Karlsruher Institut für Technologie

Parametrisierung der Spezifikation von Qualitätsannotationen in Software-Architekturmodellen

Masterarbeit von

Yves R. Schneider

an der Fakultät für Informatik
Architecture-driven Requirements Engineering

Erstgutachter:	Jun.-Prof. Dr.-Ing. Anne Koziolek
Zweitgutachter:	Prof. Dr. Ralf H. Reussner
Betreuender Mitarbeiter:	M. Sc. Axel Busch

01. April 2018 – 30. September 2018

Karlsruher Institut für Technologie
Fakultät für Informatik
Postfach 6980
76128 Karlsruhe

Ich versichere wahrheitsgemäß, die Arbeit selbstständig angefertigt, alle benutzten Hilfsmittel vollständig und genau angegeben und alles kenntlich gemacht zu haben, was aus Arbeiten anderer unverändert oder mit Änderungen entnommen wurde.

Karlsruhe, 30. September 2018

. . . . . .

(Yves R. Schneider)

Kurzfassung

Um qualitativ hochwertige Softwaresysteme zu entwickeln, muss in einem Softwareentwicklungsprozess eine Vielzahl von Qualitätsattributen berücksichtigt werden. Je höher die Komplexität von Softwaresystemen wird, desto wichtiger wird es, die zu erwartende Qualität im Vorfeld zu beurteilen. Jedoch existiert eine Reihe von Qualitätsattributen für Softwaresysteme, welche erst aus den strukturellen Eigenschaften der Softwarekomponenten in diesem Softwaresystem bestimmt werden können. Diese Qualitätsattribute werden in strukturierten und formalisierten Entscheidungsunterstützungsprozessen zur Optimierung der Softwarearchitektur oft nicht genutzt. Einer der Gründe dafür ist, dass dieses Wissen um die Qualitätsattribute einer Softwarekomponente in der Regel nur mit diesen Softwarekomponenten verknüpft ist und nicht mit den strukturellen Eigenschaften eines komponentenbasierten Softwaresystems. So bleibt ein Großteil dieses Wissens unberücksichtigt und kann daher nicht für Kompromissentscheidungen in automatisierten Softwarearchitektur-Optimierungsansätzen genutzt werden.

In dieser Masterarbeit wird ein Rahmenwerk definiert, um Regeln zu spezifizieren zum Transformieren der Qualitätsattribute einer Softwarekomponente in Relation zu ihren strukturellen Eigenschaften in ihrem komponentenbasierten Softwaresystem. Mit diesem Ansatz kann architekturdefiniertes Wissen in Abhängigkeit der Systemarchitektur parametrisiert werden. Hierdurch können die Qualitätsattribute einer Softwarekomponente, welche erst aus den spezifischen Eigenschaften einer konkreten Softwarearchitektur abgeleitet werden können, spezifiziert und so auch ausgewertet werden. Durch diese verbesserten Auswertungen von strukturellen Eigenschaften sollen die Werkzeuge für Softwarearchitekten verbessert werden, sodass diese bessere Entscheidungen in einem Softwareentwicklungsprozess treffen können.

Für die Validierung des Ansatzes werden zwei voneinander unabhängige Fallstudien durchgeführt, um dessen Anwendbarkeit und Nutzen zu zeigen. Zu diesem Zweck wird der Ansatz dieser Masterarbeit sowohl auf eine wissenschaftliche Fallstudie angewandt wie auch auf ein Beispiel, welches sich auf ein reales Industriesystem bezieht. Hiermit wird gezeigt, wie der Ansatz helfen kann, Kompromissentscheidungen über die Softwarearchitektur zwischen mehreren Qualitätsmerkmalen unter der Berücksichtigung der strukturellen Eigenschaften des Softwaresystems zu treffen.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1. Einleitung

Diese Masterarbeit ist wie folgt aufgebaut:

Erstes Kapitel Erläuterung des Beweggrundes und des Ziels sowie der Forschungsfragen dieser Masterarbeit.

Zweites Kapitel Verwandte Arbeiten zum Ansatz dieser Masterarbeit.

Drittes Kapitel Benötigte Grundlagen für die Umsetzung des Ansatzes.

Viertes Kapitel Erklärung des Ansatzes und des daraus resultierenden formalen Modells.

Fünftes Kapitel Experimentelle Validierung des Ansatzes durch zwei Fallstudien.

Sechstes Kapitel Zusammenfassung des Ergebnisses dieser Masterarbeit, Aufzeigen der Grenzen des Ansatzes und Ausblick auf mögliche zukünftige Arbeiten.

1.1. Motivation

Um qualitativ hochwertige Softwaresysteme zu entwickeln, muss in einem Softwareentwicklungsprozess eine Vielzahl von Qualitätsattributen wie beispielsweise Performance, Wartbarkeit und/oder Kosten berücksichtigt werden. Jedoch existieren eine Reihe von Qualitätsattributen für Softwaresysteme, welche erst aus den strukturellen Eigenschaften der Softwarekomponenten in einem komponentenbasierten Softwaresystem bestimmt werden können. Das Wissen um Qualitätsattribute einer Softwarekomponente ist oft nur mit diesen Softwarekomponenten verknüpft und nicht mit den strukturellen Eigenschaften eines komponentenbasierten Softwaresystems. Durch eine solche statische Verknüpfung ist dieses Wissen für jedes Softwaresystem gleich. Da Softwarequalitäts-Analyseverfahren meist auf der Basis von Qualitätsattributen arbeiten, welche mit Softwarekomponenten verknüpft sind, wird Wissen, das erst aus den strukturellen Eigenschaften des komponentenbasierten Softwaresystems abgeleitet werden kann, in der Regel nicht in vollem Umfang betrachtet.

Bislang gibt es andere Ansätze, wie beispielsweise die Service-Effekt-Spezifikation (SEFF), welche für bestimmte quantifizierte Qualitätsattribute die strukturellen Eigenschaften des Softwaresystems beachtet. Die SEFF ist eine Modellierungssprache zur abstrakten Beschreibung der Performance-Eigenschaften der Dienste von Softwarekomponenten [26, S. 6]. Die SEFF erlaubt explizit die Spezifikation von Ressourcenanforderungen und erforderlichen Aufrufen von Diensten in Abhängigkeit von abstrakten Parameter-Charakterisierungen und Nutzungsprofilen. Allerdings ist dieser Ansatz auf die Parametrisierung

von Performance-Eigenschaften beschränkt. Es gibt noch keine Ansätze zur Parametrisierung von nicht-quantifizierten Qualitätsattributen einer Softwarekomponente in Abhängigkeit der Softwarearchitektur.

Aus diesem Grund ist die Motivation des Ansatzes dieser Masterarbeit, ein Rahmenwerk zu schaffen für Regeln zum Transformieren der nicht-quantifizierten Qualitätsattribute einer Softwarekomponente in Relation zu ihren strukturellen Eigenschaften in einem komponentenbasierten Softwaresystem. Unter Verwendung des Ansatzes dieser Masterarbeit kann architekturdefiniertes Wissen in Abhängigkeit der Systemarchitektur parametrisiert werden. Durch diese Parametrisierung kann der Einfluss der Systemarchitektur auf die nicht-quantifizierten Qualitätsattribute der Komponenten modelliert werden, sodass die strukturellen Eigenschaften des Softwaresystems bei Softwarequalitätsanalyseverfahren berücksichtigt werden können. Hierdurch können bestehende Softwarequalitätsanalyseverfahren für komponentenbasierte Softwaresysteme wie gewohnt weiterverwendet werden bei einer gleichzeitigen Verbesserung der Ergebnisse dieser Analyseverfahren, da die Qualitätsattribute einer Softwarekomponente in Abhängigkeit der strukturellen Eigenschaften des Softwaresystems angepasst werden können. Durch diese verbesserten Analyseverfahren der Qualitätsattribute sollen die Werkzeuge eines Softwarearchitekten verbessert werden, sodass diese bessere Entscheidungen in einem Softwarearchitektur-Entwurfs-Prozess treffen können, da nun auch nicht-quantifizierte Qualitätsattribute in Relation zu der Systemarchitektur analysiert werden können.

1.2. Ziele und Nutzen

Das Ziel der Masterarbeit ist es, ein formales Modell zu konzipieren, zu implementieren und zu evaluieren, das zum einen die strukturellen Eigenschaften einer Softwarekomponente in einem komponentenbasierten Softwaresystem modelliert und zum anderen auch die Regeln für die Transformation von Qualitätsattributen dieser Softwarekomponenten modelliert. Hierbei sollen bei der Umsetzung dieses Ziels die folgenden Forschungsfragen beantwortet werden:

- Welche semantischen Klassen von architekturdefinierten Einflussfaktoren auf Qualitätsmodelle gibt es?
- Wie müsste ein formales Modell aussehen, das anhand von Regeln diese Einflussfaktoren modelliert?
- Wie können Qualitätsmodelle gemäß diesen Ergebnissen so verändert werden, dass ihre Aussagen über Qualitätsattribute besser zur gegebenen Systemarchitektur passen?

Einer der großen Nutzen, welcher aus einem solchen formalen Modell gewonnen werden kann, ist, dass Qualitätsattribute von Softwarekomponenten, welche erst aus den spezifischen Eigenschaften einer konkreten Softwarearchitektur abgeleitet werden können, zur weiteren Auswertung spezifiziert werden können. Durch ein solch formales Modell können Regeln für die Transformation von Qualitätsattributen in Abhängigkeit der strukturellen Eigenschaften einer Softwarekomponente in einem komponentenbasierten

Softwaresystem spezifiziert werden. Hierdurch können Analysen der Qualitätsattribute von Softwarekomponenten verbessert werden, indem diese spezifisch auf die strukturellen Eigenschaften eines komponentenbasierten Softwaresystems angepasst werden. Diese Anpassung erlaubt es, die gegebene Systemarchitektur als eine weitere Dimension bei Softwarequalitätsanalyseverfahren für beliebige nicht-quantifizierte Qualitätsattribute zu betrachten. Durch diese modifizierten Analysen der Qualitätsattribute sollen die Werkzeuge der Softwarearchitekten verbessert werden, sodass diese bessere Entscheidungen in einem Softwarearchitektur-Entwurfs-Prozess treffen können. Entscheidungsunterstützende Prozesse gewinnen so neue Dimensionen, welche zur Optimierung verwendet werden können und damit ihren Nutzen erhöhen. Softwarearchitekten erhalten hierdurch neue Einblicke in die Beeinflussung wichtiger Qualitätsattribute durch die Softwarearchitektur, welche aufgrund fehlender Wissensmodellierung und Optimierungsmöglichkeiten bisher nicht sichtbar waren.

1.3. Eigener Beitrag

Der Beitrag dieser Masterarbeit ist ein Rahmenwerk für Regeln zum Transformieren von nicht-quantifizierten Qualitätsattributen einer Softwarekomponente in Relation zu ihren strukturellen Eigenschaften in ihrem Softwaresystem.

Zu diesem Zweck wird der komponentenbasierte Entwicklungsprozess von Koziolek [24] um eine automatisierte Analyse der Struktur der Softwarearchitektur erweitert, um nicht-quantifizierte Qualitätsattribute der Softwarekomponenten in Abhängigkeit zur Softwarearchitektur zu transformieren. Um dies zu unterstützen, wird in dieser Masterarbeit erstmals eine formale, generische, flexible und erweiterbare Formulierung für Transformationen von Qualitätsattributen in Relation zur Struktur der Softwarearchitektur vorgestellt. Hierfür wird ein neuartiges formales Modell vorgestellt zur Transformation von Qualitätsattributen in Relation zur Struktur jeder komponentenbasierten Softwarearchitektur, welche sich mit UML-Komponentendiagrammen darstellen lässt. Dieses formale Modell ist generisch, da es lediglich auf der Grundlage von UML-Komponentendiagrammen arbeitet. Es ist flexibel, da verschiedene strukturelle Eigenschaften für ein System und verschiedene Transformationen von nicht-quantifizierten Qualitätsattributen kombiniert werden können. Es ist erweiterbar, da problemlos neue Definitionen von strukturellen Eigenschaften und Transformationen hinzugefügt werden können.

Eine Instanz dieses formalen Modells wird als Qualitätseffekt-Spezifikation bezeichnet, da diese den Effekt eines Softwaresystems auf ein Qualitätsattribut spezifiziert. Eine solche Qualitätseffekt-Spezifikation beschreibt auf der Grundlage des formalen Modells zum einen strukturelle Eigenschaften in einem Softwaresystem und zum anderen Regeln zum Transformieren von nicht-quantifizierten Qualitätsattributen.

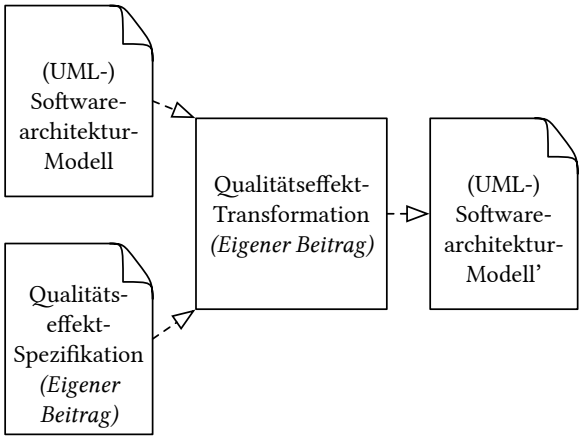

Abbildung 1.1.: Überblick über den rudimentären Vorgang des eigenen Ansatzes

In Abbildung 1.1 ist der rudimentäre Vorgang des eigenen Ansatzes dargestellt. Der Ansatz erhält als Eingabe zum einen ein Modell des komponentenbasierten Softwaresystems und zum anderen eine Qualitätseffekt-Spezifikation. Dieses Modell des Softwaresystems modelliert die Softwarearchitektur mit allen dazugehörigen Softwarekomponenten und ihren Betriebsmitteln. Die Qualitätseffekt-Spezifikation modelliert die Regeln zum Transformieren von nicht-quantifizierten Qualitätsattributen einer Softwarekomponente in Relation zu ihren strukturellen Eigenschaften in ihrem Softwaresystem. Der Ansatz dieser Masterarbeit analysiert die Struktur der gegebenen Softwarearchitektur und transformiert auf Grundlage der Regeln der gegebenen Qualitätseffekt-Spezifikation die nicht-quantifizierten Qualitätsattribute der Softwarekomponenten in der Softwarearchitektur. Das Ergebnis des Ansatzes sind transformierte Qualitätsannotationen in diesem Softwarearchitektur-Modell, welche im übergeordneten komponentenbasierten Entwicklungsprozess weiter wie gewohnt verwendet werden können.

2. Verwandte Arbeiten

Dieses Kapitel gibt einen Überblick über die mit dem Ansatz der Masterarbeit verwandten Arbeiten. Zu diesem Zweck werden die verwandten Arbeiten in fünf sich voneinander unterscheidenden Aspekte unterteilt:

Abschnitt 2.1 Der erste Aspekt ist das Architektur-Wissensmanagement (architecture knowledge management), bei welchem es darum geht, praktische Erfahrungen zu sammeln und diese in allgemeines Architekturwissen umzusetzen, um dieses Wissen in der Kommunikation zu nutzen [4].

Abschnitt 2.2 Der zweite Aspekt sind Architektur-Beschreibungssprachen (architecture description language), welche als ein konzeptionelles Modell zur Beschreibung und Darstellung von Systemarchitekturen verwendet werden.

Abschnitt 2.3 Der dritte Aspekt sind Ansätze zur Softwarearchitektur-Entscheidungsfindung (software architecture decision-making), die Techniken, Werkzeuge und Prozesse beinhalten, welche Softwarearchitekten dabei helfen, bessere Entscheidungen zu treffen [13].

Abschnitt 2.4 Der vierte Aspekt sind die Verbesserungsmethoden für Qualitätsattribute, welche auf Grundlage von Modellen einer Softwarearchitektur Vorhersagen und Verbesserungen für Qualitätsmerkmale ermöglichen.

Abschnitt 2.5 Der fünfte Aspekt sind domänenspezifische Sprachen auf Unified-Modeling-Language-Modellen, welche anwendungsspezifische formale Sprachen auf Basis von oder für die UML anbieten.

2.1. Architektur-Wissensmanagement

In einer vergleichenden Studie über Architektur-Wissensmanagement-Werkzeuge [46] haben Antony Tang u. a. fünf Architektur-Wissensmanagement-Werkzeuge und deren Unterstützung im Architektur-Lebenszyklus verglichen: ADDSS [11], Archium [20], AREL [45], Knowledge architect [21] und PAKME [3]. Der Vergleich basiert auf einem Bewertungsrahmen, welcher durch eine Menge von zehn Kriterien definiert ist.

Die Ergebnisse des Vergleichs haben gezeigt, dass sich diese fünf architektonischen Wissenswerkzeuge auf Argumentation und Kontextwissen konzentrieren, einige Werkzeuge auf Entwurfswissen, andere auf Allgemeinwissen. Der Unterschied in der Schwerpunktsetzung hebt die beiden Schlüsselaspekte des Fachwissens hervor: Die Unterstützung architektonischer Entwurfstätigkeiten und die Unterstützung der Wiederverwendung von Wissen. Die Nachvollziehbarkeit von Entwurfsentscheidungen wird von allen fünf

Architektur-Wissensmanagement-Werkzeugen gut unterstützt, jedoch die Unterstützung bei der Wissensintegration und Wissensanpassung ist begrenzt, was das Interesse dieses Ansatzes der Masterarbeit ist.

Auf Grundlage eines dieser fünf Architektur-Wissensmanagement-Werkzeuge wäre die Umsetzung einer Anpassung von Qualitätsattributen einer Softwarekomponente, unter Beachtung der strukturellen Eigenschaften eines komponentenbasierten Softwaresystems, nicht einfach umsetzbar.

In ihrer systematischen Literaturübersicht über Softwarearchitektur-Wissensmanagement-Ansätze und deren Unterstützung für Wissensmanagement-Aktivitäten [49] haben Rainer Weinreich und Iris Groher 56 verschiedene Ansätze für das Softwarearchitektur-Wissensmanagement auf der Grundlage von 115 Studien identifiziert. Sie analysierten dabei auch jeden dieser Ansätze im Hinblick auf seine Fokussierung und Unterstützung für wichtige Wissensmanagement-Aktivitäten und den Grad der Evidenz für jede unterstützte Aktivität. Hierbei kamen sie zu dem Ergebnis, dass sich die meisten der entwickelten Ansätze auf die Nutzung von bereits erfasstem Wissen konzentrieren, was auch die am besten validierte Aktivität dieser 56 Ansätze ist.

Jedoch wird das Problem der effizienten Erfassung immer noch nicht ausreichend angegangen und nur wenige Ansätze befassen sich speziell mit der Wiederverwendung, dem Austausch und vor allem mit der Wartung. Auch existieren nur zwei Ansätze in dieser Literaturübersicht, welche Transformationen unterstützen, um das dokumentierte Wissen konsistent zu halten. Das Architectural Design Decision Support Framework (ADvISE) [29] unterstützt, durch Umwandlung von wiederverwendbarem Wissen in Entwurfsansichten, die Wartung von Abbildungen zwischen Entscheidungen und Entwürfen. Der Ansatz für den Erwerb einer guten Softwarearchitektur (AQUA) [12] transformiert die Architektur durch veränderte Entwurfsentscheidungen. Dieser Umstand zeigt, dass die Notwendigkeit weiterer Untersuchungen bei Softwarearchitektur-Wissensmanagement-Ansätzen von 56 verschiedenen Ansätzen auf diese beiden Ansätze eingegrenzt werden kann.

Das Architectural Design Decision Support Framework (ADvISE) [27] dient der Modellierung von wiederverwendbaren Architekturentscheidungen unter Berücksichtigung von verschiedenen Abstraktionsebenen, das heißt sowohl auf High-Level- als auch auf technologie- und auch domänenspezifischen Ebenen. Der ADvISE Ansatz ermöglicht es, den Entscheidungsprozess für wiederkehrende architektonische Entscheidungen halbautomatisch zu gestalten und die Entwurfsentscheidungen zu dokumentieren. Um hierbei architektonische Entscheidungen und Entwürfe konsistent und nachvollziehbar zu halten, verwendet ADvISE formale Verbindungen zwischen den wiederverwendbaren architektonischen Entscheidungen und neuen architektonischen Entwürfen.

Der Kern dieser formalen Verbindungen beinhaltet grundlegende Aktionen, mit welchen einzelne Elemente der Architekturmodelle erstellt oder geändert werden können, wie zum Beispiel das Anlegen einer neuen Komponente, das Löschen eines bestehenden Konnektors oder das Aktualisieren eines Ports [29]. Hierbei werden jedoch nur die strukturellen Eigenschaften der einzelnen Elemente des Architekturmodelles auf Grundlage von architektonischen Entwurfsentscheidungen transformiert. Es werden demgegenüber keine Qualitätsattribute der einzelnen Elemente des Architekturmodelles, unter Beachtung

der strukturellen Eigenschaften des Architekturmodelles, erstellt oder geändert. Diese Transformationen für Qualitätsattribute der einzelnen Komponenten ist das Interesse des Ansatzes dieser Masterarbeit.

Heeseok Choi u. a. schlagen in [12] einen integrierten Ansatz für den Erwerb einer guten Softwarearchitektur im Hinblick auf ihre Anforderungen vor, dieser wird als AQUA bezeichnet. Dieser Ansatz definiert den entscheidungszentrierten Prozess der Entscheidungsfindung, -bewertung und -änderung. Dabei spielt die Transformation der Softwarearchitektur bei sich ändernden architektonischen Entwurfsentscheidungen eine Rolle. Hierbei gliedert sich diese Transformationen in vier Phasen:

1. Identifizierung von Transformationsanforderungen
2. Festlegung der Transformationsstrategie
3. Transformation der Architektur
4. Überprüfung der Transformationsergebnisse

Dabei werden die zu ändernden Entscheidungen identifiziert und die Zusammenhänge zwischen den Entscheidungen analysiert. Anschließend werden die für die erforderlichen Änderungen relevanten Konstruktionen rekonstruiert und die Ergebnisse validiert.

Im AQUA-Ansatz spielt die architektonische Transformation eine wichtige Rolle bei der Reduzierung von Fehlern in der Architektur oder bei Änderungen an der Architektur bei sich ändernden Entscheidungen. Jedoch transformieren diese keine Qualitätsattribute von Softwarekomponenten unter Beachtung der strukturellen Eigenschaften der Softwarearchitektur, was der Ansatz dieser Masterarbeit ist.

2.2. Architektur-Beschreibungssprache

Die Architektur-Beschreibungssprache Acme [19] von David Garlan u. a. bietet einen strukturellen Rahmen für die Charakterisierung von Architekturen zusammen mit Annotationsmöglichkeiten für zusätzliche Architektur-Beschreibungssprachen spezifischer Informationen. Die architektonische Struktur wird in Acme mit sieben Kerntypen von Entitäten definiert: Komponenten, Konnektoren, Systeme, Ports, Rollen, Repräsentationen und Rep-Maps [18]. Dies erlaubt es, mit Acme die Zusammensetzungen von Komponenten genau zu beschreiben und auch die Art und Weise wie diese Komponenten kommunizieren zu verdeutlichen. Hierbei unterstützt Acme die Verwendung von mehreren Komponenten-Schnittstellen und auch die hierarchische Beschreibung und Kapselung von Subsystemen als Komponente in einem größeren System. Ebenfalls unterstützt Acme die Spezifikation von nicht-funktionalen Eigenschaften und bietet ein explizites Umfeld für die Beschreibung der detaillierten Semantik der Kommunikationsinfrastruktur.

Die Architektur-Beschreibungssprache Acme bietet einen formalen Rahmen zur Spezifikation der strukturellen Eigenschaften einer Softwarearchitektur sowie von Qualitätsattributen der Softwarekomponenten. Jedoch ist bei diesem Ansatz das Wissen um die

Qualitätsattribute einer Softwarekomponente nur mit eben dieser Softwarekomponente verknüpft und nicht mit den strukturellen Eigenschaften des komponentenbasierten Softwaresystems. Durch diese statische Verknüpfung ist dieses Wissen um die Qualitätsattribute einer Softwarekomponente für jedes Softwaresystem gleich.

Die Architektur-Beschreibungssprache DAOP-ADL [37] von Mónica Pinto u. a. ist eine komponenten- und aspektbasierte Sprache, um die Architektur einer Anwendung in Bezug auf Komponenten, Aspekte und eine Reihe von Kompatibilitätsregeln zwischen diesen zu spezifizieren. Das Ziel der DAOP-ADL ist es, die Spezifikation der Anwendungsarchitektur mit der Implementierung zu verbinden, um somit Softwarearchitekturen zu beschreiben und zu validieren, welche dem Komponenten-Aspektmodell der DAOP-Plattform [36] entsprechen. Der Einsatz von DAOP-ADL schließt die Lücke zwischen Entwurf und Implementierung von komponenten- und aspektbasierten Anwendungen.

Die DAOP-ADL beinhaltet die Beschreibung der Schnittstellen von Komponenten und Aspekten, sowie der Beschreibung ihrer Beziehungen. An dieser Stelle wird eine Komponente lediglich über eine Reihe von Schnittstellen spezifiziert. Bei dieser Reduzierung auf die funktionalen Eigenschaften einer Komponente werden jedoch andere Qualitätsattribute nicht beachtet.

Die Architekturanalyse und Entwurfssprache (AADL) [16] ist eine von Peter H. Feiler u. a. entwickelte Modellierungssprache, welche durch eine erweiterbare Notation, ein Rahmenwerk und eine genau definierte Semantik eine frühzeitige und wiederholte Analyse der Systemarchitektur hinsichtlich performancekritischer Eigenschaften unterstützt. Zu diesem Zweck verwendet AADL formale Modellierungskonzepte zur Beschreibung und Analyse von Anwendungssystemarchitekturen im Hinblick auf einzelne Komponenten und deren Wechselwirkungen. Dabei umfasst AADL Abstraktionen von Software, Rechnerhardware und Systemkomponenten zur Spezifikation und Analyse von eingebetteten und hochzuverlässigen Echtzeit-Systemen, komplexen Systemen und spezialisierten Systemen zur Leistungsfähigkeit und Abbildung von Software auf Rechnerhardware-Elemente.

Die AADL ist hoch spezialisiert auf die modellbasierte Performance-Analyse und Performance-Spezifikation komplexer Echtzeit-Embedded-Systeme. Jedoch beinhaltet die AADL keinen Ansatz zur Modellierung von nicht-quantifizierten Qualitätsmerkmalen, was das Interesse dieser Masterarbeit ist.

2.3. Softwarearchitektur-Entscheidungsfindung

Muhammad Ali Babar und Ian Gorton beschreiben in [2] einen Ansatz für die Verwaltung von architektonischem Wissen und Entwurfsbegründung. Dieser Ansatz wurde entwickelt, um ein Rahmenwerk zur Erfassung und Nutzung von Architekturwissen zur Verbesserung des Architekturprozesses zu unterstützen. Dieses Rahmenwerk besteht aus Techniken zur Erfassung von Entwurfsentscheidungen und Kontextinformationen, einem Ansatz zur Gewinnung und Dokumentation von Architekturwissen aus Mustern und einem Datenmodell zur Charakterisierung von Architekturkonstrukten, deren Eigenschaften und Beziehungen. Zur Unterstützung dieses Rahmenwerks wurde ein webbasiertes Tool namens PAKME (Process-centric Architecture Knowledge Management Environment)

entwickelt. PAKME ist auch als Wissensquelle für diejenigen gedacht, die schnellen Zugriff auf erfahrungsbasierte Entwurfsentscheidungen benötigen, um neue Entscheidungen zu treffen oder die Gründe für frühere Entscheidungen zu entdecken.

PAKME unterstützt jedoch keine schematische Modellierung von Entwurfsentscheidungen, sondern konzentriert sich auf die Bereitstellung eines „Handbuchs“ über Architekturwissen. Aus diesem Grund unterstützt PAKME keine Regeln zum Transformieren von nicht-quantifizierten Qualitätsattributen einer Softwarekomponente in Relation zu ihren strukturellen Eigenschaften in ihrem Softwaresystem.

Svahnberg und Wohlin stellen in [44] eine empirische Untersuchung eines Prozesses vor, welche eine Quantifizierung der wahrgenommenen Unterstützung verschiedener Softwarearchitekturen für unterschiedliche Qualitätsmerkmale ermöglicht. Dieser Prozesse ist hierbei der sogenannte analytische Hierarchieprozess (AHP) [51], welcher ein Verfahren zum paarweisen Vergleich ist. Der analytische Hierarchieprozess ermöglicht es, durch alle möglichen paarweisen Vergleiche eine Ordnung der Entitäten zu erstellen und zusätzlich einen Konsistenzindex zu berechnen. Dies erlaubt eine Paarbewertung von Architekturkandidaten und ermöglicht hierdurch eine fundierte Entscheidung darüber, welcher Architekturkandidat am besten zu der von diesem zu entwerfenden System geforderten Mischung von Qualitätsmerkmalen passt. Unter Verwendung mehrerer Architekturkandidaten und Qualitätsanforderungen kann der analytische Hierarchieprozess genutzt werden, um den besten Architekturkandidaten entsprechend den Anforderungen zu identifizieren.

Der AHP-Ansatz ist jedoch ein manueller Prozess, welcher eine manuelle Navigation im Entwurfsraum erfordert; während der Ansatz dieser Masterarbeit auch eine automatisierte Suche nach optimalen Architekturen in einem automatisierten Entscheidungsunterstützungsansatz ermöglichen sollte. Daher ist der AHP-Ansatz bei der Bewertung vieler Architekturkandidaten vergleichsweise zeitaufwendig.

Rick Kazman u. a. stellen in [22] die qualitative Methode zur architektonischen Kompromissanalyse ATAM vor. Die Kompromissanalyse ATAM ist eine strukturierte Technik zum Verständnis der in den Architekturen softwareintensiver Systeme enthaltenen Kompromisse. Diese Methode wurde entwickelt, um die Eignung einer Softwarearchitektur im Hinblick auf mehrere konkurrierende Qualitätsmerkmale prinzipiell zu bewerten. Diese Qualitätsmerkmale interagieren und die Methode hilft dabei, über architektonische Entscheidungen nachzudenken, welche sich auf die Interaktionen von Qualitätsmerkmalen auswirken. Die Kompromissanalyse ATAM ist ein spiralförmiges Modell: zunächst kommt eine Postulierung von Kandidatenarchitekturen, gefolgt von Analyse und Risikominimierung, was zu verfeinerten Architekturen führt.

Der ATAM-Ansatz ist jedoch ein manueller Prozess, bei welchem die Qualität auf ihr Verhältnis von Kosten und Nutzen heruntergebrochen werden muss. Auch basiert der Ansatz auf eine manuelle Navigation im Entwurfsraum, während der Ansatz dieser Masterarbeit auch eine automatisierte Suche nach optimalen Architekturen in einem automatisierten Entscheidungsunterstützungsansatz ermöglichen sollte. Daher ist der ATAM-Ansatz bei der Bewertung vieler Architekturkandidaten vergleichsweise zeitaufwendig.

2.4. Verbesserungsmethoden für Qualitätsattribute

Felix Bachmann u. a. stellen einen Entwurfsassistenten namens ArchE [5] vor, um die Softwarearchitektur in einem frühen Stadium zu evaluieren, aber auch um Verbesserungen für diese zu planen. Dieser Ansatz fördert unter anderem die Verwendung von Qualitätsattributsmodellen bei dem Entwurf von Softwarearchitektur. Hierbei beruht der ArchE-Ansatz auf einer Sammlung von Argumentationsrahmen, welche jeweils auf ein einziges Qualitätsmerkmal spezialisiert sind, aber auch bei der Erstellung und Analyse von Architekturentwürfen zusammenarbeiten. Der Entwurfsassistent ArchE stellt basierend auf diesen Argumentationsrahmen eine Infrastruktur für Drittforscher bereit, damit diese ihre eigenen Qualitätsattributsmodelle integrieren können. Diese Infrastruktur zielt darauf ab, das Experimentieren und den Austausch von qualitativ hochwertigem Wissen in Forschung und Lehre zu erleichtern.

An dieser Stelle soll ArchE keine umfassende oder optimale Suche im Entwurfsraum durchführen, sondern ArchE ist ein Assistent des Softwarearchitekten, welcher in diesem Entwurfsraum „gute Wege“ aufzeigen kann [14]. Auch ist das zugrunde liegende Architekturmodell nicht komponentenbasiert, daher lassen sich Qualitätsmerkmale von Softwarekomponenten nicht ohne weiteres identifizieren und die Relation zu ihren strukturellen Eigenschaften in ihrem komponentenbasierten Softwaresystem nicht umsetzten.

Aldeida Aleti u. a. stellen mit ArcheOpterix [1] einen erweiterbaren Eclipse-basierten Ansatz vor, welche einen Rahmen für die Implementierung von Evaluierungstechniken und Optimierungsheuristiken für AADL-Spezifikationen bietet. Zu diesem Zweck wurden evolutionäre Strategien implementiert, um optimierte Verteilungen von Softwarekomponenten im Hinblick auf mehrere Qualitätsziele und Entwurfsbeschränkungen zu identifizieren.

Damit beschränkt sich das ArcheOpterix auf die untersuchten Verteilungsprobleme. Es gibt noch keine Unterstützung für Freiheitsgrade, welche bei mehreren Problemen auftreten. Auch bietet ArcheOpterix noch keine Möglichkeit, die Eigenschaften der Architektur zu modellieren, sodass nicht-quantifizierte Qualitätsmerkmale von Softwarekomponenten ohne weiteres berücksichtigt werden können.

