Bewertung der Qualität objektorientierter Entwürfe

Von der Fakultät Informatik der Universität Stuttgart
zur Erlangung der Würde eines Doktors der
Naturwissenschaften (Dr. rer. nat.) genehmigte Abhandlung

Vorgelegt von

Ralf Reißing

aus Waiblingen

Hauptberichter:	Prof. Dr. rer. nat. J. Ludewig
Mitberichter:	Prof. Dr.-Ing. G. Snelting
Tag der mündlichen Prüfung:	15.08.2002

Institut für Informatik der Universität Stuttgart

2002

Danksagung

An dieser Stelle möchte ich allen danken, die zum Gelingen dieser Arbeit beigetragen haben. Der größte Dank gebührt meinem Doktorvater, Prof. Jochen Ludewig, dessen Anregungen und Kritik viel zur Arbeit beitragen haben. Prof. Gregor Snelting danke ich für die bereitwillige Übernahme des Zweitgutachtens.

Dank auch den Kollegen der Abteilung Software Engineering für ein gutes Arbeitsklima und er-/entmutigende Gespräche über haltbare/unhaltbare Zeitpläne zur Promotion. Christoph Schmider danke ich für die Implementierung des Werkzeugs MOOSE im Rahmen seiner Diplomarbeit. Eckart Mayer, Jan Böhm und Phillip Müller sei gedankt für ihre kritische Durchsicht der Rohfassung.

Zum Schluss ein Dank an alle, die sich mehr oder weniger regelmäßig nach dem Fortschritt der Arbeit erkundigt haben und immer dieselbe Auskunft bekamen: Dass sie noch nicht fertig sei. Diese Anteilnahme trug viel zur Motivation bei, die Arbeit abzuschließen.

Zusammenfassung

In der Software-Entwicklung spielt der Entwurf eine zentrale Rolle. Er beeinflusst die nachfolgenden Phasen Implementierung und Wartung nachhaltig. Weil Implementierung und Wartung zusammengenommen etwa zwei Drittel des Gesamtaufwands ausmachen, ist es sinnvoll, beim Entwurf eine hohe Qualität anzustreben. Da es schwierig ist, auf Anhieb einen guten Entwurf zu erstellen, ist eine Qualitätssicherung in Form einer Entwurfsbewertung nötig. Die Bewertung dient dazu, die Qualität festzustellen und Schwachstellen im Entwurf aufzudecken; sie kann aber auch zum Vergleich von Entwurfsalternativen verwendet werden.

Diese Arbeit beschäftigt sich mit der Bewertung des objektorientierten Entwurfs. Die Basis für die Entwurfsbewertung ist ein Qualitätsmodell. Weil die Qualitätsanforderungen von Projekt zu Projekt unterschiedlich sind, sollte das Qualitätsmodell an den vorhandenen Kontext angepasst sein. Daher wird ein allgemeines Qualitätsmodell namens Quality Model for Object-Oriented Design (QOOD) eingeführt, aus dem sich spezifische Qualitätsmodelle ableiten lassen, die an die konkreten Qualitätsanforderungen angepasst sind.

Um die Kosten für die Bewertung zu reduzieren, ist eine weitgehende Automatisierung nützlich. Dies macht aber eine formale Repräsentation des Entwurfs erforderlich. Da die Unified Modeling Language (UML) der Standard für die Notation objektorientierter Entwürfe darstellt und ausreichend formal ist, wurde ein Referenzmodell namens Object-Oriented Design Model (ODEM) entwickelt, das auf dem UML-Metamodell aufsetzt. Da UML-Modelle typischerweise nur bei der statischen Beschreibung des Entwurfs für eine Bewertung hinreichend vollständig sind, deckt ODEM nur den statischen Entwurf ab. ODEM dient gleichzeitig als Grundlage für die formale Definition von Metriken; es hat sich in Fallstudien mit Metriken aus der Literatur und bei der Definition der Metriken für QOOD bewährt.

Bei der Quantifizierung des Qualitätsmodells reichen automatisierbare, objektive Metriken nicht aus, weil sich wichtige semantische Entwurfseigenschaften (z. B. Zusammenhalt) damit unzureichend erfassen lassen. Daher werden zusätzlich subjektive Metriken eingesetzt, die von einem Entwickler aufgrund seines Eindrucks erhoben werden. Er wird dabei durch Fragebögen unterstützt, damit er nichts Wesentliches übersieht. Durch die Verwendung subjektiver Metriken kann die Bewertung nicht vollständig automatisiert werden; sie kann aber durch ein Werkzeug in großem Umfang unterstützt werden.

Für die Bewertung des wichtigsten Qualitätsfaktors Wartbarkeit hat sich die Einschränkung des Verfahrens auf statische Entwurfsinformation als unproblematisch erwiesen. In einer Fallstudie konnte nachgewiesen werden, dass die relative Wartbarkeit von Entwurfsalternativen mit Hilfe des Qualitätsmodells zutreffend vorhergesagt werden kann. Der erforderliche Zeitaufwand für eine Entwurfsbewertung ist vertretbar und kann durch Werkzeugunterstützung stark reduziert werden.

Abstract

Design plays a pivotal role in software development. It strongly influences the subsequent implementation and maintenance phases. Implementation and maintenance taken together take more than two thirds of the entire effort, therefore it makes good sense to strive for high design quality. As it is difficult to create a good design straightaway, quality assurance in the form of design assessment is needed. The assessment serves to determine the design quality and to uncover weaknesses in the design; it may also be used for comparing design alternatives.

This thesis focuses on object-oriented design. The foundation of the design assessment is a quality model. As quality requirements differ for different projects, the quality model should be adapted to the actual context. Therefore a general quality model called Quality Model for Object-Oriented Design (QOOD) is introduced, from which specific quality models can be derived that are adapted to the concrete quality requirements.

In order to reduce assessment cost, extensive automation is useful. That, however, requires a formal representation of the design. As the Unified Modeling Language (UML) is the standard for notating object-oriented designs and as it is sufficiently formal, a reference model called Object-Oriented Design Model (ODEM) was developed that is based on the UML metamodel. As UML models typically are sufficiently complete for assessment only in respect to static design information, ODEM is restricted to static design. ODEM is also used as a basis for the formal definition of metrics; it has proved effective in case studies with metrics from the literature as well as in defining metrics for QOOD.

When quantifying the quality model, automatable objective metrics are not sufficient, because they do not cover semantic aspects (like cohesion) well. Therefore also subjective metrics are used that are measured by a developer according to his impression. He is assisted by questionnaires, so as not to overlook important aspects. Because of the use of subjective measures, the assessment cannot be automated completely, but it can be supported by a tool to a large extent.

The restriction of the assessment to static design information does not impede the assessment of the most important quality factor maintainability. In a case study it was proved that the relative maintainability of design alternatives can be predicted correctly by the quality model. The effort necessary for design assessment is acceptable and can be reduced heavily by tool support.

Inhaltsverzeichnis

Danksagung ... **iii**
Zusammenfassung ... **v**
Abstract ... **vi**
Inhaltsverzeichnis ... **vii**
1 Einführung ... **1**
1.1 Motivation ... 1
1.2 Zielsetzung ... 2
1.3 Lösungsansatz ... 4
1.4 Übersicht ... 5
2 Modelle und Metriken ... **7**
2.1 Modelle ... 7
2.2 Metriken ... 9
3 Objektorientierung ... **15**
3.1 Begriffe ... 16
3.2 Unified Modeling Language ... 21
4 Objektorientierter Entwurf ... **23**
4.1 Was ist Entwurf? ... 23
4.2 Klassifikationen des Entwurfs ... 27
4.3 Muster und Rahmenwerke ... 30
4.4 Dokumentation des Entwurfs ... 34
4.5 Probleme des Entwurfs ... 35
5 Ein Referenzmodell für den objektorientierten Entwurf ... **43**
5.1 Grundlagen ... 43
5.2 Umfang ... 45
5.3 Kern ... 48
5.4 Erweiterungen ... 52
5.5 Formale Definition von Metriken ... 55

6 Softwarequalität ... 59
6.1 Qualität ... 59
6.2 Qualitätsmodelle ... 63
6.3 Qualitätssicherung ... 69
7 Entwurfsqualität ... 73
7.1 Ein Beispiel ... 73
7.2 Perspektiven der Entwurfsqualität ... 78
7.3 Entwurfsregeln ... 83
7.4 Beispiele für OOD-Qualitätsmodelle ... 88
7.5 Qualitätssicherung beim Entwurf ... 96
7.6 Entwurfsbewertung ... 97
8 Das allgemeine Qualitätsmodell ... 101
8.1 Vorüberlegungen ... 101
8.2 Aufbau des Modells ... 104
8.3 Wartbarkeit ... 105
8.4 Wiederverwendung ... 112
8.5 Wiederverwendbarkeit ... 113
8.6 Brauchbarkeit ... 114
8.7 Testbarkeit ... 115
8.8 Prüfbarkeit ... 116
8.9 Weitere mögliche Faktoren ... 116
9 Quantifizierung des Qualitätsmodells ... 117
9.1 Bewertungsverfahren ... 117
9.2 Objektive Metriken ... 120
9.3 Subjektive Metriken ... 125
9.4 Fragebögen ... 127
9.5 Gesamtbewertung ... 131
9.6 Ableitung spezifischer Modelle ... 132
10 Ein spezifisches Qualitätsmodell ... 135
10.1 Ableitung des Qualitätsmodells ... 135
10.2 Anwendung des Qualitätsmodells ... 140
10.3 Besonderheiten bei Mustern ... 148
11 Werkzeugunterstützung ... 151
11.1 Werkzeuge aus anderen Arbeiten ... 151
11.2 Selbst realisierte Werkzeuge ... 154
11.3 Ausblick: Ein ideales Werkzeug ... 159

12 Zusammenfassung und Ausblick ... 161
12.1 Zusammenfassung ... 161
12.2 Bewertung des Ansatzes ... 162
12.3 Vergleich mit anderen Arbeiten ... 164
12.4 Ausblick ... 166
12.5 Schlussbemerkung ... 168
Literatur ... 169
Akronyme ... 187
A Metriken für QOOD ... 189
A.1 Knappheit ... 189
A.2 Strukturiertheit ... 192
A.3 Entkopplung ... 193
A.4 Zusammenhalt ... 196
A.5 Einheitlichkeit ... 197
A.6 Dokumentierung ... 198
A.7 Verfolgbarkeit ... 198
A.8 Wartbarkeit ... 198
A.9 Theoretische Validierung ... 199
B Fragebögen für QOOD ... 203
B.1 Knappheit ... 204
B.2 Strukturiertheit ... 205
B.3 Entkopplung ... 206
B.4 Zusammenhalt ... 209
B.5 Einheitlichkeit ... 210
B.6 Dokumentierung ... 211
B.7 Verfolgbarkeit ... 213
B.8 Wartbarkeit ... 214
C Dokumente zum Softwarepraktikum ... 215
C.1 Aufgabenstellung ... 215
C.2 Anforderungen ... 220
C.3 Begriffslexikon ... 224

Kapitel 1

Einführung

Design is one of the most elusive yet fascinating topics in the software field. It is elusive because, no matter how thoroughly academics try to shape it into a teachable, testable, fact-based topic, it just doesn't fit. It is fascinating because design holds the key to the success of most software projects.
(Glass, 1999, S. 104)

Diese Arbeit beschäftigt sich mit der Frage, wie die Qualität eines objektorientierten Entwurfs bewertet werden kann.

1.1 Motivation

The consequences of design quality (or lack thereof) propagate throughout the software life cycle.
(Card, Glass, 1990, S. ix)

Der Entwurf ist eine der wichtigsten Phasen in der Software-Entwicklung. Obwohl nur 5-10% des Gesamtaufwands über den Software-Lebenszyklus in den Entwurf selbst gehen, fließt der meiste Aufwand in vom Entwurf maßgeblich beeinflusste Phasen: Implementierung (15-20%) und Wartung (über 50%).

Ein schlechter Entwurf kann teure Folgen haben: Bis zu 80% des Gesamtaufwands müssen für die Bereinigung falscher Entwurfsentscheidungen aufgewendet werden (Bell et al., 1987). Das liegt an der Fehlerfortpflanzung: Wird ein Entwurfsfehler erst nach der Implementierung behoben, ist das etwa zehnmal teurer, als wenn der Fehler in der Entwurfsphase behoben worden wäre (Boehm, 1983; Dunn, 1984). Neuen Schätzungen von Boehm und Basili (2001) zufolge kann der Faktor auch höher ausfallen: zwischen 5 und 100.

Deshalb lohnt es sich, einen guten Entwurf zu erstellen. Obwohl der Entwurfsaufwand dabei durch sorgfältiges Arbeiten in der Regel zunimmt, ist der Gesamtaufwand geringer, da hohe Fehlerfolgekosten vermieden werden. Um eine hohe Entwurfsqualität zu garantieren, sollte der Entwurf bereits in der Entwurfsphase geprüft werden. Die tatsächliche Entwurfsqualität offenbart sich zwar erst in Implementie-

rung und Wartung, doch gibt es einige nützliche Indikatoren, die Hinweise auf die tatsächliche Entwurfsqualität geben. Die am häufigsten verwendeten Indikatoren für Entwurfsqualität sind Entwurfsmetriken. Rombach (1990) tritt sehr für die Erhebung von Metriken in den frühen Phasen der Software-Entwicklung ein. Er untermauert das durch die Feststellung einer hohen Korrelation zwischen Entwurfsmetriken (z. B. zur Modularität) und der Wartbarkeit des Systems.

1.2 Zielsetzung

In dieser Arbeit wird ein Bewertungsverfahren für objektorientierte Entwürfe entwickelt. Dass die Messung der Entwurfsqualität nicht einfach ist, beschreibt William Agresti eindrücklich in seinem Vorwort zu dem Buch von Card und Glass (1990) zu diesem Thema:

Measurement of physical reality, we learned, involved three elements: an object, an observable characteristic, and an apparatus for performing the measurement. The measurement examples, giving instances of object/observable/apparatus – "table/length/yardstick" – were clear enough. Well, it's a long way from "table/length/yardstick" to "software/design quality/design quality analyzer". In each of the three elements of measurement, we have major problems. Our "object", software, is invisible and intangible. Our "observable", design quality, raises the issue: What is quality? What is the relative importance of simplicity, maintainability, efficiency, and other characteristics that contribute to design quality? Our "apparatus" is often a tool that processes the source code for what it reveals of design because earlier design descriptions are seldom complete or formally represented.

Die wichtigsten Fragen, die ein Bewertungsverfahren für den Entwurf zu beantworten hat, sind also:

1. Messgegenstand: Was ist Entwurf? Welche Entwurfsbestandteile sind relevant?
2. Messziel: Was ist Entwurfsqualität?
3. Messverfahren: Wie wird die Entwurfsqualität gemessen?

Messgegenstand. Diese Arbeit konzentriert sich auf den Bereich des objektorientierten Entwurfs, da sich dieser in den letzten zehn Jahren zum dominanten Entwurfsansatz entwickelt hat. Die Unified Modeling Language (UML) hat sich seit ihrer Einführung 1996 und ihrer Standardisierung 1997 durch die OMG als *die* Standardnotation für den objektorientierten Entwurf etabliert, weshalb es nahe liegt, als Messgegenstand UML-Entwurfsdokumentation zu verwenden. UML ist außerdem hinreichend formal, um aus UML-Modellen Entwurfsinformation automatisch extrahieren zu können. Weil sich ein Entwurf zwar weitgehend, aber nicht auf sinnvolle Weise vollständig mit UML dokumentieren lässt, wird zusätzlich zum UML-Modell auch begleitende Entwurfsdokumentation in die Bewertung einbezogen.

Messziel. Für die Entwurfsqualität gibt es keine einheitliche Definition oder gar einen Standard. Das gilt besonders für den objektorientierten Entwurf. Allerdings gibt es einiges an dokumentiertem Erfahrungswissen, z. B. als Ratschläge in Form von Entwurfsregeln und Mustern. Aus diesem Wissen lässt sich ein Qualitätsmodell ableiten, indem man die jeweiligen Ziele der Ratschläge (z. B. verringerte Kopplung) betrachtet. Da Entwurfsqualität multidimensional ist (Bosch, 2000), wird auch das Qualitätsmodell multidimensional sein müssen.

Messverfahren. Das Messverfahren ist eine Metrik, die durch das Qualitätsmodell fundiert ist.

Zielgruppen

Das Bewertungsverfahren richtet sich sowohl an Neulinge als auch an fortgeschrittene Entwerfer. Neulinge dürften allerdings einen größeren Bedarf haben, das Modell einzusetzen, weil es ihnen an Erfahrung fehlt und sie daher auch nicht über Intuition im Bereich der Entwurfsqualität verfügen. Dieses Defizit kann teilweise durch das Bewertungsverfahren (und daraus gewonnene Richtlinien) ausgeglichen werden.

Einsatzgebiete

Das Verfahren eignet sich

- zur Feststellung der Qualität eines Entwurfs,
- zur Entscheidung zwischen Entwurfsalternativen, indem alle Alternativen bewertet werden und diejenige mit der besten Bewertung ausgewählt wird und
- zur Untersuchung eines objektorientierten Entwurfs auf mögliche Mängel hinsichtlich verschiedener Eigenschaften (z. B. Wartbarkeit).

Die Mängelanalyse liefert in der Regel nur Hinweise auf potentielle Mängel. Sie erlaubt jedoch eine bessere Fokussierung analytischer Maßnahmen zur Qualitätssicherung, welche die tatsächlichen Mängel identifizieren können. Die gefundenen Mängel können dann behoben werden. Die Entwurfsverbesserung wird erleichtert, wenn die Mängelanalyse auch gleich Hinweise auf bessere alternative Strukturen liefert. Der Alternativenvergleich kann auch bei entwurfsverbessernden Restrukturierungen zum Vorher-Nachher-Vergleich verwendet werden, um eine tatsächliche Verbesserung sicherzustellen.

Anforderungen

Damit das Bewertungsverfahren praktisch anwendbar ist, sollten die folgenden Anforderungen erfüllt sein:

1. Das Verfahren sollte so früh wie möglich in der Entwurfsphase einsetzbar sein. Daraus folgt, dass es auch mit unvollständigen und wenig detaillierten Entwurfsbeschreibungen zurechtkommen sollte.
2. Das Verfahren sollte nur wenige, in der Regel gegebene Voraussetzungen für den erfolgreichen Einsatz haben. Ist die Einstiegshürde zu hoch, wird es sonst von den meisten Entwicklern ignoriert oder schnell aufgegeben.
3. Das Qualitätsmodell sollte klein und überschaubar sein. Bewertungsverfahren mit Metriken werden in der Praxis nur dann wirklich eingesetzt, wenn sie wenige, relativ simple Metriken umfassen (Card, Glass, 1990).
4. Das Qualitätsmodell sollte an konkrete Anforderungen anpassbar sein. Wenn bestimmte Qualitätsattribute eine große, eine geringe oder auch gar keine Rolle spielen, sollte das Modell so konfiguriert werden können, dass sich dies in der Bewertung entsprechend niederschlägt (durch Umfang des Modells und Gewichtung innerhalb des Modells).

5. Zusammen mit dem Verfahren sollten Hilfsmittel zur Verfügung gestellt werden, die es erlauben, einen Entwurf mit möglichst geringem Aufwand (d. h. weitgehend automatisiert) zu bewerten. Eine Prüfung des Entwurfs durch Experten ist zwar üblich (Haynes, 1996), aber teuer. Außerdem sind Experten schwer zu bekommen (Grotehen, Dittrich, 1997).

1.3 Lösungsansatz

Der Ablauf des Bewertungsverfahrens für objektorientierte Entwürfe ist in Abbildung 1-1 dargestellt. Die Metamodelle beschreiben jeweils die rechts daneben stehenden Modelle.

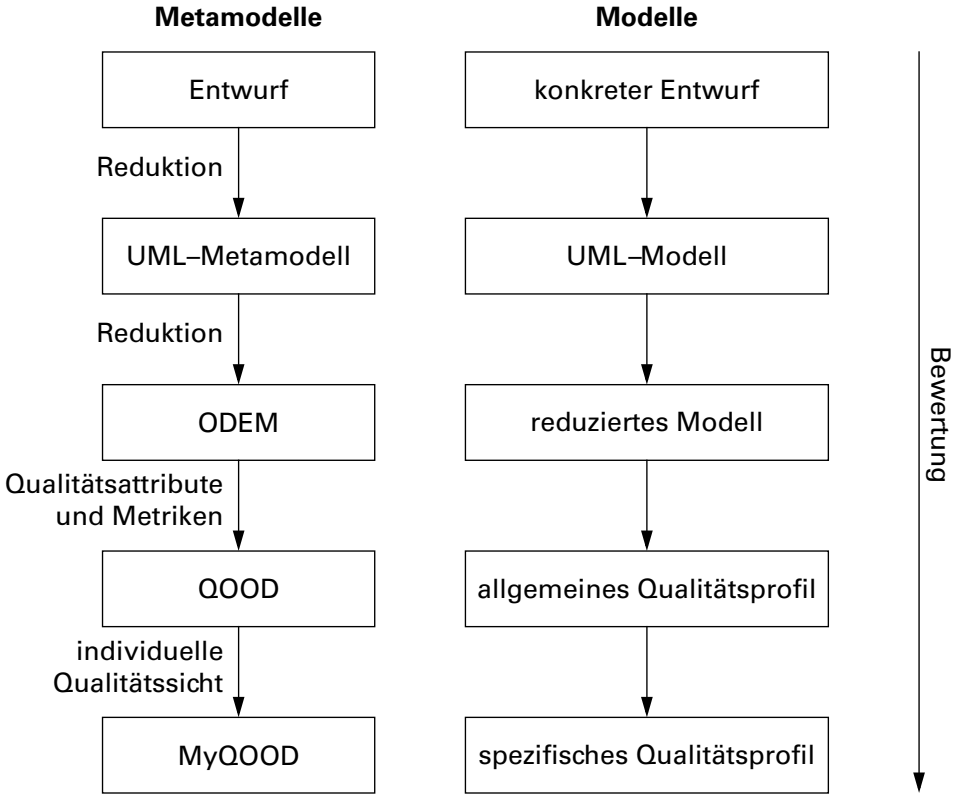

Abbildung 1-1: Verfahren für die Entwurfsbewertung

Der konkrete Entwurf ist der eigentliche Gegenstand der Bewertung. Er enthält unter anderem UML-Diagramme. Die in den UML-Diagrammen enthaltene Information, das UML-Modell, ist eine Instanz des UML-Metamodells. Der konkrete Entwurf kann also auf ein UML-Modell reduziert werden. Die Reduktion stellt bereits eine Wertung dar, da die weggelassenen Informationen keinen Einfluss mehr auf die Bewertung haben können.

Aus der Entwurfsinformation im UML-Modell wird der Teil ausgewählt, der für die Entwurfsbewertung als relevant angesehen wird. Durch diese Auswahl erhält man das reduzierte Modell des Entwurfs. Das zugrunde liegende Metamodell heißt Object-Oriented Design Model (ODEM). ODEM beschränkt sich auf statische Entwurfsinformation, die aus Klassendiagrammen gewonnen werden kann (z. B. welche Klassen vorhanden sind und welche Beziehungen diese untereinander haben).

Das allgemeine Qualitätsmodell Quality Model for Object-Oriented Design (QOOD) ordnet Entwurfseigenschaften aus dem reduzierten Modell Qualitätsattributen zu und gibt objektive und subjektive Metriken zur Messung der Eigenschaften an. Die objektiven Metriken werden auf der Basis von ODEM formal definiert. Als Hilfsmittel für die Erhebung der subjektiven Metriken werden Fragebögen eingesetzt. Durch Einsatz von QOOD kann aus dem reduzierten Modell ein allgemeines Qualitätsprofil berechnet werden.

Von QOOD wird anhand der gewählten Qualitätssicht ein spezifisches Modell abgeleitet (hier als MyQOOD bezeichnet). Dazu werden aus den vorhandenen Qualitätsattributen und Metriken von QOOD die relevanten ausgewählt und gewichtet. Das spezifische Modell erlaubt die Berechnung eines spezifischen Qualitätsprofils aus dem allgemeinen Qualitätsprofil. Die Qualitätsattribute in QOOD sind hierarchisch geordnet, so dass man durch Aggregation der Messwerte untergeordneter Attribute Messwerte für übergeordnete Attribute erhält. Die Aggregation wird dabei von der Gewichtung im spezifischen Qualitätsmodell bestimmt. Durch schrittweise Aggregation kann man so eine einzige Qualitätskennzahl für den Entwurf bestimmen. Ein spezifisches Qualitätsmodell könnte aber auch mehrere Kennzahlen liefern.

Die Trennung zwischen einem allgemeinen und einem spezifischen Qualitätsmodell erlaubt die Berücksichtigung individueller Sichten auf Entwurfsqualität (gemäß Anforderung 4 aus Abschnitt 1.2). Da es sehr viele unterschiedliche Sichten gibt, ist ein allgemein gültiges Qualitätsmodell kaum sinnvoll. Stattdessen verwendet man spezifische Qualitätsmodelle, die dem Bedarf angepasst sind.

1.4 Übersicht

Die Struktur der Arbeit ist in Abbildung 1-2 dargestellt. Die Abbildung zeigt die Kapitel und die wichtigsten Abhängigkeiten zwischen den Kapiteln (dargestellt durch Pfeile mit Informationen aus dem Kapitel, die in einem anderen Kapitel verwendet werden).

Kapitel 2 beschreibt die wichtigen Basisbegriffe Modell und Metrik. Kapitel 3 führt in die Objektorientierung und die Standardnotation UML ein. Kapitel 4 beschäftigt sich mit dem objektorientierten Entwurf und seinen Problemen. Kapitel 5 stellt ODEM vor, ein Referenzmodell für den objektorientierten Entwurf auf der Basis des UML-Metamodells.

Nachdem damit der Messgegenstand definiert ist, werden die Grundlagen für das Messverfahren gelegt. Zunächst wird auf das Messziel und bisherige Messverfahren eingegangen. Kapitel 6 beschäftigt sich mit Softwarequalität und Qualitätsmodellen im Allgemeinen, während Kapitel 7 die entwurfsspezifischen Aspekte der Softwarequalität beleuchtet und bisherige Ansätze zur Entwurfsbewertung vorstellt.

Auf dieser Grundlage wird das Messverfahren entwickelt. Kapitel 8 führt QOOD, das allgemeine Qualitätsmodell für den objektorientierten Entwurf ein. Der Aufbau sowie die Qualitätsattribute von QOOD (unterschieden in Faktoren und Kriterien) werden beschrieben. Kapitel 9 befasst sich mit der Quantifizierung des wichtigsten Faktors von QOOD, der Wartbarkeit, auf der Basis von Metriken und Fragebögen. Außerdem wird beschrieben, wie aus den Messwerten eine Bewertung gewonnen wird.

In Kapitel 10 wird ein spezifisches Qualitätsmodell für eine Fallstudie entwickelt, angewendet und validiert. In Kapitel 11 wird beschrieben, wie das Verfahren durch Werkzeuge unterstützt werden kann.

Kapitel 12 schließlich fasst die Arbeit zusammen und bewertet sie. Der vorgestellte Ansatz zur Entwurfsbewertung wird mit anderen Arbeiten verglichen. Den Abschluss der Arbeit bildet ein Ausblick.

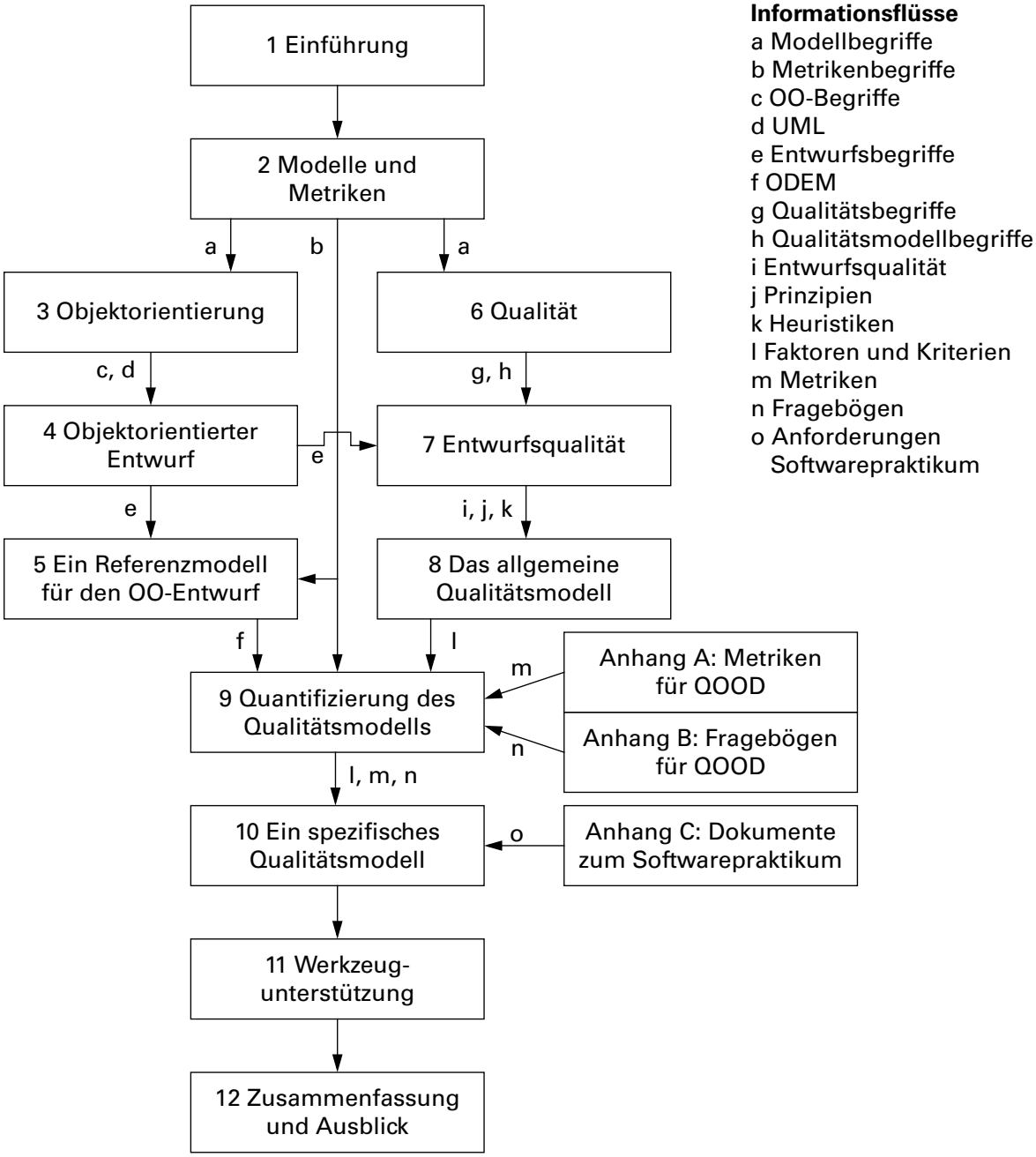

Abbildung 1-2: Struktur der Arbeit

Kapitel 2

Modelle und Metriken

In diesem Kapitel werden zwei grundlegende Begriffe eingeführt, die für die Qualitätsmodellierung von Bedeutung sind. Zunächst geht es um den Begriff Modell, dann um die Metrik, eine spezielle Form des Modells.

2.1 Modelle

Models, of course, are never true, but fortunately it is only necessary that they be useful.
(Box, 1979, S. 2)

2.1.1 Definition

Ein Modell ist ein Abbild (deskriptives Modell) oder ein Vorbild (präskriptives Modell) eines Objekts. Das Objekt, auf das sich das Modell bezieht, heißt Original. Stachowiak (1973) formuliert in seiner Allgemeinen Modelltheorie drei Hauptmerkmale eines Modells: Abbildungsmerkmal, Verkürzungsmerkmal und pragmatisches Merkmal. Diese werden im Folgenden näher erläutert.

Abbildungsmerkmal

Modelle sind stets Modelle von etwas, nämlich Abbildungen, Repräsentationen natürlicher oder künstlicher Originale, die selbst wieder Modelle sein können.
(Stachowiak, 1973, S. 131)

Ein Modell bildet Attribute (Eigenschaften) des Originals auf Modellattribute ab. Dabei kann das Original bereits ein Modell sein. Beispielsweise ist die Spezifikation ein Modell des Codes und der Code wiederum ein Modell des ausführbaren Programms (Ludewig, 1998).

Verkürzungsmerkmal

Modelle erfassen im allgemeinen nicht alle Attribute des durch sie repräsentierten Originals, sondern nur solche, die den jeweiligen Modellerschaffern und/oder Modellbenutzern relevant erscheinen.
(Stachowiak, 1973, S. 132)

Die Abbildung des Originals auf das Modell ist in der Regel kein Isomorphismus. Es können Attribute weggelassen werden (Verkürzung). Die weggelassenen Attribute heißen präterierte Attribute (vgl. Abbildung 2-1). Andererseits können auch Attribute dem Modell hinzugefügt werden, die keine Entsprechung im Original haben (abundante Attribute). Beispielsweise enthält der Code als Modell des Entwurfs einige Attribute, die nicht aus dem Entwurf abgeleitet wurden, sondern durch die gewählte Programmiersprache erforderlich wurden.

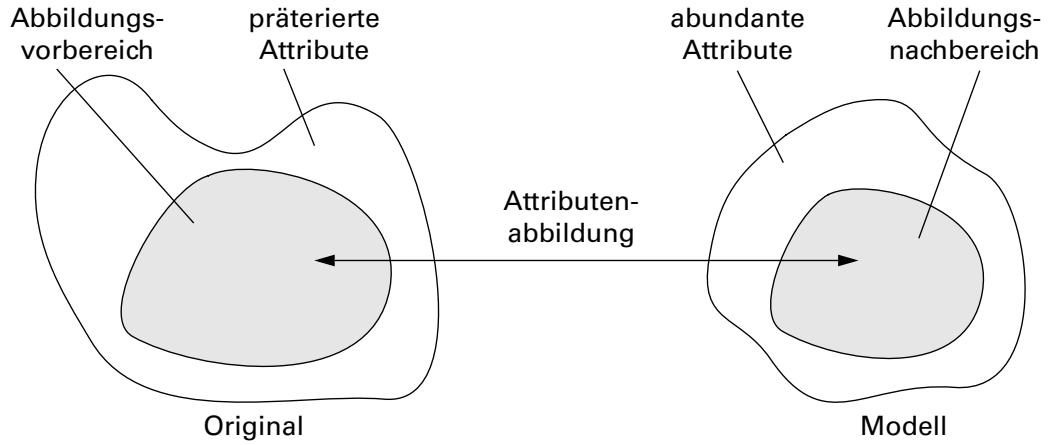

Abbildung 2-1: Original-Modell-Abbildung (nach Stachowiak, 1973, S. 157)

Pragmatisches Merkmal

Modelle sind ihren Originalen nicht per se eindeutig zugeordnet. Sie erfüllen ihre Ersetzungsfunktion a) für bestimmte – erkennende und/oder handelnde, modellbenutzende – Subjekte, b) innerhalb bestimmter Zeitintervalle und c) unter Einschränkungen auf bestimmte gedankliche oder tatsächliche Operationen.
(Stachowiak, 1973, S. 132)

Die Modellbildung ist kein zweckfreier Vorgang, sondern es liegt immer eine Absicht zugrunde. Die Modellbildung erfolgt also auf Grund von pragmatischen Erwägungen. Die Pragmatik bestimmt unter anderem, welche Attribute des Originals weggelassen werden können, ohne die angestrebte Nutzung des Modells an Stelle des Originals zu gefährden. Häufig soll das Modell nämlich dazu dienen, Erkenntnisse oder Fertigkeiten mittels des Modells zu gewinnen, um diese dann auf das Original zu übertragen (z. B. Simulationsmodelle).

Außerdem ist das Modell für einen bestimmten Nutzerkreis gedacht (z. B. dient der Entwurf für ein Software-System vor allem den Entwicklern) und unter Umständen auch nur für einen bestimmten Zeitraum gültig (z. B. das Organigramm eines Unternehmens).

2.1.2 Beispiele

[…] any program is a model of a model within a theory of a model of an abstraction of some portion of the real world or some universe of discourse.
(Lehman, 1980, S. 1061)

Modellbildung ist eine universelle Technik zum besseren Verständnis von realen oder gedachten Objekten oder Prozessen, weshalb man quasi überall auf Modelle trifft. In dieser Arbeit finden sich unter anderem die folgenden Beispiele für Modelle:

- Metrik (Modell von einen Messgegenstand, z. B. von Software)
- Spezifikation (Modell für ein Programm)
- Entwurf (Modell für ein Programm)
- Code (Modell für ein Programm)
- UML-Modell (Modell für ein Programm)
- UML-Metamodell (Modell für UML-Modelle)
- Qualitätsmodell (Modell für ein Programm o. Ä., das sich aus Qualitätsattributen zusammensetzt)

Die präskriptiven Modelle sind dabei klar in der Überzahl, ein typisches Phänomen in der Software-Entwicklung.

2.2 Metriken

I often say that when you can measure what you are speaking about, and express it in numbers, you know something about it; but when you cannot measure it, when you cannot express it in numbers, your knowledge is of a meagre and unsatisfactory kind: it may be the beginning of knowledge, but you have scarcely, in your thoughts, advanced to the stage of science, whatever the matter may be.
(William Thomson (Lord Kelvin), Lecture to the Institution of Civil Engineers, 03.05.1883)

2.2.1 Definition

In der Mathematik ist eine Metrik ein Abstandsmaß. Im Software Engineering wurde der Begriff Metrik (metric) verallgemeinert auf beliebige Maße der Software-Entwicklung. Die Definition des IEEE lautet (die Definition im ISO-Standard 9126 ist ähnlich):

Definition 2-1 (metric, IEEE Std. 610.12-1990)

A quantitative measure of the degree to which a system, component, or process possesses a given attribute.

Fenton, Pfleeger (1996) und Dumke (2000) unterscheiden drei Arten von Metriken:

- Metriken für Produkte, z. B. Größe der Spezifikation und Korrektheit des Codes,
- Metriken für den Entwicklungsprozess, z. B. Aufwand und Dauer, und
- Metriken für eingesetzte Ressourcen, z. B. Größe des Entwicklungsteams.

Da es in dieser Arbeit um die Entwurfsbewertung geht und der Entwurf ein Produkt der Software-Entwicklung ist, werden im Folgenden nur Produktmetriken betrachtet.

Eine Produktmetrik ist ein spezielles Modell für Software. Das Abbild ist meistens ein einzelner Wert, in der Regel eine Zahl. Es findet also eine sehr starke Verkürzung statt. Durch die Reduktion auf einen Wert (d. h. Abstraktion) sind Eigenschaften einer Software leichter zu erkennen. Die Pragmatik einer Metrik ist unterschiedlich. Häufig will man durch Erheben einer Metrik bestimmte Eigenschaften feststellen. Misst man mehrfach über die Zeit, kann man auch Trends erkennen. Beispielsweise ist es sinnvoll, während der Software-Entwicklung regelmäßig den Umfang des entstehenden Produkts zu messen, um Verzögerungen gegenüber dem Plan erkennen zu können.

Eine Metrik ist formal gesehen eine Abbildung (Homomorphismus) eines empirischen Relationensystems auf ein formales Relationensystem (in der Regel ein numerisches System). Kriz (1988) verdeutlicht den Zweck der Metrik: Durch die Abstraktion der Metrik kann man zu Erkenntnissen gelangen, die wegen der Beschränktheit des menschlichen Denkvermögens (Verständnisbarriere) am Original nur schwer oder gar nicht zu finden sind (siehe Abbildung 2-2).

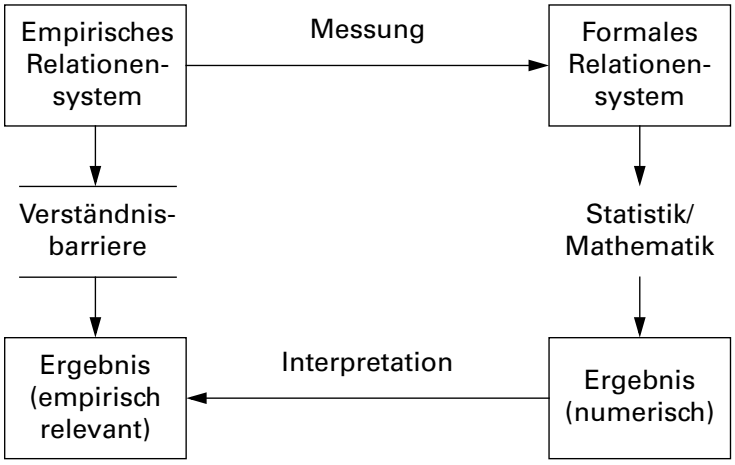

Abbildung 2-2: Messprozess nach Kriz (Abbildung nach Zuse, 1994, S. 137)

Das formale Relationensystem hat eine wichtige Eigenschaft: die Zulässigkeit bestimmter mathematischer Operationen. Diese werden durch den Skalentyp des Systems charakterisiert. Fenton und Pfleeger (1996) unterscheiden die folgenden Skalentypen: Nominalskala, Ordinalskala, Intervallskala, Rationalskala und Absolutskala. Tabelle 2-1 zeigt die Eigenschaften der Skalen und die Unterschiede.

Zählmetriken, die in dieser Arbeit eine wichtige Rolle spielen, sind Metriken mit Absolutskala. Da sich ihr Wertebereich auf die natürlichen Zahlen beschränkt, sind einige Operationen wie z. B. die Bildung von Durchschnitten eigentlich nicht mehr möglich. Daher bettet man den Wertebereich sinnvollerweise in die rationalen Zahlen ein, um besser rechnen zu können.

2.2.2 Verwendung

Using some metrics is better than using no metrics.
(Pfleeger, 2000, S. 225)

Wie wichtig Metriken für die qualitativ hochwertige Software-Entwicklung sind, zeigt die Tatsache, dass im Capability Maturity Model (CMM, Humphrey, 1988) bereits ab Stufe 2 (repeatable) der Einsatz von Metriken verlangt wird.

Merkmalsart	Qualitative Merkmale		Quantitative Merkmale (kontinuierliche oder diskrete)		
	Nominal-merkmal	Ordinal-merkmal			
Skalentyp	Topologische Skalen		Kardinalskalen (Metrische Skalen)		
	Nominalskala	Ordinalskala	Intervallskala	Rationalskala	Absolutskala
Definierte Beziehungen	=, ≠	=, ≠ <, >	=, ≠ <, > +, -	=, ≠ <, > +, - *, /	=, ≠ <, > +, - *, /
Interpretation der hinzukommenden Beziehungen	Unterscheidung gleich/ ungleich möglich	Unterscheidung kleiner/ größer möglich	Differenzen haben empirischen Sinn	Verhältnisse haben empirischen Sinn	–
Zugelassene Transformationen	umkehrbar eindeutige (bijektive)	monoton steigende (isotone)	lineare (y=ax+b, a>0)	Ähnlichkeitstransform. (y=ax, a>0)	Identität (y=x)
Beispiele für Merkmale	Postleitzahlen Autokennzeichen Artikelnummern Symbole Fehlerursachen (IEEE Std. 1044-1993, S. 9)	Schulnoten Militärische Dienstgrade Mercallische Erdbebenskala Windstärke Beaufort Prozessreifegrad (CMM)	Celsius-Temperatur Kalenderdatum zyklomatische Komplexität (McCabe,1976)	Kelvin-Temperatur Einkommen Richtersche Erdbebenskala Windgeschw. m/s Projektdauer	Teamgröße Fehlerzahl Lines of Code
Beispiele für statistische Kennwerte	Modalwert Häufigkeiten	Quantile (Median, Quartile, ...)	arithmetischer Mittelwert Standardabweichung	geometrischer Mittelwert Variationskoeffizient	wie Rationalskala, wenn der Wertebereich in die rationalen Zahlen eingebettet wird
Statistische Verfahren	nichtparametrische		parametrische, unter Beachtung der Modellvoraussetzungen		
Informationsinhalt	gering → hoch				
Empfindlichkeit gegenüber Ergebnisabweichungen	gering → hoch				

Tabelle 2-1: Übersicht über die Skalentypen (nach DIN 55350, Teil 12, S. 11, erweitert um die Absolutskala)

Anwendungsbereiche

Metriken lassen sich für unterschiedliche Zwecke einsetzen. Whitmire (1994) unterscheidet die folgenden Anwendungsbereiche von Metriken:

- Schätzung: auf der Basis anderer Produkte (historische Daten), z. B. den zu erwartenden Entwicklungsaufwand aus Daten früherer Projekte ableiten.
- Vorhersage: auf der Basis von Messwerten des Produkts andere Eigenschaften des Produkts vorhersagen, z. B. Verlässlichkeit aufgrund von Testergebnissen.
- Bewertung: Vergleich von Messwerten mit Sollwerten, die z. B. von einem Standard festgelegt sind. Anwendung z. B. für die Prüfung von Qualitätszielen (durch Schwellenwerte) oder zur Auswahl von vermutlich fehlerträchtigen Klassen für Reviews.
- Vergleich: Vergleich der Messwerte von Alternativen zur Entscheidungsunterstützung, z. B. zur Auswahl einer Entwurfsalternative.
- Untersuchung: Messung zur Stützung oder Widerlegung einer Hypothese.

In dieser Arbeit werden Metriken vor allem zur Vorhersage, Bewertung und zum Vergleich verwendet.

Anforderungen

Um eingesetzt werden zu können, sollten Metriken bestimmte Anforderungen erfüllen. Basili und Rombach (1988), Daskalantonakis (1992) und Gillies (1992) haben solche Anforderungen formuliert, die hier zusammengefasst sind:

1. Die Metrik soll leicht verständlich sein.
2. Die Metrik soll präzise definiert sein.
3. Die Metrik soll wiederholbar sein.
4. Die Metrik soll eindeutig sein und Vergleiche erlauben.
5. Die Metrik soll (soweit wie möglich) objektiv sein.
6. Die Metrik soll einfach und kosteneffektiv erhebbar sein, am besten automatisch.
7. Der Zweck der Messung soll klar sein.
8. Die Metrik soll ein Interpretationsmodell haben, das Aussagen darüber macht, was ein bestimmter Wert bedeutet.
9. Die Metrik soll informativ sein, d. h. Änderungen des Messwerts können sinnvoll interpretiert werden.

2.2.3 Qualitätsmetriken

Für diese Arbeit sind Qualitätsmetriken besonders wichtig, denn eine Bewertung der Entwurfsqualität ist eine Qualitätsmetrik, ebenso wie die bei der Bewertung als Zwischenergebnisse verwendeten Metriken.

Definition 2-2 (quality metric, IEEE Std. 1061-1992)

A function whose inputs are software data and whose output is a single numerical value that can be interpreted as the degree to which software possesses a given attribute that affects its quality.

Nach DeMarco (1982) gibt es zwei Kategorien von Qualitätsmetriken: Ergebnismetriken (result metrics) und Vorhersagemetriken (predictor metrics). Beide messen etwas Vorhandenes (z. B. die Anzahl der bisher gefundenen Fehler im Code oder die Anzahl der Klassen im System), aber mit unterschiedlicher Intention. Ergebnismetriken machen eine Aussage über das Vorhandene, während Vorhersagemetriken eine Vorhersage für eine andere Größe, z. B. die Wartbarkeit, ableiten. Die Zusammenhänge zwischen der vorhergesagten und der tatsächlichen Größe sind empirisch zu zeigen. Kitchenham (1990) formuliert diese Anforderung an eine Vorhersagemetrik wie folgt:

1. Die Eigenschaft, die als Basis der Vorhersage dienen soll, ist genau messbar,
2. es gibt einen Zusammenhang zwischen dieser Eigenschaft und der vorherzusagenden Eigenschaft und
3. dieser Zusammenhang ist klar, validiert und kann als Formel oder anderes Modell formuliert werden.

Gerade Anforderung 3 (insbesondere die Validierung) wird laut Kitchenham häufig vergessen, wenn Vorhersagemetriken vorgeschlagen werden.

Wendet man Anforderung 2 und 3 auf Qualitätsmetriken an, muss also der Zusammenhang zwischen der Metrik und dem Qualitätsattribut vorhanden und nachweisbar sein. Außerdem muss die Metrik die verschiedenen Grade des Attributs unterscheiden können und auf eine geeignete Skala abbilden (Gillies, 1992).

Bei der Auswahl von Qualitätsmetriken muss auch die Qualitätssicht desjenigen, für den die Qualität gemessen werden soll, berücksichtigt werden. Außerdem gilt für die meisten Qualitätsattribute, dass sie zu komplex sind, um durch eine einzige Metrik erfasst werden zu können (Basili, Rombach,1988), so dass mehrere Qualitätsmetriken für ein Attribut verwendet werden sollten.

2.2.4 Entwicklung von Metriken

Shepperd und Ince (1993) stellen ein Verfahren zur Entwicklung einer Metrik vor (vgl. Abbildung 2-3). Es besteht aus den folgenden Phasen:

1. Problembestimmung: Der Gegenstand (das Original) der Metrik sowie Zweck und Zielgruppe (Pragmatik) werden festgelegt. [Kapitel 1]
2. Informales Modell: Vorhandenes Wissen und Vermutungen aus dem Problembereich werden gesammelt und daraus die relevanten Faktoren für die Metrik abgeleitet. [Kapitel 3, Kapitel 4, Kapitel 6, Kapitel 7]
3. Formales Modell: Die Abbildung der Eingabe auf die Ausgabe und die gewünschte Genauigkeit werden festgelegt (Abbildungs- und Verkürzungsmerkmal). Außerdem werden Axiome aufgestellt, die für die Metrik gelten sollen. Getroffene Annahmen werden dokumentiert. Bei Bedarf wird die Metrik zur Verallgemeinerung um Modellparameter erweitert. [Kapitel 5, Kapitel 8, Kapitel 9]

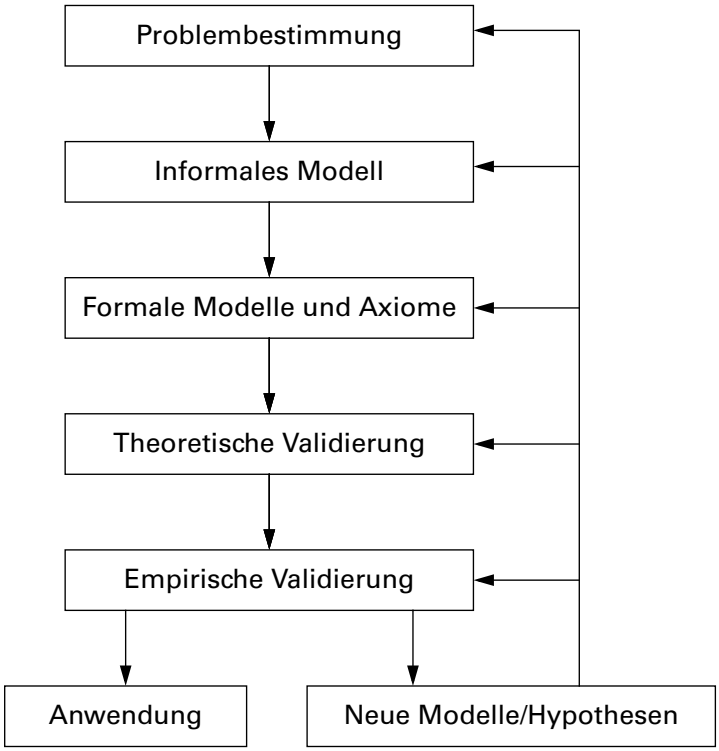

Abbildung 2-3: Verfahren zur Entwicklung von Metriken (nach Shepperd, Ince, 1993, S. 79)

4. Theoretische Validierung: Es wird überprüft, ob die postulierten Axiome für die Metrik gelten. [Anhang A]
5. Empirische Validierung: Es wird empirisch überprüft, ob die Metrik tatsächlich die Anforderungen aus der Problembestimmung erfüllt. [Kapitel 10]
   Dabei gemachte Erfahrungen können neue Modelle oder Hypothesen liefern und zu einer erneuten Iteration des Vorgehens führen, um die Metrik und ihre Grundlagen zu verbessern.
6. Anwendung: Einsatz der Metrik. [Kapitel 10]

Ein Qualitätsmodell für den objektorientierten Entwurf ist letztlich eine sehr komplexe Metrik, also lässt sich das Verfahren von Shepperd und Ince zu seiner Entwicklung verwenden. Bei der obigen Beschreibung der Phasen des Verfahrens ist in Klammern angegeben, in welchen Kapiteln dieser Arbeit die jeweiligen Aspekte behandelt werden. Für die Metriken, die für das Qualitätsmodell benötigt werden, kann das Verfahren ebenfalls angewendet werden.

Kapitel 3

Objektorientierung

What is object oriented programming? My guess is that object oriented programming will be in the 1980's what structured programming was in the 1970's. Everyone will be in favor of it. Every manufacturer will promote his products as supporting it. Every manager will pay lip service to it. Every programmer will practice it (differently). And no one will know just what it is.
(Rentsch, 1982, S. 51)

Dieses Kapitel führt die wesentlichen Begriffe der Objektorientierung ein und zeigt ihre Darstellung in der Unified Modeling Language (UML). Eine ausführlichere Einführung in die Objektorientierung, die objektorientierte Analyse und den objektorientierten Entwurf findet sich z. B. bei Booch (1994).

Philosophische Grundlagen der Objektorientierung finden sich bei Bunge (1977, 1979). Bunge entwickelt eine Ontologie, nach der die Welt aus substantiellen Individuen besteht, die eine Identität und Eigenschaften aufweisen: Objekte. Wand (1989) nutzt Bunges Ontologie, um daraus ein formales Modell für die Objektorientierung zu gewinnen. Beispielsweise entstehen Klassen, indem Objekte anhand der Ähnlichkeit ihrer Eigenschaften zusammengefasst werden. Aus dieser Weltsicht heraus ergibt sich die populäre Schlussfolgerung, dass ein objektorientiertes Programm besser verständlich und besser änderbar ist als ein prozedurales, da der kognitive Unterschied zwischen Konstrukten der Anwendungswelt und den Konstrukten der Lösungswelt geringer ist. Jacobson et al. (1995, S. 42ff.) bezeichnen den Unterschied zwischen Anwendungs- und Lösungswelt als „semantic gap“. Dieser sei bei der Objektorientierung geringer als bei der strukturierten Entwicklung. Für Rumbaugh et al. (1993) ist die Klasse eine natürliche Einheit der Modularisierung, weshalb durch die objektorientierte Vorgehensweise klare und verständliche Entwürfe entstehen.

Berard (1993, Kap. 2) empfiehlt die Verwendung des objektorientierten Ansatzes wegen seiner Vorteile aus technischer Sicht: die zunehmende Akzeptanz moderner Programmierpraktiken (z. B. Datenabstraktion), höhere Wiederverwendung und bessere Erweiterbarkeit im Vergleich zum strukturierten Ansatz. Allerdings ist bei einem Wechsel von der strukturierten Entwicklung zur Objektorientierung ein Paradigmenwechsel nötig (Fichman, Kemerer, 1992). Diesen Wechsel schafft nicht jeder Entwick-

ler: Berg et al. (1995) kommen in einer Untersuchung zu dem Ergebnis, dass 80% der neu ausgebildeten Entwickler die Grundzüge der objektorientierten Entwicklung verstanden haben und sie anwenden können. Von diesen 80% entwickelten sich 5% zu sehr guten Entwicklern (top performers), 15% waren immerhin gut (journeyman level). Die große Mehrheit blieb allerdings Mittelmaß. Dass die objektorientierte Vorgehensweise nicht ohne Schwierigkeiten gemeistert werden kann, zeigen auch die Sammlungen typischer Fehler von Webster (1995) und Alexander (2001).

3.1 Begriffe

Die Begriffe in der Objektorientierung werden häufig unterschiedlich definiert, oder es werden für dasselbe Konzept unterschiedliche Begriffe verwendet. Das führt zu einem großen Begriffschaos (Snyder, 1993). Daher werden die Definitionen der zentralen Begriffe angegeben, wie sie in dieser Arbeit verwendet werden. Die verwendeten Definitionen stützen sich vor allem auf die Begriffsbildung im Zusammenhang mit UML gemäß Rumbaugh et al. (1998).

Drei Eigenschaften zeichnen nach Wegner (1987, 1992) die objektorientierte Sichtweise aus: Objekte, Klassen und Vererbung. Viele Autoren nehmen noch Polymorphismus und das damit zusammenhängende dynamische Binden als wichtige Eigenschaften hinzu. Daher werden diese Begriffe zuerst eingeführt.

3.1.1 Objekt

Die zentrale Rolle spielt der Begriff des Objekts. Ein Objekt besteht aus Datenfeldern, den so genannten *Attributen*, und aus Funktionen auf diesen Daten, den so genannten *Methoden*. Methoden dienen zur Reaktion auf *Nachrichten*, die ein Objekt versteht. Methoden können z. B. den Zustand des Objekts (die Werte seiner Attribute) verändern oder neue Nachrichten verschicken. Die Schnittstelle einer Methode wird als *Operation* bezeichnet; eine Methode ist also die Implementierung einer Operation.

3.1.2 Klasse

Die Objekte eines Systems werden nicht individuell beschrieben, sondern anhand ihrer Gemeinsamkeiten in Klassen zusammengefasst. Eine Klasse definiert die Attribute und Methoden ihrer Objekte. Die Klasse dient als Schablone zur Instantiierung (Erzeugung) von Objekten (*Instanzen*). Bei der Instantiierung müssen nur die Werte für die Attribute angegeben werden, die Methoden übernimmt das Objekt von seiner Klasse. Bei getypten objektorientierten Programmiersprachen wie C++ (Stroustrup, 1997), Java (Gosling et al., 1998) oder Eiffel (Meyer, 1991) ist das Typkonzept mit dem Klassenkonzept verknüpft: Bei der Deklaration einer Klasse wird automatisch auch ein gleichnamiger Typ deklariert. Dieser Typ verfügt über einen Wertebereich, der sich aus den Wertebereichen der Attribute zusammensetzt, und über Operationen, die den Methoden entsprechen. Daher definiert Meyer (1997) eine Klasse auch als die Implementierung eines abstrakten Datentyps. Ein Objekt ist vom Typ seiner Klasse.

Abbildung 3-1 zeigt die UML-Darstellung einer Klasse und eines Objekts dieser Klasse. Der Name des Objekts wird zur besseren Unterscheidung unterstrichen.

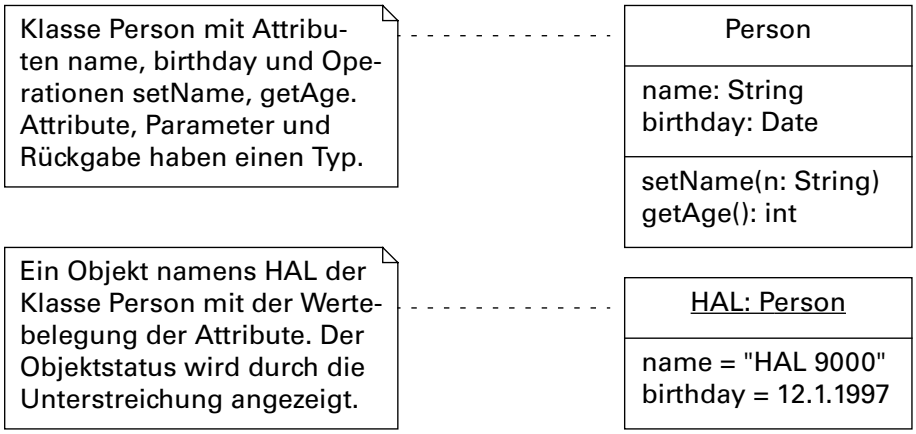

Abbildung 3-1: UML-Darstellung von Klasse und Objekt

Kommentare werden in UML durch einen Kasten mit Eselsohr dargestellt, der mit dem Modellelement, auf das sich der Kommentar bezieht, durch eine gestrichelte Linie verbunden ist.

3.1.3 Vererbung

Vererbung ist eine Beziehung zwischen Klassen. Eine Klasse kann sämtliche Eigenschaften (Attribute und Methoden) einer anderen Klasse erben, d. h. als Kopie übernehmen. Es dürfen außerdem weitere Eigenschaften hinzugefügt werden (Erweiterung) und geerbte Methoden modifiziert werden (Redefinition). Bei *Einfachvererbung* erbt eine Klasse von genau einer anderen Klasse, bei *Mehrfachvererbung* von mehreren Klassen. Die vererbende Klasse heißt *Oberklasse*, die erbende *Unterklasse*.

Bei getypten objektorientierten Programmiersprachen wird die Vererbungsrelation auf die korrespondierenden Typen übertragen: Eine erbende Klasse definiert einen Subtyp des durch die vererbende Klasse definierten Typs. Dadurch entsteht eine zur Vererbungsstruktur der Klassen isomorphe Typstruktur.

In UML wird die Vererbung durch einen Pfeil mit einer dreieckigen Spitze angezeigt, der von der Unterklasse zur Oberklasse geht (vgl. Abbildung 3-2). Geerbte Eigenschaften werden in der Darstellung der Unterklasse nicht wiederholt.

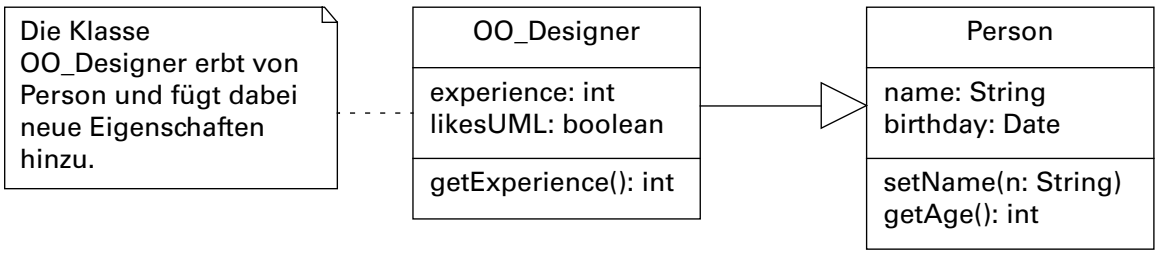

Abbildung 3-2: UML-Darstellung der Vererbung

Vererbung ist ein mächtiges Konzept; durch sie kann viel redundante Implementierung eingespart werden. Durch Erben kann aber die Kapselung durchbrochen werden, weil eine Unterklasse Zugriff auf die Implementierung der Oberklasse erhält (Snyder, 1986). Außerdem ist die Unterklasse durch das Erben stark an ihre Oberklasse gekoppelt: Jede Änderung der Oberklasse betrifft auch die Unterklasse.

Zusätzlich wirkt sich Vererbung tendenziell negativ auf Verständlichkeit, Wartbarkeit und Prüfbarkeit aus (Wilde, Huitt, 1992; Lejter et al., 1992; Wilde et al. 1993; Harrison et al., 2000b).

Vererbung kann für verschiedene Zwecke verwendet werden. Budd (1991), Meyer (1996, 1997) und Taivalsaari (1996) haben Klassifikationen für Vererbung aufgestellt, die zwar sehr detailliert sind, aber in manchen Belangen fragwürdig erscheinen. Die wichtigsten Arten sind Spezialisierung und Implementierungsvererbung.

Bei der *Spezialisierung* liegt der Schwerpunkt auf der Typ-/Subtyprelation von Ober- und Unterklasse, analog einer Spezialisierung in einer Taxonomie. Die Vererbung der Implementierung ist nur ein (willkommener) Nebeneffekt. Der Vorteil der Spezialisierung liegt darin, dass die Schnittstelle der Oberklasse auch von der Unterklasse angeboten wird – mit der Garantie, dass die Semantik der Schnittstelle erhalten bleibt (gemäß dem Liskovschen Substitutionsprinzip; vgl. Abschnitt 7.3.1). Ein Spezialfall ist die reine *Schnittstellenvererbung*, bei der gar keine Implementierung vererbt wird; dies ist beim Erben von rein abstrakten Klassen oder Interfaces (s. u.) der Fall.

Im Gegensatz dazu steht die *Implementierungsvererbung*, deren Schwerpunkt auf der Wiederverwendung von Code liegt. Hier wird eine Klasse um ihrer Implementierung willen beerbt, wobei die geerbte Implementierung mittels Redefinition, Erweiterung und Weglassen so zurechtgebogen wird, dass sie passt. Der bei der Spezialisierung garantierte Erhalt der Semantik ist, insbesondere durch das Weglassen, nicht mehr gegeben. Das kann zu schwer auffindbaren Fehlern führen, wenn Objekte der Klasse und ihrer Oberklasse gleichzeitig verwendet werden. Es gibt daher Empfehlungen, nur Spezialisierungsvererbung (am besten in Form von Schnittstellenvererbung) zu verwenden.

3.1.4 Polymorphismus und dynamisches Binden

Polymorphismus ist die Fähigkeit eines Dinges, verschiedene Gestalt anzunehmen. In der Objektorientierung gibt es zwei Formen des Polymorphismus: Datenpolymorphismus und Funktionspolymorphismus.

Datenpolymorphismus ist die Fähigkeit einer Variablen, Objekte verschiedener Klassen aufzunehmen. In ungetypten Sprachen wie Smalltalk ist dies allgemein möglich, d. h. es können beliebige Objekte abgelegt werden. In getypten Sprachen wird der Datenpolymorphismus auf Objekte des Typs der Variablen und dessen Untertypen eingeschränkt, d. h. Variablen von einem Klassentyp können nur Objekte von ihrem Typ oder eines Untertyps aufnehmen.

Funktionspolymorphismus bedeutet, dass verschiedene Operationen denselben Namen tragen (z. B. Überladen von Operationen). Nur anhand des Zielobjekts des Aufrufs und der Parameter kann entschieden werden, welche Methode aufgerufen wird. Ist gleichzeitig Datenpolymorphismus erlaubt, kann wegen möglicher Redefinitionen erst zur Laufzeit entschieden werden, welche Methode aufgerufen wird. Man spricht dann von dynamischer Bindung.

Eine detailliertere Betrachtung des Polymorphismus-Begriffs und eine feinere Unterscheidung der verschiedenen Arten von Polymorphismus findet sich bei Cardelli und Wegner (1985).

3.1.5 Abstrakte Klasse

Eine abstrakte Klasse ist eine Klasse, die nicht instantiiert werden kann. Das liegt daran, dass es Operationen in der Klasse gibt, für die keine Implementierung angegeben wurde (abstrakte Operationen). Weil keine Objekte der Klasse erzeugt werden können, scheinen abstrakte Klassen überflüssig zu sein. Sie sind aber für die Modellierung sinnvoll, weil alle Unterklassen gezwungen sind, die abstrakten Operationen zu implementieren, wenn sie *konkret*, d. h. instantiierbar, sein wollen. Damit legt die abstrakte Klasse fest, über welche Schnittstelle ihre Unterklassen mindestens verfügen sollen. Das gibt Sinn, wenn die abstrakte Klasse ein allgemeines Konzept repräsentiert, für das keine allgemeine Implementierung angegeben werden kann. Die Deklaration einer abstrakten Klasse erzeugt auch einen neuen Typ, so dass (polymorphe) Variablen von diesem Typ deklariert werden können. Instanzen konkreter Unterklassen können einer solchen Variablen zugewiesen werden.

In der UML-Darstellung werden die Namen von abstrakten Klassen und die Signatur abstrakter Operationen kursiv gesetzt, um sie von konkreten Klassen und Operationen abzuheben. Alternativ kann man auch das Stereotyp «abstract» verwenden.

3.1.6 Interface

Ein Interface ist eine spezielle Form der abstrakten Klasse. Es ist völlig abstrakt, d. h. es besitzt keine Attribute und ausschließlich abstrakte Operationen. Damit besteht sein Zweck ausschließlich in der Definition des zugehörigen Typs. Dies wird vor allem eingesetzt, um bestimmte Eigenschaften zu definieren. Beispielsweise wird in der Standardbibliothek der Programmiersprache Java das Interface `Comparable` deklariert. Dieses hat eine abstrakte Operation `compareTo`, die das Objekt mit einem anderen Objekt vergleicht. Eine Klasse, die dieses Interface *realisiert*, d. h. alle Operationen implementiert, besitzt dann die vom Interface definierte Eigenschaft, dass sie über die Vergleichsoperation verfügt.

In UML werden Interfaces wie konkrete Klassen dargestellt, nur dass man zur Unterscheidung das Stereotyp «interface» verwendet. Die Realisierung eines Interfaces durch eine Klasse wird durch einen gestrichelten Vererbungspfeil dargestellt (vgl. Abbildung 3-3). Soll nur der Name des Interfaces angegeben werden, kann es auch durch einen Kreis dargestellt werden („Lollipop-Notation“), der durch eine durchgezogene Linie mit der realisierenden Klasse verbunden ist.

3.1.7 Assoziation, Aggregation und Komposition

One of the distinguishing features of object design is that no object is an island. All objects stand in relationship to others, on whom they rely for services and control.
(Beck, Cunningham, 1989, S. 2)

Die Assoziation ist nach der Vererbung die wichtigste Beziehungsart zwischen Klassen. In ihrer allgemeinsten Form drückt sie aus, dass zwei Klassen (genauer: Instanzen der Klassen) eine Verbindung haben. Diese Verbindung kann unidirektional oder bidirektional sein. In UML wird die Assoziation durch eine durchgezogene Linie dargestellt (vgl. Abbildung 3-4). Durch eine einfache Pfeilspitze am Ende der Verbindungslinie kann zusätzlich die Navigierbarkeit angezeigt werden. Außerdem kann ein Verbindungsende mit Rollenname und Multiplizität versehen werden. Die Multi-

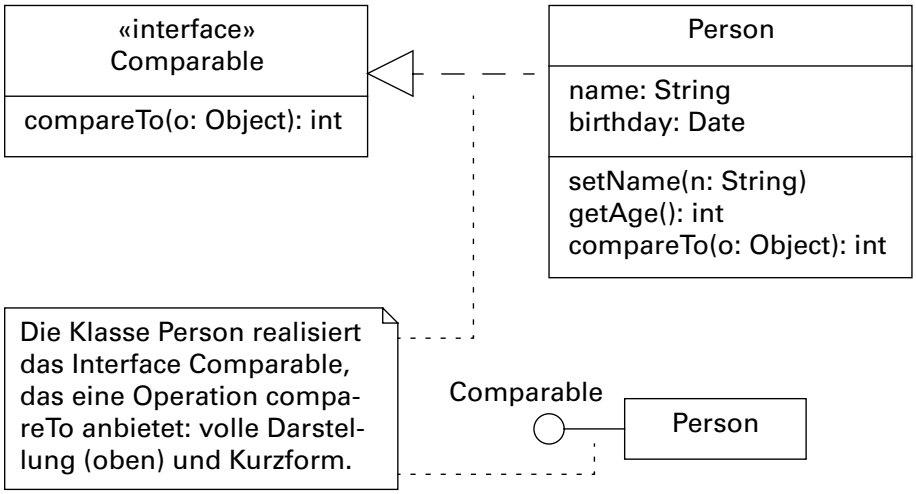

Abbildung 3-3: UML-Darstellung von Interfaces

plizität sagt aus, wie viele Instanzen der Klasse an Assoziationen mit einer bestimmten Instanz der anderen Klasse beteiligt sein können.

Die Aggregation ist eine spezielle Form der Assoziation. Sie zeigt an, dass es sich um eine Beziehung zwischen einem Ganzen und einem Teil davon handelt. Dabei ist nicht verboten, dass ein Teil gleichzeitig an gar keiner oder an mehreren Aggregationen beteiligt ist. Allerdings dürfen Aggregationen nicht zyklisch sein (auch nicht indirekt). In UML wird die Aggregation wie eine Assoziation dargestellt, nur dass beim Ganzen eine leere Raute angebracht ist.

Die Komposition ist ein Spezialfall der Aggregation. Hier ist der Teil fest an das Ganze gebunden; er darf daher auch nur an einer Komposition teilnehmen. Gleichzeitig ist die Lebenszeit des Teils höchstens so lang wie die des Ganzen. Die Komposition besteht während der gesamten Lebenszeit des Teils. Die UML-Darstellung der Komposition ist wie die der Aggregation, nur ist die Raute schwarz gefüllt.

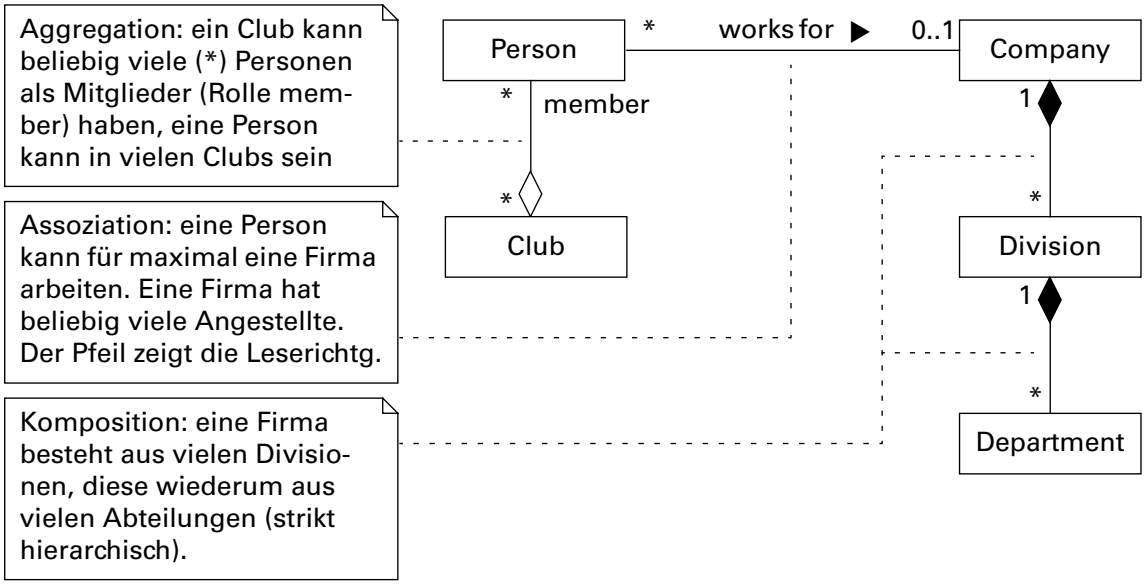

Abbildung 3-4: UML-Darstellung von Assoziationen

3.1.8 Benutzung

Neben den bereits vorgestellten Beziehungen (Vererbung, Assoziation und Realisierung) gibt es noch andere Arten, die in UML unter dem Begriff Abhängigkeit (dependency) subsumiert werden. Eine spezielle Form der Abhängigkeit ist die Benutzung (usage; Stereotyp «use»). Die Darstellung einer Abhängigkeit in UML ist ein gestrichelter Pfeil mit offener Spitze (vgl. Abbildung 3-5). Eine Differenzierung der Benutzung ist durch Vergabe von Stereotypen möglich, z. B. «call» oder «create». Ebenso wie bei der Assoziation sind Benutzungsbeziehungen Beziehungen zwischen Objekten, die aber auf Klassenebene modelliert werden.

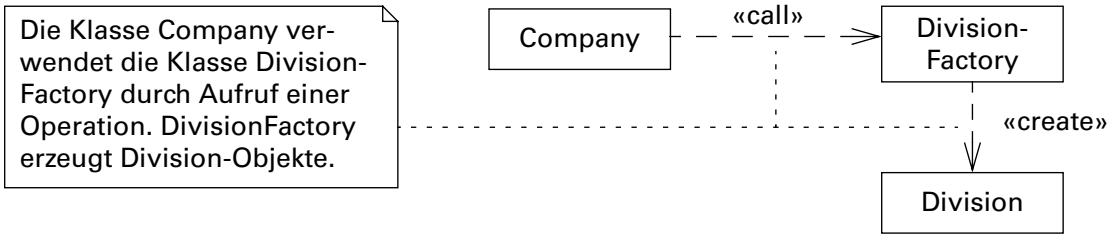

Abbildung 3-5: UML-Darstellung von Benutzungsbeziehungen

3.1.9 Paket

Pakete werden zur Gruppierung von Klassen und Interfaces verwendet. Logisch zusammengehörige Elemente werden in einem Paket zusammengefasst. Pakete können auch Pakete enthalten, so dass sich durch die Schachtelung von Paketen eine Baumstruktur ergibt. Das Gesamtsystem ist (implizit) ebenfalls ein Paket, das direkt oder indirekt alle Elemente enthält.

In UML werden Pakete durch einen Kasten mit einem Reiter dargestellt. Der Paketname wird entweder in den Reiter oder in den Kasten geschrieben. Elemente, die im Paket enthalten sind, werden hineingezeichnet (vgl. Abbildung 3-6).

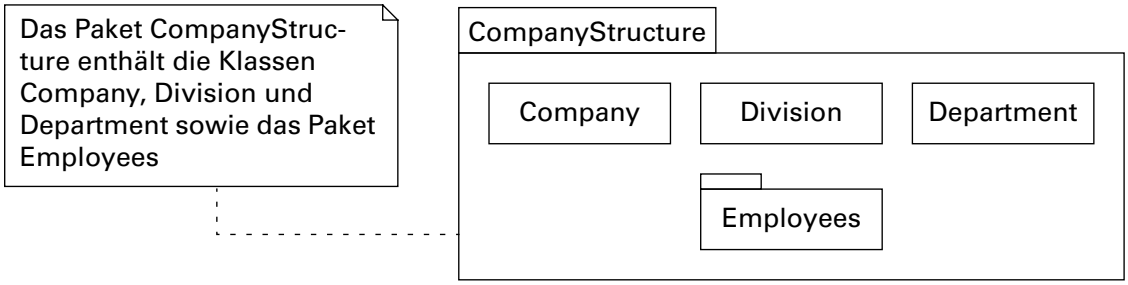

Abbildung 3-6: UML-Darstellung von Paketen

3.2 Unified Modeling Language

Die grammatischen Regeln einer Sprache, die Regeln des Satzbaus z. B., sind auch Vorschriften für die Beschreibung von Situationen. Jemand, der gelernt hat, Situationen nach bestimmten Regeln zu beschreiben, wird auch dazu neigen, Situationen gemäß diesen Regeln wahrzunehmen und zu speichern.
(Dörner, 1976, S. 53)

Bei der Einführung der Begriffe im vorhergehenden Abschnitt wurde bereits die UML-Notation verwendet. Hier soll nun ein kleiner Überblick darüber gegeben werden, was UML ausmacht. Rumbaugh et al. (1998, S. 3) umreißen die Aufgabe der UML wie folgt: „The Unified Modeling Language (UML) is a general-purpose, visual modeling language that is used to specify, visualize, construct, and document the artifacts of a software system.“ Booch et al. (1998) und Oestereich (1998) geben eine ausführliche Einführung in UML. Als Referenz für die Übersetzung von UML-Begriffen ins Deutsche wurde das Glossar von Oestereich (1998) verwendet.

In UML gibt es verschiedene Diagrammtypen (Booch et al., 1998, S. 24ff.):

1. Klassendiagramm (class diagram): zeigt Klassen, Interfaces, Pakete und ihre Beziehungen.
2. Objektdiagramm (object diagram): zeigt Objekte und ihre Beziehungen.
3. Anwendungsfalldiagramm (use case diagram): zeigt Anwendungsfälle, Aktoren und ihre Beziehungen.
4. Sequenzdiagramm (sequence diagram): ein Interaktionsdiagramm, das die zeitliche Ordnung des Austauschs von Nachrichten zwischen Objekten darstellt.
5. Kollaborationsdiagramm (collaboration diagram): ein Interaktionsdiagramm, das die strukturelle Organisation von Objekten, die Nachrichten austauschen, darstellt. Kollaborations- und Sequenzdiagramme sind inhaltlich äquivalent und können ineinander überführt werden, betonen aber unterschiedliche Aspekte.
6. Zustandsdiagramm (state diagram): zeigt einen endlichen Automaten (state machine), der das Verhalten einer Klasse oder eines Interfaces modelliert. Der Automat reagiert auf Ereignisse durch entsprechende Zustandsübergänge.
7. Aktivitätsdiagramm (activity diagram): zeigt den Kontrollfluss zwischen Aktivitäten im System.
8. Komponentendiagramm (component diagram): zeigt die Gliederung und die Beziehungen von Komponenten der Implementierung, z. B. ausführbare Programme, Bibliotheken und Dateien.
9. Verteilungsdiagramm (deployment diagram): zeigt die Verteilung von Komponenten auf Rechnerknoten zur Laufzeit.

Jeder Diagrammtyp liefert eine spezielle Sicht auf das modellierte System, die für bestimmte Gesichtspunkte besonders gut geeignet ist. Jede Sicht für sich allein genügt nicht, um das System vollständig zu beschreiben. Durch die Kombination aller Sichten lässt sich das System jedoch ausreichend festlegen. Man braucht um so mehr verschiedene Sichten, je komplexer das System ist (Budgen, 1994).

Graphische Notationen können in der Regel leichter erfasst werden als textuelle, sind dafür aber häufig nicht so exakt. Das Layout spielt bei graphischen Notationen wie UML für die Verständlichkeit eine wesentliche Rolle – das Layout ist fast so wichtig wie der Inhalt.

Kapitel 4

Objektorientierter Entwurf

In design, object orientation is both a boon and a bane. Object orientation is a boon because it allows a designer to hide behind the scenic walls of encapsulation such software eyesores as: convoluted data structures, complex combinatorical logic, elaborate relationships between procedures and data, sophisticated algorithms, and ugly device drivers.
Object orientation is also a bane because the structures that it employs (such as encapsulation and inheritance) may themselves become complex. In object orientation, it's all too easy to create a Gordian hammock of inextricable interconnections that either is unbuildable or will result in a system that runs like a horse in a sack race.
(Page-Jones, 1995, S. 61)

Dieses Kapitel beschäftigt sich mit verschiedenen Aspekten des objektorientierten Entwurfs (object-oriented design, OOD). Es wird definiert, was Entwurf ist und welche Arten von Entwurf es gibt. Für den Entwurf wichtige Techniken wie Muster und Rahmenwerke werden vorgestellt und es wird kurz auf die wesentlichen Eigenschaften der Entwurfsdokumentation eingegangen. Abschließend werden verschiedene Probleme diskutiert, die das Entwerfen schwer machen.

4.1 Was ist Entwurf?

4.1.1 Definition und Abgrenzung

Der Begriff Entwurf (oder Design) hat zwei verschiedene Bedeutungen. Zum einen bezeichnet er die (äußere) Gestaltung oder Formgebung eines Gegenstands; bei Software entspricht das vor allem der Gestaltung der Benutzungsoberfläche (user interface design). Diese Tätigkeit ist Teil der Spezifikationsphase. Winograd et al. (1996) beschäftigen sich mit dieser Art des Entwurfs.

Hingegen versteht man bei der Software-Entwicklung unter Entwurf vornehmlich die Phase, in der aus der Problemstruktur, die in der Anforderungsspezifikation beschrieben ist, eine Lösungsstruktur abgeleitet wird. Die Tätigkeiten Spezifikation und Entwurf lassen wie folgt voneinander abgrenzen: Der Spezifikation liegt die Frage „*Was*

soll das System leisten?“ zugrunde, während der Entwurf die Frage „*Wie* soll das System das tun, was es leisten soll?“ beantwortet.

Der IEEE Standard 610.12-1990 definiert den Begriff Entwurf wie folgt:

Definition 4-1 (design, Std. IEEE 610.12-1990)

(1) The process of defining the architecture, components, interfaces, and other characteristics of a system or component.
(2) The result of the process in (1)

Die Definition weist auf die doppelte Belegung des Begriffs Entwurf hin: Auch das Ergebnis der Entwurfsphase wird als Entwurf bezeichnet. In der Regel handelt es sich dabei um ein Dokument, die Entwurfsbeschreibung. Dazu lautet die Definition des IEEE Standard 610.12-1990:

Definition 4-2 (design description, IEEE Std. 610.12-1990)

A document that describes the design of a system or component. Typical contents include system or component architecture, control logic, data structures, input/output-formats, interface descriptions, and algorithms.

Abbildung 4-1 zeigt die Einbettung des Entwurfs in die Software-Entwicklung. Der Entwurf transformiert die Anforderungsspezifikation in eine Entwurfsbeschreibung, die in der Implementierungsphase in Code für ein ausführbares Programm umgesetzt wird, der danach getestet und gewartet wird.

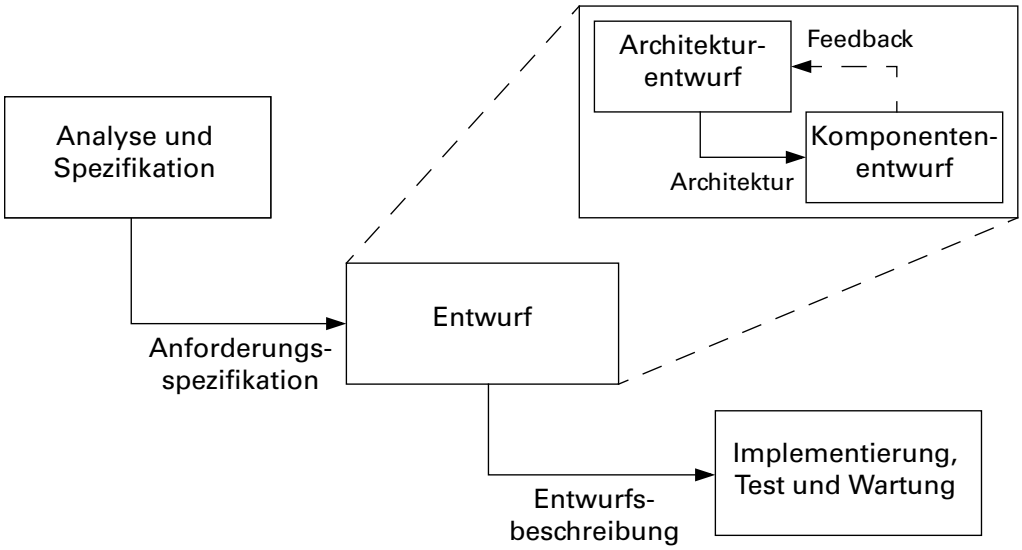

Abbildung 4-1: Einbettung des Entwurfs in die Software-Entwicklung

Die Entwurfsbeschreibung ist ein wichtiges Dokument für alle anderen nachfolgenden Phasen. In der Implementierungsphase dient der Entwurf als Vorgabe, in der Wartung wird er für das Verstehen der Implementierung benötigt. Innerhalb des Entwurfs können die Aktivitäten Architekturentwurf und Komponentenentwurf unterschieden werden (vgl. dazu Abschnitt 4.2.2).

Fehlt der Entwurf, nimmt der Aufwand in der Implementierungs- und Testphase zu, weil bei der Implementierung implizit doch ein ad-hoc-Entwurf stattfindet, dessen Ergebnis dauernd überarbeitet werden muss. Die Wartung führt zu wilden Wuche-

rungen im Code, weil eine geplante Struktur völlig fehlt. Änderungen sind schwierig umzusetzen, da nicht klar ist, welche Teile des Systems betroffen sind und welche Auswirkungen die Änderungen auf andere Teile haben können.

4.1.2 Entwurfsprozess

Design is fundamentally social and fundamentally creative.
(Berg et al., 1995, S. 61)

Die Anforderungsspezifikation ist nicht das Einzige, was den Entwurfsprozess steuert (vgl. Abbildung 4-2). Hinzu kommen noch Entwurfseinschränkungen (z. B. Mangel an Ressourcen oder an Erfahrung mit einer bestimmten Technologie) und die Entscheidungen des Entwerfers, die durch sein Wissen und seine Erfahrung bestimmt sind. Auch die vorhandene Infrastruktur hat Einfluss auf den Entwurfsprozess: Beispielsweise spiegelt die Struktur des Entwurfs oft die Struktur der entwickelnden Organisation wieder (vgl. Abschnitt 4.5.2).

