Dienstgütebehandlung im Dienstlebenszyklus – von der formalen Spezifikation zur rechnergestützten Umsetzung

Dissertation

an der
Fakultät für Mathematik, Informatik und Statistik
der
Ludwig-Maximilians-Universität München

vorgelegt von

Markus Garschhammer

Tag der Einreichung: 09. Juli 2004
Tag der mündlichen Prüfung: 02. August 2004

1. Berichterstatter: **Prof. Dr. Heinz-Gerd Hegering**, Universität München
2. Berichterstatter: **Prof. Dr. Bernhard Neumair**, Universität Göttingen

Danksagung

Eine Promotion und die damit verbundene Veröffentlichung ist ohne Unterstüzung nicht denkbar. Deshalb geht an dieser Stelle mein Dank an all diejeniegen, die zum Gelingen dieser Arbeit begetragen haben.

Zuerst an meinen Doktorvater, Herrn Prof. Dr. Heinz-Gerd Hegering, der stets das Ganze im Blick hatte und mich immer wieder ermuntert hat, meine Ideen umzusetzen und an Herrn Prof. Dr. Bernhard Neumair, der mir als Zweitgutachter stets mit konstruktiven Kommentaren zur Seite stand.

Ohne den Rückhalt der Kollegen aus dem MNM-Team wäre diese Arbeit nicht entstanden. Bedanken möchte ich mich deshalb bei den Kollegen, die den Weg der Promotion vor mir gegangen sind, und ihr Wissen und ihre Erfahrung mit mir geteilt haben, aber auch bei jenen Kollegen, die mir den Rücken frei hielten, als es notwendig war.

In dieser Form konnte die Arbeit nur entstehen, weil sich meine Ideen immer auch in der Diskussion mit Studenten messen mussten, die sich mit der praktischen Umsetzung der Konzepte in den verschiedensten Stufen beschäftigten. Besonderer Dank geht deshalb an Cécile Neu, Florian Uhl und Torsten Friedrich, die geduldige Partner auf dem Weg zur Promotion waren.

Besonderer Dank geht an Harry und Bernhard für die immer konstruktiven Disskussionsbeiträge und die Unterstützung bei jeglicher Art von Tagesgeschäft. Helmut, für die stets positive Stimmung und Anette für einen immer freundlich-fröhlichen Start in den Tag und die Bewältigung vieler Kleinigkeiten, die keiner mehr sieht, die allerdings einen Großteil dieser Arbeit ausmachen.

Weiterhin bedanken möchte ich mich bei Stefan, ohne den ich niemals angefangen hätte, Informatik zu studieren. Bei Elli und Franz, die mich zeitlebens in meinem Werden unterstützt haben. Besonders bei Elli, die mit mir so manche Nacht über Vorversionen dieser Arbeit zugebracht hat.

Nicht zuletzt möchte ich mich bei Alex für die Geduld bedanken, mit der sie alle Widrigkeiten des Promotionsvorhabens unterstützt hat, mich immer mit neuer Motivation versorgt hat und mir einen Rückzugspunkt im Promotionsstress geschaffen hat. Nicht vergessen möchte ich auch Louise und Ludwig, die immer wieder für die notwendige Ablenkung und die Besinnung aufs Wesentliche gesorgt haben.

And of course, I did it frutiger.

Zusammenfassung

Mit ein Schlüssel für die leistungs- und qualitätsorientierte Beurteilung oder Abrechnung von IT-Diensten ist die Beschreibung und Erfassung der möglichen, garantierten und erbrachten Qualität eines Dienstes. Dies geschieht an Hand unterschiedlichster Kriterien, so genannter Dienstgütemerkmale, die meist spezifisch für einen Dienst sind. Die Merkmale, an Hand derer ein Dienst in der Betriebsphase beurteilt wird, werden in der Verhandlungsphase, also vor der Implementierung des Dienstes festgelegt.

Die Umsetzung dieser Festlegungen in den folgenden Phasen des Dienstlebenszyklus, der Dienstbereitstellung / -implementierung und der Dienstnutzung ist bisher methodisch nicht unterstützt und muss „von Hand" vorgenommen werden. Somit kann die Auswirkung einer Festlegung in der Verhandlungsphase auf die folgenden Phasen nur durch Expertenwissen und nicht objektiv abgeschätzt werden.

In dieser Arbeit wird ein Verfahren zur formalen Spezifikation von Dienstgütemerkmalen eingeführt, das es erlaubt, die getroffenen Festlegungen automatisch in der Bereitstellungs- und Nutzungsphase eines Dienstes umzusetzen.

Bisher sind zur Spezifikation von Dienstgütemerkmalen nur „Insellösungen" spezifisch für eine Technologie entwickelt worden. In dieser Arbeit wird ein technologieunabhängiger Ansatz aus dem typischen Vorgehen bei der Messung von Dienstgütemerkmalen in unterschiedlichsten Technologien entwickelt.

Dazu wird ein formales Modell eines allgemeinen Messprozesses für Dienstgütemerkmale auf Basis des MNM-Dienstmodells aufgebaut und die Spezifikation eines Dienstgütemerkmals als Verfeinerung dieses Modells formalisiert. Zur Beschreibung dieser Formalisierung wird eine maschinenverarbeitbare Sprache entwickelt, mit deren Hilfe die entsprechenden Modelle später generiert werden können. Für diese Spezifikationssprache wird ein Übersetzer entwickelt, der aus der formalen Definition eines Dienstgütemerkmals automatisch ein Messsystem erzeugen kann. Somit wird eine einheitliche und technologieunabhängige Spezifikation von Dienstgütemerkmalen möglich.

Damit entsteht ein Konzept zur rechnergestützten Spezifikation und Messung von Dienstgütemerkmalen, das in den unterschiedlichsten Szenarien des Dienstmanagements eingesetzt werden kann, wie die Tauglichkeitsprüfung in dieser Arbeit zeigt. Dieses Konzept erweitert bereits bestehende Ansätze, wie etwa CSM (Customer Service Management), um den Aspekt der Qualitätsorientierung und liefert somit die Basis für den Aufbau eines qualitätsorientierten Dienstmanagements.

Abstract

Description and survey of the possible, guaranteed or offered quality of a service (QoS) is one of the key features to establish a performance and quality oriented assesment or accounting of IT-services. This task is accomplished using various criteria - the so called QoS-characteristics - which, in most cases, are specific to a certain service. From the point of view of the service life cycle, characteristics of a service are evaluated with in the usage phase but are usually determined in the negotiation phase before a service is implemented.

The realization of this specifications in the following phases of the service life cycle, service provisioning or implementation and service usage, has not been methodically supported until now and has to be done "by hand". Thus, the impact of a specification made in the negotiation phase on the other life cycle phases can only be estimated with expert knowledge. An objective assessment of this impact is impossible.

The work presented introduces a technique enabling the formal specification of QoS-characteristics, thus allowing an automatic realization of the specification in the provisioning and usage phase of a service. Until now, only isolated approaches on the specification of QoS-characteristics focusing on specific technologies have been made. In this thesis we present a completely technology independent approach distilled from the typical processes used for measuring QoS in different technologies.

In order to implement this idea, a formal model of a common measurement process, based on the MNM service model, is introduced and the specification of a QoS-characteristic is formalized as a refinement of this model. A formal language, capable of facilitating the generation of these models has been developed for this purpose. Consequently, a compiler for this language has been developed to automatically generate a measurement system out of the specification of a QoS-characteristic. This enables the uniform and technology independent specification of QoS-characteristics.

In conclusion, a concept for the computer aided specification of QoS-characteristics has been established which, as an audit included in this thesis demonstrates, can be applied to various scenarios in the area of service management. The concept introduced adds quality orientation to existing approaches like CSM (customer service management) and thus paves the way to quality oriented service management.

Inhaltsverzeichnis

Abbildungsverzeichnis

1. Einleitung

Das integrierte Dienstmanagement umfasst nicht nur den Betrieb eines Dienstes, sondern erstreckt sich über den gesamten Dienstlebenszyklus, beginnt also bereits mit der Verhandlungsphase, in der ein Provider (Dienstanbieter) und Qualitätsmerkmale des zukünftigen Dienstes verhandelt.

Für standardisierte Managementlösungen und -plattformen steht die Betriebsphase eines Dienstes meist im Vordergrund, so dass selten technische und automatische Unterstützung für die Anfangsphase des Dienstlebenszyklus geboten wird. Mit der immer mehr an Bedeutung gewinnenden Qualität eines Dienstes (QoS), stellt das Qualitätsmanagement eine Aufgabe des Dienstmanagements dar, die über den gesamten Dienstlebenszyklus hinweg konsequent verfolgt werden muss. Erster Schritt dazu ist die Spezifikation von Dienstgütemerkmalen, also die Festlegung, welche Eigenschaften oder Merkmale die Qualität eines Dienstes ausmachen. Ausgehend von dieser Spezifikation lässt sich das gesamte Dienstgütemanagement aufbauen.

Besonders für die Betriebsphase existieren vielfältige Ansätze und Konzepte zur Spezifikation der Dienstgüte. Für die restlichen Phasen des Dienstlebenszyklus wurden allerdings kaum Ansätze entwickelt. Integrale Ansätze, die mehrere Phasen des Dienstlebenszyklus überspannen, sind praktisch nicht vorhanden. Eine methodische Unterstützung, welche die Automatisierung von Abläufen innerhalb dieser Phasen und besonders zwischen ihnen ermöglichen würde, wird nicht geboten.

In der vorliegenden Arbeit wird ein Konzept entwickelt, das es erlaubt, Teile des Dienstgütemanagements über den Lebenszyklus hinweg zu automatisieren. Kernstück des Ansatzes ist dabei eine universelle Beschreibungssprache für Dienstgütemerkmale, die auf der Formalisierung grundsätzlicher Abläufe bei der Messung und Spezifikation von Dienstgütemerkmalen basiert. Sie kann damit als Ausgangspunkt für eine phasenübergreifende, integrale Betrachtung der Dienstgüte verwendet werden.

Im Folgenden wird im Abschnitt 1.1 an Hand eines typischen Szenarios aus dem Dienstmanagement die dargestellte Problematik beispielhaft verdeutlicht. Im Abschnitt 2.2 wird sie dann nochmals vertieft und losgelöst vom Beispiel dargestellt. Am Ende dieses Kapitels zeigt Abschnitt 1.3 den Aufbau der Arbeit und gibt einen Überblick über die wichtigsten Teilergebnisse.

1.1. Beispielszenario ASP - Application Service Provisioning

Um beispielhaft den Einfluss von Dienstgütemerkmalen (ihrer Festlegung, ihrer Messung, usw.) auf die einzelnen Phasen des Dienstlebenszyklus aufzeigen zu können, wird an dieser Stelle ein Beispielszenario aus dem Bereich des Application Service Provisioning (ASP) eingeführt. Abbildung 1.1 veranschaulicht das beschriebene Szenario grafisch.

Händler eines Automobilherstellers, die als eigenständige Unternehmen agieren, können bei diesem Hersteller ein Fahrzeugkonfigurations- und bestellsystem nutzen. Mit diesem System können Fahrzeuge konfiguriert werden, also Ausstattungsvarianten, Farbe etc. festgelegt werden. Dabei kann auch die Bestellung des Fahrzeugs vorgenommen werden. Die

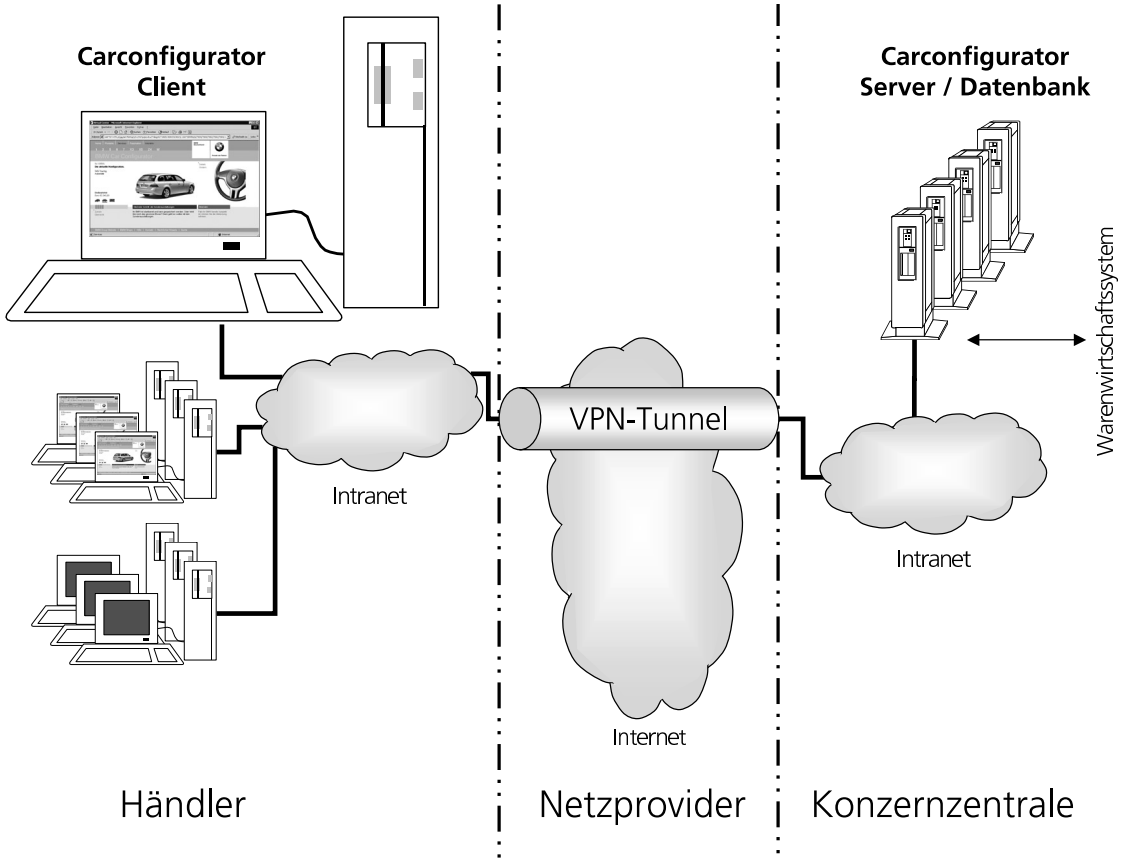

Abbildung 1.1.: Szenario ASP - Bestell- und Konfigurationssystem für Automobilhändler

Anwendung wird vom zentralen Rechenzentrum des Automobilherstellers betrieben. Der Zugriff erfolgt über eine zu diesem Zweck bestehende VPN-Struktur (Virtual Private Network), die vom Automobilhersteller bei einem Telekommunikationsdienstleister eingekauft wird und den Händlern als Bestandteil des Dienstes zur Verfügung steht.

Händler können den Zugriff auf diese Anwendung einkaufen, nutzen damit also einen Spezialfall des application outsourcing. Sie verwenden einen Dienst, der entfernt von einer fremden Organisation betrieben wird. Da für die Nutzung des Dienstes Kosten anfallen und der Dienst zugleich eine zentrale Anwendung für einen Händler darstellt, sind für diesen Dienst vielfältige Dienstgütemerkmale zusammen mit Wertebereichen festgelegt, die im Betrieb des Dienstes eingehalten werden müssen. Der Dienst kann von jedem Händler genutzt werden, allerdings besteht keine Möglichkeit, Funktionalität und Dienstgüte speziell an einen Händler anzupassen. Aus Sicht der Händler steht damit ein standardisierter Dienst mit ebenfalls standardisierten Dienstgütemerkmalen zur Verfügung.

Im Folgenden wird an Hand typischer Dienstgütemerkmale beschrieben, wie diese beim Aufbau des Dienstes festgelegt werden und welchen Einfluss diese Festlegungen auf die Implementierung und den Betrieb des Dienstes haben. Für die beschriebenen Überlegungen wird davon ausgegangen, dass sich der Dienst in der Aufbauphase befindet und relevante Dienstgütemerkmale ausgehandelt werden.

- Verfügbarkeit

  Bei der Festlegung der Verfügbarkeit eines Dienstes werden meist Prozentwerte angegeben („der Dienst ist zu 99,5% verfügbar"). Entscheidend für eine sinnvolle Umsetzung dieses Dienstgütemerkmals ist aber die Festlegung einer technischen Definition von Verfügbarkeit, aus der auch die Bezugsgröße für die Angabe des Prozentwertes hervorgeht. Je nach dem, welche technische Definition von Verfügbarkeit gewählt wird, ist auch die Implementierung des Dienstes und eventuell auch die Clientsoft- und hardware anzupassen. Wird die Verfügbarkeit z.B. als Verhältnis von erfolgreichen Anfragen an den Dienst zu allen ausgelösten Anfragen betrachtet, muss jeder Client in der Lage sein, Informationen zu sammeln, die einen Rückschluss auf den Erfolg von Anfragen zulassen. Zusätzlich müssen diese Informationen an das Management des Dienstes beim Provider übermittelt werden können. (Eine detaillierte Betrachtung der Anwendungsüberwachung aus Sicht der Dienstnutzer findet sich in [HaRe 00])

  Unabhängig von der tatsächlich verwendeten Definition der Verfügbarkeit wird deutlich, dass die, wie wie auch immer geartete, Festlegung dieses Dienstgütemerkmals Einfluss auf die Implementierung des Dienstes hat. Letztere muss je nach Festlegung des Dienstgütemerkmals angepasst werden, damit dieses überhaupt erfassbar und damit als Eingangsgröße für das Management verwendet werden kann.

- Antwortzeit

  Die Begrifflichkeit der Antwortzeit birgt deutlich weniger Unklarheiten als diejenige der Verfügbarkeit. Allerdings gibt es unterschiedlichste Möglichkeiten, die Antwortzeit technisch zu erfassen. Dies könnte ebenso aus Sicht der Nutzer wie aus Sicht des Dienstanbieters anfallen. Im letzteren Fall werden dann Verzögerungen, die durch die Netzanbindung über das VPN entstehen, nicht gemessen.

  Geht man davon aus, dass die Antwortzeit aus Sicht des Endanwenders gemessen wird und die VPN-Anbindung als Teil des Gesamtdienstes „Fahrzeugkonfiguration und -bestellung" begriffen wird, dann hat die Festlegung der Antwortzeit auch Einfluss auf Dienste, die durch den Dienstanbieter zugekauft werden, wie in diesem Fall die VPN-Anbindung.

- Einhaltung der Lieferzeit

  Bei der Bestellung eines Fahrzeugs wird vom System ein Liefertermin genannt. Für die Qualität des Dienstes ist die Einhaltung des dort angegebenen Termins besonders aus Sicht des Endanwenders ein ausschlaggebendes Merkmal.

  Allerdings unterscheidet sich dieses Merkmal deutlich von den bisher genannten. Es betrifft die „inhaltliche" Qualität des Dienstes: Die Übereinstimmung des vom System genannten Liefertermins mit dem tatsächlichen Liefertermin kann zwar leicht überprüft werden, das Management des Fahrzeugkonfigurations- und -bestelldienstes hat aber keinerlei Einfluss auf die Korrektheit der Angaben, da diese aus anderen (meist in sich geschlossenen Systemen) des Automobilherstellers (z.B. Warenwirtschaftssystem) bezogen werden. Ebenso sind die Werte durch die Implementieurng des Dienstes in keiner Weise beeinflussbar.

- Hotlineverfügbarkeit

  Vermutet ein Benutzer eine Fehlfunktion des Dienstes, so kann er sich an die Hotline des Dienstbetreibers wenden. Die Qualität eines Dienstes aus Sicht des Nutzers wird damit auch über die Verfügbarkeit der Hotline bestimmt. Für die Definition von Verfügbarkeit greifen ähnliche Überlegungen, wie sie bereits bei der Verfügbarkeit des Dienstes diskutiert wurden. Werden Entscheidungen über die Verfügbarkeit der Hotline bereits in der Verhandlungsphase getroffen, so hat dies direkten Einfluss auf die Realisierung der Hotline.

- Reparaturzeit / Lösungszeit

  In einem Fehlerfall ist es für den Nutzer entscheidend, wieviel Zeit vergeht, bis dieser behoben wird. Für jede Art von Problemfall eine Lösungszeit zu garantieren, ist praktisch unmöglich. Häufig werden Strafzahlungen - Penalties - vereinbart, deren Höhe an die Dauer der Reparatur gebunden ist.

  Selbst durch die Festlegung der Reparatur- und Lösungszeiten werden Randbedingungen für eine spätere Implementieurng des Dienstes geschaffen. Sind diese sehr eng gesteckt, kann es z.B. notwendig werden, eine Implementierung redundant auszulegen oder zumindest Ersatzgeräte permanent vorrätig zu halten.

1.2. Fragestellungen zur Dienstgütebehandlung im Dienstlebenszyklus

Bereits die exemplarische Auswahl von Dienstgütemerkmalen des oben beschriebenen Szenarios lässt den vielfältigen Einfluss von Dienstgütemerkmalen, insbesondere ihrer Festlegungen auf die Implementierung eines Dienstes und seines Managements erkennen. Obwohl diese Merkmale bereits bei den Verhandlungen über ein neues Dienstangebot festgelegt werden, legen sie direkt oder indirekt Randbedingungen für eine spätere Implementierung des Dienstes fest.

Dabei legen die Dienstgütemerkmale Anforderungen an unterschiedliche Bereiche des Dienstes fest. Häufig werden Anforderungen an die Funktionalität (im Beispiel Verfügbarkeit, Antwortzeit) gestellt. Ebenso entstehen Anforderungen an die „Verarbeitungsqualität" eines Dienstes (im Beispiel die korrekte Angabe der Lieferzeit). Desweiteren sind Merkmale erkennbar, die Anforderungen an die Qualität des Dienstmanagements stellen (im Beispiel Verfügbarkeit der Hotline sowie Reparatur- und Lösungszeit).

Dienstgütemerkmale werden also auf inhaltlicher und funktionaler Ebene ebenso festgelegt wie auf der Ebene des Dienstmanagements selbst. Mindestens die Festlegung der Bedeutung eines Merkmals geschieht bei der Verhandlung um die Bereitstellung eines neuen Dienstes. Meist werden in dieser Phase bereits Grenzwerte für die verschiedenen Dienstgütemerkmale festgelegt. Diese Festlegungen beeinflussen massiv die Implementierung des Dienstes und des zugehörigen Managements.

Dienstbetreiber (Provider) sowie Customer (Kunde) benötigen eine Möglichkeit, den Einfluss der Festlegungen über die Dienstgüte auf die Implementierung abschätzen zu können. Für den Provider gilt es, den notwendigen technischen Auf-

wand einschätzen zu können. Für den Customer ist es wichtig, den durch die eigenen Wünsche erzeugten Aufwand und die daraus resultierenden Kosten einschätzen zu können. Notwendig ist also eine Vorhersage des Einflusses der Festlegung von Dienstgütemerkmalen auf Implementierung und Management eines Dienstes.

Um die Entwicklungszeiten von Diensten gering zu halten und um flexibel auf Kundenwünsche reagieren zu können, ist eine Automatisierung oder zumindest eine Werkzeugunterstützung dieser Vorhersage notwendig. Normalerweise wird diese nämlich nicht explizit durchgeführt, sondern bei den Verhandlungen über einen neuen Dienst werden von vorneherein Experten hinzugezogen, welche die Auswirkungen der getroffenen Festlegungen auf die (noch zu erstellende) Implementierung abschätzen sollen.

Eine methodische und technische Unterstützung der Vorhersage der Auswirkungen von Dienstgütemerkmalsfestlegungen ist damit besonders für einen schnellebigen Markt mit immer kürzeren Entwicklungszeiten notwendig. Zudem werden durch die technische Unterstützung nachvollziehbare Ergebnisse erzielt. Soll diese methodisch/technische Unterstützung aufgebaut werden, sind folgende Fragen zu klären:

- In welcher Beziehung stehen Dienstgütemerkmale zu den einzelnen Phasen des Dienstlebenszyklus?
- Bietet sich eine Möglichkeit zur Formalisierung der Spezifikation von Dienstgütemerkmalen?
- Lässt sich ein Vorgehen standardisieren, das die Auswirkungen dieser Spezifikation auf die Dienstimplementierung und das Dienstmanagement explizit macht?
- Welche Anforderungen werden an ein derartiges Vorgehen gestellt?
- Lassen sich auf Basis der bestimmten Auswirkungen einer Festlegung von Dienstgütemerkmalen Regeln oder Hinweise für eine Implementierung aufstellen, die diese Spezifikation umsetzt?

Im Vorgriff auf die detaillierte Ausarbeitung der Problemstellung in Kapitel 2 sollen an dieser Stelle auch die Fragestellungen genannt werden, die sich bei einer technisch detaillierten Betrachtung des Problemfeldes ergeben:

- Lässt sich eine generische Modellierung der Abläufe bei der Messung eines Dienstgütemerkmals auf Basis eines ebenfalls generischen Dienstmodells aufbauen?
- Können die ein Dienstgütemerkmal ausmachenden Eigenschaften an Hand dieses Modells beschrieben werden?
- Lässt sich ein formales Vorgehen zur Beschreibung von Dienstgütemerkmalen mit dem Modell festschreiben?
- Lässt sich dieses Vorgehen formal dokumentieren, um dadurch die Definition eines Dienstgütemerkmals abzubilden?
- Lässt sich dieses Vorgehen auf Basis der formalen Dokumentation formalisieren?
- Welche Mechanismen sind zum Aufbau dieser Automatisierung notwendig?
- Welche Vorteile entstehen für den Dienstentwickler aus dem gesamten Vorgehen?

Bisherige Ansätze, welche die formale Spezifikation von Dienstgütemerkmalen beschreiben, untersuchen die Einordnung der Problematik in den Dienstlebenszyklus

und die Anwendbarkeit einer Spezifikation über den gesamten Lebenszyklus praktisch nicht. In Teilbereichen sind aber wiederverwendbare Ansätze mit vielversprechenden Konzepten entstanden.

1.3. Aufbau und Ergebnisse dieser Arbeit

Diese Arbeit stellt ein Konzept zur rechnergestützten Spezifikation und Messung von Dienstgütemerkmalen vor. Aus einer formalen Spezifikation von Dienstgütmerkmalen lässt sich rechnergestützt ein Messsystem für diese Merkmale ableiten. Damit werden die Auswirkungen auf die Implementierung, die durch die Festlegungen von Dienstgütemerkmalen entstehen, explizit gemacht und zugleich ein Leitfaden zur Implementierung gegeben. Dieses Konzept kann auch auf managementrelevante Dienstgütemerkmale angewandt werden und liefert damit auch für den Aufbau der Managementfunktionalität eines Dienstes einen Implementierungsleitfaden.

Im Einzelnen wird die Arbeit, wie in Abbildung 1.2 veranschaulicht, in folgenden Schritten aufgebaut:

- Kapitel 2 beschreibt das Umfeld, in das diese Arbeit einzuordnen ist, definiert wichtige Begriffe und formuliert auf dieser Basis die Problemstellung. Daraus wird, in Relation zum beschriebenen Umfeld, eine Abgrenzung der Fragestellung entwickelt. Auf deren Basis werden Anforderungen an eine mögliche Lösung aufgestellt.
- In Kapitel 3 werden verwandte Arbeiten aus dem Umfeld der Fragestellung und bereits bekannte Lösungen für die benannte Problemstellung aufgelistet und an Hand der in Kapitel 2 aufgestellten Anforderungen bewertet.

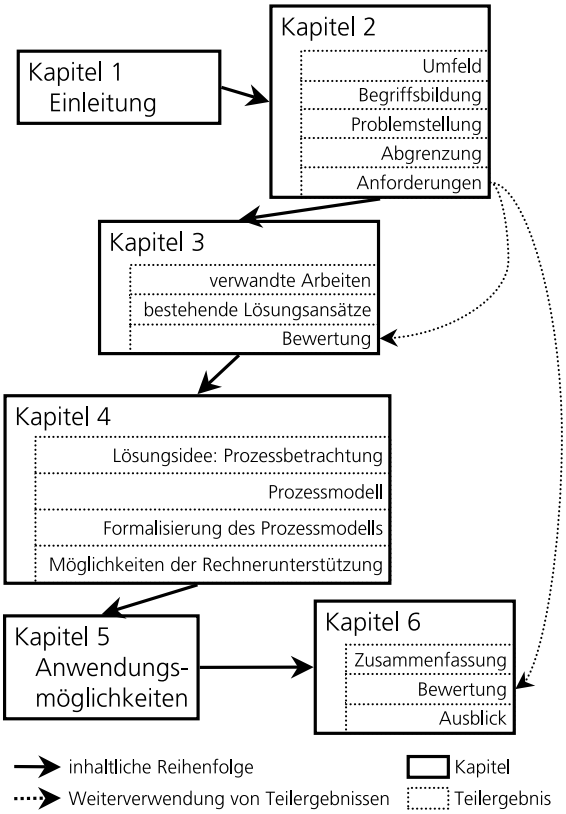

Abbildung 1.2.: Aufbau und Ergebnisse

- Die Lösungsidee, den gesamten Prozess von der Spezifikation eines Dienstgütemerkmals bis zur Implementierung eines Messsystems zu betrachten, stellt das anschließende Kapitel 4 vor. Für diesen Prozess wird ein Modell erstellt und die Ablaufschritte werden formalisiert. Abschließend werden Möglichkeiten betrachtet, den Prozess rechnergestützt ablaufen zu lassen.
- Mögliche Anwendungen des entwickelten Konzepts werden in Kapitel 5 beschrieben.
- Abschließend werden die Kernergebnisse des vorgestellten Konzepts und Anwendungsmöglichkeiten nochmals zusammengefasst und gegen den in Kapitel 2 aufgestellten Anforderungskatalog bewertet. Ein Ausblick auf weitere Fragestellungen rundet das Kapitel ab.

2. Umfeld, Begriffsbildung, Problemstellung und Anforderungen

Dieses Kapitel liefert die Grundlagen für die Betrachtung des Status Quo und die Umsetzung einer Lösungsidee. Dazu wird zunächst das Umfeld des Dienstmanagements strukturiert. Anschließend werden auf Basis dieser Struktur Begriffsdefinitionen abgeleitet, die im weiteren Verlauf der Arbeit Verwendung finden. Mit Hilfe dieser Begriffe lässt sich anschließend die in Kapitel 1 angerissene Fragestellung zu einer detaillierten Problemstellung entwickeln. Diese wiederum lässt sich in die Struktur des Umfeldes einordnen und gegen andere Fragestellungen in diesem Bereich abgrenzen. Schließlich werden detaillierte Anforderungen an eine mögliche Lösung zusammengestellt und in einem Katalog zusammengefasst.

2.1. Umfeld und Begriffsbildung - Spezifikation von Dienstgütemerkmalen

Die plakativen Fragestellungen am Ende von Kapitel 1 deuten durch die darin verwendeten Begriffe bereits das Umfeld an, in dem sich diese Arbeit bewegt: Die Spezifikation von Dienstgütemerkmalen und deren Bezug zum Dienstlebenszyklus, speziell zur Implementierung des Dienstes.

Die Darstellung des Umfeldes dieser Arbeit, auf dessen Basis im Folgenden die in der Arbeit verwendeten Begriffe festgelegt werden, beschreibt entsprechend die durch die Fragestellungen aus der Einleitung angerissenen Themenbereiche Dienstgütemerkmale und Dienste, Dienstlebenszyklus und Spezifikation von Dienstgütemerkmalen.

Dazu werden Ansätze zur Strukturierung des jeweiligen Gebietes kurz vorgestellt. Anschließend wird aus diesen Ansätzen eine integrierte Struktur entwickelt, welche die wichtigsten Aspekte der einzelnen präsentierten Strukturierungsmerkmale zusammenfasst und zur Einordnung der Fragestellung sowie zur übersichtlichen Gliederung verwandter Arbeiten in Kapitel 3 verwendet wird.

2.1.1. MNM-Dienstmodell

Für die Begrifflichkeit eines (IT-)Dienstes existieren vielfältige Definitionen. Für diese Arbeit werden Begriffe verwendet, wie sie im MNM-Dienstmodell [GHHK 02] eingeführt resp. zitiert sind. Die wichtigsten Definitionen werden an dieser Stelle nocheinmal zusammenfassend wiedergegeben.

Die Modellierung eines Dienstes im MNM-Dienstmodell ist nicht auf funktionale Elemente beschränkt, sondern es werden auch die beteiligten Rollen im Bezug zu diesen Elementen dargestellt. Das Dienstmanagement ist integraler Bestandteil des Modells. Damit wird die einheitliche Betrachtung von Funktionalität und Management eines Dienstes im Bezug zu den beteiligten Rollen möglich.

Teile des Dienstes, die nur Customer und User (Nutzer) eines Dienstes betreffen, werden in der sog. *Customer-Side* zusammengefasst. Provider-spezifische Elemente sind in der *Provider-Side* gruppiert. Die relevanten Definitionen eines Dienstes, unabhängig von den beteiligten Rollen sind im *side-independent*-Teil zusammengefasst. Die einzelnen Elemente des Dienstmodells werden im folgenden beschrieben.

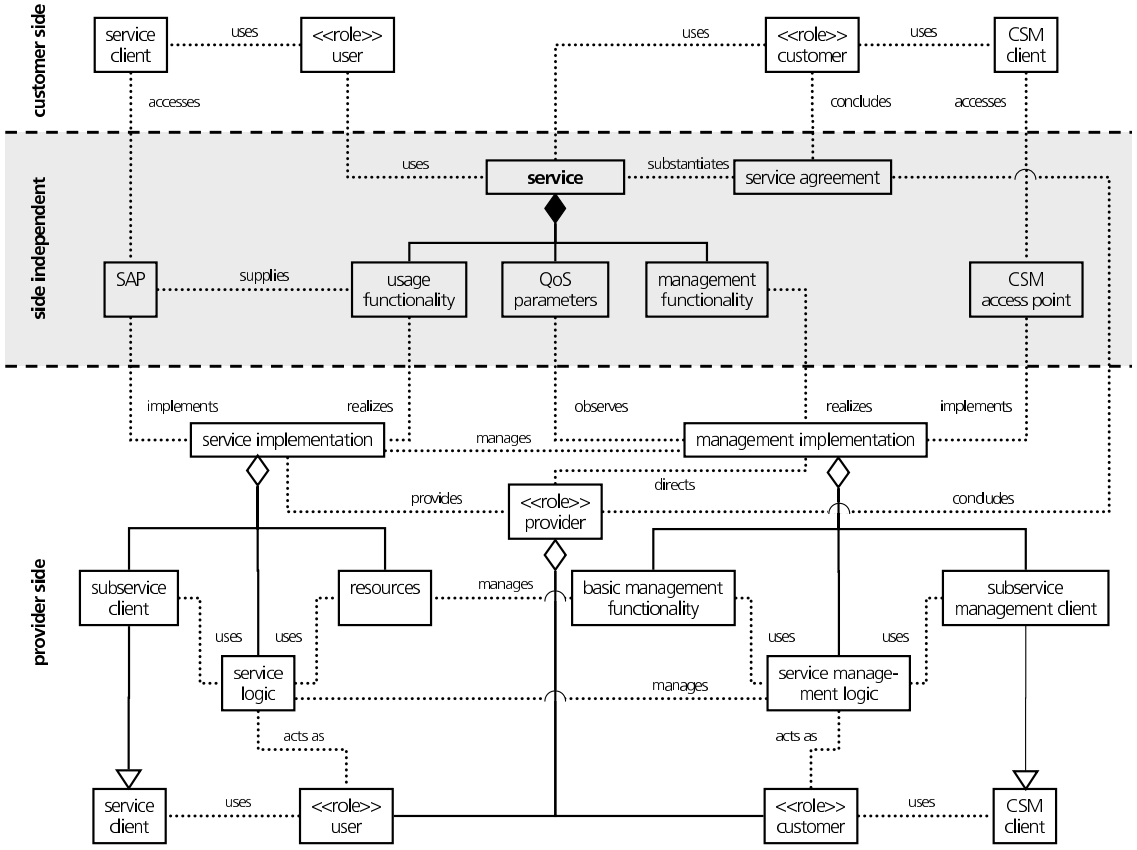

Abbildung 2.1.: MNM-Dienstmodell

2.1.1.1. Rollen

Das MNM-Dienstmodell stellt einen Dienst zunächst im Bezug zu den am Dienst selbst beteiligten Rollen dar. Dazu werden lediglich drei Rollen eingeführt, die zur Abbildung aller Beziehungen zwischen Organisationseinheiten und dem Dienst selbst ausreichen.

Die Rolle *User* repräsentiert Organisationseinheiten (z.B. auch einzelne Personen), die den Dienst resp. seine Funktionalität nutzen. Die Rolle des *Users* steht in enger Beziehung zur Rolle des *Customers*.

Die Vertretung eines *Users* in organisatorischen und Management-Angelegenheiten geschieht durch die Rolle des *Customers*, der organisatorisch eng mit dem *User* verbunden ist. Beide Rollen sind beispielsweise im selben Unternehmen angesiedelt, werden aber durch unterschiedliche Abteilungen oder Mitarbeiter wahrgenommen. *Customer* haben, i.a. im Gegensatz zu *Usern,* Zugriff auf die Managementschnittstelle des Dienstes, das sog. *CustomerServiceManagement*.

Der Betrieb des Dienstes wird schließlich von der Rolle des *Providers* übernommen. Von ihm werden Dienstimplementierung und Dienstmanagement bereitgestellt. Der *Provider* kann zur Erbringung seiner Leistung wiederum auf andere Dienstangebote zurückgreifen. In diesem Fall übernimmt er die Rolle des *Users* und des *Customers* gegenüber dem *Provider* des Sub-Dienstes.

Die Darstellung von Rollen im MNM-Dienstmodell ermöglicht es, klassische Outsourcingbeziehungen, auch innerhalb eines einzigen Unternehmens, einfach darzustellen, da eine Organisationseinheit mehrere Rollen gleichzeitig übernehmen kann.

2.1.1.2. Komponenten eines Dienstes

Das MNM-Dienstmodell gliedert den Dienst (*Service*) in funktionale Komponenten und explizit dargestellte Schnittstellen, den *ServiceAccessPoint (SAP)* und den *CSM-AccessPoint*. Zusätzlich werden die Qualität eines Dienstes beschrieben (wiedergegeben durch die Klasse *QoS-Parameters*) und vertragliche Beziehungen zwischen *Customer* und *Provider* durch die Klasse *SLA* modelliert.

Abbildung 2.1 zeigt in der *provider-side* auch die zur Implementierung eines Dienstes notwendigen funktionalen Komponenten an. Dabei besteht auch die Möglichkeit, dass sowohl zur Implementierung der Dienstfunktionalität als auch zur Implementierung der Managementfunktionalität auf Sub-Dienste zurückgegriffen wird. Provider-eigene Ressourcen, dargestellt durch die Klasse *resources,* werden von der Steuerlogik des Dienstes (*service logic*) mit den Funktionen der Sub-Dienste, die durch *sub-service clients* zugänglich sind, zur Dienstfunktionalität (*service functionality*) zusammengeführt. Auf Seiten des Managements erfolgt ein analoges Zusammenspiel von *basic management functionality*, *sub service management client* und *service logic*.

2.1.2. Dienstlebenszyklus

Bereits die in der Einleitung (siehe Abschnitt 1.2) exemplarisch angegebene Fragestellung weist auf den Einfluss des Dienstlebenszyklus beim Management hin. Deshalb wird die „Idee" der Betrachtung des Dienstlebenszyklus nochmals verdeutlicht.

Die Betrachtung eines Dienstes im Lebenszyklus stellt die dynamischen Aspekte des Dienstes im Bezug zu seinem Umfeld in den Vordergrund. Ein Dienst entsteht, wird betrieben und wieder abgebaut. Diese grundsätzliche Dynamik zeigt sich für jeden Dienst unabhängig von der Dynamik der Informationsverarbeitung.

Der Lebenszyklus wird in Phasen eingeteilt, die nach typischen Aktionen benannt werden, die in der jeweiligen Phase mit dem (und nicht vom) Dienst durchgeführt werden. Wird über den Dienst (seine Eigenschaften) verhandelt, spricht man von der Verhandlungsphase, wird der Dienst genutzt von der Nutzung- oder der Betriebsphase usw.

Die Bezeichnung Lebenszyklus zeigt bereits, dass die Phasen nicht linear aneinander angeordnet werden müssen, sondern auch Zyklen entstehen können. Ein typischer Zyklus ist der Übergang von der Nutzungsphase in die Changephase, in der Anpassungen am Dienst vorgenommen werden und aus der wieder in die Nutzungsphase übergegangen wird.

Je nach Fokus der Beschreibung werden unterschiedliche Phasen für einen Lebenszyklus definiert, die zudem noch unterschiedlich detailliert sein können. Die jeweilige Benennung und Einteilung der Phasen ist für das grundsätzliche Konzept des Lebenszyklus allerdings unerheblich. Vordringlich ist die (halb-formale) Beschreibung des Prozesses, den ein Dienst durchläuft, und die Benennung der Phasen resp. Teilschritte dieses Prozesses.

In dieser Arbeit wird bei der Betrachtung des Dienstlebenszyklus eine Einteilung in die vier Phasen Verhandlung, Bereitstellung, Nutzung und Deinstallation vorgenommen, wie sie in [GHHK 02] beschrieben ist.

2.1.3. Dienstgüte

Der Begriff der Dienstgüte ist bereits im MNM-Dienstmodell (siehe Abschnitt 2.1.1) durch die Klasse *QoS* im Bezug zu den anderen Komponenten und Rollen eines Dienstes dargestellt. In dieser Arbeit wird eine Definition des Begriffes in Anlehnung an [OWS$^+$ 03, Seite 5] verwendet:

> Die Summe aller Eigenschaften eines Dienstes, die Einfluss darauf haben, ob ein spezifizierter oder impliziter Anspruch an diesen Dienst erfüllt werden kann, wird als Dienstgüte bezeichnet.

Eigenschaften sind dabei mögliche Ausprägungen eines Merkmals, respektive fasst ein Merkmal eine Klasse von Eigenschaften zusammen. Zum Beispiel sind „schnell" oder „langsam" Eigenschaften eines Dienstes und gleichzeitig Ausprägungen des Merkmals „Verarbeitungsgeschwindigkeit".

> Einem Dienstgütemerkmal können definierte Werte, spezifische Eigenschaften eines Dienstes, zugeordnet werden, die als Ausprägung des jeweiligen Merkmals bezeichnet werden. Alle Ausprägungen der Dienstgütemerkmale eines Dienstes zusammengenommen beschreiben die (aktuelle) Qualität eines Dienstes.

Nach dieser Definition lässt sich ein Dienstgütemerkmal (als eine Klasse von Eigenschaften) festlegen, ohne die konkrete, aktuelle Ausprägung oder Wertebelegung dieses Merkmals kennen zu müssen.

2.1.4. Spezifikation von Dienstgütmerkmalen und Dienstgüte

Die Spezifikation der Dienstgüte, wie sie beispielsweise in der Verhandlungsphase vorgenommen wird, ist die Grundlage für jegliches an der Dienstgüte ausgerichtetes Management in den Folgephasen. Nach der obigen Definition von Dienstgüte sind bei deren Spezifikation sowohl Dienstgütmerkmale als auch die geforderten Ausprägungen festzulegen.

Spezifikation von Dienstgütemerkmalen Bei der Spezifikation von Dienstgütemerkmalen gilt es, deren Semantik möglichst vollständig zu erfassen und eindeutig zu beschreiben. Da Dienstgütemerkmale Eigenschaften eines bestimmten Dienstes klassifizieren, ist ihre Spezifikation inhärent dienstspezifisch. Dennoch lassen sich viele Merkmale auf unterschiedliche Dienste anwenden (z.B. Verarbeitungsgeschwindigkeit, Reaktionszeit), wenn sie abstrahiert werden.

Spezifikation der Dienstgüte Für die Spezifikation der gewünschten Dienstgüte kann auf die Spezifikation von Dienstgütemerkmalen zurückgegriffen werden, für die dann eine Belegung angegeben wird. Wird eine Spezifikation der für die Dienstgüte als relevant angesehenen Dienstgütemerkmale als existent vorausgesetzt, sind zur Festlegung der Dienstgüte lediglich die relevanten Merkmale und ihre zulässige Belegung zu benennen.

2.1.5. Vorgehensweisen in bestehenden Spezifikationskonzepten

Wie auch bei der Analyse des Status Quo in Kapitel 3 beschrieben, existiert eine Vielfalt von Konzepten zur Beschreibung der Dienstgüte. In [JiNa 04] wird eine grundsätzliche Klassifizierung solcher Konzepte im Multimedia-Umfeld vorgenommen. Die folgende Darstellung von prinzipiellen Vorgehensweisen orientiert sich an dieser Klassifizierung, die Spezifikationstechniken danach unterscheidet, welche Analysemethode für Dienstgüte jeweils als Basis für die Spezifikation verwendet wird.

Datenkommunikationsanalyse Die gebräuchlichsten Dienstgütemerkmale, wie Verzögerung (delay) oder Durchsatz (throughput) werden aus einem kommunikationsorientierten Blickwinkel be-

schrieben. Dienstgütemerkmale und ihre möglichen Ausprägungen werden aus Eigenschaften der Datenkommunikation abgeleitet. Damit wird die Güte eines Dienstes nur indirekt bestimmt, da Kommunikationseigenschaften Eigenschaften des Dienstes nicht direkt abbilden (z.B. vom jeweils verwendeten Protokoll abhängen). Da sich Eigenschaften der Datenkommunikation meist leicht erfassen lassen, ist die Datenkommunikationsanalyse als Basis der Spezifikation weit verbreitet.

Analyse von Methodenaufrufen Aus der objektorientierten Programmierung und den in verteilten Systemen eingesetzten Middleware-Konzepten und Implementierungen hat sich ein weiteres Konzept zur Spezifikation von Dienstgütemerkmalen entwickelt. Qualitätsrelevante Eigenschaften eines Dienstes werden an Methodenaufrufen und ihren Eigenschaften festgemacht (z.B. die für die Codierungen eines Bildes notwendige Zeit).

Diese Spezifikationsmethode setzt voraus, dass eine generische Möglichkeit zum „Abfangen", d.h. zur messtechnischen Erfassung, von Methodenaufrufen existiert. Da komponenten- bzw. objektorientierte Middleware-Implementierungen inhärent diese Möglichkeiten bieten, wird die Spezifikation auf Basis von Methodenaufrufen dort am häufigsten eingesetzt.

Einschleifen von zusätzlichem Programmcode Das „Einschleifen" von zusätzlichem Programmcode dient dazu, die geforderte Dienstgüte automatisch beim Ablauf des Dienststeuerungsprogramms einzuhalten. Dazu wird Programmcode eingebracht, der Reaktionen auf bestimmte Ereignisse (z.B. die Veränderung der CPU-Auslastung) festlegt. Über diese Codefragmente werden somit implizit auch Dienstgütemerkmale bestimmt. Die Festlegung erfolgt allerdings nicht in einer eigenständigen Sprache, sondern in der Implementierungssprache des Dienstes.

Analyse der verwendeten Ressourcen Aus dem System- und Komponentenmanagement ist das Konzept hervorgegangen, Dienstgütmerkmale auf Basis von Eigenschaften der an der Dienstimplementierung, genauer seiner ablaufenden Instanz, beteiligten Ressourcen zu spezifizieren. Grundidee ist dabei, die Dienstgüte durch Aggregation von (aktuellen) Eigenschaften der verwendeten Ressourcen zu ermitteln. Die Art und Weise dieser Aggregation wird dabei als Spezifikation eines Dienstgütemerkmals verstanden.

Diese Art der Spezifikation kann allerdings nur nach erfolgter Bereitstellung des Dienstes erfolgen, weil Informationen darüber benötigt werden, welche Ressourcen tatsächlich von welcher Dienstinstanz verwendet werden, um eine verwendbare Aggregationsfunktion für jedes Dienstgütemerkmal erstellen zu können. Selbst bei bekannter Implementierung eines Dienstes ist die Abbildung eines Dienstgütemerkmals auf ein entsprechendes Aggregationsverfahren nicht trivial.

2.1.6. Anwendungsbreite von Spezifikationskonzepten

Spezifikationskonzepte weisen eine unterschiedliche Anwendungsbreite auf, sind also nicht für die Spezifikation jeglicher Art von Dienstgüte gleichermaßen geeignet. Im Folgenden wird eine Gliederung der möglichen Anwendungsbreite von Spezifikationskonzepten vorgestellt.

2.1.6.1. Dienstgüteaspekt

Dienstgütemerkmale können unterschiedliche, teilweise auch nicht funktionale Aspekte eines Dienstes beschreiben. Die Betrachtung der Dienstgüte unter einem bestimmten Aspekte führt dabei nicht zu überschneidungsfreien Teilmengen der

Diensteigenschaften, sondern spiegelt Eigenschaften eines Dienstes, wenn dieser unter einem bestimmten Fokus betrachtet wird. In dieser Arbeit werden die drei Aspekte Funktion, Management und Inhalt unterschieden. Ein ideales Spezifikationskonzept deckt in seiner Anwendungsbreite alle diese Aspekt ab, d.h., es erlaubt die Spezifikation von Dienstgütemerkmalen aus dem jeweiligen Dienstgüteaspekt.

Dienstgütemerkmale, die den **Funktionsaspekt** abdecken, beschreiben, wie, in welcher Art und Weise, Funktionen vom Dienst erbracht werden. Es handelt sich dabei also um Dienstgütemerkmale, durch welche die Qualität der Dienstfunktionalität beschrieben werden kann.

Auch unter dem **Managementaspekt** wird die Qualität von Funktionalität beschrieben. Allerdings unterscheidet sich diese grundlegend von der sog. Dienstfunktionalität. Unter dem Managementaspekt wird beschrieben, in welcher Art und Weise mit welchen Funktionen das Management erfolgt.

Durch den **Inhaltsaspekt** wird schließlich beschrieben, in welcher Weise Informationen bzw. Inhalte vom Dienst verarbeitet werden. Während die anderen beiden Aspekte das „wie" beschreiben, wird hier das „was" beschrieben.

2.1.6.2. Dienstgueteebenen

In [JiNa 04] wird die Aufteilung von Dienstgütemerkmalen in unterschiedliche Ebenen vorgeschlagen. Diese Einteilung soll auch in dieser Arbeit verwendet werden. Dabei wird zwischen der Nutzer-Ebene (user-level), der Anwendungsebene (application-level) und der Implementierungs-Ebene (resource-level) unterschieden.

Dienstgütemerkmale, die aus Sicht des Nutzers eines Dienstes festgelegt werden, sind auf der Nutzer-Ebene zusammengefasst. Diese sind meist sehr abstrakt, häufig nicht in einem direkten Bezug zu einer bestimmten Anwendung und meist subjektiv, spiegeln also die persönliche Beurteilung eines Nutzers wider.

Die Anwendungsebene fasst Dienstgütemerkmale zusammen, die einer bestimmten Anwendung, allerdings keiner bestimmten Implementierung zuzuschreiben sind. Diese Merkmale beziehen sich auf technische Abläufe innerhalb einer Anwendung und sind damit i.a. objektiv beschreib- und erfassbar. Spezifika einer bestimmten Implementierung werden nicht betrachtet. Damit gelten Dienstgütemerkmale dieser Ebene immer für eine Klasse von Anwendungen (z.B. Videokonferenz, Telefonie, Web). Das angewandte Abstraktionsniveau entspricht in etwa dem des Dienstmodells aus 2.1.1.

Auf der Implementierungsebene werden schließlich Dienstgütemerkmale zusammengefasst, die sich auf eine konkrete Implementierung beziehen (z.B. Speichernutzung). Diese sind nur mit detaillierter Kenntnis der Implementierung erfassbar. Sie bilden letztlich die Grundlage für die Ausprägung aller anderen Merkmale, weisen jedoch ein hohen Grad an Heterogenität auf und sind somit mit generischen Ansätzen schwer fassbar.

2.1.7. Klassifizierungsschema für Spezifikationskonzepte

Da sich diese Arbeit in vielfältiger Weise mit Spezifikationskonzepten für Dienstgütemerkmale beschäftigt, liegt es nahe, aus den bisher eingeführten Begriffsdefinitionen und -gliederungen ein Klassifizierungsschema zu entwickeln, das es erlaubt, bestehende Ansätze einzuordnen und Anforderungen an neue Lösungen griffig darzustellen.

Das hier vorgestellte Schema verwendet zweierlei Einordnungsverfahren. Einerseits wird die Basis eines Spezifikationskonzepts im MNM-Dienstmodell grafisch dargestellt, andererseits werden mögliche Anwendungsbreite und Anwendbarkeit über

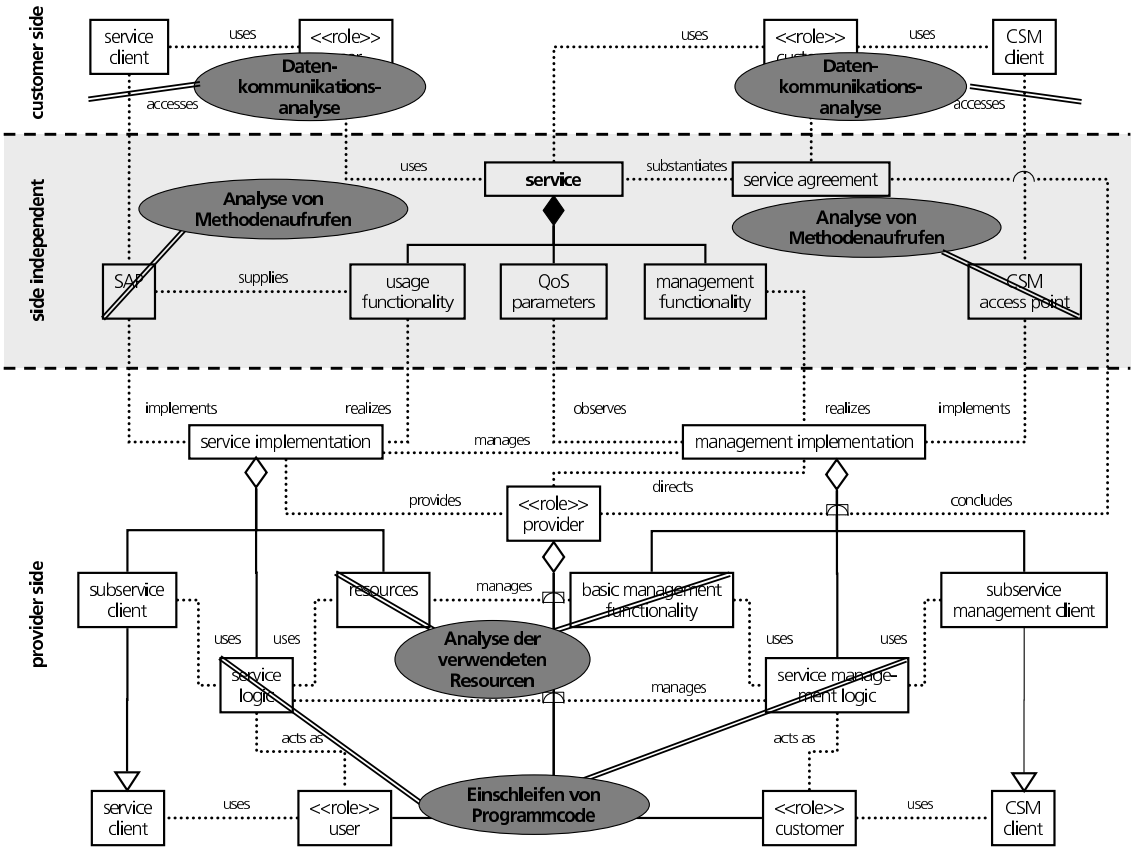

Abbildung 2.2.: Einordnung von Spezifikationsansätzen in das MNM-Dienstmodell

den Lebenszyklus in einer griffigen grafischen Darstellung, dem Dienstgütequader, dargestellt. Das Klassifizierungsschema entspricht dem in [GaRo04] eingeführten Verfahren.

2.1.7.1. Spezifikationsreferenzpunkte im MNM-Dienstmodell

Die vier möglichen Ansätze zur Spezifikation von Dienstgütemerkmalen lassen sich übersichtlich im MNM-Dienstmodell darstellen. Damit entsteht eine „grafische" Vergleichbarkeit von Ansätzen. Zudem lassen sich durch die Einordnung in das MNM-Dienstmodell weitere Eigenschaften der jeweiligen Spezifiaktionsansätze bezüglich ihrer Eignung für das Dienstmanagement ableiten. Abbildung 2.2 zeigt die Einordnung der möglichen Spezifikationsansätze, die im Folgenden detailliert beschrieben wird.

Datenkommunikationsanalyse Datenkommunikation kann im MNM-Dienstmodell grundsätzlich zwischen einem Client und der zugehörigen Schnittstelle beobachtet werden. Im Speziellen findet demnach Datenkommunikation zwischen den Klassen *service client* und *SAP* sowie zwischen *CSM client* und *CSM access point* statt. Somit lässt sich ein Spezifikationskonzept für Dienstgütemerkmale, das auf der Datenkommunikationsanalyse aufbaut, sowohl zur Beschreibung von Merkmalen mit Funktionsaspekt als auch zur Beschreibung von Merkmalen mit Managementaspekt verwenden.

In Abbildung 2.2 ist deutlich sichtbar, dass die Datenkommunikation zwischen Client und SAP die Grenze zwischen dem *side-independent* Teil und der *client-side* überschreitet. Hierin zeigt sich, dass die Datenkommunikationsanalyse von der verwendeten Client-Implementierung und

dem verwendeten Protokoll abhängig ist. Eine implementierungsunabhängige Spezifikation von Dienstgütemerkmalen ist damit mit diesem Konzept nicht möglich.

Analyse von Methodenaufrufen Der Aufruf von Methoden kann prinzipiell an jeder Komponente des MNM-Dienstmodells erfolgen. Im Sinne objektorientierter Umgebungen wird dieses Konzept jedoch an den SAPs umgesetzt, wo Dienst- bzw. Managementfunktionalität im Sinne von Methoden des Dienstes aufgerufen werden kann. Entsprechend wird das Konzept an den Klassen *SAP* und *CSM access point* dargestellt.

Von diesem Spezifikationskonzept werden nur Blöcke des Dienstmodells in Anspruch genommen, die im *side-independent* Teil des Dienstmodells liegen. Es eignet sich damit bestens, Spezifikationen von Dienstgütemerkmalen zu erstellen, die unabhängig von einer bestimmten Implementierung sind.

Einschleifen von zusätzlichem Programmcode In der Implementierungssicht des MNM-Dienstmodells, wie sie in Abbildung 2.2 dargestellt ist, wird die Implementierung eines Dienstes (dargestellt durch die Klasse *service implementation*) in die drei funktionalen Blöcke *subservice client*, *service logic* und *recources* gegliedert. Die Dienstlogik (*service logic*) sorgt dabei für das Zusammenspiel der providereigenen Ressourcen (*resources*) mit den zugekauften Subdiensten, auf die über einen entsprechenden Client (*subservice client*) zugegriffen wird.

Das Konzept des Einschleifens von zusätzlichem Programmcode lässt sich damit im Servicemodell wie folgt darstellen: Die Dienstlogik wird erweitert, so dass aus ihren Ablaufschritten Eigenschaften des Dienstes abgeleitet werden können. Reaktionen auf Veränderungen dieser Eigenschaften werden ebenfalls direkt eingebracht. Nachdem das Dienstmodell allerdings explizit zwischen der Erbringung von Funktionalität und dem Management unterscheidet, wird dieses Konzept sowohl an der Klasse *service logic* als auch an der Klasse *management logic*, welche die Steuerung des Managements realisiert, dargestellt.

Analyse der verwendeten Ressourcen Providereigene Ressourcen sind im MNM-Dienstmodell in zweierlei Hinsicht modelliert. Zum einen durch die Klasse *resources* in der Dienstimplementierung und zum anderen durch die Klasse *basic management functionality*, die vom Provider bereitgestellte Managementressourcen modelliert. Das Spezifikationskonzept der Analyse verwendeter Ressourcen lässt sich dem entsprechend im Dienstmodell an diesen beiden Klassen visualisieren.

Die Darstellung im Dienstmodell zeigt, dass dieses Konzept nur auf Teilen der Dienst- resp. Managementimplementierung aufsetzt, die zudem Teil der *provider-side* sind. Damit wird deutlich, dass dieses Konzept strikt implementierungsspezifisch ist und zudem lediglich Dienstgütemerkmale definieren kann, die nur auf providereigenen Ressourcen aufbauen. Sub-Dienste können nicht in Definitionen eingebunden werden.

2.1.7.2. Dienstgütequader

Der Dienstgütequader setzt diejenigen Dimensionen eines Spezifikationsansatzes zueinander in Beziehung, die nicht durch die Darstellung im MNM-Dienstmodell erfasst werden. Abbildung 2.3 zeigt dieses Modell mit den Dimensionen Dienstgüteebene, Dienstgüteaspekt und Lebenszyklusphase.

Die Dimension der Dienstgüteebene gibt den Abstraktionsgrad eines Dienstgütemerkmals in den drei Stufen Impelmentierung, Anwendung und Nutzer wieder, wie sie in Abschnitt 2.1.6.2 eingeführt wurden. Der Dienstgüteaspekt erfasst als Dimension die in Abschnitt 2.1.6.1 eingeführten

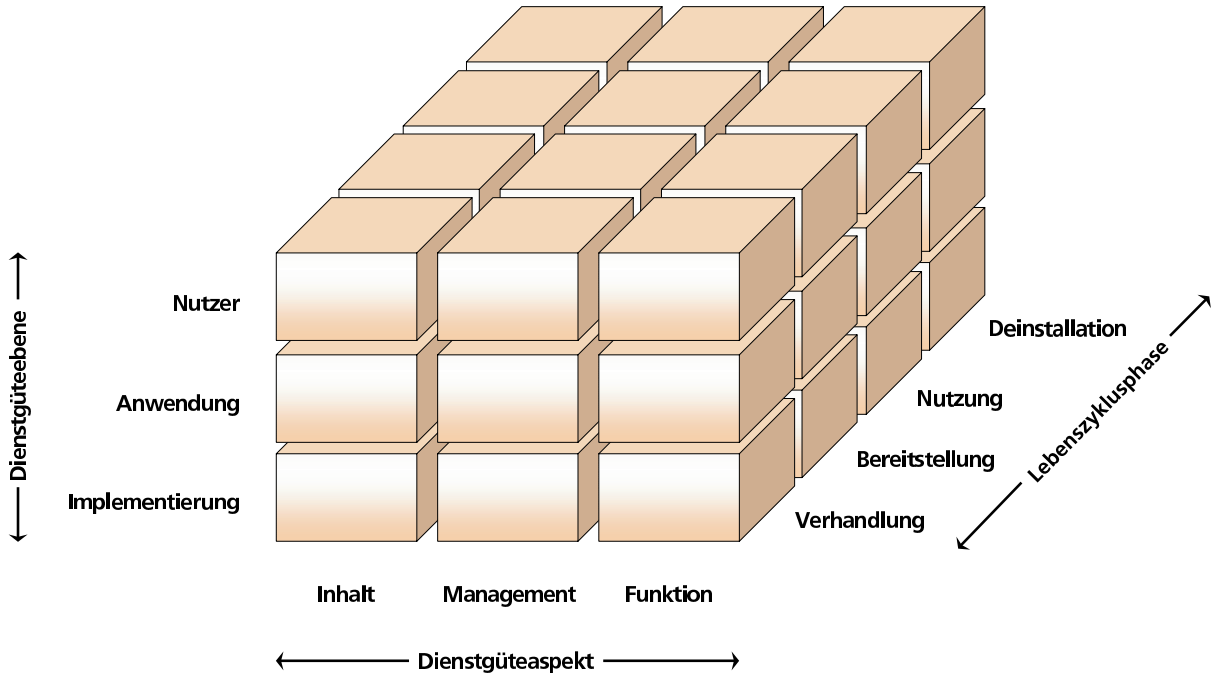

Abbildung 2.3.: Dienstgütequader

Aspekte eines Dienstgütemerkmals nämlich Inhalt, Management und Funktion. Die Phasen des Lebenszyklus, wie sie in Abschnitt 2.1.2 eingeführt wurden, bilden die Stützwerte für die dritte Dimension im Dienstgütequader.

Durch die Aufnahme des Lebenszyklus als Dimension in den Dienstgütequader wird verdeutlicht, dass eine Spezifikationstechnik nicht ohne Bezug zum Dienstlebenszyklus betrachtet werden kann. Wie bei der Vorstellung der unterschiedlichen Spezifikationsarten in Abschnitt 2.1.4 bereits angeklungen, lassen sich diese nicht unbedingt in jeder Phase des Lebenszyklus anwenden oder bauen auf Ergebnissen aus anderen Phasen auf. Diese Zusammenhänge lassen sich im hier aufgestellten Dienstgütequader übersichtlich darstellen.

Die gewählte Darstellung spannt im mathematischen Sinne einen Raum auf. Einzelne Fragestellungen lassen sich somit Teilbereichen des Quaders zuordnen, wie dies in der Grafik bereits durch die Einteilung in „Teilwürfel" angedeutet ist. Somit bietet der Quader neben dem um Spezifikationsreferenzpunkte ergänzten MNM-Dienstmodell ein weiteres Klassifizierungsschema für Techniken zur Spezifikation von Dienstgütemerkmalen.

2.2. Problemstellung

Den Ausgangspunkt der folgenden Überlegungen visualisiert Abbildung 2.4. Im linken Drittel der Abbildung wird die typische Diensthierarchie beschrieben, wie sie auch [GHKR 01] erläutert. Zwischen den beteiligten Rollen (Customer, Provider, Sub-Provider) sind die typischen Fragen des Dienstmanagements angegeben, wie sie sich bereits in der Verhandlungsphase zeigen. Nach rechts sind die Phasen des Lebenszyklus aufgetragen.

Mit ein Ergebnis der Verhandlungsphase ist eine Spezifikation der Dienstgüte, die während des Betriebs des Dienstes erreicht werden soll. Ihre Festlegung bereits in der Verhandlungspahse hat massive Auswirkungen auf die Folgephasen. Um sicherzustellen, dass die getroffenen Vereinbarungen auch eingehalten werden können, müssen diese Auswirkungen bei der Verhandlung bedacht werden.