2.5. Domänenspezifische Sprachen auf UML-Modellen

Die Object Constraint Language (OCL) ist eine formale Sprache zur Beschreibung von Ausdrücken in UML-Modellen. Diese Ausdrücke geben entweder invariante Bedingungen an, welche für das zu modellierende System gelten müssen, oder sie sind Abfragen über die in einem UML-Modell beschriebenen Objekte [48]. Ein UML-Modell kann in der Regel nicht ausreichend verfeinert werden und somit nicht alle relevanten Aspekte einer Spezifikation bereitstellen. In der Regel müssen zusätzliche Einschränkungen für die Objekte im Modell beschrieben werden. Hierfür kann OCL verwendet werden, um anwendungsspezifische Einschränkungen in UML-Modellen anzugeben. Da ein OCL eine reine Spezifikationssprache ist, hat ein OCL-Ausdruck garantiert auch keine Nebenwirkungen auf ein UML-Modell.

Wenn ein OCL-Ausdruck ausgewertet wird, gibt dieser lediglich einen Wert zurück, er kann jedoch nichts am Modell ändern [33]. Dies bedeutet, dass sich der Zustand des

Systems aufgrund der Auswertung eines OCL-Ausdrucks niemals ändert, obwohl ein OCL-Ausdruck zur Angabe einer Zustandsänderung verwendet werden kann. Da OCL keine Programmiersprache ist, ist es nicht möglich, Programmlogik oder Ablaufsteuerung in OCL zu schreiben. Folglich kann innerhalb eines OCL-Ausdrucks kein Prozess aufgerufen oder Vorgänge ohne Abfrage gestartet werden. Dies würde es zwar erlauben, die strukturellen Eigenschaften der Softwarearchitektur mit OCL abzubilden, jedoch würden sich mit OCL nicht die Transformationen der Qualitätsattribute von Softwarekomponenten umsetzen lassen.

In ihrem Ansatz vom Analysemodell zur Softwarearchitektur [35] stellen Jorge Enrique Pérez-Martínez und Almudena Sierra-Alonso eine Methode zur Ableitung der Softwarearchitektur eines Systems aus seinem Analysemodell dar. Zu diesem Zweck definieren sie Modell-zu-Modell-Transformationen, um Komponenten-Architekturmodelle aus Klassen- und Paketanalysemodellen unter Verwendung von OCL-Produktionsregeln zu generieren. Diese Produktionsregeln bestehen aus einer festgelegten Menge von Regeln, welche über das UML-Metamodell des Analysemodells funktionieren.

Die in diesem Ansatz vorgeschlagenen Transformationen generieren aus einem Analysemodell jedoch nur eine Architektur im sogenannten C2-Stil [30]. Es können hierbei keine anderen Architekturstile, wie beispielsweise Client-Server, Peer-to-Peer oder Pipes und Filter, generiert werden. Auch werden keine Annotationen zu Qualitätsattributen aus dem Analysemodell und der Softwarearchitektur transformiert.

In ihrem Ansatz zur Harmonisierung von Architekturentscheidungen mit komponentenbasierten Architekturmodellen unter Verwendung von wiederverwendbaren architektonischen Wissenstransformationen und Einschränkungen [28] führen Ioanna Lytra u. a. eine Sprache zur Transformation von Architekturwissen ein, um wiederverwendbare Aktionen zu spezifizieren, welche zur automatischen Erstellung oder Aktualisierung der zugrunde liegenden komponentenbasierten Architekturmodellen im Hinblick auf bestimmte architektonische Entwurfsentscheidungen erforderlich sind.

Dieses Transformationssprache bietet grundlegende Aktionen zur Aktualisierung einzelner Modellelemente, wie beispielsweise Komponenten, Konnektoren, Ports, Eigenschaften und Stereotypen. Hierdurch können zwar grundlegende Transformationen für die strukturellen Eigenschaften von Komponenten in einem komponentenbasierten Softwaresystem beschrieben werden, jedoch können mit dieser Transformationssprache keine Transformationen für das Hinzufügen und Verändern von Qualitätsattributen in Abhängigkeit der strukturellen Eigenschaften einer Softwarekomponente in einem komponentenbasierten Softwaresystem spezifiziert werden.

3. Grundlagen

Die folgenden Unterabschnitte dieses Kapitels geben einen Überblick der wichtigen Grundlagen und Technologien des Forschungsfeldes dieser Masterarbeit. Auch auf die für den Ansatz der Masterarbeit relevant eingestuften Grundlagen und Technologien wird genauer eingegangen.

3.1. Komponentenbasierte Softwareentwicklung

Die Kernidee der komponentenbasierten Softwareentwicklung ist nicht nur die Beschleunigung der Entwicklung neuer Softwaresysteme durch die Wiederverwendung von bereits erprobter Software, sondern auch die Verbesserung der Qualität dieser neu entwickelten Systeme. Bei der komponentenbasierten Softwareentwicklung werden einzelne miteinander kombinierbare Softwarebausteine zusammen gekapselt, sodass diese Komponenten leicht von Dritten verwendet werden können, ohne die einzelnen wiederverwendbaren Softwarekomponenten verstehen zu müssen. Im Handbuch der Softwarearchitektur wird eine Softwarekomponente wie folgt definiert:

„Komponenten sind modulare Teile eines Systems, die ihren Inhalt und somit komplexes Verhalten transparent kapseln und in ihrer Umgebung als austauschbare Einheiten mit klar definierten Schnittstellen auftauchen." ([39, S. 45])

Über diese klar definierten Schnittstellen kann eine einzelne Softwarekomponente zum einen Dienste anfordern und/oder zum anderen auch selbst Dienste bereitstellen. Wenn die Schnittstellen für den angeforderten oder bereitgestellten Dienst miteinander kompatibel sind, kann diese Softwarekomponente durch eine andere Softwarekomponente ersetzt werden. Aus dem Verbund von einzelnen Softwarekomponenten kann so wieder ein vollständiges neues komponentenbasiertes Softwaresystem erstellt werden.

3.2. Unified Modeling Language

Die Unified Modeling Language (UML) ist eine Modellierungssprache mit dem Ziel, Systemarchitekten und Softwareentwicklern Werkzeuge zur Analyse, Entwurf und zur Implementierung softwarebasierter Systeme sowie zur Modellierung von Prozessen zur Verfügung zu stellen. Hierzu gehört auch eine Spezifikation von visuell lesbaren Notationselementen zur Darstellung der einzelnen UML-Modellierungskonzepte sowie von Regeln zu der Zusammenführung von verschiedenen Diagrammtypen, welche unterschiedlichen Aspekten modellierter Systeme entsprechen [34, S. 1].

Zu diesem Zweck spezifiziert die UML mit dem Komponentendiagramm eine Menge von Konstrukten, mit welchen Softwaresysteme beliebiger Größe und Komplexität definiert

werden können [34, S. 208]. Hinzukommend spezifiziert die UML mit dem Verteilungsdiagramm eine Menge von Konstrukten, mit welchen die Konfiguration der Laufzeitverarbeitungsknoten und der darauf liegenden Komponenteninstanzen und Objekte gezeigt werden [40, S. 252].

3.2.1. UML-Komponentendiagramm

UML-Komponentendiagramme zeigen die Organisationen und Abhängigkeiten zwischen Softwarekomponenten. UML-Komponentendiagramme spezifizieren eine Komponente als modulare Einheit mit klar definierten Schnittstellen, die innerhalb ihrer Umgebung austauschbar sind [34, S. 208]. Da ein UML-Komponentendiagramm nur eine Beschreibungsform und kein Instanzform hat, muss, um Komponenteninstanzen anzuzeigen, ein UML-Verteilungsdiagramm verwendet werden. Die Spezifikationen für die formale Semantik und Notation der einzelnen Hauptkonzepte für UML-Komponentendiagramme sind in Tabelle 3.1 aufgelistet.

Hauptkonzept	Spezifikation
Komponente	[34, S. 208], [40, S. 216]
Schnittstelle	[34, S. 170], [40, S. 310]
Abhängigkeit	[34, S. 37], [40, S. 250]

Tabelle 3.1.: Spezifikationen der Hauptkonzepte von UML-Komponentendiagrammen

3.2.1.1. UML: Komponente

Die UML spezifiziert einen Satz von Konstrukten, mit welchen Softwaresysteme beliebiger Größe und Komplexität definiert werden können. Die nachfolgenden UML-spezifischen Grundlagen sind [34, 208 ff.] entnommen.

Die UML spezifiziert eine Komponente als modulare Einheit mit klar definierten Schnittstellen, welche innerhalb ihrer Umgebung austauschbar ist. Das UML-Komponentenkonzept adressiert den Bereich der komponentenbasierten Entwicklung und komponentenbasierten Systemstrukturierung, bei denen eine Komponente über den gesamten Entwicklungslebenszyklus modelliert und sukzessive in Bereitstellung und Laufzeit verfeinert wird.

Eine Komponente stellt einen modularen Teil eines Systems dar, welcher seinen Inhalt kapselt und dessen Manifestation innerhalb seiner Umgebung austauschbar ist. Eine Komponente ist eine in sich geschlossene Einheit, welche den Zustand und das Verhalten mehrerer Klassifikatoren kapselt. Eine Komponente ist eine ersetzbare Einheit, welche zur Entwurfszeit oder zur Laufzeit durch eine Komponente ersetzt werden kann, die aufgrund der Kompatibilität ihrer Schnittstellen eine gleichwertige Funktionalität bietet.

Abbildung 3.1 zeigt, mit einem UML-Klassendiagramm, die abstrakte Syntax der Spezifikation für eine Komponente in der UML.

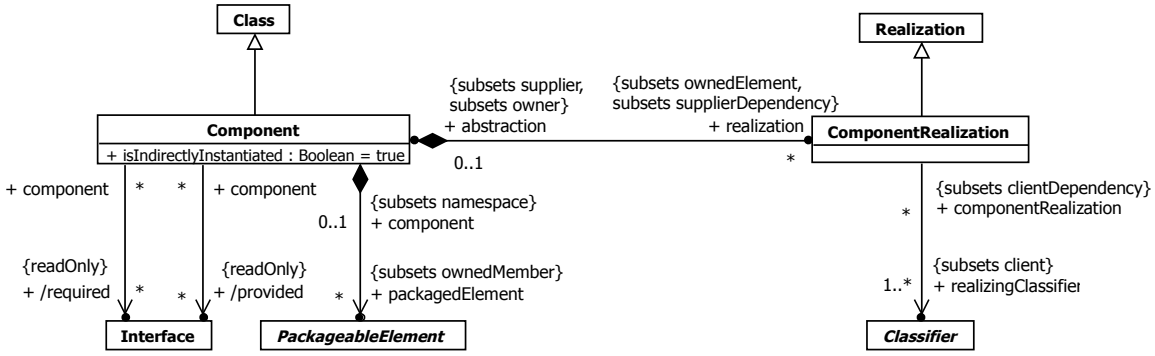

Abbildung 3.1.: Abstrakte Syntax der Spezifikation der UML-Komponente [34, S. 209]

3.2.1.2. UML: Schnittstelle

Eine UML-Schnittstelle ist eine Art Klassifikator, welcher eine Deklaration einer Reihe von öffentlichen Merkmalen und Pflichten darstellt, die zusammen einen kohärenten Dienst darstellen. Die nachfolgenden UML-spezifischen Grundlagen sind [34, S. 170] entnommen.

Eine Schnittstelle spezifiziert einen Vertrag; jede Instanz eines Klassifikators, welche die Schnittstelle realisiert, muss diesen Vertrag erfüllen. Die mit einer Schnittstelle verbundenen Verpflichtungen bestehen in Form von Einschränkungen, beziehungsweise Vor- und Nachbedingungen, oder Protokollspezifikationen, welche die Interaktionen über die Schnittstelle einschränken können.

Schnittstellen dürfen nicht instanziiert werden. Stattdessen wird eine Schnittstellenspezifikation durch einen „BehavioredClassifier“ implementiert oder realisiert, was bedeutet, dass der „BehavioredClassifier“ eine öffentliche Fassade darstellt, welche der Schnittstellenspezifikation entspricht.

Abbildung 3.2 zeigt, mit einem UML-Klassendiagramm, die abstrakte Syntax der Spezifikation für eine Schnittstelle in der UML.

3.2.1.3. UML: Abhängigkeit

Eine UML-Abhängigkeit bezeichnet eine so genannte Geber-Nehmer-Beziehung zwischen UML-Modellelementen, bei welcher die Änderung eines Gebers Auswirkungen auf die Modellelemente des Nehmers haben kann. Die nachfolgenden UML-spezifischen Grundlagen sind [34, 37 ff.] entnommen.

Eine Abhängigkeit bedeutet, dass die Semantik der Nehmer ohne die Geber nicht vollständig ist. Das Vorhandensein von Abhängigkeitsbeziehungen in einem UML-Modell hat keine semantischen Auswirkungen auf die Laufzeit. Die Semantik wird in Bezug auf die Komponenten gegeben, welche an der Beziehung teilnehmen, nicht jedoch in Bezug auf ihre Instanzen.

Eine Verwendung ist eine Abhängigkeit, in welcher eine Komponente eine andere Komponente oder einen Satz von Komponenten für ihre vollständige Implementierung oder ihren Betrieb benötigt. Die Verwendung legt nicht fest, wie der Nehmer den Geber

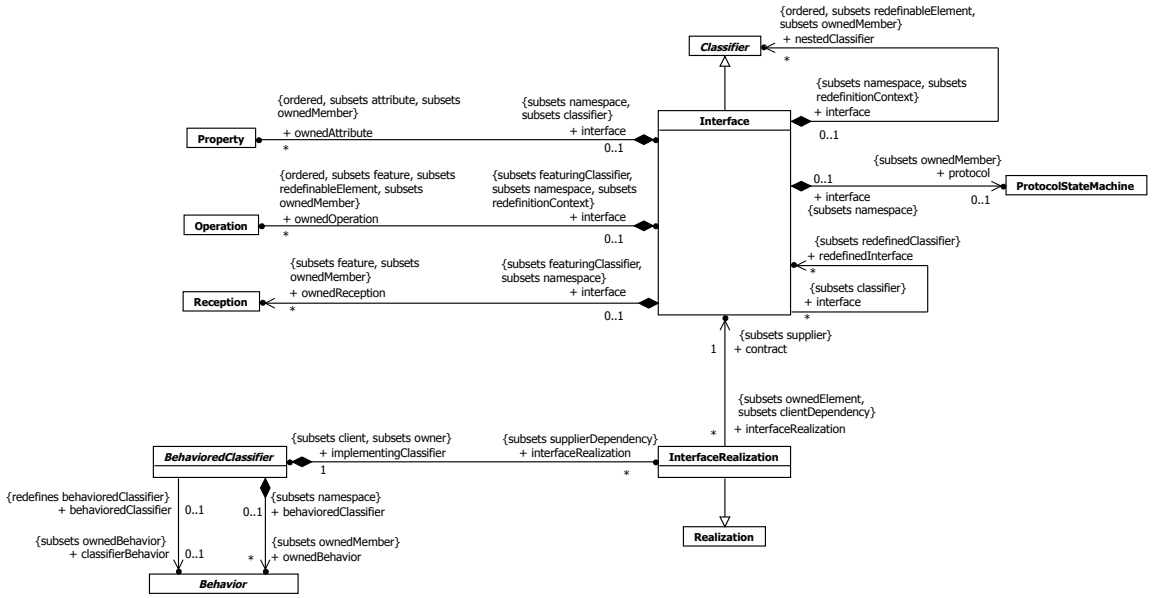

Abbildung 3.2.: Abstrakte Syntax der Spezifikation der UML-Schnittstelle [34, S. 170]

verwendet, außer dass der Geber bei der Definition oder Implementierung des Nehmers verwendet wird.

Abbildung 3.3 zeigt, mit einem UML-Klassendiagramm, die abstrakte Syntax der Spezifikation der „Abhängigkeit“ in der UML.

3.2.2. UML-Verteilungsdiagramm

UML-Verteilungsdiagramme erfassen die Beziehung zwischen einem bestimmten konzeptionellen oder physischen Element eines modellierten Systems und den ihm zugeordneten Informationsressourcen [34, S. 653]. In diesem Kontext kann ein physisches Element unter anderem physische Geräte, wie beispielsweise Maschinenkomponenten oder Rechenressource mit Verarbeitungsfähigkeit [34, S. 658], oder physische Komponenten, wie beispielsweise EJB-, CORBA- oder WSDL-Komponenten [34, S. 209], sein. UML-Verteilungsdiagramme zeigen die Konfiguration der Laufzeitverarbeitungsknoten und der Komponenteninstanzen. Hierbei zeigen UML-Verteilungsdiagramme die Instanzen von Komponenten, während ein UML-Komponentendiagramm nur die Definition der Komponenten selbst zeigt [40, S. 252]. Die Spezifikationen für die formale Semantik und Notation der einzelnen Hauptkonzepte für UML Verteilungsdiagramme sind in Tabelle 3.2 aufgelistet.

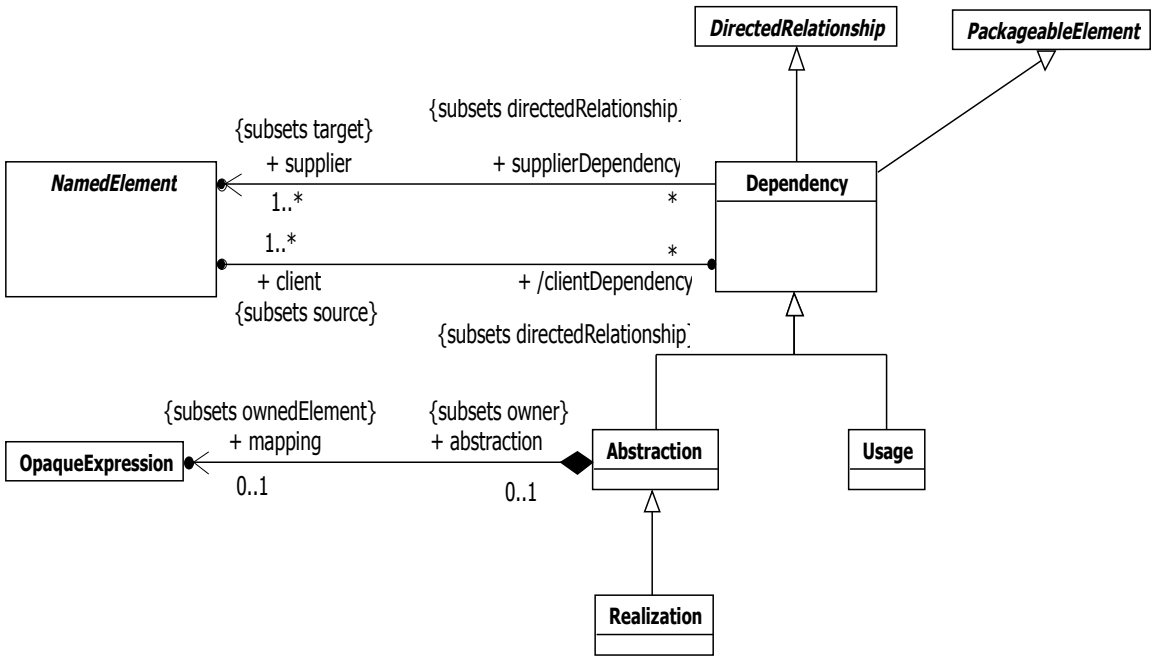

Abbildung 3.3.: Abstrakte Syntax der Spezifikation der UML-Abhängigkeit [34, S. 38]

Hauptkonzept	Spezifikation
Knoten	[34, S. 657], [40, S. 357]
Bereitstellung	[34, S. 653], [40, S. 330]

Tabelle 3.2.: Spezifikationen der Hauptkonzepte von UML-Verteilungsdiagrammen

3.2.2.1. UML: Knoten

Ein UML-Knoten ist ein Rechenbetriebsmittel, auf welchem Artefakte über Bereitstellungsbeziehungen zur Ausführung bereitgestellt werden können. Normalerweise repräsentieren Knoten entweder Hardware-Geräte oder Software-Ausführungsumgebungen. Die nachfolgenden UML-spezifischen Grundlagen sind [34, 657 ff.] entnommen.

UML-Knoten erarbeiten und verfeinern den abstrakten Begriff des „DeploymentTargets“. Sie können verschachtelt und über Kommunikationswege zu beliebig komplexen Systemen verbunden werden. Für fortgeschrittene Modellierungsanwendungen können Knoten eine komplexe interne Struktur haben, welche durch Verschachtelung definiert und miteinander verbunden werden kann, um bestimmte Situationen darzustellen. Die interne Struktur von Knoten kann nur aus anderen Knoten bestehen. Neben der Teilnahme an Bereitstellungen erwerben Knoten eine Reihe von zugehörigen Elementen, welche aus den Manifestationsbeziehungen der auf ihnen eingesetzten Artefakte abgeleitet sind.

Abbildung 3.4 zeigt, mit einem UML-Klassendiagramm, die abstrakte Syntax der Spezifikation für einen Knoten in der UML.

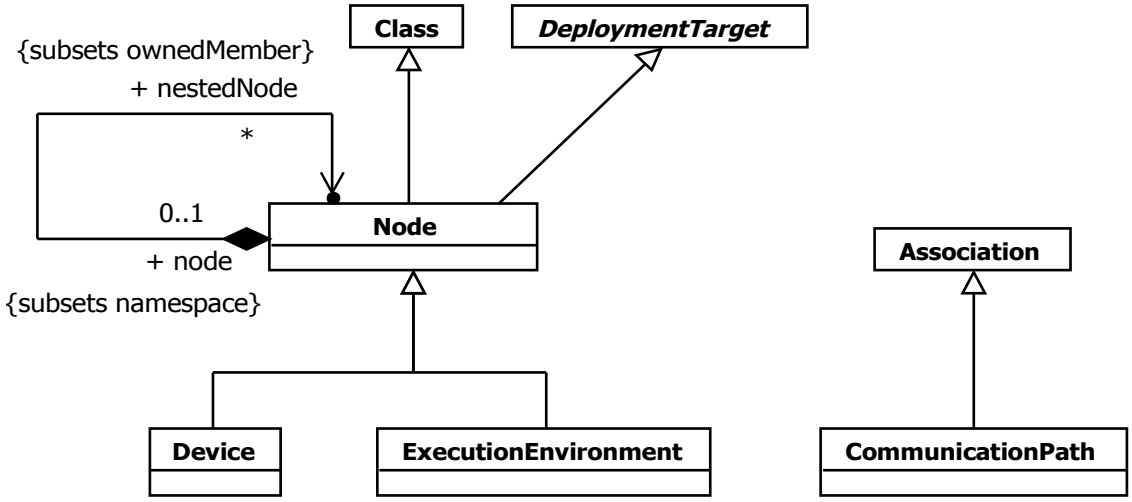

Abbildung 3.4.: Abstrakte Syntax der Spezifikation für einen UML-Knoten [34, S. 658]

3.2.2.2. UML: Bereitstellung

UML-Bereitstellungen erfassen Beziehungen zwischen logischen und/oder physischen Elementen von Systemen und den ihnen zugeordneten informationstechnischen Anlagen. Die nachfolgenden UML-spezifischen Grundlagen sind [34, S. 653] entnommen.

Das UML-Bereitstellungspaket spezifiziert Konstrukte, mit welchen die Ausführungsarchitektur von Systemen und die Zuordnung von Software-Artefakten zu Systemelementen definiert werden kann. Ein schlankes Bereitstellungsmodell, das für die meisten modernen Anwendungen ausreicht, wird somit bereitgestellt. Wo aufwändigere Bereitstellungsmodelle erforderlich sind, kann das Paket durch UML-Profile oder Metamodelle erweitert werden, um spezifische Hardware- und/oder Softwareumgebungen abzubilden.

Eine Bereitstellung erfasst die Beziehung zwischen einem konzeptionellen oder physischen Element eines modellierten Systems und den ihm zugeordneten Informationsbetriebsmittel. Systemelemente werden als „DeployedTargets“ und Informationsbetriebsmitteln als „DeployedArtifacts“ dargestellt.

Abbildung 3.5 zeigt mit einem UML-Klassendiagramm die abstrakte Syntax der Spezifikation der Bereitstellung in der UML.

3.3. Palladio

Palladio ist ein Ansatz zur Simulation von komponentenbasierten Softwarearchitekturen. Palladio analysiert dabei Softwaresysteme auf Modellebene, um so zum Beispiel Engpässe bei der Performance oder mögliche Probleme bei der Skalierbarkeit zu finden [38].

Die Entwicklung des Palladio-Projektes begann bereits im Jahr 2003 an der Universität Oldenburg und wird heute am Karlsruher Institut für Technologie (KIT), Forschungszentrum Informatik (FZI) und der Universität Stuttgart weiterentwickelt.

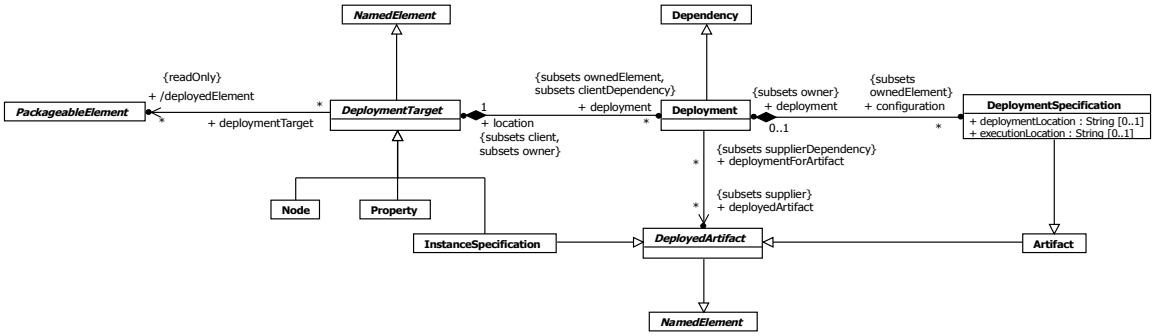

Abbildung 3.5.: Abstrakte Syntax der Spezifikation der UML-Bereitstellung [34, S. 653]

Palladio ist ein Ansatz zur Definition von Softwarearchitekturen mit Fokus auf Performance-Qualitätseigenschaften. Palladio verwendet das Palladio-Komponentenmodell als zugrundeliegendes Metamodell. Der Ansatz dieser Masterarbeit baut auf diesem Palladio Komponentenmodell auf, die Grundkonzepte können jedoch auch mit verschiedenen UML-basierten Software-Architekturmodellen verwendet werden. Die nachfolgenden Palladio-spezifischen Grundlagen sind [24] entnommen.

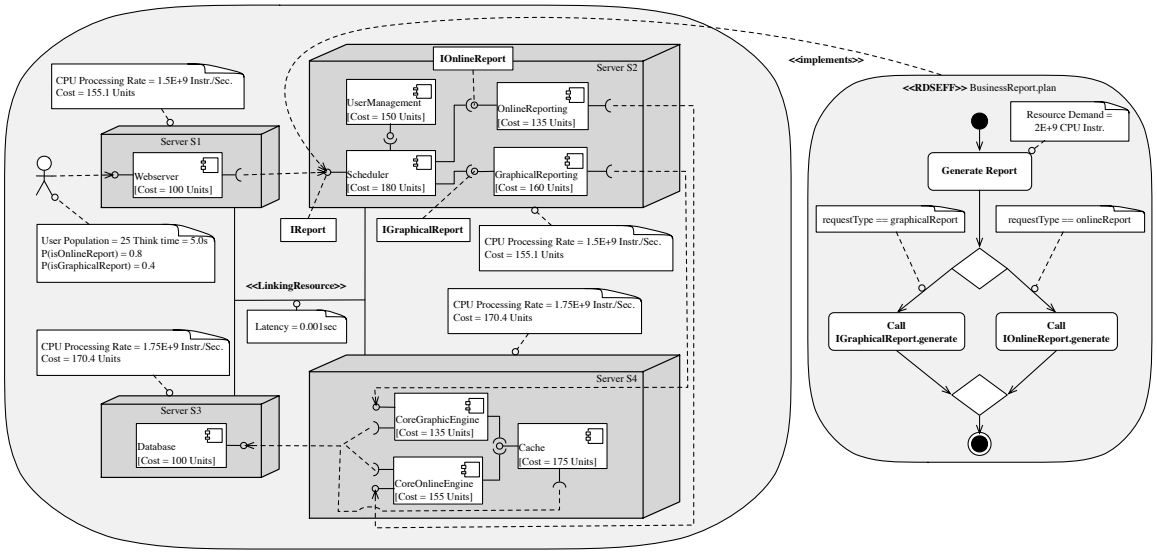

Abbildung 3.6.: Architekturmodell des Business Reporting System mit Performance-Annotationen

Als laufendes Beispiel betrachten wir das Business Reporting System (BRS) Modell, das in Abbildung 3.6, mit dem Ecore-basierten Palladio-Komponentenmodell-Metamodell realisiert und hier in UML-ähnlichen Diagrammen zum schnellen Verständnis visualisiert wird. Dieses Business Reporting System ermöglicht es dem Anwender, statistische Analysen von Geschäftsprozessen abzurufen. Zu diesem Zweck bietet das System dem Anwender die Möglichkeit, grafische und Online-Berichte zu erstellen. Das System basiert

auf einer Fallstudie zu einem realen System. Das vom Softwarearchitekten spezifizierte Architekturmodell besteht aus neun Softwarekomponenten, die auf vier verschiedenen Hardwareknoten eingesetzt werden. Die Softwarekomponenten enthalten Kostenannotationen (siehe Abschnitt 3.3.3), während die Hardwareknoten mit Performanceannotationen, wie den Verarbeitungsraten, und Kostenannotationen, wie fixe und variable Kosten, versehen sind sind.

Ein Softwarearchitekt erstellt Komponentenspezifikationen, welche von verschiedenen Komponentenentwicklern modelliert werden, um ein Systemmodell zu erstellen. Mit einem zusätzlichen Nutzungsmodell, das die Ankunftsraten der Benutzer oder der Benutzerpopulation und deren Bedenkzeit des Systems beschreibt, sowie einem zusätzlichen Modell der Ressourcenumgebung und der Zuordnung von Komponenten zu Ressourcen, ist das Modell vollständig und kann in analytische oder simulationsbasierte Modelle für Qualitätsanalysen umgewandelt werden. Die Performancessimulation wird schließlich durch die Lösung eines solchen Modells analytisch oder simulationsbasiert durchgeführt. In diesem Beitrag verwenden wir die Transformation von Palladio-Komponentenmodell zu Layered-Queueing-Netzwerken [17]. Details zur Performancessimulation sind für die Diskussion des Ansatzes dieser Masterarbeit nicht erforderlich, können jedoch in [6, 17, 38] gefunden werden.

Der Gestaltungsraum auch für ein so einfaches Beispiel ist riesig. Die manuelle Prüfung der möglichen Entwurfsalternativen in einem Trial-and-Error-Ansatz ist aufwendig und fehleranfällig. Der Softwarearchitekt kann nicht einfach Entwurfsalternativen erstellen, die lokal für alle Qualitätskriterien optimal sind, und es ist praktisch unmöglich, globale Optima zu finden, weil dies die Modellierung jeder Alternative erfordert. In der Praxis wird diese Situation oft durch Überbereitstellung, das heißt durch die Einbeziehung schneller und teurer Hardware-Ressourcen, gemildert, was zu unnötig hohen Kosten führen kann.

3.3.1. Palladio-Komponentenmodell

Der Palladio-Ansatz verwendet das Palladio-Komponentenmodell (Palladio Component Model, PCM) als zugrunde liegendes Metamodell [6, 7]. Das Palladio-Komponentenmodell ist ein Metamodell, das die formale Spezifikation von relevanten Informationen einer komponentenbasierten Architektur ermöglicht. Es wurde mit Fokus auf die Vorhersage von Quality-of-Service (QoS) Attributen wie Performance und Zuverlässigkeit ausgelegt. Das Palladio-Komponentenmodell-Metamodell ist in Ecore definiert, das eine weitere Meta-Metamodellierungssprache im Rahmen des Eclipse Modeling Framework (EMF) ist.

3.3.2. Service-Effekt-Spezifikation

Für jeden Softwarekomponenten-Dienst liefern die Komponentenentwickler eine abstrakte Verhaltensbeschreibung, die als Service-Effect-Specification (SEFF) bezeichnet wird. Service-Effect-Specifikationen modellieren den abstrakten Kontrollfluss durch einen Komponentendienst in Form von internen Aktionen, das heißt Ressourcenanforderungen beim Zugriff auf die zugrunde liegende Hardware, und externen Aufrufen, beim Zugriff auf verbundene Komponenten. Service-Effect-Specifikationen werden automatisch zusammengestellt, um das systemweite Verhalten zu bestimmen. Die Modellierung jedes Kompo-

nentenverhaltens mit separaten Service-Effect-Specificationen ermöglicht den schnellen Austausch von Komponentenspezifikationen, ohne dass systemweite Verhaltensspezifikationen manuell geändert werden müssen, wie beispielsweise in UML-Sequenzdiagrammen erforderlich.

