Automatisierte Ermittlung des Code-Quality-Index

Diplomarbeit

Torsten Möllenbeck
Matrikelnummer: 1652286

05.10.2007

Fachbereich Mathematik / Informatik
Studiengang Informatik

1. Gutachter: Prof. Dr. Rainer Koschke
2. Gutachter: Dr. Berthold Hoffmann

Erklärung

Ich versichere, die Diplomarbeit ohne fremde Hilfe angefertigt zu haben. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen entnommen sind, sind als solche kenntlich gemacht.

Bremen, den 05.10.2007

.....

(Torsten Möllenbeck)

Danksagung

Mein Dank gilt allen Personen, die mich bei der Erstellung dieser Diplomarbeit unterstützt haben.

Besonders möchte ich mich bei Prof. Dr. Rainer Koschke, für die engagierte Betreuung dieser Arbeit, bedanken.

Ebenso gilt mein Dank Dr. Berthold Hoffmann, der sich freundlicherweise als Zweitgutachter zur Verfügung gestellt hat.

Ferner gilt mein Dank, den Mitarbeitern der SQS Software Quality Systems AG und hier besonders der Einheit BUPCCCCQM für höchst interessante Einblicke in die Praxis, die unermüdliche Beantwortung von Fragen zum Code-Quality-Index und nicht zuletzt für die Überlassung des Testcodes.

Auch den Mitarbeitern des Bauhaus-Projekts an den Universitäten Bremen und Stuttgart und der Axivion GmbH gebührt Dank. Sie standen mir bei der Klärung von Fragen und Problemen jederzeit unterstützend zur Seite.

Abschließend möchte ich meinen Eltern, Karin und Norbert Möllenbeck, danken, die mir das Studium ermöglicht und mich nach Kräften unterstützt haben.

INHALTSVERZEICHNIS

KAPITEL 1

Einleitung

Ein großer Teil heutiger Softwareprojekte sind sogenannte Wartungsprojekte, in denen bestehende Software überarbeitet oder weiterentwickelt wird.
Die Wartbarkeit eines Systems objektiv einzuschätzen ist daher von hoher Bedeutung, um beurteilen zu können, ob ein System überhaupt wirtschaftlich weiterentwickelt werden kann und um eine zutreffende Kosten- und Aufwandschätzung für durchzuführende Wartungsarbeiten zu ermöglichen.
Die Abschätzung der Wartbarkeit über ein Code-Review ist jedoch sehr zeit- und kostenintensiv. Um diesem Problem zu begegnen, haben Frank Simon, Olaf Seng und Thomas Mohaupt unlängst das Buch „Code-Quality-Management“ veröffentlicht, das einen sogenannten Code-Quality-Index definiert, der auf Basis automatischer Analysen des Quellcodes bestimmt werden kann. Somit ermöglicht dieser Index eine schnelle und kostengünstige Aussage über die Wartbarkeit eines Softwaresystems zu treffen.
Die Entwicklung eines Tools, das die diesem Index zugrunde liegenden Analysen automatisiert durchführt und auswertet, ist Gegenstand dieser Diplomarbeit.
Möglich wird dieses Vorhaben durch die Einbettung in das Bauhaus-Projekt, das viele notwendige Werkzeuge – wie etwa eine gut analysierbare Programmrepräsentation – bereitstellt.

1.1 Das Bauhaus-Projekt

Das Bauhaus-Projekt[1] ist ein kooperatives Forschungsprojekt der Universitäten Bremen und Stuttgart. Es wurde 1996 vom Fraunhofer Institut für Experimentelles Software Engineering in Kaiserslautern[2] und dem Institut für Softwaretechnologie der Universität Stuttgart gegründet. Als weiterer Partner übernimmt die 2006 als Spin-Off gegründete Firma Axivion GmbH die Vermarktung der Bauhaus-Suite.

Ziel des Projekts ist vor allem die Unterstützung von Wartungsingenieuren. Zu diesem Zweck stellt die Bauhaus-Suite eine Reihe von Werkzeugen, Methoden und Analysen bereit, die durch geeignete Darstellungen helfen, das zu analysierende Programm besser und schneller zu verstehen. Unter anderem können auf diese Weise bestimmte Muster und Programmkom-

[1] http://www.bauhaus-stuttgart.de/bauhaus/
[2] http://www.iese.fhg.de/fhg/iese_DE/

ponenten und ihre Zusammenhänge identifiziert werden. Weiterhin ist es möglich im Rahmen der Analyse zum Beispiel Metriken zu erheben und nicht verwendete oder duplizierte Code-Teile zu erkennen.

Als Datenbasis dienen jeweils die beiden sprachunabhängigen Programmrepräsentationen IML (***I**ntermediate **M**odelling **L**anguage*) und RFG (***R**essource **F**low **G**raph*), die sich vor allem in ihrer Granularität unterscheiden. Je nach Programmiersprache kann über verschiedene Frontends entweder direkt aus dem Quellcode (*C/C++*) die IML erzeugt werden oder über den Bytecode (*Java*) der RFG. Das Frontend *jafe* [Kna02] kann auch Java-Quellcode in die IML überführen, ist jedoch noch in Entwicklung und nicht für Endanwender verfügbar. Der RFG für C/C++ wird mit dem Tool `iml2rfg` aus der IML erzeugt.

Für beide Repräsentationsformen wurden von Axivion Scripting-Schnittstellen entwickelt, über die Analysen durchgeführt und Metriken erhoben werden können.

1.2 Aufgabenstellung

In dieser Diplomarbeit soll auf Basis der Bauhaus-Infrastruktur und des Buchs Code-Quality-Management [SSM06] ein Werkzeug entwickelt werden, das den im Buch beschriebenen Code-Quality-Index (kurz CQI) für C++ und Java (sofern die Programmrepräsentation deckungsgleich ist) berechnet.
Hierzu ist abzuwägen, ob die Implementierung über die Scripting-Schnittstellen oder direkt in Ada erfolgt.
Die Ergebnisse der Analyse sollen in Form von generierten, statischen HTML-Seiten präsentiert werden. Diese sollen so gestaltet sein, dass es möglich ist das Zustandekommen der einzelnen Werte anhand der relevanten Code-Anomalien nachzuvollziehen.
Hierbei kann die Ausgabe einen prototypischen Charakter haben, während der Schwerpunkt auf der Erhebung der Metriken liegt.

Zur Evaluation des entwickelten Werkzeugs soll eine Reihe von Softwaresystemen vermessen und in Absprache mit der Firma SQS mit dem SQS-Repository verglichen werden.

1.3 Aufbau der Arbeit

Neben dieser Einleitung gliedert sich die Arbeit in 5 weitere Kapitel.
Die für die Arbeit wesentlichen Grundlagen werden in Kapitel 2 beschrieben. Zunächst werden die Begriffe „Metriken“ und „Qualität“ behandelt und im Anschluss der so genannte Maintainability-Index sowie der bereits in der Aufgabenstellung (Abschnitt 1.2) erwähnte Code-Quality-Index näher erläutert.
Kapitel 3 stellt den gewählten Lösungsansatz vor.
Kapitel 4 betrachtet zunächst das Laufzeitverhalten der Implementierung und befasst sich im Anschluss mit der Vermessung verschiedener Software-Systeme.
In Kapitel 5 werden die Ergebnisse und der Code-Quality-Index einer kritischen Betrachtung unterzogen.
Das letzte Kapitel fasst die Ergebnisse der Arbeit zusammen und gibt einen Ausblick auf mögliche Erweiterung und Verbesserungen.
Die Anhänge beinhalten die Schwellwerte der einzelnen Qualitätsindikatoren für *Java* und *C++* wie sie in [SSM06] abgedruckt sind, sowie eine kurze Beschreibung, wie das im Rahmen der Arbeit entwickelte Werkzeug zu bedienen ist.

KAPITEL 2

Grundlagen

Dieses Kapitel bietet eine Einführung in die Materie auf der diese Arbeit aufbaut. Das Hauptaugenmerk liegt hierbei auf dem zu implementierenden Code-Quality-Index. Hierfür wird zunächst erklärt, was eine Metrik ist und wie Qualität im Kontext dieser Arbeit zu verstehen ist.
Anschließend werden der Maintainability Index und der zu implementierende Code-Quality-Index vorgestellt. Da beide etwas über die Wartbarkeit eines Software-Systems aussagen, erfolgt im Zuge der Vorstellung des CQI eine kurze Abgrenzung der beiden Indices zueinander.
Während sich die Vorstellung des Maintainability Index vornehmlich auf eine kurze Zusammenfassung der Literatur beschränkt erfolgt die Erläuterung des CQI in ausführlicherer Form und stellt neben den Zielen und dem zugrundeliegenden Konzept auch die Bedeutung der einzelnen Metriken kurz vor.
Abschließend erfolgt eine Erläuterung der im Rahmen der Arbeit verwendeten Teile der Bauhaus-Suite.

Kapitelinhalt

2.1 Software-Metriken

„A software metric is any type of meassurement which relates to a software system, process or related documentation“ [Som07, Seite 655]

Bei den im Rahmen dieser Arbeit zu implementierenden „Qualitätsindikatoren“ handelt es sich nach obiger Definition um Software-Metriken, die als Grundlage einer weiteren Metrik (dem Code-Quality-Index) dienen, um eine Aussage über die Qualität des untersuchten Software-Systems zu treffen.
Entsprechend soll an dieser Stelle der Begriff der Software-Metrik ein wenig näher betrachtet werden. Die Betrachtung des Begriffs „Qualität“ im Kontext dieser Arbeit folgt in Abschnitt 2.2.
Das Ziel jeglicher Software-Metrik ist laut Fenton und Pfleeger [FP96, Seite 11] Erkenntnisse über Projekte, Produkte, Prozesse und Ressourcen zu gewinnen, um diese verstehen und steuern zu können.
Hierzu ist es zunächst notwendig den Status eines Projekts zu ermitteln und in der Folge kontinuierlich zu überwachen und in Relation zu getroffenen Maßnahmen zu setzen. Auf diese Weise werden Trends erkennbar und ermöglichen Voraussagen über die Entwicklung eines Projekts sowie eine gezielte Steuerung desselben.
Laut Fenton und Pfleeger [FP96] teilen sich Software-Metriken in drei verschiedene Typen auf:

- prozessbezogen
- produktbezogen
- ressourcenbezogen

Für jeden der drei Typen wird weiterhin unterschieden, ob interne oder externe Attribute untersucht werden.
Für die internen produktbezogenen Metriken unterscheiden Fenton und Pfleeger zusätzlich zwischen größen- [FP96, Seiten 244 ff.] und strukturbezogenen [FP96, Seiten 279 ff.] Metriken. Größe bezieht sich dabei nicht nur die Länge – etwa in „Lines of code“ – sondern ebenso auf den Funktionsumfang und die Komplexität sowohl des Codes als auch der zugrundeliegenden Probleme und Algorithmen.

2.2 Qualität

Qualität ist ein eher abstrakter Begriff zu dem viele verschiedene Auffassungen existieren.

„Like Beauty, erveryone may have his own idea of what quality is ...“ (ISO 9000:2000)

Da das in dieser Arbeit zu implementierende Tool eine Aussage über die Qualität einer Software treffen soll, stellt sich vor diesem Hintergrund die Frage, was im Kontext des Code-Quality-Index unter Qualität zu verstehen ist.
Um diese Frage zu beantworten werden die nachfolgenden Abschnitte zunächst einmal die Begriffe der Qualität im Allgemeinen und der Softwarequalität betrachten, um so einen groben Überblick über die verschiedenen Qualitätsverständnisse zu geben.

2.2.1 Qualität allgemein

Das Wort „Qualität“ entstammt dem lateinischen (qualitas) und wird mit „Beschaffenheit“ oder „Eigenschaft“ übersetzt.
Laut Ludewig und Lichter definiert die DIN 55350-11:1995-08 Qualität als :

Qualität: *„Gesamtheit von Eigenschaften und Merkmalen eines Produkts oder einer Tätigkeit, die sich auf die Eignung zur Erfüllung gegebener Erfordernisse beziehen“ [LL07, Seite 63]*

Hierzu wird durch die DIN-Vorschrift ergänzend festgestellt, dass ein Produkt im Sinne dieser Definition nicht nur Waren und Rohstoffe meint sondern auch die Inhalte von Entwürfen. Unter einer Tätigkeit sind ferner nicht nur Diensleistungen sondern auch maschinelle Arbeitsabläufe und Prozesse zu verstehen.
Es kann also zwischen Prozess- und Produktqualität unterschieden werden (vgl. [LL07, Seite 64]). Nach Sommerville [Som07, Seite 666] besteht eine enge Verbindung zwischen Prozess- und Produktqualität, so dass die Qualität eines Produkts in erheblichem Maße von der Qualität des Prozesses abhängt indem es entwickelt wurde.
Ähnlich wie in Abschnitt 2.1 kann auch bei der Qualität zwischen interner und externer unterschieden werden. Als externe Qualität werden hierbei Eigenschaften zusammengefasst, die der Benutzer des Produkts beobachten kann. Bei einem Bildschirm könnte dies zum Beispiel die Farbdarstellung oder die Lebensdauer sein. Die interne Qualität betrifft Eigenschaften, die dem Entwickler/Hersteller bekannt sind, beispielsweise die Vollständigkeit und Korrektheit der Spezifikation.

2.2.2 Softwarequalität

Auch bei der Softwarequalität hängt der Qualitätsbegriff stark vom Betrachter ab. Der Nutzer einer Software wird in aller Regel keinen Anstoß an unleserlichem oder chaotischem Code nehmen, solange die Software die zugedachte Aufgabe zufriedenstellend erledigt. Hier liegt das Interesse eher in der Benutzbarkeit, Ergonomie und Performanz.
Ein mit Wartungsarbeiten betrauter Entwickler hingegen, legt besonderen Wert auf einen lesbaren, vernünftig strukturierten Quellcode. So kann es vorkommen, dass der Nutzer eine Software für qualitativ hochwertig hält, während der Entwickler eine genau gegenteilige Meinung vertritt.

Ebenso wichtig ist in diesem Zusammenhang der Kontext der Software. An ein System, das in einer Echtzeitumgebung für ein einzelnes Projekt über wenige Tage oder Wochen eingesetzt werden soll, werden andere Qualitätsansprüche gestellt als an eine Anwendung die langfristig sämtliche geschäftskritischen Daten auf Abruf verfügbar halten muss.

Die in Abschnitt 2.2.1 gemachten Unterscheidungen zwischen interner und externer Qualität bleiben grundsätzlich auch in der Softwarequalität erhalten. Sommerville ist jedoch der Meinung, dass die Abhängigkeiten zwischen der Prozess- und der Produktqualität in der Softwareentwicklung weniger stark als in der herkömmlichen Produktion sind (vgl. [Som07, Seite 668]). Vielmehr konzentriert sich die Abhängigkeit auf Seiten der Prozessqualität vor allem auf den Design-Prozess, der eine entscheidende Rolle für die Qualität einer Software einnimmt, da hier die Architektur und Module definiert werden.
Neben der (Design-)Prozessqualität gibt es laut Sommerville [Som07, Figure 28.3 Seite 669] noch drei weitere Faktoren, die einen maßgeblichen Einfluß auf die Produktqualität haben: die verwendeten Technologien, die beteiligten Entwickler sowie Zeitplan und Budget.

Die Produktqualität wird im Qualitätenbaum [LL07, Abb. 5-2 Seite 66] von Ludewig und Lichter noch einmal in Wartbarkeit und Brauchbarkeit unterschieden. Dies spiegelt recht exakt die beiden verschiedenen Sichtweisen von Nutzern (Brauchbarkeit) und Entwicklern (Wartbarkeit) auf ein Software-System wieder. Die gleiche Unterteilung wird auch in dem von Simon u.a. verfassten Buch „ Code-Quality-Management“ [SSM06] vorgenommen. Sie sprechen allerdings von funktionaler (entspr. Brauchbarkeit) und technischer (entspr. Wartbarkeit) Qualität.

Die im Qualitätenbaum [LL07, Abb. 5-2 Seite 66] dargestellten Qualitäten werden jeweils durch verschiedene Unter-Qualitätseigenschaften verfeinert, so dass ein hierarchischer Baum entsteht. Man spricht auch von einem hierarchischen Qualitätsmodell. Ebenso wie beim Qualitätsmodell der ISO 9126 (vgl. Abbildung 2.1) zeichnen sich die genannten Qualitätseigenschaften dadurch aus, dass sie viel Raum für subjektive Beurteilung und Gewichtung lassen und sich ihr Erfüllungsgrad nicht direkt über eine Metrik erfassen lässt. Vielmehr müssen Indikatoren für oder wider einzelner Qualitätseigenschaften entwickelt und mittels Metriken erfasst werden, um dann über Aggregation indirekt die Erfüllung der jeweiligen Eigenschaft zu bestimmen. Die Vergleichbarkeit einzelner Beurteilungen hängt dann von den verwendeten

Metrik-Sets ab.

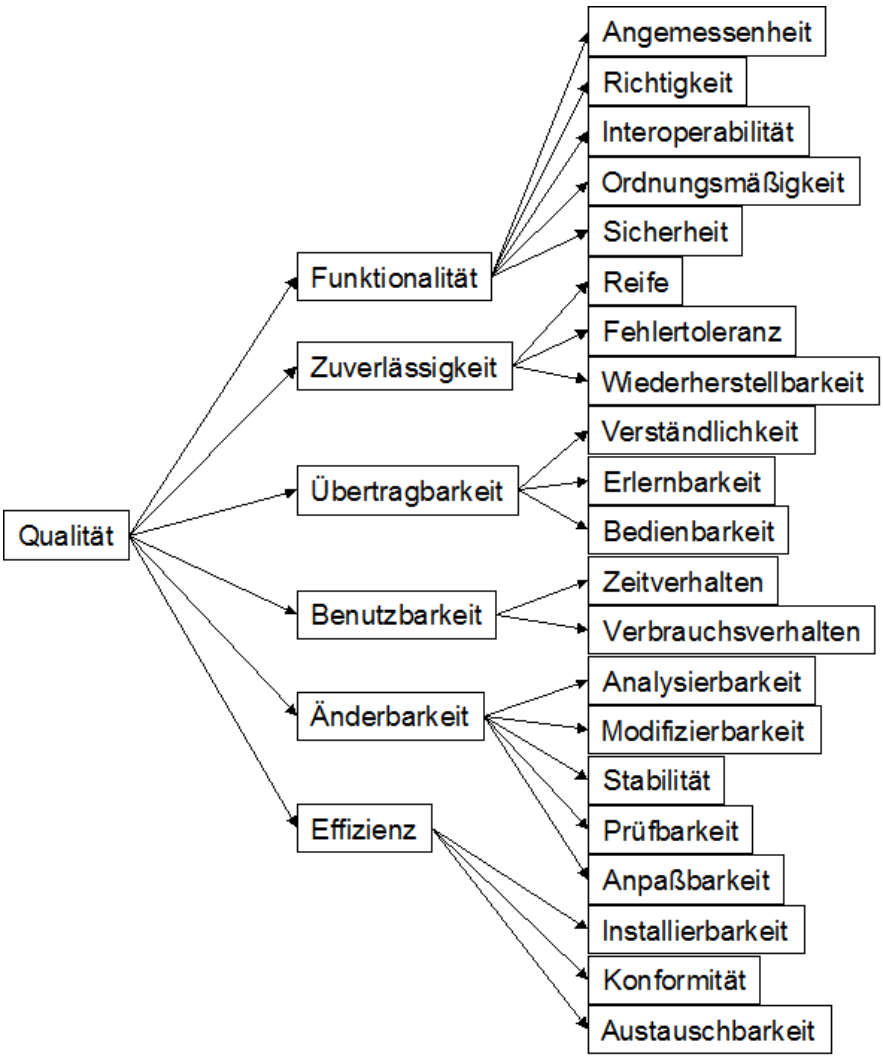


Abbildung 2.1: Qualitätsmodell nach ISO 9126 [SSM06, Abb. 3-2 Seite 38]

Beispiele für ein solches Vorgehen wären die Software-Qualitätsmodelle von Boehm [FP96, Figure 9.1 Seite 339] und McCall [FP96, Figure 9.2 Seite 339].
Da der Qualitätsbegriff in der Regel projektspezifisch ist und die genannten Qualitätsmodelle den Projektkontext nicht oder nur in geringer und abstrakter Form berücksichtigen, scheinen sie in der vorliegenden Form recht unflexibel und sind ohne ein verbindliches Metrik-Set kaum für eine reproduzierbare und vergleichbare Qualitätsbeurteilung geeignet.

Als Folge der Nichtbeachtung des Projektkontexts haben sich die sogenannten „Meta-Qualitätsmodelle“ entwickelt. Diese Modelle geben einen Rahmen vor, innerhalb dessen sich projektspezifische Qualitätsmodelle entwickeln lassen. Beispiel für ein solches Meta-Modell wäre das „Factor-Criteria-Metric-Model“ (FCM-Modell, [MRW77]).
Eine weitere Methode um projektspezifische Qualitätsmodelle zu entwickeln sind Prozesse, wie etwa das 1984 von V.R. Basili und D. Weiss vorgestellte Verfahren „Goal/Question/Metrics“ [BW84].

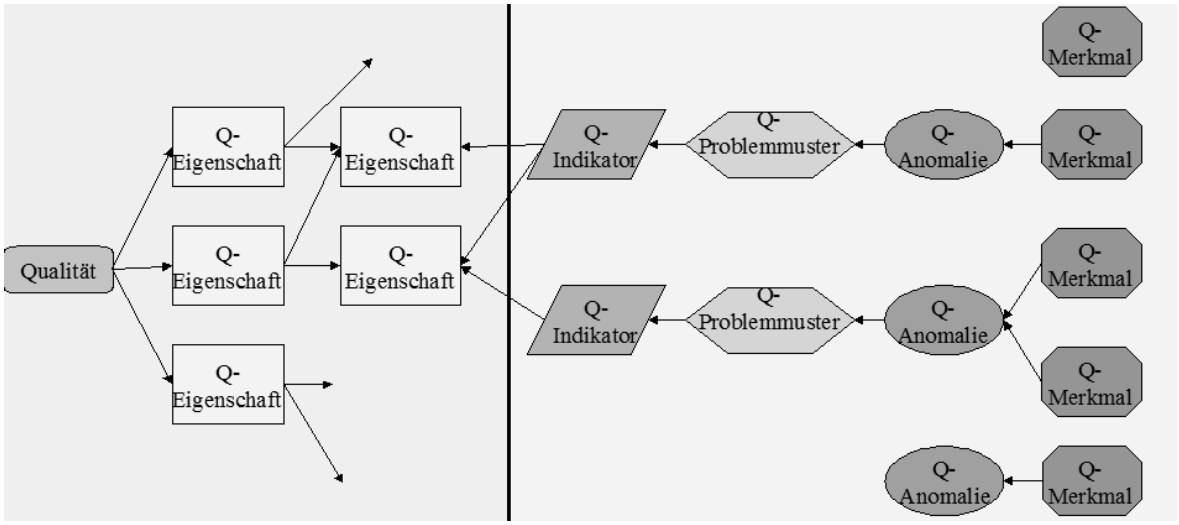


Abbildung 2.2: Konzept des bidirektionalen Qualitätsmodells [SSM06, Abb. 3-6 Seite 55]

Auch bei projektspezifischen Qualitätsmodellen ist eine projektübergreifende Vergleichbarkeit der Qualität naturgemäß nur sehr bedingt möglich.

