Fehlt diese, können Änderungen an der inneren Implementierung meist nicht ohne Auswirkungen nach außen, d.h. auf die äußeren Kommunikationspartner, durchgeführt werden.

    Abweichung zu Qualitätsteilmerkmal: Schnittstellenabstraktion.

11. **Qualitätsdefizit Starke Feature-Verschränkung in Komponenten.**

    Bei *Starker Feature-Verschränkung in Komponenten* existieren Abhängigkeitsstrukturen, welche das Verstehen der Architektur und des Code hinsichtlich der spezifizierten Features verschlechtern. Nach Arndt et al. [AHKP09] steigt in diesem Fall auch die Gefahr von Programmfehlern, da die Struktur und Kontrollflusslogik schwer erschließbar sind.

    Abweichung zu Qualitätsteilmerkmal: Geringe Feature-Verschränkung.

12. **Qualitätsdefizit Starke Feature-Streuung in Komponenten.**

    Im Fall des Qualitätsdefizits *Starke Feature-Streuung in Komponenten* sind Features stark über Komponenten verstreut. Dadurch steigt die Gefahr von Programmfehlern bei der Weiterentwicklung, da die Implementierung eines Fea-

tures im Software-System zerstreut ist und das Verstehen der Gesamtstruktur dadurch behindert wird. Zudem ist die Spezifikation der Lösung anhand von Ablauf-Modellen wie Sequenzdiagrammen unübersichtlich, da viele Komponenten und Klassen berücksichtigt werden müssen.

Abweichung zu Qualitätsteilmerkmal: Geringe Feature-Streuung.

13. **Qualitätsdefizit Fehlende Feature-Spezifikation.**

    Das Qualitätsdefizit *Fehlende Feature-Spezifikation* existiert, wenn es Entitäten der Architektur oder der Implementierung gibt, die nicht durch eine Spezifikation abgesichert sind. Eine Implementierung kann ohne eine Spezifikation nicht auf Korrektheit getestet werden [Lig02].

    Abweichung zu Qualitätsteilmerkmal: Übereinstimmung von Implementierung und Spezifikation

14. **Qualitätsdefizit Fehlende Feature-Implementierung.**

    Ein Software-System ist unvollständig, wenn nicht alle für ein Software-System geplanten Features implementiert wurden [MRW77]. Solche Software-Systeme beeinträchtigen die Weiterentwicklung, da sie nur eine unvollständige Grundlage für Erweiterungen bieten.

    Abweichung zu Qualitätsteilmerkmal: Vollständigkeit der Implementierung.

A.2 Metriken

In Tabelle A.1 sind alle Metriken dargestellt, die in dieser Arbeit verwendet wurden. Neben dem Akronym der Metrik wird auch die Referenz zur Definition der Metrik angegeben.

Tabelle A.1: Übersicht der Komponentenorientierten Metriken.

Akronym	Metriken	Referenz
ADV	Anzahl Demeter-Verstöße	[EL96]
AKS	Anzahl Komponentenschnittstellen	[Lil08]
AOA	Anzahl öffentliche Attribute	[EL96]
CC	Zyklomatische Komplexität	[McC76]
DIT	Depth of Inheritance Tree	[CK94]
FG	Feature-Größe	Abschnitt C.2.4
FIFA	Fan-In-Fan-Out	[CG90]
FSCA	Feature Scattering	Abschnitt C.2.3
FTANG	Feature Tangling	Abschnitt C.2.2
IE	Isolierte Entitäten	Abschnitt C.2.1
IF	Isolierte Features	Abschnitt C.2.1
KIOZ	Komponenten in Zyklen	[Lil08]
KLIZ	Klassen in Zyklen	[Lil08]
KOG	Komponentengröße	[Lil08]
LCOM	Lack of Cohesion in Methods	[CK94]
WMC	Weighted Method Count	[CK94]

Die Metrik *Komponentengröße KOG* ähnelt den Metriken *Anzahl von Klassen* und *Anzahl von Schnittstellen* in Subsystemen, welche in der Arbeit von Lilienthal [Lil08] vorgeschlagen wurden, um die Ausgewogenheit eines Subsystems zu ermitteln. In dieser Arbeit wurden im Unterschied zur Arbeit von Lilienthal auch Datendateien durch Metrik *KOG* berücksichtigt, die zur Konfiguration von Komponenten genutzt werden.

A.3 Reengineering-Aktivitäten

1. **Restrukturieren der Klassen durch Aufspaltung.**

   Klassen können der Arbeit von Baldwin [BC03] folgend durch Restrukturierung in separate Klassen aufgespalten werden. Die Aufspaltung einer Klasse kann durch das Verschieben von Teilen der Klasse erfolgen, wie etwa von Methoden oder von Attributen der Klasse in eine neu erstellte Klasse.

2. **Restrukturieren der Komponenten durch Demotion.**

   Lakos [Lak96] beschreibt Maßnahmen zum Auflösen von Zyklen. Unter dem Namen „Demotion“ sind in der Arbeit von Lakos mehrere Maßnahmen angegeben. Bei zyklischen Abhängigkeitsbeziehungen zwischen zwei Komponenten kann beispielsweise eine der Komponenten aufgespalten werden, sodass die Implementierung, die den Zyklus hervorruft, nur noch von einer Komponente verwendet wird.

3. **Restrukturieren der Klassen durch Refactoring.**

   Zum Restrukturieren von Klassen schlägt Fowler [Fow99] eine Menge von Maßnahmen vor, mit denen Code verändert werden kann. Speziell für die Behebung der Qualitätsdefizite *Hohe Kopplung* und *Hohe Kohäsion* konkretisiert die Arbeit von Demeyer et al. [dBDV04] anwendbare Maßnahmen durch Refactoring.

4. **Restrukturieren der Features durch Aufspaltung.**

   Ein Feature wird aufgespalten, sodass ein neues Feature mit einer eigenen Realisierung entsteht. Diese Reengineering-Aktivität geht auf die Arbeit von Potts [PT93] zurück, der das Aufspalten von Anforderungen vorschlägt, wenn diese unpassend zusammengefasst wurden. Während dieser Maßnahme sollte die Rolle des neuen Features im Software-System auf dessen Eindeutigkeit hin überprüft werden.

5. **Implementieren der Klassen durch Entwurfsmuster.**

   GOF [GHJV95] bieten für gängige Probleme der Implementierung anwendbare Entwurfsmuster an. Das Entwurfsmuster *Fassade* beispielsweise beschreibt die Struktur von Komponenten, in der eine Schnittstelle die Implementierung

abstrahiert und so einen eindeutigen Kommunikationspunkt zwischen Komponenten definiert.

6. **Restrukturieren der Komponenten durch Dekomposition.**

   Die Dekomposition, also das Aufteilen eines Software-Systems in Komponenten, geht auf die Arbeit von Parnas [Par72] zurück. Dabei sollen verschiedene Kriterien, wie etwa die Wahrscheinlichkeit von Änderungen, berücksichtigt werden. Außerdem sollten Komponenten ausgehend von einer Zuständigkeit, wie etwa für ein Feature, aufgeteilt sein und möglichst isoliert von anderen Komponenten ein Feature realisieren.

7. **Restrukturieren der Features durch Zusammenführen.**

   Das Zusammenführen von Features, mit dem Ziel, die Anzahl der in einer Komponente involvierten Features zu reduzieren, führt zu einem neuen Feature. Dessen neue Struktur muss allerdings mit dem Kunden abgestimmt werden. Diese Reengineering-Aktivität geht auf die Arbeit von Potts [PT93] zurück, der das Zusammenführen von Anforderungen vorschlägt, wenn beispielsweise dieselbe Anforderung zweimal vorkommt, mit einem etwas unterschiedlichen Text. Dies kommt häufig vor, wenn verschiedene Autoren an den Anforderungen für das Software-System arbeiten [PT93].

8. **Restrukturieren der Komponenten durch Inversion.**

   Klassen von Komponenten können in einer neuen Komponente zusammengefasst werden. Diese Reengineering-Aktivität wird in der Arbeit von Baldwin [BC03] als *Inversion* bezeichnet. Sie ist dann notwendig, wenn eine unnötige Zerstreuung eines Features in mehrere Komponenten vorliegt, die beispielsweise durch mangelnde Kommunikation im Entwicklungsteam verursacht worden ist.

9. **Löschen der unnötigen Entitäten durch Refactoring.**

   Unnötige Entitäten, wie etwa Features, Klassen oder auch Modelle, werden durch diese Reengineering-Aktivität aus einem Software-System entfernt. Fowler [Fow99] bietet dafür Maßnahmen an, wie etwa „Remove Method“ oder „Remove Parameter“.

10. **Rekonstruieren der Features durch Feature-Analyse.**

    Zur Rekonstruktion eines Features wird der Ansatz der Feature-Analyse nach [Gre07] angewandt. Es werden zunächst Feature-Modelle erzeugt, dann werden die Abhängigkeitsbeziehungen zur Implementierung rekonstruiert und schließlich werden die Abhängigkeitsbeziehungen zwischen den implementierten Features rekonstruiert. Dabei werden unterschiedliche Dokumentationsartefakte erstellt.

11. **Löschen der unnötigen Feature-Spezifikation durch Retraktion.**

    Ein Feature kann entfernt werden, falls es vom Kunden nicht benötigt wird: Features, die anfangs eingeplant, aber dann nicht mehr benötigt werden, müssen entfernt oder auf eine spätere Version verschoben werden. Diese Reengineering-Aktivität geht auf die Arbeit von Potts [PT93] zurück, der das Löschen von Anforderungen vorschlägt, wenn die Anforderungen nicht mehr in einem geplanten Software-System realisiert werden sollen.

12. **Implementieren der Features durch Feature Driven Development.**

    Das Implementieren eines spezifizierten Features kann anhand der Methode Feature-Driven-Development erfolgen, die in der Arbeit von Palmer [PF01] beschrieben wird.

A.4 Reengineering-Rezepte

Eine Übersicht über die in dieser Arbeit erstellten Reengineering-Rezepte für das Qualitätsziel *Weiterentwicklungsfähigkeit* ist in Tabelle A.2 zu sehen. Neben den Namen der Reengineering-Rezepte wird das Qualitätsteilmerkmal angegeben, dem das Reengineering-Rezept zugeordnet ist. Außerdem ist die Referenz zur jeweiligen Definition des Reengineering-Rezepts angegeben.

Tabelle A.2: Reengineering-Rezepte zur Verbesserung der Weiterentwicklungsfähigkeit.

Reengineering-Rezept	Qualitätsteilmerkmal	Tabelle
Ausgewogene Entitäten	Größenausgewogenheit	A.7
Einfache Klassenstruktur	Lose Kopplung	5.8
Feature-Entschränkung	Geringe Feature-Verschränkung	5.10
Feature-Entstreuung	Geringe Feature-Streuung	A.4
Gekapselte Datenstrukturen	Datenkapselung	A.8
Kohäsive Klassen	Hohe Kohäsion	A.5
Nachvollziehbarer Steuerfluss	Komplexer Code	A.6
Spezifizierte Implementierung	Übereinstimmung von Implementierung und Spezifikation	5.6
Unabhängige Implementierung	Schnittstellenabstraktion	A.9
Verbindliche Spezifikation	Vollständigkeit der Implementierung	A.3
Zyklenfreie Strukturen	Zyklenfreiheit	5.9

Im Folgenden wird die Definition derjenigen Reengineering-Rezepte vorgestellt, die nicht schon in Abschnitt 5.4 eingeführt wurden.

Für die Verbesserung des Qualitätsteilmerkmals *Vollständigkeit der Implementierung* kann das Reengineering-Rezept *Verbindliche Spezifikation* eingesetzt werden. Es ist in Tabelle A.3 zu finden.

Tabelle A.3: Reengineering-Rezept Verbindliche Spezifikation.

Reengineering-Rezept Verbindliche Spezifikation	
Ziel	Um das Qualitätsteilmerkmal *Vollständigkeit der Implementierung* zu erreichen, ist jedes spezifizierte Feature verbindlich zu realisieren. Dafür soll das Qualitätsdefizit *Fehlende Feature-Implementierung*erkannt und behoben werden.
Qualitätsteilmerkmal	Vollständigkeit der Implementierung
Metriken	Isolierte Features *IF*
Erkennungsregeln	`'Fehlende Feature-Implementierung' := IF > 0.`

Reengineering-Regeln	`WENN 'Fehlende Feature-Implementierung'`
	`NUTZE 'Implementieren der Features durch Feature Driven' Development'`
	`ODER`
	`NUTZE 'Löschen der unnötigen Feature-Spezifikation durch Retraktion'.`

Das Reengineering-Rezept *Feature-Entstreuung* wird verwendet, um das Qualitätsdefizit *Starke Feature-Streuung in Komponenten* zu erkennen und zu beheben, es ist in Tabelle A.4 zu sehen:

Tabelle A.4: Reengineering-Rezept Feature-Entstreuung.

Reengineering-Rezept Feature-Entstreuung	
Ziel	Um die Zuständigkeitstrennung bezüglich der Feature-Implementierung zu verbessern, soll das Qualitätsdefizit *Starke Feature-Streuung in Komponenten* erkannt und behoben werden. Die Streuung von Teilen der Implementierung eines Features soll durch Restrukturierung reduziert werden.
Qualitätsteilmerkmale	Geringe Feature-Streuung
Metriken	Isolierte Entitäten $IE(A)$, Isolierte Features IF und Feature Scattering $FSCA(A)$

Erkennungsregeln	`'Starke Feature-Streuung in Komponenten' :=`
	`VORB[(IE(A)=0) UND (IF = 0)]`
	`(FSCA(A) > 0).`

Reengineering-Regeln	`WENN 'Starke Feature-Streuung in Komponenten'`
	`NUTZE 'Restrukturieren der Komponenten durch Inversion'`
	`ODER`
	`NUTZE 'Restrukurieren der Features durch Aufspaltung'.`

Zum Erkennen und Beheben von Qualitätsdefiziten, welche das Qualitätsmerkmal *Hohe Kohäsion* beeinträchtigen, wird das Reengineering-Rezept *Kohäsive Klassen* verwendet. Es ist in Tabelle A.5 dargestellt.

