Entwurf eines Quellcode basierten Qualitätsmodells für die Softwarewartung

Meik Teßmer

Bereich Computergestützte Methoden
Fakultät für Wirtschaftswissenschaften
Universität Bielefeld
Universitätsstraße 25
33615 Bielefeld
mtessmer@wiwi.uni-bielefeld.de

Abstract: Die Wartbarkeit eines Softwaresystems ist ein wichtiger Faktor für seinen langfristigen Einsatz. In diesem Beitrag wird ein Qualitätsmodell auf Quellcode-Basis entworfen, das den Wartbarkeitsbegriff konkretisiert und messbar macht. Ziel ist die Untersuchung der Qualität eines an der Universität entwickelten großen Systems zur Prüfungsverwaltung.

1 Nachhaltigkeit als Qualitätsaspekt von Quellcode

Der langfristig aufrecht erhaltene Betrieb eines großen Softwaresystems führt zu zwei Erkenntnissen: (1) Wartung und Evolution sind Teil des sich über den Zeitpunkt der Auslieferung hinaus erstreckenden Entwicklungsprozesses, denn ohne eine kontinuierliche Anpassung verliert das System für den Anwender nach und nach den Nutzen (s. [BL76, Leh+97]). (2) Die vorgenommenen Änderungen beeinflussen Umfang und Qualität des Quellcodes des Systems (s. [BL76, Eic+98, Leh+97, MWT94]).

Vergegenwärtigt man sich die Kosten der Wartungsarbeiten, die auf 70% und mehr der Gesamtkosten des Systems geschätzt werden (s. [Boe79, Ede93, Erl00]), ist die Berücksichtigung des Faktors Nachhaltigkeit bei der Entwicklung und Wartung für das betreibende Unternehmen eine ökonomische Notwendigkeit. Besonderes Augenmerk muss dabei auf den Quellcode gerichtet werden. Er ist im Gegensatz zur Softwaredokumentation die einzige *zuverlässige* Informationsquelle (s. [DP09, Mar09]), die alle (Struktur)-Entscheidungen der an der Entwicklung Beteiligten vereint.

Nach Müller, Wong und Tilly führen vorgenommene Änderungen wie Erweiterungen oder Updates bei einem durchschnittlichen Fortune 100-Unternehmen zu einer jährlichen Zunahme des Quellcode-Umfangs von ca. 10% (s. [MWT94]). Zugleich wird 60% der für die Wartung aufgewendeten Zeit allein für die Suche nach den relevanten Quellcode-Stellen benötigt (s. [Ede93]). Dabei erschwert nicht nur der Umfang die Analyse und das Verständnis des Quellcodes. Mit fortschreitender Wartung werden die Systeme immer

komplexer, während die Code-Qualität gleichzeitig abnimmt (s. [Leh+97]). Es kommt zu einem Verfall der Strukturen, der auch als *Code Decay* oder *Software-Entropie* bezeichnet wird (s. [Eic+98, HT03]). Ohne entsprechende Gegenmaßen besteht das Risiko, dass jede Fehlerkorrektur selbst wieder zu neuen Fehlern führt und das System schließlich nicht mehr wartbar ist (s. [BL76]).

Sinnvolle Gegenmaßnahmen erfordern die Entwicklung eines geeigneten *Qualitätsmodells*. Zusammen mit einem projektspezifischen *Softwaremodell* ermöglicht es die Ermittlung und Bewertung des Ist-Zustands der für die Wartung relevanten Strukturen im Quellcode. Aus den Analyseergebnissen lassen sich dann Gegenmaßnahmen ableiten, die auf die Verbesserung der Wartbarkeit abzielen. Ein solches Qualitätsmodell hat die Aufgabe, das allgemeine und unpräzise Verständnis von „Qualität“ durch eine Taxonomie von Einzelfaktoren operationalisierbar zu machen. Dazu werden Qualitätsunterbegriffe abgeleitet und diese weiter zu Indikatoren verfeinert, um sie einem Messinstrumentarium zugänglich zu machen. Die Interpretation der Messergebnisse erlaubt schließlich, Rückschlüsse auf die Gesamtqualität des Systems zu ziehen. Wartbarkeit als relevanter Aspekt der Nachhaltigkeit kann damit in die Qualitätssicherung integriert werden und den langfristigen Betrieb des Systems sicherstellen. Zusätzlich erreicht man eine Vereinheitlichung der Vorstellung von Qualität und macht sie dadurch für Entwickler und Management kommunizierbar (s. [Wal01]).