2.2.3 Qualität im Kontext des Code-Quality-Index

Nachdem die beiden vorangegangenen Abschnitte einen Überblick über die verschiedenen Bedeutungen des Begriffs „Qualität“ vermittelt haben, wird in diesem Abschnitt die Frage betrachtet: „Was ist im Kontext des Code-Quality-Index unter Qualität zu verstehen“.

Wie in Abschnitt 2.2.2 dargelegt eignen sich hierarchische Qualitätsmodelle wie das der ISO 9126 (Abbildung 2.1) nur bedingt für eine objektive und vergleichende Qualitätsbeurteilung. Da jedoch gerade das ISO-Modell in vielen Fällen vertraglich als Qualitätsstandard gefordert wird und es sich durch seine Technikabstinenz auf Managementseite „gut verkaufen“ lässt, findet es als Ausgangspunkt für das Qualitätsmodell des Code-Quality-Index Anwendung.
Durch Anreicherung mit klar definierten Metriken, die sich jeweils mit einer bestimmten Gewichtung auf eine oder mehrere Qualitätseigenschaften auswirken wird ein bidirektionales Qualitätsmodell definiert. Der Begriff der bidirektionalen Qualitätsmodelle wird erstmals durch das QBench-Projekt[1] eingeführt, dem auch der Code-Quality-Index entstammt.
Diese Art von Qualitätsmodell unterscheidet sich von den klassisch hierarchischen vor allem dadurch, dass nicht nur von einer Qualitätseigenschaft ausgehend ermittelt werden kann, durch welche Metrik ihr Erfüllungsgrad bestimmt wird sondern auch in der anderen Richtung zu jeder Metrik bekannt ist mit welcher Ausprägung sie bestimmte Qualitätseigenschaften adressiert. Eine schematische Darstellung des bidirektionalen Qualitätsmodells ist in Abbildung 2.2 zu sehen.

[1] http://www.qbench.de

Für das Qualitätsmodell des CQI werden nicht alle Qualitätseigenschaften der ISO 9126 herangezogen, sondern hauptsächlich diejenigen, die die technische Qualität ausmachen. Konkret sind dies die folgenden Qualitätseigenschaften [SSM06, vgl. Seiten 38 & 39]:

- Änderbarkeit
- Analysierbarkeit
- Modifizierbarkeit
- Stabilität
- Prüfbarkeit
- Effizienz
- Zeiverhalten
- Verbrauchsverhalten
- Übertragbarkeit
- Austauschbarkeit

Zusammenfassend lässt sich sagen: da der Code-Quality-Index ausschließlich den Quellcode, nicht aber den Entwicklungsprozess oder die beteiligten Personen betrachtet, wird exklusiv die technische Produktqualität untersucht, um eine Aussage über die Wartbarkeit treffen zu können.
Der Qualitätsbegriff im Kontext des Code-Quality-Index begrenzt sich also ausschließlich auf die Wartbarkeit eines Software-Systems.

2.3 Maintainability Index

Dieser Abschnitt dient der Vorstellung des so genannten „Maintainability Index“ [WOA97]. Dieser Index soll (wie der Code-Quality-Index) eine Aussage über die Wartbarkeit von Software-Systemen treffen. Entwickelt wurde der Index laut [WOA97] während einer Reihe von, durch Hewlett-Packard finanzierten, Experimenten.
Als Grundlage für diesen Index dienen die folgenden Metriken:

- Die Zeilen-Anzahl (LOC)
- Die Anzahl der Kommentarzeilen (CMT)
- Die erweiterte zyklomatische Komplexität nach Myers (VG2) [Mye77]
- Die Halstead Metriken für Aufwand (E) und Umfang (V) [Hal77]

Aus den oben genannten Metriken lässt sich der Maintainability-Index in zwei Varianten berechnen: mit und ohne Berücksichtigung der Kommentarzeilen.
Die Formel für die Variante ohne Berücksichtigung der Kommentarzeilen basiert auf drei der oben genannten Metriken und lautet wie folgt:

$MI = 171 - 5,2 \times \ln(aveV) - 0,23 \times aveVG2 - 16,2 \times \ln(aveLOC)$
„where: aveV is the average Halstead volume per module, aveVG2 is the average extended cyclomatic complexity per module, and aveLOC is the average lines of code per module.“ [WOA97, Seite 133]

Mit Berücksichtigung der Kommentarzeilen werden alle vier der Metriken verwendet. Diese Variante berechnet sich wie folgt:

$MI = 171 - 5,2 \times \ln(aveV) - 0,23 \times aveVG2 - 16,2 \times \ln(aveLOC) + 50 \times \sin(\sqrt{2,4 \times perCM})$
„where: aveV is the average Halstead volume per module, aveVG2 is the average extended cyclomatic complexity per module, aveLOC is the average lines of code per module, and perCM is the average per cent of lines of comments per module.“ [WOA97, Seite 133]

Die Definition von „Modul“ ist in diesem Kontext von der jeweils betrachteten Programmiersprache abhängig und kann sowohl Methoden und Funktionen als auch Pakete umfassen. Als allgemeingültige Definition geben Welker, Oman und Attkinson [WOA97] an, dass unter dem Begriff „module“ jede benannte lexikalische Programmkomponente zu verstehen ist.
Für das Ergebniss gilt: je höher der Index, desto wartbarer ist das System.

Kuipers und Visser bestätigen in [KV07], dass der vorgestellte Index durchaus erfolgreich in der Praxis eingesetzt wird. Gleichzeitig weisen sie aber auch auf einige deutliche Schwachstellen hin.
Unter anderem führen sie an, dass die Verwendung der durchschnittlichen Komplexität zwangsweise Ungenauigkeiten ins Messverfahren einbringt, die dazu führen könnten, dass ein schlecht wartbares System durch viele „getter“ und „setter“ Methoden einen guten Wartbarkeitsindex bekommt. Ein weiterer Kritikpunkt ist die Verwendung der Anzahl an Kommentarzeilen, weil die bloße Existenz von Kommentarzeilen nichts über ihren Inhalt aussagt. Sie führen an, dass Kommentare häufig lediglich aus auskommentiertem Code bestehen und in diesem Fall keinerlei positiven Beitrag zur Wartbarkeit der Software leisten.
Ein Problem, das in der Praxis auftritt ist vor allem die mangelnde Akzeptanz des Index auf Entwicklerseite, die ihn vor allem wegen der mangelnden Möglichkeiten gezielt durch punktuelle Eingriffe auf den Wert des Index Einfluss zu nehmen ablehnen.
Dennoch stimmen laut Kuipers et al. [KV07] die Ergebnisse des Index in der Mehrzahl der Fälle mit der auf „Bauchgefühl“ beruhenden Qualitätsbeurteilung der beteiligten Entwickler überein.

2.4 Code Quality Index

Dieser Abschnitt dient der Einführung in den Code-Quality-Index (kurz CQI) wie er in „Code-Quality-Management“ [SSM06] von Simon, Seng und Mohaupt vorgestellt wird.
Entstanden ist der CQI im Rahmen des vom Bundesministerium für Bildung und Forschung geförderten Forschungsprojekts QBench[2] mit dem Ziel, der Entwicklung „eines ganzheitlichen Ansatzes zur konstruktions- und evolutionsbegleitenden Sicherung der inneren Qualität von objektorientierter Software, um den Aufwand der Softwareentwicklung und -evolution (und damit Kosten) deutlich senken zu können“ [qbe07].
Die Informationen in diesem Abschnitt sind – wo nicht anders angegeben – aus [SSM06] entnommen. Im Folgenden werden zugunsten der Lesbarkeit nur wörtliche Zitate explizit ausgewiesen.

Es wird ausschließlich die technische Qualität bewertet, um die Frage „Wie problematisch ist es an dieser Software etwas zu ändern“ (sei es zwecks Erweiterung oder BugFixing) leichter abschätzen zu können. Anders ausgedrückt liefert der CQI eine Abschätzung der Frage wie wartbar ein System ist. Hierbei soll das in Abschnitt 2.2.3 vorgestellte bidirektionale Modell sowohl dem Entwickler auf der technischen Seite eine Antwort liefern, die er verstehen und kalkulieren kann als auch dem Entscheider/Manager auf der technikfernen Seite. Diese Abschätzung ist mit den ausgewählten Qualitätsindikatoren[3] explizit ausschließlich für objektorientierte Systeme möglich, da eine Vielzahl der Indikatoren auf Vererbungsbeziehungen und ähnlichen objektorientierten Strukturen basieren.
Die in Abschnitt 2.4.2 vorgestellten Qualitätsindikatoren – und somit auch der eigentliche Code-Quality-Index – betrachten ausschließlich den Quellcode einer Software. Der Entwicklungsprozess oder die verwendeten Ressourcen finden keine Berücksichtigung. Demnach handelt es sich gemäß Abschnitt 2.1 um „interne produktbezogene Metriken“.
Der Wertebereich des Endergebnisses besteht aus fünf Stufen, dem sogenannten Quality-Benchmark-Level und ist damit deutlich an das deutsche System der Hotelzertifizierung angelehnt. Grundsätzlich gilt (wie schon beim Maintainability Index in Abschnitt 2.3): je höher der resultierende Wert desto „leichter„ ist das untersuchte System zu warten. In der Regel liefern Tools die die Indikatoren des CQI berechnen, auch detaillierte Treffer-Listen aus denen konkrete Handlungsempfehlungen zur Verbesserung der Wartbarkeit abgeleitet werden können. Dies erscheint als der größte Vorteil des CQI gegenüber dem Maintainability-Index. Ein Tool, das bei der Berechnnung des CQI die genannten Treffer-Listen generiert ist zum Beispiel das im Rahmen des QBench-Projekts enwickelte Sissy[4].
Simon, Seng und Mohaupt geben für den Code-Quality-Index folgende Definition an:

[2] http://www.qbench.de
[3] näheres zu Qualitätsindikatoren siehe: Abschnitt 2.4.2
[4] http://sissy.fzi.de

Definition: *"Der Code-Quality-Index stellt eine kollektive Kenngröße für eine konkrete, bzgl. der technischen Qualität vorgenommenen Instanziierung eines bidirektionalen Qualitätsmodells für die Programmiersprachen Java und C++ dar. Der Wertebereich des Code-Quality-Index ist hierbei durch 5 verschiedene Quality-Benchmark-Level gegeben, deren Erreichung durch Schwellwerte aus einem Repository, das den aktuellen Stand der Technik repräsentiert, bestimmt wird. Das zugrunde liegende bidirektionale Qualitätsmodell verwendet auf der Merkmalseite lediglich mit Werkzeugen der statischen Analyse automatisch erfassbare Merkmale, so dass der Index für ein Softwaresystem vollautomatisch ermittelt werden kann.„ [SSM06, Def. 4-3 Seite 71]*

Da die Vergabe von Schwellwerten für die einzelnen Qualitätsindikatoren auf empirischen Vergleichen vieler Software-Systeme basiert, sind bislang nur Schwellwerte für die Sprachen C++ und Java definiert. Eine Portierung der Schwellwerte für die Sprachen C#, Visual Basic und Cobol ist jedoch in Arbeit. Um die Vergleichbarkeit zu gewährleisten sind die Schwellwerte jeweils auf 1.000 LOC normiert.
Die Schwellwerte für die Sprachen C++ und Java sind in Kapitel 10 ab Seite 167 in [SSM06] abgedruckt und können dort nachgeschlagen werden.
Die Zuverlässigkeit der Aussagen des CQI ist laut [SSM06, Abb. 4-2 Seite 67] unterhalb der Qualität eines manuellen Code-Reviews einzustufen aber oberhalb des "Bauchgefühls„. Somit liefert der CQI bei vergleichsweise geringem Aufwand, der weit unterhalb eines manuellen Reviews liegt, eine überprüfbare Tendenz der technischen Qualität eines Softwaresystems. Wie zuverlässig diese Tendenz letztlich ist, hängt zu einem guten Teil von der Vorbereitung der Vermessung ab. Besonders die Entfernung nicht mehr verwendeter Systemteile und generierten Codes aus der Vermessung bzw. den Ergebnislisten kann das Ergebnis deutlich verschieben.

2.4.1 Ermittlung des Quality-Benchmark-Levels

An die Erreichung eines Quality-Benchmark-Level (kurz QBL) sind bestimmte Bedingungen geknüpft. Zur Erreichung von QBL 1 genügt es, wenn der Quellcode gemäß seiner Programmiersprache syntaktisch korrekt, also kompilierbar, ist. Es wird explizit nur die "Übersetzbarkeit„ gefordert, da der CQI ausschließlich statische Analysen umfasst und eine Ausführbarkeit geschweige denn eine funktionale Korrektheit weder bescheinigen will noch kann.
Zur Erreichung der höheren QBLs müssen die Schwellwerte bestimmter Qualitätsindikatoren unterboten werden. Die Anforderungssteigerung wird realisiert indem zum Einen immer mehr Indikatoren berücksichtig werden müssen und zum Anderen die Schwellwerte immer niedriger werden. Zu diesem Zweck ist jedem der 52 Indikatoren ein QBL zugeordnet, ab welchem der jeweilige Indikator zu berücksichtigen ist. Zusätzlich wurde ein Schwellwerttunnel definiert, wodurch die Anforderungen an den einzelnen Indikator von QBL zu QBL steigen.
Die Auswahl der Indikatoren, die einem bestimmten QBL zugeordnet werden erfolgt über

den Grad der Auswirkungen die sie auf bestimmte Qualitätseigenschaften haben. Hierzu werden die einzelnen Qualitätseigenschaften priorisiert und auf die QBLs verteilt. Für QBL 2 liegt der Fokus beispielsweise auf den Eigenschaften “Analysierbarkeit„ und “Stabilität„. Entsprechend werden diejenigen Qualitätsindikatoren, die besonders starken Einfluss auf die Analysierbarkeit oder Stabilität haben bereits ab QBL 2 berücksichtigt. Die fokussierten Qualitätseigenschaften der restlichen QBLs sind Abbildung 2.3 zu entnehmen.

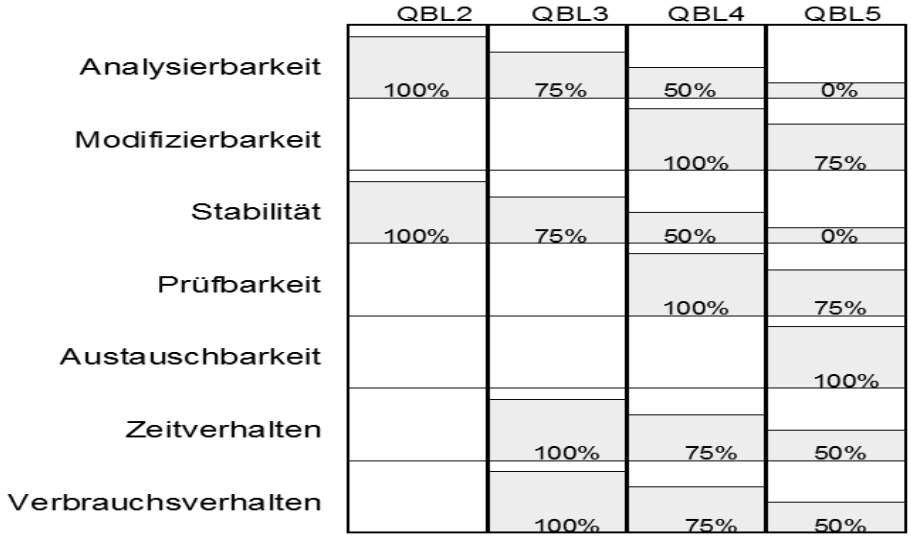


Abbildung 2.3: Fokussierte Qualitätseigenschaften pro QBL [SSM06, Abb. 4-3 Seite 73]

Bei dieser Zuordnung findet noch ein Korrekturfaktor Berücksichtigung, der den Kosten-Nutzen-Faktor in den Code-Quality-Index integriert: Wenn die Behebung von Befunden für einen Qualitätsindikator sehr aufwendig ist, der Nutzen jedoch eher gering, wird er erst ab dem nächsten oder übernächsten QBL berücksichtigt. In der anderen Richtung kann ein Qualitätsindikator, wenn die Behebung von Befunden sehr leicht und der Nutzen daraus sehr hoch ist, bereits in einem niedrigeren QBL berücksichtigt werden. Die genauen Korrekturfaktoren sind in Abb. 4-4 auf Seite 75 in [SSM06] nachzulesen.

Schwellwerttunnel

Die Schwellwerte ergeben sich aus einem Repository, in dem die mittels LOC normierte Verletzungsanzahl für jeden Qualitätsindikator aus der Vermessung von mehr als 120 Industrieprojekten abgelegt ist.

QBL	Schwellwert
2	oberes Quantil
3	Median
4	unteres Quantil
5	Minimum

Tabelle 2.1: Schwellwerte

Aus den abgelegten Daten werden die Werte Maximum, oberes Quantil, Median, unteres Quantil und Minimum bestimmt.
Für QBL 2 wird der Wert des oberen Quantils als Schwellwert verwendet. Das untersuchte System muss also bzgl. jedes Qualitätsindikators "besser„ sein als die 25% der "schlechtesten„ Systeme.
Für die einzelnen QBL gelten die in Tabelle 2.1 angegebenen Schwellwerte, so dass sich der in Abbildung 2.4 abgebildete Schwellwerttunnel ergibt.

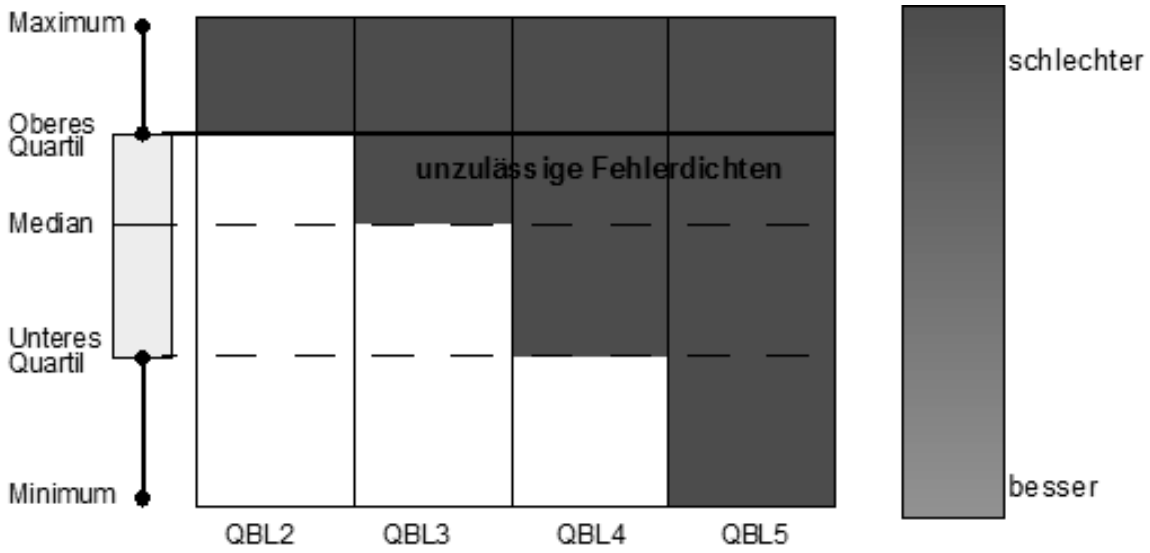


Abbildung 2.4: Schwellwerttunnel [SSM06, Abb 4-5 Seite 77]

2.4.2 Die Qualitätsindikatoren

Dieser Abschnitt stellt die 52 Qualitätsindikatoren kurz vor und nennt jeweils den Quality-Benchmark-Level, ab dem der Indikator zum Tragen kommt.
Auf Begründungen für die Auswahl und Relevanz der einzelnen Indikatoren sowie die Angabe der Grenzwerte wird verzichtet. Diesbezüglich ist bei Bedarf Kapitel 10 ab Seite 167 in [SSM06] zu konsultieren. Die Grenzwerte sind zusätzlich in Anhang A tabellarisch aufgelistet.

2.4.2.1 allgemeine Parameter

„Allgemeine Parameter“ liegen vor, wenn ein Parameter einer Methode innerhalb derselben in einen spezielleren Typ gecastet wird. Ausnahmsweise nicht gezählt wird die Methode, wenn sie in einer Oberklasse bzw. einem Interface deklariert wurde und nun lediglich implementiert oder überschrieben wird.
Jeder Parameter auf den obige Beschreibung zutrifft stellt eine seperate Verletzung dar.

Dieser Indikator wird ab QBL 3 berücksichtigt.

2.4.2.2 Attributüberdeckung

Ein nichtstatisches Attribut einer Oberklasse wird in mindestens einer direkten oder indirekten Unterklasse namensgleich erneut deklariert. Das Oberklassenattribut wird überdeckt.
Jedes derartige Oberklassenattribut stellt eine Verletzung dar.

Dieser Indikator wird ab QBL 3 berücksichtigt.

2.4.2.3 ausgeschlagenes Erbe (Implementierung)

Eine implementierte Methode wird in mehr als 50% „der direkten Unterklassen nichtleer überschrieben, ohne dabei aufgerufen zu werden“[SSM06, Seite 175].
Als Verletzung ist jeweils die Oberklassenmethode zu zählen.

Dieser Indikator wird ab QBL 4 berücksichtigt.

2.4.2.4 ausgeschlagenes Erbe (Schnittstelle)

Eine geerbte Schnittstelle wird ausgeschlagen, wenn eine Unterklasse eine Methode mit geringerer Sichtbarkeit (nur C++) oder leer überschreibt oder implementiert. Sofern die Methode einen Rückgabewert liefert gilt sie auch mit einem evtl. nötigen `return null` als leer.
Für die Reduzierung der Sichtbarkeit gilt:

```
public > protected > private
```

Gezählt werden hierbei Methodenpaare aus Ober- und Unterklassenmethode.

Dieser Indikator wird ab QBL 3 berücksichtigt.

2.4.2.5 Datenkapselaufbruch

Nichtkonstante Attribute werden durch eine nichtgeschachtelte Klasse öffentlich deklariert und von mindestens einer anderen Klasse lesend oder schreibend verwendet.
Jedes derartige Attribut wird als eine Verletzung betrachtet.

Dieser Indikator wird ab QBL 2 berücksichtigt.

2.4.2.6 duplizierter Code

Unter dupliziertem Code sind im Kontex des Code-Quality-Index mindestens 40 aufeinander folgende Codezeilen (inkl. Leer- und Kommentarzeilen) zu verstehen, die mehrmals in identischer Form existieren.
Zu zählen ist die Anzahl der betroffenen Zeilen.

Dieser Indikator wird ab QBL 3 berücksichtigt.

2.4.2.7 falsche Namenslänge

Zu zählen sind alle Artefaktnamen der folgenden Artefakte, die kürzer als zwei oder länger als 50 Zeichen sind:

- Pakete
- Dateien (ohne Erweiterung)
- Klassen und Interfaces
- Methoden
- Attribute und Konstanten

Dieser Indikator wird ab QBL 3 berücksichtigt.

2.4.2.8 Generationskonflikt

Mehr als 50% der von einer direkten Oberklasse geerbten und dort implementierten Methoden werden durch eine Unterklasse nichtleer überschrieben. Die Oberklassenmethode wird hierbei nicht aufgerufen. Sofern die Methode einen Rückgabewert liefert gilt sie auch mit einem evtl. nötigen `return null` als leer.
Jedes Klassenpaar aus Ober- und Unterklasse wird als eine Verletzung gezählt.