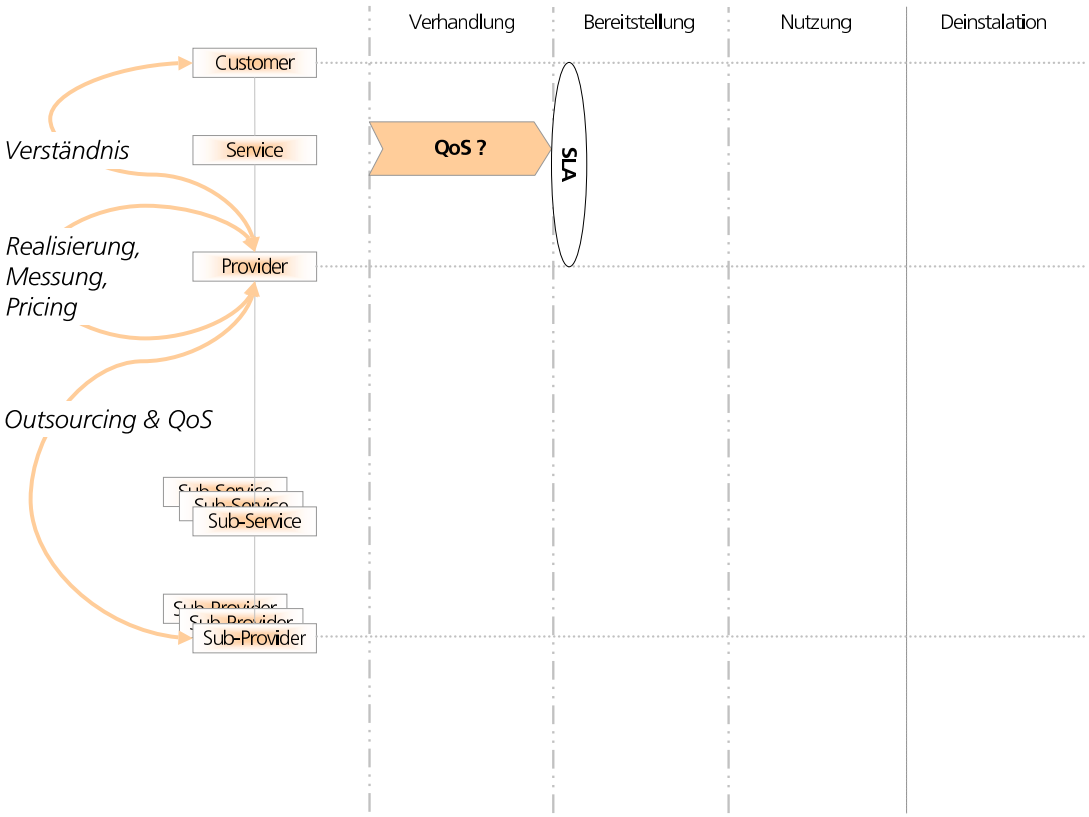

Abbildung 2.4.: Spezifikation von Dienstgüteparametern im Bezug zum Dienstlebenszyklus

2.2.1. Auswirkungen einer Dienstgütespezifikation in den Phasen des Dienstlebenszyklus

Beim Aubau eines Individual-Dienstes ist während der Verhandlungsphase der Dienst normalerweise nicht implementiert und es liegen auch keinerlei Informationen über den „Ablauf" der Dienstimplementierung während der Nutzungsphase vor. Dennoch müssen bei der Verhandlung Entscheidungen über wichtige Eigenschaften des Dienstes in diesen Phasen getroffen werden, die aus der Festlegung der Dienstgüte resultieren bzw. umgekehrt mögliche Festlegungen beeinflussen. Selbst bei Standard-Diensten kann es, sofern die Verhandlungsphase überhaupt ausgeprägt ist, zur Festlegung zusätzlicher Dienstgütemerkmale kommen. Abbildung 2.5 zeigt mögliche Fragestellungen in ihrem Bezug zum Lebenszyklus. Durch die graue Hinterlegung ist angedeutet, in welcher Weise sich Fragen gegenseitig bedingen. Ihre Position in der Grafik gibt an, wann Ergebnisse der getroffenen Entscheidungen sichtbar werden.

Wenn der zu realisierende Dienst auf Subdienste aufsetzt, wird die dargestellte Problematik noch verschärft, da sie für jeden verwendeten Subdienst erneut auftritt, weil in diesem Fall wieder Verhandlungen zwischen einem Customer (in diesem Fall dem Provider des Hauptdienstes) und einem Provider (dem Provider des Subdienstes) stattfinden.

Damit ergibt sich die, aus der Betrachtung des Lebenszyklus motivierte, Anforderung an eine Spezifikation der Dienstgüte, als Basis für Entwicklungen und Entscheidungen in den Folgephasen verwendet werden zu können. Diese Ableitung soll automatisch möglich sein, damit sich weitere Ausprägungen eines Dienstes (z.B.

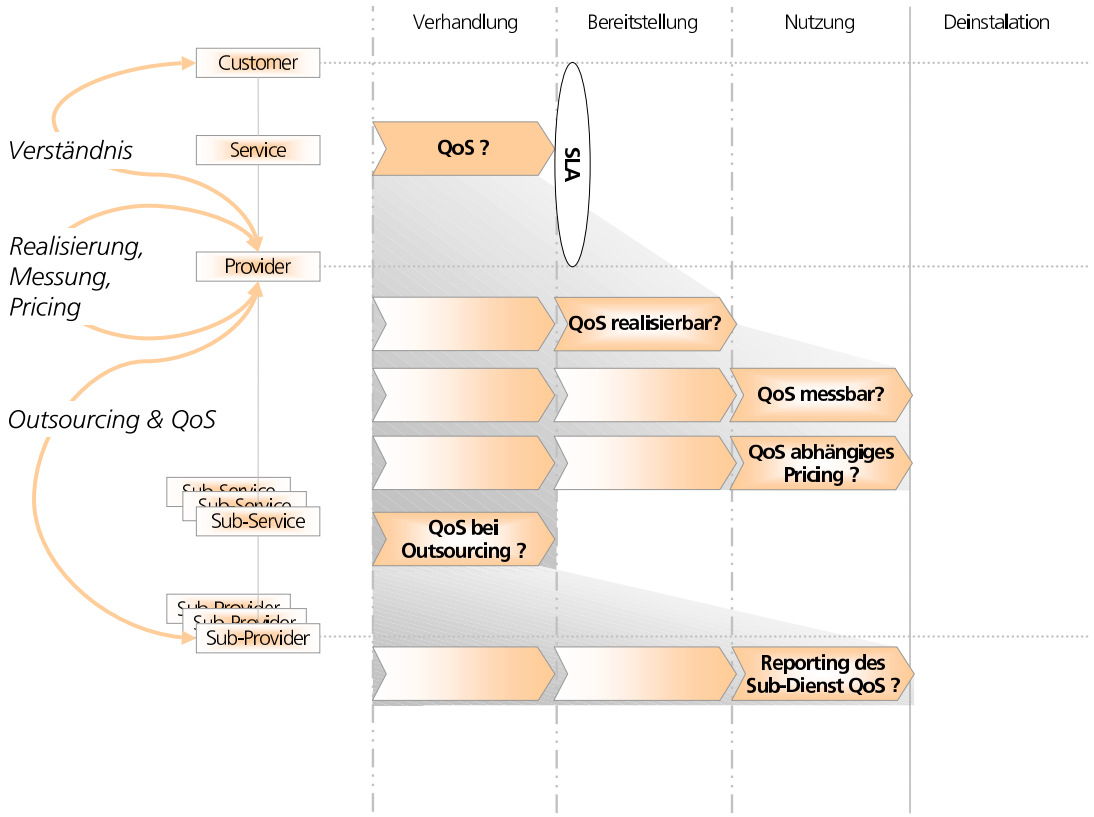

Abbildung 2.5.: Spezifikation von Dienstgüteparametern und ihre Auswirkung auf andere Phasen des Dienstlebenszyklus

die verwendete Implementierung) automatisch ergeben und praktisch durch die Spezifikation der Dienstgüte festgelegt sind. Diese automatische Ableitung bietet dann die Möglichkeit der Vorhersage der Auswirkungen von Dienstgütefestlegungen auf die restlichen Phasen des Lebenszyklus. Abbildung 2.6 veranschaulicht diese Anforderung.

Zusammen mit den Teilfragestellungen, wie sie sich aus der Darstellung der Vorhersageproblematik in Abbildung 2.5 ergeben, lässt sich die Anforderung einer automatischen Ableitung aus der Dienstgütespezifikation wie unter 2.2.2 beschrieben und in Abbildung 2.7 dargestellt, detaillieren. Die Abbildung zeigt zu erzeugende Teilergebnisse als stilisierte Puzzleteile. Die bereits eingeführten phasenspezifischen Fragestellungen sind weiterhin als Pfeile dargestellt.

2.2.2. Automatisierte Umsetzung der Dienstgütespezifikation in den einzelnen Phasen des Lebenszyklus

Folgende Teilergebnisse müssen demnach von einem Automatisierungsprozess, der auf einer Dienstgütespezifikation in der Verhandlungsphase aufsetzt, geliefert werden, um die Auswirkungen einer Dienstgütespezifikation auf die Folgephasen des Dienstlebenszyklus bestimmen zu können. Abbildung 2.8 veranschaulicht die Ableitung dieser Teilschritte aus der Spezifikation der Dienstgütemerkmale in Bezug zum Dienstlebenszyklus nochmals grafisch.

Notwendige Dienstprimitive Eine Dienstgütespezifikation, die in der Verhandlungsphase festgelegt wird, kann auf keinerlei Informationen über eine später folgende Dienstimplementierung zu-

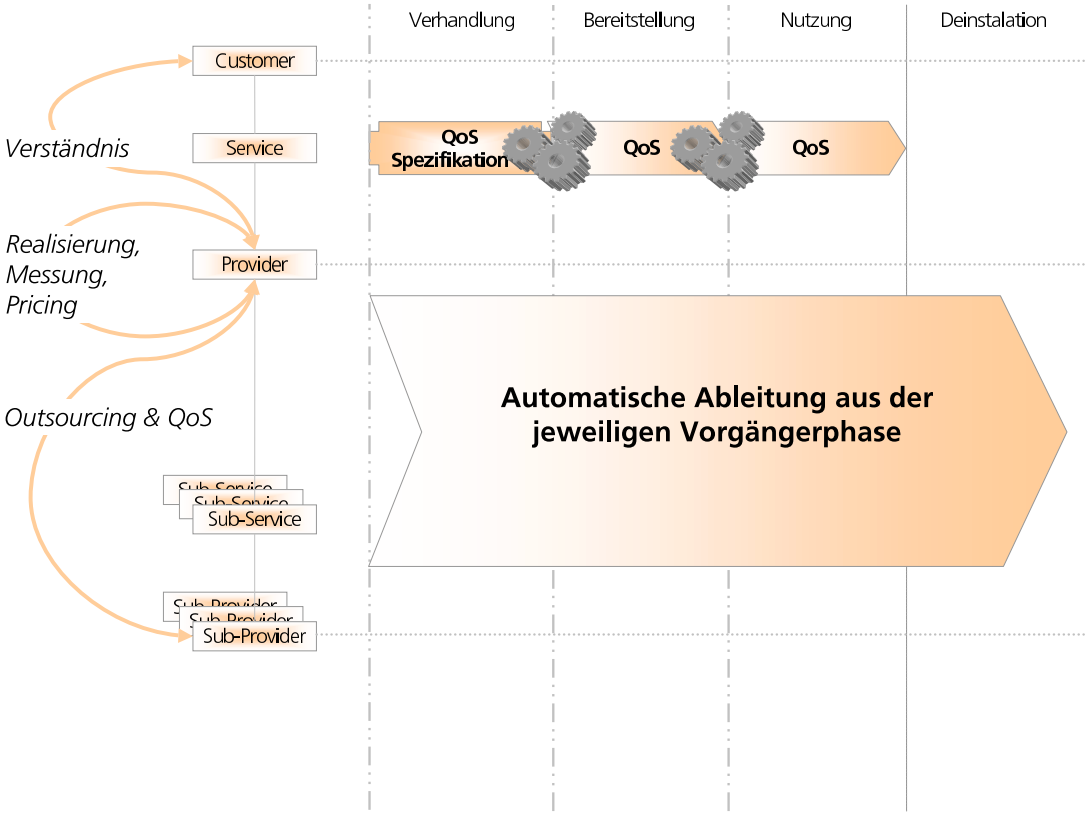

Abbildung 2.6.: Grundsätzliche Problematik der Vorhersage der Auswirkungen einer Dienstgütespezifikation

rückgreifen. Es bietet sich damit an, eine Spezifikationstechnik zu verwenden, die unabhängig von Implementierungsdetails ist. Für die weiteren Überlegungen wird deshalb davon ausgegangen, dass die Spezifikation der Dienstgüte auf Basis der Analyse von (abstrakten) Methodenaufrufen geschieht, wie sie bei der Festlegung der Dienstfunktionalität ohnehin spezifiziert werden.

Bei der Beschreibung dieser Spezifikationstechnik wurde bereits deutlich gemacht, dass es möglich sein muss, entsprechende Methodenaufrufe abfangen zu können, wenn damit auch Werte der jeweiligen Dienstgütemerkmale bestimmt werden sollen. Aus diesem Grund ist es notwendig, dass eine Spezifikation der Dienstgüte eindeutig definiert, auf welche Methodenaufrufe sie zurückgreift. Wird von der objektorientierten Sicht auf einen Dienst abstrahiert, entsprechen die Aufrufe von Methoden dem Aufruf von Dienstprimitiven an einem Dienstschnitt, wie er im OSI-Schichtenmodell für Kommunikationssysteme [ISO 7498] definiert wird.

Dementsprechend lautet die Anforderung, die für die Spezifikation und Bestimmung der Dienstgüte notwendigen Dienstprimitiven auf Basis der getroffenen Spezifikation angeben zu können. Diese Definition stellt die geringst mögliche Funktionalität des SAPs eines Dienstes nach dem MNM-Dienstmodell dar.

Messsystem In der Betriebsphase eines Dienstes muss die aktuell erbrachte Dienstgüte bestimmbar sein. Dazu muss bereits in der Bereitstellungsphase ein entsprechendes Messsystem aufgebaut worden sein, damit während der Nutzungsphase relevante Daten gesammelt werden kön-

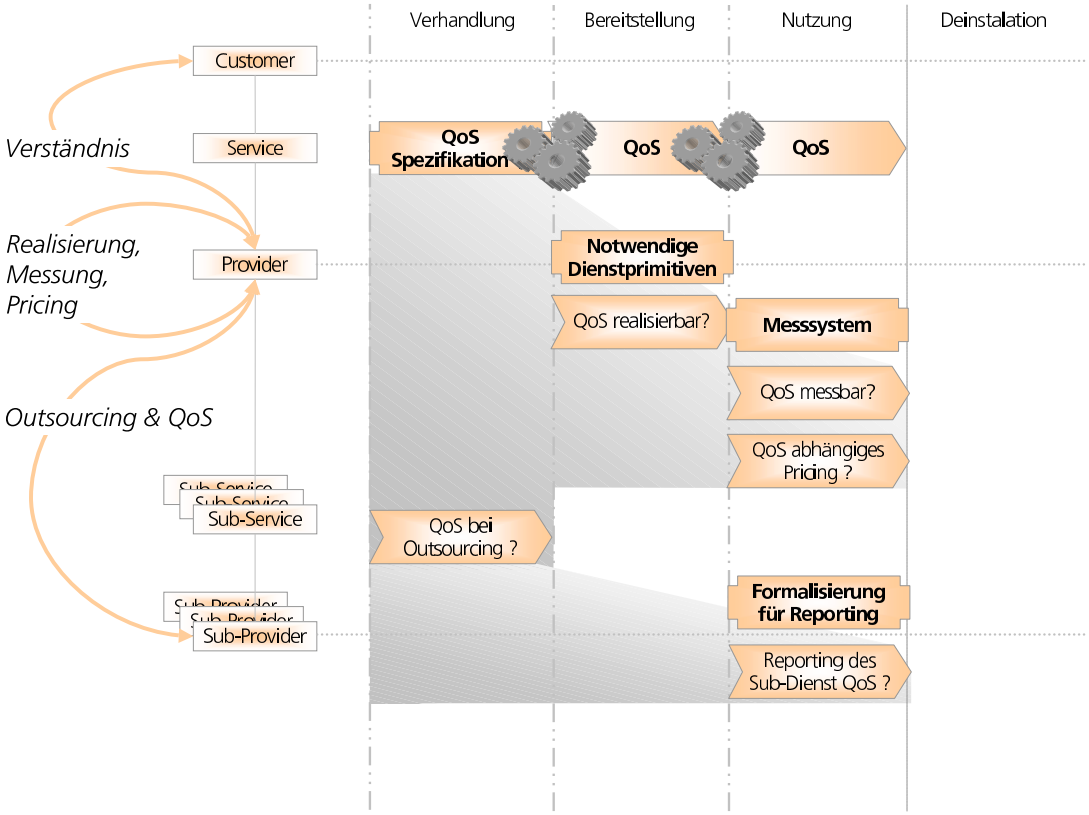

Abbildung 2.7.: Vorhersagepunkte im Detail

nen. Für die Implementierung eines Messsystems müssen die qualitätsbildenden Eigenschaften eines Dienstes für dieses System auch zugänglich sein. Im Fall des hier gewählten Ansatzes der methodenaufrufbasierenden Spezifikation bedeutet dies, dass der Aufruf relevanter Methoden resp. Dienstprimitiven auch erkannt werden können muss.

Formalisierung für Reporting Um auch bei der Einbindung von Subdiensten anwendbar zu bleiben, muss von einer Spezifikationstechnik eine Möglichkeit zum formalisierten Austausch von Dienstgütedaten geschaffen werden. Damit können Informationen über die Dienstgüte eines Subdienstes ausgetauscht werden, ohne dass Detailinformationen über die Implementierung des jeweiligen Subdienstes vorliegen müssen.

2.3. Abgrenzung

Die bisher entwickelte, detaillierte Problemstellung lässt sich mit den in Abschnitt 2.1.7 vorgestellten Schemata, dem MNM-Dienstmodell und dem Dienstgütequader, einordnen. Damit entsteht zugleich eine Abgrenzung zu anderen Fragestellungen.

Bereits zur Detaillierung der Problemstellung wurde angenommen, dass sich ein Spezifikationsansatz nur auf die Analyse von abstrakten Methoden am SAP eines Dienstes stützt, damit er, wie gefordert, bereits in der Verhandlungsphase angewandt werden kann. Als mögliche Lösung kommt damit nur ein Ansatz in Frage, der auf diesem Prinzip fußt.

Im MNM-Dienstmodell lässt sich diese Einschränkung der Fragestellung in Analogie zum verwendeten Spezifikationsansatz darstellen. Dieser wurde in Abschnitt 2.1.7.1 durch Markierungen sowohl an der Klasse *SAP* als auch an der Klasse *CSM*

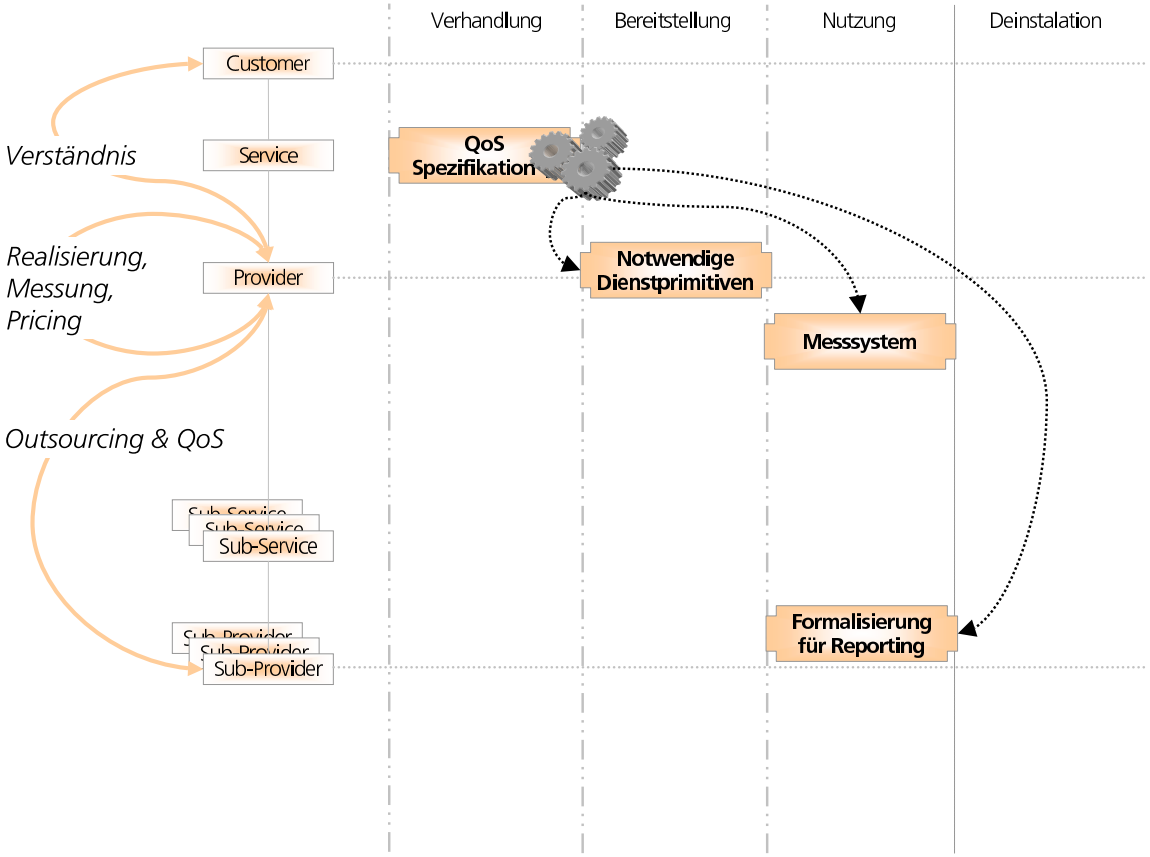

Abbildung 2.8.: Grundsätzliche Vorhersageschritte

access point visualisiert, um zu verdeutlichen, dass das Konzept der Analyse von Methodenaufrufen sowohl an der Schnittstelle zur Dienstfunktionalität als auch an der Schnittstelle zur Managementfunktionalität angewandt werden kann.

Die Problemstellung in dieser Arbeit fokussiert auf die Spezifikaiton von Dienstgütemerkmalen, die sich auf die Dienstfunktionalität beziehen, da die formale Spezifikation von Dienstgütemerkmalen, die sich auf Managementfunktionalität beziehen, bisher praktisch überhaupt nicht untersucht wurde. Entsprechend wird der Fokus der in dieser Arbeit behandelten Problemstellung im MNM-Dienstmodell an der Klasse *SAP* dargestellt. Abbildung 2.9 visualisiert diese Klassifizierung der Problemstellung.

Die Einordnung der Problemstellung lässt sich ebenso im Dienstgütequader veranschaulichen: Wie eben erläutert, liegt der Fokus auf der Spezifikation von Dienstgütemerkmalen mit Funktionsaspekt, wie er im Abschnitt 2.1.6.1 eingeführt wurde. Zugleich stehen anwendungsorientierte Dienstgütemerkmale, wie sie sich an den abstrakten Methoden der Dienstfunktionalität - hier als Primitiven bezeichnet - festmachen lassen, im Vordergrund.

Besonderes Gewicht liegt auf der Unterstützung des Dienstlebenszyklus durch den zu entwickelnden Spezifikationsansatz. Alle Phasen des Lebenszyklus sollen abgedeckt werden, in dem sich die für eine bestimmte Phase notwendigen Spezifikationen aus den in der Verhandlungsphase getroffenen Festlegungen ableiten lassen.

Damit lässt sich die Problemstellung, wie Abbildung 2.10 zeigt, in den Dienstgütequader einordnen: Aus der Fläche, die durch die Dienstgüteebene und den Dienstgüteaspekt aufgespannt wird, wird das Element am Schnittpunkt von An-

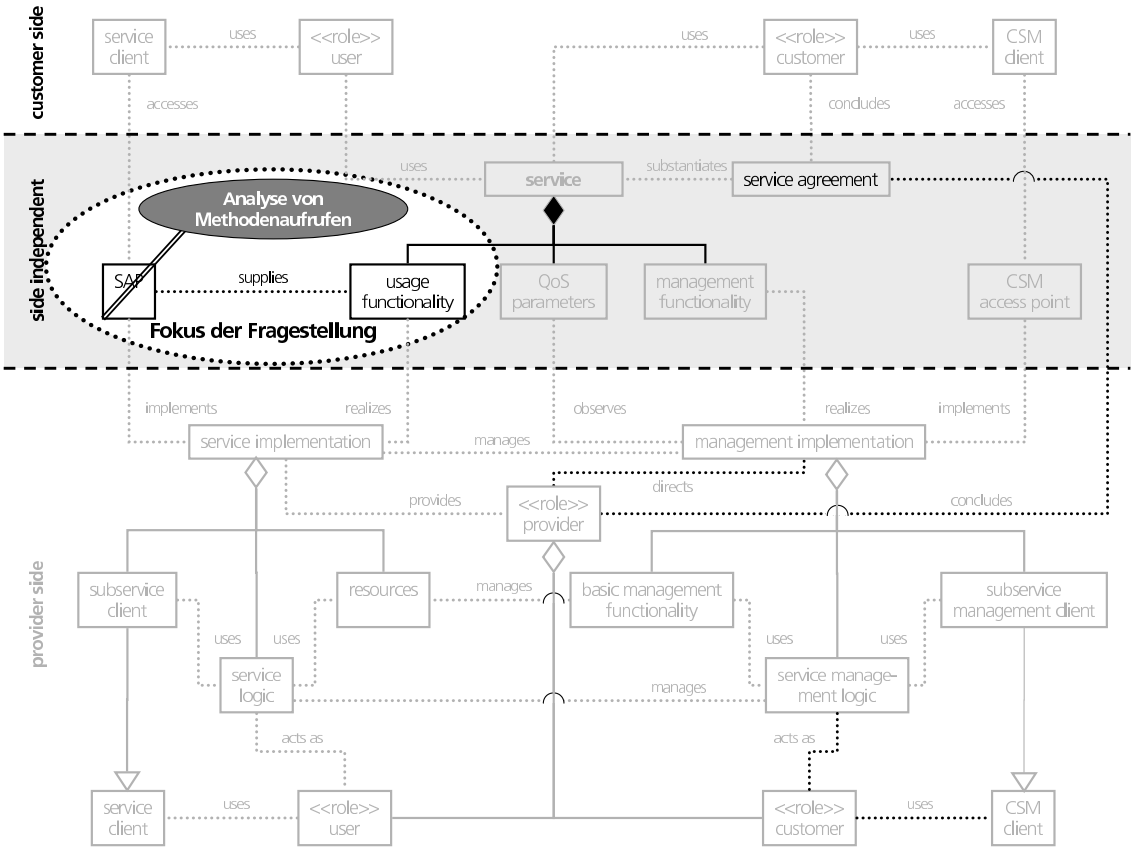

Abbildung 2.9.: Einordnung des durch die Problemstellung geforderten Spezifikationsansatzes in das MNM-Dienstmodell (analog zu Abbildung 2.2)

wendung (Dienstebene) und Funktion (Dienstaspekt) von der Problemstellung abgedeckt. Zugleich werden alle Phasen des Lebenszyklus abgedeckt, so dass der in der Abbildung dargestellte „Streifen" entsteht, der die Einordnung der Problemstellung in das Gesamtfeld der Dienstgütespezifikation visualisiert.

2.4. Anforderungen

Aus der in Abschnitt 2.2 aufgezeigten Problemstellung lassen sich, zusammen mit den in Abschnitt 2.1 angegebenen Klassifikationsmustern, Anforderungen an eine mögliche Lösung der Problemstellung, also eine Spezifikationsmethodik für Dienstgütemerkmale, ableiten und hierarchisch strukturieren.

1. Spezifikation auf Basis der Analyse von Methodenaufrufen

   Wie bereits beschrieben, soll eine mögliche Lösung als Spezifikationsansatz auf die Analyse von Methodenaufrufen, in diesem Fall am Dienstzugangspunkt, zurückgreifen, weil damit bereits im Spezifikationsansatz maximale Dienstorientierung ausgedrückt wird. Auf Basis dieses Ansatzes muss eine Vorhersage der Auswirkungen einer Spezifikation auf alle Phasen des Lebenszyklus möglich sein.

2. eindeutige Spezifikation

   Eine eindeutige Spezifikation erleichtert dabei ihre Verwendung auch als Basis eines formalisierten Reportings. Eindeutigkeit in diesem Sinn bedeutet, dass eine Spezifikation einen

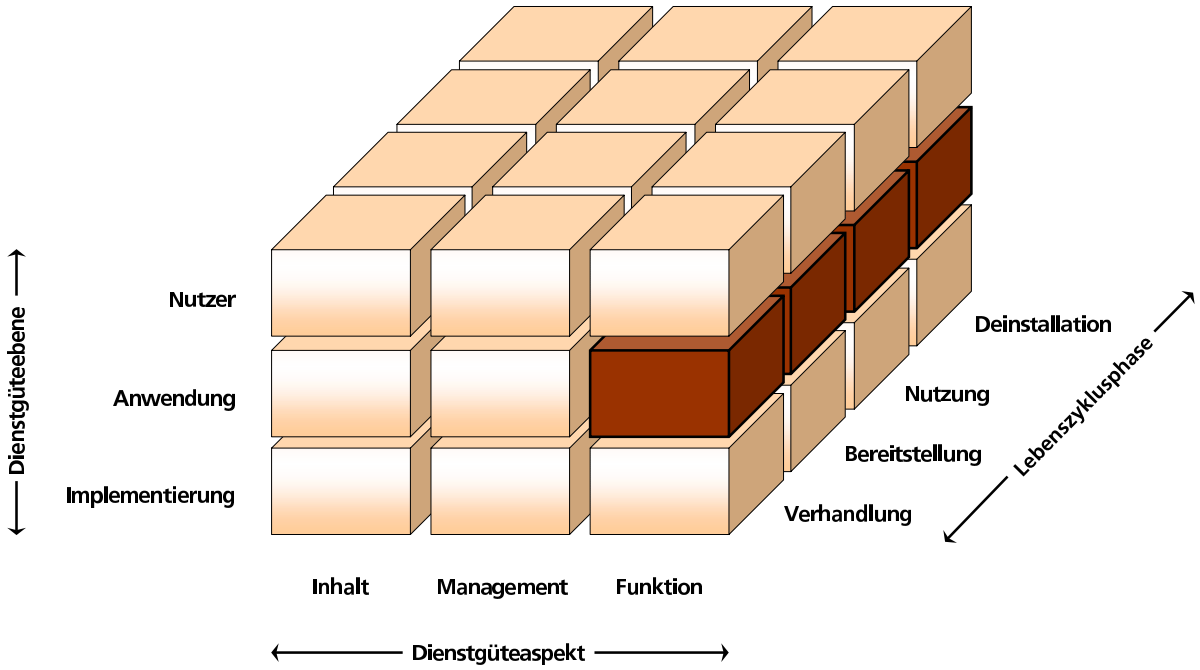

Abbildung 2.10.: Abgrenzung der Fragestellung im Dienstqütequader

möglichst geringen Interpretationsspielraum bei ihrer Umsetzung bietet, so dass auch über Organisationsgrenzen hinweg eine Vergleichbarkeit der Spezifikationen (und ihrer Umsetzung) gewährleistet bleibt.

3. Spezifikation der Semantik

   Das Spezifikationskonzept muss Möglichkeiten bieten, die Semantik eines Dienstgütemerkmals festzulegen, damit Spezifikationen untereinander vergleichbar werden.

4. Einhaltung von Schichtgrenzen

   Um die Spezifikation an Organisationsgrenzen deutlich abgrenzen zu können, also definierte Schnittstellen zum Austausch von Spezifikationsinformationen aufbauen zu können, darf sich eine Spezifikation nicht über Organisationsgrenzen ausdehnen. Um dies sicherzustellen ist es notwendig, dass funktionale Schichtungen in der Spezifikation widergespiegelt werden.

   Da organisatorische Schnitte praktisch nur an funktionalen Schnitten erfolgen, ist es damit inhärent möglich, eine Spezifikation auf unterschiedliche Organisationen „aufzuteilen" bzw. zu verhindern, dass in einer Spezifikation Informationen verwendet werden, die von „fremden" Organisationen stammen. Funktionale Schnitte sind meist einfacher zu identifizieren oder zu kreieren als organisatorische Schnitte und lassen sich deshalb einfacher zur Abgrenzung der Organisationen voneinander verwenden.

5. maschinentaugliche, formale Sprache

   Um eine maximale Rechnerunterstützung erreichen zu können, muss die Spezifikation in einer maschineverarbeitbaren Form, einer formalen Sprache im Sinne einer Programmiersprache, erfolgen.

6. Ergebnisse für den gesamten Dienstlebenszyklus

   Aus der Problemstellung ergibt sich die Forderung nach einer Spezifikationstechnik, die über den gesamten Dienstlebenszyklus einsetzbar ist.

Nach den bisherigen Anforderungen wird dies durch Ableitung der für die jeweilige Phase des Dienstlebenszyklus notwendigen Informationen aus der in der Verhandlungsphase aufgestellten Spezifikation vorgenommen. Damit ergeben sich aus dem Blickwinkel des Dienstlebenszyklus weitere Anforderungen:

7. nur Informationen aus der Verhandlungsphase notwendig

   Da aus der in der Verhandlungsphase verwendeten Spezifikation Ergebnisse für die weiteren Phasen des Dienstlebenszyklus abgeleitet werden sollen, darf diese nur auf Informationen über den Dienst aufbauen, die bereits in der Verhandlungsphase vorliegen.

8. maximale Lesbarkeit der Spezifikation

   Da die Spezifikation in der Verhandlungsphase erstellt wird, muss sie auch für die in dieser Phase agierenden Rollen, den Customer und den Provider, verwendbar bleiben. Da diese Rollen in dieser Phase des Dienstlebenszyklus durch natürliche Personen vertreten werden, muss die Spezifikation auch für diese eine maximal einfache Lesbarkeit aufweisen. Im Idealfall können Spezifikationen ohne Werkzeugunterstützung direkt von den den Customer und den Provider repräsentierenden Personen erstellt werden.

9. Ableitbarkeit eines Implementierungsleitfadens

   Die Forderung nach der Ableitbarkeit von phasenrelevanten Informationen aus der Spezifiaktion in der Verhandlungsphase zeigt sich in den weiteren Phasen des Dienstlebenszyklus durch die Vorgabe eines Implementierungsleitfadens.

10. explizite Spezifikation der notwendigen Dienstprimitiven

    Für die Bereitstellungsphase muss festgelegt werden, welche Primitiven (Funktionen) der zu implementierende Dienst unterstützt und in welcher Weise der Aufruf dieser Primitiven in die Spezifikation von Dienstgütemerkmalen eingeht, damit die Implementierung des Dienstes so vorgenommen werden kann, dass der Aufruf dieser Primitiven auch beobachtbar bleibt und in der Nutzungsphase ein Messsystem (wie unter 11) angebunden werden kann.

11. Spezifikation eines Messsystems

    Zur Erfassung der Dienstgüte in der Nutzungsphase muss ein Messsystem zur Verfügung stehen, das zu den in der Spezifikation festgelegten Dienstgütemerkmalen Werte ermittelt und somit eine Beobachtung der Dienstgüte zulässt. Dieses Messsystem, mindestens jedoch seine Spezifikation, muss aus der Spezifikation der Dienstgütemerkmale in der Verhandlungsphase, möglichst rechnergestützt, ableitbar sein.

In [JiNa 04] werden u.a. allgemeine Anforderungen an Spezifikationssprachen für Dienstgüte postuliert. Der Fokus liegt dabei auf Techniken zur Spezifikation der Dienstgüte in Multimediasystemen. Ein Großteil der Anforderungen kann hier, mit einem deutlich in Richtung der Dienstorientierung verschobenem Fokus, verwendet werden.

12. maximale Ausdrucksmächtigkeit

    Die Ausdrucksmächtigkeit der Spezifikationsprache muss möglichst groß sein oder es müssen Möglichkeiten zu ihrer Erweiterung bestehen. Für die hier behandelte Problemstellung (siehe Abschnitt 2.3) bedeutet dies,

dass alle denkbaren anwendungsorientierten Dienstgütemerkmale spezifizierbar sein müssen. Eine Erweiterbarkeit der Spezifikationstechnik auf Dienstgütemerkmale mit anderen Aspekten bzw. auf anderen Ebenen ist durch die Problemstellung nicht gefordert, wird aber in Abschnitt 5.3 diskutiert.

13. Trennung von Dienstgüte und Dienstgütemerkmal

    Die Trennung des deklarativen Teils einer Spezifikation von konkreten Wertebelegungen, also die strikte Trennung zwischen Dienstgütemerkmal und Dienstgüte, ermöglicht eine übersichtlichere und zugleich besser wiederverwendbare Spezifikation.

14. Wiederverwendbarkeit

    Spezifikationen, die für einen bestimmten Dienst entworfen wurden, sollten auch für andere, im Sinne der Objektorientierung ähnliche, Dienste verwendet werden können. Ein hoher Grad an Wiederverwendbarkeit trägt damit implizit zur Standardisierung der von Diensten gebotenen Dienstgütemerkmale bei und erleichtert damit, durch reduzierte Variantenvielfalt, das Qualitätsmanagement.

Zusätzlich zu den hier aufgestellten Anforderungen wird von [JiNa 04] noch die Unabhängigkeit von einer konkreten Implementierung (independability) gefordert. Der in dieser Arbeit verwendete Ansatz der Dienstorientierung erfordert diese Unabhängigkeit inhärent. Der gewählte Spezifikationsansatz, die Analyse des Aufrufs von Methoden (des Dienstes), setzt diese Anforderung direkt um, so dass sie explizit nicht mehr betrachtet werden muss.

Weiterhin werden in [JiNa 04] Erweiterungsmechanismen für Spezifikationsansätze gefordert. Nachdem in dieser Arbeit Dienstgütemerkmale auf der Anwendungsebene mit Funktionsaspekt betrachtet werden (siehe Abschnitt 2.3) und zugleich eine Ausdrucksmächtigkeit gefordert wird, welche diese Dienstgütemerkmale komplett erfasst, ist eine Betrachtung der Erweiterbarkeit im strikten Fokus der Fragestellung nicht notwendig. Diese wird aber im Ausblick dieser Arbeit in Abschnitt 6.3 vorgenommen.

3. Verwandte Arbeiten

Das folgende Kapitel liefert einen Überblick über bestehende Ansätze zur Spezifikation der Dienstgüte. Der Schwerpunkt liegt dabei auf Ansätzen, die sich nicht auf eine bestimmte Technologie beschränken, sondern auf einem möglichst weit gefassten Dienstbegriff aufbauen. Häufig ist dieser aber eng am Dienstbegriff verteilter, middlewarebasierter Anwendungen orientiert, so dass wirklich generische Lösungsansätze selten sind.

Die Darstellung gliedert sich deshalb in die Beschreibung von Ansätzen mit spezifischem Dienstbegriff und solchen mit explizit generisch gehaltenem Dienstbegriff. Einführend werden die wichtigsten Standardarchitekturen und Frameworks zur Beschreibung und Implementierung von Diensten auf ihre Aussagen hinsichtlich der Spezifikation von QoS (Quality of Service) untersucht.

Die Beschreibung der Spezifikationsansätze erfolgt entlang der in Abschnitt 2.4 aufgestellten Anforderungen und untersucht dabei insbesondere den Aspekt der Integration und Unterstützung des Dienstlebenszyklus. Am Ende dieses Kapitels werden die Ergebnisse tabellarisch zusammengefasst, so dass ein komprimierter Überblick über den Stand der Forschung entsteht.

3.1. Rahmenwerke zur Modellierung von Diensten und Dienstgüte

Unterschiedlichste Rahmenwerke, die sich teils mit der Modellierung von Diensten im allgemeinen, teils explizit mit der Modellierung von Dienstgüte auseinandersetzen, werden im Folgenden vorgestellt. Diese Rahmenwerke bilden vielfach die Basis für die im weiteren Verlauf dieses Kapitels vorgestellten Spezifikationsansätze.

3.1.1. ITU-T X.641 Quality of Service Framwork

Mit [X.641] wurde von der ITU-T (International Telecommunication Union) / ISO (International Organizaiton for Standardization) ein Standard vorgelegt, in dem Begriffsdefinitionen im Bereich der Dienstgüte für einen möglichst weit gefassten Dienstbegriff vorgenommen werden. Das Framework (Rahmenwerk) legt allerdings keine Sprache zur Spezifikation von Dienstgütemerkmalen fest, sondern definiert informell einen Katalog von Dienstgütemerkmalen. Die für die einzelnen Phasen des Dienstlebenszyklus typischen Vorgehensweisen bei der Umsetzung einer Spezifikation werden benannt und es wird zudem festgelegt, welche Ergebnisse aus dem jeweiligen Vorgehen resultieren müssen. Eine konkrete Implementierung wird allerdings nicht vorgeschlagen, sondern es wird davon ausgegangen, dass diese speziell im jeweiligen Anwendungsgebiet geschieht. [ODP-QOS] setzt dieses Vorgehen um und verwendet damit [X.641] als Designbasis.

3.1.2. ITU-T RM-ODP

Das RM-ODP [X.901] stellt ein Framework zum Design verteilter Anwendungen dar. Es bietet Systematiken und Orientierungshilfen, sowohl für das Entwickeln einer verteilten Anwendung von den Anforderungen bis zur Implementierung, als auch für die Festlegung von (Architektur)Standards verteilter Anwendungen.

Um die Entwicklung übersichtlich zu halten, definiert das Modell verschiedene Sichten, sog. Viewpoints, der verteilten Anwendung. Die Viewpoints stellen kein hierarchisches Schichtenmodell, sondern nur

verschiedene, sich teilweise überdeckende Blickwinkel dar. Das Referenzmodell liefert eine Reihe von Anforderungen, nach deren vollständiger Umsetzung es möglich ist, eine verteilte Anwendung ohne Rücksicht auf die Verteilung selbst zu entwickeln.

Die Einbindung von Dienstgüte und ihrer Beschreibung in dieses Referenzmodell wird in [ODP-QOS] diskutiert. Es werden grundsätzliche Aspekte bei der Umsetzung einer Dienstgütespezifikation in den einzelnen Viewpoints vorgenommen. Eine Spezifikationssprache wird dabei nicht entwickelt, vielmehr werden Vorgehensweisen informell beschrieben. Eine endültige Umsetzung der vorgeschlagenen Konzepte in das Referenzmodell ist bis heute nicht geschehen. Dennoch können die beschriebenen Konzepte bei der Entwicklung von Spezifikationstechniken verwendet werden. (siehe Abschnitt 3.2.1 und [AsVi 00, AVLdVB 01]).

3.1.3. TINA - Telecommunications Information Networking Architecture

TINA (Telecommunications Information Networking Architecture) [ChMo 95] ist eine Weiterentwicklung des RM-ODP (siehe Abschnitt 3.1.2) aus dem Blickwinkel großer TK-Anbieter. TINA bietet einen integrierten Ansatz für die Spezifikation die Implementierung und den Ablauf von objektorientierten, verteilten Systemen, die TK-Dienste im weitesten Sinne realisieren. Dabei verfolgt die Architektur die Vision einer netzweiten Software-Interoperabilität, verbunden mit wiederverwendbaren Bausteinen, wodurch die Implementierung von TK-Diensten erleichtert werden soll und zugleich die Komplexität unterliegender Technologien weitest möglich verschattet wird.

Das TINA-Consortium, das die Entwicklung von TINA vorantreibt, hat auch den Bereich der Dienstgüte adressiert. Dabei wurden beispielsweise typische Verfahren zur Erbringung typischer Dienstgüten diskutiert und Vorgehensweisen für die Implementierung dieser Verfahren auf Basis der TINA-Architektur untersucht. Die Spezifikation der Dienstgüte und der Dienstgütemerkmale ist Teil der Spezifikationssprache ODL (Object Description Language) [ODL96]. ODL ist eine echte Obermenge der IDL (Interface Description Language) [ITU X.920], wie sie beispielsweise CORBA (Common Object Request Broker Architecture) [CORBA] verwendet. Die Ausdrucksmöglichkeiten von ODL bezüglich der Dienstgüte werden in Abschnitt 3.2.6 an Hand des aufgestellten Anforderungskatalogs aus Abschnitt 2.4 diskutiert.

3.2. Ansätze für spezifische Dienste

Der Großteil der untersuchten Spezifikationsansätze widmet sich spezifischen Diensten. Die folgende Darstellung zeigt die dabei erreichte Breite, die von einem durch verteilte Anwendungen und Objektorientierung motivierten Dienstbegriff, über Echtzeit-Systeme und Grid-Systeme bis hin zu Kommunikationsdiensten reicht.

3.2.1. UML-Q, UML-M - UML-Metamodellerweiterungen zur Dienstgütemodellierung

Die Möglichkeit, die Modellierung und damit die Spezifikation von Dienstgüte in vorhandene Modellierungstechniken zu integrieren, wird in [AsVi 00] und [AVLdVB 01] untersucht. Dazu werden zwei UML (Unified Modelling Language) Metamodellerweiterungenen UML-Q und UML-M eingeführt. Erstere dient als Basis zur Beschreibung von Dienstgütemerkmalen, letztere zur Beschreibung der zur Dienstgüteerfassung notwendigen Instrumentierung. Beide Erweiterungen der UML sind zur Modellierung objektorientierter, verteilter Dienste bestimmt.

Dienstgütebeschreibungen auf Basis von UML-Q werden an die Schnittstellen eines Dienstes assoziiert und „respektieren" damit jegliche Schichtung innerhalb des Dienstes. Die Beschreibungsmöglichkeiten sind nahezu unbegrenzt und durch das Metamodell formal nicht eingeschränkt, so dass es unmöglich erscheint, eine eineindeutige Spezifikation eines Dienstgütemerkmals zu erzielen, so dass ein Dienstgütemerkmal nur durch genau eine Spezifikation beschrieben werden kann. Andererseits bietet das in UML-Q niedergelegte Modellierungskonzept vielfältige Möglichkeiten zur Wiederverwendung bestehender Spezifikationstypen, so dass bei sorgfältiger Verwendung dieser Mechanismen eine in der Praxis eindeutige Spezifikation erzielbar scheint.

Dienstgütemerkmale lassen sich auf Basis von UML-Q nach den in [ODP-QOS] entwickelten Konzepten in ihrer Semantik festlegen. Die gewünschte Dienstgüte wird im vorgeschlagenen Konzept durch zusätzliche, an die Dienstgütemerkmalsspezifikation gebundene constraints (Bediungungen) festgelegt. Damit wird auch konzeptionell eine strikte Trennung zwischen der Spezifikation der Dienstgüte und der Spezifikation von Dienstgütemerkmalen erreicht. Mit der Verwendung von UML als Basis jeglicher Modellierung wird eine, im Sinne von maschinenverarbeitbar nicht formale, Sprache gewählt, so dass eine automatische Weiterverarbeitung getroffener Spezifikationen nicht möglich ist.

Beschreibungen auf Basis von UML-Q und UML-M sind nur in der Design- resp. Verhandlungsphase eines Dienstes verwendbar. Die Verwendung eines komplexen Basismodells in UML verringert die Lesbarkeit der Spezifikation für natürliche Personen erheblich. Die Umsetzung der jeweiligen Spezifikation in eine ablaufende Implementierung lässt sich bestenfalls durch Tools des Software-Engineering unterstützen. Durch den Bezug der Spezifikation auf die Schnittstellen eines Dienstes genügen bei der Festlegung von Dienstgütemerkmalen Informationen über den Dienst aus der Verhandlungsphase.

Ein Leitfaden zur Implementieurng eines Messsystems lässt sich aus der Spezifikation auf Basis von UML-Q nicht ableiten. Allerdings wird mit UML-M die Basis für die Modellierung eines Messsystems gelegt, so dass dieses, wenn auch in Handarbeit, zumindest explizit spezifizierbar ist. Durch die Assoziation der Beschreibung an die Schnittstellen eines Dienstes ist implizit festgelegt, welche Dienstprimitiven für die Bestimmung des jeweiligen Dienstgütemerkmals relevant sind.

Da UML-Q und UML-M die UML erweitern, ist zu erwarten, dass die für die Spezifikation von Dienstgütemerkmalen resultierende Ausdrucksmächtigkeit ausreicht, um alle Dienste zu erfassen, die sich in UML modellieren lassen. Wie Eingangs bereits, erwähnt werden in Metamodellerweiterungen Metaklassen eingeführt, die eine detaillierte Strukturierung, Typisierung und damit einen hohen Grad an Wiederverwendbarkeit zulassen.

3.2.2. QIDL - Dienstgütespezifikation im MAQS Projekt

Im Projekt MAQS (Management for Adaptive QoS-enabled Services) [BeGe 97] wird QIDL [BeGe 99] als Beschreibungssprache für Dienstgüte in verteilten CORBA-basierten Systemen verwendet. Die Spezifikation wird dabei an die Definition eines Interfaces und nicht an einzelne Methoden gebunden. Dieser Ansatz wird gewählt, um die Definitionen auch bei einer Vielzahl von Objekten und Methoden übersichtlich zu halten. Bezogen auf diese gewählte Granularität bleibt eine Definition damit auch eindeutig.

Die Beschreibungssprache QIDL erlaubt es allerdings nicht, die Semantik eines Dienstgütemerkmals zu definieren. Lediglich Attribute mit Typen und Zugriffsmethoden auf die jeweiligen Attribute

können festgelegt werden. Damit wird die Trennung von Dienstgütemerkmal und Dienstgüte als dessen konkrete Ausprägung explizit durch Zugriffsmethoden kenntlich gemacht. Ebenso explizit geschieht die Assoziation einer Spezifikation an ein Interface, wodurch eine optimale Einhaltung der Schichtungsgrenzen gewährleistet ist. QIDL ist in der Syntax formalisiert und wird in MAQS durch einen Compiler verarbeitet. Dabei werden Stubs und Skeletons erzeugt, um die jeweilige Spezifikation umzusetzen. Damit wird ein minimaler Implementierungsleitfaden vorgegeben und die Spezifikation aus der Design- rsp. Verhandlungsphase auch in der Nutzungsphase (zumindest teilweise) umgesetzt.

Da QIDL die Dienstgütespezifikation an Interfacedefinitionen bindet, findet keine Definition der Dienstprimitiven statt, welche die jeweilige Definition beeinflussen könnten. Ein Messsystem, das die jeweils relevanten Eigenschaften eines Interfaces erfasst, wird von MAQS nur durch einen Stub in seiner Schnittstelle spezifiziert. Die jeweilige Umsetzung in einer Implementierung bleibt dem Entwickler überlassen.

Die deklarativen Spezifikationen in QIDL sind übersichtlich gehalten. Gemessen am Anspruch, Dienstgüte für ein Interface festzulegen, ist die Ausdrucksmächtigkeit von QIDL ausreichend. Die explizite Entkopplung der Definition eines Dienstgütemerkmals und des betreffenden Interfaces erleichtert die Wiederverwendung von Definitionen. Allerdings wird kein Strukturierungskonzept für Definitionen im Sinne der Objektorientierung (z.B. Vererbung) vorgegeben.

3.2.3. QDL (Quality Description Languages) in QuO (Quality Objects)

Mit QuO [ZBS 97, LBS+ 98] wurde ein CORBA-basiertes Framework entwickelt, das die Möglichkeit zur Spezifikation, Überwachung und Steuerung der Dienstgüte in verteilten Anwendungen erlaubt. Zur Spezifikation werden dabei die beiden Sprachen CDL (Contract Description Language) und SDL (Structure Description Language) verwendet, die unter dem Oberbegriff QDL (Quality Description Language) zusammengefasst werden. Die folgende Bewertung an Hand des Anforderungskataloges aus Abschnitt 2.4 geschieht für das gesamte Spezifikationskonzept aus dem QuO-Framework und nicht für die jeweiligen Sprachen im einzelnen.

Dienstgütespezifikationen im QuO-Framework orientieren sich strikt am Konzept der verteilten objektorientierten Anwendungen. Damit basiert die Spezifikation zwar direkt auf Methodenaufrufen, allerdings wird durch diese enge Bindung an das zugrundeliegende Verteilungskonzept der Dienstbegriff, an dem sich die Dienstgüte orientiert, stark eingeschränkt. Wie der Name des Frameworks bereits erkennen lässt, lassen sich letztlich nur die Qualitätseigenschaften von Objekten in einer verteilten Umgebung beschreiben.

Die verwendeten Sprachen erlauben in der Design- resp. Verhandlungsphase eine eindeutige Spezifikation, die „handlungsorientiert" ausfällt, also Richtlinien enthält, wie auf Veränderungen bestimmter Eigenschaften des Objektes (resp. des von ihm erbrachten Dienstes) reagiert werden soll. Die Spezifikation der Semantik eines Dienstgütemerkmals im Sinne einer wiederverwendbaren Definition ist durch dieses Vorgehen allerdings nicht möglich. Damit kann die Spezifikation auch die Definition eines Merkmals nicht von seiner aktuellen oder gewünschten Ausprägung trennen. Die Bindung der Spezifikation an ei-

ne Schnittstellenbeschreibung führt automatisch zur Einhaltung von funktionalen Schichtungsgrenzen.

Die verwendeten Sprachen sind syntaktisch formalisiert und maschinenverarbeitbar. Innerhalb des QuO-Frameworks können damit aus der Spezifikation Implementierungs- und Messvorschriften für die der Design- bzw. Verhandlungsphase folgenden Phasen des Dienstlebenszyklus abgeleitet werden. Ein Compiler übersetzt die Spezifikationen und generiert daraus ablauffähigen Code zur Steuerung des QuO-Laufzeitsystems, der die getroffenen Festlegungen in der Nutzungsphase, also auch in ein entsprechendes Messsystem umsetzt, um die jeweils benötigten Eigenschaften eines Objektes zu erfassen. Die Ableitung der Spezifikation eines Messsystems wird, weil in QuO-Laufzeitumgebung integriert, hinfällig. Ähnliches gilt für die Spezifikation der notwendigen Dienstprimitiven: Sie wird implizit durch die Sprachkonstrukte übernommen und durch den Compiler automatisch umgesetzt.

Die verwendeten Sprachen sind deklarativ aufgebaut und erreichen trotz ihrer Mächtigkeit einen hohen Grad an Lesbarkeit, auch wenn die getroffenen Festlegungen, bedingt durch das Konzept der *Quality Objects,* sehr technisch ausfallen. Dieses Konzept schränkt auch die Ausdrucksmächtigkeit der Sprache auf die Beschreibung von Objekteigenschaften ein. Innerhalb des QuO-Frameworks muss diese Ausdrucksmächtigkeit aber dennoch als maximal betrachtet werden, da alle Qualitätsaspekte innerhalb des Frameworks erfasst werden.

Eine Wiederverwendbarkeit von Definition im Sinn der Objektorientierung wird durch die explizite Verbindung einer Spezifikation mit der Festlegung der Schnittstelle eines Objektes, und der damit aufgehobenen Entkopplung der Definiton von Funktionalität und Qualität, praktisch verhindert.

3.2.4. QML - Quality Modeling Language

Die QML (Quality Modeling Language) [FrKo 98] wurde von den HP-Labs entwickelt. Zusammen mit QML wurde ein Laufzeitsystem, QRR (Quality Runtime Representation), vorgeschlagen, das als Basis bei der Umsetzung einer Spezifikation aus der Verhandlungs- resp. Designphase in die Nutzungsphase verwendet werden kann. QML erlaubt die Beschreibung von Dienstgütemerkmalen auf der Anwendungsebene mit Funktionsaspekt.

Die Definitionen von Dienstgütemerkmalen können im QML an Methodenaufrufe gebunden werden. Die Spezifikation selbst geschieht durch die Angabe von Typen und Wertebereichen, wodurch diese zwar eindeutig ist, sich aber keinerlei Semantik im engeren Sinn definieren lässt.

Der Aufbau einer Definition erfolgt strukturiert, trennt aber nicht zwischen der Definition eines Merkmals und der gewünschten Dienstgüte, deren Festlegung bei QML deutlich im Vordergrund steht. Die Bindung der Definitionen an eine Schnittstellendefinition (resp. die darin festgelegten Methoden) ermöglicht die vollständige Einhaltung funktionaler Schichtungsgrenzen. QML ist syntaktisch formalisiert und lässt sich, beispielsweise durch das Laufzeitsystem QRR (Quality Runtime Representation), weiterverarbeiten. Eine Implementierung dieses Systems liegt bisher jedoch nicht vor.

Wird davon ausgegangen, dass QRR entsprechend der Spezifikation implementiert wird, dann lassen sich aus einer Spezifikation in QML aus der Verhandlungsphase auch Ergebnisse für die weiteren Phasen des Dienstlebenszyklus ableiten. QRR stellt die Erbringung einer spezifizierten Dienstgüte in den Vordergrund. Damit liefert sie zwar einen Implementierungsleitfaden, aber keinerlei Messsystem, um qualitätsrelevante Eigenschaften eines Dienstes zu ermitteln, obwohl die zur Ermittlung der

Dienstgüte notwendigen Dienstprimitiven bereits in der Spezifikation explizit genannt werden.

QML verfolgt einen modularen Spezifikationsansatz, der zwar vielfältige Möglichkeiten der Wiederverwenbarkeit bietet, aber gleichzeitig durch die „Verteilung" der Spezifikation auf mehrere Module die Lesbarkeit einschränkt. QML erlaubt eine sehr feingranulare Spezifikation und erreicht durch die Bindung an Methoden eine hohe Ausdrucksmächtigkeit, auch wenn dieser Aspekt in der Definition von QML [FrKo 98] nicht explizit untersucht wird.

Mit CQML [Aage 01] wird ein ähnlicher Ansatz beschrieben, der allerdings eine höhere und detailliertere Ausdrucksmächtigkeit besitzt und zusätzlich die (dynamische) Anpassung und Kombination bestehender Definitionen erleichtert. Mit [RöZs 03] wurden zusätzliche Erweiterungen von CQML und dessen Umsetzung in einem XML-Schema vorgeschlagen. Im Bezug zu QML bieten diese beiden Ansätze aber keine wesentlichen Erweiterungen für die Unterstützung des Dienstlebenszyklus.

3.2.5. WS-QoS - Dienstgüte für Web-Services

Mit WS-QoS wird in [TGN+ 03] ein Ansatz vorgestellt, der es erlaubt, Web-Services um Dienstgütefunktionen zu erweitern. Dabei wird auch eine Beschreibungssprache für Dienstgütemerkmalee auf XML-Basis eingeführt. Die definierbaren Merkmale basieren auf der Datenkommunikationsanalyse und der Beobachtung der Dienstlogik. Spezifikationen werden in einem vorgegebenen „Formular" oder an Hand einer frei definierbaren Ontologie vorgenommen. Die Verwendung einer Ontologie ermöglicht die Benutzung unterschiedlicher Begriffe für den gleichen semantischen Inhalt, schränkt also die Eindeutigkeit einer Spezifikation ein, da diese nur nach Analyse der in der Ontologie abgelegten Begriffsbeziehungen bestimmt werden kann.

Die in [TGN+ 03] vorgeschlagene Spezifikationstechnik trennt die Festlegung der Semantik eines Dienstgütemerkmals deutlich von der Messung der Werte und damit von der Bestimmung / Festlegung der Dienstgüte. Da die Spezifikation ausschließlich auf Merkmalen basiert, die an der Dienstschnittstelle wahrnehmbar sind, respektiert sie inhärent die funktionale Schichtung innerhalb des Dienstes. Durch die Verwendung von XML als Rahmen für die Spezifikationssprache steht eine maschinenverarbeitbare Darstellung der vorgenommenen Definitionen zur Verfügung.

Die in WS-QoS festgelegten Definitionen von Dienstgütemerkmalen lassen sich allein auf Basis von Informationen aus der Verhandlungsphase aufbauen, sind allerdings nur in der Nutzungsphase eines Dienstes verwendbar. Aussagen über das dazu notwendige Messsystem werden in [TGN+ 03] zwar nicht getroffen, allerdings lässt die Diskussion eines Beispiels vermuten, dass ein solches System existiert. Hinweise zur Implementierung eines solchen Systmes lassen sich weder aus einer WS-QoS-Spezifikation ableiten, noch werden sie in [TGN+ 03] angegeben.

Die Verwendung von XML und darin eingebetteter Ontologien verringert die Lesbarkeit der Spezifikation für natürliche Personen erheblich, es sei denn, es werden geeignete Tools zur Verfügung gestellt. Zugleich wird durch dieses Konzept aber eine hohe Ausdrucksmächtigkeit und hochgradige Modularisierung der Spezifikationen erreicht. Damit ist eine Wiederverwendung bestehender Spezifikationen möglich, die durch den auf Web-Services eingeschränkten Anwendungsbereich von WS-QoS zusätzlich unterstützt wird.

3.2.6. ODL - Object Definition Language

ODL wird in der TINA-Architektur [ChMo 95] (siehe auch Abschnitt 3.1.3) verwendet, um Objektmodelle verteilter Systeme zu modellieren. Innerhalb von ODL wird auch die Möglichkeit zur Spezifikation von Dienstgütemerkmalen geboten. Diese ist sowohl für Methoden an der Schnittstelle eines Objekts als auch für Ströme, die an einem Objekt eintreffen oder dieses verlassen, möglich. Damit können nur „äußere" Eigenschaften eines Dienstes/Objekts zur Definition von Dienstgütemerkmalen herangezogen werden, wodurch automatisch Schichtungsgrenzen eingehalten werden.

Im ODL definierenden Standard wird allerdings explizit darauf hingewiesen, dass die Spezifikation von Semantik, also der Bedeutung eines Dienstgütemerkmals, mit ODL nicht möglich ist, sondern implizit durch den Entwickler einer Anwendung erfolgen muss. Damit bietet ODL, obwohl es die eindeutige Definition von Schnittstellen erlaubt, nicht die Möglichkeit, Dienstgütemerkmale eindeutig festzulegen. Dennoch wird eine strikte Trennung von Dienstgüte, die hier strikt als Wertebelegung eines Dienstgütemerkmals festgelegt wird, und der Definition von Dienstgütemerkmalen (durch ihre Typen und Bindungen an ein Methoden- oder Strominterface) vorgenommen.

ODL ist wie IDL durch einen Compiler übersetzbar, wobei sog. Stubs, Codegerüste, erzeugt werden. Eine maschinelle Umsetzung der ODL in ablauffähigen Code ist nicht vorgesehen, da ODL als Spezifikationssprache für die Design- bzw. Verhandlungsphase gesehen wird. ODL bleibt durch die Verwendung einer C-ähnlichen und zugleich deklarativen Syntax auch für natürliche Personen lesbar, besonders wenn Designtools aus dem Software-Engineering verwendet werden.

Entsprechend lassen sich aus einer Dienstgütespezifikation in ODL auch keinerlei Hinweise für die Implementierung eines für die jeweils spezifizierten Dienstgütemerkmale geeigneten Messsystems ableiten. Durch die Bindung der Spezifikation an eine Interface-Definition ist allerdings unmittelbar klar, auf welche Dienstprimitiven sich das jeweilige Dienstgütemerkmal bezieht.

Die Ausdrucksmächtigkit von ODL bezüglich der Definition von Dienstgütemerkmalen wird im Standard nicht untersucht. Nachdem die ODL die Spezifikation von Semantik nicht zulässt, kann aber praktisch jedes Dienstgütemerkmal (zumindest in seinem Typ und seiner Bindung an eine Schnittstelle) festgelegt werden. ODL unterstützt, wie IDL auch, Vererbung. Damit können Definitionen von Dienstgütemerkmalen bei der Vererbung unter Objekten „mitvererbt" und damit wiederverwendet werden. Ein gegliederter, und damit wiederverwendbarer Aufbau von Definitionen, unabhängig von (funktionalen) Objekten, ist aber nicht möglich. Die Wiederverwendbarkeit der Spezifikationen im Sinne der Objektorientierung ist damit nur eingeschränkt möglich.

3.2.7. HQML - Hierarchical Quality of Service Markup Language

HQML (Hierarchical Quality of Service Markup Language) [GWN 01] ist Teil von QoS-Talk, einer grafischen Entwicklungsumgebung, die den Entwurf und die Umsetzung von Dienstgütespezifikationen unterstützt. Ein HQML-Gerüst wird aus der grafischen Spezifikation einer verteilten Anwendung in Form eines hierarchischen Modells automatisch aufgebaut, so dass der Entwickler lediglich Feinarbeiten an diesem Gerüst vornehmen muss. HQML bietet die Möglichkeit, Dienstgüte auf allen Ebenen (Nutzer, Anwedung, Ressourcen) festzulegen.

Die Festlegungen der (gewünschten) Dienstgüte geschehen auf der Nutzerebene durch die Angabe informeller Werte wie „gut", „schlecht" oder „mäßig". Auf der Anwendungsebene wird ein „handlungsorientierter" Ansatz in Form von wenn-dann-Regeln gewählt. Auf der Ebene der Ressourcen werden Attributnamen direkt Werte zugeordnet. Auf allen Ebenen wird die Definition entlang des hierarchischen Dienstmodells gegliedert.

Eine Spezifikation auf Basis der Analyse von Methodenaufrufen, wie in den Anforderungen in Abschnitt 2.4 gefordert, wird somit auf keiner der drei Dienstgüteebenen vorgenommen. Syntaktisch sind die Spezifikationen eindeutig, da aber ihre Semantik, besonders auf der Nutzer- und der Ressourcenebene, nur unzureichend deutlich wird, lässt sich die semantische Eindeutigkeit der Spezifikationen kaum sicherstellen. Zudem findet keine Trennung der aktuellen resp. gewünschten Werte von der Definition eines Dienstgütemerkmals statt. Der Aufbau der Definitionen entlang eines Dienstmodells, das den hierarchischen Aufbau der Dienstfunktionalität aus Komponenten spiegelt, sorgt automatisch für die Einhaltung der funktionalen Schichtung.

Da HQML auf XML aufbaut, ist die Sprache maschinenverarbeitbar. Durch einen Compiler wird dem entsprechend ein Laufzeitsystem aufgebaut, das die in der Verhandlungs- resp. Designphase getroffene Spezifikation in der Nutzungsphase umsetzt. Damit wird automatisch ein Implementierungsleitfaden erzeugt. Nachdem HQML auf die Spezifikation der gewünschten Dienstgüte ausgerichtet ist, steht die Messung der Dienstgüte nicht im Vordergrund. Enstprechend werden weder ein Messsystem spezifiziert, noch die Dienstprimitiven benannt, deren Aufruf Einfluss auf die Ausprägung des jeweils spezifizierten Dienstgütemerkmals hat.

Durch die hierarchische Strukturierung der Spezifikation und die Verwendung von XML bleibt diese auch bei komplexen Systemen lesbar. Im Sinne des (handlungsorientierten) Spezifikationsansatzes ist die gebotene Ausdrucksmächtigkeit sicher ausreichend. Durch die Entwicklungsumgebung QoS-Talk wird zudem ein Spezifikationsrepository angeboten, das die Wiederverwendung bestehender Definitionen erleichtert.

3.2.8. IPPM - IP Performance Metrics

IPPM (IP Performance Metrics) [RFC 2330a] ist von der IETF (Internet Engineering Task Force) [IETF] als [RFC 2330b] standardisiert. Das Framework legt grundsätzliche Vorgehensweisen zur Messung von Performanzdaten fest. Diese Vorgehensweisen können auch als Richtlinien zur Spezifikation von Dienstgütemerkmalen des Dienstes "IP" verstanden werden. Die Spezifikationen richten sich strikt am IP-Protokoll in der Version 4 aus. Unterschiedlichste Dienstgütemerkmale sind in weiteren RFCs [RFC 2678, RFC 2679, RFC 2680, RFC 2681, RFC 3148] standardisiert.

Die Eindeutigkeit einer Spezifikation wird durch ein sehr detailliertes Framework erzielt, das feingranular Prozessschritte bei der Messung bzw. Ermittlung der Werte eines Dienstgütemerkmals vorgibt. Dieses feingliedrige Prozessmuster bildet damit, durch die detaillierte Festlegung der einzelnen Prozesschritte, auch die Basis für die Festlegung der Semantik eines Dienstgütemerkmals. Die Spezifikation des Messprozesses ist unabhängig von tatsächlich gemessenen Werten und erlaubt damit eine optimale Trennung der Beschreibung von Dienstgüte (den aktuellen Werten) und Dienstgütemerkmalen. Da sich Spezifikationen grundsätzlich nur auf IP (Internet Protocol) [RFC 791] beziehen, ist die Einhaltung von funktionalen Schichtgrenzen automatisch gegeben. Die Festlegung des Messprozesses erfolgt ent-

lang des im Framework festgelegten Schemas in natürlicher Sprache und ist damit nicht maschinenverarbeitbar.

