# Bachelorarbeit

Thorben Pergande

Qualitätssicherung inkl. Programmierstilkontrolle mittels Maven2

*Fakultät Technik und Informatik*
*Department Informatik*

*Faculty of Engineering and Computer Science*
*Department of Computer Science*

Thorben Pergande

# Qualitätssicherung inkl. Programmierstilkontrolle mittels Maven2

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung

im Studiengang Angewandte Informatik

am Department Informatik

der Fakultät Technik und Informatik

der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer : Prof. Dr. Jörg Raasch

Zweitgutachter : Prof. Dr. Olaf Zukunft

Abgegeben am 7. Februar 2008

**Thorben Pergande**

**Thema der Bachelorarbeit**

Qualitätssicherung inkl. Programmierstilkontrolle mittels Maven2

**Stichworte**

Softwarequalität, Metriken, Programmierstil, Buildmanagement, Maven2, Messung

**Kurzzusammenfassung**

Diese Arbeit befasst sich mit der Durchführung von Softwarequalitätsmessungen durch Einsatz des Buildmanangentwerkzeugs Maven2. Dabei soll herausgearbeitet werden, in wie weit Maven2 sowohl für die Berechnung und Präsentation von Metriken als auch für die Kontrolle des Programmierstils geeignet ist.

**Thorben Pergande**

**Title of the paper**

Quality Assurance including Codestyle checking with Maven2

**Keywords**

Software quality, Metrics, Codestyle, Buildmanagement, Maven2, Measurement

**Abstract**

Inside this report the Measurement of Software quality and Codestyle using the Buildmanagement tool Maven2 is described. The criteria for successful Quality Assurance are both the correct calculation and presentation of Metrics and the automated control of Codestyle.

# Inhaltsverzeichnis


1 Einleitung ..... 1

1.1 Umfeld dieses Projektes ..... 1

1.2 Ziele dieses Projektes ..... 2

2 Grundlagen ..... 3

2.1 Softwarequalität ..... 3

2.1.1 Qualitätsmetriken ..... 3

2.1.2 Programmierstil ..... 11

2.2 Maven2 ..... 13

2.2.1 Lebenszyklen ..... 15

2.2.2 Project Object Model (POM) ..... 16

2.3 Zusammenfassung ..... 17

3 Vision ..... 18

4 Gespräche und Anforderungsanalyse ..... 19

4.1 Gespräche ..... 19

4.2 Anforderungen ..... 20

4.2.1 Kundenanforderungen ..... 22

4.2.2 Fehler ..... 22

4.2.3 Design ..... 23

4.2.4 Stil ..... 33

4.2.5 Dokumentation ..... 33

4.2.6 Management ..... 34

4.2.7 ABC-Analyse ..... 36

5 Umsetzung ..... 37

5.1 Marktlage & Tests ..... 37

5.1.1 CheckStyle ..... 38

5.1.2 PMD ..... 39
5.1.3 JXR ..... 40
5.1.4 SureFire und Cobertura ..... 40
5.1.5 Clover ..... 41
5.1.6 JavaNCSS ..... 41
5.1.7 JDepend ..... 42
5.1.8 FindBugs ..... 43
5.1.9 Fazit der einzelnen Plug-Ins ..... 45
5.2 Aggregation ..... 45
5.2.1 Sonar ..... 46
5.2.2 QAlab ..... 47
5.2.3 Dashboard ..... 48
5.2.4 Bewertung der Aggregation Plug-Ins ..... 50
5.3 Zusammenfassung der Marktanalyse ..... 51
5.4 Vorgehen ..... 51
5.5 konkrete Umsetzung ..... 53
5.5.1 Erste Schritte ( Iteration 1) ..... 53
5.5.2 Multi-POM-Projekt (Iteration 2) ..... 56
5.5.3 Einzelnes Realsoftwareprojekt (Iteration 3) ..... 59
5.5.4 Erster Test auf das Gesamtprojekt (Iteration 4) ..... 62
5.5.5 Integration (Iteration 5) ..... 67
5.6 Evaluation ..... 72
6 Delta Bereinigung ..... 76
6.1 Metriken ..... 76
6.2 Eigner Programmierstil ..... 77
6.3 Integration ..... 78

6.4 Präsentation ..... 80
6.5 Zusammenfassung ..... 80
7 Fazit ..... 81
7.1 Erreichter Stand ..... 81
7.2 Ausblick in die Zukunft ..... 81
8 Lessons Learned ..... 83
9 Persönliche Meinung ..... 84
Literaturverzeichnis ..... 85
Glossar ..... 87
Anhang ..... 88
Abbildungs- und Tabellenverzeichnis ..... 88
Dokumente und Softwareprojekte ..... 89
Versicherung über Selbstständigkeit ..... 90

# 1 Einleitung

Qualitätssicherung inkl. Programmierstilkontrolle mittels Maven2 ist das Thema dieser Arbeit. Dabei soll herausgefunden werden, in wie weit sich das Buildmanagementwerkzeug Maven2 für die Durchführung einer unter wissenschaftlichen Kriterien durchgeführten Qualitätssicherung eignet. Dazu werden in Kapitel 2 zunächst die Grundlangen der Literatur und der eingesetzten Werkzeuge zu diesem Thema behandelt. Anschließend folgen Kapitel zu den Visionen dieses Projekts und der Anforderungsanalyse. Kapitel 5 beschreibt die Erweiterung von Maven2 für den Zweck der Qualitätssicherung. Abschließend werden in Kapitel 6 die Differenzen des erreichten Stands der Realisierung von der Vision dieser Arbeit versucht zu minimieren.

## 1.1 Umfeld dieses Projektes

Das Umfeld dieser Arbeit ist ein Realprojekt, welches bereits mehrere Monate läuft. Die Buildumgebung und das Konfigurationsmanagement sind zwei Werkzeuge des Projekts, die den Erfolg des Softwareprojektes unterstützen. Die Buildumgebung ist in diesem Projekt Maven2. Für einen Entwickler ist Maven2 ein Werkzeug, mit dem vom Testen über das Kompilieren bis hin zur Auslieferung von Software der Erstellungsprozess erleichtert und teilweise automatisiert wird.

Konfigurationsmanagement unterteilt sich in die Versionskontrolle und Releaseverwaltung. Versionskontrolle (SCM) beschreibt einen Mechanismus, der geänderte Quelltexte und andere Projektdokumente bei Veränderung mit einer neuen Versionsnummer versieht. Des Weiteren werden alle Versionsstände gespeichert und dokumentiert. Erweist sich eine Änderung als problematisch, kann auf eine ältere Version zurückgegriffen werden. Releaseverwaltung beschreibt die Bündelung von versionierten Projektobjekten. Die Bestandteile eines Release sind dokumentiert und das Zusammenspiel der einzelnen Komponenten ist getestet. Wenn bei Veränderung eines Bestandteils Probleme auftreten, kann durch die Versionskontrolle der alte, getestete Stand wiederhergestellt werden. All diese Mechanismen sind in den Entwicklungsprozess integriert und sind das Rückgrat eines Softwareprojektes. Dadurch wird ebenfalls die Qualität eines Projektes erhöht bzw. gesichert, da Veränderungen schnell und einfach rückgängig gemacht werden können und ebenfalls eine Dokumentation der einzelnen Schritte vorliegt.

Das Projekt hat einen Größenumfang an Quelltexten von ca. 700 MB. Es sind zurzeit zwölf Projektmitarbeiter beteiligt und als Entwicklungsmethode wird eine agile Softwareentwicklung praktiziert. Das Team verfolgt den Gedanken der Quelltext-Reviews, wobei ein Stück Quelltext von dem Ersteller und einem anderen Entwickler zusammen besprochen werden und evtl. Änderungen oder Verbesserungen eingebaut werden. Es wurde ein gemeinsamer Programmierstil etabliert, welcher Namenskonventionen und einige Regeln für die Erstellung von Quelltexten umfasst. Alle vier bis sechs Wochen werden neue Releases der Software an den Kunden ausgeliefert.

## 1.2 Ziele dieses Projektes

Die bisher genannten Werkzeuge reichen nicht aus, um eine Qualitätssicherung durchzuführen. Die Idee ist es, früher als bei der Versionskontrolle und Releaseverwaltung anzusetzen. Mögliche Probleme, Fehler oder Abweichungen von der Sollarchitektur sollen zeitnah erkannt und behoben werden.
Da Build- und Konfigurationsmanagement bereits in dem betrachteten Realprojekt durchgeführt werden, soll nun herausgefunden werden, in wie weit weitere qualitätssichernde Maßnahmen in diese Werkzeuge integriert werden können.

Ziel dieser Arbeit ist es daher, zu evaluieren, in welchem Umfang Maven2 in Form von Erweiterungen für qualitätssichernde Maßnahmen geeignet ist. Dabei soll eine Qualitätssicherung angewandt und prototypisch realisiert werden, wobei Metriken, Programmierstilkontrolle und Fehleraufdeckung zentrale Bestandteile dieser Qualitätssicherungsmaßnahmen sind. Neben der Erfüllung der Anforderungen des Kunden soll festgestellt werden, welche weiteren in der Literatur diskutierten Maßnahmen umsetzbar sind. Ein weiteres Kriterium ist die automatisierte, regelmäßige Ausführung der Qualitätssicherung. Die Ergebnisse dieser Arbeit sollen dem Entwicklerteam nicht nur zugänglich gemacht werden, vielmehr ist an dieser Stelle eine Schulung für den Umgang mit den neuen qualitätssichernden Maßnahmen anzubieten.

# 2 Grundlagen

Dieses Kapitel befasst sich mit den Theorien und Techniken, die für die Behandlung dieser Arbeit benötigt werden. Dabei geht es zum einen um die Frage nach der Qualität von Software und zum anderen um die Darstellung des eingesetzten Build-Environment Werkzeugs Maven2.

## 2.1 Softwarequalität

Qualität ist nach ISO 9000:2005 wie folgt definiert:

> „Qualität ist die Gesamtheit von Eigenschaften und Merkmalen eines Produktes oder einer Tätigkeit, die sich auf deren Eignung zur Erfüllung gegebener Erfordernisse bezieht.“ (ISO, 2005)

Äußere Qualität beinhaltet als ein Bestandteil die Erfüllung von funktionalen Anforderungen, die vor allem für den Adressaten der Software interessant ist. Funktionale Anforderungen lassen sich beispielsweise durch Testen oder durch Vergleichen mit den Anforderungen des Kunden prüfen.

In der Analyse der inneren Softwarequalität geht es um die Architektur und den Vergleich der Soll- mit der Ist-Architektur eines Systems. Unter der Prämisse, dass die Soll-Architekturmodellierung ein erweiterbares System umfasst, ist somit eines der Ziele zu prüfen, ob die spezifizierte Architektur eingehalten wurde. Verdeckte Fehler und Design-fragen sind aufzudecken und zu klären. Design umfasst in diesem Kontext Maße wie beispielsweise die Kopplung zwischen Modulen oder die Komplexität einer Klasse. Fehler aus nicht funktionaler Sicht stellen eher Stilfragen oder das Vermeiden von „Bugs“ dar. Ein Beispiel für Stilelemente ist der korrekte Einsatz von Methoden oder auch das Vermeiden des Einsatzes von Methoden[1]. Diese Arbeit befasst sich hauptsächlich mit der inneren Qualität, da die äußere Softwarequalität in diesem Projektkontext bereits durch verschiedene Testsysteme verfolgt wird.

### 2.1.1 Qualitätsmetriken

Qualitätsmetriken, oder in diesem Fall Softwaremetriken, bedeuten in ihrem Ursprung „Messung oder Zählung“. Das Wort „Softwaremetrik“ wird oft benutzt und eingesetzt,

[1] SUN empfiehlt z.B., die Pakte von SUN nicht direkt in ein Projekt zu integrieren, da diese häufigen Änderungen unterliegen.

jedoch bezeichnet eine Metrik eher eine Distanz in einem definierten Raum, ein besseres Wort an dieser Stelle wäre „Softwaremaß“, das in der IEEE 1061 (IEEE 1061, 1999) auf folgende Weise definiert ist:

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

#### *2.1.1.1 Anforderungen an eine Metrik*

Eine Metrik stellt die Abbildung eines Bestandteils eines Softwareprojektes auf eine Zahl dar. Softwareprojekte enthalten eine Menge von Bestandteilen, die sich wiederum aus anderen Bestandteilen zusammensetzen. Werkzeuge müssen Metriken auf verschiedenen Ebenen eines Softwareprojektes, also Subsystemen, Paketen oder einzelnen Klassen ermitteln können. Im Folgenden wird daher keine Unterscheidung zwischen Modulen, Paketen oder Klassen gemacht, sonder vereinfacht von „Einheiten“ gesprochen.
Die Erstellung der Information in Zahlenform soll unabhängig vom Betrachter wiederholbar sein. Eine Metrik soll eindeutig bewertbar, kostengünstig und nützlich sein. Niemand hat einen Vorteil von einer Metrik, die nicht benutzt oder eingesetzt wird. Des Weiteren sollten Metriken normierbar und vergleichbar sein, damit sie als Indikatoren für andere Projekte benutzt werden können.

#### *2.1.1.2 Ziele & Zielgruppen von Metriken*

Das grundlegende Ziel von Softwaremetriken ist es, Software messbar zu machen. Durch die Verwendung von Metriken wird es überhaupt erst möglich, automatisiert Qualitätskriterien an Software anzulegen und mögliche Probleme zu sondieren. Diese Zahlen repräsentieren den aktuellen Zustand des untersuchten Objektes. Metriken findet man zahlreich. Hierfür können sowohl die Literatur als auch verschiedenste Werkzeuge zu Rate gezogen werden.

Ein Projektteam besteht aus Menschen, die wiederum verschiedenen Rollen einnehmen. Jede dieser Rollen hat ein unterschiedliches Interesse an Metriken, da der Fokus der jeweiligen Rolle anders auf einem Projekt liegt. Hier nun vier Beispiele für mögliche Rollen in einem Softwareentwicklungsteam:

- Management

  Manager eines Projektes sind interessiert am Fortschritt und an Informationen über

die bisherigen Kosten, das Erreichen von Meilensteinen, die Effizienz der Entwickler und ob Termine eingehalten werden.

- Entwickler

  Entwickler sind am Zusammenhang der Komponenten, also die eingehende oder ausgehende Abhängigkeit einer Einheit, und an der Kohäsion interessiert.

- Qualitätssicherer

  Versteckte Fehler, Verstöße gegen Programmierrichtlinien, z.B. Anzahl der Fehler pro Modul etc., stehen im Fokus des Qualitätssicherers, da hiermit sowohl die Robustheit, Fehlerfreiheit und Erweiterbarkeit erhöht, als auch Wartungskosten minimiert werden können.

- Softwarearchitekt

  Mittels Metriken ist es dem Softwarearchitekten möglich, die Architektur einer Software mit der geplanten Architektur zu vergleichen. Dies geschieht über die Darstellung von Graphen, in denen die Abhängigkeiten visualisiert werden. Ebenso besteht die Möglichkeit Schwächen der Architektur, wie z.B. Gott-Klassen oder Flaschenhälse messbar und damit erkennbar zu machen.

Metriken bilden die Beurteilungsgrundlage einzelner oder aller Einheiten eines Projektes. Sie dienen der Schwachstellensuche, Fehleraufdeckung und Fortschrittskontrolle durch einen zeitlichen Vergleich.
Metriken und Werkzeuge sollten, um diese zu ermitteln, jedem Mitglied des Projektteams zur Verfügung stehen. So wird dem Entwickler ermöglicht, seinen erstellten Quelltext selbstständig einer Qualitätssicherung zu unterziehen. Ziel eines jeden Entwicklers sollte es sein, möglichst hochwertige Software zu erstellen, Metriken helfen dabei. Wenn stets eine eigenständige Qualitätssicherung eines Entwicklers durchgeführt wird, steht dieser voraussichtlich weniger in der Kritik eines Qualitätssicherers. Dennoch müssen errechnete Ergebnisse einer Qualitätssicherung auch zentral präsentiert werden, damit alle Rollen transparent über den Zustand des Gesamtprojektes informiert sind.
An dieser Stelle muss klargestellt werden, dass Metriken nicht für den Zweck eingesetzt werden sollen, das Entwicklerteam zu „kontrollieren“. Effizienz lässt sich schwer in Zahlen ausdrücken und noch schwerer in Quelltexten. Hier gilt die Devise: mehr Quelltext ist nicht mehr Effizienz. Ein Entwickler, der möglichst viele Gedanken und Ideen in ein Stück Quelltext

legt, entwickelt meist kein großes Volumen an Quelltext sondern logischen, strukturierten und zielgerichteten Quelltext.

#### *2.1.1.3 Eine erste Beispielmetrik*

Ein Beispiel für ein Softwaremaß ist die Metrik Non Commenting Source Statements (NCSS). In der Programmiersprachen Java, C und C++ zählt die Summe von Ausdrücken, welche mit einem Semikolon enden oder mit einer öffnenden geschweiften Klammer beginnen, als NCSS. In anderen Programmiersprachen werden entsprechend die Endzeichen für Ausdrücke und Blöcke gezählt. Diese Information spiegelt die Größe eines Quelltextstückes ohne Kommentare wieder. Diese Information ist für Manager und Entwickler interessant. Im Allgemeinen gilt, Softwareeinheiten sollten nicht zu groß werden, damit die Wartbarkeit und Lesbarkeit erhöht wird.

Hier nun ein Beispielquelltext zur Verdeutlichung:

```
public static void main(String[] args) {

            System.out.println("Hallo Welt");

            boolean b1 = false;

            if(b1 == true)
            {
                  System.exit(1);
            }

      }
```

Wie in der Definition der NCSS werden nun die geöffneten geschweiften Klammern und die Semikolons gezählt:

| | |
|---|---|
| Anzahl geöffneter geschweifter Klammern: | 2 |
| Anzahl Semikolons: | 3 |
| →NCSS : | 5 |

#### *2.1.1.4 Schwellenwerte von Metriken*

Ein Team von Personen müsste zu diesem Informationsstand alle Metriken manuell überprüfen. Dieser Vorgang ist sehr zeitaufwändig und somit auch teuer. Metriken sind Zahlenwerte und um die Überprüfung der Metriken schneller und günstiger zu gestalten, kommen Schwellenwerte zum Einsatz.

Die eingesetzte Software zur Ermittlung von Metriken gibt einen Alarm oder eine Meldung aus, sobald eine Metrik ihren Schwellenwert überschreitet oder gleich diesem ist. Ein Alarm

ist aber noch kein absoluter Indikator für einen Fehler, denn evtl. ist die untersuchte Klasse mit Absicht so erstellt worden oder es gibt gute Gründe, diese Klasse exakt in dieser Größe oder Komplexität zu modellieren. Wichtig ist also, dass eine gefundene Metrik einen Alarm auslöst, wodurch eine Person ihr Augenmerk auf die betreffende Metrik legen kann. Somit umgeht man das manuelle Auswerten der Metriken durch eine Person und richtet das Augenmerk vielmehr auf die Brennpunkte eines Projektes. Die Schwellenwerte müssen einstellbar und änderbar sein.

#### *2.1.1.5 Beschaffung von Metriken*

Die Wahl von geeigneten Metriken ist ein oftmals komplexer Vorgang. Es könnte einfach aus dem Pool verfügbarer Metriken vermeintlich geeignet erscheinende Metriken gewählt werden. Jedoch ist in der Definition bzw. in den Kriterien an eine Metrik dargestellt, dass eine Kennzahl nützlich sein soll. So ergibt sich, dass für jedes Projekt erneut bei Beginn der Einführung einer metrikenbasierten Qualitätssicherung herausgefunden werden muss, welche Fragen eigentlich durch Metriken beantwortet werden sollen. Ein etabliertes Verfahren für das Ableiten von Metriken stellt das sog. Ziel-Frage-Metrik (GQM) Verfahren (Fenton & Pfleeger, 1997) dar. Dabei werden zunächst Ziele definiert. Eines unserer Beispiele würde als Ziel die Erweiterbarkeit des Systems gewählt werden. Darauf ergeben sich Fragen, die dieses Ziel beschreiben, z.B. „Wie Komplex ist eine Klasse?“, „Welche Programmier-sprache bietet Konzepte hierfür?“ oder „Gibt es große Abhängigkeiten einer Klasse zu anderen Klassen?“. Genau diese Fragen ermöglichen nun eine Auswahl von Metriken. Dieses gilt jedoch nicht für alle, denn z.B. die Frage nach Konzepten einer Programmiersprache wird von dem Projektteam im Vorweg recherchiert.

Hier nun ein Beispiel in grafischer Form:

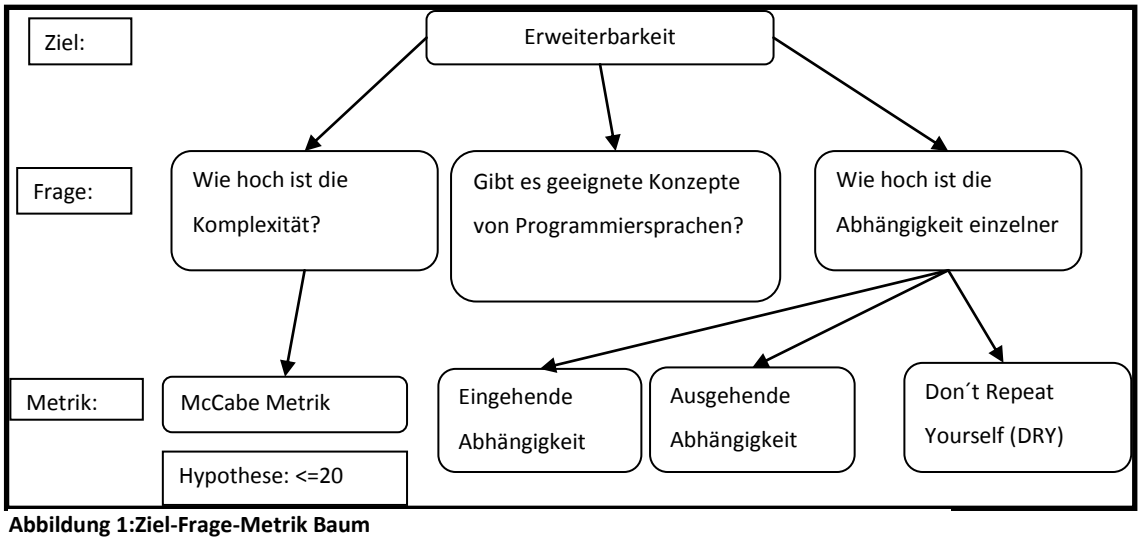



**Abbildung 1:Ziel-Frage-Metrik Baum**

Eine Auswertung erfolgt dann von den Blättern des ermittelten Baumes aus der GQM hin zur Wurzel. Es müssen entsprechend geeignete Schwellenwerte definiert sein, diese können und sollten dann als Hypothese zu den Blättern notiert werden. Mit diesem Verfahren ist es nunmehr möglich, Qualitätsmaße für ein Projekt zu ermitteln.
In der Literatur ist in Bezug auf Qualitätssicherung und Metriken oft die Rede von einem Qualitätsmodell (Fenton & Pfleeger, 1997). Solch ein Modell beschreibt Kategorien (Qualitätsmerkmale), welche für die Qualität der Software relevant sind. Nach Böhm [Boe78] stellen Portabilität, Zuverlässigkeit, Effizienz, Testbarkeit, Verständlichkeit und Veränderbarkeit solch ein Qualitätsmodell dar. Jede einzelne dieser Kategorien ist nun die Wurzel, also das Ziel, eines Ziel-Frage-Metrik Baumes und auf diese Ziele wird einzeln das Ziel-Frage-Metrik Verfahren angewandt, um an die entsprechenden Metriken zu gelangen. Für die Realisierung dieser Arbeit wurden im Abschnitt 4.2 Anforderungen ein eigenes Qualitätsmodell entwickelt, woraus dann Metriken abgeleitet wurden.

#### *2.1.1.6 Repräsentation von Metriken*

Eine Metrik hat einen Namen, einen Zweck, eine Berechnungsvorschrift und eine Beschreibung. Diese vier Fakten werden in die Beschreibung eingebaut. Diese Repräsentationsform eignet sich um den Nutzen und Ablauf einer Metrik allen Nutzern bereitzustellen. Hierzu ist der Wissensstand des Teams zu berücksichtigen, und wenn nötig eine Schulung zu geben. Als Beispiel nun die Repräsentation der schon bekannten NCSS Metrik:

| | |
|---|---|
| Name: | Non Commenting Source Statement (NCSS) |
| Zweck: | Volumenmetrik |
| Formel: | $NCSS = sL + bL$<br>sL: Anzahl von Zeilen, die mit Semikolon enden, bL: Anzahl von Zeilen, die mit geschweifter Klammer beginnen (brackets) |
| Beschreibung: | NCSS stellt eine Volumenmetrik dar, welche die Statements und Blöcke zählt. Die Summe Ergebnisse ist NCSS einer Einheit. |

#### *2.1.1.7 Kategorien von Metriken*

***Basismetriken***

Metriken werden schon seit vielen Jahren in der Literatur eingesetzt und entworfen. Man muss sich bewusst machen, dass in der Zwischenzeit viele technologische Änderungen eingetreten sind, so der Wechsel der strukturierten hin zur objektorientierten Entwicklung.

Wenn die Rede von „*alten*“ Metriken ist, dann sind solche gemeint, welche sich als Metriken seit vielen Jahren behaupten. NCSS oder die Lines of Code (LOC) stellen Beispiele für Basismetriken dar, auch Metriken wie Kopplung oder Komplexität nach McCabe sind Basismetriken.

***Abgeleitete Metriken***

Abgeleitete Metriken berechnen sich aus Basismetriken, um mehrere Metriken in Bezug zu setzen. Basismetriken sind absolute Zahlenwerte und abgeleitete Metriken sind statistische Größen mehrerer Basismetriken.

Für das bessere Verständnis nun ein Beispiel:

| | |
|---|---|
| Name: | Lines pro Klasse (LC) |
| Zweck: | Größe einer durchschnittlichen Klasse ermitteln |
| Formel: | $LC = \frac{LOC}{NC}$ |
| | LOC = Lines of Code, NC = Number of Classes |
| Beschreibung: | LC beschreibt das Verhältnis der LOC über eine Einheit zum Anzahl der Klassen in dieser Einheit. Ein Wert von größer als 500 ist bedenklich, da hier durch Refactorings und Splittung eine höhere Kohäsion pro Klasse erreicht werden kann. |

Hier sieht man, dass nur Basismetriken zur Berechnung eingesetzt werden, aber neuer Informationsgehalt hervorgehoben wird. Diese Kategorie von Metriken wird vor allem in den OO-Metriken eingesetzt. Diese stellen seit dem Paradigmenwechsel von der strukturierten Programmierung hin zur objektorientierten Programmierung Maße dar, welche die

Möglichkeiten der Objektorientierung berücksichtigen. Zu solchen Metriken gehören z.B. die *weighted methods per class*, welche die Summe der Komplexität der Methoden einer Klasse angibt und dabei durch einen definierten Faktor die einzelnen Komplexitäten gewichten kann. OO-Metriken werden nicht ausschließlich aus Basismetriken abgeleitet, sondern benötigen häufig eigens für sie entwickelte Berechnungsgrundlagen und Algorithmen. Dadurch können nicht alle OO-Metriken mit Werkzeugen, die Basismetriken ermitteln können, errechnet werden.

### *Weitere Kategorien*

Nachdem nun Basis- und abgeleitete Metriken bekannt sind, ist zu klären, welche Arten von Metriken es noch gibt. Die bisher genannten Metriken sind technische Metriken. Diese befassen sich mit dem Quelltext, also der konkreten Realisierung der Modelle und Architektur, und analysieren diesen auf Fehler und errechnen definierte Kennzahlen.

