Westsächsische Hochschule Zwickau
University of Applied Sciences

Masterarbeit

Relevante Metriken zur Bestimmung von Softwarequalität

Steffen Förster
geboren am 30. Juni 1986 in Bautzen
Studiengang Informatik

Westsächsische Hochschule Zwickau
Fakultät Physikalische Technik / Informatik
Fachgruppe Informatik

Betreuer	Einrichtung
Prof. Dr. rer. nat. habil. Wolfgang Golubski	WHZ
Dipl.- Wirt.- Inf. Claudia Eußert	N+P Informationssysteme GmbH
Abgabetermin	20. September 2012

Vielen Dank an Prof. Wolfgang Golubski für die erfrischenden und immer zielführenden Diskussionen und den Ansporn dieses Thema zu bearbeiten. Dank gebührt natürlich auch dem ganzen Team der Softwareentwicklung von N+P, insbesondere Claudia Eußert und Dirk Ziegenbalg, für ihre stets hilfreichen Hinweise und Anmerkungen. Ohne Sven wäre das Thema der Arbeit gar nicht zu Stande gekommen, daher hat auch er sich meinen Dank verdient.

Zu guter Letzt geht noch ein Dank an die wichtigste Person in meinem Leben: Deine Unterstützung während des ganzen Studiums hat mich so weit gebracht. Danke Kristin.

Die Qualität von Software lässt sich ohne eine ausgiebige Analyse nicht objektiv bewerten. Um diese Bewertung vornehmen zu können, sind Metriken das geeignete Mittel um die Beschaffenheit von Software konkret zu veranschaulichen. Diese Arbeit stellt den Weg von einem subjektiven Qualitätsgefühl für ein Softwareprodukt, hin zu einer objektiven Bewertung dar. Dazu fasst sie die Schritte in das Capability Maturity Model Integration (CMMI) Referenzmodell ein und stellt mit Hilfe von Goal Question Metric (GQM) ein Qualitätsmodell auf, dass den Anforderungen im betrachteten Projekt gerecht wird. Es erfolgt zuvor eine Evaluierung weit verbreiteter Tools und der Metriken, die mit ihnen berechnet werden können. Abschließend wird dargestellt, wie sich die Tools in den Entwicklungsprozess einbringen lassen und so eine beständige Qualitätsanalyse ermöglichen.

The quality of software cannot be evaluated without an extensive analysis. Evaluation could be done by using metrics which describe specific facts about the software. This thesis blazes the trail from the subjectively feeling to an objective software appraisal. It guides through the CMMI reference process and uses GQM to arrange a quality model. The first part compares wide spread tools and the metrics they provide. A guide how these tools can be integrated into the software development process is described at the end to enable a persistent quality analysis process.

Inhaltsverzeichnis

Abkürzungsverzeichnis **vii**

Abbildungsverzeichnis **xi**

Tabellenverzeichnis **xiii**

1 Einleitung **1**

1.1 Motivation 1

1.2 Grundlagen 3

1.3 Vorbetrachtungen 6

2 Softwarequalität **8**

2.1 Begriffsabgrenzung 8

2.2 Indikatoren 10

2.3 Anwendungsmöglichkeiten 12

3 Bestandsaufnahme **14**

4 Metriken **20**

4.1 Größenmetriken 20

4.2 Komplexitätsmetriken 21

4.3 Architekturmetriken 23

4.4 SQALE Metriken 25

4.5 Weitere Metriken 27

5 Nutzungsstrategien **30**

5.1 Capability Maturity Model Integration 30
5.2 Goal Question Metric 33
5.3 SQALE 36
5.4 Toolbasierende Erhebung 37
5.5 Continuous Integration & Inspectation 39

6 Werkzeuge **41**
6.1 Standalone Werkzeuge 41
6.1.1 JDepend 41
6.1.2 iPlasma und inFusion 42
6.1.3 CodeCity 43
6.1.4 STAN4J 43
6.2 Integrierte Werkzeuge 45
6.2.1 Eclipse Metrics 45
6.2.2 X-Ray 45
6.2.3 Checkstyle 46
6.2.4 Findbugs 47
6.2.5 PMD 48
6.2.6 Sonar 49

7 Qualitätsmodell **52**
7.1 Ausgangspunkt 52
7.2 Interview 52
7.3 Ziel 55
7.4 Fragen 56
7.5 Metriken 59

8 Implementierung **65**

9 Bewertungsergebnisse **69**

10 Fazit **72**

11 Ausblick **76**

A Anhang **a**

A.1 Entwicklungsstatistiken . . . . . a

A.2 Entwicklung einzelner Metriken . . . . . e

A.3 CD-Inhalt . . . . . h

Literaturverzeichnis **i**

Glossar **q**

Thesen **t**

Abkürzungsverzeichnis

6σ	Six Sigma
A	Abstractness
AHF	Attribute Hiding Factor
AIF	Attribute Inheritance Factor
API	Application Programming Interface
Ca	Afferent Couplings (eingehende Klassenkopplungen)
CBO	Coupling Between Objects
CC	Cyclomatic Complexity
CD	Continuous Delivery
Ce	Efferent Couplings (ausgehende Klassenkopplungen)
CIns	Continuous Inspectation
CInt	Continuous Integration
CK	Chidamber und Kemerer
CM	Configuration Management
CMM	Capability Maturity Model
CMMI	Capability Maturity Model Integration
CMMI-ACQ	Capability Maturity Model Integration for Acquisition
CMMI-DEV	Capability Maturity Model Integration for Development
CMMI-SVC	Capability Maturity Model Integration for Services
COBIT	Control Objectives for Information and Related Technology
COCOMOII	Constructive Cost Modell Version 2

COF	Coupling Factor
CPD	Copy-Paste-Detector
CS	Class Size
CVS	Concurrent Versions System
D	Martin Distanz
DIT	Depth of Inheritance Tree
FCM	Factors Criteria Metric
FOut	Fan Out (ausgehende Methodenkopplungen)
FP	Function Points
FRE	Flesch Reading Ease
GENESEZ	Generative Software Entwicklung Zwickau
GQM	Goal Question Metric
HIT	Heigth of the Inheritance Tree
I	Instability
IDE	Integrated Development Environment
ITIL	IT Infrastructure Libary
JEE	Java Enterprise Edition
LCOM	Lack of Cohesion of Methods
LCOM4	Lack of Cohesion of Methods Version 4
LoC	Lines of Code
MA	Measurement and Analysis
MHF	Method Hiding Factor
MIF	Method Inheritance Factor

MOOD	Metrics for Object-oriented Design
NCSS	Non Commented Source Statements
NDD	Number of direct Decendents
NOA	Number of Operation Added
NOC	Number of Children of a Class
NOM	Number of Methods of a Class
NOO	Number of Operation Overridden
NOP	Number of Packages
PMC	Project Monitoring and Control
POF	Polymorphie Faktor
PP	Project Planning
PPQA	Process and Product Quality Assurance
PRINCE2	Projects in Controlled Environments
QM	Qualitätsmanagement
REQM	Requirements Management
RFC	Response For a Class
ROI	Return on Investment
RSLOC	redundanzfreie Source Lines of Code
SAM	Supplier Agreement Management
SBII	SQALE Business Impact Index
SBIID	SQALE Business Impact Index Density
SCCI	SQALE Consolidated Changeability Index
SCEI	SQALE Consolidated Efficiency Index
SCI	SQALE Changeability Index
SCID	SQALE Changeability Index Density
SCM	Source Code Management

SCMI	SQALE Consolidated Maintainability Index
SCPI	SQALE Consolidated Portability Index
SCRI	SQALE Consolidated Reliability Index
SCRuI	SQALE Consolidated Reusability Index
SCSI	SQALE Consolidated Security Index
SEI	SQALE Efficiency Index
SEID	SQALE Efficiency Index Density
SI	Specialization Index
SMI	SQALE Maintainability Index
SMID	SQALE Maintainability Index Density
SPI	SQALE Portability Index
SPICE	Software Process Improvement and Capability Determination
SPID	SQALE Portability Index Density
SQALE	Software Quality Assessment based on Lifecycle Expectations
SRI	SQALE Reliability Index
SRID	SQALE Reliability Index Density
SRuI	SQALE Reusability Index
SRuID	SQALE Reusability Index Density
SSI	SQALE Security Index
SSID	SQALE Security Index Density
STI	SQALE Testability Index
STID	SQALE Testability Index Density
SVN	Subversion
TCO	Total Cost of Ownership
TPTP	Test and Performance Tools Platform
UI	User Interface
WMC	Weighed Methods per Class

Abbildungsverzeichnis

2.1 McCalls Qualitätsmatrix 11

3.1 Arbeitsablauf Featurewünsche 16

4.1 Typische Größenmetriken 20
4.2 Abstractness Instability Diagramm 24

5.1 Die Komponenten eines CMMI-Prozessgebietes 31
5.2 Schematischer Aufbau des GQM-Vorgehens 33
5.3 Darstellung des GQM-Baumes eines Ziels 35
5.4 Kivat einer SQALE Analyse 38

6.1 Screenshot von JDepend 41
6.2 Screenshot von iPlasma 42
6.3 Screenshot der untersuchten Codebasis aus CodeCity 43
6.4 Screenshot von STAN4J 44
6.5 Screenshots von X-Ray 46
6.6 Architektur von Sonar 50

9.1 Entwicklung der Richtlinieneinhaltung 69

A.1 Abgeschlossene Tickets pro Tag nach Version . . . . . d
A.2 Entwicklung der Codezeilen . . . . . e
A.3 Entwicklung des Kommentaranteils . . . . . e
A.4 Entwicklung des duplizierten Codeanteils . . . . . f
A.5 Entwicklung der dokumentierten Public API . . . . . f

A.6 Entwicklung der durchschnittlichen Komplexität pro Klasse . . . . . g
A.7 Entwicklung der technischen Schuld . . . . . g

Tabellenverzeichnis

1.1 Skalentypen 4
1.2 Gütekriterien für die Erhebung von Metriken 6

4.2 Überblick der SQALE Metriken 26

5.1 CMMI Levels 32
5.2 Erreichte Capability Levels vor den bisherigen Veränderungen 33
5.3 Beispiel eines SQALE Rating 37

10.1 Capability Levels nach der bisherigen Entwicklung 72

A.1 Erfasste Tickets nach Version . . . . . d

1 Einleitung

1.1 Motivation

In klassischen Softwareprojekten bestimmen Lastenheft und Pflichtenheft, was eine Software alles können muss. Die Qualität wird dabei meist erst bei der Endabnahme durch den Kunden oder danach geprüft. In agilen Projekten erfolgt die Qualitätsprüfung hingegen schon weitaus früher. Eine Möglichkeit, eine grobe Aussage über die Qualität von Software treffen zu können, ist die Erfassung von Metriken.

Die Durchführung von Qualitätssicherungsmaßnahmen in der Softwareentwicklung hängt nicht zuletzt vom bereitgestellten Budget ab und wird daher meist stark vernachlässigt. Allerdings sind bei der Softwareentwicklung die Kosten zum Beheben von Fehlern ungleich höher als die Kosten zur Implementierung neuer Features. Dies führt durch das Gewinnstreben von Managern dazu, dass letzteres den Vorzug genießt.

Diese Arbeit soll einen Leitfaden darstellen, welche Metriken in Softwareentwicklungsprojekten relevant sind und vor allem eine Aussage darüber treffen, welche Werte als Referenz dienen können. Wenn in den Projekten agile Methoden oder Referenzmodelle mit integrierter Qualitätssicherung, wie Capability Maturity Model Integration (CMMI), Projects in Controlled Environments (PRINCE2), Software Process Improvement and Capability Determination (SPICE), Six Sigma (6σ) oder Control Objectives for Information and Related Technology (COBIT) zur Anwendung kommen, wird eine Erhebung von Metriken unabdingbar.

Schon Tom DeMarco stellte in [DeM86] fest: „You can't control what you can't measure." Nach diesem Grundsatz gehen Forscher vor, um Metriken zu erarbeiten, die einen gewünschten nachzuvollziehenden Sachverhalt abbilden können. Die Suche nach einem universal einsetzbarem Maß ist dabei noch nicht zu einem brauchbarem Ergebnis gekommen. Oftmals wurden in der Vergangenheit Modelle ausgearbeitet, die für ihre Zeit eine hohe Aussagekraft hatten. Die Programmierparadigmen und Entwicklungszyklen haben sich aber weiterentwickelt. Daher sind die Aussagen, die heute getroffen werden sollen, nicht durch die alten Werkzeuge vollständig abgedeckt.

Eine Aussage über die Qualität eines Produktes kann ohne Hilfe von geeigneten Werkzeugen nur unqualifiziert und subjektiv erfolgen. Was ist jedoch eine subjektive Qualitätsaussage wert? Sie spiegelt maximal Einzelmeinungen wider und ist nicht vergleichbar mit anderen Aussagen und daher praktisch wertlos. Eine objektive Betrachtung und nötige Umsetzung von Verbesserungsmaßnahmen ist dringend geboten. Um eine Verbesserung bei der Softwarequalität darstellen zu können, referenziert diese Arbeit auf CMMI als Reifegradmodell. Im Speziellen soll CMMI-DEV zur Anwendung kommen. Welche Metriken im Rahmen einer Reifegradverbesserung Anwendung finden, kann mit Hilfe der GQM Methode festgelegt werden. Alternativ dazu ist eine Betrachtung nach SPICE denkbar.

Die Gründe, Softwarequalität messen zu wollen, sind vielfältig. Software ist so abstrakt, dass Metriken ein Verständnis der inneren Werte überhaupt erst möglich machen. Der wichtigste Nutzungsgrund ist die Herstellung einer Vergleichbarkeit zwischen verschiedenen Softwaresystemen oder verschiedenen Versionen eines Systems. Zahlen oder andere skalierte Werte sind auf den Managementebenen die einzig verwertbaren Fakten. Daher werden sie dort zur Planung, Steuerung und Überwachung eingesetzt.

Werden die richtigen Metriken über einen längeren Zeitraum erhoben, so kann auch eine belastbare Vorhersage über die zu erwartende Qualität erstellt werden. Die Statistik, die dabei entsteht, darf nicht nur aus Metriken über den eigentlichen Quelltext bestehen. Es sind auch viele andere Daten nötig, um die Voraussagen treffen zu können. Wobei hier ein hohes Maß an Qualität durch die Abwesenheit von Fehlern definiert werden soll.

Diese Arbeit wird, ausgehend von einer Bestandsaufnahme, eine Entwicklung von einem subjektiven Qualitätsgefühl hin zu einem objektiven Qualitätsmaß skizzieren. Grundlage ist dabei ein schon seit über zehn Jahren laufendes großes Softwareprojekt, dass in einem Team von 7-10 Leuten entsteht. Der Fokus liegt dabei in erster Linie bei den Entwicklern, deren Codequalität möglichst aufwandsarm nachvollzogen und, wenn nötig, erhöht werden soll. Ziel ist es dabei ein Vorgehen darzustellen, mit dem die zukünftige Entwicklung am Softwareprodukt effizienter und qualitätsgesichert gestaltet werden kann.

Die Betrachtung von einzelnen Entwicklern ergibt im Rahmen dieser Arbeit keinen Sinn, da es sich um eine kollaborative Entwicklung des Teams handelt. Der Fokus liegt auf dem Produkt und dem Entwicklungsprozess, der hier betrachtet werden soll. Jeder Entwickler hat mindestens einen Themenbereich, den er fachlich betreut. Die eigentliche Entwicklungsarbeit obliegt jedoch den Entwicklern unabhängig von ihren Fachkompetenzen. Jeder entwickelt an vielen Stellen im gesamten System mit und führt auch das Review neu entwickelter Funktionen anderer Entwickler, unabhängig von seiner fachlichen Verantwortlichkeit, durch.

1.2 Grundlagen

Unter einer Metrik wird im Allgemeinen, nach ihrer griechischen Bedeutung, eine Maßzahl verstanden. Ihr Wert ergibt sich aus einer jeweils definierten Berechnungsvorschrift. Diese Vorschrift einer Softwaremetrik spiegelt eine Eigenschaft der untersuchten Software wider. Es ist möglich, dass einzelne Metrikwerte die Ausgangswerte für die Berechnung anderer, höher abstrahierter, Metriken darstellen. Die Werte die sich aus einer Metrik ergeben, werden auf bestimmte Skalentypen abgebildet um eine Aussagekraft entfalten zu können. Horst Zuse definierte in seiner Dissertation [Zus85] eine Hierarchie für die anzuwendenden Skalentypen (siehe Tabelle 1.1). Die Abstufungen zeigen deutlich den Abstraktionsgrad und legen das mögliche Anwendungsgebiet der zu erwartenden Werte fest. Der IEEE Standard 1061 definiert eine Softwaremetrik wie folgt:

> Eine Softwaremetrik ist eine Funktion, die eine Software-Einheit in einen Zahlenwert abbildet. Dieser berechnete Wert ist interpretierbar als der Erfüllungsgrad einer Qualitätseigenschaft der Software-Einheit.

Skala	**Bedeutung**
Absolutskala	Die Werte sind eindeutig als Zahl darstellbar. Es existiert ein Nullpunkt und es existiert ein natürliches Maß. Diese Skala hat den geringsten Abstraktionsgrad.
Relationalskala	Die Werte sind genau so darstellbar wie in einer Absolutskala, es existiert ein Nullpunkt, aber das Maß ist willkürlich definiert.
Intervallskala	Hier kann der Abstand zwischen zwei Werten Grundlage einer Aussage sein, deren Quotienten sind es allerdings nicht.
Ordinalskala	Alle Werte dieser Skala können in eine Rangordnung gebracht werden.
Nominalskala	Die Werte dieser Skala können zwar unterschieden, aber nicht in eine Rangordnung gebracht werden. Diese Skala weist den höchsten Abstraktionsgrad auf.

Tabelle 1.1: Skalentypen

Die Metriken, die in dieser Arbeit untersucht werden, zielen auf Softwareprojekte ab, die in objektorientierten Programmiersprachen wie `Java` oder `C#` geschrieben wurden. Viele der älteren Metriken sind auch auf andere Sprachfamilien übertragbar, obwohl sie zu einer Zeit entstanden sind, als objektorientierte Sprachen noch nicht verbreitet waren und prozedurale Sprachen den Markt dominierten.

Softwaremetriken können im Rahmen einer statischen oder dynamischen Codeanalyse erhoben werden. Die statische Codeanalyse zieht als Datenquelle nur den vorliegenden Quelltext heran. Für Java gibt es Tools, die zwischen der statischen und der dynamischen Analyse arbeiten. Dabei wird nicht der Quelltext als solches betrachtet, sondern der bereits kompilierte Bytecode. Dieser wird zur Analyse aber nicht zur Ausführung gebracht. Bei der dynamischen Analyse werden die Parameterwerte für eine Metrik während Unit-Tests oder Integrationstest erhoben.

Nach [SSB10] sind Softwaremetriken in drei Dimensionen zu unterteilen. Die Grundlage bilden dabei die Größenmetriken, die in Kapitel 4.1 behandelt werden. Sie liefern einen direkten Bezug zum Umfang der zu messenden Software. Die zweite Dimension sind die Komplexitätsmetriken, die aus dem Quelltext in Abhängigkeit von den Größenmetriken errechnet werden. Sie werden in Kapitel 4.2 behandelt. Die dritte Dimension ist die Qualität selbst, die abstrakt aus einem Gerüst der beiden anderen Dimensionen hervorgeht und deren Herleitungen in Kapitel 2 behandelt werden.

Anstatt sich nur auf den entstandenen Quellcode zu beziehen, können auch in den Entwicklungsschritten vorher diverse Analysen vorgenommen werden. Die vom Karlsruher Institut für Technologie entwickelte Software Palladio analysiert die Architekturmodelle, aus denen sie in einer Simulation einen Prototypen erstellt und diese dann anschließend einer Bewertung unterzieht. Diese Form der Analyse setzt zumindest einen modellbasierten Ansatz im Softwareentwicklungsprozess voraus. Eine bessere Aussagekraft entfalten die Analysen bei einer durch und durch modellgetriebenen Entwicklung, bei der zum Beispiel das Generative Software Entwicklung Zwickau (GENESEZ)-Framework zur Anwendung kommt.

Was in diesem Kontext eine Softwareeinheit darstellt, ist je nach Sichtweise unterschiedlich. Eine mögliche Sicht sind die betrachteten Codezeilen als Grundlage. Wie eine Codezeile gezählt wird, muss allerdings ebenfalls definiert werden. Weitere mögliche Softwareeinheiten sind Function-Points, Klassen, Methoden oder Pakete. Im Folgenden werden diese Softwareeinheiten mindestens auf Methodenebene betrachtet und Artefakte genannt.

Die Erhebung von Metriken muss Gütekriterien genügen. Die grundlegenden Kriterien sind in Tabelle 1.2 dargestellt und von Wallmüller aus [Wal90] übernommen. Dabei ist darauf zu achten, dass alle Kriterien bei der Erhebung vollständig zu erfüllen sind, da die Erhebung sonst wertlos werden könnte.

Ein Detail, dass bei der Erhebung genaustens betrachtet werden sollte, ist der Widerspruch zwischen der Vergleichbarkeit und der Abbildung auf einer Nominalskala. Das Kriterium der Vergleichbarkeit ist schon dadurch erfüllt, dass die Skalenwerte entweder gleich oder ungleich sind.

Kriterium	Erklärung
Objektivität	Ein Metrik ist objektiv, wenn keine subjektiven Einflüsse des Messenden auf die Messung möglich ist.
Zuverlässigkeit	Bei der Wiederholung der Messung unter denselben Messbedingungen werden dieselben Messergebnisse erzielt. Die Messergebnisse sind also zeitlich unabhängig.
Validität	Die Messergebnisse erlauben einen eindeutigen und unmittelbaren Rückschluss auf die Ausprägung der Kenngröße. Die Metrik misst tatsächlich die zu messende Eigenschaft.
Normierung	Es muss eine Skala existieren, auf der die Messergebnisse eindeutig abgebildet und verglichen werden.
Vergleichbarkeit	Ein Metrik ist vergleichbar, wenn sie mit anderen Metriken in Relation setzbar ist.
Ökonomie	Die Messung muss mit geringen Kosten durchgeführt werden können. Die Ökonomie hängt vom Automatisierungsgrad, der Anzahl der Messgrößen und der Anzahl der Berechnungsschritte ab.
Nützlichkeit	Werden mit einer Messung praktische Bedürfnisse erfüllt, dann ist eine Metrik nützlich.

Tabelle 1.2: Gütekriterien für die Erhebung von Metriken

1.3 Vorbetrachtungen

Um in einem Projekt sinnvoll Metriken einsetzen zu können, müssen durch das Management Entscheidungen getroffen werden, welche Werte für welche Metriken akzeptabel sind. Sonst werden die erhobenen Daten zu einem großen unübersichtlichen Zahlenfriedhof. Das Management sollte auch die Entwickler in die Ausarbeitung der Kriterien mit einbeziehen. Dies hat zwar zur Folge, dass der Hawthorne-Effekt einsetzen kann, aber ihre Akzeptanz einer Messung wird dadurch steigen. Die Verhaltensänderung ist aber durchaus positiver Natur, da sie qualitätsbewusster entwickeln. Eine Zusammenarbeit über mehrere Arbeitsebenen hinweg kann auch zu einer besseren Kommunikationskultur innerhalb einer Abteilungshierarchie oder auch des ganzen Unternehmens führen.

Die Einführung und Nutzung von Metriken stellt einen kontinuierlichen Prozess dar, der in [ED07] von den Autoren als vierstufige Iteration, dem E4-Messprozess, dargestellt wird. Der Beginn ist eine Beurteilung des Ist-Zustandes, die sich wiederum unterteilt in die Analyse der Umgebung, die Charakterisierung des Projektportfolios und eine Identifizierung vorhandenen Knowhows. Die zweite Stufe ist die Orientierungsphase. Diese dient der Festlegung der Ziele, einer kleinteilig genauen Herausarbeitung von Aufgaben, der Identifikation relevanter Aktionen und einer Wahl zweckmäßiger Messmethoden und Werkzeuge.

Die dritte Phase ist die Messphase, in der der Messplan endgültig festgelegt und den beteiligten Personen die Notwendigkeit und die Ziele nahegebracht werden müssen. Des Weiteren geschieht hier die eigentliche Datenerhebung und -analyse. Die vierte Phase ist die Optimierungsphase, bei der aus den gewonnenen Daten Verbesserungen erarbeitet werden und eine Continuous Inspectation implementiert bzw. verbessert wird. Nicht nur die Continuous Inspectation erfährt hier eine Verbesserung, auch die internen Richtlinien können an diesem Punkt auf den neuen Wissensstand angepasst werden. An diesem Punkt der Iteration können hervorgebrachte Hypothesen gegebenenfalls validiert oder falsifiziert werden.

Dieser Zyklus soll im betrachteten Projekt auch durchlaufen werden, um die Qualität des entwickelten Produktes quantifizieren zu können. Ausgangspunkt dabei ist eine Bestandsaufnahme im Kapitel 3. Doch bevor auf den aktuellen Stand eingegangen werden kann, muss der Rahmen der Qualität im folgenden Kapitel 2 abgesteckt werden. Nachdem Abschluss der ersten Iteration des Messprozesses sollen die Erhebungen und Verbesserungen verstetigt werden, um dem CMMI-Reifegradmodell gerecht zu werden. Wie sich das Projekt im Rahmen dieser Arbeit verändert hat, wird in Kapitel 8 dargestellt.

2 Softwarequalität

2.1 Begriffsabgrenzung

Es existieren unzählige Definitionen für Softwarequalität, die sich zum großen Teil überschneiden. Helmut Balzert definiert in seinem Lehrbuch [Bal97], analog zur DIN ISO 9126, Softwarequalität wie folgt:

> „Unter Softwarequalität versteht man die Gesamtheit der Merkmale und Merkmalswerte eines Softwareprodukts, die sich auf dessen Eignung beziehen, festgelegte oder vorausgesetzte Erfordernisse zu erfüllen."

