Kontextsensitive Qualitätsplanung von Softwaremodellen

Fakultät für Elektrotechnik, Informatik und Mathematik der
Universität Paderborn

Genehmigte Dissertation
zur Erlangung des akademischen Grades
Doktor der Naturwissenschaften
(Dr. rer. nat.)

von
Dipl.-Wirt.-Inf. Hendrik Voigt
aus
Herdecke/Ruhr

Datum der Einreichung: 22. September 2009
Tag der mündlichen Prüfung: 11. November 2009

Abstract

The Model Quality Plan (MQP) approach provided by us allows for the systematic and efficient development of quality plans that serve as a basis for the assessment of software models. MQP emphasizes the *context* of a software model as a major factor of influence for the whole quality planning activity. In order to adjust a quality plan to project specific requirements, quality goals are derived from a characterization of that context. We achieve a differentiated description of quality goals by introducing structured goals and questions in combination with a defined quality model. Afterwards, metrics and indicators are identified for checking the fulfillment of the quality goals. The result of our approach consists of a context sensitive quality plan for software models.

Conceptually, we combine a metamodel for formulating relevant contents, a process that serves as a guideline for defining quality plans, and a rule concept for packaging and reusing experience into an integrated framework.

We show its feasibility by three case studies that include a quality plan for analysis models, design models, and test models, respectively. For that, we provide tool support for the definition and application of quality plans.

Quality of software models, analytical quality assurance, MQP, quality plan, context, quality model, indicator, metric, measure

Zusammenfassung

Der von uns erstellte Ansatz der Modell-Qualitäts-Pläne (MQP) ermöglicht die systematische und effiziente Entwicklung von Qualitätsplänen, die als Grundlage für die Bewertung von Softwaremodellen dienen. Dabei stellen wir den *Kontext* eines Softwaremodells als bedeutenden Einflussfaktor für die Qualitätsplanung heraus. Um einen Qualitätsplan auf die projektspezifischen Anforderungen abzustimmen, werden ausgehend von einer Charakterisierung des Kontextes Qualitätsziele hergeleitet und durch die Kombination von strukturierten Zielen und Fragen mit einem definierten Qualitätsmodell differenziert beschrieben. Anschließend werden Metriken und Indikatoren zur Prüfung der Qualitätsziele identifiziert. Das Ergebnis dieses Ansatzes besteht in einem kontextsensitiven Qualitätsplan für Softwaremodelle.

Auf konzeptioneller Ebene kombinieren wir ein Metamodell zur Formulierung relevanter Inhalte, einen Prozess als Leitfaden für die Qualitätsplanung sowie ein Regelkonzept zur Sicherung und Wiederverwendung von Erfahrungswissen zu einem integrierten Rahmenwerk.

Die Praxistauglichkeit des MQP-Ansatzes demonstrieren wir anhand von drei Fallstudien, in denen jeweils ein Qualitätsplan für Analysemodelle, Entwurfsmodelle und Testmodelle entwickelt wurde. Für die Definition und Anwendung von Qualitätsplänen wurden die Konzepte prototypisch umgesetzt.

Qualität von Softwaremodellen, analytische Qualitätssicherung, MQP, Qualitätsplan, Kontext, Qualitätsmodell, Indikator, Metrik, Kennzahl

Inhaltsverzeichnis

Kapitel 1

Einleitung

1.1 Motivation

Der Steuerungsanteil von Software an Produkten aller Branchen steigt zunehmend an. Neben den typischen Beispielen wie Computern und Mobiltelefonen unterstützt Software unter anderem Zugangssysteme von Gebäuden, die Elektronik von PKW's, Gepäckabfertigungssysteme am Flughafen und vieles mehr. "Software hat sich zu einem zentralen *Werkstoff* des *Informationszeitalters* entwickelt. Innovative Produkte und Dienstleistungen sind ohne (...) Software nicht mehr denkbar" [32].

Gleichzeitig steigen die Herausforderungen an die Softwareentwicklung. Einerseits werden von den Software-Entwicklern immer kürzere Entwicklungszeiten gefordert und eine strikte Einhaltung der Budgets für die Entwicklungsprojekte verlangt, andererseits steigen zunehmend die Qualitätsanforderungen an das Softwareprodukt. Dieser Konflikt verschärft sich zudem stetig durch die wachsende Komplexität und Kompliziertheit von Software.

Die modellbasierte Softwareentwicklung stellt eine vielversprechende Alternative zur herkömmlichen Softwareentwicklung dar, um den wirtschaftlichen und qualitativen Herausforderungen zu begegnen. Sie gewinnt durch den hohen Reifegrad und die weite Verbreitung der Unified Modeling Language (UML), praxiserprobter Modellierungsmethoden und unterstützender Werkzeuge zunehmend an Bedeutung. Die Gründe für die Anwendung von modellbasierten Softwareentwicklungsprozessen in Unternehmen sind vielfältig [119]. Eine Stärke von Softwaremodellen liegt in der Abstraktion. Sie vereinfachen die Kommunikation über komplexe Systeme und Probleme durch Reduktion und ermöglichen es einem Projektteam, durch schrittwei-

se Verfeinerungen Lösungen systematisch zu entwickeln. Eine Aufwandsreduzierung erhofft man sich unter anderem durch Code Generatoren, die stupide und repetitive Programmierarbeit automatisieren, eine höhere Entwicklungsgeschwindigkeit erzielen und damit Kosten sparen. Zudem erreichen Unternehmen durch entwickelte Softwaremodelle weitere Synergieeffekte während der Weiterentwicklung und der langjährigen Wartung von Softwaresystemen.

In der modellbasierten Softwareentwicklung werden Softwaremodelle für die Bearbeitung verschiedener Aufgaben verwendet. Softwaremodelle werden beispielsweise zur Analyse der Anforderungsspezifikation, zur Konzeption der Architektur oder zum Feinentwurf des zu implementierenden Softwaresystems eingesetzt. Infolgedessen bilden Softwaremodelle das Bindeglied zwischen den Wünschen des Kunden und dem fertigen Softwareprodukt und stellen zentrale Entwicklungsartefakte dar.

Qualitätsmängel in Softwaremodellen sollten möglichst früh erkannt werden, damit sich notwendige Korrekturen nur auf wenige Entwicklungsphasen und die entsprechenden Entwicklungsartefakte beziehen. Basiert z.B. die Implementierung auf einem unvollständigen Softwaremodell, das zudem noch ein ungeeignetes Design beschreibt, resultiert dies ohne aufwändige Korrekturmaßnahmen zwangsläufig in *schlechter* Software. Mangelhafte Softwaremodelle stellen somit das Vorgehen der modellbasierten Softwareentwicklung grundsätzlich in Frage.

Infolge der herausragenden Bedeutung von Softwaremodellen für den Erfolg der modellbasierten Softwareentwicklung rückt nun die Qualitätsprüfung der erstellten Softwaremodelle in den Fokus. Während sich in der Vergangenheit Qualitätsprüfungen hauptsächlich auf Software konzentrierten, müssen jetzt die Bemühungen im Bereich Qualität von Softwaremodellen zunehmend intensiviert werden. In modellbasierten Softwareentwicklungsprojekten muss dafür Sorge getragen werden, dass Softwaremodelle *richtig* sind. Aus organisatorischer Sicht bedeutet dies, dass orthogonal zum Konstruktionsprozess eine *analytische Qualitätssicherung* etabliert wird, in der Softwaremodelle überprüft und bewertet werden. Die Rolle des *Qualitätsmanagers* trägt für diese Qualitätssicherung die Verantwortung.

Der grobe Ablauf der analytischen Qualitätssicherung ist domänenübergreifend gleich und besteht darin, im Rahmen einer *Qualitätsplanung* Qualitätsziele festzulegen, zu bestimmen, wie sie geprüft und welche Prüfergebnisse erreicht werden sollen. Anschließend wird die *Qualitätsprüfung* zur Feststellung des Qualitäts-Ist-Zustandes durchgeführt, die Prüfergebnisse durch Projektverantwortliche *interpre-*

tiert und darauf basierend wird eine *Entscheidung über das weitere Vorgehen getroffen*. Eine strukturierte Abwicklung einer so umfangreichen Aufgabe wie der analytischen Qualitätssicherung von Softwaremodellen muss planvoll erfolgen. Folglich sollte ein dokumentierter Qualitätsplan die Grundlage für alle weiteren Aktivitäten bilden. Allerdings treten bei der Erstellung von Qualitätsplänen für die Prüfung von Softwaremodellen zwei essentielle Probleme auf.

1.2 Problemstellung

Softwaremodelle unterscheiden sich nicht nur inhaltlich, sondern auch in ihrer Einbettung in den modellbasierten Softwareentwicklungsprozess. Beispielsweise beziehen sich Softwaremodelle auf unterschiedliche Tätigkeiten im Prozess (Problemabgrenzung in der Analyse oder Lösungsansatz im Entwurf), stellen verschiedene Sichten bereit (Klassendiagramm, Zustandsdiagramm oder Sequenzdiagramm), weisen spezielle Entwicklungsabhängigkeiten zu anderen Entwicklungsartefakten auf (plattformabhängiges Modell als Verfeinerung eines plattformunabhängigen Modells in der Model Driven Architecture oder Modell des Problembereichs als Ergänzung zum Pflichtenheft) u.v.m.. Dieses fachliche Umfeld, in dem sich ein zu prüfendes Softwaremodell befindet, muss vom Qualitätsmanager berücksichtigt werden. Beispielsweise ist die Beurteilung der Kapselung von Attributen in einem Analyseklassendiagramm nicht sinnvoll, weil die Sichtbarkeit der Attribute während der Analysephase nicht spezifiziert wird.

Zudem muss der Qualitätsmanager den Qualitätsplan auf die Projektbeteiligten abstimmen. Er muss gewährleisten, dass die Qualitätsprüfung auch die Informationen liefert, die die Projektverantwortlichen für ihre weitreichenden Entscheidungen benötigen. Dafür muss der Qualitätsmanager unter Berücksichtigung des jeweiligen Projektteams, des zu prüfenden Softwaremodells und dessen Einbettung in den Softwareentwicklungsprozess für die *richtigen* Qualitätsziele die *richtigen* Qualitätsprüfungen auswählen bzw. entwickeln.

Deshalb kann ein Qualitätsmanager typischerweise nicht einfach einen bestehenden Qualitätsplan als Ganzes wiederverwenden. Stattdessen muss er den Qualitätsplan auf die Spezifika im gegebenen Projekt abstimmen. Diese Ausführungen führen zu unserer ersten Problemstellung, mit der ein Qualitätsmanager konfrontiert wird:

Problem 1

Welche Qualitätsziele sollen wie überprüft werden?

Bei der Lösung des obigen Problems sind eine Reihe von in der Praxis gegebenen Rahmenbedingungen zu bedenken, die restriktiv auf die Intensität der Qualitätsplanung wirken. Unseren Erfahrungen zu Folge ist die Entwicklung eines auf ein Projekt hin zugeschnittenen Qualitätsplans eine anspruchsvolle Tätigkeit, die mit einem hohen kreativen Aufwand verbunden ist und menschliche Ressourcen bindet. Infolgedessen ist die Qualitätsplanung als recht aufwändig zu beurteilen.

Der Mehrwert der analytischen Qualitätssicherung von Softwaremodellen liegt darin, dass sich Projektrisiken kontrollieren lassen. Die Beherrschung von Risiken soll langfristig zu einer kostengünstigeren Softwareentwicklung und zu einem erfolgreicheren Softwareeinsatz führen. Eine *gute* Qualitätssicherung reduziert jedoch nur die Risiken für die Entstehung von zusätzlichen Kosten. Eine *schlechte* oder *fehlende* Qualitätssicherung führt nicht zwangsläufig zu höheren Kosten, denn Fehler können, müssen aber nicht, weitere Kosten verursachen. Deshalb ist eine Verrechnung von zusätzlichen Kosten für die Qualitätssicherung einerseits und möglichen Projektkosten ohne eine angemessene Qualitätssicherung andererseits in der Praxis sehr schwierig.

Die Qualitätssicherung wird häufig als notwendig, aber unproduktiv wahrgenommen. Aus diesem Grund wird die Qualitätssicherung immer sehr beschränkten Ressourcen unterliegen und sollte möglichst optimiert gestaltet werden. Hier sind insbesondere Personalkosten zu beachten, die häufig den entscheidenden Kostenfaktor darstellen.

Darüber hinaus ist auch der zeitliche Faktor beschränkt. Die Blockierung des Entwicklungsfortschritts durch Qualitätssicherungsaktivitäten wie Qualitätsplanung und Qualitätsprüfung sollte sich auf ein Minimum beschränken. Diesen Ressourcenengpässen muss aktiv entgegengewirkt werden.

Problem 2

Wie kann ein guter Qualitätsplan mit dem Einsatz beschränkter Ressourcen erstellt werden?

1.3 Zielsetzung

Die Beantwortung dieser unter Problem 1 und 2 formulierten Fragestellungen möchten wir für die Qualitätsplanung im Rahmen der analytischen Qualitätssicherung von Softwaremodellen angehen.

Für eine Lösung der ersten Problemstellung *Welche Qualitätsziele wie geprüft werden sollen?* benötigen wir einen systematischen, für Softwaremodelle generischen Ansatz, den ein Qualitätsmanager befolgen kann und dadurch in der Qualitätsplanung unterstützt wird. Der systematische Ansatz soll den Qualitätsmanager anleiten, die wesentlichen Aspekte eines Softwaremodells, die Informationsbedürfnisse der Projektbeteiligten und deren Fachwissen für die Qualitätsplanung zu nutzen. Erst durch die aktive Einbindung des Projektteams kann gewährleistet werden, dass die Qualitätsprüfung auch die Informationen liefert, die die Entscheidungsträger für ihre weitreichenden Projektentscheidungen wirklich benötigen. Die Betrachtung des fachlichen Umfelds eines Softwaremodells kann vom Qualitätsmanager dazu genutzt werden, unabhängig von den Projektbeteiligten Qualitätsziele abzuleiten.

Der Problematik der *beschränkten Ressourcen* begegnen wir mit einem in der Softwaretechnik weit verbreiteten Konzept: *Wiederverwendung*. Wenn ein Qualitätsmanager die Möglichkeit hat, auf Basis bereits erfolgreich durchgeführter Qualitätssicherungen oder in der Literatur veröffentlichter Ergebnisse einzelne Teile einer Qualitätsplanung oder sogar ganze Qualitätspläne für die Wiederverwendung in zukünftigen Projekten aufzubereiten, dann könnte er Qualitätspläne auf Basis vorgefertigter Bausteine zusammensetzen und müsste den Qualitätsplan nur noch durch einen begrenzten, neuen Anteil vervollständigen. Dadurch würde sich auch die notwendige Kooperation mit den Projektbeteiligten reduzieren. Zudem ließe sich Erfahrungswissen sichern und die Qualität eines Qualitätsplans selbst erhöhen.

Zielsetzung

Wir bezwecken die Entwicklung eines *systematischen Ansatzes* für die Qualitätsplanung von Softwaremodellen, dessen Ergebnis ein dokumentierter Qualitätsplan ist.
Dieser *Qualitätsplan* soll alle wichtigen Informationen enthalten, damit die Qualitätsprüfung durchgeführt, die Prüfergebnisse durch Projektverantwortliche interpretiert und darauf basierend eine Entscheidung getroffen werden kann.
Der Ansatz zur Erstellung von Qualitätsplänen soll auf die Rolle des *Qualitätsmanagers* abgestimmt sein und helfen, sowohl das Projektteam konstruktiv einzubinden als auch die Besonderheiten des zu prüfenden Softwaremodells zu berücksichtigen.
Zusätzlich möchten wir diesen Ansatz um ein Konzept zur *Wiederverwendung von Erfahrungswissen* anreichern, um Ressourcen zu sparen und die Qualität eines Qualitätsplans selbst zu erhöhen.

1.4 Aufbau der Arbeit

Zuerst unterstreichen wir in Kapitel 2 die Bedeutung der analytischen Qualitätssicherung von Softwaremodellen und beschreiben ihren Ablauf sowie beteiligte Rollen. Anschließend detaillieren wir unsere in Abschnitt 1.2 formulierte Problemstellung, indem wir eine Reihe von Anforderungen an die Qualitätsplanung thematisieren.

In Kapitel 3 stellen wir die Besonderheiten von Softwaremodellen im Kontext der modellbasierten Softwareentwicklung heraus. Darauf basierend definieren wir die für uns zentralen Begriffe *Softwaremodell* und *Kontextsensitivität*.

Relevante Arbeiten führen wir im vierten Kapitel ein und bewerten ihren Nutzen für die Qualitätsplanung von Softwaremodellen anhand unserer zuvor aufgestellten Anforderungen.

In Kapitel 5 stellen wir unseren Lösungsansatz der *Modell-Qualitäts-Pläne (MQP)* vor. Wir führen den MQP-Ansatz ein, indem wir die grundlegenden Bausteine kurz zusammenfassen und ein erstes einfaches Beispiel erstellen. Anschließend erläutern wir ausführlich jeden einzelnen Schritt unseres Ansatzes zusammen mit einer dazugehörigen Strukturbeschreibung und ergänzen ein Regelkonzept für MQPs zur Sicherung und Wiederverwendung von Erfahrungswissen.

In Kapitel 6 stellen wir drei Fallstudien vor, in denen jeweils ein Qualitätsplan für Analysemodelle, Entwurfsmodelle und Testmodelle enthalten ist, und demonstrieren dadurch die Praktikabilität des MQP-Ansatzes.

Im Anschluss fassen wir unsere Ergebnisse in Kapitel 7 zusammen. Zudem ziehen wir ein Fazit der Arbeit, indem wir den MQP-Ansatz anhand unserer Anforderungen diskutieren und mit bestehenden Ansätzen vergleichen. Anschließend geben wir einen Ausblick auf sinnvolle konzeptionelle Erweiterungen des hier vorgestellten MQP-Ansatzes sowie methodische Anschlussarbeiten, bevor wir die Arbeit mit einem Schlusswort abschließen.

Kapitel 2

Anforderungen an die Qualitätsplanung von Softwaremodellen

Die Qualitätssicherung in Softwareprojekten ist von herausragender Bedeutung, weil die Qualität der produzierten Software zunehmend zum entscheidenden Faktor für die Wertschöpfung beim Kunden wird. Der Kunde akzeptiert keine minderwertige Software [159]. Die Qualitätssicherung in der Softwareentwicklung besteht in der Regel aus einer Kombination von konstruktiv vorausschauenden, analytisch prüfenden und organisatorischen Maßnahmen.

Die *organisatorischen Maßnahmen* bilden die Grundlage für die anderen Maßnahmen. Es wird ein Qualitätssicherungsprozess etabliert, der festlegt, welche konstruktiven Maßnahmen und analytischen Prüftechniken wann von wem und womit durchzuführen sind.

Konstruktive Maßnahmen sollen a priori sicherstellen, dass ein Produkt eine ausreichend hohe Qualität hat. Die Qualität wird also quasi mit *eingebaut* [145]. Hierunter fallen alle Maßnahmen, die den Softwareentwicklern die Möglichkeit eröffnen, Fehler frühzeitig zu erkennen oder gar zu vermeiden wie z.B. Software-Richtlinien und vertragsbasierte Programmierung [81].

"Kein konstruktives Verfahren kann garantieren, dass fehlerfreie Produkte entstehen. Aus diesem Grund ist die Qualität mit analytischen Mitteln zu prüfen" [107]. *Analytische Maßnahmen* dienen zur Aufdeckung von Fehlern, die sich trotz der organisatorischen und konstruktiven Maßnahmen im Produkt befinden. Sie greifen im

Gegensatz zu den anderen Maßnahmen erst, wenn das Problem schon besteht. Zu den analytischen Maßnahmen gehören z. B. Messungen, Reviews und Testtechniken.

Unserer Auffassung zufolge ist die analytische Qualitätssicherung von Softwaremodellen für eine zufriedenstellende Softwarequalität notwendig. Dabei stellt sich die Qualitätsplanung als vorbereitende Aktivität für die analytische Qualitätssicherung von Softwaremodellen als eine wichtige und anspruchsvolle Aufgabe dar. Zuerst unterstreichen wir die Bedeutung der Qualität von Softwaremodellen für die Qualität von Softwareprodukten in Abschnitt 2.1. Anschließend beschreiben wir in Abschnitt 2.2 den Ablauf der analytischen Qualitätssicherung von Softwaremodellen sowie beteiligte Rollen. Basierend auf diesem organisatorischen Umfeld besprechen wir in den Abschnitten 2.3, 2.4 bzw. 2.5 eine Reihe von Anforderungen. Die herausgestellten Anforderungen verdeutlichen die speziellen Herausforderungen, die sich im Rahmen der Qualitätsplanung von Softwaremodellen stellen. Die Anforderungen fassen wir am Ende dieses Kapitels 2 kompakt in Form einer Anforderungsmatrix zusammen. Ausgehend von diesen Anforderungen analysieren wir später im Kapitel 4 verwandte Arbeiten und bewerten in Abschnitt 7.2 unseren eigenen Lösungsansatz.

2.1 Bedeutung der Modellqualität für die Softwarequalität

Softwaremodelle und Software hängen methodisch und fachlich zusammen. Es ist unbestritten, dass infolgedessen ein qualitativer Zusammenhang existiert (vgl. hierzu [107, 160, 145, 102]).

Sind Fehler in einem Entwicklungsartefakt, führen diese Fehler ggf. zu Fehlerkosten. Je später ein Fehler entdeckt wird, desto größer ist der Korrekturaufwand für die vorangegangenen Entwicklungsartefakte. Ein unentdeckter Fehler induziert in anschließenden Entwicklungsphasen weitere Folgefehler (siehe Abb. 2.1). Fehlerkosten steigen dann sehr stark über die Entwicklungsphasen an. Deshalb ist ein frühzeitiges Finden von Fehlern möglichst direkt nach ihrer Entstehung von Bedeutung.

Die Richtwerte für den Anstieg von Fehlerkosten über Entwicklungsphasen sind in der Literatur nicht einheitlich und reichen von Verdoppelung pro Phase in [54] hin zu exponentiellem Anstieg in [107]. Empirische Untersuchungen zum Anstieg von Fehlerkosten finden sich in [149]. In der Kernaussage *Fehler möglichst früh beheben*

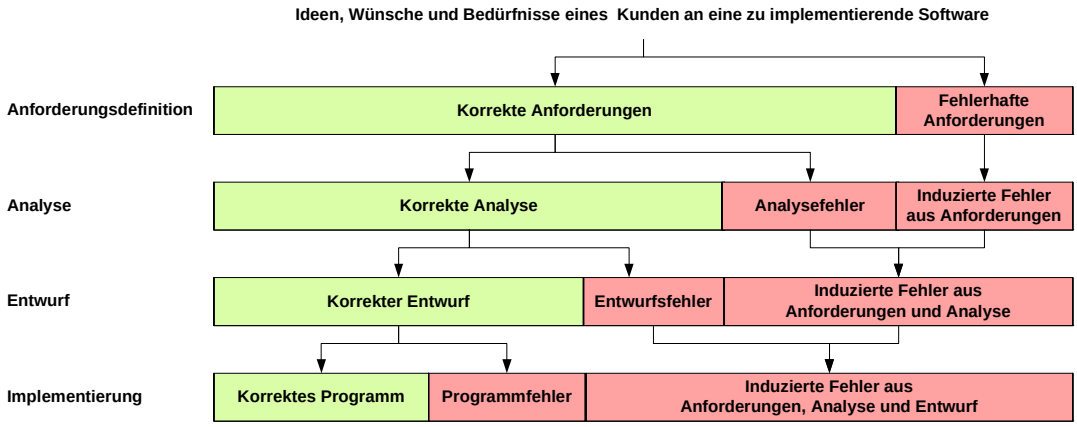


Abbildung 2.1: Auswirkung unentdeckter Fehler

sind sich dagegen alle Autoren einig.

Briand et al. weisen in [27] die Korrespondenz von einzelnen Qualitätsattributen eines Entwurfsmodells auf die entsprechende Implementierung nach. Die Evaluationen von Lange in [104] zeigen, dass Fehler in Softwaremodellen während der Implementierung erstens häufig nicht aufgedeckt werden und zweitens zu Fehlinterpretationen führen.

Zudem fordern diverse prozessorientierte Standards eine frühe und kontinuierliche Qualitätssicherung von Zwischenprodukten und bestätigen die Bedeutung der Qualität von Softwaremodellen (vgl. hierzu auch Abschnitt 4.1.1).

Zusammenfassung

Die Qualitätsbewertung von Softwaremodellen ist kein Selbstzweck. Für die Erstellung guter Software sind gute Softwaremodelle entscheidend. Erst durch eine Qualitätsprüfung von Softwaremodellen lässt sich die Softwarequalität frühzeitig sicherstellen und Fehler rechtzeitig vermeiden. Dieses Vorgehen ist zudem wirtschaftlich, denn ein frühzeitiges Finden von Fehlern direkt nach ihrer Entstehung minimiert die Fehlerkosten.

2.2 Ablauf der analytischen Qualitätssicherung

Spätestens am Ende jeder Phase, zu festgelegten Meilensteinen oder auch zu sogenannten Quality Gates [151, 41] soll die Qualität des entstandenen Softwaremodells beurteilt werden können und im Falle mangelhafter Qualität solange nicht für die darauffolgende Phase freigegeben werden, bis die definierte, ausreichende Qualität erreicht ist. Für inkrementelle oder spiralförmige Entwicklungsprozesse könnte diese Sichtweise der Phasen zu restriktiv wirken. Deshalb unterscheiden wir auch beliebig feingranulare Phasen und können auf diese Weise unsere Sichtweise für jedes beliebige Vorgehensmodell beibehalten.

Das Ziel der analytischen Qualitätssicherung besteht nun darin, den Entscheidungsprozess zur *Freigabe* oder *Verbesserung* des Softwaremodells durch eine geeignete Planung vorzubereiten. Die planerische Aktivität bezeichnen wir im Folgenden als *Qualitätsplanung* und das resultierende Ergebnis als *Qualitätsplan*. Ein Qualitätsplan muss dabei helfen, zwei zentrale Fragestellungen zu beantworten:

1. *Qualitätsziele:* Welche Qualitätsziele gelten für das Softwaremodell?
2. *Qualitätsprüfung:* Wie lässt sich der Erfüllungsgrad der festgelegten Qualitätsziele bestimmen?

Die Rolle *Qualitätsmanager* ist für die erfolgreiche Erstellung des Qualitätsplans verantwortlich. Ein Qualitätsmanager unterstützt das Projektteam dabei, die Weiterverwendung von Softwaremodellen mit mangelhafter Qualität in darauf basierenden Entwicklungsphasen zu vermeiden und auf diese Weise Folgefehler und Folgekosten zu reduzieren. Der Tätigkeitsbereich des Qualitätsmanagers muss dabei als beratende Dienstleistung für das Projektteam verstanden werden, denn die Qualitätsmanager-Rolle selbst ist weder mit fachlichen noch disziplinarischen Rechten gegenüber den Projektmitgliedern ausgestattet. Ein Qualitätsmanager bereitet wichtige Entscheidungen vor. Die Entscheidungsgewalt obliegt jedoch den Projektverantwortlichen wie z.B. dem Projektmanager.

Abbildung 2.2 skizziert den groben Ablauf der analytischen Qualitätssicherung. Die folgenden Ausführungen beziehen sich auf die nummerierten Aktivitäten in der Abbildung.

Die analytische Qualitätssicherung bezieht sich in unserem Fall auf das zu prüfende Produkt *Softwaremodell* als Ergebnis der konstruktiven Tätigkeit *Modellierung* (siehe ⓪).

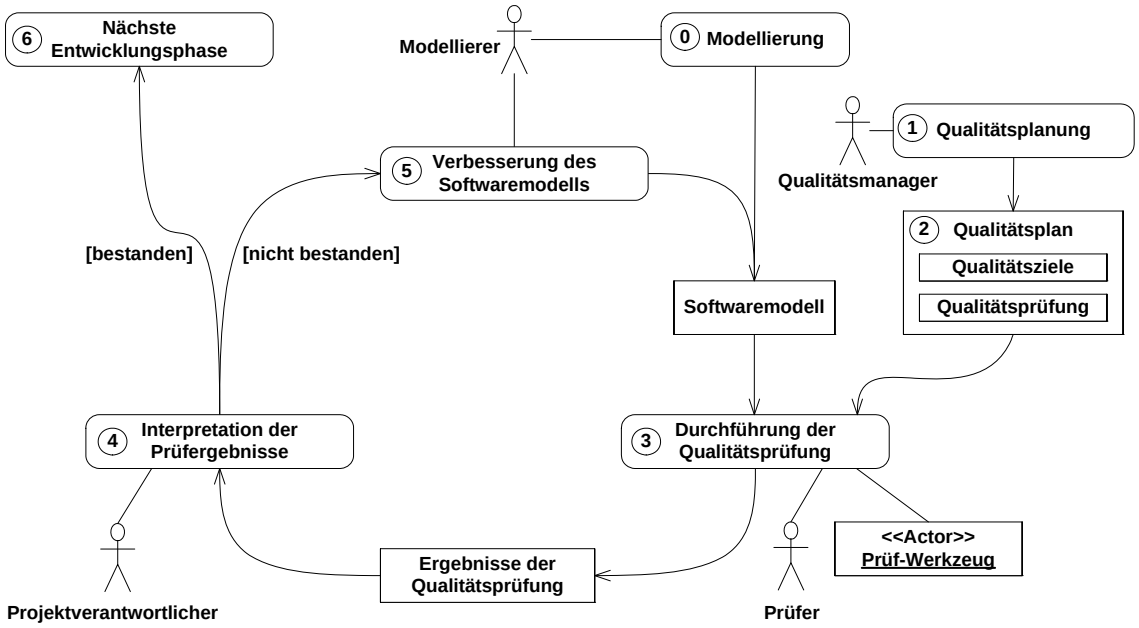


Abbildung 2.2: Grober Ablauf der analytischen Qualitätssicherung

Unabhängig von der Modellierung erstellt ein *Qualitätsmanager* während der *Qualitätsplanung* einen *Qualitätsplan*. Der Qualitätsplan muss die *Qualitätsziele* für das entsprechende Softwaremodell beschreiben und *Qualitätsprüfungen* dokumentieren, auf deren Grundlage der Erfüllungsgrad der Qualitätsziele festgestellt werden kann (siehe ① und ②).

Anschließend erfolgt die *Durchführung der Qualitätsprüfung*, um das vorliegende Softwaremodell auf die Qualitätsziele hin zu überprüfen. Die Durchführung der Qualitätsprüfung kann durch einen menschlichen Prüfer oder durch ein Prüf-Werkzeug vorgenommen werden (siehe ③).

Die *Prüfergebnisse* als Resultat der Qualitätsprüfung werden anschließend durch Projektverantwortliche *interpretiert* (siehe ④). Die Projektverantwortlichen treffen dann die Entscheidung, ob die Qualitätsprüfung bestanden ist und das Softwaremodell für die nächste Entwicklungsphase freigegeben werden kann (siehe ⑥), oder ob das Softwaremodell durch Modellierer weiter verbessert werden muss (siehe ⑤).

Falls das Softwaremodell zwecks einer Qualitätsverbesserung angepasst wurde, sollte anschließend die Qualitätssicherung erneut durchlaufen werden. Es kann schließlich passieren, dass neue Fehler in das Softwaremodell eingebaut oder die alten Fehler nicht korrekt behoben wurden.

Dieser Qualitätskreislauf verdeutlicht den Stellenwert der Qualitätsplanung innerhalb der analytischen Qualitätssicherung. Die Ergebnisse der Qualitätsplanung werden in den nachfolgenden Aktivitäten genutzt und beeinflussen ihre Ausführung entscheidend. Fehlen z.B. Qualitätsziele im Qualitätsplan, dann kann deren Erfüllung auch nicht überprüft werden. Die Qualitätsprüfung wird dadurch lückenhaft und evtl. fällt das Fehlen der Qualitätsziele den Entscheidungsträgern nicht einmal auf, so dass diese auf einer unvollständigen Grundlage Fehlentscheidungen treffen.

Bei der Erstellung eines Qualitätsplans vermischen sich zwei miteinander eng verzahnte Ebenen, die wir in unserer eingangs formulierten ersten Problemstellung in Abschnitt 1.2 nicht differenziert haben.

- *Vorgehensebene:* Der Qualitätsmanager benötigt ein Vorgehen, das ihn bei der Identifizierung von Qualitätszielen und der Ableitung von Qualitätsprüfungen unterstützt. Das Vorgehen benennt z.B. wichtige Informationen für eine erfolgreiche Erstellung des Qualitätsplans, die unbedingt genutzt werden sollten.

- *Ergebnisebene:* Zudem muss der Qualitätsmanager seine Ergebnisse für die anderen am Qualitätskreislauf beteiligten Rollen sichern, so dass diese den Qualitätsplan nutzen können.

Der vom Qualitätsmanager benötigte Ansatz sollte beide Ebenen adressieren. Für die im Folgenden aufgestellten Anforderungen an Qualitätsziele und Qualitätsprüfungen ist eine Unterscheidung der Vorgehens- und Ergebnisebene nicht von besonderem Interesse, weil die Anforderungen typischerweise durch beide Ebenen unterstützt werden können. Später im Kapitel 4, wenn wir konkrete Lösungen betrachten, werden wir differenzieren, inwiefern eine Lösung auf Vorgehens- oder Ergebnisebene greift.

Zuerst wenden wir uns im folgenden Abschnitt 2.3 den Anforderungen an Qualitätsziele zu. In Abschnitt 2.4 besprechen wir anschließend Anforderungen an Qualitätsprüfungen. Diese beiden Abschnitte präzisieren unsere Problemstellung 1 *welche Qualitätsziele wie überprüft werden sollen*, indem Anforderungen an mögliche Lösungen aufgeführt werden. Unserer zweiten Problemstellung der beschränkten Ressourcen widmen wir uns in Abschnitt 2.5 und vervollständigen die Menge der Anforderungen. Dafür nehmen wir die Perspektive der umfassenden Organisation ein. Abschließend fassen wir alle Anforderungen kompakt in einer Anforderungsmatrix

zusammen, die wir später im Kapitel 4 zur Bewertung verwandter Arbeiten und im Kapitel 7.2 zur Bewertung unserer eigenen Lösung einsetzen.

2.3 Anforderungen an Qualitätsziele

In der Literatur sind fachliche Anforderungen an die Erstellung und Dokumentation von Qualitätszielen für Softwaremodelle wenig thematisiert. Als Quelle für relevante Anforderungen an Qualitätsziele können wir Arbeiten mit einbeziehen, die *Anforderungen an Softwareanforderungsspezifikationen* festlegen. Denn zwischen Anforderungen an ein Softwareprodukt und Qualitätszielen für Softwaremodelle besteht eine Analogie. Im Umfeld der analytischen Qualitätssicherung von Softwaremodellen repräsentieren Qualitätsziele *Anforderungen an frühe Beschreibungen eines Softwareprodukts*.

Eine Softwareanforderungsspezifikation wird während der Anforderungsdefinition aufgestellt und legt fest, *was die Software leisten soll*. Sie bezieht sich immer auf eine Zielsetzung, die durch eine zu entwickelnde softwaretechnische Lösung erreicht werden soll. Den Anstoß zu einer Softwareentwicklung wird z.B. durch ein fachliches Problem gegeben. Die Überführung von einem solchen Problem in eine Softwareanforderungsspezifikation kann sehr schwierig sein, weil sowohl das Problem als auch die Zielsetzung sowie die Anforderungen zuerst nur gedanklich vorliegen. In der Folge sind die Anforderungen vage oder unvollständig und müssen noch präzisiert oder in Erfahrung gebracht und abschließend in einem Dokument festgehalten werden. Ähnlich verhält es sich auch für Softwaremodelle. Der Bezugsrahmen für Qualitätsziele ist hier allerdings durch Projektbeteiligte, ihre Verantwortungen und Tätigkeiten im Kontext des zu prüfenden Softwaremodells gegeben. Die Menge dieser Qualitätsziele beschreibt *im Kontext des zu prüfenden Softwaremodells was Modellqualität ist*. Abbildung 2.3 skizziert die oben diskutierte Analogie.

Aufgrund dieser Analogie untersuchen wir im Folgenden die Übertragbarkeit der im IEEE Standard für Softwareanforderungsspezifikationen [84] aufgestellten Anforderungen auf die Erstellung und Dokumentation von Qualitätszielen für Softwaremodelle. Der IEEE Standard fordert, dass eine Softwareanforderungsspezifikation korrekt, vollständig, unmissverständlich, widerspruchsfrei, gewichtet, überprüfbar, anpassbar und verfolgbar sein soll.

Die von uns aufgestellten Anforderungen werden wir als *Sollkriterien* formulieren.

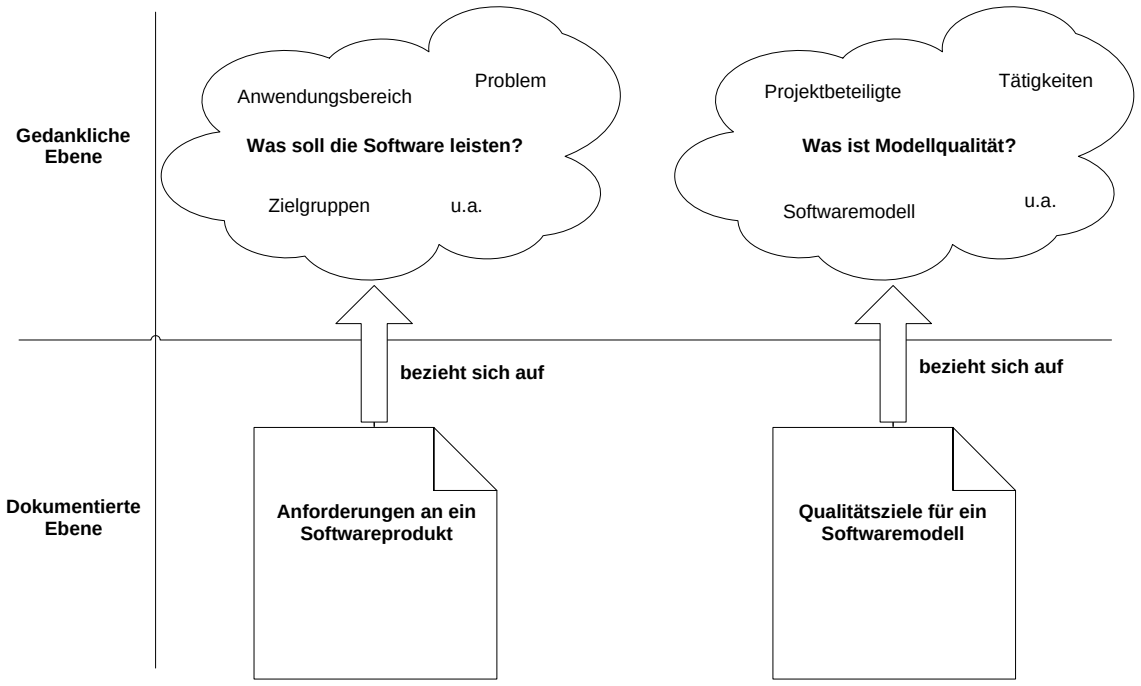


Abbildung 2.3: Analogie zwischen Anforderungen an ein Softwareprodukt (links) und Qualitätszielen für ein Softwaremodell (rechts)

Die Erfüllung dieser Anforderungen wird angestrebt, allerdings nicht als unabdingbar wie im Falle von *Musskriterien* verstanden.

Korrekte Qualitätsziele? Enthält beispielsweise ein Qualitätsplan die Forderung nach *ästhetischer Layoutgestaltung*, obwohl das zu prüfende Softwaremodell ausschließlich an einen Codegenerator übergeben werden soll, dann ist dieses Qualitätsziel *nicht korrekt* im Bezug auf den Kontext des zu prüfenden Softwaremodells. Der hier verwendete Korrektheitsbegriff kann aufgrund des nur gedanklich vorliegenden Bezugsrahmens nicht formal gefasst und somit die Korrektheit eines Qualitätsziels nicht formal bewiesen werden. Trotzdem muss ein Qualitätsmanager dafür Sorge tragen, dass ein Qualitätsplan möglichst nur korrekte Qualitätsziele enthält. Nicht korrekte Qualitätsziele bergen ein wirtschaftliches und ein fachliches Problem. Die Überprüfung dieser Qualitätsziele und die anschließende Interpretation der Prüfergebnisse binden Ressourcen und sind somit ineffizient. Aus fachlicher Perspektive können nicht korrekte Qualitätsziele die Interpretation der Prüfergebnisse durch Entscheidungsträger negativ beeinflussen und Fehlentscheidungen hinsichtlich der

Freigabe von Softwaremodellen verschulden. Deshalb gilt die Forderung nach *Korrektheit* ohne Einschränkung auch für Qualitätsziele von Softwaremodellen.

Anforderung 1 (*Korrekte Qualitätsziele*)

Qualitätsziele sollten korrekt in Bezug auf den Kontext des zu prüfenden Softwaremodells sein.

Vollständige Qualitätsziele? Fehlt in einem Qualitätsplan z.B. das für einen Softwarearchitekten wichtige Qualitätsziel *Entkopplung*, kann dies zur Weiterverwendung von Softwaremodellen führen, auf deren Basis ein schwer zu modifizierender Code implementiert wird. Fehlen Qualitätsziele, dann ist die Menge der im Qualitätsplan enthaltenen Qualitätsziele *unvollständig* in Bezug auf den Kontext des zu prüfenden Softwaremodells. Fehlende Qualitätsziele werden nicht überprüft. Ihre Abwesenheit kann fälschlicherweise zu einer Freigabe eines Softwaremodells und infolgedessen zu vermeidbaren Folgefehlern sowie Folgekosten führen. Folglich sollte die Menge der Qualitätsziele vollständig in Bezug auf den Kontext des zu prüfenden Softwaremodells sein.

Anforderung 2 (*Vollständige Qualitätsziele*)

Qualitätsziele sollten vollständig in Bezug auf den Kontext des zu prüfenden Softwaremodells sein.

Unmissverständliche Qualitätsziele? Wenn in einem Projektteam über Qualitätsziele diskutiert wird, dann sollten die wichtigsten Qualitätsbegriffe geklärt sein und einheitlich gebraucht werden. Ansonsten kann es passieren, dass Komplexität mit Kompliziertheit oder Verständlichkeit gleichgesetzt wird, eine gemeinsam akzeptierte Interpretation der Prüfergebnisse durch Projektverantwortliche nicht zu Stande kommt und der Entscheidungsprozess dadurch behindert wird. Damit alle Projektbeteiligten das Gleiche unter Qualität verstehen können, müssen die Begriffe verbindlich festgehalten werden. Ein auf diese Weise einheitlich definiertes Qualitätsverständnis verbessert die Kommunikation innerhalb des Projektteams.

Anforderung 3 (*Unmissverständliche Qualitätsziele*)

Qualitätsziele sollten für die Projektbeteiligten unmissverständlich formuliert sein.

Gewichtete Qualitätsziele? Die Gewichtung von Qualitätszielen wird wichtig, wenn sich nicht alle Qualitätsziele erfüllen lassen. Hier können Gewichte helfen, die richtigen Prioritäten festzulegen. Allerdings stellt eine sinnvolle Gewichtung von Qualitätszielen ein fundamentales Problem dar, weil z.B. geklärt werden muss, wie viele Semantikfehler man für einen Syntaxfehler bereit wäre hinzunehmen. Zur Zeit sehen wir noch genügend andere Herausforderungen, die zuvor zu bewältigen sind, und verfolgen deshalb gewichtete Qualitätsziele im Rahmen dieser Arbeit nicht weiter.

Widerspruchsfreie Qualitätsziele? Direkte Widersprüche lassen sich unserer Meinung nach für Qualitätsziele an Softwaremodelle nicht ohne weiteres konstruieren. Für einen Widerspruch müsste die Menge der Qualitätsziele Aussagen wie A und $\neg A$ enthalten. Die Erfüllung von Qualitätszielen ist aber per Definition gut. Deshalb widersprechen sich *korrekte* Qualitätsziele nicht, sondern ergänzen sich. Enthält ein Qualitätsplan die Forderungen nach A und $\neg A$, handelt es sich bei A und/oder $\neg A$ um kein *korrektes* Qualitätsziel (siehe Anforderung 1).

Für die Konstruktion von widersprüchlichen Qualitätszielen müssten quantitative Merkmale gefordert werden. Beispielsweise könnte dann anstatt Wartbarkeit bzw. Testbarkeit die widersprüchlichen Ziele *Durchschnittliche Vererbungstiefe aller Klassen* > 1 bzw. *Durchschnittliche Vererbungstiefe aller Klassen* < 1 gefordert werden. Unserer Auffassung zufolge handelt es sich bei derartigen quantitativen Zielen aber um keine Qualitätsziele. Projektverantwortliche möchten z.B. primär gute Wartbarkeit erzielen und nicht einen konkreten Wert für eine Messung über- oder unterschreiten. Quantitative Merkmale helfen lediglich bei der Bewertung, inwiefern z.B. Wartbarkeit erreicht wurde.

Folglich haben wir in unserer auf Softwaremodelle beschränkten Betrachtung die Anforderung nach widerspruchsfreien Qualitätszielen durch die Anforderung nach korrekten Qualitätszielen bereits abgedeckt.

Widerspruchsfreiheit sollte allerdings nicht mit *Konfliktfreiheit* gleichgesetzt werden. Qualitätsziele können durchaus zueinander im Konflikt stehen, ohne dass ein Widerspruch entsteht. Beispielsweise stehen die Forderungen (1.) nach einer sehr wartbaren und gleichzeitig (2.) hoch performanten Architektur häufig im Konflikt. Lösungsansätze zur Realisierung einer wartbaren und performanten Architektur beeinflussen sich wechselseitig negativ, so dass ein möglichst guter Kompromiss gesucht

werden muss. Die Qualitätsziele lassen sich jedoch vereinbaren und stehen nicht in einem Widerspruch. Ihre gleichzeitige Erfüllung ist lediglich schwierig.

Überprüfbare Qualitätsziele? Die Bedeutung der Prüfung von Qualitätszielen haben wir bereits durch die Ablaufbeschreibung der analytischen Qualitätssicherung in Abschnitt 2.2 dargelegt. Die Qualitätsprüfung liefert die Ergebnisse, auf deren Grundlage die Projektbeteiligten Entscheidungen über das weitere Projektvorgehen treffen. Lässt sich ein Qualitätsziel durch kein Verfahren prüfen, dann kann auch keine Aussage über den Erfüllungsgrad dieses Qualitätszieles getroffen werden. Deshalb fordern wir, dass Qualitätsziele durch die Anwendung von Verfahren überprüfbar sein sollten.

Allerdings kann die Prüfung von Qualitätszielen in der Praxis sehr schwierig werden. Beispielsweise kann das Qualitätsziel *Korrektheit* für ein Analysemodell nicht zweifelsfrei geprüft werden, wenn es auf einer Softwareanforderungsspezifikation basiert, die ihrerseits in Englisch geschrieben ist und infolgedessen Mehrdeutigkeiten enthält. Für andere Qualitätsziele wie der *externen Wiederverwendbarkeit*, also der Eignung eines Softwaremodells in anderen Entwicklungsprojekten wiederverwendet werden zu können, existieren bis dato überhaupt keine Verfahren zur Qualitätsprüfung.

Anforderung 4 (*Überprüfbare Qualitätsziele*)

Qualitätsziele sollten überprüfbar sein.

Anpassbare Qualitätsziele? Softwareentwicklungsprozesse ändern sich, um zusätzliche Herausforderungen zu bewältigen, Wünschen eines Auftraggebers zu entsprechen oder neuen Erkenntnissen der Forschung Rechnung zu tragen. Wenn sich z.B. der Entwicklungsprozess ändert, haben diese Änderungen mit großer Wahrscheinlichkeit auch Auswirkungen auf die verwendeten Qualitätspläne. Die Qualitätsziele müssen überarbeitet, evtl. ergänzt oder sogar verworfen werden (vgl. Abb. 2.4). Wenn notwendige Änderungen an den Qualitätszielen möglichst mit geringem Aufwand erkannt werden könnten, dann würde dies den Qualitätsmanager massiv entlasten.

Die Anpassung von Qualitätszielen hängt direkt mit ihrer Verfolgbarkeit zusammen. Deshalb können wir im Rahmen der analytischen Qualitätssicherung die For-

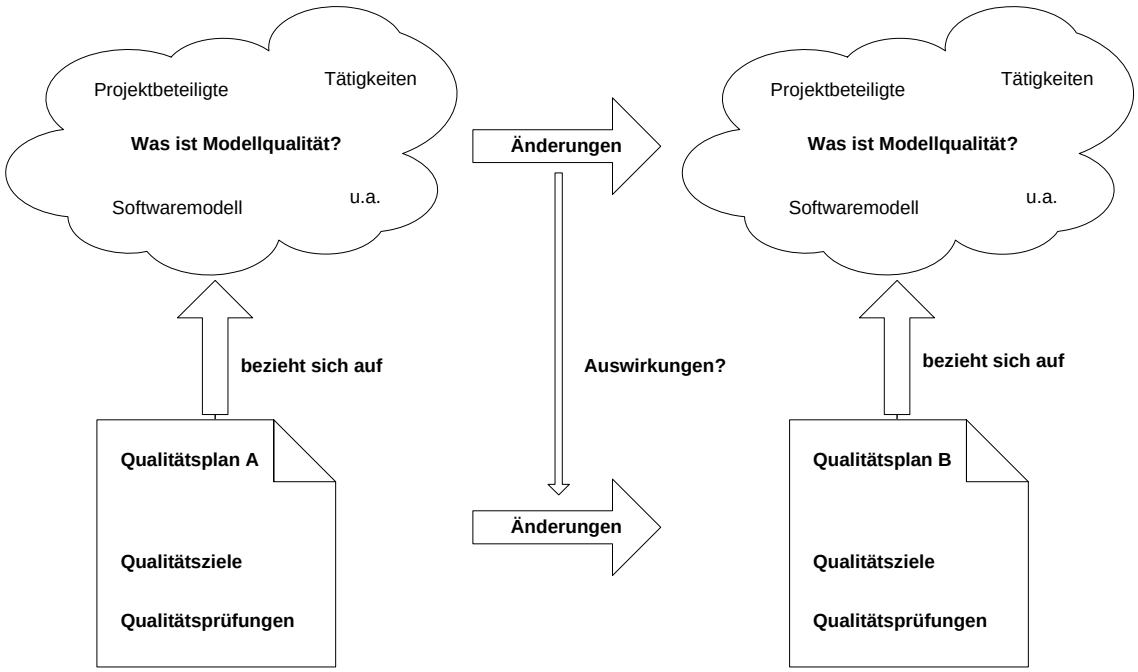


Abbildung 2.4: Wie wirken sich Änderungen in den Projektgegebenheiten auf Qualitätsziele aus?

derung nach anpassbaren Qualitätszielen unter der Forderung nach Verfolgbarkeit subsumieren.

Verfolgbare Qualitätsziele? Wenn z.B. für Qualitätsziele dokumentiert ist, auf welche Aspekte des Entwicklungsprozesses sie sich beziehen, dann zeigen Änderungen des Entwicklungsprozesses notwendige Änderungen an den Qualitätszielen auf. Diese Form der Verfolgbarkeit auf die Quelle einer Anforderung wird im Requirements Engineering als *Rückverfolgbarkeit* bezeichnet.

Anforderung 5 (*Rückverfolgbare Qualitätsziele*)

Qualitätsziele sollten rückverfolgbar sein.

Im Rahmen der Qualitätsplanung sollen aus Qualitätszielen auch passende Qualitätsprüfungen abgeleitet werden (vgl. Anforderung 4). Eine derartige *Vorwärtsverfolgbarkeit* ist zweckmäßig, da sich erstens notwendige Anpassungen an Qualitätsprüfungen anhand der zu ändernden Qualitätsziele erkennen lassen und zweitens für die Interpretation von Prüfergebnissen die zu Grunde liegenden Qualitätsziele bekannt sein sollten.

Anforderung 6 (*Vorwärtsverfolgbare Qualitätsziele*)
Qualitätsziele sollten vorwärtsverfolgbar sein.

2.4 Anforderungen an Qualitätsprüfungen

Anforderungen an Qualitätsprüfungen sind bereits viele formuliert worden (vgl. dazu [5], [52], [107], [176], [51], [56] und [10]). Im Folgenden diskutieren wir einige dieser Anforderungen im Rahmen der von uns aufgestellten Zielsetzung. Unter dem Punkt *weitere Begriffe* listen wir Wörter auf, die als Anforderungen in der Literatur genannt sind und sich im Wesentlichen auf die gleiche Anforderung beziehen.

Nützliche Qualitätsprüfungen? Wenn beispielsweise ein Softwarearchitekt wissen möchte, ob ein Softwaremodell komplex ist und die Komplexität mit den Ergebnissen eines Verfahrens zur Bestimmung der Größe von Softwaremodellen in einem kausalen Zusammenhang steht oder statistisch korreliert, dann ist diese Qualitätsprüfung nützlich. Ist dagegen die Größe des Softwaremodells für niemanden im Projekt von Interesse, weil die Qualitätsprüfung in keinem Zusammenhang zur Komplexität steht oder Komplexität für das zu prüfende Softwaremodell irrelevant ist, dann ist die entsprechende Qualitätsprüfung nicht nützlich.

Im Allgemeinen ist eine Qualitätsprüfung genau dann nützlich, wenn die Qualitätsprüfung eine Aussage über die Erreichung eines aufgestellten Qualitätsziels ermöglicht und für die Entscheidung, ob ein Softwaremodell für die nächste Phase freigegeben werden darf oder verbessert werden sollte, wichtig ist. Ähnlich wie nicht korrekte Qualitätsziele (vgl. Anforderung 1) bergen nicht nützliche Qualitätsprüfungen sowohl ein wirtschaftliches als auch fachliches Problem. Ist eine Qualitätsprüfung nicht nützlich, dann entstehen erstens während der Durchführung der Qualitätsprüfung und der entsprechenden Interpretation der Prüfergebnisse vermeidbare Kosten. Zweitens können nicht nützliche Qualitätsprüfungen während der Interpretationsphase die Projektverantwortlichen verwirren und Fehlentscheidungen provozieren.
Weitere Begriffe: valide

Anforderung 7 (*Nützliche Qualitätsprüfungen*)

Qualitätsprüfungen sollten nützlich sein.

Zuverlässige Qualitätsprüfungen? Eine Qualitätsprüfung wird als zuverlässig bezeichnet, wenn bei der Wiederholung der Durchführung unter denselben Bedingungen dieselben Prüfergebnisse erzielt werden. Die Zuverlässigkeit von Qualitätsprüfungen ist wichtig. Falls Prüfergebnisse nicht reproduzierbar sind, sinkt das Vertrauen in ihre Aussagekraft und als Konsequenz kann die Qualitätssicherung grundsätzlich in Frage gestellt werden. Zudem könnte ein Projektverantwortlicher auf die Idee kommen, die Qualitätsprüfungen solange zu wiederholen, bis die gewünschten Ergebnisse vorliegen.

Die Zuverlässigkeit von Prüfergebnissen kann allerdings nicht immer gewährleistet werden. Hierfür müssten alle Entscheidungen objektiv getroffen werden können. Dies ist beispielsweise nicht möglich, wenn die Qualitätsprüfungen semantische Analysen beinhalten und die Semantik des Softwaremodells nicht formal definiert ist.
Weitere Begriffe: stabil, präzise, objektiv, reproduzierbar

Anforderung 8 (*Zuverlässige Qualitätsprüfungen*)

Qualitätsprüfungen sollten zuverlässig sein.

Wirtschaftliche Qualitätsprüfungen? Die Erstellung und die Durchführung der Qualitätsprüfungen und die darauf basierende Interpretation der Prüfergebnisse binden Ressourcen und verursachen somit Kosten. Dabei ist das Ziel der Qualitätssicherung, Kosten einzusparen (z.B. durch Vermeidung von Fehlerkosten). Die Kosten für die Qualitätsprüfung dürfen die erwarteten Kosten für Mängel nicht übersteigen. Deshalb sollten Qualitätsprüfungen wirtschaftlich sein, also mit geringen Kosten erstellt und durchgeführt werden können.
Weitere Begriffe: einfach

Anforderung 9 (*Wirtschaftliche Qualitätsprüfungen*)

Qualitätsprüfungen sollten wirtschaftlich sein.

Interpretierbare Qualitätsprüfungen? Die Prüfergebnisse werden durch Projektverantwortliche im Rahmen des Projektkontextes interpretiert. Damit die Prüfergebnisse interpretiert werden können, sollte bereits vor ihrer Gewinnung klar sein,

was ein konkretes Ergebnis für ein Qualitätsziel bedeutet. Zudem ist für eine differenzierte Betrachtung der Prüfergebnisse der Kontext des zu Grunde liegenden Softwaremodells von Bedeutung. Prüfergebnisse müssen für ein finales Softwaremodell kritischer bewertet werden als für ein sehr frühes. Lassen sich Prüfergebnisse nicht interpretieren, dann kann auch keine fundierte Entscheidung für das weitere Vorgehen getroffen werden.
Weitere Begriffe: normiert, analysierbar, vergleichbar

Anforderung 10 (*Interpretierbare Qualitätsprüfungen*)
Durch Qualitätsprüfungen resultierende Prüfergebnisse sollten interpretierbar sein.

Verständliche Qualitätsprüfungen? Qualitätsprüfungen werden in einem Qualitätsplan definiert und danach unter Umständen sehr häufig durchgeführt. Hier kann es auch vorkommen, dass eine Person oder aber ein Prüf-Werkzeug die Qualitätsprüfung durchführt und dafür die jeweilige Definition der Qualitätsprüfung verstehen muss. Auch während der Interpretation der Prüfergebnisse greifen Projektverantwortliche auf die Definition von Qualitätsprüfungen zurück. Folglich können wir davon ausgehen, dass Qualitätsprüfungen häufiger gelesen als geschrieben werden. Qualitätsprüfungen sollten deshalb verständlich sein.

Anforderung 11 (*Verständliche Qualitätsprüfungen*)
Qualitätsprüfungen sollten verständlich sein.

2.5 Organisatorische Anforderungen an den Qualitätsplanungsprozess

Neben den aus dem Qualitätssicherungsprozess resultierenden fachlichen Anforderungen muss der Qualitätsmanager Anforderungen berücksichtigen, die sich aus der organisatorischen Perspektive ergeben.

Eine Unternehmung muss den effizienten Umgang mit knappen Ressourcen einfordern. Diese Forderung nach *Wirtschaftlichkeit* ist mit dem Ziel verknüpft, mit einem möglichst geringen Aufwand ein möglichst differenziertes und präzises Bild der Qualität des zu prüfenden Softwaremodells zu kennzeichnen. Diese wirtschaftlichen Überlegungen wirken stets restriktiv auf die Intensität der Qualitätsplanung,

also die Identifizierung von Qualitätszielen und die Ableitung von Qualitätsprüfungen sowie deren Dokumentation.

Wie in den Ausführungen zum Problem der beschränkten Ressourcen in Abschnitt 1.2 bereits dargelegt, stellt sich eine Verrechnung der vermeideten Fehlerkosten infolge einer Qualitätssicherung mit den zusätzlichen Kosten der Qualitätssicherung als sehr schwierig dar. Deshalb wird die Qualitätssicherung immer sehr beschränkten Ressourcen unterliegen, die ein Qualitätsmanager gleichzeitig effektiv wie auch sparsam einzusetzen hat.

Anforderung 12 (*Wirtschaftliche Qualitätsplanung*)

Die Qualitätsplanung sollte wirtschaftlich sein.

Die Möglichkeit der Sicherung und Wiederverwendung von Erfahrungswissen hängt eng mit der Forderung nach einer wirtschaftlichen Qualitätsplanung zusammen, hat aber darüber hinaus noch einen zweiten, entscheidenden Aspekt. Wenn Erfahrungswissen basierend auf vorangegangenen Qualitätssicherungen oder auf in der Literatur veröffentlichten Ergebnissen nicht mehr personengebunden, sondern innerhalb der Organisation allgemein zugänglich gesichert werden kann, dann wirkt sich dieses Erfahrungswissen positiv auf die Qualität von Qualitätsplänen und auf den Ressourcenbedarf während der Qualitätsplanung unabhängig vom zur Verfügung stehenden Personal aus.

Anforderung 13 (*Sicherung von Erfahrungswissen*)

Die Sicherung von Erfahrungswissen sollte möglich sein.

Neben dem Erfahrungswissen in Form von einzelnen Teilen eines Qualitätsplans oder sogar einer Qualitätsplanung als Ganzes sollte auch methodisches Wissen der Organisation als Ganzes und nicht ausschließlich einzelnen Personen zur Verfügung stehen. Deshalb ist für eine Organisation die Erlernbarkeit eines Ansatzes zur Qualitätsplanung wichtig. Dadurch kann die Rolle des Qualitätsmanagers von unterschiedlichen Personen ausgefüllt und der Ansatz innerhalb einer Organisation verbreitet werden.

Anforderung 14 (*Erlernbares Vorgehen zur Qualitätsplanung*)

Das Vorgehen zur Qualitätsplanung sollte erlernbar sein.

Anforderung		**Konzept**
Nr.	Kurzbeschreibung	
1	Korrekte Qualitätsziele	
2	Vollständige Qualitätsziele	
3	Unmissverständliche Qualitätsziele	
4	Überprüfbare Qualitätsziele	
5	Rückverfolgbare Qualitätsziele	
6	Vorwärtsverfolgbare Qualitätsziele	
7	Nützliche Qualitätsprüfungen	
8	Zuverlässige Qualitätsprüfungen	
9	Wirtschaftliche Qualitätsprüfungen	
10	Interpretierbare Qualitätsprüfungen	
11	Verständliche Qualitätsprüfungen	
12	Wirtschaftliche Qualitätsplanung	
13	Sicherung von Erfahrungswissen	
14	Erlernbares Vorgehen zur Qualitätsplanung	

Tabelle 2.1: Anforderungsmatrix

2.6 Anforderungsmatrix

Nachdem wir unsere Anforderungen an einen Lösungsansatz aufgestellt haben, fassen wir diese Anforderungen in Form einer Anforderungsmatrix zusammen (siehe Tabelle 2.1). Wir nutzen die Anforderungsmatrix um plakativ aufzuzeigen, inwiefern sich die Einbindung eines Konzepts auf eine mögliche Lösung *unserer Problemstellung* auswirkt.

Bei der Anwendung der Anforderungsmatrix auf konkrete Konzepte verwenden wir folgende Notation:

- $\oplus$ zeigt einen positiven Einfluss des bewerteten Konzepts auf die Erfüllung der jeweiligen Anforderung an.
- $\ominus$ zeigt einen negativen Einfluss des bewerteten Konzepts auf die Erfüllung der jeweiligen Anforderung an.
- n/z steht für nicht zutreffend und wird verwendet, falls uns die Bewertung eines Konzepts anhand einer Anforderung nicht sinnvoll erscheint.

- Die Notationen können beliebig kombiniert werden. Z.B. bedeutet die Konstellation $\oplus\ominus$, dass ein Konzept sowohl einen positiven als auch einen negativen Einfluss auf eine Anforderung hat. Wenn wir für ein Konzept einen stark positiven Einfluss auf eine Anforderung festhalten wollen, notieren wir $\oplus\oplus$, usw..

Kapitel 3

Modelle in der Softwareentwicklung

Im vorherigen Kapitel 2 haben wir wichtige Anforderungen formuliert, die ein *guter* Qualitätsplan erfüllen sollte. Für eine Reihe dieser Anforderungen wie z.B. korrekte Qualitätsziele (Anf. 1) oder nützliche Qualitätsprüfungen (Anf. 7) ist der gegebene Bezugsrahmen relevant. Ein *guter* Qualitätsplan kann demzufolge nicht generelle Gültigkeit besitzen und ohne Anpassungen auf alle möglichen Softwaremodelle angewendet werden. Stattdessen muss ein Qualitätsplan auf das zu prüfende Softwaremodell hin ausgerichtet sein. Deshalb verdeutlichen wir in diesem Kapitel den Bezugsrahmen, indem wir die Besonderheiten von Softwaremodellen und ihre Einbettung in den Softwareentwicklungsprozess beschreiben und dadurch den *Kontext von Softwaremodellen* charakterisieren.

Als erstes führen wir in Abschnitt 3.1 den allgemeinen Modellbegriff kurz ein und definieren ihn. Im Anschluss konzentrieren wir uns ausschließlich auf Modelle, die in der modellbasierten Softwareentwicklung genutzt werden. Modelle in der Softwareentwicklung bezeichnen wir im Folgenden als *Softwaremodelle*. Wir grenzen den Begriff Softwaremodell vom allgemeinen Modellbegriff ab, indem wir sukzessive die Besonderheiten von Softwaremodellen herausstellen und ihre Bedeutung für die Qualitätsbewertung verdeutlichen. Dafür gehen wir in Abschnitt 3.2 zuerst auf die modellbasierte Softwareentwicklung ein, in der Softwaremodelle zentrale Entwicklungsartefakte darstellen. Für die Erstellung von Softwaremodellen existieren diverse Modellierungssprachen, von denen wir einige in Abschnitt 3.3 thematisieren. Danach klassifizieren wir in Abschnitt 3.4 die Entwicklungsabhängigkeiten, die zwischen Softwaremodellen und anderen Entwicklungsartefakten entstehen können. In Abschnitt 3.5 untersuchen wir den Verwendungszweck von Softwaremodellen.

Abschließend fassen wir in Abschnitt 3.6 die wesentlichen Erkenntnisse für Softwaremodelle zusammen und definieren darauf basierend die Begriffe Softwaremodell, Kontext von Softwaremodellen und Kontextsensitivität.

3.1 Modelle im Allgemeinen

Modelle sind keine Errungenschaft der Neuzeit. Menschen verwenden Modelle seit jeher für die unterschiedlichsten Aufgaben. Ludewig bemerkt hierzu: "We use models when we think about problems, and when we talk to each other, and when we construct mechanisms, and when we try to understand phenomena (...). In short, we use models all the time" [110].

Deshalb ist es schwierig, Modelle im Allgemeinen zu definieren. Stachowiak hat ein Standardwerk zur *allgemeinen Modelltheorie* geschrieben [161]. Sein Modellbegriff ist nicht auf eine Fachdisziplin festgelegt. Seine Definition soll vielmehr domänenübergreifend, also allgemein anwendbar sein. Nach Stachowiak ist der Begriff *Modell* durch drei Merkmale gekennzeichnet:

Definition 3.1 (*Modell (Allgemein)*)

- ***Abbildung:*** *Ein Modell ist immer ein Abbild von etwas, eine Repräsentation natürlicher oder künstlicher Originale, die selbst wieder Modelle sein können.*
- ***Verkürzung:*** *Ein Modell erfasst nicht alle Attribute des Originals, sondern nur diejenigen, die dem Modellschaffer bzw. Modellnutzer relevant erscheinen.*
- ***Pragmatismus:*** *Pragmatismus bedeutet soviel wie Orientierung am Nützlichen. Ein Modell ist einem Original nicht von sich aus zugeordnet. Die Zuordnung wird durch die Fragen Für wen?, Warum? und Wozu? relativiert. Ein Modell wird vom Modellschaffer bzw. Modellnutzer innerhalb einer bestimmten Zeitspanne und zu einem bestimmten Zweck für ein Original eingesetzt. Das Modell wird somit interpretiert.*

Ein Modell zeichnet sich also durch Abstraktion aus. Bestimmte Merkmale werden

bewusst vernachlässigt, um die für den Modellierer oder den Modellierungszweck wesentlichen Informationen eines Originals hervorzuheben.

3.2 Modellbasierte Softwareentwicklung

Modellbasierte Softwareentwicklung ist ein Oberbegriff für Ansätze, die durch den gezielten Einsatz von Softwaremodellen einigen Schwächen herkömmlicher Entwicklungsprozesse entgegenwirken.

Dabei wird das eigentliche Softwaresystem anhand von Softwaremodellen schrittweise entwickelt und systematisch verfeinert. Infolgedessen stellen Softwaremodelle in der modellbasierten Softwareentwicklung das *zentrale Entwicklungsartefakt* dar. Sie sind das Bindeglied zwischen den Anforderungen der Kunden und dem Softwaresystem als Endprodukt zur Steigerung der Wertschöpfung. In der Literatur werden zahlreiche Stärken der modellbasierten Softwareentwicklung genannt [159, 23, 22, 90, 152, 182, 60].

Stärken der modellbasierten Softwareentwicklung

- Beherrschung der *Komplexität* des zu entwickelnden Softwaresystems durch Abstraktion
- *Kontrolle* von Kosten, Terminen und Entwicklungsfortschritt durch systematische, schrittweise Verfeinerungen
- Reduzierung des manuellen Implementierungsaufwandes durch automatische *Codegenerierung* aus Softwaremodellen
- *Frühzeitiges Entdecken von Fehlern* in der Anforderungsspezifikation durch Problemanalyse
- *Reduzierung des Portierungsaufwandes* auf weitere Middleware-Plattformen durch Trennung der Geschäftslogik von den Plattformspezifika
- *Koordination* und Parallelisierung von Implementierungsaufgaben anhand dokumentierter Softwarearchitekturen und detaillierter Softwareentwürfe für Komponenten, Schnittstellen, Datenstrukturen u.v.m. und dadurch bessere *Skalierung* des Entwicklungsfortschrittes

- Unterstützung der *Kommunikation* über Probleme und Lösungen im Projektteam auf Basis von Zwischenergebnissen
- *Wiederverwendung* bestehender Entwicklungsartefakte (sowohl Softwaremodelle als auch Softwarekomponenten)

Diese Stärken der modellbasierten Softwareentwicklung können zu einer *erhöhten Produktivität*, *verbesserter Teamarbeit*, *niedrigeren Entwicklungskosten* und einem *verminderten Risiko einer Fehlentwicklung* führen. Einige der oben genannten Stärken stehen im Mittelpunkt aktueller Trends wie der Model Driven Architecture (MDA) [126] oder dem modellbasierten Testen (MBT).

MDA ist eine konkrete Initiative der Object Management Group (OMG) [125] für die Umsetzung der modellbasierten Entwicklung. Der MDA-Ansatz beruht auf einer klaren Trennung von Funktionalität und Technik. Die Transformation eines Platform Independent Model (PIM) in ein oder mehrere Platform Specific Models (PSM) basiert auf plattformspezifischen Transformationen, die durch eine Menge von Regeln beschrieben sind. Ein Ziel des MDA-Ansatzes ist die Steigerung der Entwicklungsgeschwindigkeit. Bereits entwickelte Software lässt sich deutlich einfacher warten und auf andere Plattformen portieren, vorausgesetzt es existieren entsprechende Transformatoren. Die Transformation eines PIM in ein PSM kann meistens nicht vollständig automatisiert werden. Der Automatisierungsanteil variiert typischerweise zwischen 20 und 80 Prozent [94].

Das modellbasierte Testen (MBT) umfasst Ansätze zur systematischen Erstellung von Testfällen aus Modellen [18]. Die generierten Testfälle enthalten auch Informationen über das erwartete Verhalten, so dass sich durchgeführte Testfälle bewerten und Testergebnisse bestimmen lassen. Die Modelle können unabhängig zu den Entwicklungsmodellen erstellt oder aber aus diesen abgeleitet werden. [144] gibt einen Überblick über die verschiedenen MBT-Szenarien. Testmodelle ermöglichen die Abstraktion von technischen Details. Dadurch lassen sich Testfälle zentral definieren und für verschiedene Plattformen wiederverwenden. Zudem sind diese abstrakten Testfälle leichter anzupassen, falls sich z.B. das System Under Test (SUT) aufgrund behobener Mängel ändert.

Allerdings werden die Stärken eines modellbasierten Vorgehens nicht zwangsläufig erzielt. Der Erfolg hängt nicht von der bloßen Tatsache ab, dass Softwaremodelle verwendet werden. Der Erfolg bestimmt sich vielmehr durch Einflussgrößen wie der

Qualität des Entwicklungsprozesses, Qualität der eingesetzten Ressourcen und eben der Qualität der Softwaremodelle selbst.

Nachdem wir die Stärken der modellbasierten Softwareentwicklung vorgestellt haben, beschreiben wir im Folgenden den grundlegenden Ablauf der modellbasierten Softwareentwicklung. Beim Durchlaufen der einzelnen Entwicklungsphasen werden Zwischenergebnisse in Entwicklungsartefakten, auch als Entwicklungsdokumente bezeichnet, dokumentiert. Neben der eigentlichen Software werden zahlreiche andere Entwicklungsartefakte erstellt. Eine Softwareanforderungsspezifikation, ein Entwurfsdokument oder Quellcode sind geläufige Beispiele für Entwicklungsartefakte.

In der modellbasierten Softwareentwicklung nutzen Personen, die spezielle Rollen wahrnehmen, Entwicklungsartefakte, um z.B. Ergebnisse zu sichern oder weiterzuverwenden. Meistens lassen sich Entwicklungsartefakte schwerpunktmäßig einer Entwicklungsphase zuordnen, in der sie erstellt werden.

Diese Entwicklungsartefakte werden wiederum in anderen Entwicklungsphasen weiterverwendet. Auf diese Weise entstehen *Entwicklungsabhängigkeiten* zwischen Entwicklungsartefakten. Eine Entwicklungsabhängigkeit beschreibt den fachlichen Zusammenhang zwischen Entwicklungsartefakten.

Wenn Quellcode während der Entwicklungsphase *Implementierung* durch die Rolle *Entwickler* auf Basis eines Softwaremodells implementiert wird, das in der Entwicklungsphase *Entwurf* entstanden ist, dann entsteht zwischen diesen Entwicklungsartefakten eine *Entwicklungsabhängigkeit*. In der modellbasierten Softwareentwicklung werden viele Entwicklungsartefakte mit Hilfe von Softwaremodellen dargestellt.

Obwohl der genaue Lebenszyklus eines Softwareprodukts abhängig vom verwendeten Vorgehensmodell variiert, lässt sich für den Fall der modellbasierten Softwareentwicklung exemplarisch der in Abbildung 3.1 dargestellte Softwarelebenszyklus skizzieren.

Der Softwarelebenszyklus beginnt mit einem *Problem*, das softwaretechnisch vollständig bzw. teilweise gelöst werden soll. Dieses Problem wird z.B. in Form von *Text* definiert und gibt den Anstoß zur Softwareentwicklung. Anforderungen an die zu erstellende Software sowie damit verbundene Leistungen werden während der *Anforderungsdefinition* in der *Softwareanforderungsspezifikation* dokumentiert. Typischerweise ist eine Anforderungsspezifikation eher *textlastig* und enthält nur vereinzelt *Softwaremodelle*. In der *Analyse* werden Geschäftsabläufe, Anwendungsfälle

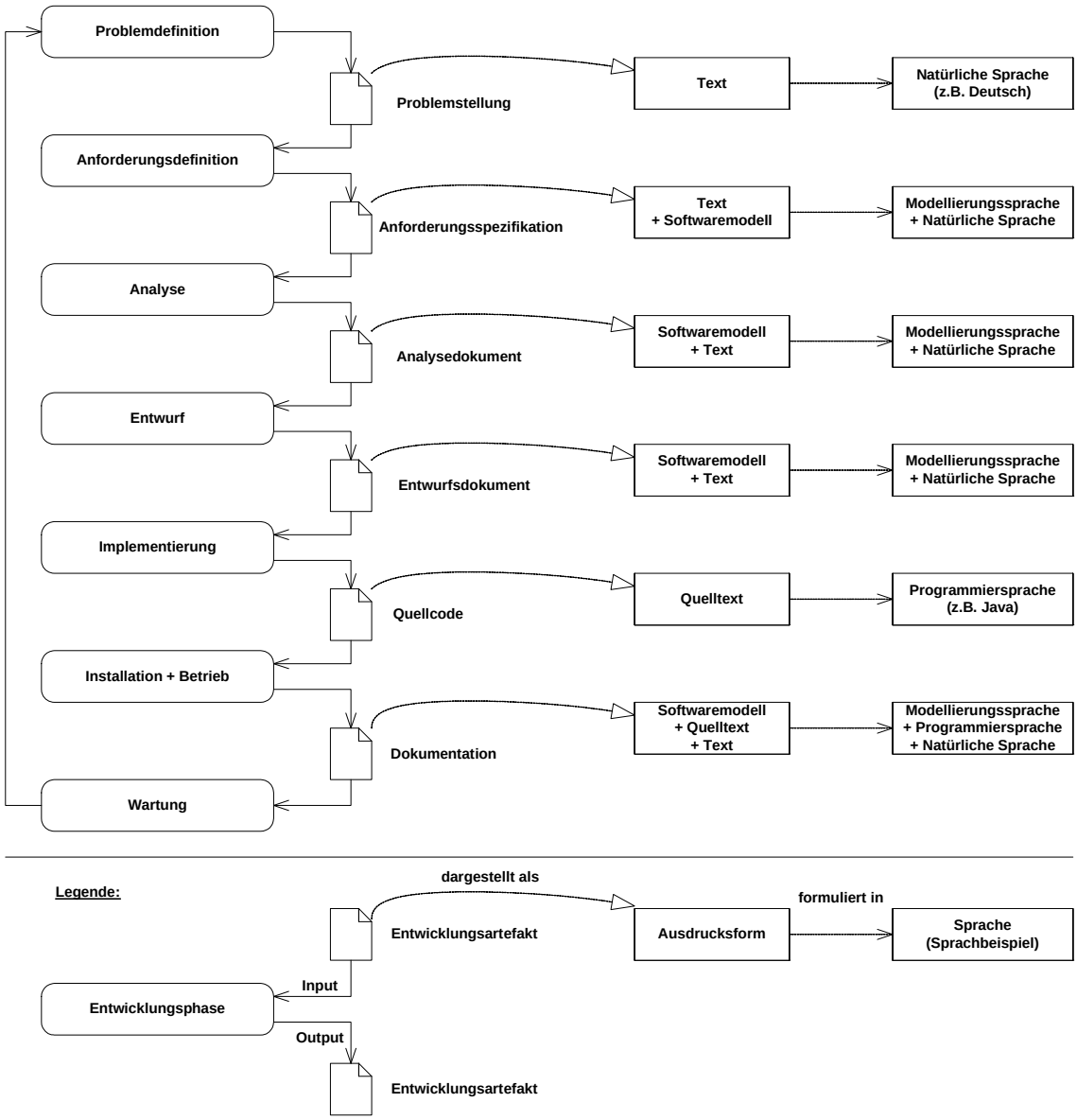


Abbildung 3.1: Softwarelebenszyklus

und damit verknüpfte Szenarien beschrieben. Die Ergebnisse fließen in das *Analysedokument* ein. Es besteht bereits zu einem großen Teil aus einem oder mehreren *Softwaremodellen*. *Text* ergänzt die Beschreibung an mehreren Stellen. Der *Entwurf* verfeinert die Ergebnisse der Analyse. Im Grobentwurf werden eine Softwarearchitektur und ihre wichtigsten Komponenten sowie Schnittstellen geplant. Im darauf basierenden Feinentwurf können anschließend die einzelnen Komponenten weiter detailliert und zu Klassenstrukturen verfeinert werden. Das *Entwurfsdokument* lässt sich überwiegend mit einem oder mehreren *Softwaremodellen* formulieren. Während der *Implementierung* wird der Softwareentwurf in eine Reihe von Programmen und Programmeinheiten umgesetzt, die beispielsweise von einem Programmierer in *Java Quellcode* geschrieben werden. Anschließend wird die fertige Software ausgeliefert und *installiert* und der eigentliche *Betrieb* der Software beginnt. Im Gegensatz zu den meisten anderen Produkten unterliegt Software keinem Verschleiß. Allerdings muss Software permanent gewartet werden. In der *Wartung* werden Fehler korrigiert, die vorher nicht aufgetreten sind oder entdeckt wurden. Zudem können Anpassungen der Software erfolgen, um geänderte oder neue Anforderungen zu erfüllen. Somit durchläuft eine Software während ihrer Wartung eine weitere Iteration im Softwarelebenszyklus. In der Wartung wird die *Dokumentation* der Software verwendet, die z.B. aus dem Quelltext, Softwaremodellen und weiteren textuellen Erläuterungen besteht.

Die Abbildung 3.1 veranschaulicht mehrere wichtige Aspekte der modellbasierten Softwareentwicklung gleichzeitig.

- **Softwaremodelle basieren auf Modellierungssprachen:** Viele Entwicklungsartefakte werden mit Hilfe von Softwaremodellen ausgedrückt. Die Formulierung von Softwaremodellen basiert auf Modellierungssprachen. Dabei kann dieselbe Modellierungssprache durchaus für mehrere, sehr unterschiedliche Entwicklungsartefakte eingesetzt werden.

- **Entwicklungsabhängigkeiten von Softwaremodellen:** Softwaremodelle hängen inhaltlich von anderen Entwicklungsartefakten ab. Dadurch entstehen Entwicklungsabhängigkeiten, in denen ein Softwaremodell ein oder mehrere Entwicklungsartefakte spezifiziert, beschreibt, verfeinert und/oder von ihnen abstrahiert.

- **Softwaremodelle als zentrale Entwicklungsartefakte:** Softwaremodelle

weisen Entwicklungsabhängigkeiten sowohl zu vorgelagerten als auch zu nachgelagerten Phasen auf. Das Softwaresystem wird auf verschiedenen Abstraktionsebenen geplant und schrittweise entwickelt. Der Entwurf wird in immer kleinere Teile zerlegt, so dass detaillierte Softwaremodelle auf abstrakteren Softwaremodellen basieren. Zur erfolgreichen Umsetzung dieses Vorgehens sind Softwaremodelle hierbei der Dreh- und Angelpunkt.