Tabelle A.5: Reengineering-Rezept Kohäsive Klassen.

Reengineering-Rezept Kohäsive Klassen	
Ziel	Um das Qualitätsteilmerkmal *Hohe Kohäsion zu verbessern*, soll das Qualitätsdefizit *Geringe Kohäsion von Klassen* erkannt und behoben werden. Klassen, die eine geringe Kohäsion haben, sollen durch Restrukturierungen verbessert werden.
Qualitätsteil-merkmal	Hohe Kohäsion
Metriken	Lack of Cohesion ***LCOM***
Erkennungs-regeln	`'Geringe Kohäsion von Klassen' := LCOM > 83.`
Reengineering-Regeln	`WENN 'Geringe Kohäsion von Klassen'` `NUTZE 'Restrukturieren der Klassen durch Refactoring'.`

Der in der Erkennungsregel verwendete Maximalwert von 83 für die Metrik ***LCOM*** wurde in der Arbeit von Chidamber et al. durch eine empirische Fallstudie ermittelt [CDK98] und in dieses Reengineering-Rezept übernommen.

Das Reengineering-Rezept *Nachvollziehbarer Steuerfluss* ist in Tabelle A.6 zu finden. Es wird zur Verbesserung des Qualitätsteilmerkmals *Geringere Streuflusskomplexität* eingesetzt.

Tabelle A.6: Reengineering-Rezept Nachvollziehbarer Steuerfluss.

Reengineering-Rezept Nachvollziehbarer Steuerfluss	
Ziel	Um das Qualitätsteilmerkmal *Geringere Streuflusskomplexität* zu verbessern, soll das Qualitätsdefizit *Komplexer Code* erkannt und behoben werden. Durch Restrukturierung soll eine einfachere Struktur im Code erreicht werden, sodass der Steuerfluss nachvollziehbar ist.
Qualitätsteilmerkmal	Geringere Streuflusskomplexität
Metriken	Zyklomatische Komplexität ***CC***
Erkennungsregeln	`'Komplexer Code' := CC > 10.`
Reengineering-Regeln	`WENN 'Komplexer Code'` `NUTZE 'Restrukturieren der Klassen durch Refactoring'.`

Der in der Erkennungsregel eingesetzte Maximalwert von 10 für die Metrik ***CC*** wurde in den Arbeiten [McC76, BR04] ermittelt und in dieses Reengineering-Rezept übernommen.

Zur Verbesserung des Qualitätsteilmerkmals *Größenausgewogenheit* kann das Reengineering-Rezept *Ausgewogene Entitäten* genutzt werden, dessen Definition in Tabelle A.7 zu sehen ist.

Tabelle A.7: Reengineering-Rezept Ausgewogene Entitäten.

Reengineering-Rezept Ausgewogene Entitäten	
Ziel	Um das Qualitätsteilmerkmal *Größenausgewogenheit* zu erreichen, sollen die Qualitätsdefizite *Unausgewogen aufgeteilter Code* oder *Unausgewogen zerlegte Features* erkannt und durch Restrukturierungen behoben werden.
Qualitätsteil-merkmal	Größenausgewogenheit
Metriken	Komponentengröße ***KOG***, Weighted Method Count ***WMC***, Feature-Größe ***FG***
Erkennungs-regeln	`'Unausgewogen aufgeteilter Code' := KOG > 66 ODER WMC>22.` `'Unausgewogen zerlegte Features' := FG > 20.`

Reengineering-Regeln	`WENN 'Unausgewogene aufgeteilter Code'`
	`NUTZE 'Restrukturieren der Klassen durch Refactoring'.`
	
	`WENN 'Unausgewogen zerlegtes Feature'`
	`NUTZE 'Restrukturieren der Features durch Aufspaltung'`
	`UND`
	`NUTZE 'Restrukturieren der Komponente durch Dekomposition'`

In der Fallstudie wurde durch das Entwicklungsteam das Feature *Fehlerverwaltung* (f_{22}) als zu groß eingeschätzt; insbesondere wurde das Feature *Datenzugriffkapselung* (f_{21}) als Ausreißer betrachtet (siehe Abbildung 7.3). Auf Grundlage dieser Beobachtung wurde vom Entwicklungsteam ein Maximalwert von 20 für die Metrik ***FG*** festgelegt und in die Erkennungsregel des Reengineering-Rezepts übernommen.

Nach den durchgeführten Reengineering-Aktivitäten wurde für die Komponente `Da-`

`tenzugriff` (als größte Komponente im Software-System) der Wert von 66 mit der Metrik ***KOG*** gemessen. Dieser Wert wurde als Maximalwert vom Entwicklungsteam bestimmt und ebenfalls in die Erkennungsregel des Reengineering-Rezepts aufgenommen.

In der Arbeit von Chidamber et al. wurde der Maximalwert von 22 für die Metrik ***WMC*** in der zweiten Erkennungsregel durch eine empirische Fallstudie ermittelt [CDK98] und in dieses Reengineering-Rezept übernommen.

In Tabelle A.8 ist das Reengineering-Rezept *Gekapselte Datenstrukturen* zu sehen. Es kann verwendet werden, um das Qualitätsteilmerkmal *Datenkapselung* zu verbessern.

Tabelle A.8: Reengineering-Rezept Gekapselte Datenstrukturen.

Reengineering-Rezept Gekapselte Datenstrukturen	
Ziel	Um das Qualitätsteilmerkmal *Datenkapselung* zu verbessern, soll das Qualitätsdefizit *Geringe Kapselung in Klassen* erkannt und behoben werden. Damit sollen nur die Datenstrukturen sichtbar sein, die zur Kommunikation mit anderen Klassen benötigt werden.
Qualitätsteilmerkmal	Datenkapselung
Metriken	Anzahl öffentliche Attribute ***AOA***, Anzahl Demeter-Verstöße ***ADV***
Erkennungsregeln	`'Geringe Kapselung in Klassen' := AOA > 0 ODER ADV > 0.`
Reengineering-Regeln	`WENN 'Geringe Kapselung in Klassen'` `NUTZE 'Restrukturieren der Klassen durch Refactoring'.`

Der Schwellwert von 0 für die Metriken ***AOA*** und ***ADV*** wurde der Arbeit von [EL96] folgend in die Erkennungsregel aufgenommen.

Das Reengineering-Rezept *Unabhängige Implementierung* ist durch Tabelle A.9 dargestellt. Es kann für die Verbesserung des Qualitätsteilmerkmals *Schnittstellenabstraktion* verwendet werden.

Tabelle A.9: Reengineering-Rezept Unabhängige Implementierung.

Reengineering-Rezept Unabhängige Implementierung	
Ziel	Um das Qualitätsteilmerkmal *Schnittstellenabstraktion* zu verbessern, soll das Qualitätsdefizit *Fehlende Schnittstellen zur Abstraktion in Komponenten* erkannt und behoben werden. Jede Komponente soll mindestens eine Schnittstelle zur Kommunikation zwischen Komponenten anbieten [Lil08].
Qualitätsteil-merkmal	Schnittstellenabstraktion
Metriken	Anzahl von Komponentenschnittstellen *AKS*
Erkennungs-regeln	`’Fehlende Schnittstelle zur Abstraktion’ :=` *AKS* `= 0.`
Reengineering-Regeln	`WENN ’Fehlende Schnittstelle zur Abstraktion’` `NUTZE ’Implementieren der Klassen durch Entwurfsmuster’.`

Der Arbeit von Lilienthal [Lil08] folgend wird der Schwellwert für *AKS* in der Erkennungsregel mit 0 festgelegt.

Anhang B

Abhängigkeitsbeziehungen im Werkzeug TraceTool

Das Abhängigkeitsmodell in Abbildung B.1 liefert die Voraussetzung für den Aufbau von Abhängigkeitsdaten im Werkzeug TraceTool (siehe Kapitel 6), da es beschreibt, wie Abhängigkeitsbeziehungen grundsätzlich aufgebaut sind und welche Regeln zur Prüfung der Integrität gelten. In den folgenden Abschnitten B.1 und B.2 werden die Elemente des Abhängigkeitsmodells beschrieben.

B.1 Struktur von Abhängigkeitsbeziehungen

Grundlage für die gewählte Struktur des Abhängigkeitsmodells sind die bisherigen Ansätze von Abhängigkeitsmodellen [Lin94, RJ01, Pin04]. Im Folgenden wird angelehnt an die Modellierungssprache UML [RJB04] die grundsätzliche Struktur des Abhängigkeitsmodells und ihre Elemente beschrieben. Abbildung B.1 zeigt die Modellelemente des Abhängigkeitsmodells.

Das Modellelement Beziehung definiert eine Beziehung zwischen zwei Entitäten. Eine Beziehung hat immer genau eine Richtung, ausgehend vom Startelement zum Endelement. Aus diesem Grund besitzt dieses Modellelement zwei Attribute: eine Entität als Startelement einer Beziehung und eine weitere Entität als Endelement. Das Modell berücksichtigt nur Beziehungen mit einer festgelegten Richtung, um genauere Prüfungen durchführen zu können.

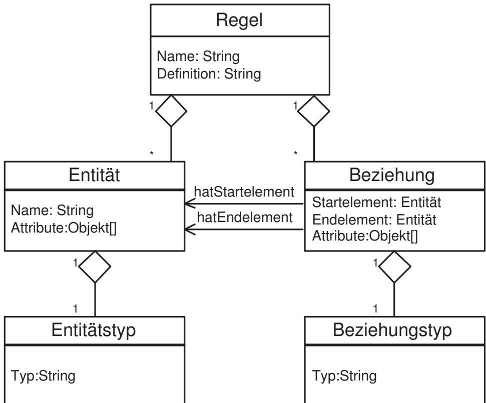


Abbildung B.1: Grundstruktur von Elementen des Abhängigkeitsmodells.

Eine Entität kann neben dem Namen weitere Attribute haben, die für die spätere Auswertung wichtig sind. Beispielsweise kann eine Versionsangabe als zusätzliches Attribut benötigt werden. Diese Attribute sind optional und es können je nach Anwendungsfall benötigte Attribute hinzukommen und innerhalb von Abhängigkeitsdaten gespeichert werden.

Die Art der Beziehung und der Entität wird durch deren Typ festgelegt, dies wird durch das Modellelement Beziehungstyp bei Beziehungen und durch das Modellelement Entitätstyp bei Entitäten ermöglicht. Auch bei einer Beziehung ist die Möglichkeit gegeben, weitere Attribute, beispielsweise den Namen des Erstellers, hinzuzufügen.

Anhand des hier vorgestellten Abhängigkeitsmodells ist es möglich, verschiedene Beziehungen zu unterscheiden:

1. Beziehungen zwischen Entitäten des gleichen Entitätstyps (z.B. eine Komponente hängt von einer anderen Komponente ab)

2. Beziehungen zwischen Entitäten unterschiedlicher Entitätstypen (z.B. ein Feature wurde durch Klassen umgesetzt)

Die effektive Nutzung und Bewertung von Beziehungen zwischen Entitäten ist erst durch den Einsatz von Regeln möglich. Dazu werden im Abhängigkeitsmodell Regeln definiert; das passende Modellelement Regel ist in Abbildung B.1 dargestellt. Neben

dem Attribut Name, wird mit dem Attribut Definition die eigentliche Regel festgelegt: Eine Regel kann textuell beschrieben sein, durch logische Ausdrücke definiert werden oder anhand von Sprachen wie der Object Constraint Language [OCL06] umgesetzt sein. Dass unvollständige Beziehungen ungültig sind und nicht innerhalb eines Repositories abgelegt werden dürfen, ist ein Beispiel für eine solche Regel.

Beschreibung der Elemente des Abhängigkeitsmodells

Im Folgenden werden alle für diese Arbeit relevanten Entitäten, Beziehungen und Regeln eingeführt und dargestellt. Ob es sich bei den eingeführten Elementen um eine Entität, Beziehung oder eine Regel handelt, wird zusätzlich durch ein entsprechendes Symbol jeweils oben rechts im Kasten gekennzeichnet: Ein Rechteck ▭ symbolisiert eine Entität, ein Pfeil $\longrightarrow$ symbolisiert eine Beziehung und das Sechseck ⬭ eine Regel.

Um Abhängigkeiten zwischen Entitäten im Software-System verfolgen und auswerten zu können, werden diese im Abhängigkeitsmodell durch den Einsatz von Beziehungen festgehalten. Im Folgenden werden die Mengen und Symbole für Entitäten, Beziehungen und Regeln definiert.

Definition 2.1: Entitäten ▭

Als Entitäten sei die Menge der Entitäten $E := \{e_1,\ ...,\ e_n\}$ definiert.

Eine Komponente ist ein Beispiel für eine Entität. In den folgenden Abschnitten werden verschiedene Untermengen betrachtet und anhand ihrer Eigenschaften definiert.

Definition 2.2: Beziehungen $\longrightarrow$

Die Menge der Beziehungen T sei definiert als: $T := \{t : t \in E \times E\}$. Jeder Beziehung wird dabei durch eine Typfunktion $TF:\ T \rightarrow TYP$ ihr jeweiliger Beziehungstyp zugeordnet. Dafür sei eine Menge von Beziehungstypen $TYP := \{b,\ r,\ t\}$ gegeben, die für 'benutzt', 'realisiert-durch' und 'teil-von' stehen. Eine Beziehung $t \in T$ existiert somit zwischen zwei Entitäten $e_i \stackrel{typ}{\rightarrow} e_j$ mit $i, j \in [1,n]$ und wird durch ein Pfeil-Symbol $\stackrel{typ}{\rightarrow}$ dargestellt. Der Beziehungstyp wird somit als Bezeichnung über dem Pfeil notiert.

Beispielsweise kann eine Beziehung $a_1 \stackrel{b}{\rightarrow} a_2$ zwischen zwei Komponenten $a_1, a_2 \in A \subseteq E$, also Entitäten vom Typ Komponente, existieren. Dies bedeutet, zwischen a_1 und a_2 herrscht die Beziehung a_1 *benutzt* a_2. In dieser Arbeit werden (siehe Def. 2.2)

ausschließlich die drei Beziehungstypen $\xrightarrow{b}$ (*benutzt*), $\xrightarrow{r}$ (*realisiert-durch*) und $\xrightarrow{t}$ (*teil-von*) betrachtet.