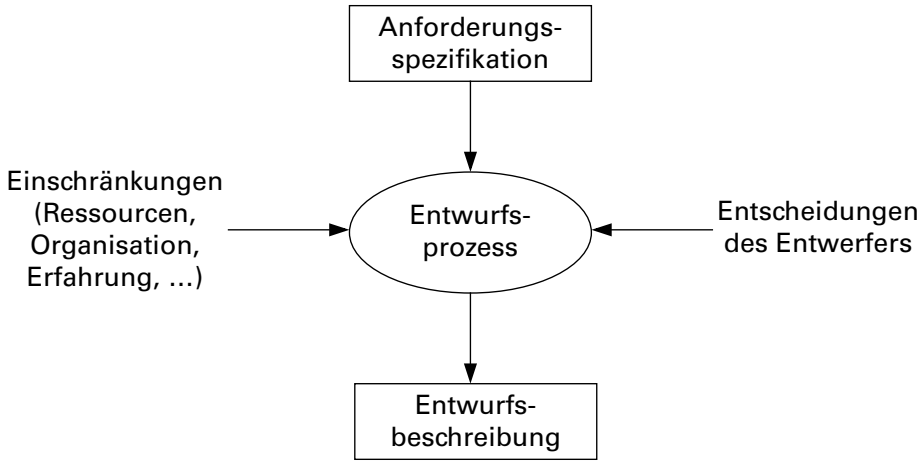

Abbildung 4-2: Modell des Entwurfsprozesses (nach Budgen, 1994, S. 27)

Nach Jones (1992) zerfällt der Entwurfsprozess in drei Phasen:

1. Divergenz (Analyse): Problemanalyse, Suchraum definieren
2. Transformation (Synthese): Generieren von Lösungsalternativen innerhalb des Suchraums
3. Konvergenz (Evaluation): Bewertung der Lösungsalternativen und Auswahl einer Lösung

Diese Phasen werden im Folgenden genauer beschrieben.

Divergenz

Das Problem wird analysiert und in seine Bestandteile zerlegt. Anschließend wird ein Suchraum für die möglichen Lösungen aufgespannt. Dies geschieht in der Regel implizit im Kopf des Entwerfers. Der Suchraum kann aber auch explizit dargestellt werden. Shaw und Garlan (1996, S. 97ff.) beschreiben dazu das Konzept des Entwurfsraums (design space). Er enthält die Menge aller Entwurfsmöglichkeiten zu einer Menge gegebener Anforderungen. Die Dimensionen des Entwurfsraums reflektieren alternative Anforderungen, Entwurfsentscheidungen (z. B. bestimmte Architektur-

muster) und Bewertungskriterien (z. B. hinsichtlich Funktion oder Leistung); sie sind in der Regel nicht voneinander unabhängig. Ein konkreter Entwurf wird durch einen Punkt im Entwurfsraum repräsentiert.

Transformation

Innerhalb des in der ersten Phase generierten Suchraums wird nach Lösungsmöglichkeiten gesucht, die den Anforderungen entsprechen. Im Entwurfsraum werden die Entwurfsmöglichkeiten – in der Regel nur die sinnvollen Alternativen – durch Einordnung auf den einzelnen Dimensionen positioniert. Indem bisher unbesetzte Bereiche im Entwurfsraum betrachtet werden, können häufig weitere Alternativen gefunden werden, die bisher übersehen wurden. Untersucht man die Korrelationen zwischen den Dimensionen – insbesondere zwischen Entwurfsentscheidungen und Bewertungen – kann die Zahl der sinnvollen Alternativen schnell eingeschränkt werden, was die Suche beschleunigt.

Konvergenz

Die Lösungsalternativen der vorhergehenden Phase werden bewertet, und die Lösung mit der besten Bewertung wird ausgewählt. Durch Einsatz von Checklisten kann bei der Bewertung sichergestellt werden, dass alle erforderlichen Kriterien erfüllt sind.

Die Bewertung der Alternativen kann auch auf quantitativer Basis durchgeführt werden, z. B. durch Vergabe von Nutzwerten für Bewertungskriterien (Nutzwertanalyse). Ein solches Vorgehen ist allerdings nicht unproblematisch, wie Jones (1992, S. 381) verdeutlicht: „All one is doing in ranking or weighting a set of objectives that cannot be otherwise compared is to conceal information about each objective that may well be useful in reaching a decision. Totals arrived at by ranking and weighting mislead because scraps of information are being abstracted from reality and fitted together into arithmetical relationships that are probably different from the relationships in practice.“ Bei einer Nutzwertanalyse ist also sicherzustellen, dass alle relevanten Aspekte berücksichtigt und Aspekte korrekt gewichtet sind, um keine irreführenden Resultate zu erhalten.

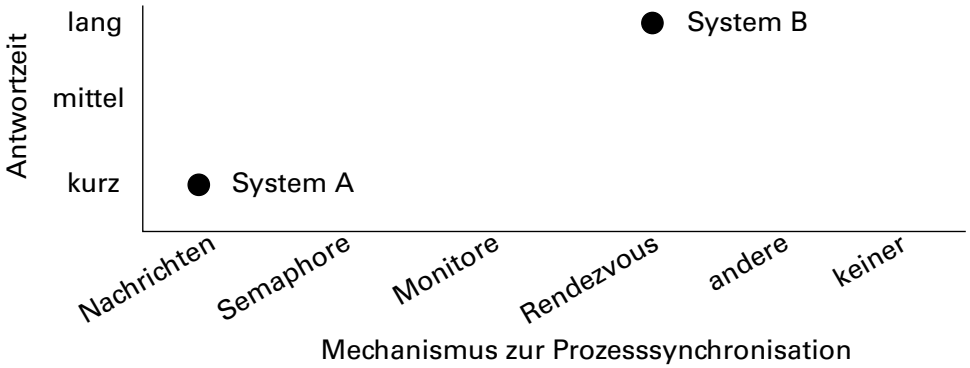

Abbildung 4-3: Beispiel für einen zweidimensionalen Entwurfsraum

Abbildung 4-3 zeigt einen zweidimensionalen Entwurfsraum, der zwei Dimensionen hat: Mechanismus zur Prozesssynchronisation und Antwortzeit. Der Mechanismus repräsentiert eine Entwurfsentscheidung, die Antwortzeit ein Bewertungskriterium.

In diesem Entwurfsraum sind zwei Systeme A und B positioniert worden. Man sieht anhand der Lücken in der Darstellung, dass einige Entwurfsalternativen bisher nicht betrachtet und eingeordnet wurden. Außerdem ist sofort ersichtlich, welches das bessere System hinsichtlich der Antwortzeit ist.

Bemerkungen

Der tatsächliche Entwurfsprozess ist nicht so geradlinig wie gerade dargestellt. Der Entwurf verläuft eher ungeordnet: Zum einen muss das Wissen über Anforderungen und Entwurfskontext (z. B. Technologien) vom Entwerfer erst einmal aufgenommen werden. Zum anderen sind viele Entwurfsziele gegeneinander abzuwägen, so dass man leicht etwas übersieht und Überarbeitungen notwendig werden. Der Entwurfsprozess verläuft daher in der Praxis iterativ und opportunistisch[1] (Robbins, 1998).

Eine Automatisierung des Entwurfsprozesses ist (im Allgemeinen) unmöglich. Einzig bei der Konvergenzphase ist die Automatisierung mit Hilfe eines Computers überhaupt denkbar, denn die ersten beiden Phasen des Entwurfsprozesses sind hochgradig kreativ (Brooks, 1987). Damit sind aber gerade die schwierigsten Phasen des Entwurfs nicht automatisierbar (Glass, 1999).

4.2 Klassifikationen des Entwurfs

In diesem Abschnitt werden verschiedene Aspekte betrachtet, nach denen sich der Entwurf klassifizieren lässt: Strategie, Aktivität, Abstraktionsebene und Struktur.

4.2.1 Strategien

Die Strategie bestimmt, auf welche Weise der Entwurf angegangen wird. Hier wird die historische Entwicklung hin zur objektorientierten Strategie grob chronologisch dargestellt. Der Stand der Praxis ist dem Stand der Wissenschaft in der Regel immer einige Jahre hinterher. Eine Übersicht von Entwurfstechniken findet sich bei Yau und Tsai (1986).

Kein Entwurf/Impliziter Entwurf

Zu Anfang wurden nur kleine Programme in Maschinencode oder Assembler geschrieben. Der Entwurf, sofern vorhanden, war im Wesentlichen das Programm in einem Pseudocode, der einer höheren Programmiersprache entsprach. Der Bedarf für einen expliziten Entwurf auf einer höheren Abstraktionsebene war nicht vorhanden oder nicht klar. Im Zusammenhang mit der Softwarekrise wurde deutlich, dass ein expliziter Entwurf vorteilhaft ist, wenn komplexere Programme erstellt werden.

Strukturierter Entwurf

Die Aufteilung des Systems wird vorgenommen anhand von Funktionen (funktionsorientierter Entwurf). Die Granularität des Entwurfs ist die Prozedur. Datenstrukturen (Records, Listen etc.) dienen zur Modellierung der Daten. Eingeführt wurden

1. Opportunistisch bedeutet hier, dass der Entwerfer beim Entwerfen als nächste Aktion diejenige mit der geringsten kognitiven Schwierigkeit auswählt. Die Schwierigkeit hängt vom Hintergrundwissen des Entwerfers, der verfügbaren Information und der Komplexität der Aufgabe ab.

auch die Ideen der Abstraktion, der Hierarchie und der Schichten (z. B. Constantine, 1965; Dijkstra, 1968).

Modulorientierter/Objektbasierter Entwurf

Das System wird in Module aufteilt. Unter einem Modul verstand man ursprünglich (beim strukturierten Entwurf) eine Prozedur. Beim modulorientierten Entwurf fasst man, beeinflusst durch das Geheimnisprinzip und die Theorie der abstrakten Datentypen, Prozeduren und Datenstrukturen zu größeren Einheiten (Modulen) zusammen. Module, die Datenstrukturen mit den notwendigen Funktionen auf diesen Datenstrukturen zusammenfassen und kapseln, werden auch als Objekte bezeichnet; man spricht dann von objektbasiertem Entwurf (z. B. Booch, 1987).

Objektorientierter Entwurf

Der objektorientierte Entwurf nimmt zum objektbasierten Entwurf noch das Konzept von Vererbung und den damit zusammenhängenden Polymorphismus hinzu. Damit kann der objektorientierte Entwurf als eine Weiterentwicklung der bisher verfolgten Entwurfsstrategien angesehen werden. Allerdings scheint dennoch ein Umdenken beim Entwerfen erforderlich zu sein, weshalb auch häufig von einem Paradigmenwechsel die Rede ist.

4.2.2 Aktivitäten

As systems become more complex, the design problem goes beyond the algorithms and data structures of the computation: designing and specifying the overall system structure emerges as a new kind of problem.
(Jacobson et al., 1998, S. 62)

Wie bereits in Abbildung 4-1 gezeigt können in der Entwurfsphase zwei verschiedene Aktivitäten unterschieden werden: Architekturentwurf und Komponentenentwurf.

Architekturentwurf

Der Architekturentwurf entwickelt die grobe Struktur der Lösung, die Architektur (zum Begriff Architektur siehe Abschnitt 4.3.1). Die wesentlichen Komponenten sind dabei in der Regel der funktionale Kern, die Benutzungsoberfläche und die Datenhaltung. Außerdem wird die Verteilung der Komponenten auf Rechnerknoten festgelegt.

Das Vorgehen beim Architekturentwurf ist wie folgt: Das System wird zunächst hierarchisch in Subsysteme zerlegt, die verschiedene Aufgaben innerhalb des Systems wahrnehmen. Diese Subsysteme werden verfeinert, bis man zu den Atomen des Architekturentwurfs, den Komponenten, gelangt. Die Beziehungen zwischen den Komponenten, die Konnektoren, werden identifiziert. Dokumentiert werden schließlich die Subsysteme, die Komponenten, die Konnektoren und ihre Interaktion zur Laufzeit.

Komponentenentwurf

Der Komponentenentwurf legt zunächst die Schnittstellen der Komponenten nach außen fest. Außerdem werden wichtige Details für die Implementierung bestimmt, vor allem die zu verwendenden Algorithmen und Datenstrukturen (Feinentwurf).

In der Praxis entstehen Architektur- und Komponentenentwurf eher parallel als sequentiell, da beim Komponentenentwurf häufig Fehler oder Schwächen im Architekturentwurf identifiziert werden; insbesondere müssen häufig andere Komponenten angepasst werden, mit denen die Komponente interagieren soll.

4.2.3 Abstraktionsebenen

Anhand des Abstraktionsgrads des Entwurfs kann zwischen logischem und physischem Entwurf unterscheiden werden.

Logischer Entwurf

Der logische Entwurf abstrahiert vom konkreten Kontext der geplanten Implementierung, z. B. der Plattform (Rechner, Betriebssystem, Netzwerke) und anderen Systemen, mit denen das System interagieren soll (z. B. Datenbank, Middleware). Auf diese Weise entsteht ein implementierungsneutraler Entwurf, der in dieser Form nicht direkt implementierbar ist.

Physischer Entwurf

Die Brücke vom logischen Entwurf zur Implementierung schlägt der physische Entwurf. Die im logischen Entwurf offen gelassenen Realisierungsentscheidungen werden auf der Basis von Anforderungen durch Kunden und Management, Kostenaspekten (z. B. Anschaffungs- und Betriebskosten, Einarbeitungszeit der Entwickler und Benutzer) und Effizienzüberlegungen (Laufzeit und Speicherbedarf) gefällt.

Der Vorteil der Trennung zwischen logischem und physischem Entwurf ist, dass sich der Entwerfer zunächst keine Gedanken um Implementierungsprobleme und Effizienz machen muss, sondern sich auf die Aufteilung der Funktionalität auf Komponenten konzentrieren kann.

4.2.4 Strukturen

An object-oriented program's run-time structure often bears little resemblance to its code structure. The code structure is frozen at compile-time, it consists of classes in fixed inheritance relationships. A program's run-time structure consists of rapidly changing networks of communicating objects. In fact, the two structures are largely independent. Trying to understand one from the other is like trying to understand the dynamism of living ecosystems from the static taxonomy of plants and animals, and vice versa.
(Gamma et al., 1995, S. 22)

Beim objektorientierten Entwurf kann zwischen statischer und dynamischer Struktur unterschieden werden. Die statische Struktur beschreibt den Aufbau des Programms zur Übersetzungszeit, während die dynamische Struktur den Aufbau des Programms zur Laufzeit beschreibt.

Statische Struktur

Die statische Struktur besteht aus Klassen, Interfaces und Paketen sowie ihren Beziehungen untereinander. Zu jeder Klasse und zu jedem Interface werden Attribute und Operationen (ggf. auch Redefinitionen) angegeben. Zwischen Klassen und Interfaces können verschiedene Beziehungen bestehen: Es lassen sich Benutzungsbeziehung,

Vererbungsbeziehung, Realisierung und Assoziation (mit ihren Spezialfällen Aggregation und Komposition) unterscheiden (vgl. Abschnitt 3.1). Die Pakete dienen vor allem der hierarchischen Strukturierung der Klassen und Interfaces.

Die statische Struktur wird in UML vor allem mit Klassendiagrammen beschrieben. Bei einer objektorientierten Implementierung stimmt die Codestruktur in hohem Maße mit der statischen Struktur überein, da der objektorientierte Code ebenfalls klassenorientiert ist.

Dynamische Struktur

Die dynamische Struktur entsteht beim Ablauf des Programms. Sie wird vor allem durch die vorhandenen Objekte geprägt, die Instanzen der Klassen aus der statischen Struktur sind. Die Objekte haben Beziehungen, schicken einander Nachrichten, ändern ihren Zustand, erzeugen neue Objekte oder zerstören vorhandene. Die Verteilung der Objekte auf unterschiedliche Rechnerknoten spielt für die dynamische Struktur ebenfalls eine Rolle. Jeder Zustand des Systems während der Ausführung ist eine Ausprägung der statischen Struktur.

Im Gegensatz zur statischen Struktur ist die dynamische Struktur dem Code nur sehr schwer zu entnehmen. Zum einen besteht sie aus einer Vielzahl von Objekten, die untereinander in den unterschiedlichsten Beziehungen stehen. Das Objekt-Netzwerk ist sehr komplex und verändert sich laufend über die Zeit, so dass es nur in Ausschnitten verstanden werden kann. Zum anderen ist die Funktion über den Code verstreut. Soloway et al. (1988) nennen diese Verstreuung logisch zusammengehöriger Teile einer Funktion über die Implementierung „delocalized plans“. Die Delokalisierung der Funktion macht das Verstehen eines objektorientierten Programms ausgesprochen schwierig, da zu einem Zeitpunkt ein Leser des Codes immer nur einen Teil des Gesamtbildes überblicken kann.

Ursache für die Delokalisierung ist, dass die Lösung datenorientiert erstellt wird. Dies ist sinnvoll, um eine bessere Abstraktion (im Sinne abstrakter Datentypen) zu erreichen. Durch die Fokussierung auf die Daten wird aber die Funktion in kleine Teile zerlegt, die sich den jeweiligen Daten zuordnen lassen. Die Verteilung der Funktion wird durch Vererbung noch verstärkt, weil man die Methoden zu den Operationen einer Klasse nicht an einem Ort findet, sondern sich diese aus der gesamten Vererbungshierarchie zusammensuchen muss. Dynamisches Binden erschwert zusätzlich eine statische Analyse des Codes, da oft nicht klar ist, welche Methode tatsächlich aufgerufen wird.

Gerade weil die dynamische Struktur im Code schlecht dokumentiert ist, muss sie im Entwurf hinreichend beschrieben werden, denn sie wird später zum Verständnis des Codes benötigt werden. Die Dokumentation kann z. B. in Form von Szenarien mit einigen Objekten und Aufrufsequenzen ihrer Methoden erfolgen. In UML dienen zur Beschreibung der dynamischen Struktur vor allem Objektdiagramme, Sequenzdiagramme, Kollaborationsdiagramme, Zustandsdiagramme und Aktivitätsdiagramme.

4.3 Muster und Rahmenwerke

Das erste Ergebnis des Entwurfs ist die Software-Architektur. In den letzten zehn Jahren hat der Bereich der Software-Architektur viel Aufmerksamkeit von Forschung

und Praxis erhalten. Ein Ergebnis war der Gedanke der Muster (patterns), der sowohl auf Architekturebene (Architekturmuster) als auch auf Klassenebene (Entwurfsmuster) angewendet werden kann. Außerdem kamen wiederverwendbare halbfertige Architekturen in Form von Rahmenwerken (frameworks) auf. Diese Begriffe werden im Folgenden näher betrachtet.

4.3.1 Software-Architektur

A software architecture is the development product that gives the highest return on investment with respect to quality, schedule, and cost. This is because an architecture appears early in a product's lifetime. Getting it right sets the stage for everything to come in the system's life: development, integration, testing, and modification. Getting it wrong means that the fabric of the system is wrong, and it cannot be fixed by weaving in a few new threads or pulling out a few existing ones, which often causes the entire fabric to unravel.
(Bass et al., 1998, S. x)

Die Verwendung des Begriffs Architektur als Metapher für die Struktur eines Software-Systems begann in den frühen 60er-Jahren durch Weinberg (Coplien, 1999). Zusammen mit dem Begriff wurde auch anderes Gedankengut übernommen, z. B. das Leitbild der guten Architektur von Vitruv mit den Eigenschaften Schönheit, Brauchbarkeit und Dauerhaftigkeit.

Der IEEE Standard 610.12-1990 definiert die Begriffe Architektur und Architekturentwurf wie folgt:

Definition 4-3 (architecture, IEEE Std. 610.12-1990)

The organizational structure of a system or component.

Definition 4-4 (architectural design, IEEE 610.12-1990)

A collection of hardware and software components and their interfaces to establish the framework for the development of a computer system.

Diese Definition führt den Begriff der Komponente ein, es fehlt allerdings ein wichtiger Aspekt: die Beziehungen zwischen den Komponenten. Dieser Aspekt kommt bei der Definition von Bass et al. (1998, S. 23) hinzu:

Definition 4-5 (architecture, Bass et al., 1998)

The software architecture of a program or computing system is the structure or structures of the system, which comprise software components, the externally visible properties of those components, and the relationships among them.

Die Architektur definiert also eine Struktur, die aus Komponenten und ihren Beziehungen (Konnektoren) besteht. Für die Komponenten und Konnektoren werden die nach außen sichtbaren Eigenschaften, z. B. Attribute und Operationen, spezifiziert. Sie bilden die Schnittstelle.

Die neueste Definition des Architekturbegriffs stammt aus dem IEEE-Standard 1471-2000 zur Beschreibung von Software-Architekturen.

Definition 4-6 (architecture, IEEE Std. 1471-2000)

The fundamental organization of a system embodied in its components, their relationships to each other and to the environment, and the principles guiding its design and evolution.

Neu ist hier die explizite Hinzunahme des Kontextes der Architektur sowie der Leitlinien der Architektur und ihrer Weiterentwicklung (also eine Art „design rationale“). Dies erleichtert das Verständnis und die Weiterentwicklung der Architektur.

4.3.2 Muster

Die Musteridee stammt interessanterweise aus dem Fachgebiet der Architektur. Dort haben Alexander et al. (1977) sie bereits in den 70er-Jahren propagiert (vgl. Lea, 1994). Muster sind erprobte Lösungen für immer wiederkehrende Entwurfsprobleme. Die Beobachtung von Entwerfern hat ergeben, dass sie dazu neigen, eher eigene frühere Lösungen für neue Probleme zu adaptieren, statt völlig neue Lösungen zu erarbeiten (Parnas, 1994). Diese Wiederverwendung spart viel Aufwand und führt zu brauchbaren Lösungen, wenn die wiederverwendete Lösung auf das neue Problem passt. Doch es kommt durchaus vor, dass alte Lösungen für das Problem „zurechtgebogen“ werden, statt nach neuen, besseren Lösungen zu suchen. Das führt dazu, dass das Ergebnis wenig brauchbar ist.

Bei den Mustern werden zum einen die essentiellen Eigenschaften herausgearbeitet, indem vom konkreten Problem abstrahiert wird. Zum anderen wird genau dokumentiert, wann das Muster anwendbar ist und wann nicht. Auf diese Weise können Lösungen wie gewohnt wiederverwendet werden. Durch die Dokumentation der Anwendungsbedingungen ist die Wahrscheinlichkeit einer falschen Verwendung eines Musters aber geringer. Ein Nachteil ist allerdings, dass die Muster wegen ihrer Abstraktheit erst noch für das konkrete Problem ausgeprägt werden müssen. Gute Musterdokumentation gibt aber auch dazu Hinweise.

4.3.3 Architekturstile und Architekturmuster

Bei der Entwicklung einer Architektur kann man sich an Architekturstilen (Shaw, Garlan, 1996) orientieren. Ein Architekturstil (architectural style) kann als Architekturphilosophie aufgefasst werden, er ist so etwas wie eine Meta-Architektur. Shaw und Garlan unterscheiden Architekturmuster (z. B. Model-View-Controller), die allgemeiner Natur sind, und Referenzmodelle (z. B. ISO-OSI 7-Schichtenmodell), die in der Regel für bestimmte Anwendungsfelder gedacht sind. Architekturstile treten häufig kombiniert auf, z. B. auf verschiedenen Abstraktionsebenen der Architektur. Tabelle 4-1 zeigt eine Übersicht über verschiedene Architekturstile.

Die Wahl eines Architekturstils legt den Entwerfer auf eine bestimmte Sichtweise fest, mit der die Problemlösung angegangen wird. Es werden Komponenten- und Konnektorentypen sowie ihre Kombinationsregeln vorgegeben. Innerhalb der Sichtweise kreiert der Entwerfer seine Architektur anhand der vorgegebenen „Spielregeln“. Das hat den Vorteil, dass dem Entwerfer ein Rahmen zu Verfügung gestellt wird, an dem er sich orientieren kann. Allerdings ist jetzt die Auswahl eines passenden Architekturstils entscheidend: Der Stil muss sich für das Problem eignen. Ansonsten entsteht zusätzlicher Aufwand, um das Problem dem Stil anzupassen. Bass et al. (1998, Kap. 5) geben einige Hinweise, welcher Stil sich für welche Probleme eignet.

Kategorie	Beispiele
Datenflusssysteme	Sequentielle Batch-Dateien
	Pipes und Filter
Aufruf-und-Rückkehr-Systeme	Hauptprogramm und Unterprogramme
	Objektorientiertes System
	Hierarchische Schichten
Unabhängige Komponenten	Kommunizierende Prozesse
	Verteilte Systeme (z. B. Client/Server, Multi-Tier)
	Ereignisgesteuertes System
Virtuelle Maschinen	Interpreter
	Regelbasiertes System
Datenzentriertes System (Repository)	Datenbank
	Hypertext-System
	Schwarzes Brett (Blackboard)

Tabelle 4-1: Beispiele für Architekturstile (nach Shaw und Garlan, 1996, S. 20)

4.3.4 Entwurfsmuster

Entwurfsmuster (design patterns) definieren Mikro-Architekturen, d. h. kleinere Einheiten innerhalb einer Architektur. Sie bieten abstrakte Lösungen für das Zusammenspiel weniger Komponenten an. Häufig geht es um die Entkopplung einzelner Komponenten, um eine bessere Änderbarkeit zu erreichen. Tabelle 4-2 zeigt eine Übersicht über die Muster des ersten und bekanntesten Buchs über Entwurfsmuster von Gamma et al. (1995).

Zweck → Ebene ↓	Objekterzeugung	Strukturmodellierung	Verhaltensmodellierung
Klasse	Factory Method	Adapter (class)	Interpreter
			Template Method
Objekt	Abstract Factory	Adapter (object)	Chain of Responsibility
	Builder	Bridge	Command
	Prototype	Composite	Iterator
	Singleton	Decorator	Mediator
		Facade	Memento
		Flyweight	Observer
		Proxy	State
			Strategy
			Visitor

Tabelle 4-2: Entwurfsmuster von Gamma et al. (1995, S. 10)

Es gibt inzwischen umfangreiche Literatur zum Thema Entwurfsmuster. Sammlungen von Entwurfsmustern finden sich bei Gamma et al. (1995), Buschmann et al. (1996), Coplien und Schmidt (1995), Vlissides et al. (1996), Martin et al. (1998) und Harrison et al. (2000a). Eine Musterbibliographie stammt von Rising (2000).

4.3.5 Rahmenwerke

Ein Architekturstil (oder eine Kombination von Architekturstilen) liefert noch keine vollständige Architektur, sondern nur ein abstraktes Architekturgerüst, das vom Entwerfer zu konkretisieren ist. Einen Schritt weiter gehen die Rahmenwerke. Ein Rahmenwerk ist ein unvollständiges, konkretes Software-System (oder -Subsystem), das die Architektur größtenteils vorgibt. Es kann zur Erstellung einer Familie von Systemen verwendet werden, definiert damit also eine Art Makro-Architektur. Rahmenwerke lassen sich häufig nur schwer kombinieren, da sie in der Regel nicht darauf ausgelegt sind (Mattsson et al., 1999).

Ein Rahmenwerk hat Stellen, an denen Anpassungen für das konkrete System gemacht werden sollen (so genannte „hot spots“ oder „hooks“). Hier wird der anwendungsspezifische Teil des Systems „angekoppelt“. Pree (1995) beschreibt die Grundidee von Rahmenwerken und die grundsätzliche Realisierung von hot spots anhand so genannter Metamuster (metapatterns). Fayad et al. (1999) legen die Grundlagen für objektorientierte Rahmenwerke ausführlich dar.

Ein bekanntes Beispiel für ein Rahmenwerk ist ET++ (Weinand et al., 1989; Weinand, Gamma, 1994), ein in C++ implementiertes Rahmenwerk für dokumentenzentrierte Anwendungen mit einer graphischen Benutzungsoberfläche. Viele der Entwurfsmuster von Gamma et al. (1995) entstammen der Arbeit an ET++. Rahmenwerke und Entwurfsmuster sind ungefähr zeitgleich mit der zunehmenden Verbreitung der objektorientierten Entwicklung entstanden. Sie sind daher vor allem für diesen Ansatz entwickelt worden. Grundsätzlich sind sie aber nicht auf die Objektorientierung beschränkt.

Der Einsatz von Architekturstilen, Rahmenwerken und Entwurfsmustern während des Entwurfs kann den Aufwand der Erstellung reduzieren und gleichzeitig die Qualität verbessern, da erprobte Lösungen wiederverwendet werden. Allerdings muss der Entwerfer die Muster bereits kennen und wissen, wann ein Einsatz angebracht ist und wann nicht. Der erhöhten Produktivität und Qualität steht also ein beträchtlicher Initialaufwand gegenüber.

4.4 Dokumentation des Entwurfs

Die Entwurfsdokumentation spielt in Implementierung und Wartung eine bedeutende Rolle. Daher sind Vollständigkeit, Konsistenz und Verständlichkeit wichtig. Für die Dokumentation sollten, soweit möglich, ein Standardaufbau und Standardnotationen verwendet werden.

Dokumentiert werden soll vor allem das Ergebnis des Entwurfs, nicht der Entwurfsprozess. Der Prozess ist eher chaotisch als geordnet (vgl. Abschnitt 4.1.2); dennoch sollte die Dokumentation so aussehen, als sei der Entwurf das Ergebnis einer wohlüberlegten Vorgehensweise (Parnas, Clements, 1986). Neben der ausgewählten

Lösung sollten allerdings auch die betrachteten Alternativen vorgestellt werden und es sollte begründet werden, warum man sich für die gewählte Alternative entschieden hat.

Die Beschreibung der Lösung muss ausführlich genug sein, dass Arbeitspakete für die Implementierung gebildet und Entwickler die Lösung implementieren können. Beim objektorientierten Entwurf wird die Dokumentation eine Beschreibung der Architektur aus dem Architekturentwurf enthalten, ebenso Beschreibungen der einzelnen Klassen mit ihren Schnittstellen und Interaktionen. Hinzu kommen die Implementierungsvorgaben aus dem Komponentenentwurf.

Der IEEE Standard 1471-2000 empfiehlt, eine Software-Architektur aus verschiedenen Sichten zu beschreiben. Dies hat auch schon Kruchten (1994) mit seinem 4+1-Sichten-Modell zur Architekturbeschreibung vorgeschlagen. Dort unterscheidet er die logische Sicht (Klassenstruktur), Prozess-Sicht (kommunizierende Prozesse), Entwicklungssicht (Organisation in Softwaremodule) und physische Sicht (Verteilung). Die fünfte Sicht wird durch Szenarios gebildet, mit deren Hilfe das Zusammenspiel der anderen vier Sichten überprüft werden kann. Alle Sichten lassen sich mit UML modellieren.

Werden in der Entwicklung Muster verwendet, sollte das dokumentiert werden: Die Dokumentation enthält einen Überblick, welche Muster ausgeprägt wurden und welche Klassen welche Rollen in den Musterausprägungen einnehmen. Auf diese Weise kann sich jemand, der mit den Mustern vertraut ist, schneller mit den grundlegenden Ideen und Strukturen des Entwurfs vertraut machen. Ein gutes Beispiel für eine Entwurfsdokumentation durch Muster ist die des Test-Rahmenwerks JUnit (Beck, Gamma, 1998). Quibeldey-Circel (1999) führt den Gedanken der Dokumentation mit Mustern weiter zum Literate Designing, das die Architektur unter Verwendung von Mustern in Hypertext-Form dokumentiert.

4.5 Probleme des Entwurfs

The best way to learn to live with our limitations is to know them.
(Dijkstra, 1972)

Einen guten Entwurf zu schaffen ist schwierig, da der Entwerfer dabei mit vielen Problemen konfrontiert wird. Dazu gehören:

- unvollständige und instabile Anforderungen,
- ökonomische Zwänge,
- Probleme bei der Beherrschung der Technologie und schließlich
- die fundamentale Komplexität des Entwurfs.

Diese Probleme werden im Folgenden vertieft dargestellt.

4.5.1 Unvollständige und instabile Anforderungen

Even if we could master all of the detail needed, all but the most trivial projects are subject to change for external reasons.
(Parnas, Clements, 1986, S. 251)

Die Anforderungen umreißen das Problem, für das eine Lösung entworfen werden soll. Je unklarer das Problem beschrieben wird, desto schwieriger wird es, eine brauchbare Lösung anzugeben. Wie eine Untersuchung der KPMG (1995) zeigt, sind bei 51% aller gescheiterten Projekte Probleme mit den Anforderungen ein Grund für das Scheitern – es handelt damit sich um den am häufigsten genannten Grund.

Die ideale Spezifikation ist vollständig, konsistent, verständlich und präzise (Ludewig, 1998). Eine solche Spezifikation ist in der Praxis allerdings äußerst selten, was auch daran liegt, dass ihre Erstellung sehr teuer ist. Außerdem bringt jeder Anwendungsbereich einige Grundannahmen mit, die für den Kunden selbstverständlich sind, weshalb sie meistens in der Spezifikation fehlen. Ist der Entwerfer mit dem Anwendungsbereich vertraut, wird er diese impliziten Anforderungen oder Entwurfseinschränkungen (aus seiner Sicht) in den Entwurf einfließen lassen. Verfügt der Entwickler hingegen nicht über Wissen aus dem Anwendungsbereich, ist es unwahrscheinlich, dass sein Entwurf brauchbar sein wird. Schließlich kann erst durch die Implementierung wirklich festgestellt werden, ob die Anforderungen auch realisierbar sind (Boehm, 1976).

In der Praxis hat der Entwerfer auch damit zu kämpfen, dass die Anforderungen instabil sind. Der häufigste Fall ist dabei nicht die Änderung einer bestehenden Anforderung, sondern neu hinzukommende Anforderungen, die sich nach und nach „einschleichen“ (creeping requirements). Änderungen in den Anforderungen ziehen aber unter Umständen tief greifende Änderungen im Entwurf nach sich.

Für den Umgang mit unklaren Anforderungen empfiehlt sich die Erstellung von Prototypen. Bei besonders unklaren oder instabilen Anforderungen ist ein inkrementeller oder evolutionärer Entwicklungsprozess am besten geeignet, bei dem der Entwurf erst nach und nach entsteht und in der Regel mehrfach überarbeitet wird.

4.5.2 Ökonomische Zwänge

Das Entwurfsproblem wird durch ökonomische Vorgaben noch verschärft. Projekte verlaufen fast immer unter Zeitdruck, der auch in der Entwurfsphase präsent ist. Da Nachdenken Zeit erfordert, findet es deshalb oft nicht oder nur in geringem Maße statt. Was auch wegfällt, ist die Dokumentation des Entwurfs und der Überlegungen, die hinter dem Entwurf stehen (design rationale). Fehlt die Dokumentation, ist sie unvollständig oder veraltet, wird sie aber in der Implementierung (und Wartung) nicht verwendet werden, was schließlich zu einer unverständlichen Struktur des Codes führt.

Neben dem Zeitdruck gibt es auch noch den Kostendruck, weshalb die durch den Entwurf verursachten Kosten beachtet werden müssen. Dazu gehören z. B. die Kosten für benötigte Hardware, Software und Personal bei der Implementierung des Entwurfs. Aber auch die Betriebskosten für das System sind zu berücksichtigen, z. B. die Kosten für Anschaffung, Betrieb und Wartung neuer Hardware, Software-Lizenzen, Software-Updates und Patent-Lizenzen oder Folgekosten bei Systemausfällen.

Zur Kostendämpfung wird oft Wiederverwendung empfohlen. Allerdings muss der Entwerfer entscheiden können, wann es sich lohnt, eine bestimmte Komponente wiederzuverwenden. Das ist vor allem dann fraglich, wenn der übrige Entwurf an die

Komponente angepasst werden muss (oder die Komponente an den Entwurf – sofern das überhaupt geht, was man meist erst weiß, wenn man es versucht hat).

Entwerfen ist eine Tätigkeit, die am effektivsten von einer kleinen Gruppe durchgeführt wird. Auf der anderen Seite muss der Projektleiter aus Effizienzgründen alle Mitarbeiter im Team beschäftigen. Das führt oft dazu, dass statt eines richtigen Entwurfs ad hoc eine Aufteilung in Teilsysteme vorgenommen wird, die dann von kleineren Gruppen implementiert werden (DeMarco, 1998, Kap. 19). Der Entwurf entsteht so mehr oder weniger implizit, was auch dazu führt, dass die Entwurfsstruktur und die Organisationsstruktur sich sehr ähnlich sind (dieses Phänomen ist bekannt als Conway's Law; Conway, 1968). Durch ein solches Vorgehen wird aber die Chance verschenkt, einen guten Entwurf (insbesondere mit minimalen Schnittstellen, Parnas, 1972a) zu erstellen. Als Folge ergeben sich unnötige Redundanz und hohe wechselseitige Abhängigkeiten zwischen den Teilsystemen. Dies bedingt eine hohe Kommunikation zwischen den Entwicklergruppen, da häufig über Schnittstellen neu verhandelt werden muss. Auf diese Weise nimmt nicht nur Implementierungsaufwand zu, sondern auch der Wartungsaufwand.

Die empfohlene Vorgehensweise ist deshalb, dass zunächst ein kleines Architekturteam (etwa drei bis fünf Entwickler) in Ruhe einen Architekturentwurf ausarbeitet. Es kann auch sinnvoll sein, einen Chefarchitekten zu bestimmen, der in Zweifelsfällen die letzte Entscheidung trifft und einen einheitlichen Stil durchsetzt. Wenn der Architekturentwurf fertig ist, können Untergruppen gebildet werden, welche die Teilsysteme und Komponenten unter der Aufsicht der Architekturteams fertig entwerfen und dann implementieren. Die Personalausstattung eines Projekts sollte entsprechend angepasst werden. Zu Beginn des Entwurfs wird nur wenig Personal eingesetzt. In der Implementierungsphase wird dann das Personal aufgestockt, weil sich die Arbeit besser verteilen lässt. Notfalls können „überflüssige" Teammitglieder während des Architekturentwurfs auch mit der Ausarbeitung von Testfällen und des Benutzerhandbuchs auf der Basis der Spezifikation beschäftigt werden.

4.5.3 Technologie

Even if we knew the requirements, there are many other facts that we need to know to design the software. Many of the details only become known to us as we progress in the implementation. Some of the things we learn invalidate our design and we must backtrack.
(Parnas, Clements, 1986, S. 251)

Nach der bereits angesprochenen Studie der KPMG (1995) ist neu eingeführte Technologie mit 45% ein weiterer häufig genannter Grund für das Scheitern von Softwareprojekten: „Technology is developing faster than the skills of the developers." Die Entwickler müssen sich zunächst in die Technologie einarbeiten und machen anfangs viele Fehler bei der Anwendung der Technologie. Außerdem sind neue Technologien in der Regel noch nicht ausgereift (das gilt insbesondere für Software-basierte Technologien). Schließlich kann die Technologie auch für das Problem unangemessen oder unbrauchbar sein, was häufig vorher nicht abzusehen ist. Daher stellt der Einsatz einer neuen Technologie beim Entwurf ein nicht zu vernachlässigendes Risiko dar.

Aber auch mit vorhandener Technologie gibt es Schwierigkeiten. Zunächst muss der Entwerfer Wissen über verfügbare Technologien haben, die er einsetzen kann (z. B. Middleware, Standardsoftware, Komponenten, Bibliotheken, Rahmenwerke). Er

muss aber auch abschätzen können, ob sich der Einsatz einer bestimmten Technologie lohnt. Dafür muss die Zukunftssicherheit der Technologien abgeschätzt werden: Im Angesicht raschen Wandels kann es sich dabei nur um Monate handeln. Ist ein solches Risiko identifiziert, der Einsatz der Technologie aber notwendig, können die Teile, die bei einer abzusehenden Portierung geändert werden müssen, im Entwurf flexibel genug gestaltet werden.

4.5.4 Komplexität des Entwerfens

Consciousness about design does not imply the application of a formal, consistent, or comprehensive theory of design or of a universal methodology. Systematic principles and methods at times may be applicable to the process of design, but there is no effective equivalent to the rationalized generative theories applied in mathematics and traditional engineering. Design consciousness is still pervaded by intuition, tacit knowledge, and gut reaction.
(Winograd et al., 1996)

Problemklassen nach Dörner

Ein Problem ist nach Dörner (1976) durch drei Merkmale gekennzeichnet: Es gibt einen unerwünschten Ausgangszustand α, einen erwünschten Zielzustand ω und eine Barriere, welche die Transformation von α in ω verhindert. Fehlt die Barriere, d. h. ist die Transformation bekannt, spricht Dörner von einer Aufgabe. Für die Transformation steht eine Menge von Operatoren o_1, o_2, …, o_n, das so genannte Operatoreninventar O, zur Verfügung. Eine Transformation ist eine Folge von Operationen, d. h. von konkreten Anwendungen der Operatoren. Dörner unterscheidet drei Klassen von Problemen: Syntheseprobleme, Interpolationsprobleme und dialektische Probleme. Diese unterscheiden sich vor allem durch die Art der Barrieren. In Abbildung 4-4 sind diese dargestellt, klassifiziert nach Klarheit der Zielkriterien und Bekanntheitsgrad der Operatoren.

		Zielkriterien	
		klar	**unklar**
Operatoren	**bekannt**	Interpolationsbarriere	dialektische Barriere
	unbekannt	Synthesebarriere	dialektische Barriere und Synthesebarriere

Abbildung 4-4: Dörners Barrierekategorien (nach Dörner, 1976, S. 10)

Abbildung 4-5 veranschaulicht die Unterschiede der verschiedenen Problemklassen und den Unterschied zwischen Problem und Aufgabe.

Interpolationsprobleme

Bei Interpolationsproblemen sind Ausgangs- und Zielzustand klar definiert und die zur Verfügung stehenden Operatoren sind bekannt. Es besteht jedoch eine Interpolationsbarriere, d. h. es ist unklar, wie die Operatoren verwendet und kombiniert werden müssen, damit die gewünschte Transformation von α in ω entsteht. Beispielsweise sind bei der Wegesuche in einem rechtwinkligen Labyrinth Anfangs- und Zielzustand festgelegt. Außerdem sind die möglichen Bewegungen (Operationen) durch das Operatoreninventar „links drehen“, „rechts drehen“ und „geradeaus

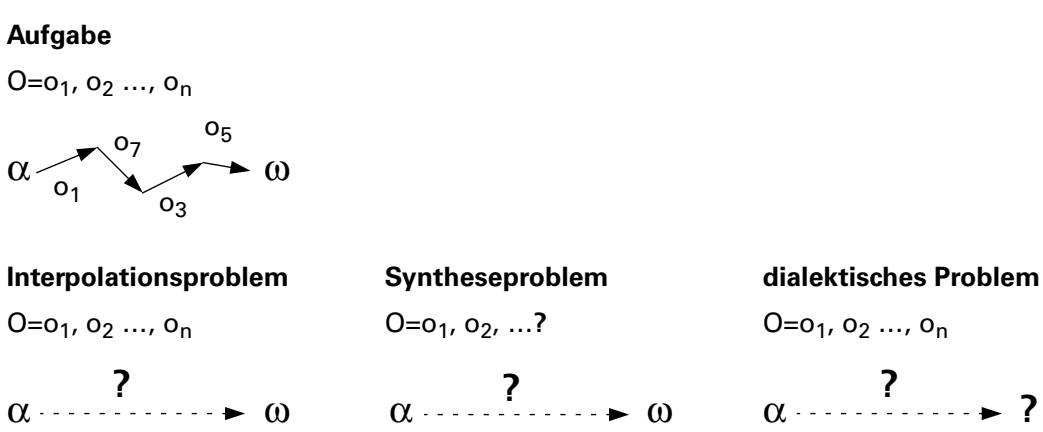

Abbildung 4-5: Aufgabe und Probleme

gehen“ klar definiert. Das Finden eines Wegs durch das Labyrinth (die Transformation) ist aber wegen der vielen Möglichkeiten trotzdem aufwendig.

Syntheseprobleme

Syntheseprobleme unterscheiden sich von Interpolationsproblemen dadurch, dass das Operatoreninventar nicht abgeschlossen ist, d. h. es kann neben den bekannten Operatoren weitere geben. Wenn die bekannten Operatoren für die Lösung nicht ausreichen, müssen zunächst neue Operatoren gefunden oder erfunden (synthetisiert) werden. Synthesebarrieren sind auch deshalb so schwer zu überwinden, weil sie oft einen Wechsel der Blickrichtung auf ein Problem erfordern, was durch individuelle Einstellungen und Denkgewohnheiten erschwert wird. Beispielsweise ist es bei Labyrinthen in Adventure-Spielen oft so, dass man mit den oben genannten Operationen nicht weiterkommt, so lange man nicht eine unerwartete Operation durchführt wie z. B. „brüchige Wand durchbrechen.“

Dialektische Probleme

Bei den dialektischen Problemen schließlich ist der Zielzustand unklar. Meistens können zwar Anforderungen an diesen Zustand formuliert werden, doch sind diese oft widersprüchlich. Es werden daher – mehr oder minder systematisch – verschiedene Transformationen ausprobiert, bis ein Zustand erreicht wird, der den Anforderungen entspricht. Währenddessen entwickelt sich auch eine genauere Vorstellung vom angestrebten Zielzustand, weil widersprüchliche Anforderungen gegeneinander abgewogen werden. Ein Beispiel für ein dialektisches Problem ist der Wunsch „Unser Dorf soll schöner werden.“

Entwurf als Problem

The fundamental problem is that designers are obliged to use current information to predict a future state that will not come about unless their predictions are correct. The final outcome of designing has to be assumed before the means of achieving it can be explored: the designers have to work backwards in time from an assumed effect upon the world to the beginning of a chain of events that will bring the effect about.
(Jones, 1992, S. 9)

Es ist ausgesprochen schwierig, einen guten Entwurf zu schaffen. Visser und Hoc (1990) stufen den Entwurf als schlecht definiertes Problem (ill-defined problem) ein.

Budgen (1994) bezeichnet den Entwurf sogar als ein bösartiges Problem (wicked problem). Bösartige Probleme sind nach Rittel und Webber (1984) unter anderem durch folgende Eigenschaften gekennzeichnet:

- Es gibt keine endgültige Formulierung des Problems. Ein Grund dafür ist, dass sich Spezifikation und Entwurf nicht klar trennen lassen (Swartout, Balzer, 1982).
- Es gibt keine Regel, die angibt, wann die optimale Lösung gefunden wurde. Das liegt daran, dass die Bewertung einer Lösung schwierig ist, weil es (meistens) keine klare Festlegung aller gewünschten Eigenschaften gibt (es fehlt also ein Qualitätsmodell).
- Lösungen für bösartige Probleme sind nicht richtig oder falsch, sondern gut oder schlecht. Es gibt keine wirklich falsche Lösung, nur weniger oder besser brauchbare. Daher ist es schwierig, den Suchraum wirksam einzugrenzen.
- Teilaspekte des Problems können nicht unabhängig voneinander gelöst werden. Die Lösung für einen Teilaspekt des Problems kann neue Probleme in anderen Teilaspekten verursachen oder deren Lösung unmöglich machen.

Beim Entwurf handelt es sich um ein Problem, bei dem sowohl dialektische Barrieren als auch Interpolations- und Synthesebarrieren vorliegen. Der Entwurf gehört also in die „schlimmste“ Problemklasse. Das Operatoreninventar ist hochgradig offen, es sind also fast immer Synthesebarrieren vorhanden. Selbst wenn irgendwann klar ist, welche Operatoren zu verwenden sind, ist ihre konkrete Verwendung immer noch ein Interpolationsproblem. Welche Ansätze ein Entwerfer verfolgt, hängt so von seiner Intuition, seinem Wissen und seiner Erfahrung ab. Dabei kommen auch Gewohnheiten und Denkbarrieren zum Tragen. Ein Ansatz, um das Syntheseproblem in ein Interpolationsproblem zu überführen, ist es, ein möglichst großes Operatoreninventar von Mustern (Architekturmuster, Entwurfsmuster etc.) anzulegen.

Die dialektische Barriere entsteht dadurch, dass die Spezifikation zwar die Kriterien für den Zielzustand vorgibt, es aber immer noch beliebig viele geeignete Zielzustände gibt. Es gibt also eine große Anzahl von Wahlmöglichkeiten, die der Entwerfer beim Entwickeln einer Lösung zu einer gegebenen Menge von Anforderungen hat (Boehm, 1976). Der Entwerfer ist daher gezwungen, verschiedene Alternativen auszuarbeiten. Unter den so entstehenden alternativen Lösungsansätzen ist es schwer zu wählen. An die Frage: Welches ist die beste Alternative? schließt sich gleich die Frage an: Wie können die Alternativen bewertet werden, um statt einer gefühlsmäßigen eine möglichst objektive Antwort zu erhalten? Ein Ansatz dazu ist die Verwendung eines Qualitätsmodells, das die Zielkriterien, die der Entwurf zu erfüllen hat, klar definiert. Dadurch wird das dialektische Problem näher an ein Interpolationsproblem herangebracht.

Entwurfsausbildung

Eine einfache Lösung für den Umgang mit der fundamentalen Komplexität des Entwurfs gibt es nicht. Man kann allerdings bei der Ausbildung der Entwerfer ansetzen, um diese so gut wie möglich auf ihre Aufgabe vorzubereiten. Brooks (1987, S. 18) stellt fest: “Great designs come from great designers.” Nach Brooks wäre es am besten, großartige Entwerfer auszubilden und nur diese einzusetzen. Einen solchen Entwerfer zeichnen aus (Curtis et al., 1988; Visser, Hoc, 1990):

- die Vertrautheit mit dem Anwendungsbereich (siehe auch Adelson, Soloway, 1985; Dvorak, Moher, 1991),
- die Fähigkeit, die technische Vision den anderen Projektmitgliedern mitzuteilen, da der Entwurf oft durch Kommunikation mit anderen entsteht,
- die Identifikation mit dem Projekterfolg und
- eine opportunistische Vorgehensweise, d. h. Entscheidungen zu vertagen, so lange die benötigten Informationen noch nicht vorliegen, und auf Grund vorliegender Informationen zukünftige Entwicklungen (z. B. zu erwartende Erweiterungen) vorwegzunehmen.

Leider haben aber nur wenige Entwickler das Potential, großartige Entwerfer zu werden. Brooks (1987, S. 18) meint: "We can get good designs by following good practices", also ist das wohl der Weg, den gewöhnliche Entwerfer einschlagen sollten. In der Ausbildung sind die guten Praktiken zu lehren, z. B. in Form von Entwurfsregeln. Zusätzlich sollte vermittelt werden, welche bewährten Lösungen es gibt und unter welchen Umständen sie funktionieren, z. B. in Form der bereits erwähnten Muster. Aber auch Lösungen, die nicht funktionieren, sollten dokumentiert und weitergegeben werden (Petroski, 1992, 1994), z. B. in Form von Anti-Mustern (antipatterns; Koenig, 1995; Brown et al., 1998). Das Wissen über mögliche Lösungsansätze sollte eingebettet sein in einen Rahmen, der angibt, welche Eigenschaften ein guter Entwurf hat. Dieser Rahmen, der nichts anderes als ein Qualitätsmodell ist, kann dann bei der Entscheidung herangezogen werden, wann eine der bewährten Lösungen angebracht ist und wann nicht. Pancake (1995, S. 42) schreibt: „The difficulty in both teaching and learning OT [=object technology] is that the quality of an OO design is difficult to evaluate." Das Problem liegt also in der Bewertung. Gerade der Bewertung nimmt sich diese Arbeit aber an.

Erfahrung und Intuition sind für einen guten Entwerfer sehr wichtig (Budgen, 1994). Der Erwerb von Erfahrung ist aber langwierig und teuer: Man muss viele Systeme entwerfen und aus den gemachten Fehlern lernen. In der Regel werden aber die Systeme für reale Projekte entworfen und danach auch implementiert. Das kann zu enormen Fehlerfolgekosten führen, wenn schlecht entworfene Systeme überarbeitet werden müssen. Für den Erwerb der praktischen Erfahrung ist daher zu Beginn ihrer Berufspraxis die Beschäftigung als Lehrling bei einem guten, erfahrenen Entwerfer sinnvoll (McBreen, 2001). Auf diese Weise wird das oben skizzierte Lernen durch Instruktion mit Lernen durch Imitation kombiniert.

Kapitel 5

Ein Referenzmodell für den objektorientierten Entwurf

Um einen objektorientierten Entwurf bewerten zu können, muss man zunächst festlegen, was man genau darunter versteht. Dazu dient das für diese Arbeit neu entwickelte formale Modell, das Object-Oriented Design Model (ODEM) genannt wird. ODEM enthält die Teile des objektorientierten Entwurfs, die als für die Bewertung relevant erachtet werden. Dabei wird von Entwurfsartefakten ausgegangen, die typischerweise vorhanden sind. ODEM kann auch für die formale Definition von Metriken auf diesen Artefakten verwendet werden.[1]

5.1 Grundlagen

Die in Kapitel 3 vorgestellte Entwurfsnotation UML ist der Standard für die Darstellung objektorientierter Entwürfe. Daher baut ODEM auf den Informationen auf, die sich aus einer Entwurfsdarstellung in UML, d. h. einem UML-Modell, gewinnen lassen. UML-Modelle sind Ausprägungen des UML-Metamodells – hier wird das UML-Metamodell aus der Version 1.3 des Standards der OMG (OMG, 2000a) verwendet. Das UML-Metamodell dient zur formalen Definition der UML. Interessanterweise ist das UML-Metamodell wiederum mit UML definiert, wir haben es also mit einer rekursiven Definition zu tun. Weil UML eine Notation für objektorientierte Analyse und Entwurf ist, sind die Elemente des UML-Metamodells als Klassen modelliert. Die Namen der Klassen entsprechen dabei den UML-Begriffen.

Abbildung 5-1 und Abbildung 5-2 zeigen die für ODEM relevanten Ausschnitte aus dem UML-Metamodell. In Abbildung 5-1 liegt der Schwerpunkt auf den Modellelementen wie Klassen und Paketen, während in Abbildung 5-2 der Schwerpunkt auf den Beziehungen wie Vererbung oder Assoziation liegt.

1. Odem, ein gehobenes Wort für Atem, ist dafür ein passender Name: Schließlich ist ein formales Modell als Grundlage für die Definition von Metriken genauso essenziell wie Atem für das Leben.

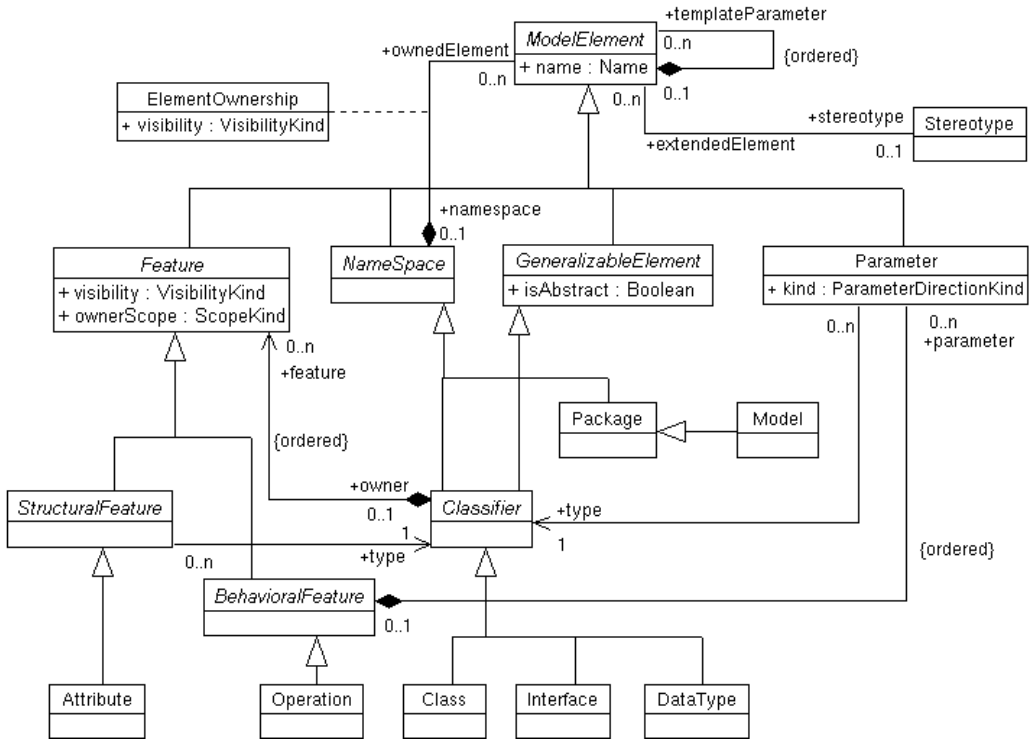

Abbildung 5-1: UML-Metamodell: Modellelemente (Ausschnitt)

Die gemeinsame Oberklasse aller Modellelemente ist die Klasse ModelElement (vgl. Abbildung 5-1). NameSpace (Namensraum), eine Unterklasse von ModelElement, enthält eine beliebige Anzahl von Modellelementen und dient zur hierarchischen Strukturierung. Die Sichtbarkeit eines Modellelements innerhalb eines Namensraums wird durch die Assoziationsklasse ElementOwnership modelliert. Package (Paket) und Model (Modell) sind Unterklassen von NameSpace. Ein Modell ist ein spezielles Paket, das alles enthält, was zu einem UML-Modell gehört. In einem UML-Modell kann es daher immer nur eine Instanz von Model geben.

Die Klasse Classifier (Klassifizierer) ist eine Unterklasse der abstrakten Klasse GeneralizableElement (generalisierbares Element), wodurch ausgedrückt wird, dass ein Klassifizierer Eigenschaften von anderen Klassifizierern erben kann. Class, Interface und DataType sind Unterklassen von Classifier. Klassifizierer können Features (Eigenschaften) enthalten; das ist hier durch eine Komposition modelliert. Attribute und Operation sind Unterklassen von Feature.

Beziehungen zwischen Modellelementen wie z. B. Klassen und Interfaces werden ebenfalls als Klassen modelliert (siehe Abbildung 5-2). Die Klassen Association (Assoziation), Generalization (Generalisierung), Usage (Benutzung) und Abstraction (Abstraktion) sind Unterklassen der Klasse Relationship, die wiederum eine Unterklasse von ModelElement ist. Assoziationen werden mit Hilfe der zusätzlichen Klasse AssociationEnd modelliert, weil eine Assoziation mehr als zwei Modellelemente verbinden kann. Die übrigen Beziehungen verbinden genau zwei Modellelemente.

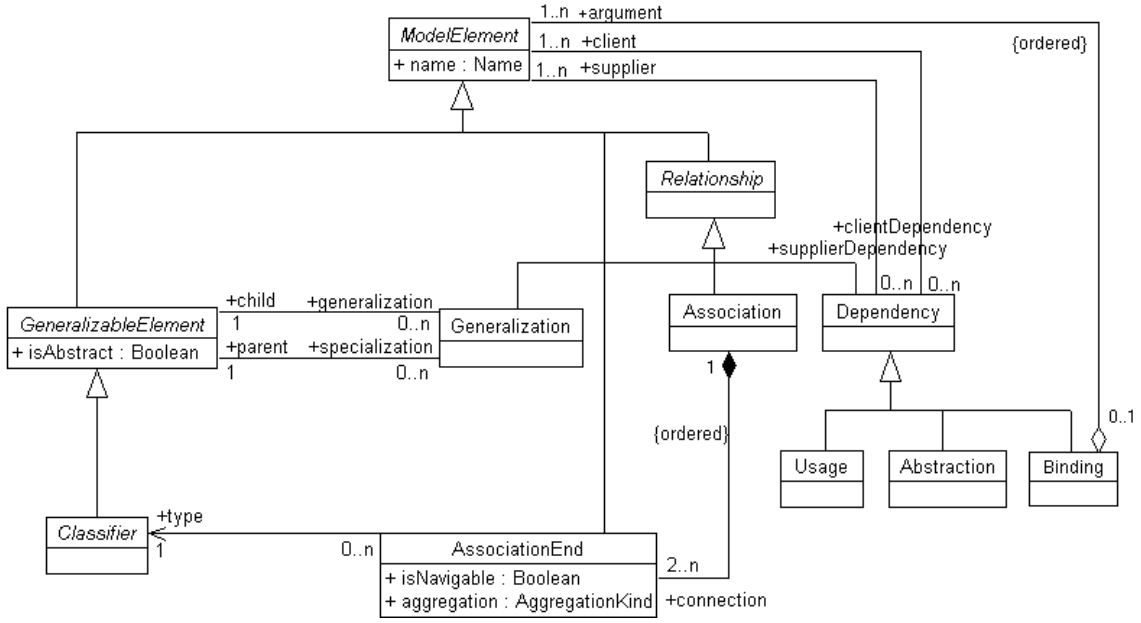

Abbildung 5-2: UML-Metamodell: Beziehungen (Ausschnitt)

5.2 Umfang

Abbildung 5-3 zeigt die konzeptionelle Struktur von ODEM. ODEM beruht auf dem UML-Metamodell. Für die Verwendung in ODEM wird das UML-Metamodell auf den tatsächlichen Bedarf reduziert. Zusätzlich werden nützliche, aus den Bestandteilen des UML-Metamodells abgeleitete Modellelemente eingeführt. Diese dienen vor allem dazu, als Abstraktionsschicht die Komplexität des UML-Metamodells nach außen zu verbergen. Außerdem sind sie praktisch bei der Definition von Metriken.

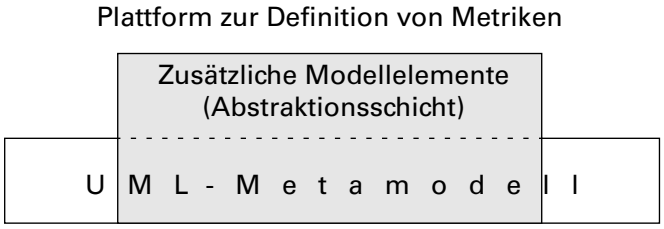

Abbildung 5-3: Konzeptionelle Struktur von ODEM

5.2.1 Einschränkungen

ODEM schränkt das UML-Metamodell in bestimmten Bereichen ein, indem Elemente und Attribute von Elementen weggelassen werden. Dahinter stecken Überlegungen über die typische Verfügbarkeit bestimmter Entwurfsinformationen. Bei einer Darstellung des Entwurfs in UML sind häufig nur Klassen- und Paketdiagramme vorhanden und relativ vollständig. Diese beschreiben allerdings nur die statische Struktur des Entwurfs.

Für die dynamische Struktur des Entwurfs werden vor allem Informationen über die Aufrufbeziehungen zwischen Methoden benötigt. Diese finden sich in den Sequenz- und Zustandsdiagrammen. Typischerweise werden diese Diagramme aber nur für

ausgewählte Szenarien erstellt, beschreiben also nur Ausschnitte aus der dynamischen Struktur. Für eine Bewertung wäre aber eine vollständige Beschreibung nötig. Daher werden alle dynamischen Informationen, die im UML-Metamodell vorgesehen sind, ausgeblendet.

Weitere Einschränkungen von ODEM gegenüber dem UML-Metamodell sind:

- Parametrisierte Klassen (Templates) werden nicht betrachtet, um das Modell überschaubarer zu gestalten (siehe dazu auch Abschnitt 5.4.4).
- Ebenfalls aus Gründen der Überschaubarkeit wird die Möglichkeit, in Klassen weitere Klassen oder Interfaces einzuschachteln, nicht berücksichtigt. Diese Form der Schachtelung sollte ohnehin nur selten verwendet werden, da sie sich negativ auf die Verständlichkeit auswirkt. Es gibt allerdings Fälle, wo sie sinnvoll ist: Beispielsweise kann eine Iterator-Klasse zur zugehörigen Container-Klasse lokal deklariert und damit der enge Zusammenhang deutlich gemacht werden.
- Methoden (Implementierungen von Operationen) und damit auch Redefinitionen von Methoden gehören zum Feinentwurf und werden daher ausgeblendet. Die durch Methodenaufrufe bedingte Benutzung von Klassen kann dennoch berücksichtigt werden: Wenn eine Aufrufbeziehung einer Methode einer Klasse zu einer Operation einer anderen Klasse bestehen wird, kann diese Beziehung als Benutzungsbeziehung dargestellt werden.
- Konstruktoren, d. h. Operationen mit Stereotyp «create» (OMG, 2000a) oder «constructor» (Rumbaugh et al., 1998), werden nicht berücksichtigt, da sie im Gegensatz zu den normalen Operationen nicht vererbt werden. Die Konstruktoren tragen in der Regel wenig zur Komplexität einer Klasse bei. Daher ist es einfacher, sie bei der Bewertung auszublenden, als ihren Sonderstatus im Modell zu berücksichtigen.
- Die Abstraktheit von Operationen wird ignoriert. Die Abstraktheit einer Operation ist lediglich eine Aussage darüber, ob eine Implementierung, d. h. eine Methode, existiert oder nicht. Methoden werden aber in ODEM gar nicht betrachtet. Würde die Abstraktheit berücksichtigt, müsste die Redefinition von abstrakten Operationen durch nicht-abstrakte Operationen speziell modelliert werden.
- Es wird nicht zwischen Konstanten und „normalen“ Attributen unterschieden, da der Unterschied relativ unbedeutend ist. Konstanten werden in UML durch Attribute mit dem Constraint {frozen} dargestellt; im UML-Metamodell hat die Klasse Attribute ein Attribut *changeability*, das bei Konstanten den Wert *frozen* hat.
- Die Multiplizität von Attributen und Assoziationen wird nicht berücksichtigt, weil sie relativ unwichtig ist.
- Datentypen (Instanzen von DataType im UML-Metamodell) stehen für die vordefinierten (primitiven) Datentypen der Implementierungssprache. Daher werden sie als Implementierungsdetail betrachtet und weggelassen. Sie tauchen nur als Typ eines Attributs oder Parameters auf.
- Assoziationsklassen, eine Mischung aus Assoziation und Klasse zur Modellierung von Attributen von Beziehungen, werden weggelassen. Ebenso entfallen Subsysteme, eine Mischung aus Paket und Klasse.

5.2.2 Erweiterungen

Neben diesen Einschränkungen gibt es auch noch Erweiterungen um zusätzliche Modellelemente. Diese bestehen aus Mengen von Entitäten und aus Relationen. Die neu eingeführten formalen Bezeichner sind in Tabelle 5-1 zusammengestellt.

Bezeichner	Bedeutung
S	Das System (enthält alle Modellelemente)
A	Menge aller Attribute in S
C	Menge aller Klassen in S
I	Menge aller Interfaces in S
M	Menge aller Parameter in S
O	Menge aller Operationen in S
P	Menge aller Pakete in S (einschließlich S)
associates	Relation für Assoziationsbeziehungen zwischen Klassen/Interfaces
associates*	Erweiterte associates-Relation (umfasst auch geerbte Assoziationen)
contains	Relation, die aussagt, dass ein Paket ein Modellelement enthält
contains*	Erweiterte contains-Relation (transitive Hülle)
depends_on	Aggregierte Relation für Beziehungen zwischen Klassen/Interfaces, die eine Abhängigkeit verursachen
depends_on*	Erweiterte depends_on-Relation (umfasst auch geerbte Abhängigkeiten)
extends	Relation für das Erben von Eigenschaften
extends*	Erweiterte extends-Relation (transitive Hülle)
has	Relation für das Enthaltensein von Eigenschaften in Klassen/Interfaces bzw. Klassen/Interfaces in Paketen
has*	Erweiterte has-Relation (umfasst auch geerbte Eigenschaften)
realizes	Relation für die Realisierung eines Interfaces durch eine Klasse
realizes*	Erweiterte realizes-Relation (umfasst auch geerbte Realisierungen)
uses	Relation für die Benutzung einer Klasse/eines Interfaces
uses*	Erweiterte uses-Relation (umfasst auch geerbte Benutzungen)

Tabelle 5-1: Überblick über die formalen Bezeichner in ODEM

Als Alternative zu den zusätzlichen Modellelementen wurde erwogen, die formale Definition von Metriken mit der Object Constraint Language (OCL; Warmer, Kleppe, 1999) durchzuführen. Die OCL ist eine formale Sprache zur Formulierung von Einschränkungen auf UML-Modellen. Erste Versuche mit OCL zeigten jedoch, dass die Erstellung von Metrikdefinitionen schwierig ist. Außerdem sind sie schlecht zu lesen und zu verstehen. Darum wurde dieser Ansatz nicht weiterverfolgt. Hingegen hat Abreu (2001) später OCL erfolgreich eingesetzt, um auf der Basis des Metamodells GOODLY Metriken zu definieren. Es ist wahrscheinlich, dass sich seine Ergebnisse auf das UML-Metamodell übertragen lassen. Allerdings sind die notwendigen komplexen OCL-Ausdrücke nicht besonders gut lesbar.

5.3 Kern

Abbildung 5-4 zeigt die Kernelemente von ODEM mit ihren Attributen und Beziehungen als UML-Klassendiagramm. (Genauso gut wäre eine Darstellung als Entity-Relationship-Diagramm möglich.) Die Elemente werden nun im Einzelnen vorgestellt. Dabei wird die Ableitung der Mengen und Relationen der Abstraktionsschicht von den Bestandteilen des UML-Metamodells erläutert.

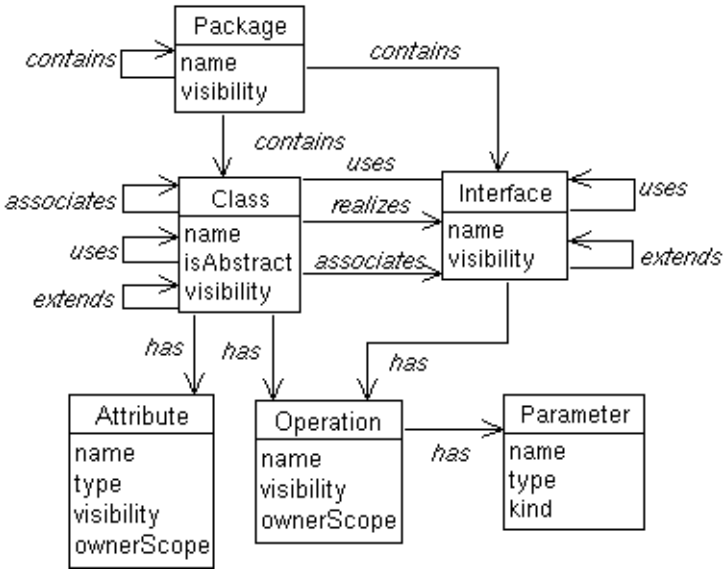

Abbildung 5-4: Der Kern von ODEM

5.3.1 System

Das entworfene System S besteht aus Paketen. S selbst wird ebenfalls als spezielles Paket aufgefasst. Daher gelten die in Abschnitt 5.3.2 für Pakete definierten Eigenschaften auch für S. Im UML-Modell ist S die einzige Instanz der Klasse Model.

5.3.2 Paket

Pakete gruppieren Klassen, Interfaces und (eingeschachtelte) Pakete. P ist die Menge aller Pakete, die im System vorkommen, einschließlich S.

Attribute

- *name*: Name. Der Name des Pakets.
- *visibility*: VisibilityKind. Sichtbarkeit des Pakets in Bezug auf den umschließenden NameSpace (also Paket oder System); kann die Werte public, protected und private annehmen. Der Wert dieses Attributs wird aus dem Wert des Attributs *visibility* der Instanz der Assoziationsklasse ElementOwnership abgeleitet, die zur Assoziation des Pakets mit seinem umschließenden NameSpace gehört.

Beziehungen

- *contains*: $P \times (P \cup C \cup I)$
  Ein Paket enthält ein Element (Paket, Klasse oder Interface). Jedes Element in P, C

und I muss in genau einem Paket enthalten sein (mit Ausnahme von S, das in keinem Paket enthalten ist).

Die Komposition im UML-Metamodell von ModelElement in NameSpace lässt sich als Relation formulieren. *contains*(p,q) gilt genau dann, wenn eine Komposition zwischen p in der Rolle namespace und q in der Rolle ownedElement existiert.

5.3.3 Interface

I ist die Menge aller Interfaces, die im System vorkommen.

Attribute

- *name*: Name. Der Name des Interfaces.
- *isAbstract*: Boolean. Interfaces sind immer abstrakt (Wert true).
- *visibility*: VisibilityKind. Bedeutung und Herleitung analog zu der bei den Paketen.

Beziehungen

- *extends*: $I \times I$
  Ein Interface erweitert ein anderes Interface, d. h. es erbt von ihm. Ein Interface kann beliebig viele Interfaces erweitern, auch keine.

  Im UML-Metamodell wird diese Beziehung durch die Klasse Generalization repräsentiert. *extends*(i,j) gilt genau dann, wenn es eine Instanz g von Generalization gibt, bei der die Rolle g.parent mit j und die Rolle g.child mit i belegt ist.
- *has*: $I \times O$
  Ein Interface besitzt eine Operation. Ein Interface kann beliebig viele Operationen haben, auch keine. Geerbte Operationen zählen nicht mit.

  Im UML-Metamodell wird das Besitzen einer Operation durch die Komposition zwischen Classifier und Feature (einer Oberklasse von Operation) modelliert. *has*(i,o) gilt genau dann, wenn es eine Instanz der Komposition gibt, bei der die Rolle owner mit i und die Rolle feature mit o besetzt ist.
- *uses*: $I \times (C \cup I)$
  Ein Interface benutzt eine Klasse oder ein Interface. Ein Interface kann beliebig viele andere Klassen/Interfaces benutzen, auch keine. Für die Benutzung gibt es unterschiedliche Gründe, z. B. die Verwendung als Parametertyp in einer Operation.

  Im UML-Metamodell steht die Klasse Usage für Benutzungsbeziehungen. *uses*(i,j) gilt genau dann, wenn es eine Instanz u von Usage gibt, bei der die Rolle u.client mit i und die Rolle u.supplier mit j belegt ist.

5.3.4 Klasse

C ist die Menge aller Klassen, die im System vorkommen.

Attribute

- *name*: Name. Der Name der Klasse.

- *isAbstract*: Boolean. Klassen können abstrakt (Wert true) oder konkret (Wert false) sein.
- *visibility*: VisibilityKind. Bedeutung und Herleitung analog zu der bei den Paketen.

Beziehungen

- *extends*: $C \times C$
  Eine Klasse erweitert eine Oberklasse, d. h. sie erbt von ihr. Eine Klasse kann beliebig viele Oberklassen erweitern, auch keine. Die Ableitung vom UML-Metamodell ist analog zur *extends*-Relation bei Interfaces.
- *realizes*: $C \times I$
  Eine Klasse realisiert ein Interface. Eine Klasse kann beliebig viele Interfaces realisieren, auch keine.

  Im UML-Metamodell steht die Klasse Abstraction, eine Unterklasse von Dependency, für diese Beziehung. *realizes*(k,i) gilt genau dann, wenn es eine Instanz a von Abstraction mit Stereotyp «realize» gibt, bei der die Rolle a.supplier mit i und die Rolle a.client mit k belegt ist.
- *has*: $C \times O$
  Eine Klasse besitzt eine Operation. Eine Klasse kann beliebig viele Operationen haben, auch keine. Geerbte Operationen zählen nicht mit. Die Ableitung vom UML-Metamodell ist analog zur *has*-Relation bei den Interfaces.
- *has*: $C \times A$
  Eine Klasse besitzt ein Attribut. Eine Klasse kann beliebig viele Attribute haben, auch keine. Geerbte Attribute zählen nicht mit. Die Ableitung vom UML-Metamodell ist analog zur obigen *has*-Relation, da Feature auch Oberklasse von Attribute ist.
- *associates*: $C \times (C \cup I)$
  Eine Klasse kann mit einer Klasse (oder einem Interface) assoziiert sein (gerichtet durch Navigierbarkeit). Eine Klasse kann beliebig viele andere Klassen assoziieren, auch keine. Eine Klasse kann sich auch selbst assoziieren. Geerbte Assoziationen zählen nicht mit.

  Im UML-Metamodell steht die Klasse Association für eine Assoziation. Association hat mindestens zwei AssociationEnds, in denen die Eigenschaften der Assoziation abgelegt sind (Navigierbarkeit: isNavigable, Art der Assoziation: aggregation). *associates*(k,l) gilt genau dann, wenn es eine Instanz a von Association gibt, die zwei Instanzen e1 und e2 von AssociationEnd hat, und wenn die Rolle e1.type mit k, die Rolle e2.type mit l belegt ist. Außerdem muss e2.isNavigable gelten, d. h. es muss möglich sein, von k nach l über die Assoziation zu navigieren.

  Das Attribut e1.aggregation gibt die Art der Assoziation an (none, aggregate, composite) und wird auf die Relation selbst übertragen. Auf diese Weise kann durch das Attribut *aggregation* der *associates*-Relation bei Bedarf zwischen normaler Assoziation, Aggregation und Komposition unterschieden werden.[2]

  Sofern eine Association mehr als zwei AssociationEnds hat, wird sie in binäre Assoziationen zerlegt. Eine n-äre Assoziation kann so (je nach Navigierbarkeit der AssociationEnds) in bis zu $n(n\text{-}1)/2$ binäre Assoziationen zerlegt werden.

- *uses*: $C \times (C \cup I)$
  Eine Klasse benutzt eine Klasse (oder ein Interface). Eine Klasse kann beliebig viele andere Klassen benutzen, auch keine. Für die Benutzung gibt es unterschiedliche Zwecke, z. B. die Instantiierung, den Aufruf einer Operation, die Verwendung als Parametertyp in einer Operation oder als Typ eines Attributs. Eine Klasse benutzt sich in der Regel selbst, diese implizite Benutzung wird aber traditionell nicht berücksichtigt. Die Ableitung vom UML-Metamodell ist analog zur *uses*-Relation bei den Interfaces.

5.3.5 Attribut

A ist die Menge aller Attribute, die im System vorkommen.

Attribute

- *name*: Name. Der Name des Attributs.
- *type*: Classifier. Typ des Attributs, der sich aus der gerichteten Assoziation von StructuralFeature mit Classifier im UML-Metamodell ergibt.
- *visibility*: VisibilityKind. Sichtbarkeit des Attributs, kann die Werte public, protected und private annehmen.
- *ownerScope*: ScopeKind. Gibt an, ob es sich um eine Klasseneigenschaft (Wert classifier) oder eine Objekteigenschaft (Wert instance) handelt.

5.3.6 Operation

O ist die Menge aller Operationen, die im System vorkommen.

Attribute

- *name*: Name. Der Name der Operation.
- *visibility*: VisibilityKind. Sichtbarkeit des Attributs, kann die Werte public, protected und private annehmen.
- *ownerScope*: ScopeKind. Gibt an, ob es sich um eine Klasseneigenschaft (Wert classifier) oder eine Objekteigenschaft (Wert instance) handelt.

Beziehungen

- *has*: $O \times M$
  Eine Operation besitzt einen Parameter. Eine Operation kann beliebig viele Parameter haben, auch keine. Sofern eine Operation eine Rückgabe hat, wird diese durch einen Pseudoparameter in der Parameterliste repräsentiert, der speziell gekennzeichnet ist (siehe Abschnitt 5.3.7).

2. Falls es zwischen zwei Modellelementen Assoziationen verschiedener Art gibt, wird das Attribut *aggregation* mit der stärksten Assoziationsart belegt, d. h. composite vor aggregate vor none. Die Alternative wäre die Einführung drei verschiedener Relationen und einer zusammenfassenden Relation. Dies erscheint hier allerdings überdimensioniert, weshalb die Ungenauigkeit im oben genannten Spezialfall in Kauf genommen wird.

Die Komposition von Parameter in BehavioralFeature (der Oberklassen von Operation) kann als Relation ausgedrückt werden: *has*(o,p) gilt genau dann, wenn es eine Instanz der Komposition gibt, bei der die Rolle type mit o und die Rolle parameter mit p belegt ist.

5.3.7 Parameter

M ist die Menge aller Parameter von Operationen.

Attribute

- *name*: Name. Der Name des Parameters.
- *kind*: ParameterDirectionKind. Die Art des Parameters (in, out, inout, return). in, out und inout stehen für die Richtung der Parameterübergabe beim Aufruf (analog zur Programmiersprache Ada, vgl. ISO/IEC 8652:1995, S. 123ff.). return kennzeichnet den Pseudoparameter für die Rückgabe, mit dessen Hilfe der Rückgabetyp modelliert wird.
- *type*: Classifier. Typ des Parameters, der sich aus der gerichteten Assoziation von Parameter mit Classifier im UML-Metamodell ableitet.

5.4 Erweiterungen

In diesem Abschnitt werden Erweiterungen von ODEM vorgestellt. Zunächst werden Relationen eingeführt, die aus den bereits eingeführten Relationen abgeleitet werden können. Diese Relationen vereinfachen die Definition von Metriken auf der Basis von ODEM. Dann wird diskutiert, wie Relationen gewichtet werden können. Abschließend werden verschiedene Aspekte einer Erweiterung von ODEM um parametrisierte Klassen (Templates) betrachtet.

5.4.1 Abgeleitete Relationen

Die allgemeine Abhängigkeitsbeziehung *depends_on* fasst alle Arten von Abhängigkeiten zwischen Modellelementen zusammen. Sie setzt voraus, dass tatsächlich alle Arten von Benutzung zwischen Klassen (z. B. Aufruf einer Methode, Zugriff auf ein Attribut, Benutzung als Attributtyp oder Parametertyp) im UML-Modell als Instanzen von Usage repräsentiert sind. Nur im Falle der Benutzung als Typ bei Attributen oder Parametern können *uses*-Beziehungen aus der vorliegenden Information automatisch abgeleitet werden. Für die anderen genannten Fälle (z. B. Zugriff auf ein Attribut) ist das nicht möglich, daher ist es wichtig, diese Arten von Benutzungsbeziehungen explizit anzugeben.

$depends\_on(x,y) \Leftrightarrow extends(x,y) \vee realizes(x,y) \vee associates(x,y) \vee uses(x,y)$.

Von dieser Relation kann eine Relation *depends_on* für Paketabhängigkeiten abgeleitet werden. Ein Paket p hängt von einem Paket q genau dann ab, wenn es eine Klasse oder ein Interface c aus p gibt, das von einer Klasse oder einem Interface d aus q abhängt.

$depends\_on(p,q) \Leftrightarrow \exists c,d \in C \cup I: c \neq d \wedge contains(p,c) \wedge contains(q,d) \wedge depends\_on(c,d)$

5.4.2 Erweiterte Relationen

Bisher wurden die Relationen *associates*, *has* und *uses* definiert, ohne Vererbung zu berücksichtigen. Faktisch ist es aber so, dass nicht nur Eigenschaften, sondern auch Beziehungen vererbt werden. Daher ist es sinnvoll, weitere Relationen einzuführen, die auch vererbte Beziehungen umfassen. Zunächst wird eine Relation *extends** benötigt, welche die transitive Hülle von *extends* darstellt.

$extends^*(x,y) \Leftrightarrow extends(x,y) \vee \exists z \in C \cup I: (extends^*(x,z) \wedge extends(z,y))$

Nun können die erwähnten erweiterten Relationen definiert werden, die auch geerbte Beziehungen berücksichtigen. Die Definition wird hier am Beispiel von *uses** gezeigt (*associates**, *depends_on** und *has** sind analog definiert[3]). Man beachte, dass es sich hier *nicht* um die transitive Hülle handelt, weil *extends** verwendet wird.

$uses^*(x,y) \Leftrightarrow uses(x,y) \vee \exists z \in C \cup I: (extends^*(x,z) \wedge uses(z,y))$

Zum Schluss wird noch die erweiterte Version *contains** eingeführt, welche die transitive Hülle von *contains* ist.

$contains^*(x,y) \Leftrightarrow contains(x,y) \vee \exists z \in C \cup I: (contains^*(x,z) \wedge contains(z,y))$

Wegen der hierarchischen Strukturierung gilt $contains^*(S,x)$ für alle Elemente x von $P \cup C \cup I$ mit Ausnahme von S.

5.4.3 Gewichte für die Relationen

Die bisher definierten Relationen berücksichtigen nicht, wie viele Beziehungen einer Art zwischen zwei Modellelementen bestehen. Es könnte allerdings einen Unterschied machen, ob eine Klasse eine oder mehrere Beziehungen einer Art mit einer anderen Klasse hat. Deshalb wird für jede Relation ein Attribut namens *weight* eingeführt. Der Wert von *weight* ist für die Relationen *contains*, *extends* und *realizes* immer gleich 1. Bei der *associates*- und der *uses*-Relation ist es möglich, dass eine Beziehung mehrfach besteht. Dann ist der Wert von *weight* die Anzahl dieser Beziehungen. Der Wert von *weight* bei der *depends_on*-Relation ergibt sich als Summe der Werte von *weight* bei den berücksichtigten Relationen (für das Gewicht nicht vorhandener Relationen wird der Wert 0 angenommen).

$depends\_on(x,y).weight = extends(x,y).weight + realizes(x,y).weight + associates(x,y).weight + uses(x,y).weight$

Für die *-Relationen bleibt bei *contains** der Wert von *weight* bei 1. Bei *extends** entspricht der Wert von *weight* der Anzahl der Pfade, über die eine Klasse von einer anderen erbt.

$extends^*(x,y).weight = extends(x,y).weight + \Sigma_{z \in C \cup I:\ extends(x,z)}\ extends^*(z,y).weight$

Bei den Relationen *associates**, *realizes**, *uses** und *depends_on** muss zur Berechnung des Gewichts die Vererbungshierarchie des ersten Modellelements nach oben hin durchsucht werden: Für jede Beziehung der gesuchten Art zwischen der Klasse selbst oder einer ihrer Oberklassen mit der anderen Klasse ist der Wert von *weight* dieser

3. Bei *associates** ist noch das Attribut *aggregation* zu beachten. Dieses wird mit der stärksten Assoziationsart der beteiligten Assoziationen belegt (wie bei *associates*, vgl. Abschnitt 5.3.4).

Relation zur Gesamtsumme hinzuzuzählen. Hier wird die Formalisierung am Beispiel von *uses** gezeigt.

$uses^*(x,y).weight = uses(x,y).weight + \Sigma_{z \in C \cup I:\ extends^*(x,z)}\ uses(z,y).weight$

Bei *realizes** sollte das Gewicht immer maximal 1 sein, da faktisch eine Klasse ein Interface nicht mehrfach realisieren kann. Es kann höchstens eine bereits vorhandene Realisierung durch Redefinition der geerbten Realisierung geändert werden. Ein Gewicht größer 1 dürfte immer auf einen Entwurfsfehler hinweisen.

5.4.4 Überlegungen zu Templates

Die Möglichkeit zur Bildung von parametrisierten Modellelementen ist in UML nicht auf Klassen beschränkt. Beispielsweise lassen sich Entwurfsmuster als parametrisierte Kollaborationen modellieren. Die hier betrachtete Erweiterung von ODEM beschränkt sich aber auf Template-Klassen, da diese den üblichen Gebrauch von Templates darstellen (z. B. in C++ oder Eiffel). Es werden auch nur vollständige Instantiierungen von Templates betrachtet; außerdem wird die Möglichkeit zur Default-Belegung von Parametern außer Acht gelassen.

Im UML-Metamodell werden Templates durch eine Assoziation der Klasse ModelElement mit sich selbst modelliert (vgl. Abbildung 5-1). Ein ModelElement kann eine Menge von Template-Parametern haben (Rolle templateParameter). Hat ein ModelElement mindestens einen solchen Parameter, ist es ein Template, sonst nicht. Die Template-Parameter sind formale Parameter und dürfen keine innere Struktur (Attribute, Operationen, etc.) besitzen. Nur ihr Name und ihr Typ sind relevant. Die formalen Parameter können Klassen sein, aber auch Datentypen und Operationen sind möglich. Es können zusätzliche Einschränkungen für den Typ der Parameterbelegung gemacht werden (z. B. der Parameter muss ein Interface implementieren oder eine ganze Zahl sein). Außerdem können Beziehungen zwischen den Parametern untereinander und mit dem Template selbst formuliert werden (z. B. Aggregation, Vererbung etc.). Diese Beziehungen müssen dann auch bei den korrespondierenden Elementen der Parameterbelegungen vorhanden sein. Das Template selbst kann zu anderen Modellelementen nur ausgehende Beziehungen haben; diese werden auf die Instanzen übertragen.

Eine Instantiierung eines Templates wird durch die Dependency-Unterklasse Binding modelliert. Diese verbindet über die von Dependency geerbten Assoziationen das Template (Rolle supplier) mit seiner Instanz (Rolle client). Die Parameterbelegung wird durch eine zusätzliche, geordnete Aggregation modelliert; die Rolle arguments enthält dabei die aktuellen Parameter.

Template-Klassen selbst sind keine echten Klassen, aber ihre Instanzen sind es. Daher gehören Template-Instanzen zu C, Template-Klassen aber nicht. Für die Templates wird daher eine neue Menge T eingeführt. Formale Parameter von Templates müssen aussortiert werden, dürfen also keinesfalls zu C hinzugenommen werden.

Auf der Basis von Binding kann nun eine weitere Relation *binds* eingeführt werden:

binds: $C \times T$
Eine Klasse ist Instanz eines Templates. *binds*(c,t) gilt genau dann, wenn es eine Instanz von Binding gibt, bei der die Rollen client mit c und supplier mit t belegt sind.