Die Entwicklung eines Qualitätsmodells wird von verschiedenen Fragen geleitet: Was ist „guter“ Code? Wodurch zeichnet sich eine wartbare Struktur aus? Wie stehen die verschiedenen Strukturmerkmale in Beziehung und welche Auswirkungen hat das auf die Wartbarkeit? Welche Grenzwerte sollten angelegt werden? Welche spezifischen Strukturierungsmöglichkeiten der verwendeten Programmiersprache haben Einfluss auf die Wartbarkeit? Erste Hinweise auf Antworten zu diesen Fragen liefern die verschiedenen Methoden zur Softwarevermessung durch *Metriken*. In Anlehnung an das Messen, wie es bspw. in der Physik stattfindet, wird versucht, durch die Quantifizierung qualitativer Aspekte Aussagen zu bestimmten Qualitäten von Softwaresystemen zu machen. Die Abbildung der Messwerte soll Rückschlüsse auf bestimmte Eigenschaften des Systems erlauben. Problematisch ist jedoch die Bestimmung der Grenzwerte, da sich bspw. absolute Zahlen für gute oder schlechte Größen von Klassen, Packages, Subsystemen oder Schichten nicht angeben lassen (s. [RL04]). Erschwerend kommt die nicht standardisierte Interpretation der Messergebnisse hinzu (s. [And+04]). Gründe dafür sind u.a. die unklare Definition von Begriffen wie Wartbarkeit. Zum Teil besteht aber auch 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 (s. [Agg+06]).

Trotz der schon angeführten Probleme unterstützen die Metrik-Suiten von McCabe (s. [McC76]), Halstead (s. [Hal77]), Chidamber und Kemerer (s. [CK91]), das MOOD-Set von Harrison et al. (s. [HCN98]) sowie Fokussierungen und Erweiterungen verschiedener qualitätsorientierter 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]).

Weitere Hinweise auf relevante Strukturen und ihre Bedeutung für die Wartbarkeit lassen sich aus den Arbeiten zu Entwurfsmustern ableiten (s. [Gam+96, Lar04]). Sie bieten Entwicklern zu häufig wiederkehrenden Problemstellungen bewährte Code-Strukturen als Lösungsvorschlag an und dienen als „Best Practices"-Leitfaden. Ebenso hilfreich sind die sog. *Code Smells*. Sie weisen auf kritische Stellen im Code hin, die durch Refactoring-Maßnahmen behoben werden können (s. [Fow99]). Solche Refactorings können auch auf der Architekturebene durchgeführt werden, wenn sog. *Architektur-Smells* vorliegen (s. [RL04]). Sie korrigieren dann keine Klassen(-verbände), sondern wirken auf der Ebene der Komponenten. Auf Entwurfsmuster ausgerichtete Refactorings, *Refactoring to Patterns*, sind eine weitere Möglichkeit, die Evolution von Systemen zu unterstützen (s. [Ker06]). Das *Clean Code*-Prinzip von Martin hingegen setzt auf die Intuition des Entwicklers als maßgebliches Qualitätskriterium: „Sauber" ist Quellcode genau dann, wenn er mit wenig Aufwand in kurzer Zeit richtig verstanden werden kann (s. [Mar09]). Diese Richtlinie lässt sich jedoch nicht direkt in ein Qualitätsmodell übertragen: Wo bei Entwurfsmustern und Code Smells mehr oder weniger eindeutige Qualitätsmerkmale identifiziert werden können, lässt sich die Intuition des Entwicklers nicht auf eindeutige Merkmale abbilden und so quantifizieren. Mit Hilfe von Ersatzmerkmalen, sog. *Prädiktoren* (s. [10e]), kann dennoch versucht werden, Eigenschaften wie Lesbarkeit und Verständlichkeit messbar zu machen und in ein Qualitätsmodell zu integrieren. Auf diese Weise lassen sich auch andere ähnlich unpräzise Hinweise wie bspw. Faustregeln zur Entwicklung einer guten Architektur (s. u. a. [BCK98]) messbar machen.