- **Verwendungszwecke von Softwaremodellen:** Softwaremodelle werden für verschiedene Aufgaben in verschiedenen Phasen des Softwarelebenszyklus verwendet. Sie dienen damit unterschiedlichen Verwendungszwecken.

Die obigen Aspekte spielen für die Qualitätsbewertung und somit auch für die Qualitätsplanung von Softwaremodellen eine wichtige Rolle. Im Folgenden gehen wir auf diese Aspekte gezielt ein. Zuerst betrachten wir im nächsten Abschnitt Modellierungssprachen, die in der modellbasierten Softwareentwicklung eingesetzt werden.

3.3 Modellierungssprachen

In der Informatik gibt es viele Modellierungssprachen. Einige Modellierungssprachen sind für ein breites Einsatzfeld konzipiert (z.B. UML [131], Petri Netze [146] oder ERM [36]). Für einige Einsatzfelder gibt es sehr spezialisierte Modellierungssprachen (UWE [98], UML Testing Profile [128], CSP-OZ-DC [108]).

Die Eignung einer Modellierungssprache für den Einsatz in einem Softwareentwicklungsprojekt hängt von vielen Faktoren ab. Dazu zählen die Anwendungsdomäne (Finanzsektor, Fahr- und Flugzeugtechnik, Medizintechnik), das Zielsystem (eingebettetes System, Web Applikation, Datenbankanwendung), verfügbares Mitarbeiter Know-how, eingesetzte Entwicklungsmethoden, Projekt- und Applikationsrisiken (Neuartigkeit der Software, sicherheitskritisches System), vorhandene Werkzeuge usw..

Die Unified Modeling Language (UML) [131] gilt de facto als Standard in der modellbasierten Softwareentwicklung. Sie vereint einen ganzen Satz an *Diagrammtypen*, die z.T. aus eigenen Modellierungssprachen hervorgegangen sind (z.B. ER Diagramme → Klassendiagramme oder Harel's Statemachines [73] → Statecharts). Insgesamt umfasst die UML 13 Diagrammtypen.

Der Beitrag der UML liegt in der Konsolidierung, Integration und Standardisierung von Modellierungsansätzen und der Auswahl eines Kanons an Modellierungskonzepten [162]. Unabhängig davon, ob man Anwendungsfälle mit einem Use Case Diagramm, Szenarien mit einem Sequenzdiagramm, eine Softwarearchitektur mit einem Komponentendiagramm oder einen Objektlebenszyklus mit einem Statechart modelliert, die UML bietet geeignete Modellierungskonzepte an. Deshalb hat die UML auch eine so hohe Verbreitung in Wissenschaft und Industrie.

Der Erfolg der UML erklärt sich auch durch die große Auswahl an Werkzeugen, Trainingsanbietern, Fachbüchern usw.. Softwaremodelle und Wissen darüber können problemlos ausgetauscht werden. Dies ist insbesondere der Standardisierung der UML zu verdanken.

Aufgrund dieser hohen Verbreitung und der breiten Ausrichtung der UML möchten wir uns insbesondere auf Softwaremodelle konzentrieren, die auf Basis der UML bzw. eines UML-Profils erstellt sind. UML-Profile sind Erweiterungen der UML-Sprachdefinition. Der Profilmechanismus ermöglicht die Definition eigener Stereotypen zur Ergänzung plattformspezifischer Konzepte.

Die Syntax von UML-Softwaremodellen wird durch die UML-Superstructure festgelegt [131]. Das UML-Metamodell legt die abstrakte Syntax formal fest. Weitere syntaktische Randbedingungen werden durch zusätzliche Wohlgeformtheitsregeln angegeben, die in der Object Constraint Language (OCL) [127] definiert sind. UML-Softwaremodelle können in der abstrakten Syntax bzw. einer entsprechenden Serialisierung gespeichert werden.

Durch den Aufbau der UML und des Profilmechanismus ergibt sich, dass wir grundsätzlich zwischen *Diagrammen* und *Softwaremodellen* unterscheiden können. Diagramme visualisieren die konkrete Syntax. Die konkrete Syntax besteht aus den bekannten Notationselementen der UML wie Linien, Pfeile, Rechtecke usw.. Mehrere Diagramme können sich auf ein und dasselbe Softwaremodell beziehen und sich dabei inhaltlich überlappen. Abbildung 3.2 stellt den Aufbau einer Modellierungssprache auf Basis eines Metamodells dar und zeigt den Unterschied zwischen abstrakter und konkreter Syntax. Die Syntax der Softwaremodelle ist ausreichend genau spezifiziert, so dass ihre Einhaltung bereits von den meisten UML-Werkzeugen weitestgehend sichergestellt wird.

Neben der Syntax definiert eine Modellierungssprache auch die Semantik. Semantik ist die Bedeutung von syntaktischen Elementen. Somit bestehen die von uns

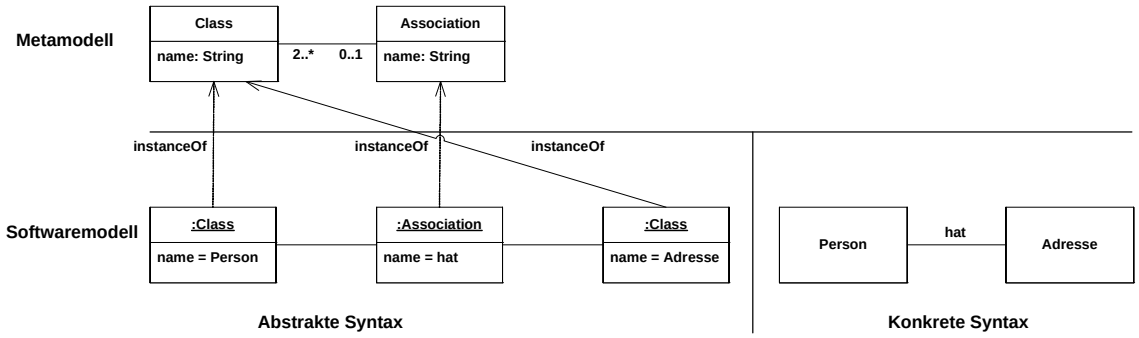


Abbildung 3.2: Syntax von Modellierungssprachen

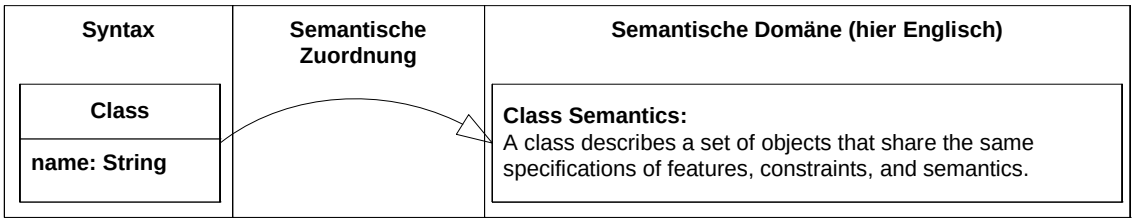


Abbildung 3.3: Semantik von Modellierungssprachen

betrachteten Modellierungssprachen aus syntaktischen Elementen und einer semantischen Zuordnung von den syntaktischen Elementen in eine semantische Domäne (vgl. hierzu auch [74]). Abbildung 3.3 verdeutlicht den Zusammenhang zwischen Syntax und Semantik.

Für die Semantikdefinition lassen sich verschiedene Formalisierungsgrade unterscheiden. Wenn eine Semantik in einer natürlichen Sprache wie Englisch gegeben ist, dann bezeichnen wir die Modellierungssprache als informal. Für textuelle Sprachspezifikationen bemerkt Hausmann, dass "the ambiguity of the specification text and actual flaws in the specification erode the users' confidence and invite the conception of individual language interpretations which deviate from the standard" [78]. Ist dagegen die semantische Domäne einer Sprachdefinition mittels eines Transitionssystems, einer Mengenalgebra, einer Aussagenlogik oder ähnlichem definiert, dann bezeichnen wir die Sprache als formal. Es existieren bereits mehrere Ansätze, die für Teile der UML eine formale Semantikdefinition vorschlagen (z.B. [78, 100, 171, 164]). Dabei ist es durchaus möglich, dass eine formale semantische Domäne erst durch mehrere, transitive, semantische Zuordnungen erreicht wird. Dies ist z.B. für UML/Z der Fall [25]. Diese Ansätze müssen enthaltene Unklarheiten in

der gegebenen Sprachdefinition auflösen und sich beim Auftreten von Konflikten für eine Interpretation entscheiden. Sie nehmen somit Veränderungen an der ursprünglichen Semantik vor. Derartige Modellierungssprachen verwenden die UML-Syntax und ersetzen die Semantikdefinition.

Der Formalisierungsgrad der Semantik einer Modellierungssprache wirkt sich auf mögliche Qualitätsprüfungen von Softwaremodellen aus. Falls keine formale Semantik gegeben ist, lassen sich semantische Analysen wie die Erkennung von Deadlocks kaum automatisieren. Werden die Analysen manuell durchgeführt, dann existieren unter Umständen mehrere gültige Interpretationen für ein Softwaremodell, so dass ein Mangel nicht zweifelsfrei erkannt werden kann.

Zusammenfassung

Wir betrachten insbesondere Softwaremodelle, die auf der *UML* bzw. auf *UML-Profilen* basieren. Die *Syntax* dieser Modellierungssprachen ist durch ein Metamodell und Wohlgeformtheitsregeln *formal* spezifiziert. Entsprechende Modellierungswerkzeuge stellen bereits einen Großteil der syntaktischen Korrektheit sicher. Die *Semantikdefinition* dieser Modellierungssprachen reicht von *informal* bis hin zu *formal*. Die Qualitätsprüfung von Softwaremodellen mit informaler Semantik ist unter Umständen einigen Einschränkungen unterworfen.

3.4 Entwicklungsabhängigkeiten

Wie zuvor in Abbildung 3.1 schematisch dargestellt, weisen Softwaremodelle Entwicklungsabhängigkeiten zu anderen Entwicklungsartefakten auf. In [110] werden zwei Arten von Entwicklungsabhängigkeiten unterschieden. Ein Softwaremodell kann z.B. per Reengineering aus einem bestehenden Softwaresystem erstellt werden. Dann *beschreibt* das Softwaremodell das Softwaresystem. Dient ein Softwaremodell z.B. als Grundlage für eine Implementierung, dann *spezifiziert* das Softwaremodell das Softwaresystem. Neben diesen beiden Begriffen *Beschreibung* und *Spezifikation* sind zwei weitere Begriffe sehr geläufig: *Verfeinerung* und *Abstraktion*. Z.B. *abstrahiert* ein Softwaremodell von konkreten Anwendungsszenarien der Anforderungsspezifikation. Ein Softwaremodell *verfeinert* ein anderes Softwaremodell, wenn beispielsweise plattformspezifische Details hinzugefügt wurden.

Dabei ist zu beachten, dass ein Softwaremodell gleichzeitig mehrere dieser Entwicklungsabhängigkeiten zu ein und demselben Entwicklungsartefakt aufweisen kann. Z.B. abstrahiert ein Softwaremodell von den Aktivitäten in einem Geschäftsprozess, die nicht softwaretechnisch unterstützt werden sollen. Alle übrigen Aktivitäten werden dagegen von dem Softwaremodell weiter verfeinert.

Zudem können Softwaremodelle andere Entwicklungsartefakte *ergänzen*, so dass beide zusammen das Ergebnis einer Entwicklungsphase bilden. Dabei sind ergänzende Entwicklungsartefakte typischerweise auf einem vergleichbaren Abstraktionsniveau dargestellt. Jedes dieser Entwicklungsartefakte stellt schwerpunktmäßig einen bestimmten Aspekt heraus. Beispielsweise werden Begriffe in einer Tabelle gelistet und definiert. Ein Softwaremodell modelliert diese Begriffe und deren Zusammenhänge mittels eines Klassendiagramms. Dann *ergänzt* das Softwaremodell die Tabelle.

Entwicklungsabhängigkeiten von Softwaremodellen

- *Softwaremodell als Beschreibung:* Ein Softwaremodell beschreibt ein Entwicklungsartefakt, indem es wichtige Informationen modelliert.
- *Softwaremodell als Spezifikation:* Ein Softwaremodell spezifiziert ein Entwicklungsartefakt, indem es wichtige Informationen vorschreibt.
- *Softwaremodell als Abstraktion:* Ein Softwaremodell abstrahiert von einem Entwicklungsartefakt, indem es Informationen verkürzt darstellt oder ganz weglässt.
- *Softwaremodell als Verfeinerung:* Ein Softwaremodell verfeinert ein Entwicklungsartefakt, indem es Informationen hinzufügt, die vorher nicht dargestellt waren.
- *Softwaremodell als Ergänzung:* Ein Softwaremodell ergänzt ein Entwicklungsartefakt, indem es Informationen modelliert, die im anderen Entwicklungsartefakt nicht oder nur beiläufig dargestellt werden. Auf diese Weise vervollständigen sich beide Entwicklungsartefakte gegenseitig.

Folglich stehen Softwaremodelle mit anderen Entwicklungsartefakten auf vielfältige Weise in Beziehung. Zudem unterstützt die Modellierungssprache UML meh-

rere Diagrammtypen. Verschiedene Diagrammtypen eignen sich für die Modellierung spezieller Aspekte wie Struktur und Verhalten. Bei den strukturellen Aspekten lassen sich konzeptionelle, logische und physische Strukturen unterscheiden. Diese Unterteilung kann noch weiter verfeinert werden. Z.B. beziehen sich Schnittstellen-, Komponenten- und Datenstruktur-Aspekte auf konzeptionelle Strukturen. Verhaltens-Aspekte zerfallen in Zustands-, Interaktions- und Ablauf-Aspekte. Ein Diagramm stellt häufig mehrere dieser Aspekte gleichzeitig dar. I.d.R. wird ein Aspekt vorrangig dargestellt, andere Aspekte dagegen weniger explizit. Modellierte Diagramme kapseln folglich spezielle Aspekte eines Gesamtmodells und ergänzen sich auf diese Weise.

Diese inhaltlichen Überlappungen in den eingesetzten Diagrammen sowie die Entwicklungsabhängigkeiten zwischen einem Softwaremodell einerseits und anderen Entwicklungsartefakten andererseits entsprechen Konsistenzbeziehungen. In [100] werden Konsistenzbeziehungen und Konsistenzprobleme untersucht und wie folgt klassifiziert:

Klassifikation von Konsistenzbeziehungen

Zwischen zwei sich inhaltlich überlappenden Softwaremodellen bzw. Diagrammen liegt ...

- eine *Horizontale Konsistenzbeziehung* vor, wenn sie in der gleichen Entwicklungsphase erstellt wurden.
- eine *Vertikale Konsistenzbeziehung* vor, wenn sie in verschiedenen Entwicklungsphasen erstellt wurden.
- eine *Intra-Modell Konsistenzbeziehung* vor, wenn die Diagramme Sichten auf dasselbe Softwaremodell sind.
- eine *Inter-Modell Konsistenzbeziehung*, wenn die Diagramme Sichten auf logisch und physisch getrennte Softwaremodelle sind.
- eine *Intra-Aspekt Konsistenzbeziehung*, wenn sie den gleichen Aspekt darstellen.
- eine *Inter-Aspekt Konsistenzbeziehung*, wenn sie verschiedene Aspekte darstellen.

- eine *Intra-Sprache Konsistenzbeziehung*, wenn die Softwaremodelle in der gleichen Modellierungssprache dargestellt sind.
- eine *Inter-Sprache Konsistenzbeziehung*, wenn die Softwaremodelle in verschiedenen Modellierungssprachen dargestellt sind.

Konsistenzbeziehungen stellen mögliche Quellen für Konsistenzprobleme dar. Enthält das zu prüfende Softwaremodell Inkonsistenzen, sollten diese so früh wie möglich erkannt und behoben werden, weil sich widersprechende Informationen nicht sinnvoll interpretieren lassen. Deshalb ist für die Qualitätsbewertung von Softwaremodellen die Berücksichtigung der Entwicklungsabhängigkeiten und der eingesetzten Diagrammtypen von Bedeutung.

Zusammenfassung

Inhaltliche Überlappungen in den eingesetzten Diagrammen sowie die Entwicklungsabhängigkeiten zwischen dem zu prüfenden Softwaremodell einerseits und Entwicklungsartefakten andererseits entsprechen Konsistenzbeziehungen, die wiederum zu Konsistenzproblemen führen können. Konsistenzprobleme sollten während der analytischen Qualitätssicherung erkannt werden.

3.5 Verwendungszweck von Softwaremodellen

In der modellbasierten Softwareentwicklung werden Softwaremodelle zielgerichtet zur Erfüllung verschiedener Aufgaben in verschiedenen Entwicklungsphasen eingesetzt. Softwaremodelle werden von Personen genutzt, die verschiedene Rollen wahrnehmen. Eine Rolle zeichnet sich durch Aufgaben, Rechte und Verpflichtungen aus. Für einen Entwicklungsprozess lassen sich eine Reihe verschiedener Rollen wie z.B. Projektleiter, Softwarearchitekt, Testmanager, Entwickler oder Requirements Engineer benennen. Softwaremodelle müssen aus Sicht mindestens einer im Projekt besetzten Rolle nützlich sein und einem Verwendungszweck in der Modellierung dienen. Der Verwendungszweck beeinflusst die Modellierung erheblich. Diese Orientierung am Nützlichen bezeichnet Stachowiak als Pragmatismus. Der Pragmatismus ist ein Merkmal des allgemeinen Modellbegriffs (vgl. Def. 3.1). Auch die UML Superstructure [131] greift den Verwendungszweck in der Definition des Konzepts *Model*

auf:

Definition 3.2 (*UML-Model*)

A model captures a view of a physical system. It is an abstraction of the physical system, with a certain purpose. This purpose determines what is to be included in the model and what is irrelevant. Thus the model completely describes those aspects of the physical system that are relevant to the purpose of the model, at the appropriate level of detail.

Mit der Nutzung von Softwaremodellen werden mehrere Verwendungszwecke gleichzeitig verfolgt. Verwendungszwecke von Softwaremodellen ordnen wir grob in zwei Kategorien ein:

Verwendungszwecke von Softwaremodellen

- *Organisatorische und prozessbezogene Verwendungszwecke:* Durch die Nutzung eines Softwaremodells soll die Projektarbeit unterstützt werden. Einige der folgenden Beispiele für Verwendungszwecke treffen sogar auf alle Einsatzszenarien von Softwaremodellen zu.
  Beispiele:
  - Unterstützung der Kommunikation, Koordination und Zusammenarbeit zwischen Projektmitgliedern
  - Dokumentation des Entwicklungsfortschrittes
  - Rechtzeitiges Erkennen der Komplexität und Kompliziertheit des Modellierungsgegenstandes und dadurch Beherrschung der Risiken im Projekt
  - Wiederverwendung bestehender Lösungen
  - Automatische Codegenerierung
- *Fachliche Verwendungszwecke:* Zudem soll durch die Nutzung eines Softwaremodells immer eine konkrete, fachliche Aufgabenstellung gelöst werden, die mit der im Softwareentwicklungsprojekt gegebenen Problemstellung zusammenhängt. Diese Verwendungszwecke sind im Vergleich zu den obigen sehr speziell.
  Beispiele:

- Modellierung eines Datenbankschemas
- Modellierung eines Ist-Geschäftsprozesses
- Modellierung eines Soll-Geschäftsprozesses
- Modellierung eines Referenz-Geschäftsprozesses
- Modellierung eines Protokolls
- Modellierung der statischen Struktur von Objekten
- Modellierung des Verhaltens von Objekten
- Modellierung der Softwarearchitektur
- Spezifikation von Testfällen für den Systemtest
- u.v.m.

Softwaremodelle können sehr umfangreich werden und viele Diagramme beinhalten. Insbesondere die Verwendungszwecke, die sich auf den fachlichen Nutzen beziehen, lassen sich für einzelne Diagrammtypen differenzieren. Sind für ein Softwaremodell die Sichten *Komponentendiagramm* und *Statechart* vorgesehen, dann besteht beispielsweise der Verwendungszweck der Komponentendiagramme in der Spezifikation der Architektur. Die Statecharts sollen dagegen dazu verwendet werden, um den Lebenszyklus einer Komponenteninstanz zu beschreiben. Beide Verwendungszwecke gelten auch für das Softwaremodell. Sie lassen sich aber auch den einzelnen Diagrammtypen zuordnen, so dass dadurch eine höhere Genauigkeit erzielt wird.

Ein Verwendungszweck ist nicht zwangsläufig an einen Diagrammtypen gekoppelt. Beispielsweise werden Klassendiagramme sehr unterschiedlich verwendet (vgl. Tab. 3.1).

Ein Diagrammtyp kann sogar im selben Softwaremodell für unterschiedliche Verwendungszwecke eingesetzt werden. Wir möchten das am Beispiel des Diagrammtyps *Statechart* verdeutlichen. Die eine Menge von Statecharts spezifiziert Protokolle für die Kommunikation. Die andere Menge an Statecharts spezifiziert den Lebenszyklus von Objekten. Folglich können Statecharts im selben Softwaremodell für unterschiedliche Zwecke eingesetzt.

Deshalb differenzieren wir zwischen dem Verwendungszweck eines Softwaremodells, im Folgenden als *Modellzweck* bezeichnet, und dem Verwendungszweck eines Diagrammtyps, im Folgenden als *Diagrammzweck* bezeichnet.

Dokument	Phase	Rolle	Zweck
Modell des Problembereichs	Anforderungs-definition	Anfoderungs-analyst	Präzisierung der Zusammenhänge von Fachbegriffen
Analysemodell	Analyse	Software-architekt	Grundlegende Softwarearchitektur
Entwurfsmodell	Entwurf	Software-architekt	Detaillierte Spezifikation der Komponenten

Tabelle 3.1: Verwendungszwecke von UML-Klassendiagrammen nach [162] und [135]

Für die Qualitätsbewertung eines Softwaremodells ist die Dokumentation der Verwendungszwecke von Bedeutung. Ist kein Verwendungszweck beschrieben, bleibt unklar, warum ein Softwaremodell bzw. ein Diagramm erstellt wurde und welche Funktion es hat. Dies kann negative Konsequenzen für die Qualitätsprüfung haben. Sollen beispielsweise die Diagrammtypen Klassendiagramm und Statechart für ein Softwaremodell modelliert werden und ihre Verwendungszwecke sind dem Qualitätsmanager nicht bekannt, dann nimmt er evtl. an, dass alle Operationen im Klassendiagramm auch im Statechart vorkommen sollen. Handelt es sich aber um ein Protokollstatechart, ist diese Forderung falsch.

Ist der Verwendungszweck nicht dokumentiert, kann es folglich zu Fehlinterpretationen bei der Erstellung des Qualitätsplans kommen. Zumindest ein fachlicher Verwendungszweck kann für jedes zu prüfende Softwaremodell angegeben werden. Ansonsten ist das Softwaremodell ohnehin nutzlos und das Projektteam verschwendet Ressourcen für die Erstellung und Bearbeitung, ohne dass ein Mehrwert produziert wird.

Zusammenfassung

Ein Modellzweck bzw. ein Diagrammzweck gibt an, wofür ein Softwaremodell bzw. Diagrammtyp in der Entwicklung genutzt wird. Ein Qualitätsmanager kennt u.U. die intendierten Verwendungszwecke nicht und trifft Annahmen, die zu falschen Qualitätszielen oder Qualitätsprüfungen in der Qualitätsplanung führen können. Deshalb ist die Dokumentation der wesentlichen Verwendungszwecke eines zu prüfenden Softwaremodells für die Qualitätsbewertung von Bedeutung.

3.6 Kontext von Softwaremodellen

In den vorigen Abschnitten haben wir die wichtigsten Besonderheiten von Softwaremodellen sowie ihre Einbettung in den modellbasierten Entwicklungsprozess untersucht und diverse Variationsmöglichkeiten festgestellt. Softwaremodelle können auf verschiedenen Modellierungssprachen basieren, mit verschiedenen Entwicklungsartefakten in Beziehung stehen, in mehreren Phasen des Entwicklungsprozesses erstellt und von verschiedenen Rollen für verschiedene Zwecke eingesetzt werden. Basierend auf diesen Ausführungen grenzen wir Softwaremodelle vom allgemeinen Modellbegriff (vgl. Def. 3.1) ab und definieren Softwaremodelle wie folgt:

Definition 3.3 (*Softwaremodell*)

Ein Softwaremodell ...

- *wird in der modellbasierten Softwareentwicklung eingesetzt,*
- *basiert auf einer formellen Syntax und einer Semantik, die beide durch eine Modellierungssprache definiert sind,*
- *wird häufig durch Diagramme visualisiert,*
- *hängt fachlich mit anderen Entwicklungsartefakten zusammen (z.B. Anforderungsspezifikation, Softwaremodell, Quellcode),*
- *abstrahiert von, verfeinert, beschreibt und/oder spezifiziert andere(n) Entwicklungsartefakte(n),*
- *wird für mindestens einen fachlichen Verwendungszweck erstellt und*
- *wird von einer Person, die eine Rolle wahrnimmt, von einem speziellen Standpunkt aus genutzt.*

Die Menge der wesentlichen Charakteristika, die sich durch die Verwendung eines Softwaremodells im Rahmen der Softwareentwicklung ergeben, bezeichnen wir als *Kontext des Softwaremodells*. Beispielsweise lässt sich der Kontext eines Softwaremodells durch die eingesetzte Modellierungssprache *UML 2.1.1* und die Diagrammtypen *Klassen- und Sequenzdiagramm* charakterisieren. Diesen Kontext nehmen wir

als gegeben an. Ihn anzupassen und zu verbessern ist Aufgabe des Prozessqualitätsmanagements und nicht Bestandteil unserer Zielsetzung.

Definition 3.4 (*Kontext von Softwaremodellen*)

Der Kontext eines Softwaremodells charakterisiert die Besonderheiten eines Softwaremodells und seine Einbettung in den modellbasierten Softwareentwicklungsprozess.

Softwaremodelle entstehen in einem bestimmten Kontext und dieser Kontext kann für verschiedene Softwaremodelle variieren. Der Kontext muss für die Qualitätsbewertung beachtet werden, da er einen erheblichen Einfluss auf Qualitätsziele und Qualitätsprüfungen aufweist. Plakativ lässt sich der Zusammenhang zwischen dem Kontext eines zu prüfenden Softwaremodells und dem entsprechenden Qualitätsplan wie folgt zusammenfassen: Es gibt nicht *die* modellbasierte Softwareentwicklung und nicht *das* Softwaremodell. Somit existiert auch nicht *der* Qualitätsplan. Diesen bedeutenden Einfluss des Kontextes auf die Qualitätsplanung bezeichnen wir als Kontextsensitivität.

Definition 3.5 (*Kontextsensitivität*)

Die Kontextsensitivität bezieht sich auf den Einfluss des Kontextes eines zu prüfenden Softwaremodells auf die entsprechende Qualitätsplanung.

Kapitel 4

Analytische Qualitätssicherung in der Softwareentwicklung

Nachdem wir in Kapitel 3 den Kontext von Softwaremodellen eingeführt haben, widmen wir uns in diesem Kapitel bestehenden Ansätzen zur analytischen Qualitätssicherung.

Zuerst stellen wir in Abschnitt 4.1 relevante Standards vor, die ausgereifte Konzepte zur analytischen Qualitätssicherung in der Softwareentwicklung spezifizieren. Prozessorientierte Standards in Abschnitt 4.1.1 nutzen wir zur Einordnung der analytischen Qualitätssicherung von Softwaremodellen in ein ganzheitliches Qualitätsmanagement für die Softwareentwicklung. In Abschnitt 4.1.2 führen wir einen Standard für Softwareproduktqualität ein, der uns in Form von *Qualitätsmodellen* eine Möglichkeit aufzeigt, Qualitätsziele zu dokumentieren. Standards zur Qualitätsprüfung thematisieren wir in Abschnitt 4.1.3. Diese Standards spezifizieren *Messungen* als das geeignete Verfahren, um Qualitätsziele zu überprüfen.

Standards sollen grundsätzlich durch eine Vielzahl an Unternehmen anwendbar sein. Sie definieren tendenziell eher Rahmenwerke und bleiben in einigen konkreten Aspekten vage. Insbesondere existieren keine Standards, die sich speziell auf die analytische Qualitätssicherung von Softwaremodellen beziehen. Deshalb ergänzen wir die Standards um einige konkrete Ansätze. Eine Reihe von Qualitätsmodellen für Modelle stellen wir in Abschnitt 4.2 vor. In Abschnitt 4.3 gehen wir auf Messungen ein, die auf Softwaremodellen basieren. Jeweils am Ende dieser beiden Abschnitte analysieren wir die vorgestellten Konzepte und bewerten ihren Nutzen für die Qualitätsplanung von Softwaremodellen anhand unserer in Kapitel 2 aufgestell-

ten Anforderungen. In Abschnitt 4.4 thematisieren wir kurz bestehende Arbeiten für die Qualitätssicherungsphasen *Interpretation von Prüfergebnissen* und *Verbesserung des Softwaremodells*. Abschließend betrachten wir in Abschnitt 4.5 die Verifikation und die Simulation als weitere Analyseverfahren, die auf Softwaremodelle angewandt werden können, denen eine mathematisch definierte Semantik zu Grunde liegt.

4.1 Standards

Standards stellen den Stand der Technik dar und basieren auf ausgereiften Lösungen, die sich gegenüber anderen Verfahrensweisen durchgesetzt haben. Deshalb sollten Standards, die sich auf die Softwareentwicklung im Allgemeinen beziehen, wenn möglich auch für die analytische Qualitätssicherung von Softwaremodellen im Speziellen eingesetzt werden.

Im Bereich der Softwareentwicklung sind eine ganze Reihe von Standards vorhanden, von denen sich allerdings kein einziger Standard ausschließlich auf Softwaremodelle bezieht. Stattdessen betrachten wir drei Gruppen von Standards, die sich zumindest teilweise auf die analytische Qualitätssicherung von Softwaremodellen übertragen lassen.

1. *Prozessorientierte Standards* legen Richtlinien für den Softwareentwicklungsprozess fest, die auch für die analytische Qualitätssicherung von Softwaremodellen gelten. Prozessorientierte Standards definieren, dass Qualitätsziele festgelegt, ihre Erfüllungsgrade gemessen und nicht akzeptable Abweichungen durch Verbesserungen des Produkts korrigiert werden müssen. Zudem bestätigen sie die Bedeutung einer frühen und kontinuierlichen Qualitätssicherung von Zwischenprodukten.

2. *Standards für Softwareproduktqualität* schreiben Qualitätsmodelle für die Dokumentation von Qualitätszielen vor. In der Entwicklung von Softwareprodukten stellt ein Softwaremodell entweder ein Zwischenprodukt oder eine ergänzende Dokumentation des Endprodukts dar. In beiden Fällen hängt die Qualität eines Softwareprodukts mit der Qualität eines Softwaremodells eng zusammen, so dass der Einsatz von Qualitätsmodellen für die Qualitätsbewertung von Softwaremodellen nahe liegt.

3. *Standards zur Qualitätsprüfung* spezifizieren Messungen als das geeignete Verfahren, um Qualitätsziele zu überprüfen, die an ein Entwicklungsartefakt gestellt werden. Diese Standards stellen den Zusammenhang zwischen Qualitätsmodellen und Messungen her, so dass quantitative und qualitative Aussagen über die Qualität eines Entwicklungsartefakts möglich sind.

Die Standards für Softwareproduktqualität und zur Qualitätsprüfung in der Softwareentwicklung fließen in den übergeordneten ISO/IEC 25000 Standard *Software product Quality Requirements and Evaluation (SQuaRE)* ein [89] und befinden sich zur Zeit in der Überarbeitung. Der aktuelle Überarbeitungsstand kann der Webseite `http://www.iso.org/` entnommen werden.

4.1.1 Prozessorientierte Standards

Es gibt eine Reihe prozessorientierter Standards, die für Entwicklungsprozesse von Softwareherstellern von Bedeutung sind. Mit der Umsetzung prozessorientierter Standards verfolgt ein Softwarehersteller meherere Ziele:

- Entwicklungsprozess weiterentwickeln
- Sicherstellung von Qualitätszielen
- Risikoabschätzung verbessern
- Vertrauen beim Kunden schaffen
- Sorgfaltspflicht erfüllen und rechtliche Basis stärken

Wir stellen zum einen den wichtigsten prozessorientierten Standard für Softwareentwicklungsprojekte vor, bei denen Bundesbehörden oder die Bundeswehr als Auftraggeber fungieren: *V-Modell XT*. Zum Zweiten fassen wir zwei der wichtigsten international verbreiteten prozessorientierten Standards zusammen: *DIN EN ISO 9001* und *CMMI*. Wir verzichten auf eine Beschreibung weiterer Standards wie z.B. die ISO/IEC 15504 Serie (SPICE) [88] oder Six Sigma [167], da sie für die analytische Qualitätssicherung keine wesentlich anderen Erkenntnisse enthalten.

Unsere nachfolgenden Ausführungen zu prozessorientierten Standards konzentrieren sich auf Richtlinien für die analytische Qualitätssicherung konstruierter Zwischen- und Endprodukte.

V-Modell XT

Das V-Modell XT [95] ist als Leitfaden zum Planen und Durchführen von Entwicklungsprojekten unter Berücksichtigung des gesamten Systemlebenszyklus konzipiert. Es beschreibt zu erstellende Ergebnisse, Vorgehensweisen und Verantwortlichkeiten. Das V-Modell XT stellt den Entwicklungsstandard für IT-Systeme im gesamten zivilen und militärischen Bundesbereich dar und ist für Auftragnehmer verbindlich.

Für die Qualitätssicherung wird gefordert, dass "die Qualität des Projektergebnisses (...) sowohl konstruktiv als auch analytisch in der Entwicklung sicher zu stellen (ist)" [95]. Die Qualitätssicherung ist ein zentraler Vorgehensbaustein im V-Modell XT und muss in allen Projekten eine Anwendung finden.

DIN EN ISO 9001

Die DIN EN ISO 9001 [48] thematisiert die Anforderungen an das Qualitätsmanagement einer Organisation, damit sie ihre Fähigkeit zur Erfüllung der Kundenforderungen und zur Erreichung fehlerfreier Produkte nachvollziehbar für Dritte nachweisen kann [176]. Eine Zertifizierung entsprechend der Norm wird oft als notwendig für softwareentwickelnde Unternehmen angesehen [107].

In der DIN EN ISO 9001 wird für die Dokumentation eines Qualitätsmanagementsystems gefordert, dass Qualitätsziele festgelegt sind. Zusätzlich muss die Organisation die Mess-, Analyse- und Verbesserungsprozesse planen und verwirklichen, damit der Erfüllungsgrad der Qualitätsziele bestimmt und bewertet werden kann und im Falle nicht erfüllter Qualitätsziele Verbesserungen angestoßen werden können. Da die Norm als branchenübergreifender Leitfaden konzipiert ist, legt sie nicht die praktische Umsetzung der einzelnen Tätigkeiten fest.

CMMI

Capability Maturity Model Integration (CMMI) [156] ist ein Ansatz zur Prozessverbesserung in Unternehmen. Durch die verbesserten Prozesse soll unter anderem die Produktqualität positiv beeinflusst werden. Der CMMI-Ansatz strukturiert sich nach den Anwendungsgebieten Produkteinkauf, Serviceerbringung und Produkentwicklung [157]. Die Softwareentwicklung ist dem zuletzt genannten Anwendungsgebiet zuzuordnen.

Der aktuelle Prozesszustand einer Organisation wird in Form von *Maturity Levels*

angegeben und bestimmt sich durch die Kombination der separaten Bewertungen der einzelnen Schlüsselbereiche, die anhand der *Capability Levels* eingestuft sind. “Eine in der Praxis häufig eingesetzte Bewertungsmethode ist die Standard CMMI Assessment Method for Process Improvement (SCAMPI)” [81]. CMMI definiert insgesamt 5 verschiedene Maturity Levels:

Maturity Levels

- *Maturity Level 1: Initial:* Initiale Prozesse sind häufig ad-hoc und chaotisch. Sie sind durch die Bewältigung von Ausnahmesituationen geprägt und werden von einem außergewöhnlichem Problem zum nächsten getrieben. Zeitpläne und Budget werden regelmäßig überschritten.

- *Maturity Level 2: Managed:* Das Projektmanagement kontrolliert den Softwareentwicklungsprozess basierend auf Erfahrung. Kosten und Zeitpläne werden erfolgreich überwacht und entstehende Probleme identifiziert. Allerdings hat die Organisation große Probleme, neue Herausforderungen zu bewältigen. Bei diesem Reifegrad kann eine Organisation ähnliche Projekte erfolgreich durchführen.

- *Maturity Level 3: Defined:* Für die gesamte Organisation gibt es einen Standard-Softwareentwicklungsprozess, der klar beschrieben ist und auf besondere Gegebenheiten eines Projekts zugeschnitten wird. Das Projektteam arbeitet effizient. Der Projektleiter kontrolliert Kosten und Zeitpläne und Qualitätsdaten werden für Kontrolltätigkeiten erhoben.

- *Maturity Level 4: Quantitatively Managed:* Quantitativ messbare Qualitätsziele werden für das Softwareprodukt und den Entwicklungsprozess definiert, erhoben und in einer organisationsweiten Erfahrungsdatenbank gesichert. Trends in Bezug auf Prozessfortschritt und Produktqualität werden frühzeitig erkannt. Ursachen für außergewöhnliche Ereignisse können schnell identifiziert und behandelt werden. Dadurch kann die Qualität des Softwareprodukts garantiert werden.

- *Maturity Level 5: Optimizing:* Die Organisation zeichnet sich durch eine kontinuierliche Verbesserung des Entwicklungsprozesses aus. Für neu einzuführende Technologien und Prozessänderungen können Kosten-Nutzen-

Analysen erstellt werden. Schwachstellen in Projekten werden analysiert und durch Erfahrungsberichte für andere Projekte vermieden.

Fazit

Im V-Modell XT stellt die analytische Qualitätssicherung einen zentralen Vorgehensbaustein dar. Der DIN EN ISO 9001 Standard fordert die Festlegung von Qualitätszielen sowie die Planung und Umsetzung von Mess-, Analyse- und Verbesserungsprozessen zur Zielerreichung. Bereits ab dem CMMI Maturity Level 3 muss ein Softwarehersteller Qualitätsdaten für Zwischenprodukte erheben. Unsere Auswahl prozessorientierter Standards bestätigt und verdeutlicht die Bedeutung der analytischen Qualitätssicherung von Softwaremodellen für Softwarehersteller.

4.1.2 Standards für Softwareproduktqualität

Für die Dokumentation von Qualitätszielen, die an Softwareprodukte gestellt werden, besteht ein Standard von herausragender Bedeutung:

ISO/IEC 9126-1

ISO/IEC 9126-1 [85] ist ein internationaler Standard für die Evaluation von Softwareprodukten. Der Standard definiert den Begriff *Softwareproduktqualität* wie folgt:

Definition 4.1 (*Softwareproduktqualität*)

Die Gesamtheit von Charakteristiken eines Softwareprodukts, die sich auf dessen Eignung beziehen, festgelegte oder vorausgesetzte Erfordernisse zu erfüllen.

Der Standard fordert, dass "software product quality should be evaluated using a defined quality model" [85] und verfeinert das definierte Qualitätsverständnis für Softwareproduktqualität durch *Qualitätsmodelle* (vgl. Def. 4.2) erstens für *interne und externe Qualität* sowie zweitens für *Gebrauchsqualität*.

Definition 4.2 (*Qualitätsmodell*)

Ein Qualitätsmodell ist eine Menge von Qualitätscharakteristiken und Beziehungen zwischen diesen.

In einem Qualitätsmodell werden drei wesentliche Konzepte verknüpft: Gesamtqualität, Qualitätscharakteristik (vgl. Def. 4.3) und Qualitätsattribut (vgl. Def. 4.4).

Definition 4.3 (*Qualitätscharakteristik*)

Eine Qualitätscharakteristik ist eine Kategorie von Qualitätsattributen, die sich auf Softwareproduktqualität bezieht. Qualitätscharakteristiken können in mehrere Ebenen von Qualitätsteilcharakteristiken und schließlich in Qualitätsattribute verfeinert werden.

Definition 4.4 (*Qualitätsattribut*)

Ein Qualitätsattribut ist eine direkt messbare, inhärente Eigenschaft eines Objekts.

Bestehende Qualitätsmodelle können anhand dieser drei Ebenen Gesamtqualität, Qualitätscharakteristik und Qualitätsattribut strukturiert werden (siehe Abb. 4.1 oben). Das in Abb. 4.1 dargestellte Qualitätsmodell für interne und externe Qualität enthält Qualitätscharakteristiken und Qualitätsteilcharakteristiken. Die Ebene der Qualitätsattribute ist nicht beschrieben. Die entsprechenden deutschsprachigen Begriffe der Qualitätscharakteristiken finden sich in der Deutschen Industrie Norm (DIN) 66272 [47].

Die vorgeschlagenen Qualitätscharakteristiken sind gemäß dem Standard grundsätzlich auf jedes Softwareprodukt anwendbar. Folglich erhebt der Standard den Anspruch, ein *generisches Qualitätsmodell* zu spezifizieren. Allerdings ist Qualität immer relativ zu gegebenen Anforderungen zu sehen. Qualität ist somit nichts Absolutes. In [57] wird bemerkt, dass das ISO Qualitätsmodell auf die Besonderheiten eines Projekts angepasst werden muss und dies auch *Kürzungen* umfasst.

Bei der Anwendung des ISO Qualitätsmodells zur analytischen Qualitätssicherung in konkreten Entwicklungsprojekten müssen die Qualitätscharakteristiken und Qualitätsteilcharakteristiken weiter in Qualitätsattribute verfeinert werden, damit konkrete Qualitätsprüfungen identifiziert und angewandt werden können. Das systematische Verfeinern von Qualitätscharakteristiken über Qualitätsattribute hin zu Qualitätsprüfungen wird als *Operationalisierung* oder umgangssprachlich auch als *messbar machen* von Qualität bezeichnet.

Der ISO/IEC 9126-1 Standard beschreibt zudem den Zusammenhang zwischen einerseits der Qualität eines Softwaremodells in der Rolle eines Zwischenprodukts als

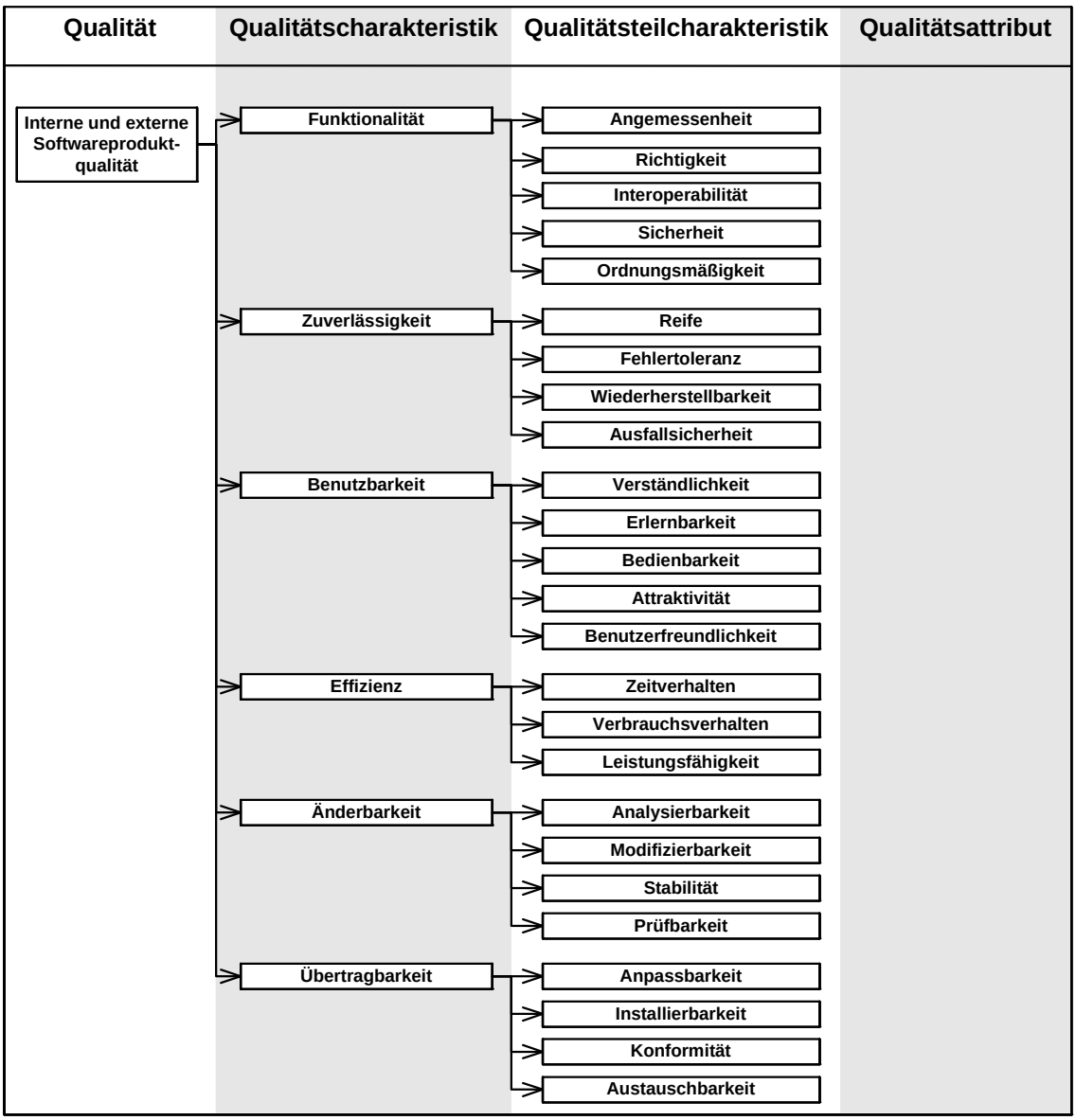


Abbildung 4.1: ISO Qualitätsmodell für interne und externe Qualität

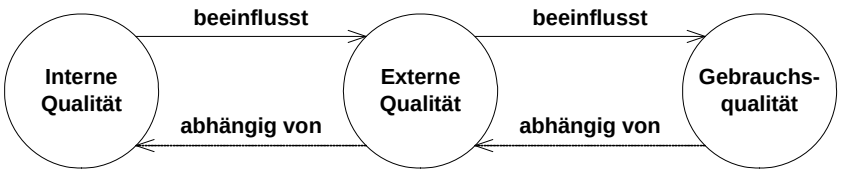


Abbildung 4.2: Einfluss der internen Qualität (in Anlehnung an [85])

auch in der Rolle eines ergänzenden Dokuments und andererseits Softwarequalität. Modellqualität zählt zur internen Qualität von Software und beeinflusst die externe Qualität und die Gebrauchsqualität (vgl. Abb. 4.2). Die interne Qualität ist die Gesamtheit der Qualitätscharakteristiken eines Softwareprodukts aus der internen Sicht. Interne Sichten werden durch Projektbeteiligte wie Modellierer, Entwickler, Architekten oder Reviewer genutzt. Zur internen Sicht eines Softwareprodukts gehört neben einem Softwaremodell unter anderem auch der Quellcode.

Fazit

Qualitätsmodelle ermöglichen die strukturierte *Dokumentation von Qualitätszielen*. Qualitätsmodelle können systematisch verfeinert werden, so dass die Gesamtqualität *operationalisiert* wird. Qualität ist immer relativ zu gegebenen Anforderungen zu sehen und nichts Absolutes. Deshalb sollten Qualitätsmodelle auf das jeweilige Projekt hin zugeschnitten werden. Sind zudem alle Qualitätscharakteristiken und Qualitätsattribute definiert, dokumentiert ein Qualitätsmodell ein *domänenspezifisches Qualitätsverständnis*. Die Qualität eines Softwaremodells in der Rolle eines Zwischenprodukts bzw. eines ergänzendes Dokuments wird der *internen Qualität* eines Softwareprodukts zugeordnet.

4.1.3 Standards zur Qualitätsprüfung

Nachdem wir im vorigen Abschnitt einen Standard vorgestellt haben, der die Bedeutung von Qualitätsmodellen zur Dokumentation von Qualitätszielen hervorhebt, wenden wir uns nun zwei Standards zu, die die Überprüfung der Qualitätsziele thematisieren. Der ISO/IEC 15939 Standard beschreiben einen generisches Messprozess zur Evaluierung von Software basierend auf Kennzahlen. Der ISO/IEC 9126-3 Standard ist auf die Bewertung der internen Qualität von Software mittels Metriken

spezialisiert. Wir führen im Folgenden beide Standards ein und stellen den Zusammenhang zwischen Kennzahlen und Metriken her.

ISO/IEC 15939

ISO/IEC 15939 [86] spezifiziert den Software-Messprozess (engl.: software measurement process). "Software measurement is (...) a key discipline in evaluating the quality of software products (...). The purpose of the software measurement process (...) is to collect, analyse, and report data relating to the products (hier: developed software products; Anm. d. Verf.)" [86].

Der Software-Messprozess ist an den Demingkreis [44] angelehnt, der auch unter dem Begriff *Do-Plan-Check-Act (DPCA) Cycle* bekannt ist. Der Messprozess definiert einen Qualitätskreislauf, der eine kontinuierliche Produkt- und Prozessverbesserung ermöglicht. Er ist in vier Aktivitäten gegliedert, die iterativ durchlaufen werden (siehe Abb. 4.3).

Während der *Planung* werden Informationsbedürfnisse identifiziert, die dafür relevanten Messungen und Soll-Werte festgelegt. Zur Feststellung des Ist-Zustandes werden anschließend diese *Messungen durchgeführt*, die Ergebnisse mit Soll-Werten verglichen und Abweichungen analysiert. Im nächsten Schritt *interpretieren und bewerten* Messnutzer die Messergebnisse, um abschließend entweder notwendige *Produktverbesserungen* zu veranlassen oder um den Qualitätskreislauf vorerst zu verlassen.

Für eine gezielte Qualitätsmessung sind dabei die Informationsbedürfnisse der Projektbeteiligten von großer Bedeutung. Produktverbesserungen werden ausschließlich von Entscheidungsträgern und nicht von Systemen getroffen. Dafür benötigen sie Berichte, die ihre Informationsbedürfnisse berücksichtigen und befriedigen. Diese Berichte werden in [86] als Informationsprodukte bezeichnet.

Das Measurement Information Model (MIM) der ISO/IEC 15939 stellt den Zusammenhang zwischen Informationsbedürfnissen und Informationsprodukten her (siehe Abb. 4.4). Es beschreibt generisch, wie relevante Attribute quantifiziert und in Indikatoren überführt werden können, die eine fundierte Grundlage für Entscheidungsfindungen bieten.

Die Messung eines *Indikators* erfolgt in 3 Schritten: Zuerst wird mittels Messmethoden die Quantifizierung eines Attributs durchgeführt. Diese *Basiskennzahlen* können dann zu *abgeleiteten Kennzahlen* kombiniert werden. Im Anschluss verknüpft

Abbildung 4.3: Qualitätskreislauf

eine Analyse die Basis- und/oder abgeleiteten Kennzahlen und vergleicht sie mit numerischen Grenzen, Zielwerten o.ä., um Werte für Indikatoren zu produzieren. Hierbei muss beachtet werden, dass die Analyse im Vorfeld geplant sein muss und nicht erst nachdem die Messergebnisse bekannt sind.

Der Standard unterscheidet zwei Arten von Messmethoden. *Subjektive Messungen* benötigen das menschliche Urteilsvermögen. Sie lassen sich nicht vollständig automatisieren. Dagegen basieren *objektive Messungen* auf numerischen Regeln und können von Werkzeugen durchgeführt werden.
Die wichtigsten Begriffe des MIM sind nachfolgend kompakt zusammengefasst:

Wichtige Begriffe des MIM

- Eine *Kennzahl* ist eine Variable, der als Ergebnis einer Messung ein Wert zugewiesen wird. Basiskennzahlen, abgeleitete Kennzahlen und Indikatoren sind Kennzahlen.

- Eine *Messung* ist eine Menge von Operationen mit der Zielsetzung, einen Wert für eine Kennzahl hinsichtlich einer spezifizierten Skala zu bestimmen. Messmethoden, Messfunktionen und Analysen sind Messungen.

- Eine *Basiskennzahl* ist eine Kennzahl, die sich auf ein Attribut bezieht, und die entsprechende Messmethode zur Quantifizierung. Eine Basiskennzahl ist

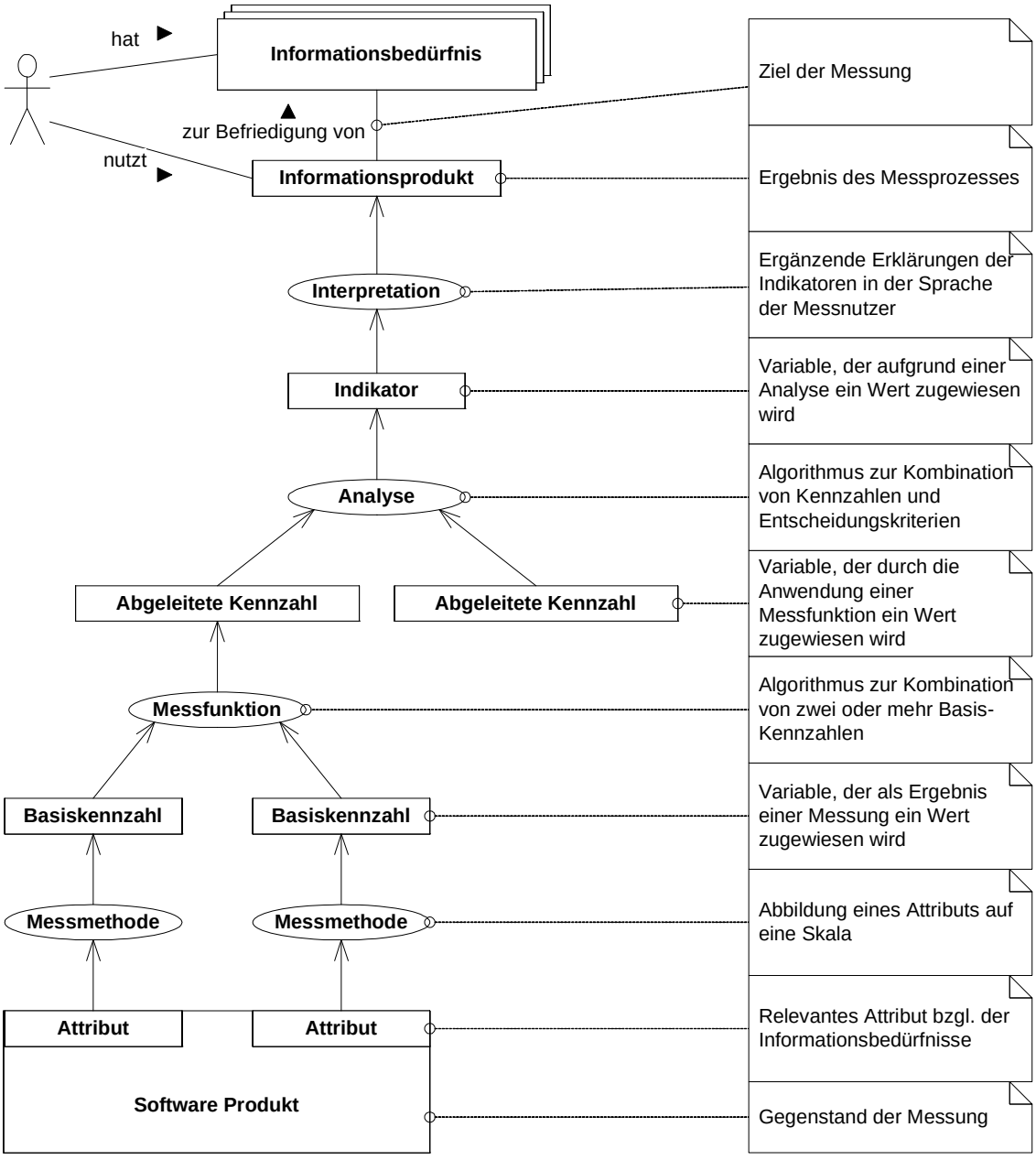


Abbildung 4.4: Measurement Information Model (MIM)

funktional unabhängig von anderen Kennzahlen.

- Eine *Messmethode* ist eine Menge von Operationen mit der Zielsetzung einen Wert für eine Basiskennzahl zu bestimmen. Es wird zwischen zwei Arten von Messmethoden unterschieden:
  - *Subjektive Messmethoden* benötigen das menschliche Urteilsvermögen.
  - *Objektive Messmethoden* basieren auf numerischen Regeln.
- Eine *abgeleitete Kennzahl* ist als Funktion von zwei oder mehr Basiskennzahlen oder abgeleiteten Kennzahlen definiert.
- Eine *Messfunktion* ist eine Kalkulation, die zwei oder mehr Basiskennzahlen oder abgeleitete Kennzahlen kombiniert.
- Ein *Indikator* ist eine Kennzahl, der eine Beurteilung eines spezifizierten Attributs ermöglicht, und durch eine Analyse bestimmt wird.
- Eine *Analyse* ist eine Kalkulation, die eine oder mehrere Basiskennzahlen oder abgeleitete Kennzahlen kombiniert und mit Entscheidungskriterien wie Grenzwerten, Zielwerten, o.ä. vergleicht.
- Eine *Skala* ist eine geordnete Menge von Werten oder eine Menge von Kategorien.

ISO/IEC 9126-3

ISO/IEC 9126-3 [87] stellt eine Reihe von internen Metriken für die Messung der internen Qualität von Softwareprodukten bereit. Der Standard bezieht sich auf das in der ISO/IEC 9126-1 vorgestellte Qualitätsmodell für interne und externe Qualität (vgl. dazu auch Abb. 4.1). "The internal metrics may be applied to a non-executable software product during its development stages (such as request for proposal, requirements definition, design specification or source code)" [87]. Es erfolgt also keine Ausführung eines Softwareprodukts bei der Erhebung von internen Metriken.

"The metrics listed in this Technical Report are not intended to be an exhaustive set. Developers, evaluators, quality managers and acquirers (...) may also modify the metrics or use metrics which are not included here" [87]. Demzufolge handelt es

sich bei der gegebenen Metriksammlung um ein Nachschlagewerk für die Auswahl von Metriken, deren einzelnen Metriken nicht verbindlich angewandt werden müssen.

Die Dokumentation einer Metrik folgt einem festen Schema. Zu jeder Metrik wird ein eindeutiger Name, ihr Zweck in Form einer Frage, eine Messbeschreibung, Interpretationsrichtlinien für mögliche Ergebnisse, die Skala und benötigte Eingabedokumente angegeben.

Zwischen Metriken und Kennzahlen besteht ein direkter Zusammenhang:

Definition 4.5 (*Metrik*)

Eine Metrik bezeichnet eine Kennzahl und das Verfahren zur Messung dieser quantifizierbaren Größe. Folglich ist eine Metrik entweder (1) eine Basiskennzahl mit assoziierter Messmethode oder (2) eine abgeleitete Kennzahl assoziiert mit einer Messfunktion und benötigten Messmethoden.

Analog wie im MIM (siehe dazu Abb. 4.4) für Basiskennzahlen und abgeleitete Kennzahlen beschrieben, eignen sich Metriken auch als Grundlage für Indikatoren.

Fazit

Die Erfüllung von Qualitätszielen kann durch einen Qualitätskreislauf realisiert werden. Ein Qualitätskreislauf ermöglicht die analytische Überprüfung von Qualitätszielen und die kontinuierliche Produktverbesserung. Er besteht aus Mess-Planung, Mess-Durchführung, Interpretation der Messergebnisse und Verbesserung der Produktqualität. Während der Planung der Messungen werden zu bestimmende Basiskennzahlen, abgeleitete Kennzahlen und Indikatoren festgelegt. Die Kombination von Basiskennzahl und Messmethode sowie die Kombination von abgeleiteter Kennzahl und Messfunktion wird auch unter dem Begriff Metrik subsumiert. Einige Metriken benötigen das menschliche Urteilsvermögen und sind subjektiv. Andere Metriken lassen sich durch numerische Regeln beschreiben und können objektiv erhoben werden.

Zusammenfassung

Standards spiegeln den ausgereiften Stand der Technik wider und sollten für die Qualitätssicherung von Softwaremodellen unbedingt beachtet werden. Für die Qualitätssicherung müssen Qualitätsziele festgelegt, ihr Erfüllungsgrad analytisch überprüft und nicht akzeptable Abweichungen durch Produktverbesserungen korrigiert werden. Qualitätsmodelle eignen sich für die Dokumentation von Qualitätszielen und zur Definition eines Qualitätsverständnisses. Die analytische Überprüfung von Qualitätszielen und die kontinuierliche Produktverbesserung zur Erfüllung der Qualitätsziele kann durch einen Qualitätskreislauf realisiert werden.

4.2 Qualitätsmodelle

Der zuvor in Abschnitt 4.1.2 vorgestellte ISO/IEC 9126-1 Standard spezifiziert Qualitätsmodelle als geeignetes Konzept zur Beschreibung von Qualitätszielen. Deshalb bieten wir in diesem Abschnitt eine vertiefende Betrachtung von Qualitätsmodellen an. Weil sich der Standard primär auf Softwarequalität bezieht, präsentieren wir im Abschnitt 4.2.1 eine Reihe weiterer *generischer Qualitätsmodelle*, die sich speziell auf Modelle beziehen und bewerten im Anschluss den Nutzen von generischen Qualitätsmodellen für unsere eingangs formulierte Zielsetzung. Allerdings fehlen für generische Qualitätsmodelle geeignete Vorgehen, um sie auf gegebene Projekte hin anzupassen. Deshalb stellen wir in Abschnitt 4.2.2 einen Ansatz vor, mit dem vollständig individuelle Qualitätsmodelle hergeleitet werden können und bewerten anschließend auch seinen Nutzen.

Qualitätsmodelle stellen einen weit verbreitenden Ansatz dar, um Qualität nach dem Divide & Conquer Prinzip in mehrere Bestandteile zu gliedern und auf diese Weise zu präzisieren. Ein allgemeiner Qualitätsbegriff wird durch das Ableiten von Unterbegriffen sukzessive verfeinert und auf diese Weise operationalisiert [176]. Durch den Einsatz eines Qualitätsmodells werden folgende Ziele verfolgt:

- Dokumentation von Qualitätszielen
- Definition eines einheitlichen Qualitätsverständnisses
- Operationalisierung von Qualität

Qualitätsmodelle sind ein Hilfsmittel zur Messung und Interpretation von Produktqualität. Sind Qualitätsmodelle bis zur Ebene der Qualitätsattribute verfeinert, dann lassen sich diese durch Metriken quantifizieren und durch Indikatoren qualifizieren.

Qualitätsmodelle sind eine etablierte Technik. Seitdem Boehm 1976 [21] und McCall 1977 [113] die ersten Qualitätsmodelle für Softwareproduktqualität veröffentlicht haben, sind einige Weiterentwicklungen vorgeschlagen worden (z.B. [24, 166]). Diese Forschungsergebnisse führten Anfang der 90er Jahre zum ISO/IEC 9126 Standard, den wir in einer aktuellen Version in Abschnitt 4.1.2 eingeführt haben.

Die Terminologie der ISO/IEC 9126 unterscheidet zwischen Qualitätscharakteristiken und Qualitätsattributen. Qualitätscharakteristiken können in mehrere Ebenen von Qualitätsteilcharakteristiken und schließlich in Qualitätsattribute verfeinert werden. Qualitätscharakteristiken werden auch als Qualitätsfaktoren oder Qualitätsmerkmale und Qualitätsattribute auch als Qualitätskriterien bezeichnet [21, 82]. Wir werden die von den ISO/IEC Standards definierten Begriffe *Qualitätscharakteristik* und *Qualitätsattribut* konsequent weiterverwenden.

Eine strikte Abgrenzung der Begriffe *Qualitätscharakteristik* und *Qualitätsattribut* ist schwierig, da es nicht möglich ist, die *richtige* Granularität für Qualitätsmodelle allgemeingültig und exakt zu bestimmen. Die Definition des Begriffs *Qualitätsattribut* erlaubt große Freiheitsgrade (vgl. Def. 4.4 in Abschnitt 4.1.2). So lassen sich Beispiele finden, in denen *Korrektheit* einerseits ein Qualitätsattribut [145] und andererseits eine Qualitätscharakteristik ist, die wiederum durch *syntaktische Korrektheit* und *semantische Korrektheit* spezialisiert wird [13].

Diese und andere Unterschiede in Qualitätsmodellen liegen darin begründet, dass Qualität relativ zu projektspezifischen Anforderungen ist. Zudem ist Qualität inhärent subjektiv. Individuelle Unterschiede im Qualitätsverständnis wirken sich auf die verwendeten Begriffe und Definitionen von Qualitätscharakteristiken und Qualitätsattributen aus. Aus diesen Gründen gibt es nicht das allgemeingültige Qualitätsmodell [173]. Vielmehr müssen projektspezifische Qualitätsmodelle erstellt werden. Dafür lassen sich im Wesentlichen zwei Ansätze differenzieren:

1. *Generische Qualitätsmodelle* definieren ein Rahmenwerk, auf dessen Grundlage ein projektspezifisches Qualitätsmodell gestaltet wird. Für die Anpassung von Qualitätsmodellen ist auch eine Vorgehensweise publiziert worden, die allgemein anwendbar sein soll. Bei Ansätzen für generische Qualitätsmodelle

unterscheiden wir zwei Tendenzen:

- Generische *high-level* Qualitätsmodelle bestehen überwiegend aus Qualitätscharakteristiken sowie Qualitätsteilcharakteristiken und vernachlässigen Qualitätsattribute. Bevor diese Qualitätsmodelle in einem konkreten Projekt für die analytische Qualitätssicherung angewendet werden können, muss die Ebene der Qualitätsattribute erweitert werden.
- Generische *low-level* Qualitätsmodelle enthalten darüber hinaus eine Vielzahl an Qualitätsattributen. Diese Qualitätsmodelle müssen für Projekte überwiegend zugeschnitten und kaum verfeinert werden.

2. *Vollständig individuelle Qualitätsmodelle* lassen sich mit Hilfe zielgetriebener Ansätze ableiten. Zielgetriebene Ansätze beschreiben ein Vorgehen für die systematische Erstellung von Qualitätsmodellen unabhängig von bestehenden Qualitätsmodellen.

Das im ISO/IEC 9126-1 Standard definierte *generische Qualitätsmodell* für die interne und externe Qualität eines Softwareprodukts haben wir bereits in Abschnitt 4.1.2 vorgestellt. Das Qualitätsmodell ist ausgereift und beinhaltet einen Großteil der zuvor in [21, 113] vorgeschlagenen Qualitätscharakteristiken. Obwohl sich die Qualität von Softwaremodellen auf die interne Qualität von Software bezieht, reicht das ISO Qualitätsmodell für Modellqualität nicht aus. Besondere Qualitätsanforderungen an Softwaremodelle wie *Konsistenz zwischen Diagrammen* sind nicht berücksichtigt. Aus diesem Grund stellen wir im folgenden Abschnitt 4.2.1 einige generische Qualitätsmodelle für Modellqualität vor.

4.2.1 Generische Qualitätsmodelle

Wir haben bei unserer Auswahl generischer Qualitätsmodelle darauf geachtet, dass die Ebene der Qualitätscharakteristiken für die Qualitätsmodelle ausreichend präzise definiert ist. Ansonsten ist es nicht möglich, die Bedeutung von zentralen Begriffen wie Komplexität, Wartbarkeit oder Korrektheit nachzuvollziehen.

Die Qualitätsmodelle klassifizieren wir anhand folgender Merkmale:

- Die *Detaillierung* des Qualitätsmodells gibt Aufschluss darüber, ob es sich in der Tendenz eher um ein high-level oder low-level Qualitätsmodell handelt.

- Generische Qualitätsmodelle sind oftmals auf bestimmte *Modelltypen* wie Entwurfs- oder Architekturmodell spezialisiert und überwiegend für diesen Modelltyp sinnvoll.

- Einige Qualitätsmodelle schlagen Messungen für die enthaltenen Qualitätsattribute vor. Die *Operationalisierung* gibt den Umfang der vorgeschlagenen Messungen an.

- Qualitätsattribute und assoziierte Messungen können auf die Besonderheiten einer *Modellierungssprache* wie UML oder ERM abgestimmt sein. Falls ein Qualitätsmodell hauptsächlich für eine bestimmte Modellierungssprache entwickelt wurde, dokumentieren wir diesen Sachverhalt. Die Einschränkung auf eine Modellierungssprache reduziert das Potential eines Qualitätsmodells, auch auf andere Softwaremodelle mit abweichender Modellierungssprache übertragbar zu sein.

Wir fassen die Qualitätsmodelle kurz zusammen. Wir werden jedoch die Vielzahl an Qualitätscharakteristiken und Qualitätsattributen nicht erklären und verweisen stattdessen auf die referenzierte Literatur.

Indem wir mehrere verschiedene Qualitätsmodelle für Modellqualität vorstellen, können wir (1.) die große Bandbreite an Qualitätscharakteristiken und Qualitätsattributen und (2.) die enorme Variabilität zwischen den einzelnen Qualitätsmodellen nachweisen.

Bevor wir auf die generischen Qualitätsmodelle eingehen, führen wir die entsprechende Vorgehensweise für die Anpassung von Qualitätsmodellen ein.

Software QUality In Development (SQUID)

Der Software QUality In Development (SQUID) Ansatz beschreibt ein Vorgehen zur Anpassung von Qualitätsmodellen [93]. Das SQUID-Verfahren geht von einem organisationsweiten Standard-Qualitätsmodell aus. Zusätzlich verlangt der Ansatz eine Beschreibung des benötigten Produktverhaltens in einer Qualitätsanforderungsspezifikation. Das weitere Vorgehen ist kurz beschrieben:

1. “identifying any elements in the standard that are not relevant to the particular product and removing them from the product quality model;

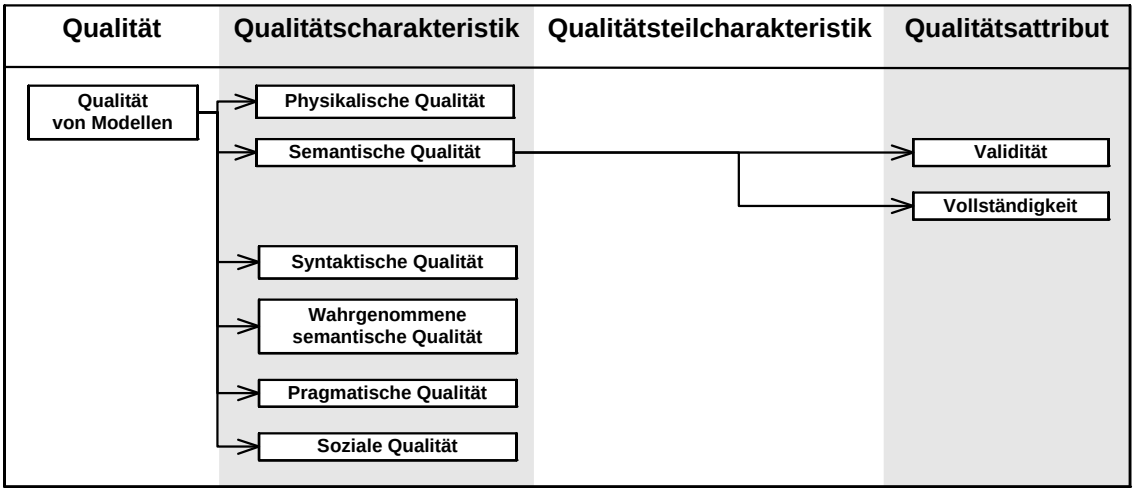


Abbildung 4.5: Qualitätsmodell für Modelle im Allgemeinen

Merkmal	**Ausprägung**
Detaillierung	high-level
Modelltyp	alle
Operationalisierung	keine
Modellierungssprache	alle

Tabelle 4.1: Klassifikation des Qualitätsmodells für Modelle im Allgemeinen

2. identifying any quality characteristics of the product that are not included in the standard and including them in the product quality model" [93].

Die Veröffentlichung enthält eine kleine Fallstudie. Darüber hinaus bietet der Ansatz leider keine weiteren Details für eine erfolgreiche Anwendung.

Qualitätsmodell für Modelle im Allgemeinen

Die Arbeit von Krogstie et al. [97] befasst sich mit der Qualität von Modellen im Allgemeinen und kann auf keinen bestimmten Modelltyp eingegrenzt werden.

Das entwickelte Qualitätsmodell (siehe Abb. 4.5) lässt sich sowohl auf die Qualität von natürlichsprachlichen Softwareanforderungsspezifikationen [96], auf die Qualität von Informationsmodellen [123] und auch auf die Qualität von UML-Entwurfsmodellen übertragen [102].

Das Qualitätsmodell umfasst kaum Qualitätsattribute und ist demzufolge *high-*

Merkmal	Ausprägung
Detaillierung	high-level
Modelltyp	Informationsmodell
Operationalisierung	niedrig
Modellierungssprache	alle

Tabelle 4.2: Klassifikation des Qualitätsmodells GOM II

level. Folglich werden auch keine Messungen für Qualitätsattribute empfohlen. Tabelle 4.1 fasst unsere Klassifikation dieses Qualitätsmodells zusammen.

Grundsätze ordnungsmäßiger Modellierung I und II (GOM)

Die Arbeiten zu den Grundsätzen ordnungsmäßiger Modellierung I [14] und die Weiterentwicklung zu GOM II von Schütte [155, 153, 154] befassen sich mit der Qualität von Informationsmodellen und haben im deutschsprachigen Raum eine gewisse Bekanntheit erlangt.

GOM I und II basieren auf zentralen Grundsätzen, die u.a. aus den Grundsätzen ordnungsmäßiger Buchführung hergeleitet sind. Wir ordnen diese Grundsätze der Ebene der Qualitätscharakteristiken zu (siehe Abb. 4.6).

Die GOM Ansätze sind unabhängig vom Aufbau der Modellierungssprache und der Anwendungsdomäne. Beispielsweise zählt der *systematische Aufbau* von Informationsmodellen zu den Modellierungsgrundsätzen. Diese Forderung wird in der UML bereits durch ihre Sprachdefinition sichergestellt.

In den GOM Ansätzen vermischt sich die Bewertung des Produkts *Informationsmodell* mit der Bewertung des Prozesses der *Modellerstellung* (vgl. dazu den Grundsatz der Spracheignung).

Inwiefern die angegebenen Verfeinerungen der Qualitätscharakteristiken eher Qualitätsteilcharakteristiken oder doch Qualitätsattribute sind, für die schließlich Messungen angegeben werden können, ist für uns kaum zu beurteilen. In der zur Verfügung stehenden Literatur werden lediglich einige Messungen zur Layoutgestaltung erwähnt. Die Mehrzahl der Qualitätscharakteristiken bleibt jedoch eher abstrakt. Z.B. bestimmt sich das für Schütte zentrale Bewertungskriterium der *Konstruktionsadäquanz* nur durch den Konsens der Modellnutzer und ist in dieser Form kaum zu messen. Deshalb stufen wir das GOM Qualitätsmodell in der Tendenz als high-level ein. Tabelle 4.2 fasst unsere Klassifikation dieses Qualitätsmodells zusammen.

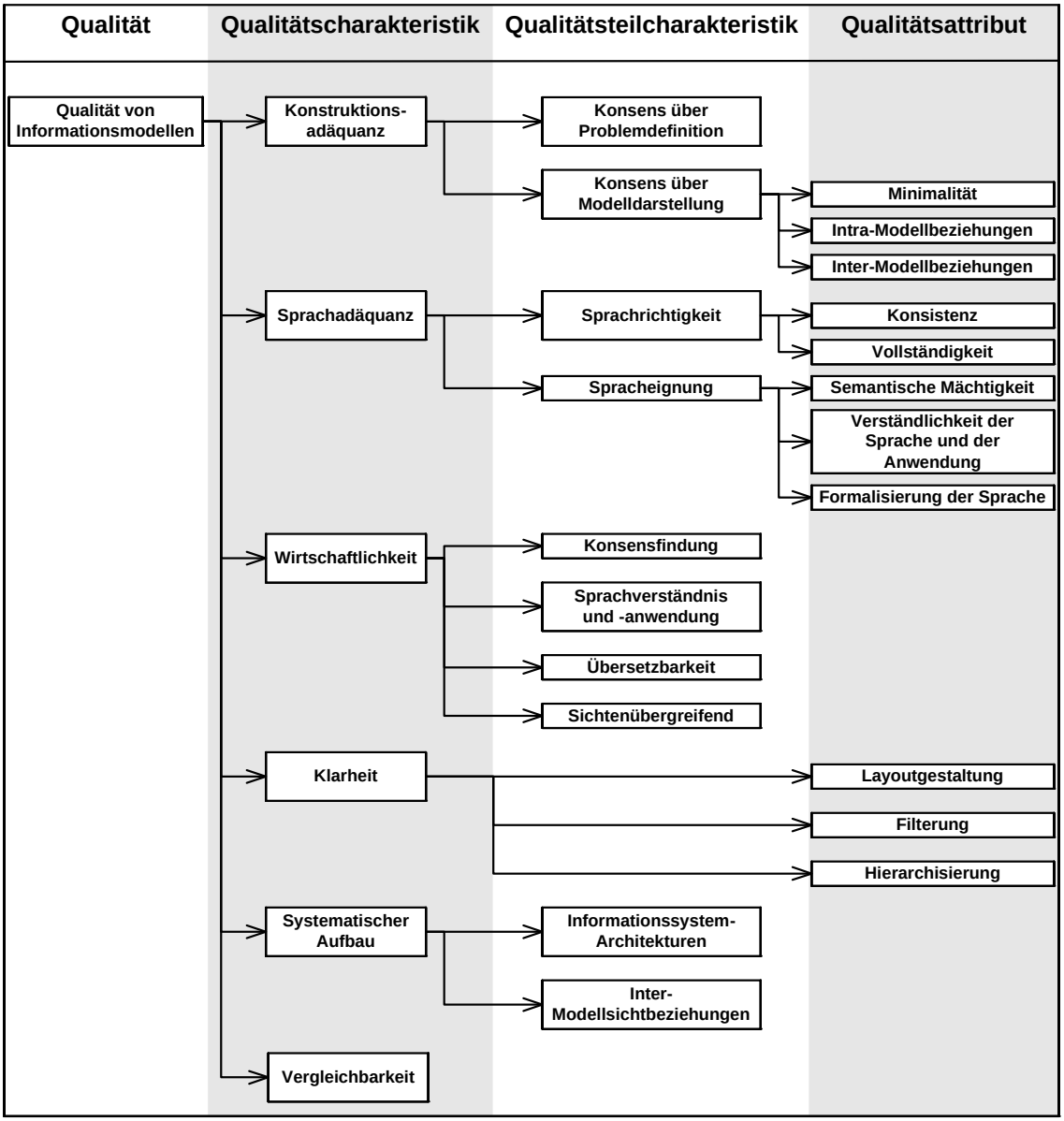


Abbildung 4.6: Qualitätsmodell GOM II

Merkmal	Ausprägung
Detaillierung	low-level
Modelltyp	Datenmodell
Operationalisierung	niedrig
Modellierungssprache	ERM

Tabelle 4.3: Klassifikation des Qualitätsmodells für Datenmodelle

Merkmal	Ausprägung
Detaillierung	low-level
Modelltyp	Entwurfsmodell
Operationalisierung	mittel
Modellierungssprache	UML

Tabelle 4.4: Klassifikation des Qualitätsmodells QOOD

Qualitätsmodell für Datenmodelle

Das älteste uns bekannte Qualitätsmodell für Modelle stammt von Batini et al. [13]. Es bezieht sich auf die Qualität von Entity-Relationship Diagrammen in der Datenmodellierung.

Der Detaillierungsgrad im Qualitätsmodell ist recht unterschiedlich. Überwiegend sind die Qualitätscharakteristiken in sehr konkrete Qualitätsattribute zerlegt wie *Lesbarkeit* zu *Kantenkreuzungen*. Andere Qualitätscharakteristiken wie *Selbsterklärungsfähigkeit* sind wiederum kaum beschrieben (siehe Abb. 4.7). Wir stufen dieses Qualitätsmodell als low-level ein.

Messungen werden lediglich indirekt dokumentiert (z.B. Minimierung der Kantenkreuzungen). Tabelle 4.3 fasst unsere Klassifikation dieses Qualitätsmodells zusammen.

Qualitätsmodell für objektorientiertes Design (QOOD)

Reißing spezialisiert sich in [145] auf die Bewertung von objektorientierten UML-Entwurfsmodellen. Er führt ein generisches Qualitätsmodell für objektorientiertes Design (QOOD) ein, in dem alle Qualitätscharakteristiken und Qualitätsattribute enthalten sein sollen. Die Abhängigkeiten zwischen Qualitätscharakteristiken und Qualitätsattributen bilden eine Netzstruktur. Diese Netzstruktur konnte in Abb.