Für Performance-Anmerkungen können Komponentenentwickler die erweiterte Ressourcen beanspruchende Service-Effect-Specification (RDSEFF) verwenden. Mit diesen Ressourcen beanspruchende Service-Effect-Specificationen spezifizieren Entwickler den Ressourcenbedarf ihrer Komponenten, wie beispielsweise in Form von auszuführenden CPU-Befehlen. Während der Analyse wird der Ressourcenbedarf durch die Verarbeitungsrate der modellierten Ressourcenumgebung geteilt, um die tatsächliche Ausführungszeit zu ermitteln, die von den Prozessoren gefordert wird. Der Ressourcenbedarf kann als Verteilungsfunktion angegeben werden, entweder über Standardfunktionen wie Exponentialverteilung oder Gammaverteilung oder durch die Definition einer schrittweisen Approximation einer beliebigen Verteilungsfunktion, beispielsweise auf Basis von Messdaten. Ein Beispiel Ressourcen beanspruchende Service-Effect-Specification ist oben in Abbildung 1.1 für den Serviceplan dargestellt. Weitere Details über die (Ressourcen beanspruchende) Service-Effect-Specification sind für die Diskussion des Ansatzes dieser Masterarbeit nicht erforderlich, können jedoch in [26, 38] gefunden werden.

3.3.3. Kostenmodell

Um die Kostendimension in Palladio mit PerOpteryx einbeziehen zu können, wird in [24, S. 38–39] ein einfaches Kosten-Nutzen-Modell realisiert, das es erlaubt, Kostenunterschiede verschiedener Entwurfsoptionen auszudrücken und um die Kostenunterschiede zwischen Architekturkandidaten zu bewerten. Das verwendete Kostenmodell ermöglicht die Zuordnung von Kosten zu Softwarekomponenten mit der Hardware und erlaubt hierdurch, Softwarearchitektur mit Kostenschätzungen zu versehen. Es unterscheidet zwischen Anschaffungskosten und Betriebskosten, sodass die Auswirkungen der Softwarearchitektur auf die Gesamtbetriebskosten abgeschätzt werden können. Der Anwender kann entweder die Gesamtkosten berechnen, beispielsweise den Kapitalwert der Kosten auf Basis eines angenommenen Zinssatzes, oder er kann die beiden Kostenarten als separate Kriterien zur Verbesserung und zum Ausgleich behandeln. Die Kosten der Softwarekomponenten spiegeln somit alle relevanten Kosten wider, welche durch die Implementierungsphasen und späteren Lebenszyklusphasen dieser Softwarekomponenten verursacht werden.

3.3.4. Strukturmodell

Das Strukturmodell ist eine Erweiterung für das PCM, um mit diesem die Eigenschaften einer Entwurfsentscheidung wiederverwendbar abbilden zu können. Das Strukturmodell umfasst Informationen wie die Struktur einer Entwurfsentscheidung und auch deren Einfluss auf die Qualität eines Softwaresystems. Mithilfe des Strukturmodells lassen sich ganze Umbaumaßnahmen für die Integration der Komponenten einer Entwurfsentscheidung formalisieren [10, 41].

Hierbei ist der Ansatz des Strukturmodells, eine Entwurfsentscheidung zusammenhängend sowohl als Provided-, Complete- und Implementation-Komponenteninstanz zu

modellieren. Der Provided-Type definiert hierbei die Qualitätseinflüsse einer Entwurfsentscheidung, im Complete-Type werden all die Komponenten definiert, welche benötigt werden, um die Entwurfsentscheidungen zu realisieren, und der Implementation-Type definiert eine vollständige Implementierung aller von einer Entwurfsentscheidung benötigten Komponenten. Zu den Komponenten einer Entwurfsentscheidung werden zusätzlich noch weitere Annotationen gespeichert, mit welchen modelliert werden kann, wie dieses Subsystem einer Entwurfsentscheidung in eine bestehende Architektur zu integrieren ist und auch wie der physische Aspekt der Verteilung dieser Komponenten einer Entwurfsentscheidung beschaffen ist.

3.3.5. PerOpteryx

Der PerOpteryx-Ansatz ist eine optionale Erweiterung für Palladio, welche in der Palladio-Bench enthalten ist [25]. Das Ziel von PerOpteryx ist es, komponentenbasierte Softwarearchitekturen auf der Basis von modellbasierten Qualitätsvorhersagetechniken zu verbessern. Hierzu nimmt PerOpteryx eine vollständige Palladio-Komponentenmodell-Instanz als Eingabe und erzeugt daraus neue Palladio-Komponentenmodell-Instanzen [24].

Das Palladio-Komponentenmodell trennt parametrisierte Komponenten-Performance-Modelle von den Kompositions- und Ressourcen-Modellen. Damit unterstützt das Palladio-Komponentenmodell natürlich viele architektonische Freiheitsgrade, wie beispielsweise den Austausch von Komponenten, die Änderung der Komponentenzuordnung und dergleichen. Der PerOpteryx-Ansatz untersucht diese vorgegebenen Freiheitsgrade und unterstützt damit fundierte Kompromissentscheidungen hinsichtlich Performance, Zuverlässigkeit und Kosten. Die nachfolgenden PerOpteryx-spezifischen Grundlagen sind [24] entnommen.

PerOpteryx nutzt für die Untersuchung Freiheitsgrade der Softwarearchitektur, die entweder automatisch aus dem Architekturmodell vordefiniert und abgeleitet oder vom Architekten manuell modelliert werden können.

Aus dem oben angeführten Business Reporting System Beispiel können zwei Arten von Freiheitsgraden abgleitet werden: Zum Beispiel können die Komponentenzuordnung und die Serverkonfiguration geändert werden. Für den Verarbeitungsgrad können mehrere Server mit jeweils unterschiedlichen möglichen Verarbeitungsraten eingesetzt werden. Als Beispiel für manuell modellierte Freiheitsgrade sei angeführt, dass einige Komponenten der Architektur Standardfunktionen bieten, für die andere Implementierungen, das heißt andere Komponenten, verfügbar sind. Hier könnte eine alternative Komponente für mehrere Komponenten im Business Reporting System verwendet werden, wie beispielsweise die Komponente Scheduler. Diese Komponente kann durch eine funktional vollwertige QuickScheduler-Komponente ersetzt werden. Wenn die QuickScheduler-Komponente weniger Ressourcen benötigt, jedoch auch teurer als der Scheduler ist, hat das resultierende Architekturmodell eine geringere Antwortzeit, jedoch höhere Kosten.

Die daraus resultierenden Freiheitsgrade spannen einen Entwurfsraum, welcher automatisch erforscht werden kann. Jedes dieser Architekturmodelle wird durch die Wahl einer Entwurfsoption für jeden Freiheitsgrad definiert. Wir nennen ein solches mögliches Architekturmodell ein Kandidatenmodell. Der Satz aller möglichen Kandidatenmodelle

entspricht dem Satz aller möglichen Kombinationen der Gestaltungsmöglichkeiten, dem Entwurfsraum.

Mithilfe der quantitativen Qualitätsbewertung von Palladio ermittelt PerOpteryx für jedes Kandidatenmodell Performance, Zuverlässigkeit und Kosten. Zusätzlich zu den Analyse-Funktionen benötigt PerOpteryx eine Angabe, ob eine Qualität maximiert oder minimiert werden soll.

Basierend auf den Freiheitsgraden, als den Optimierungsvariablen, und den Qualitätsbewertungsfunktionen, als Optimierungsziele, verwendet PerOpteryx generische Algorithmen und problemspezifische Heuristiken, um die Pareto-Front der optimalen Kandidaten zu approximieren. Details zur Optimierung sind für die Diskussion des Ansatzes dieser Arbeit nicht erforderlich, können jedoch in [24, 25, 38] gefunden werden.

3.3.6. Qualitative Argumentation

Das Qualitätsmodell ist ein Ansatz zur Modellierung von informellem Wissen und zur qualitativen Argumentation, um Softwarearchitekturen automatisch zu optimieren [42]. Zu diesem Zweck modelliert das Qualitätsmodell explizit informelles Wissen und nutzt dieses Wissen zusammen mit quantifizierten Werten, um die Optimierung der automatisierten Softwarearchitektur zu verbessern.

Zu diesem Zweck wurde ein Metamodell und Analyseverfahren entwickelt, um informelles Wissen über die Wechselwirkungen zwischen Qualitätsmerkmalen von Komponenten zu definieren. Dieses Metamodell enthält eine Beschreibung der Eigenschaften von Qualitätsmerkmalen der Softwarekomponenten und Regeln, welche beschreiben, wie diese Qualitätsmerkmale von anderen Softwarekomponenten beeinflusst werden. Wie bei komponentenbasierten Ansätzen üblich, werden diese Merkmale zusammen mit den entsprechenden Softwarekomponenten zusammengestellt. So fügt sich das modellierte Wissen nahtlos in die Softwarekomponenten ein, was die Wiederverwendung von Wissen durch die natürlichen Wiederverwendungsmechanismen von Softwarekomponenten ermöglicht.

Im Kern des Ansatzes wird ein nicht-quantifiziertes Qualitätsmerkmal (Abschnitt 3.3.6.1) verwendet, um ein Qualitätsniveau einer Komponente auszudrücken. Auch werden spezielle Abbildungsregelwerke (Abschnitt 3.3.6.2) definiert, welche ausdrücken, wie dieses Qualitätsniveau durch die Qualität anderer Komponenten beeinflusst wird. Hierdurch wird das informelle Wissen statisch in einer Komponente gespeichert. Das Qualitätsmodell ermöglicht es auch, dieses informelle Wissen der einzelnen Komponenten in einem vollständigen System auszuwerten. Details zur eigentlichen qualitativen Argumentation über dieses informelle Wissen sind für die Diskussion des Ansatzes dieser Masterarbeit nicht erforderlich, können jedoch in [42] gefunden werden.

3.3.6.1. Nicht-quantifiziertes Qualitätsmerkmal

Im Kern des Ansatzes zur qualitativen Argumentation wird ein nicht-quantifiziertes Qualitätsmerkmal (not-quantified quality attribute, NQA) verwendet, um ein Qualitätsniveau, das das informelle Wissen modelliert, einer Softwarekomponente auszudrücken [9]. Ein nicht-quantifiziertes Qualitätsmerkmal stellt hierfür eine relevante Qualitätseigenschaft

einer Softwarekomponente dar. Softwarearchitekten können diesen Mechanismus nutzen, um den Softwarekomponenten Qualitätsmerkmale, wie beispielsweise Sicherheit und Wartbarkeit, entsprechend ihrer Architekturkenntnisse zuzuordnen. Ein nicht-quantifiziertes Qualitätsmerkmal wird durch ein Paar dargestellt, das sich aus der Qualitätsdimension und dem entsprechenden Dimensionselement zusammensetzt.

Eine Qualitätsdimension ist ein Attribut, das durch den qualitativen Argumentationsansatz analysiert und bewertet werden kann. Dies kann eine Softwarequalität sein, wie beispielsweise Sicherheit, aber auch jede andere Eigenschaft von Softwarekomponenten, wie beispielsweise deren Hersteller. Eine solche Qualitätsdimension könnte beispielsweise als „Wartbarkeit“ oder „Benutzerfreundlichkeit“ bezeichnet sein.

Ein Dimensionselement stellt den konkreten Wert für eine Qualitätsdimension dar. Ein solches Dimensionselement wird in einem Dimensionsset definiert, das den Bereich aller möglichen Werte festlegt, die innerhalb einer Dimension angegeben werden können. Ein solches Dimensionsset wird durch einen geordneten Satz von ordinalskalierten Variablen modelliert, welche sowohl nominelle Informationen als auch Informationen über ihre Ordnung enthalten. Ein solches Dimensionsset könnte beispielsweise `{"--", "-", "0", "+", "++"}` sein, wobei `"-"` ein Dimensionselement aus diesem wäre.

Angenommen ein Softwarearchitekt möchte zum Ausdruck bringen, dass eine Softwarekomponente eine vergleichsweise „gute“ Wartbarkeit hat, so würde dieser das nicht-quantifizierte Qualitätsmerkmal `("Wartbarkeit", "+")` modellieren und dieses mit dieser Softwarekomponente verknüpfen.

3.3.6.2. Abbildungsregelwerk

Ein Abbildungsregelwerk (mapping rule set, MRS) ist der Kern der symbolischen nicht-numerischen Berechnung des Ansatzes zur qualitativen Argumentation. Das Abbildungsregelwerk legt fest, wie ein nicht-quantifiziertes Qualitätsmerkmal einer Softwarekomponente von anderen nicht-quantifizierten Qualitätsmerkmalen anderer Softwarekomponenten beeinflusst wird. Zu diesem Zweck besteht ein Abbildungsregelwerk aus mehreren Abbildungsregeln und einer Qualitätsdimension, für welche diese Abbildungsregeln gelten. Das Abbildungsregelwerk bestimmt ein neues nicht-quantifiziertes Qualitätsmerkmal, dessen Dimensionselement nach dessen Abbildungsregeln berechnet wird.

Eine Abbildungsregel berechnet das resultierende Dimensionselement für eine Menge von nicht-quantifizierten Qualitätsattributen. Hierfür ist eine Abbildungsregel als ein Paar aus einer Folge von Qualitätsdimensionen und einem Satz von Abbildungsregel-Einträgen definiert. Ein Abbildungsregel-Eintrag ist als Paar aus einer Folge von Dimensionselementen zur Eingabe und dem daraus resultierenden Dimensionselement definiert.

Tabelle 3.3 zeigt ein Beispiel eines Abbildungsregelwerks für die Qualitätsdimension „Wartbarkeit“. Dieses Abbildungsregelwerk zeigt, wie sich die Qualitätsdimensionen „Dokumentation“ und „Modularität“ auf die Qualitätsdimension „Wartbarkeit“ einer Softwarekomponente auswirken. Beispielsweise würde das Abbildungsregelwerk in Tabelle 3.3 die Menge der beiden nicht-quantifizierten Qualitätsmerkmale `("Dokumentation", "+")` und `("Modularität", "0")` auf das neue nicht-quantifizierte Qualitätsmerkmal `("Wartbarkeit", "+")` abbilden.

Wartbarkeit					
		Dokumentation			
		++	+	-	--
Modularität	++	++	++	+	0
	+	+	+	-	0
	0	0	+	-	-
	-	-	0	--	--
	--	-	-	--	--

Tabelle 3.3.: Abbildungsregelwerk für die Qualitätsdimension Wartbarkeit

3.4. Domänenspezifische Sprache

Eine domänenspezifische Sprache (domain-specific language, DSL) ist eine auf ein bestimmtes Anwendungsfeld, die sogenannte Domäne, spezialisierte formale Sprache. Hierbei soll eine DSL alle Probleme des Anwendungsfeldes darstellen können und dabei gleichzeitig nichts darstellen, was außerhalb des Anwendungsfeldes liegt.

Dies steht im Gegensatz zu einer allgemeinen Programmiersprache (general-purpose programming language, GPL), welche hingegen in vielen Anwendungsfeldern verwendbar ist. Eine GPL, wie beispielsweise Java, bietet grundlegend die Möglichkeit, Lösungen für alle nur denkbaren Anwendungsfelder umzusetzen. Für manche Anwendungen ist dies jedoch manchmal mehr als benötigt wird. Dem gegenüber steht eine DSL, welche zwar einen eingeschränkten Funktionsumfang hat, jedoch das Anwendungsfeld vollständig und präzise beschreibt.

Zu den Vorteilen einer DSL gegenüber der Nutzung einer GPL zählen unter anderem die bessere Lesbarkeit durch die deklarative Beschreibung eines Sachverhaltes. Auch aufgrund des beschränkten Umfangs kann eine DSL in der Regel leichter erlernt und verwenden werden als eine GPL.

3.4.1. Xtext

Eclipse Xtext ist ein Framework für die Entwicklung von allgemeinen Programmiersprachen und domänenspezifischen Sprachen. Es ist ein plattformübergreifendes Open-Source-Framework, das mit der Eclipse Public License (EPL) lizenziert ist. Zudem ist Xtext ein Teil des Eclipse Modeling Framework (EMF) [15].

Xtext deckt alle Aspekte einer vollständigen Sprachinfrastruktur ab: Von Parsern über Linker, Compiler, Interpreter bis hin zu einer vollentwickelten Integration für Eclipse. Es kann voll ausgestattete Texteditoren für allgemeine und domänenspezifische Sprachen erstellen. Um eine Vielzahl von Syntaxen abzudecken, wird im Hintergrund wird der LL(*)-Parser-Generator von ANTLR verwendet. Auch generiert Xtext ein EMF-Metamodell als ein Klassenmodell für den abstrakten Syntaxbaum (abstract syntax tree, AST). Hierdurch bietet es die Möglichkeit, den AST zu analysieren, zu validieren, zu transformieren und Java Quelltext daraus abzuleiten. Darüber hinaus können mit Xtext entwickelte Sprachen auch in Editoren integriert werden, welche das Language Server Protocol (LSP) unterstützen.

Die Xtext-Grammatiksprache ist der Grundstein von Xtext. Hierbei handelt es sich um eine domänenspezifische Sprache, welche für die Beschreibung von textbasierten Programmiersprachen entwickelt wurde. Natürlich ist diese selbst mit Xtext implementiert [8].

Um den eigenen Quelltext für die Weiterverarbeitung in ein geeigneteres Format umzuwandeln, muss dieser erst in eine neue Struktur übersetzt werden. Hierfür wird die Texteingabe in eine Folge von sogenannten Tokens umgewandelt. In diesem Zusammenhang ist ein Token eine Art stark typisierter Teil oder Bereich der Eingabesequenz. Sie besteht aus einem oder mehreren Zeichen und wird durch eine bestimmte Produktionsregel oder ein bestimmtes Schlüsselwort abgeglichen und stellt somit ein atomares Symbol dar. Die Produktionsregeln in Xtext werden mithilfe von erweiterten Backus-Naur"Form (extended Backus-Naur form , EBNF) Ausdrücken beschrieben.

3.4.2. Backus-Naur-Form

Die Backus-Naur-Form (BNF) ist eine Notationstechnik für kontextfreie Grammatiken, welche häufig zur Beschreibung der Syntax von Programmiersprachen verwendet wird [23]. Die BNF wurde nach ihren Urhebern benannt, den Informatikern John Backus und Peter Naur. Beide verwendeten diese Notationstechnik erstmalig zur Definition der Syntax der Programmiersprache ALGOL 60 [31].

Zeichenfolgen, welche zwischen den Klammern <> eingeschlossen sind, stellen metalinguistische Variablen dar, deren Werte Zeichenfolgen sind. Die zwei Marken ::= und | sind metalinguistische Konnektive. Hierbei wird die Zeichenfolge ::= zur Definition verwendet und das Zeichen | hat die Bedeutung eines logischen Oder zur Definition von Alternativen. Jede Marke in einer Formel, welche weder eine Variable noch ein Bindeglied ist, bezeichnet sich selbst (oder die Klasse von Marken, welche ihr ähnlich sind). Die Juxta-Position von Marken und/oder Variablen in einer Formel bedeutet die Gegenüberstellung der angegebenen Sequenzen.

Die Produktionsregeln der BNF sind genau die in kontextfreien Grammatiken (context-free grammar, CFG) erlaubten Regeln, folglich können beide Formalismen dieselben Sprachen erzeugen. Eine CFG ist eine formale Grammatik, welche nur solche einfachen Produktionsregeln enthält, bei welchen immer genau ein Nichtterminalsymbol auf eine beliebig lange Folge von Nichtterminal- und/oder Terminalsymbolen abgeleitet wird. Die Produktionsregeln haben folglich immer die Form $V \rightarrow w$, wobei V ein Nichtterminalsymbol ist und w eine Zeichenkette bestehend aus Nichtterminal- und/oder Terminalsymbolen. Die kontextfreien Sprachen sind genau die Sprachen, welche von einem nichtdeterministischen Kellerautomaten erkannt werden können. $LL(*)$-Grammatiken sind eine Teilmenge der kontextfreien Grammatiken, welche das Parsen durch direkte Konstruktion einer ganz linken Ableitung erlauben, und bilden damit die theoretische Basis für die Syntax der meisten Programmiersprachen.

4. Qualitätseffekt-Spezifikation

Für den Ansatz dieser Arbeit wird ein formales Modell konzipiert, das zum einen die strukturellen Eigenschaften von Softwarekomponenten in einem Softwaresystem und zum anderen die Regeln für die Transformation von Qualitätsattributen dieser Softwarekomponenten abbildet. Mit diesem formalen Modell können die strukturellen Eigenschaften einer Softwarekomponente in einem komponentenbasierten Softwaresystem abgebildet werden, sodass eine Softwarekomponente durch verschiedene voneinander unabhängige strukturelle Eigenschaften unterschiedlich aus verschiedenen Kontexten charakterisiert werden kann. Diese Charakterisierung ist eine abstrakte Beschreibung einer Softwarekomponente und stellt so ihre strukturellen Eigenschaften dar, damit in einem beliebigen komponentenbasierten Softwaresystem diese Softwarekomponente identifiziert werden kann. Auch können mit diesem formalen Modell die Regeln für die Transformation von Qualitätsattributen dieser Softwarekomponente abgebildet werden, sodass Transformationen für die Qualitätsattribute einer Softwarekomponente spezifiziert werden können.

Die Motivation für diesen Ansatz, mit welchem Qualitätsattribute von Softwarekomponenten in Relation zu ihren strukturellen Eigenschaften abgebildet werden können, ist die Existenz von Qualitätsattributen, welche lediglich aus den strukturellen Eigenschaften eines Softwaresystems bestimmt werden können. Beispielsweise, wie viele Komponenten direkt auf die Datenbank zugreifen, wären ein maßgebliches Kriterium zur Bestimmung der Modularität eines Softwaresystems.

Zu diesem Zweck wird ein formales Modell erstellt, mit welchem die strukturellen Eigenschaften einer Softwarekomponente in einem komponentenbasierten Softwaresystem und auch die Regeln für die Transformation von Qualitätsattributen für diese Softwarekomponente abgebildet werden können. Die Darstellung der Einflussfaktoren in einem Softwarearchitektur-Modell werden durch erweiterte Backus-Naur-Form ähnliche Produktionsregeln (Abschnitt 3.4.2) modelliert. Durch den Ansatz, eine erweiterte Backus-Naur-Form zu verwenden, ist das formale Modell unabhängig von bestimmten Technologien, wie beispielsweise Java. Diese Produktionsregeln bilden die formale Grammatik, durch welche das formale Modell beschrieben und erzeugt werden kann.

Diese formale Grammatik beschreibt eine domänenspezifische Sprache (formale Sprache) und legt so eindeutig fest, ob ein Wort Element einer Sprache ist oder nicht. Diese domänenspezifische Sprache bestimmt ein Regelsystem (konkrete Syntax) zur Spezifizierung dieser strukturellen Eigenschaften und der Regeln für die Transformation von Qualitätsattributen. Nach diesem Regelsystem (konkrete Syntax) der domänenspezifischen Sprache können wohlgeformte (syntaktisch korrekte) Ausdrücke gebildet werden, welche konkrete strukturelle Eigenschaften einer Softwarekomponente und der Regeln für die Transformation von Qualitätsattributen beschreiben. Ein Ausdruck, welcher nach den Regeln dieser domänenspezifischen Sprache gebildet wurde, wird als Qualitätseffekt-Spezifikation (englisch: quality effect specification; kurz: QES) bezeichnet, da dieser den Effekt

eines Softwaresystems auf Qualitätsattribute spezifiziert. Eine solche Qualitätseffekt-Spezifikation beschreibt auf der Grundlage der domänenspezifischen Sprache die strukturellen Eigenschaften von Softwarekomponenten in einem Softwaresystem und die Regeln für die Transformation von Qualitätsattributen dieser Softwarekomponenten.

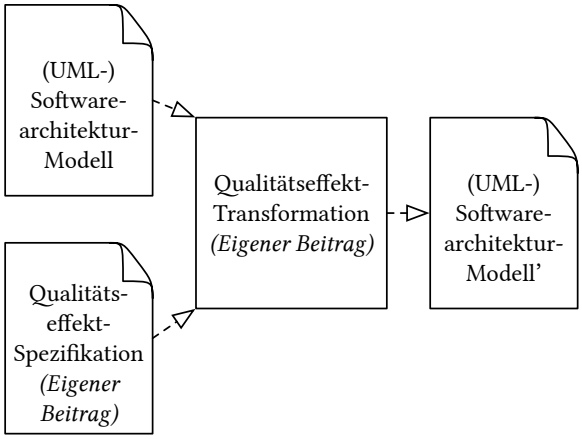

Abbildung 4.1.: Überblick über den rudimentären Vorgang des Ansatzes

Die Auswertung und Anwendung einer Qualitätseffekt-Spezifikation interpretiert eine gegebene Qualitätseffekt-Spezifikation und transformiert gegebenenfalls entsprechend dieser die Qualitätsattribute der Softwarekomponenten in einem gegebenen Softwarearchitektur-Modell. In Abbildung 4.1 ist der rudimentäre Vorgang bei der Qualitätseffekt-Transformation einer Qualitätseffekt-Spezifikation (Abschnitt 4.3) abgebildet. In diesem vereinfachten Überblick erhält die Qualitätseffekt-Transformation ein Softwarearchitektur-Modell und eine Qualitätseffekt-Spezifikation als Eingabe und als Ausgabe das transformierte Softwarearchitektur-Modell, welches die transformierten Qualitätsattribute der Softwarekomponenten beinhaltet.

Der Nutzen dieser Qualitätseffekt-Spezifikationen ist die Spezifikation der Qualitätsattribute von Softwarekomponenten, welche erst aus den spezifischen Eigenschaften einer konkreten Softwarearchitektur abgeleitet werden können, und deren Auswertung. Durch dieses formale Modell können Regeln für die Transformation von Qualitätsattributen in Abhängigkeiten der strukturellen Eigenschaften einer Softwarekomponente in einem komponentenbasierten Softwaresystem spezifiziert werden. Hierdurch können Analysen der Qualitätsattribute von Softwarekomponenten verbessert werden, indem diese auf die strukturellen Eigenschaften eines komponentenbasierten Softwaresystems angepasst werden. Durch diese verbesserten Analysen der Qualitätsattribute sollen die Werkzeuge von Softwarearchitekten verbessert werden, sodass diese fundierte Entscheidungen in einem Softwarearchitektur-Entwurfs-Prozess treffen können.

Der sachliche Ablauf des Ansatzes ist als ein Geschäftsprozessdiagramm [32] in Abbildung 4.2 dargestellt. Dieses Geschäftsprozessdiagramm zeigt in einer formalen grafischen Darstellung die sachliche Abfolge der Aufgaben mit ihren Ereignissen bei der Umsetzung des Ansatzes der Spezifikation. Zunächst wird untersucht und bewertet, welche semantischen Klassen von architekturdefinierten Einflussfaktoren in einem komponentenbasierten Softwaresystem existieren und wie ein formales Modell aussehen müsste, das anhand

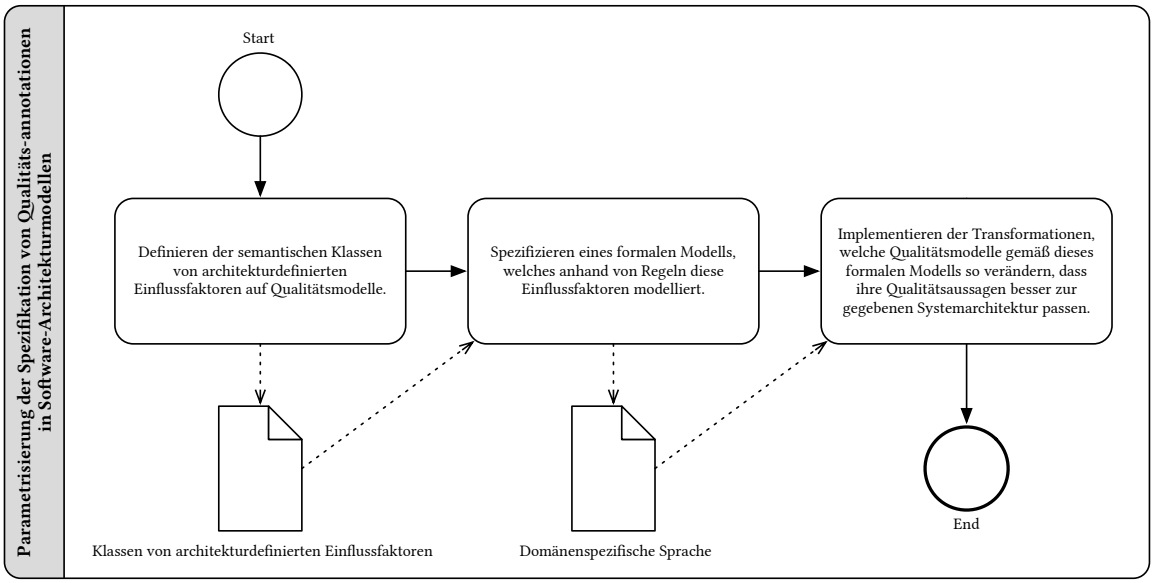

Abbildung 4.2.: Sachlicher Ablauf der Umsetzung des Ansatzes

von Regeln diese Einflussfaktoren modelliert. Das Ergebnis dieser Überlegungen wird in Abschnitt 4.1 beschrieben. Danach wird anhand dieser Überlegungen ein formales Modell konzipiert, das anhand von Regeln diese Einflussfaktoren modelliert. Dieses formale Modell als Grundlage für die Qualitätseffekt-Spezifikation wird im Abschnitt 4.2 beschrieben. Danach wird anhand dieses formalen Modells die Qualitätseffekt-Transformation für die Qualitätseffekt-Spezifikation umgesetzt, welche Qualitätsmodelle so transformieren, dass ihre Aussagen über Qualitätsattribute besser zur gegebenen Systemarchitektur passen. Diese Interpretationsregeln werden in Abschnitt 4.3 beschrieben.

4.1. Strukturelle Eigenschaften in komponentenbasierten Systemen

In diesem Abschnitt werden die strukturellen Eigenschaften von Softwarekomponenten charakterisiert, welche im Folgenden mit der domänenspezifischen Sprache im Abschnitt 4.2 beschrieben werden können. Hierfür erfolgt in den weiteren Unterabschnitten zunächst die Auseinandersetzung mit der Wahl dieser strukturellen Eigenschaften, als semantische Klassen von architekturdefinierten Einflussfaktoren, zur Charakterisierung von Softwarekomponenten in einem komponentenbasierten System. Dabei wird für jede der gewählten strukturellen Eigenschaften erläutert, weshalb an welcher Stelle diese semantischen Klassen von architekturdefinierten Einflussfaktoren in der Softwarearchitektur ausgewählt worden sind. Der Zweck jeder gewählten strukturellen Eigenschaft ist es zum einen, Softwarekomponenten exakt charakterisieren zu können. Zum anderen sollen für die Charakterisierung von Softwarekomponenten in einem komponentenbasierten Softwaresystem auch Freiheitsgrade existieren. Diese Freiheitsgrade erlauben es, Softwa-

rekomponenten durch verschiedene voneinander unabhängige strukturelle Eigenschaften unterschiedlich zu charakterisieren. Beispielsweise kann so eine Softwarekomponente einmal über ihren Namen charakterisiert werden oder ein anderes Mal über ihre Schnittstellen. Diese unterschiedlichen strukturellen Eigenschaften erlauben auch, eine Softwarekomponente aus verschiedenen Kontexten zu charakterisieren. Beispielsweise kann so eine Softwarekomponente charakterisiert werden, wenn sie nur mit einer anderen bestimmten Softwarekomponente verbunden ist.

Die Bestimmung der relevanten strukturellen Eigenschaften von Softwarekomponenten für diesen Ansatz basiert auf der Spezifikation für die formale Semantik und Notation der einzelnen Hauptkonzepte für UML-Komponentendiagramme und UML-Verteilungsdiagramme. Da die UML eine universelle visuelle Modellierungssprache ist, welche für den Einsatz mit allen Entwicklungsmethoden, Lebenszyklusstadien, Anwendungsbereichen und Medien vorgesehen ist [34, S. 3], basiert die Wahl der strukturellen Eigenschaften zur Charakterisierung von Softwarekomponenten in einem komponentenbasierten Systemen auf der UML. Die möglichen strukturellen Eigenschaften von visuell lesbaren Notationselementen der UML dienen hierbei als Grundlage für die domänenspezifische Sprache, welche in Abschnitt 4.2 spezifiziert wird.

Durch eine systematische Auswertung der UML-Spezifikation wurden insgesamt sieben verschiedene strukturelle Eigenschaften für die Charakterisierung von Softwarekomponenten bestimmt: Der Name einer Komponente (Abschnitt 4.1.1), der Identifikator einer Komponente Abschnitt 4.1.2), eine Anmerkung, welche mit einer Komponente verknüpft ist (Abschnitt 4.1.3), der Typ einer Komponente (Abschnitt 4.1.4), die Schnittstelle einer Komponente (Abschnitt 4.1.5), die Komposition einer Komponente mit anderen Komponenten (Abschnitt 4.1.6) und die Zuordnung einer Komponente auf ein Betriebsmittel (Abschnitt 4.1.7). In den folgenden sieben Unterabschnitten werden diese strukturellen Eigenschaften beschrieben.

4.1.1. Name

Ein benanntes Element ist ein Element in einem UML-Modell, das einen Namen haben kann, der direkt und/oder durch die Verwendung einer Zeichenkette angegeben werden kann [34, S. 47]. Der Name dient hierbei zur Identifizierung und Individualisierung eines Elements in einem UML-Modell. Da bei visuell lesbaren Notationselementen der Name eines Elements eines der charakteristischsten Eigenschaften ist, dient dieser als eine der Grundlagen für die semantischen Klassen von architekturdefinierten Einflussfaktoren.

