Architekturbezogene Qualitätsmerkmale für die Softwarewartung

Universität Bielefeld
Fakultät für Wirtschaftswissenschaften
Bereich Computergestützte Methoden

Dissertation zur Erlangung des Grades eines Doktors der Wirtschaftswissenschaften (Dr. rer. pol.) der Fakultät für Wirtschaftswissenschaften der Universität Bielefeld

Architekturbezogene Qualitätsmerkmale für die Softwarewartung

Entwurf eines Quellcode basierten Qualitätsmodells

Meik Teßmer

2012
überarbeitete Ausgabe

Anschrift des Verfassers:

Dipl.-Inform. Meik Teßmer
Universität Bielefeld
Postfach 10 01 31
D-33501 Bielefeld
E-Mail: mtessmer@wiwi.uni-bielefeld.de
Website: http://www.wiwi.uni-bielefeld.de/com

Gedruckt auf alterungsbeständigem Papier nach ISO 9706

Gesetzt mit Vim auf einen GNU/Linux-System mit Hilfe von XeTeX, KOMA-Script und Linux Libertine- sowie Biolinum-Schriften

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

AC	Afferent Coupling	105
AHF	Attribute Hiding Factor	105
AIF	Attribute Inheritance Factor	104
API	Application Programming Interface	56
BIS	Bielefelder Informationssystem	5
CBO	Coupling Between Objects	105
CCP	Common Closure Principle	93
CF	Coupling Factor	105
COCOMO	Constructive Cost Model	19
CORBA	Common Object Request Broker Architecture	57
CPU	Central Processing Unit	47
CRP	Common Reuse Principle	93
CSV	Comma-separated values	117
CYCLO	Cyclomatic Complexity	118
DBMS	Database Management System	47
DLL	Dynamic-link Library	57
DNS	Domain Name System	91
DSL	Domain Specific Language	127
EC	Efferent Coupling	105
FCM	Factor-Criteria-Metrics	63
FIFO	First In First Out	41
FOUT	Fan Out	118
HIT	Hierarchy of Inheritance	118
ICH	Information based Cohesion	106
IP	Internet Protocol	40
JDBC	Java Database Connectivity	57
JSP	JavaServer Pages	90

LCC	Loose Class Cohesion	106
LCOM	Lack of Cohesion	104
LOC	Lines Of Code	20
MFC	Microsoft Foundation Classes	56, 57
MHF	Method Hiding Factor	105
MIF	Method Inheritance Factor	104
MOOD	Metrics for Object Oriented Design	4
MPC	Message Passing Coupling	106
NDD	Numbers of Direct Descendants	118
NIM	Number of Instance Methods	105
NIV	Number of Instance Variables	105
NMO	Number of Methods Overridden	105
NOA	Number Of Attributes	105
NOC	Number Of Children oder Number of Classes	105
NOM	Number Of Methods	105
NOP	Number of Packages	118
PF	Polymorphism Factor	105
PIM	Public Instance Methods	105
QUASAR	Qualitäts-Softwarearchitektur	55
REP	Reuse/Relaese Equivalence Principle	93
REST	Representational State Transfer	114
RFC	Response For a Class	104
SOA	Service-orientierte Architektur	40
SQL	Structured Query Language	73
STL	Standard Template Library	56
TCC	Tight Class Cohesion	106
TCP	Transmission Control Protocol	40
TIM	Tivoli Identity Manager	114
TLOC	Total Lines Of Code	119
UML	Unified Modeling Language	22
UP	Unified Process	47
WMC	Weighted Method/Class	103

1 Einleitung

Diese Arbeit untersucht die Möglichkeiten, auf der Basis von Quellcode-Analysen Qualitätsaussagen zur Wartbarkeit zu machen. Ziel ist die Unterstützung des Wartungsprozesses, der von großer ökonomischer Bedeutung ist. Hierzu werden zunächst die Probleme beschrieben, die sich durch die Wartung als Teil der Entwicklung von Softwaresystemen ergeben, und anschließend Maßnahmen diskutiert, die aus Sicht der Entwickler zur Erhaltung der Wartbarkeit beitragen können.

1.1 Wartung von Softwaresystemen als Bestandteil des Entwicklungsprozesses

Die Entwicklung von Softwaresystemen endet nicht mit ihrer Auslieferung an den Kunden. Nach kurzer Zeit ergeben sich oft neue Anforderungen und bestehende Funktionen sollen verändert oder Fehler behoben werden. Boehm bezifferte schon 1979 den Kostenanteil solcher *Wartungsarbeiten* an den Gesamtkosten eines Systems auf ca. 70% [Boe79]. Auch die Entwicklung neuer Programmierparadigmen und -sprachen haben nicht zu einer Verringerung dieses Anteils geführt, im Gegenteil: Edelstein führt an, dass die Wartungskosten jährlich um 10% steigen [Ede93]. Sommerville verweist auf eine Untersuchung von Erlikh [Erl00], in der sogar ein Kostenanteil von 90% ermittelt wurde [Som07a].

Mittlerweile hat sich die Erkenntnis durchgesetzt, dass Wartung und Evolution Teil des Entwicklungsprozesses von Softwaresystemen sind (s. bspw. [McC96, S. 575f]). Auf Grund sich ständig ändernder Anforderungen sind sie ein nahezu „natürlicher“ und unvermeidbarer Vorgang, wie Lehman in seinen *Laws of Software Evolution* deutlich macht: Ohne eine kontinuierliche Anpassung verringert sich im Laufe der Zeit der Nutzen eines Systems für den Anwender [BL76; Leh+97]. Dies gefährdet insbesondere langlebige Informationssysteme, wie sie bspw. in Banken und Versicherungen zum Einsatz kommen. Unternehmen sind abhängig vom Produktionsfaktor Information und seiner reibungslosen Verarbeitung [Som07a; Spi06]. Maßnahmen wie die Fehlerbereinigung und die Umsetzung neuer Funktionen (=*Wartung*) sowie die Portierung auf neue Technologie-Plattformen (=*Evolution*) stellen somit eine Investition für die unternehmerische Zukunft dar.

Die Auswirkungen der Wartung auf den Quellcode und dessen Strukturen sind jedoch gravierend. Laut Müller et al. vergrößert sich der Code-Umfang durch Erweiterungen, Updates usw. jährlich um ca. 10%, die Wartung allein führt alle sieben Jahre zu einer Verdoppelung [MWT94]. Dabei erschwert nicht nur der Umfang des Codes das Verständnis und damit seine Bearbeitung. Nach Lehmans Untersuchungen werden Systeme mit fortschreitender Wartung zunehmend komplexer, während ihre Code-Qualität gleichzeitig abnimmt [Leh+97]. Eick et al. fanden bei der Analyse eines großen Systems (die gesamte Systemhistorie umfasste 100 Mio.

Zeilen Quellcode und nochmals 100 Mio. Zeilen in Header-Dateien und Makefiles, aufgeteilt auf 50 Subsysteme und 5000 Module) ebenfalls Indikatoren für einen zunehmenden *Verfall* der Code-Strukturen (*Code Decay*) [Eic+98]. Wie Lehman bezeichnen auch Hunt und Thomas einen solchen Qualitätsverlust in Anlehnung an die Physik als *Software-Entropie* [HT03]. Durch die Wartung wird so schließlich ein Stadium erreicht, in dem es nahezu unmöglich ist, Code zu korrigieren oder hinzuzufügen, ohne zugleich neue Fehler zu produzieren [BL76].

In einer solchen Situation liegt es nahe, dass eine Neuentwicklung der betroffenen Systeme in Betracht gezogen wird. Die Altsysteme beinhalten substantielles Wissen über die Organisation selbst, ihre Geschäftsprozesse und -strategien, die sich über Jahre hinweg entwickelt haben. Diese „Verzahnung“ mit dem Unternehmen spiegelt sich auch im Code wider, insbesondere im Design [BCK98; Con68]. Sie wird allerdings in vielen Fällen nicht explizit dokumentiert und ist als „tacit knowledge“ nur implizit vorhanden [Mas10]. Im Rahmen der Anforderungsanalyse für die Neuentwicklung kann sie als Informationsquelle daher nur bedingt herangezogen werden. Sommerville weist zudem auf eine Untersuchung von Ulrich (s. [Ulr90]) hin, in der die Kosten eines Reengineering (*Reengineering*: Aufbereitung eines Altsystems zwecks Wartbarkeit) als signifikant *geringer* eingeschätzt werden als die Kosten für eine Neuentwicklung gleichen „Reifegrads“ [Som07a]. Vor diesem Hintergrund ist eine qualitätssichernde und auf Evolution ausgerichtete Wartung — nicht nur von Legacy-Systemen — mindestens genauso wichtig wie die Konstruktion neuer Software, wenn nicht sogar wichtiger [MWT94; SHT05].

Bei der Betrachtung von Unternehmen aus der Fortune 100-Liste ließ sich bereits 1994 feststellen, dass diese durchschnittlich 35 Millionen Zeilen Quellcode pflegen [MWT94]. Allein der Umfang dieser Systeme führt dazu, dass 60% der Wartungszeit mit der Suche nach denjenigen Zeilen Quellcode zugebracht wird, die für die Umsetzung einer neuen Anforderung geändert werden müssen [Ede93]. Die Systemdokumentation ist dabei nur bedingt hilfreich, da die Konsistenz von Dokumentation und Code nicht zwangsläufig gegeben ist [Som07a; DP09]. Auch wenn der Pflege der Systemdokumentation viel Aufmerksamkeit gewidmet wird, gilt: Der Quellcode ist die *einzige zuverlässige* Informationsquelle, wenn es um Fragen der Wartung und Evolution eines Systems geht [Mar09; DP09]. Er bildet die Grundlage für die auszuliefernde Anwendung und vereint alle Entscheidungen, die Domänenspezialisten, Designer, Programmierer und Tester in seine Entwicklung haben einfließen lassen. Dieses Wissen muss für erfolgreiche *Software Maintenance* so weit wie möglich wieder verfügbar und der Qualitätssicherung zugänglich gemacht werden. Besonderes Augenmerk sollte dabei auf die Güte der Informationen gerichtet werden, denn statt immer mehr Informationen anzuhäufen, werden die *richtigen* — für die Wartung hilfreichen — Informationen benötigt. Welche Informationen das sind, hängt von der Art der Maßnahmen ab, die im Rahmen der Software Maintenance durchgeführt werden sollen.

1.2 Maßnahmen zur Erhaltung der Wartbarkeit

Software Maintenance wird oft übersetzt mit „Wartung“. Dieser Begriff assoziiert einen Defekt, der behoben werden muss. Auch das gehört zu Software Maintenance, jedoch ist darunter auch „Erhaltung“ zu verstehen: Anpassung, Veränderung, Optimierung und natürlich Reparatur. Eine bessere Übersetzung wäre demnach „Software-System-Erhaltung“ [SHT05]. Die IEEE

kategorisiert die Systemerhaltung in *erweiternde* und *korrigierende* Maßnahmen [06a]. Zur ersten Kategorie gehören die Verbesserung der Performance, Wartbarkeit oder die Modifikation anderer Eigenschaften (*Perfective Maintenance*) sowie die Anpassung eines Produkts an eine sich ändernde Systemumgebung — auch Portierung genannt — (*Adaptive Maintenance*), zur zweiten die Beseitigung von Fehlern (*Corrective Maintenance*) und die präventive Suche und Korrektur latenter Fehler, bevor sie effektiv werden (*Preventive Maintenance*). Jede dieser Maßnahmen erfordert ein mehr oder weniger umfangreiches Wissen über den System-Quellcode und seine Strukturen.

Dieses Wissen kann mit Methoden des Programmverstehens (*Program Comprehension*) und durch *Reverse Engineering* erlangt werden [Som07b; RW10]. Dabei wird zunächst der Quellcode analysiert mit dem Ziel, Komponenten und ihre Beziehungen untereinander zu identifizieren und Darstellungen der Software zu erzeugen, die eine andere Form oder ein höheres Abstraktionsniveau haben. Dazu gehören verschiedene *Sichten auf die Systemarchitektur* wie z. B. Aufruf-Graphen, Klassendiagramme und Kontrollfluss-Diagramme (s. dazu bspw. [Kru95]), aber auch Visualisierungen von Impact-, Trend- und Metrik-Analysen (s. bspw. [Ber10]). Neben einer Reduzierung der Komplexität, die das Verständnis erleichtern soll, gewinnt der Entwickler auch Hinweise auf Qualitätsmängel, die für präventive Wartungsmaßnahmen hilfreich sein können. Refactoring-Werkzeuge erleichtern das Reengineering, indem sie passend zu sog. *Code Smells* automatisierte Code-Transformation anbieten (s. [Fow99]).

Koschke beklagt allerdings, dass es trotz einiger Arbeiten zum Prozess der Architekturrekonstruktion bis dato keinen umfassenden Katalog von Techniken gibt, der detailliert beschreibt, wann und wo welche Maßnahmen zur Strukturverbesserung zu ergreifen sind [Kos05]. Bei der Vermessung von Software mit Hilfe von Metriken zeigen sich ähnliche Schwierigkeiten: „Absolute Zahlen für gute oder schlechte Größen von Klassen, Packages, Subsystemen oder Schichten lassen sich nicht angeben.“ [RL04] Der Einsatz solcher Metriken wird zusätzlich erschwert durch die nicht standardisierte Interpretation der Messergebnisse [And+04]. Gründe dafür sind u. a. die unklare Definition von Begriffen wie „Wartbarkeit“ und fehlende Maßstäbe, bspw. für die maximale Vererbungstiefe von Klassenhierarchien. McCabe stellt in [McC76] einen Graph-orientierten Ansatz zur Komplexitätsbestimmung von Programmen vor, der an einigen Fortran-Programmen evaluiert wurde. Diese Maße sind zwar gut für Projekte geeignet, deren Programmiersprachen dem Paradigma der *Strukturierten Programmierung* folgen (s. [Dij69; Dij70a; Dij70b]), sie haben aber nur eingeschränkten Nutzen bei Verwendung von objektorientierten Sprachen. Die bekannteste Metrik-Suite für derartige Sprachen ist die von Chidamber und Kemerer [CK91; CK94]; sie erfasst Struktur- und Volumenmaße. Allerdings gibt es auch hier Zweifel am Nutzen einzelner Metriken. Eine Untersuchung von Aggarwal et al. hat ergeben, dass von 14 untersuchten Metriken nur sechs hinreichend gute eigenständige Aussagen lieferten, während die anderen Metriken entweder Teilmengen dieser sechs bildeten oder dieselbe Information auf andere Weise darstellten [Agg+06].

Ein weiteres Problem ist die Unterstützung der Entwickler sowohl durch die Struktur der Programmiersprachen als auch durch die Programmierwerkzeuge während der Entwicklung selbst. Objektorientierte Sprachen wie C++ und Java bieten keine direkte sprachliche Unterstützung zur Umsetzung von Architekturen, wodurch die Identifizierung von Komponenten nur *indirekt*, bspw. durch Strukturanalysen, erfolgen kann. Hier fehlt neben einer Präzisierung der Begriffsdefinition „Software-Architektur“ eine Auflistung von relevanten *Merkmalen*, *Sich-*

ten und *Perspektiven*.

In den letzten Jahren wurden mehrere Ansätze entwickelt, die Qualitätsaussagen über Quellcode ermöglichen. Die von Gamma et al. katalogisierten *Entwurfsmuster* bieten dem Entwickler zu häufig wiederkehrenden Problemstellungen bewährte Code-Strukturen als Lösung an und dienen als „Best Practices"-Leitfaden und Maßstab [Gam+96]. Dabei werden auch Aspekte wie Wiederverwendbarkeit und Erweiterbarkeit betrachtet. Weiterhin hat Fowler mit seinen *Code Smells* und den entsprechenden Refactoring-Vorgaben einen ersten Schritt in Richtung Maßnahmenkatalog getan [Fow99]. Code Smells sind Indikatoren für Bereiche im Quellcode, die durch ein Refactoring verbessert werden können. Roock und Lippert heben das Refactoring auf die Architekturebene und geben Hinweise auf *Architektur-Smells*, die zu „großen" Refactorings von Komponenten führen [RL04]. Kerievsky kombiniert Entwurfsmustern und Refactoring zu „Refactoring to Patterns" und bietet so Methoden an, die die Evolution von Systemen helfen können [Ker06]. Trotz der schon angeführten Probleme unterstützen die Metrik-Suiten von McCabe, Halstead (s. [Hal77]), Chidamber und Kemerer, das MOOD-Set von Harrison et al. (s. [HCN98]) sowie Fokussierungen und Erweiterungen verschiedene qualitätsorientierte Betrachtungen des Codes und erlauben eine Lokalisierung wartungsbedürftiger Code-Bereiche und die Abschätzung des Wartungsumfangs (s. dazu u. a. [BBM96; BDW99; BMB99; Cos+05; FN01; Gos+03; HS01; LC01; WYF03]).

1.3 Zielsetzung

Im Rahmen dieser Arbeit sollen folgende Forschungsfragen aus diesem Problembereich angegangen werden:

1. *Welche Qualitätsmerkmale lassen sich aus Quellcode ermitteln?* Die Ermittlung von Qualitätsmerkmalen ist nur ein erster Schritt, um zu einem qualitäts- und evolutionsbewussten Wartungsprozess zu gelangen. Feingranulare Merkmale wie bspw. Lines of Code, Metriken nach McCabe und Halstead bilden eine Teilmenge dieser Merkmale, wichtiger sind jedoch Merkmale auf der Ebene der Software-Architektur. Zunächst muss jedoch der Qualitätsbegriff selbst näher untersucht und eine Definition abgeleitet werden, um die etablierten Merkmale hinsichtlich ihres Nutzens für diese Arbeit einer Bewertung zuführen zu können. Gleiches gilt für den Architekturbegriff und die Architekturqualität.

2. *Wie relevant sind die Merkmale für die Wartbarkeit und Weiterentwicklung eines Systems?* Die Relevanz muss insbesondere aus Sicht der Entwickler bewertet werden, denn sie sind es, die mit dem Quellcode hantieren müssen. Besonderes Augenmerk liegt dabei auf der Systemarchitektur, denn ihre Tragfähigkeit ist entscheidend für die Langlebigkeit eines Software-Produkts [BCK98].

3. *Wie lässt sich aus den gewonnenen Informationen ein Qualitätsmodell entwickeln, das sich in den Kontext eines Softwareentwicklungsprozesses eingliedert?* Ein solches Qualitätsmodell muss sich ebenfalls wie das zu bewertende Softwaresystem entwickeln können, um seine Aussagekraft zu behalten. Somit muss zunächst untersucht werden, welche etablierten Qualitätsmodelle es gibt und wie sie hinsichtlich der Definition des Qualitätsbegriffs in dieser Arbeit zu bewerten sind.

Zur Beantwortung dieser Fragen wird ein Qualitätsmodell zur Wartung entworfen, das die Wartbarkeit von Softwaresystemen aus Sicht der Entwickler bewerten kann. Eine Analyse des BIS[1] der Universität Bielefeld mit Hilfe dieses Qualitätsmodells soll zeigen, ob es für die Qualitätsbewertung dieses Systems geeignet ist.

1.4 Einordnung und Aufbau der Arbeit

Die Arbeit ist dem Forschungsbereich der Wirtschaftswissenschaften, genauer, der Wirtschaftsinformatik, zuzuordnen und verfolgt ein *Gestaltungsziel*, das sich durch die Entwicklung dieses Artefakts *für die Problemlösung* manifestiert. Weiterhin sollen Erkenntnisse über Qualitätsmerkmale, deren Verwendung und Effektivität zu erlangen. Somit wird auch ein *Erkenntnisziel* verfolgt. Wissenschaftstheoretisch gliedert sich die Arbeit damit in die *gestaltungsorientierte Wirtschaftsinformatik* ein (s. dazu auch [Öst+10]). Konstituierendes Merkmal dieses Forschungsparadigmas ist die Hinwendung zur Konstruktion von Artefakten zur Problemlösung, ganz im Gegensatz zur behavioristischen Forschung, die vornehmlich nach Erklärungen von in der Realwelt beobachtbaren Phänomenen sucht [Hev+04; Zel07]. Unter dem Begriff *Artefakt* werden dabei Modelle, Methoden, Konstrukte und Umsetzungen subsumiert. Will man dem Wunsch nach einer Generalisierbarkeit von Forschungsergebnissen Rechnung tragen, können auch Theorien als Artefakte aufgefasst werden [BKN08].

Die von Hevner et al. in [Hev+04] vorgeschlagenen Richtlinien zur Bewertung von Forschungsarbeiten werden unter Einbeziehung der dazu von Zelewski in [Zel07] angeführten Kritikpunkte folgendermaßen für diese Arbeit angewendet:

1. Das zu lösende organisatorische Problem wird in Abschnitt 1.3 umrissen und in Kapitel 2 detailliert beschrieben. Entwickelt werden zwei Artefakte: (1) Ein Satz von Qualitätsmerkmalen auf der Ebene der Software-Architektur, mit deren Hilfe (2) ein Qualitätsmodell erstellt wird.

2. Die Grundproblematik wird in den Abschnitten 1.1 und 1.2 beleuchtet und in Kapitel 2 vor dem Hintergrund der Forschungsfragen ausgearbeitet.

3. Die Effektivität und Qualität der Artefakte wird anhand ihrer Umsetzung in Kapitel 5 geprüft. Da allerdings eine alternative Sichtweise auf die Qualität eines Softwaresystems eingenommen werden soll, ist die Menge der Evaluierungsmethoden neben der Umsetzung der Erweiterung auf die Anwendung derselben im Rahmen einer Fallstudie (Kapitel 6) mit anschließender Diskussion beschränkt.

4. Der Forschungsbeitrag ist in Abschnitt 1.3 skizziert und wird in den Kapiteln 2 bis 6 entwickelt.

5. Die Stringenz der Arbeit wird gewahrt durch die Verwendung etablierter Theorien zur Softwarequalität und den Einsatz von Technik-Standards bei der Umsetzung.

[1] Das BIS ist ein Web-basiertes Informationssystem für die Universität Bielefeld. Es besteht aus dem elektronischen kommentierten Vorlesungsverzeichnis, der Prüfungsverwaltung und dem Personen- und Einrichtungsverzeichnis. Sämtliche BIS-Anwendungen werden hausintern entwickelt und betrieben.

6. Durch das Ziel, eine alternative Sichtweise anzubieten, kann diese Arbeit nur eine von mehreren möglichen Lösungen sein, die versuchen, die verschiedenen Probleme im Umfeld der Software-Wartung zu lösen.

7. Die Arbeit richtet sich ausschließlich an ein Fachpublikum, das mit den Theorien und Methoden der Softwareanalyse und Qualitätssicherung vertraut ist.

Die Arbeit ist in vier Teile untergliedert. Der erste Teil (Kapitel 2, 3 und 4) beleuchtet die theoretischen Grundlagen des Qualitätsbegriffs und untersucht ihre Anwendbarkeit auf den Quellcode eines Softwaresystems. Dazu werden existierende Qualitätsstandards herangezogen und auf ihren Nutzen für die Analyse eines Softwaresystems geprüft. Fokussiert wird auf die Bereiche Software-Architektur und Software-Vermessung mittels Metriken, da sie die Basis der Informationsgewinnung aus Quellcode darstellen. Darüber hinaus muss geprüft werden, ob es Abhängigkeiten dieser Analyse- und Bewertungsmethoden von bestimmten Programmierparadigmen oder -sprachen gibt und welche Auswirkungen dies auf die Bewertung eines Softwaresystems haben kann.

Die Untersuchungsergebnisse führen im zweiten Teil (Kapitel 5) zur Entwicklung eines Qualitätsmodells und eines zugehörigen Softwaremodells, die beide anschließend in Form konkreter Methoden der Informationsgewinnung aus Quellcode dienen. Im dritten Teil (Kapitel 6) wird ein konkretes Softwaresystem der Universität Bielefeld, das System BIS, mit dem entwickelten System exemplarisch analysiert.

Der vierte Teil (Kapitel 7) fasst die Forschungsergebnisse zusammen und gibt einen Ausblick auf noch offene Forschungsfragen und weitere Untersuchungen.

2 Qualitäten von Softwaresystemen

> Any fool can write code that a computer can understand. Good programmers write code that humans can understand.
>
> *(Martin Fowler in „Refactoring“)*

Im Zuge dieses Kapitels werden zunächst die Ziele der Entwickler und ihre Anforderungen an Software-Qualität charakterisiert. Es folgt ein Überblick zum Standard ISO 9126. Nach einer Betrachtung des vom Standard definierten Qualitätsmodells wird auf die Konkretisierung der Eigenschaften und Merkmale zu Kenngrößen eingegangen. Eine Auswertung im Hinblick auf die Eignung der Standards als Ausgangpunkt für die Entwicklung eines Qualitätsmodells beendet die Untersuchung.

2.1 Ziele der Entwickler

An der Entwicklung eines Softwaresystems sind viele sog. „Stakeholder“ beteiligt, die jeweils bestimmte Ziele verfolgen und eigene Erwartungen an die Qualitäten des Systems haben [BCK98, S. 6]. So sind bspw. für den Benutzer „äußere“ Qualitäten wie die gewünschten Funktionen, die Fehlerfreiheit, Arbeitsgeschwindigkeit und Effizienz (allg. die „Usability“) wichtig, für das Management der entwickelnden Organisation die Entwicklungsdauer und das Budget.

Die Aufgabe der Entwickler besteht in der Umsetzung der funktionalen und nicht-funktionalen Anforderungen. Da sich in den letzten 10 Jahren die Erkenntnis durchgesetzt hat, dass Änderungen ein normaler Bestandteil des Entwicklungsprozesses sind und jederzeit auftreten können, haben sich nicht nur die Entwicklungsmodelle verändert (s. zu diesen „agile Methoden“ bspw. [Mar02]), sondern auch die Anforderungen an den Quellcode und dessen Strukturierung. Martin geht sogar soweit zu behaupten, dass der Entwurf eines Systems falsch sei, wenn er diesem Änderungsdruck nicht standhalte [Mar02, S. 90]. Das Augenmerk der Entwickler liegt also nicht mehr nur auf der reinen Umsetzung der Anforderungen, sondern auch auf den „inneren“ Qualitäten wie der Systemstrukturierung (Architektur und Komponenten, Umfänge und Komplexität einzelner Bestandteile, Entwurfsmuster) und z. T. subjektiven Einschätzungen von Lesbarkeit und Verständlichkeit, umschrieben mit Begriffen wie „Schönheit“ oder „Eleganz“ [Mar09, S. 32, 39].

Publikationen wie [Gam+96], [Fow99], [RL04], [Ker05] und [Mar09] geben z. T. sehr detaillierte Hinweise auf Merkmale, die die Wartbarkeit beeinflussen und wie die Struktur eines Systems auf der Architekturebene beschaffen sein sollte. Sie betten diese Hinweise jedoch nicht in ein übergeordnetes Qualitätsmodell ein, das die inneren Qualitäten näher ausführt und ihnen einzelne Merkmale zuordnet. Es fehlt zudem der Bezug zu höheren Abstraktionsebenen,

wie er bspw. von verschiedenen *Sichten* hergestellt wird. Mit Hilfe solcher Sichten können die *Architekturen* beschrieben werden, die ein System besitzt (s. dazu u. a. [Kru95]). Die Strukturierung des Systems auf dieser Abstraktionsebene hat maßgeblichen Einfluss auf die späteren Systemeigenschaften [BCK98, S. 78]; Denert bezeichnet die Software-Architektur im Vorwort zu [Sie04] gar als „die Königsdisziplin des Software-Engineering“.

Für eine Bewertung der Wartbarkeit von Quellcode aus Sicht der Entwickler muss der Qualitätsbegriff mit Hilfe eines *Qualitätsmodells* definiert werden. Qualitätsmodelle versuchen, das allgemeine, unscharfe Verständnis von „Qualität“ durch eine Taxonomie von Einzelfaktoren mess- und damit bewertbar zu machen. Für Software-Qualitätsmodelle heißt das, dass Qualitätsunterbegriffe abgeleitet und diese weiter zu Indikatoren verfeinert werden, welche dann einem Messinstrumentarium zugänglich sind. Ziel ist, durch Messergebnisse Rückschlüsse auf die Gesamtqualität ziehen zu können (s. dazu bspw. [Wal01, S. 47f]). Zusätzlich erreicht man eine *Vereinheitlichung der Vorstellung von Qualität* und macht sie dadurch *kommunizierbar*. Qualität wird *konkretisiert* und damit mess- und bewertbar gemacht [Wal01, S. 46f]. Kommunizierbarkeit ist besonders für große und langlebige Produkte ein wichtiges Kriterium, da diese i. d. R. von einem größeren Entwickler-Team bearbeitet werden.

Der im folgenden Abschnitt beschriebene Standard ISO 9126 beschreibt ein solches Qualitätsmodell. Inwieweit er als Ausgangspunkt für die Entwicklung einer solchen Begriffsdefinition dienen kann, wird anschließend untersucht.

2.2 Qualitätsstandard ISO 9126

Die unterschiedlichen Sichtweisen der Stakeholder erschweren die Entwicklung einer umfassenden Definition eines Qualitätsbegriffs. Ein Indiz dafür zeigt sich in den vielfachen Bemühungen verschiedener Autoren und Institutionen wie DIN, IEEE und ISO sowie den daraus entstandenen Standards, die sich zudem gegenseitig referenzieren. Abbildung 2.1 versucht, diesen Zustand zu illustrieren.

2.2.1 ISO 9126 im Kontext anderer Standards

Die *International Organization for Standardization* (ISO) hat eine Reihe von Standards veröffentlicht, die auf die Bedürfnisse der für die Auswahl und Entwicklung verantwortlichen Interessengruppen zugeschnitten sind. Auch wenn der Benutzer in den Entwicklungsprozess mit einbezogen wird, so sind primär das Management und die Entwickler die Hauptakteure, wenn es um *Maßnahmen* zur Qualitätssicherung geht. Dem Management obliegt die Planung des mitunter langfristigen Einsatzes von Softwaresystemen sowie die Koordination der verfügbaren Ressourcen. Dieser *Lebenszyklus*wird in den Standards ISO 15288 (*Life Cycle Management: System Life Cycle Processes*), ISO 14598 (*Information Technology: Software Product Evaluation*) und IEEE/EIA 12207 (*Software life cycle processes*) aufgeschlüsselt und mit weiteren Arbeitsabläufen innerhalb einer Organisation verzahnt [ISO00; 99a; 98]. Sie beschreiben neben den Prozessen, die den gesamten Lebenszyklus eines Softwaresystems strukturieren, auch Verfahren für die Auswahl von Produkten.

Auch wenn Standardprodukte ausgewählt werden und keine Neuentwicklung stattfinden soll, muss in vielen Fällen eine Anpassung, zumindest jedoch eine Integration in ein vorhande-

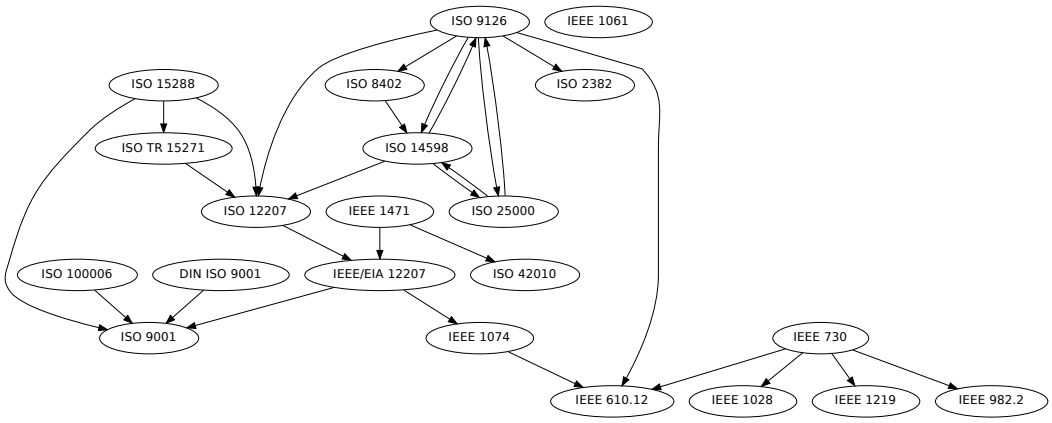

Abbildung 2.1: Standards zur Qualität von Softwaresystemen aus dem Bereich IEEE und ISO (eigene Darstellung; ohne Anspruch auf Vollständigkeit).

nes System vorgenommen werden. Diese Wartungsarbeiten werden als Teil eines allgemeinen Entwicklungsprozesses angesehen, dessen Qualitätssicherung in der Standard-Serie ISO 25000 (*Software engineering - Software product Quality Requirements and Evaluation (SQuaRE)*) umfassend erläutert werden soll (s. [05a]). Die Veröffentlichung der ersten Teilstandards begann 2005. ISO 25000 soll die beiden verwandten Standards ISO/IEC 9126 (*Software product quality*) (s. [01; 03a; 03b; 04a]) und ISO/IEC 14598 kombinieren und nach Veröffentlichung sämtlicher Teildokumente ersetzen. Dieser neue Standard verfolgt das Ziel, Inkonsistenzen zwischen den beiden Vorgängerstandards ISO 9126 und ISO 14598 zu beseitigen und einen koordinierten Zugang zur Software-Produktvermessung und -bewertung bieten.

2.2.2 Aufbau des Standards

ISO 9126 besteht aus vier Teilstandards. Teil eins beschreibt das dem Standard zugrunde liegende Qualitätsmodell, dass sich in zwei Teilmodelle untergliedert (Produktqualität=interne/externe Qualität und Gebrauchsqualität). Teil zwei spezifiziert Kenngrößen für die Bewertung der externen, Teil drei für die Bewertung der internen Qualität. Teil 4 beschreibt schließlich Kenngrößen für die Gebrauchsqualität [01; 03a; 03b; 04a].

In ISO 9126 wird wie bei Wallmüller auf einen Zusammenhang zwischen der Qualität des Entwicklungsprozesses und der Produktqualität hingewiesen: „Die Qualität eines Software-Produkts wird bestimmt durch den Prozess, in dem es entwickelt wird, und die Merkmale, die es besitzt.“ [Wal01, S. 11]. Das dem Qualitätsmodell zugrunde liegende Verständnis der Qualitätsentwicklung gleicht einer Kette, deren Glieder *Prozessqualität–interne Qualität–externe Qualität–Gebrauchsqualität* durch Einfluss- und Abhängigkeitsbeziehungen miteinander verbunden sind. Die Prozessqualität trägt zur Produktqualität bei, die wiederum die Gebrauchsqualität beeinflusst. Umgekehrt führen die Bedürfnisse und Erwartungen der Benutzer, die das Produkt in bestimmten Umgebungen einsetzen und so eine Gebrauchsqualität erwarten, zur Spezifikation

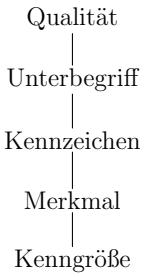

Abbildung 2.2: Zerlegungssystematik des Qualitätsbegriffs in ISO 9126-1.

der Anforderungen an die externe Qualität, die wiederum die Spezifikation der Anforderungen an die interne Qualität beeinflusst.

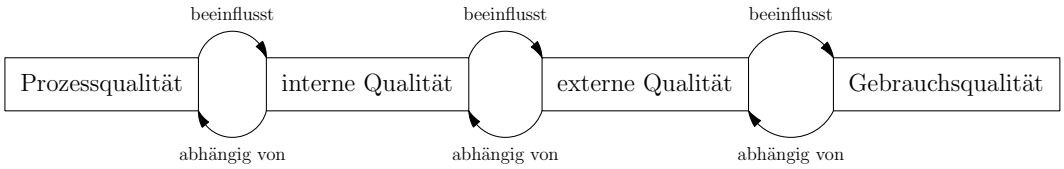

Abbildung 2.3: Verständnis der Entwicklung von Qualität (nach [01]).

Interne Qualität unterscheidet sich von externer Qualität darin, dass sie durch Untersuchungen des nicht ausführbaren Produkts (vornehmlich Quellcode, aber auch Dokumentation inkl. Modelle) mit Hilfe interner Maße ermittelt wird. Externe Qualität zeigt sich hingegen nur beim konkreten Einsatz des Produkts und wird durch externe Maße erhoben. Die Gebrauchsqualität wiederum ist abhängig von den jeweiligen Bedürfnissen, Erwartungen und Erfahrungen des Benutzers und dem Kontext, in dem er das System nutzt. Dies führt für dasselbe Produkt zu verschiedenen Qualitätsausprägungen. Um überhaupt eine Gebrauchsqualität zu erreichen, muss zunächst eine ausreichend große Effektivität vorliegen, ohne die das Produkt für den Benutzer wertlos wäre. Eine gewisse Effektivität vorausgesetzt gibt die Produktivität an, wie effizient er seinen Aufgaben mit Hilfe des Produkts nachgehen kann.

Das Verhältnis der Gebrauchsqualität zur internen/externen Qualität wird durch den Benutzertyp beeinflusst. Für den Benutzer ist primär die Funktionalität, die Zuverlässigkeit und Effizienz eines Produkts ausschlaggebend, für einen mit der Wartung beauftragten Entwickler resultiert sie eher aus der Wartbarkeit des Produkts [01, S.15]. Gebrauchsqualität wird demnach durch den Zweck und Kontext bestimmt und definiert dadurch eine bestimmte Sichtweise auf die externe Qualität.

2.2.3 Innere und äußere Produktqualität

Das in ISO 9126-1 definierte Teilmodell für *interne/externe Qualität* gründet die Qualität eines Software-Produkts auf sechs *Eigenschaftskategorien*, die sich weiter in einzelne *Untereigenschaften* und die wiederum in *Merkmale* untergliedern. Merkmale können durch *Kenngrößen*

(=metrics) vermessen werden, welche als Indikatoren in die Bewertung eingehen. Es richtet sich vornehmlich an Entwickler.

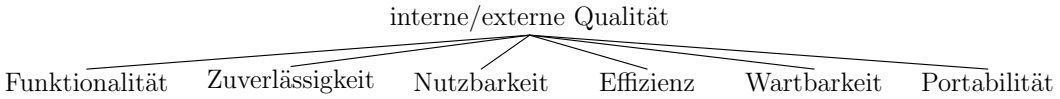

Abbildung 2.4: Taxonomie des Teilmodells für interne/externe Qualität (nach [01]).

Das Teilmodell definiert und detailliert folgende Eigenschaftskategorien:

- Funktionalität
- Zuverlässigkeit
- Nutzbarkeit
- Effizienz
- Wartbarkeit
- Portabilität

Funktionalität (functionality)

Diese Eigenschaft beschreibt die Fähigkeiten des Produkts, durch das Bereitstellen von Funktionen die Bedürfnisse eines Benutzers unter bestimmten Einsatzbedingungen zu befriedigen. Dabei kommen sowohl explizit festgehaltene Bedürfnisse (bspw. in einem Pflichtenheft) als auch implizite (Erwartungen an die Arbeitsgeschwindigkeit) in Betracht. Qualität bemisst sich dann am Grad der Erfüllung dieser Bedürfnisse.

Funktionalität wird durch fünf Untereigenschaften definiert:

- Die *Eignung* (suitability) bemisst, wie gut die durch das Produkt bereitgestellte Menge von Funktionen die Tätigkeiten des Benutzers unterstützt. Die Eignung wird zudem durch die Untereigenschaft *Betriebsfähigkeit* der Eigenschaft *Nutzbarkeit* beeinflusst.
- Die *Genauigkeit* (accuracy) des Produkts zeigt sich durch vom ihm gelieferte korrekte und vom Benutzer so erwartete Ergebnisse.
- *Interoperabilität* (interoperability) beschreibt, wie gut das Produkt mit anderen Systemen interagieren kann.
- *Sicherheit* (security) im Sinne des Datenschutzes und der Integrität wird daran gemessen, wie gut ein Produkt den nicht autorisierten Zugriff auf sensible Daten unterbinden kann.
- Konformität (*functionality compliance*) bezeichnet den Grad der vom Produkt eingehaltenen Standards, Konventionen und Gesetze.

Zuverlässigkeit (reliability)

Als Zuverlässigkeit wird die Fähigkeit eines Produkts verstanden, unter bestimmten Bedingungen ein bestimmtes Leistungsniveau beizubehalten. Dies gilt insbesondere für den Fehlerfall, seien es Bedienungs- oder Produktfehler.

Vier Untereigenschaften beschreiben die Zuverlässigkeit:

- Als *Reife* (maturity) wird das Vermögen betrachtet, Ausfälle durch Produktfehler zu vermeiden. Dies geschieht üblicherweise durch Überprüfungen von Zwischenergebnissen innerhalb des Produkts.

- *Fehlertoleranz* (fault tolerance) hingegen beschreibt die Fähigkeit des Systems, trotz Produktfehler und (mutwilliger) Fehlbedienung durch den Benutzer oder den Entwickler stabil zu bleiben.

- Sollte es trotz aller Vorkehrungen zu einem Ausfall des Systems kommen, ist die *Wiederherstellbarkeit* (recoverability) der Leistung und der betroffenen Daten ein wichtiges Merkmal.

- Ein Produkt sollte *Standards und Konventionen zur Zuverlässigkeit* (reliability compliance) einhalten. (Dieses Merkmal wird durch den Standard nicht weiter ausgeführt. Ein Argument könnte sein, dass durch die Orientierung an derartigen Standards die Umsetzung der übrigen drei Merkmale unterstützt und insgesamt eine bessere Akzeptanz erreicht werden soll.)

Nutzbarkeit (usability)

Die Nutzbarkeit eines Produkts ist die wohl augenfälligste Qualitätseigenschaft für den Benutzer. Ein Produkt sollte verständlich, gut zu erlernen und zu verwenden sowie attraktiv sein. Neben dem Endkunden werden hier auch alle von der Verwendung Betroffenen als Benutzer betrachtet, d. h. auch Administratoren und Betreuer. Dabei sollen all diese Arbeitsumgebungen mit in die Betrachtung der Nutzbarkeit eingehen.

Fünf Untereigenschaften sollen die Nutzbarkeit belegen:

- Die *Verständlichkeit* (understandability) eines Produkts erlaubt dem Benutzer einzuschätzen, ob es für seine Zwecke geeignet ist und wie es dazu eingesetzt werden kann. Ausschlaggebend ist neben dem ersten Eindruck die Produktdokumentation.

- Eine gute *Erlernbarkeit* (learnability) befähigt den Benutzer, in kurzer Zeit die Fähigkeiten des Produkts kennen zu lernen und einzusetzen.

- Unter *Betriebsfähigkeit* (operability) versteht man den Grad der möglichen Bedienung Kontrolle des Produkts durch den Benutzer. Die Bedienbarkeit wird auch von den Untereigenschaften Eignung, Änderbarkeit, Anpassungsfähigkeit und Installierbarkeit beeinflusst.

- Die *Attraktivität* (attractiveness) eines Produkts beschreibt den Reiz, den das Produkt auf einen Benutzer ausüben und ihn zum Kauf oder Einsatz animieren kann. Neben der Funktionalität spielt dabei auch die optische Gestaltung eine wesentliche Rolle.
- Ein Produkt sollte *Standards und Konventionen zur Nutzbarkeit* (usability compliance) einhalten. Da auch hier eine nähere Erläuterung fehlt, soll dieses Merkmal als Hinweis auf die etablierten Standards und Best-Practices zum GUI-Design (bspw. [Ras00]) und zu Quasi-Standards einzelner Produktklassen (Office-Produkte, Webbrowser etc.) verstanden werden.

Effizienz (efficiency)

Ein Produkt besitzt eine bestimmte *Effizienz* bei der Verwendung vorhandenen Systemressourcen. Neben Prozessor- und Speicherkapazität betrifft dies auch Verbrauchsmaterialien wie Papier. Drei Untereigenschaften detaillieren dies:

- Das *Zeitverhalten* (time behaviour) beschreibt die Antwortzeiten, Verarbeitungszeiten und Durchsatzraten eines Produkts bei seiner Verwendung.
- Allgemein wird die *Ressourcenverwendung* (resource utilisation) als Beschreibung der Verwendung bestimmter Typen und Mengen von Ressourcen festgehalten.
- Das Produkt sollte sich an etablierten *Effizienzstandards* (efficiency compliance) orientieren. Ob es hier mittlerweile zu allgemein gültigen Standard und Konventionen kommen wird, bleibt allerdings fraglich. Ein übliches Vorgehen bei zu langsam laufenden Produkten oder Systemen ist, einfach leistungsfähigere Hardware einzusetzen.

Wartbarkeit (maintainability)

Die Wartbarkeit eines Produkts beschreibt, wie gut Änderungen, Erweiterungen oder Anpassungen an die Umgebung vorgenommen werden können. Dies betrifft neben dem System-Code selbst auch die erhobenen Anforderungen und funktionalen Spezifikationen. Damit erstrecken sich Betrachtungen zur Wartbarkeit auch auf die zugehörige Dokumentation.

Anhand der folgenden fünf Untereigenschaften soll die Wartbarkeit beurteilt werden:

- *Analysierbarkeit* (analysability) ist eine Eigenschaft, die es erlaubt, Fehlerquellen oder Defizite durch eine Untersuchung des Systems — das kann der Quellcode, die Dokumentation und/oder die zugehörigen Daten sein — zu identifizieren und die zu ändernden Stellen einzugrenzen.
- Die *Änderbarkeit* (changeability) beschreibt dann, wie gut die entsprechenden Änderungen implementiert werden können. Dazu gehören auch Korrekturen und Anpassungen der Dokumente.
- Damit Änderungen keine unerwünschten Seiteneffekte zeigen, sollte das Produkt eine gewisse *Stabilität* (stability) besitzen. Erreicht werden kann dies durch Unit-Tests und Techniken wie defensives Programmieren, Design by Contract und den Einsatz von Exception Handling.

- Um Änderungen auf ihre Korrektheit hin überprüfen zu können, muss das Produkt so beschaffen sein, dass es *getestet* (testability) werden kann. Dies kann bspw. in Form von sog. *White-Box-Tests* oder eher anwendernahen *Black-Box-Tests* erfolgen. Grundlage dazu ist eine angemessene Modularisierung des Produkts.

- Das Produkt sollte *Standards und Konventionen zur Wartung* (maintainability compliance) einhalten.

Portabilität (portability)

Unter Portabilität wird nicht nur die Übertragung auf eine andere Hardware-Plattform, sondern allgemein der Transfer in eine andere Einsatzumgebung verstanden. Das können auch organisatorische und andere Software-Umgebungen sein. Durch fünf Untereigenschaften soll die Portabilität gezeigt werden:

- Unter *Anpassungsfähigkeit* (Adaptability) oder Konfigurierbarkeit versteht man die Fähigkeit eines Produkts, an den Einsatz in verschiedenen Umgebung angepasst werden zu können, ohne das Produkt selbst zu modifizieren. Dazu zählt u. a. auch die Skalierung interner Größen wie Bildschirm-Maße, Transaktionsvolumen und Reportlänge.

- Die *Installierbarkeit* (installability) eines Produkts beschreibt, wie gut es in eine bestehende Umgebung eingefügt werden kann.

- Üblicherweise stehen Systeme nicht allein, sondern arbeiten mit anderen unabhängigen Systemen zusammen und teilen deren Ressourcen. Daher ist die *Koexistenz* (co-existence) mit anderen Systemen eine wichtige Qualitätseigenschaft.

- *Ersetzbarkeit* (replaceability) beschreibt sowohl die Ersetzung des Produkts durch eine neue Version als auch die Ersetzung durch ein kompatibles Produkt. Neben den funktionalen Anforderungen müssen insbesondere auch die Schnittstellen zu anderen Systemen kompatibel sein.

- Das Produkt sollte *Portabilitätsstandards und Konventionen* (portability standards) erfüllen.

2.2.4 Gebrauchsqualität

Das Teilmodell für Gebrauchsqualität betrachtet die Qualität eines Software-Produkts auch Sicht des Benutzers. Es zerlegt den Oberbegriff in die vier Eigenschaftskategorien *Effektivität*, *Produktivität*, *Sicherheit* und *Befriedigung*, die jedoch nicht weiter unterteilt werden.

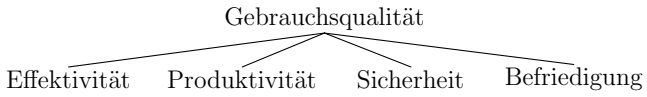

Abbildung 2.5: Taxonomie des Teilmodells für Gebrauchsqualität (nach [01]).

Unter dem Begriff *Gebrauchsqualität* wird allgemein die Fähigkeit eines Produktes verstanden, den Benutzer bei der Erreichung bestimmter Ziele effektiv zu unterstützen. Dabei handelt sich nicht direkt um Eigenschaften des Produkts selbst, vielmehr wird die Art und Weise bewertet, wie sich das Produkt in die Arbeitsumgebung des Benutzers integriert. Eine Qualitätsbewertung erfolgt somit immer im Kontext einer konkreten Arbeitsumgebung und dem Grad der Unterstützung durch das Produkt. Demnach führen mehrere mögliche Einsatzumgebungen zu entsprechend vielen möglichen Bewertungen der Gebrauchsqualität.

Effektivität (efficiency) Das Produkt sollte den Benutzer dabei unterstützen, bestimmte Ziele korrekt und vollständig zu erreichen.

Produktivität (productivity) Unter Produktivität ist die Fähigkeit des Produkts zu verstehen, dem Benutzer bestimmte Ressourcen bereitzustellen, um effektiv bestimmte Ziele zu erreichen. Dazu gehören Zeit, Aufwand, Material oder monetäre Ressourcen.

Sicherheit (safety) Die Verwendung des Produkts sollte ein akzeptables Risiko für die betroffenen Personen und Organisationen darstellen. Risiken ergeben sich in der Regel aus Defiziten bei der Funktionalität und Sicherheit, der Zuverlässigkeit, Nutzbarkeit oder Wartbarkeit.

Befriedigung (satisfaction) Die Verwendung des Produkts sollte dem Benutzer Befriedigung verleihen und die weitere Verwendung motivieren.

2.2.5 Kenngrößen

Die Teilstandards ISO 9126-2, ISO 9126-3 und ISO 9126-4 definieren die Kenngrößen für die Merkmale der internen/externen und Gebrauchsqualität. In den Texten wird eingangs darauf hingewiesen, dass die Auflistungen keinen Anspruch auf Vollständigkeit erheben und sowohl ihr Umfang als die einzelnen Kenngrößen bei Bedarf angepasst oder erweitert werden können. Ausgeschlossen wird die Vorgabe konkreter Wertebereiche für die Kenngrößen, da diese für jedes (Teil-)Produkt in Abhängigkeit von den Anforderungen, Einsatzbereich etc. eigens festzulegen sind [03a, S. 1].

Kenngrößen für externe Qualität Die in ISO 9126-2 aufgeführten Kenngrößen für externe Qualitätsmerkmale werden durch das Beobachten oder Befragen der Benutzer, durch Vergleiche von Spezifikation und Umsetzung oder durch Zählen von Fehlern und Zugriffen innerhalb eines bestimmten Zeitintervalls vermessen. Bei einigen Kenngrößen wird außerdem eine Erfassung über einen längeren Zeitraum empfohlen, um Trendanalyse durchführen und mit Hilfe von Wachstumsmodellen Prognosen erstellen zu können. Im Anhang findet sich ein Beispiel für eine Spezifikation der Gebrauchsqualität im Rahmen der Anforderungsanalyse sowie eine Beispieltabelle für die Bewertung der Verwendungs- und externen Qualität. Geschlossen wird mit einer kurzen Einführung in die Maßtheorie (Skalenarten, Typen von Maßen).

Kenngrößen für interne Qualität ISO 9126-3 schlägt für die Erhebung das Zählen von implementierten Funktionen, Bausteinen, Zugriffen und Datensätzen vor. Dazu gehören Funktionen mit Logging-Fähigkeiten, die Anzahl Kommentarzeilen, die Änderungen festhalten, wird herangezogen sowie die Anzahl Variablen, die davon betroffen sind. Die Testabdeckung und -autonomie sind Kenngrößen für die Erfassung der Testbarkeit; sie beschreiben, wie unabhängig voneinander einzelne Komponenten getestet werden können, bspw. durch Verwendung von Stubs und Mocks. Bis auf Anhang E sind die Anhänge identisch mit ISO 9126-2.

Kenngrößen für die Gebrauchsqualität Die Gebrauchsqualität wird in ISO 9126-4 durch Beobachten bzw. Befragen der Benutzer erfasst. Die Effektivität bemisst sich anhand der Anzahl korrekt erledigter Teilaufgaben; die Produktivität setzt diese Werte mit der benötigten Arbeitszeit in Beziehung, um so die Arbeitsbelastung zu bestimmen. Die Sicherheit eines Produkts erstreckt sich auf vier Bereiche: Das Gesundheitsrisiko für den Benutzer selbst durch die Verwendung des Produkts, das bestehende Risiko für Dritte, der mögliche ökonomische Schaden und das Risiko für das System selbst. Konkret aufgetretenen Fälle werden gezählt und in Beziehung zur Anzahl aller Benutzer gesetzt. Der Grad der Befriedung wird mit Hilfe von Fragebögen ermittelt, die u. a. auf die Verwendung bestimmter Funktionen eingehen.

2.3 Auswertung des Standards

Inwieweit sich der Standard als Ausgangspunkt für die Entwicklung eines Qualitätsmodells für die Wartung eignet, wird anhand der folgenden Kriterien beurteilt:

- Bietet der Standard eine nachvollziehbare Zerlegung des Qualitätsbegriffs, der die Wartbarkeit als eine Qualität berücksichtigt?
- Wird ein Bezug hergestellt zu Quellcode-Artefakten, die die Wartbarkeit beeinflussen können?
- Gibt es Angaben zu Grenzwerte von Kenngrößen, die diese Artefakte erfassen?

Der Katalog der Eigenschaften aus ISO 9126, ihre Zerlegungssystematik und der Versuch, dazu konkrete Kenngrößen zu definieren, ist nachvollziehbar und deckt sich vom Ansatz her mit den Vorschlägen anderer Autoren (s. bspw. [Wal01, S. 35ff], [Bot+04]). Dementsprechend kann dieser Teil des Standards für die Definition eines Qualitätsbegriffs verwendet werden.

Der Standard stellt jedoch keinen Bezug zu Strukturen im Quellcode her und geht nicht darauf ein, wie mit Hilfe von Kenngrößen die Wartbarkeit bewertet werden kann. Nur in ISO 9126-3, Anhang E (Pure Internal Metrics) wird informell auf das Messen von Attributen eingegangen, die das interne Design und den Code des Produkts betreffen und Einfluss auf einige oder alle Eigenschaften und Merkmale des Systems haben. Darunter finden sich Volumenmaße wie Modulgröße in *Lines of Code*, *Modulkopplung*, das *Halstead*-Maß, *Fan-In/-Out* und die *zyklomatische Komplexität* nach McCabe. Es bleibt jedoch bei diesen wenigen feingranularen Kenngrößen, Architektur-relevante Ansätze werden nicht thematisiert. Entsprechend fehlen auch Angaben zu Grenzwerten.

Auffällig ist, dass der Standard nicht auf diejenigen Techniken näher eingeht, die für ihren werkzeuggestützten, Code-zentrierten Ansatz zur Qualitätssicherung bekannt sind: Software-Vermessung mittels Metriken wird seit vielen Jahren praktiziert (s. [Zus98, S. 5]). Die Rekonstruktion von Sichten aus Quellcode ist ebenfalls ein wichtiges Hilfsmittel das Programmverstehen beim Reengineering (s. bspw. [Kos05]).

Bevor auf Basis von ISO 9126 ein Qualitätsmodell zur Bewertung der Wartbarkeit entwickelt werden kann, muss zunächst geklärt werden, welche Artefakte und Strukturen die Wartbarkeit von Quellcode beeinflussen können. Dazu werden Methoden der Software-Vermessung und Software-Architekturrekonstruktion näher untersucht. Die dabei gewonnenen Erkenntnisse fließen dann in die Entwicklung des Qualitätsmodells ein. Eine Fundierung auf Basis der Maßtheorie, wie sie Zuse in [Zus98] vorschlägt, ist vor dem Hintergrund der Diskussion um die Weyuker-Eigenschaften von Komplexitätsmaßen (s. [Zus98, S. 369]) ratsam.

Das nachfolgende Kapitel gibt zunächst einen Überblick über die Disziplin der Software-Vermessung sowie ihrer Historie und beschreibt den aktuellen Stand der Forschung in diesem Bereich. Anschließend wird der Bezug zur Rekonstruktion von Software-Architekturen hergestellt und verdeutlicht, dass eine nähere Untersuchung des Architekturbegriffs notwendig ist.

3 Die Vermessung von Quellcode

> Miss alles, was sich messen lässt, und mach alles messbar, was sich nicht messen lässt.
>
> *(Galileo Galilei)*

Die Software-Vermessung ist wohl das älteste Verfahren, um die Produkt- und Prozessqualität von Software-Produkten empirisch zu erfassen. Sie versucht — in Anlehnung an das Messen, wie es bspw. in der Physik stattfindet — durch die Quantifizierung qualitativer Aspekte Aussagen zur Qualität von Softwaresystemen oder dem zugehörigen Entwicklungsprozess zu machen. Dazu wird ein *Softwaremodell* entworfen, mit dessen Hilfe auf die notwendigen Erfassungskriterien für eine Vermessungsmethode abstrahiert wird. Diese *Maße*, oft auch *Metriken* genannt, lassen durch Abbildung der Messwerte Rückschlüsse auf bestimmte Eigenschaften des Systems oder des Prozesses zu. Neben der Feststellung des Ist-Zustandes wird auch versucht, Prognosen bspw. zur Kostenentwicklung zu stellen (s. dazu u. a. das COCOMO-Modell von Boehm [Boe81]).

Der folgende Überblick zur zeitlichen und inhaltlichen Entwicklung ist z. T. aus Kans „Metrics and Models in Software Quality Engineering“ ([Kan02]) und Zuses „A Framework of Software Measurement“ ([Zus98]) entnommen, einer der wohl umfangreichsten Arbeiten aus diesem Gebiet. Danach wird eine Zusammenfassung des aktuellen Stands der Forschung gegeben und der Bezug zur Architekturerkennung bzw. -rekonstruktion hergestellt.

3.1 Historie

Das wissenschaftliche Messen an sich bzw. eine Fundierung desselben ist gut 120 Jahre alt. Zuse zitiert eine Beschreibung der Historie der Vermessung von Lemmerich, laut der das Messen 1887 mit der Arbeit „Counting and measuring from an Epistemological point of View“ von Helmholtz erstmals fundiert beschrieben wurde. Die seitdem erfolgten Bemühungen mündeten 1990 in der modernen axiomatisch aufgestellten Theorie von Luce et al., Krantz et al. und Roberts [Zus98, S. 57].