Eine weitere Kategorie an Metriken sind die nicht-technischen. Nicht-technische Metriken befassen sich mit dem Projektablauf außerhalb des Quelltextes. Hierzu gehören die Use-Cases, Anforderungskontrolle, Aufwandsabschätzungen, Budgetkontrolle, Meilensteinerreichung, Termineinhaltung oder die Anzahl an Fehlern. Die meisten dieser Metriken sind nicht maschinell berechenbar und müssen manuell gepflegt, eingefügt und ausgewertet werden. Einige Ansätze für die maschinelle Prüfung von einzelnen nicht-technischen Metriken gibt es, so kann anhand einer gepflegten Anforderungsliste in Form von Tests das Erreichen von Anforderungen ermittelt werden. Solch eine Metrik würde wie folgt aussehen:

| | |
|---|---|
| Name: | Fortschritt anhand von Anforderungen (FA) |
| Zweck: | Fortschrittskontrolle, Budgetkontrolle, Anforderungskontrolle |
| Formel: | $FA = \frac{pT}{gT}$<br>pT =Anzahl positiv bestandener Tests, gT = Anzahl gesamter Anforderungstests |
| Beschreibung: | Für diese Metrik müssen die Anforderungen als Test modelliert werden. Wird ein Test bestanden, gilt die Anforderung als realisiert. Änderungen an den Anforderungen müssen entsprechend eingefügt werden. |

Diese Unterscheidung der Metriken mag sehr grob klingen, und es gibt weitere granularer gestaffelte Unterteilungen, aber für diese Untersuchung passt diese Unterteilung sehr gut, denn es gilt herauszufinden, welche technischen und nicht-technischen Metriken mittels Maven2 realisiert werden können.

#### *2.1.1.8 Verlauf einer Metrik*

Metriken entfalten ihren eigentlichen Informationsgehalt erst, wenn man einen Vergleich hat. So können andere, ähnliche Projekte zum Vergleich dienen, aber auch der Verlauf über die Zeit, also die Entwicklung einer Metrik ist entscheidend. Denn erst über den Verlauf kann man erkennen, ob Veränderungen auch einen Einfluss auf eine Metrik hatten oder ob Fehler beseitigt wurden. Steigt beispielsweise die Komplexität einer Klasse permanent an, so sollte hierauf besondere Aufmerksamkeit gelegt und eine Überarbeitung angesetzt werden. Programme zur Ermittlung von Metriken müssen entsprechend eine Funktion bereitstellen, welche die Ansicht der Historie und des Verlaufes von Metriken und Fehlern möglich macht.

### 2.1.2 Programmierstil

Programmierstil umfasst die Definition eines einheitlichen Verfahrens, welches festlegt, wie die Syntax einer Programmiersprache eingesetzt wird. Durch die Definition von Regeln für die Erstellung von Quelltexten soll die Einarbeitungszeit in einen Quelltext minimiert werden. Im Projekt des Kunden wird als Entwicklungsmethode die agile Software-entwicklung angewandt. Eine der in dieser Methode propagierten Regeln ist der gemeinschaftliche Quelltextbesitz. Das bedeutet, dass das gesamte Entwicklerteam Zugriff

auf alle Quelltexte hat und bei Bedarf diese überarbeiten kann. Ebenfalls wird im Entwicklerteam des Kunden die Methode der manuellen Quelltextüberprüfung angewandt. Dabei wird, bevor eine Aufgabe abgeschlossen ist, der Quelltext von dem Ersteller und einem zweiten Entwickler durchgesehen und diskutiert. Programmierstil muss aber auch maschinell überprüft werden können, um den Aufwand für das Entwicklerteam zu minimieren.

Genau für solche Aufgaben ist ein einheitlicher Programmierstil nötig. Dadurch wird es den Entwicklern vereinfacht, sich in nicht selbsterstellte Quelltexte einzuarbeiten. Zu einem guten Stil gehört laut Literatur (Sun, 1999) immer eine Dokumentation des Quelltextes. So sollte eine Softwareeinheit immer dahingehend dokumentiert sein, dass der Zweck der Einheit und der einzelnen Methoden kommentiert ist.

#### *2.1.2.1 Vordefinierte Programmierstandards*

Sun, als Hersteller der Programmiersprache Java, stellt auf seinen Webseiten einen Programmierstandard für Java zur Verfügung (Sun, 1999). In diesem Standard wird Java Entwicklern von Sun empfohlen, wie Imports oder Statements umzusetzen sind. So soll der Import von Bibliotheken nicht mittels des Sternoperator (import java.sun.*) die gesamte Bibliothek in das Softwareprojekt einbinden, sondern nur die benötigten Einheiten. Das mindert die Kopplung an die Bibliothek und macht das Projekt robuster gegenüber Änderungen in der Bibliothek. Es gibt von Sun zehn Kategorien für den Programmierstil:

- Namenskonventionen
- Dateiorganisation
- Identifikation
- Kommentare
- Deklarationen
- Ausdrücke
- Leerzeichen
- Dateinamen
- Programmieranwendungen
- Importe

Solch ein vordefinierter Programmierstil bietet einen guten Einstieg. Dennoch sollten die Regeln vom Entwicklerteam durchgesehen werden, welche Regeln wirklich auf das Softwareprojekt angewandt werden.

#### *2.1.2.2 Eigener Programmierstil*

Einem Entwicklerteam sollte die Möglichkeit zur Verfügung gestellt werden, einen eigenen Programmierstil zu entwickeln. Einmal definierte Regeln sind diskutierbar und das Regelwerk kann jederzeit um weitere Regeln ergänzt werden. Hierdurch wächst über die Zeit die Vertrautheit mit den Regeln und deren Anwendung.

#### *2.1.2.3 Einhaltung des Programmierstils*

Ein definierter Satz an Stilregeln sollte immer eingehalten werden. Nachdem ein Quelltext erstellt oder verändert wurde, sollte jeder Entwickler den Quelltext auf die Einhaltung des Stils prüfen. Dies kann er manuell durchführen, indem er den Regelsatz visuell mit dem Quelltext vergleicht. Einfacher ist die Einhaltung durch den Einsatz eines Werkzeugs. Solch ein Werkzeug muss den erstellten Regelsatz aufnehmen und auf den Quelltext anwenden können. Beispiele für solche Werkzeuge im Java Umfeld sind CheckStyle und PMD.

## 2.2 Maven2

Maven2 ist das in diesem Projekt eingesetzte Buildmanagementwerkzeug. Dem Entwickler wird durch Maven2 die Möglichkeit angeboten, verschiedene Dinge in einem Softwareprojekt zu verwalten. Zu den verwaltbaren Aspekten gehören das Erzeugen von Software und die Erstellung einer Dokumentation. Ähnlich wie andere Buildmanagementwerkzeuge wie *Ant* oder *Make* ist mit Erzeugen von Software nicht die Erstellung von Quelltexten sondern Abläufe wie Kompillierung, Testen oder Installation gemeint. Damit Softwareprojekte mit Maven2 verwaltet werden können, müssen sie eine für Maven2 kompatible Struktur aufweisen. So eine Struktur ist wie folgt aufgebaut:

```
my-app
|-- POM.xml
`-- src
    |-- main
    |   `-- java
    |       `-- com
    |           `-- mycompany
    |               `-- app
    |                   `-- App.java
    `-- test
        `-- java
            `-- com
                `-- mycompany
                    `-- app
                        `-- AppTest.java
```

**Abbildung 2: Maven2 Projektstruktur**

Dies ist eine verkürzte Darstellung, da noch mehr Elemente in einem Softwareprojekt von Maven2 vorkommen können. Anhand der Abbildung 2 ist erkennbar, dass auf der obersten Projektebene die POM.xml, die zentrale Konfigurationsdatei für Maven2, und Ordner für die Quelltexte und Tests vorhanden sein müssen. Die Entwickler von Maven2 (Group, 2008) stützen dieses Vorgehen darauf, dass Maven2 das Konzept „Konvention über Konfiguration“ einsetzt. Das besagt, dass spezielle Konfigurationsmöglichkeiten für bestimmte Projekte nötig sind, aber in den meisten Fällen nur ein geringer Prozentanteil an Aufwand in die Konfiguration gesetzt werden soll. Dafür muss eine gewisse Struktur eingehalten werden, damit Maven2 seine Funktionen auf ein Softwareprojekt anwenden kann.

Im Kern ist Maven2 ein Ausführungsframework, welches zusätzliche Funktionen über Plug-Ins anbieten kann.

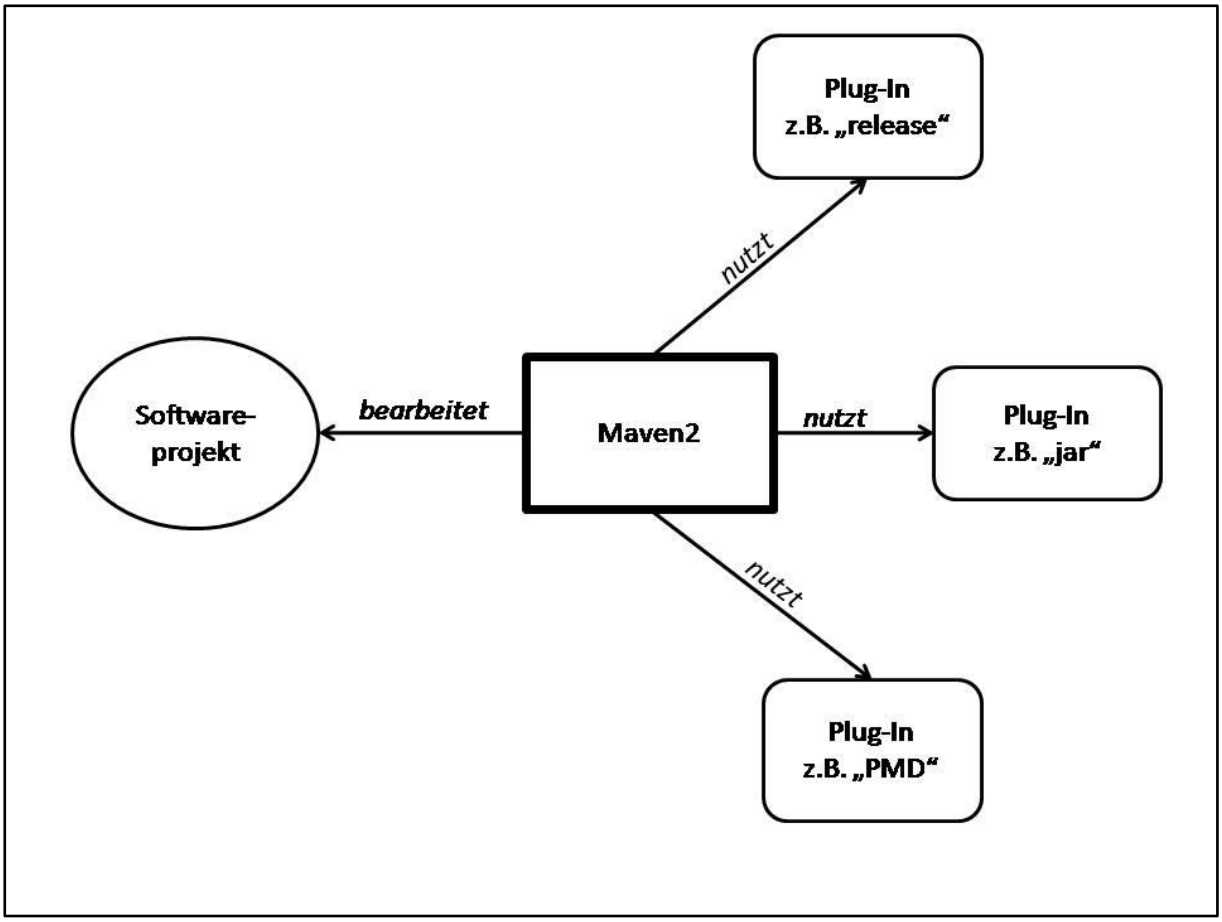



**Abbildung 3: Maven2 Architektur vgl. (Massol)**

Diese Möglichkeit des Einbindens von Plug-Ins ist im Rahmen dieser Arbeit von Interesse, da Plug-Ins für qualitätssichernde Maßnahmen vorhanden sind. An dieser Stelle sollen nun die für diese Arbeit relevanten Fakten zu Maven2 aufgearbeitet werden.

### 2.2.1 Lebenszyklen

Maven2 unterscheidet zwischen zwei Lebenszyklen, dies sind der Erstellungs- und der Seitengenerierungslebenszyklus. Das Konzept von Maven2 ist die Standardisierung von Abläufen unter Einsatz von Maven2 in einem Projekt. Deshalb sind die Grundfunktionen von Maven2 standardisiert und werden immer gleich verarbeitet. Der Erstellungslebenszyklus beschreibt den expliziten Ablauf des Erstellungsprozesses. Hier nun ein Auszug aus den einzelnen Stufen dieses Lebenszyklus:

- validate: prüft, ob das Projekt mit Maven2 kompatibel ist und alle Daten zum Ablauf vorhanden sind
- compile: kompiliert die Quelltexte
- test: lässt die im Softwareprojekt enthaltenen Tests ablaufen

- package: packt die kompilierten Quelltexte in ein verteilbares Format wie „jar“
- verify: verifiziert das erstelle Paket
- install: installiert das Paket in das lokale Repository
- deploy: installiert das Paket in ein dezentrales Repository

Diese Stufen stellen jeweils Befehle für den Entwickler bereit, mit denen die entsprechende Stufe ausgeführt wird. Dabei werden auch alle darunterliegenden Stufen durchlaufen. Entsprechend wird beim Ausführen des *compile* Befehls immer zuerst *validate* von Maven2 durchgeführt. Das bedeutet für einen Entwickler, dass er nur eine kleine Anzahl an Befehlen wissen muss, um ein Softwareprojekt durch Maven2 erstellen zu lassen.

Der zweite Lebenszyklus umfasst die Seitengenerierung. Maven2 stellt eine Funktion bereit, mit der sich aus den Softwareprojekten Projektwebseiten mit spezifischen Projekt-informationen erstellen lassen. In diesem gibt es zwei Stufen:

- site: generiert die Projetwebseite für jedes Projekt
- site-deploy: installiert die ermittelten Webseiten an einen beliebigen Ort (Webserver, Datei)

Es ist möglich über ein spezielles Dateiformat namens Almost Plain Text (APT) eigene Inhalte zu erstellen. Mittels des Seitengenerierungslebenszyklus werden die ermittelten Informationen der Qualitätssicherung für das Projektteam publiziert.

Abweichungen von den Standardabläufen in Maven2 erfolgen beim Einsatz zusätzlicher Plug-Ins, die eigene Befehle zur Verfügung stellen. Es besteht aber die Möglichkeit, diese eigenen Befehle an einzelne Stufen der Lebenszyklen zu binden. So kann ein Plug-In, welches zusätzliche Informationen über eine Paket installiert, an den *deploy* Befehl gebunden werden. Dieses Plug-In wird dann automatisch bei jedem *deploy* Aufruf ausgeführt. Plug-Ins, die ihre Ergebnisse auf der Projektwebseite veröffentlichen, sind größtenteils so vorkonfiguriert, dass sie bei der Seitengenerierung automatisch ausgeführt werden.

### 2.2.2 Project Object Model (POM)

Das *POM* stellt für jedes Projekt die zentrale Konfigurationsdatei für Maven2 dar. Hier werden Anweisungen für den Umgang mit diesem Projekt und Informationen über das Projekt definiert. So kann z.B. ein Format, wie *jar* oder *war*, für die *package* Stufe des

Erstellungslebenszyklus in das *POM* eingetragen werden. Jedes Projekt im Maven2 Umfeld hat genau ein *POM*. Maven2 ermöglicht, im Gegensatz zu anderen Erstellungswerkzeugen, die Einführung von Projekthierarchien. Dabei handelt es sich um Vererbungsbeziehungen zwischen verschiedenen Softwareprojekten, bei denen die *POM*-Einstellungen der Eltern von den Kindern automatisch übernommen werden. Dies mindert den Konfigurationsaufwand in großen Projekten, da Einstellungen nur einmal an einem Ort getätigt und nicht in jedem Projekt wiederholt werden müssen. In dem *POM* werden auch zusätzlich einzusetzende Plug-Ins deklariert. Dabei muss lediglich der komplette Name und der Ort, an dem das Plug-In im Internet zu Verfügung gestellt ist, angegeben werden und Maven2 besorgt, installiert und führt automatisch das Plug-In aus. Weiter Informationen über das *POM* und den Aufbau finden sich auf der Herstellerseite (sonatype, 2008).

## 2.3 Zusammenfassung

Dieses Kapitel beschreibt, wie funktionale und teilweise auch nichtfunktionale Qualitätskriterien in Form von Metriken dargestellt werden können. Dabei wurden auch der Nutzen, der Weg zur Ermittlung von Metriken und die Möglichkeiten ihres Einsatzes verdeutlicht. Ein einheitlicher Programmierstil in einem Entwicklerteam erhöht die Effizienz, da die Einarbeitung in „fremden“ Code erleichtert wird. Metriken und Programmierstil sollen ihren Einsatz im Kontext von Maven2 finden. Maven2 ist die eingesetzte Erstellungsumgebung und bietet aufgrund der Plug-In Architektur und der Möglichkeit der Seitengenerierung einen Ansatz für den Einsatz von qualitätssichernden Maßnahmen.

# 3 Vision

Die zentrale Idee dieser Arbeit ist es, eine kontinuierliche, begleitende Qualitätssicherung mit Hilfe von Maven2 verfügbar zu machen.

Unter kontinuierlich und begleitend versteht sich im Rahmen dieser Arbeit, dass nicht nur zu einigen Zeitpunkten, sondern in regelmäßigen, sich wiederholenden, kurzen Zeitintervallen eine Qualitätssicherung stattfindet. Dieser Vorgang findet über die gesamte Projektlaufzeit hinweg statt. Wie im Abschnitt 2.2 beschrieben, bietet Maven2 aufgrund seiner Plug-In Architektur an, automatisch Projektwebseiten zu generieren und zusätzliche Plug-Ins auszuführen. Es gilt zu evaluieren, in welchem Maß, unter Berücksichtigung der Ansprüche des Kunden und der Literatur, mittels der Plug-In Architektur eine nutzbare Qualitäts-sicherung realisierbar ist. Dabei ist die Flexibilität des Ansatzes entscheidend, denn jedes Projekt stellt andere Anforderungen an Qualität. So kann es von Interesse sein, eigene Programmierstilregeln zu definieren oder nicht zur Verfügung stehende Metriken durch ableitbare Metriken bereitzustellen. Die Qualitätssicherung soll von jedem Projekt-mitarbeiter einsehbar, verständlich und anwendbar sein. Sollten einige Punkte nicht mit dem Standardverhalten erreicht werden, sollen Wege und Ansätze gefunden werden, die Differenz zwischen Anspruch und Realisierung möglichst klein zu halten. Dazu gehört auch die Abschätzung, wie hoch der Aufwand für die Minimierung der Differenzen ist, und ob die Ansätze entsprechend weiter verfolgt werden.

# 4 Gespräche und Anforderungsanalyse

In diesem Abschnitt geht es darum, die Anforderungen an dieses Projekt zu definieren. Hierzu erfolgten Kundengespräche, aus diesen ergaben sich die Anforderungen, welche die Ziele dieser Arbeit bestimmten. Kundenanforderungen sind nicht leicht zu definieren, daher wurde versucht, die sich aus den Gesprächen ergebenden Anforderungen auf Faktoren in Form von Metriken abzubilden. Dabei wurde mehrfach das Verfahren GQM angewandt. Dieser Vorgang ist im Abschnitt 4.2 dargestellt.

## 4.1 Gespräche

Im Laufe dieser Arbeit gab es mehrere Treffen mit dem Entwicklerteam, in denen es um die Anforderungen und Präsentation von erreichten Zwischenschritten ging. Folgende Anforderungen wurden vom Kunden in diesen Treffen vorgetragen:

- Qualitätssichernden Maßnahmen
  Diese Anforderung ist der Initialwunsch des Kunden und wurde das Thema dieser Arbeit.
- Übersicht über Erweiterungen von Maven2 für eine Qualitätssicherung
  Der Kunde wünscht eine Marktanalyse über vorhandene Plug-Ins für Maven2, die als Werkzeuge für eine Qualitätssicherung dienen können.
- Design- und Architekturprüfung
  Es sollen Metriken gefunden und umgesetzt werden, die Aussagen über das Design und die Architektur vorhandener Softwareprojekte treffen.
- Fehler
  Verdeckte Fehler in Softwareprojekten sind aufzudecken.
- Programmierstil
  Ein einheitlicher Programmierstil ist einzuführen und zu prüfen. Es gilt zu evaluieren, welche Werkzeuge dafür zur Verfügung stehen und welche Stilelemente diese prüfen können. Aus dem Angebot der verfügbaren Stilelemente soll zusammen mit dem Entwicklerteam eine Auswahl getroffen werden.
- Übersicht über die Ergebnisse
  Die Ergebnisse der errechneten Metriken und Prüfungen sollen zentral auf einer Webseite verfügbar gemacht werden.

- Verlauf der Ergebnisse

  Die Ergebnisse der errechneten Metriken und Prüfungen sollen in einem Verlauf über die Zeit dargestellt werden, damit Tendenzen in den Softwareprojekten darstellbar und Probleme erkennbar gemacht werden können.

- Integration in Maven2

  Die Werkzeuge für Maven2 sollen möglichst gut in Maven2 integriert sein, d.h. es sollen so wenig wie möglich zusätzlichen Serverdienste oder Datenbanken eingeführt werden.

- Fortschrittsmessung

  Mit der Fortschrittsmessung ist eine Möglichkeit gemeint, die spezifizierten Features der Softwareprojekte prozentual darzustellen. Dabei sind 100% die Gesamtzahl an Features und es soll das Verhältnis der realisierten Features zu der Gesamtzahl dargestellt sein.

Dies sind die Anforderungen, die das Kundenteam geäußert hat. Diese Anforderungen wurden anschließend zusammen mit dem Kunden in einer ABC-Analyse in drei Kategorien eingeteilt. Die Kategorien sind „must“, „should“ und „nice to have“.

| Anforderung | must | should | nice to have |
|---|---|---|---|
| Qualitätssichernde Maßnahmen | X | | |
| Übersicht über Maven2 Erweiterungen | X | | |
| Design- und Architekturprüfung | X | | |
| Fehler | X | | |
| Programmierstil | X | | |
| Übersicht der Ergebnisse | X | | |
| Verlauf | X | | |
| Integration | | X | |
| Fortschrittsmessung | | | X |

**Tabelle 1: ABC Analyse der Anforderungen des Kunden**

Es wurde vereinbart, dass nun eine Anforderungsanalyse folgt. In der Anforderungsanalyse werden nun die Anforderungen des Kunden um generelle Anforderungen der Literatur an qualitätssichernde Maßnahmen in Softwareprojekten erweitert.

## 4.2 Anforderungen

Um eine Qualitätssicherung einführen zu können, empfiehlt es sich, zunächst ein Modell aufzubauen, welche die Qualitätskriterien beschreibt. Für solch ein Modell, wie im Abschnitt

2.1.1.5 beschrieben, müssen Aspekte und Schwerpunkte erarbeitet werden, die die Qualitätskategorien für die betrachteten Projekte darstellen. Auch in dieser Arbeit ist solch ein Modell entstanden. Dabei sind in den Anforderungen schon drei Kategorien enthalten, die in das Qualitätsmodell aufgenommen werden, diese sind Design, Fehler und Programmierstil. Zusammen mit dem Kunden wurden noch weitere Kategorien gefunden, die möglichst mittels Metriken oder anderen Mechanismen realisiert werden sollen. Eine Übersicht über das entstandene Qualitätsmodell ist nun grafisch dargestellt:

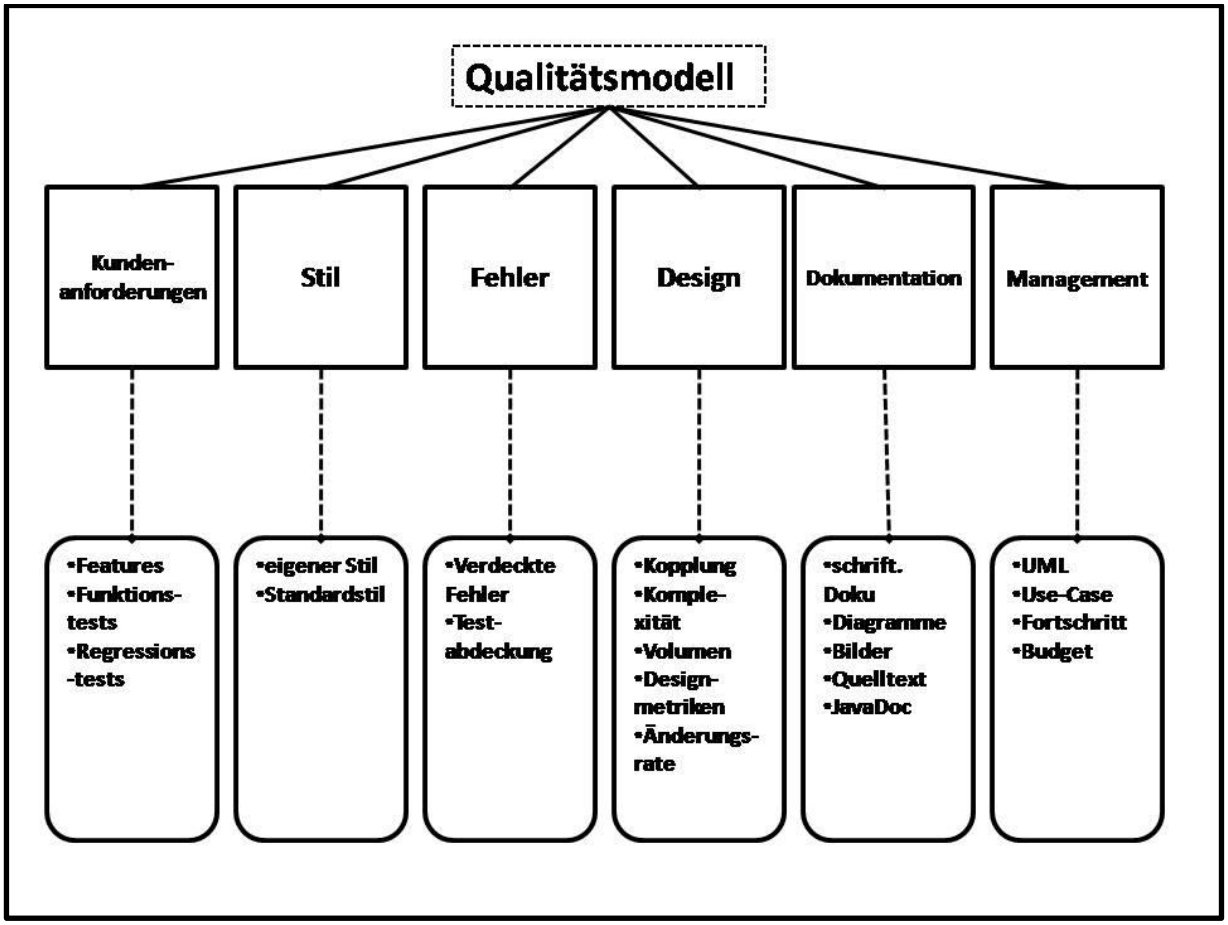



**Abbildung 2: Qualitätsmodell**

Die drei Kategorien Kundenanforderungen, Dokumentation und Management sind zu den bisherigen Anforderungen hinzugekommen. In der Kategorie Design sind einige konkrete gewünschte Metriken bestimmt worden. Um nun weitere Metriken für diese Arbeit zu finden, wurde für jede Kategorie eine Frage-Ziel-Metrik Analyse durchgeführt. Die Ergebnisse der Analysen werden nun für jede Kategorie beschrieben. Die Literatur zu den Themen Metriken und Metrikenfindung stellt die Basis für die Ermittlung der für das Qualitätsmodell relevanten Metriken dar. Dieser Vorgang wird in dieser Arbeit als

„Anforderungsmapping“ bezeichnet. Dabei werden die Anforderungen des Kunden in Absprache mit diesem um den aktuellen Stand der Literatur erweitert.

### 4.2.1 Kundenanforderungen

Kundenanforderungen stellen die Anforderungen an das Softwareprojekt dar, welches durch diese Arbeit einer Qualitätssicherung unterzogen werden soll. Qualität aus Sicht der Kundenanforderung ist größtenteils gewährleistet, wenn die Definition nach der IEEE 739-1983 erreicht wurde. Diese besagt, dass Qualität erreicht ist, wenn die Software Spezifikationskonform ist. Dieses Verhalten ist prüfbar, indem eine entsprechende Testumgebung eingesetzt wird, welche die Features des Systems testet (Funktionstests, Regressionstests). Dieser Punkt ist zwar in das Qualitätsmodell aufgenommen, wird aber im Konsens mit dem Kunden im Rahmen dieser Arbeit nicht weiter verfolgt.