Eine Spezifikation nach den Richtlinien von IPPM ist in der Verhandlungsphase durchführbar, allerdings ist ihre Anwendung nur in der Nutzungsphase möglich. Durch die Verwendung natürlicher Sprache bleiben die Spezifikationen für natürliche Personen maximal lesbar. Allerdings liefert das Framework keinerlei Hinweis auf eine Implementierung oder detaillierte Spezifikation eines Messsystems, so dass der Übergang von der Verhandlungsphase in die Nutzungsphase und damit die Bereitstellung eines geeigneten Messsystems "von Hand" in der Bereitstellungsphase erfolgen muss.

IPPM schränkt seinen Anwendungsbereich von vorneherein auf IP ein. In diesem Bereich erreicht das Spezifikationsschema durch seinen Bezug auf die Protokollelemente von IP maximale Ausdrucksfähigkeit. Da Spezifikationen nur für genau einen Dienst (IP) erstellt werden, stellt sich die Frage der Wiederverwendbarkeit von Spezifikationen nicht und wird dem entsprechend im Framework auch nicht behandelt.

3.2.9. XQoS - XML basierte Dienstgütespezifikation

In [EGP+ 03] und [EGP+ 02] wird ein integrierter Ansatz zur Dienstgütespezifikation von verteilten Multimediaanwendungen in allen Dienstgüteebenen vorgestellt, der auf XML basiert. Zur Formalisierung der Spezifikation wird ein Dienstmodell auf Basis von „time stream petri networks (TSPN)" verwendet. In diesem Formalismus kann die zeitliche Korrelation von Datenströmen, wie sie besonders bei Multimediaanwendungen wichtig ist, beschrieben werden. Aufbauend auf diesem Formalismus wird eine XML-Spezifikationssprache aufgebaut.

Die Eindeutigkeit von Spezifikationen kann nur sichergestellt werden, wenn das zugrundeliegende Streammodell eindeutig bestimmbar ist, was in den seltensten Fällen möglich sein wird. Die als XQoS bezeichnete Sprache dient zur Festlegung der gewünschten Dienstgüte in der Verhandlungsphase und bietet weder die Möglichkeit, die Semantik eines Dienstgütemerkmals festzulegen, noch dessen Definition von konkreten Werten zu entkoppeln. Zur Festlegung der Bedeutung eines Merkmals stehen lediglich vordefinierte Begriffe zur Verfügung.

Da in XQoS nur Eigenschaften von Datenströmen betrachtet werden, respektiert eine Spezifikation damit inhärent eine funktionale Schichtung, da diese aus Sicht der Datenströme transparent ist. XQoS baut auf XML auf und ist somit maschineninterpretierbar. Diese Möglichkeit wird allerdings nur am Rande diskutiert.

Die Sprache soll in der Verhandlungsphase Anwendung finden und liefert damit keinerlei Aussagen für die weiteren Phasen des Dienstlebenszyklus. Das der Spezifikation zugrundeliegende Streammodell setzt allerdings Wissen über die (geplante) Implementierung voraus. Durch die Verwendung von XML bleibt die Sprache auch für natürliche Personen lesbar.

Die Ausdrucksmächtigkeit von XQoS ist deutlich eingeschränkt, weil nur vorgefertigte Definitionen verwendet werden können. Die Wiederverwendung von bereits bestehenden Definitionen wird in [EGP+ 03, EGP+ 02] nicht diskutiert und scheint auf Grund der Spezifikation auf Basis eines (implementierungsspezifischen) Streammodells schwierig.

3.2.10. DeSiDeRaTa - Dienstgüte für Echtzeitsysteme

Möglichkeiten, die Dienstgüte für Echtzeitsysteme zu spezifizieren, werden in DeSiDeRaTa [WSRB 98] untersucht. Vorgeschlagen wird eine syntaktisch formalisierte Sprache, welche die Spezifikation von Dienstgütemerkmalen auf Basis von Eigenschaften des Implementierungsablaufs (als Ablaufpfade bezeichnet) erlaubt. Eine eindeutige Spezifikation ist damit nur möglich, wenn eine eindeutige Definition dieser Ablaufpfade existiert. Mögliche, analysierbare Eigenschaften dieser Ablaufpfade sind statisch vorgegeben. Eine Festlegung von Semantik eines Dienstgütemerkmals im engeren Sinne ist damit weder notwendig noch möglich. Ebenso verschmelzen im vorgeschlagenen Spezifikationsmuster Dienstgütemerkmale und Dienstgüte. Es wird nicht davon ausgegangen, dass die beschriebenen Echtzeitanwendungen funktional geschichtet sind. Entsprechend bietet die Sprache auch keinerlei Mechanismen, welche die Einhaltung dieser Schichtungsgrenzen sicherstellen würden.

Zusammen mit der Spezifikationssprache wird in DeSiDeRaTa auch eine Architektur zur Umsetzung der getroffenen Festlegungen vorgestellt. Damit ist es möglich, Dienstgütemerkmale automatisch zu messen, da das entsprechende Messsystem Teil der vorgestellten Architektur ist und damit nicht explizit spezifiziert werden muss. Dennoch reichen die Informationen über einen Dienst in der Verhandlungsphase nicht aus, um eine vollständige Spezifikation eines Dienstgütemerkmals zu erreichen. Implementierungsspezifisches Wissen, wie z.B. der Aufbau möglicher Ablaufpfade ist zwingend notwendiger Teil einer Spezifikation. Dennoch ist die verwendete Beschreibungssprache übersichtlich gehalten, so dass ihre Lesbarkeit auch für natürliche Personen gewährleistet bleibt.

Die Ausdrucksmächtigkeit der in DeSiDeRaTa vorgestellten Spezifikationssprache wird bei der Vorstellung des Ansatzes nicht diskutiert und ist schwer abzuschätzen. Durch die Spezifikation entlang von (implementierungsspezifischen) Ablaufpfaden wird die Wiederverwendbarkeit der Spezifikationen stark eingeschränkt. Eine grundsätzliche Diskussion dieses Aspektes findet in [WSRB 98] nicht statt.

Mit DSpec [DSpec] wird ebenfalls ein Ansatz zur Spezifikation der Dienstgüte in Echtzeitsystemen festgelegt. Auch hier bilden Eigenschaften von Ablaufpfaden die Basis für die Formalisierung der Spezifikation.

3.2.11. G-QoSM - Dienstgüte im Grid

G-QoSM [AARW+ 02] beschreibt eine QoS-Architekturerweiterung für Gridsysteme nach der OGSA (Open Grid Services Architecture) [FKNT 02]. Dabei wird auch eine Möglichkeit vorgestellt, die gewünschte Dienstgüte als Teil eines SLAs in einem XML-Schema zu spezifizieren. Diese Spezifikation bezieht sich zwar auf die Dienstschnittstelle, ist aber entweder so abstrakt, dass sich kein direkter Bezug zu Methoden des Dienstes (im Sinne funktionaler Einheiten) herstellen lässt, oder bezieht sich auf Datenkommunikationseigenschaften.

Durch die teilweise Verwendung eines sehr hohen Abstraktionsgrades wird eine eindeutige Spezifikation verhindert, da sich eine Vielzahl von Interpretationsmöglichkeiten bietet. Die Möglichkeit zur Festlegung der Bedeutung eines Dienstgütemerkmals, seiner Semantik, ist nicht gegeben. Es kann zwar auf standardisierte Begrifflichkeiten zurückgegriffen werden, deren Semantik ist aber nicht definiert. Somit wird praktisch direkt die (erwartete) Dienstgüte festgelegt, eine explizite Festlegung von Dienstgütemerkmalen an sich erfolgt nicht. Dienste werden als monolitische Implementierungen betrachtet, eine eventuelle funktionale Schich-

tung wird nicht betrachtet. Die Verwendung von XML erlaubt die maschinelle Verarbeitung von Spezifikationen. Diese Möglichkeit wird in [AARW+ 02] allerdings nur am Rande erwähnt.

Die Spzifikationen sind in der Verhandlungsphase festlegbar und werden dort verwendet, um unterschiedliche Dienstangebote vergleichen zu können. Ihre Umsetzung in der Nutzungphase und damit auch die Messung der festgelegten Dienstgütemerkmale soll nach Meinung der Autoren durch eine geeignete Middleware erfolgen. Besonders für Dienstgütemerkmale auf der Anwendungsebene ist die Spezifikation, nicht zuletzt durch den verwendeten hohen Abstraktionsgrad, für natürliche Personen gut lesbar. Aussagen über die Ausdrucksmächtigkeit des vorgeschlagenen Spezifikationsansatzes finden sich in [AARW+ 02] nicht. Ebenso bleibt der Aspekt der Wiederverwendbarkeit von Spezifikationen unbetrachtet.

Ein ähnlicher Spezifikationsansatz wie bei G-QoSM wird auch im Projekt Quarz [HHKT 00] für einen allgemeineren Dienstbegriff verfolgt. Im Amaranth Rahmenwerk [HHKT 01] wird ebenfalls ein Spezifikationsansatz der vorgestellten Art für die Festlegung von Dienstgüte und Dienstgütemerkmalen bei hochverfügbaren Anwendungen verwendet.

3.3. Ansätze für beliebige Dienste

Spezifikationsansätze, die auf eine Einschränkung des Dienstbegriffs verzichten, werden im Folgenden dargestellt. Die Ausweitung der Spezifikation auf einen allgemeinen Dienstbegriff führt häufig nur zu einer Teilerfüllung der aufgestellten Anforderungen, wie sich auch in der abschließenden Zusammenfassung in Abschnitt 3.4 zeigt. Die Aufstellung und Umsetzung eines generischen Spezifikationskonzepts stellt im Moment noch eine erhebliche Schwierigkeit dar, da viele vereinfachende Annahmen, wie z.B. das Vorhandensein einer spezifischen Middleware, nicht getroffen werden könnnen.

3.3.1. Qual - a calculation language

In [Dreo 02] und [Dreo 02a] wird eine Sprache zur Beschreibung und Berechnung von Dienstgütemerkmalen vorgeschlagen, die auf Abhängigkeitsgraphen aufbaut, die zeigen, in welcher Weise ein Dienst von welchen Ressourcen abhängt. Entlang dieser Abhängigkeitsgraphen werden Rechenvorschriften aufgebaut, die Messwerte der Komponenten (als QoD Quality of Device bezeichnet) zusammenfassen. Damit lassen sich Dienstgütemerkmale im Bereich der Resourcen und der Anwendungen beschreiben. In den Erläuterungen zu Qual wird der beschriebene Abhängigkeitsgraph als existent vorausgesetzt.

Spezifikationen in Qual basieren letztlich auf den von Ressourcen / Devices gelieferten Parametern. Eine Spezifikation auf Basis der Analyse von Methodenaufrufen, wie in den Anforderungen gefordert, erlaubt Qual nicht. Durch die Ausrichtung der Spezifikation am Abhängigkeitsgraphen bleiben diese eindeutig, solange sie auf den gleichen Graphen Bezug nehmen. Da unterschiedliche Instanzen der selben Dienstklasse (z.B. auf Grund der bei der Instanzierung verfügbaren Ressourcen) unterschiedliche Abhängigkeitsgraphen aufweisen können, ist die Eindeutigkeit einer Spezifikation im Zweifelsfall nur für genau eine Instanz eines Dienstes gegeben, im allgemeinen Fall also praktisch nicht vorhanden.

Die in Qual verwendete Beschreibung auf Basis des Dienstabhängigkeitsgraphen ermöglicht die übersichtliche Darstellung der Semantik eines Dienstgütemerkmals. Diese Beschreibung ermöglicht die Trennung von Messwerten und der Definition eines Dienstgütemerkmals. Allerdings wird davon ausgegangen, dass der Abhängigkeitsgraph nur Entitäten innerhalb einer

organisatorischen Domäne umfasst. Der allgemeinere Fall, dass ein Dienst auch von Subdiensten abhängen kann, die innerhalb weiterer organisatorischer Domänen erbracht werden, wird explizit nicht betrachtet. Somit wird auch nicht erläutert, ob und in welcher Weise eine Spezifikation mit Qual Schichtungsgrenzen im Sinne der Transparenz einhält. Qual gibt keinerlei syntaktische Formalisierung vor, so dass eine Maschinenverarbeitbarkeit nicht gegeben ist.

In Qual festgelegte Spezifikationen können in der Nutzungsphase eines Dienstes verwendet werden, lassen sich aber in der Verhandlungsphase nicht eindeutig festlegen, da der Abhängigkeitsgraph, auf dem die Spezifikation aufbaut, erst nach der vollständigen Instanzierung des Dienstes in der Bereitstellungsphase existiert. Durch die Darstellung in Abhängigkeit des (beliebig komplexen) Abhängigkeitsgraphen ist eine Spezifikation in Qual nur mehr bedingt für natürliche Personen lesbar. Vorgehensweisen zur Erstellung eines Implementierungsleitfadens werden in Qual ebensowenig angegeben wie ein konkretes Mess- bzw. Verarbeitungssystem, das die beschriebene Aggregation von Messwerten übernehmen könnte.

Die Ausdrucksmächtigkeit von Qual wird nicht explizit untersucht. Die Spezifikationssprache scheint aber beliebige Dienstgütemerkmale mit funktionalem Aspekt ausdrücken zu können. Die Problematik der Wiederverwendbarkeit von Spezifikationen wird in Qual nicht betrachtet. Durch die starke Bindung der Spezifikation an eine Dienstinstanz erscheint die Wiederverwendbarkeit bestehender Definitionen schwierig.

3.3.2. QoS Spezifikation im Projekt Aquila

Innerhalb des Projektes Aquila (Adaptive Resource Control for QoS Using an IP-based Layered Architecture) [Aquilla] wird in [Thom 02] auch die Spezifikation der Dienstgüte behandelt. Die Spezifikation der Dienstgüte erfolgt durch XML-Schemata auf Basis von (Dienst-) Methodenaufrufen und Eigenschaften der Kommunikation. Es werden Spezifikationsmuster für alle Dienstgüteebenen angeboten.

Die Eindeutigkeit der Spezifikationen scheint gegeben, wird in [Thom 02] aber nicht explizit untersucht. Eine Festlegung der Semantik eines Dienstgütemerkmals ist nur durch die Verwendung (scheinbar) definierter Begriffe (wie Jitter, Delay etc.) möglich. Die Definition der tatsächlichen Bedeutung eines Dienstgütemerkmals ist damit praktisch unmöglich. Mit den verwendeten XML-Schemata werden Dienstgütemerkmale und die dazu geforderten Werte eng verwoben festgelegt, eine echte Trennung der Beschreibung der Dienstgütemerkmale von der erbrachten resp. gewünschen Dienstgüte ist damit nicht möglich. Die Dienstgüte wird für den Dienst als Ganzes ohne Rücksicht auf eine eventuelle funktionale Schichtung vorgenommen. Durch die Festlegung der Dienstgüte in XML ist grundsätzlich eine Maschineninterpretierbarkeit gegeben, diese Möglichkeit wird in [Thom 02] aber nicht weiter diskutiert.

Die verwendete Dienstgütebeschreibung kommt zwar mit Definitionen der Dienstfunktionalität und -kommunikation aus, wie sie bereits in der Verhandlungsphase eines Dienstes vorliegen, allerdings wird der Übergang von der Spezifikation zur tatsächlichen Messung in der Nutzungphase nicht unterstützt. Die Instrumentierung einer Anwendung muss „von Hand" mit Wissen über die Implementierung geschehen. Die verwendete Dienstgütespezi-

fikation liefert keinerlei Leitlinien für die Implementierung eines Messsystems oder dessen Spezifikation. Durch die Strukturierung der Definitionen mit XML bleiben diese, besonders bei der Anwendung eines geeigneten Anzeigetools, gut lesbar.

Die Ausdrucksmächtigkeit der verwendeten Spezifikation scheint nach den Darstellungen in [Thom 02] ausreichend, wird aber nicht explizit untersucht. Obwohl die Spezifikationstechnik nur exemplarisch angewandt wird und die Festlegung weiterer Dienstgütemerkmale für weitere Dienste als weitere Aufgabenstellung gesehen wird, untersuchen die Autoren die grundsätzliche Problematik der Wiederverwendbarkeit bestehender Definitionen nicht.

3.3.3. Formal Specification of QoS Properties

Die Idee, Dienstgüteparameter an Hand einer formalen Dienstbeschreibung zu spezifizieren, wird in [DoTu 94] für einen Videoübertragungsdienst beispielhaft diskutiert. Eine formale Dienstbeschreibung ermöglicht zwar die eindeutige Definition von Dienstgütemerkmalen, allerdings ist die in [DoTu 94] verwendete Beschreibung mit Mitteln der temporalen Logik implementierungsspezifisch und benötigt ungünstigstenfalls Einblick in Kommunikationsabläufe bei der Diensterbringung, so dass eine eindeutige Definition von Dienstgütemerkmalen für (Klassen von) Dienste(n) praktisch unmöglich erscheint. Dennoch ermöglicht dieses Vorgehen eine detaillierte Festlegung der Semantik eines Parameters. Werte für Dienstgütemerkmale werden erst zur Laufzeit berechnet. Damit sind Dienstgütemerkmale und deren Wertebelegung, die Dienstgüte, deutlich getrennt. Da ein sehr detailliertes, formales Modell der Dienstimplementierung, vielmehr der darin möglichen Abläufe aufgebaut wird, ist es mit dieser Beschreibungstechnik nicht möglich, die Grenzen einer funktionalen Schichtung der Implementierung auch in der Spezifikation einzuhalten. Die verwendete Sprache zur Beschreibung der Dienstimplementierung und der Dienstgütemerkmale ist zwar formal gefasst, Hinweise zu einer möglichen rechnergestützten Verarbeitung werden aber in [DoTu 94] nicht gegeben.

Durch die Spezifikation von Dienstgütemerkmalen entlang eines formalen Modells der Implementierung bzw. einer Instanz der Implementierung ist diese Spezifikation nur innerhalb der Nutzungsphase eines Dienstes verwendbar, da sie erst mit der Existenz einer Implementierung(sinstanz) getroffen werden kann. Das formale Modell der Dienstimplementierung muss in „Handarbeit" mit detaillierter Kenntnis dieser Implementierung entwickelt werden, bevor eine Festlegung von Dienstgütemerkmalen erfolgen kann. Die verwendete Darstellung der temporalen Logik bietet einen mächtigen Beschreibungsmechanismus. Die gewählte Formalisierung ist aber von nicht eingearbeiteten Personen praktisch nicht lesbar, also kaum geeignet, Festlegungen über die Dienstgüte in der Verhandlungsphase zu treffen.

Der in [DoTu 94] beschriebene Ansatz modelliert Dienstgütemerkmale als (zusätzliche) Eigenschaften eines Dienstes, die sich durch Beobachtung der zeitlichen Abläufe bei der Diensterbringung ableiten lassen. Es wird allerdings nicht beschrieben, wie diese Beobachtung technisch realisiert werden kann. Damit fehlt letztlich die Anbindung des formalen Modells an die tatsächlich ablaufende Implementierung, das eigentliche Messsystem.

Die Ausdrucksmächtigkeit des gewählten Ansatzes wird von den Autoren nicht explizit untersucht. Allerdings scheint die gewählte Beschreibungstechnik für die formalen Modelle der Dienstimplementierung in der Lage zu sein, beliebig komplexe Dienste zu modellieren, woraus eine entsprechende Komplexität der Modelle resultieren wird. Da diese Modelle immer

auf Basis genau einer Implementierung erstellt werden, ergeben sich keinerlei Ansatzpunkte zur Wiederverwendbarkeit von Modellteilen und damit zur Wiederverwendung von Spezifikationen von Dienstgütemerkmalen.

Eine ähnlich formaler Ansatz wird auch in [SWM 95] gewählt, um die Dienstgüte von Multimediaanwendungen zu beschreiben. Dienstgüteeigenschaften werden dabei über eine mengentheoretische Formalisierung des jeweiligen Dienstes beschrieben.

3.3.4. QuAL - Quality Assurance Language

QuAL (Quality Assurance Language) [Flor 96] wurde im Rahmen einer Dissertation entwickelt, in der QoSME (QoS Management Environment) eingeführt wird. Spezifikationen in QuAL legen die gewünschte, zu sichernde Dienstgüte (im Sinne von assurance) fest, sind nicht direkt auf die Messung von Dienstgütemerkmalen ausgelegt und basieren auf der Datenkommunikationsanalyse.

Eine Spezifikation auf Basis der Analyse von Methodenaufrufen ist damit in QuAL nicht möglich. Syntaktische Eindeutigkeit wird sichergestellt, allerdings lässt QuAL nur die Benennung von Dienstgütemerkmalen zusammen mit der gewünschten Ausprägung zu, trennt also die Definition der Dienstgüte nicht von der Definition eines Dienstgütemerkmals. Die Festlegung der Semantik eines Dienstgütemerkmals ist nicht möglich. Die Spezifikation richtet sich an Kommunikationsflüssen aus, wird aber durch ein Laufzeitsystem direkt in den ablauffähigen Code einer Anwendung integriert, so dass die Einhaltung von Schichtgrenzen fraglich erscheint.

Die Spezifikationssprache ist maschinentauglich und wird von einem Laufzeitsystem verarbeitet, das Codefragmente zur Sicherstellung der getroffenen Spezifikation erzeugt. Durch die Bindung der Spezifikation an Kommunikationsflüsse lässt sich diese nur durchführen, wenn die Kommunikationsflüsse, resp. ihr potentielles Auftreten, bekannt sind, was frühestens zu Beginn der Nutzungsphase der Fall ist, wenn eine konkrete Implementierung existiert. QuAL kann damit nicht in der Verhandlungsphase angewandt werden. Die ableitbaren Ergebnisse sind wiederum nur in der Nutzungsphase verwendbar.

Durch den Ansatz von QuAL, die Sicherstellung von Qualitätseigenschaften durch das direkte Einfügen von Steuerungscode in die Dienstlogik zu erreichen, steht die Messung der (aktuellen) Dienstgüte im Hintergrund. Somit wird keinerei Messsystem spezifiziert oder ein Leitfaden zur Implementierung eines solchen angegeben. Ebenso werden die Dienstprimitiven, deren Aufruf Einfluss auf das jeweilige Dienstgütemerkmal hat, nicht benannt.

Spezifikationen in QuAL sind deklarativ und auch für natürliche Personen gut lesbar, auch wenn sie durch die Bindung an eine Implementierung eher technisch ausfallen. Gemessen am Spezifikationsansatz erreicht QuAL maximale Ausdrucksmächtigkeit. Eine Untersuchung der möglichen Wiederverwendbarkeit von QuAL-Definitionen wird allerdings in [Flor 96] nicht vorgenommen.

3.4. Zusammenfassung und Bewertung

Zur zusammenfassenden Bewertung der in Abschnitt 3.2 und 3.3 vorgestellten Spezifikationsansätze wird die tabellarische Darstellung der Anforderungserfüllung verwendet, wie sie Abbildung 3.1 zeigt.

Die Tabelle zeigt in der x-Achse die vorgestellten Ansätze und in der y-Achse die geprüften Kriterien. Weiterhin ist für jeden Spezifikationsansatz eine Gesamtbewertung im Sinne eines Erfüllungsgrades aufgeführt. Die vollständige Erfüllung aller Anforderungen würde entsprechend zu ei-

✓ erfüllt

☑ teilweise erfüllt

❍ keine Aussage

✗ nicht erfüllt

〰 Kriterium nicht anwendbar

💻 durch Architektur / Implementierung

	UML-Q & UML-M	QIDL	QDL (Quality Description Languages)	QML (QoS Modeling Language)	WS-QoS (Webservice QoS)	ODL (Object Definition Language)	HQML in QoSTalk	IPPM (Internet Provider Performance Metrcis)	XQoS	DeSiDeRaTa	QAUL (a calculation language)	Aquilla Specification	Formal Specification of QoS Properties	QuAL (Quality Assurance Language)	G-QoSM (Grid QoS Management)
Anwendungsbereich D=allg. Dienste, d=spez. Dienste	d	d	d	d	d	d	d	d	d	d	D	D	D	D	d
Spezifikationsreferenzpunkte **SAP, RES**ources, **COM**unicatin, **Co**de / **Lo**gic-injection	SAP	(SAP)	SAP	SAP	COLO, COM	SAP		COM	COM	COLO	RES	COM, SAP	COLO	COM	SAP, COM
Dienstgütequader (Ebenen Aspekte Phasen)	A f I	A f I II III	A f I II II	A f I II III	A f III	A f I	NAR f I II III	A f III	NAR f I	AR f III	AR f III	NAR f III	A f III	A f III	AR f III
Erfüllgsgrad	79%	75%	75%	70%	69%	63%	57%	50%	39%	38%	35%	34%	29%	27%	21%
30% Spezifikation auf Basis von Methodenaufrufen an der Dienstschnittstelle	✓	✗	✓	✓	✗	✓	✗	✗	✗	✗	✗	☑	✗	✗	✗
40% eindeutige Spezifikation	☑	✓	✓	✓	☑	✗	✗	✓	✗	✗	✗	☑	☑	✗	✗
30% Spezifikation der Semantik möglich	✓	✗	✗	✗	✓	✗	✗	✓	✗	✗	✓	✗	☑	✗	✗
47% Spezifikation trennt Dienstgütemerkmal von Dienstgüte	✓	✓	✗	✗	✓	✓	✗	✓	✗	✗	✓	✗	✓	✗	✗
64% Einhaltung der Schichtgrenzen	✓	✓	✓	✓	✓	✓	✓	✓	✓	〰	❍	✗	✗	✗	✗
60% maschinentaugliche (formale Sprache)	✗	✓	✓	✓	✓	✓	✓	✗	✓	✓	✗	❍	❍	✓	❍
33% liefert Ergebnisse für den gesamten Lebenszyklus	✗	✓ 💻	✓	✓ 💻	✗	✗	✓	✗	✗	✓ 💻	✗	✗	✗	✗	✗
70% benötigt nur Information aus der Verhandlungsphase	✓	✓	✓	✓	✓	✓	✓	✓	☑	✗	✗	✓	✗	✗	✓
80% maximale Lesbarkeit in der Verhandlungsphase für natürliche Personen	☑	✓	✓	☑	☑	✓ 💻	✓ 💻	✓	✓	✓	☑	✓ 💻	✗	✓	✓
29% Ableitung eines Implementierungs- leitfadens	☑	☑ 💻	✓	✓	✗	✗	✓ 💻	✗	〰	✗	✗	✗	✗	✗	✗
33% explizite Spezifikation der notwendigen Dienstprimitiven	✓	✗	〰	✗	〰	✓	✗	〰	〰	〰	〰	✗	〰	〰	〰
15% Spezifikation eines Messsystems	☑	☑ 💻	〰	✗	❍	✗	✗	✗	〰	✓ 💻	✗	✗	✗	✗	✗
68% Ausdrucksmächtigkeit deckt alle denkbaren anwendugsorientierten Dienstgütemerkmale ab	✓	✓	✓	✓	✓	✓	✓	✗	✗	❍	☑	☑	☑	✓	〰
39% Wiederverwendbarkeit bestehender (Teile) einer Spezifikation	✓	✓	✗	✓	✓	☑	✓ 💻	〰	❍	✗	❍	✗	✗	❍	✗

Abbildung 3.1.: tabellarische Darstellung der Anforderungserfüllung durch die vorgestellten Spezifikationsansätze

ner Bewertung mit 100% führen. Zu jedem Kriterium ist zudem dessen Erfüllungsgrad bezogen auf alle betrachteten Ansätze angegeben.

Auffallend ist, dass Ansätze, die auf einem spezifischen, eingeschränkten Dienstbegriff aufbauen, deutlich besser abschneiden als solche, die sich auf einen generischen Dienstbegriff beziehen. Weiterhin bezeichnend ist, dass der Ansatz mit der besten Wertung (UML-Q, UML-M siehe Abschnitt 3.2.1) mit seiner Beschreibungstechnik mit UML auf einen anerkannt generischen Standard zur Systembeschreibung aufsetzt.

Ebenso fällt auf, dass keiner der untersuchten Ansätze die Anforderungen vollständig erfüllt. Eine Analyse des Erfüllungsgrades der Anforderungen über alle vorgestellten Ansätze hinweg, wie ihn die Tabellenspalte **vor** den Kriterien durch Angabe von %-Werten darstellt, zeigt, dass besonders die Einbindung der Spezifikationstechniken in den Dienstlebenszyklus zu Wünschen übrig lässt. Die wenigsten Techniken bieten hier Konzepte. Ebenso wird die Notwendigkeit, die Implementierung eines Messsystems zu unterstützen, von den wenigsten Ansätzen gesehen oder gar unterstützt.

Ein neu definierter Spezifikationsansatz muss also von vorneherein mit Blick auf die Integration in den Dienstlebenszyklus entwickelt werden, um diese Defizite schon durch das Design auszugleichen. Bei hochgradig erfüllten Anforderungen lassen sich Konzepte aus den bestehenden Ansätzen durchaus wiederverwenden.

Im Bereich der kaum erfüllten Anforderungen liefern bestehende Ansätze, die das jeweilige Kriterium dennoch erfüllen, wertvolle Ansätze für die eigene Entwicklung. Auffallend ist beispielsweise, dass alle Ansätze, die auf einer generischen Systemmodellierung aufbauen, die Spezifikation von Dienstgütemerkmalen an der Dienstschnittstelle durchführen. Es bietet sich also an, einen Spezifikationsansatz auf eine solche Systemmodellierung aufzubauen.

Die Spezifikation eines Messsystems wird, wenn überhaupt, nur von Ansätzen durchgeführt, die einen auf eine bestimmte Technologie eingeschränkten Dienstbegriff verwenden. Die Abstraktion von Technologien bei gleichzeitiger Modellierung eines ablauffähigen Messsystems wird demnach eine weitere Aufgabe bei der Entwicklung eines Spezifikationskonzepts sein.

Die expilizite Benennung von in der Spezifikation verwendeten Dienstprimitiven wird nur von Ansätzen durchgeführt, die auf einer objektorientierten Modellierung aufsetzen. Diese Art der Modellierung scheint damit geeignet, eine generische Spezifikation von Dienstgütemerkmalen zu unterstützen.

4. Von der formalen Dienstgütespezifikation zur rechnergestützen Umsetzung

In diesem Kapitel wird eine Lösung für die Problematik der automatischen Umsetzung einer Dienstgütespezifikation aus der Verhandlungsphase in den restlichen Phasen des Dienstlebenszyklus vorgestellt. Dazu wird zunächst in Abschnitt 4.1 die Lösungsidee skizziert, die dann in den Abschnitten 4.2 und 4.3 detailliert entwickelt und beschrieben wird. Die Präsentation des Lösungansatzes wird durch die Einführung einer Spezifikationssprache für Dienstgütemerkmale in Abschnitt 4.3.3.4 abgeschlossen. Eine Bewertung der vorgestellten Lösung an Hand der in Abschnitt 2.4 aufgestellten Anforderungen schließt das Kapitel ab.

4.1. Lösungsidee

In der Problemstellung in Abschnitt 2.2 wird gefordert, dass aus einer einzigen, in der Verhandlungsphase eines Dienstes festgelegten Spezifikation der Dienstgütemerkmale, alle weiteren, zur Umsetzung dieser Spezifikation notwendigen Schritte in den Folgephasen des Dienstlebenszyklus, ableitbar sind. Komplexestes Teilergebnis ist dabei der Aufbau eines Messsystems für die Nutzungsphase.

Es bietet sich damit an, die Spezifikation an eben diesem Messsystem auszurichten, so dass dessen Ableitung per Konstruktion immer gelingt. Zweckmäßigerweise wird die Spezifikation aber nicht an einem bestimmten Messsystem ausgerichtet, sondern basiert auf einem generischen, parametrisierbaren Messprozess, den es auf Basis eines ebenfalls generischen Dienstmodells zu entwerfen gilt. Dieser allgemeine Messprozess wird dann durch die Spezifikation parametrisiert, so dass für die Erfassung des jeweils beschriebenen Dienstgütemerkmals der notwendige und anwendbare Messprozess entsteht.

Diese Parametrisierung wird wiederum automatisiert und ist damit rechnergestützt durchführbar. Entsprechend lassen sich die restlichen phasenspezifischen Teilergebnisse aus der Spezifikation ableiten bzw. rechnergestützt generieren:

- in der Verhandlungsphase

  Die Spezifikation selbst ist per Konstruktion so angelegt, dass sie als Definition eines Dienstgütemerkmals angesehen werden soll. Damit besteht keinerlei Bedarf, in dieser Phase weitere Informationen zu generieren.

- in der Bereitstellungsphase

  Die Implementierung des Dienstes muss in dieser Phase um die Implementierung eines für die spezifizierten Dienstgütemerkmale spezifischen Messprozesses ergänzt werden, d.h. der generische Messprozess wird in dieser Phase in seiner spezifischen Parametrisierung generiert.

  Zusätzlich müssen in dieser Phase Schnittstellen benannt werden, an denen das Messsystem während der Messung in der Nutzungsphase in die Dienstimplementierung eingreift, damit diese bei der Implementierung des Dienstes erstellt werden können. Die Spezifikation dieser Schnittstellen wird dementsprechend in dieser Phase aus der Spezifikation der Dienstgütemerkmale generiert. Außerdem wird eine Anbindung des Messsystems an die Dienstimplementierung nach aus der Spezifikation abgeleiteten Leitfäden erstellt.

- in der Nutzungsphase

  Der für die Bereitstellungsphase bereitgestellte, parametrisierte Messprozess wird in dieser Phase lediglich (zusammen mit der eigentlichen Dienstimplementierung) gestartet. Die in der Bereitstellungsphase erstellte Anbindung des Messsystems wird in dieser Phase ebenfalls gestartet. Per Konstruktion steht damit ein ablauffähiges Messsystem zur Verfügung.

Die für die dargestellten Generierungsschritte notwendige Spezifikation wird derart formalisiert, dass sie automatisch verarbeitbar, aber zugleich für natürliche Personen gut lesbar bleibt. Es entsteht eine deklarative, formale Sprache (im Sinne einer Programmiersprache). Damit die Generierung der phasenspezifischen Teilergebnisse möglichst einfach ablaufen kann, wird diese Sprache so weit wie möglich an die Generierungsschritte angepasst.

4.2. Generischer, parametrisierbarer Messprozess

In diesem Abschnitt wird das Modell eines generischen und parametrisierbaren Messprozesses nach den Vorgaben der Lösungsidee aus Abschnitt 4.1 entwickelt. Dazu wird der Messprozess in Abschnitt 4.2.1 zunächst beispielhaft an Hand von durch ihn umsetzbaren Dienstgütemerkmalsdefinitionen vorgestellt. Danach wird in den Abschnitten 4.2.2 und 4.2.3 ein formales Modell des Messprozesses erstellt. Vorgehensweisen bei der Parametrisierung des Messprozesses, also der Anpassung des generischen Modells an eine ausgewählte Spezifikation, werden schließlich in Abschnitt 4.2.4 beschrieben.

4.2.1. Beispielhafte Definition

Der vorgeschlagene Messprozess setzt einerseits die Anforderung nach einer Spezifikation auf der Basis der Analyse von Methodenaufrufen direkt um, in dem er am SAP (Service-Access-Point) eines Dienstes ansetzt. Andererseits erweitert er mit IPPM (IP Performance Metrics) [RFC 2330a] einen generischen Ansatz aus dem Internet-Umfeld.

4.2.1.1. Prozessmuster

Der Messprozess läuft in folgenden Schritten ab:

- Methodenaufruf am SAP abfangen

  Der Messprozess wird durch den Aufruf einer Methode am SAP des Dienstes ausgelöst. Eine Methode ist dabei im abstrakten, objektorientierten Sinn zu verstehen und repräsentiert Teile der Dienstfunktionalität.

- Dienstgüteereignisse durch Filterung ermitteln

  Wird eine bestimmte Methode mit bestimmten Parametern aufgerufen, so wird ein sog. Dienstgüteereignis erzeugt. Die Kriterien für die Filterung sind frei bestimmbar und damit Parameter des Messprozesses, die durch die Dienstgütespezifikation festgelegt werden müssen.

- Dienstgüteereignisse korrelieren

  Mehrere Dienstgüteereignisse werden nach einer frei definierbaren Vorschrift korreliert, d.h. Dientsgüteereignisse, die zusammen genommen einen Rückschluss auf eine die Dienstgüte bestimmende Eigenschaft des Dienstes erlauben, werden zueinander in Beziehung gesetzt.

- Messwert ermitteln

  Aus den Eigenschaften der korrelierten Dienstgüteereignisse wird ein Messwert ermittelt. (z.B. die Anzahl der Ereignisse, ihr maximaler zeitlicher Abstand). Die Berechnungsvorschrift ist wiederum Parameter des Messprozesses und muss durch die Dienstgütespezifikation vorgegeben werden.

- Messwerte statistisch aggregieren

  Ermittelte Messwerte werden statistisch aggregiert. Die dazu verwendete Funktion ist ebenfalls ein Parameter des Messprozesses.

Das Verfahren, relevante Ereignisse zu korrelieren, daraus einen Messwert zu ermitteln und diesen wiederum statistisch zu aggregieren, ist aus dem IPPM-Framework übernommen. Dieses Framework bietet einen generischen Ansatz zur Spezifikation von Dienstgütemerkmalen im Internet-Umfeld.

Dienstgüterelevante Ereignisse werden in IPPM jedoch von Ereignissen am Protokollschnitt abgeleitet. Nach dem in Abschnitt 2.1.7 vorgestellten Klassifizierungsschema wird damit von IPPM der Ansatz der Datenkommunikationsanalyse verfolgt. Der hier vorgestellte Prozess hingegen verfolgt das Konzept der Analyse von Methodenaufrufen, wie in den Anforderungen an eine mögliche Lösung gefordert.

4.2.1.2. Umsetzungsbeispiele

Die Tragfähigkeit des vorgestellten Messprozesses verdeutlichen die im folgenden dargestellten, informellen Spezifikationsbeispiele. Mögliche Parametrisierungen des Messprozesses werden an Hand typischer Dienstgütemerkmale demonstriert. Dabei wird aufgelistet, welche Parameter des Messprozesses in welcher Weise festgelegt werden müssen.

4.2.1.2.1. Framefehlerrate bei Ethernet

- Gängige Definition

  Fehlerhafte Frames lassen sich bei Ethernet an Hand des im Frame enthaltenen CRCs feststellen. Die Framefehlerrate stellt damit das Verhältnis zwischen fehlerhaft übertragenen Rahmen und korrekt übertragenen Rahmen dar.

- relevante Methoden am SAP des Dienstes

  Der Dienst „Ethernet" stellt, abstrakt gesprochen, die beiden Methoden *sende* und *empfange* bereit, die als Parameter die Ziel- resp. Quelladresse und die zu übertragenden Daten aufweisen. Für die Spezifikation ist die Methode *empfange* relevant.

- Dienstgüteereignisse

  Bereits aus der Definition in Prosa ist erkennbar, dass zwei Klassen von Dienstgüteereignissen betrachtet werden: Der korrekte Empfang eines Frames und der fehlerhafte (durch den CRC feststellbare) Empfang eines Frames. Beide Dienstgüteereignisse lassen sich damit durch Filterung von Aufrufen der Methode *empfange* bestimmen.

- Korrelation

  Die Korrelation der beiden genannten Dienstgüteereignisse besteht darin, jeweils die Ereignisse einer Klasse (seit Beginn einer Messung) zu zählen und das Verhältnis aus beiden Werten zu bilden. Eine komplexere Korrelation könnte zusätzlich die verstrichene Zeit berücksichtigen und in festen Intervallen die Zählung der Ereignisse von 0 beginnen, so dass Ausgleichseffekte bei Langzeitmessungen vermieden werden.

- Statistik

  Eine statistische Aufbereitung der durch die Korrelation der Dienstgüteereignisse ermittelten Werte wird in den gängigen Definitionen nicht betrachtet. Dennoch ist eine solche Nachverarbeitung möglich. Beispielsweise könnte ein gleitender Durchschnitt oder die relative Häufigkeit der Fehlerrate (dargestellt durch eine Verteilungsfunktion) berechnet werden.

4.2.1.2.2. Antwortzeit Für die hier vorgestellte Definition des Dienstgütemerkmals „Antwortzeit" wird auf das in Abschnitt 1.1 eingeführte Szenario zurückgegriffen. Als Beispiel soll hier die Antwortzeit für eine mit dem Fahrzeugkonfigurationsdienst durchgeführte Änderung an der Ausstattung eines Fahrzeugs verwendet werden.

- Gängige Definition

  Dieses Dienstgütemerkmal wird im weitesten Sinne zur Darstellung der Qualität eines entfernt erbrachten Dienstes verwendet. Betrachtet wird dabei die Zeit, die bis zur Darstellung eines Ergebnisses eines Verarbeitungsvorgangs vergeht, nach dem dieser durch den Benutzer ausgelöst wurde. Im Fall des Fahrzeugkonfigurators etwa die Zeit, die vergeht, bis eine vom Benutzer geänderte Ausstattung (z.B. Sitzbezüge) in der grafischen Darstellung des Fahrzeugs erscheint.

- relevante Methoden am SAP des Dienstes

  Für den dargestellten Beispieldienst kann davon ausgegangen werden, dass u.a. eine Methode *verändereAusstattung* Bestandteil des Dienst-SAPs ist. Als Parameter erwartet diese Methode das Ausstattungsmerkmal (z.B. Sitzbezüge, Lackierung), das verändert werden soll, sowie die neue Ausprägung dieses Merkmals.

- Dienstgüteereignisse

  Zur Bestimmung der Antwortzeit genügt es, aus einem Aufruf der Methode *verändereAusstattung* zwei Dienstgüteereignisse abzuleiten. Eines wird zu Beginn des Aufrufs generiert, ein weiteres, wenn dieser Aufruf zurückkehrt.[1]

- Korrelation

  Zur Ermittlung eines Messwertes ist lediglich die Zeitdifferenz zwischen zwei korrespondierenden Ereignissen (Beginn und Ende des Aufrufs) zu berechnen. Dabei wird davon ausgegangen, dass ein User immer nur auf die Antwort genau einer Anforderung wartet. Soll diese Randbedingung aufgehoben werden, müssen zur Korrelation der Ereignisse zusätzliche Identifikatoren verwendet werden, die einen Rückschluss auf den jeweiligen "Bearbeitungs-Thread" zulassen.

- Statistik

  Eine statistische Nachverarbeitung analog zum obigen Beispiel ist möglich, in den gängigen Spezifikation des Dienstgütemerkmals „Antwortzeit" allerdings nicht explizit beschrieben.

4.2.1.2.3. Verfügbarkeit Die Spezifikation des Dienstgütemerkmals „Verfügbarkeit" wird ebenfalls in Anlehnung an das in Abschnitt 1.1 eingeführte Szenario durchgeführt.

[1] Dabei wird davon ausgegangen, dass der Methodenaufruf synchron erfolgt. Bei einem asynchronen Aufruf muss das zweite Dienstgüte ereignis beim Aufruf der Rückgabemethode erzeugt werden.

- Gängige Definition

  Als Definition für Verfügbarkeit wird hier das Verhältnis von beantworteten Anfragen zu allen getätigten Anfragen verwendet.

- relevante Methoden am SAP des Dienstes

  Zur Messung dieses Dienstgütemerkmals müssen alle Anfrage- und Rückgabemethoden des Dienstes betrachtet werden.

- Dienstgüteereignisse

  Jeder Aufruf einer Anfragemethode löst ein Anfrageereignis aus, jeder Aufruf einer Rückgabemethode ein Rückgabeereignis.

- Korrelation

  Die Korrelation ist in diesem Fall trivial, es müssen lediglich die Anzahlen der jeweiligen Dienstgüteereignisse in einem Quotienten zueinander in Beziehung gesetzt werden.

- Statistik

  Eine statistische Nachverarbeitung ist notwendig, um praktisch sinnvolle Ergebnisse zu erzielen. Nach der vorgenommenen Spezifikation würde die Verfügbarkeit schlagartig sinken, wenn punktuell eine große Anzahl von Anfragen an den Dienst gestartet wird. Es liegt also nahe, die Messreihe, z.B. durch einen gleitenden Durchschnitt, zu glätten.

Die exemplarische Umsetzung unterschiedlichster Dienstgütemerkmale mit dem vorgeschlagenen Messprozess hat dessen Tauglichkeit als generische Basis für die Messung von Dienstgütemerkmalen gezeigt. In den folgenden Abschnitten wird der eingeführte Prozess formal modelliert und beschrieben.

4.2.2. UML-Aktivitätsdiagramm

Zur Darstellung eines Prozesses im weitesten Sinne ist ein UML-Aktivitätsdiagramm [RJB 98] ideal geeignet. Einzelne Verarbeitungsschritte eines Prozesses werden dort als Aktivitäten dargestellt. Aus der informellen Einführung des generischen Messprozesses in Abschnitt 4.2.1.1 ergibt sich kanonisch eine erste Definition dieses Prozesses in einem Aktivitätsdiagramm, das die entsprechenden Aktivitäten *SAP-Aufruf abhören*, *Dienstgüteereignis generieren*, *Ereignisse korrelieren* und *statistisch nachverarbeiten* enthält. Abbildung 4.1 zeigt dieses Diagramm. Dabei wird auch dargestellt, dass mehrere Methodenaufrufe abgehört werden und daraus Dienstgüteereignisse generiert werden. Diese werden zentral korreliert und statistisch verarbeitet.

UML-Aktivitätsdiagramme bieten auch die Möglichkeit, den Austausch von Objekten zwischen Aktivitäten explizit darzustellen. Damit lässt sich der vorgestellte Prozess weiter verfeinern. Zugleich wird an dieser Stelle der Prozess erweitert.

Die Aktivität *SAP-Aufruf abhören* liefert als Ergebnis die aufgerufene Dienstprimitive, kurz *Primitive*, im Sinne des OSI-Schichtenmodells [ISO 7498]. Aus der Aktivität *Dienstgüteereignis generieren* resultiert ein allgemeines *Dienstgüteereignis*-Objekt. Die Ausführung der Aktivität *Ereignisse korrelieren* liefert einen *Messwert*, d.h. genau einen Wert, der durch die vorausgegangene Ereigniskorrelation ermittelt wurde. Dieser dient als Eingabe für die Aktivität *statistisch nachverarbeiten*, die als Ausgabe die aktuelle, *gemessene Dienstgüte* liefert. Somit ergibt sich ein verfeinertes und erweitertes Aktivitätsdiagramm, wie es in Abbildung 4.2 dargestellt ist.

Die in dieser Abbildung verwendete Darstellung zeigt allerdings nicht den Austausch von Objekten, sondern gibt direkt die Klassen an, denen die jeweiligen Aus-

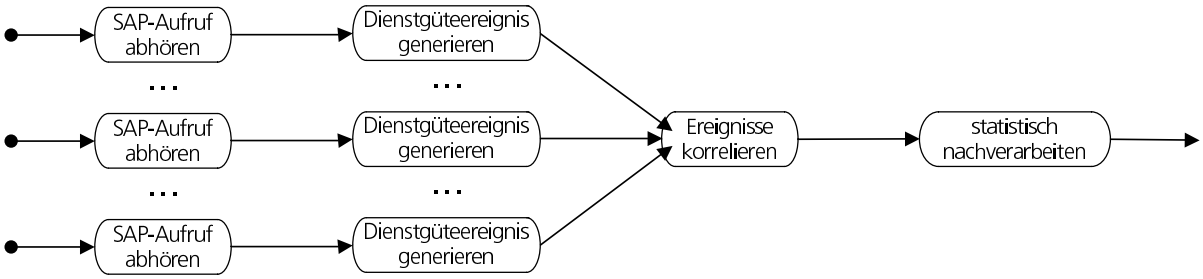

Abbildung 4.1.: prinzipielle Darstellung des Messprozesses im UML-Aktivitätsdiagramm

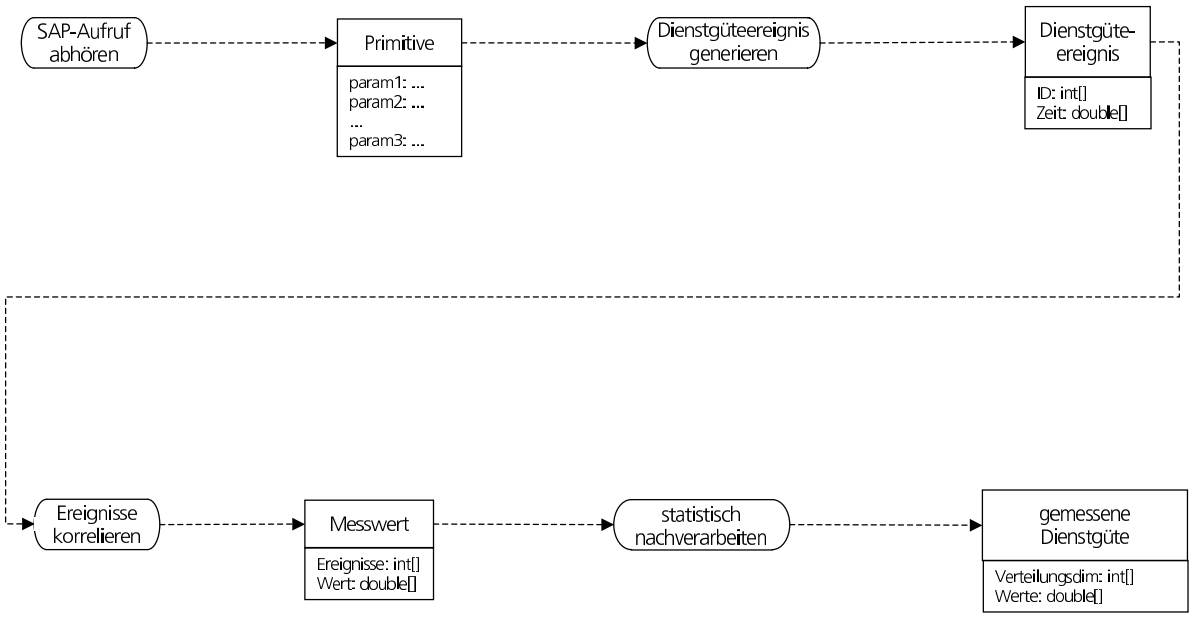

Abbildung 4.2.: verfeinerte Darstellung des Messprozesses im UML-Aktivitätsdiagramm

tauschobjekte entsprechen. Damit kann auf die Darstellung jeder einzelnen Austauschbeziehung, wie in Abbildung 4.1 vorgenommen, verzichtet werden. Zusätzlich sind für die zwischen den Aktivitäten ausgetauschten Klassen die jeweiligen Attribute angegeben. Die neu eingeführten Klassen werden im Folgenden beschrieben.

- Primitive

  Diese Klasse bildet mögliche Primitiven eines Dienstes ab. Kanonischer Weise sind alle Parameter einer solchen Dienstprimitive durch entsprechende Attribute abzubilden. Eine Primitve stellt dabei die minimale, aufrufbare Einheit der Dienstfunktionalität dar. Sie ist in ihrer Spezifikation unabhängig von den technischen Aufrufmöglichkeiten am SAP des Dienstes. Unterschiedliche SAP-Implementierungen können also Zugriff auf ein und die selbe Dienstprimitive schaffen.

  Das hier gezeigte Modell dient zur Vermittlung eines grundsätzlichen Verständnisses des Messprozesses. Detaillierte Überlegungen zur Festlegung und Modellierung der Aufrufparameter werden im folgenen Abschnitt 4.2.4 untersucht, wenn Vorgehensweisen für die Beschreibung von Dienstgütemerkmalen mit dem entwickelten Modell diskutiert werden.

- Dienstgüteereignis

  Ein vereinheitlichter Messprozess erfordert auch (soweit irgend mög-

lich) die Vereinheitlichung des Datenaustauschs zwischen den jeweiligen Teilaktivitäten. Ein Dientstgüteereignis wird entsprechend durch einen minimalen Satz an Attributen beschrieben: Ein Identifikator zur späteren Verwendung im Korrelationsprozess und ein Zeitstempel, um Zeitdauern etc. auch offline berechnen zu können. Im folgenden Abschnitt 4.2.4 wird gezeigt, dass diese Attribute zur Beschreibung von Ereignissen im standardisierten Messprozess ausreichend sind.

- Messwert

  Bezeichnet den Wert, der durch die Korrelation von Dienstgüteereignissen entsteht. Dieser ist an sich für die statistische Weiterverarbeitung ausreichend und durch ein entsprechendes Attribut *Wert* in der Klasse *Messwert* abgebildet. Zusätzlich enthält die Klasse das Attribut *Ereignisse*. Dieses gibt die Anzahl der zur Wertermittlung verwendeten Ereignisse wieder, so dass in jedem Fall eine sinnvolle statistische Aggregation der ermittelten Messwerte im Bezug zu den eingegangenen Ereignissen möglich ist.

- gemessene Dienstgüte

  Nach dem in 4.2.1.1 eingeführten Messprozess wird ein Dienstgütemerkmal immer in Form einer (numerischen) Verteilung angegeben. Entsprechende Attribute finden sich in der Klasse *gemessene Dienstgüte* wieder. Die Stützwerte der ermittelten Verteilung werden im Feld *Werte* abgelegt. Das Attribut *Verteilungsdim* gibt Auskunft über die Dimensionen der Verteilung.

4.2.3. UML-Klassendiagramm und Einbindung des Modells in das MNM-Dienstmodell

Die bisherige Modellentwicklung hat die zwischen den Aktivitäten ausgetauschten Objekte und die Aktivitäten selbst betrachtet. In diesem Abschnitt werden den einzelnen Aktivitäten implementierende Klassen zugeordnet, um abschließend ein UML-Klassendiagramm zu entwickeln. Dabei wird der Ansatz verfolgt, jede Aktivität durch genau eine Klasse zu „implementieren". Diese „Implementierungsklassen" und ihre zugehörigen Methoden werden im Folgenden beschrieben und in Abbildung 4.3 zusammen mit dem bereits bestehenden Aktivitätsdiagramm dargestellt.

- SAP-Aufruf abhören

  Der Aufruf einer Dienstprimitive geschieht nach den Definitionen von OSI [ISO 7498] an einem SAP, dem Dienstzugangspunkt. Allerdings ist dieser Aufruf direkt nicht beobachtbar. Vielmehr muss der SAP entsprechend instrumentiert sein, so dass der Aufruf einer Primitive „von außen" erkannt werden kann. Dem entsprechend wird die Aktivität *SAP-Aufruf abhören* durch die Klasse *SAP-Ankopplung* umgesetzt. Diese Klasse realisiert eine Instrumentierung des SAP und stellt Instanzen der Klasse *Primitive* zur Verfügung, welche die jeweils aufgerufene Primitive beschreiben.

  Die beschriebene Klasse ist von der jeweils verwendeten Implementierung des SAPs sowie den definierten Dienstprimitiven abhängig. Verfahren für die Bewältigung dieser Abhängigkeiten bei der Modellierung werden im folgenden Abschnitt 4.2.4 erläutert.

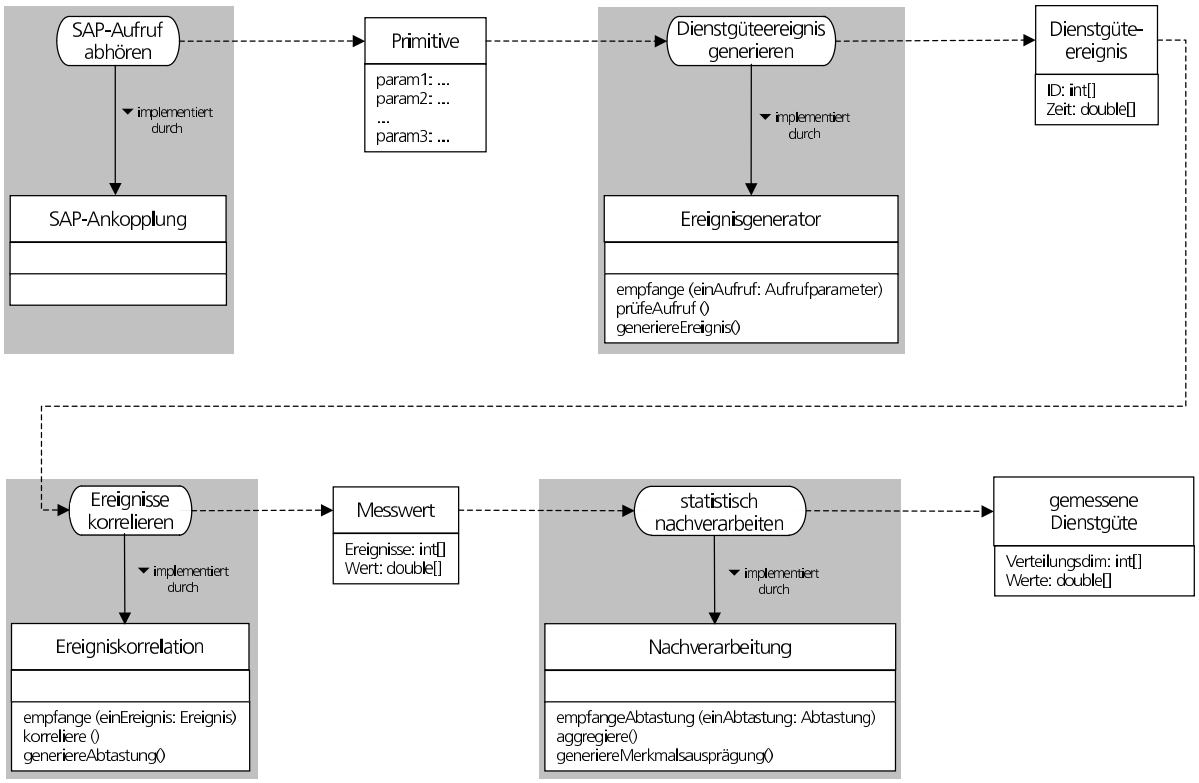

Abbildung 4.3.: verfeinerte Darstellung des Messprozesses im UML-Aktivitätsdiagramm zusammen mit aktivitätsimplementierenden Klassen

- Dienstgüteereignis generieren

  Das Resultat dieser Aktivität ist die Bereitstellung eines *Dienstgüteereignis*-Objekts. Entsprechend ist die diese Aktivität umsetzende Klasse als *Ereignisgenerator* bezeichnet. Um die Aktivität *Dienstgüteereignis generieren* umsetzen zu können, muss der Ereignisgenerator Primitiven empfangen können und stellt deshalb eine entsprechende Methode (*empfange*) zur Verfügung. Zur Generierung von Ereignissen wird ebenfalls eine entsprechende Methode zur Verfügung gestellt. Zusätzlich ist mit der Methode *prüfePrimitive* die Möglichkeit gegeben, Primitive zu überprüfen und somit nur Dienstgüteereignisse zu generieren, wenn beispielsweise die Parameterbelegung einem gewünschten Muster entspricht (bestimmte Adressen o.ä. enthält).

- Ereignisse korrelieren

  Die Klasse *Ereigniskorrelation* übernimmt die Aufgabe der Korrelation von Ereignissen. Zu diesem Zweck erhält sie eine Methode zum Empfangen von Ereignissen. Die eigentliche Korrelation der Ereignisse wird durch die Methode *korreliere* übernommen, die bei erfolgreicher Berechnung eines Abtastwerts die Methode *generiereMesswert* aufruft, um eine Instanz der Klasse *Messwert* aufzubauen und zur statistischen Nachverarbeitung weiterzureichen.

  Die Funktion der *korreliere*-Methode hängt vom jeweils modellierten Dienstgütemerkmal ab. Möglichkeiten zur Implementierung dieser Methode werden im folgenden Abschnitt 4.2.4 diskutiert.

- statistisch nachverarbeiten

  Der Grundsatz, jede Aktivität durch genau eine Klasse umzusetzen, wird

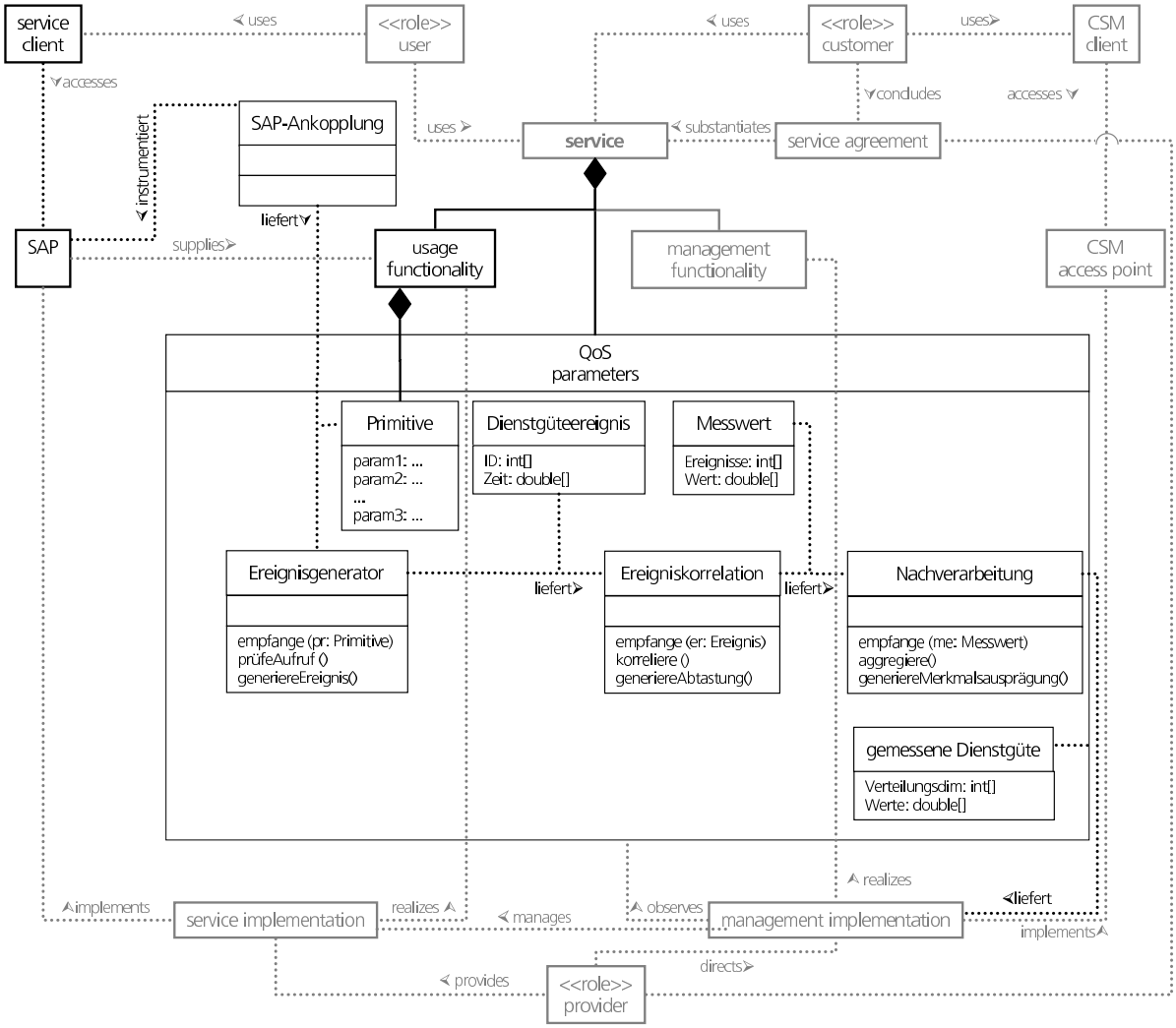

Abbildung 4.4.: UML-Klassenmodell zur Beschreibung des Messprozesses und Einordnung in das MNM-Dienstmodell

auch hier eingehalten. Die Klasse *Nachverarbeitung* implementiert mit der Kernmethode *aggregiere* die Aktivität *statistisch nachverarbeiten*. Wie bei den Klassen *Ereigniskorrelation* und *Ereignisgenerator* bereits beschrieben, enthält auch diese Klasse eine Methode zum Empfang der Eingabeobjekte, in diesem Fall vom Typ *Messwert*. Mit der Methode *generiereGemesseneDienstgüte* wird nach erfolgter Aggregation eine Instanz der Klasse *gemessene Dienstgüte* generiert und mit den ensprechenden Werten belegt.

Das entstandene Modell zeigt deutlich eine Trennung zwischen den Klassen, die zwischen Aktivitäten ausgetauscht werden, und solchen, die Teilaktivitäten des Prozesses umsetzen. Letztere weisen eine Reihe von Methoden zur Implementierung der jeweiligen Aktivität auf. Erstere können lediglich parametrisiert werden und legen damit (passive) Strukturen für den Datenaustausch fest, der wiederum von den aktivitätsimplementierenden Klassen durch entsprechende Methoden unterstützt wird. Für ein reines Klassenmodell, wie es in Abbildung 4.4 dargestellt ist, bietet es sich dementsprechend an, die aktivitätsimplementierenden Klassen durch Assoziationen aneinander zu binden und

diese Assoziationen durch die Austauschklassen als Assoziationsklassen zu beschreiben.

Bei diesem kanonischen Vorgehen fällt auf, dass für die Assoziation, die durch die Ergebnisklasse des Prozesses *gemessene Dienstgüte* beschrieben wird, keine Partnerklasse für die Klasse *Nachverarbeitung* existiert. Zur Einordnung des Gesamtmodells in das MNM-Dienstmodell [GHHK 02] bietet es sich an, dort ein entsprechendes "Assoziationsziel" zu suchen. Typischerweise werden die Werte einer Messung dem Dienstmanagement, im MNM-Dienstmodell durch die Klasse *management implementation* beschrieben, zur Verfügung gestellt. Die Klasse *SAP-Ankopplung* ist durch eine Assoziation mit der Bezeichnung *instrumentiert* an die Klasse *SAP* des MNM-Dienstmodells gebunden.

Damit zeigt sich auch eine Einordnung des gesamten Prozessmodells in das MNM-Dienstmodell. Alle beschriebenen Klassen, mit Ausnahme der Klasse *SAP-Ankopplung*, sind als Bestandteile der Klasse *QoS-Parameters* aus dem MNM-Dienstmodell dargestellt. Wie sich im Verlauf dieses Kapitels zeigen wird, reicht eine Spezifikation eben dieser Klassen aus, um ein Dienstgütemerkmal formal festzulegen.

In Abbildung 4.4 sind Klassen und Assoziationen des MNM-Dienstmodells, die nicht in direktem Bezug zur Modellierung des Messprozesses stehen ausgegraut, um die Darstellung übersichtlicher zu halten.

4.2.4. Methoden zur Parametrisierung des Messprozesses

In diesem Abschnitt werden Methoden und Verfahrensvorschriften erläutert, die eine eindeutige Abbildung bestehender Definitionen von Dienstgütemerkmalen auf das entwickelte Messprozessmodell erlauben sollen. Diese Richtlinien geben zugleich Auskunft über die Verwendung des Prozesses und liefern damit eine weitere Detaillierung.

Bei der Beschreibung eines Dienstgütemerkmals mit dem entwickelten Prozessmodell sind zwei grundsätzliche Sichtweisen zu unterscheiden. Einerseits die Definition eines Dienstgütemerkmals in seiner Bedeutung respektive in seinem Messverfahren, im Folgenden als **Definitionssicht** bezeichnet und durch das im vorherigen Abschnitt eingeführte Klassenmodell strukturiert. Andererseits die Beschreibung einer tatsächlich stattfindenden Messung, im Folgenden als **Messsicht** bezeichnet. Modellierungsverfahren für diese beiden Sichten werden im Folgenden beschrieben.