Diese Definition umfasst nur die Produktqualität einer Software, zu deren die Erfüllung nicht nur die funktionalen, sondern auch die nicht funktionalen Anforderungen gehören. Qualität im Allgemeinen wird in der ISO Norm 9000 so definiert:

> „Qualität ist die Übereinstimmung von Produkteigenschaften mit den Erwartungen des Kunden."

Nach dieser Definition sind also zusätzlich zu den oben genannten Anforderungen auch solche relevant, die nicht explizit erfasst sind. Dabei kommt es vor allem auch darauf an, welche Software die Benutzer beziehungsweise Kunden sonst gewohnt sind und welche Ansprüche sie an zum Beispiel Softwareergonomie stellen. Jeder Benutzertyp erwartet auch etwas anderes, abhängig von seinem Hintergrundwissen, welches sich auf technischer und

fachlicher Ebene manifestieren kann. Die Anforderungen des Projektmanagers, der die Software einführt, sind andere als die des Sacharbeiters, der die Software nutzen soll, oder der Geldgeber, der möglichst schnell ein positives Return on Investment (ROI) erwartet.

Die Qualität des Entstehungsprozesses von Software fällt nicht unter die Softwarequalität, sondern zum Qualitätsmanagement (QM). Diese Unterscheidung betrachtete bereits Tom Gilb in [Gil77] und kam dabei zum Ergebnis, dass beide Aspekte getrennt voneinander behandelt werden sollten.

Softwarequalitätsmetriken treffen nicht perse eine belastbare Aussage über Softwarequalität. Denn die Gesamtqualität ist die Gesamtsumme der Interpretationen aller möglichen Indikatoren. Um diese Interpretation durchführen zu können bedarf es eines möglichst vollständig und klar definiertem Qualitätsmodells, welches allumfassend mögliche Indikatoren in Betracht zieht.

Die ersten Qualitätsmodelle waren theoretischer Natur. So versuchte Donald Ervin Knuth in den 1970er Jahren die Qualität kleiner Programme auf mathematischen Wege festzustellen. Es folgten Modelle, deren Teile sich hauptsächlich auf Abhängigkeiten und Prioritäten beschränkten. Später folgten Modelle, die einige Indikatoren festlegten, die zu messen sind.

Jim McCall definierte in [MRW77b] zwei Kategorien, nach den Qualitätseigenschaften definiert werden können. Zum Einen die Eigenschaften, die direkt gemessen werden können. Das Auftreten von unerwünschtem Verhalten wie undefinierten Zuständen, nicht nutzbaren Eingabekomponenten oder richtigen Softwarefehlern, die das Programm zum Absturz bringen, sind solche messbaren Ereignisse. Die zweite Kategorie sind Indikatoren, die nicht direkt messbar sind. Beispiele dafür sind die Wartbarkeit und Verwendbarkeit des Softwareproduktes abhängig vom System. Dies ist nicht die einzige Dimension, nach der McCall die Kriterien aufteilte. Er ordnete die Indikatoren hierarchisch in einem Baum an, wobei die erste Ebene sich aufteilte in „Product Revision“, „Product Transition“ und „Product Operation“.

Auch Boehm kam in [Boe78] zu einem hierarchischen Modell, dass seiner Ansicht nach unendlich erweiterbar ist. Die Metriken, die in seinem Modell angewandt werden, beziehen sich jeweils auf einen Indikator in der untersten Ebene. Die Aussagen in den Ebenen darüber sind Aggregate einzelner Werte, die auch zu mehreren Indikatoren gehören können. So entsteht letzten Endes ein Baum, an dessen Wurzelknoten eine Gesamtaussage getroffen werden kann.

2.2 Indikatoren

Schon im Jahre 1977 veröffentlichte McCall in [MRW77b] eine Sammlung von Indikatoren, die eine Bestimmung von Softwarequalität ermöglichen. Die Abbildung 2.1 aus [SSB10] stellt die von ihm aufgestellten Abhängigkeiten zwischen den Qualitätsindikatoren dar.

Die Erkenntnis, welche sich daraus ableiten lässt, ist, dass eine sorgfältige Abwägung stattfinden muss. Indikatoren müssen nach ihrem Anteil an der Gesamtqualität gewichtet werden. Es ist durchaus möglich, dass einzelne Teile solch eine geringe Relevanz haben, dass sie in einer Erhebung zum Beispiel aus Kostengründen nicht betrachtet werden müssen.

Um eine belastbare Aussage zu einem System treffen zu können, müssen alle Indikatoren erhoben werden, da sonst die Vergleichbarkeit mit anderen Systemen auf der Strecke bleibt. Wenn allerdings aus Kostengründen einzelne Teilbereiche nicht betrachtet werden können, dann sollte dies bei keinem betrachteten Projekt erfolgen, da die Vergleichbarkeit sonst nicht gewährleistet ist.

Die Matrix in Abbildung 2.1 macht deutlich erkennbar, an welchen Stellen die Manager eines Softwareprojektes zwischen den einzelnen Merkmalen eine Rangordnung aufstellen müssen. Eine Erhöhung der Sicherheit eines Systems führt, nach McCall, unweigerlich zur einer geringeren Effizienz desselben.

Fast alle der dargestellten Indikatoren sind nicht direkt messbar. Eine Ausnahme dabei ist die Effizienz, die sich relativ einfach messen lässt, indem beispielsweise die benötigte Zeit

	Änderbarkeit	Allgemeingültigkeit	Effizienz	Funktionsabdeckung	Handhabbarkeit	Portabilität	Richtigkeit	Robustheit	Sicherheit	Testbarkeit	Verständlichkeit	Vollständigkeit
Änderbarkeit		+	-		-	+	+		-	+	+	
Allgemeingültigkeit	?			?								
Effizienz	-	-		?	-	-		-	?	-	?	
Funktionsabdeckung			?						?			
Handhabbarkeit	?	+	-						?	+		
Portabilität			-						-	?		
Richtigkeit	+	?			+	?		?	+			
Robustheit	?	?	-		+	?	+		+	+		
Sicherheit	-	-	-		-	-	+	+		-		
Testbarkeit	+				+	+	?		-			
Verständlichkeit			?		+	+				+		
Vollständigkeit					+	+	+	+	+			

Wirkung von Qualitätsmerkmalen auf andere Qualitätsmerkmale:

+ : Positive Wirkung
- : Negative Wirkung
? : Nicht eindeutige Wirkung
() : Keine/Geringe Wirkung

Abbildung 2.1: McCalls Qualitätsmatrix

für die Abarbeitung einer Aufgabe erfasst wird. Diese wird verglichen mit einer erwarteten Dauer oder einer vertretbaren Höchstdauer. Hat man diese Daten erhoben und dazu noch eine zuverlässige vorherige Schätzung des Aufwandes betrieben, können diese beiden Werte gegeneinander aufgerechnet werden. Eine Basis, wie viele Aufgaben durch das Team im Normalfall abgearbeitet werden können, ist als dritter Wert nötig. Dabei sind auch automatisierte Benchmarks eine Möglichkeit, wenn der Fokus auf einer günstigen Datenerhebung statt auf einer billigen Entwicklung liegt.

Für die Indikatoren müssen Maße und Sollwerte aufgestellt werden, die die Qualitätskriterien widerspiegeln. Nachdem eine Gewichtung der Maße und Kriterien erfolgt ist, kann eine Aussage über die Gesamtqualität eines Systems getroffen werden.

2.3 Anwendungsmöglichkeiten

In der vergangenen Zeit spielte eine hohe Softwarequalität in vielen Projekten eine eher untergeordnete Rolle. Viele Softwareprodukte wurden unfertig auf den Markt gebracht und sollten beim Kunden nachreifen. So kam es, wegen mangelnder Qualitätssicherung, zu teilweise verheerenden Katastrophen. Ein Beispiel dafür sind die Fehlfunktionen des Therac-25 Linearbeschleunigers zur Strahlentherapie bei Krebspatienten. Die Software dieses Gerätes wurde von einem einzigen Programmierer unter Verwendung vorhandener Codeteile geschrieben. Weder die alten Teile von anderen Programmierern, noch sein eigener Code waren gut dokumentiert. Er selbst war für die Qualitätssicherungsmaßnahmen zuständig. Die Maßnahmen waren derart unzureichend, dass mehrere später dokumentierte Softwarefehler zu mindestens drei Todesopfern führten. Erst nach mehreren Zwischenfällen wurde mit der Lösungssuche begonnen und eine nachträgliche Qualitätssicherung durchgeführt. Bis dahin war man davon ausgegangen, dass es lediglich Hardwarefehler geben könne. Softwarefehler, die erst nach der Auslieferung bekannt werden, waren damals noch nicht aufgetreten.

Ein weiteres Beispiel für die Wichtigkeit von beständiger und wiederkehrender Qualitätssicherung, die besonders bei der Wiederverwendung vorhandenen Codes nötig ist, war die Explosion der Ariane 5 Trägerrakete am 4. Juni 1996. Für ihr Navigationssystem war die Software aus der Ariane 4 übernommen worden. Es kam zu einem Überlauf einer Variable, der in der alten Rakete physikalisch nicht auftreten konnte. Das neue Triebwerk war um einiges leistungsstärker, was letztendlich zur Wirkung des Fehlers führte und eine vollständige Zerstörung der Rakete zur Folge hatte.

Einhergehend mit mangelnder Qualität sind meist auch wirtschaftliche Nachteile für den Hersteller eines Produktes. Der Kunde wird kein Produkt kaufen, bei dem er nicht sicher sein kann, dass seine Anforderungen erfüllt werden. Und wenn doch, so wird er versuchen, eine nachträgliche Erfüllung seiner Anforderungen einzufordern beziehungsweise durchzusetzen.

Das Ziel eines Herstellers sollte also eine möglichst hohe Qualität seiner Produkte sein, damit er eine Marktmacht aufbauen kann und sich selbst mehr um Innovationen in seinen

Produkten, statt um die Entfernung vorhandener Fehler kümmern kann. Für die Managementebene eines Unternehmens ist es wichtig zu wissen, was ihre Entwicklungsabteilung leistet. Daher hat sie ein Interesse daran, dass der Output ihrer Programmierer und Systemarchitekten objektiv messbar ist. Ohne eine konsequente Nutzung eines Qualitätsmodells kann über die Performance der Teammitglieder nur gemutmaßt werden. Eine gute Qualitätssicherung sorgt für geringere Wartungskosten, wenn sie als Prozess fest im Unternehmen verankert ist.

Viele Entwickler erhalten heutzutage ein leistungsabhängiges Gehalt, daher müssen messbare Kriterien herangezogen werden, um eine geeignete Höhe des Leistungssentgeltes zu bestimmen. Eine Vergütung nach Vertriebserfolg während der Bearbeitung von Supportfällen ist wenig motivierend für einen Entwickler, der selbst eine kreative Arbeitsweise an den Tag legt und sich eigentlich nicht mit diesem Arbeitsablauf befassen will. Viel zielführender ist die Abhängigkeit von realisierten Funktionen, die den Qualitätsansprüchen genügen. Dabei kann der Entwickler sich besser seiner eigentlichen Aufgabe widmen und kreative und elegante Lösungen für die Probleme der Kunden erschaffen.

3 Bestandsaufnahme

Die Entwicklung der Software im betrachtetem Projekt verläuft nicht nach klassischen Modellen. Es gibt regelmäßige Releases, die circa im Halbjahresrhythmus fertiggestellt werden. Es kommen teilweise agile Methoden zum Einsatz, so werden zum Beispiel die Meilensteine eines Releases in SCRUM-Sprintlänge von 4 Wochen erstellt. Es existiert weder ein definierter SCRUM-Master, noch ein Product Owner. Die Anforderungen, die im SCRUM das Product Backlog füllen, werden in einem Meeting zwischen Entwicklungsleitung, Vertrieb, Marketing und Geschäftsleitung festgelegt. In diesen Meetings werden nur die Ziele für die Releases festgelegt. Die Planung, was in den einzelnen Meilensteinen realisiert werden soll, erfolgt durch das ganze Entwicklungsteam gemeinsam auf Arbeitsebene. Die daily SCRUM Meetings finden nicht statt.

Die Größe und Zusammensetzung des Entwicklungsteams sind im betrachteten Projekt relativ stabil. Entgegen dem Trend zur Generalisierung von Entwicklungsaufgaben zeichnen sich einige Entwickler für bestimmte fachliche Bereiche verantwortlich. Die historisch gewachsene Verantwortung der Entwickler für ihren “eigenen Code“ wird von den anderen Teammitgliedern respektiert. Aus dem Besitz erwächst gleichzeitig auch Verantwortung für den Code, die die Entwickler dann gerne übernehmen. Das führt dazu, dass der Aufwand bei Code Reviews oder auch späteren Anpassungen nach Ausscheiden eines Entwicklers exponentiell steigt, wenn der Code nicht gut dokumentiert ist oder der Programmierer sich nicht an die Standards des Codestils hält.

Eine Wandlung hin zum durchgängig agilen Vorgehen verbietet das zuvor dargestellte Heldentum. Neuen Entwicklern fällt der Einstieg leichter. Urheber alten Codes verstehen ihren

eigenen Code auch nach längerer Zeit noch gut. Jedes Teammitglied ist dann flexibler einsetzbar, muss sich aber darauf einstellen, dass er auch abseits seiner präferierten Themen zum Einsatz kommt.

Entwicklungsanforderungen werden auf verschiedenen Wegen aufgenommen. Ein Weg sind die Vertriebsbeauftragten, die von Kunden und Interessenten Anforderungen aufnehmen. Bei der Betreuung von Kundenprojekten oder bei der Bearbeitung von Supportfällen zeigt sich regelmäßig Bedarf, der in Anforderungen mündet. Die Rahmenbedingungen von externen Gutachtern und Gremien tragen ebenfalls ihren Bedarf zu den Anforderungen bei. Festgelegte Prozessverantwortliche aus dem Entwicklungsteam beobachten die Rahmenbedingungen ihrer Prozesse und erfassen dazu ebenfalls Anforderungen. Diese vielfältigen Wege führen zu sehr unterschiedlicher Qualität in der Anforderungsdokumentation. Zum Teil sind Anforderungen sehr kleinteilig, gut strukturiert und vollständig. Der Großteil ist sehr schwammig formuliert und bedarf einer aufwandsreichen Nachbearbeitung, damit die einzelnen Aufgaben für die Entwickler abgeleitet werden können.

Ein Problem, das hieraus folgt, ist eine große Unschärfe bei Konzepten, die diese Anforderungen umsetzen sollen. Das Team, dass die Konzepte zu Anforderungen erfasst, ist gleichzeitig auch das Entwicklungsteam. Teilweise sind die Konzepte so gestrickt, dass sie zwar von allen anderen Entwicklern verstanden werden können, aber einige der geforderten Funktionen nicht explizit vorkommen. Der Grund dafür ist entweder nur eine kleine Nachlässigkeit des Konzeptschreibers oder die Absicht, die Umsetzung auch selbst vorzunehmen. Dieser fehlende Input führt später unweigerlich zu Missverständnissen. Es existiert kein Mitglied, dass sich dezidiert mit dem Anforderungsmanagement beschäftigt und auch dementsprechend geschult ist. Ein Vorteil ist allerdings die Erfahrung, die Teile des Teams als Projektleiter bereits jahrelang in Kundenprojekten sammeln konnten. Dieser Erfahrungsschatz führt dazu, dass Bedarf zuverlässig erkannt wird. Die Anforderungen, die daraus entstehen werden, sind in Ermangelung ausreichender Zeit jedoch nur unpräzise dokumentiert. Der Arbeitsablauf mit dem Anforderungen in den Entwicklungsplan eingebracht werden, ist in Abbildung 3.1 schematisch dargestellt.

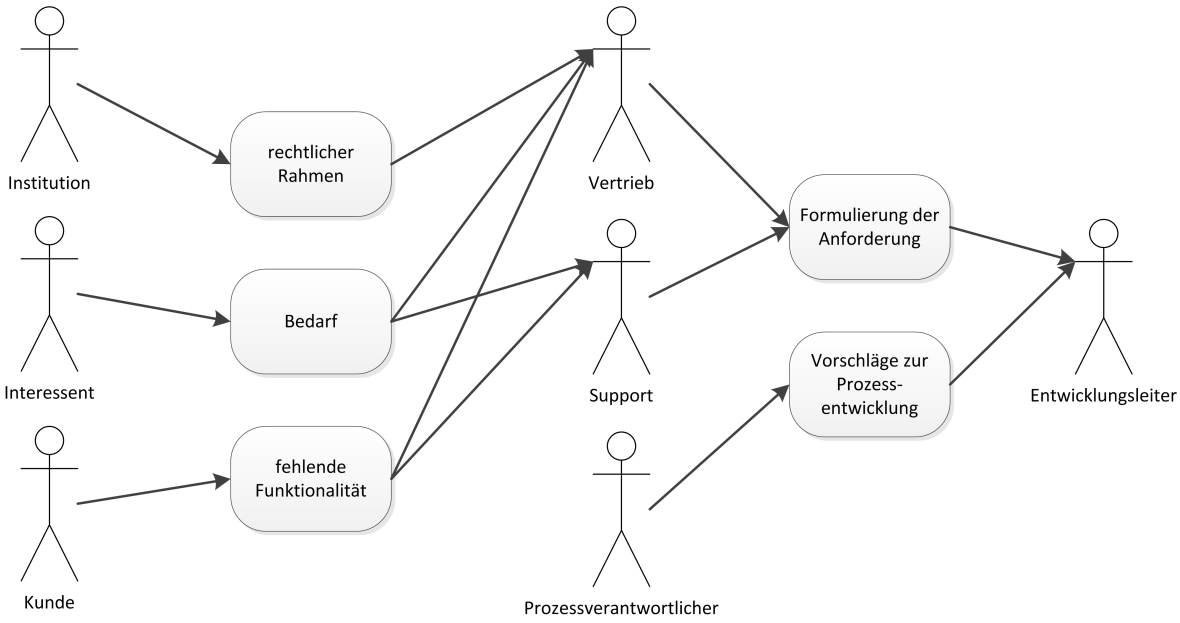


Abbildung 3.1: Arbeitsablauf Featurewünsche

Für die Umsetzung einzelner Anforderungen existiert ein wohldefinierter Arbeitsablauf. Nachdem ein Ressortverantwortlicher das Konzept zu einer Anforderung erfasst hat, schätzt er, basierend auf seiner Erfahrung, den zu erwartenden Aufwand für die einzelnen umzusetzenden Arbeitspakete. Das Entwicklerteam arbeitet selbstständig die anliegenden Aufgaben ab und meldet erledigte Aufgaben. Diese werden nach dem Vier-Augen-Prinzip von einem anderen Entwickler begutachtet und dann entweder zurückgewiesen, wenn die Anforderung nicht vollständig und richtig erfüllt ist oder abgenommen, wenn es keine Beanstandungen mehr gibt.

Zusätzlich zu den Anforderungen kommen über den Support oder bei Integrationstests durch Entwickler noch Programmfehler zum Vorschein. Diese werden als Fehlertickets gewichtet nach Priorität in das Projektmanagementtool eingetragen und dann bearbeitet. Auch bei der Aufnahme von Fehlertickets wird der zu erwartende Aufwand geschätzt.

Gemäß den Releaseplanungen und den geplanten Zeitanteilen, die Entwickler zur Verfügung haben, werden die Arbeitsaufgaben in die einzelnen Meilensteine einsortiert. Die Zeitplanung

berücksichtigt dabei fast ausschließlich die Zeit, die nötig ist, die geforderte Funktionalität zu analysieren und sie zu implementieren. Ein Test während der Entwicklung gehört zwar zur Schätzung dazu. Der nötige Aufwand für eine Abnahme der realisierten Funktion wird momentan nicht berücksichtigt. Eine Erfassung der letztendlich genutzten Zeit für die Arbeitsschritte erfolgt nur sporadisch.

Um den Entwicklern eine zügige Rückmeldung zu geben, werden deren Commits im Subversion (SVN) Repositorium zügig von einem Build-Server im Rahmen einer Continuous Integration kompiliert und als Laufzeitsystem zur Verfügung gestellt. Zusätzlich dazu werden beim nächtlichen Buildlauf die Werkzeuge Checkstyle, Findbugs und ein Copy-Paste-Detector (CPD) ausgeführt. Die Meldungen daraus werden zwar den Entwicklern zugänglich gemacht, sie stellen aber nur einen Appell an die Entwickler dar, keine Warnungen zu erzeugen. Bei einigen Teammitgliedern wecken die vorhandenen Meldungen den Ehrgeiz, diese nach und nach abzubauen.

Betrachtet man die Entwicklung der letzten 2 Jahre in diesem Projekt, zeigt sich eine signifikante Steigerung der Professionalität in der Entwicklung. Einige Arbeitsabläufe wurden in dieser Zeit erstmals klar definiert und werden nun auch strikt so umgesetzt. Das Konzept, für getrennte fachliche Bereiche Verantwortlichkeiten festzulegen, führte zu einer klareren Struktur und besseren Planung von Meilensteinen. Es wurde zudem ein Workflow definiert, mit dem Consultants und Vertriebsbeauftragte ihre Anforderungen beim Entwicklungsteam platzieren können.

Die größte Änderung stellte der Umstieg von einem Cronjob auf ein Continuous Integration System dar. Damit verbunden war eine bessere grafische Aufarbeitung der bereits erfolgten Analysen mit Checkstyle, um den Entwicklern besseren Einblick in die Konformität ihres eigenen Codes mit den vorhandenen Richtlinien zu ermöglichen. Zusätzlich zu diesen Analysen am Quelltext kommt seit dem Findbugs zum Einsatz, um den Bytecode zu prüfen. Auf diese Weise konnten einige Fehler identifiziert und aus dem Programm entfernt werden.

Deutlich stärker wird in letzter Zeit auf eine möglichst vollständige Dokumentation wichtiger Arbeitsschritte geachtet, da in der vergangenen Zeit auf Grund mangelnder Nachvollziehbarkeit Missverständnisse auf mehreren Ebenen entstanden sind. Ein Ausfall eines Teammitglieds, das noch nicht ausreichend dokumentiert hat, konnte dazu führen, dass Kundenprojekte ins Stocken gerieten. Das Ziel einer vollständigen Dokumentation kann dazu führen, dass zukünftig weniger Zeit für Abstimmung bzw. Übergabe zwischen einzelnen Personen aufgewandt werden muss.

Der Support am Kunden ist bis vor kurzem noch ausschließlich von den Entwicklern betreut worden. Daraus resultiert eine gute Kommunikationskultur zwischen Anwendern und Entwicklern. Die Anwender können neben den Problemen, die direkt in der Entwicklung aufgenommen werden, auch Wünsche und Anforderungen formulieren. Allerdings führen diese Arbeitsunterbrechungen auch zu einem regelmäßigen Zeitverlust, der bei der Wiederaufnahme von Entwicklungstätigkeiten anfällt, um sich erneut in das eben bearbeitete Problem einzudenken. Diese Unterbrechungen können zu einer schlechteren Qualität der produzierten Software führen. Um dem wirksam zu entgegnen, ist ein Teammitglied eingestellt worden, dass sich nun ausschließlich mit dem Support befasst. Die positiven Auswirkungen auf den Arbeitsfluss der Entwickler sind in kurzer Zeit schon spürbar gewesen.

Es wird das Ziel verfolgt, dass zukünftig weniger Entwicklungszeit durch mehrfache Arbeit verschwendet wird. Das vorhandene Programmframework wird nach und nach erweitert, um im Zuge kleinteiliger Refactoringmaßnahmen den Anteil an dupliziertem Code zu verringern. Trotzdem muss eine optimale Dokumentation der Arbeit und der Software erfolgen. Das optimale Verhältnis zwischen diesen Anforderungen herauszufinden und festzuschreiben ist eine der Aufgaben für die zukünftige Teamarbeit.

Um die Vorteile einer Continuous Inspectation Lösung zu evaluieren, wurde zu Beginn der Bearbeitung dieser Arbeit ein Sonar Server aufgesetzt, der unter Mitwirkung des Continuous Integration Systems periodisch Metriken erhebt. Auf den Nutzen und die Auswirkungen dieses Tools wird im Kapitel 5.5 im Detail eingegangen.

Zusammenfassend lässt sich also feststellen, dass die Etablierung eines gemessenen Qualitätsmaßstabes im vorliegendem Projekt noch in den Kinderschuhen steckt. Erste Ansätze sind bereits erkennbar und der eingeschlagene Weg zeigt in die richtige Richtung. Mit Hilfe dieser Arbeit soll eine für die anderen Abteilungen und andere Projekte adaptierbares Modell entstehen, um zukünftig die Qualität des Produktes und der Ergebnisse der Arbeit der Entwickler messen und bewerten zu können. Ein weiteres Ziel ist die Feststellung, welche Ressourcen noch nötig sind, um die gewünschten Werte mit der nötigen Sorgfalt zu erreichen.

4 Metriken

4.1 Größenmetriken

Abbildung 4.1: Typische Größenmetriken

Größenmetriken bilden die Grundlage vieler anderer Metriken. Sie sind selbst sehr leicht zu erheben, haben dabei allerdings sehr wenig Aussagekraft. Die älteste aufgestellte Metrik ist die Anzahl der Codezeilen (Lines of Code (LoC)). Dazu zählen alle Zeilen in denen Quelltext, Leerzeichen oder Kommentare stehen. Da diese Metrik nur einen begrenzten Schluss auf den Umfang eines Programms zulässt, hat sich mittlerweile die Erhebung von Non Commented Source Statements (NCSS) durchgesetzt, bei der die Anzahl der vorhandenen Anweisungen gezählt werden. Für die Sprache Java zählt man dazu einfach alle schließenden runden Klammern und alle Semikolons. Weitere Größenmetriken stellen Number of Packages (NOP), die Zahl der Klassen, der Methoden, der Accessoren, der Statements, der Dateien und Lines of Comment dar. Letztere sind aber ebenso schwer zu zählen wie die LoC, da es in vielen Programmiersprachen mehrere Kommentarkonstrukte geben kann und auskommentierte Codezeilen nicht zu den Kommentaren zählen sollten.