Dieser Indikator wird ab QBL 4 berücksichtigt.

2.4.2.9 Gottdatei

Als Gottdatei ist jede Datei zu zählen, die mehr als 2000 Brutto Lines of Code enthält.

Dieser Indikator wird ab QBL 4 berücksichtigt.

2.4.2.10 Gottklasse (Attribut)

Eine Klasse, die mehr als 50 nichtkonstante Attribute deklariert wird als „Gottklasse (Attribut)„ gezählt.

Dieser Indikator wird ab QBL 3 berücksichtigt.

2.4.2.11 Gottklasse (Methode)

Eine Klasse, die mehr als 50 Methoden deklariert wird analog zu Abschnitt 2.4.2.10 als “Gottklasse (Methode)„ gezählt. Bei der Zählung der Methoden werden nur solche berücksichtigt, die mindestens die gleiche Sichtbarkeit wie ihre umgebende Klasse haben. Für die Sichtbarkeit gilt:

```
public > protected > package-private > private
```

Dieser Indikator wird ab QBL 3 berücksichtigt.

2.4.2.12 Gottmethode

Analog zu Abschnitt 2.4.2.9 gilt als “Gottmethode„ jede Methode mit einer Länge von mehr als 200 Brutto Lines of Code. Dies schließt die Zeilen mit der öffnenden bzw. schließenden Klammer mit ein.

Dieser Indikator wird ab QBL 4 berücksichtigt.

2.4.2.13 Gottpaket

Ein Paket bzw. Verzeichnis, das mehr als 50 öffentliche, nichtgeschachtelte Klassen oder Interfaces enthält gilt als “Gottpaket„.

Dieser Indikator wird ab QBL 3 berücksichtigt.

2.4.2.14 halbherzige Operationen

Dieser Indikator ist für Java und C++ unterschiedlich definiert. Es werden in beiden Fällen alle Verletzungen seperat gezählt. (Auch mehrere innerhalb derselben Klasse)

Java

Es ist nur eine der beiden Methode `equals()` und `hashCode()` implementiert

C++

Es wird jeweils nur eine der beiden Funktionen implementiert bzw. überschrieben:

- `operator++(void)` und `operator++(int)`
- `operator--(void)` und `operator--(int)`
- Copy-Konstruktor und `operator=`

oder es existiert ein nichtvirtueller Konstruktor, obwohl virtuelle Methoden deklariert sind.

Dieser Indikator wird ab QBL 2 berücksichtigt.

2.4.2.15 heimliche Verwandtschaft

Es sind alle öffentlichen Klassen und Interfaces mit mindestens einer selbstdeklarierten öffentlichen Methode und mindestens einer Unterklasse bzw. Implementierung zu zählen, deren öffentliche Methoden nicht von außerhalb der Vererbungs- bzw. Implementierungshierarchie aufgerufen werden.

Dieser Indikator wird ab QBL 4 berücksichtigt.

2.4.2.16 Identitätsspaltung

Gezählt werden Gruppierungen von öffentlichen Klassen und Interfaces mit gleichem Namen. Der Namensvergleich wird unabhängig von Groß- und Kleinschreibung vorgenommen.

Dieser Indikator wird ab QBL 2 berücksichtigt.

2.4.2.17 Importchaos

Dieser Indikator ist für java und C++ unterschiedlich definiert. “Jede Import/Include-Anweisung kann maximal eine Probleminstanz darstellen„[SSM06, Seite 219]

Java

Eine Verletzung dieses Indikators liegt vor wenn:

- Ganze Pakete als On-Demand-Import (* am Ende des Imports) importiert werden
- Ein und dieselbe Klasse mehrfach importiert wird
- Eine Klasse aus *java.lang* importiert wird
- Eine Klasse aus dem eigenen Paket importiert wird

C++

Eine Verletzung dieses Indikators für C++ liegt vor wenn dieselbe Datei mehrfach inkludiert wird.

Dieser Indikator wird ab QBL 3 berücksichtigt.

2.4.2.18 Importlüge

Dieser Indikator behandelt überflüssige weil ungenutzte Importe. Als Verletzung werden gezählt:

- Das Importieren / Inkludieren von Klassen bzw. Dateien, die nicht verwendet werden
- Das Importieren von Paketen, wenn kein Bestandteil des Pakets verwendet wird

Dieser Indikator wird ab QBL 3 berücksichtigt.

2.4.2.19 informelle Dokumentation

Vor jeder Methodendeklaration sollte sich ein durch `/*` eingeleiteter Kommentar befinden. Fehlt dieser Kommentar gilt dies als Verletzung.

Dieser Indikator wird ab QBL 4 berücksichtigt.

2.4.2.20 Interface-Bypass

Eine öffentliche Methode die durch eine Abstraktion (Interface oder abstrakte Klasse) deklariert wird, wird nicht über diese Abstraktion sondern über eine konkrete Implementierung aufgerufen. Das Interface wird also “umgangen„.
Als Verletzung gezählt werden Methodenpaare (aufrufende und aufgerufene Methode).

Dieser Indikator wird ab QBL 3 berücksichtigt.

2.4.2.21 Klässchen

Eine öffentliche Klasse mit weniger als drei eigene Methoden und weniger als drei eigenen Attributen (inkl. Konstanten) gilt als “Klässchen„. Nicht betrachtet werden Interfaces und geschachtelte Klassen.
Jede Klasse auf die diese Bedingungen zutreffen wird als eine Verletzung gezählt.

Dieser Indikator wird ab QBL 4 berücksichtigt.

2.4.2.22 Klasseninzest

Gezählt werden Paare aus Ober- und Unterklasse bzw. Interface und Unter-Interface, bei denen die Oberklasse bzw. das Interface Artefakte (Methoden [inkl. Struktoren] oder Attribute/Konstanten) ihrer (in)direkten Unterklassen bzw. Unter-Interfaces referenziert.

Dieser Indikator wird ab QBL 2 berücksichtigt.

2.4.2.23 Konstantenregen

Gezählt werden Gruppen global sichtbarer Konstanten (*public static final* bzw. *public static const*) mit gleichem Namen. Der Datentyp der Konstanten wird hierbei nicht berücksichtigt.

Dieser Indikator wird ab QBL 5 berücksichtigt.

2.4.2.24 Labyrinthmethode

Methoden mit einem zyklomatischen Wert (vgl. [McC76]) > 10 gelten als Labyrinthmethode.
Jede entsprechende Methode wird als eine Verletzung gezählt.

Dieser Indikator wird ab QBL 2 berücksichtigt.

2.4.2.25 lange Parameterliste

Gezählt werden alle Methoden mit mehr als 7 Parametern, die keine Methode einer externen Bibliothek überschreiben oder implementieren.

Dieser Indikator wird ab QBL 2 berücksichtigt.

2.4.2.26 maskierende Datei

Dateien deren Name (ohne Erweiterung) sich nicht im Namen einer in ihr enthaltenen nicht geschachtelten Klasse wiederfindet, gilt als “maskierende Datei„. Betrachtet werden dabei nur diejenigen Klassen mit der höchsten Sichtbarkeit.

Die Groß- und Kleinschreibung der Namen wird nicht beachtet.
Jede entsprechende Datei wird als eine Verletzung gezählt.

Dieser Indikator wird ab QBL 3 berücksichtigt.

2.4.2.27 nachlässige Kommentierung

Dieser Indikator ist eine systemweite Metrik. Eine Verletzungsanzahl wird hier nicht ermittelt. Stattdessen ist ein möglichst kleiner Wert als Ergebnis der folgenden Formel erstrebenswert:

$$abs(BLOC - 2 \times CLOC)$$

Dabei steht "BLOC„ für "Brutto Lines of Code„ und "CLOC„ für die Anzahl der Kommentarzeilen im gesamten System. Als Kommentarzeile gilt in diesem Zusammenhang nur eine Zeile die ausschließlich Kommentar enthält.

Dieser Indikator wird ab QBL 2 berücksichtigt.

2.4.2.28 Namensfehler

Gezählt wird jedes Artefakt, das gegen die Namenskonventionen verstößt. Die verwendete Namenskonvention kann (und sollte) dem jeweiligen Projekt entsprechen, dem dieser Indikator insofern angepasst werden kann. Als Ausgangspunkt werden auf Seite 249 in [SSM06] die folgenden regulären Ausdrücke für verscheidene Artefakttypen genannt:

Paketname `[a-zA-z][a-zA-z0-9_]*`

Klassen- und Interfacenamen `[A-Z][a-zA-Z0-9_]*`

Dateinamen `[a-zA-Z][a-zA-Z0-9_]*\.[a-zA-Z]+`

Methodennamen in Java `[a-z][a-zA-Z0-9_]*`

Methodennamen in C++ `[A-Z][a-zA-Z0-9_]*`

Konstantennamen `[A-Z][A-Z0-9_]*`

Attributnamen `[a-z][a-zA-Z0-9_]*`

Dieser Indikator wird ab QBL 2 berücksichtigt.

2.4.2.29 Objektplacebo (Attribut)

Jeder Zugriff auf ein statisches Attribut über eine Objektinstanz anstatt des Objekttyps wird als Verletzung gezählt.

Dieser Indikator wird ab QBL 2 berücksichtigt.

2.4.2.30 Objektplacebo (Methode)

Analog zu Abschnitt 2.4.2.29 wird jeder Aufruf einer statischen Methode über eine Objektinstanz anstatt des Objekttyps als Verletzung gezählt.

Dieser Indikator wird ab QBL 2 berücksichtigt.

2.4.2.31 Paketchen

Gezählt werden Pakete, die weniger als 3 öffentliche Klassen und Interfaces anbieten.

Dieser Indikator wird ab QBL 4 berücksichtigt.

2.4.2.32 Pakethierarchieaufbruch

Gezählt werden Paare von Oberklasse und direkter Unterklasse, für die gilt, dass die Oberklasse in einem Unterpaket (auch indirekt) des Pakets der Unterklasse deklariert ist. Als Oberklasse werden in diesem Fall auch Interfaces eingestuft. Zusätzlich werden bei der Paarbildung nicht nur Vererbungsbeziehungen sondern auch Implements-Beziehungen berücksichtigt.

Dieser Indikator wird ab QBL 3 berücksichtigt.

2.4.2.33 Polymorphieplacebo

Gezählt werden Methodenpaare für die gilt: eine statische Methode wird in einer Unterklasse durch eine Methode mit gleicher Signatur überdeckt. Voraussetzung hierfür ist, dass die Methode der Oberklasse in der Unterklasse sichbar ist.

Dieser Indikator wird ab QBL 2 berücksichtigt.

2.4.2.34 potenzielle Privatsphäre (Attribut)

Ein Attribut hat eine größere Sichtbarkeit als notwendig wäre, um das System zu übersetzen.
Jedes Attribut dessen Sichtbarkeit reduziert werden kann, wird als eine Verletzung gezählt.

Dieser Indikator wird ab QBL 3 berücksichtigt.

2.4.2.35 potenzielle Privatsphäre (Methode)

Eine als `protected` deklarierte Methode kann in ihrer Sichtbarkeit reduziert werden, wenn sie in keiner Unterklasse benutzt oder überschrieben wird und selbst

keine Methode überschreibt.
Jede entsprechende Methode wird als eine Verletzung gezählt.

Dieser Indikator wird ab QBL 3 berücksichtigt.

2.4.2.36 pränatale Kommunikation

Ein Konstruktor ruft mindestens eine in seiner Klasse sichtbare virtuelle Methode direkt (nicht über weitere Objekte) auf.
Jeder derartige Konstruktor wird als eine Verletzung gezählt.

Dieser Indikator wird ab QBL 2 berücksichtigt.

2.4.2.37 Risikocode

Dieser Indikator befasst sich mit falsch eingesetzten Techniken der defensiven Programmierung.
Als Verletzung werden gezählt:

- jeder leere Exception-Handler
- jede Switch-Anweisung ohne `default`-Zweig
- jede Case-Anweisung ohne `break`

Dieser Indikator wird ab QBL 2 berücksichtigt.

2.4.2.38 signaturähnliche Klassen

Gezählt werden Klassenpaare, die sich in ihrer Signatur ähnlich sind, obwohl sie nicht in einer Vererbungsbeziehung zueinander stehen und auch keine gemeinsame Oberklasse haben. Eine Signaturähnlichkeit im Rahmen dieses Indikators liegt vor, wenn die Klassen mehr als 50% oder mindestens 10 identische nicht geerbte Methodensignaturen haben. Als geerbt gilt in diesem Zusammenhang auch eine Signatur, die bereits in einem Interface oder einer Obklasse existiert.

Dieser Indikator wird ab QBL 4 berücksichtigt.

2.4.2.39 simulierte Polymorphie

Simulierte Polymorphie liegt vor, wenn ein Objekt innerhalb einer Methode auf mehrere Typen hin überprüft wird. Die Überprüfung wird an dieser Stelle auf die Funktion instanceof (Java) bzw. typeid(c++) beschränkt.
Als Verletzung wird die fragliche Methode gezählt.

Dieser Indikator wird ab QBL 3 berücksichtigt.

2.4.2.40 späte Abstraktion

Gezählt werden abstrakte Klassen, die eine nichtabstrakte Oberklasse haben.

Dieser Indikator wird ab QBL 5 berücksichtigt.

2.4.2.41 tote Attribute

Alle privaten Attribute und Konstanten gelten als tot, wenn sie von ihren umschließenden Klassen nicht verwendet werden.
Jedes tote Attribut bzw. Konstante wird als eine Verletzung gezählt.

Dieser Indikator wird ab QBL 3 berücksichtigt.

2.4.2.42 tote Implementierung

Anweisungen, die im Quellcode nach einer return-Anweisung oder nach dem Auslösen einer Exception stehen, werden niemals ausgeführt. Sie erfüllen damit den Tatbestand “tote Implementierung„.
Gezählt werden alle return- und throw-Anweisungen, auf die nicht ausführbare Anweisungen folgen.

Dieser Indikator wird ab QBL 2 berücksichtigt.

2.4.2.43 tote Methoden

Eine private Methode gilt als tot, wenn sie von ihrer Klasse nicht aufgerufen wird.
Jede tote Methode wird als eine Verletzung gezählt.

Dieser Indikator wird ab QBL 2 berücksichtigt.

2.4.2.44 überbuchte Datei

Eine Datei die mehrere nichtgeschachtelte Klassen mit der Sichtbarkeit `public` oder `default` enthält gilt als “überbucht„ und stellt somit eine Verletzung dar.

Dieser Indikator wird ab QBL 3 berücksichtigt.

2.4.2.45 unfertiger Code

Der Indikator “unfertiger Code„ wird durch jeden Kommentar verletzt, der eines der folgenden Schlüsselwörter enthält:

- todo
- hack
- fixme

Die Groß- und Kleinschreibung ist hierbei irrelevant.

Dieser Indikator wird ab QBL 2 berücksichtigt.

2.4.2.46 unvollständige Vererbung (Attribut)

Eine "unvollständige Vererbung„ bzgl. Attribute liegt vor, wenn ein nichtstatisches Attribut namensgleich in mindestens 10 oder mehr als 50% (min. 2) der direkten Unterklassen existiert.
Jedes betroffene Oberklassenattribut wird als eine Verletzung gezählt.

Dieser Indikator wird ab QBL 3 berücksichtigt.

2.4.2.47 unvollständige Vererbung (Methode)

Analog zu Abschnitt 2.4.2.46 wird hier jede Methode als Verletzung gezählt, deren Signatur in mehr als 50% (min. 2) oder mindestens 10 der direkten Unterklassen existiert. Ausgenommen sind Methodensignaturen, die in weiteren Oberklassen der Unterklassen existieren.
Für Java werden in diesem Zusammenhang Interfaces als Klassen behandelt.

Dieser Indikator wird ab QBL 3 berücksichtigt.

2.4.2.48 verbotene Dateiliebe

Der Tatbestand der verbotenen Dateiliebe wird von allen (logischen) Dateipaaren erfüllt, die sich wechselseitig über ihre Inhalte referenzieren.

Dieser Indikator wird ab QBL 3 berücksichtigt.

2.4.2.49 verbotene Klassenliebe

Analog zu Abschnitt 2.4.2.48 stellt eine direkte zyklische Abhängigkeit zwischen zwei Klassen bzw. Interfaces eine Verletzung dieses Indikators dar. Nicht gezählt werden in diesem Zusammenhang Klassen bzw. Interfaces, die in einer Vererbungsbeziehung zueinander stehen oder verschachtelt sind.

Dieser Indikator wird ab QBL 3 berücksichtigt.

2.4.2.50 verbotene Methodenliebe

Methoden, die sich wechselseitig direkt aufrufen erfüllen den Tatbestand der verbotenen Methodenliebe.
Jedes derartige Methodenpaar wird als Verletzung gezählt.

Dieser Indikator wird ab QBL 2 berücksichtigt.

2.4.2.51 verbotene Paketliebe

Jedes Paketpaar, zwischen dem eine direkte zyklische Abhängigkeit besteht, wird als eine Verletzung gezählt.

Dieser Indikator wird ab QBL 4 berücksichtigt.

2.4.2.52 versteckte Konstantheit

Durch "versteckte Konstantheit„ zeichnen sich statische Attribute aus, die ausschließlich lesend verwendet werden ohne als Konstante deklariert zu sein.
Jedes derartige Attribute wird als eine Verletzung gezählt.

Dieser Indikator wird ab QBL 2 berücksichtigt.

2.5 Bauhaus

Das zu implementierende Werkzeug zur Bestimmung des Code-Quality-Index soll auf Basis der durch die Bauhaus-Suite bereitgestellten Infrastruktur entwickelt werden. In diesem Abschnitt sollen deshalb die – für die Arbeit relevanten – Teile der Bauhaus-Suite kurz vorgestellt werden. Zusätzlich werden ggfs. Quellen genannt, die tiefergehende Informationen zu einzelnen Themen bieten.
Zunächst werden die beiden Programm-Repräsentationsformen IML und RFG vorgestellt. Da aus diesen die Informationen für die Verletzungszählung und Auswertung der meisten Qualitätsindikatoren extrahiert werden, sind sie von zentraler Bedeutung. Im Anschluss wird das Scripting-Interface, auf dem die gesamte Implementierung beruht, vorgestellt.
Den Abschluß des Kapitels bildet die Vorstellung der bereits in Bauhaus integrierten Metriken und der Klonerkennung. Einige Analysen von Qualitätsindikatoren greifen auf einen Teil des Funktionsumfangs dieser beiden Features zurück.

2.5.1 IML

Die IML (***I**ntermediate **M**odelling **L**anguage*) ist ein abstrakter Syntaxgraph, der um Informationen zum Kontroll- und Datenfluss erweitert wurde. Die durch die IML verwirklichte feingranulare Programmrepräsentation ist sehr quellcodenah und kann in weiten Teilen in konkreten Quellcode zurückübertragen oder als solcher interpretiert werden. Dies beweist beispielsweise der IML-Interpreter von Phillippe Schober [Sch07]. Dennoch verfügt die IML über eine hinreichend starke Abstraktion vom eigentlichen Quellcode, so dass sie weitgehend sprachunabhängig ist. Dies ermöglicht in hohem Maße die Wiederverwendung bereits implementierter Analysen. Die weitgehende Sprachunabhängigkeit der IML wird stellenweise durch sprachspezifische Besonderheiten – zum Beispiel das Paketkonzept von *Java*, welches in *C++* so nicht existiert – etwas gestört, so dass es für bestimmte Analysen notwendig sein kann diese sprachspezifisch zu gestalten.
Insbesondere die von *jafe* generierte *Java*-IML befindet sich derzeit noch in der Entwicklung und stellt Sachverhalte teilweise grundlegend anders dar, als dies für *C++* der Fall ist. Diesem Umstand wurde in der Aufgabenstellung durch die Einschränkung auf die *C++* basierte Darstellung Rechnung getragen.
Durch die Fülle an Informationen, die die IML bietet, sind IML basierte Analysen in der Regel relativ kompliziert zu implementieren. Weiterhin bringen sie unter Umständen höhere Laufzeiten (als RFG basierte) mit sich, weil sehr viele Informationen verarbeitet werden müssen, die eigentlich gar nicht benötigt werden. Zusätzlich gestaltet sich das Debuging auf IML-Basis schwieriger, weil kein Visualisierungstool existiert.
Im Rahmen dieser Arbeit bietet es sich deshalb an, wann immer möglich, auf den grobgranulareren RFG (siehe auch Abschnitt 2.5.2) zurück zu greifen.
Detailliertere Informationen zu Aufbau und Struktur der IML auf Basis von *C++* und *Java* können den folgenden Quellen entnommen werden: [KS03] (C++) und [Kna02] (Java).

2.5.2 RFG

Der RFG (***R**essource **F**low **G**raph*) ist eine grobgranulare Zwischendarstellung, die noch wesentlich weiter vom Quellcode abstrahiert als die IML. Es handelt sich im Wesentlichen um einen hierarchischen Graphen, der die Bestandteile eines Software-Systems und ihre Beziehungen zueinander beinhaltet. Hierbei steigt er nicht weiter als auf die Ebene von Methoden und Klassenattributen ab.
Ein großer Vorteil des RFG ist die Möglichkeit, statt des gesamten Graphen nur einen Teilgraph – eine sogenannte Sicht (*View*) – zu betrachten und damit die zu verarbeitende Informationsmenge drastisch zu reduzieren. Auf diese Weise kann die Menge der betrachteten Informationen bei der Berechnung des Qualitätsindikators „verbotene Methodenliebe“ zum Beispiel auf die im System enthaltenen Methoden und Methodenaufrufe beschränkt werden.
Während der RFG für C++ ausschließlich aus der zuvor generierten IML erstellt werden kann, ist es im Fall von Java alternativ möglich, den RFG direkt aus dem Bytecode zu gewinnen.
Im Vergleich zur IML gestaltet sich das Debugging für den RFG verhältnismäßig unkompliziert, da mit dem Visualisierungstool *Gravis* ein mächtiges Werkeug zur Vergügung steht, mit dem ermittelt werden kann, wie ein bestimmter Sachverhalt im RFG modelliert ist.
Auch der RFG erhebt den Anspruch auf weitgehende Sprachunabhängigkeit. Wie schon in der IML müssen aber auch hier sprachspezifische Besonderheiten berücksichtigt werden, die zu deutlichen Unterschieden zwischen Java- und C++-RFGs führen, wie in Abbildung 2.5 und Abbildung 2.6 unschwer zu erkennen ist.

Weitergehende Informationen zum Aufbau des RFG und der Bedeutung einzelner Knoten und Kanten können dem Axivion Language Guide [AXI07a, Seiten 145ff.] entnommen werden.

2.5.3 Das Scripting-Interface

Für beide Programmrepräsentationsformen existiert eine Skripting-Schnittstelle auf Basis von Python. Diese Schnittstelle ermöglicht es, Analysen sowohl für IML als auch für RFG in Python zu implementieren und somit nachträglich – ohne Zugriff auf den Quellcode der Bauhaus-Suite zu haben – hinzuzufügen und bei Bedarf auszuführen. Insbesondere ergibt sich daraus die Möglichkeit, dem jeweiligen Projektkontext angepasste Analysen skriptgesteuert in den Buildprozess zu integrieren.
Während das RFG-Skripting auf Wunsch jedem Kunden zur Verfügung steht, ist das IML-Skripting ausschließlich für akademische Zwecke und im kommerziellen Umfeld exklusiv durch SQS nutzbar.
Eine umfassende Dokumentation des RFG-Scriptings findet sich im Axivion Guide [AXI07a, Seiten 285 ff.], der mit jedem Bauhaus-Release mitgeliefert wird. Die Dokumentation der Skripting-Schnittstelle für IML findet sich im IML Scripting Guide [AXI07b] und ist – wie auch das IML-Scripting – nur eingeschränkt verfügbar.