### 4.2.2 Fehler

Fehler sind unvermeidbar. Sobald eine Einheit Software erstellt wird, ist es wahrscheinlich, dass sie Fehler enthält. Dieses liegt nicht am Unvermögen der Entwickler, denn die Komplexität einer eingesetzten Programmiersprache mit all ihren kombinatorischen Möglichkeiten und Rahmenbedingungen ist nicht zu jedem Zeitpunkt erfassbar. Dennoch sollten Fehler der Architektur, der Stils, der Anwendung einer Programmiersprache oder ähnliches erkennbar gemacht werden. Dabei soll niemand im Entwicklerteam für solche Fehler verantwortlich gemacht werden. Vielmehr ist die automatisierte Suche nach möglichen Fehlern von Interesse, um die Korrektheit zu bewahren.
Testen ist eine Möglichkeit, Fehlern vorzubeugen. Entsprechend ist die Metrik der Testabdeckung hier von Interesse.

Die Testabdeckung ist wie folgt definiert:

| | |
|---|---|
| Name: | Test-Coverage (TC) |
| Zweck: | Test-Abdeckung, Erweiterbarkeit, Funktionsumfang |
| Formel: | $TC = \frac{tL+tB}{aL+aB}$<br>tL = Anzahl getesteter Zeilen, tB = Anzahl getesteter Verzweigungen (if, while, switch…), aL = Anzahl gesamter Zeilen (lines), aB= Anzahl gesamter Verzweigungen (braches) |
| Beschreibung: | TC ist der Anteil getesteter Zeilen und Verzweigungen an der Gesamtzahl von Zeilen und Verzeigungen. |

Die Interpretation der Testabdeckung ist jedoch nicht so einfach, wie eventuell angenommen. Die Testabdeckung zählt das Vorhandensein von Tests für die Zeilen und Verzweigungen einer Softwareeinheit. Dabei wird nicht der Inhalt der Tests geprüft. Deshalb kann bei einer maximalen Testabdeckung nicht davon ausgegangen werden, dass alle Überarbeitungen der Softwareeinheiten abgesichert sind. Der Grund dafür ist, dass für eine 100 %ige Testabdeckung in einer Zeile gegebenenfalls mehrere Testfälle benötigt werden, aber ein Test bereits genügt, um die Testabdeckungsmetrik zu erfüllen. Daraus folgt, dass bei Erreichen einer 100%igen Testabdeckung anschließend die Qualität der einzelnen Tests manuell überprüft werden muss.

Für die Lokalisierung verdeckter Fehler sollen Werkzeuge eingesetzt werden. Die Marktanalyse in Abschnitt 5.1 wird eine Übersicht über verfügbare Maven2 Plug-Ins für diesen Zweck darstellen.

### 4.2.3 Design

Design ist ein häufig genutztes Wort, dessen Bedeutung mehrfach besetzt ist. Design hat in dieser Arbeit, im Umfeld von Metriken, die Bedeutung des Aufbaus von Einheiten und der Beziehung zwischen Einheiten.

In die Kategorie des Designs fallen viele Metriken der Literatur. Zu diesen gehören Kennzahlen welche die Kopplung, Komplexität, Volumen oder Metriken, die architektonische Umstände repräsentieren.

Hier nun eine Sammlung von Metriken, welche die soeben genannten Kriterien beschreiben.

Volumenmetriken:

| | |
|---|---|
| Name: | Non Commenting Source Statement (NCSS) in Java |
| Zweck: | Volumenmetrik |
| Formel: | $NCSS = sL + bL$<br>sL: Anzahl von Zeilen, die mit Semikolon enden, bL: Anzahl von Zeilen, die mit geschweifter Klammer beginnen (brackets) |
| Beschreibung: | NCSS stellt eine Volumenmetrik dar, welche die Statements und Blöcke zählt. Die Summe Ergebnisse ist NCSS einer Einheit. |

An dieser Stelle wären auch weitere Metriken wie Anzahl von Kommentaren oder die Gesamtzahl an Zeilen im Quelltext zu erwähnen. Diese werden hier nicht in der bekannten Darstellungsform aufgezeigt, da der Aufbau sehr ähnlich der NCSS Metrik ist. Weitere Metriken für das Volumen sind abgeleitete Metriken, die einen Überblick über verschieden Größen geben. Interessant wäre hier z.B. die Anzahl von Zeilen pro Klasse:

| | |
|---|---|
| Name: | Zeilen pro Klasse (LC) |
| Zweck: | Volumen, Architektur |
| Formel: | $LC = \frac{LOC}{NC}$<br>LOC = Lines of Code, NC = Number of Classes |
| Beschreibung: | LC beschreibt das Verhältnis zwischen der Gesamtzahl an Zeilen und der Gesamtzahl an Klassen in einem Projekt. Das Ergebnis ist eine Zahl, die eine Aussage über das mittlere Volumen einer Klasse macht. Ist diese sehr groß, kann es sich um einen architektonischen Mangel handeln. |

Kopplungsmetriken:

| | |
|---|---|
| Name: | eingehende Kopplung (Afferent Coupling) |
| Zweck: | Abhängigkeitskontrolle, Erkennung von Schwachstellen, Erweiterbarkeit, Unabhängigkeitsmaß |
| Formel: | Ca = Anzahl von Einheiten, die diese Einheit nutzen |
| Beschreibung: | Ca zählt die Anzahl von Klassen/Paketen, welche diese Klassen nutzen. |

| | |
|---|---|
| Name: | ausgehende Kopplung (Efferent Coupling) |
| Zweck: | Abhängigkeitskontrolle, Erkennung von Schwachstellen, Erweiterbarkeit, Unabhängigkeitsmaß |
| Formel: | Ce = Anzahl von Einheiten, die diese Einheit benutzt |
| Beschreibung: | Ce zählt die Anzahl von Einheiten, welche die betrachtete Einheit bzw. Methoden dieser, benutzt. |

Kopplung beschreibt das Maß der Abhängigkeit einer Einheit von anderen Einheiten, sowie die Abhängigkeit anderer Einheiten von einer Einheit. Für die Erweiterbarkeit und Wartbarkeit eines Projektes sind dies wichtige Metriken, da diese automatisch ermittelt werden können und evtl. Schwachpunkte der Umsetzung der Architekturvorgaben ersichtlich machen. Handelt es sich um Mängel an der Soll-Architektur, so werden Einheiten mit zu hoher Verantwortlichkeit und Flaschenhälse sichtbar.

Komplexitätsmetriken:

Laut Qualitätsmodell wird eine Komplexitätsmetrik gefordert. Nun wurde geprüft, welche Metrik als Komplexitätsmetrik geeignet ist. Die zyklomatische Komplexitätsmetrik nach McCabe (McCabe, 1976) stellte sich als geeignet heraus, da es sich um eine der meist genutzten Komplexitätsmetriken handelt. Dadurch entstand auch die in der Metrikdefinition enthaltende Skala über einen Zeitraum von mehreren Jahrzehnten und kann als *best practice*

vergl. [FL] übernommen werden. Die zyklomatische Komplexität nach McCabe (CNN) gilt als berechnungsschnelle Metrik und ist in vielen Werkzeugen enthalten.

Name: Zyklomatische Komplexität (CCN)

Zweck: Komplexitätsbeschreibung, Erweiterbarkeit, Wartbarkeit

Formel: $CCN = |E| - |V| + 2$

F= Ablaufgraph eines Quelltextes ,|E|= Anzahl Kanten, |V|= Anzahl Knoten

Beschreibung: CCN beschreibt die Anzahl möglicher Ablaufpfade eines Quelltextes. Sie bildet die Grundlage für die Anzahl von Testfällen. Relevant für die Berechungen sind Verzweigungen wie *if*, *switch* oder *catch* in Java. Man kann bei der Auswertung folgende Skala verwenden:

| CCN | Risiko Auswertung |
|---|---|
| 1-10 | Einfaches Programm, kein großes Risiko |
| 11-20 | Komplexeres Programm, annehmbares Risiko |
| 21-50 | Komplexes Programm, hohes Risiko |
| >50 | Unstabiles Programm, maximales Risiko |

Um den Gebrauch und die Aussagekraft der CCN zu verdeutlichen folgt nun ein Beispielprogramm:

```
public class HelloNr {

	/**
	 * Dieses Programm berechnet eine eingegebene Operation
	 * und gibt das Ergebnis auf der Konsole aus.
	 * @param args
	 */
	public static void main(String[] args) {
		double x = new Double(args[0]);
		String op = args[1];
		double y = new Double(args[2]);

		switch(op.charAt(0))
		{
		case '+':
			System.out.println(x+y);
		case '-':
			System.out.println(x-y);
		case '*':
			System.out.println(x*y);
		case '/':
			System.out.println(x/y);
		}

		System.out.println("Berechnung beendet...");
	}
}
```

Nun soll für dieses einfache Programm der CCN Wert ermittelt werden. Dafür muss zunächst der Ablauf des Programms in einem Graphen dargestellt werden:

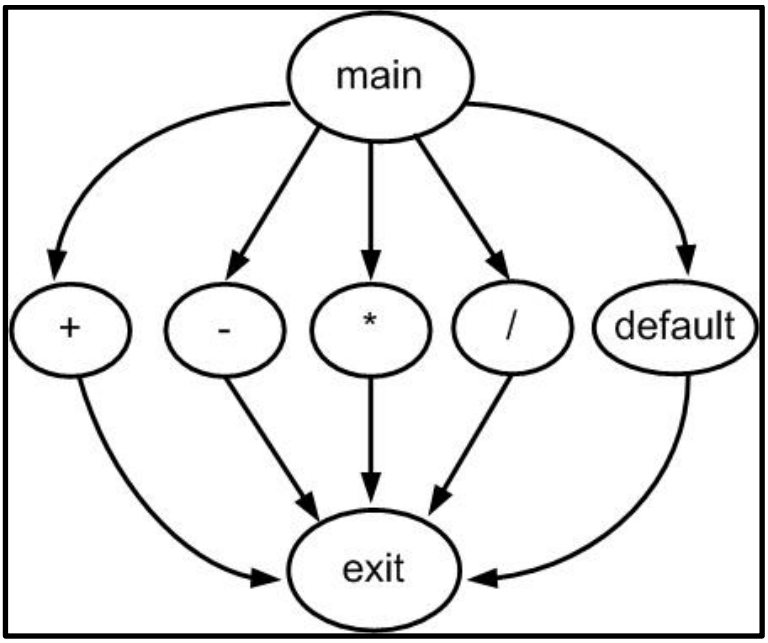



**Abbildung 3: Ablaufgraph der HelloNr Klasse**

Die Berechnungsvorschrift für die CCN Metrik besagt, dass nun die Anzahl der Knoten und Kanten gezählt werden.

Anzahl Knoten: 7

Anzahl Kanten: 10

Formel: Anzahl Kanten – Anzahl Knoten +2

10 – 7 +2 = 5

Die Klasse *HelloNr* hat nach dieser Rechnung eine zyklomatische Komplexität nach McCabe von fünf.

Die minimale zyklomatische Komplexität nach McCabe beträgt 1. Ein gutes Beispiel dafür, warum der Wert 1 ist, ist die populäre „Hello World“ Anwendung. Dabei wird in einer Klasse mit *main* Methode (Einstiegspunkt der Anwendung) nur eine Konsolenausgabe ausgegeben und anschließend wird die Anwendung beendet (exit). Der für diese Klasse zutreffende Graph hat zwei Knoten und eine Kante. Daraus resultiert folgende Rechnung:

Anzahl Knoten: 2

Anzahl Kanten: 1

Formel: Anzahl Kanten – Anzahl Knoten +2

1-2+2 = 1

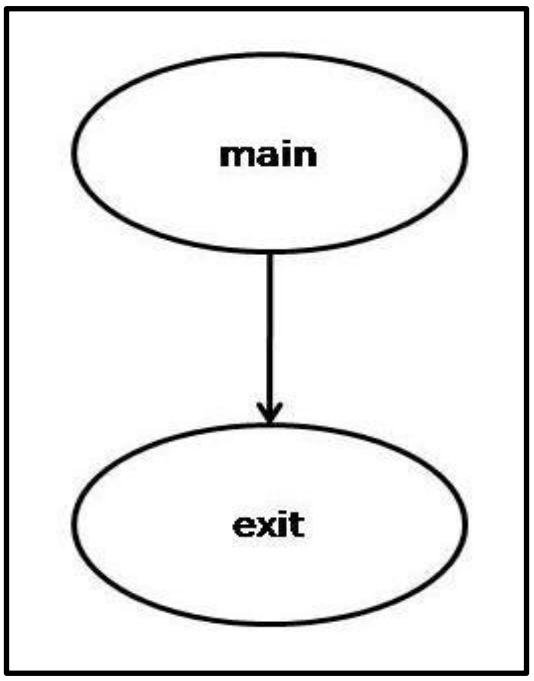



**Abbildung 4: Minimaler Anwendungsgraph**

Architektonische Umstände:

Metriken wie Instabilität, Abstraktheit und Distanz sind abgeleitete Metriken, die von Robert C. Martin entworfen wurden (Martin, 2003). Diese drei Metriken haben einen engen Zusammenhang und sollten immer zusammen betrachtet werden.

| | |
|---|---|
| Name: | Instabilität (I) |
| Zweck: | Indikator für die Auswirkung von Änderungen, Erweiterbarkeit, Wartbarkeit |
| Formel: | $I = \frac{Ce}{Ce+Ca}$ |
| | I = Instability, Ce = efferent Couplings, Ca = afferent Couplings |
| Beschreibung: | Instabilität beschreibt das Verhältnis zwischen der ausgehenden Abhängigkeit zur Gesamtabhängigkeit, berechnet aus der Summe von Ca und Ce, eines Paketes. Das Ergebnis ist ein Zahlenwert zwischen 0 und 1, dabei ist 1 ein instabiles Paket, welches sehr anfällig für Änderungen anderer Pakete ist und 0 ein so stabiles Paket, dass Änderungen anderer Pakete keine Auswirkung auf diese Paket haben |

| | |
|---|---|
| Name: | Abstraktheit (A) |
| Zweck: | Schutz vor direktem Zugriff, Design, Pattern |
| Formel: | $A = \frac{C+IF}{aC}$ |
| | A = Abstractness, C = Anzahl abstrakter Klassen, IF = Anzahl an Interfaces, aC = Anzahl aller Klassen/Interfaces |
| Beschreibung: | Abstraktheit beschreibt das Verhältnis zwischen abstrakten Klassen und Interfaces zu der Gesamtzahl an Klassen und Interfaces in einem Paket. Das Ergebnis ist ein Zahlenwert zwischen 0 und 1, dabei ist 1 ein absolut abstraktes und 0 ein absolut konkretes Paket. |

| | |
|---|---|
| Name: | Distanz (D) |
| Zweck: | abgeleitete Metrik, welche die Distanz zum idealen Pfad eines Paketes |
| Formel: | $D = 1 - (A + I)$ |
| Beschreibung: | Distanz beschreibt den Abstand der Summe von A und I der betrachteten Einheit zur idealisierte Main Sequence (A+I=1). |

Instabilität zeigt auf, wie stark eine Einheit benutzt wird. Sie ist definiert durch das Verhältnis der ausgehenden Abhängigkeiten zu der Gesamtanzahl von Abhängigkeiten einer Einheit. Liegt der Wert nahe bei eins, so ist diese Einheit instabil, liegt er nahe bei null, ist die Einheit stabil. Instabilität besagt, dass die Einheit viel mehr ausgehende als eingehende Abhängigkeiten besitzt und somit gegenüber Veränderungen anderer Einheiten anfällig ist. Stabile Einheiten haben die Eigenschaft, dass Veränderungen dieser Einheiten Auswirkung auf eine große Anzahl anderer Einheiten hat.

Abstraktheit ist das Verhältnis zwischen der Anzahl an abstrakten Klassen und Interfaces gegenüber der Anzahl aller Klassen. Eine hohe Abstraktheit ist ein Indikator dafür, dass diese Einheit nicht direkt benutzt werden kann und somit eventuell keinen Nutzen hat. Eine niedrige Abstraktheit hingegen ist ein Zeichen für einen direkten Zugriff auf eine Einheit. Direkte Zugriffe sollten durch Interface unterbunden werden.

Die Distanz beschreibt den Abstand der betrachteten Einheit zur idealisierten Main Sequence einer Einheit. Die Main Sequence bezeichnet alle Einheiten, die als Ergebnis der Summe von A und I eins sind.

Folgende Grafik zeigt das die idealisierte Main Sequence:

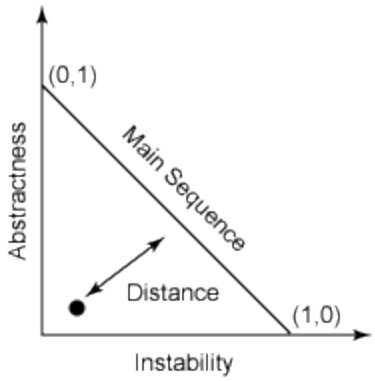



**Abbildung 5: Main Sequence**

Die x-Achse der Abbildung 6 stellt die Instabilität im Intervall 0 bis 1 dar und die y-Achse das Intervall von 0 bis 1 der Abstraktheit. Zwei ideale Einheiten nach Robert C. Martin sind Einheiten mit einer Abstraktheit von 1 und einer Instabilität von 0 und umgekehrt. Einheiten mit A=1 und I=0 sind komplett abstrakte Einheiten, die keine ausgehende Kopplung aufweisen, d.h. nicht abhängig von der Implementierung anderer Einheiten sind. Instabile Einheiten, die von den Änderungen genutzter Einheiten betroffen sein können, empfehlen sich nicht zum Vererben, da die möglichen Änderungen auf die erbende Einheit Auswirkungen haben. Diese Einheiten sollten möglichst konkret implementiert sein.

Diese beiden idealen Zustände einer Einheit haben in der Summe von A und I den Wert eins. Alle Einheiten, deren Summe von A und I als Ergebnis 1 betragen, bewegen sich auf der Main Sequence und sind in Abbildung 6 als Linie eingetragen.

Die Distanzmetrik ist der Abstand der Summe A und I von der Main Sequence und ein Indikator dafür, ob A und I ausgewogen sind. Ist die Distanz 0, liegt das Verhältnis zwischen A und I genau auf der idealisierten Main Sequence. Bei einem Wert gegen 1 ist der maximale Abstand zur Main Sequence. Einheiten mit einer sehr großen Distanz müssen überprüft werden.

Eine andere Metrik namens Change-Rate (CR) benötigt ein Versionsmanagementwerkzeug. Die Wechselrate beschreibt, welche Klassen im Projektverlauf oft geändert wurden. Diese Klassen bedürfen gesonderter Aufmerksamkeit, da diese nicht grundlos häufig verändert wurden. So kann der Einsatzzweck dieser Klasse missverstanden sein (niedrige Kohäsion), also ein architektonischer Mangel vorliegen. Hier sollten auch andere Metriken wie Abstraktheit, Instabilität und die Distanz untersucht werden, da diese Aufschluss über das Verhalten der CR geben können. Es folgt nun die Beschreibung der CR und ein Beispiel einer CR Liste:

| | |
|---|---|
| Name: | Change Rate (CR) |
| Zweck: | Architekturmängel, Verständnis des Einsatzkontextes, Soll- Ist-Vergleich der Architektur |
| Formel: | CR wird durch ein Versionsmanagementwerkzeug ermittelt und sortiert dargestellt. |
| Beschreibung: | Bei der Change-Rate handelt es sich um eine nicht – technische Metrik, da hier betrachtet wird, wie oft eine Klasse in SubVersion geändert wurde. Eine hohe Wechselrate, dargestellt als eine Liste, in der die Klassen mit der höchsten Wechselrate oben steht, ist ein Indiz dafür, dass an dieser Stelle eine architektonischer Mangel vorliegt oder der Einsatzkontext der Klasse |

| FileName | number of times changed |
|---|---|
| ClassInTheMiddle.java | 17 |
| POM.xml | 5 |
| ClassOnTheLeft.java | 2 |
| ClassOnTheBottom.java | 1 |
| ClassOnTheRight | 1 |

**Tabelle 2: Beispiel eines Change-Rate Ergebnisses**

Anhand dieser Tabelle kann man entnehmen, dass ein großes Ungleichgewicht der Änderungsrate der Klasse ClassInTheMiddle.java zu den anderen besteht. Dies stellt ein

auffälliges Verhalten dar und sollte zusammen mit den Entwicklern analysiert und besprochen werden.

### 4.2.4 Stil

Aspekte wie Erweiterbarkeit und Wartbarkeit sind klassische Qualitätskriterien. Besonders im Umfeld von agilen Entwicklungsmethoden und dem daraus resultierenden gemeinsamen Quelltextbesitz aller Entwickler ist der Stil des Quelltextes wichtig. So erleichtert, wie in Abschnitt 2.2.1 beschrieben, ein einheitlicher Programmierstil die Wiederverwendbarkeit innerhalb eines Entwicklerteams. Innerhalb des Entwicklerteams wurden bereits einige Programmierstilregeln definiert, an die sich alle Entwickler halten sollen:

- Klassen müssen mit ihrer Funktion kommentiert sein
- Öffentliche Methoden müssen kommentiert sein
- Parameter sollten kommentiert sein
- JavaDoc Sprache ist Englisch
- SubVersion Sprache ist Deutsch
- Kommentare an Attributen sollen die deutschen Namen der fachlichen Domäne enthalten

Um eine Programmierstilkontrolle in die Qualitätssicherung einfließen zu lassen, werden nun Ansätze und Werkzeuge gesucht, die bei der Erstellung und Einhaltung eines Programmierstils behilflich sind. Dafür gibt es Werkzeuge, welche regelbasiert einen Quelltext durcharbeiten und hierauf die definierten Regeln anwenden. Interessant ist es, ob es nun für Maven2 solche Werkzeuge gibt und wie gut anpassbar diese an die Anforderungen des Kunden sind. Hinzu kommt, dass es sehr wichtig ist, die schon bestehenden Vorgaben des Entwicklerteams umsetzen zu können. Entweder bringen die Werkzeuge schon Regeln mit, die dieses möglich machen, oder es sollte eine Schnittstelle geben, sodass eigene Regeln erstellt und ausgeführt werden können. Aus den angebotenen Werkzeugen werden dann mit dem Entwicklerteam nützlichen Regeln ausgewählt und eingesetzt.

### 4.2.5 Dokumentation

Dokumentation für ein Softwareprojekt umfasst sowohl die Darstellung der geleisteten Arbeit als auch die Beschreibung der erstellten Software. Dokumentation kann in schriftlicher Form erfolgen, z.B. als Anleitung. Ebenfalls sind hier Diagramme, Schaubilder

oder auch Kommentare im Quelltext sowie der Quelltext an sich (Martin, 2003) von Interesse. Das Prüfen dieser Dinge ist jedoch nicht immer maschinell unterstützbar. So kann der Grad der Dokumentation in schriftlicher und grafischer Form nicht automatisiert gemessen werden. Anders sieht es bei der Dokumentation der Quelltexte aus, hier ist ein einheitlicher Stil eventuell unterstützend. Es ist möglich, die Kommentare regelbasiert zu prüfen. So kann definiert werden, dass jede von außen zugreifbare Methode (public) mit einem beschreibenden Kommentar versehen sein muss, und dass Klassen einen Klassenkommentar haben müssen. Wichtig ist zu verstehen, dass regelbasierte und metrikerstellende Werkzeuge meist nur das Vorhandensein oder den Grad einer Abdeckung messen können. Sie können jedoch keine Aussagen über die Qualität der Kommentare treffen. Somit reicht es nicht aus, zu prüfen, ob eine 100% Abdeckung an Kommentaren vorhanden ist, der Inhalt muss weiterhin manuell mittels Reviews untersucht werden. So bleibt an dieser Stelle zu sagen, dass Kommentare, meist JavaDoc, auf Vorhandensein getestet werden können, aber die Qualitätsprüfung der Dokumentation zu einem großen Anteil ein manuelles Vorgehen ist.

### 4.2.6 Management

Im Bereich des Managements stehen nicht-funktionale Metriken und Faktoren im Mittelpunkt, dazu gehören Kosten, Budget und der Fortschritt aber auch Fragen nach der Effizienz und Folgekosten. Solche Größen in Metriken zu fassen, also auf Zahlen abzubilden, ist fast unmöglich oder aussagelos.

Aus den Gesprächen mit dem Kunden ist die Idee entstanden, dass es eine Fortschrittsmetrik geben könnte, welche wie folgt definiert ist:

| | |
|---|---|
| Name: | Fortschritt anhand von Anforderungen (FA) |
| Zweck: | Fortschrittskontrolle, Budgetkontrolle, Anforderungskontrolle |
| Formel: | $FA = \frac{pT}{gT}$<br>pT = positive Anzahl bestandener Tests, gT = Anzahl gesamter Anforderungstests |
| Beschreibung: | Für diese Metrik müssen die Anforderungen als Test modelliert werden. Wird ein Test bestanden, gilt die Anforderung als realisiert. Änderungen an den Anforderungen müssen entsprechend eingefügt werden. |

Hierfür müssten wie beschrieben alle bekannten Features vorliegen und in eine testbare Form gebracht werden. Es muss zu geprüft werden, ob es Werkzeuge gibt, mit der die FA Metrik erstellt werden kann.

Use-Cases zählen auch zu den nicht-funktionalen Metriken der Managementkategorie in diesem Qualitätsmodell. Hierbei geht es nicht um eine Deckungsrate von Use-Cases pro Modul oder ähnliches, vielmehr soll die Validierung bestehender Use-Cases möglich gemacht werden. Dafür werden vom Entwicklerteam sog. Oberflächentests (Selenium, 2006) ausgeführt. Das Softwareprojekt des Kunden basiert auf Webtechnologien. Oberflächentests ermöglichen hier den automatisierten Ablauf von Use-Cases durch eine Webseite. Solche Tests sind bereits vorhanden und sind nicht weiter Thema dieser Arbeit.

### 4.2.7 ABC-Analyse

An dieser Stelle werden die in diesem Abschnitt erläuterten Metriken zusammengefasst und gewichtet. Viele Aspekte sind maschinell erfassbar, aber nicht alle. Andere wiederum sind bereits im Einsatz. Aus der folgenden Tabelle ist entnehmbar, welche Anforderungen an Metriken für diese Einführung einer Qualitätssicherung mittels Maven2 gelten.

| Metrik | automatisierbar | must | should | nice to have | Aussagekraft |
|---|---|---|---|---|---|
| Fehler | ja | X | | | + |
| Testabdeckung | ja | X | | | + |
| Kopplung | ja | X | | | + |
| Non Commenting Source Statements | ja | | X | | + |
| Lines of Code | ja | | X | | + |
| Number of Classes | ja | | X | | + |
| Abstraktheit | ja | X | | | + |
| Instabilität | ja | X | | | + |
| Distanz | ja | X | | | + |
| Change Rate | ja | | | X | (+/-) |
| Eigener Stil | ja | | X | | + |
| Stilmittel | ja | X | | | + |
| schriftliche Dokumentation | nein | - | - | - | - |
| JavaDoc | ja | | X | | + |
| UML | nein | - | - | - | - |
| Fortschritt | ja | | | X | + |
| Budget | ja/nein | | | X | - |
| Produktivität | ja/nein | | | X | - |

**Tabelle 3: ABC-Analyse**

# 5 Umsetzung

## 5.1 Marktlage & Tests

In diesem Abschnitt werden die Ergebnisse der Recherchearbeit beschrieben, und zwar welche Werkzeuge Maven2 im Kontext einer Qualitätssicherung zur Verfügung stehen. Da es über Maven2 kaum Literatur gibt, vgl. jedoch (sonatype, 2008), beschränkte sich die Recherche zunächst auf die Webseiten des Anbieters Apache (Group, 2008). Dort findet sich eine Auflistung an verfügbaren Plug-Ins. An dieser Stelle der Webseite wird auch auf eine weitere Quelle für Plug-Ins der Codehaus Community (Mojo C. C., 2007) aufmerksam gemacht. Bei diesen Quellen fanden sich einige Plug-Ins, welche die unter 4.2 herausgearbeiteten Anforderungen erfüllen könnten. Dazu wurden die folgende Plug-Ins getestet: Change-Log, CheckStyle, Clover, JXR, PMD/CPD, SureFire-Report, JDepend, FindBugs, JavaNCSS und Cobertura.