Des Weiteren werden folgende syntaktische Regeln definiert, die übergreifend für alle Beziehungen gültig sind.

Bemerkung 2.3: Vollständige Beziehungen

Beziehungen sind nur dann vollständig und auswertbar, wenn sie sowohl ein Startelement als auch ein Endelement besitzen.

Dies ergibt sich zwar direkt aus Def. 2.2 für die Entitäten (kein leeres Element enthalten), dennoch soll durch diese Regel darauf hingewiesen werden, dass Beziehungen vollständig sein müssen. Diese Regel wird bei softwarebasierten Umsetzungen leicht durch Null-Zeiger oder Null-Werte verletzt.

Im Abhängigkeitsmodell sind solche Beziehungen ungültig, die auf sich selbst referenzieren, da dieser Fall in der Praxis keinen Nutzen bringt.

Definition 2.4: Ungültige Beziehungen

Beziehungen mit identischem Start- und Endelement werden als ungültige Beziehungen bezeichnet. Als ungültige Beziehungen sei die Menge aller Beziehungen U definiert, für die gilt:

$$U := \{(e,e) \in T\}.$$

Beim Erstellen oder Ändern von Beziehungen innerhalb der Abhängigkeitsdaten muss sichergestellt werden, dass die Syntax der Beziehung eingehalten wird. Die Syntax ergibt sich aus der Richtung, den Entitäten und dem Beziehungstyp. Eine Übersicht über alle in dieser Arbeit definierten Beziehungen ist in Tabelle B.1 am Anfang des nächsten Abschnitts dargestellt.

Die Verwendung von Beziehungen, beispielsweise die Benutzt-Beziehung (siehe Abschnitt B.2.1), kann zur Konstruktion von zyklischen Abhängigkeiten führen. Im Allgemeinen wird diese Struktur auch kurz Zyklus genannt; ein Zyklus entspricht einem gerichteten zyklischen Graphen [Par78, Mar96]. Ein Beispiel für einen Zyklus ist in Abbildung B.2 dargestellt. Ein Zyklus zwischen Entitäten, beispielsweise Klassen oder Schnittstellen, erhöht die Komplexität und verschlechtert die Änderbarkeit der Implementierung [Mar96], [Sta08b]. Ein Zyklus zwischen Komponenten weist auf eine

schlechte Architektur hin [Mar96].

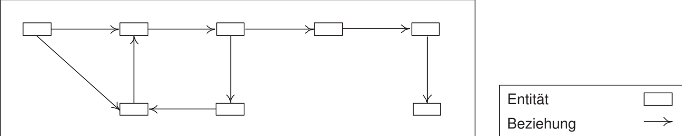


Abbildung B.2: Zyklische Beziehungen zwischen Entitäten.

In Kapitel 5.1 wurde hinsichtlich der Qualitätsbewertung der *Weiterentwicklungsfähigkeit* ausführlich auf dieses Defizit eingegangen.

Definition 2.5: Zyklische Beziehungen

Beziehungen enthalten einen Zyklus genau dann, wenn eine Entität gleichzeitig Start- und Endelement in einer Kette von Beziehungen des gleichen Typs ist:

$$\exists e_0 \in E,\ \ldots,\ e_n \in E, n \in \mathbb{N}_0 : e_0 \overset{typ}{\rightarrow} e_1 \overset{typ}{\rightarrow} e_2,\ \ldots,\ \overset{typ}{\rightarrow} e_n \overset{typ}{\rightarrow} e_0.$$

B.2 Beziehungen im Abhängigkeitsmodell

Beziehungen im Abhängigkeitsmodell repräsentieren Abhängigkeiten zwischen Entitäten, die in verschiedenen Phasen der Entwicklung erstellt werden. In diesem Abschnitt werden Entitäten und Abhängigkeiten betrachtet, welche zur Bewertung des Qualitätsziels *Weiterentwicklungsfähigkeit* verwendet werden können und für die Qualitätsbewertung durch Metriken relevant sind. Es werden Features betrachtet, welche die fachliche Sicht des Software-Systems darstellen und weitere Entitäten und Beziehungen aus der Architektur, dem Design oder der Implementierung. Sie werden innerhalb von Entwicklungsmethoden beispielsweise Object-Oriented Software Engineering (OOSE) [Jac93], Rational Unified Prozess (RUP) [Kru03] oder Object-Oriented Process, Environment, and Notation (OPEN) [Pra98] verwendet, in denen objektorientierte Modellierungsprachen wie Unified Modeling Language (UML) oder auch OPEN Modeling Language (OML) Anwendung finden.

In Tabelle B.1 sind alle Beziehungen zu sehen, die in dieser Arbeit für die Bewertung der *Weiterentwicklungsfähigkeit* betrachtet werden. Die Übersicht zeigt alle in den

folgenden Abschnitten definierten Beziehungen mit den jeweils möglichen Verknüpfungen von Startelementen zu Endelementen. Zudem ist angegeben, wo die Definition zur Beziehung zu finden ist: beispielsweise ist die Beziehung vom Typ *benutzt* zwischen zwei Komponenten $a \xrightarrow{b} a'$ durch die Def. 2.7 gegeben.

Tabelle B.1: Übersicht über die Beziehungen im Abhängigkeitsmodell.

Beziehung	zwischen	und	Referenz
benutzt $\xrightarrow{b}$	Komponente $a \in A$	Komponente $a' \in A$	Def. 2.7
	Klasse $k \in K$	Schnittstelle $s \in S$	Def. 2.11
		Klasse $k' \in K$	Def. 2.10
		Datendatei $d \in D$	Def. 2.13
teil-von $\xrightarrow{t}$	Schnittstelle $s \in S$ Klasse $k \in K$ Datendatei $d \in D$	Komponente $a \in A$	Def. 2.14
realisiert-durch $\xrightarrow{r}$	Feature $f \in F$	Komponente $a \in A$ Schnittstelle $s \in S$ Klasse $k \in K$ Datendatei $d \in D$	Def. 2.16

B.2.1 Benutzt-Beziehung

Ein UML Komponentenmodell nach Jeckle et al. [JRH+03] wird benutzt, um Eigenschaften der Architektur und des Designs eines Systems darzustellen. UML Komponentenmodelle beschreiben dazu existierende oder zu entwickelnde Entitäten und ihre Eigenschaften. Durch die gezielte Reduktion auf die wesentlichen Informationen wird das Verstehen von Beziehungen erleichtert. Abbildung B.3 zeigt links ein UML Komponentenmodell [JRH+03], welches Beziehungen zwischen drei Komponenten beschreibt. Rechts davon ist das Abhängigkeitsmodell abgebildet, in dem die Entitäten und Beziehungen zu finden sind, die innerhalb eines Repositories als Abhängigkeitsdaten erstellt werden.

Beziehungen, die durch das UML Komponentenmodell dargestellt werden können, finden sich auch im Abhängigkeitsmodell wieder. Zum Beispiel ist eine wichtige Beziehung die Benutzt-Beziehung, welche durch Parnas eingeführt wurde [Par78] und

durch Rumbaugh, Jacobson und Booch erweitert wurde [RJB04]. Sie sagt aus, dass eine Komponente eine andere benutzt und so von ihr abhängig ist.

In UML kann man diese Abhängigkeitsbeziehung mit der Bezeichnung *use* deutlich machen, wie das Beispiel links in Abbildung B.3 zeigt. Je nachdem, welche Entitäten über die Benutzt-Beziehung verknüpft sind, haben sie eine unterschiedliche Bedeutung bei der Auswertung oder Prüfung von Abhängigkeitsdaten durch Regeln, deswegen wird der Entitätstyp im Abhängigkeitsmodell unterschieden.

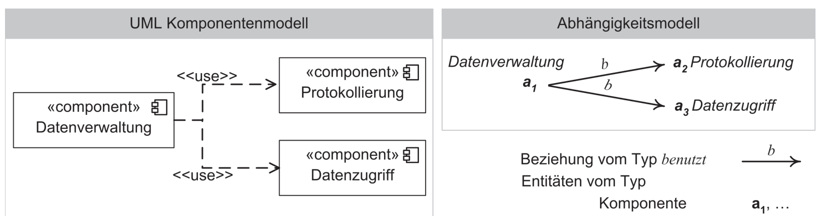


Abbildung B.3: Repräsentation von Beziehungen zwischen Komponenten im Abhängigkeitsmodell.

Im dargestellten Beispiel in Abbildung B.3 rechts ergeben sich im Abhängigkeitsmodell zwei Benutzt-Beziehungen zwischen Komponenten: die Komponente ***Datenverwaltung*** ist Startelement, die Komponente ***Datenzugriff*** Endelement und der Beziehungstyp *benutzt* dient als ein Typ der Beziehung.

Definition 2.6: Komponente

Komponenten sind modulare Teile eines Software-Systems, welche die Implementierung der Funktionalität hinter externen Schnittstellen verbergen. Sie werden notiert als:

$$a_1,\ a_2,\ \ldots,\ a_n \in A \subseteq E, n \in \mathbb{N}.$$

In der Definition der Entität Komponente, wurde der Buchstabe A im Sinne von Architekturkomponenten für die Menge der Komponenten gewählt, da K für die Menge der Klassen bereits vergeben ist.

Die Benutzt-Beziehung zwischen Komponenten, wie auch alle anderen im Folgenden notierten Beziehungen, bauen auf der allgemeinen Def. 2.2 von Beziehungen $t \in T$ auf.

Definition 2.7: Benutzt-Beziehung zwischen Komponenten $\longrightarrow$

Das Symbol $a \xrightarrow{b} a'$ steht für eine Benutzt-Beziehung zwischen Komponenten und ist definiert als

$$a \xrightarrow{b} a' :\Leftrightarrow a, a' \in A,\ t = (a, a') \in T.$$

Der Beziehungstyp von t ist *benutzt.*

Die Abbildung B.4 zeigt links ein UML Komponentenmodell, in dem auch die Schnittstellen der Komponenten explizit dargestellt sind. Rechts davon ist wieder das entsprechende Abhängigkeitsmodell zu sehen. Aus Sicht der Softwarearchitektur können Komponenten andere Komponenten benutzen (Benutzt-Beziehung), um Daten durch einen „Kanal“ auszutauschen. Dazu wird eine Schnittstelle durch eine Komponente bereitgestellt, diese Schnittstelle ist Teil von genau einer Komponente [RJB04]. Die dazugehörige Beziehung vom Typ *teil-von* ($s \xrightarrow{t} a$) wird durch die Def. 2.14 im folgenden Abschnitt B.2.2 definiert.

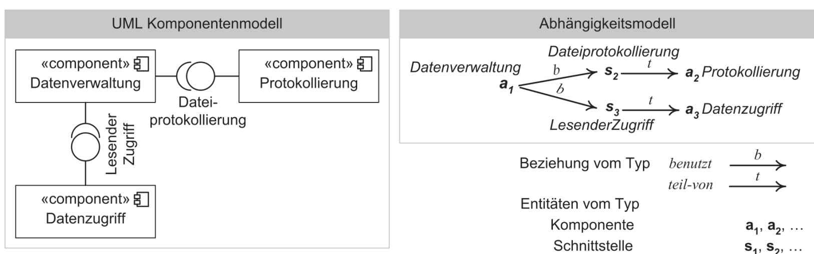


Abbildung B.4: Repräsentation von Beziehungen zwischen Komponenten und Schnittstellen im Abhängigkeitsmodell.

Im Anwendungsbeispiel in Abbildung B.4 ist die folgende Situation dargestellt: Die Komponente *Datenverwaltung* kann nur über die Schnittstelle *Dateiprotokollierung* mit der Komponente *Protokollierung* Daten austauschen.

Definition 2.8: Schnittstelle

Schnittstellen S stellen eine Menge von Operationen zusammen. Sie sind Teil einer Komponente $a \in A$, in der die Funktionen implementiert sind und werden notiert als:

$$s_1,\ s_2,\ \ldots,\ s_n \in S \subseteq E, n \in \mathbb{N}.$$

Abbildung B.5[1] links zeigt ein UML Klassendiagramm, um Entitäten wie Klassen oder Schnittstellen und ihre Abhängigkeiten darzustellen.

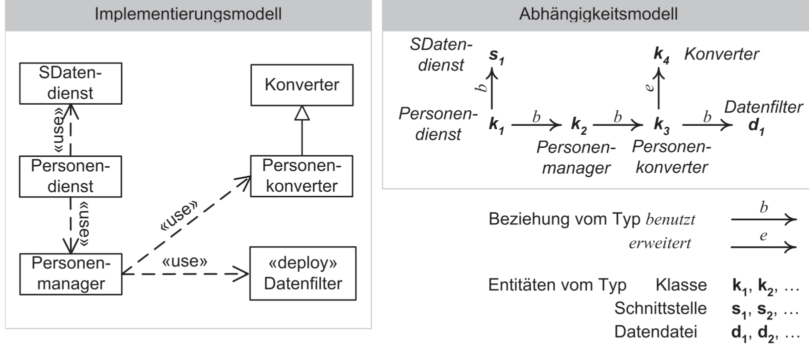


Abbildung B.5: Repräsentation von Beziehungen zwischen Klassen, Schnittstellen und Datendateien im Abhängigkeitsmodell.

Für die Entitäten Komponente, Schnittstelle und Klasse spielen die Begriffe *Abstraktion*, *Kapselung* und *Geheimnisprinzip* (engl. Information Hiding) eine wichtige Rolle [Par72, Jac93, Sch94]:

Unter Abstraktion versteht man das Konzentrieren auf das Wesentliche, dabei werden nur die Methoden angeboten, die im Kontext der Komponente oder einer Klasse interessant sind. In der Objektorientierung werden Daten und Methoden als Einheit verstanden, im Sinne einer Kapsel. Diese „Einkapselung" ist der grundlegende Unterschied zur funktions-orientierten Programmierung, da dort zwischen Daten und Methoden unterschieden wird.