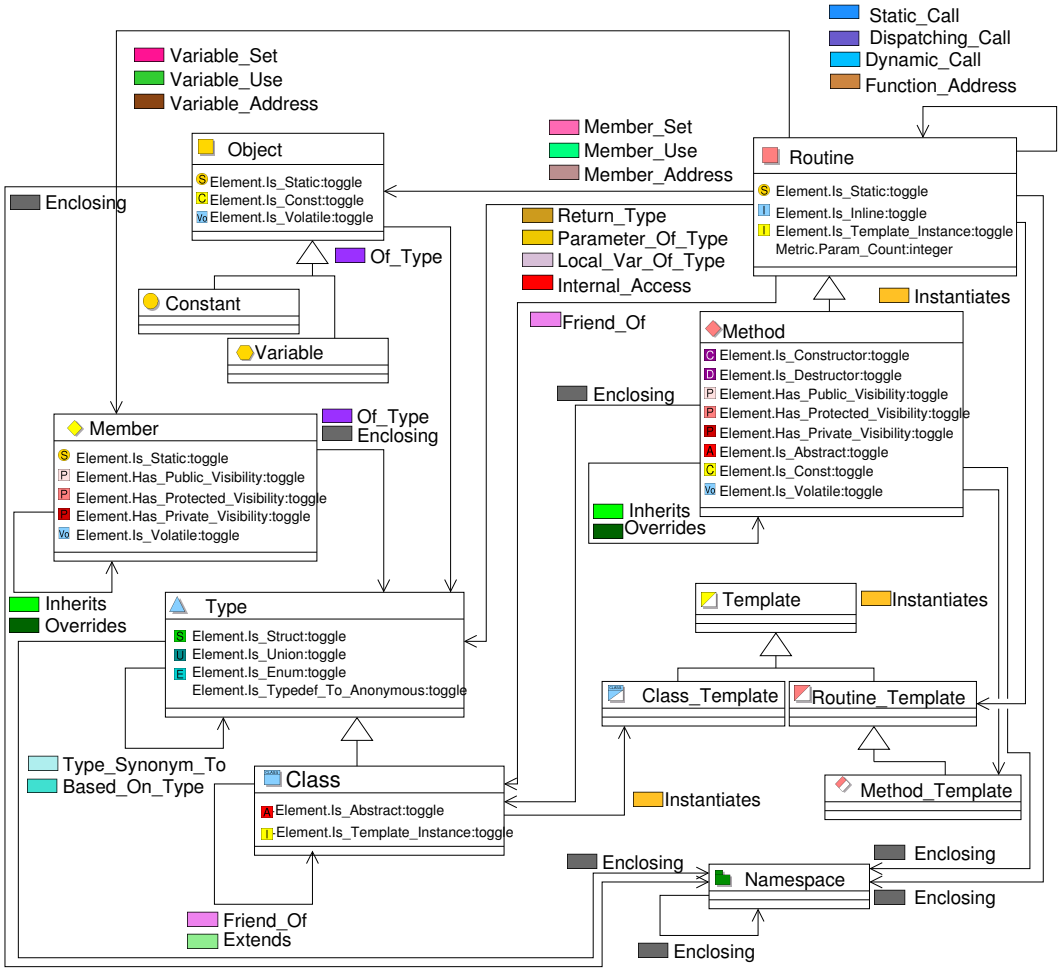


Abbildung 2.5: RFG-Schema für C++ [AXI07a, Seite 161]

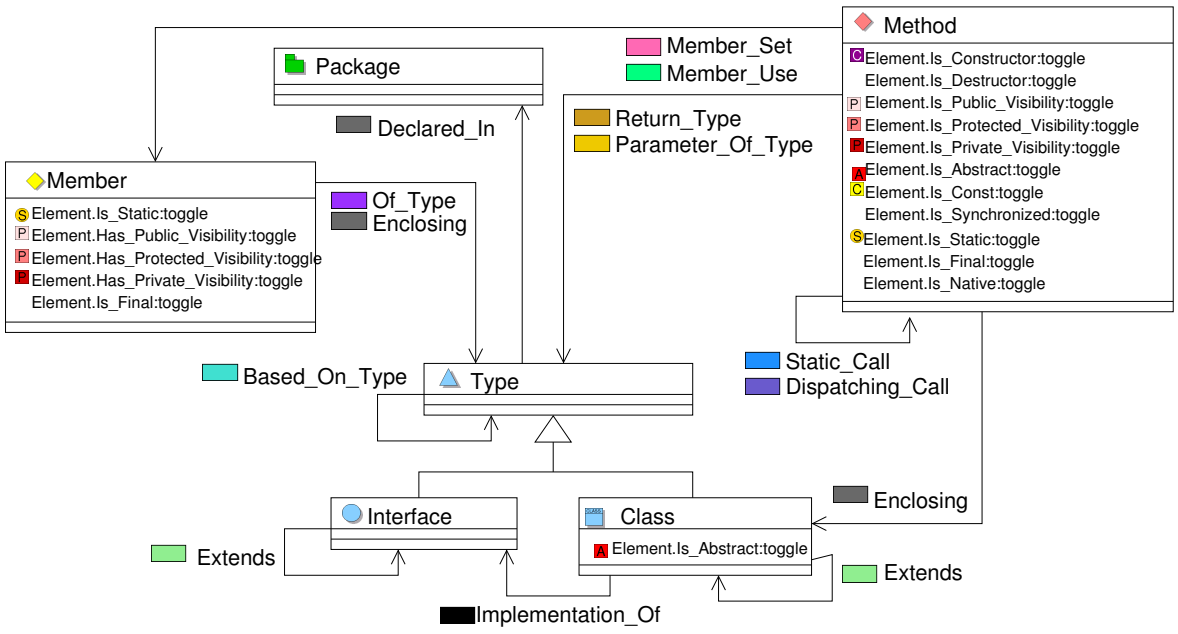


Abbildung 2.6: RFG-Schema für Java [AXI07a, Seite 167]

2.5.4 In Bauhaus integrierte Metriken

Im Lieferumfang der Bauhaus-Suite sind bereits verschiedene Metriken enthalten. Da die mitgelieferten Metriken ständig weiterentwickelt und ergänzt werden, ist es nahezu unmöglich sie vollständig aufzuzählen. Im Folgenden werden deshalb nur einige Beispiele sowie die im Rahmen dieser Arbeit verwendeten genannt.
Neben den in Ada implementierten Metriken wie „McCabe“ und „Halstead“ existiert ein von Axivion entwickeltes PlugIn, das Quellcode-Metriken wie LOC, Kommentarzeilen uvm. berechnet.
Je nach Sprache und Frontend werden auch diverse Werte aus der IML in den RFG übernommen, wie zum Beispiel die Anzahl der Parameter von Methoden.
Zusätzlich existieren einige Python-Skripte, die sich als sehr hilfreich bei der Einarbeitung in die Skripting-Schnittstelle erwiesen haben. Diese berechnen unter anderem „Depth of Inheritance“, „Number of Children Classes“ und den in Abschnitt 2.3 vorgestellten „Maintainability Index“.

2.5.5 Klonerkennung

Zusätzlich zu den bereits vorgestellten Features bringt Bauhaus auch eine Klonerkennung mit, die in der Lage ist verschiedene Klontypen zu identifizieren und visualisieren. Zu diesem Zweck stellt Bauhaus zwei verschiedene Werkzeuge zur Verfügung. Das Tool *clones* identifiziert Klone auf Basis des Quellcodes, während *ccdiml* auf Basis der IML arbeitet. Die Ergebnisse können in beiden Fällen in verschiedenen Formaten ausgegeben werden, die dann weiterverarbeitet werden können. Ein Format ermöglicht etwa die Visualisierung der gefundenen Klone in Gravis. Ein anderes Format lässt sich in eine Tabellenkalkulation importieren und dort für weitere Analysen verwenden.
Nähere Informationen zur Klonerkennung und -klassifizierung können der Diplomarbeit von Stefan Bellon entnommen werden [Bel02].

Die Möglichkeiten der Klonvisualisierung sind in der Diplomarbeit von Felix Beckwermert [Bec06] dargestellt.

KAPITEL 3

Lösungsansatz

Dieses Kapitel stellt den gewählten Lösungsansatz für die Implementierung vor und erläutert wichtige Designentscheidungen.
In diesem Zusammenhang wird zunächst die in der Aufgabenstellung erwähnte Entscheidung zwischen Ada(95) und Python dargelegt und begründet, um dann das konkrete Design des Tools darzulegen.
Anschließend wird die Umsetzung der einzelnen Indikatoren unter Berücksichtigung der Besonderheiten von IML und RFG sowie die Präsentation der Ergebnisse erläutert.
Zum Abschluß des Kapitels folgt noch ein Abschnitt, der den verwendeten Testcode beschreibt.

Kapitelinhalt

3.1 Technologie

Dieser Abschnitt gibt Auskunft darüber, welche Technologien im Rahmen der Arbeit verwendet wurden und – sofern eine Wahlmöglichkeit bestand – aus welchen Gründen die jeweilige Entscheidung für eine bestimmte Technologie gefallen ist.

Die Entwicklung des Tools wurde weitestgehend unter Linux durchgeführt. Sowohl die Bauhaus-Suite als auch der Ada-Compiler und Python sind auch für Windows verfügbar. Es wäre an dieser Stelle also grundsätzlich auch die Verwendung von Windows als Betriebssystem in Frage gekommen. Da jedoch Bauhaus (v.a. Gravis) aufgrund eines Fehlers in den Ada-Bibliotheken zeitweilig unter Windows instabil lief und der Großteil der Bauhaus-Entwicklung unter Linux stattfindet, wurde von dieser Möglichkeit Abstand genommen.

Die zweite wichtige Technologieentscheidung ist die zwischen den beiden in der Aufgabenstellung zur Wahl gestellten Programmiersprachen: Ada (95) und Python.
Nach reiflicher Überlegung fiel die Entscheidung hier zugunsten von Python aus. Der nachfolgende Abschnitt stellt zunächst die beiden genannten Programmiersprachen kurz vor und nennt dann die – die Entscheidung begründenden – Vor- und Nachteile der jeweiligen Sprache.

Entscheidung zwischen Ada und Python

Ada (95) wurde auf Veranlassung des amerikanischen Verteidigungsministeriums entwickelt (vgl. [Bar98]) und eignet sich gut für den Einsatz in sicherheitskritischen Bereichen, weil die Sprache sehr restriktiven Regeln folgt, bei deren Verletzung sofort Fehler ausgelöst werden. Es handelt sich um eine kompilierte, objektorientierte Programmiersprache mit statischer Typbindung, die als relativ komplex und schwer zu erlernen gilt. Der Ursprung der Sprache geht bis in die 1970er Jahre zurück.

Python auf der anderen Seite wurde laut [pyt07] in den 1990er Jahren von Guido van Rossum (Centrum voor Wiskunde en Informatica in Amsterdam) mit dem Ziel entwickelt, eine möglichst *einfache* und *übersichtliche* Sprache zu entwerfen. Es handelt sich um eine Skriptsprache, die verschiedene Programmierparadigmen unterstützt und im Gegensatz zu Ada auf statische Typisierung verzichtet.

Das deutsche Entwicklerwiki[1] bietet eine ausführliche Übersicht über verschiedene Programmiersprachen, zu denen auch Ada[2] und Python[3] zählen. Eine Auswahl daraus abgeleiteter Unterscheidungsmerkmale von Ada und Python ist in Form einer Gegenüberstellung in Tabelle 3.1 dargestellt.
Als besondere Vorteile, die für Python sprechen, erscheinen die *schnelle* Entwicklungszeit und die umfangreiche Standardbibliothek. Für Ada spricht vor allem, dass bei einer kompilierten Sprache – in der Regel – ein besseres Laufzeitverhalten zu vermuten ist. Die restlichen Punkte sprechen je nach Projektkontext mal für die eine, mal für die andere Sprache.

[1] `http://www.wikiservice.at/dse/wiki.cgi?StartSeite`
[2] `http://www.wikiservice.at/dse/wiki.cgi?SpracheAda`
[3] `http://www.wikiservice.at/dse/wiki.cgi?SprachePython`

ADA	Python
kompilierte Sprache	interpretierte Sprache
statische Typisierung	dynamische Typisierung
gute Eignung für sicherheitskritische Bereiche	gute Eignung für schnelle und prototypische Entwicklungen
relativ komplex; gilt deshalb als schwer / aufwendig zu lernen	*einfache* Sprache; gilt deshalb als leicht erlernbar
durch die sicherheitsorientierte Architektur hoher Tippaufwand	durch die *einfache* Architektur sind Programme in der Regel kürzer als in anderen Sprachen
	umfangreiche Standardbibliothek

Tabelle 3.1: Ada vs. Python

Neben der bislang betrachteten eher technischen Seite ist speziell im Kontext dieser Arbeit als zweites Entscheidungskriterium die Frage zu beachten, von welcher Sprache der Anwender am Meisten profitiert bzw. die geringsten Nachteile hat.
Die Verwendung von Ada hätte für den Anwender vor allem den Vorteil des –bereits erwähnten – *besseren* Laufzeitverhaltens, das für kompilierte Sprachen in der Regel gegenüber interpretierten Sprachen angenommen wird.
Auf der anderen Seite wäre es für den Anwender in der Regel nicht möglich den Index an die Erfordernisse seines Projekts anzupassen, indem er Qualitätsindikatoren hinzufügt oder entfernt. Und selbst wenn der Zugriff auf den Quellcode und damit die Änderung des Tools möglich wäre, müsste die Bauhaus-Suite nach jeder Änderung neu kompiliert werden, was je nach verwendetem Rechner durchaus einen nicht unerheblichen Zeitaufwand bedeuten kann. Auch die Behebung eventueller Fehler gestaltet sich an dieser Stelle in der Regel schwierig bis unmöglich und erfordert die Einschaltung eines Bauhaus-Entwicklers und das Warten auf eine korrigierte Version.
Ein weiterer Nachteil offenbart sich bei der Einbindung in einen Build-Prozess, zum Zwecke der kontinuierlichen Überwachung der technischen Qualität, die sich bei einem kompilierten Programm zwangsweise unflexibler gestaltet, als bei einem oder mehreren Skripten, die beliebig angepasst werden können.
Auch muss der Anwender sich auf die Aussagen einer etwaigen Dokumentation verlassen und kann in der Regel nicht selbst überprüfen welche Berechnungen dem gelieferten Ergebnis zugrunde liegen. Dadurch geht ein guter Teil an Transparenz verloren, die insbesondere Entwicklern offenbar (vgl. Akzeptanz des Maintainability-Index in Abschnitt 2.3) sehr wichtig ist.

Trotz des offensichtlichen Nachteils des zu vermutenden *schlechteren* Laufzeitverhaltens von Python – weil es sich um eine interpretierte Sprache handelt – sprechen aus Anwender-Sicht einige Gründe deutlich für diese Sprache.
Hier wäre zum Einen die hohe Transparenz der Analyse durch die Verfügbarkeit des Quellcodes und die leichtere Einbindung von Skripten in den Buildprozess zu nennen. Zum Anderen aber auch die bei der Verwendung von Ada in diesem Umfang nicht vorhandenen Möglich-

keiten, die Analyse an die spezifischen Gegebenheiten des jeweiligen Projektes anzupassen. So ist es dem Anwender bei einer Implementierung in Python zum Beispiel ohne weiteres möglich, die Grenze von 50 Attributen, die eine Klasse deklarieren „darf“ (vgl. Qualitätsindikator *Gottklasse (Attribut)* in Abschnitt 2.4.2.10), an seine eigenen Wünsche anzupassen; oder auch einen Qualitätsindikator vollständig aus der Berechnung zu entfernen oder durch einen Anderen zu ersetzen.
Zwar beeinträchtigen solche Modifikationen die Vergleichbarkeit der Ergebnisse im Schwellwert-Repository und müssten entsprechend beim Einpflegen berücksichtigt bzw. ausgefiltert werden, aber die Anpassung der Analyse an den jeweiligen Projektkontext wäre sicherlich stellenweise sinnvoll. Man nehme beispielsweise einmal an, dass es in einem Projekt die Vorgabe gibt, Kommentare vor Methoden grundsätzlich – aus welchen Gründen auch immer – mit *//** einzuleiten und keinesfalls */** zu verwenden. Das Ergebnis des Qualitätsindikators *informelle Dokumentation* wäre bei entsprechend strenger Auslegung seiner Definition zwangsweise irreführend.
Der sicherlich größte Nachteil bei der Verwendung der Scripting-Schnittstelle von Bauhaus ist die zwangsweise damit verbundene Beschränkung auf die Funktionalitäten der Schnittstelle sowohl bei der initialen Entwicklung als auch bei eventuellen Modifikationen durch den Anwender. Da der Funktionsumfang der Skripting-Schnittstelle jedoch für die gestellte Aufgabe ausreicht, stellt dies hier nur eine unbedeutende Einschränkung dar.

Zusammenfassend lässt sich festhalten, dass beide Sprachen für die Aufgabenstellung auf technischer Seite ihre Vor- und Nachteile haben, die eine Entscheidung allein auf Basis dieser Betrachtungsweise beliebig erscheinen lassen würde. Das Hauptargument ergibt sich deshalb aus der Anwendersicht.
Die wesentlichen Entscheidunskriterien für Python sind:

1. Die Möglichkeit den Index bei Bedarf schnell an spezifische Projekte anpassen zu können – indem etwa einzelne Metriken hinzugefügt, entfernt, angepasst oder erweitert werden – ohne gleich die gesamte Bauhaus-Suite neu kompilieren und linken zu müssen. Diese Möglichkeit ist natürlich insbesondere für diejenigen Bauhaus-Nutzer interessant, die keinen Zugriff auf den Bauhaus-Quellcode haben.

2. Die Beschränkung auf den Funktionsumfang der Scripting-Schnittstelle ist vernachlässigbar, da alle zur Lösung der Aufgabe nötigen Bauhaus-Funktionalitäten auch über die Scripting-Schnittstelle zugreifbar sind. Hinzu kommt, dass die Schnittstelle beständig weiterentwickelt wird und sich im Laufe der Zeit immer mehr dem Funktionsumfang den Ada bietet annähern wird.

3. Da es bei Diplomarbeiten trotz des begrenzten Zeitrahmens nicht unüblich ist verschiedene Wege prototypisch auszuprobieren, erscheint es wünschenswert diese möglichst schnell implementieren zu können.

3.2 Design und Architektur

Dieser Abschnitt dient der Vorstellung des Designs des zu entwickelnden Werkzeugs. Es wird zunächst ein Überblick über das Systemdesign geliefert und anschließend das Design der einzelnen Module vorgestellt.

Da einige der durch die Qualitätsindikatoren vorgegebenen Metriken (wie z.B. die Anzahl von Methoden/Attributen die eine Klasse deklariert) auch für andere Zwecke – außer dem Code-Quality-Index – von Nutzen sein können, liegt es nahe das Design verstärkt daraufhin auszurichten, die Wiederverwendbarkeit einzelner Metriken in einem anderen Kontext zu ermöglichen. Aus diesem Grund wird der Programmablauf in drei verschiedene Phasen unterteilt (vgl. Abbildung 3.1).

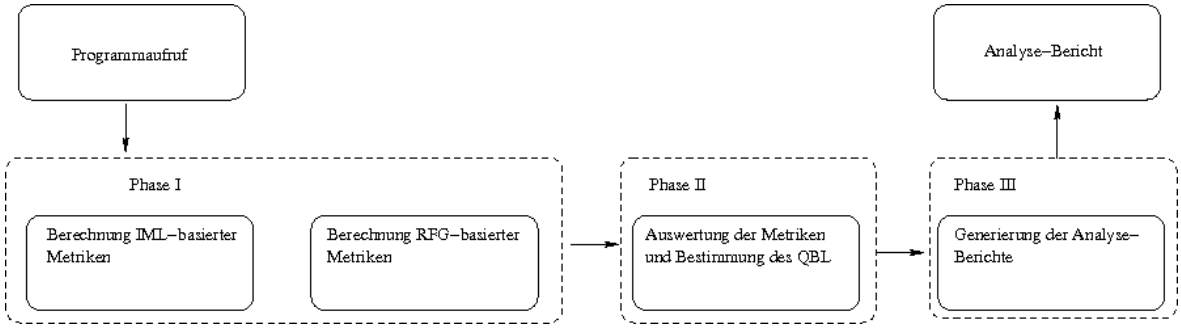


Abbildung 3.1: Programmphasen

In Phase I werden die verschiedenen Metriken berechnet und die Ergebnisse als Attribute an passende Knoten bzw. Kanten im RFG annotiert. Diese Annotation im RFG erfolgt sowohl für rein RFG-basierte Indikatoren als auch für ganz oder teilweise auf der IML basierende, weil die IML über das Skripting-Interface nur lesend zugreifbar ist. Für die Ermittlung des zu einem IML-Knoten korrespondierenden RFG-Knoten wird das von Bauhaus mitgelieferte Skript `iml2rfg` verwendet.
Phase II beinhaltet die Auswertung der einzelnen Metriken. Die Verletzungen der einzelnen Qualitätsindikatoren werden hier gezählt und soweit möglich die Code-Position des Auftretens ermittelt. Auf Grundlage der Zählung wird zum Abschluß der Auswertung noch der Quality-Benchmark-Level des untersuchten Systems berechnet.
Die Trennung von Metrikberechnung und Auswertung erleichtert die Wiederverwendung einzelner Metriken erheblich, führt aber auf der anderen Seite zu einer leichten Erhöhung der Laufzeit.
Phase III generiert den Ergebnisbericht in Form von statischen HTML-Seiten. Zusätzlich kann optional die Verletzungszahl der einzelnen Indikatoren als XML-Datei ausgegeben werden, die – nach kleinen Anpassungen – direkt in das von SQS vorgehaltene Repository importiert

werden kann. Da diese Option nicht zur Aufgabenstellung gehört wird sie an dieser Stelle nur der Vollständigkeit halber erwähnt und im Folgenden nicht weiter vertieft.

Parallel zur Aufteilung in drei Phasen verteilt sich das Werkzeug auf mehrere Module, die sich jeweils aus weiteren Skripten zusammensetzen. Eine Übersicht über die einzelnen Module bietet Abbildung 3.2. Die eingezeichneten Pfeile geben jeweils darüber Auskunft, welche Module von einem bestimmten Modul verwendet – sprich aufgerufen – werden.
Die Modularisierung erhöht sowohl die Wartbarkeit als auch die Wiederverwendbarkeit, hat jedoch auch Nachteile. So ist es systembedingt durch die Annotierung der Metrik-Resultate im RFG, insbesondere bei den IML-basierten Metriken, stellenweise nicht sinnvoll möglich die exakte Codestelle einer Verletzung festzuhalten.
Eine Switch-Anweisung ohne `default` beispielsweise hat keine direkte Entsprechung im RFG. An dieser Stelle wird lediglich an der umschließenden Methode vermerkt, dass eine solche Switch-Anweisung in der Methode gefunden wurde und nicht exakt welche (der möglicherweise mehreren) Switch-Anweisungen kein `default` hat. Dieser Nachteil wird billigend in Kauf genommen, weil in der Regel eine hinreichend genaue Annäherung an die fragliche Code-Position möglich ist, so dass die Stelle leicht zugeordnet und nachvollzogen werden kann.
Das Modul ohne eigene Bezeichnung enthält das Aufrufskript, sowie einige Hilfsfunktionen, die im Vorfeld der Analyse Bibliothekselemente ausfiltern. In diesem Zusammenhang werden die folgenden – um Bibliothekselemente bereinigt – für die eigentliche Analyse verwendeten Sichten generiert:

CQI-ANALYSIS: Dabei handelt es sich um die klassische `BASE-View` aus der die Bibliothekselemente und compilergenerierte Elemente entfernt wurden.