Die meisten Plug-Ins sind erstellt worden, um schon bestehende Werkzeuge in die Maven2 Architektur zu integrieren. Beispiele hierfür sind CheckStyle (Burn, 2007) und PMD (PMD, 2007). Beide Werkzeuge sind ursprünglich nicht als Maven2 Plug-In erstellt worden, sondern als allein stehende Projekte für qualitätssichernde Maßnahmen.

Alle betrachteten Plug-Ins sind Open Source Produkte. Lediglich ein Plug-In namens Clover, welches die Testabdeckung misst, ist ein kostenpflichtiges Werkzeug. Die Quelltexte für die Werkzeuge sind verfügbar und anpassbar. Die Dokumentation der einzelnen Plug-Ins unterscheidet sich sehr. Festzustellen ist, dass Plug-Ins einen Einstieg ermöglichen, aber Feineinstellungen nicht dargestellt werden. Das hat zu Folge, dass weitere Recherchen auf den Seiten der Anbieter der Produkte vorzunehmen sind.

Bei der Codehaus Community werden die einzelnen Plug-Ins getestet und je nach Stabilitätsstand eingestuft. Dies ermöglicht dem Anwender eine ungefähre Einstufung, wie detailliert ein Plug-In dokumentiert ist oder wie lange es dieses schon gibt.

Nun folgen die ersten Tests vorhandener Plug-Ins unter den Kriterien der unter 4.2 erarbeiteten Anforderungen.

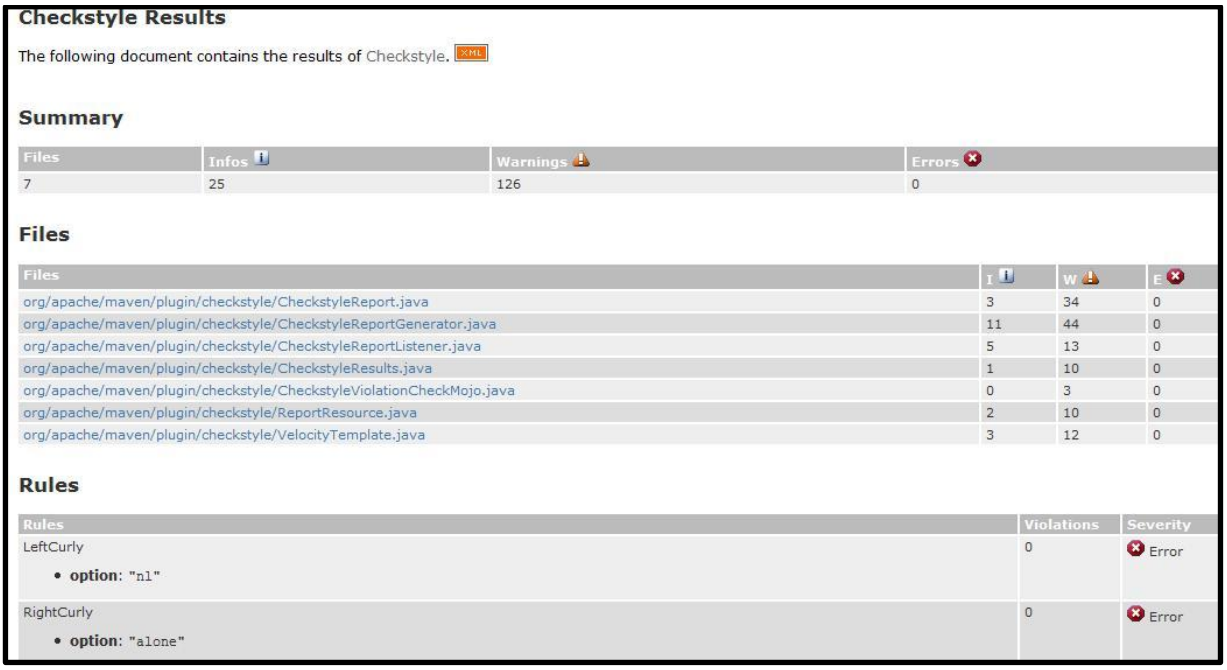
### 5.1.1 CheckStyle

CheckStyle ist ein Werkzeug, das es Entwicklern ermöglicht, den Programmierstil eines Quelltextes zu prüfen. Die Grundlage für solch eine Prüfung ist der beinhaltete Regelsatz. Dieser Regelsatz ist gut auf der Herstellerseite (Burn, 2007) dokumentiert. Das Maven2 Plug-In für CheckStyle nutzt eine XML Datei für die Konfiguration der Regeln. In der XML Datei werden die Regelsätze in Kategorien eingeteilt und können von Regel zu Regel als Fehler, Warnung oder Information klassifiziert werden.

Sowohl alle Kategorien von CheckStyle, als auch die der anderen Plug-Ins aufzuführen, würde dieses Kapitel mit zu vielen Informationen füllen, welche nicht die eigentliche Handlung wiederspiegeln. Deshalb findet sich im Anhang eine Liste der gewählten Metriken von CheckStyle, PMD und FindBugs.

Neben Kategorien für den Programmierstil bietet CheckStyle auch Softwaremetriken (z.B. CCN, NCSS) an.

Laut Herstellerseite ist das Definieren eigener Programmierstilregeln möglich. Diese Funktion kann für diese Arbeit von Interesse sein, um eigene Regeln zusammen mit dem Entwicklerteam einzuführen.



**Checkstyle Results**

The following document contains the results of Checkstyle.

**Summary**

| Files | Infos | Warnings | Errors |
|---|---|---|---|
| 7 | 25 | 126 | 0 |

**Files**

| Files | I | W | E |
|---|---|---|---|
| org/apache/maven/plugin/checkstyle/CheckstyleReport.java | 3 | 34 | 0 |
| org/apache/maven/plugin/checkstyle/CheckstyleReportGenerator.java | 11 | 44 | 0 |
| org/apache/maven/plugin/checkstyle/CheckstyleReportListener.java | 5 | 13 | 0 |
| org/apache/maven/plugin/checkstyle/CheckstyleResults.java | 1 | 10 | 0 |
| org/apache/maven/plugin/checkstyle/CheckstyleViolationCheckMojo.java | 0 | 3 | 0 |
| org/apache/maven/plugin/checkstyle/ReportResource.java | 2 | 10 | 0 |
| org/apache/maven/plugin/checkstyle/VelocityTemplate.java | 3 | 12 | 0 |

**Rules**

| Rules | Violations | Severity |
|---|---|---|
| LeftCurly<br>• **option**: "nl" | 0 | Error |
| RightCurly<br>• **option**: "alone" | 0 | Error |

**Abbildung 6: Checkstyle Auszug**

Einschätzung: gute Dokumentation, für dieses Projekt einsetzbar

| | |
|---|---|
| Zweck: | Programmierstilkontrolle und einzelne Metriken |
| Konfiguration: | gute, einfache XML-Datei, gute Dokumentation auf Herstellerseite |
| Metriken für diese Arbeit: | NCSS, CCN, Programmierstil, eigener Programmierstil |

### 5.1.2 PMD

PMD (PMD, 2007) ist im Umfeld der Qualitätssicherung mittels Maven2 ähnlich einzuordnen wie CheckStyle. Das Maven2 Plug-In kontrolliert hauptsächlich den Programmierstil regelbasiert und liefert auch einzelne Metriken. Die Konfigurationsmöglichkeiten beruhen auf Regelwerken, die in Kategorien eingeordnet sind. In einer XML-Konfigurations-Datei werden ganze Kategorien hinzugefügt, oder nur einzelne Teile der jeweiligen Kategorie gewählt oder ausgeschlossen.

Eine Unterscheidung der gefundenen Schwellwertüberschreitungen durch verschiedene Überschriften wie „Fehler“, „Warnung“ oder „Information“, ist mit PMD nicht möglich.

CheckStyle und PMD haben eine Anzahl von überschneidenden Regeln, sodass diese nicht doppelt untersucht werden müssen. Da das Softwareprojekt des Kunden ein großes Volumen umfasst, spart dieser Schritt der Optimierung Zeit bei der Erstellung der Qualitäts-sicherung. PMD wurde entsprechend so eingestellt, dass alle Regeln deaktiviert wurden, welche von CheckStyle bereits behandelt werden.

PMD birgt ein weiteres Programm namens CPD. CPD steht für Copy and Paste Detector. Mittels dieses Programms ist es möglich duplizierten Quelltext zu erkennen. Duplizierter Quelltext ist zu vermeiden, da durch ihn die Wartung und Erweiterbarkeit erschwert wird. Es kann nicht vorausgesetzt werden, dass jeder Entwickler weiß, wo sich Duplikate eines Quell-textstückes befinden, sodass bei Änderungen die Duplikate nicht gleichermaßen geändert werden. Durch das Prinzip der Vererbung und der Abstraktion moderner objektorientierter Programmiersprachen können duplizierte Quelltextstücke in eine Oberklasse abstrahiert werden. CPD wird mit einem Schwellwert versehen, welcher ausdrückt, wie viele Zeilen Quelltext dupliziert sein müssen, bevor ein Alarm ausgegeben wird.

Laut Hersteller bietet PMD auch wie CheckStyle eine Funktion an, eigene Programmierstilregeln zu definieren.

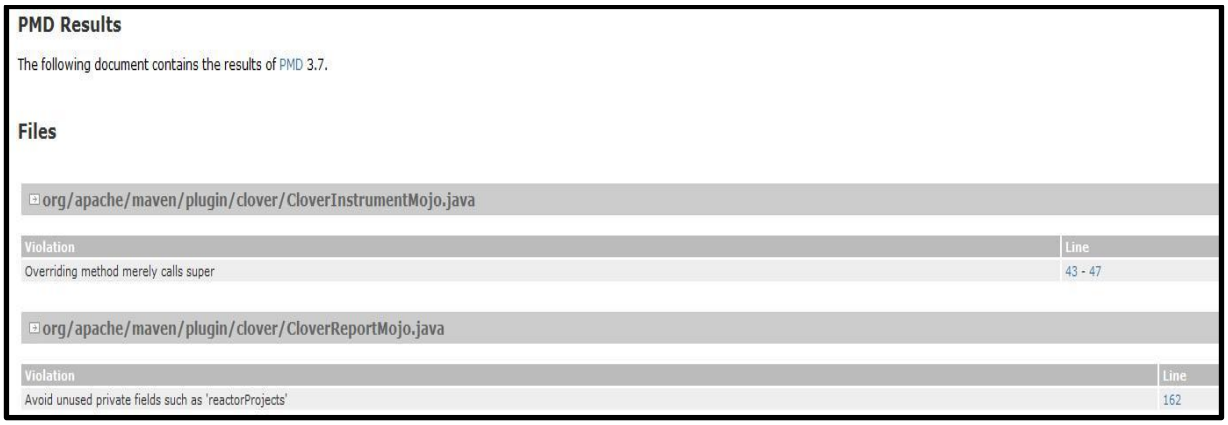



**Abbildung 7: PMD Auszug**

Fazit: ähnliche wie CheckStyle, für dieses Projekt nutzbar

Zweck: Programmierstilkontrolle, Metriken

Konfiguration: gut, gleiches Prinzip wie CheckStyle

Metriken für diese Arbeit: CCN, Programmierstil, eigener Programmierstil

### 5.1.3 JXR

JXR (JXR, 2007) ist ein Plug-In für Maven2, das die Quelltexte mit Zeilennummer auf der Projektwebseite darstellt. Hierdurch können andere Plug-Ins eventuelle Fehler gleich am Quelltext aufzeigen. JXR wird von Plug-Ins, welche diese benötigen oder nutzen, nicht selbstständig eingefügt, sodass es sich empfiehlt, das JXR Plug-In mit aufzunehmen.

Fazit: erhöht Übersicht und Verständnis, wenn Quelltexte angezeigt werden können

Zweck: Darstellung von Quelltexten auf Projektwebseite

Konfiguration: lediglich in POM.XML eintragen, andere Plug-Ins nutzen dieses

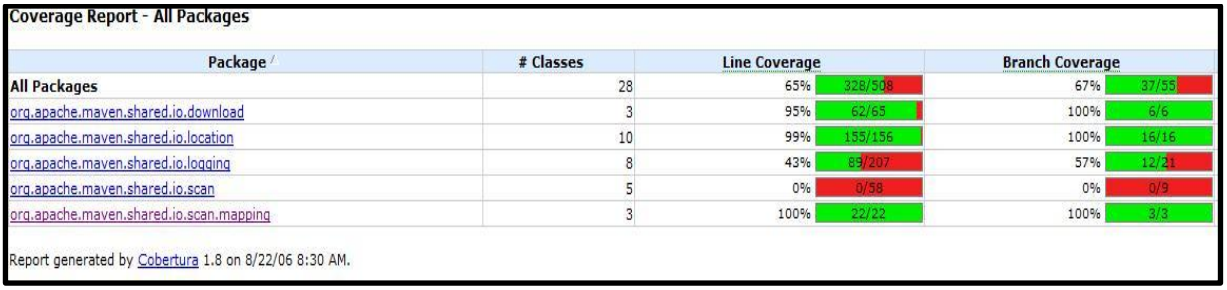
Metriken für diese Arbeit: keine, verbesserte Darstellung anderer Plug-Ins

### 5.1.4 SureFire und Cobertura

Das SureFire-Plug-In (Surefire, 2007) führt JUnit-Tests in Maven2 Projekten durch und gehört zu den Basis-Plug-Ins, die nach der Installation von Maven2 automatisch verankert sind. SureFire kann einen Report erstellen, welcher auf der Projektwebseite veröffentlicht wird. In dem Report ist eine Statistik darüber enthalten, in welcher Zeit die Tests durchgeführt

wurden. In der Statistik gibt es eine Übersicht über das Verhältnis von durchgeführten zu fehlgeschlagenen Tests in Prozent.

Cobertura (Cobertura, 2007) misst die Testabdeckung eines Projektes. Dabei unterscheidet Cobertura zwischen branch und line coverage. Line coverage zählt die Anzahl von Quelltextreihen welche in JUnit Tests durchlaufen werden und stellt diese ins Verhältnis zur Gesamtzahl verfügbarer Quelltextreihen. Bei der daraus resultierenden Prozentzahl spricht man von der Testabdeckung. Eine weitere Testabdeckung ist auf die möglichen Verzweigungen, also if oder switch Anweisungen, bezogen. Dies wird als *branch coverage* bezeichnet. *Branch coverage* erreicht den Wert von 100%, sobald jede mögliche Verzweigung in einer Einheit durch einen oder mehrere JUnit Tests abgedeckt ist.



Coverage Report - All Packages

| Package | # Classes | Line Coverage | Branch Coverage |
|---|---|---|---|
| All Packages | 28 | 65% 328/508 | 67% 37/55 |
| org.apache.maven.shared.io.download | 3 | 95% 62/65 | 100% 6/6 |
| org.apache.maven.shared.io.location | 10 | 99% 155/156 | 100% 16/16 |
| org.apache.maven.shared.io.logging | 8 | 43% 89/207 | 57% 12/21 |
| org.apache.maven.shared.io.scan | 5 | 0% 0/58 | 0% 0/9 |
| org.apache.maven.shared.io.scan.mapping | 3 | 100% 22/22 | 100% 3/3 |

Report generated by Cobertura 1.8 on 8/22/06 8:30 AM.

**Abbildung 8: Cobertura Report**

Fazit: kann eingesetzt werden

Zweck: TC Bestimmung und Statistik über Testverlauf

Konfiguration: kaum Einstellungsmöglichkeiten

Metriken dieser Arbeit: Testabdeckung

### 5.1.5 Clover

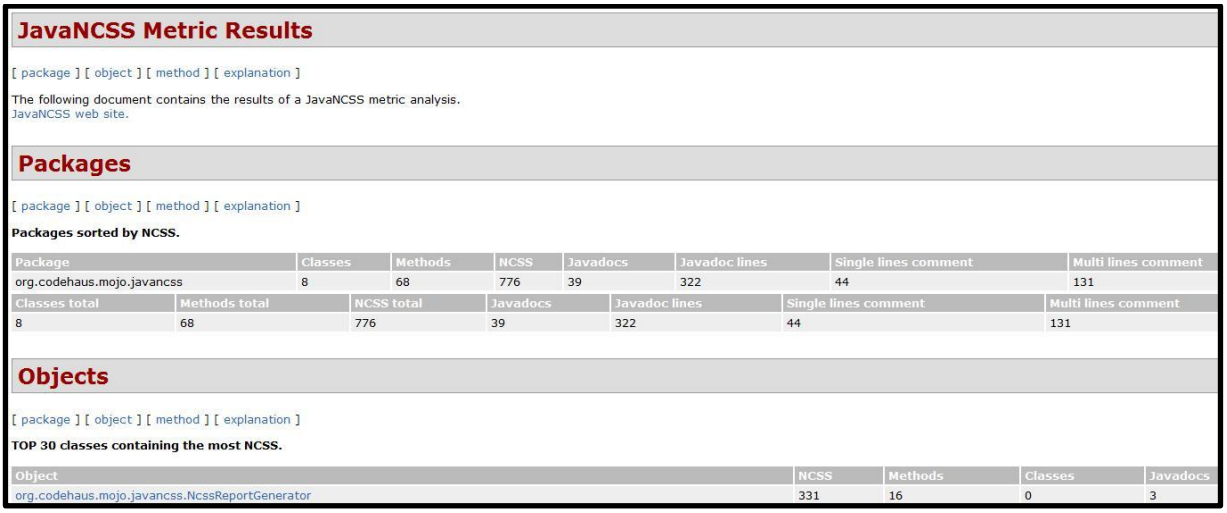
Clover ist ein Werkzeug um die Testabdeckungsmetrik (TC) zu bestimmen. Clover steht nicht unter einer freien Lizenz und ist laut Spezifikation von der Funktion her ähnlich wie Cobertura. Da Cobertura unter einer freien Lizenz steht, wurde sich für dieses entschieden und Clover nicht weiter verfolgt.

### 5.1.6 JavaNCSS

Das JavaNCSS Plug-In für Maven2 (JavaNCSS, 2007) ermittelt Volumen- und Komplexitätsmetriken für Javaquelltexte. Für die einzelnen Metriken können Schwellwerte definiert werden.

Dieses Plug-In besitzt keine XML-Konfigurationsdatei, stattdessen werden Parameter beim Aufruf von JavaNCSS benutzt um Schwellwerte zu definieren. Eine Konfigurationsdatei ist wünschenswert, damit die Einstellungen an einem zentralen Ort durchgeführt und gespeichert werden. Da Parameter beim Aufruf missachtet oder vergessen werden können, besteht das Risiko, dass definierte Schwellwerte von JavaNCSS außer acht gelassen werden. Einstellungen über das Verhalten bei der Ausführung werden in dem POM des Software-projektes definiert.

Im Gegensatz zu anderen Plug-Ins verfügt JavaNCSS über einen eigenen Aggregationsmodus, d.h. ermittelte Informationen von Submodulen werden geballt auf einer Webseite dargestellt. Weiter Ansätze für Aggregationen werden im Abschnitt 5.2 diskutiert.



**JavaNCSS Metric Results**

[ package ] [ object ] [ method ] [ explanation ]

The following document contains the results of a JavaNCSS metric analysis.
JavaNCSS web site.

**Packages**

[ package ] [ object ] [ method ] [ explanation ]

**Packages sorted by NCSS.**

| Package | Classes | Methods | NCSS | Javadocs | Javadoc lines | Single lines comment | Multi lines comment |
|---|---|---|---|---|---|---|---|
| org.codehaus.mojo.javancss | 8 | 68 | 776 | 39 | 322 | 44 | 131 |

| Classes total | Methods total | NCSS total | Javadocs | Javadoc lines | Single lines comment | Multi lines comment |
|---|---|---|---|---|---|---|
| 8 | 68 | 776 | 39 | 322 | 44 | 131 |

**Objects**

[ package ] [ object ] [ method ] [ explanation ]

**TOP 30 classes containing the most NCSS.**

| Object | NCSS | Methods | Classes | Javadocs |
|---|---|---|---|---|
| org.codehaus.mojo.javancss.NcssReportGenerator | 331 | 16 | 0 | 3 |

**Abbildung 9: JavaNCSS Auszug**

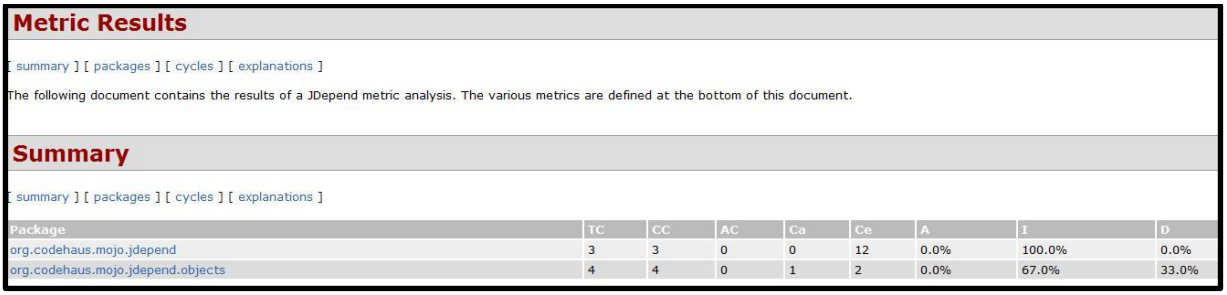
| | |
|---|---|
| Zweck: | Volumen- und Komplexitätsmetikenbestimmung |
| Konfiguration: | keine Konfigurationsdatei, Parameter bei Aufruf, Durchführungsverhalten in POM definierbar |
| Fazit: | für diese Arbeit einsetzbar |
| Metriken dieser Arbeit: | NC, NCSS, JavaDoc, CCN |

### 5.1.7 JDepend

JDepend (JDepend, 2007) misst verschiedene Metriken auf Paketebene. Zu den ermittelbaren Metriken gehören die Anzahl von Einheiten, zyklomatische Komplexität, ausgehende Kopplung, eingehende Kopplung, Abstraktheit, Instabilität und Distanz. Somit

werden durch dieses Werkzeug viele der in den Anforderungen herausgearbeiteten Designmetriken ermittelt. Des Weiteren kann JDepend Zyklen zwischen Einheiten eines Softwareprojektes ermitteln.

Leider stellt das Plug-In von JDepend für Maven2 keine Einstellungsmöglichkeiten bereit. Alle Einstellungen sind als Standard festgelegt und es gibt keine Schnittstelle oder dokumentierte Erweiterungen, um weitere Konfigurationsarbeiten durchzuführen.



**Metric Results**

[ summary ] [ packages ] [ cycles ] [ explanations ]

The following document contains the results of a JDepend metric analysis. The various metrics are defined at the bottom of this document.

**Summary**

[ summary ] [ packages ] [ cycles ] [ explanations ]

| Package | TC | CC | AC | Ca | Ce | A | I | D |
|---|---|---|---|---|---|---|---|---|
| org.codehaus.mojo.jdepend | 3 | 3 | 0 | 0 | 12 | 0.0% | 100.0% | 0.0% |
| org.codehaus.mojo.jdepend.objects | 4 | 4 | 0 | 1 | 2 | 0.0% | 67.0% | 33.0% |

**Abbildung 10: JDepend Auszug**

| | |
|---|---|
| Zweck: | Designmetriken ermitteln |
| Konfiguration: | keine Einstellungsmöglichkeiten |
| Fazit: | für diese Arbeit sehr nützlich aufgrund der bestimmbaren Metriken |
| Metriken dieser Arbeit: | CCN, Ca, Ce, A, I, D, NC |

### 5.1.8 FindBugs

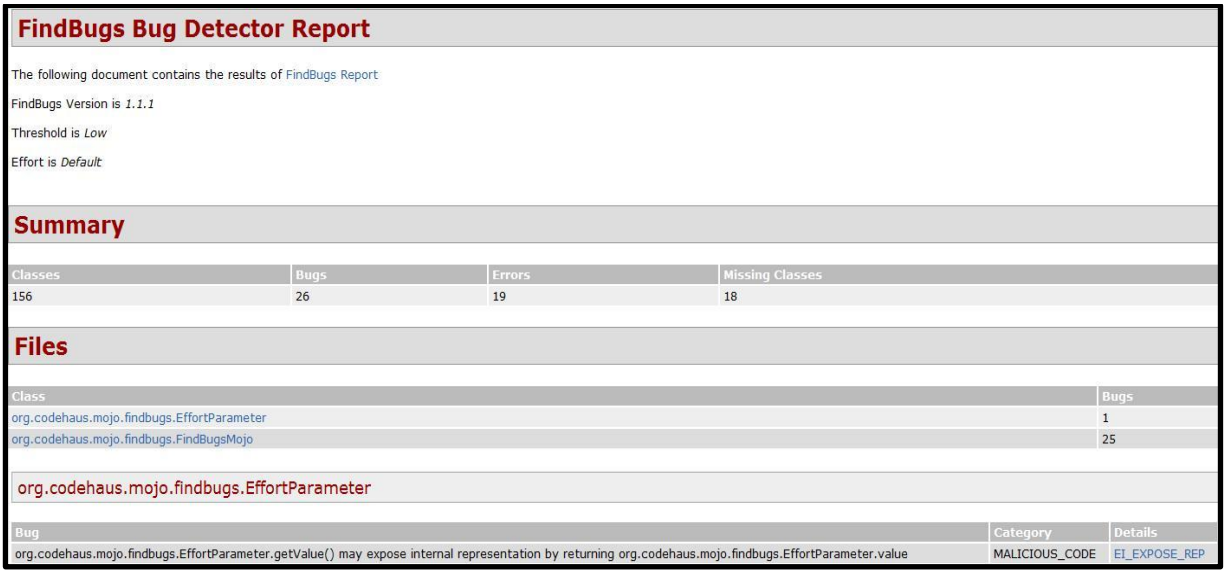
FindBugs (FindBugs, 2007) ist ein Werkzeug, welches via BugPattern nach Fehlern in Java Quelltexten sucht. Ein BugPattern ist eine Beschreibungsweise für einen häufigen Fehler. Diese Pattern werden direkt auf den Bytecode von Javaprogrammen angewandt.

Laut den Autoren von FindBugs sind mehr als 50% der gefundenen Fehler wirkliche Fehler. BugPattern auf eine Einheit angewandt, lösen häufig sog. Falsch-Positiv-Meldungen aus. Dieses sind keine Fehler, sondern Einheiten eines Softwareprojektes, welche zufällig zu den BugPattern passen.

FindBugs sortiert die verschiedenen BugPattern, ähnlich wie CheckStyle und PMD, ebenfalls in Kategorien, welche auch in einer XML-Konfigurationsdatei deklariert sind. Die Dokumentation dieser Kategorien, bzw. deren Namen, waren im Zuge dieser Recherche

nicht auffindbar. Um einzelnen Kategorien Pattern hinzuzufügen oder auszuschließen, müssen die Namen jedoch explizit angegeben werden. Die einzelnen Kategorien können dann mit einem Zahlenwert versehen werden, welche die Gewichtung bzw. die Wichtigkeit darstellt.

Die Kategorien des Plug-Ins sind derzeitig nicht konfigurierbar, aber es ist möglich, auf die Durchführung von FindBugs Einfluss zu nehmen. Das Verhalten FindBugs bei der Durchführung wird in der POM (Project Object Model (POM)vgl. Seite 16) eines Maven2 Projektes beschrieben. So kann ein *threshold* definiert werden, welcher angibt, ob auch Fehler gemeldet werden sollen, welche in ihrer Einstufung niedriger als der gesetzte *threshold* Wert sind. Ein weiterer Wert ist der *effort*, über diesen kann man in drei Stufen das Verhältnis zwischen Geschwindigkeit, Speicherverbrauch und Präzision wählen. Der *threshold* und der *effort* sind nur zwei Beispiele, weitere Konfigurationsmöglichkeiten sind auf der Herstellerseite (FindBugs, 2007) dokumentiert. Somit bietet FindBugs zu diesem Stand Potenzial, die Kategorie der Fehler in dem unter 4.2 erdachten Qualitätsmodell abzudecken.