Da mittlerweile in vielen Projekten ein großer Anteil des Codes von Werkzeugen generiert werden kann, sollte dieser anderweitig in die Größenzählung einfließen. Es muss in allen Fällen eine konkrete Definition geben, was alles zum Umfang gezählt wird. Ein Ansatz, um eine korrekte Größe eines Softwaresystems zu beziffern, sind redundanzfreie Source

Lines of Code (RSLOC). Um diese zu erheben, sollte allerdings bekannt sein, welche Teile generiert sind und welche Teile von Entwicklern dupliziert wurden. Zwischen diesen Arten muss dann extra differenziert werden. Nach einer Analyse des Quelltextes mit einem CPD kann dann der Umfang bestimmt werden. Dieses Verfahren ist vergleichsweise umständlich anzuwenden. Eine Verwendung von Annotationen, wie `@Generated` unter Java, erleichtert die Unterscheidung für die Analysetools erheblich.

Um eine Vergleichbarkeit von Programmumfängen auch über Programmiersprachbarrieren hinweg herzustellen und auch im Vorfeld einer Entwicklung den Umfang schon bestimmen zu können, werden heutzutage teilweise die von IBM in den 1970er-Jahren entwickelten Function Points (FP) benutzt. Allerdings existieren auch für diese mittlerweile eine Reihe von Definitionen. Meist werden aus statistischen Werten Funktionen abgeleitet, die eine Umrechnung von LoC nach FP darstellen. In der Praxis finden sie kaum Verbreitung, selbst wenn in einem Unternehmen verschiedensprachige Projekte entwickelt werden.

4.2 Komplexitätsmetriken

Die Maße für Komplexität können in mehreren Dimensionen dargestellt werden. Komplexität kann nicht nur für Software gemessen werden. Einige Wissenschaftler haben sich auch an einer Messung von Texten versucht. Dabei wurde von Rudolf Flesch 1948 die Berechnungsvorschrift für die Flesch Reading Ease (FRE) aufgestellt, die die Lesbarkeit eines Textes beschreiben soll. Der Ausgangspunkt dieser Messung ist die durchschnittliche Satzlänge und die durchschnittliche Anzahl Silben pro Wort. Einer der Ersten, der sich mit der Komplexität von Software auseinandersetzte, war Maurice Howard Halstead. Er stellte, vermutlich inspiriert von der Bewertung natürlicher Texte, eine Reihe von Formeln auf, die einige Aspekte darstellen. Ausgehend von der Zahl der verschiedenen Operanden n_1 und der Zahl der verschiedenen Operationen n_2, die zusammen das Vokabular eines Programms bilden und der Anzahl ihrer Vorkommnisse N_1 und N_2, deren Summe die Länge beschreibt,

stellte er für die Schwierigkeit eines Programms die Formel 4.2.1 auf. Die rechnerische Größe wird mit 4.2.2 beschrieben.

$$D := \frac{n_1}{2} \times \frac{N_2}{n_2} \tag{4.2.1}$$

$$V := (N_1 + N_2) \times \log_2(n_1 + n_2) \tag{4.2.2}$$

Untersuchungen haben ergeben, dass dieses Komplexitätsmaß die Realität nur teilweise widerspiegelt. Dieses Maß berücksichtigt nur die lexikalische Komplexität einer Software, sodass sie nur eine Aussage über die Lesbarkeit der betrachteten Komponente zulässt. Man kann diese Metrik sowohl sinnvoll für einzelne Methoden, als auch für ganze Pakete anwenden. Eine Komplexität (D) von unter 10 für eine Methode gilt als trivial, während ein Wert von mehr als 30 als zu komplex gelten kann. Das Volumen (V) einer Methode sollte höchstens 1500 betragen. Durch die negativen Untersuchungsergebnisse konnten Halsteads Metriken bisher keine große Praxisrelevanz erreichen und genießen bisher einen eher akademischen Charakter.

Beinahe zur gleichen Zeit veröffentlichte Thomas McCabe in [McC76] seine Herangehensweise zur Komplexitätsberechnung. Seinen Ansatz entlehnte er der Graphentheorie. Das Programm wird dabei als Baum mit seinen Verzweigungen zur Grundlage der Berechnung gemacht. Diese Methode lässt im Gegensatz zu Halstead die lexikalischen Aspekte vollkommen außer acht. Die Formel 4.2.3 ist die Berechnungsvorschrift für die McCabe Metrik oder auch Cyclomatic Complexity (CC). Dabei ist e die Anzahl der Kanten im Kontrollflussgraph, n die Anzahl der Knoten und p die Anzahl der beteiligten Komponenten. Es ergeben sich Werte $>= 1$. Erstrebenswert sind Ergebnisse die im Bereich < 10 liegen, da der Inhalt einer sonst Methode nur schwer von Entwicklern nachzuvollziehen ist. Ziel dieser Metrik war eine Beschreibung der Wartbarkeit und Testbarkeit einer Softwarekomponente.

$$CC := e - n + 2p \tag{4.2.3}$$

Beide Metriken entstanden zu einer Zeit, in der es noch keine objektorientierten Programmierkonzepte gab. Daher leidet ihre Aussagekraft, wenn man sie in der heutigen Zeit auf objektorientierte Systeme anwendet. Die Betrachtung des Kontrollflussgraphen einer Methode sieht Methodenaufrufe als simple Anweisung. Dies führte unter anderem dazu, dass ihre Bedeutung abgenommen hat. Die zyklomatische Komplexität wird nach wie vor verwendet, um auf Methodenebene einen Basiswert darzustellen, der dann in höheren Ebenen aufgegriffen werden kann. Relevant für die Komplexität einer Klasse, eines Paketes oder eines ganzen Systems ist jeweils der Durchschnitt der Komplexität aller enthaltenen Funktionen.

Weder Halstead noch McCabe konnten eine Korrelation zwischen ihren Metriken und der Wartbarkeit beziehungsweise der Testbarkeit empirisch nachweisen. In [OEMD08] befassen sich die Autoren mit der Korrelation zwischen Komplexität und Fehlerrate. Sie kommen bei der Betrachtung eines agil entwickelten Softwareprojektes zu einer starken Korrelation zwischen der errechneten Komplexität und der zu erwartenden Fehlerrate der Artefakte.

4.3 Architekturmetriken

Die Metriken von McCabe und Halstead eignen sich nicht gut für die Nutzung in objektorientierten Programmiersprachen. Aus diesem Grund stellten Shyam R. Chidamber und Chris F. Kemerer in [CK94] ihre Testsuite für Softwareprojekte nach diesem Paradigma vor. Sie beinhaltet die Metriken Weighed Methods per Class (WMC), Depth of Inheritance Tree (DIT), Number of Children of a Class (NOC), Coupling Between Objects (CBO),Response For a Class (RFC) und Lack of Cohesion of Methods (LCOM). Diese Metriken fokussieren sich auf die Anwendung objektorientierter Konzepte. Dabei hängt die Architektur direkt mit der Komplexität zusammen. Der Paradigmenwechsel weg von prozeduralen Sprachen führte dazu, dass den Entwicklern nicht mehr sofort ersichtlich war, welcher Code unter welchen Bedingungen ausgeführt wird, wenn Überladung und Vererbung zur Anwendung kommen. Es kommt eine absolute Skalierung zum Einsatz bei der kleinen Werten eine hohe Qualität zugeordnet wird.

Die in der Chidamber und Kemerer (CK)-Metrik Suite enthaltenen Metriken reichen nicht aus, um ein vollständiges Bild der Architektur eines Systems zu zeichnen. Daher haben Mark Lorenz und Jeff Kidd in [LK94] eine Reihe weiterer Metriken vorgestellt. Die bekannten darunter sind Class Size (CS), Number of Operation Overridden (NOO), Number of Operation Added (NOA) und Specialization Index (SI). Sie erfassen größtenteils den Grad der Wiederverwendung des Codes durch Vererbung.

Um die vorhandenen Architekturmetriken zu vervollständigen, stellten Brito e Abreu und Melo weitere auf, um statt der absoluten Skalierung von Lorenz und Kidd eine relative zum Einsatz zu bringen. Damit verbesserte sich die Vergleichbarkeit zwischen Verschiedenen Softwareprojekten. Ihre Suite umfasst Attribute Hiding Factor (AHF), Attribute Inheritance Factor (AIF), Method Hiding Factor (MHF), Method Inheritance Factor (MIF), Coupling Factor (COF) und Polymorphie Faktor (POF). Die Suite wurde unter dem Namen Metrics for Object-oriented Design (MOOD) in [AM96] vorgestellt.

Robert C. Martin definierte in [Mar94] und [Mar02] ebenfalls eine Reihe von Metriken, die in die Kategorie Architektur einzuordnen sind. Er bezog sich dabei jeweils auf die Paketebene und die enthaltenen Klassen beziehungsweise Interfaces. Aus dem Verhältnis der Kopplungen zu anderen Klassen (Afferent Couplings (eingehende Klassenkopplungen) (Ca) und Efferent Couplings (ausgehende Klassenkopplungen) (Ce)) ergibt sich die sogenannte Instability (I) mit einem Wertebereich zwischen 0 (nur Ca) und 1 (nur Ce). Zusammen mit dem Anteil der abstrakten Klassen oder Interfaces, der Abstractness (A), kann eine Metrik über die Veränderbarkeit von Paketen errechnet werden. Die Abbildung 4.2 zeigt ein typisches Abstractness Instability Diagramm. Die Pakete, die

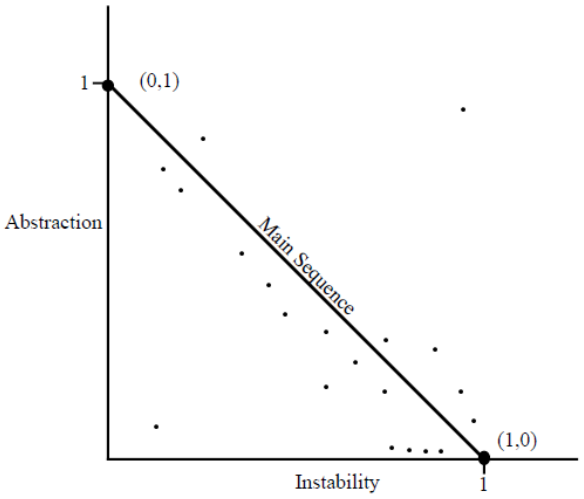


Abbildung 4.2: Abstractness Instability Diagramm

eine geringe Abstraktion und eine kleine Instability aufweisen, sind auf Grund der zu erwartenden Auswirkungen sehr schwer zu ändern. Im umgekehrten Fall der hohen Abstraktion und Instability kann davon ausgegangen werden, dass die Pakete effektiv keine Funktionen enthalten und daher nutzlos sind. Als optimal kann ein Wert angesehen werden, dessen Summe der Werte für A und I 1 ergeben. Sie liegen auf der in der Abbildung dargestellten Hauptlinie. Die Entfernung von der Hauptlinie heißt Martin Distanz (D). Der Mehrwert, den diese Metrik bietet, erschließt sich nur in einer „von Hand“ entwickelten Software, die auf wenig generiertem Code basiert und objektorientierte Konzepte gut umsetzt. Andernfalls zeigen viele Pakete sehr ähnliche Werte.

Bei der Frage, welche Metriken relevant sind, um eine Aussage über die Qualität einer Software treffen zu können, kommt es darauf an, ob Toleranzschwellen existieren, die eine Aussage zur Qualität abbilden. In [SLSN10] untersuchten die Autoren, ob sich für diese Metrikklasse Warnschwellen definieren lassen. Eine Unterscheidung zwischen fehlerhaften und fehlerfreien Klassen war nicht zu erreichen. Eine Unterscheidung zwischen kleinen, mittleren und schweren Fehler gelang dagegen. Für diese Einteilung konnten für CBO, RFC, WMC und NOO jeweils Warnschwellen errechnet werden.

4.4 SQALE Metriken

Die Metriken, die im Rahmen der Nutzung des SQALE-Modells zum Einsatz kommen, beziehen sich in der höchsten Granularität auf die Klassenebene bzw. auf jeweils ein Sourcefile. Das Modell selbst wird in Kapitel 5.3 beschrieben. In der Summe eignen sich die Metriken zur Ermittlung der technischen Schuld, die im untersuchten System vorhanden ist.

Die technische Schuld stellt eine Metapher dar, mit welchem Aufwand die untersuchte mangelhafte Software von enthaltenen Qualitätsproblemen bereinigt werden kann. Sie wird entweder in Entwicklungszeit oder in Entwicklungskosten angegeben. Errechnet werden kann sie mittels einer Gewichtung je nach Schweregrad multipliziert mit der jeweiligen Anzahl.

Aufgestellt wurde diese Metapher von Ward Cunningham, um Managern die Abwägung zwischen Qualität und Entwicklungszeit zu verdeutlichen.

In der Definition des Modells in [Let12] werden die in Tabelle 4.2 und den folgenden Berechnungsformeln dargestellten Metriken definiert. Die dazugehörigen Dichte Metriken, bei denen statt einer absoluten eine relative Skalierung zum Einsatz kommt, ergeben sich aus der Relation zwischen der Artefaktgröße, beispielsweise gemessen in LoC, und der entsprechenden Index-Metrik. Um den Lebenszyklus darstellen zu können, bauen die einzelnen Metriken aufeinander in den sogenannten konsolidierten Indizes auf.

Index	**Dichte**
SQALE Testability Index (STI)	SQALE Testability Index Density (STID)
SQALE Reliability Index (SRI)	SQALE Reliability Index Density (SRID)
SQALE Changeability Index (SCI)	SQALE Changeability Index Density (SCID)
SQALE Efficiency Index (SEI)	SQALE Efficiency Index Density (SEID)
SQALE Security Index (SSI)	SQALE Security Index Density (SSID)
SQALE Maintainability Index (SMI)	SQALE Maintainability Index Density (SMID)
SQALE Portability Index (SPI)	SQALE Portability Index Density (SPID)
SQALE Reusability Index (SRuI)	SQALE Reusability Index Density (SRuID)
SQALE Business Impact Index (SBII)	SQALE Business Impact Index Density (SBIID)

Tabelle 4.2: Überblick der SQALE Metriken

$$SCRI = STI + SRI \tag{4.4.1}$$

$$SCCI = STI + SRI + SCI \tag{4.4.2}$$

$$SCEI = STI + SRI + SCI + SEI \tag{4.4.3}$$

$$SCSI = STI + SRI + SCI + SEI + SSI \tag{4.4.4}$$

$$SCMI = STI + SRI + SCI + SEI + SSI + SMI \tag{4.4.5}$$

$$SCPI = STI + SRI + SCI + SEI + SSI + SMI + SPI \tag{4.4.6}$$

$$SCRuI = STI + SRI + SCI + SEI + SSI + SMI + SPI + SRuI \tag{4.4.7}$$

Die Indikatoren, die mit Hilfe der hier dargestellten Metriken abgeleitet werden, werden in Kapitel 5.3 näher dargestellt.

4.5 Weitere Metriken

Die bisher vorgestellten Metriken beziehen sich ausschließlich auf den Code beziehungsweise auf den Entwurf der Software. Doch bevor ein solcher Entwurf überhaupt existiert, werden Anforderungen erfasst, die den benötigten Funktionsumfang einer Software umreißen sollen. Auch die Anforderungen können herangezogen werden und für Metrikberechnungen in Betracht kommen.

Die Bereinigung von Qualitätsproblemen in frühen Entwicklungsphasen sorgt für die geringsten Kosten. Um möglichst schnell ein positives ROI zu erreichen, sollten Qualitätsmessung und -sicherung nicht erst

Der zweite Grund eine Erhebung für Anforderungen durchzuführen, ist die höhere Prozessqualität, die erreicht wird, wenn überhaupt ein Monitoring mit anschließender Auswertung stattfindet. Christof Ebert brach in [Ebe12] die Zahl der wichtigen Anforderungsmetriken auf sieben herunter. So zeigte er, wie mit wenigen Metriken die Kontrolle über das Anforderungsmanagement erlangt werden kann. Es handelt sich dabei um folgende:

- Zahl der Anforderungen im Projekt
- Fertigstellungsgrad der Anforderungen
- Änderungsrate pro Entwicklungsphase
- Anzahl der Ursachen für Änderungen an Anforderungen
- Grad der Vollständigkeit des Anforderungsmodells

- Anzahl der Mängel in den Anforderungen
- Nutzwert der einzelnen Anforderungen

Zu beachten ist dabei, dass sich der Nutzwert einzelner Anforderungen aus dem festgelegten Sollwert und dem Grad der Fertigstellung unabhängig von den entstandenen Kosten ergibt.

Die Sophist Gruppe als Spezialist im Anforderungsmanagement hat einen Regelsatz herausgegeben, um die Qualität von Anforderungen beschreiben zu können. Dazu wurde von ihnen eine Metrik zur Anforderungsqualität aufgestellt die sechs Kriterien umfasst. Die sich ergebenden Metrikwerte der Kriterien treffen eine Aussage, inwiefern das Sophist Regelwerk eingehalten wurde.

Die Anforderungsmetriken stehen vor dem Entwicklungsprozess. Es gibt auch nach der Entwicklung noch Prozesse, deren Eigenschaften mit Metriken abgebildet werden können. Nachdem ein Softwareprodukt beim Kunden in den Produktivbetrieb geht, beginnt die Wartungsphase. Sowohl die Produktivität während der Entwicklung, als auch der Wartungsaufwand nach der Entwicklung können gemessen werden.

Sneed stellt in [SSB10, Kapitel 9] heraus, dass die Produktivitätsmessung einzelner Entwickler in einigen Teilen der Welt als unsozial gilt. Daher wird die Produktivitätsbetrachtung in einem Entwicklungsprojekt meist auf der Teamebene durchgeführt, trotz des verbreiteten Verständnisses darüber, dass die Produktivität einzelner Mitglieder eines Teams stark variieren kann. Die Messung und von Produktivität kann am besten dann erfolgen, wenn es sich bei dem Projekt nicht um eine Neuentwicklung handelt. Denn dann sind die Einflussfaktoren auf die Produktivität besser zu messen und eine Planung genauer. Bewährt hat sich für die Planung des Aufwandes das Constructive Cost Modell Version 2 (COCOMOII). Ein Vergleich zwischen geplantem und tatsächlich benötigtem Aufwand liefert die Aussage über die erreichte Produktivität.

Die Metriken die zum Thema Wartung erhoben werden können, beziehen sich auch auf die Produktivität. Die meisten Kosten, die im Rahmen einer Software Total Cost of Ownership

(TCO) Betrachtung zu Buche schlagen, fallen erst nach Inbetriebnahme einer Softwarelösung im Rahmen der Wartung an. Daher sollte die Betrachtung der Wartungsproduktivität über der der Entwicklungsproduktivität stehen.

Unter der Maßgabe, dass im betrachteten Projekt das Entwicklungsteam die Wartung und die Weiterentwicklungen vornimmt, gewinnt die Wartbarkeit der Software sehr an Relevanz. Ohne eine Änderung der Teamgröße bewirkt eine schlechte Wartbarkeit bei schlechter Gesamtqualität eine Lähmung der Innovationsstärke. Eine langsamere Innovationskraft kann sich nachteilig auf die Marktposition eines Produktes auswirken.

Beim Versuch, die Wartbarkeit und die Wartungsproduktivität zu beschreiben, war bisher noch keinem Forscher erfolgreich. Die Wartbarkeit wird meist in Abhängigkeit mit den verschiedenen Komplexitätsarten und Größenmetriken gebracht. Einige der Codemetriken aus der CK-Suite und die MOOD-Metriken weisen eine starke Korrelation mit der nach subjektiven Gesichtspunkten erhobenen Wartbarkeit von Software auf.

Um die Wartungsproduktivität bestimmen zu können, ist eine Quantifizierung des Aufwandes erforderlich. Dem Aufwand können die Zahl der Fehlermeldungen, die Zahl der Änderungswünsche und vor allem die benötigten Zeiten zur Behebung eines Fehlers oder zur Umsetzung eines Änderungswunsches zugerechnet werden. Wenn dann diese Größen ins Verhältnis mit der Personalstärke des Wartungspersonals gesetzt wird, ergibt sich daraus die Wartungsproduktivität. Zu beachten ist, dass jeder Entwickler im betrachteten Softwareprojekt auch mit der Wartung der Software betraut ist. Eine von Sneed vorgeschlagene Berechnungsformel für die Wartungsproduktivität über den Zeitraum eines Jahres lautet:

$$Wartungsproduktivität = \frac{Codemenge * Änderungsrate}{Mitarbeiter} \tag{4.5.1}$$

5 Nutzungsstrategien

5.1 Capability Maturity Model Integration

Prozesse in der Softwareentwicklung waren in der früheren Zeit noch nicht durchdefiniert und folgten daher keinem geordnetem System. Als sich dies im Laufe der Zeit änderte und die Softwareentwicklung deutlich professionalisierte, kam der Wunsch auf, dass diese professionellen Prozesse bewertet werden sollten. Das Software Engineering Institute an der Carnegie Mellon University widmete sich diesem Thema in den 1990er Jahren.

Heraus kam das Capability Maturity Model (CMM), dass ein Reifegradmodell für Softwareprozesse darstellt. Aus dem Modell entstanden im Laufe der Zeit viele Implementierungen für unterschiedliche Einsatzzwecke. Zwei bekannte Vertreter davon sind SPICE und COBIT. Nachdem viele dieser Implementierungen aufkamen, wurde CMM im Jahre 2000 zu CMMI erweitert. Zu dem eigentlichen Modell gehören Empfehlungen, die sich in der Praxis bewährt haben. Ähnlich der IT Infrastructure Libary (ITIL) sind in diesem Modell nur Rezepte vorhanden, die darstellen, was zu tun ist und nicht wie die Implementierungen dieser Rezepte im Einzelnen aussehen sollen. Ziel ist die Institutionalisierung der dargestellten Prinzipien.

Das Modell existiert in drei Varianten, die für verschiedene Themengebiete gedacht sind. Die erste Variante war Capability Maturity Model Integration for Development (CMMI-DEV), dass den eigentlichen Entwicklungsprozess abbildet. Später kamen Capability Maturity Model Integration for Acquisition (CMMI-ACQ) und Capability Maturity Model Integration for Services (CMMI-SVC) hinzu. Alle wurden zuletzt auf Version 1.3 im November 2010

aktualisiert. Alle Modelle enthalten 16 Kernprozessgebiete, die in den einzelnen Modellen erweitert werden und zusätzlich auch in mehreren Modellen vorkommen können.

Die Prozessgebiete gliedern sich in benötigte, erwartete und informative Komponenten auf. Die Gliederung in Abbildung 5.1 aus [CMM10] zeigt ihren Aufbau. Im CMMI-DEV-Modell gibt es insgesamt 22 Prozessgebiete, welche eine unterschiedliche Wertigkeit beim Erreichen der Ebenen aufweisen.

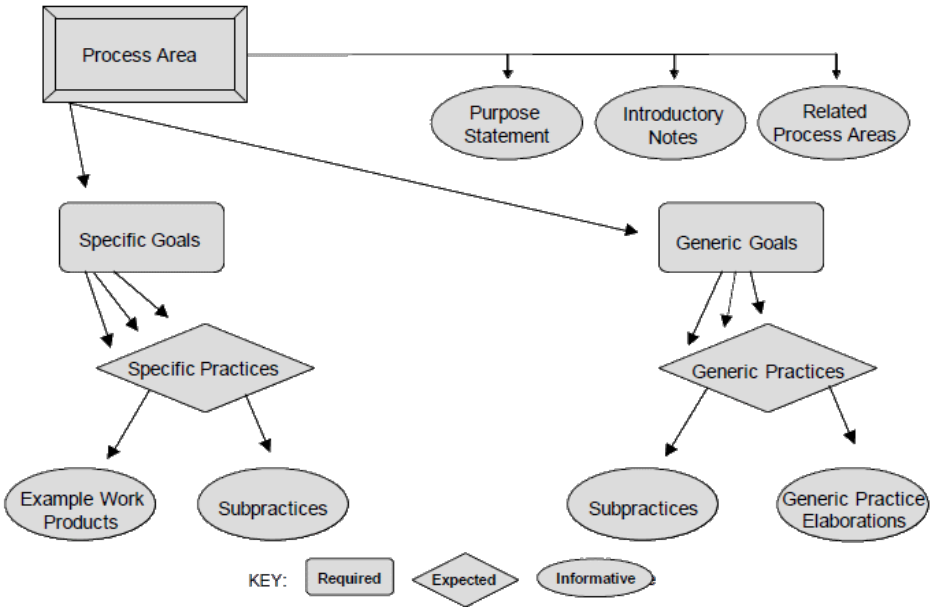


Abbildung 5.1: Die Komponenten eines CMMI-Prozessgebietes

Das Modell definiert in zwei Dimensionen verschiedene Ebenen. Der Grundgedanke, der dahinter liegt, ist die Evolution, die Prozesse und Organisationen während ihres Lebenszyklus durchlaufen. Die Anwendung von CMMI liegt in der kontinuierlichen Optimierung von Prozessen. Daher definiert das Modell Ebenen für die Umsetzung des Modells. Die Ebenen, die den Stand der einzelnen Prozessgebiete repräsentieren, werden „Maturity Levels“ genannt. Die Ebenen, welche die Fortschritt einzelner Ziele charakterisieren, werden „Capability Levels“ genannt. In Tabelle 5.1 sind die Stufen dargestellt.