Eine Hinzunahme der Template-Klassen zu ODEM ist also prinzipiell möglich. Es gibt aber wichtige Gründe, es nicht zu tun. Zum einen gibt es noch einige Unklarheiten über die tatsächliche Darstellung im UML-Metamodell. Beispielweise fehlt in der Spezifikation die Information darüber, wie die Beziehungen von formalen Template-Parametern in den Template-Instanzen modelliert werden. Zum anderen wird durch die Hinzunahme von Templates die Formulierung und Auswahl von Metriken erschwert. Wenn z. B. die Anzahl der Klassen gezählt werden soll, kann man die echten Klassen und die Template-Klassen zählen, weil das diejenigen sind, die tatsächlich implementiert werden müssen. Alternativ zählt man die echten Klassen und die Template-Instanzen, weil das die Anzahl der tatsächlichen Klassen im Modell ist. Welche Zählweise ist nun die richtige? Bei beiden gibt es Argumente dafür und dagegen. Wegen der genannten Schwierigkeiten und der zusätzlichen Komplexität wurde entschieden, Templates vorläufig nicht in ODEM aufzunehmen.

5.5 Formale Definition von Metriken

5.5.1 Stand der Praxis

Metriken für objektorientierte Systeme gibt es inzwischen viele (z. B. Chidamber, Kemerer, 1991; Chen, Lu, 1993; Kolewe, 1993; Li, Henry, 1993; Abreu, Carapuca, 1994; Chidamber, Kemerer, 1994; Hopkins, 1994; Lorenz, Kidd, 1994; Martin, 1995; Tegarden et al., 1995; Henderson-Sellers, 1996; Li, 1998; Marchesi, 1998; Genero et al., 2000). Einen Überblick über die Literatur geben Archer, Stinson (1995) und Fetcke (1995).

Ein schwer wiegendes Problem vieler dieser Metriken ist deren unpräzise natürlichsprachliche Spezifikation. Dies tritt besonders bei der Frage zutage, wie mit geerbten Eigenschaften umgegangen wird (Churcher, Shepperd, 1995a). Beispielsweise werden Begriffe wie „local method“ oder „method of a class“ (Li, Henry, 1993 bei der Definition von Number of Methods, NOM) verwendet, um eine Zählmetrik von Methoden einer Klasse zu definieren. Leider werden diese Begriffe aber nicht näher erläutert. Es bleibt damit unklar, was genau gezählt wird:

- Werden geerbte Methoden mitgezählt?
- Werden bei redefinierten Methoden sowohl die geerbte als auch die neu definierte Methode gezählt?
- Werden Klassenmethoden, Instanzmethoden oder beide gezählt?
- Werden öffentliche Methoden, private Methoden oder beide gezählt?

Das gleiche Problem hat übrigens auch die ähnliche Metrik WMC (Weighted Methods per Class; Chidamber, Kemerer, 1994), wie auch Churcher und Shepperd (1995b) feststellen. Eine formale Definition der Metriken ist also unabdingbar. Es bietet sich an, als Basis der formalen Definition ODEM als Referenzmodell zu verwenden.

5.5.2 Fallstudien

Um die Eignung von ODEM zur formalen Definition von Metriken auf bereits in der Literatur definierte Metriken zu erproben, werden in zwei Fallstudien die Paketmetriken von Martin (1995) und die bekannte Metrikensuite von Chidamber und Kemerer

(1994) mit Hilfe von ODEM formalisiert (soweit möglich). Die formalen Definitionen verwenden zum Teil Metriken aus QOOD (vgl. Tabelle 9-1).

Fallstudie 1: Martins Paketmetriken

Martin definiert Kriterien für die richtige Verteilung von Klassen auf Pakete. Diese Kriterien basieren auf dem Begriff der Abhängigkeit. Ziel ist die Minimierung von Abhängigkeiten, insbesondere Abhängigkeiten zu konkreten Klassen. Leider definiert Martin nicht genau, was eigentlich eine Abhängigkeit ist. Es sagt nur, dass Abhängigkeiten durch Klassenbeziehungen wie Vererbung, Aggregation und Benutzung entstehen. Hier wird für die Formalisierung die *depends_on*-Relation verwendet, welche die genannten Beispiele einschließt.

Relational cohesion (H). Diese Metrik soll den Zusammenhalt von Klassen innerhalb eines Pakets erfassen. Weil die Klassen innerhalb eines Pakets eng verwandt sein sollen, soll der Zusammenhalt hoch sein. H ist definiert als (R+1)/N, wobei R die Anzahl der Beziehungen zwischen Klassen innerhalb des Pakets ist und N die Anzahl der Klassen. Die folgende Definition zählt gegenseitige Abhängigkeiten zweimal, je einmal für jede Richtung. Die Zahl der Interfaces (Metrik NIP) wird zur Zahl der Klassen (Metrik NCP) hinzugezählt.

$$H(p) = ((\Sigma_{c \in C \cup I:\ contains(p,c)}\ \Sigma_{d \in C \cup I \setminus \{c\}:\ contains(p,d)} depends\_on(c,d).weight) + 1) \ /\ (NCP(p) + NIP(p))$$

Afferent Coupling (C_a). Die Anzahl der Klassen aus anderen Paketen, die von den Klassen im Paket abhängen. Die afferente Kopplung soll niedrig sein.

$$C_a(p) = |\{d \in C \cup I: \neg contains(p,d) \wedge (\exists c \in C \cup I: contains(p,c) \wedge depends\_on(d,c))\}|$$

Efferent Coupling (C_e). Die Anzahl der Klassen aus anderen Paketen, von denen Klassen im Paket abhängen. Die efferente Kopplung soll niedrig sein.

$$C_e(p) = |\{d \in C \cup I: \neg contains(p,d) \wedge (\exists c \in C \cup I: contains(p,c) \wedge depends\_on(c,d))\}|$$

Abstractness (A). Die Abstraktheit eines Pakets ist definiert als das Verhältnis von abstrakten Klassen (Metrik NCP_a) zu der Gesamtzahl der Klassen. Martin erwähnt keine Interfaces; sie werden hier zu den abstrakten Klassen dazugezählt. Der Spezialfall eines Pakets ohne Klassen (von Martin nicht erwähnt) soll eine Abstraktheit von 1 haben. Solche leeren Pakete sind sinnvoll für die Strukturierung anderer Pakete.

$$A(p) = \text{if } NCP(p) + NIP(p) > 0 \text{ then } (NCP_a(p) + NIP(p)) \ /\ (NCP(p) + NIP(p)) \text{ else } 1$$

Instability (I). Die Instabilität eines Pakets ist definiert als das Verhältnis efferenter Kopplung zur gesamten Kopplung. Der Spezialfall eines Pakets ohne Kopplung nach außen (von Martin nicht erwähnt) soll eine Instabilität von 0 haben.

$$I(p) = \text{if } C_a(p) + C_e(p) > 0 \text{ then } C_e(p) \ /\ (C_a(p) + C_e(p)) \text{ else } 0$$

Distance from the Main Sequence (D). Die Hauptlinie (main sequence) ist Teil einer Theorie von Martin, dass die Abstraktheit A und die Instabilität I eines Pakets ungefähr gleich sein sollten, d. h. reine Abstraktionen sollten sehr stabil sein, während sich konkrete Implementierungen ändern dürfen. Je weiter ein Paket von der Hauptlinie, ausgedrückt durch die Gleichung A + I = 1, entfernt ist, desto schlechter. Es gibt auch eine normalisierte Variante D', die im Intervall [0,1] liegt.

$D(p) = |A(p) + I(p) - 1| \; / \sqrt{2}$

$D'(p) = |A(p) + I(p) - 1| = \sqrt{2}\; D(p)$

Fazit. Martins Metriken konzentrieren sich auf Elemente des Architekturentwurfs, weshalb sie mit ODEM leicht formalisiert werden können. Es gibt ein paar Lücken und Unklarheiten in den ursprünglichen natürlichsprachlichen Definitionen, die bei der Formalisierung durch sinngemäße Ergänzungen überwunden werden können.

Fallstudie 2: Chidamber und Kemerers Metrikensuite

Chidamber und Kemerer waren 1991 eine der ersten, die eine Suite von objektorientierten Metriken publiziert haben. Wohl auch deshalb hat die Suite eine hohe Popularität. Hier wird die zweite Version von 1994 zur Formalisierung verwendet.

Weighted Methods per Class (WMC). WMC ist die Summe der Komplexitäten der Methoden einer Klasse. Je geringer WMC, desto besser. Die Definition der Komplexität ist absichtlich offen gelassen. WMC wird meistens mit einer Standardkomplexität von 1 verwendet (auch von den Autoren selbst; vgl. Chidamber, Kemerer, 1998). Leider haben die Autoren erst später genauere Aussagen darüber gemacht, welche Methoden gezählt werden sollen (Chidamber, Kemerer, 1995). Die Hauptinterpretation vom WMC in der Literatur ist es, nur lokal definierte Methoden zu zählen (einschließlich Redefinitionen). Da ODEM nur Operationen und keine Methoden betrachtet, kommen Redefinitionen allerdings nicht vor. Es werden sowohl Klassen- als auch Instanzmethoden gezählt.

$WMC(c) = |\{o \in O: \mathit{has}(c,o)\}|$

Depth of Inheritance Tree (DIT). DIT ist die maximale Länge aller Vererbungspfade von der Klasse zu den Wurzelklassen der Vererbungshierarchie. Je geringer DIT, desto besser. Hier wird eine rekursive Definition angegeben:

$DIT(c) = \text{if } \exists d \in C \cup I: \mathit{extends}(c,d) \text{ then } 1+\max_{d \in C:\, \mathit{extends}(c,d)} \{DIT(d)\} \text{ else } 0$

Number of Children (NOC). NOC ist die Anzahl der direkten Unterklassen einer Klasse. Ein hoher Wert deutet sowohl auf bessere Wiederverwendung als auch auf den Missbrauch von Vererbung und höheren Testaufwand hin, so dass es keine klare Aussage gibt, welche Werte besser sind.

$NOC(c) = |\{d \in C \cup I: \mathit{extends}(d,c)\}|$

Coupling between Object Classes (CBO). CBO ist die Anzahl der Klassen, an die eine Klasse gekoppelt ist. Eine Klasse A ist an eine Klasse B gekoppelt, wenn eine Methode von A eine Methode oder ein Attribut von B verwendet. Je geringer CBO, desto besser. Diese Metrik benötigt genaue Information über die Methoden, es wird also ein sehr detaillierter Entwurf oder der Code der Klasse für die Messung vorausgesetzt. Derart detaillierte Information ist in ODEM aber nicht vorhanden, weshalb diese Metrik nicht formalisiert werden kann.

Response for a Class (RFC). Die Größe der Response-Menge einer Klasse, d. h. die Anzahl der Methoden einer Klasse plus die Anzahl der Methoden anderer Klassen, die von den Methoden der Klasse benutzt werden (jede Methode zählt nur einmal). Je geringer RFC, desto besser. Auch diese Metrik benötigt genaue Informationen über die Methoden, weshalb das gleiche Problem bei der Formalisierung auftritt wie bei

CBO. Außerdem wird, wie bei WMC, keine Aussage darüber gemacht, welche Methoden eigentlich mitzählen.

Lack of Cohesion in Methods (LCOM). Diese Metrik soll den Zusammenhalt der Methoden einer Klasse erfassen, indem deren Ähnlichkeit betrachtet wird. Die Ähnlichkeit zweier Methoden ist hier die Anzahl der Attribute der Klasse, auf die beide zugreifen. LCOM ist definiert als die Anzahl von Methodenpaaren mit Ähnlichkeit 0 (keine gemeinsamen Attribute) minus die Anzahl der Paare mit einer Ähnlichkeit größer 0. Falls das Ergebnis kleiner als 0 ist, wird LCOM zu 0 gesetzt. Je geringer LCOM, desto besser. Wie bei CBO scheitert die Formalisierung daran, dass Information über die Zugriffe von Methoden auf Attribute benötigt wird.

Fazit. DIT und NOC können leicht formalisiert werden, weil sie sich auf die Vererbungshierarchie beziehen, die Teil des Architekturentwurfs ist. Die Metriken CBO, RFC und LCOM benötigen mehr detaillierte Information zur Messung, als ODEM bieten kann. WMC kann in seiner Standardform formalisiert werden, allerdings ist die ursprüngliche Definition zu ungenau, um eine eindeutige Formalisierung zu erlauben.

Zusammenfassung

Die Paketmetriken von Martin (1995) lassen sich problemlos formalisieren; gleichzeitig wurden bei der Formalisierung Lücken in der Definition aufgedeckt. Von den Klassenmetriken von Chidamber und Kemerer (1994) lässt sich hingegen nur ein Teil formalisieren. Für die übrigen Metriken ist eine Formalisierung auf der Basis von ODEM nicht möglich, weil dafür detaillierte Informationen über die Aufrufbeziehungen von Methoden und den Zugriff von Methoden auf Attribute vorausgesetzt werden. Diese Informationen sind in UML-Modellen aber selten verfügbar und daher in ODEM nicht vorhanden. Es handelt sich bei CBO, RFC und LCOM genau genommen eher um Code-Metriken als um Entwurfsmetriken.

ODEM ist sehr gut zur Definition von Metriken für den objektorientierten Entwurf geeignet, sofern sich die benötigte Information aus UML-Modellen (insbesondere aus Klassendiagrammen) gewinnen lässt. Das belegt auch die erfolgreiche Nutzung von ODEM in dieser Arbeit bei der Definition der Metriken von QOOD (vgl. Anhang A).

Kapitel 6

Softwarequalität

Quality … you know what it is, yet you don't know what it is. But that's self-contradictory. But some things are better than others, that is, they have more quality. But when you try to say what the quality is, apart from the things that have it, it all goes poof! There's nothing to talk about. But if you can't say what Quality is, how do you know what it is, or how do you know that it even exists? If no one knows what it is, then for all practical purposes it doesn't exist at all. But for all practical purposes it really does exist. What else are the grades based on? Why else would people pay fortunes for some things and throw others in the trash pile? Obviously some things are better that others … but what's the "betterness"? … So round and round you go, spinning mental wheels and nowhere finding anyplace to get traction. What the hell is Quality? What is it?
(Pirsig, 1981, S. 163f.)

In diesem Kapitel wird der Qualitätsbegriff erst allgemein und dann spezifisch für Software diskutiert. Softwarequalität wird häufig durch Qualitätsmodelle definiert, daher werden Ansätze und Beispiele für solche Modelle vorgestellt. Den Abschluss bildet eine kurze Diskussion zur Qualitätssicherung.

6.1 Qualität

6.1.1 Definition

Das Wort „Qualität" kommt vom lateinischen Wort *qualitas* (Beschaffenheit) und beschreibt die Güte oder den Wert eines Objekts. DIN 55350 definiert den Begriff Qualität ähnlich, aber nicht gleich wie die Deutsche Gesellschaft für Qualität (DGQ).

Definition 6-1 (Qualität, DIN 55350-11:1987-05)

Die Gesamtheit von Eigenschaften und Merkmalen eines Produktes oder einer Tätigkeit, die sich auf die Eignung zur Erfüllung gegebener Erfordernisse bezieht.

Definition 6-2 (Qualität, DGQ, 1995)

Die Gesamtheit von Merkmalen (und Merkmalswerten) einer Einheit bezüglich ihrer Eignung, festgelegte und vorausgesetzte Erfordernisse zu erfüllen.

Beide Definitionen liefern eine klare und eindeutige, abstrakte Definition der Qualität. Inhaltlich sind sie ähnlich, aber unterschiedlich: Die DIN-Definition beschränkt sich auf die explizit festgelegten Eigenschaften, während die DGQ-Definition auch vorausgesetzte, d. h. implizite Erfordernisse zulässt.[1] Wie bei den Standards gibt es auch in der Praxis keine wirkliche Übereinkunft über die Bedeutung des Begriffs Qualität. Man kann Qualität aus verschiedenen Perspektiven sehen und beurteilen. Garvin (1984, 1988) unterscheidet fünf Sichten, unter denen man den Begriff Qualität in Bezug auf ein Produkt definieren kann:

- transzendent (transcendent)
- produktbezogen (product-based)
- benutzerbezogen (user-based)
- herstellungsbezogen (manufacturing-based)
- kostenbezogen (value-based)

Transzendente Sicht

Die transzendente Sicht besagt, dass Qualität etwas Absolutes und universell Erkennbares ist – eine Art innewohnende Vortrefflichkeit. Jeder kann lernen, sie zu erkennen, aber nur durch Erfahrung, nicht durch Analyse. Qualität entzieht sich jeder Analyse, sie kann nicht präzise definiert werden. Anhand von Beispielen, die Qualität besitzen, kann man aber lernen, Qualität zu erkennen. Hier gibt es deutliche Parallelen zum Begriff der Schönheit, der nach Platon ebenfalls nicht definiert, sondern nur erfahren werden kann. Pirsig (1981, S. 185) formuliert das wie folgt: „But even though Quality cannot be defined, you know what Quality is."

Das dürfte auch der Grund sein, warum der Architekt Christopher Alexander der Qualität, die er bei seiner Architektur anstrebt, den Namen *quality without a name* gegeben hat (Alexander, 1977, 1979). Diese Qualität ist etwas Reales und Objektives, das jedoch nicht in Worte gefasst werden kann: „There is a central quality which is the root criterion of life and spirit in a man, a town, a building, or a wilderness. This quality is objective and precise, but it cannot be named." (Alexander, 1979, S. ix)

Produktbezogene Sicht

Die produktbezogene Sicht hingegen sieht Qualität als präzise und messbar an. Unterschiede in der Qualität reflektieren Unterschiede in den Bestandteilen oder den Attributen eines Produkts. Die Qualität wird also auf messbare Eigenschaften eines Produkts zurückgeführt, was es erlaubt, eine Rangfolge von Produkten zu erstellen. Auf diese Weise ist Qualität eine inhärente Eigenschaft eines Produkts, die objektiv bestimmt werden kann. Diese Qualitätssicht wird typischerweise bei Produktvergleichen, z. B. denen der Stiftung Warentest, eingenommen.

1. In der Nachfolgenorm der DIN 55350, DIN EN ISO 8402, sind die impliziten Anforderungen aufgenommen worden: „Qualität ist die Gesamtheit von Merkmalen einer Einheit bezüglich ihrer Eignung, festgelegte oder vorausgesetze Erfordernisse zu erfüllen."

Benutzerbezogene Sicht

Die benutzerbezogene Sicht definiert Qualität aus der Sicht des Benutzers eines Produkts. Er wird dasjenige Produkt als hochwertig ansehen, das seine Bedürfnisse optimal befriedigt. Juran (1974) z. B. definiert „quality is fitness for use“. Die Bestimmung der Qualität durch den Benutzer ist aber subjektiv: „Quality is the degree to which a specific product satisfies the wants of a specific customer.“ (Gilmore, 1974). Weinberg (1991) macht die Subjektivität von Qualität besonders deutlich: Offensichtlich haben alle dieselbe Definition für Qualität, und die lautet „Quality is whatever I like“. Daraus eine einheitliche, objektive Definition abzuleiten, ist unmöglich. Allenfalls sind Mehrheitsentscheidungen denkbar, die einen bestimmten Geschmack festlegen, an dem sich Produkte orientieren müssen, um marktfähig zu sein.

Herstellungsbezogene Sicht

Die herstellungsbezogene Sicht definiert die Qualität eines Produkts ausgehend von seinem Entwurfs- und Herstellungsprozess. Im Blickpunkt stehen die (feststehenden) Anforderungen an das Produkt: „Quality is the degree to which a specific product conforms to a design or specification.“ (Gilmore, 1974). Ein Produkt hat dann eine hohe Qualität, wenn seine Eigenschaften in hoher Übereinstimmung mit den Anforderungen stehen (Crosby, 1979). Jede Abweichung bedeutet einen Qualitätsverlust; es entsteht Ausschuss oder der Bedarf für Nacharbeit. Hier können die üblichen Qualitätssicherungsmaßnahmen eingesetzt werden, die versuchen, solche Abweichungen zu erkennen, oder besser noch, sie gleich zu vermeiden (Null-Fehler-Ziel). Diese Qualitätssicht ist typisch für die Herstellung materieller Gegenstände, insbesondere in der Massenfertigung.

Kostenbezogene Sicht

Die kostenbezogene Sicht geht noch einen Schritt weiter als die anderen Sichten. Qualität wird hier auf der Grundlagen von Kosten und Preisen definiert. Ein Produkt ist dann von hoher Qualität, wenn es die gewünschte Leistung zu einem akzeptablen Preis oder die gewünschte Übereinstimmung mit den Anforderungen zu akzeptablen Kosten bietet (Broh, 1974). Die inhärente Qualität wird so mit den Kosten für Anschaffung (und Betrieb) in Beziehung gesetzt. Entscheidend für Qualität ist somit das Preis-Leistungsverhältnis. Unter Umständen kann auch die Zeit, zu der ein Produkt zur Verfügung steht, Einfluss auf dessen Qualität haben (Time-to-Market). Je früher das Produkt beim Kunden einsetzbar ist, desto höher wird die von ihm wahrgenommene Qualität des Produkts, falls die Bereitstellungszeit für ihn eine Rolle spielt.

Konsequenzen

Die Existenz der fünf unterschiedlichen Sichten führt häufig zu Verwirrungen, wenn über Qualität gesprochen wird. Beispielsweise kommt es regelmäßig zu Missverständnissen, wenn die Marketing-Abteilung, die eher der benutzerbezogenen Sicht zuneigt, sich mit der Produktionsabteilung, die eher die herstellungsbezogene Sicht einnimmt, über die gewünschte Qualität eines Produkts einigen soll. Trotzdem ist es wichtig, verschiedene Sichten bei der Entwicklung und Herstellung eines Produkts einzubeziehen. Am Anfang des Entstehungsprozesses eines neuen Produkts steht eine Marktanalyse (liefert benutzerbezogene Qualitäten). Daraus werden die Eigenschaften des Produkts abgeleitet (produktbezogene Qualität). Schließlich muss das

Produkt nach den Anforderungen hergestellt werden (herstellungsbezogenen Qualität). Der Kunde schließlich wird bei der Entscheidung für ein Produkt auch die kostenbezogene Sicht einnehmen. Nur im Zusammenspiel der Sichten wird am Schluss ein hochwertiges und erfolgreiches Produkt entstehen.

6.1.2 Softwarequalität

Definition

Auch in der Welt der Software herrscht keine Einigkeit über den Begriff der Qualität. Jones (1996) demonstriert dies, indem er diverse Größen des Software Engineering mit ihrer Definition von Softwarequalität zitiert (vgl. Tabelle 6-1). Jede dieser Definitionen hat ihre Berechtigung, zusammengenommen sind sie aber widersprüchlich. Jones fordert für eine praxisrelevante Definition von Softwarequalität, dass Qualität messbar (nach der Fertigstellung der Software) und vorhersagbar (vor der Fertigstellung der Software) sein sollte.

Autor	Definition von Softwarequalität
Barry Boehm	Achieving high levels of user satisfaction, portability, maintainability, robustness, and fitness for use
Phil Crosby	Conformance to user requirements
W. Edwards Deming	Striving for excellence in reliability and functions by continuous improvement in the process of development, supported by statistical analysis of the causes of failure
Watts Humphrey	Achieving excellent levels of fitness for use, conformance to requirements, reliability, and maintainability
Capers Jones	The absence of defects that would make software either stop completely or produce unacceptable results
James Martin	Being on time, within budget, and meeting user needs
Thomas McCabe	High levels of user satisfaction and low defect levels, often associated with low complexity
John Musa	Low defect levels, adherence of software functions to user needs, and high reliability
Bill Perry	High levels of user satisfaction and adherence to requirements

Tabelle 6-1: Verschiedene Definitionen von Softwarequalität

Verschiedene Organisationen haben den Begriff der Softwarequalität standardisiert; hier die Definitionen von IEEE und ISO/IEC:

Definition 6-3 (quality, IEEE Std. 610.12-1990)

(1) The degree to which a system, component, or process meets specified requirements.
(2) The degree to which a system, component, or process meets customer or user needs or expectations.

Definition 6-4 (software quality, ISO/IEC 9126:1991)

The totality of features and characteristics of a software product that bear on its ability to satisfy stated or implied needs.

Die Definition des IEEE, die auch den Entwicklungsprozess einschließt, spiegelt die herstellungsbezogene und die benutzerbezogene Sicht wieder. Die Definition der ISO/IEC lässt den Prozess weg, ist ansonsten aber ähnlich. Es fällt auf, dass beide Definitionen sehr abstrakt sind. Eine universelle Qualitätsdefinition muss allerdings auch abstrakt sein, da es keine detaillierte Produkt-unabhängige Definition von Qualität geben kann (Glass, 1998). Will man die Qualität einer Software bewerten, muss man deren spezifische Anforderungen berücksichtigen.

Klassifikationen der Softwarequalität

Produkt vs. Prozess. Produktqualität ist die Güte des Produkts, Prozessqualität die Güte des Entwicklungsprozesses des Produkts. Beispielweise baut Ludewig (1998) seine Taxonomie der Qualität auf dieser Klassifikation auf. Die Prozessqualität beeinflusst die Produktqualität in der Regel positiv, z. B. die Wartbarkeit (Slaughter, Banker, 1996). Diese Arbeit beschäftigt sich ausschließlich mit der Produktqualität.

Intern vs. extern. Die interne Qualität (oder Wartungsqualität) bezieht sich auf den Entwicklungsprozess und die dabei entstandenen internen Dokumente (z. B. Entwurfsdokumentation). Sie entspricht der Entwicklersicht. Die externe Qualität (oder Gebrauchsqualität) entspricht der Sicht des Benutzers des Programms. Die geforderte externe Qualität ist in den Anforderungen festgehalten, während die geforderte interne Qualität, wenn überhaupt, überwiegend in Richtlinien und Verfahrensweisen der Entwicklungsorganisation dokumentiert ist. Die interne Qualität beeinflusst die externe positiv. In dieser Arbeit liegt der Schwerpunkt auf der internen Qualität.

Mittelbar vs. unmittelbar. Wenn Zwischenprodukte in das Endprodukt einfließen, wie das beim Entwurf der Fall ist, kann man zwischen der unmittelbaren Qualität des Zwischenprodukts und der durch das Zwischenprodukt beeinflussten Qualität des Endprodukts unterscheiden. Beim Entwurf ist z. B. Strukturiertheit eine unmittelbare, Effizienz eine mittelbare Qualität. In dieser Arbeit interessiert eigentlich die mittelbare Qualität des Entwurfs, also die Eigenschaften des Endprodukts, die durch den Entwurf bestimmt sind. Da diese Eigenschaften aber nicht gemessen werden können, bevor eine Implementierung vorliegt, misst man stattdessen Eigenschaften des Entwurfs und verwendet sie zur Vorhersage der Eigenschaften des Endprodukts. Deshalb spielen in der Arbeit beide Kategorien eine Rolle.

6.2 Qualitätsmodelle

The quality of software is measured by a number of totally incompatible criteria, which must be carefully balanced in the design and implementation of every program.
(Hoare, 1981, S. 80)

6.2.1 Definition

Ein Qualitätsmodell bestimmt den allgemeinen Qualitätsbegriff genauer, indem Unterbegriffe (Qualitätsattribute) angegeben werden, aus denen sich die Qualität zusammensetzt. Qualitätsmodelle dienen zur Definition von Qualität, als Qualitätsvorgabe und zur Qualitätsbewertung (Dißmann, 1990). In der Regel werden die Qualitätsattribute hierarchisch angeordnet (vgl. Abbildung 6-1). Die Qualitätsattribute der obersten Stufe werden als Faktoren (factors) bezeichnet, die untergeordneten

Attribute heißen Kriterien (criteria). Die unterste Stufe bilden die Metriken (metrics). Ein solches Modell wird als FCM-Modell (für factors-criteria-metrics) bezeichnet.

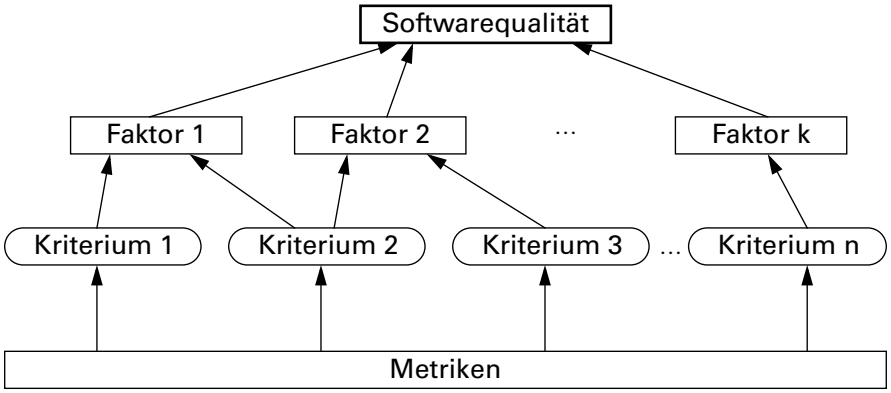

Abbildung 6-1: Aufbau eines Qualitätsmodells (nach Balzert, 1998, S. 257)

In der Software-Engineering-Literatur gibt es einige Vorschläge für Qualitätsmodelle (vgl. Roche, Jackson, 1994). Dabei gibt es im Wesentlichen zwei verschiedene Ansätze für die Gewinnung von Qualitätsmodellen. Der eine Ansatz stellt ein vollständiges, generisches Qualitätsmodell zur Verfügung, aus dem durch Streichungen und Verfeinerungen ein für den eigenen Bedarf passendes Modell generiert werden kann (vgl. Abschnitt 6.2.2). Der andere Ansatz definiert lediglich ein Vorgehensmodell, mit dem ein passendes Qualitätsmodell entwickelt werden kann (vgl. Abschnitt 6.2.3).

6.2.2 Generische Qualitätsmodelle

Die beiden ältesten generischen Qualitätsmodelle von Boehm et al. (1978) und McCall et al. (1977) stammen aus den späten 70er Jahren (später weiterentwickelt von Bowen et al., 1984). Die Standardisierung begann erst Anfang der 90er Jahre mit dem ISO/IEC Standard 9126:1991 und dem IEEE Standard 1061-1992.

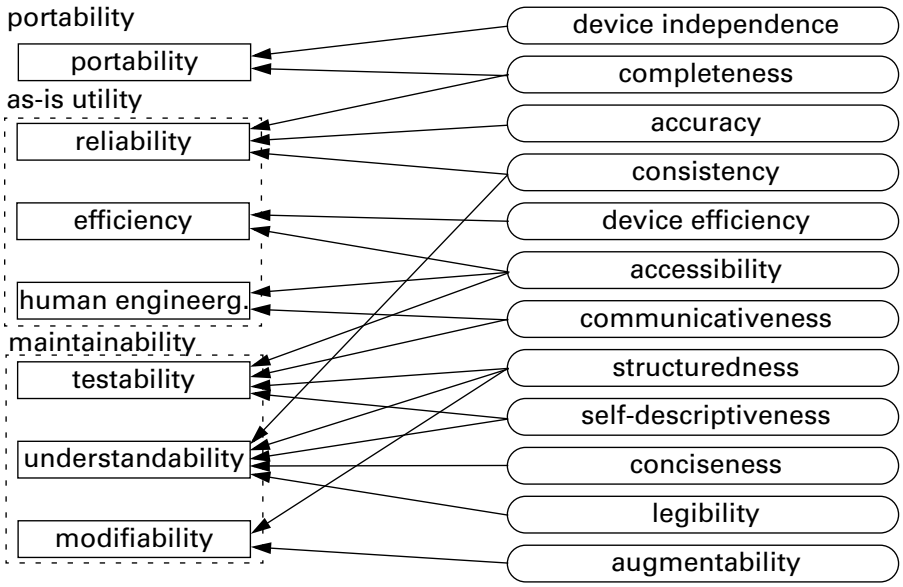

Abbildung 6-2: Qualitätsmodell von Boehm et al.

Boehm et al.

Das Qualitätsmodell von Boehm et al. (1978) ist in Abbildung 6-2 dargestellt. Die Faktoren sind in drei Kategorien eingeteilt: Portabilität, Brauchbarkeit und Wartbarkeit. Die Portabilität enthält nur sich selbst als Faktor; die beiden anderen Kategorien haben jeweils drei Faktoren. Jedem Kriterium sind Code-Metriken (für Fortran) zugeordnet, die vor allem Anomalien aufdecken sollen. Der Schwerpunkt liegt auf der Entwicklersicht, nur die Kategorie Brauchbarkeit gehört zur Benutzersicht.

McCall et al.

McCall et al. (1977; Cavano, McCall, 1978) unterscheiden wie Boehm et al. (1978) drei Kategorien: Anwendung, Änderung und Portierung. Die Kategorien korrespondieren mit den typischen Arbeiten mit und an Software. Die Faktoren werden jeweils einer dieser drei Kategorien zugeordnet (vgl. Abbildung 6-3). Die Kategorie Anwendung entspricht der Benutzersicht, während die anderen beiden Kategorien zur Entwicklersicht gehören.

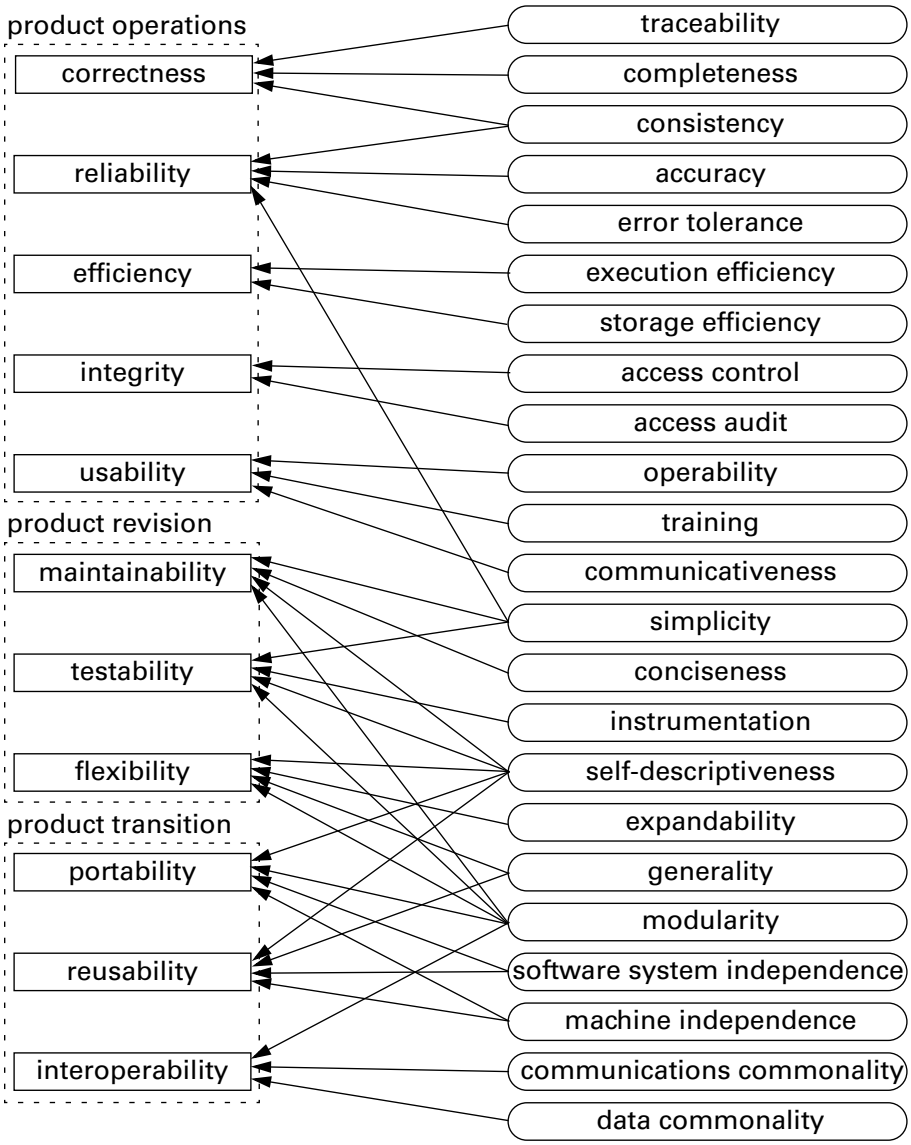

Abbildung 6-3: Qualitätsmodell von McCall et al.

ISO/IEC Standard 9126

Der ISO/IEC Standard 9126 versteht sich vor allem als Richtlinie zur Qualitätsbewertung anhand von Metriken. Um die Zusammenstellung eines Metrikprogramms zu erleichtern, wird ein Qualitätsmodell (ohne Metriken) vorgeschlagen. Die Metriken sollen spezifisch für das konkrete Projekt ergänzt werden.

In den Erläuterungen zum Modell wird darauf hingewiesen, dass Qualität aus verschiedenen Sichten beurteilt werden kann, was eine unterschiedliche Gewichtung der Faktoren nach sich zieht. Es wird die Sicht des Benutzers, des Entwicklers und des Managers unterschieden. Der Schwerpunkt im Modell liegt von der Anzahl der Faktoren her auf der Benutzersicht, denn die ersten vier Faktoren lassen sich dieser zuordnen, während die letzten beiden Faktoren der Entwicklersicht zuzuordnen sind.

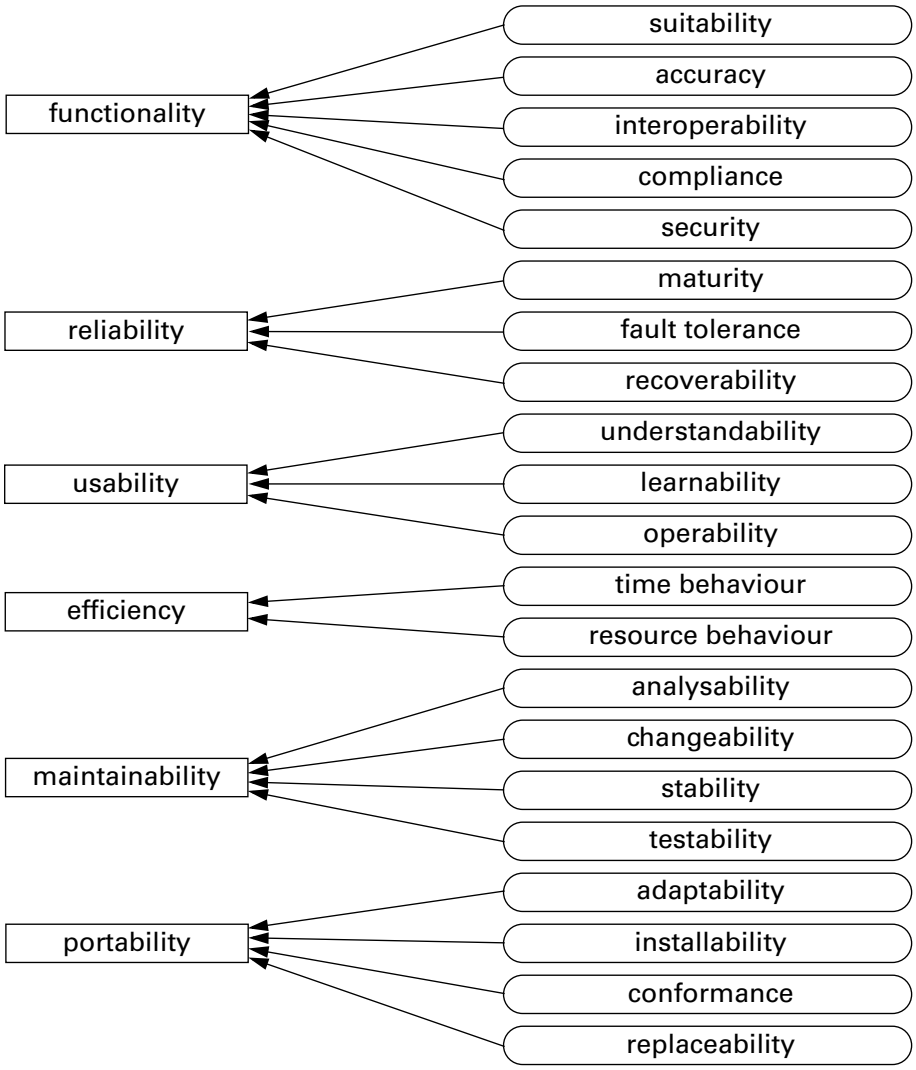

Abbildung 6-4: Qualitätsmodell in ISO/IEC 9126:1991

IEEE Standard 1061

Der Schwerpunkt dieses Standards liegt auf der Umsetzung eines Qualitätsmodells in ein Metrikenprogramm (metrics framework). Im Anhang wird ein Qualitätsmodell vorgeschlagen, das starke Ähnlichkeit mit dem ISO/IEC 9126-Modell hat. Die Faktoren sind dieselben, lediglich in den Kriterien gibt es Unterschiede (vgl. Abbildung 6-5).

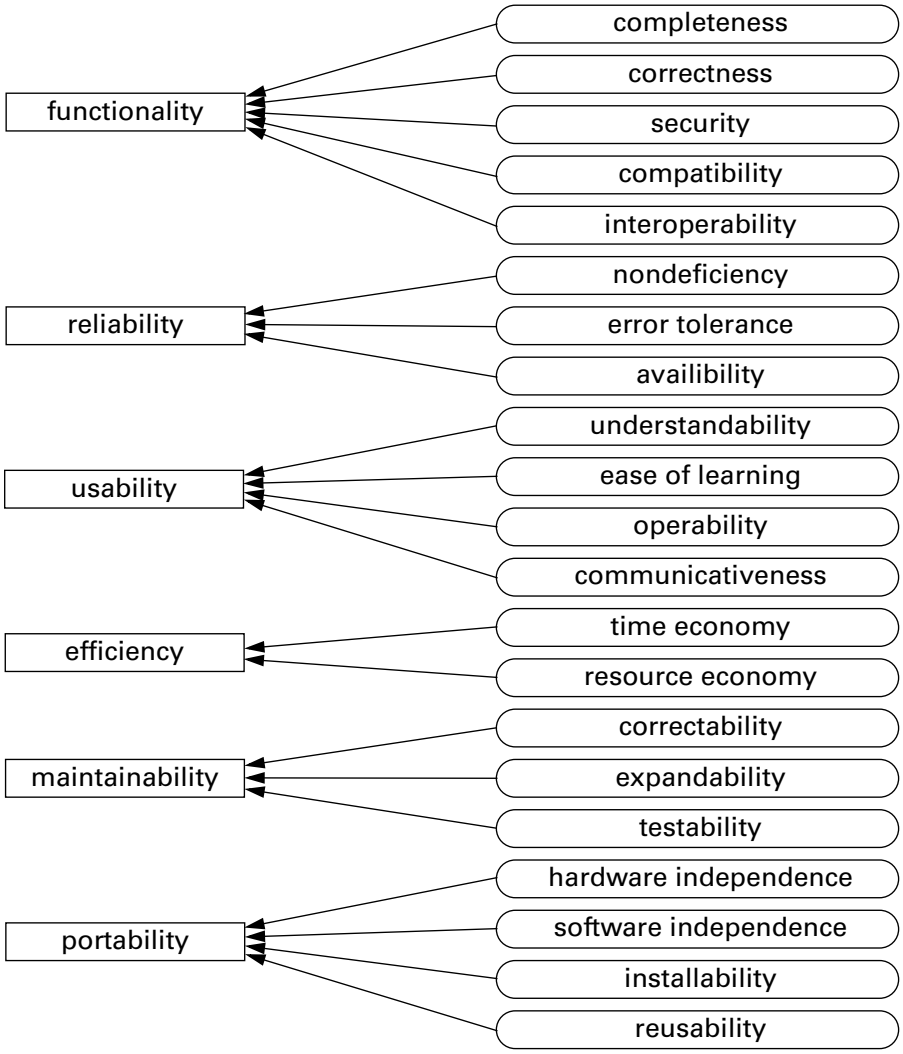

Abbildung 6-5: Qualitätsmodell im IEEE Standard 1061-1992

Zusammenfassung

Die beiden ältesten Modelle sind nicht strikt hierarchisch, weil es Kriterien gibt, die in mehrere Faktoren eingehen. Die neueren Modelle sind dagegen strikt hierarchisch, weil sie so leichter zu verstehen und zu modifizieren sind. Des Weiteren ist eine Verschiebung des Schwerpunkts der Qualitätsmodelle (was die Anzahl der Faktoren angeht) hin zu Faktoren der Benutzersicht zu beobachten. Leider gibt es bei den generischen Modellen immer noch kein allgemein akzeptiertes Modell. Wie man bei den Modellen der ISO/IEC und des IEEE sieht, bewegt man sich aufeinander zu, doch bleiben die Faktoren und Kriterien in ihren konkreten Definitionen umstritten.

6.2.3 Vorgehensmodelle

Beim Ansatz der Vorgehensmodelle wird kein Qualitätsmodell vorgegeben. Stattdessen wird eine Vorgehensweise angegeben, mit der systematisch ein passendes Qualitätsmodell entwickelt wird. Beispiele dieses Ansatzes sind QFD (Quality Function Deployment) und GQM (Goal-Question-Metric).

Quality Function Deployment

QFD wurde erstmals 1972 bei Mitsubishi in der Fabrikation eingesetzt und später von Kogure und Akao auf Software-Produkte übertragen. Der Ansatz ist hauptsächlich in Japan verbreitet (Kogure, Akao, 1983) und gehört zum Bereich des Total Quality Management (TQM). Eine ausführliche Beschreibung von QFD gibt Akao (1990).

Das Vorgehen bei QFD ist wie folgt: Zunächst werden sämtliche Qualitätsanforderungen des Kunden und der zukünftigen Benutzer durch Befragung erhoben. Anschließend werden diese Anforderungen an das Endprodukt (Benutzersicht) in Anforderungen an die Zwischenprodukte (Entwicklersicht) übersetzt. Der Zusammenhang zwischen den beiden Sichten wird für jede Anforderung klar dokumentiert, z. B. in Form von Matrizen. Auf diese Weise ist den Entwicklern immer klar, was eine Anforderung für die beiden Sichten bedeutet.

In Zusammenhang mit Joint Application Development (JAD, Entwicklung unter Integration von Vertretern des Kunden) hat sich diese Methode als sehr effektiv herausgestellt, Fehler in der Spezifikationsphase zu vermeiden (Jones, 1997, S. 266). Haag et al. (1996) berichten über die erfolgreiche Anwendung von QFD bei großen Softwareherstellern.

Goal-Question-Metric

GQM wurde Ende der 80er Jahre im Rahmen des TAME-Projekts (Tailoring a Measurement Environment) von Basili und Rombach (1988) entwickelt. Das Vorgehen nach GQM lässt ein spezifisches Qualitätsmodell entstehen, das aus den Qualitätszielen des Unternehmens oder des Projekts abgeleitet ist. Zuerst werden die Qualitätsziele (goals) erhoben. Anschließend werden Fragestellungen (questions) formuliert, die sich aus den Zielen ergeben. Zum Schluss werden diejenigen Metriken (metrics) festgelegt, welche die Frage beantworten sollen.

Das resultierende Qualitätsmodell besteht nicht aus Qualitätsattributen, sondern aus der Hierarchie von Zielen, Fragestellungen und Metriken. Das GQM-Modell kann besser zur Aufgabe passen, da Qualitätsziele vielschichtiger sein können als Qualitätsattribute. Ein Ziel besteht nämlich aus drei Dimensionen: dem Objekt, dem Qualitätsattribut und dem Blickwinkel (Rombach, 1993). Die Vorgehensweise der Entwicklung eines Qualitätsmodells nach GQM ist aufgrund ihrer Offenheit allerdings sehr schwierig. Um diesen gewichtigen Nachteil abzumildern, werden für jeden Schritt umfangreiche Hilfestellungen in Form von Schablonen, Richtlinien und Prinzipien angeboten. Daskalantonakis (1992, 1994) stellt fest, dass GQM erst ab den Stufen zwei oder drei des Capability Maturity Model funktioniert. Da sich der Großteil aller Organisationen noch auf Stufe eins befindet (Baumert, 1991), ist GQM folglich in vielen Fällen nicht anwendbar.

6.2.4 Fazit

Die generischen Qualitätsmodelle geben ein unspezifisches Qualitätsmodell vor. Nach Auffassung von Rombach (1993) sind generische Modelle zu allgemein, als dass sie wirklich verwendet werden könnten. Auf jeden Fall muss eine Anpassung an den konkreten Bedarf erfolgen. Diese Anpassung wird dadurch erschwert, dass ein generisches Modell, das alle möglichen Anwendungen abzudecken soll, groß und unübersichtlich wird. Das Modell muss vor einer Anpassung zunächst verstanden werden, wobei durch die Größe die Aufmerksamkeit vom Wesentlichen ablenkt werden kann. Die große Breite der generischen Modelle hat allerdings den Vorteil, dass auf alle Aspekte aufmerksam gemacht wird, die berücksichtigt werden könnten.

Dagegen haben Vorgehensmodelle den Vorteil, dass direkt ein für den eigenen Bedarf passendes Qualitätsmodell entsteht, während bei generischen Modellen zunächst eine Anpassung vorzunehmen ist. Allerdings ist es in der Regel einfacher, etwas Vorhandenes anzupassen, als etwas völlig Neues zu schaffen. Das Ergebnis der Anpassung ist zwar meistens nicht so vollkommen wie eine Spezialanfertigung, doch ist der Aufwand bei der Anpassung geringer. Daher wird in dieser Arbeit der Ansatz des generischen Qualitätsmodells verfolgt (vgl. Abschnitt 7.2.3). Die in Abschnitt 6.2.2 gezeigten Qualitätsmodelle sind Modelle für Softwarequalität im Allgemeinen, können also nicht direkt zur Entwurfsbewertung verwendet werden. Allerdings können sie als Ausgangspunkt für die Entwicklung eines Qualitätsmodells für den Entwurf dienen, indem z. B. Faktoren und Kriterien wiederverwendet werden.

6.3 Qualitätssicherung

6.3.1 Qualitätssicherungsmaßnahmen

In the recent struggle to deliver any software at all, the first casualty has been consideration of the quality of the software delivered.
(C. A. R. Hoare, 1981, S. 80)

You can sense it all around you, a software crisis: your bank statement's not right, the PC software has glitches, and the software you've written keeps you up all night. Everyone an feel the problem, but they can't define it. Most software engineers believe there is a crisis, but they haven't been able to figure out what to do to change it.
"The problem is quality!", they cry. Nonsense, quality is the solution to your problem.
(Arthur, 1993, S. xiv)

Qualitätssicherung (quality assurance) dient dazu, die Übereinstimmung eines hergestellten Produkts mit den Anforderungen zu gewährleisten:

Definition 6-5 (quality assurance, IEEE Std. 610.12-1990)

A planned and systematic pattern of all actions necessary to provide adequate confidence that an item or product conforms to established technical requirements.

Zur Qualitätssicherung (QS) lassen sich verschiedene Maßnahmen ergreifen. Im Softwarequalitätsmanagement zerfallen diese in drei Kategorien: organisatorische, konstruktive und analytische (Frühauf et al., 2000; vgl. Abbildung 6-6).

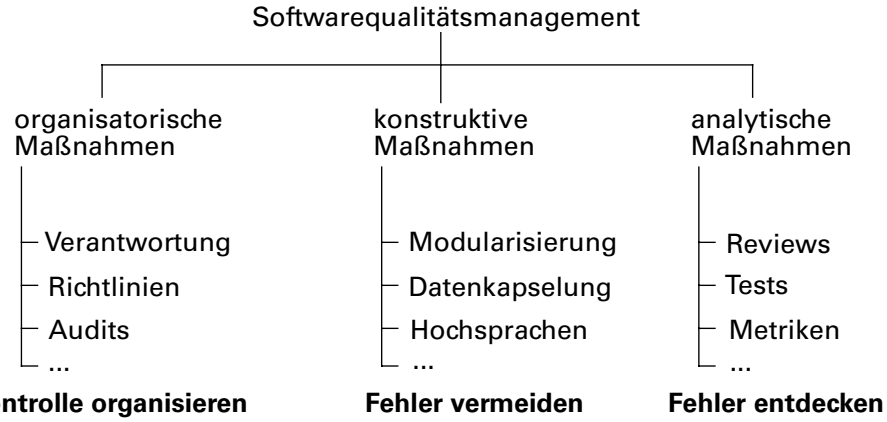

Abbildung 6-6: Qualitätssicherungsmaßnahmen

Die organisatorischen Maßnahmen bilden dabei die Grundlage, auf der die anderen Maßnahmen aufbauen. Es wird ein Qualitätssicherungsprozess etabliert, der festlegt, welche konstruktiven und analytischen Maßnahmen wann von wem durchzuführen sind und welche Richtlinien gelten. Der Prozess selbst wird durch Audits geprüft.

Konstruktive Maßnahmen sollen dafür sorgen, dass das Produkt von Anfang an eine hohe Qualität hat, Qualität also quasi mit „eingebaut“ wird. Dazu werden bestimmte Techniken und Werkzeuge verwendet, die in der Regel zu hoher Qualität führen (z. B. Datenkapselung, Hochsprachen).

Die analytischen Maßnahmen dienen zur Aufdeckung von Qualitätsmängeln, 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. Reviews, Tests und die Erhebung von Metriken.

6.3.2 Reviews

Clearly, inspections are an important way to find errors. Not only are they more effective than testing for finding many types of problems, but they also find them earlier in the program when the cost of making the corrections is far less. Inspections should be a required part of every well-run software process, and they should be used for every software design, every program implementation, and every change made either during original development, in test, or in maintenance.
(Humphrey, 1990, S. 187)

Da es in dieser Arbeit um Entwurfsbewertung geht, wird hier der Bereich der analytischen Qualitätssicherung genauer betrachtet. Analytische Maßnahmen nehmen mehr oder minder explizit eine Bewertung der Qualität des Prüfgegenstands vor, indem nach Mängeln, also Abweichungen vom Soll, gesucht wird. Für den Entwurf hat dabei das Review die größte Bedeutung. Der IEEE Standard 1028-1997 unterscheidet verschiedene Review-Arten: Management-Review, Audit, technisches Review, Inspektion und Walkthrough; für die Produktbewertung sind aber nur die letzten drei relevant. Diese Verfahren sind relativ ähnlich; die wesentlichen Unterschiede liegen in der Zielsetzung und der Art der Durchführung, z. B. ob Lösungen für Mängel oder Alternativen diskutiert werden oder nicht. Für alle drei Verfahren gilt, dass eine ganze Gruppe von Menschen daran beteiligt ist, so dass für Vorbereitung, Durchfüh-

rung und Nachbereitung viel Personal und Arbeitszeit benötigt wird. Daher ist eine weitgehende Werkzeugunterstützung wünschenswert (z. B. bei der Identifizierung von Mängeln), um den erforderlichen Aufwand zu reduzieren. Die Automatisierbarkeit von Reviews ist allerdings begrenzt, da sich viele Mängel gar nicht oder nur unzureichend automatisch erkennen lassen.

6.3.3 Good Enough Quality

We don't believe in striving for the ideal software architecture. Instead, the goal should be to design a good architecture – one in which, when the system is implemented according to the architecture, it meets its requirements and resource budgets. This means that is must be possible to implement the system according to the architecture. So an architecture that isn't explicit, comprehensive, consistent, and understandable is not good enough.
(Hofmeister et al., 2000, S. 7)

Qualitätssicherung verursacht Kosten. Diese Kosten sind idealerweise aber geringer als die Fehlerfolgekosten der Qualitätsmängel, die dank der Qualitätssicherung vor der Auslieferung verhindert oder behoben wurden (z. B. Kosten für Fehlersuche und Fehlerbehebung). Um eine kosteneffektive Qualitätssicherung durchzuführen, müssen Qualitätssicherungskosten und Fehlerfolgekosten gegeneinander abgewogen werden (Ludewig, 1994), um ein Kostenoptimum zu erreichen (vgl. Abbildung 6-7).

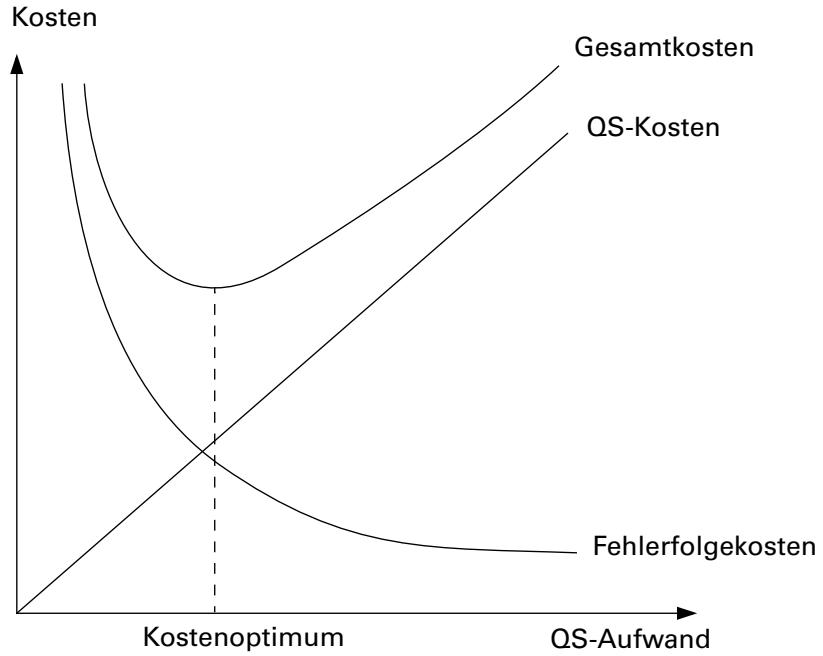

Abbildung 6-7: Kostenoptimum bei der Qualitätssicherung

Die Kosten sind aber nicht der einzige limitierende Faktor für die Qualität. Das „magische Dreieck" sieht die Qualität in Konkurrenz zu Kosten und (Liefer-)Zeit, das „magische Viereck" nimmt noch den (Funktions-)Umfang als Optimierungsgröße hinzu (vgl. Abbildung 6-8). Wird hier eine Größe (z. B. Qualität) optimiert, geht das immer zu Lasten der anderen Größen. Daher muss ein optimaler Kompromiss gefunden werden.

Abbildung 6-8: Magisches Dreieck und Viereck

Die Idee der „Good Enough Quality“ ist es, bei der Qualität nicht die größtmögliche anzustreben, sondern nur die notwendige. Dazu wird mit dem Kunden über den gewünschten Kompromiss zwischen den konkurrierenden Zielen des magischen Drei- oder Vierecks verhandelt (Yourdon, 1995). Entscheidend ist dabei, dass der Kunde die Entscheidung fällt, nicht die Entwickler, wie wichtig welches Ziel ist. Die Entwickler sagen dem Kunden nur, was machbar ist und was nicht.

Auch innerhalb der Qualität gilt es abzuwägen, denn Qualität ist multidimensional (Bosch, 2000). Bass et al. (1998, S. 75) stellen dazu fest: „No quality can be maximized in a system without sacrificing some other quality or qualities.“ Das erfordert einen Kompromiss zwischen den verschiedenen Qualitäten. Der optimale Kompromiss kann leichter ermittelt werden, wenn klar ist, welche Qualitäten in welchem Umfang gefordert sind. Ein projektspezifisches Qualitätsmodell implementiert dann diesen Kompromiss z. B. in Form einer Gewichtung.

Kapitel 7

Entwurfsqualität

Software design is not easy – not easy to do, teach, or evaluate. Much of software education these days is about products and APIs, yet much of these are transient, whereas good design is eternal – if only we could figure out what good design is.
(Fowler, 2001a, S. 97)

In diesem Kapitel wird die Frage, was eigentlich ein guter Entwurf ist, aus verschiedenen Perspektiven beleuchtet. Zunächst wird ein kleines Beispiel vorgestellt, in dem drei Entwurfsalternativen für die gleiche Aufgabenstellung miteinander verglichen werden, wobei der Entwurf auf intuitiver Basis – unterstützt durch Entwurfsregeln – bewertet wird. Analog zu den in Kapitel 6 vorgestellten Qualitätssichten werden dann die verschiedenen Sichten bei der Entwurfsqualität herausgearbeitet. Anschließend wird auf Entwurfsregeln (Prinzipien und Heuristiken) des objektorientierten Entwurfs eingegangen. Diese enthalten Erfahrungswissen, wie man zu einem guten (bzw. besseren) Entwurf kommt – sofern die Anwendung einer Regel im aktuellen Kontext sinnvoll ist. Schließlich wird auf die Frage eingegangen, wie Qualitätssicherung und Entwurfsbewertung durchgeführt werden können, wenn erst einmal klar ist, welche Kriterien relevant sind.

7.1 Ein Beispiel

A good design provides a solution that is no more complex than the problem it solves. A good design is based on deep simplicities, not on simple-mindedness.
(Linger et al., 1979)

Auf der Suche nach einem guten Entwurf stößt man häufig auf mehrere Alternativen, unter denen auszuwählen ist. Das folgende Beispiel (basierend auf einem Beispiel von Fowler et al., 1999) zeigt, dass es bei der Entscheidung, welcher Entwurf für ein gegebenes Problem der beste ist, viele Kriterien zu berücksichtigen gibt und dass diese Kriterien im Widerstreit zueinander stehen können.

Ein Videoverleih soll ein System zur Rechnungsstellung erhalten. Kunden, Ausleihen und Filme werden durch Klassen modelliert (Customer, Rental, Movie). Der Kunde hat

Ausleihen, während die Ausleihe Informationen über den ausgeliehenen Film und die Leihdauer (in Tagen) hat. Ein Film hat einen Preiscode, der zusammen mit der Leihdauer den Preis bestimmt. Es gibt Preiscodes für normale Filme (regular), Kinderfilme (children's) und Neuerscheinungen (new release). Der Preiscode eines Films kann sich ändern, z. B. von „new release" nach „regular", daher gibt es in Movie eine Operation setPriceCode.

Erste Stufe: Entkopplung und Kapselung

Entwurf A (vgl. Abbildung 7-1) sieht vor, den Rechnungsbetrag in der Klasse Customer zu berechnen und dazu die erforderliche Information bei Rental und Movie einzuholen. Customer holt sich in der Methode statement von Rental die Ausleihdauer und den Film, von diesem wiederum den Preiscode (vgl. Abbildung 7-2).

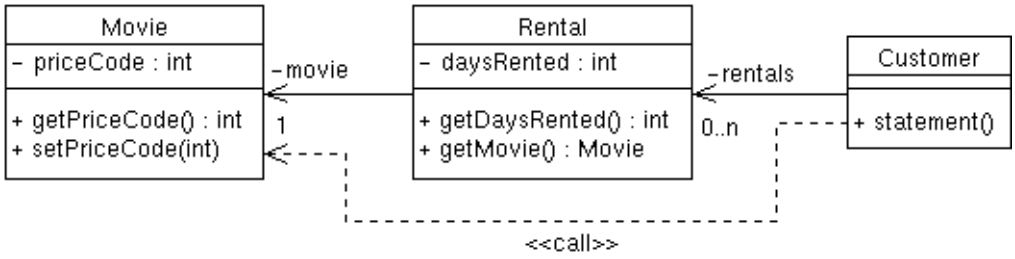

Abbildung 7-1: Klassendiagramm für Entwurf A

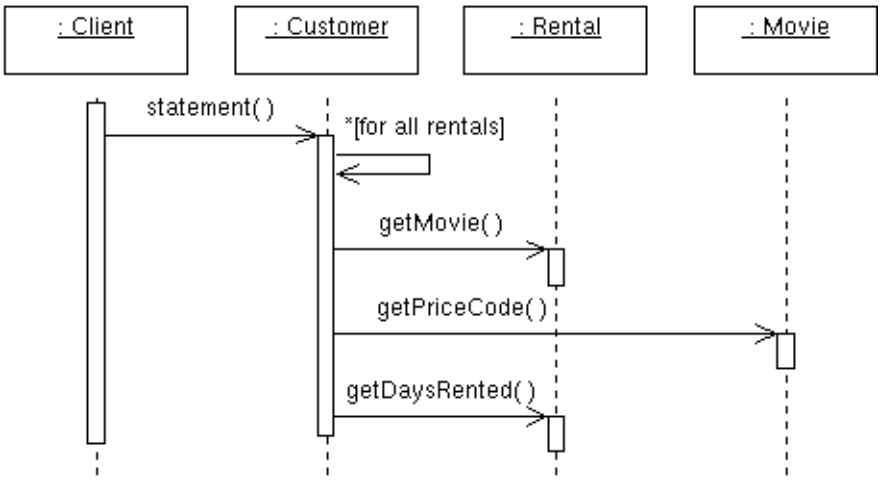

Abbildung 7-2: Sequenzdiagramm für Entwurf A

Dieses Vorgehen ist ein Verstoß gegen das Demetergesetz (Lieberherr, Holland, 1989), weil bei einem Objekt, das von einer anderen Klasse (Rental) als Resultat eines Methodenaufrufs (getMovie) geliefert wurde, eine Methode aufgerufen wird (getPriceCode). Dadurch entsteht eine unnötige Kopplung zwischen Customer und Movie, die als Benutzungsbeziehung mit Stereotyp «call» im Klassendiagramm sichtbar ist. Um diese zu vermeiden, sollte stattdessen über Rental delegiert werden. Außerdem lässt sich feststellen, dass die Preisberechnung für einen einzelnen Ausleihvorgang logisch gesehen eigentlich zu Rental gehört, da dort bereits alle erforderlichen Informationen vorliegen. In statement müssten die einzelnen Beträge dann nur noch aufsummiert werden. Die Preisberechnung anhand des Preiscodes ist bei Movie am besten aufgehoben, da sich so neue Preiscodes besser kapseln lassen (Verbergen einer Entwurfsent-

scheidung nach dem Geheimnisprinzip; Parnas, 1972b). Die zur Berechnung benötigte Leihdauer muss nun beim Aufruf als Parameter übergeben werden.

Werden die genannten Verbesserungen an Entwurf A umgesetzt, entsteht Entwurf B (vgl. Abbildung 7-3 und Abbildung 7-4). Im Vergleich zu Entwurf A besteht zwischen den Klassen weniger Kopplung (Customer weiß nichts mehr von Movie). Die Kapselung von Movie und Rental wurde ebenfalls verbessert, so dass nun auch die drei get-Methoden, die von statement vorher benötigt wurden, entfallen können (getPriceCode, getDaysRented, getMovie). Die Berechnung des Rechnungsbetrags ist stärker dezentralisiert, so dass das Problem der „God Class“ (Riel, 1996), die den Entwurf dominiert und alle Funktionalität an sich zieht, vermieden wird. Auf der anderen Seite ist die Berechnung nun über drei Klassen „verschmiert“, so dass alle drei Klassen betrachtet werden müssen, um den Algorithmus nachvollziehen zu können. Bei Entwurf A genügte im Wesentlichen die Betrachtung der Klasse Customer.

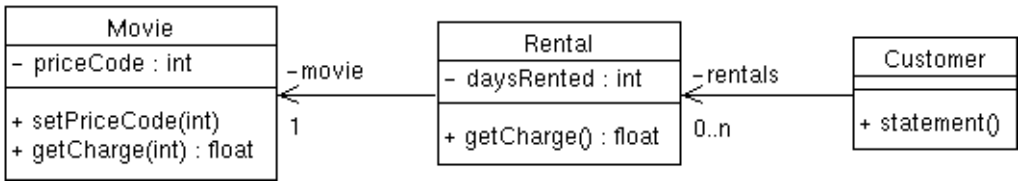

Abbildung 7-3: Klassendiagramm für Entwurf B

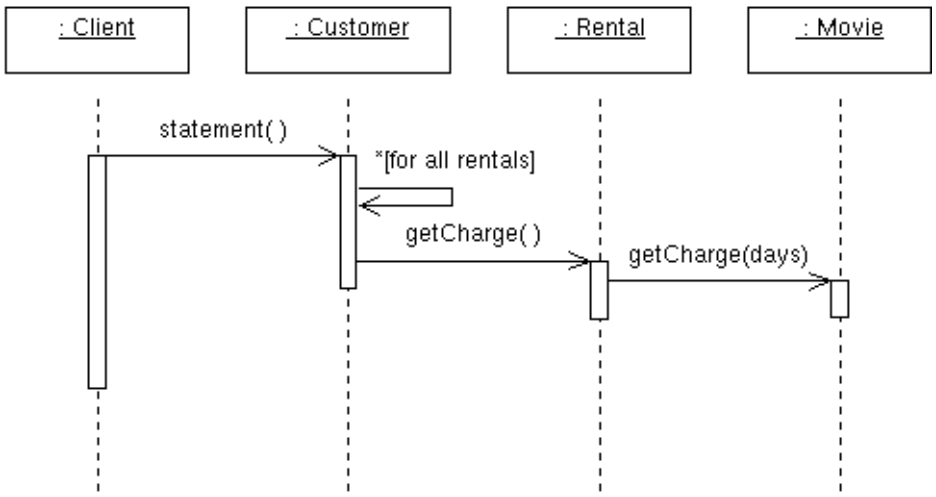

Abbildung 7-4: Sequenzdiagramm für Entwurf B

Zweite Stufe: State-Muster für Movie

Entwurf C entsteht aus B durch die Anwendung des State-Musters (Gamma et al., 1995). Die Preisberechnung für einen Film wird in eine abstrakte Klasse Price ausgelagert. Zu Price gibt es für jeden Preiscode eine konkrete Unterklasse, die in der Methode getCharge den spezifischen Preis berechnen kann (vgl. Abbildung 7-5). Die resultierende Kommunikation ist in Abbildung 7-6 dargestellt.

Der Vorteil dieser Lösung ist, dass getCharge in Movie leichter zu implementieren ist, da die bisher explizit im Code zu formulierende Fallunterscheidung nach dem Preiscode jetzt durch dynamisches Binden erledigt wird. Die Lösung kann daher einfacher um neue Preiscodes erweitert werden. Außerdem ist die Preisberechnung für einzelne

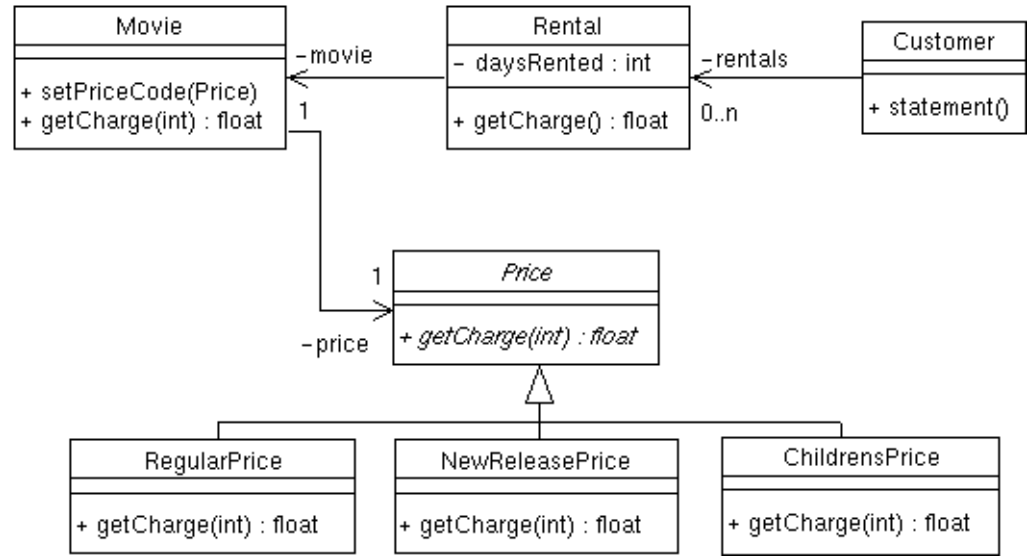

Abbildung 7-5: Klassendiagramm für Entwurf C

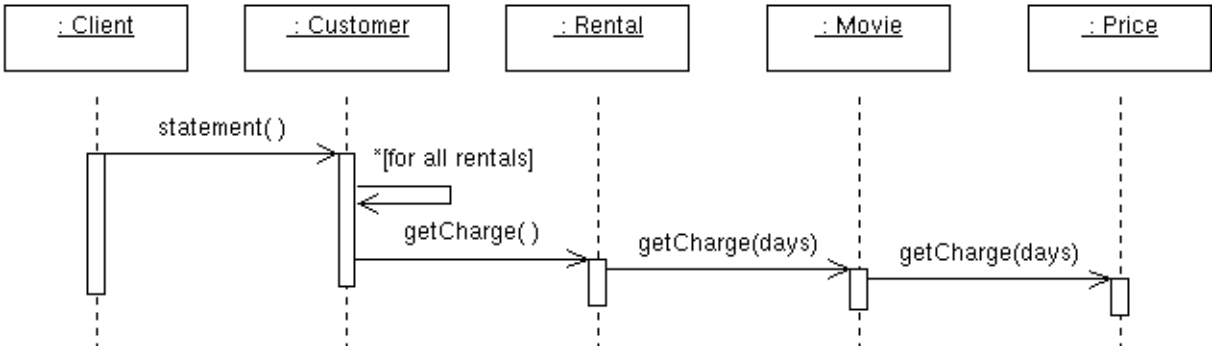

Abbildung 7-6: Sequenzdiagramm für Entwurf C

Preiscodes leichter zu ändern. Die Trennung zwischen den Geschäftsregeln und der restlichen Programmlogik (Separation of Policy and Implementation; Buschmann et al., 1996, S. 401) ist hier am besten umgesetzt. Der Preis dafür ist allerdings hoch: Vier neue Klassen, so dass sich die Anzahl der Klassen im gezeigten Ausschnitt mehr als verdoppelt. Die Verständlichkeit wird dadurch wieder verschlechtert.

Metriken

Für die drei Entwürfe können nun die bei Empirikern beliebten Klassenmetriken von Chidamber und Kemerer (1994) erhoben werden (zur Beschreibung der Metriken siehe Abschnitt 5.5.2). Für die Erhebung der Metriken muss für jede Methode bekannt sein, auf welche Attribute sie zugreift und welche Methoden sie aufruft. Die Methodenaufrufe können hier aus dem Sequenzdiagramm entnommen werden. Dagegen können die Zugriffe auf Attribute aus den UML-Diagrammen nicht abgelesen werden. Daher wird mit Vermutungen über den Zugriff auf die gezeigten privaten Attribute und zusätzliche angenommene[1] Attribute gearbeitet, um die LCOM-Metrik berechnen zu können. Tabelle 7-1 zeigt die Messwerte.

1. Es wird angenommen, dass zur Implementierung der Assoziationen in der Klasse, von der die Assoziation ausgeht, jeweils ein Attribut vorhanden ist.

	WMC			DIT			NOC			CBO			RFC			LCOM		
Klasse	A	B	C	A	B	C	A	B	C	A	B	C	A	B	C	A	B	C
Movie	2	2	2	0	0	0	0	0	0	1	0	1	2	2	3	0	0	0
Rental	2	1	1	0	0	0	0	0	0	2	2	2	2	2	2	1	0	0
Customer	1	1	1	0	0	0	0	0	0	2	1	1	4	2	2	0	0	0
Price	-	-	1	-	-	0	-	-	3	-	-	1	-	-	1	-	-	0
RegularPrice	-	-	1	-	-	1	-	-	0	-	-	0	-	-	1	-	-	0
NewReleasePrice	-	-	1	-	-	1	-	-	0	-	-	0	-	-	1	-	-	0
ChildrensPrice	-	-	1	-	-	1	-	-	0	-	-	0	-	-	1	-	-	0
Durchschnitt	1,7	1,3	1,1	0	0	0,4	0	0	0,4	1,3	1,0	0,7	2,7	2,0	1,5	0,3	0	0

Tabelle 7-1: Messwerte der Chidamber/Kemerer-Metriken

Da die Metriken Komplexitätsmaße sind, bedeutet ein geringerer Wert eine bessere Bewertung. DIT und NOC zeigen deutlich, dass hier sehr wenig mit Vererbung gearbeitet wird. Das gilt allerdings fast immer für solche kleinen Beispiele. WMC, CBO und RFC spiegeln die Verbesserung von A nach B am besten wieder, LCOM teilweise. Der Preis der Flexibilisierung in C wird durch die Zunahme bei CBO und RFC für Movie deutlich.

Betrachtet man die Durchschnitte der Metriken über die drei Entwürfe, zeigt sich bei WMC, CBO und RFC ein Trend zugunsten von Entwurf C, der aber auch durch die deutlich höhere Anzahl von Klassen im Vergleich zu A und B bedingt ist. Bei DIT und NOC jedoch verläuft der Trend zu Ungunsten von Entwurf C.

Schlussfolgerungen

Die Änderungen von Entwurf A nach B und von B nach C dienen der Entkopplung und damit der Änderbarkeit. Bei Entwurf C nimmt die Zahl der Klassen aber stark zu und verschlechtert so die durch die Entkopplung verbesserte Verständlichkeit wieder. Es wird Flexibilität eingebaut, die aber nur dann wirklich sinnvoll ist, wenn sich die Preiscodes tatsächlich ändern werden. Die Güte des Entwurfs hängt also auch vom Kontext der Software ab. Mit Hilfe der erhobenen Metriken lassen sich bei C nur Nachteile, aber keine Vorteile feststellen (außer bei einer fragwürdigen Durchschnittsbildung). Das zeigt auch, dass sich nicht alle Qualitätsattribute gleich gut quantifizieren lassen. Eine vollständig automatisierte Entwurfsbewertung dürfte damit schwierig, wenn nicht gar unmöglich sein.

Das Beispiel verdeutlicht, dass es Grundsätze gibt, die fast immer sinnvoll sind, z. B. Entkopplung sowie Kapselung von Details und von sich wahrscheinlich ändernden Entwurfsentscheidungen. Diese Grundsätze sollten immer beachtet werden, wenn dabei nur geringe Mehrkosten entstehen. Flexibilität, insbesondere durch Entwurfsmuster, ist oft teuer und sollte daher nur eingebaut werden, wenn sie benötigt wird. Häufig muss dazu auch der vorhandene Entwurf geändert werden: Beispielsweise kann das State-Muster auf Entwurf A nicht ohne vorhergehende Änderungen angewendet werden.

Eine Entwurfsbewertung sollte nicht nur von den vorhandenen Strukturen und den aktuellen Anforderungen abhängen, sondern auch den Kontext einbeziehen. Dieser besteht unter anderem aus

- den impliziten Anforderungen und Rahmenbedingungen (z. B. die Wahrscheinlichkeit, dass sich bestimmte Anforderungen ändern),
- bei den Entwicklern vorhandenes Wissen um Entwurfs- und Implementierungstechniken,
- vorhandenen Software-Bausteinen (Komponenten),
- in der Entwicklung eingesetzten Werkzeugen und
- wirtschaftlichen Überlegungen (z. B. Time-to-Market, Beschäftigung von Mitarbeitern, Schulungskosten).

Jeder am Entwurf Beteiligte oder von ihm unmittelbar Betroffene (Entwerfer, Implementierer, Manager etc.) hat andere Ansprüche an den Entwurf. Nur mittelbar vom Entwurf Betroffene (Kunde, Anwender) haben zwar keine Ansprüche direkt an den Entwurf, aber an die Implementierung, die ja vom Entwurf maßgeblich geprägt wird. So ist es nur natürlich, dass es unterschiedliche Auffassungen über Entwurfsqualität gibt. Dies wird im folgenden Abschnitt vertieft.

7.2 Perspektiven der Entwurfsqualität

Entwurfsqualität ist durch verschiedene Perspektiven geprägt:

1. Zeitliche Perspektive: kurz- oder langfristig
2. Interessengruppe: Kunde, Anwender, Entwickler, Projektmanager oder Projekteigentümer
3. Qualitätssicht: transzendent, produktbezogen, benutzerbezogen, herstellungsbezogen oder kostenbezogen

Es ist praktisch nicht möglich, für jede dieser Perspektiven alle Möglichkeiten in einem einzigen allgemeinen Modell zusammenzubringen. Stattdessen ist es sinnvoller, eine Menge spezifischer Qualitätsmodelle zu erstellen (Dißmann, 1990).

7.2.1 Zeitliche Perspektive

A sign that the Software Engineering profession has matured will be that we lose our preoccupation with the first release and focus on the long term health of our products.
(Parnas, 1994, S. 279)

Die Kriterien, nach denen ein Entwurf bewertet wird, hängen davon ab, ob eine kurz- oder eine langfristige Perspektive eingenommen wird. Die kurzfristige Perspektive betrachtet, wie schwierig es ist, den Entwurf gemäß den Anforderungen zu erstellen und zu realisieren. Die langfristige Perspektive hingegen betrachtet die Entwurfsqualität über die gesamte Lebenszeit der Software hinweg. Der Entwurf wird zwar am Ende der Lebenszeit nicht mehr derselbe sein, da er oft überarbeitet werden wird. Doch wird bei einem guten Entwurf viel von der ursprünglichen Struktur erhalten bleiben. In der Regel ist die langfristige Perspektive – auch aus ökonomischer Sicht – für alle Interessengruppen die vernünftigere.[2]

7.2.2 Interessengruppe

What drives the end user, and as a result determines his or her view of the software development, differs substantially from the view of the system or software organizations.
(Evans, Marciniak, 1987, S. 45)

Die Qualitätsdefinition hängt stark von der Interessengruppe ab (Boehm, In, 1996). Bei der Betrachtung der Güte eines Entwurfs gibt es verschiedene Interessengruppen, für die jeweils unterschiedliche Aspekte des Entwurfs relevant sind. Die folgenden Gruppen lassen sich identifizieren:

- Kunde: Der Kunde gibt das Produkt in Auftrag und bezahlt es. Er hat ein Interesse daran, für möglichst wenig Geld ein Produkt zu erhalten, das seinen Anforderungen entspricht. Das bedeutet für den Entwurf, dass er vollständig und korrekt im Sinne der Spezifikation sein soll. Außerdem soll er auch eine gewisse Zukunftssicherheit aufweisen, weshalb Wartbarkeit (vor allem für korrektive und adaptive Wartung) wichtig ist.
- Anwender: Der Anwender verwendet das Produkt. Für ihn ist ebenfalls Vollständigkeit und Korrektheit des Systems relevant. Benutzerfreundlichkeit ist für ihn besonders wichtig.
- Entwickler: Die Entwickler sind für die Herstellung des Produkts zuständig. Hier gibt es verschiedene Untergruppen, die unterschiedliche Anforderungen an den Entwurf haben:
  - Entwerfer: Der Entwerfer muss sich um alle Eigenschaften des Entwurfs kümmern, da er dafür zuständig ist, sie in den Entwurf „einzubauen“. Für seine eigene Arbeit ist Änderbarkeit und Verständlichkeit besonders wichtig, da es wahrscheinlich bereits in der Entwurfsphase zu Überarbeitungen des Entwurfs kommen wird. Vollständigkeit, Konsistenz und Prüfbarkeit der Entwurfsdokumentation ist für ihn auch von Bedeutung.
  - Implementierer: Der Implementierer sieht nur den Ausschnitt des Entwurfs, den er zu implementieren hat. Dieser Ausschnitt muss für ihn aber hinreichend verständlich sein, um eine Umsetzung in der gewünschten Programmiersprache mit den vorgesehenen Werkzeugen und Softwarekomponenten zu erlauben. Die Implementierung wird erleichtert, wenn sich die Komponenten des Entwurfs unabhängig voneinander entwickeln lassen (Entkopplung), da dann weniger Absprachen mit anderen Implementierern notwendig sind.
  - Tester: Der Tester testet zunächst die einzelnen Komponenten des Entwurfs. Auch hier ist Entkopplung wichtig, da sonst zu viele Komponenten integriert werden müssen, um ein ausführbares Teilsystem zu erhalten. Für die Fehlersu-

2. Projektmanager und -eigentümer könnten bei reinen Entwicklungsaufträgen die kurzfristige Perspektive einnehmen, denn nur, wenn auch die Wartung durch den Auftragnehmer erfolgt, lohnt sich die langfristige Perspektive. Das ist allerdings zu kurz gedacht, denn man möchte in der Regel die Geschäftsbeziehung mit dem Kunden aufrecht erhalten. Stellt der Kunde aber fest, dass das ihm gelieferte Produkt zwar alle gewünschten Funktionen aufweist, sich aber allen späteren Änderungsversuchen widersetzt, wird er keine Folgeaufträgen vergeben. Daher ist die Ersparnis durch den kurzfristig ausgelegten Entwurf mit den ausbleibenden Gewinnen aus Folgeaufträgen zu verrechnen, was in der Regel insgesamt zu einem Verlust führen wird.

che nach dem Systemtest ist es praktisch, wenn der Entwurf Möglichkeiten zur Diagnose (z. B. Debug-Ausgaben) vorsieht. Dann können Fehlerursachen leichter eingegrenzt werden.

  - Wartungsentwickler: Der Wartungsentwickler interessiert sich vor allem für die Verständlichkeit und die Änderbarkeit des Systems. Kleine Änderungen, insbesondere vorhersehbare Änderungen, sollten mit geringem Aufwand vorgenommen werden können. Außerdem sollte eine solche Änderung möglichst nur Auswirkungen auf eine Komponente haben. Der gefürchtete Welleneffekt[3] sollte höchstens lokal, d. h. innerhalb einer Komponente, auftreten.

- Projektmanager: Der Projektmanager möchte, dass zur Realisierung des Entwurfs möglichst geringe Ressourcen notwendig sind. Der Entwurf und die Realisierung sollen möglichst schnell gehen, damit der knappe Terminplan eingehalten werden kann, es sollen alle im Team verfügbaren Kräfte (aber nicht mehr) eingesetzt werden, und die Ausgaben für Werkzeuge, Softwarekomponenten, Schulung etc. sollen gering sein. Natürlich soll das entworfene System auch den Kunden zufrieden stellen, also korrekt und vollständig sein – soweit das dazu nötig ist.
- Projekteigentümer: Der Projekteigentümer möchte mit möglichst geringen Kosten eine möglichst hohe Kundenzufriedenheit zu erreichen. Allerdings kann er auch eine langfristige, projektübergreifende Position einnehmen und z. B. die Entwicklung wiederverwendbarer Komponenten im Projekt fördern, so dass die Wiederverwendbarkeit des Entwurfs (oder von Teilen daraus) eine Rolle spielt.

7.2.3 Qualitätssicht

Wie bei dem allgemeinen Qualitätsbegriff gibt es auch bei der Entwurfsqualität verschiedene Möglichkeiten, sich der Sache zu nähern. Hier werden die fünf Qualitätssichten aus Abschnitt 6.1 auf den Entwurf übertragen.

Die transzendente Sicht: Eleganz und Schönheit

Beauty is more important in computing than anywhere else in technology. [...] Beauty is important in engineering terms because software is so complicated. Complexity makes programs hard to build and potentially hard to use; beauty is the ultimate defense against complexity.
(Gelernter, 1998, S. 22)

Die Qualität des Entwurfs wird hier mit der wahrgenommenen Schönheit und Eleganz gleichgesetzt. Die Schönheit eines technischen Gegenstandes (einer Maschine) wird nach Gelernter (1998) durch zwei Aspekte bestimmt: Kraft (power) und Einfachheit (simplicity). Ein Gegenstand, der Kraft besitzt, lässt sich für viele Zwecke einsetzen; und für diese eignet er sich gut. Einfachheit trägt dazu bei, dass der Umgang mit dem Gegenstand leicht erlernbar ist und oft auch als natürlich empfunden wird. Einfachheit bedingt auch Eleganz, da hier ein Zweck mit wenigen Mitteln erreicht wird.

3. Eine Änderung zieht in der Regel Folgeänderungen nach sich, diese wiederum Folgeänderungen etc. Der Name Welleneffekt (ripple effect) kommt daher, dass sich die Änderungen im System ausbreiten wie die Wellen auf einer Wasseroberfläche nach dem Aufschlag eines Steins.

Gelernter fordert, auch bei Software Schönheit anzustreben. Dies bezieht sich sowohl auf die äußere Gestaltung (die Benutzungsoberfläche) als auch auf den inneren Aufbau (den Entwurf). Auch andere Autoren, davon viele Anhänger der Entwurfsmuster-Bewegung, sehen einen Zusammenhang zwischen Schönheit und Qualität. Da die Idee der Entwurfsmuster aus der Architektur kam, werden gerne Analogien zur Architektur bemüht (z. B. Fujino, 1999).