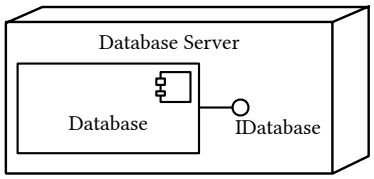

Abbildung 4.3.: Verteilungsdiagramm einer Komponente mit Schnittstelle und Server

Die Abbildung 4.3 zeigt das UML-Verteilungsdiagramm einer Datenbankkomponente mit dem entsprechenden Namen „Database“. Die Datenbankkomponente bietet hierbei die

Schnittstelle mit den Namen „IDatabase“ an und ist auf dem entsprechenden Server mit dem Namen „Database Server“ bereitgestellt. Zu der Charakterisierung von Softwarekomponenten werden nicht nur der Name einer Komponente betrachtet, sondern auch der Name der zugehörigen Schnittstellen (Abschnitt 4.1.5) und auch der Name des zugehörigen Servers (Abschnitt 4.1.7).

Der Name wurde als einer der strukturellen Eigenschaften zur Charakterisierung von Softwarekomponenten in einem komponentenbasierten Softwaresystem bestimmt, da dieser eine natürlichsprachliche Bestimmung einer Softwarekomponenten erlaubt. Auch lassen sich durch den Namen gleiche Softwarekomponenten charakterisieren, welche auch in mehrfacher Ausführung vorhanden sein können.

4.1.2. Identifikator

Die Schnittstelle „Entity“ ist eine Metaklasse, welche alle Entitäten des PCM repräsentiert, welche sowohl einen Namen als auch einen Identifikator haben [38]. Hierbei ist der Identifikator ein mit einer bestimmten Entität des PCM verknüpftes Merkmal, das zur eindeutigen Identifizierung der Entität dient.

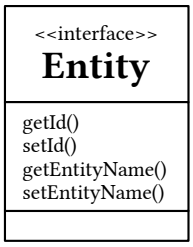

Abbildung 4.4.: Entity-Schnittstelle aus dem PCM

Die „Entity“-Schnittstelle in Abbildung 4.4 wird unter anderem von den Klassen im PCM zu Realisierung von Komponenten, Rollen und Betriebsmittel implementiert. Hierdurch wird in Palladio jeder Softwarekomponente, sowie den dazugehörigen Rollen (Abschnitt 4.1.5) und Betriebsmittel (Abschnitt 4.1.7), eine eindeutige Zeichenkette als Identifikator zugeordnet.

Der Identifikator wurde als eine der strukturellen Eigenschaften zur Charakterisierung von Softwarekomponenten in einem komponentenbasierten Softwaresystem bestimmt, da dieser eine markante und zugleich eindeutige Bestimmung einer Softwarekomponente erlaubt. Im Gegensatz zu dem Namen, welcher gleiche Softwarekomponenten charakterisiert, charakterisiert der Identifikator immer dieselbe Softwarekomponente.

4.1.3. Annotation

Ein Kommentar ist eine textuelle Anmerkung, welche an eine Gruppe von Elementen angehängt werden kann [34, S. 40]. Eine solche textuelle Anmerkung kann in einem UML-Modell Informationen von Modellelementen verschiedenster Art darstellen. Annotationen (vom Lateinischen für „Anmerkung“) sind Strukturelemente, durch welche bestimmte

Informationen in ein (UML-) Modell eingebunden werden können. In diesem Zusammenhang halten Annotationen Informationen fest, welche zwar nicht als wesentlich für das Modell erachtet werden, jedoch wichtige Zusatzinformationen darstellen. So werden im Strukturmodell (Abschnitt 3.3.4) mittels Annotationen modelliert, wie eine Entwurfsentscheidung in eine bestehende Architektur zu integrieren ist und auch wie der physische Aspekt der Verteilung dieser Komponenten einer Entwurfsentscheidung beschaffen ist [10, 41].

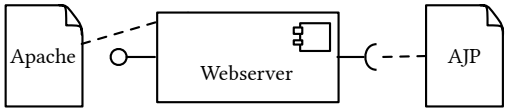

Abbildung 4.5.: Anmerkungen zur genaueren Charakterisierung einer Komponente

Die Abbildung 4.5 zeigt zwei Annotationen, welche einmal die Komponenten und einmal die Rolle dieser Komponente genauer charakterisiert. Um eine textuelle Annotation, als eine semantische Klasse der architekturdefinierten Einflussfaktoren, in einem formalen Modell zu modellieren, wird diese durch ihren enthaltenen Text charakterisiert.

Annotationen wurden als eine der strukturellen Eigenschaften zur Charakterisierung von Softwarekomponenten in einem komponentenbasierten Softwaresystem bestimmt, da diese wichtige Zusatzinformationen darstellen. Hierdurch können auch von einem Anwender eigens hinzugefügte Zusatzinformationen, welche nicht direkt aus den weiteren Eigenschaften eines (UML-) Modells abgeleitet werden können, betrachtet werden.

4.1.4. Typ

In der UML-Spezifikation hat eine Komponente eine externe Ansicht (oder „Black-Box“-Ansicht), welche über ihre öffentlich sichtbaren Eigenschaften und Operationen spezifiziert ist. Jedoch kann eine Komponente auch eine interne Ansicht (oder „White-Box“-Ansicht) haben, welche durch ihre privaten Eigenschaften und die Realisierung von Klassifikatoren spezifiziert ist [34, S. 210]. Diese beiden Konzepte werden in dem PCM durch die Entitäten „BasicComponent“ und „CompositeComponent“ realisiert. Die „BasicComponent“ stellt hierbei eine externe Ansicht („Black-Box“) der Komponentenimplementierung dar, welche die atomaren Bausteine einer Softwarearchitektur sind. Eine weitere Unterteilung in kleinere Komponenten ist bei einer „BasicComponent“ („Black-Box“) nicht möglich. Hingegen setzt sich die „CompositeComponent“ aus inneren Komponenten zusammen [38].

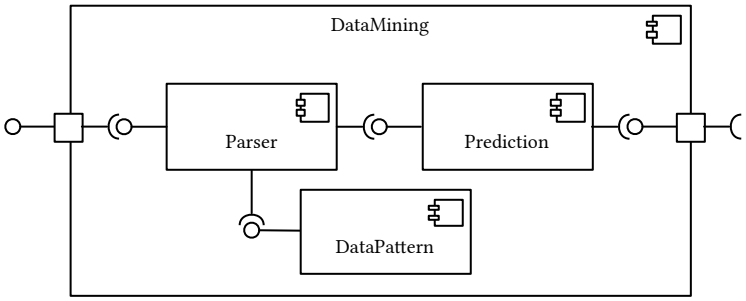

Abbildung 4.6.: Interne Struktur einer zusammengesetzten Komponente

Die Abbildung 4.6 zeigt die interne Struktur („White-Box“-Ansicht) der „DataMining“-Komponente, welche intern durch die „Parser“, „Prediction“ und „DataPattern“-Komponente realisiert wird. Daneben zeigt Abbildung 4.6 die externen Ansichten („Black-Box“) der drei internen Komponenten, welche mittels ihrer Rollen charakterisiert werden.

Der Komponenten-Typ wurde als einer der strukturellen Eigenschaften zur Charakterisierung von Softwarekomponenten in einem komponentenbasierten Softwaresystem bestimmt, da dieser wichtige Informationen über die Realisierung einer Softwarekomponente gibt. Hierdurch lässt sich unterscheiden, ob eine Softwarekomponente teil einer weiteren Softwarekomponente ist oder nicht.

4.1.5. Schnittstelle

Die UML spezifiziert eine Komponente als modulare Einheit, welche über eine oder mehrere angebotene und/oder erforderliche klar definierte Schnittstellen verfügt [34, S. 208]. Diesbezüglich spezifiziert die UML eine Schnittstelle als eine Art Klassifikator, welcher eine Deklaration einer Reihe von öffentlichen Merkmalen und Pflichten darstellt, die zusammen wiederum einen kohärenten Dienst darstellen [34, S. 171]. Zu diesem Zweck stellt im PCM die „Role“-Entität die Abstraktion einer Schnittstellenfunktion von Komponenten dar. Eine angebotene Schnittstelle einer Komponente im PCM wird hierbei durch die „ProvidedRole“-Entität realisiert und eine erforderliche Schnittstelle wird durch die „RequiredRole“-Entität realisiert [38].

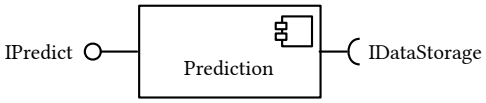

Abbildung 4.7.: Angebotene und erforderliche Schnittstelle einer Komponente

Die Abbildung 4.7 zeigt eine Komponente („Prediction“) mit einer angebotenen Schnittstelle („IPredict“) und einer erforderlichen Schnittstelle („IDataStorage“). Um eine Schnittstelle, als eine semantische Klasse der architekturdefinierten Einflussfaktoren, in einem formalen Modell zu modellieren, wird diese entweder durch ihren Namen, ihren Identifikator, ihre verknüpften Anmerkungen oder ihren genauen Typ charakterisiert. Der Typ einer Schnittstelle unterscheidet nicht nur, ob es sich um eine angebotene oder erforderliche

Rolle handelt, sondern auch, ob die Rolle einer Komponente oder einer Infrastruktur zugeordnet ist.

Die Schnittstelle wurde als eine der strukturellen Eigenschaften zur Charakterisierung von Softwarekomponenten in einem komponentenbasierten Softwaresystem bestimmt, da diese die Funktionalität einer Softwarekomponente zutreffender Weise charakterisieren kann. Hierdurch lassen sich auch ähnliche Softwarekomponenten mit gleicher Funktionalität oder die Abhängigkeiten zwischen angebotenen und erforderlichen Funktionalitäten von Softwarekomponenten charakterisieren.

4.1.6. Komposition

In der UML-Spezifikation spezifiziert ein Connector eine Verbindung, welche unter anderem die Kommunikation zwischen zwei oder mehreren Komponenten ermöglicht [34, S. 227]. Zu diesem Zweck stellt im PCM die „AssemblyConnector“-Entität eine bidirektionale Verbindung zweier Komponenten dar. Dabei verbindet die „AssemblyConnector“-Entität im PCM eine angebotene und eine erforderliche Schnittstelle von zwei verschiedenen Komponenten [38].

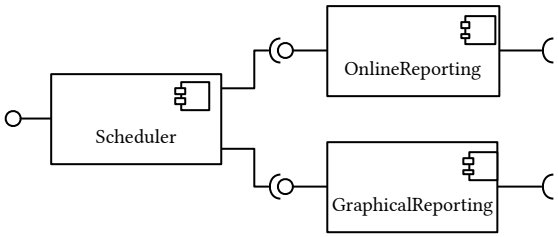

Abbildung 4.8.: Komposition dreier Komponenten

Die Abbildung 4.8 zeigt die Komposition dreier verschiedener Komponenten miteinander. Hierbei sind die beiden erforderlichen Schnittstellen der „Scheduler“-Komponente, jeweils einmal mit der angebotenen Schnittstelle der „OnlineReporting“-Komponente und einmal mit der angebotenen Schnittstelle der „GraphicalReporting“-Komponente verbunden. Um die Komposition von Komponenten, als eine semantische Klasse der architekturdefinierten Einflussfaktoren, in einem formalen Modell zu modellieren, wird dieser zum einen durch die Richtung der Verbindung und zum anderen durch die Komponente am Ende dieser Verbindungen charakterisiert. Dabei ist die Richtung der Verbindung darüber charakterisiert, ob sie mit einer angebotenen oder einer erforderlichen Schnittstelle mit einer weiteren Komponente verbunden ist.

Die Komposition von zwei Softwarekomponenten wurde als eine der strukturellen Eigenschaften zur Charakterisierung von Softwarekomponenten in einem komponentenbasierten Softwaresystem bestimmt, da dieser die Struktur der gegenseitigen Verbindungen und Abhängigkeiten zwischen den Softwarekomponenten in einem komponentenbasierten Softwaresystem charakterisiert. Hierdurch lässt sich eine Softwarekomponente allein durch ihre Beziehung zu anderen Softwarekomponenten in einem komponentenbasierten Softwaresystem charakterisieren.

4.1.7. Server

Ein Server ist ein Betriebsmittel, auf welchem Komponenten zur Ausführung bereitgestellt werden können [34, S. 658]. Normalerweise repräsentieren Server entweder Hardware-Geräte oder Software-Ausführungsumgebungen und können dabei verschachtelt und über Kommunikationswege zu beliebig komplexen Systemen verbunden werden [34, S. 657].

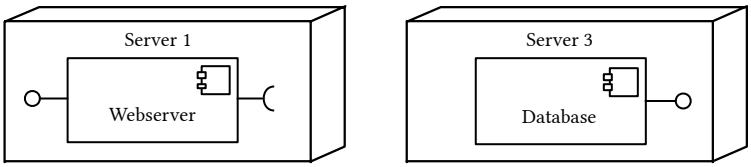

Abbildung 4.9.: Zwei Komponenten auf zwei Servern

Die Abbildung 4.9 zeigt zwei verschiedene Komponenten, „Webserver“ und „Database“, welche auf zwei verschiedenen Servern, „Server 1“ und „Server 2“, verteilt sind. Um ein Betriebsmittel, als eine semantische Klasse der architekturdefinierten Einflussfaktoren, in einem formalen Modell zu modellieren, wird dieses lediglich durch seinen Namen oder seinen Identifikator charakterisiert.

Der Server wurde als eine der strukturellen Eigenschaften zur Charakterisierung von Softwarekomponenten in einem komponentenbasierten Softwaresystem bestimmt, da dieser die Struktur der Verteilung von Softwarekomponenten in einem komponentenbasierten Softwaresystem charakterisiert. Hierdurch lässt sich eine Softwarekomponente durch ihre Beziehung zu anderen Softwarekomponenten auf einem Server in einem komponentenbasierten Softwaresystem charakterisieren.

4.2. Domänenspezifische Sprache

Um die strukturellen Eigenschaften von Softwarekomponenten in einem komponentenbasierten Softwaresystem und auch die Regeln für die Transformation von Qualitätsattributen dieser Softwarekomponenten formal abbilden zu können, wird ein formales Modell erstellt. Dieses formale Modell wird als domänenspezifische Sprache für die sogenannten Qualitätseffekt-Spezifikationen bezeichnet. Diese domänenspezifische Sprache bestimmt das Regelsystem zur Spezifizierung der strukturellen Eigenschaften von Softwarekomponenten und der Transformationen von Qualitätsattributen.

Hierbei können mit den Abfragen (Abschnitt 4.2.5) Softwarekomponenten in einem komponentenbasierten Softwaresystem anhand ihrer strukturellen Eigenschaften in komponentenbasierten Systemen (Abschnitt 4.1) beschrieben werden. Zum anderen können mit den Transformationen (Abschnitt 4.2.6) beschrieben werden, wie die Qualitätsattribute dieser zuvor beschriebenen Softwarekomponenten transformiert werden sollen. Der Ansatz hierbei ist die Spezifikation einer formalen Beschreibung der beiden Schritte (Abfragen und Transformationen) in einer domänenspezifischen Sprache.

Nach diesem Regelsystem der domänenspezifischen Sprache können wohlgeformte Ausdrücke gebildet werden, welche eine konkrete strukturelle Eigenschaft einer Softwarekomponente und der Regeln für die Transformation von Qualitätsattributen beschreiben.

Diese Ausdrücke, welche nach den Regeln domänenspezifischen Sprache erstellt werden, werden als Qualitätseffekt-Spezifikation (englisch: quality effect specifications; kurz: QES) bezeichnet.

Bei der Qualitätseffekt-Transformation (Abschnitt 4.3) werden die Regeln für Abfragen und Transformationen, welche mittels der domänenspezifischen Sprache spezifiziert worden sind, ausgewertet und angewendet. Hierbei wird bei der Qualitätseffekt-Transformation von Qualitätseffekt-Spezifikationen im ersten Schritt mithilfe der so beschriebenen Anfragen, Elemente aus einem komponentenbasierten Softwaresystem bestimmt, auf welche im zweiten Schritt die Transformationen angewendet werden.

4.2.1. Xtext-Rahmenwerk

Zu diesem Zweck wird die domänenspezifische Sprache zur Qualitätseffekt-Spezifikation durch das Open-Source-Framework Xtext (Abschnitt 3.4.1) entwickelt. Die Gründe hierfür sind unter anderem auch, dass das Xtext-Framework alle Aspekte einer vollständigen Sprachinfrastruktur abdeckt. Vor allem jedoch wird die Grammatik der domänenspezifischen Sprache mithilfe von Ausdrücken in einer erweiterten Backus-Naur-Form (EBNF) beschrieben. Die erweiterte Backus-Naur-Form ist eine allgemeine Notationstechnik für kontextfreie Grammatiken, welche häufig zur Beschreibung der Syntax von Programmiersprachen verwendet wird Abschnitt 3.4.2. Daher ist die erweiterte Backus-Naur-Form nicht an die Verwendung bestimmter Technologien, wie beispielsweise Java, gebunden. Da die Beschreibung der Grammatik in der erweiterten Backus-Naur-Form definiert ist, könnten auch andere Rahmenwerke zur Umsetzung der domänenspezifischen Sprache verwendet werden.

```
1 grammar org.palladiosimulator.qes.QualityEffectSpecification
2     with org.eclipse.xtext.common.Terminals
3
4 generate qualityEffectSpecification
5     "http://www.palladiosimulator.org/qes/QualityEffectSpecification"
```

Xtext 1: Domänenspezifische Sprache

In Listing 1 ist die Definition des Domänenmodells der Xtext-Grammatik abgebildet, welches die domänenspezifische Sprache konzipiert, mit welcher Abfragen und Transformationen beschrieben werden können. Jede Xtext-Grammatik beginnt mit einer Überschrift, die einige Eigenschaften der Grammatik definiert. Die erste Zeile gibt hierbei `QualityEffectSpecification` als den Namen der Sprache an und für die Paketdeklaration diese Sprache wird dabei `org.palladiosimulator.qes` definiert. Xtext nutzt für den Namen einer Sprache den Klassenpfadmechanismus von Java, sodass der Name ein beliebiger gültiger Java-Bezeichner sein kann.

Die zweite Zeile gibt hierbei die Beziehung zu anderen Sprachen an. Eine Xtext-Grammatik kann eine andere bereits vorhandene Grammatik wiederverwenden. So verwenden Xtext-Grammatiken, standardmäßig die `org.eclipse.xtext.common.Terminals`-Grammatik, welche einen grundlegenden Satz von Terminalsymbolen definiert. Unter anderem

werden so Integer (`INT`), Zeichenketten (`STRING`) und Identifikatoren (`ID`) definiert und sind hierdurch in der eigenen Xtext-Grammatik verwendbar. In Abschnitt 4.2.2 werden noch zwei weitere Terminalsymbole in der Grammatik für die Sprache deklariert.

Die vierte und fünfte Zeile weisen Xtext dazu an, ein Ecore-Modell aus der Grammatik abzuleiten. Xtext generiert so automatisch alle benötigten Ecore-Klassen mit den entsprechenden Attributen und den zugehörigen Referenzen für die verschiedenen Parser-Regeln in der Grammatik. So generiert Xtext für diese Grammatik ein Ecore-Paket mit dem Namen `qualityEffectSpecification` und dem einheitlichen Bezeichner für Ecore-Ressourcen (URI) `http://www.palladiosimulator.org/qes/QualityEffectSpecification`.

In Abschnitt 4.2.3 wird der konkrete Entwurf der komplexen fachlichen Zusammenhänge basierend auf einem Modell der Anwendungsdomäne, dem sogenannten Domänenmodell, gezeigt.

4.2.2. Terminalsymbole

Zusätzlich zu den standardmäßigen Terminalsymbolen von Xtext werden noch zwei weitere Terminalsymbole in der Grammatik für die Sprache deklariert. In Xtext gibt die Namenskonvention vor, dass die Namen von Terminalregeln vollständig in Großbuchstaben geschrieben werden. In Listing 2 ist die Definition dieser zwei Terminalsymbole abgebildet.

```
91 terminal NUMBER:
92     INT ('.' INT)?;
93
94 terminal NL:
95     ('\r'? '\n');
```

Xtext 2: Terminalsymbole

Das Terminalsymbol `NUMBER` ist das Symbol für eine Gleitkommazahl. Hierbei kann eine Gleitkommazahl entweder nur als ganzzahliger Wert (Integer) dargestellt werden oder optional auch mit Nachkommastellen. An dieser Stelle werden die Nachkommastellen wieder als ganzzahliger Wert (Integer) dargestellt und werden durch einen Punkt („.“) begonnen.

Das Terminalsymbol `NL` ist das Symbol für einen Zeilenumbruch. An dieser Stelle kann ein Zeilenumbruch im Unix-Format („`\n`“) oder im Windows-Format („`\r\n`“) dargestellt werden.

4.2.3. Domänenmodell

Die erste Produktionsregel einer Xtext-Grammatik wird immer als Startregel verwendet. In Listing 3 ist die Definition dieser Startregel für die Grammatik abgebildet. Diese Startregel leitet die Definition des Domänenmodells ein, welches die komplexen fachlichen Zusammenhänge der Abfragen und Transformationen modelliert.

```
7 Model:
8     specifications+=QualityEffectSpecification
9     (NL+ specifications+=QualityEffectSpecification)* NL+;
```

Xtext 3: Domänenmodell

Die Zeilen sieben bis neun besagen, dass ein Domänenmodell mindestens eine Qualitätseffekt-Spezifikation („`Quality Effect Specification`“) enthält. Ein Domänenmodell kann hierbei auch mehre Qualitätseffekt-Spezifikationen enthalten, welche dann durch einen oder mehrere Zeilenumbrüche voneinander getrennt werden.

4.2.4. Qualitätseffekt-Spezifikation

Eine Qualitätseffekt-Spezifikation („`Quality Effect Specification`“) gibt an, für welche Komponenten welche Transformationen ausgeführt werden sollen. Zu diesem Zweck beinhaltet eine Qualitätseffekt-Spezifikation in ihrem ersten Block („`For{}`“) die Spezifikation der Komponenten, auf welche die im zweiten Block („`Do{}`“) spezifizierten Transformationen angewendet werden sollen. Hierbei erfolgt die Komponenten-Spezifikation im `For`-Block mittels der ComponentSpecification-Produktionsregel und die Transformations-Spezifikationen im `Do`-Block mittels der TransformationSpecification-Produktionsregel.

```
11 QualityEffectSpecification:
12     'For' '{' components+=ComponentSpecification
13     ('and' components+=ComponentSpecification)* '}'
14     'Do' '{' transformations+=TransformationSpecification
15     ('and' transformations+=TransformationSpecification)* '}';
```

Xtext 4: Qualitätseffekt-Spezifikation

In Listing 4 ist die Definition der Produktionsregel für die Qualitätseffekt-Spezifikation mit ihren zugehörigen Komponenten-Spezifikationen (Abschnitt 4.2.5) und Transformations-Spezifikationen (Abschnitt 4.2.6) abgebildet. Eine Qualitätseffekt-Spezifikation beinhaltet in ihrem `For`-Block ein oder mehre Komponenten-Spezifikationen. Hierbei werden mehrere Komponenten-Spezifikationen durch das Schlüsselwort `and` voneinander abgetrennt. Eine Komponenten-Spezifikation beschreibt die strukturellen Eigenschaften von Softwarekomponenten eines komponentenbasierten Softwaresystems, auf welche die folgenden Transformationen angewendet werden sollen. Eine Qualitätseffekt-Spezifikation beinhaltet in ihrem `Do`-Block ein oder mehrere Transformations-Spezifikationen. Hierbei werden diese Transformations-Spezifikationen ebenfalls durch das Schlüsselwort `and` voneinander abgetrennt. Eine Transformations-Spezifikation beschreibt die Transformationen, welche auf die im vorangegangenen `For`-Block beschriebenen Softwarekomponenten angewendet werden sollen.

In den folgenden Unterabschnitten werden die verschiedenen Konzepte und syntaktischen Konstrukte der Komponenten-Spezifikationen (Abschnitt 4.2.5) und Transformations-Spezifikationen (Abschnitt 4.2.6) erläutert.

4.2.5. Komponenten-Spezifikation

Mit der Produktionsregel für die Komponenten-Spezifikation sollen sich die strukturellen Eigenschaften einer Komponente in einem Softwaresystem beschreiben lassen. Hierbei dienen diese zu der Bestimmung der Komponenten, auf welche die Transformationen angewendet werden sollen. Eine Komponenten-Spezifikation beschreibt hierzu die strukturellen Eigenschaften als semantisch Klassen von architekturdefinierten Einflussfaktoren der Komponenten in einem komponentenbasierten Softwaresystem.

Bei der Wahl dieser strukturellen Eigenschaften, welche zum Bestimmen von Komponenten eingesetzt werden, wurde sich an den strukturellen Eigenschaften in komponentenbasierten Systemen orientiert, welche zuvor in Abschnitt 4.1 beschrieben wurden. So können solche Eigenschaften beispielsweise der Name einer Komponente oder auch die Betriebsmittel einer Komponente sein. Die Anforderungen an die domänenspezifische Sprache sind zum einen, dass durch diese Sprache möglichst unkompliziert die Eigenschaften der Elemente eines komponentenbasierten Softwaresystems beschrieben werden können. Zum anderen sollen auch alle strukturellen Eigenschaften in komponentenbasierten Systemen beschrieben werden können. Aus diesen zwei Gründen werden für die unterschiedlichen strukturellen Eigenschaften in komponentenbasierten Systemen jeweils eine eigene Produktionsregel in der Grammatik der Sprache definiert. Dies erlaubt es, die strukturellen Eigenschaften in komponentenbasierten Systemen möglichst benutzerfreundlich zu beschreiben. In der Tabelle 4.1 ist der Zusammenhang zwischen den strukturellen Eigenschaften in komponentenbasierten Systemen und den zugehörigen Produktionsregeln in der Grammatik zur Komponenten-Spezifikation dargestellt.

Eigenschaft	Strukturelle Eigenschaft	Produktionsregel
Name	Abschnitt 4.1.1	Abschnitt 4.2.5.1
Identifikator	Abschnitt 4.1.2	Abschnitt 4.2.5.2
Annotation	Abschnitt 4.1.3	Abschnitt 4.2.5.3
Komponenten-Typ	Abschnitt 4.1.4	Abschnitt 4.2.5.4
Rolle	Abschnitt 4.1.5	Abschnitt 4.2.5.5
Komposition	Abschnitt 4.1.6	Abschnitt 4.2.5.6
Betriebsmittel	Abschnitt 4.1.7	Abschnitt 4.2.5.7

Tabelle 4.1.: Elemente der Komponenten-Spezifikation

In Listing 5 ist die Definition der Produktionsregel für die Komponenten-Spezifikationen in der Grammatik der domänenspezifischen Sprache abgebildet. Eine Komponenten-Spezifikation beginnt mit dem Schlüsselwort `Component`, gefolgt von einem Block, welcher durch eine öffnende Klammer („`(`“) begonnen wird und auch wieder durch eine schließende Klammer („`)`“) geschlossen wird. In diesem Block werden die strukturellen Eigenschaften („`ComponentProperty`“), welche inTabelle 4.1 aufgelistet werden, einer Softwarekomponente beschrieben.

```
17 ComponentSpecification:
18     'Component' '(' properties+=ComponentProperty
19     ('and' properties+=ComponentProperty)* ')';
20
21 ComponentProperty:
22     Name | Identifier | Annotation | Type | Role | Assembly | Resource;
```

Xtext 5: Produktionsregel für Komponenten-Spezifikationen

Um möglichst genau eine Komponente in einem komponentenbasierten Softwaresystem beschreiben zu können, können in einer Komponenten-Spezifikation die einzelnen Komponenten-Eigenschaften miteinander kombiniert werden. Hierzu können mehrere Komponenten-Eigenschaften mittels einer Und-Verknüpfung („and“) miteinander verbunden werden. Wenn eine Komponenten-Spezifikation mehrere Komponenten-Eigenschaften beinhaltet, müssen alle Eigenschaften auf eine Komponente zutreffen, damit die anschließenden Transformationen auf diese anwendet werden. In den folgenden sieben Unterabschnitten werden die Produktionsregeln für diese Komponenten-Eigenschaften einzeln beschrieben.

4.2.5.1. Name

Die Produktionsregel `Name` in Listing 6 spezifiziert die Entität in der Grammatik zur Deklaration eines Namens. Die Produktionsregel beginnt mit dem Schlüsselwort `Name` gefolgt von einem Block, welcher durch eine öffnende Klammer („`(`“) begonnen wird und auch wieder durch eine schließende Klammer („`)`“) geschlossen wird.

```
24 Name:
25     'Name' '(' not?='not'? autonym=STRING ')';
```

Xtext 6: Produktionsregel für Namen

Innerhalb des Blocks wird hierbei der Name eines benannten Elements durch die Verwendung einer Zeichenkette angegeben. Hierbei wird die Zeichenkette durch das `STRING`-Terminal repräsentiert. Zusätzlich erlaubt die optionale Verwendung des Schlüsselworts `not` vor dieser Zeichenkette diese Aussage zu verneinen.

Beispielsweise würde die Komponenten-Spezifikation `Component(Name("Database"))` alle Komponenten klassifizieren, welche den Namen „Database“ haben. Im Gegenzug würde die Komponenten-Spezifikation `Component(Name(not "Database"))` alle Komponenten klassifizieren, welche einen anderen Namen als „Database“ haben.

4.2.5.2. Identifikator

Die Produktionsregel `Identifier` in Listing 7 spezifiziert die Entität in der Grammatik zur Deklaration eines Identifikators. Die Produktionsregel beginnt mit dem Schlüsselwort

`Identifier` gefolgt von einem Block, welcher durch eine öffnende Klammer („`(`“) begonnen wird und auch wieder durch eine schließende Klammer („`)`“) geschlossen wird.

```
27 Identifier:
28     'Id' '(' not?='not'? id=STRING ')';
```

Xtext 7: Produktionsregel für Identifikatoren

Innerhalb des Blocks wird hierbei der Identifikator eines Elements durch die Verwendung einer Zeichenkette angegeben. Hierbei wird die Zeichenkette durch das `STRING`-Terminal repräsentiert. Zusätzlich erlaubt die optionale Verwendung des Schlüsselworts `not` vor der Zeichenkette diese Aussage zu verneinen.

Beispielsweise würde die Komponenten-Spezifikation `Component(Identifier("#9682"))` alle Komponenten klassifizieren, welche den Identifikator „#9682“ haben. Im Gegenzug würde die Komponenten-Spezifikation `Component(Identifier(not "#9682"))` alle Komponenten klassifizieren, welche einen anderen Identifikator als „#9682“ haben.

4.2.5.3. Annotation

Die Produktionsregel `Annotation` in Listing 8 spezifiziert die Entität in der Grammatik zur Deklaration einer Annotation. Die Produktionsregel beginnt mit dem Schlüsselwort `Annotation` gefolgt von einem Block, welcher durch eine öffnende Klammer („`(`“) begonnen wird und auch wieder durch eine schließende Klammer („`)`“) geschlossen wird.

```
30 Annotation:
31     'Annotation' '(' not?='not'? annotation=STRING ')';
```

Xtext 8: Produktionsregel für Annotationen

Innerhalb des Blocks wird hierbei der Name einer Annotation, welche ein Element annotiert, durch die Verwendung einer Zeichenkette angegeben. Hierbei wird die Zeichenkette durch das `STRING`-Terminal repräsentiert. Zusätzlich erlaubt die optionale Verwendung des Schlüsselworts `not` vor der Zeichenkette diese Aussage zu verneinen.

Beispielsweise würde die Komponenten-Spezifikation `Component(Annotation("AJP"))` alle Komponenten klassifizieren, welche mit einer Annotation mit dem Text „AJP“ annotiert wurden. Im Gegenzug würde die Komponenten-Spezifikation `Component(Annotation(not "AJP"))` alle Komponenten klassifizieren, welche nicht mit einer Annotation mit dem Text „AJP“ annotiert wurden.

4.2.5.4. Komponenten-Typ

Die Produktionsregel `Type` in Listing 9, Zeile 32 und 33, spezifiziert die Entität in der Grammatik zur Deklaration des Komponenten-Typs. Die Produktionsregel beginnt mit dem Schlüsselwort `Type` gefolgt von einem Block, welcher durch eine öffnende Klammer

(„(“) begonnen wird und auch wieder durch eine schließende Klammer („)“) geschlossen wird.

```
32 Type:
33     'Type' '(' not?='not'? type=ComponentType ')';
34 
35 enum ComponentType:
36     ANY='AnyComponentType' | BASIC='Basic' | COMPOSITE='Composite';
```

Xtext 9: Produktionsregel für den Komponenten-Typ

Innerhalb des Blocks wird hierbei der Typ einer Komponente durch die Verwendung eines Aufzählungstyps angegeben. Hierfür wird in Listing 9, Zeile 35 und 36, der Aufzählungstyp `ComponentType` spezifiziert. Der Aufzählungstyp kann dazu verwendet werden, um eine Komponente nach ihrem Typ zu differenzieren. Hierbei gibt das Element `Basic` an, dass für eine Komponente lediglich eine externe Ansicht existiert. Während hingegen das Element `Composite` angibt, dass für eine Komponente auch eine interne Ansicht existiert. Das Element `AnyComponentType` gibt schließlich an, dass der Typ einer Komponente beliebig ist.