Level	Capability Level	Maturity Level
0	Incomplete	
1	Performed	Initial
2	Managed	Managed
3	Defined	Defined
4		Quantitatively Managed
5		Optimizing

Tabelle 5.1: CMMI Levels

Die Capability Levels werden durch Anwendung der vorgeschlagenen generischen Lösungen durchlaufen. Wenn ein vorgeschlagener Prozess durchgeführt wird, gilt Capability Level 1 als erreicht. Wenn eine Anweisung des Managements, zum Beispiel in Form einer Unternehmens Policy, existiert, ist bereits Level 2 umgesetzt. Level 3 liegt dann vor, wenn es einen Unternehmensstandardprozess gibt, der für einzelne Projekte nur noch angepasst werden muss.

Um die Maturity Levels 1 bis 3 zu erreichen, müssen in einer im Modell definierten Menge an Prozessgebieten die korrespondierenden Capability Levels erreicht werden. Die Grundlage bildet dabei zuerst die Bewusstmachung, dass bestimmte Prozesse in der Organisation notwendig sind. Im Allgemeinen kann man davon ausgehen, dass Organisationen, die nur Maturity Level 1 erreichen, sehr flexible aber chaotische Prozesse nutzen. Ihr Erfolg hängt von einzelnen Stakeholdern ab und ist nicht ohne weiteres wiederholbar. Erst die Level 2 bis 3 sorgen für ein wiederholbares Ergebnis und ein Verständnis, wie Prozesse umgesetzt und Teammitglieder integriert werden können.

Das angepeilte Ziel in der Abteilung, dessen Entwicklungsprozess betrachtet wird, ist das Maturity Level 2 „Managed“. Ausgehend von den Erwartungen aus [CMM10] an diese Stufe erreichen die beteiligten Prozessgebiete zu Beginn der Betrachtungen die in Tabelle 5.2 dargestellten Capability Levels. Strategisches Ziel wird das Erreichen von Capability Level 2 in allen nötigen Prozessgebieten sein.

Prozessgebiet	Capability Level
Configuration Management (CM)	1
Measurement and Analysis (MA)	1
Project Monitoring and Control (PMC)	1
Project Planning (PP)	2
Process and Product Quality Assurance (PPQA)	1
Requirements Management (REQM)	0
Supplier Agreement Management (SAM)	0

Tabelle 5.2: Erreichte Capability Levels vor den bisherigen Veränderungen

Softwaremetriken sind eine mögliche Art, das definierte Prozessgebiet „MA“ aus dem Modell umzusetzen (siehe [CMM10, Seiten 176-190]). Das Gebiet sieht 2 spezifische Ziele vor. Ein Ziel ist die Anpassung von Messungen. Das zweite Ziel ist die Bereitstellung der Ergebnisse, die die Analysen hervorgebracht haben.

Eine konkrete Art und Weise, die gewünschten Ziele zu erreichen, ist die GQM, die im folgenden Kapitel genauer betrachtet wird. Eine weitere Methode um die Qualität einer Software zu ermitteln, ist Software Quality Assessment based on Lifecycle Expectations (SQALE), dass in Kapitel 5.3 kurz dargestellt wird.

5.2 Goal Question Metric

Das bloße Erfassen von Daten erfüllt noch nicht das Kriterium der Nützlichkeit, wenn sie dann abgelegt und nicht weiterverarbeitet werden. Aus diesem Grund gibt es dieses Vorgehensmodell, das die Erfassung zu einem wirklich nützlichen Werkzeug macht. Die Abbildung 5.2, die aus [AKP02] entnommen ist, zeigt den schematischen Aufbau des Vorgehens bei der GQM-Methode. Am Anfang steht dabei eine Zieldefinition im Rahmen eines Projektes, eines

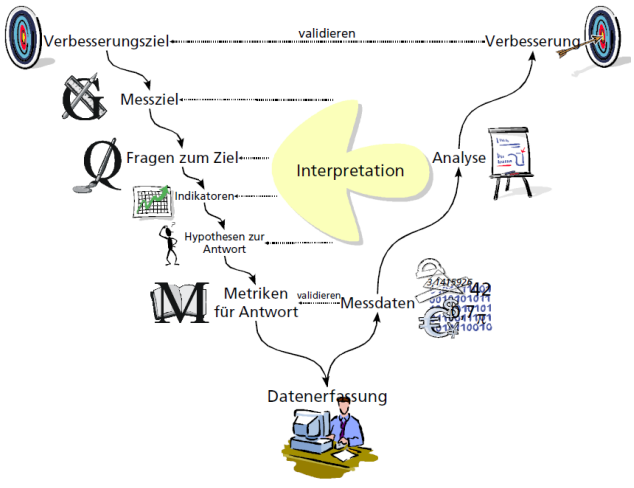


Abbildung 5.2: Schematischer Aufbau des GQM-Vorgehens

Prozesses oder ein Geschäftsziel des Unternehmens. Daraus wird in der Planungsphase ein Messziel definiert und alle relevanten Fragen zu dem Ziel festgehalten. Diese Messziele definieren sich nach [SB99] wie folgt:

- Welches Objekt ist zu messen?
- Zu welchem Zweck wird gemessen?
- Im Bezug auf welchen Qualitätshorizont wird gemessen?
- Aus wessen Sicht wird gemessen?
- In welchem Kontext wird gemessen?

Nachdem das Messziel definiert ist, werden die Fragen aufgestellt, die durch die Messung beantwortet werden sollen. Im dritten Schritt werden Metriken definiert, die der Fragestellung gerecht werden. Des Weiteren werden Hypothesen aufgestellt, welche Werte für die Metriken zu erwarten sind. Danach folgt die eigentliche Datenerhebung, deren Ergebnisse die gewählten Metriken validieren sollen. Der vorletzte Schritt ist die Analyse und Interpretation der erhobenen Daten. Aus der Interpretation werden dann letztlich Schlussfolgerungen gezogen, die das im ersten Schritt gewünschte Ziel realisieren.

Das Ergebnis einer GQM-Analyse kann als Baum verstanden werden. An seiner Wurzel steht das Messziel, darunter wird eine Frage formuliert, die das Ziel ganzheitlich abdeckt. In der Ebene darunter sind Fragen angeordnet, die einzelne Aspekte des Gesamtkontextes festlegen. Sollten die Aspekte nicht direkt mit bestimmten Metriken darstellbar sein, ist es möglich, zu einem Aspekt auch Teilaspekte in Form von weiteren Fragen zu definieren. Die Blätter des Baumes bilden die ausgewählten passenden Metriken. Der resultierende Baum ist in Abbildung 5.3 schematisch dargestellt.

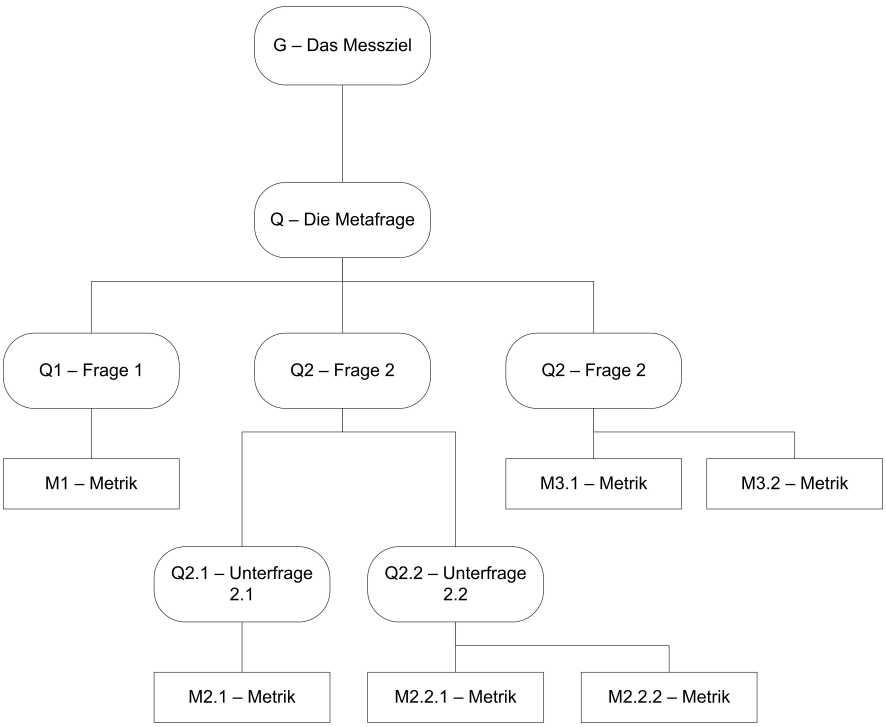


Abbildung 5.3: Darstellung des GQM-Baumes eines Ziels

Mit Hilfe dieser Methode kann für ein zu untersuchendes Projekt ein komplettes Qualitätsmodell erschaffen werden, dazu sollten mehrere der dargestellten Diagramme möglichst domänenspezifisch zusammengestellt werden. Im Kapitel 7 wird ein generisches Modell für das betrachtete Projekt dargestellt.

Ähnlich diesem Modell gibt es auch noch die Factors Criteria Metric (FCM)-Methodologie, deren Qualitätsbegriff sich auf fast die gleiche Weise definiert. Die Wurzeln dieser Methode gehen zeitlich allerdings viel weiter zurück als bei GQM. Definiert wurde sie 1977 von Jim McCall in [MRW77a].

5.3 SQALE

Das Software Quality Assessment based on Lifecycle Expectations (SQALE) Modell zielt, wie der Name erwarten lässt, auf den Lebenszyklus einer Software ab. Entwickelt wurde es von DNV IT GS France, die es im Januar 2012 in Version 1.0 unter [Let12] veröffentlichte. Auf Grund dieser Aktualität gibt es noch keine wissenschaftlich fundierten Erkenntnisse zur Validität und Nützlichkeit dieses Vorgehens. Die Methode basiert auf vier Konzepten: Dem Qualitätsmodell, dem Analysemodell, den Indizes und den Indikatoren.

Das Qualitätsmodell besteht aus drei Ebenen. Die oberste Ebene besteht aus acht Charakteristika, die an den Lebenszyklus einer Quelltextdatei angelehnt sind. Zusammen mit der Wiederverwendbarkeit spiegeln diese Charakteristika die Fähigkeiten aus dem Qualitätsmodell der ISO Norm 9126 wider [ISO91]. Die Charakteristika untergliedern sich wiederum in Subcharakteristika, die sich in Qualitätsanforderungen untergliedern. Ziel der Gliederung ist die Möglichkeit eines „drill-down“ durch die erhobenen Daten. Die Erhebung der Rohdaten erfolgt auf der niedrigst möglichen Ebene, der Klassenebene.

Die Charakteristika gliedern sich wie folgt:

- Wiederverwendbarkeit
- Portabilität
- Wartbarkeit
- Sicherheit
- Effizienz
- Änderbarkeit
- Zuverlässigkeit
- Testbarkeit

Im Mittelpunkt des Qualitätsmodells steht die Metapher der technischen Schuld, die mit Hilfe dieses Modells erhoben wird. Sie gibt an, wie teuer eine Weiterentwicklung zu einem fehlerfreien, von allen Design- und Implementierungsschwächen befreiten, System beziffert werden kann. Dabei gehen auch die Wartungskosten in die Betrachtung ein.

Das Modell sieht eine Anpassung an die Gegebenheiten im untersuchten Projekt vor. Einige Aspekte des Modells sind aber nicht veränderbar, dazu zählen die Berechnung der einzelnen Metrikwerte, die zur Wahrung der Konsistenz über die einzelnen Ebenen immer addiert werden. Der Grund dafür liegt darin, dass die einzelnen Indizes allesamt Kosten darstellen sollen. Die Indizes wurden im Einzelnen in Kapitel 4.4 bereits dargestellt.

Das SQALE-Modell definiert drei Indikatoren zur Bewertung der untersuchten Softwareartefakte. Um eine einfache Interpretation zu ermöglichen, besteht der erste Indikator, das sogenannte SQALE Rating, aus fünf Stufen. Die Einordnung in eine Stufe erfolgt auf der Basis der Relation zwischen technischer Schuld und Entwicklungskosten. Ein Beispiel für die Einteilung der Stufen ist in der Tabelle 5.3 dargestellt.

Rating	**techn. Schuld**
A	<1%
B	<2%
C	<4%
D	<8%
E	>=8%

Tabelle 5.3: Beispiel eines SQALE Rating

Der zweite Indikator, den das Modell beinhaltet, ist ein Kivat Diagramm, dessen einzelne Sektoren die Charakteristiken kennzeichnen. Zusätzlich zur Darstellung der erreichten Werte schlägt das Modell eine Markierung der gewünschten Werte vor. Pflicht sind die Darstellung aller genutzten Charakeristika in der Reihenfolge, wie sie im Modell genutzt werden. Ein Beispiel für die Anwendung des Kivat Diagramms ist Abbildung 5.4.

Der dritte Indikator den das Modell offeriert, ist die SQALE Pyramide. In ihr sind alle erhobenen Indizes ersichtlich. Sie erlaubt eine Analyse aus zwei Betrachtungsebenen. Durch die Summierung der einzelnen Indizes können zu allen Lebenszyklusstadien des aktuellen Projektes mögliche Schritte abgeleitet werden.

5.4 Toolbasierende Erhebung

Alle Metriken, die im späteren Verlauf dargestellt werden, können mit sehr großem Aufwand gemäß ihrer Berechnungsvorschriften manuell errechnet werden. Eine manuelle Erhebung

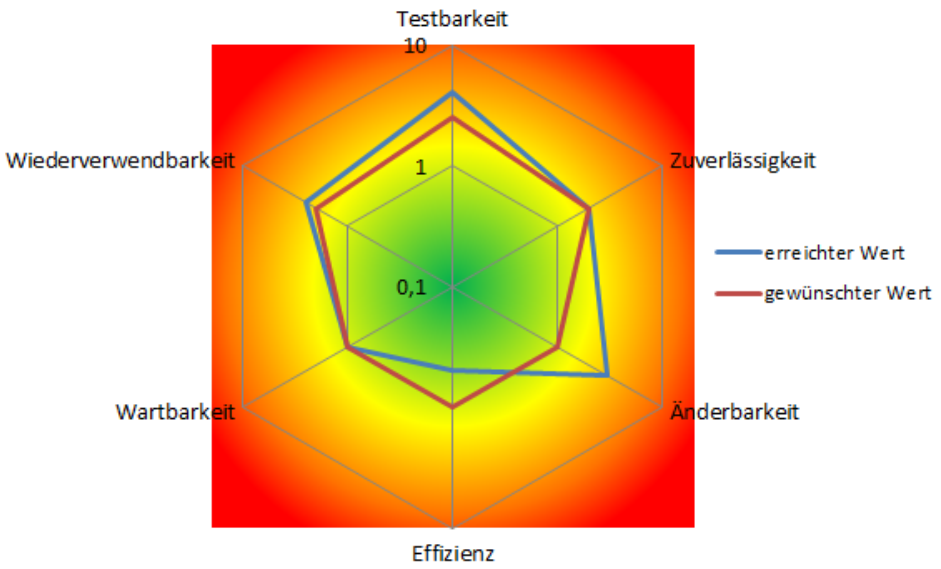


Abbildung 5.4: Kivat einer SQALE Analyse

ist daher höchstens für eine einmalige Analyse geeignet, wenn das zu untersuchende System nur sehr klein ist und aus wenigen Dateien besteht. Der Nutzen der manuellen Erhebung rechtfertigt aber in diesem Fall nicht den nötigen Aufwand und kann daher höchstens akademischen Charakter besitzen.

Für viele der gängigen Metriken existieren Tools, die den Erhebungsaufwand minimieren sollen. Eines dieser Werkzeuge ist zum Beispiel Iplasma, dass im Rahmen eines Forschungsprojektes entstand und Code in den Sprachen Java und C++ analysieren kann. In [MMMW05] geben die Entwickler einen kurzen Überblick, wie die Software zu benutzen ist. In [LM06] wird dann die Benutzung sehr ausführlich an größeren Beispielen dargestellt. Andere Werkzeuge sind zum Beispiel Source Monitor oder Understand. Diese Toolfamilie analysiert den Quelltext statisch und ermittelt daraus viele der Metriken aus Kapitel 4. Das Repertoire der Tools ist weitgehend identisch. Sie unterscheiden sich aber hinsichtlich ihrer Lizenzierung und Darstellung der Ergebnisse deutlich. Die Darstellungsmethoden reichen von schlecht lesbaren Konsolenausgaben, über gut strukturierte Berichte mit Iplasma bis hin zu abstrakten Grafiken mit CodeCity.

Weitere Werkzeuge integrieren sich nahtlos in das Integrated Development Environment (IDE) der Entwickler. Beispiele hierfür sind die Eclipse Test and Performance Tools Platform (TPTP), die einige Funktionen rund um Testing und Profiling vereinen. Ein anderes Plugin für Eclipse ist X-Ray, dass sich ganz einer Visualisierung des Softwareprojektes gewidmet hat. Die einzelnen Werkzeuge werden im Kapitel 6 behandelt.

Die toolbasierende Erhebung von Metriken birgt einen Nachteil, wenn es sich bei den Programmen nicht um Open Source Software handelt. Die Art und Weise, wie die Metriken im einzelnen Erhoben werden, ist nicht transparent. Es muss darauf vertraut werden, dass die Implementierung der Metriken der versprochenen Vorschrift entspricht. Nachgeprüft werden kann dies nur durch eine mögliche Einsichtnahme in den Quelltext.

5.5 Continuous Integration & Inspectation

Häufig sind Prozesse in der Softwareentwicklung bereits teilweise automatisiert, um die Produktivität zu erhöhen. Ein Schritt, der mittlerweile toolunterstützt genutzt wird, ist die Bereitstellung einer lauffähigen Version durch eine Continuous Integration Lösung. Der Buildprozess, der hier nach jedem Check-in eines Entwicklers angestoßen wird, kann nach entsprechender Anpassung dazu dienen, Metriken zu erheben.

Obwohl jeder Entwickler vor seinem Check-in prüfen muss, ob seine Änderungen auch mit dem gesamten Laufzeitsystem kompatibel sind, kann es vorkommen, dass auf Grund vieler Referenzen zwischen einzelnen Modulen das Gesamtsystem nicht mehr kompiliert und bereitgestellt werden kann. Ein Continuous Integration System gibt sehr schnell eine Rückmeldung an den Entwickler wenn dieser Zustand eintritt. Instabile Builds sind beispielsweise ein Indikator für mangelhafte Qualität, da dies die Testqualität der Entwickler widerspiegelt.

Da mit der Erhebung von Metriken einiger Aufwand verbunden ist, lohnt es nicht diese mit jedem Build zu errechnen. Viel zielführender ist eine Erhebung im Rahmen des nightly Builds.

Sollen langfristige Trends bei akzeptabler Datenmenge heraus gearbeitet werden, kann eine Continuous Inspectation auch seltener, beispielsweise wöchentlich, Daten erheben.

Wenn die Continuous Inspectation dazu genutzt wird, um die Qualität des Produktes zu bewerten und Verbesserungen zu eruieren, dann kann dies als Grundlage dienen, den Entwicklungsprozess umzustellen. Eine Evolution mit dem Ziel weg von seltenen Releases hin zu Continuous Delivery (CD) wäre dann möglich. Eine Open-Source Lösung, die Funktionen für eine Continuous Inspectation bereitstellt, wird in Kapitel 6.2.6 vorgestellt.

6 Werkzeuge

6.1 Standalone Werkzeuge

6.1.1 JDepend

Das Programm JDepend von der Clarkware Consulting, Inc. steht unter proprietärer Open Source Lizenz, die eine Anpassung und kommerzielle Nutzung erlaubt. Es betrachtet das Projekt auf Paketebene und erhebt dabei die Anzahl der Klassen und Interfaces, Ca, Ce, A , I und errechnet daraus dann die Martin-Distanz und Abhängigkeitszyklen.

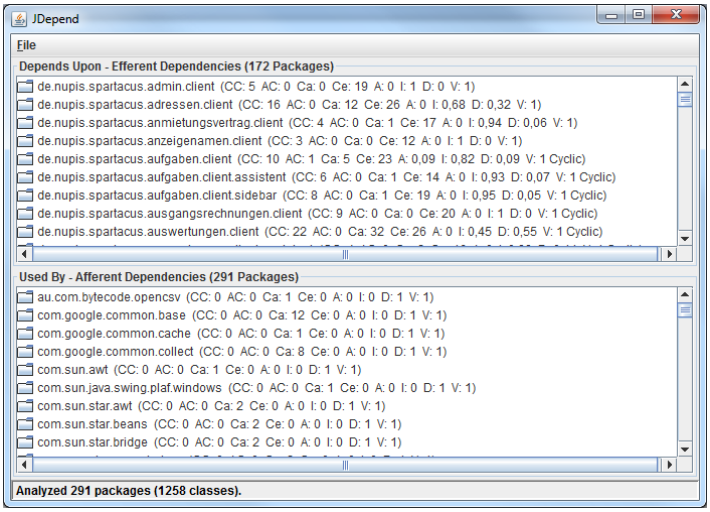


Abbildung 6.1: Screenshot von JDepend

Der Fokus liegt hierbei auf der Gruppe der Architekturmetriken, welche die Designqualität widerspiegeln. Es existieren mehrere User Interface (UI)-Implementierungen, die sich auch in die IDE Eclipse integrieren lassen und zur Entwicklungszeit einen Hinweis liefern. Eine Tendenz, wie sich die Werte im Einzelnen entwickeln, ist mit diesem Tool nicht automatisch zu erheben, sondern bedarf der manuellen Auswertung.

6.1.2 iPlasma und inFusion

Das Java Programm iPlasma analysiert Java und C++ Projekte statisch. Es ging aus einem Projekt an der LOOSE Research Group an der Polytechnischen Universität Timisoara hervor. Mit diesem Tool erhebt man auf Projekt- beziehungsweise Paketebene einige Größenmetriken wie LoC, Number of Methods of a Class (NOM), NOC und NOP. Dazu kommt die zyklomatische Komplexität und die Architekturmetriken Heigth of the Inheritance Tree (HIT), Number of direct Decendents (NDD), Fan Out (ausgehende Methodenkopplungen) (FOut) und die Methodenaufrufe in Methoden. Es liefert einen Satz von Hinweisen mit, in welchen Grenzen die Werte liegen sollten. Die Ergebnisse werden als Pyramide dargestellt und es wird zusätzlich dazu eine kurze textuelle Zusammenfassung der Werte dargestellt.

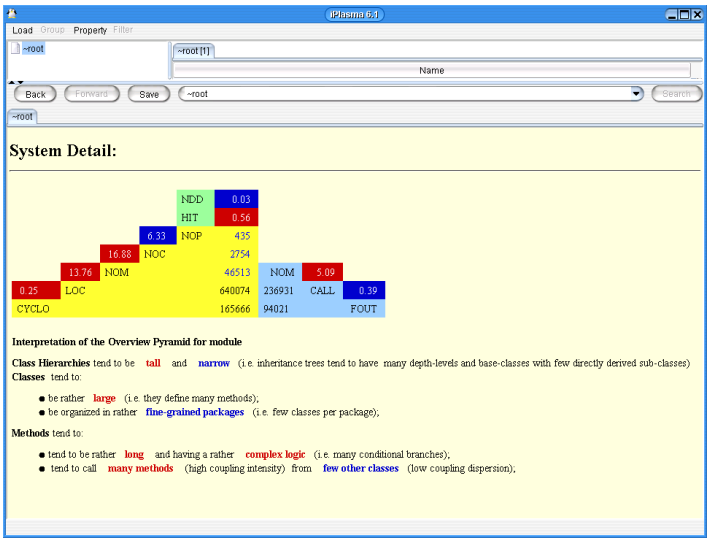


Abbildung 6.2: Screenshot von iPlasma

Auch dieses Programm liefert nur eine Momentaufnahme des aktuellen Quelltextes. Eine weitergehende Tendenzverfolgung kann nur manuell erfolgen und gestaltet sich mit diesem Programm schwieriger als bei JDepend, da die Ausgabe der Ergebnisse nur über die gezeigte UI erfolgen kann.

Aus diesem Programm entstand später inFusion, dass einige Funktionen mehr bietet und kommerziell vertrieben wird. Es erlaubt einen Drill-Down von der Projekt-Ebene bis hin zu einzelnen Methoden. Dabei werden bekannte Architektur-Anti-Pattern, wie z.B. Gott-Klassen, aufgespürt und dargestellt. Dieser Hinweis wird angereichert mit einer Empfehlung zum Refactoring.

6.1.3 CodeCity

Codecity ist eine gute Möglichkeit, um Zusammenhänge zwischen Größe und dem Auftreten von Anti-Pattern darzustellen. Dazu kann aus iPlasma das Programmmodell exportiert und mit dieser Software eingelesen werden. Das Hauptaugenmerk liegt dabei darauf, die Evolution eines Softwareprojektes darzustellen und besonders anfällige Fragmente hervorzuheben.

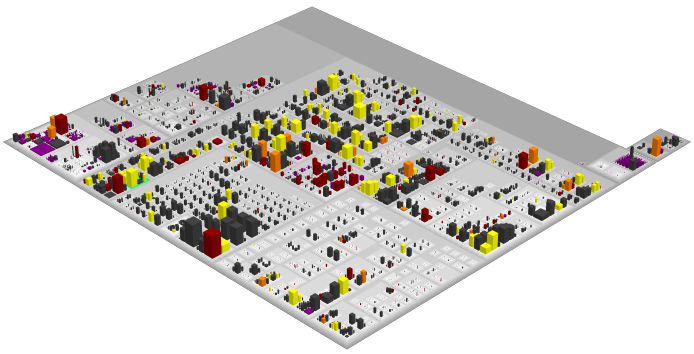


Abbildung 6.3: Screenshot der untersuchten Codebasis aus CodeCity

Im Gegensatz zu den zuvor dargestellten Programmen, kann dieses Programm den Entwicklungsverlauf einer Software berücksichtigen. Dazu sind weitere Produkte, wie iPlasma oder inFusion nötig, um die zu analysierenden Modelle bereit zustellen. Codecity betrachtet nicht nur Anti-Pattern und Größenmetriken, sondern auch die CK-Metriken. Es zeigt deutlich den hierarchischen Aufbau der verwendeten Namensräume und stellt zusätzlich auch auf Wunsch die Afferent Couplings (eingehende Klassenkopplungen) und Efferent Couplings (ausgehende Klassenkopplungen) von Klassen dar.