Ein guter Entwurf gemäß der transzendenten Sicht besitzt Eleganz und Schönheit. Diese sind etwas für den erfahrenen Entwerfer Wahrnehmbares, was aber nicht fassbar oder gar quantifizierbar ist. Darum sind diese Eigenschaften für ein Qualitätsmodell leider unbrauchbar. Gelernter (1998, S. 33) schreibt: „You know beauty when you feel it; there is no way to *demonstrate* its presence. What we *can* do, though, is point out some of the telltales of simplicity and power." Was also benötigt wird, sind messbare Eigenschaften, aus denen heraus sich Eleganz und Schönheit begründen lassen. Dies geht aber bereits in die Richtung der produktbezogenen Sicht.

Die produktbezogene Sicht

Die Qualität des Entwurfs wird eindeutig über ausgewählte, messbare Eigenschaften seiner selbst und seiner Bestandteile definiert. Im Gegensatz zur herstellungsbezogenen Sicht (siehe unten) sind diese Eigenschaften aber nicht von der Spezifikation des Systems abhängig, sondern es handelt sich um allgemeine Eigenschaften, die jeder Entwurf haben sollte.

Die benutzerbezogene Sicht

Die benutzerbezogene Sicht wird durch die Zugehörigkeit des Verwenders des Entwurfs zu den bereits diskutierten Interessengruppen bestimmt. Jede Interessengruppe hat ihre eigene Qualitätsdefinition, die darüber hinaus noch für jeden Vertreter einer Gruppe variieren kann. Die benutzerbezogene Sicht ist subjektiv.

Die herstellungsbezogene Sicht

Die Beurteilung der Qualität erfolgt anhand der Qualitätsanforderungen aus der Spezifikation. Damit ist der Qualitätsbegriff des Entwurfs spezifisch für jede Spezifikation. Bosch (2000) nimmt diese herstellungsbezogene Sicht ein und beschreibt mehrere Verfahren, um einen Soll-Ist-Vergleich des Entwurfs hinsichtlich der Qualitätsanforderungen durchzuführen (siehe Abschnitt 7.6).

Die kostenbezogene Sicht

The ultimate goal of Software Engineering (like any engineering) is to achieve highest benefits at lowest cost. Software quality is desirable exactly to the degree it contributes to that goal.
(Ludewig, 1994, S. 15)

A good design is one that balances trade-offs to minimize the total cost of the system over its entire lifetime.
(Coad, Yourdon, 1991, S. 128)

Die kostenbezogene Qualitätssicht nimmt bei der Qualitätsbewertung die Kosten für das Erreichen eines bestimmten Qualitätsniveaus (z. B. in Bezug auf die herstellungsbezogene Sicht) hinzu. Wenn beispielsweise die Qualität des Entwurfs mit einer möglichst geringen Zahl von Fehlern gleichsetzt wird, besitzt der Entwurf ohne Fehler die

höchste Qualität. Erfahrungsgemäß nehmen aber die Qualitätssicherungskosten exponentiell zu, wenn eine sehr geringe Fehlerrate angestrebt wird. Also ist die Qualität mit den Kosten in Beziehung zu setzen (vgl. Abschnitt 6.3.3). Die Einbeziehung der Kosten in die Qualitätsbetrachtung kann so weit getrieben werden, dass die Qualität mit den Kosten über die gesamte Lebenszeit der Software gleichgesetzt wird.[4]

Die Ermittlung der Gesamtkosten ist aber während der Erstellung des Entwurfs nicht möglich, weil die vom Entwurf bestimmten Kosten hauptsächlich erst in späteren Phasen anfallen. Da die Kosten nicht direkt gemessen werden können, werden traditionell Kriterien gemessen, von denen angenommen wird, dass sie mit den Kosten korreliert sind, also Indikatoren darstellen. Daher stützen sich alle brauchbaren Definitionen von Entwurfsqualität auf Indikatoren.

Im Laufe der Zeit sind viele solcher Indikatoren in Form von erwünschten Eigenschaften vorgeschlagen worden, z. B. Konsistenz, Verfolgbarkeit (traceability) und Wiederverwendbarkeit. Die Zusammenhänge zwischen Kosten und Indikatoren sind für den objektorientierten Entwurf allerdings empirisch kaum belegt. Manche sind immerhin für den strukturierten Entwurf belegt, von dem einige Kriterien wie z. B. Kopplung und Zusammenhalt übernommen wurden.

Diskussion

Der transzendenten und der benutzerbezogenen Definition fehlt es an Eindeutigkeit und Objektivität, daher können sie in diesem Zusammenhang ausgeschlossen werden. Die produktbezogene Sicht definiert eine allgemeine, die herstellungsbezogene eine spezifische Qualitätssicht. Sowohl die allgemeine als auch die spezifische Definition haben ihre Vorteile.

Der Vorteil eines allgemeinen Qualitätsmodells ist, dass es in allen Fällen ohne Anpassung verwendbar ist. Allerdings müsste ein solches Modell, um allgemein gültig zu sein, entweder sämtliche Qualitätsaspekte abdecken, die jemals relevant sein könnten (der ganze Bereich in Abbildung 7-7), oder sich auf diejenigen Aspekte konzentrieren, die allen Entwürfen gemeinsam sind (der dunkle Schnittbereich in Abbildung 7-7). Wie man sieht, umfasst der gesamte Bereich viele Teile, die für einzelne Produkte ohne Belang sind, während der gemeinsame Bereich viele Teile, die für die Produkte relevant sind, nicht enthält. Also muss für ein brauchbares allgemeines Modell ein Kompromiss zwischen den beiden Extremen gesucht werden.

Der Vorteil eines spezifischen Modells liegt darin, dass es an das Produkt angepasst ist, also alle relevanten Aspekte abdeckt und keine irrelevanten Aspekte enthält. Allerdings ist zunächst einmal für jedes Produkt ein solches Modell zu erstellen. Daher sollte es zumindest eine Art Schablone geben, aus der sich ein produktspezifisches Modell ableiten lässt, oder ein Vorgehensmodell, mit dem mit möglichst geringem Aufwand ein spezifisches Modell erzeugt werden kann.

Am sinnvollsten ist die Kombination der beiden Ansätze: Ein Vorgabemodell, das ein allgemeines Modell ist, wird durch Adaptionsschritte nach einem definierten Vorge-

4. Beim Vergleich der Gesamtkosten von Entwurfsalternativen sollte eine unterschiedliche Nutzungsdauer berücksichtigt werden. Das kann geschehen, indem die Kosten für ein Nachfolgesystem in die Kostenbetrachtung mit einbezogen werden oder indem die Kosten auf die Nutzungsdauer umgelegt werden.

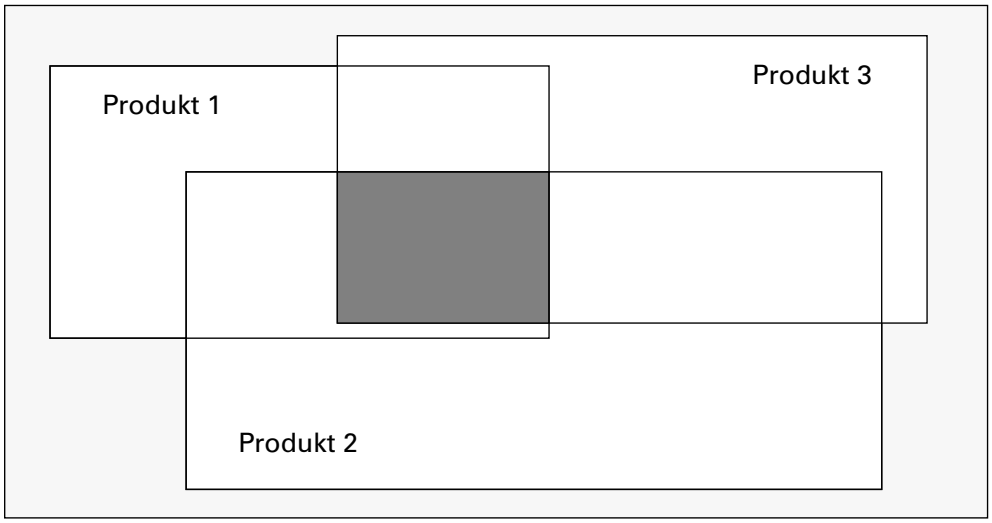

Abbildung 7-7: Gültigkeitsbereiche für allgemeine Modelle

hen an das konkrete Produkt angepasst und so in ein spezifisches Modell überführt. Die produktbezogene Sicht kann dabei um eine kostenbezogene erweitert werden, indem diejenigen Aspekte besonders berücksichtigt werden, welche die Kosten stark beeinflussen.

7.3 Entwurfsregeln

Software design is hard, and we need all the help we can get.
(Bjarne Stroustrup)

Um die gewünschten Eigenschaften des Entwurfs in hohem Maße zu erreichen, wurden Unmengen von Ratschlägen publiziert: Methoden, Prinzipien, Heuristiken, Entwurfsmuster und vieles andere mehr. In diesem Abschnitt sollen die Prinzipien und Heuristiken genauer betrachtet werden, da sie so etwas wie den Erfahrungsschatz des objektorientierten Entwurfs darstellen. Daher können aus ihnen Kriterien für einen guten Entwurf gewonnen werden.

7.3.1 Prinzipien

Neither SA nor SD as currently practiced have proved to be very good routes to actually deriving a sound OO design, but nearly all of the basic principles still apply: problem partitioning, component integrity (cohesion), independence (coupling), etc. All science and engineering builds on what has gone before.
(Constantine, 1991)

Balzert (1985a, S. 2) umreißt Prinzipien wie folgt: „Prinzipien sind Grundsätze, die man seinem Handeln zugrundelegt. Sie sind allgemeingültig, abstrakt, allgemeinster Art. Prinzipien bilden eine theoretische Grundlage. Sie werden aus der Erfahrung und Erkenntnis hergeleitet und durch sie bestätigt.“ Das Sammeln von Prinzipien des Software Engineering begann schon früh (z. B. Ross et al., 1975). Balzert (1985a, 1985b), Davis (1995) und Buschmann et al. (1996, Kap. 6.3) haben weitere Sammlungen von Prinzipien veröffentlicht.

Hier werden einige Prinzipien für den objektorientierten Entwurf, von denen viele aus dem strukturierten Entwurf übernommen wurden, kurz vorgestellt. Dabei werden strategische (Tabelle 7-2) und taktische Prinzipien (Tabelle 7-3) unterschieden. Strategische Entwurfsprinzipien sind fundamentaler Art, sie geben eine Art Entwurfstheorie vor. Die taktischen Prinzipien sind technischer Art, sie schlagen die Anwendung bestimmter Techniken vor. Eine ausführliche Erläuterung der Prinzipien findet sich bei Reißing (2002).

Prinzip	Beschreibung
Führe den Entwurf auf die Anforderungen zurück	Jede Entwurfsentscheidung muss auf ihre zugehörigen Anforderungen zurückgeführt werden können (und umgekehrt). Diese Eigenschaft wird als Verfolgbarkeit (traceability) bezeichnet.
Erfinde nicht das Rad neu	Falls immer es möglich ist, sollte für eine vorgesehene Komponente des Entwurfs eine bereits vorhandene Komponente wiederverwendet werden, statt eine neue zu schaffen.
Kenne den Anwendungsbereich	Der Entwerfer sollte die Begriffswelt und typische Abläufe der Anwendung kennen, aber auch mit ihrer Arbeitsumgebung vertraut sein. Dazu gehört die technische Umgebung genauso wie die soziale Umgebung, die aus den Benutzern und den geltenden Gesetzen und Standards besteht. Nur dann können die optimale Architektur und geeignete Algorithmen gewählt werden.
Sorge für intellektuelle Kontrolle (Witt et al., 1994)	Sowohl die Entwickler als auch die Wartungsprogrammierer sollen in der Lage sein, den Entwurf vollständig zu verstehen. Dies wird durch hierarchische Strukturierung, Abstraktion und Einfachheit der einzelnen Komponenten erleichtert. Außerdem muss der Entwurf gut dokumentiert sein.
Minimiere den intellektuellen Abstand (Structural Correspondence, Jackson, 1975)	Der intellektuelle Abstand ist der Unterschied (z. B. in der Struktur) zwischen dem Problem und der Software-Lösung. Um einen geringen intellektuellen Abstand zu erreichen, sollten sich die relevanten Begriffe der Problemwelt (in der Regel die reale Welt) möglichst originalgetreu in der Lösung wiederfinden.
Stelle konzeptionelle Integrität her (Witt et al., 1994)	Konzeptionelle Integrität bedeutet, dass der gesamte Entwurf einem einheitlichen Stil folgen soll. Der Entwurf soll am Ende so aussehen, als sei er von einer einzigen Person geschaffen worden.
Verberge Realisierungsentscheidungen (Geheimnisprinzip, Parnas, 1972b)	Alle Realisierungsentscheidungen sollen in Module gekapselt und so vor dem Rest des Systems verborgen werden. Die Modulschnittstelle soll also über die innere Struktur möglichst wenig Auskunft geben. Gerade bei Entscheidungen, die sich wahrscheinlich ändern, ist das hilfreich.
Minimiere die Komplexität	Komplexität erschwert das Verständnis und damit die intellektuelle Kontrolle. Daher sollte sie so gering wie möglich sein.
Vermeide Redundanz (Beck, 1996; Hunt, Thomas, 1999)	Jede Form von Redundanz stellt ein Risiko dar, da es bei Änderungen leicht zu Inkonsistenzen kommt. Stattdessen sollte eine Funktion an genau einer Stelle realisiert werden.

Tabelle 7-2: Strategische Prinzipien

Prinzip	Beschreibung
Benutze Abstraktion	Abstraktion ist ein für den Menschen natürliches Verfahren, mit komplexen Sachverhalten umzugehen. Bei der Abstraktion werden irrelevante Details ausgeblendet. Um einen Sachverhalt zu verstehen, ist es dann nicht mehr notwendig, alle Details auf einmal im Gedächtnis zu haben. Es genügt, die an der aktuellen Aufgabenstellung beteiligten Abstraktionen zu beherrschen.
Teile und herrsche	Ein Problem wird in kleinere, möglichst unabhängige Teilprobleme zerlegt, die gelöst werden. Aus den Einzellösungen wird dann die Gesamtlösung zusammengesetzt. Das Verfahren lässt sich rekursiv anwenden, bis man zu einfach lösbaren Teilproblemen gelangt. Der Vorteil dieses Vorgehens besteht darin, dass zu einem Zeitpunkt nur ein Problem relativ geringer Komplexität gelöst werden muss.
Strukturiere hierarchisch (Simon, 1962; Parnas, 1972b)	Der Mensch kann nur dann mit komplexen Systemen umgehen, wenn sie hierarchisch sind. Daher sind hierarchische Strukturen im Entwurf vorteilhaft.
Modularisiere	Das System wird – im Geiste des Teile-und-herrsche-Prinzips – in sinnvolle Subsysteme und Module zerlegt. Das Modul dient dabei als Behälter für Funktionen oder Zuständigkeiten des Systems.
Trenne die Zuständigkeiten (separation of concerns)	Das System wird anhand von Zuständigkeiten (häufig auch als Verantwortlichkeiten bezeichnet) in Komponenten aufgeteilt. Komponenten, die an der gleichen Aufgabe beteiligt sind, werden gruppiert und von denen abgegrenzt, die für andere Aufgaben zuständig sind.
Trenne Verhalten und Implementierung (separation of policy and implementation; Rumbaugh et al., 1993)	Eine Komponente (oder eine Methode) soll entweder für das Verhalten oder die Implementierung zuständig sein, nicht für beides. Eine Komponente für das Verhalten trifft Entscheidungen anhand des Kontextes, interpretiert Ergebnisse und koordiniert andere Komponenten. Eine Komponente für die Implementierung dagegen führt einen Algorithmus auf vorliegenden Daten aus. Komponenten für die Implementierung sind in der Regel stabil, während sich die Komponenten für das Verhalten oft ändern, daher gibt es Sinn, sie zu trennen.
Kapsele Zusammengehöriges	Zusammengehörige Bestandteile einer Abstraktion werden zu einem Ganzen zusammengefasst und von anderen abgegrenzt. Die Implementierung wird hinter einer Schnittstelle verborgen. Bei der objektorientierten Sichtweise wird die Kapselung durch die Definition von Klassen erreicht.
Trenne Schnittstelle und Implementierung	Eine Komponente soll aus einer Schnittstelle und einer Implementierung bestehen. Die Schnittstelle soll nicht von der Implementierung abhängen, so dass der Verwender nicht durch Änderungen von Implementierungsdetails betroffen sein kann.
Vermeide Abhängigkeiten von Details (Martin, 1996c)	Komponenten sollten nicht von Details (i. d. R. Implementierungsdetails) abhängig sein, sondern von abstrakten Schnittstellen. In der Objektorientierung bedeutet das, dass die wesentlichen Abstraktionen in Form von Interfaces oder abstrakten Klassen formuliert sind.

Tabelle 7-3: Taktische Prinzipien (Abschnitt 1 von 2)

Prinzip	Beschreibung
Sorge für schmale Schnittstellen (Martin, 1996d)	In der Basisklasse sollen nur die für die eigentliche Abstraktion benötigten Methoden definiert sein. Andere Aspekte sollten durch separate Schnittstellen (z. B. durch Interfaces) definiert werden, die bei Bedarf von Subklassen zur Schnittstelle hinzugefügt werden können.
Sorge für lose Kopplung (Stevens et al., 1974)	Kopplung ist ein Maß für die Stärke der Verbindung (und damit der Abhängigkeit) von Komponenten untereinander. Angestrebt wird eine möglichst lose Kopplung von Komponenten. Damit steigt die Wahrscheinlichkeit, dass Änderungen sich nur lokal auf eine Komponente auswirken und dadurch nicht auf andere Komponenten ausstrahlen.
Sorge für hohen Zusammenhalt (Stevens et al., 1974)	Zusammenhalt (Kohäsion) ist ein Maß für die Stärke der Zusammengehörigkeit von Bestandteilen einer Komponente. Angestrebt wird ein möglichst hoher Zusammenhalt der Bestandteile. Damit steigt die Wahrscheinlichkeit, dass bei einer Änderung in der Regel nur eine Komponente betroffen ist, da alle Aspekte einer Abstraktion an einem Ort zusammengefasst sind.
Module sollen offen und geschlossen sein (Open-Closed-Principle, Meyer, 1997, S. 57; Martin, 1996a)	Geschlossen bedeutet: Das Modul kann gefahrlos verwendet werden, da seine Schnittstelle stabil ist, d. h. sich nicht mehr ändert. Offen dagegen bedeutet: Das Modul kann problemlos erweitert werden. Das scheint zunächst widersprüchlich zu sein, ist aber realisierbar: Eine abstrakte Oberklasse (oder ein Interface) realisiert eine stabile (geschlossene) Schnittstelle. Konkrete (offene) Unterklassen implementieren sie (und können sie bei Bedarf erweitern).
Sorge bei Redefinition für den Erhalt der Semantik (Liskov Substitution Principle, Liskov, 1988; Martin, 1996b)	Jede Unterklasse U einer Klasse K muss für die durch K gegebene Schnittstelle dieselbe Semantik anbieten. Nur dann kann man bei der Verwendung der Schnittstelle einer Klasse sichergehen, dass sich das Programm gleich verhält, wenn an Stelle einer Instanz von K eine Instanz von U verwendet wird.
Minimiere die Anzahl der Objekte, mit denen ein Objekt interagiert (Law of Demeter, Lieberherr et al., 1988, 1989)	In den Methoden einer Klasse dürfen nur Methoden der Klasse selbst, der Argumentobjekte oder der Attributwerte der Klasse aufrufen werden. Dahinter steht die Überlegung, dass sich die meisten Abhängigkeiten zwischen Klassen durch Methodenaufrufe manifestieren, eine Beschränkung also zu weniger Abhängigkeiten führt.

Tabelle 7-3: Taktische Prinzipien (Abschnitt 2 von 2)

Leider gibt es für die meisten dieser Prinzipien wenig empirische Untersuchungen über ihre Wirksamkeit und ihre Relevanz. Sie stellen eher eine Art Tradition im Software Engineering dar, die durch überwiegend positive Erfahrungen aufrechterhalten wird. Obwohl der empirische Nachweis der Brauchbarkeit fehlt, werden in vielen wissenschaftlichen Arbeiten diese Prinzipien herangezogen, um aus ihnen Entwurfsziele zu extrahieren. Aus diesen werden dann Qualitätsattribute abgeleitet, die zur Entwurfsbewertung verwendet werden.

7.3.2 Heuristiken

Useful suggestions about quality, when they are brought to our attention, usually strike us at once as familiar and revelatory. We see them as sensible, reflecting what we have felt but perhaps not expressed.
(Dromey, 1996, S. 43)

Eine Heuristik ist eine Daumenregel, die meistens funktioniert, aber nicht immer optimale Ergebnisse liefert. Daher ist eine Heuristik im Gegensatz zum Prinzip nicht allgemein gültig, sondern es muss anhand des Kontextes entschieden werden, ob ihr Einsatz an einer bestimmten Stelle im Entwurf sinnvoll ist.

Eine der ersten Sammlungen von Heuristiken stammt von Korson und McGregor (1990, S. 54). Riel (1996) veröffentlichte die bisher umfangreichste Sammlung, aber auch Booch et al. (1998) führen in ihrem UML-Handbuch viele Heuristiken auf. Weitere Heuristiken finden sich z. B. bei Firesmith (1995), Lakos (1996) und Johnson und Foote (1988).

Die Heuristiken lassen sich in fünf Bereiche einteilen: Heuristiken für Klassen, Interfaces, Pakete, Vererbungsbeziehungen und sonstige Beziehungen. Tabelle 7-4 zeigt einige Beispiele für jede Kategorie (siehe auch Reißing, 2002).

Heuristiken für Klassen
Die Klasse soll eine abgegrenzte Abstraktion eines Begriffs aus dem Problem- oder Lösungsbereich sein (Booch et al., 1998).
Die Klasse soll eine kleine, wohldefinierte Menge von Verantwortlichkeiten enthalten und diese alle gut ausführen (Booch et al., 1998).
Die Attribute und Operationen der Klassen sollen (wirklich) notwendig sein, um die Verantwortlichkeiten der Klasse zu realisieren (Booch et al., 1998).
Enthält eine Klasse zu viele Verantwortlichkeiten, sollen diese auf neue Klassen verteilt werden (Booch et al., 1998).
Enthält eine Klasse zu wenig Verantwortlichkeiten, soll sie mit anderen Klassen zusammengefasst werden (Booch et al., 1998).
Die Klasse soll eindeutig zwischen der Schnittstelle und der Implementierung der Abstraktion trennen (Booch et al., 1998).
Die Klasse soll verständlich und einfach sein, aber trotzdem erweiterbar und anpassbar (Booch et al., 1998).
Die öffentliche Schnittstelle einer Klasse soll nur aus Operationen bestehen (Korson, McGregor, 1990).
Jede Methode einer Klasse soll Attribute der Klasse verwenden (lesend oder schreibend) (Korson, McGregor, 1990).

Tabelle 7-4: Heuristiken (Abschnitt 1 von 2)

Heuristiken für Interfaces
Das Interface soll einfach, aber trotzdem vollständig sein (Booch et al., 1998).
Das Interface soll alle nötigen (aber nicht mehr) Operationen für einen einzigen Dienst zur Verfügung stellen (Booch et al., 1998).
Das Interface soll verständlich sein, d.h. es stellt genügend Information zur Verwendung und zur Implementierung zur Verfügung (Booch et al., 1998).
Das Interface soll zugänglich sein, d.h. sein Verwender kann die Haupteigenschaften verstehen, ohne durch eine Vielzahl von Operationen überwältigt zu werden (Booch et al., 1998).
Heuristiken für Pakete
Das Paket soll einen hohen Zusammenhalt haben, d. h. eine Menge zusammengehöriger Elemente enthalten (Booch et al., 1998).
Das Paket soll mit anderen Paketen lose gekoppelt, d. h. nur die Elemente, die andere Pakete wirklich sehen müssen, exportieren, und nur wirklich benötigte Elemente aus anderen Paketen importieren (Booch et al., 1998).
Pakete sollen nicht zu tief verschachtelt sein (Booch et al., 1998).
Die Anzahl der in einem Paket enthaltenen Elemente soll (im Vergleich zu den anderen Paketen im System) weder zu groß noch zu klein sein (Booch et al., 1998).
Zwischen Paketen soll es keine zyklischen Abhängigkeiten geben. Treten diese auf, soll eines der Pakete zerschlagen werden, um den Zyklus aufzulösen (Martin, 1996e).
Heuristiken für Vererbungsbeziehungen
Vererbungshierarchien sollen balanciert sein: nicht tiefer als (etwa) fünf Stufen und nicht zu breit. Um die Breite zu reduzieren, können zur Gruppierung abstrakte Zwischenklassen in die Vererbungshierarchie eingefügt werden (Booch et al., 1998).
Vererbung soll nur verwendet werden, um eine Spezialisierungshierarchie zu modellieren (Riel, 1996).
Basisklassen sollen abstrakt sein (Riel, 1996).
Mehrfachvererbung soll zunächst als Entwurfsfehler angesehen werden, bis das Gegenteil bewiesen ist (Riel, 1996).
Heuristiken für sonstige Beziehungen
Benutzungsbeziehungen sollen nur verwendet werden, wenn es sich nicht um eine strukturelle Beziehung handelt. Ansonsten soll mit Assoziationen gearbeitet werden (Booch et al., 1998).
Bei Aggregation soll eine Klasse wissen, was sie enthält, aber sie soll nie wissen, wer sie enthält (Riel, 1996).
Eine Klasse soll von möglichst wenigen anderen Klassen abhängen (Korson, McGregor, 1990)

Tabelle 7-4: Heuristiken (Abschnitt 2 von 2)

7.4 Beispiele für OOD-Qualitätsmodelle

Die Fragestellung „Was macht einen guten objektorientierten Entwurf aus?“ hat schon viele Autoren beschäftigt. Aus der Literatur soll hier eine kleine Zusammenstellung der Meinung verschiedener Autoren gegeben werden, die sich mit dieser Frage beschäftigt haben. Alle Autoren geben Kriterien bzw. Fragestellungen an, mit denen die Qualität eines Entwurfs bewertet werden soll. Viele der oben bereits beschriebenen Überlegungen haben Eingang in die Qualitätsmodelle gefunden.

7.4.1 Booch

Booch (1994, S. 136ff.) gibt fünf wichtige Eigenschaften einer Abstraktion (d. h. einer Klasse) an. Leider fehlen Messvorschriften für diese Eigenschaften, obwohl er sie als Metriken bezeichnet.

Kopplung (coupling). Geringe Kopplung wird, wie beim strukturierten Entwurf, auch beim objektorientierten Entwurf angestrebt; die Module sind hier die Klassen. Es gibt zwei verschiedene Arten der Kopplung: durch Verwendung und durch Vererbung. Während die Kopplung durch Verwendung so gering wie möglich sein sollte, ist Vererbung an sich erwünscht. Die dadurch erzeugte Kopplung wird in Kauf genommen.

Zusammenhalt (cohesion). Hoher Zusammenhalt ist ebenfalls erwünscht; es geht vor allem um den Zusammenhalt der Bestandteile (Attribute und Methoden) von Klassen.

Angemessenheit (sufficiency). Eine Klasse ist angemessen, wenn sie genügend Eigenschaften einer Abstraktion umfasst, so dass eine sinnvolle und effiziente Verwendung der Komponente möglich ist. In der Regel wird dabei eine minimale Schnittstelle angestrebt, die dennoch sicherstellt, dass die Klasse im System verwendbar ist.

Vollständigkeit (completeness). Eine Klasse ist vollständig, wenn sie alle relevanten Eigenschaften einer Abstraktion umfasst. Das bedeutet, dass eine allgemeine Schnittstelle erwünscht ist, welche die Wiederverwendung der Klasse in anderen Systemen erlaubt. Eine vollständige Schnittstelle ist angemessen, aber nicht minimal. Beim Klassenentwurf muss daher häufig ein Kompromiss zwischen Angemessenheit und Vollständigkeit eingegangen werden – zumal Booch davor warnt, Vollständigkeit zu übertreiben.

Primitivität (primitiveness). Eine Methode einer Klasse ist primitiv, wenn sie nur mit direktem Zugriff auf die interne Repräsentation (die Attribute) effizient implementiert werden kann. Die Schnittstelle einer Klasse sollte möglichst nur aus primitiven Methoden bestehen. Eine Methode, die sich ausschließlich durch Aufruf primitiver Methoden realisieren lässt, sollte nur dann in die Klassenschnittstelle aufgenommen werden, wenn eine solche Implementierung nicht effizient genug ist. Ansonsten sollte sie in eine Service-Klasse ausgelagert werden. Dies trägt dazu bei, die Klassenschnittstelle schlank zu halten.

7.4.2 Coad und Yourdon

Coad und Yourdon (1991, Kap. 8) geben einige Kriterien an, die bei einer Entwurfsbewertung angewendet werden sollen. Ihrer Meinung nach werden Entwurfskriterien benötigt, um das Entwickeln eines schlechten Entwurfs zu verhindern. Guter Entwurf bedeutet für sie eher, schlechte Eigenschaften zu vermeiden, als aus dem Stand einen perfekten Entwurf abzuliefern. Letzteres sei nämlich völlig unrealistisch.

Kopplung (coupling). Wie Booch (1994) unterscheiden Coad und Yourdon zwei Arten der Kopplung: durch Interaktion (entspricht der Kopplung durch Verwendung) und durch Vererbung. Um eine geringe Interaktionskopplung zu erreichen, wird vorgeschlagen, die Anzahl der versendeten und empfangenen Nachrichten zu

minimieren und die Anzahl der Parameter einer Nachricht auf drei zu limitieren. Die Delegation von Nachrichten an andere Objekte wird als Ursache für unnötige Interaktionskopplung angesehen.

Die Vererbungskopplung soll hingegen hoch sein. Die Vererbung spiegelt vornehmlich Generalisierungs-/Spezialisierungsbeziehungen. Das Überschreiben von geerbten Attributen oder ein Erben ohne Erweiterung sind Indikatoren für eine geringe Vererbungskopplung, da keine wirkliche Spezialisierung vorliegt.

Zusammenhalt (cohesion): Drei Ebenen des Zusammenhalts werden unterschieden: Methode, Klasse und Vererbungshierarchie. Methoden sollten nur genau eine Funktion haben. Klassen sollten nur Attribute und Methoden haben, die aufgrund der Verantwortlichkeiten der Klasse notwendig sind; und diese Attribute und Methoden sollen zusammenhängen. In der Vererbungshierarchie schließlich sollen nur „echte“ Spezialisierungen vorkommen.

Wiederverwendung (reuse). Die Wiederverwendung vorhandener Artefakte (z. B. Klassen, Komponenten, Bibliotheken, Entwürfe) kann die Kosten senken und die Qualität erhöhen.

Klarheit (clarity). Der Entwurf muss verständlich sein, um seine Verwendung und Wiederverwendung zu fördern. Als Maßnahmen werden die Verwendung eines einheitlichen Vokabulars, eines einheitlichen Stils (insbesondere bei Schnittstellen) und die klare Zuordnung von Verantwortlichkeiten zu Klassen genannt.

Einfachheit (simplicity). Methoden, Objekte und Klassen sollen so einfach wie möglich sein. Methoden sollten nur wenige Anweisungen umfassen. Objekte sollten so wenig wie möglich andere Objekte kennen müssen, um ihren Zweck zu erfüllen. Klassen sollten möglichst wenig Attribute und eine kleine Schnittstelle haben sowie klare Verantwortlichkeiten besitzen.

Größe (system size). Die Größe des Gesamtsystems sollte möglichst klein sein, um die Handhabbarkeit zu verbessern. Je größer das System wird, desto größer wird das Risiko, dass das Entwurfsteam den Entwurf nicht mit einem wohlstrukturierten, eleganten Ergebnis abschließen kann. Coad und Yourdon geben dafür eine Obergrenze von etwa hundert Klassen an.

Eleganz (elegance). Die Autoren geben zu, dass dieses Kriterium von allen am schlechtesten messbar ist. Weil der Begriff der Eleganz in der Praxis immer wieder auftauche, müsse er aber von Bedeutung sein. Sie geben zwei Beispiele für Eleganz an: wiederkehrende Muster im Entwurf und Ähnlichkeit der Entwurfsstruktur zur Struktur der Realität.

Zur Entwurfsbewertung schlagen Coad und Yourdon Reviews vor. Zum einen sollen sich diese an Szenarien orientieren, die mit den Klassen durchgespielt werden. Außerdem sollten kritische Erfolgsfaktoren wie Wiederverwendbarkeit, Effizienz und Implementierbarkeit geprüft werden.

7.4.3 Cockburn

Cockburn (1998) geht von der folgenden Annahme an: „Discussing the quality of an OO design is discussing the futures it naturally supports“. Es geht ihm also vor allem um Faktoren wie Änderbarkeit und Erweiterbarkeit. Cockburn gibt ein Bewertungs-

schema für objektorientierte Entwürfe an, das aus den folgenden sechs Kriterien besteht.

Verbundenheit der Daten (data connectedness). Reicht das von einer Klasse ausgehende Beziehungsnetzwerk aus, um ihr die Ausführung der von ihr verlangten Dienste zu erlauben? Oder anders ausgedrückt: Verfügt die Klasse über die nötigen „Kontakte“, um ihre Aufgabe zu erfüllen?
Hier könnte m. E. noch zusätzlich geprüft werden, ob es überflüssige Kontakte gibt, die für die Erfüllung der Aufgabe nicht notwendig sind.

Abstraktion (abstraction). Entspricht der Name der Klasse der durch sie dargestellten Abstraktion? Hat der Name eine entsprechende Bedeutung in der Sprache der Experten des Anwendungsgebiets? Dieser Test ist, wie Cockburn zugibt, subjektiv, aber seiner Ansicht nach dennoch effektiv.

Verteilung der Verantwortlichkeiten (responsibility alignment). Passen der Name der Klasse sowie die Attribute und Methoden zu ihrer Hauptverantwortlichkeit? Dies ist eine Frage, die mit zunehmender Evolution des Entwurfs an Bedeutung gewinnt, da Klassen dazu neigen, sich im Laufe der Zeit immer weiter aufzublähen (d. h. um Funktionalität erweitert zu werden), wobei sie sich häufig von der ursprünglichen Intention des Entwerfers entfernen.

Datenvariationen (data variations). Kann das System mit allen Varianten von Daten zurechtkommen, mit denen es im Laufe seiner Ausführung konfrontiert werden kann? Leider wird dieser Punkt nicht genauer ausgeführt. Man kann die Aussage aber so interpretieren, dass sichergestellt sein sollte, dass das System sowohl von den anfallenden Mengen als auch von der Strukturierung der Daten auf alle möglichen Fälle vorbereitet sein soll.

Evolution (evolution). Welche Änderungen in den Abläufen des Anwendungsbereichs, der Technologie, der angebotenen Dienste etc. sind wahrscheinlich, und wie lassen sich die daraus resultierenden neuen Anforderungen und Änderungen im Entwurf umsetzen? Wie viele Entwurfskomponenten müssen dazu geändert werden?

Kommunikationsmuster (communication patterns). Entstehen außergewöhnliche Kommunikationsmuster zwischen Objekten während der Laufzeit des Systems? Als Beispiel für verdächtige Muster nennt Cockburn Zyklen. Andere Muster werden nicht genannt, hier ist wohl vor allem die Erfahrung des Prüfers gefragt.

Erfahrene Entwerfer richten nach Cockburn ihr Augenmerk vor allem auf die Prüfung der Abstraktionen und der Verteilung der Verantwortlichkeiten. Cockburn äußert die Vermutung, dass – sofern über die zukünftigen Veränderungen nichts Genaues bekannt ist – das Bestehen der beiden Prüfungen ein Indikator für die Robustheit des Entwurfs ist.

7.4.4 Kafura

Kafura (1998, S.389ff.) gibt in einer Einführung in C++ eine Art Checkliste zur Evaluation des Entwurfs einer einzelnen Klasse an. Die Checkliste umfasst fünf verschiedene Kriterien, jeweils mit einer Reihe von Unterkriterien:

Abstraktion (abstraction). Stellt die Klasse eine brauchbare Abstraktion dar?

- Identität: Man kann einen einfachen, einleuchtenden Namen für die Klasse finden. Denselben Test kann man auch auf Methoden anwenden.
- Klarheit: Man kann den Zweck der Klasse vollständig in zwei kurzen, präzisen Sätzen zusammenfassen.
- Einheitlichkeit: Alle Methoden der Klasse arbeiten auf demselben Abstraktionsniveau.

Verantwortlichkeiten (responsibilities). Hat die Klasse eine sinnvolle Menge von Verantwortlichkeiten?

- Es ist klar, welche Verantwortlichkeiten eine Klasse hat und welche nicht.
- Die Verantwortlichkeiten sind auf diejenigen begrenzt, die durch die zugrunde liegende Abstraktion erforderlich sind.
- Die Verantwortlichkeiten sind zusammenhängend, d. h. sie gehören alle zu einer einzigen Abstraktion und geben als Ganzes Sinn.
- Die Verantwortlichkeiten sind vollständig, d. h. es sind alle durch die zugrunde liegende Abstraktion erforderlichen Verantwortlichkeiten vorhanden.

Schnittstelle (interface). Ist die Schnittstelle klar und einfach?

- Benennung: Die Methodennamen geben die beabsichtigte Wirkung auf Objekte der Klasse wieder. Zwei Methoden mit unterschiedlichen Absichten sollten auch unterschiedlich heißen.
- Symmetrie: zueinander komplementäre Methoden (z. B. Abfrage- und Setzen-Methode) sollen als solche erkennbar sein, z. B. durch die Benennung. Falls zu einer Methode erkennbar eine komplementäre Methode fehlt, sollte dafür ein Grund vorhanden sein.
- Flexibilität: Falls es sinnvoll ist, sollte ein Klasse mehrere Varianten zum Aufruf einer Methode zur Verfügung stellen. Dies kann durch Überladen (overloading) einer Methode mit unterschiedlichen Parameterlisten erreicht werden.
- Bequemlichkeit: Aus Gründen der Bequemlichkeit sollten möglichst viele Default-Parameter (Parameter mit Wertevorbelegung) angeboten werden.

Meiner Meinung nach sind die letzten beiden Punkte fraglich. Sie leiten sich wohl vor allem aus der C- und C++-Tradition her, so viel wie möglich mit so wenig wie möglich Zeichen zu sagen. Deshalb müssen auch Methodenaufrufe so kurz wie möglich sein. Allerdings bläht das Überladen die Klassenschnittstelle oft unnötig auf, vor allem, wenn es aus Bequemlichkeit geschieht. Default-Parameter erschweren es, später im Code herauszufinden, welche Methode aufgerufen wird, da beim Aufruf Parameter weggelassen werden können. Daher wirken sich beide Punkte in der Regel negativ auf die Verständlichkeit aus.

Verwendung (usage). Stimmt der Entwurf der Klasse mit der tatsächlichen Verwendung überein? Dazu werden die tatsächlich vorkommenden Verwendungen der Klasse untersucht und daraufhin überprüft, ob die Schnittstelle der Klasse für diese Verwendungen geeignet ist. Dabei kann man fehlende Methoden identifizieren – und auch falsche, d. h. ungeeignete, Verwendungen.

Implementierung (implementation). Gibt es eine vernünftige Implementierung für die Klasse? Falls die Implementierung sehr groß oder komplex ist, sollte überprüft werden, ob die Klasse zu viele Verantwortlichkeiten hat oder eine ungeeignete Abstraktion gewählt wurde. Bei Bedarf kann die Klasse in neue Klassen zerschlagen werden oder die Implementierung durch Hilfsklassen besser strukturiert werden.

7.4.5 Gillibrand und Liu

Gillibrand und Liu (1998) geben ein rudimentäres Qualitätsmodell für den objektorientierten Entwurf an, das nach dem FCM-Schema (vgl. Abschnitt 6.2.1) aufgebaut ist. Das Modell enthält die drei Faktoren Zuverlässigkeit, Komplexität und Wiederverwendbarkeit. Zu diesen Faktoren werden Kriterien angegeben, für die wiederum eine Reihe von Metriken angegeben wird. Abbildung 7-8 zeigt den Aufbau des Modells. Der Begriff Zuverlässigkeit wird hier in einem völlig anderen Sinne verwendet: Er bezeichnet die Korrektheit des Entwurfs in Bezug auf die Spezifikation. Für die anderen beiden Faktoren wird keine Definition angegeben, da wohl irgendeine übliche Definition angenommen wird. Übrigens bedeutet eine höhere Zuverlässigkeit oder Wiederverwendbarkeit eine höhere Qualität, während eine höhere Komplexität eine niedrigere Qualität bedeutet. Eine solche Gegenläufigkeit der Faktoren ist – schon aus psychologischen Gründen – keine gute Idee. Statt Komplexität wäre wohl Einfachheit die bessere Wahl gewesen.

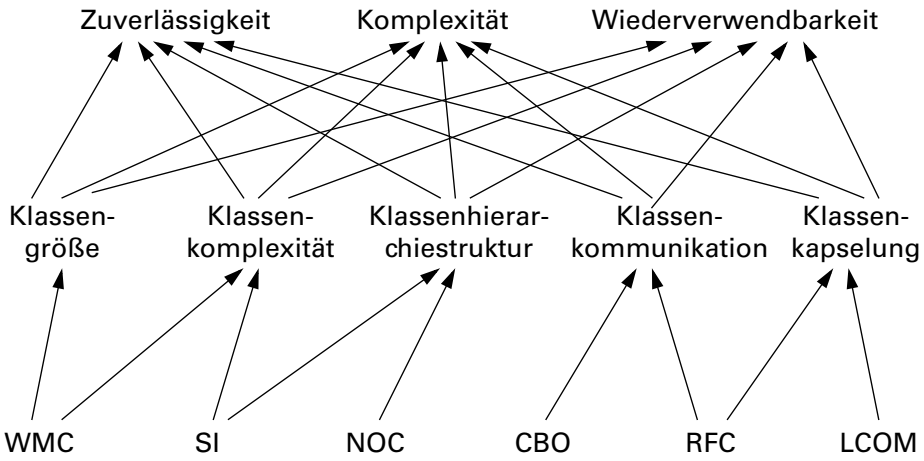

Abbildung 7-8: Qualitätsmodell von Gillibrand und Liu

Die Kriterien werden nicht näher erläutert, dafür die Metriken[5] ausführlich. Die Bewertung eines Kriteriums ergibt sich aus den Werten seiner Metriken, die gewichtet werden sollen. Konkrete Gewichte geben die Autoren aber nicht an. Eine Gewichtung soll ebenfalls erfolgen, um aus den Werten der Kriterien Werte für die Faktoren zu erhalten. Explizite Zusammenhänge zwischen Faktoren und Kriterien fehlen aber in der Beschreibung, Gewichte werden auch nicht genannt. Es gibt zwar eine Diskussion, welche Metriken Einfluss auf welche Faktoren haben – wie diese Einflüsse mit den Kriterien zusammenhängen, bleibt aber unklar. Es wird nur gesagt, dass alle fünf Kriterien für jeden Faktor relevant sind.

5. Die Metrik SI (specialization index) stammt von Lorenz und Kidd (1994), die übrigen Metriken von Chidamber und Kemerer (1994).

7.4.6 Erni

Erni (1996; s. a. Erni und Lewerentz, 1996) definiert ebenfalls ein Qualitätsmodell nach dem FCM-Schema. Da das Modell zur Bewertung von Rahmenwerken dienen soll, ist es auf den Bereich der Wiederverwendbarkeit beschränkt. Die Ebene der Kriterien wird in zwei Ebenen aufgeteilt: Entwurfsprinzipien und Entwurfsregeln. Auf diese Weise kann das Modell die zugrunde liegenden Prinzipien und Regeln reflektieren, so dass die Messungen auch gleich Verstöße gegen die Regeln und Prinzipien aufzeigen. Falls die Werte bestimmter Metriken nicht im gewünschten Bereich liegen, wird es dadurch für den Entwerfer einfacher, festzustellen, was das Problem ist und was geändert werden sollte. Abbildung 7-9 zeigt das resultierende Qualitätsmodell.

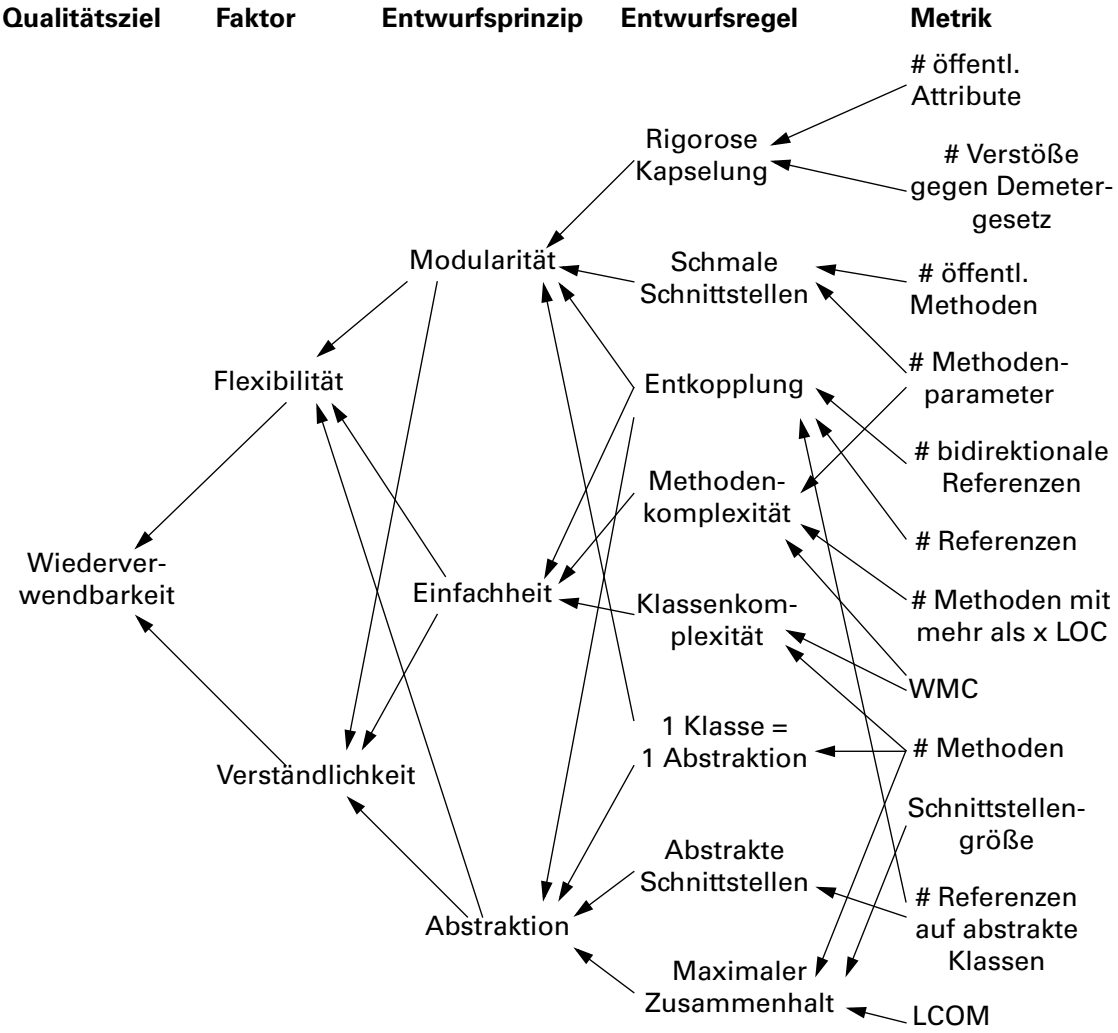

Abbildung 7-9: Qualitätsmodell von Erni (# = Anzahl)

Zu den Metriken werden Schwellenwerte angegeben, die aus der Literatur oder aus Erfahrung stammen. Für manche Metriken ist es allerdings sinnvoller, eine erwünschte Richtung (z. B. so groß wie möglich) anzugeben als einen oberen oder unteren Schwellenwert. Dann werden Schwellenwerte gewählt, die sich aus dem Mittel und der Standardabweichung der Messwerte berechnen (im Beispiel die Summe aus Mittel und Standardabweichung). Mit Hilfe der Schwellenwerte können die Messwerte in gut/schlecht oder akzeptabel/inakzeptabel unterteilt werden.

7.4.7 Bansiya und Davis

Bansiya und Davis (2002) haben ein hierarchisches Qualitätsmodell für den objektorientierten Entwurf namens QMOOD (Quality Model of Object-Oriented Design) entwickelt. Das Modell besteht aus drei Ebenen: Qualitätsattribute, Entwurfseigenschaften und Metriken (vgl. Abbildung 7-10). Basis der Bewertung ist ein in C++ dokumentierter Entwurf (in Form von Header-Dateien). Auf diesen Dateien werden mit Hilfe des Werkzeugs QMOOD++ (Bansiya, Davis, 1997) Metriken erhoben. Jede dieser Metriken ist genau einer Entwurfseigenschaft zugeordnet. Die Entwurfseigenschaften beeinflussen die Qualitätsattribute positiv oder negativ. Für die Stärke des Einflusses geben die Autoren Gewichte an, so dass sich aus den Metriken Qualitätskennzahlen für die Qualitätsattribute berechnen lassen. Die Gesamtbewertung ergibt sich durch Aufsummieren der Qualitätskennzahlen.

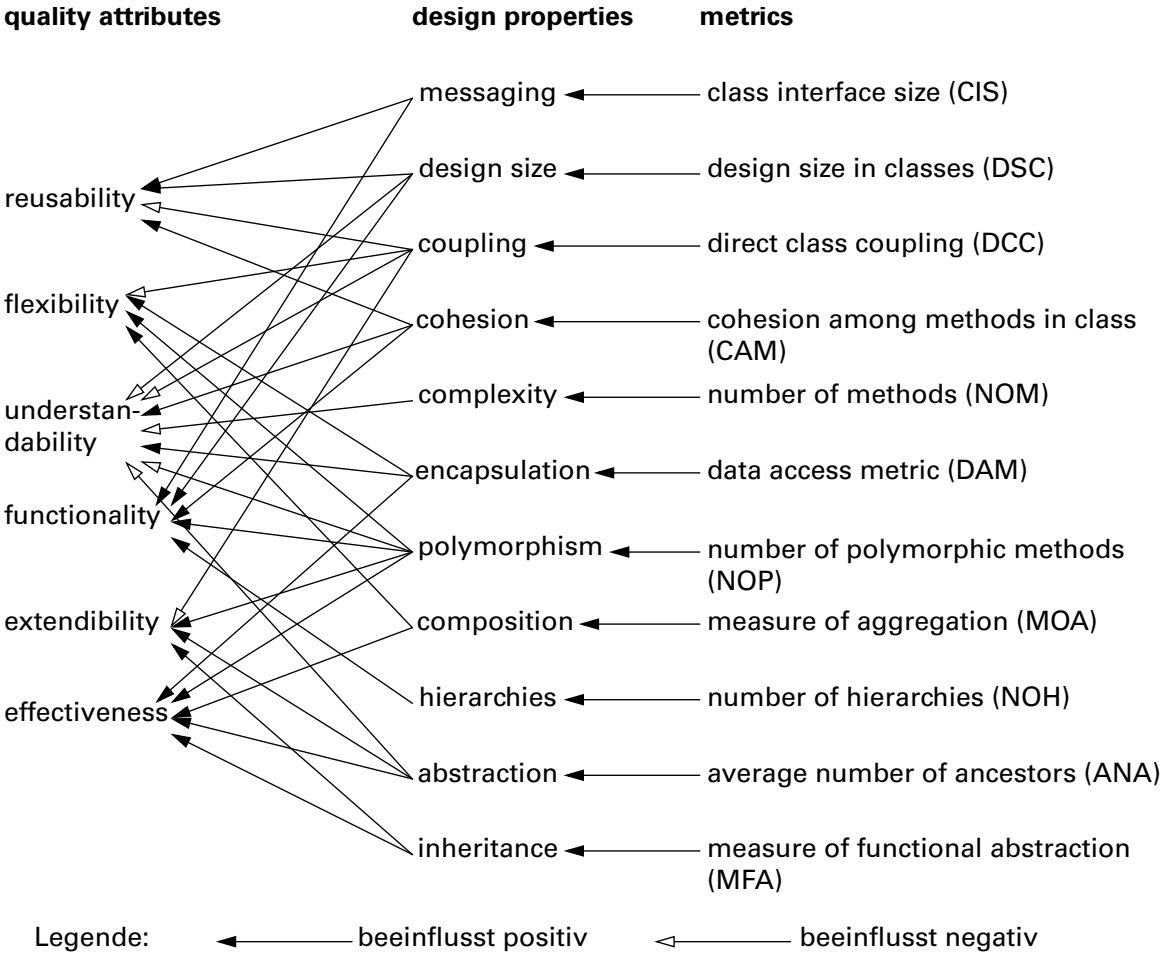

Abbildung 7-10: Qualitätsmodell von Bansiya und Davis

7.4.8 Kritische Bewertung

Wie die vorgestellten Beispiele zeigen, gibt es kein einheitliches Modell zur Bewertung objektorientierter Entwürfe. Es gibt zwar einige Kriterien, die öfter auftreten (z. B. Kopplung und Zusammenhalt) – diese unterscheiden sich aber jeweils in ihren Definitionen. In der Struktur der Modelle ist keine klare Trennung der Detaillierungsebenen (Methode, Klasse etc.) erkennbar. Außerdem werden die Systemebene kaum

und die Paketebene gar nicht betrachtet. Stattdessen ist die Sichtweise meist auf Klassen und ihre Bestandteile fokussiert. Die Beziehungen zwischen Klassen (Vererbung, Assoziation) werden selten explizit und oder gar detailliert betrachtet.

Nur die letzten drei Modelle enthalten eine Quantifizierung. Die Quantifizierung bei Gillibrand und Liu ist allerdings nur fragmentarisch dokumentiert und daher unbrauchbar. Außerdem deckt das Modell nur einen kleinen Teil der relevanten Qualitätsattribute ab. Bei Erni findet sich eine brauchbare Quantifizierung, die allerdings auf die unterste Ebene, die Metrikenebene, beschränkt ist. Nur bei Bansiya und Davis gibt es eine Quantifizierung für alle Ebenen.

Interessanterweise geben manche Autoren von Metrikensammlungen für den objektorientierten Entwurf (z. B. Whitmire, 1997) zwar Metriken und damit zusammenhängende Entwurfskriterien an, legen sich aber nicht auf den gewünschten Erfüllungsgrad der Kriterien oder gar auf Schwellenwerte für Metriken fest. Eine lobenswerte Ausnahme ist das Buch von Lorenz und Kidd (1994), das eine Sammlung von einfachen objektorientierten Metriken (auf Code-Ebene) enthält, für die Schwellenwerte aus der Erfahrung der Autoren angegeben sind.

7.5 Qualitätssicherung beim Entwurf

Quality must be built into designs, and cannot be inspected in or tested in. Nevertheless, any prudent development process verifies quality through inspection and testing. […] The very fact that designs face inspections motivates even the most conscientious designers to greater care, deeper simplicities, and more precision in their work.
(Mills, 1980, S. 418)

Bisher wurden Überlegungen zur Definition der Entwurfsqualität angestellt. In diesem Abschnitt wird gezeigt, mit welchen Maßnahmen die Entwurfsqualität überprüft und verbessert werden kann.

7.5.1 Organisatorische Maßnahmen

Entwicklungsprozess. Der Entwurf muss geeignet eingebettet sein und den erforderlichen Stellenwert haben, selbst wenn evolutionär entwickelt wird.

Standards und Richtlinien. Es werden Vorgaben sowohl für den Entwurf selbst (z. B. Namenskonventionen, Qualitätskriterien) als auch für die Entwurfsdokumentation (z. B. Aufbau, Umfang und Aktualität) gemacht.

Ausbildung der Entwerfer. Die Entwerfer erarbeiten sich durch Schulung und Projektarbeit das notwendige Wissen und die nötige Erfahrung für die Erstellung guter Entwürfe. Dabei ist der Einsatz von Experten als Berater oder Mentoren in der Entwurfsphase sinnvoll. Alternativ kann auch ein Architekturteam gebildet werden, dem ein erfahrener Experte als Chefarchitekt vorsteht.

Etablierung einer Wiederverwendungskultur. Geeignete Rahmenwerke, Bibliotheken und Muster sind bekannt und werden verwendet. Darüber hinaus kann auch durch Produktlinien (product lines) und den Aufbau eigener Bausteinbibliotheken ein Beitrag zur Wiederverwendung gemacht werden.

7.5.2 Konstruktive Maßnahmen

Object-oriented programming has shown that one way to attack complexity is to organize messes into smaller messes, and repeat the process.
(Meyer, 2001, S. 30)

Die Vorgaben aus den organisatorischen Maßnahmen schlagen sich in den konstruktiven Maßnahmen nieder.

Wiederverwendung. Man greift auf bewährte Entwürfe in Form von Architekturmustern, Entwurfsmuster, Rahmenwerken und fertigen Komponenten, z. B. aus Bibliotheken, zurück. Code-Generatoren können ebenfalls zur Architektur-Wiederverwendung eingesetzt werden.

Entwurfsregeln. Sie geben Hinweise, auf welche Eigenschaften während des Entwurfs zu achten ist, z. B. Modularisierung unter Beachtung von Kopplung und Zusammenhalt.

Notation. Es wird eine geeignete Notation verwendet (vgl. auch Abschnitt 8.3.6).

Werkzeuge. Es werden geeignete Werkzeuge eingesetzt, die den Entwerfer unterstützen, indem z. B. Konsistenz gewährleistet wird.

7.5.3 Analytische Maßnahmen

Review. Sobald der erste Entwurf vorhanden ist, kann er einem Review unterzogen werden; in der Regel wird es sich um eine Inspektion handeln. Bass et al. (1998) und Shull et al. (1999) geben einige Hinweise für effektive Entwurfsinspektionen.

Bewertung. Die Bewertung ist hilfreich, um den Entwurf mit möglichen Alternativen zu vergleichen und die beste Möglichkeit auszuwählen. Eine Schwachstellenanalyse auf der Grundlage der Bewertung kann mögliche Probleme des Entwurfs aufzeigen. Dann können Refaktorisierungen und Transformationsmuster eingesetzt werden, um diese Schwachstellen zu bereinigen. Ansätze zur Entwurfsbewertung werden in Abschnitt 7.6 vorgestellt.

Prototyping. Eine mögliche, wenn auch in der Regel teurere Alternative zur Bewertung ist das sofortige Ausprobieren durch eine Implementierung. Diese Implementierung kann prototypischen Charakter haben (Coad, Yourdon, 1991), muss es aber nicht. Beispielsweise wird bei der Vorgehensweise des extremen Programmierens (Beck, 1999a und 1999b) empfohlen, inkrementell zu entwerfen und jedes Entwurfsinkrement sofort zu implementieren. Falls eine Überarbeitung notwendig werden sollte, soll auf Refaktorisierung (Fowler et al., 1999) zurückgegriffen werden.

7.6 Entwurfsbewertung

By their very nature software products are not readily evaluated. There are no visible characteristics to give any clue to the quality of the product, and the behaviour of software systems is often complex and non-continuous.
(Dick, Hunter, 1994)

Entwurfsbewertung, der Schwerpunkt dieser Arbeit, ist eine analytische Qualitätssicherungsmaßnahme. In diesem Abschnitt werden einige Alternativen von Bosch (2000) beschrieben, wie die Bewertung durchgeführt werden kann.

7.6.1 Bewertungsverfahren

Anforderungen

Der ISO/IEC Guide 25 (ISO/IEC, 1990) beschreibt die ideale Bewertung als

- wiederholbar, d. h. wenn dasselbe Produkt nach derselben Prozedur von derselben Stelle evaluiert wird, kommt dasselbe heraus,
- reproduzierbar, d. h. wenn dasselbe Produkt nach derselben Prozedur von einer anderen Stelle evaluiert wird, kommt dasselbe heraus,
- unparteiisch, d. h. die Bewertung ist frei von Vorurteilen, und
- objektiv, d. h. die Bewertung kommt möglichst ohne subjektive Urteile aus.

Ansätze

Bosch (2000, S. 34ff.) unterscheidet vier Ansätze zur Entwurfsbewertung:

1. auf der Basis von Szenarien,
2. durch Simulation,
3. durch mathematische Modellierung und
4. auf der Basis bisheriger Erfahrungen.

Die ersten beiden bewerten durch Ausführung, der dritte durch statische Analyse und der vierte durch Inspektion. Diese Ansätze werden im Folgenden vorgestellt.

Evaluation mit Szenarien. In diesem Ansatz, der schon von Bass et al. (1998) propagiert wurde, werden Entwurfseigenschaften mit Hilfe von Szenarien bewertet, in denen der Entwurf sich bewähren muss. Beispielsweise kann die Änderbarkeit bewertet werden, indem einige Szenarien für wahrscheinliche Änderungen an den Anforderungen erstellt werden und dann versucht wird, die Anforderungsänderungen im Entwurf umzusetzen. Die Änderbarkeit ist um so höher, je weniger Aufwand man dafür aufbringen muss. Mit dem Verfahren kann auch die Umsetzung der Funktionsanforderungen im Entwurf geprüft werden, indem die Anwendungsfälle aus der Spezifikation am Entwurf durchgespielt werden.

Der Bewertungsansatz mit Szenarien hat den Vorteil, dass er mit jeder Form der Entwurfsdokumentation zurecht kommt, da ein Mensch die Bewertung durchführt. Zugleich wird die Vollständigkeit des Entwurfs geprüft, da beim Durchspielen sehr viele Detailinformationen zu den Abläufen gebraucht werden, wodurch Lücken in der Beschreibung aufgedeckt werden.

Der Nachteil des Ansatzes ist es, dass er nicht automatisierbar ist, weshalb bei der Bewertung ein hoher Aufwand entsteht. Daher können praktisch nur ausgewählte Szenarien durchgespielt werden, die Entwurfsbewertung bleibt also mehr oder weniger stichprobenartig.

Evaluation durch Simulation. Liegt der ganze Entwurf in einer hinreichend formalen Beschreibung vor, kann das entworfene System auch von einer Maschine simuliert werden. Die Beschreibung wird sich allerdings auf einem eher abstrakten Niveau befinden, weil sonst der Aufwand für die Erstellung der Beschreibung zu hoch ist. Alternativ kann man sich auch auf Ausschnitte aus dem Entwurf beschränken, die potentiell kritisch sind. Ein Beispiel sind Machbarkeitsstudien in Form von explorativen Prototypen, die speziell für bestimmte Qualitätsaspekte erstellt werden, z. B. zur Ermittlung der Leistung eines Transaktionsmonitors in einem Datenbanksystem.

Der Vorteil des Simulationsansatzes ist es, dass handfestere Aussagen entstehen als beim Szenario-Ansatz, weil mehr und vor allem präzisere Information hineingesteckt werden muss. Außerdem kann die Simulation im günstigsten Fall vollautomatisch durchgeführt werden. Dem steht allerdings der gravierende Nachteil gegenüber, dass die Formalisierung des Entwurfs erzwungen wird, was einen hohen Aufwand bedeuten kann und Expertenwissen voraussetzt. Die Formalisierung wiederum hat allerdings den Vorteil, dass auf diese Weise Lücken und unpräzise Beschreibungen im Entwurf viel eher auffallen. Ein exploratives Prototyping gibt auch Hinweise zur Implementierbarkeit, da die für die Implementierung benötigte Technologie in Form von Werkzeugen, Middleware etc. quasi mitgeprüft wird.

Evaluation durch mathematische Modellierung. Wie die Simulation setzt die mathematische Modellierung auf die Formalisierung des Entwurfs (oder eines Ausschnitts des Entwurfs). Die mathematischen Modelle dienen aber mehr der statischen Analyse des Entwurfs. Ein bekanntes Beispiel für mathematische Modelle sind die Leistungsmodelle zur Abschätzung des Zeit- und Platzbedarfs eines Algorithmus und die Einteilung in Leistungsklassen mit der O()-Notation. Ein weiteres Beispiel ist die Analyse eines nebenläufigen oder parallelen Systems auf Verklemmungsfreiheit (ein Aspekt der Robustheit). Auch wenn Bosch darauf nicht eingeht, sind auch Entwurfsmetriken (oft vergleichsweise primitive) mathematische Modelle eines Entwurfs. Die Vor- und Nachteile dieses Ansatzes sind ähnlich wie bei der Simulation.

Evaluation aufgrund von Erfahrung. Dieser ganzheitliche Ansatz baut vor allem auf der Erfahrung von Entwurfsexperten auf. Diese haben schon viele gute und schlechte Erfahrungen gemacht, die sie – auch in Form von Intuition – in die Entwurfsbewertung einbringen können. Ergebnis einer solchen Bewertung sind vor allem Hinweise auf (mögliche) Mängel im Entwurf und daraus abgeleitete Verbesserungsvorschläge. Werden diese Ergebnisse genauer untersucht, lässt sich meist durch logische Beweisführung nachvollziehbar machen, warum die Hinweise auf Mängel tatsächlich gerechtfertigt sind. Alternativ können auch die drei anderen Bewertungsansätze verwendet werden, um das identifizierte potentielle Problem zu untersuchen.

Der Vorteil dieses Verfahrens besteht in der ganzheitlichen Betrachtungsweise und in der Nutzung der Erfahrung langjähriger Entwerfer. Letzteres setzt aber voraus, dass ein solcher Entwerfer verfügbar ist und seine Erfahrung etwas taugt. Fehlt der Experte, besteht die Gefahr, dass der Ansatz wenig effektiv ist. Das gilt auch dann, wenn der Entwerfer seinen Entwurf selbst prüft (wegen kognitiver Dissonanz). Die Expertenbewertung ist subjektiv und kaum nachvollziehbar, die Ergebnisse können allerdings nachträglich nachvollziehbar gemacht werden.

Die bereits vorstellten Prinzipien (Abschnitt 7.3.1) und Heuristiken (Abschnitt 7.3.2) des Entwurfs sind der Versuch, das Expertenwissen von seinem Träger unabhängig

und allen Entwerfern zugänglich zu machen (ebenso wie z. B. die Entwurfsmuster). Obwohl es zur Anwendung dieser Entwurfsregeln doch wieder einiger Erfahrung bedarf, liegt damit die Schwelle zur qualifizierten Entwurfseinschätzung insgesamt niedriger. Der menschliche Experte ist teilweise auch durch Checklisten oder Fragebögen ersetzbar. Diese enthalten bestimmte Prüfkriterien, die dazu geeignet sind, potentielle Probleme aufzudecken. Die Entscheidung, ob ein identifiziertes potentielles Problem ein echtes Problem ist, setzt aber häufig wieder Expertenwissen voraus.

Diskussion

Die Beschreibung der vier Verfahren macht deutlich, dass jedes seine besonderen Stärken und Schwächen hat. Folglich sind für jedes Qualitätsattribut bestimmte Bewertungsverfahren besser geeignet als andere. Das Bewertungsverfahren sollte also je nach Attribut variieren – zum Teil sind auch Kombinationen sinnvoll. Laut Bosch ist die Evaluation mit Szenarien vor allem für Aspekte der Entwicklungsqualität wie Wartbarkeit und Flexibilität geeignet, während Simulation und mathematische Modellierung für Aspekte der Betriebsqualität wie Robustheit und Leistung besser sind. Für subjektive Qualitätsattribute wie Verständlichkeit bleibt letzten Endes wohl eine Begutachtung durch den Menschen erforderlich, auch wenn es automatisch messbare Indikatoren dafür gibt. In dieser Arbeit wird daher ein Bewertungsverfahren verwendet, das eine Kombination aus Expertenschätzung und mathematischer Modellierung mit Entwurfsmetriken ist.

Da es bei der Entwurfsbewertung vor allem darum geht, zukünftige Kosten vorherzusagen, bleibt es zum Zeitpunkt der Bewertung ungewiss, ob die gemessene Qualität des Entwurfs zutreffend ist. Wie jede Vorhersage der Zukunft kann die Bewertung mehr oder weniger falsch sein. Durch geeignete Validierung des Bewertungsverfahrens lässt sich zwar die statistische Sicherheit des Eintreffens der Vorhersage erhöhen, absolute Sicherheit gibt es aber nicht.

7.6.2 Quantitative Bewertung

Die Bewertung ist grundsätzlich in qualitativer und quantitativer Form möglich. Die qualitative Bewertung erlaubt nur relative Aussagen in Bezug auf ein einzelnes Qualitätsattribut, z. B. dass eine Entwurfsalternative portabler ist als eine andere. Für einen Alternativenvergleich ist die qualitative Bewertung ausreichend, sofern nur ein Attribut betrachtet wird und es unwichtig ist, um wie viel besser die eine Alternative als die andere ist.

Wenn mehrere Qualitätsattribute in die Bewertung einfließen sollen, können nur bei einer quantitativen Bewertung fundierte relative (Entwurf A ist besser als B) und absolute Aussagen (Entwurf A ist gut, Entwurf B ist befriedigend) gemacht werden. Daher ist eine Quantifizierung des Qualitätsmodells anzustreben. Abwägungen zwischen verschiedenen Attributen können in Form von Gewichten dargestellt werden, mit denen die einzelnen Bewertungen zu einer Gesamtbewertung verrechnet werden.

Zur Quantifizierung werden Metriken eingesetzt. Die schwierige Frage ist, wie sich die Qualitätsattribute mittels Metriken messen lassen. Es müssen für jedes Attribut eine oder mehrere Metriken gefunden werden, die mit dem Attribut korreliert sind. Außerdem ist zu entscheiden, was genau gemessen wird, beispielsweise ob man geerbte Eigenschaften mitzählt oder nicht.

Kapitel 8

Das allgemeine Qualitätsmodell

Im diesem und im folgenden Kapitel wird ein allgemeines Qualitätsmodell für den objektorientierten Entwurf namens QOOD (Quality Model for Object-Oriented Design) vorgestellt. Der Name QOOD (auszusprechen wie „could“) hat absichtlich einen Anklang an das Wort „good“. Das Modell soll nämlich dazu geeignet sein, einen Entwurf zu bewerten, d. h. seine Güte festzustellen. Es kann eine Antwort auf die Frage liefern, ob ein gegebener Entwurf gut ist (absolute Aussage) bzw. ob er besser ist als ein anderer Entwurf (relative Aussage).

Dieses Kapitel führt die Qualitätsattribute (Faktoren und Kriterien) ein, die in QOOD betrachtet werden sollen. Außerdem werden die Beziehungen zwischen den Qualitätsattributen (z. B. Korrelation, Widerspruch) diskutiert.

8.1 Vorüberlegungen

8.1.1 Ideales Modell

QOOD hat nicht den Anspruch, die endgültige Antwort auf die Frage nach Entwurfsqualität zu geben. Diese Antwort wird es niemals geben – ebenso wenig wie die endgültige Antwort auf die Frage nach Schönheit. Trotzdem kann man die Eigenschaften eines idealen allgemeinen Qualitätsmodells angeben:

- eindeutig und widerspruchsfrei,
- deckt alle Sichten (z. B. der verschiedenen Interessengruppen) ab,
- vollständig in Hinsicht auf die Aspekte der einzelnen Sichten,
- vollständig quantifiziert, d. h. alle Aspekte sind messbar,
- unterstützt Bewertungsverfahren, welche die Kriterien des ISO/IEC Guide 25 (vgl. Abschnitt 7.6.1) erfüllen, und

- anpassbar, d. h. es können Aspekte ausgeblendet oder Bewertungsschwerpunkte gebildet werden.

Man kann sich leicht vorstellen, dass ein Modell mit diesen Eigenschaften sehr groß ist und die Verflechtungen innerhalb des Modells sehr kompliziert sind. Also ist die Erstellung eines solches Modells sehr aufwendig. Selbst wenn es verfügbar wäre, wäre der Umgang damit nicht einfach, weil es vermutlich eine nicht mehr handhabbare Komplexität hat.

8.1.2 Pragmatischer Ansatz

Aus diesem Grund ist der hier verfolgte Ansatz ein pragmatischer: Ziel ist nicht das ideale Modell, sondern ein Qualitätsmodell, mit dessen Hilfe der Stand der Praxis der Entwurfsbewertung verbessert werden kann. Card und Glass (1990, S. 20) empfehlen für die Praxis: „Focus on a few key (quality) characteristics rather than attempt to measure everything." Außerdem empfehlen sie: „rely on simple measures extractable from common design products." Üblicherweise vorliegende Entwurfsartefakte sind relativ vollständige Klassendiagramme sowie vereinzelte Sequenz- und Zustandsdiagramme. Eine Quantifizierung muss dies berücksichtigen, da man nur das messen kann, was auch vorhanden ist. Verwendet werden daher nur Metriken, die sich auf UML-Klassendiagrammen (gemäß dem Referenzmodell ODEM) erheben lassen.

8.1.3 Mögliche Qualitätsattribute

Dennoch ist es interessant, sich zunächst zu überlegen, was denn potentiell bei der Entwurfsbewertung berücksichtigt werden könnte, um dann aus dem Angebot auszuwählen. Die folgende Liste möglicher Qualitätsattribute wurde auf der Basis der Überlegungen des vorhergehenden Kapitels zusammengestellt.

- Qualität der Entwurfsdokumentation, z. B. Vollständigkeit, Verständlichkeit, Übersichtlichkeit und Nachvollziehbarkeit von Entwurfsentscheidungen
- Brauchbarkeit des resultierenden Systems, z. B. Abdeckung der Anforderungen in Hinblick auf Funktion und Qualität (z. B. Effizienz, Benutzerfreundlichkeit)
- Implementierbarkeit/Realisierbarkeit in Hinblick auf Technologie, Organisation und Wissen bzw. Erfahrung (Wissensträger ist vor allem das Personal)
- Kosten der Realisierung, z. B. für Bausteine (Soft- und Hardwarekomponenten), Werkzeuge, Schulung und Personaleinsatz (einschließlich externe Kräfte)
- Wartbarkeit des Entwurfs, seiner Dokumentation und des resultierenden Systems, vor allem Änderbarkeit und Erweiterbarkeit (im Hinblick auf zu erwartende Änderungen der Anforderungen und der Implementierungstechnologie)
- Wiederverwendung vorhandener Entwürfe und Komponenten
- potentielle Wiederverwendbarkeit und tatsächliche Wiederverwendung des Entwurfs und des resultierenden Systems in anderen Systemen
- weitgehende Übereinstimmung mit Entwurfsregeln (Prinzipien und Heuristiken)
- Einhaltung von Standards, z. B. Standardnotationen, Namenskonventionen, Dokumentenvorlagen und Technologievorgaben

8.1.4 Indikatoren

We cannot build high-level quality attributes like reliability or maintainability into software. What we can do is identify and build in a consistent, harmonious, and complete set of product properties (such as modules without side effects) that result in manifestations of reliability and maintainability. We must also link these tangible product properties to high-level quality attributes.
(Dromey, 1996, S. 33)

Die eigentlich interessanten Eigenschaften des entworfenen Systems (z. B. Effizienz oder Wartbarkeit) lassen sich erst aus der Realisierung heraus zuverlässig bewerten. In der Entwurfsphase ist man daher auf Indikatoren angewiesen, die sich aus dem Entwurf heraus erheben lassen (z. B. Kopplung). Deshalb konzentriert sich das Qualitätsmodell auf Aspekte, die sich auf der Basis von ODEM beurteilen lassen.

8.1.5 Schwerpunkt Wartbarkeit

I suspect that some programmers think that their program will be so good that it won't have to be changed. This is foolish. The only programs that don't get changed are those that are so bad that nobody wants to use them. Designing for change is designing for success.
(Parnas, 1994, S. 282)

Der Schwerpunkt des Qualitätsmodells liegt auf der Wartbarkeit, da die Wartung im Software-Lebenszyklus den größten Kostenfaktor darstellt. Mindestens die Hälfte des Gesamtaufwands fließt in die Wartung (Boehm, 1976; Lientz, Swanson, 1980). Daher kann eine Kostenreduktion am ehesten dadurch erreicht werden, dass man den Aufwand für diese Phase reduziert. Der Wartungsaufwand selbst zerfällt nach Sneed (1988) in vier Bereiche:

- Stabilisierung, d. h. Fehlerkorrektur,
- Optimierung, d. h. Verbesserungen der Effizienz und der Struktur,
- Anpassung, d. h. durch die Umwelt erzwungene Modifikationen (technische oder fachliche), und
- Erweiterung, d. h. Hinzunahme weiterer Funktionen.

Tabelle 8-1 zeigt die Verteilung des Aufwands auf die vier Bereiche nach Lientz, Swanson (1980) und Sneed (1988). Die Verteilung ist bei beiden ähnlich: Der Anteil der Fehlerkorrektur ist mit etwa 25% relativ gering; die übrigen Wartungstätigkeiten, die meistens Entwurfsstrukturen beeinflussen, überwiegen mit etwa 75%. Daraus lässt sich schließen, dass die Entwurfsqualität bei der Wartung eine wesentliche Rolle spielt. Arthur (1988) betont dies mit der Aussage: „Maintenance productivity is a direct function of the quality of the existing system."

Dass Anpassung und Erweiterung unvermeidlich sind, hat sich auch schon in „Gesetzen" des Software Engineerings niedergeschlagen. Lehmans First Law of Software Evolution (Lehman, 1980) sagt: „A program that is used and that as an implementation of its specification reflects some reality, undergoes continual change or becomes progressively less useful." Das bedeutet, das System wird sich im Laufe der Zeit ändern – oder als unbrauchbar weggeworfen. Bersoffs First Law of System Engineering (Bersoff et al., 1980) verschärft diese Aussage noch: „No matter where you are in the system life cycle, the system will change and the desire to change it will persist

Wartungsart	Verteilung nach Lientz, Swanson (1980)	Verteilung nach Sneed (1988)
Stabilisierung	21,7%	25%
Optimierung	9,5%	15%
Anpassung	23,6%	20%
Erweiterung	41,8%	40%
Sonstiges	3,4%	–

Tabelle 8-1: Aufwandsverteilung in der Wartung

throughout the life cycle.“ Das System wird sich also schon während der Entwicklung ändern, z. B. weil neue Anforderungen hinzukommen oder vorhandene Anforderungen sich ändern.

Das Problem laufender Änderungen am Entwurf wird durch evolutionäre Entwicklungsprozesse (z. B. extreme Programmierung) noch verschlimmert. Bei dieser Vorgehensweise wird das System Schritt für Schritt entworfen und implementiert, so dass Entwurfserweiterungen und -restrukturierungen an der Tagesordnung sind. Bei extremer Programmierung wird das Änderungsproblem noch verschärft, da dort vorausschauendes Entwerfen (z. B. in Hinblick auf zukünftige Änderungen) explizit untersagt ist; es darf nur für die momentan zu bearbeitenden Anforderungen entworfen werden. Kommen dann später neue Anforderungen hinzu, muss notfalls der Entwurf durch Refaktorisierung überarbeitet werden.

Innerhalb der Wartung spielt die Verständlichkeit eine große Rolle. Nach Grady (1997, Kap. 4) entfallen momentan 28% der Kosten bei der Software-Entwicklung (einschließlich Wartung) auf das Programmverständnis (knowledge recovery). Wenn man nur die Wartung betrachtet, sind es sogar 50% (Corbi, 1989) bis 60% (Canfora et al., 1996). Demzufolge können Einsparungen erzielt werden, wenn das Programmverständnis (z. B. durch bessere Dokumentation) erleichtert wird. Grady rechnet mit Netto-Einsparungen von 3 bis 7%, falls solche Maßnahmen ergriffen werden. Daher liegt innerhalb der Wartbarkeit im Qualitätsmodell ein Schwerpunkt auf Kriterien, die die Verständlichkeit beeinflussen.

8.2 Aufbau des Modells

Das allgemeine Qualitätsmodell wird gemäß dem Factors-Criteria-Metrics-Schema (vgl. Abschnitt 6.2.1) aufgebaut. Die oberste Ebene bilden die Faktoren. Diese setzen sich aus Kriterien zusammen, die wiederum auf Metriken aufbauen.

8.2.1 Faktoren

Faktoren stehen für die Eigenschaften des Systems, die vorhergesagt werden sollen. Der wichtigste Faktor des Modells ist die Wartbarkeit (vgl. Abschnitt 8.1.5). Darüber hinaus gibt es die Faktoren Wiederverwendbarkeit, Wiederverwendung, Brauchbarkeit, Testbarkeit und Prüfbarkeit (vgl. Abschnitt 8.3 und die folgenden). Weitere Faktoren sind denkbar, wurden jedoch nicht aufgenommen, weil ihre Bewertung auf der Basis von ODEM schwierig ist (vgl. Abschnitt 8.9). Die Faktoren repräsentieren relativ

unabhängige Module des Modells. Dies erleichtert die Anpassung des Modells bei der Ableitung eines spezifischen Qualitätsmodells.

8.2.2 Kriterien

Die Kriterien stehen für Entwurfseigenschaften. Bei der Auswahl der Kriterien eines Faktors sind die folgenden Anforderungen ausschlaggebend:

- Relevanz: Alle Kriterien sollen für den Faktor von hoher Bedeutung sein.
- Verständlichkeit: Es sollen möglichst wenige Kriterien verwendet werden. Außerdem soll die Zahl der Hierarchiestufen gering sein, um Überschaubarkeit zu gewährleisten.
- Überdeckung: Die Kriterien sollen möglichst viele Aspekte des Faktors abdecken.
- Unabhängigkeit: Die Kriterien sollen sich so wenig wie möglich überlappen.
- Bewertbarkeit: Die Kriterien sollen möglichst objektiv bewertbar sein.

Die Faktoren und Kriterien wurden so gewählt, dass ein hoher Erfüllungsgrad (in der Regel) positiv für die Qualität ist. Daher wird z. B. statt des gebräuchlichen Begriffs Kopplung der Begriff Entkopplung verwendet, da geringe Kopplung, also hohe Entkopplung angestrebt wird.

8.2.3 Metriken

Bei den Metriken werden sowohl objektive als auch subjektive Metriken eingesetzt. Die objektiven Metriken sind automatisch bestimmbar und lassen sich auf der Basis von ODEM formal definieren. Leider können mit den objektiven Metriken nicht alle Aspekte der Kriterien abgedeckt werden. Daher werden zusätzlich subjektive Metriken eingesetzt, die von einem Bewerter nach seiner persönlichen Einschätzung bestimmt werden. Dem Bewerter werden zusätzlich Fragebögen zur Verfügung gestellt, die ihn bei der Meinungsbildung unterstützen.

Dieses Kapitel beschäftigt sich nur mit den Faktoren und Kriterien des Modells. Die Metriken und die Quantifizierung des Modells auf der Basis der Metriken sowie der Ablauf der Bewertung im Detail werden im folgenden Kapitel behandelt.

8.3 Wartbarkeit

Definition

Die Wartbarkeit ist hoch, wenn der Entwurf leicht korrigiert, überarbeitet und erweitert werden kann. Die oft separat verwendeten Begriffe Änderbarkeit und Erweiterbarkeit werden hier unter Wartbarkeit subsumiert.

Unter Wartung wird hier alles verstanden, was mit Änderungen des Produkts nach der ersten Version zu tun hat, sei es nun in einem neuen Iterationszyklus während der Entwicklung oder nach der Auslieferung. Testbarkeit wird häufig als zur Wartung gehörig angesehen (z. B. Boehm et al., 1978); hier wird sie aber als eigenständiger Faktor betrachtet. Wie sich zeigen wird, sind die Kriterien der Testbarkeit eine Teilmenge

der Kriterien der Wartbarkeit, weshalb bei der Bewertung der Wartbarkeit nicht wirklich etwas fehlt.

Kriterien bisheriger Qualitätsmodelle

Um die Auswahl der Kriterien für die Wartbarkeit zu erleichtern, werden hier zunächst die Kriterien der Qualitätsmodelle aus Kapitel 6 und Abschnitt 7.4 zusammengetragen, welche die Wartbarkeit betreffen (siehe Tabelle 8-2). Zusammen mit jedem Kriterium, für das unter Umständen in der Literatur mehrere Bezeichnungen verwendet werden, ist die Quelle des Modells angegeben, in dem es vorkommt.

Kriterium	Nennungen
Allgemeinheit	McCall et al. (1977)
Änderbarkeit/Erweiterbarkeit	McCall et al. (1977), Boehm et al. (1978), ISO/IEC 9126:1991, IEEE 1061-1992
Einfachheit	McCall et al. (1977), Coad, Yourdon (1991), Erni (1996)
Flexibilität	McCall et al. (1977), Erni (1996)
Knappheit/Größe	McCall et al. (1977), Coad, Yourdon (1991)
Konsistenz	Boehm et al. (1978)
Kopplung/Entkopplung	Booch (1994), Coad, Yourdon (1991), Erni (1996)
Lesbarkeit	Boehm et al. (1978)
Modularität	McCall et al. (1977), Erni (1996)
Selbsterklärung	McCall et al. (1977), Boehm et al. (1978)
Stabilität	ISO/IEC 9126:1991
Strukturiertheit	Boehm et al. (1978)
Verständlichkeit/Analysierbarkeit/Klarheit	Boehm et al. (1978), ISO/IEC 9126:1991, Coad, Yourdon (1991), Erni (1996)
Wartbarkeit	McCall et al. (1977), Boehm et al. (1978), ISO/IEC 9126:1991, IEEE 1061-1992
Zusammenhalt	Booch (1994), Coad, Yourdon (1991), Erni (1996)

Tabelle 8-2: Kriterien zur Wartbarkeit aus der Literatur

Gewählte Kriterien

Aus der Sammlung der Kriterien in Tabelle 8-2 werden Kriterien für die Wartbarkeit ausgewählt. Aus Tabelle 8-2 direkt übernommen werden Knappheit, Strukturiertheit, Entkopplung und Zusammenhalt. Änderbarkeit, Flexibilität und Stabilität werden unter Wartbarkeit subsumiert. Selbsterklärung, Verständlichkeit, Konsistenz und Lesbarkeit werden zu den Kriterien Einheitlichkeit und Dokumentierung zusammengefasst. Weggelassen wird Einfachheit, weil Einfachheit etwas ist, was sich – genau so wie die in Abschnitt 7.2.3 angesprochene Eleganz – nirgendwo festmachen lässt. Einfachheit zieht sich vielmehr durch alle Kriterien hindurch. Am ehesten spiegelt sich die Einfachheit noch in der Knappheit, der Strukturiertheit und der Entkopplung. Die Allgemeinheit wird ebenfalls weggelassen und stattdessen dem Faktor Wiederverwendung zugeordnet, da sich Allgemeinheit in der Regel negativ auf die Wartbarkeit

auswirkt (höhere Komplexität). Neu hinzugenommen wird die Verfolgbarkeit, die in keinem der untersuchten Modelle vorkommt.

Die resultierenden Kriterien des Faktors Wartbarkeit sind in Abbildung 8-1 dargestellt. Ein Pfeil bedeutet, dass das Kriterium den Faktor positiv beeinflusst. Im Folgenden werden die einzelnen Kriterien (von links nach rechts) diskutiert.

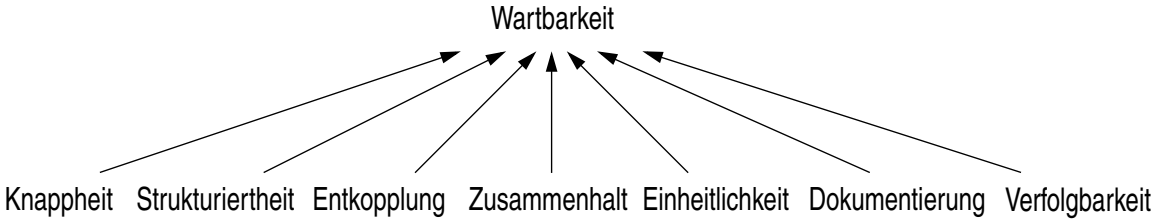

Abbildung 8-1: Kriterien des Faktors Wartbarkeit

8.3.1 Knappheit

Weniger ist mehr.
(Mies van der Rohe)

Vollkommenheit entsteht offensichtlich nicht dann, wenn man nichts mehr hinzuzufügen hat, sondern wenn man nichts mehr wegnehmen kann.
(Antoine de Saint-Exupéry, Wind, Sand und Sterne)

Inside every large program there is a small program trying to get out.
(C.A.R. Hoare)

Ultimately, the best software design assimilates all its functions to a few clear and simple principles.
(Nelson, 1990, S. 236)

Definition

Knappheit bedeutet, im Entwurf mit wenig Konstrukten in einer geringen Anzahl von Ausprägungen auszukommen. Schon die große Zahl an Zitaten aus verschiedenen Bereichen (Architektur, Flugzeugbau und Software-Entwicklung) deutet auf die große Bedeutung der Knappheit hin.