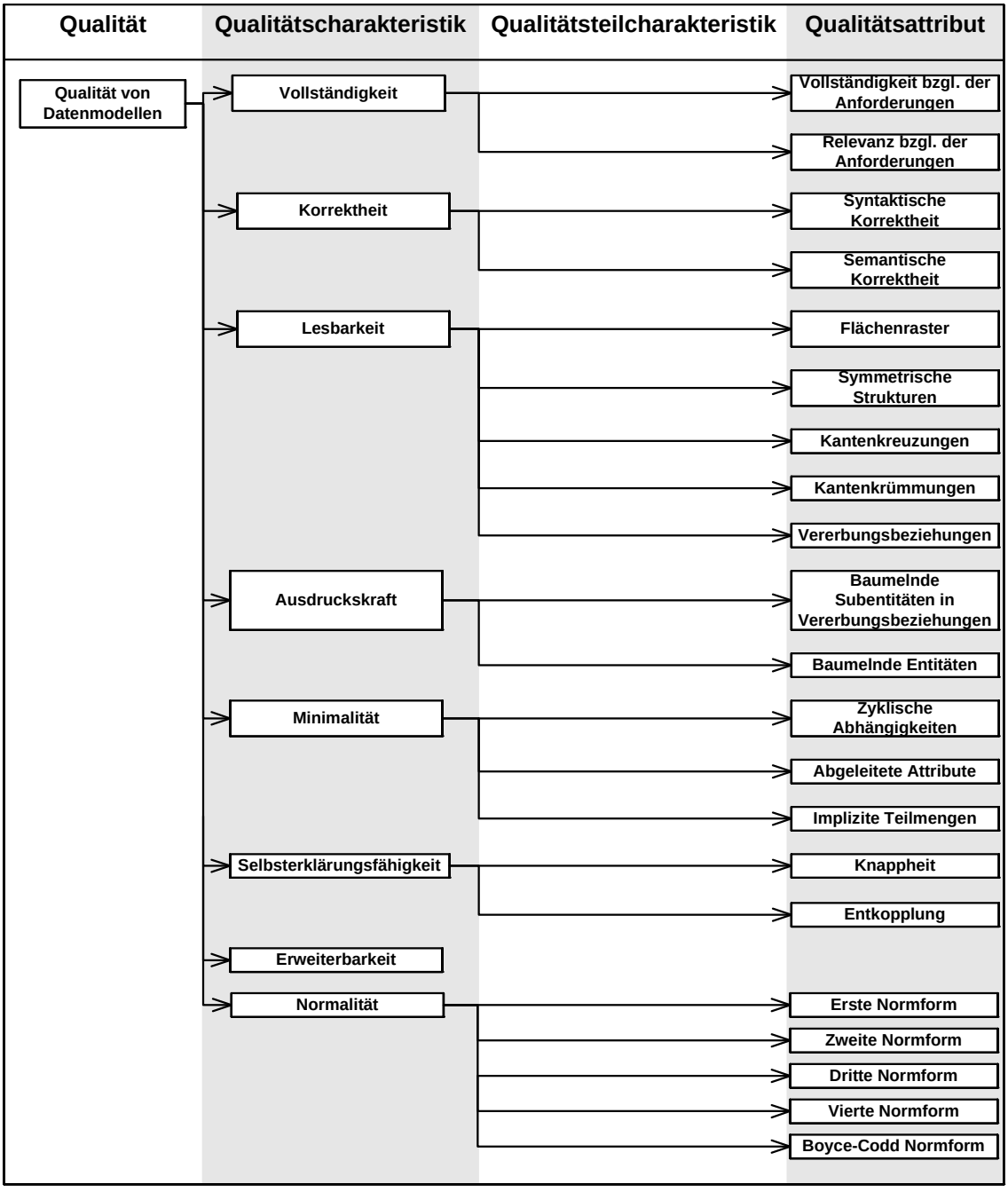


Abbildung 4.7: Qualitätsmodell für Datenmodelle

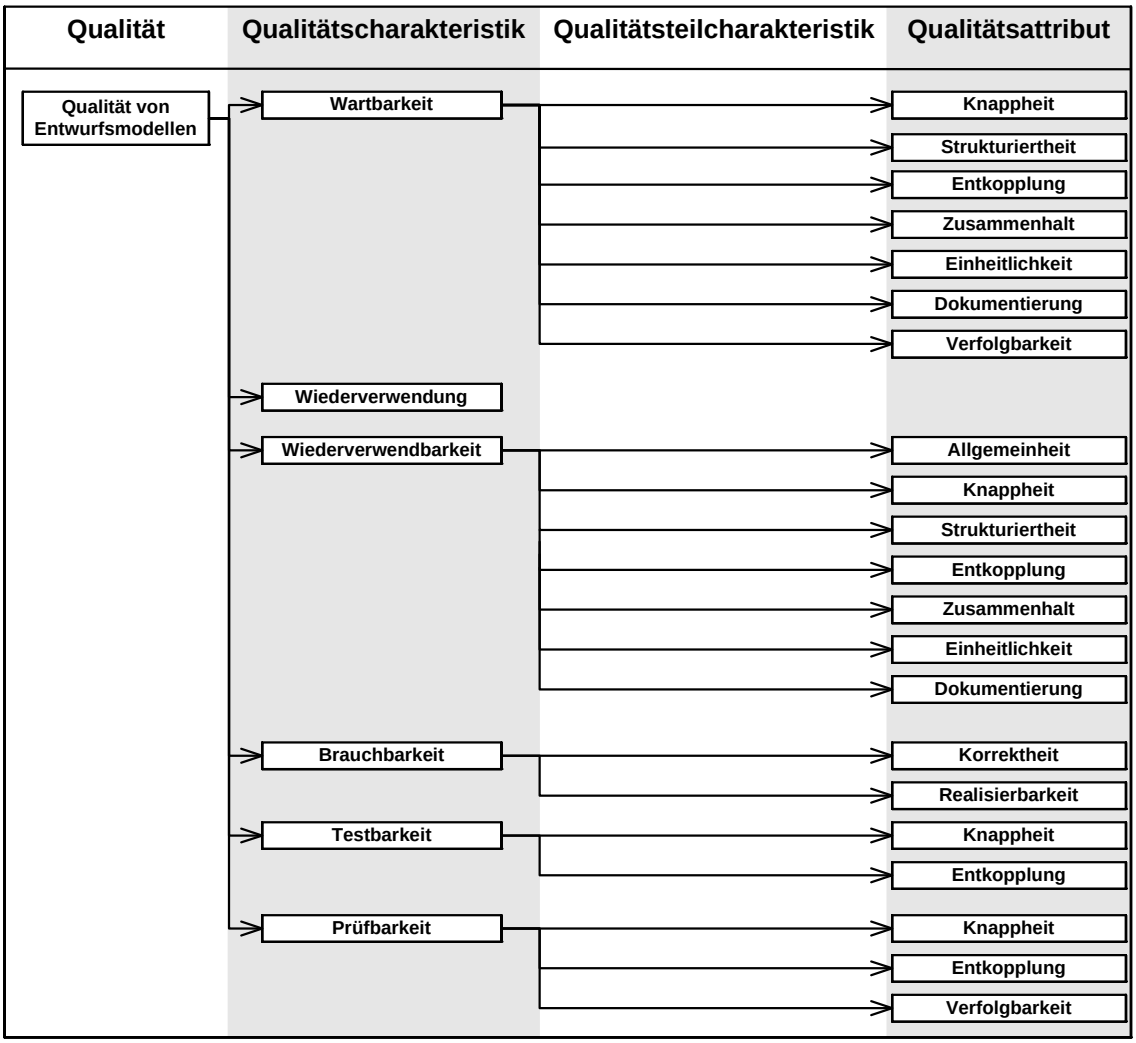


Abbildung 4.8: Qualitätsmodell QOOD

Merkmal	**Ausprägung**
Detaillierung	high-level
Modelltyp	Architekturmodell
Operationalisierung	keine
Modellierungssprache	keine

Tabelle 4.5: Klassifikation des Qualitätsmodells für Software-Architekturen

4.8 nicht übersichtlich visualisiert werden, so dass eine ganze Reihe von Qualitätsattributen mehrfach aufgeführt sind (z.B. Knappheit und Strukturiertheit).

Der Schwerpunkt von QOOD liegt auf der Qualitätscharakteristik *Wartbarkeit*. Für jedes Qualitätsattribut, das der Wartbarkeit untergeordnet ist, schlägt Reißing mehrere objektive und subjektive Messungen vor. Für andere Qualitätsattribute wie *Korrektheit*, das die Qualitätscharakteristik *Brauchbarkeit* verfeinert, fehlen dagegen Messungen. Die Qualitätscharakteristik *Wiederverwendung* lässt sich beispielsweise Reißing zu Folge nicht in sinnvolle Qualitätsattribute zergliedern.

Ausgehend von dem generischen low-level Qualitätsmodell QOOD lassen sich spezifische Qualitätsmodelle zuschneiden. Der Zuschneidemechanismus für die Erstellung spezifischer Qualitätsmodelle beschreibt Reißing kurz. Zu den einzelnen Schritten wie *relevante Qualitätscharakteristiken auswählen* oder *für jede Qualitätscharakteristik die relevanten Qualitätsattribute auswählen* gibt es keine weiteren Hilfestellungen. Reißing bemerkt lediglich dazu, dass eine Voraussetzung für die Ableitung eines spezifischen Qualitätsmodells eine Anforderungsanalyse ist, die Firmen- und Projektrichtlinien, Qualitätsanforderungen an das zu entwickelnde System und eingenommene Qualitätssicht berücksichtigt. Tabelle 4.4 fasst unsere Klassifikation dieses Qualitätsmodells zusammen.

Qualitätsmodell für Software-Architekturen

In [12] stellen Bass et al. ein einfaches Qualitätsmodell auf, das weder Qualitätsattribute noch Messungen enthält. Die Qualitätscharakteristiken sind anhand von Anwendungsszenarien hinreichend konkret beschrieben. Kompakte Definitionen der Qualitätscharakteristiken fehlen dagegen. Annahmen über die Modellierungssprache werden nicht getroffen.

Hinsichtlich sinnvoller Erweiterungen des Qualitätsmodells wird bemerkt, dass “a number of other attributes can be found in the attribute taxonomies in the research

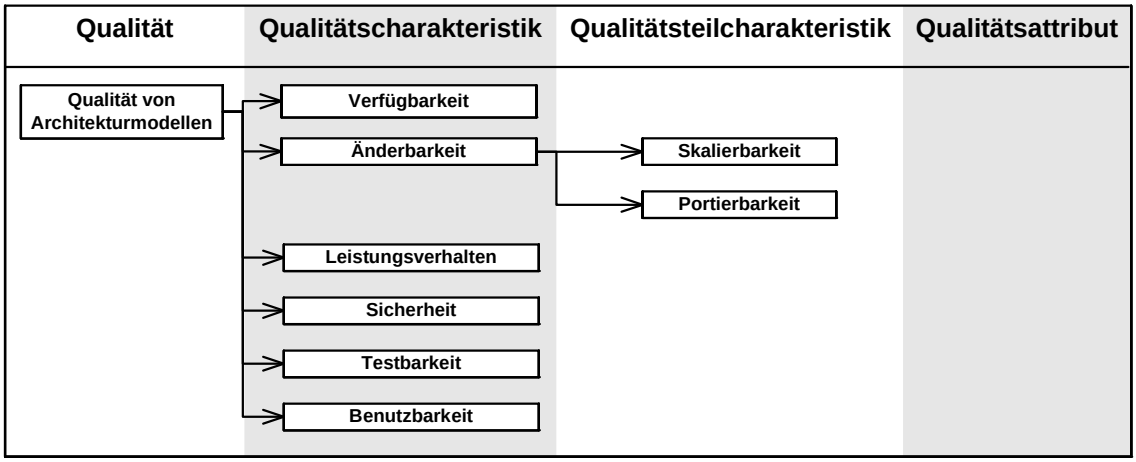


Abbildung 4.9: Qualitätsmodell für Software-Architekturen

Merkmal	**Ausprägung**
Detaillierung	high-level
Modelltyp	Softwaremodell
Operationalisierung	keine
Modellierungssprache	keine

Tabelle 4.6: Klassifikation des Qualitätsmodells für Softwaremodelle

literature and in standard software engineering textbooks" [12].

In [133, 134] wird das generische high-level Qualitätsmodell aus Abb. 4.9 für Service-orientierte Architekturen leicht angepasst. Tabelle 4.5 fasst unsere Klassifikation dieses Qualitätsmodells zusammen.

Qualitätsmodell für Softwaremodelle

Der aktuellste uns bekannte Vorschlag für ein Qualitätsmodell stammt von Fieber et al. [57]. Das in Abb. 4.10 dargestellte Qualitätsmodell knüpft an die ISO/IEC 9126 an und spezialisiert die interne Qualität von Softwareprodukten speziell für Softwaremodelle. Die interne Qualität zerfällt in innere und äußere Qualität.

Es werden keine Einschränkungen hinsichtlich des Modelltyps oder der Modellierungssprache getroffen.

Die Autoren stellen ein sehr umfangreiches Qualitätsmodell auf. Sie erheben trotzdem keinen Anspruch auf Vollständigkeit und bemerken selbst, dass sie z.B. sprach-

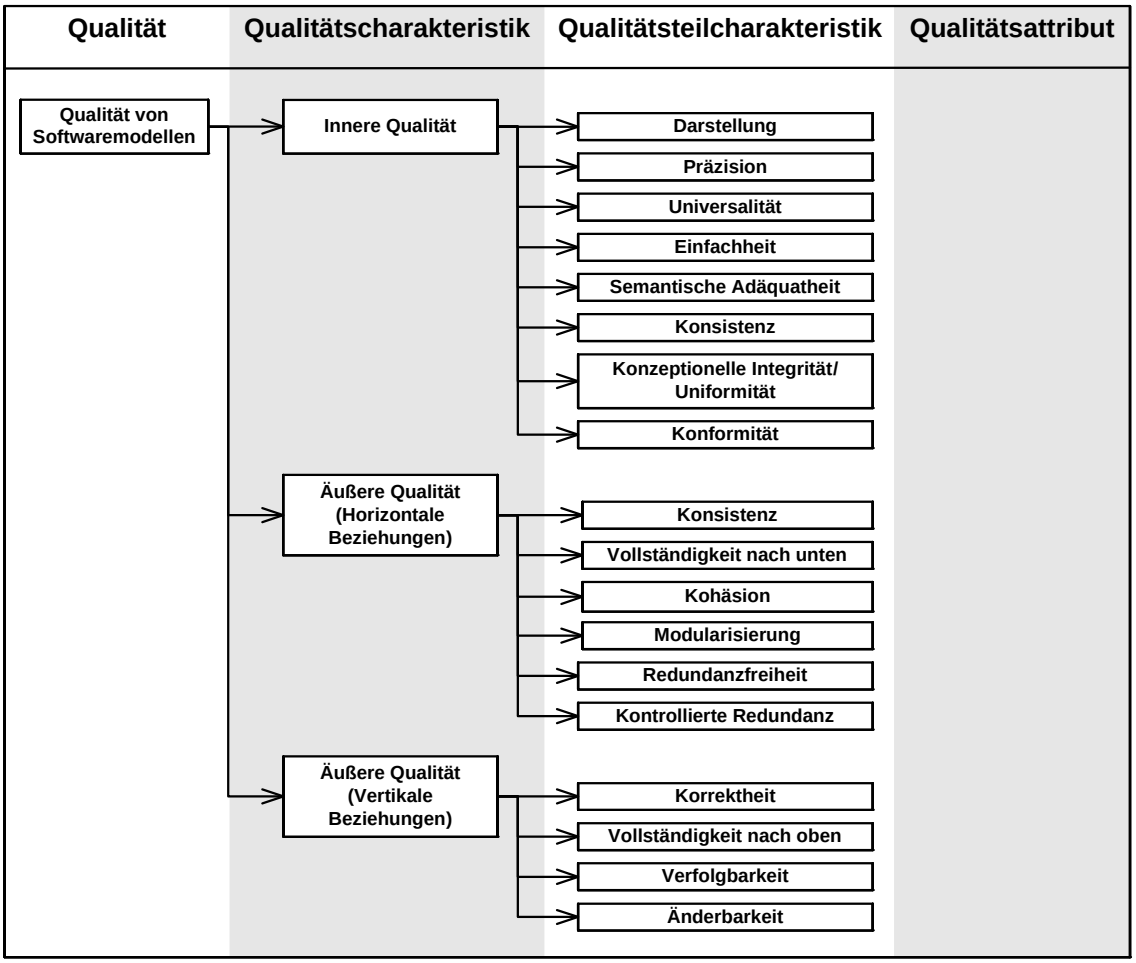


Abbildung 4.10: Qualitätsmodell für Softwaremodelle

spezifische, semantische Qualitätsanforderungen, Tranformierbarkeit und Prüfbarkeit nicht berücksichtigen.

Messungen bleiben in ihrem Ansatz vollständig unbeachtet. Deshalb ist eine eindeutige Zuordnung der untersten Ebene des Qualitätsmodells entweder zu Qualitätsteilcharakteristiken bzw. Qualitätsattributen schwierig. In der Tendenz klassifizieren wir das Qualitätsmodell als high-level, weil Begriffe wie *Darstellung*, *Semantische Adäquatheit* oder *Konzeptionelle Integrität* nur schwer direkt zu messen sind und diese unserer Meinung nach weiter verfeinert werden sollten. Tabelle 4.6 fasst unsere Klassifikation dieses Qualitätsmodells zusammen.

Bewertung

Unsere Bewertung des Konzepts *generische Qualitätsmodelle* haben wir in der Tabelle 4.7 zusammengefasst. Die folgenden Ausführungen erklären die Ausprägungen für die jeweiligen Anforderungen.

Anforderung		**Konzept**
Nr.	Kurzbeschreibung	Generisches Qualitätsmodell
1	Korrekte Qualitätsziele	⊖
2	Vollständige Qualitätsziele	⊕
3	Unmissverständliche Qualitätsziele	⊕
4	Überprüfbare Qualitätsziele	⊕⊖
5	Rückverfolgbare Qualitätsziele	⊖
6	Vorwärtsverfolgbare Qualitätsziele	⊕
7	Nützliche Qualitätsprüfungen	⊕
8	Zuverlässige Qualitätsprüfungen	⊖
9	Wirtschaftliche Qualitätsprüfungen	⊖
10	Interpretierbare Qualitätsprüfungen	⊖
11	Verständliche Qualitätsprüfungen	⊕⊖
12	Wirtschaftliche Qualitätsplanung	n/z
13	Sicherung von Erfahrungswissen	n/z
14	Erlernbares Vorgehen zur Qualitätsplanung	n/z

Tabelle 4.7: Anforderungsmatrix für das Konzept *generisches Qualitätsmodell*

Qualitätsmodelle sind eine etablierte Technik zur Dokumentation von Qualitätszielen. Es existieren bereits mehrere Vorschläge für generische Qualitätsmodelle, die sich speziell auf Modellqualität beziehen. Generische Qualitätsmodelle kapseln wichtiges Erfahrungswissen. Dieses Erfahrungswissen lässt sich durch einen Qualitätsmanager bei der Identifizierung von Qualitätszielen nutzen. Auf diese Weise helfen uns bestehende Qualitätsmodelle, eine vollständige Menge an Qualitätszielen aufzubauen (Anf. 2 $\oplus$). Zudem haben eine Reihe von Autoren die vorgeschlagenen Qualitätscharakteristiken bzw. Qualitätsattribute präzise definiert. Diese Definitionen wirken sich positiv auf unsere Forderung nach unmissverständlichen Qualitätszielen aus (Anf. 3 $\oplus$).

Generische Qualitätsmodelle sind nicht allgemeingültig und müssen auf konkrete Projektanforderungen hin angepasst werden. Die Vorgehensbeschreibungen für die Anpassung generischer Qualitätsmodelle sind jedoch kaum dokumentiert und die Einbindung des Projektteams nicht vorgesehen (vgl. Ausführungen zu SQUID und QOOD). Wir beurteilen die Vorgehen in ihrer Methodik als schwach und finden sie in dieser Form wenig hilfreich. Deshalb besteht bei der Wiederverwendung von bestehenden Qualitätsmodellen zur Zeit die Gefahr, dass Qualitätscharakteristiken und Qualitätsattribute in einen Qualitätsplan übernommen werden, obwohl sie für das aktuell zu prüfende Softwaremodell irrelevant und somit inkorrekt sind (Anf. 1 $\ominus$). Aufgrund dieser methodischen Schwäche werden wir unsere Anforderungen zur Qualitätsplanung für generische Qualitätsmodelle nicht evaluieren (Anf. 12, 13 und 14 n/z).

Die zuvor vorgestellten generischen Qualitätsmodelle unterscheiden sich im Aspekt *Operationalisierung* sehr stark voneinander. Die Bandbreite reicht von Ansätzen, die ihr Qualitätsmodell zusätzlich um eine Reihe an Messungen ergänzen bis hin zu Ansätzen, die die eingeführten Qualitätscharakteristiken nicht einmal zu Qualitätsattributen verfeinern, so dass für diese weitestgehend unklar bleibt, wie die Qualitätsziele zu prüfen sind (Anf. 4 $\oplus\ominus$).

In den zur Verfügung stehenden Qualitätsmodellen sind die getroffenen Annahmen über den Kontext des zu prüfenden Softwaremodells sehr grob. Die Autoren geben wenn überhaupt Informationen über den Modelltyp und die Modellierungssprache an. Einen detaillierten Zusammenhang zwischen dem Kontext des Softwaremodells und einzelnen Qualitätscharakteristiken bzw. Qualitätsattributen lässt sich nicht finden. Infolgedessen ist eine Rückverfolgbarkeit nicht gegeben (Anf. 5 $\ominus$).

Dagegen können wir zumindest für die Qualitätsmodelle mit entsprechenden assoziierten Messungen eine Möglichkeit der Vorwärts-Verfolgbarkeit attestieren (Anf. 6 $\oplus$).

Die Auswahl nützlicher Qualitätsprüfungen für gegebene Qualitätsziele stellt sich in der Praxis als problematisch dar. Das Erfahrungswissen in Qualitätsmodellen mit assoziierten Messungen kann hier sehr hilfreich sein (Anf. 7 $\oplus$). Die Messungen sind i.d.R. informell in natürlicher Sprache (z.B. Englisch) definiert und können von allen Projektbeteiligten gelesen werden (Anf. 11 $\oplus$). Im Falle informeller Messbeschreibungen bleibt aber häufig unklar, inwiefern beispielsweise bei einer Messung *Anzahl Methoden im Modell* vererbte Methoden gezählt werden sollen (Anf. 11 $\ominus$). Dies wirkt sich auch automatisch negativ auf die Reproduzierbarkeit von Prüfergebnissen aus (Anf. 8 $\ominus$). Zudem lässt sich bei vielen beschriebenen Messungen das Automatisierungspotential nicht direkt erkennen, so dass der Aufwand für ihre Erhebung schwer abzuschätzen ist (Anf. 9 $\ominus$). Auch Interpretationshilfen, wann ein Wert für eine Messung gut oder schlecht ist, fehlen in diesen Ansätzen (Anf. 10 $\ominus$).

4.2.2 Vollständig individuelle Qualitätsmodelle

Goal Question Metric (GQM)

Die Goal Question Metric (GQM) ist ein systematisches Vorgehensmodell für die Entwicklung und Durchführung eines Messprogramms. GQM wird in der Softwareentwicklung für Messungen eingesetzt, die auf Produkten, Prozessen und Ressourcen basieren. GQM ist durch eine konsequente Zielorientierung geprägt. Die Grundidee von GQM basiert auf der Annahme, "that for an organization to measure in a purposeful way it must first specify the goals for itself and its projects, then it must trace those goals to the data that are intended to define those goals operationally, and finally provide a framework for interpreting the data with respect to the stated goals" [7]. Der Mehrwert des GQM-Ansatzes für die Bewertung und darauf basierenden Verbesserung der Produkt- und Prozessqualität wurde in [19, 71] evaluiert und bestätigt. Der GQM-Prozess besteht aus sechs wesentlichen Schritten (vgl. hierzu [26, 69]):

1. Charakterisierung des Unternehmens- und Projektkontextes.
2. Identifizierung von Informationsbedürfnissen mittels Zielen und korrespondierenden Fragestellungen unter aktiver Einbeziehung der Projektbeteiligten oder

des Managements.

3. Definition von Messungen zur quantitativen Beantwortung der in Schritt 2 formulierten Fragen.

4. Durchführung der Messungen sowie anschließende Analyse und Interpretation der resultierenden Messdaten.

5. Analyse und Interpretation der Messdaten im Hinblick auf ihre Bedeutung für die Struktur und den Ablauf in der Organisation und im Gesamtprojekt sowie die Diskussion allgemeiner Lehren, die aus den Messergebnissen gezogen werden dürfen.

6. Nachbereitung der Messungen durch Sicherung der in Schritt 5 gewonnen Erkenntnisse und Speicherung des Messprogramms zur späteren Wiederverwendung in ähnlichen Projekten.

Für die Definition von Zielen existiert ein festes Schema. Es besteht aus den fünf Dimensionen Untersuchungsobjekt, Zweck, Qualitätsschwerpunkt, Sichtweise und Kontext der Analyse.

Dimension	**Definition**	**Beispiel**
Untersuchungs-objekt	Was wird analysiert?	Entwicklungsprozess, Entwurfs-artefakt, Softwareprodukt
Zweck	Warum wird das Objekt analysiert?	Charakterisierung, Evaluation, Prognose
Qualitäts-schwerpunkt	Welche Charakteristik wird analysiert?	Zuverlässigkeit, Kosten, Korrektheit, Benutzerfreundlichkeit
Sichtweise	Wer nutzt die Analyseergebnisse?	Projektleiter, Entwickler, Qualitätsmanager, SW-Architekt
Kontext	In welchem Kontext wird analysiert?	Projekt X, Kooperation A

Tabelle 4.8: Schema für die Definition von Zielen nach [26]

GQM wird z.B. in [57] und [145] als Vorgehen zur Entwicklung eines spezifischen Qualitätsmodells vorgeschlagen. Zwischen den von der ISO/IEC 9126 geforderten

Qualitätsmodellen und den durch die Anwendung der GQM-Methode erstellten Qualitätsmodellen, im Folgenden mit GQM-QM abgekürzt, gibt es einige Unterschiede. Das GQM-QM besteht nicht aus Qualitätscharakteristiken und -attributen, sondern aus einem Netz von Zielen und zugeordneten Fragestellungen. Jedes Ziel hat einen Qualitätsschwerpunkt. Auf diese Weise entsteht ein Qualitätsmodell mit genau einer Ebene von Qualitätscharakteristiken. Der Definitionsgrad der Qualitätsmodelle ist ein weiterer Unterschied. In einem GQM-QM sind weder Qualitätsschwerpunkt noch verwendete Begrifflichkeiten in den Fragestellungen definiert. Zudem hängen im GQM-Ansatz Messungen mit Fragen und nicht mit Qualitätsattributen zusammen.

Dagegen stellt die Dimension *Sichtweise* eines GQM-Zieles eine interessante, konzeptionelle Erweiterung der Qualitätsmodelle dar. Durch die Berücksichtigung von verschiedenen Qualitätssichten bei der Gestaltung von Qualitätsmodellen entstehen Qualitätsmodelle für unterschiedliche Personengruppen. Auf diese Weise lässt sich z.B. die Relativität von Qualität modellieren. Die Idee sichtenorientierter Qualitätsmodelle wurde auch in [46] angedacht.

Weitere GQM-Ansätze

Die obigen Ausführungen erwecken den Anschein, als ob es nur genau einen GQM-Ansatz gibt. Jedoch existieren eine ganze Reihe von Arbeiten, die den GQM-Ansatz beschreiben, erweitern oder spezialisieren. Deshalb möchten wir einen kurzen Überblick über die uns bekannten GQM-Varianten geben.

Das *ursprüngliche GQM-Vorgehensmodell* wurde im Rahmen des TAME (Tailoring A Measurement Enviroment) Projekts beschrieben und durch ein einfaches Metamodell ergänzt (vgl. [9, 11, 91, 7, 136]). Das Metamodell ist zweigeteilt und auf Basis von Entity Relationship Diagrammen modelliert. Das SE (Software Engineering) Metamodell enthält Konzepte zur Charakterisierung des Umfeldes (vgl. Schritt 1). Das GQM-Metamodell setzt sich aus den Konzepten *Ziel*, *Frage* und *Metrik* zusammen und bezieht sich somit auf Schritt 2 und 3.

In [141] wird der GQM-Ansatz um eine Definition von Indikatoren erweitert und dadurch der *GQ(I)M* Ansatz definiert. Die Indikatoren sind zwischen den Fragen und den Messungen angeordnet und sollen einerseits die Auswahl effektiver Messungen und andererseits die Interpretation der Messergebnisse erleichtern.

GQM++ verfeinert den GQM-Ansatz um die optionale Möglichkeit, Ziele durch Subziele, Fragen durch Subfragen und Metriken durch Submetriken zu verfeinern

[68, 111]. Durch diese Verfeinerungen wird ein systematisches Ableiten von Messungen gefördert. Zudem lässt sich auf diese Weise eine höhere Genauigkeit der Qualitätsplanung erzielen. Z.B. werden die Abhängigkeiten zwischen Messungen und entsprechenden Fragen detailliert. Wie in [69] beschrieben, ermöglicht GQM++ durch die Verfeinerung von Zielen in Subziele zusätzlich die Definition eines mehrschichtigen Qualitätsmodells.

V-GQM fügt dem GQM-Ansatz analog zum rechten Zweig des V-Modells Validierungsschritte hinzu [138]. Der Ansatz ermöglicht auf diese Weise, eine mit GQM erstellte Qualitätsplanung zu bewerten.

Aufgrund des mit dem GQM-Ansatzes verbundenen Aufwands für die Erstellung von Qualitätsplänen hat von Wangenheim *GQM Lightweight* für kleine und mittelständische Unternehmen vorgeschlagen [72, 70]. In diesem Ansatz werden gängige Ziele unabhängig von einem konkreten Projekt oder Unternehmen beschrieben. Diese Ziele können anschließend durch Parametrisierung auf ein konkretes Unternehmen und dessen Softwareentwicklungsprojekt angewendet werden.

GQM$^+$Strategies ist eine sehr aktuelle Erweiterung des ursprünglichen GQM-Ansatzes [8]. Das Ziel von GQM$^+$Strategies besteht darin, Messziele in GQM mit übergeordneten Organisationszielen zu verknüpfen.

Bewertung

Unsere Bewertung des Konzeptes *GQM* haben wir in der Tabelle 4.9 zusammengefasst. Die folgenden Ausführungen erklären die Ausprägungen für die jeweiligen Anforderungen.

GQM ist eine etablierte und ausführlich dokumentierte Technik für die Erstellung individueller Qualitätsmodelle. Basierend auf Zielen und korrespondierenden Fragestellungen, die unter aktiver Einbeziehung der Projektbeteiligten identifiziert werden, lassen sich systematisch Messungen ableiten (Anf. 4 $\oplus$ und Anf. 7 $\oplus$). Durch die gezielte Einbindung des Projetteams stellt das GQM-Vorgehen sicher, dass die Qualitätsziele korrekt sind (Anf. 1 $\oplus$). Der in einem Ziel referenzierte Qualitätsschwerpunkt wird bezeichnet, jedoch nicht definiert (Anf. 3 $\ominus$).

Für die Anwendung von GQM bzw. darauf basierenden Ansätzen existieren bereits gute Leitfäden, die das Vorgehen und die zu erstellenden Dokumente detailliert beschreiben (Anf. 14 $\oplus$). Durch den hohen Dokumentationsaufwand bei der Anwendung von GQM ist der Ansatz recht schwergewichtig (Anf. 12 $\ominus$). Zudem ist uns

Anforderung		Konzept
Nr.	Kurzbeschreibung	GQM
1	Korrekte Qualitätsziele	$\oplus\oplus$
2	Vollständige Qualitätsziele	$\ominus$
3	Unmissverständliche Qualitätsziele	$\ominus$
4	Überprüfbare Qualitätsziele	$\oplus$
5	Rückverfolgbare Qualitätsziele	$\ominus$
6	Vorwärtsverfolgbare Qualitätsziele	$\oplus$
7	Nützliche Qualitätsprüfungen	$\oplus\oplus$
8	Zuverlässige Qualitätsprüfungen	$\ominus$
9	Wirtschaftliche Qualitätsprüfungen	$\ominus$
10	Interpretierbare Qualitätsprüfungen	$\ominus$
11	Verständliche Qualitätsprüfungen	$\oplus\ominus$
12	Wirtschaftliche Qualitätsplanung	$\ominus\ominus$
13	Sicherung von Erfahrungswissen	$\oplus\ominus$
14	Erlernbares Vorgehen zur Qualitätsplanung	$\oplus\ominus$

Tabelle 4.9: Anforderungsmatrix für das Konzept *GQM*

keine einzige Anwendung von GQM auf den Untersuchungsgegenstand *Softwaremodell* bekannt (Anf. 14 $\ominus$).

Die mit GQM erstellten Qualitätsmodelle entsprechen strukturell nicht den bereits existierenden Qualitätsmodellen (vgl. ISO Qualitätsmodell in Abschnitt 4.1.2 und Qualitätsmodelle im vorherigen Abschnitt 4.2.1). Die Wiederverwendung existierender Qualitätsmodelle innerhalb des GQM-Vorgehens ist aufgrund dieser Inkompabilität stark eingeschränkt (Anf. 12 $\ominus$). Als direkte Konsequenz daraus folgt, dass sich bestehende Qualitätsmodelle auch nur bedingt eignen, um die Vollständigkeit von Zielen in GQM-Plänen zu gewährleisten (Anf. 2 $\ominus$).

Die Sicherung von Erfahrungswissen ist grundsätzlich vorgesehen. Allerdings berücksichtigen die dazu zur Verfügung stehenden Mechanismen aufgrund der Ausrichtung von GQM auf eine Unternehmens- bzw. Projektsicht nicht die Besonderheiten von Softwaremodellen (Anf. 13 $\oplus\ominus$).

Zum GQM-Ansatz komplementäre Ansätze wie V-GQM wirken sich zusätzlich positiv auf die Verwendung korrekter Qualitätsziele und nützlicher Qualitätsplanungen aus (Anf. 1 und 7 $\oplus$).

Der Detaillierungsgrad von Messbeschreibungen ist vergleichbar mit dem der Qualitätsmodelle. Zudem fehlt auch in GQM eine Kontextbeschreibung von Softwaremodellen. Demzufolge stimmen die Bewertungen der Anforderungen zu verfolgbaren Qualitätszielen (Anf. 5 und 6) als auch die Mehrheit der Anforderungen, die sich auf die Dokumentation von Qualitätsprüfungen beziehen (Anf. 8, 9, 10 sowie 11) mit denen für generische Qualitätsmodelle überein. Ergänzend hierzu möchten wir noch bemerken, dass die im GQ(I)M vorgesehenen Indikatoren lediglich auf die Visualisierung der Messwerte abzielen und folglich unsere Anforderung nach interpretierbaren Qualitätsprüfungen nicht positiv beeinflussen.

Zusammenfassung

Sowohl Qualitätsmodelle als auch GQM sind etablierte Techniken. Es existieren bereits diverse generische Qualitätsmodelle für Modellqualität, die jedoch im Aspekt *Messung* unvollständig sind. Für die Anpassung dieser Qualitätsmodelle ist uns kein nützliches Vorgehen bekannt. Wir beurteilen GQM als grundsätzlich geeignet, um ein individuelles Qualitätsmodell aufzubauen. Jedoch lassen sich bestehende Qualitätsmodelle nicht mit dem GQM-Ansatz wiederverwenden. Zudem ist GQM bis dato nicht für die Qualitätsbewertung von Softwaremodellen eingesetzt worden. Sowohl generische Qualitätsmodelle als auch GQM berücksichtigen die Besonderheiten von Softwaremodellen bei der Kontextbeschreibung und Messdefinition unzureichend.

4.3 Messungen

Zur Festlegung von Qualitätszielen in der Qualitätssicherung von Softwaremodellen haben wir im vorausgegangenen Abschnitt 4.2 das Konzept *Qualitätsmodell* eingeführt. Projektspezifische Qualitätsmodelle können entweder durch die Anpassung *generischer Qualitätsmodelle* oder durch die Anwendung der *Goal Question Metric (GQM)* erstellt werden. Der Umfang der in Qualitätsmodellen referenzierten Messungen als auch die eingesetzten Beschreibungsschemata sind jedoch unbefriedigend. Dabei sind Messungen zur Bestimmung des Erfüllungsgrads von Qualitätszielen wichtig. Deshalb gehen wir in diesem Abschnitt gezielt auf Ansätze ein, deren Schwerpunkt in der Beschreibung von Messungen für Softwaremodelle besteht.

Messungen spielen seit jeher in Wissenschaft und Praxis eine wichtige Rolle. Beispielsweise wird Galileo Galilei das Zitat “Alles messen, was messbar ist - und messbar machen, was noch nicht messbar ist” zugeschrieben. Messungen sind ein exzellenter Abstraktionsmechanismus und ermöglichen, umfangreiche und komplexe Sachverhalte verhältnismäßig einfach darzustellen [56]. Messungen lassen sich gut skalieren und eignen sich deshalb auch für sehr große Modellsysteme.

Bei Messungen unterscheiden wir zwischen *Metriken* und *Indikatoren*. *Metriken* ermöglichen die *Quantifizierung von Qualitätsattributen* in einem Qualitätsmodell. Sie beschreiben eine Abbildung von der Domäne *Softwaremodell* in eine *numerische* Domäne. Für eine qualifizierende Aussage über Qualitätsattribute bieten sich *Indikatoren* und zugehörige Analysen an.

Auf eine weitergehende *Bewertung der Qualitätscharakteristiken*, die auf der Quantifizierung oder Qualifizierung von Qualitätsattributen basiert, gehen wir nicht ein. Bei der Bewertung von Qualitätscharakteristiken müssen mehrere Qualitätsattribute und für jedes Qualitätsattribut u.U. eine ganze Reihe von Messungen betrachtet werden. Weil Messungen sich wiederum auch auf unterschiedliche Qualitätsattribute beziehen können, muss hierbei jeder Messung/Qualitätsattribut-Kombination der *richtige* Einfluss auf die betrachtete Qualitätscharakteristik beigemessen werden. Dabei ist der Einfluss von einzelnen Messungen auf einzelne Qualitätsattribute insbesondere für Modellqualität lückenhaft untersucht. Deshalb muss die Bewertung von Qualitätscharakteristiken vorerst den Entscheidungsträgern überlassen bleiben.

Für die Bestimmung von Metriken lassen sich subjektive und objektive Messmethoden unterscheiden (vgl. dazu die Ausführungen zu den Standards zur Qualitätsprüfung in Abschnitt 4.1.3). Vor diesem Hintergrund differenzieren wir zwischen *objektiven Metriken* und *subjektiven Metriken*.

Objektive Messmethoden basieren auf numerischen Regeln und können deshalb von Werkzeugen automatisch durchgeführt werden. Dadurch wird die Erhebung von Metriken deutlich einfacher und i.d.R. kostengünstiger.

Jedoch können Metriken wie *Anzahl modellierter Anforderungen*, *Anzahl relevanter Klassen* oder *Anzahl ähnlicher Operationen in anderen Klassen* nicht durch ein Werkzeug alleine quantifiziert werden. Auch die Bewertung von Entwurfsentscheidungen benötigt häufig die Einbeziehung eines Experten. Card und Glass bemerken dazu, dass “the intellectual nature of the software design process means that important components of it must be measured subjectively” [35]. Deshalb lassen sich

einige Qualitätsattribute ausschließlich durch subjektive Metriken bestimmen.

Indikatoren bauen auf Metriken auf und transformieren quantitative Aussagen mit Hilfe von Entscheidungskriterien in qualitative Aussagen. Entscheidungskriterien wie z.B. Grenzwerte für annehmbare oder nicht annehmbare Zielerfüllung leiten sich entweder aus theoretischen Überlegungen oder Erfahrungswerten ab. Indikatoren erleichtern die Interpretation von Messungen durch Messnutzer, indem höherwertige Aussagen über Qualitätsattribute generiert werden.

Zunächst stellen wir in Abschnitt 4.3.1 Metriken vor, die auf objektiven Messmethoden beruhen. Anschließend gehen wir in Abschnitt 4.3.2 auf subjektive Metriken ein. Indikatoren thematisieren wir ausführlicher in Abschnitt 4.3.3. Am Ende jedes Abschnitts bewerten wir die vorgestellten Konzepte anhand unserer aufgestellten Anforderungen.

4.3.1 Objektive Metriken

Messungen für objektive Metriken lassen sich formal beschreiben, infolgedessen automatisieren und preiswert erheben. In der Literatur gibt es viele Vorschläge für objektive Metriken für Softwareprodukte. Einige von diesen Metriken lassen sich auch auf UML-Softwaremodelle anwenden. Deutlich weniger Arbeiten beziehen sich ausschließlich auf objektive Metriken für bestimmte UML-Diagramme.

Metriken sind in der Literatur sehr unterschiedlich dokumentiert. Häufig besteht die Dokumentation ausschließlich aus einem Metriknamen und einer textuellen Beschreibung der Messvorschrift (z.B. [102, 16, 62, 106, 4]). Einige Metriksammlungen enthalten formale Definitionen der Metriken. Eine allgemein akzeptierte Beschreibungssprache hat sich allerdings noch nicht durchgesetzt. In [28] und [115] sind Metriken mit Hilfe der Object Constraint Language (OCL) [127] definiert. Dagegen wird in in [53] XQuery [181] verwendet und [145] sowie [112] nutzen jeweils ein selbst aufgestelltes mathematisches Modell zur Definition ihrer Metriken. Vielen Metriksammlungen mangelt es zudem an der Nennung und Definition der gemessenen Qualitätsattribute, so dass es in diesen Fällen häufig unklar bleibt, welches Qualitätsziel mit einer Metrik überprüft werden kann. Für die uns bekannten Metriksammlungen ist allgemein festzuhalten, dass Metriken nicht so ausführlich, sondern so knapp wie möglich dokumentiert sind.

In den Tabellen 4.10 und 4.11 geben wir eine Reihe von Beispielen für objektive Metriken an. Unsere Auswahl objektiver Metriken gruppieren wir in den Tabellen

anhand der Entwicklungsphasen *Analyse* (siehe Tab. 4.10) und *Entwurf* (siehe Tab. 4.11), so dass dieses Beschreibungselement für die Dokumentation einer Metrik kein Pendant in den Tabellenspalten, sondern in der Tabellenbezeichnung hat. Damit die folgenden Beispiele für objektive Metriken einheitlich dargestellt sind, haben wir uns für ein durchgängiges Beschreibungsschema entschieden und fehlende Informationen nach unserem Verständnis der uns vorliegenden Literatur ergänzt. Das in diesem Abschnitt genutzte Beschreibungsschema für objektive Metriken besteht aus den folgenden Teilen:

- *QA* benennt das Qualitätsattribut, das mit Hilfe der Metrik quantifiziert wird.
- *Diagr.-Typ* gibt den als Eingabe benötigten UML-Diagrammtyp für eine Metrik an. Für die Diagrammtypen verwenden wir folgende Abkürzungen: Use Case Diagramm (UCD), Klassendiagramm (KD), Objektdiagramm (OD), Sequenzdiagramm (SD), Komponentendiagramm (KomD), Statechart (SC) und OCL Constraints (OCL)
- *Akronym* und *Name* bezeichnen eine Metrik.
- *Informelle Messbeschreibung* skizziert die notwendigen Berechnungen zur Bestimmung der Kennzahl.
- *Ref* verweist auf die jeweilige Literatur.

Damit die Tabellen übersichtlich bleiben, verzichten wir an dieser Stelle auf die für objektive Metriken wichtige *formale Messbeschreibung* und die Dokumentation von *Skalentyp* sowie *Skala* zur Eingrenzung der erwarteten Messwerte.

Es gibt vereinzelt Arbeiten, die Metriken für Softwaremodelle in der Analyse vorschlagen. Deutlich umfangreicher ist die Menge der verfügbaren Metriken speziell für Klassendiagramme in der Entwurfsphase. Dies hängt damit zusammen, dass das Klassendiagramm der verbreiteste UML-Diagrammtyp ist und viele Code-Metriken sich auf Entwurfs-Klassendiagramme übertragen lassen.

Die von uns getroffene Auswahl an Metriken versucht eine möglichst große Bandbreite an Qualitätsattributen und Diagrammtypen innerhalb zweier Tabellen abzudecken. Dabei zitieren wir Metriken vieler unterschiedlicher Autoren. Dies kann als zusätzliches Indiz für die Bedeutung von Metriken im Rahmen der Qualitätssicherung in der Softwareentwicklung gedeutet werden.

QA	Diagr.-typ	Akronym (Name)	Informelle Messbeschreibung	Ref
Anwendungs-komplexität	UCD	UC3 *(o.A.)*	Anzahl der Kommunikationsbeziehungen zwischen Anwendungsfällen und Aktoren ohne Zählung der Redundanzen, die durch `«extend»` und `«include»` Beziehungen entstehen	[112]
Strukturelle Komplexität	KD	TNC *(Total Number of Classes)*	Anzahl Klassen	[137]
Strukturelle Komplexität	KD	NCH *(Number of Class Hierarchies)*	Anzahl verschiedener Vererbungshierarchiestrukturen; Klassen, die in keiner Vererbungshierarchie vorkommen, bilden ihre eigene Klassenhierarchiestruktur	[137]
Intra-modell Konsistenz	KD und SD	NIC *(Non Instantiated Classes)*	Anzahl nicht instantiierter Klassen im Softwaremodell	[16]
Zuverlässiges Verhalten	SD	NPO *(Number of Passive Objects)*	Anzahl Objekte, die Nachrichten erhalten, aber keine Nachrichten versenden	[16]

Tabelle 4.10: Metriken für UML-Softwaremodelle in der Analyse

QA	Diagr.-typ	Akronym (Name)	Informelle Messbeschreibung	Ref
Strukturelle Komplexität	KD	DIT *(Depth of Inheritance Tree)*	Maximale Anzahl von Vererbungsbeziehungen zwischen der betrachteten Klasse und der Wurzelklasse	[38]
Kopplung	KD	DAC *(Data Abstraction Coupling)*	Anzahl der Attribute einer Klasse, die eine andere Klasse als Typ haben	[106]
Interne Wiederverwendung	KD	OIF *(Operations Inheritance Factor)*	Quotient zwischen der Anzahl vererbter Operationen und der Anzahl verfügbarer Operationen für alle Klassen im Softwaremodell	[28]
Interne Wiederverwendung	KD	SIX *(Specialization IndeX Metric)*	Anzahl überschriebener Methoden $*$ Vererbungstiefe / Anzahl aller Methoden	[109]
Intra-modell Konsistenz	KD, SD	MWM *(Messages without Method)*	Anzahl der Nachrichten, die keine korrespondierende Methode in der Klassendefinition des empfangenden Objekts hat	[103]
Validität	KD	ACNI *(Abstract Classes Not Inherited)*	Anzahl der abstrakten Klassen, die nicht vererbt werden	[37]
Verhaltenskomplexität	SC	CC *(Cyclomatic Complexity for SC)*	Anzahl einfacher Zustände − Anzahl Transitionen +2	[62]
Kognitive Komplexität	OCL	NKW *(Number of OCL KeyWords)*	Anzahl von OCL KeyWords	[147]

Tabelle 4.11: Metriken für UML-Softwaremodelle im Entwurf

Unsere Metrikauswahl gibt nur einen groben Überblick über existierende Arbeiten. Weitere Metriken für Softwaremodelle finden sich beispielsweise in den referenzierten Arbeiten und im Anhang A.

Bewertung

Unsere Bewertung des Konzepts *objektive Metrik* haben wir in der Tabelle 4.12 zusammengefasst. Die folgenden Ausführungen erklären die Ausprägungen für die jeweiligen Anforderungen.

Anforderung		**Konzept**
Nr.	Kurzbeschreibung	Objektive Metrik
1	Korrekte Qualitätsziele	n/z
2	Vollständige Qualitätsziele	n/z
3	Unmissverständliche Qualitätsziele	n/z
4	Überprüfbare Qualitätsziele	$\oplus\ominus$
5	Rückverfolgbare Qualitätsziele	n/z
6	Vorwärtsverfolgbare Qualitätsziele	n/z
7	Nützliche Qualitätsprüfungen	$\oplus$
8	Zuverlässige Qualitätsprüfungen	$\oplus$
9	Wirtschaftliche Qualitätsprüfungen	$\oplus$
10	Interpretierbare Qualitätsprüfungen	n/z
11	Verständliche Qualitätsprüfungen	$\oplus$
12	Wirtschaftliche Qualitätsplanung	n/z
13	Sicherung von Erfahrungswissen	n/z
14	Erlernbares Vorgehen zur Qualitätsplanung	n/z

Tabelle 4.12: Anforderungsmatrix für das Konzept *objektive Metrik*

Prinzipiell können objektive Metriken durch Werkzeuge automatisch und dadurch kostengünstig sowie reproduzierbar erhoben werden (Anf. 8 und 9 $\oplus$). Dafür müssen die Metriken in einer formalen und maschinenlesbaren Beschreibung vorliegen. Weil viele der für uns interessanten Metriken ursprünglich für Quellcode anstatt für Softwaremodelle entwickelt wurden, ist dies leider nicht immer der Fall, so dass wir zumeist lediglich informelle Beschreibungen zur Verfügung haben. Zwar ist für eine Reihe von Projektbeteiligten eine informelle Messdefinition wichtig, damit sie zu-

mindest eine grobe Vorstellung über die Metrik entwickeln können. Jedoch bleiben evtl. einige Details ungeklärt. Anhand einer formalen Messdefinition kann ein Experte weitere Informationen über Skalentyp, Skala, benötigte Diagrammtypen oder Einheit der Messung ableiten. Wenn also objektive Metriken sowohl informell als auch formal beschrieben sind, dann wirkt sich dies positiv auf die Lesbarkeit und den Detaillierungsgrad der Beschreibung aus (Anf. 11 $\oplus$).

Es existiert eine Vielzahl an objektiven Metriken, die für uns wichtiges Erfahrungswissen darstellen (Anf. 7 $\oplus$). Ein Großteil der objektiven Metriken bezieht sich auf UML-Entwurfsklassendiagramme. Leider werden bei weitem nicht alle in der UML spezifizierten Diagrammtypen oder uns bekannten Qualitätsattribute durch bestehende, objektive Metriken abgedeckt. Zudem stoßen objektive Metriken bei einigen Qualitätsattributen an ihre Grenzen und können diese nicht quantifizieren (Anf. 4 $\oplus\ominus$).

Nicht alle Metriken lassen sich problemlos mit den im Abschnitt 4.2.1 vorgestellten Qualitätsmodellen kombinieren, da die gemessenen Qualitätsattribute regelmäßig nicht bezeichnet und definiert werden. Deshalb evaluieren wir die Anforderungen 1 bis 3 und 5 sowie 6 nicht für objektive Metriken. Im Rahmen dieser ausschließlich auf objektive Metriken abgestimmten Betrachtung ist eine Bewertung der Anforderungen 12 bis 14 ebenfalls nicht zweckmäßig, da es sich bei diesem Konzept um kein Vorgehen zur Qualitätsplanung handelt. Auf Möglichkeiten der Interpretation von Metriken gehen wir im Rahmen von Indikatoren im übernächsten Abschnitt 4.3.3 ein (Anf. 10 n/z).

4.3.2 Subjektive Metriken

Messungen für subjektive Metriken benötigen das menschliche Urteilsvermögen und lassen sich infolgedessen nicht automatisieren. Subjektive Metriken werden durch einen Bewerter manuell erhoben und sind deshalb häufig teurer als objektive Metriken. Allerdings sollte auf subjektive Metriken nicht verzichtet werden. Objektive Metriken stoßen häufig bei Messungen an Grenzen, die z.B. die Relevanz oder Bedeutung von Modellelementen betreffen. Domänenexperten können dagegen auch diese Messungen durchführen, so dass der Einsatz subjektiver Metriken die objektiven Metriken ergänzt.

Subjektive Metriken werden in Software Reviews erhoben. In Software Reviews werden Entwicklungsartefakte manuell geprüft. Diese manuellen Prüfungen von Ent-

wicklungsartefakten sind in der Praxis verbreitet.

Es existieren mehrere Reviewarten unterschiedlicher Intensität. Eine *formale Inspektion*, auch unter dem Begriff *Fagan Inspektion* bekannt, ist die am stärksten strukturierte Methode unter den manuellen Prüftechniken. Ein streng spezifizierter Inspektionsprozess sorgt einerseits für eine hohe Effektivität bei der Fehlerfindung, andererseits führt die Formalität zu einem hohen Ressourcenaufwand. Ein *Walkthrough* ist im Vergleich deutlich flexibler. Durch den niedrigeren Aufwand lassen sich Walkthroughs besser auf frühe oder umfangreichere Entwicklungsartefakte anwenden. Allerdings sind Walkthroughs im Vergleich zu formalen Inspektionen nicht besonders effektiv beim Finden von Fehlern. Weitere Reviewarten, die in ihrer Intensität zwischen formalen Inspektionen und Walkthroughs liegen, sind *Management Reviews*, *Technische Reviews* und *Audits* [83].

In der Literatur zu Software Reviews wird der Begriff *Metrik* nicht verwendet. Stattdessen sind *Fragen* oder *Checklist Items* gebräuchlich, die in *Checklisten* gruppiert sind. "Checklists are usually, though not always, stated in a question form, so that a negative answer means you have found an issue" [66].

Einige der Checklist Items lassen sich als objektive Metriken darstellen und Messwerte dementsprechend durch Werkzeuge ermitteln. Andere Items entsprechen eindeutig subjektiven Metriken [174]. Checklisten unterscheiden mit wenigen Ausnahmen nicht zwischen objektiven und subjektiven Messungen (z.B. [16, 145]).

Wir verstehen Checklisten insbesondere als ideale Ergänzung zu objektiven Metriken. Objektive Metriken werden dann an ein Mess-Werkzeug übergeben und Checklisten in einem Software Review von Reviewern genutzt. Diese Checklisten enthalten dann *ausschließlich* subjektive Metriken, so dass wir die Definition für Checklisten in [177] entsprechend anpassen können.

Definition 4.6 (*Checklist*)

Eine Checkliste ist eine Liste von subjektiven Metriken, die während eines Software Reviews genutzt werden, um Mängel oder Fehler in einem Entwicklungsartefakt zu finden.

Checklisten für UML-Softwaremodelle finden sich in diversen Arbeiten, angefangen bei UML-Büchern [23] über Methodensammlungen [124] hin zu sehr allgemein formulierten Ansätzen, die zur Überprüfung von Architekturen oder Entwürfen im Allgemeinen konzipiert wurden [118, 140].

Phase	**QA**	**Diagr.-typ**	**Informelle Messbeschreibung**	**Ref**
Analyse	Vollständigkeit	UCD	Sind alle wesentlichen Anwendungsfälle festgelegt?	[124]
Analyse	Verständlichkeit	KD	Trägt die Geschäftsklasse einen sprechenden Namen, der auch für Fachabteilungen und Entscheidungsträger verständlich ist?	[135]
Analyse	Vollständigkeit	AD	Sind alle Abbruchbedingungen berücksichtigt?	[135]
Entwurf	Well-structured	KompD	Sind alle enthaltenen Elemente für das Verständnis essentiell?	[23]
Entwurf	Realisierbarkeit	Alle	Kann der Entwurf implementiert werden?	[140]
Entwurf	Korrektheit	Alle	Sind alle Sicherheitsanforderungen erfüllt?	[178]
Entwurf	Zuverlässigkeit	KD	Werden korrekte Default-Werte verwendet?	[118]
Entwurf	Flexibilität	KD	Wie hoch sind die erwarteten Änderungkosten?	[121]
Entwurf	Vollständigkeit	KD	Wie viele Benutzeranforderungen sind nicht modelliert?	[122]
Entwurf	Verständlichkeit	SC	Sind die Zustandsnamen sinnvoll bezogen auf den Anwendungskontext?	[18]

Tabelle 4.13: Subjektive Metriken für Softwaremodelle in der Analyse und im Entwurf

Subjektive Metriken sind durchgängig noch spärlicher als objektive Metriken dokumentiert. Sie werden z.B. in Form einer Fragestellung oder Forderung angegeben. Weitere Informationen wie ein Akronym, Name, Skalentyp, Skala u.v.m. fehlen.

Die Tabelle 4.13 listet einige Beispiele für subjektive Metriken auf. Zur Beschreibung der subjektiven Metriken dünnen wir unser Beschreibungsschema für objektive Metriken weiter aus und streichen die Spalte *Akronym (Name)*. Die Spalte *Phase* können wir jetzt problemlos in die Tabelle einfügen. Sie gibt die entsprechende Entwicklungsphase an, in der das Softwaremodell erstellt wird. Wir haben die *informellen Messbeschreibungen* einheitlich als Fragen formuliert.

Bewertung

Unsere Bewertung des Konzepts *subjektive Metrik* haben wir in der Tabelle 4.14 zusammengefasst. Die folgenden Ausführungen erklären die Ausprägungen für die jeweiligen Anforderungen.

Anforderung		**Konzept**
Nr.	Kurzbeschreibung	Subjektive Metrik
1	Korrekte Qualitätsziele	n/z
2	Vollständige Qualitätsziele	n/z
3	Unmissverständliche Qualitätsziele	n/z
4	Überprüfbare Qualitätsziele	$\oplus$
5	Rückverfolgbare Qualitätsziele	n/z
6	Vorwärtsverfolgbare Qualitätsziele	n/z
7	Nützliche Qualitätsprüfungen	$\oplus$
8	Zuverlässige Qualitätsprüfungen	$\ominus$
9	Wirtschaftliche Qualitätsprüfungen	$\ominus$
10	Interpretierbare Qualitätsprüfungen	n/z
11	Verständliche Qualitätsprüfungen	$\ominus$
12	Wirtschaftliche Qualitätsplanung	n/z
13	Sicherung von Erfahrungswissen	n/z
14	Erlernbares Vorgehen zur Qualitätsplanung	n/z

Tabelle 4.14: Anforderungsmatrix für das Konzept *subjektive Metrik*

Subjektive Metriken können in Checklisten gruppiert und anschließend im Rah-

men von Software Reviews genutzt werden. Sie müssen durch Menschen erhoben werden und sind deshalb häufig teurer als objektive Metriken (Anf. 9 $\ominus$). Die Einbeziehung des menschlichen Urteilsvermögens kann auch dazu führen, dass zwei Reviewer für das gleiche Softwaremodell zu verschiedenen Ergebnissen gelangen (Anf. 8 $\ominus$). Für die Messung bestimmter Qualitätsattribute kann man jedoch auf subjektive Metriken nicht verzichten (Anf. 4 und 7 $\oplus$).

Die Beispiele in Tabelle 4.13 zeigen, dass subjektive Metriken sehr unterschiedlich präzise formuliert sein können. Einige Metriken sind z.B. derart allgemein formuliert, dass sie unserer Meinung nach keine wirkliche Hilfestellung für einen Reviewer darstellen (Anf. 11 $\ominus$).

Analog zu objektiven Metriken lassen sich eine Reihe von Anforderungen für subjektive Metriken nicht sinnvoll bewerten.

4.3.3 Indikatoren

Objektive und subjektive Metriken beschreiben die Quantifizierung von Qualitätsattributen. Allerdings ist die Interpretation gemessener Werte durch Messnutzer durchaus problematisch. Wenn Metrikdefinitionen keine Interpretationsmöglichkeiten für die Beurteilung konkreter Zahlenwerte enthalten, dann sind die Messnutzer auf sich gestellt. Sie können lediglich einige Informationen über den Messkontext wie Qualitätsattribute, Diagrammtyp oder Entwicklungsphase mit ihrem eigenen, individuellen Erfahrungswissen für die Interpretation kombinieren. Bei diesem Vorgehen besteht das Risiko, dass sich Messnutzer zu stark durch ihre persönliche Stellung im Projekt beeinflussen lassen. Einerseits bewerten sie Messwerte wohlwollend und erkennen unbequeme Tatsachen nicht oder andererseits begegnen sie Messwerten zu kritisch und unterschätzen eine gute Modellierung.

Dieses Risiko kann nicht grundsätzlich vermieden, aber abgeschwächt werden, indem *Indikatoren* eingeführt werden. Indikatoren bauen auf Metriken auf und analysieren deren Messwerte mit Hilfe von Entscheidungskriterien wie Grenzwerte, Zielwerte oder Intervalle für die Einteilung in akzeptable bzw. nicht akzeptable Werte. Indikatoren überführen die quantitativen Aussagen der Metriken in qualitative Aussagen und erleichtern die Interpretation von Messungen durch Messnutzer. Indikatoren helfen höherwertige Aussagen zu generieren.

Die Entscheidungskriterien leiten sich entweder aus *Erfahungswerten bzw. empirischen Untersuchungen* oder aus *theoretischen Überlegungen* ab. Im ersten Fall

sind die Metriken bereits mehrfach erhoben, die Ergebnisse der Messreihen gesichert und ihre Bedeutung für ein Modellierungsprojekt evaluiert worden. Die Erfahrungen befähigen dann Qualitätsmanager dazu, Analysen und Entscheidungskriterien aufzustellen.

Allerdings stehen lediglich in Einzelfällen Evaluationen von Metriken zur Verfügung. Spezielle Untersuchungen von Metriken für Softwaremodelle sind äußerst selten (z.B. [102, 61, 63]). Inwiefern die ermittelten Entscheidungskriterien auf andere modellbasierte Softwareentwicklungsprojekte übertragen werden dürfen, ist zudem noch gar nicht untersucht.

Aufgrund einer fehlenden oder unvollständigen Datenbasis dürfen häufig feste Entscheidungskriterien nicht eingeführt und Indikatoren somit nicht aufgestellt werden. Stattdessen bieten sich einfache Interpretationsrichtlinien wie *je näher der Messwert an 0 heranreicht, desto besser* an, die auch in dieser Form sehr hilfreich für die Analyse möglicher Ergebnisse sein können. Der ISO/IEC 9126-3 Standard schlägt derartige Interpretationsrichtlinien auch explizit vor (vgl. Abschnitt 4.1.3). Allerdings finden sich solche Interpretationsrichtlinien nur sehr selten in Metriksammlungen. Bei diesen weichen Interpretationsrichtlinien handelt es sich im Sinne des ISO/IEC 15939 Standards um keine Indikatoren (vgl. Abschnitt 4.1.3). Vielmehr sollten sie als obligatorischer Bestandteil einer Metrikbeschreibung verstanden werden. Diese Interpretationsrichtlinien lassen sich durch Evaluationen stetig präzisieren bis schließlich ein Indikator definiert werden darf.

Im Folgenden zeigen wir drei einfache Indikator-Beispiele. Weitere Indikatoren lassen sich dem Anhang A.3 entnehmen. Das erste Beispiel bezieht sich auf die *OIF*-Metrik aus Tabelle 4.11.

Beispiel: Indikator basierend auf OIF-Metrik Ein akzeptabler Wert für die OIF-Metrik liegt [75, 30] demzufolge zwischen 20 % und 80 %. Ausgehend von diesem Erfahrungswissen für die OIF-Metrik lässt sich das Intervall für die Definition eines Indikators aufstellen:

Indikator	
Analyse	**Indikator-Wert**
$0.8 \geq$ OIF-Wert ≥ 0.2	akzeptabel
sonst	nicht akzeptabel

Neben dem erwähnten Erfahrungswissen können auch theoretische Überlegungen zu Analysen führen. Das nächste Beispiel basiert auf der *ACNI*-Metrik aus Tabelle 4.11.

Beispiel: Indikator basierend auf ACNI-Metrik Abstrakte Klassen als Blattknoten in einer Vererbungshierarchie sind überflüssig, weil sie nie ausgeführt werden können und keine einzige Klasse Eigenschaften von ihnen erbt. Deshalb ist der einzige akzeptable Wert für die objektive Metrik ACNI der Wert 0. Alle anderen Werte sind nicht akzeptabel:

Indikator	
Analyse	**Indikator-Wert**
ACNI-Wert = 0	akzeptabel
sonst	nicht akzeptabel

In den obigen Beispielen setzt sich die Skala für die Indikatoren aus den Kategorien *akzeptabel* und *nicht akzeptabel* zusammen. Dies entspricht einer Ordinalskala (akzeptabel *besser als* nicht akzeptabel). Für Indikatoren sind feinere Abstufungen prinzipiell möglich und sinnvoll. Wie stark dabei differenziert werden kann, hängt von den Erfahrungswerten und natürlich von den Informationsbedürfnissen der Messnutzer ab. Das letzte Beispiel verweist auf die Metrik *TNC* aus Tabelle 4.10.

Beispiel: Indikator basierend auf TNC Der Projektleiter möchte im Laufe des Entwicklungsprojekts immer über die aktuelle Größe des Modellsystems informiert werden und hält die Metrik TNC für geeignet. Er unterscheidet zwischen einem kleinen ($\leq$ 100 Klassen), mittleren ($\leq$ 1000 Klassen), großen ($\leq$ 4000 Klassen) und riesigen ($>$ 4000 Klassen) Modellsystem.

Indikator	
Analyse	**Indikator-Wert**
TNC-Wert $\leq$ 100	klein
100 $<$ TNC-Wert $\leq$ 1000	mittel
1000 $<$ TNC-Wert $\leq$ 4000	groß
sonst	riesig

Bis jetzt haben wir recht einfache Analysen für Indikatoren gezeigt. Analysen sind aber nicht auf den Vergleich von Metrik-Werten mit Grenzwerten beschränkt. Analysen können auch zwei Metriken miteinander vergleichen oder mehrere Metriken durch logische Operatoren kombinieren. Einige Beispiele für kompliziertere Indikatoren existieren für Software [168].

Erfahrungswissen und Metrikevaluationen muss jedoch kritisch begegnet werden. Für eine ältere und viel untersuchte Metrik *Depth of Inheritance of a class (DIT)* (vgl. Tab. 4.11) widersprechen sich die Ergebnisse zum Teil. Eine Untersuchung von Daly et al. [43] legt nahe, dass Systeme mit einer Vererbungstiefe von drei signifikant leichter zu warten sind im Vergleich mit Systemen, in denen keine Vererbung eingesetzt wurde. Dagegen besagt das Experiment von Harrison et al. [76], dass Systeme ohne Vererbung leichter zu verstehen und zu warten sind als solche mit drei oder fünf Vererbungsebenen. Die Experimente wurden von einer anderen Forschergruppe wiederholt [170, 143]. Die Ergebnisse bestätigen keine der vorherigen und deuten stattdessen daraufhin, dass die Vererbungstiefe nicht als Indikator für die Wartungskosten taugt.

Für viele Metriken gibt es keine empirischen Studien hinsichtlich ihrer Validität und hinsichtlich adäquater Entscheidungskriterien, die in Analysen verwendet werden können. Und auch wenn Studien vorliegen, sollten die Ergebnisse nur unter bestimmten Voraussetzungen wiederverwendet werden. Im Grunde ist nämlich nicht zu erwarten, dass identische Grenzwerte für Metriken über alle Anwendungsgebiete gültig sind.

Bewertung

Unsere Bewertung des Konzepts *Indikator* haben wir in der Tabelle 4.15 zusammengefasst. Die folgenden Ausführungen erklären die Ausprägungen für die jeweiligen Anforderungen.

Indikatoren überführen die quantitativen Aussagen der Metriken in qualitative Aussagen und erleichtern die Interpretation von Messungen durch Messnutzer (Anf. 10 $\oplus$). Indikatoren werden durch Analysen bestimmt. Analysen können sich dabei aus Metriken, Entscheidungskriterien, Vergleichsoperatoren und logischen Operatoren zusammensetzen. Erfahrungswissen und empirische Untersuchungen zeigen häufig sinnvolle Analysen auf (Anf. 13 $\oplus$). Allerdings ist nicht zu erwarten, dass identische Grenzwerte für Metriken über alle Anwendungsgebiete gültig sind.

Anforderung		**Konzept**
Nr.	Kurzbeschreibung	Indikator
1	Korrekte Qualitätsziele	n/z
2	Vollständige Qualitätsziele	n/z
3	Unmissverständliche Qualitätsziele	n/z
4	Überprüfbare Qualitätsziele	n/z
5	Rückverfolgbare Qualitätsziele	n/z
6	Vorwärtsverfolgbare Qualitätsziele	n/z
7	Nützliche Qualitätsprüfungen	n/z
8	Zuverlässige Qualitätsprüfungen	n/z
9	Wirtschaftliche Qualitätsprüfungen	n/z
10	Interpretierbare Qualitätsprüfungen	$\oplus\oplus$
11	Verständliche Qualitätsprüfungen	n/z
12	Wirtschaftliche Qualitätsplanung	n/z
13	Sicherung von Erfahrungswissen	$\oplus$
14	Erlernbares Vorgehen zur Qualitätsplanung	n/z

Tabelle 4.15: Anforderungsmatrix für das Konzept *Indikator*

Wenn nicht auf feste Entscheidungskriterien zurückgegriffen werden kann, sollten unbedingt Interpretationsrichtlinien wie *je näher der Messwert an 0 heranreicht, desto besser* zur Beschreibung einer Metrik eingebunden werden, die Projektbeteiligten zumindest Tendenzen vorgeben (Anf. 10 $\oplus$).

Eine gesonderte Bewertung der anderen Anforderungen bietet sich für Indikatoren nicht an. Je nachdem, ob Indikatoren auf subjektiven oder objektiven Metriken basieren, ergeben sich die weiteren Bewertungen.

Zusammenfassung

Messungen sind für die Bestimmung des Erfüllungsgrads von Qualitätszielen von herausragender Bedeutung. Bei Messungen unterscheiden wir zwischen Metriken und Indikatoren.

Objektive und subjektive Metriken quantifizieren Qualitätsattribute und ergänzen sich. Objektive Metriken lassen sich formal beschreiben und infolgedessen von Werkzeugen automatisch erheben. Subjektive Metriken können nicht automatisch, sondern nur manuell erhoben werden, aber für die Messung bestimmter Qualitätsattribute sind subjektive Messungen unverzichtbar. Bei der Verwendung von bestehenden Metriken ist Skepsis angebracht. Metriken sind häufig mangelhaft oder gar nicht evaluiert und ihre Aussagekraft ist unklar. Die uns bekannten Metriksammlungen decken nicht alle UML-Diagrammtypen und nicht alle Qualitätsattribute ab. Trotzdem stellen die bestehenden Metriksammlungen wichtiges Erfahrungswissen dar.

Indikatoren bauen auf Metriken auf und überführen deren quantitative Aussagen in qualitative Aussagen, um Fehlinterpretationen von Messnutzern vorzubeugen. Auch das für Analysen notwendige Erfahrungswissen muss immer kritisch gesehen werden, denn identische Grenzwerte für Metriken über alle Anwendungsgebiete sind unwahrscheinlich. Wenn keine Indikatoren aufgestellt werden können, sollten zumindest Interpretationsrichtlinien dokumentiert werden.

4.4 Interpretation und Verbesserung

In diesem Abschnitt stellen wir grundsätzliche Möglichkeiten der Interpretation von Messergebnissen und darauf basierenden Verbesserungen von Softwaremodellen kurz vor und schließen damit den Kreislauf der analytischen Qualitätssicherung. Wir möchten aber explizit darauf hinweisen, dass die *Aufbereitung von Messergebnissen in der Interpretationsphase* und *die Auswahl konkreter Verbesserungsstrategien* nicht Teil unserer Zielsetzung ist. Somit entfällt auch eine Bewertung der im Folgenden vorgestellten Konzepte.

Nachdem wichtige Messungen im Rahmen der Qualitätsplanung festgelegt wurden, können diese Messungen anschließend für konkrete Softwaremodelle durchgeführt werden. Projektbeteiligte verwenden und interpretieren die resultierenden

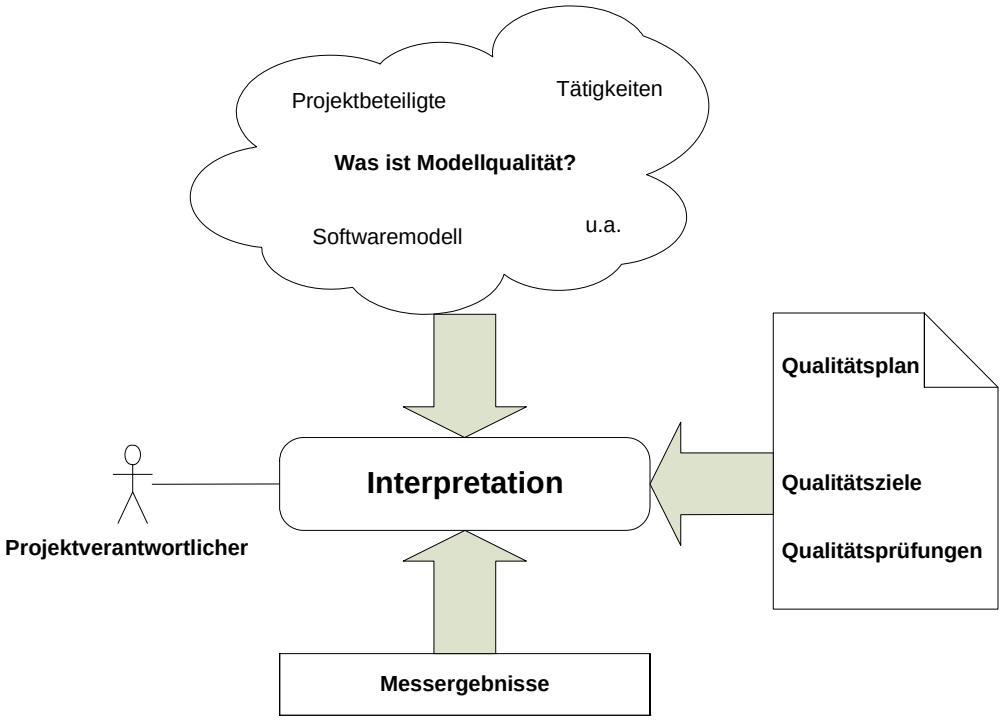


Abbildung 4.11: Wichtige Informationen für die Interpretation von Messergebnissen

Messergebnisse. Dazu sind die aufgestellten Qualitätsziele und der Kontext des untersuchten Softwaremodells von Bedeutung (siehe Abb. 4.11).

Auf dieser Informationsbasis beurteilen die Projektverantwortlichen die Modellqualität und treffen anschließend die Entscheidung, ob das Softwaremodell für die darauffolgende Phase freigegeben werden darf oder ob es überarbeitet werden muss. Falls die Qualität des Softwaremodells von den Projektverantwortlichen als nicht ausreichend beurteilt wird, stellt sich als nächstes die Frage, *was wie verbessert werden soll.*

Mögliche Verbesserungsstrategien hängen sehr stark von einem nicht erreichten Qualitätsziel ab, das ein Qualitätsproblem repräsentiert. Verbesserungsstrategien für Softwaremodelle lassen sich wie folgt differenzieren: Die Verbesserung der Qualität eines Softwaremodells kann durch *Refaktorisieren* oder durch *Evolution* erfolgen.

Refaktorisieren ist der Prozess, ein Modellsystem so zu verändern, ohne dass das spezifizierte und nach außen sichtbare Verhalten geändert wird, das Softwaremodell aber eine bessere Struktur erhält (in Anlehnung an [59]). Refaktorisieren eignet sich insbesondere für die Behebung von Entwurfsproblemen wie *zu starker Kopp-*

lung oder *Gottklassen*. Für das Refaktorisieren von Softwaremodellen existieren eine Reihe unterschiedlicher Ansätze [101, 117, 163]. In [150] ist ein Qualitätszyklus bestehend aus den Aktivitäten Messung, Diagnose und Refaktorisieren beschrieben, der sich für eine kontinuierliche Verbesserung von Softwaremodellen anbietet, deren Semantikdefinition formal durch CSP-OZ [58] gegeben ist. Dieser Ansatz nutzt Indikatoren für die Diagnose von Entwurfsproblemen. Die Entwurfsprobleme wiederum sind mit konkreten Refaktorisierungen verknüpft. Die Diagnose in diesem Ansatz stellt somit eine Möglichkeit der Interpretation von Messergebnissen dar.

Unter Evolution versteht man die Weiterentwicklung von Softwaremodellen, die sowohl Änderungen an der Struktur als auch am Verhalten umfasst (in Anlehnung an[116]). Ist beispielsweise ein relevanter Use Case nicht modelliert und das Softwaremodell dementsprechend unvollständig, müssen für die Vervollständigung Informationen hinzugefügt werden. In anderen Fällen kann ein Qualitätsproblem darin bestehen, dass zu viel modelliert wurde und das Softwaremodell reduziert werden muss, indem Modellelemente entfernt werden. Auch wenn Inkonsistenzen auftreten, ist die Verbesserung des Softwaremodells schwierig. Damit eine Inkonsistenz auftreten kann, müssen mindestens zwei Modellelemente A und B widersprüchlich sein. Allerdings kann die Quelle für eine Inkonsistenz sehr verschieden sein. Folgende Varianten sind grundsätzlich denkbar: Erstens können entweder Modellelement A oder Modellelement B falsch sein. Oder zweitens sind Modellelement A und Modellelement B falsch.

Auch wenn die Quelle für eine Inkonsistenz geklärt ist, kann man noch nicht absehen, ob Informationen hinzugefügt oder entfernt werden müssen. Im Gegensatz zum Refaktorisieren werden während der Evolution von Softwaremodellen Informationen nicht nur umstrukturiert, sondern es können Informationen hinzugefügt und entfernt werden. Dies entspricht einem gewöhnlichen Konstruktionsprozess mit einigen Zusatzinformationen über Qualitätsprobleme und die Vorgängerversion des zu konstruierenden Softwaremodells. Deshalb kann die Evolution von Softwaremodellen nicht so konsequent systematisiert werden wie Refaktorisierungen.

Mit der Interpretation der Messwerte wird primär die Bewertung des Softwaremodells angestrebt und somit auf ein Endekriterium für die Modellierung abgezielt. Darüber hinaus können die Messergebnisse auch dazu verwendet werden, Schwächen im angewandten Entwicklungsprozess oder in der Organisation aufzudecken. Wenn beispielsweise sehr viele syntaktische Fehler für ein Softwaremodell gemessen wer-

den, kann dies in Zukunft entweder durch zusätzliche Mitarbeiterschulungen oder durch den Einsatz eines anderen Modellierungswerkzeugs verbessert werden.

Allerdings sollten die Messergebnisse nicht zur Beurteilung einzelner Modellierer verwendet werden [80]. Modellierer können ihre Handlungsweise auf die verwendeten Messungen hin optimieren und verfehlen evtl. dadurch das eigentliche Modellierungsziel, gute Software zu spezifizieren. Sie können gezielt Schwächen in den Messungen ausnutzen, um besonders gute Messergebnisse für sich zu erreichen. Dies führt aber nicht unbedingt zu wirklich besseren Softwaremodellen. Wenn z.B. die Messung *Anzahl Klassen* zur Bewertung der Produktivität von Modellierern genutzt wird, werden die Modellierer Delegation und Vererbung weniger einsetzen. Später in der Implementierung kann dieses Vorgehen zu redundantem Code führen.

Zusammenfassung

Für die Interpretation der Messergebnisse durch Projektbeteiligte sind die aufgestellten Qualitätsziele und der Kontext des untersuchten Softwaremodells von Bedeutung. Nicht erreichte Qualitätsziele stellen Qualitätsprobleme dar, die durch die Anwendung von Verbesserungsstrategien gelöst werden können. Die Verbesserungsstrategien hängen eng mit den gegebenen Qualitätsproblemen zusammen. Refaktorisierungen eignen sich zur Lösung von Entwurfsproblemen. Viele andere Qualitätsprobleme lassen sich nur durch eine weitere Iteration im Konstruktionsprozess lösen. In diesem Konstruktionsprozess stehen den Modellierern weitere Informationen wie bestehende Qualitätsprobleme und die Vorgängerversion des zu konstruierenden Softwaremodells zur Verfügung. Zudem lassen sich die Messergebnisse für eine Analyse von Schwächen in dem angewandten Entwicklungsprozess und in der Organisation nutzen.

4.5 Weitere Analyseverfahren

Die zuvor eingeführten analytischen Verfahren zur Bewertung von Softwaremodellen eignen sich grundsätzlich für alle Softwaremodelle *unabhängig* von ihrem Formalisierungsgrad. Werden im Softwareentwurf formale Spezifikationstechniken verwendet, kann für die Qualitätsbewertung ebenfalls auf spezialisierte Analyseverfahren wie die *Verifikation* und die *Simulation* zurückgegriffen werden.

Einerseits sind die Kosten und Schwierigkeiten bei der Einführung formaler Spezifikationstechniken in den Softwareentwicklungsprozess sehr hoch. Andererseits führt ihre Anwendung zu einer geringeren Fehlerzahl in der ausgelieferten Software und ist somit für die Entwicklung kritischer Anwendungsbereiche gerechtfertigt, bei denen Systemeigenschaften wie Sicherheit, Zuverlässigkeit und Schutz sehr wichtig sind [159].

In der aktuellen Version unseres Lösungsansatzes, den wir im Kapitel 5 vorstellen, sind die beiden Analyseverfahren Verifikation und Simulation nicht integriert. Prinzipiell halten wir ihre Integration jedoch für sinnvoll und thematisieren diese Erweiterung im Ausblick dieser Arbeit. Deshalb führen wir die beiden Analyseverfahren an dieser Stelle kurz ein.

4.5.1 Verifikation

Mit Verifikation wird der formale Beweis der Korrektheit eines Softwaremodells gegenüber den formulierten Anforderungen bezeichnet, in dem mathematische oder auf der Mathematik basierende Techniken der Informatik angewendet werden. Somit ist ein zentraler Vorteil der Verifikation die Vollständigkeit der erzielten Ergebnisse.

Als notwendige Voraussetzung für die Anwendung von Verifikationsverfahren muss sowohl eine formale Beschreibung des Softwaremodells als auch eine formale Beschreibung der Anforderungen vorliegen. Einen Überblick über formale Spezifikationssprachen bietet [180]. Die konkreten Verifikationsverfahren hängen dabei eng mit der zu Grunde liegenden formalen Semantik der eingesetzten Spezifikationssprachen zusammen.

Typische Fragestellungen, die durch die Verifikation beantwortet werden können, betreffen die Deadlockfreiheit, Lebendigkeit, Sicherheit, Beschränktheit, Erreichbarkeit oder Soundness eines Softwaremodells. Zudem kann die Korrektheit von Modelltransformationen in einem modellbasierten Entwurf geprüft werden.