Das Messen im Kontext der Softwareentwicklung begann erst Mitte der 1960er Jahre. Motiviert wurde es durch das Wissen, dass die Programmstruktur und die Modularität wichtige Aspekte bei der Entwicklung zuverlässiger Softwaresysteme sind. Die Relevanz dieser Aspekte wurde u. a. von Parnas (s. [Par75]) beschrieben. Nach Shapiro trat in dieser Zeit die Effizienz der Systeme, die lange Zeit als das ausschlaggebende Kriterium der Entwicklung zugrunde gelegt wurde, zugunsten von Eigenschaften wie Wartbarkeit zunehmend in den Hintergrund [Zus98, S. 58].

Das erste Maß, das zudem auch noch heute breite Anwendung findet, ist *Lines of Code* (LOC). Dieses 1955 vorgestellte Volumenmaß zählt die Anweisungen im Quellcode und erfasst so die Größe von Systemen und Systembestandteilen. 1974 unternahm Wolverton einen der ersten Versuche, mit Hilfe von LoC die Entwicklerproduktivität zu messen. Shepperd demonstrierte, dass es eine starke Korrelation zu anderen Maßen wie bspw. zu McCabes Zyklomatischer Komplexität gibt [SI93, S. 10]. Nach diesen Versuchen mit Volumenmaßen entwickelte Rubey 1968 das erste Maß, das die Komplexität erfassen sollte. Auch Knuth befasste 1971 sich mit einer empirischen Untersuchung von FORTRAN-Programmen hinsichtlich ihrer Komplexität [**Knuth1971**]. Eine populäre Reihe von Komplexitätsmaßen wurde 1976 von McCabe aus der Graphentheorie abgeleitet [McC76]. Halstead prägte in [Hal77] den Begriff „Software Science“, dem die Idee zugrunde lag, dass wissenschaftliche Methoden auf Strukturen und Eigenschaften von Programmen angewendet werden sollten. Mit ihrer Hilfe konnte anhand des Quellcodes u. a. der Wartungsaufwand abgeschätzt werden. Seine Maße und die von McCabe gehören bis heute zu den bekanntesten Maßen überhaupt. Gilb veröffentlichte 1977 eine umfassende Arbeit zur Software-Vermessung, die sowohl die verschiedenen Anwendungsbereiche von Metriken beleuchtet als auch eine Einbettung in den Entwicklungsprozess vornimmt [Gil77]. Yourdon und Constantine befassten sich 1979 mit dem damals populären Paradigma der Strukturierten Programmierung [YC79]. Sie stellten dabei auch Überlegungen zum Entwurf von Systemen an. Auf den Arbeiten von Myers (s. [Mye75]) aufbauend untersuchten sie die verschiedenen Strukturierungsmöglichkeiten im Hinblick auf ihre Auswirkungen bei Änderungen. Sie entwarfen dazu Richtlinien zur Modularisierung, die sich an der Kopplung und Kohäsion von Bausteinen orientieren. Henry und Kafura versuchten 1981, mit Hilfe von Fan-In/Fan-Out die Komplexität von Prozeduren und Modulen zu bestimmen und sie zu Änderungen in Beziehung zu setzen [HK81]. Mitte der 1980er Jahre führte Zuse die Maßtheorie der Physik in die Softwaretechnik ein. Ihr axiomatischer Ansatz erlaubte die Charakterisierung und den Vergleich der hinter den verschiedenen Maßen stehenden qualitativen Modelle [Zus98, S. 66].

Zu beobachten ist eine Verschiebung von der Untersuchung feingranularer Strukturen, wie sie bspw. Halstead und McCabe durchführten, hin zur Betrachtung übergeordneter Strukturen und ihrer Relationen. Mit der beginnenden Verbreitung der Objektorientierung verfestigte sich dieser Trend und die Modularität von Systemen stand zunehmend im Mittelpunkt. Dabei wurden die klassischen Maße ergänzt durch Richtlinien und Heuristiken zum Entwurf objektorientierter Systeme.

Die der Einführung der Objektorientierung gemachten Versprechungen von einfacherer, besser zu verstehender Software haben sich nach Broy und Siedersleben bis heute nicht erfüllt [BS02], Qualitätsbetrachtungen müssen weiterhin vorgenommen werden, um wartbare Systeme zu erhalten. Demzufolge wurde von Morris 1989 mit der Anpassung bestehender Maße bzw. mit der Entwicklung neuer Maße begonnen. Lieberherr und Holland übertrugen im Rahmen des Demeter-Projekts die Ideen der Kapselung und Kohäsion auf die Objektorientierung und prägten den Begriff des „shy code“ [LH89]. Sie versuchten, den Impact einer Änderung so effektiv wie möglich zu begrenzen. Chidamber und Kemerer veröffentlichten 1991 mit „Towards a Metrics Suite for Object Oriented Design“ ([CK91]) eine Reihe bis heute verbreiteter Maße für die Vermessung von OO-Systemen, obwohl Untersuchung wie bspw. [Agg+06] Überlappungen der einzelnen Maße gezeigt haben. 1993 entwarf Lorenz elf Maße, mit deren Hilfe das Design eine OO-Systems bewertet werden können sollte. Für jedes Maß machte er auch Vorschläge

hinsichtlich der Grenzwerte (für die Programmiersprachen Smalltalk und C++). Die Liste der Maße beinhaltet auch zwei Prozessmaße, für die Daten aus dem Projektumfeld erforderlich sind [Kan02, S. 334ff]. Basili und Melo haben die C&K-Suite 1996 empirisch auf ihre Tauglichkeit als Qualitätsindikatoren hin untersucht und bestätigen können [BBM96].

Auch das 1995 von Abreu und Carapuça veröffentlichte MOOD-Metrik-Set hat die Bewertung des objektorientierten Designs zum Ziel [AC94]. Harrison et al. haben die Maße 1998 im Rahmen einer empirischen Untersuchung verwendet und dabei ihren Nutzen bestätigen können [HCN98]. Welche Auswirkungen das Design auf die Qualität haben kann, ermittelten Abreu und Melo 1996 durch den Vergleich der MOOD-Messergebnisse von acht Systemen, die nach identischen Anforderungen entwickelt wurden [AM96]. Briand et al. haben diese Beziehung 2000 näher untersucht, indem sie die Qualität mit der Fehlerhäufigkeit von Klassen verglichen [Bri+00].

Die Organisation von immer größeren Softwaresystemen erfordert eine Abstraktionsebene oberhalb der von Klassen. Die Einheiten dieser Ebene werden als *Module* oder *Pakete* bezeichnet. In seinen Artikeln für den C++-Report ([Mar96b; Mar96a; Mar96c; Mar96d; Mar96e; Mar97]), die er 2003 als Teil eines Buches veröffentlichte (s. [Mar02]), überträgt Martin das Kopplungs- und Kohäsionskonzept der Klassen auf die Ebene der Pakete, indem er sechs Entwurfsprinzipien beschreibt. Diese greifen z. T. Aspekte aus anderen Arbeiten auf, bspw. das Open-Close-Prinzip von Meyer (s. [Mey97, S. 57ff]) oder das Liskovsche Ersetzungsprinzip [Lis87]. Neben der Übertragung der Konzepte ist sein Ansatz, die *positionale Stabilität* eines Pakets zu ermitteln, sowie der Versuch, seine *Abstraktheit* zu berechnen, ein wichtiger Beitrag zur Thematik.

Für alle aufgeführten Ansätze gilt, dass vor dem Einsatz einer Metrik geprüft werden muss, inwieweit sie die gewünschten Kenngrößen ermitteln kann und ob die allgemein vorgegebenen Grenzwerte im jeweiligen Analysekontext sinnvoll sind. Ein Beispiel: Die Größe von Methoden einer Klasse ist normalerweise weitaus geringer als die einer Prozedur oder Funktion in einem dem strukturierten Programmierparadigma folgenden System. Die McCabe-Metriken könnten hier nur unter Zuhilfenahme zusätzlicher Technik greifen, da sie nicht mit Klassen und Vererbung umgehen können. Der Nutzen einzelner Metriken kann nicht in jedem Fall empirisch eindeutig nachgewiesen werden (s. [SI93, S. 3f]). Vor einem Einsatz muss daher eine Auswahl aus der Menge aller in Frage kommenden Maße stattfinden.

Mögliche Grenzwerte stehen als Problem aber erst an dritter Stelle, denn die Spezifikation einzelner Maße selbst kann nicht immer eindeutig interpretiert werden. Ein Beispiel dafür ist die Anzahl der Methoden einer Klasse (Number of Methods, NOM). Die Spezifikation macht keine Angaben, ob dabei nur die lokal definierten Methoden erfasst oder auch die von einer Oberklasse geerbten Methoden mit einbezogen werden müssen. Dasselbe gilt für überschriebene, Klassen- und Instanzmethoden. Ohne eine vorher erfolgte detaillierte Spezifikation sind derartig ungenau definierte Maße für eine Bewertung nicht geeignet.

3.2 Aktueller Stand

Zwar gibt es immer noch Forschungen im Bereich der Software-Vermessung, in der Wirtschaft etabliert haben sich jedoch hauptsächlich die im vorigen Abschnitt aufgeführten Maße (vgl.

[Zus98, S. 491]). Damit hat sich in der Praxis ein weitgehend stabiles Instrumentarium gebildet. Vergleicht man den Anteil an Maßen für die Prozessqualität mit dem Anteil an Maßen für die Produktqualität, die vom Entwickler direkt während der Entwicklung erhoben werden können, so liegt in der Literatur der Schwerpunkt eher bei der Analyse der Prozessqualität (vgl. bspw. [EDB04] oder [Kan02]). Die zunehmende Verbreitung agiler und iterativer Vorgehensmodelle wird ebenfalls aufgegriffen, bspw. untersuchen Alshayeb und Li de Nutzen von Maßen, um in derartigen Projekten Vorhersagen über den Wartungsaufwand machen zu können [AL03].

Bei der Produktqualität ist eine Entwicklung von der Code-Ebene weg hin zur Analyse und Bewertung des Systementwurfs zu beobachten. Durch die Bewertung des Entwurfs wird versucht, frühzeitig eine gute Qualität zu erzielen, die sich anschließend auch im Quellcode niederschlagen soll. Neben Quellcode werden dabei auch UML-Modelle analysiert, die im Rahmen der Entwurfstätigkeit entstehen (bspw. [GPC05]). In enger Verbindung steht dazu die Untersuchung der Qualitätseigenschaften von Entwurfsmustern. Huston betrachtet dazu die Kompatibilität von Software-Maßen und Entwurfsmustern [Hus01]. Muraki und Saeki hingegen verwenden in [MS01] Maße, um den Einsatz von Entwurfsmustern im Rahmen von Refactoring-Maßnahmen zu prüfen. Auch die Wiederverwendung von Komponenten bzw. deren Eignung wird untersucht. Insgesamt stehen Komponenten und ihre Einsatzmöglichkeiten zunehmend im Fokus (s. bspw. [WYF03; GA04]).

Lanza und Marinescu versuchen in [LM06] den Schritt vom Maß zur Eigenschaft, indem sie Messergebnisse durch Filter auf die relevanten Werte reduzieren und sie anschließend mit einem auf den logischen UND- und ODER-Operationen basierenden Kompositionsmechanismus verbinden. Auf diese Weise können auch komplexere Design-Regeln untersucht werden. Damit eröffnen sich zwei Wege, um ein System mit solchen *Detektoren* zu analysieren: (1) Man versucht, „gutes Design“ mit Regeln und heuristischen Daten zu definieren, oder (2) Symptome „schlechten Designs“ zu finden und damit die Artefakte, die einer Überarbeitung bedürfen. Diesen letzten Ansatz verfolgt auch Fowler in [Fow99] mit den sog. *Code-Smells*. Sie weisen auf Code-Abschnitte hin, die strukturell verbessert werden müssen. Code-Smells und die zugehörigen Refactoring-Maßnahmen sind in der Entwicklergemeinschaft weithin anerkannt und als Umsetzung in modernden integrierten Entwicklungsumgebungen wie der Eclipse Workbench vorhanden.

Software-Vermessung ist zunehmend eng verbunden mit der Darstellung der Ergebnisse in Form einer *Visualisierung*. Die Darstellung von Vererbungsgraphen gehört mittlerweile zu jedem Design-Werkzeug dazu. Ducasse und Lanza haben mit den *Class Blueprints* eine Darstellung vorgeschlagen, durch die von den Detektoren gesammelten „verdächtigen“ Artefakte so dargestellt werden können, dass ihre potentiellen Design-Defizite unmittelbar zutage treten [DL05]. Für einen Gesamtüberblick über die Komplexität eines Systems kann die „Overview Pyramid“ benutzt werden, mit der Messergebnisse zu Größe und Komplexität, Kopplung und Vererbung kombiniert gezeigt werden können. Die Systemstruktur hingegen bilden sie mit Hilfe *polymetrischer Darstellungen* ab [LM06].

Neuere Techniken wie die *Aspektorientierte Programmierung* erlauben aufgrund ihrer vergleichsweise geringen Verbreitung und der fehlenden Erfahrung in der Wartung großer Legacy-Systeme noch keine belastbaren Aussagen hinsichtlich ihrer Qualität und bieten auch keine entsprechend gerüsteten Maße.

Die Bewertung von Software-Architekturen, insbesondere hinsichtlich der Wartbarkeit, ist

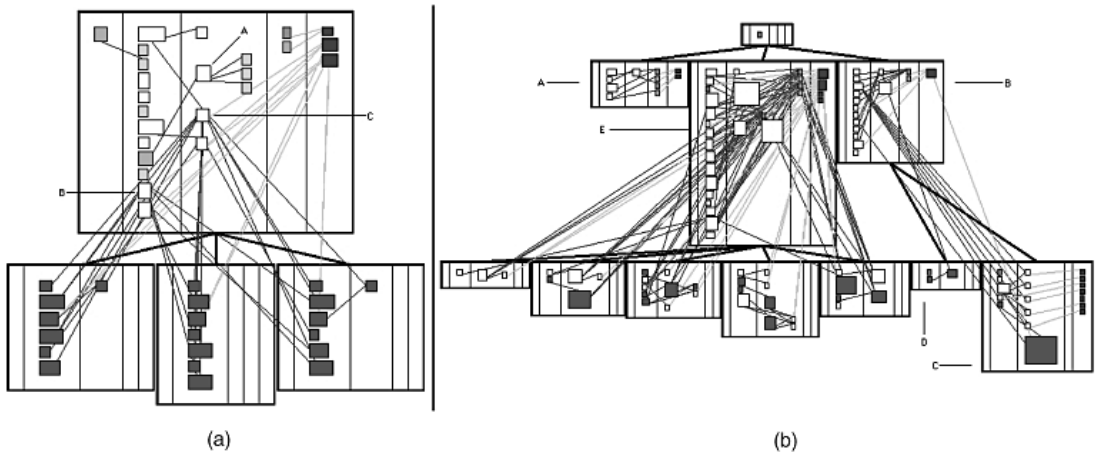

Abbildung 3.1: Beispiel für einen *Class Blueprint* (aus [DL05]).

noch Forschungsgebiet. Für eine Vermessung und Bewertung von Architektur-relevanten Artefakten ist zunächst eine begriffliche Auseinandersetzung mit anschließender Definition erforderlich; danach folgt die Identifizierung der zu bewertenden Artefakte. Erst dann kann eine Übertragung der Metrikkonzepte versucht werden. Es geht schlicht um die Frage: „Was ist eine gut wartbare Architektur?“ Wie Kapitel 4 zeigen wird, ist allein schon die Definition des Architekturbegriffs schwierig, von Handlungsempfehlungen für die Bewertung ganz zu schweigen. Die Identifizierung Architektur-relevanter Artefakte ist zudem ein ganz eigener aktiver Forschungsbereich.

Der nachfolgende Abschnitt beschreibt den Bezug der Vermessung zur Software-Architektur, der geprägt ist von der Erkennung bzw. Rekonstruktion und gibt einen Überblick über den Stand der Forschung.

3.3 Architekturerkennung aus Quellcode

Die Vermessung des Quellcodes eines Systems auf den verschiedenen Abstraktionsebenen ist kein Problem, solange es für die Elemente dieser Ebenen eine syntaktische Entsprechung in der Programmiersprache gibt. Leider bieten die etablierten Sprachen eine Unterstützung nur bis zur Ebene der Module bzw. Pakete an. Für eine Vermessung oberhalb dieser Ebenen kann das Messobjekt daher nicht immer zweifelsfrei bestimmt werden. Darüber hinaus stellt sich auch die Frage, ob die üblichen Maße auf diesen Ebene effektiv anwendbar sind. Bevor also eine Untersuchung stattfinden kann, muss die Architektur eines Systems auf möglichst vielen Abstraktionsebenen rekonstruiert werden. Dabei ist die Vermessung nur eines von mehreren Zielen; oftmals geht es im Rahmen von Wartungsarbeiten darum, zunächst einen Überblick über das System zu gewinnen.

Um Informationen zur Architektur eines Systems überhaupt in irgendeiner Form festzuhalten, nutzen viele Entwickler die zugehörige Systemdokumentation oder versuchen, höhere Abstraktionsebenen bspw. mit Hilfe von projektspezifischen Präfix-/Suffix-Konventionen für

Bezeichner nachzubilden oder mit Vorgaben zur Verwendung bestimmter Entwurfsmuster (s. dazu bspw. [JES03]) eine systemweite Strukturierung zu erreichen. Sowohl der Dokumentationsansatz als auch die artifizielle Nachbildung können nur bis zu einem gewissen Grad automatisiert untersucht werden, da sie entweder keinen formalen Richtlinien folgen oder so projektspezifisch sind, dass umfangreiche Anpassungen am Analysewerkzeug erforderlich sind. Die enge Projektbindung kann dabei auch die Vergleichbarkeit der Messergebnisse zu Untersuchungen anderer Systeme einschränken. Da die Dokumentation der Architektur nach einer Änderung am Quellcode i. d. R. nur unzureichend nachgeführt wird [Kos05, S. 127], konzentriert sich die Vermessungsproblematik auf die Untersuchung von Architektur-relevanten Artefakten im Quellcode.

Die Verwendung von Software-Maßen beschränkt sich jedoch nicht nur auf die Bewertung vorhandener Artefakte. Bevor eine Messung vorgenommen werden kann, müssen die *Messobjekte* identifiziert und in den Systemkontext eingeordnet werden. Maße können diesen Prozess mit ihrer Bewertungsfunktionalität unterstützen. Die Bestimmung der Kopplung und Kohäsion von Klassen hilft bspw. bei der Identifizierung von Klassen-Clustern. Eine konkrete Architektur kann aber auch als Lösung eines Partitionierungsproblems aufgefasst werden, bei dem anhand der Kopplung und Kohäsion eine optimale Gruppierung verfolgt wird [Kos05, S. 130].

Ebenso wie ihre Bewertung ist auch die Erkennung von Architektur noch ein aktives Forschungsgebiet. Durchgesetzt hat sich die Erkenntnis, dass es abhängig vom konkreten Interesse eines Betrachters unterschiedliche *Blickwinkel* gibt, die auf bestimmte Aspekte einer Architektur abzielen. Demzufolge spricht man auch nicht mehr von *der* Architektur, sondern von einer Menge von Architekturen. Koschke gibt in [Kos05] einen umfangreichen Überblick über die Literatur und die Methoden zur Architekturrekonstruktion. Ausgehend vom Standard IEEE 1471 ([IEE00]) kategorisiert er die verschiedenen Ansätze anhand der Blickwinkel, die rekonstruiert werden sollen. Formal definiert ein solcher Blickwinkel die Regeln und Semantik einer konkreten *Sicht* auf ein System und kann damit auch als ihr *Typ* verstanden werden. Eine Sicht ist demnach eine Repräsentation eines Systems aus der Perspektive einer Menge verwandter Anliegen. Die Rekonstruktion einer Architektur lässt sich nun beschreiben als Abbildung von einer Ausgangssicht auf eine Zielsicht, wobei die Ausgangssicht diejenigen Daten beschreibt, auf die sich die Rekonstruktion stützt. Die Zielsicht ist das Resultat der Rekonstruktion. Diese Abbildungen sind allerdings nur selten formal definiert; in der Praxis besteht der Rekonstruktionsprozess in weiten Teilen aus einer Kombination von manuellen und semi-automatischen Schritten. Er ist damit noch immer geprägt von Kreativität, Heuristiken und erfordert Erfahrung [Kos05, S. 129].

Die Ausgangssichten stützen sich in der Regel auf den Quellcode eines Systems und auf Beobachtungen zur Laufzeit. Dementsprechend unterscheidet man *statische* und *dynamische* Analysen. Statische Analysen umfassen abstrakte Syntaxbäume, Kontroll-, Datenfluss- und Abhängigkeitsgraphen. Sie haben das Ziel, Elemente wie Module, Klassen etc. zu beschreiben, die als Repräsentationen das Ziel der Rekonstruktionsbestrebungen sind. Dynamische Analysen liefern Darstellungen eines konkreten Programmablaufs. Solche Traces sind jedoch abhängig von Eingaben und erlauben keine allgemein gültigen Aussagen über das System. Gerade diese Aussagen werden aber mit einer Rekonstruktion gesucht. Für die Analyse kommen noch andere Informationsquellen in Frage: der Entwicklungsprozess des Systems, ein verwendetes Versionsverwaltungssystem und natürlich die zugehörige Dokumentation. Diese Quellen bieten

u. U. Ansatzpunkte für die Entwicklung von Hypothesen, die eine Analyse erleichtern können, indem sie den Umfang des Suchraums verringern.

3.3.1 Der Modulblickwinkel

Vor dem Hintergrund der Vermessung sind Ausgangssichten auf Basis statischer Analysen von Interesse, da sich die meisten Maße auf die statische Struktur des Quellcodes stützen. Der überwiegende Teil der von Koschke aufgeführten Arbeiten hierzu befasst sich mit der Rekonstruktion des *Modulblickwinkels*. Dieser Blickwinkel beschreibt die verschiedenen statischen Strukturen des Quellcodes, wobei ein Modul als Übersetzungseinheit verstanden wird, die eine bestimmte Funktionalität umsetzt. Die Untersuchung der *Dekomposition* nimmt dabei den größten Raum ein. Daneben werden u. a. Klassenhierarchien und -modelle sowie Schnittstellen und Entwurfsmuster untersucht, die Differenz zwischen Ist- und Sollarchitektur betrachtet und eine Zuordnung der Produktfunktionalitäten zu Modulen vorgenommen. Die verschiedenen Ansätze werden nun kurz vorgestellt.

- Dekomposition: Ziel der Dekompositionsbestimmung ist, die Unterteilung der Systemfunktionalität durch Analyse der Teil-von-Beziehungen zu ermitteln, welche die Quellcode-Menge in Gruppierungen verwandter Artefakte untergliedern (z. B. kooperierende Klassen oder ähnliche Datentypen). Im Wesentlichen werden für die Zuordnung statische Abhängigkeitsbeziehungen untersucht. Neben bekannten Clustering-Verfahren und genetischen Algorithmen sind für die Eingruppierung auch Techniken aus dem Bereich der Graphentheorie geeignet. Dazu gehört u. a. die Dominanzanalyse und die Mustererkennung auf Graphen [Kos05].

- Schnittstellen: Die Untersuchung von Schnittstellen als strukturierende Elemente auf syntaktischer Ebene ist nur auf den ersten Blick trivial. Sie besitzen mehr oder weniger eindeutige syntaktische Entsprechungen (mehr in Java, weniger in C++), die die direkten Beziehungen zwischen Klassen und Modulen widerspiegeln. Allerdings spielen neben diesen direkten auch indirekte Beziehungen eine Rolle. Yourdon und Constantine thematisieren Beziehung dieser Art vor dem Hintergrund von Änderungen und deren Seiteneffekten (s. [YC79]). Neben den rein syntaktischen Aspekten repräsentieren Schnittstellen immer auch eine Menge von Annahmen, die sich in Form von Vor- und Nachbedingungen gegenüber der „Umwelt“ (s. dazu auch [Mey97, Kapitel 11]) sowie in Protokollen manifestieren. Protokolle beschreiben die Art und Weise, wie eine bestimmte Schnittstelle zu benutzen ist bzw. wie dies im Quellcode geschieht. Ihre Analyse erfordert Datenflussanalysen, die sich bei objektorientierten Systemen mit ihren Late-Binding-Eigenschaften als schwierig erweisen.

- Klassenmodelle und -hierarchien: Die Ermittlung einer Klassenhierarchie auf syntaktischer Ebene ist trivial. Die tatsächlich im Quellcode umgesetzte Verwendung entspricht allerdings nicht immer der Hierarchie und dem dabei intendierten Vererbungsgedanken, weshalb eine Ermittlung derselben oftmals sinnvoll ist. Klassenmodelle beleuchten die Beziehungen von Klassen untereinander, wobei formal zwischen Assoziation und Ag-

gregation unterschieden wird. Da es aber keine syntaktische Entsprechung dieses Unterschieds gibt, gestaltet sich eine Herleitung schwierig.

- Entwurfsmuster: Neueren Datums ist die Suche nach Entwurfsmustern (s. zu Entwurfsmustern auch [Gam+96; Lar04]). Vornehmlich werden dabei Ansätze der statischen Mustererkennung über Graphen oder abstrakte Syntaxbäume oder Data-Mining-Techniken verwendet. Bei eher verhaltensorientierten Mustern kommen zusätzliche Informationen aus der Protokollanalyse hinzu. Das Hauptproblem bei der Suche ist allerdings die kombinatorische Explosion, der einige Methoden mit Hilfe von Heuristiken beizukommen versuchen.

- Konformität: Sobald Annahmen über eine Architektur vorliegen, sei es im Rahmen einer Analyse oder als Entwicklungsvorgabe, können sie mit der tatsächlich vorliegenden Architektur abgeglichen werden. Die Ergebnisse dieser Konformitätsprüfungen zeigen Differenzen auf, die durch Anpassungen des Modells oder der Implementierung beseitigt werden können.

- Merkmalsabbildung: Die Implementierung einer Funktionalität oder *Verantwortlichkeit* eines Softwareprodukts erfolgt idealerweise durch ein Modul oder, je nach Definition des Modulbegriffs, durch einen eng gefassten Modulverbund. Inwieweit diese Abgrenzung eingehalten wird, lässt sich anhand einer Abbildung der Produktfunktionen auf Module ermitteln. Diese Abbildung kann sowohl statisch als auch dynamisch erfolgen, wobei die statische Form auf Seiten des Analytikers Wissen über die Implementierung voraussetzt. Dabei sucht der Analytiker in einem extrahierten Abhängigkeitsgraphen nach merkmalsrelevanten Routinen. Dynamische Abbildungen verfolgen, welche Modulaufrufe die Anwendung einer bestimmten Produktfunktion auslöst und visualisieren diese Zuordnung.

Vergegenwärtigt man sich die Details des üblichen Wartungsprozesses, so ist besonders die Merkmalsabbildung hilfreich, grenzt sie doch den Suchraum, sprich: die zu untersuchende Quellcode-Menge massiv ein. Die Voraussetzungen für diese Abbildung (=Wissen um die Implementierung) sind durch ein Werkzeug jedoch nur teilweise zu erfüllen. Zusammen mit der Dekomposition, die letztendlich die Basis der Klassenhierarchien- und -modelle inkl. Schnittstellen eines Systems ist, zeigen die Arbeiten u. a. von Yourdon und Constantine ([YC79]) zur Kopplung und den verschiedenen Formen der Kohäsion eine mögliche Alternative zum Implementierungswissen auf.

Weitere Untersuchungen befassen sich mit der Rekonstruktion von Use Cases und dem Einfluss von Konfigurationen auf ein System bzw. dessen Quellcode.

3.3.2 Komponenten-Konnektoren-Blickwinkel

Der Komponenten-Konnektoren-Blickwinkel stützt sich auf die Architekturbetrachtungen von Shaw und Garlan ([SG96]). Sie unterteilen die Bestandteile eines Systems in *Komponenten*, die eine bestimmte Funktionalität erfüllen, und *Konnektoren*, die die Komponenten miteinander verbinden. Dementsprechend erlangen die Elemente dieses Blickwinkels erst zur Laufzeit Bedeutung. Trotz dieses Umstands sind Ansätze zur dynamischen Analyse zwar stärker, aber

nicht übermäßig vertreten. Gesucht werden Quellcode-Strukturen, die als Konnektoren den Daten- oder Kontrolltransfer zwischen Komponenten realisieren. Dabei wird i. d. R. vereinfachend angenommen, dass eine Komponente eine Laufzeitinstanz eines Moduls aus dem Modulblickwinkel (z. B. einer Klasse) ist. *Dass* zwei Komponenten interagieren und in welcher Reihenfolge und Häufigkeit dies geschieht, kann mit Hilfe dynamischer Analysen nur näherungsweise bestimmt werden. Wie bei der Merkmalsabbildung ist auch hierbei Wissen über die Implementierung notwendig.

Statische Objektspuren zeichnen den Weg der Aufrufe nach, die ein Objekt erhält bzw. tätigt. Sie verfeinern die Untersuchungsergebnisse zur Kollaboration um Angaben zur Aufrufreihenfolge und -häufigkeit.

Sichten des *konzeptionellen Blickwinkels* beschreiben ein System mit Konzepten aus der Anwendungsdomäne. Diese enge Bindung an die Anwendungsdomäne verhindert allerdings eine vollständig automatische Rekonstruktion, da dazu Wissen über die Domäne und etwaige Referenzarchitekturen unabdingbar ist. Derartige Hintergrundinformationen können durch Interviews und die Analyse existierender Dokumentation gewonnen werden und ergänzen die aus anderen, implementierungsnäheren Sichten ermittelten Erkenntnisse.

3.3.3 Weitere Blickwinkel

Ein Großteil der Veröffentlichungen befasst sich mit den beiden bisher vorgestellten Blickwinkeln (s. [Kos05]). Die übrigen Arbeiten widmen sich den nachfolgend aufgeführten „Nischen"-Perspektiven.

- Einbettung: Die Sichten dieses Blickwinkels betrachten die Einbettung des Systems in seine Umgebung. Das betrifft sowohl die Koexistenz mit anderen Systemen als auch die Betriebssystemumgebung.
- Zuständigkeit: Dieser Blickwinkel zielt eher auf die Projektorganisation ab, da er die Zuordnung einzelner Entwickler bzw. Entwicklergruppen zu Quellcode-Bestandteilen aufzeigt. Dies ist insbesondere dann von Interesse, wenn die Entwicklung von (räumlich) getrennten Teams durchgeführt werden soll.
- Generierung: Unter Generierung wird der Build-Prozess eines Systems verstanden. Dementsprechend befassen sich die Sichten dieses Blickwinkels mit den zugehörigen Skripten, Makefiles und Generatoren.
- Dateien: Das Hauptaugenmerk dieses Blickwinkels liegt in der Zuordnung der logischen Module der Sichten des Modulblickwinkels zu Elementen der Dateiebene, wie also der Quellcode in Dateiform in der Entwicklungsumgebung organisiert ist. Je nach Programmiersprache ist diese Überbrückung zwischen physischer und logischer Sicht mehr oder weniger gut nachvollziehbar.

Ein umfassendes Systemverständnis ist nur zu erlangen, wenn die Sichten der verschiedenen Blickwinkel *zusammen* betrachtet werden können. Dazu müssen Abbildungen die Elemente der einzelnen Sichten zueinander in Beziehung setzen. Zur Erzeugung derartiger Meta-Sichten

werden u. a. Überlappungen im Quellcode und lexikalische Übereinstimmungen von Elementbezeichnern als Beziehungsmerkmale herangezogen.

3.4 Zusammenfassung

Die Disziplin der Software-Vermessung ist trotz der verfügbaren Methoden und Techniken noch nicht in der Lage, aus dem Stand einem Entwickler eine Bewertung der Wartbarkeit einer gegebenen Architektur zu liefern. Als problematisch erweist sich die Anwendbarkeit der Maße auf Artefakte der Architekturebene und die Ermittlung dieser Artefakte aus dem Quellcode. Die Bewertung der „Wartbarkeit“ wird nicht tiefergehend behandelt. Es bleibt unklar, was eine diesbezüglich „gute“ Architektur charakterisiert.

Für die Beantwortung der Forschungsfragen dieser Arbeit ergeben sich daraus folgende Aufgaben:

- Der Begriff der Software-Architektur muss näher untersucht werden mit dem Ziel, ihn für eine Qualitätsbewertung enger zu fassen und Hinweise zu finden, welche Merkmale eine wartbare Architektur ausmachen.
- Die dabei gewonnen Erkenntnisse fließen in den Aufbau eines Qualitätsmodells ein, das eine Brücke schlägt von der abstrakten Qualitätseigenschaft „Wartbarkeit“ zu konkreten Maßen, die auf aus dem Quellcode rekonstruierte Sichten anwendbar sind und deren Tauglichkeit empirisch nachgewiesen ist.

Der Standard IEEE 1471 bietet einen geeigneten Ausgangspunkt für die Suche nach einer Definition des Architekturbegriffs, stützt er sich doch in weiten Teilen auf die Arbeiten von Bass et al., Perry und Wolf, Shaw und Garlan sowie Zachman (s. [BCK98; PW92; SG96; Zac87]). Ein Studium dieser Quellen ist daher sinnvoll, um einerseits die aus Sicht der Autoren relevanten Aspekte von Software hinsichtlich ihrer Architektur zu verstehen. Andererseits werden mit der Betrachtung auch Ratschläge zu ihrer Umsetzung einhergehen, die für das zu entwickelnde Qualitätsmodell hilfreich sein können.

Besonders die Kopplungs- und Kohäsionseigenschaften scheinen Merkmale zu sein, die für eine wartbare Strukturierung des Quellcodes eine wichtige Rolle spielen. Sie tauchen sowohl auf der feingranularen Ebene der Funktion, Methoden und Klassen auf als auch bei der Organisation größerer Systeme auf. Bei der Rezeption der Arbeiten zur Architektur sollte daher ein besonderes Augenmerk auf Hinweise dazu gelegt werden.

4 Software-Architektur

> Architektur ist im Idealfall immer direkte Auseinandersetzung mit den Menschen.
>
> *(Richard Meier)*

Die Objektorientierung bzw. die sie umsetzenden Programmiersprachen besitzen keine Werkzeuge für die Abbildung Architektur-relevanter Artefakte. Die zur Verfügung stehenden Konstrukte wie Klassen oder Pakete sind dafür zu feingranular [BS02, S. 5]. Dieses Defizit zeigt sich auch bei der Bewertung der Qualität auf dieser Ebene: Aus Sicht der Software-Vermessung ist sie weitgehend ungeklärt. Allerdings können Richtlinien zum Architekturentwurf Hinweise geben, wie eine „gute“ Architektur beschaffen sein sollte. Dementsprechend werden die nachfolgend beschriebenen Arbeiten zum Thema Software-Architektur mit dem Ziel untersucht, die eine Architektur konstituierenden Artefakte zu identifizieren und charakteristische Merkmale für die Qualitätsbewertung zu sammeln.

Auch wenn im Pflichtenheft kein Dokument zur Architektur zu finden ist, so gilt trotzdem: „Jedes Softwaresystem hat eine Architektur, auch wenn diese nicht explizit modelliert wurde.“ [Has06, S. 49]. Diese kann mittels Techniken zur Architekturrekonstruktion bis zu einem gewissen Grad aus Quellcode extrahiert und einer Bewertung zugänglich gemacht werden. Die Beschreibung bzw. explizite Modellierung ist kein Selbstzweck, sondern verfolgt in der Regel die folgenden Hauptziele:

- Dokumentation der Entwurfsentscheidungen
- Kommunikation und Verständnis der Systemstruktur
- Bewertung der Qualität nicht-funktionaler Aspekte

Voraussetzung für eine Beschreibung ist eine geeignete Begriffsdefinition, die anschließend von den verschiedenen Beschreibungsmethoden und Darstellungstechniken wie bspw. der UML aufgegriffen werden kann. Die mittlerweile erreichte Akzeptanz des Begriffs und seiner Relevanz für Softwaresysteme veranlasste die IEEE Computer Society, mit dem Standard IEEE Std 1471-2000 einen entsprechenden Definitionsversuch zu wagen und Richtlinien für die Beschreibung von Softwaresystemen zu verfassen. Im nachfolgenden Abschnitt werden die für eine Qualitätsbetrachtung relevanten Aspekte des Standards untersucht und geprüft, ob sie für eine Qualitätsbewertung der Systemarchitektur durch den Entwickler geeignet sind.

4.1 Software-Architekturen nach IEEE 1471-2000

Der Standard IEEE Std 1471-2000 ist ein Leitfaden für die *Beschreibung der Architektur Software-intensiver Systeme* [IEE00]. Charakteristisch für derartige Systeme ist der maßgebliche Einfluss von Software auf den Systementwurf, die Konstruktion, Auslieferung und Evolution. Trotz einer fehlenden Definition zeigt die breite, wenn auch unpräzise Nutzung von Begriffen wie „architecture“ oder „architecture level“ durch die Entwickler- und Forschergemeinde die Akzeptanz einer Architekturmetapher. Das Bewusstsein für die Notwendigkeit einer derartigen Sichtweise auf Systeme hat zur Entwicklung verschiedener Methoden und Werkzeuge für den Entwurf und die Beschreibung von Software-Architektur geführt. Es herrscht jedoch weder Einigkeit bei der Begriffsdefinition noch bei der Art und Weise, wie Architektur zu beschreiben ist. Unter [09] ist bspw. eine Liste vieler gängiger Definitionen zu finden. Allen Definitionsversuchen gemein ist die Idee, die Architektur mit *unterschiedlichen Sichten* zu beschreiben. Diese unterscheiden sich nicht nur im Detaillierungsgrad voneinander, sondern insbesondere dadurch, *was* dargestellt werden soll: Statische Aspekte wie Klassenhierarchien oder Systemschichten, dynamische Aspekte wie Aufrufgraphen oder nebenläufige Prozesse usw. Jede Sicht zeigt dabei nur die für den jeweiligen Betrachter relevanten Details und reduziert so die Komplexität und den Umfang des Systems. Das Sichtenkonzept wird vom Standard aufgegriffen und in einem *Architekturbeschreibungskonzept* weiter ausgeführt.

4.1.1 Begriffsdefinitionen und konzeptueller Rahmen

Die IEEE versteht Software-Architektur als „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“ [IEE00, S. 3]. Die Definition weist schon auf den konzeptuellen Rahmen für die Architekturbeschreibung hin, in dem die weiteren Begriffsdefinitionen erfolgen. Abbildung 4.1 illustriert die Beziehung der Begriffe.

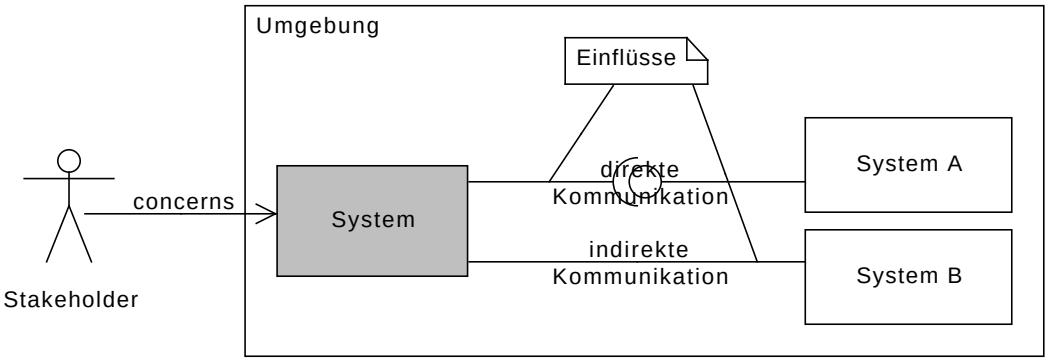

Abbildung 4.1: System, Kontext und Stakeholder (eigene Darstellung).

Systemumgebung Ein System wird immer im Zusammenspiel mit der Umgebung („context“, „environment“) betrachtet, in die es eingebettet ist. In der Umgebung können andere Systeme existieren, die direkt über Schnittstellen oder indirekt auf andere Weise mit dem

System interagieren können. Die sich daraus ergebenden Einflüsse können entwicklungsbezogener, operationeller, politischer oder anderer Natur sein. Ein System ist kein Selbstzweck, sondern erfüllt immer eine oder mehrere Aufgaben („mission“), die sich aus den Bedürfnissen („concerns“) eines oder mehrerer Interessengruppen („stakeholder“) ergeben. Neben den funktionalen Anforderungen können dabei auch Vorgaben zur Performance, Zuverlässigkeit etc. gemacht werden.

Komponenten und Beziehungen Der fundamentale Aufbau eines Systems besteht aus einer Menge von Komponenten („components“), die miteinander in Beziehung („relationships“) stehen. Dieser Aufbau ist die *Architektur des Systems* und kann durch *Software-Architekturbeschreibungen* („architectural descriptions“) näher erläutert werden. Die Art der Komponenten und der Beziehungen werden durch die *Sichtweise* bestimmt.

Views und Viewpoints Die Beschreibung der Architektur eines Systems erfolgt durch sog. *Sichten.* („views“). Eine Sicht ist eine „representation of a whole system from the perspective of a related set of concerns“ [IEE00, S. 3]. Die Architektur tritt somit immer vor dem Hintergrund einer bestimmten Interessenlage zutage. Das wiederum bedeutet, dass es zu jedem System immer mehrere Sichten gibt.

Wie eine Sicht zustande kommt, bestimmt der *Blickwinkel* oder die *Perspektive* („viewpoint“). Er gibt vor, wie eine konkrete Sicht erstellt und abgebildet wird, er ist sozusagen der *Typ* der Sicht. Dazu gehören sowohl die verwendeten Beschreibungsmethoden (Notationsarten, Modelltypen etc.; s. bspw. UML [BHK04]) als auch die Analyse- und Modellierungstechniken. Üblicherweise benutzt eine Architekturbeschreibung mehrere Blickwinkel. Die Auswahl wird bestimmt durch den Zweck, den die Architekturbeschreibung für den oder die Betrachter erfüllen soll.

Informationen, die nicht Teil einer Sicht sind, können in der üblichen Systemdokumentation aufgeführt werden. Dies betrifft bspw. die Systemübersicht, Angaben zur Systemumgebung und zu den Interessengruppen inkl. deren Bedürfnisse sowie die Begründung der Entwurfsentscheidung für eine bestimmte Architektur.

Architekturmodelle Eine Sicht kann aus einem oder mehreren *Architekturmodellen* bestehen. Jedes Modell wird mit denen durch den zugehörigen Blickwinkel bestimmten Methoden erstellt. Ein Architekturmodell kann dabei mehr als einer Sicht zugeordnet werden, vorausgesetzt, die Blickwinkel sind kompatibel (s. Abb. 4.2 auf S. 32).

Der Standard empfiehlt, bei großen Systemen zu jeder Komponente eine eigene Architekturbeschreibung zu verfassen. Die dabei entstehenden Modelle können durchaus in den Beschreibungssichten der anderen Komponenten auftauchen. Trotz dieser Verbindung bleiben die Beschreibungen aber eigenständig. Blickwinkel müssen nicht direkt in der Architekturbeschreibung definiert werden, sondern stattdessen darf auf eine anderswo vorliegende Definition verwiesen werden. Diese so definierten Blickwinkel heißen „library viewpoints“ und sollten bei wiederholtem Einsatz in einen Katalog aufgenommen werden.

Abbildung 4.3 auf S. 33 zeigt den Aufbau des konzeptuellen Modells der Architekturbeschreibung, wie es im Standard definiert wird. Als Notation wurde ein UML-Klassenmodell gewählt.

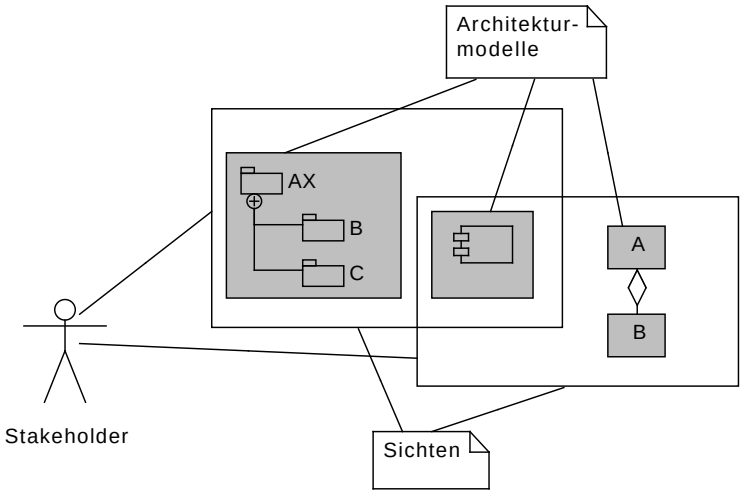

Abbildung 4.2: Sichten mit gemeinsamen Architekturmodellen (eigene Darstellung).

Standardkonforme Architekturbeschreibung sollten die folgenden sechs Elemente aufweisen:

- Identifikation, Version und Überblick: Die Identifikation wird für die Einordnung in die übrige Dokumentation des Systems benötigt. Eine Versionierung inkl. Historie zeigt die Entwicklung der Beschreibung auf und ermöglicht rückblickend eine Überprüfung von Architekturentscheidungen. Neben einer Zusammenfassung sollte auch ein Glossar sowie eine Liste referenzierter Dokumente und Standards aufgeführt sein. Als Orientierung dazu kann der Standard IEEE/EIA Std 12207.0-1996 dienen.

- Identifikation der für die Architektur relevanten Interessengruppen und Bedürfnisse: Zu den aufzuführenden Interessengruppen gehören mindestens Anwender, Käufer, Entwickler und Wartungsingenieure; die relevante Bedürfnisse umfassen neben dem eigentlichen Zweck auch Angaben zur Eignung, zu Risiken und zur Wartbarkeit. Weiterhin sollten Angaben zu den aus der Systemumgebung kommenden Anforderungen gemacht werden.

- Auswahl der Blickwinkel, die für die Darstellung der Architektur verwendet werden, sowie die Begründung der Auswahl: Jeder Blickwinkel sollte spezifiziert werden durch einen eindeutigen Namen, die Zielgruppe und deren Belange. Außerdem sollten die zu verwendenden Sprachen, Modellierungstechniken und Analysemethoden, die zur Konstruktion einer Sicht dieses Blickwinkels dienen, aufgelistet werden. Dies gilt auch für Verweise auf anderswo definierte Blickwinkel (Library Viewpoints). Die Auswahl sollte zudem begründet werden.

- Sichten auf das System: Grundvoraussetzung ist, dass jede Sicht konform zu genau einem Blickwinkel sein muss. Sie besitzt eine eindeutige Kennung sowie den Vorgaben der Organisation entsprechende einleitende Informationen. Eine Sicht kann ein oder mehrere

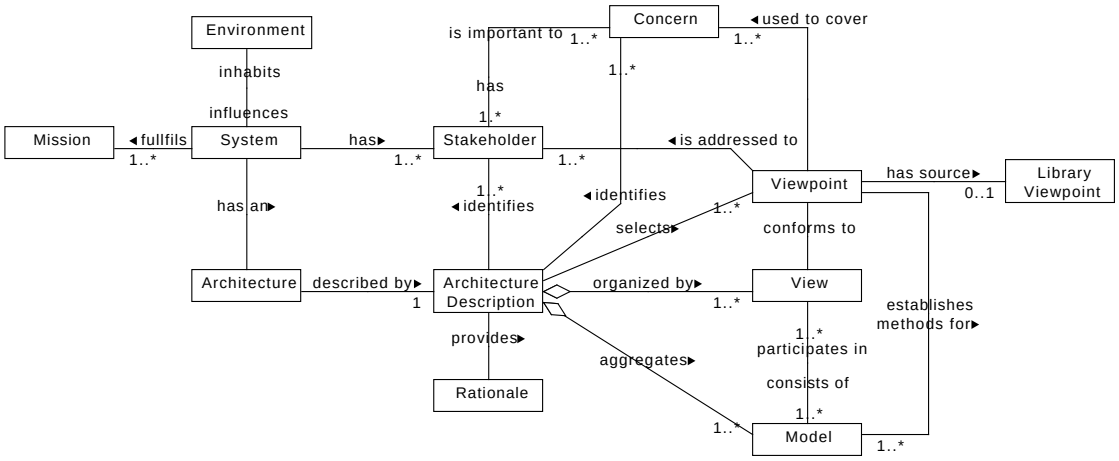

Abbildung 4.3: Modell des Architekturbeschreibungskonzepts (nach [IEE00, S. 5]).

Architekturmodelle enthalten, die mit den durch den zugehörigen Blickwinkel vorgegebenen Hilfsmitteln umgesetzt sein müssen.

- Liste mit Inkonsistenzen: Sämtliche Inkonsistenzen, die zwischen den verwendeten Sichten vorliegen, sollten aufgeführt werden. Zudem sollte eine Konsistenzanalyse der übrigen Sichten vorgenommen und das Ergebnis mit in die Dokumentation eingefügt werden.

- Begründung für die Auswahl der vorliegenden Architektur: Die durch die Architektur umgesetzten Konzepte sollten aufgeführt und begründet werden. Wurden alternative Architekturen untersucht und abgelehnt, sollte dies ebenfalls festgehalten werden. Dies gilt auch bei der nachträglichen Dokumentation von Altsystemen, sofern noch bekannt ist, welche Entscheidungen warum getroffen wurden.

4.1.2 Beispiele für Blickwinkel

Die im Anhang C des Standards aufgeführten Beispiele für Blickwinkel entstammen den Arbeiten von Perry und Wolf (s. [PW92]) sowie Shaw und Garlan (s. [SG96]). Blickwinkel für die Struktur eines Systems verwenden *Komponenten* ([SG96]) oder *Elemente* ([PW92]), die mittels *Konnektoren* bzw. *verbindenden Elementen* zueinander in Beziehung stehen. Nach Shaw und Garlan definiert ein *Architekturstil* ein Vokabular von Komponenten und Konnektoren sowie eine Menge von Regeln für deren Einsatz, die für Ausprägungen dieses Stils verwendet werden dürfen. In der Spezifikation eines solchen Blickwinkels müssen also die Komponenten eines Systems identifiziert werden, ihre Schnittstellen und die Verbindungen der Komponenten untereinander. Verhaltensorientierte Blickwinkel gehen der Frage nach, welche Art von Aktionen es in einem System gibt, wie sie zueinander in Beziehung stehen (geordnet, nebenläufig etc.) und wie die Systemkomponenten interagieren. Im Fokus stehen dabei Ereignisse, Prozesse, Zustände und zugehörige Operationen.

Neben den als Notationsart häufig verwendeten Diagrammen kann ein Blickwinkel durch eine Formel definiert werden. Der Standard zeigt als Beispiel dafür Modellierung der Bit-Fehlerrate eines Kommunikationskanals.

4.2 Die Bedeutung des Standards für Entwickler

Die Übernahme des Sichtenkonzepts von Zachman, Perry und Wolf sowie Shaw und Garlan ist insofern aus Entwicklersicht positiv zu beurteilen, als sie eine Reduzierung der Komplexität des Systems auf die für den jeweiligen Betrachter relevanten Aspekte ermöglicht. Konkret bedeutet das eine Dekomposition des Systems in statische und/oder dynamische Komponenten, die je nach Blickwinkel weiter unterteilt und durch Modelle beschrieben werden können. Ebenfalls positiv zu sehen ist die Einbettung in einen konzeptuellen Rahmen, der die Zweckbindung der Beschreibung verdeutlicht: Die Einschränkung der Sichtenvielfalt durch die Anforderungslage vermeidet überflüssige Betrachtungen und führt zu zielgruppengerecht zugeschnittenen Blickwinkeln und Sichten.

Allerdings verbleibt die Richtlinie auf dieser abstrakten Ebene und führt auch mit den Beispielen im Anhang keinen Katalog bewährter Blickwinkel ein. Die Empfehlungen beschränken sich auf die Metadaten der Dokumente wie Identifikation oder Version, liefern darüber hinaus aber nur Hinweise auf Quellen wie [BCK98] und [Zac87], denen der Entwickler konkrete Vorschläge zu Blickwinkeln entnehmen können soll. Dementsprechend werden auch keine Angaben zu Techniken und Vorgehensweise bei der Architekturrekonstruktion gemacht. Der Verweis auf den Standard IEEE/EIA 12207.0-1996 ([98]) ist keine Hilfe, da auch an dieser Stelle nur allgemeine Empfehlungen zu finden sind. Damit fehlt die für Entwickler wichtige Verbindung zum Quellcode und den daraus extrahierbaren Artefakten, welche eine Voraussetzung für die Qualitätsbeurteilung der Architektur sind. Interessant ist zudem, dass die Begriffe „Komponente“ und „Konnektor“ nicht im Modell des Architekturbeschreibungskonzepts auftauchen.

Hinsichtlich der Bewertung von Architekturen macht der Standard keine Angaben, sondern stellt nur ihre Relevanz für die zu treffenden Designentscheidungen heraus.

Zusammenfassend ist festzuhalten, dass sich der Standard aus Sicht eines Entwicklers nur als Ausgangspunkt für die Formulierung eines eigenen Architekturbegriffs eignet, der eine Ermittlung von Artefakten aus Quellcode ermöglicht mit dem Ziel, die Wartbarkeit bewerten zu können. Benötigt wird ein *Wartungsblickwinkel* mit entsprechenden Sichten bzw. Architekturmodellen. Dazu werden in den nachfolgenden Abschnitten die Quellen zum Standard IEEE 1471-2000 und weitere relevante Arbeiten untersucht.

4.3 Die Entwicklung des Architekturbegriffs und sein Bezug zur Wartbarkeit

Die fortschreitende Entwicklung der Hardware, insbesondere der Speicherfähigkeit, sowie die an Größe und Komplexität zunehmenden Softwaresysteme erforderten neue Ansätze, um die Entwicklung und Wartung kontrollieren zu können. Die Techniken zur Modularisierung, wie sie bspw. Parnas in seinem Artikel „The influence of software structure on reliability“ (s. [Par75])

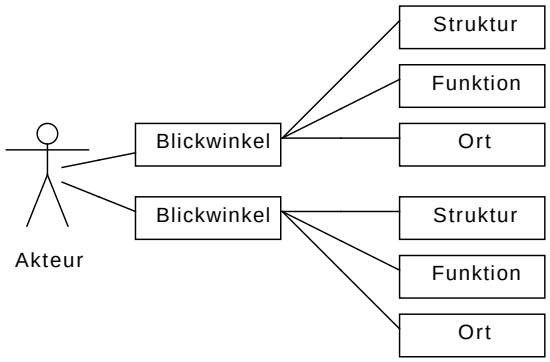

Abbildung 4.4: Beispiel: Akteur mit zwei Blickwinkeln und zugehörigen Beschreibungsarten (eigene Darstellung).

beschreibt, waren ein erster Ansatz, der auch heute noch Bestand hat. Damit trat die Zuverlässigkeit und Wartbarkeit der Systeme an die Stelle der zuvor alles beherrschenden Effizienz. Der gewachsene Gestaltungsspielraum erforderte eine neue Sichtweise auf die Systemstruktur, um Schnittstellen und Komponenten definieren und kontrollieren zu können.

4.3.1 Sichtenkonzept nach Zachman

In Anlehung an die klassische Architektur schlägt Zachman in [Zac87] eine Reihe von *Blickwinkeln* vor, deren konkrete *Sichten* bestimmte Aspekte eines Informationssystems herausstellen und so seine Architektur konstituieren sollen.

In seinem Framework kombiniert er die beiden folgenden Ideen: Auf der einen Seite gibt es eine Reihe von Darstellungsklassen, die die Architektur aus einem bestimmten Blickwinkel erfassen, auf der anderen Seite verschiedene Beschreibungsarten für einen Blickwinkel, die jeweils einen bestimmten Aspekt des Systems hervorheben. Für die beteiligten Akteure ergibt sich also eine durch ihre Interessen determinierte Menge von Blickwinkeln, die jeweils durch eine ausgewählte Beschreibungsart auf einen spezifischen Aspekt des Systems eingehen. Das Beispiel in Abbildung 4.4 illustriert die Zusammenhänge.

Die wesentliche Erkenntnis aus Zachmans Beitrag besteht darin, dass es zu einem System nicht *die* Architektur gibt, sondern eine *Menge* von Architekturen. Diese Menge setzt sich zusammen aus den Sichtweisen der Beteiligten, und zwar in Abhängigkeit davon, welche Aufgabe er oder sie zu erfüllen hat. Er weist in seinem Fazit auch auf mögliche Schwierigkeiten bei der Kommunikation und der Bewertung hin, da es nicht „die bessere“ Architektur gibt und betont, dass eine nicht explizit erarbeitete Architektur zu einem Risiko für die Entwicklung werden kann.

Die Definition eines Wartungsblickwinkels erfordert also zunächst die genauere Bestimmung der Aufgabe eines Entwicklers im Kontext der Wartung sowie der sich daraus ergebenden Anforderungen an die Architektur(en) eines Systems.

4.3.2 Architektur nach Perry und Wolf

Perry und Wolf greifen in ihren „Foundations for the Study of Software Architecture“ ([PW92]) das Sichtenkonzept von Sowa und Zachman auf. Die Sichten ihres Architekturmodells bestehen aus Modellelementen, deren Anordnung sowie Erläuterungen, die die Wahl gerade dieser Konstellation begründen. Die Faktoren, die aus Sicht der Autoren mit für die hohen Software-Kosten verantwortlich sind und auch die Architektur betreffen, sind die *Evolution* und die *Anpassung* eines Systems. Die mit der Evolution einhergehenden tiefgreifenden Änderungen führen dabei oft zu einem Verfall der ursprünglichen Struktur. Sie identifizieren zwei wesentliche Risiken: Eine *Architekturerosion* tritt dann auf, wenn die Regeln der gegebenen Architektur verletzt werden, wie bspw. bei der Nichtbeachtung der Ordnung einer geschichteten Architektur. Eine *Architekturdrift* hingegen ist das Resultat einer unzureichenden Kenntnis der Strukturen und Regeln, die nicht zu einer Verletzung, aber zu einer Verwässerung der Architektur führt. Sie kann die Erosion begünstigen.