Beispiele für Knappheit beim objektorientierten Entwurf sind:

- eine minimale Anzahl von Klassen,
- eine minimale öffentliche Schnittstelle einer Klasse,
- eine minimale Anzahl von Attributen und Methoden in einer Klasse,
- eine minimale Anzahl von Parametern bei Methoden,
- eine minimale Anzahl von Beziehungen (Assoziationen, Vererbung etc.) zwischen Klassen
- das Vermeiden von Code-Duplikation durch entsprechenden Entwurf (z. B. Template Methods; Gamma et al., 1995).

Diskussion

Knappheit verbessert die Verständlichkeit eines Entwurfs. Außerdem ist er wegen des geringeren Umfangs leichter zu dokumentieren und schneller zu implementieren. Schließlich entsteht auch weniger Code, der getestet und gewartet werden muss. Durch einen geringeren Umfang nimmt auch der Aufwand für Prüfung und Überarbeitung des Entwurfs ab. Allerdings führt sehr hohe Knappheit (z. B. nur sehr wenige, dafür umfangreiche Klassen) zu einer geringeren Verständlichkeit, da dann Dinge zusammengefasst werden, die eigentlich nichts miteinander zu tun haben (führt zu einem geringen Zusammenhalt). Im Extremfall besteht das System aus einer einzigen Klasse System mit einer einzigen Methode run, in der sämtliche Funktionalität implementiert ist – für die Wartung wohl der schlimmste Fall.

Die Knappheit steht in Konkurrenz zu Entkopplung und Zusammenhalt. Beispielsweise kann man zur Entkopplung zweier Komponenten eine weitere Komponente einführen, wie das beim Mediator (vgl. Gamma et al., 1995) der Fall ist. Damit verringert sich zwar die Kopplung, aber die Anzahl der Klassen steigt. In der anderen Richtung kann man zwar die Zahl der Klassen verringern, indem man einige miteinander verschmilzt, man erhält dadurch aber meistens Klassen mit sehr geringem Zusammenhalt. Ein guter Entwurf sucht also nach der richtigen Balance zwischen Knappheit auf der einen Seite, Entkopplung und Zusammenhalt auf der anderen Seite.

Reduktion, also eine Vereinfachung ohne Verlust an anderen Qualitäten, ist immer erstrebenswert; häufig führt sie auch zu mehr Eleganz. Allerdings ist die Reduktion oft teuer, da sie viel Kreativität und Arbeit erfordert. Mies van der Rohes Wahlspruch „So einfach wie möglich, koste es was es wolle“ ist daher nur dann sinnvoll, wenn die Kosten keine Rolle spielen. Andernfalls ist absolute Knappheit unwirtschaftlich.

Die Vermeidung von Redundanz ist ein spezieller Aspekt der Knappheit. Redundanzfreiheit ist besonders wichtig für die Wartbarkeit, da bei vorhandener Redundanz (z. B. durch Copy-and-Paste-Programming, Brown et al., 1998) immer alle Kopien geändert werden müssen (Burd, Munro, 1997). Dazu müssen diese Kopien erst einmal gefunden und dann konsistent geändert werden. Das ist viel fehleranfälliger, als wenn die Änderung auf eine Stelle beschränkt wäre.

8.3.2 Strukturiertheit

The only problems we can really solve in a satisfactory manner are those that finally admit a nicely factored solution.
(Dijkstra, 1972)

Definition

Die Strukturiertheit ist hoch, wenn die Struktur des Entwurfs von einem Menschen leicht überblickt und erfasst werden kann. Laut Melton et al. (1990) sind die psychologische (d. h. die wahrgenommene) Komplexität und die strukturelle Komplexität nicht dasselbe, da bei der psychologischen Komplexität der Leser eine wesentliche Rolle spielt, d. h. sie ist individuell verschieden. Die strukturelle Komplexität ist allerdings ein guter Indikator für die psychologische Komplexität.

Diskussion

Für den Menschen scheinen hierarchische Strukturen besonders verständlich zu sein (Simon, 1962). Hierarchische Strukturen erlauben den Umgang mit einer großen Anzahl von Teilen, weil zu einem Zeitpunkt immer nur ein Ausschnitt betrachtet werden muss. Durch die Hierarchie besteht die Möglichkeit zur Abstraktion (z. B. die Bildung von Schichten), da Verfeinerungen bei Bedarf ausgeblendet werden können. Parnas (1974) stellt fest, dass man bei dem Begriff hierarchisch aufpassen muss, da es verschiedene Hierarchien gibt. In der Objektorientierung gibt es z. B. bei den Paketen eine hierarchische Struktur durch Schachtelung, bei den Klassen durch Vererbung. Hierarchien sind zyklenfrei, daher ist die Vermeidung von Zyklen ein wichtiges Ziel. Weitere Kriterien für die Strukturiertheit einer Hierarchie sind ihre Tiefe und der Verzweigungsgrad innerhalb der Hierarchie.

8.3.3 Entkopplung

[…] the best programs are designed in terms of loosely coupled functions that each does a simple task.
(Kernighan, Plauger, 1974)

If one intends to build quality models of OO design, coupling will very likely be an important structural dimension to consider.
(Briand et al., 1998, S. 30)

Definition

Kopplung (coupling; vgl. Stevens et al., 1974) ist ein Maß für die Stärke der Verbindung (und damit der Abhängigkeit) von Komponenten untereinander. Entkopplung ist das Gegenteil von Kopplung. Obwohl Kopplung der in der Literatur normalerweise verwendete Begriff ist, wurde hier der Begriff Entkopplung gewählt, damit ein hoher Erfüllungsgrad hohe Qualität impliziert.

In der Objektorientierung kann die Kopplung zunächst in drei verschiedene Arten unterschieden werden (Li, 1992):

- Kopplung durch Vererbung: Eine Unterklasse ist durch das Erben von Eigenschaften mit ihrer Oberklasse gekoppelt. Jede Änderung der Oberklasse hat Auswirkungen auf die Unterklasse.
- Kopplung durch Methodenaufruf: Jeder Methodenaufruf durch andere Klassen und jeder Aufruf von Methoden anderer Klassen erhöht die Kopplung.
- Kopplung durch Datenabstraktion: Die Verwendung anderer Klassen als Typ von Attributen erhöht die Kopplung.

Betrachtet man diese Arten genauer, stellt man fest, dass es insgesamt folgende Kopplungsarten im objektorientierten Entwurf gibt:

- Methode-Operation: Eine Methode ruft eine Operation (oder einen Konstruktor/Destruktor) auf. Es ergibt sich eine Kopplung in Richtung des Aufrufs; implizit entsteht auch eine Kopplung zwischen den Klassen[1], welche die Methode bzw. die Operation enthalten.

1. Statt einer Klasse kann es auch ein Interface sein (das gilt ebenso bei den anderen Kopplungsarten).

- Methode-Attribut: Eine Methode greift (lesend oder schreibend) auf ein Attribut zu. Durch den Zugriff ergibt sich implizit eine Kopplung zwischen den Klassen, welche die Methode bzw. das Attribut enthalten.
- Operation-Klasse: Eine Operation besitzt einen Parameter oder eine Rückgabe von einem Klassentyp. Dadurch ergibt sich wiederum eine implizite Kopplung zwischen der Klasse mit der Operation und der Klasse des Parameters bzw. der Rückgabe.
- Klasse-Klasse (Vererbung): Eine Klasse erbt von einer anderen Klasse. Faktisch werden sämtliche Kopplungen der Oberklasse ebenfalls geerbt. Durch Redefinition einer Methode könnten zwar Kopplungen durch Methodenaufruf oder Attributzugriff wegfallen, dies ist allerdings in der Praxis eher unwahrscheinlich.
- Klasse-Klasse (Assoziation): Es gibt eine (gerichtete) Assoziation der Klasse mit der anderen Klasse.
- Paket-Paket: Die explizite oder implizite Kopplung von Klassen überträgt sich als implizite Kopplung auf die Pakete, in denen die Klassen liegen.

Diskussion

Angestrebt wird eine hohe Entkopplung der Komponenten. Das bedeutet zum einen eine möglichst geringe Anzahl von Verbindungen zwischen Komponenten, zum anderen bei den vorhandenen Verbindungen eine möglichst schwache Kopplungsart. Entkopplung erhöht die Wahrscheinlichkeit, dass Änderungen sich nur lokal auf eine Komponente auswirken und nicht auf andere Komponenten ausstrahlen (Wellenеffekt). Außerdem ist die Wahrscheinlichkeit gering, dass Fehler in anderen Komponenten Folgefehler in der Komponente selbst verursachen. Schließlich ergibt sich auch eine bessere Verständlichkeit, da zum Verständnis einer Komponente weniger andere Komponenten verstanden werden müssen.

Dass sich hohe Entkopplung positiv auf die Wartbarkeit auswirkt, zeigen empirische Untersuchungen von Yin und Winchester (1978), Troy und Zweben (1981) sowie von Rombach (1990) für den strukturierten Entwurf und von Binkley und Schach (1996) für den objektorientierten Entwurf. Die genannten Untersuchungen haben auch gezeigt, dass Entkopplungsmetriken die stärksten Indikatoren für die Wartbarkeit sind.

8.3.4 Zusammenhalt

Definition

Zusammenhalt (cohesion; vgl. Stevens et al., 1974) ist ein Maß für die Stärke der (semantischen) Zusammengehörigkeit von Bestandteilen einer Komponente. Alles, was zusammengehört, sollte sich in einer Komponente befinden, aber nicht mehr. In der Objektorientierung lassen sich zwei Ebenen des Zusammenhalts unterscheiden: die Zusammengehörigkeit von Attributen und Operationen in einer Klasse und die Zusammengehörigkeit von Klassen in einem Paket.

Diskussion

Zusammenhalt verbessert die Verständlichkeit und damit die Wartbarkeit. Durch hohen Zusammenhalt ist die Wahrscheinlichkeit hoch, dass bei einer Änderung in der Regel nur eine Komponente betroffen ist, da alle Aspekte einer Abstraktion an einem Ort zusammengefasst sind. Welleneffekte treten daher in geringerem Maße auf.

8.3.5 Einheitlichkeit

Definition

Ein Entwurf ist einheitlich, wenn er einem einheitlichen Stil folgt (auch bekannt als konzeptionelle Integrität, siehe Abschnitt 7.3.1). Der Entwurf soll, auch wenn er von verschiedenen Personen und in mehreren Iterationen bearbeitet wurde, so aussehen, als sei er von einer Person in einem Guss erzeugt worden.

Diskussion

Einheitlichkeit erhöht die Verständlichkeit des Entwurfs. Sie kann mit einfachen Mitteln wie Namenskonventionen erreicht werden. Auch die Einhaltung von Standards (auf die in der Entwurfsdokumentation hingewiesen werden muss) ist hilfreich. Die Konventionen und Standards sollten so gewählt sein, dass alle Entwickler, die den Entwurf bearbeiten oder als Vorlage für die Implementierung verwenden sollen, mit ihnen vertraut sind.

8.3.6 Dokumentierung

Definition

Dokumentierung ist die Güte der Darstellung des Entwurfs in der Entwurfsdokumentation. Sie wird beeinflusst durch die Wahl und Nutzung der Notation, die Strukturierung der Dokumentation und deren Vollständigkeit.

Diskussion

Die Dokumentation ist für die Verständlichkeit wichtig, weil sie Bedeutung und Zweck der Entwurfsbestandteile sowie ihr Zusammenspiel festhält. Sie sollte gut strukturiert sein, d. h. sich durch ihren Aufbau dem Leser leicht erschließen. Außerdem sollte sie vollständig, konsistent, präzise und korrekt sein.

Die Notation sollte allen Entwicklern vertraut und leicht erlernbar sein. Außerdem sollte sich möglichst viel möglichst präzise in ihr ausdrücken lassen. Schließlich sollte sie robust sein, das heißt kleine Änderungen der Syntax sollten nicht große Änderungen der Semantik verursachen. Ansonsten können kleine Fehler in der Dokumentation zu großen Fehlern in der Implementierung führen. Die beste Notation ist allerdings ohne Nutzen, wenn die darin erstellte Dokumentation nichts taugt.

8.3.7 Verfolgbarkeit

Definition

Verfolgbarkeit (traceability) ist die Möglichkeit, Entwurfsentscheidungen auf die Anforderungen zurückzuführen (und umgekehrt). Beispielsweise sollte zu einer Klasse angegeben werden, für welche Anwendungsfälle der Spezifikation sie benötigt wird.

Diskussion

Wenn sich Anforderungen ändern, ist es durch Verfolgbarkeit leichter, die betroffenen Stellen im Entwurf zu identifizieren. Dadurch wird die Wartbarkeit verbessert. Strukturähnlichkeiten zwischen Problem- und Lösungsstruktur (structural correspondence, vgl. Abschnitt 7.3.1) erleichtern die Verfolgbarkeit ungemein, ohne dass es dazu aufwendiger Dokumentation bedarf. Diese Strukturähnlichkeit ist eine der versprochenen Vorteile der Objektorientierung, weshalb eine entsprechende objektorientierte Vorgehensweise für gute Verfolgbarkeit sorgen kann.

Allerdings hält Royce (2000) das Streben nach detaillierter Verfolgbarkeit für überholt, da es häufig dazu führt, dass der Entwurf auf dieselbe Weise strukturiert wird wie die Anforderungsdokumentation. Das erweist sich als hinderlich, wenn viele Bestandteile aus anderen Quellen wiederverwendet werden sollen, weil diese oft ganz andere Strukturanforderungen mit sich bringen. Daher ergibt sich bei hoher Wiederverwendung eine Entwurfsstruktur, die mit der Struktur der Anforderungen nicht mehr viel gemein hat. Hohe Verfolgbarkeit herzustellen ist dann mit hohem Aufwand verbunden; sie steht also im Widerspruch zur Wiederverwendung.

8.4 Wiederverwendung

Definition

Es gibt zwei Arten von Wiederverwendung: die mehrfache Nutzung neu erstellter Komponenten innerhalb desselben Projekts oder die Wiederverwendung früher erstellter Komponenten in anderen Projekten (Rumbaugh et al., 1993). Biemann (1992) nennt diese Arten interne und externe Wiederverwendung. Interne Wiederverwendung verringert die Redundanz. Externe Wiederverwendung hat den Vorteil, dass bewährte Komponenten erneut verwendet werden (Zweben et al., 1995). Im Optimalfall, der unveränderten Wiederverwendung, entstehen gar keine Entwicklungskosten; aber auch bei einer modifizierenden Wiederverwendung sind die Kosten meistens geringer als bei einer Neuentwicklung. In der Objektorientierung ist zusätzlich eine Zwischenform möglich: Eine Klasse wird durch Vererbung unverändert wiederverwendet und trotzdem modifiziert durch Redefinitionen und Erweiterungen in der Unterklasse.

Diskussion

Interne Wiederverwendung kann zu geringerer Redundanz führen, weil die Entwerfer sich darum bemühen müssen, Gemeinsamkeiten herauszulösen und an einer Stelle zusammenzufassen. Dies führt in der Regel zu höherer Knappheit. Externe Wie-

derverwendung hingegen kann sich negativ auf die Knappheit auswirken, da die wiederverwendete Komponente möglicherweise mehr Funktion bietet als benötigt wird. Es ist allerdings selten wirtschaftlich, sie nur darum zu modifizieren. Der Faktor Wiederverwendung lässt sich nicht in sinnvolle Kriterien zergliedern.

8.5 Wiederverwendbarkeit

Definition

Wiederverwendbarkeit bezieht sich auf die Möglichkeit, das entworfene System als Ganzes leicht modifiziert in einem anderen Kontext wiederzuverwenden, oder Teilsysteme und einzelne Komponenten unverändert oder leicht modifiziert in anderen Systemen wiederzuverwenden. Je einfacher das ist, desto höher ist die Wiederverwendbarkeit.

Diskussion

Für die Wiederverwendbarkeit spielt die Allgemeinheit des Entwurfs eine wichtige Rolle, da sich allgemeine Bausteine leichter an neue Kontexte anpassen lassen. Außerdem hilft Verständlichkeit bei der Wiederverwendung, da man kaum etwas wiederverwenden möchte, was man nicht versteht. Daher sind die Kriterien Knappheit, Strukturiertheit, Entkopplung, Zusammenhalt, Einheitlichkeit und Dokumentation aus der Wartbarkeit auch für die Wiederverwendbarkeit wichtig. Schwach gekoppelte Komponenten können besser wiederverwendet werden, da weniger andere Komponenten mittransferiert werden müssen (vgl. z. B. Page-Jones, 1988). Knappheit kann allerdings auch negativen Einfluss auf die Wiederverwendbarkeit haben, nämlich dann, wenn Klassen so speziell für den Anwendungszweck entworfen werden, dass sie sich in einem anderen Kontext nicht wiederverwenden lassen.

Vergleicht man die Kriterien der Wiederverwendbarkeit und der Wartbarkeit, stellt man fest, dass sie nahezu deckungsgleich sind (in Abbildung 8-2 sind die „importierten“ Kriterien kursiv gesetzt; es fehlt nur die Verfolgbarkeit).

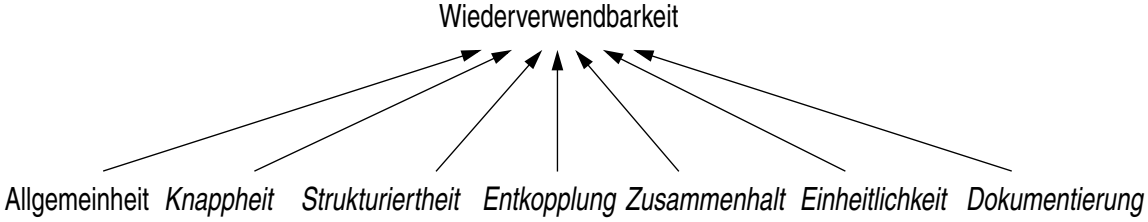

Abbildung 8-2: Kriterien des Faktors Wiederverwendbarkeit

8.5.1 Allgemeinheit

Definition

Da man nicht alle Möglichkeiten der Wiederverwendung voraussehen kann, muss eine Komponente oft vor der Wiederverwendung angepasst werden. Dies ist leichter möglich, wenn sie Allgemeinheit (auch Flexibilität genannt) besitzt. Allgemeinheit bedeutet auch Vollständigkeit der Abstraktion, d. h. eine Komponente deckt einen möglichst großen Anwendungsbereich ab (vgl. Abschnitt 7.4.1).

Diskussion

Eine Komponente allgemein zu machen ist eine Investition in die Zukunft, von der oft nicht klar ist, ob sie sich lohnt. Wird die Komponente später auf die vorgesehene Weise wiederverwendet, hat sich die Investition gelohnt – wenn nicht, war sie unnötiger Aufwand. Allgemeinheit wirkt sich in der Regel negativ auf die Knappheit aus, weil zur Realisierung mehr Entwurfsbestandteile benötigt werden.

8.6 Brauchbarkeit

Definition

Dieser Faktor umfasst Kriterien, die sich auf die Brauchbarkeit des Entwurfs als Basis einer Realisierung der Anforderungen beziehen. Ein Entwurf ist brauchbar, wenn er eine tatsächliche Lösung für das Problem darstellt, d. h. die Anforderungen wiederspiegelt und realisierbar ist.

Diskussion

Für die Brauchbarkeit sind Korrektheit des Entwurfs bezüglich der Spezifikation und Realisierbarkeit wichtig (vgl. Abbildung 8-3).

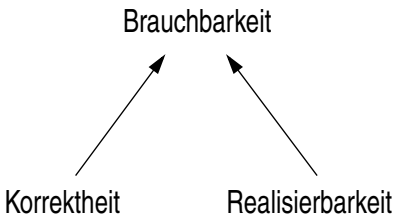

Abbildung 8-3: Kriterien des Faktors Brauchbarkeit

8.6.1 Korrektheit

Looking honestly at the situation, we are never looking for the best program, seldom looking for a good one, but always looking for one that meets the requirements.
(Weinberg, 1971, S. 17)

A good design is a design that conforms to specifications.
(Martin, 1995, S. 189)

Definition

Ein Entwurf ist korrekt, wenn er alle funktionalen Anforderungen erfüllt. Er muss eine vollständige Lösung des Problems beschreiben (Forderung nach Funktionsvollständigkeit), aber nicht mehr. Die umgesetzte Funktionalität muss der Spezifikation gerecht werden (Forderung nach Funktionskorrektheit).

8.6.2 Realisierbarkeit

Definition

Der Entwurf ist realisierbar, wenn er vom vorgesehenen Personal in technischer und organisatorischer Hinsicht mit dem vorgesehenen Aufwand implementiert werden

kann. Technische Aspekte sind z. B. die Zielplattformen, die einzusetzenden Hilfsmitteln (Werkzeuge, Programmiersprachen etc.) und die zu verwendenden Komponenten (Hard- und Software). Wenn also zwei zu verwendende Komponenten unverträglich sind, ist der Entwurf aus technischen Gründen nicht realisierbar. Der Entwurf muss auch auf die entwickelnde Organisation passen: Diese muss über die Ressourcen verfügen, um die Lösung zu realisieren, z. B. genügend Entwickler mit dem notwendigen Wissen (z. B. über Methoden, Programmiersprachen, Werkzeuge und den Anwendungsbereich) und die notwendigen Werkzeuge (Rechner, Entwicklungswerkzeuge). Ansonsten ist der Entwurf aus organisatorischen Gründen nicht realisierbar.

Diskussion

Ein Lösungsvorschlag ist völlig wertlos, wenn er nicht realisiert werden kann. Deshalb muss die Realisierbarkeit vollständig sichergestellt sein. Um realisierbar zu sein, muss der Entwurf in sich widerspruchsfrei (also konsistent) sein. Ein bereits genanntes Beispiel für eine Inkonsistenz sind zwei miteinander unverträgliche Komponenten, was spätestens bei der Integration zu Problemen führen wird. Ein anderes Beispiel ist eine Schnittstelle einer Komponente, die nicht mächtig genug ist, um den (tatsächlichen) Bedarf einer anderen Komponente gerecht zu werden. Ein solches Problem stellt man häufig erst in der Implementierung fest.

8.7 Testbarkeit

Definition

Die Testbarkeit eines Entwurfs ist hoch, wenn Komponenten- und Integrationstests der implementierten Lösung leicht möglich sind.

Diskussion

Zunächst bestimmt die Knappheit den Testaufwand bei Komponententests. Je weniger zu testen ist, desto schneller kann es gehen. Außerdem wird die Testbarkeit durch Entkopplung verbessert: Bei hoher Entkopplung werden für den Test einer einzelnen Komponente nicht so viele andere Komponenten benötigt. Dadurch kann auch früher mit dem Test begonnen werden und die Fehlersuche wird einfacher.

Für den Integrationstest ist es hilfreich, wenn der Entwurf es gestattet, bottom-up einzelne Komponenten und Teilsysteme zu integrieren und nach jedem Integrationsschritt einen Test auszuführen, bis schließlich das vollständige System entstanden ist und einem Systemtest unterzogen werden kann. Auch hier spielt die Entkopplung die wichtigste Rolle.

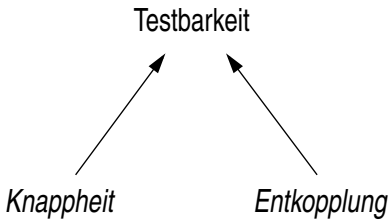

Abbildung 8-4: Kriterien des Faktors Testbarkeit

Die Testbarkeit von objektorientierten Programmen hat ihre fundamentalen Probleme. Smith und Robson (1990) nennen als Ursache unter anderem Vererbung und Polymorphismus sowie Überladen und Redefinition von Methoden. Beispielsweise kann das Hinzufügen einer neuen Unterklasse die Funktion eines bisher korrekten Systems untergraben, wenn auf einmal die Methoden der neuen Unterklasse aufgerufen werden statt wie bisher die Methoden der Oberklasse.

8.8 Prüfbarkeit

Definition

Ein Entwurf ist prüfbar, wenn mit geringem Aufwand Prüfungen (z. B. Inspektionen) auf ihm durchgeführt werden können.

Diskussion

Zum einen muss geprüft werden können, ob der Entwurf eine Lösung des Problems ist, d. h. ob er korrekt ist. Dafür ist Verfolgbarkeit wichtig. Zum anderen sollte der Entwurf zur Prüfung so aufgeteilt werden können, dass der Einsatz mehrerer Gutachter möglich ist. Dies wird durch hohe Knappheit und Entkopplung erleichtert; hohe Knappheit sorgt auch dafür, dass weniger zu prüfen ist. Dunsmore et al. (2000) stellen allerdings fest, dass eine sinnvolle Aufteilung zur Inspektion bei objektorientierten Systemen im Vergleich zu strukturierten Systemen grundsätzlich schwieriger ist, da mehr Abhängigkeiten der Komponenten untereinander vorhanden sind.

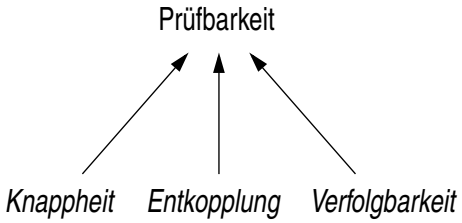

Abbildung 8-5: Kriterien des Faktors Prüfbarkeit

8.9 Weitere mögliche Faktoren

Die bisher vorgestellten Faktoren decken bei weitem nicht alle möglichen ab. Kandidaten für weitere Faktoren sind: Portabilität, Effizienz (Performance), Robustheit, Interoperabilität, Sicherheit, Verfügbarkeit, Zuverlässigkeit und Skalierbarkeit. Will man auch die Benutzungsoberfläche aus Benutzersicht bewerten, kann man entsprechende Faktoren hinzunehmen, z. B. Benutzungsfreundlichkeit. Je nach den konkreten Qualitätsanforderungen des zu entwickelnden Systems können noch spezielle Kriterien hinzukommen, z. B. Verklemmungsfreiheit bei parallelen Systemen.

Für alle diese Faktoren wird zur Bewertung zusätzliche Entwurfsinformation benötigt, die in ODEM nicht verfügbar ist. Deshalb werden sie hier nicht weiter in Betracht gezogen.

Kapitel 9

Quantifizierung des Qualitätsmodells

An important lesson learned from the application of software measurement is that it is better to start from a set of metrics addressing important improvement areas, and evolve these metrics over time, instead of debating forever, trying to find perfect metrics.
(Daskalantonakis, 1992, S. 1008)

Im vorhergehenden Kapitel wurden die Faktoren und Kriterien angegeben, die im Rahmen von QOOD als für die Entwurfsqualität wichtig angesehen werden. Dieses Kapitel beschäftigt sich damit, wie eine quantitative Bewertung der Entwurfsqualität durchgeführt werden kann. Die Quantifizierung wird am Beispiel des wichtigsten Faktors, der Wartbarkeit, gezeigt. Da viele Kriterien der Wartbarkeit auch den anderen Faktoren zugeordnet sind, müssen für eine vollständige Quantifizierung von QOOD nur noch wenige weitere Kriterien quantifiziert werden.

Zunächst wird das quantitative Bewertungsverfahren skizziert, bevor auf die einzelnen Komponenten des Verfahrens und ihre Auswertung eingegangen wird. Dann wird beschrieben, wie die verschiedenen Komponenten kombiniert werden, um eine Gesamtbewertung zu erhalten, und welche Bewertungsschritte automatisiert werden können. Abschließend wird diskutiert, wie spezifische Modelle aus QOOD abgeleitet werden.

9.1 Bewertungsverfahren

Das Bewertungsverfahren beruht auf drei Komponenten: objektiven Metriken, subjektiven Metriken und Fragebögen. Diese bilden zusammen die Basis der Bewertung. Durch Aggregation von Einzelbewertungen über Kriterien und Faktoren erhält man eine Gesamtbewertung.

Die Metriken und Fragebögen sind klassifiziert nach dem Kriterium, zu dem sie gehören. Zusätzlich gibt es eine Unterklassifikation nach der Ebene (oder Granularität) der Bewertung. Im Metrikenrahmenwerk von Henderson-Sellers et al. (1993) werden die

Ebenen Methode, Dienst, Klasse, Subsystem und (Gesamt-)System unterschieden. Da hier kein Feinentwurf oder Code betrachtet wird, werden nur die Ebenen Klasse/ Interface, Paket (anstelle von Subsystem) und System verwendet.

9.1.1 Objektive Metriken

Zur Quantifizierung werden soweit wie möglich objektive, automatisch bestimmbare Metriken eingesetzt, um den Bewertungsprozess durch Werkzeugunterstützung erleichtern zu können. Die Metriken messen Eigenschaften der Entwurfsartefakte, hier also Eigenschaften des in UML-Diagrammen beschriebenen UML-Modells. Die Metriken sind auf der Basis des formalen Modells ODEM definiert, das in Kapitel 5 eingeführt wurde.

9.1.2 Subjektive Metriken

Die Quantifizierung durch objektive Metriken ist nicht immer möglich, wie Card und Glass (1990, S. 116) bestätigen: „The intellectual nature of the software design process means that important components of [the design] must be measured subjectively." Die tatsächliche Qualität eines Entwurfs hängt in hohem Maße von seinem Kontext und von semantischen Fragen ab, die durch objektive Metriken schwer zu erfassen sind. Lewerentz et al. (2000, S. 68) empfehlen zum Einsatz von Metriken zur Qualitätsbewertung: „Use them but do not trust them: The ultimate assessment has to be done by human inspection." Auch Dick und Hunter (1994, S. 321) betonen die Bedeutung der zusätzlichen subjektiven Bewertung durch einen Experten: „software evaluation […] is best seen as a symbiosis of human and machine, each performing the tasks to which it is best suited." Der Computer übernimmt die Erhebung der automatisierbaren objektiven Metriken, während sich der Mensch um die nicht automatisierbaren subjektiven Metriken und die Gesamtbewertung kümmert.

9.1.3 Fragebögen

Um die subjektive Bewertung zu erleichtern, werden Fragebögen angegeben, die zur Prüfung auf erwünschte und unerwünschte Eigenschaften dienen. Außerdem sorgen die Fragebögen dafür, dass die Bewertungen durch verschiedene Bewerter nicht zu stark voneinander abweichen. Auf diese Weise wird die Reproduzierbarkeit der subjektiven Bewertung verbessert, ebenso die übrigen Kriterien des ISO/IEC Guide 25 (vgl. Abschnitt 7.6.1).

9.1.4 Gesamtbewertung

Als Gesamtbewertung wird für jedes Kriterium, für jeden Faktor und für den Entwurf insgesamt eine subjektive Metrik verwendet. Diese stützt sich auf die objektiven Metriken, die Antworten zu den Fragebögen und die subjektiven Metriken untergeordneter Elemente (sofern verfügbar), aus denen Bewertungsvorschläge berechnet werden können (vgl. Abbildung 9-1).

Die objektiven Metriken allein sind zur Bewertung der Entwurfsqualität nicht ausreichend, da sie im Wesentlichen auf syntaktische Aspekte des Entwurfs beschränkt sind. Für die semantische Beurteilung sind Fragebögen besser geeignet. Die Fragebögen enthalten allerdings Fragen, für deren Beantwortung Messwerte der objektiven

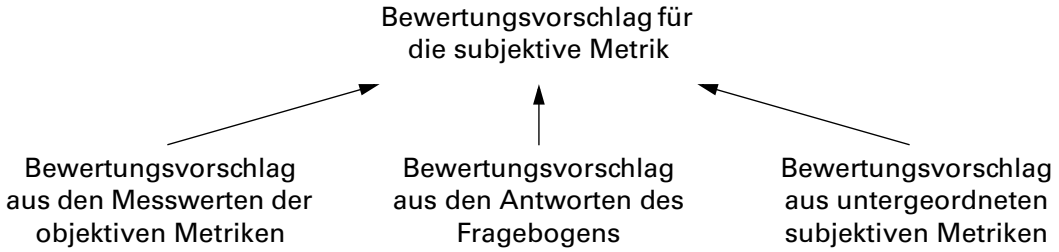

Abbildung 9-1: Aggregation der Bewertungsvorschläge

Metriken benötigt werden. Also führt erst die Kombination der beiden Ansätze zu einer gut funktionierenden Lösung.

Die Bewertung läuft bottom-up ab (vgl. Abbildung 9-2). Zunächst werden für die unterste Ebene (die Klassen und Interfaces) für jedes Kriterium die Metriken und Fragebögen erhoben und aus diesen für jedes Kriterium Bewertungsvorschläge für die subjektive Metrik abgeleitet. Der Bewerter bestimmt dann die Werte für die subjektiven Metriken. Aus den subjektiven Metriken der Kriterien werden dann die subjektiven Metriken für die Faktoren bestimmt und aus diesen wiederum die subjektive Metrik für die Gesamtqualität. Anschließend wird für die nächsthöhere Ebene (die Pakete) das Verfahren wiederholt, wobei in die subjektiven Metriken auch die jeweiligen Bewertungen der im Paket enthaltenen Klassen und Interfaces einfließen. Schließlich wird das Verfahren auf der obersten Ebene (System) wiederholt, wobei die subjektiven Metriken auch die subjektiven Bewertungen der Pakete berücksichtigen. So entsteht schließlich eine Bewertung der Gesamtqualität des Systems.

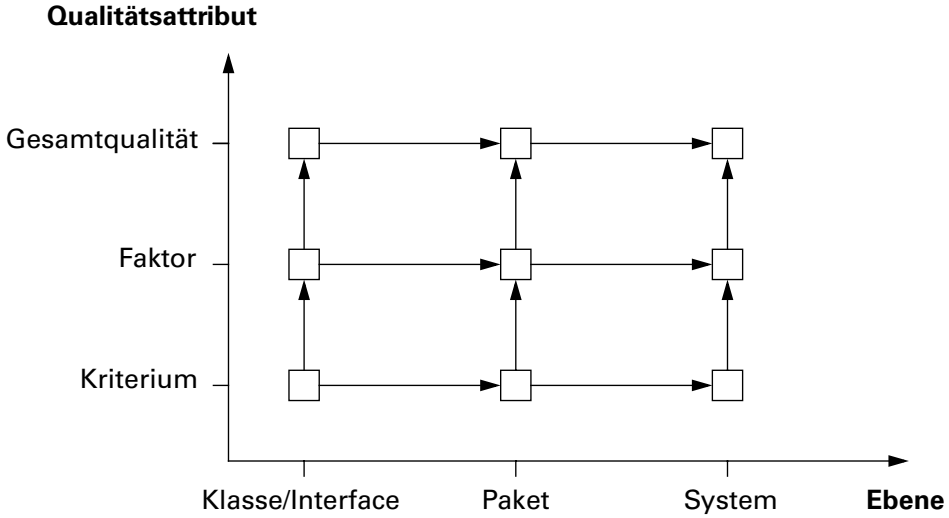

Abbildung 9-2: Aggregation bei der Bewertung

Eine einzelne Qualitätskennzahl als Ergebnis des gezeigten Aggregationsprozesses ist allerdings mit Vorsicht zu genießen, da zwischen den Qualitätskriterien häufig Abwägungen notwendig sind (Boehm et al., 1978). So kann z. B. die Knappheit häufig nur auf Kosten anderer Kriterien gesteigert werden. Diese widerstrebenden Kräfte sind bei einer einzelnen Zahl nicht mehr sichtbar. Daher sollten nach der Bewertung nicht nur das Endergebnis, sondern auch die Zwischenergebnisse ausgewiesen werden.

9.2 Objektive Metriken

Objektive Metriken sind vom Bewerter unabhängig und lassen sich automatisch erheben. Sie messen Eigenschaften des Entwurfs auf der Basis seines ODEM-Modells.

9.2.1 Anforderungen

In diesem Abschnitt werden die Anforderungen an die objektiven Metriken formuliert. Neben den allgemeinen Anforderungen an Metriken aus Abschnitt 2.2.2 gibt es hier weitere spezifische Anforderungen. Die wichtigste Anforderung ist, dass die Metriken mit dem zugeordneten Kriterium auf der zugeordneten Ebene korreliert sind.

Die Metriken sollen sich auf UML-Diagrammen, die typisch in einem objektorientierten Entwurf verwendet werden, erheben lassen. Die Erhebung der Metriken soll nicht voraussetzen, dass der Entwurf detailliert ausgearbeitet wurde. Da gleichzeitig eine präzise und eindeutige Definition der Metriken erwünscht ist, werden die Metriken mit Hilfe von ODEM formal definiert und lassen sich dann auf ODEM-Instanzen automatisch erheben.

Es sollen so wenig Metriken wie möglich verwendet werden. Dadurch kann das Modell einfacher angewendet werden. Allerdings können durch zu starke Reduktion der Anzahl der Metriken interessante Aspekte verloren gehen. Beispielsweise kann es sinnvoll sein, die Anzahl der Attribute einer Klasse sowohl mit als auch ohne geerbte Attribute zu zählen. Das eine ist ein Maß für die tatsächliche Größe, das andere ein Maß für die Größe der Klassendefinition in einer Programmiersprache. Daher werden zu den Zählmetriken Verfeinerungen angeboten, z. B. bei der Anzahl der Attribute eine Unterscheidung in geerbte und nicht geerbte. Diese Verfeinerungen brauchen nur bei Bedarf betrachtet werden und erhöhen daher die Komplexität des Modells nur unwesentlich. Andererseits erlauben sie eine differenziertere Betrachtung.

Es sollen möglichst einfache Metriken verwendet werden. Komplexe, zusammengesetzte Metriken sind schwieriger zu verstehen, zu erheben und zu validieren als einfache Metriken. Kitchenham et al. (1990) schreiben dazu: „It would therefore seem preferable to use design metrics based on primitive counts rather than synthetics, unless it is very clear how the values obtained from the synthetics may be interpreted.“ Im Zuge der Erstellung spezifischer Qualitätsmodelle können immer noch komplexe Metriken auf der Basis der einfachen Metriken eingeführt und validiert werden.

Die ausgewählten Metriken sollen möglichst viele Aspekte des Kriteriums abdecken. Gleichzeitig sollen sie aber auch möglichst voneinander unabhängig sein. Das bedeutet, dass derselbe Sachverhalt nicht mehrfach gemessen werden soll, weil er sonst unbeabsichtigt ein zu hohes Gewicht bekommt. Beispielsweise sind die Gesamtzahl der Attribute einer Klasse und die Anzahl der privaten Attribute so stark korreliert, dass nur eines von beiden verwendet werden sollte. Häufig kann eine Metrik auch zur Messung verschiedener Kriterien herangezogen werden (z. B. ist die Anzahl der Vererbungsbeziehungen sowohl eine Entkopplungs- als auch eine Strukturiertheitsmetrik) – hier ist also Vorsicht geboten (vgl. Abschnitt 9.6.2, Weitere Hinweise).

Akronym	Bedeutung	Ebene	Kriterium
NADC	number of afferent dependencies of a class	Klasse	Entkopplung (-)
NEDC	number of efferent dependencies of a class	Klasse	Entkopplung, Knappheit (-)
NEEC	number of efferent extends relationships of a class	Klasse	Entkopplung, Knappheit (-)
NERC	number of efferent realization relationships of a class	Klasse	Entkopplung, Knappheit (-)
NEUC	number of efferent uses relationships of a class	Klasse	Entkopplung, Knappheit (-)
NEAC	number of efferent association relationships of a class	Klasse	Entkopplung, Knappheit (-)
NACP	number of afferently coupled packages of a package	Paket	Entkopplung (-)
NADP	number of afferent dependencies of a package	Paket	Entkopplung (-)
NECP	number of efferently coupled packages of a package	Paket	Entkopplung (-)
NEDP	number of efferent dependencies of a package	Paket	Entkopplung (-)
NAC	number of attributes of a class	Klasse	Knappheit (-)
NOC	number of operations of a class	Klasse	Knappheit (-)
NCP	number of classes in a package	Paket	Knappheit (-)
NIP	number of interfaces in a package	Paket	Knappheit (-)
NPP	number of packages in a package	Paket	Knappheit, Strukturiertheit (-)
NAS	number of attributes in the system	System	Knappheit (-)
NCS	number of classes in the system	System	Knappheit (-)
NIS	number of interfaces in the system	System	Knappheit (-)
NOS	number of operations in the system	System	Knappheit (-)
NPS	number of packages in the system	System	Knappheit (-)
DITC	depth of inheritance tree of a class	Klasse	Strukturiertheit (-)
$NAEC_l$	number of local afferent extends relationships of a class	Klasse	Strukturiertheit, Entkopplung (-)
$NEEC_l$	number of local efferent extends relationships of a class	Klasse	Strukturiertheit, Entkopplung (-)
DNHP	depth in nesting hierarchy of a package	Paket	Strukturiertheit (-)
DITS	depth of inheritance tree of the system	System	Strukturiertheit (-)
DNHS	depth of nesting hierarchy of the system	System	Strukturiertheit (-)
MNCS	maximum number of child classes in the system	System	Strukturiertheit (-)
MNPS	maximum number of subpackages in the system	System	Strukturiertheit (-)
RTTR	ratio of traceable to total requirements	System	Verfolgbarkeit (+)

Tabelle 9-1: Übersicht der objektiven Metriken

9.2.2 Beschreibung

Auf der Basis der obigen Anforderungen wurden objektive Metriken für die Kriterien der Wartbarkeit ausgewählt und auf der Basis von ODEM formal definiert. Wie in der Literatur üblich, erhält jede Metrik ein drei- oder vierstelliges Akronym (z. B. NAC, Number of Attributes of a Class). Manche Metriken lassen sich verfeinern, indem nach einem bestimmten Aspekt klassifiziert wird. Beispielweise kann die Metrik NAC verfeinert werden, indem die Sichtbarkeitsbereiche public, protected und private unterschieden werden. Der Name einer Verfeinerung ergibt sich aus dem Namen der Ursprungsmetrik und einem Index, der die Art der Verfeinerung bezeichnet. Verfeinerungen sind teilweise kombinierbar, so dass es auch mehrfach indizierte Metriken geben kann. Beispielsweise lassen sich die Verfeinerungen nach Sichtbarkeitsbereich noch einmal verfeinern nach dem Definitionsort (geerbt oder lokal).

In Tabelle 9-1 sind die ausgewählten Metriken aufgelistet; sie sind in Anhang A im Detail beschrieben. In der Tabelle sind zu den Metriken die Ebene (Klasse, Paket oder System) und das Kriterium angegeben, für das die Metrik verwendet wird. Hinter dem Kriterium wird durch (-) angezeigt, dass die Metrik negativ mit dem Kriterium korreliert ist, (+) bedeutet positive Korrelation. Ist eine Metrik (z. B. NEDC) für mehrere Kriterien relevant, wird das Kriterium zuerst genannt, für das die Metrik am wichtigsten ist. Alle Metriken haben eine Absolutskala – mit Ausnahme von RTTR, das eine Rationalskala hat.

Es fällt auf, dass nicht für alle Kriterien und Ebenen objektive Metriken angegeben werden können, sondern sich die Metriken vor allem auf die Kriterien Knappheit, Strukturiertheit und Entkopplung beschränken. Das liegt daran, dass diese Kriterien sich gut an syntaktischen Aspekten der Entwurfsbeschreibung festmachen lassen. Bei den anderen Kriterien überwiegen die semantischen Aspekte, die sich schlecht durch objektive Metriken erfassen lassen.

9.2.3 Beispiel

Als Beispiel werden hier die wichtigsten objektiven Metriken für das Kriterium Knappheit und ihre formale Definition gezeigt. Eine ausführlichere Darstellung findet sich in Abschnitt A.1. Da Knappheit unter anderem eine geringe Größe bedeutet, werden vor allem die Bestandteile der Entwurfselemente gezählt. Diese Zählmetriken sind negativ mit der Knappheit korreliert.

Klasse/Interface

Bei Klassen und Interfaces werden geerbte Bestandteile mitgezählt, da geerbte Eigenschaften in der Klasse vorhanden sind und damit ihre Größe mitbestimmen (siehe dazu auch Abschnitt A.1).

NAC (number of attributes of a class)
$NAC(c) = |\{a \in A: \mathit{has}^*(c,a)\}|$

NOC (number of operations of a class)
$NOC(c) = |\{o \in O: \mathit{has}^*(c,o)\}|$

NEDC (number of efferent dependencies of a class)
$NEDC(c) = \Sigma_{d \in C \cup I}\ \mathit{depends\_on}^*(c,d).\mathit{weight}$

Paket

NCP (number of classes in a package)
NCP(p) = $|\{c \in C: \textit{contains}(p,c)\}|$

NIP (number of interfaces in a package)
NIP(p) = $|\{i \in I: \textit{contains}(p,i)\}|$

NPP (number of packages in a package)
NPP(p) = $|\{q \in P: \textit{contains}(p,q)\}|$

System

NAS (number of attributes in the system)
NAS(S) = $|A|$

NOS (number of operations in the system)
NOS(S) = $|O|$

NCS (number of classes in the system)
NCS(S) = $|C|$

NIS (number of interfaces in the system)
NIS(S) = $|I|$

NPS (number of packages in the system)
NPS(S) = $|P| - 1$ (da S in P enthalten ist, ist 1 abzuziehen)

Verfeinerungen

Als Beispiel einer Verfeinerung wird hier die Verfeinerung der Metrik NCP gezeigt, die sich ergibt, wenn abstrakte und konkrete Klassen unterschieden werden:

NCP_a (number of abstract classes in a package)
$NCP_a(p) = |\{c \in C: \textit{contains}(p,c) \wedge c.\textit{isAbstract}\}|$

NCP_c (number of concrete classes in a package)
$NCP_c(p) = |\{c \in C: \textit{contains}(p,c) \wedge \neg c.\textit{isAbstract}\}|$

9.2.4 Auswertung

Aus den objektiven Metriken kann ein Bewertungsvorschlag für die zugehörige subjektive Metrik gewonnen werden. Dazu werden die Metriken eines Kriteriums einer Ebene (z. B. Knappheit Paket) aggregiert und das Ergebnis auf den Wertebereich der subjektiven Metriken abgebildet.

Verfahren

Die Berechnung eines Bewertungsvorschlags verläuft in drei Schritten:

1. Normierung der Metriken, um aus den Messwerten Qualitätsaussagen abzuleiten,
2. Gewichtung der normierten Metriken, um die relative Bedeutung der Qualitätsaussagen zu berücksichtigen, und
3. Transformation des Resultats auf den Wertebereich der subjektiven Metriken.

Normierung. Jede Metrik wird auf den Wertebereich [0;1] normiert. Ein höherer normierter Wert steht dabei für eine höhere Qualität. Daher ist bei der Normierung darauf zu achten, ob die Metrik positiv oder negativ mit dem Kriterium korreliert ist. Die Normierung beruht auf einem Schwellenwert S mit einer Toleranz T.

Die Auswertung von Metriken durch Schwellenwerte ist die einfachste Art einer Wertung, daher wird in der Literatur davon in hohem Maße Gebrauch gemacht, z. B. bei der Ausreißeranalyse. Eine häufige Kritik bei Schwellenwerten ist aber, dass man oft Argumentationsschwierigkeiten bekommt, warum ein Wert gleich dem Schwellenwert gut, ein Wert leicht über dem Schwellenwert aber schlecht sein soll. Daher wurde hier zusätzlich eine Toleranz eingeführt, mit welcher der abrupte Übergang abgemildert wird, also die eigentlich vorhandene Unschärfe in der Bewertung besser repräsentiert werden kann. Durch die Wahl einer Toleranz von 0 kann allerdings explizit auf diese Erweiterung verzichtet werden.

Bei negativ korrelierten Metriken sollen Werte unterhalb des Schwellenwerts einen hohen normierten Wert ergeben, Werte oberhalb des Schwellenwerts einen niedrigen. Die Toleranz legt die Art des Übergangs zwischen der bestmöglichen und der schlechtestmöglichen Bewertung fest. Ist die Toleranz T größer 0, werden alle Werte kleiner oder gleich S-T auf 1 normiert, alle Werte größer S+T auf 0, und die dazwischen liegenden Werte werden linear interpoliert (vgl. Abbildung 9-3 oben, Fall a).[1] Ist die Toleranz T gleich 0, werden alle Werte kleiner oder gleich dem Schwellenwert S auf 1, alle Werte größer S auf 0 normiert (vgl. Abbildung 9-3 oben, Fall b). Bei positiv korrelierten Metriken wird die Normierung entsprechend entgegengesetzt durchgeführt (vgl. Abbildung 9-3 unten).

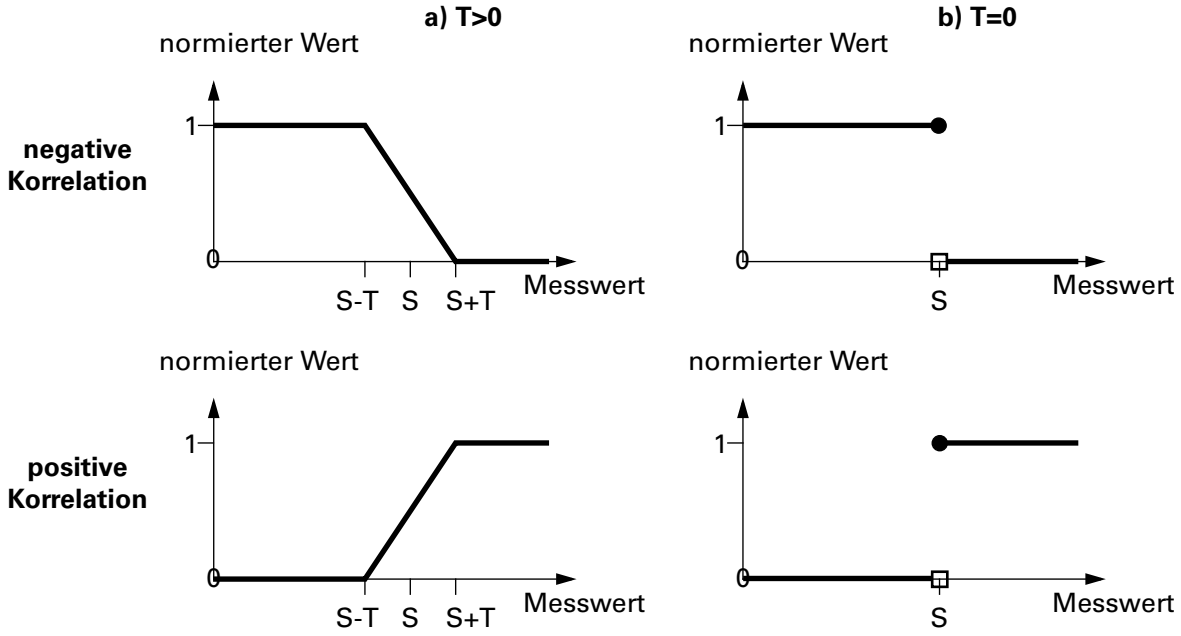

Abbildung 9-3: Normierung einer objektiven Metrik

1. Durch die lineare Interpolation erhält ein Messwert in Höhe des Schwellenwerts die Bewertung 0,5. Ist dies nicht erwünscht, kann durch Verschieben des Schwellenwerts die Auswertung verändert werden. Wenn beispielsweise erreicht werden soll, dass kein Wert oberhalb von S eine positive Bewertung erhält, kann ein neuer Schwellenwert S' = S-T vergeben werden.

Gewichtung. Die normierten Metriken werden mit Gewichten multipliziert und dann aufsummiert. Die Summe wird anschließend durch die Summe der Gewichte dividiert, um so einen gewichteten Durchschnitt zu erhalten. Die Gewichte werden mit dem zweistelligen Kürzel für das Kriterium, das auch für die subjektiven Metriken verwendet wird, benannt (vgl. Tabelle 9-2) und mit dem Namen der Metrik indiziert; beispielsweise ist CC_{NAC} das Gewicht für die Metrik NAC beim Kriterium Knappheit. Der so entstehende Bezeichner ist eindeutig.

Transformation. Der gewichtete Durchschnitt wird nun auf den Wertebereich der subjektiven Metrik (0, 1, …, 9; vgl. Abschnitt 9.3.2) transformiert, indem er mit 9 multipliziert wird. Sinnvollerweise wird das Resultat nicht auf ganze Zahlen gerundet, sondern auf die erste Nachkommastelle, damit der Bewerter besser erkennen kann, welche Tendenz der Vorschlag hat (also z. B. 8,4 oder 7,6 statt 8).

9.3 Subjektive Metriken

Subjektive Metriken spiegeln die Einschätzung eines Qualitätsattributs auf einer Ebene durch einen Bewerter wieder. Die Erhebung der subjektiven Metriken kann nicht automatisch, sondern nur von einem Bewerter vorgenommen werden. Bei der Bewertung kann er sich auf die objektiven Metriken, die Fragebögen und die subjektiven Metriken untergeordneter Ebenen stützen (vgl. Abschnitt 9.5).

9.3.1 Anforderungen

Der Wertebereich der Metrik soll eine möglichst hohe Differenzierung der Bewertung erlauben. Gleichzeitig soll aber die Anzahl der Skalenpunkte möglichst gering sein, um die Skala überschaubar zu halten.

Die Anzahl der Skalenpunkte sollte gerade sein, um einen „neutralen“ Mittelwert zu vermeiden. Dieser wird erfahrungsgemäß besonders dann gewählt, wenn sich der Bewerter nicht entscheiden kann oder will, ob die Bewertung eher positiv oder negativ ausfallen soll. Dies kann aber zu einer Verfälschung des Gesamtbildes führen.

9.3.2 Beschreibung

Das Akronym einer subjektiven Metrik setzt sich aus einem S für „subjective“ und einem zweistelligen Kürzel für das Kriterium zusammen (vgl. Tabelle 9-2). Am Schluss steht ein C für class, P für package oder S für system (analog zu den objektiven Metriken). Beispielsweise steht SSTC für „subjective structuredness of a class“.

Für die Erhebung der Metriken wird eine Intervallskala mit zehn Werten eingesetzt, die einen Bereich von 0 = sehr schlecht bis 9 = sehr gut abdecken. Die Wahl des Wertebereichs ist ein Kompromiss zwischen den Anforderungen einer möglichst großen Möglichkeit zur Differenzierung und einer trotzdem noch überschaubaren Anzahl an Skalenpunkten. Bewertungsskalen mit 10 Punkten sind außerdem relativ häufig; Bewerter sind also gewohnt damit umzugehen.

Kürzel	Kriterium
CC (conciseness)	Knappheit
CO (cohesion)	Zusammenhalt
CS (consistency)	Einheitlichkeit
DC (decoupling)	Entkopplung
DO (documentation)	Dokumentierung
MA (maintainability)	Wartbarkeit
ST (structuredness)	Strukturiertheit
TR (traceability)	Verfolgbarkeit

Tabelle 9-2: Kürzel für die Kriterien und Faktoren

9.3.3 Beispiel

Die vollständige Definition von SSTC sieht wie folgt aus:

SSTC (subjective structuredness of a class)
SSTC(c) = Beurteilen Sie die Strukturiertheit der Klasse c auf der folgenden Skala:
0 = sehr schlecht, 1, 2, 3, 4, 5, 6, 7, 8, 9 = sehr gut.

9.3.4 Auswertung

Im Gegensatz zu den objektiven Metriken oder den Fragebögen machen die subjektiven Metriken eine direkte Aussage zur Qualität, müssen also nicht mehr auswertet werden. Allerdings werden sie für Bewertungsvorschläge für subjektive Metriken der übergeordneten Ebenen (Paket und System), der Faktoren und der Gesamtqualität verwendet.

Für die Ebenen Paket bzw. System liegen subjektive Metriken für das jeweilige Kriterium der untergeordneten Ebene Klasse/Interface bzw. Paket vor. Aus diesen kann ein Bewertungsvorschlag gewonnen werden, indem ein auf eine Nachkommastelle gerundeter Durchschnitt der subjektiven Metriken der untergeordneten Ebene berechnet wird. Beispielsweise kann ein Vorschlag für SCCP(p) gewonnen werden, indem der gerundete Durchschnitt von SCCC für alle Klassen im Paket p gebildet wird.

Der Bewertungsvorschlag für die subjektive Metrik eines Faktors wird aus den subjektiven Metriken der Kriterien auf der *gleichen* Ebene gewonnen. Die Kriterien werden dazu mit Gewichten versehen, um einen gewichteten Durchschnitt berechnen zu können. Beispielsweise kann für die subjektive Metrik SMAC(k) für den Faktor Wartbarkeit einer Klasse k ein Bewertungsvorschlag aus den Metriken SCCC(k), SSTC(k) etc. gebildet werden. Die Gewichte werden analog zu denen der objektiven Metriken mit dem Kürzel des Faktors bezeichnet und mit der Metrik indiziert (z. B. MA_{SCCC}). Der so entstehende Bezeichner ist eindeutig.

Auf die gleiche Weise wird der Bewertungsvorschlag für die Gesamtqualität aus den subjektiven Metriken der Faktoren der gleichen Ebene gewonnen, indem ein gewichteter Durchschnitt gebildet wird. Die Benennung der Gewichte ist ebenfalls analog, als Kürzel wird DQ (für design quality) verwendet (z. B. DQ_{SMAS})

9.4 Fragebögen

Checklists are the simplest and perhaps the most immediately useful aids to design thinking that have appeared so far.
(Jones, 1992, S. 369)

Fragebögen sind ein wertvolles Hilfsmittel bei Analyse und Bewertung von Produkten und Prozessen. Ein Fragebogen besteht aus einer Folge von Fragen, die von einem Bewerter auszufüllen sind. Fragebögen können bei Entwurfsinspektionen eingesetzt werden, um erwünschte Eigenschaften sicherzustellen oder um unerwünschte Eigenschaften festzustellen. Hat der Entwerfer den Fragebogen im Kopf, kann er dessen Inhalt auch bereits während des Entwurfs berücksichtigen.

Eine spezielle Form des Fragebogens ist die Checkliste, eines der sieben Werkzeuge zur Qualitätsüberwachung von Ishikawa (1989). Eine Checkliste besteht nur aus Fragen, die mit ja oder nein zu beantworten sind. Bei klassischen Checklisten (z. B. bei Flugzeugen vor dem Start) müssen alle Fragen mit ja beantwortet werden, damit der Bewertungsgegenstand die Prüfung besteht. In der Literatur werden unter der Bezeichnung Checklisten aber auch Fragenlisten publiziert, die den üblichen Kriterien für Checklisten nicht entsprechen, z. B. können nicht alle Fragen eindeutig mit ja oder nein beantwortet werden (Würthele, 1995). Als Beispiel folgt ein Auszug aus einer Fragenliste für den objektorientierten Entwurf von Page-Jones (1995, S. 325 ff.), die vom Autor als Checkliste bezeichnet wird.

7. Does the class rely on any assumptions in order to work correctly? Do any other classes also rely on the same assumptions? How likely are those assumptions to change? Where are the assumptions documented?

13. Is the class's invariant documented?

15. Does each of the class's methods have a documented pre- and postcondition?

22. Do subclasses of a common superclass contain similar or identical features that should be moved to the superclass?

40. Is the class too restrictive for its current purpose, that is, the applications in which it's likely to be used?

41. Is the class too general or broad for its current purpose? In other words, does the class contain a lot of "unnecessary baggage" based on fantasy rather than firm requirements?

46. Does the design fulfill the spec, the whole spec, and nothing but the spec?

48. What are the most likely changes that the user will make to the system requirements? How much impact would each one make on the design? Would it cost a great deal to carry out any of the more minor changes?

51. Can the design actually be coded? Will it work?

Abbildung 9-4: Checkliste von Page-Jones (Ausschnitt)

Die Fragen können fast alle mit ja oder nein beantwortet werden – doch bis man zu einer Antwort gelangt, kann wenig Aufwand (z. B. Frage 13) oder viel Aufwand (z. B. Frage 46) notwendig sein. Die Antwort auf Frage 51 schließlich kann letztendlich nur eine Implementierung liefern.

Bei dem gezeigten Beispiel handelt es sich nicht um eine klassische Checkliste, denn die Punkte 7. und 48. enthalten Fragen, die nicht mit ja oder nein beantwortbar sind. Dennoch können diese Fragen wertvolle Hinweise auf mögliche Probleme im Entwurf liefern – z. B. weist die Frage nach den wahrscheinlichsten Änderungen und

ihren Konsequenzen im Entwurf auf Probleme bei der adaptiven Wartung hin. Allerdings ist gerade diese Frage kaum auswertbar, weil kein Hinweis darauf gegeben wird, wie die Antwort als Aussage zur Qualität zu interpretieren ist.

In dieser Arbeit wird zur besseren Unterscheidung anstelle des Begriffs Checkliste der allgemeinere Begriff Fragebogen verwendet, wenn es sich nicht um eine klassische Checkliste handelt.

9.4.1 Anforderungen

Nach Würthele (1995, S. 60ff.) soll eine gute Checkliste (und damit auch ein Fragebogen) die folgenden Eigenschaften besitzen:

- kurz, um nicht von der Verwendung abzuschrecken (Gilb, 1988, empfiehlt aufgrund praktischer Erfahrung, nur Checklisten zu verwenden, die auf eine Seite passen, also maximal 25 Fragen enthalten),
- selbsterklärungsfähig, um ohne zusätzliche Dokumentation verwendbar zu sein,
- neutral, um eine Beeinflussung der Antwort durch die Fragestellung auszuschließen, sowie
- verständlich und präzise formuliert, um Missverständnisse und Interpretationsspielräume zu vermeiden.

Spezielle Anforderungen an die Fragebögen sind hier:

- relevant, d. h. es besteht ein positiver oder negativer Zusammenhang der Fragen mit dem Kriterium,
- entscheidbar, d. h. die Fragen sind eindeutig zu beantworten (optimal sind in dieser Hinsicht eindeutige Ja/Nein-Fragen),
- auswertbar, d. h. aus den Antworten lässt sich eine Bewertung ableiten,
- hinreichend, d. h. die Fragen liefern eine größtmögliche Überdeckung des Kriteriums, und
- redundanzfrei, d. h. die Fragen überlappen sich inhaltlich nicht.

9.4.2 Beschreibung

Ein Fragebogen besteht aus einer Liste von Fragen. Jede Frage setzt sich hier aus fünf Teilen zusammen:

Bedingung. Gibt an, ob die Frage im aktuellen Kontext angewendet werden kann. Die meisten Fragen sind zwar immer anwendbar, es gibt aber z. B. auf der Ebene Klasse/Interface Fragen, die nur bei Interfaces sinnvoll sind. Die Bedingung kann unter Verwendung der Elemente von ODEM und der objektiven Metriken als Prädikat formuliert werden. Auf diese Weise kann die Anwendbarkeit der Frage automatisch entschieden werden.

Fragetext. Die eigentliche Fragestellung, ggf. mit Kommentar zur Präzisierung der Fragestellung.

Antwortskala. Die Skala der möglichen Antworten. Im einfachsten Fall handelt es sich um eine Ordinalskala mit den Werten nein und ja (binäre Frage). Es gibt aber

auch Fragen, für die es schwierig ist, eindeutige Ja/Nein-Antworten zu geben, weil eine graduelle Aussage wie „zu 70% ja“ zutreffender ist als ein absolutes Ja. Daher ist es sinnvoll, dem Bewerter Zwischenwerte zwischen ja und nein anzubieten. Dazu wird hier die Rationalskala mit dem Wertebereich [0;1] verwendet. Der Wert 0 wird mit nein assoziiert, der Wert 1 mit ja. Bei binären Fragen werden die Werte 0 und 1 vergeben, bei graduellen Fragen können auch Zwischenwerte aus [0;1] gewählt werden.[2] Die Fragen werden so formuliert, dass ein Ja positiv für die Qualität ist. Daraus folgt, dass höhere Werte bei einer graduellen Frage eine höhere Qualität bedeuten.

Gewicht. Gibt an, wie wichtig die Frage für die Bewertung ist. Es wird hier mit den generischen Werten weniger wichtig, wichtig und sehr wichtig gearbeitet, die in einem spezifischen Modell durch konkrete Werte zu ersetzen sind. Fragen, die wichtige erwünschte Eigenschaften des Entwurfs sicherstellen oder gravierende Fehler aufdecken sollen, erhalten das höchste Gewicht (sehr wichtig), darunter wird abgestuft zwischen normaler Wichtigkeit (wichtig) und nicht notwendigerweise erforderlichen Eigenschaften (weniger wichtig).

Automatisierbarkeit. Gibt an, ob die Frage automatisch beantwortet werden kann, d. h. ob unter Verwendung der Elemente von ODEM und der objektiven Metriken ein Prädikat formuliert werden kann, das der Fragestellung entspricht.

Wird eine Frage negativ beantwortet, sollte vom Bewerter zusätzlich notiert werden, was der Grund für die negative Antwort war. Die so entstehende Mängelliste ist für eine anschließende Überarbeitung des Entwurfs aufgrund der Bewertung sehr nützlich.

Mögliche Erweiterungen

Manche Fragen bedürfen eigentlich eines ausführlichen Kommentars, insbesondere für Entwurfsanfänger. Diese Kommentare geben Erläuterungen zu den Fragen und deuten auf mögliche Sonderfälle hin. Aus Platzgründen ist es nicht sinnvoll, diese Kommentare auf dem Fragebogen abzudrucken. Allerdings kann ein zusätzliches Dokument angeboten werden, das diese Kommentare enthält.

Bei einer Bewertung, die der Schwachstellenanalyse dient, ist es außerdem hilfreich, wenn die Fragen, die Probleme aufdecken sollen, mit Empfehlungen versehen sind, wie die Probleme behoben werden können. Da es meistens mehrere Möglichkeiten gibt, ist es ebenfalls nicht sinnvoll, diese Empfehlungen auf dem Fragebogen abzudrucken. Stattdessen können sie in das Kommentardokument aufgenommen werden. Sofern der Fragebogen in Form eines Hypertext-Dokuments vorliegt, können Kommentare und Empfehlungen zur Problembehebung durch Hyperlinks mit den einzelnen Fragen verknüpft werden (siehe auch Abschnitt 11.2.2, Review-Bögen).

9.4.3 Beispiel

Die Gesamtheit der Fragebögen wird in Anhang B vorgestellt. Das Aussehen eines Fragebogens wird hier am Beispiel des Kriteriums Knappheit auf der Ebene Klasse/

2. Graduelle Antworten erlauben es, zutreffendere Antworten zu geben, führen auf der anderen Seite aber auch zu einer geringeren Reproduzierbarkeit der Bewertung. Daher kann man in einem spezifischen Modell für graduelle Fragen den Wertebereich der Skala einschränken, im Extremfall sogar eine binäre Antwort erzwingen.

Interface verdeutlicht (vgl. Abbildung 9-5). Da Knappheit geringe Größe bedeutet, enthält der Fragebogen Fragen nach unnötigen und redundanten Entwurfsteilen.

Die Bedingungen der Fragen werden durch Prädikate formalisiert. Die Prädikate können einen impliziten Parameter *this* verwenden, der den aktuellen Bewertungsgegenstand bezeichnet. Die Gewichte weniger wichtig, wichtig und sehr wichtig werden durch Sternchen visualisiert (* für weniger wichtig, ** für wichtig und *** für sehr wichtig). Ist eine Frage automatisch beantwortbar, wird in der letzten Spalte ein Häkchen gesetzt.

Bedingung	Fragetext	Antwortskala	Gewicht	auto.
–	Ist das Vorhandensein der Klasse notwendig?	0 = nein, 1 = ja	***	
this∈C	Enthält die Klasse nur die nötigen Attribute? (z. B. keine nicht (mehr) verwendeten oder für die Verantwortlichkeiten der Klasse nicht relevanten)	0 = nein, 1 = ja	**	
NOC(this) > 0	Enthält die Klasse nur die nötigen Operationen? (z. B. keine nicht (mehr) verwendeten oder für die Verantwortlichkeiten der Klasse nicht relevanten)	0 = nein, 1 = ja	**	
NOC(this) > 0	Enthält die Klasse keine überflüssigen Operationen? (z. B. überladene Operationen oder andere „Komfort-Operationen“)	0 = nein, 1 = ja	*	
NOC(this) > 0	Gibt es keine ähnlichen Operationen in anderen Klassen? Wird die Implementierung vermutlich keinen redundanten Code enthalten?	0 = nein, 1 = ja (Aussage trifft zu)	**	
NOC(this) > 0	Benötigt jede Operationen alle ihre Parameter?	0 = nein, 1 = ja	**	
NEEC (this) > 0	Fügt die Unterklasse neue Attribute oder Operationen hinzu?	0 = nein, 1 = ja	**	✔
this∈C ∧ this.isAbstract	Hat die abstrakte Klasse mindestens eine Unterklasse?	0 = nein, 1 = ja	***	✔
this∈I	Wird das Interface realisiert oder von anderen Interfaces geerbt?	0 = nein, 1 = ja	***	✔

Abbildung 9-5: Fragebogen für Kriterium Knappheit, Ebene Klasse/Interface

9.4.4 Auswertung

Bei den Fragebögen lässt sich aus den Antworten ein Bewertungsvorschlag generieren. Das Verfahren dazu ähnelt dem zur Berechnung eines Bewertungsvorschlags aus objektiven Metriken. Eine Normierung ist hier allerdings nicht erforderlich, weil die Fragen so formuliert sind, dass eine Antwort mit ja (oder eine hohe Zustimmung) positiv für die Qualität ist, und weil der Wertebereich für die Antwort bereits [0;1] ist.

Ausgewertet werden nur anwendbare Fragen, d. h. Fragen, deren Bedingung erfüllt ist. Nicht anwendbare Fragen wirken sich somit nicht auf den Bewertungsvorschlag aus. Zur Auswertung wird die Antwort auf jede anwendbare Frage mit dem Gewicht multipliziert und das Resultat für alle Fragen aufsummiert. Dann wird durch Summe

der Gewichte aller anwendbaren Fragen geteilt. Zur Umrechnung auf den Wertebereich der subjektiven Metriken wird dann mit 9 multipliziert und das Ergebnis (wie bei den objektiven Metriken) auf eine Nachkommastelle gerundet.

9.5 Gesamtbewertung

9.5.1 Gesamtvorschlag

Pro Kriterium und Ebene können für die zugehörige subjektive Metrik bis zu drei verschiedene Bewertungsvorschläge berechnet werden:

1. aus den objektiven Metriken (vgl. Abschnitt 9.2.4),
2. aus untergeordneten subjektiven Metriken (vgl. Abschnitt 9.3.4) und
3. aus dem Fragebogen (vgl. Abschnitt 9.4.4).

Je nach Kriterium und Ebene liegen dann ein bis drei Vorschläge vor. Aus diesen Bewertungsvorschlägen kann ein Gesamtvorschlag berechnet werden, indem die Vorschläge mit Gewichten versehen werden und dann ein gewichteter Durchschnitt gebildet wird (auf eine Nachkommastelle gerundet).

Der Bewerter kann nun auf der Basis der einzelnen Bewertungsvorschläge und des Gesamtbewertungsvorschlags einen eigenen Wert für die subjektive Metrik festlegen. Dabei wird er seine eigene Einschätzung mit einbringen, weshalb er von dem Vorschlag nach oben oder unten abweichen kann.

9.5.2 Automatisierung

Das vorgestellte Bewertungsverfahren ist weitgehend automatisierbar. Dazu kann an den folgenden Stellen angesetzt werden:

1. Die Erhebung der objektiven Metriken kann der Rechner übernehmen, ebenso ihre Auswertung (d. h. die Berechnung der Bewertungsvorschläge).
2. Wenn die vergebenen Werte für die subjektiven Metriken erfasst werden, können die Bewertungsvorschläge für die übergeordneten subjektiven Metriken automatisch berechnet werden.
3. Bei den Fragebögen können automatisch die Fragen aussortiert werden, deren Bedingungen nicht erfüllt sind, so dass dem Bewerter nur anwendbare Fragen präsentiert werden. Wenn die Antworten des Bewerters erfasst werden, können die Fragebögen anschließend automatisch ausgewertet werden. Einige Fragen lassen sich sogar automatisch beantworten, so dass diese vom Bewerter nicht selbst beantwortet werden müssen.
4. Liegen alle Bewertungsvorschläge vor (z. B. wenn sie automatisch berechnet wurden), kann auch der Gesamtbewertungsvorschlag automatisch berechnet werden.

Wird das gesamte aufgezeigte Automatisierungspotential genutzt, kann der Aufwand für den Bewerter auf ein unbedingt erforderliches Maß reduziert werden.

Um die Bewertung *vollautomatisch* durchführen zu können, muss bei den Fragebögen auf alle Fragen verzichtet, die nicht automatisch beantwortet werden können. Außer-

dem wird die Erhebung einer subjektiven Metrik durch die Verwendung des (gerundeten) Gesamtvorschlags ersetzt. Auf diese Weise kann im Bewertungsverfahren auf den Bewerter verzichtet werden. Allerdings gehen dadurch alle Aspekte verloren, die der Bewerter bei der Erhebung der subjektiven Metrik zusätzlich hätte einfließen lassen, z. B. eigene Erfahrung.

Außerdem fallen durch den notwendigen Verzicht auf nicht-automatisierbare Fragen Kriterien wie Einheitlichkeit und Dokumentierung gänzlich unter den Tisch, weil es dort weder objektive Metriken noch automatisch beantwortbare Fragen gibt. Bei anderen Kriterien wie Zusammenhalt gehen durch den Verzicht sehr viele Aspekte verloren. Eine vollständige Automatisierung geht also auf Kosten der Vielfalt in der Bewertung.

9.6 Ableitung spezifischer Modelle

In diesem Abschnitt wird beschrieben, wie aus QOOD ein spezifisches Qualitätsmodell abgeleitet wird.

9.6.1 Vorgehen

Voraussetzung für die Ableitung eines spezifischen Modells aus QOOD ist eine Anforderungsanalyse, welche die gewünschten Qualitäten und ihre Gewichtung ermittelt. Die Anforderungen stammen aus dem Kontext (z. B. Firmen- oder Projektrichtlinien), den konkreten Qualitätsanforderungen für das System und der eingenommenen Qualitätssicht. Sie bestimmen die Auswahl der Modellbestandteile (Schritte 1., 3., 5. und 7.) und ihre Gewichtung (Schritte 2., 4., 6., 8. und 9.).

1. relevante Faktoren auswählen
2. die Gewichtung der Faktoren festlegen
3. für jeden Faktor die relevanten Kriterien auswählen
4. die Gewichtung der Kriterien festlegen
5. für jedes Kriterium die objektiven Metriken auswählen
6. die Schwellenwerte, Toleranzen und Gewichte für die Metriken festlegen
7. für jedes Kriterium die Fragen aus den Fragebögen auswählen
8. die Gewichte für die Fragen festlegen
9. die Gewichte für die Ableitung des Gesamtvorschlags aus den einzelnen Bewertungsvorschlägen festlegen

Falls eine vollautomatische Bewertung angestrebt wird, sollten nur Metriken und Fragen aus QOOD übernommen werden, deren Berechnung bzw. Beantwortung automatisch möglich ist (vgl. Abschnitt 9.5.2). Dadurch entfallen zwangsläufig alle Kriterien und Faktoren, für die weder objektive Metriken noch automatisch beantwortbare Fragen verfügbar sind.

Das spezifische Modell kann bei Bedarf auch noch um zusätzliche Bestandteile (Faktoren, Kriterien, Metriken und Fragen) erweitert werden; darauf wird hier aber nicht näher eingegangen.

9.6.2 Wahl der Modellparameter

In diesem Abschnitt werden einige Hinweise gegeben, wie die Modellparameter eines spezifischen Modells festgelegt werden können. Der Einfachheit halber wird davon ausgegangen, dass die nicht benötigten Faktoren, Kriterien, Metriken und Fragen von QOOD bei der Spezialisierung nicht in das spezifische Modell übernommen wurden.[3]

Faktoren

Momentan ist in QOOD nur ein Faktor (die Wartbarkeit) quantifiziert, daher ist die Auswahl und Gewichtung trivial. Sind mehrere Faktoren vorhanden, orientiert sich die Gewichtung an den konkreten Qualitätsanforderungen. Aus diesem Grund ist es schwierig, allgemein gültige Regeln für die Wahl der Gewichte anzugeben.

Kriterien

Gleiches gilt für die Wahl der Gewichte der Kriterien für den Bewertungsvorschlag für die subjektive Metrik eines Faktors, da auch hier die konkreten Qualitätsanforderungen die wichtigste Rolle spielen. Aus der Erfahrung heraus lassen sich für die Kriterien des Faktors Wartbarkeit allerdings Faustregeln angeben:

- Knappheit und Entkopplung werden allgemein als die wichtigsten Einflussfaktoren angesehen, also sollten ihre Gewichte relativ hoch sein.
- Zusammenhalt ist das wichtigste semantische Kriterium (im Vergleich zu Dokumentierung und Einheitlichkeit), sollte also ein höheres Gewicht erhalten.

Objektive Metriken

Schwellenwerte. Der Schwellenwert ist eine gerade noch akzeptabler Grenzwert für eine Metrik. Bei positiv korrelierten Metriken ist er eine Untergrenze, bei negativ korrelierten Metriken eine Obergrenze. Vorschläge für allgemein gültige Schwellenwerte sind in der Literatur äußerst selten. Meistens wird nur gesagt, dass die Schwellenwerte projektspezifisch festgelegt werden müssen. In Tabelle 9-3 sind die vorhandenen Vorschläge – übertragen auf die objektiven Metriken aus QOOD – zusammengetragen. Die Schwellenwerte sind eigentlich für Code gedacht, weshalb die Programmiersprache angegeben wird, für die sie ausgelegt sind. Wie man sieht, decken die Vorschläge nur einen kleinen Teil der Metriken ab, so dass bei der Wahl der Schwellenwerte vor allem auf eigene Erfahrung zurückgegriffen werden muss.

Ein alternativer Ansatz zur Festlegung von Schwellenwerten ist statistischer Natur; er stammt ursprünglich aus dem Bereich der Ausreißeranalyse. Zunächst wird die Metrik für alle Elemente eines oder mehrerer Entwürfe erhoben. Dann wird der Schwellenwert aufgrund der Verteilung der Messwerte festgelegt. Bei Metriken auf einer Rationalskala wählt man z. B. als Schwellenwert den Mittelwert plus die Standardabweichung (Vorschlag von Erni, 1996). Dieses Verfahren lässt sich leicht angepasst auch für Metriken mit Absolutskala verwenden. Bei Intervall- oder Ordinalskalen kann man Quantile verwenden (z. B. 80%-Quantil).

3. Die Alternative wäre eine Gewichtung mit 0. Allerdings wäre dann unnötiger Aufwand für die Datenerhebung zu leisten. Eine Gewichtung mit 0 ist daher höchstens bei einem vollautomatisierten Verfahren sinnvoll, um Aufwand für die Anpassung des Bewertungswerkzeugs einzusparen.

Metrik	Schwellenwert	Quelle	Programmiersprache
NAC	5	Morschel (1994)	Smalltalk
	6	Johnson und Foote (1988)	C++
NAC_i	3 (9 für GUI-Klassen)	Lorenz und Kidd (1994)	C++, Smalltalk
NAC_c	3	Lorenz und Kidd (1994)	C++, Smalltalk
NOC	30	Morschel (1994)	Smalltalk
	50	Johnson und Foote (1988)	C++
NOC_i	20 (40 für GUI-Klassen)	Lorenz und Kidd (1994)	C++, Smalltalk
NOC_c	4	Lorenz und Kidd (1994)	C++, Smalltalk
DITC	5 bis 6	Rumbaugh et al. (1993)	–
	6	Lorenz und Kidd (1994)	C++, Smalltalk
$NEEC_l$	1	Lorenz und Kidd (1994)	C++, Smalltalk

Tabelle 9-3: Schwellenwerte aus der Literatur

Toleranzen. Bei einigen Metriken ist klar, dass die Toleranz 0 sein sollte, z. B. bei der Anzahl der lokalen Vererbungsbeziehungen $NEEC_l$, die zur Vermeidung von Mehrfachvererbung den Schwellenwert 1 mit Toleranz 0 erhält. Ansonsten kann mit einer Default-Toleranz von 0 oder aber von einem Prozentsatz des Schwellenwerts (z. B. 20%) gearbeitet werden.

Gewichte. Sofern nicht klar ist, dass bestimmte Metriken eine deutlich höhere Aussagekraft besitzen als andere, kann mit einem Default-Gewicht von 1 gearbeitet werden. Ansonsten können z. B. analog zu den Fragebögen drei Wichtigkeitsklassen mit entsprechenden Gewichten verwendet werden.

Fragebögen

Bei den Fragebögen ist zur Gewichtung nur festzulegen, welche Gewichte für die Kategorien weniger wichtig, wichtig und sehr wichtig vergeben werden. Mögliche Default-Werte für diese Gewichte sind 1 für weniger wichtig, 2 für wichtig und 3 für sehr wichtig.

Gesamtbewertung

Bei der Gewichtung der Bewertungsvorschläge für den Gesamtvorschlag kann mit einer Default-Gewichtung von 1 für jeden Vorschlag gearbeitet werden. Wird ein Vorschlag als wichtiger als die anderen angesehen, sollte sein Gewicht entsprechend erhöht werden. Soll ein Vorschlag ignoriert werden, vergibt man das Gewicht 0.

Weitere Hinweise

Es kann vorkommen, dass Metriken für die Berechnung von Bewertungsvorschlägen für mehrere Kriterien verwendet werden. Bei der Verrechnung der Bewertung von Kriterien miteinander geht dieselbe Metrik dann mehrfach ein, erhält also implizit ein höheres Gewicht. Es sollte daher darauf geachtet werden, dass nicht *unabsichtlich* solche Metriken die Bewertung dominieren. Analoges gilt auch dann, wenn Kriterien in mehrere Faktoren und damit mehrfach in die Bewertung eingehen.

Kapitel 10

Ein spezifisches Qualitätsmodell

For an actual design task, the designer's choices and decisions will need to be resolved solely on the basis of the needs of the particular problem that requires to be solved.
(Budgen, 1994, S. 151)

In diesem Kapitel wird anhand eines Beispiels gezeigt, wie aus dem allgemeinen Qualitätsmodell QOOD ein spezifisches Qualitätsmodell abgeleitet wird. Dieses Qualitätsmodell wird in einer Fallstudie auf zwölf alternative Entwürfe angewendet und validiert. Während der Bewertung aufgefallene Details der Entwürfe werden ebenfalls diskutiert. Abschließend wird die Problematik der Berücksichtigung von Entwurfsmustern bei der Bewertung anhand des Beispiels aus Kapitel 7 behandelt.

10.1 Ableitung des Qualitätsmodells

10.1.1 Bewertungsgegenstand

Bewertungsgegenstand ist ein Fahrplanauskunftssystem, das im Sommersemester 2001 im Rahmen eines Softwarepraktikums im Studiengang Softwaretechnik an der Universität Stuttgart entwickelt wurde. Zwölf Gruppen mit jeweils drei Studierenden (Grundstudium, 4. Semester) lieferten u. a. je eine Spezifikation, einen objektorientierten Entwurf und eine Implementierung in Java ab. Da die zugrunde liegenden Anforderungen dieselben waren, sind die zwölf Entwürfe gut vergleichbar (bis auf Abweichungen in der Gestaltung der Benutzungsoberfläche, die nicht vorgegeben war). Die Aufgabenstellung, eine Aufstellung der Anforderungen und das Begriffslexikon sind in Anhang C abgedruckt.

Das Fahrplaninformationssystem setzt auf einer Datenbasis auf, die Linien, Haltestellen, Abfahrtszeiten an den Endhaltestellen und Fahrzeiten zwischen den Haltestellen umfasst. Das System besitzt zwei Modi, den Fahrgastmodus und den Administratormodus. Im Fahrgastmodus kann der Benutzer Verbindungen suchen und ausdru-

cken. Bei der Verbindungssuche können verschiedene Optimierungsziele angegeben werden, z. B. kürzeste Fahrtzeit oder möglichst wenige Umsteigehaltestellen. Im Administratormodus können die Fahrplandaten verändert werden, z. B. können neue Linien hinzugefügt werden. Als Beispielsdatenbasis stand den Studierenden ein Auszug aus dem Fahrplan des Verkehrsverbunds Stuttgart (VVS) zur Verfügung.

In Tabelle 10-1 sind einige Kennzahlen der zwölf Projekte angegeben (graphisch aufbereitet in Abbildung 10-1):

- Java-Dateien: Anzahl der Java-Dateien der Implementierung. Die Anzahl der Java-Dateien dient zur Abschätzung der Anzahl der implementierten Klassen und Interfaces.
- Codezeilen: Anzahl der Java-Codezeilen der Implementierung. Die Codezeilen enthalten auch Leerzeilen und Kommentare. Da die Gruppen unterschiedlich viel Kommentare im Code haben, kommt es hier zu einer großen Varianz.
- Projektaufwand (in Personenstunden): der gesamte Aufwand des Teams für die Durchführung des Projekts. Beim Projektaufwand ist ein größerer Messfehler möglich, da die Zahlen auf den Angaben der Teilnehmer beruhen und diese teilweise erst nach Ende des Projekts ihren Aufwand geschätzt haben.

Gruppe	Java-Dateien	Codezeilen	Projektaufwand (h)
1	21	6795	486
2	21	4029	331
3	25	5671	577
4	26	4661	455
5	27	8488	693
6	34	7450	498
7	38	5458	333
8	39	7826	510
9	42	8946	485
10	43	6907	610
11	47	9734	447
12	56	8804	581
Durchschnitt	35	7064	501
Minimum	21	4029	331
Median	36	7178	492
Maximum	56	9734	693

Tabelle 10-1: Projektkennzahlen der Gruppen

10.1.2 Faktoren und Kriterien

Um die zwölf Entwurfsalternativen vergleichen zu können, ist es am praktischsten, mittels des speziellen Qualitätsmodells eine einzige Qualitätskennzahl zu berechnen, anhand der eine Rangfolge der Alternativen bestimmt werden kann.

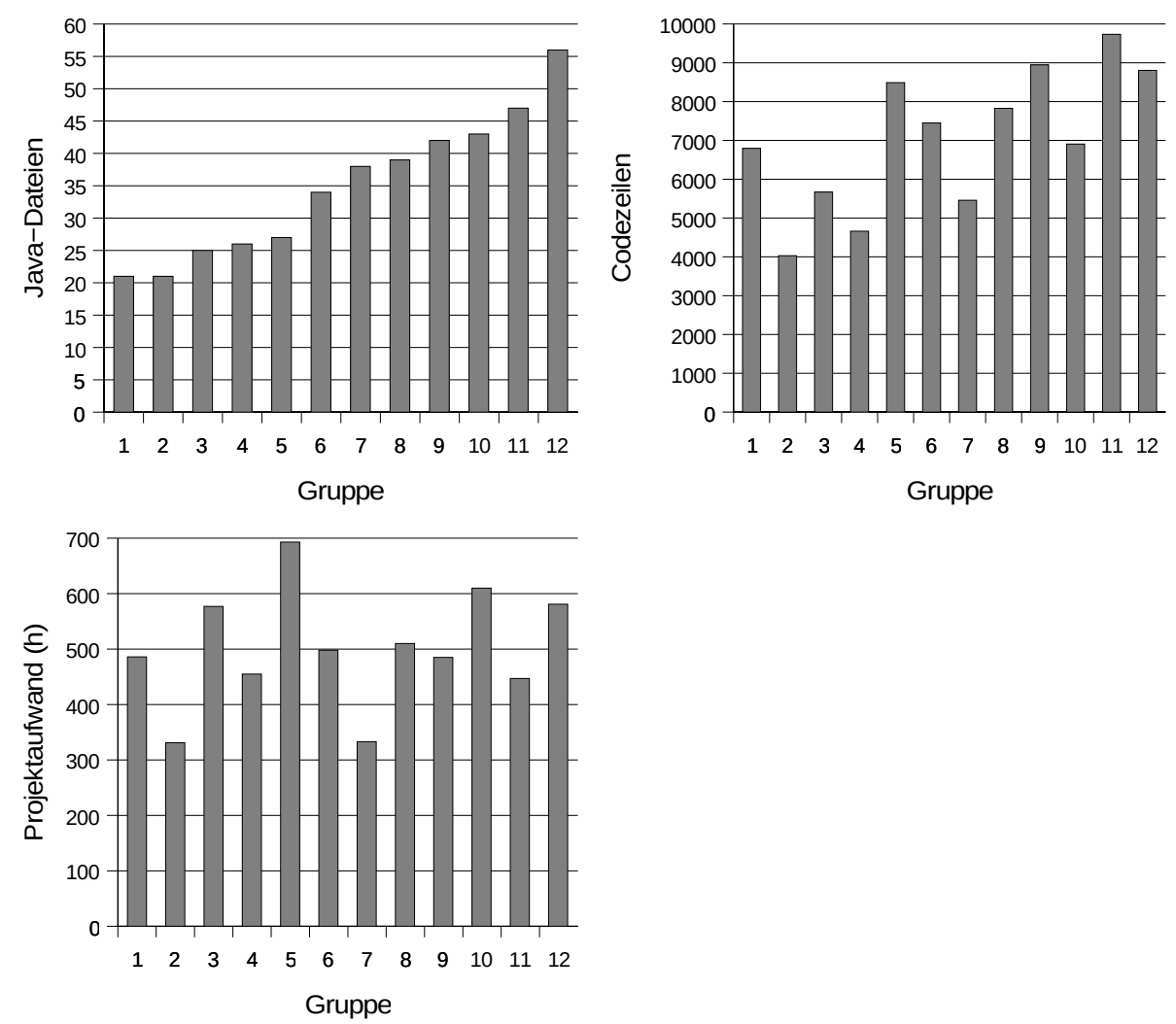

Abbildung 10-1: Projektkennzahlen nach Gruppen

Bei den Qualitätsanforderungen in Abschnitt C.2.8 werden genannt: Bedienbarkeit, Portabilität und Wartbarkeit. Bedienbarkeit gehört nicht zu dem von QOOD betrachteten Qualitätsbereich, daher kann sie hier nicht bewertet werden. Die Portabilität ist bei allen Entwürfen gleich gut, da die Implementierungssprache Java so verwendet wurde, dass keine Abhängigkeiten zur Plattform oder zu anderen Systemen bestehen. Daher konzentriert sich das spezifische Qualitätsmodell auf den Faktor Wartbarkeit. Innerhalb der Wartbarkeit werden bei den Anforderungen keine Schwerpunkte gesetzt. Daher werden alle Kriterien des allgemeinen Modells zur Wartbarkeit unverändert übernommen. Als Perspektive wird die des Entwicklers gewählt.