[1] Die im Abhängigkeitsmodell dargestellten Beziehungen werden ab Def. 2.10ff. definiert.

Komponenten oder Klassen tauschen Daten über eine zuvor festgelegte Schnittstelle aus, dabei sollen nicht alle Daten und auch nicht die Implementierung von Methoden für den jeweiligen Kommunikationspartner sichtbar sein (Geheimnisprinzip), sondern nur die öffentliche Aufrufschnittstelle. Änderungen an der inneren Implementierung von Komponenten oder Klassen haben keine Auswirkungen auf andere, solange die Schnittstellen stabil bleiben.

Wie bereits erwähnt, werden innerhalb des Abhängigkeitsmodells z.B. rechts in Abbildung B.5 Informationen durch Beziehungen dargestellt, die dann in Form von Abhängigkeitsdaten in einem Repository gespeichert werden.

Definition 2.9: Klasse

Klassen innerhalb der Objektorientierung fassen Attribute und Methoden zusammen [RJB04]. Aus Sicht des Entwurfs sind Klassen eine Abstraktion des Problembereichs und ein Zerlegungsprodukt als Teil einer Komponente, sie werden notiert als:

$$k_1,\ k_2,\ \ldots,\ k_n \in K \subseteq E, n \in \mathbb{N}.$$

Im Allgemeinen wird eine Klasse durch Attribute strukturiert und Methoden bestimmen ihr Verhalten.

Im Anwendungsbeispiel in Abbildung B.5 benutzt die Klasse *Personenmanager* die Klasse *Personenkonverter*, um eine Adresse in ein speicherbares Format zu konvertieren, bevor diese dann weiterverarbeitet werden können. Die Benutzt-Beziehung zwischen Komponenten wurde bereits eingeführt, zwischen Klassen wird sie folgendermaßen notiert:

Eine Klasse kann auch von einer anderen Klasse abhängig sein, wenn beispielsweise Methoden aus der anderen Klasse benutzt werden. Dieser Typ von Abhängigkeitsbeziehung wird auch durch die Benutzt-Beziehung notiert:

Definition 2.10: Benutzt-Beziehung zwischen Klassen $\longrightarrow$

Das Symbol $k \xrightarrow{b} k'$ steht für die Benutzt-Beziehung zwischen zwei Klassen und ist definiert als:

$$k \xrightarrow{b} k' :\Leftrightarrow k, k' \in K,\ t = (k, k') \in T.$$

Der Beziehungstyp von t ist *benutzt*.

Eine Klasse kann auch von einer Schnittstelle abhängig sein, wenn beispielsweise

Attribute wie Konstanten aus der Schnittstelle benutzt werden. Dieser Typ von Abhängigkeitsbeziehung wird auch durch die Benutzt-Beziehung notiert:

Definition 2.11: Benutzt-Beziehung zwischen Klasse und Schnittstelle $\longrightarrow$

Das Symbol $k \xrightarrow{b} s$ zwischen einer Klasse und einer Schnittstelle steht für die Beziehung *benutzt* und ist definiert als:

$$k \xrightarrow{b} s :\Leftrightarrow k \in K, s \in S,\ t = (k,s) \in T.$$

Der Beziehungstyp von t ist *benutzt*.

Im Allgemeinen kann ein Datenspeicher in Software-Systemen unterschiedlich implementiert sein, wie etwa in Datendateien oder auch in einer Datenbank. In dieser Arbeit werden Datenspeicher in Form von Datendateien betrachtet. Sie enthalten Informationen, welche für den Build (Erstellungsvorgang einer Softwareversion) oder das Deployment (Verteilung oder Installation auf einem Server) relevant sind, indem sie beispielsweise das Aktivieren oder Deaktivieren von Features vor der Installation ermöglichen.

Definition 2.12: Datendatei

Datendateien D sind Teil einer Komponente und werden notiert als:

$$d_1,\ d_2,\ \ldots,\ d_n \in D \subseteq E, n \in \mathbb{N}.$$

Im Anwendungsbeispiel in Abbildung B.5 verwendet die Klasse *Personenmanager* die Datendatei *Datenfilter*, um die dort gespeicherten Daten auszulesen, die bestimmen, ob bestimmte Attribute einer Person bei Suchanfragen unterdrückt werden sollen. Neben der Benutzt-Beziehung zwischen Komponenten oder Klassen, wird die Benutzt-Beziehung zwischen Klassen und Datendateien wie folgt notiert:

Definition 2.13: Benutzt-Beziehung zwischen Klasse und Datendatei $\longrightarrow$

Das Symbol $k \xrightarrow{b} d$ steht für die Benutzt-Beziehung zwischen einer Klasse und einer Datendatei und wird notiert als:

$$k \xrightarrow{b} d :\Leftrightarrow k \in K, d \in D,\ t = (k,d) \in T.$$

Der Beziehungstyp von t ist *benutzt*.

B.2.2 Teil-von-Beziehung

Eine Komponente enthält in der Regel verschiedenste Entitäten wie Klassen, Schnittstellen oder Datendateien. Damit können diese Entitäten zu genau einer Komponente zugeordnet werden, da sie Teil genau einer Komponente sind:

Definition 2.14: Teil-von-Beziehung von Komponenten $\longrightarrow$

Die Teilmenge X der Entitäten E enthält Klassen K, Schnittstellen S und Datendateien D. Das Symbol $x \xrightarrow{t} a$ zwischen Komponenten A und Entitäten $x \in X$ steht für die Teil-von-Beziehung und beschreibt, dass eine Entität aus der Menge X Teil von der Komponente a ist.

$$x \xrightarrow{t} a :\Leftrightarrow a \in A, x \in X,\ s = (x,a) \in T.$$

Der Beziehungstyp von s ist *teil-von*.

Neben den bisher definierten Entitäten K, S, $D \subseteq X \subseteq E$ können weitere Entitäten berücksichtigt werden. Beispielsweise ist der Entitätstyp Paket, ein Mittel zur Gruppierung von Entitäten innerhalb einer Komponente. Ein Paket ermöglicht die funktionale Gliederungen von Entitäten. Dieser Entitätstyp könnte auch innerhalb des Abhängigkeitsmodells integriert werden. Da sie aber in dieser Arbeit hinsichtlich der Bewertung der Qualität eine untergeordnete Rolle spielt, wird sie im Abhängigkeitsmodell nicht weiter betrachtet.

B.2.3 Realisiert-durch-Beziehung

Während der Software-Entwicklung stehen die Anforderungen des Kunden im Mittelpunkt. In einem featureorientierten Entwicklungsprozess ([PF01]) bilden Features, die aus den Anforderungen abgeleitet werden, den Kern der Entwicklung. Ein wichtiges Kriterium dabei ist die Eindeutigkeit von Features über die verschiedenen Entwicklungsphasen und Produktversionen hinweg. Dazu ist es notwendig, ausgehend vom Feature alle Entitäten ermitteln zu können, welche für die Realisierung des Features notwendig sind.

Definition 2.15: Features

Ein Feature entstammt der Menge aller Features, die aus den Anforderungen eines Software-Systems abgeleitet worden sind:

$$f \in F = \{f_1,\ f_2,\ \ldots,\ f_n, n \in \mathbb{N}\}, F \subseteq E.$$

In Abbildung B.6 sind zwei Features der Fallstudie dargestellt. Features im Feature-Modell sind aus Sicht des Abhängigkeitsmodells Entitäten mit jeweils verschiedenen verfolgbaren Attributen, die Auswirkungen auf das Design und die Implementierung der Features haben. In dieser Arbeit sind dies der Name des Features z.B. ***Personenverwaltung*** und der Name des betrachteten Software-Systems.

Die Realisierung von Features wird in der Regel durch verschiedene Modelle dokumentiert, da verschiedene Entitäten beteiligt sind. Im Abhängigkeitsmodell werden folglich auch unterschiedliche Entitätstypen als Endelement einer Realisiert-durch-Beziehung berücksichtigt.

Definition 2.16: Realisiert-durch-Beziehung $\longrightarrow$

Die Teilmenge X der Entitäten E besteht aus Komponenten A, Schnittstellen S, Klassen K oder Datendateien D. Das Symbol $f \xrightarrow{r} x$ symbolisiert eine Beziehung vom Typ *realisiert-durch* zwischen einem Feature f und einer Entität $x \in X$; die Entität ist für die Realisierung des Features notwendig:

$$f \xrightarrow{r} x :\Leftrightarrow f \in F,\ x \in X,\ t = (f,x) \in T.$$

Der Beziehungstyp von t ist *realisiert-durch*.

Dieser Beziehungstyp wird als Traceability-Link bezeichnet [SR02] und repräsentiert im Vergleich zu den bisherigen Beziehungen eine Entwurfsentscheidung in Richtung der Problemlösung und ihrer Realisierung; er ist sozusagen Träger einer Entwurfsentscheidung. Nach Boehm sollte jede Entwurfsentscheidung auf die ursprüngliche Anforderungen zurückführbar sein [Boe74].

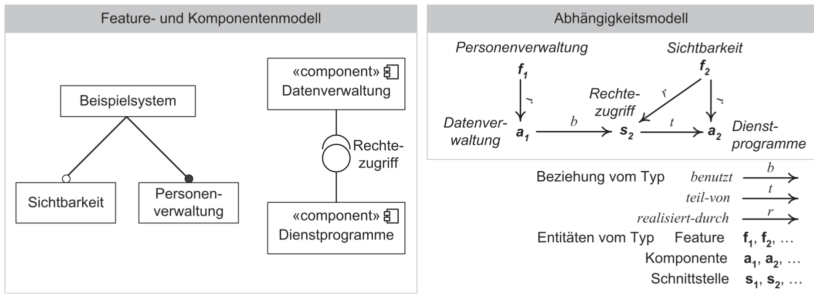


Abbildung B.6: Repräsentation von Beziehungen und Traceability-Links zwischen Features, Komponenten und Schnittstellen im Abhängigkeitsmodell.

Das Abhängigkeitsmodell kann diese Traceability-Links darstellen, wie rechts in Abbildung B.6 zu sehen ist. Beispielsweise sagt der Traceability-Link vom Typ *realisiert-durch* von Feature *Personenverwaltung* zu Komponente *Datenverwaltung* aus, dass das Feature *Personenverwaltung* durch die Komponente *Datenverwaltung* umgesetzt ist. Das rechts in Abbildung B.6 gezeigte Abhängigkeitsmodell zeigt alle Beziehungen, die als Abhängigkeitsdaten für die Qualitätsbewertung durch Metriken in dieser Methode erstellt werden.

Im Abhängigkeitsmodell wird auch deutlich, dass die Schnittstelle *Rechtezugriff* und die Komponente *Dienstprogramme* eng zusammenhängen, da die Schnittstelle ein Teil der Komponente ist. Technologisch ermöglicht erst diese Schnittstelle die Nutzung der Funktionalität, die durch das Feature *Personenverwaltung* spezifiziert ist. Damit ist die Schnittstelle für die Realisierung des Features essentiell, da das Feature durch die Komponente *Dienstprogramme* und auch durch die Schnittstelle *Rechtezugriff* realisiert wird.

B.2.4 Zusammenfassung

Wie in den vergangenen Abschnitten dargestellt, werden in dieser Arbeit folgende Mengen verwendet:

- E = Menge aller Entitäten eines Software-Systems.
- F = Menge aller Features ($F \subseteq E$).

- A = Menge aller Komponenten ($A \subseteq E$).
- K = Menge aller Klassen ($K \subseteq E$).
- S = Menge aller Schnittstellen($S \subseteq E$).
- D = Menge aller Datendateien ($D \subseteq E$).

Außerdem gilt: $E = F \cup A \cup K \cup S \cup D$.

Anhang C

Erarbeitete Metriken

In diesem Kapitel werden die Metriken vorgestellt, die im Rahmen der Arbeit zur Bewertung von Qualitätsteilmerkmalen des Qualitätsziels *Weiterentwicklungsfähigkeit* definiert wurden, da für diese Zwecke keine geeigneten Metriken in der Literatur gefunden wurden. Sie wurden vorab bereits in [BR08b, BBR09] veröffentlicht. In Abschnitt C.1 werden grundlegende Informationen zu den erarbeiteten Metriken gegeben. In Abschnitt C.2 ist eine detaillierte Beschreibung der Metriken zu finden.

C.1 Definition von Metriken

Bisher gibt es in der Literatur keine allgemeine Übereinstimmung über das Format, wie eine Metrik definiert werden soll [Lil08]. Die folgende Arbeit orientiert sich an Bennicke et al. [BR04]:

Jede der in Abschnitt C.2 vorgestellten Metriken hat einen eindeutigen Namen, unter dem sie referenziert werden kann. Die Metriken haben als Produktmodell[1] den Quelltext und Abhängigkeitsbeziehungen, die mit Hilfe des in Abschnitt B eingeführten Abhängigkeitsmodells definiert wurden.

In der Arbeit von Bennicke et al. [BR04] wird außerdem das Festlegen der benutzten Skalentypen gefordert. In dieser Arbeit wird stets der Skalentyp *Absolutskala* verwendet, da eine Absolutskala die einzige Möglichkeit darstellt, einen Sachverhalt zu

[1] Dieses Modell legt die Grundlage zur Messung fest. Beispiele sind Quelltext oder Strukturdiagramme [BR04].

messen [Lig02]. Beispiele für Sachverhalte sind Zählergebnisse, wie etwa die Anzahl von Fehlern in einem Software-System oder Häufigkeiten, wie Lines of Code.

C.2 Vorstellung der erarbeiteten Metriken

Metriken, welche strukturelle Eigenschaften der Entitäten auf Grundlage von Abhängigkeitsbeziehungen zwischen Komponenten oder Klassen im Verbund bewerten, gehören zu der Kategorie „Strukturelle Metriken“ [Hof08]. In Tabelle C.1 sind alle strukturellen Metriken dargestellt, die in dieser Arbeit hergeleitet und in der Fallstudie angewandt wurden.

Tabelle C.1: Übersicht der in dieser Arbeit erstellten Metriken.