6.1.4 STAN4J

Structural Analysis for Java ist ein auf der Eclipse Rich Client Platform basierendes Werkzeug. Dieses kommerziell vertriebene Programm analysiert die fertig compilierten Java Archive. Es legt das Hauptaugenmerk auf die Struktur des untersuchten Programms und stellt Abhängigkeiten über Paket- und Archivgrenzen hinweg gut nachvollziehbar dar.

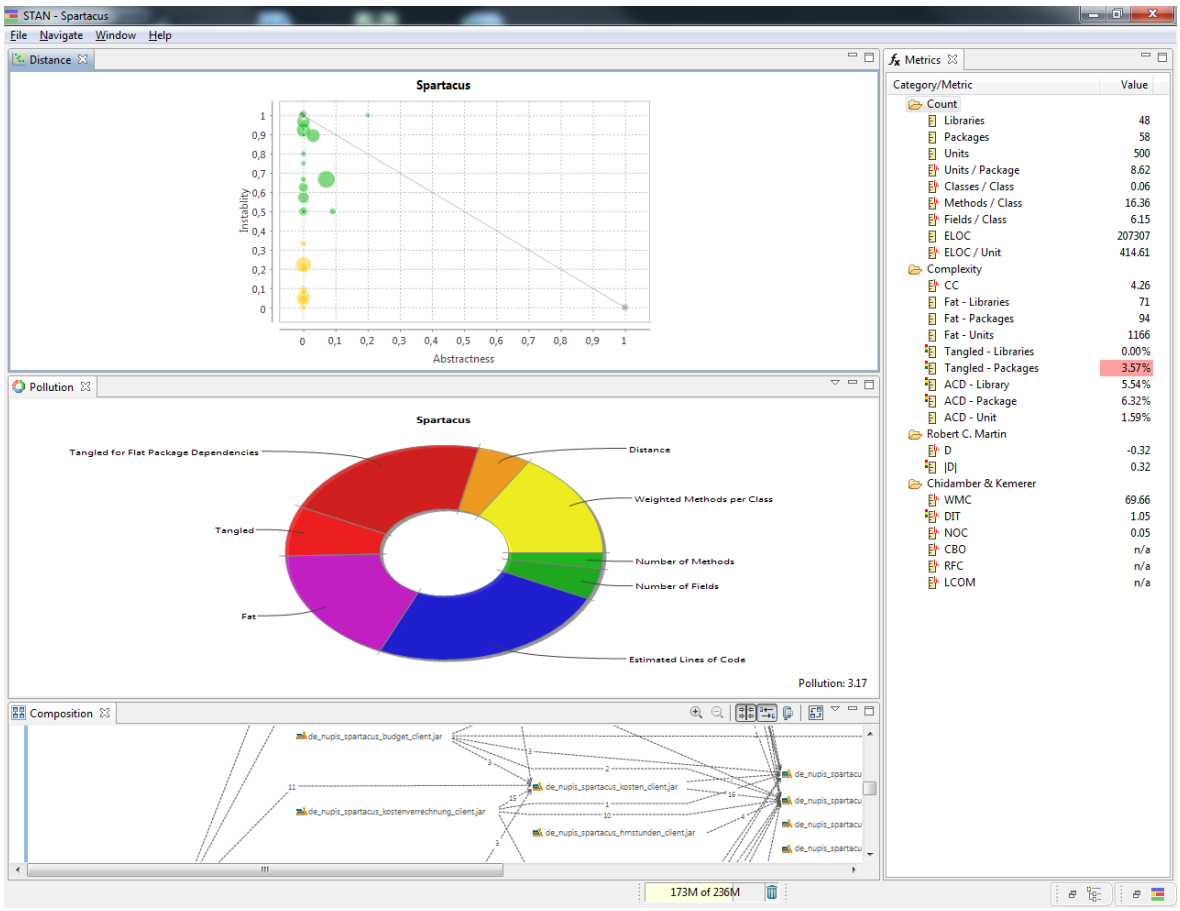


Abbildung 6.4: Screenshot von STAN4J

Die Abbildung 6.4 stellt einen Überblick über ein komplettes Projekt dar. Die Größenmetriken und die zyklomatische Komplexität werden entweder auf Klassen- oder auf Archivebene ausgewiesen. Die CK-Metriken und die Martin-Distanz werden zusätzlich dazu berechnet und dargestellt. Auf Grund eines hinterlegten anpassbaren Qualitätsmodells, dass auf Schwellwerten basiert, errechnet das Programm die Verschmutzung der untersuchten Artefakte. Es zeigt sehr anschaulich, welche Probleme im Einzelnen zu der berechneten Verschmutzung führen.

Der Funktionsumfang dieses Programms reicht aus, um Pakete und Klassen zu identifizieren die einen hohen Kopplungsgrad aufweisen. Für eine qualifizierte Aussage über die erreichte

Qualität reichen die erhobenen Daten aber nicht aus. Des Weiteren kann das Programm nur mittelbar eine Historie darstellen, wenn die eingebaute Reportingfunktion genutzt wird und ihre Ergebnisse manuell abgeglichen werden.

6.2 Integrierte Werkzeuge

6.2.1 Eclipse Metrics

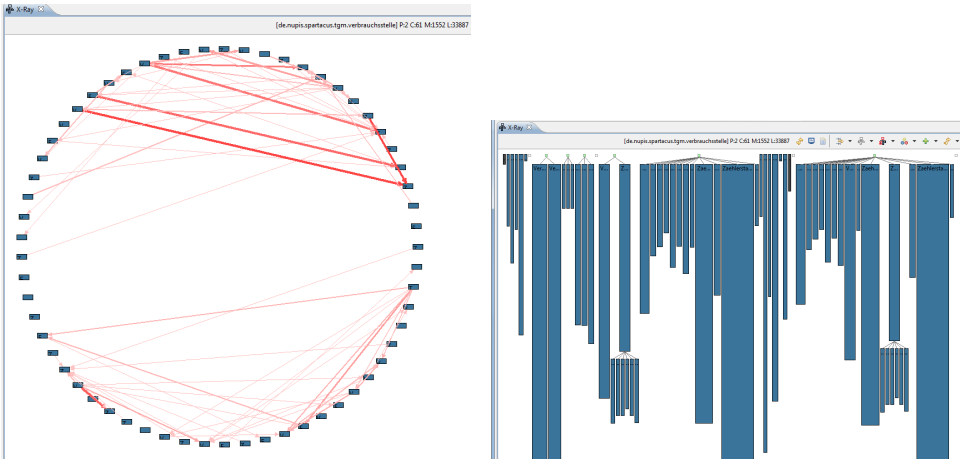
Eclipse Metrics integriert sich als Plugin in die IDE Eclipse. Es ermöglicht die Analyse einzelner Projekte zur Berechnung der gängigen Größenmetriken und die gesamte Palette der CK-Metriken und weiterer Vertreter für objektorientierte Sprachen. Es beinhaltet ein Feature zur Definition von Warnschwellen, die den Entwicklern bei der Nutzung die Interpretation erleichtern sollen. Zusätzlich bietet Eclipse Metrics zu jeder Metrik noch die Möglichkeit, einen Hinweis zu hinterlegen, wie der angepeilte Wert erreicht werden kann.

Ein weiteres Feld, das dieses Plugin abdeckt, ist die Abhängigkeitsanalyse. Es ist möglich die Beziehungen zwischen den Paketen im Workspace als Graph darzustellen. Die Ergebnisse werden aus einer statischen Codeanalyse gewonnen.

6.2.2 X-Ray

Bei X-Ray handelt es sich ebenfalls um ein Plugin für Eclipse. Es wertet statisch die Komplexität von Java Quelltext von der Klassenebene bis zur Projektebene aus. Bei pluginbasierten Systemen fehlt somit der Gesamtüberblick, da die einzelnen Plugins nicht selten auch in einzelnen Java Projekten zusammengefasst sind. Die Komplexitätsansicht stellt neben der Komplexität der Klassen des untersuchten Projektes auch die darüber liegenden Hierarchiestufen außerhalb des Projektes dar.

Ein wichtiger Aspekt der Komplexität sind die Abhängigkeiten zwischen den Klassen, die dieses Plugin in einer anderen Ansicht präsentiert. Die Darstellung ist entweder auf Pakete oder Klassen innerhalb des Projektes limitiert. In Softwaresystemen, die komplett in einem Eclipseprojekt entwickelt werden, kann mit den dargestellten Informationen eine Aussage über die Wartbarkeit getroffen werden.



(a) Klassenabhängigkeiten (b) Klassenhierarchie und -größe

Abbildung 6.5: Screenshots von X-Ray

Das Datenmodell, dass dieses Plugin intern aufbaut, kann abgegriffen und so durch Eigenentwicklungen noch besser nutzbar gemacht werden. Es könnten beispielsweise die Daten mehrere Eclipseprojekte auf diese Weise aggregiert werden, um einen Gesamtüberblick zu ermöglichen.

6.2.3 Checkstyle

Checkstyle ist ein sehr weit verbreitetes Tool, dass sich nicht nur in IDEs integriert, sondern auch in anderen Umgebungen, wie zum Beispiel Continuous Integration Servern oder auch als Standalone Programm funktioniert. Darüber hinaus lässt es sich in Build Systeme wie Ant oder Maven einbinden. Es analysiert statisch den Quelltext und greift auf eine große Menge Regeln zurück, die zumeist parametrierbar und auch erweiterbar sind. Der Fokus

dieser Software liegt nicht auf dem Erheben von Metriken, sondern auf der Einhaltung eines Programmierstils, der zum Beispiel in einem Unternehmen genutzt werden soll, damit alle Entwickler nicht nur ihren eigenen Code, sondern auch den ihrer Kollegen verstehen.

Einige der Regeln zielen auf Komplexität der vorhandenen Artefakte ab. Mit den Parametern dieser Regeln können Richtlinien abgebildet werden, welche Ausmaße und Komplexität die Methoden und Klassen erreichen dürfen. Viele Regeln spiegeln schlechten Stil oder obsolete Muster wieder, die nicht mehr in einer fertigen Software vorkommen sollten. Die Gesamtzahl der Regelverletzungen gibt einen Anhaltspunkt für die Qualität des entwickelten Codes. Es sollte jedoch sorgfältig abgewogen werden, welche Regeln mit welchen Parametern zur Anwendung kommen und welche Auswirkungen die Darstellung der Verletzungen haben kann.

Es existiert nicht nur die Möglichkeit, Regeln global zu aktivieren, sondern sie auch unter bestimmten Umständen mit einem Markerkommentar zu deaktivieren, wenn es sich um generierten Quelltext handelt, auf den kein Einfluss genommen werden kann oder soll. Dieses Programm wird bereits seit einiger Zeit im untersuchten Projekt angewandt und es existiert daher ein vollständiges Regelset und eine Definition von Ausnahmen und den zugehörigen Marker Kommentaren.

6.2.4 Findbugs

Im Gegensatz zu Checkstyle analysiert Findbugs statisch nicht den Quelltext, sondern den bereits compilierten Bytecode. Auf dieser Grundlage kann das Tool sich in IDEs und Continuous Integration Servern integrieren. Eine Anwendung in Buildscripten und als Standalone Programm mit grafischer UI ist ebenfalls vorgesehen.

Die teilweise dynamische Analyse sorgt für die Möglichkeit, Komplexität auf Basis der Ausführungspfade zu ermitteln, da der Bytecode besser auf Schleifen durchforstet werden kann. Die Findbugs Regeldatenbank umfasst viele Muster, die auf mögliche Programmfehler hinweisen. Die Muster verteilen sich größtenteils auf die Kategorien Korrektheit, angreifbarer

Code, schlechte Gewohnheiten und Performance. Ein paar Prüfungen weisen auf Sicherheitsbedenken hin. Die Bytecodeanalyse offenbart auch toten Code, der nicht mehr erreicht werden kann. Zusätzlich zu den Mustern für schlechten Code werden auch Größen- und Komplexitätsmetriken erhoben. Der Basiswert der Größenmetriken wird hier nicht in LoC angegeben sondern in NCSS.

Als Nutzer kann eingestellt werden, welche Regeln bei der Analyse zur Untersuchung herangezogen werden sollen. Alle Regeln sind nach ihren Auswirkungen in einer Skala von 1 (kleinste Auswirkungen) bis 20 (nachgewiesener Defekt) bewertet und können abhängig vom Scanaufwand zu- und abgeschaltet werden.

Die Regelverletzungen, die dieses Tool offenbart, können dabei direkt als Malus für die Codequalität angesehen werden. Je mehr Verletzungen pro NCSS auftreten, desto schlechter ist der entwickelte Code. Die Regeln, die in der Standardeinstellung die höchste Kritikalität aufweisen sind in der Regel echte Programmierfehler, die dringend abgestellt werden müssen.

Auch in großen Unternehmen kommt Findbugs zum Einsatz. Beispielsweise nutzt die Firma Google dieses Programm in all ihren Projekten. Im Jahr 2009 wurde ein großer Reviewprozess innerhalb der ganzen Unternehmenscodebasis durchgeführt. Dieser führte, wie in [AP10] dargestellt, dazu, dass 282 Entwickler in einem Zeitraum von 3 Monaten etwa 4000 Problemfälle begutachteten. Die Begutachtung führte zu 1746 Bug Reports von denen 640 im Untersuchungszeitraum abgearbeitet werden konnten.

6.2.5 PMD

Dieses Toolkit analysiert den vorliegenden Sourcecode statisch und wendet dabei ein sehr feingliedrig konfigurierbares Regelset an. Der Fokus liegt bei diesem Programm nicht auf der Erhebung von Metriken, sondern auf dem Aufspüren von möglichen Programmfehlern, umständlich programmiertem Code und nicht mehr erreichbarem oder ungenutzten Code.

Trotz des eigentlichen Programmziels, Programmfehler aufzudecken, ermittelt PMD in einem Regelset Größenmetriken, die zyklomatische Komplexität und die Zahl der Codepfade auf Methoden- und Klassenebene. Zu den enthaltenen Programmen gehört ein sogenannter CPD. Der Anteil an kopiertem Code kann als ein Indikator für schlechten Code dienen, wenn es sich nicht um generierten Code handelt. Eine Betrachtung des Anteils kann aber in jedem Fall von Vorteil sein, um eine gewisse Awareness auszuprägen, wie viel Code neu entwickelt wurde und wie viele Fehler im duplizierten Code sich auf das Gesamtsystem auswirken können.

Die Darstellung der Analyseergebnisse des Stand-Alone-Programms ist nicht gut dafür geeignet, einen Review der Ergebnisse durchzuführen. Es empfiehlt sich daher die Nutzung in den unterstützten IDEs, in einen vorhandenem Continuous Integration Server oder auch innerhalb der Analyseplattform Sonar, auf die im folgendem Kapitel eingegangen wird.

Genau wie bei FindBugs sollte dieses Werkzeug als Teil einer ganzen Analyseumgebung genutzt werden, um die Softwarequalität bestimmen zu können. Die Anzahl der Warnungen in Abhängigkeit der Artefaktgröße ist ein brauchbarer Indikator für Qualität.

6.2.6 Sonar

Sonar ist ein Werkzeug, deren Frontend als Stand Alone Lösung und als Anwendung in einem Java Enterprise Edition (JEE)-Container laufen kann. Es handelt sich bei genauerer Betrachtung, um ein mehrschichtiges System das in Abbildung 6.6, aus dem Wiki zur Software, dargestellt ist. Um eine Analyse durchzuführen kann der Runner auf verschiedenen Wegen aufgerufen werden. Er bezieht alle aktuellen Einstellungen vom Web Interface und schreibt seine Analyseergebnisse direkt in die Sonar Datenbank. Nach dem Abschluss der Analyse können die Ergebnisse im Web Frontend betrachtet werden und sind dann auch in der IDE Eclipse darstellbar. Sonar vereinigt eine ganze Reihe von statischen Analysetools und Unittest-basierten dynamischen Werkzeugen.

Durch diese Menge an vereinigten Programmen ist die Datenmenge, die anfällt, groß genug, um eine Aussage über die erreichte Codequalität zu treffen. Die Qualität definiert Sonar über die Abwesenheit von Regelverletzungen pro Größeneinheit. Diese Aussage trifft aber nur dann zu, wenn das Regelset entsprechend den genauen Anforderungen und Zielsetzungen im Projekt eingestellt worden ist. Die Feinjustierung nimmt einige Zeit in Anspruch, da jeder Regel auch ein Schweregrad zugeordnet wird.

Die Daten, die bei der Analyse anfallen, sind von Beginn an historisiert, sodass sich Trends klar nachweisen lassen und der Erfolg einer Qualitätsoffensive sichtbar wird. Die Möglichkeit, den Umfang der aggregierten Daten durch eigene Plugins zu erweitern, steigern den Nutzwert dieser Platform nochmals. Eine eingebaute Reviewfunktion sorgt für eine mögliche Arbeitsteilung zwischen Entwicklern und Testern.

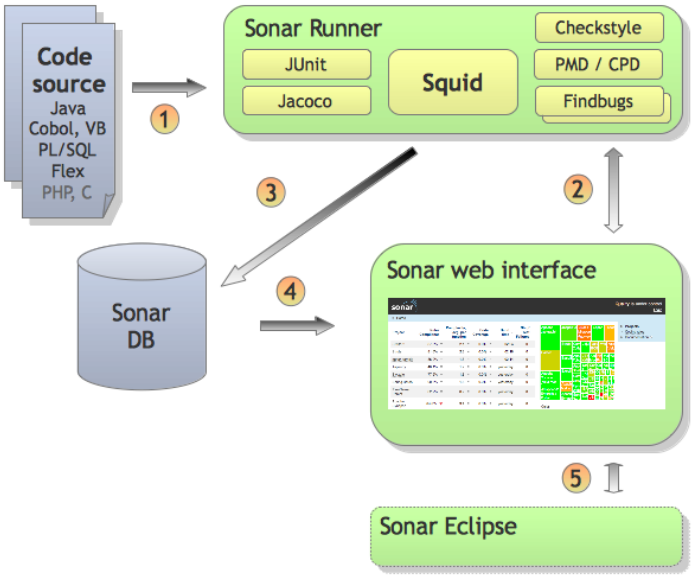


Abbildung 6.6: Architektur von Sonar

Die Daten, die Sonar bereitstellt, sind gleichsam auch geeignet, um der Managementebene eines Unternehmens ein Gefühl zu vermitteln, wie viel die entwickelte Codebasis in den Projekten wert ist. Sie sollte allerdings mit Bedacht eingesetzt werden, um voreilige Schlüsse zu vermeiden. Unreflektierte Zahlen sagen wenig aus, wenn dahinter keine Erklärung eines Entwicklers oder ähnlichen Stakeholders liegt. Des Weiteren sollten die Entwickler, deren

Arbeit überwacht werden soll auch wissen, dass dies geschieht und die Möglichkeit haben, die Daten selbst in Augenschein zu nehmen und für sich zu werten. Dies hat zur Folge, dass sie selbst ihre Ergebnisse reflektieren und zukünftig eine stetige Verbesserung anstreben oder zumindest ihr Niveau halten.

Der Funktionsumfang beinhaltet alle Ergebnisse von Checkstyle, Findbugs und PMD. Zusätzlich dazu liefert Sonar selbst noch einige Metriken und bindet wenn gewünscht JUnit für Unittests oder diverse andere Programme für die Berechnung der Testabdeckung an. Die Darstellung ist umfassend im Detaillierungsgrad und nach Rollenkonzept anpassbar. Es steht ein Application Programming Interface (API) bereit, die eine Erweiterung der Messungen und Reportingfunktionen ermöglichen.

7 Qualitätsmodell

7.1 Ausgangspunkt

Im untersuchten Projekt kam bisher keine konkrete Metrikerhebung zum Einsatz. Stattdessen wurde eine rudimentäre Untersuchung mit Checkstyle durchgeführt. Diese beinhaltete eine kleine Anzahl an Größenprüfungen und hatte kaum Einfluss auf die Entwicklungstätigkeit. Der Test der Software verlief und verläuft auch nach wie vor ausschließlich manuell. Das Modell, das im Folgenden erarbeitet werden soll, entsteht nach dem GQM-Ansatz.

Trotz der vorgeschlagenen Trennung zwischen Produktqualität und Prozessqualität sollten beide mit ihren jeweiligen Wechselwirkungen zueinander betrachtet werden. Auch wenn das Ziel eine optimale Produktqualität unter akzeptablem Einsatz finanzieller Mittel ist, kann dies nur durch einen guten Entwicklungsprozess erreicht werden.

7.2 Interview

Um die einzelnen Punkte gewichten zu können, wurde innerhalb des Entwicklungsteams eine Diskussionsrunde geführt, deren Ziel es war einige Qualitätskriterien festzulegen. Dabei stellte sich heraus, dass solch eine Diskussion alleine nicht ausreicht, um alle Kriterien auszuwählen und abzuwägen. Daher wurde gemeinschaftlich beschlossen, alle Beteiligten einzeln zu befragen und die Ergebnisse zur Entwicklung eines Modells heranzuziehen. Die

Fragen waren dabei angepasst an die befragte Gruppe des Teams abhängig von ihrem Aufgabenbereich.

Der Fokus der Befragung zielte zwar auf die untersuchte Software ab, die waren jedoch offen genug formuliert, um ein allgemeines Set an Qualitätskriterien für die Softwaregattung „verteilte Anwendungssoftware“ zu gewinnen.

Alle Befragten gaben an, dass sie eine intuitive Bedienbarkeit bei einer guten Software voraussetzen. Dieses Kriterium lässt sich nur schwer mit vertretbarem Aufwand durch eine Metrik abbilden. Wenn es für eine Software einen Styleguide gibt, dann können zwar Verstöße dagegen gezählt werden. Dies lässt aber keine allgemeine Aussage zu, wie intuitiv bedienbar die Software ist, da der Styleguide selbst fehlerhaft oder unschlüssig sein könnte. Eine Software in der in allen Teilen klar ersichtlich ist, welchem Zweck eine Oberfläche dient und in der auch alle Oberflächen einen gleichen Stil haben, finden sich Nutzer letzten Endes gut zurecht, da sie gleiche Bedienelemente auch immer am gleichen Platz finden.

Ein weiterer Punkt, der vielen Befragten wichtig schien, war die Datenintegrität sowohl während der täglichen Arbeit, als auch bei der Durchführung von Updates. Sie erwarten, dass ihre Daten erhalten bleiben und selbst bei fehlerhafter Eingabe keine Datenverluste entstehen. Dieses Kriterium lässt sich mit Hilfe von Unittests zumindest auf der Serverseite testen. Dazu muss eine ausreichende Pfadüberdeckung innerhalb der Klassen erreicht werden und eine vollständige Implementierung aller geforderten Use-Cases vorliegen. Es müssen weiterhin alle denkbaren Fehlerklassen in den Tests in Betracht gezogen werden. Nur dann lässt hohe Testabdeckung auch auf eine starke Datenintegrität schließen.

Eine weitere Anforderung sind schlüssige Fehlermeldungen in allen möglichen oder zumindest denkbaren Fällen. Zusammen mit einer guten Dokumentation, sowohl auf der Codeseite als auch einer ansprechenden Hilfe und Dokumentation für den Anwender, liegt damit die Grundlage für eine hohe Akzeptanz durch Transparenz. Entwickler freuen sich dann über eine schnellere Einarbeitungszeit in den Code. Die Anwender können sich auf die Meldungen des Systems einstellen.

Etwas seltener wurde im Interview die Verarbeitungsgeschwindigkeit als Qualitätskriterium genannt. Das lässt auf den Gewöhnungseffekt bei umfangreichen Softwaresystemen schließen. Es gibt Verarbeitungsabläufe, bei denen der Nutzer eine lange Verarbeitungszeit akzeptiert. Es gibt aber auch Jobs, die direkt innerhalb der Serverkomponente zur Anwendung kommt, bei denen das Ergebnis nicht zeitkritisch ist.

Ein Teil der Befragung befasste sich mit der Zufriedenheit mit dem untersuchtem Softwaresystem. Die Befragten sollten ihre Meinung zur allgemeinen Bedienbarkeit, zur Dokumentationsqualität (abhängig von der Tätigkeit des Befragten), zur Produktqualität und zur Erfüllung der Erwartungen an Softwaresysteme abgeben. Es kam eine Skala von 1 bis 10 zum Einsatz, dabei stellte 1 den schlechtesten Wert und 10 den besten Wert dar.

Die Bedienbarkeit lag mit geringer Streuung bei etwa $6,85$ und wird damit als gut empfunden. Die Dokumentationsqualität für Anwender war mit $4,64$ eher mittelmäßig und ist noch sehr stark ausbaufähig. Die Erwartungen an ein Facility Management System erfüllt das untersuchte Produkt nach Ansicht der Befragten bei mäßiger Streuung ebenfalls gut $(7,25)$. Als Softwareprodukt allgemein betrachtet geht die Streuung stark auseinander. Der Durchschnitt erreicht $7,11$ wobei die Antworten sich auf den Bereich zwischen 4 und 10 verteilen.

Bei der Frage nach den Stärken und Schwächen der untersuchten Software kamen die Erwartungen zum Vorschein, die die Befragten entweder gut oder schlecht erfüllt sahen. Als größte Stärke gaben viele der Befragten die hohe Flexibilität und Anpassbarkeit an, die unter anderem damit begründet wird, dass die Software auf einem soliden Framework steht.

Das solide Framework führt aber nach Ansicht einiger Befragter auch zu einem wenig attraktiven und als nicht mehr zeitgemäß empfundenen UI. Als weitere Schwachstellen sind die langsame Massendatenverarbeitung und die zum Teil nicht intuitiv bedienbaren Programmteile genannt worden. Auch an einer mangelnde Integration zwischen einzelnen Teilen des Programmes störten sich einige der Interviewteilnehmer.