4.2.4.1. Definitionssicht

Bei der Definition eines Dienstgütemerkmals durch das Modell wird die prinzipielle Funktion dieses Merkmals, also der ablaufende Prozess, dargestellt. Dazu muss das existierende, generische Prozessmodell entsprechend parametrisiert werden, um die Spezifika eines Dienstgütemerkmals zu spiegeln. Diese Parametrisierung entspricht einer Verfeinerung des Modells nach den Vorgaben des jeweils zu modellierenden Dienstgütemerkmals. Es bietet sich somit an, ein Dienstgütemerkmal durch Verfeinerung der bestehenden Klassen zu modellieren.

Zur Verfeinerung werden drei unterschiedliche Formen der Vererbung benutzt:

- echte Vererbung:

  eine Verfeinerung, bei der in der abgeleiteten Klasse zusätzliche Methoden oder Attribute spezifiziert werden.

- Überladen:

  eine Verfeinerung, bei der bestehende Methoden der Oberklasse in der

abgeleiteten Klasse mit neuen/unterschiedlichen Parametern und/oder neuer Funktionalität definiert werden.

- konzeptionelle Vererbung:

  eine Verfeinerung, bei der zwar eine Klasse abgeleitet wird, innerhalb derer aber keinerlei Verfeinerung oder Neuimplementierung von Funktionalität erfolgt.

Diese Verfeinerungsformen sind in den jeweiligen Abbildungen durch entsprechende Symbole gekennzeichnet. Im Folgenden werden die Ableitungen von den einzelnen Basisklassen im Detail an Hand eines Beispiels beschrieben. Aus dieser Beschreibung werden dann grundsätzliche Verfahrensvorgaben für die Modellierung abgeleitet.

4.2.4.1.1. Beispieldefinition Abbildung 4.5 zeigt ein verfeinertes Modell, wie es sich für die Definition des Dienstgütemerkmals „Framefehlerrate bei Ethernet" ergibt. Die Verfeinerung des Modells wird in der Reihenfolge der Basisklassen vorgenommen, die zur besseren Übersicht ausgegraut sind.

- SAP-Ankopplung

  Speziell zur Rekonstruktion von Dienstprimitiven aus den Aufrufen am Ethernet-SAP wird die Klasse *Ethernet-Ankopplung* durch echte Vererbung von der Klasse *SAP-Ankopplung* abgeleitet. Die Implementierung dieser Klasse ist in der Lage, die für die Definition des Dienstgütemerkmals notwendigen Dienstprimitiven *send* und *receive* zu ermitteln.

- Primitive

  Die Dienstprimitive von Ethernet, die für die Spezifikation des Dienstgütemerkmals „Framefehlerrate" notwendig ist (receive) lässt sich durch echte Vererbung aus der Klasse *Primitive* ableiten. Die Attribute der Klasse *recv#* geben die Parametrisierungsmöglichkeiten der Dienstprimitive durch MAC-Adresse usw. wieder. Die jeweiligen Aufrufparameter können als Attribute der jeweiligen Klassen modelliert werden.

  Da diese verfeinerte Primitive von der *Ethernet-Ankopplung* erzeugt wird und später als „Eingabe" für den Ereignisgenerator dient, wird die Assoziation *liefert* auch zwischen der Klasse *recv#* und dem zugehörigen Ereignisgenerator etabliert.

- Ereignisgenerator

  Ein Ereignisgenerator, der aus einer bestimmten Unterklasse einer *Primitive* ein Dienstgüteereignis generieren soll, muss zu diesem Zweck angepasste Methoden besitzen. Allerdings müssen keine zusätzlichen Methoden beschrieben werden, sondern nur die Funktionalität der bisherigen Methoden angepasst, also überladen werden. Somit entsteht für jedes Ereignis eine Unterklasse von *Ereignisgenerator* mit entsprechend überladenen Methoden.

  Mit der Methode *prüfeAufruf* besteht die Möglichkeit, die übergebenen Parameter einer Primitive auf ein bestimmtes Muster bzw. bestimmte Eigenschaften zu prüfen. Im betrachteten Beispiel unterscheiden sich die beiden Unterklassen von *Ereignisgenerator* lediglich durch eine unterschiedliche Festlegung dieser Methode. Das Dienstgüteereignis *recv* wird immer erzeugt, wenn eine entsprechende Primitive beim Ereignisgenerator eintrifft. Das Dienstgüteereignis *fault* wird nur erzeugt, wenn der im Parameter *CRC* der Primitive übergebene CRC nicht mit einem auf Basis

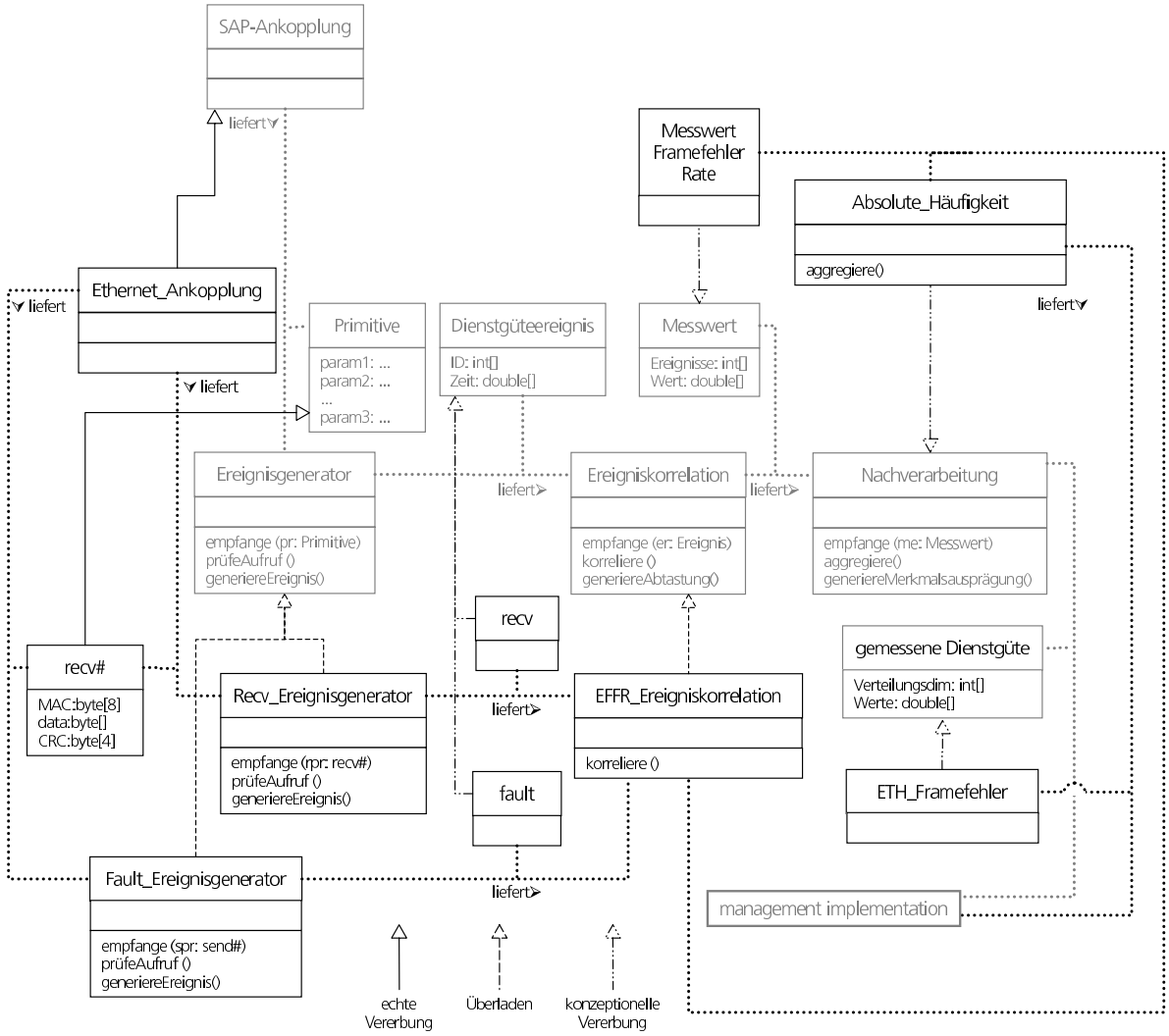

Abbildung 4.5.: verfeinertes Klassenmodell zur Beschreibung eines Dienstgüteparameters

der Daten im Attribut *data* berechneten CRC übereinstimmt.

- Ereignis

  Die bestehende Definition eines Ereignisses ist durchaus ausreichend, um auch für spezialisiertere Ereignisgeneratoren verwendet werden zu können. Bei der Übertragung von Ereignissen zwischen spezifischen Ereignisgeneratoren (hier im Beispiel *Send_Ereignisgenerator* und *Fault_Ereignisgenerator*) und der Klasse *Ereigniskorrelation* oder einer ihrer Unterklassen ließen sich die Instanzen der Klasse *Dienstgüteereignis* durch ihren jeweiligen Absender, dem entsprechenden Ereignisgenerator, einem bestimmten Typ zuordnen.

  Damit in der Modellierung alle für die Definition des jeweiligen Dienstgütemerkmals notwendigen Dienstgüteereignisse auch explizit definiert sind, werden die zu den jeweiligen Ereignisgeneratoren gehörenden Dienstgüteereignisse durch konzeptionelle Vererbung aus der Klasse *Dienstgüteereignis* dargestellt und analog zum Vorgehen bei der Verfeinerung der Klasse *Primitive* wieder an eigenständige Assoziationen zwischen den jeweiligen Verfeinerungen

von *Ereignisgenerator* und *Ereigniskorrelation* gebunden.

- Ereigniskorrelation

  Die Ereigniskorrelation muss in ihrer Funktionalität an die jeweils für das spezifische Dienstgütemerkmal notwendige Funktionalität (Empfang der jeweiligen Ereignisse und deren Korrellation) angepasst werden. Damit ist es sinnvoll, für jedes Dienstgütemerkmal einen spezifischen Ereigniskorrelator als Unterklasse von *Ereigniskorrelation* zu modellieren und die Methoden entsprechend zu überladen. Wie bereits für die Verfeinerungen der Klasse *Dienstgüteereignis* beschrieben, ist es sinnvoll, die Assoziation zur jeweiligen Verfeinerung von *Ereignisgenerator* darzustellen, um damit die Zusammenhänge zwischen den für die Definition eines Dienstgütemerkmals notwendigen Komponenten explizit zu machen.

- Messwert

  Ebenso, wie für die Klasse *Ereignis* geschehen, wird auch die Klasse *Messwert* konzeptionell verfeinert, um ihren Typ explizit kennzeichnen zu können. Diese Verfeinerung dient wiederum, analog zum Vorgehen bei den Klassen *Primitive* und *Dienstgüteereignis*, als Assoziationsklasse für die Relation zwischen der Verfeinerung von *Ereigniskorrelation* (hier im Beispiel *EFFR_Ereigniskorrelation*) und einer Verfeinerung von *Nachverarbeitung*.

- Nachverarbeitung

  Die Nachverarbeitung der ermittelten Messwerte muss ebenfalls nach den spezifischen Vorgaben des jeweils modellierten Dienstgütemerkmals erfolgen. Im Beispiel wird eine Darstellung der absoluten Häufigkeiten gewählt und eine entsprechende Klasse als Verfeinerung von *Nachverarbeitung* modelliert. Die Methode *aggregiere* wird entsprechend der gewünschten Funktionalität (Bildung einer absoluten Häufigkeit) überladen. Die Klasse *Absolute_Häufigkeit* wird durch eine Assoziation, die durch die Klasse *MesswertFramefehlerrateRate* beschrieben ist, an die Klasse *EFFR_Ereigniskorrelation* gebunden. Damit wird der Austausch einer bestimmten Klasse von Messwerten zwischen der Ereigniskorrelation und der Nachverarbeitung explizit dargestellt.

- gemessene Dienstgüte

  Aus der Nachverarbeitung resultiert die aktuelle, gemessene Dienstgüte. Diese wird durch eine konzeptionelle Vererbung aus der Klasse *gemesseneDienstgüte* modelliert, um auch in diesem Fall die Typisierung explizit zu machen. Entsprechend dem bisherigen Vorgehen wird auch diese Klasse, die letztlich zur Festlegung eines Datenaustauschformats dient, als Assoziationsklasse definiert. Sie beschreibt die Assoziation, die zwischen den Klassen *Absolute_Häufigkeit* und *management implementation* besteht.

4.2.4.1.2. Spezifikationsrichtlinien Das obige Beispiel zeigt bereits sinnvolle, grundsätzliche Vorgehensweisen, die im Folgenden nochmals explizit zusammengestellt werden, um als Spezifikationsrichtlinien für weitere Modellierungen verwendet werden zu können. Die entstehende Liste ist nach den einzelnen Klassen des Messprozesses entlang der Verarbeitungsreihenfolge gegliedert.

1. SAP-Ankopplung

   Die Ankopplung an den SAP des jeweils betrachteten Dienstes er-

folgt durch eine spezifische SAP-Ankopplung und damit durch eine durch echte Vererbung entstandene Verfeinerung der Klasse *SAP-Ankopplung*. Ihre Implementierung richtet sich nach der Implementierung des jeweiligen SAPs, an den „angekoppelt" wird. Als definiertes Ergebnis muss diese Implementierung die später benötigten Dienstprimitiven liefern können, z.B. in dem sie den SAP entsprechend instrumentiert.

Die Assoziation, welche die Oberklasse *SAP-Ankopplung* mit der Klasse *Ereignisgenerator* verbindet, ist bei der Verfeinerung ebenfalls abzubilden und wiederum als Assoziation mit den verfeinerten Klassen (von *Primitive* und *Ereignisgenerator*) zu modellieren. Die Klasse *SAP-Ankopplung* ist somit eine abstrakte Oberklasse, die ein grundsätzliches Konzept modelliert und keinerlei Methoden oder Attribute besitzt.

2. Primitive

   Aus der Klasse *Primitive* müssen für jede durch die Dienstimplementierung definierte Primitive, aus deren Auftreten Dienstgüteereignisse abgeleitet werden sollen, eigenständige Klassen abgeleitet werden, die keinerlei Methoden, aber die Parameter der jeweiligen Dienstprimitive als Attribute enthalten. Damit ist ein expliziter Austausch der Dienstprimitiven möglich.

   Wie bereits bei der Verfeinerung der Klasse *SAP-Ankopplung* ist auch in diesem Fall zu beachten, dass die Assoziation, die durch die Klasse *Primitive* beschrieben wird, für ihre Verfeinerung neu etabliert werden muss. Assoziationsendpunkte sind dabei die entsprechenden Verfeinerungen von *SAP-Ankopplung* und *Ereignisgenerator*.

   Auch die Klasse *Primitive* stellt damit eine abstrakte Oberklasse dar, die eine rein konzeptionelle Sicht wiedergibt. Die für die Definition eines Dienstgütemerkmals relevanten Festlegungen geschehen in entsprechenden Unterklassen.

3. Ereignisgenerator

   Die Funktion eines Ereignisgenerators ist spezifisch für die jeweils erzeugten Dienstgüteereignisse bzw. die verarbeiteten Dienstprimitiven . Deshalb muss für jede Ereignisklasse, die später verwendet werden soll, eine Unterklasse der Klasse *Ereignisgenerator* definiert werden, welche die Methoden *prüfeAufruf* und *generiereEreignis* entsprechend anpasst.

   Die Klasse *Ereignisgenerator* stellt diese Methoden bereits als Stub mit minimaler Implementierung zur Verfügung. In der Methode *prüfeAufruf* können für den jeweiligen Typ der Dienstprimitive spezifische Filterfunktionen festgelegt werden. Die Methode *generiereEreignis* muss Funktionalität implementieren, die einen enstprechenden Typ eines Dienstgüteereignisses (Unterklasse von *Dienstgüteereignis*) erzeugt und dessen Attribute entsprechend belegt, also einen Zeitstempel und einen eindeutigen Identifikator für das jeweilige Ereignis festlegt.

4. Dienstgüteereignis

   Aus der Klasse *Dienstgüteereignis* werden durch konzeptionelle Vererbung für jeden Ereignistyp, der durch eine eigenständige Verfeinerung der Klasse *Ereignisgenerator* erzeugt wird, explizit Unterklassen abgeleitet, um die für die Definition

des Dienstgütemerkmals notwendigen Ereignistypen explizit zu machen.

Da die Klasse *Dienstgüteereignis* eine Assoziationsklasse ist, wird auch in diesem Fall das bereits mehrfach angewandte Vorgehen, die Assoziation zwischen den verfeinerten Klassen explizit zu etablieren, erneut umgesetzt.

5. Ereigniskorrelation

   Ein Ereigniskorrelator realisiert Funktionalität, die spezifisch für ein bestimmtes Dienstgütemerkmal ist. Zur Definition eines Dienstgütemerkmals wird genau **ein** Ereigniskorrelator benötigt. Dieser wird als Verfeinerung der Klasse *Ereigniskorrelation* definiert. Die im Gerüst bereits vorhandenen Methoden (Stub) werden mit der für die Korrelation notwendigen Funktionalität überladen.

   Im Sinne einer einfachen Vergleichbarkeit von Definitionen bietet es sich an, für die Definition von Korrelationsfunktionalität auf ein Repository zurückzugreifen, das gegebenenfalls erweitert werden kann. Die dafür notwendigen Mechanismen werden im folgenden Abschnitt 4.3 entwickelt.

   Um die Durchgängigkeit der bisherigen Modellierung weiterzuführen, werden auch bei diesem Verfeinerungsschritt die Assoziationen explizit mitverfeinert. Dabei erhält die eben definierte Verfeinerung von *Ereigniskorrelation* eine Assoziation zu jeder Verfeinerung von *Ereignisgenerator*, von der sie Ereignisse verarbeitet. Diese Assoziationen werden durch die zugehörigen Verfeinerungen von *Dienstgüteereignis* beschrieben (Assoziationsklasse). In gleicher Weise wird mit der Assoziation zur Verfeinerung der Klasse *Nachverarbeitung* verfahren.

6. Messwert

   Aus der Klasse *Messwert* wird durch konzeptionelle Vererbung der Typ eines Messwerts, den ein bestimmter (verfeinerter) Korrelator liefert, explizit abgeleitet. Diese Verfeinerung dient als Assoziationsklasse zur Beschreibung der Assoziation zwischen der Verfeinerung von *Ereigniskorrelation* und der entsprechenden Verfeinerung der Klasse *Nachverarbeitung*. Durch diese Art der Modellierung wird wiederum die explizite Typisierung der Datenaustauschstrukturen sichergestellt.

7. Nachverarbeitung

   Für jedes Dienstgütemerkmalmerkmal wird gewöhnlich eine eigenständige Art der statistischen Nachverarbeitung definiert. Häufig werden aber ähnliche Nachverarbeitungsfunktionen verwendet. Zur Modellierung einer speziellen Nachverarbeitungssklasse für ein Dienstgütemerkmal bietet sich damit folgendes Vorgehen an:

   Aus der Klasse *Nachverarbeitung* wird eine für das jeweilige Dienstgütemerkmal spezifische Klasse abgeleitet. Die Verarbeitungsmethoden werden entsprechend überladen. Dabei bietet es sich an, eine Bibliothek von Funktionen zur statistischen Nachverarbeitung bereitzustellen, um, ähnlich wie bei der Ereigniskorrelation, einen höheren Grad an Vergleichbarkeit für die einzelnen Definitionen sicherstellen zu können. (Möglichkeiten zur Definition von Nachverarbeitungsklassen auf Basis eines Repositories beschreibt der folgende Abschnitt 4.3.2.3.)

   Wie bereits für die Klassen *Ereigniskorrelation* und *Messwert* beschrieben, ist eine Assoziation zur Verfeinerung von *Ereigniskorrelation* zu

erstellen, die durch die Verfeinerung von *Messwert* als Assoziationsklasse beschrieben wird. Ebenso ist eine Assoziation zur Klasse *management implementation* (aus dem MNM-Dienstmodell) zu etablieren, die durch eine Verfeinerung der Klasse *gemessene Dienstgüte* als Assoziationsklasse beschrieben wird. Da eine Verfeinerung der Klasse *management implementation* nicht zwingend existiert, kann das sonst angewandte Konzept der „Verfeinerung von Assoziationen" hier nicht umgesetzt werden.

8. gemessene Dienstgüte

   Zur Beschreibung des Ergebnisses des Messprozesses dient die Klasse *gemesseneDienstgüte*. Für ein konkretes Dienstgütemerkmal ist auch in diesem Fall durch konzeptionelle Vererbung eine Unterklasse zu erzeugen, um den Ergebnistyp explizit zu machen.

   Jegliche gemessene Dienstgüte stellt nach der Definition des allgemeinen Messprozesses eine Art Verteilungsfunktion dar. Es bietet sich also an, zusätzliche Klassen mit typischen Verteilungsfunktionen zur Verfügung zu stellen und die Verfeinerung von *gemessene Dienstgüte* zusätzlich von diesen Klassen erben zu lassen, um somit eine einfache und vergleichbare Definition eines speziellen Dienstgütemerkmals zu erlauben.

   Die Verfeinerung der Klasse *gemessene Dienstgüte* ist an die Assoziation zur Klasse *management implementation* zu binden, wie oben für die Klasse *Nachverarbeitung* bereits erläutert.

Zusammenfassend lassen sich aus den Spezifikationsvorschriften die folgenden grundsätzlichen Regeln ableiten: Echte Vererbung wird angewandt, wenn in der Modellierung Anpassungen an vorhandene Funktionalität notwendig sind (z.B. bei der Modellierung der SAP-Ankopplung). Überladen von Klassen wird angewandt, wenn die Signaturen von Funktionalitäten festgelegt sind und/oder Bibliotheken von Funktionen vorliegen (z.B. bei der Modellierung der Ereigniskorrelation oder der Nachverarbeitung). Konzeptionelle Vererbung wird angewandt, wenn auszutauschende Informationen explizit zu typisieren sind (z.B. bei der Definition von Ereignissen, Abtastungen, Dienstgütemerkmalsausprägung).

4.2.4.2. Messsicht

Neben der statischen Betrachtung eines Klassenmodells, wie im vorherigen Abschnitt, ist zur Modellierung und Beschreibung des in Abschnitt 4.2.1.1 eingeführten allgemeinen Messprozesses die Betrachtung der dynamischen Abläufe unerlässlich. Diese Dynamik tritt allerdings erst beim Ablauf einer konkreten Messung auf. Die Beschreibung der Messsicht erfolgt deshalb durch ein UML-Sequenzdiagramm auf Basis der bereits definierten Klassen.

Wie bereits bei der statischen Beschreibung, der Definitionssicht, im vorherigen Abschnitt wird auch hier zunächst eine beispielhafte Definition beschrieben, aus der dann grundsätzliche Spezifikationsrichtlinien abgeleitet werden.

4.2.4.2.1. Beispieldefinition Das bereits eingeführte Beispiel, die Framefehlerrate bei Ethernet, wird in dieser Betrachtung weitergeführt. Abbildung 4.6 zeigt ein UML-Sequenzdiagramm, das eine Messung in ihrem zeitlichen Ablauf veranschaulicht. Zur besseren Übersicht sind Instanzen, die verarbeitende Funktionen umsetzen, hochkant, solche, die zum Austausch von Informationen dienen, waa-

gerecht angeordnet. Zur Vereinfachung werden Instanzen direkt auf einen Übergabepfeil angeordnet. Diese Kurzform steht für den Aufbau einer neuen Instanz mit anschließendem Versenden derselben. Der Pfeil beschreibt in diesem Fall also nicht den Aufruf einer Methode, sondern die Übertragung einer Instanz (waagerecht dargestellt) zwischen zwei anderen Instanzen (hochkant dargestellt). Zusätzlich zeigt das Diagramm noch die Assoziationen zwischen den Instanzen, wie sie bei der Beschreibung des Klassenmodells im vorherigen Abschnitt eingeführt wurden.

Die folgende Darstellung beschreibt den im Sequenzdiagramm dargestellten Ablauf entlang der Zeitachse, fasst dabei aber logisch aufeinander aufbauende Abläufe zusammen. Zur besseren Übersicht sind die einzelnen Erklärungsschritte im Diagramm durch Nummern wiedergegeben.

1. Registrieren eines Aufrufs durch *:EthernetAnkopplung*

   Der beispielhafte Ablauf beginnt mit dem Registrieren eines Aufrufs durch die Instanz *:EthernetAnkopplung*. Dies wird durch einen eingehenden gestrichelten Pfeil dargestellt. Daraufhin wird eine Instanz der Klasse *recv#* erzeugt und an den entsprechenden Ereignisgenerator, in diesem Fall eine Instanz der Klasse *Recv_Ereignisgenerator*, versandt.

2. Erzeugung eines Recv-Ereignisses

   Dieser Ereignisgenerator prüft die Parameter der Primitive und erzeugt ein passendes Dienstgüteereignis, eine Instanz der Klasse *recv*, welche an den Korrelator, eine Instanz der Klasse *EFFR_Ereigniskorrelation*, geschickt wird.

3. Registrieren weiterer Aufrufe durch *:EthernetAnkopplung*

   Von der Instanz *EthernetAnkopplung* werden auch Primitve an einen weiteren Ereignisgenerator *:Fault_Ereigniskorrelator* geschickt. Dieser erzeugt allerdings nur beim Empfang eines fehlerhaften Frames ein Dienstgüteereignis. Damit bleibt bei einem Großteil der empfangenen Primitiven eine Reaktion aus.

4. Erzeugung eines Fault-Ereignisses

   Durch das Objekt *:Fault_Ereignisgenerator* wird eine Instanz der Klasse *fault* erzeugt und an die Instanz *:ERFW_Ereigniskorrelation* gesandt, wenn festgestellt wird, das berechneter CRC und CRC im entsprechenden Attribut der Primitive nicht übereinstimmen.

5. Berechnung eines Messwerts

   Aus den erhaltenen, eingegangenen Ereignissen kann das Objekt *:EFFR_Ereigniskorrelation* einen Messwert berechnen. Im Beispiel sind für diese Berechnung beliebig viele Recv-Ereignisse und mindestens ein Fault-Ereignis notwendig. Passend zum Messwert wird eine Instanz der Klasse *MesswertFramefehlerrate* erzeugt und zur statistischen Verarbeitung an das Objekt *:Absolute_Häufigkeit* geschickt.

6. Sammlung von Messwerten und Berechnung einer Verteilung

   Beim Verarbeitungsobjekt *:Absolute_Wahrscheinlichkeit* müssen genügend Objekte eingehen, um eine statistische Zusammenfassung berechnen zu können. In Abbildung 4.6 ist die dafür verstrichene Zeit durch eine „Bruchkante" angedeutet. Liegen ausreichend viele Objekte zur Berechnung vor, erfolgt die Aggregation durch das Objekt *:Absolute_Häufigkeit*. Dort wird eine entsprechende Instanz der Klasse *ETH_Framefehler* erzeugt und dem Dienstmanagement, durch

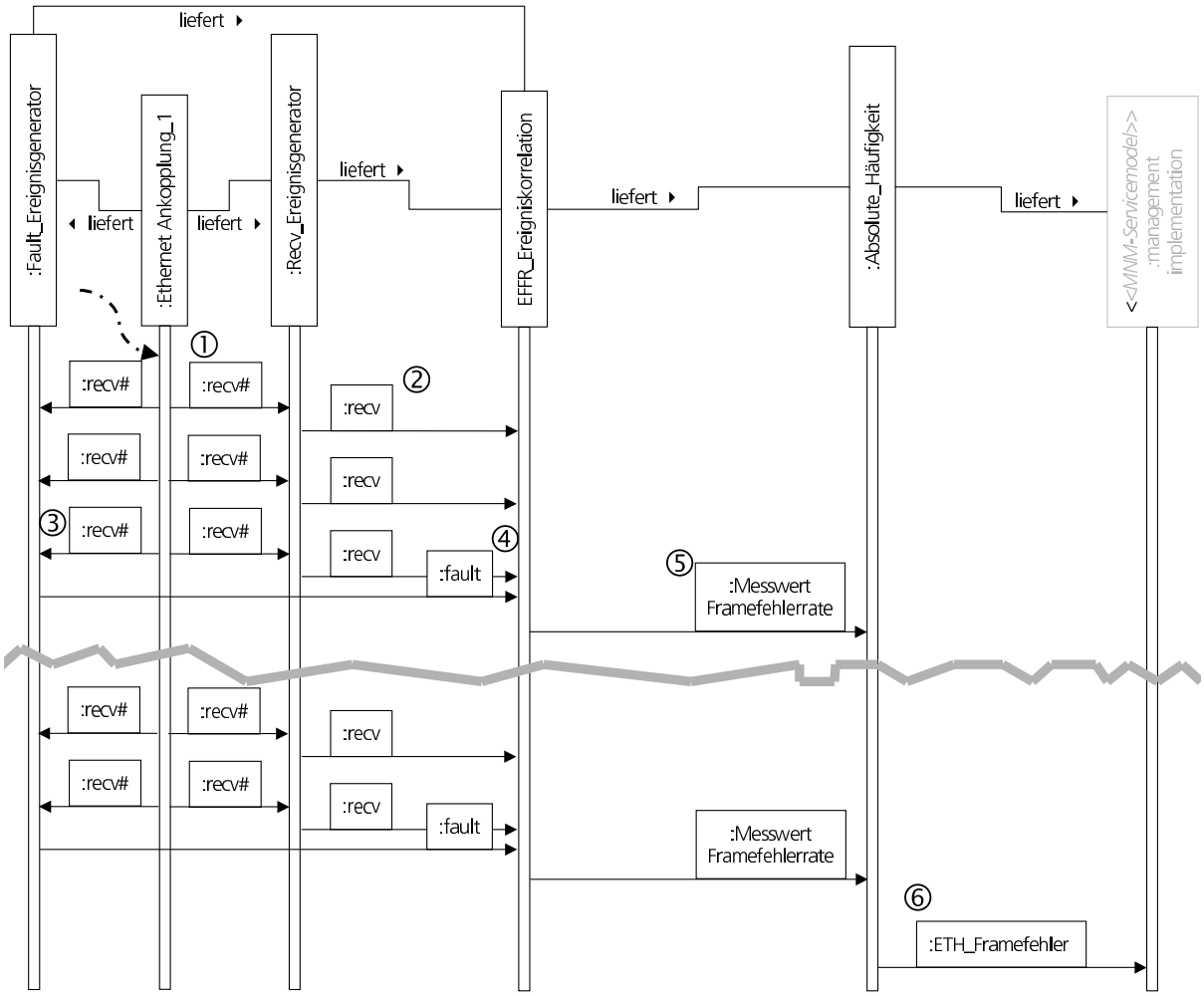

Abbildung 4.6.: Messung eines Dienstgütemerkmals (Messsicht), dargestellt im UML-Sequenzdiagramm

Übertragung an das Objekt *:management_implementation* aus dem MNM-Dienstmodell, zurVerfügung gestellt.

4.2.4.2.2. Spezifikationsrichtlinien Der im Beispiel an Hand des bisherigen Klassenmodells dargestellte Messvorgang veranschaulicht bereits die prinzipielle Reihenfolge bei der Instanzierung der eingeführten Klassen. Aus der Darstellung lassen sich folgende Spezifikationsrichtlinien ableiten:

1. Rechtzeitige Bereitstellung der Verarbeitungsobjekte

   Bevor eine Messung beginnen kann, müssen die der Definition des zu messenden Dienstgütemerkmals entsprechenden Verarbeitungsobjekte existieren.

2. Explizite Assoziationen zwischen Verarbeitungsobjekten

   Bei der Bereitstellung / Instanzierung der Verarbeitungsobjekte zu Beginn einer Messung ist sicherzustellen, dass alle Assoziationen explizit festgelegt sind, die Objekte also Referenzen auf die jenigen Objekte halten, zu denen sie in Beziehung stehen.

3. Möglichkeiten zur Übertragung von Objekten

   Im dargestellten Beispiel wird implizit davon ausgegangen, dass Möglichkeiten existieren, Objekte zwischen

anderen Objekten zu übertragen. Entsprechende Funktionalitäten müssen explizit, in Form geeigneter Methoden, zur Verfügung stehen.

4. Attributbelegungen der Datentransportklassen bei Instanzierung

   Bei der Instanzierung von Datentransportklassen muss sichergestellt sein, dass ihre jeweiligen Attribute korrekt belegt werden. Dafür müssen geeignete Konstruktoren bereitgestellt werden.

5. Keinerlei Zeitsynchronisation

   Die Darstellung in Abbildung 4.6 suggeriert eine totale Zeitachse. Das in Abschnitt 4.2.1.1 eingeführte, allgemeine Messverfahren trifft aber bezüglich der Zeitachsen keinerlei Festlegungen. Es ist also davon auszugehen, dass keine globale Zeitachse existiert. Die Korrelationsmechanismen, aber auch die Techniken zur Übertragung von Objekten, müssen entsprechend realisiert werden. D.h. es müssen Zeitstempel der jeweiligen Primitiven und Dienstgüteereignisse bei der Korrelation verwendet werden und es wird nicht auf den Empfangszeitpunkt eines Dienstgüteereignisses zurückgegriffen. Wird für eine Messung (z.B. des Delays) eine globale Zeitachse benötigt, so wird diese am sinnvollsten in der SAP-Ankopplung implementiert. Damit bleibt die weitere Verarbeitung im Messporzess von der Problematik einer globalen Zeitachse unberührt.

6. Puffermöglichkeiten bei Empfängern

   Da beim Austausch von Objekten keine gemeinsame Zeitachse existiert, sind mögliche Empfänger so auszulegen, dass sie eingehende Objekte puffern können und nicht auf eine eindeutige Anordnung der gesendeten Objekte auf einer globalen Zeitachse angewiesen sind, sondern die zeitliche Ordnung der Objekte über ihre Zeitstempel sicherstellen.

Aus der Untersuchung der Messsicht auf die Definition eines Dienstgütemerkmals sind eine Reihe von Spezifikationsrichtlinien hervorgegangen, die in ihrem Detaillierungsgrad nur direkt bei der Spezifikation eines ausgewählten Dienstgütemerkmals umgesetzt werden können, es sei denn, die Spezifikation wird weitgehend automatisch durchgeführt. Ein entsprechendes Automatisierungskonzept wird im Folgenden vorgestellt und umgesetzt.

4.3. Rechnergestützte Umsetzung der Dienstgütespezifikation im Dienstlebenszyklus

In den vorhergehenden Abschnitten wurde im Detail beschrieben, wie die Spezifikation eines Dienstgütemerkmals auf Basis des im Abschnitt 4.2.1.1 eingeführten generischen Messprozesses umgesetzt werden kann. Dabei wurden mit den Spezifikationsrichtlinien Regeln aufgestellt, nach denen bei der Umsetzung einer Dienstgütemerkmalsspezifikation verfahren werden sollte.

In diesem Abschnitt wird ein rechnergestützter Prozess vorgestellt, der die Umsetzung einer Spezifikation, den Spezifikationsrichtlinien folgend automatisiert. Bevor dieser Prozess im Abschnitt 4.3.2 entworfen und detailliert beschrieben wird, stellt der folgende Abschnitt 4.3.1 die bei der Umsetzung einer Spezifikation mit dem vorgestellten Messprozessmodell entstehenden Teilergebnisse im Bezug zum Dienstlebenszyklus dar und legt somit das Fundament für das anschließend beschriebene Automatisierungskonzept.

4.3.1. Ableitungsschritte im Bezug zum Lebenszyklus

Die Darstellung der Methoden zur Parametrisierung des generischen Messprozesses in Abschnitt 4.2.4 hat verdeutlicht, dass aus dem generischen Modell schrittweise unterschiedliche Teilergebnisse, die sog. Sichten auf die Spezifikation eines Dienstgütemerkmals, entstehen.

Nach der Problemstellung aus Abschnitt 2.2 gilt es, eine Spezifikationstechnik zu entwickeln, welche die in der jeweiligen Phase des Dienstlebenszyklus notwendigen Informationen automatisch aus den Festlegungen in der Verhandlungsphase generieren kann. Deshalb wird hier der Bezug der Lebenszyklusphasen zu den bereits eingeführten Sichten verdeutlicht um darstellen zu können, welche Sicht in welcher Phase des Lebenszyklus verwendet werden muss. Damit ergibt sich zugleich eine Reihenfolge bei der Erzeugung der Sichten, die der noch zu entwickelnde Automatisierungsprozess einhalten muss.

- Verhandlungsphase

  In der Verhandlungsphase steht außer der Spezifikation, die vom Provider zusammen mit dem Customer entworfen wurde, keine weitere Information zur Verfügung; es werden aber in dieser Phase auch keinerlei weitere Informationen benötigt. Um das Konzept der Sichten durchgängig anzuwenden, wird die in der Verhandlungsphase vorliegende Spezifikation als Entwicklersicht bezeichnet, da sie von den Entwicklern des Dienstes, Customer und Provider, entworfen wird.

- Bereitstellungsphase

  Implementierungsleitfäden, die angeben, welche funktionalen Blöcke zu realisieren sind, werden in der Bereitstellungsphase benötigt, um eine strukturierte, an den Maßgaben der Dienstgütespezifikation ausgerichtete Implementierung zu erreichen. Die Definitionssicht erfüllt genau diesen Anspruch. Sie spezifiziert notwendige Schnittstellen und gibt zugleich, durch die definierten Klassen, einen Rahmen für die Implementierung funktionaler Blöcke wieder.

- Nutzungsphase

  In der Nutzungsphase muss ein ablauffähiges Messsystem zur Verfügung stehen, um die spezifizierten Dienstgütemerkmale auch erfassen zu können. Mit der Messsicht steht eine Festlegung eben dieses Messsystems bis auf die Ebene einzelner Instanzen zur Verfügung.

4.3.2. Rechnergestützer Ableitungsprozess

Nach den oben getroffenen Überlegungen muss ein rechnergestützter Ableitungsprozess die Definitionssicht und die Messsicht aus der Entwicklersicht generieren können und im Idealfall das Instanzmodell der Messsicht soweit vorbereiten, dass es durch den Provider beim Übergang in die Nutzungsphase gestartet werden kann. In den folgenden Abschnitten wird dementsprechend das grundsätzlich verfolgte Automatisierungskonzept (Abschnitt 4.3.2.1), eine mögliche Umsetzung der aufgestellten Spezifikationsrichtlinien innerhalb dieses Konzeptes (Abschnitt 4.3.2.2) sowie abschließend ein detailliertes Design des Automatisierungsprozesses (Abschnitt 4.3.2.3) als Basis für eine spätere Implementierung beschrieben.

4.3.2.1. Automatisierungskonzept

Bei den bisher vorgenommenen Spezifikationen von Dienstgütemerkmalen wurde das Modell des allgemeinen Messprozesses schrittweise erweitert. Diese Erweiterungen wurden einerseits durch die im-

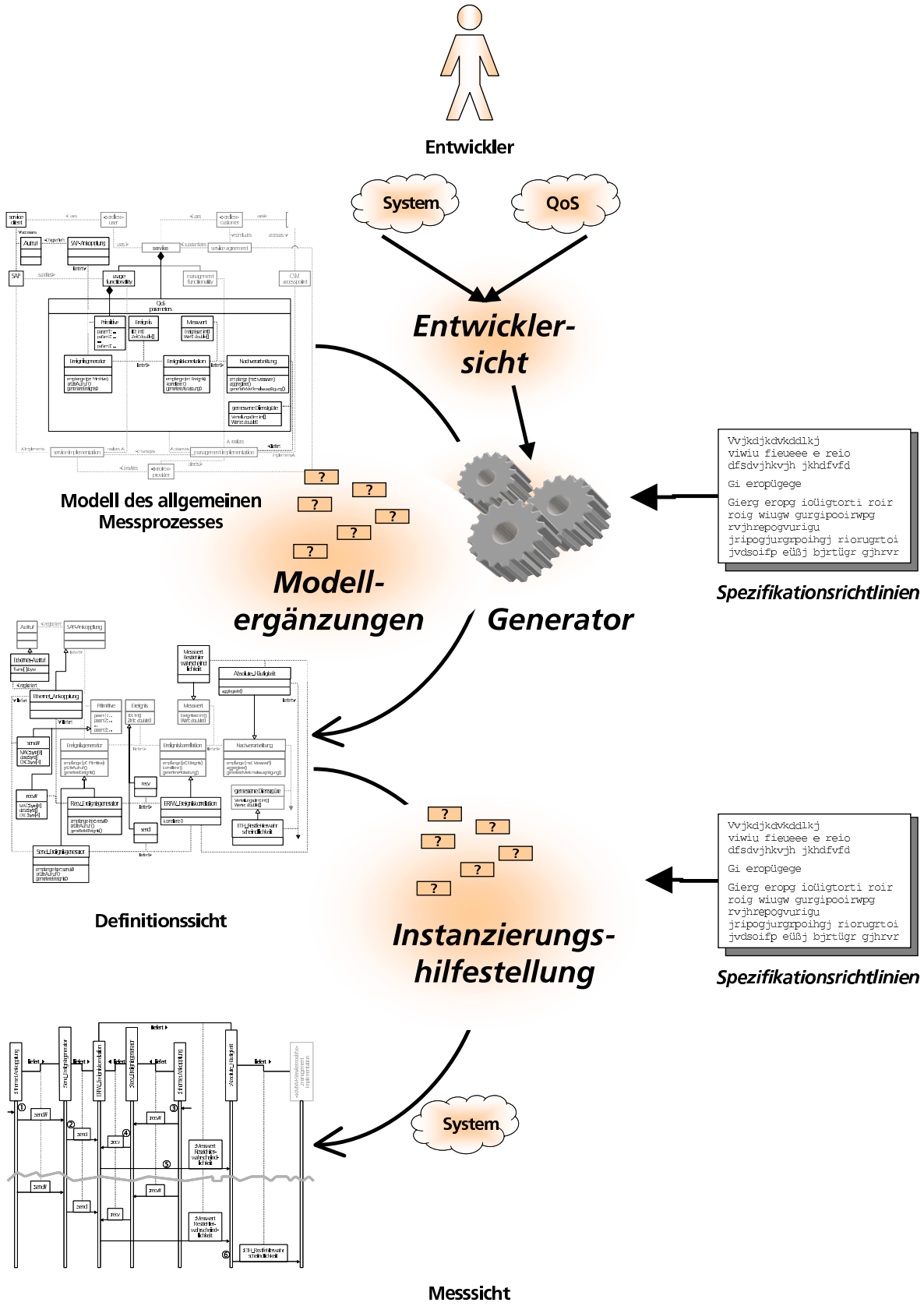

Abbildung 4.7.: Automatisierungskonzept (rechts) im Vergleich zum manuellen Entwurf (links)

plizite Definition des zu beschreibenden Dienstgütemerkmals und anderseits durch die Spezifikationsrichtlinien geleitet.

Entsprechend muss ein Automatisierungskonzept von einer expliziten Definition des zu beschreibenden Dienstgütemerkmals in Form der Entwicklersicht ausgehen, aus der die weitern Sichten, Definitions- und Messsicht, abgeleitet werden. Da sowohl die Definitionssicht als auch die Messsicht Anbindungen an die Dienstfunktionalität aufweisen, müssen in der Entwicklersicht auch Informationen über die Dienstfunktionalität, sofern diese für die Spezifikation von Dienstgütemerkmalen relevant ist, enthalten sein.

Zweckmäßigerweise wird der Umfang der in der Entwicklersicht festzulegenden Informationen erst bestimmt, wenn der Automatisierungsprozess vollständig spezifiziert ist. Damit lässt sich sicherstellen, dass in der Entwicklersicht nur wirklich notwendige Informationen festgelegt werden und diese somit übersichtlich und leicht strukturierbar bleiben.

Der in Abbildung 4.7 dargestellte Automatisierungsprozess verfolgt das Ziel, notwendige Festlegungen nur genau einmal in den Spezifikationsprozess einzubringen. Dazu werden Aussagen über das betrachtete System und das zu spezifizierende Dienstgütemerkmal in der Entwicklersicht zusammengefasst. Aus dieser Darstellung, deren genaue Form, wie oben beschrieben, erst nach der Spezifikation des Automatisierungsprozesses erfolgt, erzeugt ein Generator die Definitionssicht und eine Instanzierungshilfestellung, mit deren Hilfe die Messsicht automatisch aus der Definitionssicht aufgebaut werden kann. Beim Aufbau der Messsicht ist allerdings nochmals ein „Eingriff" durch den Entwickler notwendig: Die Ankopplung an einen bestehenden, implementierungsspezifischen SAP muss hier manuell durch den Entwickler erfolgen, da zur Realisierung dieser Ankopplung Wissen über die Implementierung des SAPs notwendig ist. Zur Unterstützung des Generators wird das Modell des allgemeinen Messprozesses ergänzt, um die Generierung der Definitionssicht zu erleichtern.

Sowohl der Aufbau dieser Modellergänzungen als auch die Funktion des Generators und der Instanzierungshilfestellung werden im Folgenden schrittweise aus den bereits bestehenden Spezifikationsrichtlinien für die Definitions- und die Messsicht abgeleitet. Aus diesen Design-Entscheidungen lassen sich bottom-up Aussagen über die in der Entwicklersicht notwendigen Festlegungen treffen. Wird der Automatisierungsprozess vollständig spezifiziert, ergeben sich die in der Entwicklersicht notwendigen Definitionen als notwendige Eingabe für den Generator automatisch.

Damit steht mit der Entwicklersicht ein definitiver Ausgangspunkt bei der Spezifikation von Dienstgütemerkmalen zur Verfügung. Wie sich in Abschnitt 4.4 zeigen wird, erfüllt die Entwicklersicht die in Abschnitt 2.4 aufgestellten Anforderungen an eine umfassende und zugleich leichtgewichtige Beschreibung von Dienstgütemerkmalen. Zusammen mit dem eingeführten Automatisierungsprozess steht damit eine durchgängige Methode zur Spezifikation und Messung von Dienstgütemerkmalen auf Basis einer einzigen Definition, der Entwicklersicht, zur Verfügung.

Der folgende Abschnitt 4.3.2.2 führt ein grundsätzliches Mapping der bestehenden Spezifikationsrichtlinien auf den vorgeschlagenen Automatisierungsprozess durch. Im daran anschließenden Abschnitt 4.3.2.3 wird auf Basis dieser Zuordnung von Spezifikationsrichtlinien zu einzelnen Bausteinen des Automatisierungsprozesses ein Design für dessen Implementierung entworfen. Eine konkrete, prototypische Implementierung dieses Designs wird in Abschnitt 4.3.3 vorgestellt.

4.3.2.2. Umsetzung der Spezifikationsrichtlinien

Entsprechend den Überlegungen am Ende des vorherigen Abschnitts werden die im Verlauf von Abschnitt 4.2.4 entwickelten Spezifikationsrichtlinien im vorgestellten Automatisierungsprozess durch dessen Bausteine (Entwicklersicht, Generator, Modellergänzung, Instanzierungshilfestellung) umgesetzt. Das heißt, aus den Spezifikationsrichtlinien werden Anforderungen an die einzelnen Bausteine des Automatisierungsprozesses abgeleitet.

Oberstes Ziel ist dabei, die neu festzulegende Entwicklersicht so minimal wie irgendmöglich zu definieren, damit sich der Einsatz des Automatisierungsmechanismus in einem für den Entwickler deutlich niedrigeren Spezifikationsaufwand niederschlägt.

Zur besseren Übersicht sind nachfolgend alle Spezifikationsrichtlinien nochmals in der Reihenfolge ihrer Erklärung in Abschnitt 4.2.4 aufgelistet. Für jede dieser Richtlinien wird innerhalb dieser Aufzählung beschrieben, in welcher Weise für welchen Baustein Anforderungen entstehen. Abbildung 4.8 veranschaulicht, in welchen Teilen des Automatisierungsprozesses die einzelnen Spezifikationsanforderungen umgesetzt werden.

1. SAP-Ankopplung

    a) Für jede mögliche Assoziation zwischen *SAP* und *service client* genau eine Verfeinerung von *SAP-Ankopplung* bilden

       Die Ankopplung an einen SAP ist hochgradig implementierungsspezifisch und kann damit nicht vom Automatisierungsprozess direkt unterstützt werden. Es kann aber vorgegeben werden, welche Primitiven diese Ankopplung liefern muss, wie in 1b gefordert.

    b) SAP-Ankopplung muss benötigte Primitiven liefern

       Bei der Implementierung der spezifischen SAP-Ankopplung ist darauf zu achten, dass alle für die Spezifikation eines Dienstgütemerkmals notwendigen Dienstprimitiven von der/den SAP-Ankopplungen geliefert werden können.

    c) Assoziationen explizit mit verfeinern

       Die Verfeinerung von *SAP-Ankopplung* muss eine entsprechend verfeinerte Assoziation zum jeweiligen Ereignisgenerator aufbauen, die durch die jeweils gelieferte Primitive beschrieben wird. Dazu können durch Modellergänzungen und vom Generator Anknüpfungspunkte im Skelett der Verfeinerungen von *Primitive* und *Ereignisgenerator* geschaffen werden, so dass der Aufwand für den Entwickler minimiert wird (siehe 2a) .

    d) SAP-Ankopplung ist abstrakte Oberklasse

       Die Klasse *SAP-Ankopplung* an sich erhält keinerlei Implementierung. Diese erfolgt erst in einer für das Dienstgütemerkmal und den jeweiligen SAP spezifischen Verfeinerung. *SAP-Ankopplung* ist somit per se eine abstrakte Oberklasse.

2. Primitive

    a) eigenständige Unterklassen für jede Primitive bilden (echte Vererbung)

       Werden geeignete Festlegungen in der Entwicklersicht getroffen, kann der Generator aus einer

		"Handarbeit"	Entwicklersicht	Modell-ergänzungen	Generator	Instanzierungs-hilfestellung
SAP-Ankopplung	Unterklasse für jede Assoziation zwischen SAP und service client	●				
	muss benötigte Primitiven liefern	●				
	Assoziationen explizit mitverfeinern	●		●		●
	abstrakte Oberklasse	per se				
Primitive	Unterklassen für jede Dienstprimitive		●		●	
	Parameter der jeweiligen Primitive als Attribute		●		●	
	Assoziationen explizit mitverfeinern		●	●	●	
	abstrakte Oberklasse	per se				
Ereignisgenerator	Unterklasse für jedes benötigte Ereignis		●		●	
	Methode prüfeAufruf überladen		●		●	
	Methode generiereEreignis überladen		●		●	
	Assoziationen explizit mitverfeinern		●	●		●
Dienstgüte-ereignis	Unterklasse für jeden Ereignistyp		●		●	
	Assoziationen explizit mitverfeinern		●	●	●	
Ereigniskorrelation	Unterklasse für jede Definition eines Dienstgütemerkmals		●		●	●
	Korrelationsmethode überladen		●		●	
	Repository für Korrelationmethoden bereitstellen		●	●		
	Assoziationen explizit mitverfeinern			●		●
Messwert	Unterklasse für jede Verfeinerung von Ereigniskorrelation		●		●	
	Assoziationen explizit mitverfeinern		●	●	●	
Nachverarbeitung	Unterklasse für jedes definierte Dienstgütemerkmal		●		●	●
	Verarbeitungsmethoden überladen		●		●	
	Repository für Verarbeitungsmethoden bereitstellen		●	●		
	Assoziation management implementation erhalten	●		●		●
gemessene Dienstgüte	Unterklasse für jedes definierte Dienstgütemerkmal		●		●	
	"Vererbungsschicht" mit typischen " Zusammenfassungsformaten"		●	●	●	
	Anbindung an die Assoziation zu management implementation.	●		●	●	
Dynamik	Rechtzeitige Bereitstellung der Verarbeitungsobjekte					●
	explizite Assoziationen zwischen Verarbeitungsobjekten			●		●
	Möglichkeiten zur Übertragung von Objekten			●		
	Attributbelegungen der Datentransportklassen bei Instanzierung			●	●	
	Keinerlei Zeitsynchronisation			●		
	Puffermöglichkeiten bei Empfängern			●		

Abbildung 4.8.: Abbildung der Spezifikationsrichtlinien (Zeilen) auf die Bestandteile des Automatisierungsprozesses (Spalten)

Auflistung der relevanten Primitiven eines Dienstes automatisch entsprechende Ableitungen der Klasse *Primitive* erzeugen.

b) Parameter der jeweiligen Primitive als Attribute

Auf Basis der Definition einer Dienstprimitive in der Entwicklersicht kann vom Generator die Attributliste für eine Verfeinerung der Klasse *Primitive* automatisch erzeugt werden.

c) Assoziation explizit mit verfeinern

Um auch in den erzeugten Unterklassen die Assoziationen der Oberklassen explizit etablieren zu können, werden durch Modellergänzungen bereitgestellte Qualifierattribute entsprechend belegt. Dazu muss in der Entwicklersicht festgelegt werden, welche Ereignisgeneratoren die jeweilige Primitive als „Eingabe" verwenden. Vom Generator kann dann ein Konstruktor erzeugt werden, der automatisch die konsistente Belegung der Qualifier in den Verfeinerungen von *Primitive* sicherstellt (siehe auch 9d).

d) *Primitive* ist abstrakte Oberklasse

Ähnlich wie in 1d wird auch die Klasse *Primitive* nach dem bisher definierten Vorgehen nie explizit instanziert, bleibt also automatisch abstrakte Oberklasse.

3. Ereignisgenerator

a) Für jedes benötigte Dienstgüteereignis eine Unterklasse von *Ereignisgenerator* bilden (überladen)

Werden die für die Spezifikation eines Dienstgütemerkmals notwendigen Dienstgüteereignisse in der Entwicklersicht explizit benannt, kann der Generator automatisch die entsprechenden Verfeinerungen der Klasse *Ereignisgenerator* erzeugen.

b) Methode *prüfeAufruf* überladen

Nach den Festlegungen in 4.2.3 wird ein Dienstgüteereignis nur vom Ereignisgenerator erzeugt, wenn die eingegangenen Primitiven bestimmte Kriterien erfüllen. Die Überprüfung dieser Kriterien erfolgt in der Methode *prüfeAufruf*. Das Gerüst dieser Methode ist bereits im Modell des allgemeinen Messprozesses festgelegt. Wird in der Entwicklersicht festgelegt, nach welchen Kriterien die Filterung erfolgen soll, kann die Methode *prüfeAufruf* durch den Generator vollständig erzeugt werden.

c) Methode *generiereEreignis* überladen

Eine Methode zum Generieren eines speziellen Dienstgüteereignisses ist automatisch erzeugbar, sobald der Name des zu generierenden Dienstgüteereignisses in der Entwicklersicht festgelegt wurde, wie in 3a gefordert, und eine entsprechende Verfeinerung des Modells existiert, die ebenfalls generiert werden kann (siehe 4a).

d) Assoziation explizit mit verfeinern

Wie bereits bei (1c) angewandt, geschieht die explizite Verfeinerung der Assoziationen einerseits durch die Bereitstellung geeigneter Qualifierattribute in Modellergänzungen und andererseits durch eine vom Generator entsprechend erzeugte Instanzierungshilfestellung. Um

diese Assoziation etablieren zu können, muss in der Entwicklersicht festgelegt werden, auf welche Primitive (bzw. Unterklasse davon) ein Dienstgüteereignis und damit der zugehörige Ereignisgenerator aufbaut. Diese Informationen werden dann von der Instanzierungshilfestellung verwendet, um einen konsistenten Aufbau der Messsicht sicherzustellen.

4. Dienstgüteereignis

   a) Für jeden Typ eines Dienstgüteereignisses explizite Unterklasse von *Dienstgüteereignis* bilden (konzeptionelle Vererbung)

      Durch die bereits getroffene Festlegung in der Entwicklersicht über die zur Definition eines Dienstgütemerkmals notwendigen Dienstgüteereignisse mit ihrem Namen zusammen mit ihrer Ableitung von einer Primitive, kann vom Generator automatisch eine entsprechende Unterklasse von *Dienstgüteereignis* generiert und in die Definitionssicht eingebunden werden.

      In der zugehörigen Verfeinerung von *Ereignisgenerator* wird nach dem bisher festgelegten Vorgehen eine Methode zur Erzeugung von Instanzen dieser Ereignisunterklasse definiert. Weiterer Instanzierungshilfestellungen bedarf es damit nicht.

   b) Assoziation zwischen verfeinerten Klassen explizit erneut etablieren

      Ähnlich dem Vorgehen in 1c und 2c müssen Modellergänzungen Qualifier bieten, um die Assoziationen etablieren zu können. Eine korrekte Belegung dieser Attribute kann durch einen entsprechend generierten Default-Konstruktor (siehe 9d) unter Verwendung der Festlegungen in der Entwicklersicht (siehe 3d) sichergestellt werden.

5. Ereigniskorrelation

   a) Für jede Definition eines Dienstgütemerkmals genau eine Verfeinerung von *Ereigniskorrelation* bilden (überladen)

      Eine Instanz einer Verfeinerung von *Ereigniskorrelation* liefert eine Instanz der Klasse *Messwert* (respektive eine ihrer Verfeinerungen). Wird in der Entwicklersicht der Messwert festgelegt, auf dem die Definition eines Dienstgütemerkmals aufbaut, dann lässt sich aus dieser Festlegung automatisch eine entsprechende Verfeinerung der Klasse *Ereigniskorrelation* ableiten und in die Definitionssicht integrieren.

      Die Instanzierungshilfestellung muss vom Generator so erzeugt werden, dass beim Aufbau der Messsicht genau eine Instanz der entsprechenden Unterklasse von *Ereigniskorrelation* aufgebaut wird und die Assoziationen entsprechend den Modellvorgaben eingehalten werden (siehe auch 5d).

   b) Korrelationsmethode überladen

      Die Funktionalität eines Ereigniskorrelators wird maßgeblich durch die verwendete Korrelationssmethode bestimmt. Diese ist zwar durch das Modell des allgemeinen Messprozesses in ihrer Schnittstelle festgelegt, ihre tatsächliche Funktionalität muss aber vom Entwickler in der Entwicklersicht festgelegt werden,

da eben diese Funktionalität zu einem großen Teil die Definition eines Dienstgütemerkmals bestimmt.

c) Repository für Korrelationsmethoden bereitstellen

Vorgefertigte Korrelationsfunktionen zur Verwendung in der Korrelationsmethode können in Modellergänzungen definiert werden. Zusätzlich müssen in der Entwicklersicht passende Konstrukte bereitgestellt werden, um diese Funktionsbausteine referenzieren und damit letztlich die Funktionalität der Korrelationsmethode festlegen zu können.

d) Assoziationen explizit mit verfeinern

Wie schon für 1c und 3d werden auch in diesem Fall Qualifierattribute in Modellergänzungen festgelegt und die vom Generator erzeugte Instanzierungshilfestellung so angepasst, dass diese Attribute entsprechend der Festlegungen in der Entwicklersicht belegt werden. Dort muss dementsprechend festgelegt sein, welche Ereignisse der Ereigniskorrelator benötigt. Wie in 5a beschrieben, wird vom Generator eine Instanzierungshilfestellung erstellt. Diese muss auch die korrekte Belegung der Qualifier-Attribute sicherstellen.

6. Messwert

a) Für jeden aus einer speziellen Ereigniskorrelation resultierenden Messwert eine Verfeinerung bilden (konzeptionelle Vererbung)

In 5a wurde bereits festgelegt, den für ein Dienstgütemerkmal notwendigen Messwert explizit in der Entwicklersicht festzulegen. Auf Basis dieser Festlegung lässt sich vom Generator einfach eine Verfeinerung der Klasse *Messwert* aus dem Modell des allgemeinen Messprozesses bilden. Da diese Verfeinerung konzeptionell ist, genügt es, in der Entwicklersicht lediglich den Namen der jeweiligen Abtastung festzulegen.

b) Assoziationen explizit mit verfeinern

Die Klasse *Messwert* und damit auch alle ihre Verfeinerungen sind im Modell des allgemeinen Messprozesses als Datenaustauschklassen festgelegt. Um die Assoziationen dieser Klasse auch bei der Verfeinerung zu erhalten, müssen durch Modellergänzungen geeignete Qualifierattribute bereitgestellt werden. Die korrekte Belegung dieser Attribute muss aber bei jeder Instanzierung möglich sein, kann also nur durch die jeweils verwendete Verfeinerung der Klasse *Ereigniskorrelation* erfolgen. In der Entwicklersicht muss demnach auch festgelegt werden, in welcher Weise (durch welche Instanz der Klasse *Nachverarbeitung*) ein Messwert weiterverarbeitet wird und aus welchem Ereigniskorrelator er resultiert.

7. Nachverarbeitung

a) Für jedes definierte Dienstgütemerkmal eine Verfeinerung von *Nachverarbeitung* bilden (überladen)

Um eine Verfeinerung der Klasse *Nachverarbeitung* generieren zu können, muss in der Entwicklersicht festgelegt werden, welche

Abtastung durch welche Art der Nachverarbeitung zur (gemessenen) Dienstgüte aggregiert werden soll (siehe auch 7c).

b) Verarbeitungsmethoden überladen

Aus der in der Entwicklersicht festgelegten Art der Nachverarbeitung kann vom Generator eine entsprechende Verarbeitungsmethode generiert werden. Durch ein vorhandenes Repository an Verarbeitungsfunktionen (siehe 7c) wird die Definition dieser Verarbeitungsmethode erheblich erleichtert.

c) Repository für Verarbeitungsmethoden bereitstellen

Durch Modellergänzungen können vorgefertigte Verarbeitungsfunktionen als Bausteine zur Verfügung gestellt werden. Um ihre Verwendbarkeit zu erleichtern, müssen in der Entwicklersicht Konstrukte bereitstehen, um diese Bausteine referenzieren zu können.

d) Assoziation an die Klasse *management implementation* (aus dem MNM-Dienstmodell) erhalten

Durch Modellergänzungen kann die Klasse *Nachverarbeitung* und damit ihre Verfeinerungen mit Qualifierattributen ausgestattet werden, die eine Abbildung der Assoziationen gewährleisten. Für die Klasse *management implementation* aus dem MNM-Dienstmodell ist dieses Verfahren nicht anwendbar. Letztendlich muss die Anbindung hier „von Hand" durch den Entwickler geschehen.

8. gemessene Dienstgüte

a) Für jedes definierte Dienstgütemerkmal eine Verfeinerung von *gemessene Dienstgüte* bilden (konzeptionelle Vererbung)

Die Art der gemessenen Dienstgüte und damit auch die Verfeinerung der Klasse *gemessene Dienstgüte* ergibt sich aus der gewählten Nachverarbeitungsmethode. Damit kann vom Generator aus den bereits in der Entwicklersicht getroffenen Festlegungen eine entsprechende Verfeinerung von *gemessene Dienstgüte* erzeugt werden.

b) zusätzliche „Vererbungsschicht" mit typischen „Zusammenfassungsformaten" (Verteilungsfunktionen etc.) aufbauen

Werden durch Modellergänzungen analog zum in 7c beschriebenen Repository von Nachverarbeitungsmethoden typische Arten/Formate für die Darstellung der gemessenen Dienstgüte festgelegt, können die Ableitungen von spezifischen Klassen aus diesen Ergänzungen ohne weitere Eingriffe des Entwicklers erfolgen, da sich diese Ableitungen aus den in 7c bereitgestellten Nachverarbeitungsmethoden ergeben.

c) Anbindung an die Assoziation zu *management implementation*.

Im Modell des allgemeinen Messprozesses ist *gemessene Dienstgüte* eine Datenübertragungsklasse, die mit jeder Datenübertragung erneut instanziert wird. Werden durch Modellergänzungen geeignete Qualifierattribute bereitgestellt, kann deren korrekte Belegung durch einen

generierten Konstruktor übernommen werden.

9. Dynamik

   a) Rechtzeitige Bereitstellung der Verarbeitungsobjekte

      Bei der Erzeugung der Messsicht muss die vom Generator erzeugte Instanzierungshilfestellung sicherstellen, dass zu Beginn einer Messung alle notwendigen Verarbeitungsinstanzen existieren.

   b) Explizite Assoziationen zwischen Verarbeitungsobjekten

      Die nach den Vorgaben der Entwicklersicht erzeugte Instanzierungshilfestellung sorgt damit auch dafür, dass die einzelnen Verarbeitungsobjekte entsprechend der festgelegten Assoziationen aneinander gebunden sind. Sie verwendet dazu Festlegungen aus der Entwicklersicht.

   c) Möglichkeiten zur Übertragung von Objekten

      Das Modell des allgemeinen Messprozesses nimmt explizit die Unterscheidung zwischen Datentransport- und Datenverarbeitungsklassen vor. Die Übertragung von Instanzen der Datentransportklassen muss durch Modellergänzungen nach einem geeigneten Designpattern unterstützt werden.

   d) Attributbelegungen der Datentransportklassen bei Instanzierung

      Korrekte Attributbelegungen der Datentransportklassen (insbesonder für Qualifierattribute) können am einfachsten direkt bei der Instanzierung dieser Klassen umgesetzt werden. Dazu werden zweckmäßigerweise vom Generator Konstruktoren erzeugt, die eine konsistente Attributbelegung (z.B. automatische Belegung des Zeitstempels) sicherstellen.

   e) Keinerlei Zeitsynchronisation

      Das Designpattern, das eine Übertragung von Instanzen der Datentransportklassen ermöglicht, muss so gewählt werden, dass es ohne Zeitsynchronisation auskommt. Zudem dürfen Korrelationsmechanismen zur zeitlichen Ordnung von Ereignissen lediglich deren Zeitstempel verwenden.

   f) Puffermöglichkeiten bei Empfängern

      Die fehlende Zeitsynchronisation bedeutet auch, dass im Designpattern, das die Übertragung ermöglichen soll, Puffer auf der Empfängerseite eingeplant werden müssen.

Beim Abbilden der Spezifikationsanforderungen, wie sie in Abschnitt 4.2.4 entwickelt wurden, auf die Bestandteile des vorgeschlagenen Automatisierungsprozesses hat sich dieser Prozess als tragfähig erwiesen, da sich alle Anforderungen abbilden lassen, wenn davon ausgegangen wird, dass besonders die implementierungsspezifischen Arbeiten des Entwicklers, wie die Anbindung an einen vorhandenen SAP, nicht vollständig automatisierbar sind und weiterhin „per Hand" erfolgen müssen.

4.3.2.3. Design des Automatisierungsprozesses

Im vorherigen Abschnitt wurde gezeigt, in welchen Teilen des Automatisierungsprozesses sich die aufgestellten Spezifikationsanforderungen umsetzen lassen und welche Anforderungen dabei an die einzelnen Bausteine des Automatisierungsprozesses gestellt werden. Damit wurde auch die prinzipielle Erfüllbarkeit dieser Anforderungen gezeigt. Eine konkrete Implementierung des gesamten Prozesses ist in Abschnitt 4.3.3 beschrieben. Dieser Abschnitt zeigt zunächst, aufbauend auf den bisherigen Überlegungen, ein Design des Automatisierungsprozesses als Ganzes, das dann, gegliedert nach den in 4.3.2.1 vorgestellten und in 4.3.2.2 ergänzten Bausteinen resp. Teilverarbeitungsschritten, im Detail beschrieben wird. Bei der Beschreibung der jeweiligen Designentscheidung wird in Klammern auf die entsprechenden Anforderungen aus dem vorhergehenden Abschnitt 4.3.2.2 verwiesen.

4.3.2.3.1. Automatisierungsprozess im Überblick Die prägende Entscheidung beim Design des Automatisierungsprozesses nach dem bereits vorgeschlagenen Muster besteht darin, die Funktionalität des Generators durch einen Compiler zu realisieren, der als Eingabesprache eine geeignete Formalisierung der Entwicklersicht verwendet. Als Ausgabe wird eine Verfeinerung der Definitionssicht nach den Vorgaben der Entwicklersicht erzeugt. Damit daraus auch die entsprechende Messsicht aufgebaut werden kann, wird vom Compiler ebenfalls eine Verfeinerung der Instanzierungshilfestellung erzeugt. Mit deren Hilfe kann der Entwickler eine Messsicht erzeugen, wenn er zusätzlich die notwendigen Instanzen der SAP-Ankopplung bereitstellt. Abbildung 4.9 zeigt dieses Design.