Um diese Risiken zu minimieren, sollte eine Darstellung nicht nur *be*schreibenden, sondern auch *vor*schreibenden Charakter haben. Die Autoren leiten daraus folgende Anforderungen an Architekturbeschreibungen ab:

- Die Beschreibung der Vorgaben der spezifischen Architektur muss den nötigen Detaillierungsgrad aufweisen. Dabei sollen jedoch nur diejenigen Bedingungen berücksichtigt werden, die für die jeweilige Architekturebene relevant sind.
- Es muss eine Unterscheidung der relevanten, „tragenden“ Elemente der Architektur von „dekorativen“ Elementen erfolgen. Auf diese Weise bleibt die Kernaufgabe der Architektur sichtbar erhalten und die Erosion wird unterbunden.
- Die verschiedenen Aspekte einer Architektur müssen auf angemessene Weise beschrieben werden. Dies kann durch Wahl geeigneter Blickwinkel erfolgen.
- Eine Beschreibung soll Abhängigkeits- und Konsistenzanalysen ermöglichen. Die Abhängigkeitsanalyse soll die Wechselwirkung von Architektur, Anforderungen und Design herausstellen; durch die Konsistenzanalyse kann die (möglicherweise fehlende) Übereinstimmung von Architektur und -stil sowie der Elemente nachgewiesen werden.

Vor diesem Hintergrund verstehen Perry und Wolf Software-Architektur als „... a set of architecture (or, if you will, design) elements that have a particular form“ [PW92, S. 44]. Sie definieren ihr Architekturmodell als Tripel

$$\text{Software Architecture} = \{\text{Elements, Form, Rationale}\}.$$

Elemente können dabei sein: (1) *processing elements*, die Transformationen auf Daten ausführen, (2) *data elements*, die die zu transportierenden und zu verarbeitenden Daten enthalten, und (3) *connecting elements*, die die Bestandteile der Architektur bspw. durch Prozeduraufrufe, gemeinsame Datenbereiche etc. miteinander verbinden. Die *Architekturform* wird bestimmt durch die Gewichtung von Eigenschaften und Beziehungen der Elemente. Die *Gewichtung* kann entweder die Relevanz betonen und so die Unterscheidung von notwendigen und dekorativen

Elementen ermöglichen, oder die alternativen Entwürfe bzgl. der umzusetzenden Vorgaben bewerten. Die *Eigenschaften* geben das Minimum an Bedingungen vor, die die ausgewählten Architekturelemente erfüllen müssen. Die *Beziehung* der Elemente zueinander definiert ihre Anordnung, d. h. den Aufbau bestehend aus Prozess-, Daten- und Verbindungselementen. Die *Erläuterung* (Rationale) ist ein Teil der Beschreibung, die die Vielzahl an Entscheidungen festhält, welche zu dieser konkreten Architekturausprägung geführt haben. Dazu gehört sowohl die Wahl der Elemente als auch der Form und damit des *Architekturstils*. Geleitet werden die Entscheidungen von den funktionalen und nicht-funktionalen Anforderungen an das System.

Im Gegensatz zu einer konkreten Architektur gibt ein *Architekturstil* abstrakte Elemente und mögliche Verbindungen vor und kann so als *Typ* einer Architektur verstanden werden. Er spezifiziert charakteristische Eigenschaften, die sich bspw. mit Begriffen wie „parallel“ oder „verteilt“ beschreiben lassen, und hebt so die bestimmende Anordnung der Bestandteile hervor. Beispiele für verschiedene Architekturstile sind u. a. in [GS94] und [SG96] zu finden. Die Grenze zwischen Stil und Ausprägung ist jedoch fließend und hängt nicht zuletzt auch vom Betrachter und seiner Definition des Architekturbegriffs ab; was für den einen Architekten noch ein Architekturstil ist, kann für den anderen schon eine konkrete Ausprägung sein.

Die drei wichtigsten Sichten auf ein System sind den Autoren nach die Prozess-, Daten- und Verbindungssicht. In der Prozesssicht wird der Datenfluss durch die Processing Elements betrachtet und die damit einhergehenden Anforderungen an die Verbindungen der Elemente. Die Datensicht hingegen orientiert sich am Verarbeitungsfluss und weniger an den Verbindungen. Prozess- und Datenelemente stehen darüber hinaus in einer wechselseitigen Abhängigkeitsbeziehungen: Die Transformation eines Datums durch ein Prozesselement erzeugt als Resultat einen neuen Zustand dieses Datenelements. Die Prozesselemente müssen diese Zustände berücksichtigen, wenn sie eingehende Datenelemente verarbeiten wollen, d. h. die Verbindungen dürfen nur zwischen kompatiblen Prozesselementen existieren. Trotz der verbreiteten Objektorientierung glauben Perry und Wolf, dass diese drei Sichten mit ihren jeweiligen Schwerpunkten notwendig und für die Architekturebene von Bedeutung sind. Sie bieten neben einem Rahmen für die Anforderungsanalyse und den Entwurf auch eine Möglichkeit, Analysen durchführen zu können. Durch die Identifikation von Komponenten und ihrer Abhängigkeiten wird zudem die Wiederverwendung verbessert. Beide raten allerdings dazu, diese auf dem Architekturlevel zu definieren und nicht auf die Implementierungsebene auszudehnen, da dort zu viele weitere Bedingungen berücksichtigt werden müssen.

Der Beitrag von Perry und Wolf erweitert das Architekturverständnis dahingehend, dass ihre Definition eine formale Identifikation einzelner Architekturstile ermöglicht. Die Beschreibung der Architekturform erlaubt zudem eine Bewertung alternativer Entwürfe; im Gegensatz zu Zachmans Ansicht lassen sich Architekturen so anhand ihrer möglichen Eignung auswählen. Motiviert durch die Hauptbedrohungen (Drift und Erosion) erweitern sie außerdem die Funktion der Architekturdarstellung von einer reinen Beschreibung hin zu einer Vorgabe für den Systementwurf. Wie an ihrer Definition des Architekturbegriffs zu erkennen ist, sehen sie die Dokumentation der Entwurfsentscheidungen gleichberechtigt zur Darstellung. Dies ist ein Versuch, die Nachvollziehbarkeit der Entwicklung der Systemstruktur sicherzustellen und das Drift- bzw. Erosionsrisiko bei weiteren Arbeiten am System möglichst gering zu halten. Wie eine Verletzung der Regeln zu bewerten ist, führen Perry und Wolf nicht näher aus.

Aus Sicht des Entwicklers ist ein geringes Drift-/Erosionsrisiko demnach gleichzusetzen mit

einer besseren Wartbarkeit eines Systems. Daraus ergibt sich die Frage, wie Architektur bzw. die ihr zugrunde liegenden Regeln und die „tragenden“ Elemente im Quellcode (wieder-)erkannt werden können und sich so die Architektur rekonstruiert lässt. Der Blickwinkel auf die technische Umsetzung („designers view“ und „builders view“) mit seinen Beschreibungsarten (E-R- und Datenfluss-Diagramme) ist aufgrund seiner Abstraktheit dazu nicht geeignet.

4.3.3 Empirische Untersuchungen zu Architekturstilen

Die bisher vorgestellten Arbeiten bereiteten durch das Sichtenkonzept und zugehörige Begriffsdefinitionen den Weg für detaillierte Betrachtungen einzelner Aspekte von Software-Architektur. Die Interessen „nicht-technisch“ orientierter Beteiligter wie bspw. der Auftraggeber werden aufgewertet und als Voraussetzung für eine effektive Entwicklung angesehen. Eine Betrachtung der technischen Umsetzung, deren Strukturen weitreichende Auswirkungen auf das spätere System haben, wird jedoch nicht weiter ausformuliert. Die von Perry und Wolf angebotenen Beispiele betrachten Systeme zwar aus der Perspektive der Entwickler, sind dabei aber zu abstrakt, als dass sie für konkrete Darstellungen herangezogen werden könnten.

Einen für Entwickler geeigneteren Zugang präsentieren Garlan und Shaw in ihrem Bericht „An Introduction to Software Architecture“ ([GS94]). Auch sie machen die zunehmende Größe der Systeme verantwortlich für die Verlagerung des Design-Problems von der Ebene der Algorithmen und Datenstrukturen auf eine höhere Ebene, die *Architekturebene* des Designs. Auf dieser Ebene geht es primär um die Organisation und Kommunikation der Systembestandteile, die Zuordnung von Funktionalität zu Komponenten und um die Skalierung und Performanz. Ermöglicht wird dies durch den Fortschritt der Programmiersprachen, der regelmäßig zu „größeren“ konzeptuellen Entwurfsblöcken geführt hat: Von linearem Code über Schleifen-Strukturen, Prozeduren/Funktionen bis hin zu abstrakten Datentypen und Modulen, Klassen und Paketen. Mit Hilfe dieser neueren Sprachelemente können Bausteine der Architekturebene besser implementiert werden.

Garlan und Shaw weisen zwar darauf hin, dass eine allgemein anerkannte Taxonomie architekturaler Paradigmen noch in weiter Ferne liegt, jedoch können schon jetzt eindeutige Muster identifiziert werden. Um die Merkmale einzelner Muster herausarbeiten zu können, definieren sie ihren Architekturbegriff wie folgt:

- Ein System bzw. seine Architektur besteht aus einer Menge von *Komponenten*, die auf bestimmte Weise durch *Konnektoren* miteinander verbunden sind.
- Ein *Architekturstil* wird definiert durch die Beschreibung der strukturellen Organisation dieser Bestandteile. Dabei wird die Menge erlaubter Komponenten und Konnektoren und ihre Anordnung spezifiziert sowie eine Reihe von Bedingungen vorgeschrieben, die erfüllt sein müssen, damit ein System einen bestimmten Stil aufweist. Die Bedingungen können bspw. topologischer Natur sein.

Garlan und Shaw unterscheiden Architekturstile für Einzelsysteme und sog. verteilte Systeme, die sich über mehrere Prozesse oder auch Maschinen erstrecken. Anhand der Definition konnten sie die Architektur der untersuchten Softwaresysteme verschiedenen Stilen zuordnen.

Pipes and Filters: Eine Komponente eines Systems dieses Architekturstils besitzt eine Reihe von Eingabe- und Ausgabekanälen. Die eingehenden Daten erfahren eine Transformation, deren Ergebnis als Ausgabe an weitere Komponenten geleitet werden kann. Die Transformation erfolgt i. d. R. inkrementell, so dass schon eine Ausgabe verfügbar wird, bevor die Eingabe vollständig verarbeitet ist (daher die Bezeichnung „filter“). Die erzeugten Datenströme werden über Konnektoren, die „pipes“, an die Komponenten weitergeleitet, wobei die zu transportierenden Daten keinen Einschränkungen unterliegen.

Eine wichtige Invariante dieses Architekturstils setzt voraus, dass die Komponenten unabhängig voneinander arbeiten, d. h. sie haben keine Kenntnis voneinander und verfügen auch nicht über gemeinsame Zustände oder Speicherbereiche. Sie teilen einzig die ein- und ausgehenden Formate der Datenströme. Eine Spezialisierung dieser Invariante sind die sog. „pipelines“; sie beschränken die Topologie auf eine lineare Sequenz von Filtern. Bekannte Vertreter hierfür sind Unix-Shell-Programme und ältere Compiler-Architekturen.

Data Abstraction and Object-Oriented Organization: Die Komponenten dieses Architekturstils sind Objekte, die als Instanzen abstrakter Datentypen Daten und zugehörige Operationen kapseln. Sie verantworten die Integrität einer Ressource (=Daten), weshalb Garlan und Shaw sie auch *Manager* nennen. Als Konnektoren fungieren die Methodenaufrufe, über die Objekte miteinander interagieren.

Die Vorteile der Kapselung, die entsprechend dem Parnas'schen Geheimnisprinzip die Daten nur über definierte Schnittstellen verfügbar macht, sind weithin bekannt: Ohne dem Anwender des Objekts etwas mitteilen zu müssen, kann die interne Datenrepräsentation modifiziert werden, solange die Schnittstelle unverändert bleibt. Daraus ergibt sich eine Entkopplung der einzelnen Objekte, die die Wartung und Wiederverwendung verbessert. Außerdem unterstützt dieser Stil einen reduktionistischen Lösungsansatz im Sinne des „Teile-und-Herrsche“-Prinzips: Zugriffsroutinen, kombiniert mit den von ihnen adressierten Daten, können als Menge von interagierenden Agenten umgesetzt werden, die jeweils ein bestimmtes Teilproblem lösen.

Ein großer Nachteil der Objektorientierung im Vergleich zum Pipe-Filter-Stil besteht in der Notwendigkeit, dass einem Objekt die Identität desjenigen Objekts bekannt sein muss, dessen Methoden aufgerufen werden sollen. Diese Zwangsverbindung führt zu möglichen Seiteneffekten, wenn ein so von mehreren Objekten referenziertes Objekt bspw. gelöscht wird. Zusätzlich kann man noch anführen, dass die Beobachtungen der letzten Jahre darauf hinweisen, dass OO-Systeme mit zunehmender Größe nicht nur schnell an Übersichtlichkeit verlieren können, sondern der Ansatz generell von den populären OO-Sprachen nur unzureichend umgesetzt wird (s. dazu bspw. [BS02]).

Garlan und Shaw weisen in einer Fußnote darauf hin, dass sie die Vererbung — eines der Hauptmerkmale der Objektorientierung — nicht als Methode zur Architekturgestaltung ansehen. Dies ist insofern richtig, da die Vererbungsbeziehung mit den Möglichkeiten, die die verwendete Definition des Architekturbegriffs bietet, nicht als Konnektor angesehen werden kann. Allerdings gibt es eine Reihe von Möglichkeiten, über eine Vererbungsbeziehung Architektur-relevante Strukturen zu schaffen (z. B. über Mehrfachver-

erbung, Ableiten von abstrakten Klassen und Schnittstellen). Dies lässt sich gut an den von Gamma et al. in [Gam+96] vorgestellten und mittlerweile weit verbreiteten Entwurfsmustern zeigen.

Event-Based, Implicit Invocation: Anders als bei den bisher vorgestellten Architekturstilen, bei denen Prozeduren bzw. Methoden direkt aufgerufen werden, erfolgt die Kommunikation in solchen ereignisbasierten Systemen implizit über die Systeminfrastruktur. Jede Komponente kann sich für bestimmte Ereignisse beim System registrieren und dabei Prozeduren seiner Schnittstelle angeben, die beim Auftreten eines dieser Ereignisse aufgerufen werden sollen. Neben den Prozeduren kann die Schnittstelle auch Ereignisse aufweisen, die die Komponente emittieren kann. Konnektoren setzen sich also zusammen aus den Ereignissen der Komponenten, der Aufruf- oder *Broadcast*-Infrastruktur des Systems und den für diese Ereignisse registrierten Komponentenprozeduren. Da sich mehr als eine Komponente für ein bestimmtes Ereignis registrieren kann, führt das Emittieren zu beliebig vielen, in ihrer Reihenfolge unbekannten Prozeduraufrufen. In vielen Systemen dieser Art ist es allerdings auch möglich, Prozeduren direkt aufzurufen und so die Infrastruktur zu umgehen.

Die Integration von Komponenten ist mittlerweile soweit gediehen, dass Begriffe wie *Enterprise Service Bus* und *Middleware* bei der Entwicklung großer Anwendungen zunehmend eine Rolle spielen. Nicht nur der Datentransfer, auch die Kommunikation der Komponenten wurde durch neue Protokolle standardisiert. Die Integration verschiedenster Bausteine — die sogar in unterschiedlichen Programmiersprachen implementiert sein können — ist heute als *Service-orientierte Architektur* (SOA) bekannt (s. dazu bspw. [ST07]).

Layered Systems: Schichtarchitekturen gehören mit zu den bekanntesten Architekturstilen. Die Komponenten dieses Stils sind in hierarchischen Schichten angeordnet, wobei eine Schicht immer nur die Prozeduren der direkt unterliegenden Schicht aufrufen kann. Diese unterliegende Schicht repräsentiert so eine „virtuelle Maschine“, auf der die nächsthöhere Schicht implementiert wird. Die Kommunikation erfolgt ausschließlich an den Schichtgrenzen. Die Konnektoren werden damit auf die Schnittstellen der Komponenten reduziert. Invarianten dieses Stils durchbrechen die strikte Trennung der Schichten und isolieren tiefer liegende Schichten nicht so stark. Dies erleichtert u. U. die Zuordnung von Funktionalitäten zu einzelnen Schichten und vermeidet die Implementierung ausschließlich nach unten delegierender Prozeduren, die keine eigene Logik besitzen. Eines der bekanntesten Beispiele für eine Schichtarchitektur das ISO OSI-Protokoll und das daraus abgeleitete Kommunikationsprotokoll TCP/IP. Weiterhin sind die meisten der etablierten Betriebssysteme in Schichten bzw. Ringen organisiert.

Repositories: Dieser etwas irreführende Name bezeichnet Systeme, die eine zentrale Komponente besitzen, deren Aufgabe in der Verwaltung einer systemweiten Datenstruktur besteht. Diese Datenstruktur repräsentiert den Systemzustand. Neben dieser zentralen Komponente operieren alle weiteren „Satelliten“-Komponenten unabhängig voneinander unter Verwendung der Datenstruktur. Zwei Kategorien von Repository-basierten

Systemen können anhand des Kontrollflusses im System identifiziert werden: Bestimmen die Arten der eingehenden Transaktionen die auszuführenden Prozesse, kann die zentrale Datenstruktur in Form einer *Datenbank* umgesetzt werden. Ist hingegen der Zustand der Datenstruktur Auslöser von Prozessen, eignet sich ein sog. *Blackboard* für die Implementierung. Die Komponenten werden dann als *knowledge source* bezeichnet, da sie einen Teil des anwendungsspezifischen Wissens enthalten. Weil die Kommunikation der Komponenten mit der Datenstruktur — sei es nun Datenbank oder Blackboard — sehr vielfältig sein kann, spezifizieren Garlan und Shaw die Konnektoren dieses Stils nicht näher. Die Interaktion der Satelliten-Komponenten untereinander erfolgt jedoch ausschließlich über die zentrale Komponente.

Die Blackboard-Ausprägung dieses Stils weist Gemeinsamkeiten mit dem Stil ereignisbasierter Systeme auf; auch dort wird dezentral an der Problemlösung gearbeitet. Die Komponenten überwachen den Zustand des Blackboards und reagieren eigenständig, wenn die Datenlage eine weitere Bearbeitung ermöglicht, und schreiben die Ergebnisse anschließend zurück ins Blackboard. Auf diese Weise wird inkrementell eine Lösung des Problems herbeigeführt. Datenbank-gestützte Systeme nutzen ihre zentrale Komponente als umfangreichen, effizient implementierten und persistenten Speicher.

Die vorgestellten Architekturstile haben universellen Charakter und sind nicht für einen bestimmten Einsatzbereich vorgesehen. Durch die zunehmende Verbreitung der IT haben sich verschiedene *Anwendungsdomänen* herauskristallisiert, für die es spezifische *Referenzarchitekturen* gibt. Durch die Beschränkung auf eine Domäne kann mit diesen domänenspezifischen Architekturen auf Anforderungen wesentlich besser eingegangen werden als dies in einer universellen, nicht genau abzugrenzenden Umgebung möglich wäre. Ein Beispiel für eine solche Domäne ist die Automobilindustrie, in der u. a. Sicherheitsaspekte großen Einfluss auf die IT-Systeme in den Fahrzeugen haben. Aus Sicht der Architekturerkennung sind an Referenzarchitekturen angelehnte Systementwürfe einfacher zu rekonstruieren, da durch die Vorgaben der Referenz die Kernelemente der Struktur feststehen und für die Rekonstruktion direkt genutzt werden können.

Weiterhin wird durch die Auflistung suggeriert, dass die Stile in „Reinkultur“ vorkommen. Tatsächlich ist dies eher die Ausnahme, vielmehr sind *heterogene Architekturen* die Regel. Architekturstile können hierarchisch kombiniert werden, d. h. auf jeder Hierarchieebene kann eine andere Architektur vorherrschen. Die Komponente einer Pipes-Filters-Architektur kann also selbst eine objektorientierte Struktur besitzen. Auf diese Weise können auch Konnektoren hierarchisch dekomponiert werden: Der Pipe-Konnektor wird mglw. intern durch eine FIFO realisiert. Die Art der Konnektoren muss für eine Komponente zudem nicht zwingend dieselbe sein: Die Komponente kann einen Konnektor für die Verbindung zu einem Repository besitzen und einen anderen für die Verbindung zu einem Filter.

Heterogene Architekturen erschweren die Rekonstruktion, da u. U. für jedes Element ein anderes Analyseverfahren eingesetzt werden muss. Die automatische Identifizierung eines Architekturstils anhand seiner spezifischer Merkmale wird damit wesentlich aufwändiger und die Genauigkeit und Zuverlässigkeit eines allgemeineren Untersuchungsansatzes sinkt.

Die bis dato eher theoretisch geführten Diskussion über Architektur und -stile ergänzen Garlan und Shaw um konkrete Beispiele, die in existierenden Systemen gefunden wurden und den

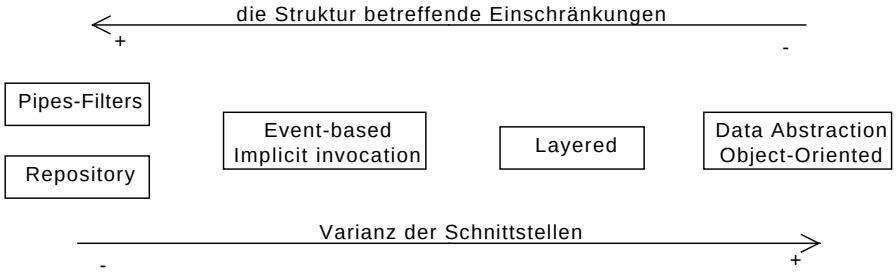

Abbildung 4.5: Varianz der Schnittstellen und Struktureinschränkungen.

meisten Entwicklern schon begegnet sein dürften. Sie sind damit die ersten, die einen Katalog von realen und verbreiteten Architekturstilen aufgestellt haben sowie Vor- und Nachteile durch direkten Vergleich von Implementierungen zu ermitteln versuchten.

Bei näherer Betrachtung der Vor- und Nachteile der Architekturstile lassen sich zwei Tendenzen erkennen:

- Die Komponenten sind in ihrer Funktionsweise weitgehend eigenständig und je nach Stil bis zu einem gewissen Grad voneinander abhängig. Die Eigenständigkeit und eine möglichst geringe gegenseitige Kopplung fördern die Wiederverwendung und Wartbarkeit der Komponenten und sind damit Zeichen einer „guten“ Architektur.

- Die Vorgaben der Stile führen zu einer unterschiedlichen *Varianz der Schnittstellen* der Komponenten. Beim Pipes-Filters- und Repository-Stil ist das Datenformat bzw. das Datenbankschema das verbindende Element. Änderungen daran erfordern Änderungen an allen Komponenten. Aus Sicht des Wartungsingenieurs sollte das Format bzw. Schema daher möglichst unveränderlich sein. Beim Event-basierten Stil nimmt die Schnittstelle zur Systeminfrastruktur diese Rolle ein; die übrigen Schnittstellen der Komponenten sind von den Stilvorgaben nicht betroffen. Bei einer Schichtenarchitektur reduziert sich die Vorgabe darauf, die *Richtung des Kontrollflusses* zwischen den Komponenten einzuschränken. Dieser Stil ist damit nur ein Spezialfall des Daten abstrahierenden bzw. objektorientierten Stils, bei dem keinerlei Einschränkungen der Kommunikation bestehen.

Abbildung 4.5 illustriert die Einordnung der Architekturstile anhand ihrer Schnittstellenvarianz bzw. Einschränkungen.

Aus diesen Tendenzen lässt sich Folgendes ableiten: Je höher die Varianz der Schnittstellen ist, desto flexibler kann eine Komponente eingesetzt werden, desto schwieriger ist jedoch die Rekonstruktion eines Architekturstils, da die Stilvorgaben abnehmen, die die Analyse unterstützen können. Praktisch bedeutet das: Ist das Datenformat bekannt, können die Komponenten eines Pipes-Filters-Stil ohne größere Probleme auch im Quellcode gefunden werden. Dasselbe gilt auch für den Event-basierten Stil, denn die Schnittstelle zur Systeminfrastruktur ist eindeutig. Sowohl die Schichtenarchitektur als auch der objektorientierte Stil hingegen können eine unüberschaubare Vielfalt an Schnittstellen bieten. Hier ist eine syntaktische Unterstützung seitens der verwendeten Programmiersprachen notwendig, um sowohl die Schnittstellen als auch die Komponenten auf der Architekturebene identifizieren zu können. Andernfalls muss

mit Hilfe von Heuristiken oder anderer Verfahren der Architekturerkennung gearbeitet werden. Die Schnittstellenerkennung ist damit die zentrale Aufgabe der Architekturrekonstruktion, da mit ihrer Hilfe sowohl Komponenten identifiziert als auch Kontrollflüsse aufgedeckt werde können.

4.3.4 Sichtenbezug nach Soni et al.

Anstatt zunächst ein theoretisches Modell zu entwickeln und mit informellen Beispielen zu erläutern, nutzen Soni et al. in [SNH95] die Idee der Blickwinkel, um die bei ihren Untersuchungen vorgefundenen Architekturbeschreibungen zu kategorisieren. Sie konnten vier verschiedene Blickwinkel identifizieren, die von Systemarchitekten und Designern genutzt werden.

Durch die Untersuchung von im Einsatz befindlichen Systemen unterschiedlicher Größenordnungen wollen sie offenlegen, wie Architektur *praktisch* umgesetzt wird und welche Fragen sich bei der Entwicklung stellen. Bei Interviews mit Systemarchitekten und Designern sowie der Analyse von Design-Dokumenten und Quellcode wurde deutlich, dass für die meisten Systeme zunächst eine Unterscheidung getroffen wird in eine produktspezifische Architektur und eine Architektur für die darunter liegenden Plattform, die als „virtuelle Maschine“ für das Produkt fungiert. Weiterhin befassen sich die Strukturbeschreibungen der beiden Architekturklassen hauptsächlich mit vier Architekturkategorien:

- Die *konzeptuelle Architektur* beschreibt die wesentlichen Design-Elemente und ihre Beziehungen. Dabei handelt es sich um aus der Problemdomäne abgeleitete Elemente, die dementsprechend einen hohen Abstraktionsgrad aufweisen. Diese Architektur ist unabhängig von Implementierungsentscheidungen und betont die Interaktion der Elemente. Bei den untersuchten Systemen wurde die konzeptuelle Architektur nur implizit in der Beschreibung anderer Strukturen aufgeführt und nicht eigenständig erfasst.

- Durch die *Modul-/Verbindungsarchitektur* erfolgt eine funktionale Dekomposition in logische Subsysteme, die aus Modulen und abstrakten Einheiten bestehen. Ziel ist dabei, „Programming-in-the-Large“ zu ermöglichen und interne und externe Abhängigkeiten zu reduzieren und zu isolieren. Dies geschieht i. a. durch Verwendung eines Schichtenmodells. Die Beschreibungen dieser Kategorie reflektieren die verschiedenen Implementierungsentscheidungen, die zunächst unabhängig von der Programmiersprache bleiben. Die Elemente der konzeptuellen Architektur finden hier ihre Umsetzung in einem oder mehreren Modulen.
  Im Vergleich zu den Beschreibungen der anderen Kategorien zeigten die Modul-/Verbindungsarchitekturen der untersuchten Systeme den höchsten Detaillierungsgrad.

- Die *Ausführungsarchitektur* beschreibt das dynamische Verhalten des Systems inkl. Laufzeitelementen wie Prozessen sowie Kommunikationsmechanismen. Sie weist den einzelnen Funktionalitäten der konzeptuellen Architektur Laufzeitelemente und Ressourcen zu. Dies beinhaltet auch Entwurfsentscheidungen bzgl. Lokalisation, Migration und Replikation von Laufzeitelementen (dies betrifft sog. verteilte Systeme). Gesteuert wird die Entwicklung der Ausführungsarchitektur durch die nicht-funktionalen Anforderungen, die auch die Betriebssysteme und die Hardware betreffen. Durch den ständigen

Technologie-Wandel ist sie eher von Änderungen betroffen als die beiden bisherigen Kategorien. Beschreibungen dieser Kategorie sind ähnlich ausführlich wie die der vorigen Kategorie.

- Die *Code-Architektur* betrachtet die Organisation von Quellcode, Binaries und Bibliotheken in der Entwicklungsumgebung. Dazu gehört, je nach Programmiersprache, die Unterteilung in Module oder Verzeichnisse sowie die Einrichtung zu verwendender Werkzeuge (Konfigurationsmanagement, Build-Werkzeuge, Unit Tests, Deployment usw.). Sie liegt zwar explizit vor, wird aber nicht als Teil der Software-Architektur beschrieben, sondern hauptsächlich für die Konzeption der Konfigurationen, des Baus und der Auslieferung eingesetzt.

Die Beschreibungen mehrerer Systeme lassen sich nicht eindeutig einer Kategorie zuordnen. In mehreren Fällen wurde die Architektur auch nicht explizit formuliert, sondern lag implizit in Textform (also nicht als Diagramm o. ä.), im Code oder einer anderen Strukturbeschreibung vor.

Die Untersuchungsergebnisse von Soni et al. bestätigen sowohl die Anwendbarkeit des Sichtenkonzepts als auch die bisher vorgestellten Definitionsversuche zur Architektur. Für eine Bewertung der Wartbarkeit ist insbesondere die Unterscheidung von Produkt- und Plattformarchitektur hilfreich, da sie in Kombination mit einem Schichtansatz per se die Portabilität eines Systems unterstützt. Für einer Qualitätsbewertung können Sichten des Modul-/Verbindungsblickwinkels herangezogen werden, da sie eine Verbindung zwischen Quellcode-Elementen (Modulen) und Entwurfsentscheidungen (logische Dekomposition) herstellen. Die Sichten stehen damit auf der Architekturebene des Designs.

Neben der Bestätigung der bisherigen, theoriegetriebenen Ansätze ist die Unterscheidung von Plattform- und Produktarchitektur von nachhaltiger Relevanz, stellt sie doch einen Bezug her zu den Begriffen *Design Patterns* und *Frameworks*, die für die Architektur im Rahmen heutiger industrieller Softwareentwicklung von großer Bedeutung sind.

4.3.5 Architektur im Entwicklungsprozess nach Kruchten

Eine industrielle Softwareentwicklung erfordert umfassende Prozessmodelle, die die Arbeiten von der Anforderungsanalyse an über die Entwicklung bis hin zur Wartung mit den Erkenntnissen über Architektur verbindet. Ein Großteil der heutzutage entwickelten Software wird mit den Mitteln der Objektorientierung bewältigt, die in allen Stadien der Entwicklung Verwendung finden und dadurch nachhaltige Auswirkungen auf Qualitätseigenschaften der Architektur haben können. Darüber hinaus wurden für die Darstellung objektorientierter Strukturen verschiedene Notationsformen entwickelt, von denen sich die UML als Quasi-Standard etabliert hat.

Kruchten versucht mit seinem Modell, die Architektur als wesentlichen Bestandteil der Softwareentwicklung in einen iteratives Vorgehensmodell zu integrieren [Kru95]. Er greift die Definition des Architekturbegriffs von Perry und Wolf auf und kombiniert ihn mit dem Blickwinkel- und Sichtenkonzept von Zachman (allerdings ohne direkt darauf Bezug zu nehmen). Er nennt seinen Ansatz das „4+1“-Sichtenmodell: Mit vier Blickwinkeln können die verschiedenen Architekturaspekte beschrieben werden und ein weiterer illustriert anhand von Szenarien deren

Zusammenspiel. Auch wenn Kruchten auf andere Techniken wie E-R-Diagramme hinweist, so zielt sein Ansatz eindeutig auf eine objektorientierte Herangehensweise ab. Dies zeigt sich nicht nur bei der Definition der Blickwinkel, sondern auch bei den verwendeten Notationsformen, die später Eingang in die *Unified Modeling Language* (UML) fanden. Erklären lässt sich diese Ausrichtung durch die Mitarbeit Kruchtens bei der Firma Rational, die zu der Zeit maßgeblich an der Entwicklung objektorientierter Werkzeuge und der UML beteiligt war. Das von ihm zugrunde gelegte Architekturmodell entspricht dem von Boehm modifizierten Modell von Perry und Wolf: *Software Architecture = {Elements, Form, Rationale/Constraints}.*

Das „4+1"-Sichtenmodell besteht aus folgenden Blickwinkeln:

- Der *logische Blickwinkel* auf ein System betrachtet seine (objektorientierte) Dekomposition in *Objekte* bzw. *Objektklassen*. Das Hauptziel ist die Unterstützung der funktionalen Anforderungen. Die Objekte repräsentieren Schlüsselkonzepte der Problemdomäne und setzen die Prinzipien Abstraktion, Kapselung und Vererbung um. Neben der funktionalen Analyse dient die Dekomposition aber auch zum Auffinden von systemweiten Mechanismen und Entwurfsmustern. Kruchten definiert hier allerdings nicht, ob es sich dabei ausschließlich um Aspekte des zukünftigen Systems oder auch der Problemdomäne handelt.

- Hauptaufgabe des *Prozess-Blickwinkels* ist die Untersuchung der nicht-funktionalen Anforderungen wie Performanz, Verfügbarkeit, Nebenläufigkeit und Verteilung. Die in den logischen Sichten erfassten Hauptabstraktionen werden dazu auf eine (Arbeits-)Prozess-Architektur abgebildet. Der Beschreibung legt Kruchten ein Schichtenmodell zugrunde: Auf oberster Ebene wird das IT-System als *Netz kommunizierender Prozesse* abgebildet, die möglicherweise über mehrere Hardware-Knoten in einem Netzwerk verteilt sind. Die Prozesse agieren als Dienste, die jeweils gestartet, gestoppt und konfiguriert werden können. Jeder Prozess besteht wiederum aus einer *Gruppe von Tasks*, separaten Kontrollflüssen, die entweder als eindeutig adressierbare Haupttask oder als unterstützende, lokal arbeitende Nebentask auftreten. Haupttasks können damit vom jeweiligen Betriebssystem direkt verwaltet werden und entsprechen einem Betriebssystem-Prozess oder -Thread. Nebentasks lassen sich bspw. durch Multithreading implementieren. Die Kommunikation der Haupttasks untereinander erfolgt mittels dedizierter Protokolle wie RPC oder Event Broadcasts. Nebentasks hingegen nutzen systemnahe Techniken wie Shared Memory oder FIFOs auf Dateiebene.
  Dieser Blickwinkel eignet sich laut Kruchten auch zur Untersuchung der Umsetzungsqualität der nicht-funktionalen Anforderungen. Indem in die Rahmenstruktur des Systems lasterzeugende Dummy-Tasks integriert werden, kann man sein Verhalten unter verschiedenen Bedingungen beobachten.

- Den Entwicklern ist der *Entwicklungsblickwinkel* zugedacht. Die Sichten dieses Blickwinkels dienen zur Dekomposition des Systems in Subsysteme, die auf Anraten Kruchtens in Schichten organisiert werden sollten. Subsysteme stehen untereinander durch Import/Export-Beziehungen in Verbindung. Die Schichtung erlaubt einem Subsystem allerdings nur Zugriff auf Subsysteme, die sich in derselben oder direkt unterliegen Schicht

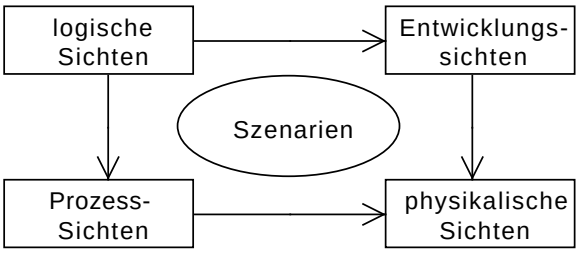

Abbildung 4.6: „4+1“-Modell (nach [Kru95, S. 2]).

befinden. Dieser Ansatz soll neben einer besseren Verwaltung auch die dezentrale Entwicklung von Subsystemen ermöglichen. Das Ziel dieses Blickwinkels besteht darin, eine möglichst einfache Entwicklung des Gesamtsystems zu gewährleisten, was u. a. die Unterstützung von Software Management und die Wiederverwendung von Bausteinen beinhaltet.

- Die in den einzelnen Prozess-Sichten vorbereitete Distribution der Arbeitsprozesse auf Tasks wird durch den *physikalischen Blickwinkel* und die Verteilung auf Hardware-Knoten vervollständigt. Konkret wird festgehalten, wo welcher Dienste bzw. welche Haupt- inkl. Nebentasks arbeiten sollen. Dabei spielen nicht-funktionale Anforderungen wie Verfügbarkeit, Skalierbarkeit, Zuverlässigkeit etc. eine Rolle, wobei im Vergleich zum Prozess-Blickwinkel auch die Möglichkeiten und Einschränkungen der einzelnen Betriebssysteme zu beachten sind.

Ein *Szenarium* ist eine Instanz eines allgemeineren Anwendungsfalls oder *Use Case*s. Es dient zur Veranschaulichung der Arbeitsweise des Systems in einer konkreten Situation. Für jedes Szenarium werden Sichten der vier Blickwinkel entworfen und ihre Beziehungen und Kommunikation aufgezeigt. Mit einer sinnvollen Szenarien-Auswahl kann sichergestellt werden, dass alle Anforderungen vollständig umgesetzt wurden.

Die einzelnen Blickwinkel bzw. ihre Sichten stehen in gegenseitiger Abhängigkeit. Ausgehend vom logischen Blickwinkel werden die Prozess- und die Entwicklungssichten beeinflusst. Diese wiederum haben Auswirkungen auf die physikalischen Sichten. In einer logischen Sicht stehen die Objekte zunächst einzeln, fokussiert auf ihre Funktionalität und Rolle in der Problemdomäne und im Systemkontext. Eine potentielle Parallelität zu anderen Objekten ist zunächst nicht von vorrangigem Interesse. In der Prozess-Sicht jedoch muss einer Nebenläufigkeit — sofern sie tatsächlich vorliegt — Rechnung getragen werden, indem für die Objekte entsprechend parallelisierbare Prozesse oder Threads angelegt werden. Die in den logischen Sichten auftauchenden Klassen werden in Form eines separat nutzbaren Moduls implementiert (die konkrete Umsetzung hängt von der verwendeten Programmiersprache ab). Zusammenhängende Klassen werden in Subsystemen zusammengefasst. Einer der von den Befürwortern der Objektorientierung propagierten Vorteile des OO-Ansatzes besteht in der Verringerung der Kluft zwischen Analyse und Entwurf; die in der Problemdomäne gefundenen Objekte sollen als Vorlage mit in den Entwurf einfließen. Bei Untersuchungen hat Kruchten aber festgestellt, dass diese Kluft mit zunehmendem Projektumfang größer wird. Domänenklassen müssen dem-

nach selbst wieder in mehrere Klassen dekomponiert werden. Die Organisation der Prozesse bzw. Threads ergeben bestimmte Anforderungen, die bei der Zuordnung auf die verfügbare Hardware beachtet werden müssen. Dazu gehören auch Teststellungen und Deployment.

Interessanterweise geht Kruchten nicht auf den Übergang zwischen Entwicklungssicht und physikalischer Sicht ein, obwohl erfahrungsgemäß auch hier wechselseitige Abhängigkeiten bestehen. Eine Organisation in Subsysteme und Schichten muss für eine mögliche Verteilung entsprechende Schnittstellen und Kommunikationsprotokolle vorsehen. Je nach Umfang der Subsysteme können auch Anforderungen an die Leistungsfähigkeit der Hardware bestehen (bspw. benötigt ein DBMS weniger CPU-Leistung, aber viel Hauptspeicher).

Kruchten empfiehlt ausdrücklich, seinen Ansatz nach Bedarf anzupassen. Zusätzlich schlägt er einen iterativen Prozess bei der Entwicklung der einzelnen Sichten vor, der von den jeweils kritischsten Szenarien ausgeht. Dieser Versuch, die Architekturentwicklung in ein Entwicklungsprozessmodell aufzunehmen, kann als einer der Grundsteine der heutigen agilen Entwicklung und als Vorläufer des *Unified Process* (UP) angesehen werden. Das ebenfalls von ihm vorgeschlagene *Software Architecture Document* entspricht in weiten Teilen dem gleichnamigen Artefakt im UP (zum UP s. bspw. [Lar04]). Der Zusammenhang zwischen dem „4+1"-Ansatz und dem Architekturmodell von Perry und Wolf lässt sich allerdings nur auf konzeptioneller Ebene herstellen, d. h. es gibt Elemente, die durch ihre Anordnung eine bestimmte Form hervorbringen und deren Auswahl erläutert wird. Die jeweiligen Herangehensweisen (strukturierte bzw. objektorientierte Analyse) unterscheiden sich grundlegend voneinander und führen zu einem anderen Verständnis der Sichten. Die Datensicht von Perry und Wolf entspricht in gewisser Weise der logischen Sicht, die aber durch die Fokussierung auf die Objektorientierung die relevanten Aspekte aus der Problemdomäne betrachtet und nicht zwangsläufig auf zu verarbeitende konkrete Datentypen und Daten abzielt. Im Gegenteil: Objekte in der Implementierung unterscheiden sich in vielen Belangen von denen in der Analyse. Außerdem kombiniert diese Sicht Daten mit zugehörigen Aktionen, wie es in der Objektorientierung üblich ist. Die Prozess-Sicht des „4+1"-Modells hingegen ähnelt der Sicht von Perry und Wolf, denn sie betrachtet die nicht-funktionalen Aspekte über mehrere Abstraktionsschichten hinweg bis zur Implementierung einzelner Tasks. Die Verbindungssicht, die Daten und ihre Verarbeitung in Beziehung setzen soll, ist bei einem objektorientierten Vorgehen durch das Paradigma selbst implizit vorgegeben. Im weitesten Sinne handelt es sich also um eine „Übersetzung" des Ansatzes von Perry und Wolf in die Welt der Objektorientierung.

Laut Kruchten wurde das Modell erfolgreich bei mehreren großen Projekten eingesetzt [Kru95, S. 14]. Die heutige Verbreitung des Unified Process und der Objektorientierung untermauert die Leistungsfähigkeit seines Ansatzes. Andere Modelle wie bspw. von Soni, Nord und Hofmeister können nach seiner Auffassung auf das „4+1"-Modell abgebildet werden und stellen nur eine Spezialisierung dar. Ein Vergleich der Modelle zeigt tatsächlich eine weitgehende Übereinstimmungen der Blickwinkel, obwohl sich die Vorgehensweise von Soni et al. bei der Entwicklung ihres Ansatzes drastisch von der Kruchtens unterscheidet.

Kruchten geht nicht auf die Qualität von Software-Architektur ein, allerdings ist die Verwendung von Szenarien als anforderungsinduzierter Einstieg in die Architekturbeschreibung ein neuer Ansatz. Er spielt in dem im nachfolgenden Abschnitt erläuterten Buch ([BCK98]) eine maßgebliche Rolle bei der Bewertung einzelner Sichten. Die Beschreibung des Entwicklungsblickwinkels bleibt schwammig und reduziert die zu empfehlenden Programming-in-the-

Large-Ansätze auf das Schichtenmodell. Trotzdem ist der Beitrag ein wichtiger Meilenstein, konsolidiert er doch die verschiedenen Ansätze zur Architekturbeschreibung in ein umfassendes Modell und integriert es in einen iterativen Entwicklungsprozess.

4.3.6 Architecture Business Cycle nach Bass et al.

In ihrem Buch „Software Architecture in Practice“ ([BCK98]) stellen Bass, Clemens und Kazman die Architektur ins Zentrum der Systementwicklung. Dabei präsentieren sie sie nicht nur als theoretisches Artefakt des Entwurfs, sondern betrachten ihre verschiedenen Ausprägungen ganz konkret anhand mehrerer Fallstudien. Darüber hinaus formulieren sie Regeln für die Evaluierung von Architekturstilen und erläutern fundamentale Grundoperationen, die im Hinblick auf bestimmte Eigenschaften beim Entwurf von neuen Architekturstilen eingesetzt werden können. Zusammen mit einer eigenen Architekturanalysemethode markiert dieses Buch einen vorläufigen Endpunkt bei der Suche nach einer geeigneten Architekturmetapher und der Bewertung von Architekturqualitäten; nahezu alle späteren Arbeiten beziehen sich darauf.

Bass et al. greifen sowohl das Sichtenkonzept und den Stilbegriff als auch die Komponenten-Konnektoren-Metapher auf, legen jedoch einen anderen Schwerpunkt. Ihre zentrale Frage lautet: „Was sind die Beziehungen der Architektur eines Systems zu der Umgebung, in der das System entwickelt und eingesetzt werden soll?“ Als Antwort definieren sie den *Architecture Business Cycle* (ABC), der die Wechselwirkungen der technischen, geschäftlichen und sozialen Einflüsse mit der Architektur eines Softwaresystems begleiten und koordinieren soll. Neben den funktionalen und nicht-funktionalen Anforderungen spiegeln demnach auch organisatorische Faktoren eine Rolle bei der Entwicklung. Ihre Definition des Architekturbegriffs beinhaltet das Sichtenkonzept sowie den Komponentenbegriff, verfeinert allerdings den Schnittstellengedanken:

> „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.“ [BCK98, S. 6]

Die Architektur eines Systems besteht aus mehr als einer Strukturbeschreibung (darin folgen sie Zachman). Weiterhin definiert eine Architektur eine oder mehrere Komponenten. Eine Komponente abstrahiert von allen Details, die nichts mit der Verwendung von und durch andere Komponenten zu tun haben. Sämtliche Informationen, die nicht Teil der Schnittstelle — der für andere Bestandteile „sichtbaren“ Eigenschaften der Komponente — sind, werden ausgelassen. Damit ist auch das Verhalten jeder Komponente Teil der Architektur. Aus der Definition folgt ferner, dass *jedes* System eine Architektur besitzt, auch wenn es sich nur um ein Ein-Komponenten-System handelt.

Die Definition macht keine Aussage über die Güte der Architekturqualität. Bass et al. schreiben dazu: „There is no such thing as an inherently good or bad architecture.“ [BCK98, S. 17], zeigen aber auch einen Weg hin zu einer Evaluierung auf: „…architectures can in fact be evaluated (…) but only in the context of specific goals“. Eine Bewertung kann nur zusammen mit konkreten Zielvorstellungen hinsichtlich einer „guten“ Architektur vorgenommen werden. Sie

weisen jedoch darauf hin, dass es eine Reihe von allgemein gültigen „Faustregeln“ für die Entwicklung „guter“ Architekturen und deren Strukturen gibt, die sie „process recommendations“ (für den Entwicklungsprozess) und „product (structural) recommendations“ (für das Produkt) nennen ([BCK98, S. 18ff]).

Neben diesen Regeln greifen Bass et al. weitere schon bekannte Konzepte auf. Dazu gehört der Begriff des Architekturstils, den sie im Unterschied zu Perry und Wolf bzw. Shaw und Garlan unter Verwendung einer Typisierung der Komponenten und eines Verweises auf Verwendungsmuster definieren:

> „An architectural style is a description of component types and pattern of their runtime control and/or data transfer.“ [BCK98, S. 24]

Die Typisierung ermöglicht eine einfachere Spezifikation der Komponenten, die in einem Stil vorkommen können. Wie bei Shaw und Garlan ist auch bei dieser Definition die topologische Anordnung der Komponenten via Konnektoren konstituierendes Merkmal eines Architekturstils. Die Konnektoren regeln dabei die Kommunikation, aber auch die Koordination bzw. Kooperation der Komponenten zur Laufzeit. Nach Ansicht von Bass et al. gehört zu einem Stil auch eine Reihe semantischer Randbedingungen, die die Arbeitsweise der Komponenten und die Auswirkungen der Kommunikation einschränken können (bspw. darf ein Repository nicht eigenständig seine Daten modifizieren). Wie Entwurfsmuster (s. dazu [Gam+96]) sind auch Architekturstile vorgefertigte Segmente, die mit ihren jeweiligen Eigenschaften für eine bestimmte Anwendung modifiziert werden können [BCK98, S. 93]. Jedes Muster steht damit für eine Abfolge von Entwurfsentscheidungen, die zu bestimmten Eigenschaften und Qualitäten des Musters geführt haben.

Bass et al. setzen den von ihnen verwendeten Begriff „structure“ mit „view“ gleich und stellen auf diese Weise die Verbindung zu dem schon bekannten Blickwinkel-Konzept her. Ohne Anspruch auf Vollständigkeit führen sie die folgenden Blickwinkel auf, wobei die Liste viele der schon von anderen Autoren aufgeführten Blickwinkel umfasst:

- Die *Modulstruktur* beschreibt die Aufgabenzuteilung, mit der die bei der Problemlösung anfallende Arbeit auf Module verteilt wird. Module stehen über *is-a-submodule-of*-Relation in Beziehung zu anderen Modulen, sie definiert als eine vertikale Anordnung in Sinne einer Hierarchie. Einem Modul werden neben seiner Schnittstelle auch Code, Testpläne etc. zugeordnet.
- Durch Abstraktion der funktionalen Anforderungen ergeben sich die Einheiten der *konzeptuellen/logischen Struktur*, die über *shares-data-with*-Relationen verbunden sein können. Auf diese Weise können Referenzmodelle abgebildet werden.
- Mit Hilfe der *Prozess-/Koordinationsstruktur*, die orthogonal zur Modul- und Konzeptstruktur steht, lassen sich die dynamischen Aspekte eines laufenden Systems beschreiben. Die Einheiten entsprechen Prozessen oder Threads, die über verschiedene Synchronisationsbeziehungen interagieren.
- Die Abbildung der Software auf die Hardware wird durch die *physikalische Struktur* beschrieben, die besonders für parallele und verteilte Systeme interessant ist. Verbunden sind Prozessoren oder andere Hardware-Einheiten via *communicates-with*-Relationen.

- Die *Verwendungsstruktur* ergänzt die Modulstruktur, indem die dort aufgeführten Module bzw. ihre Bestandteile — Prozeduren — über *assumes-the-correct-presence-of* -Relationen horizontal angeordnet werden. Dies gilt auch für die *Aufrufstruktur*, die die Aufrufe zwischen den (Sub-)Prozeduren mittels *calls-*/*invokes*-Relationen beschreiben.

- *Daten-* und *Kontrollfluss* geschieht zwischen Programmen und Modulen — beim Kontrollfluss zusätzlich zwischen Systemzuständen —, die über *may-send-data-to-* bzw. *becomes-active-after*-Relationen kommunizieren. Durch die Beschreibung des Datenflusses kann die Umsetzung von Anforderungen an das System verfolgt werden. Der Kontrollfluss ist u. a. relevant für das funktionale Verhalten des Systems.

- Handelt es sich um ein objektorientiertes System, existiert außerdem eine *Klassenstruktur*. Klassen bzw. Objekte können über *is-instance-of* bzw. *inherits-from* in Beziehung stehen.

Wie Kruchten weisen auch Bass et al. darauf hin, dass zwar jeder Blickwinkel seine Existenzberechtigung und eine gewisse Eigenständigkeit hat, er aber nicht unabhängig von den Ausgestaltungen der anderen Blickwinkel ist.

Durch ihre Spezifikation von Komponenten definiert eine Architektur auch die Umstände, unter denen eine Änderung erfolgt und bestimmt auch die betroffenen Bestandteile. Dabei können die von einer Änderung ausgelösten Modifikationen unterschiedliche Ausmaße annehmen: Eine Änderung schlägt sich in der Modifikation nur einer Komponente nieder, es sind mehrere Komponenten betroffen oder es sind drastische Anpassungen wie bspw. ein Wechsel des zugrunde liegenden Architekturstils notwendig.

Änderbarkeit gehört zu den nicht zur Laufzeit beobachtbaren Qualitätseigenschaften eines Systems. Sie wird wesentlich durch die Architektur bestimmt:

> „Modifiability is largely a function of the location of change. Making a widespread change to the system is more costly then making a change to a single component, all other things being equal.“ [BCK98, S. 82]

Eine „gute“ Software-Architektur sollte vor dem Hintergrund der Änderbarkeit so angelegt sein, dass Modifikationen auf möglichst wenige Komponenten und Konnektoren beschränkt bleiben. Dies kann bei der Entwicklung in zu einem gewissen Grad berücksichtigt werden, indem man die Menge der wahrscheinlich zukünftig anfallenden Änderungen abschätzt, bspw. durch einen Vergleich mit ähnlichen Systemen, und Ungenauigkeiten oder Mehrdeutigkeiten in den Anforderungen untersucht. Die Präzisierung selbiger führt u. U. zu Modifikationen des Systems.

Neben den systemeigenen (nicht-funktionalen) und geschäftsbezogenen (funktionalen) Qualitäten beschreiben die Autoren zusätzlich drei *architektureigene* Qualitäten:

1. Unter *konzeptueller Integrität* verstehen sie ein allgemeines Konzept oder Prinzip, das dem Systementwurf innewohnt und systemweit umgesetzt wird. Sie verweisen auf ein Zitat von Brooks, der diese Qualität außerordentlich wichtig einschätzt und sogar behauptet, dass eine System ohne konzeptuelle Integrität mangelhaft ist [BCK98, S. 88].

2. *Korrektheit und Vollständigkeit* sind erforderlich, um alle Systemanforderungen mit den zur Laufzeit verfügbaren Ressourcen zu erfüllen.

3. *Baubarkeit* gewährleistet eine Architektur dann, wenn mit ihrer Hilfe das System innerhalb des vorgegebenen Zeit- und Kostenrahmens fertiggestellt werden kann und gleichzeitig so flexibel bleibt, dass die im Laufe der weiteren Entwicklung anfallenden Änderungen gut umgesetzt werden können.

Die Analyse von Architektur vor der eigentlichen Entwicklung ist für Bass et al. eine absolute Notwendigkeit, um die Anforderungen erfüllen zu können. Sie muss auch während der Entwicklung fortgesetzt werden, da sie die Voraussetzung für eine effektive Wartung ist. Insbesondere vor dem Hintergrund der Langlebigkeit von Systemen und den damit verbundenen Anforderungsänderungen und Anpassungen an neue Umgebungen zeigt sich, dass zu den wichtigsten Qualitäten die Änderbarkeit und die Portierbarkeit gehören. Neben der Umsetzung der funktionalen Anforderungen machen sie eine „gute“ Architektur aus.

Eine Analyse stützt sich zunächst auf die vorhandenen Beschreibungen: Sichten der oben erläuterten Blickwinkel, vorhandene Anforderungsdokumente, Gespräche mit den verschiedenen Interessengruppen etc. Da verschiedene Qualitätsattribute nicht unabhängig voneinander sind, sondern ihre jeweilige Bedeutung erst im Kontext einer Änderung gewinnen, wählen Bass et al. als Ausgangspunkte sog. Änderungsszenarien. Diese werden im nächsten Schritt aus den bis dahin gewonnenen Informationen entwickelt und anschließend der vorhandenen bzw. geplanten Architektur gegenüber gestellt. Sie unterscheiden dabei *direkte* von *indirekten* Szenarien: Direkte Szenarien können ohne Architekturänderungen vom System durchgeführt werden, das System funktioniert also wie erwartet. Indirekte Szenarien hingegen erfordern eine oder mehrere Änderungen und zeigen dadurch die diesbezüglichen Qualitäten der Architektur auf.

Ändern zwei Szenarien dieselbe Komponente (oder eine zusammengehörende Gruppe von Komponenten), haben sie eine Bezug zueinander. Dieser Bezug ist eng verbunden mit Metriken der strukturellen Komplexität, Kopplung und Kohäsion. Handelt es sich um inhaltlich verwandte Szenarien, deutet dies auf eine hohe Kohäsion und geringe strukturelle Komplexität der Komponenten hin. Sind die Szenarien jedoch fundamental unterschiedlich, liegt wahrscheinlich nur eine geringe Kohäsion und demzufolge eine höhere strukturelle Komplexität vor. Wenn also viele nicht zusammen gehörende Szenarien tendenziell dieselben Komponenten betreffen, handelt es sich um eine eher schlecht zu modifizierende Architektur.

Bei Legacy-Systemen gestaltet sich eine Analyse weitaus schwieriger. Wenn eine Dokumentation der Architektur vorliegt, ist zunächst fraglich, inwieweit sie mit der tatsächlichen Architektur des im Betrieb befindlichen Systems übereinstimmt — die Konsistenz von Dokumentation und Quellcode muss geprüft werden. U. U. sind die ursprünglichen Entwickler nicht mehr verfügbar und können bei einer notwendigen Nachdokumentation nicht helfen. Letztendlich bleibt dann nur eine Rekonstruktion auf Basis des Quellcodes.

Die Definition des Architekturbegriffs und seine Implikationen formulieren Bass et al. im Wesentlichen entlang den Definitionen der bisher vorgestellten Arbeiten. Sie legen jedoch durch ihre Betonung der Schnittstellen einen anderen Schwerpunkt und stellen die Kommunikation der Komponenten in den Mittelpunkt. Die aufgeführten Konzepte wie Sichten/Blickwinkel

und Architekturstile entsprechend weitgehend den schon bekannten Ansätzen. Wie Shaw und Garlan versuchen auch sie, Qualitätsaussagen zu einzelnen Architekturstilen zu machen, gehen dabei allerdings einen Schritt weiter. Mit ihren „product recommendations“ beschreiben sie sprach- und systemunabhängige Richtlinien, denen eine Architektur genügen sollte, und definieren Grundoperationen, die für die Entwicklung von Architekturstilen hinsichtlich bestimmter Qualitäten relevant sind. Sie weisen weiter darauf hin, dass nicht alle Qualitätsattribute durch die Ausgestaltung der Architektur beeinflusst werden. Darüber hinaus identifizieren sie architektureigene Qualitäten, die gänzlich unabhängig von den funktionalen und nichtfunktionalen Anforderungen existieren, aber eine Rolle für das Architekturverstehen spielen. Sie betonen, dass Qualitäten ihre Bedeutung ausschließlich in Gegenwart einer konkreten Anforderung erhalten — zum Beispiel angesichts eines Änderungsszenariums — und für sich genommen wertlos sind.