CQI-FILE: Es handelt sich um die `FILE-View` aus der die Bibliothekselemente und compilergenerierte Elemente entfernt wurden.

CQI-MODULE: Es handelt sich um die `MODULE-View` aus der die Bibliothekselemente und compilergenerierte Elemente entfernt wurden.

CQI-LIFTEDMODULE: Diese Sicht entspricht in ihrer Hierarchie der Sicht `CQI-MODULE` Die Kanten sind der `BASE-View` entnommen. Diese Sicht wird vorangig für die Berechnung der Indikatoren verbotene Paket- und Dateiliebe verwendet.

Die ausgefilterten Elemente ergeben sich zum Einen aus der `ENVIRONMENT`-View und zum andern aus Elementen deren Attribut `Source.Path` einen Pfad enthält, der beim Aufruf als Kommandozeilen-Parameter übergeben werden kann.

Die von den einzelnen Modulen bereit gestellte Funktionalität wird in den nachfolgenden Abschnitten kurz erläutert.

Um die Wiederverwendbarkeit der Metriken noch weiter zu erleichtern, wird auf Objektorientierung und die damit verbundene Abstraktion weitgehend verzichtet, um die Les- und Verstehbarkeit des Codes zu erhöhen. Die meisten Metriken können mit weniger als 150 Code-Zeilen implementiert werden und die Auswertung bleibt in der Regel unter 100 häufig sogar unter 50 Zeilen. Die mit einer Objektorientierung einhergehenden Abstraktionen würden das Code-Volumen an dieser Stelle nur unnötig aufblähen und sich damit negativ auf Les- und Verstehbarkeit auswirken.

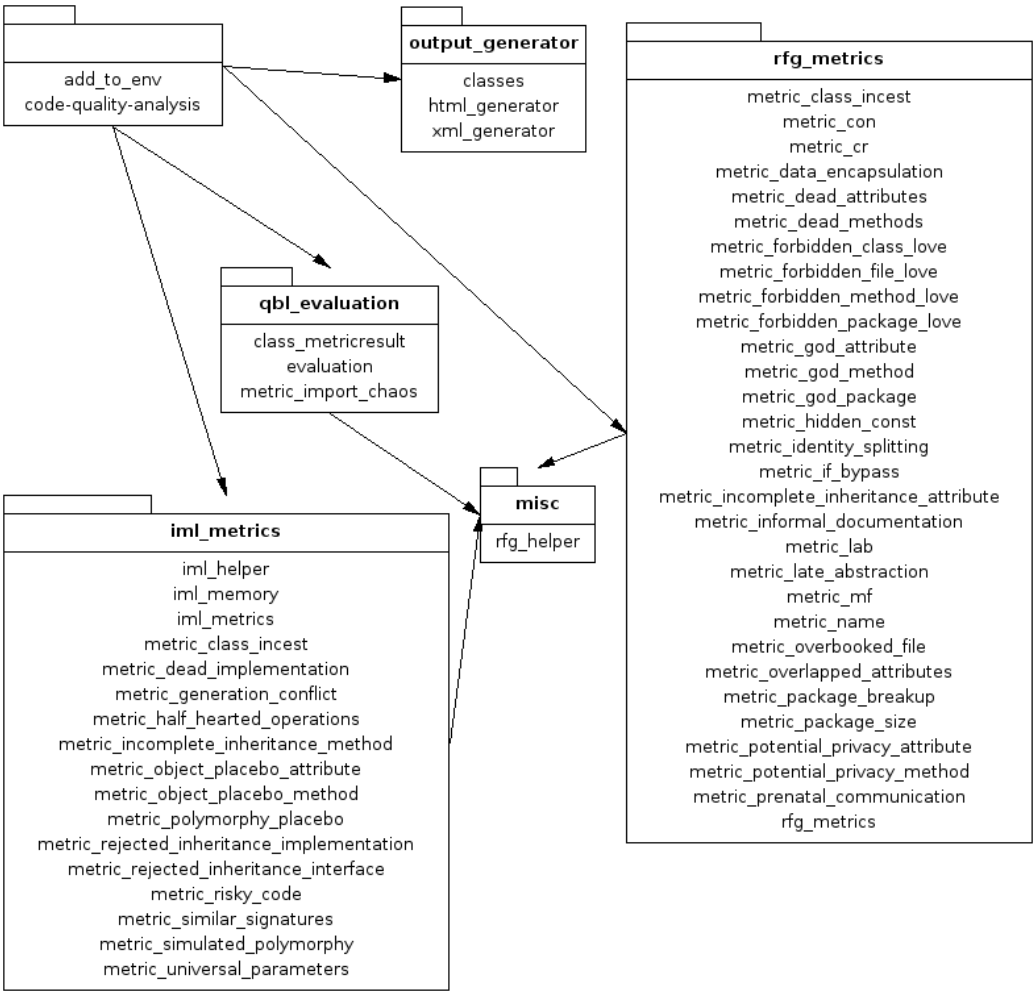


Abbildung 3.2: Module

3.2.1 Das Modul `iml_metrics`

Das Module beinhaltet alle IML-basierten Metriken jeweils nach Qualitätsindikatoren gruppiert für die sie berechnet werden.
Das Skript *iml_helper* stellt einige Hilfsfunktionen bereit, die im Zuge der IML-Analyse mehrfach benötigt werden.
Auch das Skript *iml_memory* stellt eine Hilfsfunktionalität bereit, die sich jedoch grundlegend von dem Charakter der Hilfsfunktionen in *iml_helper* unterscheidet.

Die IML für industrielle Systeme relevanter Größe kann sehr schnell Dimensionen annehmen, die der Arbeitsspeicher eines 32Bit-PCs nicht mehr fassen kann, was eine Analyse zumindest der IML-basierten Indikatoren unmöglich machen würde. Da alle systemweiten Indikatoren auf Basis eines RFG analysiert werden können, ist es für die Analyse jedoch nicht zwingend notwendig eine IML-Repräsentation des Gesamtsystems zu erstellen. Statt dessen können dem Analyse-Werkzeug mehrere Teil-IMLs übergeben werden, die dann nacheinander abgearbeitet werden. Lediglich ein RFG des Gesamtsystems ist zwingend notwendig, um die systemweiten Indikatoren (z.B. *nachlässige Kommentierung*) berechnen zu können und die Ergebnisse der IML-basierten Metriken zu vermerken.
Um nun zu vermeiden, dass die gleichen Verletzungen – weil sie in verschiedenen Teil-IMLs enthalten ist – mehrfach gezählt wird, „merkt“ sich *iml_memory* die bereits gefundenen Verletzungen.

Die Metriken für die folgenden Qualitätsindikatoren müssen IML-basiert erhoben werden:

- allgemeine Parameter
- ausgeschlagenes Erbe (Implementierung)
- ausgeschlagenes Erbe (Schnittstelle)
- Generationskonflikt
- halbherzige Operationen
- Objektplacebo (Attribut)
- Objektplacebo (Methode)
- Polymorphieplacebo
- Risikocode
- signaturähnliche Klassen
- simulierte Polymorphie
- tote Implementierung
- unvollständige Vererbung (Methode)

3.2.2 Das Modul `rfg_metrics`

Dieses Modul beinhaltet die Metriken für Qualitätsindikatoren, die auf Basis des RFG erhoben werden können und nicht auf von Bauhaus vorberechnete Metriken zurückgreifen können:

- Attributüberdeckung
- Datenkapselaufbruch
- falsche Namenslänge
- Gottklasse (Attribut)
- Gottklasse (Methode)
- Gottpaket
- heimliche Verwandtschaft
- Identitätsspaltung
- Importlüge
- informelle Dokumentation
- Interface Bypass
- Klässchen
- Klasseninzest
- Konstantenregen

- Labyrinthmethode
- maskierende Datei
- Namensfehler
- Paketchen
- Pakethierarchieaufbruch
- potentielle Privatsphäre (Attribut)
- potentielle Privatsphäre (Methode)
- pränatale Kommunikation
- späte Abstraktion
- tote Attribute
- tote Methoden
- überbuchte Datei
- unvollständige Vererbung (Attribut)
- verbotene Dateiliebe
- verbotene Klassenliebe
- verbotene Methodenliebe
- verbotene Paketliebe
- versteckte Konstantheit

3.2.3 Das Modul `qbl_evaluation`

Dieses Modul ist für die Auswertung der zuvor berechneten Metriken zuständig und ermittelt auf Basis dieser Werte den Quality-Benchmark-Level des analysierten Systems.
Die Ergebnisse bzgl. der einzelnen Qualitätsindikatoren werden in Objekten vom Typ `MetricResult` vorgehalten. Diese verwalten die Resultate in einem Python-Dictionary mit den zwei Listen „overview“ und „details“. Während die Liste „overview“ lediglich die Gesamtzahl an Verletzungen und den pro 1000 Lines of Code normalisierten Wert enthält, werden in der Liste „details“ die detaillierten Trefferlisten vorgehalten.
Die einzelnen Ergebnisse werden wiederum zusammen mit anderen Werten in einem Python-Dictionary gesammelt, welches letztlich das Ergebnis darstellt und in beliebige Formate überführt und ausgegeben werden kann.
Die erwähnten „anderen Werte“ beinhalten neben dem Quality-Benchmark-Level auch Informationen über das analysierte System wie z.B. LOC, die in der Analyse berücksichtigten Code-Dateien und die Programmiersprache (C++ oder Java) des analysierten Systems.
Auch wenn die Verwendung von mehrfach ineinander verschachtelten Dictionaries und Listen zunächst etwas umständlich wirkt, hat dies den Vorteil, das die Funktionsweise bekannt ist und die zugrundeliegenden Datenstrukturen effizient implementiert sind. Zusätzlich kann auf diese Weise bei der Generierung der Ausgabe auf die umfangreiche Python-Standardbibliothek zurückgegriffen werden, was die Implementierung neuer Ausgabeformen immens beschleunigen kann.
Zusätzlich wird in diesem Modul der Qualitätsindikator *Importchaos* berechnet, da dieser weder auf Basis der IML noch des RFG realisiert werden kann. Stattdessen ist es in diesem Fall erforderlich, direkt auf den Quellcode zuzugreifen.

3.2.4 Das Modul `output_generator`

Das Modul `output_generator` beinhaltet Skripte, die das in Abschnitt 3.2.3 erwähnte „Ergebnis-Dictionary“ in das jeweilige Ausgabeformat überführen. Da die Generierung der statischen HTML Seiten, die die Aufgabenstellung fordert, in der Regel keine Anpassungen an einen Projektkontext mit sich bringt, erscheint es zweckdienlich für die Datenstrukturen der zu generierenden HTML-Seiten einen objektorientierten Ansatz zu wählen. Die resultierende Klassenstruktur ist in Abbildung 3.3 abgebildet. Durch die dynamische Typisierung von Python werden die Datentypen der Klassenattribute erst zur Laufzeit festgelegt und sind im Klassendiagramm entsprechend nicht enthalten. Die Klassen `Code_Snippet`, `HTML_Table` und `HTML_Clones_Table` werden als Hilfsstrukturen innerhalb der verschiedenen Seitentypen (`Page`-Klassen) verwendet. Die Verwendungszwecke der einzelnen Seitentypen werden in Abschnitt 3.4 näher erläutert. Dort finden sich außerdem auch Beispielabbildungen der generierten HTML-Seiten.

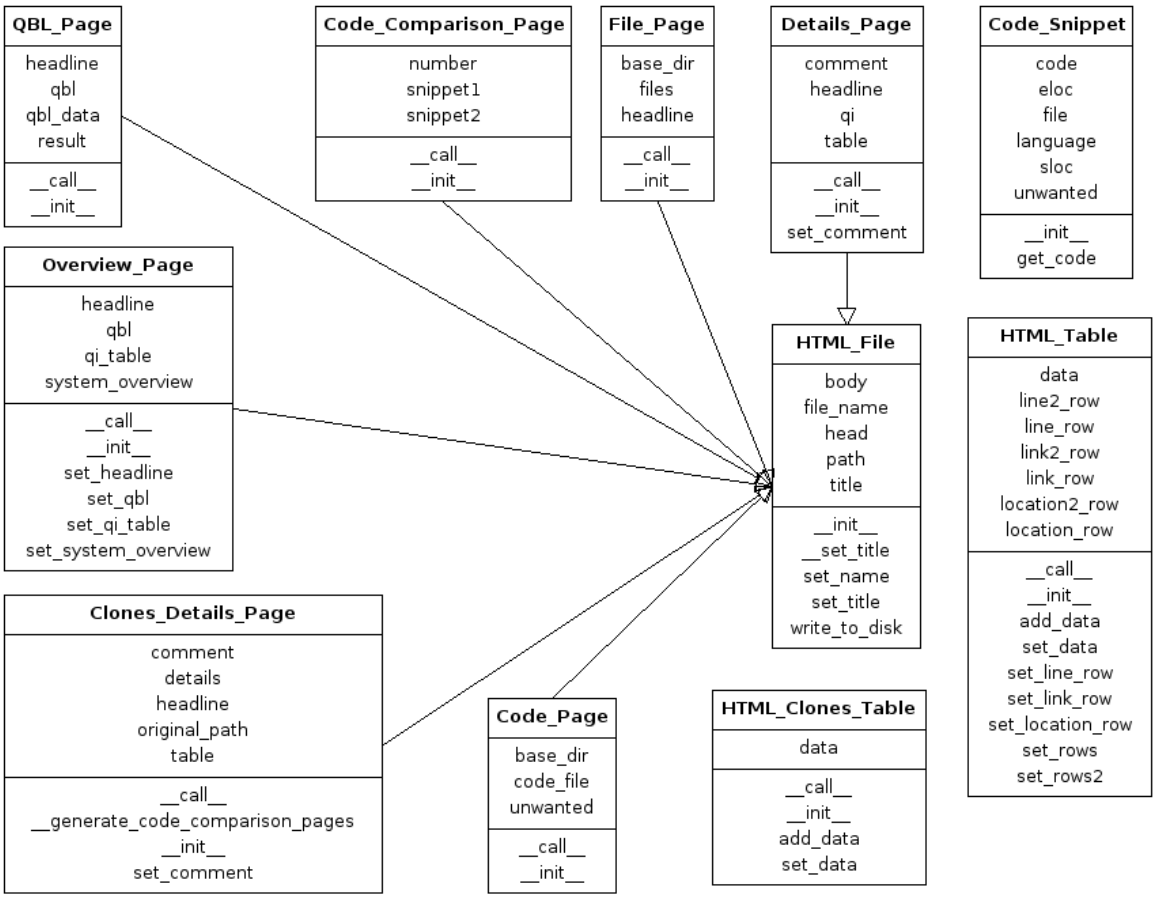


Abbildung 3.3: Klassendiagramm

3.2.5 Das Modul `misc`

In diesem Modul sind verschiedene Hilfsfunktionen enthalten, die im Zuge der Analyse an verschiedenen Stellen benötigt werden.
Hierzu gehören zum Beispiel Methoden, die alle (auch indirekten) Unterklassen bzw. Oberklassen einer Klasse ermitteln. Ebenso stellt dieses Modul die Möglichkeit bereit, die ein Artefakt umschließende Klasse oder deren Namen zu ermitteln.

3.2.6 Konfiguration

Der Code-Quality-Index beinhaltet mehrere Aspekte, die verändert werden können bzw. sollen. Um diesem Umstand Rechnung zu tragen wurde eine Konfigurationsdatei im XML-Format definiert, in der die entsprechenden Werte bzw. Defintionen geändert werden können. Wie der zugehörigen *Document-Type-Definition* (Listing 3.1) entnommen werden kann, sind die Namenskonventionen, die Schwellwerte und QBL-Einstufungen der Qualitätsindikatoren und die `Package`-Definition konfigurierbar. Letztere wirkt sich jedoch ausschließlich bei der Analyse von `C++` aus.
Die Anpassung der Namenskonventionen an den jeweiligen Projektkontext wird bereits in der Definition des Qualitätsindikators (vgl. [SSM06, Seite 249]) ausdrücklich zugelassen und war entsprechend zu ermöglichen.

Die Schwellwerte basieren laut [SSM06] auf einem Repository, das ständig durch die Vermessung weiterer Software-Systeme erweitert wird. Entsprechend ändern sich diese Werte unter Umständen regelmäßig und müssen entsprechend leicht anpassbar sein. Die Schwellwerte für jeden Qualitätsindikator werden für die beiden definierten Sprachen *C++* und *Java* seperat gespeichert, um bei einer möglichen Erweiterung des Index auf weitere Sprachen eine möglichst unkomplizierte Ergänzung der neuen Schwellwerte zu ermöglichen.

Die Definierbarkeit des Paketbegriffs für die Sprache *C++* hat den Hintergrund, dass aus [SSM06] stellenweise nicht klar hervorgeht, ob als Paket ein *Namespace* oder ein *Verzeichnis* zu betrachten ist. Da beide Varianten möglich sind, gibt diese Option dem Anwender die Möglichkeit selbst zu entscheiden, welche die für ihn sinnvollere Variante ist.

Listing 3.1: Document-Type-Definition

```
1 <!ELEMENT code_quality_config_file (naming_conventions_config,
     package_config, qbl_data_cpp, qbl_data_java)>
2
3 <!ELEMENT naming_conventions_config (name_setting*)>
4
5 <!ELEMENT name_setting EMPTY>
6 <!ATTLIST name_setting
7     name CDATA #REQUIRED
8     value CDATA #REQUIRED
```

```
 9 >
10
11 <!ELEMENT package_config (package_type)>
12 <!ELEMENT package_type EMPTY>
13 <!ATTLIST package_type
14     type (namespace|directory) #REQUIRED
15 >
16
17 <!ELEMENT qbl_data_cpp(quality_indicator_cpp*)>
18
19 <!ELEMENT qbl_data_java(quality_indicator_java*)>
20
21 <!ELEMENT quality_indicator_cpp EMPTY>
22 <!ATTLIST quality_indicator_cpp
23         name CDATA #REQUIRED
24         relevance (2|3|4|5) #REQUIRED
25         threshold2 CDATA #REQUIRED
26         threshold3 CDATA #REQUIRED
27         threshold4 CDATA #REQUIRED
28         threshold5 CDATA #REQUIRED
29 >
30
31 <!ELEMENT quality_indicator_java EMPTY>
32 <!ATTLIST quality_indicator_java
33         name CDATA #REQUIRED
34         relevance (2|3|4|5) #REQUIRED
35         threshold2 CDATA #REQUIRED
36         threshold3 CDATA #REQUIRED
37         threshold4 CDATA #REQUIRED
38         threshold5 CDATA #REQUIRED
39 >
```

3.3 Umsetzung der einzelnen Indikatoren

In diesem Abschnitt werden die Implementierungen der einzelnen Qualitätsindikatoren vorgestellt. Bei denjenigen Indikatoren, deren Umsetzung sich exakt an den in [SSM06] genannten informellen Hinweisen für die Erkennung orientiert, wird in der Regel auf eine detaillierte Beschreibung verzichtet und lediglich die Programmrepräsentation genannt, auf der die Implementierung basiert sowie Name und Ort (Kanten- bzw. Knotentyp) des Attributes, das die Ergebnisse im RFG annotiert. Bei als IML-basiert angegebenen Indikator-Implementierungen wird immer auch der RFG genutzt. Häufig lediglich um das Ergebnis zu annotieren, in einigen Fällen jedoch auch um eine Vorauswahl verdächtiger Artefakte zu ermitteln. Dies hat den Vorteil, dass eventuell verwendete Bibliotheken vorab ausgefiltert wurden und nicht mit überprüft werden.
Ausführlich betrachtet werden vor allem solche Indikatoren, bei denen in irgendeiner Weise von den Vorgaben des Buches „Code-Quality-Management“ [SSM06] abgewichen wird.

Eine sehr häufig auftretende Abweichung von den Angaben des Buches ist folgende:
die informellen Hinweise zur Erkennung sprechen in der Regel davon, dass zunächst eine Teilmenge von Artefakten ermittelt werden soll, die ein bestimmtes, relativ allgemeines Kriterium erfüllen. Anschließend wird diese Teilmenge weiter eingeschränkt. Ein Beispiel für diese Art von Vorgehen liefert der Indikator *potenzielle Privatsphäre (Attribut)*:

> - *Eine Liste mit allen* `protected`*-Attributen erstellen.*
> - *Alle* `protected`*-Attribute aus der Liste löschen, auf die durch andere Klassen zugegriffen wird (Diese Attribute müssen* `protected` *sein/bleiben).*
> - *Hinzufügen aller* `public`*-Attribute zu der Liste.*
> - *Alle Attribute aus der Liste löschen, auf die von außerhalb der Vererbungshierarchie zugegriffen wird (diese müssen* `public` *bleiben.*
> - *Die verbleibende Liste enthält alle Attribute, deren Sichtbarkeit eingescrhänkt werden kann. Jedes Attribut stellt eine Probleminstanz dar.*
>
> *[SSM06, Seite 270]*

Diese Vorgehensweise ist bei datenbankbasierten Tools empfehlenswert, würde aber im Fall der Bauhaus-Suite unnötig hohe Aufwände verursachen. Stattdessen bietet sich aufgrund der Graph-Struktur folgendes Vorgehen an:

- für jeden im Graph enthaltenen Memberknoten prüfen, ob er `public` oder `protected` ist
- `protected` Memberknoten auf Zugriffe anderer Klassen überprüfen
- `public` Memberknoten auf Zugriffe von außerhalb der Vererbungshierarchie prüfen

- jeder Member bei dem die Frage nach den Zugriffen verneint werden kann, stellt eine Probleminstanz dar und kann in seiner Sichtbarkeit reduziert werden

Im Gegensatz zu dem im Buch dargelegten Vorgehen wird also die Treffermenge schrittweise erhöht, anstatt sie durch das Ausschlußverfahren zu vermindern. Da diese Abweichung durch die Methodik der Programmrepräsentation von Bauhaus in vielen Fällen notwenig bzw. sinnvoll ist und weder Auswirkungen auf die Ergebnisse noch auf das generelle – in den Hinweisen zur Erkennung formulierte – Vorgehen hat, wird diese Abweichung im Folgenden nicht mehr explizit erwähnt.

Sofern nicht anders angegeben, wird jeweils nur die Umsetzung der „Metrikerhebung“ also Phase I betrachtet. Die Auswertung (Phase II) arbeitet in diesen Fällen so, dass alle Knoten bzw. Kanten des Typs, an dem das jeweilige Attribut annotiert ist, durchlaufen werden und das Attribut auf seinen Inhalt hin überprüft wird. Ist ein positiver Befund vermerkt, wird ein Zähler erhöht und die Code-Position der Verletzung ermittelt. Diese so ermittelten Ergebnisse werden dann in der in Abschnitt 3.2.3 beschriebenen Art und Weise gesammelt und aufbereitet.