Die Abbildung zeigt links oben den Entwickler, der ein Dienstgütemerkmal in der Entwicklersicht spezifiziert. Diese Festlegung dient mit ihren einzelnen Anweisungen dem Compiler als Eingabe. Dieser generiert, gesteuert durch geeignete Produktionsregeln (in der Abbildung dargestellt durch ::=), eine Verfeinerung der Definitionssicht und eine Verfeinerung der Instanzierungshilfestellung.

Als Basis für die Verfeinerung der Definitionssicht dient das erweiterte Modell des allgemeinen Messprozesses, wie es in Unterabschnitt 4.3.2.3.3 entwickelt wird. Die für das jeweilige Diestgütemerkmal spezifische Instanzierungshilfestellung wird aus dem Basismodell der Instanzierungshilfestellung, wie es in Unterabschnitt 4.3.2.3.4 entwickelt wird, erzeugt.

Mit Hilfe dieser spezifischen Instanzierungshilfestellung kann automatisch die Messsicht für ein Dienstgütemerkmal erzeugt werden, wenn vom Entwickler die dafür notwendigen Instanzen der Klasse *SAP-Ankopplung* resp. einer Verfeinerung angegeben werden.

Das Design der Entwicklersicht, also der dort verwendeten Sprachelemente, beeinflusst entscheidend das Design der Produktionsregeln für den Compiler, die zusätzlich durch die Festlegungen der Modelle des erweiterten allgemeinen Messprozesses und der Basis der Instanzierungshilfestellung beeinflusst werden. Die folgende Darstellung beschreibt deshalb zunächst das Design der Entwicklersicht (Abschnitt 4.3.2.3.2) und der beiden Modelle (Abschnitt 4.3.2.3.3 und Abschnitt 4.3.2.3.4). Darauf aufbauend werden die Produktionsregeln für den Compiler entworfen (Abschnitt 4.3.2.3.5). Abschließend wird betrachtet, welche Aktionen weiterhin vom Entwickler ausgeführt werden müssen, weil sie sich durch das vorgeschlagene Schema nicht automatisieren lassen (Abschnitt 4.3.2.3.6). In Abschnitt 4.3.3 wird das vorgestellte Design prototypisch implementiert. Die dabei resultierende formale

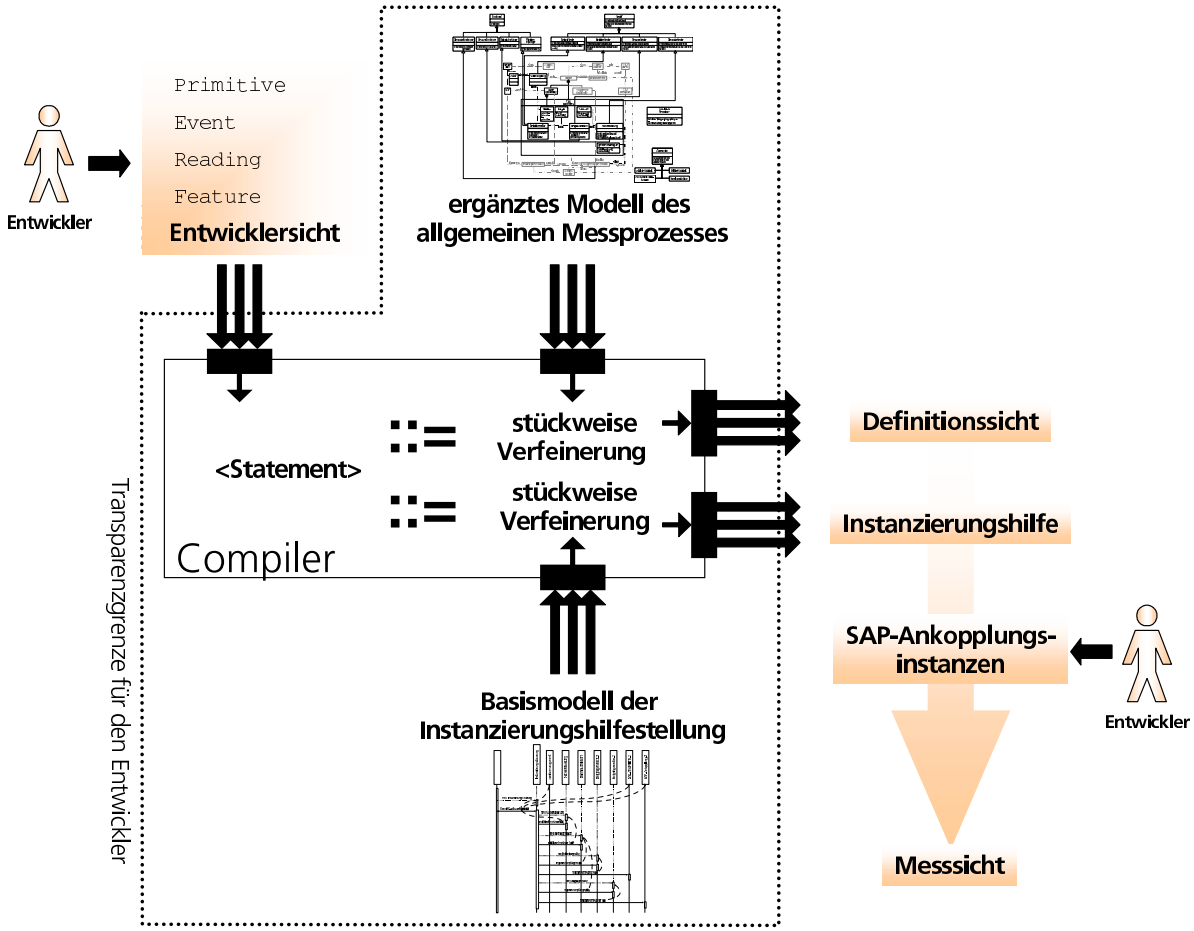

Abbildung 4.9.: Design des Automatisierungsprozesses

Sprache zur Festlegung der Entwicklersicht wird im Anschluss an das Design des Automatisierungsprozesses in Abschnitt 4.3.3.4 vorgestellt.

4.3.2.3.2. Entwicklersicht Die Entwicklersicht soll einerseits beim Entwickler implizit vorhandenes Wissen bündeln und andererseits als Eingabe für den Compiler verwendet werden können. Damit bietet es sich an, die Entwicklersicht durch programmiersprachenähnliche Konstrukte festzulegen. Somit kann letztlich eine formale Sprache festgelegt werden, die vom Compiler übersetzt wird. Aus den Anforderungen an die Entwicklersicht, wie sie im vorherigen Abschnitt 4.3.2.2 beschrieben sind, ergeben sich die in der Sprache notwendigen Konstrukte, die im Folgenden informell beschrieben werden. Grundidee beim Design der Sprachelemente ist es, lediglich Konstrukte zu schaffen, aus denen sich direkt die notwendigen Datentransportklassen ableiten und notwendig Bezüge zwischen diesen festlegen lassen. Damit sind auch die Verarbeitungsklassen - implizit - festgelegt und können entsprechend vom Compiler erstellt werden. Die Definition einer konkreten Sprache auf Basis dieser Festlegungen wird in Abschnitt 4.3.3.4 vorgestellt.

1. Konstrukt zur Festlegung von Primitiven

   In einer deklarativen Anweisung werden Primitiven mit ihren Attributen festgelegt. Neben dem Namen der jeweiligen Primitive werden dort alle ihre Attribute mit Namen und Typen festgelegt. (2a, 2b)

2. deklaratives Typenkonzept

   Um die Festlegung von Datentypen zu ermöglichen, stehen einfache Datentypen durch entsprechende Deklarationen zur Verfügung.

3. Konstrukt zur Festlegung von Dienstgüteereignissen

   Mit einer Deklaration kann ein Dienstgüteereignis mit seinem Namen (3a, 3c) und der für seine Generierung notwendigen Primitive (2c) festgelegt werden. Dabei muss auch bestimmt werden, welche SAP-Ankopplungsinstanz die jeweilige Primitive liefert. Um eine sinnvolle Entkopplung der Definitionssicht von der Messsicht zu erlauben, werden diese Instanzen in der Entwicklersicht nicht explizit angegeben. Um deutlich zu machen, dass unterschiedliche Primitiven von unterschiedlichen Ankopplungsinstanzen stammen, werden diese symbolisch, etwa mit Nummern, referenziert. Innerhalb dieses Konstrukts wird auch definiert, in welcher Weise eingehende Primitiven gefiltert werden und wie aus den Attributen der Primitive der eindeutige Identifikator des resultierenden Ereignisses berechnet wird. (3b)

4. Konstrukt zur Festlegung von Messwerten

   Der Name eines Messwerts (6a) wird zusammen mit den für seine Berechnung notwendigen Ereignissen (3d, 4b,6b) in einer Deklaration angegeben. Dabei wird festgelegt, in welcher Weise die Korrelation (5b) der Ereignisse und damit die Berechnung eines einzelnen Messwertes erfolgt.

5. Schlüsselwörter zur Referenzierung von Korrelationsfunktionen

   Elemente aus dem Repository (5c) für Korrelation sfunktionen werden durch entsprechende Schlüsselwörter mit entsprechender Parameterleiste referenziert. Korrelation sfunktionen sind dabei immer 2-stellig, setzen also immer genau zwei Ereignisse in Bezug und liefern als Ergebnis entweder ein Ereignis oder einen Zahlenwert. Die Funktionen lassen sich beliebig schachteln.

6. Konstrukt zur Beschreibung der gemessenen Dienstgüte

   Ein weiteres Konstrukt legt den Namen der gemessenen Dienstgüte und damit des Dienstgütemerkmals an sich fest (8a). Innerhalb dieses Konstrukts wird definiert, welcher Messwert (5a, 6b) in welcher Weise statistisch nachverarbeitet wird (7a, 7b), um die gemessene Dienstgüte zu ermitteln.

7. Schlüsselwörter zur Referenzierung von Nachverarbeitungsfunktionen

   Zur Festlegung der Nachverarbeitungsfunktionen stehen entsprechende Schlüsselwörter bereit, so dass alle in den Modellergänzungen festgelegten Nachverarbeitungsmethoden (7c) und die zugehörigen Ergebnisformate (8b) referenziert werden können.

4.3.2.3.3. Modellergänzungen Die geforderten Modellergänzungen sollen einerseits so einfach wie möglich sein und andererseits noch deutlich vom Basismodell abgegrenzt sein, so dass ein klares Design erhalten bleibt. Es bietet sich an, die geforderten Modellergänzungen durch eine Verfeinerung des bestehenden Modells des allgemeinen Messprozesses umzusetzen. Alle weiteren Ableitungen für die Spezifikation eines Dienstgütemerkmals durch den Compiler werden dann auf Basis dieses erweiterten Modells durchgeführt. Abbildung 4.12 zeigt dieses erweiterte

Modell, das im Folgenden in seinen einzelnen Erweiterungen beschrieben und somit sukzessive aufgebaut wird.

- Basisklassen zur Bereitstellung von Übertragungsmechanismen zwischen den Verarbeitungsklassen

  Das bisherige Modell des allgemeinen Messprozesses zeigt den Austausch der sog. Datenübertragungsklassen zwischen den Verarbeitungsklassen nur konzeptionell durch die Darstellung letzterer als Assoziationsklassen. Die hier vorgenommenen Modellergänzungen bilden eine Vorlage zur Implementierung des durch die Spezifikationsrichtlinien geforderten Mechanismus zur Datenübertragung (9c).

  Grundlage des hier umgesetzten Designs ist ein einfaches Sender / Empfänger-Muster: Empfänger können sich beim Sender registrieren. Dieser verwaltet eine entsprechende Liste aller registrierten Empfänger. Wird die Methode *sende* des Senders aufgerufen, so ruft dieser von allen in der Empfängerliste gespeicherten Objekten die Methode *empfange* auf und übergibt dabei das zu übertragende Objekt.

  Dieses Grundmodell wird verwendet, um für jede Datenübertragungsklasse ein entsprechendes Sender/Empfänger-Paar abzuleiten. Innerhalb eines jeweiligen Paares besteht eine Assoziation zwischen Sender und Empfänger. Das Attribut *empfängerListe* dient jeweils als Qualifier für diese Assoziation, erlaubt also die Identifizierung von durch die Assoziation verbundenen Objekten. Abbildung 4.10 zeigt diese Erweiterungen zusammen mit den weiteren hier vorgestellten Modellergänzungen. Zur besseren Übersicht ist das Modell des allgemeinen Messprozesses nur durch ein graues Rechteck wiedergegeben. Die Abbildungen 4.11 und 4.12 zeigen schrittweise das vollständige Modell, in dem die vorhandenen Verarbeitungsklassen dann von den entsprechenden Sender- bzw. Empfängerklassen abgeleitet werden.

  Die in den Senderklassen eingeführten Qualifier (Empfängerlisten) genügen auch, um die Assoziationen zwischen den eigentlichen Verarbeitungsklassen zu qualifizieren. Die Datenübertragungsklassen werden durch die Übergabe in der Methode *empfange* implizit zu Assoziationsklassen. Der in den Spezifikationsrichtlinien geforderte Erhalt der Assoziationen (1c, 2c, 3d,3d, 4b, 5d, 6b, 7d, 8c, 9b) ist damit für alle Verarbeitungs- und Datenübertragungsklassen, und damit auch für etwaige Ableitungen davon, automatisch gegeben. Lediglich die Assoziation an die Klasse *management implementation* muss eventuell durch „Handarbeit" des Entwicklers (Abschnitt 4.3.2.3.6) angepasst werden.

  Die von den Spezifikationsrichtlinien geforderten Puffermöglichkeiten in den Empfänger-Klassen (9f) werden in Abschnitt 4.3.3 beschrieben und implementiert, wenn eine durchgängige Implementierung des gesamten Automatisierungsprozesses vorgestellt wird.

- Repository für Korrelationsmethoden

  In der Klasse *KorrelationsMethoden* werden unterschiedlichste Korrelationsmethoden, wie in den Spezifikationsrichtlinien (5c) gefordert, implementiert. Die Klasse *Ereigniskorrelation* erbt von dieser Klasse, so dass alle diese Methoden in der Klasse *Ereigniskorrelation* und in ihren Ableitungen zur Verfügung

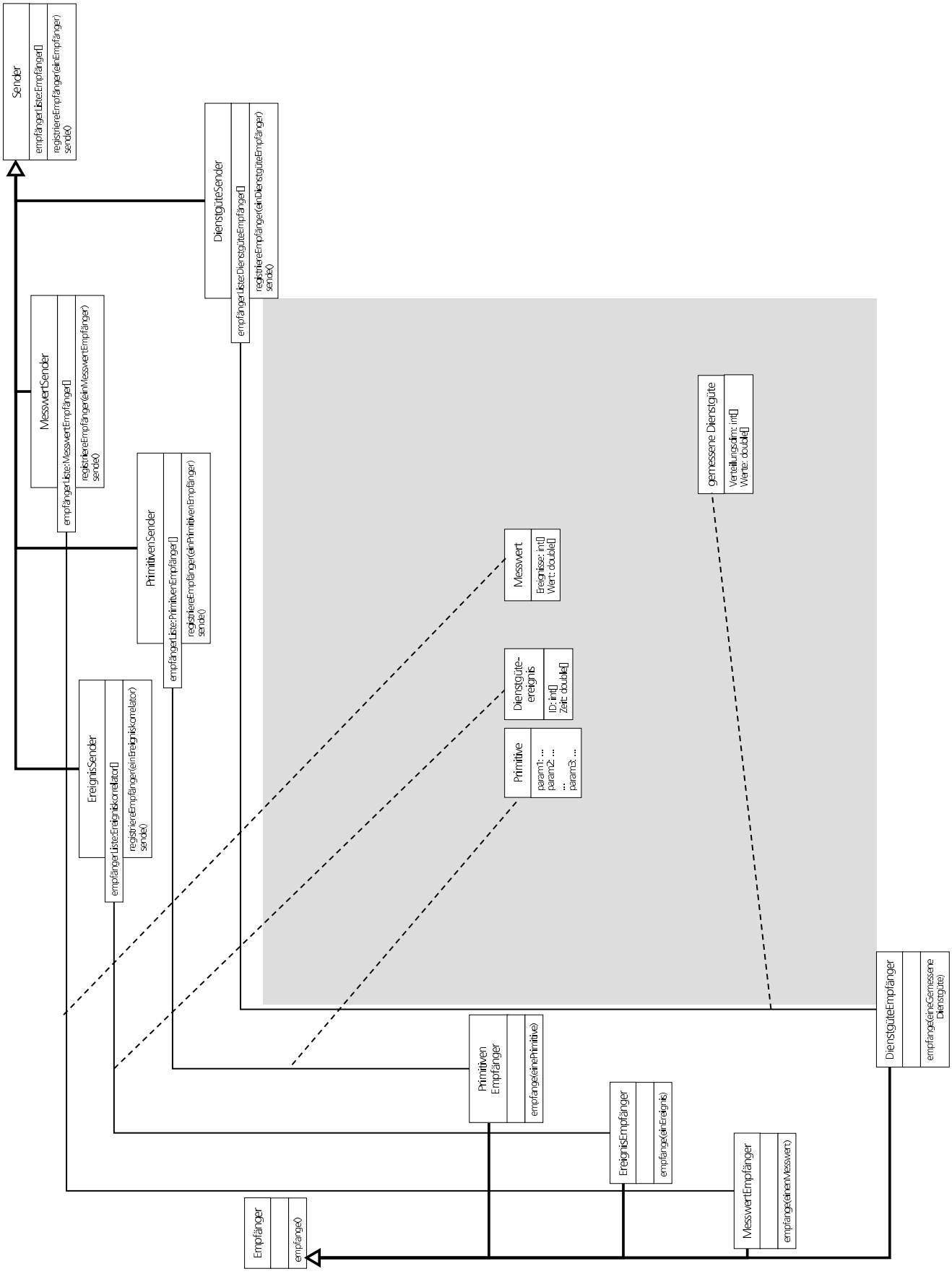

Abbildung 4.10.: Modellergänzungen für die Übertragung der Datentransportklassen

stehen. Diese Korrelationsmethoden sind entsprechend den Festlegungen in der Entwicklersicht benannt und richten sich in ihrer Signatur ebenfalls nach den dortigen Vorgaben: Die Methoden nehmen als Eingabe immer zwei Ereignisse entgegen und liefern entweder ein Ereignis oder einen Wert zurück. Eine Implementierung dieser Methoden und damit eine Festlegung des Funktionsumfangs des Korrelationsmechanismus geschieht in Abschnitt 4.3.3. Dabei wird auch auf die geforderte Unabhängigkeit des Korrelationsmechanismus von einer globalen Uhr (9e) eingegangen. Abbildung 4.10 zeigt am rechten Bildrand die zusätzlich eingeführte Klasse *KorrelationsRepository* mit beispielhaft aufgelisteten Methoden. Die vollständige Einbindung dieser Klasse in das vorhandene Modell zeigt Abbildung 4.12.

- Repository für Nachverarbeitungsmethoden und Formate für gemessene Dienstgüte

  Um den Aufbau der Nachverarbeitungsprozeduren zu vereinfachen und zu vereinheitlichen, wurde ein Repository von Nachverarbeitungsmethoden zusammen mit zugehörigen Darstellungsklassen gefordert (7c, 8b). Nachverarbeitungsmethoden lassen sich auf Grund der vielfältigen Möglichkeiten der gewählten Ergebnisformen in ihrer Signatur praktisch nicht vereinheitlichen. Mit der zusätzlichen Forderung nach entsprechenden Klassen zur Darstellung der Ergebnisse bietet es sich an, die Aggregationsmethoden an diese Darstellungsklassen zu binden und hinter einem einheitlichen Interface zu verschatten.

  Dazu wird die Klasse *Aggregation* eingeführt. Sie bietet Methoden zum Einfügen und Entfernen eines Wertes aus der Aggregation sowie eine Ausgabemethode. Von dieser Klasse werden spezifische Verteilungen abgeleitet und die Zugriffsmethoden entsprechend überladen. Eine spezifische Verfeinerung von *gemessene-Dienstgüte*, die für die Spezifikation eines bestimmten Dienstgütemerkmals vom Compiler aufgebaut wird, erbt dann sowohl von einer Verfeinerung von *Aggregation* (je nach Vorgabe in der Entwicklersicht) als auch von der Klasse *gemesseneDienstgüte* selbst. Die Methode *aggregiere* kann somi die standardisierten Zugriffsmethoden der gewählten Aggregation verwenden, um die gewählte Verfeinerung von *gemesseneDienstgüte* entsprechend zu belegen.

- Konstruktoren für Datentransportklassen

  Die Spezifikationsrichtlinien fordern eine konsistente Belegung der Attribute der Datentransportklassen (9d) bereits bei ihrer Instanzierung. Dazu werden die Konstruktoren der Klassen so implementiert, dass eine Instanz der jeweiligen Klasse zumindest initialisierte Attribute aufweist. Das eingeführte Sender/Empfänger-Muster macht Attribute zur Qualifizierung von Assoziationen in den Datenübertragungsklassen überflüssig, da deren Bindung an den jeweiligen Sender bzw. Empfänger implizit durch den Aufruf der Methode *empfange* gegeben ist. Damit entfällt auch eine korrekte Belegung dieser qualifizierenden Attribute, wie ursprünglich gefordert.

Die nachfolgende Abbildung 4.12 zeigt nochmals das gesamte Modell, wie es sich nach den vorgenommen Erweiterungen ergibt. Hellgrau unterlegt ist das bisher

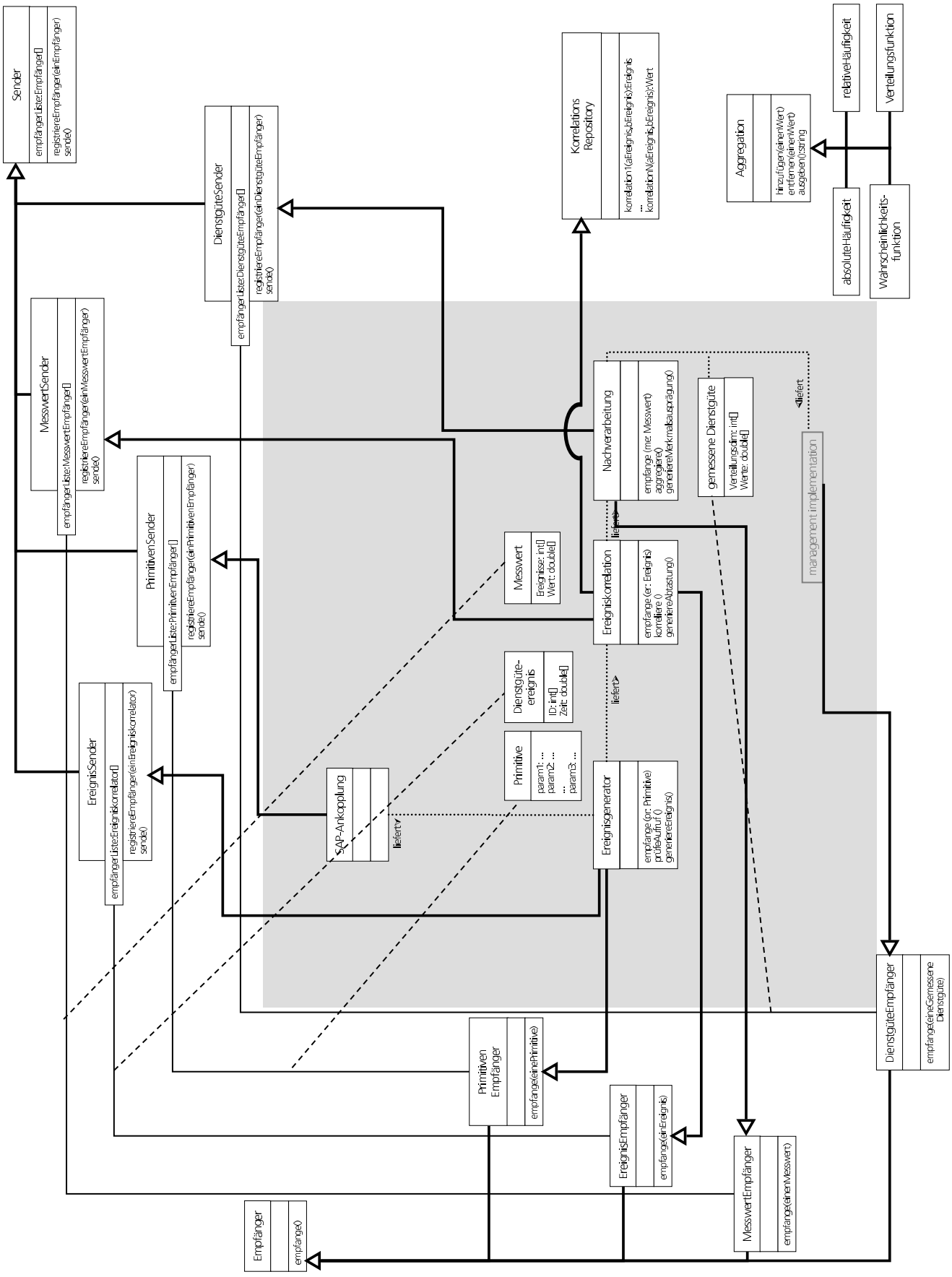

Abbildung 4.11.: Modellergänzungen zur Bereitstellung der Repositories (zusammen mit der vollständigen Modellierung der Datenübertragung)

entwickelte Modell des allgemeinen Messprozesses. Darin eingebettet ist die Klasse *QoS-Parameters* dunkelgrau dargestellt. Zur besseren Übersicht sind neu eingeführte Vererbungsbeziehungen ebenso wie neu eingeführte Assoziationen hervorgehoben. Am rechten Bildrand ist dort bereits die Klasse *Instanzierungssteuerung* eingetragen, welche die Funktion der Instanzierungshilfestellung übernehmen wird. Die Beschreibung der durch diese Klasse realisierten Funktionalitäten erfolgt in Abschnitt 4.3.2.3.4 im Rahmen der Beschreibung des Designs der Instanzierungshilfestellung.

4.3.2.3.4. Instanzierungshilfestellung

Die Instanzierungshilfestellung übernimmt die Aufgabe, die Messsicht für ein spezifisches Dienstgütemerkmal automatisch aus der vom Compiler generierten Definitionssicht zu erzeugen. In Anlehnung an das in [GHJV 95] vorgestellte *Builder-Pattern* übernimmt ein einziges Objekt aus einer Verfeinerung der Klasse *Instanzierungssteuerung*, die vom Compiler für jedes spezifizierte Dienstgütemerkmal erzeugt wird, den Aufbau der Messsicht. Die Basisklasse *Instanzierungssteuerung* wird entsprechend als Modellergänzung aufgenommen, wie Abbildung 4.12 im vorherigen Abschnitt bereits 4.3.2.3.3 zeigt.

Das grundsätzliche Vorgehen für den Aufbau der Messsicht veranschaulicht beispielhaft Abbildung 4.13. Das Beispiel geht davon aus, dass zur Messung des Dienstgütemerkmals zwei Instanzen der *SAP-Ankopplung* benötigt werden (z.B. zur Messung des One-Way-Delays). Die Instanzierung der Messsicht kann von einem beliebigen Objekt angestoßen werden. Deshalb ist die auslösende Instanz am linken Rand von Abbildung 4.13 nicht bezeichnet.

Nach der Erzeugung eines Objekts der Klasse *Instanzierungssteuerung* kann die Methode *Messsicht_aufbauen* angestoßen werden, welche von dieser Klasse zur Verfügung gestellt wird. Diese Methode benötigt als Parameter Referenzen auf die Instanzen der Klasse *management implementation* und *SAP-Anbindung*. Die Anzahl der Aufrufparameter dieser Methode hängt somit vom jeweils spezifizierten Dienstgütemerkmal ab.

Durch die Möglichkeit, beim Aufbau der Messsicht direkt auf bestehende Instanzen zu verweisen, kann diese in beliebige bestehende Umgebungen eingebunden werden, sofern die jeweiligen Instanzen die Methoden ihrer zugehörigen Oberklassen, wie in der Beschreibung des erweiterten Modells des allgemeinen Messprozesses (siehe Abbildung 4.12) vorgegeben, implementieren.

Bei bestehenden Implementierungen kann auch eine Instanz einer entsprechenden Wraper-Klasse bereitgestellt werden, welche die erforderlichen Interfaces zur Verfügung stellt und die Anbindung an die bestehende Implementierung sicherstellt. Diese Aufgabe ist strikt implementierungsabhängig und kann deshalb, wie in Abschnitt 4.3.2.3.6 beschrieben, nicht automatisiert werden.

Die Instanzierungssteuerung baut die Messsicht entgegen dem Prozessablauf, ausgehend von der Instanz der Klasse *management implementation* auf. Dieses Vorgehen ist notwendig, damit die bereits aufgebauten Instanzen als Empfänger der jeweils vorgelagerten Verarbeitungsstufe registriert werden können. In Abbildung 4.13 ist durch gestrichelte Pfeile dargestellt, auf welche Instanzen sich die jeweils beim Methodenaufruf übergebenen Referenzen beziehen. Entsprechend zeigen die Pfeile von der Referenz, dargestellt durch ein graues Quadrat, auf die Instanz, die durch diese Referenz bezeichnet wird.

Die beim Aufruf der Methode *Messsicht_aufbauen* übergebene Referenz auf eine Instanz der Klasse *management implementation* ist notwendig, damit diese Instanz als Empfänger bei der Instanz

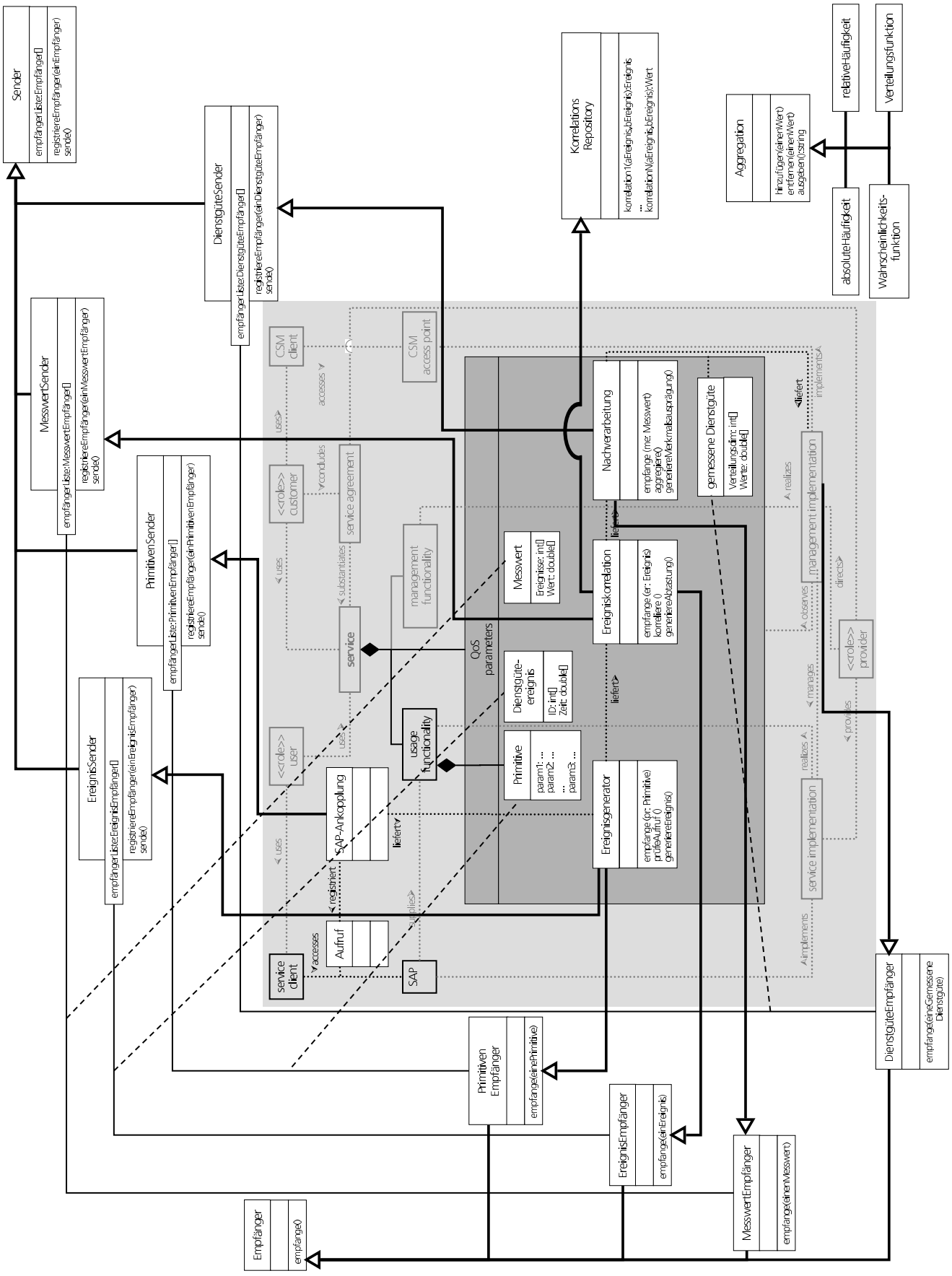

Abbildung 4.12.: erweitertes Modell des allgemeinen Messprozesses

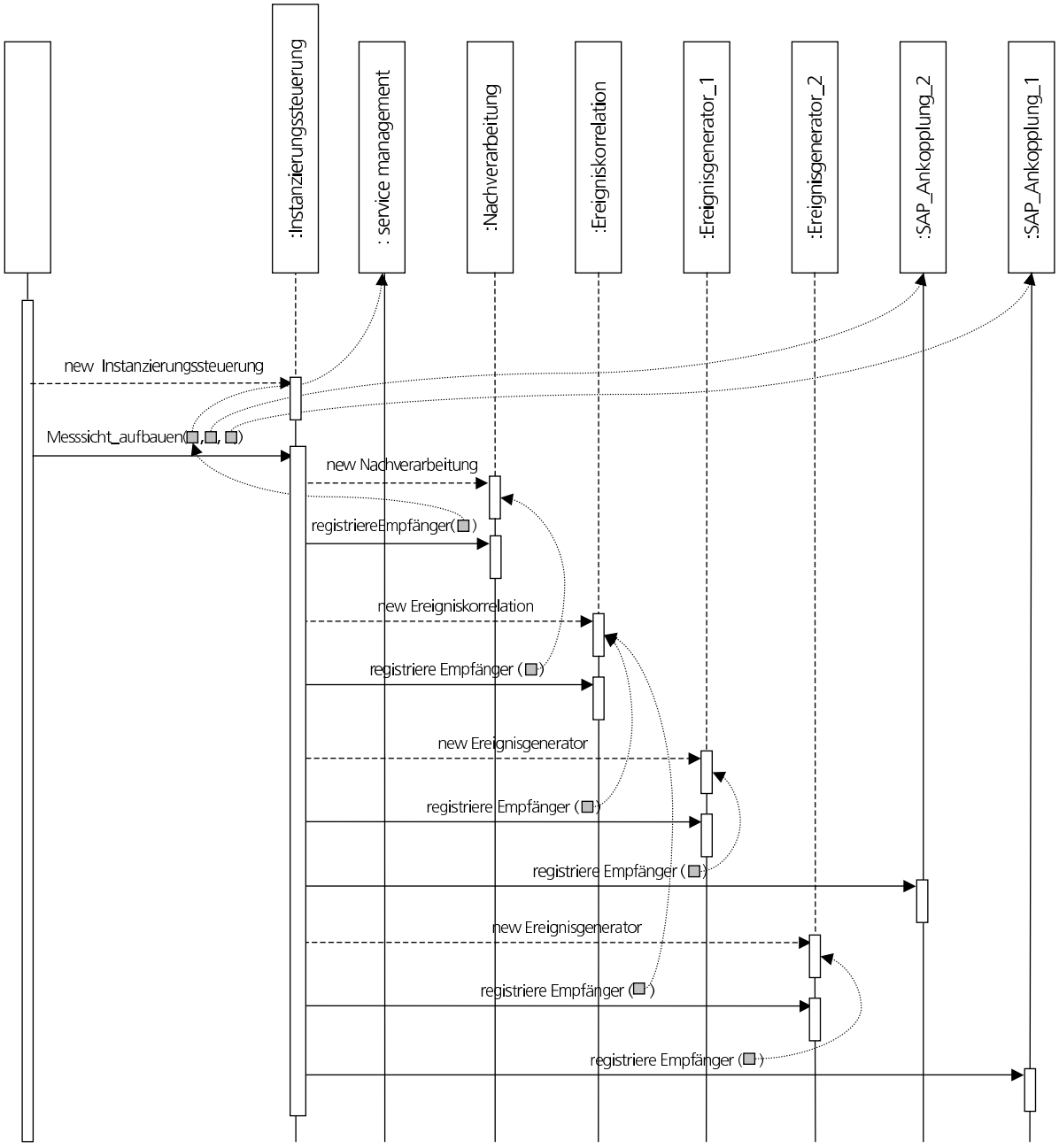

Abbildung 4.13.: Beispielhafter Ablauf beim Aufbau der Messsicht, dargestellt im UML-Sequenzdiagramm

:Nachverarbeitung eingetragen werden kann. Die Referenzen auf die Instanzen der Klasse *SAP-Ankopplung* werden benötigt, damit bei diesen Instanzen die Methode *registriere Empfänger* aufgerufen werden kann, um die zugehörigen Ereignisgeneratoren als Empfänger zu registrieren.

Durch die jeweilige Registrierung als Empfänger werden die Assoziationen zwischen den Instanzen implizit aufgebaut. Damit erfüllt das vorgestelte Design der Instanzierungshilfestellung die Anforderungen (1c, 3d, , 5d, 7d). Die Anbindung der Assoziationsklassen (*Primitive*, *Ereignis*, *Messwert* und *gemessene Dienstgüte*) erfolgt, wie bereits beim Design derModellergänzungen in Abschnitt 4.3.2.3.3 beschrieben, implizit durch die Übergabe der jeweiligen Instanzen als Parameter der Methode *empfange*. Beim Aufbau der Messsicht durch eine Instanz der Klasse *Instanzierungssteuerung* müssen damit

keine weiteren Assoziationen explizit aufgebaut werden.

Durch die Kappselung des gesamten Instanzierungsvorgangs der Messsicht kann auch gewährleistet werden, dass jeweils genau eine Instanz der Klassen *Ereigniskorrelation* und *Nachverarbeitung* aufgebaut wird , wie in (5a, 7a) gefordert.

Die Zusammenfassung aller Instanzierungsvorgänge in einer Methode gewährleistet auch, dass vor Beginn einer Messung alle zur Messsicht gehörenden Instanzen aufgebaut und durch die entsprechenden Assoziationen aneinander gebunden sind (9a, 9b). Wenn der Kontrollfluss aus der Methode *Messsicht_aufbauen* zum aufrufenden Objekt zurückkehrt, ist die Messsicht vollständig aufgebaut.

Durch die Bündelung aller zum Aufbau der Messsicht notwendigen Instanzierungsschritte in einer einzigen Methode wird optimale Transparenz für den Entwickler resp. späteren Anwender der Messsicht erreicht. Es müssen lediglich die implementierungsabhängigen Instanzen referenziert werden. Alle übrigen Instanzen werden nach den Vorgaben der Entwicklersicht automatisch aufgebaut, nachdem vom Compiler, ebenfalls automatisch, eine geeignete Verfeinerung des Modells des allgemeinen Messprozesses erzeugt wurde. Damit kann aus der formalen Spezifikation in der Entwicklersicht automatisch ein lauffähiges Messsystem generiert werden. Die Anwendungsmöglichkeiten, die sich durch diesen Automatismus erschließen, werden in Kapitel 5 ausführlich beschrieben.

4.3.2.3.5. Compiler Der Compiler übersetzt die einzelnen Elemente der Sprache der Entwicklersicht in eine entsprechende Verfeinerung des erweiterten Modells des allgemeinen Messprozesses, wie in Abschnitt 4.3.2.3.3 beschrieben, und erzeugt zugleich eine auf die Spezifikation des jeweiligen Dienstgütemerkmals abgestimmte Verfeinerung der Klasse *Instanzierungssteuerung*, wie sie im vorherigen Abschnitt 4.3.2.3.4 eingeführt wurde. Er übernimmt damit die Funktion des Generators, der bei der Einführung des Automatisierungskonzepts in Abschnitt 4.3.2.1 als zentrales Element vorgeschlagen wurde.

Das hier vorgestellte Design des Compilers beschreibt nicht die Implementierung desselben, sondern grundsätzliche Produktionsregeln, wie sie in einer Implementierung umgesetzt werden müssen. Diese können zur Implementierung, in Abhängigkeit von der für die Entwicklersicht gewählten Sprache, soweit verfeinert werden, dass sie als Eingabe für einen Compilergenerator wie yacc (Yet Another Compiler Compiler) [yacc] oder javacc (Java Compiler Compiler) [JavaCC] verwendet werden können. Dieses Vorgehen wird im nachfolgenden Abschnitt 4.3.3 umgesetzt, wenn die Sprache für die Entwicklersicht zusammen mit dem Compiler implementiert wird.

Als Basis für die möglichen Produktionen des Compilers stehen die beim Design der Entwicklersicht in Abschnitt 4.3.2.3.2 eingeführten Sprachkonstrukte zur Verfügung. Das vorgestellte Design verknüpft dementsprechend die den Compiler betreffenden Spezifikationsrichtlinen resp. Anforderungen mit den im Design der Entwicklersicht bereits festgelegten Sprachkonstrukten.

Die folgende Darstellung gliedert sich nach dem Aufbau der Produktionsregeln. Für jedes beim Design der Entwicklersicht eingeführte Sprachelement werden im Folgenden informell Produktionsregeln beschrieben, die festlegen, in welcher Weise die für die Definitionssicht und für den Aufbau der geforderten Verfeinerungen von den jeweiligen Sprachelementen abhängen. Dabei zeigt sich nochmals die Erfüllung der aufgestellten Spezifikationsrichtlinien. Entsprechende Referenzen sind wieder in Klammern angegeben. Die Darstellung wird durch eine entsprechende

Nummerierung gegliedert. Diese wird in Abbildung 4.14, die einen beispielhaften Aufbau der Definitionssicht aus dem in Abschnitt 4.3.2.3.3 aufgebauten erweiterten Modell des allgemeinen Messprozesses zeigt, verwendet, um zu verdeutlichen, welche Produktionsregeln, in welcher Weise das Modell verfeinern.

In Abbildung 4.15 ist mit einem analogen Vorgehen beispielhaft dargestellt, wie die Instanzierungshilfestellung durch die einzelnen Produktionsregeln aufgebaut wird. Die Nummern geben dabei wieder, welche Produktionsregeln die im Diagramm dargestellten Ablaufschritte festlegen. Alle Produktionsregeln werden im Folgenden detailliert beschrieben. Diese Festlegungen können in einem weiteren Schritt zusammen mit den Sprachelementen der Entwicklersicht formalisiert werden und als Eingabe für einen Compilergenerator verwendet werden. Dieser Schritt wird im folgenden Abschnitt 4.3.3 durchgeführt.

Die Darstellung der für den Compiler notwendigen Produktionsregeln orientiert sich in ihrer Reihenfolge an den bereits in der Entwicklersicht (Abschnitt 4.3.2.3.2) eingeführten Sprachelementen. Aus den meisten Sprachelementen ergeben sich mehrere Produktionsregeln, die entsprechend beim jeweiligen Sprachelement angegeben werden. Die Nummerierung der Sprachelemente richtet sich nach der bereits in der Entwicklersicht (Abschnitt 4.3.2.3.2) verwendeten Reihenfolge. Tabelle 4.1 am Ende dieses Abschnitts fasst die Produktionsregeln nochmals zusammen.

1. Konstrukt zur Festlegung von Primitiven

    a) Generierung der Unterklasse von Primitive für jede Dienstprimitive (2a)

    Mit der Festlegung des Namens einer Primitive lässt sich diese als Verfeinerung von der Oberklasse *Primitive* ableiten.

    b) Parameter einer Dienstprimitive als Attribute dieser Verfeinerung modellieren (2b)

    Innerhalb des Konstrukts zur Festlegung von Primitiven werden auch die Attribute einer Primitive bestimmt. Diese können an Hand der Definitionen im Konstrukt in die bereits definierte Verfeinerung eingetragen werden. Die Attribute müssen auch in ihrem Typ festgelegt werden. Dazu bietet die Entwicklersicht entsprechende Konstrukte an, deren Übersetztungsregeln im folgenden Absatz beschrieben werden.

2. deklaratives Typenkonzept

    Abhängig von den in der Entwicklersicht vorgesehenen Möglichkeiten zur Festlegung von Datentypen müssen diese in die jeweils gewählte Zielsprache des Compilers übersetzt werden. Wird als Zielsprache eine übliche Programmiersprache wie C++ oder Java [GJSB 00] gewählt, bietet es sich an, das Typenkonzept strikt am Typenkonzept der Zielsprache zu orientieren. Damit werden die Produktionsregeln zu einfachen Textersetzungsregeln.

3. Konstrukt zur Festlegung von Dienstgüteereignissen

    a) Verfeinerung von Ereignisgenerator für jedes in der Entwicklersicht benannte Dienstgüteereignis (3a)

    Für jede in der Entwicklersicht angegebene Ereignisklasse muss eine Verfeinerung der Klasse *Ereignisgenerator* aufgebaut werden, welche diese Ereignisse aus den entsprechenden

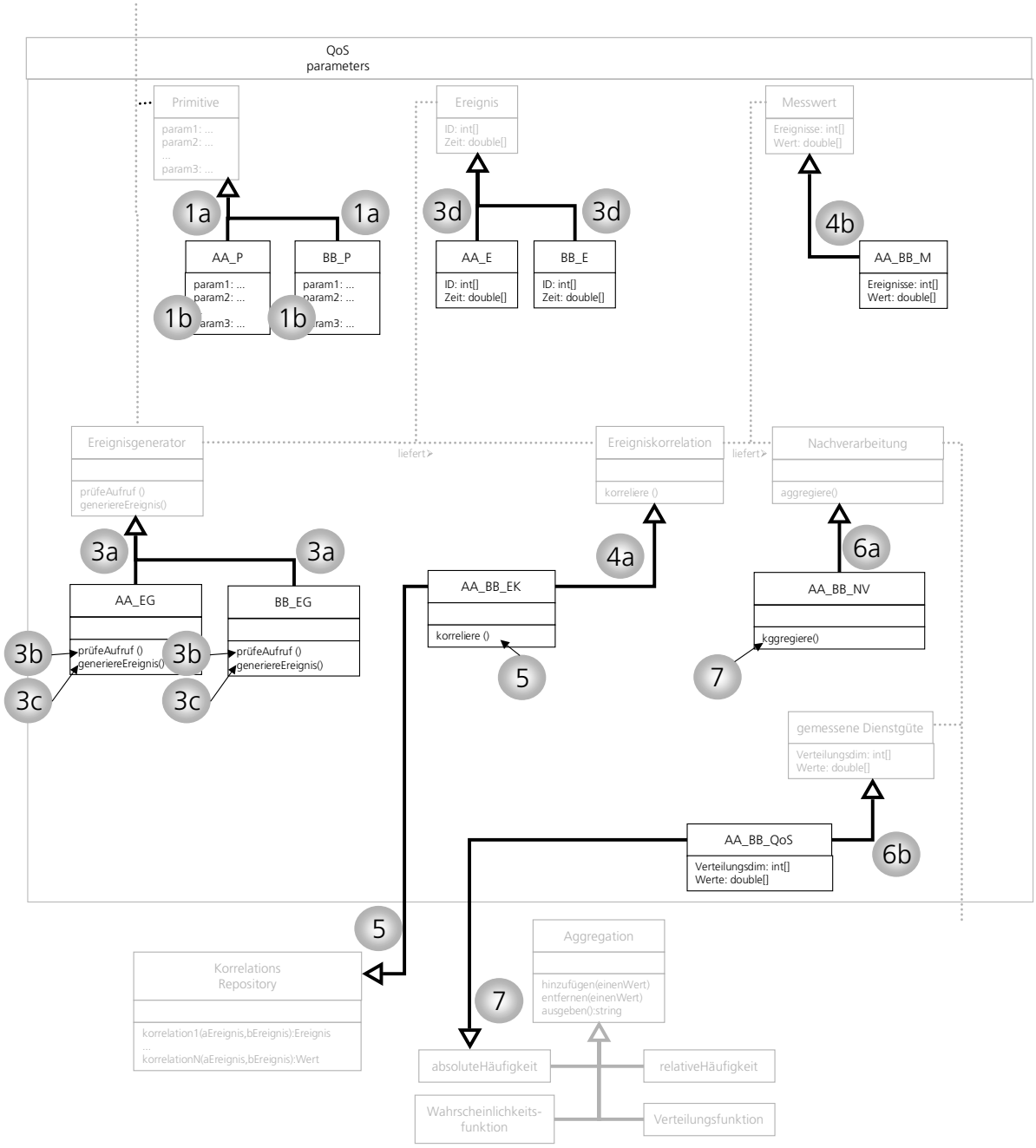

Abbildung 4.14.: Bezug zwischen Verfeinerungsschritten beim Aufbau der Definitionssicht für ein Dienstgütemerkmal und einzelnen Produktionsregeln (Nummern)

Primitiven ableiten kann. Welche Primitiven zur Ableitung der Dienstgüteereignisse verwendet werden, wird erst bei der Instanzierung und der damit verbundenen Registrierung der Verarbeitungskomponenten bei ihrer jeweiligen Vorstufe festgelegt. Eine verfeinerte Ereignisgeneratorklasse erhält zweckmäßiger Weise einen Namen, der auf das generierte Ereignis schließen lässt.

b) Methode *prüfePrimitive* nach Vorgaben der Entwicklersicht verfeinern (3b)

   In der Entwicklersicht wird auch

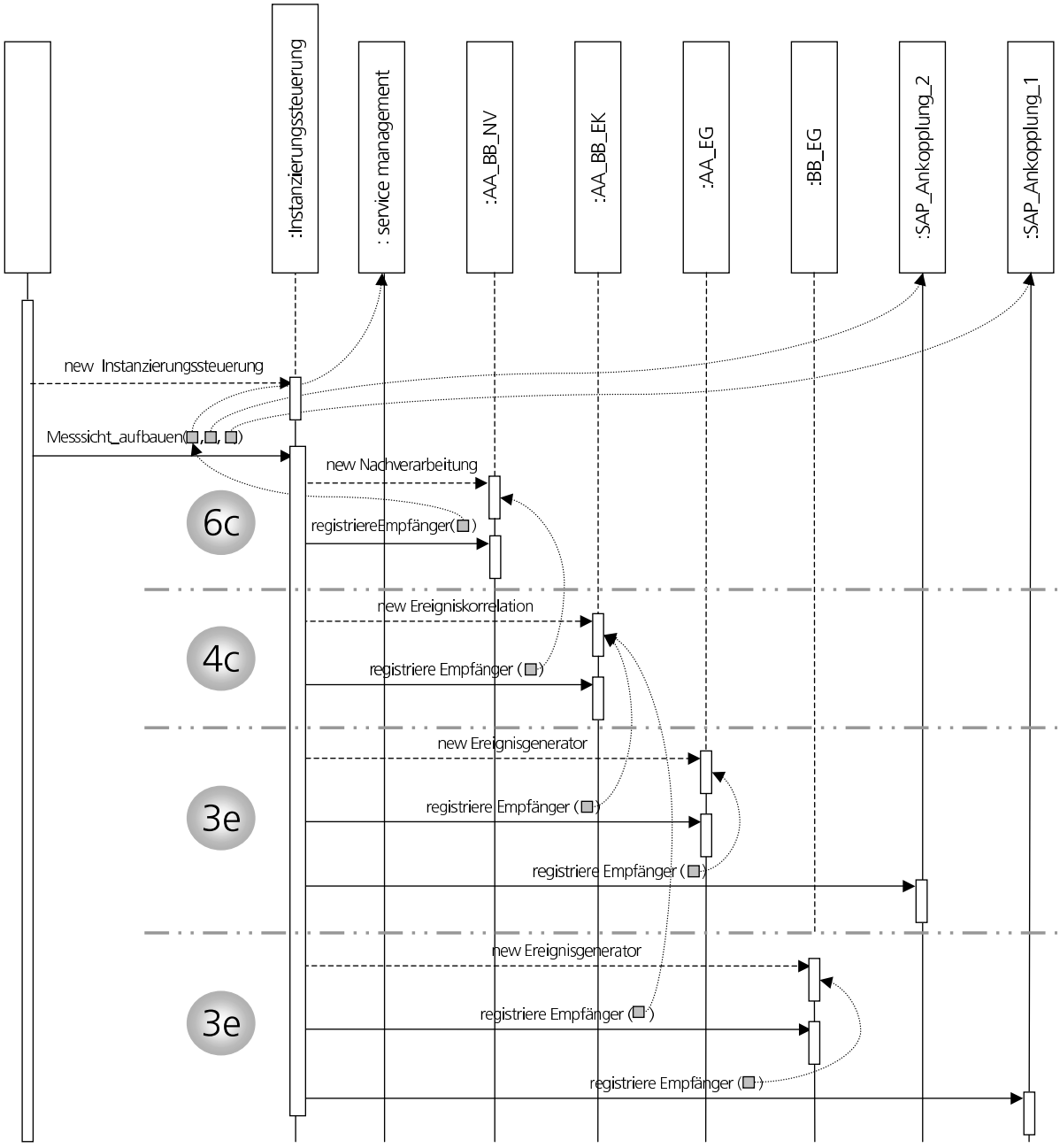

Abbildung 4.15.: Bezug zwischen dem Aufbau der Instanzierungshilfestellung und einzelnen Produktionsregeln (Nummern)

festgelegt, in welcher Weise eine Primitive, die als Basis für die Generierung eines Ereignisses verwendet werden soll, gefiltert werden soll. Diese Angaben können direkt verwendet werden, um die Methode *prüfePrimitive* zu überladen. Dabei kann auch das Interface dieser Methode so definiert werden, dass diese Methode als Argumente nur Instanzen der jeweils zutreffenden Primitivenklasse, wie sie in der Definition des jeweiligen Ereignisses angegeben ist, akzeptiert.

c) Methode *generiereEreignis* nach Vorgaben der Entwicklersicht verfeinern (3c, 4a)

Nach den Festlegungen der für die Generierung eines Dienstgüteereignisses notwendigen Primitive und der Vorschrift zur

Bildung des Ereignisidentifikators lässt sich die Methode *generiereEreignis* verfeinern. In dieser Methode werden die Attribute des Ereignis-Objekts belegt und das Objekt wird durch Aufruf der Methode *empfange* beim jeweiligen Ereignisgenerator übertragen.

d) Verfeinerung von *Dienstgüteereignis* nach den Vorgaben der Entwicklersicht generieren (4a)

Aus der Benennung eines Ereignisses in der Entwicklersicht ergibt sich direkt die entsprechende Verfeinerung der Klasse *Dienstgüteereignis*. Die Verfeinerung wird nach der Vorgabe der Entwicklersicht benannt.

e) Generierung der Instanzierungshilfestellung nach (4.3.2.1)

Für jedes in der Entwicklersicht benannte Ereignis kann ein Teil der Methode *Messsicht_aufbauen* für eine Verfeinerung der Klasse *Instanzierungssteuerung*, die in Abhängigkeit vom jeweils in der Entwicklersicht festgelegten Dienstgütemerkmal erzeugt wird (siehe Produktionsregel 6c), definiert werden. Dabei wird die Methode entgegen ihrer Abarbeitungsreihenfolge aufgebaut, wie Abbildung 4.15 verdeutlicht. Nachdem der für den Aufbau eines Dienstgüteereignisses notwendige Ereignisgenerator als Verfeinerung der Klasse *Ereignisgenerator* definiert wurde, kann eine Instanz dieser Klasse, die von der Methode *Messsicht_aufbauen* bereits erzeugt wurde, als Empfänger bei der zugehörigen SAP_Ankopplung registriert werden. Beim Aufruf der Methode müssen die Instanzen der Klasse *SAP_Ankopplung* in der Reihenfolge angegeben werden, die derjenigen der symbolischen Referenzen aus der Entwicklersicht entspricht (bei einer einfachen Nummerierung entsprechend in der Reihenfolge dieser Nummerierung) . Nur so kann sichergestellt werden, dass die Ereignisgeneratoren mit den jeweils „passenden" Instanzen der Klasse *SAP_Ankopplung* verbunden werden. Die Bereitstellung und damit die logisch richtige Einbindung dieser Instanzen muss durch den Entwickler in „Handarbeit" erfolgen, wie in Abschnitt 4.3.2.3.6 ausführlich beschrieben. Aus der Anzahl der in der Entwicklersicht verwendeten symbolischen Referenzen lässt sich die Anzahl der beim Aufruf der Methode *Messsicht_aufbauen* notwendigen Referenzen auf SAP-Ankopplungsinstanzen ableiten.

4. Konstrukt zur Festlegung von Messwerten

a) *Ereigniskorrelation* nach den Vorgaben der Entwicklersicht (wie ergibt sich ein Messwert) verfeinern (5a)

Aus der Definition eines Messwerts und seinen Abhängigkeiten von Ereignissen in der Entwicklersicht lässt sich zunächst eine Verfeinerung der Klasse *Ereigniskorrelation* bilden, die über die Methode *korreliere* den entsprechenden Messwert erzeugt, wie bei den Produktionsregeln 4b und 5 weiter beschrieben.

b) Verfeinerung von *Messwert* nach den Vorgaben der Entwicklersicht generieren (6a)

   Aus der Definition des Messwertes in der Entwicklersicht kann eine entsprechende Verfeinerung der Klasse *Messwert* erzeugt werden. Für Messwert und Ereignisgenerator bietet es sich an, einen Namen zu verwenden, der auf die verarbeiteten Diensgüteereignisse schließen lässt.

c) Generierung der Instanzierungshilfestellung nach (4.3.2.1)

   Nachdem die Verfeinerungen der Klassen *Ereigniskorrelation* und *Messwert* nach den Vorgaben der Entwicklersicht aufgebaut sind, kann auch die Methode *Messsicht_aufbauen* ergänzt werden: Eine Instanz der Verfeinerung von *Ereigniskorrelation* wird aufgebaut. Bei dieser wird die entsprechende Nachverarbeitungsinstanz als Empfänger registriert. Auch hier erfolgt der Aufbau der Methode *Messsicht_aufbauen* entgegen ihrer Verarbeitungsreihenfolge. Innerhalb der Methode müssen deshalb alle Referenzen auf die benötigten Verarbeitungsinstanzen in entsprechenden Variablen gehalten werden.

5. Schlüsselwörter zur Referenzierung von Korrelationsfunktionen

   Die zur Berechnung eines Messwertes notwendigen Korrelationsfunktionen werden in der Entwicklersicht direkt bezeichnet. Zur einfachen Implementierung der Korrelationsfunktionalität bietet das erweiterte Modell des allgemeinen Messprozesses (siehe Abschnitt 4.3.2.3.3) die Klasse *KorrelationsRepository* an, welche unterschiedliche Korrelationsfunktionen als Methoden bereitstellt. Damit diese Methoden einfach zur Implementierung der Methode *korreliere* verwendet werden können, entsteht eine Verfeinerung der Klasse *Ereigniskorrelation* nicht nur durch Ableitung von dieser Klasse, sondern auch durch Ableitung von der Klasse *KorrelationsRepository*, wie in Abbildung 4.14 im Schritt 5 dargestellt. Damit lässt sich die Korrelationsfunktionalität nach den Vorgaben der Entwicklersicht einfach in die Methode *korreliere* integrieren, wie in (5b) gefordert.

6. Konstrukt zur Beschreibung der gemessenen Dienstgüte

   a) Verfeinerung von *Nachverarbeitung* nach den Vorgaben der Entwicklersicht bilden (7a)

      Durch die Angabe der notwendigen statistischen Nachverarbeitung zur Darstellung eines Dienstgütemerkmals in der Entwicklersicht kann eine Verfeinerung der Klasse *Nachverarbeitung* aufgebaut werden. Die eigentliche Funktionalität wird durch die Methode *aggregiere* übernommen. Ihr Aufbau wird durch die Produktionsregel 7 festgelegt.

   b) Verfeinerung von *gemessene Dienstgüte* nach Vorgaben der Entwicklersicht generieren (8a, 8b)

      Aus der Definition der gemessenen Dienstgüte in der Entwicklersicht lässt sich kanonisch eine Ableitung von der Klasse *gemesseneDienstgüte* bilden. Abhängig von der gewählten Aggregationsform ändern sich auch die Attribute und Methoden dieser Klasse. Produktionsregel 7

übernimmt die Abbildung dieser Abhängigkeiten.

c) Generierung der Instanzierungshilfestellung nach (4.3.2.1)

Nachdem die Verfeinerungen der Klassen *Nachverarbeitung* und *gemesseneDienstgüte* erzeugt wurden, kann auch der Aufbau der Instanzierungshilfestellung vervollständigt werden: Eine Instanz der eben angelegten Verfeinerung der Klasse *Nachverarbeitung* wird erzeugt. Die beim Methodenaufruf übergebene Instanz der Klasse *service management* wird als Empfänger registriert. Zudem wird der Methodencode komplettiert, so dass eine ausführbare Methode entsteht.

7. Schlüsselwörter zur Referenzierung von Nachverarbeitungsfunktionen (7b)

Um die Aggregationsfunktionalität und die zugehörigen Darstellungsformen einfach verwenden zu könnnen, werden diese in Klassen als Verfeinerung der Klasse *Aggregation* mit einem standardisierten Interface angeboten. Die Methode *aggregiere* der Klasse *Nachverarbeitung* resp. einer Verfeinerung erstellt eine Instanz der Verfeinerung von *gemesseneDienstgüte*. Diese wird auch als Verfeinerung der entsprechenden Aggregationsklasse definiert, so dass die zur gewählten Aggregationsform gehörenden Methoden automatisch zur Verfügung stehen und direkt bei der Implementierung der Methode *aggregiere* verwendet werden können. Die Unterklassen der Klasse *Aggregation* liefern damit ein Repository als Basis zur Definition der Aggragationsmethode.

Die informelle Darstellung der für den Compiler notwendigen Produktionsregeln hat nochmals die Tauglichkeit des eingeführten Automatisierungsprozesses gezeigt. Alle durch die Spezifikationsrichtlinien aus Abschnitt 4.2.4 aufgestellten Anforderungen an den Aufbau der Definitions- und Messsicht lassen sich mit dem gewählten Design umsetzen. Zum Aufbau der Messsicht ist allerdings noch unvermeidbare „Handarbeit" des Entwicklers notwendig, wie im Folgenden beschrieben wird.

4.3.2.3.6. „Handarbeit" des Entwicklers

Zur Vervollständigung des Designs des Automatisierungsprozesses wird in diesem Abschnitt nochmals zusammenfassend dargestellt, welche Aufgaben bei der Spezifikation eines Dienstgütemerkmals nicht automatisierbar und deshalb vom Entwickler in „Handarbeit" durchzuführen sind.

Zu diesen Aufgaben gehört natürlich die Festlegung der Entwicklersicht selbst. Mit dem vorgestellten Design kann daraus die Definitionssicht, also eine Verfeinerung des erweiterten Modells des allgemeinen Messprozesses, automatisch generiert werden, ohne dass weitere Eingriffe durch den Entwickler notwendig wären.

Beim Aufbau der Messsicht wird die Definitionssicht nach den Vorgaben der Entwicklersicht automatisch instanziert. Zur Ankopplung an eine bestehende Dienstimplementierung müssen vom Entwickler noch die im Folgenden beschriebenen Schritte durchgeführt werden. Dabei wird die Messsicht an die durch die Dienstimplementierung vorgegebenen Gegebenheiten angepasst. Diese Anpassungen sind nur mit Wissen über diese Implementierung möglich und deshalb nicht automatisierbar. Das vorliegende Design begrenzt diese Arbeitschritte jedoch auf das absolut notwendige Maß.

Konstrukt zur Festlegung von Primitiven	1	a	Generierung der Unterklasse von *Primitive* für jede Dienstprimitive
		b	Parameter einer Dienstprimitive als Attribute dieser Verfeinerung modellieren
deklaratives Typenkonzept	2		möglichst direkte Abbildung in die Zielsprache
Konstrukt zur Festlegung von Dienstgüteereignissen	3	a	Verfeinerung von *Ereignisgenerator* für jedes in der Entwicklersicht benannte Dienstgüteereignis
		b	Methode *prüfeAufruf* nach Vorgaben der Entwicklersicht verfeinern
		c	Methode *generiereEreignis* nach Vorgaben der Entwicklersicht verfeinern
		d	Verfeinerung von *Dienstgüteereignis* nach den Vorgaben der Entwicklersicht generieren
		e	Generierung der Instanzierungshilfestellung (als Empfänger von Primitiven registrieren)
Konstrukt zur Festlegung von Messwerten	4	a	*Ereigniskorrelation* nach den Vorgaben der Entwicklersicht (wie ergibt sich ein Messwert) verfeinern
		b	Verfeinerung von *Messwert* nach den Vorgaben der Entwicklersicht generieren
		c	Generierung der Instanzierungshilfestellung (als Empfänger von Dienstgüteereignissen registrieren)
Schlüsselworte zur Referenzierung von Korrelationsfunktionen	5		Korrelationsfunktionalität nach den Vorgaben der Entwicklersicht unter Verwendung der Methoden aus *Korrelationsrepository* in die Methode *korreliere* integrieren
Konstrukt zur Bescheribung der gemessenen Dienstgüte	6	a	Verfeinerung von *Nachverarbeitung* nach den Vorgaben der Entwicklersicht bilden
		b	Verfeinerung von *gemesseneDienstgüte* nach Vorgaben der Entwicklersicht auf Basis der den Nachverarbeitungmethoden assozierten Ausprägungsklassen generieren
		c	Generierung der Instanzierungshilfestellung (als Empfänger von Messwerten registrieren)
Schlüsselworte zur Referenzierung von Nachverarbeitungsfunktionen	7		Aggregationsklasse zur Ableitung von *gemesseneDienstgüte* nach der in der Entwicklersicht festgelegten Funktionalität (aus Repository) auswählen

Tabelle 4.1.: Übersicht der Produktionsregeln

1. Für die Funktion der Messsicht ist die Bereitstellung von Instanzen der Klasse *SAP_Ankopplung* notwendig. Diese müssen die benötigten Primitiven, wie sie in der Entwicklersicht festgelegt sind, bereitstellen können (1b) und das durch das erweiterte Modell des allgemeinen Messprozesses festgelegte Interface implementieren.

2. Der Entwickler muss ebenso sicherstellen, dass die bereitgestellten Instanzen der Klasse *SAP_Ankopplung* Primitiven an den jeweils zur Messung gewünschten Schnittstellen zwischen Client und Dienst rekonstruieren. Damit legt er fest, zwischen welchen „Punkten" gemessen wird. (1a)

3. Die Instanzierungshilfestellung bindet eine Instanz der Klasse *management implementation* automatisch in die Messsicht ein, wenn vom Entwickler eine entsprechende Referenz beim Aufruf der Methode *Messsicht_aufbauen* übergeben wird. Er muss dabei dafür sorgen, dass diese Instanz zugleich die Methoden der Klasse *DienstgüteEmpfänger* implementiert (7d). Wenn ein direkter Eingriff in die Implementierung nicht möglich ist, kann dies durch eine geeignet angelegte Wrapper-Klasse erfolgen.