Vor dem Hintergrund der Rekonstruktion und Bewertung von Architekturen sind besonders die Folgerungen aus den Definitionen und Aussagen sowie die eingestreuten Zitate interessant. Bass et al. zitieren Gamma et al. dahingehend, dass es unmöglich sei, aus der Laufzeit-Struktur eines objektorientierten Programms auf die Code-Strukturen zu schließen, die zur Compile-Zeit vorliegen; Klassen und -hierarchien zeigten sich nicht in dem Netzwerk kommunizierender Objekte, das zur Laufzeit entsteht und sich permanent ändert (s. [BCK98, S. 45]). Dieses Zitat legt nahe, dass eine Architekturanalyse zur Laufzeit nicht sinnvoll ist, wenn es um Qualitäten wie Änderbarkeit oder Wiederverwendung geht. Diese gehören laut Bass et al. zu denjenigen Qualitätsattributen, die gerade dann *nicht* erkennbar sind. Die bei der Vorstellung ihrer „Software Architecture Analysis Method“ geäußerten Vermutung bzgl. einer Korrespondenz bestimmter Metriken zu verwandten Szenarien erlaubt außerdem möglicherweise Rückschlüsse auf die Güte der Modellierung sowie einen Überblick darüber, welche Komponenten „zusammengehören“. Inwieweit architektureigene Qualitäten im Rahmen einer Architekturrekonstruktion überhaupt erfasst und bewertet werden können, bleibt unklar, da die Autoren konkrete Definitionen zur konzeptuellen Integrität oder zur Baubarkeit schuldig bleiben.

4.3.7 Architektur in der aktuellen Softwareentwicklung

Auch wenn mit Garlan und Shaw bzw. Bass et al. die Blickwinkel für die Entwickler im Vergleich zu den anderen aufgeführten Arbeiten mehr in den Vordergrund stellen, so wird doch nur bedingt auf die technischen Anforderungen einer industriellen Software-Fertigung und die damit verbundenen Trends in der Softwareentwicklung der letzten Jahre eingegangen. Dazu gehören insbesondere die dezentrale Entwicklung von Systemen und die zunehmende Verwendung vorgefertigter bzw. gekaufter Bausteine. Der Einsatz solcher Bausteine soll verschiedene Qualitäten eines Systems wie Fehlerfreiheit und Performance bis zu einem gewissen Grad planbar machen und anschließend den Aufwand und die Kosten insbesondere der Wartung nachhaltig reduzieren.

Komponentenorientierte Entwicklung

Szyperski geht davon aus , dass der Trend hin zu einer großteilig assemblierenden Entwicklung Zeichen jeder gereiften Ingenieursdisziplin ist und sich ein Markt von Anbietern und Kompo-

nenten entwickeln wird, der analog zu anderen Märkten gute und preiswerte Bausteine hervorbringt [Szy99, S. xiii, 6f]. Dies würde für die Entwicklung Software-intensiver Systeme zur Konsequenz haben, dass im Rahmen einer industriellen Fertigung der Integration von Bausteinen und den damit verbundenen Anforderungen an die Architektur mehr Beachtung geschenkt werden muss. Er hat nicht das Ziel, eine weitere Form von Architekturbeschreibung zu präsentieren, sondern er untersucht, welche Voraussetzungen für eine solche *komponentenorientierte Entwicklung* gegeben sein müssen und wie sich dies auf die Entwicklung und die Systemstrukturierung auswirkt. Die sich daraus ergebenden neuen Maßstäbe für die Qualitätsbewertung der Architektur sind insbesondere für Entwickler interessant, weshalb die Komponentenorientierung hier im Kontext der Betrachtung der Architekturqualität eine Rolle spielt.

Den Schlüssel zur Komponentenorientierung sieht Szyperski in der Modularität. Sie führt zu einem Umdenken nicht nur bei der Implementierung, sondern beeinflusst auch die frühen Entwicklungsphasen: Die (Wieder-)Verwendung bestehender Bausteine erfordert eine Modularität nicht nur bei der Implementierung, sondern auch in den Bereichen Anforderungen, Architektur und Entwurf. Dies führt schon vor der ersten Code-Zeile zu einer entsprechenden Kapselung der funktionalen Anforderungen und fördert die Adaptivität, Skalierung und Wartbarkeit des Systems. Allerdings geht er in seinem Buch auf gerade diese Schritte nicht weiter ein und konzentriert sich ausschließlich auf die Behandlung von Implementierungsaspekten. Änderungen an einer Komponente wirken sich im Vergleich zu monolithischen Umsetzungen nur begrenzt aus. Dies entspricht den Empfehlung von Bass et al., dass Änderungen möglichst „lokal“ gehalten werden sollen; s. dazu auch Abschnitt 4.3.6 [Szy99, S. xiii ff]. Die Kapselung führt weiterhin dazu, dass die bei monolithischen Systemen üblichen umfangreichen Update-Zyklen entfallen, da durch die Entkopplung des Systems einzelne Bausteine unabhängig voneinander ausgetauscht werden können. Damit eröffnet sich ein passabler Weg, nicht nur Fehler bereinigen und Optimierungen ausliefern zu können, sondern auch das Funktionsportfolio entsprechend den Kundenwünschen aufzustellen. Er erhält so ein auf seine Bedürfnisse zugeschnittenes Produkt. Nicht zuletzt entspricht die Komponentenorientierung einer konsequenten Weiterentwicklung des Teile-und-Herrsche-Ansatzes: Wo in der Objektorientierung oder strukturierten Programmierung Objekte bzw. Prozeduren/Funktionen in Paketen oder Namensräumen zusammengefasst werden und aus Architektursicht kein weiteres Abstraktionsniveau bieten, implementiert eine Komponente bestimmte Schnittstellen und besitzt eine genau umrissene Funktionalität. Dadurch erschließt sich ein zusätzliches Abstraktionsniveau, auf dem mit Komponenten auf ähnliche Weise hantiert werden kann wie dies mit Objekten bzw. Prozeduren geschieht.

Anders als bei den bisher aufgeführten Methoden zur Architekturbeschreibung ist eine Komponente bei Szyperski kein Abstraktum, sondern ein Artefakt mit bestimmten, über die Eignung als Komponente entscheidenden Eigenschaften und konkretem Verwendungsziel [Szy99, S. 3]. Sie haben darüber hinaus auch eine direkte Bedeutung für den Kunden [Szy99, S. 12]. Hier zeigt sich die Verbindung zu den funktionalen Anforderungen, die ein System mit Hilfe dieser Komponente erfüllen soll; seine Arbeitsweise und sein Aufbau wird kommunizierbar. Zwar ist die Komponentenorientierung kein originär objektorientiertes Konzept, allerdings wird sie mittlerweile umfassend in diesem Bereich verwendet, um die dortigen Defizite bei der Beschreibung der Architektur großer Systeme auszugleichen [Szy99, S. 144] Auf dieser Basis definiert er seinen Komponentenbegriff wie folgt:

> „A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.“[Szy99, S. 34]

Im Unterschied zu einem Objekt eines objektorientierten Systems ist eine Komponente in sich geschlossen und kann separat ausgeliefert und verwendet werden [Szy99, S. 30]. Die daraus resultierende Unabhängigkeit der Komponenten voneinander und vom Systemkontext schließt verschiedene Abstraktionen wie bspw. Typdeklarationen, Makros oder C++-Templates für die Implementierung aus (sie alle sind nur zum Zeitpunkt des Kompilierens relevant). Prozeduren, Klassen oder Module, aber auch ganze Anwendungen hingegen können in einer Komponente zusammengefasst werden. Voraussetzung ist, dass sie in einer aufrufbaren, „binären“ Form vorliegen und ihre Funktionalität über Schnittstellen verfügbar machen. Im Gegensatz zu einem Objekt, das nach seiner Erzeugung immer einen Zustand und eine eigene Identität besitzt, weisen Komponenten zudem keinen persistenten Zustand auf, d. h. Kopien einer Komponente sind folglich auf der Integrationsebene nicht voneinander zu unterscheiden. Objekte hingegen sind anhand ihrer Speicheradresse immer eindeutig identifizierbar und können nur darüber Methoden anderer Objekte aufrufen.

Ein weiterer Begriff, der häufig im Kontext von Systembeschreibungen verwendet wird, ist *Modul*. Szyperski versteht darunter die Zusammenfassung abstrakter Datentypen und bzw. Klassen zu Kompilationseinheiten. Sie haben allerdings keine Bedeutung auf funktionaler oder Architekturebene und dienen hauptsächlich zur Organisation des Codes auf Dateiebene, bspw. für verteiltes Kompilieren. Damit sind sie nur für die Entwickler von Interesse [Szy99, S. 32].

Auf der Systemebene (der *Systemarchitektur* nach Szyperskis eigenem Verständnis) haben sich erfolgreiche Systeme durch einige wenige Prinzipien hervorgetan: *Schichtung* (strikt, nichtstrikt) und *Hierarchien* bzw. *Heterarchien*. Die Schichtarchitektur entspricht dem schon bei Shaw und Garlan aufgeführten Schichten-Architekturstil (s. Abschnitt 4.3.3). Abweichungen wie bspw. eine nicht-strikte Schichtung können möglicherweise dazu führen, dass die Systemstruktur nicht mehr dargestellt werden kann. Diese Problematik ist insofern wichtig, da laut Szyperski Architekturbeschreibungen immer bestrebt sein sollten, ein Gleichgewicht zwischen Verständlichkeit, Funktionalität und Ökonomie zu suchen. Die Darstellbarkeit bietet also einen ersten Test, ob eine Architektur (schon) so komplex ist, dass sie nicht mehr durch ein einfaches Diagramm beschrieben werden kann. Auch Coad und Yourdon weisen darauf hin, dass eine Beschränkung notwendig ist [CY90, S. 107]. Szyperski untermauert dieses Bewertungskriterium mit einem Verweis auf Beck:

> „It may seem strange to base architectural decisions on such a vague criteria of human perceptibility. (…) However, they all fade to gray when the perceived complexity of an architecture prevents understandability and thus prevents effective teaching, maintenance, and evolution.“ [Szy99, S. 142]

Modularität ist für Szyperski *das* Schlüsselkonzept zur Entwicklung von Software-Architekturen, um die Wiederverwendung von Bausteinen zu ermöglichen und eine langfristige Wartung zu gewährleisten. Die Wiederverwendung hat dabei nicht nur positive Auswirkungen auf die Projektlaufzeit, -kosten und -qualität, sondern führt auch zu einem Wissenstransfer in Form von Entwurfsmustern und Frameworks, der wiederum die der Architekturqualität

zugute kommt. Sie durchzieht als Kerngedanke den gesamten Entwicklungsprozess. Auf der technischen Seite profitieren Wartung und Weiterentwicklung vom begrenzten Einfluss von Änderungen — den Impacts —, die sowohl beim Entwurf als auch bei der Umsetzung eine Rolle spielen. In Kombination mit einer Beschränkung der Komplexität fördert sie effektiv das Systemverstehen. Allerdings erlaubt erst die Beschränkung der Komplexität, wie sie bspw. durch eine Begrenzung der Anzahl von Beschreibungselementen erfolgen kann, dass Entwickler und Wartungsingenieure ein Verständnis für das System und die ihm zugrunde liegenden Strukturen entwickeln können, um eine Architekturdrift bzw. -erosion zu vermeiden. Eine ausgefeilten Strukturierung ist wertlos, wenn das Verstehen durch die wahrgenommene Komplexität der Architektur behindert wird [Szy99, S. 142]. Eine „gute“ Architektur ist für Szyperski demnach hoch modular, dabei in ihrer Komplexität beschränkt und folgt einer gut dokumentierten zentralen Strukturierungsidee, wie sie bspw. von einem Framework geliefert wird.

Dennoch verschweigt er nicht die Nachteile, die ein derartig modularer Ansatz mit sich bringt. Zunächst weist er darauf hin, dass die herkömmlichen Objektmodelle objektorientierter Sprachen nicht wirklich als Vorlage für ein Komponentenmodell taugen; es müssen also alternative Ansätze gefunden werden. Weiterhin birgt gerade die oft als Vorteil hervorgehobene Möglichkeit der Vererbung das Risiko, eine vermeintlich entkoppelte Bausteine durch die Verwendung einer gemeinsamen Basisklasse doch wieder eng aneinander zu binden. Szyperski beschreibt dies als semantisches und syntaktisches „fragile base class problem“ [Szy99, S. 103]. Auch die Alternative zur Vererbung, die Objektkomposition mittels Forwarding bzw. Delegation, ist nicht immer geeignet, um sämtliche Probleme zu lösen [Szy99, S. 117]. Letztendlich besteht zumindest auf der Ebene der Parameterübergabe die Notwendigkeit, dass ein kleinster gemeinsamer Nenner einerseits auf syntaktischer Ebene, andererseits auf semantischer Ebene gefunden wird. Beginnend bei den verwendeten Sprachelementen gibt es bis hin zu systemweit genutzten sog. Werkzeug-Klassen also immer eine gewisse Verbindung der Bausteine untereinander. Auf der Quellcode-Ebene muss für die Bewertung der Wartbarkeit sowohl die Vererbung als auch die Objektkomposition untersucht werden, um zumindest eine Vererbung über Komponentengrenzen hinweg explizit zu machen bzw. sie auszuschließen. Zur sinnvollen Verwendung der Delegation verweist Szyperski auf Gamma et al., die den Einsatz der Delegation nur dann empfehlen, wenn sie sich vereinfachend auf das Design auswirkt [Szy99, S. 122].

QUASAR

Bei der Implementierung betrieblicher Informationssysteme hat sich eine Schichtarchitektur als Standardarchitekturstil durchgesetzt. Sie besteht aus der Benutzerschnittstelle, dem Anwendungskern mit der fachlichen Logik, und der Datenverwaltung, die meist in Form einer Datenbank vorliegt. Dazu kommen weitere Bestandteile etwa für die Fehlerbehandlung. Wie schon in Abschnitt 4.3.3 beschrieben, ist eine Kommunikation zwischen den Schichten auf die Schichtschnittstellen begrenzt. Nach Ansicht von Siedersleben und Denert ist die Schichtarchitektur in der Literatur jedoch „notorisch unterspezifiziert“, da weder die Schichtzuständigkeiten klar definiert werden noch eine in der Praxis weitgehende Unabhängigkeit der Schichten voneinander erreicht wird [SD00, S. 247]. Im Rahmen ihrer Verfeinerung der Schichtarchitektur namens QUASAR (*Qua*litäts-*S*oftware*ar*chitektur) beschreiben sie ein im Vergleich zu den bisher aufgeführten Bewertungsverfahren recht pragmatischen Ansatz. Als Ausgangs-

punkt für ihre Entwicklung wählen sie eine Schlussfolgerung, die sie der Arbeit von Lai und Weiss entnehmen (s. [LW99]): Die Qualität einer Architektur wird maßgeblich bestimmt durch ihre Robustheit gegenüber Variabilität. Anders ausgedrückt: Welche Komponenten müssen bei welcher Änderung modifiziert werden? Zunächst teilen sie alle Änderungen in drei Kategorien ein:

- *R-Änderungen* betreffen die externe Repräsentation fachlicher Objekte. Dazu gehören Datenbank-Schemata oder Maskenlayouts. Fachliche Funktionen werden nicht berührt.
- *T-Änderungen* betreffen die Technik, also API-Anpassungen oder ein Wechsel der Technologie (bspw. von MFC nach Qt).
- *A-Änderungen* betreffen die Anwendung selbst, es werden also fachliche Funktionen bearbeitet.

Diese drei Kategorien hängen in bestimmter Weise zusammen: R-Änderungen sind unabhängig von T- und A-Änderungen, jedoch implizieren sowohl T- als auch A-Änderungen in der Regel R-Änderungen. Wie Bass et al. verwenden Siedersleben und Denert Änderungsszenarien, deren Auswirkungen auf die Architektur sie für eine Qualitätsbewertung heranziehen. Ihre Bewertung der Standardarchitektur fällt denkbar schlecht aus: Die Dialog- und Datenverwaltungsschicht enthalten den Großteil des Systemcodes und ist primär durch Technik bestimmt. Das hat zur Folge, dass bei T-Änderungen mindestens eine der beiden Schichten umgebaut werden muss. R-Änderungen betreffen sogar das gesamte System. Damit ist eine Wiederverwendung kaum mehr möglich.

Um eine gute, sprich: klare und modulare Architektur, zu erhalten, müssen ihrer Meinung nach zwei Ziele verfolgt werden:

1. Der Einfluss der verschiedenen Änderungstypen soll klar definiert werden.
2. Die von R- und/oder T-Änderungen betroffene Code-Menge soll minimiert werden.

Als Leitlinie dient ihnen das Prinzip der Trennung von Zuständigkeiten (Separation of Concerns), nach dem sich jedes System ausschließlich mit der Anwendung befassen sollte, für die es gebaut wurde. Formal gehört jedes Stück Software zu genau einer der folgenden Kategorien:

- *0-Software* ist unabhängig von Anwendung und Technik.
- *A-Software* wird bestimmt durch Anwendung, ist aber unabhängig von Technik.
- *T-Software* hingegen ist unabhängig von Anwendung, jedoch bestimmt durch Technik.
- *AT-Software* wird durch beides bestimmt.

0-Software wie bspw. die C++-STL ist hervorragend für die Wiederverwendung geeignet (üblicherweise auch mit diesem Ziel entworfen), setzt aber meist nur abstrakte Konzepte wie Container oder Sortieralgorithmen um. Damit ist sie allein nicht von Nutzen. A-Software kann immer dann wiederverwendet werden, wenn die vorhandene Anwendungslogik ganz oder

teilweise benötigt wird. Technisch kann dies auf verschiedene Art und Weise erfolgen, z. B. über die Einbindung von DLLs oder durch CORBA-Komponenten. T-Software kann (in Teilen) dann wiederverwendet werden, wenn ein neues System dieselben technischen Komponenten einsetzt; Beispiele sind JDBC oder MFC. AT-Software hingegen ist allgemein schwer wartbar und kaum zu ändern. Eine Ausnahme bildet nur R-Software, die eine schwache Form von AT-Software ist. Sie befasst sich hauptsächlich mit der Transformation von fachlichen Objekten und externen Repräsentationen. Diese lassen sich allerdings gut generieren und sind demzufolge besser zu warten.

Eine gute Architektur zeichnet sich also durch folgende Eigenschaften aus:

1. Die Software-Bausteine lassen sich eindeutig kategorisieren.
2. Es sollte möglichst keine AT-Software vorhanden sein.
3. Der Anteil an T-Software sollte minimal gehalten werden.

Ihr Ziel — eine Standardarchitektur mit klar definierten Änderungstypen und nur wenig R-/T-Änderungen — erreichen Siedersleben und Denert u. a. durch die Verwendung von Fassaden und Adaptern, die die konkreten technischen APIs von fachlichem Anwendungscode entkoppeln und eine Serialisierung und damit einen Transfer von Objekten ermöglichen (s. dazu [Sie04]). Sie weisen allerdings darauf hin, dass für eine derartig weitreichende Entkopplung ein nicht zu unterschätzender Aufwand betrieben werden muss, der sich möglicherweise nicht immer lohnt.

Der Ansatz, Software-Bausteine anhand von Änderungsauswirkungen zu bewerten, ist nicht neu, wird er doch schon bei Bass et al. mit ihren Änderungsszenarien eingesetzt, um die Kohäsion und Kopplung von Komponenten zu bestimmen. Siedersleben und Denert nutzen ihn, um eine Kategorisierung der Änderungen und der Software-Bausteine vorzunehmen. Mit ihrer Einteilung bestätigen sie die von Soni et al. gefundene Trennung von Plattform- und Produktarchitektur (s. S. 43). Die Technik und die 0-Software wäre demnach der Plattform zuzurechnen und die A-Software dem Produkt. Allerdings abstrahieren sie von den tatsächlich vorliegenden Abhängigkeiten wie Vererbungsketten über Komponentengrenzen. Abhängigkeiten der Schichten können ihrer Meinung nach durch die Implementierung zusätzlicher Wrapper reduziert werden. Dabei stellt sich aber die Frage, wie groß der Aufwand für die Implementierung solcher Wrapper ist und inwiefern damit tatsächlich Abhängigkeiten reduziert oder beseitigt werden können. Für eine solide Bewertung ist dieses Abstraktionsniveau zu hoch und es wäre ratsam, weitere Detailuntersuchungen vorzunehmen, die bspw. die Befolgung der Law of Demeter betreffen (s. dazu auch [LH89]).

4.4 Folgerungen

Im Vergleich zur Software-Architektur hat die Vermessung von Software sowohl methodisch als auch praktisch einen hohen Reifegrad erreicht. Obwohl es noch Detailprobleme wie bspw. bei der Definition von Schwellenwerten oder der Interpretation von Messwerten gibt (s. S. 21), wird sie auf breiter Ebene bei der Bewertung eingesetzt. Es fehlt jedoch an Maßen zur

Bewertung von architekturrelevanten Strukturen, die dazu oft nur sehr feingranulare Strukturen erfassen. Die Metriken von Lanza und Marinescu versuchen, auch „höhere“ Strukturen zu erfassen, beide warnen aber gleichzeitig vor einem allzu blauäugigen Einsatz: Es ist umfangreiches Expertenwissen notwendig, um (a) die Detektoren auszusuchen, und (b) die so gefundenen Symptome zu interpretieren und für die Entwicklung nutzbar zu machen. Populär sind die McCabe-Metriken, die Halstead-Maße, die OO-Metriken nach Chidamber und Kemerer und das MOOD-Set von Abreu und Carapuça, wobei die Eignung einer Metrik durch das verwendete Programmierparadigma beeinflusst wird. McCabe-Metriken, die vorwiegend bei der strukturierten Programmierung eingesetzt werden, finden, allerdings mit Einschränkungen, auch bei der Bewertung von objektorientierten Softwaresystemen Anwendung. Die Ansätze zum Entwurf und zur Analyse von Software-Architektur hingegen gleichen zur Zeit eher einem Konglomerat aus verschiedenen Begriffsdefinitionen, Strukturierungsvorschlägen und Faustregeln als einer konsolidierten Vorgehensweise samt Werkzeugkasten.

Der ISO-Standard 9126 für die Bewertung der Software-Produktqualität zeigt durch die Unterscheidung von innerer und äußerer Qualität die Zielgruppen auf. Für Entwickler sind hauptsächlich innere Qualitätsmerkmale relevant, besonders die Wartbarkeit (zerlegt in Analysierbarkeit, Änderbarkeit, Stabilität (=seiteneffektfrei) und Testbarkeit) und Portabilität (bestehend aus Anpassungsfähigkeit (die im Wesentlichen der Änderbarkeit entspricht), Installierbarkeit, Koexistenz und Ersetzbarkeit). Die aufgeführten Kenngrößen für diese internen Qualitätsmerkmale sind allerdings unzureichend. Dies gilt auch für die Vorschläge zur Ermittlung von Werten dieser Kenngrößen. Der Standard ist damit zunächst nicht praxistauglich und kann nur als Ausgangspunkt für konkrete Qualitätsvorstellungen dienen.

Dass Software-Architektur ein mittlerweile anerkannt wichtiger Aspekt bei der Entwicklung und Wartung ist, zeigt sich durch den Versuch, diesbezüglich einen Standard vorzulegen. ISO 1471-2000 subsumiert die gängigsten Definitionen und Konzepte aus diesem Bereich. Dazu zählt das Sichtenkonzept von Zachman und die damit verbundene Zielgruppenorientierung, die Kategorisierung der Systembestandteile in Komponenten und Konnektoren, sowie Architekturmodelle, die den Sichten zugrunde liegen und auf bestimmte Strukturen fokussieren. Wie ISO 9126 verbleibt auch dieser Standard auf einer abstrakten Ebene und gibt keine konkreten Empfehlungen zur Verwendung von Sichten bzw. Perspektiven und Rekonstruktion von architekturrelevanten Artefakten aus Quellcode. Ebenso wenig werden Architekturstile als zentrales Strukturierungselement näher behandelt oder Hinweise zur Qualitätsbewertung von Architekturen gegeben. Stattdessen finden sich nur Verweise auf die üblichen Quellen wie Bass et al. und Zachman und es wird deutlich, *dass* auf dieser Ebene wichtige Designentscheidungen getroffen werden müssen, nicht aber, *wie* dies erfolgen sollte. Die besonders für Entwickler und Wartungsingenieure wichtige Verbindung von Architektur und Quellcode-Strukturen fehlt völlig. Wie ISO 9126 kann auch dieser Standard nur als Ausgangspunkt für die Entwicklung eigener, konkreter Architekturvorstellungen dienen.

Architekturerosion und -drift sind die wesentlichen Risiken, denen ein System im Laufe seines Lebens durch Wartungsarbeiten ausgesetzt ist. Die Entwicklung des Architekturbegriffs und die Methoden zur Untersuchung haben eine Reihe von Merkmalen identifiziert, die aus Sicht der Entwickler eine „gute“ Software-Architektur aufweisen sollte. Das Prinzip des Separation of Concerns ist dabei Grundlage vieler Konzepte; es liegt sowohl dem Sichtenkonzept zugrunde als auch vielen Entwurfsempfehlungen bei der Programmierung (z. B. Funktions-

umfang von Schnittstellen, Kapselung). Komponenten und Konnektoren finden sich sich auch auf Quellcode-Ebene, allerdings bieten die meisten Mainstream-Programmiersprachen keine direkte Unterstützung geschweige denn eine Definition des Begriffs „Komponente“, obschon durch den SOA-Ansatz bestimmte ESB-Schnittstellenfunktionalitäten eine Komponente als solche kennzeichnen. Die Anordnung und Klassifizierung von Bausteinen und die damit verbundene Umsetzung von Architekturstilen ist jedoch nur durch eigene Programmierarbeit möglich; die Einhaltung von Randbedingungen, wie sie Kruchten fordert, ist auch dann nicht umfassend überprüfbar. Die wichtigsten im Code sichtbaren Merkmale, die mehr oder weniger direkt mit der Wartbarkeit in Verbindung gebracht werden können, sind die Kopplung und die Kohäsion von Komponenten. Durch sie wird eine diesbezügliche Qualitätsbewertung bis zu einem gewissen Grad möglich, die sich aber ausschließlich angesichts konkreter Änderungsanforderungen zeigt und dies auch nur in statischen Sichten. Weiterhin wird der Dokumentation als Artefakt einer Architekturbeschreibung ein höherer Stellenwert zuteil, auch wenn sie sich einer automatischen Erfassung und Auswertung durch die Verwendung natürlicher Sprache weitgehend entzieht (Anstrengungen in dieser Richtung werden aber auch unternommen, s. bspw. [Rup07]). Auf ein nicht zu unterschätzendes Problem gehen die meisten Autoren nicht ein: Die Synchronisation und Konsistenz der Sichten und der Sichtbegriff selbst (s. [Hil99; Hil00]).

Die Richtlinien von Bass et al., die Code Smells von Fowler und die Vorgaben Szyperskis zur Komponentenorientierung sind Versuche, sich aus unterschiedlichen Perspektiven an eine gute Architektur im Sinne von „wartbar“ anzunähern, die teilweise erfolgreich sind. Die vielfach geforderte Entkopplung von Bausteinen funktioniert nur bis zu einer bestimmte Modellierungsebene/Code-Ebene, allerspätestens bei der Parameterübergabe muss Übereinstimmung herrschen. Ob dann ausschließlich auf die zur jeweiligen Programmiersprache gehörenden Basistypen zurückgegriffen oder doch mit Fachklassen gearbeitet wird, ist entscheidend für den Kopplungsgrad. Das gilt auch für die Vererbung, insbesondere über Klassengrenzen hinweg. Die Delegation als Alternative dazu ist laut Gamma nur dann sinnvoll, wenn sie Dinge vereinfacht, was seiner Ansicht nach nur bei entsprechend hoch formalisierten Strukturen wie Entwurfsmustern der Fall ist. Die beim SOA-Ansatz verfolgte vollständige Entkopplung nimmt dem Entwickler zudem ein wichtiges Werkzeug: Durch die Typüberprüfung des Compilers können viele Fehler aufgedeckt werden, was auch einer der Gründe für die Verbreitung statisch getypter Sprachen ist. Dazu kommt, dass aus der losen Bindung zur Laufzeit umgekehrt zustandslose Services resultieren. Eine Zustandsverwaltung muss damit ggf. durch andere Bestandteile des Systems erfolgen (s. dazu bspw. [ST07, S. 245]).

Alle Ansätze steuern auf das eigentliche Ziel hin: Das Systemverstehen. Der Entwickler muss ausgehend vom Quellcode in möglichst kurzer Zeit ein korrektes mentales Modell des Systems aufbauen und dabei die Architektur(en), die Regeln des Architekturstils und die daran geknüpften Bedingungen erfassen. Eigenschaften wie Kopplung, Kapselung und Modularität beeinflussen diesen Prozess. Eine gute, sprich: verständliche, Architektur ist hoch modular, besitzt eine geringe Komplexität mit geringer Kopplung zwischen und hoher Kohäsion innerhalb der Komponenten und folgt einer zentraler Strukturierungsidee, die zudem noch gut dokumentiert ist. Ausgehend von diesem Ziel stellen sich die Frage: Wie können aus Quellcode architekturrelevante Artefakte ermittelt werden, die eine derartige Bewertung ermöglichen? Zuvor muss allerdings die Frage beantwortet werden, wie ein Entwickler Architektur versteht, welchen „Wartungsblickwinkel“ er also einnimmt.

Bei diesen Ansätzen darf jedoch der menschliche Faktor nicht vergessen werden, der bei der Entwicklung eines Systems und später bei der Wartung immer eine Rolle spielt. Inwieweit tatsächlich Ergebnisse aus Quellcode ermittelt werden können, die eindeutige Aussagen zu bestimmten qualitative Eigenschaften erlauben, ist möglicherweise nicht endgültig zu beantworten. Wallmüller weist mit Bezug auf Pirsig darauf hin, dass Software-Produkte zu einem erheblichen Teil das Ergebnis intuitiver und kreativer Arbeit sind. Er unterscheidet messbare Qualität in Form von Kenngrößen von der ebenfalls als qualitativ relevant anzusehenden kreativen Tätigkeit, die sich einer Messung entzieht. Damit erhält der Qualitätsbegriff neben einer logisch/rationalen Seite zusätzlich eine subjektive, vom direkten Wahrnehmen und Empfinden geprägte Seite, die sich durch Begriffe wie Originalität, Einfachheit und Mächtigkeit beschreiben lässt [Wal01, S. 26]. Pirsig stellt eine direkte Verbindung zwischen dem Erfahrungsschatz und dem subjektiven Qualitätsbegriff her: „Die Namen, die Gestalten und Formen, die wir der Qualität geben, sind außerdem auch in den apriorischen Bildern begründet, die sich in unserem Gedächtnis angesammelt haben. Wir sind beständig bestrebt, im Ereignis der Qualität Analogien zu unseren früheren Erfahrungen zu finden.“ [Pir06, S. 263]. Daraus ergibt sich als Anforderung an ein Analysesystem, dass der Entwickler die Schwellenwerte an seine Bedürfnisse bzw. Fähigkeiten und Erfahrung anpassen können muss. Weiterhin folgt daraus, dass sämtliche Bewertungen einen unbekannten subjektiven Anteil haben. Eine objektive Bewertung kann damit nur eingeschränkt vorgenommen werden. Alle auf diese Weise ermittelten Kennzahlen können also nur als Indizien dienen, die hinreichend für nähere Untersuchungen sind, aber nicht notwendigerweise eine schlechte Qualität bezeugen.

Im nächsten Kapitel wird auf Basis dieser Erkenntnisse ein Qualitätsmodell und ein zugehöriges Softwaremodell aufgebaut, die beide anschließend in Form konkreter Methoden der Informationsgewinnung aus Quellcode dienen.

5 Ein Qualitätsmodell zur Beurteilung von Quellcode

> Complexity kills. It sucks the life out of developers, it makes products difficult to plan, build and test, it introduces security challenges, and it causes end-user and administrator frustration.
>
> *(Ray Ozzie, Mail „The Internet Services Disruption“)*

Die im vorigen Kapitel gewonnenen Erkenntnisse hinsichtlich der Ermittlung und Bewertung von Qualitätsmerkmalen auf Basis von Quellcode bilden die Grundlage für die Entwicklung eines Qualitäts- und des zugehörigen Software-Modells sowie der benötigten Extraktionsmethoden. Zunächst werden die Grundlagen des Messens im Software-Engineering erläutert und anschließend das Qualitätsmodell entworfen, dass im Wesentlichen die für Entwickler relevanten Eigenschaften *Wartbarkeit* und *Portabilität* in eine Taxonomie von Einzelfaktoren überführt. Bevor die jeweiligen Indikatoren definiert werden können, ist zunächst ein Softwaremodell notwendig, das die benötigten Code-Strukturen abbildet, die für die Ermittlung der Kenngrößen erforderlich sind.

Ein Ziel dieser Untersuchung ist die Bewertung einer Fallstudie. Es handelt sich dabei um ein in der Programmiersprache Java geschriebenes System zur Verwaltung von Veranstaltungs- und Prüfungsdaten. Java ist ähnlich wie C++ eine objektorientierte Programmiersprache, die auch klassische imperative Elemente besitzt. Beide Modelle müssen daher sowohl die Möglichkeiten der strukturierten Programmierung als auch die Objektorientierung im Allgemeinen und ihre Umsetzung in der Sprache Java im Speziellen berücksichtigen.

5.1 Ausgangssituation

Die Qualität von Quellcode zeigt viele Aspekte, die für eine Bewertung relevant sein können; Abbildung 5.1 auf S. 62 versucht eine Zuordnung der bisher in diesem Kontext gefallenen Begriffe. Die Hauptaufgabe eines Qualitätsmodells besteht darin, eine für den jeweiligen Anwendungskontext passende Kombination dieser Aspekte anzubieten.

Das vorige Kapitel kam zu folgenden Ergebnissen:

1. Die langfristig für Entwickler relevanten Eigenschaften eines Systems sind die *Wartbarkeit* und *Portabilität* des Quellcodes. Beide Eigenschaften werden im Wesentlichen geprägt durch die *Kopplung*, die *Kohäsion* und *Komplexität* der einzelnen Bausteine sowie

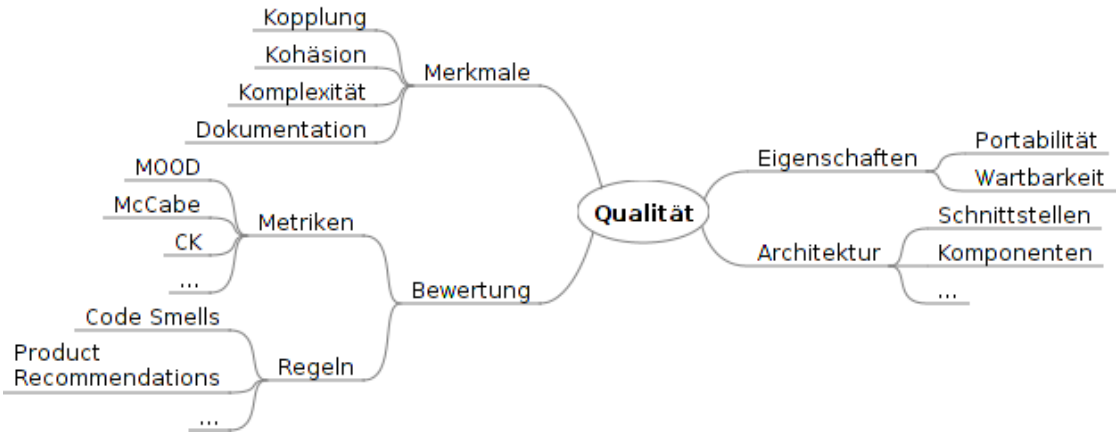

Abbildung 5.1: Aspekte der Qualität von Quellcode.

durch ihre zugehörige *Dokumentation*.
Die automatisierte Analyse von Dokumentation ist im Vergleich zur Quellcode-Analyse ungleich schwieriger und noch nicht soweit fortgeschritten (s. dazu S. 59 unten). Daher wird sie im nachfolgend entwickelten Qualitätsmodell nicht als Merkmal berücksichtigt.

2. Die Merkmale lassen sich nur aus *statischen Perspektiven* bzw. deren *Sichten* auf das System (=den Quellcode) ermitteln. Ein Software-Modell beschreibt demnach nur statische Sichten auf das System, bspw. in Form von Klassen- und Komponentendiagrammen.

3. Das Abstraktionsniveau des Software-Modells einer Programmiersprache ist einerseits zu feingranular, andererseits nicht ausreichend, um auf der Architekturebene Komponenten (wie auch immer dieser Begriff zu definieren ist) zu erheben und zu bewerten. Entsprechend muss hier das Software-Modell angepasst bzw. erweitert werden.

4. Neben den „klassischen“ Metriken sind auch eher weich definierte „Faustregeln“ wie die Product Recommendations von Bass et al. oder Qualitätsdefekte wie die Code Smells von Fowler wichtige Qualitätsmerkmale, die zur Bewertung des Systems herangezogen werden können.

5.2 Grundlagen

Der Sinn und Zweck eines Qualitätsmodells wurde schon in Abschnitt 2.2.2 erläutert: Das Verständnis von „Qualität“ soll durch eine Taxonomie von Einzelfaktoren präzisiert und messbar werden, um von den Messergebnissen auf die jeweiligen Qualitätseigenschaften und damit auf die Gesamtqualität eines Systems schließen zu können. Bei dieser Operationalisierung des Qualitätsbegriffs stößt man zunächst auf zwei bekannte Probleme:

1. Es gibt keine allgemein anerkannte Definition des Begriffs „Software-Qualität“.

2. Wie „gut“ ein System ist, zeigt sich erst, wenn es produktiv eingesetzt wird [CM78, S. 133].

Dem zweiten Problem kann nur begegnet werden, indem schon im Vorfeld die Qualität des Systems ermittelt und die weitere Entwicklung so gesteuert wird, dass sich im Produktivbetrieb die gewünschte Qualität zeigt. Das erste Problem lässt sich durch eine möglichst klare und widerspruchsfreie Definition des Qualitätsbegriffs zumindest produktspezifisch bzw. produktfamilienspezifisch lösen. Allerdings stellt sich bei jeder Definition die Frage, inwieweit die ermittelten Ergebnisse tatsächlich dazu beitragen, die Entwicklung hinsichtlich der Qualität positiv zu beeinflussen. Eine Möglichkeit bestünde im Hinzuziehen von Experten, die eine entsprechende Bewertung der Ergebnisse vornehmen könnten, was jedoch mangels Personal und auch aus Kostengründen nicht immer möglich oder gewünscht ist. Die automatische Ermittlung von Qualitätseigenschaften ist damit die einzige Alternative, die auf Basis einer begrifflichen Zerlegung des Qualitätsbegriffs den Zusammenhang von Merkmalen und Eigenschaften beleuchten kann.

In den nachfolgenden Abschnitten wird zunächst auf die Grundlagen der Begriffszerlegung eingegangen und anschließend erläutert, wie ein Qualitätsmodell entwickelt werden kann.

5.2.1 Zerlegung des Qualitätsbegriffs

Der ISO-Standard 9126-1 bietet einen geeigneten Ausgangspunkt, um mit der Entwicklung eines eigenen Qualitätsmodells zu beginnen. Die vom Standard verwendete Zerlegungssystematik basiert auf der *Factor-Criteria-Metrics*-Methode (FCM) von Cavano und McCall (s. [CM78]). Auf der obersten Ebene finden sich die Beschreibungen der jeweiligen Produktqualitäten (die *Qualitätsfaktoren*), die auf der nächsttieferen Ebene in ein oder mehrere Attribute der Software aufgelöst werden (*Qualitätskriterien* und *Unterkriterien*). Diese wiederum können durch Metriken zum Quellcode in Beziehung gesetzt und bspw. in Zahlform erfasst werden, welche auf Programmierempfehlungen aus der Fachliteratur basieren. Abbildung 5.2 auf S. 64 zeigt die drei Ebenen und ihre jeweilige Bedeutung.

Die im FCM beschriebene Zerlegung besitzt nur drei Ebenen, die Erfahrung zeigt aber, dass zusätzliche Ebenen sinnvoll sind. Entsprechend wurde die Zerlegungssystematik des ISO-Standards entworfen. In [10d] wird bspw. eine weitere Schicht mit Entwurfsregeln vorgeschlagen, die Angaben zur Merkmalsausprägung macht (z. B. „hohe Kohäsion“) und diese in fünf Schritten mit den Kenngrößen verbindet (s. Abbildung 5.3). Die Abbildung veranschaulicht zudem, wie die abgeleiteten Ziele durch Metriken mit dem Quellcode in Beziehung gesetzt werden können.

Eine derartig angelegte Definition des Qualitätsbegriffs bietet mehrere Vorteile. Zunächst werden die Zusammenhänge zwischen hoch abstrakten Qualitätsfaktoren und konkreten Softwarestrukturen transparent und plausibel gemacht. Entwickler sind somit in der Lage einzuschätzen, welche Änderungen möglicherweise Auswirkungen auf bestimmte Faktoren haben. Das wiederum ermöglicht bei negativen Auswirkungen ein sofortiges Gegensteuern in der Entwicklung, indem auf Basis der Zerlegung Handlungsempfehlungen gegeben werden.

Nicht immer ist es jedoch möglich, eine präzise Konkretisierung zu erreichen. Setzt man bspw. die Eigenschaft *Wartbarkeit* als Qualitätsziel fest, wird unter „guter Wartbarkeit“ übli-

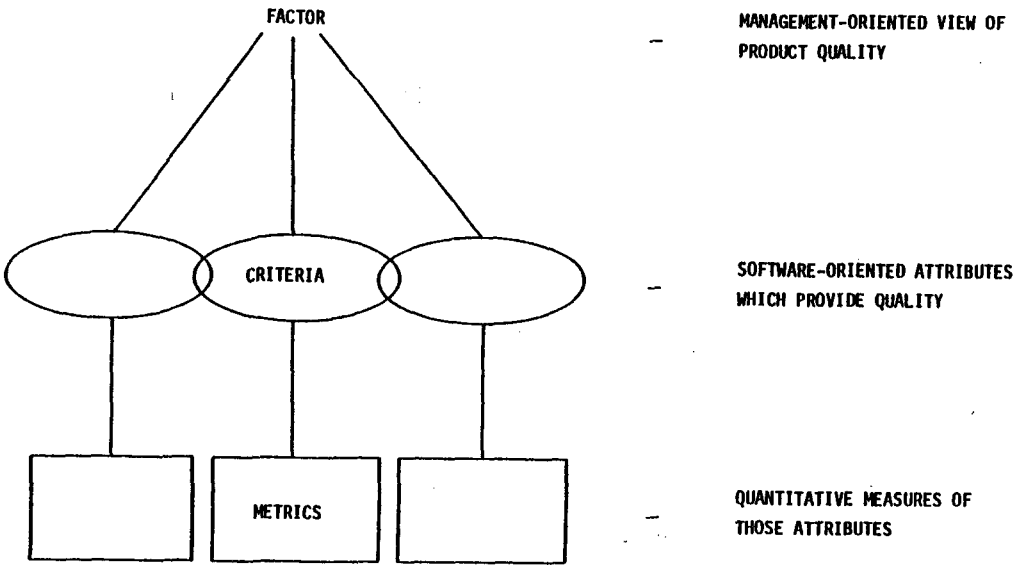

Figure 1. Software Quality Framework

Abbildung 5.2: Aufbau der FCM-Zerlegung ([CM78, S. 135]).

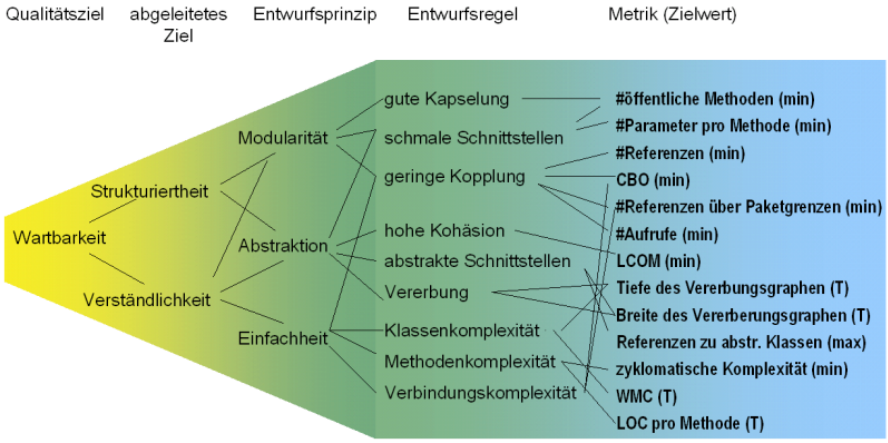

Abbildung 5.3: Qualitätsmodell aus [10d] (Portal des ViSEK-Projekts des Fraunhofer IESE, gefördert vom BMBF).

cherweise verstanden, dass in kürzester Zeit unter Belegung weniger Ressourcen Änderungen am System vorgenommen werden können. Das wiederum bedeutet, dass eine Bewertung nur angesichts einer konkreten Änderungsanforderung (s. dazu auch S. 48) durch die Messung der benötigten Zeit und des Ressourcenverbrauchs möglich ist. Eine prädiktive Bewertung *während* der Entwicklungszeit ist so nicht möglich. Stattdessen greift man auf Merkmale zurück, denen man unabhängig von einer bestimmten Anforderung einen mehr oder weniger direkten Einfluss auf die Qualitätseigenschaft zuspricht. Für die Wartbarkeit wird bspw. die Kopplung von Software-Bestandteilen herangezogen. Solche Ersatzmerkmale nennt man *Prädiktoren* (s. [10e]). Ihnen liegt immer eine Hypothese zugrunde, die ihren Einfluss auf die im Modell aufgeführten Qualitätseigenschaften beschreibt.

5.2.2 Vorgehensweise nach Wallmüller

Welche Schritte müssen unternommen werden, um ein eigenes Qualitätsmodell zu entwickeln? Wallmüller beschreibt dazu eine mögliche Vorgehensweise (s. [Wal01, S. 28]):

1. Definieren der Messziele
2. Ableiten der Messaufgaben aus den Messzielen
3. Bestimmung der Messobjekte
4. Festlegen der Messgröße und Messeinheit
5. Zuordnung der Messmethoden/-werkzeuge zu den Messobjekten/-größen
6. Ermitteln der Messwerte
7. Interpretation der Messwerte

Er betont, dass bekannt sein muss, *warum* gemessen wird, denn die Interpretation der Messwerte basiert immer auf einer bestimmten Hypothese (s. o.). Entsprechend dieser Vorgehensweise stellt sich zunächst die Frage, welche Qualitätsfaktoren durch eine Messung bewertet werden sollen. Anschließend müssen Hypothesen aufgestellt werden, welche Merkmale — bzw. an ihrer Stelle die Prädiktoren — als Konkretisierung in Frage kommen. Handelt es sich im die Bewertung von systeminternen Qualitätseigenschaften, spielt auch die verwendete Programmiersprache eine Rolle bei der Auswahl der Prädiktoren. An dieser Stelle ist ein Softwaremodell notwendig, welches einerseits die Menge der möglichen Prädiktoren beschreibt, die die eingesetzte Programmiersprache bietet, und das andererseits auch Messobjekte anbietet. Dabei muss das Softwaremodell nicht auf die Möglichkeiten der Programmiersprache beschränkt bleiben, sondern kann darüber hinaus auch weitere Abstraktionsstufen abbilden, die bspw. die Architektur des Systems betreffen. Zusammen mit den aus dem Qualitätsmodell abgeleiteten Kriterien definiert es dann den in Abschnitt 4.2 angekündigten Wartungsblickwinkel, in dem die dafür relevanten Aspekte des Quellcodes herausgearbeitet werden. Das Softwaremodell beeinflusst auch die Messmethoden und Werkzeuge, die für die Ermittlung der Messwerte in Frage kommen. Zusammen mit den Hypothesen werden die Messwerte schließlich interpretiert.

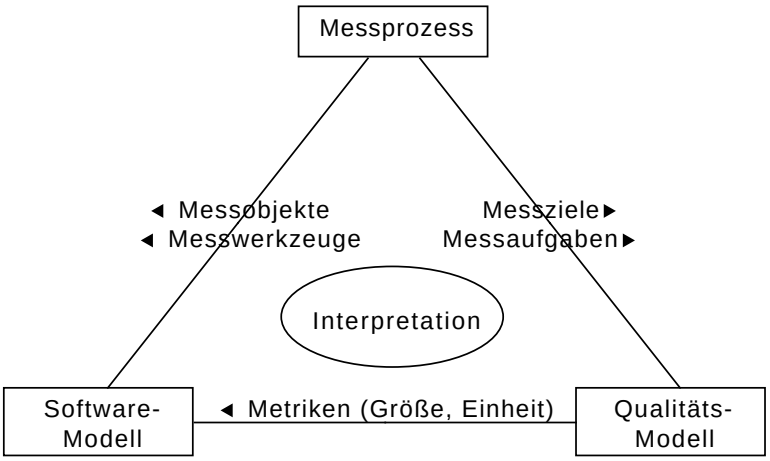

Abbildung 5.4: Messprozess bei der Vermessung von Quellcode (eigene Darstellung).

Der Messprozess im Software Engineering verbindet damit das Qualitätsmodell mit einem zugehörigen Softwaremodell. Die Interpretation der Messergebnisse als abschließender Schritt wird dabei sowohl vom Aufbau des Qualitätsmodells als auch vom verwendeten Softwaremodell maßgeblich bestimmt (s. Abb. 5.4; die Pfeile zeigen von ihrem Definitionskontext auf den Anwendungskontext).

5.3 Messziel

Der erste Schritt bei der Entwicklung ist die Bestimmung des Messziels, von dem sich die einzelnen Messaufgaben ableiten. Als Ergebnis der Untersuchungen in den Kapiteln 2 bis 4 wird folgendes Messziel definiert:

Definition 1. Das *Messziel* umfasst die Untersuchung und Bewertung der *internen Qualität* eines Softwaresystems in Form der Qualitätseigenschaften *Wartbarkeit* und *Portabilität*.

Der ISO-Standard 9126-1 untergliedert die Wartbarkeitseigenschaft weiter in die Untereigenschaften *Analysierbarkeit*, *Änderbarkeit*, *Stabilität*, *Testbarkeit* und *Standardkonformität*, die Portabilitätseigenschaft in die Untereigenschaften *Anpassungsfähigkeit*, *Installierbarkeit*, *Koexistenz*, *Ersetzbarkeit* und ebenfalls *Standardkonformität*. Da im ISO-Standard kaum auf etwaige Qualitätsmerkmale eingegangen wird, die diese Unterbegriffe beeinflussen oder konkretisieren, müssen sie für eine Integration in ein Qualitätsmodell zunächst näher betrachtet werden. Dabei stellt sich die Frage, inwieweit ein Zusammenhang mit den Qualitätsmerkmalen *Kopplung*, *Kohäsion* und *Komplexität* besteht, denen eine sie prägende Eigenschaft zugeschrieben wird.

Die jeweiligen Untereigenschaften werden nun näher untersucht mit dem Ziel, Hypothesen für die Bestimmung konkreter Messaufgaben zu formulieren. Die Aufgaben müssen sich dabei auf messbare Eigenschaften von Strukturen im Quellcode bzw. dem zugehörigen Softwaremodell beziehen, um eine Zuordnung der Messmethoden zu ermöglichen.

5.3.1 Wartbarkeit

Analysierbarkeit

Analysierbarkeit ermöglicht durch Untersuchungen des Quellcodes und weiterer Informationsquellen die Identifizierung von Fehlern oder Defiziten (die auch als noch nicht vorhandene Funktionalität zu verstehen sind) und das Eingrenzen von (korrigierenden) Änderungen.

Diese recht unscharfe Definition bedarf einer näheren Erläuterung und der Interpretation. Zunächst schlägt der ISO-Standard 9126-3 für die Bewertung der Analysierbarkeit das Zählen von Logging-fähigen und eingebauten Diagnose-Funktionen. Dies deckt sich nicht mit der Definition, die sich im zugehörigen Standard ISO 9126-1 befindet. Sie sieht ganz klar eine Analyse des Quellcodes und zugehöriger Daten und Dokumente für die Bewertung vor. Durch Logging und Diagnose-Funktionen kann zwar das Systemverhalten zur Laufzeit bis zu einem definierten Grad beobachtet werden, nicht aber der Aufbau des Quellcodes und keinesfalls die Dokumentation. Insofern ist diese Vorgabe nicht geeignet und es wird für die Definition des Qualitätsmodells der im ISO-Standard 9126-1 festgelegte Analysierbarkeitsbegriff gewählt, der auf das Systemverstehen durch den Entwickler abzielt.

Bennicke und Rust bieten mit [BR04] einen umfassenden Überblick über Theorien und Untersuchungen zum Systemverstehen. Das Hauptproblem fassen sie folgendermaßen zusammen: „Während es bei der Programmentwicklung darum geht, Antworten auf diese Fragen [„Was wird entwickelt?“ (Spezifikation) und „Wie wird es entwickelt?“ (Umsetzung)] zu entwickeln oder zumindest zu präzisieren, geht es beim Programmverstehen darum, schon einmal gefundene Antworten zu regenerieren“ [BR04, S. 17]. Das Verstehen versucht eine Rekonstruktion des Entwicklungsprozesses („Forward Engineering“), der von einem potentiell anderen Entwickler geführt wurde, und wird auch „Reverse Engineering“ oder „Backward Engineering“ genannt. Ziel ist also, aus den verfügbaren Informationsquellen die Historie der Entwurfsentscheidungen abzuleiten, die zu diesem System geführt haben. Als Informationsquellen benennen sie den Quellcode, die interne Dokumentation in Form von Quellcode-Kommentaren, zusätzliche externe Dokumentation sowie das Wissen des Entwicklers über die Problemdomäne. Der Analyseprozess sollte dabei analog zum Entwurfsprozess organisiert werden, da so die Ergebnisse des Entwurfsprozesses besser nachzuvollziehen sind [BR04, S. 16].

Neben der für die Analyse aufgewendeten Ressourcen müsste auch die Fachkenntnis des Entwicklers als Faktor mit in die Bewertung der Analysierbarkeit einfließen, da sie sich u. U. massiv auf das Systemverstehen auswirken kann. Ein Beispiel: Ein Entwickler, dem Entwurfsmuster fremd sind, könnte angesichts einer Zuständigkeitskette (s. [Gam+96, S. 410]) die Frage stellen, ob die Implementierung nicht vereinfacht werden könnte.

Die Rekonstruktion von Entwurfsentscheidungen wird durch die beim Forward Engineering vorgenommenen Transformationen erschwert (vgl. [BR04, S. 8]):

- Dekomposition: Die Dekompositionsstrategie ist nicht festgelegt, weshalb eine Spezifikation durch verschiedene Dekompositionen umgesetzt werden kann.
- Spezialisierung: Ähnlichkeitsbeziehungen, wie sie in der Problemdomäne zu finden sind, können auf verschiedene Arten im Quellcode realisiert werden oder sie sind dort nicht mehr explizit aufzufinden.

- Wahl des Algorithmus: Eine abstrakte Funktion wie das Sortieren kann durch verschiedene konkrete Algorithmen (Quicksort, Bubblesort) realisiert werden.
- Verzahnung: Mehrere abstrakte Konzepte können in einer einzigen Struktur zusammengefasst werden.
- Delokalisierung: Ein Konzept wird nicht immer an nur einer Stelle umgesetzt, sondern über den Code verteilt.
- Optimierung: Leistungssteigerungen werden oft durch Anpassungen der Implementierungen erreicht, die allerdings das zugrunde liegende Konzept häufig verwischen.

Den Transformationen liegt die Idee der schrittweisen Verfeinerung zugrunde; dieser Ansatz impliziert die Entwicklung einer Hierarchie von Elementen. Laut Courtois ist dieses Konzept essentiell für das Verstehen großer Systeme, denn durch die unterschiedliche Kopplungsstärken innerhalb und zwischen den Subsystemen können sie im Rahmen der Analyse als eigenständig betrachtet werden [Cou85, S. 597]. Meyer weist allerdings darauf hin, dass objektorientiertes Design gerade *nicht* dem üblichen uniformen Dekompositionsgedanken folgt [Mey97, S. 111ff].

Die Informationssammlung bei hypothesengesteuerten Vorgehensweisen ist immer auch von ökonomischen Zielen gelenkt: Es wird in der Regel diejenige Strategie gewählt, die die kognitiven Ressourcen möglichst gering belastet. Eine Informationssammlung ausschließlich auf Basis eines mentalen Modells, also ohne direkte Bezugnahme auf den Quellcode, ist allerdings erst dann möglich, wenn das Modell schon ausreichend elaboriert ist. Bis dahin verbessert eine klare Strukturierung des Quellcodes das Verstehen erheblich. Experten im Programmverstehen setzen dabei das sog. „chunking“ ein, um ihre mentale Kapazitätsgrenze des Kurzzeitgedächtnisses nicht zu überschreiten. Als „chunking“ bezeichnet Miller die Fähigkeit, Informationen zu größeren Einheiten zusammenzufassen (sog. „chunks of information“) [Mil56]. Der ökonomische Aspekt wird allerdings schon daran erkennbar, welche Code-Bereiche in welchem Umfang untersucht werden. Eine *systematische Analyse* zeichnet sich dadurch aus, dass für sämtliche Programmteile versucht wird, ihre Bedeutung und Funktionsweise zu erschließen. Erst dann wird eine etwaige Änderung am Code vorgenommen. Eine *bedarfsgesteuerte Analyse* hingegen betrifft nur die für die Änderung vermeintlich relevanten Teile des Codes. Auch wenn Untersuchungen wie die von Littman et al. zeigen, dass systematische Analysen, die ein vollständiges und auch die Programmdynamik widerspiegelndes Modell erbringen, allgemein zu besseren Ergebnissen führen, so eignen sie sich nicht für umfangreiche Systeme. An dieser Stelle müssen mit Hilfe von Werkzeugen die relevanten Code-Bereiche ermittelt werden.

Bedarfsgesteuerte Analysen werden laut Brooks immer Top-Down durchgeführt [Bro83], d. h. der Entwickler beginnt mit einer Ausgangshypothese, die er abhängig von den Analyseergebnissen sukzessive verfeinert. Ausdrucksstarke Programmiersprachen bieten syntaktische Konstrukte wie bspw. Klassen und Module, um Konzepte auf einem hohen Abstraktionsniveau formulieren zu können. Sie unterstützen damit die Zuordnung von Funktionalitäten entlang einer hierarchischen Dekomposition bis hinunter auf die Ebene des Quellcodes. Eine bedarfsgesteuerte Analyse kann dementsprechend an dieser Hierarchie ausgerichtet werden.

Solche syntaktischen Konstrukte reduzieren auch den Code-Umfang, da sich mit ihrer Hilfe die Sachverhalte ausdrücken lassen, die in einfacheren Programmiersprachen umständlich ausprogrammiert werden müssten.