Beispielsweise würde die Komponenten-Spezifikation `Component(Type(Composite))` alle Komponenten klassifizieren, für welche eine interne Ansicht („White-Box“) spezifiziert ist. Im Gegenzug würde die Komponenten-Spezifikation `Component(Type(Basic))` alle Komponenten klassifizieren, für welche nur eine externe Ansicht („Black-Box“) spezifiziert ist.

4.2.5.5. Rolle

Die Produktionsregel `Role` in Listing 10, Zeile 38 bis 40, spezifiziert die Entität in der Grammatik zur Deklaration der Schnittstelle einer Komponente. Die Produktionsregel beginnt mit dem Schlüsselwort `Role` gefolgt von einem Block, welcher durch eine öffnende Klammer („(“) begonnen wird und auch wieder durch eine schließende Klammer („)“) geschlossen wird.

```
38 Role:
39     'Role' '(' not?='not'? type=RoleType ('with'
40     properties+=RoleProperty ('and' properties+=RoleProperty)*)? ')';
41
42 enum RoleType:
43     ANY='AnyRoleType' |
44     COMPONENT_REQUIRED_PROVIDED='ComponentRequiredProvided' |
45     COMPONENT_REQUIRED='ComponentRequired' |
46     COMPONENT_PROVIDED='ComponentProvided' |
47     INFRASTRUCTURE_REQUIRED_PROVIDED='InfrastructureRequiredProvided' |
48     INFRASTRUCTURE_REQUIRED='InfrastructureRequired' |
49     INFRASTRUCTURE_PROVIDED='InfrastructureProvided';
50
51 RoleProperty:
52     Name | Identifier | Annotation;
```

Xtext 10: Produktionsregel für die Rolle von Komponenten

Innerhalb des Blocks wird hierbei der Typ einer Schnittstelle durch die Verwendung eines Aufzählungstyps angegeben. Hierfür wird in Listing 10, Zeile 42 bis 49, der Aufzählungstyp `RoleType` spezifiziert. Der Aufzählungstyp kann dazu verwendet werden, um eine Schnittstelle nach ihrem Typ zu differenzieren. Zum einen lässt sich mittels des Aufzählungstyps differenzieren, ob es sich bei einer Schnittstelle um eine angebotene (`Provided`) oder erforderliche (`Required`) Schnittstelle einer Komponente handelt. Zum anderen lässt sich auch mittels des Aufzählungstyps differenzieren, ob eine Schnittstelle einer Komponente (`Component`) oder einer Infrastruktur (`Infrastructure`) zugeordnet ist. Das Element `AnyRoleType` gibt schließlich an, dass der Typ einer Schnittstelle beliebig ist.

Zusätzlich zur Spezifizierung des Typs einer Schnittstelle können noch weitere Eigenschaften zum genauen Differenzieren mit angegeben werden. So kann zusätzlich neben dem Typ noch der Name (Abschnitt 4.2.5.1), ein Identifikator (Abschnitt 4.2.5.2) und/oder eine Annotation (Abschnitt 4.2.5.3) zur Klassifikation einer Schnittstelle mit angegeben werden. Hierbei können in einer Rollen-Spezifikation die einzelnen Eigenschaften miteinander kombiniert werden. Hierzu können mehrere Rollen-Eigenschaften mittels einer Und-Verknüpfung („`and`“) miteinander verbunden werden. Wenn eine Rollen-Spezifikation mehrere Eigenschaften beinhaltet, müssen alle Eigenschaften auf eine Schnittstelle zutreffen, damit die anschließenden Transformationen auf diese angewendet werden.

Beispielsweise würde die Komponenten-Spezifikation `Component(Role(ComponentProvided with Name("IDatabase")))` alle Komponenten klassifizieren, welche eine angebotene Schnittstelle mit dem Namen "IDatabase" haben. Des Weiteren würde beispielsweise die Komponenten-Spezifikation `Component(Role(InfrastructureRequiredProvided))` alle Komponenten klassifizieren, welche eine angebotene oder erforderliche Schnittstelle haben, die der Infrastruktur zugeordnet ist. Des Weiteren würde beispielsweise die Komponenten-Spezifikation `Component(Role(AnyRoleType with Annotation("AJP")))` alle Komponenten klassifizieren, die eine Schnittstelle haben, welche mit "AJP" annotiert ist.

4.2.5.6. Komposition

Die Produktionsregel `Assembly` in Listing 11, Zeile 54 bis 56, spezifiziert die Entität in der Grammatik zur Deklaration über die Komposition einer Komponente. Die Produktionsregel beginnt mit dem Schlüsselwort `Assembly` gefolgt von einem Block, welcher durch eine öffnende Klammer („`(`“) begonnen wird und auch wieder durch eine schließende Klammer („`)`“) geschlossen wird.

```
54 Assembly:
55     'Assembly' '(' not?='not'? type=AssemblyType
56     ('with' component=ComponentSpecification)? ')';
57
58 enum AssemblyType:
59     ANY='AnyAssembly' | REQUIRED='Required' | PROVIDED='Provided';
```

Xtext 11: Produktionsregel für die Komposition von Komponenten

Innerhalb des Blocks wird hierbei der Typ der Komposition durch die Verwendung eines Aufzählungstyps angegeben. Hierfür wird in Listing 11, Zeile 58 und 59, der Aufzählungstyp `AssemblyType` spezifiziert. Der Aufzählungstyp kann dazu verwendet werden, um die Komposition zweier Komponenten nach ihrem Typ zu differenzieren. Zum einen lässt sich mittels des Aufzählungstyps differenzieren, ob es sich bei einer Komponente um eine verbundene angebotene (`Provided`) oder eine verbundene erforderliche (`Required`) Schnittstelle zu einer Komponente handelt. Zum anderen lässt sich auch mittels des Elements `AnyAssembly` angeben, dass der Typ der Schnittstelle zu einer anderen Komponente beliebig ist.

Zusätzlich zum Typ der Komposition lässt sich durch das Schlüsselwort `with` ein weiterer Block zur Komponenten-Spezifikation anfügen. Dies Komponenten-Spezifikation dient der Deklaration, mit welcher eine Komponente mit einer anderen Komponente zusammengebaut ist. Hierfür wird die Produktionsregel für die Komponenten-Spezifikationen, welche in Abschnitt 4.2.5 beschrieben ist, wiederverwendet.

Beispielsweise würde die Komponenten-Spezifikation `Component(Assembly(Required with Component(Name("Database"))))` alle Komponenten klassifizieren, welche über ihre erforderliche (`Required`) Schnittstelle mit einer Komponente mit dem Namen „Database“ verbunden ist. Die Komponenten-Spezifikation `Component(Assembly(Required with Component(Type(Composite))))` alle Komponenten klassifizieren, die über ihre angebotene (`Provided`) Schnittstelle mit einer Komponente verbunden sind, für welche eine interne Ansicht spezifiziert ist. Und die Komponenten-Spezifikation `Component(Assembly(not AnyAssembly))` würde alle Komponenten klassifizieren, welche mit keiner anderen Komponente verbunden sind.

4.2.5.7. Betriebsmittel

Die Produktionsregel `Resource` in Listing 12, Zeile 61 bis 63, spezifiziert die Entität in der Grammatik zur Deklaration des Servers einer Komponenten. Die Produktionsregel beginnt

mit dem Schlüsselwort `Resource` gefolgt von einem Block, welcher durch eine öffnende Klammer („`(`“) begonnen wird und auch wieder durch eine schließende Klammer („`)`“) geschlossen wird.

```
61 Resource:
62     'Resource' '(' properties+=ResourceProperty
63     ('and' properties+=ResourceProperty)* ')';
64
65 ResourceProperty:
66     Name | Identifier;
```

Xtext 12: Produktionsregel für das Betriebsmittel von Komponenten

Innerhalb des Blocks wird hierbei der Typ des Servers einer Komponente durch den Namen (Abschnitt 4.2.5.1) oder den Identifikator (Abschnitt 4.2.5.2) des Servers angegeben. Hierbei können in einer Resource-Spezifikation die einzelnen Eigenschaften miteinander kombiniert werden. Hierzu können mehrere Eigenschaften von Servern miteinander mittels einer Und-Verknüpfung („`and`“) verbunden werden. Wenn eine Betriebsmittel-Spezifikation mehrere Eigenschaften beinhaltet, müssen alle Eigenschaften auf einen Server zutreffen, damit die anschließenden Transformationen auf diese angewendet werden.

Beispielsweise würde die Komponenten-Spezifikation `Component(Resource(Name(not "Server 1")))` alle Komponenten klassifizieren, welche nicht auf einem Server mit dem Namen „Server 1“ verteilt sind. Und die Komponenten-Spezifikation `Component(Resource(Identifier("#9682")))` würde alle Komponenten klassifizieren, welche auf einem Server verteilt sind, der den Identifikator „#9682“ hat.

4.2.6. Transformations-Spezifikation

Mit einer Transformations-Spezifikation lassen sich Transformationen an Qualitätsattributen von Softwarekomponenten beschreiben. Hierbei dienen Transformations-Spezifikationen zur Abänderung von Qualitätsattributen einer Komponente, auf welche die zuvor beschriebenen Eigenschaften passen. Hierzu beschreiben Transformationen, wie diese Qualitätsattribute einer Komponente verändert werden sollen. Da eine mit dieser domänenspezifischen Sprache beschriebene Transformation immer nur auf die durch die vorangegangenen Komponenten-Spezifikationen beschriebenen Komponenten angewendet wird, können diese folglich auch keine anderen Elemente eines Softwaresystems verändern.

Die Produktionsregel `TransformationSpecification` in Listing 13 spezifiziert die Entität in der Grammatik zur Deklaration der verschiedenen Transformationen. Eine Qualitätseffekt-Spezifikation (Abschnitt 4.2.4) beinhaltet in ihrem `Do`-Block eine oder mehrere solcher Transformations-Spezifikationen.

```
67 TransformationSpecification:
68     NQA | Reasoning | NumericValue;
```

Xtext 13: Produktionsregel für die Transformations-Spezifikation

Solche Transformationen können die Veränderung von nicht-quantifizierten Qualitätsmerkmalen einer Softwarekomponente (Abschnitt 4.2.6.1), Regeln wie ein nicht-quantifiziertes Qualitätsmerkmal von anderen Komponente beeinflusst wird (Abschnitt 4.2.6.2) oder allgemeine Werte von numerischen Variablen Abschnitt 4.2.6.3) sein. In den folgenden drei Unterabschnitten werden die Produktionsregeln für diese drei Transformations-Spezifikationen einzeln beschrieben.

4.2.6.1. NQA

Ein nicht-quantifiziertes Qualitätsmerkmal (Abschnitt 3.3.6.1) repräsentiert eine relevante Qualitätseigenschaft einer Softwarekomponente. Es ist dargestellt durch eine Qualitätsdimension (`quality`) und das entsprechende Dimensionselement (`element`). Die Qualitätsdimension und das Dimensionselement werden in der domänenspezifischen Sprache durch jeweils eine Zeichenkette dargestellt.

Die Produktionsregel für eine NQA-Transformation ist in Listing 14, Zeile 70 und 71, spezifiziert und beginnt mit dem Schlüsselwort `NQA` gefolgt von einem Block, welcher durch eine öffnende Klammer („(“) begonnen wird und auch wieder durch eine schließende Klammer („)“) geschlossen wird.

```
70 NQA:
71     'NQA' '(' quality=STRING type=TransformationType element=STRING ')';
72
73 enum TransformationType:
74     IS='=' | PLUS='+' | MINUS='-' | MULTIPLICATION='*' | DIVISION='/';
```

Xtext 14: Produktionsregel für die NQA-Transformationen

Für die Beschreibung einer NQA-Transformation kann neben der Qualitätsdimension und dem Dimensionselement auch noch angeben werden, wie genau dieses Dimensionselement verändert werden soll. Zu diesem Zweck ist in Listing 14, Zeile 73 und 74, der Aufzählungstyp `TransformationType` spezifiziert. Dieser Aufzählungstyp deklariert fünf verschiedene Operationen wie ein Dimensionselement genau transformiert wird. Das Element Gleichheitszeichen (=) des Aufzählungstyps deklariert, dass für die mitangegebene Qualitätsdimension das entsprechende Dimensionselement gesetzt wird. Hierbei wird ein möglicherweise zuvor existierendes Dimensionselement überschrieben, ohne dieses zu berücksichtigen.

Durch die Verwendung der Elemente für die vier Grundrechenarten (Addition (+), Subtraktion (-), Multiplikation (∗) und Division (/)) werden auch zuvor existierende Dimensionselemente berücksichtigt. Da es sich bei den Dimensionselementen um Kategorien einer Ordinalskala handelt, können diese Kategorien auch durch Zahlen kodiert werden.

Hierdurch werden mathematische Operationen, wie die vier Grundrechenarten, mit diesen Zahlen (Dimensionselementen) möglich. Hierbei werden für mathematische Operationen mit Dimensionselementen diese zunächst durch Zahlen kodiert, auf welchen anschließend die Operationen angewendet werden, abschließend werden die Zahlen des Ergebnisses wieder den Kategorien zugeordnet.

Beispielsweise würde die Transformations-Spezifikation `NQA("Usability" = "+")` das nicht-quantifizierte Qualitätsmerkmal `(Usability, +)` einer Komponente zuordnen. Semantisch würde dies zum Ausdruck bringen, dass eine Softwarekomponente eine vergleichsweise „gute“ Gebrauchstauglichkeit hat. Und die Transformations-Spezifikation `NQA("Security" = "-")` würde das entsprechende nicht-quantifizierte Qualitätsmerkmal einer Komponente zuordnen und so zum Ausdruck bringen, dass eine Softwarekomponente eine vergleichsweise „schlechte“ Informationssicherheit hat.

4.2.6.2. Reasoning

Ein Reasoning (englisch für Schlussfolgerung) dient in der domänenspezifischen Sprache zur Abbildung des Kerns der symbolischen nicht-numerischen Berechnung für nicht-quantifizierte Qualitätsmerkmale. Eine Reasoning-Transformation dient zur Spezifikation eines Abbildungsregelwerks (Abschnitt 3.3.6.2) aus dem Ansatz zur qualitativen Argumentation (Abschnitt 3.3.6) und ermöglicht so, mittels der domänenspezifischen Sprache Transformationen für diese Abbildungsregelwerke zu definieren.

Die Produktionsregel für eine Reasoning-Transformation ist in Listing 15, Zeile 76 bis 78, spezifiziert und beginnt mit dem Schlüsselwort `Reasoning` gefolgt von einem Block, welcher durch eine öffnende Klammer („`(`“) begonnen wird und auch wieder durch eine schließende Klammer („`)`“) geschlossen wird. Die Produktionsregel für ein Rule-Element ist in Zeile 80 bis 82 spezifiziert und beginnt mit dem Schlüsselwort `Rule` gefolgt von einem Block, welcher durch eine öffnende Klammer begonnen wird und auch wieder durch eine schließende Klammer geschlossen wird. Die Produktionsregel für ein Entry-Element ist in Zeile 84 und 85 spezifiziert und beginnt mit dem Schlüsselwort `Entry` gefolgt von einem Block, welcher durch eine öffnende Klammer begonnen wird und auch wieder durch eine schließende Klammer geschlossen wird.

```
76 Reasoning:
77     'Reasoning' '(' quality=STRING '='
78     rules+=Rule (',' rules+=Rule)* ')';
79
80 Rule:
81     'Rule' '(' qualities+=STRING (',' qualities+=STRING)* '='
82     entries+=Entry (',' entries+=Entry)* ')';
83
84 Entry:
85     'Entry' '(' key+=STRING (',' key+=STRING)* '=' value=STRING ')';
```

Xtext 15: Produktionsregel für die Reasoning-Transformationen

Mit einer Reasoning-Transformation wird spezifiziert, wie eine Qualitätsdimension einer Komponente von einem nicht-quantifizierten Qualitätsmerkmal (NQA) einer anderen Komponente beeinflusst wird. Zu diesem Zweck besteht eine Reasoning-Transformation aus mehreren Abbildungsregeln (`rules`) und einer Qualitätsdimension (`quality`), für welche diese Regeln gelten. Alle Qualitätsdimensionen werden in einer Reasoning-Transformation durch jeweils eine Zeichenkette dargestellt. Eine Abbildungsregel (`Rule`) spezifiziert das resultierende Dimensionselement für eine Menge von nicht-quantifizierten Qualitätsmerkmalen. Die Produktionsregel für ein Rule-Element ist hierbei definiert aus einer Folge von Qualitätsdimensionen (`qualities`) und einem Satz von Abbildungsregel-Einträgen (`entries`). Ein Abbildungsregel-Eintrag (`Entry`) spezifiziert eine Folge von Dimensionselementen und ordnet dieser Folge ein einzelnes Dimensionselement zu. Die Produktionsregel für ein Entry-Element ist hierbei definiert aus der Reihenfolge der Dimensionselemente zur Eingabe (`key`) und dem daraus resultierenden Dimensionselement zur Ausgabe (`value`).

Beispielsweise würde die Transformations-Spezifikation `Reasoning("Reliability" = Rule("FaultTolerance" = Entry("+"="+"), Entry("0"="0"), Entry("-"="-")))` dieses Reasoning (Abbildungsregelwerk) einer Komponente zuordnen. Semantisch würde dies zum Ausdruck bringen, dass die Zuverlässigkeit (Reliability) der Komponente direkt von der Fehlertoleranz (FaultTolerance) einer anderen Komponente beeinflusst wird. Und die Transformations-Spezifikation `Reasoning("Maintainability" = Rule("Testability", "Modifiability" = Entry("+", "+" ="+")))` würde dieses Reasoning einer Komponente zuordnen. Semantisch würde dies zum Ausdruck bringen, dass die Wartbarkeit (Maintainability) der Komponente positiv beeinflusst wird, wenn sowohl die Testbarkeit (Testability) als auch die Modifizierbarkeit (Modifiability) einer anderen Komponente „gut“ (+) sind.

4.2.6.3. Numeric

Mittels einer Numeric-Transformation kann eine Transformation für den Wert einer numerischen Variable beschrieben werden. Hierdurch kann eine Veränderung an numerischen Attributen einer Softwarekomponente, wie beispielsweise an ihren Kosten, beschrieben werden. Mit einer Numeric-Transformation kann die Veränderung eines numerischen Attributes einer Softwarekomponente in Abhängigkeit der Eigenschaften dieser Komponente in ihrem komponentenbasierten Softwaresystem definiert werden.

Die Produktionsregel für eine Numeric-Transformation ist in Listing 16, Zeile 87 bis 89, spezifiziert. Die Produktionsregel beginnt mit einem beliebigen Schlüsselwort gefolgt von einem Block, welcher durch eine öffnende Klammer („`(`“) begonnen wird und auch wieder durch eine schließende Klammer („`)`“) geschlossen wird. Dieses Schlüsselwort `valueType` bestimmt hierbei den Namen des Attributs einer Softwarekomponente, auf welche die in den Klammern beschriebene Transformation angewendet werden soll.

```
87 NumericValue:
88     valueType=ID '(' transformationType=TransformationType
89     transformationNumber=NUMBER ')';
```

Xtext 16: Produktionsregel für die Numeric-Transformationen

Innerhalb des Blocks wird hierbei ein numerischer Wert durch die Verwendung eines NUMBER-Terminalsymbol (Abschnitt 4.2.2) angegeben. Dieser numerische Wert bestimmt unter anderem, um wie viel ein numerisches Attribut einer Softwarekomponente durch die Transformation verändert wird. Für die Beschreibung einer Numeric-Transformation kann neben dem Namen des Attributs, das transformiert werden soll, und dem numerischen Wert auch noch angeben werden, wie genau dieses Attribut um den angegebenen Wert transformiert werden soll. Zu diesem Zweck wird ebenfalls der Aufzählungstyp `TransformationType` (Listing 14, Zeile 73 und 74) verwendet. Dieser Aufzählungstyp deklariert fünf verschiedene Operationen wie ein numerisches Attribut einer Softwarekomponente genau verändert wird. Das Element Gleichheitszeichen (=) des Aufzählungstyps deklariert, dass für den angegebenen Namen des Attributs der entsprechende angegebene numerische Wert gesetzt wird. Hierbei wird ein zuvor existierender numerischer Wert überschrieben, ohne diesen zu berücksichtigen. Durch die Verwendung der Elemente für die vier Grundrechenarten (Addition (+), Subtraktion (-), Multiplikation (∗) und Division (/)) werden auch zuvor existierende numerische Attribute einer Softwarekomponente berücksichtigt. So wird bei der Verwendung eines Elementes einer der vier Grundrechenarten aus dem Aufzählungstyp der mitangegebene numerische Wert von rechts auf das zuvor existierende numerische Attribut einer Softwarekomponente gerechnet.

Beispielsweise würde die Transformations-Spezifikation `InitialCost(* 1.5)` die einmaligen Initialkosten für eine Komponente um 50 % erhöhen und die Transformations-Spezifikation `OperatingCost(/ 2)` die laufenden Betriebskosten für eine Komponente halbieren.

4.2.7. Anwendungsfälle für Qualitätseffekt-Spezifikationen

In den folgenden Unterabschnitten werden sechs konkrete Anwendungsfälle für den Einsatz von Qualitätseffekt-Spezifikationen gezeigt. Diese sechs Anwendungsfälle stehen stellvertretend für typische Szenarien, welche eintreten können, wenn ein Anwender versucht, mithilfe von Qualitätseffekt-Spezifikationen ein bestimmtes fachliches Ziel zu erreichen. In den folgenden sechs Unterabschnitten wird jeweils zunächst der Anwendungsfall beschrieben, an welcher Stelle Qualitätsannotationen in Abhängigkeit einer konkreten Softwarearchitektur parametrisiert werden sollen. Anschließend wird für jeden der sechs Anwendungsfälle eine Qualitätseffekt-Spezifikation gezeigt, mit welcher eben dieses fachliche Ziel des jeweiligen Anwendungsfalles erreicht werden kann.

Die ersten beiden Anwendungsfälle (Abschnitt 4.2.7.1 und Abschnitt 4.2.7.2) wären auch ohne Qualitätseffekt-Spezifikationen erreichbar, da die entsprechenden Softwarekomponenten lediglich anhand ihrer statischen Eigenschaften bestimmt werden. Hierbei beziehen sich statische Eigenschaften auf die Eigenschaften einer Softwarekomponente, welche nicht direkt aus den strukturellen Eigenschaften ihres Softwaresystems abgeleitet werden müssen. Beispielsweise wäre der Name einer Softwarekomponente eine statische Eigenschaft dieser Komponente und der Komposition dieser Komponente mit einer anderen Komponente wäre keine statische Eigenschaft.

Die restlichen vier Anwendungsfälle (Abschnitt 4.2.7.3 bis Abschnitt 4.2.7.6) waren ohne den Ansatz der Qualitätseffekt-Spezifikation bisher nicht zu modellieren, da die

Softwarekomponenten, auf welche die Transformationen angewendet werden sollen, auf Grundlage ihrer strukturellen Eigenschaften im Softwaresystem bestimmt werden.

4.2.7.1. Anwendungsfall: Kosten

Der erste Anwendungsfall beschreibt das Transformieren der Initialkosten und Betriebskosten einer Softwarekomponente. Dieser Anwendungsfall wäre auch ohne Qualitätseffekt-Spezifikation erreichbar, da das zugehörige Kostenmodell für die entsprechenden Komponenten abgeändert werden könnte. Jedoch soll mit diesem Anwendungsfall aufgezeigt werden, wie eine Transformation der Kosten einer bestimmten Softwarekomponente beschrieben werden kann. Für alle Softwarekomponenten in einem Softwaresystem, welche über eine angebotene Schnittstelle mit dem Namen „SQL“ verfügen, sollen die Initialkosten um 50 % erhöht und die Betriebskosten halbiert werden.

In Listing 17 wird diese entsprechende Qualitätseffekt-Spezifikation gezeigt, mit welcher eben dieses fachliche Ziel des Anwendungsfalles erreicht werden kann.

```
1 For {
2     Component(
3         Role(ComponentProvided with Name("SQL"))
4     )
5 } Do {
6     InitialCost(* 1.5) and
7     OperatingCost(/ 2)
8 }
```

Xtext 17: Transformation für die Kosten einer Komponente

4.2.7.2. Anwendungsfall: Reasoning

Der zweite Anwendungsfall beschreibt das Transformieren eines Reasoning zu einer Softwarekomponente. Dieser Anwendungsfall wäre auch ohne Qualitätseffekt-Spezifikation erreichbar, da das Reasoning im zugehörigen Modell für die entsprechenden Komponenten hinzufügt werden könnte. Jedoch soll mit diesem Anwendungsfall aufgezeigt werden, wie ein Reasoning beschrieben werden kann und dieses mehreren verschiedenen Softwarekomponenten zugeordnet werden kann. Für alle Softwarekomponenten in einem Softwaresystem, welche entweder den Namen „Webserver“ oder „Webinterface“ haben oder den Identifikator „#FAE614“ besitzen, soll das angegebene Reasoning zugeordnet werden. Semantisch würde dieses angegebene Reasoning zum Ausdruck bringen, dass die Benutzerfreundlichkeit (Usability) dieser Komponenten direkt von der Verfügbarkeit (Availability) einer anderen Komponente beeinflusst wird.

In Listing 18 wird diese entsprechende Qualitätseffekt-Spezifikation gezeigt, mit welcher eben dieses fachliche Ziel des Anwendungsfalles erreicht werden kann.

```
10 For {
11     Component(
12         Name("Webserver")
13     )
14     Component(
15         Name("Webinterface")
16     )
17     Component(
18         Id("#FAE614")
19     )
20 } Do {
21     Reasoning("Usability" =
22         Rule("Availability" =
23             Entry("++" = "++"),
24             Entry("+" = "+"),
25             Entry("0" = "0"),
26             Entry("-" = "-"),
27             Entry("--" = "--")
28         )
29     )
30 }
```

Xtext 18: Transformation für ein Reasoning einer Komponente

4.2.7.3. Anwendungsfall: IDS-Sensor

Der dritte Anwendungsfall beschreibt das Transformieren eines nicht-quantifizierten Qualitätsmerkmals an Softwarekomponenten. Hierbei soll unterschieden werden, ob Softwarekomponenten, welche über eine angebotene der Infrastruktur zugeordnete Schnittstelle verfügen, mit der Annotation „IdsSensor“ versehen sind. Diese Annotation „IdsSensor“ stammt aus dem Strukturmodell (Abschnitt 3.3.4) und deklariert einen Verknüpfungspunkt, an welchem eine Sensor-Softwarekomponente eines Intrusion Detection Systems (IDS) eingefügt werden soll.

Motiviert wird dieser Anwendungsfall dadurch, dass alle angebotenen öffentlichen Schnittstellen eines Systems durch einen Sensor eines Intrusion Detection Systems überwacht werden sollten. Wenn eine angebotene der Infrastruktur zugeordnete öffentliche Schnittstelle durch einen IDS-Sensor überwacht wird, wirkt sich dies positiv auf die Sicherheit des kompletten Softwaresystems aus. Wenn jedoch eine angebotene der Infrastruktur zugeordnete öffentliche Schnittstelle nicht durch einen IDS-Sensor überwacht wird, wirkt sich dies negativ auf die Sicherheit des kompletten Softwaresystems aus.

In Listing 19 sind diese entsprechenden zwei Qualitätseffekt-Spezifikationen gezeigt, mit welchen eben dieses fachliche Ziel des Anwendungsfalles erreicht werden kann.

```
For {
    Component(
        Role(InfrastructureProvided) and
        Annotation("IdsSensor")
    )
} Do {
    NQA("Security" = "+")
}
For {
    Component(
        Role(InfrastructureProvided) and
        Annotation(not "IdsSensor")
    )
} Do {
    NQA("Security" = "-")
}
```

Xtext 19: Transformation für ein nicht-quantifiziertes Qualitätsmerkmal einer Komponente

Für alle Softwarekomponenten in einem Softwaresystem, welche über eine angebotene Infrastruktur-Schnittstelle verfügen und auch noch mit der Annotation „IdsSensor“ versehen sind, soll das entsprechende nicht-quantifizierte Qualitätsmerkmal `NQA("Security" = "+")` zugeordnet werden. Semantisch würde dies zum Ausdruck bringen, dass eine solche Softwarekomponente eine vergleichsweise „gute“ Informationssicherheit hat.

Für alle Softwarekomponenten in einem Softwaresystem, welche über eine angebotene Infrastruktur-Schnittstelle verfügen, jedoch nicht mit der Annotation „IdsSensor“ versehen sind, soll das entsprechende nicht-quantifizierte Qualitätsmerkmal `NQA("Security" = "-")` zugeordnet werden. Semantisch würde dies zum Ausdruck bringen, dass eine solche Softwarekomponente eine vergleichsweise „schlechte“ Informationssicherheit hat.

4.2.7.4. Anwendungsfall: Demilitarisierte Zone

Der vierte Anwendungsfall beschreibt das Transformieren eines nicht-quantifizierten Qualitätsmerkmals an Softwarekomponenten in Abhängigkeit davon, welchem Server eine bestimmte Softwarekomponente zugeordnet ist. Hierbei soll beachtet werden, ob eine Softwarekomponente, welche eine HTTP-Schnittstelle hat, einem DMZ-Server zugeordnet worden ist oder nicht. Dabei bezeichnet die demilitarisierte Zone (DMZ) ein Computernetzwerk mit sicherheitstechnisch kontrollierten Zugriffsmöglichkeiten auf die daran angeschlossenen Server. An dieser Stelle werden die der demilitarisierten Zone zugeordneten Softwarekomponenten durch eine Firewall gegen andere Computernetzwerke abgeschirmt.

Motiviert wird dieser Anwendungsfall dadurch, dass durch eine demilitarisierte Zone der Zugriff auf öffentlich erreichbare Dienste gestattet und gleichzeitig das interne Computernetzwerk vor unberechtigten Zugriffen von außen geschützt werden kann. An dieser

Stelle entfaltet eine demilitarisierte Zone ihre Schutzwirkung durch die Isolation einer oder mehrerer Softwarekomponenten in einem Computernetzwerk.

In Listing 20 wird diese entsprechende Qualitätseffekt-Spezifikation gezeigt, mit welcher eben dieses fachliche Ziel des Anwendungsfalles erreicht werden kann.

```
49 For {
50     Component(
51         Role(AnyRoleType with Name("HTTP")) and
52         Resource(Name(not "DMZ"))
53     )
54 } Do {
55     NQA("Security" = "--")
56 }
```

Xtext 20: Transformation in Abhängigkeit der Bereitstellungsbeziehungen von Komponenten

Für alle Softwarekomponenten in einem Softwaresystem, welche über eine Schnittstelle mit dem Namen „HTTP“ verfügen und nicht einem Server mit dem Namen „DMZ“ zugeordnet sind, soll das nicht-quantifizierte Qualitätsmerkmal `NQA("Security" = "--")` zugeordnet werden. Semantisch würde dies zum Ausdruck bringen, dass eine solche Softwarekomponente eine vergleichsweise „sehr schlechte“ Informationssicherheit hat.

4.2.7.5. Anwendungsfall: Komposition

Der fünfte Anwendungsfall beschreibt erneut das Transformieren eines nicht-quantifizierten Qualitätsmerkmals an Softwarekomponenten in Abhängigkeit davon, ob eine Verbindung zwischen zwei bestimmten Softwarekomponenten besteht. Hierbei werden diese beiden Softwarekomponenten über ihren Identifikator bestimmt. Motiviert wird dieser Anwendungsfall dadurch, dass durch die Komposition zweier bestimmter Softwarekomponenten Änderungen in einem Softwaresystem einfacher durchgeführt werden können. In Listing 21 wird diese entsprechende Qualitätseffekt-Spezifikation gezeigt, mit welcher dieses fachliche Ziel des Anwendungsfalles erreicht werden kann.

```
58 For {
59     Component(
60         Id("#A01E28") and
61         Assembly(AnyAssembly with
62             Component(Id("#4664AA"))
63         )
64     )
65 } Do {
66     NQA("Maintainability" = "++")
67 }
```

Xtext 21: Transformation in Abhängigkeit der Komposition von Komponenten

Für alle Softwarekomponenten in einem Softwaresystem, welche über den Identifikator „#A01E28“ verfügen und zusätzlich noch mit einer Softwarekomponente, welche über den Identifikator „#4664AA“ verfügt, verbunden sind, soll das entsprechende nichtquantifizierte Qualitätsmerkmal `NQA("Maintainability" = "++")` zugeordnet werden. Semantisch würde dies zum Ausdruck bringen, dass eine solche Softwarekomponente über eine vergleichsweise „sehr gute“ Wartbarkeit verfügt.

4.2.7.6. Anwendungsfall: Deployment

Der sechste Anwendungsfall beschreibt das Transformieren der Betriebskosten einer Softwarekomponente in Abhängigkeit davon, ob eine Softwarekomponente einem bestimmten Server zugeordnet ist. Hierbei wird dieser Server über seinen Identifikator bestimmt und die entsprechenden Softwarekomponenten werden über die Annotation „Application“ bestimmt. Motiviert wird dieser Anwendungsfall dadurch, dass durch das Verteilen von bestimmten Softwarekomponenten auf bestimmte Server die Kosten zur Aufrechterhaltung des operativen Betriebes dieser Softwarekomponenten verändert werden kann. In Listing 22 wird diese entsprechende Qualitätseffekt-Spezifikation gezeigt, mit welcher dieses fachliche Ziel des Anwendungsfalles erreicht werden kann.

```
69 For {
70     Component(
71         Annotation("Application") and
72         Resource(Id("#A01E28"))
73     )
74 } Do {
75     OperatingCost(* 0.9)
76 }
```

Xtext 22: Transformation in Abhängigkeit der Bereitstellungsbeziehungen von Komponenten

Für alle Softwarekomponenten in einem Softwaresystem, welche mit der Annotation „Application“ versehen sind und ebenso einem Server mit dem Identifikator „#A01E28“ zugeordnet sind, sollen die Betriebskosten um 10 % verringert werden.