Die Verifikation von Softwaremodellen lässt sich allerdings nicht gut skalieren, d.h. für kleine wie große Softwaremodelle einsetzen. Beispielsweise besteht im *Model Checking* als automatisiertes Verifikationsverfahren die erhebliche Herausforderung der Zustandsexplosion. Eine Zustandsexplosion tritt bei der Analyse von Softwaremodellen auf, die viele nebenläufige Komponenten oder große bzw. unendliche Datentypen enthalten [40]. Es existieren eine Reihe von Ansätzen, die der Zustandsexplosion entgegenwirken können (z.B. Symbolic Model Checking [34, 114], Partial Order Reduction [67, 92] oder Data Abstraction Techniques [39]). Allerdings steigt zunehmend die Komplexität der zu entwickelnden Softwaresysteme und in der Folge auch die Komplexität der Softwaremodelle, so dass häufig lediglich einzelne Bereiche effizient verifiziert werden können. Lässt sich die Verifikation nicht mehr effizient durchführen, bietet sich in diesen Fällen die Simulation als Analyseverfahren an.

4.5.2 Simulation

Während der Simulation werden virtuelle Experimente auf dem Computer durchgeführt [33]. Die Simulation ist ein höchst komplexer Prozess, bestehend aus einer Folge mehrerer Schritte, die in verschiedenen Feedback-Schleifen typischerweise mehrfach durchlaufen werden. Dabei wird ein Softwaremodell, das eine vereinfachende formale Beschreibung eines Systemausschnitts darstellt, durch konkrete Werte stimuliert, darauf basierend die Simulation berechnet und wichtige Daten protokolliert. Anschließend werden die Ergebnisse interpretiert und auf das simulierte System übertragen. Auf diese Weise lassen sich mehrere Szenarien durchspielen.

Beschreibt beispielsweise ein Softwaremodell eine Systemarchitektur, so lassen sich durch die Simulation kritische Bereiche (z.B. ein Flaschenhals) in der Architektur rechtzeitig vor dem Beginn der Implementierung finden. Ein weiteres typisches Einsatzszenario für die Simulation in der Informatik ist die Analyse von Abläufen wie Datenflüsse und Workflows.

Die Simulation von Softwaremodellen weist Ähnlichkeiten zum Testen von Software auf. Clarke et al. bemerken hierzu, dass “simulation and testing both involve making experiments before deploying the system in the field” [40]. Im Gegensatz zur Verifikation werden sowohl bei der Simulation als auch beim Testen nicht alle möglichen Systemzustände und Abläufe betrachtet. Deshalb kann die Simulation lediglich Fehler oder Probleme aufdecken, jedoch nicht ihre Abwesenheit zeigen.

Kapitel 5

Modell-Qualitäts-Pläne für Softwaremodelle

Nachdem wir im letzten Kapitel die Stärken und Defizite bestehender Konzepte für die analytische Qualitätssicherung hinsichtlich unserer in Kapitel 2 aufgestellten Anforderungen dargelegt haben, stellen wir in diesem Kapitel den *Modell-Qualitäts-Plan(MQP)-Ansatz* vor.

Der MQP-Ansatz beschreibt ein Vorgehen zur Qualitätsplanung speziell für die Bewertung von Softwaremodellen. Die dadurch erstellten Qualitätspläne bilden die Grundlage für die Durchführung der Messungen und dokumentieren zudem nützliche Informationen zur Interpretation der Messwerte und anschließenden Entscheidungsfindung. Das Ergebnis unseres Qualitätsplanungsansatzes bezeichnen wir als Modell-Qualitäts-Plan (MQP). Zur Erstellung eines MQPs definieren wir ein MQP-Metamodell und einen MQP-Prozess. Das MQP-Metamodell beschreibt relevante Konzepte sowie deren Zusammenhänge und spezifiziert die möglichen Inhalte eines MQPs. Der MQP-Prozess spezifiziert das Vorgehensmodell zur Erstellung eines MQPs und dient als Leitfaden bei der Qualitätsplanung.

Der MQP-Ansatz kombiniert die in Kapitel 4 vorgestellten Konzepte, um auf diese Weise deren Vorteile zu vereinen. Im Folgenden motivieren wir kurz unser weiteres Vorgehen, indem wir das Potential einer solchen Kombination aufzeigen. Tabelle 5.1 stellt die einzelnen Bewertungen für die Konzepte gegenüber. Die grau hinterlegten Zellen heben die Bewertungen der Konzepte hervor, die wir durch eine geeignete Kombination der bestehenden Konzepte grundsätzlich erzielen können.

Qualitätsmodelle eignen sich hervorragend zur Dokumentation von Qualitätszie-

Anforderung		**Konzepte**				
Nr.	Kurzbeschreibung	Generisches Qualitätsmodell	GQM	Messungen		
				Objektive Metrik	Subjektive Metrik	Indikator
1	Korrekte Qualitätsziele	⊖	⊕⊕	n/z	n/z	n/z
2	Vollständige Qualitätsziele	⊕	⊖	n/z	n/z	n/z
3	Unmissverständliche Qualitätsziele	⊕	⊖	n/z	n/z	n/z
4	Überprüfbare Qualitätsziele	⊕⊖	⊕	⊕⊖	⊕	n/z
5	Rückverfolgbare Qualitätsziele	⊖	⊖	n/z	n/z	n/z
6	Vorwärtsverfolgbare Qualitätsziele	⊕	⊕	n/z	n/z	n/z
7	Nützliche Qualitätsprüfungen	⊕	⊕⊕	⊕	⊕	n/z
8	Zuverlässige Qualitätsprüfungen	⊖	⊖	⊕	⊖	n/z
9	Wirtschaftliche Qualitätsprüfungen	⊖	⊖	⊕	⊖	n/z
10	Interpretierbare Qualitätsprüfungen	⊖	⊖	n/z	n/z	⊕⊕
11	Verständliche Qualitätsprüfungen	⊕⊖	⊕⊖	⊕	⊖	n/z
12	Wirtschaftliche Qualitätsplanung	n/z	⊖⊖	n/z	n/z	n/z
13	Sicherung von Erfahrungswissen	n/z	⊕⊖	n/z	n/z	⊕
14	Erlernbares Vorgehen zur Qualitätsplanung	n/z	⊕⊖	n/z	n/z	n/z

Tabelle 5.1: Anforderungsmatrix für bestehende Konzepte

len. Es existieren bereits mehrere generische Qualitätsmodelle für Modellqualität, die wichtiges Erfahrungswissen kapseln. Allerdings müssen generische Qualitätsmodelle auf den Kontext eines Softwaremodells hin angepasst werden und für ein solches Zuschneiden besteht kein angemessenes Vorgehen.

Die Goal Question Metric (GQM) beschreibt ein praktikables Vorgehen, in dem mit der Unterstützung des Projektteams zielorientiert ein projektspezifisches Qualitätsmodell sowie entsprechende Messungen systematisch abgeleitet werden. Allerdings unterscheiden sich die bestehenden Qualitätsmodelle für Modellqualität von den Qualitätsmodellen, die nach Anwendung des GQM-Ansatzes vorliegen. Deshalb integrieren wir die Erstellung von Qualitätsmodellen vollständig in GQM, so dass keine strukturellen Unterschiede zu existierenden Qualitätsmodellen bestehen und deren Wiederverwendung problemlos möglich ist.

Allerdings berücksichtigen sowohl Qualitätsmodelle als auch der GQM-Ansatz die Besonderheiten von Softwaremodellen bei der Messdefinition unzureichend. Zudem referenzieren Qualitätsmodelle nur wenige Messungen, die in einer Qualitätsplanung wiederverwendet werden können. Deshalb lassen wir auch gezielt Ansätze zu Messungen auf Softwaremodellen in den MQP-Ansatz einfließen. Für Messungen verwenden wir ein Beschreibungsschema, das die detaillierte Dokumentation und die Wiederverwendung von Messungen aus Metriksammlungen ermöglicht. Wie die Tabelle 5.1 nahelegt, bleiben auch bei einer Kombination aller Vorteile der Konzepte einige Schwächen bestehen. Um diesen Defiziten entgegenzuwirken, werden wir darüber hinaus noch weitere Ergänzungen vornehmen.

Auf die Kombination der bestehenden Konzepte und die Ergänzung fehlender Konzepte gehen wir in den nächsten Abschnitten ein. Wir beschreiben den MQP-Ansatz in drei Stufen mit absteigendem Detaillierungsgrad. Zuerst fassen wir in Abschnitt 5.1 die grundlegenden Bausteine des Ansatzes kurz zusammen. Anschließend erklären wir in Abschnitt 5.2 die wesentlichen Schritte des MQP-Ansatzes und erstellen ein erstes, einfaches Beispiel für einen MQP. Ausführlich erläutern wir jeden einzelnen Schritt unseres Ansatzes zusammen mit dem entsprechenden Ausschnitt aus dem MQP-Metamodell im Abschnitt 5.3. In Abschnitt 5.4 führen wir ein Regelkonzept für MQPs ein, um dem Qualitätsmanager eine Möglichkeit an die Hand zu geben, sein Erfahrungswissen zu sichern und um die Wiederverwendbarkeit bestehender MQPs zu erleichtern. *Gute* Qualitätspläne sind die Voraussetzung für eine *erfolgreiche* Qualitätssicherung. Deshalb widmen wir uns in Abschnitt 5.5 einigen

Konzepten zur Qualitätssicherung von Qualitätsplänen.

5.1 Grundlegende Bausteine

Die Goal Question Metric (GQM) ist ein sehr generelles Vorgehen, um *nützliche* Messungen zu identifizieren. Die verfügbare Literatur ist umfangreich und diverse Evaluationen unterstreichen die Praxistauglichkeit des mit GQM verbundenen top down Vorgehens.

In Leitfäden zur GQM-Anwendung wird vorgeschlagen, Informationsbedürfnisse wichtiger Vertreter des Projektteams durch Interviews zu erheben und mittels Zielen und korrespondierenden Fragen zu dokumentieren. Die gezielte Einbindung von Projektbeteiligten in einen Dialog ist für eine erfolgreiche Qualitätsplanung von großer Bedeutung, denn Projektbeteiligte wie Projektmanager, Modellierer, Analysten oder Softwarearchitekten nutzen erstellte Qualitätspläne für die Durchführung von Qualitätsprüfungen und für die Entscheidungsfindung. Ihre Informationsbedürfnisse und die von ihnen verwendeten Begriffe sollten sich in einem Qualitätsplan niederschlagen.

Im Rahmen unserer Zielsetzung steht die Bestimmung des Erfüllungsgrads von Qualitätszielen im Fokus und dies entspricht einer besonderen Form von Informationsbedürfnissen. Die Kombination aus Zielen und Fragen gibt an, *was wichtig zu messen ist*. In GQM werden diese Informationsbedürfnisse als Grundlage für die Auswahl von Messungen verwendet. Die Dokumentation der Messungen beantwortet dann die Frage *wie gemessen wird*.

Aus diesen Gründen beurteilen wir den GQM-Ansatz als geeignetes Vorgehen für die Erstellung von Qualitätsplänen und verwenden ihn als Basis für unsere Lösung.

1. Baustein

Wir verwenden GQM als Basis für die Qualitätsplanung von Softwaremodellen. In GQM werden Informationsbedürfnisse durch Interviews mit wichtigen Vertretern des Projektteams erhoben und in Form von Zielen und korrespondierenden Fragen erfasst. Fragen wiederum werden mit Messungen in Verbindung gesetzt.

Die Identifizierung von Informationsbedürfnissen stellt in GQM eine sehr kreative Tätigkeit dar. Es ist allgemein davon auszugehen, dass kein Verfahren garantieren

kann, alle relevanten Qualitätsziele zu finden. Deshalb sind weitere Hilfestellungen von Bedeutung. Neben den Projektbeteiligten gibt es eine weitere, wichtige Informationsquelle, die ein Qualitätsmanager für die Identifizierung von Qualitätszielen für Softwaremodelle beachten sollte:

Der *Kontext des zu prüfenden Softwaremodells*, also dessen Einbettung in den Softwareentwicklungsprozess (vgl. dazu auch Definition 3.4 in Abschnitt 3.6), besitzt ein großes Potential und sollte nach unserer Auffassung Berücksichtigung finden. Wir möchten diese Forderung anhand zweier einfacher Beispiele kurz motivieren:

- Sind in einem Softwaremodell verschiedene *Diagrammtypen* eingebunden, so sollten diese Diagramme auch konsistent sein, damit sich modellierte Informationen nicht widersprechen. Die verwendeten Diagrammtypen weisen auf potentielle Inkonsistenzen hin und eignen sich folglich als Quelle für Qualitätsziele.

- Sind in einem Softwaremodell keine Sichtbarkeiten von Attributen und Operationen spezifiziert, weil es sich um ein Softwaremodell einer frühen *Entwurfsphase* handelt (z.B. Anforderungsspezifikation oder Analyse), dann ist die Forderung nach Datenkapselung an dieser Stelle verfrüht und nicht korrekt. Die Sichtbarkeitsinformationen liegen nicht vor.

Am Kontext des Softwaremodells lässt sich z.B. erkennen, welche Modellierungssprache und Diagrammtypen für die Modellerstellung verwendet werden und in welcher Entwicklungsphase das Softwaremodell erstellt wird. Wie anhand der Beispiele gezeigt, kann der Kontext ausgenutzt werden, um Qualitätsziele zu identifizieren, und sollte folglich festgehalten werden.

Aufgrund seiner Ausrichtung auf die gesamte Organisation bzw. auf ein Projekt als Ganzes berücksichtigt der GQM-Ansatz die Besonderheiten und somit den Kontext von Softwaremodellen nicht. Deshalb spezialisieren wir den GQM-Ansatz und erweitern das Vorgehen um die *Kontextdokumentation* des zu prüfenden Softwaremodells und stellen dadurch heraus *in welchem Kontext gemessen wird*. Die Dokumentation des Kontextes lagern wir der Identifizierung der Informationsbedürfnisse vor. Die Feststellung von Kontextinformationen erfordert Wissen über den im Projekt angewandten Entwicklungsprozess. Ansonsten lässt sich der Kontext aber problemlos und ohne entscheidenden subjektiven Einfluss erheben. Die Kontextinformationen lassen sich dann zum systematischen Ableiten von Qualitätszielen verwenden und

verringern den Anteil notwendiger kreativer Arbeit. Die Kontextinformationen stellen somit eine zweite Informationsquelle neben den Projektbeteiligten dar, die durch einen Qualitätsmanager genutzt werden kann.

2. Baustein

Wir verwenden die Dokumentation des Kontextes von Softwaremodellen zum systematischen Ableiten von Informationsbedürfnissen und können auf diese Weise neben den Interviews wichtiger Vertreter des Projektteams eine zweite Informationsquelle etablieren.

In GQM können Qualitätscharakteristiken wie Wartbarkeit zwar im Rahmen der Zieldefinition in Form eines Qualitätsschwerpunktes eingebunden werden. Aber Qualitätsmodelle lassen sich dadurch weder definieren noch vollständig in GQM einbinden. Dabei sind Qualitätsmodelle eine Standardtechnik für die Dokumentation von Qualitätszielen und es gibt eine ganze Reihe von Qualitätsmodellen speziell für Softwaremodelle. Wir halten die Wiederverwendung von diesen Qualitätsmodellen für sinnvoll, da (1.) Erfahrungswissen in ihnen gebunden ist, (2.) die zu messenden Attribute definiert und (3.) auch einige Messungen enthalten sind. Qualitätsmodelle beschreiben also *was gemessen wird* und in einigen Fällen *wie gemessen wird*. Deshalb binden wir als weiteren Schritt in das GQM-Vorgehen die Erstellung von Qualitätsmodellen ein, die strukturell den bestehenden Qualitätsmodellen für Modellqualität gleichen. Durch diese GQM-Erweiterung entsteht weiterer Anpassungsbedarf. Messungen werden in GQM mit Fragen und im Gegensatz zu Qualitätsmodellen nicht mit Qualitätsattributen in Verbindung gesetzt. Deshalb weisen wir Fragen einen Qualitätsschwerpunkt in Form von Qualitätsattributen zu und verknüpfen auf diese Weise Messungen direkt mit Qualitätsattributen und transitiv mit Fragen. Das Qualitätsmodell verfeinert durch die Definition des Qualitätsschwerpunkts von Zielen und Fragen die Informationsbedürfnisse des GQM-Ansatzes und verknüpft diese Informationsbedürfnisse indirekt mit Messungen.

3. Baustein

Wir integrieren Qualitätsmodelle vollständig in den GQM-Ansatz. Qualitätsmodelle präzisieren Ziele sowie Fragen und fungieren als Bindeglied zu Messungen.

Zur Bestimmung des Erfüllungsgrads von Qualitätszielen sehen sowohl Qualitätsmodelle als auch GQM Metriken als Messtechnik vor, vernachlässigen jedoch die Bedeutung von Indikatoren. Zudem sind die verwendeten Beschreibungsschemata für Metriken unbefriedigend. Sie bestehen häufig nur aus einem Metriknamen und einer textuellen Beschreibung der Messvorschrift. Deswegen ergänzen wir zum einen die Ebene der Messungen um das Konzept *Indikator*, damit auf Basis eines MQPs auch qualitative Aussagen getroffen werden können und Projektverantwortliche während der Interpretation der Messwerte besser unterstützt werden. Zum Zweiten synthetisieren wir bestehende Beschreibungsschemata für Metriken und Indikatoren, um dadurch spezielle Arbeiten zu Metriken und Indikatoren einordnen und einheitlich erfassen zu können. Diese Beschreibungsschemata für Metriken und Indikatoren ermöglichen in einem MQP eine sehr detaillierte Dokumentation der Messungen. Dadurch verbessern wir die Verständlichkeit von Messungen und erleichtern die Interpretation von Messwerten.

4. Baustein

Wir erweitern die Ebene der Messungen um Indikatoren und verwenden für alle Messungen ein detailreiches Beschreibungsschema.

Ein Problem im Zusammenhang mit Qualitätsplänen besteht darin, dass ihre Erstellung sehr aufwändig ist. Durch korrekt durchgeführte Qualitätsbewertungen schon während der Planung, Modellierung und Entwicklung eines Softwareprojekts können Mängel früh erkannt und beseitigt werden. Dadurch können Entwicklungszeit und Kosten deutlich gesenkt werden. Dennoch ist ein sehr aufwändiges Verfahren zur Durchführung dieser Qualitätsbewertungen nur schwer vermittelbar. Für Entscheidungsträger zeigt sich zunächst der hohe Aufwand, der mit der Erstellung von Qualitätsplänen verbunden ist, der daraus resultierende Nutzen ist jedoch zunächst nicht offensichtlich.

Bei der Betrachtung von Qualitätsplänen für Softwaremodelle zeigt sich schnell, dass sich viele Informationen in verschiedenen Qualitätsplänen wiederholen. Im MQP-Vorgehen werden Ziele und daraus abgeleitete Fragen ausgehend von den zu untersuchenden Entwicklungsartefakten und deren Kontexten definiert. Mithilfe von Qualitätscharakteristiken und Qualitätsattributen wird dann ein Qualitätsmodell aufgebaut, welches letztendlich über Indikatoren und Metriken konkret gemessen werden kann. Die einzelnen Schritte bauen dabei aufeinander auf. Während viele

Ziele und die dafür zu beantwortenden Fragen direkt vom Kontext des zu untersuchenden Softwaremodells abhängen, bauen die weiteren Schritte auf den Fragen auf und versuchen diese zu beantworten. Dadurch hängen alle weiteren Schritte implizit vom gegebenen Kontext ab. Ziele und Fragen wiederholen sich dabei in ähnlichen Kontexten. Die zu formulierenden Ziele und Fragen und die dazugehörigen Qualitätscharakteristiken und Qualitätsattribute sowie die Indikatoren und Metriken stellen Erfahrungswissen dar, welches für diesen Kontext gesichert werden sollte, damit dieses Wissen dementsprechend auch in anderen Qualitätsplänen wiederverwendet werden kann. Der Kontext des zu untersuchenden Softwaremodells ist also der entscheidende Einflussfaktor für einen Qualitätsplan.

Um Qualitätspläne möglichst effizient anhand von vorhandenem Erfahrungswissen über bestimmte Kontexte aufzubauen, bieten wir ein einfaches Regelkonzept an, das anhand von Kontextfaktoren entscheidet, welche Elemente in den Qualitätsplan eingefügt werden sollten.

Die Regeln funktionieren nach einem Wenn-Dann-Prinzip und gliedern sich dementsprechend in zwei Teile. Der erste Teil (Wenn-Teil) legt fest, welche Kontextinformationen vorliegen müssen, damit die im zweiten Teil (Dann-Teil) festgelegten Informationsbedürfnisse, Qualitäten und Messungen eingefügt werden. Mit Hilfe der Regeln lassen sich Erfahrungen in der Qualitätsplanung einfach sichern und wiederverwenden.

5. Baustein

Wir etablieren ein Regelkonzept, mit dem Erfahrungswissen für die Qualitätsplanung auf Basis von Kontextfaktoren gesichert und wiederverwendet werden kann.

Der Kontext erfüllt noch eine weitere Funktion. Bei der Interpretation der Messwerte können wichtige Kontextinformationen helfen, die Ergebnisse im *richtigen Licht* zu sehen.

Qualitätsziele werden in unserem Ansatz vielschichtig beschrieben: Zuerst dienen uns Ziele und Fragen dazu, die Schwerpunkte der Qualitätsbewertung festzustellen. Das Qualitätsmodell verfeinert anschließend die Ziele und Fragen, indem die verwendeten Qualitätscharakteristiken und Qualitätsattribute definiert und in Beziehung gesetzt werden. Auf dieser Stufe sind Qualitätsziele durch ein Netz von definierten Qualitätseigenschaften beschrieben. Durch die Einführung von Indikatoren und Metriken erreichen wir ihre *Überprüfbarkeit*.

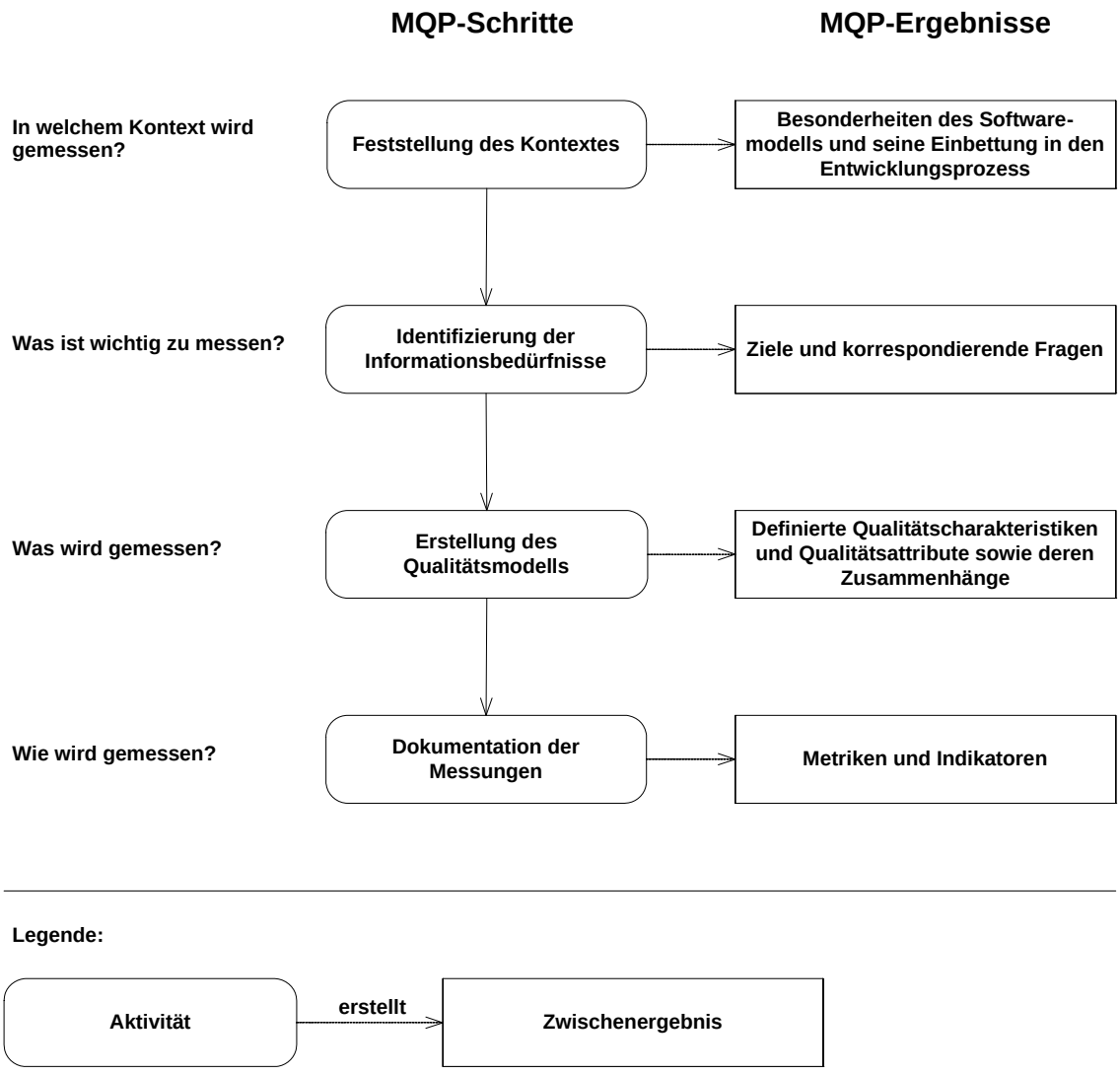


Abbildung 5.1: MQP-Ansatz: Fundamentale Schritte und ihre Ergebnisse

Die Abbildung 5.1 gibt einen Überblick über die fundamentalen Schritte und die jeweiligen Ergebnisse. Die durchgezogenen Pfeile zeigen dabei das *typische* top down Vorgehen bei der Anwendung des MQP-Ansatzes. Abweichungen von diesem Vorgehen sind jedoch grundsätzlich vorgesehen. Das MQP-Vorgehen ist ein inkrementeller und iterativer Prozess, damit schrittweise Verfeinerungen und Korrekturen jederzeit möglich sind.

5.2 Kurzes Beispiel

Im vorherigen Abschnitt haben wir die wesentlichen Schritte des MQP-Ansatzes eingeführt. Jetzt stellen wir einen konkreten MQP anhand eines einfachen, aber durchgängigen Beispiels vor.

5.2.1 Feststellung des Kontextes

In dem vorliegenden Beispiel soll für ein Softwaremodell ein Qualitätsplan ausgehend von seinem Kontext erstellt werden. Unter realen Projektbedingungen ist das zu bewertende Softwaremodell während der Qualitätsplanung noch nicht entwickelt, so dass auch wir für den im Folgenden entwickelten MQP bewusst auf ein konkretes und durch Diagramme visualisiertes Softwaremodell verzichten. Stattdessen widmen wir uns zuerst dem Kontext des zu bewertenden Softwaremodells. Dieser Kontext des Softwaremodells ist durchaus vor der Modellierung bekannt.

Der Kontext beschreibt das Umfeld, in dem sich das zu überprüfende Softwaremodell befindet. Die Feststellung des Kontextes erleichtert die Berücksichtigung von Besonderheiten des betrachteten Softwaremodells in seinem Entwicklungsumfeld. Die Kontextinformationen beschränken sich in diesem Beispiel auf die verwendete Modellierungssprache, die Entwicklungsphase, den Modellierungszweck, eine Rolle, die das Softwaremodell nutzt, und eingesetzte Diagrammtypen (siehe Tab. 5.2). Weitere Kontextfaktoren führen wir in Abschnitt 5.3.1 ein.

Die Informationen über die *Modellierungssprache* sind wichtig für die Überprüfung der Syntax, also die Bewertung des Softwaremodells bzgl. seiner Sprachdefinition. Anstatt der UML 2.1.1 können hier z.B. auch UML-Profile, unternehmens- oder auch werkzeugspezifische Varianten der UML angegeben werden. Die Modellierungssprache lässt zudem auf mögliche Formalisierungen der Metriken schließen. Z.B. eignet sich im Falle von UML-Modellen die Object Constraint Language (OCL)

Kontextfaktor		**Ausprägung**
Modellierungssprache		UML 2.1.1
Entwicklungsphase		Entwurf
Modellierungszweck		Grundlegendes Design des zu implementierenden Softwaresystems spezifizieren
Rolle(n)	Name Beschreibung	Softwarearchitekt Trifft high-level Entwurfsentscheidungen und bestimmt technische Standards
Diagrammtype(n)		Klassendiagramm Statechart

Tabelle 5.2: Kontextinformationen eines zu prüfenden Softwaremodells

für Metrikdefinitionen, weil OCL eine standardisierte Erweiterung der UML darstellt und in OCL geschriebene Ausdrücke auf den in der UML definierten Typen (z.B. Klassen, Schnittstellen, etc.) beruhen.

Die *Entwicklungsphase* gibt Aufschluss darüber, wie implementierungsnah das Softwaremodell ist. Handelt es sich beispielsweise um ein Entwurfsmodell, dann können bereits mehr Qualitätsziele an das Softwaremodell gestellt werden, die auch für Quell-Code gelten. Für ein Modell des Problembereichs hätten viele der folgenden Qualitätsziele dagegen keine Gültigkeit.

Die UML spezifiziert insgesamt 13 *Diagrammtypen*. Alle uns bekannten Modellierungsprojekte nutzen nur eine kleine Auswahl davon. Dabei schränkt diese Auswahl die verwendbaren Sprachelemente immer implizit ein. Die Informationen über verwendete Diagrammtypen sind notwendig um festzustellen, ob die gewünschten Sichten konsistent und tatsächlich modelliert sind.

Ohne einen *Modellierungszweck* würde ein Modell erst gar nicht erstellt werden. Der Modellierungszweck gibt den Nutzen des Softwaremodells an und beantwortet die Frage *warum und wozu modelliert wird*. Der Modellierungszweck kann sogar für gleiche Diagrammtypen variieren. UML-Klassendiagramme werden als Modellierung eines Problembereichs in der Anforderungsanalyse, als Analysedokument, Design-Spezifikation, Datenmodell u.v.m. genutzt. In dem vorliegenden Beispiel geben wir den Modellierungszweck ausschließlich für das gesamte Softwaremodell und nicht für die einzelnen Diagrammtypen an.

Qualität ist nichts Absolutes, sondern relativ zu gegebenen Qualitätszielen. Qua-

litätsziele hängen von Personen ab, die die Softwaremodelle nutzen. Diese Nutzung der Softwaremodelle bestimmt sich durch spezielle *Rollen*, die durch Personen im Entwicklungsprojekt wahrgenommen werden. Deshalb halten wir wichtige Rollen wie *Softwarearchitekt* für das aktuelle Beispiel während der Qualitätsplanung fest und beschreiben sie kurz. Liegen mehrere Rollen vor, können diese im nächsten Schritt dazu eingesetzt werden, um verschiedene Qualitätssichten zu berücksichtigen.

5.2.2 Identifizierung der Informationsbedürfnisse

Den Kontext nutzen wir zur Identifizierung wesentlicher Ziele: Das Softwaremodell soll syntaktisch korrekt bzgl. der Modellierungssprache UML 2.1.1 sein, eine *gute* Wartbarkeit des zu implementierenden Softwaresystems spezifizieren und Klassendiagramme sowie Statecharts sollen konsistente Sichten auf ein Softwaremodell darstellen (siehe Tabelle 5.3).

Wir haben für unser laufendes Beispiel lediglich die Rolle des Softwarearchitekten explizit spezifiziert. Daneben gibt es aber immer das Projektteam als Ganzes, das nicht als spezielle Rolle definiert werden muss. Dadurch können wir für die obigen Ziele zwei Qualitätssichten benennen: Das Projektteam (incl. Softwarearchitekt) ist an syntaktischer Korrektheit und Intramodellkonsistenz interessiert. Für den Softwarearchitekten ist darüber hinaus die Wartbarkeit von Bedeutung. Auf diese Weise können wir auf Rollen spezialisierte Qualitätssichten in der Qualitätsplanung berücksichtigen.

Die richtigen Fragen bzgl. der Ziele zu stellen ist für eine weitere Präzisierung der Ziele wichtig. Allerdings erfordert diese Tätigkeit viel Domänenwissen und sollte durch den Qualitätsmanager und die Vertreter des Projektteams gemeinsam erledigt werden. Der Anteil notwendiger kreativer Arbeit bleibt hier verhältnismäßig hoch. Für das laufende Beispiel sollte beispielsweise der Softwarearchitekt interviewt werden.

Tabelle 5.4 enthält eine kleine Auswahl wichtiger Fragen zum Ziel *Wartbarkeit*. Diesen Fragen sind bereits die entsprechenden Qualitätsschwerpunkte in Form von zu messenden Qualitätsattributen zugeordnet.

Wir erklären anhand der Frage *Enthält das Softwaremodell große Klassen?* exemplarisch den Zusammenhang zwischen Zielen und Fragen und zeigen auf, dass hierbei eine Transferleistung zu erbringen ist.

Das Design eines Softwaresystems sollte Wartungsarbeiten erleichtern. Fowler zu-

Zieldimension	**Ausprägung**
Untersuchungsobjekt	Analysiere das Softwaremodell
Zweck	zum Zweck der Bewertung
Qualitätscharakteristik	in Bezug auf syntaktische Korrektheit
Sichtweise	aus der Sicht des Projektteams
Kontext	im Kontext der Modellierungssprache UML 2.1.1
Untersuchungsobjekt	Analysiere das Softwaremodell
Zweck	zum Zweck der Bewertung
Qualitätscharakteristik	in Bezug auf Wartbarkeit
Sichtweise	aus der Sicht des Softwarearchitekten
Kontext	im Kontext der Modellierungssprache UML 2.1.1, der Entwicklungsphase Entwurf, des Modellierungszwecks Design-Spezifikation und der Diagrammtypen Klassendiagramm und Statechart
Untersuchungsobjekt	Analysiere das Softwaremodell
Zweck	zum Zweck der Bewertung
Qualitätscharakteristik	in Bezug auf Intramodellkonsistenz
Sichtweise	aus der Sicht des Projektteams
Kontext	im Kontext der Modellierungssprache UML 2.1.1, der Diagrammtypen Klassendiagramm und Statechart

Tabelle 5.3: Identifizierte Ziele

Ziel	**Frage**	**Qualitätsattribut**
Wartbarkeit	Enthält das Softwaremodell große Klassen?	Große Klasse
	Sind die Statecharts komplex?	Komplexität
	Schlagen Klassen ihr Erbe aus?	Ausgeschlagenes Erbe

Tabelle 5.4: Ausgewählte Fragen zum Ziel *Wartbarkeit*

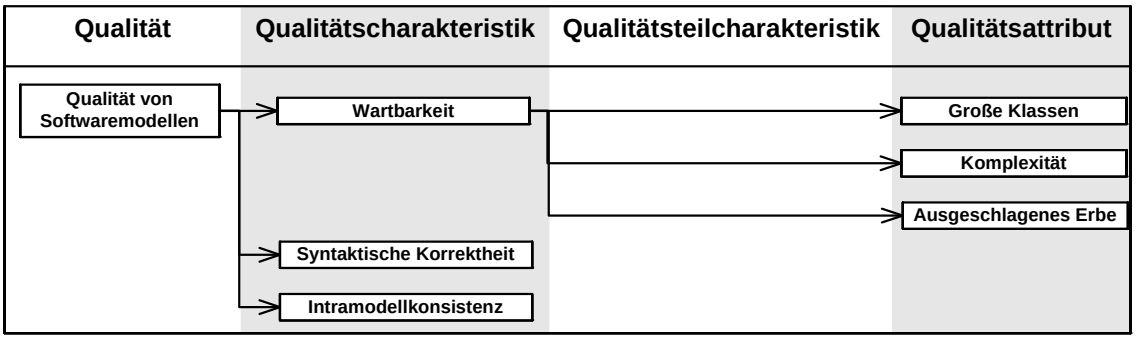


Abbildung 5.2: Rahmen für ein Qualitätsmodell

folge beeinflussen zu große Klassen die Wartbarkeit eines Softwaresystems negativ und er schlägt zur Behebung Refaktorisierungen vor [59]. Damit in der Implementierung dieser Aufwand erst gar nicht entsteht, sollten Entwurfsmodelle diese Anforderungen bereits berücksichtigen und somit keine großen Klassen enthalten. Den Qualitätsschwerpunkt zu dieser Frage bezeichnen wir mit dem entsprechenden Qualitätsattribut *Große Klasse*. In den nächsten Schritten des MQP-Vorgehens wird dieses Qualitätsattribut definiert und mit Messungen assoziiert.

Den Schritt *Identifizierung von Informationsbedürfnissen* schließen wir für das laufende Beispiel ab. Wir möchten mit dem Beispiel nicht den Eindruck erwecken, als müssten alle Ziele aus den gegebenen Kontextfaktoren abgeleitet werden. Die Kontextinformationen dienen als eine Hilfestellung und ersetzen das Wissen der Projektbeteiligten nicht. Somit sind auch Ziele möglich, die keinen Kontextfaktoren zugeordnet sind. Die Menge der Fragen ist für das laufende Beispiel stark begrenzt. Wenn wir davon ausgehen, dass ein Qualitätsmanager einen Personentag für die Interviews mit Vertretern des Projektteams verwendet, wird die Menge der Fragen schnell sehr groß. In Abschnitt 5.3.2 gehen wir auf die Stärken des MQP-Ansatzes bei der Strukturierung einer großen Anzahl an Fragen ein.

5.2.3 Erstellung des Qualitätsmodells

Basierend auf den Zielen und Fragen können wir einen ersten Rahmen für das Qualitätsmodell systematisch ableiten (siehe Abbildung 5.2). Der Rahmen für das Qualitätsmodell wird durch die Qualitätsschwerpunkte der Ziele und Fragen gegeben.

Ausgehend von diesem Rahmen kann das Qualitätsmodell anschließend weiter vervollständigt werden. Durch die Informationsbedürfnisse sind die Qualitätscharak-

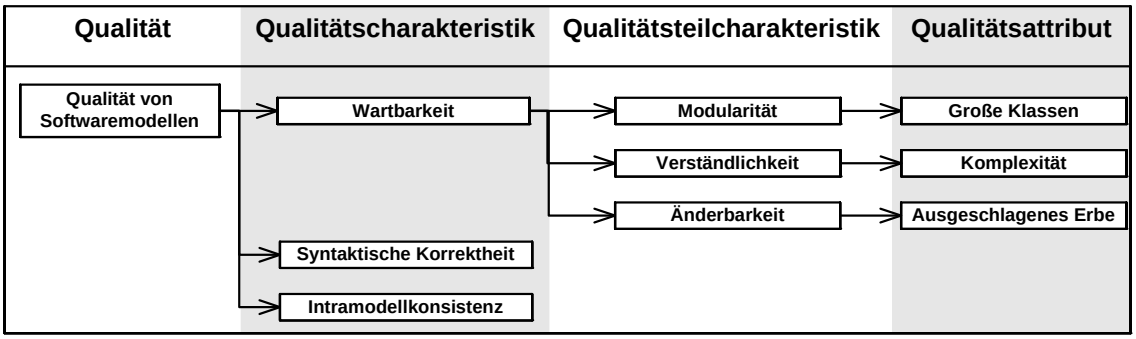


Abbildung 5.3: Qualitätsmodell für das laufende Beispiel

teristiken und Qualitätsattribute bezeichnet und in Beziehung gesetzt, so dass ersichtlich ist, wie sich die Qualitätscharakteristiken und Qualitätsattribute untereinander beeinflussen. Für die Qualitätscharakteristiken und Qualitätsattribute müssen jetzt noch Definitionen und evtl. Synonyme angegeben werden. Zudem können weitere Qualitätscharakteristiken eingeführt werden, um den Zusammenhang zwischen Qualitätscharakteristiken und Qualitätsattributen weiter zu präzisieren und bestehende Qualitätscharakteristiken zu gruppieren. In unserem Beispiel ergänzen wir *Modularität, Verständlichkeit und Änderbarkeit* als Qualitätsteilcharakteristiken zwischen den Qualitätsattributen und der Qualitätscharakteristik Wartbarkeit. Für das laufende Beispiel sind die verfeinerten Zusammenhänge zwischen Qualitätscharakteristiken und Qualitätsattributen in Abbildung 5.3 gegeben. Die entsprechenden Definitionen befinden sich in Tabelle 5.5.

Der Qualitätsmanager kann das projektspezifische Qualitätsmodell mit bestehenden Qualitätsmodellen vergleichen und stellt dabei u.U. fest, dass das projektspezifische Qualitätsmodell unvollständig ist. Er möchte dann weitere Qualitätscharakteristiken und Qualitätsattribute hinzufügen und konsequenterweise die Informationsbedürfnisse überarbeiten. Derartige Rückschritte sind im MQP-Vorgehen grundsätzlich vorgesehen. Für das vorliegende Qualitätsmodell verzichten wir auf einen Vergleich mit bestehenden Qualitätsmodellen und gehen zum Schritt *Dokumentation der Messungen* über.

5.2.4 Dokumentation der Messungen

Qualitätsattribute lassen sich durch Metriken quantifizieren und durch Indikatoren qualifizieren. Für das laufende Beispiel benötigen wir Messungen für die Qualitäts-

Qualität	**Definition**
Intramodellkonsistenz	Der Grad der Widerspruchsfreiheit von Informationen, die durch Diagramme modellierten sind.
Syntaktische Korrektheit	Die Übereinstimmung zwischen dem Softwaremodell und seiner Sprachdefinition.
Wartbarkeit	Das Potential eines Softwaremodells, geändert zu werden.
Modularität	Modularisierung ist die Forderung, dass ein Modellelement oder Teilmodell auch nur einen inhaltlichen oder technisch unabhängigen Aspekt behandelt.
Verständlichkeit	Der benötigte Aufwand, um ein Softwaremodell zu verstehen.
Änderbarkeit	Das Potential eines Softwaremodells modifiziert zu werden, um Verbesserungen, Fehlerbeseitigungen oder Anpassungen an Umgebungsänderungen zu ermöglichen.
Große Klasse	Eine Klasse, die zu viele Features hat.
Komplexität	Der Schwierigkeitsgrad für Vorhersagen, die das Verhalten und Zusammenspiel des modellierten Systems betreffen.
Ausgeschlagenes Erbe	Unterklassen erben Methoden und Daten ihrer Oberklassen, die sie gar nicht brauchen oder nicht haben wollen.

Tabelle 5.5: Definitionen von Qualitätscharakteristiken und Qualitätsattributen

attribute *Große Klasse*, *Komplexität* und *Ausgeschlagenes Erbe*. Tabelle 5.7 ordnet den Qualitätsattributen jeweils eine Messung zu.

In Abschnitt 4.3 haben wir bereits eine Reihe von Messungen vorgestellt. Die Metrik *CC* quantifiziert das Qualitätsattribut *Komplexität* und das Qualitätsattribut *Ausgeschlagenes Erbe* kann durch die Metrik *SIX* gemessen werden (vgl. Metriken in Tabelle 4.11). Für die Quantifizierung des Qualitätsattributs *Große Klasse* konnten wir allerdings in der Literatur keine geeigneten Messungen finden. Fowler hat zwar den Begriff *Große Klasse* geprägt. Jedoch schlägt er keine fixen Kriterien für die Identifizierung vor. Aus diesem Grund muss hier erneut das Fachwissen des Qualitätsmanagers und eines Softwarearchitekten bemüht werden. Beispielsweise könnten sie zu der Ansicht kommen, dass eine große Klasse u.a. dann vorliegt, wenn sie im Verhältnis zu den anderen modellierten Klassen signifikant mehr Operationen aufweist. Im Rahmen dieses Beispiels wird für ein angemessenes Verhältnis zwischen der Anzahl der eigenen Operationen einer Klasse und dem Durchschnitt über alle Klassen des UML-Entwurfsmodells die Obergrenze 3 angenommen. Wenn also eine Klasse A über dreimal mehr Operationen verfügt als der Durchschnitt, dann wertet die Metrik *NLC* diese Klasse A als *große Klasse* (siehe Tabelle 5.6).

In der Metrik *NLC* sind die Kennzahlen *NOA* und *AOA* referenziert. *NOA* bestimmt die absolute Anzahl an verfügbaren Attributen und Operationen einer Klasse. *AOA* berechnet die durchschnittliche Anzahl an verfügbaren Attributen und Operationen in einem Modell. Diese Zerlegung einer Metrik in weitere Basiskennzahlen bzw. abgeleitete Kennzahlen verkürzt eine OCL-Anfrage und erhöht dadurch ihre Lesbarkeit. Zudem können solche Kennzahlen auch in anderen Metriken verwendet werden, so dass Berechnungen während der Durchführung der Qualitätsprüfung nur einmalig durchgeführt werden müssen. Die weiteren Details der Kennzahlen *NOA* und *AOA* führen wir hier nicht auf und verweisen stattdessen auf [25].

Zu diesem Zeitpunkt der Qualitätsplanung kann man erkennen, dass nicht alle Qualitätsziele durch Anwendung des vorgestellten MQPs überprüft werden. In der Regel lässt sich ein MQP daraufhin noch erweitern. Aber einige Qualitätsattribute wie z.B. Validität (Korrektheit des Softwaremodells bzgl. der Anforderungsspezifikation) lassen sich nicht immer durch objektive Metriken beurteilen und werden deshalb evtl. nicht mit Messungen assoziiert, da eine Überprüfung einfach zu teuer oder mit den zur Verfügung stehenden Ressourcen nicht möglich wäre. Trotzdem muss das gesamte Qualitätsmodell bei der Interpretation der Messergebnisse betrachtet

<table>
<tr><th colspan="3">Objektive Metrik</th></tr>
<tr><td colspan="2">Name</td><td>Number of Large Classes</td></tr>
<tr><td colspan="2">Acronym</td><td>NLC</td></tr>
<tr><td rowspan="2">Measurement-method</td><td>Informal Definition</td><td>Total number of Large Classes in the model. A class is a Large Class if the total number of its operations and its attributes is more than three times higher than the average for all Classes.</td></tr>
<tr><td>Formal Definition</td><td><pre>
context Model::NLC(): Integer
post: result = AllClasses() -> iterate (
 elem:Class; acc: Integer = 0 |
        if ((elem.NOA() / self.AOA()) >
         3)
                then acc + 1
        endif
</pre></td></tr>
<tr><td colspan="2">Type of Measurement</td><td>Objective</td></tr>
<tr><td colspan="2">Unit of Measurement</td><td>Model</td></tr>
<tr><td colspan="2">Scope</td><td>Class diagram</td></tr>
<tr><td colspan="2">Scale (Type)</td><td>Integers from zero to infinity (Ratio)</td></tr>
<tr><td colspan="2">Interpretation</td><td>The closer to 0, the better.</td></tr>
</table>

Tabelle 5.6: Definition einer Metrik

QA	Name	Details
Große Klasse	Number of Large Classes (NLC)	Tab. 5.6
Komplexität	Cyclomatic Complexity (CC)	Tab. 4.11
Ausgeschlagenes Erbe	Specialization IndeX Metric (SIX)	Tab. 4.11

Tabelle 5.7: Zuordnung der Messungen zu Qualitätsattributen

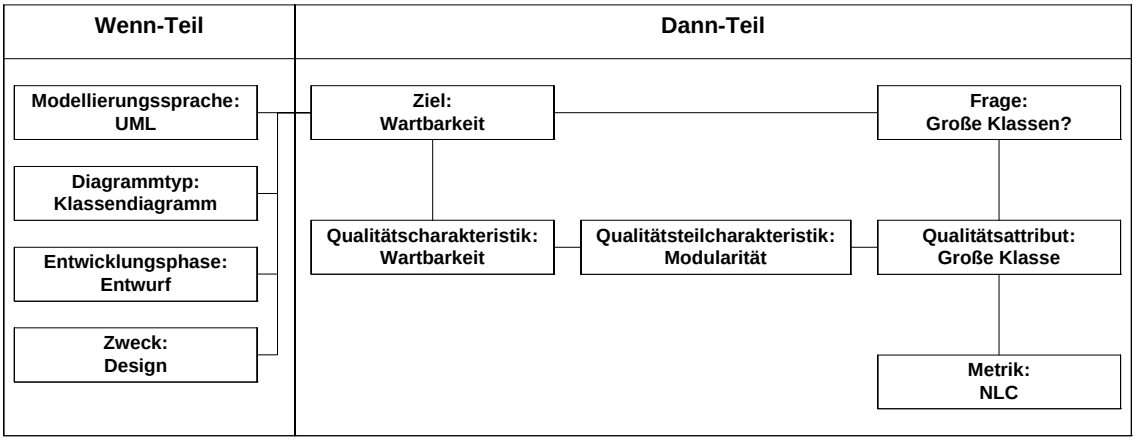


Abbildung 5.4: Beispiel für eine MQP-Regel

werden. Die Projektbeteiligten wissen dann, dass nicht für alle Qualitätsziele Messungen durchgeführt wurden und können diese Information bei der Interpretation der zur Verfügung stehenden Messwerte berücksichtigen.

Bei der Interpretation der Messwerte wird diese Folge von MQP-Schritten quasi rückwärts durchlaufen. Eine Messung bildet ein Qualitätsattribut auf eine Skala ab. Wird die Messung schließlich durchgeführt, generiert sie einen Wert für ein Qualitätsattribut. Dieser Wert wird durch die Projektverantwortlichen im Hinblick auf gegebene Informationsbedürfnisse und Kontextinformationen interpretiert. Wichtige Begriffe für die Interpretation sind im Qualitätsmodell definiert.

5.2.5 Erfahrungswissen sichern

Nachdem wir anhand eines kurzen Beispiels gezeigt haben, wie man vom Kontext über Ziele, Fragen und dem Qualitätsmodell hin zu Messungen kommt, ist es darüber hinaus wünschenswert, Erfahrungswissen für zukünftige Qualitätsplanungen sichern zu können, die unter einem ähnlichem Kontext stattfinden. Der Zusammenhang zwischen dem Kontext und dem Rest eines MQPs kann in folgenden Qualitätsplanungen für eine Teilautomatisierung bei der Erstellung von MQPs genutzt werden. Dafür bieten wir ein einfaches Regelkonzept für MQPs an.

Abbildung 5.4 enthält ein Beispiel für eine MQP-Regel, die auf dem laufenden Beispiel basiert. Die Regel soll für den gegebenen Kontext (Wenn-Teil) festhalten, dass es sinnvoll ist, die Metrik *NLC* zu erheben. Der detaillierte Zusammenhang zwischen Metrik und dem Kontext wird ebenfalls gespeichert. Diese MQP-Beispielregel

ist vereinfacht dargestellt. Messbeschreibung, Definitionen und Ähnliches haben wir nicht dargestellt.

Solche Regeln lassen sich später einfach wiederverwenden. Am Wenn-Teil ist ersichtlich, inwiefern dieser Teil eines MQPs verwendet werden kann. Dadurch lassen sich MQPs deutlich schneller aufbauen und der Anteil kreativer Arbeit wird durch Wiederverwendung reduziert.

5.3 Prozess und Struktur im Detail

In diesem Abschnitt stellen wir den Modell-Qualitäts-Plan(MQP)-Ansatz zur kontextsensitiven Qualitätsplanung von Softwaremodellen im Detail vor. Der MQP-Ansatz besteht aus einem inkrementellen und iterativen Prozess, dessen mögliche Ergebnisse durch ein zugehöriges Metamodell strukturiert sind.

- Der MQP-Prozess beschreibt die *systematische Vorgehensweise* und dient als Leitfaden bei der Qualitätsplanung. Er legt fest, welche Aktivitäten von wem bei der Bearbeitung eines Schrittes durchzuführen sind, wie die einzelnen Schritte aufeinander aufbauen, welche Dokumente für die Bearbeitung eines Schrittes benötigt und welche Teile des MQPs erstellt werden.

- Das MQP-Metamodell spezifiziert die wichtigsten *strukturellen Konzepte*, die in der Qualitätsplanung benötigt werden. Das Metamodell schränkt die möglichen Inhalte eines MQPs ein, die durch Anwendung des MQP-Prozesses formuliert werden können.

Das MQP-Metamodell und der MQP-Prozess sind eng ineinander verflochten. Jeder einzelne Schritt des MQP-Prozesses bezieht sich auf dazugehörende Ausschnitte des MQP-Metamodells.

Das MQP-Metamodell formalisieren wir basierend auf der Meta Object Facility (MOF) Spezifikation [129]. MOF wurde entwickelt, um Metamodelle zu spezifizieren und beschreibt dafür selbst ein Meta-Metamodell, auf dessen Grundlage auch die UML definiert ist. MOF enthält eine Reihe von essentiellen Sprachkonzepten wie z.B. *Class*, *Enumeration*, *Association*, *Role*, *Property* und *Primitive Type*. Mit Hilfe dieser Sprachkonzepte beschreiben wir wichtige Konzepte für die Qualitätsplanung wie z.B. Qualitätsattribut und Metrik. Damit die von uns eingeführten

Konzepte von Nutzern des MQP-Ansatzes richtig verstanden und bei seiner Anwendung richtig verwendet werden, definieren wir die wichtigsten von ihnen. Das MQP-Metamodell modelliert definierte Konzepte und deren Zusammenhänge und beschreibt somit auch eine Ontologie für die Domäne *analytische Qualitätssicherung von Softwaremodellen* (vgl. [49, 2, 3, 55]).

Wir stellen das MQP-Metamodell mit Hilfe von Diagrammen dar, wie sie auch in der Sprachdefinition des UML-Metamodells Verwendung finden [130, 131]. Diese Visualisierung entspricht einer reduzierten Variante der sehr bekannten UML-Klassendiagramme. Angaben über Sichtbarkeit, Operationen, Interfaces u.ä., die man häufig in Klassendiagrammen findet, sind jedoch für das MQP-Metamodell nicht sinnvoll, da es kein Softwaresystem beschreibt.

Der MQP-Prozess besteht aus vier aufeinander aufbauenden Schritten, die eine inkrementelle Verfeinerung der Qualitätsplanung ermöglichen. Für jeden dieser Schritte sind alle denkbaren Rückkopplungen zu einem der vorangegangenen Schritte grundsätzlich vorgesehen. Diese textuelle Beschreibung des Ablaufs halten wir für ausreichend präzise und verzichten deshalb auf eine formale Darstellung.

Die eigentliche Schwierigkeit im MQP-Prozess liegt in der Bearbeitung eines einzelnen Schrittes. Hierfür bieten wir einem Qualitätsmanager konkrete Hilfestellungen an, indem wir z.B. optionale und notwendige Eingangsdokumente, Richtlinien zur Ausführung u.v.m. angeben. Bei der Beschreibung eines Schrittes haben wir uns an bestehenden Dokumentationen zum GQM-Ansatz [1, 141, 69] sowie am Software & Systems Process Engineering Meta-Model (SPEM) [132] orientiert.

Das Ergebnis des MQP-Prozesses bezeichnen wir als einen Modell-Qualitäts-Plan (MQP). Der Aufbau eines MQPs wird durch das MQP-Metamodell beschrieben. Aus technischer Perspektive wird bei der Anwendung des MQP-Prozesses ein Modell erstellt. Die im Modell enthaltenen *ValueSpecifications* und *Links* sind über dem MQP-Metamodell getypt. Dieses Modell instanziiert das MQP-Metamodell und lässt sich in abstrakter Syntax darstellen (vgl. dazu auch [130]). Diagramme, die die abstrakte Syntax dieses Modells visualisieren, entsprechen im Wesentlichen den UML-Objektdiagrammen. Ein MQP lässt sich folglich als eine Menge von Objektdiagrammen darstellen. Die Visualisierung als Objektdiagramm ist aber i.d.R. unpraktisch. Die zahlreichen in einem MQP miteinander verknüpften Informationen bilden ein Netz und können nicht kompakt dargestellt werden. Deshalb haben wir für das MQP-Beispiel in Abschnitt 5.2 als konkrete Syntax die Verwendung von

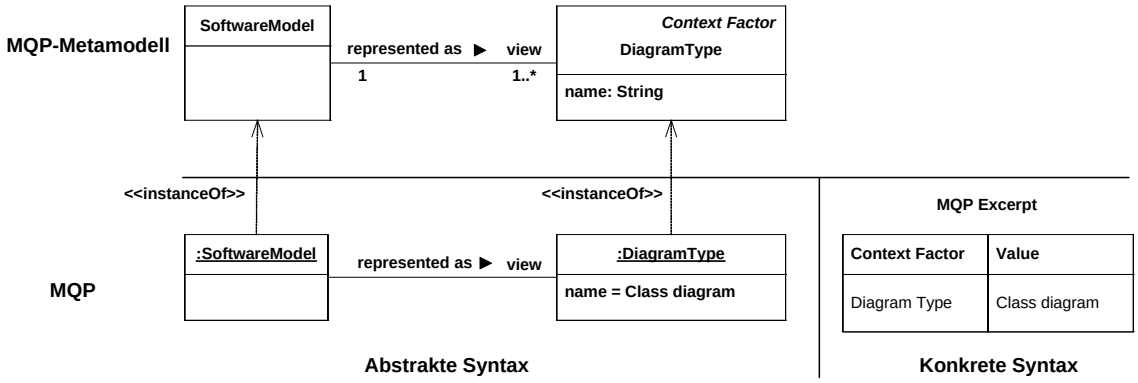


Abbildung 5.5: Zusammenhang zwischen dem MQP-Metamodell, der abstrakten und konkreten Syntax eines MQPs

Tabellen und Grafiken vorgezogen. Die konkrete Syntax führen wir durch Beispiele ein, beschreiben sie ansonsten nicht weiter. Der Zusammenhang zwischen dem MQP-Metamodell, der abstrakten Syntax und der konkreten Syntax eines MQPs ist in Abbildung 5.5 exemplarisch dargestellt.

Nachdem ein Qualitätsmanager den MQP-Prozess erfolgreich angewendet und einen MQP erstellt hat, kann die Qualitätsprüfung auf Basis des erstellten MQPs erfolgen. Der MQP wird auf ein konkretes Softwaremodell angewandt und Messergebnisse berechnet. Das MQP-Metamodell stellt auch für die Qualitätsprüfung Attribute bereit, die zum Zeitpunkt der Qualitätsplanung nicht belegt sind. Damit wir die entsprechenden Attribute unterscheiden können, kennzeichnen wir sie durch zwei verschiedene Stereotypen:

- Der Stereotyp «planning» bedeutet, dass das Attribut während der Qualitätsplanung belegt wird. Für alle Attribute des MQP-Metamodells ist dieser Stereotyp die Standardeinstellung (default).

- Der Stereotyp «execution» besagt, dass das Attribut erst beim Anwenden des MQPs belegt wird und während der Qualitätsplanung ungebunden bleibt. Die betreffenden Klassen und Attribute markieren wir explizit mit diesem Stereotyp.

Alle Package-, Klassen- und Rollennamen sind ohne Leerzeichen geschrieben, damit wir später im Abschnitt 5.5 OCL nutzen können, um weitere Einschränkungen

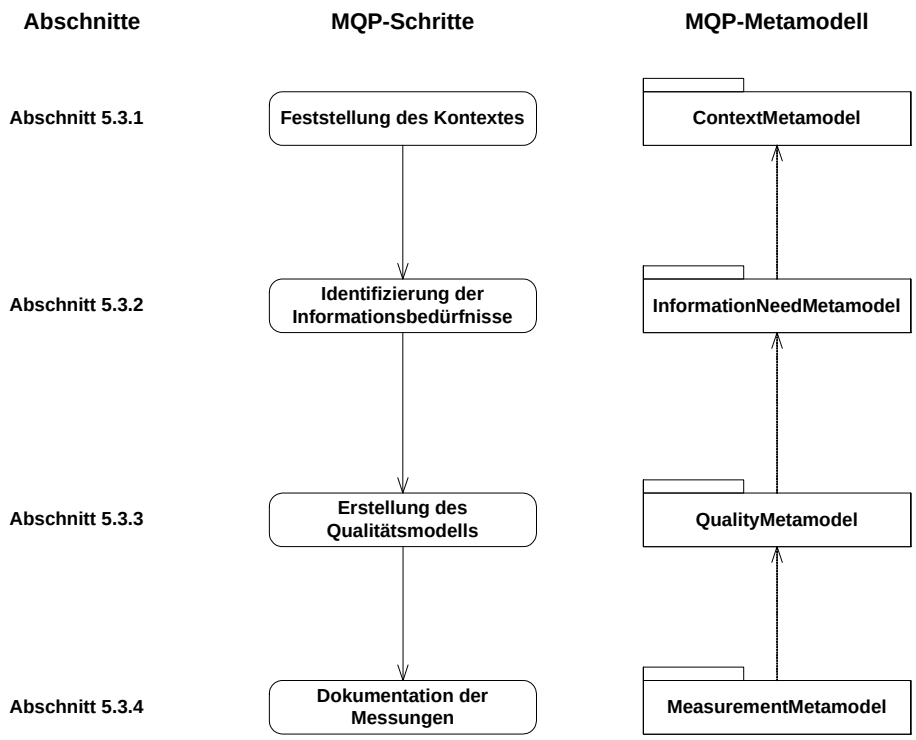


Abbildung 5.6: Strukturierung des MQP-Ansatzes: Überblick

für MQPs zu formulieren. Dabei haben wir uns an die in der UML verwendete Konvention gehalten, Klassennamen mit einem großen Anfangsbuchstaben und Rollennamen mit einem kleinen Anfangsbuchstaben beginnen zu lassen.

Wie bereits in Abbildung 5.1 herausgestellt, besteht der MQP-Prozess aus vier fundamentalen Schritten. Diese Schritte nutzen wir zur Strukturierung des MQP-Metamodells in vier Pakete. Abbildung 5.6 ordnet den folgenden Abschnitten 5.3.1 bis 5.3.4 die jeweiligen Schritte des MQP-Prozesses und die korrespondierenden Pakete des MQP-Metamodells zu.

Neben diesen vier Paketen gibt es das Paket *Core*, das zum einen den Aufbau eines MQPs beschreibt und zum anderen Angaben von Metadaten zu einem MQP unterstützt (siehe Abbildung 5.7).

Die folgenden Abschnitte beschreiben jeweils einen fundamentalen MQP-Schritt und sind wie folgt aufgebaut:

- Zielsetzung: Motivation für die Durchführung des Schritts
- Eingangsdokumente: Dokumente oder Teile des vorliegenden MQPs, die für

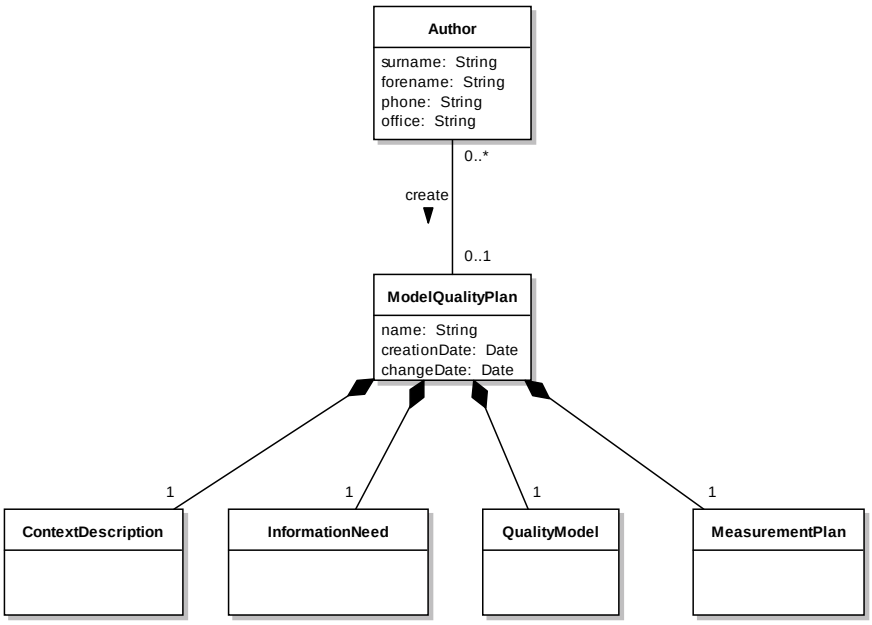


Abbildung 5.7: MQP-Metamodell: Paket Core

die Bearbeitung des Schritts benötigt werden oder hilfreich sind.

- Ergebnis: Teile des vorliegenden MQPs, die hinzugefügt oder modifiziert werden.

- Beteiligte Rollen: An der Ausführung beteiligte Rollen; dabei unterscheiden wir zwischen Qualitätsmanager, Projektleiter und Projektteam.

- Prozessbeschreibung: Notwendige und mögliche Aktivitäten zur erfolgreichen Bearbeitung des Schritts sowie Rückkopplungen zu vorherigen Schritten.

- Richtlinien zur Ausführung: Dos und don'ts, Risiken, Best Practises sowie Anmerkungen zur Ausführung des Schritts.

- Aufbau des Metamodells: Zuerst ein Überblick, anschließend schrittweise Einführung des zum Schritt korrespondierenden Teils des Metamodells, Erläuterungen und Anmerkungen zu Modellierungsentscheidungen sowie Definition der zentralen Konzepte.

5.3.1 Feststellung des Kontextes

Zielsetzung: Die Kontextbeschreibung dokumentiert die Einbettung eines Softwaremodells in seinen übergeordneten Softwareentwicklungsprozess und stellt auf diese Weise heraus, *in welchem Kontext gemessen wird*. Mit der Feststellung des Kontextes verfolgen wir zwei Ziele:

1. Ableiten von Informationsbedürfnissen: Die Feststellung des Kontextes erleichtert die Berücksichtigung von Besonderheiten des betrachteten Softwaremodells in seinem Entwicklungsumfeld. Die Kontextinformationen lassen sich dazu nutzen, um Informationsbedürfnisse systematisch abzuleiten. Durch die vorgelagerte Kontextdokumentation verringert sich der Anteil notwendiger kreativer Arbeit während der Identifizierung der Informationsbedürfnisse. Die Kontextdokumentation stellt neben dem Projektteam eine zweite Informationsquelle für die Identifizierung der Informationsbedürfnisse dar.

2. Interpretation von Prüfergebnissen: Im Anschluss an die Durchführung der Qualitätsprüfung werden die Prüfergebnisse durch Projektverantwortliche interpretiert. Dabei erschließt sich sowohl die Bedeutung des Softwaremodells als auch die Bedeutung der zugehörigen Prüfergebnisse einem Betrachter nicht unabhängig von dem gegebenen Softwareentwicklungsprojekt. Deshalb sollten die Projektverantwortlichen den Kontext des zu bewertenden Softwaremodells für die Deutung der Prüfergebnisse kennen. Da der Kontext des Softwaremodells ohnehin durch Kontextinformationen prägnant wiedergegeben werden soll, eignet sich die für die erste Zielsetzung erstellte Kontextbeschreibung auch für die Interpretation von Prüfergebnissen. Die Kontextbeschreibung ist insbesondere für Entscheidungsträger im Management oder für Kundenvertreter von Interesse, die mit dem Projektvorgehen tendenziell weniger vertraut sind.

Die Durchführungsintensität des Schritts Feststellung des Kontextes muss begrenzt werden, damit der Aufwand für die Erledigung dieses Schrittes den Nutzen rechtfertigt. Bei der Feststellung des Kontextes ist es nicht erheblich, alle theoretisch möglichen Details des Entwicklungsprozesses zu beschreiben. Die Kontextdokumentation soll keine Beschreibung ersetzen, die den Softwareentwicklungsprozess im Projekt vollständig in allen Details abbildet, wie es durch das *Generic Integrated Software Engineering Metamodel* in [183] ermöglicht wird. Die Kontextinformationen müssen nicht der Zielsetzung genügen, einen Mitarbeiter im Projektvorgehen

zu schulen. Vielmehr soll ein Qualitätsmanager unter Einsatz minimaler Ressourcen maximalen Nutzen für die Aktivitäten Informationsbedürfnisse ableiten (siehe 1.) und Prüfergebnisse interpretieren (siehe 2.) erzielen.

Eingangsdokumente: Dokumentation des Projektvorgehens; Projektplan

Ergebnis: MQP-Kontextbeschreibung

Beteiligte Rollen: Qualitätsmanager; Projektleiter

Prozessbeschreibung: Für die Bearbeitung des Schritts lassen sich Fragestellungen verwenden, die sich auf die modellierten Konzepte beziehen:

- Mit welchem Werkzeug wird das Softwaremodell erstellt?
- Auf welche Domäne bezieht sich das Softwaremodell?
- Welche Rollen nutzen das Softwaremodell?
  - Auf welche Weise nutzen diese Rollen das Softwaremodell?
  - Wie lassen sich die Rollen beschreiben?
- Während welcher Entwicklungsphasen wird das Softwaremodell erstellt?
- ...

Richtlinien zur Ausführung: Die Feststellung des Kontextes sollte direkt zu Beginn der Qualitätsplanung vor allen anderen Schritten erfolgen.

Aufbau des Kontext-Metamodells: Im Folgenden erklären wir das Kontext-Metamodell schrittweise und definieren die zentralen Konzepte. Die Grundlagen für die Kontextbeschreibung eines Softwaremodells haben wir bereits in Kapitel 3 gelegt und wiederholen nur die wichtigsten Aussagen. Abbildung 5.8 zeigt einen Überblick über das Kontext-Metamodell.

Die folgenden Ausführungen beziehen sich auf den in Abbildung 5.9 gezeigten Ausschnitt des Kontext-Metamodells: Eine Kontextbeschreibung bezieht sich immer auf genau ein *Softwaremodell* (Definition 5.1). Softwaremodelle stellen spezielle *Softwareentwicklungsartefakte* (Definition 5.2) dar. Softwareentwicklungsartefakte

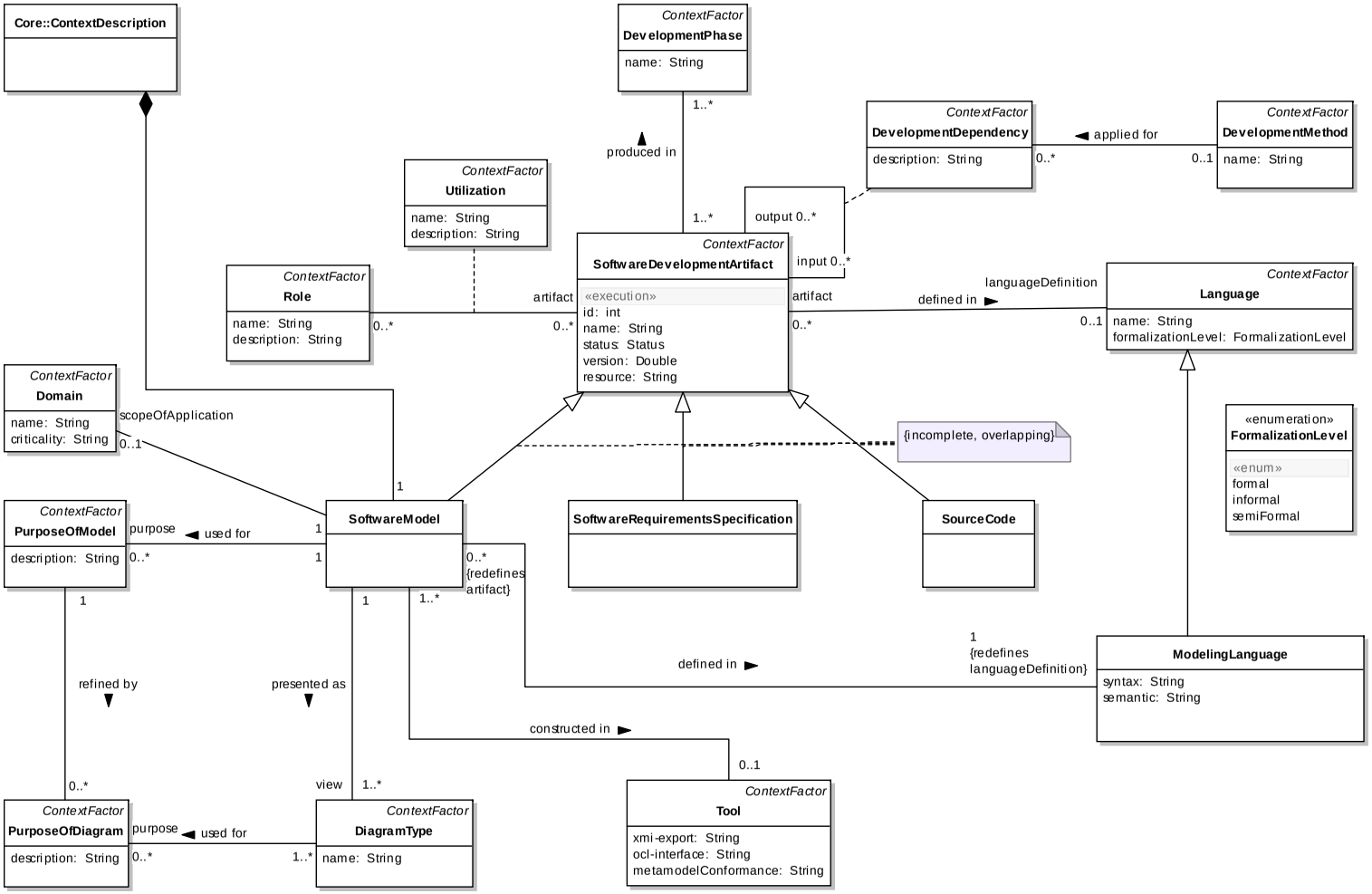


Abbildung 5.8: Kontext-Metamodell: Überblick

dokumentieren (Zwischen-) Ergebnisse der einzelnen *Entwicklungsphasen* (Definition 5.3). Die Entwicklungsphase gibt Aufschluss darüber, wie implementierungsnah das Softwaremodell ist. Dabei können Softwareentwicklungsartefakte z.B. im Falle spiralförmiger Vorgehensmodelle über mehrere Entwicklungsphasen hinweg entstehen. *Softwareanforderungsspezifikationen* (Definition 5.4) oder *Source Code* (Definition 5.5) sind wie Softwaremodelle auch spezielle Softwareentwicklungsartefakte, werden aber im Rahmen des vorliegenden Kontext-Metamodells weniger ausführlich modelliert. Die von uns modellierte Menge der speziellen Entwicklungsartefakte erhebt keinen Anspruch auf Vollständigkeit. Personen, die spezielle *Rollen* (Definition 5.6) wahrnehmen, nutzen Entwicklungsartefakte, um Ergebnisse zu sichern, zu prüfen oder weiterzuentwickeln. Diese Rollen stehen somit in einem direkten Zusammenhang mit Entwicklungsartefakten und Vertreter dieser Rollen sollten im nächsten Schritt hinsichtlich ihrer Informationsbedürfnisse befragt werden. Softwaremodelle werden mit Hilfe eines Werkzeugs entwickelt. Modellierungswerkzeuge unterscheiden sich in ihrem Funktionsumfang und ihrer Konformität zur eingesetzten Modellierungssprache. Der Funktionsumfang beeinflusst die notwendigen Überprüfungen der Sprachrichtigkeit und die formalen Definitionen objektiver Messungen im Schritt *Dokumentation der Messungen*. Softwaremodelle beziehen sich auf eine Domäne (z.B. Bankensektor, Gesundheitswesen, Chemiebranche, Automobilbranche). Die Domäne beeinflusst *vermutlich* die Fachbegriffe im Qualitätsmodell und die genauen Grenz- und Zielwerte für Indikatoren. Die Domäne ist streng genommen bereits in der Einleitung einer Softwareanforderungsspezifikation charakterisiert [84]. Trotzdem weisen wir diese Eigenschaft den Softwaremodellen zu, weil sich erstens die Domäne auf alle im Entwicklungsprozess erstellten Softwareentwicklungsartefakte bezieht und in unserem Fall das Softwaremodell das zentrale Artefakt ist und zweitens eine Softwareanforderungsspezifikation u.U. gar nicht in der Kontextbeschreibung erwähnt wird, die Domäne aber doch genannt werden soll.

Definition 5.1 (*Software Model*)

A software model is a description or specification of a software system. It replaces the software system for some purpose and only includes those aspects that are relevant to its purpose, at the appropriate level of detail. A software model is defined in a modeling language and is presented by diagrams and/or text (for more details cf. Definition 3.3).

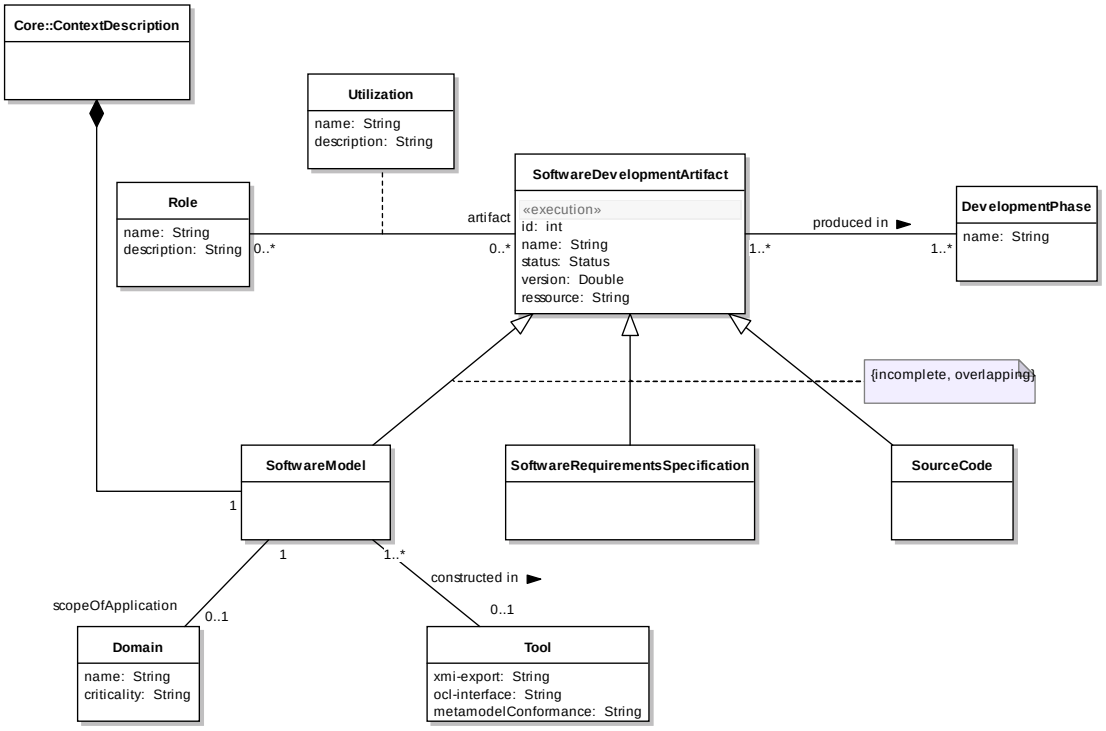


Abbildung 5.9: Kontext-Metamodell: Entwicklungsartefakte

Definition 5.2 (*Software Development Artifact*)

A software development artifact is a tangible byproduct or a product produced during a software development phase (based on [99]).

Definition 5.3 (*Software Development Phase*)

Software development is the set of phases that results in software products. In each software development phase a variety of activities take place (based on [99]).

Definition 5.4 (*Software Requirements Specification*)

A software requirements specification (SRS) is a complete description of the requirements that a software system to be developed must or should fulfill (based on [84]).

Definition 5.5 (*Source Code*)

Source Code is any collection of statements or declarations written in some human-readable computer programming language (based on [139]).

Definition 5.6 (*Role*)

A role depicts a set of connected behaviors, rights and obligations in a software development process (based on [132]).

Die folgenden Ausführungen beziehen sich auf den in Abbildung 5.10 gezeigten Ausschnitt des Kontext-Metamodells: Für die Weiterentwicklung von Softwareentwicklungsartefakten lassen sich Entwicklungsmethoden (Definition 5.7) einsetzen. Wenn dabei Softwareentwicklungsartefakte für die Erstellung anderer Softwareentwicklungsartefakte weiterverwendet werden, entstehen *Entwicklungsabhängigkeiten* (Definition 5.8) zwischen diesen Softwareentwicklungsartefakten. Diese Entwicklungsabhängigkeiten repräsentieren fachliche Zusammenhänge und deuten auf potentielle, vertikale Inkonsistenzen hin. Inkonsistenzen können z.B. auf eine fehlerhafte Anwendung einer Entwicklungsmethode zurückgeführt werden.

Definition 5.7 (*Development Method*)

A development method is a described procedure for developing a software development artifact (based on [124]).

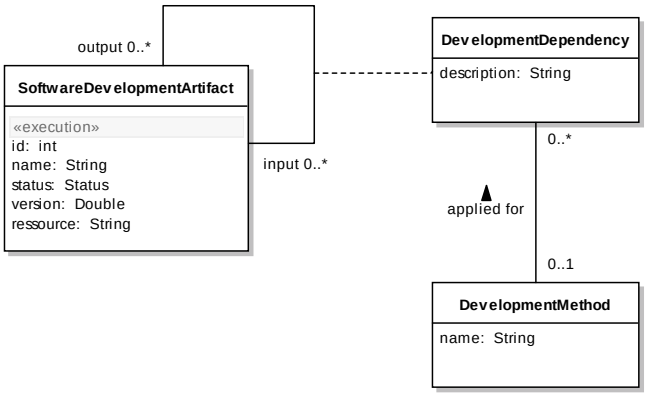


Abbildung 5.10: Kontext-Metamodell: Entwicklungsabhängigkeiten

Definition 5.8 (*Development Dependency*)

A development dependency occurs between two sets of software development artifacts, if the one set is constructed based on the other set.