Bei auf Zyklen beruhenden Indiktatoren, bei denen Knoten mit einer ID markiert werden, werden die Knoten nach IDs gruppiert und dann die für die Ergebnispräsentation notwendigen Daten extrahiert.

Grundlage der Beschreibungen ist das Bauhaus-Release Version 5.5.0. IML-basierte Indikatoren enthalten – sofern nicht anders angegeben – nur eine Beschreibung der auf der C++-IML basierenden Implementierung. Diese liefert mit der – in der Entwicklerversion von Bauhaus enthaltenen – Java-IML nur dort korrekte Ergebnisse, wo die beiden IML-Versionen deckungsgleich sind. Eine gesonderte Berücksichtigung der Java-IML bei der Implementierung ist aus Zeitgründen leider nicht möglich.
Die Implementierungen für RFG-basierte Metriken für *Java* basieren auf RFGs, die mit `java2rfg` generiert werden. Dies hat einerseits den Nachteil, dass sich einige Implementierungen komplizierter gestalten und ein schlechteres Laufzeitverhalten als die *C++*-Variante aufweisen weil mehr Knoten verarbeitet werden müssen. Andererseits hat dieses Vorgehen den Vorteil, dass Paket-Metriken wie *Gottpaket* implementierbar sind (in `jafe/iml2rfg`-basierten RFGs werden in Version 5.5.0 keine `Package`-Knoten generiert). Zusätzlich können auf diese Weise die RFG-basierten Metriken für *Java* auch von Anwendern genutzt werden, die keinen Zugang zu `Jafe` haben.

3.3.1 allgemeine Parameter

Die Umsetzung dieses Qualitätsindikators basiert auf der Programmrepräsentationsform IML.

Die Ergebnisse werden durch das Attribut `Metric.Universal_Parameters` vom Typ `string`, welches an Method- und Routine-Knoten geschrieben wird, im RFG vermerkt.

Um Verletzungen des Qualitätsindikators *allgemeine Parameter* zu finden, werden alle IML-Knoten vom Typ `Explicit_Conversion` durchlaufen und jeweils daraufhin untersucht, ob sie einen Parameter der sie umschließenden Methode betreffen. Ist dies der Fall, werden die betroffenen Knoten nach dem `Mangled_Name` der sie umschließenden Methode gruppiert.
In einem zweiten Schritt wird überprüft, ob der zu konvertierende Typ eine Verallgemeinerung (also eine Oberklasse) des Zieltyps ist. In diesem Fall wird der Name des betroffenen Parameters (das zu konvertierende Objekt) an der fraglichen Methode annotiert. Sind mehrere Parameter in der selben Methode betroffen, werden sie durch Semikolon getrennt.
Derzeit existiert die Einschränkung, dass `static-cast`-Anweisungen scheinbar von Bauhaus nicht korrekt modelliert werden. Entsprechend können Verletzungen durch eine solche Anweisung nicht erkannt werden.

3.3.2 Attributüberdeckung

Die Umsetzung dieses Qualitätsindikators basiert auf der Programmrepräsentationsform RFG.

Die Ergebnisse werden durch das Attribut `Metric.OVERLAPPED_ATTRIBUTES und Metric.Overlapped_By` vom Typ `int bzw. string`, welches an Klassen- bzw. Member-Knoten geschrieben wird, im RFG vermerkt.

Während das eigentliche Ergebnis – die Anzahl der überdeckten Attribute – als Zahl an den jeweiligen Klassen-Knoten geschrieben wird (`Metric.OVERLAPPED_ATTRIBUTES`), erhalten zusätzlich die betroffenen Member (Attribute) zwecks erleichterter Generierung der Detaillisten die Information, durch welches Attribut in welcher Unterklasse sie überdeckt werden. Dieser String hat das Format `Unterklassenname::Datei::Zeile`. Sollten mehrere Attribute an der Überdeckung beteiligt sein, werden die Angaben durch ein Semikolon getrennt. Der Name des Attributes wird nicht mit aufgenommen, weil er laut Indikator-Definition dem Namen des Oberklassen-Attributes entspricht.
Zur Erkennung der Attributüberdeckung werden die Attributnamen aller Klassen ermittelt und jeweils gegen die Attributnamen der direkten und indirekten Unterklassen abgeglichen.

3.3.3 ausgeschlagenes Erbe (Implementierung)

Die Umsetzung dieses Qualitätsindikators basiert auf der Programmrepräsentationsform IML.

Die Ergebnisse werden durch das Attribut `Metric.Rejected_Inheritance(Implementation)` vom Typ `toggle`, welches an Method-Knoten geschrieben wird, im RFG vermerkt.

Die in der Indikatordefinition verlangte Redefinierung einer Methode wird im RFG anhand

des Vorhandenseins einer *Overrides-Kante* nachgewiesen. Für die Überprüfung, ob dies nichtleer geschieht, wird unter Verwendung des Bauhaus-Skripts `iml2rfg` der korrespondierende Knoten in der IML ermittelt und die darin hinterlegten Anweisungen (`Statement-Sequence`) daraufhin überprüft, ob Anweisungen außer `return null` existieren.
Abschließend wird jede nichtleer überdeckende Methode im RFG daraufhin überprüft, ob sie die überdeckte Methode aufruft.

3.3.4 ausgeschlagenes Erbe (Schnittstelle)

Die Umsetzung dieses Qualitätsindikators basiert auf der Programmrepräsentationsform IML.

Die Ergebnisse werden durch das Attribut `Metric.Rejected_Inheritance(Interface)` vom Typ `string`, welches an Method-Knoten geschrieben wird, im RFG vermerkt.

Da eine Unterklasse immer die Schnittstelle ihrer direkten Oberklasse erbt (inkl. eventuell geerbter Artefakte), sind hier nur direkte Vererbungsbeziehungen zu betrachten.
Es werden deshalb die Methoden aller durch eine `Extends`-Kante verbundenen Klassen auf die in der Definition genannten Kriterien hin untersucht.
Die IML wird zum Einen verwendet, um das Kriterium *nichtleer* zu überprüfen und zum Anderen, um im Falle von *Java* Methodensignaturen zu ermitteln. Dies wurde notwendig, weil in auf *Java* basierenden RFGs zum Zeitpunkt der Konzeption durch einen Bug in Bauhaus keine `Inherits-/Override-`Kanten generiert wurden.
Das Attribut wird an die Methode der Unterklasse geschrieben und enthält den Namen der überschriebenen Methode, den Namen ihrer Klasse sowie die Code-Position (bestehend aus Datei und Zeile) der Methodendeklaration. Die Werte sind jeweils durch zwei Doppelpunkte („::“) getrennt.

3.3.5 Datenkapselaufbruch

Die Umsetzung dieses Qualitätsindikators basiert auf der Programmrepräsentationsform RFG.

Die Ergebnisse werden durch das Attribut `Metric.DATA_ENCAPSULATION und Metric.DATA_ENCAPSULATION.ACCESSED_BY` vom Typ `toggle bzw. string`, welches an Member-Knoten geschrieben wird, im RFG vermerkt.

Die Implementierung orientiert sich an den informellen Hinweise zur Erkennung [SSM06, Seite 182]. Zusätzlich zum bloßen Vermerk, dass ein Memberknoten eine Verletzung verursacht (`Metric.DATA_ENCAPSULATION`), werden in `Metric.DATA_ENCAPSULATION.ACCESSED_BY` die Daten der zugreifenden Artefakte festgehalten, um die Generierung der Detaillisten zu beschleunigen.

3.3.6 duplizierter Code

Die Umsetzung dieses Qualitätsindikators basiert nicht auf einer Programmrepräsentationsform von Bauhaus.

Für diesen Indikator wird kein Attribut in den RFG geschrieben.

Dieser Indikator ist ausschließlich über eine Auswertungs-Komponente implementiert.
Um die geforderten Code-Duplikate mit einer Länge von mindestens 40 Zeilen zu ermitteln, wird das in Bauhaus enthaltene Tool *clones* mit den folgenden Parametern aufgerufen:

C++: `clones -scope -min 40 -lang c+++h`

+ [Pfad des analysierten Systems]

Java: `clones -scope -min 40 -lang java`

+ [Pfad des analysierten Systems]

Die Ausgabe des Tools wird zeilenweise in ein Array gespeichert.
Jede Zeile enthält eine Reihe von durch Leerzeichen getrennten Informationen, von denen in diesem Zusammenhang ausschließlich die ersten sieben von Interesse sind.
Der siebte Wert jeder Zeile gibt Auskunft über den Typ des gefundenen Klons. Ist dieser mit `1` angegeben, handelt es sich um ein exaktes Duplikat, wie es die Indikatordefinition verlangt.
Alle in Typ-1 Klonen enthaltene Zeilen werden in einem Array (als Tupel aus Datei und Zeilennummer) registriert, dessen Länge schließlich die Gesamtverletzungszahl für diesen Indikator darstellt.
Die restlichen 6 Werte enthalten die folgenden Informationen:

- Datei 1
- Startzeile des Klons in Datei 1
- Endzeile des Klons in Datei 1
- Datei 2
- Startzeile des Klons in Datei 2
- Endzeile des Klons in Datei 2

Diese Informationen werden in gleicher Reihenfolge auch für die Detaillisten an das für diesen Indikator erzeugte Objekt vom Typ `MetricResult` übergeben.

3.3.7 falsche Namenslänge

Die Umsetzung dieses Qualitätsindikators basiert auf der Programmrepräsentationsform RFG.

Die Ergebnisse werden durch das Attribut `Metric.NAME_LENGTH` vom Typ `int`, welches an die Knoten der in der Definition genannten Artefakte geschrieben wird, im RFG vermerkt.

Für die von diesem Indikator betroffenen Artefakte wird lediglich die Länge des Names ermittelt und an das jeweilige Artefakt geschrieben. Die eigentliche Bewertung (< 2 bzw. > 50) erfolgt erst bei der Auswertung. Dies führt dazu, dass die Metrik universell auch für andere Zwecke eingesetzt werden kann.

3.3.8 Generationskonflikt

Die Umsetzung dieses Qualitätsindikators basiert auf der Programmrepräsentationsform IML.

Die Ergebnisse werden durch das Attribut `Metric.GENERATION_CONFLICT` vom Typ `string`, welches an (Ober)Klassen-Knoten geschrieben wird, im RFG vermerkt.

Die Implementierung hält sich an die Vorgaben des Buches (vgl. [SSM06, Seite 191]). Die IML kommt wie schon zuvor ausschließlich zur Überprüfung des Kriteriums *nichtleer* zum Einsatz.

3.3.9 Gottdatei

Die Umsetzung dieses Qualitätsindikators basiert auf der Programmrepräsentationsform RFG.

Die Ergebnisse werden durch das Attribut `Metric.Lines.Loc` vom Typ `int`, welches an Module-Knoten geschrieben wird, im RFG vermerkt.

Die Ermittlung der Anzahl der in einer Datei enthaltenen Codezeilen wird von dem in Abschnitt 2.5.4 erwähnten Plug-In übernommen. Deshalb existiert für diesen Indikator lediglich eine Auswertungskomponente, die die ermittelten LOC gegen den Grenzwert von 2000 prüft. Da für Java keine `FILE`-View existiert, werden hier die konkreten Dateien geöffnet und die Anzahl der enthaltenen Zeilen gezählt. Eine Annotierung im RFG erfolgt in diesem Fall nicht.

3.3.10 Gottklasse (Attribut)

Die Umsetzung dieses Qualitätsindikators basiert auf der Programmrepräsentationsform RFG.

Die Ergebnisse werden durch das Attribut `Metric.ATTRIBUTES` vom Typ `int`, welches an Klassen-Knoten geschrieben wird, im RFG vermerkt.

Es wird die Anzahl der deklarierten Attribute jeder Klasse ermittelt. Die Bewertung, ob es sich um eine *Gottklasse (Attribut)* handelt, folgt erst im Zuge der Auswertung.
Attribute gelten als in der jeweiligen Klasse deklariert, wenn sie keine ausgehenden `Inherits`-Kanten haben.

3.3.11 Gottklasse (Methode)

Die Umsetzung dieses Qualitätsindikators basiert auf der Programmrepräsentationsform RFG.

Die Ergebnisse werden durch das Attribut `Metric.METHODS` vom Typ `int`, welches an Klassen-Knoten geschrieben wird, im RFG vermerkt.

Wie bei *Gottklasse (Attribut)* wird lediglich die Anzahl der selbst deklarierten Methoden ermittelt. Die Bewertung, ob die gefundene Anzahl eine Verletzung darstellt, folgt erst im Zuge der Auswertung.
Es werden alle Methoden gezählt, die keine ausgehende `Inherits`-Kante haben und mindestens die gleiche Sichtbarkeit wie ihre umgebende Klasse.

3.3.12 Gottmethode

Die Umsetzung dieses Qualitätsindikators basiert auf der Programmrepräsentationsform RFG.

Die Ergebnisse werden durch das Attribut `Metric.Lines.Loc` vom Typ `int`, welches an Method-Knoten geschrieben wird, im RFG vermerkt.

Die Ermittlung der Anzahl der in einer Methode enthaltenen Codezeilen wird von dem in Abschnitt 2.5.4 erwähnten Plug-In übernommen. Deshalb existiert für diesen Indikator lediglich eine Auswertungskomponente, die die ermittelten LOC gegen den Grenzwert von 200 prüft.

3.3.13 Gottpaket

Die Umsetzung dieses Qualitätsindikators basiert auf der Programmrepräsentationsform RFG.

Die Ergebnisse werden durch das Attribut `Metric.CLASSES` vom Typ `int`, welches an Paket-Knoten (je nach Sprache und Definition vom Typ `Package`, `Directory` oder `Namespace`) geschrieben wird, im RFG vermerkt.

Es wird die Anzahl der öffentlichen nicht geschachtelten Klassen, die in einem `Package` (*Java*)

bzw. `Verzeichnis` oder `Namespace` (*C++*) deklariert sind, bestimmt. Die Bewertung erfolgt erst im Zuge der Auswertung. Auf diese Weise kann die für diesen Indikator implementierte Metrik zusätzlich für den Qualitätsindikator *Paketchen* verwendet werden.

3.3.14 halbherzige Operationen

Die Umsetzung dieses Qualitätsindikators basiert auf der Programmrepräsentationsform IML (im Fall von *Java* nur RFG).

Die Ergebnisse werden durch das Attribut `Metric.Half-Hearted_Operations` vom Typ `string`, welches an Klassen-Knoten geschrieben wird, im RFG vermerkt.

Die Existenz der fraglichen Methodenpaare wird für jede Klasse im RFG überprüft. Für *C++* wird für jede Methode der Typ ihres Parameters (`int` oder `void`) unter Verwendung der IML überprüft, um in der Detailliste anführen zu können, welche der beiden Varianten fehlt.
Da einige der Methoden in mindestens einer Variante auch kompilergeneriert in Erscheinung treten können (z.B. `operator=`), wird als Kriterium dafür das eine Methode tatsächlich implementiert wurde, die Nichtexistenz des Attributes `Element.Is_Artificial` überprüft.
Die fehlenden Methoden werden durch Kommata getrennt an den Klassenknoten annotiert.

3.3.15 heimliche Verwandtschaft

Die Umsetzung dieses Qualitätsindikators basiert auf der Programmrepräsentationsform RFG.

Die Ergebnisse werden durch das Attribut `Metric.CR` vom Typ `bool`, welches an Klassen- und Interface-Knoten geschrieben wird, im RFG vermerkt.

Am Beispiel dieses Indikators wird exemplarisch dargelegt, wie die informellen Hinweise zur Erkennung auf Seite 213 in [SSM06] über die Bauhaus-Scripting-Schnittstelle umgesetzt werden. Aus Gründen der Lesbarkeit werden einzelne Schritte zusammengefasst.

- Es werden alle im Graph enthaltenen Klassen und Interfaces ermittelt
- Für jede Klasse bzw. jedes Interface wird überprüft:
  - ob sie `public` sind und mindestens eine öffentliche Methode bereitstellen. Alle öffentlichen Methoden einer Klasse werden gespeichert, um sie später auf Aufrufe überprüfen zu können.
  - ob mindestens eine Unterklasse bzw. Implementierung existiert.
- Unabhängig von der Erfüllung der Bedingungen wird das Knoten-Attribut für diesen Indikator auf `false` gesetzt.

- Erfüllt ein Artefakt alle zuvor genannten Bedingungen, werden seine Methoden auf Aufrufe von außerhalb der Vererbungshierarchie überprüft.
  - um zu bestimmen, ob ein Aufruf von außerhalb oder innerhalb der Vererbungshierarchie kommt, werden die aufrufenden Klassen ermittelt und daraufhin überprüft, ob sie zu den direkten und indirekten Unterklassen der Klasse bzw. des Interfaces gehören.
- Sofern keine entsprechenden Aufrufe existieren, wird das Knoten-Attribut auf `true` gesetzt.

3.3.16 Identitätsspaltung

Die Umsetzung dieses Qualitätsindikators basiert auf der Programmrepräsentationsform RFG.

Die Ergebnisse werden durch das Attribut `Metric.Identity_Splitting` vom Typ `string`, welches an Klassen- und Interface-Knoten geschrieben wird, im RFG vermerkt.

Alle Klassen und Interfaces werden nach Namen (unabhängig von Groß- und Kleinschreibung) gruppiert. Jede Gruppe erhält eine *ID*, welche an jedes Mitglied der Gruppe annotiert wird. Auf diese Weise können alle Gruppenmitglieder in der Detailliste aufgelistet werden.
Die Gesamtzahl der Verletzungen ergibt sich aus der Anzahl der Gruppen.

3.3.17 Importchaos

Die Umsetzung dieses Qualitätsindikators basiert nicht auf einer Programmrepräsentationsform von Bauhaus.

Für diesen Indikator wird kein Attribut in den RFG geschrieben.

Da Imports und Includes weder in der IML noch im RFG repräsentiert werden, kann dieser Indikator nur auf Basis des Quellcodes ermittelt werden und funktioniert deshalb auch nur, wenn die entsprechenden Dateien im RFG angegebenen Pfad vorhanden sind.
Unabhängig von der zugrundeliegenden Sprache wird in allen im RFG erwähnten Quellcode-Dateien nach include- bzw. import-Anweisungen gesucht.
Bei *C++*-Systemen müssen diese Anweisungen nur miteinander verglichen werden. Wird die gleiche Datei mehrfach inkludiert, liegt ein Treffer vor. Folglich wird der Zähler erhöht und der fragliche Import unter Angabe von Datei und Zeile, in der er gefunden wurde, in die Trefferliste aufgenommen.
Für *Java*-Systeme wird zusätzlich jeder Import auf `java.lang` und `*` überprüft. Um das Importieren einer Klasse aus dem eigenen Paket zu ermitteln, wird wie folgt vorgegangen:
Beim Ermitteln der Import-Anweisungen wird auch eine eventuelle Package-Zugehörigkeit

ermittelt. Wenn das Paket, dem die Datei angehört, in einer Import Anweisung auftritt, wird überprüft ob dem mehr als ein „.“ folgt. Ist dies nicht der Fall, handelt es sich um den Import einer Klasse aus dem eigenen Paket.

3.3.18 Importlüge

Die Umsetzung dieses Qualitätsindikators basiert auf der Programmrepräsentationsform RFG.

Die Ergebnisse werden durch das Attribut `Metric.Import_Lie` vom Typ `string`, welches an Module-Knoten geschrieben wird, im RFG vermerkt.

Da weder in der IML noch im RFG importierte bzw. inkludierte Artefakte ausgewiesen werden, kann dieser Indikator nur mit gewissen Einschränkungen implementiert werden.
Weil für *Java* keine File-View generiert wird, erfolgt die Implementierung gemeinsam mit der Auswertung. Sie folgt jedoch der im Folgenden vorgestellten Implementierung für *C++*.
Es werden sämtliche im RFG enthaltenen (bzw. im Fall von *Java* „erwähnten“) Dateien durchlaufen. Sofern sie in den im RFG angegebenen Pfaden liegen, werden die im Quellcode stehenden Include- bzw. Import-Anweisungen nach dem Vorbild von *Importchaos* extrahiert. Für jedes Include wird nun im RFG nach den dort deklarierten Inhalten gesucht und anschließend geprüft, ob die Inhalte der aktuell überprüften Datei diese verwenden. Ist dies nicht der Fall, liegt eine `Importlüge` vor.
Die Ermittlung der Inhalte sowohl der aktuellen Datei als auch des Includes erfolgt für *C++* über die `CQI-FILE`-View. Da diese für *Java* nicht existiert, werden hier alle Knoten der `BASE`-View durchlaufen und ihre Attribute `Source.Path`, `Source.File` und `Source.Name` entsprechend überprüft.
Viele Inhalte insbesondere von Standardincludes für *C++* (wie zum Beispiel `#include <iostream>`) werden nicht in diesen direkt implementiert sondern sind ihrerseits inkludiert (beim genannten Beispiel ist das etwa `cout`). Weil die rekursive Bestimmung, der inkludierten Dateien und Inhalte von inkludierten Dateien, sehr aufwändig und die Gefahr von false-positives sehr hoch wäre, wird bewusst eine Sicht verwendet, in der diese Includes ausgefiltert sind. Dadurch können nur Importlügen erkannt werden, die Includes innerhalb des untersuchten Systems betreffen.

3.3.19 informelle Dokumentation

Die Umsetzung dieses Qualitätsindikators basiert auf der Programmrepräsentationsform RFG.

Die Ergebnisse werden durch das Attribut `Metric.Informal_Documentation` vom Typ `toggle`, welches an Method-Knoten geschrieben wird, im RFG vermerkt.

Es wird zu jeder im RFG enthaltenen öffentlichen Methode die Codeposition ihrer Deklaration ermittelt. Diese ergibt sich aus den Attributen `Source.File`, `Source.Path` und `Source.Line`. Sofern die Datei im angegeben Pfad gefunden wird, werden die der Methodendefinition vorangestellten Zeilen auf einen formalen Kommentar gemäß Indikatordefinition hin untersucht. Das Vorhandensein eines solchen Kommentars wird entgegen der Definition durch die Beendigung desselben mit `*/` angenommen und nicht durch seine Einleitung mit `/*`. Die Abweichung hat den einfachen Grund, dass das Ende eines solchen Kommentars unmittelbar vor einer Methodendeklaration leichter zu erkennen ist, als der Beginn eines Solchen, der möglicherweise viele Zeilen höher liegt. Da sowohl in *C++* als auch in *Java* Kommentare, die mit `/*` eingeleitet werden in jedem Fall mit `*/` beendet werden müssen, wird das Analyse-Ergebnis durch diese Abweichung nicht verändert.

3.3.20 Interface-Bypass

Die Umsetzung dieses Qualitätsindikators basiert auf der Programmrepräsentationsform RFG.

Die Ergebnisse werden durch das Attribut `Metric.IF_BYPASS` vom Typ `toggle`, welches an Call-Kanten geschrieben wird, im RFG vermerkt.