Der nächste Abschnitt befasst sich mit der Sammlung von Informationen, die für die Entwicklung eines Qualitätsmodells notwendig sind. Dabei werden unterschiedliche Quellen konsultiert: Der Standard ISO 9126, Veröffentlichungen zu Qualitätsmodellen sowie Arbeiten zur Softwarevermessung und zur Softwarearchitektur. Die Entwicklung zielt darauf ab, ein großes Softwaresystem der Universität Bielefeld, das BIS[1], zu analysieren und hinsichtlich seiner Wartbarkeit zu bewerten. Im dann folgenden Abschnitt wird das Qualitätsmodell entworfen und mit einem Softwaremodell konkretisiert. Die Untersuchung des BIS und abschließende Betrachtungen zu Möglichkeiten und Grenzen dieses Bewertungsansatzes bilden den letzten Abschnitt.

2 Vorüberlegungen

2.1 Aufbau eines Qualitätsmodells

Zunächst stellt sich die Frage, wie ein Qualitätsmodell entworfen sein sollte. Der ISO-Standard 9126 (mittlerweile von ISO 25000 abgelöst) beschreibt in vier Teilen ein Qualitätsmodell für die Bewertung der *Produktqualität* und der *Gebrauchsqualität*. Der erste Teil (ISO 9126-1) beschreibt die Zerlegung des Qualitätsbegriffs, der zweite und dritte

[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.

spezifiziert Kenngrößen für die externe bzw. interne Produktqualität. Im letzten Teil finden sich Kenngrößen zur Gebrauchsqualität.

Der Standard schlägt folgende Zerlegungssystematik vor:

Abbildung 5: Zerlegungssystematik nach ISO 9126-1

Die Bewertung der Wartbarkeit betrifft die *interne* Produktqualität, die der ISO-Standard zusammen mit der externen Produktqualität wie folgt untergliedert:

- Funktionalität: angebotene Funktionalität (Eignung, Genauigkeit, Interoperabilität, Sicherheit, Einhalten von Standards)
- Zuverlässigkeit: Leistungsniveau (Reife, Fehlertoleranz, Wiederherstellbarkeit, Einhalten von Standards)
- Nutzbarkeit: Einsatz des Systems aus Sicht des Anwenders (Verständlichkeit, Erlernbarkeit, Betriebsfähigkeit, Attraktivität, Einhalten von Standards)
- Effizienz: technische Aspekte des Einsatzes (Zeitverhalten, Ressourcenverwendung, Effizienzstandards)
- Wartbarkeit: Änderungen und Erweiterungen des Systems (Analysierbarkeit, Änderbarkeit, Stabilität, Testbarkeit, Einhalten von Standards)
- Portabilität: Übertragen auf andere Hardware-/Software-Plattformen (Anpassungsfähigkeit, Installierbarkeit, Koexistenz, Ersetzbarkeit, Portabilitätsstandards)

Die Wartbarkeit eines Systems wird vom Standard als eine Untereigenschaft der internen Produktqualität definiert. Angesichts der hohen Änderungsrate sowohl von Software- als auch Hardware-Plattformen erfordert der langfristige Betrieb eines Softwaresystems auch die Berücksichtigung von Portierungsfragen, weshalb diese Untereigenschaft des Modells ebenfalls mit in die Überlegungen einbezogen werden sollte.

Die in ISO 9126-3 vorgeschlagenen Kenngrößen zur Ermittlung von internen Merkmalen, wie sie in ISO 9126-1 beschrieben werden, beschränken sich auf Vorschläge zum Zählen von Funktionen, Kommentarzeilen, Logging-Funktionen usw. und liefern damit nur Hinweise auf mögliche Ausgestaltungen. Konkrete Bezüge zu wartungsrelevanten Strukturen im Quellcode fehlen und die Verwendung bekannter Metriken, die bei der

Beschreibung der Kenngrößen zu erwarten wäre, findet nur am Rande im Anhang E Erwähnung.

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 des Systems 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. 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“).

Der Katalog der Eigenschaften aus ISO 9126, ihre Zerlegungssystematik und der Versuch, dazu konkrete Kenngrößen zu definieren, sind nachvollziehbar und decken sich vom Ansatz her mit den Vorschlägen anderer Autoren (s. bspw. [Bot+04, Wal01]). Dementsprechend kann dieser Teil des Standards als Vorlage für die Definition eines Qualitätsmodells dienen. Bevor jedoch ein vollständiges Modell entwickelt werden kann, müssen zunächst die relevanten Artefakte und Strukturen identifiziert werden, die die Wartbarkeit beeinflussen können, um sie anschließend als Merkmal zu integrieren.