Die folgenden Ausführungen beziehen sich auf den in Abbildung 5.11 gezeigten Ausschnitt des Kontext-Metamodells: Softwarentwicklungsartefakte werden mit Hilfe einer Sprache definiert. Softwaremodellen im Speziellen liegt eine *Modellierungssprache* (Definition 5.9) zu Grunde. Informationen über die Sprachdefinition sind für syntaktische und semantische Analysen von Bedeutung. Ein Softwaremodell besteht aus einer Menge von Elementen (z.B. Pakete, Klassen und Assoziationen). Die korrespondierenden *Diagramme* (Definition 5.10) sind grafische Repräsentationen von Teilen dieses Softwaremodells. Diagramme visualisieren die konkrete Syntax. Mehrere Diagramme können sich auf ein und dasselbe Softwaremodell beziehen und sich dabei inhaltlich überlappen. Die Überlappungen deuten auf potentielle horizontale Inkonsistenzen hin.

Definition 5.9 (*Modeling Language*)

A modeling language is an artificial language that consists of syntax and semantics. Syntax is the language's notation and defines the structure of a model. The syntactic elements can be words, boxes, arrows, rectangles, closed curves, and so on. Semantic is the meaning of the syntactic elements. A semantic is defined by a semantic mapping from the syntactic elements to a semantic domain (based on [74]).

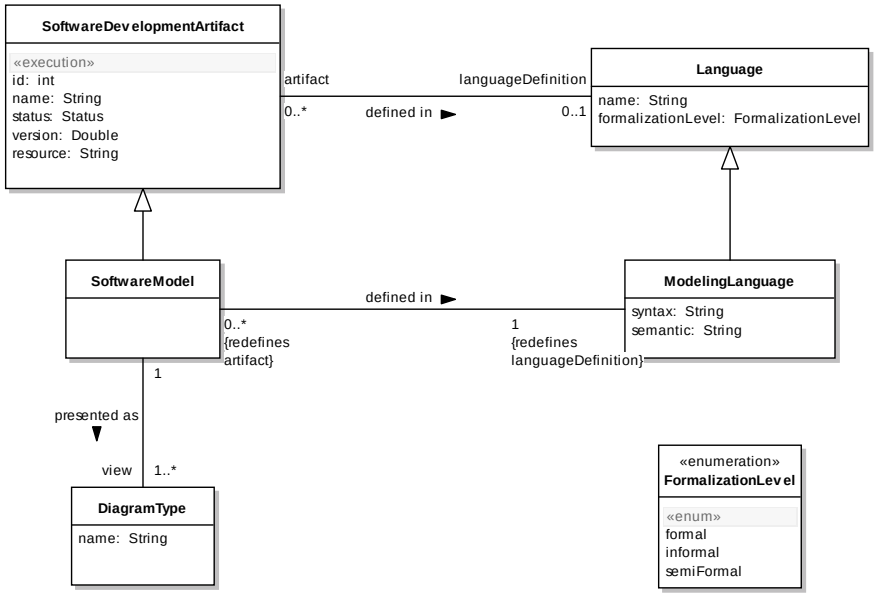


Abbildung 5.11: Kontext-Metamodell: Sprachdefinitionen

Definition 5.10 (*Diagram*)

A model consists of elements such as packages, classes, and associations. The corresponding diagrams are graphical representations of parts of the model. Diagrams contain graphical elements (nodes connected by paths) that represent elements in the software model (based on [131]).

Die folgenden Ausführungen beziehen sich auf den in Abbildung 5.12 gezeigten Ausschnitt des Kontext-Metamodells: Softwaremodelle werden zielgerichtet zur Erfüllung bestimmter Aufgaben eingesetzt. Sie dienen einem oder mehreren Verwendungszwecken. Besteht ein Softwaremodell aus mehreren Diagrammen, dann können die Verwendungszwecke für einzelne Diagrammtypen differenziert werden. Der Verwendungszweck kann sogar für den gleichen Diagrammtypen variieren. UML-Klassendiagramme werden als Modellierung eines Problembereichs in der Anforderungsanalyse, als Analysedokument, Design-Spezifikation, Datenmodell u.v.m. genutzt.

Definition 5.11 (*Purpose of Model*)

The use of the software model (viz. the reason for developing the software model) (Based on [161]).

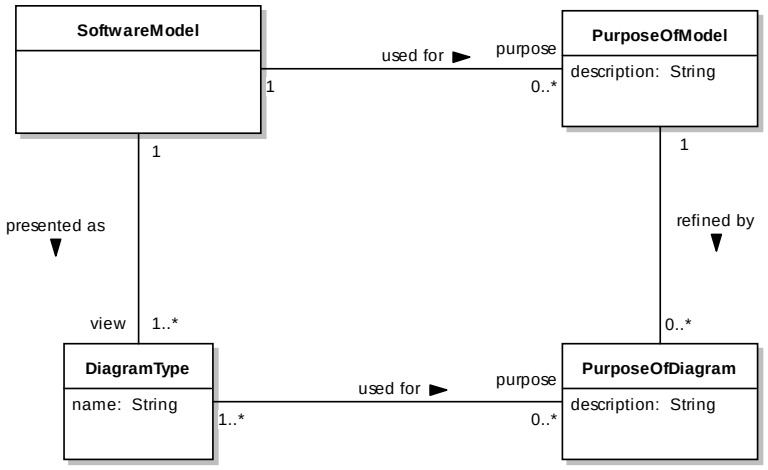


Abbildung 5.12: Kontext-Metamodell: Verwendungszwecke

Definition 5.12 (*Purpose of Diagram*)

The use of a diagram type with that the software model is presented.

Die zuvor vorgestellten Konzepte der Kontextbeschreibung bezeichnen wir allgemein als *Kontextfaktor* (siehe Abbildung 5.13). Diese Kontextfaktoren spielen eine wichtige Funktion für den Übergang zum nächsten Schritt.

5.3.2 Identifizierung der Informationsbedürfnisse

Zielsetzung: Die Informationsbedürfnisse dokumentieren die Qualitätsziele der am Entwicklungsprozess beteiligten Rollen. Sie geben an, *was wichtig zu messen ist*. Die Bewertung der Qualität eines Softwaremodells hängt nicht von einem fixen Qualitätsmaßstab ab, sondern ist immer relativ zu gegebenen Qualitätszielen. Diese Qualitätsziele hängen wiederum sehr stark von den Modellnutzern ab, die z.B. in der Kontextbeschreibung durch Rollen repräsentiert werden. Durch die Identifizierung der Informationsbedürfnisse bezwecken wir, alle relevanten Qualitätsziele an das zu überprüfende Softwaremodell aus Sicht der am Entwicklungsprozess beteiligten Rollen in Erfahrung zu bringen und dadurch die Qualitätsplanung einerseits zu vervollständigen und andererseits auf das Wesentliche zu beschränken.

Eingangsdokumente: Kontextbeschreibung (vgl. Abschnitt 5.3.1); falls verfügbar: existierende MQPs bzw. MQP-Regeln für gleiche oder ähnliche Kontexte (vgl.

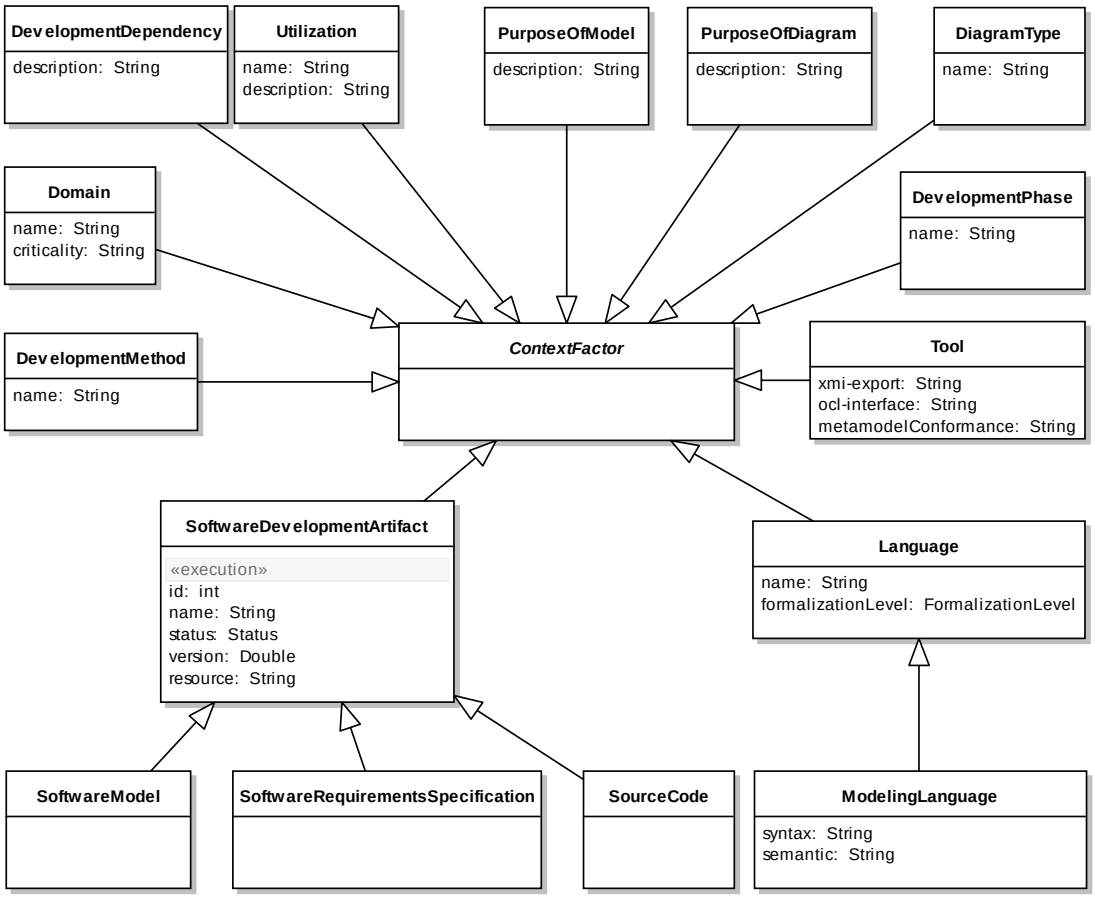


Abbildung 5.13: Kontext-Metamodell: Kontextfaktoren

Abschnitt 5.4)

Ergebnis: Informationsbedürfnisse

Beteiligte Rollen: Qualitätsmanager; Projektleiter; Projektteam

Prozessbeschreibung: Für die Identifizierung der Informationsbedürfnisse eignen sich drei Informationsquellen:

1. Der Qualitätsmanager interviewt Stellvertreter des Projektteams und hält die Gesprächsergebnisse in Form von Zielen und damit verknüpften Fragen fest. Strukturiertes Brainstorming und ähnliche Methoden zur Ideenfindung sind insbesondere für das Ermitteln von Fragen hilfreich. Während der Interviews können die Ziele informell dokumentiert und erst im Anschluss durch den Qualitätsmanager mit Hilfe des GQM-Templates für Ziele formalisiert werden. Zudem muss der Qualitätsmanager die Gesprächsergebnisse zusammenführen und konsolidieren.

2. Ausgehend von den Kontextinformationen kann der Qualitätsmanager selbstständig systematisch Ziele ableiten.

3. Falls existierende MQPs bzw. MQP-Regeln für gleiche oder ähnliche Kontexte zur Verfügung stehen, kann der Qualitätsmanager dieses Erfahrungswissen wiederverwenden. Die genaue Funktionsweise von MQP-Regeln erläutern wir im Abschnitt 5.4.

Richtlinien zur Ausführung: Die Identifizierung der Informationsbedürfnisse sollte direkt im Anschluss an den Schritt Feststellung des Kontextes erfolgen. Es kann jederzeit zu diesem Schritt gesprungen werden. Allerdings sollte dabei beachtet werden, dass dieser Schritt sehr kostenintensiv ist.

Um den zusätzlichen Aufwand für das Personal des Projektteams zu begrenzen, sollte der Qualitätsmanager nicht alle am Entwicklungsprozess beteiligten Personen in diesen Schritt integrieren, sondern nur einen oder wenige Vertreter für jede involvierte Rolle interviewen. Als Untergrenze empfehlen wir mindestens genau einen Vertreter der im Kontext dokumentierten Rollen und einen Vertreter der Projektleitung zu befragen. Der Qualitätsmanager ist auf eine konstruktive und wohlwollende

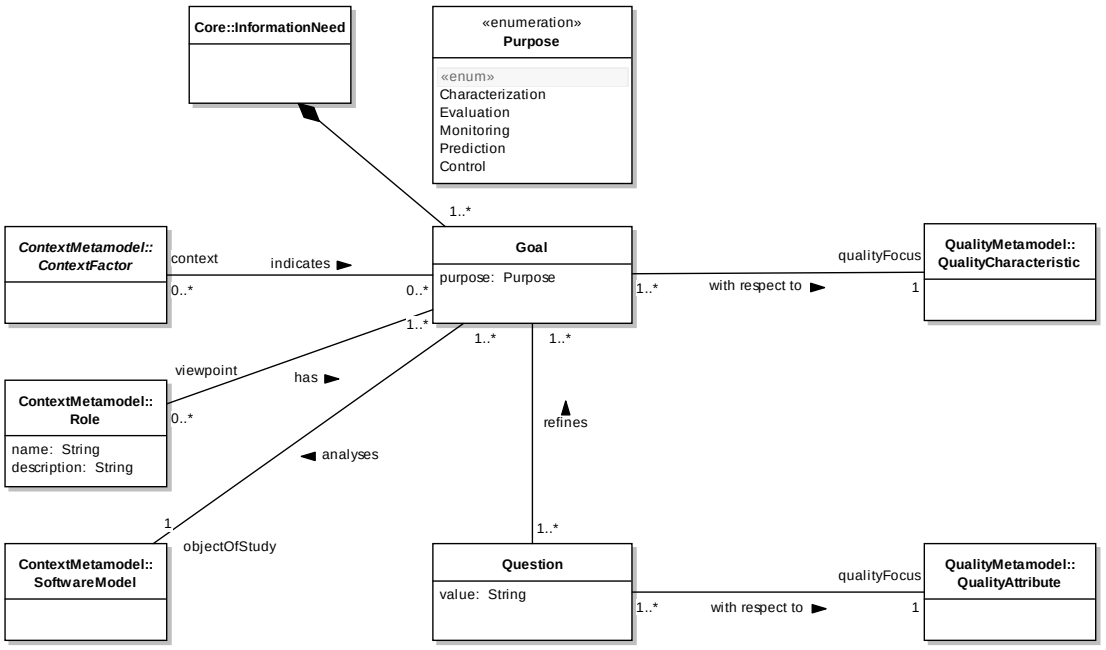


Abbildung 5.14: Informationsbedürfnis-Metamodell

Zusammenarbeit mit dem Projektteam angewiesen und benötigt dafür die Unterstützung der Projektleitung. Aus diesem Grund sollte der Qualitätsmanager die Projektleitung von der Zweckmäßigkeit der analytischen Qualitätssicherung mittels des MQP-Ansatzes überzeugen, indem auch für sie relevante Qualitätsziele überprüft werden.

Aufbau des Informationsbedürfnis-Metamodells: Im Folgenden erklären wir das Informationsbedürfnis-Metamodell und definieren die zentralen Konzepte. Abbildung 5.14 zeigt das vollständige Informationsbedürfnis-Metamodell.

Die Informationsbedürfnisse (Definition 5.13) bestehen aus einer Menge von Zielen (Definition 5.14), die wiederum durch Fragen (Definition 5.15) verfeinert sind. Für jedes Ziel soll dabei folgende Dimensionen geklärt werden:

- Untersuchungsobjekt: Was wird analysiert?
- Zweck: Warum wird das Objekt analysiert?
- Qualitätsschwerpunkt: Welche Qualitätscharakteristik wird analysiert?

- Sichtweise: Wer nutzt die Prüfergebnisse?
- Kontext: In welchem Kontext wird analysiert?

Fragen werden einfach als Fragestellung formuliert und den Zielen zugeordnet. Zusätzlich erhalten sie einen eigenen Qualitätsschwerpunkt in Form eines Qualitätsattributs.

Definition 5.13 (*Information Need*)

Insight necessary to manage objectives, goals, risks, and problems (based on [86]).

Definition 5.14 (*Goal*)

A goal is defined for a software model with respect to its quality, from various points of view, relative to a particular context (based on [7]).

Definition 5.15 (*Question*)

Questions try to characterize the object of measurement with respect to a selected quality issue and to determine its quality from the selected viewpoint (based on [7]).

5.3.3 Erstellung des Qualitätsmodells

Zielsetzung: Das Qualitätsmodell enthält die relevanten Qualitätscharakteristiken und Qualitätsattribute des zu prüfenden Softwaremodells und beschreibt auf diese Weise *was gemessen wird*. Mit der Erstellung eines Qualitätsmodells verfolgen wir drei Ziele:

1. Dokumentation von Qualitätszielen: Das Qualitätsmodell präzisiert die Informationsbedürfnisse, indem die mit den Zielen und Fragen assoziierten Qualitätscharakteristiken und Qualitätsattribute definiert sind. Zudem lassen sich deren Zusammenhänge innerhalb des Qualitätsmodells durch weitere Qualitätsteilcharakteristiken verfeinern. Durch die Kombination von Informationsbedürfnissen, die analog zum GQM-Ansatz beschrieben sind, und einem Qualitätsmodell erhöht sich die Detaillierung der Qualitätsziele.

2. Definition eines einheitlichen Qualitätsverständnisses: Durch die Definitionen der Qualitätscharakteristiken und Qualitätsattribute legt ein Qualitätsmodell eine verbindliche Terminologie für das Projektteam fest. Diese fixierte Terminologie ist wichtig, um die Kommunikation der Projektbeteiligten zu fördern. Qualität bestimmt sich durch die Wahrnehmung von Menschen und ist somit inhärent subjektiv. Individuelle Unterschiede im Qualitätsverständnis werfen innerhalb eines Projektteams Kontroversen auf, die durch einen gemeinsamen Sprachgebrauch leichter zu klären sind.

3. Operationalisierung von Qualität: Qualitätsmodelle stellen einen weit verbreiteten Ansatz dar, um Qualität nach dem Divide & Conquer Prinzip in mehrere Bestandteile zu gliedern und auf diese Weise zu präzisieren. Ein allgemeiner Qualitätsbegriff wird durch das Ableiten von Unterbegriffen sukzessive verfeinert und auf diese Weise operationalisiert. Sind Qualitätsmodelle bis zur Ebene der Qualitätsattribute verfeinert, dann lassen sich diese durch Metriken quantifizieren und durch Indikatoren qualifizieren.

Eingangsdokumente: Informationsbedürfnisse (vgl. Abschnitt 5.3.2); falls verfügbar: existierende MQPs bzw. MQP-Regeln für gleiche oder ähnliche Kontexte (vgl. Abschnitt 5.4); bestehende Qualitätsmodelle (vgl. Abschnitt 4.2)

Ergebnis: Qualitätsmodell

Beteiligte Rollen: Qualitätsmanager; Projektteam

Prozessbeschreibung: Basierend auf den Qualitätsschwerpunkten der Ziele und Fragen, die zusammen die Informationsbedürfnisse beschreiben, leitet der Qualitätsmanager einen ersten Rahmen für das projektspezifische Qualitätsmodell ab. Die Qualitätscharakteristiken und Qualitätsattribute sind zu diesem Zeitpunkt bezeichnet und grob in Beziehung gesetzt, so dass ersichtlich ist, wie sich diese untereinander beeinflussen. Ausgehend von diesem Rahmen wird das Qualitätsmodell vervollständigt, indem Qualitätsteilcharakteristiken eingeführt werden, die den Zusammenhang zwischen den gegeben Qualitätscharakteristiken und Qualitätsattributen weiter präzisieren. Natürlich kann der Qualitätsmanager auch bestehende Qualitätscharakteristiken durch neue, übergeordnete Qualitätscharakteristiken gruppie-

ren. Für jede Qualitätscharakteristik und für jedes Qualitätsattribut müssen eine Definition und evtl. Synonyme angegeben werden. Der Qualitätsmanager kann das projektspezifische Qualitätsmodell nun mit bestehenden Qualitätsmodellen vergleichen und dadurch potentielle Inkonsistenzen, aufgetretene Konflikte oder fehlende Elemente (z.B. Fragen ohne Qualitätsattribute) aufdecken. Diese potentiellen Fehler im Qualitätsmodell sollte der Qualitätsmanager auf die Informationsbedürfnisse übertragen und mit den zuvor involvierten Vertretern des Projektteams besprechen. Zudem muss das Projektteam prüfen, ob die Qualitätscharakteristiken und Qualitätsattribute in seinem Sinne definiert sind. Jede Qualitätscharakteristik und jedes Qualitätsattribut wird im Qualitätsmodell durch einen eindeutigen Begriff repräsentiert und darf nur einmal auftreten. Der Qualitätsmanager muss insofern gewährleisten, dass jeder der in diesem Zusammenhang verwendeten Begriffe der Terminologie des Projektteams entspricht. Dafür sind möglicherweise zuerst noch Einigungen im Projektteam zu erzielen. Das beschriebene Vorgehen zeigt die enge Verzahnung der Prozesschritte *Identifizierung der Informationsbedürfnisse* und *Erstellung des Qualitätsmodells*. Der MQP-Prozess fordert lediglich, dass mit dem Schritt *Identifizierung der Informationsbedürfnisse* begonnen werden muss. Darüber hinaus stellt der Ansatz keine Restriktionen für das Springen zwischen diesen beiden Schritten auf. Das Projektteam ist am Schritt *Erstellung des Qualitätsmodells* lediglich in kontrollierender Tätigkeit beteiligt. Die Ausführung übernimmt der Qualitätsmanager.

Richtlinien zur Ausführung: Zuerst erstellt der Qualitätsmanager das Qualitätsmodell selbstständig, ohne das Projektteam einzubeziehen. Zur Überprüfung des von ihm erstellten Qualitätsmodells ist er dagegen auf das Projektteam angewiesen. Jede Einbindung des Projektteams stellt indessen einen erheblichen Kostenfaktor dar und sollte durch den Qualitätsmanager maßvoll in Anspruch genommen werden. Wenn ein unternehmensspezifisches oder projektspezifisches Qualitätsmodell bereits existiert, sollte der Qualitätsmanager unbedingt die dort verwendeten Begriffe und Definitionen weitestgehend übernehmen, um den Aufwand für die Prüfung des Qualitätsmodells durch das Projektteam und etwaige Anpassungen zu reduzieren.

Aufbau des Qualitäts-Metamodells: Im Folgenden erklären wir das Qualitäts-Metamodell und definieren die zentralen Konzepte. Die Grundlagen für Qualitäts-

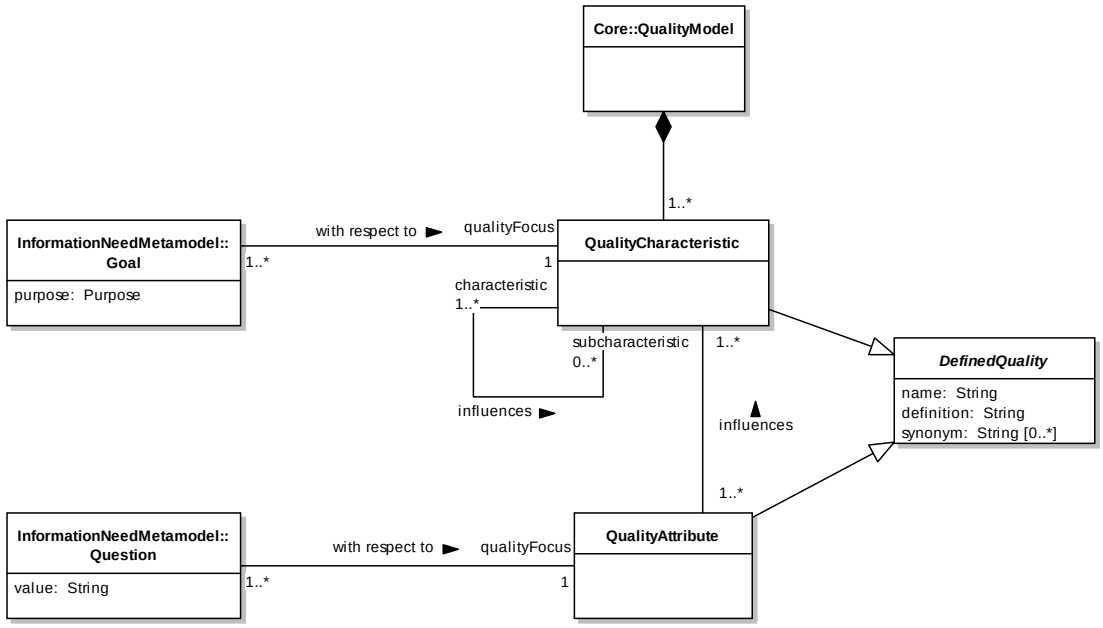


Abbildung 5.15: Qualitäts-Metamodell

modelle haben wir bereits in Abschnitt 4.1.2 und 4.2 gelegt und wiederholen nur die wichtigsten Aussagen. Abbildung 5.15 zeigt das vollständige Qualitäts-Metamodell.

Ein Qualitätsmodell (Definition 5.16) ist eine Menge von Qualitätscharakteristiken und Beziehungen zwischen diesen. Qualitätsmodelle verfeinern den allgemeinen Qualitätsbegriff für Modellqualität (Definition 5.17). Diese Definition von Modellqualität basiert auf verfügbaren Definitionen für Produktqualität [89, 82, 31] und ist in dieser Form allgemeingültig. Infolgedessen verzichten wir darauf, Modellqualität als eigenständige Klasse mit dieser Definition als Belegung eines nicht veränderbaren Attributs (Property mit den Eigenschaften *default = Definition 5.17* und *isReadOnly = true*) im Qualitäts-Metamodell zu modellieren, weil diese Information nicht projektspezifisch ist und somit für einen MQP keinen Mehrwert darstellt. Eine Verfeinerung der allgemeinen Definition findet erst durch die konkreten Ausprägungen der Qualitätscharakteristiken und Qualitätsattribute statt. Eine Qualitätscharakteristik (Definition 5.18) ist eine Kategorie von Qualitätsattributen, die sich auf Produktqualität (hier: Modellqualität) bezieht. Qualitätscharakteristiken können in mehrere Ebenen von Qualitätsteilcharakteristiken und schließlich in Qualitätsattribute verfeinert werden. Ein Qualitätsattribut (Definition 5.19) ist eine direkt mess-

bare, inhärente Eigenschaft eines Objekts.

Ein Qualitätsmodell bezieht sich immer auf genau ein Softwaremodell und definiert für dieses ein Qualitätsverständnis. Diesen Zusammenhang modellieren wir im Qualitäts-Metamodell nicht explizit, da sich dieser Sachverhalt bereits aus den bestehenden Assoziationen ergibt. Das zu prüfende Softwaremodell wird bereits im Rahmen der Kontextbeschreibung festgelegt (siehe Abbildung 5.8).

Den Zusammenhang zu den Informationsbedürfnissen haben wir bereits im vorherigen Abschnitt 5.3.2 geklärt.

Definition 5.16 (*Quality Model*)

The set of quality characteristics, quality attributes, and the relationships between them which provide the basis for specifying quality requirements and evaluating quality (based on [85]).

Definition 5.17 (*Model Quality*)

The totality of quality characteristics of a software model that bear on its ability to satisfy stated and implied needs (based on [85]).

Definition 5.18 (*Quality Characteristic*)

Category of quality attributes that bears on model quality. Quality characteristics may be refined into multiple levels of sub characteristics and finally into quality attributes (based on [85]).

Definition 5.19 (*Quality Attribute*)

A quality attribute is a directly measureable property of a software model. A property is directly measureable if it does not depend upon a measure of any other attribute (based on [85]).

5.3.4 Dokumentation der Messungen

Zielsetzung: Die Dokumentation der Messungen vervollständigt den MQP und beschreibt einen Messplan, der festlegt *wie gemessen wird*. Mit einem solchen Messplan beabsichtigen wir die Durchführung der Qualitätsprüfung vorzubereiten. Er schreibt vor, wie Qualitätsattribute quantifiziert und qualifiziert werden sollen.

Aus konzeptioneller Sicht soll der Messplan alle für die Durchführung der Qualitätsprüfung relevanten Informationen enthalten. Damit MQPs möglichst unabhängig von einer konkreten IT-Landschaft formuliert werden und leichter auf andere Entwicklungsprojekte zu übertragen sind, enthält der Messplan keine Informationen über technische Details zu Messwerkzeugen und zur Speicherung der Messergebnisse. Für die Durchführung der Qualitätsprüfungen müssen die objektiven Messungen noch an Messwerkzeuge und die subjektiven Messungen an Reviewer übergeben werden. Auf Messwerkzeuge gehen wir im Rahmen zweier Fallstudien in Kapitel 6 ein. Die Koordinierung subjektiver Messungen setzt organisatorische Maßnahmen voraus. Jedoch diskutieren wir diese Maßnahmen im Rahmen der vorliegenden Arbeit nicht und verweisen stattdessen auf [66, 177, 118].

Eingangsdokumente: Qualitätsmodell (vgl. Abschnitt 5.3.3); falls verfügbar: existierende MQPs bzw. MQP-Regeln für gleiche oder ähnliche Kontexte (vgl. Abschnitt 5.4); bestehende Qualitätsmodelle mit assoziierten Messungen (vgl. Abschnitt 4.2); bestehende Sammlungen von Messungen (vgl. Abschnitt 4.3)

Ergebnis: Messplan mit Messungen

Beteiligte Rollen: Qualitätsmanager, Projektleitung

Prozessbeschreibung: In diesem Schritt soll der Qualitätsmanager jedem Qualitätsattribut des Qualitätsmodells eine Menge von Messungen zuordnen. Messungen kann er aus bestehenden Sammlungen von Messungen auswählen. Falls für ein Qualitätsattribut keine geeigneten Messungen vorliegen oder die bestehenden Messungen nicht ausreichend erscheinen, muss der Qualitätsmanager neue Messungen definieren. Für alle Messungen ist hierbei wichtig, dass sie das betreffende Qualitätsattribut *nützlich* quantifizieren oder qualifizieren und helfen, die mit dem Qualitätsattribut verknüpfte Frage zu beantworten (vgl. Anforderung 7 in Abschnitt 2.4). Allerdings kann die Eignung einer Messung nicht bewiesen werden und stellt sich häufig erst nach mehrfacher Anwendung eines MQPs heraus.

Für die Dokumentation neuer und auch bestehender Messungen empfehlen wir folgendes Vorgehen: Zuerst sollte der Qualitätsmanager Indikatoren und Metriken festlegen und informell beschreiben. Können Indikatoren aufgrund fehlender exakter Entscheidungskriterien nicht wie in Abschnitt 4.3.3 formuliert werden, dann sollte

er für Metriken Interpretationshilfen angeben wie beispielsweise: Je näher der Wert zu 0 ist, desto besser. Anschließend werden die Indikatoren bzw. Metriken präzisiert, indem sie in abgeleitete Kennzahlen und Basiskennzahlen zerlegt, Skalen festgelegt und objektive Messungen formal definiert werden.

Für jede Basiskennzahl muss der Qualitätsmanager abwägen, ob sie mit einer objektiven Messung bestimmt werden kann, oder ob eine subjektive Messung unvermeidbar ist. Subjektive Messungen werden durch Reviewer erhoben und stellen somit einen erheblichen Kostenfaktor in der Durchführung der Qualitätsprüfung dar. Zudem lassen sich subjektive Messungen häufig nicht ad-hoc bestimmen und bremsen bei mangelnder Verfügbarkeit von Reviewern die Durchführung der Qualitätsprüfung. In welchem Umfang überhaupt subjektive Messungen eingesetzt werden dürfen, muss der Qualitätsmanager in Absprache mit der Projektleitung klären. Dabei sollte der Umfang subjektiver Messungen dem Qualitätsmanager vorher nicht bekannt sein, damit er das Projektteam nicht unabsichtlich während des Schritts Identifizierung der Informationsbedürfnisse beeinflusst und von bestimmten Fragen abrät. Unserer Meinung nach ist auch die Information, dass gewisse Qualitätsattribute und in der Konsequenz Qualitätscharakteristiken und Ziele *nicht* überprüft werden, für die Interpretation der Messergebnisse sehr wichtig.

Die beschriebenen Messungen müssen in der Qualitätsprüfung in einer festgelegten Reihenfolge bearbeitet werden (z.B. Berechnung von Basiskennzahlen vor abgeleiteten Kennzahlen). Ein entsprechender Abhängigkeitsgraph, der die Reihenfolge der Messungen angibt, kann problemlos durch Abarbeitung der Kennzahlen generiert werden. Im Gegensatz zu objektiven Messungen lassen sich subjektive Messungen weniger spontan erheben, da sie von Reviewern ausgeführt werden. Diese subjektiven Messungen werden dafür thematisch (z.B. anhand der Diagrammtypen) in Checklisten gruppiert und an die entsprechenden Reviewer zur Durchführung verteilt.

Richtlinien zur Ausführung: Die Zerlegung der Indikatoren und Metriken in abgeleitete Kennzahlen und Basiskennzahlen sollte sehr feingranular sein. Dies erhöht zum einen die Performanz der Messungen durch ein Messwerkzeug, weil z.B. die Anzahl der Klassen im Softwaremodell in mehrere Metriken einfließt, aber nur einmalig bestimmt wird. Zum anderen lassen sich komplizierte Messungen verständlicher beschreiben und drittens können Basiskennzahlen für die Definition neuer

Metriken wiederverwendet werden. Diverse Beispiele für die Verwendung von Basiskennzahlen und abgeleiteten Kennzahlen zur Definition von Metriken finden sich in [28, 5, 25].

Wie bereits im obigen Paragraphen Prozessbeschreibung dargelegt, sollte der Qualitätsmanager subjektive Messungen immer sparsam einsetzen.

Entscheidungskriterien für Indikatoren dürfen nicht geraten werden, da dies zu Fehlentscheidungen während der Interpretation der Messergebnisse mit schwerwiegenden Folgen führen kann. Die Entscheidungskriterien sollten statistisch abgesichert sein, aber auch logisch sinnvoll erscheinen. Zudem sollte bei bestehenden Entscheidungskriterien ihre Anwendbarkeit auf das aktuelle Softwaremodell in seinem Entwicklungskontext berücksichtigt werden. Im Zweifel muss der Qualitätsmanager sich gegen die Definition eines Indikators entscheiden. Bei der Einführung neuer Messungen raten wir grundsätzlich davon ab, Indikatoren auf diesen Metriken zu definieren.

Der Qualitätsmanager sollte versuchen, möglichst für alle Qualitätsattribute Messungen zu finden.

Aufbau des Mess-Metamodells: Im Folgenden erklären wir das Mess-Metamodell schrittweise und definieren die zentralen Konzepte. Die Grundlagen für Messungen eines Softwaremodells haben wir bereits in Abschnitt 4.1.3 und 4.3 dargelegt und wiederholen nur die wichtigsten Aussagen. Abbildung 5.16 zeigt einen Überblick über das Mess-Metamodell.

Ein Messplan besteht aus einer Menge von Kennzahlen (Definition 5.20). Eine Kennzahl ist eine Variable, der als Ergebnis einer Messung ein Wert zugewiesen wird. Innerhalb eines Softwaremodells kann eine Kennzahl für mehrere Modellelemente (z.B. alle Klassen des Softwaremodells) ermittelt werden. Diese Messwerte entsprechen dann einer Skala (Definition 5.21), die der Kennzahl zugeordnet ist.

Definition 5.20 (*Measure*)

Variable to which a value is assigned as the result of measurement (based on [86]).

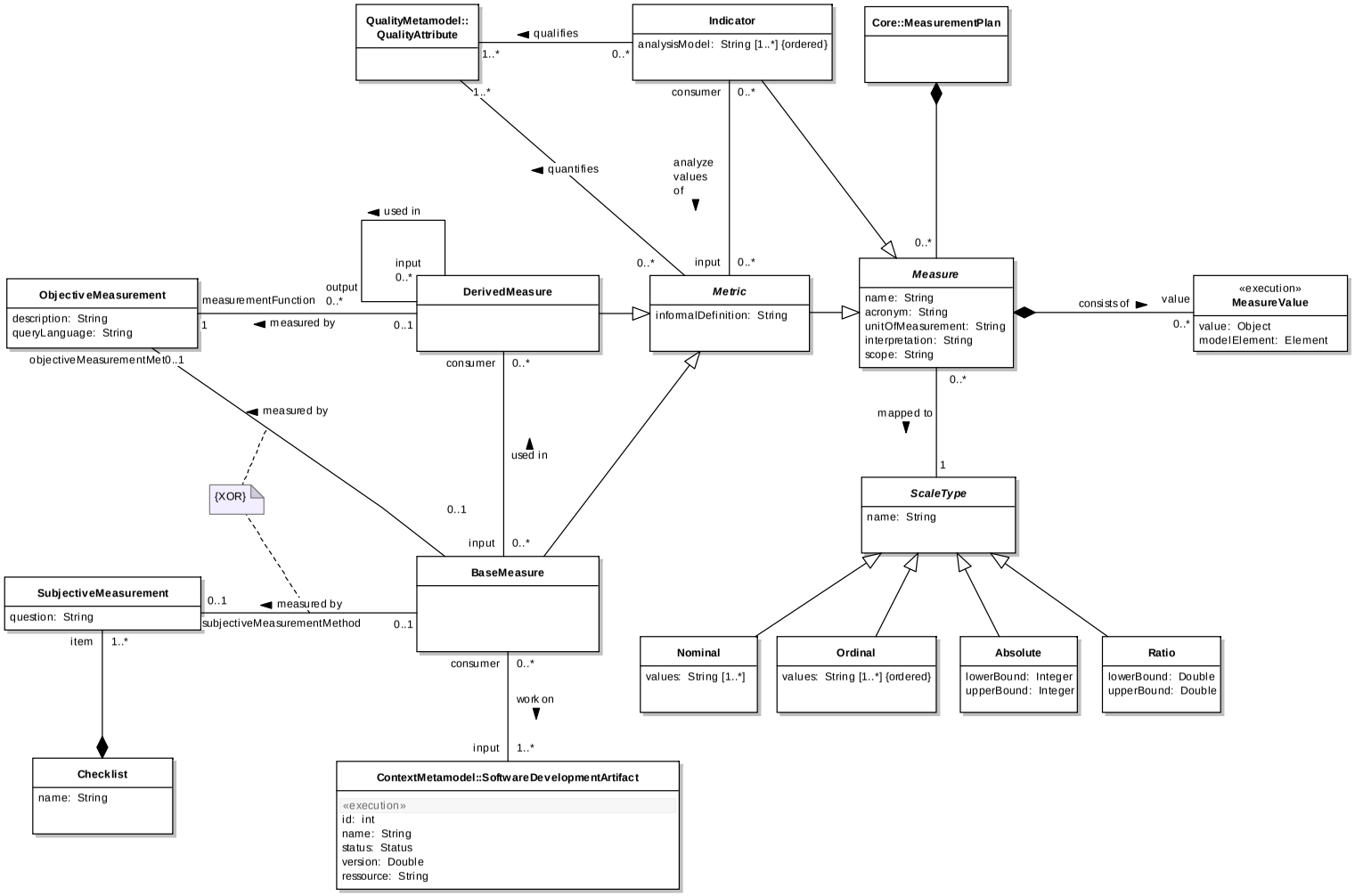


Abbildung 5.16: Mess-Metamodell: Überblick

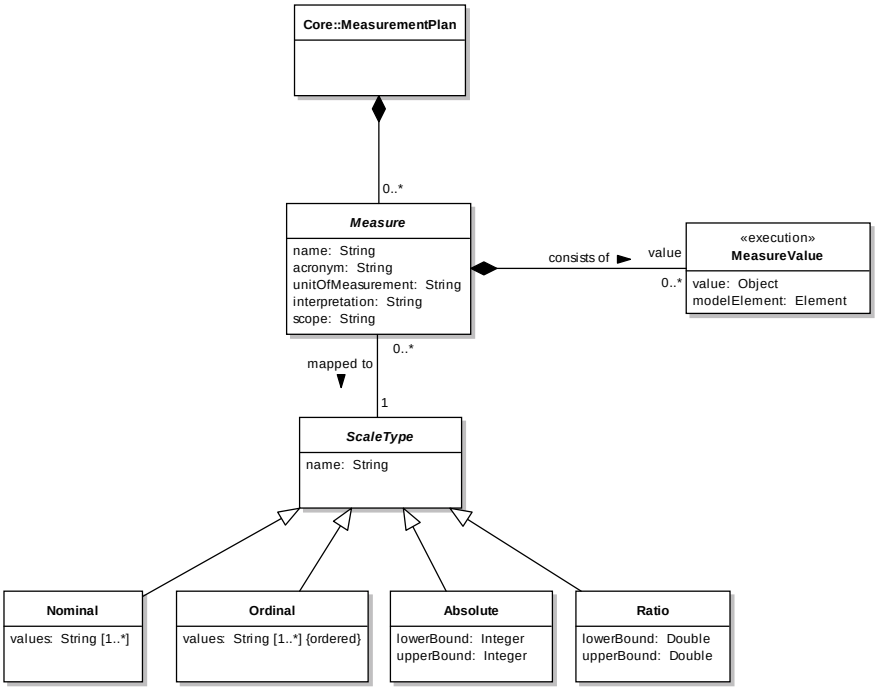


Abbildung 5.17: Mess-Metamodell: Kennzahlen und Skalen

Definition 5.21 (*Scale*)

Ordered set of values, continuous or discrete, or a set of categories to which a measure is mapped. The type of scale depends on the nature of the relationship between values on the scale. Four types of scales are commonly defined: Nominal, Ordinal, Absolute and Ratio (based on [86]).

Die folgenden Ausführungen beziehen sich auf den in Abbildung 5.18 gezeigten Ausschnitt des Mess-Metamodells: Basiskennzahlen (Definition 5.22) sind funktional unabhängig von anderen Kennzahlen und lassen sich durch Messmethoden (Definition 5.23) bestimmen. Abgeleitete Kennzahlen (Definition 5.24) kombinieren mehrere Basiskennzahlen und/oder abgeleitete Kennzahlen mit Hilfe von Messfunktionen (Definition 5.25). Messmethoden und Messfunktionen sind spezielle Messungen. Wir unterscheiden zwei Arten von Messungen: Subjektive Messungen benötigen das menschliche Urteilsvermögen und sind ausschließlich für die Berechnung von Basiskennzahlen vorgesehen. Checklisten (Definition 5.28) gruppieren subjektive Messungen. Objektive Messungen basieren auf numerischen Regeln. Sowohl eine

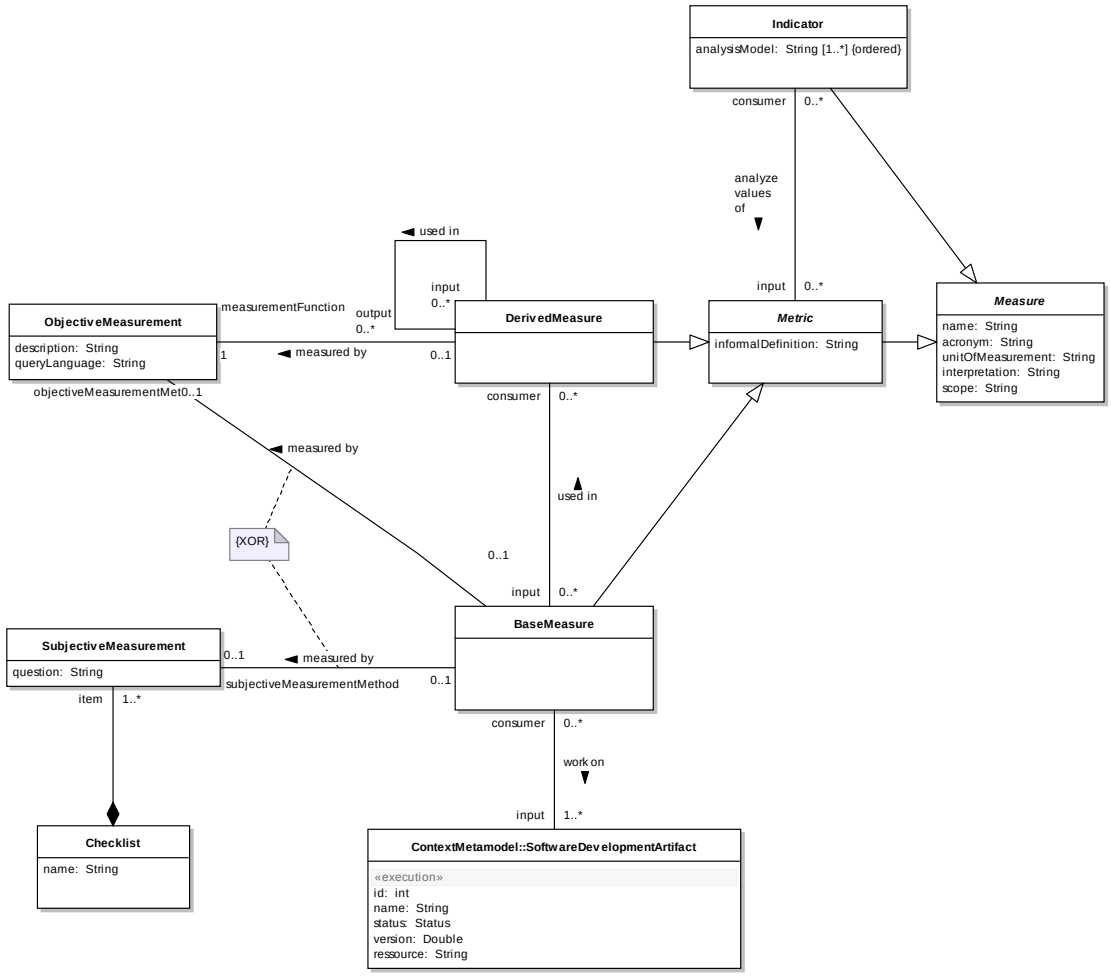


Abbildung 5.18: Mess-Metamodell: Metriken und Indikatoren

Basiskennzahl mit assoziierter Messmethode als auch eine abgeleitete Kennzahl mit assoziierter Messfunktion sowie benötigten Messmethoden bezeichnen wir als Metrik (Definition 5.27). Indikatoren (Definition 5.26) analysieren Metriken, indem die entsprechenden Werte mit Entscheidungskriterien wie Grenzwerten, Zielwerten, o.ä. verglichen werden.

Definition 5.22 (*Base Measure*)

Measure defined in terms of a model quality attribute and the measurement method for quantifying it. A base measure is functionally independent of other measures (based on [86]).

Definition 5.23 (*Measurement Method*)

A measurement method is a set of operations having the object of determining a value of a base measure with respect to a specified scale (based on [86]).

Definition 5.24 (*Derived Measure*)

Measure that is defined as a function of two or more values of base measures or derived measures (based on [86]).

Definition 5.25 (*Measurement Function*)

Algorithm or calculation performed to combine two or more base measures (based on [86]).

Definition 5.26 (*Indicator*)

Measure that provides an estimate or evaluation of defined qualities derived from an analysis model with respect to information needs. An analysis model is an algorithm or calculation combining one or more base and/or derived measures with associated decision criteria (based on [86]).

Definition 5.27 (*Metric*)

A metric is (1) a base measure associated with a measurement method or (2) a derived measure associated with a measurement function and measurement methods (based on [87]).

Definition 5.28 (*Checklist*)

A checklist is a list of questions used during a quality inspection to help identify common defects in a software model.

Der Zusammenhang zwischen einem Qualitätsmodell und einem Messplan besteht darin, dass Metriken Qualitätsattribute quantifizieren und Indikatoren Qualitätsattribute qualifizieren (vgl. Abbildung 5.19).

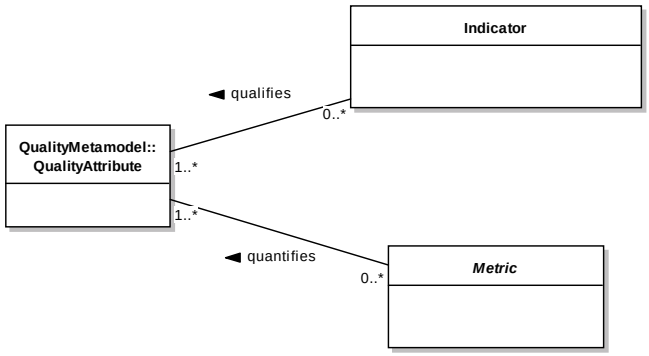


Abbildung 5.19: Mess-Metamodell: Zusammenhang zum Qualitäts-Metamodell

5.4 Regeln für die kontextsensitive Qualitätsplanung

Im vorangegangenen Abschnitt 5.3 haben wir die systematische Vorgehensweise des MQP-Prozesses detailliert vorgestellt. Die Erstellung von MQPs erfordert die Unterstützung des jeweiligen Projektteams und ist mit einem hohen kreativen Aufwand verbunden. Infolgedessen bindet die Erstellung eines MQPs menschliche Ressourcen und ist als recht aufwändig zu beurteilen.

Dabei wiederholen sich viele Informationen in verschiedenen MQPs. Im MQP-Prozess werden Ziele und daraus abgeleitete Fragen ausgehend von den zu untersuchenden Entwicklungsartefakten und deren Kontexten definiert. Mit Hilfe von Qualitätscharakteristiken und Qualitätsattributen wird dann ein Qualitätsmodell aufgebaut, welches letztendlich über Indikatoren und Metriken konkret gemessen werden kann. Die einzelnen Schritte bauen dabei aufeinander auf. Während viele Ziele und die dafür zu beantwortenden Fragen direkt vom Kontext des zu untersuchenden Softwaremodells abhängen, bauen die weiteren Schritte auf den Fragen auf und versuchen diese zu beantworten. Dadurch hängen die Ergebnisse aller weiteren Schritte auch indirekt vom gegebenen Kontext ab. Ziele und Fragen wiederholen sich dabei in ähnlichen Kontexten. Die zu formulierenden Ziele und Fragen und die dazugehörigen Qualitätscharakteristiken und Qualitätsattribute sowie die Indikatoren und Metriken stellen Erfahrungswissen dar, welches für einen Kontext gesichert werden sollte, damit dieses Wissen dementsprechend auch in anderen MQPs wiederverwendet werden kann. Der Kontext des zu untersuchenden Softwaremodells ist also der entscheidende Einflussfaktor für einen MQP und besitzt das Potential, den

Aufwand für die MQP-Erstellung zu reduzieren.

Um MQPs möglichst effizient anhand von vorhandenem Erfahrungswissen aufzubauen, bieten wir ein einfaches Regelkonzept an. MQP-Regeln ermöglichen einem Qualitätsmanager Erfahrungswissen für die Qualitätsplanung zu sichern und für kommende Qualitätsplanungen wiederzuverwenden. Mit Hilfe des Regelkonzepts lassen sich anhand von MQP-Regeln und dem Kontext des zu prüfenden Softwaremodells Elemente wie konkrete Qualitätscharakteristiken und Kennzahlen vorschlagen, die in einen MQP eingefügt werden sollten.

Durch die Erweiterung des in Abschnitt 5.3 vorgestellten MQP-Ansatzes um ein Regelkonzept ermöglichen wir eine Teilautomatisierung der MQP-Schritte Identifizierung der Informationsbedürfnisse (vgl. Abschnitt 5.3.2), Erstellung des Qualitätsmodells (vgl. Abschnitt 5.3.3) und Dokumentation der Messungen (vgl. Abschnitt 5.3.4).

Zuerst stellen wir in Abschnitt 5.4.1 den grundlegenden Aufbau und die Funktionsweise einer MQP-Regel vor. In Abschnitt 5.4.2 erläutern wir, wie MQP-Regeln kombiniert und auf bestehende MQPs angewandt werden. Die Ausdrucksmächtigkeit des Regelkonzepts erweitern wir in Abschnitt 5.4.3, indem wir negative Anwendungsbedingungen einführen. Anschließend stellen wir in Abschnitt 5.4.4 den allgemeinen Ablauf bei der Anwendung von MQP-Regeln vor und integrieren das Regelkonzept in den MQP-Prozess. Abschließend beschreiben wir in Abschnitt 5.4.5 die Grenzen der Anwendbarkeit des Regelkonzepts.

Die grundlegenden Konzepte der MQP-Regeln sind im Rahmen einer Studienarbeit entwickelt und anhand diverser Beispiele ausführlich beschrieben worden [45].

5.4.1 Aufbau der Regeln

Die Regeln funktionieren nach einem Wenn-Dann-Prinzip und gliedern sich dementsprechend in zwei Teile (siehe Abbildung 5.20). Der erste Teil (Wenn-Teil) legt fest, welche Kontextfaktoren auftreten und wie sie mit Attributwerten belegt sein müssen, damit die im zweiten Teil (Dann-Teil) festgelegten Elemente in den MQP eingefügt werden. Die Elemente des Wenn-Teils werden mit Elementen des Dann-Teils verlinkt. Diese Links werden durch eine MQP-Regel ebenfalls zum MQP hinzugefügt und sind folglich dem Dann-Teil zuzuordnen.

Je weniger Kontextfaktoren im Wenn-Teil gefordert sind, desto allgemeiner kann die Regel angewandt werden. Im Extremfall könnte der Wenn-Teil für eine Regel leer

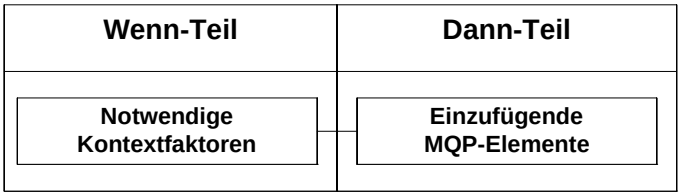


Abbildung 5.20: Aufbau einer MQP-Regel

sein, sofern sie für alle Softwaremodelle angewandt werden soll. Mit zunehmender Anzahl an Kontextfaktoren im Wenn-Teil wird eine Regel spezifischer.

MQP-Regeln sind über dem MQP-Metamodell getypt. Allerdings dürfen MQP-Regeln gegen Untergrenzen von Kardinalitäten und gegen geforderte Attributbelegungen verstoßen. Im Wenn-Teil lässt sich das Kontext-Metamodell und im Dann-Teil alle übrigen Metamodelle instanziieren. Wie im MQP-Prozess auch dürfen Klassen und Attribute des MQP-Metamodells nicht belegt werden, die mit dem Stereotypen «execution» markiert sind.

Negative Anwendungsbedingungen erlauben es, MQP-Regeln zu formulieren, die bestimmte Elemente auf der Wenn-Seite ausschließen. Um die negativen Anwendungsbedingungen (vgl. Abschnitt 5.4.3) basierend auf dem MQP-Metamodell beschreiben zu können, ist eine Erweiterung erforderlich. Dafür führen wir einen dritten Stereotypen ein:

- Der Stereotyp «rule» kennzeichnet Klassen und Attribute, die ausschließlich im Zusammenhang mit MQP-Regeln von Bedeutung sind.

Wir passen dafür das MQP-Metamodell an und fügen den Kontextfaktoren das Attribut *negativeApplicationCondition* vom Typ Boolean hinzu. Dieses Attribut ist mit dem Stereotyp «rule» markiert. Abbildung 5.21 zeigt den Ausschnitt aus dem aktualisierten Metamodell.

5.4.2 Kombination von Regeln

Durch die Kombination mehrerer MQP-Regeln wird es vorkommen, dass gleiche Elemente mehrfach in den zu erstellenden MQP eingefügt werden müssten. Dieser Effekt könnte beispielsweise bei der Kombination feingranularer MQP-Regeln wie denen in Abbildung 5.22 und 5.23 auftreten. Diese zwei Regeln sind für den gleichen

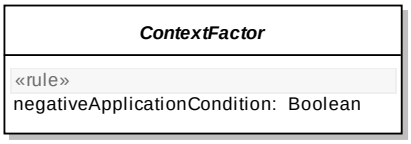


Abbildung 5.21: MQP-Metamodell: Erweiterung um negative Anwendungsbedingung

Kontext spezifiziert und verbinden die gleiche Qualitätscharakteristik mit verschiedenen Qualitätsattributen. Die Qualitätscharakteristik sollte in einem MQP aber nur einmalig auftreten.

Deshalb führen wir Regeln zusammen, die gleiche Elemente in einen MQP einfügen. Die Gleichheit der Elemente wird anhand des Elementtyps und der Attributbelegungen erkannt. Abbildung 5.24 zeigt ein Beispiel für die Zusammenführung der Regeln aus den Abbildungen 5.22 und 5.23 bei ihrer Anwendung. Diese Lösungsstrategie funktioniert auch bei der Kombination von MQP-Regeln mit verschiedenen Wenn-Teilen.

Trotzdem sollten Elemente verschiedenen Typs nicht innerhalb von einem MQP mit dem gleichen Namen bezeichnet werden. Da verschiedene Elementtypen verschiedene Bedeutungen haben, macht es wenig Sinn, z.B. eine Qualitätscharakteristik und ein Qualitätsattribut gleich zu benennen. Während dies innerhalb eines MQPs oder einer MQP-Regel noch offensichtlich ist, könnte es aber passieren, dass unter demselben Namen in zwei verschiedenen MQP-Regeln zwei verschieden Elementtypen vorhanden sind. Solche MQP-Regeln widersprechen sich und ihre Kombination ist daher nicht zulässig.

5.4.3 Negative Anwendungsbedingung

Um bestimmte Sachverhalte ausdrücken zu können, ist es erforderlich, neben Kontextelementen, die für die Anwendung einer Regel vorhanden sein sollen, auch Kontextelemente ausschließen zu können. Soll zum Beispiel eine Regel zur Anwendung kommen, wenn ausschließlich ein Klassendiagramm vorliegt, jedoch dieselbe Regel nicht verwendet werden, wenn neben dem Klassendiagramm noch ein Statechart zur Eingabe gehört, lässt sich dieser Sachverhalt nur im Ausschlussverfahren abbilden.

Dazu lassen sich MQP-Regeln auf der Wenn-Seite um negative Anwendungsbe-

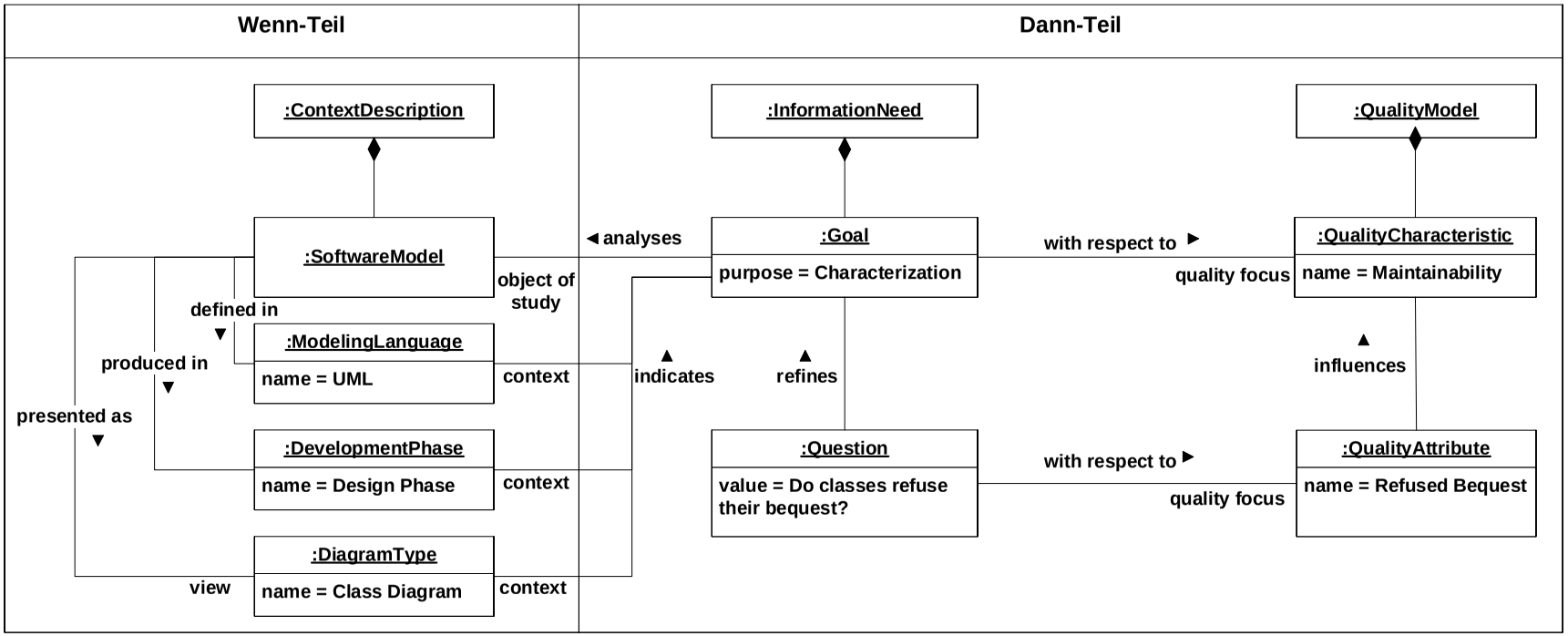


Abbildung 5.22: Beispiel für eine MQP-Regel

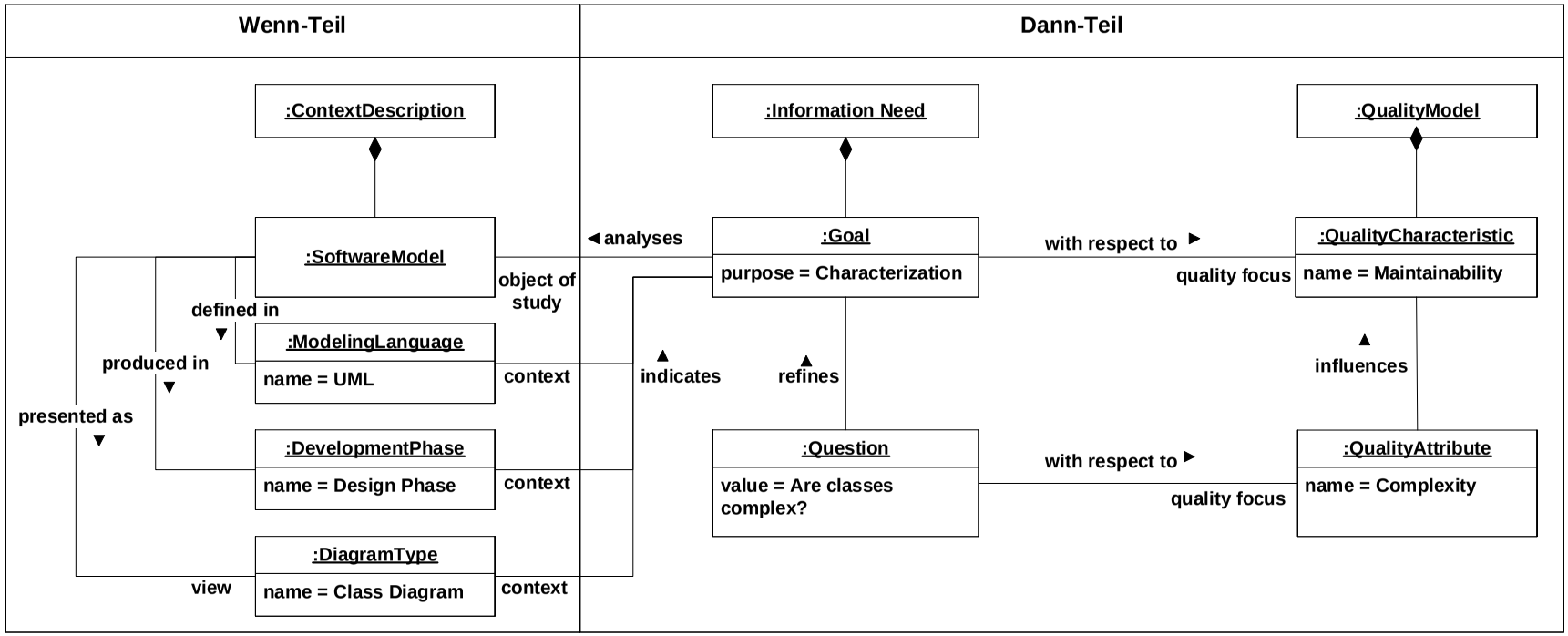


Abbildung 5.23: Beispiel für eine MQP-Regel

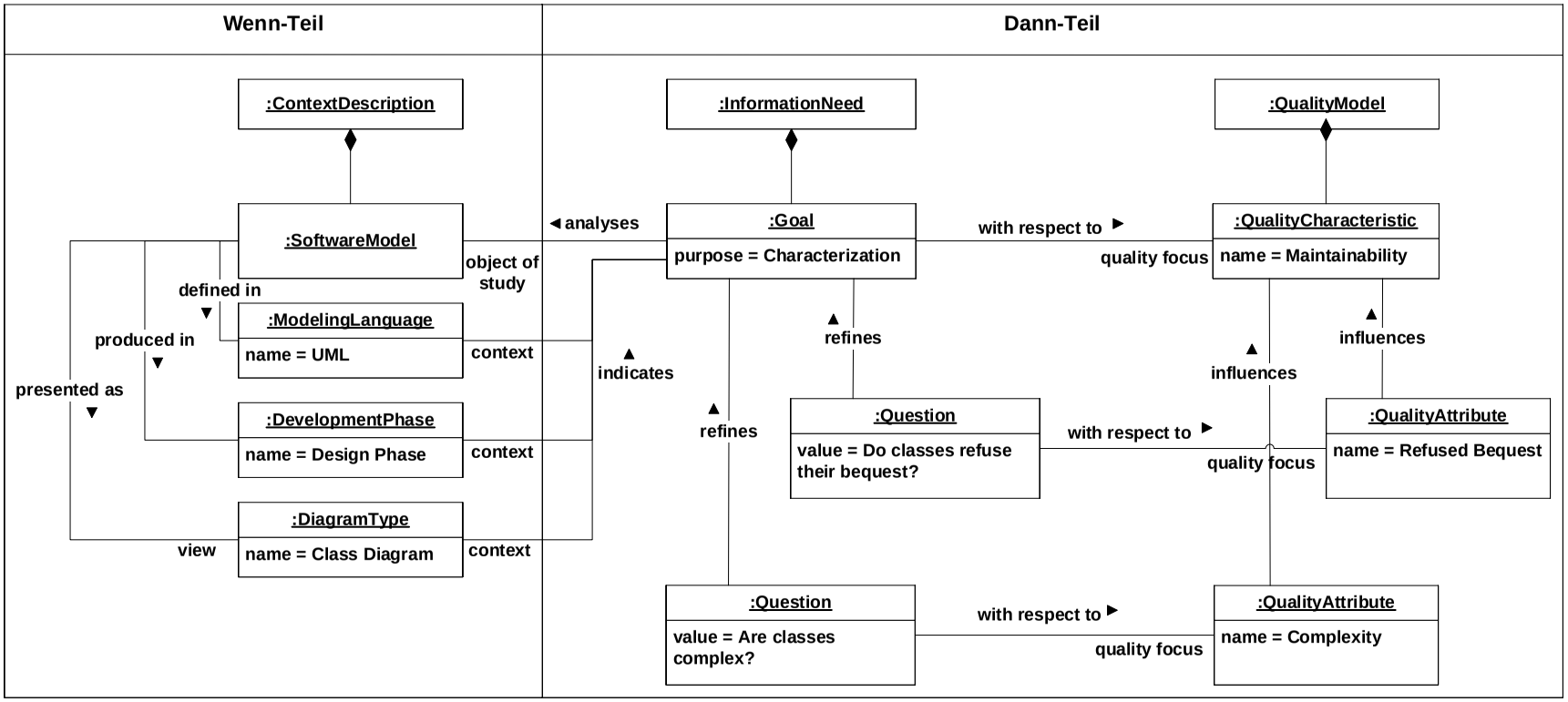


Abbildung 5.24: Beispiel für eine Kombination zweier MQP-Regeln

dingungen ergänzen. Abbildung 5.25 zeigt eine MQP-Regel, die für die Bestimmung der *Größe eines Softwaremodells* die *Anzahl der Klassen* als Metrik vorschlägt, wenn das Softwaremodell durch Klassendiagramme aber *nicht* durch Statecharts dargestellt wird.

5.4.4 Anwendung der Regeln

Die Anwendung der MQP-Regeln auf eine gegebene Kontextbeschreibung erfolgt in vier wesentlichen Schritten:

1. **Feststellung des Kontextes wie in Abschnitt 5.3.1 beschrieben und Selektion anzuwendender Regeln:** Zunächst muss ein MQP existieren, der ein zu untersuchendes Softwaremodell und weitere Informationen über dessen Kontext enthält. Anschließend wird eine Vorauswahl an Regeln ermittelt, die zur gegebenen Kontextbeschreibung passen. Basierend auf dieser Menge von MQP-Regeln können dann eine oder mehrere zur automatisierten Erstellung der restlichen Teile des MQPs selektiert werden.

2. **Konsistenzprüfung der MQP-Regeln:** Diese Menge von ausgewählten Regeln wird anschließend einer einfachen Konsistenzprüfung unterzogen. Hierbei wird überprüft, ob in verschiedenen Regeln die gleichen Namen für verschiedene Elementtypen verwendet wurden und inwiefern Attributbelegungen wie z.B. Definitionen für gleich benannte Elemente gleichen Typs übereinstimmen. Unstimmigkeiten deuten auf eine inhaltliche Inkompatibilität der Regeln zueinander hin. Würden z.B. zwei Regel *Komplexität* als Qualitätscharakteristik vorsehen, aber unterschiedlich definieren, so liegt in beiden Regeln vermutlich ein verschiedenes Qualitätsverständnis vor. Werden derart inkompatible Regeln gefunden, hat der Anwender die Möglichkeit, diese Regeln für die Anwendung auszuschließen oder ggf. vor der Anwendung anzupassen.

   Nicht alle Konsistenzprobleme können derart automatisiert aufgedeckt werden. Während unterschiedliche Attributbelegungen für gleich benannte Elemente gleichen Typs oder auch gleiche Namensgebungen für Elemente verschiedenen Typs in unterschiedlichen Regeln als Indiz für semantische Inkompatibilität gewertet werden kann, sind andere semantische Widersprüche nicht offensichtlich erkennbar. Sie bedürfen einer Interpretation dessen, was mit der Regel

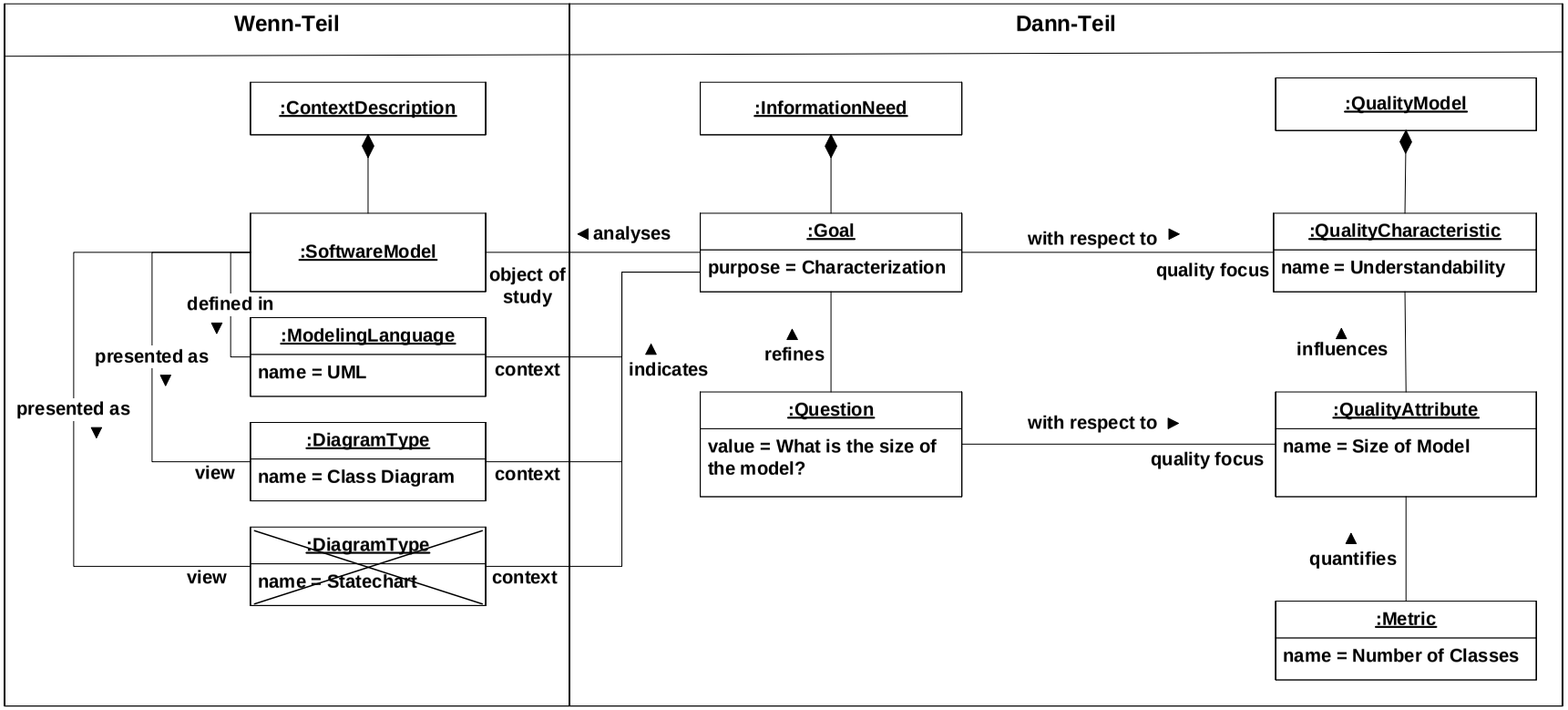


Abbildung 5.25: Beispiel für eine MQP-Regel mit negativer Anwendungsbedingung

bezweckt werden sollte. Denkbar sind hier Kombinationen von zu allgemein formulierten Regeln oder ein unterschiedliches Qualitätsverständnis bei den Erstellern der Regeln. Das hier vorgestellte Regelkonzept stößt in diesem Fall an seine Grenzen, auf die wir im folgenden Abschnitt 5.4.5 genauer eingehen.

3. **Anwendung der MQP-Regeln:** Im Anschluss an die Kompatibilitätsprüfung werden die Regeln abgearbeitet. Hierbei kommen nur Regeln zur Anwendung, die zur gegebenen Kontextbeschreibung passen.

4. **Fortfahren mit dem MQP-Prozess:** Abschließend kann der MQP nach dem in den Abschnitten 5.3.2 bis 5.3.4 beschriebenen Verfahren vervollständigt werden.

5.4.5 Grenzen der Anwendbarkeit

Durch ungeeignetes Kombinieren von nicht aufeinander abgestimmten Regeln kann es zu widersprüchlichen oder unerwünschten Ergebnissen im generierten MQP kommen. Grundsätzlich lässt sich sagen, dass die Kombination von Regeln verschiedener Herkunft zu Problemen führen kann, da Unterschiede im Qualitätsverständnis zu unterschiedlichen Einordnungen von Qualitätscharakteristiken und -attributen führen können. Auch die Tatsache, dass gleiche Elemente auf der Dann-Seite anhand ihres Namens erkannt werden müssen, kann hier schon zu Problemen führen, wenn inhaltlich gleiche Qualitätscharakteristiken oder Qualitätsattribute verschieden benannt sind (z.B. Verständlichkeit und Klarheit als Qualitätscharakteristiken).

Ein weiteres Problem stellen Regeln dar, die zu allgemein formuliert sind. Durch diese Regeln würden unter Umständen Elemente fälschlicherweise in einen MQP eingefügt werden. Ein Qualitätsmanager kann zwar durch die Verwendung negativer Anwendungsbedingungen MQP-Regeln sehr exakt formulieren. Allerdings muss er dafür bei der Formulierung einer MQP-Regel die relevanten Sonderfälle beachten und kann hierbei Fehler machen.

5.5 Qualitätssicherung von Qualitätsplänen

Nachdem wir im letzten Abschnitt 5.4 ein Regelkonzept eingeführt haben, um die Erstellung von MQPs effizienter gestalten zu können, widmen wir uns jetzt der

Frage, wie wir die Qualität eines MQPs sicherstellen können. Mit der Qualitätssicherung eines MQPs zielen wir auf dessen Effektivität im Projekteinsatz ab, so dass dieser MQP die analytische Qualitätssicherung möglichst *gut* vorbereitet und die Projektverantwortlichen zu den *richtigen* Entscheidungen führt.

Bevor wir eine Reihe konkreter Maßnahmen beschreiben, weisen wir auf die fundamentale Problematik bei der Qualitätssicherung von MQPs hin, die darin besteht, dass ein Qualitätsplan wiederum anhand eines Qualitätsplans bewertet wird. Jeder Versuch, einen unzweifelbar guten Qualitätsplan aufzustellen, muss aber daran scheitern, dass dessen Güte wiederum sicher nachgewiesen werden muss, so dass man zu keinem Ende kommt (in Anlehnung an das Münchhausen-Trilemma in [142]). Aufgrund dieser Problematik und dem Gebot der Wirtschaftlichkeit führen wir vier Maßnahmen ein, die mit begrenztem Aufwand anzuwenden sind.

1. Maßnahme - Review eines MQPs: In Abschnitt 4.3.2 haben wir bereits Reviews im Bereich der Softwareentwicklung vorgestellt. Reviews lassen sich nicht nur auf Softwareentwicklungsartefakte wie Softwaremodelle und Quellcode anwenden. Im Rahmen eines Reviews kann grundsätzlich jedes Arbeitsergebnis manuell überprüft werden. In der ausführlichen Prozessbeschreibung zur Anwendung des GQM-Ansatzes in [69] wird ein abschließender Review gefordert, in dem die Korrektheit eines GQM-Plans überprüft werden soll. Allerdings werden keine weiteren Handlungsanweisungen empfohlen.

Deshalb schlagen wir für den Review eines MQPs die in Tabelle 5.8 dargestellte Checkliste vor. Diese Checkliste kann von Vertretern des Projektteams genutzt werden, um die Korrektheit eines MQPs anhand einfacher Fragestellungen zu prüfen. Die Fragestellungen orientieren sich an den in Abschnitt 2.4 definierten Anforderungen an Qualitätsprüfungen und sollen für jede einzelne Messung beantwortet werden. Demzufolge wird der MQP anhand der aus der Qualitätsplanung resultierenden Messungen bewertet.
Die Fokussierung auf Messungen halten wir im Rahmen eines kurzen Reviews

- für *ausreichend*, weil ein erneuter top-down Durchlauf des MQP-Prozesses keinen Mehrwert verspricht und

- für *sinnvoll*, weil durch eine Beurteilung der Messungen hinsichtlich der anderen MQP-Teile eine bottom-up Analyse des MQPs ermöglicht wird und sich

Für jede Messung sind folgende Fragen zu beantworten:	
Anforderung	**Frage**
Nützlichkeit	Ist die Messung nützlich, d.h. relevant für die Entscheidungsfindung? Falls ja: Ist ihre Nützlichkeit anhand des MQPs nachvollziehbar dokumentiert?
Nützlichkeit	Ist die Messung redundant, also bereits durch eine andere Messung abgedeckt?
Verständlichkeit	Ist die Messung vollständig beschrieben (siehe Definition, Skala, Interpretation, Einheit der Messung usw.)?
Verständlichkeit	Ist die informelle Definition der Messung verständlich?
Für jede subjektive Messung sind zudem folgende Fragen zu beantworten:	
Anforderung	**Frage**
Wirtschaftlichkeit	Kann die Messung mit akzeptablem Aufwand durchgeführt werden?
Zuverlässigkeit	Ist für die Messung unmissverständlich dargelegt, wie sie zu erheben ist?
Für die Menge aller Messungen ist noch folgende Frage zu beantworten:	
Anforderung	**Frage**
Vollständigkeit	Sind alle wichtigen Messungen im MQP enthalten?

Tabelle 5.8: Checkliste für MQPs

bei diesem Vorgehen Messungen, die man erwartet hätte, aber nicht im MQP findet, direkt als fehlend markieren lassen.

2. Maßnahme - Invarianten für MQPs: Das in Abschnitt 5.3 eingeführte Metamodell soll Qualitätsplanungen für Softwaremodelle ganz unterschiedlicher Kontexte erlauben und muss einem Qualitätsmanager aus diesem Grund bei der Erstellung eines MQPs große Freiheiten einräumen. Es sind eine ganze Reihe sowohl sehr spezieller als auch grundlegender Einschränkungen für einen MQP denkbar, die durch das bestehende MQP-Metamodell noch nicht abgedeckt sind.

Z.B. ermöglicht das Qualitäts-Metamodell einem Qualitätsmanager redundante Einflussbeziehungen zwischen Qualitätsattributen und Qualitätscharakteristiken zu modellieren. Solche Konstellationen würden wir gerne verbieten, können dies allerdings ausschließlich auf Basis eines Metamodells nicht. Zudem sind einige Kardinalitäten mit $0..*$ belegt, die in bestimmten Qualitätssicherungsszenarien weiter eingeschränkt werden können.

Deswegen bieten wir einem Qualitätsmanager zusätzlich die Möglichkeit an, Invarianten auf Basis von OCL zu formulieren, die dem MQP-Metamodell hinzugefügt werden können. Solche OCL-Invarianten wirken auf das entsprechende Metamodell restriktiv und schränken dadurch die Menge möglicher Aussagen stärker ein.

Im Folgenden zeigen wir einige sehr unterschiedliche Beispiele für OCL-Invarianten (siehe Tabelle 5.9). Einige dieser Invarianten wirken sich auf bestimmte in Abschnitt 2 aufgestellte Anforderungen sowohl zum Vorteil als auch zum Nachteil aus.