Neben der Strukturierung des Codes zwecks Abbildung unterschiedlicher Abstraktionsebenen spielt auch die Anzahl von Elementen und deren Verbindungen untereinander eine Rolle. Coad und Yourdon beziehen sich bei der Angabe von Grenzwerten auf die Untersuchungsergebnisse von Miller (s. [CY90, S. 107]), der als Kapazitätsgrenze des Kurzzeitgedächtnisses sieben plus/minus zwei „Bits“ ermittelt hat [Mil56]. Auch Beck meint, dass die Architektur gut strukturierter Systeme mit Hilfe von drei bis fünf Objekten beschrieben werden können sollte (s. [And+93, S. 358]). Für das Paradigma der Strukturierte Programmierung vertreten Yourdon und Constantine allerdings bei Modulen (Module im Sinne der Strukturierten Programmierung) einen anderen Umfang [YC79, S. 166]. Dabei beziehen sie sich auf Baker, der die „klassische“ Ansicht vertritt, dass maximal 50 Zeilen — das entsprach damals einer auf einem Terminal darstellbaren Seite — eine gute Modulgröße sei. Weinberg hingegen sieht schon bei 30 Zeilen eine kritische Größe erreicht. Dieser Wert deckt sich mit den Empfehlungen, die Roock und Lippert geben [RL04, S. 35].

Die interne Ausgestaltung der feingranularen Elemente wie Funktionen und Methoden beeinflusst ebenfalls die Analysierbarkeit. Verzweigungen und Schachtelungen können die Anzahl möglicher Pfade durch ein solches Element in nur wenigen Zeilen schnell vergrößern. Ein Beispiel: Mit nur 255 Zeilen, wovon 42 deklarative und 131 ausführbare Statements sind, kommen die Elemente aus `TopPVCReinforcement.cpp` auf eine zyklomatische Komplexität von 14 [Teß04, S. 83], verursacht durch mehrfach verschachtelte `switch`-Anweisungen und Schleifen.

Zusammen mit den übrigen Erkenntnissen lassen sich für die Analysierbarkeit als Qualitätsuntereigenschaft damit folgende Schlüsse ziehen:

- Die Untersuchungen von Soloway et al. (s. [Sol+88]) bestätigen die Aussagen aus Kapitel 2, dass die Dokumentation — wenn überhaupt — nur eine untergeordnete Rolle bei der Wartung spielt. Bei einer Qualitätsbewertung des Quellcodes ist sie als ergänzende Informationsquelle somit nicht relevant.

- Die Analyse geschieht bedarfsgesteuert und folgt primär einem Top-Down-Ansatz. Die Orientierung des Entwicklers im Quellcode wird von jeder ihm bekannten Struktur unterstützt. Je nach Ausbildung und Erfahrung kann es sich dabei um das Ergebnis der Anwendung von Regeln aus der Objektorientierung handeln, um allgemein bewährte Programmier-Schemata (s. [SE89]), oder auch um Instanzen der herkömmlichen Komponenten- und Framework-Technologien sowie der Referenzarchitekturen des Anwendungsbereichs, wie sie bspw. in Produktfamilien auftreten. Das gilt auch für die konzeptionelle Trennung von Plattform- und Produktarchitektur, wie sie Soni et al. beobachtet haben (S. 43 oben) und die Komponenten, die Szyperski beschreibt. Die hierarchische Strukturierung und die dabei erzeugten Elemente und deren Verbindungen haben aufgrund des Top-Down-Ansatzes maßgeblichen Einfluss auf Analysierbarkeit.

- Der Umfang der Entitäten auf jeder Ebene sowie ihrer Verbindungen sollte sich an der kognitiven Leistungsfähigkeit des Menschen orientieren. Legt man als Richtwerte die in den Untersuchungsergebnissen von Miller et al. gefundenen Zahlen zugrunde, dann

sollten auf jeder Abstraktionsebene der verschiedenen Sichten auf das System nicht mehr als maximal neun Elemente vorliegen bzw. jedes Element maximal neun Verbindungen besitzen. Drei bis sieben Elemente/Verbindungen gelten als gut handhabbar.

- Die interne Strukturierung feingranularer Elemente sollte möglichst einfach ausfallen, um die Anzahl zu untersuchender Pfade gering zu halten.
- Je mehr Unterstützung durch syntaktische Konstrukte seitens der Programmiersprache besteht, wenn es um die Formulierung von Entwurfsentscheidungen geht, desto eher lassen sich Hypothesen bzgl. der Funktionsweise des Codes bestätigen bzw. sind aufgrund der Explizität ihrer Abbildung im Code gar nicht nötig. Dies entspricht dem von Bennicke und Rust angesprochenen „single source“-Prinzip [BR04, S. 37 1. Abs.]. Alle weiteren auf der Sprache aufbauenden Artefakte folgen einer eigenen Logik und Struktur und müssen indirekt erschlossen werden.
- Existiert eine bestimmte Funktionalität mehrfach und wird sie dazu von jeweils verschiedenen Stellen referenziert, muss der Code an all diesen Stellen korrigiert werden, wenn eine Änderung gerade diese Funktionalität betrifft. Damit steigt das Risiko unbeabsichtigter Seiteneffekte. Solche Redundanzen sind ein Zeichen fehlerhafter Dekomposition und weisen u. U. auf tiefer gehende Missverständnisse bei der Problemanalyse und dem Systementwurf hin.

Wie sich diese Schlussfolgerungen konkret im Qualitätsmodell niederschlagen, kann im Detail erst anhand des zugehörigen Softwaremodells gezeigt werden, da die verwendete Programmiersprache die Strukturierungsmöglichkeiten und Ausdrucksformen maßgeblich bestimmt. Auf den bisherigen Erkenntnissen lässt sich die folgende Hypothese aufstellen:

Hypothese 1. Die Bewertung der *Analysierbarkeit* eines Softwaresystems ist abhängig von (1) dem Umfang der Entitäten, (2) der verwendeten Programmiersprache und ihren Ausdrucksmöglichkeiten, (3) der Strukturierung des Codes (inkl. der Anzahl von Entitäten und Verbindungen in jeder Sicht), und (4) dem Auftreten von Redundanzen.

Änderbarkeit

Die Änderbarkeit eines Systems bestimmt, wie gut Änderungen implementiert werden können. Neben dem Quellcode gehören auch Korrekturen und Anpassungen der Dokumente dazu.

Wie bei der Analysierbarkeit zeigt sich auch hier eine Diskrepanz zwischen der Definition im ISO-Standard 9126-1 und den Vorschlägen für eine praktische Umsetzung im Standard ISO 9126-3. Als Kenngröße wird der Umfang der Änderungen in der Dokumentation und im Quellcode (in Form von Kommentarzeilen) angegeben. Diese Kenngröße ist aufgrund der Probleme bei der Analyse von Dokumentation ungeeignet, um eine prädiktive Bewertung des Quellcodes hinsichtlich der Änderbarkeit vorzunehmen, da keine Ableitung des Ressourcenbedarfs möglich ist.

Für die Änderbarkeit spielen ähnliche Faktoren eine Rolle wie für die Analysierbarkeit, denn auch hier geht es um einen möglichst geringen Zeit- und Ressourcenverbrauch. Allerdings besitzt die Strukturierung ein größeres Gewicht, denn wurde die Möglichkeit von Änderungen

oder Erweiterungen schon im Vorfeld bei der Entwicklung bedacht und an den jeweiligen Stellen entsprechend flexible Strukturen geschaffen, ist es einfacher, Änderungen vorzunehmen. Dies entspräche dann einer „besseren“ Änderbarkeit. Die Strukturierung besitzt eine weitere Eigenschaft, die sich auf die Änderbarkeit auswirkt: Sie kann den Impact von Änderungen einschränken, so dass korrigierende Arbeiten, die möglicherweise nach der eigentlichen Änderungen anfallen, verringert oder vermieden werden.

Änderbarkeit ist im Wesentlichen eine Funktion der Lokalität [BCK98, S. 82], wie sie durch das Kapselungsprinzip erzeugt wird. Die daraus resultierende Modularität und das Information Hiding führen zu einer Trennung von Schnittstelle und Implementierung und erlaubt nicht nur unabhängige Anpassungen der Implementierung, sondern hebt auch die Bedeutung der Schnittstelle als strukturierendes Element hervor [Amb01, S. 144ff]. Je nach verwendetem Programmierparadigma kann Kapselung durch ganz unterschiedliche Konzepte auf verschiedenen Abstraktionsebenen umgesetzt werden: Module, Pakete, Klassen, Funktionen, Methoden etc.

Der durch das Information Hiding eingeschränkte Zugriff auf die Interna eines Bausteins vermeidet die Entwicklung von eng gekoppelten Code-Strukturen, die die Änderbarkeit massiv beeinträchtigen können [Amb01, S. 144ff unten]. Je geringer die Kopplung, desto geringer sind die Auswirkungen einer Änderung. Information Hiding wird laut Meyer allerdings oft falsch verstanden [Mey97, S. 52]: Es geht *nicht* darum, aus irgendwelchen Sicherheit- oder Geheimnisgründen die Implementierung zu verbergen, sondern vielmehr darum, einen verlässlichen Kontrakt zwischen Anbieter und Nutzer zu gewährleisten . Die Trennung von Schnittstelle und Umsetzung ermöglicht allerdings auch den Verkauf von kompiliertem Code in Form von Bibliotheken. So müssen nur die Schnittstellen offen liegen und die Umsetzung bleibt in der Hand des Anbieters. In dieser Form wird das geistige Eigentum des Herstellers dem direkten Zugriff entzogen.

Eine hohe Modularität lässt sich laut Meyer erreichen [Mey97, S. 46ff], indem eine gute Übertragung der Strukturen der Problemdomäne auf das Anwendungssystem stattfindet. Die Güte dieser Dekomposition kann mit Hilfe von Kohäsionsmessungen abgeschätzt werden (s. S. 51). Weiterhin sollte die Anzahl der Verbindungen zwischen den beteiligten Bausteinen minimiert („few interfaces“) und der Informationsaustausch über die so verbliebenen Verbindungen möglichst gering gehalten werden („small interfaces“). Besonders um Seiteneffekte zu vermeiden, sollten Verbindungen auch immer *explizit* sein. Dies betrifft nicht nur die Verwendung der Schnittstellen, sondern auch indirekte Verbindungen über gemeinsame Datenstrukturen. D. h. wenn zwei Bausteine miteinander kommunizieren, so sollte dies aus dem Quellcode *beider* Bausteine hervorgehen („explicit interfaces“).

Neben der Anzahl und Bandbreite beeinflusst auch die Art der Kopplung und der Kohäsion sowie die Komplexität der Schnittstelle die Änderbarkeit. Yourdon und Constantine unterscheiden *normale* und *pathologischen* Kopplungen [YC79, S 260ff]. Zu Letzteren zählen sie die Kommunikation über Sichtbarkeitsgrenzen hinweg (Umgehen von Schichten und Schnittstellen durch direkte Referenzierung von Elementen) und die Kommunikation über globale Variablen oder ähnliche globale Strukturen („common environment coupling“). Kopplungen dieser Art verschlechtern nicht nur die Änderbarkeit, sie haben auch die Tendenz sich zu vermehren [YC79, S. 261]. Der Kohäsion ordnen Yourdon und Constantine anhand von sieben assoziativen Prinzipien verschiedene Gütegraden zu. *Zufällige*, *logische* und *temporäre* Kohäsionen sollten bei der Entwicklung möglichst vermieden werden, da sie Elemente zusammenfassen,

die aus funktionaler Sicht nichts miteinander zu tun haben. Die übrigen Gütegrade (prozedural, kommunikationsbetont, sequenziell, funktional) sind aus ihrer Sicht tragbar, aber allgemein schwierig zu erfassen. Die Komplexität einer Schnittstelle umfasst einerseits die Anzahl der Parameter, andererseits aber auch die Sequenz von Aufrufen, die notwendig ist, um die Funktionalität eines Bausteins abzurufen (z. B. Initialisierung, Datenübergabe, Aufruf der eigentlichen verarbeitenden Funktion, Abruf des Ergebnisses). Jeder Parameter kann über seinen Datentyp außerdem das Risiko einer weiteren Kopplung mit sich bringen. Sichtbare Datenelemente eines Bausteins gehören ebenfalls zur Schnittstelle und tragen zu ihrer Komplexität bei.

Um die Modularität systemweit mit Hilfe der Kopplung bewerten zu können, schlagen Yourdon und Constantine eine Heuristik auf Basis der sog. *Kontrollspanne* („span of control“) vor, die auch „fan-out“ genannt wird [YC79, S. 171]. Die Kontrollspanne umfasst alle direkt von einem Baustein aufgerufenen Bausteine. Ähnlich wie bei der Größe eines Elements deutet eine sehr große oder sehr kleine Kontrollspanne auf einen schlechten Entwurf hin. Yourdon und Constantine sehen eine untere Grenze bei eins bis zwei, die obere Grenze bei zehn. Sie weisen allerdings darauf hin, dass gut entworfene Bausteine oft an der unteren Grenze liegen. Eine zu große Kontrollspanne hingegen lässt vermuten, dass die Modularisierung übertrieben wurde, d. h. der Entwurf ist zu feingranular. Analog zum „fan-out“ beschreibt der „fan-in“ die Anzahl der eingehenden Aufrufe. Dieser Wert sollte immer möglichst groß sein, da er auf eine Konzentration von Funktionalität und eine Vermeidung von Redundanz hinweist und damit ein ein Zeichen guter Modularität ist.

Die Messung der Kohäsion und Kopplung nach der Definition ist in der bisher vorgestellten Form unabhängig vom verwendeten Programmierparadigma. Allerdings ist die Eignung der Grenzwerte von „fan-in“ bzw. „fan-out“ bei objektorientierten Systemen fraglich. Anders als Systeme, die mit den Mitteln der Strukturierten Programmierung entwickelt werden und einer strengen Aufruf-Hierarchie folgen (es gibt immer eine Kontroll-„Wurzel“), sind objektorientierte Systeme als Netzwerk kommunizierender Agenten organisiert. Dazu kommt, dass sich die Kontrollspanne nicht immer eindeutig feststellen lässt, da das Late-Bindung mehrere Aufrufziele ermöglicht.

Lieberherr und Holland übertragen die Kopplungs- und Kohäsionsbetrachtungen auf die objektorientierte Entwicklung. Im Rahmen des Demeter-Projekts entwickelten sie das danach benannte *Gesetz von Demeter* [LH89]. Dieses Gesetz besagt, dass jedes Objekt in einer Methode, das eine Nachricht empfängt, nur aus einer bestimmten Menge, den sog. „preferred objects“, stammen darf. Diese Menge umfasst:

- die Parameter der Methodenschnittstelle
- die `self`-Pseudovariable
- Objekte der zugehörigen Klasse (=Instanz- und Klassenvariablen)
- Objekte, die in der Methode selbst erzeugt werden.

Salopp könnte man übersetzen: „Sprich nur mit den engsten Bekannten.“ Die Auswirkung dieses Gesetzes hinsichtlich der Kopplung entspricht weitgehend den Zielen von Yourdon und Constantine: Pathologische Kopplungen werden vermieden, die Anzahl der Nachrichten wird

minimiert und die Kohäsion der Klassen wird verbessert, da mehr Wert auf Klasseninformationen (Variablen) gelegt wird. Darüber hinaus berücksichtigt das Gesetz die Vererbung. Kopplung und Kohäsion können damit als Qualitätsuntereigenschaft für eine Bewertung objektorientierter Systeme herangezogen werden. Die konkret in Frage kommenden Strukturen hängen von der eingesetzten Programmiersprache ab und lassen sich daher erst bestimmen, wenn das Software-Modell vorliegt. Als Hypothese lässt sich jedoch schon festhalten:

Hypothese 2. Die Bewertung der Änderbarkeit ist abhängig von (1) der Strukturierung des Quellcodes, (2) seinem Umfang, und (3) seiner Redundanz.

Stabilität

Die Stabilität beschreibt, bis zu welchen Grad ein System vor unerwünschten Seiteneffekten geschützt ist, die durch Änderungen ausgelöst werden (können).

Welche Seiteneffekte die Stabilität beeinflussen, wird im ISO-Standard nicht näher erläutert. In ISO 9126-2 werden ausschließlich Fehler beschrieben, die während Tests oder im Produktivbetrieb auftreten, ISO 9126-3 verweist exemplarisch auf die Anzahl von einer Änderung betroffener Variablen. Dieser Umstand legt nahe, dass Auswirkungen auf die Architektur, die u. U. massiv die Struktur des Systems beeinflussen, nicht als Seiteneffekte verstanden werden. In Frage kommen somit nur *syntaktische* und *semantische* Auswirkungen. Erstere zeigen sich bei der Verwendung statisch getypter Programmiersprachen inkl. statischer Typüberprüfung während des Kompiliervorgangs im Rahmen der Wartungsarbeiten und treten daher normalerweise nicht in Produktivsystemen auf. Dynamische getypte Sprachen hingegen quittieren unzulässige Aufrufe zur Laufzeit mit einem Typfehler und anschließendem Programmabbruch, wenn dies nicht als Möglichkeit beim Entwurf bedacht und der Typfehler entsprechend abgefangen wird.

Semantische Seiteneffekte zeigen sich in ihrer schwächsten Form durch die Produktion fehlerhafter Ergebnisse (der Wertebereich einer Funktion ist falsch, es wird ein falscher Zahltyp zugrunde gelegt, es wird mit einem falschen Zwischenergebnis aufgrund von Rundungsfehlern weiter gerechnet etc.). Sie können aber auch zur Sicherheitslücke werden, wenn bspw. Eingaben nicht geprüft werden; derartige Angriffe auf Datenbanken sind unter der Bezeichnung SQL-Injection bekannt. Besonders Programmiersprachen ohne automatisches Speichermanagement können ein Sicherheitsrisiko darstellen, da durch die manuelle Speicherverwaltung die Gefahr von Speicherlecks und unzureichend geprüften Speicherbereichen steigen kann. Unrühmlich hervorgetan haben sich in diesem Bereich C und C++. Meyer sieht das automatische Speichermanagement als absolute Notwendigkeit für die Entwicklung zuverlässiger objektorientierte Systeme an [Mey97, S. 332, S. 294ff]. Dadurch dass die Ursache allgemein in den Variablen*werten* zu suchen ist, hilft eine Typüberprüfung an dieser Stelle nicht weiter. Eine Möglichkeit ist, an definierten Stellen im Code Tests der Werte zu platzieren. Gemeinhin wird dies auch mit dem Aufbau einer „internen Firewall“ verglichen. Die Methode des *Defensive Programming* befasst sich mit dieser Art der Überprüfung: Sämtliche eingehende Daten werden zunächst als potentiell unsicher angesehen und müssen getestet werden, bevor die Funktion oder Methode weiterarbeitet. Diese Tests können entweder mit den üblichen Sprachmitteln wie Konditionalausdrücken oder mit Hilfe sog. *Assertions* implementiert werden. Assertions

prüfen durch einfache logische Ausdrücke, ob eine Variable bzw. ihr Wert eine bestimmte Bedingung erfüllt. Ist dies nicht der Fall, kommt es zu einem Assertion-Fehler und die Programmausführung wird abgebrochen, da die Methode oder Funktion kein sinnvolles Ergebnis mehr produzieren kann. Für eine mathematisch ausgerichtete Betrachtung von Assertions und ihre Bedeutung bei der Zuordnung von Semantik zu Programmen sei auf [Flo67] verwiesen. Meyer zitiert Hoare, der darauf hinweist, dass Alan Turing höchstselbst 1950 Assertions als einfaches Mittel zur Überprüfung eingeführt hat [Mey97, S. 407].

Assertions sind mittlerweile Teil aller gängigen Programmiersprachen, bieten in diesen Ausprägungen allerdings nur punktuellen Schutz, der zudem die Wartbarkeit schon durch die reine Menge an Zusatzcode eher erschwert denn verbessert. Meyer geht mit seiner Alternative des *Design by Contract* einen wesentlichen Schritt weiter und fasst die Assertions nur als Teil eines größeren Konzepts zur Sicherstellung der Programmkorrektheit auf [Mey97, S. 331ff]. Basis dieses Konzepts ist die Auffassung, dass die Beziehung einer Klasse zu einem sie benutzenden „Klienten“ (üblicherweise ein Objekt einer anderen Klasse) als formale Übereinkunft mit beiderseitigen Rechten und Pflichten zu sehen ist. Für die Umsetzung nutzt er ebenfalls Assertions, die in dieser Form jedoch nur noch die Logik mit den obigen Assertions gemein haben:

> „An assertion is an expression involving some entities of the software, and stating a property that these entities may satisfy at certain stages of software execution.“ [Mey97, S. 337]

Damit können Assertions nicht nur auf Methoden, sondern auch auf Klassen angewendet werden; sie bilden die Grundbausteine für die Formulierung sog. *Pre-* und *Postconditions* sowie *Klasseninvarianten*. Methoden (oder Routinen, wie Meyer sie nennt) können mit Hilfe von Preconditions Bedingungen zugewiesen werden, die beim Aufruf der Methode erfüllt sein müssen. Wie der Name nahe legt, beschreiben Postconditions Bedingungen, die zum Methodenende erfüllt sein müssen. Eine Klasseninvariante überträgt die Bedingungen auf eine Klasse.

Auf Folgendes sei jedoch hingewiesen: Auch wenn durch die Formulierung von Bedingungen durch Assertions dem normalen Code ähnliche Zeilen erzeugt werden, so darf nicht vergessen werden, dass damit keine *imperative* Aussage erzielt werden soll, sondern eine *applikative*. Assertions sind keine Instruktionen, die wie normale Befehle abgearbeitet werden und den Zustand der „virtuellen Maschine“ verändern, sondern es handelt sich dabei um eine beschreibende Formulierung, die spezifiziert, wie bestimmte mathematisch-logische Methoden *angewendet* werden. Sie beschreibt, „was“ getan wird, nicht aber das „wie“, und ergänzt damit die syntaktische Seite des Codes (das „wie“) um eine semantische (und dokumentiert damit den Code). Dadurch kann die *Korrektheit* eines Programms bis zu einem gewissen Grad garantiert werden. Meyer betont den methodologischen Aspekt von Assertions: Sie sind ein konzeptuelles Werkzeug für die Analyse, den Entwurf, die Implementierung und Dokumentation mit dem Ziel, „eingebaute Zuverlässigkeit“ zu erreichen [Mey97, S. 341].

Mit Hilfe von Assertions lässt sich also konzeptionell die Zuverlässigkeit bzw. Stabilität eines Systems sicherstellen. Was aber soll passieren, wenn eine Assertion nicht erfüllt wird? Zuverlässigkeit wird von Meyer unterteilt in *Korrektheit* und *Robustheit*. Die Korrektheit bezieht sich auf die Konsistenz von Implementierung und Spezifikation, die durch Assertions sichergestellt werden kann. Die Robustheit betrifft diejenigen Situationen, in denen Assertions

nicht eingehalten werden oder das System aus anderen Gründen nicht weiterarbeiten kann. Ein Programmabbruch ist allgemein keine passable Lösung, insbesondere nicht bei Systemen im Produktivbetrieb. Meyer schlägt daher die Verwendung von *Exceptions* vor. Sie eröffnen zwei Lösungswege:

- erneuter Versuch: Nach einer Korrektur der Umstände, die zu der Exception geführt haben, wird erneut versucht, die Routine auszuführen.
- Fehler (auch bekannt als „organized panic“): Die „Umgebung“ wird aufgeräumt (temporäre Objekte löschen, Datei-Handles schließen usw.), der Aufruf terminiert und ein Fehlerreport an den Aufrufer abgesetzt.

Der erste Lösungsweg stellt den Idealfall eines stabilen und robusten Systems dar. Allerdings können nicht immer alle Eventualitäten beim Entwurf bzw. der Implementierung bedacht werden, so dass früher oder später der zweite Lösungsweg beschritten werden muss. Dazu müssen allerdings zwei Bedingungen erfüllt sein ([Mey97, S. 417ff]):

1. Der aufrufende Klient muss eine Ausnahme erhalten (der „panic-“Teil), die ihn über den Fehler informiert.
2. Das System muss in einen konsistenten Zustand überführt werden (der „organized“-Teil), damit anschließend korrekt weitergearbeitet werden kann .

Exceptions sind mittlerweile Teil der meisten objektorientierten Programmiersprachen, auch wenn das Design-by-Contract-Konzept von ihnen i. a. nicht direkt unterstützt wird.

Fasst man die einzelnen Aspekte, die Einfluss auf die Stabilität haben, vor dem Hintergrund des entstehenden Qualitätsmodells zusammen, lässt sich für die syntaktische Seite festhalten, dass die Verwendung statisch getypter Sprachen durch die statische Typüberprüfung einen gewissen Schutz vor Seiteneffekten bieten. Gewiss deshalb, weil jede der aktuell verwendeten Sprachen die Möglichkeit bietet, die Typisierung durch Typumwandung (Cast) zu unterlaufen. Trotz dieses Risikos kann davon ausgegangen werden, dass die Stabilität in diesem Bereich nur gering beeinflusst wird. Daher wird die „syntaktische Stabilität“ nicht in das zu entwickelnde Qualitätsmodell eingehen. Die Wahl einer Programmiersprache mit einer automatischen Speicherverwaltung (*Garbage Collection*) hingegen trägt nachhaltig zur Stabilität bei. Der Einsatz von Assertions (in Form von Pre-/Postconditions und Klasseninvarianten) bzw. die Eingabe-/Ausgabekontrolle an kritischen Stellen im System im Sinne des Defensive Programming verbessert die semantischen Stabilität. Das gilt auch für die Fehlerbehandlung durch Exceptions.

In welcher konkreten Form sich Stabilität als Struktur im Code manifestiert, hängt von der verwendeten Programmiersprache ab und ist damit Teil des Software-Modells. Eine Integration dieses Qualitätsaspekts in das Qualitätsmodell ist jedoch nur eingeschränkt möglich. Wie auch bei den beiden vorherigen Untereigenschaften zeigt sich die Stabilität erst im Betrieb, d. h. *nachdem* die Änderungen eingepflegt worden sind. Alle hier aufgeführten Sicherungsmaßnahmen sind daher ausschließlich *präventiver* Natur. Dementsprechend ist eine Bewertung der Stabilität nur über Prädiktoren möglich, die das Vorhandensein von stabilisierenden Maßnahmen bzw. die Verwendung sich negativ auswirkender Konstrukte wie Return-Codes anzeigen.

Sinnvoll erscheint die Konkretisierung des Stabilitätsbegriff durch die von Meyer unter dem Oberbegriff Zuverlässigkeit aufgeführten Qualitätseigenschaften *Korrektheit* und *Robustheit*, da sie durch die eine Verbindung zu Strukturen im Code hergestellt werden kann. Eine tiefer gehende Bewertung gestaltet sich jedoch schwierig: Ist die Abwesenheit von Return-Codes als positiv für die Robustheit zu bewerten? Können sie überhaupt von normalen „getter"-Return-Angaben unterschieden werden? Wie umfangreich müssen Exceptions zum Einsatz kommen, damit von einem „stabilen" System gesprochen werden kann?

Zunächst kann für das Qualitätsmodell folgende Hypothese aufgestellt werden:

Hypothese. *Die Stabilität eines Softwaresystems — die Abwesenheit von Seiteneffekten, hervorgerufen durch Modifikationen — wird bestimmt durch seine Korrektheit und Robustheit.*

Korrektheit und Robustheit hängen zwar partiell von den Eigenschaften der verwendeten Programmiersprache ab, da es in dieser Arbeit aber um einen allgemeinen Ansatz zur Qualitätsbewertung von Quellcode geht, werden Spezifika wie Speicherverwaltung und Klasseninvarianten als gegeben vorausgesetzt und gehen nicht weiter in die Bewertung ein. Entsprechend lässt sich die Hypothese verfeinern:

Hypothese 3. Die Stabilität bzw. die Korrektheit und Robustheit eines Systems hängt ab von (1) der Verwendung von Assertions (dies betrifft die Korrektheit) und (2) von der Verwendung von Exceptions zur Fehlerbehandlung (dies betrifft die Robustheit).

Die Untereigenschaften Korrektheit und Robustheit können zwar anhand der beschriebenen konzeptionellen Code-Strukturen identifiziert werden, eine Bewertung ist aber nur nominal möglich. Es lässt sich nur bedingt abschätzen, in welchem Umfang Assertions und Exceptions die Stabilität verbessern. Für Assertions kann folgende Annahme gemacht werden: Besitzt eine Funktion oder Methode Parameter, so hängt ihre Funktionsweise von den übermittelten Werten ab. Dem Defensive Programming-Ansatz zufolge müssten diese mit Assertions geprüft werden. Für die Bewertung der Stabilität wäre dann ein Abgleich der Parameterliste und der zugehörigen Assertions denkbar, soweit diese denn vorhanden sind. Bei der Verwendung von Exceptions könnte die Anzahl gefangener und weitergeleiteter Ausnahmen und ein Vergleich beider Werte Hinweise auf die Robustheit geben. Bspw. deuten zahlreiche `catch`-Blöcke in einer Methode und wenige/keine weitergereichten Ausnahmen darauf hin, dass hier ein erneuter Ausführungsversuch unternommen werden soll.

Bei der Bewertung stellt sich außerdem die Frage, wie Assertions und Defensive-Programming-Techniken vor dem Hintergrund der allgemeinen Wartbarkeit einzuschätzen sind. Beide Ansätze erhöhen den Code-Umfang, der bei Modifikationen potentiell angepasst, auf jeden Fall aber überprüft werden muss. Je nach Systemgröße kann dies zu einen kaum überschaubaren Mehraufwand führen [Mey97, S. 343f]. Eine Antwort kann hier jedoch nicht gegeben werden und muss Gegenstand weiterer Untersuchungen sein.

Testbarkeit

Die Testbarkeit soll die Korrektheit des Programms nach Änderungen möglichst umfassend sicherstellen.

Die im ISO-Standard 9126-3 vorgeschlagenen Verfahren zur Erfassung der Testbarkeit befassen sich mit dem Umfang autonomer Hilfsfunktionen im Quellcode. Die dazu angeführten Metriken zur Vollständigkeit bzgl. der in der Spezifikation aufgeführten Tests und der Darstellung von Testergebnissen während eines Testlaufs sind für eine Beurteilung des Quellcodes jedoch irrelevant. Die Ermittlung der Testautonomie des Systems hingegen ist sowohl auf der Systemebene als auch systemintern wichtig, da besonders bei umfangreichen Systemen ein manueller Test eine umfassende Prüfung der Korrektheit nicht mehr gewährleisten kann. Für eine Bewertung der systeminternen Testbarkeit ist damit aus Sicht der Autonomie der Grad und die Güte der Modularisierung ausschlaggebend.

Autonomie ist allerdings nur ein Aspekt der Testbarkeit. Binder versteht Software-Tests ganz allgemein als das Ausführen von Code unter der Verwendung von ausgewählten Eingabe- und Zustandskombinationen, um Fehler aufzuspüren [Bin99, S. 44]. Die Grundlage dafür sind *Komponenten*. Aus Testsicht ist eine solche Komponente ein während der Entwicklung sichtbares Software-Aggregat wie bspw. eine Klasse, Funktion oder Subsystem, das über eine Schnittstelle verfügt. Dadurch existiert eine Möglichkeit, unter Wahrung der Kapselung auf die Funktionalitäten des Aggregat zuzugreifen. Eine „angemessene“ Modularität ist damit nicht nur eine Voraussetzungen für die Testautonomie, sondern für das Testen überhaupt.

Tests sollten immer einer definierten *Teststrategie* folgen, um in der Menge aller möglichen Tests diejenigen zu finden, die mit vertretbarem Aufwand eine möglichst umfangreiche Korrektheit erreichen. Dabei kann ein Test jedoch nie die Abwesenheit von Fehlern zeigen, sondern nur ihr Vorhandensein (Dijkstra in [DDH72]). Konkrete Teststrategien werden immer auf der Basis eines *Fehlermodells* entwickelt [Bin99, S. 66f]. Zwei allgemeine Fehlermodelle und zugehörige Strategien sind bekannt: Das „conformance-directed testing“ prüft, ob das System einer bestimmten Spezifikation oder Anforderung genügt. Die Konformität kann allerdings auch für Systeme gezeigt werden, deren Implementierung fehlerhaft sind. Die zweite Strategie, das „fault-directed testing“, sucht nach solchen Implementierungsfehlern. Da die Anzahl möglicher Kombinationen von Eingaben, Systemzuständen, Ausgaben und Ausführungspfaden schon bei kleinen System sehr groß werden kann, ist für die zweiten Strategie ein Fehlermodell notwendig, dass neben den besonderen Eigenheiten des Systems auch die für das Programmierparadigma spezifischen Fehlerformen und -schwerpunkte berücksichtigt. Das Fehlermodell ermöglicht damit eine Abschätzung, mit welcher Menge an Testdaten die Korrektheit überprüft werden kann, ohne dass wesentliche Fehler unerkannt bleiben.

Für eine Bewertung der Testbarkeit von Quellcode ist nur das „fault-directed testing“ und die damit verbundenen Strukturen im Quellcode relevant. Aufbauend auf der Teststrategie werden Test-Suiten entworfen, wobei die zwei folgenden Ansätze relevant sind:

- Responsibility-based/Black Box Testing: Ausgehend von den Verantwortlichkeiten gegenüber anderen Systembestandteilen wird die Funktionalität einer Komponente getestet. Die konkrete Implementierung der Komponente ist dabei nicht von Interesse, es wird ausschließlich über die Schnittstelle kommuniziert.

- Implementation-based/White Box Testing: Im Gegensatz zum Black-Box-Testen wird bei diesem Ansatz ganz explizit auf die Implementierung eingegangen. Dazu ist allerdings entweder auf der Seite der Programmiersprache ein Mechanismus notwendig, der einen

Zugriff auf die Interna einer Komponenten ermöglicht (z. B. der friend-Mechanismus in C++), oder es muss eine beim Testen nachträgliche Instrumentierung des Codes stattfinden, die einen solchen Zugriff herstellt.

Das System wird für den Test üblicherweise in die Bereiche „Unit“, „Integration“ und „System“ unterteilt. *Unit-Tests* befassen sich mit den kleinsten Komponenten, die getestet werden können. Dazu zählen bspw. Klassen. *Integrationstests* prüfen die Komposition der Komponenten und testen die Kommunikation sowie die Schnittstellen. *Systemtests* untersuchen die funktionalen und nicht-funktionalen Eigenschaften des Gesamtsystems. Dieser Ansatz ermöglicht — eine entsprechende Modularität vorausgesetzt — umfassende automatische Tests und hat schon vor geraumer Zeit Eingang in die Softwareentwicklung gefunden. Bekannt ist der *Continuous Integration*-Ansatz (s. bspw. [DMG07]), der von verschiedenen Implementierungen flankiert wird (u. a. CruiseControl [10a] und BuildBot [War10]) und die existierenden Unit-Test-Frameworks nutzt, z. B. JUnit ([06b]).

Unbeantwortet ist allerdings noch immer Frage nach der adäquaten Testmenge, die mit Hilfe des Fehlermodells abgeschätzt werden soll. Weyuker hat dazu einen Kriterienkatalog entworfen [Wey88], mit dessen Hilfe die Güte von Testdaten für das White-Box-Testen bestimmt werden kann. Aus den Kriterien lassen sich Anforderungen an die Strukturierung des Quellcodes ableiten.

Welche Auswirkungen haben diese Axiome zusammen mit den durch die Objektorientierung eingeführten Strukturierungsmöglichkeiten auf die Testbarkeit? Das objektorientierte Paradigma bietet mächtige Konstrukte für die Entwicklung von Systemen, die jedoch mit ganz eigenen Fehlerrisiken und Testproblemen zu kämpfen haben. Die Kapselung von Operationen und Variablen in Klassen, die vielen Wege, ein System aus Objekten zu konstruieren sowie die Formulierung komplexen Verhaltens mit Hilfe weniger einfacher Statements ermöglicht es dem Entwickler, eleganten und schlanken Quellcode zu schreiben [Bin99, S. 67]. Perry und Kaiser widerlegen jedoch die intuitiv naheliegende Vermutung, dass bspw. durch die Vererbung der Testaufwand in ähnlichem Maße wie den Programmieraufwand reduziert werden kann [PK90]. Indem sie die Axiome von Weyuker auf die Vererbung anwenden, kommen sie zu dem Ergebnis, dass obwohl eine Klasse, die als adäquat getestet angesehen wird, alle davon abgeleiteten Klassen separat getestet werden müssen (folgt aus dem Antidecomposition-Axiom). Darüber hinaus erfordern diese Tests für jede Klasse eine *eigene* Testmenge (folgt aus dem Antiextensionality-Axiom). Dies gilt auch für in Subklassen überschriebene Methoden. Insgesamt attestieren Perry und Kaiser der Objektorientierung einen höheren Testaufwand, da jede Vererbungsebene separat getestet werden muss, und das sowohl komponentenweise als auch integrativ. Das „Urgestein“ des prozeduralen Tests, die statische Pfadanalyse, ist angesichts der Möglichkeiten des Polymorphismus (sowohl statisch zur Compile-Zeit, bspw. durch Templates oder Generics, oder dynamisch zur Laufzeit) und des Late Binding für den Entwurf von Tests nur von geringen Nutzen. Zudem erhöhen beide Techniken massiv die Anzahl möglicher Ausführungspfade. Bis dato gibt es keine Werkzeuge, die bei der Test Coverage-Analyse über die Methode und deren statischen Aufruf hinausgehen [Bin99, S. 105].

In Anbetracht der für einen vollständigen Test notwendigen Menge von Einzeltests und des Statement Coverage-Axioms ist die Testbarkeit großer objektorientiert implementierter Systeme insgesamt fraglich, sowohl was den Entwurf der Testmenge, ihren Umfang als auch den

Zeitbedarf angeht. Es gibt jedoch Spekulationen, nach denen durch einen umfassenden Test der verwendeten Basisklassen der Aufwand für die Tests der abgeleiteten Klassen um 40–70% reduziert werden kann [Bin99, S. 97]. Belege wurden dafür zwar bisher noch nicht erbracht, aber Harrold et al. konnten zeigen, dass ihr *Hierarchical Incremental Testing*-Ansatz in der Lage ist, die Testmenge zu reduzieren bzw. die Möglichkeit der Wiederverwendung von Tests zu demonstrieren [Bin99, S. 100]. Die Strukturierung des Quellcodes mit Hilfe der Vererbung verbessert damit potentiell die Testbarkeit.

Zusammenfassend betrachtet wird die Testbarkeit eines Systems im Wesentlichen bestimmt durch den Grad der Autonomie der Tests und dem Testumfang. Die Autonomie betrifft dabei sowohl die funktionale als auch die strukturelle Dekomposition. Erstere erlaubt eine separate Prüfung der einzelnen Funktionalitäten und betrifft damit die Kohäsion der Komponenten. Die strukturelle Dekomposition zielt ab auf die Möglichkeit, Komponenten unabhängig voneinander testen zu können und wird entsprechend beeinflusst durch deren Kopplung. Die Beschränkung des Testumfangs, wie sie bspw. durch den Einsatz von Vererbung möglich scheint, ist noch nicht soweit erforscht, als dass sie eine valide Bewertungsmöglichkeit bieten könnte. Die Vererbungstiefe wird daher nur insoweit einbezogen, dass sich flache Hierarchien positiv auf die Testbarkeit auswirken. Ebenso wenig existieren konkrete Zahlen, die einen akzeptablen Autonomiegrad beschreiben. Daher werden für die Bewertung der Testbarkeit dieselben Kriterien zugrunde gelegt wie für die Bewertung der Änderbarkeit (s. Abschnitt 5.3.1, S. 70). Vererbungshierarchien verschlechtern die Testbarkeit durch den höheren Testaufwand. Somit kann folgende Hypothese zur Testbarkeit aufgestellt werden:

Hypothese 4. Die Testbarkeit eines Systems wird bestimmt durch (1) den Grad der Autonomie der zu testenden Komponente und damit durch die Modularität, und (2) die Größe der in der Komponente vorliegenden Vererbungstiefen.

Standardkonformität

Der ISO-Standard 9126-3 schlägt zur Ermittlung der Standardkonformität des Quellcodes bzgl. seiner Wartbarkeit vor, anhand von dazu existierenden Standards die Implementierung zu prüfen und Verstöße aufzuzeigen. Etablierte und auch *brauchbare* Standards sind jedoch nicht verfügbar, wie Kapitel 2 deutlich macht. Auch der einzige weit verbreitete Standard, der sich mit der Wartbarkeit näher befasst (ISO 9126), macht keine konkreten Angaben zu Quellcode-Strukturen u. ä. (s. zur Eignung auch Abschnitt 2.3, S. 16).

Was einem Standard für Quellcode-Konformität am Nächsten kommt, sind sog. *Coding Standards*, wie sie für die meisten Programmiersprachen existieren. Für Java gibt es bspw. die *Java Code Conventions* ([99b]), die sehr detailliert auf die Organisation der Quellcode-Dateien, Einrücktiefe, Kommentierung, und Namenskonventionen für Bezeichner eingehen. Darüber hinaus geben sie Hinweise auf Best Practices bei der Programmierung. Besonders bei populären Sprachen stehen jedoch mehrere solcher Coding Standards in Konkurrenz (für Java gibt es bspw. auch Coding-Konventionen von IBM und der US Navy).

Zusätzlich zu solchen allgemeinen Richtlinien werden für größere Projekt oft eigene Vorgabenkataloge erstellt. Sie enthalten Angaben zu projektspezifischen Bezeichnern oder zur Dokumentation des Quellcodes, um daraus mit Werkzeugen wie Javadoc ([04b]) API-Beschreibungen

extrahieren zu können. Ein Beispiel für einen derart strukturierten Bezeichner: `CTopFittingTableNamesData` ([Teß04, S. 90]); dies ist der Name einer Klasse (`C`) aus dem Modul top (`Top`), die sich der Beschlagsermittlung (`Fitting`) befasst.

Durch die projektspezifische Entscheidung für einen der konkurrierenden Coding Standards und den zusätzlichen eigenen Vorgaben ist ein allgemeiner Bewertungsansatz nur schwer zu realisieren. Sollte auch ein kleinster gemeinsamer Nenner aus den betreffenden Vorgaben ermittelt werden können, so stellt sich weiterhin die Frage, wie Verstöße zu bewerten sind. Ist ein nicht konformer Klassenname ebenso gravierend wie der Verstoß gegen eine Strukturvorgabe? Letztendlich zielen Standards dieser Art darauf ab, dass sich ein Entwickler schnell im Quellcode zurecht findet und bekannte Fallstricke in der Programmierung durch Anwendung von Best Practices vermieden werden. Die Orientierung wird schon mit der Analysierbarkeit bewertet (s. Abschnitt 5.3.1, S. 67). Die Befolgung von Best Practices kann nicht ohne Weiteres ermittelt werden, da es sich in der Regel um allgemein formulierte Faustregeln handelt und nicht im konkrete Hinweise. Die Kapitel 2 bis 4 beleuchten derartige Regeln und die damit verbundenen Schwierigkeiten. Aus diesen Gründen findet die separate Bewertung der Standardkonformität keinen Eingang in das in dieser Arbeit entwickelte Qualitätsmodell.

5.3.2 Portabilität

Die Portabilität eines Systems wird geprägt durch seine *Anpassungsfähigkeit*, *Installierbarkeit*, *Koexistenz*, *Ersetzbarkeit* und *Standardkonformität*.

Portabilität bezeichnet die Eigenschaft eines Systems, in anderen Systemumgebungen eingesetzt werden zu können. Dabei kann es sich um andere Hardware- aber auch Softwareumgebungen handeln sowie um eine Übertragung auf eine neue Version der verwendeten Programmiersprache. In kleinerem Umfang tritt Portabilität in Form der *Konfiguration* in Erscheinung und bezeichnet dann die Anpassung einer Installation an die konkreten Einsatzbedingungen.

Anpassungsfähigkeit

Die Anpassungsfähigkeit eines Systems zeigt sich daran, wie gut es an eine konkrete Einsatzumgebung angepasst werden kann.

Der ISO-Standard 9126-3 empfiehlt für die Ermittlung dieser Untereigenschaft, die Anpassungsfähigkeit von Datenstrukturen zu erheben, die Anzahl an Hardware- und Software-abhängigen Funktionen zu erfassen und deren Änderbarkeit zu bestimmen. Die Abhängigkeit von Hard- und Software sollte nach den Empfehlungen von Bass et al. in wenigen Modulen konzentriert werden, was praktisch durch die Trennung von Plattform- und Produktarchitektur auch umgesetzt wird, wie Soni et al. ermitteln konnten. Für eine Identifikation dieser Stellen im Quellcode ist allerdings Zusatzwissen auf Seiten des Entwicklers erforderlich, da aus dem Code selbst nicht notwendigerweise hervorgeht, ob bspw. eine verwendete Klasse vom Betriebssystem oder einem Zusatzprodukt bereitgestellt wird. Dies gilt auch für die Hardware-nahen Module. Eine gute Modularisierung ist aber die Voraussetzung für eine derartige Kapselung von Funktionen, weshalb eine Abschätzung der Anpassungfähigkeit auch durch die Bewertung der Änderbarkeit erfolgen kann.

Hypothese 5. Die Bewertung der *Anpassungsfähigkeit* ist abhängig von der Strukturierung des Quellcodes in der Form, wie sie auch zur Bewertung der Änderbarkeit herangezogen wird.

Installierbarkeit

Die Installierbarkeit untersucht, wie gut ein System in eine bestehende Umgebung eingefügt werden kann.

Die Empfehlungen aus ISO 9126-3 beschränken sich darauf, die Anzahl der Installationsschritte und deren Automatisierungsgrad zu erfassen. Da sowohl die Anforderungen, die durch die neue Umgebung bestehen, als auch die dazu passenden Schnittstellen auf der Systemseite sich allgemein nicht aus dem Quellcode herleiten lassen, ist dieses Untereigenschaft nicht geeignet, Teil des Qualitätsmodells zu werden und wird daher nicht weiter berücksichtigt.

Koexistenz

Die Koexistenz zeigt, wie gut ein System mit anderen Systemen zusammenarbeiten kann.

Ermittelt werden soll laut Empfehlung dazu die Anzahl der Einheiten, mit denen das Produkt laut Spezifikation zusammenarbeiten kann, und die Anzahl der im produktiven Einsatz befindlichen Einheiten, die eine Koexistenz benötigen. Da sich die Angaben dazu nur in der Spezifikation wiederfinden, ist wie bei der Bewertung der Anpassungsfähigkeit auch hier Zusatzwissen notwendig, um die Abhängigkeiten im Quellcode identifizieren zu können. In welchen Umfang diese Identifikation möglich ist, lässt sich angesichts der vielen Formen von Abhängigkeit kaum vorhersagen Im einfachsten Fall wird möglicherweise nur ein Paket importiert, schwieriger nachzuweisen ist hingegen die Verwendung einer bestimmten Datenbank. Eine vergleichbare Orientierung der Bewertung an der Strukturierung, wie sie bei der Anpassungfähigkeit möglich ist, kann daher nicht vorgenommen werden. Dementsprechend gibt es keine allgemein eindeutigen im Quellcode zu ermittelnden Merkmale, die eine Aufnahme der Koexistenz als Untereigenschaft in das Qualitätsmodell rechtfertigen könnten.

Ersetzbarkeit

Die Ersetzbarkeit eines Systems beschreibt die Ersetzung einer bestehenden Installation durch eine neue Version oder durch ein kompatibles Produkt.

Bewertet wird die Ersetzbarkeit nach ISO 9126-3 durch die Anzahl der Daten und Funktionen, die bei einem Wechsel unverändert übernommen bzw. verwendet werden können. Ausgangspunkt ist die Spezifikation der betroffenen Daten und Funktionen, wodurch eine automatisierte Herleitung aus dem Quellcode schwierig ist. Für eine Beurteilung könnte man sich einen Vergleich des Quellcodes der alten und neuen Version vorstellen, der die Unterschiede in der Datenbehandlung bzw. Funktionalität offenlegt. Da Versionswechsel aber üblicherweise umfassende Änderungen am Quellcode mit sich bringen, würden diese Unterschiede in der Menge wahrscheinlich untergehen. Darüber hinaus dürfte es schwierig werden, an den Quellcode eines gekauften Produkts zu gelangen. Die Ersetzbarkeit ist damit insgesamt nicht geeignet, die Portabilität im Rahmen des Qualitätsmodells zu bewerten.

Standardkonformität

Die Bewertung der Standardkonformität bzgl. der Portabilität wird aus denselben Gründen nicht mit in das Qualitätsmodell aufgenommen, wie sie in Abschnitt 5.3.1 auf S. 79 beschrieben werden.

Zusammenfassung Abschnitt 5.3

Die in den Unterabschnitten aufgestellten Hypothesen zielen im Wesentlichen ab auf die hierarchische Dekomposition in Form der Vererbung und Aggregation bzw. Komposition von Strukturelementen. Dabei ist sowohl der Umfang der Vererbung und Elemente als auch die Anzahl und Art der Verbindungen von Interesse. Maßgebliches strukturierendes Konstrukt ist die Schnittstelle. Sie begrenzt nach außen die Art und den Umfang der Kopplung und hat Einfluss auf die Autonomie der einzelnen Elemente.

Die Programmiersprache ist als externer Einflussfaktor kein Teil des Qualitätsmodells, sondern konkretisiert die möglichen Ausprägung einzelner Merkmale durch ihre syntaktische Elemente. Je nach Mächtigkeit kann sie allerdings auch die Konkretisierung einschränken; dies betrifft hauptsächlich die Stabilität mit ihren Pre-/Postconditions und Klasseninvarianten, da diese von nur wenigen Sprachen unterstützt werden.

Die Zerlegung der Portabilität in Untereigenschaften hat gezeigt, dass diese Qualitätseigenschaft im Bereich der Anpassungsfähigkeit große Ähnlichkeit mit der Änderbarkeit hat und daher eine weitere Unterteilung der Anpassungsfähigkeit nicht sinnvoll ist. Sämtliche weiteren Untereigenschaften sind für eine Ermittlung aus Quellcode nicht geeignet und werden nicht mit in das Qualitätsmodell aufgenommen.

Nach einer kurzen Betrachtung des Komplexitätsbegriffs kann jetzt die Zerlegungssystematik des Qualitätsbegriffs definiert werden. Die Zuordnung der Hypothesen bzw. der daraus abgeleiteten Qualitätsmerkmale bereitet dann die Definition der Messaufgaben und -objekte vor.

5.4 Komplexität als Qualitätsmerkmal?

Der Komplexitätsbegriff taucht im Bereich des Software-Engineering und der Analyse an vielen Stellen auf. Es gibt Maße für die Ermittlung von Komplexität und Maßnahmen, um mit zu großer Komplexität umzugehen. Manche versuchen auch eine Abgrenzung von Komplexität und Kompliziertheit: Komplizierte Systeme erlauben sichere Vorhersagen bei Kenntnis aller Komponenten und deren Beziehungen, komplexe Systeme hingegen sind auch dann nicht vorhersagbar. Im allgemeinen Sprachgebrauch werden beide Begriffe synonym verwendet, wobei in der Informatik wohl eher die Kompliziertheit eines Systems gemeint ist. Der Entwickler sollte schließlich genau wissen, was das von ihm geschriebene Programm tut.

Komplexität bzw. Kompliziertheit ist immer dann ein Problem, wenn sie zu Verständnisschwierigkeiten führt. Die wahrgenommene Komplexität eines Systems bzw. des zugrunde liegenden Quellcodes könnte daher als negative Qualitätseigenschaft in Betracht gezogen werden. Der Versuch einer Einbeziehung ist allerdings problematisch:

1. Zunächst wäre eine Definition des Begriffs notwendig. Viele Autoren haben sich daran versucht, allein in der Informatik gibt es Definitionen mit völlig unterschiedlichen Zielrichtungen: Die Komplexitätstheorie versucht, den Ressourcenbedarf eines Problems abzuschätzen; die Definition im Kontext der Informationstheorie beschreibt den Informationsgehalt einer Nachricht (Kolmogorov-Komplexität). Für die Qualitätsbewertung kommt Komplexität als Gegenteil von Einfachheit einer brauchbaren Definition wohl am Nächsten.

2. Komplexität ist immer abhängig vom Betrachter, d. h. es handelt sich hierbei um eine subjektive Einschätzung, die abhängig ist von Erfahrung und Vorbildung. Die gebotenen Informationen weisen eine nicht angemessene Strukturierung auf bzw. werden nicht entsprechend präsentiert, wodurch es zu einer Überforderung der Kognitionsfähigkeiten kommt (s. dazu auch [Mil56]).

Booch behauptet, dass Software inhärent komplex ist [Boo93, S. 5]. Als Gründe führt er die Komplexität der Problemdomäne und des Entwicklungsprozesses selbst an sowie die große Flexibilität von Software im Hinblick auf die Möglichkeiten der Problemlösung und die Schwierigkeit, das Verhalten von diskreten Systemen zu beschreiben (s. dazu auch [Cou85]). Eine bewährte Strategie, um mit dieser inhärenten Komplexität umzugehen, ist die Etablierung von Struktur, sowohl in der Problemdomäne und im Entwicklungsprozess als auch im Quellcode selbst in Form von Kapselung (vereinfacht die Beschreibung) und Mustern (beschränken die Flexibilität). Damit wären Komplexität und Struktur aber zwei Seiten derselben Medaille: Komplexität ist die Abwesenheit von Struktur. Der Grad der Strukturierung spielt bei vielen der in diesem Kapitel aufgeführten Qualitätsuntereigenschaften wie bspw. der Analysierbarkeit eine zentrale Rolle und wird dort mit entsprechenden Merkmalen abgeschätzt, die Komplexität ist damit in gewisser Weise schon Teil des Qualitätsmodells. Eine explizite Berücksichtigung der Komplexität ist neben der Subjektivität der Bewertung und der schwierigen Definition auch aus diesem Grund nicht sinnvoll. Trotz dieser Schwierigkeiten gibt es Versuche, die Komplexität als Merkmal mit in die Bewertung aufzunehmen. Dazu gehört die Betrachtung der Anzahl möglicher linear unabhängiger Pfade durch eine Funktion oder Methode, wie sie bspw. durch die zyklomatische Komplexität von McCabe erfolgt. Sie hat sich als ein Indikator für risikobehaftete Stellen im Quellcode bewährt. Die Bezeichnung dieses Maßes deutet schon auf die zugrunde liegende Hypothese hin, dass eine hohe Pfadanzahl gleichgesetzt wird mit einer hohen wahrgenommenen Komplexität und entsprechenden Häufung von Fehlern.

5.5 Basis des Qualitätsmodells

5.5.1 Zerlegungssystematik

Die Zerlegungssystematik für das Qualitätsmodell basiert auf dem FCM-Ansatz, der jedoch entsprechend der Definition des Qualitätsbegriffs in ISO 9126-1 und dem Modell von Wallmüller (s. [Wal01, S. 46ff]) erweitert wird, um die in den vorigen Abschnitten aufgestellten Hypothesen einordnen zu können (s. auch Abb. 5.5 auf S. 84):

- Ziel ist die interne Produkt*qualität*.

- Diese lässt sich bewerten anhand der *Eigenschaften* Wartbarkeit und Portabilität.
- Jede dieser Eigenschaften kann weiter aufgeschlüsselt werden in verschiedene *Untereigenschaften* wie Analysierbarkeit oder Stabilität.
- Jede Untereigenschaft wird geprägt von einem oder mehreren *Merkmalen*.
- Die Ebene der *abgeleiteten Merkmale* umfasst Entwurfsregeln wie geringe Kopplung, hohe Kohäsion und schmale Schnittstellen. Sie bestimmen zusammen mit dem Softwaremodell die Messaufgaben.
- Die *Kenngrößen* zu den abgeleiteten Merkmalen können schließlich aus dem Quellcode ermittelt und anhand der zugehörigen *Wertebereiche* interpretiert werden.

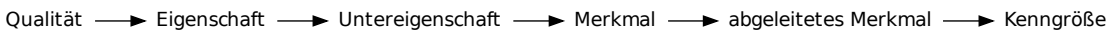

Abbildung 5.5: Zerlegungssystematik für das Qualitätsmodell.

Die abgeleiteten Merkmale und damit auch die Kenngrößen werden teilweise bestimmt durch verschiedene *Einflussfaktoren*, die abhängig sind von spezifischen im Projekt getroffenen Entscheidungen. Dazu gehört die Wahl der Programmiersprache und der Entwurfsmethode (objektorientiert, funktional etc.), u. U. auch die Wahl des Komponentenmodells. Dies hat Auswirkungen insbesondere auf die Strukturierungsmöglichkeiten, die Blickwinkel und die verfügbaren Kenngrößen. Bevor eine vollständige Definition des Qualitätsmodells gegeben werden kann, muss daher das zugehörige Softwaremodell vorliegen. Einige der aufgestellten Hypothesen greifen dem schon vor, indem sie Annahmen zu den Einflussfaktoren machen, z. B. wird die Testbarkeit beeinflusst von der Vererbung. Dies ist allerdings ein normaler Auswahlprozess, der immer im Hinblick auf das Untersuchungsobjekt erfolgt, um eine möglichst realitätsnahes Qualitätsmodell zu erhalten. Im Fall der in dieser Arbeit untersuchten Fallstudie sind als Einflussfaktoren sowohl die Entwurfsmethode (objektorientiert) als auch die Programmiersprache (Java) relevant.

5.5.2 Zuordnung der Hypothesen

Die Zerlegung der Eigenschaften Wartbarkeit und Portabilität wurde in den Abschnitten 5.3.1 bzw. 5.3.2 erläutert. Es folgt die Zuordnung der konkreten und der davon abgeleiteten Merkmale zu den Untereigenschaften, um den ersten Teil des Qualitätsmodells zu vervollständigen.

Analysierbarkeit Ein in der Hypothese aufgeführtes Merkmal ist der Umfang von Entitäten. Damit ist die Größe feingranularer Elemente wie Methoden und Klassen gemeint, aber auch die Anzahl der Elemente pro Struktureinheit. Letzteres wird hier als Hierarchisierung bezeichnet. Die Anzahl der Verbindungen pro Element wird bestimmt durch die Kopplung. Neben der Anzahl ist auch die Ausgestaltung feingranularer Elemente, speziell der Umfang möglicher Pfade, für die Bewertung von Belang.

Die Programmiersprache als externer Einflussfaktor definiert die Ausgestaltungen des Quellcodes. In diesem Fall betrifft dies die Möglichkeiten der Strukturierung, insbesondere die Erzeugung von Hierarchien: Die Vererbung, die Organisation von syntaktischen Elementen in Klassen und Paketen sowie die Komposition von Komponenten.