Akronym	Metrik	Abschnitt
IF	Isolierte Features	C.2.1
IE	Isolierte Entitäten	C.2.1
$FSCA$	Feature Scattering	C.2.2
$FTANG$	Feature Tangling	C.2.3
FG	Feature-Größe	C.2.4

C.2.1 Isolierte Entitäten

Die Funktionalität eines Systems wird durch die Features bestimmt, die es zur Verfügung stellt. In dieser Arbeit sind Entitäten isoliert, wenn sie existieren aber nicht an ein Feature gebunden sind. Sie haben damit auch keine Berechtigung, umgesetzt zu werden. Im Allgemeinen können sie in verschiedenen Phasen auftreten und daher für jeden Entitätstyp zutreffen.

Für die Definition von Metriken werden die Mengen verwendet, die im vorherigen Abschnitt B.2 bereits definiert wurden.

Außerdem werden Traceability-Links vom Typ *realisiert-durch* ausgewertet (dargestellt als $\xrightarrow{r}$, wie in Kapitel B.2.3 definiert) jeweils zwischen einem Feature f und X,

einer Teilmenge von E.

Definition 3.17: Isolierte Features IF
Gegeben sei ein Feature f aus der Gesamtmenge F der Features und eine Teilmenge X der Entitäten E; X sind die Entitäten des Systems, welche Features realisieren sollen: $X \subseteq \{A \cup S \cup K \cup D\}$. Dann ist die Anzahl der *Isolierten Features* IF, definiert als:

$$IF(F,X) := \left| \{ f \in F : \nexists x \in X : f \xrightarrow{r} x \} \right|.$$

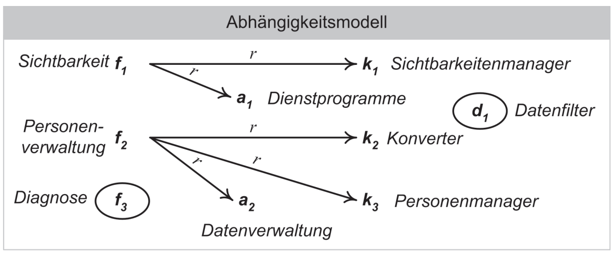
Beispielsweise ist in Abbildung C.1 das Feature *Diagnose* isoliert, dies zeigt der Wert von $IF = 1$. Ein typischer Grund für isolierte Features sind Features niedriger Priorität, die für eine Produktauslieferung eingeplant wurden, aber nicht umgesetzt werden können, da während der Analysephase nicht alle Informationen zur Klärung der Realisierung erfasst werden konnten.

Falls Entitäten aus der Teilmenge X existieren, die keinen Traceability-Link vom Typ *realisiert-durch* zu mindestens einem Feature haben, sind diese Entitäten isoliert.

Definition 3.18: Isolierte Entitäten IE
Gegeben sei ein Feature f aus der Gesamtmenge F der Features und eine Teilmenge X der Entitäten E; X sind die Entitäten des Systems, welche Features realisieren sollen: $X \subseteq \{A \cup S \cup K \cup D\}$. Die Anzahl der isolierten Entitäten ist dann definiert als:

$$IE(F,X) := \left| \{ x \in X : \nexists f \in F : f \xrightarrow{r} x \} \right|.$$

Betrachtet man in Def. 3.18 für X nur typgleiche Entitäten, also A, S, K oder D, dann erhält man spezielle Ausprägungen der Metrik IE: *Isolierte Komponenten* $IE(A)$, *Isolierte Klassen* $IE(K)$ und *Isolierte Datendateien* $IE(D)$. Beispielsweise ist mit $IE(D) = 1$ die Datendatei in Abbildung C.1 mit dem Namen *Datenfilter* isoliert.



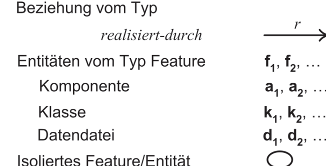


Abbildung C.1: Beispiele für Isolierte Entitäten.

C.2.2 Feature Scattering

Die beiden Metriken *SCA* und *FSCA* bewerten Feature Scattering (Feature-Streuung) hinsichtlich verschiedener Typen von Entitäten.

Definition 3.19: Scattering eines Features

Gegeben sei ein Feature f aus der Gesamtmenge F und eine Teilmenge X aller typgleichen Entitäten der Menge der Entitäten E, also eine der Mengen A, S, K oder D. Das Feature f sei nicht isoliert bezüglich X, d.h. es gibt ein $x \in X$ mit $f \xrightarrow{r} x$. Das Scattering des Features f bezüglich der Menge X wird definiert als die Anzahl von Realisiert-durch-Beziehungen, die zwischen f und Entitäten aus X existieren:

$$SCA(f,X) := \left|\{x \in X : f \xrightarrow{r} x\}\right| - 1.$$

Der theoretisch ideale Fall (siehe Abbildung 4.10 links), bei dem jedes Feature f durch genau eine Entität x implementiert wird, ergibt durch die Subtraktion von 1 den Bestwert 0. Wird beispielsweise ein Feature f zu mehr als einer Komponente $a \in A$ abgebildet, ist das Ergebnis größer als 0.

Ein normierter Gesamtwert über alle Features wird durch $FSCA(F,X)$ ermittelt:

Definition 3.20: *Feature Scattering* $FSCA$
Gegeben sei die Menge aller Features F und eine Teilmenge X aller typgleichen Entitäten der Menge der Entitäten E, also eine der Mengen A, S, K oder D. Dann ist das Feature Scattering $FSCA(F,X)$ definiert als:

$$FSCA\,(F,X) := \frac{\sum_{f\in F} SCA\,(f,X)}{|F|\cdot|X|},\ FSCA \in [0,1)\,.$$

Je höher die Streuung von Features ist, desto schlechter wird die Änderbarkeit bzw. Erweiterbarkeit und umso mehr nähert sich das Ergebnis der Metrik $FSCA(F,X)$ dem Wert 1 an. Dies ist beispielsweise dann der Fall, wenn nur sehr wenige Features durch sehr viele Entitäten realisiert werden.

Wenn das isolierte Feature f_3 und die isolierte Datendatei d_1 in Abbildung C.1 nach Anwendung des Reengineering-Rezepts *Spezifizierte Implementierung* entfernt worden sind, dann existiert beispielsweise Streuung durch das Feature ***Personenverwaltung*** (f_2), da es von den Klassen ***Konverter*** k_2 und ***Personenmanager*** k_3 realisiert wird. Dies zeigt die Metrik mit $SCA(f_2,K) = 1$. Insgesamt existiert eine Feature-Streuung bezüglich der Klassen von einem Gesamtwert $FSCA(F,K) = 1/6$.

C.2.3 Feature Tangling

Die beiden Metriken $TANG$ und $FTANG$ bewerten Feature Tangling (Feature-Verschränkung) hinsichtlich verschiedenster Typen von Entitäten.

Definition 3.21: Feature Tangling bezüglich einer Entität
Gegeben sei die Menge aller Features F und eine Entität x aus einer der Mengen A, S, K oder D. x sei nicht isoliert, d.h. es gibt ein $f \in F$ mit $f \xrightarrow{r} x$. Das Feature Tangling der Entität x wird definiert als die Anzahl von Realisiert-Durch-Beziehungen zwischen den Features aus F und x:

$$TANG\,(x,F) := \left|\{f \in F : f \xrightarrow{r} x\}\right| - 1.$$

Der ideale Fall, bei dem jedes Feature f durch genau eine Entität x implementiert wird (siehe Abbildung 4.10 links), ergibt durch die Subtraktion von 1 den Bestwert

0.

Wird beispielsweise ein Feature f zu mehr als einer Komponente $a \in A$ abgebildet, ist das Ergebnis größer als 0. Im Falle von Feature-Verschränkung existieren mindestens zwei Traceability-Links vom Typ *realisiert-durch*, zwischen mindestens zwei Features und einer Entität $x \in X$.

Ein normierter Gesamtwert über alle Entitäten und Features wird durch $\boldsymbol{FTANG(F,X)}$ ermittelt:

Definition 3.22: Feature Tangling *FTANG*
Gegeben sei die Menge aller Features $\boldsymbol{F}$ und eine Teilmenge $\boldsymbol{X}$ aller typgleichen Entitäten der Menge der Entitäten $\boldsymbol{E}$, also eine der Mengen $\boldsymbol{A}$, $\boldsymbol{S}$, $\boldsymbol{K}$ oder $\boldsymbol{D}$. Dann ist das Feature Tangling $\boldsymbol{FTANG(F,X)}$ definiert als:

$$FTANG(F,X) := \frac{\sum_{x \in X} TANG(x,F)}{|F| \cdot |X|},\; FTANG \in [0,1)\,.$$

Je höher die Feature-Verschränkung innerhalb der betrachteten Entitäten ist, desto schwieriger ist eine Anpassung der Entitäten und umso mehr nähert sich das Ergebnis der Bewertungsmetrik $\boldsymbol{FTANG(F,X)}$ dem Wert 1 an. Dies ist zum Beispiel dann der Fall, wenn sehr viele Features durch nur ganz wenige Entitäten realisiert werden.

C.2.4 Feature-Größe

Nach Baker et al. [BBF+90] sollte eine Größenmetrik dem intuitiven menschlichen Verstehen von Größe entsprechen. Die Größe des betrachteten Systems lässt sich durch die Anzahl der beteiligten Entitäten bestimmen [GC87], beispielsweise durch die *Anzahl von Klassen*. In dieser Arbeit wird die Anzahl von Features $\boldsymbol{F}$, von Komponenten $\boldsymbol{A}$, von Schnittstellen $\boldsymbol{S}$, von Klassen $\boldsymbol{K}$ und von Datendateien $\boldsymbol{D}$ eines Software-Systems betrachtet. Neben diesen Standardgrößen wird die *Feature-Größe* ermittelt, die definiert ist als die Anzahl von Entitäten, die verwendet werden, um ein Feature

zu realisieren.

Definition 3.23: Anzahl Entitäten pro Feature $AEPF$
Gegeben sei ein Feature f aus der Gesamtmenge F und eine Teilmenge X aller typgleichen Entitäten der Menge der Entitäten E, also eine der Mengen A, S, K oder D. Dann ist die Anzahl von Entitäten pro Feature $AEPF(f,X)$ definiert als:

$$AEPF(f,X) := \left|\{x \in X : f \xrightarrow{r} x\}\right|.$$

Nach Def. 3.23 lassen sich verschiedene Ausprägungen der Metrik zur Bestimmung der Größe eines Feature bezüglich eines Entitätstyps X ableiten: Dies sind die Anzahl von Komponenten $AEPF(f,A)$, Schnittstellen $AEPF(f,S)$, Klassen $AEPF(f,K)$ und Datendateien $AEPF(f,D)$, die ein Feature f aus der Gesamtmenge von Features F realisieren.

Definition 3.24: *Feature-Größe* FG
Gegeben sei ein Feature f aus der Gesamtmenge F und die Teilmengen A, S, K oder D typgleicher Entitäten der Menge der Entitäten E. Dann ist die Größe eines Features $FG(f)$ definiert als:

$$FG(f) := \sum_{X \in \{A,S,K,D\}} AEPF(f,X).$$

Beispielsweise ist die Größe des Features *Personenverwaltung* (f_2) in Abbildung C.1 $FG(f_2) = 3$. Diese ergibt sich daraus, dass $AEPF(f_2,A)$ den Wert 1 hat (Beziehung zwischen f_2 und der Komponente *Datenverwaltung* (a_2)), und $AEPF(f_3,K)$ den Wert 2 hat (Beziehungen zwischen f_3 und den Klassen *Konverter* (k_2) und *PersonenManager* (k_3)).

Die Metrik FG hat eine gewisse Ähnlichkeit zu der Metrik $FSCA$. $FSCA$ normiert allerdings auf die Systemgröße (Division durch $|F| \cdot |X|$), während FG auch bei guten Entwürfen in der Regel mit der Systemgröße wächst. Somit kann sie zu Vergleichszwecken eingesetzt werden, um Klassen von Projekten ähnlicher Komplexität gegenüberzustellen.

Anhang D

Tabellen der Messergebnisse

Im Folgenden werden weitere Messergebnisse aus der Fallstudie aufgeführt. Tabelle D.1 zeigt eine Übersicht über die in der Fallstudie erzeugten Baselines und die Messergebnisse anhand der in dieser Arbeit erstellten Metriken.

Tabelle D.1: Übersicht der Ergebnisse der in dieser Arbeit erstellten Metriken.

Baseline	IF	$IE(A)$	$IE(S)$	$IE(K)$	$IE(D)$	$FTANG(A)$	$FSCA$	$KOIZ$	$KLIZ$
$V32$	–	–	–	–	–	–	–	–	–
$V32.1$	28	1	46	131	9	–	–	0	2
$V32.2$	0	0	0	0	1	0,964	0,000	0	2
$V32.3$	0	0	0	0	0	0,964	0,000	0	2
$V32.4$	0	0	0	0	0	0,482	0,018	1	2
$V32.5$	0	0	0	0	0	0,310	0,012	1	2
$V32.6$	0	0	0	0	0	0,223	0,018	1	2
$V32.7$	0	0	0	0	0	0,179	0,014	1	2
$V32.8$	0	0	0	0	0	0,149	0,012	1	2
$V32.9$	0	0	0	0	0	0,149	0,012	0	0

Um weitere Systemkennzahlen über die Implementierung der Komponente *Datenverwaltung* der Fallstudie zu ermitteln und dem Leser einen Eindruck über das Software-System zu geben, wurden die in der Arbeit von [CK94] vorgeschlagenen Metriken zur Bewertung von Objekt-Orientierten Software-Systemen auf die initiale Baseline $V32$ angewandt. Das Ergebnis ist in Tabelle D.2 dargestellt.