Nur ein geringer Teil der Befragten ging genauer auf die Fragen zu Websoftware ein. Das lag vermutlich daran, dass der Funktionsumfang im Webfrontend des untersuchten Systems nur einen kleinen Teil der Möglichkeiten des Gesamtsystems widerspiegelt. Die vereinzelten Rückfragen, was eine Websoftware ausmacht, lässt erahnen, dass Teile der Befragten bisher nur wenige Erfahrungen mit Websoftware hatten oder sie nicht als solche wahrnahmen.

Es sei an dieser Stelle bemerkt, dass insgesamt nur 12 Personen interviewt wurden und sich daraus kein empirisch einwandfreies und fundiertes Bild ergeben kann.

7.3 Ziel

Der erste Schritt zu einer Berechnung, beziehungsweise Schätzung, der erreichten Produktqualität soll größtenteils von den Codemetriken abhängen, da diese aufwandsarm und automatisch erhoben werden können. Die Prozessmetriken sollen auch in Betracht gezogen werden. Da sie unter den aktuellen Voraussetzungen schwer zu erheben sind, sollen sie mit geringerem Gewicht in das Modell eingehen. Das Ziel, dass an der Spitze des Modells liegt ist die Frage:

Ziel 1

Welche Produktqualität erreicht unser Produkt unter dem aktuellen Mitteleinsatz?

Ziel 2

Welche Produktqualität kann unser Produkt unter dem aktuellen Mitteleinsatz erreichen?

Vorhandene Datenquellen, die schon zur Verfügung stehen, sind die Quelltexte im SVN-Repository, die vorhandenen Ticketdaten im Projektmanagementtool Redmine, die erhobenen Daten aus dem Interview und die Daten über Nutzeranfragen aus dem verwendetem Servicemanagementtool helpLine.

Um die Grundlage für eine stetige Qualitätsmessung zu legen, sollen punktuelle Maßnahmen ergriffen werden, die wiederholbar zu Aussagen über den momentanen Stand der Qualität

führen. Diese Maßnahmen sollen insgesamt in eine Unternehmensrichtlinie münden, die im CMMI Maturity Level „Managed“ gefordert ist.

7.4 Fragen

Die Grundlage für die Aufstellung der Fragen war das Interview, deren Ergebnisse in Kapitel 7.2 dargestellt sind. Dabei wurde der Rahmen abgesteckt, der durch das Qualitätsmodell abgedeckt werden soll.

Um die Fragen aus den Zielen beantworten zu können, müssen möglichst umfassend die gewünschten Aspekte als Fragen formuliert werden. Die erste Frage dreht sich darum, zu welcher Größe das Projekt bereits herangewachsen ist.

Frage 1
Welchen Umfang hat das Produkt?

Diese Information soll in erster Linie dazu dienen, die Vergleichbarkeit mit anderen Systemen mit ähnlich großem Funktionsumfang zu quantifizieren.

Frage 2
Wie gut ist das API dokumentiert?

Damit soll festgestellt werden, wie gut sich neue Teammitglieder in die Schnittstellen einarbeiten können. Aus der Beantwortung der beiden Fragen lassen sich Erkenntnisse gewinnen, mit welchem Aufwand neue Entwickler in das Projekt eingebunden werden können.

Frage 3
Wie stark hängen die einzelnen Module voneinander ab?

Diese Frage ist wichtig im Hinblick auf Testbarkeit und Wartbarkeit, da das Softwareprodukt bei jedem Kunden einen anderen Funktionsumfang besitzen kann. Dies ist in der Regel auch der Fall, da es nur selten Kunden gibt, die ein identisches Anforderungsprofil haben und die gleichen Geschäftsprozesse mit dem Produkt unterstützen.

Die nächste Frage befasst sich mit der Thematik, wie gut nutzbar die Software vom Anwender ist. Dabei kommt es darauf an, dass alle Funktionen von den Nutzern intuitiv gefunden und nach einer entsprechenden Schulung auch genutzt werden können.

Frage 4
Wie gut ist die Funktionalität dokumentiert?

Die folgende Fragengruppe dreht sich um die Entwickler. Der Fokus liegt auf der Menge an produzierter Funktionalität bzw. entfernten Softwaredefekten. Bei dieser Gruppe liegt der eigentliche Fokus des Modells, da hier die meisten Daten mit geringstem Aufwand erhoben und interpretiert werden können. Die beiden letzten Fragen der Gruppe zielen darauf ab zu quantifizieren, wie sich die Änderungsrate des Softwareproduktes eigentlich zusammen setzt. Zusammen ergibt sich daraus ein Bild, an welcher Stelle im Softwarelebenszyklus sich das Produkt befindet.

Frage 5
Wie viele Fehler werden pro Release gemeldet oder gefunden?

Frage 6
Wie viele Anforderungen werden pro Release eingebracht?

Frage 7
Wie viele Fehler werden entfernt?

Frage 8
Wie viele Anforderungen werden umgesetzt oder abgewiesen?

Die folgenden Fragen gehen insbesondere auf die bereits etablierten Richtlinien ein, die mit den Werkzeugen aus Kapitel 6 erhoben werden.

Frage 9
Wie gut halten sich die Entwickler an die vorgegebenen Programmierrichtlinien?

Frage 10
Wie viele sicherheitsrelevante Richtlinienverletzungen gibt es?

Frage 11
Wie gut halten sich die Entwickler an die vorgegebenen Dokumentationsrichtlinien?

Hierbei geht es um die Einhaltung der Richtlinien in den unterstützenden Tools, die zum Einsatz kommen. Dabei geht es insbesondere um die täglichen Änderungen und den Trend, wie sich die Anzahl der Richtlinienverletzungen entwickelt.

Frage 12
Wie stark sind die Entwickler ausgelastet?

Diese Frage zielt darauf ab, ob die geplante Entwicklungskapazität und die geplanten Releasezyklen mit ihrem Umfang zusammen ein stimmiges Bild ergeben. Die folgenden Fragen befassen sich direkt mit dem Quelltext und den damit verbundenen Eigenschaften zur Komplexität und Wiederverwendbarkeit.

Frage 13
Wie hoch ist der Anteil an dupliziertem, nicht generierten, Quelltext?

Frage 14
Wie hoch ist der Anteil generiertem Quelltextes?

Frage 15
Wie komplex sind die einzelnen Artefakte?

Frage 16
Wie hoch ist der Anteil unerreichbarem Quelltextes?

Die nächste Frage dreht sich um die Testbarkeit der einzelnen Artefakte, aus denen sich das Produkt zusammensetzt.

Frage 17

Wie hoch ist die Testabdeckung von manuellen und Unittests.

Frage 18

Wie zuverlässig ist die Architektur des Gesamtsystems?

Frage 19

Wie komplex sind die einzelnen Programmteile aufgebaut?

Frage 20

Wie hoch ist die technische Schuld zur Beseitigung alle Qualitätsprobleme?

Diese Sammlung von Fragen deckt den Informationsbedarf des Entwicklungsleiters, der Entwickler und des Managements. Der Entwicklungsleiter und das Management bekommen schnell einen Überblick über den aktuellen Zustand des Produktes. Die Entwickler können selbstständig ihre eigene Arbeit überprüfen und gegebenenfalls auch nachsteuern. Diese Transparenz kann zu einer hohen Akzeptanz führen und damit auch einer stetigen Verbesserung.

7.5 Metriken

Im folgenden werden die gewählten Metriken dargelegt und deren Zuordnung zu den vorangegangenen Fragen klargestellt.

Metrik 1

Non Commented Source Statements (NCSS)

Die NCSS werden erhoben, damit die Frage 1 beantwortet werden kann. Dabei wird sie so kleinteilig wie möglich, also auf Methodenebene, erhoben. Sie stellen den Basiswert dar, um Relationen für Komplexität und Regelkonformität zu ermöglichen. Sie wird auf Klassen-, Paket-, Plugin- und schließlich Projektebene aufsummiert.

Metrik 2

redundanzfreie Source Lines of Code (RSLOC)

Die RSLOC beantworten zum Teil die Frage danach, wie hoch der generierte Anteil des Codes ist. Eine zweite Information, die aus dieser Metrik gewonnen werden kann, ist der Anteil des manuell duplizierten Codes. Diese Codeduplikate stellen ein Qualitätsrisiko dar, denn sie können Fehler enthalten. Da sich diese Fehler dann über das Gesamtsystem verteilen, leidet im Allgemeinen auch die Wartbarkeit.

Metrik 3

Lines of Comment

Die Erhebung der Lines of Comment ist nötig, um den Dokumentations- und Codeanteil in den folgenden beiden Metriken beziffern zu können.

Metrik 4

Anteil Code

Metrik 5

Anteil Kommentare

Die vorangegangenen vier Metriken beantworten die Fragen 1, 2 ,13 und 14. Damit stehen grundsätzliche Aussagen zur API-Dokumentation und zur Codedokumentation für die Einarbeitung neuer Entwickler bereit. Eine Aussage über die Qualität der vorhanden Kommentare kann dabei nicht getroffen werden, da die Kommentarqualität nicht erhoben wird.

Metrik 6

Unerreichbare Codezeilen

Metrik 7

Anzahl auskommentierter Codezeilen

Die Anzahl der unerreichbaren Codezeilen liefert die Antwort auf die Frage 16. Diese Metrik verdeutlicht, wie sich die Weiterentwicklung eines Produktes auswirken kann, wenn z.B. Funktionalität entfällt oder ersetzt wird. Das Bild wird durch die auskommentierten Codezeilen vervollständigt. Sie stellen eine schlechte Gewohnheit von Entwicklern dar und sollten deshalb nicht toleriert werden. Zudem vergessen Entwickler irgendwann, warum der Code auskommentiert oder unerreichbar gemacht wurde, wenn dies nicht aus Kommentaren ersichtlich ist.

Unerreichbarer Code sollte nicht in der Codebasis existieren, da er den Aufwand zur Beseitigung eines Fehlers erhöhen kann. Auskommentierte Codezeilen führen bei Entwicklern nach einiger Zeit zu Verwirrung und stellen die Nutzung eines richtigen Source Code Management (SCM)-Systems in Frage. Diese Zeilen sollten mit hoher Priorität überprüft und aus der Codebasis entfernt werden.

Metrik 8

Anteil generierter Code

Diese Metrik beantwortet die Fragen 13 und 14. Der Anteil ist wichtig für die Bewertung der verwendeten Templates. Generierter Quelltext sollte im Rahmen der Programmierrichtlinien alle Vorgaben einhalten, um dem gewünschten Qualitätsanspruch gerecht zu werden. Richtlinienverletzungen, die in den Templates vorhanden sind, sind mit einer höheren Priorität zu beheben als normal entwickelter Code.

Metrik 9

Zyklomatische Komplexität

Metrik 10

Halstead Difficulty

Metrik 11

Weighed Methods per Class (WMC)

Diese drei Metriken geben Auskunft über die Komplexität der einzelnen Artefakte aus Frage 15. Dabei wird die zyklomatische Komplexität und die Halstead Difficulty auf Methodenebene berechnet und die McCabe Metrik in WMC auf Klassenebene aggregiert. Die durchschnittliche Komplexität wird auch auf Pluginebene betrachtet. Eine Betrachtung der durchschnittlichen Komplexitätswerte auf Klassen-, Paket- oder Artefaktebene liefert Hinweise, an welchen Stellen der Codebasis im einzelnen Refactoring nötig ist. Für die Bewertung hat die Halstead Difficulty nur informativen Charakter, da die Erhebung nicht den kognitiven Fähigkeiten der Entwickler Rechnung trägt.

Metrik 12
Abstractness

Metrik 13
Instability

Metrik 14
Martin Distance

Diesen drei Metriken haben bei der momentanen Architektur des Produktes nur eine kleine Bedeutung. Da viele der Klassen generiert sind, ergeben sich für alle Klassen ähnliche Werte, deren Aussagekraft nur marginal ist. Bei Paketen, die eine hohe Martin Distanz aufweisen, ist zu prüfen, ob sich der Wert durch Refactoringmaßnahmen verbessern lässt. Eine sinnvolle Vorgehensweise wäre die Erhöhung der Abstractness durch Generalisierung oder eine Veränderung der Kopplungen.

Metrik 15
Anzahl Supportickets (neu, gelöst) nach Anfrageart

Die Anzahl der Supportfälle wird für den Entwicklungszeitraum eines Releases betrachtet. Daraus lassen sich Antworten für die Fragen 4,5 und 6 herleiten. Sie gehen mit einer hohen Gewichtung je nach Anfrageart in die Gesamtbewertung ein, da sie sich direkt auf den Entwicklungsfortschritt auswirken können. Das Ziel, die Zahl der kritischen Supportanfragen zu senken, wirkt sich direkt auf die Innovationsfähigkeit des Entwicklungsteams aus.

Metrik 16

Anzahl Entwicklungsaufgaben nach Typ und Ergebnis

Metrik 17

Anzahl der Aufgabetags im Quelltext nach Art und Klasse

Die Übersicht über die geplanten Entwicklungsaufgaben ist eine der täglichen Arbeitsschritte der Entwickler und des Entwicklungsleiters. Die Daten, die sich aus dem Projektverwaltungstool extrahieren lassen, sind nötig für die Beantwortung der Frage nach der Auslastung der Entwickler, der Anzahl der entfernten Fehler und der Zahl der umgesetzten, eingebrachten und abgewiesenen Anforderungen.

Metrik 18

Realisierte geplante Entwicklungszeit

Diese Metrik dient der Erfassung der Entwicklerauslastung und zur zukünftigen besseren Planung derselben.

Metrik 19

Anzahl Regelverletzungen nach Priorität

Metrik 20

Anzahl sicherheitsrelevanter Regelverletzungen

Diese beiden Metriken liefern die Antworten, zu den Fragen 9 und 10. Dabei stellen sie eine breite Grundlage für die Gesamtbewertung dar. Ihre Anzahl sollte stetig abgebaut werden. Eine Priorisierung muss nach der zugeordneten Wichtigkeit erfolgen. Die Zielwert dieser Metriken liegt bei jeweils 0.

Metrik 21

Afferent Couplings (eingehende Klassenkopplungen) (Ca)

Metrik 22

Efferent Couplings (ausgehende Klassenkopplungen) (Ce)

Die Klassenkopplungen geben einen Aufschluss über die Architektur und die Kopplung zwischen den einzelnen Modulen. Sie sind die Antwort auf die Fragen 3 und 18. Eine starke Kopplung wirkt sich negativ auf die Wartbarkeit aus, da Änderungen an einer Klasse Auswirkungen auf die verbunden Klassen haben können.

Metrik 23

Ergebnis der Unittests

Metrik 24

Pfadabdeckung der Unittests

Metrik 25

Abdeckung der manuellen Tests

Diese drei Metriken beantworten ausführlich die Frage 17. Ihre Ergebnisse sind von größter Wichtigkeit für die Bewertung des Gesamtsystems. Idealerweise sollten Unittests mindestens 85% aller Pfade abdecken, um viele der vorhandenen Fehler vor der Auslieferung schon entdecken zu können. Manuelle Tests sollten alle nicht automatisch erfassten Codeteile abdecken, um eine hohe Fehlererkennungsrate sicherstellen zu können.

Metrik 26

technische Schuld

Die technische Schuld gibt von der Klassenebene bis hin zur Projektebene einen Aufschluss darüber, wo mit dem Refactoring begonnen werden sollte. Nach und nach sind die größten Problemfälle abzubauen, um kontinuierlich die Qualität zu erhöhen. Damit wird die Frage Nummer 20 beantwortet. Die Technische Schuld sollte einen Ausschlag dafür geben, in welchem Umfang zukünftig Refactoringmaßnahmen in den Entwicklungsprozess eingeplant werden sollen.

8 Implementierung

Vor der Erarbeitung dieser Masterarbeit kamen im Entwicklungsprozess nur wenige institutionalisierte Qualitätssicherungsmaßnahmen zum Einsatz. Wie bereits beschrieben, waren zwar Programmierrichtlinien vorhanden, deren Einhaltung wurde aber nicht konsequent überprüft oder angemahnt.

Die Tools, mit denen auf Richtlinienverletzungen geprüft wurde, integrierten sich nicht in ein Entwicklungsprozess und wurden daher auch nicht von den Entwicklern im gewünschten Maße eingesetzt. Der Umstieg vom Cronjob auf ein Continuous Integration System sorgte für den ersten Schritt zu einer institutionalisierten Qualitätssicherung. Die Richtlinien wurden weiterhin im nächtlichen Build geprüft. Die Visualisierung der Ergebnisse erfolgte nun aber direkter und fand mehr Beachtung bei den Entwicklern.

Ein Schritt hin zur Verbesserung der Dokumentation war der Entschluss des Managements, mehr über die Aktivitäten innerhalb der Entwicklungsabteilung erfahren zu wollen. Diese administrative Entscheidung erhöhte die Dokumentationsdisziplin für alle Entwicklungsvorgänge. Der bereits vorhandene Workflow wurde erweitert und konsequent umgesetzt. Es erfolgte eine Einteilung der Aufgaben in 17 Kategorien, was die Übersichtlichkeit und die fachlichen Verantwortlichkeiten deutlich transparenter gestaltete.

Um den Entwicklern die Dokumentation zu erleichtern, wurden die Daten im Projektmanagementtool mit denen des Continuous Integration Systems und des SCM integriert. Eine

Kopplung zwischen der von den Entwicklern genutzten IDE und dem Projektmanagementtool sorgt für die Möglichkeit, direkt aus der IDE heraus Entwicklungsaktivitäten zu dokumentieren.

Zu einer guten Dokumentation gehören neben der Dokumentation im Quelltext und der Aktivitätsdokumentation auch die Programmdokumentation und Nutzerhilfe. Der Workflow in der die Hilfe und Programmdokumentation entsteht, wurde im Rahmen der Arbeit nicht angepasst. Die Hilfe selbst wurde auch nicht auf Qualität überprüft da es keine Richtlinie dafür gibt wie sie gestaltet sein soll. Im Zuge einer weitergehenden Institutionalisierung der Prozesse sollten Richtlinien zur Dokumentationserstellung erarbeitet werden, die zukünftig auch zum Qualitätstest herangezogen werden können.

Für die Erhebung der Metriken wurde eine Installation des Continuous Inspectation Tools Sonar eingerichtet. Die Erhebung läuft wöchentlich durch den Continuous Integration Server auf dem aktuellen Entwicklungsstand. Der verwendete Richtliniensatz ist auf der Grundlage der schon vorhandenen Richtlinien entstanden und zusammen mit den Entwicklern angepasst und auf die Bedürfnisse und die Ausrichtung des Projektes zugeschnitten. Durch die wöchentliche Erhebung sind bereits genug Daten zusammen gekommen, die eine Trendanalyse ermöglichen.

Der Entwicklungsleiter beobachtet in regelmäßigen Abständen den Aktivitätsverlauf im Projektmanagementtool und passt die Aufgabenverteilung im Bedarfsfall an. Dabei zieht er momentan noch keine während der Continuous Inspectation erkannten Qualitätsschwächen in Betracht. Der Grund dafür ist das Fehlen eines Standardprozesses zum Refactoring vorhandener Artefakte. Einige Entwickler versuchen kleinere Refactoringschritte während der Bearbeitung normaler Entwicklungsaufgaben mit einfließen zu lassen. Solange die Einhaltung der Programmierrichtlinien nicht verpflichtend ist, gibt es kaum einen Anreiz dafür einen Standardprozess zu entwickeln. Da sich das Qualitätsbewusstsein langsam verbessert, wird dieser Schritt nach und nach an Priorität gewinnen und schließlich auch umgesetzt werden.

Parallel zur Beobachtung durch den Entwicklungsleiter dient die transparente Erhebung auch der Selbstreflexion. Die Entwickler können selbstständig die Qualität ihrer Arbeit nachprüfen und gegebenenfalls verbessern. Diese Transparenz schaffte eine grundlegende Akzeptanz beim Entwicklerteam. Im Zuge der Einführung erhielten die Beteiligten eine Einweisung, wo welche Informationen im Continuous Inspectation System zu finden sind. Die Strategien zur Behebung der aufgetretenen Richtlinienverletzungen obliegen den Entwicklern selbst.

Der Job für den nächtlichen Build im Continuous Integration Server Jenkins wurde ergänzt um ein Feature, das aus den Änderungen an den Richtlinienverletzungen in Checkstyle und Findbugs einen Punktewert errechnet. Diese Punkte werden allen Entwicklern gutgeschrieben oder abgezogen, die seit dem letzten nächtlichen Build Änderungen an der Codebasis vorgenommen haben. Diese Maßnahme hatte mehrere Auswirkungen. Zuerst betrachteten die Entwickler den Vorgang mit Argwohn, da das Team gemeinsam für die Verletzungen der Kollegen Punkte abgezogen bekam. Später erwachte der Ehrgeiz großer Teile des Teams, konzertiert für einen Abbau der Richtlinienverletzungen in der Codebasis zu sorgen. Es kam darüber hinaus zu kleinen Wettkämpfen, die die Qualität spürbar verbesserten. Die Unzulänglichkeiten, die das Jenkins-Plugin „CI-Game“ mit sich bringt, können auf Grund des geringen Umfangs und der Verfügbarkeit unter Open-Source Lizenz schnell durch Weiterentwicklung abgebaut werden.

Die Punktevergabe war ein Ansporn, den Auswirkungen der Broken Windows Theorie entgegenzuwirken, die an der Codebasis beobachtet werden kann. Einige Plugins aus dem betrachteten System konnten im Zuge der Bearbeitung einer größeren Programmieraufgabe großflächig refactored werden. Es hat sich heraus gestellt, dass eine kleine Zahl an Regelverletzungen von den Entwicklern gern nebenbei behoben wird, statt extra Zeit für das Refactoring einzuplanen und eine groß angelegte Reinigungsmaßnahme durchzuführen. Die kleinen Verbesserungen werden aber nur in Angriff genommen, wenn sich die Zahl der Regelverletzungen im bearbeiteten Artefakt in Grenzen hält. Hat die Zahl der Verstöße eine Größe erreicht, die nicht mehr nebenbei korrigiert werden kann, stellt sich eine gewisse Toleranz gegenüber neuen Verletzungen ein. Wenn dieser Punkt erreicht ist, kann nur noch eine geplante Korrekturaktion die gewünschte Qualitätssteigerung herbeiführen.

Das transparente Verfahren bei der Einführung von Sonar sorgte beim Großteil der Entwickler umgehend für eine Steigerung der Disziplin bei der Anwendung der Programmierrichtlinien. Die Ergebnisse der Erhebungen spiegelten größtenteils die vorhandene Meinung zur Qualität des untersuchten Produktes wider. Da das subjektive Qualitätsgefühl nun mit Metrikwerten untermauert werden kann, fällt es den Entwicklern nun leichter, Missstände zu erkennen und zu korrigieren.

Zusätzlich zu der wöchentlichen Erhebung sollten die Metriken, die zwar zum Qualitätsmodell gehören aber nicht automatisch in Sonar erfasst sind, regelmäßig manuell erfasst werden. Sonar bietet eine Möglichkeit, solche manuellen Metriken in das interne Modell einfließen zu lassen. Die umfassende Bewertung sollte zuletzt von einem versierten Anwender vorgenommen werden, der die Ergebnisse entsprechend interpretiert und an übergeordnete Abteilungen und Personen kommuniziert, wenn nötig.

Es lässt sich zusammenfassen, dass der aktuelle Entwicklungsstand des untersuchten Systems eine eher durchschnittliche Gesamtqualität erreicht. Die Testbarkeit ist deutlich ausbaufähig. Für eine aussagekräftige Produktivitätsbewertung liegen nicht genügend Daten vor. Die Entwicklung der Anzahl abgearbeiteten und abgenommenen Tickets lässt auf eine hohe Produktivität schließen. Die bloße Anzahl Tickets ist zumindest ein Indiz für eine gestiegene Dokumentationstätigkeit, da es sich vielleicht um jeweils kleinere Aufgaben handeln könnte.

Um diesen Teufelskreis der unzureichend aufgenommenen Anforderungen, wie im Kapitel 3 dargestellt, durchbrechen zu können, sollte das Personal, dass die Consultingaufgaben wahrnimmt, regelmäßig in Methoden des Requirements Engineering geschult werden. Diese Investition wird dazu führen, dass die Kunden zufriedener mit der Projektbetreuung werden, da sie sich einfach besser verstanden fühlen. Weiterer Nutzen kann daraus gezogen werden, dass dann auch die Konzepte, die aus diesen Anforderungen entstehen, mitunter detaillierter gestaltet werden und die genannten Probleme damit minimiert werden.

9 Bewertungsergebnisse

Nachdem die Continuous Inspectation mit Sonar für mehrere Monate wöchentlich durchgeführt wurde, sind bereits erste Trends zu erkennen. Die Einhaltung der Programmierrichtlinien hat sich seit Beginn der Messung stark verbessert. Der Verlauf der Entwicklung ist in Abbildung 9.1 dargestellt. Die Zahl der Richtlininenverletzungen ging von 56727 auf 44425 zurück.