2.2 Wartungsrelevante Merkmale

Die Programmstruktur und die Modularität sind wichtige Aspekte bei der Entwicklung zuverlässiger Softwaresysteme. Die Relevanz dieser Aspekte wurde u. a. von Parnas (s. [Par75]) beschrieben. Das erste und auch heute noch sehr verbreitete Maß ist Lines of Code (LOC). Dieses 1955 vorgestellte Volumenmaß zählt die Anweisungen im Quellcode und erfasst so die Größe von Systemen und Systembestandteilen. Zudem gibt es eine starke Korrelation zu anderen Maßen (s. [SI93]).

Eine populäre Reihe von Komplexitätsmaßen wurde 1976 von McCabe aus der Graphentheorie abgeleitet (s. [McC76]). Halstead prägte in [Hal77] den Begriff „Software Science“ mit dem Ziel, wissenschaftliche Methoden auf Strukturen und Eigenschaften von Programmen anzuwenden. Seine Maße und die von McCabe gehören bis heute zu den bekanntesten Maßen überhaupt. Yourdon und Constantine befassten sich 1979 mit dem damals populären Paradigma der Strukturierten Programmierung (s. [YC79]). Sie stellten dabei auch Überlegungen zum Entwurf von Systemen an. Aufbauend auf den Arbeiten von Myers (s. [Mye75]) 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 (s. [HK81]).

Mit der Einführung der Objektorientierung mussten Maße angepasst und neue entwickelt werden, um die neuen Strukturierungsmöglichkeiten entsprechend zu berücksichtigen. 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“ (s. [LH89]). Sie versuchten, den Impact einer Änderung so effektiv wie möglich zu begrenzen. Chidamber und Kemerer veröffentlichten 1991 eine Reihe bis heute verbreiteter Maße für die Vermessung objektorientierter Systeme (s. [CK91]). 1993 entwarf Lorenz elf Maße, mit deren Hilfe das Design eines OO-Systems bewertet werden können sollte. Dazu machte er auch Vorschläge zu Grenzwerten für die Sprachen Smalltalk und C++ (s. [LK94]). In dieselbe Richtung arbeiteten auch Abreu und Carapuça mit ihrem MOOD-Metrik-Set (s. [AC94]).

Alle bisher aufgeführten Metriken untersuchen mehr oder weniger feingranulare Strukturen wie Funktionen, Methoden oder Klassen. Die Organisation großer Softwaresysteme erfordert jedoch eine Strukturierung auf höheren Abstraktionsebenen wie sie bspw. Module oder Pakete anbieten. Eine Qualitätsbewertung muss daher auch diese Ebene berücksichtigen. Martin überträgt dazu das Kopplungs- und Kohäsionskonzept der Klassen auf die Ebene der Pakete, indem er sechs Entwurfsprinzipien beschreibt (s. [Mar02]). Diese greifen z. T. Aspekte aus anderen Arbeiten auf, bspw. das Open-Close-Prinzip von Meyer (s. [Mey97]) oder das Liskovsche Ersetzungsprinzip (s. [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.

Lanza und Marinescu versuchen mit einem auf logischen UND- und ODER-Operationen basierenden Kompositionsmechanismus, die durch Filter auf die relevanten Werte reduzierten Messergebnisse zu kombinieren. Auf diese Weise können auch komplexere Design-Regeln untersucht werden (s. [LM06]). 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 mit den sog. Code Smells. Sie weisen auf Code-Abschnitte hin, die strukturell verbessert werden müssen (s. [Fow99]).

Insgesamt gibt es für die Bewertung von Strukturen auf den unteren Abstraktionsebenen eine ganze Reihe von Metriken, die jeweils bestimmte Aspekte der Strukturierung erfassen können. Die Frage ist, welche Merkmale für die Beurteilung der Wartbarkeit relevant sind. Laut Bass et al. zeigt sich die Güte der Änderbarkeit durch den *Impact*, der von einer Änderung ausgelöst wird (s. [BCK98]). Je geringer der Impact, desto größer die *Lokalität* der Änderung. Sie lässt sich zurückführen auf die *Kopplung* und *Kohäsion* von Artefakten der jeweiligen Abstraktionsebene und zielt damit auf die *Modularität* des Systems.