Bei keinem der Entwürfe ist Information zur Verfolgbarkeit von Anforderungen vorhanden. Allenfalls durch die Benennung der Klassen gibt es implizite Hinweise auf die Anforderungen. Weil die Verfolgbarkeit damit einheitlich mit 0 = sehr schlecht bewertet werden muss, wird sie bei der weiteren Bewertung ausgeblendet.

Gewichte

Die Gewichte für die Kriterien wurden gemäß den in Abschnitt 9.6.2 aufgestellten Faustregeln gewählt und dann durch die Verwendung in ausgewählten Szenarien fein abgestimmt. Die Gewichtung ist in Tabelle 10-2 dargestellt.

Kriterium	Knappheit	Struktu-riertheit	Entkopp-lung	Zusam-menhalt	Einheit-lichkeit	Dokumen-tierung	Verfolg-barkeit
Gewicht	$MA_{SCCx} = 4$	$MA_{SSTx} = 2$	$MA_{SDCx} = 4$	$MA_{SCOx} = 4$	$MA_{SCSx} = 1$	$MA_{SDOx} = 1$	$MA_{STRx} = 0$

Tabelle 10-2: Gewichtung der Kriterien

Da es nur einen Faktor gibt, kann seine Bewertung direkt als Qualitätskennzahl interpretiert werden, weshalb der letzte Aggregationsschritt entfallen kann. Daher werden keine Gewichte für die Faktoren benötigt.

Die Gewichte für die Berechnung eines Gesamtvorschlags aus den Bewertungsvorschlägen für subjektive Metriken auf der Basis von objektiven Metriken, subjektiven Metriken und Fragebögen wurden alle gleich 1 gewählt.

10.1.3 Metriken

Messgegenstand

Da die Entwürfe von unterschiedlichen Gruppen kamen, unterscheiden sie sich in Detaillierung und Vollständigkeit. Das gilt insbesondere für die Entwürfe, die vor der Implementierung abgegeben wurden. Daher werden hier nur die nach der Implementierung überarbeiteten Entwürfe betrachtet. Da die Entwurfsdokumentation trotz der geforderten Überarbeitung nicht immer auf dem neuesten Stand war, wurde sie vor der Bewertung mit der Implementierung abgeglichen.

Um die Entwürfe besser vergleichbar zu machen, wurden die folgenden Konventionen zur Messung eingeführt:

- Wiederverwendete Klassen, hier ausschließlich Klassen aus der Standardbibliothek von Java, werden bei den Knappheitsmetriken nicht mitgezählt. Realisierungs-, Benutzungs- und Vererbungsbeziehungen zu solchen Klassen werden ebenfalls nicht berücksichtigt. Das bedeutet auch, dass von wiederverwendeten Klassen geerbte Attribute und Operationen nicht mitgezählt werden. Assoziationen mit solchen Klassen werden als Attribut gezählt. Die Redefinition einer von einer solchen Klasse geerbten Methode wird als neue Operation gezählt.
- Exception-Klassen werden als Implementierungsdetail aufgefasst und nicht mitgezählt. Einige Entwürfe definieren viele eigene Exceptions, andere verwenden nur die Exceptions der Standardbibliothek. Würden die eigenen Exception-Klassen mitgezählt, würden sonst diejenigen bestraft, die sich um verständlichere Exceptions bemühen, weil z. B. die Knappheitsmetriken schlechtere Werte liefern.
- Debug-Infrastruktur (Klassen, Operationen, Attribute etc.) wird nicht berücksichtigt, weil sie nicht in allen Entwürfen vorkommt, also sonst die Ergebnisse verfälscht würden.

- Häufig sind in den Entwürfen Assoziationen nicht explizit angegeben. Daher werden diese Assoziationen aus den Attributen der Implementierung anhand der Attributtypen abgeleitet. Die zur Implementierung von Assoziationen verwendeten Attribute werden nicht als echte Attribute gezählt.
- Benutzungsbeziehungen sind in den Entwürfen kaum explizit angegeben. Daher werden sie nur in den wichtigsten Fällen anhand der Entwurfsdokumentation und der Implementierung wiederhergestellt.
- Die Namenskonvention für Bezeichner des Entwurfs entspricht der aus den Java Code Conventions von Sun, die für die Implementierung vorgegeben waren (vgl. Abschnitt C.1.5).

Auswahl der Metriken

Da das in Kapitel 11 beschriebene Werkzeug zum Zeitpunkt der Erhebung noch nicht fertig war und auch die oben beschriebenen Zählregeln nicht so leicht automatisch umzusetzen sind, wurden die Daten von Hand erhoben. Um den dadurch hohen Aufwand zu reduzieren, wurden nur wenige Verfeinerungen der Metriken explizit erhoben. Bei der Messung der Entkopplung von Paketen wurde die einfachere Alternative gewählt, nur die gekoppelten Pakete zu zählen, also die Kopplungsstärke in Form der Anzahl der Beziehungen nicht zu berücksichtigen. Tabelle 10-3 zeigt die tatsächlich eingesetzten Metriken zusammen mit Schwellenwert und Toleranz. Sofern verfügbar, wurden die Schwellenwerte aus der Literatur verwendet, wobei diese zum Teil etwas verringert wurden, um durch Toleranzen eine differenziertere Bewertung zu erreichen. Für die übrigen Schwellenwerte wurden zunächst die Metriken für alle Entwürfe erhoben und dann auf der Basis der Ergebnisse geeignete Schwellenwerte und Toleranzen festgelegt.

Kriterium	Ebene	Metriken
Knappheit	Klasse/Interface	NAC (4 ± 2), NOC (20 ± 10)
	Paket	NCP (9 ± 3), NIP (9 ± 3), NPP (6 ± 3)
	System	NAS (100 ± 50), NOS (150 ± 60), NCS (20 ± 10), NIS (20 ± 10), NPS (10 ± 5)
Strukturiertheit	Klasse/Interface	DITC (6 ± 1), $NEEC_I$ (1 ± 0)
	Paket	DNHP (6 ± 1), NPP (6 ± 3)
	System	DITS (4 ± 2), DNHS (4 ± 2), MNCS (6 ± 3), MNPS (6 ± 3)
Entkopplung	Klasse/Interface	NEEC (6 ± 1), NERC (6 ± 2), NEAC (7 ± 4), NEUC (7 ± 4)
	Paket	NECP (4 ± 1), NACP (4 ± 1)

Tabelle 10-3: Verwendete objektive Metriken

Gewichte

Die Gewichte für die objektiven Metriken sind in Tabelle 10-4 dargestellt. Als Default wurde 1 verwendet. Haben die durch die Metrik gemessenen Entwurfseigenschaften einen größeren Einfluss auf die Qualität, wurde ein höheres Gewicht gewählt.

Kriterium	Ebene	Gewichte
Knappheit	Klasse/Interface	$CC_{NAC} = 1$, $CC_{NOC} = 2$
	Paket	$CC_{NCP} = 1$, $CC_{NIP} = 1$, $CC_{NPP} = 1$
	System	$CC_{NAS} = 1$, $CC_{NOS} = 2$, $CC_{NCS} = 3$, $CC_{NIS} = 3$, $CC_{NPS} = 4$
Struktu-riertheit	Klasse/Interface	$ST_{DITC} = 2$, $ST_{NEEC\_I} = 1$
	Paket	$ST_{DNHP} = 2$, $ST_{NPP} = 1$
	System	$ST_{DITS} = 2$, $ST_{DNHS} = 2$, $ST_{MNCS} = 1$, $ST_{MNPS} = 1$
Entkopp-lung	Klasse/Interface	$DC_{NEEC} = 1$, $DC_{NERC} = 1$, $DC_{NEAC} = 1$, $DC_{NEUC} = 1$
	Paket	$DC_{NECP} = 2$, $DC_{NACP} = 1$

Tabelle 10-4: Gewichtung der objektiven Metriken

10.1.4 Fragebögen

Die Fragebögen wurden vollständig in das spezifische Qualitätsmodell übernommen.

Gewichte

Bei den Fragebögen wurde der in Abschnitt 9.6.2 vorgeschlagene Default für die Gewichte verwendet: 1 für weniger wichtig, 2 für wichtig und 3 für sehr wichtig.

10.2 Anwendung des Qualitätsmodells

Das spezifische Qualitätsmodell wurde auf die Entwürfe der zwölf Gruppen angewendet. Die Bewertung eines Entwurfs dauerte – je nach Größe – zwischen zwei und vier Stunden, was aber vor allem an der Metrikenerhebung von Hand lag. Bei einer entsprechenden Werkzeugunterstützung dürfte die Bewertungszeit für die Entwürfe bei ein bis zwei Stunden liegen. Weil begleitende Dokumentation in die Bewertung einbezogen wird, ist deren Qualität ein wichtiger Einflussfaktor auf die Dauer der Bewertung. Von hoher Bedeutung ist z. B. die Verfügbarkeit einer Gesamtsicht des Entwurfs.

Im Folgenden werden die Ergebnisse der Bewertungen präsentiert. Bei der Bewertung sind einige Entwurfsauschnitte positiv und negativ aufgefallen, von denen eine Auswahl vorgestellt wird. Abschließend wird das Modell validiert.

10.2.1 Ergebnisse

Tabelle 10-5 und Tabelle 10-6 zeigen die Messwerte der objektiven und subjektiven Systemmetriken für die Entwürfe der zwölf Gruppen. Die Bewertungen (die subjektiven Metriken) bewegen sich größtenteils im akzeptablen Bereich. Gerade bei der abschließenden Bewertung der Wartbarkeit (SMAS) ist die Streubreite eher gering (5 bis 8). Das liegt auch daran, dass die Gruppen ihre Entwürfe dem Betreuer vorlegen mussten und dieser die schlimmsten Fehler und Mängel (auch in der Dokumentation) beseitigen ließ. Außerdem wurden die Entwürfe implementiert und aufgrund der Erfahrungen bei der Implementierung überarbeitet, so dass zumindest die Realisierbarkeit sichergestellt ist.

Gruppe	NAS	NOS	NCS	NIS	NPS	RTTR
1	142	165	20	0	0	0
2	141	97	21	0	5	0
3	159	217	24	1	7	0
4	79	124	21	0	3	0
5	139	140	19	0	3	0
6	323	168	25	2	5	0
7	148	172	25	0	5	0
8	106	237	31	3	5	0
9	133	259	28	1	4	0
10	126	193	32	0	8	0
11	230	289	34	1	6	0
12	141	323	36	1	6	0
Minimum	79	97	19	0	0	0
Median	141	183	25	0.5	5	0
Maximum	323	323	36	3	8	0

Tabelle 10-5: Objektive Systemmetriken der Entwürfe

Gruppe	SCCS	SSTS	SDCS	SCOS	SCSS	SDOS	STRS	SMAS
1	6	9	5	5	7	7	0	5
2	7	8	6	7	8	7	0	6
3	6	8	6	7	8	8	0	6
4	7	9	7	7	8	9	0	7
5	7	9	5	7	9	8	0	6
6	5	8	6	7	9	8	0	6
7	6	9	7	8	9	8	0	8
8	6	8	8	8	9	9	0	8
9	5	9	7	7	9	8	0	6
10	6	8	7	8	9	9	0	7
11	4	8	7	8	9	8	0	7
12	5	7	6	8	9	5	0	5
Minimum	4	7	5	5	7	5	0	5
Median	6	8	6.5	7	9	8	0	6
Maximum	7	9	8	8	9	9	0	8

Tabelle 10-6: Subjektive Systemmetriken der Entwürfe

10.2.2 Bemerkungen zu den Entwürfen

Vererbung

Bei den Entwürfen fällt auf, dass Vererbung kaum verwendet wird. Das deckt sich auch mit Beobachtungen von Cartwright (1998) aus studentischen Projekten. Wenn Vererbung verwendet wurde, beschränkt sie sich auf eine Stufe. Anhand positiver und negativer Beispiele wird die Nutzung der Vererbung in den Entwürfen illustriert.

Gruppe 8 verwendet das Strategy-Muster (Gamma et al., 1995), um den Algorithmus zur Verbindungssuche je nach Optimierungsziel zu parametrisieren. Dazu dient die abstrakte Klasse Optimization mit ihren drei Unterklassen, welche die verschiedenen Optimierungsziele repräsentieren (vgl. Abbildung 10-2). Wichtigste Operation ist dabei getWeight, welche die Bewertung einer Verbindung gemäß dem Optimierungsziel liefert. Die Unterklassen implementieren die Operation aus der Oberklasse mit der korrekten Bewertungsmethode.

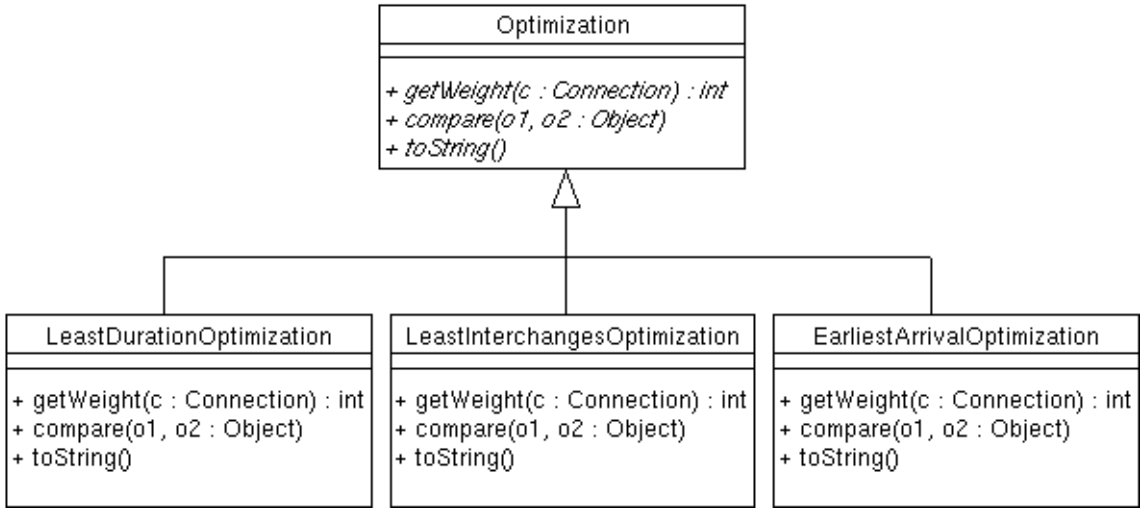

Abbildung 10-2: Strategy-Muster bei Gruppe 8

Eine Alternative dazu ist die Verwendung des Template-Method-Musters (Gamma et al., 1995), wie sie Gruppe 4 vorgenommen hat. Die Bewertung wird der Verbindung selbst zugeordnet. Die abstrakte Klasse Connection besitzt eine Operation getDistance, welche die Bewertung liefert. Diese wird von den konkreten Unterklassen je nach Optimierungsziel implementiert (vgl. Abbildung 10-3). Die Operation processNode (implementiert in Connection) verwendet dann getDistance bei der Verbindungssuche.

Gruppe 12 hingegen benutzt Vererbung eher problematisch. Von einer Klasse TwoDigitNumber, die eine zweistellige Zahl repräsentiert, werden zwei Unterklassen Hour und Minute abgeleitet, ohne dabei Erweiterungen oder Redefinitionen vorzunehmen (vgl. Abbildung 10-4, linke Seite). Das deutet darauf hin, dass die Klassen (abgesehen vielleicht von einem semantischen Unterschied durch den Namen) überflüssig sind. Hour und Minute sind eigentlich Instanzen (und keine Spezialisierungen) von TwoDigitNumber und sollten daher nicht als Klassen modelliert werden. Der Fragebogen zur Strukturiertheit (vgl. Abschnitt B.2) enthält eine entsprechende Frage, die das Problem aufdeckt.

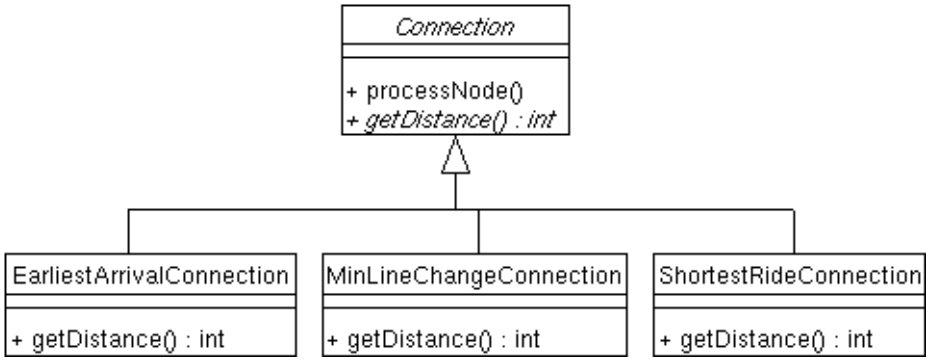

Abbildung 10-3: Template-Method-Muster bei Gruppe 4

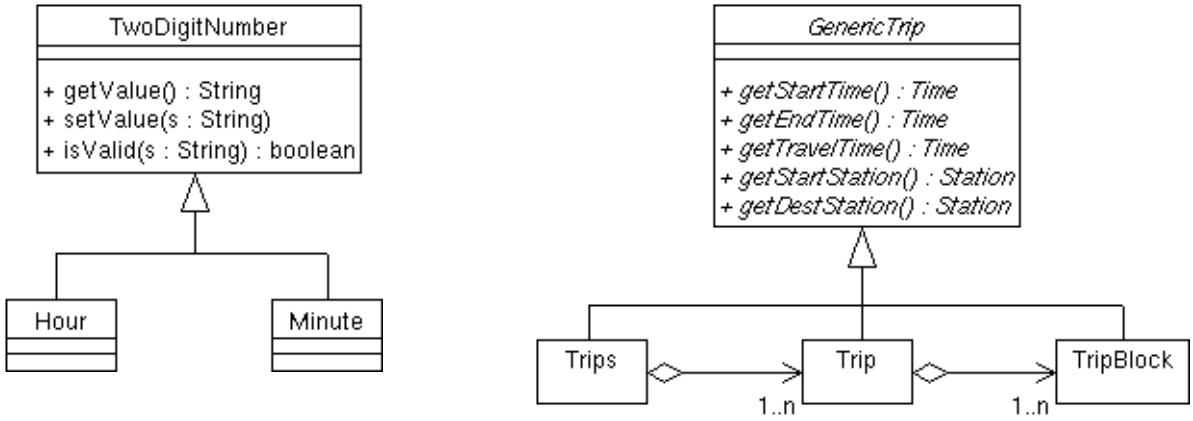

Abbildung 10-4: Fragwürdige Vererbung bei Gruppe 12

Ein weiteres Beispiel für fragwürdige Vererbung stammt von derselben Gruppe (vgl. Abbildung 10-4, rechte Seite). Von einer abstrakten Klasse GenericTrip, die für allgemeine Verbindungen steht, werden drei konkrete Unterklassen abgeleitet: TripBlock (ein Abschnitt einer Verbindung), Trip (eine vollständige Verbindung) und Trips (eine Menge von Verbindungen). Alle implementieren die Operationen der Oberklasse und fügen weitere Operationen hinzu. Außerdem gibt es Aggregationsbeziehungen: Trips aggregiert Trip, Trip wiederum TripBlock.

Betrachtet man die Vererbungsbeziehungen, kann man nur bei Trip von einer Spezialisierung sprechen. Fasst man GenericTrip als Interface auf, kann man auch noch begründen, warum TripBlock dieses Interface implementiert, da auch Verbindungsabschnitte die Attribute einer Verbindung im Sinne von GenericTrip aufweisen. Problematisch wird es allerdings bei Trips, weil dort selbst das Interface nicht passt. Was ist denn z. B. die Startzeit einer Menge von Verbindungen? Ein Blick in die Implementierung verrät, dass immer der Wert für die erste Verbindung in der Verbindungsmenge zurückgegeben wird, doch das scheint eher willkürlich – genauso gut könnte das Minimum oder der Durchschnitt zurückgegeben werden. Eine Spezialisierungsbeziehung zwischen Trips und GenericTrip liegt jedenfalls nicht vor. Übrigens kann auch dieses Problem durch eine entsprechende Frage im Fragebogen zur Entkopplung (vgl. Abschnitt B.3) aufgedeckt werden.

In der Implementierung zeigt sich, dass GenericTrip nie als Typ (z. B. einer Variablen oder eines Parameters) verwendet wird. Der durch die Vererbung ermöglichte Poly-

morphismus wird also gar nicht genutzt. Da von der Klasse auch keine Implementierung vererbt wird, ist sie also gänzlich überflüssig!

Entkopplung durch Interfaces

Ein Beispiel für die Entkopplung durch Interfaces (siehe auch Fowler, 2001b) findet sich im Entwurf von Gruppe 8. Dort gibt es zur zentralen Steuerung der Programmlogik eine Klasse ProcessHandler, die einige untergeordnete Kontrollklassen assoziiert (z. B. DataManager). Diese Klassen benötigen aber Zugriff auf die übergeordnete Kontrollklasse, wodurch ein (unerwünschter) Abhängigkeitszyklus entsteht. Dieser lässt sich aufbrechen, indem ein Interface ProcessManager eingeführt wird, das von ProcessHandler implementiert wird. Die untergeordneten Kontrollklassen assoziieren statt ProcessHandler das Interface ProcessManager (vgl. Abbildung 10-5).

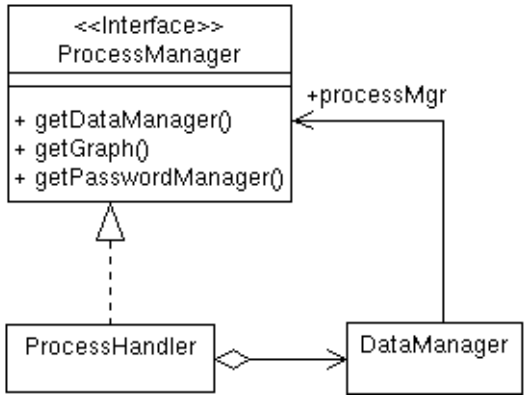

Abbildung 10-5: Entkopplung durch Interface bei Gruppe 8

Zusammenhalt

Ein Beispiel für schlechten Zusammenhalt findet sich bei Gruppe 6. Dort gibt es eine God Class (Riel, 1996) namens Scheduler (vgl. Abbildung 10-6). Eine God Class ist ein typischer Anfängerfehler, nämlich die gesamte Funktion des Systems in einer Klasse zu konzentrieren oder zumindest von dort aus zu steuern. Dies läuft der objektorientierten Vorgehensweise, Funktion zu dezentralisieren, diametral entgegen.

Die Klasse Scheduler verwaltet sowohl die Passwort-Informationen (aus Datei lesen, zugreifen, ändern, in Datei schreiben) als auch die Fahrplaninformation (aus Datei lesen, zugreifen, ändern, in Datei schreiben). Außerdem enthält die Klasse auch noch den Suchalgorithmus für die Verbindungssuche. Insgesamt kommt so die stolze Zahl von 35 Operationen zusammen (13 öffentlich, 22 privat).

Dass hier unterschiedliche Aufgaben in einer Klasse realisiert werden, ist den Entwerfern auch selbst aufgefallen: Sie haben zwei Interfaces vorgesehen, die zur Trennung der Aufgaben nach außen dienen sollen und von der Klasse implementiert werden. Das Interface SearchFrameDataAdapter repräsentiert den Suchalgorithmus. Das Interface AdminDataAdapter allerdings steht für Passwort-Verwaltung *und* Fahrplandatenverwaltung, also sind auch hier bereits Aufgaben verquickt.

Sofern über die Interfaces auf die Klasse zugegriffen wird, ist das Problem des geringen Zusammenhalts nach außen also etwas abgemildert. Besser wäre es allerdings gewesen, die Klasse Scheduler wirklich in drei verschiedene Klassen (oder mehr) aufzuspalten. Ändert sich nämlich bei einer der drei Aufgaben etwas, muss jedes Mal die

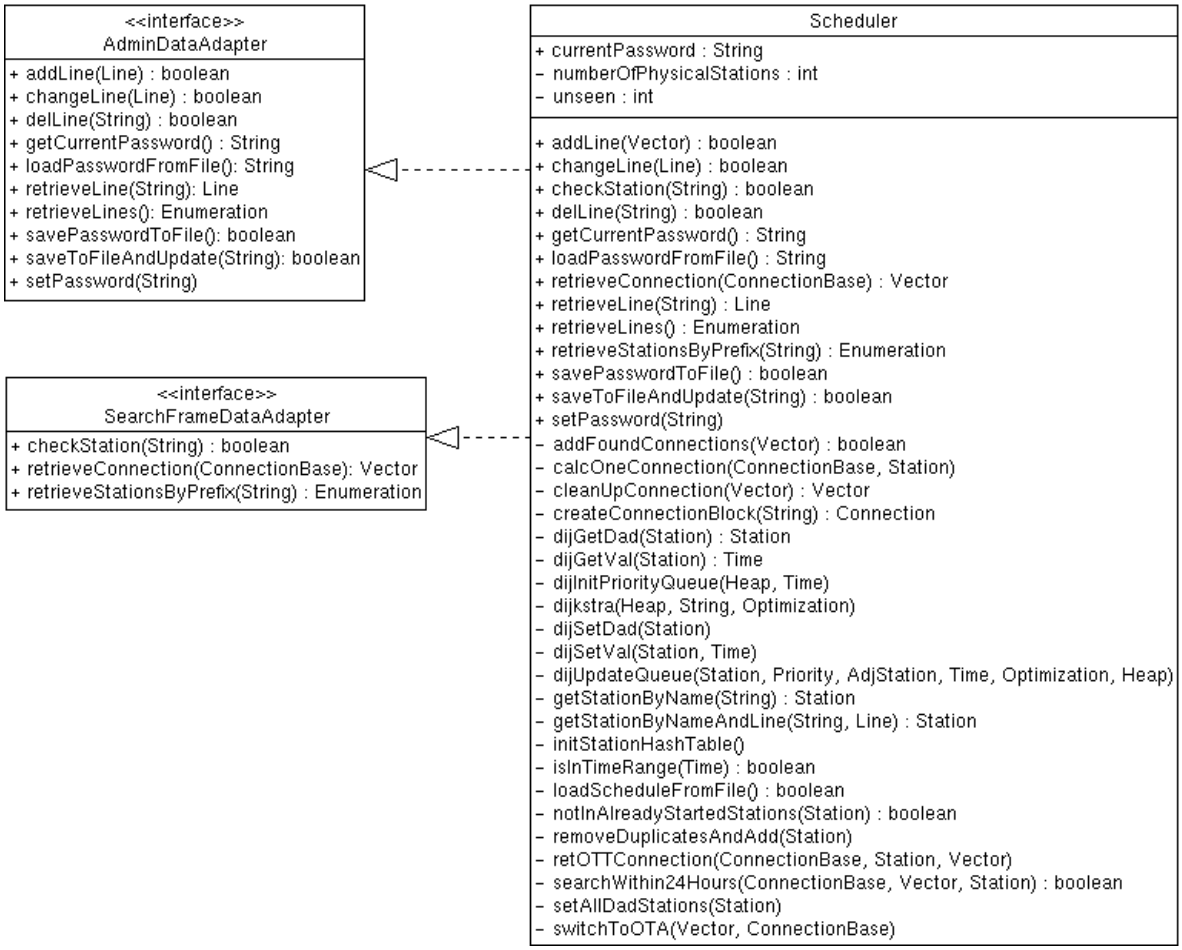

Abbildung 10-6: God Class bei Gruppe 6

Klasse geändert werden, was potentiell alle Benutzer der Klasse betreffen kann. Weil Scheduler quasi die gesamte Funktionalität des Programms realisiert oder steuert, sind das fast alle Klassen im System.

10.2.3 Modellvalidierung

Zur Validierung des Modells wird die Bewertung der Wartbarkeit mit tatsächlichen Wartungsaufwänden der Implementierung verglichen, um die Vorhersagefähigkeit des Modells zu überprüfen. Idealerweise fällt der Wartungsaufwand um so niedriger aus, je besser die Bewertung der Wartbarkeit aufgrund des Entwurfs ist. Da es sich hier um studentische Projekte im Rahmen eines Praktikums handelt, sind die Implementierungen allerdings Wegwerfprodukte. Sie wurden nie gewartet, so dass keine Wartungsdaten vorliegen. Die Alternative, die zwölf Implementierungen durch andere Entwickler warten zu lassen, musste leider wegen des dafür notwendigen Aufwands verworfen werden.

Stattdessen wird hier eine Plausibilitätsprüfung durchgeführt. Für die Entwürfe (und ihre Implementierungen) wird ermittelt, welche Klassen, Attribute und Operationen[1] geändert bzw. hinzugefügt werden müssen, um drei adaptive Wartungsaufgaben

1. Bei den geänderten Operationen werden nicht nur diejenigen mitgezählt, deren Signatur sich ändert, sondern auch die, bei denen die Implementierung (die Methode) geändert werden muss.

durchzuführen. Die Aufgaben sind aus der Liste der wahrscheinlichen Änderungen abgeleitet, die Bestandteil der Spezifikation war (vgl. Abschnitt C.2.8). Dort sind fünf Änderungen vorgesehen. Da zwei dieser Änderungen vor allem Auswirkungen auf die Benutzungsoberfläche haben, für die keine einheitlichen Anforderungen festgelegt sind, konzentriert sich die Untersuchung auf die ersten drei Änderungen.

Die so ermittelten notwendigen Änderungen am Entwurf können zur Abschätzung des Wartungsaufwands verwendet werden. Die Anzahl der Klassen wiegt dabei am schwersten, weil vor der Änderung einer Klasse diese zunächst einmal ausreichend verstanden werden muss, was einen hohen Einarbeitungsaufwand erfordert. Die Anzahl der Operationen wiegt mehr als die Anzahl der Attribute, weil eine Änderung in der Regel aufwendiger ist. Um zu einer quantitativen Schätzung des Wartungsaufwands zu gelangen, wird pro Klasse von 30 Minuten, pro Operation von 15 Minuten und pro Attribut von 5 Minuten Gesamtaufwand ausgegangen.

Anhand der Bewertung der Wartbarkeit wurden der beste (+), der schlechteste (-) und ein mittlerer (o) Entwurf ausgewählt. Diese Entwürfe stammen von den Gruppen 7, 1 und 3, die vom selben Betreuer betreut wurden. Die drei Entwürfe wurden daraufhin untersucht, welche Änderungen für die drei Änderungsszenarien notwendig sind.

Änderung 1: Die vier besten Verbindungen ausgeben

Bisher soll das System nur die beste Verbindung ausgeben (und alle weiteren Verbindungen, die das Optimierungskriterium gleich gut erfüllen). Diese Änderung verlangt nun, dass stattdessen immer die besten vier Verbindungen ausgegeben werden, wie es auch bei der elektronischen Fahrplanauskunft (z. B. bei www.vvs.de) üblich ist.

Von dieser Änderung sind potentiell betroffen:

- die Verbindungssuche, deren Datenhaltung und Rückgabe sowie
- die Verbindungsausgabe auf dem Bildschirm und in die HTML-Datei.

Hier hat der schlecht bewertete Entwurf echte Schwächen, weil er entgegen der ursprünglichen Anforderungen immer nur eine Verbindung als Resultat der Suchanfrage liefert. Daher sind hier mehr Änderungen nötig als bei den anderen Entwürfen, bei denen lediglich der Suchalgorithmus leicht modifiziert werden muss. Tabelle 10-7 zeigt den Wartungsaufwand für diese Änderung bei den drei Entwürfen.

	Klassen	Operationen	Attribute	Aufwand
Gruppe 1 (-)	3	4	1	155
Gruppe 3 (o)	1	1	0	45
Gruppe 7 (+)	1	1	0	45

Tabelle 10-7: Wartungsaufwand für Änderung 1

Änderung 2: Verbindungsanfrage mit gewünschter Umsteigehaltestelle

Zusätzlich zu Start- und Zielbahnhof soll es möglich sein, bei der Verbindungsanfrage noch eine zusätzliche Umsteigehaltestelle anzugeben, über die alle gefundenen Verbindungen gehen müssen.

Von dieser Änderung sind potentiell betroffen:

- die Benutzungsoberfläche zur Verbindungsanfrage sowie
- die Verbindungssuche und deren Aufrufparameter (z. B. Suchdatensatz-Klasse).

Hier schneiden alle Entwürfe etwa gleich gut ab. Der Entwurf von Gruppe 3 besitzt eine Klasse zur Speicherung der Verbindungsanfrage, die auch betroffen ist, weshalb hier eine Klasse mehr geändert werden muss. Die Änderungen sind nicht schwer – bis auf die Änderung der einen Operation zur Ermittlung der Verbindungen. Tabelle 10-8 zeigt den Wartungsaufwand für diese Änderung bei den drei Entwürfen.

	Klassen	Operationen	Attribute	Aufwand
Gruppe 1 (-)	2	2	2	100
Gruppe 3 (o)	3	3	3	150
Gruppe 7 (+)	2	2	4	110

Tabelle 10-8: Wartungsaufwand für Änderung 2

Änderung 3: Unterschiedliche Fahrpläne für Werktage und Wochenende

Bisher ist der Fahrplan für alle Tage gleich. Soll aber zwischen Werktagen und Wochenende unterschieden werden, sind jeweils zwei Datensätze für die Anfahrtszeiten an den Endhaltestellen notwendig. Die Fahrplandatei wird dazu entsprechend erweitert. Außerdem muss die Verbindungsanfrage auch das Datum der gesuchten Verbindung abfragen, um feststellen zu können, welcher Fahrplan gilt.

Von dieser Änderung sind potentiell betroffen:

- das Einlesen und Speichern der Fahrplandatei,
- die internen Datenstrukturen zur Repräsentation des Fahrplans,
- die Verbindungssuche und ihre Datenstrukturen,
- die Benutzungsoberfläche zur Verbindungsanfrage,
- die Benutzungsoberfläche zur Anzeige der Fahrplandaten sowie
- die Benutzungsoberfläche zum Neuanlegen einer Linie und zur Änderung der Abfahrtszeiten.

Die meisten Klassen müssen beim schlechtesten Entwurf geändert werden. Beim besten und beim mittleren Entwurf sind es gleich viele Klassen, doch sind es beim besten Entwurf weniger Attribute und Operationen, die zu ändern sind. Dies liegt vor allem an der Realisierung der Benutzungsoberfläche. Tabelle 10-9 zeigt den Wartungsaufwand für diese Änderung bei den drei Entwürfen.

	Klassen	Operationen	Attribute	Aufwand
Gruppe 1 (-)	6	17	8	475
Gruppe 3 (o)	5	17	17	490
Gruppe 7 (+)	5	10	7	335

Tabelle 10-9: Wartungsaufwand für Änderung 3

Ergebnis

Tabelle 10-10 zeigt den Wartungsaufwand in der Summe über alle drei Änderungen. Der am schlechtesten bewertete Entwurf hat mit 730 Minuten den höchsten, der am besten bewertete Entwurf mit 490 Minuten den geringsten Wartungsaufwand. Der mittlere Entwurf liegt mit 685 Minuten in der Mitte.

	Klassen	Operationen	Attribute	Aufwand
Gruppe 1 (-)	11	23	11	730
Gruppe 3 (o)	9	21	20	685
Gruppe 7 (+)	8	13	11	490

Tabelle 10-10: Wartungsaufwand insgesamt

Damit bestätigt sich in der Gesamtbetrachtung die Rangordnung durch die ursprüngliche Bewertung, obwohl es in den einzelnen Szenarien z. T. leichte Abweichungen davon gibt. Die Bewertung durch das spezifische Modell kann also als plausibel angesehen werden. Leider enthalten die Entwürfe in dieser Fallstudie kaum Vererbung. Daher ist es schwierig festzustellen, ob die Entscheidung, bei den Metriken für Knappheit und Entkopplung geerbte Eigenschaften und Beziehungen grundsätzlich mitzuberücksichtigen, tatsächlich gerechtfertigt ist.

10.3 Besonderheiten bei Mustern

Vergleicht man zwei funktional gleichwertige Entwurfsalternativen, von denen die eine Muster enthält und die andere nicht, stellt man häufig fest, dass die objektiven Metriken den Entwurf mit den Mustern schlechter bewerten (Reißing, 2001b). Das liegt vor allem daran, dass die Musteranwendung mehr Entwurfselemente und Beziehungen erfordert, auch wenn gleichzeitig bestimmte Messwerte verbessert werden (z. B. kann das Mediator-Muster die Entkopplung verbessern, s. a. Huston, 2001).

Dieses Phänomen wird hier an einem Beispiel demonstriert: Das spezifische Qualitätsmodell wird auf das Beispiel aus Abschnitt 7.1 angewendet. Dort wurden drei Entwurfsalternativen für ein Videoverleihsystem vorgestellt.

Tabelle 10-11 und Tabelle 10-13 zeigen die Messwerte für die wichtigsten Klassen- und Systemmetriken der Entwürfe A bis C. Die Paketmetriken werden weggelassen, da es nur ein Paket (das System) gibt. Tabelle 10-12 und Tabelle 10-14 enthalten die subjektiven Metriken für die Kriterien des Faktors Wartbarkeit.

Entwurf A geht aus Entwurf B hervor, indem unnötige Kopplung und dadurch unnötige Operationen entfernt werden. Dies schlägt sich in einem Rückgang der Metriken NOC (bei Rental) und NEDC (bei Customer) nieder. Dadurch verbessern sich die Bewertungen bei der Knappheit (SCCC/SCCS) und der Entkopplung (SDCC/SDCS), wodurch die Wartbarkeit besser bewertet wird (SMAC/SMAS).

Entwurf C geht aus Entwurf B hervor, indem das State-Muster auf die Klasse Movie angewendet wird (für die Preiscodes). Dadurch kommen die abstrakte Klasse Price und ihre drei Unterklassen hinzu. Die Anzahl der Klassen (NCS) nimmt stark zu, wodurch sich die Knappheit (SCCS) verschlechtert. Durch die hinzukommende Ver-

Metrik	NAC			NOC			DITC			NDCC			NEDC		
	A	B	C	A	B	C	A	B	C	A	B	C	A	B	C
Customer	0	0	0	1	1	1	0	0	0	0	0	0	2	1	1
Rental	1	1	1	2	1	1	0	0	0	0	0	0	1	1	1
Movie	1	1	1	2	2	2	0	0	0	0	0	0	0	0	2
Price	-	-	0	-	-	1	-	-	0	-	-	3	-	-	0
RegularPrice	-	-	0	-	-	1	-	-	1	-	-	0	-	-	1
NewReleasePrice	-	-	0	-	-	1	-	-	1	-	-	0	-	-	1
ChildrensPrice	-	-	0	-	-	1	-	-	1	-	-	0	-	-	1

Tabelle 10-11: Objektive Klassenmetriken der Entwürfe A, B und C

Metrik	SCCC			SSTC			SDCC			SCOC			SCSC			SDOC			STRC			SMAC		
	A	B	C	A	B	C	A	B	C	A	B	C	A	B	C	A	B	C	A	B	C	A	B	C
Customer	9	9	9	9	9	9	7	8	8	9	9	9	9	9	9	4	4	4	0	0	0	6	7	7
Rental	8	9	9	9	9	9	8	8	8	9	9	9	9	9	9	4	4	4	0	0	0	7	8	8
Movie	8	8	8	9	9	9	9	9	8	9	9	9	9	9	9	4	4	4	0	0	0	7	7	6
Price	-	-	9	-	-	9	-	-	9	-	-	9	-	-	9	-	-	4	-	-	0	-	-	8
RegularPrice	-	-	9	-	-	8	-	-	8	-	-	9	-	-	9	-	-	4	-	-	0	-	-	7
NewReleasePrice	-	-	9	-	-	8	-	-	8	-	-	9	-	-	9	-	-	4	-	-	0	-	-	7
ChildrensPrice	-	-	9	-	-	8	-	-	8	-	-	9	-	-	9	-	-	4	-	-	0	-	-	7

Tabelle 10-12: Subjektive Klassenmetriken der Entwürfe A, B und C

Metrik	NAS	NOS	NCS	NIS	NPS
Entwurf A	2	5	3	0	0
Entwurf B	2	4	3	0	0
Entwurf C	2	5	7	0	0

Tabelle 10-13: Objektive Systemmetriken der Entwürfe A, B und C

Metrik	SCCS	SSTS	SDCS	SCOS	SCSS	SDOS	STRS	SMAS
Entwurf A	8	9	7	9	9	4	0	6
Entwurf B	8	9	8	9	9	4	0	7
Entwurf C	7	8	8	9	9	4	0	6
Entwurf C (alt.)	8	9	8	9	9	4	0	8

Tabelle 10-14: Subjektive Systemmetriken der Entwürfe A, B und C

erbungshierarchie verschlechtert sich auch die Strukturiertheit (SSTS). Insgesamt verschlechtert sich damit die Wartbarkeit (SMAS). Im Vergleich zu Entwurf B schneidet Entwurf C damit schlechter ab.

Nur eine sehr musterfreundliche Bewertung (letzte Zeile in Tabelle 10-14) kann das verhindern: Die durch den Mustereinsatz hinzukommenden Elemente werden nicht so stark gewichtet, da ihre negative Auswirkung auf die Verständlichkeit durch die Zugehörigkeit zum Muster geringer ist als die Hinzunahme beliebiger Elemente. Dadurch lassen sich bessere Bewertungen bei der Knappheit (SCCS) und der Strukturiertheit (SSTS) rechtfertigen. Schließlich lässt sich argumentieren, dass die Wartbarkeit insgesamt verbessert wird, weil die wahrscheinlichste Änderung das Hinzufügen von Preiscodes und geänderte Berechnungsformeln für die Ausleihgebühren sind. Diese Änderungen sind durch den Mustereinsatz leichter durchzuführen, weil nur noch die Preiscode-Klassen direkt betroffen sind.

Durch dieses Beispiel wird deutlich, dass bei der Bewertung die Verwendung von Mustern speziell berücksichtigt werden muss, um die in der Regel schlechteren Werte der vorhandenen objektiven Metriken zu kompensieren. Daher wäre eine Erweiterung um spezielle objektive Metriken denkbar, welche die Verwendung von Mustern messen. Beispielsweise könnte als Systemmetrik die Anzahl der Musteranwendungen im Entwurf verwendet werden. Eine solche Metrik wäre allerdings ein schlechter Qualitätsindikator, da gerade Entwurfsanfänger dazu neigen, Muster im Übermaß oder falsch zu verwenden, was zu einer hohen Anzahl von Musteranwendungen führt. Mangels geeigneter objektiver Metriken ist man daher auf die Berücksichtigung der Muster in den subjektiven Metriken angewiesen.

Kapitel 11

Werkzeugunterstützung

Human errors can only be avoided if one can avoid the use of humans.
(Parnas, Clements, 1986, S. 251)

In diesem Kapitel werden Werkzeuge zur Entwurfsbewertung vorgestellt. Zunächst wird auf Werkzeuge eingegangen, die in anderen Arbeiten entstanden sind. Dann werden die Werkzeuge präsentiert, die für das vorgestellte Bewertungsverfahren entwickelt wurden. Abschließend wird ein ideales Werkzeug zur Entwurfsbewertung skizziert.

11.1 Werkzeuge aus anderen Arbeiten

In diesem Abschnitt werden bestehende Ansätze und Werkzeuge zur Entwurfsbewertung vorgestellt, unterschieden nach den Artefakten, auf denen die Bewertung durchgeführt wird: Entwurf und Code.

11.1.1 Qualitätsbewertung auf Entwurfsbasis

Baumann (1997). Baumann beschreibt ein Verfahren zur Bewertung von objektorientierten Analysemodellen[1] auf der Basis von fest vorgegebenen Metriken mit variablen Schwellenwerten und Toleranzen. Das Werkzeug MEMOS, eine Erweiterung des MAOOAM-Werkzeugs[2], erhebt die Metriken und teilt sie mittels Schwellenwert und Toleranz in die drei Kategorien gering, mittel und hoch ein. Die derart aufbereiteten Messwerte werden dann zu Teilbewertungen (pro Analyse-Element und Qualitätskriterium) und schließlich zu einer Gesamtbewertung verdichtet. Das Verfahren zur Verdichtung und die Form des Berichts lassen sich über Steuerdateien konfigurieren. Zu den Bewertungen der Kriterien sind im Werkzeug allgemeine Ratschläge zur Verbesserung abrufbar; für die Metriken sind Beschreibungen verfügbar.

1. Der Begriff Analyse schließt bei diesem Ansatz den Entwurf ein!
2. MAOOAM*Tool ist ein Werkzeug für die objektorientierte Systemanalyse (inkl. Entwurf). MAOOAM (= Mannheimer objektorientiertes Analysemodell) ist ein proprietäres Metamodell für objektorientierte Analyse- und Entwurfsmodelle.

Robbins, Redmiles (1999). Das UML-Werkzeug Argo/UML verfügt über eine eingebaute Kritikkomponente (sog. „design critics“; Robbins, 1998). Diese prüft im Hintergrund laufend das UML-Modell auf die Einhaltung bestimmter Regeln. Bei einem Regelverstoß wird das entsprechende Modellelement mit einer Markierung versehen, über die Informationen über den Regelverstoß abgerufen werden können. Es ist auch eine Liste von Regelverstößen für das ganze Modell verfügbar. Zu jedem Befund können Vorschläge zur Verbesserung abgerufen werden. Teilweise werden auch automatische Korrekturen (gesteuert durch Wizards) angeboten. Zusätzlich bietet Argo/UML auch Checklisten an, die typische Probleme aufdecken sollen. Das Besondere dabei ist, dass die Fragen der Checkliste „personalisiert“ werden können, d. h. zum einen werden für den Prüfling irrelevante Fragen weggelassen und zum anderen dessen konkrete Eigenschaften (z. B. Klassennamen oder Attributnamen) in die Checkliste eingesetzt. Durch die Checklisten sollen Bereiche abgedeckt werden, die nicht automatisch (mit design critics) geprüft werden können.

Nenonen et al. (2000). Das Werkzeug MAISA arbeitet auf UML-Modellen. Dabei werden sowohl Klassendiagramme als auch Kollaborations-, Sequenz-, Zustands- und Aktivitätsdiagramme verarbeitet, d. h. es werden statische und dynamische Eigenschaften ausgewertet. Das Werkzeug kann fest eingebaute Metriken erheben; geplant ist auch die Erkennung von Entwurfsmustern und Anti-Mustern. Die Analyse arbeitet dabei auf einem FAMIX[3]-Modell und einer Prolog-Wissensbasis, die durch externe Werkzeuge aus einem UML-Modell generiert wurden. Aus dem Angebot vordefinierter Metriken können die gewünschten ausgewählt sowie obere und untere Schwellenwerte für sie angegeben werden. Messwerte außerhalb der Schwellenwerte werden dann bei der Ausgabe speziell gekennzeichnet.

11.1.2 Qualitätsbewertung auf Codebasis

Ein großer Teil des Entwurfs kann aus dem Code extrahiert werden. Einiges an Entwurfsinformation ist zwar verloren gegangen (z. B. ist die Unterscheidung zwischen Assoziation, Aggregation und Komposition nicht mehr aus dem Code ersichtlich), aber andererseits ist mehr detaillierte Information verfügbar (z. B. Methoden, ihre Aufrufbeziehungen und Zugriffe auf Attribute). Beim Reengineering ist Code häufig das einzig verfügbare Dokument, aus dem Entwurfsinformation gewonnen werden kann. Daher gibt es auch einige codebasierte Ansätze zur Entwurfsbewertung.

Erni (1996). Das Werkzeug dient zur Bewertung der Wartbarkeit von Rahmenwerken mit Hilfe von Metriken. Es arbeitet mit Schwellenwerten, um Ausreißer bei den Messwerten zu identifizieren, wobei auch statistische Schwellenwerte möglich sind (z. B. Mittelwert + Standardabweichung). Besonders an diesem Ansatz ist die Möglichkeit, durch einen Filter (dort Fokus genannt) die Bewertung auf bestimmte Ausschnitte des Entwurfs zu beschränken. Dies kann entweder durch manuelle Auswahl geschehen oder durch Festlegung bestimmter Eigenschaften, welche die zu selektierenden Elemente haben sollen (z. B. abstrakte Klassen). Eine weitere Besonderheit ist die Unterstützung von Trendanalysen, bei denen die Messwerte über verschiedene Versionen des Rahmenwerks betrachtet werden können.

3. FAMIX ist ein Metamodell für Modelle objektorientierte Systeme, das im FAMOOS-Projekt als Austauschstandard entwickelt wurde; siehe http://www.iam.unibe.ch/~famoos/FAMIX/

Gibbon (1997). Das Werkzeug TOAD erhebt Code-Metriken zur Überprüfung von Heuristiken. Da das Werkzeug für die Ausbildung objektorientierter Entwickler gedacht ist (Gibbon, Higgins, 1996), werden aber nicht nur Verstöße identifiziert, sondern auch erklärende Texte angeboten, die beschreiben, warum die Verstöße problematisch sind und wie die Verstöße behoben werden können. Die Heuristiken konzentrieren sich auf den Bereich der Wartbarkeit.

Bär (1998). Das Werkzeug überprüft die Einhaltung von Heuristiken im Code und zeigt Verstöße auf. Die Prüfung der Heuristiken wird durch Anfragen an eine Wissensbasis in Prolog realisiert, die aus dem Code gewonnene Entwurfsinformation enthält. Das Werkzeug ist integriert in das Visualisierungswerkzeug GOOSE[4], das vor allem für das Reverse-Engineering objektorientierter Programme gedacht ist. GOOSE verfügt auch über eine Komponente zur Erhebung und Visualisierung von Metriken.

Köhler et al. (1998). Das Metrikenwerkzeug Crocodile, das in die Entwicklungsumgebung SNiFF+ integriert ist, greift auf das (Code-)Repository von SNiFF+ zu, um darauf Metriken zu erheben. Die Metriken können mit Schwellenwerten versehen werden, so dass das Werkzeug Modellelemente identifizieren kann, die besonders häufig oder besonders schwerwiegend unerwünschte Messwerte aufweisen. Auf diese Weise soll der Umfang an Klassen o. Ä. eingeschränkt werden, die anschließend einem Code-Review unterzogen werden.

Together. Inzwischen enthalten auch kommerzielle UML-Werkzeuge automatisierte Hilfsmittel zur Qualitätssicherung. Ein Beispiel ist Together, bei dem es eine Funktion zur Durchführung von „Audits“ gibt. Dabei kann der Benutzer vordefinierte Regeln automatisch prüfen lassen. Außerdem kann er vordefinierte Metriken auswählen, optional Schwellenwerte angeben und sich dann die Metriken berechnen lassen. Der Ansatz von Together ist allerdings sehr stark Code-zentriert, z. B. entsprechen die geprüften Regeln vor allem Programmierrichtlinien.

11.1.3 Bewertung

Die vorgestellten Werkzeuge besitzen viele nützlichen Funktionen, die zur Entwurfsbewertung und teilweise auch zur Entwurfsverbesserung genutzt werden können. Für den Bewertungsansatz in dieser Arbeit lassen sie sich leider jedoch nicht verwenden. Die Code-basierten Werkzeuge können von vornherein ausgeschlossen werden, da der Ansatz in dieser Arbeit auf UML aufbaut. Ebenso scheidet das Werkzeug MEMOS von Baumann aus, da es auf einem proprietären Modellierungsansatz basiert. Argo/UML von Robbins und Redmiles bietet keine Möglichkeit zur Erhebung von Metriken, kann also ebenfalls nicht verwendet werden. MAISA von Nenonen et al. unterstützt zwar Metriken, kann aber nicht mit den Metriken von QOOD konfiguriert werden. Dazu wären Eingriffe in den Quellcode notwendig, der aber nicht zur Verfügung steht. Außerdem wären die Fragebögen sowie die Auswertungsfunktion zu ergänzen.

Deshalb wurde entschieden, Werkzeuge zu entwickeln, die das Bewertungsverfahren optimal unterstützen. Diese Werkzeuge werden im folgenden Abschnitt vorgestellt.

4. GOOSE = Graphs on Object-Oriented Systems, siehe http://www.fzi.de/prost/tools/goose/

11.2 Selbst realisierte Werkzeuge

Schmider (2002) hat das Werkzeug MOOSE (Metrikenwerkzeug für den objektorientierten Systementwurf) für die Erhebung von Metriken auf UML-Modellen erstellt. Dabei war zunächst festzulegen, wie die Metrikenerhebung realisiert wird. Liegt das UML-Modell in einem UML-Werkzeug vor, gibt es drei Möglichkeiten:

1. Das UML-Werkzeug erzeugt eine Repräsentation des UML-Modells im standardisierten Austauschformat XMI (XML Metadata Interchange; OMG, 2000b). Die XMI-Datei wird anschließend geparst, um die Metriken zu erheben.
2. Besitzt das UML-Werkzeug eine Skriptsprache oder eine geeignete API, mit der auf die Modellinformation zugegriffen werden kann (wie z. B. bei Rational Rose), können die Metriken durch Skripte o. Ä. erhoben werden.
3. Die Funktionalität zur Metrikenerhebung wird direkt in den Code des Werkzeugs eingebaut.

Möglichkeit 2 und 3 haben den Nachteil, dass sie spezifisch für ein bestimmtes UML-Werkzeug sind. Dagegen kann bei Möglichkeit 1 jedes UML-Werkzeug verwendet werden, das standardkonformes XMI erzeugt. Daher wurde für die Realisierung die XMI-Variante gewählt.

Das Werkzeug MOOSE besteht aus zwei Teilwerkzeugen: Konverter und Reportgenerator. Der Konverter überführt ein UML-Modell im XMI-Format in ein ODEM-Modell. Der Reportgenerator erzeugt anhand einer Vorlage Berichte über den Entwurf, in die Modellinformationen und Messwerte der Metriken aus QOOD eingebettet sind.

11.2.1 Der Konverter

Der Konverter zieht aus UML-Modellen, die in einer XMI-Datei gespeichert sind, die für ODEM erforderliche Information heraus und legt sie in einer relationalen Datenbank ab (vgl. Abbildung 11-1). Dies ist problemlos möglich, weil beim Entwurf von ODEM die Speicherung der Modellinformation in Datenbank-Relationen bereits vorgesehen wurde. Die Verwendung einer Datenbank als Zwischenspeicher hat den Vorteil, dass die Konvertierung nur einmal vorgenommen werden muss. Außerdem können auf der gespeicherten Information beliebige Anfragen in SQL formuliert werden. Man ist also nicht auf fest in das Werkzeug hineincodierte Metriken beschränkt, sondern kann z. B. auch statistische Auswertungen durchführen.

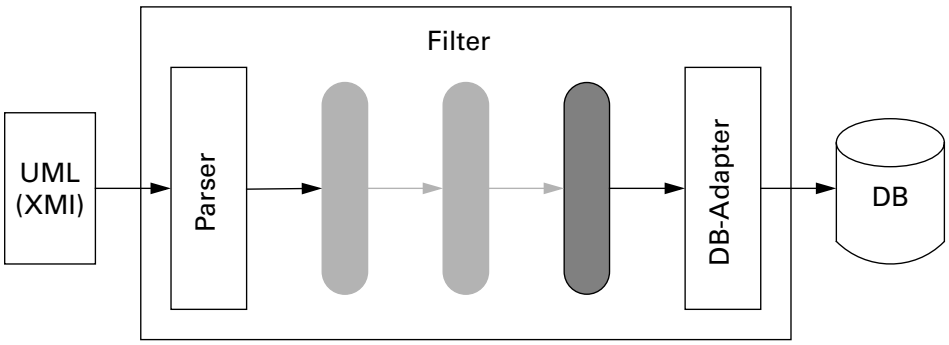

Abbildung 11-1: Architektur des Konverters

Bei der Implementierung des Konverters hat sich herausgestellt, dass die Hersteller verschiedener UML-Werkzeuge unterschiedliche Interpretationen des XMI-Standards haben und entsprechend unterschiedliches XMI erzeugen. Erprobt wurden in diesem Zusammenhang Argo/UML 0.8, Poseidon 1.0 und Together 5.02. Together erzeugt zum Teil sogar inhaltlich falsches XMI (z. B. falscher *ownerScope* bei Attributen und Operationen).[5] Um die Unterschiede und Fehler auszugleichen, wurde der Konverter um ein Filterkonzept erweitert, das die Verarbeitung werkzeugspezifischer Eigenheiten erlaubt. Es können beliebig viele Filter hintereinander geschaltet werden. Außerdem gibt es einen Standardfilter, der offensichtlich unsinnige Eingaben entfernt. Der Parser filtert bereits alles heraus, was für die Konvertierung nach ODEM keine Rolle spielt.

11.2.2 Der Reportgenerator

Der Reportgenerator liest die Datenbank aus und erhebt die gewünschten Metriken. Aus den Modellinformationen und den Messwerten generiert er dann anhand einer Vorlage einen Report. Abbildung 11-2 zeigt die Architektur des Reportgenerators. Der Generator liest die Vorlage ein, in der Platzhalter für Metriken stehen. Er lässt den Evaluator die benötigten Metriken in der Datenbank erheben und fügt die Messwerte anstelle der Platzhalter in die Vorlage ein. Den Report gibt der Generator dann aus.

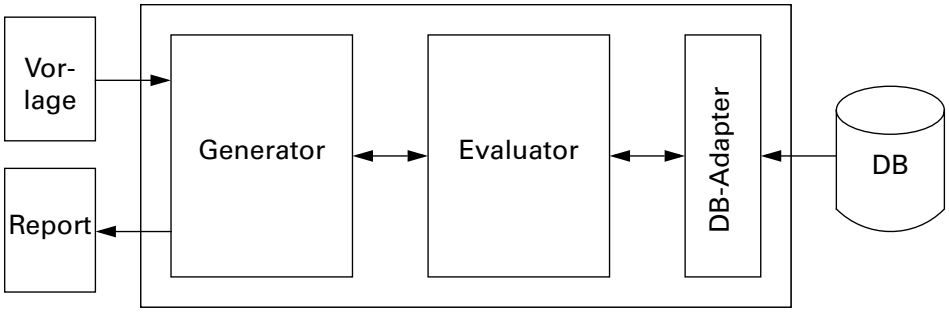

Abbildung 11-2: Architektur des Reportgenerators

Die Definitionen der Metriken (als SQL-Anfragen oder Skripte) findet der Evaluator in einer speziellen Tabelle in der Datenbank, was eine flexible Auswahl und Erweiterung der Metriken erlaubt. Durch die Verwendung von Vorlagen können Reports für die verschiedensten Zwecke generiert werden. Hier werden zwei Beispiele vorgestellt, die realisiert wurden: Dateien für den Import von Messwerten in eine Tabellenkalkulation und Review-Bögen in HTML.

Export in Tabellenkalkulation

Der Reportgenerator erzeugt eine Datei, die in eine Tabellenkalkulation importiert werden kann (ASCII oder HTML). Dabei werden die Messwerte strukturiert aufbereitet. Würden sämtliche Verfeinerungen auf einen Schlag präsentiert, wäre der Bewerter sonst von der Menge der Messwerte überfordert. Es ist sinnvoll, zunächst nur die unverfeinerten Metriken darzustellen und die Verfeinerungen erst anschließend als Untertabellen zu präsentieren.

5. Dieses Problem tritt auch noch in Version 5.5 auf. Dem Support von Togethersoft waren die Fehler noch nicht bekannt, was darauf schließen lässt, dass den XMI-Export bisher kaum jemand benutzt.

In der Tabellenkalkulation können anhand der Messwerte z. B. statistische Auswertungen oder Darstellungen in Diagrammen vorgenommen werden. Auch das Bewertungsverfahren kann innerhalb der Tabellenkalkulation durchgeführt werden. Dazu müssen zusätzlich die Formeln für die Bewertungsvorschläge mit den Gewichten des spezifischen Qualitätsmodells eingegeben werden. Der Bewerter muss außerdem noch die Messwerte der subjektiven Metriken eingeben, damit sich die Qualitätskennzahlen des spezifischen Modells berechnen lassen. Die Fragebögen lassen sich bei diesem Ansatz allerdings nur schwer integrieren.

Review-Bögen

Der Ansatz der Bewertung innerhalb einer Tabellenkalkulation hat den Nachteil, dass die Formeln von Hand eingegeben oder angepasst werden müssen. Praktischer ist die Generierung eines Review-Bogens, in dem die Berechnung weitestgehend automatisiert ist. Ein möglicher Ansatz dazu sind HTML-basierte Reports, die

- eine Zusammenfassung der Eigenschaften und Beziehungen einer Komponente,
- alle zugehörigen objektiven Metriken, z. B. geordnet nach Kriterien,
- die daraus abgeleiteten Bewertungsvorschläge und
- die Fragebögen

enthalten. Ein solcher Report kann ausgedruckt und als Review-Bogen verwendet werden, der von Hand ausgefüllt wird.

Abbildung 11-3 zeigt den Überblicksteil des HTML-Reports für eine Klasse.

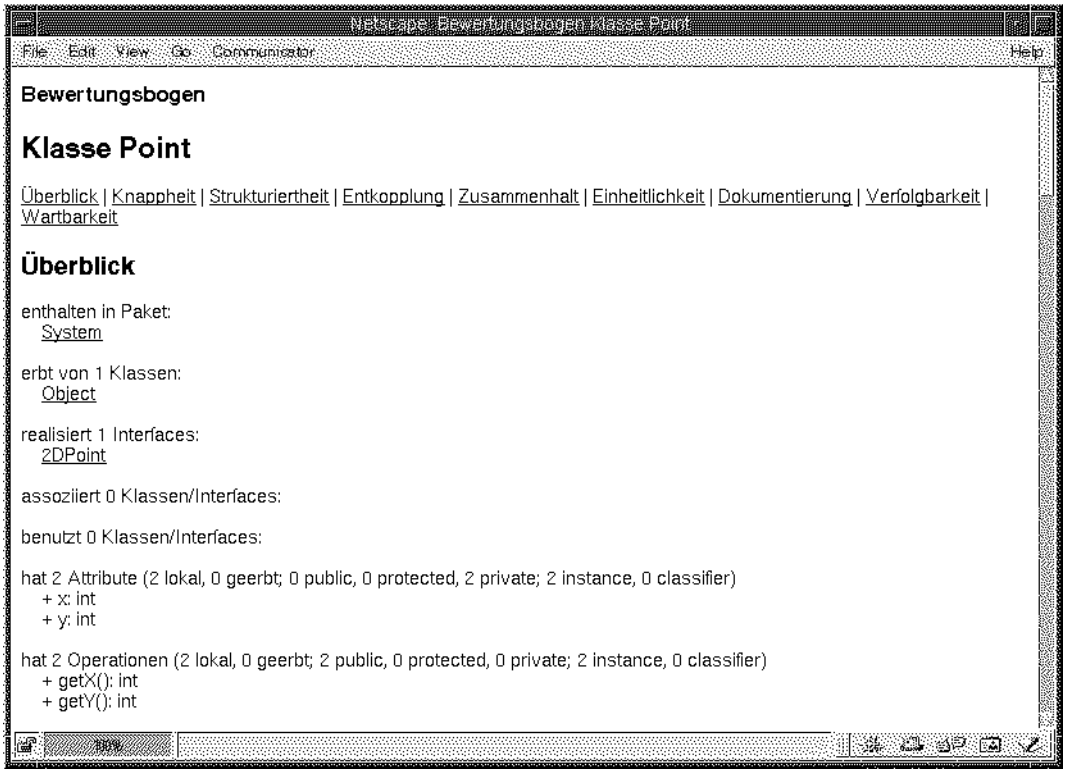

Abbildung 11-3: Überblick über eine Klasse

Durch Verwendung von JavaScript kann die automatische Auswertung in den HTML-Report integriert werden. Man benutzt eine HTML-Form, um die Fragen der Fragebögen direkt im HTML-Browser beantworten zu lassen. Die Antworten werden durch JavaScript-Funktionen ausgewertet (Berechnung des Bewertungsvorschlags). Auch die subjektiven Metriken können eingegeben werden, um daraus Bewertungsvorschläge berechnen zu können. Abbildung 11-4 zeigt den Bereich zur Bewertung des Kriteriums Knappheit einer Klasse mit Metriken, Fragebogen und subjektiver Metrik.

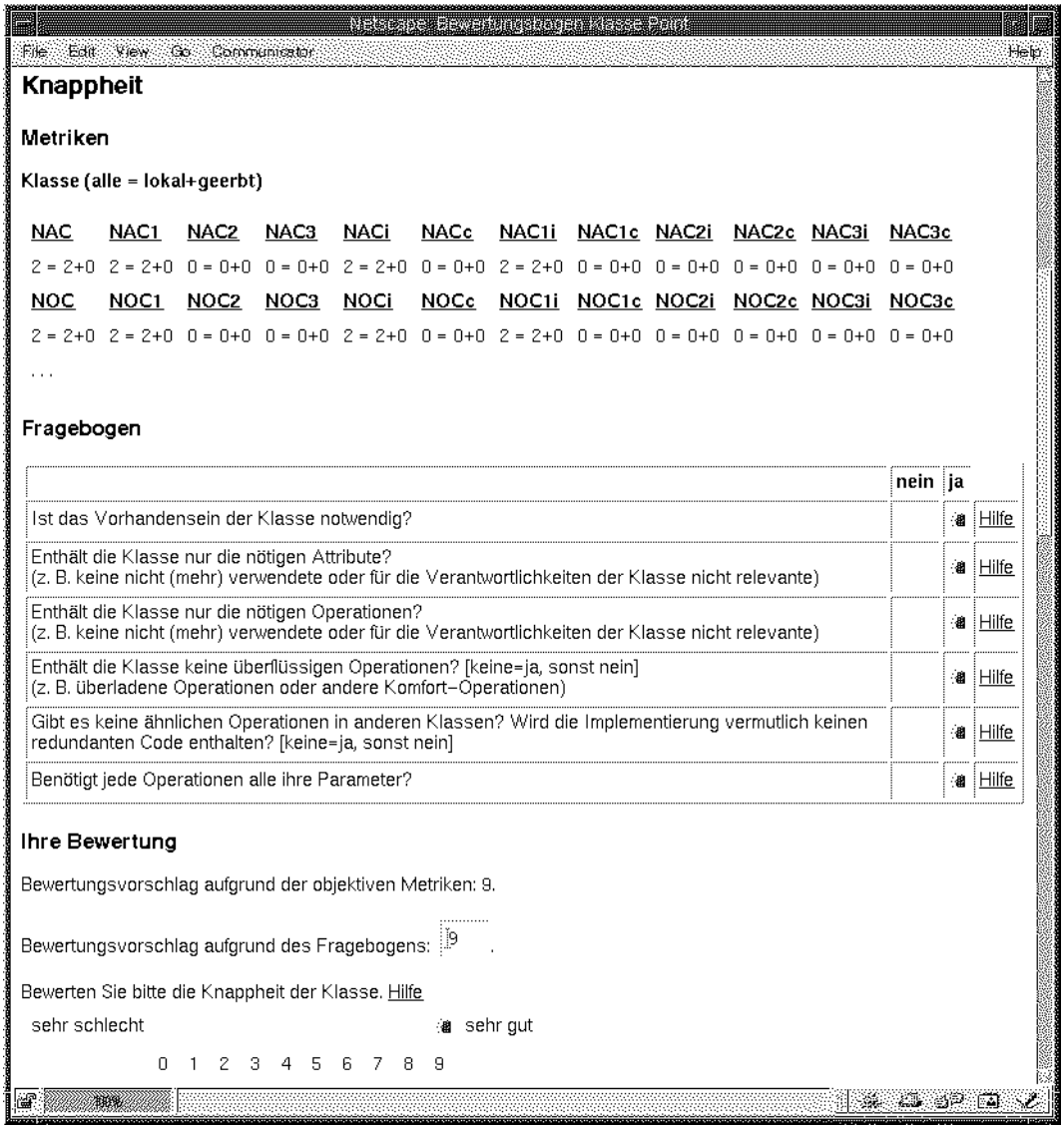

Abbildung 11-4: Der Bewertungsbereich Knappheit

Um dem Bewerter beim Ausfüllen des Review-Bogens zu helfen, kann der Bogen durch Links mit Hilfe-Seiten verknüpft werden. Diese enthalten

- Erläuterungen zu den Metriken,
- Kommentare zu den Fragen der Fragebögen
- Verbesserungsvorschläge zu durch Fragen aufgedeckten Problemen und

- Hinweise zur Abwägung verschiedener Aspekte bei der subjektiven Bewertung.

Die Hilfe-Seiten bilden zusammen eine Art Handbuch zur Entwurfsbewertung. Abbildung 11-5 zeigt ein Beispiel für eine Hilfe-Seite passend zum in Abbildung 11-4 dargestellten Ausschnitt des Review-Bogens.

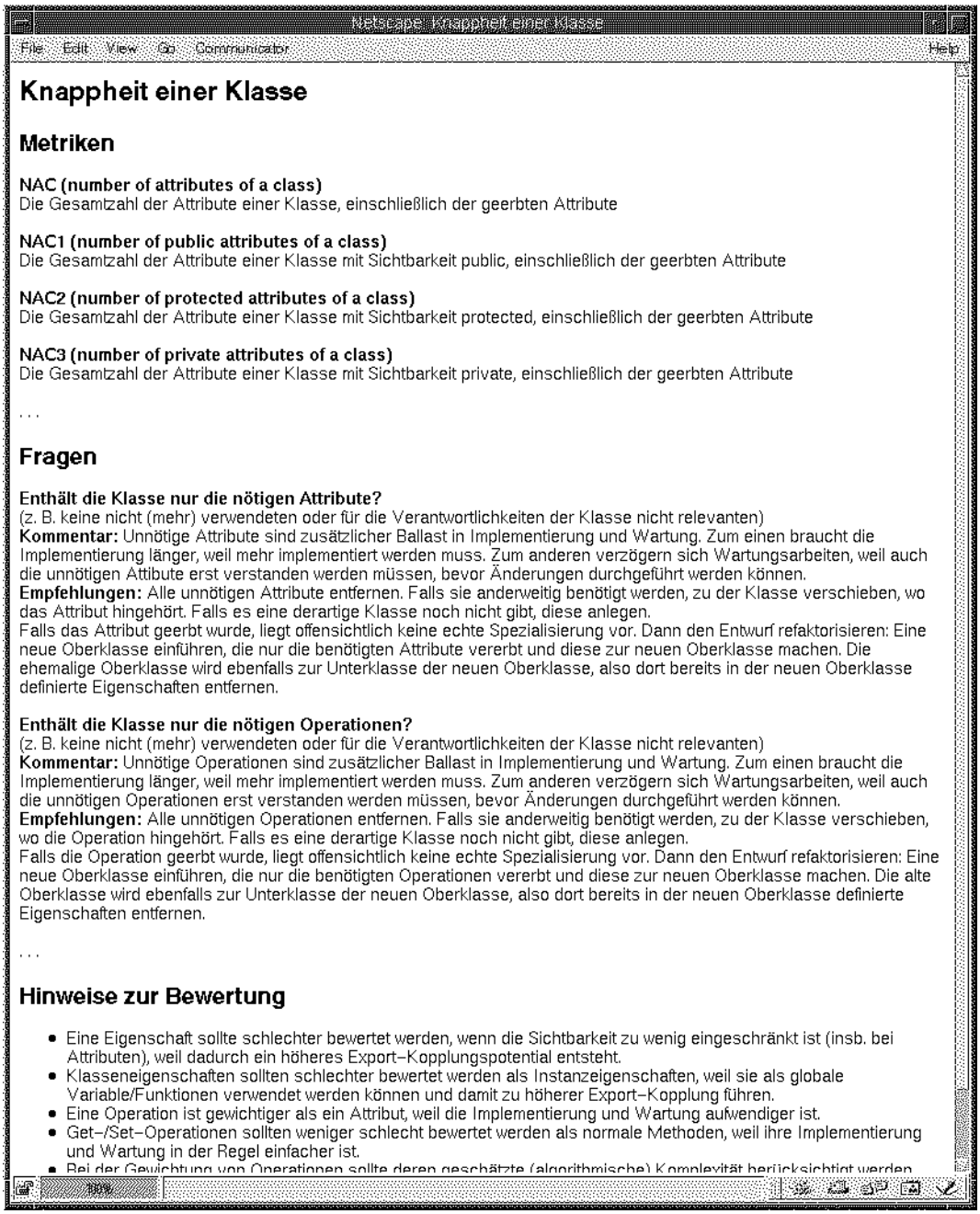

Abbildung 11-5: Hilfeseite zur Bewertung der Knappheit (gekürzt)

Einige der Fragen in den Fragebögen lassen sich automatisch beantworten, z. B. in Abbildung 11-4 die Frage, ob es Assoziationen der Klasse gibt, die in keiner Richtung navigierbar ist. Der Reportgenerator kann so erweitert werden, dass er solche Fragen beantwortet, die Antwort in den Bogen ausgibt und ggf. die „Übeltäter“ hinter der

Frage auflistet. Dazu müssen nur entsprechende SQL-Anfragen in der Datenbank hinterlegt werden, die der Evaluator des Reportgenerators nutzen kann.

11.3 Ausblick: Ein ideales Werkzeug

Betrachtet man die in diesem Kapitel vorgestellten Werkzeuge und ihre Konzepte, lassen sich mögliche Anforderungen an ein ideales Werkzeug zur Entwurfsbewertung. Die folgende Funktionalität scheint mir sinnvoll zu sein:

- Die Bewertungsfunktion ist integriert in ein UML-Werkzeug.
- Es können Metriken erhoben werden. Dabei kann ausgewählt werden, welche Metriken für welche Teile des Entwurfs (Filterung) erhoben werden sollen.
- Den Metriken können Schwellenwerte zugeordnet werden: obere oder untere Schwellenwerte, die absolut oder statistisch (z. B. 80%-Quantil) festgelegt werden. Dabei können auch mehrere Schwellenwerte angegeben werden, so dass mehr als zwei Äquivalenzklassen unterschieden werden können.
- Anhand der Messwerte und der Schwellenwerte kann eine Ausreißeranalyse durchgeführt werden.
- Es können Trendanalysen durchgeführt werden, d. h. Messwerte werden über die Zeit verglichen (dazu werden die Messwerte nach jeder Bewertung archiviert)
- Die Messwerte werden in tabellarischer Form und in Diagrammen aufbereitet.
- Der Export der Messwerte in externe Werkzeuge, z. B. eine Tabellenkalkulation, ist möglich.
- Die Messwerte können auf der Basis eines frei konfigurierbaren Schemas (d. h. eines Qualitätsmodells) aggregiert werden, um Qualitätskennzahlen zu erhalten.
- Ein Bewertungsbrowser erlaubt es, das Entstehen der Endbewertung durch die Aggregationsschritte nachzuvollziehen.
- Optional können subjektive Metriken in das Bewertungsschema aufgenommen werden. Dabei sind Bewertungsvorschläge aus verschiedenen Quellen (z. B. Metriken, Regeln, Fragebögen) verfügbar.
- Ausgewählte Entwurfsregeln können auf der Basis der Metriken automatisch geprüft werden, Verstöße werden aufgelistet.
- Es können Checklisten, Fragebögen oder Review-Bögen generiert werden, die an den Prüfling angepasst sind.
- Es können erklärende Texte zu den Regeln, Checklisten, Fragebögen und Metriken abgerufen werden.
- Es können Texte mit Vorschlägen zur Problembehebung (passend zum Problem) abgerufen werden.
- Es sind automatische Entwurfstransformationen verfügbar (z. B. als Wizards).

Weitere Anforderungen sind im Bereich einer weitergehenden Visualisierung von Messwerten denkbar, die dem Bewerter zusätzliche Erkenntnisse ermöglichen. Hier gibt es bereits erste Ansätze (z. B. Simon et al., 2001).

Das skizzierte ideale Werkzeug kann modular aufgebaut werden, beispielsweise gemäß einem Vorschlag von Hitz und Neuhold (1998) für Code-basierte Werkzeuge (vgl. Abbildung 11-6). Im Zentrum der Architektur steht ein Repository, das die notwendige Information (als Instanz eines Metamodells, z. B. ODEM) enthält. Dieses Repository wird von Parsern für UML-Modelle oder für Code gefüllt. Eine breite Palette von Analyse-Werkzeugen kann dann Auswertungen auf dem Repository durchführen. Die Ergebnisse der Analysen können ebenfalls in dem Repository gespeichert werden (zur Weiterverarbeitung durch andere Werkzeuge und für Trendanalysen).

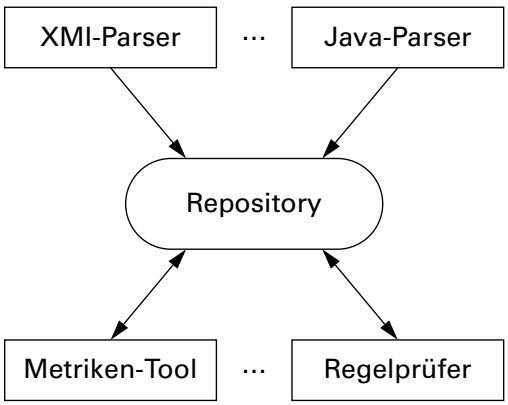

Abbildung 11-6: Architektur des idealen Werkzeugs

Eine vollständige Implementierung des idealen Werkzeugs ist wünschenswert, war im Rahmen dieser Arbeit allerdings nicht vorgesehen. Die von Schmider (2002) realisierten Werkzeuge bilden aber eine geeignete Basis für ein umfassenderes Werkzeug, denn die Werkzeuge Konverter und Reportgenerator sowie die ODEM-Datenbank als Repository entsprechen der Architektur aus Abbildung 11-6.

Kapitel 12

Zusammenfassung und Ausblick

12.1 Zusammenfassung

Diese Arbeit hat sich mit der Frage beschäftigt, wie die Qualität eines objektorientierten Entwurfs bewertet werden kann. Dabei liegt der Schwerpunkt aus praktischen Erwägungen auf statischer Entwurfsinformation, die aus UML-Diagrammen gewonnen werden kann. Für diese Entwurfsinformation wurde ein formales Referenzmodell entwickelt, das Object-Oriented Design Model (ODEM). ODEM definiert den Gegenstand des Bewertungsverfahrens und kann außerdem zur formalen Definition von Metriken eingesetzt werden.

Das allgemeine Qualitätsmodell Quality Model for Object-Oriented Design (QOOD) definiert die Qualitätsfaktoren und -kriterien, für die eine Bewertung auf der Basis von ODEM möglich ist. Der Schwerpunkt von QOOD liegt auf dem Faktor Wartbarkeit, für den auch eine Quantifizierung angegeben ist. Diese Quantifizierung besteht nur zum Teil aus objektiven Metriken, weil sich wesentliche Aspekte des Entwurfs (vor allem semantische Aspekte) nicht oder nur unzureichend durch objektive Metriken erfassen lassen. Daher wurden subjektive Metriken hinzugenommen, deren Erhebung durch Fragebögen erleichtert und gesteuert wird.

Bei der Auswahl der objektiven Metriken für QOOD hat sich gezeigt, dass sich für viele Metriken Verfeinerungen angeben lassen, mit denen differenziertere Aussagen möglich werden. Andererseits erhöhen die Verfeinerungen den Mess-, Kalibrierungs- und Interpretationsaufwand.

Aus dem allgemeinen Qualitätsmodell QOOD lassen sich spezifische Qualitätsmodelle ableiten, die den Kontext, die konkreten Qualitätsanforderungen und die Qualitätssicht berücksichtigen. Bei der Ableitung des spezifischen Modells werden Faktoren, Kriterien, Metriken und Fragen ausgewählt sowie eine Gewichtung festgelegt.

12.2 Bewertung des Ansatzes

Die in Abschnitt 1.2 formulierten Anforderungen konnten mit dem Ansatz insgesamt erfüllt werden. Hier werden nun verschiedene Aspekte des Ansatzes betrachtet.

12.2.1 ODEM

ODEM eignet sich für die formale Definition von Entwurfsmetriken, wie sich in Fallstudien (vgl. Abschnitt 5.5.2) und bei der Definition der Metriken für QOOD (vgl. Anhang A) gezeigt hat. Die Abstraktionsschicht über dem UML-Metamodell erhöht dabei die Verständlichkeit der formalen Definitionen. Allerdings deckt ODEM in seiner jetzigen Form keine parametrisierten Elemente (Templates) ab. Außerdem sind keine Informationen über die Implementierung der Operationen (die Methoden) verfügbar, so dass gewisse Entwurfsmetriken aus der Literatur mit ODEM nicht formalisiert werden können.

12.2.2 QOOD

In einer Fallstudie (vgl. Kapitel 10) wurde gezeigt, wie einfach die Ableitung eines spezifischen Modells von QOOD ist. Die Durchführung der Bewertung ist ebenfalls einfach, insbesondere wenn entsprechende Werkzeugunterstützung verfügbar ist. Die Dauer der Bewertung ist dabei vor allem von der Größe des Entwurfs und der Qualität der Dokumentation abhängig (z. B. erleichtert das Vorhandensein einer brauchbaren Entwurfsübersicht den Einstieg). Die Resultate der Bewertung haben sich beim Vergleich von Entwurfsalternativen als zuverlässige Indikatoren für den (relativen) Wartungsaufwand erwiesen.

Die Entwürfe der Fallstudie sind allerdings nicht besonders groß, weshalb keine sichere Aussage darüber gemacht werden kann, wie gut das Verfahren auf große Entwürfe skaliert. Meiner Ansicht nach dürfte es aber keine größeren Probleme geben, da der Ansatz bottom-up arbeitet und daher die Schwierigkeit der Durchführung nur schwach mit der Größe des Entwurfs zunimmt.

Ein wichtiges Nebenprodukt der Bewertung ist eine Liste von entdeckten Mängeln, die behoben werden sollten. Diese Mängel werden vor allem bei der Beantwortung der Fragebögen identifiziert. Zusätzlich können auch Ausreißer bei den objektiven Metriken als Indikatoren für potentielle Schwachstellen verwendet werden.

12.2.3 Automatisierung

Die Entwurfsinformation in ODEM kann in eine relationale Datenbank abgebildet werden. Dazu wurde ein Werkzeug implementiert, dass die nötige Information aus UML-Modellen (dargestellt in XMI) extrahiert und in eine Datenbank schreibt (vgl. Abschnitt 11.2.1). Auf der Datenbank können dann die Messungen vorgenommen werden. Die formalen Definitionen der Metriken lassen sich leicht in entsprechende Datenbankabfragen umsetzen, so dass sich die Erhebung der objektiven Metriken voll automatisieren lässt. Für die Erhebung und Aufbereitung der Messwerte in Form von Reports ist ebenfalls ein Werkzeug verfügbar (vgl. Abschnitt 11.2.2). Umfang und Form der Reports sind frei wählbar, da sie anhand von Vorlagen generiert werden.

Die Beantwortung der Fragen der Fragebögen kann auch automatisiert werden, sofern sich die Fragen formalisieren und auf der Basis von ODEM beantworten lassen. Dies gilt aber nur für einen Teil der Fragen, denn vor allem semantische Fragen, z. B. nach Zusammenhalt, nach aussagekräftigen Bezeichnern oder nach funktionaler Redundanz, entziehen sich der Automatisierung.

Sofern auf alle Aspekte verzichtet wird, die sich nicht automatisieren lassen, kann aus QOOD ein spezifisches Modell entwickelt werden, das sich für eine vollautomatische Bewertung eignet. Dieses wird sich allerdings vor allem auf die Bereiche Knappheit, Strukturierung und Entkopplung beschränken müssen. Damit sind zwar einige der wichtigsten Kriterien der Wartbarkeit abgedeckt, doch fallen andere wichtige Kriterien wie Zusammenhalt oder Dokumentierung faktisch weg.

12.2.4 Kosten und Nutzen

Qualitätssicherung beim Entwurf verursacht Kosten. Wird der vorgestellte Ansatz zur Entwurfsbewertung verwendet, muss zunächst ein spezifisches Qualitätsmodell abgeleitet werden. Dann ist die Bewertung durchzuführen. Der Zeitaufwand für die Ableitung des spezifischen Qualitätsmodells ist nur einmal zu leisten, während der Aufwand für die Bewertung für jede Bewertung erneut anfällt. Deshalb ist Werkzeugunterstützung bei der Bewertung wichtig.

Der Nutzen des Ansatzes besteht darin, bereits früh in der Entwicklung Rückkopplung über die Qualität des Entwurfs zu bekommen. Identifizierte Mängel im Entwurf (und in seiner Dokumentation) können ausgebessert werden, so dass Fehlerfolgekosten in Implementierung und Wartung vermieden werden. Da diese Folgekosten sehr hoch sein können, fallen die Qualitätssicherungskosten dagegen kaum ins Gewicht. Es ist also mit einem positiven Nettonutzen zu rechnen.

Ein guter Entwurf garantiert allerdings noch keine gute Implementierung, da bei der Implementierung immer noch z. B. Fehlentscheidungen des Managements oder Abweichungen vom Entwurf möglich sind. Die Wahrscheinlichkeit für eine gute Implementierung wird aber durch einen guten Entwurf deutlich erhöht.

12.2.5 Einsatz in der Praxis

Für den Einsatz in der Praxis (z. B. in der Industrie) ist der Ansatz grundsätzlich geeignet. Allerdings sollte die Werkzeugunterstützung noch ausgebaut werden. Außerdem wäre es hilfreich, wenn das Bewertungsverfahren in UML-Modellierungswerkzeuge integriert wäre. Ein Problem dürfte die bisher nur rudimentäre Unterstützung bei der Ableitung spezifischer Modelle sein. Vorgefertigte spezifische Modelle, die als Vorlagen dienen können, dürften die Ableitung erleichtern.

Eine wichtige psychologische Voraussetzung für das Verfahren ist, dass es nicht dazu eingesetzt wird, den Entwerfer zu bewerten. Ansonsten wird der Entwerfer seine Mitwirkung verweigern – oder, falls er das Verfahren selbst durchführen soll, geschönte Ergebnisse vorlegen. Es wird allerdings ohnehin empfohlen, Entwürfe nur von erfahrenen Entwerfern bewerten zu lassen, die nicht Mitglieder des Entwicklungsteams sind (Abowd et al., 1996).

12.2.6 Grenzen des Ansatzes

Die wichtigste Einschränkung des Ansatzes ist, dass sämtliche dynamischen Entwurfseigenschaften unberücksichtigt bleiben. Das führt dazu, dass wichtige Qualitäten wie Effizienz oder Zuverlässigkeit nicht bewertet werden können. Ursache für die Beschränkung ist die Beobachtung, dass dynamische Entwurfsinformation in UML zwar ausgedrückt werden kann, dies in der Praxis aber nur bruchstückhaft getan wird. Soll auch dynamische Entwurfsinformation bei der Bewertung berücksichtigt werden, müssen entsprechende Anforderungen an Umfang und Detaillierungsgrad der UML-Dokumentation gestellt werden.