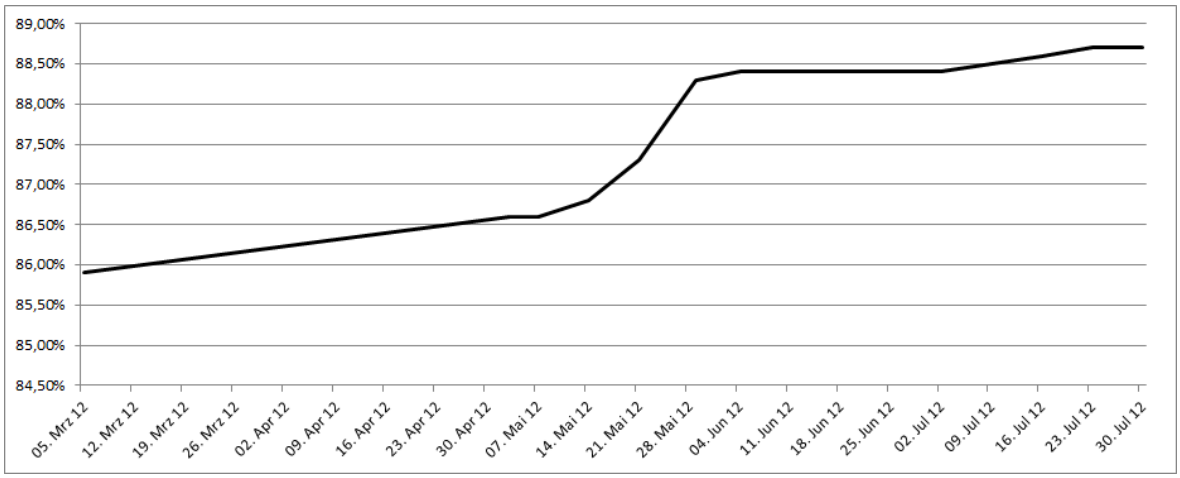


Abbildung 9.1: Entwicklung der Richtlinieneinhaltung

Während die Richtlinien umgesetzt wurden, wuchs die Codebasis von $1.074.450$ Zeilen Anfang März 2012 bis auf $1.131.761$ zum Ende Juli. Der Anteil der Kommentare erhöhte sich in diesem Zeitraum nur marginal von $25,4\%$ auf $25,7\%$.

Die durchschnittliche zyklomatische Komplexität pro Methode liegt über alle Programmteile verteilt unverändert bei $5,7$. Dies entspricht einer angemessenen Komplexität. Es existieren, vornehmlich im Bereich der Schnittstellen zu anderen Systemen, siebzehn Pakete, deren durchschnittliche Komplexität pro Methode bei über 10 liegt. In dieser Gruppe liegt der Anteil der Kommentare bei unterdurchschnittlichen $12,36\%$. Diese mangelnde Dokumentation erhöht den Aufwand bei der Beseitigung der hohen Komplexität enorm.

Bei der Betrachtung der Martin Distanz für die einzelnen Teile des Softwaresystems fiel auf, dass die Abstractness (A) als abhängige Größe bei fasst allen Paketen einen Wert von 0 hat. Dies ist auf die Architektur des betrachteten Systems zurückzuführen, die eine sehr flache Vererbungshirachie mit nur sehr wenigen abstrakten Klassen beinhaltet. Auf einem Abstractness-Instability Diagramm, wie in Abbildung 4.2 in Kaptitel 4.3 dargestellt, befänden sich fast alle Pakete auf der X-Achse. Eine Martin Distanz ist berechenbar, sie führte aber zu annähernd gleichen Werten für fast alle Pakete innerhalb des Systems, da viele Pakete eine ähnliche Anzahl Kopplungen aufweisen.

Die Lack of Cohesion of Methods Version 4 (LCOM4) Metrik aus der CK-Suite stellt zusammen mit der zyklomatischen Komplexität einen Ansatzpunkt dar, wo mit den Refactoringmaßnahmen begonnen werden sollte. Fast alle Klassen erreichen den für LCOM4 besten erreichbaren Wert von 1. Bei einigen der Klassen deren Werte mit > 1 angegeben werden, handelt es sich um Rechenfehler. Diese entstehen durch die Berechnung auf Grundlage des Java Bytecodes, der während des Build-Prozesses noch angereichert wird. Da im Zuge einer Architekturänderung des betrachteten Projektes die Anreicherung entfällt, werden die falschen Messergebnisse an absehbarer Zeit entfallen.

Die in der Codebasis enthaltene technische Schuld konnte im Betrachtungszeitraum nicht signifikant gesenkt werden. Durch die Fokussierung auf die Beseitigung niedrig priorisierter, aber leicht zu lösender, Richtlinienverletzungen wurde weniger Schuld abgebaut, als sich im gleichen Zeitraum durch unachtsame Weiterentwicklung aufbaute. Durch die bereits erwähnte Architekturänderung könnten auch einige Entwicklungsaufgaben zur Senkung der Schuld eingeplant und durchgeführt werden. Das Ziel für diese Metrik sollte eine Absenkung

des Wertes sein. Nachdem ein noch festzulegender Wert erreicht wurde, sollte dieser sich möglichst nicht wieder merklich erhöhen.

Einige Werte für Metriken und ihre Entwicklung im Betrachtungszeitraum sind im Anhang in Kapitel A.2 in Abbildungen A.2 bis A.7 dargestellt. Die Auswahl der dargestellten Metriken umfasst nur Metriken deren Werte gut darstellbare Entwicklungen aufwiesen.

Die Gesamtqualität des betrachteten Softwareprojektes ist durchschnittlich. Die Wartungsintensität der sehr großen Codebasis überstiegt die Innovationskraft noch nicht. Dieser Fakt ließe sich auch ohne Qualitätsmessung feststellen. Die Messung offenbart jedoch, welche Artefakte eine hohe Wartungsintensität erwarten lassen. Eine Verbesserung der Wartbarkeit durch Senkung der Komplexität, sowie Auflösung der Kopplung zwischen einzelnen Programmteilen scheint ratsam.

10 Fazit

Der Zweck dieser Arbeit war es heraus zu finden, in wie fern sich die Qualität von Software mit Hilfe von Metriken bestimmen lässt. Die Frage, welche Metriken relevant sind, um diese Bestimmung vorzunehmen, ist von Entwicklungsprojekt zu Entwicklungsprojekt verschieden.

Im Zuge der Erarbeitung entstand ein Qualitätsmodell, dass den Anforderungen im betrachteten Projekt gerecht wird. Die meisten nötigen Daten werden, wie gefordert, automatisiert und ohne weitere Eingriffe in die bestehende Umgebung erhoben. Das für die Interpretation nötige Know How entwickelt sich bei den Entwicklern langsam. Das Bewusstsein des ganzen Entwicklungsteams für nachhaltig gute Codequalität konnte entschieden gestärkt werden.

Im Kontext von CMMI konnte das Maturity Level „2 - Managed“ noch nicht erreicht werden, da das Anforderungsmanagement noch nicht vollständig etabliert ist. Um in diesem Prozessgebiet das nötige Capability Level „2 - Managed“ zu erreichen, müssen die Prozesse zur Änderung und Nachverfolgung von Anforderungen noch definiert und etabliert werden. Eine regelmäßige wechselseitige Anpassung zwischen Projektplanung und Anforderungen findet nur selten statt.

Prozessgebiet	**Capability Level**	
	vorher	nachher
CM	1	2
MA	1	2
PMC	1	2
PP	2	2
PPQA	1	1
REQM	0	0
SAM	0	0

Tabelle 10.1: Capability Levels nach der bisherigen Entwicklung

Das Maturity Level 2 sieht für insgesamt 7 Prozessgebiete das Capability Level 2 vor. Im Rahmen dieser Arbeit wurden die Bestrebungen im Prozessgebiet „Measurement and Analysis (MA)“ massiv ausgebaut und für diesen Prozess das Capability Level 2 erreicht. Die Tabelle 10.1 spiegelt die erreichten Capability Level wider.

Es bleibt festzuhalten, dass es viele Softwaremetriken gibt, die geeignet sind, um die Qualität von Software zu messen. Es muss jedoch ein Qualitätsmodell existieren, dass den Metriken eine Gewichtung zuordnet und für deren Wertebereich im Kontext der Erhebung eine Interpretation geliefert wird. Das Qualitätsmodell muss so viele Metriken enthalten, dass alle gewünschten Informationen erhoben werden können.

Um ein Qualitätsmodell aufzustellen eignet sich die GQM Methode. Mit ihrer Hilfe lässt sich nicht nur das Set der zu verwendenden Metriken ermitteln. Sie bietet weiterhin die Möglichkeit auch den Bedarf für neue Metriken aufzuzeigen, damit diese zum Beispiel auf Basis vorhandener Metriken definiert und angewandt werden können.

Die Erarbeitung des Modells unter Einbeziehung der beteiligten Stakeholder führte zu einer großen Akzeptanz der folgenden Erhebungen. Ein weiterer Nebeneffekt war die Tendenz zur selbstständigen Qualitätskontrolle der eigenen Arbeit. Die Disziplin, die bereits vorhandenen Regeln umzusetzen, nahm ebenfalls zu.

Eine weitestgehend automatisierte Erhebung ersetzt keine manuelle Interpretation der erhobenen Metriken. Die Erhebung konnte mit geringem Aufwand in die bestehenden Entwicklungsprozesse eingebettet werden. Die Interpretation wird jeweils während der Reviews vorgenommen. Die Reviews sind aber noch nicht im gleichen Maße institutionalisiert. Sie werden nur selten durchgeführt, wenn es im Verlauf eines Releasezyklus zu starken Verzögerungen gekommen ist.

Die Qualitätsanalyse eines schon laufenden Entwicklungsprojekts ist sinnvoll, wenn das Produkt noch nicht die letzten Phasen seines Lebenszyklus angekommen ist. Nicht zuletzt, weil selbst eine handwerklich unausgereifte Analyse besser ist als keine Analyse. Der Sprung von

subjektiven Empfinden hin zu objektiver Betrachtung stellt bereits eine Qualitätsverbesserung dar. Die Analyse und das Qualitätsmodell entwickeln sich durch beständige regelmäßige Anpassungen an den gewünschten Erkenntnisgewinn weiter. Dabei wird der Aufwand auf einem vertretbarem Niveau gehalten. Es muss jedoch ein Verantwortlicher benannt werden, der als Ansprechpartner für die Entwickler zur Verfügung steht. Zusätzlich dazu sollte dieser Posten gleichzeitig die Beobachtung der Daten und die Steuerung von Verbesserungsmaßnahmen übernehmen.

Die Maßnahmen, die in der betrachteten Entwicklungsabteilung umgesetzt wurden, lassen sich ohne größere Probleme auch in anderen Abteilungen des Unternehmens etablieren. Durch die gemeinsame Verwendung von Ressourcen ließen sich selbige dann wirkungsvoll einsparen. Die Voraussetzungen sind das Vorhandensein von Quellcode, der in einer Sprache vorliegt, die von der gewählten Continuous Inspectation Lösung Sonar unterstützt wird.

Eine Anwendung der CMMI wäre für die gesamte Firma nur dann denkbar, wenn eine kleine Verlagerung der Arbeitskraft weg von der Entwicklung hin zum Management möglich ist. Eine strukturierte, an die Vorgaben eines Rahmenwerks angepasste Arbeitsweise, erfordert mehr Verwaltungsaufwand und Disziplin als eine ohne institutionalisierte Prozesse. Die Vorgaben dazu muss die Geschäftsführung nach einer ausgiebigen Kosten-Nutzen-Analyse aufstellen und durchsetzen.

Als leicht zu interpretierende Metriken haben sich die Größenmetriken heraus kristallisiert. Sie lassen allerdings keine Aussage zur Qualität zu. Relevanter erschien die WMC-Metrik, die potentiell schlecht wartbare Klassen offenbart und so Hinweise zu nötigen Refactoringaufgaben gibt. Zusammen mit der LCOM4 Metrik wird die Priorisierung noch deutlicher. Die Martin-Distanz ist als Indikator für Wartbarkeit im betrachteten Projekt nicht aussagekräftig genug, um angewandt zu werden. Die Metriken der CK-Suite sind leicht interpretierbar aber nicht relevant für die betrachtete Software, da die objektorientierten Konzepte nicht sehr ausgiebig zum Einsatz kommen. Die Halstead Metriken werden nicht in Betracht gezogen, da ihre Aussagekraft nicht verifiziert werden konnte und Sonar sie nicht berechnen kann.

Die Richtschnur bei der Bestimmung der Softwarequalität im betrachteten Projekt bildet die Einhaltung der Programmierrichtlinien. Da bereits ein Regelwerk existierte, konnte dies ausgeweitet und zur automatischen Erhebung genutzt werden. Zukünftig wird es regelmäßige Anpassungen der Richtlinien geben. Einige Indikatoren wurden für die Bestimmung definiert und werden in Ermangelung automatischer Tools manuell erhoben.

Eine untergeordnete Rolle spielte im betrachteten Projekt die Testbarkeit, die ein großes Verbesserungspotential aufweist. Die Testabdeckung der wenigen vorhandenen Unittests liegt, über die gesamte Codebasis ermittelt, bei weniger als $0,15\%$. Sie ist auf vergangene Managemententscheidungen zurückzuführen, die auf eine flächendeckende Testentwicklung zu Gunsten höherer Innovationsdichte verzichteten.

Das Qualitätsbewusstsein, das die Entwickler ausgebildet haben, legitimiert den Aufwand der nötig war um die Messungen einzuführen. Der neu entwickelte Code enthält im Vergleich zur bisherigen Codebasis deutlich weniger Richtlinienverletzungen. Einige Entwickler begannen bereits mit der Aufteilung besonders komplexer Artefakte. Die zukünftige Planung von Code Reviews wird den Wartungsaufwand senken und so Ressourcen freistellen, die für eine schnellere Weiterentwicklung nutzbar sind. Die konsequentere Nutzung von Retrospektiven im Nachgang von Entwicklungszyklen sorgt für eine beständige Prozessverbesserung durch Steuerungseffekte innerhalb des Entwicklungsteams.

Sollte sich zukünftig die Dokumentationsqualität verbessern, liegt eine Grundlage für die Entwicklung umfassender Tests vor. Im Allgemeinen sollte die Testabdeckung eine weitaus wichtigere Rolle spielen, da hier bereits ein Großteil der Fehler entdeckt werden können. Im Laufe der Weiterentwicklung einer Software können Regressionen auftreten, die sofort zu fehlgeschlagenen Tests führen.

11 Ausblick

Nachdem eine Qualitätsmessung und deren regelmäßige Anpassung zu einem andauernden Prozess erhoben wurden, sollten Managemententscheidungen folgen, inwiefern eine Verbesserung im Sinne von CMMI erreicht werden soll und was die nächsten Schritte dahin sind. Um eine bessere Produktqualität erreichen zu können, sollte eine Verbesserung im Bereich des Anforderungsmanagements angestrebt werden. Ein sorgfältiges Anforderungsmanagement ist der Grundpfeiler einer guten Dokumentation, die wiederum ein Grundstein von hoher Qualität darstellt. Zu diesem Zweck sollte es auch Standardprozesse geben, sowie geschultes Personal, das die Anforderungen gewissenhaft erfasst.

Um die Entwickler stärker an den erhobenen Daten partizipieren zu lassen, wäre eine tiefere Integration zwischen den einzelnen unterstützenden Tools ratsam. Es existieren in vielen Tools Schnittstellen zu anderen Systemen. Diese Schnittstellen können zur Grundlage neuer Integrationskomponenten werden. Eine für das aufgestellte Qualitätsmodell förderliche Verbindung sollte zwischen dem Projektmanagementtool Redmine und dem Continuous Inspectation System Sonar entstehen. Redmine sollte dabei Daten bereitstellen, die später in die Bewertung innerhalb der Sonar Umgebung einbezogen werden. In die andere Richtung wäre eine Darstellung eines kleinen Dashboard mit Daten aus Sonar innerhalb von Redmine denkbar.

Der erste Schritt könnte die automatische Erfassung der Ticketzahlen sein, die momentan noch manuell eingebracht werden. Der zweite Schritt wäre eine Einbeziehung der verbuchten Bearbeitungszeit und ein Vergleich mit den geplanten Zeiten und der geplanten Kapazität.

Der Informationsgewinn, der daraus entsteht, kann für die Planung zukünftiger Kundenprojekte von essenziellem Nutzen sein.

Um den Spieltrieb der Entwickler besser auszureizen, empfiehlt sich eine Weiterentwicklung des verwendeten Jenkins Plugins, sodass jeder Entwickler einzeln bewertet und diese Bewertung für andere Arten der Motivation herangezogen werden kann.

Eine funktionierende Continuous Inspectation Infrastruktur stellt die Basis dar, auf der später eine CD Lösung aufgebaut werden kann. Diese Entwicklung kann Teil einer weitergehenden Professionalisierungsstrategie werden, deren Anfang mit der Messung der Softwarequalität beschritten wurde.

A Anhang

A.1 Entwicklungsstatistiken

Die folgenden Ticketzahlen spiegeln den Stand zur Fertigstellung eines Release wider. In der aktuell entwickelten Version 4.2 werden die offenen Tickets zusätzlich betrachtet. Geprüfte Tickets werden in der Regel nicht nachträglich abgewiesen, sodass alle geprüften Tickets nach Abschluss der Entwicklung und Tests zu abgenommenen Tickets werden.

Version	**Tracker**	**Status**	**Anzahl**
3.1.1	Projekt	Abgenommen	2
3.1.1	Anforderung	Abgenommen	1
3.1.1	Fehler	Abgenommen	37
3.1.1	Fehler	Abgewiesen	1
3.1.1	Umsetzung	Abgenommen	94
3.1.1	Umsetzung	Abgewiesen	2
3.1.1	Konzept	Abgenommen	5
3.1.1	**Summe**	**gesamt**	142
3.1.2	Anforderung	Abgewiesen	1
3.1.2	Anforderung	Abgenommen	4
3.1.2	Umsetzung	Abgenommen	63
3.1.2	Umsetzung	Abgewiesen	1
3.1.2	Umsetzung	Zugewiesen	3
3.1.2	Fehler	Abgenommen	18

Version	Tracker	Status	Anzahl
3.1.2	Konzept	Abgenommen	1
3.1.2	**Summe**	**gesamt**	88
3.1.3	Anforderung	Abgenommen	3
3.1.3	Fehler	Abgenommen	18
3.1.3	Umsetzung	Abgenommen	50
3.1.3	Konzept	Abgenommen	2
3.1.3	Projekt	Abgenommen	1
3.1.3	**Summe**	**gesamt**	74
3.2	Konzept	Abgenommen	4
3.2	Anforderung	Abgenommen	2
3.2	Umsetzung	Abgenommen	150
3.2	Umsetzung	Zugewiesen	3
3.2	Umsetzung	Abgewiesen	4
3.2	Fehler	Abgenommen	96
3.2	Projekt	Abgenommen	2
3.2	**Summe**	**gesamt**	258
3.2.1	Anforderung	Abgenommen	2
3.2.1	Umsetzung	Abgenommen	38
3.2.1	Fehler	Abgenommen	12
3.2.1	**Summe**	**gesamt**	52
3.2.2	Anforderung	Abgenommen	6
3.2.2	Konzept	Abgenommen	4
3.2.2	Umsetzung	Abgenommen	49
3.2.2	Umsetzung	Abgewiesen	2
3.2.2	Fehler	Abgenommen	9
3.2.2	Projekt	Abgenommen	1
3.2.2	**Summe**	**gesamt**	71
3.2.3	Anforderung	Abgenommen	6
3.2.3	Umsetzung	Abgewiesen	1

Version	Tracker	Status	Anzahl
3.2.3	Umsetzung	Abgenommen	37
3.2.3	Konzept	Abgenommen	1
3.2.3	Fehler	Abgenommen	71
3.2.3	Fehler	Abgewiesen	3
3.2.3	Projekt	Abgenommen	4
3.2.3	**Summe**	**gesamt**	123
4.0	Anforderung	Abgenommen	14
4.0	Umsetzung	Abgenommen	92
4.0	Umsetzung	Abgewiesen	1
4.0	Fehler	Abgenommen	103
4.0	Fehler	Abgewiesen	2
4.0	**Summe**	**gesamt**	212
4.1	Projekt	Abgenommen	7
4.1	Umsetzung	Abgenommen	131
4.1	Umsetzung	Abgewiesen	3
4.1	Fehler	Abgenommen	81
4.1	Fehler	Abgewiesen	4
4.1	**Summe**	**gesamt**	226
4.1.1	Konzept	Abgenommen	4
4.1.1	Fehler	Abgenommen	93
4.1.1	Fehler	Abgewiesen	5
4.1.1	Umsetzung	Abgenommen	180
4.1.1	Umsetzung	Abgewiesen	4
4.1.1	**Summe**	**gesamt**	286
4.1.2	Anforderung	Abgenommen	29
4.1.2	Fehler	Abgenommen	207
4.1.2	Fehler	Abgewiesen	7
4.1.2	Konzept	Abgenommen	17
4.1.2	Konzept	Abgewiesen	1

Version	Tracker	Status	Anzahl
4.1.2	Umsetzung	Abgenommen	183
4.1.2	Umsetzung	Abgewiesen	15
4.1.2	**Summe**	**gesamt**	459
4.2	Anforderung	Abgenommen	41
4.2	Anforderung	Abgewiesen	12
4.2	Konzept	Abgenommen	41
4.2	Konzept	Abgewiesen	1
4.2	Umsetzung	Abgenommen	346
4.2	Umsetzung	Abgewiesen	9
4.2	Fehler	Abgenommen	370
4.2	Fehler	Abgewiesen	20
4.2	**Summe**	**gesamt**	840

Tabelle A.1: Erfasste Tickets nach Version

Der Tabelle A.1 ist zu entnehmen, dass sich der Grad der Dokumentation innerhalb des Projektmanagementtools im Laufe der Zeit immer mehr gesteigert hat. Die Abbildung A.1 zeigt, in welcher Frequenz die Tickets im Durchschnitt pro Release angelegt und abgearbeitet werden.

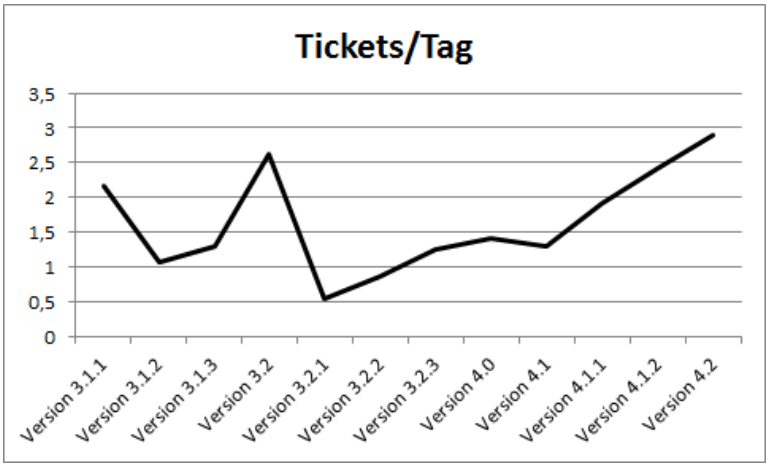


Abbildung A.1: Abgeschlossene Tickets pro Tag nach Version

A.2 Entwicklung einzelner Metriken

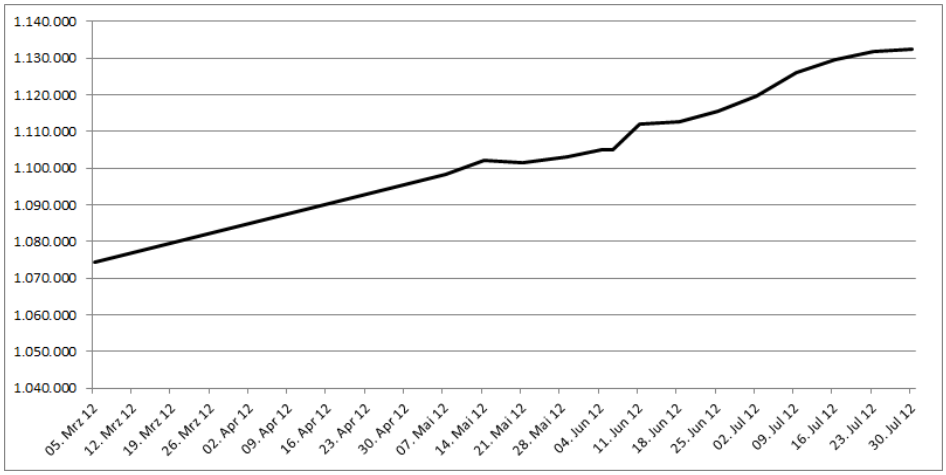


Abbildung A.2: Entwicklung der Codezeilen

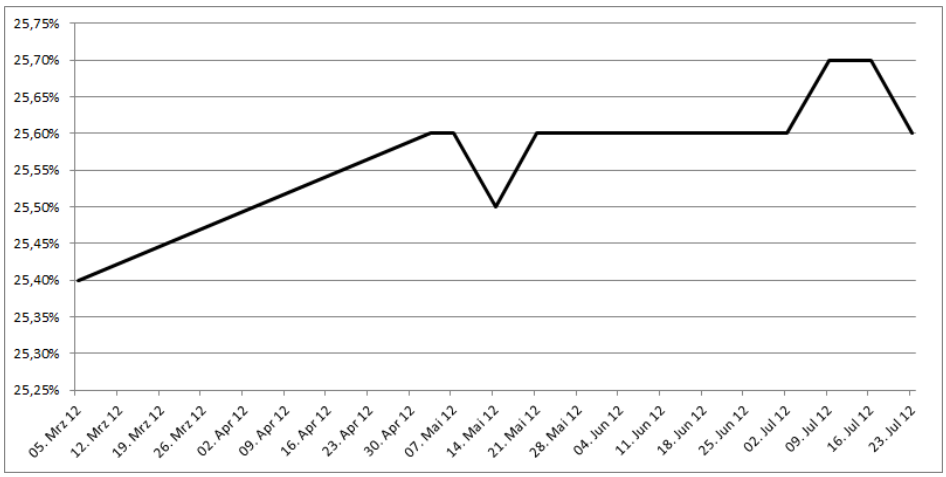


Abbildung A.3: Entwicklung des Kommentaranteils

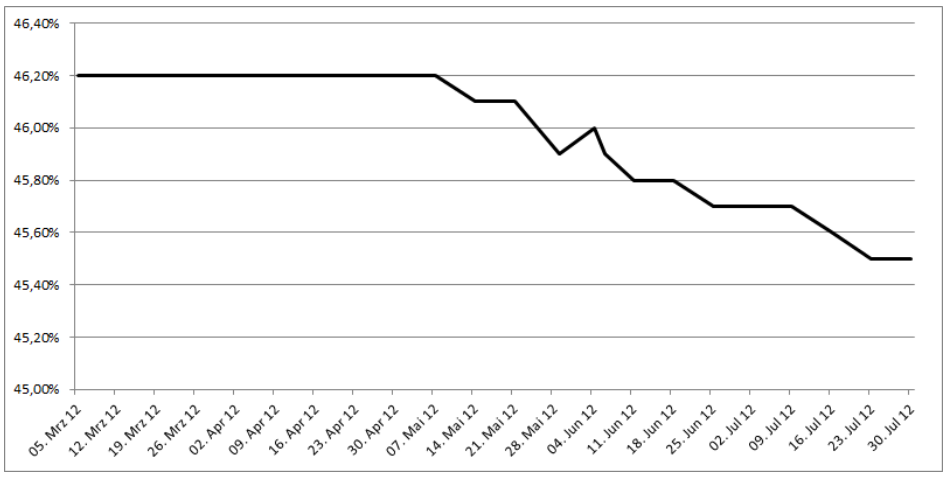


Abbildung A.4: Entwicklung des duplizierten Codeanteils

Der Anteil des duplizierten Codes beinhaltet auch den generierten Codeanteil, der etwa die Hälfte der Duplikate ausmacht.