4.3. Qualitätseffekt-Transformation

Bei der Qualitätseffekt-Transformation werden die Qualitätseffekt-Spezifikationen mit ihren Komponenten-Spezifikationen und Transformations-Spezifikationen, welche mittels der domänenspezifischen Sprache definiert worden sind, ausgewertet und auf die entsprechenden Modelle angewendet. Hierbei wird bei der Auswertung im ersten Schritt auf der Grundlage der beschriebenen Komponenten-Spezifikation, die Softwarekomponenten aus dem komponentenbasierten Softwaresystem bestimmt, auf welche im zweiten Schritt die Transformations-Spezifikationen angewendet werden. Um Qualitätsmodelle gemäß der Qualitätseffekt-Spezifikationen so transformieren zu können, dass ihre Aussagen über Qualitätsattribute besser zur gegebenen Systemarchitektur passen, wurden Teile der Analyseverfahren für diese Qualitätsmodelle erweitert.

In Abbildung 4.10 wird der vollständige Überblick über die Integration des Ansatzes zur Qualitätseffekt-Transformation in die bestehenden Analyseverfahren von Palladio mit PerOpteryx gezeigt. Während die Abbildung 4.1 am Anfang dieses Kapitels lediglich den groben Überblick über die Funktionsweise dieses Ansatzes gezeigt hat, zeigt die Abbildung 4.10 nun die vollständige Integration der Qualitätseffekt-Transformation. Hierbei ist dieser Überblick in drei Hauptteile aufgeteilt, zunächst kommen die Modelle zur Eingabe, anschließend kommen die verschiedenen Transformationen dieser Modelle und abschließend kommen die verschiedenen Analyseverfahren für diese Modelle.

An dieser Stelle setzt sich die Eingabe zusammen aus dem Palladio-Komponentenmodell, dem Freiheitsgrade-Modell, dem Entwurfsentscheidungs-Modell, dem Kosten-Modell, dem Qualitative-Reasoning-Modell und der Qualitätseffekt-Spezifikation. Das Palladio-Komponentenmodell (Abschnitt 3.3.1) definiert die Softwarearchitektur mit allen dazugehörigen Komponenten mit ihren Service-Effekt-Spezifikationen (Abschnitt 3.3.2) und weiteren Betriebsmitteln. Das Freiheitsgrade-Modell (Abschnitt 3.3.5) definiert die Freiheitsgrade, wie beispielsweise den Austausch von Komponenten oder die Änderung der Komponentenzuordnung, welche vom PerOpteryx-Ansatz untersucht werden. Das Entwurfsentscheidungs-Modell (Abschnitt 3.3.4) umfasst Informationen über die Struktur einer Entwurfsentscheidung und die Umbaumaßnahmen für die Integration der Softwarekomponenten dieser Entwurfsentscheidung. Das Kosten-Modell (Abschnitt 3.3.3) umfasst unter anderem Informationen über die Initialkosten und die Betriebskosten von Softwarekomponenten und Betriebsmitteln. Das Qualitative-Reasoning-Modell (Abschnitt 3.3.6) definiert explizit informelles Wissen über nicht-quantifizierte Qualitätsmerkmale von Softwarekomponenten. Die Qualitätseffekt-Spezifikation (Abschnitt 4.2.4) gibt an, für welche Softwarekomponenten welche Transformationen ausgeführt werden sollen.

Die Transformationen bestehen an dieser Stelle zu einem aus PerOpteryx, dem Weber und dem eigenen Ansatz zur Auswertung und Anwendung der Qualitätseffekt-Spezifikation. Der PerOpteryx-Ansatz (Abschnitt 3.3.5) verwendet domänenspezifisches Wissen, um durch Manipulation bestimmter Parameter aus einer übergebenen Palladio-Kompo-

nentenmodell-Instanz neue Palladio-Komponentenmodell-Instanzen zu erzeugen, um so die ursprüngliche Palladio-Komponentenmodell-Instanz auf der Basis von modellbasierten Qualitätsvorhersagetechniken zu optimieren. Der Weber (Abschnitt 3.3.4) integriert die Softwarekomponenten einer Entwurfsentscheidung aus dem Entwurfsentscheidungs-Modell in eine bestehende Architektur. Bei einer Qualitätseffekt-Transformation für eine Qualitätseffekt-Spezifikation werden zunächst die Komponenten-Spezifikation ausgewertet und darauf aufbauend die Transformations-Spezifikationen auf die gegebenen Qualitätsmodelle angewendet.

Die Analyseverfahren bestehen an dieser Stelle zum einen aus Palladio, dem PCM Cost Solver und dem Qualitative-Reasoning. Der Palladio-Ansatz (Abschnitt 3.3) simuliert die übergebene Palladio-Komponentenmodell-Instanz und analysiert und bewertet hierbei die nicht-funktionalen Eigenschaften von Softwarearchitekturen wie beispielsweise die Performance. Der PCM-Cost-Solver (Abschnitt 3.3.3) ist eine Erweiterung für die Analyseverfahren von Palladio und ermöglicht, Informationen über die Initialkosten und die Betriebskosten von Softwarekomponenten und Betriebsmitteln zu analysieren. Qualitative-Reasoning (Abschnitt 3.3.6) ist ebenfalls eine Erweiterung für die Analyseverfahren von Palladio und ermöglicht die Analyse von informellem Wissen über nicht-quantifizierte Qualitätsmerkmale der Softwarekomponenten.

Die Integration des Ansatzes in die bestehenden Arbeitsabläufe von Palladio mit PerOpteryx ist es, die gegebenen Qualitätsmodelle, wie beispielsweise das Kosten-Modell oder Qualitative-Reasoning-Modell, vor ihrer jeweiligen Analyse zu transformieren. In diesem Zusammenhang erfolgt diese Transformation auf Grundlage der aktuellen konkreten Palladio-Komponentenmodell-Instanz, welche die Softwarearchitektur repräsentiert, und der gegebenen Qualitätseffekt-Spezifikationen. Diese konkrete Softwarearchitektur folgt an diesem Punkt auf die Architektur-Transformationen von PerOpteryx und dem Weber. Dabei generieren beide Ansätze neue Palladio-Komponentenmodell-Instanzen; PerOpteryx transformiert hierbei auf Basis des Freiheitsgrade-Modells und der Weber verwendet das Entwurfsentscheidungs-Modell als Basis für die Architektur-Transformationen.

Der Ansatz, welcher mit der Qualitätseffekt-Spezifikation verfolgt wird, ist die Transformation der Qualitätsmodelle von Softwarekomponenten, welche für die weiteren Analysen gebraucht werden, direkt vor dem jeweiligen Analyseverfahren in Relation zur konkreten Palladio-Komponentenmodell-Instanz, welche die Softwarearchitektur repräsentiert. An dieser Stelle werden lediglich die Werte aus den eingegebenen Qualitätsmodellen, wie beispielsweise dem Kosten-Modell oder Qualitative-Reasoning-Modell, temporär für den direkt folgenden Aufruf des zugehörigen Analyseverfahrens angepasst. Diese vorübergehende Anpassung dieser Werte hat keinen Einfluss auf die eingegebenen Qualitätsmodelle und wird nach dem jeweiligen Analyseverfahren wieder verworfen. Durch diese Vorgehensweise hat die Qualitätseffekt-Transformation keine weiteren Nebenwirkungen auf zukünftige Aufrufe der Analyseverfahren.

In diesem Zusammenhang kommt bei der Auswertung einer Qualitätseffekt-Spezifikation immer zunächst der sogenannte Sucher (Abschnitt 4.3.1) zu Einsatz. Dieser Sucher bestimmt die Softwarekomponenten in einem komponentenbasierten Softwaresystem anhand der Komponenten-Spezifikation und gibt die eindeutigen Referenzen auf diese Softwarekomponenten zurück. Anschließend werden für die Softwarekomponenten die zugehörigen Transformations-Spezifikationen ausgewertet und auf die zugehörigen Qua-

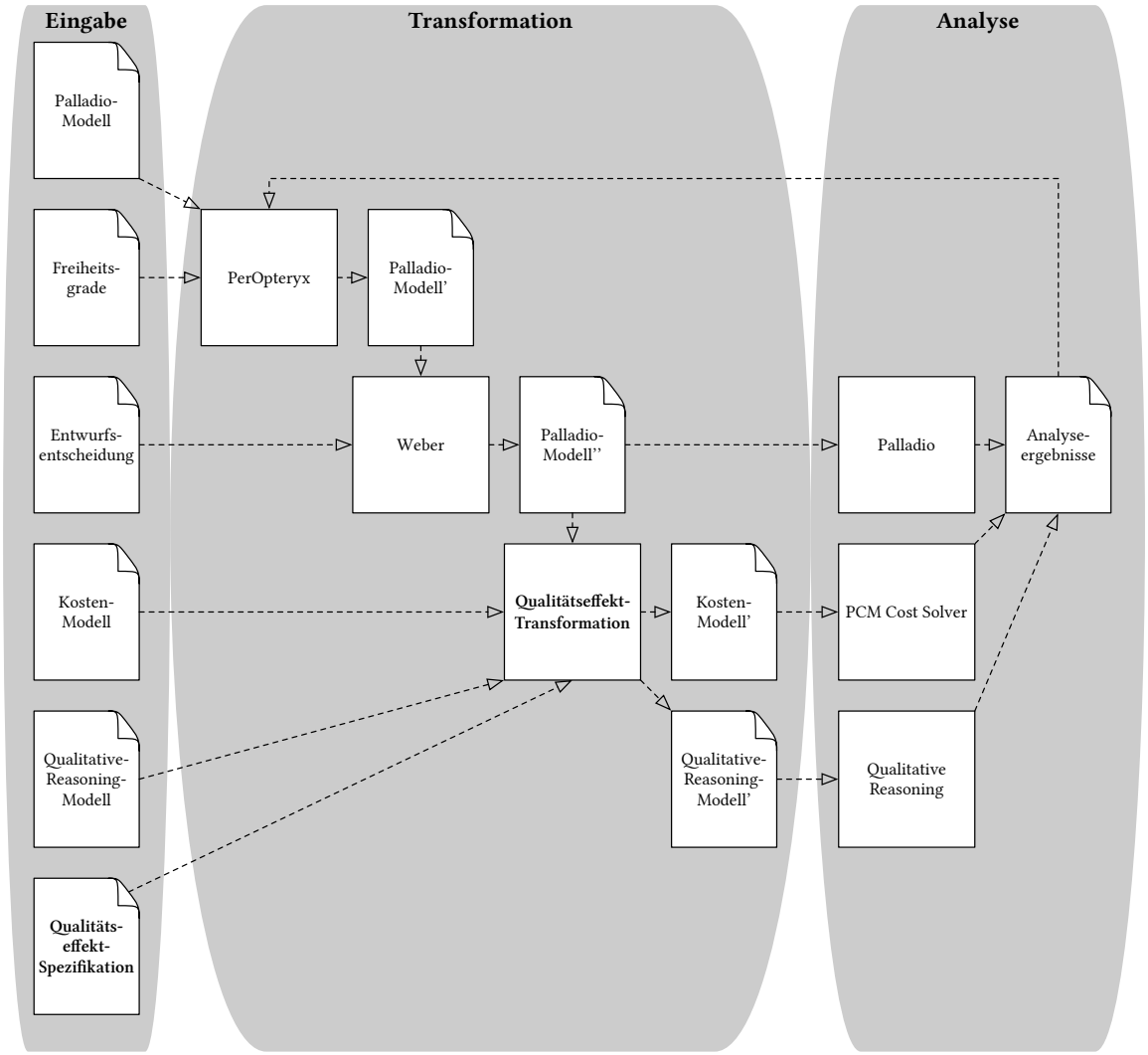

Abbildung 4.10.: Vollständiger Überblick über die Integration des Ansatzes

litätsmodelle angewendet. Hierbei wurden zwei Analyseverfahren von PerOpteryx, die qualitative Argumentation (Abschnitt 4.3.2) und die Kosten-Analyse (Abschnitt 4.3.3), erweitert.

4.3.1. Sucher

Der sogenannte Sucher ist eines der grundlegenden Bestandteile bei der Auswertung von Qualitätseffekt-Spezifikationen. Dieser Sucher bestimmt die Softwarekomponenten in einem komponentenbasierten Softwaresystem anhand der Komponenten-Spezifikation und gibt die eindeutigen Referenzen auf diese Softwarekomponenten zurück.

Zu diesem Zweck erhält der Sucher zum einen eine Palladio-Komponentenmodell-Instanz, welche die Softwarearchitektur repräsentiert, und zum anderen eine Qualitätseffekt-Spezifikation, welche die auszuwertenden Komponenten-Spezifikationen beinhaltet. Der

Sucher analysiert zunächst die strukturellen Eigenschaften der ihm übergebenen Palladio-Komponentenmodell-Instanz. Auf Grundlage dieser Analyse holt sich der Sucher anschließend die Komponenten-Spezifikation aus der gegebenen Qualitätseffekt-Spezifikation und sucht für diese alle Softwarekomponenten aus dem gegebenen Softwarearchitektur-Modell, welche die beschriebenen strukturellen Eigenschaften haben.

Das Ergebnis des Suchers ist die Ausgabe einer Menge von Zeichenketten. Diese Zeichenketten sind die jeweiligen eindeutigen Identifikatoren für die Instanz einer Softwarekomponente aus der gegebenen Palladio-Komponentenmodell-Instanz. Die Zeichenketten in dieser Menge sind an dieser Stelle immer eindeutig, das heißt, die zurückgegebene Menge der Zeichenketten des Suchers beinhaltet nie zwei gleiche Zeichenketten. Jedoch wenn keine Softwarekomponente in der gegebenen Palladio-Komponentenmodell-Instanz auf die gegebene Komponenten-Spezifikation passt, kann die zurückgegebene Menge auch leer sein.

Anhand dieser eindeutigen Identifikatoren für die Instanz einer Softwarekomponente können weitere Analyseverfahren, welche durch den Ansatz der Qualitätseffekt-Spezifikation erweitert wurden, prüfen, ob eine zu analysierende Softwarekomponente durch eine gegebene Qualitätseffekt-Spezifikation beeinflusst wird. Hierdurch werden diese Analyseverfahren von der Auswertung von Komponenten-Spezifikationen losgekoppelt und müssen lediglich die Interpretationsregeln der zugehörigen Transformations-Spezifikationen umsetzen.

4.3.2. Qualitative-Reasoning-Auswertung

Zur Umsetzung der Interpretationsregeln der Transformations-Spezifikationen NQA (Abschnitt 4.2.6.1) und Reasoning (Abschnitt 4.2.6.2) wurde die Qualitative-Reasoning-Auswertung (Abschnitt 3.3.6) als ein bereits bestehendes Analyseverfahren von Palladio erweitert. Die Qualitative-Reasoning-Auswertung ermöglicht, informelles Wissen über nicht-quantifizierte Qualitätsmerkmale von Softwarekomponenten zu analysieren.

Wenn in der Analyse die Werte für die nicht-quantifizierten Qualitätsmerkmale (NQA) oder Abbildungsregelwerke (Reasonings) einer Softwarekomponente benötigt werden, wird zunächst über den Sucher festgestellt, ob es eine Qualitätseffekt-Spezifikation gibt, welche die Qualitätsattribute dieser Softwarekomponente verändert. Wenn es keine Qualitätseffekt-Spezifikation gibt, welche eine Komponenten-Spezifikation beinhaltet, die auf die zu analysierende Softwarekomponente passt, werden die ursprüngliche Werte im Modell für die weitere Analyse verwendet. Wenn jedoch eine oder mehrere Qualitätseffekt-Spezifikationen existieren, welche auf die zu analysierende Softwarekomponente passen, werden die NQA-Transformationen (Abschnitt 4.2.6.1) oder Reasoning-Transformationen Abschnitt 4.2.6.2) der Qualitätseffekt-Spezifikation auf die Qualitätsattribute der zu analysierenden Softwarekomponente angewendet. Diese veränderten Qualitätsattribute werden anschließend für die weitere Analyse verwendet.

4.3.3. Kosten-Auswertung

Zur Umsetzung der Interpretationsregeln der Transformations-Spezifikationen Numeric (Abschnitt 4.2.6.3) für die Schlüsselwörter "InitialCost" und "OperatingCost" wurde die

PCM-Kosten-Auswertung (Abschnitt 3.3.3) als ein bereits bestehendes Analyseverfahren von Palladio erweitert. Die PCM-Kosten-Auswertung ermöglicht unter anderem, Informationen über die Initialkosten und die Betriebskosten von Softwarekomponenten und Betriebsmitteln zu analysieren.

Wenn in der PCM-Kosten-Auswertung die Werte für die Initialkosten oder Betriebskosten einer Softwarekomponente benötigt werden, wird zunächst über den Sucher festgestellt, ob es eine Qualitätseffekt-Spezifikation gibt, welche die Qualitätsattribute dieser Softwarekomponente verändert. Wenn es keine Qualitätseffekt-Spezifikation gibt, welche eine Komponenten-Spezifikation beinhaltet, die auf die zu analysierende Softwarekomponente passt, werden die ursprünglichen Werte im Modell für die weitere Analyse verwendet. Wenn jedoch eine oder mehrere Qualitätseffekt-Spezifikationen existieren, welche auf die zu analysierende Softwarekomponente passen, werden die Numeric-Transformationen (Abschnitt 4.2.6.3) der Qualitätseffekt-Spezifikation auf die Qualitätsattribute der zu analysierenden Softwarekomponente angewendet. Diese veränderten Qualitätsattribute werden anschließend für die weitere Analyse verwendet.

5. Experimentelle Validierung

Für die experimentelle Validierung der Qualitätseffekt-Spezifikation, als eine Erweiterung der Analyseverfahren von Palladio mit PerOpteryx, werden zwei unabhängige Fallstudien durchgeführt. Zu diesem Zweck wurde der Ansatz dieser Masterarbeit auf diese Fallstudien angewandt, um die Anwendbarkeit und den Nutzen zu zeigen und um die Beantwortung der folgenden Forschungsfragen (Abschnitt 1.2) zu berücksichtigen:

- Welche semantischen Klassen von architekturdefinierten Einflussfaktoren auf Qualitätsmodelle gibt es?
- Wie müsste ein formales Modell aussehen, das anhand von Regeln diese Einflussfaktoren modelliert?
- Wie können Qualitätsmodelle gemäß diesen Ergebnissen so verändert werden, dass ihre Aussagen über Qualitätsattribute besser zur gegebenen Systemarchitektur passen?

Mit den folgenden beiden Fallstudien wird gezeigt, wie Qualitätseffekt-Spezifikationen helfen können, Kompromissentscheidungen über die Softwarearchitektur zwischen mehreren Qualitätsmerkmalen zu treffen. Die experimentelle Validierung der Eignung und des wissenschaftlichen Mehrwerts dieses Ansatzes der Qualitätseffekt-Spezifikation erfolgt auf Grundlage der drei Validierungs- und Verifikationstypen nach Koziolek [26, S. 232].

5.1. Typen von Validierungen

Die drei Typen der Evaluation und Validierung im Wissensmanagement nach Koziolek werden für die experimentelle Validierung der Qualitätseffekt-Spezifikation übernommen. Dies soll vor allem ein klar strukturiertes Vorgehen und eine hohe Nachvollziehbarkeit bei der experimentellen Validierung des Ansatzes der Qualitätseffekt-Spezifikation erlauben.

Machbarkeit Die Bewertung der Anwendbarkeit der Qualitätseffekt-Spezifikation auf der grundlegendsten Ebene. Ziel ist es, zu zeigen, dass die Qualitätseffekt-Spezifikation für das Ziel verwendet werden kann, die strukturellen Eigenschaften von Softwarekomponenten in einem Softwaresystem und die Regeln für die Transformation von Qualitätsattributen dieser Softwarekomponenten zu modellieren.

Angemessenheit Die Bewertung der Angemessenheit der Qualitätseffekt-Spezifikation. Ziel ist es, zu zeigen, dass der Ansatz der Qualitätseffekt-Spezifikation genutzt werden kann, um Qualitätsattribute von Softwarekomponenten in Relation zu ihren strukturellen Eigenschaften zu erfassen und zu verwalten.

Anwendbarkeit Die Validierung der Anwendbarkeit der Qualitätseffekt-Spezifikation, wenn diese aus dem Sichtwinkel der Zielnutzer (Softwarearchitekten) verwendet wird. Ziel ist es, den Ansatz der Qualitätseffekt-Spezifikation in einen realitätsnahen Kontext zu stellen und mithilfe der Umsetzung der Qualitätseffekt-Transformation in einer automatisierten Entwurfsraumuntersuchung zu nutzen.

Auf der Grundlage dieser drei Typen werden für die experimentelle Validierung der Qualitätseffekt-Spezifikation in Tabelle 5.1 die Forschungsfragen und Ziele erfasst. Hierfür werden zunächst die Ziele definiert, welche eigentlich bei der experimentellen Validierung verfolgt werden. Anschließend werden die Fragen definiert, deren Beantwortung aussagen, ob ein Ziel erreicht wurde. Zuletzt wird die Metrik definiert, welche eine messbare Beantwortung dieser Fragen darstellt.

Machbarkeit	**Ziel:** Verknüpfung von Qualitätsattributen mit den strukturellen Eigenschaften eines komponentenbasierten Softwaresystems.	
Frage: Welche semantischen Klassen von architekturdefinierten Einflussfaktoren auf Qualitätsmodelle gibt es?		Metrik in Abschnitt 4.1
Frage: Wie müsste ein formales Modell aussehen, das anhand von Regeln diese Einflussfaktoren modelliert?		Metrik in Abschnitt 4.2
Angemessenheit	**Ziel:** Simulation in Abhängigkeit von Parametern, welche aus den spezifischen Eigenschaften einer Softwarearchitektur abgeleitet werden.	
Frage: Wie können Qualitätsmodelle gemäß diesen Ergebnissen so verändert werden, dass ihre Aussagen über Qualitätsattribute besser zur gegebenen Systemarchitektur passen?		Metrik in Abschnitt 4.3
Anwendbarkeit	**Ziel:** Neue Dimensionen, welche in einer automatisierten Entwurfsraumuntersuchung genutzt werden können.	
Frage: Gewinnen die Zielnutzer neue Erkenntnisse über den Einfluss der Architektur auf Qualitätsmerkmale, welche bisher mangels Wissensmodellierung und Optimierungsmöglichkeiten nicht sichtbar waren?		Metrik in Abschnitt 5.3

Tabelle 5.1.: Ziele und Forschungsfragen der experimentellen Validierung

5.2. Fallstudien

Zur Beantwortung der Forschungsfragen und für die Bestimmung der Maßzahlen der Metriken wurden zwei voneinander unabhängige Fallstudien durchgeführt. Hierbei wurden

diese zwei Fallstudien auf den Ansatz dieser Masterarbeit angewendet, um Messungen durchzuführen, welche die Fragen des Goal-Question-Metric-Plans beantworten.

Hierbei folgen beide Fallstudien einem dreistufigen Entwurf:

1. Erstellen aller für die Optimierung notwendigen Modelle.
2. Erweitern der PCM-Instanzen um Qualitätseffekt-Spezifikationen.
3. Durchführung der Entwurfsraumoptimierung zum Ermitteln der Pareto-optimalen Kandidaten.

Abschnitt 5.2.1 beschreibt die erste Fallstudie, in welcher der architektonische Einfluss eines Frameworks zum Loggen von Anwendungsmeldungen in einem System zur Generierung und Abfrage von Geschäftsprozessen mittels Qualitätseffekt-Spezifikation beschrieben und ausgewertet werden soll. Abschnitt 5.2.2 beschreibt die zweite Fallstudie, in welcher der architektonische Einfluss eines Angrifferkennungssystems in einem Online-Marktplatz mittels Qualitätseffekt-Spezifikation beschrieben und ausgewertet werden soll.

5.2.1. Business Reporting System

In der ersten Fallstudie wird der architektonische Einfluss eines Frameworks zum Loggen von Anwendungsmeldungen in einem System zur Generierung und Abfrage von Geschäftsprozessen mittels Qualitätseffekt-Spezifikation beschrieben und ausgewertet. Die ausführliche Charakterisierung und eine visuelle Darstellung der Komponenten des Frameworks und Systems erfolgt im Abschnitt 5.2.1.1. Die Beschreibung des Einflusses eines Frameworks zum Loggen von Anwendungsmeldungen auf die Qualitätsattribute erfolgt hierbei über eine Qualitätseffekt-Spezifikation, welche im Abschnitt 5.2.1.2 genauer beschrieben wird. Bei der Auswertung werden einmal die Abweichung zu vorherigen Ergebnissen ohne Verwendung von Qualitätseffekt-Spezifikationen betrachten, sowie die Ausführungszeit mit und ohne Qualitätseffekt-Spezifikationen. Die Ergebnisse hierüber sind im Abschnitt 5.2.1.3 genauer beschrieben.

5.2.1.1. Systemkonfiguration

Das System zur Generierung und Abfrage von Geschäftsprozessen wird als Business Reporting System (BRS) bezeichnet und ist in Abbildung 5.1 als ein UML-Diagramm visuell dargestellt. Dieses Business Reporting System ermöglicht es dem Anwender, statistische Auswertungen über die Ausführung eines Geschäftsprozesses aus einer Datenbank abzurufen. Zu diesem Zweck bietet das System dem Anwender die Möglichkeit, sowohl grafische als auch digitale Berichte zu erstellen. Das Business Reporting System basiert grob auf einem realen System, welches in [50] beschrieben wird.

Die Anfangskonfiguration des Business Reporting System ist ein vierstufiges System, das aus einem Webserver, zwei Servern für die Geschäftslogik und einem Datenbankserver besteht. Um einen grafischen Bericht oder eine digitale Ansicht der Rohdaten zu erstellen, sendet der Benutzer eine Anfrage an die Webserver-Komponente. Diese Benutzeranfragen werden von der Web-Server-Komponente an die Scheduler-Komponente delegiert. Für die

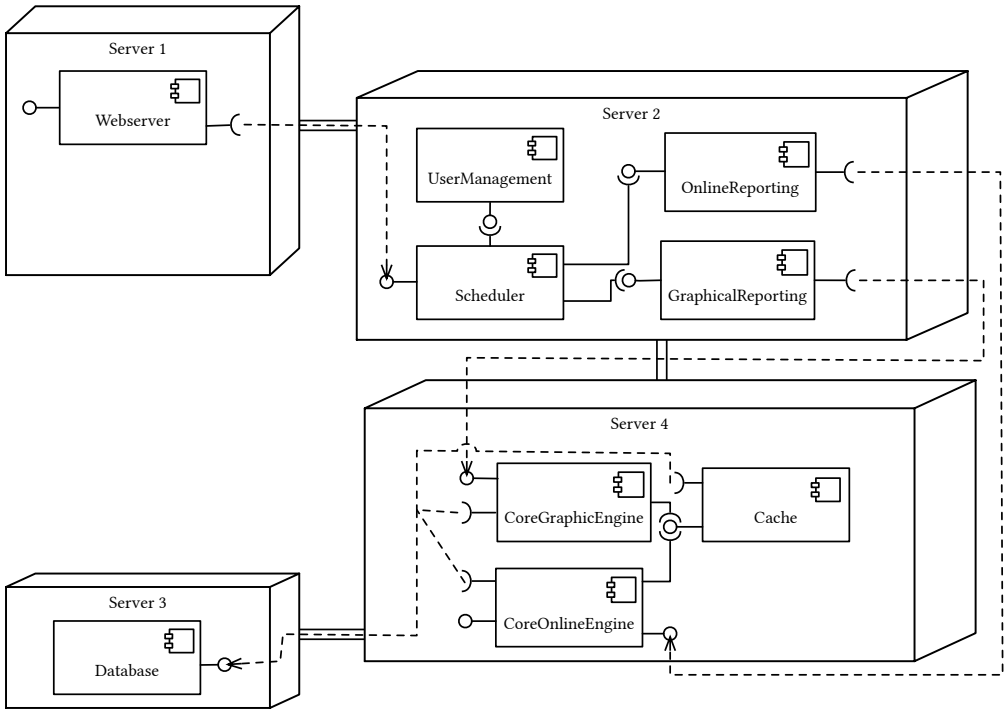

Abbildung 5.1.: Systemarchitektur des Business Reporting System ohne log4j-Framework

Benutzerverwaltung ruft der Scheduler zunächst die Komponente Benutzerverwaltung zur Sitzungsbearbeitung auf. Je nach Art der Anfragen delegiert die Scheduler-Komponente die Anfragen entweder an die grafische Reporting-Komponente oder an die digitale (online) Reporting-Komponente. Beide Komponenten benötigen die Datenbankkomponente. Um die Datenbanklast zu reduzieren, werden bereits geladene oder generierte Daten in der Cache-Komponente zwischengespeichert. Die digitale Reporting-Komponente bietet zudem noch eine Schnittstelle für die Administration des Systems an.

Als Framework zum Loggen von Anwendungsmeldungen wird log4j verwendet. Das log4j-Framework ist ein in Java implementiertes Open-Source-Framework, dessen Entwicklung von der Apache Software Foundation organisiert wird. Das Framework hat sich zu einem Industriestandard für das Loggen von Anwendungsmeldungen in Java-basierten Anwendungen entwickelt. Das log4j-Framework dient dazu, auftretende Fehler und Informationsmeldungen über die jeweilige Logger-Komponente an das gewählte Loggingsystem weiterzuleiten, die sogenannte Appender-Komponente. Dieses Loggingsystem verwendet wiederum eine sogenannte Formater-Komponente, welche die Anwendungsmeldungen in ein einheitliches zuvor konfiguriertes Format konvertiert.

In Abbildung 5.2 ist die Integration des log4j-Frameworks in das Business Reporting System visuell dargestellt. Hierbei erfolgt die Integration nur an der Webserver-Komponente und der Online-Reporting-Komponente, da nur diese beiden Komponenten eine öffentliche für Benutzer zugängliche Schnittstelle haben. An dieser Stelle setzt sich das log4j-Framework aus vier Komponenten zusammen. Den beiden Logger-Komponenten, welche verantwortlich sind für die Erfassung von Anwendungsmeldungen. Hierbei wird jeder Softwarekomponente, für welche Anwendungsmeldungen geloggt werden sollen, eine

eigene Logger-Komponente zugeordnet. Die Appender-Komponente, welche verantwortlich ist für die Veröffentlichung von Logging-Informationen an verschiedene bevorzugte Ziele. Die Formater-Komponente, welche verantwortlich ist für die Formatierung von Anwendungsmeldungen in verschiedenen Formaten.

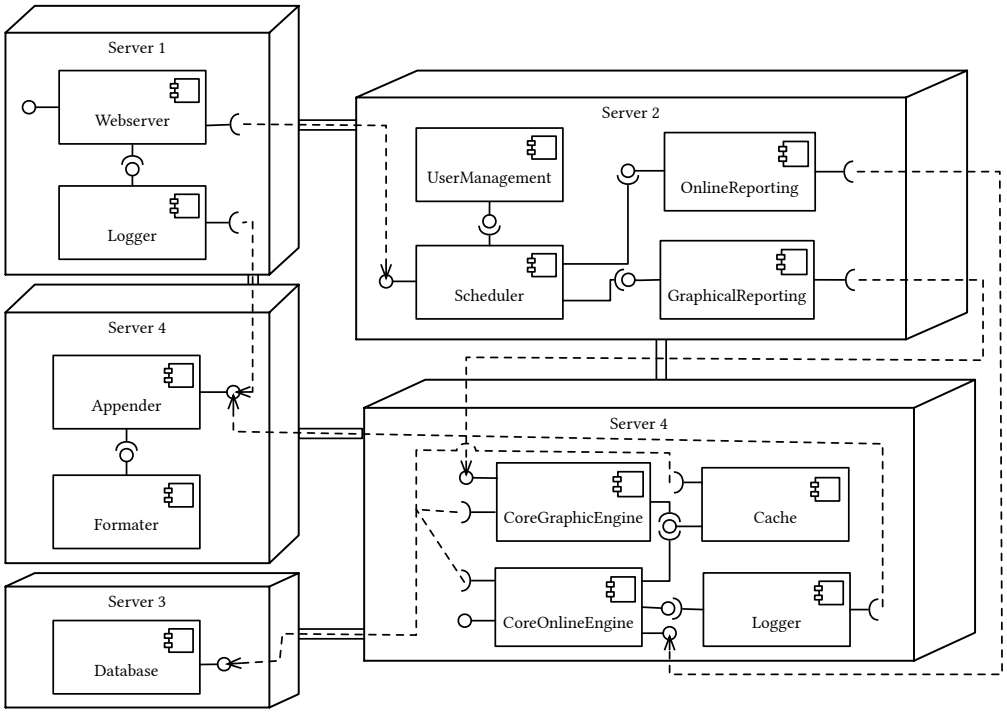

Abbildung 5.2.: Systemarchitektur des Business Reporting System mit log4j-Framework an der CoreOnlineEngine und dem Webserver

Die Integration der Komponenten des log4j-Framework in das Business Reporting System lässt sich mithilfe des Strukturmodells (Abschnitt 3.3.4) formalisieren. Dies erlaubt auch Freiheitsgrade für die Integration der Komponenten des log4j-Framework. An dieser Stelle kann das log4j-Framework entweder an keine der beiden Komponenten oder an beide gleichzeitig oder einmal nur an die Webserver-Komponente oder einmal nur an die Online-Reporting-Komponente integriert werden. Der erste Fall, dass das log4j-Framework nicht integriert worden ist, ist visuell in Abbildung 5.1 dargestellt. Der zweite Fall, dass das log4j-Framework vollständig integriert worden ist, ist visuell in Abbildung 5.2 dargestellt. Der dritte Fall, dass das log4j-Framework nur an die Webserver-Komponente integriert worden ist, ist visuell in Abbildung 5.3 dargestellt. Und der vierte Fall, dass das log4j-Framework nur an die Online-Reporting-Komponente integriert worden ist, ist visuell in Abbildung 5.4 dargestellt. Diese vier Freiheitsgrade für die Integration des log4j-Framework erweitern die architektonischen Freiheitsgrade der Palladio-Komponentenmodell-Instanz und können von PerOpteryx zur Optimierung genutzt werden.