Der Einfluss von redundantem Code auf die Analysierbarkeit wird nicht nur die Anzahl der Redundanzen, sondern auch durch ihren jeweiligen Umfang bestimmt.

Änderbarkeit Die Hypothese zur Änderbarkeit zielt ab auf die Strukturierung, genauer: die Modularität des Quellcodes. Sie wird bestimmt durch die Kohäsion der einzelnen Elemente und durch ihre Schnittstellen. Relevant sind die Komplexität der Methodenschnittstellen, die Anzahl der Verbindungen zu anderen Elementen über die Schnittstelle (=Kopplung) und die Breite der gesamten Klassenschnittstelle (=Anzahl öffentlicher Variablen und Methoden).

Sowohl der Umfang als auch die Redundanz der von einer Änderung betroffen Code-Bereiche haben Einfluss auf die Änderbarkeit.

Stabilität Sowohl die Bewertung der Korrektheit als auch Robustheit sind nur eingeschränkt möglich. Für Exceptions kann ein Vergleich der Anzahl gefangener und weitergeleiteter Ausnahmen Hinweise auf die Robustheit geben. Die Korrektheit hingegen kann abgeschätzt werden durch Assertions, die die eingehenden Parameter einer Methode prüfen.

Testbarkeit Die Testbarkeit ist abhängig vom Grad der Autonomie der Elemente und der Vererbungstiefe und sind damit Teil der Modularität.

Portabilität und ihre Untereigenschaften Die Portabilität geht nur über die Untereigenschaft der Anpassungsfähigkeit in das Qualitätsmodell ein, die auf dieselbe Weise bewertet wird wie die Änderbarkeit. Eine nährere Untersuchung dieser Untereigenschaft ist damit nicht erforderlich.

Zusammenfassung Abschnitt 5.5

Auf Basis der Zerlegungssystematik wurden die in den Hypothesen aufgeführten Bewertungshinweise bis zur Ebene der abgeleiteten Merkmale zugeordnet. Abbildung 5.6 zeigt eine Darstellung dieser Zerlegung. Damit ist der erste Teil des Qualitätsmodells abgeschlossen.

Aus den abgeleiteten Merkmalen lassen sich nun zusammen mit dem Softwaremodell, das die besonderen Eigenschaften der Programmiersprache berücksichtigt und die relevanten Strukturierungsmöglichkeiten herausstellt, die Messobjekte und -aufgaben bestimmen. Die Zuordnung von Grenzwerten und eine Berechnungsvorschrift, mit der die Werte zu Aussagen über einzelne Merkmale kombiniert werden können, vervollständigen das Qualitätsmodell.

5.6 Messaufgaben und -objekte

Die im Rahmen des Messprozesses definierten Ziele und die daraus abgeleiteten Messaufgaben definieren die Ausgestaltung des ersten Teils des Qualitätsmodells. Der zweite Teil besteht aus

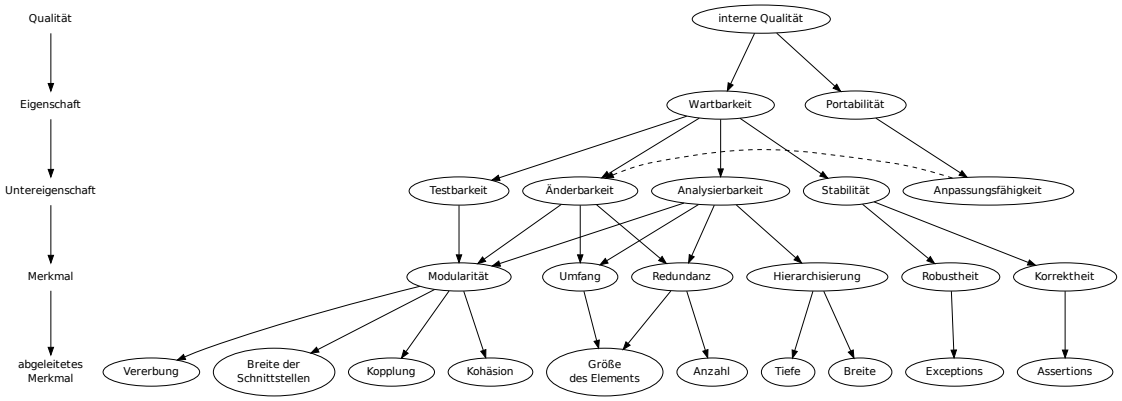

Abbildung 5.6: Basis des Qualitätsmodells.

den konkreten Messaufgaben und den Maßen, mit deren Hilfe Kenngrößen für die Bewertung ermittelt werden können. Die von der Programmiersprache vorgesehenen Strukturierungselemente bestimmen dabei die zugehörigen Messobjekte. Die Interpretation der Kenngrößen beschließt dann die Entwicklung des Qualitätsmodells.

5.6.1 Das Softwaremodell

Softwaremodelle oder Abstraktionen von Programmen sind die Grundlage jeder statischen Analyse von Quellcode und haben die Aufgabe, die im Kontext der jeweiligen Untersuchung relevanten Attribute herauszustellen. Dabei beeinflusst sowohl die verwendete Programmiersprache als auch das Messziel die Menge dieser Merkmale.

Ein Programm bzw. dessen Quellcode spiegelt eine bestimmte Idee einer Abfolge von Aktionen wider. Diese Abfolge kann sich nur der Mittel bedienen, die das der Sprache zugrunde liegende Abstraktionsmodell zur Verfügung stellt. Ein Beispiel ist die Verwendung von konditionalen Bedingungen oder Schleifen. Frühe Sprachen wie die verschiedenen Assembler-Dialekte halten sich mit ihrem Abstraktionsmodell sehr eng an die unterliegende Hardware; der Kapselungsgedanke findet sich nur in rudimentärer Form, bspw. als benannte Code-Abschnitte. Moderne Sprachen besitzen ein Typenkonzept, kennen Schleifen und Konditionalausdrücke, und können mit Hilfe von Funktionen bzw. Prozeduren eine Sichtbarkeitsbeschränkung von Elementen definieren. Damit kann das Information Hiding und der Modularisierungsgedanke von Parnas praktisch umgesetzt werden (s. [Par72]).

Die zur Implementierung der Fallstudie verwendete Programmiersprache gehört zu den objektorientierten Sprachen. Das für die Bewertung benötigte Softwaremodell muss daher die durch dieses Paradigma eingeführten Strukturierungsmöglichkeiten berücksichtigen. Der nachfolgende Abschnitt bietet in einen kurzen Überblick die relevanten Aspekte (s. dazu bspw. [Amb01]), bevor ihre Umsetzung in Form der Sprache Java betrachtet wird (nach [Krü06]).

Objektorientierung

Die grundlegende Idee der objektorientierten Weltsicht ist, dass Probleme durch interagierende Agenten gelöst werden können, die jeweils einen kleinen Teil zur Gesamtlösung beitragen. Der Vorteil im Vergleich zur Strukturierten Analyse bzw. Programmierung besteht in der feineren problemorientierten Dekomposition des Gesamtsystems im Sinne des Separation of Concerns, die zu einem besseren Systemverständnis und in Folge zu einer besseren Entwicklung und Wartung führen soll.

Die Agenten werden auch *Objekte* genannt und sind Instanzen von *Klassen*. Eine Klasse abstrahiert von konkreten Objekten, indem sie Daten (=*Attribute*) und Funktionen (=*Methoden*) kapselt und über eine definierte *Schnittstelle* für andere Objekte in einem bestimmten Umfang (=Information Hiding) verfügbar macht. Die Sichtbarkeit (public, private, protected) bestimmt dabei, welche Elemente von außen erreichbar sind. Sie ist damit eine Art Blaupause für konkrete Objekte. Untereinander kommunizieren Objekte über *Nachrichten*, die auch eine Nutzlast transportieren können.

Klassen stehen in zwei Arten von Beziehungen zueinander: Vererbung und Verwendung. Die *Vererbung* nutzt die Tatsache, dass es in den meisten Problemdomänen hierarchische Ableitungsstrukturen gibt, die durch die Phrase „ist ein“ („is a“) beschrieben werden können, z. B. „ein Hund ist ein Tier“. Auf diese Weise können gemeinsame Merkmale in abgeleiteten Klassen wiederverwendet und so der Implementierungsaufwand reduziert werden. Attribute und Methoden können in einer erbenden Klasse auch überschrieben werden, um das Verhalten weiter zu spezifizieren. Nach Liskov sollte die Vererbung immer so erfolgen, dass es bei der Verwendung egal ist, auf welcher Ebene der Vererbungshierarchie eine zu einer Instanz gehörenden Klasse angesiedelt ist (s. [Lis87]). Allerdings wird diese Möglichkeit der Wiederverwendung auch zur Implementierungsvererbung genutzt, die keine Spezialisierung mehr darstellt.

Das Vererbungskonzept ermöglicht *Polymorphismus*, d. h. Instanzen von Klassen einer Vererbungshierarchie können auf die gleiche Nachricht (potentiell unterschiedlich) reagieren. Welche Instanz nun tatsächlich die Nachricht erhält, wird erst zur Laufzeit entschieden, weshalb hier auch von *Late Binding* gesprochen wird.

Eine Klasse kann von einer anderen Klasse für die Erzeugung lokaler Instanzen oder als Nachrichtenargument verwendet werden. Diese Verwendungsform nennt man *Assoziation*. Eine Klasse kann auch aus mehreren Klassen aufgebaut sein, es besteht also eine „ist Teil von“-Beziehung („part of“). Haben die Teile-Klassen eine eigene Daseinsberechtigung, spricht man von *Aggregation*, teilen sie hingegen ihren Lebenszyklus mit der zusammengesetzten Klasse, so nennt man diese eine *Komposition*. Die meisten Programmiersprachen setzen diesen feinen Unterschied nicht um.

Instanzen von Klassen und damit die Daten des Systems existieren zunächst nur zur Laufzeit und werden bei Programmende gelöscht. Um zu einem späteren Programmlauf auf diese Instanzen zugreifen zu können, müssen sie *persistent* gemacht werden. Bei Sprachen, die keinen Image-basierte Persistenz beherrschen wie bspw. Smalltalk (s. [LP90]), geschieht dies meist mit Hilfe von Dateien oder Datenbanken. Eine *Serialisierung* bringt die Instanzen dazu in eine speicherbare Form. Neben den Instanzen selbst müssen auch die Beziehungen zwischen ihnen gesichert werden.

Klassen definieren eine Schnittstelle, indem sie bestimmte Attribute und Methoden als öf-

fentlich deklarieren. Eine Klasse kann aber auch eine gegebene Schnittstelle implementieren, um bestimmte Anforderungen zu erfüllen, wie sie bspw. die Verwendung der Klasse in einem Container mit sich bringt. Schnittstellen sind also nicht nur ein Effekt des Information Hiding von Klassen, sondern stellen ein wichtiges, eigenständiges Strukturierungsinstrument dar.

Die Objektorientierung selbst macht keine Angaben zur Organisation von Klassen über die Vererbung und Verwendung hinaus. Besonders bei großen Systemen ist aber eine weitere Abstraktionsebene erforderlich, eben besagte Architekturebene. Etabliert hat sich hier der *Komponentenbegriff*. Eine Komponente ist im Bereich der objektorientierten Entwicklung eine modulare, erweiterbare und separat auslieferbare Einheit, die klar definierte Schnittstelle besitzt und Abhängigkeiten explizit ausweist, sofern vorhanden. Sie besitzen im Kontext der Problemlösung eine für den Anwender verständliche Bedeutung. Komponenten können zu Subsystemen und diese wiederum zu einem Gesamtsystemen assembliert werden (s. dazu auch [Szy99, S. 132ff]). Wie diese Integration zu erreichen ist und welche Details dabei zu beachten sind, ist allerdings völlig offen.

Oft wird statt Komponente der Begriff *Modul* verwendet. Tatsächlich gibt es große Ähnlichkeiten zwischen beiden, allerdings dienen Module vorwiegend zur Organisation von Quellcode mit dem Ziel, eine dezentrale Entwicklung und Kompilation zu ermöglichen. Sie besitzen über hart kodierte Konstanten keine weiteren unveränderlichen Bestandteile. Komponenten hingegen können weitgehend parametrisiert werden, indem man die ihnen mitgegebenen Ressourcen verändert; dies erfordert dann aber keine erneuten Compiler-Lauf.

Objektorientierung mit Java

Die Programmiersprache Java implementiert eine Entwicklung auf Basis von Dateien, die mittels eines Compilers in Bytecode übersetzt werden, der auf einer virtuellen Maschine, der *Java Virtual Machine*, ausgeführt werden kann. Dieser Ansatz ist bis auf den generierten Bytecode identisch mit der Entwicklung von Programmen in C oder C++. Die Verwendung einer virtuellen Maschine ähnelt dem Ansatz bspw. von Smalltalk, allerdings wird dort statt der dateibasierten Entwicklung ein Image-Konzept verfolgt.

Als objektorientierte Sprache implementiert Java das Klassen- und Objektkonzept so wie im vorigen Abschnitt beschrieben. Eine Klasse wird definiert mit dem Schlüsselwort `class`, dem ein Sichtbarkeitsparameter vorangestellt werden muss. Dieser hat Auswirkungen auf die Sichtbarkeit der Klasse außerhalb des Pakets, in dem sie definiert wird. Per Konvention sollte jede Klasse in einer eigenen Datei definiert werden. Eine Instanz einer Klasse kann mit dem Schlüsselwort `new` unter Angabe des Klassenbezeichners erzeugt werden. Die Übergabe von Objekten an Methoden erfolgt generell per *Referenz*. Das bedeutet, dass in den die Nachricht empfangenen Methoden immer auf dem Originalobjekt gearbeitet wird. Besonders bei der Verwendung von Containern kann dies zu Missverständnissen führen

Die Attribute und Methoden werden durch Angabe eines Sichtbarkeitsparameters und des Typs definiert. Ihre Verfügbarkeit für die Vererbung kann außerdem mit `final` begrenzt und mit `static` unabhängig vom Lebenszyklus einer Instanz gemacht werden. Statische Attribute und Methoden heißen im Java-Jargon *Klassenvariablen*. Normale Instanzvariablen (Attribute) werden zusammen mit dem Objekt erzeugt und sind an dessen Lebensdauer gebunden. Klassenvariablen/-methoden hingegen existieren unabhängig von Objekten, können also ohne

vorherige Instantiierung verwendet werden.

Eine Methode kann auf alle Attribute und Methoden seiner Klasse und der Superklasse zugreifen. Attribute und Methoden der Superklasse müssen dazu als `public` oder `protected` deklariert sein. Innerhalb einer Methode können Instanzen aller im Paket bekannt gemachten Klassen erzeugt werden. An eine Methode übergebene Objekte müssen ein Teil der Parameterliste sein. Innerhalb einer Methode ist eine Strukturierung in Form von *Blöcken* möglich, die wiederum aus *Anweisungen* bestehen. Ein Block begrenzt die Lebensdauer der in seinem Bereich angelegten lokalen Variablen und deren Sichtbarkeit. In Blöcken können durch Konditionalausdrücke wie `if-then-else` und `switch` verzweigte Programmpfade formuliert und mit `while` und `for` Schleifen beschrieben werden.

Java definiert zwei zusätzliche Klassenarten, *lokale* und *anonyme* Klassen. Klassen werden in einer Datei definiert, d. h. die Definition erfolgt auf Paketebene (s. u.). Seit Version 1.1 können zusätzlich eingebettete Klassen, sog. lokale Klassen, definiert werden (Java bezeichnet sie als *Inner Classes*). Es handelt sich dabei um normalen Klassen, die allerdings innerhalb einer vorhandenen Klasse oder auch Methode definiert werden können. Die Methoden dieser Klasse können auf alle Attribute der umgebenden Klasse zugreifen. Lokale Klassen können auch anonym sein, d. h. der Bezeichner entfällt bei der Definition. Anonyme Klassen werden häufig in Methoden verwendet, sozusagen als „Einwegklasse“, da sie nur einmal instanziert werden kann. Die Lesbarkeit solcher Konstruktionen war und ist Gegenstand vieler Diskussionen, auch weil sie den Programmierern die Möglichkeit gibt, „mal eben“ eine benötigte Klasse zu definieren, ohne im Vorfeld über den Entwurf nachdenken zu müssen. Vorteilhaft ist jedoch die erwirkte Flexibilität, die sich bspw. bei der Definition von *Listenern* im Kontext des Event Handling bewährt hat.

Java implementiert eine Einfachvererbung, d. h. es kann bei einer Spezialisierungsbeziehung nur eine Superklasse angeben werden. Im Vergleich zu C++ erscheint dies zunächst als Beschränkung. Diese Sprache verwendet den Vererbungsmechanismus aber auch für die Formulierung und Implementierung von Schnittstellen. Dazu bietet Java die Möglichkeit, Schnittstellen mit dem Syntaxelement `Interface` *explizit* zu konstruieren. Eine Klasse kann beliebig viele dieser Schnittstellen implementieren. Schnittstellen können auch an andere Schnittstellen vererbt werden. Klassen können mit dem Schlüsselwort `final` gegen das Ableiten von neuen Klassen geschützt werden. Attribute mit diesem Modifier werden zu Konstanten, d. h. sie können nicht verändert werden. Derart geschützte Methoden dürfen in abgeleiteten Klassen nicht überlagert werden.

Die Klasse `Object` ist die Basisklasse aller in Java definierten Klassen und muss bei der Definition nicht explizit angegeben werden. Diese Vererbungsbeziehung ermöglicht eine einfache Implementierung von Containern wie Vektoren, Stacks, Listen etc. Diese Container heißen in Java *Collections*. Die Verwendung der obersten Basisklasse hat allerdings einen Nachteil: Um auf die Klasse des Objekts und damit auf die Spezialisierung zugreifen zu können, muss ein Cast stattfinden. Dieser ist nicht typsicher. Seit J2SE 5.0 gibt es ein an die C++-Templates angelehnten Mechanismus namens *Generics*, der parametrisierbare Container definiert, z. B. `List<String>`. Damit sind Container typsicher und Casts unnötig. Im Gegensatz zu C++ wird dabei aber kein zusätzlicher Code erzeugt.

Die Behandlung von Ausnahmen ist kein originäres Element der Objektorientierung, wird in Java aber mit Hilfe von Klassen umgesetzt und sind damit eine weitere Form der Assoziation.

Durch die `catch-or-throw`-Regel lassen sich „explizite“ und „implizite“ Ausnahmen unterscheiden. Explizite Ausnahmen müssen bei der Klassendefinition an den entsprechenden Methoden mit dem Schlüsselwort `throws` aufgeführt werden. Sie weisen damit auf potentiell geworfene Ausnahmen hin. Implizite Ausnahmen werden innerhalb einer Methode mit einem `try-catch`-Block abgefangen und behandelt. Der Typ der Ausnahme muss im Kopf des `catch`-Blocks aufgeführt werden, ist aber außerhalb der Methode nicht relevant.

Java ein standardisiertes Komponentenmodell. *Java Beans* sind eigenständige Bausteine, die unabhängig von Hardware und Betriebssystem plattformübergreifend verwendet werden können. Beans zeichnen sich dadurch aus, dass sie einen parameterlosen Konstruktor besitzen und (de-)serialisierbar sind. Letzteres wird durch die Implementierung der Schnittstelle `Serializable` erreicht. Das Paket `java.beans` stellt viele für die Implementierung von Beans hilfreichen Klassen und Schnittstellen bereit. Einen *eindeutigen* Hinweis im Quellcode, dass eine Klasse eine Bean implementiert, gibt es jedoch nicht. Ein Teil der Standardisierung bezieht sich auf Namenskonventionen, die eine Abschätzung erlauben, bspw. bei der Registrierung von Methoden zur Eigenschaftsverwaltung. Einzig ein Eintrag in der Manifest-Datei der Form

```
Name BeanClassFile.class
Bean: True
```

weist eine Klasse explizit als Bean aus. Diese Datei ist aber kein Teil des eigentlichen Quellcodes. Damit bleibt als charakteristisches Merkmal für die Identifizierung nur die Implementierung der Serialisierungsschnittstelle. Der ursprüngliche Anwendungsbereich von Beans war die Modellierung von graphischen Schnittstellen mit Hilfe eines GUI-Builders, sie haben sich jedoch auch für den Einsatz in JSPs und Servlets als sehr nützlich erwiesen. Die Implementierung der Serialisierungsschnittstelle ist für die Verwendung in Servlets allerdings nicht zwingend notwendig [DD04, S. 1302], was eine eindeutige Identifizierung einer Klasse als Komponente allein aus dem Quellcode faktisch unmöglich macht. Die Integration von Komponenten zu Subsystemen bzw. die Umsetzung einer Strukturierung auf dieser Ebene erfolgt damit ohne direkte Unterstützung durch entsprechende Konstrukte der Programmiersprache. Eine Rekonstruktion könnte mit Hilfe von Techniken wie die Dominanz- und Clusteranalyse versucht werden, die auf die Lokalität bzw. die Detektion von eng gekoppelten oder einander ähnlichen Einheiten abzielen. Aus den dabei ermittelten Clustern kann dann möglicherweise auf höhere Strukturen wie Schichten geschlossen werden. Voraussetzung dafür ist jedoch eine im Quellcode vorhandene, diesbezüglich „sinnvolle“ Kopplung und ein Kopplungsgrenzwert, der die Zugehörigkeit einer Einheit zu einem Cluster definiert, beides projektspezifische Aspekte. Im Rahmen einer allgemeinen Analyse können die den in Abschnitt 5.3.1 auf Seite 69 aufgeführten Wertebereiche für die Kopplung als eine erste Annäherung an die vom Entwickler intendierte Clusterung dienen.

Alternativ zu einer Integration auf Komponentenbasis kann eine Strukturierung des Gesamtsystems auch mit Hilfe einer Pakethierarchie erfolgen. Angesichts mangelnder Alternativen dürfte das wohl der Regelfall sein, zumindest werden Pakete in vielen einschlägigen Werken zur Java-Programmierung als *das* Organisationswerkzeug oberhalb der Klassenebene geführt. Für die Organisation von Klassen bietet Java die Möglichkeit, selbige in *Paketen* zusammen zu fassen. Pakete selbst können in eine Pakethierarchie eingegliedert werden. Die Organisation der Pakete zeigt sich auch auf Dateiebene: Jeder Teil eines Paketnamens repräsentiert ein

Unterverzeichnis, z. B. sind Klassen des Pakets `com.sun.image` im Verzeichnis `/com/sun/image` zu finden. Über das Schlüsselwort `import` können Pakete vollständig oder Klassen aus diesem Paket explizit verfügbar gemacht werden. Importe sind nicht transitiv, d. h. eine in einem Unterpaket importierte Klasse ist durch einen Import des Unterpakets nicht für andere Klassen verfügbar. Der Compiler muss immer einen voll qualifizierten Bezeichner ermitteln können.

Pakete besitzen keinen zu den Klassen vergleichbaren eigenen Schnittstellen-Mechanismus, sondern definieren nur einen Namensraum. Die Kriterien für die Zugehörigkeit einer Klasse sind denn auch völlig beliebig: „Ein Paket ist eine Sammlung von Klassen, die einen gemeinsamen Zweck verfolgen oder aus anderen Gründen zusammengefasst werden.“ [Krü06, S. 295]. Die Zuordnung der Klassen und Pakete der Java-Klassenbibliothek zeigen eine logische Kohäsion, d. h. eine Zuordnung erfolgt aufgrund einer als verwandt gesehenen Funktionalität. Ein Beispiel: Sämtliche Klassen, die etwas mit der Ein-/Ausgabe zu tun haben, gehören zum Paket `java.io`. Pakete erzeugen über ihren Namensraum allerdings eine Kapselung. Eine Klasse darf eine anderen verwenden, sie entweder im selben Paket definiert sind oder die zu verwendende Klasse als `public` deklariert wurde. Die „Schnittstelle“ eines Pakets könnte demnach als die Menge der öffentlichen Klassen aufgefasst werden. Wird kein Namensraum mittels `package` definiert, liegt die Definition einer Klasse im `Default`-Paket. Globale Variablen oder Funktionen gibt es daher in Java nicht. Pakete entsprechen im Wesentlichen den schon beschriebenen Modulen; sie helfen, die Mengen von Klassen überschaubar zu halten. Die Orientierung der Paketnamen am DNS-System des Internet deutet schon an, dass auch die dezentrale Entwicklung großer Projekte beim Entwurf der Sprache berücksichtigt wurde [Krü06, S. 301]. So lassen sich im Vorfeld Namenskollisionen vermeiden.

Seit Version 1.4 besitzt Java ein Schlüsselwort zur Implementierung von Assertions namens `assert`. Dem Schlüsselwort muss ein Ausdruck folgen, der entweder zu `True` oder `False` ausgewertet werden kann. Die Umsetzung von Assertions ist im Vergleich zu der von Meyer vorgestellten Form im Kontext des *Design by Contract* (s. [Mey97]) eher rudimentär, stellen aber dennoch ein nützliches Hilfsmittel dar.

Java bietet eine Reihe weiterer Mechanismen und APIs an, besonders für die Implementierung großer Systeme in Unternehmen (J2EE). Dazu gehören Schnittstellen u. a. für den Datenbank-Zugriff, Remote Method Invocation, verteilte CORBA-Objekte, Directory Services wie Active Directory oder OpenLDAP, verteilte Transaktionen, Enterprise Messaging etc. (s. bspw. [FFC99; Joh02]). Mit ihrer Hilfe lässt sich die klassische Schichtenarchitektur komfortabel umsetzen. Für eine automatische Ermittlung von Qualitätsmerkmalen können sie als projektspezifischer Einflussfaktor allerdings nicht im Detail berücksichtigt werden.

Zuordnung der Messobjekte und -aufgaben zu abgeleiteten Merkmalen

Den abgeleiteten Merkmale werden nun mögliche Messobjekte zugeordnet. Diese erhalten wiederum im Kontext des Merkmals sinnvolle und durch das Messobjekt ermöglichte Messaufgaben zugeteilt. Das Ziel ist, die Voraussetzungen zu bestimmen, um anschließend aus der Menge der bekannte Softwaremaße eine mögliche Auswahl zu treffen.

Die Herleitung der Messaufgaben ergibt sich aus der Kombination der verschiedenen Strukturierungsmöglichkeiten der Programmiersprache und den zu ermittelnden Aspekten der abgeleiteten Merkmale. Nachfolgend werden zu jedem Merkmal die für eine Ermittlung relevan-

ten Messobjekte auf den verschiedenen Abstraktionsebenen betrachtet, die die Sprache bietet, bevor im daran anschließenden Abschnitt eine vollständige Übersicht vorgestellt wird.

Breite der Schnittstelle Als Messobjekte kommen nur Strukturelemente in Frage, die entweder eine eigene Schnittstelle besitzen (Methoden und Klassen) oder für sich eine Schnittstelle herleiten lässt (Pakete). Subsysteme besitzen ebenfalls eine Schnittstelle, oft in Form einer Fassade (s. [Gam+96, S. 212]). Die zuverlässige automatische Erkennung von Subsystemen ist jedoch schwierig, daher werden sie hier nicht als Messobjekt berücksichtigt. Ähnlich verhält es sich mit Komponenten. Legt man die Implementierung der Serialisierungsschnittstelle als Merkmal zugrunde, so ist die Breite der Komponentenschnittstelle identisch mit der der Klasse. Sie braucht also nicht separat berücksichtigt werden.

Kopplung Eine Methode ist definitionsgemäß immer an die sie definierende Klasse gekoppelt. Damit bestehen potentiell weitere Verbindungen zu Attributen und Methoden in ihren verschiedenen Ausprägungen. Kopplungen zu anderen Klassen erfolgen durch die Benennung von Klassen als Typ in der Parameterliste, durch die Erzeugung Instanzen von Klassen innerhalb einer Methode oder über die Definition einer lokalen Klasse.

Bei Klassen entsteht Kopplung zunächst durch die Zugehörigkeit zu einem Paket und weiter über die Deklaration von Attributen oder die Definition lokaler Klassen. Da das Verhalten einer Klasse auch von ihrer Superklasse abhängt (soweit vorhanden), besteht auch hier eine Abhängigkeitsbeziehung und damit eine Kopplung. Diese Abhängigkeit gibt es auch bei Schnittstellen untereinander, die ebenfalls Teil einer Vererbungshierarchie sein können. Die Implementierung durch Klassen sind eine weitere Kopplungsform. Die Abhängigkeit einer Methode bzw. Klasse von einer Klasse kann auch zu einer Abhängigkeit von einem „Fremd“-Paket führen, wenn erst der Import diese Klasse verfügbar macht.

Die Kopplung von Komponenten untereinander ist abhängig davon, wie ein System sie integriert. Die Verwendung eines Application Servers zur Bean-Integration erlaubt keine Rückschlüsse auf die Beziehungen zu anderen Komponenten. Eine Ermittlung der Kopplung ist daher nicht ohne Zusatzwissen möglich; das gilt ebenso für die Kopplung von Subsystemen.

Pakete definieren über die `import`-Anweisung eine Abhängigkeitsbeziehung zu anderen Paketen. Weiterhin gilt, dass eine Abhängigkeitsbeziehung aus Sicht des Gesamtsystems transitiv ist, d. h. instabile Pakete sind (in-)direkt abhängig von stabilen Paketen (s. [Mar02, S. 261f]). Aus diesem Grund ist die Position eines Pakets innerhalb dieses Abhängigkeitsgraphen von Interesse. Zusammen mit der Abstraktheit kann eine Überprüfung hinsichtlich des Stable-Abstraction-Prinzips erfolgen (s. [Mar02, S. 264ff]).

Kohäsion Die verschiedenen Formen der Kohäsion, wie sie u. a. von Yourdon und Constantine beschrieben wurden ([YC79]), untersuchen die Beziehung von Elementen auf der Ebene von Funktionen bzw. Prozeduren. Da in der Objektorientierung die Klasse das eigentliche Modellierungselement ist, steht ihre Kohäsion bei der Bewertung im Vordergrund. Ermittelt wird die Kohäsion anhand der Methoden und deren Zugriffe auf disjunkte Attributmengen der Klasse. Die Kohäsion von Komponenten kann aufgrund ihrer zur Klasse identischen Struktur auf dieselbe Weise ermittelt werden.

Eine Übertragung dieses Ansatzes auf die Paketebene ist nicht möglich. Die Bewertung der Kohäsion von Paketen ist nach Auffassung von Martin abhängig vom gewünschten Grad der Wiederverwendung [Mar02, S. 254ff]. Dementsprechend beschreiben die von ihm aufgeführten drei Prinzipien REP, CRP und CCP Kriterien, anhand derer Klassen in Paketen organisiert werden sollten. Inwieweit ein System diesen Prinzipien Folge leistet, ist aus dem Quellcode allerdings nicht zu ermitteln. Da auch die Anzahl der die Wiederverwendung nutzenden Client-Klassen nicht vorherzusehen ist, kann für Pakete keine zuverlässige Kohäsionsbestimmung vorgenommen werden.

Vererbung Die Bewertung der „Güte“ einer Vererbung im Sinne des Liskovschen Ersetzungsprinzips ist aufgrund der semantischen Bedeutung der Vererbung in der Problemdomäne nicht möglich. Hinweise können allerdings die Zahlen geerbter, eigener und überdeckter Attribute bzw. Methoden liefern. Darüber hinaus ist die Anzahl von Kindklassen und die Position der Klasse in der Vererbungshierarchie relevant, um die Stabilität der Klasse zu bewerten. Dazu gehört auch die Anzahl finaler Attribute und Methoden. Besonders wichtig ist die Vererbung über Paketgrenzen hinweg, wobei sie für Pakete selbst keine Rolle spielt. Bis auf die Ermittlung der Attribute und finalen Bestandteile sind die Messaufgabe für Schnittstellen identisch.

Größe der Entität Der Analyseaufwand steht in Relation zum Umfang eines Messobjekts und lässt sich recht einfach in Zeilen erfassen. Für die Bewertung des Umfangs einer Methode ist zusätzlich die Anzahl möglicher Pfade wichtig. Der Umfang von Klassen wird ebenso anhand ihrer Zeilenanzahl bestimmt; relevant ist außerdem die Anzahl der Attribute und Methoden. Für Schnittstellen kann nur die Anzahl von Methoden ermittelt werden. Die Größe eines Pakets kann sinnvoll nur in Form von Zeilen, Klassen oder Schnittstellen angegeben werden.

Anzahl Redundanz kann auf verschiedenen Strukturierungsebenen vorkommen. Die mögliche Spannbreite reicht von nahezu identischen Blöcken (vergleichsweise geringer Umfang) über Methoden bis hin zu kompletten Klassen (umfangreiche Redundanz), die kopiert und nur marginal verändert werden (die entspräche einem Umgehen der Vererbung).

Breite und Tiefe von Hierarchien Die Vererbung als eine Möglichkeit der hierarchischen Strukturierung wird als eigenes abgeleitetes Merkmal separat betrachtet (s. o.). Eine weitere statische Hierarchie bildet sich durch die möglichen Zugehörigkeiten von Elementen, die Java bietet: Ein Block gehört zu einer Methode, die wiederum einer Klasse oder Schnittstelle (hier geht allerdings nur die Schnittstelle der Methode ein) zugeordnet ist usw. (s. dazu Abb. 5.7 auf S. 94). Lokale Klassen müssen auch berücksichtigt werden, solange sie einen Bezeichner tragen. Anonyme Klassen fallen damit weg, was angesichts ihres geringen Einflusses auf die Struktur unproblematisch ist. Die Elemente dieser Hierarchie definieren jeweils einen Sichtbarkeitsbereich und bestimmen die Lebensdauer der eingeschachtelten Elemente und besitzen optional eine Schnittstelle. Für eine Bewertung ist die Breite und Tiefe, also die *Morphologie* dieser Dekompositionshierarchie, von Interesse. Da Blöcke keine Bezeichner tragen, beginnt die Untersuchung der Hierarchie auf der Ebenen der Methoden. Aus demselben Grund werden auch Attribute nicht mit aufgeführt.

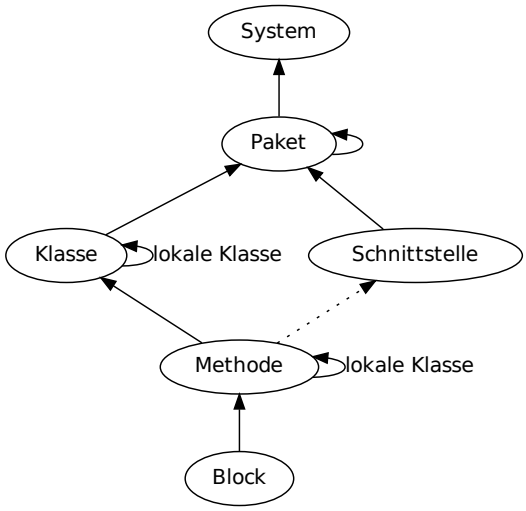

Abbildung 5.7: Dekomposition in Java auf Basis der Zugehörigkeit.

Assertions und Exceptions Als Messobjekte für die Ermittlung kommen in Java nur Methoden in Frage, da für sie eine Schnittstelle mit Parametern definiert werden kann. Relevant ist der Grad der Abdeckung bei der Überprüfung eingehender Parameter durch Assertions und die Anzahl gefangener und weitergeleiteter Exceptions.

5.6.2 Bestimmung der Messaufgaben

Mit Hilfe der Zuordnung können nun die durch die Messobjekte ermöglichten Messaufgaben bestimmt werden. Der nachfolgende Baum ordnet jedem abgeleiteten Merkmal die relevanten Messobjekte und die sich daraus ergebenden Messaufgaben zu. Das Zeichen # wird dabei stellvertretend für das Wort „Anzahl“ benutzt. Bei der Bestimmung kommt es zu inhaltlichen Überschneidungen, weil die zugehörigen Sprachkonstrukte Einfluss auf mehrere Merkmale haben. Die einzelnen Ebenen haben die Bedeutung *abgeleitetes Merkmal—Messobjekt—Messaufgabe*, ggf. ergänzt um weitere Hinweise.

- Breite der Schnittstelle
  - Methode
    - #Parameter der Signatur
    - #spezifizierter Ausnahmen
  - Klasse (Komponente)
    - #öffentlicher Attribute
    - #öffentlicher Methoden
  - Paket
    - #öffentlicher Klassen
    - #öffentlicher Schnittstellen

- Kohäsion
  - Klasse (Komponente)
    - #disjunkter Attributmengen

- Kopplung
  - Methode (lokale Kopplung)
    - #verwendeter Klassenvariablen
    - #verwendeter Klassenmethoden
    - #verwendeter Instanzvariablen
    - #verwendeter Instanzmethoden
    - #lokaler Klassen
    - #verschiedener Klassen in Parameterliste
    - #verschiedener instanzierter Klassen
  - Methode (paketweite Kopplung)
    - #verschiedener Klassen aus Fremdpaketen in Parameter-Liste
    - #verschiedener instanzierter Klassen aus Fremdpaketen
    - #verwendeter Klassenvariablen aus Fremdpaketen
    - #verwendeter Instanzvariablen aus Fremdpaketen
  - Klasse
    - #Instanzvariablen einer Klasse
    - #Klassenvariablen einer Klasse
    - #Klassenvariablen einer Klasse aus Fremdpaket
    - #lokaler Klassen
    - Superklasse (falls Vererbung vorliegt) . . . . . Boolscher Wert
    - #implementierter Schnittstellen
    - #implementierter Schnittstellen aus Fremdpaketen
  - Schnittstelle
    - #implementierender Klassen
    - Superschnittstelle (falls Vererbung vorliegt) . . . . . Boolscher Wert
  - Paket
    - #importierter Pakete
    - Abstraktheit . . . . . Wert aus dem Intervall $[0, 1]$
    - Stabilität . . . . . Wert aus dem Intervall $[0, 1]$

- Anzahl
  - Methode
    - # und Umfang redundanter Blöcke
  - Klasse
    - # und Umfang redundanter Methoden
  - Paket
    - # und Umfang redundanter Klassen

- Vererbung
  - Klasse
    - Position in Vererbungshierarchie
    - #eigener Attribute
    - #geerbter Attribute
    - #überdeckter Attribute
    - #eigener Methoden
    - #geerbter Methoden
    - #überdeckter Methoden
    - #finaler Attribute
    - #finaler Methoden
    - #Kindklassen
    - #paketübergreifender Vererbung
  - Schnittstelle
    - Position in Vererbungshierarchie
    - #eigener Methoden
    - #geerbter Methoden
    - #überdeckter Methoden
    - #Kindschnittstellen
    - #paketübergreifender Vererbung

- Größe des Elements
  - Methode
    - #Zeilen
    - #möglicher Pfade
  - Klasse
    - #Zeilen
    - #Attribute
    - #Methoden
    - #Zeilen eines redundanten Blocks
    - #redundanter Methoden
  - Schnittstelle
    - #Anzahl Methoden
  - Paket
    - #Zeilen
    - #Klassen
    - #abstrakter Klassen
    - #Schnittstellen
    - #redundanter Klassen

- Morphologie
  - Methode (ermittelt für jede Hierarchieebene)
    - #Zeilen
  - Klasse
    - #Methoden
  - Schnittstelle
    - #Methoden
  - Paket
    - #Klassen
    - #abstrakter Klassen
    - #Schnittstellen
    - #Pakete
  - System
    - #Klassen
    - #Pakete

- Assertions und Exceptions
  - Methode
    - #geprüfter Parameter
    - #gefangener Ausnahmen
    - #weitergeleiteter Ausnahmen

5.6.3 Zuordnung und Kombination von Grenzwerten

Die Suprema und Richtlinien für die Bewertung der aus den Messaufgaben bestimmten Kenngrößen basieren auf den in Abschnitt 5.3.1 auf Seite 69f. hergeleiteten Grenzwerten. Demnach besteht für alle strukturrelevanten Merkmale ein Intervall $[0, 9]$ mit 0 als natürlichem Infimum und 9 als Indikatorgrenzwert; das Teilintervall $[3, 7]$ beschreibt einen akzeptablen Wertebereich für Kenngrößen. Die Angaben zu den übrigen Merkmalen werden nachfolgend beschrieben.

Im Detail gilt für die einzelnen Messaufgaben:

- Die Größe von Methoden sollte weniger als 30 Zeilen betragen.
- Die Kohäsion von Klassen sollte möglichst maximiert und entsprechend die Anzahl disjunkter Attributmengen minimiert ($<$) werden.
- Die Anzahl möglicher Pfade sollte < 9 sein.
- Redundanzen sollten möglichst nicht auftreten, daher wird hier der Wert 0 angestrebt.
- Die Kenngrößen bei der Vererbung überschneiden sich, es gilt aber als Richtwert die in der jeweiligen Klasse bekannte Menge von Attributen bzw. Methoden. Damit gilt für die Bestandteile von Klassen:

  $$\text{\#geerbter Attr./Meth.} + \text{\#eigene Attr./Meth.} - \text{\#überdeckter Attr./Meth.} < 9$$

  Die Vererbung über Paketgrenzen hinweg sollte allerdings minimal sein.

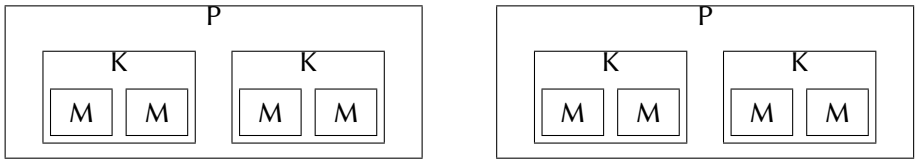

Abbildung 5.8: Elementzugehörigkeiten eines Pakets P, dargestellt in Form eines geschachtelten Baums.

- Die lokale Kopplung sollte weniger als 9 Verbindungen ausweisen. Das gilt auch für die Kopplung über Paketgrenzen hinweg.
- Die Abstraktheit und Stabilität eines Pakets liegt immer zwischen 0 und 1. Sie hängt ab von der Position der Klassen in der Vererbungshierarchie und der Kopplungsrichtung der Pakete. Als Richtlinie gilt: Die Stabilität eines Pakets *A* sollte immer größer sein als die Stabilität eines Pakets *B*, das von *A* abhängt. Für die Abstraktheit gilt: Ein Paket sollte so abstrakt wie stabil sein.
- Die Anzahl der in einer Methode gefangenen im Vergleich zur Anzahl der weitergeleiteten Exceptions sollte möglichst groß sein, da sie ein Hinweis auf die Robustheit der Klasse ist.
- Die Bewertung von Assertions kann nur vor dem Hintergrund des Defensive-Programming-Ansatzes erfolgen, d. h. es ist ein Abgleich der Parameterliste und etwaiger zugehöriger Assertions notwendig. Ein sinnvoller Grenzwert kann hier aber nicht angegeben werden, so dass die Bewertung nur nominal erfolgt (Assertion vorhanden oder nicht).

Bei der Bewertung der Morphologie steht die Gleichmäßigkeit der Baumstruktur im Vordergrund, es wird also auf eine möglichst ausgewogene Verteilung der bei der Dekomposition von Java-Quellcode entstehenden Elemente auf die Strukturierungsebenen abgezielt. Je Teilbaum sollten maximal 9 Elemente vorliegen, also 9 Methoden je Klasse, 9 Klassen je Paket etc. Die Zugehörigkeit von Elementen lässt sich gut als geschachtelter Baum darstellen (s. Abb. 5.8; M=Methode, K=Klasse, P=Paket). Für die Bewertung der Morphologie ist auch die Schachtelungstiefe relevant, die besser an einer klassischen Baumstruktur abzulesen ist (s. Abb. 5.9 auf der nächsten Seite).

Die Bewertung ergibt sich aus der Abweichung jedes Teilsbaums von einer idealen Struktur. Die Abweichung eines Teilbaums $t \in T$ ergibt sich aus:

$$A(t) = |1 - \frac{n}{c}|$$

mit n: Anzahl der Kindknoten, und c: angenommener idealer Schachtelungsfaktor. Je kleiner $A(t)$, desto gleichmäßiger ist die Strukturierung des Teilbaums. Für die mittlere Gesamtabweichung der Baumstruktur T gilt dann:

$$A(T) = \frac{\sum_i |1 - \frac{n_i}{c}|}{t}$$

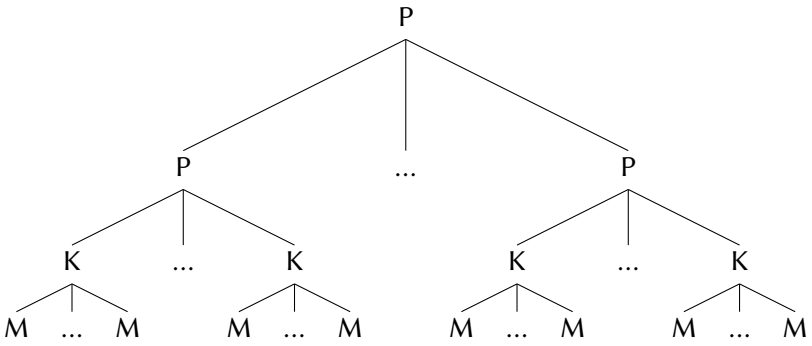

Abbildung 5.9: Elementzugehörigkeiten eines Pakets P, dargestellt als klassische Baumstruktur.

mit n_i: Kindknoten des Teilbaums t_i von T mit Tiefe 1, und t: Anzahl aller Teilbäume von T. Eine mittlere Gesamtabweichung von ≥ 1 bedeutet, dass einzelne Teilbäume um mehr als c vom idealen Wert abweichen.

5.6.4 Interpretation von Kenngrößen

Die Grenzwerte nehmen bei der Bewertung die Rolle einer Indikatorfunktion ein. Das dazu benötigte Intervall ergibt sich aus dem jeweiligen angegebenen Maximalwert und dem Nullpunkt der zugehörigen Skala (hier immer 0). Liegt ein Wert x nun außerhalb eines solchen angegebenen Intervalls T, so ist er ein Indikator für eine potentiell risikobehaftetes Messobjekt:

$$x_T = \begin{cases} 1, & x \in T \\ 0, & \text{sonst} \end{cases}$$

Damit ist allerdings noch nicht die Frage beantwortet, wie die Ergebnisse der einzelnen Messaufgaben zu einer Aussage bzgl. des abgeleiteten Merkmals kombiniert werden können. Nahezu alle etablierten Maße nutzen verschiedene Rechenoperationen, um eine Bewertung abgeleiteter Merkmale zu erreichen. Problematisch ist, dass dabei immer mehrere Kenngrößen verschiedener Messobjekte und Merkmale kombiniert werden und nicht immer klar ist, welchen Einfluss die einzelnen Werte haben. Ein Beispiel: Die Berechnung des Kopplungsfaktors (CF) berücksichtigt die Kopplung zwischen Klassen inkl. der Kopplung durch Vererbung, ignoriert aber Vererbung über Paketgrenzen hinweg. Allgemein sollten geringe Kopplungswerte angestrebt und grenzüberschreitende Vererbung vermieden werden. Die Vererbung, eine allgemein sinnvolle Beziehung, wirkt sich aber in hier unbestimmtem Umfang ungewollt negativ auf die Bewertung aus.

Bestechend einfach hingegen ist der Ansatz von Lanza und Marinescu. Sie verwenden eine sehr einfache und elegante Form der Kombination auf Basis logischer Verknüpfungen, die sie *Detection Strategy* nennen [LM06, S. 49]:

> „A *Detection Strategy* is a composed logical condition, based on metrics, by which design fragments with specific properties are detected in the source code."

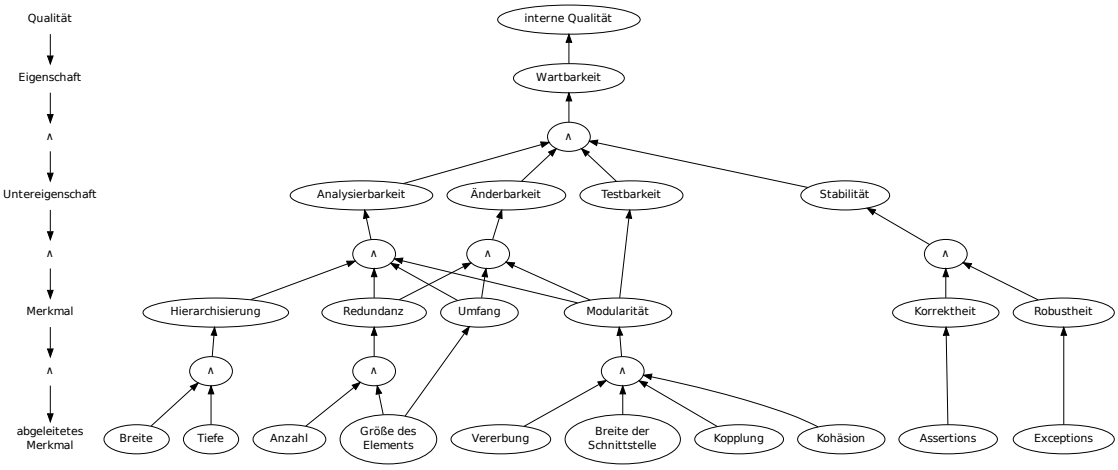

Abbildung 5.10: Kombination der Kenngrößen zu abgeleiteten Aussagen.

Praktisch bedeutet das, dass die Ergebnisse der Indikatorfunktionen eines abgeleiteten Merkmals mit Hilfe eines logischen Operators wie $\wedge$ und $\vee$ kombiniert werden. Ein Beispiel: Die Interpretationsergebnisse der Kenngrößen zur Morphologie werden mittels $\wedge$ verknüpft. Das bedeutet: Solange alle Kenngrößen in einem „guten“ Bereich liegen, ist die Hierarchie des vermessenen Systems hinsichtlich der Analysierbarkeit unauffällig. Diese Kombinationsform ist auf allen Ebenen des Qualitätsmodells anwendbar, so dass schließlich eine Antwort auf die Frage nach der internen Qualität der Art „in Ordnung“ oder „nicht in Ordnung“ möglich wird. Abbildung 5.10 auf dieser Seite illustriert die Kombination anhand der Zerlegung in Abbildung 5.6 auf Seite 86 (die Abweichung der Elementanordnung ist dem verwendeten Werkzeug anzulasten, hat aber keine inhaltlichen Auswirkungen).

Zusammenfassung Abschnitt 5.6

Die Entwicklung des Qualitätsmodells, angelehnt an ein an der Programmiersprache Java orientiertes Softwaremodell, findet mit der Interpretation der Kenngrößen seinen Abschluss. Die Ermittlung von Messwerten und ihre Interpretation sind Teil eines konkreten Messprozesses, wie er in Kapitel 6 stattfindet. Der nächste Abschnitt vergleicht das Qualitätsmodell mit etablierten Softwaremaßen und beschreibt die Vor- und Nachteile aus theoretischer Sicht, bevor es im Rahmen der Fallstudie in Kapitel 6 exemplarisch angewendet wird.

5.7 Vergleich der Messaufgaben des Qualitätsmodells mit etablierten Softwaremaßen

Nahezu alle Qualitätsmodelle, die für eine Bewertung von Quellcode herangezogen werden, nutzen verbreitete Softwaremaße wie *Weighted Method/Class*, *Lack of Cohesion* oder *Response for a Class*. Die Ermittlung und Verarbeitung von Kenngrößen des in dieser Arbeit entwickelten Qualitätsmodells ist dazu vergleichsweise simpel. Es berücksichtigt jedoch die speziellen

Anforderungen der Entwickler im Rahmen von Wartungstätigkeiten, zu denen u. a. auch die Nachvollziehbarkeit der Bewertungsergebnisse gehört.

Bevor ein Vergleich beider Ansätze — der etablierten Maße mit dem entwickelten Qualitätsmodell — vorgenommen werden kann, müssen zunächst die maßtheoretischen Grundlagen, Basis aller Softwaremaße, beschrieben werden. Ihnen folgt eine kurze Vorstellung der relevanten Maße. Für einen Vergleich kommen nur Softwaremaße in Frage, deren Eignung empirisch nachgewiesen wurde (s. dazu u. a. [BWL99], [Kan02, S. 339ff], [Agg+06]). Neben den obligatorischen *Lines of Code* kommen dafür Maße aus Metrik-Suiten der folgenden Arbeiten in Betracht:

- Chidamber und Kemerer ([CK94])
- Martin ([Mar02])
- McCabe ([McC76])
- MOOD ([HCN98])
- Lorenz und Kidd ([LK94])
- Li und Henry ([LH93])

5.7.1 Maßtheoretische Grundlagen

Aus der Sicht der Maßtheorie, wie sie von Zuse in die Softwareentwicklung eingeführt wurde (s. [Zus85]), besteht beim Messen folgender mathematischer Sachverhalt: Auf der einen Seite gibt es einen empirischen Betrachtungsbereich, auf der anderen Seite einen durch mathematische Hilfsmittel definierbaren Zahlenbereich. Messen bedeutet nun, dass es einen *Homomorphismus* zwischen dem empirischen und dem Zahlenbereich gibt, also eine *strukturerhaltende Abbildung*, die die Relationen des Messbereichs im Zahlenbereich widerspiegelt (zu Homomorphismen s. bspw. [Fis08]). Das wiederum bedeutet:

- es existiert eine Verhältnis-Relation wie „größer als“
- es gibt eine Additionsoperation
- der Wahrheitswert ist gegenüber einer Transformationen invariant, d. h. eine Darstellung in einer anderen Einheit bleibt ohne Auswirkungen

Viele der etablierten Metriken verletzen diese mathematischen Implikationen und weisen daneben noch weitere Defizite auf [Zus98, S. 42ff]. Dazu gehören fehlerhafte und mehrdeutige Interpretationen der Messwerte, fehlende Skalenangaben und damit unzulässige Zusammenfassung und Interpretation kombinierter Messwerte, sowie *Simpsons Paradoxon* (Verfälschung durch Umrechnung in Prozentzahlen).

Eine weitere geforderte Eigenschaft von Maßen beschreibt Pressman mit dem Begriff „wholeness“, den er folgendermaßen erklärt (s. [PI94, S. 365]; der Komplexitätsbegriff ist in diesem Zusammenhang nicht wichtig): Sei $C(x)$ die *empfundene Komplexität* eines Problems und $E(x)$ der für die Lösung benötigte Zeitaufwand. Dann gilt:

$$C(p_1) > C(p_2) \Rightarrow E(p_1) > E(p_2)$$

Allerdings haben Untersuchungen gezeigt, dass für die empfundene Komplexität auch gilt:

$$C(p_1 + p_2) > C(p_1) + C(p_2)$$

woraus für den Zeitaufwand folgt:

$$E(p_1 + p_2) > E(p_1) + E(p_2)$$

Der Wert der Kombination der Messobjekte ist *nicht* gleich der Summe der Einzelwerte. Zuse verallgemeinert den Ansatz und formuliert für die Komplexität von Software:

$$u(p_1 \circ p_2) > u(p_1) + u(p_2)$$

mit u als Softwaremaß. Das Beispiel zeigt anschaulich, dass ein allzu sorgloser Umgang mit Kenngrößen unter Umständen zu fatalen Fehlinterpretationen führen kann. Allerdings relativiert Fahrmeir die strenge Auslegung der Skalenzugehörigkeit [FPT02, S. 17]. Als Beispiel führt er die Berechnung von Durchschnittsnoten an. Noten sind ordinalskaliert, d. h. sie können unter Anwendung einer Ordnungsrelation in eine Reihenfolge gebracht werden. Für diese Skala ist jedoch eine Quotientenbildung eigentlich nicht zulässig; erst verhältnisskalierte Merkmale können so verwendet werden. Die Durchschnittsnote lässt sich aber sinnvoll als vergleichende Maßzahl interpretieren, weshalb die Berechnung durchaus Sinn macht. Entsprechend gilt für die Auswahl von Softwaremaßen für das Qualitätsmodell, dass zu jedem Maß neben den Messobjekten auch eine Angabe zur Skala und damit zu den erlaubten Operationen und einer potentiellen Vergleichbarkeit gemacht werden müsste. Bei näherer Betrachtung der Messaufgaben zeigt sich, dass sich bis auf sechs Aufgaben bei jeder Messung elementare, verhältnisskalierte Werte ergeben, ein Vergleich untereinander und die Berechnung des Durchschnitts u. ä. sind also zulässig und eine Angabe der Skala erübrigt sich.

5.7.2 Beschreibung der etablierten Maße

Es folgt eine kurze Vorstellung der einzelnen Maße der etablierten Metrik-Suiten, bei der ihre Ermittlungsmethode und die Wertebereiche für die Kenngrößen beschrieben werden. Unterschieden werden können elementare Maße wie *Lines of Code* oder die Anzahl von Methoden sowie abgeleitete Maße, bspw. *Response for a Class* oder der *Method Hiding Factor*. Die Definitionen sind den angegebenen Quellen entnommen und werden ggf. näher erläutert. Gleiches gilt für die Wertebereiche, wobei fehlende Angaben durch die hergeleiteten Werte aus Abschnitt 5.3.1 ergänzt werden. Die Abkürzungen der Messobjekte stehen für: M=Methode, C=Klasse, S=Schnittstelle, P=Paket und G=Gesamtsystem, W/M=Grenzwert pro Maß.