**FindBugs Bug Detector Report**

The following document contains the results of FindBugs Report

FindBugs Version is *1.1.1*

Threshold is *Low*

Effort is *Default*

**Summary**

| Classes | Bugs | Errors | Missing Classes |
|---|---|---|---|
| 156 | 26 | 19 | 18 |

**Files**

| Class | Bugs |
|---|---|
| org.codehaus.mojo.findbugs.EffortParameter | 1 |
| org.codehaus.mojo.findbugs.FindBugsMojo | 25 |

org.codehaus.mojo.findbugs.EffortParameter

| Bug | Category | Details |
|---|---|---|
| org.codehaus.mojo.findbugs.EffortParameter.getValue() may expose internal representation by returning org.codehaus.mojo.findbugs.EffortParameter.value | MALICIOUS_CODE | EI_EXPOSE_REP |

**Abbildung 11: FindBugs Auszug**

| | |
|---|---|
| Zweck: | Fehler aufdecken |
| Konfiguration: | Kategorien aufgrund schlechter Dokumentation nicht einstellbar, Verhalten des Plug-Ins ist in dem POM des Projektes definierbar |
| Fazit: | im Rahmen dieser Arbeit einsetzbar |

Metriken dieser Arbeit: verdeckte Fehler

### 5.1.9 Fazit der einzelnen Plug-Ins

Nachdem nun die vorhanden Plug-Ins der Anbieter Apache und Codehaus auf ihre Funktion hin getestet wurden, lässt sich festhalten, dass in den Bereichen Fehler, Design und Stil scheinbar Resultate erreichbar sind. Einige Plug-Ins lassen sich zusätzlich um eigene Regeln erweitern. Es folgt eine Tabelle, die eine Übersicht über die Abdeckung der Metriken dieser Arbeit darstellt:

| Metriken dieser Arbeit | Werkzeuge |
|---|---|
| Testabdeckung | Cobertura, SureFire |
| Non Commenting Source Statement | CheckStyle, JavaNCSS |
| eingehende Kopplung | JDepend |
| ausgehende Kopplung | JDepend |
| zyklomatische Komplexität | CheckStyle, PMD, JavaNCSS, JDepend |
| Distanz | JDepend |
| Instabilität | JDepend |
| Abstraktheit | JDepend |
| Wechselrate | ChangeLog |
| eigener Programmierstil | CheckStyle, PMD |
| Standardprogrammierstil | CheckStyle, PMD |
| JavaDoc | JavaNCSS, |
| NC | JDepend, JavaNCSS |
| Fortschritt anhand von Anforderungen | - |
| Fehler | FindBugs |

**Tabelle 4: Metrikenabdeckung**

## 5.2 Aggregation

Gesucht ist ein Werkzeug, welches die ermittelten Daten der einzelnen Projekte so darstellt, dass sie durch das Entwicklerteam schnell und effizient ausgewertet werden können. Wünschenswert ist die Möglichkeit, gezielt Daten abfragen zu können, ebenso ist eine Datenhaltung nötig, die einen Verlauf von Metriken anzeigbar macht. Hierdurch entsteht eine Historie über den bisherigen Projektverlauf, mit dem die Entwicklung nachvollzogen und Anzeichen für Mängel beobachtet und beseitigt werden können. Maven2 bietet zum derzeitigen Stand keine direkte Unterstützung für das Aggregieren von Daten aus dem

Seitengeneierungslebenszyklus, diese ist erst in folgenden Versionen geplant. Die Recherche für geeignete Werkzeuge ergab drei Ergebnisse, welche nun vorgestellt werden.

### 5.2.1 Sonar

Die Schweizer Firma Hortis (Hortis, 2008) bietet ein Werkzeug namens Sonar unter freier Lizenz an, welche die Daten von SureFire, Cobertura, JavaNCSS, PMD, CheckStyle und ChangeLog zu einer Projektwebseite zusammenfasst. Sonar stellt sich selbst als Plug-In zur Verfügung, jedoch müssen einige Vorbedingungen erfüllt sein, um Sonar auszuführen. Zu diesen gehören:

- JDK 5
  Wird für die Ausführung des JAVA Code von Sonar benötigt. Der Stand des Projektes des Kunden benutzt jedoch JDK 1.4.
- Maven 2.0.X
- JRuby 1.1
  Wird derzeit nicht im Entwicklerteam eingesetzt, müsste also integriert werden.
- MySQL
  Dient als Datenspeicher für die ermittelten Daten und ermöglicht eine Metrikenverlaufsfunktion. Auf jedem Rechner ist MySQL 4.X vorhanden.

Sonar nutzt einen eigenen Serverdienst um die Funktionen anzubieten. Darin sind sowohl die Anbindung der Datenbank, als auch die Konfiguration enthalten. Somit ist hier keine hohe Integration in Maven2 vorhanden, da neben Maven2 immer der Serverdienst gestartet sein muss. Sonar ermöglicht die Konfiguration der einzelnen Plug-Ins über die erstellte Webseite, dies ist komfortabel und erhöht die Übersichtlichkeit. Es werden auch Diagramme erzeugt, welche die Architekturschichten abbilden, darin wird die Fehlerrate anhand einer Farbdarstellung gezeigt.

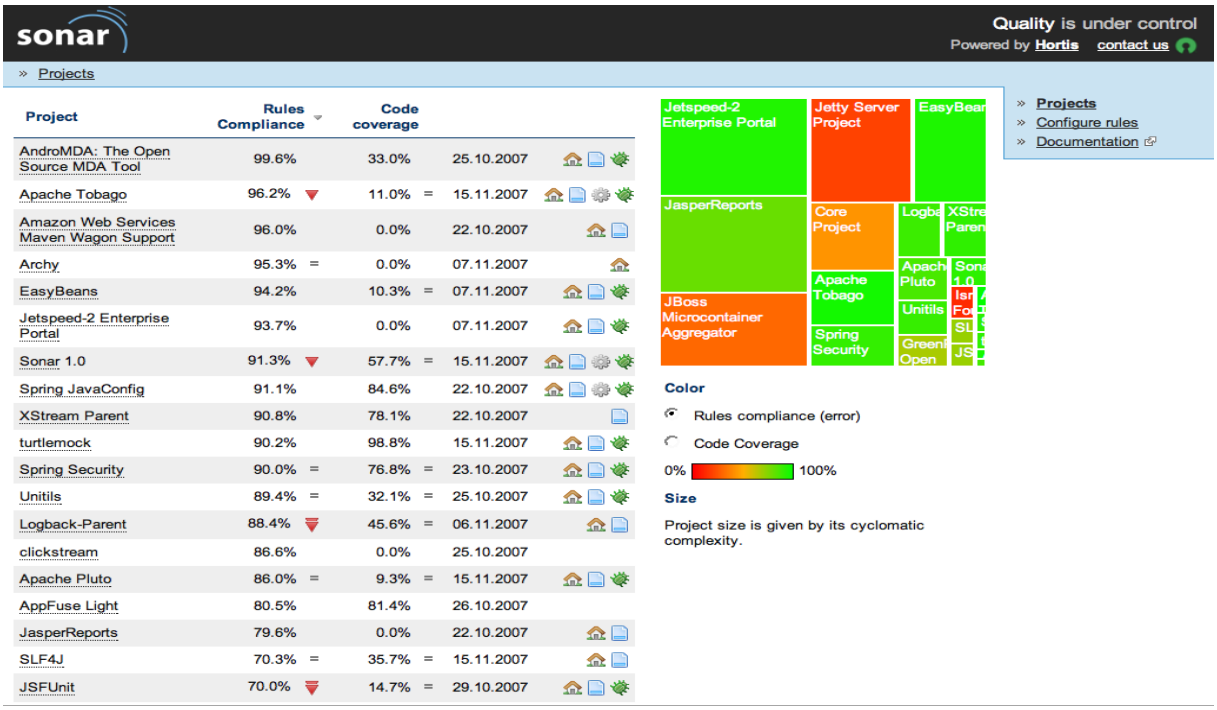



**Abbildung 12: Sonar im Einsatz**

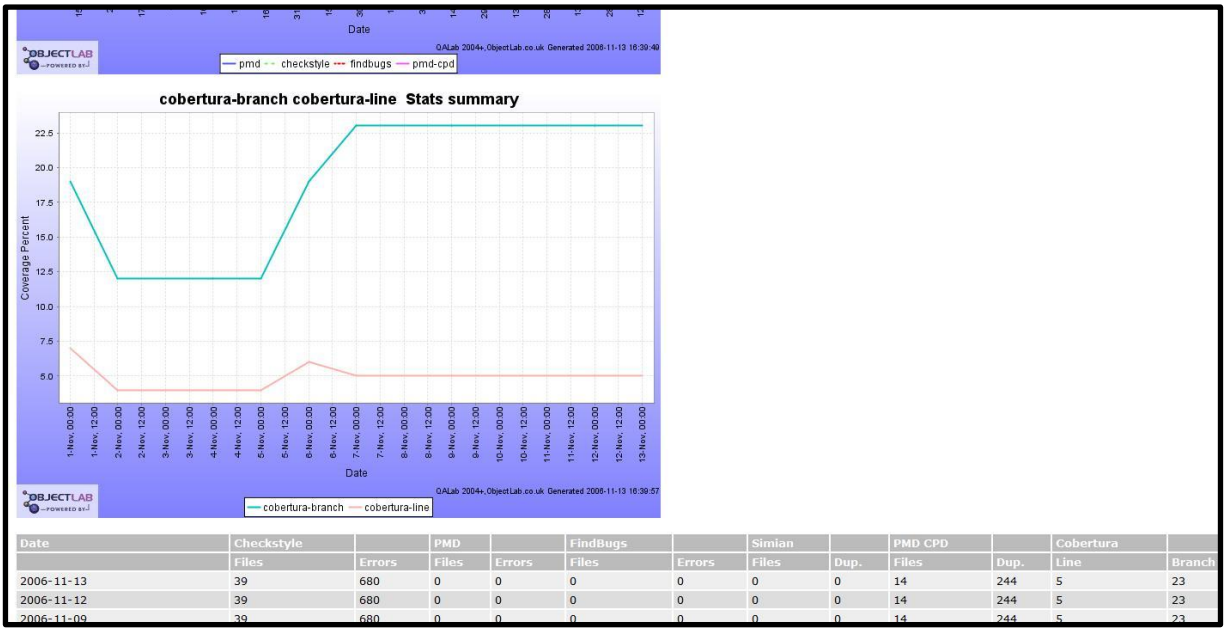
Diese Darstellung ist der Webseite des Anbieters entnommen und stellt ein Beispielprojekt dar. An dieser kann man sehen, wie die einzelnen Projekte anhand ihrer Fehlerquote sortiert und angezeigt werden. Sonar bietet noch mehr Möglichkeiten der Darstellung, so kann z.B. eine Übersicht über jedes einzelne Projekt aufgerufen werden.

Zum Stand der Recherche befand sich Sonar noch in einem Beta Stadium und stellte die Informationen aufgelistet dar. Zwar waren hier auch die Daten enthalten, es mangelte jedoch an Übersicht. Dieser Aspekt und der Einsatz eines ständig laufenden Serverdienstes, in welchem zum damaligen Zeitpunkt die einsetzbare Datenbank enthalten war, begründeten diese Lösung nicht einzusetzen.

### 5.2.2 QAlab

QAlab (QALab, 2006) bietet ebenfalls unter freier Lizenz ein Plug-In an, welches die Plug-Ins PMD, CheckStyle, Cobertura, FindBugs und Simian (ein Plug-In ähnlich dem CPD von PMD) ausführt und entsprechend anzeigt. Ein erster Test zeigte, dass auch hier die Informationen aufgelistet dargestellt werden, jedoch um Verlaufsdiagramme erweitert. Die Diagramme sind nützlich, da hier bildlich Probleme gezeigt werden und diese schnell erfassbar sind. Die Konfiguration von QAlab ist aufwändig, denn für Multi-Projekt-Strukturen muss ein vom Anbieter bereitgestellter Workaround aufgrund eines Fehlers in Maven2 angewandt werden.

Hier nun ein Bild zur Verdeutlichung der Abbildung der Daten:




| Date | Checkstyle | | PMD | | FindBugs | | Simian | | PMD CPD | | Cobertura | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Files | Errors | Files | Errors | Files | Errors | Files | Dup. | Files | Dup. | Line | Branch |
| 2006-11-13 | 39 | 680 | 0 | 0 | 0 | 0 | 0 | 0 | 14 | 244 | 5 | 23 |
| 2006-11-12 | 39 | 680 | 0 | 0 | 0 | 0 | 0 | 0 | 14 | 244 | 5 | 23 |
| 2006-11-09 | 39 | 680 | 0 | 0 | 0 | 0 | 0 | 0 | 14 | 244 | 5 | 23 |

**Abbildung 13: QALab Auszug**

QALab bietet die interessante Funktion, eine „Movers“ Liste zu führen. Dabei handelt es sich um eine Sortierung, bei denen Projekte, in denen die angebliche Fehlerrate steigt, je nach Anzahl sortiert sind. Somit sind Projekte mit den meisten Fehlern oder Verstößen gegen Regeln an der Spitze und stellen somit das derzeitige Hauptaugenmerk der Qualitäts-sicherung dar. Dennoch ist die aufgelistete Darstellungsform nach wie vor nicht wünschenswert, da hier die auswertende Person mit einer Zahlenkolonne konfrontiert wird, die keinen genauen Überblick ermöglicht.

### 5.2.3 Dashboard

Dashboard (Dashboard, 2007) ist ein Open Source Projekt, welches sich zum Ziel gemacht hat, die ermittelten Daten von CheckStyle, PMD, SureFire, FindBugs, JDepend und Cobertura zentral verfügbar zu machen und grafisch zu unterlegen. Es unterstützt standardmäßig aus Multi-Projekt-Strukturen und benötigt zur Ausführung mit History-Funktion eine Datenbank. Ansonsten ist dieses Plug-In voll in Maven2 integrierbar. In dem POM wird Dashboard deklariert und somit ist es bei entsprechender Projektstruktur für alle Submodule verfügbar. Die Daten werden in verschiedenen Detailstufen anzeigbar. So sind die jeweiligen ausgeführten Plug-Ins auf der Startseite des Dashboard in Diagrammen dargestellt. Dabei werden nicht die einzelnen Module, wie in allen anderen Werkzeugen, dargestellt, sondern die Anzahl der Fehler und die prozentuale Verteilung nach Plug-Ins. In der nächsten Detailstufe sind die einzelnen Module in den verschiedenen Plug-Ins angezeigt und werden

mittels Balken zu besseren Vergleichbarkeit grafisch hervorgehoben. Aus dieser zweiten Detailstufe kann in die Ansicht der Submodule gewechselt werden, welche noch detailierte Informationen der Einheiten ermöglicht. Durch diese Art von „Zoom“ können Ergebnisse der Qualitätssicherung bis zum Quelltext der Einheiten repräsentiert werden.

Ein besonderes Augenmerk liegt auf der Verlaufsfunktion des Dashboard, denn diese ermöglicht für jedes der eingesetzten Plug-Ins einen Verlauf über die Zeit anzuzeigen. Die Zeitachse kann von Minuten bis zu Monaten gewählt werden. Das Dashboard selbst ist in dem Maven2 Projekt zu deklarieren, welches das oberste Projekt der Maven2 Struktur ist (vgl. Seite 16). Hier wird definiert, wie Datenbankzugriffe erfolgen und in welchem Zyklus das Dashboard ausgeführt werden soll. Die benutzten Plug-Ins müssen manuell in das POM eingefügt werden. Es fehlt eine zentrale Konfigurationsdatei für die Regeln und Einstellungen der einzelnen genutzten Plug-Ins des Dashboard. Diese Einstellungen müssen ebenfalls manuell getätigt werden. Das hat zur Folge, dass die anzuwendenden Regeln einzeln in jeweiligen XML Dateien einzustellen sind.

Für die Darstellung der Verlaufsfunktion kann die Anzahl von Diagrammen und die Spanne der Zeitachsen sehr flexibel gewählt werden. In diesem Test wurden pro Plug-In drei Darstellungsformen gewählt. Diese sind der Verlauf für die jeweilige Woche (Zeiteinheit: Tag), der jeweilige Monat (Zeiteinheit: Woche) und das aktuelle Jahr (Zeiteinheit: Monat). Die Daten der Datenbank für die jeweilige Einheit werden gemittelt und dann in die Diagramme eingetragen.

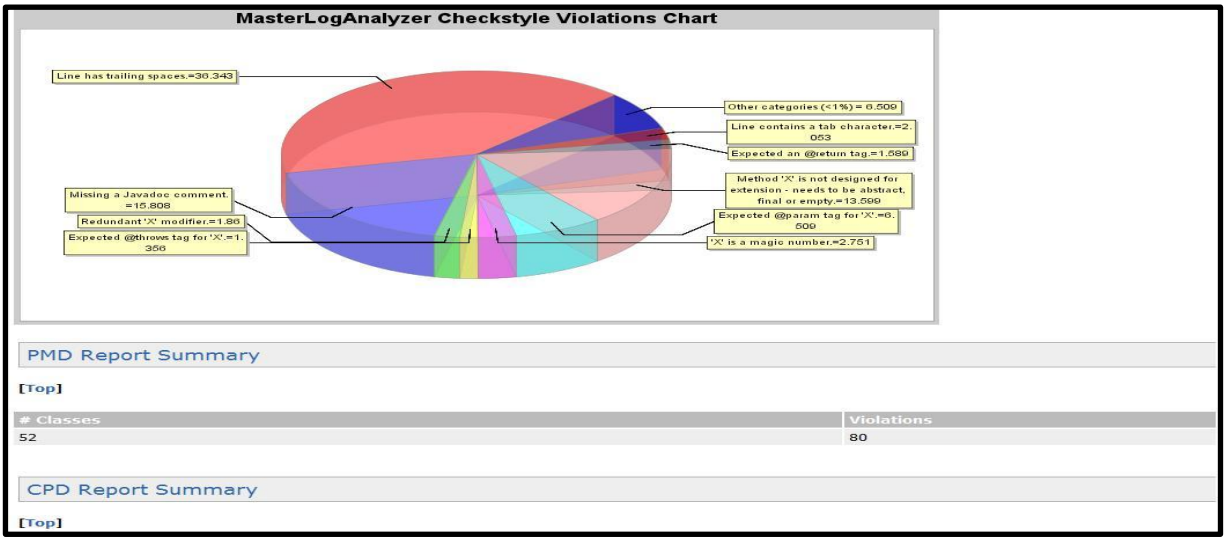



**Abbildung 14: Dashboard Auszug**

Abbildung 14 zeigt schemenhaft, wie solch eine Verteilungsdiagramm optisch aufgebaut ist.

### 5.2.4 Bewertung der Aggregation Plug-Ins

Die drei zur Verfügung stehenden Aggregationswerkzeuge wurden mit den Standard-einstellungen getestet. Um eine Auswahlentscheidung zu treffen, wurde erneut eine ABC-Analyse durchgeführt. Die Ergebnisse der Analyse sind in der folgenden Tabelle dargestellt:

| Anforderung | Dashboard | Sonar | QAlab |
|---|---|---|---|
| Fehler | ++ | ++ | ++ |
| Testabdeckung | 0 | 0 | 0 |
| Kopplung | + | + | + |
| Non Commenting Source Statements | 0 | ++ | 0 |
| Lines of Code | 0 | 0 | 0 |
| Number of Classes | + | 0 | 0 |
| Abstraktheit | ++ | 0 | 0 |
| Instabilität | ++ | 0 | 0 |
| Distanz | ++ | 0 | 0 |
| Change Rate | -- | ++ | -- |
| Eigener Stil | ++ | ++ | ++ |
| Stilmittel | ++ | ++ | ++ |
| Repräsentation | ++ | + | - |
| Konfigurierbarkeit | + | ++ | -- |
| Last | 0 | -- | 0 |
| Dauer | 1h | 0.75h | 0.35h |
| Integration | ++ | - | + |
| Verlauf | ++ | + | - |

**Legende:**

0: im ersten Test nicht gefunden

++ : sehr gut

+: gut

-: ungenügend

--: gar nicht enthalten

**Tabelle 5: Aggregation ABC Analyse**

Entsprechend der Tabelle fällt die Wahl nicht auf QAlab, da hier nicht die Designmetriken berechnet und in die Projektwebseite eingebaut werden können. Sonar bietet zum Recherchestand durch den Einsatz von JavaNCSS verschiedene Basismetriken wie NCSS oder NC an, jedoch ist die aggregierte Darstellung der Projektwebseite hauptsächlich in Listenform vorhanden. Dies ist unübersichtlich und schwer auszuwerten, da hier übersichtliche Grafiken oder verdichtete Informationen fehlen. Des Weiteren nutzt Sonar einen eigenen Serverdienst, welcher die Datenbank und die Funktionalität von Sonar zur Verfügung stellt. Dieser muss zur Ausführungszeit gestartet sein und ist somit nicht vollständig in die Maven2 Lebenszyklen eingebunden. Vor allem jedoch fehlen Design-metriken wie Abstraktheit, Distanz und Instabilität (vgl. Abschnitt 4.2.3, Seite 23), die vom

Plug-In JDepend verfügbar gemacht werden. Dieses Plug-In wird nicht von Sonar unterstützt. Da JavaNCSS eine eigene Aggregationsmöglichkeit anbietet und JDepend interessante Designmetriken anbietet, aber ansonsten die Schnittmenge zwischen Sonar und Dashboard recht gleich sind, fiel die Wahl des einzusetzenden Plug-Ins in diesem Fall auf Dashboard.

## 5.3 Zusammenfassung der Marktanalyse

Nachdem nun die verschiedenen Werkzeuge für die Erstellung einer Qualitätsanalyse einmal getestet und betrachtet wurden, wurde mit dem Kunden vereinbart, den Ansatz des Dashboard Plug-Ins mit den dazugehörigen Werkzeugen zu verfolgen. Dieses stellt zum derzeitigen Stand die größte Abdeckung des geforderten Qualitätsmodells und die höchste Integration in Maven2 dar. Die nun folgenden Absätze stellen die Feineinstellung und Herangehensweise an das Projekt dar, welches einer kontinuierlichen Qualitätssicherung mittels Maven2 unter Einsatz des Dashboard Plug-Ins unterzogen wird.

## 5.4 Vorgehen

Das zu untersuchende Projekt hat, wie bereits erwähnt, ein beachtliches Volumen. Um die Möglichkeiten des Dashboards bestmöglich zu nutzen und das geforderte Qualitätsmodell zu realisieren, wurde entschieden, dass in mehreren Schritten eine Annäherung an das Real-projekt von statten gehen soll. Dazu soll in fünf Iterationen am Ende ein Stand erreicht werden, in dem das gesamte Softwareprojekt des Kunden permanent und automatisch der in dieser Arbeit entstandenen Qualitätssicherung unterliegt. Die ersten vier Iterationen dienen als Annäherung an diesen Stand. Dies umfasst die Konfigurationsarbeit, sodass nach und nach die Regeln der Plug-Ins verfeinert und projektbezogener eingestellt werden. Folgende Ziele sind für die einzelnen Iterationen gesetzt:

1 Erste Schritte (Iteration 1):
   Hier soll ein selbsterstelltes Projekt bestehend aus einem Maven2 Projekt mit selbsterstellten Quelltexten untersucht werden. Anhand der selbsterstellten Projekte kann dem Kunden gezeigt werden, wie das Dashboard arbeitet. Weiterhin wird erarbeitet, wie die Einstellungen der Plug-Ins durchzuführen sind.

2 Multi-POM-Projekt (Iteration 2):
   In dieser Iteration wird ein eigenes Maven2 Projekt mit eigenen Quelltexten erstellt, welches aber aus mehreren geschachtelten Maven2 Projekten besteht. Hier sollen

die Einstellungen getestet und bearbeitet werden, die diese Maven2 Struktur unterstützen.

3 Einzelnes Realsoftwareprojekt (Iteration 3):
Hier wird ein einzelnes Projekt aus der Struktur des Kunden herausgelöst und einzeln via Dashboard untersucht. Dadurch soll ein erster Überblick über die Ergebnisse der Metriken geliefert werden. Aus Basis dieser Ergebnisse werden weitere Feineinstellungen folgen.

4 Erster Test auf das Gesamtprojekt (Iteration 4):
Es soll das gesamte Kundenprojekt auf einem lokalen Testsystem untersucht werden. Entsprechend werden die Ergebnisse ausgewertet und weitere Feineinstellungen folgen.

5 Integration (Iteration 5):
Nun wird die Qualitätssicherung in die IT-Infrastruktur integriert und so konfiguriert, dass sie automatisiert abläuft.

6 Evaluation & Differenzbegleichung (Iteration 6):
Sowohl Erweiterungen noch nicht realisierter Features sollen hier durchgeführt werden, als auch die Möglichkeit des selbsterstellten Regeleinsatzes für verschiedene Plug-Ins soll hier evaluiert werden. Diese Iteration umfasst auch den endgültigen Abgleich des SOLL-Stands mit dem nun erreichten IST-Stand und den daraus folgenden Überlegungen, wie die Differenz minimiert werden kann. Diese Iteration ist in Abschnitt 5.6 und Kapitel 6 dargestellt.

## 5.5 konkrete Umsetzung

Der Abschnitt „konkrete Umsetzung“ stellt die Dokumentation der einzelnen Iterationen wie unter 5.4 beschrieben dar. Hierbei werden nun für jede Iteration zunächst das untersuchte Projekt mit dem jeweiligen Ziel und dann die Konfigurationsschritte beschrieben. Anschließend folgt eine Auswertung, wobei die unter Anforderungen gestellten Kriterien mit dem erreichten Stand verglichen werden. Vorausnehmend ist zu erwähnen, dass wie in jedem Projekt, von Zeit zu Zeit einige Probleme gefunden wurden. Dabei handelt es sich um Probleme oder Fehler, welche bei der Umsetzung auftraten. Lösungswege und Erklärungen werden ausführlich zum jeweiligen Zeitpunkt gegeben, soweit vorhanden.

### 5.5.1 Erste Schritte ( Iteration 1)

***Ziel***

In der ersten Iteration soll ein selbsterstelltes Softwareprojekt einer Qualitätssicherung unterzogen werden. Dafür sind die benötigten Plug-Ins namens Dashboard, CheckStyle, PMD, Cobertura, FindBugs, JDepend und SureFire so in dem POM des Softwareprojektes zu konfigurieren, dass das Qualitätsmodell wie in Abschnitt 4.2 umgesetzt wird.

***Softwareprojekt***

Zunächst wurde ein einfaches Maven2 Projekt erstellt, wobei Maven2 für die Erstellung des Softwareprojekts genutzt wurde. Standardmäßig gibt es hierfür Befehle im Maven2 Kontext, jedoch muss auf der Konsole ein langer Befehl eingegeben werden. Anschließend muss das Softwareprojekt für die Entwicklungsumgebung angepasst werden. Da davon auszugehen ist, dass noch weitere Projekte erstellt werden, wird die Arbeit durch den Einsatz einer Entwicklungsumgebungserweiterung ungemein vereinfacht. Solche eine Erweiterung wird auch bereits vom Projektteam genutzt und heißt „M2“. Dieses Plug-In bietet die Möglichkeit, direkt aus der Entwicklungsumgebung ein Maven2 kompatibles Projekt zu erstellen.

In dieser Iteration wird ein einfaches Maven2 Projekt ohne Vererbungshierarchien benötigt und namens TEST1 erstellt. Die Quelltexte finden sich im Anhang. In TEST1 wurde zusätzlich eine Klasse namens CLASS1 und CLASS2 erstellt, welche lediglich einfache Integer- und Doubleoperationen über Methoden bereitstellt. Dabei wurden auch *switch* Anweisungen genutzt, um die zyklomatische Komplexität zu erhöhen. Um auch die Möglichkeiten von JDepend zu testen, mit besonderem Augenmerk auf die Erkennung von Zyklen, wurde ein

Zyklus zwischen CLASS1 und CLASS2 realisiert. Dashboard und die dazugehörigen Plug-Ins unterstützen die Programmierstilkontrolle. Entsprechend wurden auch diese getestet, indem bewusst einige Fehler eingebaut wurden, die CheckStyle und PMD finden sollten. Dazu gehört die Instanziierung von boolschen Werten. Diese sollten laut Angaben von PMD wie folgt geschehen:

```
Boolean b1 = Boolean.TRUE;
```

und nicht wie folgt über den Stringkonstruktor:

```
Boolean b1 = new Boolean(„true“);
```

Die Booleaninstanziierung ist zwar kein Fehler, der Auswirkungen auf den Erfolg eines Projekts hat, aber als Indikator für den Programmierstil und das Auslösen einer Regel ist dieses einfache Beispiel ausreichend.