Die zu zählenden Methodenpaare aus aufrufender und aufgerufener Methode werden auf Basis der `Call`-Kanten ermittelt. Diese zeigen jeweils von der aufrufenden Methode auf die aufgerufene. Sofern die, die aufgerufene Methode umschließende Klasse keine Abstraktion ist, wird geprüft, ob die Methode bereits in einer Abstraktion deklariert wurde.
Ist dies der Fall, muss noch ausgeschlossen werden, dass die Klasse der aufrufenden Methode mit der Abstraktion, in der die aufgerufene Methode deklariert ist, in einer Vererbungsbeziehung steht.
Wenn alle Bedingungen erfüllt sind, stellt der Aufruf einen Interface-Bypass dar und die `Call`-Kante wird entsprechend markiert.

3.3.21 Klässchen

Die Umsetzung dieses Qualitätsindikators basiert auf der Programmrepräsentationsform RFG.

Die Ergebnisse werden durch das Attribut `Metric.ATTRIBUTES und Metric.METHODS` vom Typ `int`, welches an Klassen- und Interface-Knoten geschrieben wird, im RFG vermerkt.

Der Qualitätsindikator *Klässchen* verfügt ausschließlich über eine Auswertungskomponente. Diese stützt sich auf die für *Gottklasse (Attribut)* und *Gottklasse (Methode)* implementierten Metriken.
Eine gesonderte Markierung von *Klässchen* im RFG erfolgt nicht.

3.3.22 Klasseninzest

Die Umsetzung dieses Qualitätsindikators basiert auf der Programmrepräsentationsform RFG.

Die Ergebnisse werden durch das Attribut `Metric.Class_Incest` vom Typ `string`, welches an Klassen- und Interface-Knoten geschrieben wird, im RFG vermerkt.

Ursprünglich wurde dieser Indikator auf Basis der IML implementiert. Diese Lösung stellte sich jedoch als sehr komplex und dadurch fehleranfällig heraus. Hinzu kamen sehr hohe Laufzeiten. Nachdem der RFG in neueren Bauhaus-Versionen um die Kantentypen `Return_Type` , `Parameter_of_Type`, `Local_Var_Of_Type` und `Of_Type` erweitert wurde und eine Referenzierung der Unterklasse damit im RFG erkennbar ist, wurde dieser Indikator auf Basis des RFG erneut implementiert, wodurch neben dem deutlich unkomplizierterem Vorgehen vor allem eine deutliche Verkürzung der Laufzeit erreicht wurde.
Die RFG-basierte Implementierung arbeitet für jede Klasse bzw. jedes Interface die folgenden Schritte ab:

- Ermittlung aller direkten und indirekten Unterklassen/-Interfaces
- Ermittlung der von der Klasse umschlossenen Artefakte
- Ermittlung aller von den zuvor ermittelten Elementen referenzierten Artefakte
- Für jede Unterklasse/-Interface prüfen, ob die Klasse selbst oder eines der von ihr umschlossenen Artefakte in der Menge der referenzierten Artefakte enthalten ist.
- Jede so als referenziert erkannte Unterklasse wird unter Angabe von `Source.Name`, `Source.Path` + `Source.File` und `Source.Line` (jeweils durch zwei Doppelpunkte getrennt) in das Attribut der Oberklasse geschrieben. Sollten mehrere Unterklassen referenziert werden, werden die einzelnen Datensätze durch ein Semikolon getrennt.

3.3.23 Konstantenregen

Die Umsetzung dieses Qualitätsindikators basiert auf der Programmrepräsentationsform RFG.

Die Ergebnisse werden durch das Attribut `Metric.CON` vom Typ `int`, welches an Constant- und Member-Knoten geschrieben wird, im RFG vermerkt.

Die Implementierung folgt den Vorgaben des Buches Code-Quality-Management [SSM06]. Den einzelnen Gruppen namensgleicher Konstanten werden – wie schon bei *Identitätsspaltung* – *IDs* zugeordnet und alle Gruppenmitglieder entsprechend markiert.

3.3.24 Labyrinthmethode

Die Umsetzung dieses Qualitätsindikators basiert auf der Programmrepräsentationsform RFG.

Die Ergebnisse werden durch das Attribut `Metric.LAB` vom Typ `toggle`, welches an Method-Knoten geschrieben wird, im RFG vermerkt.

Alle Methoden mit einer – von Bauhaus berechneten – McCabe-Komplexität > 10 werden als Labyrinthmethode markiert.

3.3.25 lange Parameterliste

Die Umsetzung dieses Qualitätsindikators basiert auf der Programmrepräsentationsform RFG.

Die Ergebnisse werden durch das Attribut `Metric.Number_Of_Parameters` vom Typ `int`, welches an Method-Knoten geschrieben wird, im RFG vermerkt.

Es existiert ausschließlich eine Auswertungskomponente, weil die Anzahl der Parameter einer Methode durch die Bauhaus-Suite bestimmt und im RFG eingetragen werden.
Ist der Wert dieser Metrik für eine Methode > 7, wird ein Zähler erhöht und die Methode (inkl. Code-Position) in die Detailliste aufgenommen.

3.3.26 maskierende Datei

Die Umsetzung dieses Qualitätsindikators basiert auf der Programmrepräsentationsform RFG.

Die Ergebnisse werden durch das Attribut `Metric.MF` vom Typ `toggle`, welches an Module-Knoten geschrieben wird, im RFG vermerkt.

Entgegen der Buchempfehlung werden nicht nur die Klassen mit der höchsten Sichtbarkeit ermittelt. Da zum Bestimmen der höchsten Sichtbarkeit die Sichtbarkeit aller in einer Datei enthaltenen Klassen ermittelt werden muss, werden der Einfachheit halber alle Klassen ermittelt und nach ihrer Sichtbarkeit gruppiert. Anschließend werden die Namen der Klassen in der Gruppe mit der höchsten Sichtbarkeit ermittelt und geprüft, ob mindestens einer davon den Namen der Datei vollständig enthält.
Ist dies nicht der Fall gilt die Datei als „maskierende Datei“ und wird entsprechend markiert.

Weil für Java keine `File`-View generiert wird, ist es für Java nicht möglich maskierende Dateien im RFG zu markieren. Deshalb wird das oben beschriebene Vorgehen in die Auswertung verlagert. Die Markierung eines positiven Treffers entfällt in diesem Fall. Stattdessen wird sofort der Zähler erhöht und die Datei in die Detailliste aufgenommen.

3.3.27 nachlässige Kommentierung

Die Umsetzung dieses Qualitätsindikators basiert auf der Programmrepräsentationsform RFG.

Für diesen Indikator wird kein Attribut in den RFG geschrieben.

Für diesen Indikator existiert ausschließlich eine Auswertungskomponente. Diese stützt sich auf die – durch das von Axivion entwickelte Plug-In berechneten – Metriken `Metric.Lines.Loc` und `Metric.Lines.Only_Comment` .
Für *C++* wird die Gesamtzahl an LOC und Kommentarzeilen auf Basis der Dateien berechnet. Da diese Sicht für *Java* nicht generiert wird, stützt sich die Auswertung für *Java* auf Pakete.
Entsprechend unterscheidet sich auch die Detailliste, die für *C++* die Einzel-Wertungen für Datein enthält und für *Java* die Einzel-Wertungen für Pakete.
Die eigentliche Berechnung erfolgt in jedem Fall nach der folgenden Formel:

$$abs(BLOC - 2 \times CLOC)$$

3.3.28 Namensfehler

Die Umsetzung dieses Qualitätsindikators basiert auf der Programmrepräsentationsform RFG.

Die Ergebnisse werden durch das Attribut `Metric.VIOLATION_OF_NAMING_CONVENTIONS` vom Typ `toggle`, welches an diverse Artefakte geschrieben wird, im RFG vermerkt.

Die Implementierung entspricht exakt der Buchempfehlung (vgl. [SSM06, Seite 252]).

3.3.29 Objektplacebo (Attribut)

Die Umsetzung dieses Qualitätsindikators basiert auf der Programmrepräsentationsform IML.

Die Ergebnisse werden durch das Attribut `Metric.Objectplacebo_Attribute` vom Typ `int`, welches an Member-Knoten geschrieben wird, im RFG vermerkt.

Die Implementierung dieses Indikators stellt, im Vergleich zu den bisher vorgestellten, in mehrfacher Hinsicht eine Besonderheit dar und wird deshalb exemplarisch detallierter vorgestellt. Zunächst einmal dient in diesem Fall die IML nicht nur zum Überprüfen eines bestimmten Sachverhalts sondern als Ausgangspunkt der Analyse; während der RFG tatsächlich ausschließlich zum Annotieren des Ergebnisses verwendet wird. Durch das zu diesem Indikator gehörende Knotenattribut wird die Anzahl der „bösen“ Zugriffe auf einen bestimmten

Klassen-Member an selbigen annotiert. Da jedoch Member-Knoten nicht vom Skript `iml2rfg` erfasst werden, wurde auch für die Übersetzung von IML nach RFG eine individuelle Lösung entworfen.
Da die Erkennung dieses Problemmusters auf Basis der IML relativ leicht und unkompliziert ist, dient dieser Indikator gleichzeitig als exemplarisches Beispiel für die Extraktion von Informationen aus der IML.

Ausgangspunkt für die Analyse bilden die Zugriffstellen auf jede Art von Variable und Konstante. In der IML sind das die Knotentypen `Read` und `Assignment`. Für diese muss nun jeweils überprüft werden, ob es sich um einen Zugriff auf ein statisches Attribut (`Static_Field_Selection`) handelt. Hierzu wird der Typ des Feldes `Operand` (`Read`) bzw. `Target` (`Assignment`) abgefragt.
Handelt es sich um einen Zugriff auf ein statisches Attribut muss als nächstes geprüft werden, ob der Zugriff über ein Objekt oder eine Klasse erfolgt. Der Zugriff über ein Objekt ist dann gegeben, wenn das Feld `Name` des `Static_Field_Selection`-Knotens zu einem Knoten vom Typ `O_Static_Field` führt.
In diesem Fall wird unter Verwendung des *Mangled_Name* und des *Sloc*-Eintrags der korrespondierende RFG-Knoten ermittelt. Dazu wird das Attribut `Linkage.Name` aller `Member`-Knoten im RFG daraufhin überprüft, ob die einzelnen Komponenten des *Mangled_Name* in ihm enthalten sind. Zusätzlich werden die Angaben des *Sloc*-Eintrags (Datei, Zeile, Spalte) mit den Angaben des jeweiligen Knoten verglichen. Wenn alle Werte übereinstimmen, ist der richtige Knoten gefunden. Am so gefundenen Knoten wird nun der Wert des Attributs `Metric.Objectplacebo_Attribute` um eins erhöht.
Der Pfad durch den Graph der IML ist in Abbildung 3.4 noch einmal anschaulich dargestellt.

3.3.30 Objektplacebo (Methode)

Die Umsetzung dieses Qualitätsindikators basiert auf der Programmrepräsentationsform IML.

Die Ergebnisse werden durch das Attribut `Metric.Objectplacebo_Method` vom Typ `int`, welches an Method-Knoten geschrieben wird, im RFG vermerkt.

Dieser Indikator nutzt – wie der vorherige – die IML als Ausgangspunkt der Analyse. Die Indikatordefinition kann allerdings nur eingeschränkt umgesetzt werden. Weil es nicht möglich ist, den Aufruf über eine Klasse vom Aufruf über eine Instanz der Klasse zu unterscheiden (beide werden identisch modelliert), können nur Aufrufe von statischen Methoden über Zeiger-Objekte, die per Definition eine Verletzung darstellen, erkannt werden.
Um diese zu erkennen werden alle `Indirect_Call`s daraufhin überprüft, ob sie eine statische Methode aufrufen.
Die Ermittlung der Methoden-Knoten im RFG zwecks Annotierung des unerwünschten Auf-

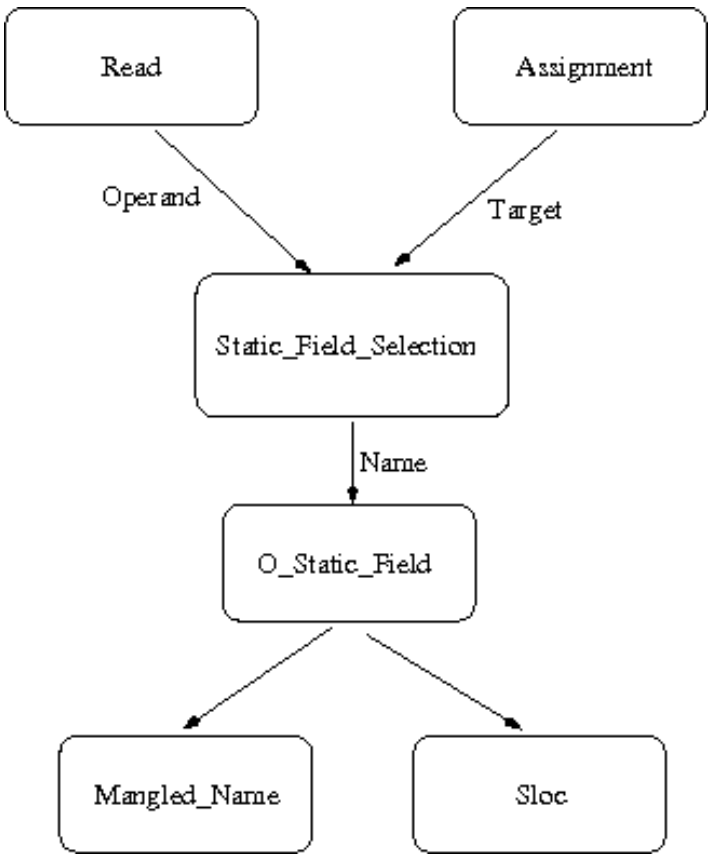


Abbildung 3.4: IML-Pfad

rufs erfolgt unter Verwendung des Skripts `iml2rfg` (nähere Informationen dazu sind der entsprechenden Dokumentation [AXI07b] zu entnehmen).

3.3.31 Paketchen

Die Umsetzung dieses Qualitätsindikators basiert auf der Programmrepräsentationsform RFG.

Die Ergebnisse werden durch das Attribut `Metric.CLASSES` vom Typ `int`, welches an Paket-Knoten (Knotentyp hängt von der analysierten Sprache und der verwendeten Paketdefinition ab) geschrieben wird, im RFG vermerkt.

Es wird die Anzahl der öffentlichen, nicht geschachtelten Klassen, die in einem `Package` (Java) bzw. `Verzeichnis` oder `Namespace` (C++) bestimmt. Die Bewertung erfolgt erst im Zuge der Auswertung.

3.3.32 Pakethierarchieaufbruch

Die Umsetzung dieses Qualitätsindikators basiert auf der Programmrepräsentationsform RFG.

Die Ergebnisse werden durch das Attribut `Metric.PACKAGE_BREAKUP` vom Typ `toggle`, welches an Extends- und Implementation_Of-Kanten geschrieben wird, im RFG vermerkt.

Die Implementierung erfolgt gemäß den Hinweisen des Buches (vgl. [SSM06, Seite 263]).

3.3.33 Polymorphieplacebo

Die Umsetzung dieses Qualitätsindikators basiert auf der Programmrepräsentationsform IML.

Die Ergebnisse werden durch das Attribut `Metric.Polymorphy_Placebo` vom Typ `string`, welches an Mehod-Knoten geschrieben wird, im RFG vermerkt.

Für jede Oberklasse werden zunächst im RFG alle statischen Methoden ermittelt und nach Namen gruppiert, die aufgrund ihrer Sichtbarkeit in einer Unterklasse sichtbar sind.
Anschließend werden alle nichtgeerbten Methoden der Unterklassen daraufhin untersucht, ob sie namensgleich mit einer Oberklassen-Methode sind. Ist dies der Fall werden über die IML die Signaturen der fraglichen Methoden (Unterklassen-Methode und Oberklassen-Methode(n)) bestimmt. Dies muss auf Basis der IML geschehen, weil im RFG zwar die Anzahl der Parameter aber nicht ihr Typ und ihre Reihenfolge vermerkt sind. Nachdem die Signaturen ermittelt sind wird untersucht, ob die Signatur der Unterklassen-Methode bereits in der Oberklasse vorhanden ist und somit ein Polymorphieplacebo begründet.
Im Fall eines positiven Befundes werden die folgenden Daten der überdeckten Methode ausgelesen und an ihr durch je zwei Doppelpunkte getrennt annotiert: der Name der Methode, der Name der die Methode umschließenden Klasse sowie Datei und Zeilennummer.

3.3.34 potenzielle Privatsphäre (Attribut)

Die Umsetzung dieses Qualitätsindikators basiert auf der Programmrepräsentationsform RFG.

Die Ergebnisse werden durch das Attribut `Metric.POTENTIAL_PRIVACY` vom Typ `toggle`, welches an Member-Knoten geschrieben wird, im RFG vermerkt.

Die Implementierung dieses Qualitätsindikators wurde bereits in der Einleitung zu Abschnitt 3.3 beschrieben. Eine gesonderte Beschreibung erfolgt deshalb an dieser Stelle nicht.

3.3.35 potenzielle Privatsphäre (Methode)

Die Umsetzung dieses Qualitätsindikators basiert auf der Programmrepräsentationsform RFG.

Die Ergebnisse werden durch das Attribut `Metric.POTENTIAL_PRIVACY` vom Typ `toggle`,

welches an Method-Knoten geschrieben wird, im RFG vermerkt.

Entgegen der Buch-Empfehlung wird zuerst überprüft, ob die Methoden selbst eine `protected`-Methode überschreiben oder implementieren und im Anschluß, ob sie verwendet oder selbst überschrieben werden.
Dieses Vorgehen bietet sich deshalb an, weil auf diese Weise eine Betrachtung der Aufrufe der Methode unter Umständen vermieden werden kann.

3.3.36 pränatale Kommunikation

Die Umsetzung dieses Qualitätsindikators basiert auf der Programmrepräsentationsform RFG.

Die Ergebnisse werden durch das Attribut `Metric.PRENATAL_COMMUNICATION` vom Typ `toggle`, welches an Method-Knoten geschrieben wird, im RFG vermerkt.

Für jede Methode mit gesetztem Attribut `Element.Is_Constructor` werden die ausgehenden Kanten vom Typ `Direct_Call` daraufhin überprüft, ob die auf eine virtuelle Methode – in der eigenen Klasse – verweisen.

3.3.37 Risikocode

Die Umsetzung dieses Qualitätsindikators basiert auf der Programmrepräsentationsform IML.

Die Ergebnisse werden durch das Attribut `Metric.RiskCode.missing_default, Metric.RiskCode.missing_break und Metric.RiskCode.empty_exception_handler` vom Typ `int`, welches an Routine- und Method-Knoten geschrieben wird, im RFG vermerkt.

Dieser Indikator ist in zwei Phasen unterteilt. Zunächst werden alle in der IML befindlichen Switch-Anweisungen (Typ: `C_Switch_Statement`) daraufhin untersucht, ob sie einen *default*-Pfad haben und die *break*-Anweisungen existieren.
Anschließend werden alle Catch-Blöcke (Typ: `Catch_Block`) daraufhin untersucht, ob sie keine Anweisungen enthalten.
An beide Phasen schließt sich bei einem Befund jeweils unmittelbar die Bestimmung der umschließenden Methode sowie ihres korrespondierenden RFG-Knotens an, um an diesem die Anzahl der gefundenen Verletzungen zu annotieren.
Um das Vorhandensein eines *default*-Pfades nachzuweisen, werden die `Case_Labels` daraufhin untersucht, ob sie einen Knoten vom Typ `Anonymous_Label` enthalten.
Das Fehlen von *break*-Anweisungen liegt vor, wenn es ein Delta zwischen der Anzahl der Case-Zweige und der Anzahl der Knoten vom Typ `Exit_Switch` gibt.
Ein leerer Catch-Block besteht aus zwei Knoten, einem vom Typ `Initialize` und einem vom

Typ `Null_Expression`. Leider ist es nicht möglich zu bestimmen, ob der Catch-Block möglicherweise Kommentarzeilen enthält. In diesem Punkt weicht die Implementierung deshalb von der Definition des Indikators ab.

3.3.38 signaturähnliche Klassen

Die Umsetzung dieses Qualitätsindikators basiert auf der Programmrepräsentationsform IML.

Die Ergebnisse werden durch das Attribut `Metric.Similar_Signatures` vom Typ `string`, welches an Class-Knoten geschrieben wird, im RFG vermerkt.

Zunächst werden die Methoden-Signaturen aller Klassen ermittelt und in einem Array gespeichert, welches wiederum zusammen mit der jeweiligen Klasse als Tupel in einem Array gespeichert wird. Es ergibt sich folgendes Format:

```
[(class, [signatures]), (class, [signatures]), ...]
```

Um nun die signaturähnlichen Klassen-Paare zu ermitteln, wird jeweils das erste Tupel aus dem Array entfernt und seine Signaturen mit den Signaturen aller anderen noch im Array befindlichen Klassen verglichen. Vor den einzelnen Vergleichen wird noch sichergestellt, dass die beiden Klassen weder miteinander verwandt sind noch gemeinsame Oberklassen haben. Jedem so gefundenen signaturähnlichen Klassen-Paar wird eine eindeutige ID zugeordnet, welche über das Knotenattribut annotiert wird. Ist eine Klasse mit mehreren Klassen signaturähnlich, werden die einzelnen IDs durch Kommata getrennt.

3.3.39 simulierte Polymorphie

Die Umsetzung dieses Qualitätsindikators basiert auf der Programmrepräsentationsform IML.

Die Ergebnisse werden durch das Attribut `Metric.Simulated_Polymorphy` vom Typ `toggle`, welches an Routine- und Method-Knoten geschrieben wird, im RFG vermerkt.

Dieser Indikator ist einer der wenigen, die für *Java* und *C++* völlig unterschiedliche Implementierungen verlangen. Die Beschreibung erfolgt deshalb strikt getrennt.

3.3.39.1 Java

Ausgangspunkt der Analyse ist die IML. Zunächst werden alle Knoten vom Typ `Instanceof_Operator` nach den sie umschließenden Methoden gruppiert. Anschließend werden für jede Methode die Typüberprüfungen nach dem überprüften Objekt gruppiert. Enthält eine dieser Gruppen mehr als einen Typ auf den das Objekt überprüft wird, liegt ein Treffer vor.

Es wird nun unter Verwendung von `iml2rfg` der RFG-Knoten der Methode ermittelt und dieser entsprechend markiert.

3.3.39.2 C++

Für *C++* ist dieser Indikator nur mit einigen Einschränkungen implementierbar. Wie schon bei *Objektplacebo (Methode)* können Instanzen und Typen nicht unterschieden werden. Dadurch können ausschließlich mehrfache Typüberprüfungen von Pointer-Objekten erkannt werden. Zusätzlich kann in einem Konstrukt wie „`if typeid(pointer-object)== typeid(type ) or typeid(instanz)== typeid(type)`“ nicht erkannt werden, dass eigentlich zwei verschiedene Objekte auf jeweils einen Typ hin überprüft werden. Dadurch kann es zu false-positives kommen. Insbesondere der letzte Punkt sollte bei der Interpretation der Ergebnisse im Hinterkopf behalten werden.