Name	Definition	W/M	Quelle
Lines of Code (LOC)	#Zeilen	M<30 C<900 P<27000	[RL04]
Zyklom. Komplexität	$\text{CYCLO}(M) = e - n + 2p$	M<10	[McC76]
Weighted Method/Class	$\text{WMC}(C) = \sum_{n=1}^{n} c_i$	C<100	[CK94] (1)

Depth of Inheritance	DIT=Pfadlänge zur Basisklasse	C<6	[CK94] (2)
Number of Children	NOC=Anzahl direkter Kindklassen	C<9	[CK94] (3)
Coupling between Objects	$\text{CBO}(C) = \lvert K\rvert + \lvert M\rvert + \lvert A\rvert + \lvert V\rvert$	C<6	[CK94] (4)
Response for a Class	$\text{RFC}(C) = M \cup \left(\bigcup_i R_i\right)$	C<100	[CK94] (5)
Lack of Cohesion	$\text{LCOM}(C) = \lvert P\rvert - \lvert Q\rvert$, wenn|P|>|Q|; 0 sonst	C<9	[CK94] (6)
Efferent Couplings	EC=#Abhängigk. von Kl. aus Fremdpaketen	-	[Mar02]
Afferent Couplings	AC=#abhängiger Klassen aus Fremdpaketen	-	[Mar02]
Instability	$I(P) = \frac{\text{EC(P)}}{\text{AC(P)+EC(P)}}$	0<P<1	[Mar02]
Abstraction	$A(P) = \frac{N_a}{N_c}$	0<P<1	[Mar02] (7)
Main Sequence Distance	$\text{D'}(P) = \lvert A(P) + I(P) - 1\rvert$	P<0,2	[Mar02]
Attribute Hiding Factor	$\text{AHF}(G) = \frac{\sum_{i=1}^{T}\sum_{m=1}^{A_d(C_i)}(1-V(A_{mi}))}{\sum_{i=1}^{T} A_d(C_i)}$	G>0,25	[HCN98] (8)
Method Hiding Factor	$\text{MHF}(G) = \frac{\sum_{i=1}^{T}\sum_{m=1}^{M_d(C_i)}(1-V(M_{mi}))}{\sum_{i=1}^{T} M_d(C_i)}$	G>0,25	[HCN98] (9)
Attribute Inheritance Factor	$\text{AIF}(G) = \frac{\sum_{i=1}^{T} A_i(C_i)}{\sum_{i=1}^{T} A_a(C_i)}$	G→1	[HCN98] (10)
Method Inheritance Factor	$\text{MIF}(G) = \frac{\sum_{i=1}^{T} M_i(C_i)}{\sum_{i=1}^{T} M_a(C_i)}$	G→1	[HCN98] (11)
Coupling Factor	$\text{CF}(G) = \frac{\sum_{i=1}^{T}[\sum_{j=1}^{T} \text{is client}(C_i,C_j)]}{T^2-T}$	G<0,5	[HCN98] (12)
Polymorphism Factor	$\text{PF}(G) = \frac{\sum_{i=1}^{T} M_o(C_1)}{\sum_{i=1}^{T}[M_n(C_i \times DC(C_i))]}$	G<0,5	[HCN98] (13)
Number of Methods	$\text{NOM}(C) = M_l(C)$	C<20	[LH93] (14)
Number of Attributes	$\text{NOA}(C) = A_l(C)$	C<6	[Hen95] (15)
Number of Methods overr.	$\text{NMO}(C) = M_o$	C=S<9	[Hen95] (16)
Data Abstraction Coupling	$\text{DAC}(C) = \sum_{i=1}^{T} \text{uses}\,(C, C_i)$	C=S<9	[LK94] (17)
Public Instance Methods	$\text{PIM}(C) = M_v(C)$	C<9	[LK94] (18)
Number of Instance Methods	$\text{NIM}(C) = M_d(C)$	C<20	[LK94] (19)
Number of Instance Variables	$\text{NIV}(C) = A_d(C)$	C<6	[LK94] (20)

Tabelle 5.1: Definition der verwendeten etablierten Maße.

1. Problematisch ist die Bestimmung der Menge der Methoden: Werden eigene und geerbte Methoden für die Ermittlung verwendet oder nur eigene Methoden? Oft wird aus diesem Grund nur die Anzahl der Methoden einer Klasse ohne einen Komplexitätsfaktor verwendet ($c_i = 1$) [Kan02, S. 338]. Dann entspricht WMC der Anzahl der Methoden (NOM); diesen Effekt haben Aggarwal et al. bei ihren Untersuchungen festgestellt.

2. Je tiefer die Position einer Klasse in der Vererbungshierarchie ist, desto mehr Methoden erbt sie von den Superklassen und desto schwerer ist ihr Verhalten vorherzusagen. Umso größer ist allerdings die Wahrscheinlichkeit, dass sie Attribute und Methoden wiederverwendet.

3. Je größer die Zahl der Kindklassen/-schnittstellen ist, desto eher findet Wiederverwendung statt. Bei zu großen Zahlen hingegen kann eine unpassende Abstraktion der Superklasse vorliegen.

4. K: Anzahl verwendeter Klassenvariablen, M: Anzahl verwendeter Klassenmethoden, A: Anzahl verwendeter Instanzmethoden, V: Anzahl verwendeter Instanzmethoden.
   Hohe Kopplungswerte weisen auf schlechte Modularität und Kapselung hin.

5. M: Menge aller Methoden von C, und $\bigcup_i R_i$: Menge aller Methoden, die von M aufgerufen werden.
   Der Wertebereich ist stark abhängig von der Programmiersprache, bspw. sind bei Smalltalk-Projekten wesentlich höhere Werte zu verzeichnen als bei C++-Projekten. Die hier vorgeschlagene RFC-Obergrenze sollte abhängig vom jeweiligen Projekt angepasst werden.

6. I_i: Menge aller Instanzvariablen, die eine Methode M_i nutzt, $P = \{(I_i, I_j) | I_i \cap I_j = \emptyset\}$ und $Q = \{(I_i, I_j) | I_i \cap I_j \neq \emptyset\}$.
   Niedrige LCOM-Werte weisen auf eine hohe Kohäsion der Klasse hin, d. h. die Methoden sind um gemeinsame Attribute aufgebaut. Hohe Werte weisen darauf hin, dass eine Klasse geteilt werden sollte.

7. N_a: Anzahl abstrakter Klassen in einem Paket P, N_c: Anzahl Klassen in P.

8. $A_d(C_i) = A_v(C_i) + A_h(C_i)$ mit A_v: Anzahl sichtbarer Attribute der Klasse C_i und A_h: Anzahl nicht sichtbarer Attribute von C_i;

   $$V(A_{mi}) = \frac{\sum_{j=1}^{T} visible(A_{mi}, C_{j)}}{T-1}$$

   mit $visible(A_{mi}, C_j) = 1$ wenn j≠i und C_j may reference A_{mi}; T: Anzahl aller Klassen im System.

9. $M_d(C_i) = M_v(C_i) + M_h(C_i)$ mit M_v: Anzahl sichtbarer Methoden der Klasse C_i und M_h: Anzahl nicht sichtbarer Methoden von C_i;

   $$V(M_{mi}) = \frac{\sum_{j=1}^{T} visible(M_{mi}, C_{j)}}{T-1}$$

   mit $visible(M_{mi}, C_j) = 1$ wenn j≠i und C_j may call M_{mi}; T: Anzahl aller Klassen im System.

10. $A_a(C_i) = A_d(C_i) + A_i(C_i)$: in Klasse C_i verfügbare Attribute mit $A_d(C_i) = A_n(C_i) + A_o(C_i)$: in Klasse C_i definierte Attribute, wobei A_n: neu definierte, nicht überschreibende Attribute und A_o: neu definierte, überschreibende Attribute sind; A_i: geerbte und nicht überschriebene Attribute von C_i; T: Anzahl aller Klassen im System.
    Je größer der AIF-Wert, desto mehr wird die Vererbung genutzt.

11. $M_a(C_i) = M_d(C_i) + M_i(C_i)$: in Klasse C_i verfügbare Methoden mit $M_d(C_i) = M_n(C_i) + M_o(C_i)$: in Klasse C_i definierte Methoden, wobei M_n: neu definierte, nicht überschreibende Methoden und M_o: neu definierte, überschreibende Methoden sind; M_i: geerbte und nicht überschriebene Methoden von C_i; T: Anzahl aller Klassen im System.
    Je größer der MIF-Wert, desto mehr wird die Vererbung genutzt.

12. is client$(C_C, C_S) = 1$, wenn $C_C \Rightarrow C_S$ und $C_C \neq C_S$, sonst 0, wobei $C_C \Rightarrow C_S$ für eine Beziehung zwischen der Client-Klasse C_C und der Supplierklasse C_S steht. T: Anzahl aller Klassen im System.
    Die Interpretation dieser Werte ist nicht unproblematisch, da bspw. die Kopplung via Vererbung durchaus sinnvoll und im Sinne der Wiederverwendung ist, sie aber den Kopplungfaktor auch vergrößert. Allgemein sind geringe Kopplungswerte anzustreben.

13. $M_0(C_i)$: überschreibende Methoden in der Klasse C_i, $M_n(C_i)$: neue Methode der Klasse C_i und $DC(C_i)$: Anzahl aller Nachkommen von C_i. T: Anzahl aller Klassen im System.
    Die Bewertung des PF-Maßes ist ähnlich schwierig wie die von CF: In einigen Fällen kann das Überschreiben von Methoden das Verständnis verbessern, wohingegen ein zu hoher Grad an Polymorphismus das Gegenteil bewirkt.

14. $M_l(C)$: Anzahl in Klasse C lokal definierter Methoden.

15. $A_l(C)$: Anzahl in Klasse C lokal definierter Attribute (NOA).

16. $M_o(C)$: Anzahl in Klasse C überschriebener Methoden (NMO).

17. uses $(C, C_i) = 1$, wenn die Klasse C eine Klasse C_i als Klassen- oder Instanzvariable nutzt, sonst 0. T: Anzahl aller Klassen im System.

18. $M_v(C)$: Anzahl öffentlicher Methoden einer Klasse C (PIM).

19. $M_d(C)$: Anzahl aller Methoden (public, protected, private) einer Klasse C (NIM).

20. $A_d(C)$: Anzahl aller Instanzvariablen (public, protected, private) einer Klasse C (NIV).

Es wird deutlich, dass es für jede Strukturebene, von der Methode bis hin zum Gesamtsystem, ein oder mehrere Maße gibt. Das gilt sowohl für die Bewertung der Kopplung (CBO, EC, AC, CF) als auch für die Vererbung. Letztere ist sogar Messobjekt einer systemweiten Betrachtung durch AHF, MHF, AIF und MIF. Darüber hinaus finden sich elementare Maße wie NOM, NOC etc., die teilweise für die Ableitung der komplexeren Maße verwendet werden.

So verführerisch die Verwendung eines systemweiten Maßes wie CF sein kann, so unklar ist, inwieweit sich die verschiedenen Strukturierungsmöglichkeiten von Quellcode auf die Berechnung und auf die Güte der anschließende Bewertung auswirken. Dies entspricht nicht der Forderung nach einer transparenten und durch den Entwickler nachvollziehbaren Bewertungsmethode.

5.7.3 Vergleich der Wertebereiche

Für den Vergleich kommen für die etablierten Maße die Angaben aus den jeweiligen Untersuchungen zum Einsatz, wie sie bei Lorenz ([Kan02, S. 337]; aufgrund der Ähnlichkeit von Java zu C++ werden die Aussagen zu C++ verwendet), Kan ([Kan02, S. 342]), Lanza und Marinescu ([LM06, S. 16]) und Roock und Lippert ([RL04, S. 35]) zu finden sind. Tabelle 5.2 auf der nächsten Seite zeigt eine Zusammenstellung der Angaben zu den elementaren Maßen aus der Literatur

Kenngrößen	Lorenz	L. & M.	Kan	R. & L.	eigene Werte
Anzahl Zeilen/Methode	<24	<20	-	<30	<30
Anzahl Methoden/Klasse	<20	<15	<40	<30	<9
Anzahl Instanzvar./Klasse	<6	-	-	-	<9
Anzahl Klassen/Paket	-	-	-	<30	<9
Anzahl Pakete/Subsystem	-	-	-	<30	<9
Tiefe d. Vererbungshierarchie	<6	-	-	-	<9
Subsystemkopplung	<6	-	-	-	<9
Klassenkopplung im Subsys.	max	-	<5	-	<9

Tabelle 5.2: Grenzen für die elementaren Kenngrößen.

und stellt ihnen die eigenen Werte gegenüber. Aufgeführt sind allerdings nur diejenigen Angaben, die als Zahlwerte vorliegen oder als eindeutige Grenzwerte erkennbar sind.

Der Vergleich zeigt, dass die Grenzwerte der etablierten Softwaremaße stellenweise recht konservativ angesetzt werden, ein Element also eher als auffällig in Erscheinung tritt als bei dem hier entwickelten Qualitätsmodell. Es gibt aber auch Werte, die vom jeweils selben Autor weit höher angesetzt werden. Dies betrifft hauptsächlich die Hierachisierung, also die Anzahl Methoden pro Klasse, die Anzahl Klassen pro Paket und die Anzahl Pakete pro Subsystem.

Da viele der abgeleiteten Maße auf diesen elementaren Kenngrößen aufsetzen, wird schnell deutlich, wie groß die Spannweite der dann möglichen Wertebereiche sein kann. Insgesamt zeigt sich, dass die Ebene der Architektur, in der Objektorientierung im Wesentlichen repräsentiert durch Paketstrukturen, im Vergleich zur feingranularen Ebene der Klassen und Methoden noch weiterer Untersuchungen bedarf. Die hier aufgeführten eigenen Werte dienen daher auch zunächst nur zur Orientierung, basierend auf den theoretischen Überlegungen aus Abschnitt 5.3.1.

5.7.4 Vergleich der Messaufgaben mit Softwaremaßen

Kosten-Nutzen-Betrachtung der etablierten Maßen

Der Nutzen jeder der aufgeführten Metrik-Suiten wurde in mehreren Untersuchungen für sich empirisch gezeigt. Der Einsatz aller Maße für eine Bewertung ist allerdings nicht unbedingt notwendig, da aufgrund von Überschneidungen nicht notwendigerweise zusätzliche Erkenntnisse gewonnen werden können. Aggarwal et al. haben aus den aufgeführten Suiten 22 Maße ausgewählt und sie anhand der Ergebnisse dreier Fallstudien statistisch untersucht [Agg+06]. Berechnet wurden Abweichung, Hauptkomponenten und Korrelation. Dabei konnten sie zeigen, dass teilweise dieselben Messwerte ermittelt wurden (bspw. entsprechen die Werte von WMC denen von NOM) und dass Teilmengen-Beziehungen bestanden (bspw. ist NOM Teilmenge von RFC). Abbildung 5.11 auf der nächsten Seite illustriert die gefundenen Beziehungen zwischen diesen Klassenmetriken (LCC: Loose Class Cohesion, ICH: Information based Cohesion, TCC: Tight Class Cohesion, MPC: Message Passing Coupling).

Angesichts der Leistungsfähigkeit heutiger Rechner kann der Mehraufwand einer Ermittlung aller Maße jedoch vernachlässigt werden, zumal die Fallstudien, auf die Aggarwal et al. ihre Er-

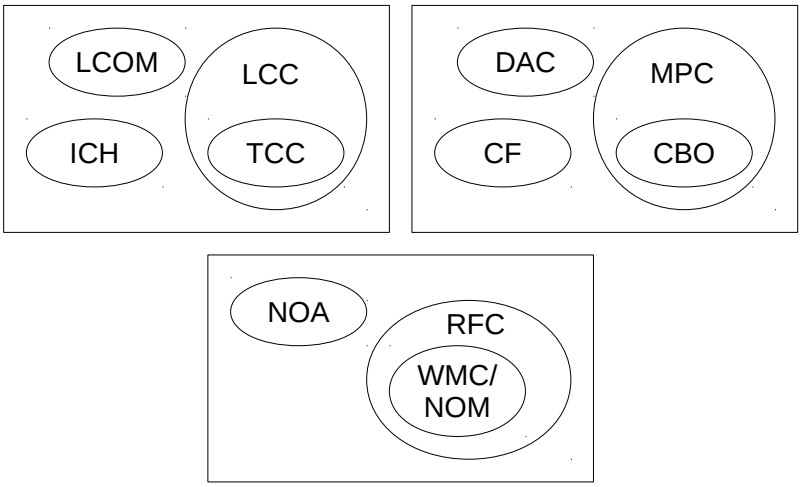

Abbildung 5.11: Beziehungen zwischen Metriken (nach [Agg+06, S. 168]).

kenntnisse stützen, von demselben Autor stammen. Die Überschneidungen könnten demnach auch ein Artefakt der Programmiereigenheiten des Autors der Fallstudien sein.

Abdeckung der Messaufgaben durch etablierte Maße

Tabelle 5.3 ordnet den Messaufgaben elementare und abgeleitete Maße aus der angegebenen Literatur zu, um den Grad der Abdeckung der Messaufgaben durch die etablierten Maße zu bestimmen. Einige der Messaufgaben, die nicht direkt abgedeckt werden, können allerdings aus vorhandenen Maßen berechnet werden, z. B. NIM-PIM=Anzahl nicht-öffentlicher Methoden (protected, private).

Messobjekt	Messaufgabe	elem. Maße	abgel. Maße
Methode	#Parameter der Signatur		
	#spezifizierter Ausnahmen		
Klasse	#öffentlicher Attribute		AHF
(Komponente)	#öffentlicher Methoden	PIM	MHF
Paket	#öffentlicher Klassen		AC
	#öffentlicher Schnittstellen		AC
Methode	#verwendeter Klassenvariablen		CBO, CF
(lokale	#verwendeter Klassenmethoden		CBO, CF, RFC
Kopplung)	#verwendeter Instanzvariablen		CBO, CF
	#verwendeter Instanzmethoden		CBO, CF, RFC
	#lokaler Klassen		CF
	#verschiedener Klassen in Parameterliste		CF
	#versch. instanzierter Klassen		CF
(paketweite	#versch. Klassen aus Fremdpak. in Param.-Liste	EC	CF, RFC
Kopplung)	#versch. instanzierter Klassen aus Fremdpaketen	EC	CF, RFC

Messobjekt	Messaufgabe	elem. Maße	abgel. Maße
	#verwendeter Klassenvariablen aus Fremdpaketen	EC	CF
	#verwendeter Instanzvariablen aus Fremdpaketen	EC	CF
Klasse	#Instanzvariablen einer Klasse	NIV, NOA	CF, DAC
	#Klassenvariablen einer Klasse	NOM	CF, DAC
	#Klassenvariablen einer Klasse aus Fremdpaket		CF, DAC
	#lokaler Klassen		CF, DAC, CF
	Superklasse (falls Vererbung vorliegt)		CF,
	#implementierter Schnittstellen		CF,
	#implementierter Schnittst. aus Fremdpaketen		CF,
Schnittstelle	#implementierender Klassen		CF,
	Superschnittstelle (falls Vererbung vorliegt)		CF,
Paket	#importierter Pakete		CF, D’
	Abstraktheit		A, D’
	Stabilität		D’, I
Klasse (Komponente)	#disjunkter Attributmengen	LCOM	
Klasse	Position in Vererbungshierarchie	DIT	
	#eigener Attribute	NOA	
	#geerbter Attribute		AIF, CF
	#überdeckter Attribute	NOA	PF
	#eigener Methoden	NOM	
	#geerbter Methoden		MIF, CF
	#überdeckter Methoden	NMO, NOM	MHI, PF
	#finaler Attribute	NOA	
	#finaler Methoden	NOM	
	#Kindklassen	NOC	
	#paketübergreifender Vererbung	EC	CF
Schnittstelle	Position in Vererbungshierarchie	DIT	
	#eigener Methoden	NOM	
	#geerbter Methoden		CF, MIF
	#überdeckter Methoden	NMO, NOM	MHF, PF
	#Kindschnittstellen	NOC	
	#paketübergreifender Vererbung	EC	CF
Methode	#Zeilen	LOC	
	#möglicher Pfade	CYCLO	WMC
Klasse	#Zeilen	LOC	
	#Attribute	NOA, NIV	
	#Methoden	NOM, NIM	
	#Zeilen eines redundanten Blöcke		
	#redundanter Methoden		
Schnittstelle	#Anzahl Methoden	NOM, NIM	
Paket	#Zeilen	LOC	

Messobjekt	Messaufgabe	elem. Maße	abgel. Maße
	#Klassen		
	#abstrakter Klassen		
	#Schnittstellen		
	#redundanter Klassen		
Methode	# und Umfang redundanter Blöcke		
Klasse	# und Umfang redundanter Methoden		
Paket	# und Umfang redundanter Klassen		
Methode	#Zeilen	LOC	
Klasse	#Methoden	NOM, NIV, PIM	
Schnittstelle	#Methoden	NOM	
Paket	#Klassen		A
	#abstrakter Klassen		
	#Schnittstellen		
	#Pakete		
System	#Klassen		
	#Pakete		
Methode	#geprüfter Parameter		
	#gefangener Ausnahmen		
	#weitergeleiteter Ausnahmen		

Tabelle 5.3: Zuordnung der Maße zu Messaufgaben und -objekten.

Java-spezifische Strukturen wie lokale Klassen, Assertions und Exceptions, aber auch die Redundanzaspekte, werden nicht durch die etablierten Maße berücksichtigt. Schnittstellen können durch ihre Nähe zu Klassen zwar indirekt erfasst werden (z. B. NOM), ihrer Bedeutung für die Struktur eines Softwaresystems wird das aber nicht gerecht. Beides darf aber nicht verwundern, da die Softwaremaße keine derartigen sprachspezifischen Aspekte erfassen können. Trotzdem kann das nicht die Lücke verbergen, die sich bei der Ermittlung der Morphologie auftut. Die dritte Spalte macht zudem nochmals deutlich, wie vielfältig die in die Berechnung eingehenden Grundwerte sein können (z. B. CF). Abbildung 5.12 auf der nächsten Seite zeigt die Zuordnung der Maße anhand der Zerlegung aus Abschnitt 5.5, angelehnt an Abbildung 5.6 auf Seite 86.

Vor- und Nachteile beider Ansätze

Wie schon zu Beginn von Abschnitt 5.7 auf Seite 100 angedeutet positioniert sich das in dieser Arbeit entwickelte Qualitätsmodell nicht als Konkurrent zu bestehenden Ansätzen, die die in den vorigen Unterabschnitten beschriebenen Softwaremaße verwenden, sondern als Ergänzung. Das gilt sowohl für die Ermittlung und Verarbeitung von Kenngrößen als auch für die Bewertung nicht erfasster Messaufgaben wie Redundanz, Assertion- und Ausnahmebehandlung sowie der Morphologie.

Die etablierten Maße haben den Vorteil, dass sie ihre Tauglichkeit bei der Bewertung von Strukturen in mehreren Untersuchungen gezeigt haben. Gerade die für die Bewertung der

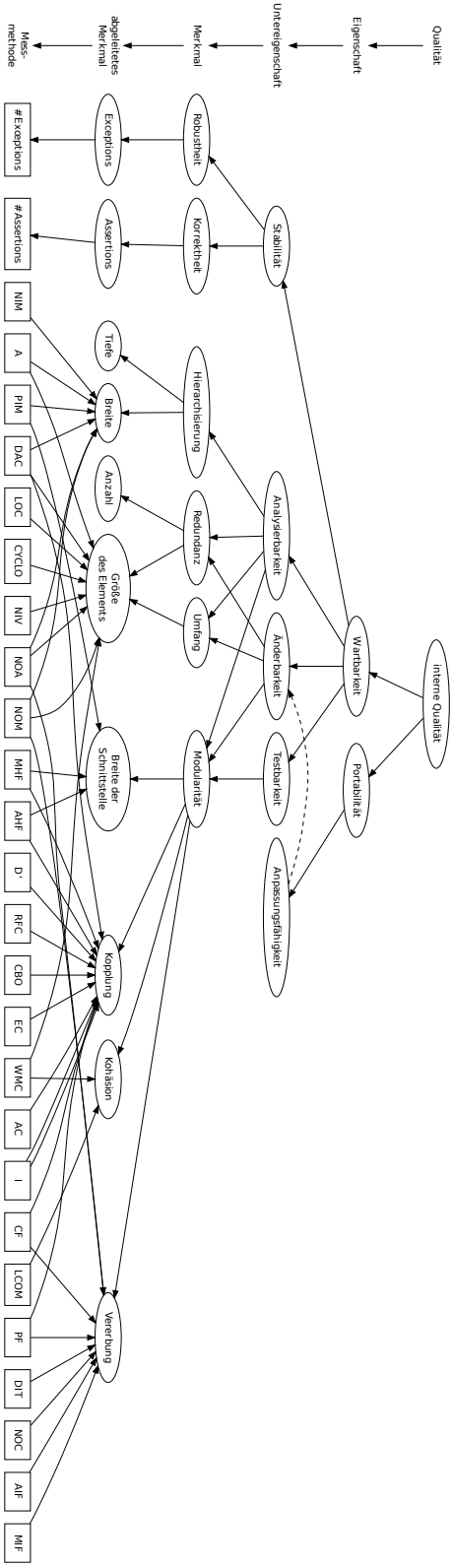

Abbildung 5.12: Qualitätsmodell Teil 2.

Wartbarkeit wichtigen Merkmale *Modularität* und *Vererbung* werden auf verschiedenen Strukturebenen erfasst und sogar systemweit bewertet. Eine am jeweiligen Projekt ausgerichtete Kalibrierung vorausgesetzt bilden sie ein relativ zuverlässiges Instrumentarium, um problematische Elemente und Strukturen zu identifizieren.

Gerade auf der Architekturebene zeigen sich jedoch für den Entwickler auch Nachteile. Eine Bewertung soll ihm als Orientierung dienen, die entsprechend als schlecht bewerteten Bereiche zu finden und zu korrigieren. Dazu ist es allerdings erforderlich, dass klar wird, *welche* Elemente bzw. Strukturen in *welchen Maße* für das Bewertungsergebnis verantwortlich sind. Speziell für einige abgeleitete Maße ist das nicht der Fall. Dazu kommt, dass sich eine Bewertung am Kenntnisstand des Entwicklers ausrichten sollte. Dies lässt sich durch Veränderung der Grenzwerte relativ problemlos realisieren. Es ist dann aber wiederum nicht eindeutig ersichtlich, wie sich die Bewertung der Strukturen im Detail ändert.

Das in dieser Arbeit entwickelte Qualitätsmodell versucht, diesen Schwächen der etablierten Modelle durch einen vergleichsweise einfachen Messansatz zu begegnen. Dieser Ansatz erlaubt nicht nur eine unmittelbar nachzuvollziehende Einflussnahme der einzelnen Kenngrößen, er ist auch so flexibel, dass die Grenzwerte prinzipiell auf jeder Strukturierungsebene für jedes Element individuell angepasst werden können. Dadurch können viele False-Positive-Ergebnisse, wie sie bei der Verwendung bestimmter Entwurfsmuster auftreten können (bspw. hohe Kopplungswerte bei Singleton und Listener), vermieden und die besonderen Eigenheiten einzelner Subsysteme berücksichtigt werden. Durch die Verwendung verhältnisskalierter Werte entsteht außerdem eine Vergleichbarkeit der Elemente innerhalb eines Projekts, die bspw. bei der Verrechnung mit einer Fehlerrate hilfreich ist, um Grenzwerte anzupassen. Der Einsatz einer Indikatorfunktion für die Bewertung vermeidet zudem eine Reihe von Problemen vieler Maße, wie sie Zuse beschrieben hat (s. [Zus98, S. 42ff]). Sie kann aber trotzdem die Kernaussage eines Prädiktors transportieren, was für die Suche nach Auffälligkeiten im Quellcode ausreicht. Die Kombination der Aussagen durch logische Operatoren ist dann nur eine konsequente Fortführung dieses Gedankens.

Ein Alleinstellungsmerkmal des Qualitätsmodells ist die Integration der Redundanz und der Stabilität als konstituierende Eigenschaften der Wartbarkeit. Die Redundanz ist eine sprachunabhängige Eigenschaft, Assertions und Exceptions sind schon länger Teil jeder verbreiteten objektorientierten Sprache. Daher verwundert es etwas, dass diese beiden Eigenschaften von den etablierten Softwaremaßen bisher nicht berücksichtigt werden. Auch die Hierarchisierung, Resultat der Dekomposition eines Systems und damit maßgeblich verantwortlich für die bedarfsgesteuerte Top-Down-Analyse, wird nicht entsprechend berücksichtigt. Weiterhin ermöglicht das Qualitätsmodell die Integration der etablierten Softwaremaße in den Messprozess mittels der *Detection Strategy* von Lanza und Marinescu (s. Abschnitt 5.6.4 auf Seite 99). Die Kenngrößen erlauben zunächst nur die Detektion eines potentiell risikobehafteten Messobjekts, nicht aber eine Abschätzung des Risikos. Dies kann aber ein vorhandenes abgeleitetes Maß. Zwar ist aufgrund der Ordinalskalierung der berechneten Werte nur eine bedingte Vergleichbarkeit möglich, aber sie können die Reihenfolge der weiteren Untersuchungen der Messobjekte bestimmen.

Es muss an dieser Stelle nochmals betont werden, dass sämtliche ermittelten Werte und deren Interpretationen ausschließlich als Indikatoren für mögliche Probleme dienen und keinesfalls absolute Aussagen liefern können. Sollten sich also Auffälligkeiten zeigen, muss durch eine

Untersuchung der betroffenen Abschnitte im Quellcode eine Bestätigung erfolgen.

5.8 Zusammenfassung

Auf Basis der in Abschnitt 4.4 beschriebenen Ergebnisse wurde nach dem von Wallmüller vorgeschlagenen Prozess ein Qualitätsmodell entworfen, das sich an den etablierten Standards ISO 9126 und IEEE 1471 orientiert und diese anhand der Erkenntnisse aus der Untersuchung des Wartbarkeitsbegriffs erweitert.

Die Erweiterungen erstrecken sich sowohl auf die Menge der zu ermittelnden Merkmale — hier ist besonders die Stabilität und die Testbarkeit zu nennen — als auch auf die Grenzwerte der einzelnen elementaren Kenngrößen. Die Verwendung logischer statt algebraischer Operationen zur Herleitung höherwertiger Qualitätsaussagen vermeidet unzulässige Berechnungen und verbessert die Nachvollziehbarkeit der Aussagen, da der Entwickler den Einfluss einzelner Kenngrößen direkt erkennen und ggf. wirksame Gegenmaßen einleiten kann.

Die Strukturierungsmöglichkeiten der Objektorientierung und ihrer Umsetzung in Form der Sprache Java und die Modularität haben im Qualitätsmodell einen besonderen Stellenwert: Die hierarchische Zuordnung von Elementen — sowohl durch Vererbung als auch durch Pakete — sollte nicht nur einer einheitlichen Dekompositionsstrategie folgen, sondern auch entsprechend ihrer Stabilität und unter Berücksichtigung von *Scope of Control* und *Scope of Effect* organisiert werden (s. dazu [Mar02, S. 261] und [YC79, S. 178]), um den Impact von Änderungen möglichst gering zu halten. Die Bestimmung der Morphologie eines Systems und eine anschließende Abbildung der Kopplung der einzelnen Bestandteile kann hierzu wertvolle Hinweise liefern.

Der Vergleich des Qualitätsmodells mit den etablierten Maßen macht deutlich, dass das Modell i. a. kleinere Grenzwerte ansetzt. Die Beschränkung auf elementare, verhältnisskalierte Kenngrößen vermeidet außerdem die von Aggarwal et al. ([Agg+06]) beschriebenen Nebeneffekte. Aggregierte Aussagen zur Qualität eines System wie der Kopplungsfaktor (CF) kämpfen zudem immer mit Informationsverlusten, die durch die Verdichtung der beteiligten Kenngrößen verursacht werden. Die dabei entstehende Diskrepanz zur tatsächlichen Ausprägung der einzelnen Merkmale lässt sich gut an der Fallstudie zeigen, deren Morphologie exemplarisch im nachfolgenden Kapitel untersucht wird.

6 Analyse des BIS-Quellcodes

> Programs must be written for people to read, and only incidentally for machines to execute.
>
> *(Abelson & Sussman, „Structure and Interpretation of Computer Programs“)*

Die Entwicklung des Qualitätsmodells erfolgte mit Blick auf ein Refactoring des BIS. In diesem Kapitel soll nun exemplarisch ein abgeleitetes Merkmal aus dem Quellcode ermittelt und bewertet werden. Die Untersuchung erfolgt in vier Phasen:

1. Beschreibung des Messgegenstands
2. Auswahl der Messwerkzeuge und Bestimmung der Messaufgaben vor dem Hintergrund des entwickelten Qualitätsmodells
3. Ermittlung der zu den Messaufgaben gehörenden Kenngrößen
4. Interpretation der Kenngrößen und Folgerungen

Die durch den Messgegenstand vorgegebenen Einflussfaktoren grenzen sowohl die Auswahl der Messaufgaben als auch die Werkzeugpalette ein. Da das Qualitätsmodell im Wesentlichen elementare Kenngrößen verwendet, stehen für die Ermittlung mehrere Messwerkzeuge zur Verfügung. Die Interpretation der Kenngrößen und die Bewertung des abgeleiteten Merkmals hingegen muss selbst entwickelt werden.

6.1 Das BIS

6.1.1 Historie

Die Beschreibung orientiert sich am Tagungsbeitrag „Ein Campus-Management-System als evolutionäre Entwicklung - Ein Erfahrungsbericht“ von Henning Brune et al. [Bru+09]. Demnach entstand das „Bielefelder Informationssystem“ (BIS) aus dem Bedarf heraus, die Überschneidung von Veranstaltungen für die Studierenden zu reduzieren. Die ersten Arbeiten dazu begannen 1998 im Kontext der Lehramtsausbildung, die besonders mit diesem Problem zu kämpfen hat. Der Kern des BIS, das *elektronische kommentierte Vorlesungsverzeichnis* (eKVV) vereint die vorher von den einzelnen Fakultäten und Lehrstühlen auf eigenen Webseiten bereit gestellten Veranstaltungsbeschreibungen und bietet Lehrenden wie Studierenden einen

einheitlichen Zugang zu diesen Informationen über das Internet. Mit der Zeit ist der Funktionsumfang des Systems stetig größer geworden. Nach Einführung der Prüfungsverwaltung als letzte großen Neuerung im Jahr 2006 wird es 2011 die Möglichkeit geben, die an der Universität Bielefeld existierenden Studiengänge zu modellieren und damit eine automatische Notenberechnung anzustoßen.

Das BIS ist bis dato ein weitgehend buchendes System, das nur rudimentäre Auskunftsfunktionen besitzt. Die Verwendung offener Standard ermöglicht jedoch die Entwicklung von Satellitensystemen, die aus den originären Daten des BIS verdichtete Daten ableiten können. Ein erster Prototyp einer solchen Entwicklung wurde 2008 in Betrieb genommen und wird seitdem erfolgreich für die Erstellung von Bachelor-Abschlüssen genutzt. Ein weiteres System, das primär Auskünfte für Prüfungsämter geben können soll, befindet sich derzeit an der Fakultät für Wirtschaftswissenschaften der Universität Bielefeld in der Entwicklung.

Die gewählte Architektur auf Basis eines Application Servers ermöglicht die Verteilung des Systems auf mehrere Server-Instanzen, um die besonders zu Semesterbeginn und in den Prüfungsphasen auftretenden Lastspitzen abfangen zu können. Außerdem erlaubt die Umsetzung das Einspielen von Updates oder den Einbau neuer Komponenten, ohne dass das gesamte System heruntergefahren werden muss.

6.1.2 Einflussfaktoren

Die Implementierung des BIS erfolgt zum Großteil in der Programmiersprache Java. Java hat sich innerhalb der letzten Dekade an die Spitze der beliebtesten Sprachen gestellt, unterstützt durch die aktive Entwicklergemeinde und die zahlreichen Projekte, die als Open Source verfügbar sind. Hier ist besonders die Apache Foundation als Schirmherr zu nennen, die mittlerweile mehr als 100 Top-Level-Projekte betreut, darunter u. a. die für Java wichtigen Anwendungen *Ant*, *Commons*, *Maven* und *log4j* [10f]. Das System ist als Client-Server-Anwendung ausgelegt: Auf der Server-Seite befindet sich die Datenbank und die Anwendungslogik des Systems, auf der Client-Seite ist nur ein Webbrowser erforderlich. Für die Entwicklung der Anwendungslogik wurde eine komponentenorientierte Vorgehensweise auf Basis von *Apache Tomcat* gewählt, einem Open Source Application Server, der die Java Servlet- und Java-Server Pages-Technologien ([Mor07], [Chu09]) implementiert. Zusammen mit *Struts*, einem Framework für die Entwicklung Java-basierter Webanwendungen ([10g]), wurde so eine REST-Architektur umgesetzt. Als Datenbank wird die vom Hochschulrechenzentrum betriebene Oracle 11-Installation verwendet, jedoch ist das System prinzipiell mit beliebigen DBMS einsatzfähig.

Das BIS wird mit Hilfe der *Eclipse Workbench* entwickelt, einer integrierten Entwicklungsumgebung, die sich besonders durch ihre Flexibilität und Erweiterbarkeit auszeichnet [10b]. Für die Implementierung werden verschiedene Java-Bibliotheken genutzt, die bis auf zwei Pakete alle frei verfügbar sind. Die nicht frei erhältlichen Pakete sind P4DML, ein PDF-Konverter, und ein TIM-Adapter. TIM (Tivoli Identity Manager) ist eine vom Hochschulrechenzentrum der Universität Bielefeld verwendete Authentifizierungslösung. Damit kommt bei der Entwicklung durchgehend Open Source zum Einsatz, wodurch die Kosten auf der Werkzeugseite minimal sind.

Folgende relevante Einflussfaktoren können der Beschreibung des Systems entnommen wer-

den:

- Die Verwendung der externen Bibliotheken verursacht Kopplungen. Das Ausmaß und die Verwendungsstellen sind nicht bekannt.
- Der komponentenorientierte Ansatz lässt vermuten, dass die verschiedenen Teilprojekte des BIS separat entwickelt werden. Das einzige Strukturelement der Programmiersprache auf dieser Abstraktionsebene ist das Paket. Daher ist zu vermuten, dass anders als bei einem monolithischen Projekt mit einem zentralen Paket auf der obersten Projektebene für jedes Teilprojekt ein separates Paket vorgesehen ist.
- Durch die Verwendung des Struts-Frameworks besteht das BIS nicht vollständig aus Java-Quellcode. Die Präsentationsschicht verwendet HTML mit eingebetteten Java-Statements oder JSP-Tags. Eine Analyse des Quellcodes wird damit beschränkt auf die Kernbausteine (Request- und Response-Handler sowie die Anwendungslogik).

Für die Analyse wurden vom BIS-Team die Pakete der Systembasis zur Verfügung gestellt. Der Größe der Systems beläuft sich auf ca. 14,1 MB, verteilt auf 162 Verzeichnisse mit insgesamt 1863 Dateien (Java-Quellcode und HTML-Dokumentation der einzelnen Pakete).

6.2 Auswahl der Analysewerkzeuge

Für die Ermittlung der elementaren Kenngrößen kommen die Eclipse-Plugins Metrics und JDepend zum Einsatz sowie Understand for Java. Das Analysewerkzeug iPlasma wird für die Erzeugung der Übersichtspyramiden benutzt. Die Darstellung der Systemstruktur basiert auf den vom Metrics-Plugin ermittelten Werten und wird durch eine Eigenentwicklung erzeugt.

6.2.1 iPlasma

iPlasma ([10c]) integriert mehrere Analysewerkzeuge in eine konsistente Umgebung. Das Werkzeug unterstützt dabei alle für eine Untersuchung notwendigen Schritte: Extraktion eines Softwaremodells aus Quellcode bis hin zu aggregierten Bewertungen auf Basis von Metriken. Außerdem ist die Suche nach Code-Duplikaten möglich. Der Großteil der Werkzeuge befasst sich mit der Visualisierung von Strukturen, darunter auch die Darstellung der Class Blueprints nach Lanza ([Lan03]).

Die Entwickler des Systems haben damit mehrere große Projekte untersucht, darunter die Eclipse Workbench sowie den Webbrowser Mozilla Firefox (beide besitzen mehr als 2 Millionen Lines of Code). Der Umfang des BIS sollte daher bei einer Analyse keine Schwierigkeiten bereiten.

6.2.2 Metrics

Metrics ([05b]) ist als Plugin für die Eclipse Workbench das ideale Werkzeug, um eine entwicklungsbegleitende Qualitätssicherung in dieser Umgebung umsetzen. Im Gegensatz zu iPlasma analysiert Metrics jedoch nicht den Java-Quellcode, sondern instrumentiert diesen, so dass

erst nach einem Build Kenngrößen ermittelt werden können. Da durch diesen Ansatz die tatsächlichen Strukturen des Quellcodes erfasst werden können, so wie sie der Java-Compiler extrahiert, dürften die Ergebnisse sehr präzise sein. Die Notwendigkeit eines Compile-Laufs erfordert allerdings nicht nur die Bereitstellung sämtlicher externer Bibliotheken, sondern für die zwei nicht zur Verfügung stehenden Pakete auch Anpassungen am BIS-Quellcode. Um den Quellcode trotz der fehlenden Pakete kompilieren zu können, wurden die betroffenen Bereiche daher entweder auskommentiert oder durch die Ermittlung nicht beeinflussende Elemente ersetzt. Erforderlich war dies bei nur drei Dateien.

Zum Umfang der von Metrics ermittelten Kenngrößen gehören die klassischen Volumenmaße, die Maße für objektorientierte Systeme von Chidamber und Kemerer ([CK94]) sowie das MOOD-Set ([HCN98]) und die Kopplungsmaße von Martin ([Mar02]). Neben der Darstellung der Werte in der Eclipse Workbench besteht für die Weiterverarbeitung auch die Möglichkeit eines XML-Exports. Über die Leistungsfähigkeit beim Umgang mit großen Systemen sind keine Angaben zu finden, ebenso wenig Vergleiche mit anderen Werkzeugen.

6.2.3 JDepend

Das Ziel von JDepend ([Cla06]) ist die Ermittlung der Entwurfsqualität von Paketen mit Hilfe von Metriken, wie sie Martin beschrieben hat ([Mar02]). Dazu gehören afferente und efferente Kopplung, Abstraktheit, Instabilität und die Detektion von Abhängigkeitszyklen. Das Werkzeug ist sowohl als Plugin als auch in Form einer Standalone-Lösung (auf Konsolenebene oder mit GUI) verfügbar.

JDepend macht bei der Analyse einige vereinfachende Annahmen: Java-Interfaces werden wie abstrakte Klassen behandelt. Dadurch wird die Berechnung der Abstraktheit beeinflusst. Die Berechnung der beiden Kopplungsformen unterscheiden sich von der ursprünglichen Definition nach Martin, da nicht direkt die Klassenabhängigkeiten ermittelt, sondern die Import-Beziehungen der Pakete untereinander erfasst werden. Geht man davon aus, dass nur Pakete importiert werden, deren Klassen tatsächlich auch Verwendung finden, sollten die Werte allerdings korrekt sein. Wie das Metrics-Plugin bietet auch JDepend die Möglichkeit, Messwerte in XML-Format zu exportieren.

Auch zu diesem Werkzeug gibt es keine Referenzangaben oder Hinweise für den Umgang mit großen Projekten. Die Ergebnisse sowohl von Metrics als auch von JDepend sollten daher auf Übereinstimmung geprüft werden. Alternativ könnte man einen Abgleich mit einem weiteren, allerdings kommerziellen Werkzeug wie Understand for Java vornehmen.

6.2.4 Understand for Java

Die von SciTools entwickelte Analyseplattform *Understand* ist für eine ganze Reihe von Programmiersprachen verfügbar, u. a. auch für C/C++ und Java (s. [05c]). Die C/C++-Variante wurde schon erfolgreich bei der Analyse eines großen kommerziellen Modellierungswerkzeugs eingesetzt, das von einem der weltgrößten Unternehmen im Fassaden- und Profilbau entwickelt wird (s. [Teß04]). Die Leistungsfähigkeit beim Umgang mit großen Systemen steht daher außer Frage. Von allen beschriebenen Werkzeugen bietet Understand for Java zudem die umfassendste Menge elementarer Maße. Ermittelte Werte können in einem XML-Format und auch

in Form einer CSV-Datei exportiert werden.

Aufgrund ihrer Integration in die Eclipse Workbench wird zunächst den beiden Plugins Metrics und JDepend der Vorzug bei der Analyse gewährt.

6.2.5 Messaufgaben

Bei Systemen von der Größenordnung des BIS ist zunächst ein Überblick sinnvoll, bevor eine Detailanalyse erfolgt. Dazu werden folgende Messaufgaben bearbeitet: Lines of Code, Anzahl Methoden, Anzahl Klassen und die Anzahl von Paketen, ermittelt für das gesamte System. Informationen über die Nutzung von Vererbung können ebenfalls hilfreich sein, weshalb auch die Vererbungstiefe und -breite ermittelt wird.

Die Detailanalyse betrachtet ausgesuchte abgeleitete Merkmale des Qualitätsmodells, die sich im Wesentlichen mit der Organisation des Quellcodes auf Paketebene befassen. Dazu gehören die Morphologie des Systems sowie die Qualitätsmaße von Martin, die auf die Kopplung von Paketen abzielen. Zu ermitteln ist also:

- Anzahl Klassen pro Paket
- Anzahl Pakete pro Paket
- Schachtelung von Klassen und Paketen
- Anzahl verschiedener Klassen und Schnittstellen aus Fremdpaketen, sowohl in der Parameterliste als auch bei der Definition von Klassen (afferente Kopplung)

Die Werte für den Überblick werden mit Hilfe von iPlasma ermittelt und an Stelle einer Tabelle in einer alternativen Darstellungsform abgebildet. Für die Ermittlung der Detailanalyse kommen zunächst Metrics und JDepend zum Einsatz und die Kenngrößen werden anschließend mit den Werten von Understand for Java abgeglichen.

6.3 Ermittlung der Kenngrößen

6.3.1 Rahmenbedingungen der Analyse

Die Analyse eines Projekts dieser Größenordnung erfordert auch eine für heutige Verhältnisse gut ausgestattete Maschine. Die Instrumentierung des Quellcodes durch das Metrics-Plugin setzt einen kompletten Build des Projekts voraus, dem dann die eigentliche Extraktion der Kenngrößen folgt. Beides ist sehr speicherintensiv, so dass hier eine minimale Ausstattung von mindestens 2 GB RAM anzuraten ist. Die Analyse der Pakethierarchie auf Basis eines XML-Exports der ermittelten Kenngrößen benötigt weitere 2,6 GB RAM und ist sehr rechenlastig. Soll eine die Entwicklung begleitende Qualitätssicherung durch den Entwickler stattfinden, ist daher ein System mit Mehrkernprozessor und mindestens 4 GB RAM bereitzustellen.

6.3.2 Probleme bei der Ermittlung

Die Ermittlung der elementaren Kenngrößen ist für keines der aufgeführten Werkzeuge ein Problem. Auffällig ist aber die Diskrepanz, die sich bspw. bei der Anzahl der Klassendefinitionen zeigt. Aufgrund der Fehlens von diesbezüglichen Definitionen auch in der Dokumentation des kommerziellen Werkzeugs Understand for Java ist nicht zweifelsfrei klar, wie es dazu kommt. iPlasma zählt offenbar Schnittstellen und abstrakte Klassen nicht zu den Klassendefinitionen, Understand for Java hingegen schon. Metrics stellt Schnittstellen und Klassen gleich. Für eine präzise Auswertung muss zunächst für das dabei verwendete Werkzeug sicher gestellt werden, auf welche Weise die einzelnen Kenngrößen zustande kommen. Bis dahin sollten die Messwerte insgesamt mit Vorsicht verwendet werden.

6.3.3 Ein erster Systemüberblick

Im Vergleich zu einer tabellarischen Darstellung bietet eine sog. *Übersichtspyramide* nach Lanza neben den absoluten Werten auch eine kompakte Zuordnung verschiedener Mittelwerte [LM06, S. 24ff]. Abbildung 6.1 zeigt die Übersichtspyramide für das BIS. Zu sehen sind drei farblich markierte Bereiche:

- gelb: Dieser Bereich der Pyramide beschreibt die Größe und Komplexität (im Sinne von McCabe) des Systems. Die Zahlen in der rechten Spalte sind die absoluten Zahlen zur zyklomatischen Komplexität (CYCLO), Lines of Code (LOC), Anzahl definierter Methoden in einer Klasse (NOM) sowie die Anzahl Klassen (NOC) und Pakete (NOP).
  Die Werte auf der linken „Treppe“ sind die Durchschnittsangaben wie die Anzahl Methoden pro Klasse. Sie berechnen sich immer durch Division der nächst unteren Zahl durch die nächst höhere. Die durchschnittliche zyklomatische Komplexität von 0,16 ergibt sich also aus $CYCLO = \frac{CYCLO}{LOC}$.

- hellgrün: Die Spitze der Pyramide zeigt die Mittelwerte zweier vererbungsrelevanter Kenngrößen. NDD (Numbers of Direct Descendants) beschreibt die durchschnittliche Anzahl direkter Kindklassen, HIT (Hierarchy of Inheritance) die durchschnittliche Tiefe der Vererbungsbäume.

- hellblau: Der dritte Bereich auf der rechten Seite zeigt zwei Kopplungswerte: FOUT (Fan-Out) steht dabei für die Anzahl Klassenreferenzen und CALL für die Anzahl disjunkter Methodenaufrufe im gesamten System.

Für eine Interpretation der Größen- und Komplexitätsmittelwerte der Übersichtspyramide können prinzipiell die Grenzwerte des Qualitätsmodells angelegt werden. Damit wären alle Werte kleiner neun i. a. unauffällig. iPlasma nutzt geringfügig andere Grenzwerte, wie die Farbgebung erkennen lässt. Rot steht dabei für eine Überschreitung des Grenzwertes, grün und blau für akzeptable Werte.

Im folgenden Abschnitt wird als ein Beispiel für die Anwendung des Qualitätsmodells die Morphologie des BIS untersucht. Beantwortet werden soll die Frage, ob sich die schlanke und feingliedrige Struktur, die sich aus den Werten der Übersichtspyramide ableiten lässt, durch eine Detailanalyse bestätigt werden kann.

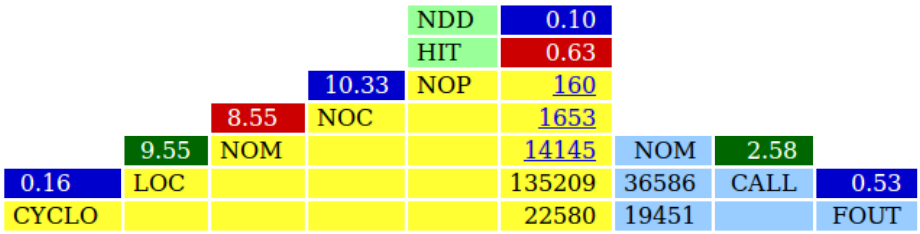

Abbildung 6.1: Übersichtspyramide zum BIS.

6.3.4 Untersuchung der Morphologie des BIS

Zunächst werden die Pakete anhand ihres Volumens, gemessen in Lines of Code, gegenüber gestellt, um einen Eindruck von der allgemeinen Verteilung der Code-Menge zu erhalten. Dann folgt eine Detailuntersuchung der Schachtelung eines aufgrund seiner Größe auffälligen Pakets.

Abbildung 6.2 zeigt ein Balkendiagramm, das die Pakete der ersten Ebene der Strukturierungshierarchie in TLOC (Total Lines of Code) aufführt. Auffällig ist die Breite der ersten Ebene: Hier sind 18 Pakete zu finden. Damit wird der Grenzwert dieses abgeleiteten Merkmals, den das Qualitätsmodell zugrunde legt, weit überschritten. Vergegenwärtigt man sich allerdings die Entwicklung des BIS und den doch sehr heterogenen Funktionsumfang, relativiert sich die Bewertung dieser Kenngröße: Die Namen der Pakete deuten auf eine funktionsorientierte Organisation hin, sie implementieren also jeweils eine eigene Teilanwendung im Kontext des BIS und können demnach auch separat bewertet werden.

Der Name und der Umfang des ersten Pakets lässt vermuten, dass es sich hierbei um ein Paket zur Datenabstraktion handelt. Damit deutet sich eine Trennung der Anwendung in eine Anwendungs- und Datenhaltungsschicht an. Eine Analyse der Kopplung mit anderen Paketen der ersten Ebene und deren Bestandteilen könnte diesen Verdacht bestätigten, wird hier aber nicht weiter verfolgt.

Lässt man das Paket `org.unibi.kvv` (Implementierung des kommentierten Vorlesungsverzeichnisses) als einen der ältesten und vom Funktionsumfang her größten Baustein außer Acht, so zeigt sich, dass die Pakete insgesamt einen sehr moderaten Umfang besitzen (s. Tabelle 6.1 auf der nächsten Seite). Nur `org.unibi.common` fällt aus dem Rahmen, wobei der Name vermuten lässt, dass es sich hierbei um ein ähnlich fundamentales Paket handelt wie bei `org.uni.data`. Eine Analyse der afferenten Kopplung mit JDepend bestärkt diesen Verdacht: Allein das Teilpaket `org.unibi.common.dao` besitzt einen Kopplungswert von 95, der nur noch übertroffen wird vom Wert des Pakets `org.unibi.common.tools`: 108. Die auf Klassen dieses Pakets zugreifenden Klassen und Methoden sind über das gesamte System verteilt, d. h. der Impact einer Änderung an diesem Paket hat unvorhersehbare Auswirkungen. Abbildung 6.3 (S. 121) zeigt eine Darstellung dieser von `org.unibi.common` abhängigen Pakete des BIS inklusive der Verbindungsstärke, gemessen anhand der Anzahl der Verwendungskontexte von Klassen aus `org.unibi.common`. Gut zu erkennen ist die enge Kopplung zur vermuteten Datenabstraktionsschicht mit dem Wert 6770. Betrachtet man nun die Abhängigkeiten `org.unibi.common` selbst (Abbildung 6.4, S. 122),

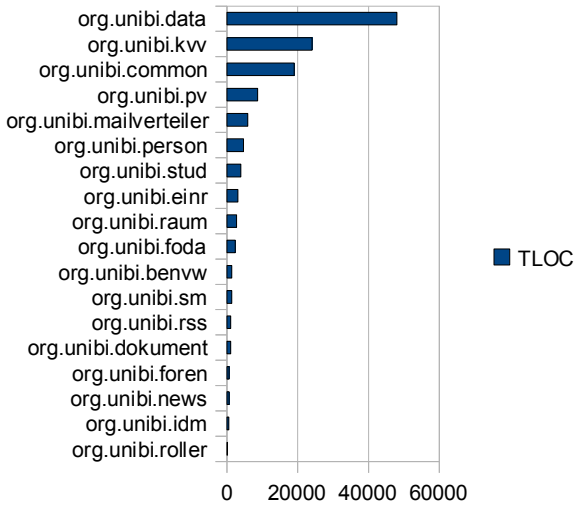

Abbildung 6.2: Größe der Pakete der ersten Hierarchieebene in LOC.

so zeigt sich, dass `org.unibi.common` von `org.unibi.data` abhängt. Zwischen den beiden größten Paketen des BIS besteht also einen Abhängigkeitszyklus!

Name	TLOC
org.unibi.benvw	1405
org.unibi.common	19008
org.unibi.dokument	1078
org.unibi.einr	3130
org.unibi.foda	2310
org.unibi.foren	655
org.unibi.idm	557
org.unibi.kvv	24066
org.unibi.mailverteiler	5883
org.unibi.news	653
org.unibi.person	4665
org.unibi.pv	8651
org.unibi.raum	2740
org.unibi.roller	191
org.unibi.rss	1084
org.unibi.sm	1401
org.unibi.stud	3939

Tabelle 6.1: Größe der Pakete der ersten Hierarchieebene.

Bei der Bestimmung der Morphologie eines Systems wird die Schachtelung der Elemente betrachtet, wie also Klassen in Paketen zusammengefasst werden und Pakete mit anderen Pa-

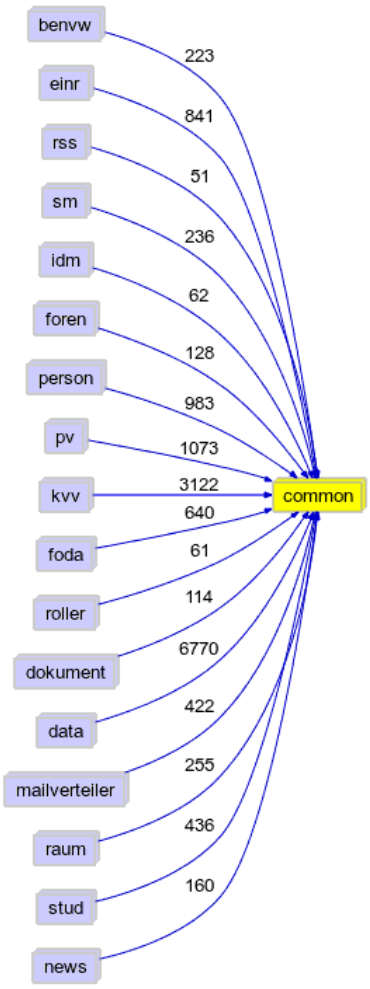

Abbildung 6.3: Von `org.unibi.common` abhängige Pakete inkl. Anzahl der Verwendungen.

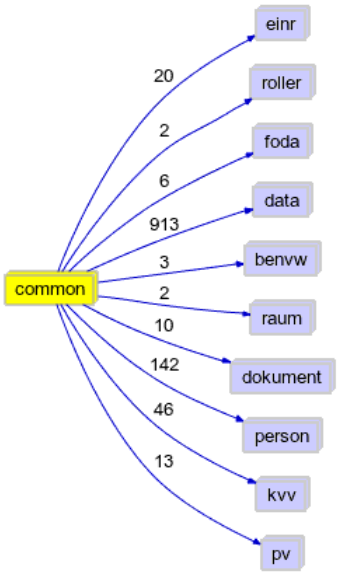

Abbildung 6.4: Abhängigkeiten des Pakets `org.unibi.common` inkl. Anzahl der Verwendungen.

keten und Klassen zu größeren Paketen. Solche Teil-von-Beziehungen können mit Hilfe eines geschachtelte Baum dargestellt werden.

Abbildung 6.5 auf der nächsten Seite zeigt eine solche Darstellung der Morphologie des BIS. Pakete werden als Quadrate abgebildet, Klassen als Ovale. Die Größe der Quadrate bzw. Ovale steht hierbei nicht in Bezug zur Elementgröße. Befindet sich ein Quadrat innerhalb eines größeren Quadrats, so handelt es sich dabei um ein Unterpaket desselben. In der Abbildung stellt das große Quadrat links der Bildmitte das Paket `org.unibi.data` dar. In der rechten oberen Ecke ist das Paket `org.unibi.common` zu sehen. Zu erkennen ist, dass die Hierarchisierung des Systems insgesamt sehr heterogen ist: Die Schachtelungstiefen reichen von nur einer bis hin zu sieben Paketebenen. Außerdem werden teilweise extrem viele Klassen in einem Paket zusammengefasst.

Die Hierarchisierung des Pakets `org.unibi.common`, als Ausschnitt zu sehen in Abbildung 6.6 (S. 124), soll nun näher untersucht werden. Zunächst ist festzustellen, dass das Paket schon auf oberster Ebene 17 Unterpakete aufweist; einzelne Klassen sind auf dieser Ebene nicht vorhanden. Weiterhin kann man gut erkennen, dass einzelne Hierarchieebenen deutlich zu breit angelegt sind: Es werden sehr viele Elemente zu einer Ebene zusammengefasst. Dabei sticht besonders das Unterpaket `user` heraus, das aus zwei Paketen und 86 Klassen besteht. Abbildung 6.7 auf Seite 125 zeigt das Paket in Form einer klassischen Baumstruktur, wobei die Blätter jeweils für die Anzahl von Klassen stehen. Bei einem angenommenen Schachtelungsfaktor von 7 beträgt die mittlere Gesamtabweichung der 33 Teilbäume 1,23. Dieser Wert deutet auf eine große Abweichung von einer idealen Strukturierung hin.