Mit der abschließenden Vorstellung des Designs eines Automatisierungsprozesses ist die umfassende Vorstellung des Konzepts zur Spezifikation und Messung von Dienstgütemerkmalen abgeschlossen. Ausgehend von der in Abschnitt 4.2.1.1 erläuterten Lösungsidee wurden unterschiedliche Schritte bei Spezifikation und Messung, also in den Phasen des Dienstlebenszyklus, identifiziert und für diese sog. Sichten eine formale Grundlage in

Form eines Klassenmodells gelegt. Die Ableitung von Spezifikationsrichtlinien aus dem typischen Vorgehen bei der Umsetzung einer Spezifikation ermöglichte es schließlich, einen grundsätzlichen automatisierten Ablauf festzulegen, dessen einzelne Teilergebnisse formal vorliegen. Abhängigkeiten zwischen den einzelnen Teilergebnissen werden automatisch erfasst resp. realisiert und müssen vom Entwickler nicht explizit beachtet werden.

Im folgenden Abschnitt wird, basierend auf den bisher erarbeiteten Ergebnissen, eine mögliche Implementierung aller bisher festgelegten Teile des Gesamtkonzepts vorgestellt, wodurch eine ablauffähige und damit praktisch einsetzbare Umsetzung des vorgestellten Konzepts entsteht, dessen Anwendungsmöglichkeiten in Kapitel 5 vorgestellt werden. Zentrales Element dieser Implementierung ist eine formale Sprache zur Spezifikation der Entwicklersicht, die im Abschnitt 4.3.3.4 prototypisch implementiert wird.

4.3.3. Mögliche Implementierung des Ableitungsprozesses

Für das im vorhergehenden Abschnitt 4.3.2.3 vorgestellte Design eines Automatisierungsprozesses zur Spezifikation von Dienstgütemerkmalen wird im Folgenden eine mögliche Implementierung vorgestellt, so dass ein ablauf- und einsatzfähiges System zur Spezifikation und Messung von Dienstgütemerkmalen entsteht. Das vorgestellte Design, und damit auch die im Folgenden beschriebene Implementierung, bietet bei der Spezifikation und Messung von Dienstgütemerkmalen weitestgehende Rechnerunterstützung. Ähnlich wie bei CAD (Computer Aided Design) oder CASE (Computer Aided Software Engineering) entsteht damit CASAM (Computer Aided Specification And Measurement).

Die hier entwickelte Implementierung verwendet im Gegensatz zum bisherigen Design englische Namen für Sprachkonstrukte und Bezeichner, um eine weitest mögliche Einsetzbarkeit sicherzustellen. Diese implementierungsspezifische Festlegung wird durch das Design nicht vorgegeben. Die vorliegende Implementierung ist also als beispielhaft zu betrachten. Ebenso beispielhaft ist die Umsetzung mit der Programmiersprache Java [GJSB 00] und dem Compilergenerator javacc.

Die vorliegende Implementierung verwendet so weit wie möglich bereits bestehende Systeme und bindet diese ein. Mit Java als Zielsprache für die Implementierung und die generierte Definitions- und Messsicht wurde eine weit verbreitete Programmiersprache gewählt. Der folgende Abschnitt erläutert die Einbindung bestehender Tools bei der Implementieurng des gesamten Automatisierungsprozesses. Die daran anschließenden Abschnitte beschreiben prinzipiell die einzelnen Komponenten des Automatisierungsprozesses. Implementieurngsspezifischer Java-Code ist im Anhang zusammengefasst, um die Darstellung übersichtlich zu halten.

4.3.3.1. Implementierung des Automatisierungsprozesses

Der prinzipielle Aufbau der hier entwickelten Implementierung des Automatisierungsprozesses ist in Abbildung 4.16 grafisch dargestellt. Entsprechend der Legende unterscheidet diese Abbildung standardisierte Softwarelösungen, die bei der Implementierung wiederverwendet werden, generierte bzw. vom Entwickler getroffene Festlegungen und Spezifikationen, die für die vollständige Implementierung noch entwickelt werden müssen.

Eine vom Entwickler festgelegte Dienstgütemerkmalsdefiniton in der Sprache QoSSL (Quality of Service Specification Language) wird vom QoSSL-Compiler zu einer Spezifikation der Definitionssicht in Java

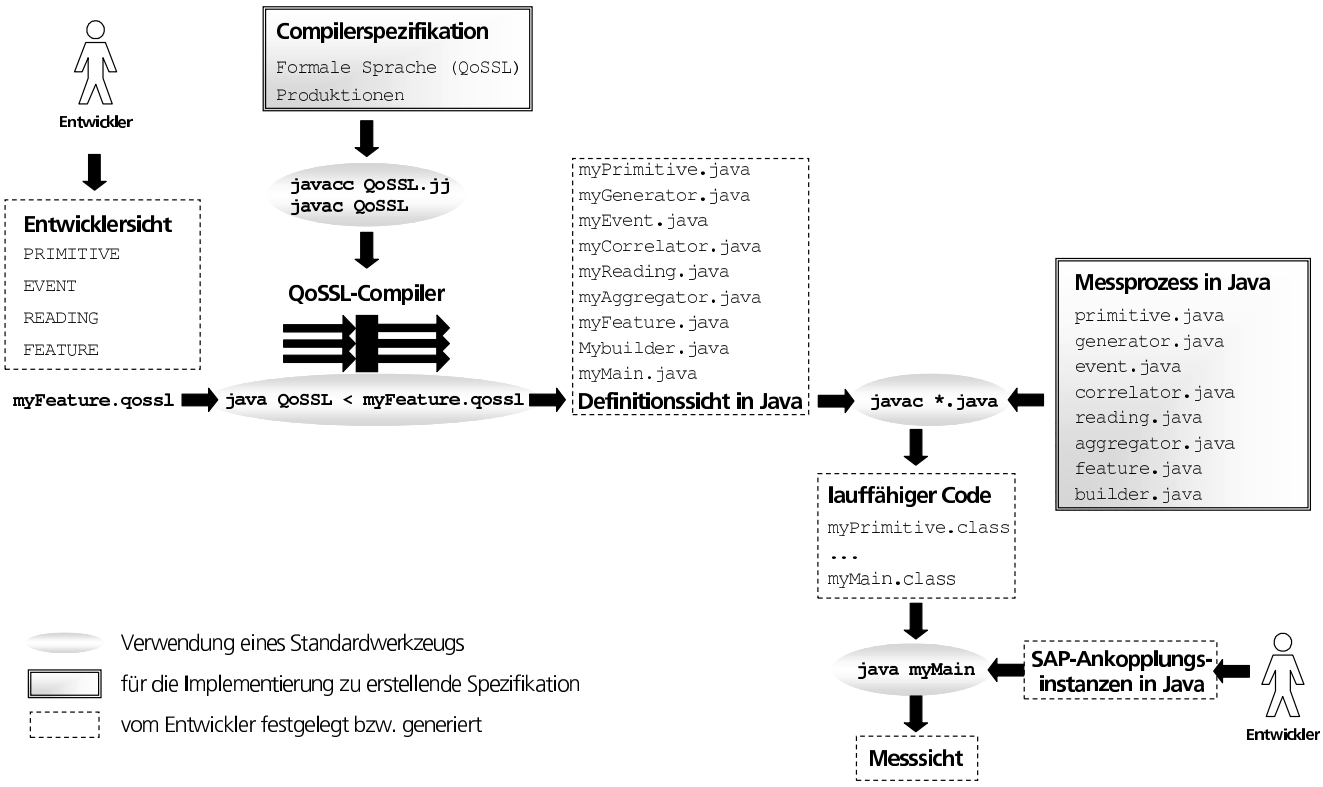

Abbildung 4.16.: Implementierung des Automatisierungsprozesses auf Basis von java und javacc

verarbeitet und zusätzlich ein Steuerprogramm *myMain* erzeugt. Definitionssicht und Steuerprogramm werden zusammen mit der ebenfalls in Java vorliegenden Spezifikation des Messprozesses, wie er in Abschnitt 4.3.2.3.3 festgelegt wurde, durch den Java-Compiler übersetzt. Damit entsteht ein durch das Java-Laufzeitsystem ausführbares Programm, das vom Entwickler durch die Angabe von Ankopplungsinstanzen angepasst und gestartet werden kann. Damit wird eine ablaufende Messsicht erzeugt.

Der QoSSL-Compiler selbst ist ein Java-Programm, das durch den Compilergenerator javacc aus einer entsprechenden Spezifikation erzeugt wurde. Diese umfasst sowohl die formale Festlegung von QoSSL als auch die notwendigen Produktionsregeln zum Aufbau der Definitionssicht.

4.3.3.2. Java-Implementierung des Modells des erweiterten Messprozesses

Um den vorgestelleten Automatisierungsprozess aus Abschnitt 4.3.2.3 ablauffähig implementieren zu können, muss das erweiterte Modell des allgemeinen Messprozesses, wie es in Abschnitt 4.3.2.3.3 im Detail festgelegt wurde, in einer Programmiersprache implementiert werden. Nach den zu Beginn von Abschnitt 4.3.3 aufgestellten, grundsätzlichen Implementierungsentscheidungen wurde als Sprache zur Implementierung des Automatisierungsprozesses, aber auch zur Beschreibung der Definitions- und Messsicht, Java gewählt. Nachfolgend wird ausgeführt, wie das in UML beschriebene erweiterte Modell des Messprozesses in Java umgesetzt wird. Da sich die Umsetzung in großen Teilen kanonisch aus der UML-Beschreibung ergibt, werden hier lediglich nicht triviale Implementierungsentschei-

dungen beschrieben. Der Quellcode, der das gesamte Modell implementiert, findet sich im Anhang unter A.1.

Das in Abschnitt 4.3.2.3.3 vorgestellte Design lässt noch einige Aspekte der Implementierung offen: Ereignisse müssen mit Zeitstempeln versehen werden, der Ereigniskorrelator muss empfangene Ereignisse puffern können und für die bei der Festlegung von QoSSL (siehe 4.3.3.4) definierten Korrelationsfunktionen und Darstellungsformen müssen entsprechende Methoden resp. Klassen implementiert werden. Zusätzlich sind nicht alle UML-Konstrukte (wie z.B. Mehrfachvererbung) direkt in Java abbildbar. Eine detaillierte Beschreibung findet sich im Anhang in Abschnitt 4.3.3.2.

4.3.3.3. Aufbau des Compilers mit javaCC

Auf Basis des entwickelten Designs und der aufgestellten Produktionsregeln lässt sich ein Compiler implementieren, der automatisch aus der formal spezifizierten Entwicklersicht die Definitionssicht und eine Unterstützung zum Aufbau der Messsicht generieren kann. In der durchgeführten prototypischen Implementierung wurde der Compiler mit dem Compilergenerator javacc aufgebaut. Der Quellcode für den Aufbau des Compilers ist im Anhang in Abschnitt A.3 beschrieben.

4.3.3.4. QoSSL- Formale Sprache zur Spezifikation von Dienstgüte merkmalen

Auf Basis des Designs des Automatisierungsprozesses, das in Abschnitt 4.3.2 entwickelt und festgelegt wurde, lässt sich im Zuge der Implementierung des Prozesses, wie sie prototypisch im Abschnitt 4.3.3 vorgenommen wird, eine formale Sprache zur Spezifikation von Dienstgütemerkmalen entwerfen.

Eine mögliche Definition dieser Sprache wird im Folgenden aufgestellt. Damit wird die gesamte Anwendungsbreite des vorgestellten Spezifikationskonzepts als Summe von Spezifikationssprache und rechnergestütztem Umsetzungsprozess deutlich.

Da bisher lediglich das Design der Spezifikationssprache, ihre grundsätzlichen Elemente, festgelegt wurden, sind vielfältige Festlegungen einer konkreten Syntax denkbar. Mit der vorgestellten Sprache QoSSL wird eine mögliche Syntax vorgestellt, die den deklarativen Ansatz auch in ihrer Syntax widerspiegelt. Die Syntaxdefinition ist dabei ebenfalls als prototypischer Ansatz zu verstehen, der die prinzipielle Umsetzbarkeit des Konzepts zeigt.

Die Elemente der Sprache sind prinzipiell festgelegt, lediglich die Bezeichnungen für bestimmte Konstrukte (syntactic sugar) und die Form etwaiger Parameterangaben können noch festgelegt werden.

Da das vorliegende Design offen lässt, welche Korrelationsfunktion und welche Darstellungsformen für die gemessene Dienstgüte zur Verfügung stehen sollen, sind diese Funktionalitäten bei der endgültigen Definition einer Spezifikationssprache ebenfalls festzulegen.

Die für QoSSL notwendigen Sprachelemente werden nachfolgend in der Reihenfolge, in der ihre prinzipielle Notwendigkeit beim Design der Entwicklersicht erläutert wurde, eingeführt und beschrieben. Neben der formalen Definition des jeweiligen Konstrukts in BNF [ISO 14977] wird auch ein kurzes Beispiel einer mit diesem Konstrukt möglichen Anweisung angegeben.

1. Konstrukt zur Festlegung von Primitiven

```
<PRIMITIVE> := "PRIMITVE"
   <Primitive_Identifier>
   "("
   (    <Attribute_Identifier> ":" <type>)?
   ("," <Attribute_Identifier> ":" <type>)*
   ");"
```

Mit dem Schlüsselwort PRIMITIVE wird die Beschreibung einer Dienstprimitive mit ihren zugehörigen Attributen eingeleitet. Eine Primitive kann auch völlig ohne Attribute beschrieben werden, um auch einfachste Dienstzugriffe modellieren zu können. Der Bezeichner einer Primitive kann frei gewählt werden, sollte sich aber am jeweils beschriebenen Dienst resp. Protokoll orientieren. Zur Festlegung der Attributtypen wird das im nächsten Punkt beschriebene Konzept verwendet. Die Definition der receive-Primitive für UDP lautet nach diesen Vorgaben[2]:

```
PRIMITIVE recv (  data:       BYTE[]
                  len:        BYTE[2]
               );
```

2. deklaratives Typenkonzept

```
<TYPE> :=   <BIT> <ARRAY_DEFINITION>
          | <BIT>
          | <BYTE> <ARRAY_DEFINITION>
          | <BYTE>
          | <CHAR> <ARRAY_DEFINITION>
          | <CHAR>
```

In der vorgestellten Version von QoS-SL werden nur einfachste Datentypen unterstützt. Als Basis stehen die Typen BIT, BYTE und CHAR zur Verfügung. Alle Basistypen können zu Feldern mit bestimmter oder unbestimmter Länge zusammengefasst werden. Felder unbestimmter Länge sind z.B. für die Abbildung von Payloads mit nicht feststehender Länge notwendig, z.B. bei der Datenübertragung mit UDP, wie im obigen Beispiel bereits dargestellt. Eine Erweiterung des Typenkonzepts ist denkbar und zur Beschreibung von Dienstprimitiven auf höheren Schichten auch notwendig. Übergabeparameter eines Primitivenaufrufs können dabei wiederum Objekte sein, die sich durch einfache Datentypen nicht beschreiben lassen.

3. Konstrukt zur Festlegung von Ereignissen

```
<Event> := "EVENT"   <Event_Identifier>
           "USES"    <Primitive_Identifier>
           "ON TAP"  <Tap_Symbol>
           "ISOLATE"
              "(" (<Filter_Term>)

                   ("," <Filter_Term>)*

              ")"
           "ID"       "AUTO"
                    | <Attribute_Identifier>
                    | <Hash>
           ";"

<TAP_SYMBOL> := [1..9][0..9]

<Filter_Term> :=   "*"
                 | ( <COMPARE>
                     (   <NUM_CONSTANT>
                       | <ALPHA_CONSTANT>
                       | <IDENTIFIER>
                     )
                   )
                 | ( <EVAL_FUNCTION>  <COMPARE>
                     (   <NUM_CONSTANT>
                       | <ALPHA_CONSTANT>
                       | <IDENTIFIER>
                     )
                   )

<EVAL_FUNCTION> := "size"
<Hash> := "HASH"
          "("
              <Attribute_Identifier>
             ("," <Attribute_Identifier> )*
          ")"
```

Mit dem Schlüsselwort EVENT wird die Beschreibung eines Dienstgüteereignisses eingeleitet. Das Ereignis wird durch einen eindeutigen Identifikator bezeichnet. Nach dem Schlüsselwort USES wird die Primitive angegeben, aus der sich das beschriebene Ereignis ableiten lässt. Die Instanz der SAP-Ankopplung, die in der späteren Messsicht die bezeichnete Primitive liefern soll, wird nach dem Schlüsselwort ON TAP durch ein Symbol angegeben. Als mögliche Symbole sind

[2] Bei dieser Definition wird davon ausgegangen, dass der Dienstzugangspunkt, in der konkreten Implementierung meist als Socket bezeichnet, bereits erstellt worden ist. An einem geöffneten Socket werden beim Empfang von Daten gewöhnlich die Adressinformationen, welche die Kommunikationrelation bestimmen, an der der Socket teilnimmt, nicht übergeben und sind deshalb hier nicht spezifiziert.

hier maximal zweistellige natürliche Zahlen zugelassen. Der englische Begriff tap ist hier als Entsprechung für das deutsche Wort Ankopplung / Anzapfung gewählt.

Um Ereignisse nur beim Eintreffen von Primitiven mit bestimmten Belegungen zu generieren muss nach dem Schlüsselwort ISOLATE eine Filtervorschrift angegeben werden. Dabei muss für jedes Attribut aus der Definiton der verwendeten Primitve eine Filterregel angegeben werden. Möglich sind dabei Vergleiche der Attributwerte oder einer auf das Attribut angewandten wertgebenden Funktion mit Konstanten. Als wergebende Funktion bietet die vorliegende Implementierung nur die Funktion SIZE an, welche die Länge / Größe einer Variable wiedergibt. Soll nicht nach den Belegungen des Attributs gefiltert werden, wird ein * an die entsprechende Stelle gesetzt.

Nach dem bisherigen Design muss für jedes Ereignis ein eindeutiger Identifikator festgelegt werden, der es erlaubt, die Ereignisse einer Klasse, die von einer SAP-Ankopplung geliefert werden, zu unterscheiden. Dazu wird nach dem Schlüsselwort ID entweder direkt ein Attribut angegeben, aus dem sich dieser Identifikator ableiten lässt (z.B. die Sequenznummer bei TCP), oder der Identifikator wird als Hash-Wert, durch Angabe des entsprechenden Schlüsselworts, eines oder mehrerer Attribute, die als Argumente angegeben werden, erzeugt. Die Bildung eines Hash-Wertes ermöglicht es auch, aus Datenfeldern einen eindeutigen Identifikator zu erzeugen.

Das folgende Beispiel zeigt die Definition eines auf der *data* Primitive aus dem vorherigen Beispiel aufbauenden Ereignisses. Durch die Filtervorschrift werden nur dann Ereignisse erzeugt, wenn die Länge des Datenfeldes größer als 64 ist. Die Identifikation eines Ereignisses geschieht im Beispiel durch einen Hashwert, der aus den Attributen *dest* und *data* gebildet wird.

```
EVENT tcp_foo USES    data
              ON TAP  1
              ISOLATE (*,*,*,*,size > 64)
              ID      HASH(dest,data);
```

4. Konstrukt zur Festlegung von Messwerten

```
<Reading> := "READING"  <Reading_Identifier>
             "REQUIRES" <Event_Identifier>
                  "AS" <Queue_Identifier>
                  ( "," <Event_Identifier>
                  "AS" <Queue_Identifier> )*
             "COMPUTES" <Term>
             "UNIT"     "bit" | "bit/sec"
                      | "sec"
                      | "%"   | "NULL"
             ";"
```

Die Beschreibung eines Messwertes wird durch das Schlüsselwort READING eingeleitet. Dannach folgt ein Bezeichner für den durch den Ausdruck definierten Messwert. Nachfolgend wird, eingeleitet durch REQUIRES, angegeben, welche Ereignisse, genauer Klassen von Ereignissen, für die Berechnung des Messwertes benötigt werden. Vor der Korrelation werden beim Korrelator eingehende Ereignisse in Warteschlangen gepuffert. Wie in Abschnitt 4.3.3.2 beschrieben, erfolgt die Berechnung eines Messwertes erst, wenn in allen angegebenen Warteschlangen Ereignisse bereitstehen. Zur späteren Angabe der Berechnungsvorschrift müssen die jeweiligen Warteschlangen durch einen Identifikator, nach dem Schlüsselwort AS, bezeichnet werden. Ein Messwert kann auf beliebig vielen Klassen von Ereignissen basieren, die Anzahl der nach REQUIRES aufzählbaren Ereignisklassen

und Warteschlangen ist daher nicht beschränkt.

Sind alle benötigten Ereignisse angegeben, wird nach dem Schlüsselwort COMPUTES durch einen Term festgelegt, wie der numerische Messwert aus den vorliegenden Ereignissen zu berechnen ist. Die syntaktischen Möglichkeiten dazu beschreibt der nächste Punkt (5) dieser Aufzählung. Die Einheit, die den ermittelten Messwert kennzeichnet, wird zum Abschluss des Konstrukts nach dem Schlüsselwort UNIT angegeben. Eine Beispieldefinition eines Messwertes wird nach der Beschreibung der Korrelationsmöglichkeiten angegeben.

5. Schlüsselwörter zur Referenzierung von Korrelationsfunktionen

```
<Term> :=   <numGivingEventFunction>
          | <numGivingQueueFunction>
          | <Term> <OPERAND> <Term>
          | "(" <Term> ")"
          | <CONSTANT>

<numGivingEventFunction> :=
            <Event_Identifier>
          | <eventGivingFunction>
            "."
          ( "time" | "id" )

<numGivingQueueFunction> :=
          <Queue_Identifier>
            "."
          ( "count" )

<eventGivingFunction>    :=
           <Queue_Identifier>
            "."
           ("mymatch" | "hismatch")
            "(" <Queue_Identifier> ")"
```

Die Definition der Korrelationsfunktion, die zur Berechnung eines Messwertes verwendet wird, stellt den komplexesten Ausdruck in QoSSL dar, da die wenigen vorhandenen Korrelationsfunktionen beliebig geschachtelt werden können. Zusätzlich können numerische Ergebnisse noch mit den gängigen mathematischen Operationen verknüpft werden.

QoSSL unterscheidet drei grundsätzliche Arten von Korrelationsfunktionen. Solche, die direkt auf Ereignisse angewendet werden können und als Ergebnis einen Wert liefern (numGivingEventFunction), solche, die auf Warteschlangen angewendet werden können und einen Wert liefern (numGivingQueueFunctions) und schließlich solche, die auf Warteschlangen angewendet werden können und ein Ereignis liefern (eventGivingQueueFunctions). Das somit entstehende Repository an Korrelationsfunktionen ist innerhalb dieser Klassifizierung beliebig erweiterbar.

In der hier vorgestellten Implementierung wird der folgende, minimale Satz unterstützt: Die Funktionen time und id sind auf Ereignisse anwendbar und erlauben den Zugriff auf die Attribute eines Ereignisses, dessen Identifikator und die Ereigniszeit. Die Funktion count ist auf eine Warteschlange anwendbar und liefert die Länge dieser Warteschlange resp. die Anzahl der darin enthaltenen Ereignisse zurück. Die Funktionen mymatch und hismatch werden ebenfalls auf Warteschlangen angewandt, benötigen allerdings als Argument eine weitere Warteschlange. Sie erlauben die Ermittlung eines Paares von Ereignissen aus beiden Warteschlangen, das den selben Identifikator aufweist. mymatch liefert das entsprechende Ereignis aus der Warteschlange, von der aus die Funktion aufgerufen wurde, hismatch liefert jenes Ereignis, aus der als Argument übergebenen Warteschlange.

Für das im Folgenden beschriebene Beispiel wird angenommen, daß drei Ereignisse *tcp_syn_sent*, *tcp_syn_ack_received* und *tcp_ack_received* so definiert sind, dass Ereignisse, die den ordnungsgemäßen Aufbau einer bestimmten TCP-Verbindung widerspiegeln, mit dem

selben Identifikator versehen sind. Die Verbindungsaufbauzeit ergibt sich dann wie folgt. Detaillierte Beschreibungen von Spezifikationen für unterschiedlichste Dienstgüteparameter finden sich im Abschnitt 5.1.1.

```
READING tcp_cet_reading
        REQUIRES tcp_syn_sent         AS a,
                 tcp_syn_ack_received AS b,
                 tcp_ack_received     AS c
        COMPUTES    a.myMatch(c).time
                  - a.hisMatch(c).time
        UNIT ms;
```

6. Konstrukt zur Beschreibung der gemessenen Dienstgüte

```
<FEATURE> := "FEATURE" <Feature_Identifier>
             "JOINS"   <NUMCONST>
                       <Reading_Identifier>
             "TO"      <Aggregation>
```

Zur vollständigen Spezifikation eines Dienstgütemerkmals muss noch festgelegt werden, in welcher Weise einzelne Messwerte zusammengefasst werden. Nach dem Schlüsselwort FEATURE (in Anlehnung an das deutsche Wort Merkmal wird hier die direkte Übersetzung ins Englische verwendet) wird die Bezeichnung des Dienstgütemerkmals angegeben. Nach JOINS wird angegeben, wieviele einzelne Messwertee jeweils zusammengefasst werden. Einerseits ergibt sich damit, wie oft (nach wievielen erfolgreich bestimmten Messwerten) die Darstellung der Werte des Dienstgütemerkmals aktualisiert werden. Andererseits ist für die meisten statistischen Aggregationen eine bestimmte Anzahl von Werten erforderlich. Nach dem Schlüsselwort TO kann aus den möglichen Aggregationsfunktionen, wie sie im Folgenden beschrieben sind, ausgewählt werden.

7. Schlüsselwörter zur Referenzierung von Nachverarbeitungsfunktionen

```
<Aggregation> :=   "SUM"
                   "HISTOGRAM"
                 | "DISTRIBUTION"
                 | "BOXPLOT"
                 | "MOVING AVARAGE"
                 | ...
```

In der hier vorgestellten Implementierung stehen grundsätzliche Funktionen zur statistischen Zusammenfassung von Messwerten zur Verfügung, die bei der Spezifikation von Dienstgütemerkmalen (siehe 6) verwendet werden können. Die in der obigen Definition angegebenen Schlüsselwörter sorgen jeweils für die dem Namen entsprechende Aggregation der Messwerte. In Abschnitt 4.3.3.2 wird detailliert auf die Implementierung der jeweiligen Aggregationen eingegangen. Abschließend zeigt der folgende Ausdruck beispielhaft die Festlegung eines Dienstgütemerkmals, das den gleitenden Durchschnitt aus 10 einzelnen Messungen der Verbindungsaufbauzeit bei TCP wiedergibt.

```
FEATURE tcp_connection_time

JOIN 10 tcp_cet_reading TO MAVG;
```

Durch das in Abschnitt 4.3.2.3.2 klar aufgestellte Design der Entwicklersicht konnte kanonisch eine mögliche Sprache zur Spezifikation von Dienstgütemerkmalen abgeleitet werden. Die vorgestellte Sprache QoSSL kommt mit einfachen und zugleich direkt lesbaren Konstrukten aus. Die gesamte Syntax der Sprache, wie sie als Vorlage für die Implementierung des Compilers in Abschnitt 4.3.3.3 dient, ist im Anhang A.2 abgedruckt.

4.4. Zusammenfassung und Bewertung

In diesem Kapitel ist zunächst eine informelle Beschreibung eines generischen Messprozesses zur Umsetzung von Spezifikationen von Dienstgütemerkmal en entwickelt worden. Durch weitere systematische Verfeinerungen und Anwendung formaler Beschreibungstechniken gelang es, unterschiedliche Sichtweisen, die sich bei der Spezifikation und Messung von Dienstgütemerkmalen, abhängig von der jeweiligen Phase des Dienstlebenszyklus, ergeben, zu identifizieren und dabei gleichzeitig Beschreibungsmittel für diese Sichten zu identifizieren bzw. zu entwickeln.

Der hohe Grad an Formalisierung der einzelnen Sichten und die aus der Analyse des Messprozesses gewonnenen Abhängigkeiten zwischen diesen Sichten, die sog. Spezifikationsrichtlinien, bilden die Basis für eine höchst mögliche Rechnerunterstützung bei der Umsetzung einer Dienstgütemerkmalsspezifikation in den einzelnen Phasen des Dienstlebenszyklus.

Ein integriertes Design des für die Rechnerunterstützung verwendeten Automatisierungsprozesses zusammen mit den für die einzelnen Sichten notwendigen Beschreibungstechniken und -elementen bildet die Basis für eine Implementierung, wie sie prototypisch im Abschnitt 4.3.3 durchgeführt wurde. Die klare Abgrenzung zwischen Design und Implementierug ermöglicht es, die Implementierung mit beliebigen Programmiersprachen und/oder Werkzeugen umzusetzen.

Mit dem vorliegenden System wird die gesamte Prozesskette von der Spezifikation bis hin zur Messung von Dienstgütemerkmalen automatisiert. Durch die enge, formale Verknüpfung der einzelnen Sichten, die auch als Stufen dieser Prozesskette entlang des Dienstlebenszyklus betrachtet werden können, kann auf die erneute Festlegung von Informationen für eine bestimmte Sicht verzichtet werden. Alle Sichten werden automatisch, ohne Medienbrüche, aus der Entwicklersicht generiert. Somit entsteht ein umfassendes Konzept zur Spezifikation von Dienstgütemerkmalen und der rechnergestützten Umsetzung dieser Spezifikation im Dienstlebenszyklus, dessen vielfältige Anwendungsmöglichkeiten im folgenden Kapitel dargestellt werden.

Die Tauglichkeit des Konzepts lässt sich aber auch an Hand der in Abschnitt 2.4 aufgestellten Anforderungen bewerten. Eine grafische Darstellung dieser im Folgenden dargestellten Bewertung zeigt Abbildung 4.17. Die Darstellung richtet sich nach der bei der Aufstellung der Anforderungen angegebenen Gliederung und stellt für jede Anforderung im Detail dar, wie diese durch das aufgestellte Konzept abgedeckt wird. In der Abbildung wird der Grad der Anforderungserfüllung entsprechend grafisch wiedergegeben.

1. Spezifikation auf Basis der Analyse von Methodenaufrufen

   Per Konstruktion wird diese Anforderung erfüllt. Jegliche getroffene Spezifikation bezieht sich auf Primitive des Dienstes, also auf Aufrufe von Methoden am SAP.

2. eindeutige Spezifikation

   Die Möglichkeit, den generischen Messprozess an verschiedenen Stellen frei zu parametrisieren (bei der Ereigniskorrelation und der statistischen Aggregation), lässt zunächst Mehrdeutigkeiten zu. Wird aber durch eine Spezifikationssprache auf ein Repository von Verarbeitungsfunktionen zurückgegriffen, kann durch geeignete Wahl dieser Funktionen die Eindeutigkeit einer Spezifikation sichergestellt werden. Letztlich prüfbar wird diese Anforderung nur, wenn die Implementierung der Spezifikationssprache, also ihre Syntaxdefinition vorliegt.

Spezifikation auf Basis der Analyse von Methodenaufrufen an der Dienstschnittstelle	✓
eindeutige Spezifikation	✓🖳
Spezifikation der Semantik möglich	✓
Spezifikation trennt Dienstgütemerkmal von Dienstgüte (Definition von Werten)	✓
Einhaltung der Schichtgrenzen, um Organisationsgrenzen grundsätzlich nicht durchbrechen zu müssen	✓
maschinentaugliche (formale Sprache)	✓🖳
liefert Ergebnisse für den gesamten Lebenszyklus	✓🖳
benötigt nur Information aus der Verhandlungsphase	✓
maximale Lesbarkeit der Spezifikation in der Verhandlungsphase für Customer und Provider vertreten durch natürliche Personen	✓🖳
Ableitung eines Implementierungsleitfadens	✓🖳
explizite Spezifikation der notwendigen Dienstprimitiven	✓
Spezifikation eines Messsystems	✓
Ausdrucksmächtigkeit muss möglichst alle denkbaren anwendugsorientierten Dienstgütemerkmale abdecken	✓🖳
Wiederverwendbarkeit bestehender (Teile) einer Spezifikation	✓🖳

✓ erfüllt ☑ teilweise erfüllt ❍ keine Aussage × nicht erfüllt ≈ Kriterium nicht anwendbar 🖳 durch Architektur / Implementierung

Abbildung 4.17.: Bewertung des aufgestellten Konzepts an Hand der Anforderungen aus Abschnitt 2.4

3. Spezifikation der Semantik

   Der vorgestellte Ansatz ist per Konstruktion darauf ausgelegt, die Semantik eines Dienstgütemerkmals explizit zu machen. Dies geschieht, indem, formalisiert auf Basis eines allgemeinen Messprozesses, angegeben wird, in welcher Weise die Messung des jeweiligen Merkmals abläuft.

4. Trennung von Dienstgüte und Dienstgütemerkmal

   Bereits durch das Konzept, lediglich Dienstgütemerkmale zu spezifizieren, findet inhärent eine Trennung von Dienstgüte und Dienstgütemerkmal statt. Auf Basis von Dienstgütemerkmalsspezifikationen nach dem vorgeschlagenen Konzept kann natürlich eine Spezifikation der (erwarteten) Dienstgüte erfolgen. Diese lässt sich damit einfach durch die Angabe von gültigen Werten resp. Wertebereichen realisieren.

5. Einhaltung von Schichtgrenzen

   Durch die Bindung der Spezifikation an den SAP des Dienstes wird die durch diesen vorgegebene Schichtgrenze automatisch eingehalten. Wie die Modellierung des generischen Messprozesses auf Basis des MNM-Dienstmodells gezeigt hat, werden keinerlei „Durchgriffe" durch die Schichtung einer Dienstimplementierung vorgenommen.

6. maschinentaugliche, formale Sprache

   Mit der Umsetzung des Automatisierungskonzepts durch einen Compiler und der Festlegung der Entwicklersicht als Eingabesprache für diesen Compiler steht die Basis für die Implementierung einer maschinentauglichen, formalen Sprache bereit. Ein konkrete Realisierung des vorgestellten Konzepts ist somit in der Lage, diese Forderung zu erfüllen.

7. Ergebnisse für den gesamten Dienstlebenszyklus

   Die gesamte Entwicklung des Lösungskonzepts ist am Dienstlebenszyklus ausgerichtet, so dass die folgenden Anforderungen inhärent erfüllt sind.

8. nur Informationen aus der Verhandlungsphase notwendig

   Mit der Einführung der Entwicklersicht und der zugehörigen Spezifikationssprache ist formal per Konstruk-

tion festgelegt, dass lediglich Informationen über den Dienst, die in der Verhandlungsphase bekannt sind resp. festgelegt werden können, Eingang in eine Spezifikation finden.

9. Lesbarkeit der Spezifikation

   Durch die Festlegung der Entwicklersicht als minimaler Satz der zur Umsetzung einer Dienstgütemerkmalsdefiniton notwendigen Informationen und ihrer gleichzeitigen Darstellung in einer programmiersprachenähnlichen deklarativen, formalen Sprache wird eine sehr gute Lesbarkeit der Spezifikation erreicht.

10. Ableitbarkeit eines Implementierungsleitfadens

    Die unterschiedlichen, aus der Entwicklersicht ableitbaren Sichten bieten die „Stützpunkte" für einen Implementierungsleitfaden. Zusammen mit den entwickelten Spezifikationsrichtlinien wird dieser Leitfaden bereits soweit umgesetzt, dass die Implementierung, bis auf die Anbindung an die Dienstfunktionalität, vollständig generiert werden kann.

11. explizite Spezifikation der notwendigen Dienstprimitiven

    Sowohl in der Entwicklersicht als auch in der Definitionssicht werden die zur Umsetzung einer Spezifikation benötigten Dienstprimitvien explizit benannt. Damit entsteht ein Gerüst zur Anbindung des Messsystems an die Dienstfunktionalität, das im konkreten Anwendungsfall lediglich ausprogrammiert werden muss.

12. Spezifikation eines Messsystems

    Durch die Konstruktion, die Spezifikation eines Dienstgütemerkmals durch die Parametrisierung eines generischen Messprozesses zu beschreiben, legt diese Spezifikation inhärent ein geeignetes Messsystem fest. Der eingeführte Automatisierungsprozess ist zudem in der Lage, dieses Messsystem automatisch aus der Spezifikation eines Dienstgütemerkmals abzuleiten.

13. maximale Ausdrucksmächtigkeit

    Die Ausdrucksmächtigkeit einer Spezifikationssprache, welche die Vorgaben des vorgestellten Konzepts umsetzt, hängt von den durch die Sprache vorgegebenen Funktionen für Korrelation und Nachverarbeitung ab. Diese Anforderung kann damit nur bei Vorlage einer konkreten Sprache bewertet werden.

14. Wiederverwendbarkeit

    Die Definitionssicht eines Dienstgütemerkmals wird durch Verfeinerung eines bestehenden Modells erstellt. Alle Spezifikationen ordnen sich damit in die Gliederung dieses Basismodells ein. Die Implementierung einer Spezifikationssprache kann damit die Möglichkeit bieten, bestehende Definitionen, auch von Teilaspekten, in Bibliotheken zusammenzufassen. Wird gleichzeitig die Möglichkeit geboten, diese Bibliotheken mit einem Spezifikationsmuster zu durchsuchen, entsteht ein hoher Grad an Wiederverwendbarkeit bestehender Spezifikationen.

Die abschließende Betrachtung des vorgestellten Konzepts im Vergleich zu den aufgestellten Anforderungen an eine Lösung hat gezeigt, dass diese optimal abgedeckt werden, wenn eine geeignete Implementierung des Lösungskonzepts gewählt wird. Zugleich bietet das Konzept aber auch die Möglichkeit, zunächst „einfachere" Implementierungen aufzustellen, die sich nachträglich erweitern lassen. Dieser Ansatz wurde in Abschnitt 4.3.3 verfolgt.

5. Anwendungs- und Erweiterungsmöglichkeiten

In diesem Kapitel werden Anwendungsmöglichkeiten für das in Kapitel 4 eingeführte Spezifikationskonzept und die darauf aufbauenden Generierungsmöglichkeiten beschrieben. Dazu werden zunächst in Abschnitt 5.1 auf die einzelnen Phasen des Lebenszyklus beschränkte und schließlich, in Abschnitt 5.2, phasenübergreifende Verwendungsmöglichkeiten für die beschriebenen Konzepte aufgezeigt. Ausblickend werden daran anschließend in Abschnitt 5.3 Möglichkeiten zur Anwendung des Spezifikationskonzepts und seiner nachgeordneten Methoden vorgestellt, wenn der Fokus von Konzept und Methoden geringfügig erweitert wird. Eine Bewertung des Spezifikationskonzepts zusammen mit seinen Erweiterungs- und Anwendungsmöglichkeiten in Abschnitt 5.4 rundet das Kapitel ab.

5.1. Phasenspezifische Aufgaben

Gegliedert nach den einzelnen Phasen des Lebenszyklus werden im Folgenden Möglichkeiten aufgezeigt, wie für die jeweilige Phase spezifische Aufgaben durch die Anwendung des eingeführten Spezifikationskonzepts und seiner nachgelagerten Generierungsschritte unterstützt resp. automatisiert werden können. Somit wird auch die Umsetzung der in Kapitel 4 eingeführten Konzepte exemplarisch deutlich.

5.1.1. Verhandlungs- und Designphase

Die Spezifikation der gewünschten resp. benötigten Dienstgüte und der zugehörigen Dienstgütemerkmale ist neben der Festlegung der Funktionalität eines Dienstes die wichtigste Aufgabe in der Verhandlungs- und Designphase. Sind entsprechende Festlegungen für Dienstfunktionalität und -güte vorhanden, kann an Hand dieser Festlegungen auch aus einem vorhandenen Angebot von Diensten im Sinne des Tradings ausgewählt werden. Alle diese beschriebenen Fälle lassen sich durch das eingeführte Spezifikationskonzept unterstützen, wie die folgende Darstellung zeigt.

Formale Spezifikation von Dienstgütemerkmalen Wird zur Spezifikation von Dienstgütemerkmalen beispielsweise auf die in Abschnitt 4.3.3.4 umgesetzte Spezifikationssprache QoSSL (Quality of Service Specification Language) zurückgegriffen, lassen sich Dienstgütemerkmale wie folgt beschreiben. Die Beispiele verdeutlichen die Struktur einer formalen Dienstgütespezifikation nach dem in Kapitel 4 eingeführten Konzept. Die jeweilige syntaktische Ausprägung der Spezifikationssprache, der sog. syntaktic suggar, hängt von der jeweiligen Implementierung dieser Sprache ab. Beispielhaft werden hier die zu Beginn von Kapitel 4 in Abschnitt 4.2.1.2 angegebenen informellen Definitionen der Framefehlerrate und der Antwortzeit des Konfigurations- und Bestelldienstes in QoSSL dargestellt.

Framefehlerrate bei Ethernet

```
PRIMITIVE eth_recv (  data:       BYTE[]
                      crc:        BYTE[4]
                   );


EVENT recv_ok USES    eth_recv
              ON TAP  1
              ISOLATE (*,==crc(data))
              ID      AUTO;


EVENT recv_faulty USES    eth_recv
                  ON TAP  1
                  ISOLATE (*,!=crc(data))
                  ID      AUTO;


READING eth_faulty_frame
        REQUIRES recv_ok         AS a,
                 recv_faulty     AS b,
        COMPUTES b.count / a.count * 100
            UNIT %;


FEATURE eth_faulty_frame_rate

JOIN 10 eth_faulty_frame TO MAVG;
```

Die Definition der Framefehlerrate baut auf nur eine einzige Primitive, den Empfang eines Rahmens auf. Stimmt der im Rahmen gespeicherte CRC mit dem lokal berechneten CRC überein, wird ein Ereignis erzeugt, das den korrekten Empfang eines Rahmens anzeigt. Entsprechend wird bei fehlerhaften CRC ein Ereignis erzeugt, das anzeigt, dass ein fehlerhafter Rahmen empfangen wurde.

Die Ereignisse werden korreliert, indem ihre Anzahl zueinander ins Verhältnis gesetzt wird. Diese Rechenvorschrift wird nur ausgeführt, wenn von jedem Ereignis mindestens eine Instanz zur Auswertung vorliegt. Zur Nachverarbeitung wird ein gleitender Durchschnitt aus zehn erfolgten Einzelmessungen berechnet.

Antwortzeit

```
PRIMITIVE service_req (      data:   BYTE[]
                          req_type:   BYTE[2]
                            req_id:   BYTE[4]
                       );


PRIMITIVE service_rep (      data:   BYTE[]
                          rep_type:   BYTE[2]
                            req_id:   BYTE[4]
                       );


EVENT a_service_req USES    service_req
                    ON TAP  1
                    ISOLATE (*,*,*)
                    ID      req_id;


EVENT a_service_rep USES    service_req
                    ON TAP  1
                    ISOLATE (*,*,*)
                    ID      req_id;


READING rep_time
        REQUIRES a_service_req     AS a,
                 a_service_rep     AS b,
        COMPUTES a.myMatch(b).time
                 - a.hisMatch(b).time
        UNIT ms;


FEATURE rep_time_stats

JOIN 50 tcp_cet_reading TO BOXPLOT;
```

Zur Messung der Antwortzeit sind zwei Primitiven erforderlich: Eine die angibt, dass eine Funktionalität angefordert wird, und eine weitere, die anzeigt, dass die Rückgabe von Werten nach einer Anfrage erfolgt. Für die hier dargestellte Definition wird davon ausgegangen, dass der Dienst für Anfrage und Rückgabe jeweils genau eine Primitive anbietet.

Beim Auftreten dieser Primitiven werden entsprechende Ereignisse generiert. Da der Aufruf der Dienstfunktionaliät asynchron erfolgt, trägt jede Anfrage und jede Antwort einen eindeutigen Identifikator. Dieser wird übernommen, um das resultierende Ereignis eindeutig auszeichen zu können.

Zur Korrelation werden beide Ereignistypen in unterschiedliche Warteschlangen eingeordnet. Aus diesen werden Ereignispaare mit gleichem Identifikator (durch die Funktionen hisMatch() und myMatch()) ausgewählt. Die Zeitstempel beider Ereignisse werden voneinander abgezogen, so dass der resultierende Wert die Antwortzeit für genau eine Anfrage an den Dienst anzeigt.

Zur statistischen Zusammenfassung werden immer 50 dieser Werte in einem Boxplot zusammen gefasst. Dieser wird nicht grafisch dargestellt, sondern es werden die typischen Stützwerte (Median, erstes und drittes Quartil sowie 95% und 5% Quantil) angegeben.

Aus den obigen Darstellungen lässt sich erkennen, dass die Spezifikation von Dienstgütemerkmalen mit der eingeführten formalen Sprache geeignet ist, eindeutige Festlegungen von Dienstgütemerkmalen ohne freie Interpretationsmöglichkeiten zu treffen. Die Anwendung von QoSSL als Implementierung der in Abschnitt 4.3.2.3.2 eingeführten Entwicklersicht zeigt damit deren Fähigkeit, Dienstgütemerkmalsspezifikationen zu formalisieren und eindeutig zu definieren.

Festlegung der gewünschten / benötigten Dienstgüte Ist die Definition von Diennstgütemerkmalen vorgenommen, können an diese Werte oder Wertebereiche gebunden werden, um die gewünschte bzw. benötigte Dienstgüte formal festzulegen. Durch die klare Definition eines Dienstgütemerkmals in der Entwicklersicht (siehe Abschnitt 4.3.2.3.2) kann auch die Dienstgüte formal eindeutig festgelegt werden.

Unterstützung des Tradings Ist die benötigte Dienstgüte festgelegt, kann aus einem bestehenden Angebot an (standardisierten) Diensten ausgewählt werden. Das im Folgenden dargestellte Beispiel verdeutlicht diesen Vorgang: Mehrere Dienste gleicher Funktionalität, z.B. IP-Connectivity, stehen einem Customer zur Auswahl. In einer Darstellung, wie sie der in [GHHK 02] eingeführten Nomenklatur entspricht, steht also eine Instanz der *Customer-View* mehreren Instanzen des *side-independent*-Teils, eben mehreren Diensten, gegenüber. Entsprechend werden diese unterschiedlichen Dienste von verschiedenen Implementierungen erbracht. Abbildung 5.1 zeigt eine beispielhafte Instanzierung des MNM-Dienstmodells in der Beschränkung auf die *Customer-View* und, grau hinterlegt, den *side-independent*-Teil.

Dienste in solchen Szenarien unterscheiden sich durch die gebotene Managementfunktionalität sowie die gebotene Dienstgüte und die gebotenen Dienstgütemerkmale. Stellen diese Dienste Spezifikationen der möglichen Dienstgüte und Dienstgütemerkmale bereit, können unterschiedlichste Angebote gegeneinander verglichen und derjenige Anbieter mit der „passenden" Spezifikation ausgewählt werden. Da die Spezifikationen auch syntaktisch formalisiert sind, kann diese Auswahl völlig automatisch geschehen.

Da die Spezifikation von Dienstgütemerkmalen nach den vorgeschlagenen Konzepten nur auf den Schnittstellen eines Dienstes aufbaut, kann sie unter Einhaltung der durch die Funktionalität vorgegebenen Schnittstellen auch über Organisationsgrenzen ausgetauscht werden. Die Erbringung der Dienstgüte bleibt, ebenso wie die Erbringung der Funktionalität, transparent. Provider können damit Sub-Dienste anbieten, ohne Einblicke in die gewählte Implementierung gewähren zu müssen.

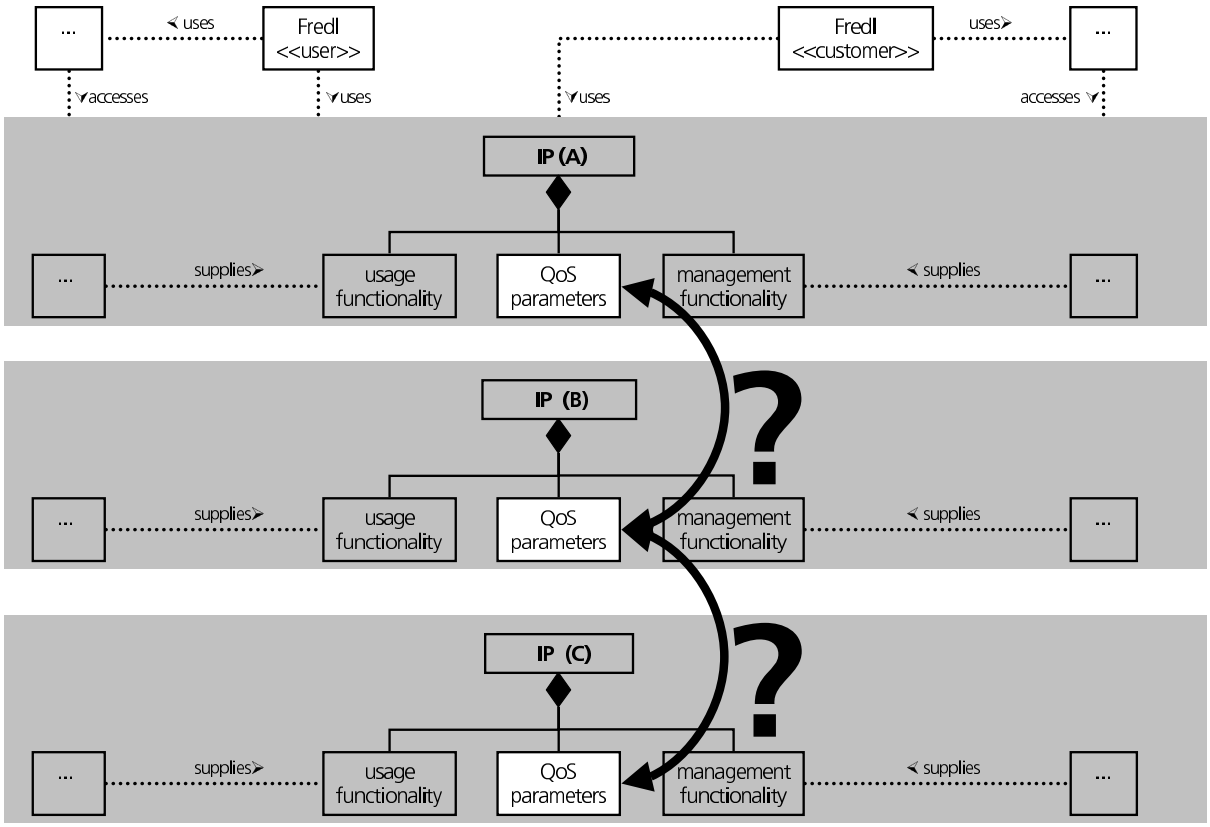

Abbildung 5.1.: Auswahl von Diensten, dargestellt mit dem MNM-Servicemodell

Wird zudem eine Vergleichsmetrik von Dienstgütemerkmalsbeschreibungen etabliert, wie in Abschnitt 5.2 vorgeschlagen, kann auch dann noch ein Dienstangebot ausgewählt werden, wenn keines der Angebote direkt den Forderungen entspricht. An hand der Vergleichsmetrik lässt sich immer ein Dienst bestimmen, der den geforderten Eigenschaften am nächsten kommt.

5.1.2. Bereitstellungsphase

Mit der Auswahl eines geeigneten (Sub-)Dienstes, wie oben beschrieben, befindet sich der Dienstlebenszyklus bereits am Übergang in die Bereitstellungsphase. Um ein Dienstgütemanagement etablieren zu können, müssen zunächst geeignete Systeme zur Messung der erbrachten Dienstgüte aufgebaut werden.

Generierung eines Messsystems Durch die Anwendung des in Abschnitt 4.3.2.1 vorgeschlagenen Automatisierungsprozesses lässt sich ein Großteil des notwendigen Messsystems in Form der Messsicht generieren. Wird der Automatisierungsprozess, wie in Abschnitt 4.3.3 beschrieben, durch einen Compiler implementiert, ist sogar die automatische Erstellung von ablauffähigem Programmcode möglich.

Leitfaden zur Instrumentierung Zum vollständigen Aufbau des Messsystems muss noch eine Instrumentierung der Dienstschnittstelle erfolgen, damit Aufrufe von Dienstmethoden durch das Messsystem registriert werden können. Da in der Spezifikation der Dienstgütemerkmale bereits explizit angegeben ist, welche Primitiven des Dienstes zur Erfassung der festgelegten Dienstgütemerkmale benötigt werden, lässt sich für die Instrumen-

tierung automatisch eine Schnittstelle zum Messsystem (in Form der notwendigen Primitiven) festlegen.

Die Instrumentierung kann durch den Aufbau auf eine vorhandene Middleware oder eine häufig verwendete API deutlich vereinfacht werden, weil dann nicht jede einzelne Implementierung instrumentiert werden muss. Die praktische Umsetzung dieses Vorgehens bietet sich beispielsweise für die Java.net API an.

Bietet sich keinerlei Möglichkeit, eine vorhandene Implementierung direkt am Dienstschnitt zu instrumentieren, lassen sich die für die Erfassung der Dienstgüte notwendigen Primitiven häufig aus der Datenkommunikationsanalyse am Protokollschnitt rekonstruieren.

Aufbau des Dienstmanagements Ist das Messsystem vollständig erstellt, können Funktionen des Dienstgütemanagements auf Basis dieses Systems aufgebaut werden. Die Sicherstellung der festgelegten Dienstgüte kann nur mit genauer Kenntnis der gewählten Implementierung erfolgen. Ein entsprechendes Managementsystem ist damit praktisch nicht automatisch zu erzeugen. Reaktionen auf die Veränderung der aktuell erbrachten Dienstgüte hängen von Vereinbarungen im SLA ab, die sich nicht direkt auf die Funktion oder die Qualität des Dienstes beziehen. Zur Abschätzung von Qualitätsschranken, die aus Sicht des Providers sinnvoll, also noch einzuhalten sind, kann eine Simulation des Dienstes in Bezug auf seine Dienstgüteeigenschaften verwendet werden, wie sie im Abschnitt 5.3 beschrieben wird.

5.1.3. Nutzungsphase

In der Nutzungsphase steht das generierte Messsystem zusammen mit der eventuell erstellten Instrumentierung der Dienstschnittstelle zur Verfügung, so dass während des Betriebs die Messung der aktuell erbrachten Dienstgüte erfolgen kann. Die ermittelten Messwerte können einerseits als Eingabe für das Dienstmanagement verwendet werden und andererseits an der CSM-Schnittstelle dem Customer zur Verfügung gestellt werden.

Unterstützung des Reportings Die in der Verhandlungsphase vorgenommene Spezifikation der Dienstgütemerkmale (z.B. mit QoSSL) kann bei diesem Reporting wieder als Bezugspunkt verwendet werden. Da dem Customer die Definition der Dienstgütemerkmale bekannt ist, genügt es, an der Reporting-Schnittstelle des CSM (Customer Service Management) [Lang 01, Nerb 01] die aktuell gemessenen Werte, zusammen mit der Bezeichnung des jeweiligen Dienstgütemerkmals, auszutauschen. Die eindeutige Festlegung der Dienstgütemerkmale, unabhängig von den gemessenen Werten, erleichtert die Weiterverarbeitung der Messwerte im Managementsystem des Customers, insbesondere, wenn dieser seinerseits wieder als Provider auftritt.

Dynamischer Aufbau von Dienstketten Im Fall des dynamischen Aufbaus von Dienstketten wird besonders deutlich, wie eine formale Spezifikation der Dienstgütemerkmale die Abläufe erleichtert. Das Konzept des CSM wurde eingeführt, um ein Dienstmanagement über Organisationsgrenzen hinweg zu ermöglichen. CSM dient dabei als Rahmenwerk, das alle Phasen des Dienstlebenszyklus mit entsprechenden Managementschnittstellen unterstützt. Für den Bereich der Dienstgüte sind in der Betriebsphase Reportingschnittstellen vorgesehen. Wird dieses Reporting wie vorgeschlagen formalisiert, ist ein "ad-hoc" Zugriff auf einen Dienst bei gleichzeitig weiter betriebenem Management möglich, wie das folgende Beispiel zeigt:

Das LRZ tritt als Customer gegenüber der T-Systems auf und nutzt dort Connectivity-Dienste. Bei einem Ausfall dieser Dienste

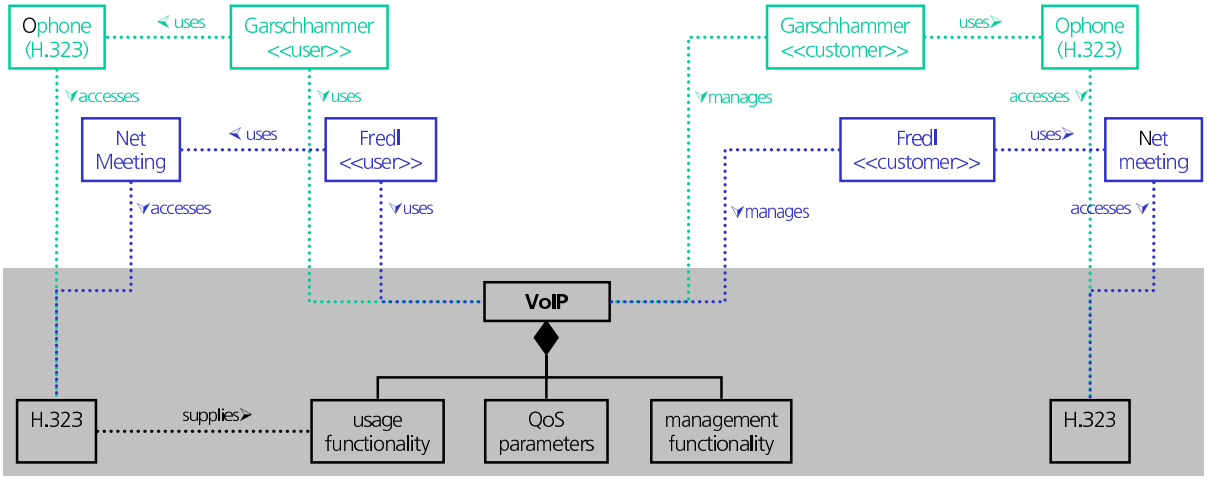

Abbildung 5.2.: ideale Sicht auf einen Endnutzer-Dienst (VoIP), dargestellt mit dem MNM-Service-Modell

kann alternativ ein lokaler Anbieter (M-Net) einspringen. Beide Anbieter stellen eine Dienstgüteüberwachung über CSM zur Verfügung. Da sie aber unterschiedliche Implementierungen für den Dienst verwenden, sind die zur Verfügung gestellten Messwerte nicht direkt vergleichbar. Weiterhin unterscheidet sich das Spektrum der zur Verfügung gestellten Messwerte. Somit ist eine Anpassung des Dienstmanagements an den jeweils benutzten Provider notwendig, die händisch erfolgen müsste, wenn keine formale Beschreibung der verfügbaren Dienstgütemerkmale vorläge.

Endnutzerorientiertes Dienstgütemanagement Die Möglichkeit zum formal gesicherten Austausch der Dienstgüte und der zugrunde liegenden Merkmalsdefinitionen erleichtert nicht nur das Dienstmanagement in Providerketten, wie eben dargestellt, sondern ermöglicht es auch, ein Dienstgütemanagement von Ende-zu-Ende-Diensten über Organisationsgrenzen hinweg zu etablieren.

Aus Sicht eines Endnutzers wird, wie im Beispiel VoIP, ein Dienst organisationsübergreifend erbracht, so dass es ebenfalls möglich wird, dass verschiedene Nutzer aus unterschiedlichen Organisationseinheiten ein und denselben Dienst nutzen. Abbildung 5.2 zeigt diese Sicht der Endnutzer auf den Dienst.

Eine Betrachtung des selben Anwendungsfalls aus Sicht der beteiligten Provider zeigt ein deutlich anderes Bild: Für jede Organisationseinheit wird mindestens eine eigene Dienstinstanz betrieben. Die Kopplung dieser Instanzen erfolgt dann durch Sub-Dienste (im Beispiel über mehrere Instanzen des IP-Diensts). Abbildung 5.3 zeigt die beschriebene Situation als Instanzmodell des MNM-Dienstmodells. Zur Veranschaulichung sind die Grenzen zwischen unterschiedlichen Organisationseinheiten deutlich hervorgehoben.

Über die Kopplung durch Sub-Dienste beeinflussen sich die Dienstmanagement-Instanzen in den unterschiedlichen Domänen gegenseitig. Diese Kopplung wird allerdings erst wirksam, wenn tatsächlich eine Dienstnutzung stattfindet. Um ein für die Nutzer durchgängiges Management zu etablieren, müssen sich alle beteiligten Provider über die jeweiligen Dienstgütemerkmale und ihre Werte austauschen können (Pfeil in Abbildung 5.3). Die einheitliche Wahrnehmung **eines** Dienstes durch die Endnutzer ist also nur möglich, wenn das Dienstgütemanagement der unterschiedlichen Dienstinstanzen eng aneinander gekoppelt wird.

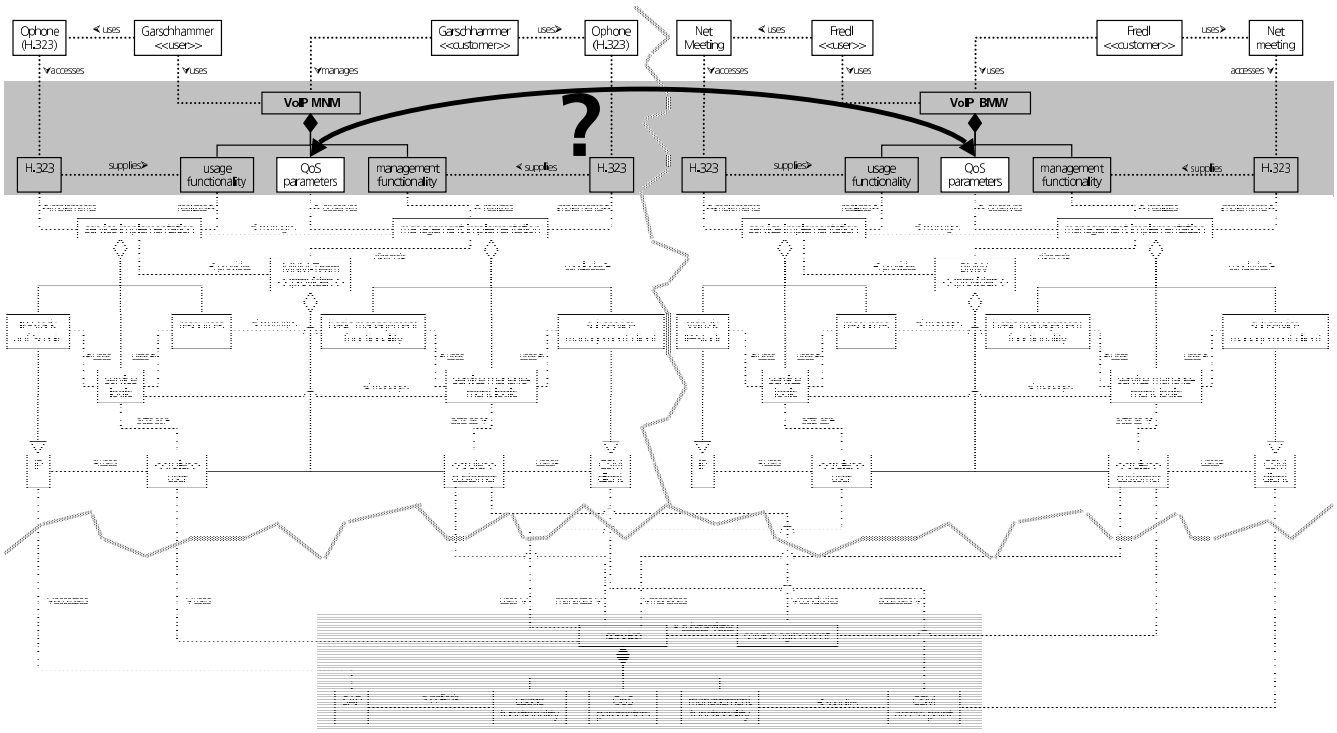

Abbildung 5.3.: reale Sicht auf einen Endnutzer-Dienst (VoIP), dargestellt mit dem MNM-Service-Modell

Wird die vorgeschlagene formale Beschreibung von Dienstgütemerkmalen verwendet, lassen sich die Dienstgütemerkmale, die den beiden Nutzern resp. Customern geboten werden, vergleichen. Über die Reportingschnittstelle im CSM können sowohl Informationen über die Dienstgüte als auch über die messbaren Dienstgütemerkmale ausgetauscht werden. Zudem bietet sich die Möglichkeit, aus der Menge der gebotenen Dienstgütemerkmale einen Satz auszuwählen, der von beiden Providern unterstützt wird, so dass ein Ende-zu-Ende-Management dieser Merkmale möglich wird. Ebenso könnte die in Abschnitt 5.2 eingeführte Vergleichsmetrik als Basis für ein Mapping zwischen unterschiedlichen Merkmalsdefnitionen in unterschiedlichen Organisationsdomänen verwendet werden, so dass wiederum ein einheitliches Ende-zu-Ende-Management möglich wird.

5.2. Phasenübergreifend / -unabhängig

Zusätzlich zu den vorgestellten phasenspezifischen Anwendungsmöglichkeiten können die vorgestellten Konzepte auch Aufgaben unterstützen, die sich nicht direkt einer Phase des Dienstlebenszyklus zuordnen lassen. Die folgende Darstellung zeigt exemplarisch einige dieser Anwendungsmöglichkeiten auf.

Klassifikationsschema für Dienstgütemerkmale Die feingranulare Modellierung eines Dienstgütemerkmals in der Definitionssicht bietet die Möglichkeit zum Aufbau eines Klassifikationsschemas. Dieses richtet sich an den Komponenten der Definitionssicht (Ereignisgenerator, Ereigniskorrelator usw.) aus. Dienstgütemerkmale lassen sich demnach z.B. an hand der Ausprägungen der jeweiligen Elemente der Definitionssicht, der Anzahl der Ereignistypen, des verwendeten Kor-

relationsschemas oder der verwendeten Nachverarbeitungsfunktion klassifizieren. Zusätzlich bietet sich die Möglichkeit, eine Abstandsmetrik zu definieren, welche die Unterschiede bestehender Definitionen numerisch fasst. Diese wird als Erweiterungsmöglichkeit in Abschnitt 5.3 beschrieben.

Modellierung bestehender Systeme Ebenso wie das MNM-Dienstmodell [GHHK 02] kann auch das eingeführte Modellierungskonzept für Dienstgüte und Dienstgütemerkmale verwendet werden, um bestehende Systeme zu modellieren und damit das Verständnis für die Abläufe eines bestehenden Dienstes resp. einer bestehenden Implementierung zu verbessern. Da sich das eingeführte Modellierungskonzept nahtlos in das MNM-Dienstmodell integriert, kann die Analyse der dienstgüterelevanten Begrifflichkeiten in einem Zug mit der Analyse der funktionalen Gliederung eines Dienstes durchgeführt werden. Durch den modularen Aufbau des Gesamtmodells ist es aber ebenso möglich, lediglich die dienstgüterelevanten Aspekte zu untersuchen.

Die Modellierung nach dem eingeführten Konzept kann damit als Diskussionsbasis verwendet werden, um bestehende Dienste in ihrem Dienstgütemanagement miteinander zu vergleichen. Zudem bietet sich die Möglichkeit des Know-How-Transfers auf einer formal gesicherten Ebene. Zusätzlich können Provider Modelle der erbrachten Dienste um Aussagen zur Dienstgüte erweitern und so die Verhandlungsbasis weiter formalisieren.