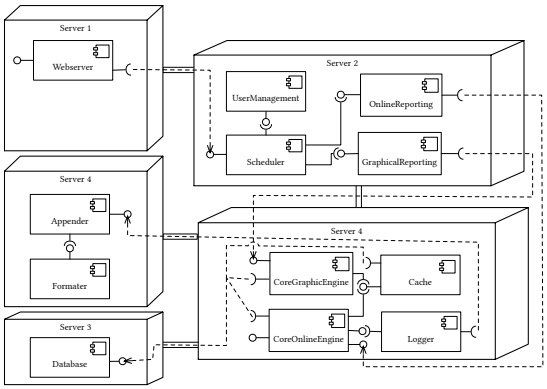

Abbildung 5.3.: Systemarchitektur des Business Reporting System mit log4j-Framework nur an der CoreOnlineEngine

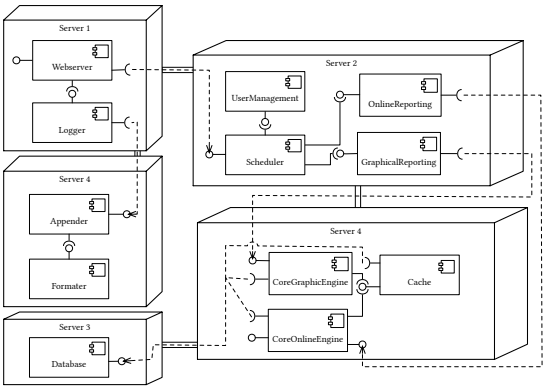

Abbildung 5.4.: Systemarchitektur des Business Reporting System mit log4j-Framework nur an dem Webserver

5.2.1.2. Experimentelles Vorgehen

In dieser Fallstudie sollen die Freiheitsgrade des Strukturmodells zur Integration des log4j-Framework untersucht werden. Diese Freiheitsgrade des Strukturmodells beeinflussen die Softwarequalität der gesamten Softwarearchitektur. Hierbei soll PerOpteryx eingesetzt werden, um die Pareto-optimalen Kompromisse zwischen diesen Freiheitsgraden und den Qualitätsmerkmalen des Systems zu finden. Zu diesem Zweck werden sowohl die Performance wie auch die Kosten des Systems als quantifizierte Qualitätsmerkmale betrachtet und die Wartbarkeit wird als nicht-quantifiziertes Qualitätsmerkmal mit dem Ansatz der qualitativen Argumentation betrachtet. Durch die Analyse dieser Qualitätsmerkmale kann PerOpteryx die Pareto-optimalen Lösungen für Freiheitsgrade des Strukturmodells finden.

Um die Kosten des Systems zu modellieren und zu bewerten, wird das Kostenmodell (Abschnitt 3.3.3) verwendet. Um die Performance des Systems zu modellieren, wird der RDSEFF-Mechanismus von Palladio verwendet und um den zweiten Teil des Performance-Modells zu modellieren, wird ebenfalls der Hardware-Kontext des BRS modelliert. Um die Performance zu bewerten, wird das Palladio-Komponentenmodell in Layered-Queuing-Netzwerke transformiert und analysiert (Abschnitt 3.3). Die Analyse der Performance erfolgt mit einem konstanten Nutzungsprofil und ermittelt für dieses die Antwortzeit für jedes System.

Um die Wartbarkeit als ein nicht-quantifiziertes Qualitätsmerkmal zu analysieren, wird zunächst das notwendige Dimensions Set `{"--", "-", "0", "+", "++"}` für das Qualitätsmodell definiert. An dieser Stelle ist das Element `"++"` der am besten bewertete und das Element `"--"` der am schlechtesten bewertete Wert (geordnete Menge). Das nicht-quantifizierte Qualitätsmerkmal für die Wartbarkeit soll aus den strukturellen Eigenschaften der Softwarekomponente in dem Business Reporting System abgleitet werden. Wenn eine Softwarekomponente in dem System eine öffentliche für Benutzer zugängliche Schnittstelle hat und mit der Logger-Komponente aus dem log4j-Framework verbunden ist, hat

diese eine gute Wartbarkeit. Ausgedrückt wird dies durch das nicht-quantifizierte Qualitätsmerkmal `("Maintainability", "+")`. Im Gegenzug, wenn eine Softwarekomponente in dem System eine öffentliche für Benutzer zugängliche Schnittstelle hat und nicht mit der Logger-Komponente aus dem log4j-Framework verbunden ist, hat diese eine schlechte Wartbarkeit. Ausgedrückt wird dies durch das nicht-quantifizierte Qualitätsmerkmal `("Maintainability", "-")`. Um die Qualitätsattribute der Softwarekomponenten in Relation zu ihren strukturellen Eigenschaften in dem komponentenbasierten Softwaresystem zu spezifizieren, wird der Ansatz der Qualitätseffekt-Spezifikation verwendet.

Zu diesem Zweck wird für diese Fallstudie die Qualitätseffekt-Spezifikation in Listing 23 spezifiziert. Diese Qualitätseffekt-Spezifikation ordnet für eine Softwarekomponente, welche eine öffentliche für Benutzer zugängliche Schnittstelle hat, das nicht-quantifizierte Qualitätsmerkmal für die Wartbarkeit in Abhängigkeit, ob diese Komponente mit der Logger-Komponente aus dem log4j-Framework verbunden ist oder nicht. Durch diese Qualitätseffekt-Spezifikation kann das nicht-quantifizierte Qualitätsmerkmal für die Wartbarkeit aus den strukturellen Eigenschaften der Softwarekomponente in dem Business Reporting System abgeleitet und analysiert werden.

```
1  For {
2      Component(
3          Role(InfrastructureRequiredProvided) and
4          Assembly(AnyAssembly with
5              Component(Name("Log4j1Logger"))
6          )
7      )
8  } Do {
9      NQA("Maintainability" = "+")
10 }
11 For {
12     Component(
13         Role(InfrastructureRequiredProvided) and
14         Assembly(not AnyAssembly with
15             Component(Name("Log4j1Logger"))
16         )
17     )
18 } Do {
19     NQA("Maintainability" = "-")
20 }
```

Xtext 23: Qualitätseffekt-Spezifikation für die BRS-Fallstudie

5.2.1.3. Ergebnisse

Um zu validieren, ob der Ansatz der Qualitätseffekt-Spezifikation eine Analyse von Qualitätsattributen, welche erst aus den strukturellen Eigenschaften einer konkreten Softwarearchitektur abgeleitet werden können, ermöglicht und hierdurch nun die zu den spezifischen

Eigenschaften der Softwarearchitektur besser passenden Ergebnisse liefert, wurde die Abweichung zu den Ergebnissen ohne Verwendung des Ansatz der Qualitätseffekt-Spezifikation ermittelt. Zu diesem Zweck wurden einmal mit und einmal ohne Ansatz der Qualitätseffekt-Spezifikation jeweils insgesamt 800 Iterationen mit jeweils 20 Kandidaten pro Iteration bewertet. Die Ergebnisse für die 800 Iterationen mit jeweils 20 Kandidaten mit dem Ansatz der Qualitätseffekt-Spezifikation ist in Abbildung 5.5 dargestellt. Und die Architekturkandidaten für die 800 Iterationen mit jeweils 20 Kandidaten ohne den Ansatz der Qualitätseffekt-Spezifikation sind in Abbildung 5.6 dargestellt. Diese beiden Diagramme zeigen jeweils auf ihrer X-Achse die Antwortzeit in Sekunden und ihrer Y-Achse die Kosten für die Architektur-Kandidaten. An dieser Stelle sind die Pareto-optimalen Kandidaten in beiden Diagrammen farbig hervorgehoben. Die Optimierung durch PerOpteryx mit dem Ansatz der Qualitätseffekt-Spezifikation ergab insgesamt 8040 Architekturkandidaten, davon 51 Pareto-optimale Ergebnisse. Und die Optimierung durch PerOpteryx ohne den Ansatz der Qualitätseffekt-Spezifikation ergab insgesamt 8040 Architekturkandidaten, davon 25 Pareto-optimale Ergebnisse.

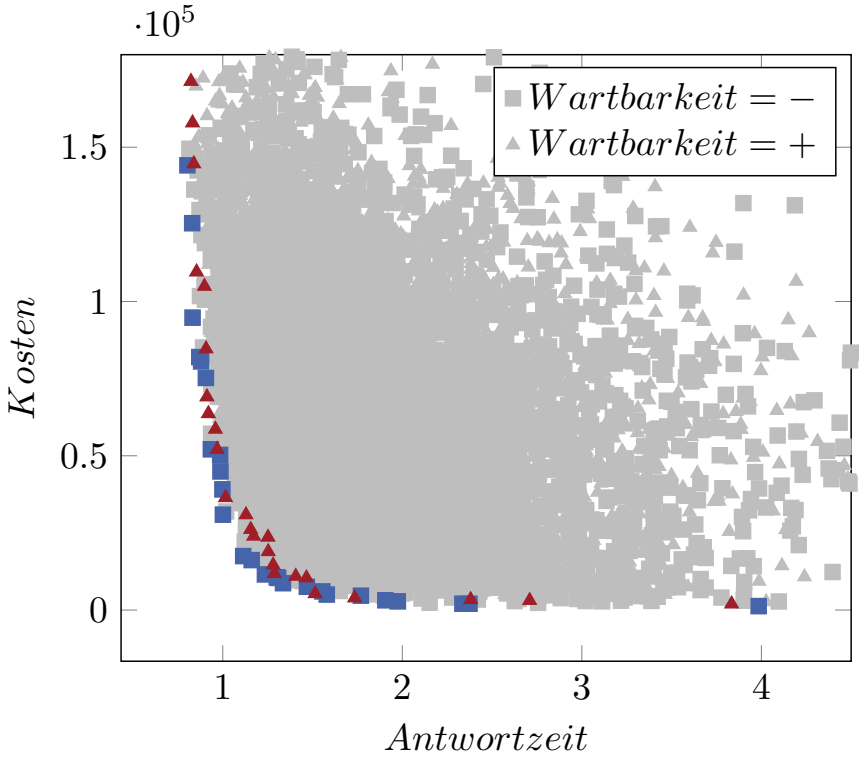

Abbildung 5.5.: BRS-Ergebnisse: Antwortzeit, Kosten und Wartbarkeit

Die Auswertung zeigt, wie sich die Pareto-optimalen Kandidaten verändern, wenn andere Qualitätsmerkmale wie Wartbarkeit berücksichtigt werden. Ohne die Auswertung dieses Qualitätsmerkmales wären einige Pareto-optimale Kandidaten mit besserer Wartbarkeit, jedoch schlechterem Verhältnis von Performance und Kosten nicht berücksichtigt worden. Anhand dieser Ergebnisse kann ein Software-Architekt nun entscheiden, welcher Architekturkandidat nach Performance, Kosten und Wartbarkeit ausgewählt wird. Wenn

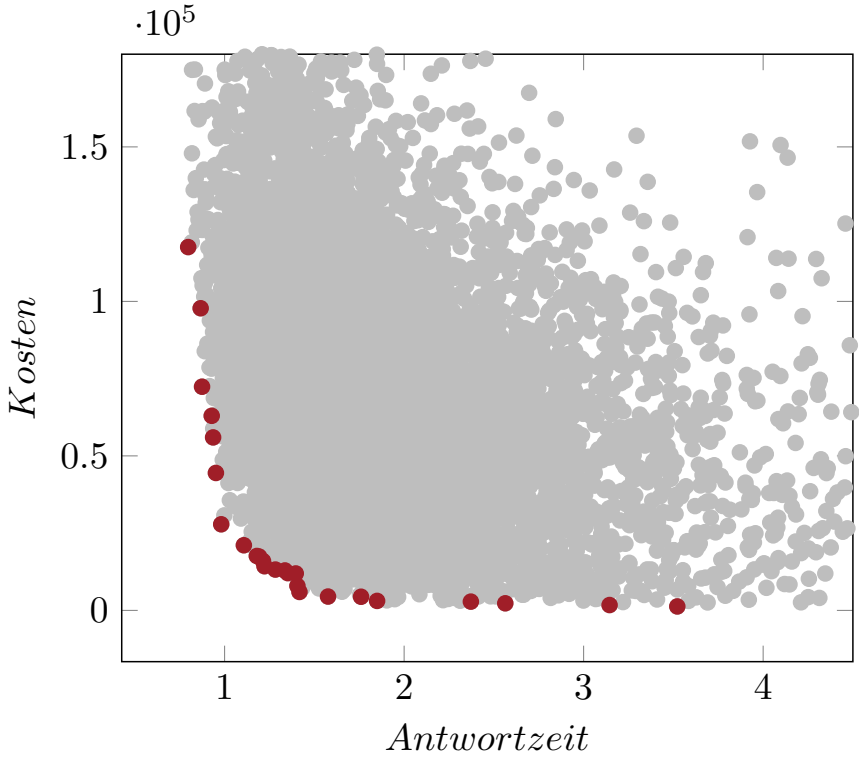

Abbildung 5.6.: BRS-Ergebnisse: Antwortzeit und Kosten

die spezifischen Projektanforderungen auf einer besseren Wartbarkeit des Gesamtsystems basieren, müsste der Software-Architekt die teureren Softwarekomponenten auswählen. Die Ergebnisse zeigen jedoch, dass die Performance nicht signifikant reduziert werden muss. Soll die Performance weiter verbessert werden, muss dies zum Beispiel durch eine Verbesserung der Prozessorleistung erreicht werden.

Die Ergebnisse zeigen, dass die letztendlich gewählten Softwarekomponenten stark von den Anforderungen und den verfügbaren Betriebsmitteln abhängig sind. Der Ansatz der Qualitätseffekt-Spezifikation macht diesen Kompromiss deutlich, der sich aus mehreren konkurrierenden Anforderungen ergibt.

Um zu validieren, ob der Ansatz der Qualitätseffekt-Spezifikation keinen zu hohen negativen Einfluss auf die Verarbeitungsgeschwindigkeit hat, wurde die Ausführungszeit mit und ohne der Verwendung die Ansatzes ermittelt. Zu diesem Zweck wurden einmal mit und einmal ohne Ansatz der Qualitätseffekt-Spezifikation jeweils insgesamt 800 Iterationen mit jeweils 20 Kandidaten pro Iteration bewertet. Die Optimierung durch PerOpteryx mit dem Ansatz der Qualitätseffekt-Spezifikation dauerte insgesamt 350 Minuten und die Optimierung durch PerOpteryx ohne den Ansatz der Qualitätseffekt-Spezifikation dauerte hingegen 344 Minuten. Die Differenz zwischen den Ausführungszeiten mit und ohne die Verwendung des Ansatzes der Qualitätseffekt-Spezifikation beträgt hierbei lediglich 6 Minuten. Dies bestätigt, dass für diese Fallstudie die Qualitätseffekt-Spezifikation keinen zu hohen negativen Einfluss auf die Verarbeitungsgeschwindigkeit hat.

5.2.2. mRUBiS Online-Marktplatz

In der zweiten Fallstudie wird der architektonische Einfluss eines Angrifferkennungssystems in einem Online-Marktplatz mittels Qualitätseffekt-Spezifikation beschrieben und ausgewertet. Die ausführliche Charakterisierung und eine visuelle Darstellung der Komponenten des Angrifferkennungssystems und des Online-Marktplatzes erfolgt im Abschnitt 5.2.2.1. Die Beschreibung des Einflusses eines Angrifferkennungssystems auf die Qualitätsattribute erfolgt hierbei über eine Qualitätseffekt-Spezifikation, welche im Abschnitt 5.2.2.2 genauer beschrieben wird. Bei der Auswertung werden die verschiedenen Freiheitsgrade zur Integration eines Angrifferkennungssystems mithilfe einer Qualitätseffekt-Spezifikation untersucht, die Ergebnisse hierüber sind im Abschnitt 5.2.2.3 genauer beschrieben.

5.2.2.1. Systemkonfiguration

Das Softwaresystem für den Online-Marktplatz wird als mRUBiS [47] bezeichnet und ist in Abbildung 5.7 in einer leicht vereinfachten Form als ein UML-Diagramm dargestellt. mRUBiS ist ein Online-Marktplatz, auf welchem Benutzer Artikel verkaufen oder versteigern können. Wie in Abbildung 5.7 dargestellt verfügt mRUBiS über Softwarekomponenten zur Verwaltung von Artikeln (Item Management Service), Benutzern (User Management Service), Auktionen und Einkäufen (Bid and Buy Service), Inventar (Inventory Service) und Benutzerbewertungen (Reputation Service), zur Authentifizierung von Benutzern (Authentication Service) und zur Persistenz und zum Abrufen von Daten (Persistence and Query Services). mRUBiS ist mit der EJB3-Technologie realisiert und kann so unter anderem auf dem GlassFish-Anwendungsserver bereitgestellt werden. mRUBiS strebt ein hohes Umsatzvolumen an, indem es die Kundenzufriedenheit erreicht und Kunden zu zusätzlichen Käufen anregt. Daher sollte das System hochverfügbar und seine Antwortzeit gering sein.

Als Rahmenwerk für ein komponentenbasiertes Angrifferkennungssystes wird ein generisches Intrusion Detection System (IDS) verwendet. Ein Intrusion Detection System ist ein Softwaresystem, das durch aktive Überwachung eines anderen Softwaresystems, Angriffe oder Missbrauch dieses Systems erkennt. An dieser Stelle wird als Intrusion Detection System eine Komposition von Softwarekomponenten bezeichnet, welche den gesamten Prozess von der Ereigniserfassung bis hin zur Analyse unterstützen. Im Allgemeinen setzen sich ein Intrusion Detection System aus Sensoren-, Datenbank-, Management- und Auswertung-Softwarekomponenten zusammen [43, S. 6].

Hostbasierte Sensoren werden dafür eingesetzt, um Angriffe und Missbrauch zu erkennen, welche auf Anwendungsebene durchgeführt werden. Beispiele für solche Angriffe sind Log-in-Fehlversuche oder Rechteüberschreitungen von Benutzern. Die Datenbank dient der Speicherung der Ereignisdaten zur späteren Weiterverarbeitung. Um Angriffe erkennen zu können, müssen die Ereignisdaten der Sensoren über einen längeren Zeitraum gespeichert werden. Über die Managementstation erfolgt die Konfiguration des kompletten Systems. Hierzu gehören unter anderem die Aufnahmen der Komponenten in das Intrusion Detection System. Die Auswertungsstation verfügt über die Funktionalität zur

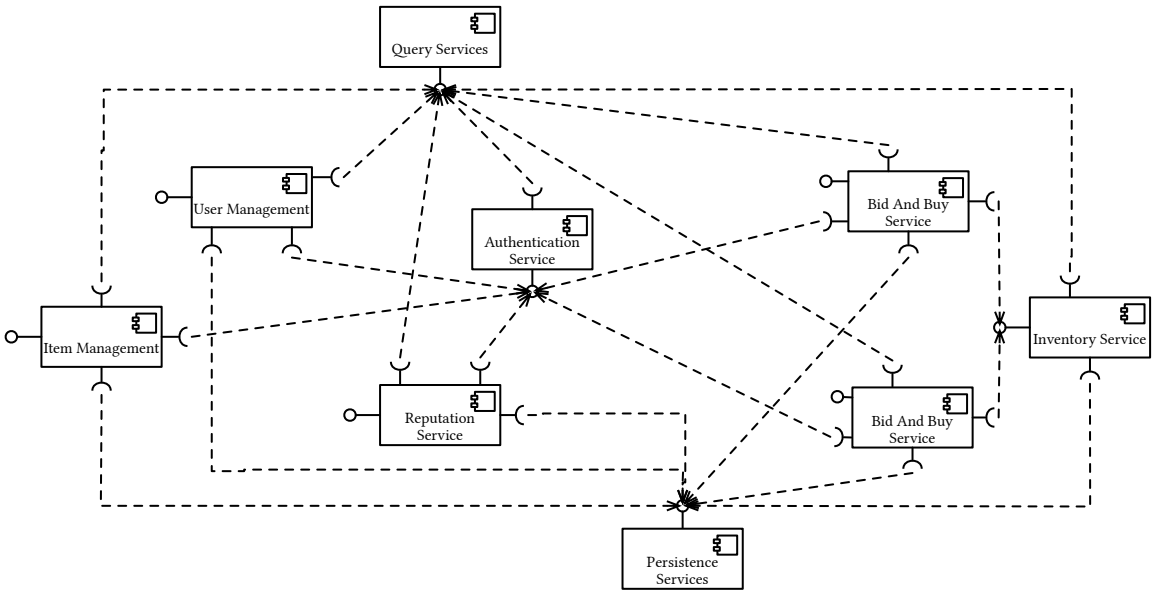

Abbildung 5.7.: Systemarchitektur des mRUBiS Online-Marktplatz

Auswertung aufgezeichneter Ereignisse. Diese Auswertung dient primär der Erkennung und Erstanalyse eingehender Ereignismeldungen.

In Abbildung 5.8 ist die Integration dieser vier Softwarekomponenten des Intrusion Detection System an alle fünf Komponenten von mRUBiS dargestellt. Um Angriffe durch Benutzer auf den Online-Marktplatz umfassend zu erkennen, sollten alle Softwarekomponenten, welche hauptverantwortlich für die Verarbeitung der Benutzereingaben sind, mittels IDS-Sensoren überwacht werden. Im mRUBiS Online-Marktplatz wären dies die Softwarekomponenten zur Authentifizierung von Benutzern (Authentication Service), der Verwaltung von Artikeln und Benutzern (Item and User Management Service) sowie für die Auktionen und Einkäufen (Bid and Buy Service).

Die Integration der Komponenten eines Intrusion Detection System in ein anderes Softwaresystem lässt sich mithilfe des Strukturmodells (Abschnitt 3.3.4) formalisieren. Das Strukturmodell erlaubt hierzu auch Freiheitsgrade für die Integration der Komponenten des Intrusion Detection System an diese fünf Komponenten des mRUBiS Online-Marktplatz.

Für diese Fallstudie wurden diese vier verschiedenen Freiheitsgrade für die Integration der Komponenten des Intrusion Detection System umgesetzt, sodass diese die architektonischen Freiheitsgrade der Palladio-Komponentenmodell-Instanz erweitern und so auch von PerOpteryx zur Optimierung genutzt werden können.

5.2.2.2. Experimentelles Vorgehen

In dieser Fallstudie sollen die Freiheitsgrade des Strukturmodells zur Integration eines Intrusion Detection System untersucht werden. Diese Freiheitsgrade des Strukturmodells beeinflussen die Softwarequalität der gesamten Softwarearchitektur. Hierbei soll PerOpteryx eingesetzt werden, um die Pareto-optimalen Kompromisse zwischen diesen Freiheitsgra-

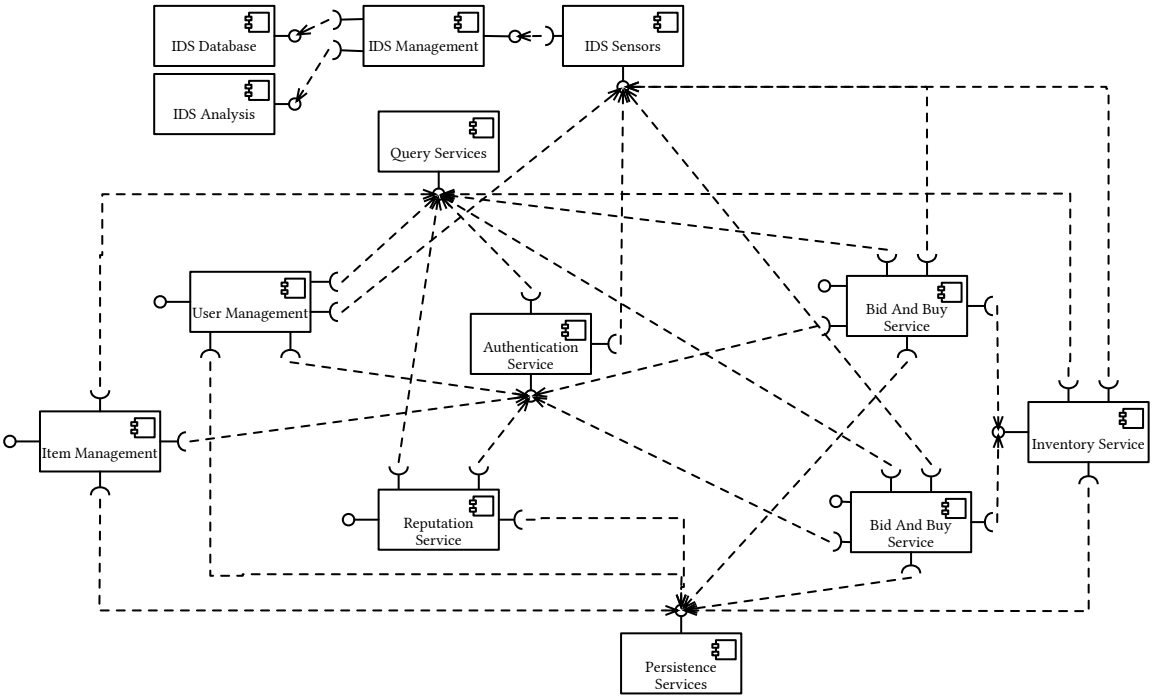

Abbildung 5.8.: Systemarchitektur des mRUBiS Online-Marktplatz mit einem Angrifferkennungssystem

den der Integration und der Angriffserkennung des Intrusion Detection System zu finden. Zu diesem Zweck werden sowohl die Performance wie auch die Kosten des Systems als quantifizierte Qualitätsmerkmale betrachtet und die Anzahl der Integrationspunkte der Sensoren des Intrusion Detection System wird als nicht-quantifiziertes Qualitätsmerkmal mit dem Ansatz der qualitativen Argumentation (Abschnitt 3.3.6) betrachtet. Durch die Analyse dieser Qualitätsmerkmale kann PerOpteryx die Pareto-optimalen Kompromisse für Freiheitsgrade des Strukturmodells finden.

Um die Kosten des Systems zu modellieren und zu bewerten, wird das Kostenmodell (Abschnitt 3.3.3) verwendet. Um die Performance des Systems zu modellieren, wird der RDSEFF-Mechanismus von Palladio verwendet und um den zweiten Teil des Performance-Modells zu modellieren, wird ebenfalls der Hardware-Kontext modelliert. In dieser Fallstudie besteht der Hardware-Kontext lediglich aus einem Server mit variabler Verarbeitungsrate, welche ebenfalls als Freiheitsgrad genutzt werden kann. Um die Performance zu bewerten, wird das Palladio-Komponentenmodell in Layered-Queuing-Netzwerke transformiert und analysiert (Abschnitt 3.3). Die Analyse der Performance erfolgt mit einem konstanten Nutzungsprofil und ermittelt für dieses die Antwortzeit für jedes System.

Um die Anzahl der Integrationspunkte der Sensoren des Intrusion Detection System als ein Qualitätsmerkmal analysieren zu können, muss zunächst das notwendige Dimensions Set für das Qualitätsmodell definiert werden. Für diese Fallstudie wird das folgende Dimensions Set zur Repräsentation der Anzahl der Integrationspunkte definiert: `{"0", "1", "2", "3", "4", "5", "6", "7", "8", "9", "10"}`. An dieser Stelle bedeutet das

Element `"0"`, dass die Sensoren des Intrusion Detection System nicht integriert wurden und das Element `"10"` bedeutet, dass die Sensoren des Intrusion Detection System an zehn verschiedenen Stellen integriert wurden.

Zu diesem Zweck wird für diese zweite Fallstudie die Qualitätseffekt-Spezifikation in Listing 24 spezifiziert. Diese Qualitätseffekt-Spezifikation ordnet für eine Softwarekomponente ein Qualitätsmerkmal für die Integration der Sensoren des Intrusion Detection System zu. Wenn eine Komponente mit einem Sensor des Intrusion Detection System verbunden ist, wird hierfür das Qualitätsmerkmal mit dem entsprechenden numerischen Wert gesetzt. Durch diese Qualitätseffekt-Spezifikation kann die Anzahl der Integrationspunkte des Intrusion Detection System aus den strukturellen Eigenschaften der Softwarekomponenten in beliebigen Softwaresystemen abgleitet werden. Durch den Ansatz der qualitativen Argumentation (Abschnitt 3.3.6) können darauf aufbauend diese Zahlen aufsummiert werden und in PerOpteryx als eine neue Dimension für die automatisierte Entwurfsraumuntersuchung genutzt werden.

```
1 For {
2     Component(
3         Assembly(Required with
4             Component(Name("idsSensor"))
5         )
6     )
7 } Do {
8     NQA("IntrusionDetectionAbility" = "1")
9 }
```

Xtext 24: Qualitätseffekt-Spezifikation für die mRUBiS-Fallstudie

5.2.2.3. Ergebnisse

Um zu validieren, ob der Ansatz der Qualitätseffekt-Spezifikation eine Analyse der strukturellen Eigenschaften einer Softwarearchitektur ermöglicht und hierdurch die automatische Exploration des Entwurfsraums verbessert, wurde mit dem Ansatz die Anzahl der Integrationspunkte eines Sensors direkt in Relation zur Antwortzeit und den Kosten des Softwaresystems gesetzt.

Zu diesem Zweck wurden der mRUBiS Online-Shop einmal mit und einmal ohne die vollständige Integration der Sensoren eines Intrusion Detection System durch PerOpteryx optimiert. Hierbei wurden zweimal 300 Iterationen mit jeweils 30 Kandidaten pro Iteration bewertet. Die Pareto-optimalen Kandidaten mit und ohne Integration der Sensoren sind in Abbildung 5.9 dargestellt. Abbildung 5.9 zeigt auf der X-Achse die Antwortzeit in Sekunden und auf der Y-Achse die Kosten der Architektur-Kandidaten. An dieser Stelle wurden zur deutlicheren Visualisierung lediglich die Pareto-optimalen Kandidaten dargestellt. Die links angelegte Kurve aus den blauen Vierecken zeigt die Pareto-optimalen Kandidaten des Systems, bei welchen keine Sensoren des Intrusion Detection System integriert worden sind. Die rechts angelegte Kurve aus den roten Dreiecken zeigt hingegen die Pareto-

optimalen Kandidaten des Systems, bei welchen an allen fünf Komponenten Sensoren des Intrusion Detection System integriert worden sind.

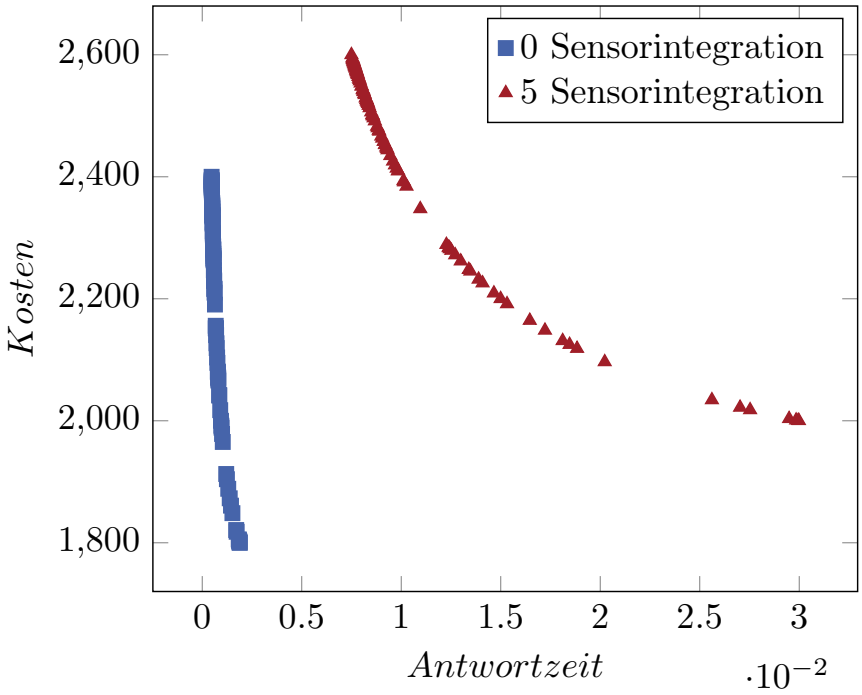

Abbildung 5.9.: mRUBiS-Ergebnisse: Antwortzeit, Kosten und Sensorintegration

Die Auswertung zeigt, wie sich die Pareto-optimalen Kandidaten verändern, wenn auch konkrete strukturelle Eigenschaften einer Softwarearchitektur berücksichtigt werden. Ohne die Auswertung dieser strukturellen Eigenschaften wären die Pareto-optimalen Kandidaten, bei welchen die Sensoren des Intrusion Detection System integriert worden sind, nicht berücksichtigt worden. Da diese allesamt ein schlechteres Verhältnis von Antwortzeit und Kosten haben.