Die größten Risiken für die Strukturen eines Systems sowohl im Großen als auch im Kleinen sind nach Perry und Wolf die *Architekturdrift* (Verwässerung der Struktur durch Unkenntnis) und die *Architekturerosion* (massive Verletzung von Strukturen, bspw. durch Umgehung einer Schicht) (s. [PW92]). Um diese Risiken zu vermeiden bzw. zu

minimieren, muss der Entwickler sich ihrer bewusst werden. D .h., er muss in der Lage sein, die architekturrelevanten Artefakte und ihre Verbindungen entweder der zugehörigen Dokumentation zu entnehmen oder aus dem Quellcode ermitteln zu können. Damit spielt die Lesbarkeit des Quellcodes eine wichtige Rolle: Geringe Komplexität der Strukturen, überschaubare Größen und möglichst explizite Verbindungen der Artefakte [CY90, Szy99]. Das entspricht dem Clean Code-Prinzip von Martin und den Code Smells von Fowler: 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. Daraus lässt sich für eine Architektur ableiten:

> *Eine gute/verständliche Architektur ist hoch modular, besitzt eine geringe Komplexität bei gleichzeitig geringer Kopplung zwischen und hoher Kohäsion in den Komponenten. Sie folgt außerdem einer zentralen Strukturierungsidee.*

Hinsichtlich der Bewertung der Lesbarkeit, Komplexität und Größe darf jedoch der menschliche Faktor nicht vergessen werden, der bei der Entwicklung eines Systems und später bei der Wartung immer eine Rolle spielt. Wallmüller weist mit Bezug auf Pirsig darauf hin, dass Software-Produkte zu einem erheblichen Teil das Ergebnis intuitiver und kreativer Arbeit sind (s. [Wal01]). Er unterscheidet messbare Qualität in Form von Kenngrößen von der ebenfalls als qualitativ relevant anzusehenden kreativen Tätigkeit, die sich jedoch 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, aber sich nicht unbedingt in Form von Merkmalen niederschlägt. Wie bei den angesprochenen Metriken generell gilt hier im Besonderen, dass etwaige Grenzwerte je nach Projekt und Entwickler u. U. sehr unterschiedlich ausfallen können und ein Vergleich auch ähnlicher Projekte aufgrund dieser Subjektivität nur bedingt sinnvoll ist.

3 Entwurf des Qualitätsmodells

Wallmüller schlägt für die Entwicklung und Anwendung eines Qualitätsmodells sieben Schritte vor (s. [Wal01]):

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

Auf diese Weise soll verdeutlicht werden, *warum* gemessen wird, denn die Interpretation der Messwerte basiert immer auf einer bestimmten Hypothese. Zunächst stellt sich also 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 geeignet sind. Handelt es sich um die Bewertung von systeminternen Qualitätseigenschaften wie die Wartbarkeit, 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. Das Softwaremodell beeinflusst auch die Messmethoden und Werkzeuge, die für die Ermittlung der Messwerte in Frage kommen:

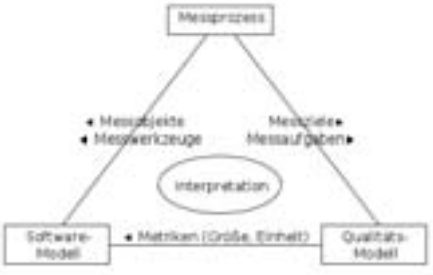

Abbildung 6: Beziehung von Messprozess, Software- und Qualitätsmodell

3.1 Begriffszerlegung

Aus den Vorüberlegungen und im Hinblick auf die geplante Untersuchung des BIS wird zunächst folgendes Messziel abgeleitet: *Das Messziel umfasst die Untersuchung und Bewertung der internen Qualität eines Softwaresystems in Form der Qualitätseigenschaften* Wartbarkeit *und* Portabilität.