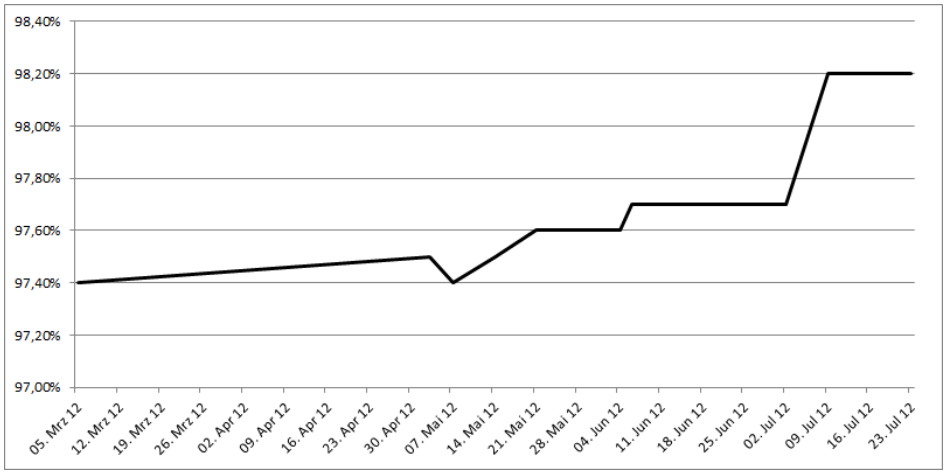


Abbildung A.5: Entwicklung der dokumentierten Public API

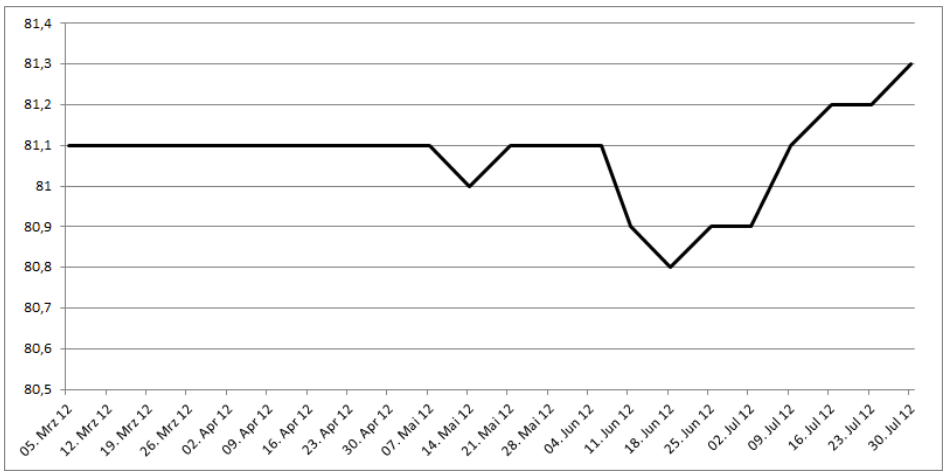


Abbildung A.6: Entwicklung der durchschnittlichen Komplexität pro Klasse

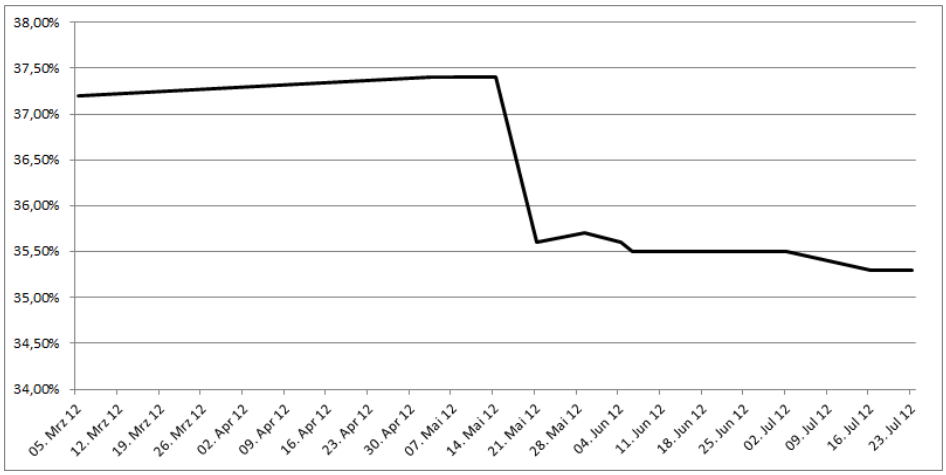


Abbildung A.7: Entwicklung der technischen Schuld

Dieses Diagramm spiegelt den Anteil der technischen Schuld bezogen auf die Entwicklungskosten wider.

A.3 CD-Inhalt

- thesis.pdf (PDF Version der Masterarbeit)
- verteidigung.ppsx (Powerpoint Präsentation der Verteidigung)
- /literatur (PDF Dokumente, die im Literaturverzeichnis genannt wurden und öffentlich zugänglich sind)

Literaturverzeichnis

[AAJE11] Alshayeb, Mohammad ; Al-Jamimi, Hamdi ; Elish, Mahmoud O.: Empirical taxonomy of refactoring methods for aspect-oriented programming. In: *Journal of Software Maintenance and Evolution: Research and Practice* (2011). http://dx.doi.org/10.1002/smr.544. – DOI 10.1002/smr.544. – ISSN 1532–0618

[AKP02] Assmann, Danilo ; Kalmar, Ralf ; Punter, Dr. T.: *Messen und Bewerten von Webapplikationen mit der Goal/Question/Metric Methode*. 2002

[AM96] Abreu, F. Brito e. ; Melo, W.: Evaluating the impact of object-oriented design on software quality. In: *Software Metrics Symposium, 1996., Proceedings of the 3rd International* IEEE, 1996, S. 90–99

[AP10] Ayewah, Nathaniel ; Pugh, William: The google findbugs fixit. In: *Proceedings of the 19th international symposium on Software testing and analysis* (2010), 241–252. http://www.cs.umd.edu/~ayewah/web/pubs/Google-ISSTA2010.pdf

[Bal97] Balzert, Helmut: *Lehrbuch der Software-Technik, Bd. 2: Software-Management, Software-Qualitätssicherung, Unternehmensmodellierung*. Spektrum Akademischer Verlag, 1997. – ISBN 9783827400659

[Bei12] Beine, Gerrit: *Vortrag: Die Macht der Zahlen*. http://goo.gl/diKW5. Version: 2012 (Chemnitzer Linuxtage)

[Boe78] Boehm, Barry W.: *Characteristics of Software Quality (TRW series of software technology)*. Elsevier Science Ltd, 1978. – ISBN 9780444851055

[CK94] Chidamber, Shyam R. ; Kemerer, Chris F.: A Metrics Suite for Object Oriented Design. In: *IEEE Transactions on Software Engineering* 20 (1994), Juni, S. 476–493

[CMM10] CMMI Product Team: *CMMI® for Development*. 1.3. Software Engineerung Institute, Carnegie Mellon University, 2010 http://www.sei.cmu.edu/reports/10tr033.pdf

[Cov12] Coverity Inc: *Coverity Scan: 2011 Open Source Integrity Report*. http://goo.gl/SifFq. Version: 2012

[DABCH11] Ducasse, Stéphane ; Anquetil, Nicolas ; Bhatti, Muhammad U. ; Cavalcante-Hora, Andre: Software Metrics for Package Remodularisation. Version: November 2011. http://hal.inria.fr/hal-00646878. 2011 (hal-00646878). – Forschungsbericht. – RMOD - INRIA Lille - Nord Europe

[DeM86] DeMarco, Tom: *Controlling Software Projects: Management, Measurement, and Estimates*. Prentice Hall, 1986. – ISBN 9780131717111

[DSKG09] Dumke, Reiner R. ; Schmietendorf, Andreas ; Kunz, Martin ; Gorgieva, Konstantina: *Software Metrics for Agile Software Development*. März 2009

[EARAK12] Elish, Mahmoud O. ; Al-Rahman Al-Khiaty, Mojeeb: A suite of metrics for quantifying historical changes to predict future change-prone classes in object-oriented software. In: *Journal of Software: Evolution and Process* (2012). http://dx.doi.org/10.1002/smr.1549. – DOI 10.1002/smr.1549. – ISSN 2047–7481

[Ebe12] Ebert, Christof: *Systematisches Requirements Engineering*. Dpunkt.Verlag GmbH, 2012. – ISBN 9783898648127

[ED07] Ebert, Christof ; Dumke, Reiner: *Software measurement - establish, extract, evaluate, execute*. 1. Aufl. Springer, 2007. – ISBN 9783540716483

[Fal10] Falcone, Giovanny: *Hierarchy-Aware Software Metrics in Component Composition Hierarchies*. Logos Verlag Berlin GmbH, 2010. – ISBN 9783832525688

[Gil77] Gilb, Tom: *Software metrics*. Winthrop Publishers, 1977. – ISBN 9780876268551

[GK09] Grabski, B. ; Krüger, L.: Analysen zu Qualität und Qualitätsmanagement von Software und Dienstleistungen / Technical Report, Very Large Business Applications Lab, Fakultät für Informatik, Otto-von-Guericke-Universität Magdeburg. Version: 2009. `http://goo.gl/WMaqP`. 2009 (FIN-015-2009). – Forschungsbericht. – Very Large Business Applications Lab

[HKV07] Heitlager, Ilja ; Kuipers, Tobias ; Visser, Joost: A Practical Model for Measuring Maintainability / Software Improvement Group. 2007 (TUD-SERG-2010-006). – Forschungsbericht. – Software Engineering Research Group

[Hof08] Hoffmann, Dirk W.: *Software-Qualität*. Springer, 2008. – ISBN 9783540763228

[Hum11] Hummel, Oliver: Software messbar machen. Version: 2011. `http://dx.doi.org/10.1007/978-3-8274-2752-6_3`. In: *Aufwandsschätzungen in der Software- und Systementwicklung kompakt*. Spektrum Akademischer Verlag, 2011 (IT kompakt). – ISBN 978–3–8274–2751–9, 35-67

[HW91] Henry, Sallie ; Wake, Steve: Predicting maintainability with software quality metrics. In: *Journal of Software Maintenance: Research and Practice* 3 (1991), Nr. 3, 129–143. http://dx.doi.org/10.1002/smr.4360030302. – DOI 10.1002/smr.4360030302. – ISSN 1096–908X

[ISO91] Norm ISO/IEC 9126 Dezember 1991. *Informationstechnik - Bewertung von Software-Produkten - Qualitätsmerkmale und Richtlinien für deren Andwendung*

[ISO11] Norm ISO/IEC 25010 März 2011. *Software Engineering - Qualitätskriterien und Bewertung von Softwareprodukten (SQuaRE) - Qualitätsmodell und Leitlinien*

[KLST05] Koskinen, Jussi ; Lintinen, Heikki ; Sivula, Henna ; Tilus, Tero: Evaluation of Software Evolution Options. In: *Publications of the Information Technology Research Institute* 14/2004 (2005). http://dogbert.mse.cs.cmu.edu/mse2009/projects/imhotep/Requirements/Case%20Studies/Evaluation_of_Software_Evolution_Options_(Koskinen,_et_al)).pdf

[KZMD08] Kunz, Martin ; Zenker, Niko ; Mencke, Steffen ; Dumke, Reiner: *Unit Metrics - A Tool to support Refactoring in Agile Software Development*. Mai 2008

[Lan08] Lanza, Michele: *Of Code and Change: Beautiful Software*. http://goo.gl/KU89Q. Version: 2008 (JAOO Conference)

[Let12] Letouzey, Jean-Louis: *The SQALE Method Definition Document*. 1.0. DNV ITGS France, 2012 http://www.sqale.org/wp-content/uploads/2010/08/SQALE-Method-EN-V1-0.pdf

[Lig09] Liggesmeyer, Peter: *Software-Qualität: Testen, Analysieren und Verifizieren von Software*. Spektrum Akademischer Verlag, 2009. – ISBN

9783827420563

[LK94] Lorenz, Mark ; Kidd, Jeff: *Object-oriented software metrics : a practical guide*. Englewood Cliffs, NJ : PTR Prentice Hall, 1994. – ISBN 9780131792920

[LM06] Lanza, Michele ; Marinescu, Radu: *Object-Oriented Metrics in Practice: Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems*. Springer, 2006. – ISBN 9783540244298

[LV10] Luijten, Bart ; Visser, Joost: Faster Defect Resolution with Higher Technical Quality of Software / Delft University of Technology. Version: 2010. `http://swerl.tudelft.nl/twiki/pub/Main/TechnicalReports/TUD-SERG-2010-006.pdf`. 2010. – Forschungsbericht

[Mar94] Martin, Robert: *OO Design Quality Metrics An Analysis of Dependencies*. Oktober 1994

[Mar02] Martin, Robert: *Agile Software Development: Principles, Patterns, and Practices*. Prentice Hall, 2002. – ISBN 9780135974445

[McC76] McCabe, Thomas J.: A complexity Measure. In: *IEEE Transactions on Software Engineering* SE-2 (1976), December, S. 308–320

[MMMW05] Marinescu, C. ; Marinescu, Radu ; Mihancea, P. F. ; Wettel, R.: iplasma: An integrated platform for quality assessment of object-oriented design. In: *In ICSM (Industrial and Tool Volume)* (2005), S. 77–80

[MRW77a] MCCALL, Jim A. ; RICHARDS, Paul K. ; WALTERS, Gene F.: *Factors in software quality. Volume 1: Concepts and definitions of software quality*. 1977

[MRW77b] MCCALL, Jim A. ; RICHARDS, Paul K. ; WALTERS, Gene F.: *Factors in software quality. Volume 2: Metric data collection and validation*. 1977

[OEMD08] OLAGUE, Hector M. ; ETZKORN, Letha H. ; MESSIMER, Sherri L. ; DELUGACH, Harry S.: An empirical validation of object-oriented class complexity metrics and their ability to predict error-prone classes in highly iterative, or agile, software: a case study. In: *Journal of Software Maintenance and Evolution: Research and Practice* 20 (2008), Nr. 3, 171–197. http://dx.doi.org/10.1002/smr.366. – DOI 10.1002/smr.366. – ISSN 1532–0618

[OW11] ORAM, Andy ; WILSON, Greg: *Making Software : what really works, and why we believe it*. 1. ed. O'Reilly, 2011. – ISBN 9780596808327

[SB99] SOLINGEN, Rini van ; BERGHOUT, Egon: *The Goal/Question/Metric method: a practical guide for quality improvement of software development*. McGraw-Hill, 1999. – ISBN 9780077095536

[SBS08] SNEED, Harry M. ; BAUMGARTNER, Manfred ; SEIDL, Richard: *Der Systemtest - Von den Anforderungen zum Qualitätsnachweis*. 2. aktualisierte und erweiterte Auflage. München : Hanser Verlag, 2008. – ISBN 9783446417087

[Sch10] SCHACKMANN, Holger: *Metrik-basierte Auswertung von Software-Entwicklungsarchiven zur Prozessbewertung*, RWTH Aachen University, Diss., 2010. http://goo.gl/H9iFf

[Sie06] SIEBER, Andrea: *Arbeitsstile in der Softwareentwicklung*, Technische Universität Chemnitz, Diss., 2006

[SLSN10] SHATNAWI, Raed ; LI, Wei ; SWAIN, James ; NEWMAN, Tim: Finding software metrics threshold values using ROC curves. In: *Journal of Software Maintenance and Evolution: Research and Practice* 22 (2010), Nr. 1, 1–16.

http://dx.doi.org/10.1002/smr.404. – DOI 10.1002/smr.404. – ISSN 1532–0618

[SMKN11] Staron, Miroslaw ; Meding, Wilhelm ; Karlsson, GÃ¶ran ; Nilsson, Christer: Developing measurement systems: an industrial case study. In: *Journal of Software Maintenance and Evolution: Research and Practice* 23 (2011), Nr. 2, 89–107. http://dx.doi.org/10.1002/smr.470. – DOI 10.1002/smr.470. – ISSN 1532–0618

[SSB10] Sneed, Harry M. ; Seidl, Richard ; Baumgartner, Manfred: *Software in Zahlen : die Vermessung von Applikationen*. München : Hanser, 2010. – ISBN 9783446421750

[SSH+11] Shaik, Amjan ; Satyanarayana, N. ; Huzaifa, Mohammed ; Shaik, Nazeer ; Naveed, Mohd Z. ; Rao, S. V. A. ; Reddy, C.R.K.: *Investigate the Result of Object Oriented Design Software Metrics on Fault-Proneness in Object Oriented Systems: A Case Study*. Version: 2011. http://www.cisjournal.org/archive/vol2no4/vol2no4_7.pdf

[SSM06] Simon, Frank ; Seng, Olaf ; Mohaupt, Thomas: *Code-Quality-Management / technische Qualität industrieller Softwaresysteme transparent und vergleichbar gemacht*. 1. Aufl. Heidelberg : dpunkt-Verl., 2006. – ISBN 9783898643887

[Str11] Streit, Jonathan: *Wertorientierung - nicht nur für agile Projekte*. 2011 (BITKOM Forum "Agil vs. Industrialisierung - IT-Entwicklung am Scheideweg")

[Tha94] Thaller, Georg E.: *Software-Metriken : einsetzen, bewerten, messen*. Hannover : Heise, 1994. – ISBN 9783882290382

[VFHR10] Vogelsang, Andreas ; Fehnker, Ansgar ; Huuck, Ralf ; Reif, Wolfgang: Software Metrics in Static Program Analysis. Version: 2010. http:

//dx.doi.org/10.1007/978-3-642-16901-4_32. In: DONG, Jin (Hrsg.) ; ZHU, Huibiao (Hrsg.): *Formal Methods and Software Engineering* Bd. 6447. Springer Berlin / Heidelberg, 2010. – ISBN 9783642169007, 485-500

[Wal90] WALLMÜLLER, Ernest: *Software-Qualitätssicherung in der Praxis*. Hanser, 1990. – ISBN 9783446158464

[WBD+10] WAGNER, Stefan ; BROY, Manfred ; DEISSENBÖCK, Florian ; KLÄS, Michael ; LIGGESMEYER, Peter ; MÜNCH, Jürgen ; STREIT, Jonathan: Softwarequalitätsmodelle Praxisempfehlungen und Forschungsagenda. In: *Informatik-Spektrum* 33 (2010), 37-44. http://dx.doi.org/10.1007/s00287-009-0339-4. – ISSN 0170–6012

[WOA97] WELKER, Kurt D. ; OMAN, Paul W. ; ATKINSON, Gerald G.: Development and Application of an Automated Source Code Maintainability Index. In: *Journal of Software Maintenance: Research and Practice* 9 (1997), Nr. 3, S. 127–159. – ISSN 1096–908X

[Zus85] ZUSE, Horst: *Messtheoretische Analyse von statischen Softwarekomplexitätsmassen.*, TU Berlin, Diss., 1985

Glossar

Accessor Als Accessor wird eine meist triviale Methode bezeichnet, deren Aufgabe darin besteht ein internes Datum bereitzustellen oder in der Instanz abzuspeichern. Sie werden auch Getter und Setter genannt.

Anti-Pattern Ein Anti-Pattern ist ein Programmmuster, dass unter Entwicklern als schlechter Code angesehen wird. Er ist entweder fehleranfällig, schwer nachzuvollziehen, beeinträchtigt die Wiederverwendbarkeit des Codes oder all dies zusammen.

Artefakt Als Artefakt ist ein Teil eines Systems zu verstehen, dass in seiner Gesamtheit ein geschlossenes Subsystem darstellen kann. Es handelt sich mindestens um eine Klasse und kann bis zu einer Größe von mehreren Paketen reichen.

Broken Windows Theorie Diese Theorie von James Q. Wilson und George Kelling aus dem Jahre 1982 befasst sich mit der möglichen Verwahrlosung eines Stadtviertels auf Grund eines einzelnen kaputten Fensters. Diese Verwahrlosung lässt sich auch in Softwaresystemen beobachten, wenn die Codebasis eine großen Anzahl an Qualitätsmängeln stecken.

Capability Maturity Model Integration Es handelt sich hierbei um eine Reihe von Referenzmodellen für die Produktentwicklung ähnlich wie ITIL im IT Servicemanagement.

Cronjob Dies ist ein aus unixoiden Betriebssystemen bekanntes Konzept zur Ausführung periodischer Aufgaben auf einem System.

Dashboard Ein Dashboard stellt in einer schnell erfassbaren Form, zum Beispiel einer Webseite, viele Daten dar. Dazu werden beispielsweise Daten aggregiert und die Skalierung vereinfacht.

E4-Messprozess Hierbei handelt es sich um einenen iterativen Prozess in der Softwaremetrie. Der Name leitet sich von den vier Aktionen „Establish“, „Extract“, „Evaluate“ und „Execute“ ab.

Gott-Klasse Dieses Anti-Pattern ist erfüllt, wenn eine Klasse eine große Anzahl Attribute und Methoden besitzt. Sie gibt sich als allwissend zu erkennen und kann in viele Dinge eingreifen.

Hawthorne-Effekt Wenn eine Gruppe von Personen unter Beobachtung steht und davon wissen, ändern sie auf Grund des Wissens ihr Verhalten. Dieser Effekt wurde erstmals in den 1920er Jahren in den Hawthorne Werken in den USA bei Produktivitätsstudien entdeckt.

Integrated Development Environment Diese Software unterstützt Entwickler dabei ihrer Arbeit nachzugehen. Es sorgt mit Funktionen wie Syntax Highlighting dafür, dass man sich im gesamten Softwareprojekt zurechtfindet. Im Unterschied zu reinen Texteditoren gibt es auch noch andere Hilfen, die aktiv die Entwicklung unterstützen.

Regression Unter einer Regression wird in der Softwareentwicklung eine Art Programmierfehler verstanden, der auftritt, wenn sich das Verhalten einer Software im Zuge der Weiterentwicklung in einer Weise verändert, die nicht mehr dem gewünschten Verhalten entspricht.

Requirements Engineering Dieser Begriff umfasst Vorgehensweisen, die dazu führen, dass Anforderungen eines Kunden später auch in einem Produkt umgesetzt werden können. Es geht dabei um eine strukturierte Analyse der Wünsche des Kunden mit dem Ziel, den Entwicklern ein präzises Ziel vorzugeben.

SCRUM Dieses agile Vorgehensmodell geht davon aus, dass große Projekte nicht bis zum Ende planbar sind. Es definiert Rollen, Zeremonien und Artefakte, die zu einem iterativen Entwicklungsprozess führen.

Source Code Management Dieses System sorgt, dafür, dass mehrere Entwickler an einem Projekt arbeiten können und eine Historie über den entwickelten Quellcode existiert. Gebräuchliche frei nutzbare Systeme sind SVN, Git und meist in älteren Projekten Concurrent Versions System (CVS)

Steffen Förster
Relevante Metriken zur Bestimmung von Softwarequalität

Thesen

1. Für die Bestimmung von Softwarequalität sind Metriken ein geeignetes Mittel.

2. Schlechte Codequalität sorgt für Toleranz gegenüber neuem schlechtem Code.

3. Die Auswahl von Softwaremetriken muss dem gewünschten Umfang der Qualitätsaussage entsprechen.

4. Die Werte von Softwaremetriken müssen dem Kontext entsprechend interpretiert werden.

5. Goal Question Metric (GQM) ist eine geeignete Methode, um im Rahmen von Capability Maturity Model Integration (CMMI) ein Qualitätsmodell zu definieren.

6. Die Nutzung von Continuous Integration unterstützt die Nutzung von Softwaremetriken in Rahmen von Continuous Inspectation.

7. Transparente Verfahren und offene Tools verhelfen der Qualitätsmessung zu einer hohen Akzeptanz unter den Entwicklern.

8. Eine institutionalisierte regelmäßige Messung kann die Produktivität eines Entwicklungsteams steigern.

Selbständigkeitserklärung gem. § 21 Absatz 5 MPO

Hiermit versichere ich, Steffen Förster, dass ich die vorliegende Masterarbeit mit dem Titel

Relevante Metriken zur Bestimmung von Softwarequalität

selbständig und ohne fremde Hilfe verfasst und keine anderen als die angegebenen Hilfsmittel benutzt habe. Die Stellen der Arbeit, die dem Wortlaut oder dem Sinne nach anderen Werken entnommen wurden, sind in jedem Fall unter Angabe der Quelle kenntlich gemacht. Die Arbeit ist noch nicht veröffentlicht oder in anderer Form als Prüfungsleistung vorgelegt worden.

Chemnitz, 20.09.2012	.....
Ort, Datum	Unterschrift