1. MQP-Invariante: Subjektive Messungen verbieten	
Vorteil:	Qualitätsprüfungen lassen sich vollständig automatisieren und infolgedessen mit geringeren Kosten durchführen (Anf. 9 $\oplus$).
Nachteil:	Einige Qualitätsziele können ohne subjektive Messungen nicht mehr überprüft werden (Anf. 4 $\ominus$).
OCL:	**context** BaseMeasure **inv**: self.subjectiveMeasurement->isEmpty()

b.w.

2. MQP-Invariante: Jedes Qualitätsattribut mit mindestens einer Messung verbinden	
Vorteil:	Alle Qualitätsziele werden zumindest partiell überprüft (Anf. 4 $\oplus$).
Nachteil:	Unter Umständen müssen Messungen hinzugefügt werden, deren Nutzen unklar ist (Anf. 7 $\ominus$).
	Tendenziell steigt die benötigte Menge subjektiver Messungen (Anf. 9 $\ominus$).

OCL:	`context QualityAttribute`
	`inv: (self.metric->union(self.indicator))`
	`->notEmpty()`

3. MQP-Invariante: Qualitätsmodell hat Baumstruktur	
Vorteil:	Weist ein Qualitätsmodell eine Baumstruktur auf, dann beeinflussen Qualitätsziele ausschließlich übergeordnete Qualitätsziele und einzelne Zweige im Qualitätsmodell können isoliert betrachtet werden. Qualitätsmodelle, die eine Baumstruktur aufweisen, werden auch als orthogonal bezeichnet. Solche Qualitätsmodelle verbessern die Verständlichkeit der Qualitätsziele (Anf. 3 $\oplus$) und vereinfachen später die Interpretation von Messwerten (Anf. 10 $\oplus$).
Nachteil:	Evtl. kann eine Baumstruktur für das Qualitätsmodell nicht realisiert werden (Anf. 1 und 2 $\ominus$).

OCL:	`context QualityAttribute`
	`inv: self.qualityCharacteristic->size() = 1`
	
	`context QualityCharacteristic`
	`inv: self.qualityCharacteristic->size() = 1`

b.w.

4. MQP-Invariante: OCL-Interface impliziert OCL-Messungen	
Vorteil:	Bietet das verwendete Modellierungswerkzeug ein OCL-Interface an, lassen sich die Messungen direkt auf dem Modell ausführen (9 $\oplus$).
Nachteil:	Unter Umständen stellt sich die Extraktion der ermittelten Messwerte aus dem Modellierungswerkzeug als schwierig dar, so dass eine Visualisierung und längerfristige Speicherung der Messwerte behindert wird (Anf. 10 und 13 $\ominus$).
OCL:	**context** ContextDescription **inv**: (self.softwareModel.tool->notEmpty() **and** self.softwareModel.tool.ocl-interface = true) **implies** self.modelQualityPlan.measurementPlan.measure->forAll(m:Measure | (m.oclIsTypeOf(DerivedMeasure) **implies** m.measurementFunction.queryLanguage = OCL)) **and** ((m.oclIsTypeOf(BaseMeasure) **and** m.objectiveMeasurementMethod->notEmpty()) **implies** m.objectiveMeasurementMethod.queryLanguage = OCL)

b.w.

5. MQP-Invariante: Keine redundanten Einflussbeziehungen im Qualitätsmodell	
Vorteil:	Die Modellierung von redundanten Einflussbeziehungen im Qualitätsmodell sollte grundsätzlich unterbleiben. Redundante Einflussbeziehungen sind überflüssig und erschweren nur die Verständlichkeit der Qualitätsziele (Anf. 3 $\oplus$).
Nachteil:	keine
OCL:	**context** QualityAttribute **inv**: self.qualityCharacteristic->intersection (self.qualityCharacteristic->iterate(qc: QualityCharacteristic; acc:**Set**=**Set**{}| acc->union(qc)->union(qc. getAllSuperiorCharacteristics()))).isEmpty() -Auxiliary Function **context** QualityCharacteristic def: getAllSuperiorCharacteristics():**Set** = **if** (self.characteristic->isEmpty()) **then** **Set**{} **else** self.characteristic->iterate(qc: QualityCharacteristic; acc:**Set**=**Set**{}| acc->union(qc)->union(qc. getAllSuperiorCharacteristics()))

Tabelle 5.9: MQP-Invarianten

3. Maßnahme - Pilotprojekt für neu erstellte MQPs: Die Anwendung eines MQPs ist risikobehaftet, weil

1. für die Automatisierung objektiver Metriken, die Kalkulation von Indikatoren sowie die Darstellung der Messergebnisse technische Herausforderungen bestehen, die eine erfolgreiche Qualitätssicherung behindern können und

2. der Aufwand für die Erhebung subjektiver Metriken zuerst schwer abzuschät-

zen ist, ein zu großer Aufwand zu Ressourcenengpässen und schließlich zum Scheitern der Qualitätssicherung führen kann.

Deshalb empfehlen wir einen neu erstellten MQP zuerst im Rahmen eines Pilotprojekts einzusetzen und zu erproben, damit Risiken reduziert werden können. An ein solches Pilotprojekt stellen wir zwei zentrale Anforderungen:

1. Die Kontextbeschreibung des Pilotprojekts muss zum MQP passen.
2. Das Pilotprojekt verfügt über einen ausreichenden Spielraum (z.B. Zeit, Personal) für zusätzliche Qualitätssicherungsaktivitäten. Folglich darf sich das Pilotprojekt selbst nicht in Verzug befinden.

4. Maßnahme - Validierung von MQPs: Nach der Anwendung des MQP-Ansatzes zur Qualitätssicherung von Softwaremodellen sollte das Projektteam seine Erfahrungen wieder in die Qualitätsplanung einfließen lassen. Deshalb bietet sich für die Validierung eines MQPs ein Feedback-orientiertes Verfahren an. Für GQM-Pläne existiert bereits mit V-GQM eine entsprechende Methode [138], die aufgrund der Ähnlichkeit von MQPs zu GQM-Plänen auch von uns eingesetzt werden kann.

Bei der Anwendung von V-GQM auf MQPs müssen einige Ergänzungen beachtet werden:

- Bei der Validierung von Messungen sind auch die Entscheidungskriterien der Indikatoren zu überprüfen.
- Die Analyse der Fragen bzw. Ziele umfasst auch die Überprüfung der Definitionen von Qualitätsattributen bzw. Qualitätscharakteristiken.

Diese Adaption von V-GQM auf den MQP-Ansatz bezeichnen wir im Folgenden als *V-MQP*. Abbildung 5.26 stellt V-GQM (links) und V-MQP (rechts gegenüber).

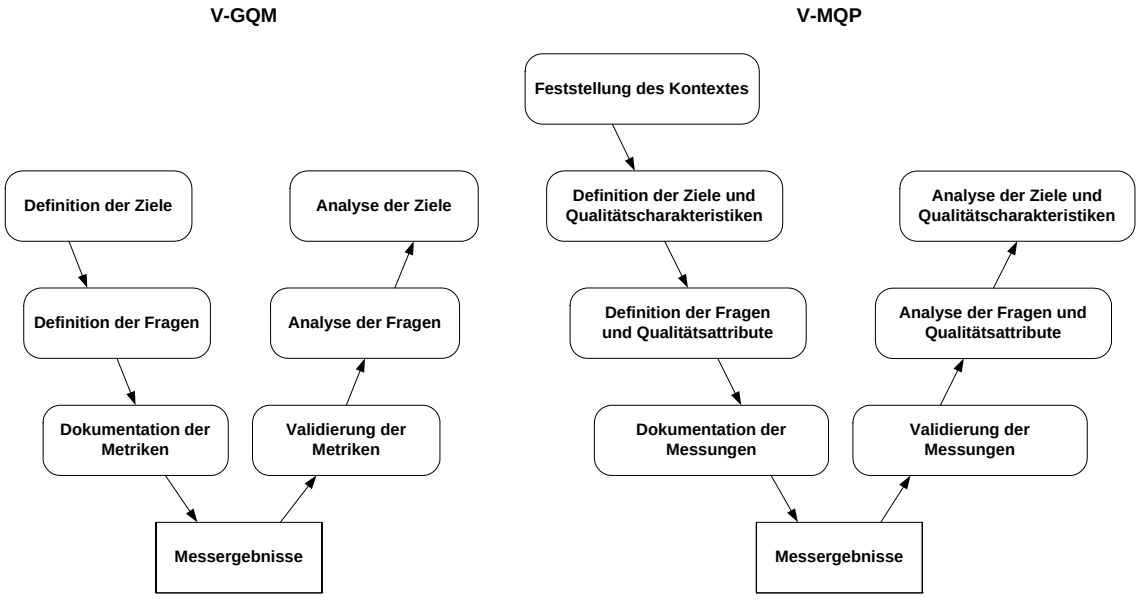


Abbildung 5.26: Gegenüberstellung von V-GQM und V-MQP

Kapitel 6

Fallstudien

Im vorherigen Kapitel 5 haben wir den MQP-Ansatz ausführlich eingeführt. Dafür haben wir in Abschnitt 5.2 ein kurzes Beispiel vorgestellt. In den übrigen Abschnitten haben wir uns diverser Beispiele in Form von MQP-Fragmenten bedient, um gezielt einzelne Aspekte des Ansatzes zu erklären. Diese Herangehensweise hat sich für die Einführung des MQP-Ansatzes bewährt. Allerdings haben wir bis jetzt zwei zentrale Aspekte außer Acht gelassen:

- Der MQP-Ansatz soll sich für die ganzheitliche Qualitätsbewertung von Softwaremodellen eignen. Diese Zielsetzung erfordert die Erstellung umfangreicher Qualitätspläne, die eine ganze Reihe an Messungen beinhalten.
- Wir haben zwar den Kontext eines Softwaremodells als herausragenden Einflussfaktor für die Qualitätsplanung herausgestellt und stichhaltig begründet. Zusätzlich möchten wir diese Kernaussage anhand mehrerer MQPs zweifelsfrei belegen, indem wir ausgehend von verschiedenen Kontexten sehr spezialisierte MQPs entwickeln.

Deshalb stellen wir im Folgenden drei Fallstudien vor, in denen sehr verschiedene MQPs mittlerer Größe entwickelt wurden. Jeder MQP eignet sich dabei für die Qualitätsbewertung von Softwaremodellen eines ganz speziellen Kontextes. Die Kontexte der unterschiedlichen Softwaremodelle decken einen Großteil der in Abschnitt 5.3.1 vorgestellten Kontextfaktoren ab.

Die erste Fallstudie in Abschnitt 6.1 zeigt einen MQP für Analysemodelle, die nach einer besonderen Entwicklungsmethode erstellt sind. Abschnitt 6.2 enthält einen MQP für Entwurfsmodelle. Die dritte Fallstudie in Abschnitt 6.3 bezieht sich

auf einen MQP für Testmodelle. Alle drei MQP-Beispiele sind in Englisch formuliert. Zur zweiten und dritten Fallstudie gibt es jeweils eine spezielle Werkzeugunterstützung.

Jede Fallstudie umfasst zum Ersten die Dokumentation eines kontextsensitiven MQPs sowie zum Zweiten eine Bewertung, in der wir seine speziellen und allgemeinen Bestandteile exemplarisch hervorheben und das Potential für seine Verwendung diskutieren. Diese Bewertungen bilden die Grundlage für den abschließenden Vergleich der verschiedenen MQPs in Abschnitt 6.4.

Jede Dokumentation eines MQPs setzt sich aus der Beschreibung des Kontextes, der Erfassung der Informationsbedürfnisse, der Definition des Qualitätsmodells und der Aufstellung eines Messplans zusammen. Für die Dokumentation der MQPs verwenden wir eine tabellen-orientierte Darstellung, wie wir sie bereits für unser erstes MQP-Beispiel in Abschnitt 5.2 eingesetzt haben. Wir verzichten auf die Modellierung entsprechender Objektdiagramme, die die abstrakte Syntax basierend auf dem MQP-Metamodell wiedergeben würden. Die tabellen-orientierte Darstellung ermöglicht die Visualisierung aller wesentlichen Informationen und ist aufgrund der übersichtlichen Angabe von Attributwerten deutlich kompakter als Objektdiagramme. Im Folgenden führen wir die verwendeten Tabellenschemata kurz ein und geben die Zusammenhänge zwischen ihnen an.

Für die **Kontextbeschreibung** benutzen wir folgendes Tabellenschema:

- *Context Factor* verweist auf Konzepte des MQP-Paketes ContextMetamodel.
- *Value* gibt die jeweilige Ausprägung des entsprechenden Kontextfaktors an.

Dadurch ergibt sich für die Kontextbeschreibung folgender Tabellenkopf:

Context Factor	**Value**

Die **Informationsbedürfnisse** strukturieren wir mit Hilfe zweier Tabellenschemata. Für Ziele geben wir die folgenden Informationen an:

- *Object of Study* benennt das zu prüfende Untersuchungsobjekt.
- *Purpose* gibt den Zweck der Qualitätsprüfung an.
- *Quality Focus* benennt die zu untersuchende Qualitätscharakteristik.

- *Viewpoint* legt den Personenkreis fest, der die Messergebnisse nutzt, und kann auf Rollen aus der Kontextbeschreibung verweisen.
- *Context* verweist auf eine Menge von Kontextfaktoren, die in der Kontextbeschreibung enthalten sind.

Dadurch ergibt sich für die Zieldefinition folgendes Tabellenschema:

Goal Dimension	**Value**
Object of Study	
Purpose	
Quality Focus	
Viewpoint	
Context	

Für Fragen setzen wir das untere Tabellenschema ein:

- *Goal* verweist auf ein zuvor beschriebenes Ziel.
- *Value* gibt die jeweilige Fragestellung an.
- *QA* benennt das Qualitätsattribut, das mit der Fragestellung assoziiert ist.

Dadurch ergibt sich für die Spezifikation der Fragen folgender Tabellenkopf:

Goal	**Value**	**QA**

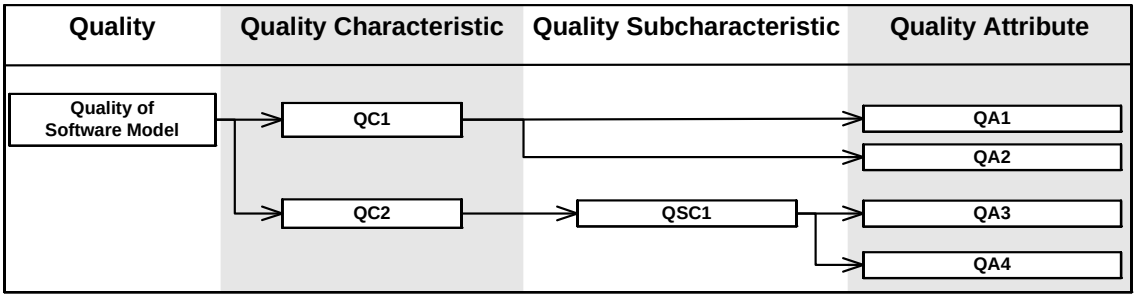
Zur Definition des **Qualitätsmodells** nutzen wir das folgende Tabellenschema:

- *QC* (bzw. *QA*) bezeichnet die Qualitätscharakteristik (bzw. das Qualitätsattribut) und kann auf den Qualitätsschwerpunkt eines Zieles (bzw. einer Frage) verweisen.
- *Definition* gibt eine Definition der Qualitätscharakteristik (bzw. des Qualitätsattributes) an.

Dadurch ergibt sich für die Definition des Qualitätsmodells folgender Tabellenkopf:

QC (bzw. **QA**)	**Definition**

Zusätzlich visualisieren wir das Qualitätsmodell und verdeutlichen dort die Abhängigkeiten zwischen den Qualitätscharakteristiken und Qualitätsattributen:



Aufgrund der großen Anzahl an Messungen haben wir die detaillierten Messdefinitionen in den Anhang A ausgegliedert. Deshalb verwenden wir für die **Zuordnung von Messungen zu Qualitätsattributen** folgendes Tabellenschema:

- *QA* benennt das Qualitätsattribut, das mit Hilfe der Messung bestimmt wird.
- *Name* bezeichnet die Messung.
- *Appendix* verweist auf die Nummer und Seite der jeweiligen Tabelle im Anhang, in der weitere Details zur Messung zu finden sind.
- *Ref* verweist auf die entsprechende Literatur, in der die Messung publiziert wurde. Von uns entwickelte und hier erstmals vorgestellte Messungen markieren wir durch *NEW*.

Dadurch ergibt sich folgender Tabellenkopf für die Zuordnung von Messungen zu Qualitätsattributen:

QA	**Name**	**Appendix**	**Ref**

6.1 Modell-Qualitäts-Plan für Analysemodelle

Der in diesem Abschnitt vorgestellte MQP eignet sich für die Qualitätsbewertung von Softwaremodellen in der Analysephase, die nach einer in der Vorlesung *Softwareentwurf* an der Universität Paderborn gelehrten Entwicklungsmethode entwickelt werden.

Ausgehend von Anwendungsfällen, die nach der Anforderungsdefinition als Use Case Diagramme vorliegen und einen Überblick über die Funktionalität des Systems und die beteiligten Nutzer geben, werden durch eine objektorientierte Dekomposition Klassen- und Sequenzdiagramme modelliert.

Für die Definition der objektiven Metriken verwenden wir OCL-Anfragen, weil keine Annahmen über den Einsatz eines bestimmten Modellierungswerkzeugs vorliegen und die für die UML-Standardisierung zuständige Object Management Group (OMG) OCL als Ergänzung zur UML spezifiziert. OCL als Anfragesprache stößt leider in Einzelfällen an seine Grenzen, da z.B. die mathematische *Wurzelfunktion* nicht vorgesehen ist. Bei der Auswahl objektiver Metriken haben wir uns bemüht, möglichst viele bestehende Metriken wiederzuverwenden. Für die Überprüfung einiger Qualitätsattribute reichen diese jedoch nicht aus, so dass wir auch neue Messungen entwickelt haben. Die Anzahl subjektiver Metriken haben wir auf ein Minimum beschränkt, damit der MQP einen möglichst hohen Automatisierungsgrad erreicht. Eine spezielle MQP-Implementierung gibt es für diese Fallstudie nicht.

6.1.1 Kontextbeschreibung

Context Factor		**Value**
Development Phase		detailed object-oriented analysis
Purpose of Model		integrated description of behavior and structure of the software design
Role(s)	Name	Project Manager
	Description	Ensures the success of a project by minimizing risk throughout the lifetime of the project

b.w.

Context Factor		Value
	Name	Software Architect
	Description	Makes high-level design choices and dictates technical standards, including coding standards, tools, or platforms
	Name	Modeler
	Description	Develops the software system on a conceptual level
Modeling Language	Syntax	UML 2.1.1 metamodel and well-formedness rules
	Semantic	UML 2.1.1 semantics given in English
	Formalization Level	semi formal
Diagram Type(s)	Diagram Type	class diagram
	Purpose of Diagram	specification of the basic software architecture
	Diagram Type	sequence diagram
	Purpose of Diagram	refinement of product functions by relevant scenarios
Development Dependency	Description	vertical inter-model consistency dependency
	Input Artifact	a software model as part of the software requirements specification
	Development Phase	requirements definition
	Diagram Type	use case diagram
	Purpose of Diagram	product functions: overview towards the functionality of the software system and the involved users
	Development Method	object-oriented decomposition

Tabelle 6.2: Kontextinformationen eines Analysemodells

6.1.2 Informationsbedürfnisse

Goal Dimension	Value
Object of Study	Analyze the **software model**
Purpose	for the purpose of **evaluation**
Quality Focus	with respect to **semantic quality**
Viewpoint	from the viewpoint of **project manager** and **software architect**
Context	in the context of the given **development dependency** and the different **diagram types**.
Object of Study	Analyze the **software model**
Purpose	for the purpose of **evaluation**
Quality Focus	with respect to **maintainability**
Viewpoint	from the viewpoint of the **software architect**
Context	in the context of the **purpose of model** and the **development phase**.
Object of Study	Analyze the **software model**
Purpose	for the purpose of **evaluation**
Quality Focus	with respect to **understandability**
Viewpoint	from the viewpoint of the **modeler**
Context	in the context of the **purpose of model**.

Tabelle 6.3: Ziele für die Qualitätsbewertung von Analysemodellen

Goal	Value	QA
Semantic Quality	Is the software model complete?	Completeness
	Is the software model consistent?	Intra-model Consistency
	Are the model elements valid?	Validity
Understandability	Does the software model include statements that are speculative general?	Speculative Generality

b.w.

Goal	Value	QA
	How complex is the behavior specified by the software model?	Behavior Complexity
	How complex is the structure specified by the software model?	Structural Complexity
	Does the software model adhere to modeling conventions?	Modeling Conventions
Maintainability	How strong are classes coupled?	Coupling
	Are classes cohesive?	Cohesion
	Does the software model describe a three-tier architecture?	Three-Tier Architecture
	How are the services allocated to classes?	Service Allocation
	How is the data allocated to classes?	Data Allocation

Tabelle 6.4: Fragen für die Qualitätsbewertung von Analysemodellen

6.1.3 Qualitätsmodell

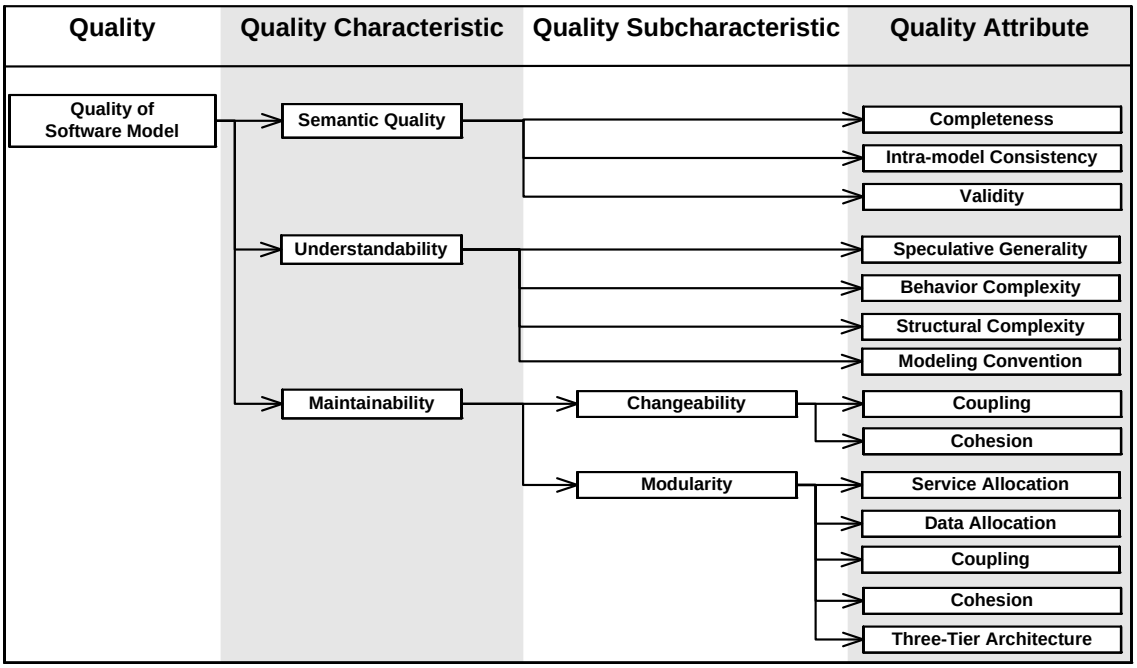


Abbildung 6.1: Qualitätsmodell für Analysemodelle

QC	Definition
Semantic Quality	Semantic Quality is the correspondence between the software model and the domain.
Understandability	The effort required to understand the software model.
Maintainability	The capability of the software model to be modified.
Changeability	The capability of the software model to enable a specified modification to be designed, documented and coded.
Modularity	The extend to which the parts of system specified by a software model are systematically structured and separated such that they can be understood in isolation.

Tabelle 6.5: Definition der Qualitätscharakteristiken

QA	Definition
Completeness	A software model is complete when it represents all relevant features of the domain.
Intra-model Consistency	A software model is intra-model consistent when its statements do not contain a contradiction.
Validity	Validity means that all statements made in the software model are correct and relevant to the domain.
Speculative Generality	Classes are designed to have features that in fact are not required.
Behavior Complexity	The degree of difficulty in predicting the behavior and cooperation of a system defined by a software model.
Structural Complexity	The degree of difficulty in understanding the static structure of a system defined by a software model.
Modeling Convention	The degree of conformance to modeling conventions. A modeling convention is a rule or guideline used by a project team for labeling model elements in a software model.
Coupling	The degree of interaction or dependency of a class to other related classes in order to function correctly.
Cohesion	The degree to which the methods within a class are related to one another.
Service Allocation	The degree of balanced service allocation to classes.
Data Allocation	The degree of balanced data allocation to classes.
Three-Tier Architecture	The degree of adherence to an architecture in which the presentation, the application processing and the data management are logically separated. Communication should only occur between directly related tiers.

Tabelle 6.6: Definition der Qualitätsattribute

6.1.4 Messplan

QA	Name	Appendix	Ref
Completeness	Use Case without SD	Tab. 83 on p. 308	[104]
	Method not called in SD	Tab. 19 on p. 272	[104]
	Class not in SD	Tab. 6 on p. 262	[104]
Intra-model Consistency	Object has no Class in CD	Tab. 22 on p. 274	[104]
	Messages without Method	Tab. 17 on p. 270	[104]
	Law of Demeter Violations	Tab. 74 on p. 303	NEW
Validity	Messages without Name	Tab. 18 on p. 271	[104]
	Abstract Classes in SD	Tab. 1 on p. 259	[103]
	Classes without Methods	Tab. 8 on p. 263	[103]
Speculative Generality	Association with Navigation	Tab. 2 on p. 260	NEW
	Association with unlimited Cardinalities	Tab. 3 on p. 261	[25]
Behavior Complexity	Messages	Tab. 16 on p. 270	NEW
Structural Complexity	Associations and Attributes	Tab. 4 on p. 261	[64] & [6]
Modeling Convention	Unnamed Associations	Tab. 31 on p. 280	[25]
	Untyped Attributes	Tab. 32 on p. 280	[50]
Coupling	Methods called by a Class	Tab. 20 on p. 273	NEW
	Classes used as Attribute Types	Tab. 7 on p. 263	[106]
	Directly related Classes	Tab. 10 on p. 265	[4]
Cohesion	Relatedness among Methods	Tab. 26 on p. 277	[4]
Service Allocation	Maximum of Operations	Tab. 15 on p. 269	NEW
	Variability of Operations	Tab. 86 on p. 309	NEW
Data Allocation	Maximum of Attributes	Tab. 14 on p. 269	NEW
	Variability of Attributes	Tab. 85 on p. 309	NEW

b.w.

QA	Name	Appendix	Ref
Three-Tier Architecture	Invocation Across Layers	Tab. 12 on p. 267	NEW

Tabelle 6.7: Zuordnung von Messungen zu Qualitätsattributen

6.1.5 Bewertung

Der von uns entwickelte MQP für die Qualitätsbewertung von Analysemodellen beruht u.a. auf dem Zusammenhang zwischen Diagrammen verschiedener Entwicklungsphasen. Dieser Zusammenhang ist in der Kontextbeschreibung in Form einer *Development Dependency* angegeben (siehe Tab. 6.2). Diese vertikale Inter-Modell Konsistenzbeziehung in Kombination mit dem *Purpose of Diagram* für den Einsatz von Sequenzdiagrammen führt zu der noch recht allgemeinen Forderung nach *Completeness* (dt. Vollständigkeit) und im Weiteren zu der sehr speziellen Messung *Use Case without SD*, in der die Verfeinerung von Use Cases durch entsprechende Sequenzdiagramme geprüft wird.

Eine weitere Besonderheit dieses MQPs besteht in dem Qualitätsattribut *Three-Tier Architecture* (dt. Drei-Schichten-Architektur). Allerdings leitet sich diese Forderung nicht unmittelbar und zwangsläufig aus der Kontextbeschreibung ab, sondern wird erst aufgrund einer Fragestellung als Bestandteil der Informationsbedürfnisse aufgenommen. Anstatt dieser Drei-Schichten-Architektur könnte sich ein Softwarearchitekt auch für ein anderes Architekturmuster entscheiden.

Andere Qualitätsattribute wie *Coupling* (dt. Kopplung) oder *Cohesion* (dt. Kohäsion) spielen generell bei der Verteilung von Funktionalitäten auf Klassen bzw. Komponenten in objektorientierten Entwicklungsmethoden eine entscheidende Rolle und sind für diesen MQP weniger speziell. Allerdings hängen die Möglichkeiten, diese Qualitätsattribute zu messen, stark von den eingesetzten Diagrammtypen ab. Nur wenn die benötigten Informationen vorliegen, wie z.B. in Form von Sequenzdiagrammen, kann die Kopplung einer Klasse durch die Messung *Methods called by a Class* bestimmt werden. Ähnliches gilt für die Messungen *Messages*, *Messages without Method* oder *Abstract Classed in SD*.

Die obigen Ausführungen weisen bereits auf den Stellenwert der Kontextbeschreibung für die Auswahl und Entwicklung von Messungen hin. Sie zeigen aber auch die

Grenzen der Kontextbeschreibung auf. Einige Qualitätsattribute entziehen sich einer systematischen Ableitung ausgehend von der Kontextbeschreibung und müssen in der Aktivität *Feststellung der Informationsbedürfnisse* durch die jeweiligen Experten, wie z.B. einem Softwarearchitekten, in die Qualitätsplanung eingebracht werden. Die wiederkehrenden Bestandteile lassen sich gezielt mit Hilfe von MQP-Regeln definieren. Variierende Qualitätsziele, die sich z.B. durch Architekturentscheidungen ergeben, müssen dagegen für jedes zu prüfende Analysemodell angepasst oder ergänzt werden.

Ein MQP zeichnet sich nicht nur durch die Bestandteile aus, die in ihm dokumentiert sind, sondern auch durch diejenigen Bestandteile, die nicht enthalten sind. Analysemodelle enthalten keine Informationen über die Sichtbarkeit von Operationen und Attributen. Infolgedessen enthält der aktuelle MQP keine Messungen zur internen Wiederverwendbarkeit. Derartige Messungen stellen wir im nächsten MQP vor.

6.2 Modell-Qualitäts-Plan für Entwurfsmodelle

Der in diesem Abschnitt vorgestellte MQP basiert im Wesentlichen auf den Ergebnissen aus [150, 25, 175]. Er eignet sich für die Qualitätsbewertung von Softwaremodellen in der Entwurfsphase, die auf der formalen Sprache UML/Z [25] basieren. Eine Möglichkeit der Generierung von Skeleton-Klassen mit Variablendeklarationen, Signaturen, Vor- und Nachbedingungen von Methoden für Java-Implementierungen ausgehend von formalen Modellen wird in [120] aufgezeigt.

Der folgende MQP beschränkt sich ausschließlich auf die *Wartbarkeit* und *Verständlichkeit* von UML/Z-Modellen. Andere Qualitätsziele wie *Semantische Qualität* werden hier nicht betrachtet. Diese Einschränkung wird durch den entwickelten Editor für die Modellierung von UML/Z-Modellen abgeschwächt. Der Editor stellt bereits einen Großteil der Intra-Modell Konsistenzbeziehungen sicher.

Für die Definition und Anwendung des MQPs existiert die Werkzeugunterstützung *RMC* [25]. Darüber hinaus ermöglicht das RMC-Werkzeug eine Weiterverarbeitung der Messungen, um auf Basis der Messergebnisse Refaktorisierungen des Softwaremodells vorzuschlagen und dadurch seine Wartbarkeit und Verständlichkeit zu verbessern.

Damit geeignete Refaktorisierungen besser erkannt werden können, beinhaltet der MQP eine Reihe von Indikatoren, die Entwurfsprobleme anzeigen. Die den Indikatoren zu Grunde liegenden objektiven Metriken sind in den jeweiligen Indikator-Beschreibungen in Anhang A.3 referenziert. Diese objektiven Metriken haben wir ebenfalls als OCL-Anfragen formalisiert, die vom RMC-Werkzeug direkt ausgewertet werden können. Um die Diagnose von Entwurfsproblemen vollständig automatisieren zu können, haben wir auf subjektive Metriken zur Bewertung der Wartbarkeit und Verständlichkeit verzichtet.

6.2.1 Kontextbeschreibung

Context Factor		**Value**
Development Phase		object-oriented design
Purpose of Model		integrated description of behavior and structure of the software design
Role(s)	Name Description	Software Architect Makes high-level design choices and dictates technical standards, including coding standards, tools, or platforms
	Name Description	Programmer Writes the software
Modeling Language	Syntax	UML/Z
	Semantic	CSP-OZ
	Formalization Level	formal
Diagram Type(s)	Diagram Type Purpose of Diagram	class diagram class design specification for implementation
	Diagram Type Purpose of Diagram	statechart specification of behavior of class instances
	Diagram Type Purpose of Diagram	Z description specification of class invariants as well as delta-lists, pre- and post-conditions for methods
Domain		embedded systems
Tool	OCL-Interface	available
	Metamodel-Conformance	ensured by editor

Tabelle 6.8: Kontextinformationen eines Entwurfsmodells

6.2.2 Informationsbedürfnisse

Goal Dimension	Value
Object of Study	Analyze the **software model**
Purpose	for the purpose of **improvement by applying refactorings**
Quality Focus	with respect to **understandability**
Viewpoint	from the viewpoint of the **programmer**
Context	in the context of the **purpose of model** and the **diagram types**.
Object of Study	Analyze the **software model**
Purpose	for the purpose of **improvement by applying refactorings**
Quality Focus	with respect to **maintainability**
Viewpoint	from the viewpoint of the **software architect**
Context	in the context of the **purpose of model** and the **development phase**.

Tabelle 6.9: Ziele für die Qualitätsbewertung von Entwurfsmodellen

Goal	Value	QA
Understandability	How complex is the behavior specified by the software model?	Behavior Complexity
	Does the software model adhere to modeling conventions?	Modeling Conventions
	How complex is the structure specified by the software model?	Structural Complexity
	Is the software model defined in a general way?	Universality
Maintainability	How strong or weak are classes coupled?	Coupling
	Are classes cohesive?	Cohesion

b.w.

Goal	Value	QA
	How are the services allocated to classes?	Service Allocation
	Does the software model provide the reuse of features with slight or no modification?	Internal Reusability
	Is the software model defined in a general way?	Universality

Tabelle 6.10: Fragen für die Qualitätsbewertung von Entwurfsmodellen

6.2.3 Qualitätsmodell

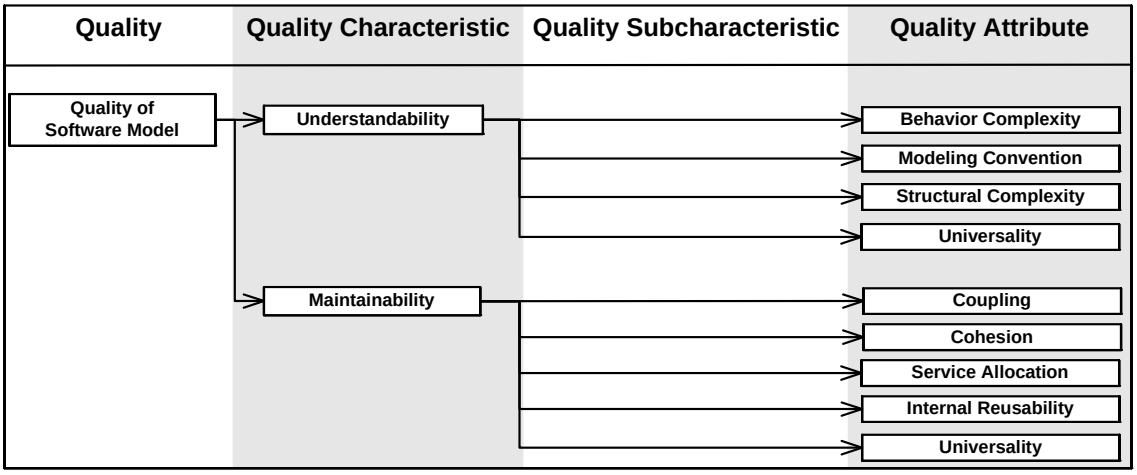


Abbildung 6.2: Qualitätsmodell für Entwurfsmodelle

QC	Definition
Understandability	The effort required to understand the software model.
Maintainability	The capability of the software model to be modified.

Tabelle 6.11: Definition der Qualitätscharakteristiken

QA	Definition
Behavior Complexity	The degree of difficulty in predicting the behavior and cooperation of a system defined by a software model.
Modeling Convention	The degree of conformance to modeling conventions. A modeling convention is a rule or guideline used by a project team for labeling model elements in a software model.
Structural Complexity	The degree of difficulty in understanding the static structure of a system defined by a software model.
Universality	Potential of a software model to be easily adaptable to or to meet changing requirements.
Coupling	The degree of interaction or dependency of a class to other related classes in order to function correctly.
Cohesion	The degree to which the methods within a class are related to one another.
Internal Reusability	The degree of reuse in a model in order to reduce redundancy and thus to reduce effort for code modifications, implementation, and testing.
Service Allocation	The degree of balanced service allocation to classes.

Tabelle 6.12: Definition der Qualitätsattribute

6.2.4 Messplan

QA	Name	Details	Ref
Behavior Complexity	Indicating Hidden Concurrency	Tab. 91 on p. 313	[150]
	Indicating Unnecessary Behavioral Complexity based on Border Crossings	Tab. 96 on p. 315	[150]
	Indicating Unnecessary Behavioral Complexity based on Nesting Level	Tab. 97 on p. 316	[42]

b.w.

QA	Name	Details	Ref
	Transitions to Operations Indicator	Tab. 102 on p. 318	[150]
Modeling Convention	Unnamed Associations Indicator	Tab. 104 on p. 320	[25]
Structural Complexity	Depth of Inheritance Tree Indicator	Tab. 90 on p. 312	[38] [76]
Universality	Unlimited Cardinalities Indicator	Tab. 103 on p. 319	[25]
Coupling	Indicating Too Strong Coupling based Parameter List	Tab. 95 on p. 315	[150]
	Indicating Relatedness among Methods	Tab. 93 on p. 314	[4]
Cohesion	Lack of Cohesion of Methods Indicator	Tab. 98 on p. 316	[79]
	Indicating Tight and Loose Class Cohesion	Tab. 94 on p. 314	[17]
Service Allocation	Lack of Cohesion of Methods Indicator	Tab. 98 on p. 316	[79]
	Small Class Indicator	Tab. 101 on p. 318	[25]
	Large Class Indicator	Tab. 99 on p. 317	[173]
Internal Reusability	Operations Inheritance Factor Indicator	Tab. 100 on p. 317	[28] [29]
	Indicating Refused Bequest	Tab. 92 on p. 313	[150]

Tabelle 6.13: Zuordnung von Messungen zu Qualitätsattributen

6.2.5 Bewertung

Dieser MQP zielt auf die Qualitätsbewertung von formalen Entwurfsmodellen ab, um diese anschließend mit Hilfe von Refaktorisierungen zu verbessern (siehe Tab. 6.9). Durch Refaktorisierungen wird ein Softwaremodell verändert, ohne das spezifizierte und nach außen sichtbare Verhalten anzupassen. Aufgrund dieser besonderen Zielsetzung und der Tatsache, dass keine *Development Dependency* angegeben ist, entfallen in diesem MQP Messungen, die auf die Ermittlung falscher, fehlender oder

widersprüchlicher Modellelemente ausgerichtet sind.

Stattdessen steht ausschließlich die Untersuchung der Qualitätscharakteristiken *Maintainability* (dt. Wartbarkeit) und *Understandability* (dt. Verständlichkeit) im Vordergrund. Damit wir entsprechende Entwurfsprobleme automatisiert erkennen können, benötigen wir Indikatoren. Diese Indikatoren zeigen für Messwerte eindeutig an, ob ein Mangel im Softwaremodell vorliegt oder nicht.

Im Gegensatz zur vorangegangenen Fallstudie bezieht sich dieser MQP auf die Entwurfsphase, so dass in einem zu prüfenden Softwaremodell Sichtbarkeitsinformationen für Operationen und Attribute vorliegen. Dadurch wird in diesem Szenario die Prüfung des Qualitätsattributes *internal reusability* (dt. interne Wiederverwendbarkeit) sinnvoll.

Neben diesen Unterschieden weist der vorliegende MQP auch einige Gemeinsamkeiten zum MQP für Analysemodelle aus Abschnitt 6.1 auf. Diese Gemeinsamkeiten deuten sich bereits anhand der beiden Kontextbeschreibungen an (vgl. dazu die Kontextfaktoren *Purpose of Model: integrated description of behavior and structure of the software design*, *Role: Software Architect* und *Diagram Type: Class diagram* in den Tabellen 6.2 und 6.8) und resultieren in Überschneidungen bei den Qualitätsattributen Kopplung, Kohäsion und strukturelle sowie Verhaltens-Komplexität.

Jedoch müssen diese Qualitätsattribute für formale Entwurfsmodelle besonders gemessen werden. Das Verhalten der Klasseninstanzen wird durch Statecharts spezifiziert. Sequenzdiagramme sind dagegen nicht vorgesehen. Dies hat unmittelbare Auswirkungen auf die Messungen für Verhaltens-Komplexität (vgl. z.B. die Messung *Messages* in Tab. 6.7 mit *Indicating Hidden Concurrency* in Tab. 6.13). Zudem ist der Zusammenhang zwischen den Operationen und Attributen in den *delta-lists* des Diagrammtypen *Z description* enthalten, so dass Messungen wie *Lack of Cohesion of Methods* und *Tight and Loose Class Cohesion* eingesetzt werden können.

Dieser MQP ist für ein sehr spezielles Szenario entwickelt worden. Dies wird insbesondere an der Zielsetzung der Qualitätsbewertung, der Modellierungssprache und dem damit verbundenen Diagrammtypen *Z description* deutlich, der die Angabe von Vor- und Nachbedingungen, Invarianten und delta-lists für Operationen ermöglicht. Die generelle Verwendung dieses MQPs für die Bewertung von UML/Z-Entwurfsmodellen ist möglich, weil er im Gegensatz zum MQP für Analysemodelle aus Abschnitt 6.1 keine Architekturentscheidungen überprüft. Sobald sich jedoch der zu Grunde liegende Kontext ändert, ist die Übertragbarkeit des MQPs aufgrund

der hoch spezialisierten Messungen nur bedingt gegeben.

6.3 Modell-Qualitäts-Plan für Testmodelle

Die erste Idee für den folgenden MQP wurde in [174] veröffentlicht. Im Rahmen einer Diplomarbeit ist dieser MQP bedeutend erweitert und auf die Bewertung von AGEDIS-Testmodellen abgestimmt worden [50]. Die Diplomarbeit ist in Deutsch verfasst, so dass wir allein aufgrund der Übersetzung ins Englische einige Anpassungen vornehmen mussten.

AGEDIS steht für Automated Generation and Execution of Test Suites for DIstributed Component-based Software und ist ein dreijähriges von der Europäischen Kommission gefördertes Forschungsprojekt, das sich mit der Automatisierung des modellbasierten Testens beschäftigt hat [77]. Primäres Ziel dieses Projektes war es, eine Methodik für das modellbasierte Testen zu entwickeln. Dazu wurde ein UML-Profil für die verwendeten Testmodelle erstellt und ein Prototyp implementiert.

Die AGEDIS Modeling Language (AML) ist ein UML-Profil, basierend auf der UML 1.4 [169]. In AML werden Klassendiagramme, Zustandsdiagramme und Objektdiagramme genutzt. Viele Möglichkeiten der UML sind in AML eingeschränkt oder gar nicht vorhanden. Beispielsweise gibt es keine Vererbung oder abstrakte Klassen in Klassendiagrammen.

Ein Testmodell in AGEDIS besteht aus der Beschreibung des zu testenden Systems (kurz: SUT für System Under Test) und der Spezifikation der Testfälle. Ein Systemmodell in AGEDIS muss lediglich die für das Testen relevanten Aspekte eines existierenden Systems beschreiben. Neben der Beschreibung des zu testenden Systems werden Testfälle durch Zustands- und Objektdiagramme spezifiziert. Beteiligt an der Systemmodellierung sind die Rollen *Softwarearchitekt* und *Testmanager*. Die Spezifikation der Testfälle führt der Testmanager alleine durch.

Für die Definition und Anwendung des MQPs existiert die Werkzeugunterstützung MATFAM, die im Rahmen der oben erwähnten Diplomarbeit implementiert wurde [50]. Weil das AGEDIS-Werkzeug kein OCL-Interface, dafür aber einen XMI-Export der AML-Testmodelle anbietet, sind die Messungen als XQuery-Anfragen formuliert.

Aufgrund des erheblichen Umfangs dieses MQPs haben wir das Qualitätsmodell in die Bereiche *Qualität der SUT Spezifikation* und *Qualität der Testfallspezifikation* getrennt und entsprechend in zwei Abbildungen visualisiert. Die damit zusammenhängenden Definitionen der Qualitätscharakteristiken und Qualitätsattribute sowie

die Zuordnung der Messungen zu den Qualitätsattributen haben wir dementsprechend ebenfalls aufgeteilt.

6.3.1 Kontextbeschreibung

Context Factor		**Value**
Development Phase		testing
Purpose of Model		specification of system under test (SUT)
		specification of test cases
Role(s)	Name Description	Software Architect Makes high-level design choices and dictates technical standards, including coding standards, tools, or platforms
	Name Description	Test Manager Is tasked with the overall responsibility for the test effort's success
Modeling Language	Name	AGEDIS Modeling Language (AML)
	Syntax	UML 1.4 Profile
Diagram Type(s)		class diagram
		statechart
		object diagram
Tool	XMI-Export	available
	OCL-Interface	not available

Tabelle 6.14: Kontextinformationen eines Testmodells

6.3.2 Informationsbedürfnisse

Goal Dimension	Value
Object of Study	Analyze the **software model**
Purpose	for the purpose of **evaluation**
Quality Focus	with respect to the **Quality of SUT Specification**
Viewpoint	from the viewpoint of the **software architect** and **test manager**
Context	in the context of the development phase **testing**, the purpose of model **specification of SUT**, the modeling language **AML** and **all** diagram types.
Object of Study	Analyze the **software model**
Purpose	for the purpose of **evaluation**
Quality Focus	with respect to the **Quality of Test Case Specification**
Viewpoint	from the viewpoint of the **test manager**
Context	in the context of the development phase **testing**, the purpose of model **specification of test cases**, the modeling language **AML** and the diagram types **statechart and object diagram**.

Tabelle 6.15: Ziele für die Qualitätsbewertung von Testmodellen

Goal	Value	QA
Quality of SUT Specification	Can the SUT be observed and controlled?	Observability and Controllability
	Is the relevant behavior of the SUT completely described?	Behavior Completeness
	Are the used links typed?	Typed Links
	Are the objects typed?	Typed Objects
	Are the guards consistent?	Consistent Guards
	Are the events consistent?	Consistent Events
	Can only relevant attributes be	Relevant Attribute

b.w.

Goal	Value	QA
	accessed?	Access
	Can only relevant operations be accessed?	Relevant Operation Access
	Do dead states exist?	Dead States
	Are the used diagram types necessary?	Necessary Diagram Types
	Are the attributes typed?	Typed Attributes
	Are the objects validly used?	Valid Use of Objects
	Are the attribute types valid?	Valid Attribute Types
	Is the start configuration unambiguous?	Unambiguous Start Configuration
	Are the start states valid?	Valid Start States
	Are the end states valid?	Valid End States
	Are the end states reachable?	Reachable End States
	Are the object names valid?	Valid Object Names
	Are the class names valid?	Valid Class Names
	Are the attribute names valid?	Valid Attribute Names
	Are the operation names valid?	Valid Operation Names
	Are the associations completely labeled?	Complete Labeling of Associations
	Are the statecharts unnecessary complex?	Unnecessary Complexity
	Are the statecharts deterministic?	Determinism in Transitions
	Is the embedded code readable?	Readability of Embedded Code
	Are the state names expressive?	Expressive State Names
Quality of Test Case Specification	Are the interfaces covered completely?	Complete Interface Coverage
	Are the operations consistent?	Consistent Operations
	Are the attributes consistent?	Consistent Attributes
	Are the objects initialized?	Initialized Objects

b.w.

Goal	Value	QA
	Are the object diagrams stereotyped?	Stereotyped Object Diagrams
	Are the used model elements valid?	Valid Model Elements
	Is the test purpose specified by statecharts ?	Test Purpose
	Are the statecharts valid?	Valid Statecharts
	Are the state names expressive?	Expressive State Names
	Is the embedded code readable?	Readability of Embedded Code

Tabelle 6.16: Fragen für die Qualitätsbewertung von Testmodellen

6.3.3 Qualitätsmodell

QC	Definition
Quality of SUT Specification	The SUT specification must be correct and should be understandable.
Correctness	The SUT specification is correct when each of its statements is valid and the set of all statements is complete with respect to the AGEDIS testing domain.
Completeness	The SUT specification is complete when it represents all relevant features of the system to enable the AGEDIS tool to generate and execute test cases.
Intra-model Consistency	The SUT specification is intra-model consistent when its statements do not contain a contradiction.
Minimality	The SUT specification is minimal when all its statements can be processed by the AGEDIS compiler and each statement only appears once in the specification.

b.w.

QC	Definition
Validity	Validity means that all statements made in the SUT specification are feasible and relevant with respect to the AGEDIS method for Test Case generation.
Understandability	The effort required to understand the SUT specification.
Modeling Convention	The degree of conformance to modeling conventions. A modeling convention is a rule or guideline for labeling model elements and that can be derived from the AGEDIS testing domain.
Complexity	The degree of difficulty in predicting the behavior and cooperation of the SUT specification.

Tabelle 6.17: Definition der Qualitätscharakteristiken: Qualität der SUT Spezifikation

QA	Definition
Observability and Controllability	Some operations and attributes must be observable and controllable.
Behavior Completeness	The behavior of all classes should be specified by at least one statechart.
Typed Links	Links must be typed by associations.
Typed Objects	Used objects must be typed by classes.
Consistent Guards	Guards attached to transitions must use variables that are defined in the according class.
Consistent Events	Events attached to transitions must correspond to operations or signals defined in the according class.
Relevant Attribute Access	Only attributes that should be observed and controlled should be marked as observable and controllable.
Relevant Operation Access	Only operations that should be observed and controlled should be marked as observable and controllable.
Dead States	Every state should be reachable from the start state.
Necessary Diagram Types	Only diagram types defined in AML are allowed. AML specifies class diagrams, object diagrams, and

b.w.

QA	Definition
	statecharts. All other diagrams are not processed.
Typed Attributes	Attributes must be typed.
Valid Use of Objects	Objects used in test cases must be created in the start configuration.
Valid Attribute Types	Only attribute types defined by AML are allowed.
Unambiguous Start Configuration	The start configuration must be defined by one object diagram.
Valid Start States	Each statechart must exactly have one start state with outgoing transitions. A start state must not have incoming transitions.
Valid End States	In a statechart only zero or one end states are allowed. An end state must have at least one incoming transition and must not have outgoing transitions.
Reachable End State	An end state of a statechart should be reachable from all other states.
Valid Object Names	Object names must start with a lower case letter and must not include spaces.
Valid Class Names	Class names must start with an upper case letter and must not include spaces.
Valid Attribute Names	Attribute names must start with a lower case letter and must not include spaces.
Valid Operation Names	Operation names must start with a lower case letter and must not include spaces.
Complete Labeling of Associations	Either an association has a name or a role attached to each association end. In addition all cardinalities should be explicitly specified.
Expressive State Names	A state name is expressive, if it represents meaningful information about the state.
Readability of Embedded Code	Embedded code in the SUT specification should be used consistently. Either the code is attached by comments or it is directly placed in the model elements.

b.w.

QA	Definition
Unnecessary Complexity	Statechart is complex without any need.
Determinism in Transitions	All outgoing transitions of a state should be mutually exlusive.

Tabelle 6.18: Definition der Qualitätsattribute: Qualität der SUT Spezifikation

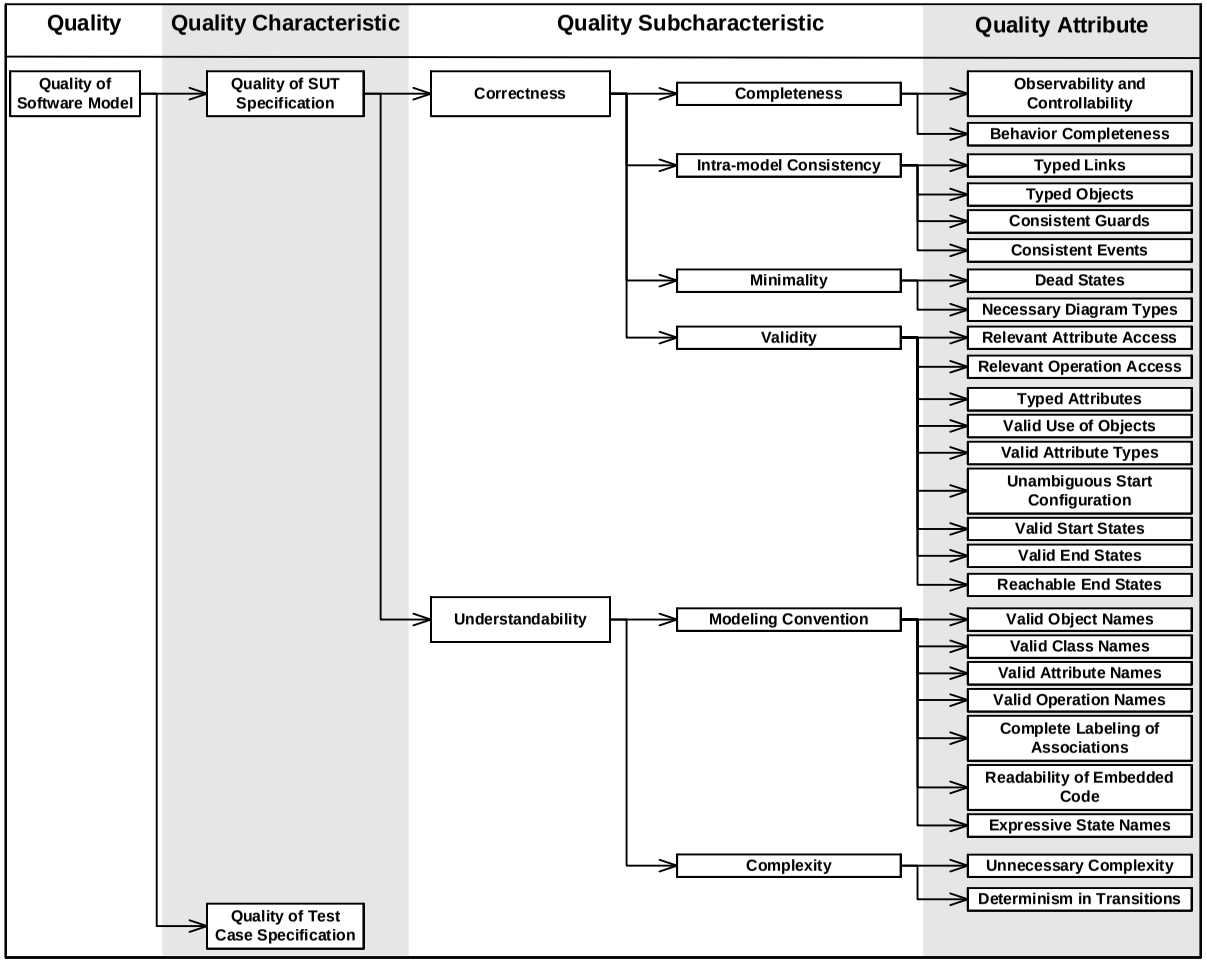


Abbildung 6.3: Qualitätsmodell für Testmodelle - Qualität der SUT-Spezifikation

QC	Definition
Quality of Test Case Specification	The test case specification must be correct and should be understandable.
Correctness	The set of test cases should be correct with respect to the software requirements.
Completeness	The set of test cases is complete when it specifies test cases for all relevant features of the SUT.
Intra-model Consistency	The test case specification is intra-model consistent when its statements do not contain a contradiction and are not in conflict to statements of the SUT specification.
Validity	Validity means that all statements made in the test case specification are correct and relevant with respect to the AGEDIS method for test case generation.
Understandability	The effort required to understand the test case specification.
Modeling Convention	The degree of conformance to modeling conventions. A modeling convention is a rule or guideline for labeling model elements and that can be derived from the AGEDIS testing domain.

Tabelle 6.19: Definition der Qualitätscharakteristiken: Qualität der Testfallspezifikation

QA	Definition
Complete Interface Coverage	The interfaces of the SUT should be tested by according invocations.
Consistent Operations	Only operations defined in the SUT specification can be used in the transitions of statecharts that specify the test purpose.
Consistent Attributes	Only attributes defined in the SUT specification can be used in the transitions of statecharts that specify the test purpose.

b.w.

QA	Definition
Initialized Objects	Objects must be initialized to be used in test cases.
Stereotyped Object Diagrams	Object diagrams that specify system states for a test purpose must be on model level and must be stereotyped by AML stereotypes.
Test Purpose	Statecharts that specify the test purpose must be on model level and must be marked explicitly as test purpose.
Valid Statecharts	Statecharts that specify the test purpose must not be empty and for the included elements used for specifying test purposes only valid tagged values defined by AML are allowed for controlling the test case.
Expressive State Names	A state name is expressive, if it represents meaningful information about the state.
Readability of Embedded Code	Embedded code in the model should be used consistently. Either the code is attached by comments or it is directly placed in the model elements.

Tabelle 6.20: Definition der Qualitätsattribute: Qualität der Testfallspezifikation

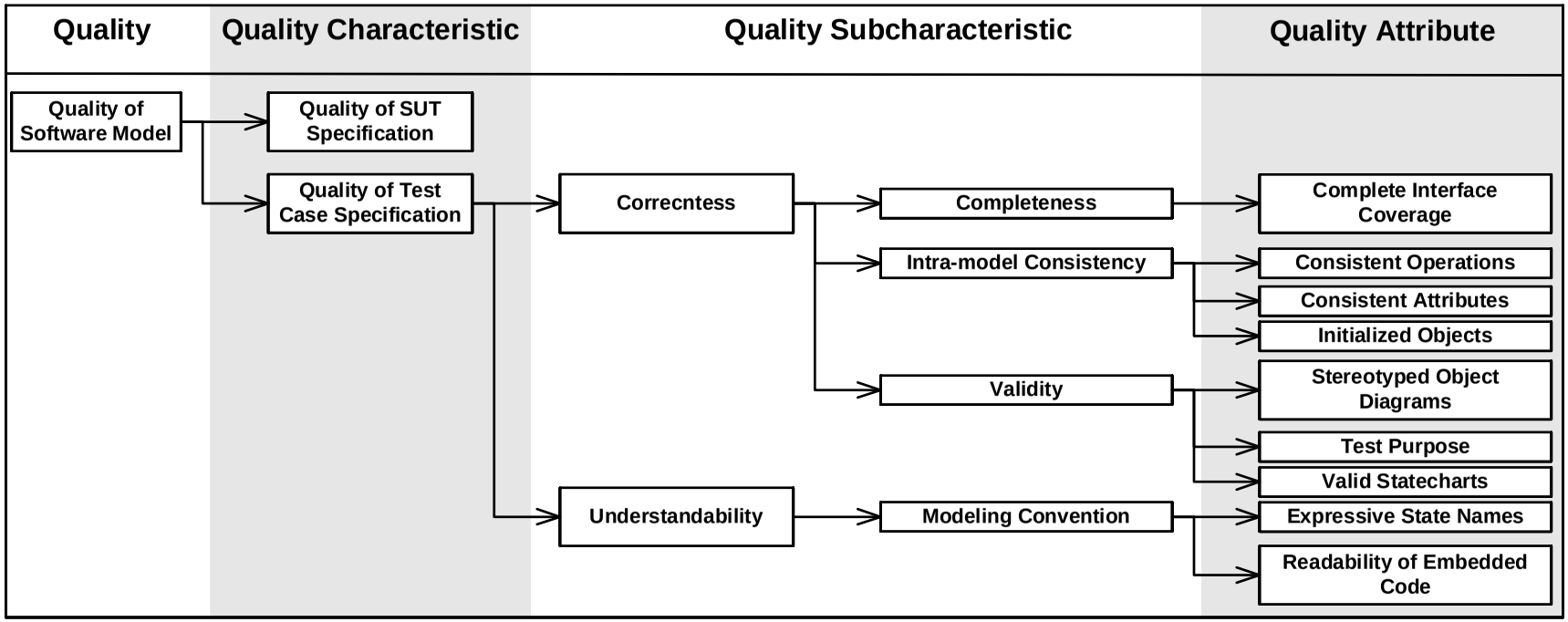


Abbildung 6.4: Qualitätsmodell für Testmodelle - Qualität der Testfallspezifikation

6.3.4 Messplan

QA	Name	Details	Ref
Observability and Controllability	Ratio Testable Attributes	Tab. 59 on p. 295	[50]
	Ratio Testable Operations	Tab. 60 on p. 296	[50]
Behavior Completeness	Statecharts for Classes	Tab. 63 on p. 298	[50]
Typed Links	Correct Instantiation of Associations	Tab. 40 on p. 285	[50]
Typed Objects	Correct Instantiation of Classes	Tab. 41 on p. 285	[50]
Consistent Guards	Variables for Guards	Tab. 87 on p. 310	[50]
Consistent Events	Correct Events for Operations	Tab. 42 on p. 286	[50]
	Correct Events for Signals	Tab. 43 on p. 287	[50]
Relevant Attribute Access	Ratio Observable and Controllable Attributes	Tab. 57 on p. 294	[50]
Relevant Operation Access	Ratio Observable and Controllable Operations	Tab. 58 on p. 294	[50]
Dead States	Non Reachable States	Tab. 80 on p. 306	[18]
	Incoming and Outgoing Transitions	Tab. 45 on p. 288	[50]
Necessary Diagram Types	Available Diagram Types	Tab. 70 on p. 301	[50]
Typed Attributes	Untyped Attributes	Tab. 36 on p. 283	[50]
Valid Use of Object	Non Usable Objects	Tab. 49 on p. 290	[50]
Valid Attribute Types	Attributes of Type String	Tab. 35 on p. 282	[50]
Unambiguous Start Configuration	Initial Object Diagram	Tab. 46 on p. 288	[18]

b.w.

QA	Name	Details	Ref
Valid Start States	Start States	Tab. 62 on p. 297	[50]
	Transitions of Start States	Tab. 68 on p. 300	[50]
Valid End States	Number of End States	Tab. 50 on p. 290	[50]
	Transitions of End States	Tab. 66 on p. 299	[50]
Reachable End State	Non Terminating States	Tab. 81 on p. 307	[18]
Valid Object Names	Object Names Begin Small	Tab. 51 on p. 291	[50]
	Object Names Include Spaces	Tab. 52 on p. 291	[50]
Valid Class Names	Class Names Begin Small	Tab. 38 on p. 284	[50]
	Class Names Include Spaces	Tab. 39 on p. 284	[50]
Valid Attribute Names	Attribute Names Begin Small	Tab. 33 on p. 281	[50]
	Attribute Names Include Spaces	Tab. 34 on p. 282	[50]
Valid Operation Names	Operation Names Begin Small	Tab. 53 on p. 292	[50]
	Operation Names Include Spaces	Tab. 54 on p. 292	[50]
Complete Labeling of Associations	Names or Roles for Associations	Tab. 48 on p. 289	[50]
	Cardinalities for Associations	Tab. 37 on p. 283	[50]
Expressive State Names	Meaningful State Names (SUT)	Tab. 78 on p. 305	[18]
	Waiting States (SUT)	Tab. 88 on p. 310	[18]
	Invariant in State Names (SUT)	Tab. 72 on p. 302	[18]
	Mealy States (SUT)	Tab. 75 on p. 304	[18]
Readability of Embedded Code	Placement of IF-Code (SUT)	Tab. 55 on p. 293	[50]
Unnecessary Complexity	Transitions to Operations	Tab. 67 on p. 300	[150]
Determinism in Transitions	Non Deterministic Guards	Tab. 79 on p. 306	[18]

Tabelle 6.21: Zuordnung von Messungen zu Qualitätsattributen: Qualität der SUT Spezifikation

QA	Name	Details	Ref
Complete Interface Coverage	Interface Coverage	Tab. 47 on p. 289	[50]
Consistent Operations	Available Operations	Tab. 71 on p. 302	[50]
	Usable Operations	Tab. 82 on p. 307	[50]
Consistent Attributes	Attribute Variables for Guards	Tab. 69 on p. 301	[50]
Initialized Objects	Used Objects	Tab. 84 on p. 308	[50]
Stereotyped Object Diagrams	Stereotypes for Object Diagrams	Tab. 61 on p. 297	[50]
Test Purpose	Test Purpose Stereotyp	Tab. 65 on p. 299	[50]
Valid Statecharts	Empty Statecharts	Tab. 44 on p. 287	[50]
	Tagged Values	Tab. 64 on p. 298	[50]
Expressive State Names	Meaningful State Names (Test Cases)	Tab. 77 on p. 305	[18]
	Waiting States (Test Cases)	Tab. 89 on p. 311	[18]
	Invariant in State Names (Test Cases)	Tab. 73 on p. 303	[18]
	Mealy States (Test Cases)	Tab. 76 on p. 304	[18]
Readability of Embedded Code	Placement of IF-Code (Test Cases)	Tab. 56 on p. 293	[50]

Tabelle 6.22: Zuordnung von Messungen zu Qualitätsattributen: Qualität der Testfallspezifikation

6.3.5 Bewertung

Dieser MQP eignet sich für Qualitätsbewertung von AGEDIS-Testmodellen. Ein AGEDIS-Testmodell definiert erstens Testfälle und spezifiziert zweitens die Aspekte eines System Under Test (SUT), die für eine erfolgreiche Testfallgenerierung und -durchführung benötigt werden. Die SUT-Spezifikation als Teil eines AGEDIS-Testmodells beschreibt folglich keine Softwarearchitektur, die als Grundlage für die Implementierung dient. Aus diesem Grund fehlt in der Modellierungssprache AML

auch die Möglichkeit, z.B. Vererbungsbeziehungen oder abstrakte Klassen zu modellieren. Qualitätsziele wie Wartbarkeit sind somit hinfällig und die Überprüfung von Qualitätsattributen wie Kopplung oder Kohäsion entfällt. Insofern unterscheidet sich dieser MQP signifikant von den beiden vorherigen MQPs und dieser Unterschied lässt sich bereits dem *Purpose of Model* der Kontextbeschreibung entnehmen.

Dafür enthält der MQP eine ganze Reihe sehr spezieller Qualitätsattribute, die sehr typisch für Testaktivitäten sind. Dies sind z.B. *Observability and Controllability* (dt. Beobachtbarkeit und Kontrollierbarkeit) sowie *Test Purpose* (dt. Testzweck).

Weil ein Experte für die Beschreibung des SUT benötigt wird, kommt die Rolle des Softwarearchitekten auch in diesem MQP vor. Allerdings stellt der Testmanager eine Spezialität in diesem MQP dar, der die AGEDIS-Testmodelle zur Testautomatisierung einsetzt.

Diese zahlreichen Besonderheiten schlagen sich auf die für Testmodelle sehr speziellen Messungen nieder. Ein weiteres besonderes Kennzeichen dieses MQPs ist die große Anzahl an Messungen. Zudem zeichnet sich dieser MQP durch eine sehr starke Verfeinerung im Qualitätsmodell aus. Qualitätsziele wie *Validity* (dt. Validität) sind in diesem MQP weiter in Qualitätsattribute unterteilt und somit selbst als Qualitätscharakteristiken realisiert. Der Vergleich der Fallstudien zeigt, dass das Abstraktionsniveau für Qualitätsattribute nicht eindeutig festgelegt werden kann.

Die Verwendung dieses MQPs für die Bewertung von AGEDIS-Testmodellen ist nicht eingeschränkt, weil keine domänen- oder architekturspezifischen Qualitätsziele und Messungen enthalten sind. Allerdings ist die Übertragbarkeit dieses MQPs auf andere Softwaremodelle aufgrund der oben genannten Besonderheiten nur bedingt möglich. Die Übertragbarkeit der Messungen auf andere Softwaremodelle lässt sich grob in drei Gruppen einteilen. Erstens existieren eine ganze Reihe von Messungen, die auf die Verständlichkeit von Bezeichnern abzielen (z.B. *Expressive State Names* oder *Complete Labeling of Associations*) und auch in anderen MQPs unabhängig vom gegebenen Modellierungszweck interessant sind. Zweitens sind viele Messungen derart speziell, dass sie zwar evtl. auf andere Testmodelle übertragen werden können, aber nicht für Analysemodelle oder Entwurfsmodelle geeignet sind (z.B. *Interface Coverage*, *Used Objects* oder *Ratio Testable Operations*). Drittens enthält der MQP auch solche Messungen, die ausschließlich für AGEDIS-Testmodelle eingesetzt werden können, weil sie Spezifika von AML berücksichtigen (z.B. *Stereotypes for Object Diagrams* oder *Placement of IF-Code*).

6.4 Vergleich der Fallstudien

Nachdem wir drei sehr verschiedene MQPs vorgestellt und bewertet haben, vergleichen wir sie in diesem Abschnitt ausführlich und schließen damit unsere Ausführungen zu den Fallstudien ab. In diesem Vergleich analysieren wir insbesondere die Auswirkungen der Kontextbeschreibung auf die gegebenen Messpläne. Damit sich dieser Vergleich besser veranschaulichen lässt, klären wir zuerst die folgenden Zusammenhänge:

- Die Formalisierung der objektiven Messungen (vgl. z.B. OCL-Messungen im Anhang A.1.1 und XQuery-Messungen im Anhang A.1.2) wird entscheidend durch den Kontextfaktor *Tool* beeinflusst, weil dieser die Zugriffsmöglichkeiten auf das Softwaremodell beschreibt.

- Für die in der zweiten Fallstudie (siehe Abschnitt 6.2) definierte Zielsetzung der Qualitätsbewertung ist der Zweck *improvement by refactoring* eine Besonderheit. Mit Hilfe dieses MQPs sollen Entwurfsprobleme automatisiert erkannt werden, so dass wir ausschließlich Indikatoren aufgestellt haben.

Im Folgenden können wir von diesen beiden bereits erkannten Zusammenhängen abstrahieren, indem wir die *Formal Definition* in den Messbeschreibungen vernachlässigen und anstatt der Indikatoren die *referenzierten Metriken* betrachten.

In der Tabelle 6.23 stellen wir die Ausprägungen der verbliebenen Kontextfaktoren für die drei MQPs gegenüber. Die grau hinterlegten Zellen heben die wenigen Überschneidungen in den Kontextbeschreibungen hervor. Das zeigt, dass sich die MQPs in den meisten Kontextfaktoren unterscheiden. Auf diese in Tabelle 6.23 dargestellte Gegenüberstellung der Kontextbeschreibungen werden wir uns in dem folgenden Vergleich der Messpläne beziehen.

Als nächstes visualisieren wir in Abbildung 6.5 die Gemeinsamkeiten und Unterschiede der MQPs hinsichtlich der eingebundenen Messungen. Diese Abbildung ist sehr umfangreich und verknüpft viele wichtige Informationen, die wir schrittweise erläutern werden.

Zentrales Gestaltungselement der Abbildung sind die *drei umfassenden Mengen*. Jede Menge bezieht sich auf einen MQP und beinhaltet alle Messungen des entsprechenden MQPs. Dabei sind die Messungen anhand ihrer *Akronyme* repräsentiert. Diejenigen Messungen, die in zwei bzw. in allen drei MQPs vorkommen, sind in den

Values of / Context Factor		MQP for Analysis Models	MQP for Design Models	MQP for Testing Models
Development Phase		detailed object-oriented analysis	object-oriented design	testing
Purpose of Model		integrated description of behavior and structure of the software design	integrated description of behavior and structure of the software design	specification of system under test (SUT)
				specification of test cases
Role(s)		Project Manager	n/a	n/a
		Software Architect	Software Architect	Software Architect
		Modeler	Programmer	Test Manager
Modeling Language	Syntax	UML 2.1.1	UML/Z	AML
	Semantic	UML 2.1.1 semantics given in English	CSP-OZ	n/a
	Formalization Level	semi formal	formal	n/a
Diagram Type(s)		class diagram	class diagram	class diagram
		sequence diagram	statechart	statechart
		n/a	n/a	object diagram
Development Dependency		vertical inter-model consistency dependency	n/a	n/a

Tabelle 6.23: Vergleich der Kontextbeschreibungen

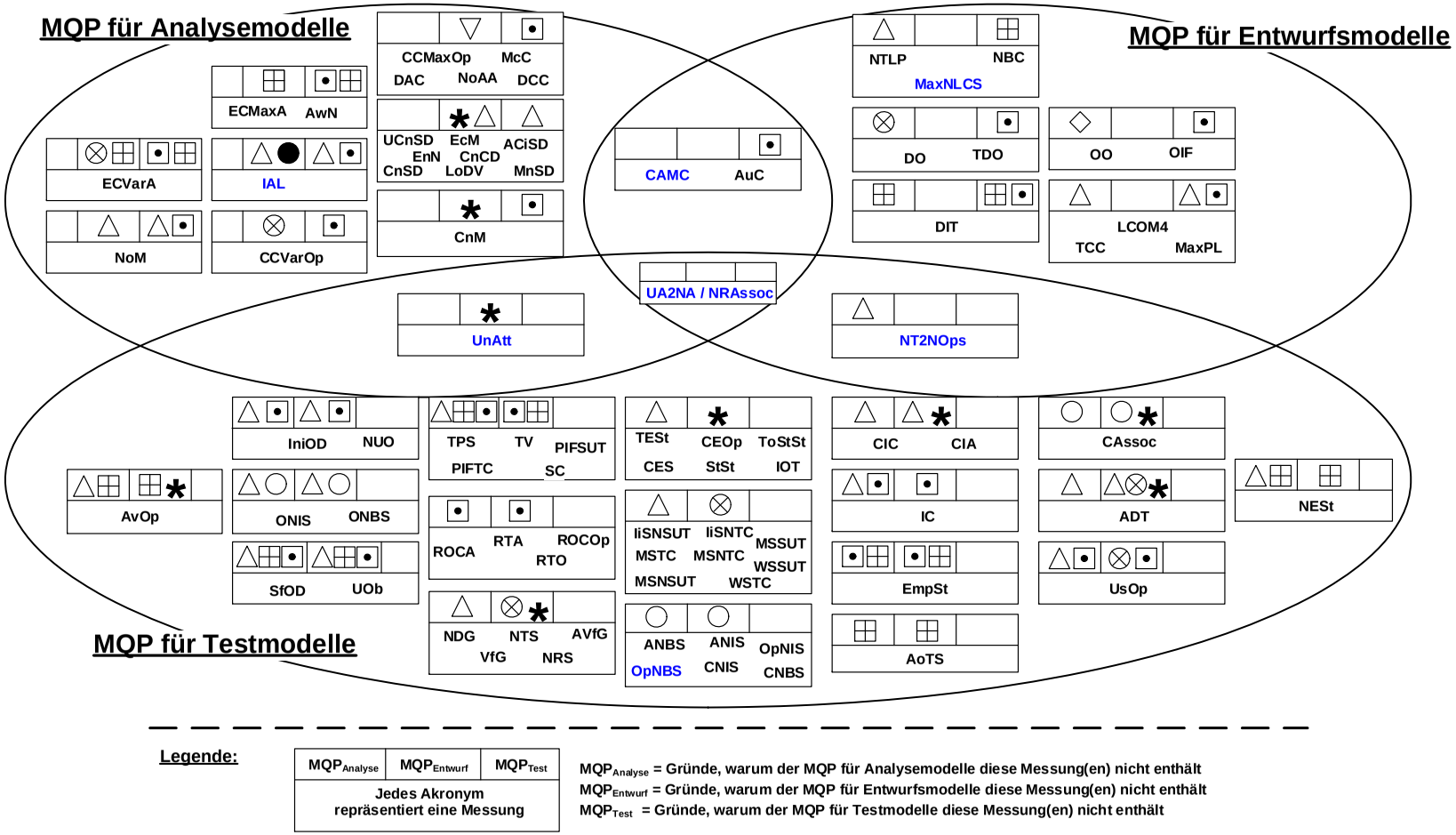


Abbildung 6.5: Vergleich der Messpläne

vier *Schnittmengen* dargestellt. Die übrigen drei *Differenzmengen* enthalten Messungen, die ausschließlich in einem MQP eingebunden sind. Auf diese Weise verdeutlicht die Abbildung die Gemeinsamkeiten und Unterschiede der drei MQP-Messpläne.

Für diese Gemeinsamkeiten und Unterschiede in den Messplänen gibt es diverse Ursachen, die wir im Folgenden zuerst generisch und anschließend für die farblich hervorgehobenen Messung detailliert begründen. Die Gründe, warum eine Messung nicht in einem anderen MQP vorkommt, haben wir durch spezielle *Symbole* in der Abbildung gekennzeichnet. Die Messungen sind anhand dieser Symbole gruppiert.

Den Symbolen weisen wir folgende Bedeutung zu:

- **Symbol** $\triangle$: Die zur Verfügung stehenden *Diagrammtypen* wirken sich direkt auf die Messungen aus. Sind Messungen auf bestimmte Diagrammtypen wie Statecharts, Objekt- oder Sequenzdiagramme spezialisiert, muss der Entwicklungskontext als Voraussetzung für ihre Verwendung die Erstellung der entsprechenden Diagrammtypen vorsehen. Ansonsten lassen sich diese Messungen nicht übertragen. Hierzu zählen auch Messungen, die sich auf spezielle Konsistenzprobleme beziehen. Können diese Konsistenzprobleme aufgrund des gegebenen Kontextes grundsätzlich nicht auftreten, ist ihre Überprüfung nicht sinnvoll.

- **Symbol** $\diamondsuit$: Die Beurteilung der Kapselung von Attributen und Operationen ist nur dann möglich, wenn die benötigten Sichtbarkeitsinformationen in der *Entwicklungsphase* spezifiziert werden. In diesem Aspekt unterscheiden sich beispielsweise Analyse- und Entwurfsmodelle voneinander.

Der *Zweck der Modellierung* führt in den ersten beiden MQPs zu Überschneidungen bei den Qualitätsattributen Kopplung, Kohäsion, Modellierungskonvention und strukturelle sowie Verhaltens-Komplexität. Deshalb weisen diese MQPs auch einige gemeinsame Messungen auf. Allerdings werden die oben genannten Qualitätsattribute zum Teil sehr unterschiedlich gemessen. Hierfür lassen sich insbesondere folgende Argumente anführen:

- **Symbol** $\otimes$: Aufgrund der *Einschränkung auf objektive Messungen* zu Gunsten einer vollständigen Automatisierung der Qualitätsbewertung können subjektive Messungen vollständig entfallen. Die subjektiven Messungen werden

evtl. durch objektive Messungen ersetzt, die zwar das gleiche Qualitätsattribut quantifizieren, aber nicht die gleiche Nützlichkeit aufweisen.

- **Symbol** $\bigtriangledown$: Unterscheiden sich zwei MQPs im Kontextfaktor *Diagrammtyp*, kann dies dazu führen, dass Messungen weiterentwickelt werden, damit sie alle zur Verfügung stehenden Informationen eines zu prüfenden Softwaremodells ausnutzen.

- **Symbol** $\boxdot$: Insbesondere in dem Aspekt *Zweck der Modellierung* differenziert sich der MQP für Testmodelle signifikant von den beiden anderen. Der MQP für Testmodelle enthält eine ganze Reihe sehr spezieller Qualitätsattribute (z.B. Beobachtbarkeit, Kontrollierbarkeit oder Testzweck) mit assoziierten Messungen. Diese Messungen überprüfen für ein Softwaremodell, inwiefern es die Testfallgenerierung und -durchführung ermöglicht. Qualitätsattribute wie Kopplung oder Kohäsion haben dagegen keine Bedeutung, weil Testmodelle keine Spezifikation darstellen, auf dessen Basis der Quellcode implementiert oder angepasst werden soll.

- **Symbol** $\boxplus$: Die *Modellierungssprache* und die damit verknüpften Ausdrucksmöglichkeiten wirken sich auf eine Reihe von Messungen aus. Beispielsweise sehen die *Modellierungssprachen* UML/Z und AML keine Verwendung von Stereotypen wie `«control»` oder `«entity»` für Klassen vor. Deshalb wird in den entsprechenden MQPs die Ausgewogenheit der Attributverteilung für Datenklassen nicht gemessen. Insbesondere AML schränkt die Modellierungsmöglichkeiten stark ein. In AML gibt keine Vererbung oder abstrakte Klassen und der einzige unterstützte primitive Datentyp ist `String`.

- **Symbol** $\star$: Aufgrund der besonderen Zielsetzung des MQPs für Entwurfsmodelle *improvement by refactoring* entfallen alle Messungen, die auf die Ermittlung falscher, fehlender oder widersprüchlicher Modellelemente ausgerichtet sind.

- **Symbol** $\bullet$: Das Qualitätsattribut Drei-Schichten-Architektur mit den assoziierten Messungen stellt im MQP für Analysemodelle eine Besonderheit dar. Diese Forderung leitet sich nicht aus der Kontextbeschreibung, sondern aus einer Fragestellung in den Informationsbedürfnissen ab. Anstatt dieser Drei-Schichten-Architektur könnte sich ein Softwarearchitekt auch für ein anderes

Architekturmuster entscheiden. Diese Architekturentscheidung ist unabhängig von dem gegebenen Entwicklungskontext. Weil in den anderen MQPs keine Schichtenarchitektur gefordert ist, können die entsprechenden Messungen nicht übertragen werden.