Die Zerlegung orientiert sich an der im Standard ISO 9126 vorgeschlagenen Zerlegungssystematik:

Abbildung 7: Zerlegungssystematik für das Qualitätsmodell

Wartbarkeit wird vom ISO-Standard in die Untereigenschaften *Analysierbarkeit*, *Änderbarkeit*, *Stabilität*, *Testbarkeit* sowie *Standardkonformität* zerlegt. Portabilität geht nach näheren Untersuchungen nur über die Untereigenschaft *Anpassungsfähigkeit* in das Qualitätsmodell ein, die auf dieselbe Weise bewertet wird wie die *Änderbarkeit*. Damit ist eine nähere Untersuchung dieser Untereigenschaft nicht erforderlich. [Teß12]

Analysierbarkeit zielt auf das Systemverstehen ab. Einen Überblick zu Theorien und Untersuchungen findet sich in [BR04]. Wichtige Merkmale sind der Umfang von Entitäten, die Größe feingranularer Elemente (z. B. Anzahl Methoden pro Klasse) sowie die Anzahl der Elemente pro Struktureinheit (Hierarchisierung). Ebenso relevant sind die Anzahl von Kopplungen der Elemente sowie die Anzahl möglicher Pfade durch ein Element (Schleifen, Bedingungen). Die verwendete Programmiersprache definiert die möglichen hierarchischen Strukturen: Die Vererbung, die Organisation von Elementen in Klassen und Paketen sowie die Komposition von Komponenten. Zu beachten ist weiterhin der Einfluss redundanten Codes, sowohl was die Anzahl als auch seinen Umfang betrifft.

Die Güte der Änderbarkeit zeigt sich anhand der Strukturierung, genauer: in der Modularität des Quellcodes. Sie wird bestimmt durch die Kopplung und Kohäsion der einzelnen Elemente sowie 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).

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

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

Damit ergibt sich folgende Basis des Qualitätsmodells:

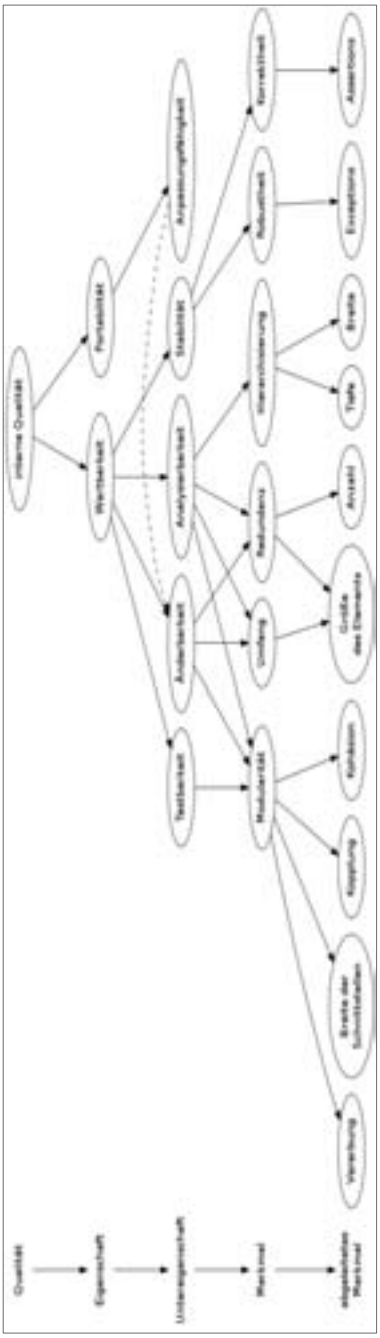

Abbildung 8: Basis des Qualitätsmodells

3.2 Softwaremodell

Softwaremodelle sind die Grundlage jeder statischen Analyse von Quellcode und haben die Aufgabe, die im Kontext der jeweiligen Untersuchung relevanten Attribute herauszustellen. Dabei beeinflussen sowohl die verwendete Programmiersprache als auch das Messziel die Menge dieser Merkmale. 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 Java gehört zu den objektorientierten Sprachen. Das für die Bewertung benötigte Softwaremodell muss daher die durch dieses Paradigma eingeführten Strukturierungsmöglichkeiten berücksichtigen. Aus Sicht des Qualitätsmodells sind folgende Merkmale relevant:

- 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]). Die zuverlässige automatische Erkennung von Subsystemen ist jedoch schwierig, daher werden sie hier nicht als Messobjekt berücksichtigt. Ähnlich verhält es sich mit Komponenten.

- 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 von 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 ist 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, 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]). Aus diesem Grund ist die Position eines Pakets innerhalb dieses Abhängig-

keitsgraphen von Interesse. Zusammen mit der Abstraktheit kann eine Überprüfung hinsichtlich des Stable-Abstraction-Prinzips erfolgen.