### *Konfiguration*

In dem POM des Softwareprojektes TEST1 müssen nun die einzelnen Plug-Ins eingetragen werden. Alle Plug-Ins werden in dem Abschnitt <reporting> des POM, der den Seitengenerierungszyklus beschreibt, nach folgendem Muster eingetragen:

```
<project>
      …
      <reporting>
      <plugin>
            <groupId> Herstellergruppen ID </groupId>
            <artifactId> Name des Plug-In </artifactId>
            <version> Versionsnummer </version>
      <plugin>
      </reporting>
      …
</project>
```

Das Dashboard, SureFire und Cobertura Plug-In müssen zusätzlich noch im Erstellungslebenszyklus in der POM unter <build> eingetragen werden. Konfigurationsanleitungen der einzelnen Plug-Ins finden sich auf der jeweiligen Herstellerseite.

Zusätzliche Einstellungen sind an dem Dashboard Plug-In vorzunehmen. Damit ein Verlauf der Metriken und anderen Ergebnisse erzeugt werden kann, benötigt Dashboard eine Datenbankanbindung. Dazu müssen zunächst ein Datenbankschema für diesen Zweck und ein Benutzeraccount für das Schema in einer Datenbank angelegt werden. Auf jedem Arbeitsplatz und Server in der IT-Infrastruktur des Kunden ist eine MySQL Datenbank vorhanden, die mit dem Dashboard kompatibel ist. Das erzeugte Datenbankschema und der Zugriffsaccount werden im Abschnitt <configuration> der Dashboarddefinition in dem POM eingetragen.

### *Ausführung*

Da die Konfiguration wie auf den Herstellerseiten beschrieben durchgeführt wurde, wird nun die Qualitätssicherung auf das Softwareprojekt TEST1 angewandt. Die folgenden Befehle werden für das starten genutzt:

```
1. mvn site
2. mvn org.codehaus.mojo:dashboard-maven-plugin:1.0-SNAPSHOT:
   persist
3. mvn org.codehaus.mojo:dashboard-maven-plugin:1.0-
   SNAPSHOT:dashboard
```

Der Befehl *mvn site* führt die Seitengenerierung durch, dabei werden automatisch die eingestellten Plug-Ins aktiviert. Der Befehl *mvn org.codehaus.mojo:dashboard-maven-plugin:1.0-SNAPSHOT: persist* speichert die ermittelten Daten in der Datenbank und *mvn org.codehaus.mojo:dashboard-maven-plugin:1.0-SNAPSHOT:dashboard* aktualisieren die Daten auf der Reportseite des Dashboard. Anschließend kann das Ergebnis in der Ordnerstruktur des Projekts TEST1 im Ordner *site* mit einem HTML-Darstellungsprogramm angesehen werden.

### *Auswertung und Probleme*

Das Resultat stellt sich erwartungskonform dar, die „eingebauten“ Fehler wurden gefunden, die Zyklen dargestellt und die Designmetriken, wie gefordert, erstellt.Leider fällt auf, dass JDepend auf der Startseite des Dashboard nur die ausgehende Kopplung und die Anzahl der Pakete anzeigt.

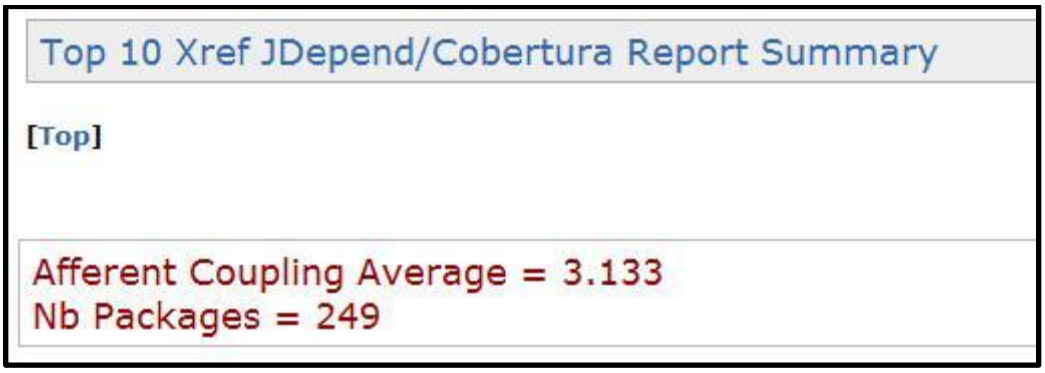



**Abbildung 15: JDepend Auszug Übersichtsseite Dashboard**

Die anderen errechneten Metriken sind nur über den direkten Link von der JDepend Reportseite erkennbar. Die Suche nach einer Lösung hat keine Möglichkeit gefunden, dieses Verhalten unkompliziert zu ändern. Weder das Dashboard noch JDepend bieten über die Konfigurationsschnittstelle Möglichkeiten an, die Informationen gezielt und verdichtet auf oberster Ebene darzustellen. Auch Maven2 bietet auf dem derzeitigen Versionsstand keine Möglichkeit an, die Ergebnisse von JDepend auf einer höheren Ebene, als der des jeweiligen Projektes anzuzeigen. Dieses Problem wird sich auch durch die weiteren Iterationen ziehen und nur über massive Veränderungen des Dashboard Plug-Ins lösbar sein. Dazu mehr im Abschnitt Evaluation und Delta Bereinigung.

***Zusammenfassung***

Das Ziel dieser Iteration ist erreicht, indem die Plug-Ins nun auf ein eigenes erstelltes Projekt angewandt wurden und die erwarteten Ergebnisse in Form einer Projektwebseite lieferten. Es wurden Mängel aufgedeckt, die die Repräsentation der ermittelten Metriken betrifft. Die Behebung dieser Mängel wird im Kapitel 6 Delta Bereinigung diskutiert.

### 5.5.2 Multi-POM-Projekt (Iteration 2)

***Ziel***

Es soll ein aus wenigen Klassen bestehendes Maven2 Projekt erstellt werden, das Vererbungshierarchien beinhaltet. Zum besseren Verständnis solch eines Aufbaus folgt ein Beispiel aus der Dashboarddokumentation (Dashboard, 2007):

```
whizbang
|-- POM.xml
|-- core
|   `-- POM.xml
|-- gui
|   `-- POM.xml
|-- jmx
|   `-- POM.xml
`-- src
```

Solch eine Projektstruktur enthält ein Projekt, welches nur aus einer POM besteht, in dem eine Konfiguration für das gesamte Softwareprojekt definiert wird. Weitere Softwareprojekte (Module) erben von diesem POM und übernehmen somit die Einstellungen, sofern die Einstellungen nicht direkt überschrieben werden. Diese multimodulare Struktur ist in dem Softwareprojekt des Kunden vorhanden. Es soll in dieser Iteration mit Hilfe eines kleinen Softwareprojektes die nötige Konfigurationsarbeit des Dashboard für diese Struktur erarbeitet werden.

### *Softwareprojekt*

Zunächst wurde aus der Entwicklungsumgebung heraus ein Maven2 Projekt namens Multi1 erstellt. Dann wurden zwei Unterordner namens First und Second innerhalb von Multi1 erstellt, in denen wiederum Maven2 Projekte enthalten sind. In First und Second wurde jeweils eine Klasse erstellt, die mehrere einfache Integeroperationen durchführt.

Klassen aus verschiedenen Projekten greifen dabei auf Klassen anderer Projekte zu. Es sind zyklische Abhängigkeiten enthalten. Einzelne Klassen haben Testfälle, die einzelne Zeilen und Verzweigungen prüfen.

### *Konfigurationsarbeit*

Nachdem ein Maven2 Softwareprojekt mit einzelnen Klassen und Testfällen erstellt wurde, beginnt an dieser Stelle die Konfigurationsarbeit. Damit die Vererbungshierarchie dargestellt werden kann, muss in dem POM von First und Second ein <parent>-Tag gesetzt sein, indem der Maven2 Name von Multi1 eingetragen wird. Nun sollten die beiden Projekte die Einstellungen des Multi1 POM übernehmen.

Nun muss das Dashboard und die dazugehörigen Plug-Ins in dem POM von Multi1 eingetragen werden. Die Einstellungen sind äquivalent zu denen aus Iteration1.

### *Ausführung*

Laut Herstellerseite des Dashboard Plug-Ins müssen nun für das Multi1 Projekt die Maven2 Befehle

```
1. mvn site
2. mvn org.codehaus.mojo:dashboard-maven-plugin:1.0-SNAPSHOT: persist
3. mvn org.codehaus.mojo:dashboard-maven-plugin:1.0-SNAPSHOT:dashboard
```

ausgeführt werden, und die Informationen aller Untermodule, also First und Second, auf der Projektwebseite in einem Report dargestellt werden. Die Darstellung der Übersichtsseite, und somit die Ausführung der Plug-Ins auf alle Module des Softwareprojektes, funktionierte, die Hyperlinks zu den detailierten Informationen der Module funktionierten jedoch nicht.

### *Probleme*

Die Darstellung der Übersichtsseite wurde mit den gewählten Einstellungen erreicht, die Hyperlinks zu den Detailinformationen der Module sind allerdings fehlerhaft. Um dieses Verhalten zu ändern, ist der Seitengenerierungslebenszyklus noch einmal genauer zu betrachten. Zu diesem Zeitpunkt wird durch den Befehl `mvn site` von jedem Modul eines Softwareprojektes ein Report erstellt. Jedoch behandelt `mvn site` jedes Modul als eigenständiges Projekt und nutzt als Hyperlinks die Ordnernamen, in denen die Module auf dem Speichermedium gesichert sind. Das Dashboard Plug-In aggregiert die Informationen aller Submodule, übernimmt aber die Hyperlinks nicht direkt, sondern setzt den Ordnernamen des Mutli1 Projektes als Basislink voraus.

Um dieses Verhalten zu umgehen, ist der zweite Befehl des Seitengenerierungslebenszyklus `mvn site-deploy` hilfreich. Dieser Befehl verschiebt die Reports der Plug-Ins an einen in dem POM definierten Ort auf einem Speichermedium oder auf einen Webserver. Hierfür kann im Abschnitt <`distributionManagement`> in dem POM des Multi1 Projektes ein beliebiger Ordner gewählt werden. `Mvn site-deploy` passt die Hyperlinks so an, dass die Struktur wieder eingehalten wird. Somit ergibt sich, dass nun nach dem Ausführen der bisherigen drei Befehle noch `mvn site-deploy` ausgeführt wird.

### *Zusammenfassung*

Die Konfiguration eines selbsterstellten, kleinen Softwareprojektes, das eine Vererbungshierarchie beinhaltet, wurde umgesetzt. Es gab ein Problem mit den Hyperlinks der

einzelnen Detailstufen des Reports von Dashboard, welches durch die Einführung eines weiteren Maven2 Befehls behoben werden konnte. Anschließend wurden die Informationen wie erwartet dargestellt. Es ist mittels dieser Konfiguration möglich, Konfigurationen und Maven2 Befehle nur innerhalb des Projektes auszuführen, welche in der Vererbungshierarchie ganz oben stehen. Die Konfigurationen und Befehle werden dann automatisch von Maven2 an die Untermodule weitergereicht. Somit kann durch Befehle an einer Stelle die komplette Qualitätssicherung auf ein wie hier aufgebautes Projekt durchgeführt werden.

### 5.5.3 Einzelnes Realsoftwareprojekt (Iteration 3)

#### *Ziel*

Ein einzelnes Teilprojekt des Softwareprojektes soll mittels des Dashboard Plug-Ins untersucht werden. An diesem Teilprojekt sollen die konfigurierbaren Regeln so eingestellt werden, dass der Kunde mit den Ergebnissen arbeiten kann.

#### *Softwareprojekt*

Das Teilprojekt des gesamten Softwareprojektes des Kunden, das in dieser Iteration betrachtet wird, beinhaltet die Fachwerte des Projektes. Die Fachwerte sind gut für diese Iteration geeignet, da nur auf diese zugegriffen wird und keine Abhängigkeiten zu anderen Teilsystemen bestehen. Bei dem Vorhandensein von ausgehenden Abhängigkeiten würde Maven2 alle abhängigen Teilprojekte ebenfalls untersuchen. Das Fachwerteprojekt hat ein Volumen von 11707 NCSS (Statements und Blöcken), ist in 11 Pakete unterteilt und umfasst 246 Dateien. Es handelt sich um das bisher voluminöseste Softwareprojekt, das im Rahmen dieser Arbeit einer Qualitätssicherung unterliegt.

#### *Programmierstil definieren*

PMD und CheckStyle werden in dieser Iteration nicht mehr im Standardverhalten durchgeführt. Stattdessen sollen die einzusetzenden Regeln von PMD und CheckStyle zusammen mit dem Entwicklerteam gewählt werden. Dazu sind die verfügbaren Regeln mit einer kurzen Beschreibung in einem Dokument zusammengefasst und anschließend mit zwei Vertretern der Entwickler ausgewählt worden.

### *Konfigurationsarbeit*

Zunächst wurde das POM der Fachwerte um die Grundeinstellungen des Dashboard- und aller dazugehörigen Plug-Ins analog zu Iteration1 und Iteration2 erweitert. Um die gewählten Regeln für CheckStyle und PMD zu konfigurieren, muss jeweils eine zentrale XML-Datei erstellt werden. Die CheckStylekonfiguration erfolgt, indem die Regeln in einem Tag eingebettet werden. Der Tag ist wie folgt aufgebaut:

```
<module name =“Name der Regel“>
      <property name=“severity“ value=“warning oder error oder info“>
      <property name=“spezifische Einstellung für eine Regel“/>
</module>
```

Einige Regeln, wie beispielsweise die maximale Zeilenlänge, können um einen Schwellenwert erweitert werden. Solche Einstellungen werden ebenfalls in einen *property* Tag eingetragen.

PMD benutzt auch eine zentrale Konfigurationsdatei. Dabei ist der Aufbau aber ein anderer als bei CheckStyle. PMD nutzt die Kategorien der einzelnen Regeln als Ausgangspunkt für die Einstellungen. Wenn eine Kategorie in der XML-Datei nach folgender Schablone

```
<rule ref=“rulesets/Name_der_Kategorie.xml“/>
```

eingetragen wurde, werden automatisch alle enthaltenen Regeln angewandt. Um nun die mit dem Kunden zusammen erarbeitete Einstellung möglich zu machen, wurden nicht angewandte Regeln exkludiert.

Dies erreicht man über den Zusatz des folgenden Tags:

```
<exclude name=“Regelname“/>
```

### *Ausführung*

Für die Ausführung wurden erneut die vier Maven2befehle analog zu Iteration2 ausgeführt. Maven2 konnte das Ausführen aber nicht beenden, da einige Tests fehlschlugen. Des Weiteren war die HEAP Größe für die Ausführung der Maven2befehle zu gering. Jedes Programm hat zu Ausführungszeit einen Teil des HEAP im Hauptspeicher des Betriebs-systems, in dem Daten der Laufzeit gespeichert werden. Wie dieses Problem gelöst wurde,

findet sich im folgenden Abschnitt. Die konfigurierten Regeln wurden erwartungskonform bearbeitet und die Metriken aller Plug-Ins vom Dashboard Plug-In dargestellt.

### *Probleme*

In dieser Iteration sind zwei Probleme aufgetreten. Zum einen der Abbruch der Qualitätssicherung bei einem fehlgeschlagenen Test und zum anderen der Abbruch der Qualitätssicherung aufgrund zu geringer HEAP Größe für die Ausführung.

Die HEAP Größe, welche Maven2 zur Verfügung steht, lässt sich über eine Systemvariable definieren. Unter einem Windows Betriebssystem kann dazu folgender Konsolenbefehl ausgeführt werden:

```
set MAVEN_OPTS="-XmxTTTm"
```

TTT stellt bei der oberen Darstellung einen Platzhalter für eine Megabytezahl dar. In dieser Arbeit hat sich eine HEAP Größe von 768 MB (`set Maven_OPTS="-Xmx768m"`) als funktionierend erwiesen.

Das SureFire Plug-In führt die JUnit-Tests eines Maven2 Projektes durch. Dabei wird der Erstellungsprozess abgebrochen, sobald ein Test fehlschlägt. Dieses Verhalten ist im Erstellungsprozess auch sinnvoll, da ein fehlgeschlagener Test ein Indikator für einen noch vorhandenen Fehler im System ist. Bei den qualitätssichernden Maßnahmen dieser Arbeit soll aber ein fehlgeschlagener Test nicht die Durchführung stoppen, sondern als Ergebnis in dem Report dargestellt werden. Das SureFire Plug-In bietet keine direkte Konfigurationsmöglichkeit, um den Abbruch bei einem Fehler zu unterdrücken. Es kann nur im POM definiert werden, ob alle Tests ausgeführt werden sollen oder nicht. Eine Recherche auf der Herstellerseite von Maven2 ergab, dass der Zusatz eines Parameters beim Aufruf der Befehle, dass in diesem Fall gewünschte Verhalten ermöglicht. Dazu müssen die Befehle wie folgt um den Parameter „`-Dmaven.test.failure.ignore=true`“ erweitert werden:

```
mvn <Befehlsname> -Dmaven.test.failure.ignore=true
```

### *Zusammenfassung*

In dieser Iteration wurde das erste Mal ein Teil des Softwareprojektes des Kunden mit qualitätssichernden Erweiterungen für Maven2 untersucht. Zusammen mit dem Entwicklerteam sind die Regel der Programmierstil Plug-Ins gezielt konfiguriert worden. Problemlösungen für die zu geringe HEAP Größe und das Verhalten bei fehlgeschlagenen

Tests wurden gefunden. Die Ergebnisse wurden erwartungskonform auf der Projektwebseite dargestellt

### 5.5.4 Erster Test auf das Gesamtprojekt (Iteration 4)

***Ziel***

In der Iteration 4 wird der bisherige Stand der Arbeit erstmals auf das gesamte Softwareprojekt des Kunden angewandt. Die Ergebnisse werden Aufschluss darüber liefern, ob die Einstellungen der Regeln und Metriken noch weiter verbessert werden können. Es soll die Dauer gemessen werden, welche die Ausführung der Qualitätssicherung benötigt. Es sollen Möglichkeiten gefunden werden, die Dauer der Ausführung zu optimieren.

***Softwareprojekt***

Das Softwareprojekt des Kunden umfasst zum Stand im November 2007: 279 Pakete, 2471 Dateien und eine Anzahl von 169478 ausführbaren Statements. Dieses Softwareprojekt ist multimodular aufgebaut und beinhaltet entsprechend eine Vererbungshierarchie. Das an der Spitze der Vererbungshierarchie stehende Projekt wird im Folgenden als KR benannt. Die Quelltexte des Softwareprojektes sind über das vom Kunden eingesetzte SCM-Werkzeug abrufbar und werden auf dem für diese Arbeit eingerichteten Arbeitsplatz bearbeitet.

***Konfigurationsarbeit***

In dem POM des KR Projektes (an der Spitze der Vererbungshierarchie stehendes Softwareprojekt) ist die Dashboardkonfiguration und aller zugehörigen Plug-Ins einzutragen. Die Konfigurationsdateien für Dashboard, CheckStyle und PMD wurden in einen Ordner namens dash_conf verlagert und müssen entsprechend an den Konfigurationsstellen im POM eingetragen werden. Die Verlagerung ist darin begründet, dass die einzelnen Konfigurationsdateien für die Qualitätssicherung an einem Ort im Projekt gebündelt sein sollten. Dies erhöht die Übersicht über die Konfiguration des Systems.

### *Ausführung*

Aus dem KR Projekt (an der Spitze der Vererbungshierarchie stehendes Softwareprojekt) sind folgende Befehle auszuführen:

```
1. mvn site -Dmaven.test.failure.ignore=true
2. mvn org.codehaus.mojo:dashboard-maven-plugin:1.0-SNAPSHOT: persist
3. mvn org.codehaus.mojo:dashboard-maven-plugin:1.0-SNAPSHOT:dashboard
4. mvn site-deploy -Dmaven.test.failure.ignore=true
```

Der erste Durchlauf dauerte ca. 2 h. Diese Zeitspanne erscheint sehr lang und ist darin begründet, dass beim ersten Durchlauf zunächst die Quellen der einzelnen Plug-Ins aus den jeweiligen Repositories heruntergeladen wurden. Ein erneuter Durchlauf wurde nach 59 min erfolgreich beendet. Bisher ist der Standpunkt in dieser Arbeit vertreten, dass Entwickler nach Veränderungen an Quelltexten selbstständig und automatisiert eine Qualitätssicherung ausführen. Dabei ist eine Dauer von 59 min zu lang, da hier fast eine Stunde Arbeitszeit verloren ginge. Damit dem Entwickler dennoch die Möglichkeit der Qualitätssicherung zur Verfügung gestellt wird, werden im Abschnitt Probleme nach Optimierungsansätzen für die Performance gesucht.

Die Ergebnisse der Qualitätssicherung wurden an dem in dem POM definierten Ort als Webseite von Maven2 verfügbar gemacht. Nach wie vor sind die von JDepend ermittelten Metriken nicht auf der Übersichtsseite verfügbar. Hier nun einige Auszüge aus dem Report:

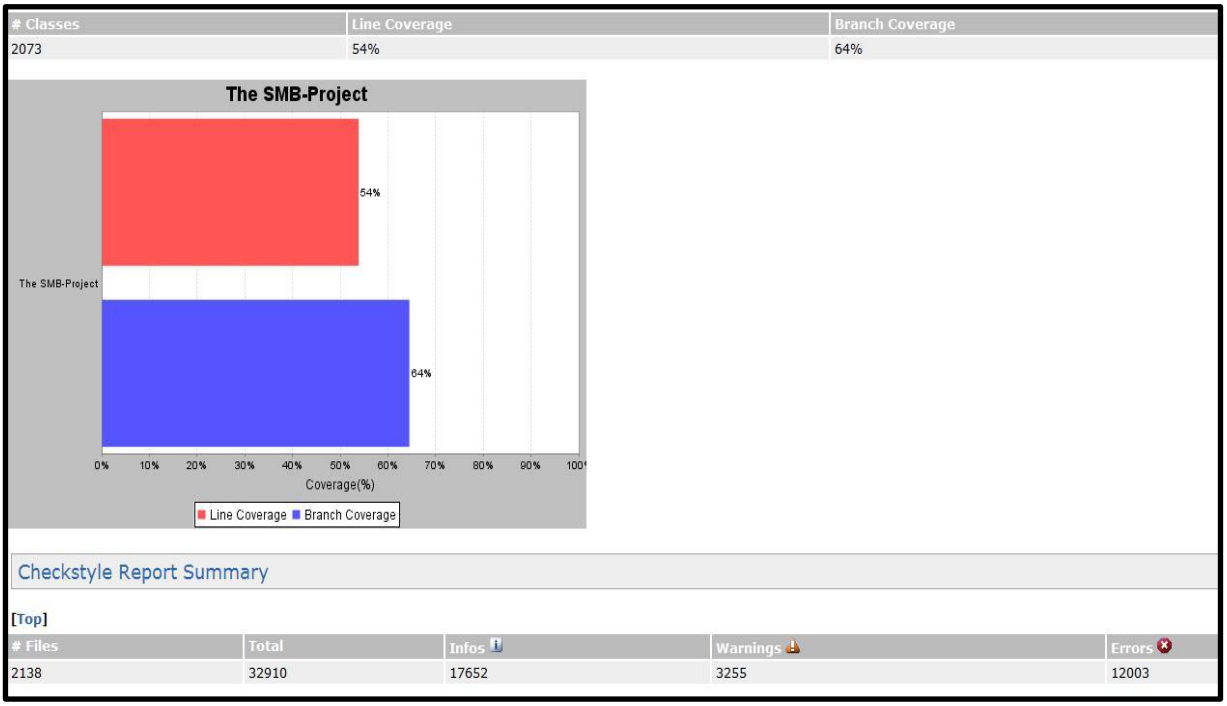



**Abbildung 16: Auszug Dashboard Report 1**

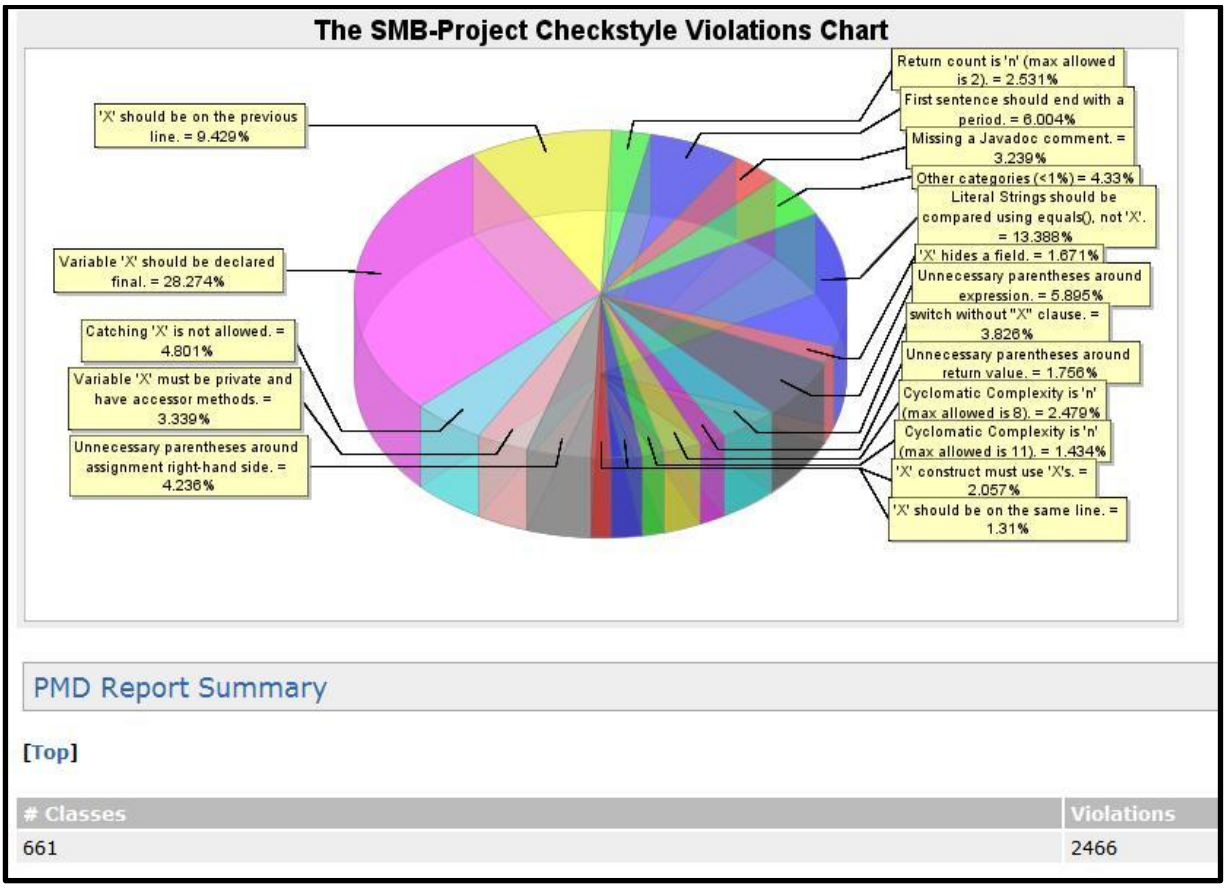



**Abbildung 17: Auszug Dashboard Report 2**

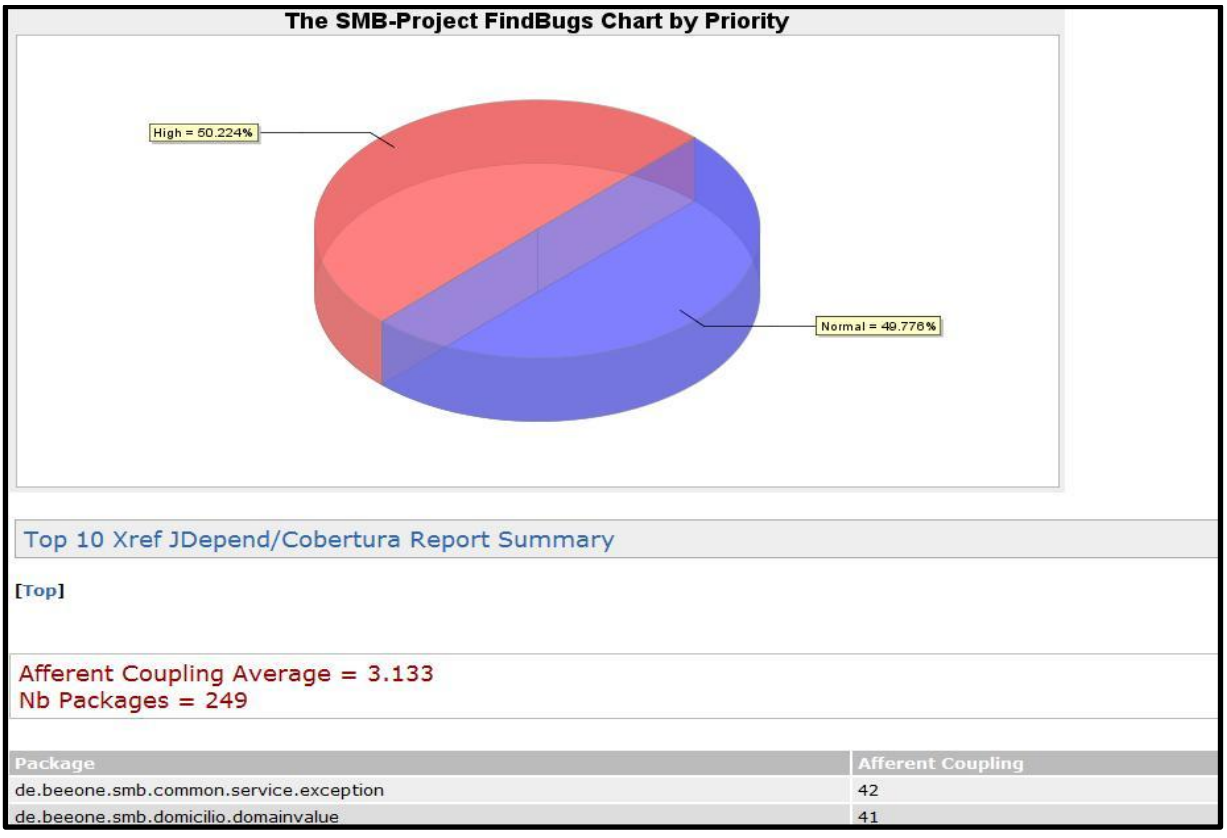



**Abbildung 18: Auszug Dashboard Report 3**

Die für CheckStyle und PMD definierten Regeln wurden korrekt auf die Quelltexte der Einheiten angewandt. In den Diagrammen auf der Übersichtsseite des Dashboard wird die prozentuale Verteilung der Fehler angezeigt. Dabei fällt auf, das einzelne Regeln, wie die maximale Zeilenlänge, weit über 75% der Fehlerverteilung einnehmen. Dies verdeckt jedoch weitaus interessantere Fakten wie leere Blöcke oder leere catch-Klauseln.