Anhand dieser Ergebnisse kann ein Software-Architekt nun entscheiden, welcher Architekturkandidat nach Antwortzeit, Kosten und Integration der Sensoren des Intrusion Detection System ausgewählt wird. Wenn die spezifischen Projektanforderungen auf einer vollständigen Integration eines Intrusion Detection System basieren, müsste der Software-Architekt die teureren und langsamen Architekturkandidaten auswählen. Soll die Performance weiter verbessert werden, muss dies zum Beispiel durch weitere Hardware Rechenknoten erreicht werden, auf welchen beispielsweise die Management-, Analyse- und Datenbank-Komponente des Intrusion Detection System bereitgestellt werden.

Die Ergebnisse zeigen, dass die letztendlich gewählten Softwarekomponenten stark von den Anforderungen und den verfügbaren Betriebsmitteln abhängig sind. Der Ansatz der Qualitätseffekt-Spezifikation macht diesen Kompromiss deutlich, der sich aus mehreren konkurrierenden Anforderungen ergibt.

5.3. Diskussion der Ergebnisse

Beide Fallstudien zeigen, wie Qualitätsattribute für einzelne Komponenten in Relation zu ihren strukturellen Eigenschaften in ihrem komponentenbasierten Softwaresystem modelliert und dann in einer automatisierten Entwurfsraumuntersuchung genutzt werden können. An dieser Stelle zeigen die beiden Fallstudien auch, dass der Ansatz der Qualitätseffekt-Spezifikation keinen nennenswerten negativen Einfluss auf die Verarbeitungsgeschwindigkeit hat.

Die erste Fallstudie zeigt, wie der Ansatz hilft, komplexere Zusammenhänge zwischen verschiedenen Qualitätsmerkmalen und der Softwarearchitektur zu verstehen. Durch die Ergebnisse in dieser Fallstudie kann ein Architekt entscheiden, ob die bessere Wartbarkeit die schlechtere Performance und die höheren Kosten rechtfertigt.

Die zweite Fallstudie zeigt, wie mittels des Ansatzes die strukturellen Eigenschaften der Softwarearchitektur direkt als eine neue Dimension zur automatischen Exploration des Entwurfsraums genutzt werden kann. Durch die Ergebnisse in dieser Fallstudie kann ein Architekt die Anzahl der Integrationspunkte eines Sensors direkt in Relation zur Antwortzeit und Kosten des Softwaresystems setzen.

Die beiden Fallstudien zeigen, wie sich die Bewertung struktureller Eigenschaften einer Softwarekomponente in ihrem komponentenbasierten Softwaresystem auf die Paretooptimalen Kandidaten auswirkt. In den Fallstudien wurde gezeigt, wie Softwarearchitekten die strukturellen Einflüsse in Verbindung zu den Eigenschaften von Qualitäten für ihre Komponenten modellieren können und wie diese unabhängigen Modelle zusammengesetzt werden können, um die Gesamtqualität eines Systems zu bewerten. Es wurde auch gezeigt, dass sich mit dem Ansatz der Qualitätseffekt-Spezifikation komplexere Zusammenhänge wie die Wirkung der Systemarchitektur auf Qualitäten zueinander modellieren und bewerten lassen. Dies kann dazu führen, dass bestehende Abhängigkeiten, die bisher nicht erkannt wurden, erkannt werden. Ohne Berücksichtigung dieser Effekte würde PerOpteryx nur die optimalen Kandidaten auf Basis der quantifizierten Performance- und Kostenwerte ermitteln.

Die Erstellung der ersten Qualitätseffekt-Spezifikationen, welche das Architekturwissen modelliert, kann vergleichsweise (zeit-) aufwendig sein. Die Erstellung der Modelle muss jedoch nur einmalig erfolgen. Außer bei projektspezifischen Änderungen können die Teile der Qualitätseffekt-Spezifikationen wiederverwendet werden. Die Wiederverwendung des Architekturwissens und die Optimierung ist vergleichbar mit der Wiederverwendung der Komponente selbst oder der Optimierung der Softwarearchitektur und kann daher mit vergleichsweise geringem Aufwand durchgeführt werden. Durch die Nutzung des zusätzlichen Wissens kann der Softwarearchitekt jedoch nun neue Erkenntnisse über das Zusammenspiel der Komponenten und Auswirkungen einzelner Architekturentscheidungen gewinnen, die bisher weder explizit sichtbar waren noch optimiert werden konnten.

6. Fazit

Im letzten Kapitel wird in Abschnitt 6.1 nochmals das Ergebnis dieser Masterarbeit zusammengefasst, in Abschnitt 6.2 werden die Grenzen des Ansatzes aufgezeigt und in Abschnitt 6.3 wird ein Ausblick auf mögliche zukünftige Arbeiten gegeben.

6.1. Zusammenfassung

In dieser Masterarbeit wurde ein Rahmenwerk vorgestellt, um Regeln zu spezifizieren für die Transformation von Qualitätsattributem einer Softwarekomponente in Relation zu ihren strukturellen Eigenschaften in einem komponentenbasierten Softwaresystem. Mit diesem Ansatz kann architekturdefiniertes Wissen in Abhängigkeit der Systemarchitektur parametrisiert werden. Hierdurch können die Qualitätsattribute einer Softwarekomponente, welche erst aus den spezifischen Eigenschaften einer konkreten Softwarearchitektur abgeleitet werden können, spezifiziert und so auch ausgewertet werden. Bestehende komponentenbasierte Ansätze zur automatischen Entwurfsraumuntersuchung können mit diesem neu entwickelten Ansatz kombiniert werden. Entscheidungsunterstützende Prozesse gewinnen neue Dimensionen, welche zur Optimierung genutzt werden können und somit ihren Nutzen erhöhen. Softwarearchitekten können hierdurch neue Erkenntnisse über den Einfluss der Softwarearchitektur auf wichtige Qualitätsmerkmale gewinnen, welche bisher mangels Wissensmodellierung und Optimierungsmöglichkeiten nicht sichtbar waren. Durch diese verbesserten Auswertungen von Qualitätsattributen werden die Werkzeuge eines Softwarearchitekten verbessert, sodass diese besseren Entscheidungen in einem Softwareentwicklungsprozess treffen können.

Für die Validierung des Ansatzes wurden zwei voneinander unabhängige Fallstudien durchgeführt, um dessen Machbarkeit, Angemessenheit und Anwendbarkeit zu zeigen. Hierbei wurde der Ansatz der Qualitätseffekt-Spezifikation in realitätsnahe Kontexte gestellt und mithilfe der Umsetzung der Qualitätseffekt-Transformation in einer automatisierten Entwurfsraumuntersuchung genutzt. Zu diesem Zweck wurde der Ansatz sowohl auf eine wissenschaftliche Fallstudie angewandt wie auch auf ein Beispiel, welches sich auf ein reales Industriesystem bezieht. Dabei wurde das modellierte Wissen genutzt, um die Qualität von Softwarearchitekturen in entscheidungsunterstützenden Prozessen zu optimieren. Hiermit wurde gezeigt, wie der Ansatz helfen kann, Kompromissentscheidungen über die Softwarearchitektur zwischen mehreren Qualitätsmerkmalen unter der Berücksichtigung der strukturellen Eigenschaften des Softwaresystems zu treffen. Die experimentelle Validierung hat gezeigt, wie Qualitätsattribute von Softwarekomponenten in Relation zu ihren strukturellen Eigenschaften ihres Softwaresystems modelliert und zur Bewertung von Softwarearchitektur-Entwurfsentscheidungen während der Entwurfsphase genutzt werden können.

6.2. Annahmen und Einschränkungen

Die Hauptannahme des Ansatzes ist, die Verwendung von Softwarearchitekturmodellen, welche mit Qualitätsinformationen versehen werden können. Auch ist die Annahme des Ansatzes, dass eine Abbildung der Entitäten dieser Softwarearchitekturmodelle auf die Konzepte der UML-Komponenten- und Verteilungsdiagramme möglich ist. Diese beiden Hauptannahme bilden die Grundlage des Ansatzes für die Parametrisierung der Spezifikation von Qualitätsannotationen in Softwarearchitekturmodellen.

Da die Unified Modeling Language eine universell einsetzbare Modellierungssprache ist, gibt es in dieser eine Vielzahl an Regeln und Ableitungsmöglichkeiten von Sprachelementen. Dem gegenüber steht die geringere Anzahl an Regeln und ein eingeschränkter Umfang der domänenspezifischen Sprache des Ansatzes. Aufgrund der Fokussierung auf die deklarative Beschreibung der strukturellen Eigenschaften von Softwarekomponenten, kann ist die domänenspezifische Sprache des Ansatzes weniger mächtig, als die Unified Modeling Language.

Im Rahmen der Modellierung kommt die Zerlegung der strukturellen Eigenschaften zum Einsatz. Dabei wird davon ausgegangen, dass die strukturellen Eigenschaften von Softwarekomponenten in komponentenbasierten Softwaresystemen entsprechend in hierarchisch untergeordnete Teile zerlegbar sind.

6.3. Zukünftige Arbeit

Zum Abschluss dieser Masterarbeit soll ein Ausblick auf weitere Themen erfolgen, welche zusätzliche Ansatzpunkte für eventuell nachfolgende Arbeiten behandeln. Es sollen mehrere mögliche Verbesserungen und Erweiterungen der Qualitätseffekt-Spezifikation kurz erörtert werden, welche im Rahmen dieser Masterarbeit nicht umgesetzt wurden.

So könnten zum Beispiel weitere semantische Klassen von architekturdefinierten Einflussfaktoren auf Qualitätsmodelle bestimmt werden. Die funktionalen Eigenschaften einer Schnittstelle, wie beispielsweise Methodensignaturen, könnten weiter charakterisiert werden.

Auch könnten weitere Klassen von Transformationen auf Qualitätsmodelle von Qualitätsannotationen in Software-Architekturmodellen bestimmt werden. Transformationen für das Qualitätsmodell für die Zuverlässigkeit mit Ausfallwahrscheinlichkeit für Anfragen könnten umgesetzt werden spezifiziert werden. Oder es könnten neben numerischen Werten auch stochastische Ausdrücke transformiert werden.

Auch wären die Umsetzung beliebiger numerischer Transformation via Reflexion (reflection) denkbar. Die Idee wäre hierbei, für alle Entitäten in einer Instanz mittels Reflexion nach Abfragemethoden (getter) und Änderungsmethoden (setter) zu suchen und diese auch gegebenenfalls aufzurufen. Dabei bestimmt der Wert des valueType-Attributes einer numerischen Transformation den Namen Zugriffsfunktion, auf welche die in den Klammern beschriebene Transformation angewendet werden soll.

Eine weitere mögliche Verbesserung des Ansatzes wäre die Integration der Qualitätseffekt-Spezifikation in bereits bestehende Modelle. So könnte beispielsweise die Qualitätseffekt-Spezifikation direkt mit einer Entwurfentscheidung aus dem Strukturmodell verknüpft werden. Dies hätte den Vorteil, dass mit einer Entwurfentscheidung auch der Einfluss auf weitere nicht-quantifizierte Qualitätsmerkmalen verknüpft werden kann.

Danksagung

Viele Personen haben mich bei der Erstellung dieser Arbeit unterstützt, ihnen möchte ich an dieser Stelle danken.

Zuerst möchte ich mich bei meiner Freundin Fabienne Kirschner bedanken, welche mich jeden Tag unterstützt und mir geholfen hat, an dieser Arbeit weiter zu arbeiten.

Vielen Dank an meine Familie Inès, Kerstin und Friedbert Schneider. Alles, was ich bisher erreicht habe, geht auf sie zurück und wäre ohne sie nicht möglich gewesen.

Für das Korrekturlesen und die ausdauernden motivierenden Worte und die jederzeit gewährte Unterstützung sei Cornelia Klenkler ganz herzlich gedankt.

Abschließend danke ich meinem Betreuer Axel Busch für seine hervorragende Unterstützung und für die wertvollen Diskussionen und Vorschläge, welche mir geholfen haben, diese Arbeit zu verbessern.

Literatur

[1] Aldeida Aleti u. a. „ArcheOpterix: An extendable tool for architecture optimization of AADL models“. In: *2009 ICSE Workshop on Model-Based Methodologies for Pervasive and Embedded Software*. IEEE, Mai 2009, S. 61–71. DOI: 10.1109/MOMPES.2009.5069138.

[2] Muhammad Ali Babar und Ian Gorton. „A Tool for Managing Software Architecture Knowledge“. In: *Second Workshop on Sharing and Reusing Architectural Knowledge - Architecture, Rationale, and Design Intent (SHARK/ADI07: ICSE Workshops 2007)*. IEEE, Mai 2007, S. 11–11. DOI: 10.1109/SHARK-ADI.2007.1.

[3] Muhammad Ali Babar, Ian Gorton und Barbara Kitchenham. „A Framework for Supporting Architecture Knowledge and Rationale Management“. In: *Rationale Management in Software Engineering*. Hrsg. von Allen H. Dutoit u. a. Berlin, Heidelberg: Springer Berlin Heidelberg, 2. Feb. 2007, S. 237–254. ISBN: 978-3-540-30998-7. DOI: 10.1007/978-3-540-30998-7_11.

[4] Muhammad Ali Babar u. a. *Software Architecture Knowledge Management. Theory and Practice*. Hrsg. von Muhammad Ali Babar u. a. 1. Aufl. Springer-Verlag Berlin Heidelberg, 2009, S. 279. ISBN: 978-3-642-02374-3. DOI: 10.1007/978-3-642-02374-3. URL: https://www.springer.com/de/book/9783642023736.

[5] Felix Bachmann, Len Bass und Mark Klein. *Preliminary Design of ArchE: A Software Architecture Design Assistant*. Techn. Ber. Software Engineering Institute, Carnegie Mellon University, Sep. 2003. 66 S. DOI: ADA421618. URL: http://www.dtic.mil/dtic/tr/fulltext/u2/a421618.pdf (besucht am 09/2003).

[6] Steffen Becker, Heiko Koziolek und Ralf Reussner. „Model-Based Performance Prediction with the Palladio Component Model“. In: *Proceedings of the 6th international workshop on Software and performance - WOSP 07*. WOSP '07. Buenes Aires, Argentina: ACM Press, 2007, S. 54–65. ISBN: 1-59593-297-6. DOI: 10.1145/1216993.1217006.

[7] Steffen Becker, Heiko Koziolek und Ralf Reussner. „The Palladio component model for model-driven performance prediction“. In: *Journal of Systems and Software* 82.1 (Jan. 2009). Special Issue: Software Performance - Modeling and Analysis, S. 3–22. ISSN: 0164-1212. DOI: 10.1016/j.jss.2008.03.066. URL: http://www.sciencedirect.com/science/article/pii/S0164121208001015.

[8] Lorenzo Bettini. *Implementing Domain-Specific Languages with Xtext and Xtend*. Packt Publishing, Aug. 2016. 426 S. ISBN: 9781786463272. URL: https://books.google.de/books?id=NkrWDQAAQBAJ.

[9] Axel Busch und Anne Koziolek. „Considering Not-Quantified Quality Attributes in an Automated Design Space Exploration“. In: *2016 12th International ACM SIGSOFT Conference on Quality of Software Architectures (QoSA)*. IEEE, Apr. 2016, S. 50–59. DOI: 10.1109/QoSA.2016.10.

[10] Axel Busch u. a. „Modelling the Structure of Reusable Solutions for Architecture-based Quality Evaluation“. In: *Proceedings of the 2nd Workshop on Cloud Security and Data Privacy by Design co-located with the 8th IEEE International Conference on Cloud Computing Technology and Science (CloudCom 2016)*. CloudSPD'16. Luxembourg: IEEE, Dez. 2016, S. 521–526. DOI: 10.1109/cloudcom.2016.0091. URL: http://ieeexplore.ieee.org/document/7830732/.

[11] Rafael Capilla u. a. „A Web-based Tool for Managing Architectural Design Decisions“. In: *SIGSOFT Softw. Eng. Notes* 31.5 (Sep. 2006). ISSN: 0163-5948. DOI: 10.1145/1163514.1178644.

[12] Heeseok Choi, Youhee Choi und Keunhyuk Yeom. „An Integrated Approach to Quality Achievement with Architectural Design Decisions“. In: *Journal of Software* 1.3 (Sep. 2018), S. 40–49.

[13] Sandun Dasanayake u. a. „Software Architecture Decision-Making Practices and Challenges: An Industrial Case Study“. In: *2015 24th Australasian Software Engineering Conference*. IEEE, Sep. 2015, S. 88–97. DOI: 10.1109/ASWEC.2015.20.

[14] Andres Diaz-Pace u. a. „Integrating Quality-Attribute Reasoning Frameworks in the ArchE Design Assistant“. In: *Quality of Software Architectures. Models and Architectures*. Hrsg. von Steffen Becker, Frantisek Plasil und Ralf Reussner. Berlin, Heidelberg: Springer Berlin Heidelberg, 10. Okt. 2008, S. 171–188. ISBN: 978-3-540-87879-7. URL: https://www.ebook.de/de/product/25494917/quality_of_software_architectures_models_and_architectures.html.

[15] Moritz Eysholdt und Heiko Behrens. „Xtext: Implement Your Language Faster Than the Quick and Dirty Way“. In: *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion - SPLASH 10*. OOPSLA '10. Reno/Tahoe, Nevada, USA: ACM Press, Okt. 2011, S. 307–309. ISBN: 978-1-4503-0240-1. DOI: 10.1145/1869542.1869625.

[16] Peter H. Feiler, David P. Gluch und John J. Hudak. *The architecture analysis & design language (AADL): An introduction*. Techn. Ber. Software Engineering Institute, Carnegie Mellon University, Feb. 2006. 144 S. DOI: ADA455842. URL: http://www.dtic.mil/dtic/tr/fulltext/u2/a455842.pdf (besucht am 02/2006).

[17] G. Franks u. a. „Enhanced Modeling and Solution of Layered Queueing Networks“. In: *IEEE Transactions on Software Engineering* 35.2 (März 2009), S. 148–161. ISSN: 0098-5589. DOI: 10.1109/TSE.2008.74.

[18] David Garlan, Robert T. Monroe und David Wile. „Acme: Architectural Description of Component-based Systems“. In: *Foundations of Component-based Systems*. Hrsg. von Gary T. Leavens und Murali Sitaraman. New York, NY, USA: Cambridge University Press, 11. März 2000, S. 47–67. ISBN: 0-521-77164-1. URL: http://dl.acm.org/citation.cfm?id=336431.336437.

[19] David Garlan, Robert Monroe und David Wile. „Acme: An Architecture Description Interchange Language“. In: *Proceedings of the 1997 Conference of the Centre for Advanced Studies on Collaborative Research*. CASCON '97. Toronto, Ontario, Canada: IBM Press, 1997, S. 7–22. URL: http://dl.acm.org/citation.cfm?id=782010.782017.

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

[21] Anton Jansen u. a. „Sharing the Architectural Knowledge of Quantitative Analysis“. In: *Quality of Software Architectures. Models and Architectures*. Hrsg. von Steffen Becker, Frantisek Plasil und Ralf Reussner. Berlin, Heidelberg: Springer Berlin Heidelberg, 10. Okt. 2008, S. 220–234. ISBN: 978-3-540-87879-7. URL: https://www.ebook.de/de/product/25494917/quality_of_software_architectures_models_and_architectures.html.

[22] R. Kazman u. a. „The architecture tradeoff analysis method“. In: *Proceedings. Fourth IEEE International Conference on Engineering of Complex Computer Systems (Cat. No.98EX193)*. IEEE Comput. Soc, Aug. 1998, S. 68–78. DOI: 10.1109/ICECCS.1998.706657.

[23] Donald E. Knuth. „Backus Normal Form vs. Backus Naur Form“. In: *Communications of the ACM* 7.12 (Dez. 1964), S. 735–736. ISSN: 0001-0782. DOI: 10.1145/355588.365140.

[24] Anne Koziolek. „Automated Improvement of Software Architecture Models for Performance and Other Quality Attributes“. Englisch. Diss. Karlsruhe, Germany: Institut für Programmstrukturen und Datenorganisation (IPD), Juli 2011. DOI: 10.5445/KSP/1000032342. URL: http://digbib.ubka.uni-karlsruhe.de/volltexte/1000024955.

[25] Anne Koziolek, Heiko Koziolek und Ralf Reussner. „PerOpteryx: Automated Application of Tactics in Multi-objective Software Architecture Optimization“. In: *Proceedings of the joint ACM SIGSOFT conference – QoSA and ACM SIGSOFT symposium – ISARCS on Quality of software architectures – QoSA and architecting critical systems – ISARCS - QoSA-ISARCS 11*. QoSA-ISARCS '11. Boulder, Colorado, USA: ACM Press, 2011, S. 33–42. ISBN: 978-1-4503-0724-6. DOI: 10.1145/2000259.2000267.

[26] Heiko Koziolek. „Parameter dependencies for reusable performance specifications of software components“. Englisch. Zugl.: Oldenburg, Univ., Diss., 2008. Diss. Institut für Programmstrukturen und Datenorganisation (IPD), 2008. 333 S. ISBN: 978-3-86644-272-6. DOI: 10.5445/KSP/1000009096.

[27] Ioanna Lytra, Huy Tran und Uwe Zdun. *Architectural Design Decision Support Framework (ADvISE)*. Hrsg. von Software Architecture, Faculty of Computer Science und University of Vienna. 1. Jan. 2017. URL: https://swa.univie.ac.at/Software_Architecture/research-projects/architectural-design-decision-support-framework-advise/.

[28] Ioanna Lytra, Huy Tran und Uwe Zdun. „Harmonizing architectural decisions with component view models using reusable architectural knowledge transformations and constraints“. In: *Future Generation Computer Systems* 47 (Juni 2015). Special Section: Advanced Architectures for the Future Generation of Software-Intensive Systems, S. 80–96. ISSN: 0167-739X. DOI: 10.1016/j.future.2014.11.010. URL: http://www.sciencedirect.com/science/article/pii/S0167739X14002441.

[29] Ioanna Lytra, Huy Tran und Uwe Zdun. „Supporting Consistency between Architectural Design Decisions and Component Models through Reusable Architectural Knowledge Transformations“. In: *Software Architecture*. Hrsg. von Khalil Drira. Berlin, Heidelberg: Springer Berlin Heidelberg, 25. Juni 2013, S. 224–239. ISBN: 978-3-642-39031-9. URL: https://www.ebook.de/de/product/25412320/software_architecture.html.

[30] Nenad Medvidovic. *Formal definition of the chiron-2 software architectural style*. UCI-ICS Technical Report 95-24. Department of Information und Computer Science, University of California, Irvine, CA 92717-3425, 19. Apr. 1996, S. 24. URL: http://isr.uci.edu/architecture/papers/TR-UCI-ICS-95-24.pdf.

[31] Peter Naur u. a. „Revised Report on the Algorithmic Language Algol 60“. In: *Algol-like Languages*. Hrsg. von Peter W. O'Hearn und Robert D. Tennent. Boston, MA: Birkhauser Boston, 12. März 2013, S. 19–49. ISBN: 978-1-4612-4118-8. DOI: 10.1007/978-1-4612-4118-8_2.

[32] OMG. *About the Business Process Model And Notation Specification*. Techn. Ber. 2.0.2. Object Management Group, Inc. (OMG), Jan. 2014. 502 S. URL: https://www.omg.org/spec/BPMN/.

[33] OMG. *Object Constraint Language Specification*. Techn. Ber. 2.5.1. Object Management Group, Inc. (OMG), Feb. 2014. 754 S. URL: %5Curl%7Bhttps://www.omg.org/spec/UML/%7D.

[34] OMG. *Unified Modeling Language Specification*. Techn. Ber. 2.4. Object Management Group, Inc. (OMG), 2017. 245 S. URL: https://www.omg.org/spec/OCL/.

[35] Jorge Enrique Pérez-Martínez und Almudena Sierra-Alonso. „From Analysis Model to Software Architecture: A PIM2PIM Mapping“. In: *Model Driven Architecture – Foundations and Applications*. Hrsg. von Arend Rensink und Jos Warmer. Berlin, Heidelberg: Springer Berlin Heidelberg, 29. Juni 2006, S. 25–39. ISBN: 978-3-540-35910-4. URL: https://www.ebook.de/de/product/25407864/model_driven_architecture_foundations_and_applications.html.

[36] Mónica Pinto, Lidia Fuentes und José María Troya. „A Dynamic Component and Aspect-Oriented Platform“. In: *The Computer Journal* 48.4 (2005), S. 401–420. DOI: 10.1093/comjnl/bxh083.

[37] Mónica Pinto, Lidia Fuentes und José María Troya. „DAOP-ADL: An Architecture Description Language for Dynamic Component and Aspect-Based Development“. In: *Generative Programming and Component Engineering*. Hrsg. von Frank Pfenning und Yannis Smaragdakis. Berlin, Heidelberg: Springer Berlin Heidelberg, 19. Nov. 2003,

S. 118–137. ISBN: 978-3-540-39815-8. URL: https://www.ebook.de/de/product/33546837/generative_programming_and_component_engineering.html.

[38] Ralf H. Reussner u. a. *Modeling and Simulating Software Architectures: The Palladio Approach*. Modeling and Simulating Software Architectures. MIT Press Ltd, Dez. 2016. 400 S. ISBN: 9780262034760. URL: https://books.google.de/books?id=QztMDQAAQBAJ.

[39] Ralf Reussner und Wilhelm Hasselbring. *Handbuch der Software-Architektur*. 2., überarb. und erw. Aufl. Heidelberg: dpunkt.verlag, 17. Dez. 2008. ISBN: 978-3-89864-559-1. URL: https://www.ebook.de/de/product/7514035/martin_kuetz_handbuch_der_software_architektur.html.

[40] J. Rumbaugh, G. Booch und I. Jacobson. *The Unified Modeling Language Reference Manual*. Addison-Wesley object technology series. Addison-Wesley Professional, 2010. ISBN: 9780321718952. URL: https://books.google.de/books?id=T7c3RwAACAAJ.

[41] Yves R. Schneider. „Modellierung der Struktureigenscha en von Subsystemen bei architekturellen Entwurfsentscheidungen in komponentenbasierten Systemen“. Bachelor's Thesis. Karlsruhe Institute of Technology (KIT), Mai 2016, S. 51.

[42] Yves Schneider, Axel Busch und Anne Koziolek. „Using Informal Knowledge for Improving Software Quality Trade-Off Decisions“. In: *Software Architecture*. Hrsg. von Carlos E. Cuesta, David Garlan und Jennifer Pérez. Cham: Springer International Publishing, 2018, S. 265–283. ISBN: 978-3-030-00761-4.

[43] Bundesamt für Sicherheit in der Informationstechnik (BSI). *Einführung von Intrusion-Detection-Systemen. Grundlagen*. Version 1.0. Techn. Ber. 31. Okt. 2002, S. 6. 46 S. URL: https://www.bsi.bund.de/SharedDocs/Downloads/DE/BSI/Publikationen/Studien/IDS/Grundlagenv10_pdf.pdf?__blob=publicationFile&v=3 (besucht am 31. 10. 2002).

[44] Mikael Svahnberg und Claes Wohlin. „An Investigation of a Method for Identifying a Software Architecture Candidate with Respect to Quality Attributes“. In: *Empirical Software Engineering* 10.2 (1. Apr. 2005), S. 149–181. ISSN: 1573-7616. DOI: 10.1007/s10664-004-6190-y.

[45] Antony Tang, Yan Jin und Jun Han. „A rationale-based architecture model for design traceability and reasoning“. In: *Journal of Systems and Software* 80.6 (Juni 2007), S. 918–934. ISSN: 0164-1212. DOI: 10.1016/j.jss.2006.08.040. URL: http://www.sciencedirect.com/science/article/pii/S0164121206002287.

[46] Antony Tang u. a. „A comparative study of architecture knowledge management tools“. In: *Journal of Systems and Software* 83.3 (März 2010), S. 352–370. DOI: 10.1016/j.jss.2009.08.032.

[47] Thomas Vogel. „mRUBiS: An Exemplar for Model-Based Architectural Self-Healing and Self-Optimization“. In: *SEAMS'18: 13th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, May 28-29, 2018, Gothenburg, Sweden* abs/1804.00954 (3. Apr. 2018). DOI: 10.1145/3194133.3194161. arXiv: 1804.00954 [cs.SE]. URL: http://arxiv.org/abs/1804.00954.

[48] Jos B. Warmer und Anneke G. Kleppe. *The Object Constraint Language: Precise Modeling with UML*. Addison-Wesley object technology series. Addison-Wesley, 1999. ISBN: 9780201379402. URL: https://books.google.de/books?id=L-BQAAAAMAAJ.

[49] Rainer Weinreich und Iris Groher. „Software architecture knowledge management approaches and their support for knowledge management activities: A systematic literature review“. In: *Information and Software Technology* 80 (Dez. 2016), S. 265–286. ISSN: 0950-5849. DOI: 10.1016/j.infsof.2016.09.007. URL: http://www.sciencedirect.com/science/article/pii/S0950584916301707.

[50] Xiuping Wu und Murray Woodside. „Performance Modeling from Software Components“. In: *ACM SIGSOFT Software Engineering Notes* 29.1 (Jan. 2004), S. 290–301. ISSN: 0163-5948. DOI: 10.1145/974043.974089.

[51] Liming Zhu u. a. „Tradeoff and Sensitivity Analysis in Software Architecture Evaluation Using Analytic Hierarchy Process“. In: *Software Quality Journal* 13.4 (1. Dez. 2005), S. 357–375. ISSN: 1573-1367. DOI: 10.1007/s11219-005-4251-0.

A. Anhang

```
grammar org.palladiosimulator.qes.QualityEffectSpecification
    with org.eclipse.xtext.common.Terminals

generate qualityEffectSpecification
    "http://www.palladiosimulator.org/qes/QualityEffectSpecification"

Model:
    specifications+=QualityEffectSpecification
    (NL+ specifications+=QualityEffectSpecification)* NL+;

QualityEffectSpecification:
    'For' '{' components+=ComponentSpecification
    ('and' components+=ComponentSpecification)* '}'
    'Do' '{' transformations+=TransformationSpecification
    ('and' transformations+=TransformationSpecification)* '}';

ComponentSpecification:
    'Component' '(' properties+=ComponentProperty
    ('and' properties+=ComponentProperty)* ')';

ComponentProperty:
    Name | Identifier | Annotation | Type | Role | Assembly | Resource;

Name:
    'Name' '(' not?='not'? autonym=STRING ')';

Identifier:
    'Id' '(' not?='not'? id=STRING ')';

Annotation:
    'Annotation' '(' not?='not'? annotation=STRING ')';
```

Xtext 25: Erster Abschnitt der Grammatikdefinition

```
32 Type:
33     'Type' '(' not?='not'? type=ComponentType ')';
34
35 enum ComponentType:
36     ANY='AnyComponentType' | BASIC='Basic' | COMPOSITE='Composite';
37
38 Role:
39     'Role' '(' not?='not'? type=RoleType ('with'
40     properties+=RoleProperty ('and' properties+=RoleProperty)*)? ')';
41
42 enum RoleType:
43     ANY='AnyRoleType' |
44     COMPONENT_REQUIRED_PROVIDED='ComponentRequiredProvided' |
45     COMPONENT_REQUIRED='ComponentRequired' |
46     COMPONENT_PROVIDED='ComponentProvided' |
47     INFRASTRUCTURE_REQUIRED_PROVIDED='InfrastructureRequiredProvided' |
48     INFRASTRUCTURE_REQUIRED='InfrastructureRequired' |
49     INFRASTRUCTURE_PROVIDED='InfrastructureProvided';
50
51 RoleProperty:
52     Name | Identifier | Annotation;
53
54 Assembly:
55     'Assembly' '(' not?='not'? type=AssemblyType
56     ('with' component=ComponentSpecification)? ')';
57
58 enum AssemblyType:
59     ANY='AnyAssembly' | REQUIRED='Required' | PROVIDED='Provided';
60
61 Resource:
62     'Resource' '(' properties+=ResourceProperty
63     ('and' properties+=ResourceProperty)* ')';
64
65 ResourceProperty:
66     Name | Identifier;
```

Xtext 26: Zweiter Abschnitt der Grammatikdefinition

```
67 TransformationSpecification:
68     NQA | Reasoning | NumericValue;
69
70 NQA:
71     'NQA' '(' quality=STRING type=TransformationType element=STRING ')';
72
73 enum TransformationType:
74     IS='=' | PLUS='+' | MINUS='-' | MULTIPLICATION='*' | DIVISION='/';
75
76 Reasoning:
77     'Reasoning' '(' quality=STRING '='
78     rules+=Rule (',' rules+=Rule)* ')';
79
80 Rule:
81     'Rule' '(' qualities+=STRING (',' qualities+=STRING)* '='
82     entries+=Entry (',' entries+=Entry)* ')';
83
84 Entry:
85     'Entry' '(' key+=STRING (',' key+=STRING)* '=' value=STRING ')';
86
87 NumericValue:
88     valueType=ID '(' transformationType=TransformationType
89     transformationNumber=NUMBER ')';
90
91 terminal NUMBER:
92     INT ('.' INT)?;
93
94 terminal NL:
95     ('\r'? '\n');
```

Xtext 27: Dritter Abschnitt der Grammatikdefinition