Es hat sich gezeigt, dass die UML-Dokumentation eines Entwurfs nicht ausreicht, um den Entwurf vollständig zu beschreiben. Zusätzlich ist weitere erläuternde Dokumentation notwendig. Diese ließe sich zwar theoretisch in Form von Notizen auch in UML darstellen, doch ist ein beschreibendes Entwurfsdokument, in das UML-Diagramme eingebettet sind, übersichtlicher und leichter verständlich. Diese über die UML-Diagramme hinausgehende Information wird bei der Bewertung genutzt, ist aber nicht im Referenzmodell ODEM repräsentiert.

Eine Entwurfsbewertung ist am effektivsten, wenn sie in Zusammenhang mit einem Entwicklungsprozess eingesetzt wird, der dem Wasserfallmodell (Royce, 1970) entspricht, weil am Ende der Entwurfsphase der Entwurf vollständig vorliegt, aber noch keine Realisierung vorgenommen wurde. Ein solcher Entwicklungsprozess ist aber nur dann sinnvoll, wenn die Anforderungen klar und stabil sind, was selten gegeben ist. Daher findet Software-Entwicklung häufig in einem iterativen, evolutionären Prozess statt. Der Entwurf entsteht (und wächst) schrittweise und wird immer wieder überarbeitet. Auf der einen Seite kann dieses Vorgehen zu einem besseren Entwurf führen als beim Wasserfallmodell. Auf der anderen Seite kann es schwieriger werden, Fehlentwicklungen zu erkennen, weil eine Bewertung des Gesamtbilds erst spät im Entwicklungsprozess möglich ist. Der vorgestellte Ansatz zur Bewertung kann jederzeit durchgeführt werden, auch auf Zwischenversionen des Entwurfs. Nur kann eben immer nur die bereits vorliegende Entwurfsinformation berücksichtigt werden.

Eine völlig rationale, objektive und gleichzeitig weitgehend zutreffende Bewertung des Entwurfs ist theoretisch denkbar. Der dazu zu treibende Aufwand ist allerdings immens: Der Entwurf muss hinreichend formalisiert werden, es muss ein spezifisches Qualitätsmodell erstellt und validiert werden, und für jede Bewertung sind die erforderlichen Daten zu erheben und zu verarbeiten. Der Gesamtaufwand dürfte also so groß sein, dass es auch in Zukunft schon aus ökonomischen Erwägungen erforderlich sein wird, die Bewertung im Wesentlichen mit Hilfe der Intuition und Erfahrung eines guten Entwerfers durchzuführen. Qualitätsmodelle mit objektiven Metriken und daraus abgeleitete automatisierte Entwurfsregeln sind dabei praktische Hilfsmittel der Analyse, die dem Bewerter die Arbeit erleichtern, ihn aber nicht ersetzen.

12.3 Vergleich mit anderen Arbeiten

Grundlage der Entwurfsbewertung

In dieser Arbeit ist die Grundlage der Entwurfsbewertung ein UML-Modell. Die meisten Arbeiten im Bereich der Entwurfsbewertung stützen sich allerdings entweder

auf eine nicht näher spezifizierte Architekturdokumentation (z. B. Bass et al., 1998; Clements et al., 2002) oder auf Code (z. B. Erni, 1996; Bär, 1998). Für die Bewertung auf der Basis einer Entwurfsdokumentation in UML gibt es bisher nur die Ansätze von Robbins, Redmiles (1999) und Nenonen et al. (2000). Bei Robbins und Redmiles werden weder Metriken erhoben noch wird eine Gesamtbewertung angestrebt; der Schwerpunkt liegt im Aufzeigen von Mängeln auf der Basis von Entwurfsregeln. Der Ansatz von Nenonen et al. arbeitet mit Metriken und bezieht dabei auch UML-Bestandteile zur Modellierung der dynamischen Struktur ein. Die erhobenen Metriken sollen zur Qualitätsbewertung dienen – wie diese allerdings genau durchgeführt werden soll, sagen die Autoren nicht.

Qualitätsmodellierung

In Abschnitt 7.4 wurden einige Qualitätsmodelle für den objektorientierten Entwurf vorgestellt. Die wenigsten sehen eine quantitative Bewertung vor. Diese Modelle eignen sich daher besser für eine qualitative Bewertung bzw. für eine Mängelanalyse auf der Basis von Fragebögen und Checklisten. Ein quantitatives, validiertes Qualitätsmodell findet sich bei Erni (1996), es ist jedoch nur für die Bewertung der Wartbarkeit von Rahmenwerken gedacht. Auch dieses Qualitätsmodell dient vor allem der Mängelanalyse, auf eine Gesamtbewertung wird verzichtet. Nur bei Bansiya und Davis (2002) gibt es eine Gesamtbewertung, die durch teilweise gewichtete Aggregation von Metriken entsteht. Eine Gesamtbewertung ist Voraussetzung für den Vergleich von Entwurfsalternativen, die auch durch spezifische Modelle auf der Basis von QOOD ermöglicht wird.

Bewertungstechniken

Wie in Abschnitt 7.6 gezeigt gibt es verschiedene Techniken zur Bewertung von Entwürfen: Szenarien, Simulation, Metriken, Fragebögen und Checklisten. Metriken werden häufig zur Bewertung verwendet, z. B. arbeiten fast alle der in Abschnitt 11.1 vorgestellten Arbeiten mit Metriken. Allerdings werden die Metriken nie exakt definiert – ein Mangel, der in dieser Arbeit durch die Einführung des Referenzmodells ODEM vermieden wird. Außerdem werden hier die objektiven Metriken mit Fragebögen (und subjektiven Metriken) kombiniert. Dies hat sich als vorteilhaft erwiesen, weil die Fragebögen die Unzulänglichkeiten der objektiven Metriken bei der Bewertung ausgleichen können. Diese Kombination ist in dieser Form bei der Entwurfsbewertung bisher einzigartig. Neu ist ebenfalls das Konzept der Verfeinerung von Metriken.

Eine mögliche Alternative zu den Fragebögen ist die Kombination von Metriken mit Szenarien, wie sie Briand und Wüst (2001) in einer Fallstudie zur Bewertung der Wartbarkeit vorgenommen haben. Das Ergebnis dieser Fallstudie deutet darauf hin, dass sich die beiden Bewertungstechniken ebenfalls gut ergänzen. In eine ähnliche Richtung gehen die Erkenntnisse von Laitenberger et al. (2000), die durch ein Experiment festgestellt haben, dass bei der Inspektion von UML-Dokumenten ein Szenario-basierter Prüfansatz einem Checklisten-basierten überlegen ist, was Effektivität und Kosten angeht.

Abowd et al. (1996) und Bosch (2000) raten bei der Bewertung von Software-Architekturen von der Verwendung von Metriken eher ab und empfehlen stattdessen Szenarien, Fragebögen, Checklisten und Simulationen. Das könnte allerdings daran liegen, dass Architekturbeschreibungen häufig nicht formal genug sind, oder dass Metriken

in der Praxis oft ohne vorhergehende Validierung eingesetzt werden und daher die Resultate unbefriedigend sind.

12.4 Ausblick

12.4.1 Erweiterungen

Als erster Schritt der Weiterentwicklung des Ansatzes sollten die übrigen Faktoren von QOOD ebenfalls quantifiziert werden. Da viele ihrer Kriterien auch bei der Wartbarkeit vorkommen, dürfte das relativ leicht möglich sein. Dann ist zu überlegen, ob es sinnvoll ist, einige Einschränkungen von ODEM aufzuheben, z. B. das Weglassen geschachtelter Klassen oder der Multiplizitäten von Assoziationen.

Ein Problem des Ansatzes ist die eingeschränkte Informationsbasis, auf der die Bewertung durchgeführt wird. Hier ist es sinnvoll, zusätzlich dynamische Entwurfsinformation zu berücksichtigen, so dass QOOD um weitere wichtige Faktoren erweitert werden könnte. Wenn Code verfügbar ist, sollte ergänzend zum UML-Modell aus diesem detaillierte Entwurfsinformation (z. B. Methodenaufrufe durch Methoden) extrahiert werden. Dazu muss ODEM entsprechend erweitert werden, damit QOOD mit passenden Metriken angereichert werden kann.

12.4.2 Vision

Abschließend wird eine Vision entwickelt, wie eine umfassende Entwurfsunterstützung auf der Basis von QOOD erreicht werden kann. Die vorliegende Arbeit stellt nur einen kleinen Schritt in diese Richtung dar, kann jedoch als Basis dienen.

QOOD wurde in dieser Arbeit für die Entwurfsbewertung entwickelt, kann aber auch noch anderweitig genutzt werden. Zum einen können aus QOOD Richtlinien abgeleitet werden, welche die Entstehung eines guten Entwurfs begünstigen. Zum anderen können Restrukturierungen abgeleitet werden, die zu Entwurfsverbesserungen führen. In Abbildung 12-1 sind diese drei Anwendungsmöglichkeiten in einen iterativen Entwurfsprozess integriert.

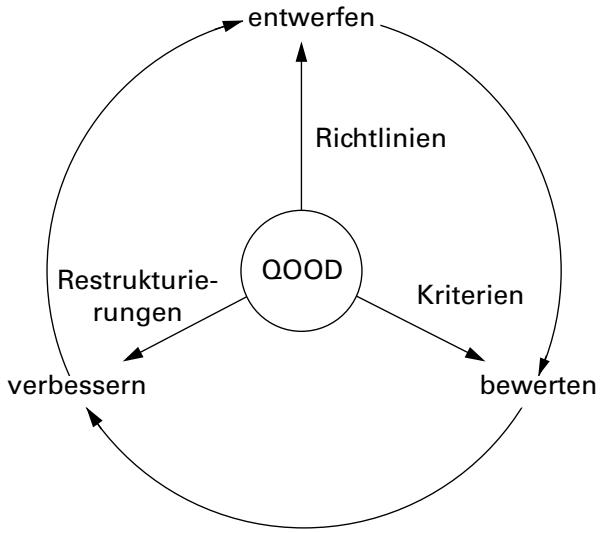

Abbildung 12-1: Entwurfsunterstützung durch QOOD

Entwerfen

QOOD kann den Entwurfsprozess durch abgeleitete Richtlinien und Empfehlungen unterstützen. Die erläuternden Texte zu den Kriterien können Hilfestellungen zur Lösung von auftretenden Entwurfsproblemen geben. Außerdem kann mit Hilfe von QOOD bereits in der Analysephase geprüft werden, ob alle erforderlichen Qualitäten in der Anforderungsspezifikation definiert sind.

Bewerten

Die Entwurfsbewertung ist das eigentliche Thema dieser Arbeit. Die Verwendung der Bewertung zum Vergleich von Alternativen und zur Schwachstellenanalyse wurde bereits angesprochen. Allerdings sind noch einige Erweiterungen bei der Auswertung der Metriken denkbar. Neben einer statistischen Auswertung, die Durchschnitte, Standardabweichungen, Maxima etc. liefert, ist auch eine Visualisierung der Messwerte sinnvoll.

Eine weitere sinnvolle Ergänzung sind Trendanalysen. Dazu werden die Messwerte und Bewertungen eines Entwurfs über seine Entstehungsgeschichte hinweg gesammelt. Dadurch können insbesondere ungewollte Verschlechterungen erkannt werden, die sonst bei einer einzelnen Bewertung nicht aufgefallen wären.

Verbessern

Add quality to every design and piece of code touched: Maintainers cannot become more productive without quality improvements.
(Arthur, 1988, S. 225)

Die Bewertung zeigt Probleme auf, löst sie aber nicht (Card, Glass, 1990). Häufig ist jedoch die Problembehebung viel schwieriger als die Problementdeckung – insbesondere für Entwurfsanfänger, deren Wissen über alternative Lösungsansätze beschränkt ist. In der Literatur werden einige Verbesserungsmaßnahmen in Form von Transformationen vorgeschlagen, z. B. „unit operations“ für die Architektur von Bass et al. (1998), Problem-Lösungs-Muster von Page-Jones (1995), Transformationsmuster von Riel (1996) oder Refaktorisierungen von Fowler et al. (1999). Diese können dem Entwerfer in aufbereiteter Form zugänglich gemacht werden. Schließlich kann auch noch auf Entwurfsmuster hingewiesen werden, deren Verwendung ebenfalls eine Lösung sein kann.

Entwurfsbewertung und -verbesserung sind eng miteinander verknüpft. Wie schon in Abschnitt 11.2.2 angedeutet gibt es Sinn, bereits bei der Identifikation von Problemen in der Bewertung auf geeignete Verbesserungsmaßnahmen hinzuweisen. Unter Verwendung von QOOD können die Verbesserungsmaßnahmen nach ihren Auswirkungen auf die einzelnen Kriterien und Ebenen klassifiziert werden, so dass klar ist, welche Maßnahmen betrachtet werden sollten, wenn z. B. die Entkopplung auf Paketebene niedrig ist. Außerdem können so positive und negative Auswirkungen der Maßnahmen auf andere Kriterien dokumentiert werden. Hat sich der Entwerfer für eine Verbesserungsmaßnahme oder ein Bündel von Maßnahmen entschieden, kann er durch eine erneute Entwurfsbewertung nach Durchführung der Verbesserungsmaßnahme überprüfen, ob tatsächlich eine Verbesserung eingetreten ist.

12.5 Schlussbemerkung

Sollte ich zufällig irgendetwas mehr oder weniger hierher Gehörendes oder Notwendiges ausgelassen haben, so bitte ich um Nachsicht, da es niemanden gibt, der in allen Dingen frei von Tadel ist und an alles im Voraus denken kann.
(Leonardo Fibonacci, Liber Abaci)

Das Gebiet der Qualität ist ein weites Feld, in dem umfassende und gleichzeitig objektiv richtige, d. h. wissenschaftlich haltbare Erkenntnisse nur schwer zu erlangen sind. Ursache dafür ist zum einen die Definitionsproblematik: Die Frage „Was ist Qualität?“ wird von jedem Entwerfer unterschiedlich beantwortet. Weil es keinen Standard gibt, muss eine eigene Definition eingeführt werden. Zum anderen ist es auch nötig, die Brauchbarkeit dieser Definition zu zeigen, wozu umfassende empirische Studien erforderlich sind.

Diese Arbeit hat vorhandene Ansichten zur Entwurfsqualität zusammengetragen und einen Vorschlag gemacht, wie Entwurfsqualität definiert und gemessen werden kann. Auch wenn nicht alle Fragen zur Brauchbarkeit des Ansatzes geklärt werden konnten, halte ich meinen Beitrag für geeignet, als Diskussionsgrundlage für die weitere Erforschung des objektorientierten Entwurfs und seiner Qualität zu dienen.

Literatur

Abowd et al. (1996) Abowd, G.; Bass, L.; Clements, P.; Northrop, L.; Zaremski, A.: Recommended Best Industrial Practice for Software Architecture Evaluation. Technical Report CMU/SEI-96-TR-025 , 1996.

Abreu (2001) Abreu, F.: Using OCL to Formalize Object-Oriented Metrics Definitions. Proceedings of the 5th International ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering (QAOOSE 2001), Budapest, 2001, 99-134.

Abreu, Carapuca (1994) Abreu, F.; Carapuca, R.: Candidate Metrics for Object-Oriented Software within a Taxonomy Framework. Journal of Systems and Software, 26(1), 1994, 87-96.

Adelson, Soloway (1985) Adelson, B.; Soloway, E.: The Role of Domain Experience in Software Design. IEEE Transactions on Software Engineering, 11(11), 1985, 1351-1360.

Akao (1990) Akao, Y.: Quality Function Deployment: Integrating Customer Requirements into Product Design. Productivity Press, Cambridge, 1990.

Alexander et al. (1977) Alexander, C.; Ishikawa, S.; Silverstein, M.: A Pattern Language. Oxford University Press, Oxford, 1977.

Alexander (1979) Alexander, C.: The Timeless Way of Building. Oxford University Press, Oxford, 1979.

Alexander (2001) Alexander, R.: Improving the Quality of Object-Oriented Programs. IEEE Software, 18(5), 2001, 90-91.

Archer, Stinson (1995) Archer, C.; Stinson, M.: Object-Oriented Software Measures. Technical Report CMU/SEI-95-TR-002, 1995.

Arthur (1988) Arthur, L.: Software Evolution: The Software Maintenance Challenge. Wiley, New York, 1988.

Arthur (1993) Arthur, L.: Improving Software Quality: An Insider's Guide to TQM. Wiley, New York, 1993.

Balzert (1985a) Balzert, H.: Allgemeine Prinzipien des Software Engineering. Angewandte Informatik, 1/1985, 1-8.

Balzert (1985b) Balzert, H.: Phasenspezifische Prinzipien des Software Engineering. Angewandte Informatik, 3/1985, 101-110.

Balzert (1998) Balzert, H.: Software-Management, Software-Qualitätssicherung, Unternehmensmodellierung. Lehrbuch der Softwaretechnik, Bd. 2. Spektrum, Heidelberg, 1998.

Balzert (1999) Balzert, H.: Lehrbuch der Objektmodellierung. Spektrum, Heidelberg, 1999.

Bansiya, Davis (1997) Bansiya, J.; Davis, C.: Automated Metrics for Object-Oriented Development: Using QMOOD++ for Object-Oriented Metrics. Dr. Dobb's Journal, December 1997, 42-48.

Bansiya, Davis (2002) Bansiya, J.; Davis, C.: A Hierarchical Model for Object-Oriented Design Quality Assessment. IEEE Transactions on Software Engineering, 28(1), 2002, 4-17.

Bansiya et al. (1999) Bansiya, J.; Etzkorn, L.; Davis, C.; Li, W.: A Class Cohesion Metric for Object-Oriented Designs. Journal of Object-Oriented Programming, 11(8), January 1999, 47-52.

Basili, Rombach (1988) Basili, V.; Rombach, H.: The TAME Project: Towards Improvement-Oriented Software Environments. IEEE Transactions on Software Engineering, 14(6), 1988, 758-773.

Bass et al. (1998) Bass, L.; Clements, P.; Kazman, R.: Software Architecture in Practice. Addison-Wesley, Reading, MA, 1998.

Baumann (1997) Baumann, K.: Unterstützung der objektorientierten Systemanalyse durch Softwaremaße. Physica-Verlag, Heidelberg, 1997.

Baumert (1991) Baumert, J.: New SEI Maturity Model Targets Key Practices. IEEE Software, 8(6), 78-79.

Beck (1996) Beck, K.: Make it Run, Make it Right: Design Through Refactoring. Smalltalk Report, 6(4), 1997, 19-24.

Beck (1999a) Beck, K.: Extreme Programming Explained: Embrace Change. Addison-Wesley, Reading, MA, 1999.

Beck (1999b) Beck, K.: Embracing Change with Extreme Programming. IEEE Computer 32(10), 1999, 70-77.

Beck, Cunningham (1989) Beck, K.; Cunningham, W.: A Laboratory for Teaching Object-Oriented Thinking. Proceedings of OOPSLA'89; ACM SIGPLAN Notices, 24(10), 1989, 1-6.

Beck, Gamma (1998) Beck, K.; Gamma, E.: Test-infiziert: Wie Programmierer das Tests-Schreiben lieben lernen. Java Spektrum, 5/1998, 22-32.

Bell et al. (1987) Bell, G.; Morrey, I.; Pugh, J.: Software Engineering: A Programming Approach. Prentice Hall, New York, 1987.

Berard (1993) Berard, E.: Essays on Object-Oriented Software Engineering, Bd. 1. Prentice Hall, Englewood Cliffs, NJ, 1993.

Berg et al. (1995) Berg, W.; Cline, M.; Girou, M.: Lessons Learned from the OS/400 OO Project. Communications of the ACM, 38(10), 1995, 54-64.

Bersoff et al. (1980) Bersoff, E.; Henderson, V.; Seigel, S.: Software Configuration Management. Prentice Hall, 1980.

Beyer et al. (2000) Beyer, D.; Lewerentz, C.; Simon, F.: Impact of Inheritance on Metrics for Size, Coupling, and Cohesion in Object-Oriented Systems. In: Dumke, R.; Abran, A. (Hrsg.): New Approaches in Software Measurement: Proceedings of the 10th International Workshop on Software Measurement (IWSM 2000); Lecture Notes on Computer Science, 2006. Springer, Berlin, 2000, 1-17.

Bieman (1992) Bieman, J.: Deriving Measures of Software Reuse in Object-Oriented Systems. In: Denvir, T.; Herman, R.; Whitty, R. (Hrsg.): Formal Aspects of Measurement: Proceedings of the BCS-FACS Workshop on Formal Aspects of Measurement, London. Springer, London, 1992, 63-83.

Bieman, Kang (1995) Bieman, J.; Kang, B.: Cohesion and Reuse in an Object-Oriented System. Proceedings of the ACM Symposium on Software Reusability (SSR'95), 1995, 259-262.

Bieman, Kang (1998) Bieman, J.; Kang, B.: Measuring Design-Level Cohesion. IEEE Transactions on Software Engineering, 24(2), 1998, 111-124.

Binkley, Schach (1996) Binkley, A.; Schach, S.: A Comparison of Sixteen Quality Metrics for Object-Oriented Design. Information Processing Letters, 58(6), 1996, 271-275.

Boehm (1976) Boehm, B.: Software Engineering. IEEE Transactions on Computers, 25(12), 1976, 1226-1241.

Boehm (1983) Boehm, B.: Seven Basic Principles of Software Engineering. Journal of Systems and Software, 5(3), 1983, 3-24.

Boehm, Basili (2001) Boehm, B.; Basili, V.: Software Defect Reduction Top 10 List. IEEE Computer, 34(1), 2001, 135-137.

Boehm, In (1996) Boehm, B.; In, H.: Identifying Quality-Requirement Conflicts. IEEE Software, 13(2), 1996, 25-35.

Boehm et al. (1978) Boehm, B.; Brown, J.; Kaspar, H.; Lipow, M.; MacLeod, G.; Merritt, M.: Characteristics of Software Quality. North Holland, Amsterdam, 1978.

Booch (1987) Booch, G.: Software Engineering with Ada (2. Auflage). Benjamin/ Cummings, Menlo Park, CA, 1987.

Booch (1994) Booch, G.: Object-Oriented Analysis and Design with Applications (2. Auflage). Benjamin/Cummings, Redwood City, CA, 1994.

Booch et al. (1998) Booch, G.; Rumbaugh, J.; Jacobson, I.: The Unified Modeling Language User Guide. Addison-Wesley, Reading, MA, 1998.

Bosch (2000) Bosch, J.: Design and Use of Software Architectures. Addison-Wesley, Harlow, 2000.

Bowen et al. (1984) Bowen, T.; Wigle, G.; Tsai, J.: Specification of Software Quality Attributes; Volumes I, II, and III. Boeing Report D182-11678-1, D182-11678-2 and D182-11678-3, 1984.

Box (1979) Box, G.: Some Problems of Statistics and Everyday Life. Journal of the American Statistical Association, 74(365), 1979, 1-4.

Briand et al. (1997) Briand, L.; Daly, J.; Wüst, J.: A Unified Framework for Cohesion Measurement in Object-Oriented Systems. ESE-Report 040.97/E, 1997.

Briand et al. (1998) Briand, L.; Wüst, J.; Lounis, H.: Investigating Quality Factors in Object-Oriented Design: An Industrial Case Study. Technical Report ISERN-98-29 (Version 2), 1998.

Briand et al. (1999) Briand, L.; Daly, J.; Wüst, J.: A Unified Framework for Coupling Measurement in Object-Oriented Systems. IEEE Transactions on Software Engineering, 25(1), 1999, 91-121.

Briand, Wüst (1999) Briand, L.; Wüst, J.: The Impact of Design Properties on Development Cost in Object-Oriented Systems. Technical Report ISERN-99-16, 1999.

Briand, Wüst (2001) Briand, L.; Wüst, J.: Integrating Scenario-Based and Measurement-Based Software Product Assessment. Journal of Systems and Software, 59(1), 2001, 3-22.

Broh (1974) Broh, R.: Managing Quality for Higher Profits. McGraw-Hill, New York, 1974.

Brooks (1987) Brooks, F.: No Silver Bullet: Essence and Accidents of Software Engineering. IEEE Computer, 20(4), 1987, 10-19.

Brown et al. (1998) Brown, W.; Malveau, R.; McCormick, H.; Mowbray, T.: AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis. Wiley, Chichester, 1998.

Budd (1991) Budd, T.: An Introduction to Object-Oriented Programming. Addison-Wesley, Reading, MA, 1991.

Budgen (1994) Budgen, D.: Software Design. Addison-Wesley, Reading, MA, 1994.

Bunge (1977) Bunge, M.: Ontology I: The Furniture of the World. Treatise on Basic Philosophy, Bd. 3. Reidel, Dordrecht, 1977.

Bunge (1979) Bunge, M.: Ontology II: The World of Systems. Treatise on Basic Philosophy, Bd. 4. Reidel, Dordrecht, 1979.

Burd, Munro (1997) Burd, E.; Munro, M.: Investigating the Maintenance Implications of the Replication of Code. Proceedings of the International Conference on Software Maintenance (ICSM'97), Bari, Italy. IEEE Computer Society Press, Los Alamitos, CA, 1997, 322-329.

Buschmann et al. (1996) Buschmann, F.; Meunier, R.; Rohnert, H.; Sommerlad, P.; Stal, M.: Pattern-Oriented Software Architecture: A System of Patterns. Wiley, Chichester, 1996.

Canfora et al. (1996) Canfora, G.; Mancini, L.; Tortorella, M.: A Workbench for Program Comprehension During Software Maintenance. Fourth Workshop on Program Comprehension, Berlin. IEEE Computer Society Press, Los Alamitos CA, 1996, 30-39.

Card, Glass (1990) Card, D.; Glass, R.: Measuring Software Design Quality. Prentice Hall, Englewood Cliffs, NJ, 1990.

Cardelli, Wegner (1985) Cardelli, L.; Wegner, P.: On Understanding Types, Data, Abstraction, and Polymorphism. Computing Surveys, 17(4), 1985, 471-522.

Cartwright (1998) Cartwright, M.: An Empirical View of Inheritance. Information and Software Technology, 40(14), 1998, 795-799.

Cavano, McCall (1978) Cavano, J.; McCall, J.: A Framework for the Measurement of Software Quality. Proceedings of the Software Quality and Assurance Workshop; Software Engineering Notes, 3(5), 1978, 133-139.

Cherniavsky, Smith (1991) Cherniavsky, J.; Smith, C.: On Weyuker's Axioms for Software Complexity Measures. IEEE Transactions on Software Engineering, 17(6), 1991, 636-638.

Chen, Lu (1993) Chen, J.; Lu, J.: A New Metric for Object-Oriented Design. Information and Software Technology, 35(4), 1993, 232-240.

Chidamber, Kemerer (1991) Chidamber, S.; Kemerer, C.: Towards a Metrics Suite for Object Oriented Design. Proceedings of OOPSLA'91; ACM SIGPLAN Notices, 26(11), 1991, 197-211.

Chidamber, Kemerer (1994) Chidamber, S.; Kemerer, C.: A Metrics Suite for Object Oriented Design. IEEE Transactions on Software Engineering, 20(6), 1994, 476-493.

Chidamber, Kemerer (1995) Chidamber, S.; Kemerer, C.: Authors' Reply to "Comments on A Metrics Suite for Object Oriented Design". IEEE Transactions on Software Engineering, 21(3), 1995, 265.

Chidamber et al. (1998) Chidamber, S.; Darcy, D.; Kemerer, C.: Managerial Use of Metrics for Object-Oriented Software: An Exploratory Analysis. IEEE Transactions on Software Engineering, 24(8), 1998, 629-639.

Churcher, Shepperd (1995a) Churcher, N.; Shepperd, M.: Towards a Conceptual Framework for Object-Oriented Software Metrics. ACM SIGSOFT Software Engineering Notes, 20(2), 1995, 69-76.

Churcher, Shepperd (1995b) Churcher, N.; Shepperd, M.: Comments on "A Metrics Suite for Object Oriented Design". IEEE Transactions on Software Engineering, 21(3), 1995, 263-265.

Clements et al. (2002) Clements, P.; Kazman, R.; Klein, M.: Evaluating Software Architectures: Methods and Case Studies. Addison-Wesley, Boston, 2002.

Coad, Yourdon (1991) Coad, P.; Yourdon, E.: Object Oriented Design. Prentice Hall, Englewood Cliffs, NJ, 1991.

Cockburn (1998) Cockburn, A.: Object-Oriented Analysis and Design: Part 2. C/C++ Users Journal, 16(6), 1998.

Constantine (1965) Constantine, L.: Towards a Theory of Program Design. Data Processing Magazine, 7(12), 1965, 18-21.

Constantine (1991) Constantine, L.: Larry Constantine on Structured Methods and Object Orientation. UNIX Review, 9(2), 1991, 409.

Conway (1968) Conway, M.: How Do Committees Invent? Datamation, 14(4), 1968, 28-31.

Coplien (1999) Coplien, J.: Reevaluating the Architectural Metaphor: Toward Piecemeal Growth. IEEE Software, 16(5), 1999, 40-44.

Coplien, Schmidt (1995) Coplien, J.; Schmidt, D. (Hrsg.): Pattern Languages of Program Design. Addison-Wesley, Reading, MA, 1995.

Corbi (1989) Corbi, T.: Program Understanding: Challenge for the 1990s. IBM Systems Journal, 28(2), 1989, 294-306.

Crosby (1979) Crosby, P.: Quality Is Free: The Art of Making Quality Certain. McGraw-Hill, New York, 1979.

Curtis et al. (1988) Curtis, B.; Krasner, H.; Iscoe, N.: A Field Study of the Software Design Process for Large Systems. Communications of the ACM, 31(11), 1988, 1268-1287.

Daly et al. (1996) Daly, J.; Brooks, A.; Miller, J.; Roper, M.; Wood, M.: Evaluating Inheritance Depth on the Maintainability of Object-Oriented Software. Empirical Software Engineering, 1(2), 1996, 109-132.

Daskalantonakis (1992) Daskalantonakis, M.: A Practical View of Software Measurement and Implementation Experience Within Motorola. IEEE Transactions on Software Engineering, 18(11), 1992, 998-1010.

Daskalantonakis (1994) Daskalantonakis, M.: Achieving Higher SEI Levels. IEEE Software, 11(7), 1994, 17-24.

Davis (1995) Davis, A.: 201 Principles of Software Engineering. McGraw-Hill, New York, 1995.

DeMarco (1982) DeMarco, T.: Controlling Software Projects: Management, Measurement, and Estimation. Prentice Hall, Englewood Cliffs, NJ, 1982.

DeMarco (1998) DeMarco, T.: Der Termin: Ein Roman über Projektmanagement. Hanser, München, 1998.

DGQ (1995) Deutsche Gesellschaft für Qualität e.V.: Begriffe zum Qualitätsmanagement. DGQ-Schrift Nr. 11-04 (6. Auflage), Frankfurt, 1995.

Dick, Hunter (1994) Dick, R.; Hunter, R.: Subjective Software Evaluation. In: Ross, M.; Brebbia, C.; Staples, G.; Stapleton, J. (Hrsg.): Software Quality Management II, Vol. 2: Building Quality into Software (Proceedings of the Second International Conference on Software Quality Management, SQM '94). Computational Mechanics Publications, Southampton, 1994, 321-334.

Dijsktra (1968) Dijkstra, E.: The Structure of the "THE"-Multiprogramming System. Communications of the ACM, 11(5), 1968, 341-346.

Dijkstra (1972) Dijkstra, E.: The Humble Programmer. Communications of the ACM, 15(10), 1972, 859-866.

DIN 55350, Teil 11 Deutsches Institut für Normung e.V.: DIN 55350-11: 1987-05: Qualitätsmanagement und Statistik, Teil 11: Begriffe des Qualitätsmanagements, 1987.

DIN 55350, Teil 12 Deutsches Institut für Normung e.V.: DIN 55350 Teil 12 Mrz 1989: Begriffe der Qualitätssicherung und Statistik, Teil 12: Merkmalsbezogene Begriffe, 1989.

DIN EN ISO 8402 Deutsches Institut für Normung e.V.: DIN EN ISO 8402:1995-08: Qualitätsmanagement - Begriffe, 1995.

Dißmann (1990) Dißmann, S.: Anforderungsflüsse in der Software-Entwicklung als Grundlage für die Qualitätssicherung. Forschungsbericht Nr. 362/1990, Fachbereich Informatik, Universität Dortmund, 1990.

Dörner (1976) Dörner, P.: Problemlösen als Informationsverarbeitung. Kohlhammer, Stuttgart, 1976.

Dromey (1996) Dromey, R.: Cornering the Chimera. IEEE Software, 13(1), 1996, 33-43.

Dumke (2000) Dumke, R.: Erfahrungen in der Anwendung eines allgemeinen objektorientierten Measurement Framework. In: Dumke, R.; Lehner, F. (Hrsg.): Software-Metriken.Gabler, Wiesbaden, 2000, 71-93.

Dunn (1984) Dunn, R.: Software Defect Removal. McGraw-Hill, New York, 1984.

Dunsmore et al. (2000) Dunsmore, A.; Roper, M.; Wood, M.: Object-Oriented Inspection in the Face of Delocalisation. Proceedings of the 22nd International Conference on Software Engineering, Limerick. ACM Press, New York, 2000, 467-476.

Dvorak, Moher (1991) Dvorak, J.; Moher, T.: A Feasibility Study of Early Class Hierarchy Construction in Object-Oriented Development. Empirical Studies of Programmers: Fourth Workshop. Ablex, Norwood, NJ, 1991, 23-35.

El Emam, Melo (2001) El Emam, K.; Melo, W.; Machado, J.: The Prediction of Faulty Classes Using Object-Oriented Design Metrics. Journal of Systems and Software, 56(1), 2001, 63-75.

Erni (1996) Erni, K.: Anwendung multipler Metriken bei der Entwicklung objektorientierter Frameworks. Krehl Verlag, Münster, 1996.

Erni, Lewerentz (1996) Erni, K.; Lewerentz, C.: Applying Design-Metrics to Object-Oriented Frameworks. Proceedings of the 3rd International Software Metrics Symposium. IEEE Computer Society Press, Los Alamitos, CA, 1996, 64-74.

Evans, Marciniak (1987) Evans, M.; Marciniak, J.: Software Quality Assurance and Management. Wiley, New York, 1987.

Fayad et al. (1999) Fayad, M.; Schmidt, D.; Johnson, R. (Hrsg.): Building Application Frameworks: Object-Oriented Foundations of Framework Design. Wiley, New York, 1999.

Fenton, Pfleeger (1996): Fenton, N.; Pfleeger, S.: Software Metrics: A Rigorous & Practical Approach (2. Auflage). Thomson Computer Press, London, 1996.

Fetcke (1995) Fetcke, T.: Softwaremetriken bei objektorientierter Programmierung. GMD-Studien, Band 259, Gesellschaft für Mathematik und Datenverarbeitung mbH (GMD), Sankt Augustin, 1995.

Fichman, Kemerer (1992) Fichman, R.; Kemerer, C.: Object-Oriented and Conventional Analysis and Design Methodologies: Comparison and Critique. IEEE Computer, 25(10), 1992, 22-39.

Firesmith (1995) Firesmith, D.: Inheritance Guidelines. Journal of Object-Oriented Programming, 8(2), 1995, 67-72.

Fowler (2001a) Fowler, M.: Avoiding Repetition. IEEE Software, 18(1), 2001, 97-99.

Fowler (2001b) Fowler, M.: Reducing Coupling. IEEE Software, 18(4), 2001, 102-104.

Fowler, Scott (1997) Fowler, M.; Kendall, S.: UML Distilled: Applying the Standard Object Modeling Language. Addison-Wesley, Reading, MA, 1997.

Fowler et al. (1999) Fowler, M.; Beck, K.; Brant, J.; Opdyke, W.; Roberts, D.: Refactoring: Improving the Design of Existing Code. Addison-Wesley, Reading, MA, 1999.

Frühauf et al. (2000) Frühauf, K.; Ludewig, J.; Sandmayr, H.: Software-Projektmanagement und -Qualitätssicherung (3. Auflage). vdf, Zürich, 2000.

Fujino (1999) Fujino, T.: Traditional Japanese Architecture Blends Beauty and Rationale. IEEE Software, 16(6), 1999, 101-103.

Gamma et al. (1995) Gamma, E.; Helm, R.; Johnson, R.; Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, Reading, MA, 1995.

Garvin (1984) Garvin, D.: What Does "Product Quality" Really Mean? Sloan Management Review, 25(3), 1984, 25-43.

Garvin (1988) Garvin, D.: Managing Quality: The Strategic and Competitive Edge. Free Press, New York, 1988.

Gelernter (1998) Gelernter, D.: Machine Beauty: Elegance and the Heart of Technology. BasicBooks, New York, 1998.

Genero et al. (2000) Genero, M.; Piattini, M.; Calero, C.: Early Measures for UML Class Diagrams. L'Objet, 6(4), 2000, 489-515.

Gibbon (1997) Gibbon, C.: Heuristics for Object-Oriented Design. Ph.D. Thesis, University of Nottingham, Nottingham, 1997.

Gibbon, Higgins (1996) Gibbon, C.; Higgins, C.: Teaching Object-Oriented Design with Heuristics. ACM SIGPLAN Notices, 31(7), 1996, 12-16.

Gilb (1988) Gilb, T.: Principles of Software Engineering Management. Addison-Wesley, Wokingham, 1988.

Gilmore (1974) Gilmore, H.: Product Conformance Cost. Quality Progress, June 1974.

Gillibrand, Liu (1998) Gillibrand, D.; Liu, K.: Quality Metrics for Object-Oriented Design. Journal of Object-Oriented Programming, 10(8), 1998, 56-59.

Gillies (1992) Gillies, A.: Software Quality: Theory and Management. Chapman & Hall, London, 1992.

Glass (1998) Glass, R.: Defining Quality Intuitively. IEEE Software, 15(3), 1998, 103-107.

Glass (1999) Glass, R.: On Design. IEEE Software, 16(2), 1999, 103-104.

Gosling et al. (1998) Gosling, J.; Joy, B.; Steele, G.: The Java Language Specification, Second Edition. Addison-Wesley, Reading, MA, 1998.

Grady (1997) Grady, R.: Successful Software Process Improvement. Prentice Hall, Upper Saddle River, NJ, 1997.

Grotehen, Dittrich (1997) Grotehen, T.; Dittrich, K.: The MeTHOOD Approach: Measures, Transformation Rules, and Heuristics for Object-Oriented Design. Technical Report ifi-97.09, Universität Zürich, 1997.

Gursaran, Roy (2002) Gursaran; Roy, G.: On the Applicability of Weyuker Property 9 to Object-Oriented Structural Inheritance Complexity Metrics. IEEE Transactions on Software Engineering, 27(4), 2001, 381-384.

Haag et al. (1996) Haag, S.; Raja, M.; Schkade, L.: Quality Function Deployment Usage in Software Development. Communications of the ACM, 39(1), 41-49, 1996.

Harrison et al. (2000a) Harrison, N.; Foote, B.; Rohnert, H. (Hrsg.): Pattern Languages of Program Design 4. Addison-Wesley, Reading, MA, 2000.

Harrison et al. (2000b) Harrison, R.; Counsell, S.; Nithi, R.: Experimental Assessment of the Effect of Inheritance on the Maintainability of Object-Oriented Systems. Journal of Systems and Software, 52(2-3), 2000, 173-179.

Haynes (1996) Haynes, P.: Detection and Prevention of Software Cancer in OO Systems. OOPSLA'96 Workshop on Software Metrics, San Jose, CA, 1996.

Henderson-Sellers (1996) Henderson-Sellers, B.: Object-Oriented Metrics. Prentice Hall, Englewood Cliffs, NJ, 1996.

Henderson-Sellers et al. (1993) Henderson-Sellers, B.; Moser, S.; Seehusen, S.; Weinelt, B.: A Proposed Multi-Dimensional Framework for Object-Oriented Metrics. Proceedings of the First Australian Software Metrics Conference, Sydney, November 1993, 24-30.

Hitz, Neuhold (1998) Hitz, M.; Neuhold, K.: A Framework for Product Analysis. Proceedings of the OOPSLA 1998 Workshop on Model Engineering, Methods, and Tool Interaction with CDIF, Vancouver, 1998.

Hoare (1981) Hoare, C. A. R.: The Emperor's Old Clothes. Communications of the ACM, 24(2), 1981, 75-83.

Hofmeister et al. (2000) Hofmeister, C.; Nord, R.; Soni, D.: Applied Software Architecture. Addison-Wesley, Reading, MA, 2000.

Hopkins (1994) Hopkins, T.: Complexity Metrics for Quality Assessment of Object-Oriented Designs. In: Ross, M.; Brebbia, C.; Staples, G.; Stapleton, J. (Hrsg.): Software Quality Management II, Vol. 2: Building Quality into Software (Proceedings of the Second International Conference on Software Quality Management, SQM'94). Computational Mechanics Publications, Southampton, 1994, 467-481.

Humphrey (1988) Humphrey, W.: Characterizing the Software Process: A Maturity Framework. IEEE Software, 5(3), 1988, 73-79.

Humphrey (1990) Humphrey, W.: Managing the Software Process. Addison-Wesley, Reading, MA, 1990.

Hunt, Thomas (1999) Hunt, A.; Thomas, D.: The Pragmatic Programmer: From Journeyman to Master. Addison-Wesley, Reading, MA, 1999.

Huston (2001) Huston, B.: The Effects of Design Pattern Application on Metric Scores. Journal of Systems and Software, 58(3), 2001, 261-269.

IEEE Std. 610.12-1990 IEEE: IEEE Standard Glossary of Software Engineering Terminology. IEEE Std. 610.12-1990.

IEEE Std. 1028-1997 IEEE: IEEE Standard for Software Reviews. IEEE Std. 1028-1997.

IEEE Std. 1044-1993. IEEE: IEEE Standard for Classification of Software Anomalies. IEEE Std. 1044-1993.

IEEE Std. 1061-1992 IEEE: IEEE Standard for a Software Quality Metrics Methodology. IEEE Std. 1061-1992.

IEEE Std. 1471-2000 IEEE: IEEE Recommended Practice for Architectural Description of Software-Intensive Systems. IEEE Std. 1471-2000.

Ishikawa (1989) Ishikawa, K.: Guide to Quality Control. Quality Resource, White Plains, NY, 1989.

ISO/IEC 8652:1995 Intermetrics, Inc.: Ada 95 Reference Manual: The Language, The Standard Libraries. ANSI/ISO/IEC-8652:1995.

ISO/IEC 9126:1991 ISO/IEC: Information Technology - Software Product Evaluation - Quality Characteristics and Guidelines for Their Use. ISO/IEC 9126:1991.

ISO/IEC Guide 25 (1990) ISO/IEC: General Requirements for the Competence of Calibration and Testing Laboratories. ISO/IEC Guide 25, 1990.

Jackson (1975) Jackson, M.: Principles of Program Design. Academic Press, London, 1975.

Jacobson et al. (1995) Jacobson, I.; Christerson, M.; Jonsson, P.; Övergaard, G.: Object-Oriented Software Engineering: A Use Case Driven Approach. Addison-Wesley, Reading, MA, 1995.

Jacobson et al. (1998) Jacobson, I.; Booch, G.; Rumbaugh, J.: The Unified Software Development Process. Addison-Wesley, Reading, MA, 1998.

Johnson, Foote (1988) Johnson, R.; Foote, B.: Designing Reusable Classes. Journal of Object-Oriented Programming, 1(2), 1988, 22-35.

Jones (1992) Jones, J.: Design Methods (2. Auflage). Van Nostrand Reinhold, New York, 1992.

Jones (1996) Jones, C.: Applied Software Measurement: Assuring Productivity and Quality (2. Auflage). McGraw-Hill, New York, 1996.

Jones (1997) Jones, C.: Software Quality: Analysis and Guidelines for Success. Thomson, London, 1997.

Juran (1974) Juran, J. (Hrsg.): Quality Control Handbook (3. Auflage). McGraw-Hill, New York, 1974.

Kafura (1998) Kafura, D.: Object-Oriented Software Design and Construction with C++. Prentice Hall, Upper Saddle River, NJ, 1998.

Kernighan, Plauger (1974) Kernighan, B.; Plauger, P.: The Elements of Programming Style. McGraw-Hill, New York, 1974.

Kitchenham (1990) Kitchenham, B.: Software Metrics. In: Rook, P. (Hrsg.): Software Reliability Handbook, Elsevier, London, 1990.

Kitchenham et al. (1990) Kitchenham, B.; Pickard, L.; Linkman, S.: An Evaluation of Some Design Metrics. Software Engineering Journal, 5(1), 1990, 50-58.

Koenig (1995) Koenig, A.: Patterns and Antipatterns. Journal of Object-Oriented Programming, 8(1), 1995, 46 - 48.

Kogure, Akao (1983) Kogure, M.; Akao, Y.: Quality Function Deployment and CWQC in Japan. Quality Progress, October 1983.

Köhler et al. (1998) Köhler, G.; Rust, H.; Simon, F.: An Assessment of Large Object Oriented Software Systems. Proceedings of the Object-Oriented Product Metrics Workshop at ECOOP'98, Montreal, 1998, 36-41.

Kolewe (1993) Kolewe, R.: Metrics in Object-Oriented Design and Programming. Software Development, October 1993, 53-62.

Korson, McGregor (1990) Korson, T.; McGregor, J.: Understanding Object-Oriented: A Unifying Paradigm. Communications of the ACM, 33(9), 1990, 40-60.

KPMG (1995) KPMG: Runaway Projects: Causes and Effects. Software World (UK), 26(3), 1995.

Kriz (1988) Kriz, J.: Facts and Artefacts in Social Science: An Ephistemological and Methodological Analysis of Empirical Social Science Research Techniques. McGraw-Hill, Hamburg, 1988.

Kruchten (1994) Kruchten, P.: The 4+1 View Model of Architecture. IEEE Software, 11(6), 1994, 42-50.

Laitenberger et al. (2000) Laitenberger, O.; Atkinson, C.; Schlich, M.; El Emam, K.: An Experimental Comparison of Reading Techniques for Defect Detection in UML Design Documents. Technical Report ISERN-00-01, 2000.

Lakos (1996) Lakos, J.: Large-Scale C++ Software Design. Addison-Wesley, Reading, MA, 1996.

Lea (1994) Lea, D.: Christopher Alexander: An Introduction for Object-Oriented Designers. ACM SIGSOFT Software Engineering Notes, 19(1), 1994.

Lehman (1980) Lehman, M.: Programs, Life Cycles, and Laws of Software Evolution. Proceedings of the IEEE, 68(9), 1980, 1060-1076.

Lejter et al. (1992) Lejter, M.; Meyers, S.; Reiss, S.: Support for Maintaining Object-Oriented Programs. Transactions on Software Engineering, 18(12), 1992, 1045-1052.

Lewerentz et al. (2000) Lewerentz, C.; Rust, H.; Simon, F.: Quality - Metrics - Numbers - Consequences: Lessons Learned. In: Dumke, R.; Lehner, F. (Hrsg.): Software-Metriken. Gabler, Wiesbaden, 2000, 51-70.

Li (1992) Li, W.: Applying Software Maintenance Metrics in the Object-Oriented Software Development Life Cycle. Ph.D. Dissertation, Virginia Polytechnic Institute and State University, Blacksburg, VA, 1992.

Li (1998) Li, W.: Another Metric Suite for Object-Oriented Programming. Journal of Systems and Software, 44(2), 1998, 155-162.

Li, Henry (1993) Li, W.; Henry, S.: Object-Oriented Metrics that Predict Maintainability. Journal of Systems and Software, 23(2), 1993, 111-122.

Lieberherr et al. (1988) Lieberherr, K.; Holland, I.; Riel, A.: Object-Oriented Programming: An Objective Sense of Style. Proceedings of OOPSLA'88; ACM SIGPLAN Notices, 23(11), 1988, 323-334.

Lieberherr, Holland (1989) Lieberherr, K.; Holland, I.: Assuring Good Style for Object-Oriented Programs. IEEE Software, 6(5), 1989, 38-48.

Lientz, Swanson (1980) Lientz, B.; Swanson, E.: Software Maintenance Management. Addison-Wesley, Reading, MA, 1980.

Linger et al. (1979) Linger, R.; Mills, H.; Witt, B.: Structured Programming: Theory and Practice. Addison-Wesley, Reading, MA, 1979.

Liskov (1988) Liskov, B.: Data Abstraction and Hierarchy. ACM SIGPLAN Notices, 23(5), 1988, 17-34.

Lorenz, Kidd (1994) Lorenz, M.; Kidd, J.: Object-Oriented Software Metrics: A Practical Guide. Prentice Hall, Englewood Cliffs, NJ, 1994.

Ludewig (1994) Ludewig, J.: People Make Quality Happen (or Don't). Proceedings of the 4th European Conference on Software Quality. vdf, Zürich, 1994, 11-21.

Ludewig (1998) Ludewig, J.: Software Engineering: Vorläufiges, unvollständiges Skript zur Vorlesung Software Engineering an der Fakultät Informatik der Universität Stuttgart, Dezember 1998.

Marchesi (1998) Marchesi, M.: OOA Metrics for the Unified Modeling Language. Proceedings of the 2nd Euromicro Conference on Software Maintenance and Reengineering (CSMR'98), Palazzo degli Affari, Italy, 1998, 67-73.

Martin (1995) Martin, R.: Designing Object-Oriented C++ Applications Using the Booch Method. Prentice Hall, Englewood Cliffs, NJ, 1995.

Martin (1996a) Martin, R.: The Open-Closed Principle. C++ Report, 8(1), 1996.

Martin (1996b) Martin, R.: The Liskov Substitution Principle. C++ Report, 8(3), 1996.

Martin (1996c) Martin, R.: The Dependency Inversion Principle. C++ Report, 8(5), 1996.

Martin (1996d) Martin, R.: The Interface Segregation Principle. C++ Report, 8(8), 1996.

Martin (1996e) Martin, R.: Granularity. C++ Report, 8(11), 1996.

Martin et al. (1998) Martin, R.; Riehle, D.; Buschmann, F. (Hrsg.) (1998): Pattern Languages of Program Design 3. Addison-Wesley, Reading, MA.

Mattsson et al. (1999) Mattsson, M.; Bosch, J.; Fayad, M.: Framework Integration: Problems, Causes, Solutions. Communications of the ACM, 42(10), 81-87, 1999.

Mayrand et al. (1996) Mayrand, J.; Guay, F.; Merlo, E.: Inheritance Graph Assessment Using Metrics. Proceedings of the 3rd International Software Metrics Symposium, Berlin, 1996. IEEE Computer Society Press, Los Alamitos, CA, 1996, 54-63.

McBreen (2000) McBreen, P.: OO Design Inspection Checklist.
http://www.mcbreen.ab.ca/papers/QAOODesign.html, Version vom 25.04.2000.

McBreen (2001) McBreen, P.: Software Craftsmanship. Addison Wesley, Boston, 2001.

McCabe (1976) McCabe, T.: A Complexity Measure. IEEE Transactions on Software Engineering, 2(4), 1976, 308-320.

McCall et al. (1977) McCall, J.; Richards, P.; Walters, G.: Factors in Software Quality; Volumes I, II, and III. US Rome Air Development Center Report NTIS AD/A-049 014, NTIS AD/A-049 015 and NTIS AD/A-049 055, 1977.

Melton et al. (1990) Melton, A.; Gustafson, D.; Bieman, J.; Baker, L.: A Mathematical Perspective for Software Measures Research. Software Engineering Journal, 5(5), 1990, 246-254.

Meyer (1991) Meyer, B.: Eiffel: The Language. Prentice Hall, Upper Saddle River, NJ, 1991.

Meyer (1996) Meyer, B.: The Many Faces of Inheritance: A Taxonomy of Taxonomy. IEEE Computer, 29(5), 1996, 105-108.

Meyer (1997) Meyer, B.: Object-Oriented Software Construction (2. Auflage). Prentice Hall, Upper Saddle River, NJ, 1997.

Meyer (2001) Meyer, B.: Software Engineering in the Academy. IEEE Computer, 34(5), 2001, 28-35.

Mills (1980) Mills, H.: The Management of Software Engineering: Part I: Principles of Software Engineering. IBM Systems Journal, 19(4), 1980, 414-420.

Morschel (1994) Morschel, I.: Applying Object-Oriented Metrics to Enhance Software Quality. In: Dumke, R.; Zuse, H. (Hrsg.): Theorie und Praxis der Softwaremessung. Deutscher Universitäts-Verlag, Wiesbaden, 1994, 97-110.

Nelson (1990) Nelson, T.: The Right Way to Think About Software Design. In: Laurel, B. (Hrsg.): The Art of Human-Computer Interface Design. Addison-Wesley, Reading, MA, 1990.

Nenonen et al. (2000) Nenonen, L.; Gustafsson, J.; Paakki, J.; Verkamo, A. (2000): Measuring Object-Oriented Software Architectures from UML Diagrams. Proceedings of the 4th International ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering, Sophia Antipolis, France, 2000, 87-100.

Oestereich (1998) Oestereich, B.: Objektorientierte Softwareentwicklung: Analyse und Design mit der Unified Modeling Language (4. Auflage). Oldenbourg, München, 1998.

OMG (2000a) Object Management Group: OMG Unified Modeling Language Specification, Version 1.3, March 2000.

OMG (2000b) Object Management Group: OMG XML Metadata Interchange (XMI) Specification, Version 1.1, November 2000.

Page-Jones (1988) Page-Jones, M.: The Practical Guide to Structured Systems Design (2. Auflage). Prentice Hall, Englewood Cliffs, New Jersey, 1988.

Page-Jones (1995) Page-Jones, M.: What Every Programmer Should Know About Object-Oriented Design. Dorset House, New York, 1995.

Pancake (1995) Pancake, C.: The Promise and the Cost of Object Technology: A Five-Year Forecast. Communications of the ACM, 38(10), 1995, 32-49.

Parnas (1972a) Parnas, D.: Information Distribution Aspects of Design Methodology. In: Freiman, C. (Hrsg.): Information Processing 71, Volume I - Foundations and Systems. North Holland Publishing Company, Amsterdam, 1972, 339-344.

Parnas (1972b) Parnas, D.: On the Criteria to be Used in Decomposing Systems into Modules. Communications of the ACM, 15(12), 1972, 1053-1058.

Parnas (1974) Parnas, D.: On a 'Buzzword': Hierarchical Structure. In: Rosenfeld, J. (Hrsg.): Information Processing 74. North Holland Publishing Company, Amsterdam, 1974, 336-340.

Parnas (1994) Parnas, D.: Software Aging. Proceedings of the 16th International Conference on Software Engineering (ICSE-16), Sorrento, Italy. IEEE Computer Society Press, Los Alamitos, CA, 1994, 279-287.

Parnas, Clements (1986) Parnas, D.; Clements, P.: A Rational Design Process: How and Why to Fake It. IEEE Transactions on Software Engineering, 12(2), 1986, 251-257.

Petroski (1992) Petroski, H.: To Engineer is Human: The Role of Failure in Successful Design.Vintage Books, New York, 1992.

Petroski (1994) Petroski, H.: Design Paradigms: Case Histories of Error and Judgment in Engineering. Cambridge University Press, Cambridge, 1994.

Pfleeger (2000) Pfleeger, S.: Use Realistic, Effective Software Measurement. In: Clements, P. (Hrsg.): Constructing Superior Software. MacMillan, Indianapolis, IN, 2000, 211-236.

Pirsig (1981) Pirsig, R.: Zen and the Art of Motorcycle Maintenance. Bantam, New York, 1981.

Pree (1995) Pree, W.: Design Patterns for Object-Oriented Software Development. Addison-Wesley, 1995.

Pressman (1994) Pressman, R.: Software Engineering, European Edition (3. Auflage). McGraw-Hill, New York, 1994.

Quibeldey-Circel (1999) Quibeldey-Circel, K.: Entwurfsmuster: Design Patterns in der objektorientierten Softwaretechnik. Springer, Berlin, 1999.

Reißing (2001a) Reißing, R.: Towards a Model for Object-Oriented Design Measurement. Proceedings of the 5th International ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering (QAOOSE 2001), Budapest, 2001, 71-84.

Reißing (2001b) Reißing, R.: The Impact of Pattern Use on Design Quality. Position Paper, Workshop „Beyond Design: Patterns (mis)used" at OOPSLA 2001.

Reißing (2002) Reißing, R.: Entwurfsregeln für den objektorientierten Entwurf. http://www.informatik.uni-stuttgart.de/ifi/se/service/design_rules/index.html, Version vom 11.01.2002.

Rentsch (1982) Rentsch, T.: Object Oriented Programming. ACM SIGPLAN Notices, 17(9), 1982, 51-57.

Riel (1996) Riel, A.: Object-Oriented Design Heuristics. Addison-Wesley, Reading, MA, 1996.

Rising (2000) Rising, L.: The Pattern Almanac 2000. Addison-Wesley, Reading, MA, 2000.

Rittel, Webber (1984) Rittel, H.; Webber, M.: Planning Problems are Wicked Problems. In: Cross, N. (Hrsg.): Developments in Design Methodology. Wiley, London, 1984, 143-159.

Robbins (1998) Robbins, J.: Design Critiquing Systems. Technical Report UCI-98-41, University of California, Irvine, 1998.

Robbins, Redmiles (1999) Robbins, J.; Redmiles, D.: Cognitive Support, UML Adherence, and XMI Interchange in Argo/UML. Proceedings of the First International Symposium on the Construction of Software Engineering Tools (CoSET'99), Los Angeles, May 1999.

Roche, Jackson, 1994 Roche, J.; Jackson, M.: Software Measurement Methods: Recipes for Success? Information and Software Technology, 36(3), 1994, 173-189.

Rombach (1990) Rombach, H.: Design Measurement: Some Lessons Learned. IEEE Software, 7(2), 1990, 17-25.

Rombach (1993) Rombach, H.: Software-Qualität und -Qualitätssicherung. Informatik-Spektrum, 16(5), 1993, 267-272.

Ross et al. (1975) Ross, D.; Goodenough, J.; Irvine, C.: Software Engineering: Process, Principles, and Goals. IEEE Computer, 14(5), 1975, 17-27.

Royce (1970) Royce, W.: Managing the Development of Large Software Systems: Concepts and Techniques. Proceedings of the IEEE WESCON, Los Angeles, 1970, 1-9.

Royce (2000) Royce, W.: Software Management Renaissance. IEEE Software, 17(4), 2000, 116-121.

Rumbaugh et al. (1993) Rumbaugh, J.; Blaha, M.; Premerlani, W.; Eddy, F.; Lorensen, W.: Objektorientiertes Modellieren und Entwerfen. Hanser, München, 1993.

Rumbaugh et al. (1998) Rumbaugh, J.; Jacobson, I.; Booch, G.: The Unified Modeling Language Reference Manual. Addison-Wesley, Reading, MA, 1998.

Schmider (2002) Schmider, C.: Konzeption und Realisierung eines Metrikenwerkzeugs für die Unified Modeling Language. Diplomarbeit Nr. 1952, Institut für Informatik, Universität Stuttgart, 2002.

Shaw, Garlan (1996) Shaw, M.; Garlan, D.: Software Architecture: Perspectives on an Emerging Discipline. Prentice Hall, Upper Saddle River, NJ, 1996.

Shepperd, Ince (1993) Shepperd, M.; Ince, D.: Derivation and Validation of Software Metrics. Clarendon Press, Oxford, 1993.

Shull et al. (1999) Shull, F.; Travassos, G.; Basili, V.: Towards Techniques for Improved Design Inspections. Proceedings of the 3rd Workshop on Quantitative Approaches in Object-Oriented Software Engineering (QAOOSE 1999), Lisbon, 1999.

Simon (1962) Simon, H.: The Architecture of Complexity. Proceedings of the American Philosophical Society, 106(6), 1962, 467-482.

Simon et al. (2001) Simon, F.; Steinbrückner, F.; Lewerentz, C.: Anpassbare, explorierbare virtuelle Informationsräume zur Qualitätsbewertung großer Software-Systeme: Erste Erfahrungen. Proceedings of the Third Workshop on Reengineering, Bad Honnef, 2001.

Slaughter, Banker (1996) Slaughter, S.; Banker, R.: A Study of the Effects of Software Development Practices on Software Maintenance Effort. Proceedings of the International Conference on Software Maintenance (ICSM'96), Monterey, CA. IEEE Computer Society Press, Los Alamitos, 1996, 197-205.

Smith, Robson (1990) Smith, M.; Robson, D.: Object-Oriented Programming. Proceedings of the Conference on Software Maintenance, San Diego, CA. IEEE Computer Society Press, Los Alamitos, CA, 1990, 272-281.

Sneed (1988) Sneed, H.: Einleitung zu Wix, B.; Balzert, H. (Hrsg.): Softwarewartung. BI-Wissenschaftsverlag, Mannheim, 1988, 11-19.

Snyder (1986) Snyder, A.: Encapsulation and Inheritance in Object-Oriented Programming Languages. Proceedings of OOPSLA'86; ACM SIGPLAN Notices, 21(11), 1986, 38-45.

Snyder (1993) Snyder, A.: The Essence of Objects: Concepts and Terms. IEEE Software, 10(1), 1993, 31-42.

Soloway et al. (1988) Soloway, E.; Pinto, J.; Letovsky, S.; Littman, D.; Lampert, R.: Designing Documentation to Compensate for Delocalized Plans. Communications of the ACM, 31(11), 1988, 1259-1267.

Stachowiak (1973) Stachowiak, H.: Allgemeine Modelltheorie. Springer, Wien, 1973.

Stevens et al. (1974) Stevens, W.; Myers, G.; Constantine, L.: Structured Design. IBM Systems Journal, 13(2), 1974, 115-139.

Stroustrup (1997) Stroustrup, B.: The C++ Programming Language (3. Auflage). Addison-Wesley, Reading, MA, 1997.

Swartout, Balzer (1982) Swartout, W.; Balzer, R.: On the Inevitable Intertwining of Specification and Implementation. Communications of the ACM, 25(7), 1982, 438-440.

Taivalsaari (1996) Taivalsaari, A.: On the Notion of Inheritance. ACM Computing Surveys, 28(3), 1996, 438-479.

Tegarden et al. (1995) Tegarden, D.; Sheetz, S.; Monarchi, D.: A Software Complexity Model of Object-Oriented Systems. Decision Support Systems, 13(3,4), 1995, 241-262.

Troy, Zweben (1981) Troy, D.; Zweben, S.: Measuring the Quality of Structured Designs. Journal of Systems and Software, 2, 1981, 113-120.

Visser, Hoc (1990) Visser, W.; Hoc, J.: Expert Software Design Strategies. In: Hoc, J.; Green, T.; Samurçai, R.; Gilmore, D. (Hrsg.): Psychology of Programming. Academic Press, London, 1990, 235-249.

Vlissides et al. (1996) Vlissides, J.; Coplien, J.; Kerth, N. (Hrsg.): Pattern Languages of Program Design 2. Addison-Wesley, Reading, MA, 1996.

Wand (1989) Wand, Y.: A Proposal for a Formal Model of Objects. In: Kim, W.; Lochovsky, F. (Hrsg.): Objects-Oriented Concepts, Applications and Databases. Addison-Wesley, Reading, MA, 1989, 537-559.

Warmer, Kleppe (1999) Warmer, J.; Kleppe, A.: The Object Constraint Language: Precise Modeling with UML. Addison-Wesley, Boston, 1999.

Webster (1995) Webster, B. F.: Pitfalls of Object-Oriented Development. M&T Books, New York, 1995.

Wegner (1987) Wegner, P.: Dimensions of Object-Based Language Design. Proceedings of OOPSLA'87; ACM SIGPLAN Notices, 22(12), 1987, 168-182.

Wegner (1992) Wegner, P.: Dimensions of Object-Oriented Modeling. IEEE Computer, 25(10), 1992, 12-19.

Weinand et al. (1989) Weinand, A.; Gamma, E.; Marty, R.: Design and Implementation of ET++, a Seamless Object-Oriented Application Framework. Structured Programming, 10(2), 1989, 63-87.

Weinand, Gamma (1994) Weinand A., Gamma E.: ET++ – a Portable, Homogenous Class Library and Application Framework. In: Bischofberger, W.; Frei, H. (Hrsg.): Computer Science Research at UBILAB: Strategy and Projects; Proceedings of the UBILAB '94 Conference, Zurich. Universitätsverlag Konstanz, Konstanz, 1994, 66-92.

Weinberg (1971) Weinberg, G.: The Psychology of Computer Programming. Van Nostrand Reinhold, New York, 1971.

Weinberg (1991) Weinberg, G.: First-Order Measurement. Quality Software Management, Bd. 2. Dorset House, New York, 1991.

Weiss, Basili (1985) Weiss, D.; Basili, V.: Evaluating Software Development by Analysis of Changes. IEEE Transactions on Sofware Engineering, 2(2), 1985, 157-168.

Weyuker (1988) Weyuker, E.: Evaluating Software Complexity Measures. IEEE Transactions on Software Engineering, 14(9), 1988, 1357-1365.

Whitmire (1994) Whitmire, S.: Object-Oriented Measurement of Software. In: Marciniak, J. (Hrsg.): Encyclopedia of Software Engineering. Wiley, New York, 1994, 737-739.

Whitmire (1997) Whitmire, S.: Object-Oriented Design Measurement. Wiley, New York, 1997.

Wilde, Huitt (1992) Wilde, N.; Huitt, R.: Maintenance Support for Object-Oriented Programs. Transactions on Software Engineering, 18(12), 1992, 1038-1044.

Wilde et al. (1993) Wilde, N.; Matthews, P.; Huitt, R.: Maintaining Object-Oriented Software. IEEE Software, 10(1), 1993, 75-80.

Winograd et al. (1996) Winograd, T.; Bennett, J.; De Young, L.; Hartfield, B. (Hrsg.): Bringing Design to Software. Addison-Wesley, Reading, MA, 1996.

Witt et al. (1994) Witt, B.; Baker, F.; Merritt, E.: Software Architecture and Design: Principles, Models, and Methods. Van Nostrand Reinhold, New York, 1994.

Würthele (1995) Würthele, V.: Checklisten für die Software-Bearbeitung. Diplomarbeit Nr. 1299, Fakultät für Informatik, Universität Stuttgart, 1995.

Yau, Tsai (1986) Yau, S.; Tsai, J.: A Survey of Software Design Techniques. IEEE Transactions on Software Engineering, 12(6), 1986, 713-721.

Yourdon (1995) Yourdon, E.: When Good Enough Software is Best. IEEE Software, 12(3), 1995, 79-81.

Yin, Winchester (1978) Yin, B.; Winchester, J.: The Establishment and Use of Measures to Evaluate the Quality of Software Designs. Proceedings of the Software Quality and Assurance Workshop; Software Engineering Notes, 3(5), 1978, 45-52.

Zuse (1994) Zuse, H.: Complexity Metrics. In: Marciniak, J. (Hrsg.): Encyclopedia of Software Engineering. Wiley, New York, 1994, 131-165.

Zweben et al. (1995) Zweben, S.; Edwards, S.; Weide, B.; Hollingsworth, J.: The Effects of Layering and Encapsulation on Software Development Cost and Quality. IEEE Transactions on Software Engineering, 21(3), 1995, 200-208.

Akronyme

Allgemeine Akronyme

CMM	Capability Maturity Model
DGQ	Deutsche Gesellschaft für Qualität
DIN	Deutsches Institut für Normung
FCM	Factors-Criteria-Metrics
GQM	Goal-Question-Metric
IEC	International Electrotechnical Commision
IEEE	Institute of Electrical and Electronical Engineers
ISO	International Organization for Standardization
MOOSE	Metrikenwerkzeug für den objektorientierten Systementwurf
OCL	Object Constraint Language
ODEM	Object-Oriented Design Model
OMG	Object Management Group
OOA	Object-Oriented Analysis
OOD	Object-Oriented Design
QFD	Quality Function Deployment
QOOD	Quality Model for Object-Oriented Design
UML	Unified Modeling Language
XMI	XML Metadata Interchange
XML	Extended Markup Language

Metrikakronyme (Literatur)

ANA	Average Number of Ancestors
CAM	Cohesion among Methods in Class
CBO	Coupling between Object Classes
CIS	Class Interface Size
DAM	Data Access Metric
DCC	Direct Class Coupling
DIT	Depth of Inheritance Tree
DSC	Design Size in Classes
LCOM	Lack of Cohesion in Methods
MFA	Measure of Functional Abstraction
MOA	Measure of Aggregation
NOC	Number of Children
NOH	Number of Hierarchies

NOM	Number of Methods
RFC	Response for a Class
SI	Specialization Index
WMC	Weighted Methods per Class

Metrikakronyme (QOOD)

DITC	depth of inheritance tree of a class
DITS	depth of inheritance tree of the system
DNHP	depth in nesting hierarchy of a package
DNHS	depth of nesting hierarchy of the system
MNCS	maximum number of child classes in the system
MNPS	maximum number of subpackages in the system
NAC	number of attributes of a class
NACP	number of afferently coupled packages of a package
NADC	number of afferent dependencies of a class
NADP	number of afferent dependencies of a package
NAS	number of attributes in the system
NCP	number of classes in a package
NCS	number of classes in the system
NEAC	number of efferent association relationships of a class
NECP	number of efferently coupled packages of a package
NEDC	number of efferent dependencies of a class
NEDP	number of efferent dependencies of a package
NEEC	number of extends relationships of a class
NERC	number of efferent realization relationships of a class
NEUC	number of efferent uses relationships of a class
NIP	number of interfaces in a package
NIS	number of interfaces in the system
NOC	number of operations of a class
NOS	number of operations in the system
NPP	number of packages in a package
NPS	number of packages in the system
RTTR	ratio of traceable to total requirements
SCCx	subjective conciseness of a Class/a Package/a System
SCOx	subjective cohesion of a Class/a Package/a System
SCSx	subjective consistency of a Class/a Package/a System
SDCx	subjective decoupling of a Class/a Package/a System
SDOx	subjective documentation of a Class/a Package/a System
SMAx	subjective maintainability of a Class/a Package/a System
SSTx	subjective structuredness of a Class/a Package/a System
STRx	subjective traceability of a Class/a Package/a System

Anhang A

Metriken für QOOD

Dieser Anhang stellt die objektiven Metriken für QOOD (Übersicht siehe Tabelle 9-1) im Detail vor und begründet ihre Auswahl. Die Metriken werden geordnet nach den Entwurfskriterien des Faktors Wartbarkeit (siehe Abbildung 8-1) präsentiert. Innerhalb eines Kriteriums sind sie nach Ebenen sortiert, wobei mit der untersten Ebene (Klassen/Interfaces) begonnen wird. Wenn für eine Ebene oder ein Kriterium keine objektive Metrik verfügbar ist, wird eine Begründung dafür angegeben. Im Anschluss an die Präsentation der Metriken wird gezeigt, wie die Metriken theoretisch validiert werden können (siehe Abschnitt A.9).

A.1 Knappheit

Knappheit bedeutet eine möglichst geringe Zahl an Modellelementen. Daher zählt die Quantifizierung zunächst diese Modellelemente. Briand und Wüst (1999) haben festgestellt, dass solche einfachen Größenmetriken die besten Indikatoren für den späteren Implementierungsaufwand des Entwurfs darstellen. Je niedriger die Werte der Zählmetriken sind, um so besser ist die Knappheit (negative Korrelation).

Klasse/Interface

Metriken für Eigenschaften. Bei den Klassen zählt man Attribute und Operationen. Da geerbte Eigenschaften in der Klasse vorhanden sind und damit ihre Größe mitbestimmen, zählen sie mit. Eine solche Vorgehensweise wird auch von den Untersuchungen von Beyer et al. (2000) gestützt, denen zufolge Größen-, Kopplungs- und Zusammenhaltsmetriken bei der Berücksichtigung geerbter Eigenschaften besser interpretierbare Werte liefern.

NAC (number of attributes of a class)
$NAC(c) = |\{a \in A: \mathit{has}^*(c,a)\}|$

Bei Interfaces ist gemäß der Definition im UML-Metamodell NAC immer 0, weil Interfaces keine Attribute haben dürfen.

NOC (number of operations of a class)
$NOC(c) = |\{o \in O: has^*(c,o)\}|$

Diese Zählmetriken können verfeinert werden, um spezielle Charakteristika der Eigenschaften zu berücksichtigen. Das wird hier am Beispiel von NAC demonstriert, bei NOC funktioniert es analog. Die erste Verfeinerung berücksichtigt den Sichtbarkeitsbereich des Attributs (1 = public, 2 = protected, 3 = private). Diese Verfeinerung ist praktisch, um die öffentliche Schnittstelle oder die Vererbungsschnittstelle einer Klasse beurteilen zu können.

NAC_1 (number of public attributes of a class)
$NAC_1(c) = |\{a \in A: has^*(c,a) \wedge a.visibility = \text{public}\}|$

NAC_2 (number of protected attributes of a class)
$NAC_2(c) = |\{a \in A: has^*(c,a) \wedge a.visibility = \text{protected}\}|$

NAC_3 (number of private attributes of a class)
$NAC_3(c) = |\{a \in A: has^*(c,a) \wedge a.visibility = \text{private}\}|$

Die zweite Verfeinerung betrachtet die Zugehörigkeit des Attributs zur Klasse (c = *ownerScope* classifier) oder zum Objekt (o = *ownerScope* instance). Klassenattribute tragen in der Regel nicht so viel zur Komplexität der Methodenimplementierungen bei, können aber andererseits erhöhte Kopplung mit sich bringen, da sie wie globale Variable verwendet werden können.

NAC_c (number of class attributes of a class)
$NAC_c(c) = |\{a \in A: has^*(c,a) \wedge a.ownerScope = \text{classifier}\}|$

NAC_o (number of object attributes of a class)
$NAC_o(c) = |\{a \in A: has^*(c,a) \wedge a.ownerScope = \text{instance}\}|$

Schließlich kann auch nach lokal definierten (l = local) und geerbten (i = inherited) Attributen unterschieden werden. Geerbte Attribute tragen in der Regel, insbesondere bei eingeschränkter Sichtbarkeit, nicht soviel zur Komplexität einer Klasse bei wie lokal definierte Attribute.

NAC_l (number local attributes of a class)
$NAC_l(c) = |\{a \in A: has(c,a)\}|$

NAC_i (number of inherited attributes of a class)
$NAC_i(c) = NAC(c) - NAC_l(c) = |\{a \in A: has^*(c,a) \wedge \neg has(c,a)\}|$

Die Verfeinerungen lassen sich kombinieren, so dass man z. B. die Metrik $NAC_{1,c,i}$ als die Anzahl der öffentlich sichtbaren, geerbten Klassenattribute einführen kann.

Weitere Verfeinerungen aus semantischer Sicht sind ebenfalls denkbar, z. B. eine Unterscheidung in fachliche und technische Klassen oder eine Zuordnung zu Schichten oder Aufgabenbereichen wie z. B. Benutzungsoberfläche, funktionaler Kern und Datenhaltung. Da diese Verfeinerungen aber aufgrund der in ODEM vorliegenden Information nicht automatisch gebildet werden können, sind hier keine Metriken dafür definiert. Entsprechende Metriken könnten aber einem spezifischen Qualitätsmodell hinzugefügt werden.

Metriken für Beziehungen. Zusätzlich sind die ausgehenden (efferenten) Beziehungen der Klasse zu zählen, weil sie negative Auswirkungen auf die Knappheit haben:

Ihre Verwaltung muss in der Klasse implementiert werden, so dass jede Beziehung die Knappheit verringert. Die entsprechende Metrik NEDC und ihre Verfeinerungen sind allerdings bei der Entkopplung (siehe Abschnitt A.3) definiert, da die Beziehungen dort eine wichtigere Rolle spielen.

Paket

Auf Paketebene zählt man die (direkt) in einem Paket enthaltenen Klassen, Interfaces und Pakete.

NCP (number of classes in a package)
$NCP(p) = |\{c \in C: \mathit{contains}(p,c)\}|$

NIP (number of interfaces in a package)
$NIP(p) = |\{i \in I: \mathit{contains}(p,i)\}|$

NPP (number of packages in a package)
$NPP(p) = |\{q \in P: \mathit{contains}(p,q)\}|$

Die drei Metriken lassen sich verfeinern, indem der Sichtbarkeitsbereich (public, protected, private) betrachtet wird. Die Sichtbarkeit eines Elements in einem Paket sollte immer so eingeschränkt wie möglich gewählt werden, daher gibt diese Verfeinerung Sinn. Sie wird hier am Beispiel von NCP demonstriert.

NCP_1 (number of public classes in a package)
$NCP_1(p) = |\{c \in C: \mathit{contains}(p,c) \wedge c.\mathit{visibility} = \text{public}\}|$

NCP_2 (number of protected classes in a package)
$NCP_2(p) = |\{c \in C: \mathit{contains}(p,c) \wedge c.\mathit{visibility} = \text{protected}\}|$

NCP_3 (number of private classes in a package)
$NCP_3(p) = |\{c \in C: \mathit{contains}(p,c) \wedge c.\mathit{visibility} = \text{private}\}|$

Die Metrik NCP kann auch verfeinert werden, indem abstrakte und konkrete Klassen unterschieden werden. Abstrakte Klassen dienen (wie Interfaces) vornehmlich der Modellierung von Schnittstellen, haben also einen geringeren Implementierungsaufwand als konkrete Klassen und sind in der Regel stabiler, d. h. sie haben eine geringere Änderungswahrscheinlichkeit.

NCP_a (number of abstract classes in a package)
$NCP_a(p) = |\{c \in C: \mathit{contains}(p,c) \wedge c.\mathit{isAbstract}\}|$

NCP_c (number of concrete classes in a package)
$NCP_c(p) = |\{c \in C: \mathit{contains}(p,c) \wedge \neg c.\mathit{isAbstract}\}|$

Die beiden Verfeinerungen von NCP lassen sich auch kombinieren.

System

Auf Systemebene zählt man alle Attribute, Operationen, Klassen, Interfaces und Pakete.

NAS (number of attributes in the system)
$NAS(S) = |A|$

NOS (number of operations in the system)
$NOS(S) = |O|$

NCS (number of classes in the system)
NCS(S) = |C|

NIS (number of interfaces in the system)
NIS(S) = |I|

NPS (number of packages in the system)
NPS(S) = |P| - 1 (da S in P enthalten ist, ist 1 abzuziehen)

Jede Eigenschaft, ob vererbt oder nicht, wird bei den Metriken NAS und NOS grundsätzlich nur einmal gezählt. Deshalb kann man hier nur nach Sichtbarkeitsbereich und nach Zugehörigkeit zu Klasse oder Objekt verfeinern (analog zu NAC und NOC). NCS, NIS und NPS können nach Sichtbarkeitsbereich verfeinert werden (analog zu NCP, NIP und NPP). Bei der Metrik NCS kann man auch nach abstrakten und konkreten Klassen verfeinern (analog zu NCP); diese Verfeinerung ist mit der nach Sichtbarkeitsbereich kombinierbar.

A.2 Strukturiertheit

Die Messung der Strukturiertheit konzentriert sich hier auf die Form der hierarchischen Strukturen, d. h. der Vererbungshierarchie der Klassen/Interfaces und der Schachtelungshierarchie der Pakete. Die reine Anzahl an Bestandteilen der Hierarchien wird bei der Knappheit betrachtet, die Verknüpfung der Bestandteile untereinander bei der Entkopplung.

Als weitere Hierarchie könnte noch die Aggregationsstruktur betrachtet werden. Genero et al. (2000) schlagen entsprechende Metriken vor. Da es aber noch keine praktischen Erfahrungen mit diesen Metriken gibt, werden sie hier nicht berücksichtigt.

Klasse/Interface: Vererbung

Zur Strukturiertheit tragen übersichtliche Baumstrukturen bei der Vererbung bei. Für Klassen kann die Tiefe in der Vererbungshierarchie (nach Chidamber und Kemerer, 1994, die Länge des längsten Pfads zur Wurzel der Hierarchie) bestimmt werden. Je größer die Messwerte werden, desto schlechter wird die Strukturiertheit.[1]

Die Definition von DITC (wie auch nachher von DNHP) ist rekursiv, weil sie sich aufgrund der rekursiven Struktur der Hierarchie so am leichtesten ausdrücken lässt.

DITC (depth of inheritance tree of a class)
DITC(c) = 0 für Wurzelklassen, d. h. Klassen ohne Oberklasse ($NEEC_I(c) = 0$), sonst
$DITC(c) = 1+\max_{d \in C \cup I:\ extends(c,d)} \{DITC(d)\}$

1. Der Zusammenhang zwischen DITC und der Strukturiertheit ist eigentlich intuitiv klar. Empirische Untersuchungen kamen interessanterweise aber zu unterschiedlichen Resultaten. So ergab sich in einer Untersuchung, dass ein System mit Vererbung leichter zu warten ist als ein funktional gleiches ohne Vererbung (Cartwright, 1998), in einer anderen war es umgekehrt (Daly et al., 1996). Dabei können aber auch grundsätzliche Schwierigkeiten der Experimentteilnehmer mit Vererbung die Ursache gewesen sein. El Eman und Melo (2001) geben eine Übersicht über die widersprüchlichen Ergebnisse der Experimente in diesem Bereich. In ihrer Untersuchung kommen sie zu dem Schluss, dass DITC ein guter Indikator für die Fehleranfälligkeit einer Klasse ist (aufgrund geringer Verständlichkeit).

Neben der Tiefe der Vererbungshierarchie kann noch ihre Verzweigung betrachtet werden. Aus Sicht einer Klasse ist das zum einen die Anzahl der Unterklassen, gemessen durch die Metrik $NAEC_l$ (number of local afferent extends relationships of a class), definiert beim Kriterium Entkopplung. Ein anderes Verzweigungsmaß ist die Anzahl der Oberklassen.[2] Die zugehörige Metrik $NEEC_l$ (number of local efferent extends relationships of a class) ist ebenfalls beim Kriterium Entkopplung definiert. Hohe Verzweigung ist schlecht für die Strukturiertheit.

Paket: Schachtelung

Auf Paketebene gibt es die Tiefe und den Verzweigungsgrad der Schachtelungshierarchie. Die Metrik für die Tiefe in der Schachtelungshierarchie ist:

DNHP (depth in nesting hierarchy of a package)
DNHP(p) = DNHP(q: *contains*(q,p)) + 1
DNHP(S) = -1 (damit Pakete auf der obersten Ebene ein DNHP von 0 haben)

Der Verzweigungsgrad (d. h. die Anzahl der eingeschachtelten Pakete) wurde bereits bei der Knappheit als NPP (number of packages in a package) definiert.

System

Auf Systemebene werden die Vererbungs- und die Schachtelungshierarchie als Ganzes betrachtet. Dazu verwendet man die maximale Tiefe und den maximalen Verzweigungsgrad:

DITS (depth of inheritance tree of the system)
$DITS(S) = \max_{c \in C \cup I} \{DITC(c)\}$

MNCS (maximum number of child classes in the system)
$MNCS(S) = \max_{c \in C \cup I} \{NEEC_l(c)\}$

DNHS (depth of nesting hierarchy of the system)
$DNHS(S) = \max_{p \in P} \{DNHP(p)\}$

MNPS (maximum number of subpackages in the system)
$MNPS(S) = \max_{p \in P} \{NPP(c)\}$

A.3 Entkopplung

Wie bereits im Abschnitt 8.3.3 ausgeführt stellt die Entkopplung einen der wichtigsten Indikatoren für die Wartbarkeit dar. Daher ist es nicht erstaunlich, dass schon viele Kopplungsmetriken vorgeschlagen wurden. Briand et al. (1999) geben einen Überblick über die Literatur. Sie stellen auch wesentliche Fragestellungen zusammen, die bei der Wahl von Kopplungsmetriken zu beantworten sind:

1. Welche Kopplungsarten werden betrachtet?
2. Wie wird die Stärke einer Kopplung bewertet (Kopplungsart und -häufigkeit)?
3. Wird die Richtung der Kopplung betrachtet (Import-/Export-Kopplung)?

2. Für eine echte Hierarchie sollte diese Zahl eigentlich immer 1 (oder 0 für die Wurzel) sein. Allerdings könnte es Mehrfachvererbung geben, was zu einer höheren Komplexität führt.

4. Wird nur direkte oder auch indirekte Kopplung betrachtet?

Diese Fragen werden hier wie folgt beantwortet:

1. Kopplung entsteht durch Beziehungen zwischen den Einheiten des Entwurfs. Je mehr Beziehungen es gibt, desto geringer ist die Entkopplung. Die geerbten Beziehungen werden mitgezählt.
2. Mehrfache Beziehungen der gleichen Art zwischen zwei Einheiten werden auch mehrfach gezählt. Dadurch haben die Zählmetriken bei derartiger Redundanz einen höheren Wert. Darüber hinaus werden keine Unterschiede in der Kopplungsstärke gemacht.
3. Die Richtung der Kopplung spielt eine Rolle. Die Richtung ist gegeben durch die Richtung der Abhängigkeit (z. B. wenn Klasse B von Klasse A erbt, ist B an A gekoppelt, aber nicht umgekehrt).
4. Es werden nur direkte Beziehungen betrachtet, denn die zusätzliche Berücksichtigung indirekter Beziehungen erbringt keinen Zusatznutzen (Briand et al., 1999).

Klasse/Interface

Auf Klassenebene ergibt sich damit zunächst die folgende Zählmetrik, die alle von der Klasse ausgehenden (efferenten) Beziehungen berücksichtigt:

NEDC (number of efferent dependencies of a class)
$NEDC(c) = \Sigma_{d \in C \cup I}\, depends\_on^*(c,d).weight$

Für diese wie für alle folgenden Metriken zur Entkopplung gilt, dass sie negativ mit der Entkopplung korreliert ist.

Eine Unterscheidung in lokal definierte und geerbte Beziehungen (wie bei Attributen und Operationen) ist möglich:

$NEDC_l$ (number of local efferent dependencies of a class)
$NEDC_l(c) = \Sigma_{d \in C \cup I}\, depends\_on(c,d).weight$

Beziehungen der Klasse mit sich selbst stellen einen Sonderfall dar, weil die Abhängigkeit lokal beschränkt ist. Dadurch ist diese Kopplung weniger schlecht für die Wartbarkeit als eine Kopplung nach außen. Deshalb ist es sinnvoll, mit reflexiven Beziehungen zu verfeinern:

$NEDC_r$ (number of reflexive efferent dependencies of a class)
$NEDC_r(c) = depends\_on^*(c,c).weight$

Eine weitere Verfeinerung unterscheidet zwischen Abhängigkeiten von abstrakten Klassen/Interfaces und konkreten Klassen. Abstrakte Klassen und Interfaces sind in der Regel stabiler, d. h. ihre Schnittstelle ändert sich nicht so häufig. Deshalb könnten solche Abhängigkeiten geringer gewichtet werden als Abhängigkeiten von konkreten Klassen.

$NEDC_a$ (number of abstract efferent dependencies of a class)
$NEDC_a(c) = \Sigma_{d \in C \cup I:\ d.isAbstract}\, depends\_on^*(c,d).weight$

Außerdem können Beziehungen zu Modellelementen im gleichen Paket von Beziehungen zu Modellelementen in anderen Paketen unterschieden werden.

$NEDC_p$(number of package-internal efferent dependencies of a class)
$NEDC_p(c) = \Sigma_{p \in P:\ contains(p,c)} \Sigma_{d \in C \cup I:\ contains(p,d)}\ depends\_on^*(c,d).weight$

Schließlich kann noch nach der Art der Beziehung verfeinert werden: Vererbung, Realisierung, Assoziation und Benutzung. Wegen der großen Bedeutung dieser Verfeinerung werden statt Indizes neue Akronyme verwendet.

NEEC (number of efferent extends relationships of a class)
$NEEC(c) = \Sigma_{d \in C \cup I}\ extends^*(c,d).weight$

NERC (number of efferent realization relationships of a class)
$NERC(c) = \Sigma_{d \in C \cup I}\ realizes^*(c,d).weight$

NEAC (number of efferent association relationships of a class)
$NEAC(c) = \Sigma_{d \in C \cup I}\ associates^*(c,d).weight$

NEUC (number of efferent uses relationships of a class)
$NEUC(c) = \Sigma_{d \in C \cup I}\ uses^*(c,d).weight$

NERC und NEAC sind für Interfaces immer 0. NEAC lässt sich noch anhand der Assoziationsart (1 = normal, 2 = Aggregation und 3 = Komposition) verfeinern:

$NEAC_1$ (number of efferent normal association relationships of a class)
$NEAC_1(c) = \Sigma_{d \in C \cup I \setminus \{c\}:\ associates^*(c,d).aggregation = none}\ associates^*(c,d).weight$

$NEAC_2$ (number of efferent aggregation relationships of a class)
$NEAC_2(c) = \Sigma_{d \in C \cup I \setminus \{c\}:\ associates^*(c,d).aggregation = aggregate}\ associates^*(c,d).weight$

$NEAC_3$ (number of efferent composition relationships of a class)
$NEAC_3(c) = \Sigma_{d \in C \cup I \setminus \{c\}:\ associates^*(c,d).aggregation = composite}\ associates^*(c,d).weight$

Alle genannten Verfeinerungen lassen sich miteinander kombinieren, was in diesem Fall eine sehr hohe Zahl von Kombinationsmöglichkeiten mit sich bringt.