Bei der Auswertung der Ergebnisse fällt auf, dass Cobertura immer 100% Zeilenabdeckung und 0% Verzweigungsabdeckung anzeigt. Die Quelltexte haben nach manueller Betrachtung eine andere Verteilung der Testabdeckung. Nach einer Lösung für dieses Verhalten muss gesucht werden.

### *Probleme*

Die Laufzeit der Qualitätssicherung umfasst ca. eine Stunde. Damit diese Zeit minimiert werden kann, wurde zusammen mit dem Entwicklerteam nach Einheiten im Softwareprojekt gesucht, die im Rahmen der Qualitätssicherung nicht betrachtet werden müssen. Solche Einheiten sind Quelltexte, die für Webservice automatisch generiert wurden. Hierbei spricht man von Stubs und Skeletons, die die Schnittstellen von Webservices beschreiben. Des Weiteren gibt es im Softwareprojekt Quelltexte, die nicht vom Entwicklerteam gefertigt, sondern von Fremdanbietern bearbeitet wurden. Diese beiden Kategorien von Quelltexten sollen aus dem Fokus der im Rahmen dieser Arbeit durchgeführten Qualitätssicherung herausgenommen werden und eventuell gesondert betrachtet werden. Das Ziel ist es, nur selbsterstellte und bearbeitete Quellen zu untersuchen.

Um einzelne Quelltexte aus der Qualitätssicherung herauszunehmen, können in den Plug-Ins CheckStyle, PMD, Cobertura und SureFire sogenannte „excludes“ definiert werden. Dabei ist im <configurations> Abschnitt der Plug-Ins im POM des KR Projektes folgendes einzutragen:

```
<exlucde> **/*_Skel.class </exclude>
<exlucde> **/*_Stub.class </exclude>
<exlucde> **/de/Anbieter1/*.class </exclude>
<exlucde> **/de/Anbieter2/*.class </exclude>
```

Die ersten beiden <exclude> Tags betreffen alle Skeletons und Stubs im System. Die Namenskonvention, dass alle Skeletonklassen mit _Skel und alle Stubklasse mit _Stub enden,

ermöglicht, dass durch die <exclude> Tags diese Klassen nun nicht mehr von den Plug-Ins untersucht werden.

Der dritte und vierte <exclude> Tag bezeichnet eine Ordnerstruktur. Im Softwareprojekt sind Quelltexte von Fremdanbietern immer in eigenen Paketen abgelegt. Pakete namens Anbieter1 und Anbieter2 (Platzhalter) werden entsprechend nicht mehr von den Plug-Ins untersucht.

Durch diese Maßnahmen wurde ein Performancegewinn von 29% erreicht. Nun benötigt ein Durchlauf der Qualitätssicherung statt einer Stunde nur noch 42 min. Dennoch sind 42 min für einen Durchlauf zu lang, als dass ein Entwickler nach jeder Änderung an Quelltexten dieses an seinem Computerarbeitsplatz ausführen könnte. Mögliche Lösungsansätze für dieses Problem werden im Kapitel 6 Delta Bereinigung diskutiert.

Ein zweites Problem in dieser Iteration ist, dass einige Programmierstilregeln sehr oft auftreten und damit in der Übersicht weitere Fehler verdecken. Dabei handelt es sich um Designprobleme, wie beispielsweise die maximale zugelassene Zeilenlänge. Fehler wie fehlende JavaDoc Kommentare oder die maximale Endpunktanzahl (d.h. Return Anweisungen) einer Klasse werden dabei verdeckt, indem sie nicht mehr in den Diagrammen auftreten. An dieser Stelle gilt es, mit dem Entwicklerteam abzuwägen, wie wichtig einzelne Regeln für das Softwareprojekt sind. So ist man übereingekommen, dass die maximale Zeilenlänge als Regel herausgenommen wurde, damit für das Team wichtigere Fehler sichtbar werden. Zu einem späteren Zeitpunkt können dann die ausgesetzten Regeln wieder aktiviert und auftretende Verstöße behoben werden.

Als letztes Problem dieser Iteration ist aufgefallen, das Cobertura, welches die Test-abdeckung einer Einheit misst, immer 100% Zeilenabdeckung und 0% Verzweigungs-abdeckung anzeigt. Dieser Fehler ist in dem Fehlerverfolgungssystem von Cobertura im Zusammenhang mit dem Dashboard Plug-In bekannt (Fehlerverfolgungssystem, 2007). Als Lösungsansatz wird empfohlen, von der Version 2.1 auf Version 2.0 des Cobertura Plug-Ins zu wechseln. Versionseinstellungen für Plug-Ins sind im POM des Projektes zu konfigurieren. Nach dem Versionswechsel ist die Testabdeckung korrekt durchgeführt worden.

***Zusammenfassung***

Das gesamte Softwareprojekt wurde in dieser Iteration durch eine Qualitätssicherung mittels Maven2 untersucht. Dabei wurden zum Performancegewinn einige Einheiten aus der Untersuchung ausgeschlossen, damit nur selbsterstellte Einheiten betrachtet werden. Dieses Vorgehen ergab einen Performancegewinn von 29% und somit eine Ausführungszeit von 42 min. Diese Zeitspanne ist zu lang, als dass damit ein Entwickler in seiner Arbeit kontinuierlich von seinem Computerarbeitsplatz auf diese Art eine Qualitätssicherung durchführen könnte. Durch einen Versionswechsel des Cobertura Plug-Ins wurden Fehler in der Darstellung der Testabdeckung behoben.

### 5.5.5 Integration (Iteration 5)

***Ziel***

Der erarbeitete Stand der qualitätssichernden Maßnahmen mittels Maven2 sollen in dieser Iteration in die IT-Infrastruktur des Kunden integriert werden. Der Kunde nutzt als Continuous Integration Server ein Produkt namens Continuum (Continuum, 2008). Es soll nach einer Möglichkeit gesucht werden, die Qualitätssicherung durch Continuum starten zu lassen, damit die automatisierte Durchführung der Qualitätssicherung in den *continuous integration* Ablauf aufgenommen wird. Ein täglicher Durchlauf der qualitätssichernden Maßnahmen wurde als minimales Ziel mit dem Kunden vereinbart.

***Continuous Integration Server***

Continuum stellt den in dem Projekt des Kunden eingesetzten Continuous Integration Server (CIS) dar. Continuum überprüft automatisch periodisch, ob Änderungen an den Quellen des Softwareprojektes durchgeführt wurden. Sobald eine Änderung durchgeführt wurde, startet Continuum den Erstellungsprozess von Maven2. Bei Fehlern in der Erstellung, wie z.B. durch fehlgeschlagene Tests oder Fehlern im Quelltext, wird das Entwicklerteam per Email benachrichtigt und eine Fehlermeldung auf der Projektwebseite des CIS bereitgestellt. Somit wird durch einen CIS das Ziel verfolgt, dass Änderungen von Einheiten keine Fehler in der Erstellung und in den Tests aufweisen. Dadurch wird sichergestellt, dass andere Mitglieder des Teams mit den geänderten Quelltexten arbeiten können. Die fachliche Korrektheit kann dabei nur über entsprechende qualitative Tests gewährleistet werden. Folgende Grafik zeigt das Vorgehen von Continuum auf:

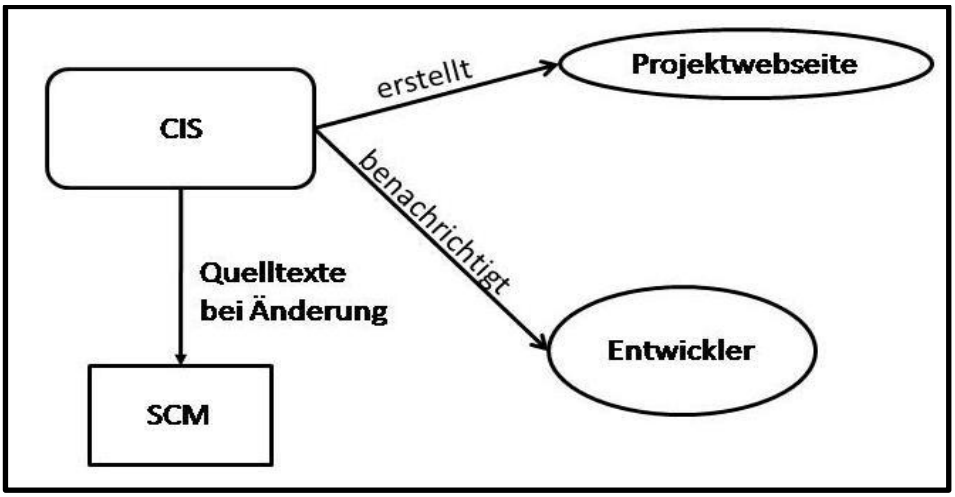



**Abbildung 19: Integration CIS**

Continuum ist in dem Projekt so konfiguriert, dass jedes modulare Softwareprojekt einzeln, ohne die Vererbungshierarchie aus Maven2, auf Änderungen der dazugehörigen Quelltexte reagiert. Das bedeutet, dass nur Teilprojekte des Gesamtsoftwareprojektes durch Continuum getestet werden. Das gesamte Softwareprojekt unterliegt einer täglichen Erstellung durch Continuum. Dieser Vorgang beginnt zyklisch morgens um 02.00 Uhr.

Die eingesetzte Version und Konfiguration des Continuum Servers kann nur auf den Erstellungslebenszyklus von Maven2 zugreifen. Für die qualitätssichernden Maßnahmen für diese Arbeit benötigt man jedoch Zugriff auf den Seitengenerierungslebenszyklus. Somit ist eine direkte Integration der Qualitätssicherung in die vorhandene Konfiguration des Continuum nicht möglich.

Als Alternative ist der Einsatz eines Skripts in Betracht gezogen worden, welches auf dem Trägersystem des CIS ablaufen soll. Dafür ist ein Ordner nötig, in dem die Quelltexte aus dem SCM-Werkzeug geladen werden können. Auf diese Quellen wird dann ein automatisch ablaufendes Skript angewandt, welches die Maven2 Befehle enthält, die auch in den vorherigen Iterationen genutzt wurden.

### *Konfigurationsarbeit*

Zunächst wurde ein Shell-Skript erstellt, welches die benötigten Befehle für die Ausführung der Qualitätssicherung beinhaltet. Zusätzlich wurde ein Befehl integriert, der die aktuellen Quelltexte aus der Konfigurationsmanagementlösung des Kunden lädt. Damit eine automatisierte Ausführung auf dem Linuxbetriebssystem möglich ist, ist ein `cron` Auftrag erstellt worden, der das Shell-Skript jeden Wochentag um 04.00 Uhr morgens ablaufen lässt.

Die Einstellungen der Plug-Ins, die für die Qualitätssicherung mittels Maven2 benötigt werden, wurden so getroffen, wie in dem letzten Stand der Iteration 4. Zu diesem Zeitpunkt ist ein Stand erreicht, der dem Kunden eine Programmierstilkontrolle und die Ermittlung von Metriken ermöglicht. Damit die Einstellungen nicht immer wieder manuell eingefügt werden müssen, sind die Einstellungen des POM, sowie der Ordner mit den Konfigurationsdateien in das SCM-Werkzeug des Kunden hinzugefügt worden. Dadurch sind die Einstellungen nun von jedem Computerarbeitsplatz und Server aus erreichbar und verfügbar.

Eine Änderung in den Einstellungen im Vergleich zu Iteration 5 wurde getätigt. Der Speicherort, an den der `mvn site-deploy` Befehl die erstellten Reporte ablegt, wurde auf den im Projekt eingesetzten Webserver umgeleitet. Dadurch sind die Ergebnisse nun für jedes Mitglied der Projektteams einsehbar.

### *Ausführung*

Das eingesetzte Shell-Skript lief automatisiert jeden Arbeitstag um 04.00 Uhr morgens und veröffentlichte durch die Maven2 Befehle die Ergebnisse auf dem projektinternen Webserver. Die Ergebnisse der Qualitätssicherung sind äquivalent zu denen aus Iteration 5.

### *Probleme*

Da nun die Änderungen an dem POM der Softwareprojekte für alle Mitglieder automatisch verfügbar waren, wurden die qualitätssichernden Plug-Ins auf jedem Rechner ausgeführt, sobald der `mvn site` Befehl aufgerufen wurde. Der Kunde liefert alle sechs Wochen neue Versionen an seinen Auftraggeber aus. Die Auslieferung wird unter anderem über eine Reihe von Maven2 Befehlen durchgeführt. Darunter befindet sich auch die Seitengenerierung. Somit wurde die Ausgabe neuer Versionen um 40 min verlängert, diese Zeitspanne ist für den Kunden zu lang. Damit nur noch gezielt mit dem Seitengenerierungslebenszyklus die Qualitätssicherung gestartet werden kann, und im Standardverhalten die Qualitätssicherung nicht ausgeführt wird, wurde ein Profil für das POM des Projektes eingeführt.

Profile ermöglichen die kontextbezogene Konfiguration eines Projektes. So kann die Ausführung von Plug-Ins an ein Profil gebunden werden, welches nur beim expliziten Aufruf dieses Profils ausgeführt werden. Ein Profil stellt ein POM innerhalb einer POM dar. Profile haben wie das POM eine Unterteilung der Lebenszyklen in <*build*> und <*reporting*> und

müssen einen eindeutigen Namen haben. Für diese Arbeit wurde ein Profil namens *dashboard* mit folgendem Aufbau erstellt:

```
<projekt>
      POM spezifische Informationen
      …
      <profiles>
            <profile>
            <id>dashboard</id>
            <activation>
                  <activationByDefault> false <activationByDefault>
            </activation>
            <reporting>
            Hier werden die Plug-Ins für die Qualitätssicherung eingetragen
            </reporting>
            <build>
            Hier werden weitere Einstellungen der Plug-Ins für die
            Qualitätssicherung beschrieben
            </build>
            </profile>
      </profiles>
      Weitere Einstellungen dem POM
      <reporting>
            …
      </reporting>
      <build>
            …
      </build>
      …
</project>
```

Das Profil stellt in diesem Fall eine Kapselung der Plug-Ins für die Qualitätssicherung von allen anderen Plug-Ins und Abläufen dar. Das Profil wird über einen zusätzlichen Parameter der Maven2 Befehle aufgerufen:

```
mvn site –Pdashboard
```

Hierdurch wird der Ablauf des Deployment des Kunden nicht weiter verzögert und für diese Arbeit müssen alle genutzten Befehle in den Skripten um den „–Pdashboard“ Parameter erweitert werden.

Als weiteres Problem zeigt sich, dass der Ablauf der Qualitätssicherung zu einer Wechselwirkung mit dem CIS des Kunden auf dem Server führte. So schlugen regelmäßig die nächtlichen Abläufe von Continuum fehl. Einige Tests konnten nicht korrekt durchgeführt werden, da anscheinend der Zeichensatz geändert wurde und somit Sonderzeichen nicht mehr korrekt erkannt wurden. Wenn die Qualitätssicherung eine Nacht ausgeschaltet wurde, trat das Fehlverhalten der Tests nicht auf. Eine dreitägige Fehlersuche mit dem Mitarbeiter, der Maven2 und Continuum in das Projekt des Kunden einführte, führte zu keinen Ergebnissen. Auch die Recherche auf den Fehlerverfolgungsseiten von Maven2 und Continuum zeigten keine Lösung auf. Continuum und die Qualitätssicherung werden beide unter demselben Account auf dem Linuxserver durchgeführt. Somit besitzen beide zusammen nur ein Repository für Maven2 was durch folgende Grafik verdeutlicht wird:

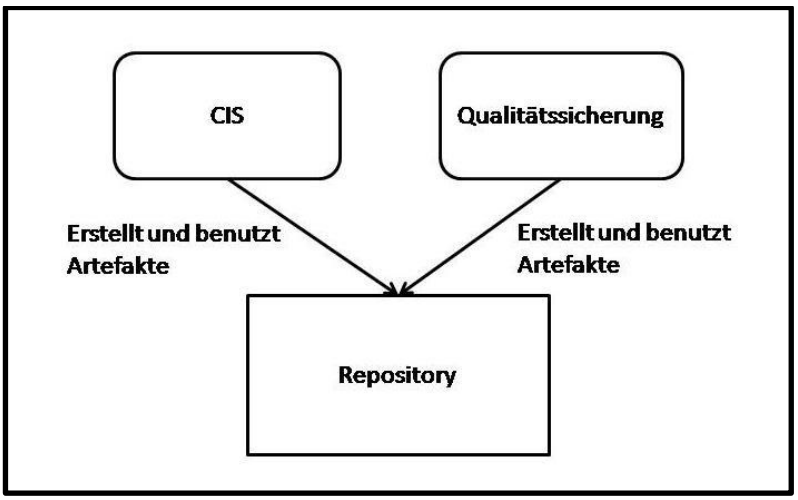



**Abbildung 20: Wechselwirkung CIS und Qualitätssicherung**

Ein Verdacht geht dahin, dass ein Plug-In der Qualitätssicherung die Ergebnisse der Testdurchläufe in dem gemeinsamen Repository speichert und Continuum diese Daten benutzt. An dieser Stelle kann dann eine Änderung des Zeichensatzes geschehen sein, welches den korrekten Durchlauf der Tests unter Continuum verhindert. Leider konnte kein genauer Beweis für diesen Verdacht gefunden werden. Ein Lösungsansatz ist, Continuum und die Qualitätssicherung auf zwei unterschiedliche Maven2 Repositories zu verteilen. Jedoch ist der Speicherplatz zum Zeitpunkt der Erstellung dieser Arbeit zu gering. Anfang 2008 soll eine

Migration auf neue Server vorgenommen werden und dann wird auch die Qualitäts-sicherung unter einem eigenen Account laufen und die Speicherplatzprobleme werden gelöst sein.

***Zusammenfassung***:

Eine Integration der Qualitätssicherung in die CIS Umgebung ist in der derzeitigen Konfiguration den CIS nicht möglich. Stattdessen wurde ein Skript erstellt, das die benötigten Maven2 Befehle automatisch zu einer gegeben Zeit ausführt. Probleme mit dem Auslieferungsvorgang des Kunden wurden durch die Einführung eines Profils in dem POM gelöst. Continuum als eingesetzter CIS und die Durchführung der Qualitätssicherung haben Wechselwirkungen aufgezeigt, deren Ursache nach mehrtägiger Recherche nicht gefunden werden konnte. Bei der anstehenden Migration der Serverumgebung des Kunden werden der CIS und die Qualitätssicherung auf getrennten Systemen laufen und somit die Wechselwirkungen umgangen.

## 5.6 Evaluation

Der bisher erreichte Stand durch den Einsatz des Dashboard Plug-Ins wird nun mit dem erarbeiteten Qualitätsmodell verglichen. Das Qualitätsmodell aus Kapitel 4.2 (vgl. Kapitel 4.2 Seite 20 ff.) stützt sich auf sechs Kategorien namens Kundenanforderungen, Programmierstil, Fehler, Design, Dokumentation und Management. Daraus abgeleitete Kriterien wurden in einer ABC-Analyse dokumentiert. Es soll ermittelt werden, wie und welche Qualitätskriterien durch den Einsatz des Dashboard Plug-Ins und allen dazugehörigen Plug-Ins abgedeckt werden. Dazu wurde die Tabelle der ABC-Analyse aus Kaptitel 4.2.7 (vgl. Seite 36) um drei Spalten erweitert. Diese Spalten stellen dar, ob eine Metrik mittels des Dashboard Plug-Ins berechenbar ist, ob die Darstellung der Ergebnisse übersichtlich ist und welches Plug-In genau für die Ermittlung der geforderten Metriken einzusetzen wäre.

| Kategorie | Metrik | automatisierbar | berechenbar | Übersicht | Plug-In |
|---|---|---|---|---|---|
| Fehler | Fehler | ja | ja | gut | FindBugs |
| Fehler | Testabdeckung | ja | ja | sehr gut | Cobertura |
| Design | Kopplung | ja | ja | sehr gut/schlecht | JDepend |
| Design | Non Commenting Source Statements | ja | nein | nein | JavaNCSS |
| Design | Lines of Code | ja | nein | nein | JavaNCSS |
| Design | Number of Classes | ja | ja | sehr schlecht | JDepend |
| Design | Abstraktheit | ja | ja | sehr schlecht | JDepend |
| Design | Instabilität | ja | ja | sehr schlecht | JDepend |
| Design | Distanz | ja | ja | sehr schlecht | JDepend |
| Design | Change Rate | ja | nein | nein | Change-Log |
| Stil | Eigener Stil | ja | ja | gut | PMD,CheckStyle |
| Stil | Stilmittel | ja | ja | gut | PMD,CheckStyle |
| Dokumentation | schriftliche Doku | nein | nein | nein | - |
| Dokumentation | JavaDoc | ja | ja | mittel | PMD,CheckStyle |
| Dokumentation | UML | nein | nein | nein | - |
| Management | Fortschritt | ja | nein | nein | - |
| Management | Budget | nein | nein | nein | - |
| Management | Produktivität | nein | nein | nein | - |

**Tabelle 6: ABC-Analyse Kap. 4.2.7 abgeglichen mit Realisierung**

Tabelle 7 zeigt, dass die Kategorien Programmierstil und Fehler aus dem Qualitätsmodell vollends umgesetzt werden konnten. JavaDoc Kommentare, die einzige maschinell erstellbare Metrik auf der Kategorie Dokumentation, konnten ebenfalls ermittelt und dargestellt werden. Dabei ist die Darstellung jedoch in den Ergebnissen des CheckStyle Reports enthalten, und ist somit nicht als einzelne Metrik darstellbar.

Die geforderten Designmetriken wie Kopplung, Komplexität, Abstraktheit, Instabilität und Distanz und die Volumenmetrik *Number of Classes* sind allesamt von dem Plug-In namens

JDepend berechnet worden. Bei der Repräsentation der Designmetrikenergebnisse fehlen die Designmetriken, außer der Metrik der ausgehenden Kopplung, auf den Übersichtsseiten des Dashboard. Dieses Problem wurde bereits in der Iteration 1 der konkreten Umsetzung erkannt.

Weitere gewünschte Volumenmetriken wie Lines of Code (LOC) und Non Commenting Source Statements (NCSS) sind mit den eingesetzten Plug-Ins gar nicht berechenbar und entsprechend nicht zu repräsentieren. Hierfür müsste das Plug-In namens JavaNCSS eingesetzt werden, dessen Daten aber nicht vom Dashboard Plug-In aggregiert werden. Die Folge daraus ist, dass nicht über eine zentrale Projektwebseite auf die Ergebnisse der Volumenmetriken zugegriffen werden kann.

Fortschrittskontrolle und die Änderungsrate von Softwareeinheiten werden wie Teile der Volumenmetriken nicht im Dashboard unterstützt. Für die Fortschrittkontrolle wurde bisher kein geeignetes Plug-In für Maven2 erstellt. Solch ein Plug-In müsste somit selbst erstellt werden. Die Änderungsrate (Change-Rate) kann durch ein Plug-In namens ChangeLog realisiert werden. Die Ergebnisse werden aber ebenso wie LOC oder NCSS nicht mittels des Dashboard Plug-Ins aggregiert und auf einer zentralen Webseite festgehalten.

Zusammen mit dem Entwicklerteam wurden die Ergebnisse nun auf die Einsatzfähigkeit hin untersucht. Die vom Dashboard erstellten Diagramme und Statistiken stellen eine gute Übersicht für die Programmierstilkontrolle, Testabdeckung und mögliche Fehler im Softwareprojekt dar. Hier können die häufigsten Fehlerquellen und Verstöße gegen die Programmierstilrichtlinien schnell erfasst werden. Die nächste detailierte Darstellung stellt die Verteilung der Verstöße und Fehler der einzelnen Module des Softwareprojektes im Verhältnis zur Gesamtzahl an Fehlern und Verstößen dar. Dadurch können die Maßnahmen, diese Fehler und Verstöße zu beheben, für ein Projekt geplant werden, welches zurzeit die meisten Probleme darstellt.

Die Ergebnisse der Qualitätssicherung sind immer nach den Plug-Ins, welche die Ergebnisse liefern, sortiert. Es ist nicht möglich, gezielt eine bestimmte Metrik für das gesamte Softwareprojekt abzufragen. Entweder die entsprechende Metrik wird auf der Übersichts-seite dargestellt, oder es ist nötig, die einzelnen Einheiten manuell auf der Webseite nach den entsprechenden Metriken zu durchsuchen. Im Falle der zyklomatischen Komplexität wird diese Metrik nicht dargestellt. Es gibt also keine durchschnittliche CCN für das gesamte

Softwareprojekt. Stattdessen kann die CCN für die Pakete der einzelnen Module ermittelt werden. Dazu muss auf der Projektwebseite zu dem zu untersuchenden Projekt navigiert und die Informationen an dieser Stelle aus dem JDepend Report entnommen werden. Die Dashboard Projektwebseite ist rein statisch, und es ist keine Abfrage von Informationen und keine Suche integriert.

Da die Erstellung der Ergebnisse der Qualitätssicherung mittels Maven2 42 min beansprucht, können die Entwickler aus Zeitmangel nicht kontinuierlich die Qualitätssicherung des gesamten Projektes von ihrem Arbeitsplatz starten. Die jede Nacht aus den Ergebnissen der Qualitätssicherung generierte Projektwebseite bietet eine gute Übersicht über den derzeitigen Stand der Qualität des Projektes. Jedoch ist die Behebung der Verstöße und Fehler dort nicht möglich. Zwar können mögliche Fehler bis in den Quelltext auf der Webseite angezeigt werden, jedoch kann der Quelltext von der Projektwebseite aus nicht geändert werden. Das parallele Arbeiten mit Hilfe eines Browser, sowie das Suchen der benötigten Stelle aus der Entwicklungsumgebung heraus, erwies sich als zu aufwändig. Zwar werden jetzt mögliche Problemfelder visualisiert, es können aber keine Lösungen direkt aus den Ergebnissen umgesetzt werden.