Tabelle D.2: Ergebnisse zur Baseline $V32$.

k	WMC	DIT	NOC	CBO	RFC	LCOM	CA	NPM	MCCABE
1	3	3	0	1	8	3	6	3	1,000
2	28	1	0	7	28	378	2	28	0,000
3	1	1	0	0	4	0	1	0	0,000
4	19	1	0	3	25	171	27	18	0,000
5	3	5	0	1	6	3	2	3	1,000
6	7	1	0	5	7	21	2	7	0,000
7	2	1	0	2	10	1	1	2	3,000
8	14	1	0	14	59	7	1	0	3,385
9	2	1	0	0	2	1	35	2	0,000
10	12	1	0	7	12	66	2	12	0,000
11	3	5	0	1	6	3	11	3	1,000
12	1	1	0	0	4	0	1	0	0,000
13	8	1	0	4	14	28	16	7	0,000
14	23	1	0	7	23	253	2	23	0,000
15	1	1	0	0	4	0	1	0	0,000
16	14	1	0	4	20	91	15	13	0,000
17	3	5	0	1	6	3	1	3	1,000
18	13	1	0	24	47	34	5	3	1,667
19	3	5	0	1	6	3	9	3	1,000
20	1	1	0	0	1	0	63	1	0,000
21	3	5	0	1	6	3	1	3	1,000
22	3	5	0	1	6	3	6	3	1,000
23	4	6	0	1	8	6	1	4	1,000
24	3	5	0	1	6	3	10	3	1,000
25	3	5	0	1	6	3	10	3	1,000
26	3	6	0	1	6	3	1	3	1,000
27	3	5	0	1	6	3	1	3	1,000
28	1	1	0	0	4	0	1	0	0,000
29	9	1	0	4	15	36	16	8	0,000
30	7	1	0	0	7	21	10	7	0,000
31	22	1	0	7	22	231	2	22	0,000
32	1	1	0	0	4	0	1	0	0,000
33	35	1	0	7	41	595	13	34	0,000
34	6	1	0	0	6	15	8	6	0,000
35	6	1	0	0	22	15	0	5	2,167

Tabelle D.2: Ergebnisse zur Baseline $V32$.

k	WMC	DIT	NOC	CBO	RFC	LCOM	CA	NPM	MCCABE
36	5	5	2	1	8	0	23	5	1,000
37	88	1	0	48	283	688	25	0	4,632
38	71	1	0	6	85	2281	27	0	1,338
39	29	2	0	25	92	196	1	29	2,000
40	31	2	0	24	89	0	7	0	1,581
41	25	2	1	5	39	226	3	20	1,000
42	18	2	1	31	72	109	2	0	2,167
43	8	2	0	17	28	28	1	8	1,000
44	12	4	0	4	34	0	10	10	2,000
45	21	1	0	12	51	134	2	0	2,300
46	20	1	0	10	50	112	3	1	3,158
47	13	2	0	21	53	72	1	13	1,615
48	12	2	0	21	35	0	4	0	1,167
49	12	2	0	6	26	42	2	9	1,083
50	13	2	0	27	54	52	1	0	2,077
51	3	2	0	3	5	3	1	1	1,000
52	3	2	0	3	5	3	1	1	1,000
53	3	2	0	3	5	3	1	1	1,000
54	3	2	0	3	5	3	1	1	1,000
55	4	1	4	2	5	4	5	1	1,000
56	0	1	0	0	0	0	6	0	1,824
57	22	1	0	6	41	149	6	11	1,609
58	24	2	0	25	87	66	1	24	2,500
59	28	2	0	27	88	0	5	0	1,786
60	19	2	0	5	35	125	2	15	1,000
61	18	2	0	30	70	115	1	0	1,944
62	9	1	0	5	28	36	13	0	2,000
63	9	1	0	2	17	20	12	8	1,667
64	31	2	2	12	67	395	2	7	1,677
65	15	2	2	26	63	79	2	0	2,667
66	13	1	0	14	51	78	9	0	3,385
67	16	1	0	2	26	84	1	8	1,438
68	25	2	0	30	104	60	1	23	3,200
69	30	2	0	29	106	0	7	0	2,367
70	54	3	0	12	104	1363	3	38	1,577

Tabelle D.2: Ergebnisse zur Baseline $V32$.

k	WMC	DIT	NOC	CBO	RFC	LCOM	CA	NPM	MCCABE
71	10	3	0	20	35	43	1	0	2,100
72	16	1	0	1	31	84	1	8	1,533
73	2	1	0	2	7	1	1	2	2,000
74	119	1	0	45	266	2659	17	2	4,593
75	5	1	0	1	11	10	24	1	1,000
76	26	1	0	9	66	311	26	6	2,040
77	15	2	0	23	73	49	1	15	2,867
78	18	2	0	23	58	0	6	0	1,278
79	19	3	0	8	45	171	3	17	1,000
80	5	3	0	17	21	10	1	0	2,000
81	22	1	0	2	35	63	7	7	1,048
82	23	1	0	5	44	15	9	18	2,870
83	14	2	0	18	47	71	1	14	1,714
84	15	2	0	25	65	0	7	0	1,533
85	21	2	0	5	37	160	2	16	1,000
86	17	2	0	29	71	98	1	0	1,941
87	36	2	0	27	105	0	1	35	2,444
88	36	2	0	26	89	0	4	0	1,972
89	15	3	0	4	30	81	2	12	1,077
90	8	3	0	27	62	22	1	0	3,875
91	22	1	8	28	69	137	8	0	1,636
92	6	1	7	12	22	1	7	0	1,167
93	46	1	5	9	102	521	5	18	2,457
94	28	1	5	19	84	318	5	0	2,500
95	3	5	0	1	6	3	11	3	1,000
96	10	1	0	0	22	21	1	9	2,375
97	5	1	0	1	9	2	3	4	1,400
98	1	1	0	0	1	0	24	1	0,000
99	14	1	0	7	14	91	2	14	0,000
100	14	1	0	5	14	91	13	14	0,000
101	6	1	0	0	6	15	44	6	0,000
102	7	1	0	1	7	21	52	7	0,000
103	13	1	0	7	13	78	2	13	0,000
104	1	1	0	0	4	0	1	0	0,000
105	15	1	0	4	21	105	18	14	0,000

Tabelle D.2: Ergebnisse zur Baseline $V32$.

k	WMC	DIT	NOC	CBO	RFC	LCOM	CA	NPM	MCCABE
106	33	1	0	7	33	528	2	33	0,000
107	11	1	0	2	11	55	9	11	0,000
108	1	1	0	1	1	0	1	1	0,000
109	164	1	0	13	164	13366	2	164	0,000
110	174	1	0	21	369	0	0	171	1,919
111	167	1	0	37	360	10307	4	0	2,404
112	163	1	0	22	341	0	0	163	1,006
113	4	5	0	0	10	2	1	4	1,000
114	3	4	11	1	6	3	52	0	1,000
115	19	3	0	5	40	147	0	5	2,059
116	2	1	0	1	4	1	1	2	1,000
117	0	1	0	1	0	0	11	0	0,000
118	0	1	0	2	0	0	8	0	0,000
119	0	1	0	1	0	0	8	0	0,000
120	2	1	0	2	2	1	6	2	0,000
121	0	1	0	2	0	0	9	0	0,000
122	0	1	0	1	0	0	9	0	0,000
123	5	1	0	4	17	8	1	1	1,750
124	1	1	0	1	1	0	17	1	0,000
125	0	1	0	1	0	0	7	0	0,000
126	1	1	0	1	1	0	11	1	0,000
127	7	1	0	1	7	21	20	7	0,000
128	2	1	0	1	2	1	13	2	0,000
129	0	1	0	1	0	0	12	0	0,000
130	1	1	0	0	1	0	41	1	0,000
131	10	1	0	1	12	45	2	10	1,000
132	5	2	0	7	15	8	3	1	1,800
133	5	2	0	7	15	8	3	1	1,800
134	8	2	0	7	18	24	3	2	1,625
135	14	1	5	9	28	63	10	7	1,357
136	8	2	0	7	19	22	3	3	1,625
137	3	2	0	7	12	3	3	0	2,000
138	30	2	0	46	99	421	1	0	2,214
139	3	2	0	1	4	1	5	0	1,000
140	13	1	0	5	26	66	6	11	1,462

Tabelle D.2: Ergebnisse zur Baseline $V32$.

k	WMC	DIT	NOC	CBO	RFC	LCOM	CA	NPM	MCCABE
141	13	3	0	2	23	76	5	9	1,077
142	17	2	0	36	64	130	1	0	2,733
143	3	2	0	1	4	0	5	0	1,000
144	7	4	0	6	20	21	3	6	1,167
145	7	4	0	6	20	21	2	6	1,167
146	7	4	0	6	21	21	1	6	1,167
147	9	4	0	7	25	36	3	8	1,375
148	7	4	0	6	19	21	2	6	1,167
149	16	2	0	31	57	114	1	0	2,643
150	6	2	0	1	7	11	5	0	1,000
151	27	1	0	22	76	331	1	16	1,593
152	7	1	5	1	11	17	11	2	1,000
153	9	3	0	3	24	4	2	7	1,111
154	22	1	0	20	41	231	4	20	1,000
155	3	3	0	8	12	3	1	3	4,667
156	111	1	0	65	227	6079	1	77	1,236
157	37	2	0	44	104	642	2	2	2,457
158	5	2	0	1	6	0	5	0	1,000
159	11	1	0	2	17	33	1	6	1,000
160	1	4	0	1	2	0	4	1	0,000
161	12	1	0	2	20	62	1	12	1,000
162	8	1	0	2	20	14	3	4	2,125
163	16	2	0	33	56	114	2	0	2,357
164	1	2	0	1	2	0	8	0	1,000
165	15	1	5	26	46	99	5	2	2,615
166	77	1	0	28	77	2926	3	77	0,000
167	3	5	0	1	6	3	6	3	1,000
168	3	5	0	1	6	3	9	3	1,000
169	3	5	0	1	6	3	8	3	1,000
170	3	5	0	1	6	3	17	3	1,000
171	4	5	0	1	8	6	9	4	1,000
172	3	5	0	1	6	3	9	3	1,000
173	3	5	0	1	6	3	9	3	1,000
174	9	1	0	0	9	36	29	9	0,000
175	15	1	0	2	15	105	5	15	0,000

Tabelle D.2: Ergebnisse zur Baseline *V*32.

k	WMC	DIT	NOC	CBO	RFC	LCOM	CA	NPM	MCCABE
176	3	1	0	1	15	3	4	1	1,667
177	6	4	9	1	13	3	13	6	1,000
178	20	1	0	8	20	190	13	20	0,000
179	9	1	0	7	20	32	1	3	1,556
180	6	1	0	0	6	15	7	6	0,000
181	8	1	0	0	8	28	4	8	0,000
182	2	1	0	0	2	1	8	2	0,000
183	3	5	0	1	6	3	11	3	1,000
184	3	5	0	1	6	3	10	3	1,000
185	15	1	0	2	15	105	25	15	1,000

Um Systemkennzahlen über die in der Baseline *V*32.4 aufgeteilten Komponente ***Datenzugriff*** zu bekommen, welche das Feature ***Datenzugriffkapselung*** (f_{21}) realisiert, wurden die Metriken ***WMC***, ***CBO***, ***LCOM*** und ***MCCABE*** angewandt. Das Ergebnis ist in Tabelle D.3 zu finden.

Tabelle D.3: Ergebnisse zur Baseline *V*32.4 bezüglich Feature Datenzugriffkapselung f_{21}.

k	WMC	CBO	LCOM	MCCABE
42	18	31	109	2,167
50	13	27	52	2,077
61	18	30	115	1,944
65	15	26	79	2,667
71	10	20	43	2,100
74	119	45	2659	4,593
80	5	17	10	2,000
86	17	29	98	1,941
90	8	27	22	3,875
94	28	19	318	2,500
132	5	7	8	1,800
134	8	7	24	1,625
135	14	9	63	1,357

Tabelle D.3: Ergebnisse zur Baseline $V32.4$ bezüglich Feature Datenzugriffkapselung f_{21}.

k	WMC	CBO	LCOM	MCCABE
136	8	7	22	1,625
137	3	7	3	2,000
138	30	46	421	2,214
139	3	1	1	1,000
140	13	5	66	1,462
141	13	2	76	1,077
142	17	36	130	2,733
143	3	1	0	1,000
144	7	6	21	1,167
146	7	6	21	1,167
147	9	7	36	1,375
148	7	6	21	1,167
149	16	31	114	2,643
150	6	1	11	1,000
151	27	22	331	1,593
152	7	1	17	1,000
153	9	3	4	1,111
154	22	20	231	1,000
155	3	8	3	4,667
156	111	65	6079	1,236
157	37	44	642	2,457
158	5	1	0	1,000
159	11	2	33	1,000
161	12	2	62	1,000
162	8	2	14	2,125
163	16	33	114	2,357
164	1	1	0	1,000
165	15	26	99	2,615
176	3	1	3	1,667
177	6	1	3	1,000
179	9	7	32	1,556

Die Komponente ***Fehlerverarbeitung*** realisiert das Feature ***Fehlerverwaltung*** f_{22}. Um Systemkennzahlen über die in der Baseline *V*32.5 aufgeteilten Komponente ***Fehlerverarbeitung*** zu bekommen, wurden die Klassen der Komponente ***Fehlerverarbeitung*** anhand der in Tabelle D.4 gezeigten Metriken vermessen.

Tabelle D.4: Klassen der Komponente Fehlerverarbeitung, welche das Feature Fehlerverwaltung f_{22} realisieren.

k	WMC	DIT	NOC	CBO	LCOM	CA	MCCABE
5	3	5	0	1	3	2	1,000
11	3	5	0	1	3	11	1,000
17	3	5	0	1	3	1	1,000
19	3	5	0	1	3	9	1,000
21	3	5	0	1	3	1	1,000
22	3	5	0	1	3	6	1,000
23	4	6	0	1	6	1	1,000
24	3	5	0	1	3	10	1,000
25	3	5	0	1	3	10	1,000
26	3	6	0	1	3	1	1,000
27	3	5	0	1	3	1	1,000
36	5	5	2	1	0	23	1,000
95	3	5	0	1	3	11	1,000
113	4	5	0	0	2	1	1,000
114	3	4	11	1	3	52	1,000
167	3	5	0	1	3	6	1,000
168	3	5	0	1	3	9	1,000
169	3	5	0	1	3	8	1,000
170	3	5	0	1	3	17	1,000
171	4	5	0	1	6	9	1,000
172	3	5	0	1	3	9	1,000
173	3	5	0	1	3	9	1,000
177	6	4	9	1	3	13	1,000
183	3	5	0	1	3	11	1,000
184	3	5	0	1	3	10	1,000

Literaturverzeichnis

[AHKP09] Christian Arndt, Christian Hermanns, Herbert Kuchen, and Michael Poldner. Best Practices in der Softwareentwicklung. Technical Report, Westfälische Wilhelms-Universität Münster, 2009.