An diesem Beispiel wird deutlich, wie sehr der in der Übersicht 6.1 (S. 119) präsentierte

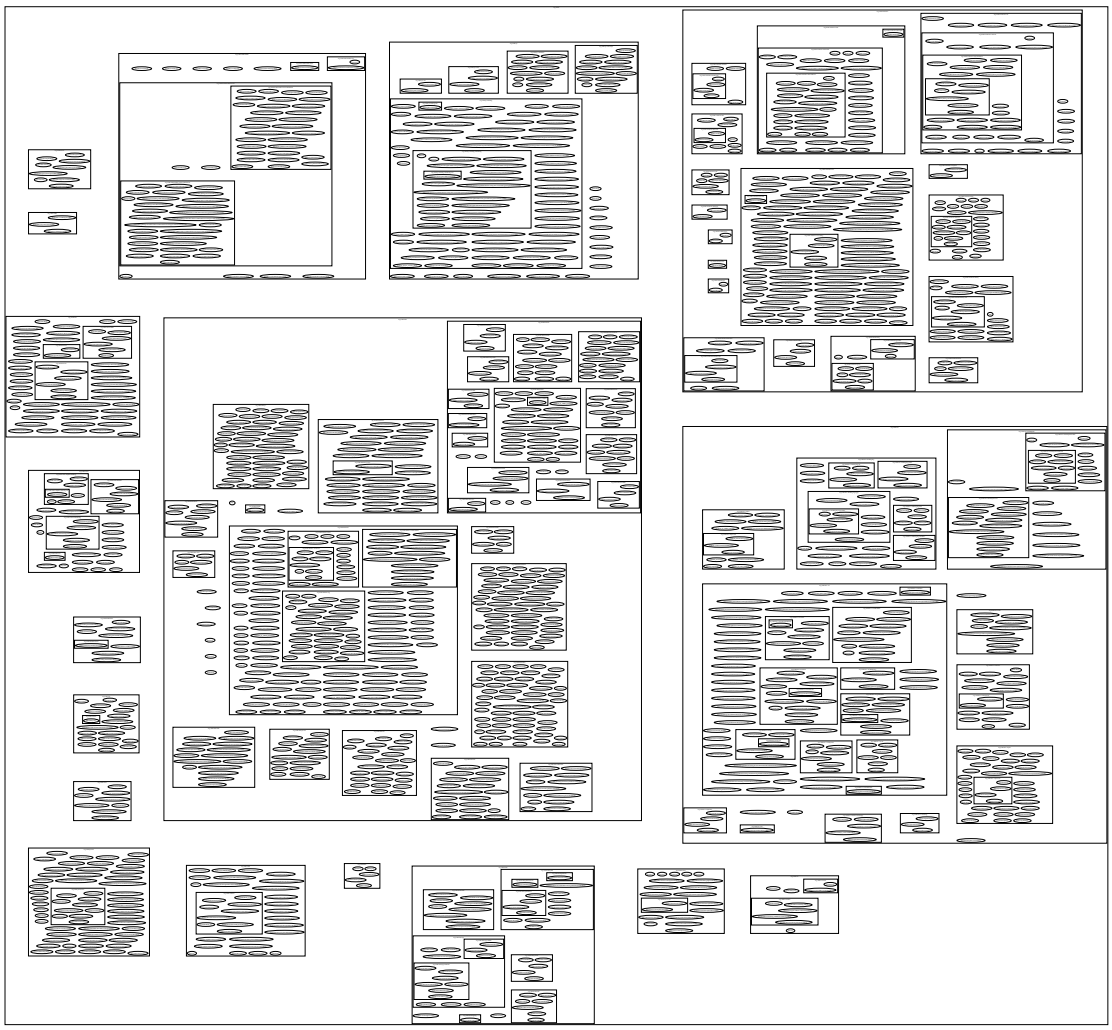

Abbildung 6.5: Darstellung der Paketstruktur des BIS als geschachtelter Baum.

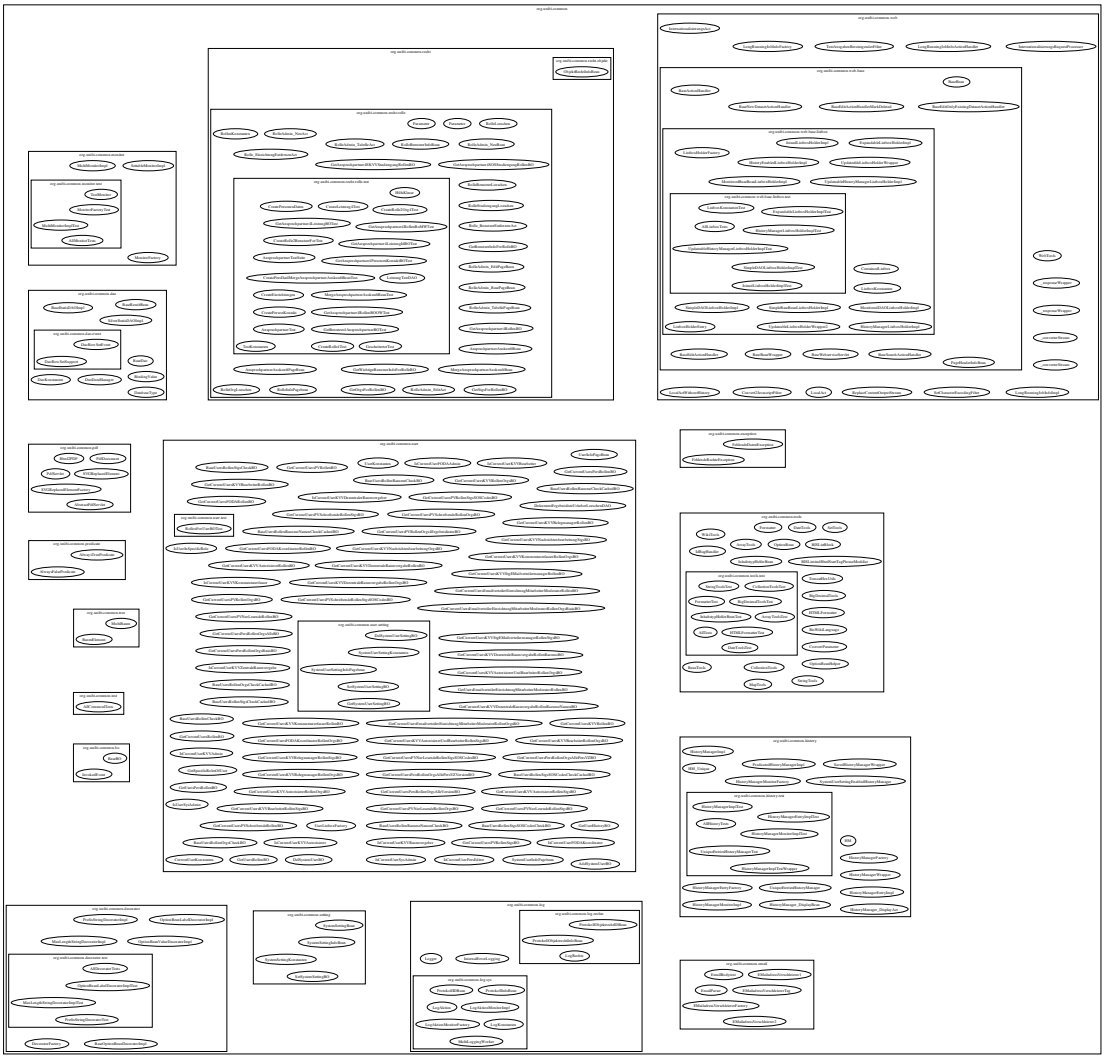

Abbildung 6.6: Paket `org.unibi.common`.

Abbildung 6.7: Strukturbaum des Pakets `org.unibi.common`.

Durchschnitt von 10 Klassen pro Paket die tatsächlich vorliegende Hierarchisierung verschleiern kann.

6.4 Anwendung des Qualitätsmodells

Aus Sicht des entwickelten Qualitätsmodells betreffen die hier für `org.unibi.common` ermittelten Kenngrößen die abgeleiteten Merkmale *Kopplung*, *Breite* und *Tiefe* der Hierarchisierung sowie die *Größe eines Elements*. Damit lassen sich die Merkmale *Modularität* und *Hierarchisierung* teilweise bestimmen. Einige abgeleiteten Merkmale sind in Klammern eingefasst, da keine zugehörigen Kenngrößen ermittelt wurden. Aufgrund der logischen Und-Verknüpfung ergibt sich daraus aber kein Problem für die Anwendung, solange nur ein abgeleitetes Merkmal als „nicht in Ordnung“ bewertet wird:

- Modularität = (Vererbung) $\wedge$ (Breite der Schnittstelle) $\wedge$ Kopplung
- Hierarchisierung = Breite $\wedge$ Tiefe

Die Kopplung ist aufgrund der zyklischen Abhängigkeit als gravierend fehlerhaft zu werten. Die enge Kopplung von `org.unibi.common` und `org.unibi.data` mit 6770 Verbindung bzw. 913 in der Gegenrichtung liegen aus Sicht des Qualitätsmodells weit außerhalb aller empfohlenen Grenzwerte. Mit einer Tiefe von maximal 7 Ebenen, jedoch einer Breite von 17 und mehr Elementen ist die Hierarchisierung nicht gleichmäßig. Die berechnete mittlere Gesamtabweichung beträgt 1,23 bei einem angenommenen Schachtelungsfaktor von 7 und ist damit ebenfalls zu groß. Sowohl Modularität als auch Hierarchisierung sind aufgrund dieser Ergebnisse als „nicht in Ordnung“ zu betrachten.

Die auf diesen Merkmalen aufsetzenden Untereigenschaften *Analysierbarkeit*, *Änderbarkeit* und *Testbarkeit* müssen auch ohne Untersuchung auf mögliche Redundanzen als „nicht in Ordnung“ gewertet werden:

- Analysierbarkeit = Hierarchisierung $\wedge$ (Redundanz) $\wedge$ Umfang $\wedge$ Modularität
- Änderbarkeit = (Redundanz) $\wedge$ Umfang $\wedge$ Modularität
- Testbarkeit = Modularität

Die durchgeführte exemplarische Untersuchung zeigt, dass das BIS aus Sicht des entwickelten Qualitätsmodells z. T. erschreckende Mängel aufweist und die Wartbarkeit des Systems als gefährdet eingestuft werden muss. Die zyklischen Abhängigkeiten von Paketen wiegen dabei besonders schwer.

Vergleicht man dieses Ergebnis mit den auf Seite 119 aufgeführten Durchschnittswerten der Übersichtspyramide, welche das BIS insgesamt unauffällig erscheinen lassen, wird deutlich, in welchem Umfang solche aggregierten Werte den tatsächlichen Systemzustand verschleiern und die Entwickler täuschen können.

6.5 Folgerungen

Bevor weitere Entwicklungen in Angriff genommen werden, sollte zunächst ein Refactoring der Basis-Pakete `org.unibi.data` und `org.unibi.common` durchgeführt werden mit dem Ziel, den Abhängigkeitszyklus zu durchbrechen und die Abhängigkeit untereinander auf wenige, klar definierte Bereiche zu reduzieren.

Die streckenweise sehr großen Anzahlen von Klassen auf den untersten Hierarchieebenen lässt die Frage nach der Kohäsion der zugehörigen Pakete aufkommen und danach, welcher Idee die Integration der Klassen zu einem Paket folgt.

Eine mögliche Ursache für die stellenweise große Anzahl an Klassen (und evtl. auch für die Kopplungs- und Kohäsionswerte) kann die Verwendung von Code-Generatoren sein, die seit einiger Zeit bei der BIS-Entwicklung zu Einsatz kommen. Die Entwickler arbeiten damit auf einer anderen Abstraktionsebene und nutzen domänenspezifische Sprachen (DSL), die vom Generator in Java-Code übersetzt werden. Dieser Umstand wird vom Qualitätsmodell nicht erfasst. Da die Entwickler den generierten Code jedoch nicht mehr direkt modifizieren müssen, entsteht daraus für das System aus Sicht der Wartbarkeit kein Problem, auch wenn die ermittelten Kenngrößen über den Grenzwerten liegen.

Es bleibt fraglich, wie die weitere Entwicklung in Zukunft gestaltet werden soll. Geht man bspw. von einem 5-Jahres-Zyklus aus, d. h. alle fünf Jahre wird eine Komponente neu entwickelt oder umfassend angepasst, kann die enge Kopplung ein Heraustrennen eben dieser Komponente erschweren oder unmöglich machen. Ob sich allerdings ein umfassendes Refactoring-Projekt angesichts eines angekündigten kommerziellen Ersatzsystems lohnt, ist noch unklar.

7 Fazit

> The dominant view is that software developers, to deserve the title of software engineers, need to do what other engineers are supposed to: produce a kilogram of paper for every gram of actual deliverable.
>
> *(Meyer, „Object-Oriented Software Construction“)*

7.1 Zusammenfassung der Forschungsergebnisse

Ziel dieser Arbeit war die Entwicklung eines Quellcode basierten Qualitätsmodells für die Beurteilung der Wartbarkeit von Softwaresystemen. Die Entwicklung wurde von den folgenden drei Forschungsfragen geleitet:

1. Welche architekturbezogenen Qualitätsmerkmale lassen sich aus Quellcode ermitteln?
2. Wie relevant sind die Merkmale für die Wartbarkeit und Weiterentwicklung eines Systems?
3. Wie lässt sich aus den gewonnenen Informationen ein Qualitätsmodell entwickeln?

Zur Beantwortung dieser Fragen wurden in Kapitel 2 zunächst etablierte Qualitätsstandards untersucht. Im Fokus stand dabei die Zerlegung des Qualitätsbegriffs und die Überleitung zu Merkmalen, die sich automatisiert aus Quellcode ermitteln lassen. Die Ermittlung dieser Merkmale war Thema des dritten Kapitels, welches zu dem Ergebnis kommt, dass die vorgestellten Metriken allein nicht für eine Qualitätsbewertung der Architektur geeignet sind. Die sich daraus ergebende Notwendigkeit, die Architektur einer Bewertung zugänglich zu machen, führte in Kapitel 4 zu einer detaillierten Untersuchung der Entwicklung des Architekturbegriffs im Kontext der Softwareentwicklung. Das Systemverstehen durch den Entwickler — beeinflusst von der Modularität der Architektur, ihrer internen Kopplung und Kohäsion sowie der zugrunde liegenden zentralen Strukturierungsidee — konnte dabei als die wichtigste Qualitätseigenschaft identifiziert werden.

Ausgehend von den formulierten Anforderungen an die Struktur von Quellcode wurde in Kapitel 5 ein Qualitätsmodell entworfen, das eine Zerlegung des Qualitätsbegriffs bis zur Ebene des Quellcodes vornimmt und in Kombination mit einem speziell an die Programmiersprache Java ausgerichteten Softwaremodells eine einfache Bewertung von Softwaresystemen vorschlägt. Durch die Art der Ableitung höherwertiger Qualitätsaussagen aus den Merkmalen

wurde darüber hinaus eine Integration mit den bestehenden etablierten Softwaremaßen möglich. In Kapitel 6 wurde das BIS exemplarisch unter Anwendung des Qualitätsmodells bewertet, wobei das Hauptaugenmerk auf der Untersuchung der Morphologie lag.

Das Qualitätsmodell als Ergebnis der Untersuchungen leistet im Detail folgenden Forschungsbeitrag:

- Es bietet eine Auswahl von Qualitätseigenschaften und -merkmalen auf Basis bestehender Standards und den Anforderungen der Wartbarkeit von Architektur aus Sicht des Entwicklers.
- Diese Merkmale werden in ein Qualitätsmodell eingebettet und zueinander in Beziehung gesetzt. Dadurch wird der Qualitätsbegriff kommunizierbar.
- Für die Ermittlung stellt das Modell eine Auswahl von Grenzwerten für Kenngrößen auf Basis theoretischer Erkenntnisse bereit.
- Durch die Berücksichtigung der spezifischen Eigenschaften einer Programmiersprache verfeinert das Modell die Menge der Messaufgaben und damit die Genauigkeit der Bewertung.
- Die Beschränkung auf elementare Maße macht die Auswirkungen von Änderungen am Quellcode auf die Bewertung direkt nachvollziehbar. Dabei bleibt es aber flexibel und kann sowohl an die Fähigkeiten der Entwickler als auch an projektspezifische Vorgaben angepasst werden.

Im Gegensatz zu den etablierten Qualitätsmodellen liefert das in dieser Arbeit entworfene Modell keine höherwertigen quantitativen Aussagen, sondern funktioniert als reiner Indikator. Aus diesem Grund können keine quantitativen Abschätzungen zu den einzelnen Qualitätseigenschaften, geschweige denn zum potentiellen Aufwand von Wartungsarbeiten erbracht werden. Derartige Angaben haben jedoch immer nur im Kontext der ganz spezifischen Messsituation eine Bedeutung und sind zu einem guten Teil abhängig von der subjektiven Einschätzung des Messenden.

Die Anwendung des Qualitätsmodells wird derzeit noch von der fehlenden Werkzeugunterstützung behindert. Zwar können viele der elementaren Kenngrößen durch die gängigen Werkzeuge zur Metrikanalyse ermittelt werden, aber die Interpretation dieser Werte und die Kombination der Ergebnisse der Indikatorfunktion fehlt. Für die Untersuchungen im Rahmen dieser Arbeit wurden die XML-Exporte des Metrics-Plugins mit Hilfe von selbst entwickelten Python-Skripten ausgewertet. Zweckmäßig wäre eine Erweiterung des Plugins um diese Auswertungsfunktionalität.

7.2 Ausblick

Die Bearbeitung der Forschungfragen führte neben Erkenntnissen auch zu neuen Fragen und Aufgaben, die zum Thema weiterer Forschungarbeiten werden sollten:

- Sowohl die Grenzwerte des Qualitätsmodells als auch die der etablierten Softwaremaße berücksichtigen nicht die verschiedenen Arten von Software. Eine Ermittlung Software spezifischer Werte sollte daher in Betracht gezogen werden. Zur Typisierung könnte man zunächst eine Unterteilung in Betriebssysteme und Anwendungssoftware vornehmen. Letztere wiederum ließe sich weiter unterteilen in Anwendungen mit klassischer Architektur nach QUASAR, moderne Webanwendungen, die oft mit Hilfe von Frameworks umgesetzt werden, und schließlich Systeme, die dem SOA-Ansatz folgen. Es steht zu vermuten, dass für jede Softwareart ein eigener Satz an Grenzwerten angepasst werden muss.
- Die Bewertung von Quellcode aus mehreren Programmiersprachen ist bisher kaum ein Thema. Ein möglicher Grund könnten die Schwierigkeiten sein, ein konsistentes Softwaremodell zu erstellen. Das gilt besonders für die Kombination imperativer Sprachen mit ihren deklarativen Verwandten wie CSS oder XSL.
- Die Stabilität wird in der Literatur zur Qualitätsbewertung nicht als Qualitätsmerkmal geführt. Hier fehlt es nicht nur an Richtlinien für die Verwendung von Strukturen im Quellcode, sondern auch an empirischen Untersuchungen zu Grenzwerten und dem Aufbau interner Firewalls. Besonders sicherheitskritische Software würde davon profitieren.
- Mit der zunehmenden Verbreitung domänenspezifischer Sprachen (DSL) und den damit verbundenen Code-Generatoren verändert sich die Situation für die Entwickler: Sie müssen sich mehr mit der Modellierung von Software befassen und weniger mit Quellcode. Dabei stellt sich aber die Frage, wie eine Qualitätsbewertung bei derart entwickelten Systemen realisiert werden kann.

Die Dokumentation als Qualitätsmerkmal wird von den meisten Qualitätsmodellen nicht berücksichtigt. Sie ist allerdings eines der wichtigsten Qualitätsmerkmale, die ein Softwaresystem ausmachen. Daher sollte ihr auch ein entsprechender Stellenwert in den Modellen zugebilligt und Techniken entwickelt werden, die eine Bewertung ermöglichen. Einen Versuch in diese Richtung unternahm Knuth mit seinem *Literate Programming*-Ansatz [Knu84]. Er tauschte die Rollen von Dokumentation und Quellcode: Anstatt den Quellcode mit Kommentaren nachträglich zu dokumentieren, wird ein Softwaresystem als Dokument verstanden, dass mit Hilfe sog. *Code Chunks* um konkrete Anweisungen erweitert wird. Knuth lieferte mit seinem TEX-System, das er auf diese Weise entwickelte, auch gleich einen Machbarkeitsnachweis für seine Idee.

Mit der Integration des Quellcodes in die Dokumentation führte Knuth gleichzeitig eine völlig neue Perspektive in die Softwareentwicklung ein: Ausschlaggebend für den Entwickler ist nicht mehr die Struktur des Quellcodes, sondern das Dokument: Die *Kommunizierbarkeit* von Software ist nun das Ziel. Das Programmverstehen wird damit unabhängig von der logischen Dekomposition des Systems in Form von Paketen, Klassen usw. Dieser Wechsel führt aber auch dazu, dass die bisher gültigen Qualitätsmerkmale, die direkt auf den Quellcode aufsetzen, für eine Bewertung nutzlos werden. Relevant ist die Qualität der Beschreibung im Dokument, nicht des Codes. Inwieweit es Bestrebungen hinsichtlich der Qualitätsuntersuchung von literat programmierter Software gibt, ist allerdings nicht bekannt.

Der positive Effekt der Kommunizierbarkeit konnte an einem literat programmierten System der Fakultät für Wirtschaftswissenschaften, Universität Bielefeld, empirisch gezeigt werden.

Das *Fakultätsinformationssystem*, in der Programmiersprache Python und mit dem Literate-Programming-Werkzeug Noweb ([Ram08]) entwickelt, wurde in drei Diplomarbeiten von programmierunerfahrenen Studierenden auf seine Wartbarkeit hin untersucht und positiv bewertet [Car10; Hei10; Huc10].

Die literate Softwareentwicklung ist trotz des Erfolgs von TEX nicht weit verbreitet. Ein Grund dafür könnte die im Vergleich zu modernen Entwicklungsumgebungen rudimentäre Werkzeugunterstützung sein. Wahrscheinlicher aber ist, dass das eigentliche Hindernis die Entwickler selbst sind:

> „Without wanting to be elitist, the thing that will prevent literate programming from becoming a mainstream method is that it requires thought and discipline. The mainstream is established by people who want fast results while using roughly the same methods that everyone else seems to be using, and literate programming is never going to have that kind of appeal. This doesn't take away from its usefulness as an approach.“ Patrick TJ McPhee in [Ram08]

Literatur

[01] *ISO/IEC 9126-1: Software engineering - Product quality. Part 1: Quality Model*. International Organization for Standardization. 2001. URL: http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=22749 (siehe S. 9–11, 14).

[03a] *ISO/IEC 9126-2: Software engineering - Product quality. Part 2: External metrics*. International Organization for Standardization. 2003. URL: http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=22750 (siehe S. 9, 15).

[03b] *ISO/IEC 9126-3: Software engineering - Product quality. Part 3: Internal metrics*. International Organization for Standardization. 2003. URL: http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=22891 (siehe S. 9).

[04a] *ISO/IEC 9126-4: Software engineering - Product quality. Part 4: Quality in use metrics*. International Organization for Standardization. 2004. URL: http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=39752 (siehe S. 9).

[04b] *Javadoc*. Oracle. 2004. URL: http://www.oracle.com/technetwork/java/javase/documentation/index-jsp-135444.html (besucht am 12. 11. 2011) (siehe S. 79).

[05a] *ISO/IEC 25000: Software engineering - Software product Quality Requirements and Evaluation SQuaRE. Guide to SQuaRE*. International Organization for Standardization. 2005. URL: http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=35683 (siehe S. 9).

[05b] *Metrics*. 2005. URL: http://metrics.sourceforge.net/ (besucht am 10. 10. 2011) (siehe S. 115).

[05c] *Understand for Java*. Scientific Toolworks, Inc. Mai 2005. URL: http://www.scitools.com (besucht am 12. 05. 2005) (siehe S. 116).

[06a] *International Standard - ISO/IEC 14764 IEEE Std 14764-2006. Software Engineering — Software Life Cycle Processes — Maintenance*. IEEE. 2006. DOI: 10.1109/IEEESTD.2006.235774 (siehe S. 3).

[06b] *JUnit*. Dez. 2006. URL: http://www.junit.org/index.htm (besucht am 18. 12. 2006) (siehe S. 78).

[09] *Community Software Architecture Definitions*. Software Engineering Institute, Carnegie Mellon University. 2009. URL: http://www.sei.cmu.edu/architecture/start/community.cfm (besucht am 07. 12. 2009) (siehe S. 30).

[10a] *Cruise Control*. 2010. URL: http://cruisecontrol.sourceforge.net/ (besucht am 11. 10. 2010) (siehe S. 78).

[10b] *Eclipse*. Eclipse Foundation. 2010. URL: http://www.eclipse.org/ (besucht am 12. 05. 2010) (siehe S. 114).

[10c] *iPlasma*. Politehnica University from Timisoara, Romania. 2010. URL: http://loose.upt.ro/iplasma/ (besucht am 20. 11. 2010) (siehe S. 115).

[10d] *Metrikbasierte Qualitätsanalyse - Einstieg*. Virtuelles Software Engineering Kompetenzzentrum (ViSEK), Fraunhofer IESE. 2010. URL: http://www.softwarekompetenz.de/ (besucht am 10. 10. 2011) (siehe S. 63, 64).

[10e] *Prädiktor*. 2010. URL: http://de.wikipedia.org/w/index.php?title=Prognose&oldid=80317094 (besucht am 15. 10. 2010) (siehe S. 65).

[10f] *Programming Language Index*. Apache Foundation. 2010. URL: http://projects.apache.org/indexes/language.html#Java (besucht am 20. 11. 2010) (siehe S. 114).

[10g] *Struts*. Apache Foundation. 2010. URL: http://struts.apache.org/ (besucht am 20. 11. 2010) (siehe S. 114).

[98] *IEEE/EIA 12207.0-1996: Software life cycle processes*. IEEE. März 1998. URL: http://standards.ieee.org/findstds/standard/12207.0-1996.html (siehe S. 8, 34).

[99a] *ISO/IEC 14598-1: Software Product Evaluation*. International Organization for Standardization. 1999. URL: http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=24902 (siehe S. 8).

[99b] *Java Code Conventions*. Oracle. 1999. URL: http://www.oracle.com/technetwork/java/codeconvtoc-136057.html (besucht am 12. 11. 2011) (siehe S. 79).

[AC94] Fernando Brito e Abreu und Rogério Carapuça. „Object-Oriented Software Engineering: Measuring and Controlling the Development Process“. In: *Proceedings of the 4th Int. Conf. on Software Quality*. McLean, VA, USA, Okt. 1994 (siehe S. 21).

[Agg+06] K. K. Aggarwal u. a. „Empirical Study of Object-Oriented Metrics“. In: *Journal of Object Technology* 5.8 (Dez. 2006), S. 149–173 (siehe S. 3, 20, 101, 106, 107, 112).

[AL03] M. Alshayeb und Wei Li. „An empirical validation of object-oriented metrics in two different iterative software processes“. In: *IEEE Transactions on Software Engineering* 29.11 (2003), S. 1043–1049 (siehe S. 22).

[AM96] Fernando Brito e. Abreu und Walcelio Melo. „Evaluating the Impact of Object-Oriented Design on Software Quality“. In: *Proceedings of the 3rd International Symposium on Software Metrics: From Measurement to Empirical Results*. IEEE Computer Society, 1996, S. 90–99 (siehe S. 21).

[Amb01] Scott W. Ambler. *The Object Primer: The Application Developer's Guide to Object-orientation and the UML*. 2. Auflage. Cambridge: University Press, Aug. 2001 (siehe S. 71, 86).

[And+04] Christoph Andriessens u. a. *QBench Projektergebnis: Stand der Technik*. 2004. URL: http://www.qbench.de (siehe S. 3).

[And+93] Bruce Anderson u. a. „Software architecture: the next step for object technology (panel)“. In: *SIGPLAN Not.* 28.10 (1993), S. 356–359 (siehe S. 69).

[BBM96] Victor R. Basili, Lionel C. Briand und Walcelio L. Melo. „A Validation of Object-Oriented Design Metrics as Quality Indicators“. In: *IEEE Transactions on Software Engineering* 22.10 (Okt. 1996), S. 751–761 (siehe S. 4, 21).

[BCK98] Len Bass, Paul Clements und Rick Kazman. *Software Architecture in Practice*. Reading et al.: Addison-Wesley, 1998 (siehe S. 2, 4, 7, 8, 28, 34, 47–50, 52, 71).

[BDW99] Lionel C. Briand, John W. Daly und Jürgen K. Wüst. „A Unified Framework for Coupling Measurement in Object-Oriented Systems“. In: *IEEE Transactions on Software Engineering* 25.1 (Jan. 1999), S. 91–121 (siehe S. 4).

[Ber10] Fernando Berzal. „Program Representation“. In: *Encyclopedia of Software Engineering*. Hrsg. von Phillip A. Laplante. Bd. 2. Boca Raton et al.: Auerbach Publications, 2010, S. 761–771. ISBN: 978-1420059779 (siehe S. 3).

[BHK04] Marc Born, Eckhardt Holz und Olaf Kath. *Softwareentwicklung mit UML 2*. München: Addison Wesley, 2004 (siehe S. 31).

[Bin99] Robert Binder. *Testing Object Oriented Systems: Models, Patterns and Tools*. Boston, MA, USA: Addison-Wesley Professional, Nov. 1999 (siehe S. 77–79).

[BKN08] Jörg Becker, Helmut Krcmar und Björn Niehaves. *Wissenschaftstheorie und gestaltungsorientierte Wirtschaftsinformatik*. Arbeitsbericht 120. Universität Münster, 2008 (siehe S. 5).

[BL76] L. A. Belady und M. M. Lehman. „A model of large program development“. In: *IBM Systems Journal* 15.3 (1976), S. 225–252 (siehe S. 1, 2).

[BMB99] Lionel C. Briand, Sandro Morasca und Victor R. Basili. „Defining and Validating Measures for Object-Based High-Level Design“. In: *IEEE Transactions on Software Engineering* 25.5 (Sep. 1999), S. 722–743 (siehe S. 4).

[Boe79] Barry W. Boehm. „Software Engineering: R and D trends and defense needs“. In: *Research Directions in Software Technology*. Hrsg. von Peter Wegener. Cambridge, Massachusetts und London, England: M. I. T. Press, 1979 (siehe S. 1).

[Boe81] Barry W. Boehm. *Software Engineering Economics*. Upper Saddle River, NJ, USA: Prentice Hall, Sep. 1981 (siehe S. 19).

[Boo93] Grady Booch. *Object Oriented Analysis and Design With Applications*. 2. Auflage. Amsterdam: Addison-Wesley Longman, Okt. 1993 (siehe S. 83).

[Bot+04] P. Botella u. a. „ISO/IEC 9126 in practice: what do we need to know?“ In: *Procs. First Software Measurement European Forum (SMEF)*. Rom, Jan. 2004 (siehe S. 16).

[BR04] Marcel Bennicke und Heinrich Rust. *Programmverstehen und statische Analysetechniknen im Kontext des Reverse Engingeering und der Qualitätssicherung*. Projektbericht ViSEK/025/D. Kaiserslautern: Fraunhofer IESE, Feb. 2004 (siehe S. 67, 70).

[Bri+00] Lionel C. Briand u. a. „Exploring the relationship between design measures and software quality in object-oriented systems“. In: *Journal . Syst. Softw.* 51.3 (2000), S. 245–273 (siehe S. 21).

[Bro83] Ruven Brooks. „Towards a theory of the comprehension of computer programs“. In: *International Journal of Man-Machine Studies* 18.6 (Juni 1983), S. 543–554 (siehe S. 68).

[Bru+09] Henning Brune u. a. „Ein Campus-Management-System als evolutionäre Entwicklung - Ein Erfahrungsbericht“. In: *9. Int. Tagung Wirtschaftsinformatik: Business Services: Konzepte, Technologien und Anwendungen*. Hrsg. von H.-R. Hansen, D. Karagiannis und H.-J. Fill. Wien, 2009, S. 483–492 (siehe S. 113).

[BS02] Manfred Broy und Johannes Siedersleben. „Objektorientierte Programmierung und Softwareentwicklung“. In: *Informatik Spektrum* 25 (Feb. 2002), S. 3–11 (siehe S. 20, 29, 39).

[BWL99] Lionel C. Briand, Jürgen Wüst und Hakim Lounis. *Using Coupling Measurement for Impact Analysis in Object-Oriented Systems*. Techn. Ber. IESE Report 010.99. International Software Engineering Research Network, März 1999 (siehe S. 101).

[Car10] Marco Carolla. „Controlling mit einem Campus-Management-System. Konzept und Implementierung zur Beurteilung eines Frameworks“. Diplomarbeit. Bielefeld: Universität Bielefeld, 2010 (siehe S. 132).

[Chu09] Kin-man Chung, Hrsg. *JSR 245: JavaServer(TM) Pages 2.1*. Oracle. Dez. 2009. URL: http://jcp.org/en/jsr/detail?id=245 (besucht am 20. 11. 2010) (siehe S. 114).

[CK91] Shyam R. Chidamber und Chris F. Kemerer. „Towards a Metrics Suite for Object Oriented Design“. In: *Conference proceedings on Object-oriented programming systems, languages, and applications* 06–11 (Okt. 1991), S. 197–211 (siehe S. 3, 20).

[CK94] Shyam R. Chidamber und Chris F. Kemerer. „A Metrics Suite for Object Oriented Design“. In: *IEEE Transactions on Software Engineering* 20.6 (Juni 1994), S. 476–493 (siehe S. 3, 101–103, 116).

[Cla06] Mike Clark. *JDepend*. Jan. 2006. URL: http://clarkware.com/software/JDepend.html (besucht am 12. 01. 2006) (siehe S. 116).

[CM78] Joseph P. Cavano und James A. McCall. „A framework for the measurement of software quality“. In: *Proceedings of the software quality assurance workshop on Functional and performance issues*. ACM, 1978, S. 133–139 (siehe S. 63, 64).

[Con68] Melvin E. Conway. „How do Committees Invent?“ In: *Datamation* 14.5 (Apr. 1968) (siehe S. 2).

[Cos+05] Gennaro Costagliola u. a. „Class Point: An Approach for the Size Estimation of Object-Oriented Systems“. In: *IEEE Transactions on Software Engineering* 31.1 (Jan. 2005), S. 52–74 (siehe S. 4).

[Cou85] P. J. Courtois. „On time and space decomposition of complex structures“. In: *Communications of the ACM* 28.6 (1985), S. 590–603 (siehe S. 68, 83).

[CY90] Peter Coad und Edward Yourdon. *Object Oriented Analysis*. 2. Auflage. Englewood Cliffs, New Jersey 07632: Yourdon Press, Nov. 1990 (siehe S. 54, 69).

[DD04] Paul J. Deitel und Harvey M. Deitel. *Java How to Program*. 6. Auflage. Prentice Hall International, Aug. 2004 (siehe S. 90).

[DDH72] Ole-Johan Dahl, Edsger W. Dijkstra und C. A. R. Hoare. *Structured Programming*. London, New York: Academic Press, Feb. 1972 (siehe S. 77).

[Dij69] Edsger W. Dijkstra. „Structured programming“. circulated privately. Aug. 1969. URL: http://www.cs.utexas.edu/users/EWD/ewd02xx/EWD268.PDF (siehe S. 3).

[Dij70a] Edsger W. Dijkstra. „Notes on Structured Programming“. circulated privately. Apr. 1970. URL: http://www.cs.utexas.edu/users/EWD/ewd02xx/EWD249.PDF (siehe S. 3).

[Dij70b] Edsger W. Dijkstra. „Structured programming“. In: *Software Engineering Techniques*. NATO Science Committee, Aug. 1970 (siehe S. 3).

[DL05] Stéphane Ducasse und Michele Lanza. „The Class Blueprint: Visually Supporting the Understanding of Classes“. In: *IEEE Transactions on Software Engineering* 31.1 (Jan. 2005), S. 75–90 (siehe S. 22, 23).

[DMG07] Paul M. Duvall, Steve Matyas und Andrew Glover. *Continuous Integration: Improving Software Quality and Reducing Risk*. Addison-Wesley, Juni 2007 (siehe S. 78).

[DP09] Stéphane Ducasse und Damien Pollet. „Software Architecture Reconstruction. A Process-Oriented Taxonomy“. In: *IEEE Transactions on Software Engineering* 35.4 (Juli 2009), S. 573–591. ISSN: 0098-5589 (siehe S. 2).

[EDB04] Christof Ebert, Reiner Dumke und Manfred Bundschuh. *Best Practices in Software Measurement. How to use metrics to improve project and process performance*. Berlin, Heidelberg, New York: Springer, Sep. 2004 (siehe S. 22).

[Ede93] D. Vera Edelstein. „Report on the IEEE STD 1219-1993–standard for software maintenance“. In: *SIGSOFT Softw. Eng. Notes* 18.4 (1993), S. 94–95 (siehe S. 1, 2).

[Eic+98] Stephen G. Eick u. a. „Does Code Decay? Assessing the Evidence from Change Management Data“. In: *Technical Report NISS* 81 (März 1998) (siehe S. 2).

[Erl00] L. Erlikh. „Leveraging Legacy System Dollars for EBusiness“. In: *IT Pro* May/June (2000), S. 17–23 (siehe S. 1).

[FFC99] David Flanagan, Jim Farley und William Crawford. *Java Enterprise in a Nutshell. A Desktop Quick Reference in a Nutshell*. Sebastopol, CA: O'Reilly Associates, Mai 1999 (siehe S. 91).

[Fis08] Gerd Fischer. *Lineare Algebra. Eine Einführung für Studienanfänger*. 15., verbesserte Auflage. Vieweg+Teubner, Aug. 2008 (siehe S. 101).

[Flo67] Robert W. Floyd. „Assigning meanings to programs“. In: *Proceedings of Symposia in Applied Mathematics*. Bd. XIX. American Math. Society, 1967, S. 19–32 (siehe S. 74).

[FN01] Fabrizio Fioravanti und Paolo Nesi. „Estimation and Prediction Metrics for Adaptive Maintenance Effort of Object-Oriented Systems“. In: *IEEE Transactions on Software Engineering* 27.12 (Dez. 2001), S. 1062–1084 (siehe S. 4).

[Fow99] Martin Fowler. *Refactoring. Improving the Design of Existing Code*. Amsterdam: Addison-Wesley Longman, 1999. ISBN: 0-201-48567-2 (siehe S. 3, 4, 7, 22).

[FPT02] Ludwig Fahrmeir, Iris Pigeot und Gerhard Tutz. *Statistik: Der Weg zur Datenanalyse*. 4., verbesserte Auflage. Berlin: Springer, Sep. 2002 (siehe S. 102).

[GA04] Miguel Goulão und O Brito E Abreu. „Cross-Validation of a Component Metrics Suite“. In: *In Proceedings of the IX Jordnadas de Ingeniería del Software y Bases de Datos* (2004) (siehe S. 22).

[Gam+96] Erich Gamma u. a. *Entwurfsmuster*. 5. korrigierte Auflage. München et. al.: Addison-Wesley, 1996. ISBN: 3-8273-1862-9 (siehe S. 4, 7, 26, 40, 49, 67, 92).

[Gil77] Tom Gilb. *Software Metrics*. Cambridge, Massachusetts: Winthrop Publishers, Inc., Dez. 1977 (siehe S. 20).

[Gos+03] Katerina Goseva-Popstojanova u. a. „Architectural-Level Risk Analysis Using UML“. In: *IEEE Transactions on Software Engineering* 29.10 (Okt. 2003), S. 946–960 (siehe S. 4).

[GPC05] Marcela Genero, Mario Piattini und Coral Calero. „A Survey of Metrics for UML Class Diagrams“. In: *Journal of Object Technology* 9.4 (Dez. 2005), S. 59–92 (siehe S. 22).

[GS94] David Garlan und Mary Shaw. *An Introduction to Software Architecture*. Carnegie Mellon University, 1994 (siehe S. 37, 38).

[Hal77] Maurice H. Halstead. *Elements of Software Science*. Amsterdam: Elesevier Scientific Publishing Company, 1977 (siehe S. 4, 20).

[Has06] Wilhelm Hasselbring. „Software-Architektur“. In: *Informatik-Spektrum* 29.1 (2006), S. 48–52 (siehe S. 29).

[HCN98] Rachel Harrison, Steve J. Counsell und Reuben V. Nithi. „An Evaluation of the MOOD Set of Object-Oriented Software Metrics“. In: *IEEE Transactions on Software Engineering* 24.6 (Juni 1998), S. 491–496 (siehe S. 4, 21, 101, 103, 116).

[Hei10] Sven Heidemann. „Informationen zur Führung einer Fakultät. Erprobung eines Frameworks“. Diplomarbeit. Bielefeld: Universität Bielefeld, 2010 (siehe S. 132).

[Hen95] Brian Henderson-Sellers. *Object-Oriented Metrics: Measures of Complexity*. Faksimile. Prentice Hall, Dez. 1995 (siehe S. 103).

[Hev+04] Alan R. Hevner u. a. „Design Science in Information System Research“. In: *MIS Quartely* 28 (2004), S. 75–105 (siehe S. 5).

[Hil00] Rich Hilliard. „Views as Modules“. In: *Proceedings 4th International Software Architecture Workshop (ISAW-4), 4 and 5 June 2000* (2000), S. 7–10 (siehe S. 59).

[Hil99] Rich Hilliard. „Aspects, Concerns, Subjects, Views, …“ In: *First Workshop on Multi-Dimensional Separation of Concerns in Object-oriented Systems (at OOPSLA '99)*. 1999 (siehe S. 59).

[HK81] S. Henry und D. Kafura. „Software Structure Metrics Based on Information Flow“. In: *IEEE Trans. Softw. Eng.* 7.5 (Sep. 1981), S. 510–518 (siehe S. 20).

[HS01] T. E. Hastings und A. S. M. Sajeev. „A Vector-Based Approach to Software Size Measurement and Effort Estimation“. In: *IEEE Transactions on Software Engineering* 27.4 (Apr. 2001), S. 337–350 (siehe S. 4).

[HT03] Andrew Hunt und David Thomas. *Der Pragmatische Programmierer*. München, Wien: Carl Hanser, 2003 (siehe S. 2).

[Huc10] Sebastian Hucke. „Unterstützung des Lehrprozesses einer Fakultät durch Führungsinformationen. Informationskonzept und teilweise Implementierung zur Beurteilung eines Frameworks“. Diplomarbeit. Bielefeld: Universität Bielefeld, 2010 (siehe S. 132).

[Hus01] Brian Huston. „The effects of design pattern application on metric scores“. In: *Journal of Systems and Software* 58.3 (Sep. 2001), S. 261–269 (siehe S. 22).

[IEE00] IEEE. „IEEE Std 1471-2000: IEEE Recommended Practice for Architectural Description of Software-Intensive Systems“. In: (Sep. 2000) (siehe S. 24, 30, 31, 33).

[ISO00] ISO/IEC. *ISO/IEC 15288: Life Cycle Management – System Life Cycle Processes*. ISO/IEC, 2000 (siehe S. 8).

[JES03] Jean-Marie Favre Jacky, Jacky Estublier und Remy Sanlaville. „Tool Adoption Issues in a Very Large Software Company“. In: *In Proceedings of 3rd International Workshop on Adoptioncentric Software Engineering (ACSE'03)* 9 (2003), S. 81–89 (siehe S. 24).

[Joh02] Rod Johnson. *Expert One-on-One J2EE Design and Development*. John Wiley & Sons, Okt. 2002 (siehe S. 91).

[Kan02] Stephen H. Kan. *Metrics and Models in Software Quality Engineering*. 2. Auflage. Amsterdam: Addison-Wesley Longman, Sep. 2002 (siehe S. 19, 21, 22, 101, 103, 105).

[Ker05] Joshua Kerievsky. *Refactoring to Patterns*. The Addison Wesley Signature Series. Boston-San Franciso et. al.: Addison-Wesley, 2005. ISBN: 978-0321213358 (siehe S. 7).

[Ker06] Joshua Kerievsky. *Refactoring to Patterns*. München: Addison-Wesley, 2006 (siehe S. 4).

[Knu84] Donald E. Knuth. „Literate Programming“. In: *The Computer Journal* 27.2 (1984), S. 97–111 (siehe S. 131).

[Kos05] Rainer Koschke. „Rekonstruktion von Software-Architekturen“. In: *Informatik Forschung und Entwicklung* 19.3 (Apr. 2005), S. 127–140 (siehe S. 3, 17, 24, 25, 27).

[Krü06] Guido Krüger. *Handbuch der Java-Programmierung*. München, Boston: Addison-Wesley, 2006 (siehe S. 86, 91).

[Kru95] P.B. Kruchten. „The 4+1 View Model of architecture“. In: *IEEE Software* 12.6 (Nov. 1995), S. 42–50 (siehe S. 3, 8, 44, 46, 47).

[Lan03] Michele Lanza. „Object-Oriented Reverse Engineering: Coarse-grained, Fine- grained, and Evolutionary Software Visualization“. Diss. Switzerland, 2003 (siehe S. 115).

[Lap10] Phillip A. Laplante, Hrsg. *Encyclopedia of Software Engineering*. Bd. 2. Boca Raton et al.: Auerbach Publications, 2010. ISBN: 978-1420059779.

[Lar04] Craig Larman. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development*. 3. Auflage. Prentice Hall, Nov. 2004 (siehe S. 26, 47).

[LC01] Victor Laing und Charles Coleman. *Principal Components of Orthogonal Object-Oriented Metrics*. White Paper. NASA Software Assurance Technology Center, 2001. URL: http://satc.gsfc.nasa.gov/support/OSMASAS_SEP01/Principal_Components_of_Orthogonal_Object_Oriented_Metrics.pdf (siehe S. 4).

[Leh+97] M. M. Lehman u. a. „Metrics and Laws of Software Evolution - The Nineties View“. In: *Proceedings Metrics 97 Symposium, Nov. 5-7th* (1997) (siehe S. 1).

[LH89] Karl J. Lieberherr und Ian Holland. „Assuring Good Style for Object-Oriented Programs“. In: *IEEE SOFTWARE* 6 (1989), S. 38–48 (siehe S. 20, 57, 72).

[LH93] Wei Li und Sallie Henry. „Object-oriented metrics that predict maintainability“. In: *Journal of Systems and Software* 23.2 (1993), S. 111–122 (siehe S. 101, 103).

[Lis87] Barbara Liskov. „Data Abstraction and Hierarchy“. In: *ACM SIGPLAN Notices*. Bd. 23 (5). Mai 1987, S. 17–34 (siehe S. 21, 87).

[LK94] Mark Lorenz und Jeff Kidd. *Object-Oriented Software Metrics*. Upper Saddle River, NJ, USA: Prentice Hall, Juli 1994 (siehe S. 101, 103).

[LM06] Michele Lanza und Radu Marinescu. *Object-Oriented Metrics in Practice: Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems*. Berlin, Heidelberg, New York: Springer, Okt. 2006 (siehe S. 22, 99, 105, 118).

[LP90] Wilf R. LaLonde und John R. Pugh. *Inside Smalltalk*. Bd. 1. London: Prentice Hall, 1990 (siehe S. 87).

[LW99] Chi Tau Robert Lai und David M. Weiss. *Software Product Line Engineering: A Family-Based Software Development Process [With CDROM]*. Reading, Massachusetts et al.: Addison Wesley, Aug. 1999 (siehe S. 56).

[Mar02] Robert Cecil Martin. *Agile Software Development. Principles, Patterns, and Practices*. Upper Saddle River, New Jersey 07458: Prentice Hall International, Nov. 2002 (siehe S. 7, 21, 92, 93, 101, 103, 112, 116).

[Mar09] Robert C. Martin. *Clean Code. Refactoring, Patterns, Testen und Technik für sauberen Code*. Heidelberg et al.: mitp, 2009. ISBN: 978-3-8266-5548-7 (siehe S. 2, 7).

[Mar96a] Robert C. Martin. „Granularity“. In: *C++ Report* 8 (Dez. 1996) (siehe S. 21).

[Mar96b] Robert C. Martin. „The Dependency Inversion Principle“. In: *C++ Report* 8 (Mai 1996) (siehe S. 21).

[Mar96c] Robert C. Martin. „The Interface Segregation Principle“. In: *C++ Report* 8 (Aug. 1996) (siehe S. 21).

[Mar96d] Robert C. Martin. „The Liskov Substitution Principle“. In: *C++ Report* 8 (März 1996) (siehe S. 21).

[Mar96e] Robert C. Martin. „The Open-Closed Principle“. In: *C++ Report* 8 (Jan. 1996) (siehe S. 21).

[Mar97] Robert C. Martin. „Stability“. In: *C++ Report* (Feb. 1997) (siehe S. 21).

[Mas10] Dieter Masak. *Der Architekturreview*. Berlin Heidelberg: Springer, 2010 (siehe S. 2).

[McC76] Thomas J. McCabe. „A Complexity Measure“. In: *IEEE Transactions on Software Engineering* SE-2.4 (1976), S. 308–320 (siehe S. 3, 20, 101, 102).

[McC96] Steve McConnell. *Rapid Development*. Redmond, Washington: Microsoft Press, 1996 (siehe S. 1).

[Mey97] Bertrand Meyer. *Object-oriented Software Construction*. 2. Auflage. Upper Saddle River 07458: Prentice Hall International, Mai 1997 (siehe S. 21, 25, 68, 71, 73–76, 91).

[Mil56] George A. Miller. „The Magical Number Seven, Plus or Minus Two: Some Limits on Our Capacity for Processing Information“. In: *The Psychological Review* 63 (1956), S. 81–97 (siehe S. 68, 69, 83).

[Mor07] Rajiv Mordani, Hrsg. *JSR 154: Java(TM) Servlet 2.4 Specification*. Sep. 2007. URL: http://jcp.org/en/jsr/detail?id=154 (besucht am 20. 11. 2010) (siehe S. 114).

[MS01] Taichi Muraki und Motoshi Saeki. „Metrics for applying GOF design patterns in refactoring processes“. In: *Proceedings of the 4th International Workshop on Principles of Software Evolution*. Vienna, Austria: ACM, 2001, S. 27–36 (siehe S. 22).

[MWT94] H. Müller, K. Wong und S. Tilley. „Understanding software systems using reverse engineering technology“. In: *The 62nd Congress of L'Association Canadienne Francaise pour l'Avancement des Sciences Proceedings (ACFAS)* (1994) (siehe S. 1, 2).

[Mye75] Glenford J. Myers. *Reliable Software Through Composite Design*. New York: Petrocelli/Charter, 1975 (siehe S. 20).

[Öst+10] Hubert Österle u. a. „Memorandum zur gestaltungsorientierten Wirtschaftsinformatik“. In: *Zeitschrift für betriebswirtschaftliche Forschung* 62.6 (Sep. 2010), S. 664–672 (siehe S. 5).

[Par72] D. L. Parnas. „On the criteria to be used in decomposing systems into modules“. In: *Commun. ACM* 15.12 (1972), S. 1053–1058 (siehe S. 86).

[Par75] D. L. Parnas. „The influence of software structure on reliability“. In: *Proceedings of the international conference on Reliable software*. Los Angeles, California: ACM, 1975, S. 358–362 (siehe S. 19, 34).

[PI94] Roger S. Pressman und Darrel Ince. *Software Engineering: A Practitioner's Approach*. 4. aktualisierte Auflage. New York et al.: McGraw-Hill Publishing Co., Apr. 1994 (siehe S. 101).

[Pir06] Robert M. Pirsig. *Zen und die Kunst ein Motorrad zu warten: Ein Versuch über Werte*. 29. Auflage. Fischer (Tb.), Frankfurt, Juli 2006 (siehe S. 60).

[PK90] D. E. Perry und G. E. Kaiser. „Adequate testing and object-oriented programming“. In: *J. Object Oriented Program.* 2.5 (1990), S. 13–19 (siehe S. 78).

[PW92] Dewayne E. Perry und Alexander L. Wolf. „Foundations for the Study of Software Architecture“. In: *Software Engineering Notes* 17.4 (Okt. 1992) (siehe S. 28, 33, 36).

[Ram08] Norman Ramsey. *Noweb homepage*. 2008. URL: http://www.cs.tufts.edu/~nr/noweb/ (besucht am 01. 11. 2010) (siehe S. 132).

[Ras00] Jef Raskin. *The Humane Interface. New Directions for Designing Interactive Systems*. Amsterdam: Addison-Wesley Longman, Apr. 2000 (siehe S. 13).

[RL04] Stefan Roock und Martin Lippert. *Refactoring in großen Softwareprojekten*. Heidelberg: dpunkt, 2004. ISBN: 978-3898642071 (siehe S. 3, 4, 7, 69, 102, 105).

[Rup07] Chris Rupp. *Requirements-Engineering und -Management. Professionelle, iterative Anforderungsanalyse für die Praxis*. 4. aktualisierte und erweiterte Auflage. Hanser Fachbuchverlag, 2007 (siehe S. 59).

[RW10] Václav Rajich und Leon Wilson. „Program Comprehension“. In: *Encyclopedia of Software Engineering*. Hrsg. von Phillip A. Laplante. Bd. 2. Boca Raton et al.: Auerbach Publications, 2010, S. 753–770. ISBN: 978-1420059779 (siehe S. 3).

[SD00] Johannes Siederleben und Ernst Denert. „Wie baut man Informationssysteme? Überlegungen zur Standardarchitektur“. In: *Informatik-Spektrum* 23.4 (Aug. 2000), S. 247–257 (siehe S. 55).

[SE89] E. Soloway und K. Ehrlich. „Empirical studies of programming knowledge“. In: *Software reusability: vol. 2, applications and experience*. ACM, 1989, S. 235–267 (siehe S. 69).

[SG96] Mary Shaw und David Garlan. *Software Architecture: Perspectives on an emerging discipline*. Upper Saddle River, New Jersey 07458: Prentice-Hall, Inc., Simon & Schuster / A Viacom Company, 1996 (siehe S. 26, 28, 33, 37).

[SHT05] Harry M. Sneed, Martin Hasitschka und Maria-Therese Teichmann. *Software-Produktmanagement*. Heidelberg: dpunkt, 2005 (siehe S. 2).

[SI93] Martin Shepperd und Darrel Ince. *Derivation and Validation of Software Metrics*. Oxford: Oxford University Press, Sep. 1993 (siehe S. 20, 21).

[Sie04] Johannes Siedersleben. *Moderne Software-Architektur: Umsichtig planen, robust bauen mit Quasar*. Heidelberg: dpunkt, Juli 2004 (siehe S. 8, 57).

[SNH95] Dilip Soni, Robert L. Nord und Christine Hofmeister. „Software architecture in industrial applications“. In: *Proceedings of the 17th international conference on Software engineering*. Seattle, Washington, United States: ACM, 1995, S. 196–207 (siehe S. 43).

[Sol+88] Elliot Soloway u. a. „Designing documentation to compensate for delocalized plans“. In: *Commun. ACM* 31.11 (1988), S. 1259–1267 (siehe S. 69).

[Som07a] Ian Sommerville. *Software Engineering*. 8. Auflage. Harlow et al.: Addison Wesley, 2007 (siehe S. 1, 2).

[Som07b] Ian Sommerville. *Software Engineering*. 6. Auflage. Harlow et al.: Addison Wesley, 2007 (siehe S. 3).

[Spi06] Thorsten Spitta. *Informationswirtschaft: Eine Einführung*. Berlin Heidelberg: Springer, 2006 (siehe S. 1).

[ST07] Gernot Starke und Stefan Tilkov, Hrsg. *SOA-Expertenwissen: Methoden, Konzepte und Praxis serviceorientierter Architekturen*. Heidelberg: dpunkt, Juni 2007 (siehe S. 40, 59).

[Szy99] Clemens Szyperski. *Component Software*. Harlow, England et. al.: Addison-Wesley, 1999 (siehe S. 53–55, 88).

[Teß04] Meik Teßmer. „Architekturermittlung aus Quellcode. Möglichkeiten und Grenzen anhand einer Fallstudie“. Diplomarbeit. Bielefeld: Universität Bielefeld, 2004 (siehe S. 69, 80, 116).

[Ulr90] W. M. Ulrich. „The evolutionary growth of software reengineering and the decade ahead“. In: *American Programmer* 3.10 (1990), S. 14–20 (siehe S. 2).

[Wal01] Ernest Wallmüller. *Software - Qualitätssicherung in der Praxis*. Hanser Fachbuch, Aug. 2001 (siehe S. 8, 9, 16, 60, 65, 83).

[War10] Brian Warner. *BuildBot*. 2010. URL: `http://buildbot.net` (besucht am 11. 11. 2010) (siehe S. 78).

[Wey88] E. J. Weyuker. „The evaluation of program-based software test data adequacy criteria“. In: *Commun. ACM* 31.6 (1988), S. 668–675 (siehe S. 78).

[WYF03] Hironori Washizaki, Hirokazu Yamamoto und Yoshiaki Fukazawa. „A Metrics Suite for Measuring Reusability of Software Components“. In: 2003, S. 211–223 (siehe S. 4, 22).

[YC79] Edward Yourdon und Larry L. Constantine. *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design: Fundamentals of a Discipline of Computer Programme and Systems Design*. Englewood Cliffs, New Jersey 07632: Prentice Hall, 1979 (siehe S. 20, 25, 26, 69, 71, 72, 92, 112).

[Zac87] J. A. Zachman. „A framework for information systems architecture“. In: 26.3 (1987), S. 276–292 (siehe S. 28, 34, 35).

[Zel07] Stephan Zelewski. *Kann Wissenschaftstheorie behilflich sein für die Publikationspraxis?* Beiträge in Sammelwerken. Berlin, 2007, S. 71–120 (siehe S. 5).

[Zus85] Horst Zuse. „Meßtheoretische Analyse von statischen Softwarekomplexitätsmaßen“. Diss. Berlin: TU Berlin, 1985 (siehe S. 101).

[Zus98] Horst Zuse. *A Framework of Software Measurement*. Berlin, New York: de Gruyter, 1998 (siehe S. 17, 19, 20, 22, 101, 111).