- **Symbol** ○: In den ersten beiden Fallstudien liegen keine Annahmen hinsichtlich der Programmiersprache(n) oder des Codegenerators zu Grunde. Deshalb können diese MQPs beispielsweise keine Vorgaben zur Groß- und Kleinschreibung von Attributen und Operationen definieren. Für die dritte Fallstudie steht dagegen der Testfallgenerator fest. Somit können Messungen bereits einige Konventionen überprüfen, damit der Testfallgenerator bei der Verarbeitung eines Softwaremodells keine Fehlermeldungen wirft und die generierten Testfälle anschließend durchgeführt werden können.

Nach diesen allgemeinen Ausführungen zu den Unterschieden der drei Messpläne gehen wir an dieser Stelle auf einige Messungen detailliert ein. Durch die bereits oben thematisierten Überschneidungen der drei Mengen an Messungen in Abbildung 6.5 entstehen insgesamt sieben Teilmengen. Für jeweils eine ausgewählte Messung aus jeder dieser sieben Mengen erklären wir exemplarisch, warum sie nicht in einem anderen MQP enthalten ist bzw. warum sie in allen MQPs vorkommt. Diese Messungen haben wir in der Abbildung farblich hervorgehobenen:

- Die Messung *Unnamed Associations (UA2NA)* bzw. *Names or Roles for Associations (NRAssoc)* unterscheiden sich in nur zwei für diesen Vergleich unerheblichen Aspekten: Erstens ist die Messung *UA2NA* als OCL-Messung und *NRAssoc* dagegegen als XQuery-Messung definiert. Zweitens zählt *UA2NA* im Gegensatz zu *NRAssoc* nicht nur diejenigen Assoziationen, die nicht gelabelt sind, sondern bildet zusätzlich das Verhältnis zu allen gegebenen Assoziationen im Softwaremodell.

  Fehlen Beschriftungen an Assoziationen, erschließt sich einem Betrachter häufig nicht ihre Bedeutung. Deshalb kann generell für den Einsatz von Klassendiagrammen die Beschriftung von Assoziationen gefordert werden. In dem Kontextfaktor *Klassendiagramm* treten bei den Kontextbeschreibungen Überschneidungen auf (vgl. Tab. 6.23), so dass diese Messung in allen MQPs enthalten ist.

- Die Messung *Relatedness among Methods (CAMC)* quantifiziert die Kopplung von Klassen anhand der referenzierten Parameter in Methodensignaturen und ist deshalb für die Bewertung *objektorientierter Entwürfe* (vgl. Tab. 6.23) unabhängig von der Entwicklungsphase von Bedeutung. Für die Qualitätsbewertung der SUT-Spezifikation in der dritten Fallstudie ist die Kopplung von Klassen irrelevant, weil die SUT-Spezifikation nicht das komplette System modellieren muss, sondern nur die für den Test interessanten Teile.

- Die Messung *Untyped Attributes (UnAtt)* zählt die Anzahl der Attribute im Softwaremodell, deren Typ nicht deklariert ist. Die Typisierung von Attributen erhöht die Genauigkeit eines Softwaremodells. Ergänzt ein Modellierer Attributtypen, dann fügt er fehlende Information hinzu. Diese Weiterentwicklung stellt eine Evolution und keine Refaktorisierung eines Softwaremodells dar. Aus diesem Grund enthält der MQP für Entwurfsmodelle diese Messung nicht.

- Die Messung *Transitions to Operations (NT2NOps)* bezieht sich auf das Qualitätsattribut *Behavior Complexity*, das auch im Qualitätsmodell des MQPs für Analysemodelle enthalten ist. Allerdings sieht dessen Kontext den Diagrammtyp Statechart nicht vor (vgl. Tab. 6.23), so dass diese Messung unter den gegebenen Rahmenbedingungen nicht erhoben werden kann.

- Die Messung *Invocation Across Layers (IAL)* misst die Anzahl der Operationsaufrufe über Architekturschichten hinweg. Diese Messung ist nur dann sinnvoll, wenn das zu prüfende Softwaremodell eine Schichtenarchitektur beschreiben soll. In diesem Aspekt unterscheidet sich der MQP für Analysemodelle von den MQPs für Entwurfsmodelle und Testmodelle. Zudem kann diese Messung nur erhoben werden, wenn z.B. in einem Sequenzdiagramm die Operationsaufrufe modelliert sind. Ausschließlich der Kontext des MQPs für Analysemodelle sieht diesen Diagrammtypen vor (vgl. Tab. 6.23). Auch wenn die SUT-Spezifikation auf einer Schichtenarchitektur basieren und der entsprechende Diagrammtyp vorliegen würde, wäre diese Messung in der dritten Fallstudie irrelevant, weil die SUT-Spezifikation das Softwaresystem nicht vollständig modelliert und von der exakten Kommunikation abstrahieren kann.

- Die Messung *Nesting Level (MaxNLCS)* berechnet die maximale Verschachte-

lungstiefe von zusammengesetzten Zuständen. Da Statechart im Kontext des MQPs für Analysemodelle nicht vorkommen (vgl. Tab. 6.23) und die Modellierungssprache AML nur flache Statecharts erlaubt, ist diese Messung ausschließlich im MQP für Entwurfsmodelle sinnvoll.

- Die Messung *Operation Names Begin Small (OpNBS)* überprüft, ob Operationen mit einem kleinen Buchstaben beginnen. Die *Java Coding Conventions* [165] schreiben zwar für die Definition von Operationen einen kleinen Anfangsbuchstaben vor, aber in C-Programmen ist es nicht unüblich, Methoden mit einem Großbuchstaben beginnen zu lassen. Die MQPs für Analyse- und Entwurfsmodelle treffen jedoch keine Annahmen bzgl. der Programmiersprache. Folglich wird diese Konvention ausschließlich durch den MQP für Testmodelle überprüft.

Die obigen Ausführungen erklären anhand ausgewählter Beispiele, warum sich die Messpläne grundlegend unterscheiden und nur in wenigen Ausnahmen gleichen. Die Gründe für die diversen Unterschiede sind sehr vielfältig. Diese Vielfalt wird in der Abbildung 6.5 anhand der zahlreichen Kombinationen von Symbolen deutlich. Beispielsweise sieht in einigen Fällen lediglich der Entwicklungsprozess die benötigten Diagrammtypen nicht vor, so dass bereits kleinere Änderungen zur Wiederverwendung bestehender Messungen führen können. In anderen Fällen dagegen liegen grundsätzliche Unterschiede wie z.B. ein abweichender Modellierungszweck vor.

Ein Großteil der Gemeinsamkeiten und Unterschiede in den Messplänen der verschiedenen MQPs lässt sich auf die jeweiligen Gemeinsamkeiten und Unterschiede in den Kontextbeschreibungen zurückführen. Weitere Ursachen liegen in dem Automatisierungsgrad der Messungen, in den Verbesserungsstrategien oder in speziellen Architekturentscheidungen begründet. Der Vergleich der umfangreichen Messpläne und die damit verbundene Analyse der Auswirkungen einer Kontextbeschreibung belegen, dass Qualitätspläne eine hohe Spezialisierung aufweisen und der Kontext eines zu prüfenden Softwaremodells den Messplan stark beeinflusst und somit eine herausragende Bedeutung für die Qualitätsplanung hat.

Kapitel 7

Zusammenfassung, Fazit und Ausblick

Gegenstand dieser Arbeit war die Entwicklung eines systematischen Ansatzes für die Qualitätsplanung von Softwaremodellen, dessen Ergebnis ein dokumentierter Qualitätsplan ist. Den von uns entwickelten Lösungsansatz fassen wir als erstes in Abschnitt 7.1 kompakt zusammen. Wir sind davon überzeugt, mit diesem Ansatz einen gleichzeitig strukturierten und effizienten Weg definiert zu haben, um Qualitätspläne für Softwaremodelle zu entwickeln, die alle wichtigen Informationen enthalten, damit die Qualitätsplanung durchgeführt, die Prüfergebnisse durch Projektverantwortliche interpretiert und darauf basierend Entscheidungen getroffen werden können. Diesen Mehrwert unseres Ansatzes für die verschiedenen Aktivitäten in der analytischen Qualitätssicherung diskutieren wir ausführlich im Abschnitt 7.2, indem wir den Ansatz anhand unserer in Kapitel 2 formulierten Anforderungen bewerten. Weiterhin geben wir in Abschnitt 7.3 einen Ausblick auf sinnvolle konzeptionelle Erweiterungen des hier vorgestellten MQP-Ansatzes sowie methodische Anschlussarbeiten, bevor wir die Arbeit in Abschnitt 7.4 mit einem Schlusswort abschließen.

7.1 Zusammenfassung

In der modellbasierten Softwareentwicklung stellen Softwaremodelle zentrale Entwicklungsartefakte dar. Die Qualität dieser Softwaremodelle sollte analytisch überprüft werden, damit Mängel möglichst früh erkannt und behoben werden können. Bestehende Ansätze zur analytischen Qualitätssicherung von Softwaremodellen be-

schränken sich auf die Beschreibung starrer Qualitätsmodelle und Sammlungen spezieller Messungen. Jedoch ist Qualität immer relativ zu gegebenen Qualitätszielen und somit nichts Absolutes.

Um einen Qualitätsplan als Grundlage für die analytische Qualitätssicherung von Softwaremodellen auf projektspezifische Anforderungen abzustimmen, haben wir einen Ansatz für die Qualitätsplanung entwickelt, der ausgehend vom Kontext eines Softwaremodells die systematische Herleitung von Qualitätszielen und anschließende Identifizierung von Qualitätsprüfungen ermöglicht. Unser Ansatz spezialisiert die Goal Question Metric (GQM) für Softwaremodelle. Dafür haben wir die Charakterisierung des Kontextes grundlegend überarbeitet, die Identifizierung der Informationsbedürfnisse angepasst, Qualitätsmodelle vollständig integriert und die Beschreibungsschemata für Messungen ersetzt.

Abbildung 7.1 zeigt einen schematischen Überblick über unseren Lösungsansatz. Der Ansatz besteht aus einem Metamodell zur Formulierung relevanter Inhalte, einem Prozess, der als Leitfaden bei der Planung dient und einem Regelkonzept zur Sicherung und Wiederverwendung von Erfahrungswissen. Das Ergebnis der Qualitätsplanung bezeichnen wir als Modell-Qualitäts-Plan (MQP), das zugrunde liegende Metamodell als MQP-Metamodell, den Prozess als MQP-Prozess und die Regeln als MQP-Regeln.

Der MQP-Prozess besteht aus vier wesentlichen Schritten. Zuerst wird der Kontext festgestellt, um die Besonderheiten des zu prüfenden Softwaremodells und seine Einbettung in den modellbasierten Entwicklungsprozess herauszustellen und auf diese Weise zu dokumentieren, *in welchem Kontext gemessen wird*. Basierend auf der Kontextbeschreibung und unter aktiver Einbindung von Projektvertretern lassen sich im Anschluss Informationsbedürfnisse mittels Zielen und korrespondierenden Fragen erfassen. Informationsbedürfnisse zeigen an, *was wichtig zu messen ist*. Ausgehend von den Informationsbedürfnissen kann anschließend ein projektspezifisches Qualitätsmodell bestehend aus Qualitätscharakteristiken und Qualitätsattributen systematisch abgeleitet, ergänzt und um Definitionen angereichert werden. Das Qualitätsmodell beschreibt, *was gemessen werden soll*. Durch die Kombination von Informationsbedürfnissen mit einem Qualitätsmodell lassen sich relevante Qualitätsziele finden und präzise formulieren. Abschließend wird der MQP durch die Angabe von Qualitätsprüfungen vervollständigt, indem für die Qualitätsattribute Metriken und Indikatoren ausgewählt und detailliert beschrieben werden. Diese

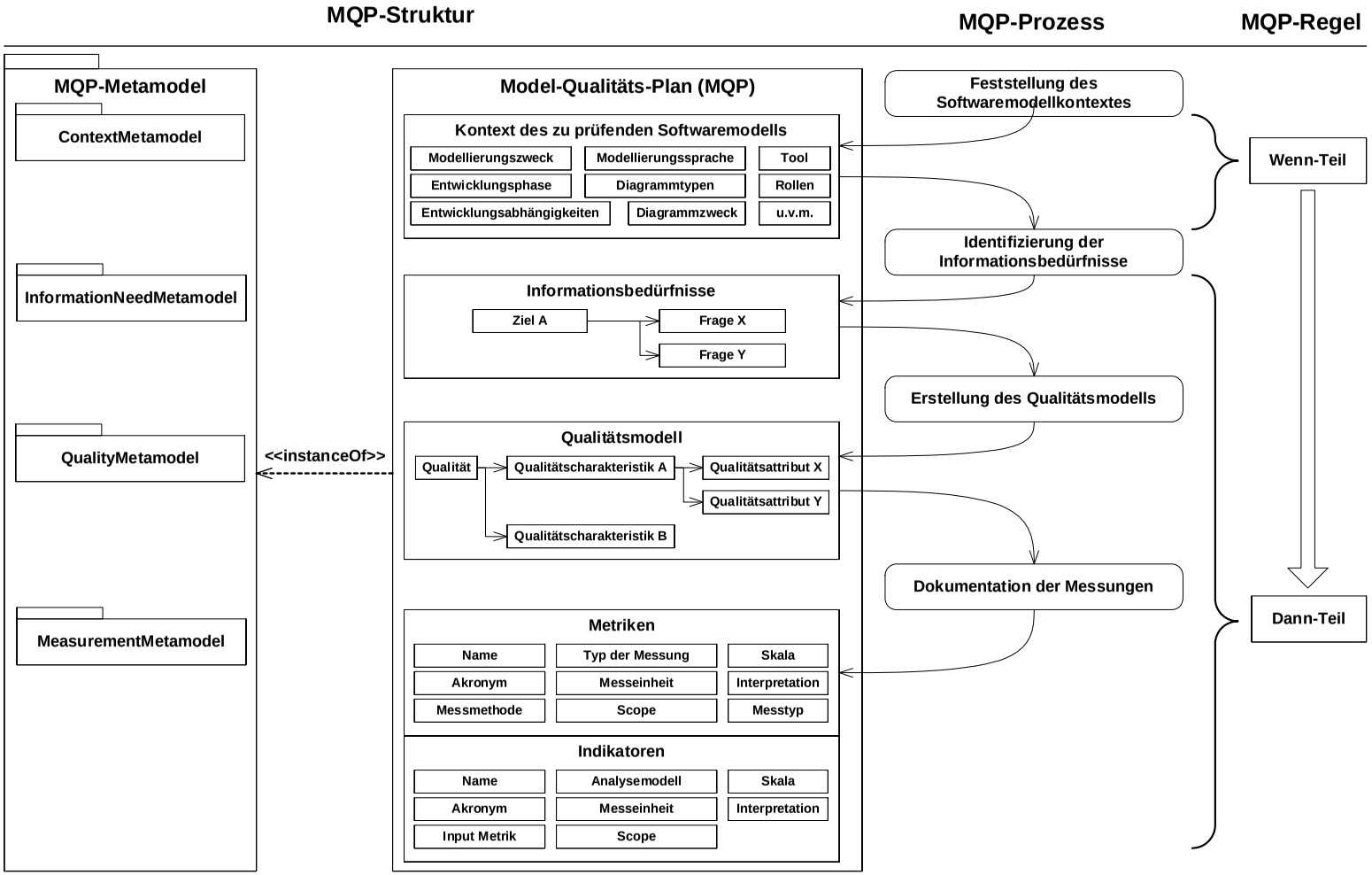


Abbildung 7.1: Qualitätsplanung mit MQP

Messungen spezifizieren, *wie gemessen wird*.

Die Ergebnisse der einzelnen Schritte sind miteinander verknüpft und bilden zusammen einen MQP als Endergebnis der Qualitätsplanung. Der Aufbau eines MQPs wird durch das MQP-Metamodell beschrieben. Das MQP-Metamodell gliedert sich entsprechend der fundamentalen MQP-Schritte in die vier Pakete ContextMetamodel, InformationNeedMetamodel, QualityMetamodel und MeasurementMetamodel. Jedes Paket spezifiziert die wichtigsten strukturellen Konzepte, die für die Formulierung der Ergebnisse des korrespondierenden Prozessschrittes benötigt werden.

Weil die Qualitätsplanung aufgrund der aktiven Einbeziehung des Projektteams und des hohen Dokumentationsaufwandes als recht kostenintensiv bewertet werden muss, haben wir für die Sicherung und Wiederverwendung von Erfahrungswissen ein einfaches Regelkonzept auf Basis von Kontextfaktoren etabliert, mit dem sich MQPs effizient erstellen lassen. Die MQP-Regeln funktionieren nach einem Wenn-Dann-Prinzip und gliedern sich dementsprechend in zwei Teile. Der Wenn-Teil legt fest, welche Kontextinformationen vorliegen müssen, damit die im Dann-Teil festgelegten Informationsbedürfnisse, Qualitäten und Messungen eingefügt werden.

Der grobe Ablauf der analytischen Qualitätssicherung auf Basis des MQP-Ansatzes ist in Abbildung 7.2 skizziert. Die folgenden Ausführungen orientieren sich an den in der Abbildung enthaltenen Nummern ⓪ bis ⑤. Im Rahmen der Qualitätssicherung soll ein *Softwaremodell* als Ergebnis der konstruktiven Tätigkeit *Modellierung* bewertet werden (siehe ⓪). Unabhängig von der Modellierungsaktivität erstellt ein Qualitätsmanager während der Qualitätsplanung durch Anwendung des MQP-Prozesses einen MQP (siehe ①). Dieser MQP bildet die Grundlage für die Durchführung der Messungen (siehe ②). Anschließend interpretieren die Projektverantwortlichen die Messergebnisse und treffen die Entscheidung, ob die Qualitätsprüfung bestanden ist und das Softwaremodell für die nächste Entwicklungsphase freigegeben werden kann oder ob das Softwaremodell durch Modellierer weiter verbessert werden muss (siehe ③). Der zu Grunde liegende MQP enthält wichtige Informationen wie formulierte Qualitätsziele und eine Charakterisierung des Kontextes, die für die Interpretation der ermittelten Messergebnisse unmittelbar von Bedeutung sind und somit einen dokumentierten Bezugsrahmen für die Interpretation der Messergebnisse darstellen (siehe ④). Das MQP-Regelkonzept kann ein Qualitätsmanager dazu nutzen, einzelne Bestandteile des von ihm entwickelten MQPs für die Verwendung in zukünftigen Qualitätsplanungen aufzubereiten (siehe ⑤).

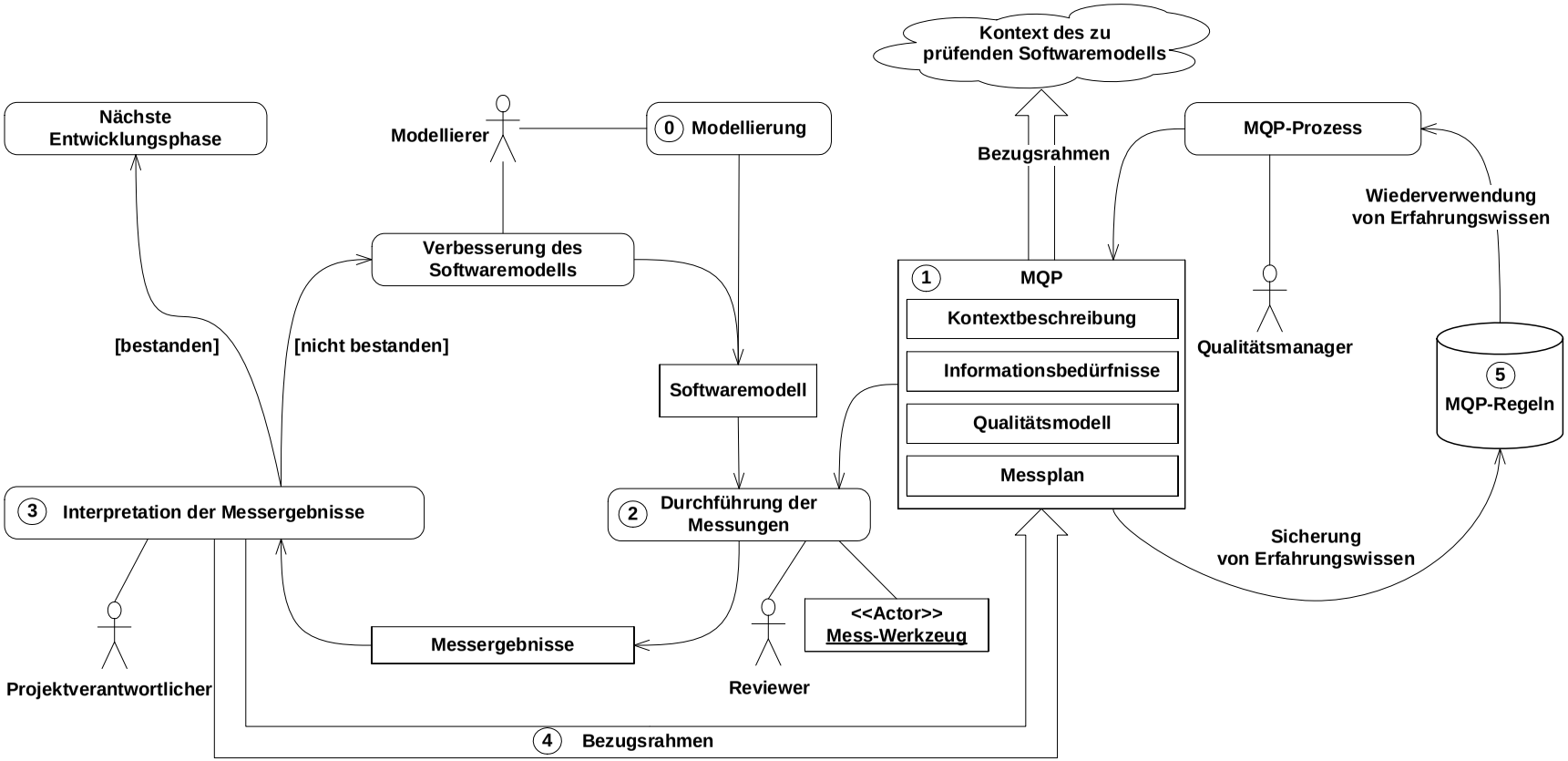


Abbildung 7.2: Grober Ablauf der analytischen Qualitätssicherung mit dem MQP-Ansatz

Die Praktikabilität des MQP-Ansatzes haben wir anhand von drei Fallstudien demonstriert, in denen jeweils ein MQP für Analysemodelle, Entwurfsmodelle und für Testmodelle entwickelt wurde. Alle drei Fallstudien zusammen spezifizieren mehr als 100 verschiedene Messungen. Ein Großteil dieser Messungen kann durch entsprechende MQP-Implementierung automatisch erhoben werden.

Der Kontext des zu bewertenden Softwaremodells ist für uns von herausragender Bedeutung. Der MQP-Ansatz sieht die Feststellung des Softwaremodellkontextes zu Beginn der Qualitätsplanung vor. Die Kontextbeschreibung fasst die wichtigsten Informationen des fachlichen Umfelds kompakt zusammen und wird für zahlreiche Zwecke wie ...

- zum Ableiten von Informationsbedürfnissen,
- zur Definition rollenspezifischer Qualitätssichten,
- zur Interpretation von Messergebnissen und
- zur Sicherung von Erfahrungswissen eingesetzt.

Den Mehrwert des MQP-Ansatzes im Allgemeinen und den Nutzen der Kontextbeschreibung im Speziellen für die verschiedenen Aktivitäten in der analytischen Qualitätssicherung diskutieren wir im folgenden Abschnitt.

7.2 Fazit

Als Fazit wollen wir den MQP-Ansatz anhand unserer in Kapitel 2 formulierten Anforderungen bewerten. Dazu füllen wir zuerst in Abschnitt 7.2.1 unsere Anforderungsmatrix aus und erläutern ausführlich die jeweiligen Ausprägungen. Dabei reflektieren wir gezielt die Relevanz der Kontextbeschreibung für die analytische Qualitätssicherung von Softwaremodellen. Anschließend ziehen wir in Abschnitt 7.2.2 einen Vergleich mit anderen Konzepten, die wir in Kapitel 4 dargestellt und bewertet haben. Abschließend stellen wir in Abschnitt 7.2.3 die Bedeutung unserer Ergebnisse für die Definition von Metriken heraus, die sich auf Softwaremodelle beziehen.

7.2.1 Bewertung

Unsere Bewertung des Konzepts *MQP* haben wir in der Tabelle 7.1 zusammengefasst. Die folgenden Ausführungen erklären die Ausprägungen für die jeweiligen Anforderungen. Wir gliedern unsere Ausführungen anhand des groben Ablaufs der analytischen Qualitätssicherung mit dem MQP-Ansatz, der in Abbildung 7.3 noch einmal dargestellt ist, und verweisen zur besseren Orientierung auf die in der Abbildung enthaltenen Nummern. Weil die Kontextbeschreibung für mehrere Zwecke genutzt wird, diskutieren wir ihren Mehrwert an verschiedenen Stellen, so dass sich die Ausführungen zum Kontext über den ganzen vorliegenden Abschnitt erstrecken.

Anforderung		**Konzept**
Nr.	Kurzbeschreibung	MQP
1	Korrekte Qualitätsziele	⊕⊕
2	Vollständige Qualitätsziele	⊕⊕
3	Unmissverständliche Qualitätsziele	⊕
4	Überprüfbare Qualitätsziele	⊕⊕
5	Rückverfolgbare Qualitätsziele	⊕
6	Vorwärtsverfolgbare Qualitätsziele	⊕
7	Nützliche Qualitätsprüfungen	⊕⊕⊖
8	Zuverlässige Qualitätsprüfungen	⊕
9	Wirtschaftliche Qualitätsprüfungen	⊕⊕
10	Interpretierbare Qualitätsprüfungen	⊕⊕⊕
11	Verständliche Qualitätsprüfungen	⊕
12	Wirtschaftliche Qualitätsplanung	⊕⊕⊖
13	Sicherung von Erfahrungswissen	⊕⊕
14	Erlernbares Vorgehen zur Qualitätsplanung	⊕⊕⊖

Tabelle 7.1: Anforderungsmatrix für das Konzept *MQP*

Die Bewertung der Qualität eines Softwaremodells ist relativ zu gegebenen Qualitätszielen. Qualitätsziele wiederum hängen vom Kontext des zu prüfenden Softwaremodells und von den Modellnutzern ab (siehe ①). Deshalb binden wir für die Identifizierung von Informationsbedürfnissen (siehe ②) zwei wichtige Quellen ein. Zum Einen nutzen wir die Kontextbeschreibung als Ausgangspunkt für das systematische Herleiten von Informationsbedürfnissen (Anf. 1 und 2 ⊕). Durch die

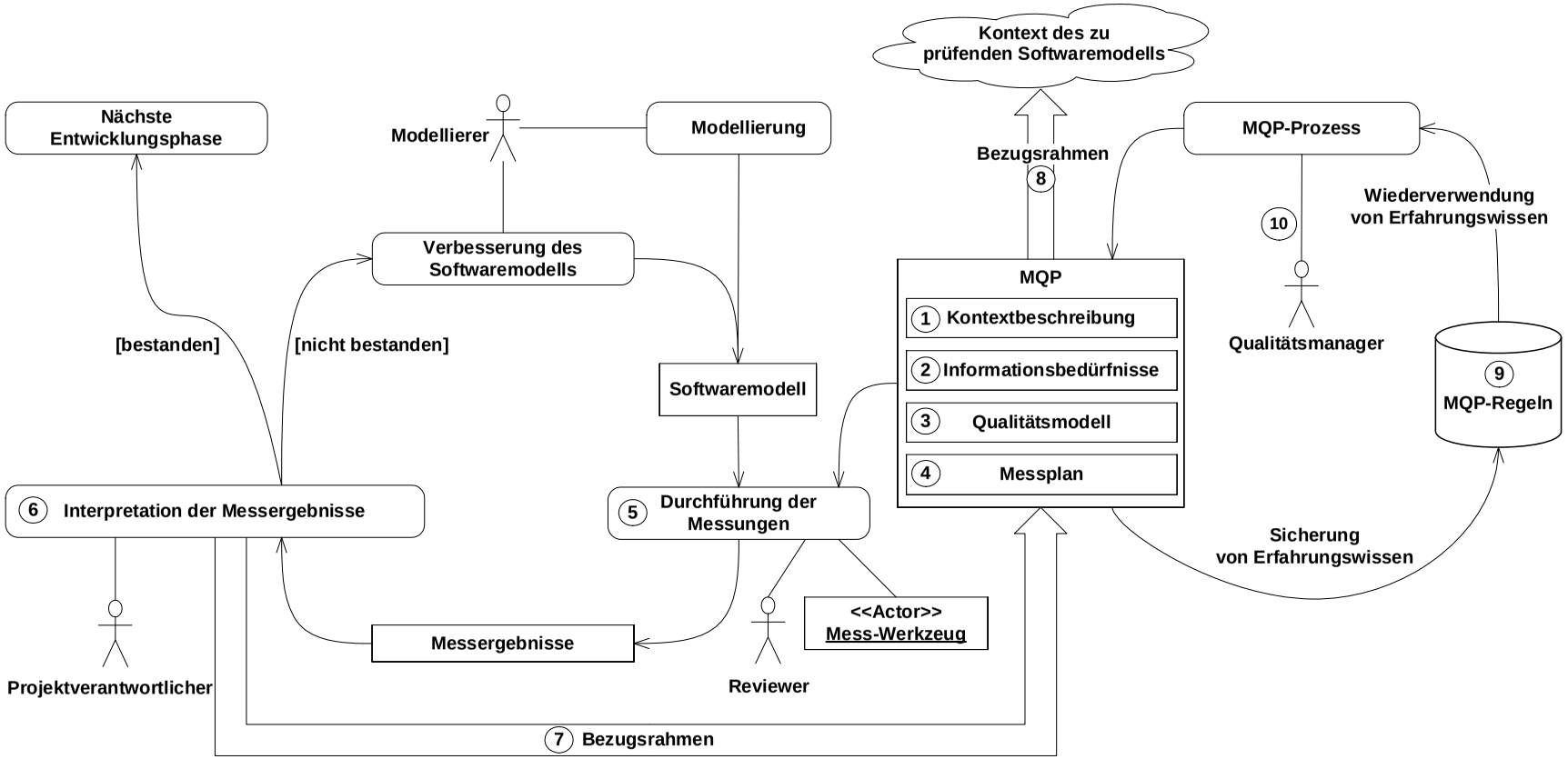


Abbildung 7.3: Grober Ablauf der analytischen Qualitätssicherung mit dem MQP-Ansatz

vorgelagerte Kontextdokumentation verringert sich der Anteil notwendiger kreativer Arbeit, so dass die Erstellung eines MQPs effizienter wird (Anf. 12 ⊕). Zum Zweiten berücksichtigen wir auch Vertreter der am Modellierungsprozess beteiligten Rollen und binden sie aktiv in die Qualitätsplanung ein. Durch die Einbindung des Projektteams lässt sich am besten gewährleisten, zumindest einen Großteil aller relevanten Qualitätsziele zu finden (Anf. 1 und 2 ⊕). Allerdings müssen die Interviews einiger Projektvertreter als sehr kostenintensiv beurteilt werden (Anf. 12 ⊖).

Auf Basis der Informationsbedürfnisse wird ein projektspezifisches Qualitätsmodell spezifiziert (siehe ③). Durch die Kombination der Informationsbedürfnisse mit einem definierten Qualitätsmodell erhöht sich der Detaillierungsgrad der Qualitätsziele (Anf. 3 ⊕). Weil das Qualitätsmodell dadurch auch beschreibt, *was gemessen werden soll*, lassen sich Qualitätsziele besser überprüfen (Anf. 4 ⊕). Hieraus darf aber nicht abgeleitet werden, dass eine stetige Verfeinerung der Qualitätsziele zwangsläufig zu einer besseren Überprüfbarkeit führt. Wenn in einem Softwaremodell notwendige Informationen schlicht fehlen, dann bleibt die Überprüfung einiger Qualitätsziele evtl. unmöglich (z.B. Beurteilung der Effizienz einer modellierten Architektur bei fehlenden Plattforminformationen).

Die Qualitätscharakteristiken des Qualitätsmodells hängen indirekt über die Ziele der Informationsbedürfnisse mit der Kontextbeschreibung zusammen. Ist der Zusammenhang zwischen einzelnen Kontextfaktoren und Zielen gekennzeichnet, lassen sich Qualitätscharakteristiken bis auf Kontextfaktoren zurückverfolgen, so dass z.B. Änderungen am Entwicklungsprozess notwendige Anpassungen am Qualitätsplan indizieren (Anf. 5 ⊕).

Die Messungen im MQP-Messplan (siehe ④) sind direkt den Qualitätsattributen des Qualitätsmodells zugeordnet, so dass sich aus dieser Verknüpfung eine Möglichkeit der Vorwärts-Verfolgbarkeit ergibt (Anf. 6 ⊕). Die Messungen schreiben vor, wie Qualitätsattribute quantifiziert und qualifiziert werden sollen und klären auf diese Weise die Durchführung der Messungen (siehe ⑤). In einem MQP-Messplan können Metriken sich aus mehreren Basiskennzahlen und/oder abgeleiteten Kennzahlen zusammensetzen. Sich mehrfach wiederholende Bestandteile von Messungen können dadurch zentral definiert und müssen während der Qualitätsprüfung nur einmal ermittelt werden (Anf. 9 ⊕). Kennzahlen werden dabei mit anderen Kennzahlen in Relation gesetzt, damit z.B. numerische Wertebereiche verglichen werden können. Die Kombination von Kennzahlen setzt die Angabe von Skalentypen vor-

aus. Beispielsweise darf eine Ordinalskala nicht mit einer Absolutskala kombiniert werden.

Wie auch in Abschnitt 4.3 unterscheiden wir zwischen subjektiven und objektiven Metriken. Beide Konzepte sind im MQP-Ansatz integriert und ermöglichen eine umfassende Überprüfung von Qualitätszielen (Anf. 4 $\oplus$). Objektive Metriken für UML-Modelle können beispielsweise durch OCL- oder XQuery-Anfragen formalisiert und automatisch erhoben werden (Anf. 8 und 9 $\oplus$). Weil subjektive Metriken der Einflussnahme des Messenden ausgesetzt sind, können wir für sie selbstverständlich keine Zuverlässigkeit garantieren. Wir wirken diesem Effekt jedoch entgegen, indem wir einige Freiheitsgrade einschränken. Beispielsweise sind die Antwortalternativen durch die Skala vorgegeben. Durch die ausführlichen Beschreibungsmöglichkeiten der Messungen fördern wir die Verständlichkeit der Qualitätsprüfungen (Anf. 11 $\oplus$) sowie die spätere Interpretation der Messergebnisse (siehe ⑥) (Anf. 10 $\oplus$).

Durch die Integration von Indikatoren in den MQP-Ansatz lässt sich durch die Angabe von Entscheidungskriterien wichtiges Erfahrungswissen für Metriken in Qualitätsplanungen nutzen (Anf. 13 $\oplus$). Ein Qualitätsmanager kann Messergebnisse in höherwertige Aussagen über Qualitätsattribute überführen und dadurch die Interpretation der Messergebnisse (siehe ⑥) erleichtern (Anf. 10 $\oplus$).

Die intensive Einbindung des Projektteams und die systematische Herleitung der Messungen ausgehend vom Kontext soll zu nützlichen Qualitätsprüfungen führen (Anf. 7 $\oplus$). Zudem haben wir den GQM-Ansatz gerade so spezialisiert, dass bestehende Arbeiten zur Modellqualität wie Qualitätsmodelle mit assoziierten Messungen oder Metriksammlungen problemlos wiederverwendet werden können (Anf. 7 $\oplus$). Allerdings können wir nicht sicherstellen, dass ausschließlich nützliche Messungen eingebaut werden (Anf. 7 $\ominus$). Dafür müssten wir für jede Messung gewährleisten, dass sie mit den entsprechenden Qualitätsattributen hinreichend stark korreliert, diese Korrelation für zumindest ähnliche Kontexte empirisch abgesichert ist und die Evaluationen zusätzlich einen eindeutigen und unmittelbaren Rückschluss auf den Erfüllungsgrad der Qualitätsattribute zulässt.

Bei der Interpretation ermittelter Messergebnisse müssen Projektverantwortliche die formulierten Qualitätsziele und den Kontext des geprüften Softwaremodells kennen, damit sie die Ursachen für Mängel erkennen und das weitere Vorgehen planen können (siehe ⑦). Die MQP-Kontextbeschreibung bietet eine kompakte Charakterisierung dieses Kontextes an und kann somit als dokumentierter Bezugsrahmen für

die Qualitätsinterpretation der Projektbeteiligten dienen (siehe ⑧) (Anf. 10 ⊕).

Das MQP-Regelkonzept (siehe ⑨) dient der Sicherung und Wiederverwendung von Erfahrungswissen. Aufgrund der Einschränkung des MQP-Ansatzes auf Softwaremodelle kann der Kontext sehr detailliert dokumentiert werden. Dadurch besitzt die Kontextbeschreibung in Kombination mit dem Regelkonzept ein sehr großes Potential. Zum einen kann ein Qualitätsmanager Erfahrungswissen in Abhängigkeit vom Kontext differenziert einordnen (Anf. 13 ⊕). Zum Zweiten lassen sich zukünftige MQPs effizienter aufbauen (Anf. 12 ⊕).

Die Erlernbarkeit des MQP-Vorgehens ist aus unserer Perspektive schwierig zu beurteilen (siehe ⑩). Der MQP-Ansatz wurde bereits im Rahmen einiger Konferenzvorträge, Workshops, einer einjährigen Projektgruppe und im Rahmen von Studien- und Diplomarbeiten verschiedenen Personen vermittelt. Dabei hat sich gezeigt, dass die grundlegende Idee für Kenner des GQM-Ansatzes sehr leicht zu verstehen ist (Anf. 14 ⊕). Allerdings bestehen gerade in den Übergängen zwischen den einzelnen Phasen gewisse Freiheitsgrade. Die Definition von Qualitätscharakteristiken und Qualitätsattributen sowie die Auswahl geeigneter Messungen erfordert unseren Beobachtungen zufolge Erfahrung und ein umfangreiches Wissen in den Bereichen *modellbasierte Entwicklung* und *analytische Qualitätssicherung*. Zudem hat sich im Rahmen der Projektgruppe gezeigt, dass Messdefinitionen in OCL den Studenten z.T. sehr schwer gefallen sind. Diese Beobachtungen weisen darauf hin, dass die Rolle des Qualitätsmanagers in der Qualitätsplanung nur von sehr versierten und erfahrenen Personen eingenommen werden kann (Anf. 14 ⊖).

Bei der Einarbeitung in die bestehende Literatur bestehen häufig Hindernisse. Beispielsweise werden in der Literatur häufig verschiedene Wörter für die gleichen Begriffe verwendet (Synonyme: z.B. Eigenschaft, Merkmal, Charakteristik) und in anderen Fällen Wörter synonym verwendet, die zwar eng miteinander zusammenhängen, sich aber auf unterschiedliche Begriffe beziehen (z.B. Metrik, Kennzahl, Messung, Analyse, Indikator, Interpretation). Hinter diesen Begriffen verbergen sich wichtige Konzepte des Qualitätsmanagements. Basierend auf Standards haben wir im Rahmen des MQP-Ansatzes eine einheitliche und definierte Terminologie geschaffen, die beim Erlernen und bei der Einordnung bestehender Literatur sehr hilfreich ist. Die Definition der MQP-Konzepte und die Modellierung der zentralen Zusammenhänge im MQP-Metamodell wirkt sich somit positiv auf die Erlernbarkeit des MQP-Ansatzes und seine Anwendung aus (Anf. 14 ⊕).

Der Kontext des zu bewertenden Softwaremodells wird im MQP-Ansatz für das Ableiten von Informationsbedürfnissen, die Interpretation von Messergebnissen, die Definition rollenspezifischer Qualitätssichten und zur Sicherung sowie Wiederverwendung von Erfahrungswissen genutzt. Diese vielfältige Verwendung der Kontextbeschreibung bestätigt die herausragende Bedeutung des MQP-Schritts *Feststellung des Kontextes* für die analytische Qualitätssicherung von Softwaremodellen. Infolgedessen realisiert der MQP-Ansatz eine *kontextsensitive Qualitätsplanung von Softwaremodellen*.

7.2.2 Vergleich mit anderen Konzepten

Goal Question Metric (GQM): Wir verwenden GQM als Basis für unsere Lösung und haben dieses Konzept durch erhebliche Erweiterungen sowie einige Reduzierungen gezielt auf die Qualitätsplanung von Softwaremodellen angepasst. Abbildung 7.4 verdeutlicht die wesentlichen Gemeinsamkeiten und Unterschiede.

Unsere Zielsetzung ist die Qualitätsbewertung von Softwaremodellen. GQM zielt auf eine übergeordnete Unternehmens- und Projektsicht ab und ist nicht auf Softwaremodelle zugeschnitten. Bedingt durch diese Ausrichtung auf ein Unternehmen bzw. Projekt unterstützt GQM die Feststellung des Kontextes von Softwaremodellen nur unzureichend. Deshalb mussten wir die in GQM vorgesehene Charakterisierung des Unternehmens- und Projektkontextes für unsere Zielsetzung grundlegend überarbeiten.

Ein weiterer wesentlicher Unterschied zwischen GQM und MQP besteht darin, dass wir die Kontextbeschreibung zur Ableitung von Informationsbedürfnissen vorsehen. Ansonsten ist in beiden Ansätzen die Identifizierung der Informationsbedürfnisse durch Einbindung des Projektteams und die anschließende Dokumentation der Gesprächsergebnisse ähnlich. Die Informationsbedürfnisse werden mittels Zielen und korrespondierenden Fragen erfasst.

Qualitätsmodelle sind eine Standardtechnik für die Dokumentation von Qualitätszielen. Wir halten ihre Wiederverwendung für sinnvoll, weil vor allem wichtiges Erfahrungswissen in ihnen gebunden ist. Aber Qualitätsmodelle sind in GQM nicht vollständig eingebunden. Durch die notwendige Integration von Qualitätsmodellen in GQM ergab sich weiterer Anpassungsbedarf.

Im Rahmen der GQM-Zieldefinition lassen sich Qualitätscharakteristiken wie Wartbarkeit zwar in Form eines Qualitätsschwerpunktes einfügen. Aber Qualitäts-

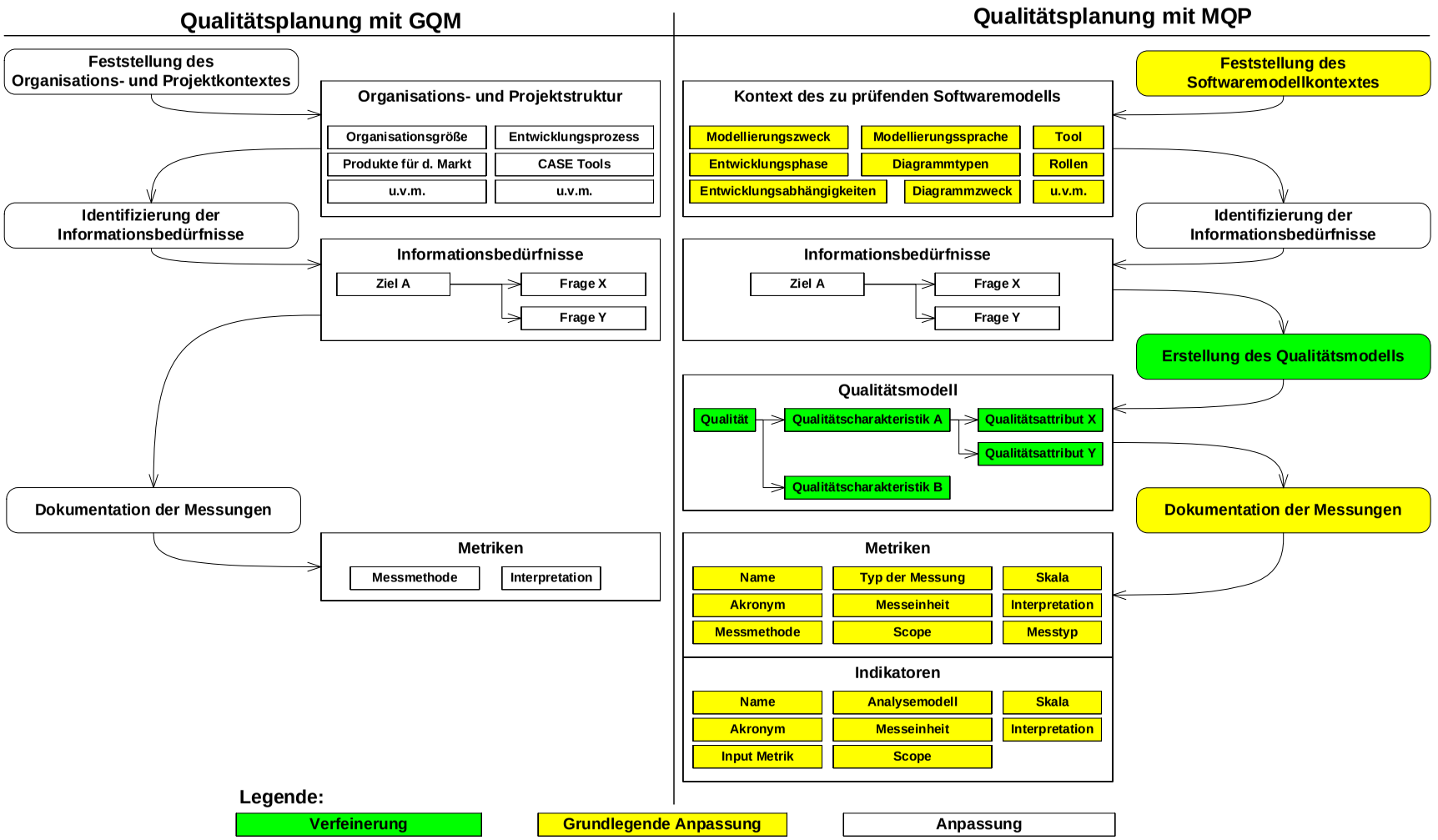


Abbildung 7.4: Vergleich zwischen den Konzepten GQM und MQP

modelle bestehen aus einer Menge definierter und verknüpfter Qualitätscharakteristiken und Qualitätsattribute, die in dieser Form in GQM keine Entsprechung haben. Zudem werden Messungen in GQM mit Fragen und nicht mit Qualitätsattributen in Verbindung gesetzt. Deshalb haben wir Qualitätsmodelle derart in den GQM-Ansatz integriert, so dass Qualitätsmodelle Ziele und Fragen präzisieren und als Bindeglied zu Messungen fungieren.

Definition von Messungen: Weder GQM noch Qualitätsmodelle definieren ein angemessenes Beschreibungsschema für Messungen, das den Besonderheiten von Softwaremodellen gerecht wird. Selbst Metriksammlungen für UML-Modelle dokumentieren Metriken häufig ausschließlich durch einen Metriknamen und eine textuelle Beschreibung. Für eine detaillierte Dokumentation von Metriken und Indikatoren synthetisieren wir erstens die Standards zu internen Softwaremetriken, zweitens Standards zum Software-Messprozess (vgl. ISO/IEC 9126-3 und ISO/IEC 15939 Standard in Kapitel 4.1.3) und drittens Beschreibungsschemata für Metriken und Indikatoren, die in diversen Veröffentlichungen genutzt werden (vgl. Kapitel 4.3). Dadurch können wir bestehende Informationen zu Metriken und Indikatoren einordnen und einheitlich erfassen. Zudem ermöglicht das im MQP-Ansatz verwendete Beschreibungsschema für Messungen eine sehr detailreiche Dokumentation. Ein Vergleich bestehender Metrik-Beschreibungsschemata mit dem im MQP-Ansatz genutzten Beschreibungsschema findet sich in Tabelle 7.2.

Die erste Spalte in der Tabelle benennt das jeweilige Beschreibungskonzept des MQP-Ansatzes. Die Qualitätsprüfungsstandards ISO/IEC 9126-3 bzw. ISO/IEC 15939 haben wir in der zweiten bzw. dritten Spalte platziert. In der vierten Spalte unter *Model Metrics* haben wir eine Reihe von Veröffentlichungen zusammengefasst, in denen konkrete Beispiele für Metriken enthalten sind, die für Softwaremodelle eingesetzt werden können.

Die Seitenzahl für Publikationen ist i.d.R. stark beschränkt. Allein aufgrund dieser Seitenrestriktionen können die Autoren ihre Metriken nicht in aller Ausführlichkeit beschreiben. Deshalb ist es kaum möglich, einen objektiven Vergleich zwischen dem MQP-Beschreibungsschema und publizierten Metriken zu ziehen. Trotzdem haben wir uns dazu entschieden, diese Menge an konkreten Beispielmetriken mit in unseren Vergleich aufzunehmen. Dadurch können wir bis auf eine Ausnahme zeigen, dass alle MQP-Beschreibungskonzepte auch für die Beschreibung bestehender

Metriken bereits genutzt wurden.

Einige Beschreibungskonzepte sind in der Literatur anders benannt (z.B. *Input to Measurement* oder *Relevant Entities* anstatt *Scope*). Derartige Differenzen haben wir nicht weiter dokumentiert, um die Tabelle möglichst übersichtlich zu halten.

In der Tabelle haben die verwendeten Symbole folgende Bedeutung:

- Mit $\surd$ kennzeichnen wir Übereinstimmungen.
- Wenn wir für ein Konzept des MQP-Ansatzes keine Entsprechung in der jeweiligen Literatur finden konnten, notieren wir $-$.
- In der Spalte *Model Metrics* verweisen wir repräsentativ auf einige Publikationen. Die Auflistung haben wir auf drei Publikationen pro Zeile begrenzt.

Approach / MQP Concept		ISO/IEC 9126-3 [87]	ISO/IEC 15939 [86]	Model Metrics [details below]
Quality Attribute		$\surd$	$\surd$	[145, 102, 4]
Name		$\surd$	$\surd$	[102, 16, 4]
Acronym		$-$	$-$	[6, 109, 4]
Measurement-method	Informal Definition	$\surd$	$\surd$	[102, 16, 4]
	Formal Definition	$-$	$-$	[28, 115, 53]
Type of Measurement		$-$	$\surd$	[121, 16, 145]
Unit of Measurement		$-$	$\surd$	[20, 27, 102]
Scope		$\surd$	$\surd$	[20, 65, 62]
Scale		$-$	$\surd$	[28, 115, 53]
Scale Type		$\surd$	$\surd$	$-$
Interpretation		$\surd$	$-$	[17, 29, 4]

Tabelle 7.2: Vergleich von Beschreibungsschemata für Metriken

Eine weitere Besonderheit für Messungen im MQP-Ansatz besteht darin, dass sich Messungen auch auf Kontextfaktoren beziehen.

Auf einen Vergleich für Indikator-Beschreibungsschema verzichten wir, da das von uns eingesetzte Beschreibungsschema für Indikatoren im Wesentlichen dem ISO/IEC 15939 Standard entspricht.

7.2.3 Bedeutung des Kontextes für die Definition von Metriken

Die bestehende Literatur dokumentiert den zugrunde liegenden Kontext von Metriken nur unzureichend. Dabei spezifiziert der Kontext die *Anwendungsbedingungen* einer Metrik. Fehlt die Kontextbeschreibung, ist nicht immer ersichtlich, unter welchen Umständen welche Metrik sinnvoll eingesetzt werden kann.

Im MQP-Ansatz bestehen Metrikbeschreibungen nicht nur aus dem im Anhang A eingesetzten Beschreibungsschema. Vielmehr sind Metriken zusätzlich direkt mit Qualitätsattributen und indirekt mit Fragen, Zielen und Kontexten verknüpft. Anhand dieser umfangreichen Informationen kann der Nutzen einer Messung für ein gegebenes Softwaremodell besser differenziert werden.

Die in diesem Abschnitt vorgenommene Analyse des MQP-Ansatzes belegt, dass der Ansatz die von uns aufgestellten Anforderungen überwiegend erfüllt. Allerdings sehen wir weiterhin einige Verbesserungspotentiale, die in Zukunft erforscht werden sollten. Einen Ausblick auf mögliche Anschlussarbeiten thematisieren wir im nächsten Abschnitt 7.3.

7.3 Ausblick

Ein Qualitätsmanager sieht sich während der analytischen Qualitätssicherung von Softwaremodellen mit erheblichen Herausforderungen konfrontiert, bei deren Bewältigung unser MQP-Ansatz ein geeignetes Instrument für die Qualitätsplanung darstellt. Jedoch ist das Gebiet der Qualitätssicherung ein weites Feld, in dem umfassende und gleichzeitig objektiv richtige Erkenntnisse nur schwer zu erlangen sind. Dementsprechend werden einige Verbesserungspotentiale durch die aktuelle Version des MQP-Ansatzes noch nicht vollständig ausgeschöpft. Darüber hinaus gibt es eine Reihe von thematisch verwandten, aber nicht behandelten Problemstellungen. Deshalb gehen wir im Folgenden auf die wichtige Anknüpfungspunkte an unsere Arbeit ein.

Nützliche Messungen: Die Auswahl nützlicher Messungen ist vielleicht die anspruchsvollste Aufgabe während der Qualitätsplanung. Durch die konstruktive Einbindung des Projektteams und dem schrittweisen, systematischen Vorgehen ausge-

hend vom Softwaremodellkontext lassen sich durchaus Messungen ableiten, die zweifelsfrei in einem kausalen Zusammenhang zu den spezifizierten Qualitätsattributen stehen (z.B. Konsistenzmetriken). Für andere Messungen ist dies aber nicht immer der Fall. Beispielsweise sind einige Messungen, die sich auf die Wartbarkeit beziehen, durchaus umstritten (vgl. Ausführungen zur Depth of Inheritance Tree (DIT) Metrik in Abschnitt 4.3.3). Dies hängt u.a. damit zusammen, dass es sehr schwierig ist, den Aufwand für Wartungsarbeiten zu bestimmen, weil die Wartung erst nach dem eigentlichen Projektabschluss stattfindet. Zudem basieren die uns bekannten empirischen Untersuchungen auf Quellcode. Inwiefern die dortigen Ergebnisse auf die Modellebene übertragen werden dürfen, muss noch weiter geklärt werden.

Im Allgemeinen fehlt bislang die statistische Bestätigung der Nützlichkeit eines Großteils der Messungen, die sich speziell auf Softwaremodelle beziehen, und die wenigen existierenden Untersuchungen haben den Kontext der betrachteten Softwaremodelle nur unzureichend charakterisiert. Deshalb sehen wir für die empirische Untersuchung der Nützlichkeit von Messungen gerade im Hinblick auf unterschiedliche Kontexte erheblichen Handlungsbedarf, denn ein Qualitätsmanager ist auf solche Ergebnisse angewiesen. Hier wäre eine konzeptionelle Erweiterung des MQP-Ansatzes zur Speicherung von gemessenen Ergebnissen für explizit dokumentierte Kontexte hilfreich, über die nach ausreichender Datensammlung die Nützlichkeit von Messungen geprüft und ggf. geeignete Entscheidungskriterien abgeleitet werden können.

Indirekte Messungen: Die Qualitätsprüfungen im MQP-Ansatz beruhen auf Messungen, die *direkt* auf dem Softwaremodell bzw. auf seiner serialisierten Darstellung arbeiten. Direkte Messungen sind robust gegenüber Unvollständigkeiten im Softwaremodell, weil die dem Softwaremodell zu Grunde liegende Formalisierung vernachlässigt wird.

Um die Semantik eines Softwaremodells stärker berücksichtigen zu können, benötigen wir *indirekte* Messungen. Indirekte Messungen können erst durchgeführt werden, nachdem das Softwaremodell in eine semantische Domäne transformiert und das Transformationsergebnis analysiert wurde. Indirekte Messungen arbeiten dann auf den Analyseergebnissen.

Dementsprechend existieren für die Qualitätsprüfung von Softwaremodellen, die auf eine formale Semantik zurückgeführt werden können, weitere Verfahren, die bis

jetzt noch nicht in den MQP-Ansatz integriert sind. Hier sind beispielsweise Verfahren für die Konsistenzprüfung von Statecharts [100] und Aktivitätendiagrammen [172] oder auch die Prüfung von speziellen Qualitätsattributen wie Soundness für Aktivitätendiagramme [158] zu nennen. Diese Qualitätsprüfungen setzen eine Transformation des Softwaremodells in eine semantische Domäne (z.B. CSP [148]) voraus, um mit Hilfe von Model Checkern (z.B. FDR2) Anforderungen an das Softwaremodell zu analysieren.

Für Prognosen der Performanz oder Zuverlässigkeit eines Komponentenentwurfs benötigt man ebenfalls deutlich aufwändigere Messmethoden. Im Rahmen des Palladio Projekts [15] werden Komponentenentwürfe auf Basis des Palladio Component Models zuerst in ein stochastisches Modell, ein Netzwerkmodell oder einen Prototypen transformiert, um anschließend auf dieser Grundlage die modellierte Architektur zu analysieren und Prognosen über das spätere Verhalten einer Implementierung unter realen Systembedingungen zu treffen.

Qualitätsdarlegung: Nach einer abgeschlossenen Durchführung der Qualitätsprüfung werden die Prüfergebnisse an die Projektverantwortlichen zur Interpretation übergeben. Dafür müssen die Prüfergebnisse aufbereitet und idealerweise in maßgeschneiderten Qualitätsberichten dargelegt werden. Im Prinzip enthält ein MQP alle wichtigen Informationen. Diese müssen aber auf jede Rolle hin angepasst und angemessen präsentiert werden. In den MQP-Implementierungen wurden für die Visualisierung unterschiedliche Konzepte verfolgt [25, 50]. Weitere Überlegungen zur Visualisierung von Messergebnissen finden sich z.B. in [105]. Wir haben die Darlegung von Prüfergebnissen im Rahmen dieser Arbeit nicht thematisiert und sehen für diesen Bereich noch weiteren Handlungsbedarf.

Kosten-Nutzen Evaluation: Für den Einsatz in der Praxis sollte der MQP-Ansatz aufgrund der Verwandtschaft zu GQM grundsätzlich geeignet sein. Erste Evaluationen hinsichtlich der Praktikabilität des MQP-Ansatzes haben wir durch unsere in Kapitel 6 beschriebenen Fallstudien vorgenommen. Allerdings verursacht die analytische Qualitätssicherung von Softwaremodellen auf Basis des MQP-Ansatzes Kosten. Aufgrund des hohen Detaillierungsgrades ist insbesondere der initiale Erstellungsaufwand für einen MQP erheblich.

Zwar steckt in der Einbeziehung des Kontextes viel Potential. Die Wiederverwen-

dung eines MQPs als Ganzem oder einzelner MQP-Teile kann an der Dokumentation des Kontextes erkannt werden. Nachdem erst einmal eine Organisation eine Wissensbasis mit einer Vielzahl an MQP-Regeln gefüllt hat, kann auf dieser Grundlage der Aufwand für die Qualitätsplanung erheblich reduziert werden. Eine wesentliche Stärke des Ansatzes kann folglich darin bestehen, dass langfristig eine Wissensbasis aufgebaut wird, in der MQP-Bausteine differenziert nach Kontexten beschrieben sind. Wenn in folgenden Qualitätsplanungen lediglich der Kontext definiert werden muss und die restlichen Bestandteile des MQPs automatisch generiert werden können, dann beschränkt sich der Aufwand im Wesentlichen auf einen abschließenden Review des MQPs mit einigen Projektvertretern, in dem die Vollständigkeit des MQPs zu prüfen ist, und ggf. vereinzelte Anpassungen vorgenommen werden müssen.

Allerdings sind weitere Evaluationen notwendig, um auch diese Kosten-Nutzen Relation beurteilen zu können. Dafür sind wir auf eine ganze Reihe von unterschiedlichen, modellbasierten Entwicklungsprojekten angewiesen und müssten außerhalb des universitären Bereichs den MQP-Ansatz mit Unterstützung von Unternehmensberatungen oder Softwareherstellern produktiv einsetzen. Erst eine Vielzahl an Projekten kann gesicherte Erkenntnisse über die Kosteneffektivität des MQP-Ansatzes liefern.

Gewichtete Qualitätsziele: Die Forderung nach einer Gewichtung von Qualitätszielen haben wir im Rahmen dieser Arbeit ausgeklammert. Allerdings existieren eine Reihe von Qualitätszielen, die im Konflikt zueinander stehen, so dass nicht alle Qualitätsziele erfüllt werden können. In solchen Situationen helfen Gewichte dabei, die richtigen Prioritäten für die Modellierung festzulegen und beispielsweise Entwurfsentscheidungen differenzierter zu bewerten.

Output-Dokument der Interpretation: Die Betrachtung von Output-Dokumenten der Qualitätssicherungsphase *Interpretation der Prüfergebnisse* haben wir vernachlässigt. Dabei sollten die Ergebnisse und Entscheidungen der Interpretationsphase für die nachfolgenden Tätigkeiten in einer strukturierten Form vorliegen, damit z.B. den Modellierern klar ist, warum ein Softwaremodell die Qualitätsprüfung nicht bestanden hat und was verbessert werden soll.

Verschmelzung von Produkt- und Prozessqualität: Wir haben den MQP-Ansatz auf die Bewertung der Produktqualität ausgerichtet. Doch der Übergang

zwischen Produkt- und Prozessqualität ist fließend. Sieht beispielsweise der Entwicklungsprozess die Verwendung einer ungeeigneten Modellierungssprache vor, kann dieser Mangel mit Hilfe des MQP-Ansatzes nicht festgestellt werden. Die Wahl der Modellierungssprache wird nicht analysiert. Die Modellierungssprache bildet aber das Fundament für die Modellierung und beeinflusst somit indirekt die Produktqualität. Bei einer ganzheitlichen Betrachtung von Prozess- und Produktqualität ergeben sich zudem Synergieeffekte, indem Produkt- und Prozesskennzahlen in Beziehung gesetzt werden (z.B. Berechnung der Modellierungseffizienz als Verhältnis zwischen dem Modellierungsergebnis (Produkt) und dem Modellierungsaufwand (Prozess)).

7.4 Abschluss

> Now, things do not, in general, run around with their measures stamped on them like the capacity of a freight-car: it requires a certain amount of investigation to discover what their measures are.
>
> Norbert Wiener, 1921 - [179]

Spezielle qualitätssichernde Maßnahmen für die Ebene der Softwaremodelle sind wichtig für den Erfolg der modellbasierten Softwareentwicklung. Allerdings kann Modellqualität nicht durch einen fixen Satz an Messungen bestimmt werden, weil Messungen die Erreichung von Qualitätszielen prüfen sollen und Qualitätsziele wiederum in Abhängigkeit vom gegebenen Kontext variieren.

In dieser Arbeit haben wir einen *kontextsensitiven* Qualitätsplanungsansatz für die systematische Auswahl der *richtigen* Messungen eingeführt. Durch die Kombination aus Prozess, Metamodell und Regeln haben wir gleichzeitig ein *Rahmenwerk* für die Domäne der analytischen Qualitätssicherung aufgebaut. Dieses Rahmenwerk weiterhin anzuwenden, zu pflegen, zu erweitern, immer wieder zu prüfen, um es als festen Bestandteil der modellbasierten Softwareentwicklung zu integrieren, das sind die zukünftigen Herausforderungen eines effizienten Qualitätsmanagements.

Abbildungsverzeichnis

Tabellenverzeichnis

Literaturverzeichnis

[1] A. Anacleto, T. Punter, and C. Gresse von Wangenheim. GQM-Handbook and Overview of GQM-plans. Technical Report IESE-Report, 008.03/E, Fraunhofer Institut for Experimental Software Engineering, Kaiserslautern, Germany, 2003.

[2] K. Baclawski, M. Kokar, P. Kogut, L. Hart, J. Smith, W. Holmes, J. Letkowski, and M. Aronson. Extending UML to Support Ontology Engineering for the Semantic Web. In *Proceedings of the 4th International Conference on The Unified Modeling Language: Modeling Languages, Concepts, and Tools (UML'01)*, pages 342–360. Springer, 2001.

[3] K. Baclawski, M. Kokar, P. Kogut, L. Hart, J. Smith, J.Letkowski, and P. Emery. Extending the Unified Modeling Language for ontology development. *Software and System Modeling*, 1(2):142–156, 2002.

[4] J. Bansiya and C. G. Davis. A hierarchical model for object-oriented design quality assessment. *IEEE Transactions on Software Engineering*, 28(1):4–17, 2002.

[5] A. L. Baroni, S. Braz, and F. Brito e Abreu. Using OCL to Formalize Object-Oriented Design Metrics Definitions. In *Proceedings of the 6th International ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering (QUAOOSE'2002)*, 2002.

[6] A. L. Baroni and F. Brito e Abreu. An OCL-Based Formalization of the MOOSE Metric Suite. In *Proceedings of the 7th International ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering (QUAOOSE'2003)*, 2003.

[7] V. Basili, G. Caldiera, and D. Rombach. The Goal Question Metric Approach. *Encyclopedia of Software Engineering*, pages 528–532, 1994.

[8] V. Basili, J. Heidrich, M. Lindvall, J. Münch, M. Regardie, and A. Trendowicz. GQM$^+$ Strategies - Aligning Business Strategies with Software Measurement. In *Proceedings of the 1st International Symposium on Empirical Software Engineering and Measurement (ESEM'07)*, pages 488–490. IEEE Computer Society, 2007.

[9] V. Basili and M. Oivo. Representing Software Engineering Models: The TAME Goal-Oriented Approach. Technical Report UMIACS-RE-91-155, CS-TR-2798, Department of Computer Science, University of Maryland, October 1991.

[10] V. Basili and D. Rombach. TAME: Integrating Measurement into Software Environments. Technical Report CS-TR-1764, TAME-TR-1-1987, Department of Computer Science, University of Maryland, June 1987.

[11] V. Basili and D. Rombach. The tame project: Towards improvement-oriented software environments. *IEEE Transaction on Software Engineering*, 14(6):758–773, 1988.

[12] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. Addison-Wesley, 2nd edition, 2007.

[13] C. Batini, S. Ceri, and S. Navathe. *Conceptual Database Design: An Entity-Relationship Approach*. Benjamin/Cummings, 1992.

[14] J. Becker, R. Schütte, T. Geib, and H. Ibershoff. Grundsätze ordnungsmäßiger Modellierung (GoM). Technical Report 01 IS 604, Westfälische Wilhelms-Universität Münster, 2000.

[15] S. Becker, H. Koziolek, and R. Reussner. The Palladio component model for model-driven performance prediction. *Journal of Systems and Software (JSS)*, 82(1):3–22, 2009.

[16] B. Berenbach. The Evaluation of Large, Complex UML Analysis and Design Model. In *Proceedings of the 26th International Conference on Software Engineering (ICSE'04)*, pages 232–241. IEEE Computer Society, 2004.

[17] J. Bieman and B.-K. Kang. Cohesion and reuse in an object-oriented system. *Proceedings of the ACM SIGSOFT Symposium on Software Reusability (SSR'95)*, 20:259–262, 1995.

[18] R. Binder. *Testing Object-Oriented Systems - Models, Patterns, and Tools*. Addison-Wesley, 1999.

[19] A. Birk, R. van Solingen, and J. Järvinen. Business Impact, Benefit, and Cost of Applying GQM in Industry: An In-Depth, Long-Term Investigation at Schlumberger RPS. In *Proceedings of the 5th IEEE International Software Metrics Symposium (METRICS'98)*, pages 93–96. IEEE Computer Society, 1998.

[20] M. G. Bocco, M. Piattini, and C. Calero. A Survey of Metrics for UML Class Diagrams. *Journal of Object Technology*, 4(9):59–92, 2005.

[21] B. Boehm, J. R. Brown, and M. Lipow. Quantitative Evaluation of Software Quality. In *Proceedings of the 2nd International Conference on Software Engineering (ICSE'76)*, pages 592–605, 1976.

[22] G. Booch. *Objektorientierte Analyse und Design*. Addison-Wesley, 1994.

[23] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language: User Guide*. Addison-Wesley, 1999.

[24] T. P. Bowen, G. B. Wigle, and J. T. Tsai. Specification of Software Quality Attributes. Volume 2. Software Quality Specification Guidebook. Technical Report ADA153989, Boeing Aerospace Co Seattle WA, 1985.

[25] R. Braun, B. Dietrich, S. Dohle, T. Friedrich, A. Kemena, M. Kleine, C. Oberhokamp, M. Piepmeyer, M. Raacke, Y. Tan, and P. Winkelhane. RMC - Refaso und Model Cockpit. Projektgruppe Abschlussbericht, Universität Paderborn, Fachgruppen Spezifikation und Modellierung von Softwaresystemen und Datenbank- und Informationssysteme, 2008.

[26] L. Briand, C. Differding, and D. Rombach. Practical Guidelines for Measurement-Based Process Improvement. *Special issue of International Journal of Software Engineering & Knowledge Engineering*, 2, 1997.

[27] L. Briand, S. Morasca, and V. Basili. Defining and Validating Measures for Object-Based High-Level Design. *IEEE Transactions on Software Engineering*, 25:722 – 743, 1999.

[28] F. Brito e Abreu. Using OCL to formalize object oriented metrics definitions. Technical Report ES007/2001, FCT/UNL and INESC, Portugal, June 2001.

[29] F. Brito e Abreu, R. Esteves, and M. Goulão. The Design of Eiffel Programs: Quantitative Evaluation Using the MOOD Metrics. In *Proceedings of Technology of Object Oriented Languages and Systems (TOOLS'96)*, August 1996.

[30] F. Brito e Abreu and W. Melo. Evaluating the Impact of Object-Oriented Design on Software Quality. In *Proceedings of the 3rd International Software Metrics Symposium*, pages 90–99, 1996.

[31] Brockhaus. *Brockhaus Enzyklopädie in 30 Bänden.* F.A. Brockhaus GmbH, Leipzig, Bibliografisches Institut und F.A. Brockhaus AG, Mannheim, 21st edition, 2006.

[32] M. Broy, M. Jarke, M. Nagl, and D. Rombach. Manifest: Strategische Bedeutung des Software Engineering in Deutschland. *Informatik Spektrum*, 29(3):210–221, Mai 2006.

[33] H.-J. Bungartz, M. Buchholz, and D. Pflüger. *Modellbildung und Simulation.* Springer, 2009.

[34] J. Burch, E. Clarke, K. McMillan, D. Dill, and L. Hwang. Symbolic model checking: 10^{20} states and beyond. In *Proceedings of the Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 1–33. IEEE Computer Society Press.

[35] D. Card and R. Glass. *Measuring software design quality.* Prentice-Hall, Inc., 1990.

[36] P. Chen and H.-D. Knöll. *Der Entity-Relationship-Ansatz zum logischen Systementwurf - Datenbank- und Programmentwurf.* BI Wissenschaftsverlag, 1991.

[37] B. Cheng, R. Stephenson, and B. Berenbach. Lessons Learned from Automated Analysis of Industrial UML Class Models (An Experience Report).

In *Proceedings of the 8th International Conference Model Driven Engineering Languages and Systems (MoDELS'05)*, volume 3713 of *Lecture Notes in Computer Science*, pages 324–338. Springer, 2005.

[38] S. Chidamber and C. Kemerer. A Metrics Suite for Object Oriented Design. *IEEE Transactions on Software Engineering*, 20(6):476–493, 1994.

[39] E. Clarke, O. Grumberg, and D. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, 1994.

[40] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. The MIT Press, 2000.

[41] R. Cooper. *Winning at New Products - Accelerating the Process from Idea to Launch*. Perseus Books Group, 3rd edition, 2001.

[42] J. Cruz-Lemus, M. Genero, M. Piattini, and J. Álvarez. An Empirical Study of the Nesting Level of Composite States Within UML Statechart Diagrams. In *Proceedings of the Perspectives in Conceptual Modeling, ER 2005 Workshops AOIS, BP-UML, CoMoGIS, eCOMO, and QoIS*, volume 3770 of *Lecture Notes in Computer Science*, pages 12–22. Springer, 2005.

[43] J. Daly, A. Brooks, J. Miller, M. Roper, and M. Wood. An Empirical Study Evaluating Depth of Inheritance on the Maintainability of Object-Oriented Software. *Empirical Software Engineering, An international Journal*, 1:109–132, 1996.

[44] W. Deming. *Out of the Crisis*. Cambridge University Press, 1986.

[45] B. Dietrich. Regeln für die kontextsensitive Qualitätsplanung von Software-Modellen. Universität Paderborn, Studienarbeit, 2008.

[46] S. Dißmann and V. Zurwehn. Vorschlag für ein sichtenorientiertes Qualitätsmodell. Technical report, Fachbereich Informatik, Universität Dortmund, 1986.

[47] DIN. Deutsche Industrie Norm (DIN) 66272: 1994-10 - Bewerten von Softwareprodukten: Qualitätsmerkmale und Leitfaden zu ihrer Verwendung, 1994.

[48] DIN. DIN EN ISO 9001:2000: Qualitätsmanagementsysteme - Anforderungen, 2000.

[49] D. Djuric, D. Gasevic, and V. Devedzic. Ontology Modeling and MDA. *Journal of Object Technology*, 4(1):109–128, 2005.

[50] S. Dohle. Bewertung der Qualität von AGEDIS-Testmodellen durch Anwendung und Implementierung des MQP-Ansatzes. Master's thesis, Universität Paderborn, 2009.

[51] C. Ebert, R. Dumke, M. Bundschuh, and A. Schmietendorf. *Best Practices in Software Measurement*. Springer, 2005.

[52] M. El-Wakil, A. El-Bastawisi, M. Boshra, and A. Fahmy. Object-Oriented Design Quality Models - A Survey and Comparison. In *Proceedings of the 2nd International Conference on Informatics and Systems (INFOS'04)*, 2004.

[53] M. M. El-Wakil, A. El-Bastawisi, M. B. Riad, and A. A. Fahmy. A novel approach to formalize object-oriented design metrics. Technical report, Information Systems Department, Faculty of Computers and Information, Cairo University - Cairo, Egypt, 2005.

[54] T. M. Fehlmann. *Six Sigma in der SW-Entwicklung*. Vieweg+Teubner Verlag, 2005.

[55] A. Felfernig, G. Friedrich, D. Jannach, M. Stumptner, and M. Zanker. Transforming UML domain descriptions into Configuration Knowledge Bases for the Semantic Web. In *Proceedings of the Workshop on Knowledge Transformation for the Semantic Web (KTSW'02)*, pages 11–18, 2002.

[56] N. Fenton and S. L. Pfleeger. *Software Metrics: A Rigorous & Practical Approach*. Thomson Computer Press, 2 edition, 1996.

[57] F. Fieber, M. Huhn, and B. Rumpe. Modellqualität als Indikator für Softwarequalität: eine Taxonomie. *Informatik-Spektrum*, 31(5):408–424, October 2008.

[58] C. Fischer. *Combination and Implementation of Processes and Data: from CSP-OZ to Java*. PhD thesis, University of Oldenburg, 2000.

[59] M. Fowler. *Refactoring: Improving The Design Of Existing Code*. Addison-Wesley Professional, June 1999.

[60] T. Gardner and L. Yusuf. Explore model-driven development (MDD) and related approaches: A closer look at model-driven development and other industry initiatives. *IBM developerWorks*, 2006. `http://www.ibm.com/developerworks/library/ar-mdd3/` [online; accessed 09-07-2009].

[61] M. Genero, D. Miranda, and M. Piattini. Defining and Validating Metrics for UML Statechart Diagrams. In *Proceedings of the 6th International ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering (QAOOSE'02)*, pages 120–136, 2002.

[62] M. Genero, D. Miranda, and M. Piattini. Defining Metrics for UML Statechart Diagrams in a Methodological Way. In *Proceedings of Conceptual Modeling for Novel Application Domains, ER 2003 Workshops ECOMO, IWCMQ, AOIS, and XSDM*, volume 2814 of *Lecture Notes in Computer Science*, pages 118–128. Springer, 2003.

[63] M. Genero and M. Piattini. Empirical validation of measures for class diagram structural complexity through controlled experiments. In *Proceedings of the 5th International ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering (QAOOSE'01)*, 2001.

[64] M. Genero, M. Piattini, and C. Calero. Empirical validation of class diagram metrics. In *Proceedings of the International Symposium on Empirical Software Engineering*, pages 195–203, 2002.

[65] M. Genero, M. Piattini-Velthuis, J.-A. Cruz-Lemus, and L. Reynoso. Metrics for UML Models. *UPGRADE - The European Journal for the Informatics Professional*, 5:43–48, 2004.

[66] T. Gilb and D. Graham. *Software Inspection*. Addison-Wesley, 1994.

[67] P. Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems*. Springer, 1996.

[68] A. Gray and S. MacDonell. GQM++ A Full Life Cycle Framework for the Development and Implementation of Software Metric Programs. In *Proceedings of the Australian Software Measurement Conference, Implementation of Software Metric Programs*, pages 22–35, 1997.

[69] C. Gresse, B. Hoisl, and J. Wuest. STTI-Report: A Process Model for GQM-Based Measurement. Technical Report STTI- 95-04-E, University of Kaiserslautern, Department of Computer Science, 1995.

[70] C. Gresse von Wangenheim. *Operationalizing the Reuse of Software Measurement Planning Knowledge*. PhD thesis, University of Kaiserslautern, 2002.

[71] C. Gresse von Wangenheim, B. Hoisl, H. Rombach, and G. Ruhe. *Evaluation und Evaluationsforschung in der Wirtschaftsinformatik. Handbuch für Praxis, Lehre und Forschung*, chapter Zielorientiertes Messen und Bewerten zur Software-Qualitätsverbesserung. Eine Kosten/Nutzen-Analyse, pages 253–266. Oldenbourg, 2000.

[72] C. Gresse von Wangenheim, T. Punter, and A. Anacleto. Software Measurement for Small and Medium Enterprises - A Brazilian-German view on extending the GQM method. In *Proceedings of Empirical Assessment in Software Engineering (EASE'03)*, pages 139–156, 2003.

[73] D. Harel and M. Politi. *Modeling Reactive Systems With Statecharts: A Statemate Approach*. McGraw-Hill, 1998.

[74] D. Harel and B. Rumpe. Meaningful Modeling: What's the Semantics of 'Semantics'? *IEEE Computer*, 37(10):64–72, 2004.

[75] R. Harrison, S. Counsell, and R. Nithi. An Evaluation of the MOOD Set of Object-Oriented Software Metrics. *IEEE Transactions on Software Engineering*, 24(6):491–496, 1998.

[76] R. Harrison, S. Counsell, and R. Nithi. Experimental assessment of the effect of inheritance on the maintainability of object-oriented systems. *Journal of Systems and Software*, 52(2-3):173–179, 2000.

[77] A. Hartman. AGEDIS (Automated Generation and Execution of Test Suites for DIstributed Component-based Software) final project report. Technical Report AGEDIS 1999-20218, 2004.

[78] J. H. Hausmann. *Dynamic Meta Modeling - A Semantics Description Technique for Visual Modeling Languages*. PhD thesis, University of Paderborn, 2005.

[79] M. Hitz and B. Montazeri. Measuring product attributes of object-oriented systems. In *Proceedings of 5th European Software Engineering Conference (ESEC '95)*, pages 124–136. Springer, 1995.

[80] D. Hoffman. The Darker Side of Metrics. In *Proceedings of 8th Pacific Northwest Software Quality Conference (PNSQC '00)*, 2000.

[81] D. Hoffmann. *Software-Qualität*. Springer eXamen.press, 2008.

[82] IEEE. IEEE Std 610-1990: Institute of Electrical and Electronics Engineers (IEEE) Standard Computer Dictionary: A Compliation of IEEE Standard Computer Glossaries, 1990.

[83] IEEE. IEEE Std 1028-1997: Institute of Electrical and Electronics Engineers (IEEE) Standard for Software Reviews, 1997.

[84] IEEE. IEEE Std 830-1998: Institute of Electrical and Electronics Engineers (IEEE) Recommended Practice for Software Requirements Specifications, 1998.

[85] ISO/IEC. International Organization for Standardization (ISO) and International Electrotechnical Commision (IEC) 9126:2001: Software engineering - Product quality - Part 1: Quality model, 2001.

[86] ISO/IEC. International Organization for Standardization (ISO) and International Electrotechnical Commision (IEC) 15939:2002: Software engineering - Software measurement process, 2002.

[87] ISO/IEC. International Organization for Standardization (ISO) and International Electrotechnical Commision (IEC) 9126:2001: Software engineering - Product quality - Part 3: Internal metrics, 2003.

[88] ISO/IEC. International Organization for Standardization (ISO) and International Electrotechnical Commision (IEC) 15504:2004: Software Process Improvement Capability dEtermination (SPICE), 2004.

[89] ISO/IEC. International Organization for Standardization (ISO) and International Electrotechnical Commision (IEC) 25000:2005: Software engineering - Software product Quality Requirements and Evaluation (SQuaRE) - Guide to SQuaRE, 2005.

[90] M. Jeckle, C. Rupp, J. Hahn, B. Zengler, and S. Queins. *UML2 glasklar*. Hanser, 2004.

[91] R. Jeffery and V. Basili. Validating the TAME Resource Data Model. In *Proceedings of the 10th International Conference on Software Engineering (ICSE'88)*, pages 187–201. IEEE Computer Society, 1988.

[92] S. Katz and D. Peled. An efficient verification method for parallel and distributed programs. In *Workshop on Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency*, number 354 in LNCS, pages 489–507. Springer-Verlag, 1988.

[93] B. Kitchenham, S. Linkman, A. Pasquini, and V. Nanni. The SQUID approach to defining a quality model. *Software Quality Journal*, 6:211–233, 1997.

[94] A. Kleppe, J. Warmer, and W. Bast. *MDA Explained: The Model Driven Architecture: Practice and Promise*. Addison-Wesley, 2003.

[95] Koordinierungs- und Beratungsstelle der Bundesregierung für Informationstechnik in der Bundesverwaltung (KBSt). V-Modell XT. `http://www.kbst.bund.de/nn_836802/Content/Standards/V__Modell_ _xt/v__modell__xt__node.html__nnn=true`, 2004.