5.3. Erweiterte Anwendungsmöglichkeiten

Neben den direkten, in den beiden vorhergehenden Abschnitten vorgestellten Anwendungsmöglichkeiten ergeben sich auch Möglichkeiten zur Erweiterung der vorgestellten Konzepte. Die folgenden Abschnitte beschreiben einige dieser Erweiterungen und zeigen damit, wie sich die eingeführten Konzepte als universelle Basis für vielfältige Aufgaben des Dienstgütemanagements verwenden lassen.

Ausweitung des Diensgütebegriffs Bei der Festlegung der in dieser Arbeit untersuchten Fragestellung in Kapitel 2 wurde deren Fokus an hand des Dienstgütequaders auf die Ebene der anwendungsorientierten Dienstgütemerkmale und zugleich auf den Funktionsaspekt gelegt. Es bietet sich an, das Lösungskonzept auf den gesamten Dienstgütequader auszudehnen.

Für die Dimension der Dienstgüteebenen ist dies, besonders für die Ebene der Ressourcen, einfach möglich, wenn diese grundsätzlich mit einem funktionalen Interface ähnlich dem eines einfachen Dienstes modelliert werden. Für nutzerorientierte Dienstgütemerkmale ist eine analoge Erweiterung nur bedingt denkbar, da auf dieser Ebene der Abstraktionsgrad häufig so gewählt ist, dass eine Bindung eines Dienstgütemerkmals an bestimmte, funktionale Methoden des Dienstes unmöglich ist.

In der Dimension der Aspekte ergibt sich kanonisch die Möglichkeit, auch Dienstgütemerkmale mit Managementaspekt mit den eingeführten Konzepten zu modellieren. Anstatt an funktionale Methoden des Dienstes wird hier die Spezifikation eines Dienstgütemerkmals an Methoden des Managementinterfaces gebunden. Damit lassen sich dann Dienstgütemerkmale, wie z.B. die Hotlineverfügbarkeit (sie-

he Abschnitt 1.1), analog zur Verfügbarkeit des Dienstes modellieren.

Ebenso bietet sich die Möglichkeit, bestehende Konzepte des Managements der funktionalen Dienstgüte auf Managementaspekte zu übertragen. Diese Symmetrie von Funktionalität und Management wurde bereits beim Aufbau des MNM-Dienstmodells konsequent umgesetzt und spiegelt sich nun erneut in der (möglichen) Definiton von Dienstgütemerkmalen wider.

Die Ausweitung des vorgestellten Spezifikationskonzepts und seiner Umsetzung im Dienstlebenszyklus auf Dienstgütemerkmale mit Inhaltsaspekt ist kanonisch nicht möglich, aber denkbar, wenn beispielsweise die vom Ereigniskorrelator gebotenen Korrelationsfunktionen erweitert werden. Diese Möglichkeit wird in Abschnitt 6.3 angedacht.

Ausgangspunkt zur Analyse der Mapping-Problematiken Nach dieser Ausweitung des Anwendungsbereichs deckt das vorgestellte Spezifikationskonzept mit seinen nachgeordneten Methoden, wie in Abbildung 5.4 durch Einfärbung gezeigt, den funktionalen und den Managementaspekt von Dienstgütemerkmalen, die Ebene der ressourcen- und der anwendungsorientierten Dienstgütemerkmale sowie den gesamten Dienstlebenszyklus ab. Somit kann es als Basis für die Analyse der Mapping-Problematiken zwischen den einzelnen „Teilwürfeln", die es abdeckt, verwendet werden.

Damit ist es (im einfachsten Fall empirisch) möglich zu untersuchen, welchen Einfluss die ressourcenorientiert gemessene Dienstgüte auf anwendungsorientierte Merkmale hat. Ebenso ist eine Analyse der umgekehrten Abbildung möglich. Da auf allen Ebenen dasselbe Modellierungskonzept verwendet wird, ist ein erstes Problem bei der Analyse der Mapping-Problematiken, nämlich unterschiedliche Beschreibungstechniken auf unterschiedlichen Ebenen, bereits per Konstruktion gelöst.

Accounting Der vorgestellte Ansatz zur Modellierung und Spezifikation von Dienstgütemerkmalen kann ebenso als Basis für ein detailliertes Accounting (im Sinn der Dienstnutzungserfassung) verwendet werden. Das generierte Messsystem ist bereits in der Lage, spezifische Ereignisse an der Dienstschnittstelle zu registrieren und zu korrelieren. Anstatt zur Berechnung von Werten für Dienstgütemerkmale kann dieses Verfahren auch zur Berechnung und Erfassung von Dienstnutzungsdaten verwendet werden. Lediglich die Nachverarbeitungsmethoden müssten angepasst werden.

Damit lassen sich mit ein und derselben Instrumentierung sowohl Werte für die Dienstgüte als auch Aussagen über die erfolgte Dienstnutzung gewinnen. Zudem bietet sich die Möglichkeit, beide Systeme zu koppeln und somit ein dienstgüteorientiertes Accounting zu etablieren. Die Erfassungsgranularität des Messsystems ist dabei so hoch, dass jeder Dienstaufruf in seiner Qualität einzeln erfasst werden könnte. Soll ein entsprechendes System produktiv eingesetzt werden, müssen allerdings Implementierungsfragestellungen, wie die erwartete Performance oder das zu bewältigende Datenvolumen untersucht, werden.

Klassifikation von Merkmalen durch den Grad der Wiederverwendung Bereits in Abschnitt 5.2 wurde die Möglichkeit vorgestellt, Definitionen von Dienstgütemerkmalen an Hand der Ausprägung ihrer Definitionssicht zu klassifizieren. Zusätzlich bietet sich hier die Möglichkeit, eine Abstandsmetrik zu definieren und diese in unterschiedlicher Gewichtung an die einzelnen Elemente der Definitionssicht zu binden.

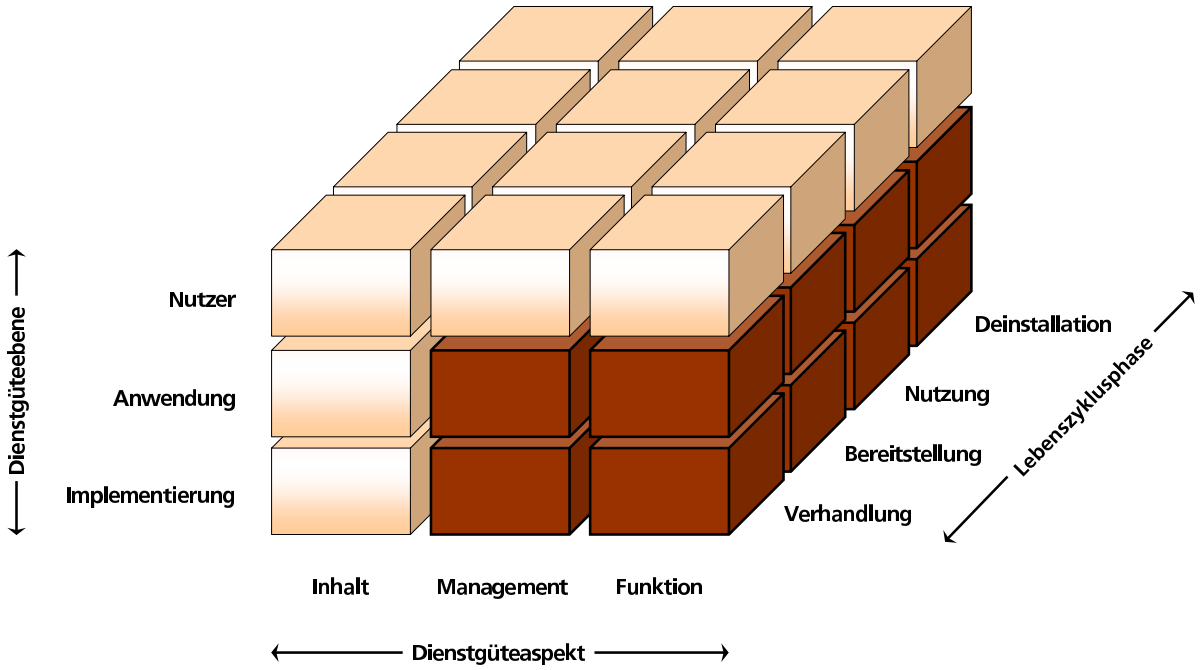

Abbildung 5.4.: Erweiterte Anwendungsmöglichkeiten, dargestellt im Dienstgütequader

So ist es z.B. denkbar, Dienstgütemerkmale zu erst nach den verwendeten Ereignissen, dann nach der Korrelationsmethode und der Nachverarbeitung zu klassifizieren. Ein genau umgekehrtes Schema ist ebenso denkbar. Ersteres empfiehlt sich für Provider als Unterstützung um Aussagen darüber zu treffen, welche Dienstgütemerkmale mit einer bestehenden Implementierung und Instrumentierung überhaupt spezifizier- und messbar sind.

Wird das Klassifikationsschema benutzt, um die bestmögliche Übereinstimmung zweier Definitionen bei der (dynamischen) Auswahl eines Dienstes zu bestimmen, so erscheint es sinnvoll, die Klassifizierung zunächst entlang der verwendeten Korrelationsfunktionen aufzubauen, da diese ein Dienstgütemerkmal maßgeblich beeinflussen.

Auf jeden Fall ist es für einen Provider, aber auch providerübergreifend, sinnvoll, einen Katalog aller bestehenden Definitionen von Dienstgütemerkmalen zu erstellen und diesen nach den dargestellten Klassifizierungsmöglichkeiten zu gliedern. Damit kann, je nach Anwendungsfall, einfach auf eine bereits bestehende Definition zurückgegriffen werden.

Simulation der Dienstgüteabhängigkeiten Wird ein Dienst durch die Zusammenführung mehrerer Dienste, beispielsweise in einer Dienstkette, erbracht, so ist es in der Verhandlungsphase kaum möglich, abzuschätzen, welche Dienstgüte untergeordnete Dienste liefern müssen, um die am oberen Ende der Dienstkette, zum Endkunden hin, vereinbarte Dienstgüte zu erreichen.

Mit dem eingeführten Spezifikationskonzept bietet sich die Möglichkeit, diesen Prozess zu simulieren, sobald die Implementierung oder zumindest die Dienstlogik des eigenen Dienstes festgelegt ist. Subdienste können dann in ihren Schnittstellen simuliert und gleichzeitig die Dienstgüte dieser Simulationen (mit dem selben Messsystem wie für eine reale Implementierung) erfasst werden.

Auf diese Weise entsteht eine Simulation der Dienstkette, die iterativ an die gewünschte Dienstgüte angepasst werden kann. Damit bietet sich die Möglichkeit,

die Mapping-Problematik der Diensthierarchien empirisch zu lösen, ohne dass eine Implementierung der gesamten Hierarchie notwendig wäre. Wird dieser iterative Prozess durch geeignete Werkzeuge unterstützt, existiert ein mächtiger Ansatz zur Erleichterung der Dienstgütespezifikation in Diensthierarchien.

5.4. Bewertung

Die Darstellung der möglichen Anwendungen des eingeführten Spezifikationskonzepts und seiner nachgelagerten Methoden in den vorhergehenden Abschnitten hat dessen vielfältige Einsatzmöglichkeiten aufgezeigt. Das ursprünglich für Dienstgütemerkmale der Anwendungsebene mit Funktionsaspekt entwickelte Konzept lässt sich dank seiner generischen Auslegung auf weitere Bereiche des Diensgütequaders ausdehnen, wie aus Abbildung 5.4 ersichtlich. Per Konstruktion bietet das Spezifikationskonzept mit seinen Automatisierungsmethoden optimale Unterstützung für den gesamten Dienstlebenszyklus.

Die Effektivität des vorgestellten Konzepts ist in der vorangegangenen Schilderung seiner Anwendungsmöglichkeiten aufgezeigt. Seine Effizienz hängt in weiten Teilen von der gewählten Implementierung ab. Die prototypische Umsetzung, wie sie im Rahmen dieser Arbeit vorgenommen wird, lässt die Sicherstellung der Performance außer Acht.

Bei der Umsetzung der Messsicht in eine konkrete Implementierun bieten sich allerdings alle Möglichkeiten, eine Lösung zu wählen, die optimal auf Performanceaspekte abgestimmt ist, da die Messsicht trotz ihres Detaillierungsgrades immer noch implementierungsunabhängig gehalten ist. Der höchste Aufwand bei der Implementierung eines Messsystems liegt, Dank der Generierung großer Teile des Messsystems, in der Instrumentierung der funktionalen Dienstschnittstellen. Die Implementierung des Generators kann ebenfalls, durch die Verwendung eines Compilergenerators, vereinfacht werden, allerdings muss in dieser Phase besonders auf die resultierende Implementierung des Messsystems geachtet werden, um dessen spätere Performance sicherzustellen.

Das Konzept des Messsystems trifft keinerlei Annahmen darüber, ob die Messung online im tatsächlichen Betrieb des Dienstes erfolgt, oder durch eine Simulation von Ereignissen angestoßen wird. Damit kann das System sowohl zur „online"- als auch zur „offline"-Messung eingesetzt werden, ohne dass eine Anpassung notwendig wäre.

Messfehler können lediglich bei der Ableitung von Primitiven aus den Aufrufen der Dienstmethoden entstehen. In der Spezifikation wird zunächst davon ausgegangen, dass die übermittelten Zeitstempel absolut korrekt sind. Eine Erweiterung des Messsystems um eine Fehlerberechnungskomponente, welche mögliche Fehler entlang der Komponenten des Systems hochrechnet, ist aber einfach möglich, indem zu jeder Korrelationsmethode angegeben wird, wie sich Messfehler der verarbeiteten Ereignisse fortpflanzen. Analog muss bei der Festlegung von Nachverarbeitungsmethoden verfahren werden.

Zusammengefasst bietet das vorliegende Spezifikationskonzept mit seinen Automatisierungsmethoden und einer Festlegung der Entwicklersicht durch eine formale Sprache optimale Unterstützung für den Dienstlebenszyklus. Zugleich sorgt der gewählte integrierte Ansatz dafür, dass von der Spezifikation eines Dienstgütemerkmals über den Aufbau eines geeigneten Messsystems und schließlich der tatsächlichen Messung dieses Dienstgütemerkmals durchgängig in ein und dem selben Modell gearbeitet werden kann. Damit werden Inkonsistenzen von der Spezifikation und ihrer Umsetzung schon per Konstruktion vermieden.

6. Zusammenfassung, Bewertung und Ausblick

In diesem Kapitel werden als Abschluss der gesamten Arbeit nochmals die wichtigsten Teilergebnisse zusammengefasst, eine abschließende Bewertung der entwickelten Konzepte und ihrer Anwendungsmöglichkeiten dargestellt sowie ausblickend weitere Fragestellungen, Forschungsmöglichkeiten und Implementierungsansätze im Kontext dieser Arbeit beschrieben.

6.1. Zusammenfassung

Diese Arbeit untersuchte Möglichkeiten, die Spezifikation von Dienstgütemerkmalen in der Verhandlungsphase des Dienstlebenszyklus zu formalisieren und gleichzeitig die Umsetzung dieser Spezifikation in den folgenden Phasen des Dienstlebenszyklus zu automatisieren und konsistent zu halten. Die untersuchte Fragestellung ergab sich direkt aus einem Szenario, das typische Aspekte der Dienstorientierung aufgreift. Zugleich war diese Fragestellung aber auch mit bestehenden Formalismen im Bereich der Dienstmodellierung, wie dem MNM-Dienstmodell, zu beschreiben. Beides zeigt, dass ein vordringliches Problem des Dienstmanagements untersucht wurde.

Die vorgestellte Lösung etabliert einen integralen Ansatz, der per Konstruktion den gesamten Dienstlebenszyklus überspannt. Er erreicht einen hohen Grad an Automatisierung und kann damit auch in dynamischen Umgebungen angewandt werden. Die Anpassung und Umsetzung einer Spezifikation in der jeweiligen Phase des Lebenszyklus wird durch einen Generierungsprozess vorgenommen und erfolgt damit ohne Zutun eines Entwicklers. Generische Verfahren zur Umsetzung einer Spezifikation in der jeweiligen Phase des Lebenszyklus wurden im Verlauf dieser Arbeit entwickelt und in den Generierungsprozess integriert.

Da diese Verfahren aus formalen Modellen abgeleitet und selbst auf Basis dieser Modelle dargestellt sind, bleibt sämtliches in den Generierungsprozess einfließende „Methodenwissen" explizit nachvollziehbar. Neue Erkenntnisse in der Modellierung können damit einfach, auch nachträglich, in den Generierungsprozess integriert werden.

Bisherige Arbeiten, die sich mit der Spezifikation von Dienstgütemerkmalen beschäftigen, haben die Orientierung am Dienstlebenszyklus bestenfalls am Rande untersucht. Ein integraler Ansatz wurde nicht vorgestellt. Ebenso wurde bisher keine explizite Modellierung der Umsetzung einer Spezifikation in den jeweiligen Phasen des Lebenszyklus vorgenommen, so dass eine Abbildung einer Spezifikation von einer Lebenszyklusphase in die folgende zwar möglich, aber nicht nachvollziehbar war.

Mit dem vorgestellten Spezifikationskonzept und seinen nachgelagerten Automatisierungschritten steht jetzt ein integraler Ansatz zur Verfügung, der das Dienstgütemanagement von der Spezifikation von Dienstgütemerkmalen in der Verhandlungsphase bis hin zur Messung der Dienstgüte, also der resultierenden Implementierung, unterstützt.

Die vorgestellten Anwendungsmöglichkeiten des Gesamtkonzepts zeigen seine vielfältige Einsetzbarkeit in unterschiedlichsten Bereichen der Diensterbringung und des Dienstmanagements. Das ursprünglich zur Lösung der Spezifikationsproblematik entwickelte Konzept eignet sich, ohne dass konzeptionelle Anpassun-

gen notwendig wären, auch zur Kategorisierung von Dienstgütemerkmalen, als Basis zum Aufbau einer detaillierten Reportingschnittstelle in Diensthierarchien sowie zur Unterstützung des Tradings bei der dynamisch verhandelten Diensterbringung.

Durch den Aufbau auf ein generisches Dienstmodell sind für die Anwendung des Konzepts keinerlei Einschränkungen auf einen bestimmten Diensttyp notwendig. Die Modellierung, die ursprünglich auf den rein funktionalen Aspekten eines Dienstes aufbaute, ließ sich, dank ihrer Generik, auch auf Dienstgütemerkmale mit Managementaspekt übertragen. Damit können sowohl Diestgütemerkmale, die sich an der Nutzungsfunktionalität des Dienstes orientieren, als auch solche, welche die Güte von Managementfunktionalitäten beschreiben, mit dem eingeführten Konzept modelliert und implementiert werden.

6.2. Bewertung

Die Bewertung des vorliegenden Spezifikationskonzepts, wie sie im Folgenden durchgeführt wird, behandelt das Konzept, seine Automatisierungsprozesse und seine Anwendungsmöglichkeiten als Ganzes. Die Bewertung des Lösungsansatzes als solchem wurde bereits zum Abschluß von Kapitel 4 an Hand des Kriterienkataloges, wie er in Abschnitt 2.4 beschrieben wurde, vorgenommen. Die hier durchgeführte Bewertung erfolgt informell, stellt die Anwendbarkeit und die Praktikabilität des Lösungsansatzes in den Vordergrund und zeigt damit dessen Tauglichkeit als Konzept zur Lösung eines Teilproblems des Dienstmanagements.

Das eingeführte Spezifikationskonzept mit den nachgelagerten Automatisierungsverfahren erreicht einen hohen Grad an Generik und ist ohne Einschränkung des Dienstbegriffs für alle Dienste anwendbar, die sich durch das MNM-Dienstmodell beschreiben lassen. Zugleich bietet es einen geschlossenen Weg von der Spezifikation eines Dienstgütemerkmals bis hin zu dessen Implementierung, also des Aufbaus eines entsprechenden Messsystems.

Hervorzuheben ist, dass dieser Weg mit minimalem Aufwand für den Entwickler durchlaufen werden kann, da er praktisch vollständig automatisiert ist. Diese Automatisierung gelingt, weil von vorneherein bereits bei der Modellierung generische von dienstspezifischen Aspekten getrennt werden. Diese Trennung sowie die strikte Abkopplung der Definition eines Dienstgütemerkmals von der aktuell gemessenen Dienstgüte wird von den beschriebenen Ansätzen, wie sie in Kapitel 3 vorgestellt wurden, bisher nicht durchgeführt. Das hier gezeigte Konzept erschließt sich durch die Umsetzung dieser Trennung ein weites Einsatzfeld, da praktisch keine Rücksichten auf Dienstspezifika genommen werden müssen.

Die Entwicklung des Spezifikationskonzepts startete mit einer deutlichen Orientierung auf Dienstgütemerkmale der Anwendungsebene mit Funktionsaspekt. Dank der strukturierten Modellierung, die eine maximale Kapselung gegenüber dem MNM-Dienstmodell im Sinn der Objektorientierung erreicht, ließ sich das Spezifikationskonzept auch auf Dienstgütemerkmale mit Managementaspekt ausdehnen. Die konzeptuell ausgerichtete Modellierung des Messprozesses, die als Basis des Spezifikationkonzepts dient, bietet somit, dank ihrer generischen Ausrichtung, optimale Wiederverwendungsmöglichkeiten.

Das zusätzlich zum Spezifikationskonzept angegebene Implementierungsdesign erleichtert die Anpassung an eine spezielle Implementierungsumgebung erheblich. Zugleich hilft es, eine Implementierung zügig umzusetzen, und erleichtert damit die Verbreitung des Spezifikationskonzepts und seiner Automatisierungsmethoden auch in bestehenden Implementierungsumgebungen. Da das vorge-

stellte Spezifikationskonzept unabhängig von der konkreten Implementierung einer Spezifikationssprache ist, kann diese einfach an das jeweilige Umfeld angepasst werden.

Die Stärke des vorgestellten Konzepts liegt darin, dass es sowohl die detaillierte und strukturierte, aber implementierungsunabhängige Modellierung von Dienstgütemerkmalen unterstützt als auch größtenteils automatisierte Möglichkeiten bietet, aus den vorgenommenen Definitionen eine Implementierung zu erstellen.

Die implementierungsunabhängige Modellierung eines Dienstgütemerkmals auf Basis des MNM-Dienstmodells erleichtert die explizite Darstellung der häufig nur implizit bekannten semantischen Zusammenhänge erheblich. Durch die Ausrichtung der Modellierung an der Nutzungsphase des Dienstes und damit an der Messung eines Merkmals, ist eine auf Basis dieser Modellierung vorgenommene Spezifikation automatisch in der Nutzungsphase ohne weitere Abbildungsschritte verwendbar.

Dieses Vorgehen stellt auch explizit dar, welche Ergebnisse die Instrumentierung der Dienstschnittstelle liefern muss, damit die Messung des spezifizierten Dienstgütemerkmals möglich wird. Zudem lässt sich, dank der detaillierten Modellierung, kanonisch ein Klassifizierungsschema aufbauen, das an den einzelnen Komponenten des Modells ausgerichtet ist. Existierende Definitionen von Dienstgütemerkmalen können so mit dem vorgestellten Spezifikationskonzept beschrieben werden, um ihre Semantik explizit zu machen und gleichzeitig eine Klassifizierung zu ermöglichen.

Das vorgestellte Konzept wurde als Spezifikationskonzept entwickelt und abstrahiert in weiten Teilen von technischen Gegebenheiten. Entsprechend lag die Betrachtung von Messfehlern und -ungenauigkeiten oder das Problem der Uhrensynchronisation bei einer verteilt stattfindenden Messung nicht im Fokus der Entwicklung, muss aber spätestens bei der konkreten Umsetzung einer Dienstgütemerkmalsspezifikation bedacht werden.

Gelingt es, genau eine Spezifikationssprache bei einer Vielzahl von Providern, Customern und Dienstentwicklern zu etablieren, lassen sich die beschriebenen Anwendungsfälle von einer dezidierten Customer-Provider-Beziehung auf eine Art „Dienst-Markt" ausdehnen. Die verwendete Sprache muss allerdings von allen Beteiligten als optimal angesehen werden. Die Entwicklung von Mechanismen, die einerseits eine Bewertung dieser Optimalität und andererseits deren Sicherstellung ermöglichen, ist notwendig, liegt aber nicht im unmittelbaren Fokus dieser Arbeit und stellt eine weiterführende Forschungsfragestellung dar, wie sie im nächsten Abschnitt beschrieben werden.

6.3. Ausblick

Aus der Zusammenfassung und Bewertung des vorgestellten Lösungsansatzes ergeben sich weitere Forschungsfragestellungen, die im Folgenden kurz vorgestellt werden. Die Darstellung gliedert sich nach den verschiedenen Komponenten des Lösungskonzepts und zeigt damit Fragestellungen im gesamten Umfeld des eingeführten Ansatzes auf.

Verbesserungen des Messsystems Da die Fehlerabschätzung bei der Messung an sich ein eigenständiges Forschungsgebiet darstellt, geht das durch den Automatisierungsprozess in Form der Messsicht generierte Messsystem vereinfachend von einer fehlerfreien Messung aus.

Dennoch bietet sich die Möglichkeit, existierende Forschungsansätze in das eingeführte Spezifikations- und Automatisierungskonzept zu integrieren. Ein erster Schritt hierzu wäre die Integration der Fehlerbetrachtung in das existierende Modell der Messsicht. Weiterhin muss das Design

der Datenübertragungsklassen hinsichtlich der Darstellung von Informationen über Messfehler und Abhänigkeiten angepasst werden. Ebenso muss geprüft werden, inwiefern die Verarbeitungsklassen mit zusätzlichen Funktionalitäten ausgestattet werden müssen, um die Propagierung von Fehlerinformationen über die unterschiedlichen Verarbeitungsschritte sicherzustellen. Abhängig von den durchgeführten Modellanpassungen muss auch die Entwicklersicht und die sie beschreibende Spezifikationssprache angepasst werden.

Das im Rahmen dieser Arbeit vorgenommene Implementierungsdesign und damit auch die prototypische Implementierung verfolgen einen synchronen Ansatz bei der Verarbeitung. Ebenso ist es denkbar, ein threadbasiertes, asynchrones Design zu wählen, in dem die einzelnen Verarbeitungsfunktionen unabhängig voneinander in eigenständigen Threads implementiert sind. Weiterhin ist es möglich, das Messsystem selbst als verteilte Anwendung zu realisieren und die einzelnen Komponenten auch räumlich zu entkoppeln. Das beschriebene objektorientierte Design muss dafür lediglich um verteilungsunterstützende Funktionen, beispielsweise aus einer klassischen Middleware wie CORBA, erweitert werden.

Erweiterung der Generierungsmöglichkeiten Die Instrumentierung einer Dienstimplementierung, sei es nachträglich oder im Moment des Dienstaufbaus, ist eine der Hauptaufgaben beim Aufbau des Messsystems für ein Dienstgütemerkmal. Besonders die Instrumentierung einer bestehenden Implementierung ist praktisch nicht zu automatisieren, da dazu formalisiertes Wissen über eben diese Implementierung notwendig wäre. Wird ein Dienst aber inklusive Instrumentierung implementiert und wird dabei eine standardisierte Plattform verwendet, dann lässt sich auch die Instrumentierung automatisieren. In [Hauc 01] wird beispielsweise ein Konzept zur automatischen Instrumentierung von komponentenbasierten Anwendungen vorgestellt.

Grundsätzlich bietet sich damit die Möglichkeit, bestehende Automatisierungsmethoden für die Instrumentierung einer Dienstimplementierung mit in das vorgestellte Spezifikations- und Automatisierungskonzept einzubinden. Dabei ist zu klären, inwieweit die vorhandene Modellierung angepasst werden muss, um eine genügend detaillierte Eingabe für die Automatisierung der Instrumentierung liefern zu können. Weiterhin ist zu untersuchen, ob diese zusätzlich benötigten Informationen aus der Entwicklersicht in ihrer bestehenden Form abgeleitet werden können, oder ob diese und entsprechend auch die zugehörige Spezifikationssprache erweitert werden muss.

Die Spezifikation von Dienstgütemerkmalen steht bei dem in dieser Arbeit entwickelten Konzept im Vordergrund. Das eigentliche Dienstgütemanagement ist eine Teilaufgabe des Managements, die zwingend auf eine eindeutige Spezifikation der Dienstgütemerkmale angewiesen ist, also bereits auf den hier vorgestellten Methoden und Konzepten aufbaut. Es bietet sich damit an, zu untersuchen, in welcher Weise sich ein System zum Dienstgütemanagement aus der Spezifikation von Dienstgütemerkmalen und der Festlegung der gewünschten bzw. erwarteten Dienstgüte generieren lässt.

Dazu müssen zunächst grundsätzliche Verfahren des Dienstgütemanagements ermittelt und, analog dem Vorgehen bei der Modellierung des allgemeinen Messprozesses, als Ergänzungen des bestehenden Dienstmodells modelliert werden. Ist diese Umsetzung der bestehenden Managementmethoden in das Modell erfolgt, lassen sich Generierungsmöglichkeiten und damit auch der Bedarf an Erweiterungen der Entwicklersicht und der zugehörigen Spezifikationssprache abschätzen.

Weiterentwicklung der Spezifikationssprache Zentrales Element des in dieser Arbeit eingeführten Spezifikationskonzepts ist eine formale Sprache zur Beschreibung der Entwicklersicht, aus der alle weiteren Darstellungen, die sog. Sichten, generiert werden. Da in dieser Arbeit ein detailliertes Design dieser Spezifikationssprache beschrieben ist, wurde eine mögliche Implementierung nur prototypisch durchgeführt, um die praktische Tauglichkeit des Konzeptes zu zeigen. Die tatsächliche Ausprägung der Spezifikationssprache im Sinne einer festgeschriebenen Syntax wird durch das vorgestellte Design nicht festgelegt.

Viele der vorgestellten Anwendungsmöglichkeiten, z.B. der Vergleich von Spezifikationen oder das dienstgüteorientierte Trading, erreichen allerdings eine größere Einsatzbreite, wenn unterschiedliche Organisationen eine syntaktisch gleiche Spezifikationssprache verwenden. Es bleibt also zu untersuchen, welche Syntax für die Spezifikationssprache eine Idealform darstellt bzw. ob diese Idealform unabhängig vom Anwendungsgebiet überhaupt erreichbar ist. Sollte dies nicht der Fall sein, können Abbildungsvorschriften zwischen den nur syntaktisch unterschiedlichen Sprachen geschaffen werden, so dass die weite Anwendbarkeit im Ergebnis erhalten bleibt.

Modellerweiterungen Einige der beschriebenen Anwendungsfälle gehen davon aus, dass bestehende Spezifikationen beliebig zwischen Organisationseinheiten ausgetauscht werden können. Um diese Funktionalität zu realisieren bietet es sich an, die vorhandenen Definitionen des Customer-Service-Managements, wie es in [Lang 01, Nerb 01] beschrieben ist, um Mechanismen zum Austausch von Spezifikationen zu erweitern.

Das vorgestellte Spezifikations- und Automatisierungskonzept ist für Dienstgütemerkmale mit Funktions- oder Managementaspekt einsetzbar. Mittlerweile werden zunehmend auch Dienstgütemerkmale mit Inhaltsaspekt festgelegt, die beispielsweise die Farbtreue eines Videoübertragungssystems oder die Sprachqualität bei der Internettelefonie beschreiben.

Es gilt zu untersuchen, inwieweit Dienstgütemerkmale dieser Art mit dem bestehenden Ansatz zu beschreiben sind bzw. dessen Defizite aufzudecken und durch geeignete Modellerweiterungen zu beheben. Kernstück bei diesen Modellerweiterungen muss eine Modellierung grundsätzlicher Vorgehensweisen bei der Informationsverarbeitung sein, um darauf aufbauend entsprechende Dienstgütemerkmale, die eben die Qualität dieser Informationsverarbeitung beschreiben, spezifizieren zu können.

In diesem Zuge können auch Möglichkeiten untersucht werden, den Spezifikationsansatz auf Dienstgütemerkmale der Nutzer-Ebene auszudehnen. Dazu müssen Mechanismen gefunden werden, um die in der Nutzer-Ebene üblichen, teilweise sehr abstrakten und zudem inhärent ungenauen Festlegungen zu formalisieren oder beispielsweise auf einen Katalog gängiger und bereits formalisierter Definitionen abzubilden.

Nach der erfolgreichen Umsetzung dieser Modellerweiterungen stünde ein integrales Konzept zur Verfügung, das den gesamten Dienstgütequader abdeckt und damit auch die Möglichkeit bietet, die Abbildungsproblematiken zwischen den einzelnen Schichten des Quaders zu untersuchen und dafür ähnlich generische Lösungen aufzubauen, wie sie mit dem vorliegenden Konzept für die Problematik der Dienstgütespezifikation und ihrer Umsetzung im Dienstlebenszyklus gefunden wurden.

A. Prototypische Implementierung

Dieser Anhang gibt einen gerafften Überblick über die prototypische Implementierung des in Kapitel 4 eingeführten Spezifikationskonzepts und seiner Automatisierungsverfahren. Zu Beginn wird eine Umsetzung des Modells des erweiterten Messprozesses mit Java an Hand der jeweils implementierten Klassen beschrieben. Daran anschließend wird als Referenz die gesamte Syntax von QoSSL angegeben, bevor der Aufbau des QoSSL-Compilers beschrieben und schließlich die Funktion des Compilers an einer Beispieldefinition in QoSSL demonstriert wird.

A.1. Java-Implementierung des Modells des erweiterten Messprozesses

Die Darstellung beginnt mit der Erläuterung von Java-spezifischen Vorgehensweisen bei der Umsetzung des vorliegenden UML-Designs. Anschließend wird die Implementierung jeder Klasse im einzelnen beschrieben. Zur besseren Wiederverwendbarkeit werden in der Implementierung, wie bereits bei der Festlegung von QoSSL, alle Identifikatoren mit englischen Namen bezeichnet.

A.1.1. Java-Spezifika

Das vorgegebene Design muss im Zuge einer prototypischen Implementierung mit einer konkreten Programmiersprache umgesetzt werden. Dabei werden die im Design festgelegten Konzepte in die Programmiersprache umgesetzt. Häufig bietet die Programmiersprache nicht alle im Design verwendeten Konzepte zur direkten Verwendung an. Im Falle von Java, das in diesem Fall zur Implementierung verwendet wurde, ist z.B. das Konzept der Mehrfachvererbung nicht direkt umsetzbar, da Java nur Einfach-Vererbung zulässt. Statt dessen wird das Konzept der Interfaces (Methodendefinitionen ohne Implementierung) verwendet. Bei jeder Abbildung einer Mehrfachvererbung aus dem Design in die Programmiersprache Java muss also entschieden werden, welche Klassen aus dem Design durch Interfaces realisiert werden.

Wie alle objektorientierten Sprachen bietet auch Java die Möglichkeit, polymorphe Methoden zu implementieren. In der vorliegenden Implementierung wird dieser Mechanismus verwendet, um übersichtliche Klassendefinitionen zu erstellen.

In Java können Klassendefinitionen in sog. Packages zusammengefasst werden, um die gegenseitigen Sichtbarkeiten zu regeln und eine Art Modulkonzept zu ermöglichen. Die vorliegende Implementierung baut die beiden Packages *qossl.base* und *qossl.generated* auf. Ersteres enthält diejenigen Klassen, welche das erweiterte Modell des Messprozesses umsetzen, zusammen mit den Klassen der Automatisierungsunterstützung. Letzteres enthält Klassen, die nach den Vorgaben der Entwicklersicht generiert wurden.

In der vorliegenden, prototypischen Implementierung wird so weit wie möglich auf die Standard-Java-API und ihre Datenstrukturen zurückgegriffen. Dabei steht nicht die Effizienz, sondern die Effektivität, also die prinzipielle Darstellung eines Implementierungsweges, im Vordergrund.

A.1.2. Datenübertragungsklassen

Die Datenübertragungsklassen können direkt aus der Designvorlage implementiert werden. Sie enthalten lediglich die im Design angegebenen Attribute. Die Java-Klasse *Primitive* implementiert die Klasse *Primitive* aus dem Design. Die Klasse *Event* repräsentiert die Ereignis-Klasse, die Klasse *Reading* einen Messwert und die Klasse *Feature* ein Dienstgütemerkmal. Alle Klassen werden im Package qossl.base zusammengefasst. Zu allen vier Klassen ist der Java-Code im Folgenden angegeben.

Primitive.java Die Klasse *Primitive* dient zur Spiegelung eines Konzepts und als Basis für die Ableitung weiterer Primitiven. Sie enthält daher keinerlei Attribute oder Zugriffsmethoden und ist abstract deklariert.

```
    package qossl.base;
2
    abstract public class  Primitive {
4

6   };
```

Event.java In der Klasse Event sind die Attribute eines Ereignisses sowie entsprechende Zugriffsmethoden für diese Attribute enthalten. Die explizite Implementierung eines Konstruktors stellt sicher, dass nur Instanzen mit belegten Attributen existieren können. Zusätzlich ist (ab Zeile 31) eine Methode *toString* implementiert, welche die einfache Ausgabe aller Attributwerte erlaubt. Um den Vergleich von unterschiedlichen Instanzen der Klasse Event bei der Verarbeitung im EventKorrelator zu ermöglichen, ist zusätzlich (ab Zeile 43) eine Methode equals implementiert, welche die Übereinstimmung zweier Instanzen der Klasse Event an Hand der Attributbelegungen prüft.

```
    package qossl.base;
2
    public class  Event {
4
      long timeStamp;
6     long id;
      String name;
8
      public Event (String nameToSet, long idToSet, long timeStampToSet){
10      name = nameToSet;
        id = idToSet;
12      if (timeStampToSet == 0) {
          java.util.Date d = new java.util.Date();
14        timeStamp = d.getTime();
        }
16      else
        {
18        timeStamp = timeStampToSet;
        }
20
      };
```

```
22
    public long getTime (){
24    return timeStamp;
    };
26
    public long getId (){
28    return id;
    };
30
    public String toString () {
32    java.util.Date d = new java.util.Date(getTime());
      return "EVENT␣DATA␣[" + name + "\n" +
34            "\t␣Id\t\t␣" + getId() + "\n" +
              "\t␣TimeStamp\t␣" + d + "\n" +
36            "\t␣TimeTicks\t␣" + getTime() + "\n";
    };
38
    public String getName() {
40    return name;
    }
42
    public boolean equals(Event e) { //this one is neccesarry for the operations in
44                                   // java.util.vector
      if ((e.getId() == id) & (e.getTime() == timeStamp) & (e.getName() == name))
46    {
        return true;
48    }
      else
50    {
        return false;
52    }
    }
54
  };
```

Reading.java Die Attribute eines Messwertes werden mit ihren entsprechenden Zugriffsmethoden in der Klasse Reading zusammengefasst. Auch bei dieser Klasse wird durch die explizite Angabe eines Konstruktors sichergestellt, das nur Instanzen mit belegten Attributen erzeugt werden können.

```
  package qossl.base;
2
  public class Reading {
4
    private double value;
6   private long   processedEvents;
    private double relativeError;
8

10  public Reading(double valueToSet, long processedEventsToSet, double relativeErrorToSet){
```

```
        value          = valueToSet;
12      processedEvents = processedEventsToSet;
        relativeError  = relativeErrorToSet;
14    };

16    public double getValue() {
        return value;
18    }

20    public long getProcessedEvents() {
        return processedEvents;
22    }

24    public double getRelativeError() {
        return relativeError;
26    }

28
    };
```

Feature.java Ein Feature repräsentiert ein Dienstgütemerkmal, das im Laufe des Automatisierungsprozesses als Verfeinerung einer bestimmten Verteilung beschrieben wird. Da Java keine Mehrfachvererbung unterstützt und Feature aus Sicht des Designs eine abstracte Oberklasse ohne Attribute und Methoden ist, wird dieses als sog. tagging-Interface implementiert.

```
    package qossl.base;
2
    public interface Feature {
4   };
```

A.1.3. Datenverarbeitungsklassen

Wie bereits im Design des erweiterten Messprozesses festgelegt, realisieren die sog. Datenverarbeitungsklassen Funktionsblöcke dieses Messprozesses. Im Design wurde zudem ein Sender/Empfänger-Schema festgelegt, über das der Datenaustausch zwischen den einzelnen Verarbeitungsklassen realisiert wird. Dieses Schema wird in der Java-Implementierung durch entsprechende Interfaces realisiert. Im Folgenden werden die einzelnen Klassen zusammen mit den jeweiligen Interfaces und eventuell für die Realisierung von Zusatzklassen notwendigen Klassen entlang der Verarbeitungsreihenfolge des Messprozesses beschrieben.

Tap.java Die Klasse *Tap* realisiert die sog. SAP-Anbindung. Diese Anbindung muss implementierungsspezifisch erstellt werden und damit als Verfeinerung der Klasse *Tap* erstellt werden. Die Basisklasse *Tap* stellt eine *send*-Methode bereit, mit der Instanzen der Klasse *Primitive* an eine Instanz der Klasse *EventGenerator* übergeben werden können. Bei der Implementierung einer spezifischen Verfeinerung von *Tap* kann vom Entwickler auf diese Methode zurückgegriffen werden. Sie stellt die Übertragung der Primitive

an alle registrierten Empfänger sicher. Dabei wird allerdings die generierte, verfeinerte *Primitive* mit dem Typ der abstrakten Oberklasse *Primitive* übergeben, so dass jede Verfeinerung von *EventGenerator* vor der Verarbeitung einen expliziten Typecast durchführen muss, um auf die Attribute der Primitive zugreifen zu können. Die Klasse *Tap*, resp. eine ihrer Verfeinerungen, versendet Instanzen der Klasse *Primitive* und implementieren deshalb das Interface *PrimitiveSender*.

```
    package qossl.base;
 2
    public abstract class  Tap implements PrimitiveSender {
 4
      PrimitiveReceiver receiver = null;
 6
      public final  void send (Primitive primitiveToSend) {
 8

10      // System.out.println ("Tap [" + toString() + " sendet Primitive [" +
        //                              primitiveToSend.toString() + " an [" + receiver.toString () + "]");
12      // System.out.println (primitiveToSend.getClass().getName());

14      receiver.receive(primitiveToSend);
       }
16

18     public void bind (PrimitiveReceiver whoItGoesTo) {

20       receiver = whoItGoesTo;

22     }
    };
```

PrimitiveSender.java Das Interface *PrimitiveSender* muss von jeder Klasse, die Instanzen der Klasse *Primitive*, beispielsweise an einen *EventGenerator,* versendet, implementiert werden. Neben der Methode *send* muss auch eine *bind*-Methode implementiert werden, die den Sender an den entsprechenden Empfänger bindet.

```
    package qossl.base;
 2
    public interface PrimitiveSender {
 4
      void bind (PrimitiveReceiver whoItGoesTo);
 6
      void send (Primitive toSend);
 8
    };
```

EventGenerator.java Die Aufgabe, Primitiven in Ereignisse umzusetzen, wird von der Klasse *EventGenerator* übernommen. Damit ist die Klasse Empfänger von Primitiven einerseits und Sender von Ereignissen andererseits. Entsprechend implementiert sie die

beiden Interfaces *PrimitiveReceiver* und *EventSender*. Die Klasse dient als Basis zur Implementierung eines spezifischen Ereignisgenerators. Dazu muss die *receive*-Methode überladen und mit der für die Generierung eines Ereignisses notwendigen Funktionalität versehen werden. (Diese Aufgabe wird vom Generator übernommen.)

Das generierte Ereignis, eine Instanz der Klasse *Event,* muss an eine Instanz der Klasse *EventCorrelator* durch Aufruf der entsprechenden *receive*-Methode übergeben werden. Wie unten beschrieben, kann eine Instanz einer Verfeinerung von *EventCorrelator* mehrere Implementierungen der *receive*-Methode, für unterschiedliche Typen resp. Verfeinerungen der Klasse *Event*, enthalten.

Die von der Basisklasse *EventGenerator* implementierte *send*-Methode verschattet diese Aufgabe für den Entwickler resp. Codegenerator, der einen spezifischen Ereignisgenerator als Verfeinerung der Klasse *EventGenerator* aufbaut. Die *send*-Methode verarbeitet allgemein Instanzen der Klasse *Event*. Damit kann zur Compile-Zeit nicht mehr die zum Typ des übergebenen *Events* passende *receive*-Methode des *EventCorrelators* aufgerufen werden. Deshalb implementiert die *send*-Methode der Klasse *EventGenerator* einen dynamischen Implementierung, bei dem zur Laufzeit die zur jeweiligen Verfeinerung von *Event* „passende" *receive*-Methode aufgerufen wird. Der aktuelle Typ der Verfeinerung von *Event* sowie die von der jeweiligen Instanz einer Verfeinerung von *EventCorrelator* implementierten *receive*-Methoden und ihre Signaturen werden durch Aufrufe des Java-Reflection-APIs ermittelt.

```
    package qossl.base;
 2
    public   class  EventGenerator implements PrimitiveReceiver, EventSender{
 4
      long idCounter;
 6    java.util.Vector receiverList;

 8    public EventGenerator (){
        receiverList = new java.util.Vector();
10      idCounter = 1;
      };
12
      public   void receive (Primitive myPrimitive)
14    {
        System.out.print ("␣receive-method@" + this.toString());
16      System.out.print ("␣invoked␣with␣");
        System.out.print (myPrimitive.toString());
18      System.out.println ("\n");

20       // build an event
        Event e;
22      java.util.Date d = new java.util.Date();
        e = new Event("generated_with_" + myPrimitive.toString(), idCounter, d.getTime());
24      send(e);
        idCounter ++;
26
      };
28
      public void bind (EventReceiver sendTo){
```

```
30        receiverList.addElement(sendTo);
      };
32
      public final void send (Event eventToSend){
34

36      // the matching receive-method in the current
        // instance of the EventCorrelator has to be determined
38
        EventReceiver whoItGoesTo = null;
40
        whoItGoesTo = (EventReceiver)receiverList.firstElement();
42

44      // matching method is searched using the reflection api

46      try {
          Class argTypes[] = new Class[1];
48        argTypes[0]      = eventToSend.getClass();

50        // Object to iterate

52        Object atTheMoment = whoItGoesTo;

54        // search suitable methode
          java.lang.reflect.Method dynamicMethod;
56        while (true) {
            try    {
58                    dynamicMethod =
                        atTheMoment.getClass().getDeclaredMethod( "receive", argTypes);
60                    // leave loop if there is a matching method
                      break;
62                  }
            catch (java.lang.NoSuchMethodException ex) {
64                    if (atTheMoment.getClass().getName() == "java.lang.Class")
                      {
66                      // paramter object has been up-casted as often as possible
                        throw ex;
68                    }
                      {
70                      // dynamically up-cast and try again
                        // because we are still in the loop
72                      atTheMoment = atTheMoment.getClass().getSuperclass();

74                    }
                    }
76          }
          // build dyn. method invocation
78        Object argList[] = new Object[1];
          argList[0]       = eventToSend;
80
```

```
            dynamicMethod.invoke( whoItGoesTo, argList);
82
          }
84        catch ( java.lang.NoSuchMethodException ex) {
            System.out.println("***␣method␣not␣found");
86        }
          catch ( java.lang.reflect.InvocationTargetException ex1 ) {
88          System.out.println("***␣error␣in␣method␣invocation␣" + ex1 );
          }
90        catch ( java.lang.IllegalAccessException ex2) {
            System.out.println("***␣Illigal␣access");
92        }

94
        };
96
      }
```

PrimitiveReceiver.java Dieses Interface muss von Klassen implementiert werden, die Instanzen von *Primitive* empfangen können sollen. Dazu muss eine geeignete *receive*-Methode implementiert werden.

```
    package qossl.base;
2

4   interface PrimitiveReceiver {

6     void receive (Primitive myPrimitive);

8   };
```

EventSender.java Zum Versenden von Ereignissen (*Events*) muss eine Klasse das Interface *EventSender* implementieren. Zu diesem Zweck muss eine *send*-Methode und eine *bind*-Methode zum Binden an den/die jeweiligen Empfänger erstellt werden.

```
    package qossl.base;
2
    interface EventSender{
4
      void bind (EventReceiver whoItGoesTo);
6
      void send (Event toSend);
8
    };
```

EventCorrelator.java Durch die Klasse *EventCorrelator* wird die eigentliche Korrelationsfunktionalität des Messsystems implementiert. Alle Methoden außer *bind* und *send* müssen von einer Verfeinerung, einem für ein Dienstgütemerkmal spezifischen *EventCorrelator* implementiert werden und sind deshalb *abstract* deklariert. Für jeden vom

EventCorrelator verarbeiteten Typ einer Verfeinerung von *Event* muss eine entsprechende *receive*-Methode implementiert werden. Die *receive*-Methode der Oberklasse *EventCorrelator* ist in einem einfachen Rumpf implementiert und würde nur dann aufgerufen, wenn für die vom *EventGenerator* übergebene Instanz von *Event*, resp. ihrem Typ, keine passende *receive*-Methode existiert.

Bei einer Verfeinerung der Klasse *EventCorrelator* muss die Methode *correlate* implementiert werden, die einen Messwert durch die Korrelation von Ereignissen berechnet. Als Hilfestellung wird die Klasse *Queue* (siehe Abschnitt A.1.4) angeboten, die eine Pufferung von Ereignissen erlaubt und Korrelationsmethoden bereitstellt.

Ein Eventkorrelator empfängt Ereignisse und sendet Messwerte an die statistische Nachverarbeitung. Entsprechend werden von der Klasse *EventGenerator* die beiden Interfaces *EventReceiver* und *ReadingSender* implementiert.

```
    package qossl.base;
 2
    public abstract class  EventCorrelator implements EventReceiver, ReadingSender{
 4
      public double          internValue;
 6    ReadingReceiver readingGoesTo;

 8    public EventCorrelator (){
        internValue = 0;
10    };

12    public void receive (Event myEvent)
      // this  method is only invoked if no specific one
14    // is  implemented in a derivation
      {
16      System.out.print (myEvent);
        System.out.print (" was received @ ");
18      System.out.print (this);
        System.out.println (" the top level class EventCorrelator");
20    };

22    public void bind(ReadingReceiver myReceiver) {
        readingGoesTo = myReceiver;
24
      }
26
      public final  void send(Reading theReading) {
28      readingGoesTo.receive(theReading); }

30    public abstract void correlate();
        // has to be implemented in a derivation of this class
32  }
```

EventReceiver.java Das Interface *EventReceiver* legt die Funktionalität zum Empfangen von Instanzen der Klasse *Event* in ihrer Signatur, also die Signatur der *receive*-Methode, fest.

```
    package qossl.base;
2

4   interface EventReceiver {

6     void receive (Event myEvent);

8   };
```

ReadingSender.java Analog zum bisherigen Vorgehen bei der Festlegung von Interfaces, die das Konzept eines Senders umsetzen sollen, fordert das Interface *ReadingSender* die Implementierung einer *bind*- und einer *send*-Methode.

```
    package qossl.base;
2
    interface ReadingSender {
4
      void bind (ReadingReceiver whoItGoesTo);
6
      void send (Reading toSend);
8
    };
```

PostProcessor.java Funktionen zur statistischen Nachverarbeitung werden durch die Klasse *PostProcessor* implementiert. Die eigentliche Aggregationsfunktionalität wird später durch den Aufbau einer Verfeinerung von *Feature* auf Basis einer Klasse aus dem Aggregationsrepository realisiert, so dass die Klasse *PostProcessor* lediglich einen konzeptionellen Rumpf darstellt, der, analog zum bisherigen Vorgehen, die Interfaces *ReadingReceiver* und *FeatureSender* implementiert.

```
    package qossl.base;
2
    public interface PostProcessor extends ReadingReceiver, FeatureSender {
4
      void receive (Reading theReading) ;
6   }
```

ReadingReceiver.java Durch dieses Interface wird angezeigt, dass eine Klasse Instanzen der Klasse *Reading* empfangen kann und dazu eine *receive*-Methode bereitstellt.

```
    package qossl.base;
2

4   public interface ReadingReceiver {

6     void receive (Reading myReading);

8   };
```

FeatureSender.java Das Interface *FeatureSender* zeigt an, dass eine Klasse Dienstgütemerkmale in Form von Instanzen der Klasse *Feature* versendet und dazu die Methoden *send* und *bind* bereitstellt.

```
    package qossl.base;
2
    public interface FeatureSender {
4
      void bind (FeatureReceiver whoItGoesTo);
6
      void send (Feature featureToSend);
8
    };
```

FeatureReceiver.java Das Interface *FeatureReceiver* muss von einer Klasse implementiert werden, die nach dem MNM-Dienstmodell *mangement functionality* oder Teile davon umsetzt, also Dienstgüte merkmale mit ihrer Wertebelegung empfangen und verarbeiten kann.

```
    package qossl.base;
2
    public interface FeatureReceiver {
4
      void receive(Feature myFeature);
6   };
```

A.1.4. Ergänzungen für Automatisierung

Zusätzlich zu den Komponenten, welche das erweiterte Modell des Messprozesses umsetzen, wurden Klassen implementiert, welche es durch die Bereitstellung von Automatisierungsfunktionen erlauben, den Aufbau der Definitionssicht übersichtlicher und einfacher zu gestalten, so dass diese einerseits besser lesbar bleibt und andererseits vom Generator einfacher erzeugt werden kann.

A.1.4.1. Korrelationsrepository

Bereits im Design ist ein Korrelationsrepository vorgesehen, um die verwendbaren Korrelationsfunktionen zu standardisieren und einfach referenzierbar zu machen. In der vorliegenden prototypischen Implementierung wird diese Aufgabe von der Klasse *Queue* übernommen. Ein weiterer Ausbau des Korrelationsrepositories, ähnlich wie beim weiter unten beschriebenen Aggregationsrepository, ist natürlich möglich.

Queue.java Die Klasse *Queue* wurde implementiert, um den Aufbau der Korrelationsfunktionalität einfach und übersichtlich halten zu können. Sie implementiert eine Warteschlange, in die Instanzen der Klasse *Event* nach ihrem Zeitstempel eingeordnet werden können. Dazu werden entsprechende Zugriffsmethoden sowie Methoden zum Suchen von Paaren identischer Events in unterschiedlichen Warteschlangen implementiert. Damit stellt die Klasse *Queue* eine einfache Implementierung des im Design geforderten Korrelationsrepositories dar.

```
package qossl.base;

public class Queue {

  // establishes a priority queue - event id is used as priority
  // first element has lowerst priority

  java.util.Vector queueData = new java.util.Vector();

  class EventPair {

    Event one;
    Event two;

    EventPair(Event oneEvent, Event anOtherEvent) {
      one = oneEvent;
      two = anOtherEvent;
    }

    Event getOne() { return one;}

    Event getTwo() { return two;}
  };

  public void add (Event anEvent) {

    long id = anEvent.getId();
    int index = 0;
    int size = queueData.size();
    Event currentEvent;

    if (!( size >0)) {  // an empty queue
      queueData.addElement(anEvent);
    }
    else
    {
      // queue is not empty
      index =0; // start at top of the queue
      currentEvent = (Event)queueData.elementAt(index);


      while ((currentEvent.getId() <= id) & (index < size )) {
        currentEvent = (Event)queueData.elementAt(index);
        index ++;

      }
      if (( index == size ) | ( index ==0)) {  // event has to be appended
        queueData.insertElementAt(anEvent,index);
      }
      else
```

```
            {
52              // element has to be replaced
                queueData.insertElementAt(anEvent,index-1);
54          }
          }
56      }
        public void removeElement(Event e) {
58
          queueData.removeElement(e);
60      }

62      public String toString () {

64        String toReturn ="";
          int index;
66        int size;
          Event currentEvent;
68
          index = 0;
70        size = queueData.size();
          currentEvent = (Event)queueData.elementAt(index);
72

74        while ((index < size )) {
            currentEvent = (Event)queueData.elementAt(index);
76          if (currentEvent == null) {
              toReturn = toReturn + "Event␣ID␣<NULL>␣@␣" + index +"\n";
78            currentEvent = new Event ("<NULL>",0,0);
            }
80          else
            {
82            toReturn = toReturn + "Event␣ID␣<" + currentEvent.getId() + ">␣@␣" + index + "\n";
            }
84          index ++;

86        }

88        return toReturn;
        }
90
        public int getSize () {
92        return queueData.size();
        }
94
        public Event getElementAt(int at) {
96        return (Event)queueData.elementAt(at);
        }
98
        public Event hisMatch(Queue theOtherQueue) {
100       EventPair toReturn;
```

```
102       toReturn = match(theOtherQueue);

104       if (toReturn == null) {
            return null;
106       }
          else
108       {
            return toReturn.getTwo();
110       }
        };
112
        public Event myMatch(Queue theOtherQueue) {
114       EventPair toReturn;

116       toReturn = match(theOtherQueue);

118       if (toReturn == null) {
            return null;
120       }
          else
122       {
            return toReturn.getOne();
124       }
        };
126

128     EventPair match (Queue theOtherQueue) {

130       Event myEvent;
          Event hisEvent;
132       int hisIndex = 0;
          int myIndex = 0;
134       int hisSize = 0;
          int mySize = 0;
136
          hisSize = theOtherQueue.getSize();
138       mySize = this.getSize();

140       if ((hisSize == 0) | (mySize == 0))
          {
142        return null;
          }
144

146       //outer loop
          while (myIndex < mySize) {
148
            myEvent = this.getElementAt(myIndex);
150         hisIndex = 0;

152
```

```
            // inner loop
154         while (hisIndex < hisSize ) {
              hisEvent = theOtherQueue.getElementAt(hisIndex);
156
              if  (myEvent.getId() == hisEvent.getId()) {
158             // found a match
                return new EventPair(myEvent, hisEvent);
160           }

162           hisIndex ++;
            }
164         myIndex ++;
          }
166       // no match found
          return null;
168     }

170
      };
```

A.1.4.2. Aggregationsrepository

Im Aggregationsrepository werden Funktionalitäten zur statistischen Zusammenfassung von Messergebnissen nach den Vorgaben des Designs mit einer standardisierten Schnittstelle zusammen gefasst. Ein Dienstgütemerkmal, repräsentiert durch eine Verfeinerung der Klasse *Feature,* kann von einer Klasse aus dem Korrelationsrepository erben und damit einfach die entsprechende Aggregationsfunktionalität benutzen.

AggregationRepository.java Die Klasse *AggregationRepository* ist Oberklasse für alle im Aggregationsrepository enthaltenen Klassen und stellt zugleich ein standardisiertes Interface für alle angebotenen Funktionen bereit. Dieses stellt die beiden Methoden *addValue* zum Hinzufügen eines Wertes sowie *toString* zum Ausgeben aller Daten in der Verteilung, welche durch die gewählte Aggregationsfunktionalität erstellt wird. In einer weiteren Ausbaustufe müssten auch standardisierte Methoden zum Zugriff auf die Werte der erstellten Verteilung implementiert werden, um deren Verarbeitung durch das Dienstmanagement ermöglichen zu können.

```
    package qossl.base;
 2
    public class   AggregationRepository {
 4
      public void addValue (double theValue) {
 6    };

 8    public String toString () {
        return new String("AggregationRepository:␣not␣yet␣implemented");
10    };

12  };
```

Sum.java In der vorliegenden Implementierung wird als einfachste Aggregationsfunktion die Klasse *Sum* implementiert, die übergebene Werte addiert. Weitere Aggregationsfunktionen, wie sie bereits in der Definition von QoSSL vorgesehen sind, können jederzeit ergänzt werden.

```
    package qossl.base;
 2
    public class   Sum extends AggregationRepository {
 4
    double theSum = 0;
 6
      public void addValue (double theValue) {
 8      theSum = theSum + theValue;
      };
10
      public String toString(){
12      return new String("Sum: " + theSum);
      };
14

16  };
```

A.1.4.3. Instanzierungshilfestellung

Dem Design folgend baut die Instanzierungshilfestellung das Messsystem nach den Vorgaben der Entwicklersicht, die in dieser Implementierung durch die QoSSL beschrieben wird, auf und sorgt für die korrekte Bindung der einzelnen Komponenten aneinander.

Builder.java Die Klasse *Builder* stellt den grundsätzlichen Rahmen für die Implementierung einer spezifischen Instanzierungshilfestellung. Die jeweils notwendige Funktionalität kann allerdings erst durch eine vom Generator erzeugte Verfeinerung der Klasse *Builder* erbracht werden. Als abstrakte Oberklasse deklariert *Builder* lediglich die Methode *buildInstances* als abstrakte Methode ohne Rumpf, die von einer Verfeinerung implementiert werden muss.

```
    package qossl.base;
 2
    public abstract class  Builder {
 4
      java.util.Vector tapList = new java.util.Vector();
 6

 8    public abstract void buildInstances();

10

12  };
```

A.2. QoSSL-Syntax

Bevor im nächsten Abschnitt der Aufbau des QoSSL-Compilers mit dem Compiler-Generator javaCC beschrieben wird, ist an dieser Stelle nochmals die komplette Syntaxdefinition von QoSSL wiedergegeben. Der systematische Aufbau der Syntaxdefinitionen und ihr Anwendung wurden bereits in Abschnitt 4.3.3.4 ausführlich beschrieben, so dass die Darstellung hier als Referenz zu verstehen ist.

```
    // Macros
 2
    <LBRACE>            ::= "{"
 4
    <RBRACE>            ::= "}"
 6
    <END_STATEMENT> ::= ";"
 8
    <NULL>              ::= "NULL"
10
    <UNIT>              ::= "bit"|"bit/sec"|"sec"|"%"|"NULL"
12
    <FEATURE_KEYWORD> ::= "FEATURE"
14
    <READING_KEYWORD> ::= "READING"
16
    <EVENT_KEYWORD> ::= "EVENT"
18
    <PRIMITIVE_KEYWORD> ::= "PRIMITIVE"
20
    <USES_KEYWORD> ::= "USES"
22
    <COMPUTES_KEYWORD> ::= "COMPUTES"
24
    <UNIT_KEYWORD> ::= "UNIT"
26
    <AS_KEYWORD>    ::= "AS"
28
    <EVERY_KEYWORD> ::= "EVERY"
30
    <JOIN_KEYWORD> ::= "JOIN"
32
    <TO_KEYWORD>    ::= "TO"
34
    <REQUIRES_KEYWORD> ::= "REQUIRES"
36
    <ON_TAP_KEYWORD> ::= "ON␣TAP"
38
    <ISOLATE_KEYWORD> ::= "ISOLATE"
40
    <ID_KEYWORD>    ::= "ID"
42
    <AUTO_KEYWORD> ::= "AUTO"
```

```
44
    <HASH_KEYWORD> ::= "HASH"
46
    <COUNT_DEF>      ::= "count"
48
    <MYMATCH_DEF>  ::= "myMatch"
50
    <HISMATCH_DEF>  ::= "hisMatch"
52
    <TIME_DEF>       ::= "time"
54
    <ID_DEF>           ::= "id">
56
    <AGGREGATION>  ::= "ACF"     // absolute cumulative frequency
58                       | "RCF"     // relative  cumulative frequency
                         | "CF"      // cumulative frequency
60                       | "AF"      // absolute frequency
                         | "BOXPLOT" // create a boxplot
62                       | "MAVG"  // moving avarge
                         | "SUM" > // simply sum up
64                                   // more aggregation-functions
                                     // to be added here
66

68  <ALPHA_CONSTANT> ::= ("'"(["a"-"z","A"-"Z","0"-"9","_","-"])*"'")

70
    <ARRAY_DEFINITION> ::= "["["1"-"9"](["0"-"9"])*"]"
72                       | "[]"

74

76  <NUM_CONSTANT> ::= ("-")? ("0")? <UNSIGNEDINTEGER> ("."["0"-"9"](["0"-"9"])*)?

78  <UNSIGNEDINTEGER> ::= (["1"-"9"])(["0"-"9"])*

80  <STAR>             ::= "*"

82  <PLUS>             ::= "+"

84  <QUESTIONMARK> ::= "?"

86  <OPERAND>        ::= <STAR> | <PLUS> | "-" | "**"

88  <CORRELATION>   ::= "left_equal_id"
                        | "right_equal_id"
90                      | "count"
                        | "time_dif"
92
    <COMPARE>        ::= "<"|"<="|"=="|"!="|">="|">"
94
```

```
<CONSTANT_ARRAY> ::= "["(<NUM_CONSTANT>|<ALPHA_CONSTANT>)
                        (","(<NUM_CONSTANT>|<ALPHA_CONSTANT>))*
                     "]"

<BIT>            ::= "BIT"

<BYTE>           ::= "BYTE"

<CHAR>           ::= "CHAR"

<EVAL_FUNCTION> ::= "size"
                     // more functions to be added here

<IDENTIFIER>     ::= [ "a"-"z","A"-"Z"](["a"-"z","A"-"Z","_","0"-"9"])*

<COMMENT>        ::= "//" (~["\n"])*"\n"

<DOT_SEPARATOR> ::= "."


// LANGUAGE STATEMENTS


<QoSSL>          ::= (
                        <COMMENT>
                      | <Feature>
                      | <Reading>
                      | <Event>
                      | <Primitive>
                     )* <EOF>


<Feature>        ::= <FEATURE_KEYWORD> <IDENTIFIER>
                     <JOIN_KEYWORD> <NUM_CONSTANT> <IDENTIFIER>
                     <TO_KEYWORD> <AGGREGATION>
                     <END_STATEMENT>

<Reading>        ::= <READING_KEYWORD> <IDENTIFIER>
                     <REQUIRES_KEYWORD><IDENTIFIER> <AS_KEYWORD> <IDENTIFIER>
                     // allow multiple required events
                     (
                        "," <IDENTIFIER> <AS_KEYWORD> <IDENTIFIER>
                     )*
```

```
146                     <COMPUTES_KEYWORD> <Term>
                        <UNIT_KEYWORD> <UNIT>
148                     <END_STATEMENT>

150             <Term>          ::= <Unexpandable><Expandable>
                                  | "(" <Term> ")"
152
                <Expandable> ::= (
154                                 <OPERAND> term=Term()
                                  )?
156

158             <Unexpandable> ::= <numGivingEventFunction>
                                  | <numGivingQueueFunction>
160                               | <NUM_CONSTANT>
                                  | <ALPHA_CONSTANT>
162

164     <numGivingEventFunction> ::= <eventGivingFunction> "." ( <TIME_DEF>
                                                               | <ID_DEF>
166                                                          )
                                  | <IDENTIFIER>          "." (   <TIME_DEF>
168                                                            | <ID_DEF>
                                                             )
170
        <numGivingQueueFunction> ::= <IDENTIFIER> "." <COUNT_DEF>
172

174     <eventGivingFunction>  ::= <IDENTIFIER> "." (   <MYMATCH_DEF>
                                                      | <HISMATCH_DEF>
176                                                  )
                                                     "(" <IDENTIFIER> ")"
178

180 <Event>         ::= <EVENT_KEYWORD> <IDENTIFIER>
                        <USES_KEYWORD> <IDENTIFIER>
182                     <ON_TAP_KEYWORD> <NUM_CONSTANT>
                        <ISOLATE_KEYWORD>
184                       "(" (<Filter_Term>)? ( "," <Filter_Term> )* ")"
                        <ID_KEYWORD>  (    <AUTO_KEYWORD>
186                                      | <IDENTIFIER>
                                         | Hash()
188                                   )
                        <END_STATEMENT>
190
                 <Filter_Term> ::= <STAR>
192                                | ( < COMPARE>
                                       (    <NUM_CONSTANT>
194                                       | <ALPHA_CONSTANT>
                                          | <IDENTIFIER>
196                                    )
```

```
                                    )
198                                 | ( < EVAL_FUNCTION> <COMPARE>
                                        (    <NUM_CONSTANT>
200                                       | <ALPHA_CONSTANT>
                                          | <IDENTIFIER>
202                                     )
                                      )
204

206                         <Hash> ::= <HASH_KEYWORD> "(" <IDENTIFIER>
                                                        ("," <IDENTIFIER>)*
208                                                   ")"

210

212  <Primitive>     ::= <PRIMITIVE_KEYWORD> <IDENTIFIER>
                         "("
214                            (    <IDENTIFIER> ":" <Type>)?
                               ("," <IDENTIFIER> ":" <Type>)*
216                      ")" <END_STATEMENT>

218

220                         <Type> ::= <BIT><ARRAY_DEFINITION>
                                     | <BIT>
222                                  | <BYTE><ARRAY_DEFINITION>
                                     | <BYTE>
224                                  | <CHAR><ARRAY_DEFINITION>
                                     | <CHAR>
```

A.3. Aufbau des Compilers mit javaCC

Mit dem Compiler-Generator javaCC kann ein Compiler aus der Definition von Produktionsregeln generiert werden. Neben Vorschriften, welche die Syntax der entstehenden Sprache steuern, werden in javaCC Java-Codefragmente eingestreut, welche die jeweilige Produktion steuern. Nach einem Durchlauf des Generatorprogramms javaCC entsteht eine Reihe von Java-Klassen. Übersetzt und ausgeführt implementieren diese Klassen den spezifizierten Compiler.