- Kohäsion: Die verschiedenen Formen der Kohäsion, wie sie u. a. von Yourdon und Constantine beschrieben wurden (s. [YC79]), untersuchen die Kohäsion 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 (s. [Mar02]). 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 Messaufgaben 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. 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. 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 Ebene der Methoden. Aus demselben Grund werden auch Attribute nicht mit aufgeführt.

- 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.

3.3 Bewertung von Kenngrößen

Für alle strukturrelevanten Merkmale sollten die ermittelten Kenngrößen im Intervall [0, 9] mit 0 als natürlichem Infimum und 9 als Indikatorgrenzwert liegen; das Teilintervall [3, 7] beschreibt einen akzeptablen Wertebereich für Kenngrößen. 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: #geerbter Attr./Meth. + #eigene Attr./Meth. − #überdeckter Attr./Meth. < 9. Die Vererbung über Paketgrenzen hinweg sollte allerdings minimal sein.

- 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 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. Die Bewertung eines Teilbaums der hierarchischen Struktur ergibt sich aus seiner Abweichung von einer idealen Struktur (max. 9 Methoden/Klasse, 9 Klassen/Paket usw.). 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 ein potentiell risikobehaftetes Messobjekt. Damit ist allerdings noch nicht die Frage beantwortet, wie die Ergebnisse der einzelnen Messaufgaben zu einer Aussage bzgl. des abgeleiteten Merkmals kombiniert werden können. Zuse hat in [Zus98] auf die verschiedenen Probleme solcher Kombinationen hingewiesen. Daher wird beim vorliegenden Qualitätsmodell die von Lanza und Marinescu vorgeschlagene Detection Strategy verwendet, die letztendlich nur eine aussagenlogische Komposition der Indikatorergebnisse ist. Praktisch bedeutet das, dass die Ergebnisse der Indikatorfunktionen eines abgeleiteten Merkmals mit Hilfe eines logischen Operators wie ∧ und ∨ kombiniert werden. 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.

4 Fallstudie

4.1 Das BIS

Das „Bielefelder Informationssystem“ (BIS) entstand 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 Vorle-

sungsverzeichnis (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.

Nach Einführung der Prüfungsverwaltung als letzte große Neuerung im Jahr 2006 gibt es seit 2011 die Möglichkeit, die an der Universität Bielefeld existierenden Studiengänge zu modellieren und damit eine automatische Notenberechnung anzustoßen.

4.2 Untersuchung der Morphologie

Eine Untersuchung befasste sich mit der Morphologie des Systems, insbesondere der des Pakets org.unibi.common. Ermittelt wurde Kopplung, Breite und Tiefe der Hierarchisierung sowie die Größe der Elemente.

Die Kopplung muss aufgrund der zyklischen Abhängigkeit als gravierend fehlerhaft bewertet werden. Die enge Kopplung von org.unibi.common und org.unibi.data mit 6770 Verbindungen 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“ anzusehen.

Insgesamt zeigen sich für die Untereigenschaften Analysierbarkeit, Änderbarkeit und Testbarkeit z. T. erschreckende Mängel, die dazu führen, dass die Wartbarkeit des Systems insgesamt hinterfragt werden muss. Diesbezügliche Gespräche mit dem Entwickler-Team sind geplant, inwieweit es aber zu entsprechenden Refactoring-Arbeiten kommen wird, ist angesichts des derzeit geplanten neuen Campus-Management-Systems unklar.

5 Möglichkeiten und Grenzen des Ansatzes

Das vorgestellte Qualitätsmodell ist nicht als Konkurrent zu bestehenden Ansätzen zu verstehen, die von den beschriebenen Softwaremaßen verwendet werden, 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 Wartbarkeit wichtigen Merkmale Modularität und Vererbung werden auf verschiedenen Strukturebenen erfasst und stellenweise 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 welchem 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 jedoch relativ problemlos realisieren. Es ist dann aber wiederum nicht eindeutig ersichtlich, wie sich die Bewertung der Strukturen im Detail ändert.

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 Eclipse Metrics-Plugins mit Hilfe von selbst entwickelten Python-Skripten ausgewertet. Zweckmäßig wäre eine Erweiterung des Plugins um diese Auswertungsfunktionalität.

Weiter untersucht werden muss außerdem die Berücksichtigung der Stabilität. Wann ist eine Methode bzw. Klasse so mit Assertions und Exceptions ausgestattet, dass sie als stabil gelten kann? Auch eine Ermittlung der Testabdeckung könnte dazu herangezogen werden, die allerdings aufgrund des Late Bindings schwierig zu ermitteln sein wird.