Bisher wurden nur von der Klasse ausgehende (efferente) Beziehungen betrachtet. Es können aber auch die zur Klasse hingehenden (afferenten) Beziehungen gezählt werden:

NADC (number of afferent dependencies of a class)
$NADC(c) = \Sigma_{d \in C \cup I}\ depends\_on^*(d,c).weight$

Diese Metrik gibt Hinweise auf Klassen, deren Änderung weit reichende Konsequenzen auf den Rest des Entwurfs hat: je höher, desto kritischer. Auf die Wartbarkeit der Klasse selbst hat das zwar keinen großen Einfluss, die Metrik sollte aber zumindest bei der Gesamtbetrachtung des Systems berücksichtigt werden. Alle Verfeinerungen der efferenten Abhängigkeiten lassen sich auch auf die afferenten Abhängigkeiten anwenden.

Paket

Bei Paketen gibt es bis auf die *contains*-Relation keine direkten Abhängigkeitsrelationen. Für die Kopplung ist die *contains*-Relation allerdings nicht relevant. Die Abhängigkeiten der Pakete sind daher aus den Abhängigkeiten der enthaltenen Klassen abzuleiten: Wenn eine enthaltene Klasse c eines Pakets p von einer Klasse d in einem anderen Paket q abhängt, so hängt das Paket p von q ab (vgl. Definition von *depends_on* für Pakete in Abschnitt 5.4.1). Die Kopplungsstärke ergibt sich dabei aus

der Anzahl der Beziehungen. Alle in diesem Abschnitt definierten Metriken sind zum Kriterium Entkopplung negativ korreliert.

NEDP (number of efferent dependencies of a package)
$NEDP(p) = \Sigma_{q \in P \setminus \{p\}}\ depends\_on^*(p,q).weight$

Zusätzlich kann noch gemessen werden, wie stark andere Pakete an ein Paket gekoppelt sind (also die Gegenrichtung zur bisher betrachteten).

NADP (number of afferent dependencies of a package)
$NADP(p) = \Sigma_{q \in P \setminus \{p\}}\ depends\_on^*(q,p).weight$

Diese Metriken berücksichtigen alle Abhängigkeiten, auch geerbte. Dadurch wird die tatsächliche Vernetzung deutlicher, als wenn nur die direkten Abhängigkeiten betrachtet werden. Eine mögliche Verfeinerung betrachtet nur die direkten Abhängigkeiten, hier gezeigt am Beispiel von NEDP:

$NEDP_l$ (number of local efferent dependencies of a package)
$NEDP_l(p) = \Sigma_{q \in P \setminus \{p\}}\ depends\_on(p,q).weight$

Ist man nur an der Anzahl der Pakete interessiert, an die ein Paket gekoppelt ist, so kann man *alternativ* zu NEDP und NADP auch die folgenden Metriken verwenden:

NECP (number of efferently coupled packages of a package)
$NECP(p) = |\{q \in P \setminus \{p\}: depends\_on^*(p,q))\}|$

NACP (number of afferently coupled packages of a package)
$NACP(p) = |\{q \in P \setminus \{p\}: depends\_on^*(q,p))\}|$

Auch für diese Metriken lässt sich die oben gezeigte Verfeinerung mit der Beschränkung auf direkte Abhängigkeiten anwenden.

System

Auf Systemebene lassen sich keine Metriken angeben, die eine wirklich neue Sicht zur Kopplung ermöglichen. Denkbar sind zwar Durchschnitte o. Ä. der Kopplungsmetriken auf Klassen- und Paketebene, doch kann diese Form der Aggregation genauso gut auf der Ebene der subjektiven Metriken vorgenommen werden (vgl. Abschnitt 9.3.4). Daher wird darauf verzichtet, objektive Metriken anzugeben.

A.4 Zusammenhalt

Zusammenhalt ist in hohem Maße eine semantische Eigenschaft, die sich schlecht automatisch (d. h. auf syntaktischer Ebene) erfassen lässt.

Klasse/Interface

In der Literatur vorgeschlagene Zusammenhaltsmetriken für Klassen benötigen detaillierte Entwurfsinformation (vgl. z. B. Bieman, Kang, 1995, 1998; Chidamber, Kemerer, 1994; Briand et al., 1997). Im Wesentlichen stützen sich diese Metriken auf Gemeinsamkeiten zwischen den Methoden einer Klasse, z. B. bei LCOM (Chidamber, Kemerer, 1994) auf Zugriffe zweier Methoden auf die gleichen Attribute. Solch detaillierte Entwurfsinformation ist aber im UML-Modell und damit in ODEM nicht vorhanden.

Ein alternativer Vorschlag zur Messung des Zusammenhalts, der auf der Basis von ODEM möglich ist, stammt von Bansiya et al. (1999), die dazu Parametertypen der Operationen der Klasse heranzuziehen. Die Metrik CAMC (Cohesion Among Methods in Class) berechnet die durchschnittliche Überschneidung der Parametertypen einer Methode mit der Gesamtmenge der Parametertypen. Einen ähnlichen Vorschlag gibt es auch von Chen und Lu (1993). Beide Vorschläge beruhen allerdings auf der fragwürdigen Annahme, dass Operationen mit gemeinsamen Parametertypen auf höheren Zusammenhalt hindeuten. Die empirische Validierung der Vorschläge vermag jedenfalls nicht zu überzeugen. Mangels geeigneter objektiver Metriken muss daher für den Zusammenhalt auf eine subjektive Metrik zurückgegriffen werden.

Paket

Auf Paketebene wurde bisher nur ein Indikator für den Zusammenhalt der Klassen und Interfaces in einem Paket vorgeschlagen. Martin (1995) führte die Metrik „relational cohesion“ ein, welche die Anzahl der Beziehungen der Modellelemente untereinander ins Verhältnis zur Gesamtzahl der Modellelemente setzt (vgl. Abschnitt 5.5.2). Die Metrik misst also eine normierte paketinterne Kopplung. Eigene Erfahrungen beim Einsatz dieser Metrik in Fallstudien haben allerdings ergeben, dass die Metrik kein geeigneter Indikator für den Zusammenhalt der Modellelemente innerhalb eines Pakets ist. Trotz hoher Werte kann es sein, dass das Paket in mehrere nur schwach verbundene Teile zerfällt. Daher kann bei hohen Werten nicht auf hohen Zusammenhalt geschlossen werden, sondern nur bei sehr geringen Werten auf geringen Zusammenhalt. Aus diesem Grund wird die Metrik hier nicht verwendet. Man ist also auf subjektive Metriken angewiesen.

System

Vorschläge für systemweite Zusammenhaltsmetriken gibt es bisher nicht. Außerdem ist unklar, was Zusammenhalt auf Systemebene eigentlich bedeutet. Um keine Lücke im Verfahren entstehen zu lassen, wird hier der systemweite Zusammenhalt als Gesamteindruck auf der Basis des Zusammenhalts der enthaltenen Pakete, Klassen und Interfaces interpretiert. Auch hier ist eine subjektive Metrik am sinnvollsten.

A.5 Einheitlichkeit

Die Einheitlichkeit ist eine überwiegend semantische Eigenschaft, weshalb auch hier auf subjektive Metriken zurückgegriffen werden muss. Je nach den Vorgaben aus den Namenskonventionen lassen sich gewisse Minimalanforderungen automatisch prüfen, z. B. eine minimale Bezeichnerlänge oder Lesbarkeitskriterien, die Buchstaben-Ziffern-Gemische wie X22Z1I verbieten. Solche Prüfungen liefern aber nur ein syntaktisches Bild. Der wichtigere semantische Aspekt kann nicht erfasst werden. Außerdem hängen die zu prüfenden Kriterien von den konkreten Richtlinien ab. Will man sie erfassen, ist das spezifische Modell um weitere objektive Metriken und Fragen (zu den Fragebögen) zu erweitern.

A.6 Dokumentierung

Auch für die Dokumentierung ist eine subjektive Bewertung nötig. Zwar kann bei entsprechenden Dokumentierungskonventionen automatisch geprüft werden, ob Kommentare im geforderten Umfang vorhanden sind, doch ist es sehr schwer, automatisch zu prüfen, ob die Kommentare sinnvoll und hilfreich sind.

A.7 Verfolgbarkeit

Die Verfolgbarkeit lässt sich nicht direkt aus dem UML-Modell ermitteln, da es für die Verknüpfung von UML-Artefakten mit den Anforderungen kein standardisiertes Verfahren gibt.

Für Klassen und Pakete kann man nur beurteilen, wie gut dokumentiert ist, auf welche Anforderungen ihr Vorhandensein zurückzuführen ist. Eine (sinnvolle) objektive Metrik dafür lässt sich leider nicht angeben. Auf der Systemebene kann man eine objektive Metrik angeben, die sich allerdings nicht auf der Basis von ODEM ermitteln lässt: Die Metrik RTTR (ratio of traceable to total requirements) ist das Verhältnis der Anzahl der im Entwurf verfolgbaren Anforderungen zur Gesamtzahl der Anforderungen. Je größer der Wert, desto besser.

Neben der reinen Anzahl verfolgbarer Anforderungen könnte auch zusätzlich eine Gewichtung der Anforderungen vorgenommen werden, so dass die Verfolgbarkeit wichtiger oder instabiler Anforderungen stärker berücksichtigt wird als die Verfolgbarkeit unwichtiger oder stabiler Anforderungen. Eine solche Verfeinerung ist allerdings Aufgabe des spezifischen Qualitätsmodells.

Sofern ein bestimmtes Verfahren zur Verknüpfung von UML-Artefakten mit den Anforderungen existiert, kann diese Metrik auch automatisiert erhoben werden, nachdem ODEM um die entsprechenden Konstrukte erweitert wurde. In UML ist ein spezielles Stereotyp «trace» für Abstraktionsbeziehungen zwischen UML-Modellelementen definiert (im UML-Metamodell repräsentiert durch die Klasse Abstraction, vgl. Abbildung 5-2). Mit Hilfe einer solchen Beziehung lässt sich Tracing in UML realisieren. Es kann aber wohl nicht davon ausgegangen werden, dass sich alle Anforderungen in UML sinnvoll darstellen lassen (z. B. als Anwendungsfälle) und dass alle «trace»-Beziehungen vorhanden sind.

A.8 Wartbarkeit

Die wesentlichen Metriken zur Wartbarkeit stecken bereits in den Kriterien. Neu hinzugenommen werden können nur noch Metriken, die über das UML-Modell hinausgehen. Beispielsweise kann auf der Basis eines Szenario-basierten Bewertungsansatzes die (adaptive) Wartbarkeit gemessen werden, indem für jedes Änderungsszenario die Schwierigkeit der Änderung bewertet und eine Gesamtbewertung ermittelt wird (vgl. Abschnitt B.8). Solche Metriken müssen aber speziell für jedes spezifische Qualitätsmodell entwickelt werden.

A.9 Theoretische Validierung

Bevor die objektiven Metriken angewendet werden, sollten sie theoretisch validiert sein. Die Metriken zu Knappheit, Strukturiertheit und Entkopplung in QOOD lassen sich als Komplexitätsmetriken auffassen, weshalb sich das Axiomensystem von Weyuker (1988) für Komplexitätsmetriken auf sie anwenden lässt. Zusätzlich gibt es noch ein Axiomensystem vom Briand et al. (1999) für Kopplungsmetriken, die spezifisch für die Metriken der Entkopplung betrachtet werden können.

12.5.1 Komplexitätsmetriken

Weyuker (1988) hat eine Liste von neun Axiomen publiziert, die für Komplexitätsmetriken, die Programme auf syntaktischer Basis bewerten, gelten sollten. Sei m eine Komplexitätsmetrik und P, Q, R seien Programme. Dann muss gelten:

Axiom W1. Es muss Programme geben, die unterschiedliche Komplexität besitzen. Damit sollen Metriken ausgeschlossen werden, die allen Programmen die gleiche Komplexität zuweisen.

$\exists\, P, Q: m(P) \neq m(Q)$

Axiom W2. Die Anzahl der Programme, welche die gleiche Komplexität besitzen, muss endlich sein. Damit wird eine feinere Unterscheidung in Komplexitätsklassen gefordert als bei Axiom W1.

$\forall\, c \geq 0: |\{P \mid m(P) = c\}| < \infty$

Axiom W3. Es muss Programme geben, welche die gleiche Komplexität besitzen. Damit soll (als Gegenpol zu Axiom W2) eine zu feine Klassifikation (z. B. eine Gödelisierung) ausgeschlossen werden.

$\exists\, P, Q: P \neq Q \wedge m(P) = m(Q)$

Axiom W4. Es muss funktional äquivalente Programme geben, die unterschiedliche Komplexität besitzen. Damit soll sichergestellt werden, dass die Komplexität der Implementierung und nicht die der implementierten Funktion gemessen wird.

$\exists\, P, Q: P \equiv Q \wedge m(P) \neq m(Q)$ ($\equiv$ ist die funktionale Äquivalenz)

Axiom W5. Werden zwei Programme kombiniert, muss die Komplexität des Ganzen mindestens so groß sein wie die seiner Teile.

$\forall\, P, Q: m(P) \leq m(P;Q) \wedge m(Q) \leq m(P;Q)$ (; kombiniert zwei Programme)

Axiom W6. Wird ein Programm mit zwei verschiedenen Programmen gleicher Komplexität kombiniert, können die Komplexitäten der Resultate unterschiedlich sein. Das Axiom soll sicherstellen, dass es mindestens einen solchen Fall je Kombinationsreihenfolge gibt.

a) $\exists\, P, Q, R: m(P) = m(Q) \wedge m(P;R) \neq m(Q;R)$
b) $\exists\, P, Q, R: m(P) = m(Q) \wedge m(R;P) \neq m(R;Q)$

Axiom W7. Werden die Anweisungen eines Programms permutiert, kann sich die Komplexität ändern (sie muss aber nicht). Daher wird gefordert, dass es eine Permutation eines Programms geben muss, die eine andere Komplexität hat als das Programm selbst. Damit wird aber von allen Komplexitätsmetriken verlangt, dass sie

sich mit den Details der Programme beschäftigen. Die meisten üblichen (und sinnvollen!) Metriken (z. B. McCabes zyklomatische Komplexität) sind aber nicht sensitiv gegenüber einer Permutation von Anweisungen, weshalb dieses Axiom eher fragwürdig ist.

$\exists\, P, Q: Q = \pi(P) \wedge m(P) \neq m(Q)$ (π permutiert die Anweisungen von P)

Axiom W8. Werden nur Bezeichner umbenannt, darf sich die Komplexität nicht ändern. Da hier nur strukturelle Komplexität, nicht psychologische Komplexität betrachtet wird, gibt diese Forderung Sinn.

$\forall\, P, Q: Q = \rho(P) \Rightarrow m(P) = m(Q)$ (ρ benennt Bezeichner in P um)

Axiom W9. Es kann Fälle geben, bei denen die Komplexität einer Kombination zweier Programme echt größer ist als die Summe der Komplexitäten der beiden Programme (durch zusätzliche Interaktion der Teile). Hier wird verlangt, dass es mindestens einen solchen Fall geben muss.

$\exists\, P, Q: m(P) + m(Q) < m(P;Q)$

Diskussion

Diese Axiome bedeuten zum Teil sehr starke Einschränkungen (Shepperd, Ince, 1993). Die von Weyuker betrachteten Beispiele wie Lines of Code oder McCabes zyklomatische Komplexität erfüllen höchstens sieben der neun Axiome, scheinen aber trotzdem nützliche Komplexitätsmetriken zu sein. Daher scheint es fragwürdig, ob eine Komplexitätsmetrik wirklich alle neun Axiome erfüllen muss. Andererseits sind die Axiome von Weyuker trotz ihrer Restriktivität nur notwendige Bedingungen für Komplexitätsmetriken, keine hinreichenden. Beispielsweise geben Cherniavsky und Smith (1991) eine Metrik an, für die alle Axiome gelten, die aber keine sinnvolle Komplexitätsmetrik ist.

Chidamber und Kemerer (1994) haben für ihre objektorientierten Entwurfsmetriken untersucht, welche Axiome von Weyuker gelten. Dabei stellten sie fest, dass die Axiome W7 und W9 für keine der Metriken gelten. Daraus folgern sie, dass diese Axiome vermutlich für objektorientierte Metriken allgemein nicht anwendbar sind. Für Axiom W7 ist der Grund offensichtlich: Überträgt man die Axiome von Programmen auf Entwürfe in UML, gibt es keine sinnvolle Interpretation für eine Permutation von Entwurfselementen, da hier im Gegensatz zu Programmen keine Reihenfolge der Elemente vorhanden ist. Bei Axiom W9 ist es wohl so, dass bei der Kombination objektorientierter Entwürfe keine Effekte auftreten, die zu einer höheren Komplexität als der Summe der Teile führen. Gursaran und Roy (2002) kommen für W9 zu dem Ergebnis, dass das Axiom zumindest für Vererbungsmetriken grundsätzlich nicht anwendbar ist. Daher werden die Axiome W7 und W9 nicht weiter betrachtet.

Untersuchung der Metriken

Die Metriken für Knappheit, Strukturiertheit und Entkopplung sind Komplexitätsmetriken. Daher wird untersucht, welche der Axiome für die Metriken gelten. Das wird hier am Beispiel der Metrik NAC (Number of Attributes of a Class) demonstriert.

Axiom W1. Das Axiom gilt. Man wähle eine Klasse P mit einem Attribut und eine Klasse Q mit zwei Attributen. Dann gilt $\mathrm{NAC}(P) \neq \mathrm{NAC}(Q)$.

Axiom W2. Das Axiom gilt, sofern man von der vernünftigen Annahme ausgeht, dass Klassen endlich viele Attribute, Methoden und Beziehungen haben sowie Bezeichner eine endliche Länge haben. Dann kann es nur endliche viele Klassen mit einer bestimmten Anzahl von Attributen geben, so dass die Bedingung erfüllt ist.

Axiom W3. Das Axiom gilt. Man wähle eine Klasse P mit einem Attribut a und eine Klasse Q mit einem Attribut b. Dann gilt $P \neq Q \wedge \mathrm{NAC}(P) = \mathrm{NAC}(Q)$

Axiom W4. Das Axiom gilt. Man wähle eine Klasse P mit einem Attribut und eine funktional äquivalente Klasse Q mit zwei Attributen (z. B. P mit einem weiteren, unnötigen Attribut). Dann gilt $P \equiv Q \wedge \mathrm{NAC}(P) \neq \mathrm{NAC}(Q)$

Axiom W5. Das Axiom gilt. Werden zwei Klassen P und Q miteinander verschmolzen, gehen keine Attribute verloren. Daher ist die Zahl der Attribute in der kombinierten Klasse mindestens so hoch wie in den Ursprungsklassen. Damit gilt $\mathrm{NAC}(P) \leq \mathrm{NAC}(P;Q) \wedge \mathrm{NAC}(Q) \leq \mathrm{NAC}(P;Q)$.

Axiom W6. Das Axiom gilt. Man wähle P und Q so, dass sie gleich viele Attribute haben, aber Q mindestens ein Attribut besitzt, das P nicht hat. R wählt man gleich P. Bei der Verschmelzung von P und R kommt nichts dazu, bei der Verschmelzung von R und Q schon. Die Reihenfolge spielt bei der Verschmelzung keine Rolle, daher lässt sich das Beispiel auf a) und b) anwenden. Damit gilt $\mathrm{NAC}(P) = \mathrm{NAC}(Q) \wedge \mathrm{NAC}(P;R) \neq \mathrm{NAC}(Q;R) \wedge \mathrm{NAC}(R;P) \neq \mathrm{NAC}(R;Q)$.

Axiom W8. Das Axiom gilt, weil das Umbenennen von Bezeichnern keinen Einfluss auf die Anzahl der Attribute hat.[3]

Die am Beispiel gezeigte Beweisführung lässt sich auf die anderen Metriken übertragen. Ergebnis der Untersuchung ist, dass die Axiome W1, W2, W3, W4, W6 und W8 für alle Metriken gelten. W5 gilt für alle Metriken mit Ausnahme von DITC, bei der es einen Spezialfall gibt, der W5 nicht erfüllt (siehe DIT bei Chidamber, Kemerer, 1994, S. 483f.). Für die Verfeinerungen gelten dieselben Axiome wie die ursprüngliche Metrik. Alle Metriken (mit einer leichten Einschränkung bei DITC) besitzen also theoretische Validität als Komplexitätsmetriken.

12.5.2 Kopplungsmetriken

Für die theoretische Validierung objektorientierter Kopplungsmetriken können auch die Axiome von Briand et al. (1999) verwendet werden:

Axiom BDW1 (Nichtnegativität). Kopplung ist nie negativ.

$\forall\ P: m(P) \geq 0$

Axiom BDW2 (Nullwert). Klassen ohne Beziehungen nach außen können keine Kopplung haben, also soll die Kopplung 0 sein.

$\forall\ P: m(P) = 0 \Leftrightarrow P$ hat keine Beziehungen nach außen

Axiom BDW3 (Monotonie). Fügt man einer Klasse weitere Beziehungen hinzu, wird die Kopplung nicht kleiner, sondern bleibt gleich oder steigt.

3. Der Sonderfall, durch Umbenennung zwei Attribute desselben Namens (und desselben Typs) zu erhalten, ist möglich. Es gibt aber eine Wohlgeformtheitsbedingung in der UML, die eine solche Klasse für ungültig erklärt, so dass eine solche Klasse kein Messgegenstand sein kann.

$\forall\ P, Q$: $Q = \xi(P) \Rightarrow m(P) \leq m(Q)$ (ξ erweitert P um weitere Beziehungen)

Axiom BDW4 (Verschmelzen von Klassen). Werden zwei Klassen miteinander verschmolzen, wird die Kopplung des Resultats höchstens so groß sein wie die Summe der Kopplungen der beiden Klassen. Das liegt daran, dass Beziehungen der beiden Klassen untereinander durch die Verschmelzung entfallen. Interessanterweise ist die Aussage dieses Axioms die entgegengesetzte von W9!

$\forall\ P, Q$: $m(P) + m(Q) \geq m(P;Q)$

Axiom BDW5 (Verschmelzen unverbundener Klassen). Ein Spezialfall bei der Verschmelzung von Klassen sind Klassen, die keine Beziehungen untereinander haben. Dann ist die Kopplung des Resultats gleich der Summe der Kopplungen der beiden Klassen.

$\forall\ P, Q$: P und Q haben keine Beziehungen miteinander $\Rightarrow m(P) + m(Q) = m(P;Q)$

Untersuchung der Metriken

Die Metriken für Entkopplung sind Kopplungsmetriken, daher können sie zusätzlich im Hinblick auf diese Axiome untersucht werden. Das wird hier am Beispiel der Metrik NEDC (Number of Efferent Dependencies of a Class) demonstriert.

Axiom BDW1. Das Axiom gilt, denn der kleinstmögliche Wert einer Zählmetrik ist 0.

Axiom BDW2. Das Axiom gilt. Wenn eine Klasse keine Beziehungen nach außen hat, liefert NEDC den Wert 0. Hat sie hingegen Beziehungen nach außen, liefert NEDC die Anzahl dieser Beziehungen, also einen Wert größer 0.

Axiom BDW3. Das Axiom gilt. Fügt man einer Klasse eine Beziehung nach außen hinzu, erhöht sich NEDC um 1. Damit ist die Ungleichung erfüllt.

Axiom BDW4. Das Axiom gilt. Werden zwei Klassen miteinander verschmolzen, entfallen alle Beziehungen zwischen den beiden Klassen, werden also von NEDC nicht mehr mitgezählt. Neue Beziehungen können dagegen durch das Verschmelzen nicht hinzukommen. Damit ist die Ungleichung erfüllt.

Axiom BDW5. Das Axiom gilt. Werden zwei Klassen miteinander verschmolzen, die keine Beziehungen untereinander haben, gehen durch das Verschmelzen keine Beziehungen verloren[4], es kommen aber auch keine dazu. Damit ist die Gleichung erfüllt.

Die am Beispiel gezeigte Beweisführung lässt sich auf die anderen Entkopplungsmetriken übertragen. Ergebnis der Untersuchung ist, dass alle fünf Axiome für alle Metriken gelten. Daher besitzen die Entkopplungsmetriken theoretische Validität im Sinne von Briand et al. (1999).

4. Ein Sonderfall ist das Verschmelzen zweier Klassen, welche Beziehungen zur selben Klasse haben. Entstehen in der Resultatklasse zwei Beziehungen desselben Typs zur selben Klasse, handelt es sich dennoch um zwei verschiedene Beziehungen, die nicht zusammenfallen.

Anhang B

Fragebögen für QOOD

Dieser Anhang stellt die Fragebögen für QOOD im Detail vor. Die Fragebögen werden geordnet nach den Entwurfskriterien des Faktors Wartbarkeit (siehe Abbildung 8-1) präsentiert. Innerhalb eines Kriteriums sind sie nach Ebenen sortiert, wobei mit der untersten Ebene (Klassen/Interfaces) begonnen wird. Wenn für eine Ebene oder ein Kriterium kein Fragebogen verfügbar ist, wird eine Begründung dafür angegeben.

Quellen

Für die Fragebögen für den Faktor Wartbarkeit wurde unter anderem Material aus den folgenden Quellen verwendet:

- Balzert (1999) gibt mehrere Checklisten für objektorientierte Analysemodelle an, die sich auf den Entwurf übertragen lassen
- Booch et al. (1998) geben in ihrem UML-Handbuch Listen mit erwünschten Eigenschaften für die Bestandteile des UML-Modells an.
- Page-Jones (1995) enthält eine Checkliste für Klassen.
- McBreen (2000) gibt eine Checkliste für den objektorientierten Entwurf an.
- Riel (1996) gibt viele Heuristiken an, die gute und schlechte Entwurfseigenschaften beschreiben. Daraus lassen sich leicht Fragen für einen Fragebogen generieren.
- Die Qualitätsmodelle aus Abschnitt 7.4 liefern weitere Fragen.

Zwischen diesen Quellen gibt es große inhaltliche Überlappungen, so dass zunächst ein Abgleich stattfand. Da hier davon ausgegangen wird, dass die UML-Modelle wohlgeformt sind, wurde auf Fragen verzichtet, welche die Wohlgeformtheit überprüfen. Eine solche Prüfung kann ein UML-Werkzeug automatisch durchführen.

Kann eine Frage nicht vollkommen positiv beantwortet werden, sollte zusätzlich notiert werden, was der Grund für die Abwertung ist. Die so entstehende Mängelliste ist für eine spätere Überarbeitung des Entwurfs sehr nützlich.

Darstellung

Die Bedingungen der Fragen werden durch Prädikate formalisiert. Die Prädikate können einen impliziten Parameter *this* verwenden, der den aktuellen Bewertungsgegenstand bezeichnet. Die Gewichte weniger wichtig, wichtig und sehr wichtig werden durch Sternchen visualisiert (* für weniger wichtig, ** für wichtig und *** für sehr wichtig). Ist eine Frage automatisch beantwortbar, wird in der letzten Spalte ein Häkchen gesetzt.

Bei manche Fragen geht es um das Fehlen bestimmter Eigenschaften, z. B. dass keine zyklischen Abhängigkeiten vorhanden sind. Bei der Antwort soll – entgegen dem Sprachgebrauch – mit ja geantwortet werden, wenn die Aussage zutrifft und mit nein, wenn die Aussage nicht zutrifft.

B.1 Knappheit

Da Knappheit geringe Größe bedeutet, enthalten die Fragebögen verschiedene Fragen nach unnötigen und redundanten Entwurfsteilen.

Klasse/Interface

Bedingung	Fragetext	Antwortskala	Gewicht	auto.
–	Ist das Vorhandensein der Klasse notwendig?	0 = nein, 1 = ja	***	
this∈C	Enthält die Klasse nur die nötigen Attribute? (z. B. keine nicht (mehr) verwendeten oder für die Verantwortlichkeiten der Klasse nicht relevanten)	0 = nein, 1 = ja	**	
NOC(this) > 0	Enthält die Klasse nur die nötigen Operationen? (z. B. keine nicht (mehr) verwendeten oder für die Verantwortlichkeiten der Klasse nicht relevanten)	0 = nein, 1 = ja	**	
NOC(this) > 0	Enthält die Klasse keine überflüssigen Operationen? (z. B. überladene Operationen oder andere „Komfort-Operationen“)	0 = nein, 1 = ja	*	
NOC(this) > 0	Gibt es keine ähnlichen Operationen in anderen Klassen? Wird die Implementierung vermutlich keinen redundanten Code enthalten?	0 = nein, 1 = ja	**	
NOC(this) > 0	Benötigt jede Operationen alle ihre Parameter?	0 = nein, 1 = ja	**	
NEEC(this) > 0	Fügt die Unterklasse neue Attribute oder Operationen hinzu?	0 = nein, 1 = ja	**	✔
this∈C ∧ this.isAbstract	Hat die abstrakte Klasse mindestens eine Unterklasse?	0 = nein, 1 = ja	***	✔
this∈I	Wird das Interface realisiert oder von anderen Interfaces geerbt?	0 = nein, 1 = ja	***	✔

Fragebogen B-1: Knappheit Klasse/Interface

Paket

Bedingung	Fragetext	Antwortskala	Gewicht	auto.
–	Enthält das Paket mindestens eine Komponente?	0 = nein, 1 = ja	***	✔
–	Ist das Vorhandensein des Paket notwendig?	0 = nein, 1 = ja	***	

Fragebogen B-2: Knappheit Paket

System

Bedingung	Fragetext	Antwortskala	Gewicht	auto.
–	Gibt es keine zwei oder mehr Klassen, welche die gleiche Aufgabe haben, also bis auf eine überflüssig sind?	0 = nein, 1 = ja	***	
–	Gibt es keine zwei oder mehr Pakete, welche die gleiche Aufgabe haben, also bis auf eines überflüssig sind?	0 = nein, 1 = ja	***	
–	Gibt keine zwei oder mehr Unterklassen einer Klasse, die ähnliche oder gleiche Eigenschaften haben, die in die Oberklasse verschoben werden sollten?	0 = nein, 1 = ja	***	

Fragebogen B-3: Knappheit System

B.2 Strukturiertheit

Die Fragen beschäftigen sich mit der Pakethierarchie und der Vererbungshierarchie.

Klasse/Interface

Bedingung	Fragetext	Antwortskala	Gewicht	auto.
–	Ist die Klasse in der Vererbungshierarchie höchstens 6 Stufen tief?	0 = nein, 1 = ja	**	✔
–	Hat die Klasse höchstens 9 Unterklassen?	0 = nein, 1 = ja	**	✔
NAEC(this) = 1	Hat die Klasse nur eine Unterklasse, aber es gibt keinen Sinn, die beiden zu verschmelzen?	0 = nein, 1 = ja	*	
–	Hat die Klasse keine Unterklassen, die eigentlich Instanzen der Klasse sein sollten?	0 = nein, 1 = ja	**	

Fragebogen B-4: Strukturiertheit Klasse/Interface

Paket

Bedingung	Fragetext	Antwortskala	Gewicht	auto.
DNHP(this) > 0	Ist das Paket höchstens 6 Stufen tief eingeschachtelt?	0 = nein, 1 = ja	**	✔
–	Ist das Paket weder leer noch enthält es nur sehr wenige (ein oder zwei) Elemente?	0 = nein, 1 = ja	***	✔
–	Enthält das Paket höchstens 30 Elemente?	0 = nein, 1 = ja	**	✔
–	Enthält das Paket höchstens 9 Pakete?	0 = nein, 1 = ja	**	✔

Fragebogen B-5: Strukturiertheit Paket

System

Bedingung	Fragetext	Antwortskala	Gewicht	auto.
–	Ist die Schachtelungshierarchie der Pakete höchstens 6 Stufen tief?	0 = nein, 1 = ja	**	✔
–	Sind die Abstraktionen hoch in der Vererbungshierarchie stabil (d. h. sie werden sich wahrscheinlich nicht ändern)?	0 = nein, 1 = ja	***	
–	Sind alle Vererbungshierarchien höchstens 6 Stufen tief?	0 = nein, 1 = ja	**	✔
–	Umfasst jede Vererbungshierarchie höchstens 50 Klassen?[a]	0 = nein, 1 = ja	**	✔

Fragebogen B-6: Strukturiertheit System

a. Quelle des Schwellenwerts: Mayrand et al. (1996)

B.3 Entkopplung

Um die Entkopplung zu verbessern, wird allgemein eine hohe Kapselung empfohlen, da dadurch die Abhängigkeiten zwischen den Entwurfseinheiten reduziert werden können. Beispielsweise sollten Attribute grundsätzlich nicht öffentlich sichtbar sein. Die öffentliche Schnittstelle einer Klasse, die demzufolge nur aus Operationen bestehen sollte, sollte so klein wie möglich sein. Kopplung zu abstrakten Klassen und Interfaces ist der Kopplung zu konkreten Klassen vorzuziehen, da sich Abstraktes weniger häufig ändert als Konkretes.

Klasse/Interface

Bedingung	Fragetext	Antwortskala	Gewicht	auto.
–	Sind alle Entwurfsentscheidungen so weit wie möglich verborgen?	0 = nein, 1 = ja	***	
¬isAbstract(this)	Sind von der konkreten Klasse mehr als 9 andere Klassen abhängig?	0 = nein, 1 = ja	*	✔
NEEC(this) > 1	Ist das Erben von mehreren Klassen hier nötig?	0 = nein, 1 = ja	**	
–	Sind alle Assoziationen mit anderen Klassen nötig?	0 = nein, 1 = ja	**	
–	Sind mehrfache Assoziationen mit derselben Klasse unnötig? (nötig, wenn sie unterschiedliche Bedeutung besitzen oder unterschiedliche Multiplizitäten)	0 = nein, 1 = ja	*	
–	Gibt es nur nötige Benutzungsbeziehungen? (Kriterium z. B. Law of Demeter, Lieberherr et al., 1988, 1989)	0 = nein, 1 = ja	**	
–	Realisiert die Klasse kein Interface, das bereits von einer Oberklasse realisiert wird?	0 = nein, 1 = ja	**	✔
–	Gibt es keine Abhängigkeit zu konkreten Klassen?	0 = nein, 1 = ja	*	✔
–	Gibt es keine Abhängigkeit zu einer direkten oder indirekten Unterklasse?	0 = nein, 1 = ja	**	✔
–	Gibt es keine Assoziation mit einer anderen Klasse, welche die betrachtete Klasse enthält (durch Aggregation oder Komposition)?	0 = nein, 1 = ja	**	✔
–	Gibt es keine Assoziation der Klasse zu einer anderen Klasse A, wobei beide Klassen von einer Klasse B aggregiert/komponiert werden?	0 = nein, 1 = ja	*	✔
–	Gibt es keine zyklischen Abhängigkeiten mit anderen Klassen?	0 = nein, 1 = ja	**	✔
–	Ist bei einer bidirektionalen Assoziation die Navigierbarkeit in beide Richtungen nötig?	0 = nein, 1 = ja	*	
–	Liegt bei einer bidirektionalen 1:1-Assoziation eine echte Assoziation vor? (Kriterien: Verbindung ist in eine oder beide Richtungen optional oder kann sich ändern, es handelt sich um zwei umfangreiche Klassen oder sie besitzen eine unterschiedliche Semantik)	0 = nein, 1 = ja	***	
–	Handelt es sich bei einer Aggregation um eine echte Aggregation? (Kriterien: Teil-Ganzes-Beziehung, aber der Teil kann mehrere Besitzer gleichzeitig haben oder den Besitzer wechseln)	0 = nein, 1 = ja	***	

Fragebogen B-7: Entkopplung Klasse/Interface (Abschnitt 1 von 2)

Bedingung	Fragetext	Antwortskala	Gewicht	auto.
–	Handelt es sich bei einer Komposition um eine echte Komposition? (Kriterien: Teil-Ganzes-Beziehung, Multiplizität der Aggregatklasse ist 0..1 oder 1, Lebensdauer der Teile ist an das Ganze gebunden, Funktionen des Ganzen werden automatisch auf die Teile angewendet)	0 = nein, 1 = ja	***	
–	Ist jede Vererbungsbeziehung eine Spezialisierungsbeziehung?	0 = nein, 1 = ja	**	
–	Gibt es keine Benutzungsbeziehung, die eine strukturelle Beziehung (z. B. Assoziation) modelliert?	0 = nein, 1 = ja	**	
–	Gibt es keine lokalen Attribute, Operationen oder Beziehungen, die weiter oben in der Vererbungshierarchie definiert sein sollten?	0 = nein, 1 = ja	**	
NEEC(this) > 0	Benötigt jede Unterklasse alle geerbten Attribute, Operationen und Beziehungen?	0 = nein, 1 = ja	**	
this∈ C	Ist der Sichtbarkeitsbereich jedes Attributs so gering wie möglich?	0 = nein, 1 = ja	***	
this∈ C	Gibt es keine öffentlich sichtbaren (public) Attribute?	0 = nein, 1 = ja	**	
–	Ist der Sichtbarkeitsbereich jeder Operation so gering wie möglich?	0 = nein, 1 = ja	***	
–	Ist jede öffentlich sichtbare (public) Operation wirklich nötig?	0 = nein, 1 = ja	***	
–	Hat keine der Operationen mehr als 6 Parameter?	0 = nein, 1 = ja	**	✔

Fragebogen B-7: Entkopplung Klasse/Interface (Abschnitt 2 von 2)

Paket

Bedingung	Fragetext	Antwortskala	Gewicht	auto.
–	Sind alle Entwurfsentscheidungen so weit wie möglich verborgen?	0 = nein, 1 = ja	***	
–	Gibt es keine unnötige Abhängigkeiten zu anderen Paketen? (durch Abhängigkeiten zu Klassen/Interfaces aus diesen Paketen)	0 = nein, 1 = ja	***	
–	Gibt es keine zyklische Abhängigkeiten mit anderen Paketen?	0 = nein, 1 = ja	**	✔

Fragebogen B-8: Entkopplung Paket

System

Bedingung	Fragetext	Antwortskala	Gewicht	auto.
–	Sind alle Entwurfsentscheidungen so weit wie möglich verborgen?	0 = nein, 1 = ja	***	
–	Gibt es weder globale Variablen noch werden öffentliche Klassenattribute als solche verwendet?	0 = nein, 1 = ja	***	

Fragebogen B-9: Entkopplung System

B.4 Zusammenhalt

Wesentliche Prinzipien für die Erhöhung des Zusammenhalts sind die Trennung der Zuständigkeiten (separation of concerns) und die Trennung von Verhalten und Implementierung (separation of policy and implementation). Klassen mit hohem Zusammenhalt realisieren eine abgegrenzte Aufgabe vollständig.

Klasse/Interface

Bedingung	Fragetext	Antwortskala	Gewicht	auto.
–	Ist die Klasse eine abgegrenzte Abstraktion eines Begriffs aus dem Problem- oder Lösungsbereich?	0 = nein, 1 = ja	***	
–	Realisiert die Klasse nur eine Verantwortlichkeit (oder sehr wenige)? (wenn viele Operationen vorhanden sind, deutet das auf das Gegenteil hin)	0 = nein, 1 = ja	***	
–	Realisiert die Klasse ihre Verantwortlichkeiten vollständig?	0 = nein, 1 = ja	***	
–	Hat die Klasse mindestens eine Operation (einschließlich der geerbten)?	0 = nein, 1 = ja	*	✔
–	Hat die Klasse nicht nur get-/set-Operationen?	0 = nein, 1 = ja	***	
–	Verfügt die Klasse höchstens über wenige get-/set-Operationen? (viele get/set-Operationen deuten auf eine schlechte Aufteilung hin, da sich offensichtlich Funktionalität außerhalb der Klasse befindet)	0 = nein, 1 = ja	**	
this∈I	Stellt das Interface alle nötigen Operationen für einen einzigen Dienst zur Verfügung?	0 = nein, 1 = ja	***	
this∈I	Enthält das Interface ausschließlich die für seinen Dienst notwendige Operationen?	0 = nein, 1 = ja	***	
–	Realisiert jede Operation eine einzige Funktion?	0 = nein, 1 = ja	**	

Fragebogen B-10: Zusammenhalt Klasse/Interface (Abschnitt 1 von 2)

Bedingung	Fragetext	Antwortskala	Gewicht	auto.
–	Realisiert jede Operation ihre Funktion (auf einer gewissen Abstraktionsebene) vollständig?	0 = nein, 1 = ja	**	
–	Wird jedes Attribut von mindestens einer Operation (mit Ausnahme der get-/set-Operationen) der Klasse benötigt?	0 = nein, 1 = ja	**	
–	Benötigt jede Operation mindestens ein Attribut der Klasse?	0 = nein, 1 = ja	**	

Fragebogen B-10: Zusammenhalt Klasse/Interface (Abschnitt 2 von 2)

Paket

Bedingung	Fragetext	Antwortskala	Gewicht	auto.
–	Bildet das Paket eine abgeschlossene Einheit? (Kriterien: eigenständiger Themenbereich, einheitliche Abstraktionsebene, enthaltene Komponenten gehören zusammen)	0 = nein, 1 = ja	***	
–	Liegen Vererbungsstrukturen vollständig im Paket? (eine Ausdehnung der Vererbungsstruktur auf Unterpakete ist unter Umständen akzeptabel)	0 = nein, 1 = ja	**	✔

Fragebogen B-11: Zusammenhalt Paket

System

Für den Zusammenhalt des Systems gibt es keinen Fragebogen. Wie bereits in Abschnitt A.4 ausgeführt, gibt es auf Systemebene keine neuen Aspekte, nach denen man im Zusammenhang mit dem Zusammenhalt fragen könnte.

B.5 Einheitlichkeit

Die Fragebögen sollen sicherstellen, dass die Standards und Konventionen eingehalten werden und der Entwurf einem einheitlichen Stil folgt. In spezifischen Modellen sollten diese Fragen noch durch Kriterien aus den Namenskonventionen, Entwurfsstandards etc. erweitert werden (hier repräsentiert durch entsprechende generische Fragen).

Klasse/Interface

Bedingung	Fragetext	Antwortskala	Gewicht	auto.
–	Ist die Namenskonvention für Klassen, Interfaces, Attribute und Operationen eingehalten worden?	0 = nein, 1 = ja	***	
–	Sind alle Entwurfsstandards eingehalten worden? (z. B. Parameterreihenfolge bei Operationen, Deklarationsreihenfolge von Klasseneigenschaften, Mindestumfang der Klassenschnittstelle)	0 = nein, 1 = ja	***	
–	Ist die Namensgebung für Attribute, Operationen und Parameter einheitlich? (z. B. gleiche Namen für Operationen mit dem gleichen Zweck)	0 = nein, 1 = ja	***	

Fragebogen B-12: Einheitlichkeit Klasse/Interface

Paket

Bedingung	Fragetext	Antwortskala	Gewicht	auto.
–	Ist beim Paketnamen die Namenskonvention für Pakete eingehalten worden?	0 = nein, 1 = ja	***	
NPP(this) > 0	Ist die Namensgebung für die enthaltenen Pakete einheitlich?	0 = nein, 1 = ja	***	
NCP(this)+ NIP(this)>0	Ist die Namensgebung für die enthaltenen Klassen/Interfaces einheitlich?	0 = nein, 1 = ja	***	

Fragebogen B-13: Einheitlichkeit Paket

System

Bedingung	Fragetext	Antwortskala	Gewicht	auto.
–	Ist die Namensgebung für Pakete insgesamt einheitlich?	0 = nein, 1 = ja	***	
–	Ist die Namensgebung für Klassen/Interfaces insgesamt einheitlich?	0 = nein, 1 = ja	***	
–	Ist die Namensgebung für Attribute, Operationen und Parameter insgesamt einheitlich?	0 = nein, 1 = ja	***	
–	Folgt der Entwurf insgesamt einem einheitlichen Stil?	0 = nein, 1 = ja	***	

Fragebogen B-14: Einheitlichkeit System

B.6 Dokumentierung

Die Fragebögen konzentrieren sich vor allem auf die semantischen Aspekte der Namensgebung und die Qualität der begleitenden Dokumentation (z. B. Kommentare oder separate Entwurfsdokumentation). In spezifischen Modellen sollten diese Fragen noch durch Kriterien aus den Dokumentationsstandards erweitert werden (hier repräsentiert durch eine generische Frage zu Beginn der Fragebögen).

Klasse/Interface

Bedingung	Fragetext	Antwortskala	Gewicht	auto.
–	Sind alle Dokumentationsstandards eingehalten worden?	0 = nein, 1 = ja	***	
–	Ist dokumentiert, wie die Klasse/das Interface verwendet wird?	0 = nein, 1 = ja	***	
–	Ist dokumentiert, welche Bedingungen die Implementierung erfüllen muss? (z. B. Beschreibung der Semantik durch Zustände und Klasseninvarianten, Beschreibung des Zusammenspiels von Operationen)	0 = nein, 1 = ja	***	
–	Ist die Klassen-/Interface-Dokumentation strukturiert, vollständig, präzise, konsistent und korrekt?	0 = nein, 1 = ja	**	
–	Ist der Name der Klasse/des Interfaces geeignet?	0 = nein, 1 = ja	***	
this∈ C	Sind die Attributnamen geeignet?	0 = nein, 1 = ja	***	
NOC(this) > 0	Ist für jede Operation dokumentiert, wie sie verwendet wird?	0 = nein, 1 = ja	***	
NOC(this) > 0	Ist für jede Operation dokumentiert, welche Bedingungen die Implementierung erfüllen muss? (z. B. Beschreibung der Semantik durch Vor- und Nachbedingungen)	0 = nein, 1 = ja	***	
NOC(this) > 0	Sind die Operationennamen geeignet?	0 = nein, 1 = ja	***	
NOC(this) > 0	Sind die Parameternamen geeignet?	0 = nein, 1 = ja	***	
–	Sind alle Namen in der Klasse/im Interface so unterschiedlich, dass keine Verwechslungsgefahr besteht?	0 = nein, 1 = ja	**	

Fragebogen B-15: Dokumentierung Klasse/Interface

Paket

Bedingung	Fragetext	Antwortskala	Gewicht	auto.
–	Sind alle Dokumentationsstandards eingehalten worden?	0 = nein, 1 = ja	***	
–	Ist der Name des Pakets geeignet?	0 = nein, 1 = ja	***	
–	Ist dokumentiert, wozu das Paket und die enthaltenen Klassen/Interfaces dienen?	0 = nein, 1 = ja	***	
–	Ist die Paketdokumentation strukturiert, vollständig, präzise, konsistent und korrekt?	0 = nein, 1 = ja	***	
–	Sind alle Namen im Paket so unterschiedlich, dass keine Verwechslungsgefahr besteht?	0 = nein, 1 = ja	**	

Fragebogen B-16: Dokumentierung Paket

System

Bedingung	Fragetext	Antwortskala	Gewicht	auto.
–	Sind alle Dokumentationsstandards eingehalten worden?	0 = nein, 1 = ja	***	
–	Stellt jedes Diagramm nur einen (oder wenige) Aspekt(e) des Entwurfs vollständig dar?	0 = nein, 1 = ja	***	
–	Ist die Systemdokumentation insgesamt sinnvoll aufgebaut und verständlich?	0 = nein, 1 = ja	***	
–	Ist die Form der Präsentation annehmbar? (Layout, Typographie, ...)	0 = nein, 1 = ja	**	
–	Werden Begriffe aus der Anwendungswelt angemessen verwendet?	0 = nein, 1 = ja	***	
–	Sind alle Namen im System so unterschiedlich, dass keine Verwechslungsgefahr besteht?	0 = nein, 1 = ja	**	

Fragebogen B-17: Dokumentierung System

B.7 Verfolgbarkeit

Für die Verfolgbarkeit ist es wichtig, dass Entwurfsentscheidungen auf die Anforderungen zurückgeführt werden können. Umgekehrt muss auch klar sein, welche Anforderungen Einfluss auf welche Entwurfsbestandteile haben.

Klasse/Interface

Bedingung	Fragetext	Antwortskala	Gewicht	auto.
–	Ist für die Klasse klar, für welche Anforderungen/ Anwendungsfälle sie eine Rolle spielt? (es kann allerdings Klassen geben, die nur aus Entwurfsgründen vorhanden sind, z. B. Adapter-Klassen; dann sollte mit ja geantwortet werden)	0 = nein, 1 = ja	***	

Fragebogen B-18: Verfolgbarkeit Klasse/Interface

Paket

Bedingung	Fragetext	Antwortskala	Gewicht	auto.
–	Ist für das Paket klar, für welche Anforderungen/ Anwendungsfälle es eine Rolle spielt? (Dies ist vor allem für Pakete wichtig, die Subsysteme repräsentieren. Viele Pakete werden allerdings nur aus Entwurfsgründen vorhanden sein, z. B. zur übersichtlichen Gliederung; dann sollte mit ja geantwortet werden)	0 = nein, 1 = ja	***	

Fragebogen B-19: Verfolgbarkeit Paket

System

Bedingung	Fragetext	Antwortskala	Gewicht	auto.
–	Ist für alle Anforderungen dokumentiert, wo im Entwurf sie umgesetzt sind?	0 = nein, 1 = ja	***	

Fragebogen B-20: Verfolgbarkeit System

B.8 Wartbarkeit

Die meisten Aspekte der Wartbarkeit sind bereits durch die Fragebögen der Kriterien abgedeckt. Daher wird hier nach der Änderbarkeit in Hinblick auf wahrscheinliche Änderungen gefragt. Die Frage, ob sich die wahrscheinlichsten zukünftigen Änderungen der Anforderungen leicht umsetzen lassen, lässt sich am besten durch eine Szenario-basierte Bewertung beantworten (vgl. Abschnitt 7.6.1).

Klasse/Interface

Bedingung	Fragetext	Antwortskala	Gewicht	auto.
–	Sind die wahrscheinlichsten zukünftigen Änderungen der Anforderungen leicht umzusetzen?	0 = nein, 1 = ja	***	
–	Sind die Entwurfsentscheidungen, die sich ändern können, vor dem Rest des Systems gemäß dem Geheimnisprinzip verborgen?	0 = nein, 1 = ja	**	

Fragebogen B-21: Wartbarkeit Klasse/Interface

Paket

Bedingung	Fragetext	Antwortskala	Gewicht	auto.
–	Sind die Entwurfsentscheidungen, die sich ändern können, vor dem Rest des Systems gemäß dem Geheimnisprinzip verborgen?	0 = nein, 1 = ja	**	

Fragebogen B-22: Wartbarkeit Paket

System

Bedingung	Fragetext	Antwortskala	Gewicht	auto.
–	Sind die wahrscheinlichsten zukünftigen Änderungen der Anforderungen leicht im Entwurf umzusetzen?	0 = nein, 1 = ja	***	
–	Sind die Entwurfsentscheidungen, die sich ändern können, vor dem Rest des Systems gemäß dem Geheimnisprinzip verborgen?	0 = nein, 1 = ja	**	

Fragebogen B-23: Wartbarkeit System

Anhang C

Dokumente zum Softwarepraktikum

Dieser Anhang umfasst Dokumente zum Softwarepraktikum, das die Abteilung Software Engineering im Sommersemester 2001 im Studiengang Softwaretechnik durchgeführt hat. Abschnitt C.1 enthält die Aufgabenstellung, Abschnitt C.2 eine Aufstellung der Anforderungen (als Grundlage des Kunden/Betreuers für die Kundenbefragung in der Analyse) und Abschnitt C.3 das Begriffslexikon.

C.1 Aufgabenstellung

C.1.1 Organisation

In diesem Praktikum müssen Sie gruppenweise ein kleines Software-Projekt durchführen und damit Ihr Wissen aus der Vorlesung *Einführung in die Softwaretechnik I* umsetzen. Sie planen das Projekt selbst und besprechen die Ergebnisse jeweils mit Ihrem Betreuer. Dieser ist auch gleichzeitig der Kunde, also der Abnehmer für Ihre Ergebnisse.

Die Kenntnisse aus den Vorlesungen *Einführung in die Softwaretechnik I* inklusive *Übungen*, *Einführung in die Informatik I & II* sowie *Programmentwicklung* werden vorausgesetzt. Sie bilden eine wichtige Grundlage für Ihre Arbeit.

Alle Gruppen sind jeweils einem Betreuer zugeordnet. Diesen Betreuer können Sie *nach Terminabsprache* aufsuchen und ihn bei allen auftretenden Problemen zu Rate ziehen. Zu den von Ihnen geplanten Meilensteinterminen müssen Sie Ihren Betreuer aufsuchen, die entsprechenden Unterlagen abgeben und mit ihm durchsprechen.

In einem ersten Schritt müssen Sie Ihr Projekt selbst planen. Weiter unten sind allerdings einige Rahmenbedingungen für diese Planung von unserer Seite angegeben. Diese müssen Sie unbedingt beachten. Der Plan wird dann von Ihrem Betreuer genehmigt. Sie sind dann an Ihren Plan gebunden, d.h. die darin genannten Termine sind

unbedingt einzuhalten. Sollten Sie feststellen, dass Sie Ihren Plan ändern müssen, so tun Sie das bitte und legen diesen veränderten Plan wieder Ihrem Betreuer vor. Es gilt immer der letzte von Ihrem Betreuer genehmigte Plan!

Achtung: Sehr kurzfristige Änderungen am Projektplan sind in der Regel nicht möglich, nur langfristige Anpassungen werden akzeptiert (also mindestens eine Woche vor dem nächsten Meilensteintermin). Das Versäumen der im Projektplan genannten *Termine* führt beim zweiten Mal zum *Scheinverlust*!

Planen Sie alle Abgaben, Kundenbefragungen und Besprechungstermine mit Ihrem Betreuer und sprechen Sie diese vorher mit ihm ab. Sie sollten nur in Ausnahmefällen ohne vorherige Absprache bei Ihrem Betreuer aufkreuzen. Scheuen Sie sich jedoch nicht, Probleme rechtzeitig anzusprechen. Das gilt insbesondere für Probleme bei der Zusammenarbeit der Gruppen und dem „Aussteigen“ einzelner Gruppenmitglieder.

Führen Sie während des Software-Praktikums einen Stundenzettel, in dem Sie alle Arbeitsstunden verzeichnen. Eine Kopie der Stundenzettel ist am Ende dem Betreuer abzugeben. Der Inhalt der Stundenzettel hat keinen Einfluss auf die Bewertung.

C.1.2 Projektdurchführung

Ihr Projekt soll mindestens die folgenden Meilensteine enthalten:

1. Analyse
2. Projektplanung
3. Spezifikation (inklusive Begriffslexikon)
4. Review und Überarbeitung der Spezifikation
5. Entwurf
6. Implementierung
7. Benutzerhandbuch
8. Test

Zu jedem Meilenstein müssen Sie in Ihrem Projektplan mindestens ein definiertes Abgabedatum und die abzugebenden Dokumente eintragen. Alle Meilensteindokumente (Abgaben) werden durch Ihren Betreuer geprüft und abgenommen. Die Prüfung der Dokumente durch den Betreuer erfolgt allerdings nicht im Sinne einer Qualitätssicherung; dafür sind Sie selbst verantwortlich (und Sie sollten sie *vor der Abgabe* durchführen!). Sie dürfen Ihr Projekt im Übrigen auch durch weitere Meilensteine untergliedern. Planen Sie auch Aufwand für die Überarbeitung der Dokumente ein. Erfahrungsgemäß werden nach der Durchsicht durch den Betreuer größere Änderungen notwendig.

Die *Analyse* wird in Form einer Kundenbefragung stattfinden. Jeder Betreuer trifft sich mit seinen Gruppen zu einer gemeinsamen Sitzung. Sie haben dann Zeit, Ihre Fragen an den Kunden, repräsentiert durch Ihren Betreuer, zu stellen. Sie sollten zu diesem Termin unbedingt vorbereitet erscheinen, also vorbereitete Fragen haben.

Auf der Grundlage Ihrer Problemanalyse erstellen Sie dann einen *Projektplan*. Dieser muss Ihrem Betreuer zur Genehmigung vorgelegt werden. Nach jeder Überarbeitung

muss der Plan dem Betreuer zur Genehmigung vorgelegt werden. Dieser Projektplan regelt die weiteren Termine. Planen Sie auch Pufferzeiten ein und vergessen Sie die Zeiten nicht, die Sie eventuell für das Projekt nicht zur Verfügung stehen (insbesondere in der vorlesungsfreien Zeit, z. B. durch Prüfungsvorbereitung, Urlaub usw.).

Es kommt darauf an, dass Sie einen Plan erstellen, dem Sie auch tatsächlich folgen können. Sie brauchen keinen „Scheinplan" aufzustellen, der uns dauernde Aktivität vorgaukelt. Wichtig ist, dass Sie zu den Meilensteinterminen die geforderten Dokumente in guter Qualität abgeben. Dazu benötigen Sie sicher die anvisierte Stundenzahl (s.u.). Sie müssen diese Stunden aber so einplanen, dass Sie diese auch zu gegebener Zeit leisten können. Bedenken Sie das bei der Aufstellung des Projektplans. Ob Sie noch im Plan sind, können Sie dann jederzeit leicht durch einen Vergleich mit den Stundenzetteln und den erbrachten Ergebnissen feststellen. Führen Sie daher den Stundenzettel gewissenhaft und verschieben Sie das Aufschreiben der Arbeitszeiten nicht ans Projektende.

C.1.3 Projektrahmen

Sie und Ihre beiden Mitstreiter stellen ein Entwicklungsteam dar. Sie bekommen von einem kleinen Verkehrsbetrieb einen Software-Entwicklungsauftrag für ein kleines, maßgeschneidertes Auskunftssystem. Damit möchte Ihr Auftraggeber seinen Fahrgästen eine attraktive Möglichkeit zur Fahrtenplanung bieten.

Normalerweise berechnen Sie einen Stundensatz von 200,- DM pro Entwicklerarbeitsstunde, der Kunde möchte aber ein Festpreisprojekt und ist bereit, dafür 96.000,- DM zu bezahlen. Das haben Sie dem Verkehrsbetrieb auch zugesagt. Natürlich ist Ihr Chef sehr daran interessiert, dass Sie diesen Kosten- und in diesem Falle auch Zeitrahmen genau einhalten, da Ihre Firma sonst bei diesem Auftrag keinen Gewinn machen kann. Ihr Chef möchte daher eine Abrechnung der Stunden, die Ihre Entwicklergruppe für dieses Projekt aufgebracht hat.

Der Verkehrsbetrieb, Ihr Kunde, verwendet für all seine Projekte die Programmiersprache Java. Damit soll unter anderem sichergestellt werden, dass das Programm auf verschiedenen Rechnern und unter verschiedenen Betriebssystemen gleichermaßen zum Einsatz kommen kann. Ihr Chef konnte für Sie diesen Auftrag nur abschließen, weil er Ihre Kompetenz in Java besonders herausgestellt hat.

Ihre Aufgabe ist es, dem Kunden ein qualitativ hochwertiges, genau auf seine Bedürfnisse zugeschnittenes Programm zu erstellen. Auf darüber hinausgehende Leistungen, die Ihnen Ihr Kunde nicht honoriert, müssen Sie dabei aber verzichten.

C.1.4 Aufgabenstellung

Es ist ein kleines, nur für einen Benutzer ausgelegtes Fahrplaninformationssystem zu entwickeln. Das System sollte einfach zu bedienen sein und es einem potentiellen Fahrgast ermöglichen, eine Verkehrsverbindung zwischen zwei Haltestellen zu finden. Der Fahrgast gibt hierzu den gewünschten Startzeitpunkt und die Start- und Zielhaltestelle ein. Daraufhin errechnet das System eine günstige Verbindung und gibt diese inklusive Liniennummer und Umsteigehaltestellen aus.

Das System optimiert die Verbindung nach Wahl des Fahrgasts. Es bietet die folgenden Optimierungsziele an:

- frühestmögliche Ankunftszeit
- kürzeste Fahrtzeit
- wenigste Umsteigehaltestellen

Die Ergebnisse werden dem Fahrgast angezeigt. Er kann sie sich aber auch in eine Datei im HTML-Format drucken lassen.

Das System entnimmt die Fahrpläne der Linien einer Fahrplandatei. Diese Datei kann nur verändert werden, wenn das Fahrplaninformationssystem nicht von Fahrgästen benutzt wird. Dies geschieht durch einen Servicetechniker, der hierzu einen nur ihm zugänglichen Programmteil verwendet. Hiermit lassen sich Fahrplandaten verändern, löschen oder neu eingeben.

Das Programm muss unter Unix laufen und eine graphische Benutzungsoberfläche besitzen.

C.1.5 Entwicklungsumgebung, Werkzeuge und Richtlinien

Wie bereits angesprochen muss das gesamte Programm in Java (Version 2) geschrieben werden. Das Java 2 Software Development Kit (kurz JDK) in der Version 1.2 steht Ihnen auf den Rechnern des ehemaligen Grundstudium-Pools sowie auf den Linux-Rechnern des Pools zur Verfügung. Das JDK in den Versionen 1.2 oder 1.3 gibt es auch kostenlos für eine ganze Reihe von Rechnerplattformen, insbesondere Linux und Windows. Sie können diese Versionen gerne verwenden, wir können aber keine Hilfestellung dafür bieten.

Bei der Abgabe muss das Programm auf unseren Maschinen (Sun Solaris) mit installiertem JDK 1.2.2 ausführbar sein. Bitte achten Sie darauf! Insbesondere wenn Sie unter Windows entwickelt haben, sollten Sie auf jeden Fall Ihr Programm *vor der Abgabe unter Unix testen*, z. B. im Pool. Geben Sie immer mit Ihrem Quellcode auch eine compilierte, ausführbare Version Ihres Programmes ab. Diesem Programm muss eine Testdatei (README) beiliegen, aus der hervorgeht, wie das Programm gestartet werden kann. Nicht ausführbare Programme gelten als nicht abgegeben!

Für die Programmierung in Java müssen Sie sich an die Programmierrichtlinie von Sun halten. Diese ist weit verbreitet und wurde auch schon für die Scheinaufgabe zur Vorlesung *Programmentwicklung* gefordert. Sie finden Sie im Web unter `http://java.sun.com/docs/codeconv/index.html`. Eine Kopie der Richtlinie ist auch im Semesterapparat zur Vorlesung *Programmentwicklung* einzusehen. Beachten Sie diese Richtlinie schon beim Entwurf, da sie zum Beispiel auch Aussagen über die konforme Wahl von Bezeichnern macht.

Für die Spezifikation und den Entwurf müssen Sie das CASE-Werkzeug *Innovator* verwenden. Die Spezifikation wird mit diesem Werkzeug als Use-Case-Spezifikation erstellt, für den Entwurf sind UML-Diagramme anzufertigen. Vergessen Sie dabei nicht, die Diagramme und die entsprechenden Elemente ausreichend zu beschriften und zu kommentieren. Für Spezifikation wie für den Entwurf sind weitere Dokumententeile notwendig, die nur Texte enthalten (in der Spezifikation zum Beispiel die nicht-funktionalen Anforderungen). Diese Texte können ebenfalls mit Innovator im selben Dokument erstellt werden. Dies ist aber nicht sehr komfortabel. Sie dürfen diese Texte daher auch als ein separates Dokument erstellen. Dieses Dokument ist

dann Ihr Hauptdokument. Sehen sie darin an geeigneter Stelle Kapitel für die Innovator-Dokumentation vor. Schreiben Sie in diesem Kapitel dann nur einen Verweis auf das entsprechende Innovator-Dokument.

Innovator ist wie das JDK auf den Rechnern des Grundstudiumpools installiert. Die Version dort enthält im Gegensatz zur kostenlos erhältlichen PC-Demo-Version keine Beschränkungen. Sie können damit auch parallel am gleichen Dokument (= Repository) arbeiten. Um allen Problemen aus dem Weg zu gehen, verwenden Sie diese unbeschränkte Version. Lösungen, die auf Grund der Demo-Beschränkungen etwas zu einfach geraten sind, werden nicht akzeptiert. Sollte es im Grundstudiumspool zu größeren Engpässen und Problemen kommen, informieren Sie uns.

Über die Bedienung und den Aufruf von Innovator informiert Sie eine kurze Anleitung und die Online-Dokumentation. Konsultieren Sie diese beiden Hilfestellungen, *bevor* Sie mit Fragen zu Ihrem Betreuer gehen! Verwenden Sie in der Anleitung nicht beschriebene Möglichkeiten von Innovator nur dann, wenn Sie sicher wissen, was Sie tun. Wir können hierfür keine Hilfestellungen geben. Insbesondere ist es nicht vorgesehen, dass Sie den in Innovator eingebauten Code-Generator für Java verwenden.

C.1.6 Abgaben

Wie bereits erwähnt müssen Sie zu allen Meilensteinterminen die entsprechenden Dokumente bei Ihrem Betreuer abgeben. Was Sie wann abzugeben haben, muss Ihrem Projektplan entnommen werden können. Die entsprechenden Dokumente müssen genau am geplanten Tag abgegeben werden, d.h. sie müssen spätestens am Morgen des folgenden Tages (8.00 Uhr) beim Betreuer angekommen sein!

Beim Stand der heutigen Technik sollte es Ihnen möglich sein, Abgaben sauber zu gestalten, d.h. ein Textverarbeitungsprogramm zu verwenden. Welches Programm Sie verwenden, bleibt Ihnen überlassen.

Am Ende des Praktikums (Abgabetermin genau beachten) geben Sie ein elektronisches Archiv (durch Verzeichnisse gegliedert, mit `tar` und `gzip` zusammengepackt und per Mail an den Betreuer versandt) ab, das den folgenden Inhalt haben muss:

- das ausführbare Programm (lauffähig auf unseren Rechnern mit JDK 1.2) und alle für die Ausführung notwendigen Dateien,
- den Quelltext des Programms,
- evtl. erstellte Testrahmen, Testprogramme oder Testdaten,
- eine Textdatei (README), die den Übersetzungsvorgang, den Programmstart, die Abweichungen des fertigen Programms von der Spezifikation, Fehler und Einschränkungen beschreibt,
- alle Meilensteindokumente und den Projektplan im Postscript-Format,
- sowie all diese Dokumente in ihrem Originalformat, also im entsprechenden Format der verwendeten Textverarbeitung.

Alle Dokumente in diesem Archiv müssen auf dem aktuellsten Stand sein. Außerdem müssen Sie zu diesem Termin eine Kopie Ihres Stundenzettels abgeben und das fertige Programm vorführen können.

Bitte beachten Sie, dass diese Abgabe vollständig und rechtzeitig erfolgen muss. Dies ist eine wichtige Bedingung für den Erhalt des Scheins!

C.2 Anforderungen

C.2.1 Übersicht

Das System, ein Fahrplanauskunftssystem, sollte einfach zu bedienen sein und es einem potentiellen Fahrgast ermöglichen, eine Verbindung zwischen zwei Haltestellen zu finden. Der Fahrgast gibt hierzu den gewünschten Startzeitpunkt die Start- und Zielhaltestelle und seinen Optimierungswunsch ein. Daraufhin errechnet das System eine günstige Verbindung und gibt diese inklusive der Namen (Liniennummer) der zu benutzenden Linien und die Umsteigehaltestellen aus.

Das System entnimmt die Fahrpläne der Linien einer Fahrplandatei. Diese Datei kann nur verändert werden, wenn das Fahrplaninformationssystem nicht von Fahrgästen benutzt wird. Dies geschieht durch einen Servicetechniker, der hierzu einen nur ihm zugänglichen Programmteil verwendet. Hiermit lassen sich Fahrplandaten verändern, löschen oder neu eingeben.

Die Implementierung soll in Java durchgeführt werden. Das Programm soll unter Unix (Solaris/Linux) laufen und mit einer graphischen Benutzungsschnittstelle ausgestattet sein (Java Swing). Das System soll nur für einen Benutzer ausgelegt sein.

C.2.2 Systemumgebung

Die Systemumgebung des Verkehrsbetriebs besteht aus einer Unix-Workstation, die nur einen einzigen Bildschirm mit graphischer Benutzungsoberfläche besitzt. Daran sollte auch auf keinen Fall etwas geändert werden. Das Endprodukt muss demzufolge auf einer UNIX-Maschine lauffähig sein und mit einer graphischen Benutzungsoberfläche ausgestattet werden. Für die Programmierung der Benutzungsoberfläche muss Java Swing verwendet werden, dies gibt der Verkehrsbetrieb im Sinne einer einheitlichen Benutzungsoberfläche für all seine Applikationen so vor.

Die Ausgaben des Programmes erfolgen auf dem Bildschirm oder in eine Datei (im Falle der HTML-Ausgabe), die Eingabe über Maus und Tastatur. Des Weiteren liest das Programm Daten aus einer Fahrplandatei bzw. greift schreibend auf diese Datei zu. Eine solche Datei stellt der Verkehrsbetrieb als Beispiel zur Verfügung. Zu beziehen ist diese Datei von unseren Web-Seiten (Adresse siehe Aufgabenblatt). Die Hardware verfügt über eine Festplatte von mindestens 10 MByte Kapazität zum Speichern der Programmdaten. Der Auftraggeber hat keinerlei weitere Peripheriegeräte.

In der jetzigen Version ist es vorgesehen, dass max. ein Benutzer das Programm gleichzeitig ausführen kann. Es müssen also keine Konflikte beachtet werden, die auftreten können, wenn mehrere Personen gleichzeitig mit den gleichen Daten umgehen.

C.2.3 Modi des Fahrplanauskunftssystems

Das Fahrplanauskunftssystem kann in zwei Modi arbeiten: dem Fahrgast-Modus, in dem ein potentieller Fahrgast Verbindungsanfragen stellen kann, und in dem Admin-Modus, in dem neue Linien und Abfahrtszeiten eingegeben werden können.

C.2.4 Fahrgast-Modus

Der Benutzer muss unter Eingabe der Start- und Zielhaltestelle sowie des frühestmöglichen Startzeitpunkts eine mögliche Verbindung erfragen können. Der Benutzer gibt zusätzlich an, unter welchem Aspekt die Verbindung optimiert werden soll. Das System berechnet dann die beste Verbindung. Es gibt folgende Optimierungsziele:

- *frühestmögliche Ankunftszeit*, d.h. die Verbindung, die zum frühestmöglichen Zeitpunkt an der Zielhaltestelle ankommt und frühestens zum gegebenen Startzeitpunkt startet.
- *kürzeste Fahrtzeit*, d.h. die Verbindung, die die kürzeste Zeitspanne zwischen Einsteigen an der Start- und Aussteigen an der Zielhaltestelle ermöglicht. Der Startzeitpunkt sollte dabei nicht mehr als eine Stunde vom eingegebenen, frühestmöglichen Startzeitpunkt entfernt sein. Es wird also die Verbindung mit der kürzesten Fahrtzeit gewählt, deren Startzeitpunkt nicht länger als eine Stunde vom frühestmöglichen Startzeitpunkt entfernt liegt. Gibt es in dieser Zeit keine Verbindung, so wird die Verbindung mit der frühestmöglichen Ankunftszeit ausgegeben (s.o.).
- *wenigste Umsteigehaltestellen*, d.h. die Verbindung, die am wenigsten Umsteigepunkte enthält. Gibt es mehrere Möglichkeiten, so wird die Verbindung angezeigt, die die frühestmögliche Ankunftszeit verspricht.

Es werden alle zu benutzenden Linien angegeben sowie die Umsteigehaltestellen, die Abfahrtszeitpunkte an der Starthaltestelle und allen Umsteigehaltestellen und der Ankunftszeitpunkt an der Zielhaltestelle. Umsteigezeiten (Aufenthaltszeiten an der Umsteigehaltestelle) von mehr als 60 Minuten führen dazu, dass eine Verbindung ungültig ist. Solche Verbindungen werden grundsätzlich nicht betrachtet. Findet das System mehr als eine Verbindung, die das gewünschte Optimierungsziel erfüllt, so werden alle möglichen Verbindungen angezeigt.

Wurde eine Verbindung gefunden und angezeigt, so muss der Benutzer die Möglichkeit haben, sich die nächste Verbindung anzeigen zu lassen. Das System soll dann eine Verbindung anzeigen, die mindestens eine Minute später startet, aber dieselben Optimierungsziele verfolgt.

Wurde eine Verbindung gefunden und angezeigt, so muss der Benutzer die Möglichkeit haben, die Verbindung „auszudrucken“. Es wird eine HTML-Datei mit einem generierten (aber beliebigen) Namen in das Verzeichnis geschrieben, in dem das Programm gestartet wurde. Diese Datei enthält eine HTML-Seite, auf der die entsprechende Verbindung beschrieben ist. Das Programm überschreibt niemals existierende Dateien, sondern generiert einen eindeutigen Dateinamen. Dieser Dateiname wird dem Benutzer angezeigt. Es obliegt nun dem Benutzer, die Datei weiterzubearbeiten, das Programm fasst diese Datei nicht mehr an. Es ist insbesondere nicht für das Anzeigen in einem HTML-Browser und das Ausdrucken selbst zuständig.

Es muss möglich sein, das Programm zu beenden.

C.2.5 Admin-Modus

Das System muss es einem Benutzer ermöglichen, in den Admin-Modus zu wechseln. Um den Admin-Modus zu betreten, fragt das System ein Passwort ab, danach ist der Benutzer im Admin-Modus und kann die folgenden Aktionen durchführen:

- neue Linie einfügen (dazu gehören alle Haltestellen und die Fahrzeiten zwischen den Haltestellen)
- neue Abfahrtszeit eingeben (an den beiden Endhaltestellen)
- Abfahrtszeit löschen
- Linie löschen
- Admin-Modus verlassen
- Passwort ändern
- Anzeigen aller Linien (nur Name und die beiden Endhaltestellen)
- Anzeigen einer Linie mit allen Informationen, die zu dieser Linie gehören (Haltestellen, Fahrzeiten zwischen den Haltestellen, Abfahrtszeiten an den beiden Endhaltestellen)

C.2.6 Fahrplandatei

Eine Beispielfahrplandatei mit den notwendigen Informationen wird auf den Web-Seiten angeboten. Diese Datei enthält die Linien S1 bis S6 sowie U1, U6 und U14. Diese Datei soll direkt verwendet werden, das heißt, das Programm soll diese Datei lesen können und auch eigene Informationen in diesem Format speichern.

Der Einfachheit halber fahren alle Bahnen immer an den Endhaltestellen los. Es gibt also keine Bahnen, die an Zwischenhaltestellen starten oder enden. Deshalb werden in der Datei nur die Abfahrtszeiten an den Endhaltestellen gespeichert. Es wird ebenso kein Unterschied zwischen Ankunfts- und Abfahrtszeiten an Zwischenhaltestellen gemacht, d.h. die Züge haben an Zwischenhaltestellen keinen Aufenthalt, die Ankunfts- und die Abfahrtszeit ergibt sich durch die Startzeit an der Endhaltestelle und die Addition der Fahrzeiten.

Es soll keine Unterscheidung zwischen Wochentagen und Wochenende gemacht werden. Alle Linien fahren an allen Tagen zu den gleichen Zeitpunkten.

Aufbau der Fahrplandatei

Die Fahrplandatei ist eine reine Textdatei, die aber auch Umlaute enthalten kann. Die Zeichen sind nach ISO 8859-1 kodiert (landläufig als Latin-1-Zeichensatz bezeichnet; dies ist der Standardzeichensatz unter Solaris/Linux).

Die Fahrplandatei besteht aus einzelnen Abschnitten, die jeweils genau eine Linie beschreiben. Die einzelnen Abschnitte sind durch eine Leerzeile voneinander getrennt. Außer dem Zeilentrenner steht kein Zeichen in diesen Zeilen. Nach dem letzten Abschnitt endet die Datei unter Umständen ohne eine zusätzliche Leerzeile. Die Abschnitte sind nach folgendem Muster aufgebaut:

1. Zeile: Name der Linie (Bsp.: S1)

2. Zeile: Name der ersten Endhaltestelle (Bsp.: Herrenberg)

3. Zeile: Name der anderen Endhaltestelle (Bsp.: Plochingen)

4. Zeile: Abfahrzeiten an der ersten Endhaltestelle

5. Zeile: Abfahrzeiten an der anderen Endhaltestelle

alle weiteren Zeilen: Haltestellen zwischen erster und zweiter Endhaltestelle

Diese Darstellung impliziert eine Vorzugsrichtung, nämlich von der ersten Endhaltestelle zur anderen Endhaltestelle. In Wirklichkeit hat eine Linie keine solche Richtung, sie hat nur zwei Endhaltestellen. Diese implizite Richtung ist nur für die Interpretation der Daten notwendig (und der Darstellung des Fahrplans im Programm), sie wirkt sich aber ansonsten nicht auf den Programmablauf aus.

Die Abfahrtszeiten sind die Abfahrtszeiten an einem Kalendertag (von 0.00 Uhr bis 23.59 Uhr), nicht die üblicherweise in Fahrplänen abgedruckte Reihenfolge von Betriebsbeginn am frühen Morgen bis Betriebsende. Die einzelnen Uhrzeiten werden im Format SS:MM (Stunde, Minute) eingetragen und durch Kommata getrennt.

Die Liste der Zwischenhaltestellen wird nach folgendem Schema erstellt: In jeder Zeile steht eine Zwischenhaltestelle. Dabei wird zunächst die Fahrzeit zwischen der vorherigen Haltestelle und der in dieser Zeile beschriebenen Haltestelle in Minuten angegeben, durch ein Komma getrennt folgt der Name der Haltestelle. Die erste Endhaltestelle (also gewissermaßen die Starthaltestelle) wird in dieser Liste nicht aufgeführt. Der erste Eintrag enthält also die Fahrzeit zwischen Endhaltestelle und der dort beschriebenen zweiten Haltestelle. Die zweite Endhaltestelle wird hingegen in der letzten Zeile aufgeführt, dort muss schließlich auch die Fahrzeit zwischen vorletzter und Endhaltestelle vermerkt werden.

Beispiel: Linie S1 von Herrenberg nach Plochingen (und zurück).

```
S1
Herrenberg
Plochingen
04:47,05:17,05:47,06:17,07:17,07:47,08:17,08:47,09:17,09:47,10:17,10:47,11:
17,11:47,12:17,12:47,13:17,13:47,14:17,14:47,15:17,15:47,16:17,16:47,17:17,
17:47,18:17,18:47,19:17,19:47,20:17,20:47,21:17,21:47,22:47,23:17
00:09,04:39,05:09,05:39,06:09,06:39,07:09,07:39,08:09,08:39,09:09,09:39,10:
09,10:39,11:09,11:39,12:09,12:39,13:09,13:39,14:09,14:39,15:09,15:39,16:09,
16:39,17:09,17:39,18:09,18:39,19:09,20:09,20:39,21:09,21:39,22:09,22:39,23:
09,23:39
4,Nufringen
2,Gärtringen
3,Ehningen
3,Hulb
2,Böblingen
3,Goldberg
5,Rohr
2,Vaihingen
2,Österfeld
2,Universität
5,Schwabstraße
2,Feuersee
1,Stadtmitte
2,Hauptbahnhof
4,Bad Cannstatt
3,Neckarstadion
2,Untertürkheim
3,Obertürkheim
2,Mettingen
2,Esslingen
2,Oberesslingen
```

```
3,Zell
2,Altbach
3,Plochingen
```

C.2.7 Was das Programm nicht leisten soll

- Änderung der Fahrzeiten
- Hinzufügen neuer Haltestellen
- alle Verbindungen in gewünschtem Zeitrahmen anzeigen
- Unterscheidung von Wochenenden und Wochentagen

C.2.8 Qualitätsanforderungen

Portabilität. Die Software soll zum einen *geräteunabhängig* sein, d.h. keine Merkmale spezieller Geräte enthalten oder verwenden. Weiterhin soll die Software *abgeschlossen* sein, d.h. keine Schnittstellen zu anderen Programmen enthalten.

Bedienbarkeit. Der Benutzer soll rasch verstehen können, wie er mit der Software umgehen muss.

Wartbarkeit. Beachten Sie die folgenden möglichen Erweiterungen:

- bei der Verbindungssuche nicht nur die beste, sondern auch weitere mögliche Verbindungen ausgeben
- Verbindungsanfrage um gewünschte Umsteigehaltestellen erweitern
- Unterscheidung von Wochentagen und Wochenende im Fahrplan
- im Administratormodus können auch die Fahrzeiten zwischen den Haltestellen geändert werden
- im Administratormodus können auch neue Haltestellen aufgenommen werden

Alle weiteren Qualitäten spielen keine entscheidende Rolle, sie sind daher nicht weiter zu beachten.

C.3 Begriffslexikon

Abfahrtszeit. Zeitpunkt, zu dem der Zug einer Linie eine Haltestelle verlässt.

Admin-Modus. Zustand des Fahrplaninformationssystems. In diesem Zustand kann ein Servicetechniker die Fahrplandaten des Systems ändern.

Ankunftszeit. Zeitpunkt, zu dem der Zug einer Linie eine Haltestelle erreicht. Außer an Endhaltestellen verlässt der Zug eine Haltestelle zum selben Zeitpunkt, d.h. Ankunftszeit und Abfahrtszeit sind identisch.

Auftraggeber. Der Auftraggeber ist eine Firma, die bei den Software-Entwicklern das Fahrplaninformationssystem in Auftrag gibt und die Software abnimmt und bezahlt. Der Auftraggeber ist in diesem Projekt der Verkehrsbetrieb VVS.

Benutzer. Ein Benutzer ist eine Person, die das Fahrplaninformationssystem benutzt. Es kann sich dabei um einen Fahrgast oder einen Servicetechniker handeln.

Endhaltestelle. Eine Endhaltestelle ist eine spezielle Haltestelle, an der ein Zug einer Linie jede seiner Fahrten beginnt oder beendet.

Fahrgast. Ein Fahrgast ist eine Person, die die Transportmöglichkeiten eines Verkehrsbetriebs nutzen möchte und auch nutzt. Er ist der Benutzer des Fahrplanauskunftsystems. Er kann das Fahrplanauskunftssystem ausschließlich im Fahrgast-Modus nutzen.

Fahrgast-Modus. Zustand des Fahrplaninformationssystems. In diesem Zustand können Fahrgäste Verbindungen erfragen.

Fahrplan. synonym zu Fahrplandaten

Fahrplandatei. Die Fahrplandatei ist eine Datei, die die Fahrplandaten enthält.

Fahrplandaten. Die Fahrplandaten enthalten Informationen über die Linien des Verkehrsbetriebs. Zu jeder Linie werden alle Haltestellen inklusive der beiden Endhaltestellen, die Fahrzeiten zwischen den Haltestellen sowie die Abfahrtszeiten an den beiden Endhaltestellen Fahrzeiten zwischen diesen Haltestellen gespeichert. Weiterhin müssen alle Abfahrtszeiten an beiden Endhaltestellen einer Linie und der Name der Linie (die Liniennummer) eingetragen sein.

Fahrplaninformationssystem. Das zu realisierende Software-System zur Ermittlung von Verbindungen.

Fahrzeit. Unter der Fahrzeit versteht man die Zeitdauer, die ein Fahrgast oder ein Zug für die Fahrt zwischen zwei Haltestellen benötigt, ein Fahrgast kann dabei auch umsteigen. Die Fahrzeit ist die Zeit vom Einsteigen/Losfahren an der einen Haltestelle bis zum Aussteigen/Ankommen an der anderen Haltestelle.

frühestmöglicher Startzeitpunkt. Der frühestmögliche Startzeitpunkt ist der Startzeitpunkt, den der Fahrgast bei einer Abfrage angibt. Die Verbindung darf zu diesen oder einem späteren Zeitpunkt starten.

Haltestelle. Eine Haltestelle ist ein Ort, den ein Zug einer Linie bei seiner Fahrt anfährt. Fahrgäste können den Zug nur an Haltestellen betreten oder verlassen. Haltestellen werden eindeutig durch ihren Namen gekennzeichnet. Außer an Endhaltestellen sind Ankunftszeit und Abfahrtszeit eines Zuges an einer Haltestelle identisch.

Linie. Eine Linie bestimmt sich durch ihren Namen (die Liniennummer), aus einer Menge von Haltestellen, die von den Zügen der Linie angefahren werden, sowie den Fahrzeiten zwischen den einzelnen Haltestellen und den Abfahrtszeiten an den beiden Endhaltestellen. Die Haltestellen sowie die Fahrzeiten sind konstant. Jede Linie hat für jede ihrer Fahrten die gleichen Endhaltestellen.

Liniennummer. Eindeutiger Name einer Linie, z. B. S1, U14.

Optimierungsziel. Vorgabe für die Auswahl einer möglichen Verbindung. Es kann die Verbindung mit der frühestmöglichen Ankunftszeit, mit der kürzesten Fahrtzeit oder mit den wenigsten Umsteigehaltestellen als Optimierungsziel gewählt werden.

Servicetechniker. Der Servicetechniker ist ein Mitarbeiter des Verkehrsbetriebs, der die Fahrplandaten verwaltet. Er nimmt Änderungen an den Fahrplandaten vor, gibt neue Daten ein oder löscht vorhandene Angaben. Er erledigt diese Aufgaben im Admin-Modus des Fahrplanauskunftssystems.

Starthaltestelle. Die Starthaltestelle ist eine spezielle Haltestelle, an der der Fahrgast seine Fahrt beginnen möchte.

Startzeitpunkt. Zeitpunkt, an dem der Fahrgast die Fahrt von der Starthaltestelle aus beginnen möchte.

Umsteigehaltestelle. Eine Umsteigehaltestelle ist eine spezielle Haltestelle, die von den Zügen mehrerer Linien angefahren wird. An dieser ist deshalb ein Wechsel vom Zug einer Linie in den Zug einer anderen Linie möglich.

Umsteigezeit. Umsteigezeiten sind die Aufenthaltszeiten an einer Umsteigehaltestelle. Die Umsteigezeiten dürfen bei einer gültigen Verbindung maximal 60 Minuten und mindestens 5 Minuten betragen.

Verbindung. Eine Verbindung zeigt an, wie ein Fahrgast unter Benutzung von Zügen der vorhandenen Linien von einer Starthaltestelle zu einer gewünschten Zielhaltestelle gelangt. Zu einer Verbindung zählt die Angabe (Liniennummer) aller zu benutzenden Linien, der Umsteigehaltestellen, des Abfahrtszeitpunktes an der Starthaltestelle und an allen Umsteigehaltestellen und des Ankunftszeitpunkts an der Zielhaltestelle. Umsteigezeiten von weniger als 5 oder mehr als 60 Minuten führen dazu, dass eine Verbindung ungültig ist.

Verbindung mit den wenigsten Umsteigehaltestellen. Verbindung, bei der die Anzahl von Umsteigehaltestellen verglichen mit anderen Verbindungen zwischen Start- und Zielhaltestelle am geringsten ist. Gibt es mehrere Möglichkeiten, so wird die Verbindung gewählt, die die frühestmögliche Ankunftszeit verspricht. Der Startzeitpunkt darf in keinem Fall mehr als 60 Minuten nach dem frühestmöglichen Startzeitpunkt liegen.

Verbindung mit frühestmöglicher Ankunftszeit. Eine Verbindung, die zum frühestmöglichen Zeitpunkt (Ankunftszeit) an der Zielhaltestelle ankommt und frühestens zum gegebenen Startzeitpunkt startet.

Verbindung mit kürzester Fahrtzeit. Eine Verbindung, die die kürzeste Zeitspanne zwischen Abfahrtszeit an der Start- und Ankunftszeit an der Zielhaltestelle ermöglicht. Der Startzeitpunkt darf dabei nicht mehr als eine Stunde vom eingegebenen, frühestmöglichen Startzeitpunkt entfernt sein. Gibt es in dieser Zeit keine Verbindung, so wird die Verbindung mit der frühestmöglichen Ankunftszeit gewählt.

Verkehrsbetrieb. Ein Verkehrsbetrieb ist eine Firma, die Personentransporte organisiert und verkauft.

Zeitpunkt. Eine Uhrzeit unter Angabe von Minuten und Stunden an einem beliebigen Tag zwischen 00.00 Uhr und 23.59 Uhr. Alle Tage werden gleich behandelt, eine Unterscheidung zwischen Zeitpunkten an verschiedenen Tagen wird im Programm nicht vorgenommen.

Zielhaltestelle. Die Zielhaltestelle ist eine spezielle Haltestelle, an der der Fahrgast seine Fahrt beenden möchte.