Eine Produktionsregel wird durch eine Methode umgesetzt, die eigene Variablen spezifizieren und zudem Aufrufparameter verwenden kann. Damit kann der Austausch von Informationen zwischen Produktionsregeln sichergestellt werden.

Die folgende Darstellung beschreibt in groben Zügen den prinzipiellen Aufbau der Produktionsmethoden und gibt ihren Sourcecode wieder. Zusätzlich wird die Funktion des Compilers durch das nachfolgend dargestellte Umsetzungsbeispiel, das die generierten Klassen im Bezug zur zugrundeliegenden QoSSL-Definiton darstellt, verdeutlicht.

Einleitende Definitionen Bevor der eigentliche Aufbau des Compilers beginnt, werden Optionen für den Generator sowie globale Variablen festgelegt. Die vier Instanzen der Klasse *StringBuffer* werden benutzt, um sukzessive eine Verfeinerung der Klasse *Builder*, also der spezifischen Instanzierungshilfestellung, zu erstellen.

```
    options {
 2    LOOKAHEAD = 5;
      CHOICE_AMBIGUITY_CHECK = 2;
 4    OTHER_AMBIGUITY_CHECK = 1;
      STATIC = true;
 6    DEBUG_PARSER = false;
      DEBUG_LOOKAHEAD = false;
 8    DEBUG_TOKEN_MANAGER = false;
      ERROR_REPORTING = true;
10    JAVA_UNICODE_ESCAPE = false;
      UNICODE_INPUT = false;
12    IGNORE_CASE = false;
      USER_TOKEN_MANAGER = false;
14    USER_CHAR_STREAM = false;
      BUILD_PARSER = true;
16    BUILD_TOKEN_MANAGER = true;
      SANITY_CHECK = true;
18    FORCE_LA_CHECK = false;
    }
20

22  PARSER_BEGIN(QoSSL)

24  public class  QoSSL {

26


28    static  StringBuffer builderInit = new StringBuffer(" ");
      static  StringBuffer builderAttributes = new StringBuffer(" ");
30    static  StringBuffer builderConstructor = new StringBuffer(" ");
      static  StringBuffer builderConstructorHead = new StringBuffer(" ");
32

34    public static  void main(String args[]) throws ParseException {
        QoSSL parser = new QoSSL(System.in);
36      parser.Input();
      }
38

40


42

    }
44
    PARSER_END(QoSSL)
```

Token Der Aufbau des Compilers beginnt mit einer Reihe von lexikalischen Definitionen, der Festlegung sog. Tokens. Dabei werden beispielsweise Schlüsselwörter oder Rechenzeichen, der sog. syntactic suggar, festgelegt.

```
SKIP :
{
  " "
| "\t"
| "\n"
| "\r"
}

TOKEN :
{
  <LBRACE          : "{">

|<RBRACE          : "}">

|<END_STATEMENT : ";">

|<NULL              : "NULL">

|<UNIT              : "bit"|"bit/sec"|"sec"|"%"|"NULL">

|<FEATURE_KEYWORD : "FEATURE">

|<READING_KEYWORD : "READING">

|<EVENT_KEYWORD : "EVENT">

|<PRIMITIVE_KEYWORD : "PRIMITIVE">

|<USES_KEYWORD : "USES">

|<COMPUTES_KEYWORD: "COMPUTES">

|<UNIT_KEYWORD : "UNIT">

|<AS_KEYWORD  : "AS">

|<EVERY_KEYWORD : "EVERY">

|<JOIN_KEYWORD : "JOIN">

|<TO_KEYWORD  : "TO">

|<REQUIRES_KEYWORD : "REQUIRES">

|<ON_TAP_KEYWORD : "ON TAP">

|<ISOLATE_KEYWORD : "ISOLATE">
```

```
48
    | <ID_KEYWORD   : "ID">
50
    | <AUTO_KEYWORD : "AUTO">
52
    | <HASH_KEYWORD : "HASH">
54
    | <COUNT_DEF     : "count">
56
    | <MYMATCH_DEF : "myMatch">
58
    | <HISMATCH_DEF : "hisMatch">
60
    | <TIME_DEF        : "time">
62
    | <ID_DEF          : "id">
64
    | <AGGREGATION :  "ACF"      //  absolute cumulative frequency
66                    | "RCF"      //  relative  cumulative frequency
                      | "CF"       //  cumulative frequency
68                    | "AF"       //  absolute frequency
                      | "BOXPLOT" // create a boxplot
70                    | "MAVG"   //  moving avarge
                      | "SUM" >  //  simply sum up
72                                 //  more aggregation-functions
                                   //  to be added here
74

76  | <ALPHA_CONSTANT : ("'"(["a"-"z","A"-"Z","0"-"9","_","-"])*"'") > //alpha

78
    | <ARRAY_DEFINITION: "["["1"-"9"](["0"-"9"])*"]"
80                    | "[]"  // undefined length
                      >
82


84
    | <NUM_CONSTANT : ("-")? ("0")? <UNSIGNEDINTEGER> ("."["0"-"9"](["0"-"9"])*)?>
86
    | <UNSIGNEDINTEGER : (["1"-"9"])(["0"-"9"])*>
88
    | <STAR            : "*">
90
    | <PLUS            : "+">
92
    | <QUESTIONMARK : "?">
94
    | <OPERAND      : <STAR> | <PLUS> | "-" | "**" >
96
    | <CORRELATION  :  "left_equal_id"|
98                      "right_equal_id"|
```

```
                        "count"|
100                     "time_dif">
                        // more correlation-functions to be added here
102
    | <COMPARE       : "<"|"<="|"=="|"!="|">="|">" >
104
    | <CONSTANT_ARRAY : "["(<NUM_CONSTANT>|<ALPHA_CONSTANT>)
106                        (","(<NUM_CONSTANT>|<ALPHA_CONSTANT>))*
                        "]">
108
    | <BIT             : "BIT">
110
    | <BYTE            : "BYTE">
112
    | <CHAR            : "CHAR">
114
    | <EVAL_FUNCTION : "size">
116                     // more functions to be added here

118 | <IDENTIFIER      : [ "a"-"z","A"-"Z"](["a"-"z","A"-"Z","_","0"-"9"])*>

120 | <COMMENT        : "//" (~["\n"])*"\n">

122 | <DOT_SEPARATOR : ".">

124 }
```

QoSSL In einer von javaCC in ihrer Signatur festgelegten Methode *input* wird festgelegt, welche Ausdrücke in der Sprache QoSSL überhaupt zulässig sind. Analog zur Syntaxdefiniton sind die vier Blöcke Feature, Reading, Event und Primitive aufgeführt. Sie werden ihrerseits durch Methoden realisiert. Als Aufrufparameter erhalten sie Puffer zum sukzessiven Aufbau der Instanzierungshilfestellung, einer Verfeinerung der Klasse *Builder*.

```
 2  void Input() :
    {}
 4
    {
 6    // every statement may add code to the builder
      (<COMMENT>
 8              | Feature(builderInit, builderAttributes,
                          builderConstructorHead, builderConstructor)
10              | Reading(builderInit, builderAttributes, builderConstructor)
                | Event(builderInit, builderAttributes,
12                      builderConstructorHead, builderConstructor)
                | Primitive()
14    )* <EOF>
    }
```

Feature Als nächster Block werden die Produktionsregeln um das Schlüsselwort FEATURE in einer entsprechend bezeichneten Methode festgelegt. Die Belegungen von Identifikatoren und Funktionsbezeichnern werden in Variablen abgelegt. (Die dazu notwendigen Funktionalitäten werden von Compiler-Generator javaCC angeboten, so dass eine übersichtliche Code-Entwicklung möglich ist.

Zunächst werden innerhalb der Methode Teile der Instanzierungshilfestellung durch stückweises Zusammensetzen von Java-Code in den jeweiligen Puffern aufgebaut. Anschließend wird der Java-Code für eine Verfeinerung der Klasse *Feature* erzeugt und direkt in eine entsprechende Datei geschrieben. Die Ausgabe in Dateien wird dabei durch die Klasse *CodeWriter* verschattet.

Abschließend wird aus der Definition eines Features auch der Sourcecode für die Nachverarbeitung, also eine Verfeinerung der Klasse *PostProcessor*, erzeugt. Dabei wird die *send*-Methode nur als Rumpf implementiert, der die jeweils erzeugte Verfeinerung der Klasse *Feature* lediglich ausgibt. Damit ist ein Test der generierten Implementierung ohne Anbindung an ein Managementsystem möglich.

Zum Abschluss der Methode werden die durch den bisherigen Compilerlauf entstandenen Fragmente der Instanzierungshilfestellung zu einer Verfeinerung der Klasse *Builder* zusammengefasst und in eine Datei geschrieben. Dabei wird davon ausgegangen, dass die Definition eines Features in der QoSSL-Source erst dann vorgenommen wird, wenn alle Definitionen, auf die dabei aufgebaut wird, bereits vorgenommen wurden. Eine Prüfung, ob die Definition eines Features abgeschlossen werden kann, weil alle dafür notwendigen Teildefinitionen (Primitiven, Events, usw.) vorliegen, ist durch den Compiler möglich, wird aber in der vorliegenden prototypischen Implementieurng nicht durchgeführt.

```
 2
   void Feature(StringBuffer builderInit, StringBuffer builderAttributes,
 4                StringBuffer builderConstructorHead, StringBuffer builderConstructor) :
   {
 6
     Token featureName;
 8   Token readingCounter;
     Token readingName;
10   Token aggregationAbr;
     String aggregation;
12 }
   {
14   <FEATURE_KEYWORD> featureName=<IDENTIFIER>
     <JOIN_KEYWORD> readingCounter=<NUM_CONSTANT> readingName=<IDENTIFIER>
16   <TO_KEYWORD> aggregationAbr=<AGGREGATION>
     <END_STATEMENT>
18
     {
20     //build a postprocessor

22     builderAttributes.append(
       "␣␣␣␣PostProcessor_" + featureName.toString() + "\t\t\t" +
24     "␣the_" + featureName.toString() + "_PostProcessor;\n");
```

```
26      //build an instance
        builderConstructor.append("␣␣␣␣the_" + featureName.toString() +
28                                  "_PostProcessor␣=␣" +
                                    "new␣PostProcessor_" +
30                                  featureName.toString() +"();\n");

32      //bind postprocessor to eventcorrelator

34      builderInit.append(
        "␣␣␣␣the_" + readingName.toString() +
36          "_EventCorrelator.bind(the_" + featureName.toString() + "_PostProcessor);\n");

38
        CodeWriter feature = new CodeWriter("Feature_" + featureName.toString());
40

42      //                  "ACF"     //  absolute cumulative frequency
        //                | " RCF"     //  relative  cumulative frequency
44      //                | " CF"      //  cumulative frequency
        //                | " AF"      //  absolute frequency
46      //                | " BOXPLOT" // create a boxplot
        //                | " MAVG" > // moving avarge
48      //                            //  more aggregation-functions
        //                            //  to be added here
50

52      aggregation = "notImplemented" + aggregationAbr.toString();

54      //  determine class corresponding to aggregation method

56      if (aggregationAbr.toString().equals("ACF")) {
          aggregation = "AbsoluteCumulativeFrequnecy";
58      }
        if (aggregationAbr.toString().equals("RCF")) {
60        aggregation = "RelativeCumulativeFrequnecy";
        }
62      if (aggregationAbr.toString().equals("CF")) {
          aggregation = "CumulativeFrequnecy";
64      }
        if (aggregationAbr.toString().equals("AF")) {
66        aggregation = "AbsoluteFrequnecy";
        }
68      if (aggregationAbr.toString().equals("BOXPLOT")) {
          aggregation = "Boxplot";
70      }
        if (aggregationAbr.toString().equals("MAVG")) {
72        aggregation = "MovingAvarage";
        }
74      if (aggregationAbr.toString().equals("SUM")) {
          aggregation = "Sum";
76      }
```

```
        feature.println("package␣qossl.generated;\n");
 78     feature.println("import␣qossl.base.*;\n");

 80     feature.println(  "public␣class␣Feature_" + featureName + "␣extends␣" + aggregation
                        + "␣implements␣Feature␣{");
 82     feature.println("}␣//␣end␣of␣class␣" + featureName);

 84

 86

 88     CodeWriter postProcessor = new CodeWriter("PostProcessor_"
                                                  + featureName.toString());
 90
        postProcessor.println("package␣qossl.generated;\n");
 92     postProcessor.println("import␣qossl.base.*;\n");

 94     postProcessor.println("public␣class␣" + "PostProcessor_"
                              + featureName.toString()
 96                           + "␣extends␣PostProcessor␣{\n");

 98     //add an attribute to store the receiver (a PostProcessor only has one)
        postProcessor.println("␣␣FeatureReceiver␣theReceiver;");
100
        //add an attribute to count readings
102     postProcessor.println("␣␣int␣readingCounter␣=␣0;");

104     //attribute to store the feature generated
        postProcessor.println("␣␣Feature_" + featureName + "␣featureToBuild;\n");
106
        //build a bind method
108     postProcessor.println("␣␣public␣void␣bind(FeatureReceiver␣receiverToRegister)␣{\n" +
                              "␣␣␣␣theReceiver␣=␣receiverToRegister;\n" +
110                           "␣␣}\n\n");

112     //build receive method which counts received reading and
        //add them to the feature which is than sent
114     postProcessor.println("␣␣public␣void␣receive(Reading␣aReading)␣{\n" +
                              "␣␣␣␣if␣(readingCounter==0)␣{\n" +
116                           "␣␣␣␣␣␣featureToBuild␣=␣new␣Feature_" + featureName + "();\n" +
                              "␣␣␣␣}\n" +
118                           "␣␣␣␣readingCounter␣++;\n" +
                              "␣␣␣␣featureToBuild.addValue(aReading.getValue());\n" +
120                           "␣␣␣␣if␣(readingCounter=="
                                    + readingCounter.toString() + ")␣{\n" +
122                           "␣␣␣␣␣␣send(featureToBuild);\n" +
                              "␣␣␣␣␣␣//reset␣counter\n" +
124                           "␣␣␣␣␣␣readingCounter␣=␣0;\n" +
                              "␣␣␣␣}\n" +
126                           "␣␣}\n\n");
```

```
128         // build a send method which is currently left blank
            postProcessor.println("  public void send(Feature featureToSend) {\n" +
130                                 "     //uncomment if there really is a receiver\n" +
                                    "     //theReceiver.receive(featureToSend);\n\n" +
132                                 "     //simply print the current value of the feature\n" +
                                    "     System.out.println(\"Feature [" + featureName
134                                      + "] : <\" +featureToSend.+\">\");\n" +
                                    "  }\n\n");
136
            postProcessor.println("} // end of class " + "PostProcessor_"
138                                                     + featureName.toString());

140

142         // finaly  complete builderSource and write to file

144         CodeWriter builder = new CodeWriter("Builder_" + featureName.toString());

146
            // write the class  header
148         builder.println("package qossl.generated;\n");
            builder.println("import qossl.base.*;\n\n");
150
            builder.println("public class Builder_" + featureName.toString()+" {");
152         builder.println("");
            builder.println("   //measurement components\n");
154         builder.println(builderAttributes.toString());
            builder.println("");
156          //  the constructor
            builder.println("  public Builder_" + featureName.toString() + "("
158                               + builderConstructorHead.toString() +") {");
            builder.println(builderConstructor.toString());
160         builder.println("  };\n");

162         builder.println("    public void buildInstances() {\n");
            builder.println(builderInit.toString ());
164         builder.println("    }\n");
            builder.println("}\n");
166
            // reset  buffers
168         builderInit = new StringBuffer();
            builderAttributes = new StringBuffer();
170         builderConstructor = new StringBuffer();
            builderConstructorHead = new StringBuffer();
172     }
      }
```

Reading Im folgenden Konstrukt wird die Umsetzung der Spezifikation eines Messwertes, eingeleitet durch das Schlüsselwort READING, festgelegt. Analog zum bisherigen Vorgehen werden die Belegungen von Identifikatoren und Referenzen auf bestehende Definitionen wieder in Variablen gespeichert. JavaCC lässt praktisch die beliebige Verschränkung von Syntax-Definitionen und Anweisungen zu, so dass die Umsetzung einer Syntaxfestlegung direkt an der Stelle ihres Auftretens durchgeführt werden kann. Somit lassen sich mögliche Mehrfachaufrufe eines Codefragments einfach umsetzen.

Dieser Mechanismus wird zu Beginn der Methode angewandt, um die beliebige Anzahl von Typen von Ereignissen (Events), die zur Berechnung eines Messwertes (Reading) notwendig sind, in der generierten Verfeinerung der Klasse *EventCorrelator* abzubilden.

Ebenso werden die Codefragmente für die Instanzierungshilfestellung weiterhin ergänzt, die Verfeinerung der Klasse *EventCorrelator* in eine entsprechende Datei geschrieben sowie eine Verfeinerung von *Reading* erzeugt und in einer Datei abgelegt.

Zum Aufbau der Korrelationsfunktionalität werden von der Methode *Reading* weitere Methoden verwendet, die beispielsweise die Umsetzung von Rechenvorschriften oder den Zugriff auf Warteschlangen bei der Verarbeitung von Events umsetzen.

```
    void Reading(StringBuffer builderInit, StringBuffer builderAttributes,
 2                StringBuffer builderConstructor) :
    {
 4    Token readingName;
      Token eventName;
 6    Token queueName;
      CodeWriter eventCorrelator;
 8    String variables = "";
      String constructor ="";
10    String methods = "";
      String correlation = "";
12  }
    {
14    <READING_KEYWORD> readingName=<IDENTIFIER>
      //build a  file  defining the corresponding class
16    {
        eventCorrelator = new CodeWriter("EventCorrelator_" + readingName.toString());
18
        eventCorrelator.println("package␣qossl.generated;\n");
20      eventCorrelator.println("import␣qossl.base.*;\n");
        eventCorrelator.println("public␣class␣EventCorrelator_" + readingName.toString() +
22                              "␣extends␣EventCorrelator␣{");

24    }
      <REQUIRES_KEYWORD> eventName=<IDENTIFIER>
26                       <AS_KEYWORD> queueName=<IDENTIFIER>
                         //requires at  least  one event
28                       {
                           //add queue-definition to variables-string
30                         variables = "␣␣Queue␣" + queueName.toString() + ";\n";
```

```
32                          //add init line to constructor
                            constructor = "␣␣␣␣" + queueName.toString() +
34                                          "␣=␣new␣Queue();\n";

36                          //add receive method
                            methods = "␣␣public␣void␣receive␣(Event_" +
38                                  eventName.toString() + "␣receivedEvent)␣{\n" +
                                    "␣␣␣␣" + queueName.toString() +
40                                      ".add(receivedEvent);\n" +
                                    "␣␣␣␣correlate();\n" +
42                                  "␣␣}\n\n";

44                      }
                        (
46                        "," eventName=<IDENTIFIER>
                            <AS_KEYWORD> queueName=<IDENTIFIER>
48                        {
                            //for each newly defined event queue pair
50                          //add queue-definition to variables-string
                            variables = variables + "␣␣Queue␣" + queueName + ";\n";
52
                            //add init line to constructor
54                          constructor = "␣␣␣␣" + queueName.toString() +
                                              "␣=␣new␣Queue();\n";
56
                            //add receive method
58                          methods = methods + "\n␣␣public␣void␣receive␣(Event_" +
                                    eventName.toString() + "␣receivedEvent)␣{\n" +
60                                  "␣␣␣␣" + queueName.toString() +
                                        ".add(receivedEvent);\n" +
62                                  "␣␣␣␣correlate();\n" +
                                    "␣␣}\n\n";
64

66                        }
                        )*
68                      //allow multiple required events
    <COMPUTES_KEYWORD> correlation = Term()
70  <UNIT_KEYWORD> <UNIT> <END_STATEMENT>
    {
72    //build a EventCorrelator

74    builderAttributes.append(
      "␣␣␣␣EventCorrelator_" + readingName.toString() + "\t\t\t" +
76    "␣the_" + readingName.toString() + "_EventCorrelator;\n");

78    //build an instance

80    builderConstructor.append("␣␣␣␣the_" + readingName.toString() +
                                "_EventCorrelator␣=␣" +
82                              "new␣EventCorrelator_" +
```

```
                                        readingName.toString() + "();\n");
84
        //bind eventcorrelator to eventgenerator
86
        builderInit.append(
88      "    the_" + eventName.toString() +
            "_EventGenerator.bind(the_" + readingName.toString() +
90          "_EventCorrelator);\n");

92

94
        //  finaly  write variables to  file
96      eventCorrelator.println(variables);

98      //  write constructor
        eventCorrelator.println("  public EventCorrelator_" +
100                              readingName.toString() + " () {");
        eventCorrelator.println(constructor);
102     eventCorrelator.println("  }\n\n");

104     //write methods
        eventCorrelator.println(methods);
106
        //write correlation method
108     //quick hack only
        correlation = "  public void correlate(){\n" +
110                   "    internValue = " + correlation +";\n"+
                      "    //send Value to PostProcessor\n" +
112                   "    //processed events and error to be added later\n" +
                      "    send(new Reading_" + readingName.toString() +
114                        " (internValue,1,0) );\n" +
                      "  }\n";
116     eventCorrelator.println(correlation);

118     //close  class  def
        eventCorrelator.println("}");
120
        //generate the corresponding reading class
122     CodeWriter reading = new CodeWriter("Reading_" + readingName.toString());

124     reading.println("package qossl.generated;\n");
        reading.println("import qossl.base.*;\n");
126     reading.println("public class Reading_" + readingName.toString() +
                        " extends Reading {\n");
128     //extra constructor required
        reading.println("  Reading_" + readingName.toString() +
130                       "(double valueToSet, long processedEventsToSet, " +
                            "double relativeErrorToSet) {\n" +
132                     "    super(valueToSet, processedEventsToSet, " +
                            "relativeErrorToSet);\n" +
```

```
134                             "␣␣}\n");
          reading.println("};");
136
      }
138 }

140
    String Term() :
142 {
      String unexpandable;
144   String expandable;
      String term;
146 }
    {
148       unexpandable = Unexpandable() expandable = Expandable()
          {
150         term = unexpandable + "␣" + expandable;
            return term;
152       }
        | "(" term = Term() ")"
154
        {
156       return term;
        }
158 }

160 String Expandable() :
    {
162   Token operand;
      String term;
164 }
    {
166   (
        operand = <OPERAND> term=Term()
168     {
          return operand.toString() + "␣" + term;
170     }
      )?
172   {
        return "";
174   }
    }
176

178 String Unexpandable() :
    {
180   Token num_constant=null;
      Token alpha_constant=null;
182   String numGivingEventFunction;
      String numGivingQueueFunction;
184 }
```

```
    {
186     numGivingEventFunction = numGivingEventFunction()
        {
188       return numGivingEventFunction;
        }
190   | numGivingQueueFunction = numGivingQueueFunction()
        {
192       return numGivingQueueFunction;
        }
194   | <NUM_CONSTANT>
        {
196       return num_constant.toString();
        }
198   | <ALPHA_CONSTANT>
        {
200       return alpha_constant.toString();
        }
202 }

204
    String numGivingEventFunction() :
206 {
      String eventGivingFunction="";
208   Token identifier;
    }
210 {
        eventGivingFunction =
212         eventGivingFunction() "." (   <TIME_DEF>
                                          {
214                                         return eventGivingFunction + ".getTime()";
                                          }
216                                     | <ID_DEF>
                                          {
218                                         return eventGivingFunction + ".getId()";
                                          }
220                                   )

222   | identifier = <IDENTIFIER> "." (   <TIME_DEF>
                                          {
224                                         return identifier.toString() + ".getTime()";
                                          }
226                                     | <ID_DEF>
                                          {
228                                         return identifier.toString() + ".getId()";
                                          }
230                                   )
        {
232       return "ERROR";
        }
234 }
```

```
236

238  String numGivingQueueFunction() :
     {
240    Token identifier;
     }
242  {
       identifier = <IDENTIFIER> "." <COUNT_DEF>
244    {
        return identifier.toString() + ".getSize()";
246    }
     }
248

250  String eventGivingFunction() :
     {
252    Token identifier;
       String toReturn;
254  }
     {
256    identifier= <IDENTIFIER> "." (   <MYMATCH_DEF>
                                        {
258                                       toReturn = identifier.toString() + ".myMatch";
                                        }
260                                   | <HISMATCH_DEF>
                                        {
262                                       toReturn = identifier.toString() + ".hisMatch";
                                        }
264                                 ) "(" identifier=<IDENTIFIER> ")"
                                    {
266                                   return  toReturn + "(" + identifier.toString() + ")";
                                    }
268  }
```

Event Mit dem Schlüsslewort EVENT wird die Definition eines Ereignisses eingeleitet. Die Methode *Event* setzt die entsprechenden QoSSL-Syntax-Definitionen um. Sie generiert eine Verfeinerung der Klasse *EventGenerator* sowie eine Verfeinerung der Klasse *Event.* Zudem wird auch in diesem Generierungsschritt die Instanzierungshilfestellung erweitert.

Die Methode *Event* stützt sich auf die Methoden *Filter_Term* zum Aufbau der Filterfunktionalität und *Hash* zum Einfügen von Hashfunktionen. Beide Methoden sind im Rahmen der prototypischen Implementierung nur im Rumpf ausgeführt, so dass eine korrekte Prüfung der Syntax sichergestellt ist.

```
     void Event(StringBuffer builderInit, StringBuffer builderAttributes,
2                StringBuffer builderConstructorHead, StringBuffer builderConstructor) :
     {
4      Token eventName;
       Token primitiveName;
6      Token tapNumber;
```

```
    CodeWriter event;
 8  CodeWriter eventGenerator;
    String  filter ="";
10 }
   {
12  <EVENT_KEYWORD> eventName = <IDENTIFIER>
    <USES_KEYWORD> primitiveName = <IDENTIFIER>
14  <ON_TAP_KEYWORD> tapNumber = <NUM_CONSTANT>
    {
16    //build an EventGenerator

18    builderAttributes.append(
      "␣␣␣␣EventGenerator_" + eventName.toString() + "␣␣" +
20    "␣the_" + eventName.toString() + "_EventGenerator;\n");

22    //add an attribute for the tap
      builderAttributes.append("␣␣␣␣Tap␣␣␣tap" + tapNumber.toString() +";\n");
24
      //build an instance
26
      builderConstructor.append("␣␣␣␣the_" + eventName.toString() + "_EventGenerator␣=␣" +
28                                "new␣EventGenerator_" + eventName.toString() + "();\n");

30    //bind eventgenerator to tap
      //comment source
32    builderInit.append(
      "␣␣␣␣tap" + tapNumber.toString() + ".bind(the_"
34          + eventName.toString() + "_EventGenerator);\n");

36
      //buildup constructor parameter line
38
      if (builderConstructorHead.length()==0) {
40      // first  time
        builderConstructorHead.append("Tap␣_tap" + tapNumber.toString());
42    }
      else {
44      builderConstructorHead.append(",␣Tap␣_tap" + tapNumber.toString());
      }
46
      //build up constructor statement
48    builderConstructor.append("␣␣␣␣tap" + tapNumber.toString() + "␣=␣" +
                                "␣_tap" + tapNumber.toString() +";\n");
50

52    //write a class  file  for  the event
      event = new CodeWriter("Event_" + eventName.toString());
54
      event.println("package␣qossl.generated;\n");
56    event.println("import␣qossl.base.*;\n");
      event.println("public␣class␣Event_" + eventName.toString() +
```

```
58                          " extends Event {");

60      // event needs an extra constructor
        event.println("  public Event_" + eventName.toString() +
62                          " (String nameToSet, long idToSet, long timeStampToSet) {\n" +
                        "    super(nameToSet,idToSet,timeStampToSet);\n" +
64                      "  }");

66      // close the class
        event.println("};" );
68

70      // write a class for the event generator

72      eventGenerator = new CodeWriter("EventGenerator_" + eventName.toString());

74      eventGenerator.println("package qossl.generated;\n");
        eventGenerator.println("import qossl.base.*;\n");
76      eventGenerator.println("public class EventGenerator_" + eventName.toString() +
                        " extends EventGenerator {");
78



80



82

    }
84  <ISOLATE_KEYWORD> "(" (Filter_Term())? ( "," Filter_Term() )* ")"
    <ID_KEYWORD> (    <AUTO_KEYWORD>
86                  | <IDENTIFIER>
                    | Hash()
88                )
    <END_STATEMENT>
90
    {
92    // continue to build the event generator
      eventGenerator.println("\n" +
94                           "  public void receive(Primitive aPrimitive) {\n" +
                             "  //down cast because class-type is known \n" +
96                           "    Primitive_" + primitiveName.toString() +
                                   " myPrim = (Primitive_" +
98                                 primitiveName.toString() + ") aPrimitive;\n" +
                             "    //filter ();\ n" +
100                          "    Event_" + eventName.toString() + " newEvent = " +
                                 "new Event_" + eventName.toString() +
102                              "(\"newEvent\",0,0);\n" +
                             "    //suitable constructor to be added here\n" +
104                          "    send(newEvent);\n" +
                             "  }\n\n");
106   // close class
      eventGenerator.println("}");
108 }
```

```
    }
110

112 void Filter_Term () :
    {}
114 {
      <STAR>
116   | ( < COMPARE>
        (   <NUM_CONSTANT>
118       | <ALPHA_CONSTANT>
          | <IDENTIFIER>
120     )
      )
122   | ( < EVAL_FUNCTION> <COMPARE>
        (   <NUM_CONSTANT>
124       | <ALPHA_CONSTANT>
          | <IDENTIFIER>
126     )
      )
128 }

130
    void Hash() :
132 {}
    {
134   <HASH_KEYWORD> "(" <IDENTIFIER> ("," <IDENTIFIER>)* ")"
    }
```

Primitive Produktionsregeln zur Erzeugung von Verfeinerungen der Klasse *Primitive* sind in der Methode *Primitive* zusammengefasst. Der generierte Java-Quellcode wird in einer entsprechenden Datei abgelegt. Die unterstützten Datentypen werden in der Methode *Type* festgelegt. In der prototypischen Implementierung werden nur einfache Datentypen und Felder dieser Typen unterstützt. Diese werden eins zu eins in entsprechende Java-Datentypen übersetzt.

```
    void Primitive () :
  2 {
      StringBuffer javaTypeFront = new StringBuffer();
  4   StringBuffer javaTypeBack = new StringBuffer();
      StringBuffer newInstance = new StringBuffer();
  6   Token primitiveName;
      Token attributeName;
  8   CodeWriter primitive;
    }
 10 {
      <PRIMITIVE_KEYWORD> primitiveName = <IDENTIFIER>
 12   {
        //write a class for the primitive
 14     primitive = new CodeWriter("Primitive_" + primitiveName.toString());
```

```
16      primitive.println("package␣qossl.generated;\n");
        primitive.println("import␣qossl.base.*;\n");
18      primitive.println("public␣class␣Primitive_" + primitiveName.toString() +
                          "␣extends␣Primitive␣{");
20
    }
22  "("
        (       attributeName = <IDENTIFIER> ":"
24              Type(javaTypeFront, javaTypeBack, newInstance)
          {
26          primitive.println("␣␣" + javaTypeFront + "␣" +
            attributeName.toString() + javaTypeBack + ";");
28        }
        )?
30      (
          "," attributeName = <IDENTIFIER> ":"
32            Type(javaTypeFront, javaTypeBack, newInstance)
          {
34          primitive.println("␣␣" + javaTypeFront + "␣" +
            attributeName.toString() + javaTypeBack + ";");
36        }

38      )*
    ")"
40  <END_STATEMENT>
    {
42    primitive.println("}");
    }
44  }

46

48  // returns stringbuffers to rebuild a java typedefinition
    void Type(StringBuffer javaTypeFront, StringBuffer javaTypeBack,
50            StringBuffer newInstance) :
    {
52    Token t;
      javaTypeFront.setLength(0);
54    javaTypeBack.setLength(0);
      newInstance.setLength(0);
56  }
    {
58                      <BIT> t= <ARRAY_DEFINITION>
                          {
60                          javaTypeFront.append("boolean");
                            javaTypeBack.append("[]");
62                          if (t.toString().equals("[]")) {
                              newInstance.append("boolean[1]");
64                          }
                            else {
66                            newInstance.append("boolean" + t.toString());
```

```
                    }
68                                  }
                  | <BIT>
70                  {
                      javaTypeFront.append("boolean");
72                    newInstance.append(" ");
                                    }
74
                  | <BYTE>t=<ARRAY_DEFINITION>
76                  {
                      javaTypeFront.append("byte");
78                    javaTypeBack.append("[]");
                      if (t.toString().equals("[]")) {
80                      newInstance.append("byte[1]");
                      }
82                    else {
                        newInstance.append("byte" + t.toString());
84                    }
                                    }
86
                  | <BYTE>
88                  {
                      javaTypeFront.append("byte");
90                    newInstance.append(" ");

92                                  }

94                | <CHAR>t=<ARRAY_DEFINITION>
                    {
96                    javaTypeFront.append("char[]");
                      javaTypeBack.append("[]");
98                    if (t.toString().equals("[]")) {
                        newInstance.append("char[1]");
100                   }
                      else {
102                     newInstance.append("char" + t.toString());
                      }
104                                 }

106               | <CHAR>
                    {
108                   javaTypeFront.append("char");
                      newInstance.append(" ");
110
                                    }
112

114
    }
```

A.4. Übersetzung einer QoSSL-Definition

Zum Abschluss der Beschreibung der prototypischen Implementierung wird in diesem Abschnitt die Definition eines Dienstgütemerkmals zusammen mit den daraus generierten Klassen vorgestellt. Damit wird, am Beispiel, nochmals die Funktion des QoSSL-Compilers demonstriert.

A.4.1. QoSSL-Definition

Die folgende Definition in QoSSL definiert eine einfache Primitive *socket_send* und darauf aufbauend das Event *any_data*, das immer beim Aufruf der Primitive erzeugt wird. Die Primitive spiegelt den Aufruf einer send-Routine in einer Socket-Implementierung. Als Messwert wird die Anzahl der Events festgelegt. Da keine weiteren Korrelationen stattfinden, wird als Messwert immer nach dem Eintreffen eines Events 1 zurückgegeben. Die anschließende Definition des Dienstgütemerkmals, des sog. Features, legt fest, dass die Messwerte schlicht summiert werden sollen. Im Ergebnis zeigt das auf diese Art und Weise definierte Dienstgütemerkmal die Anzahl aller send-Aufrufe an einem Socket seit Beginn der Messung.

```
    PRIMITIVE socket_send ( src:       BYTE[4],
 2                          dest:      BYTE[4],
                            src_port:  BYTE[2],
 4                          dest_port: BYTE[2],
                            data:      BYTE[]
 6                        );

 8  EVENT any_data USES socket_data
                    ON TAP 1
10                  ISOLATE (*,*,*,*,*)
                    ID      AUTO;
12

14  READING data_transfer
            REQUIRES any_data      AS trans
16          COMPUTES trans.count
            UNIT NULL;
18
    FEATURE data_transfer_rate
20          JOIN 1 data_transfer TO SUM;
```

A.4.2. generierte Klassen

Durch den Aufruf des QoSSL-Compilers mit dem oben beschriebenen QoSSL-Quellcode werden die im Folgenden beschriebenen Klassen erzeugt.

Primitive_socket_send.java Die Klasse *Primitive_socket_send* wird als Verfeinerung der Klasse *Primitive* aufgebaut. Die Klasse wird ohne Methoden erzeugt und die definierten Attribute werden *public* zur Verfügung gestellt. Eine korrekte Belegung muss durch eine Instanz der Klasse *Tap* sichergestellt werden.

```
// Generated with QoSSL [Primitive_socket_send.java]
package qossl.generated;

import qossl.base.*;

public class Primitive_socket_send extends Primitive {
  public byte src [];
  public byte dest[];
  public byte src_port[];
  public dest_port[];
  public data[];
}
```

SpecialTap.java Zum Test der generierten Klassen wurde neben dem Testprogramm selbst, das am Ende dieser Auflistung beschrieben ist, mit der Klasse *SpecialTap* eine Verfeinerung der Klasse *Tap* implementiert. Sie enthält die Methode *doIt*, welche die Anbindung an einen realen SAP simmuliert und Instanzen der Klasse *Primitive_socket_send* erzeugt und verschickt. Die Attribute werden dabei nur zu Demonstrationszwecken belegt. Um den zeitlichen Ablauf einer Messung simmulieren zu können, wird zwischen den Aufrufen der *send*-Methode eine zufällig ausgewählte Zeit gewartet.

```
import qossl.generated.*;
import qossl.base.*;


  class SpecialTap extends Tap {

   public void doIt () {

    Primitive_socket_send P1 = new Primitive_socket_data();
    P1.src = new byte[2];
    P1.src[0] = 110;

    send (P1);
     java. util .Random r = new java.util.Random();

     try {
              Thread.currentThread().sleep((long)(r.nextGaussian()*100)+1100);
         }
     catch (java.lang.InterruptedException e){};
    P1 = new Primitive_socket_send();
    P1.src = new byte[2];
    P1.src[0] = 123;
    send (P1);
   }

  };
```

EventGenerator_any_data.java Einen Ereignisgenerator entsprechend der in QoSSL vorgenommenen Definition spezifiziert die Klasse *EventGenerator_any_data*. In der *receive*-Methode wird zunächst ein expliziter down-cast durchgeführt, also die empfangene Instanz der Klasse *Primitive* in eine Instanz der Unterklasse *Primitive_socket_send* konventiert. Diese Konvertierung ist notwendig, damit die standardisierte *send*-Methode der Klasse *Tap* zum Übertragen von Primitiven benutzt werden kann.

Die Generierung von Events aus Primitiven ist im Compiler noch nicht vollständig implementiert. Insbesondere fehlen die Funktionen zum Aufbau einer eindeutigen Identifikation des Events oder zum Filtern von Primitiven. Im generierten Java-Code ist dieser Umstand durch entsprechende Kommentare gekennzeichnet.

```
    // Generated with QoSSL [EventGenerator_any_data.java]
 2  package qossl.generated;

 4  import qossl.base.*;

 6  public class EventGenerator_any_data extends EventGenerator {

 8    public void receive(Primitive aPrimitive) {
      // down cast because class-type is known
10      Primitive_socket_send myPrim = (Primitive_socket_send) aPrimitive;
        // filter ();
12      Event_any_data newEvent = new Event_any_data("newEvent",0,0);
        // suitable constructor to be added here
14      send(newEvent);
      }
16

18  }
```

Event_any_data.java Aus der Spezifikation des zur erzeugenden Events wird vom Compiler die Klasse *Event_any_data* generiert. Da nach den Vorgaben des Designs hier nur die konzeptionelle Vererbung Anwendung findet, wird neben der Definition der Klasse als Verfeinerung der Klasse *Event* lediglich ein Konstruktor generiert, der den Konstruktor der Oberklasse aufruft. Somit können keine Instanzen mit ungültigen Attributbelegungen erzeugt werden.

```
    // Generated with QoSSL [Event_any_data.java]
 2  package qossl.generated;

 4  import qossl.base.*;

 6  public class Event_any_data extends Event {
      public Event_any_data (String nameToSet, long idToSet, long timeStampToSet) {
 8      super(nameToSet,idToSet,timeStampToSet);
      }
10  };
```

EventCorrelator_data_transfer.java Die für die Berechnung des Messwertes (reading) notwendige Korrelationsfunktionalität wird in einer Verfeinerung der Klasse *EventCorrelator* nach den Vorgaben aus der QoSSL-Definition nach dem Schlüsselwort READING generiert.

Entsprechend der Vorgaben nach dem Schlüsselwort REQUIRES werden vom Compiler Queues als Attribute mit den in der QoSS-Definition angegebenen Namen definiert und im Konstruktor initialisiert.

Für jeden Event-Typ wird (in diesem Fall genau) eine *receive*-Methode generiert, die das empfangene Event der jeweiligen Queue hinzufügt und die Methode *correlate* zur eigentlichen Berechnung des Messwertes aufruft.

Die Generierung dieser Methode ist momentan nur in groben Zügen implementiert. Das prinzipiell notwendige Vorgehen wird aber deutlich: Der interne Messwert muss nach der angegebenen Messvorschrift berechnet werden, eine Instanz der Klasse *Reading_data_transfer* muss aufgebaut und versandt werden.

```
    // Generated with QoSSL [EventCorrelator_data_transfer.java]
 2  package qossl.generated;

 4  import qossl.base.*;

 6  public class EventCorrelator_data_transfer extends EventCorrelator {
      Queue trans;
 8
      public EventCorrelator_data_transfer () {
10      trans = new Queue();

12    }

14
      public void receive (Event_any_data receivedEvent) {
16      trans.add(receivedEvent);
        correlate ();
18    }

20
      public void correlate(){
22      internValue = trans.getSize() ;
        //send Value to PostProcessor
24      //processed events and error to be added later
        send(new Reading_data_transfer (internValue,1,0) );
26    }

28  }
```

Reading_data_transfer.java Die Klasse *Reading_data_transfer* wird analog zur Klasse *Event_any_data* durch konzeptionelle Vererbung aus der Klasse *Reading* erzeugt. Vom Compiler wird dazu lediglich das Klassengerüst und ein entsprechender Konstruktor erstellt.

```
// Generated with QoSSL [Reading_data_transfer.java]
package qossl.generated;

import qossl.base.*;

public class Reading_data_transfer extends Reading {

  Reading_data_transfer(double valueToSet, long processedEventsToSet,
                        double relativeErrorToSet)
  {
    super(valueToSet, processedEventsToSet, relativeErrorToSet);
  }

};
```

PostProcessor_data_transfer_rate.java Die Klasse *PostProcessor_data_transfer* wird vom Compiler auf Basis des Interfaces *PostProcessor* erstellt. Das zu erzeugende Dienstgütemerkmal (Feature) wird als Attribut angelegt und in der Methode *receive* immer beim Empfang eines Messwertes neu berechnet. Das Ergebnis wird weiter versandt, wenn die Anzahl der aggregierten Messwerte (gespeichert im Attribut *readingCounter*) der durch die Spezifikation nach dem Schlüsselwort JOIN angegebenen Anzahl entspricht.

Die *send*-Methode ist in der vorliegenden Implementierung nur im Rumpf erstellt und führt keine Übertragung im eigentlichen Sinn durch, sondern gibt das berechnete Dienstgütemerkmal aus. Damit kann ein Test der Implementierung auch ohne Vorhandensein eines Managementsystems durchgeführt werden.

```
// Generated with QoSSL [PostProcessor_data_transfer_rate.java]
package qossl.generated;

import qossl.base.*;

public class PostProcessor_data_transfer_rate implements PostProcessor {

  FeatureReceiver theReceiver;
  int readingCounter = 0;
  Feature_data_transfer_rate featureToBuild;

  public void bind(FeatureReceiver receiverToRegister) {
    theReceiver = receiverToRegister;
  }


  public void receive(Reading aReading) {
    if (readingCounter==0) {
      featureToBuild = new Feature_data_transfer_rate();
    }
    readingCounter ++;
    featureToBuild.addValue(aReading.getValue());
```

```
        if (readingCounter==1) {
24        send(featureToBuild);
          // reset counter
26        readingCounter = 0;
        }
28    }

30
      public void send(Feature featureToSend) {
32      // uncomment if there really is a receiver
        // theReceiver.receive(featureToSend);
34
        // simply print the current value of the feature
36      System.out.println("Feature␣[data_transfer_rate]␣:␣<" + featureToSend + ">");
      }
38

40  } // end of class PostProcessor_data_transfer_rate
```

Feature_data_transfer_rate.java Zwar schlägt das Design für alle Datenübertragungsklassen, also auch für die Klasse *Feature_data_transfer_rate*, die konzeptionelle Vererbung von den jeweiligen Basisklassen vor, allerdings fordert es aber auch den Aufbau eines Aggregationsrepositories. Deshalb ist *Feature* als Interface implementiert und ermöglicht es damit, dass die Klasse *Feature_data_transfer_rate* zusätzlich von der Klasse *Sum* erbt, welche die eigentliche Aggregationsfunktionalität implementiert. Die resulierende einfache Definition wird entsprechend vom Compiler generiert.

```
    // Generated with QoSSL [Feature_data_transfer_rate.java]
2   package qossl.generated;

4   import qossl.base.*;

6   public class Feature_data_transfer_rate extends Sum implements Feature {
    } // end of class data_transfer_rate
```

Builder_data_transfer_rate.java Der Compiler erzeugt auch die spezifische Instanzierungshilfestellung in Form einer Verfeinerung der Klasse *Builder*. Aus dem unten dargestellten Beispiel ist das dabei angewandte Vorgehen gut erkennbar: Für jede Komponente des Messsystems wird ein entsprechend typisiertes Attribut angelegt. Der Konstruktor wird so aufgebaut, dass alle notwendigen Anbindungen (Taps) als Parameter angegeben werden müssen. Im Konstruktor selbst werden die notwendigen Instanzen des Messsystems aufgebaut. Die Methode *buildInstances* bindet schließlich die einzelnen Komponenten aneinander.

```
// Generated with QoSSL [Builder_data_transfer_rate.java]
package qossl.generated;

import qossl.base.*;


public class Builder_data_transfer_rate {

    // measurement components

     EventGenerator_any_data the_any_data_EventGenerator;
     Tap    tap1;
     EventCorrelator_data_transfer                the_data_transfer_EventCorrelator;
     PostProcessor_data_transfer_rate             the_data_transfer_rate_PostProcessor;


   public Builder_data_transfer_rate(Tap _tap1) {
     the_any_data_EventGenerator = new EventGenerator_any_data();
     tap1 =  _tap1;
     the_data_transfer_EventCorrelator = new EventCorrelator_data_transfer();
     the_data_transfer_rate_PostProcessor = new PostProcessor_data_transfer_rate();

   };

    public void buildInstances() {

     tap1.bind(the_any_data_EventGenerator);
     the_any_data_EventGenerator.bind(the_data_transfer_EventCorrelator);
     the_data_transfer_EventCorrelator.bind(the_data_transfer_rate_PostProcessor);

    }

}
```

Test.java Das Testprogramm zeigt den Aufbau eines Messsystems zu Testzwecken einmal in Einzelschritten und einmal (ab Zeile 42) unter Verwendung der generierten Instanzierungshilfestellung. Schon der Code-Umfang macht die Vereinfachungen deutlich, die der Entwickler dank der generierten Instanzierungshilfestellung in Anspruch nehmen kann.

```
import qossl.generated.*;
import qossl.base.*;


class Test {


  public static void main(String args[]) {

```

```
12      //build measurement system
        EventGenerator_any_data         EG1 = new EventGenerator_any_data();
14
        EventCorrelator_data_transfer   EC1 = new EventCorrelator_data_transfer();
16
18      PostProcessor_data_transfer_rate PP1 = new PostProcessor_data_transfer_rate();

20      //build bindings

22      EG1.bind(EC1);

24      EC1.bind(PP1);

26
        //send
28
        Primitive_socket_send P2 = new Primitive_socket_send();
30
32      P2.src = new byte[2];
          P2.src[0] = 100;
34
36      EG1.receive(P2);
        P2.src = new byte[2];
38        P2.src[0] = 101;

40      EG1.receive(P2);

42      //again using the builder

44
        SpecialTap myTap = new SpecialTap();
46
        Builder_data_transfer_rate myBuilder = new Builder_data_transfer_rate(myTap);
48
50      myBuilder.buildInstances();

52      myTap.doIt();

54      }
    }
```

Symbolverzeichnis

Aquila	Adaptive Resource Control for QoS Using an IP-based Layered Architecture
ASP	Application Service Provisioning
CAD	Computer Aided Design
CASAM	Computer Aided Specification And Measurement
CASE	Computer Aided Software Engineering
CDL	Contract Description Language
constraints	Bediungungen
CORBA	Common Object Request Broker Architecture
CSM	Customer Service Management
Customer	Kunde
Framework	Rahmenwerk
HQML	Hierarchical Quality of Service Markup Language
IDL	Interface Description Language
IETF	Internet Engineering Task Force
IP	Internet Protocol
IPPM	IP Performance Metrics; generischer Ansatz zur Beschreibung von Dienstgütemessungen im Internetumfeld
ISO	International Organizaiton for Standardization
ITU-T	International Telecommunication Union
javacc	Java Compiler Compiler
MAQS	Management for Adaptive QoS-enabled Services
ODL	Object Description Language
OGSA	Open Grid Services Architecture
Provider	Dienstanbieter
QDL	Quality Description Language
QML	Quality Modeling Language

QoS	Quality of Service
QoSME	QoS Management Environment
QoSSL	Quality of Service Specification Language
QRR	Quality Runtime Representation
QuAL	Quality Assurance Language
SAP	Service-Access-Point; Dienstzugangspunkt
SDL	Structure Description Language
TINA	Telecommunications Information Networking Architecture
UML	Unified Modelling Language
User	Nutzer
VPN	Virtual Private Network
yacc	Yet Another Compiler Compiler

Index

Literaturverzeichnis

[Aage 01] AAGEDAL, J. Ø: *Quality of Service Support in Development of Distributed Systems*. Dr. scient. thesis, Department of Informatics, Faculty of Mathematics and Natural Sciences, University of Oslo, März 2001, http://www.ifi.uio.no/~janoa/papers/thesis.pdf .

[AARW+ 02] AL-ALI, RASHID J., OMER F. RANA, DAVID W. WALKER, SANJAY JHA und SHALEEZA SOHAOL: *G-QoSM: Grid Service Discovery Using QoS Properties* . In: *COMPUTING AND INFORMATICS*, 2002, http://www.wesc.ac.uk/learn/publications/pdf/cijnl.pdf .

[Aquilla] *Aquila: Adaptive Resource Control for QoS Using an IP-based Layered Architecture*. Technischer Bericht, http://www-st.inf.tu-dresden.de/aquila/ .

[AsVi 00] ASENSIO, JUAN I. und VÍCTOR A. VILLAGRÁ: *A UML Profile for QoS Management Information Specification in Distributed Object-based Applications*. In: *7th International Workshop of the HP OpenView University Association (HPOVUA 2000)*, Santorini, Greece, Juni 2000. , http://www.hpovua.org/PUBLICATIONS/PROCEEDINGS/7_HPOVUAWS/posters/poster-A-4.pdf .

[AVLdVB 01] ASENSIO, JUAN I., VÍCTOR A. VILLAGRÁ, JORGE E. LÓPEZ-DE VERGARA und JULIO J. BERROCAL: *UML Profiles for the Specification and Instrumentation of QoS Management Information in Distributed Object-based Applications*. In: *5th Fifth World Multi-Conference on Systemics, Cybernetics and Informatics (SCI 2001)*, Orlando, Florida, Juli 2001. , http://jungla.dit.upm.es/~jlopez/publicaciones/sci01jasensio.pdf . ISBN 980-07-7543-9.

[BeGe 97] BECKER, C. und K. GEIHS: *MAQS - Management for Adaptive QoS-enabled Services*. In: *Proceedings of IEEE Workshop on Middleware for Distributed Real-Time Systems and Services*, San Francisco, USA, 1997. , http://www.informatik.uni-stuttgart.de/ipvr/vs/de/people/beckercn/WMRTSS97.ps.gz .

[BeGe 99] BECKER, C. und K. GEIHS: *Generic QoS Specifications for CORBA*. In: *Proceedings of Kommunikation in verteilten Systemen (KIVS'99)*, Darmstadt, 1999. , http://www.informatik.uni-stuttgart.de/ipvr/vs/de/people/beckercn/kivs.pdf .

[ChMo 95] CHAPMNA, MARTIN und STEFANO MONTESI: *Overall Concepts and Principles of TINA*. Technischer Bericht, TINA-C, Februar 1995, http://www.tinac.com/specifications/documents/overall.pdf .

[CORBA] *Common Object Request Broker Architecture (CORBA/IIOP)*. Specification Version 3.0.2, OMG, März 2004, http://www.omg.org/cgi-bin/doc?formal/04-03-12 .

[DoTu 94] DONALDSON, A. JOHN M. und KENNETH J. TURNER: *Formal specification of QoS properties*. In: *Proceedings of Workshop on Distributed Multimedia Applications and QoS Verification*, Seiten 1–14, Montreal, Canada, Juni 1994. , http://www.cs.stir.ac.uk/~kjt/research/pdf/form-qos.pdf .

[Dreo 02] DREO RODOSEK, G.: *A Framework for IT Service Management*. Habilitation, Ludwig–Maximilians–Universität München, Juni 2002.

[Dreo 02a] DREO RODOSEK, G.: *Quality Aspects in IT Service Management*. In: FERIDUN, M., P. KROPF und G. BABIN (Herausgeber): *Proceedings of the 13th IFIP/IEEE International Workshop on Distributed Systems: Operations & Management (DSOM 2002)*, Lecture Notes in Computer Science (LNCS) 2506, Seiten 82–93, Montreal, Canada, Oktober 2002. IFIP/IEEE, Springer.

[DSpec] *D-Spec: A QoS Specification Language for Dynamic Real-time Systems*. Technischer Bericht, http://zen.ece.ohiou.edu/projects/dsdrt/download/manuals/dspec-manual.html .

[EGP+ 02] EXPOSITO, E., M. GINESTE, R. PEYRICHOU, P. SENAC und M. DIAZ: *XQoS: a quality of service specification language*. In: *Proceedings of IADIS International Conference on WWW/Internet*, Seiten 648–652, Lisbonne, Protugal, November 2002. , http://dmi.ensica.fr/IMG/pdf/final_xqos.pdf .

[EGP+ 03] EXPOSITO, E., M. GINESTE, R. PEYRICHOU, P. SENAC und M. DIAZ: *XQOS : XML-based QoS specification language*. In: *Proceedings of 9th International Conference on Multi-Media Modeling* , Taiwan, Januar 2003. , http://dmi.ensica.fr/IMG/pdf/xqos_-_xml-based_qos_specification_language.pdf .

[FKNT 02] FOSTER, I., C. KESSELMAN, J. NICK und S. TUECKE: *The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration*. Technischer Bericht, Open Grid Service Infrastructure WG, Global Grid Forum, Juni 2002, http://www.globus.org/research/papers/ogsa.pdf .

[Flor 96] FLORISSI, PATRÍCIA GOMES SOARES: *QoSME: QoS Management Environment*. PhD thesis, Columbia University, 1996, http://www.cs.columbia.edu/dcc/publications/thesis/pgsf/pgsf_thesis.ps .

[FrKo 98] FRØLUND, SVEND und JARI KOISTINEN: *QML: A Language for Quality of Service Specification*. Report HPL-98-10, Software Technology Laboratory, Hewlett-Packard Company, September 1998, http://www.hpl.hp.com/techreports/98/HPL-98-159.pdf .

[GaRo04] GARSCHHAMMER, MARKUS und HARALD ROELLE: *Requirements on Quality Specification Posed by Service Orientation*. In: *Proceedings of the 15th IFIP/IEEE International Workshop on Distributed Systems: Operations & Management (DSOM 2004)*, Seiten 1–14, 2004 2004, http://wwwmnmteam.informatik.uni-muenchen.de/php-bin/pub/show_pub.php?key=garo04 .

[GHHK 02] GARSCHHAMMER, M., R. HAUCK, H.-G. HEGERING, B. KEMPTER, I. RADISIC, H. RÖLLE und H. SCHMIDT: *A Case–Driven Methodology for Applying the MNM Service Model*. In: STADLER, R. und M. ULEMA (Herausgeber): *Proceedings of the 8th International IFIP/IEEE Network Operations and Management Symposium (NOMS 2002)*, Seiten 697–710, Florence, Italy, April 2002. IFIP/IEEE, IEEE Publishing, http://wwwmnmteam.informatik.uni-muenchen.de/php-bin/pub/show_pub.php?key=ghhk02 .

[GHJV 95] GAMMA, E., R. HELM, R. JOHNSON und J. VLISSIDES: *Design Patterns — Elements of Reusable Object–Oriented Software*. Addison-Wesley Professional Computing Series. Addison-Wesley, 1995. ISBN 0-201-63361-2.

[GHKR 01] GARSCHHAMMER, M., R. HAUCK, B. KEMPTER, I. RADISIC, H. ROELLE und H. SCHMIDT: *The MNM Service Model — Refined Views on Generic Service Management*. Journal of Communications and Networks, 3(4):297–306, Dezember 2001, http://wwwmnmteam.informatik.uni-muenchen.de/php-bin/pub/show_pub.php?key=ghkr01 .

[GJSB 00] GOSLING, J., B. JOY, G. STEELE und G. BRACHA: *The Java Language Specification*. Addison–Wesley, Reading, Mass., 2 Auflage, Juni 2000, http://java.sun.com/docs/books/jls/second_edition/html/j.title.doc.html .

[GWN 01] GU, XIAOHUI, DUANGDAO WICHADAKUL und KLARA NAHRSTEDT: *Visual QoS Programming Environment for Ubiquitous Multimedia Services*. In: *Proceedings of IEEE International Conference on Multimedia and Expo 2001 (ICME2001)*, Tokyo, Japan, August 2001. , http://cairo.cs.uiuc.edu/publications/paper-files/icme2001-xgu.pdf .

[HaRe 00] HAUCK, R. und H. REISER: *Monitoring Quality of Service across Organizational Boundaries*. In: *Trends in Distributed Systems: Towards a Universal Service Market. Proceedings of the third International IFIP/GI Working Conference, USM 2000*, September 2000, http://wwwmnmteam.informatik.uni-muenchen.de/php-bin/pub/show_pub.php?key=hare00 .

[Hauc 01] HAUCK, R.: *Architektur für die Automation der Managementinstrumentierung bausteinbasierter Anwendungen*. Dissertation, Ludwig–Maximilians–Universität München, Juli 2001, http://wwwmnmteam.informatik.uni-muenchen.de/php-bin/pub/show_pub.php?key=hauc01 .

[HHKT 00] HOOVER, C., J. HANSEN, P. KOOPMAN und S. TAMBOLI: *Quartz: A QoS Architecture for Open Systems* . In: *Proceedings of 20th International Conference on Distributed Computing Systems (ICDCS 2000)*, Taipei, Taiwan, April 2000. , http://www.inf.ufsc.br/~frank/papers/ICDCS2000.pdf .

[HHKT 01] HOOVER, C., J. HANSEN, P. KOOPMAN und S. TAMBOLI: *The Amaranth Framework: policy-based quality of service management for high-assurance computing*. In: *International Journal of Reliability, Quality, and Safety Engineering*, Seiten 1–28, 2001, http://www.ece.cmu.edu/~koopman/am/ijrqse01/ijrqse01.pdf .

[IETF] *IETF: The Internet Engineering Task Force*. Technischer Bericht, http://www.ietf.org/ .

[ISO 14977] *Information Technology – Syntactic metalanguage – Extended BNF*. IS 14977, International Organization for Standardization and International Electrotechnical Committee, 1996.

[ISO 7498] *Information Processing Systems – Open Systems Interconnection – Basic Reference Model*. IS 7498, ISO/IEC, 1984.

[ITU X.920] *Open Distributed Processing – Interface Definition Language*. Draft Recommendation X.920, ITU, November 1997.

[JavaCC] *Java Compiler Compiler (JavaCC)*. Technischer Bericht, java.net The Source for Java Technology Colloboration, https://javacc.dev.java.net/ .

[JiNa 04] JIN, JINGWEN und KLARA NAHRSTEDT: *QoS Specification Languages for Distributed Multimedia Applications: A Survey and Taxonomy*. In: *IEEE Multimedia Magazine*, Band 11, Seiten 74 –87. IFIP/IEEE, IEEE Publishing, Juli 2004, http://cairo.cs.uiuc.edu/publications/paperfiles/ieee_multimedia_jin.pdf .

[Lang 01] LANGER, M.: *Konzeption und Anwendung einer Customer Service Management Architektur*. Dissertation, Technische Universität München, März 2001, http://tumb1.biblio.tu-muenchen.de/publ/diss/in/2001/langer.html .

[LBS+ 98] LOYALL, J.P., D.E. BAKKEN, R.E. SCHANTZ, J.A. ZINKY, D.A. KARR, R. VANEGAS und K.R. ANDERSON: *QoS Aspect Languages and Their Runtime Integration*. In: *Lecture Notes in Computer Science, Vol. 1511, Proceedings of the Fourth Workshop on Languages, Compilers, and Run-time Systems for Scalable Computers (LCR98)*, Pittsburgh, Pennsylvania, Mai 1998. Springer-Verlag, http://www.dist-systems.bbn.com/papers/1998/LCR/lcr.ps .

[Nerb 01] NERB, M.: *Customer Service Management als Basis für interorganisationales Dienstmanagement*. Dissertation, Technische Universität München, März 2001, http://tumb1.biblio.tu-muenchen.de/publ/diss/in/2001/nerb.html .

[ODL96] *TINA Object Definition Language Manual*. Technischer Bericht, TINA-C, Juli 1996, http://www.tinac.com/specifications/documents/odl96_public.pdf .

[ODP-QOS] *Working document in QoS in ODP*. Technischer Bericht, ISO/IEC, 1997, ftp://ftp.dstc.edu.au/pub/arch/RM-ODP/Canberra_Output/7N1192.ps .

[OWS+ 03] OODAN, ANTONY, KEITH WARD, CATHERINE SAVOLAINE, MAHMOUD DANMESHMAND und PETER HOATH: *Telecommunications Quality of Service Management — from legacy to emerging services*. The Institution of Electrical engineers, ISBN 0–85296–424–2, 2003.

[RFC 2330a] PAXSON, V., G. ALMES, J. MAHDAVI und M. MATHIS: *RFC 2330: Framework for IP Performance Metrics*. RFC, IETF, Mai 1998, ftp://ftp.isi.edu/in-notes/rfc2330.txt .

[RFC 2330b] PAXSON, V., G. ALMES, J. MAHDAVI und M. MATHIS: *RFC2330: Framework for IP Performance Metrics*, 1998, ftp://ftp.internic.net/rfc/rfc2330.txt .

[RFC 2678] MAHDAVI, J. und V. PAXSON: *RFC 2678: IPPM Metrics for Measuring Connectivity*. RFC, IETF, September 1999, ftp://ftp.isi.edu/in-notes/rfc2678.txt .

[RFC 2679] ALMES, G., S. KALIDINDI und M. ZEKAUSKAS: *RFC 2679: A One-way Delay Metric for IPPM*. RFC, IETF, September 1999, ftp://ftp.isi.edu/in-notes/rfc2679.txt .

[RFC 2680] ALMES, G., S. KALIDINDI und M. ZEKAUSKAS: *RFC 2680: A One-way Packet Loss Metric for IPPM*. RFC, IETF, September 1999, ftp://ftp.isi.edu/in-notes/rfc2680.txt .

[RFC 2681] ALMES, G., S. KALIDINDI und M. ZEKAUSKAS: *RFC 2681: A Round-trip Delay Metric for IPPM*. RFC, IETF, September 1999, ftp://ftp.isi.edu/in-notes/rfc2681.txt .

[RFC 3148] MATHIS, M. und M. ALLMAN: *RFC3148: A Framework for Defining Empirical Bulk Transfer Capacity Metrics*, 2001, ftp://ftp.internic.net/rfc/rfc3148.txt .

[RFC 791] POSTEL, J.: *RFC 791: Internet Protocol*. RFC, IETF, September 1981, ftp://ftp.isi.edu/in-notes/rfc791.txt .

[RJB 98] RUMBAUGH, J., I. JACOBSON und G. BOOCH: *Unified Modeling Language — Reference Manual*. Addison–Wesley, 1998.

[RöZs 03] RÖTTGER, SIMONE und STEFFEN ZSCHALER: *CQML+: Enhancements to CQML*. In: *Proceedings of 1st International Workshop on Quality of Service in Component-Based Software Engineering*, Seiten 43–56, Toulouse, France, Juni 2003. Cépaduès-Éditions, http://www-st.inf.tu-dresden.de/comquad/qoscbse03-language.pdf .

[SWM 95] STAEHLI, RICHARD, JONATHAN WALPOLE und DAVID MAIER: *A quality-of-service specification for multimedia presentations*. In: *Multimedia Systems archive – Special issue on multimedia database systems*. Springer-Verlag New York, ISSN:0942-4962, 1995, http://www.inf.tu-dresden.de/ST2/ST/papers/SEA2002AnneThomas.pdf .

[TGN+ 03] TIAN, M., A. GRAMM, T. NAUMOWICZ, H. RITTER und J. SCHILLER: *A Concept for QoS Integration in Web Services*. In: *Proceedings of 1st Web Services Quality Workshop (WQW 2003), in conjunction with 4th International Conference on Web Information Systems Engineering (WISE 2003)*, Dezember 2003, http://page.mi.fu-berlin.de/~tian/pdf/tian_wqw2003.pdf .

[Thom 02] THOMAS, ANNE: *Applying legacy applications with QoS: a description syntax at application, end-user and network level*. In: *Proceedings of Software Engineering and Applications (SEA 2002)*, November 2002, http://www.inf.tu-dresden.de/ST2/ST/papers/SEA2002AnneThomas.pdf .

[WSRB 98] WELCH, LONNIE R., BEHROOZ A. SHIRAZI, BINOY RAVINDRAN und CARL BRUGGEMAN: *DeSiDeRaTa: QoS Management Technology for Dynamic, Scaleable, Dependable, Real-Time System*. Technischer Bericht, 1998, http://www.ee.vt.edu/~binoy/papers/ifac98.pdf .

[X.641] *OSI Networking and System Aspects - Quality of Service*. Recommendation X.641, ITU-T, Dezember 1997.

[X.901] *Information Technology – Open Distributed Processing – Reference Model – Part 1: Overview and Guide to Use*. Recommendation X.901, ITU-T, August 1997. (Technically identical with ISO/IEC DIS 10746-1).

[yacc] *Yacc: Yet Another Compiler-Compiler*. Technischer Bericht, http://dinosaur.compilertools.net/yacc/ .

[ZBS 97] ZINKY, J., D. BAKKEN und R. SCHANTZ: *Architectural Support for Quality of Service for CORBA Objects*. In: *Theory and Practice of Object Systems*, Januar 1997, http://www.dist-systems.bbn.com/papers/1997/TAPOS/tapos.ps .