[96] J. Krogstie. Integrating the understanding of quality in requirements specification and conceptual modeling. *ACM SIGSOFT Software Engineering Notes*, 23(1):86–91, 1998.

[97] J. Krogstie, O. I. Lindland, and G. Sindre. Defining quality aspects for conceptual models. In E. D. Falkenberg and W. Hesse, editors, *Proceedings of the IFIP international working conference on information system concepts (ISCO'95)*, volume 26 of *IFIP Conference Proceedings*, pages 216–231. Chapman & Hall, 1995.

[98] C. Kroiß and N. Koch. UWE Metamodel and Profile: User Guide and Reference. Technical Report 0802, Ludwig-Maximilians-Universität München (LMU), 2008.

[99] P. Kroll and P. Kruchten. *The Rational Unified Process Made Easy: A Practitioner's Guide to the RUP*. Addison-Wesley, 2003.

[100] J. Küster. *Consistency Management of Object-Oriented Behavioral Models*. PhD thesis, University of Paderborn, March 2004.

[101] J. Küster, R. Heckel, and G. Engels. Defining and validating transformations of UML models. In *Proceedings of the 2003 IEEE Symposium on Human Centric Computing Languages and Environments (HCC'03)*, pages 145–152. IEEE Computer Society, 2003.

[102] C. Lange. *Assessing and Improving the Quality of Modeling - A Series of Empirical Studies about the UML*. PhD thesis, Technical University of Eindhoven, 2007.

[103] C. Lange and M. Chaudron. An Empirical Assessment of Completeness in UML Design. In *Proceedings of the 8th Conference on Empirical Assessment in Software Engineering (EASE'04)*, pages 111–121, 2004.

[104] C. Lange and M. Chaudron. Effects of Defects in UML Models - An Experimental Investigation. In *Proceedings of the 28th International Conference on Software Engineering (ICSE'06)*, pages 401–411, 2006.

[105] C. Lange, M. Wijns, and M. Chaudron. A Visualization Framework for Task-Oriented Modeling Using UML. In *Proceedings of the 40th Hawaiian International Conference on System Sciences (HICSS'07)*, page 289. IEEE Computer Society, 2007.

[106] W. Li and S. Henry. Maintenance Metrics for the Object-Oriented Paradigm. In *Proceedings of the 1st International Software Metrics Symposium*, pages 52–60, 1993.

[107] P. Liggesmeyer. *Software-Qualität*. Spektrum Akademischer Verlag, 2002.

[108] S. Linker. Ein UML-Profil für CSP-OZ-DC. Technical report, Carl von Ossietzky Universität Oldenburg, 2005.

[109] M. Lorenz and J. Kidd. *Object-Oriented Software Metrics: A Practical Guide*. Prentice Hall, 1994.

[110] J. Ludewig. Models in Software Engineering. *Software and System Modeling*, 2(1):5–14, 2003.

[111] S. MacDonell. Deriving Relevant Functional Measures for Automated Development Projects. *Information and Software Technology*, 35:499–512, 1993.

[112] M. Marchesi. OOA Metrics for the Unified Modeling Language. In *Proceedings of the 2nd Euromicro Conference on Software Maintenance and Reengineering (CSMR'98)*, pages 67–74. IEEE Computer Society, 1998.

[113] J. McCall, P. Richards, and G. Walters. Factors in Software Quality; Volumes I, II, and III. Technical Report NTIS AD/A-049 014, NTIS AD/A-049 015 and NTIS AD/A-049 055, US Rome Air Development Center, 1977.

[114] K. L. McMillan. *Symbolic Model Checking: An Approach to the State Explosion Problem.* Kluwer Academic Publishers, 1993.

[115] J. A. McQuillan and J. F. Power. A Definition of the Chidamber and Kemerer Metrics suite for UML. Technical Report NUIM-CS-TR-2006-03, Department of Computer Science, National University of Ireland, 2006.

[116] T. Mens. *Software Evolution*, chapter Introduction and Roadmap: History and Challenges of Software Evolution, pages 1–11. Springer, 2008.

[117] T. Mens, N. V. Eetvelde, S. Demeyer, and D. Janssens. Formalizing refactorings with graph transformations. *Journal of Software Maintenance*, 17(4):247–276, 2005.

[118] C. W. Mertz, L. E. Hyatt, and A. Robinson. Software Formal Inspections Guidebook: National Aeronautics and Space Administration. Software Formal Inspections Standard. Technical Report NASA-STD-2202-9, NASA, 1993.

[119] P. Mohagheghi and V. Dehlen. Where Is the Proof? - A Review of Experiences from Applying MDE in Industry. In *Proceedings of the 4th European conference on Model Driven Architecture (ECMDA-FA'08)*, pages 432–443. Springer, 2008.

[120] M. Möller, E.-R. Olderog, H. Rasch, and H. Wehrheim. Integrating a formal method into a software engineering process with UML and Java. *Formal Aspects of Computing*, 20(2):161–204, 2008.

[121] D. Moody. Metrics for Evaluating the Quality of Entity Relationship Models. In *Proceedings of the 17th International Conference on Conceptual Modeling (ER'98)*, volume 1507 of *Lecture Notes in Computer Science*, pages 211–225. Springer, 1998.

[122] D. Moody and G. Shanks. What Makes a Good Data Model? Evaluating the Quality of Entity Relationship Models. In *Proceedings of the 13th International Conference on the Entity Relationship Approach, Business Modelling and Re-Engineering (ER'94)*, volume 881 of *Lecture Notes in Computer Science*, pages 94–111. Springer, 1994.

[123] D. L. Moody, G. Sindre, T. Brasethvik, and A. Solvberg. Evaluating the quality of information models: empirical testing of a conceptual model quality framework. In *Proceedings of the 25th International Conference on Software Engineering (ICSE '03)*, pages 295–305. IEEE Computer Society, 2003.

[124] J. Noack. *Techniken der objektorientierten Softwareentwicklung.* Springer, 2001.

[125] Object Management Group (OMG). Official Homepage. `http://www.omg.org/`.

[126] Object Management Group (OMG). Model Driven Architecture (MDA) Guide Version 1.0.1. `http://www.omg.org/docs/omg/03-06-01.pdf`, 2003.

[127] Object Management Group (OMG). Object Constraint Language (OCL) 2.0 Specification. `http://www.omg.org/docs/ptc/05-06-06.pdf`, 2005.

[128] Object Management Group (OMG). UML Testing Profile. `http://www.omg.org/cgi-bin/doc?formal/05-07-07`, 2005.

[129] Object Management Group (OMG). Meta Object Facility (MOF) Core Specification, Version 2.0, 2006.

[130] Object Management Group (OMG). Unified Modeling Language: Infrastructure. `http://www.omg.org/spec/UML/2.1.1/`, February 2007.

[131] Object Management Group (OMG). Unified Modeling Language (UML): Superstructure 2.1.1. `http://www.omg.org/docs/formal/07-02-03.pdf`, 2007.

[132] Object Management Group (OMG). Software & Systems Process Engineering Meta-Model Specification, Version 2.0. `http://www.omg.org/technology/documents/formal/spem.htm`, 2008.

[133] L. O'Brien, L. Bass, and P. Merson. Quality Attributes and Service-Oriented Architectures. Technical Report CMU/SEI-2005-TN-014, Software Engineering Institute Carnegie Mellon, 2005.

[134] L. O'Brien, P. Merson, and L. Bass. Quality Attributes for Service-Oriented Architectures. In *Proceedings of the 29th International Conference on Software Engineering Workshops (ICSEW '07)*, page 113. IEEE Computer Society, 2007.

[135] B. Oestereich. *Objektorientierte Softwareentwicklung - Analyse und Design mit der UML 2*. Oldenbourg, 2004.

[136] M. Oivo and V. Basili. Representing Software Engineering Models: The TAME Goal Oriented Approach. *IEEE Transactions on Software Engineering*, 18(10):886–898, 1992.

[137] N. Ojha and J. McGregor. Object oriented metrics for early system characterization: A CRC card based approach. Technical Report TR 94-107, Deptartment of Computer Science, Clemson University, 1994.

[138] T. Olsson and P. Runeson. V-GQM: A Feed-Back Approach to Validation of a GQM Study. In *Proceedings of the 7th IEEE International Software Metrics Symposium (METRICS'01)*, pages 236–245. IEEE Computer Society, 2001.

[139] Oxford University Press. Oxford Reference Online (ORO). `http://www.oxfordreference.com/` [online; accessed 09-07-2009], 2004.

[140] M. Page-Jones. *What every programmer should know about object-oriented design*. Dorset House Publishing Co., 1995.

[141] R. Park, W. Goethert, and W. Florac. Goal-Driven Software Measurement - A Guidebook. Technical Report CMU/SEI-96-HB-002, Software Engineering Institute, Carnegie Mellon University, August 1996.

[142] K. Popper. *Die beiden Grundprobleme der Erkenntnistheorie*. Mohr, 1994.

[143] L. Prechelt, B. Unger, M. Philippsen, and W. Tichy. A controlled experiment on inheritance depth as a cost factor for code maintenance. *Journal of Systems and Software*, 65(2):115–126, 2003.

[144] A. Pretschner and J. Philipps. Methodological Issues in Model-Based Testing. In *Model-Based Testing of Reactive Systems*, volume 3472 of *Lecture Notes in Computer Science*, pages 281–291. Springer, 2004.

[145] R. Reißing. *Bewertung der Qualität objektorientierter Entwürfe [Assessment of the Quality of Object-Oriented Designs]*. PhD thesis, Universität Stuttgart, Fakultät Informatik, 2002.

[146] W. Reisig. *Petrinetze - Eine Einführung*. Springer, 1990.

[147] L. Reynoso, M. Genero, and M. Piattini. Validating Metrics for OCL Expressions Expressed within UML/OCL models. In *Proceedings of the 1st International Workshop on Software Audit and Metrics (SAM'04)*, pages 59–68. INSTICC Press, 2004.

[148] A. Roscoe. *The Theory and Practice of Concurrency*. Prentice Hall Series in Computer Science, 1998.

[149] RTI. The Economic Impacts of Inadequate Infrastructure for Software Testing. Technical report, International University of North Carolina, Research Triangle Institute, Health, Social, and Economics Research, 2002. `http://www.nist.gov/director/prog-ofc/report02-3.pdf` [online; Zugriff 4.11.2008].

[150] T. Ruhroth, H. Voigt, and H. Wehrheim. Measure, diagnose, refactor: A formal quality cycle for software models (to appear). In *Proceedings of the 35th EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA'09)*. IEEE Computer Society, 2009.

[151] F. Salger, M. Bennicke, G. Engels, and C. Lewerentz. Comprehensive Architecture Evaluation and Management in Large Software-Systems. In *Proceedings of the 4th International Conference on Quality of Software-Architectures (QoSA '08)*, pages 205–219. Springer, 2008.

[152] D. Schmidt. Guest Editor's Introduction: Model-Driven Engineering. *IEEE Computer*, 39(2):25–31, 2006.

[153] R. Schütte. Die neuen Grundsätze ordnungsmäßiger Modellierung. In *Paper zum Forschungsforum*, 1997.

[154] R. Schütte. Vergleich alternativer Ansätze zur Bewertung der Informationsmodellqualität. *Informationssystem-Architekturen*, Heft 2:49–55, 1998.

[155] R. Schütte and T. Rotthowe. The Guidelines of Modeling - An Approach to Enhance the Quality in Information Models. In *Proceedings of the 17th International Conference on Conceptual Modeling (ER'98)*, Lecture Notes in Computer Science, pages 240–254. Springer, 1998.

[156] Software Engineering Institute Carnegie Mellon (SEI). Capability Maturity Model Integration (CMMI). `http://www.sei.cmu.edu/cmmi/`, 2006.

[157] Software Engineering Institute Carnegie Mellon (SEI). CMMI-DEV, V1.2: Capability Maturity Model Integration for Development, Version 1.2: Improving processes for better products. `http://www.sei.cmu.edu/pub/documents/06.reports/pdf/06tr008.pdf`, 2006.

[158] C. Soltenborn. Analysis of UML Workflow diagrams with Dynamic Meta Modeling techniques. Master's thesis, University of Paderborn, Department of Computer Science, June 2006.

[159] I. Sommerville. *Software Engineering*. Addison Wesley, 7th edition, 2004.

[160] A. Spillner and T. Linz. *Basiswissen Softwaretest*. dpunkt.verlag, 2005.

[161] H. Stachowiak. *Allgemeine Modelltheorie*. Springer-Verlag, Wien, 1973.

[162] H. Störrle. *UML 2 erfolgreich anwenden*. Addison-Wesley, 2005.

[163] R. V. D. Straeten and M. D'Hondt. Model Refactorings through Rule-Based Inconsistency Resolution. In *Proceedings of the ACM Symposium on Applied Computing*, pages 1210–1217, 2006.

[164] R. V. D. Straeten, V. Jonckers, and T. Mens. Supporting Model Refactorings Through Behaviour Inheritance Consistencies. In *Proceedings of the 7th International Conference on The Unified Modelling Language: Modelling Languages and Applications (UML'04)*, volume 3273 of *Lecture Notes in Computer Science*, pages 305–319. Springer, 2004.

[165] Sun Microsystems. Java code conventions. Technical report, 1997.

[166] T. Sunazuka, M. Azuma, and N. Yamagishi. Software quality assessment technology. In *Proceedings of the 8th international conference on Software engineering (ICSE '85)*, pages 142–148. IEEE Computer Society Press, 1985.

[167] H. Toutenburg and P. Knöfel. *Six Sigma: Methoden und Statistik für die Praxis*. Springer, 2008.

[168] A. Trifu and R. Marinescu. Diagnosing Design Problems in Object Oriented Systems. In *Proceedings of the 12th Working Conference on Reverse Engineering (WCRE'05)*, pages 155–164. IEEE Computer Society, 2005.

[169] J. Trost and A. Cavarra. AGEDIS Modelling Language Tutorial. Technical report, 2005.

[170] B. Unger and L. Prechelt. The impact of inheritance depth on maintenance tasks: Detailed description and evaluation of two experiment replications. Technical report, Universität Karlsruhe, Fakultät für Informatik, Germany, 1998.

[171] D. Varró. *Automated Model Transformations for the Analysis of IT Systems*. PhD thesis, Budapest University of Technology and Economics, 2003.

[172] H. Voigt. Modell-basierte Analyse von ausführbaren Geschäftsprozessen für Web Services. Master's thesis, University of Paderborn, Department of Computer Science, 2003.

[173] H. Voigt and G. Engels. Kontextsensitive Qualitätsplanung für Software-Modelle. In *Proceedings of Modellierung*, volume 127 of *LNI*, pages 165–180. Lecture Notes in Informatics, 2008.

[174] H. Voigt, B. Güldali, and G. Engels. Measuring the Testability of Models by Quality Plans. In *Proceedings of the 11th International Conference on Quality Engineering in Software Technology (CONQUEST'08)*, pages 353–370, 2008.

[175] H. Voigt and T. Ruhroth. A Quality Circle Tool for Software Models. In *Proceedings of the 27th International Conference on Conceptual Modeling (ER'08)*, volume 5231 of *Lecture Notes in Computer Science*, pages 526–527. Springer, 2008.

[176] E. Wallmüller. *Software-Qualitätsmanagement in der Praxis - Software-Qualität durch Führung und Verbesserung von Software-Prozessen.* Hanser, 2001.

[177] D. Wheeler, B. Brykczynski, and R. Meeson. *Software Inspection - An Industry Best Practice.* IEEE Computer Society, 1996.

[178] K. Wiegers. *Software Requirements.* Microsoft Press Corp., 2nd edition, 2003.

[179] N. Wiener. A new theory of measurement: A study in the logic of mathematics. In *Proceedings of the London Mathematical Society*, pages 181–205, 1921.

[180] Wikipedia. Formal methods. `http://en.wikipedia.org/wiki/Formal_methods` [online; accessed 02-09-2009].

[181] World Wide Web Consortium (W3C). XQuery 1.0: An XML Query Language. `http://www.w3.org/TR/xquery/` [online; accessed 9-07-2009], January 2007.

[182] L. Yusuf, M. Chessell, and T. Gardner. Implement model-driven development to increase the business value of your IT system. *IBM developerWorks*, 2006. `http://www.ibm.com/developerworks/library/ar-mdd1/` [online; accessed 09-07-2009].

[183] A. Zamperoni. *Formal Integration of Software Engineering Aspects.* PhD thesis, University of Paderborn, 1999.

Anhang A

Messungen

A.1 Objektive Metriken

A.1.1 OCL-Metriken

Objective Metric		
Name		Abstract Classes in SD
Acronym		ACiSD
Measurement-method	Informal Definition	Number of instantiated abstract classes in all sequence diagrams.
	Formal Definition	**context** Model def:ACiSD(): **Integer** = self.getAllInteractions()−> iterate (i: Interaction; acc: **Set**(Lifeline) = **Set**{} | i.getAllLifelines() −> iterate (l: Lifeline; accLifeline :**Bag**(Lifeline) = **Bag**{} | **if** (l.getClass().isAbstract) **then** accLifeline−>including(l) **else** accLifeline **endif**) acc−>union(accLifeline))−> size()
Type of Measurement		Objective
Unit of Measurement		Model
Scope		Class Diagram; Sequence Diagram
Scale (Type)		$0 \leq$ ACiSD (Absolute)
Interpretation		The closer to 0, the better.

Messung 1: Abstract Classes in SD

Objective Metric		
Name		Association with Navigation
Acronym		AwN
Measurement-method	Informal Definition	Divide the number of associations with navigation arrows by the number of all associations.
	Formal Definition	**context** Model def:AwN(): **Real** = (self.getAllAssociations()−> iterate (assoc:Association; acc:**Real** = 0 | acc + assoc.navigableOwnedEnd−>size ()))/ self.getAllAssociations()−>size()
Type of Measurement		Objective
Unit of Measurement		Model
Scope		Class Diagram
Scale (Type)		$0 \leq$ AwN (Ratio)
Interpretation		The closer to 1, the better.

Messung 2: Association with Navigation

Objective Metric		
Name		Association with unlimited Cardinalities
Acronym		AuC
Measurement-method	Informal Definition	Divide the number of unlimited cardinalities by the number of all cardinalities defined for all associations in the software model.
	Formal Definition	**context** Model def:AuC(): **Real** = (MultiplicityElement.allInstances()−> iterate (me: MultiplicityElement; acc:**Real** = 0 | **if** ((me.lower = 0)**and** (me.upper = ∗))**then** acc+1 **else** acc **endif**))/ MultiplicityElement.allInstances()−>size()
Type of Measurement		Objective
Unit of Measurement		Model
Scope		Class Diagram
Scale (Type)		$0 \leq$ AuC ≤ 1 (Ratio)
Interpretation		A value close to 1 is poor.

Messung 3: Association with unlimited Cardinalities

Objective Metric		
Name		Associations and Attributes
Acronym		NoAA
Measurement-method	Informal Definition	Number of available associations and available attributes for all classes
	Formal Definition	**context** Model def:NoAA(): **Integer** = self. getAllClasses()−> iterate (c:Class; acc:**Integer** = 0 | acc = acc + c.getAvailableAttributes()−>size()+ c. getAvailableAssociations()−>size())
Type of Measurement		Objective
Unit of Measurement		Model
Scope		Class Diagram
Scale (Type)		$0 \leq$ NoAA (Absolute)
Interpretation		The higher, the more complex.

Messung 4: Associations and Attributes

Objective Metric		
Name		Border Crossings
Acronym		NBC
Measurement-method	Informal Definition	Number of border crossings of transitions in a statechart.
	Formal Definition	**context** Class def:NBC(): **Integer** = self.getStatechart ().getAllTransitions() −> iterate (t: Transition; acc :**Integer** = 0 | **if** (t.target.container <> t.source. container) acc + 1 **else** acc **endif**)
Type of Measurement		Objective
Unit of Measurement		Class
Scope		Statechart
Scale (Type)		0 ≤ NBC (Absolute)
Interpretation		The closer to 0, the better.

Messung 5: Border Crossings

Objective Metric		
Name		Class not in SD
Acronym		CnSD
Measurement-method	Informal Definition	Number of classes that are not instantiated in any sequence diagram.
	Formal Definition	**context** Model def:CnSD(): **Integer** = self. getAllClasses()−> iterate (c:Class; acc:**Set**(Class) = **Set**{} | **if** (self .getAllClassesRelatedToObjectsInSD() −>include(c))**then** acc−>including(c)**else** acc **endif**) −> size()
Type of Measurement		Objective
Unit of Measurement		Model
Scope		Class Diagram; Sequence Diagram
Scale (Type)		0 ≤ CnSD (Absolute)
Interpretation		The closer to 0, the better.

Messung 6: Class not in SD

Objective Metric		
Name		Classes used as Attribute Types
Acronym		DAC
Measurement-method	Informal Definition	Number of different classes that are used as types of attributes in a class.
	Formal Definition	**context** Class def:DAC(): **Integer** = self.getAllAttributes()−> iterate (p:Property; acc:**Bag**(Property)= **Bag**{} | **if** (self.getModel().getAllClasses()−>excluding(self)−>includes(p.datatype))**then** acc−>including(p)**else** acc **endif**)−> size()
Type of Measurement		Objective
Unit of Measurement		Class
Scope		Class Diagram
Scale (Type)		0 ≤ DAC (Absolute)
Interpretation		A high value is poor.

Messung 7: Classes used as Attribute Types

Objective Metric		
Name		Classes without Methods
Acronym		CnM
Measurement-method	Informal Definition	Number of classes without methods.
	Formal Definition	**context** Model def:CnM(): **Integer** = self.getAllClasses()−> iterate (c:Class; acc:**Integer** = 0 | **if** (c.getAvailableOperations().isEmpty()) **then** acc +1 **else** acc **endif**)
Type of Measurement		Objective
Unit of Measurement		Model
Scope		Class Diagram
Scale (Type)		0 ≤ CnM (Absolute)
Interpretation		The closer to 0, the better.

Messung 8: Classes without Methods

Objective Metric		
Name		Depth of Inheritance Tree
Acronym		DIT
Measurement-method	Informal Definition	Maximum inheritance path from the class to the root class.
	Formal Definition	**context** Class def:DIT(): **Integer** = (**if** self.IsRoot() **then** 0 **else** 1 + self.parents()−>iterate(elem:Class; acc:**Integer**=0 if elem.DIT() > acc then elem.DIT() else acc endif) endif)|
Type of Measurement		Objective
Unit of Measurement		Class
Scope		Class diagram
Scale (Type)		$0 \leq$ DIT (Absolute)
Interpretation		The deeper a class is in the hierarchy, the more methods and variables it is likely to inherit, making it more complex.

Messung 9: Depth of Inheritance Tree

Objective Metric		
Name		Directly Related Classes
Acronym		DCC
Measurement-method	Informal Definition	The Direct Class Coupling metric is a count of the different number of classes that a class is directly related to. The metric includes classes that are directly related by attribute declarations and message passing (parameters) in methods.
	Formal Definition	**context** Class def:DCC(): **Integer** = self. getAllAttributes()−> iterate (p:Property; acc:**Bag** (Class)= **Bag**{} | acc−>including(p.datatype)) −> union(self.getAllAssociations()−> iterate (a: Association; acc:**Bag**(Class)= **Bag**{} | acc−>including (a.memberEnd.asBag()))union(self.getAllOperations ()−> iterate (op:Operation; acc:**Bag**(Class)= **Bag**{} | **if**(acc−>including(op.ownedParameter.asBag()))) intersection(self.getAllClasses())−> excluding(self)
Type of Measurement		Objective
Unit of Measurement		Class
Scope		Class Diagram
Scale (Type)		$0 \leq$ DCC (Absolute)
Interpretation		A high value is poor.

Messung 10: Directly Related Classes

Objective Metric		
Name		Defined Operations
Acronym		DO
Measurement-method	Informal Definition	Number of the defined operations of a class.
	Formal Definition	**context** Class def:DO(): **Integer** = self.definedOperations()−>size()
Type of Measurement		Objective
Unit of Measurement		Class
Scope		Class Diagram
Scale (Type)		$0 \leq$ DO (Absolute)
Interpretation		n/a.

Messung 11: Defined Operations

Objective Metric		
Name		Invocation Across Layers
Acronym		IAL
Measurement-method	Informal Definition	Number of method invocations from objects that are instances of boundary-classes to objects that are instances of entity-classes and vice versa in all sequence diagrams.
	Formal Definition	**context** Model def:IAL(): **Integer** = self. getAllInteractions() −> iterate (i : Interaction ; acc: **Set**(Message){} | acc−>including(i.getAllMessages())) −> iterate (m:Message; accCount:**Integer** = 0 | **if**((m .sendEvent.getClass().isStereotypedByEntity()**and** m. receiveEvent.getClass().isStereotypedByBoundary())**or** (m.receiveEvent.getClass().isStereotypedByEntity()**and** m.sendEvent.getClass().isStereotypedByBoundary())) **then** accCount+1 **else** accCount **endif**)
Type of Measurement		Objective
Unit of Measurement		Model
Scope		Sequence Diagram
Scale (Type)		$0 \leq$ IAL (Absolute)
Interpretation		The closer to 0, the better.

Messung 12: Invocation Across Layers

Objective Metric		
Name		Lack of Cohesion of Methods
Acronym		LCOM4
Measurement-method	Informal Definition	Number of connected components in a class. A connected component is a set of related methods (and class-level variables).
	Formal Definition	**context** Class def:LCOM4(): **Integer** = self. getAvailableOperations()−> iterate (op:Operation; acc:**Sequence** = **Sequence**{**Set**{}, 0} | **if**(acc−> indexOf(0)−>excludes(op))acc−>insertAt(0,acc−> indexOf(0)−>including(op.getRelatedOperations(self, op.getDeltaList()))))−>insertAt(1,acc−>indexOf(1)+1) **else** acc **endif**)−>indexOf(1)
Type of Measurement		Objective
Unit of Measurement		Class
Scope		Class Diagram, Z description
Scale (Type)		$0 \leq$ LCOM4 (Absolute)
Interpretation		LCOM4 = 1 indicates a cohesive class, which is the good class. LCOM4 $\geq$ 2 indicates a problem. The class should be split into so many smaller classes. LCOM4 = 0 happens when there are no methods in a class. This is also a "bad" class.

Messung 13: Lack of Cohesion of Methods

Objective Metric		
Name		Maximum of Attributes
Acronym		ECMaxA
Measurement-method	Informal Definition	Maxium number of attributes of all classes stereotyped with «Entity»
	Formal Definition	**context** Model def:ECMaxA(): **Integer** = Model. getAllEntityClasses()−> iterate (c:Class; acc:**Integer** = 0 | **if** (acc > c.getAvailableAttributes()−>size()) **then** acc **else** acc = c.getAvailableAttributes()−>size() **endif**)
Type of Measurement		Objective
Unit of Measurement		Model
Scope		Class Diagram
Scale (Type)		0 ≤ ECMaxA (Absolute)
Interpretation		The closer to 10, the better.

Messung 14: Maximum of Attributes

Objective Metric		
Name		Maximum of Operations
Acronym		CCMaxOp
Measurement-method	Informal Definition	Maxium number of operations of all classes stereotyped with «Control»
	Formal Definition	**context** Model def:CCMaxOp(): **Integer** = Model. getAllControlClasses()−> iterate (c:Class; acc:**Integer** = 0 | **if** (acc > c.getAvailableOperations()−>size()) **then** acc **else** acc = c.getAvailableOperations()−>size ()**endif**)
Type of Measurement		Objective
Unit of Measurement		Model
Scope		Class Diagram
Scale (Type)		0 ≤ CCMaxOp (Absolute)
Interpretation		The closer to 10, the better.

Messung 15: Maximum of Operations

Objective Metric		
Name		Messages
Acronym		NoM
Measurement-method	Informal Definition	Number of messages in all sequence diagrams
	Formal Definition	**context** Model def:NoM(): **Integer** = self.getAllInteractions()−> iterate (i: Interaction ; acc: **Integer** = 0 | acc + i.getAllMessages()−>size())
Type of Measurement		Objective
Unit of Measurement		Model
Scope		Sequence Diagram
Scale (Type)		0 ≤ NoM (Absolute)
Interpretation		The higher, the more complex.

Messung 16: Messages

Objective Metric		
Name		Messages without Method
Acronym		EcM
Measurement-method	Informal Definition	Number of messages that have not a corresponding method in the class definition of the callee.
	Formal Definition	**context** Model def:EcM(): **Integer** = self.getAllInteractions()−> iterate (i: Interaction ; acc: **Set**(Message)= **Bag**{} | i.getAllMessages()−> iterate (m:Message; accMessage:**Bag**(Message)= **Bag**{} | **if** (m.receiveEvent.getClass().getAvailableOperations()−> excludes(m.signature))**then** accMessage−>including(m)**else** accMessage **endif**)acc−>union(accMessage))−> size()
Type of Measurement		Objective
Unit of Measurement		Model
Scope		Class Diagram; Sequence Diagram
Scale (Type)		0 ≤ EcM (Absolute)
Interpretation		The closer to 0, the better.

Messung 17: Messages without Method

Objective Metric		
Name		Messages without Name
Acronym		EnN
Measurement-method	Informal Definition	Number of messages that are not annotated with a name.
	Formal Definition	**context** Model def:EnN(): **Integer** = self.getAllInteractions()−> iterate (i : Interaction ; acc :**Integer** = 0 | i.getAllMessages() −> iterate (m: Message; accMessage:**Integer** = 0 | **if** (m.name.size() = 0)**then** accMessage+1 **else** accMessage **endif**)acc+ accMessage)−>size()
Type of Measurement		Objective
Unit of Measurement		Model
Scope		Sequence Diagram
Scale (Type)		$0 \leq$ EnN (Absolute)
Interpretation		The closer to 0, the better.

Messung 18: Messages without Name

Objective Metric		
Name		Method not called in SD
Acronym		MnSD
Measurement-method	Informal Definition	Number of methods that are defined in classes, but that are not called as a message in any sequence diagram.
	Formal Definition	**context** Model def:MnSD(): **Integer** = self.getAllClasses()−> iterate (c:Class; acc:**Set**(Operation)= **Set**{} | c.getAllOperations() −> iterate (op: Operation; accClassOp:**Set**(Operation)= **Set**{} | **if** (op.hasNoMessageInSD())**then** accClassOp−>including(op)**else** accClassOp **endif**)acc−>union(accClassOp))−>size()
Type of Measurement		Objective
Unit of Measurement		Model
Scope		Class Diagram; Sequence Diagram
Scale (Type)		$0 \leq$ MnSD (Absolute)
Interpretation		The closer to 0, the better.

Messung 19: Method not called in SD

Objective Metric		
Name		Methods called by a Class
Acronym		McC
Measurement-method	Informal Definition	Number of methods of other classes called by a class
	Formal Definition	**context** Class def:McC(): **Integer** = self. getAllInteractionsClassIsInvolved() −> iterate (i: Interaction; acc:**Integer** = 0 | i.getAllMessages() −> iterate (m:Message; accMessage: **Integer** = 0 | **if**(m.sendEvent = self **and** m.receiveEvent <> self) **then** accMessage + 1 **else** accMessage **endif**)acc + accMessage)
Type of Measurement		Objective
Unit of Measurement		Class
Scope		Sequence Diagram
Scale (Type)		$0 \leq$ McC (Absolute)
Interpretation		A high value is poor.

Messung 20: Methods called by a Class

Objective Metric		
Name		Nesting Level
Acronym		MaxNLCS
Measurement-method	Informal Definition	Maximum nesting level of composite states.
	Formal Definition	**context** Class def:MaxNLCS(): **Integer** = self.getStatechart().getStates()−> iterate (s:State; acc: **Integer** = 0 | **if**(s.getNestingLevel() > acc) **then** acc = s.getNestingLevel() **else** acc)
Type of Measurement		Objective
Unit of Measurement		Class
Scope		Statechart
Scale (Type)		0 ≤ MaxNLCS (Absolute)
Interpretation		A too high value is poor.

Messung 21: Nesting Level

Objective Metric		
Name		Object has no Class in CD
Acronym		CnCD
Measurement-method	Informal Definition	Number of objects in all sequence diagrams that are not defined in any class diagram.
	Formal Definition	**context** Model def:CnCD(): **Integer** = self.getAllInteractions()−> iterate (i:Interaction; acc: **Set**(Lifeline) = **Set**{} | i.getAllLifelines() −> iterate (l: Lifeline; accLifeline :**Bag**(Lifeline) = **Bag**{} | **if** (l.hasNoClassDefinition()) **then** accLifeline−>including(l) **else** accLifeline **endif**) acc−>union(accLifeline))−>size()
Type of Measurement		Objective
Unit of Measurement		Model
Scope		Class Diagram; Sequence Diagram
Scale (Type)		0 ≤ CnCD (Absolute)
Interpretation		The closer to 0, the better.

Messung 22: Object has no Class in CD

Objective Metric		
Name		Operations Inheritance Factor
Acronym		OIF
Measurement-method	Informal Definition	Number of inherited operations divided by the total number of available operations.
	Formal Definition	**context** Model def:OIF(): **Real** = self.TOI()/ self.TOA()
Type of Measurement		Objective
Unit of Measurement		Model
Scope		Class diagram
Scale (Type)		$0 \leq \text{OIF} \leq 1$ (Ratio)
Interpretation		OIF should be in a reasonable range, not too low and not too high either. A too high a value indicates either superfluous inheritance or too wide member scopes. A low value indicates lack of inheritance or heavy use of overrides.

Messung 23: Operations Inheritance Factor

Objective Metric		
Name		Overridden Operations
Acronym		OO
Measurement-method	Informal Definition	Number of overridden operations.
	Formal Definition	**context** Class def:OO(): **Integer** = self .definedOperations()−>intersection(self. inheritedOperations())−>size()
Type of Measurement		Objective
Unit of Measurement		Class
Scope		Class diagram
Scale (Type)		0 ≤ OO (Absolute)
Interpretation		A too high value is poor.

Messung 24: Overridden Operations

Objective Metric		
Name		Parameter List
Acronym		MaxPL
Measurement-method	Informal Definition	Maximum length of all parameter lists used by a class.
	Formal Definition	**context** Class def:MaxPL(): **Integer** = self. definedOperations()−> iterate (o : Operation ; acc : **Integer** = 0 | **if** (o.ownedParameter−>size()> acc) **then** acc = o.ownedParameter−>size()**else** acc **endif**)
Type of Measurement		Objective
Unit of Measurement		Class
Scope		Class diagram; Z description
Scale (Type)		0 ≤ MaxPL (Absolute)
Interpretation		A too high value is poor.

Messung 25: Parameter List

Objective Metric		
Name		Relatedness among Methods
Acronym		CAMC
Measurement-method	Informal Definition	This measure computes the relatedness among methods of a class based upon the parameter list of the methods. The metric is computed using the summation of the intersection of parameters of a method with the maximum independent set of all parameter types in the class.
	Formal Definition	**context** Class def:CAMC(): **Real** = self. getAllOperations()−> iterate (op:Operation; acc: **Integer** = 0 | self.getAllParametersInOperations() .asBag()−> iterate (p:Paramter; accRow: **Integer** = 0 | **if** (op.ownedParameter−>includes(p))**then** accRow + 1 **else** acc **endif**)acc + accRow)/ self. getAllParametersInOperations().asBag()−>size()∗ self. getAllOperations()−>size()
Type of Measurement		Objective
Unit of Measurement		Class
Scope		Class Diagram
Scale (Type)		$0 \leq$ CnCD ≤ 1(Ratio)
Interpretation		The closer to 1, the better.

Messung 26: Relatedness among Methods

Objective Metric		
Name		Tight and Loose Class Cohesion
Acronym		TCC
Measurement-method	Informal Definition	Number of direct via variables connected methods divided by the maximum number of possible connections.
	Formal Definition	**context** Class def:TCC(): **Real** = self. getAvailableOperations()−> iterate (op:Operation; acc1:**Integer** = 0 | acc1 + (self.getAvailableOperations () −> iterate (op2:Operation; acc2:**Integer** = 0 | **if**(op.isRelatedTo(op2,op.getDeltaList())) **then** acc2 + 1 **else** acc2 **endif**))/ (self .getAvailableOperations()−> size()∗ (self .getAvailableOperations()−>size()− 1))
Type of Measurement		Objective
Unit of Measurement		Class
Scope		Class diagram; Z description
Scale (Type)		$0 \leq$ TCC ≤ 1(Ratio)
Interpretation		The closer to 1, the better.

Messung 27: Tight and Loose Class Cohesion

Objective Metric		
Name		Top Level Parallelizations
Acronym		NTLP
Measurement-method	Informal Definition	Number of top level parallelizations.
	Formal Definition	**context** Class def:NTLP(): **Integer** = self. getStatechart().region−>size()
Type of Measurement		Objective
Unit of Measurement		Class
Scope		Statechart
Scale (Type)		$0 \leq$ NTLP (Absolute)
Interpretation		n/a

Messung 28: Top Level Parallelizations

Objective Metric		
Name		Total Defined Operations
Acronym		TDO
Measurement-method	Informal Definition	Total number of operations defined in the software model.
	Formal Definition	**context** Model def:TDO(): **Integer** = self.allClasses ()−>iterate(elem: Class; acc: **Integer** = 0 acc + elem.DO())|
Type of Measurement		Objective
Unit of Measurement		Model
Scope		Class Diagram
Scale (Type)		0 ≤ TDO (Absolute)
Interpretation		n/a.

Messung 29: Total Defined Operations

Objective Metric		
Name		Transitions to Operations
Acronym		NT2NOps
Measurement-method	Informal Definition	Divide the number of transitions in a statechart by the number of available operations of a class (cp. XQuery-Measurement in Tab. 67 on p. 300).
	Formal Definition	**context** Class def:NT2NOps(): **Integer** = self. getStatechart().getTransitions()−>size()/ self. getAvailableOperations()−>size()
Type of Measurement		Objective
Unit of Measurement		Class
Scope		Class Diagram, Statechart
Scale (Type)		0 ≤ NT2NOps (Ratio)
Interpretation		A value higher than 2 is poor.

Messung 30: Transitions to Operations

Objective Metric		
Name		Unnamed Associations
Acronym		UA2NA
Measurement-method	Informal Definition	Divide the number of unnamed associations by the number of labeled assocations considering roles.
	Formal Definition	**context** Model def:UA2NA(): **Real** = Association.allInstances()−> iterate(elem: Association; acc:**Integer** = 0 | **if**((((elem.name−>size()= 0)**or** (elem.name−>size().oclIsTypeOf(**Integer**)= false)))**and** elem.AssociationEnd.oclIsUndefined())**then** acc + 1 **else** acc **endif**)/ Association.allInstances()−>size()
Type of Measurement		Objective
Unit of Measurement		Model
Scope		Class diagram
Scale (Type)		$0 \leq$ UA2NA ≤ 1 (Ratio)
Interpretation		The closer to 0, the better.

Messung 31: Unnamed Associations

Objective Metric		
Name		Untyped Attributes
Acronym		UnAtt
Measurement-method	Informal Definition	Number of attributes without type declaration (cp. XQuery-Measurement in Tab. 36 on p. 283).
	Formal Definition	**context** Model def:UnAtt(): **Integer** = self.getAllAttributes()−> iterate (elem:Property; acc: **Integer** = 0 | **if** (elem.datatype−>oclIsUndefined()) **then** acc + 1 **else** acc
Type of Measurement		Objective
Unit of Measurement		Model
Scope		Class diagram
Scale (Type)		$0 \leq$ UnAtt (Absolute)
Interpretation		The closer to 0, the better.

Messung 32: Untyped Attributes

A.1.2 XQuery-Metriken

Objective Metric		
Name		Attribute Names Begin Small
Acronym		ANBS
Measurement-method	Informal Definition	Number of attribute names that do not start with a lower case letter.
	Formal Definition	declare function local:ANBS()as xs:double {fn:count($doc/Model/Class/Attribute/Name[fn:matches(., "^[^a-z]")])};
Type of Measurement		Objective
Unit of Measurement		Model
Scope		Class diagram
Scale (Type)		$0 \leq$ ANBS (Absolute)
Interpretation		The closer to 0, the better.

Messung 33: Attribute Names Begin Small

Objective Metric		
Name		Attribute Names Include Spaces
Acronym		ANIS
Measurement-method	Informal Definition	Number of attribute names that do include spaces in name.
	Formal Definition	declare function local:ANIS()as xs:double {fn:count($doc/Model/Class/Name[fn:matches(., "[ ]")])};
Type of Measurement		Objective
Unit of Measurement		Model
Scope		Class diagram
Scale (Type)		$0 \leq$ ANIS (Absolute)
Interpretation		The closer to 0, the better.

Messung 34: Attribute Names Include Spaces

Objective Metric		
Name		Attributes of Type String
Acronym		AoTS
Measurement-method	Informal Definition	Number of attributes that are of type string.
	Formal Definition	declare function local:AoTS()as xs:double {fn:count ($doc/Model/Class/Attribute/Type[@idref = $doc/ Model/DataType[Name = "string"]/data(@id)])= 0};
Type of Measurement		Objective
Unit of Measurement		Model
Scope		Class diagram
Scale (Type)		$0 \leq$ AoTS (Absolute)
Interpretation		The closer to 0, the better. Other data types than String are not supported by AML.

Messung 35: Attributes of Type String

Objective Metric		
Name		Untyped Attributes
Acronym		UnAtt
Measurement-method	Informal Definition	Number of attributes without type declaration. (cp. OCL-Measurement in Tab. 32 on p. 280)
	Formal Definition	declare function local:UnAtt()as xs:double \{fn:count ($doc/Model/Class/Attribute/Type[@idref = $doc/ Model/DataType[Name = "undefined"]/data(@id)])\};
Type of Measurement		Objective
Unit of Measurement		Model
Scope		Class Diagram
Scale (Type)		$0 \leq$ UnAtt (Absolute)
Interpretation		The closer to 0, the better.

Messung 36: Untyped Attributes

Objective Metric		
Name		Cardinalities for Associations
Acronym		CAssoc
Measurement-method	Informal Definition	Number of association ends without cardinalities.
	Formal Definition	declare function local:CAssoc()as xs:double {fn:count($doc/Model/Class/AssociationEnd/ MultiplicityMinimum[string−length(.)=0])};
Type of Measurement		Objective
Unit of Measurement		Model
Scope		Class diagram
Scale (Type)		$0 \leq$ CAssoc (Absolute)
Interpretation		The closer to 0, the better.

Messung 37: Cardinalities for Associations

Objective Metric		
Name		Class Names Begin Small
Acronym		CNBS
Measurement-method	Informal Definition	Number of class names that do not start with a lower case letter.
	Formal Definition	declare function local:CNBS()as xs:double {fn:count($doc/Model/Class/Name[fn:matches(., "^[^A−Z]")])};
Type of Measurement		Objective
Unit of Measurement		Model
Scope		Class diagram
Scale (Type)		$0 \leq$ CNBS (Absolute)
Interpretation		The closer to 0, the better.

Messung 38: Class Names Begin Small

Objective Metric		
Name		Class Names Include Spaces
Acronym		CNIS
Measurement-method	Informal Definition	Number of class names that do include spaces in name.
	Formal Definition	declare function local:CNIS()as xs:double {fn:count($doc/Model/Class/Name[fn:matches(., "[ ]")])};
Type of Measurement		Objective
Unit of Measurement		Model
Scope		Class diagram
Scale (Type)		$0 \leq$ CNIS (Absolute)
Interpretation		The closer to 0, the better.

Messung 39: Class Names Include Spaces

Objective Metric		
Name		Correct Instantiation of Associations
Acronym		CIA
Measurement-method	Informal Definition	Number of links that are not an instance of any association of the according class diagram.
	Formal Definition	declare function local:CIA() as xs:double {fn:count($doc/Model/AssociationInstance[fn:**not**(./Association/ @idref = $doc/Model/Association/@id)])};
Type of Measurement		Objective
Unit of Measurement		Model
Scope		Class diagram, Object diagram
Scale (Type)		$0 \leq$ CIA (Absolute)
Interpretation		A value higher than 0 indicates inconsistencies.

Messung 40: Correct Instantiation of Associations

Objective Metric		
Name		Correct Instantiation of Classes
Acronym		CIC
Measurement-method	Informal Definition	Number of objects that are not an instance of any class of the according class diagram.
	Formal Definition	declare function local:CIC() as xs:double {fn:count ($doc/Model/Object[fn:**not**(./Type/@idref = $doc/ Model/Class/@id)])};
Type of Measurement		Objective
Unit of Measurement		Model
Scope		Class diagram, Object diagram
Scale (Type)		$0 \leq$ CIC (Absolute)
Interpretation		A value higher than 0 indicates inconsistencies.

Messung 41: Correct Instantiation of Classes

Objective Metric		
Name		Correct Events for Operations
Acronym		CEOp
Measurement-method	Informal Definition	Number of operation events in a statechart that are not defined in the class of the according class diagram.
	Formal Definition	declare function local:CEOp()as xs:double {fn:count ($doc/Model/StateMachine[./Class]/Transitions/ Transition/Trigger/CallEvent[fn:**not**(./Operation/ @idref = $doc/Model/Class/Operation/@id)])};
Type of Measurement		Objective
Unit of Measurement		Statechart
Scope		Class diagram, Statechart
Scale (Type)		$0 \leq$ CEOp (Absolute)
Interpretation		A value higher than 0 indicates inconsistencies.

Messung 42: Correct Events for Operations

Objective Metric		
Name		Correct Events for Signals
Acronym		CES
Measurement-method	Informal Definition	Number of signal events in a statechart that are not defined in the class of the according class diagram.
	Formal Definition	declare function local:CES()as xs:double {fn:count ($doc/Model/StateMachine/Transitions/Transition/ Trigger/SignalEvent[fn:**not**(./Signal/@idref = $doc/ Model/SignalClass/@id)])};
Type of Measurement		Objective
Unit of Measurement		Statechart
Scope		Class diagram, Statechart
Scale (Type)		$0 \leq$ CES (Absolute)
Interpretation		A value higher than 0 indicates inconsistencies.

Messung 43: Correct Events for Signals

Objective Metric		
Name		Empty Statecharts
Acronym		EmpSt
Measurement-method	Informal Definition	Number of statecharts stereotyped by test purpose without any state.
	Formal Definition	declare function local:EmpSt()as xs:double {fn:count ($doc/Model/StateMachine[fn:**not**(./Class)][fn:data(./ Stereotype)= "test purpose"][fn:**not**(.//SimpleState)]) };
Type of Measurement		Objective
Unit of Measurement		Model
Scope		Statechart
Scale (Type)		$0 \leq$ EmpSt (Absolute)
Interpretation		The closer to 0, the better.

Messung 44: Empty Statecharts

Objective Metric		
Name		Incoming and Outgoing Transitions
Acronym		IOT
Measurement-method	Informal Definition	Number of states excluding start and end states have not at least one incoming and one outgoing transition.
	Formal Definition	declare function local:IOT()as xs:double {fn: count($doc/Model/StateMachine//SimpleState[fn:count(IncommingTransitions)=0 **or** fn:count(OutgoingTransitions)=0])};
Type of Measurement		Objective
Unit of Measurement		Statechart
Scope		Statechart
Scale (Type)		$0 \leq$ IOT (Absolute)
Interpretation		The closer to 0, the better.

Messung 45: Incoming and Outgoing Transitions

Objective Metric		
Name		Initial Object Diagram
Acronym		IniOD
Measurement-method	Informal Definition	Does exactly one object diagram stereotyped with Initial describe the start configuration of the SUT?
	Formal Definition	declare function local:IniOD()as xs:boolean {(fn: count($doc/Model/ObjectDiagram[Stereotype="initial "])=1)};
Type of Measurement		Objective
Unit of Measurement		Model
Scope		Object diagram
Scale (Type)		IniOD $\in$ { true, false } (Nominal)
Interpretation		The value FALSE is bad and indicates a problem.

Messung 46: Initial Object Diagram

Objective Metric		
Name		Interface Coverage
Acronym		IC
Measurement-method	Informal Definition	Divide number of invoked methods in test cases (methods attached to transitions in test purpose statemachines) by the number of methods in the class diagram.
	Formal Definition	declare function local:IC() as xs:double {**if** (local:AO()= 0) **then** 0 **else** local:AaO()div local:AO()};
Type of Measurement		Objective
Unit of Measurement		Model
Scope		Class diagram, Statechart
Scale (Type)		$0 \leq$ IC (Ratio)
Interpretation		Values equal or higher than 1 are good.

Messung 47: Interface Coverage

Objective Metric		
Name		Names or Roles for Associations
Acronym		NRAssoc
Measurement-method	Informal Definition	Number of associations without names or roles.
	Formal Definition	declare function local:NRAssoc()as xs:double {fn:count($doc/Model/Association[./Name/string−length(.)=0 **and** ./AssociationEnd/@idref = $doc/Model/Class/AssociationEnd[./Name/string−length(.)=0]/@id])};
Type of Measurement		Objective
Unit of Measurement		Model
Scope		Class diagram
Scale (Type)		$0 \leq$ NRAssoc (Absolute)
Interpretation		The closer to 0, the better.

Messung 48: Names or Roles for Associations

Objective Metric		
Name		Non Usable Objects
Acronym		NUO
Measurement-method	Informal Definition	Number of objects that are not constructed on system level but rather above class level.
	Formal Definition	declare function local:NUO()as xs:double {fn:count($doc/Model/ObjectDiagram/Object[fn:**not**(@idref = $doc/Model/Object/@id)])};
Type of Measurement		Objective
Unit of Measurement		Model
Scope		Class diagram, Object diagram
Scale (Type)		$0 \leq$ NUO (Absolute)
Interpretation		The closer to 0, the better.

Messung 49: Non Usable Objects

Objective Metric		
Name		Number of End States
Acronym		NESt
Measurement-method	Informal Definition	Does only one end state per class exist?
	Formal Definition	declare function local:NESt()as xs:boolean {(fn:count ($doc/Model/StateMachine[./Class **and** fn:count(.// Final)>1]//Final)= 0)};
Type of Measurement		Objective
Unit of Measurement		Model
Scope		Statechart
Scale (Type)		NESt $\in$ { true, false } (Nominal)
Interpretation		The value FALSE is bad and indicates a problem.

Messung 50: Number of End States

Objective Metric		
Name		Object Names Begin Small
Acronym		ONBS
Measurement-method	Informal Definition	Number of object names that do not start with a lower case letter.
	Formal Definition	declare function local:ONBS()as xs:double {fn:count($doc/Model/Object/Name[fn:matches(., "^[^a−z]")])};
Type of Measurement		Objective
Unit of Measurement		Model
Scope		Object diagram
Scale (Type)		$0 \leq$ ONBS (Absolute)
Interpretation		The closer to 0, the better.

Messung 51: Object Names Begin Small

Objective Metric		
Name		Object Names Include Spaces
Acronym		ONIS
Measurement-method	Informal Definition	Number of object names that do include spaces in name.
	Formal Definition	declare function local:ONIS()as xs:double {fn:count($doc/Model/Object/Name[fn:matches(., "[ ]")])};
Type of Measurement		Objective
Unit of Measurement		Model
Scope		Object diagram
Scale (Type)		$0 \leq$ ONIS (Absolute)
Interpretation		The closer to 0, the better.

Messung 52: Object Names Include Spaces

Objective Metric		
Name		Operation Names Begin Small
Acronym		OpNBS
Measurement-method	Informal Definition	Number of operation names that do not start with a lower case letter.
	Formal Definition	declare function local:OpNBS()as xs:double {fn:count ($doc/Model/Class/Operation/Name[fn:matches(., "^[^a−z]")])};
Type of Measurement		Objective
Unit of Measurement		Model
Scope		Class diagram
Scale (Type)		$0 \leq$ OpNBS (Absolute)
Interpretation		The closer to 0, the better.

Messung 53: Operation Names Begin Small

Objective Metric		
Name		Operation Names Include Spaces
Acronym		OpNIS
Measurement-method	Informal Definition	Number of operation names that do include spaces in name.
	Formal Definition	declare function local:OpNIS()as xs:double {fn:count($doc/Model/Class/Operation/Name[fn:matches(., "[ ]")])};
Type of Measurement		Objective
Unit of Measurement		Model
Scope		Class diagram
Scale (Type)		$0 \leq$ OpNIS (Absolute)
Interpretation		The closer to 0, the better.

Messung 54: Operation Names Include Spaces

Objective Metric		
Name		Placement of IF-Code (SUT)
Acronym		PIFSUT
Measurement-method	Informal Definition	Mixing ration of IF-Code in model elements and IF-Code in comments in the SUT specification.
	Formal Definition	declare function local:PIFSUT()as xs:double {**if** (local:PvIFiKS()= 0) **then** 0 **else** local:PvIFiES()div local:PvIFiKS()};
Type of Measurement		Objective
Unit of Measurement		Model
Scope		Statechart
Scale (Type)		$0 \leq$ PIFSUT ≤ 1 (Ratio)
Interpretation		Values close to 1 are bad. Indicates that both notations are strongly mixed.

Messung 55: Placement of IF-Code (SUT)

Objective Metric		
Name		Placement of IF-Code (Test Cases)
Acronym		PIFTC
Measurement-method	Informal Definition	Mixing ration of IF-Code in model elements and IF-Code in comments in the test case specification.
	Formal Definition	declare function local:PIFTC()as xs:double {**if** (local :PvIFiKTG()= 0)**then** 0 **else** local:PvIFiETG()div local:PvIFiKTG()};
Type of Measurement		Objective
Unit of Measurement		Model
Scope		Statechart
Scale (Type)		$0 \leq$ PIFTC ≤ 1 (Ratio)
Interpretation		Values close to 1 are bad. Indicates that both notations are strongly mixed.

Messung 56: Placement of IF-Code (Test Cases)

Objective Metric		
Name		Ratio Observable and Controllable Attributes
Acronym		ROCA
Measurement-method	Informal Definition	Divide the number of attributes that are observable or controllable by the number of all attributes that are either observable or controllable.
	Formal Definition	declare function local:ROCA()as xs:double {**if** (local:BxKA()= 0)**then** 0 **else** local:BuKA()div local:BxKA()};
Type of Measurement		Objective
Unit of Measurement		Model
Scope		Class diagram
Scale (Type)		$0 \leq$ ROCA ≤ 1 (Ratio)
Interpretation		A value close to 1 is bad.

Messung 57: Ratio Observable and Controllable Attributes

Objective Metric		
Name		Ratio Observable and Controllable Operations
Acronym		ROCOp
Measurement-method	Informal Definition	Divide the number of operations that are observable or controllable by the number of all operations that are either observable or controllable.
	Formal Definition	declare function local:ROCOp()as xs:double {**if** (local:BxKA()= 0)**then** 0 **else** local:BuKA()div local:BxKA()};
Type of Measurement		Objective
Unit of Measurement		Model
Scope		Class diagram
Scale (Type)		$0 \leq$ ROCOp ≤ 1 (Ratio)
Interpretation		A value close to 1 is bad.

Messung 58: Ratio Observable and Controllable Operations

Objective Metric		
Name		Ratio Testable Attributes
Acronym		RTA
Measurement-method	Informal Definition	Divide the number of attributes that are observable or controllable by the number of all attributes.
	Formal Definition	declare function local:RTA()as xs:double {**if** (local:AA()= 0) **then** 0 **else** local:BoKA()div local:AA()};
Type of Measurement		Objective
Unit of Measurement		Model
Scope		Class diagram
Scale (Type)		$0 \leq$ RTA ≤ 1 (Ratio)
Interpretation		A value close to 0 may indicate that many attributes are in fact not required for the test purpose.

Messung 59: Ratio Testable Attributes

Objective Metric		
Name		Ratio Testable Operations
Acronym		RTO
Measurement-method	Informal Definition	Divide the number of operations that are observable or controllable by the number of all operations.
	Formal Definition	declare function local:RTO()as xs:double {**if** (local:AO()= 0) **then** 0 **else** local:BoKO()div local:AO()};
Type of Measurement		Objective
Unit of Measurement		Model
Scope		Class diagram
Scale (Type)		$0 \leq \text{RTO} \leq 1$ (Ratio)
Interpretation		A value close to 0 may indicate that many operations are in fact not required for the test purpose.

Messung 60: Ratio Testable Operations

Objective Metric		
Name		Stereotypes for Object Diagrams
Acronym		SfOD
Measurement-method	Informal Definition	Number of object diagrams that are not stereotyped by start, end exclude or include.
	Formal Definition	declare function local:SfOD()as xs:double {fn:count ($doc/Model/ObjectDiagram[fn:**not**(data(Stereotype) = "initial")][fn:**not**(fn:data(Stereotype)= "start" **or** fn :data(Stereotype) = "end" **or** fn:data(Stereotype)= " exclude" **or** fn:data(Stereotype) = "include")])};
Type of Measurement		Objective
Unit of Measurement		Model
Scope		Object diagram
Scale (Type)		$0 \leq$ SfOD (Absolute)
Interpretation		The closer to 0, the better.

Messung 61: Stereotypes for Object Diagrams

Objective Metric		
Name		Start States
Acronym		StSt
Measurement-method	Informal Definition	Does exactly one start state exist per class?
	Formal Definition	declare function local:StSt() as xs:boolean {fn: count($doc/Model/StateMachine[./Class][fn:count(./ SequentialCompositeState/Initial)>1])=0};
Type of Measurement		Objective
Unit of Measurement		Model
Scope		Statechart
Scale (Type)		StSt $\in$ { true, false } (Nominal)
Interpretation		The value FALSE is bad and indicates a problem.

Messung 62: Start States

Objective Metric		
Name		Statecharts for Classes
Acronym		SC
Measurement-method	Informal Definition	Number of classes without at least one statechart.
	Formal Definition	declare function local:SC() as xs:double {fn:count($doc/Model/Class[fn:**not**(./StateMachine)])};
Type of Measurement		Objective
Unit of Measurement		Model
Scope		Class diagram, Statechart
Scale (Type)		$0 \leq$ SC (Absolute)
Interpretation		A value higher than 0 indicates missing behavior.

Messung 63: Statecharts for Classes

Objective Metric		
Name		Tagged Values
Acronym		TV
Measurement-method	Informal Definition	Number of states in statecharts stereotyped by test purpose that have a different tagged value from the following values: init, accept, reject, start, end.
	Formal Definition	declare function local:TV()as xs:double {fn:count($doc/Model/StateMachine[fn:**not**(./Class)][fn:data(./ Stereotype)= "test purpose"]//SimpleState/∗/self::∗[**not**(∗)][string(node−name(.))!= "init" **and** string(node−name(.))!= "accept" **and** string(node−name(.)) != "reject" **and** string(node−name(.))!= "start" **and** string(node−name(.))!= "end" **and** string(node−name (.))!= "Name"])};
Type of Measurement		Objective
Unit of Measurement		Model
Scope		Statechart
Scale (Type)		$0 \leq$ TV (Absolute)
Interpretation		The closer to 0, the better.

Messung 64: Tagged Values

Objective Metric		
Name		Test Purpose Stereotyp
Acronym		TPS
Measurement-method	Informal Definition	Number of statecharts that are not stereotyped by test purpose.
	Formal Definition	declare function local:TPS()as xs:double {fn:count ($doc/Model/StateMachine[fn:**not**(./Class)][fn:**not**(Stereotype)**or** fn:data(Stereotype)!= "test purpose"])};
Type of Measurement		Objective
Unit of Measurement		Model
Scope		Statechart
Scale (Type)		$0 \leq$ TPS (Absolute)
Interpretation		The closer to 0, the better.

Messung 65: Test Purpose Stereotyp

Objective Metric		
Name		Transitions of End States
Acronym		TESt
Measurement-method	Informal Definition	Does each end state has only incoming transitions?
	Formal Definition	declare function local:TESt()as xs:boolean {fn:count($doc/Model/StateMachine//Final/OutgoingTransitions)= 0};
Type of Measurement		Objective
Unit of Measurement		Model
Scope		Statechart
Scale (Type)		TESt $\in$ { true, false } (Nominal)
Interpretation		The value FALSE is bad and indicates a problem.

Messung 66: Transitions of End States

Objective Metric		
Name		Transitions to Operations
Acronym		NT2NOps
Measurement-method	Informal Definition	Divide the number of transitions in a statechart by the number of available operations of a class (cp. OCL-Measurement in Tab. 30 on p. 279)
	Formal Definition	declare function local:NT2NOps()as xs:double \{**if** (local:AO()= 0) **then** 0 **else** local:AT()div local:AO() \};
Type of Measurement		Objective
Unit of Measurement		Model
Scope		Class Diagram, Statechart
Scale (Type)		0 ≤ NT2NOps (Ratio)
Interpretation		A value higher than 2 is poor.

Messung 67: Transitions to Operations

Objective Metric		
Name		Transitions of Start States
Acronym		ToStSt
Measurement-method	Informal Definition	Does the start state only have outgoing transitions?
	Formal Definition	declare function local:ToStSt() as xs:boolean {fn:count($doc/Model/StateMachine//Initial/IncommingTransitions)= 0};
Type of Measurement		Objective
Unit of Measurement		Model
Scope		Statechart
Scale (Type)		ToStSt ∈ { true, false } (Nominal)
Interpretation		The value FALSE is bad and indicates a problem.

Messung 68: Transitions of Start States

A.2 Subjektive Metriken

Subjective Metric		
Name		Attribute Variables for Guards
Acronym		AVfG
Measurement-method	Informal Definition	How many guards operate on variables that are not defined as attributes in in the model?
Type of Measurement		Subjective
Unit of Measurement		Model
Scope		Class diagram, Statechart
Scale (Type)		$0 \leq$ AVfG (Absolute)
Interpretation		The closer to 0, the better.

Messung 69: Attribute Variables for Guards

Subjective Metric		
Name		Available Diagram Types
Acronym		ADT
Measurement-method	Informal Definition	How many diagrams in the model are not of the type class diagram, statechart or object diagram?
Type of Measurement		Subjective
Unit of Measurement		Model
Scope		Model
Scale (Type)		$0 \leq$ ADT (Absolute)
Interpretation		A value higher than 0 indicates model statements not processable for test case generation.

Messung 70: Available Diagram Types

Subjective Metric		
Name		Available Operations
Acronym		AvOp
Measurement-method	Informal Definition	How many transitions have action code that invokes operations that are not available in the model?
Type of Measurement		Subjective
Unit of Measurement		Model
Scope		Statechart
Scale (Type)		$0 \leq$ AvOp (Absolute)
Interpretation		The closer to 0, the better.

Messung 71: Available Operations

Subjective Metric		
Name		Invariant in State Names (SUT)
Acronym		IiSNSUT
Measurement-method	Informal Definition	How many state names of the SUT repeat the invariant?
Type of Measurement		Subjective
Unit of Measurement		Model
Scope		Zustandsdiagramme
Scale (Type)		$0 \leq$ IiSNSUT (Absolute)
Interpretation		The closer to 0, the better.

Messung 72: Invariant in State Names (SUT)

Subjective Metric		
Name		Invariant in State Names (Test Cases)
Acronym		IiSNTC
Measurement-method	Informal Definition	How many state names of the test case specification repeat the invariant?
Type of Measurement		Subjective
Unit of Measurement		Model
Scope		Statechart
Scale (Type)		$0 \leq$ IiSNTC (Absolute)
Interpretation		The closer to 0, the better.

Messung 73: Invariant in State Names (Test Cases)

Subjective Metric		
Name		Law of Demeter Violations
Acronym		LoDV
Measurement-method	Informal Definition	How many method invocations definitely violate the Law of Demeter (LoD)? The Law of Demeter (LoD) for objects requires that a method M of an object O may only invoke the methods of the following kinds of objects: • O itself • M's parameters • any objects created/instantiated within M • associated objects (as association or attribute)
Type of Measurement		Subjective
Unit of Measurement		Model
Scope		Class Diagram; Sequence Diagram
Scale (Type)		$0 \leq$ LoDV (Absolute)
Interpretation		The closer to 0, the better.

Messung 74: Law of Demeter Violations

Subjective Metric		
Name		Mealy States (SUT)
Acronym		MSSUT
Measurement-method	Informal Definition	How many mealy-type states are in the SUT? Mealy-type states have a passive name.
Type of Measurement		Subjective
Unit of Measurement		Model
Scope		Statechart
Scale (Type)		$0 \leq$ MSSUT (Absolute)
Interpretation		The closer to 0, the better.

Messung 75: Mealy States (SUT)

Subjective Metric		
Name		Mealy States (Test Cases)
Acronym		MSTC
Measurement-method	Informal Definition	How many mealy-type states are in the test case specification? Mealy-type states have a passive name.
Type of Measurement		Subjective
Unit of Measurement		Model
Scope		Statechart
Scale (Type)		$0 \leq$ MSTC (Absolute)
Interpretation		The closer to 0, the better.

Messung 76: Mealy States (Test Cases)

Subjective Metric		
Name		Meaningful State Names (Test Cases)
Acronym		MSNTC
Measurement-method	Informal Definition	How many states of the test case specification do not have a meaningful name in the context of the application domain?
Type of Measurement		Subjective
Unit of Measurement		Model
Scope		Statechart
Scale (Type)		$0 \leq$ MSNTC (Absolute)
Interpretation		The closer to 0, the better.

Messung 77: Meaningful State Names (Test Cases)

Subjective Metric		
Name		Meaningful State Names (SUT)
Acronym		MSNSUT
Measurement-method	Informal Definition	How many states of the SUT do not have a meaningful name in the context of the application?
Type of Measurement		Subjective
Unit of Measurement		Model
Scope		Statechart
Scale (Type)		$0 \leq$ MSNSUT (Absolute)
Interpretation		The closer to 0, the better.

Messung 78: Meaningful State Names (SUT)

Subjective Metric		
Name		Non Deterministic Guards
Acronym		NDG
Measurement-method	Informal Definition	For how many states with more than one outgoing transition can it happen that the assignments of the guards are not exclusively fulfilled so that more than one transition can fire?
Type of Measurement		Subjective
Unit of Measurement		Model
Scope		Statechart
Scale (Type)		$0 \leq$ NDG (Absolute)
Interpretation		The closer to 0, the better.

Messung 79: Non Deterministic Guards

Subjective Metric		
Name		Non Reachable States
Acronym		NRS
Measurement-method	Informal Definition	How many states are not reachable from the start state?
Type of Measurement		Subjective
Unit of Measurement		Statechart
Scope		Statechart
Scale (Type)		$0 \leq$ NRS (Absolute)
Interpretation		A value higher than 0 indicates unnecessary states.

Messung 80: Non Reachable States

Subjective Metric		
Name		Non Terminating States
Acronym		NTS
Measurement-method	Informal Definition	How many states do not have a path to an end state?
Type of Measurement		Subjective
Unit of Measurement		Statechart
Scope		Statechart
Scale (Type)		$0 \leq$ NTS (Absolute)
Interpretation		A value higher than 0 indicates unnecessary states.

Messung 81: Non Terminating States

Subjective Metric		
Name		Usable Operations
Acronym		UsOp
Measurement-method	Informal Definition	In how many transitions are operations invoked that are not controllable or observable?
Type of Measurement		Subjective
Unit of Measurement		Model
Scope		Statechart
Scale (Type)		$0 \leq$ UsOp (Absolute)
Interpretation		The closer to 0, the better.

Messung 82: Usable Operations

Subjective Metric		
Name		Use Case without SD
Acronym		UCnSD
Measurement-method	Informal Definition	How many Use Cases are not illustrated by at least one sequence diagram?
Type of Measurement		Subjective
Unit of Measurement		Model
Scope		Sequence Diagram, Use Case Diagram of Software Requirements Specification
Scale (Type)		$0 \leq$ UCnSD (Absolute)
Interpretation		The closer to 0, the better.

Messung 83: Use Case without SD

Subjective Metric		
Name		Used Objects
Acronym		UOb
Measurement-method	Informal Definition	How many transitions do operate on objects that are not specified in the initial or start object diagram?
Type of Measurement		Subjective
Unit of Measurement		Model
Scope		Object diagram, Statechart
Scale (Type)		$0 \leq$ UOb (Absolute)
Interpretation		The closer to 0, the better.

Messung 84: Used Objects

Subjective Metric		
Name		Variability of Attributes
Acronym		ECVarA
Measurement-method	Informal Definition	What is the standard deviation of attributes of all classes stereotyped with «Entity»? The standard deviation σ is calculated as follows: $\sigma = \sqrt{\frac{1}{N-1}\sum_{i=1}^{N}(x_i - \bar{x})}$ with N = Number of entity-classes in the software model, x_i the number of available attributes of the i-th entity-class, and $\bar{x}$ the mean of the values.
Type of Measurement		Subjective
Unit of Measurement		Model
Scope		Class Diagram
Scale (Type)		$0 \leq$ ECVarA (Absolute)
Interpretation		The closer to 0, the better.

Messung 85: Variability of Attributes

Subjective Metric		
Name		Variability of Operations
Acronym		CCVarOp
Measurement-method	Informal Definition	What is the standard deviation of operations of all classes stereotyped with «Control»? The standard deviation σ is calculated as follows: $\sigma = \sqrt{\frac{1}{N-1}\sum_{i=1}^{N}(x_i - \bar{x})}$ with N = Number of control-classes in the software model, x_i the number of available operations of the i-th control-class, and $\bar{x}$ the mean of the values.
Type of Measurement		Subjective
Unit of Measurement		Model
Scope		Class Diagram
Scale (Type)		$0 \leq$ CCVarOp (Absolute)
Interpretation		The closer to 0, the better.

Messung 86: Variability of Operations

Subjective Metric		
Name		Variables for Guards
Acronym		VfG
Measurement-method	Informal Definition	How many guards include variables that are not defined in the according class?
Type of Measurement		Subjective
Unit of Measurement		Model
Scope		Class diagram, Statechart
Scale (Type)		$0 \leq$ VfG (Absolute)
Interpretation		A value higher than 0 indicates inconsistencies.

Messung 87: Variables for Guards

Subjective Metric		
Name		Waiting States (SUT)
Acronym		WSSUT
Measurement-method	Informal Definition	How many states of the SUT are named waiting or the like?
Type of Measurement		Subjective
Unit of Measurement		Model
Scope		Statechart
Scale (Type)		$0 \leq$ WSSUT (Absolute)
Interpretation		The closer to 0, the better.

Messung 88: Waiting States (SUT)

Subjective Metric		
Name		Waiting States (Test Cases)
Acronym		WSTC
Measurement-method	Informal Definition	How many states of the test case specification are named waiting or the like?
Type of Measurement		Subjective
Unit of Measurement		Model
Scope		Statechart
Scale (Type)		$0 \leq$ WSTC (Absolute)
Interpretation		The closer to 0, the better.

Messung 89: Waiting States (Test Cases)

A.3 Indikatoren

<table>
<tr><th colspan="4">Indicator</th></tr>
<tr><td>Name</td><td colspan="3">Depth of Inheritance Tree Indicator</td></tr>
<tr><td>Acronym</td><td colspan="3">I-DIT</td></tr>
<tr><td>Input Metric</td><td colspan="3">Depth of Inheritance Tree (DIT) in Tab. 9 on p. 264</td></tr>
<tr><td>Analysis Model</td><td></td><td>
<table>
<tr><th>Analysis</th><th>I-DIT</th></tr>
<tr><td>DIT > 5</td><td>true</td></tr>
<tr><td>else</td><td>false</td></tr>
</table>
</td><td></td></tr>
<tr><td>Unit of Measurement</td><td colspan="3">Class</td></tr>
<tr><td>Scope</td><td colspan="3">Class Diagram</td></tr>
<tr><td>Scale (Type)</td><td colspan="3">I-DIT $\in$ { true, false } (Nominal)</td></tr>
<tr><td>Interpretation</td><td colspan="3">$I\text{-}DIT = true$ may indicate a great design complexity.</td></tr>
</table>

Messung 90: Depth of Inheritance Tree Indicator

Indicator	
Name	Indicating Hidden Concurrency
Acronym	I-HC
Input Metric	Lack of Cohesion of Methods (LCOM4) in Tab. 13 on p. 268 and Top Level Parallelizations (NTLP) in Tab. 28 on p. 278
Analysis Model	**Analysis** / **I-HC** LCOM4 > NTLP / true else / false
Unit of Measurement	Class
Scope	Class Diagram, Z description, Statechart
Scale (Type)	I-HC ∈ { true, false } (Nominal)
Interpretation	*I-HC = true* may indicate the design problem hidden concurrency.

Messung 91: Indicating Hidden Concurrency

Indicator	
Name	Indicating Refused Bequest
Acronym	I-RB
Input Metric	Overridden Operations (OO) in Tab. 24 on p. 276
Analysis Model	**Analysis** / **I-RB** OO > 0 / true else / false
Unit of Measurement	Class
Scope	Class diagram
Scale (Type)	I-RB ∈ { true, false } (Nominal)
Interpretation	*I-RB = true* may indicate the design problem refused bequest.

Messung 92: Indicating Refused Bequest

Indicator	
Name	Indicating Relatedness among Methods
Acronym	I-CAMC
Input Metric	Relatedness among Methods (CAMC) in Tab. 26 on p. 277
Analysis Model	**Analysis** / **I-CAMC** CAMC ≥ 0.35 / true else / false
Unit of Measurement	Class
Scope	Class Diagram
Scale (Type)	I-CAMC $\in$ { true, false } (Nominal)
Interpretation	*I-CAMC = false* may indicate a non-cohesive class.

Messung 93: Indicating Relatedness among Methods

Indicator	
Name	Indicating Tight and Loose Class Cohesion
Acronym	I-TCC
Input Metric	Tight and Loose Class Cohesion (TCC) in Tab. 27 on p. 278
Analysis Model	**Analysis** / **I-TCC** TCC $= 1$ / maximally cohesive class $0.75 <$ TCC < 1 / class cohesion above average $0.5 \leq$ TCC < 0.75 / quite cohesive class TCC < 0.5 / non-cohesive class
Unit of Measurement	Class
Scope	Class diagram; Z description
Scale (Type)	I-TCC $\in$ { maximally cohesive class, class cohesion above average, quite cohesive class, non-cohesive class } (Ordinal)
Interpretation	*Non-cohesive class* may indicate the design problem too low cohesion.

Messung 94: Indicating Tight and Loose Class Cohesion

Indicator	
Name	Indicating Too Strong Coupling based Parameter List
Acronym	I-MaxPL
Input Metric	Parameter List (MaxPL) in Tab. 25 on p. 276
Analysis Model	**Analysis** / **I-MaxPL** MaxPL > 5 / true else / false
Unit of Measurement	Class
Scope	Class diagram; Z description
Scale (Type)	I-MaxPL ∈ { true, false } (Nominal)
Interpretation	*I-MaxPL = true* may indicate the design problem too strong coupling.

Messung 95: Indicating Too Strong Coupling based Parameter List

Indicator	
Name	Indicating Unnecessary Behavioral Complexity based on Border Crossings
Acronym	I-NBC
Input Metric	Border Crossings (NBC) in Tab. 5 on p. 262
Analysis Model	**Analysis** / **I-NBC** NBC > 5 / true else / false
Unit of Measurement	Class
Scope	Statechart
Scale (Type)	I-MaxNLCS ∈ { true, false } (Nominal)
Interpretation	*I-NBC = true* may indicate the design problem unnecessary behavioral complexity.

Messung 96: Indicating Unnecessary Behavioral Complexity based on Border Crossings

Indicator			
Name	Indicating Unnecessary Behavioral Complexity based on Nesting Level		
Acronym	I-MaxNLCS		
Input Metric	Nesting Level (MaxNLCS) in Tab. 21 on p. 274		
Analysis Model		**Analysis**	**I-MaxNLCS**
		UBC > 5	true
		else	false
Unit of Measurement	Class		
Scope	Statechart		
Scale (Type)	I-MaxNLCS $\in$ { true, false } (Nominal)		
Interpretation	*I-MaxNLCS = true* may indicate the design problem unnecessary behavioral complexity.		

Messung 97: Indicating Unnecessary Behavioral Complexity based on Nesting Level

Indicator			
Name	Lack of Cohesion of Methods Indicator		
Acronym	I-LCOM4		
Input Metric	Lack of Cohesion of Methods (LCOM4) in Tab. 13 on p. 268		
Analysis Model		**Analysis**	**I-LCOM4**
		LCOM4 $= 1$	cohesive class
		LCOM4 $= 0$	lazy class
		else	non-cohesive class
Unit of Measurement	Class		
Scope	Class Diagram, Z description		
Scale (Type)	I-LCOM4 $\in$ { cohesive class, lazy class, non-cohesive class } (Nominal)		
Interpretation	A *cohesive class* is a good class. A *non-cohesive class* should be split into smaller classes. A *lazy class* has no methods and is a bad class.		

Messung 98: Lack of Cohesion of Methods Indicator

Indicator			
Name	Large Class Indicator		
Acronym	I-LC		
Input Metric	Defined Operations (DO) in Tab. 11 on p. 266 and Total Defined Operations (TDO) in Tab. 29 on p. 279		
Analysis Model		**Analysis**	**I-LC**
		DO / TDO > 3	true
		else	false
Unit of Measurement	Class		
Scope	Class Diagram		
Scale (Type)	I-LC $\in$ { true, false } (Nominal)		
Interpretation	*I-LC = true* may indicate a large class.		

Messung 99: Large Class Indicator

Indicator			
Name	Operations Inheritance Factor Indicator		
Acronym	I-OIF		
Input Metric	Operations Inheritance Factor (OIF) in Tab. 23 on p. 275		
Analysis Model		**Analysis**	**I-SC**
		OIF > 0.8	too high
		$0.8 \geq$ OIF ≥ 0.2	acceptable
		else	too low
Unit of Measurement	Model		
Scope	Class Diagram		
Scale (Type)	I-OIF $\in$ { acceptable, too-high, too-low} (Nominal)		
Interpretation	*I-OIF = too high* may indicate either superfluous inheritance or too wide member scopes. A *too low value* may indicate lack of inheritance or heavy use of overrides.		

Messung 100: Operations Inheritance Factor Indicator

Indicator			
Name	Small Class Indicator		
Acronym	I-SC		
Input Metric	Defined Operations (DO) in Tab. 11 on p. 266 and Total Defined Operations (TDO) in Tab. 29 on p. 279		
Analysis Model		**Analysis**	**I-SC**
		DO / TDO < 0.33	true
		else	false
Unit of Measurement	Class		
Scope	Class Diagram		
Scale (Type)	I-SC $\in$ { true, false } (Nominal)		
Interpretation	*I-SC = true* may indicate a small class.		

Messung 101: Small Class Indicator

Indicator			
Name	Transitions to Operations Indicator		
Acronym	I-NT2NOps		
Input Metric	Transitions to Operations (NT2NOps) in Tab. 30 on p. 279		
Analysis Model		**Analysis**	**I-NT2NOps**
		NT2NOps > 2	true
		else	false
Unit of Measurement	Class		
Scope	Class Diagram, Statechart		
Scale (Type)	I-NT2NOps $\in$ { true, false } (Nominal)		
Interpretation	*I-NT2NOps = true* may indicate the design problem hidden concurrency and unnecessary behavioral complexity.		

Messung 102: Transitions to Operations Indicator

<table>
<tr><th colspan="3">Indicator</th></tr>
<tr><td>Name</td><td colspan="2">Unlimited Cardinalities Indicator</td></tr>
<tr><td>Acronym</td><td colspan="2">I-AuC</td></tr>
<tr><td>Input Metric</td><td colspan="2">Unlimited Cardinalities (AuC) in Tab. 3 on p. 261</td></tr>
<tr><td>Analysis Model</td><td colspan="2"><table>
<tr><th>Analysis</th><th>I-AuC</th></tr>
<tr><td>$1 \geq \text{AuC} > 0.9$</td><td>very high</td></tr>
<tr><td>$0.9 \geq \text{AuC} > 0.8$</td><td>high</td></tr>
<tr><td>$0.8 \geq \text{AuC} > 0.7$</td><td>middle</td></tr>
<tr><td>$0.7 \geq \text{AuC} > 0.5$</td><td>low</td></tr>
<tr><td>else</td><td>very low</td></tr>
</table></td></tr>
<tr><td>Unit of Measurement</td><td colspan="2">Model</td></tr>
<tr><td>Scope</td><td colspan="2">Class diagram</td></tr>
<tr><td>Scale (Type)</td><td colspan="2">I-AuC $\in$ { very high, high, middle, low, very low} (Ordinal)</td></tr>
<tr><td>Interpretation</td><td colspan="2">The Ratio for unlimited cardinalities is value of I-AuC. A low or very low value may indicate the design problem speculative generality</td></tr>
</table>

Messung 103: Unlimited Cardinalities Indicator

<table>
<tr><th colspan="3">Indicator</th></tr>
<tr><td>Name</td><td colspan="2">Unnamed Associations Indicator</td></tr>
<tr><td>Acronym</td><td colspan="2">I-UA2NA</td></tr>
<tr><td>Input Metric</td><td colspan="2">Unnamed Associations (UA2NA) in Tab. 31 on p. 280</td></tr>
<tr><td>Analysis Model</td><td colspan="2"><table>
<tr><th>Analysis</th><th>I-UA2NA</th></tr>
<tr><td>$1 \geq$ UA2NA > 0.5</td><td>too many</td></tr>
<tr><td>$0.5 \geq$ UA2NA > 0.3</td><td>many</td></tr>
<tr><td>$0.3 \geq$ UA2NA > 0.1</td><td>some</td></tr>
<tr><td>else</td><td>marginal</td></tr>
</table></td></tr>
<tr><td>Unit of Measurement</td><td colspan="2">Model</td></tr>
<tr><td>Scope</td><td colspan="2">Class diagram</td></tr>
<tr><td>Scale (Type)</td><td colspan="2">I-UA2NA $\in$ { too many, many, some, marginal } (Ordinal)</td></tr>
<tr><td>Interpretation</td><td colspan="2">There are value of I-UA2NA unnamed associations in the software model.</td></tr>
</table>

Messung 104: Unnamed Associations Indicator