[AJMM05] Silvia T. Acuna, Natalia Juristo, Ana Maria Moreno, and Alicia Mon. *A Software Process Model Handbook for Incorporating People's Capabilities*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.

[Ars08] Danijel Arsenovski. *Professional Refactoring in Visual Basic*. Wrox, 2008.

[Bas92] Victor R. Basili. Software modeling and measurement: the Goal/Question/Metric paradigm. www.cs.umd.edu/~basili/publications/technical/T78.pdf, 1992.

[BBF+90] A. L. Baker, J. M. Bieman, N. Fenton, D. A. Gustafson, A. Melton, and R. Whitty. A philosophy for software measurement. *Journal of Systems and Software*, 12(3):277–281, 1990.

[BBG+97] J. W. Backus, F. L. Bauer, J. Green, C. Katz, J. McCarthy, A. J. Perlis, H. Rutishauser, K. Samelson, B. Vauquois, J. H. Wegstein, A. van Wijngaarden, and M. Woodger. *Revised report on the algorithmic language ALGOL 60*, pages 19–49. Birkhauser Boston Inc., Cambridge, MA, USA, 1997.

[BBM96] Victor R. Basili, Lionel C. Briand, and Walclio L. Melo. A Validation of Object-Oriented Design Metrics as Quality Indicators. *IEEE Transactions on Software Engineering*, 22:751–761, 1996.

[BBR09] Robert Brcina, Stephan Bode, and Matthias Riebisch. Optimisation Process for Maintaining Evolvability during Software Evolution. In

ECBS ´09: Proceedings of the 16th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems, 2009.

[BC03] Carliss Y. Baldwin and Kim B. Clark. The Value, Costs and Organizational Consequences of Modularity. http://www.people.hbs.edu/cbaldwin/DR1/DR1Overview.pdf, 2003.

[BCE07] Hongyu Pei Breivold, Ivica Crnkovic, and Peter Eriksson. Evaluating Software Evolvability. In *SERPS '07: Proceedings of the 7th Conference on Software Engineering Research and Practice in Sweden*, 2007.

[BCK03] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice*. Addison-Wesley Professional, 2nd edition, 2003.

[BCLL08] Hongyu Pei Breivold, Ivica Crnkovic, Rikard Land, and Stig Larsson. Using Dependency Model to Support Software Architecture Evolution. In *ASE '08: 4th International ERCIM Workshop on Software Evolution and Evolvability at the 23rd IEEE/ACM International Conference on Automated Software Engineering*, 2008.

[BD02] Jagdish Bansiya and Carl Davis. A Hierarchical Model for Object-Oriented Design Quality Assessment. *IEEE Transactions on Software Engineering*, 28(1):4–17, 2002.

[BDW98] Lionel C. Briand, John W. Daly, and Jürgen Wüst. A Unified Framework for Cohesion Measurement in Object-Oriented Systems. *Empirical Software Engineering*, 3(1):65–117, 1998.

[Ble08] Oliver Bleicher. Extension of a knowledge based framework for automatic generation of traceability information to manage software evolution. Master's thesis, Brunel University, West London, 2008.

[BMM98] William J. Brown, Raphael C. Malveau, and Thomas J. Mowbray. *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*. Wiley, 1998.

[Bod11] Stephan Bode. *Quality Goal Oriented Architectural Design and Traceability for Evolvable Software Systems*. PhD thesis, Ilmenau University of Technology, 2011.

[Boe74] Barry W. Boehm. Some Steps Toward Formal and Automated Aids to Software Requirements Analysis and Design. In *Proceedings, IFIP Congress*, pages 192–197, 1974.

[Boo94] Grady Booch. *Object-Oriented Analysis and Design with Applications.* Benjamin–Cummings, Redwood City, California, 2nd edition, 1994.

[BR04] M. Bennicke and H. Rust. Programmverstehen und statische Analysetechniken im Kontext des Reverse Engineering und der Qualitätssicherung. Technical Report, 2004.

[BR08a] Robert Brcina and Matthias Riebisch. Architecting for Evolvability by Means of Traceability and Features. In *ASE '08: 4th International ERCIM Workshop on Software Evolution and Evolvability at the 23rd IEEE/ACM International Conference on Automated Software Engineering*, 2008.

[BR08b] Robert Brcina and Matthias Riebisch. Defining Traceability Links Semantics for Design Decision Support. In *ECMDA Traceability Workshop*, Berlin, 2008.

[BR10] Stephan Bode and Matthias Riebisch. Impact evaluation for quality-oriented architectural decisions regarding evolvability. In *ECSA '10: Proceedings of the 4th European conference on Software architecture*, pages 182–197. Springer-Verlag Berlin Heidelberg, 2010.

[BR11] Stefan Bode and Matthias Riebisch. EMFTrace. https://pi0.theoinf.tu-ilmenau.de/trac/EMFTrace, letzter Zugriff: Mar 2011.

[BV00] Dragan Bojic and Dusan Valesevic. Reverse Engineering of Use Case Realisations in UML. In *SAC '00 Proceedings of the 2000 ACM symposium on applied computing - Volume 2*, 2000.

[CB91] Gianluigi Caldiera and Victor R. Basili. Identifying and qualifying reusable software components. *IEEE Computer*, 24:61–70, 1991.

[CC90] Elliot J. Chikofsky and James H. Cross II. Reverse Engineering and Design Recovery: A Taxonomy. *IEEE Software*, 7(1):13–17, 1990.

[CDK98] Shyam R. Chidamber, David P. Darcy, and Chris F. Kemerer. Managerial Use of Metrics for Object-Oriented Software: An Exploratory Analysis. *IEEE Transactions on Software Engineering*, 24:629–639, 1998.

[CG90] David N. Card and Robert L. Glass. *Measuring Software Design Quality*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1990.

[CJH01] Stephen Cook, He Ji, and Rachel Harrison. Dynamic and Static Views of Software Evolution. In *ICSM '01: Proceedings of the 17th IEEE International Conference on Software Maintenance*, pages 592–601. IEEE Computer Society, 2001.

[CK94] Shyam R. Chidamber and Chris F. Kemerer. A Metrics Suite for Object Oriented Design. *IEEE Transactions on Software Engineering*, 20(6):476–493, 1994.

[CNYM00] Lawrence Chung, Brian A. Nixon, Eric Yu, and John Mylopoulos. *Non-functional Requirements in Software Engineering*, volume 5 of *International Series in Software Engineering*. 2000.

[CvdB06] Selim Ciraci and Pim van den Broek. Evolvability as a Quality Attribute of Software Architectures. In Maja D'Hondt Laurence Duchien and Kim Mens, editors, *Proceedings of the International ERCIM Workshop on Software Evolution*, 2006.

[DB85] Carl W. Doerflinger and Victor R. Basili. Monitoring Software Development Through Dynamic Variables. *IEEE Transactions on Software Engineering*, 11(9):978–985, 1985.

[dBDV04] Bart du Bois, Serge Demeyer, and Jan Verelst. Refactoring - Improving Coupling and Cohesion of Existing Code. In *WCRE '04: Proceedings of the 11th Working Conference on Reverse Engineering*, pages 144–151, 2004.

[DDN02] Serge Demeyer, Stéphane Ducasse, and Oscar Nierstrasz. *Object Oriented Reengineering Patterns*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002.

[DeM86] Tom DeMarco. *Controlling Software Projects: Management, Measurement, and Estimates*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1986.

[DG02] Leigh A. Davis and Rose F. Gamble. Identifying Evolvability for Integration. In *ICCBSS '02: Proceedings of the First Intl. Conf. on COTS-Based Software Systems*, 2002.

[Dij76] Edsger W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

[DIN94] ISO 9126 (DIN 66272), Beurteilen von Softwareprodukten - Qualitätsmerkmale und Leitfaden zu deren Verwendung, 1994.

[DS92] Ernst Denert and Johannes Siedersleben. *Software-Engineering. Methodische Projektabwicklung*. Springer Verlag, Berlin, 1992.

[DS04] Mike Dean and Guus Schreiber. OWL Web Ontology Language Reference. W3C Recommendation, W3C, 2004.

[DSP08] Karim Dhambri, Houari Sahraoui, and Pierre Poulin. Visual Detection of Design Anomalies. In *CSMR '08: Proceedings of the 12th European Conference on Software Maintenance and Reengineering*, pages 279–283, 2008.

[EG04] Alexander Egyed and Paul Grünbacher. Identifying Requirements Conflicts and Cooperation: How Quality Attributes and Automated Traceability Can Help. *IEEE Software*, 21(6):50–58, 2004.

[EGK+98] Stephen G. Eick, Todd L. Graves, Alan F. Karr, J. S. Marron, and Audris Mockus. Does Code Decay? Assessing the Evidence from Change Management Data. *IEEE Transactions on Software Engineering*, 27:1–12, 1998.

[Egy01] Alexander Egyed. A Scenario-Driven Approach to Traceability. In *23rd International Conference on Software Engineering, ICSE'01*, pages 123–132. IEEE, 2001.

[Egy02] Alexander Egyed. Reasoning about Trace Dependencies in a Multi-Dimensional Space. In *Proceedings of the 1st International Workshop on Traceability, co-located with ASE 2002*, Edinburgh, Scotland, UK, 2002.

[EL96] Karin Erni and Claus Lewerentz. Applying Design-Metrics to Object-Oriented Frameworks. In *Proceedings of the 3rd International Symposium on Software Metrics: From Measurement to Empirical Results*, 1996.

[Fen95] Norman E. Fenton. *Software Metrics: A Rigorous Approach.* Thomson Computer Press, et al., 1995.

[For09] Neal Ford. Evolutionary architecture and emergent design: Emergent design through metrics. http://www.ibm.com/developerworks/java/library/j-eaed6/index.html, 2009.

[Fow97] Martin Fowler. *Analysis Patterns: Reusable Object Models.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.

[Fow99] Martin Fowler. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley Longman Publishing Co., Inc., Amsterdam, 1999.

[Fow01] Martin Fowler. Separating User Interface Code. *IEEE Software*, 18:96–97, 2001.

[FZL05] Tie Feng, Jiachen Zhang, and Wenjin Li. Applying Change Impact Analysis and Design Metrics in CBR Based Software Design Improvement. In *Proceedings of the ISCIT 2005 IEEE International Symposium on Communications and Information Technology*, volume 01, pages 169–172, 2005.

[GC87] Robert B. Grady and Deborah L. Caswell. *Software Metrics: Establishing a Company-Wide Program.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1987.

[GF94] Orlena C. Z. Gotel and Anthony C. W. Finkelstein. An Analysis of the Requirements Traceability Problem. In *Proceedings of the First International Conference on Requirements Engineering, Colorado Springs, CO, USA*, pages 94–101. IEEE Computer Society Press, 1994.

[GHJV95] Erich Gamma, Richard Helm, Ralph E. Johnson, and John Vlissides. *Design Patterns. Elements of Reusable Object-Oriented Software.* Addison-Wesley, Reading, MA, 1995.

[GL98] David Gillibrand and Kecheng Liu. Quality Metrics for Object-Oriented Design. *Journal of Object-Oriented Programming*, 10(8):56–59, 1998.

[GPEM09a] Joshua Garcia, Daniel Popescu, George Edwards, and Nenad Medvidovic. Identifying Architectural Bad Smells. In *CSMR '09: Proceedings*

of the 2009 European Conference on Software Maintenance and Reengineering, pages 255–258, 2009.

[GPEM09b] Joshua Garcia, Daniel Popescu, George Edwards, and Nenad Medvidovic. Toward a Catalogue of Architectural Bad Smells. In *QoSA '09 Proceedings of the 5th International Conference on the Quality of Software Architectures: Architectures for Adaptive Software Systems*, pages 146–162. Springer, 2009.

[Gre07] Orla Greevy. *Enriching Reverse Engineering with Feature Analysis*. PhD thesis, Berne University, Switzerland, 2007.

[Hat99] Les Hatton. Testing: the influence of complexity, coupling, diagnosis and repetitive failure. In *EuroStar'99, European Conference on Software Testing, Analysis and Review, Nov 8 - 12, 1999, Barcelona, Spain*, 1999.

[HC07] Sunny Huynh and Yuanfang Cai. An Evolutionary Approach to Software Modularity Analysis. In *ACoM '07: Proceedings of the First International Workshop on Assessment of Contemporary Modularization Techniques*, page 6, 2007.

[HNS00] Christine Hofmeister, Robert Nord, and Dilip Soni. *Applied Software Architecture*. Addison-Wesley Longman, Boston, MA, USA, 2000.

[Hof08] Dirk W Hoffmann. *Software-Qualität*. Springer-Verlag Berlin Heidelberg, 2008.

[HS96] Brian Henderson-Sellers. *Object-Oriented Metrics: Measures of Complexity*. Prentice-Hall, NJ, USA, 1996.

[IEE90] IEEE Std 610.12-1990. IEEE Standard Glossary of Software Engineering Terminology, 1990.

[IK06] Igor Ivkovic and Kostas Kontogiannis. A Framework for Software Architecture Refactoring using Model Transformations and Semantic Annotations. In *CSMR '06: Proceedings of the Conference on Software Maintenance and Reengineering*, pages 135–144, 2006.

[ISO01] ISO/IEC 9126-1. Software engineering – Product quality – Part 1: Quality model, 2001.

[IST10] ISTQB/GTB. Standardglossar der Testbegriffe, 2010.

[Jac93] Ivar Jacobson. *Object-Oriented Software Engineering: A Use Case Driven Approach*. Addison-Wesley, 1993.

[JRH+03] Mario Jeckle, Chris Rupp, Jürgen Hahn, Barbara Zengler, and Stefan Queins. *UML 2 glasklar*. Hanser Fachbuchverlag, 2003.