Die Minimierung der offenen Probleme und Abweichungen vom SOLL-Zustand der Realisierung dieser Arbeit wird im folgenden Kapitel diskutiert.

# 6 Delta Bereinigung

Dieses Kapitel beschäftigt sich mit der Differenz zwischen dem IST- und dem SOLL-Zustand der Realisierung einer Qualitätssicherung mittels Maven2. Dies ist die sechste und letzte Iteration dieser Arbeit. Probleme, für die bisher keine Lösung gefunden wurden, werden in diesem Kapitel analysiert und es wird versucht, Lösungsansätze für diese Probleme aufzuzeigen. Zum derzeitigen Stand der Realisierung sind vier Problemfelder gefunden worden und werden nun stichwortartig vorgestellt:

- Metriken
  Einzelne geforderte Metriken werden durch den Einsatz des Dashboard Plug-Ins nicht abgedeckt
- Eigener Programmierstil
  Der intern im Entwicklerteam definierte Programmierstil wird durch den Einsatz des Dashboard Plug-Ins abgedeckt. Zukünftige Erweiterungen des Programmierstils sollen aber durch den Einsatz selbstdefinierter Regeln gewährleistet werden
- Integration
  Die Abläufe der Qualitätssicherung sollen in den Entwicklungszyklus des Entwicklerteams integriert werden. Außerdem ist eine Integration in die IT-Infrastruktur des Kunden nötig
- Präsentation
  Viele Metriken können errechnet aber nicht auf der Übersichtsseite des Dashboard angezeigt werden

## 6.1 Metriken

Bisher werden Volumenmetriken wie LOC und NCSS, die Wechselrate (CR) und der Fortschritt des Projekts nicht berechnet. LOC und NCSS können durch das JavaNCSS Plug-In abgedeckt werden, welches zu den wenigen Maven2 Plug-Ins gehört, die einen eigenen Aggregationsmechanismus ausweisen. Die Konfiguration ist nicht aufwändig, JavaNCSS muss lediglich in dem POM des Softwareprojektes eingetragen werden. Anschließend werden Volumenmetriken wie die Anzahl der Pakete, Anzahl der Klassen, Anzahl der Methoden, NCSS, JavaDoc und Anzahl der JavaDoc-Zeilen eines Softwareprojektes berechnet und in einem einzigen Report angezeigt.

Die Wechselrate von Einheiten wird durch ein Plug-In namens Changelog berechnet. Changelog generiert drei Reports. Der Changelog Report zeigt die letzten Änderungen mit den Kommentaren aus dem SCM-Werkzeugt an. Im Entwickler Aktivitätsreport ist die Anzahl an Änderungen, die die Entwickler gemacht haben, enthalten. Der letzte Report namens Datei Aktivitätsreport stellt eine sortierte Liste der am meisten geänderten Dateien bereit. Dabei ist die Datei mit den meisten Änderungen auf Platz eins. Auch dieses Plug-In muss in dem POM des Softwareprojektes konfiguriert werden. Dabei können einzelne zu erstellende Reports gewählt werden. Für dieses Projekt genügt die Ermittlung des Datei Aktivitäts-reports, da diese eine direkte Abbildung der Wechselratenmetrik darstellt. Es ist möglich, einen Zeitraum zu definieren, aus dem die Änderungen der Dateien betrachtet werden.

Die Plug-Ins für die Volumenmetriken und die Wechselrate werden nicht auf der Dashboard-übersichtsseite dargestellt, da diese Plug-Ins derzeitig nicht vom Dashboard unterstützt werden. Stattdessen werden eigene Reports erstellt, die dann von den Entwicklern auf der Projektwebseite eingesehen werden können.

Die Fortschrittsmetrik aus dem Qualitätsmodell für dieses Softwareprojekt kann derzeit von keinem Maven2 Plug-In realisiert werden. Da es sich laut ABC-Analyse um eine „nice to have“ Metrik handelt, wird diese Metrik zu diesem Zeitpunkt nicht in die Konfiguration der Qualitätssicherung aufgenommen. Sollte aber zu einem späteren Zeitpunkt diese Metrik benötigt werden, so muss ein eigenes Plug-In zur Ermittlung erstellt werden. Die Erstellung eigener Plug-Ins ist auf der Herstellerseite von Maven2 beschrieben (Mojo, 2008).

## 6.2 Eigner Programmierstil

Der Programmierstandard des Entwicklerteams kann mit der aktuellen Konfiguration der Qualitätssicherung abgedeckt werden. Dennoch empfiehlt es sich für den späteren Gebrauch nach Möglichkeiten zu suchen, wie der Programmierstil um eigene Regeln erweitert werden kann. Dazu sind in PMD und CheckStyle bereits Funktionen bereitgestellt.

Auf der Herstellerseite von CheckStyle (Burn, 2007) gibt es eine detailierte Anleitung, wie eigene Regeln erstellt und verbreitet werden können. Dabei ist der Ablauf der Erstellung gut beschrieben und die benötigten Entwicklerbibliotheken werden bereitgestellt. Die Regeln werden in CheckStyle als Javaklassen repräsentiert. Zur Integration in die Qualitätssicherung wird die selbsterstellte Regel in die XML-Konfigurationsdatei von CheckStyle als Modul

geschrieben. Danach wird die Regel automatisch angewandt, sobald das CheckStyle Plug-In ausgeführt wird.

PMD ist auch um eigene Regeln erweiterbar. Eine detailierte Anleitung dazu befindet sich auf der Herstellerseite (PMD, 2007). Für die Erstellung von eigenen Regeln kann zwischen zwei Varianten gewählt werden. Zum einen kann äquivalent zu CheckStyle vorgegangen werden, indem eine Javaklasse die Regel repräsentiert und anschließend als Modul in die XML-Konfigurationsdatei des PMD Maven2 Plug-Ins geschrieben wird. Zum anderen kann eine Regel durch eine XPath Regel repräsentiert werden. XPath (XPath, 1999) ist eine Abfragesprache, die einzelne Abschnitte einer XML-Datei auslesen kann. Dieses Vorgehen wird von den PMD Entwicklern der Erstellung einer Javaklasse vorgezogen, da es sich laut Hersteller um die einfachere Variante handeln soll.

Durch die gute Dokumentation und Anleitung der Hersteller von CheckStyle und PMD ist in einem zu verantwortbaren Zeitraum die Einführung eigener Programmierstilregeln, für die in dieser Arbeit entwickelte Qualitätssicherung, möglich.

## 6.3 Integration

In Iteration 5 wurde bereits versucht die derzeitige Lösung in die IT-Infrastruktur zu integrieren. Dabei kam es zu Wechselwirkungen mit dem eingesetzten Continuous Integration Server namens Continuum. Zu einem späteren Zeitpunkt wird ein großer Teil der IT-Infrastruktur auf neue Hardware und Anbieter migriert. Dann wird auch die automatisierte Qualitätssicherung durch Maven2 einen eigenen Hardwarekontext erhalten, sodass keine Wechselwirkungen mit anderen Systemen mehr vorliegen. Als Zwischenlösung, bis zu der Migration, wird jede Nacht auf einem Arbeitsplatz automatisiert die Qualitäts-sicherung durchgeführt und die Ergebnisse werden auf einem Webserver zur Verfügung gestellt.

Der eben beschriebene Ablauf stellt als Ergebnis eine Übersicht über das gesamte Softwareprojekt bereit und wird automatisiert jeden Tag ausgeführt. Die Ergebnisse auf der Projektwebseite sind vor allem für Manager und Qualitätssicherer interessant, da es die Informationen über das gesamte Softwareprojekt beinhaltet. Einem Entwickler sollte es aber ermöglicht werden, seine erstellte Arbeit zeitnah von seinem Arbeitsplatz aus einer Qualitätssicherung zu unterziehen. Die Zeit, die dieser Ablauf durch Maven2 dauert, ist zu

lang. Es kann dem Entwickler nicht zugemutet werden, zwischen zehn und 40 min auf ein Ergebnis zu warten. Für diesen Einsatzzweck und der Größe des Softwareprojekts ist eine Maven2 Erweiterung für eine Qualitätssicherung nicht tauglich. Ein Entwickler benötigt ein Werkzeug, welches mögliche Fehler und Verstöße schnell und direkt am Quelltext ermittelt. Im Rahmen dieser Arbeit bedeutet das, dass die Entwicklungsumgebung (Eclipse) so erweitert wird, dass die Erweiterungen denselben Regeln und Kriterien folgen, wie die Maven2 Lösung. Die Erweiterungen können entweder automatisiert oder bei Bedarf auf den Quelltext angewandt werden. Da die Maven2 Plug-In grundlegend nur bestehende Werkzeuge erweitert, können die in dieser Arbeit erstellten Konfigurationsdateien direkt in die Entwicklungsumgebungserweiterungen integriert werden

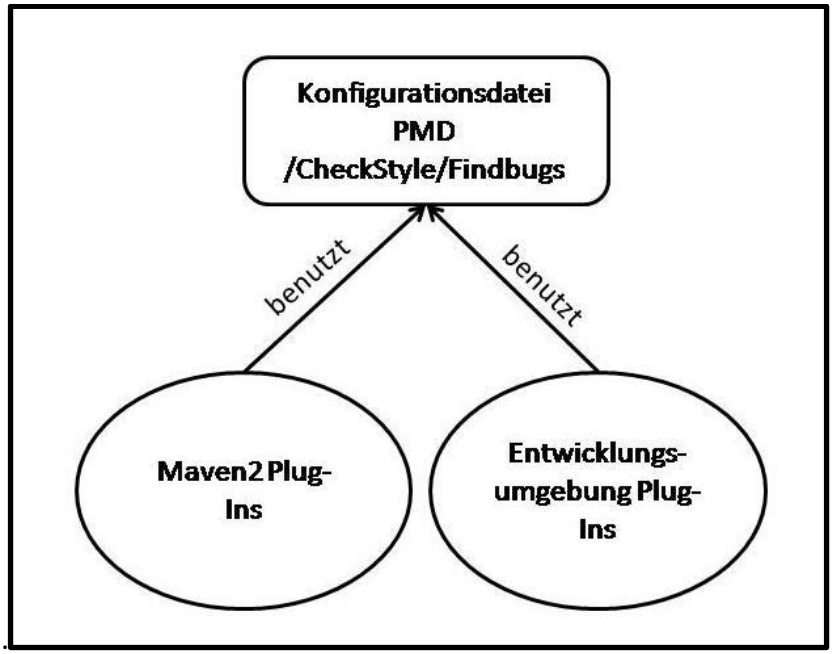



**Abbildung 21: gemeinsame Konfigurationsdatei für Maven2 und Eclipse**

Wo die einzelnen Entwicklungsumgebungserweiterungen zu finden und anzuwenden sind, wurde auf der Projektwebseite veröffentlicht. Dort findet sich auch eine Anleitung, wo die Konfigurationsdateien gefunden und wie diese in die Erweiterungen integriert werden. Alle Erweiterungen bieten eine gute Benutzerführung an und können entweder permanent aktiviert sein oder manuell gestartet werden. Die permanente Aktivierung hat aber zur Folge, dass die Performance der Entwicklungsumgebung gemindert werden kann. Es wird empfohlen, nach Fertigstellung von Quelltexten die einzelnen Erweiterungen manuell auszuführen. Die Ergebnisse werden anschließend sehr schnell direkt am Quelltext gezeigt und können entsprechend verändert werden. Durch die Anwendung dieses Workaround

werden nur qualitätsgeprüfte Quelltexte in die Konfigurationsmanagementwerkzeuge geladen. So können die bisher angefallenen Verstöße und Fehler schneller beseitigt werden.

## 6.4 Präsentation

Zwar werden nun alle geforderten Metriken durch das Dashboard, JavaNCSS und Changelog berechnet, aber die Präsentation der Ergebnisse ist nicht in jedem Fall zufriedenstellend. Vor allem die durch JDepend errechneten Metriken (A, I, D, Ca, Ce und CCN) werden nicht auf der Übersichtsseite dargestellt. Die Entwickler des Dashboard schreiben auf ihrer Webseite nicht, dass eine andere Präsentation der Ergebnisse in folgenden Versionen angestrebt wird. Um dieses Defizit zu lösen, gibt es zwei Ansätze. Der erste Ansatz ist, dass das Dashboard Plug-In selbst erweitert und verändert wird. Der zweite Ansatz wäre das Erstellen eines eignen Aggregation Plug-Ins für JDepend.

Eine Erweiterung des Dashboard Plug-Ins ist möglich, da es sich um ein Open Source Produkt handelt. Die Quelltexte sind öffentlich verfügbar. Der Aufwand für solch eine Erweiterung ist sehr hoch, da zunächst eine Einarbeitung in die fremden Quelltexte ohne Kontakt zu den Erstellern anstünde. Anschließend müsste ein Zusatzmodul gefertigt werden, das die Ergebnisse nach eigenem Wunsch auf der Übersichtsseite, der detaillierten Darstellung und dem Verlaufsreport darstellt. Hierdurch würde ein Konflikt der Versionen des Dashboard vorliegen. So müssen bei einer neuen Version des Dashboard Plug-Ins alle Änderungen an den Quelltexten nachgepflegt oder auf die neue Version verzichtet werden.

Der zweite Ansatz beschreibt die Erstellung eines eigenen Aggregation Plug-Ins für JDepend. Hier können dieselben Anleitungen für die Erstellung von eigenen Maven2 Plug-Ins, wie bei der Fortschrittskontrolle weiter oben beschrieben, genutzt werden.

Beide Lösungsansätze sind sehr zeitintensiv, denn es bedarf einer Einarbeitung in die Erstellung eigener Plug-Ins und in den Quelltext des Dashboard Plug-Ins. Dieser Lösungsansatz wird nicht weiter verfolgt, denn er sprengt den Zeitrahmen dieser Arbeit.

## 6.5 Zusammenfassung

Für drei der vier Problemfelder konnten Lösungen oder Workarounds gefunden werden. Lediglich die Präsentation der Ergebnisse wird im Rahmen dieser Arbeit nicht weiter verfolgt, da die Umsetzung der dargestellten Lösungswege sehr zeitintensiv ist.

# 7 Fazit

In diesem Kapitel werden die realisierten Ergebnisse dieser Arbeit zusammenfassend dargestellt. Zukünftige Aufgaben und mögliche Weiterentwicklung dieses Projektes sollen auch erläutert werden.

## 7.1 Erreichter Stand

In einem bereits zu Beginn der Arbeit vorangeschrittenen Softwareprojekt wurden qualitätssichernde Maßnahmen eingeführt, die durch das eingesetzte Buildmanagement-werkzeug Maven2 automatisiert durchgeführt werden. Dabei wurden zunächst Kriterien für Softwarequalität, die auf das Softwareprojekt des Kunden angewandt werden sollen, definiert. Diese Kriterien galt es dann durch Maven2 Erweiterungen umzusetzen. Durch die Konfiguration verschiedener Plug-Ins werden ein Großteil der geforderten Metriken und Regeln auf einer zentralen Webseite des Kunden angezeigt. Maven2 bietet für regelmäßige Qualitätssicherung die nötigen Werkzeuge. Die Ergebnisse sind für die Rollen des Managers und Qualitätssicherers des Softwareprojektes nutzbar. Jedoch ist die Dauer der Durch-führung bei dieser Softwareprojektgröße für die Rolle des Entwicklers und Software-architekten an ihrem Arbeitsplatz zu lang. Damit auch die Entwickler und Softwarearchitekten zeitnahe ihren eigenen Quelltext prüfen können, wurde nach Erweiterungen für die eingesetzte Entwicklungsumgebung gesucht. Alle eingesetzten Maven2 Plug-Ins sind auch als Erweiterung für die Entwicklungsumgebung vorhanden und können einfach und schnell mit der gleichen Konfiguration wie die Maven2 Plug-Ins ausgestattet werden. Das zusammen mit dem Kunden ermittelte Qualitätsmodell wurde größtenteils mittels Maven2 realisiert und steht dem Kunden nun zur Verfügung.

## 7.2 Ausblick in die Zukunft

Zukünftig wird zunächst eine Schulung stattfinden, in der dem Entwicklerteam sowohl die Konfigurationsabläufe dieser Arbeit dargestellt und gezeigt werden, als auch die Ergebnisse zu nutzen sind.

Damit bei Bedarf eigene Erweiterungen des Systems realisiert werden können, sind die Schritte wie in Kapitel 6.2 und 6.4 beschrieben durchzuführen.

Sobald die Migration der Hardware auf ein neues System beim Kunden durchgeführt wurde, wird die in dieser Arbeit dargestellte Lösung auf einem eigenen System oder in einem eigenen Systemkontext laufen.

# 8 Lessons Learned

Qualitätssicherung ist kein reines Anwendungsgebiet von Werkzeugen. Um eine Qualitätssicherung für ein Softwareprojekt durchzuführen, bedarf es zunächst einer gründlichen Planung, in der gemeinsam mit dem Entwicklerteam Qualitätsstandards definiert werden müssen. Dazu benötigt man viele Quellen der Literatur, die sich mit dem Thema Softwarequalität beschäftigen. In der Literatur finden sich vielfältige Ansätze und Verfahren zur Durchführung einer Qualitätssicherung, die dann auf das Realprojekt angewandt werden können.

Eine wichtige Erkenntnis im Rahmen dieser Arbeit ist, dass der Faktor Zeit eine wichtige Rolle für Entwickler spielt. So darf eine Durchführung der Qualitätssicherung den Entwickler nicht zu lange von der Arbeit abhalten. Der Zeitraum, der für eine Durchführung akzeptiert wird, ist von Team zu Team unterschiedlich. Jedoch ist in diesem Fall eine Durchführung von 40 min definitiv zu lang.

Maven2 bietet aufgrund seiner Plug-In Architektur weit mehr Möglichkeiten als ein reines Erstellungswerkzeug. So hat sich Maven2 in dem Anwendungskontext dieser Arbeit als gutes und flexibles Werkzeug dargestellt. Ebenso sind fast alle Komponenten und Plug-Ins für Maven2 Open Source und somit selbständig erweiterbar. Solche Erweiterungen sind jedoch sehr zeitintensiv und deshalb für die meisten Projekte, die einem Zeitrahmen unterliegen, nicht nutzbar. Für Erweiterungen dieser Art müsste sich ein Entwickler für den Zeitraum der Erstellung ausschließlich damit beschäftigen können.

# 9 Persönliche Meinung

Diese Arbeit soll dem Leser vermitteln, dass für eine Qualitätssicherung nicht nur Werkzeuge eingesetzt werden können. Am Anfang der Planung und Durchführung von qualitätssichernden Maßnahmen steht die Definition eines für das Projekt geltenden Qualitätsmodelles. Aus solch einem Modell lassen sich beispielsweise durch das Ziel-Frage-Metrik Verfahren benötigte Metriken für ein Projekt ermitteln. Anschließend kann in einer Marktanalyse nach geeigneten Werkzeugen für die ermittelten Metriken gesucht werden.

Maven2 bietet für das erstellte Qualitätsmodell geeignete Plug-Ins. Es lassen sich alle gewünschten Metriken errechnen. Lediglich die Repräsentation der Ergebnisse ist mangelhaft und lässt sich nur mit viel Aufwand ändern. Als Arbeitsplatzlösung für Qualitätssicherung braucht Maven2 bei der Größe dieses Softwareprojekts zu lange, um Ergebnisse zu liefern. Somit ist die vorgestellte Lösung nicht als Arbeitsplatzlösung einsetzbar.

Stattdessen lässt sich Maven2 gut als Werkzeug für eine Übersicht über die Qualität nutzen. Dabei ist besonders der in Maven2 integrierte Mechanismus der Seitengenerierung nützlich. Dadurch wird dem gesamten Team z.B. täglich eine Übersicht über den Stand des Projekts gegeben.

## Literaturverzeichnis

Burn, O. (2007). *CheckStyle*. Abgerufen am 11. Januar 2008 von CheckStyle: http://checkstyle.sourceforge.net/

Cobertura. (13. Dezember 2007). *Cobertura Maven Plugin*. Abgerufen am 11. Januar 2008 von http://mojo.codehaus.org/cobertura-maven-plugin/

Continuum. (06. Januar 2008). *Welcome to Continuum*. Abgerufen am 15. Januar 2008 von http://maven.apache.org/continuum

Dashboard. (09. Juli 2007). *About Custom Configuration of historic support for Multimodules project*. Abgerufen am 14. Januar 2008 von http://mojo.codehaus.org/dashboard-maven-plugin/custom_multi_config.html

Fehlerverfolgungssystem. (25. August 2007). *Issue Navigator*. Abgerufen am 15. Januar 2008 von Codehaus: http://jira.codehaus.org/secure

Fenton, N. E., & Pfleeger, S. L. (1997). *Software Metrics A Rigorous & Practical Approach 2nd E.* Revised Printing.

FindBugs. (20. August 2007). *Maven2 Findbugs Plugin*. Abgerufen am 11. Januar 2008 von http://mojo.codehaus.org/findbugs-maven-plugin/

Group, A. (08. Januar 2008). *Apache Maven Project*. Abgerufen am 18. September 2007 von http://maven.apache.org

Hortis. (10. Januar 2008). *sonar*. Abgerufen am 12. Januar 2008 von http://sonar.hortis.ch/

IEEE 1061, I. C. (8. November 1999). IEEE Standard for a Software Quality Metrics Methodology. www.ieee.de.

ISO, 9. (20. September 2005). *Quality management systems - Fundamentals and vocabulary.* Abgerufen am 21. Januar 2008 von http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=42180

JavaNCSS. (12. August 2007). *Maven2 JavaNCSS Plugin*. Abgerufen am 11. Januar 2008 von http://mojo.codehaus.org/javancss-maven-plugin/

JDepend. (12. August 2007). *JDepend Maven Plugin*. Abgerufen am 11. Januar 2008 von http://mojo.codehaus.org/jdepend-maven-plugin/

JXR. (08. April 2007). *Maven2 JXR Plugin*. Abgerufen am 11. Januar 2008 von http://maven.apache.org/plugins/maven-jxr-plugin/

Martin, R. C. (2003). *Agile Software Develpment.* Upper Saddle River New York: Prentice Hall.

McCabe, T. J. (Dezember 1976). *ccn.* Abgerufen am 2008. 01 2008 von literateprogramming: http://www.literateprogramming.com/mccabe.pdf

Mojo. (16. Januar 2008). *Plugin Development Center*. Abgerufen am 19. Januar 2008 von http://maven.apache.org/plugin-developers/index.html

Mojo, C. C. (11. Juni 2007). *mojo*. Abgerufen am 11. Januar 2008 von mojo: http://mojo.codehaus.org/

PMD, I. (2007). *PMD*. Abgerufen am 11. Januar 2008 von PMD: http://pmd.sourceforge.net/

QALab. (13. November 2006). *QALab Module*. Abgerufen am 12. Januar 2008 von http://qalab.sourceforge.net/multiproject/qalab/index.html

Selenium. (2006). *Selenium*. Abgerufen am 10. Januar 2008 von OpenQA: http://www.openqa.org/selenium/

sonatype. (08. Januar 2008). *sonatype*. Abgerufen am 8. Januar 2008 von http://www.sonatype.com/book/pom-relationships.html#the_pom

Sun. (20. April 1999). *Coding Standards for Java*. Abgerufen am 19. Januar 2008 von http://java.sun.com/docs/codeconv/

Surefire. (14. Juni 2007). *Maven Surefire Plugin*. Abgerufen am 11. Januar 2008 von http://maven.apache.org/plugins/maven-surefire-plugin/

XPath, w. (16. November 1999). *XML Path Language*. Abgerufen am 19. Januar 2008 von http://www.w3.org/TR/xpath

# Glossar

**A**: Abstractness (Abstraktheit)

**Ca**: Eingehende Kopplung

**Ce**: Ausgehende Kopplung

**CCN**: Zyklomatische Komplexität nach McCabe

**CIS**: Continuous Integration Server

**CR**: Change Rate (Wechselrate)

**D**: Distance (Distanz)

**GQM**: Goal-Question-Metrik (Ziel Frage Metrik) Verfahren

**I**: Instability (Instabilität)

**LOC**: Lines of Code

**NC**: Number of Classes

**NCSS**: Non Commenting Source Statements

**SCM**: Source Code Management

**TC**: Test Coverage (Testabdeckung)

# Anhang

## Abbildungs- und Tabellenverzeichnis

Abbildung 1:Ziel-Frage-Metrik Baum ..... 7
Abbildung 3: Qualitätsmodell ..... 21
Abbildung 4: Ablaufgraph der HelloNr Klasse ..... 27
Abbildung 5: Minimaler Anwendungsgraph ..... 28
Abbildung 6: Main Sequence ..... 31
Abbildung 7: Checkstyle Auszug ..... 38
Abbildung 8: PMD Auszug ..... 40
Abbildung 9: Cobertura Report ..... 41
Abbildung 10: JavaNCSS Auszug ..... 42
Abbildung 11: JDepend Auszug ..... 43
Abbildung 12: FindBugs Auszug ..... 44
Abbildung 13: Sonar im Einsatz ..... 47
Abbildung 14: QALab Auszug ..... 48
Abbildung 15: Dashboard Auszug ..... 49
Abbildung 16: JDepend Auszug Übersichtsseite Dashboard ..... 56
Abbildung 17: Auszug Dashboard Report 1 ..... 63
Abbildung 18: Auszug Dashboard Report 2 ..... 64
Abbildung 19: Auszug Dashboard Report 3 ..... 64
Abbildung 20: Integration CIS ..... 68
Abbildung 21: Wechselwirkung CIS und Qualitätssicherung ..... 71
Abbildung 22: gemeinsame Konfigurationsdatei für Maven2 und Eclipse ..... 79

Tabelle 1: ABC Analyse der Anforderungen des Kunden ..... 20
Tabelle 3: Beispiel eines Change-Rate Ergebnisses ..... 32
Tabelle 4: ABC-Analyse ..... 36
Tabelle 5: Metrikenabdeckung ..... 45
Tabelle 6: Aggregation ABC Analyse ..... 50
Tabelle 7: ABC-Analyse Kap. 4.2.7 abgeglichen mit Realisierung ..... 73

## Dokumente und Softwareprojekte

Die Anhänge zu dieser Arbeit befinden sich auf der digitalen Kopie dieser Arbeit, die in der Bibliothek der HAW Hamburg eingesehen werden können.

***Adresse***:

*Hochschule für Angewandte Wissenschaften Hamburg*

*Fakultät Technik und Informatik*

*Department Informatik (2.OG)*

*Berliner Tor 7*

*20099 Hamburg*

Folgende Anhänge sind auf der CD im Ordner <*Anhang*> enthalten:

- **Metrikenauswahl_CheckStyle.pdf**
  In diesem Dokument stehen die ausgewählten und verfügbaren Kategorien und Regeln, die CheckStyle anbietet. Anhand dieses Dokuments wurden die Regeln für CheckStyle zusammen mit dem Entwicklerteam ausgewählt.
- **Metrikenauswahl_PMD.pdf**
  Dieses Dokument umfasst die Kategorien und Regeln von PMD. Anhand dieses Dokumentes wurden die Regeln für PMD zusammen mit dem Entwicklerteam ausgewählt.
- **Softwareprojekt_Iteration1.zip**
  Das selbsterstellte Softwareprojekt TEST1 (Eclipseprojekt) für die Iteration1 der konkreten Umsetzung ist hier enthalten. Die durch die Qualitätssicherung generierte Webseite liegt im Ordner <*target*>.
- **Softwareprojekt_Iteration2.zip**
  Das selbsterstellte Softwareprojekt MULTI1 inkl. der Subprojekte FIRST und SECOND (alle Projekte sind Eclipseprojekte) aus der Iteration2 der konkreten Umsetzung sind hier enthalten. Die generierten Webseiten mit den Ergebnissen der Qualitätssicherung liegen in jedem Projekt in einem Ordner namens <*target*>.

## Versicherung über Selbstständigkeit

Hiermit versichere ich, dass ich die vorliegende Arbeit im Sinne der Prüfungsordnung nach §24(5) ohne fremde Hilfe selbstständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, 7. Februar 2008

| | |
|---|---|
| _____ | _____ |
| Ort, Datum | Unterschrift |