Literaturverzeichnis

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

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

[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.

[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.

[And+04] Christoph Andriessens u. a. QBench Projektergebnis: Stand der Technik. 2004. URL: http://www.qbench.de.

[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.

[BCK98] Len Bass, Paul Clements und Rick Kazman. Software Architecture in Practice. Reading et al.: Addison-Wesley, 1998.

[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.

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

[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.

[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.

[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.

[BR04] Marcel Bennicke und Heinrich Rust. Programmverstehen und statische Analysetechniken im Kontext des Reverse Engingeering und der Qualitätssicherung. Projektbericht ViSEK/025/D. Kaiserslautern: Fraunhofer IESE, Feb. 2004.

[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.

[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.

[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.

[CY90] Peter Coad und Edward Yourdon. Object Oriented Analysis. 2. Auflage. Englewood Cliffs, New Jersey 07632: Yourdon Press, Nov. 1990.

[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.

[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.

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

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

[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.

[Fow99] Martin Fowler. Refactoring. Improving the Design of Existing Code. Amsterdam: Addison-Wesley Longman, 1999. ISBN: 0-201-48567-2.

[Gam+96] Erich Gamma u. a. Entwurfsmuster. 5. korrigierte Auflage. München et. al.: Addison-Wesley, 1996. ISBN: 3-8273-1862-9.

[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.

[Hal77] Maurice H. Halstead. Elements of Software Science. Amsterdam: Elesevier Scientific Publishing Company, 1977.

[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.

[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.

[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.

[HT03] Andrew Hunt und David Thomas. Der Pragmatische Programmierer. München, Wien: Carl Hanser, 2003.

[Lar04] Craig Larman. Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development. 3. Auflage. Prentice Hall, Nov. 2004.

[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.

[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).

[LH89] Karl J. Lieberherr und Ian Holland. „Assuring Good Style for Object-Oriented Programs“. In: IEEE SOFTWARE 6 (1989), S. 38–48.

[Lis87] Barbara Liskov. „Data Abstraction and Hierarchy“. In: ACM SIGPLAN Notices. Bd. 23 (5). Mai 1987.

[LK94] Mark Lorenz und Jeff Kidd. Object-Oriented Software Metrics. Upper Saddle River, NJ, USA: Prentice Hall, Juli 1994.

[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.

[Mar02] Robert Cecil Martin. Agile Software Development. Principles, Patterns, and Practices. Upper Saddle River, New Jersey 07458: Prentice Hall International, Nov. 2002.

[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.

[McC76] Thomas J. McCabe. „A Complexity Measure“. In: IEEE Transactions on Software Engineering SE-2.4 (1976), S. 308–320.

[Mey97] Bertrand Meyer. Object-oriented Software Construction. 2. Auflage. Upper Saddle River 07458: Prentice Hall International, Mai 1997.

[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).

[Mye75] Glenford J. Myers. Reliable Software Through Composite Design. New York: Petrocelli/Charter, 1975.

[Par72] D. L. Parnas. „On the criteria to be used in decomposing systems into modules“. In: Commun. ACM 15.12 (1972), S. 1053–1058.

[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.

[RL04] Stefan Roock und Martin Lippert. Refactoring in großen Softwareprojekten. Heidelberg: dpunkt, 2004. ISBN: 978-3898642071.

[SI93] Martin Shepperd und Darrel Ince. Derivation and Validation of Software Metrics. Oxford: Oxford University Press, Sep. 1993.

[Szy99] Clemens Szyperski. Component Software. Harlow, England et. al.: Addison-Wesley, 1999.

[Teß12] Architekturbezogene Qualitätsmerkmale für die Softwarewartung: Entwurf eines Quellcode basierten Qualitätsmodells. Dissertation. Bielefeld: Universität Bielefeld, 2012.

[Wal01] Wallmüller, Ernest: Software – Qualitätssicherung in der Praxis. München et al.: Hanser Fachbuch, 2001.

[WYF03] Hironori Washizaki, Hirokazu Yamamoto und Yoshiaki Fukazawa. „A Metrics Suite for Measuring Reusability of Software Components“. In: 2003, S. 211–223.

[Zus98] Horst Zuse. A Framework of Software Measurement. Berlin, New York: de Gruyter, 1998.