[KCH+90] Kyo Kang, Sholom Cohen, James Hess, William Novak, and A. Spencer Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-021, SEI Institute, Carnegie Mellon University, 1990.

[Ker04] Joshua Kerievsky. *Refactoring to Patterns*. Addison-Wesley, 2004.

[KP96] Rainer Koschke and Erhard Plödereder. Ansätze des Programmverstehens. In *Franz Lehner (Hrsg.) Softwarewartung und Reengineering*. Deutscher Universitätsverlag, 1996.

[Kru03] Philippe Kruchten. *The Rational Unified Process: An Introduction*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[Lak96] John Lakos. *Large-scale C++ software design*. Addison-Wesley Longman, Redwood City, CA, USA, 1996.

[Lar05] Craig Larman. *UML und Patterns angewendet: objektorientierte Softwareentwicklung*. Mitp-Verlag, 1st edition, 2005.

[Lig02] Peter Liggesmeyer. *Software-Qualität, Testen, Analysieren und Verifizieren von Software*. Spektrum Akademischer Verlag, 2002.

[Lil08] Carola Lilienthal. *Komplexität von Softwarearchitekturen, Stile und Strategien*. PhD thesis, Universität Hamburg, 2008.

[Lin94] Mikael Lindvall. A study of Traceability in Object-Oriented System Development. Linköping University, Institute of Technology, Sweden, 1994.

[LS99] Ora Lassila and Ralph R. Swick. Resource Description Framework (RDF) Model and Syntax. http://www.w3.org/TR/1999/REC-rdf-syntax-19990222, 1999.

[Mar96] Robert C. Martin. Granularity. http://www.objectmentor.com/resources/articles/granularity.pdf, 1996.

[Mar00] Robert C. Martin. Design Principles and Design Patterns. http://www.objectmentor.com/resources/articles/Principles_and_Patterns.pdf, 2000.

[McC76] Thomas J. McCabe. A Complexity Measure. *IEEE Transactions on Software Engineering*, 2:308–320, 1976.

[Mey00] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall PTR, 2nd edition, 2000.

[MGDM10] Naouel Moha, Yann-Gael Gueheneuc, Laurence Duchien, and Anne-Francoise Le Meur. DECOR: A Method for the Specification and Detection of Code and Design Smells. *IEEE Transactions on Software Engineering*, 36:20–36, 2010.

[Mil56] George A. Miller. The Magical Number Seven, Plus or Minus Two: Some Limits on Our Capacity for Processing Information. *The Psychological Review*, 63:81–97, 1956.

[MM98] Tom Mens and Kim Mens. Assessing the Evolvability of Software Architectures. In *ECOOP '98 Workshop on Object-Oriented Technology*, pages 54–55, 1998.

[MR04] Radu Marinescu and Daniel Ratiu. Quantifying the Quality of Object-Oriented Design: The Factor-Strategy Model. In *WCRE '04: Proceedings of the 11th Working Conference on Reverse Engineering*, pages 192–201, 2004.

[MRW77] Jim A. McCall, Paul K. Richards, and Gene F. Walters. Factors in Software Quality. Volume I. Concepts and Definitions of Software Quality. Technical Report ADA049014, Nov 1977.

[Mun05] Matthew James Munro. Product Metrics for Automatic Identification of Bad Smell Design Problems in Java Source-Code. *Software Metrics, IEEE International Symposium on*, 0:15, 2005.

[OCL06] UML 2.0 Object Constraint Language Specification. Object Management Group, http://www.omg.org/spec/OCL/2.0/PDF, 2006.

[Ost04] Klaus Ostermann. Bessere Software durch Querschneidende Module. In *Ausgezeichnete Informatikdissertationen 2004*, GI-Edition Lecture Notes in Informatics, 2004.

[OT00] Harold Ossher and Peri Tarr. Multi-Dimensional Separation of Concerns and The Hyperspace Approach. In *Proceedings of the Symposium on Software Architectures and Component Technology: The State of the Art in Software Development*. Kluwer, 2000.

[Par72] David L. Parnas. On the Criteria To Be Used in Decomposing Systems into Modules. *Communications of the ACM*, 15(12):1053–1058, 1972.

[Par78] David L. Parnas. Designing Software for Ease of Extension and Contraction. In *ICSE '78: Proceedings of the 3rd international conference on Software engineering*, pages 264–277, 1978.

[PF01] Steve R. Palmer and Mac Felsing. *A Practical Guide to Feature-Driven Development*. Pearson Education, 2001.

[PG96] Francisco A. C. Pinheiro and Joseph Goguen. An Object-Oriented Tool for Tracing Requirements. *IEEE Software*, 13(2):52–66, 1996.

[Pin04] Francisco A. C. Pinheiro. Requirements traceability. In *Requirements traceability in Perspectives on Software Requirements.* Julio C. S. P. Leite and Jorge Doorn, Kluwer Academic Publishers, pp 91-113, 2004.

[Poh96] K. Pohl. PRO-ART: Enabling Requirements Pre-Traceability. In *Proceedings of the Second International Conference on Requirements Engineering, ICRE*, pages 76–84. IEEE Computer Society, 1996.

[Pra98] Michael Prasse. Evaluation of Object-Oriented Modelling Languages: A Comparison between OML and UML. In *The Unified Modeling Language technical aspects and applications*, pages 58–78. Physica Verlag, 1998.

[Pre97] Roger S. Pressman. *Software Engineering, A Practitionner's Approach.* McGraw-Hill Higher Education, 1997.

[Pre09] Markus Prechtel. *Kennzahlenbasiertes Qualitätsmanagement von Softwareplattformen im Testprozess.* PhD thesis, Universität Ulm, 2009.

[PT93] Colin Potts and Kenji Takahashi. An Active Hypertext Model for System Requirements. In *IWSSD '93: Proceedings of the 7th international workshop on Software specification and design*, pages 62–68, 1993.

[RB09] Matthias Riebisch and Stephan Bode. Software-Evolvability. *Informatik- Spektrum*, 32(4):339–343, 2009.

[RGP86] C. V. Ramamoorthy, V. Garg, and A. Prakash. Programming in the large. *IEEE Transactions on Software Engineering*, 12(7):769–783, 1986.

[RJ01] Balasubramaniam Ramesh and Matthias Jarke. Toward reference models for requirements traceability. *IEEE Transactions on Software Engineering*, 27(1):58–93, 2001.

[RJB04] James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 2nd edition, 2004.

[Roo04] Martin Roock, Stefan und Lippert. *Refactoring in großen Softwareprojekten*. dpunkt.verlag, 1st edition, 2004.

[Rup04] Chris Rupp. *Requirements-Engineering und -Management*. SOPHIST GROUP, Carl Hanser Verlag München Wien, Nürnberg, 3rd edition, 2004.

[Sam97] Johannes Sametinger. *Software Engineering with Reusable Components*. Springer-Verlag New York, Inc., New York, NY, USA, 1997.

[SB03] Régis P. S. Simão and Arnaldo Dias Belchior. Quality Characteristics for Software Components: Hierarchy and Quality Guides. In *Component-Based Software Quality*, pages 184–206, 2003.

[SC03] Nary Subramanian and Lawrence Chung. Process-Oriented Metrics for Software Architecture Evolvability. In *Proceedings Sixth International Workshop on Principles of Software Evolution*, pages 65–70, 2003.

[Sch94] Steffen Schäfer. *Objektorientierte Entwurfsmethoden*. Addison-Wesley, 1994.

[SH10] Melanie Späth and Alexander Hofmann. Qualität sichtbar machen: Ein Erfolgsrezept in moderner Softwareentwicklung. In *Proceedings of Software Engineering 2010 (SE2010)*, 2010.

[Sie05] Johannes Siedersleben. *Moderne Software Architektur, umsichtig planen, robust bauen mit Quasar.* dpunkt.verlag GmbH Heidelberg, 2005.

[SMC79] Wayne Stevens, Glen Myers, and Larry Constantine. Structured design. pages 205–232, 1979.

[Soc06] Periklis Sochos. *The Feature-Architecture Mapping Method for Feature-Oriented Development of Software Product Lines.* PhD thesis, Technical University of Ilmenau, Germany, 2006.

[Som01] Ian Sommerville. *Software engineering.* 6. Auflage Addison-Wesley, Pearson Studium, Boston, MA, USA, 2001.

[SR02] Johannes Sametinger and Matthias Riebisch. Evolution Support by Homogeneously Documenting Patterns, Aspects and Traces. 6th European Conference on Software Maintenance and Reengineering, Budapest, Hungary, Computer Society Press, 2002.

[Sta08a] Michael Stal. Architecture refactoring. Interview with Michael Stal on Architecture Refactoring in OOPSLA 2007, http://www.infoq.com/interviews/michael-stal-on-architecture-refactoring, 2008.

[Sta08b] Gernot Starke. *Effektive Software-Architekturen.* Carl-Hanser Verlag, 2008.

[vdBCH06] Klaas van den Berg, José María Conejero, and Juan Hernández. Analysis of Crosscutting across Software Development Phases based on Traceability. In *EA '06 Proceedings of the 2006 Intl. Workshop on Early Aspects at ICSE*, 2006.

[vK02] Anja von Knethen. A Trace Model for System Requirements Changes on Embedded Systems. In *Proc. of 4th International Workshop on Principles of Software Evolution*, pages 17–26, Vienna, Austria, 2002.

[vKP02] Anja von Knethen and Barbara Paech. A survey on tracing approaches in Practice and Research. IESE-Report 095.01/E, Fraunhofer Institut Experimentelle Software Engineering, Kaiserslautern, Germany, 2002.

[VMD+03] Auri Marcelo Rizzo Vincenzi, José Carlos Maldonado, Márcio Eduardo Delamaro, Edmundo Sérgio Spoto, and W. Eric Wong. Component-

Based Software: An Overview of Testing. In *Component-Based Software Quality*, pages 99–127, 2003.

[Vog07] David Vogler. Design and Implementation of an Extensible Java Framework Based on an Ontology Knowledge Base to Support Software Comprehension. Master's thesis, Brunel University, West London, 2007.

[WBD+10] Stefan Wagner, Manfred Broy, Florian Deißenböck, Miachel Kläs, Peter Liggesmeyer, Jürgen Münch, and Jonathan Streit. Softwarequalitätsmodelle. *Informatik-Spektrum*, 33(1):37–43, 2010.

[Wik10] Wikipedia. Begriff Abstraktion. http://de.wikipedia.org/wiki /Abstraktion, letzter Zugriff: Mar 2010.

[WST97] Nelson Weiderman, Dennis Smith, and Scott Tilley. Approaches to Legacy System Evolution. Technical Report CMU/SEI-97-TR-014, CMU/-SEI, 1997.

[ZK04] Holger Ziekow and Philipp Küfner. Feature Modeling. Technical Report, Technische Universität Berlin, Seminar Entwicklung verteilter eingebetteter Systeme, 2004.

[Zus91] Horst Zuse. *Software Complexity: Measures and Methods.* De Gruyter, Berlin, 1991.

Erklärung

Ich versichere, dass ich die vorliegende Arbeit ohne unzulässige Hilfe Dritter und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe. Die aus anderen Quellen direkt oder indirekt übernommenen Daten und Konzepte sind unter Angabe der Quelle gekennzeichnet.

Weitere Personen waren an der inhaltlich-materiellen Erstellung der vorliegenden Arbeit nicht beteiligt. Insbesondere habe ich hierfür nicht die entgeltliche Hilfe von Vermittlungs- bzw. Beratungsdiensten (Promotionsberater oder anderer Personen) in Anspruch genommen. Niemand hat von mir unmittelbar oder mittelbar geldwerte Leistungen für Arbeiten erhalten, die im Zusammenhang mit dem Inhalt der vorgelegten Dissertation stehen.

Die Arbeit wurde bisher weder im In- noch im Ausland in gleicher oder ähnlicher Form einer Prüfungsbehörde vorgelegt. Ich bin darauf hingewiesen worden, dass die Unrichtigkeit der vorstehenden Erklärung als Täuschungsversuch angesehen wird und den erfolglosen Abbruch des Promotionsverfahrens zu Folge hat.

(Ort, Datum) (Unterschrift)

Thesen

1. Die Qualität eines Software-Systems wird durch Qualitätsdefizite beeinträchtigt.
2. Ein Qualitätsdefizit stellt eine Abweichung zu einem gewünschten Qualitätsmerkmal dar und beeinträchtigt ein Qualitätsziel negativ.
3. Das systematische Erkennen von Qualitätsdefiziten hinsichtlich eines Qualitätsziels, erfordert eine am Qualitätsziel orientierte Analyse, in der das Qualitätsziel in einzelne bewertbare Qualitätsmerkmale zerlegt wird.
4. Das Qualitätsziel Weiterentwicklungsfähigkeit lässt sich durch konkrete Qualitätsteilziele, Qualitätsmerkmale und Qualitätsteilmerkmale zerlegen. Die Zusammenhänge zwischen den ermittelten Qualitätsteilzielen, Qualitätsmerkmalen und Qualitätsteilmerkmalen lassen sich durch einen Zuordnungsgraph dokumentieren.
5. Zur Verbesserung der Weiterentwicklungsfähigkeit sollten neben Code-spezifischen Qualitätsdefiziten, vor allem auch Qualitätsdefizite im Bereich der Spezifikation und Architektur aufgedeckt und mit Hilfe von Reengineering-Aktivitäten behoben werden.
6. Ausgehend von den Ergebnissen der qualitätszielorientierten Analyse, lassen sich Reengineering-Rezepte erstellen, welche eine objektive Erkennung von Qualitätsdefiziten und deren Behebung ermöglichen. Jedem Qualitätsteilmerkmal kann mindestens ein Reengineering-Rezept zugeordnet werden.
7. Durch den regelbasierten Aufbau der Reengineering-Rezepte ist deren Anwendung besser automatisierbar, sodass frühzeitig und damit bereits während des Entwicklungsprozesses Qualitätsdefizite erkannt und behoben werden können.

8. Durch die zielorientierte Methode kann die Anwendung der Reengineering-Rezepte auf ein Software-System gemäß der Prioritäten im Projekt gesteuert werden.