Auch für *C++* dient die IML als Ausgangspunkt der Analyse. Hier werden zunächst alle Knoten vom Typ `Type_Typeid` nach den sie umschließenden Methoden gruppiert.
In den Methoden, die mehr als zwei `Type_Typeid`-Knoten enthalten, werden diese nun noch nach den umschließenden If-Anweisungen gruppiert. Als Erkennungsmerkmal werden dabei die SLoc-Werte der If-Anweisungen verwendet. Hintergrund dieser Gruppierung ist der, dass `typeid` lediglich den Typ des übergebenen Objekts zurückliefert. Um ein Objekt auf einen Typ hin zu überprüfen, müsste ein Konstrukt wie „`if typeid(object)== typeid(type)`“ verwendet werden.
Um false-positives möglichst zu vermeiden, werden nun nur diejenigen if-Anweisungen weiter betrachtet, die ein Pointer-Objekt auf einen Typ hin überprüfen.
Die überprüften Objekte werden in einer Liste gesammelt und der Index an dem sie in der Liste stehen als Dictionary-Schlüssel verwendet, hinter dem in einer weiteren Liste die Typen – auf die das Objekt überprüft wird – gespeichert werden.
Beinhaltet eine dieser Listen am Ende mehr als einen Typ, wird der zur aktuellen Methode korrespondierende RFG-Knoten ermittelt und entsprechend markiert.

3.3.40 späte Abstraktion

Die Umsetzung dieses Qualitätsindikators basiert auf der Programmrepräsentationsform RFG.

Die Ergebnisse werden durch das Attribut `Metric.LATE_ABSTRACTION` vom Typ `string`, welches an Class-Knoten geschrieben wird, im RFG vermerkt.

Die Implementierung hält sich an die Empfehlung des Buches (vgl. [SSM06, Seite 290]).

3.3.41 tote Attribute

Die Umsetzung dieses Qualitätsindikators basiert auf der Programmrepräsentationsform RFG.

Die Ergebnisse werden durch das Attribut `Metric.DEAD_ATTRIBUTE` vom Typ `toggle`, welches an Member-Knoten geschrieben wird, im RFG vermerkt.

Es werden alle Member-Knoten durchlaufen. Alle, die `private` sind und keine eingehenden Zugriffskanten (egal ob lesend oder schreibend) haben, werden als tot markiert.

3.3.42 tote Implementierung

Die Umsetzung dieses Qualitätsindikators basiert auf der Programmrepräsentationsform IML.

Die Ergebnisse werden durch das Attribut `Metric.Dead_Implementations` vom Typ `int`, welches an Routine- und Method-Knoten geschrieben wird, im RFG vermerkt.

Es wird von `Statement_Sequence`-Knoten ausgehend nach Anweisungen gesucht, die unmittelbar auf einen Knoten vom Typ `End_Of_Lifetime` folgt.
Ist eine solche Folge gefunden, wird die Methode, die die Anweisungen enthält ermittelt und der Wert des Attributs im RFG entsprechend erhöht.

3.3.43 tote Methoden

Die Umsetzung dieses Qualitätsindikators basiert auf der Programmrepräsentationsform RFG.

Die Ergebnisse werden durch das Attribut `Metric.DEAD_ATTRIBUTE` vom Typ `toggle`, welches an Method-Knoten geschrieben wird, im RFG vermerkt.

Die Implementierung entspricht den Empfehlungen des Buches (vgl. [SSM06, Seite 299]).

3.3.44 überbuchte Datei

Die Umsetzung dieses Qualitätsindikators basiert auf der Programmrepräsentationsform RFG.

Die Ergebnisse werden durch das Attribut `Metric.OVERBOOKED_FILE` vom Typ `toggle`, welches an Module-Knoten geschrieben wird, im RFG vermerkt.

Da für *Java* keine `File`-View generiert wird, existiert hier nur eine auswertende Komponente. Die Annotierung im RFG entfällt deshalb für *Java*.

Für *C++* werden zu jeder Datei die enthaltenen Klassen mit Sichtbarkeit `public` oder `default` bestimmt. Ist das mehr als eine, wird die Datei entsprechend markiert.

Für *Java* werden alle Klassen durchlaufen und – sofern sie die Sichtbarkeit `public` oder `default` haben – anhand der sie umschließenden Dateien gruppiert. Enthält eine solche Gruppe mehr als eine Klasse, wird der Zähler erhöht und die entsprechende Datei in die Detailliste aufgenommen.

3.3.45 unfertiger Code

Die Umsetzung dieses Qualitätsindikators basiert auf der Programmrepräsentationsform RFG.

Die Ergebnisse werden durch das Attribut `Metric.Lines.Fixme, Metric.Lines.Hack und Metric.Lines.Todo` vom Typ `int`, welches an Module- (C++) bzw. Package-Knoten (Java) geschrieben wird, im RFG vermerkt.

Dieser Indikator stüzt sich auf Metriken, die durch das Plug-In von Axivion berechnet werden. Entsprechend existiert lediglich eine Auswertungskomponente.
Da auch Kommentare außerhalb von Klassen mitgezählt werden müssen, werden für *C++* Dateien als Grundlage verwendet. Da Dateien in *Java*-basierten RFGs keine Repräsentation erfahren, werden hier ersatzweise Package-Knoten verwendet.
Die Zählung beruht in beiden Fällen auf Zeilen, die die Schlüsselwörter enthalten und nicht – wie in der Definition gefordert – auf Kommentar-Blöcken. Dies kann dazu führen, dass ein Kommentar-Block, der sowohl „todo“ als auch „tofix“ enthält, doppelt als Verletzung gezählt wird, ist aber aufgrund der Tatsache, dass Kommentare keinen Eingang in die Programmrepräsentation finden, nicht zu vermeiden.

3.3.46 unvollständige Vererbung (Attribut)

Die Umsetzung dieses Qualitätsindikators basiert auf der Programmrepräsentationsform RFG.

Die Ergebnisse werden durch das Attribut `Metric.INCOMPLETE_INHERITANCE` vom Typ `string`, welches an Class-Knoten geschrieben wird, im RFG vermerkt.

Für jede Oberklasse werden die Namen der nichtstatischen Attribute und die direkten Unterklassen ermittelt. Die eigentliche Analyse wird nun für jede Unterklasse einzeln durchgeführt. Es werden jeweils alle Attribute (auch geerbte) ermittelt und nach Namen gruppiert. Sind in einer Gruppe mehr Attribute enthalten als in der Oberklasse existierten, wird überprüft, ob die überzähligen Attribute aus weiteren (in)direkten Oberklassen stammen. Wenn dies nicht der Fall ist, handelt es sich um einen Treffer im Sinne von *unvollständige Vererbung (Attribut)*. Die fragliche Unterklasse wird deshalb vorerst in einem `Node_Set` gespeichert.

Nachdem die oben genannten Schritte für alle direkten Unterklassen durchgeführt wurden, wird überprüft, ob mindestens 10 oder mehr als die Hälfte der direkten Unterklassen Eingang in das erwähnte `Node_Set` gefunden haben. Ist dies der Fall, werden zu den einzelnen Klassen Datensätze bestehend aus dem Namen der Klasse, ihrer Datei und der im RFG angegebenen Zeilennummer generiert. Die einzelnen Werte sind jeweils durch Kommata getrennt.
Die so generierten Datensätze werden – durch je zwei Doppelpunkte getrennt – in das Attribut der Oberklasse geschrieben.

3.3.47 unvollständige Vererbung (Methode)

Die Umsetzung dieses Qualitätsindikators basiert auf der Programmrepräsentationsform IML.

Die Ergebnisse werden durch das Attribut `Metric.INCOMPLETE_INHERITANCE_METHOD_COUNT und Metric.INCOMPLETE_INHERITANCE_METHOD_VIOLATIONS` vom Typ `int bzw. string`, welches an Class- bzw. Interface-Knoten geschrieben wird, im RFG vermerkt.

Die Implementierung dieses Indikators entspricht in ihrer Vorgehensweise weitgehend der Implementierung des Qualitätsindikators *unvollständige Vererbung (Attribut)*. Da hier jedoch statt Attributnamen Methodensignaturen Untersuchungsgegenstand sind, muss die Implementierung in Teilen IML-basiert erfolgen, weil nur über die IML die Methodensignaturen (inkl. Parametertypen) bestimmt werden können.

3.3.48 verbotene Dateiliebe

Die Umsetzung dieses Qualitätsindikators basiert auf der Programmrepräsentationsform RFG.

Die Ergebnisse werden durch das Attribut `Metric.CYCLE_ID` vom Typ `string`, welches an Module- (C++) bzw. Type-Knoten (Java) geschrieben wird, im RFG vermerkt.

Die Implementierung dieses Indikators stützt sich in sehr hohem Maße auf Spezialitäten der Bauhaus-Suite und wird deshalb etwas ausführlicher erläutert. Da wegen der für *Java* nicht generierten `Module`-Knoten die Implementierung für die beiden Sprachen deutlich voneinander abweicht, werden sie getrennt vorgestellt.

3.3.48.1 C++

Auf Basis der im Vorfeld generierten Sicht `CQI-LIFTEDMODULE` wird die in Bauhaus integrierte Zyklen-Berechnung ausgeführt. Auf der dadurch generierten Sicht werden alle Knoten vom Typ `Module` im Hinblick auf ihre Beteiligung an einem direkten Zyklus zwischen zwei Dateien betrachtet. Zu diesem Zweck werden die Vorgänger- und die Nachfolger-Knoten ermittelt

(die Ermittlung beschränkt sich auf `Module`-Knoten). Die Schnittmenge dieser beiden Mengen ist die Gruppe der „Liebhaber“. Jedes Mitglied dieser Gruppe stellt zusammen mit der ursprünglichen Datei ein Paar gemäß der Indikatordefinition dar.
Es wird nun jedem der so ermittelten Dateipaare eine eindeutige ID zugewiesen und in das Attribut beider Dateien geschrieben. Ist eine Datei in mehrere direkte Zyklen involviert, werden die einzelnen IDs durch Semikolon getrennt.

3.3.48.2 Java

Da für *Java* die `MODULE`-View fehlt, muss hier ein anderer Weg beschritten werden. Auf Basis der Sicht `CQI-ANALYSIS` werden zunächst die Kanten „gelifted“. Das bedeutet, das alle Kanten durch die Hierarchie hindurch nach oben propagiert werden. Die Kante eines Methodenaufrufs zwischen zwei – in verschiedenen Klassen enthaltenen – Methoden verbände somit nicht mehr nur die betroffenen Methoden, sondern auch die umschließenden Klassen und ggfs. die die Klassen umschließenden Pakete.
Auf Basis der so generierten Sicht wird nun die in Bauhaus integrierte Zyklen-Berechnung ausgeführt, die eine neue Sicht generiert, auf der die eigentliche Analyse durchgeführt wird.

Es werden nun analog zur *C++*-Implementierung `Type`-Paare (`Type` vereint Klassen- und Interface-Knoten unter einem Oberbegriff) ermittelt. Für jedes dieser Paare wird im Anschluß überprüft, ob die Artefakte in verschiedenen Dateien enthalten sind, die sich somit gegenseitig über ihre Inhalte referenzieren würden.
Die Annotierung der Ergebnisse erfolgt dann wieder analog zur *C++*-Implementierung.

3.3.49 verbotene Klassenliebe

Die Umsetzung dieses Qualitätsindikators basiert auf der Programmrepräsentationsform RFG.

Die Ergebnisse werden durch das Attribut `Metric.FORBIDDEN_LOVE` vom Typ `string`, welches an Class-Knoten geschrieben wird, im RFG vermerkt.

Diese Implementierung entspricht weitgehend der *Java*-Implementierung von *verbotene Dateiliebe*. Es wird lediglich ein anderes Knoten-Attribut verwendet und auf den Vergleich der Dateien, in denen die Klassen enthalten sind, verzichtet.

3.3.50 verbotene Methodenliebe

Die Umsetzung dieses Qualitätsindikators basiert auf der Programmrepräsentationsform RFG.

Die Ergebnisse werden durch das Attribut `Metric.FORBIDDEN_LOVE` vom Typ `string`, wel-

ches an Method-Knoten geschrieben wird, im RFG vermerkt.

Es werden alle `Method-Knoten` durchlaufen und jeweils geprüft, ob es Überschneidungen zwischen der Menge der aufgerufenen Methoden und der Menge der aufrufenden Methoden gibt. Ist dies der Fall, stellt das beteiligte Methodenpaar eine Probleminstanz dar und wird analog zu den bisherigen „*verbotene *-Liebe*“ Indikatoren mit einer eindeutigen ID versehen.

3.3.51 verbotene Paketliebe

Die Umsetzung dieses Qualitätsindikators basiert auf der Programmrepräsentationsform RFG.

Die Ergebnisse werden durch das Attribut `Metric.CYCLE_ID` vom Typ `string`, welches an Package-, Directory- oder Namespace-Knoten (je nach gewählter Paket-Definition) geschrieben wird, im RFG vermerkt.

Zur Vorbereitung der Analyse wird für *C++* eine Zyklenberechnung auf der Sicht `CQI-LIFTEDMODULE` durchgeführt und für *Java* vor der Zyklenberechnung die Sicht `CQI-ANALYSIS` „gelifted“.
Anschließend werden die der Paketdefinition entsprechenden Knoten durchlaufen und jeweils auf Überschneidungen bezüglich ihrer Vorgänger und Nachfolger geprüft. Analog zu *verbotene Dateiliebe* werden hieraus Paare gebildet, die anschließend mit eindeutigen IDs versehen werden.

3.3.52 versteckte Konstantheit

Die Umsetzung dieses Qualitätsindikators basiert auf der Programmrepräsentationsform RFG.

Die Ergebnisse werden durch das Attribut `Metric.HIDDEN_CONST` vom Typ `toggle`, welches an Member-Knoten geschrieben wird, im RFG vermerkt.

Alle nichtkonstanten, statischen Klassenattribute, auf die nicht schreibend zugegriffen wird, werden als *versteckte Konstanten* markiert.

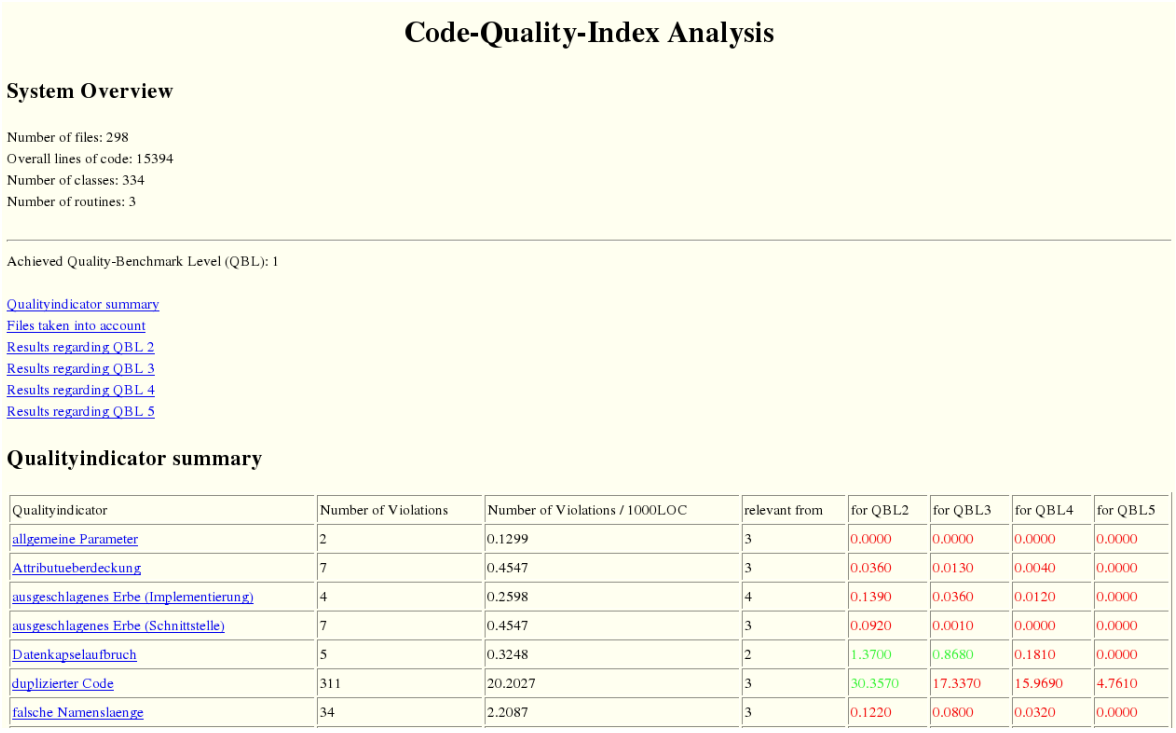
3.4 Ausgabe

Dieser Abschnitt dient der Erläuterung der durch die Aufgabenstellung geforderten HTML-Ausgabe.
Hauptziel der Ausgabe, neben der Darstellung der Ergebnisse, ist vor allem die Nachvollziehbarkeit der einzelnen Ergebnisse. Aus diesem Grund finden nicht nur die Gesamtverletzungszahlen der einzelnen Indikatoren Eingang in den Ergebnisbericht, sondern auch die einzelnen Verletzungen, aus denen sich die Zahlen ergeben. Um noch größere Transparenz zu gewährleisten, ist es in der Regel möglich, von einer Verletzung aus in den jeweiligen Quellcode zu springen.

Ein Ergebnisbericht, wie er im Rahmen dieser Arbeit erzeugt wird, besteht aus 7 verschiedenen Seitentypen, die durch jeweils eine Klasse in Abbildung 3.3 in Abschnitt 3.2.4 auf Seite 42 repräsentiert werden.
Alle Seiten sind von der Klasse `HTML_File` abgeleitet, welche für ein einheitliches Farbschema sorgt und die Funktionalität für das Schreiben der HTML-Dateien auf die Festplatte bereitstellt.
Der so generierte Bericht wird standardmäßig im HOME-Verzeichnis des aktuellen Nutzers (für Windows unter *Eigene Dateien*) als Verzeichnis *analysis_output* abgelegt. Dieser Pfad kann bei Bedarf per Kommandozeilenparameter (`-o`) beliebig angepasst werden.

3.4.1 Startseite

Die Start-Seite des Berichts ist eine Übersichtsseite, die die wichtigsten Fakten der Analyse wiedergibt. Implementiert wird dieser Seitentyp durch die Klasse `Overview_Page`.
Zuoberst findet sich ein Überblick über die wichtigsten Kennzahlen des analysierten Systems (z.B. Lines of Code, Anzahl der berücksichtigten Dateien, Anzahl der enthaltenen Klassen u.ä.). Es folgt die Angabe des erreichten Quality-Benchmark-Level sowie einige Links zu „QBL-Seiten“ sowie einer Verknüpfung zu einer Seite vom Typ „`File-Page`“. Diese gibt einen Überblick über die in der Analyse berücksichtigten Dateien.
Der Hauptteil der Seite wird wird von einer Tabelle eingenommen, die die Anzahl der Verletzungen pro Qualitätsindikator wiederspiegelt und farblich abgesetzt darüber Auskunft gibt, inwieweit die Grenzwerte für die Quality-Benchmark-Level erfüllt bzw. verletzt sind. Ein exemplarischer Ausschnitt einer solchen Startseite ist in Abbildung 3.5 abgebildet.



Code-Quality-Index Analysis

System Overview

Number of files: 298
Overall lines of code: 15394
Number of classes: 334
Number of routines: 3

Achieved Quality-Benchmark Level (QBL): 1

Qualityindicator summary
Files taken into account
Results regarding QBL 2
Results regarding QBL 3
Results regarding QBL 4
Results regarding QBL 5

Qualityindicator summary

Qualityindicator	Number of Violations	Number of Violations / 1000LOC	relevant from	for QBL2	for QBL3	for QBL4	for QBL5
allgemeine Parameter	2	0.1299	3	0.0000	0.0000	0.0000	0.0000
Attributueberdeckung	7	0.4547	3	0.0360	0.0130	0.0040	0.0000
ausgeschlagenes Erbe (Implementierung)	4	0.2598	4	0.1390	0.0360	0.0120	0.0000
ausgeschlagenes Erbe (Schnittstelle)	7	0.4547	3	0.0920	0.0010	0.0000	0.0000
Datenkapselaufbruch	5	0.3248	2	1.3700	0.8680	0.1810	0.0000
duplizierter Code	311	20.2027	3	30.3570	17.3370	15.9690	4.7610
falsche Namenslaenge	34	2.2087	3	0.1220	0.0800	0.0320	0.0000

Abbildung 3.5: Start-Seite

3.4.2 QBL-Seiten

Die auf der Startseite verlinkten „QBL-Seiten“ enthalten eine ähnliche Tabelle, die allerdings nur diejenigen Qualitätsindikatoren darstellt, die im jeweiligen QBL berücksichtigt werden. Hier werden allerdings nur die für den jeweiligen QBL relevanten Schwellenwerte ausgewiesen. Ein exemplarisches Beispiel einer QBL-Seite ist in Abbildung 3.6 zu sehen.

3.4.3 Detail-Seiten

Die Namen der Qualitätsindikatoren in den Tabellen der Start- und QBL-Seiten sind jeweils mit den Detail-Seiten verlinkt, die die einzelnen Verletzungen des jeweiligen Indikators enthalten. In der linken Spalte ist hier jeweils eine fortlaufende Nummer zu finden. Über diese Nummer werden bei Bedarf Gruppenzusammengehörigkeiten ausgedrückt. So werden beispielsweise auf der Detailseite zum Indikator *Attributüberdeckung* jeweils alle überdeckenden Attribute einzeln aufgelistet. Gezählt werden aber nur die überdeckten Attribute, so dass hier mehrere Einträge pro Verletzung auftauchen können. Abbildung 3.7 enthält ein solches Beispiel und besteht aus 12 Einträgen bei nur 7 tatsächlich gezählten Verletzungen.
Der konkrete Inhalt der Detail-Seiten variiert zwischen den einzelnen Qualitätsindikatoren, gibt aber in jedem Fall möglichst exakt Auskunft über das betroffene Artefakt. In den meisten Fällen ist dies der Name des Artefakts, seine Klasse sowie die Code-Position bestehend aus Pfad, Datei und Zeilennummer.

Results for QBL2

Qualityindicator	Number of violations	Number of violations / 1000 LOC	Threshold for current QBL
Datenkapselaufbruch	5	0.3248	1.3700
Identitaetsspaltung	2	0.1299	0.0450
Klasseninzest	7	0.4547	0.0160
Labyrinthmethode	2	0.1299	1.1100
Namensfehler	916	59.5037	22.2680
Objektplacebo (Attribut)	6	0.3898	0.9950
Objektplacebo (Methode)	2	0.1299	1.1290
Polymorphieplacebo	1	0.065	0.0800
Risikocode	4	0.2598	0.1340
halbherzige Operationen	61	3.9626	0.3790
lange Parameterliste	2	0.1299	0.0720
nachlaessige Kommentierung	7134.0	463.4273	370.2640
praenatale Kommunikation	2	0.1299	0.0450
tote Implementierung	2	0.1299	0.0000
tote Methoden	58	3.7677	2.1610
unfertiger Code	21	1.3642	1.1520
verbotene Methodenliebe	1	0.065	0.0200
versteckte Konstantheit	16	1.0394	0.1360

Abbildung 3.6: QBL-Seite

Details table for Qualityindicator "Attributueberdeckung"

Nr.	Base class	Derived class	overlapping attribute	Location of base class
1	OberklasseunvollstaendigeVererbungAttribut	Unterklasse3unvollstaendigeVererbungAttribut	attribut1	/testcode/QBenchCppTestcode/unvollstaendig
1	OberklasseunvollstaendigeVererbungAttribut	Unterklasse2unvollstaendigeVererbungAttribut	attribut1	/testcode/QBenchCppTestcode/unvollstaendig
1	OberklasseunvollstaendigeVererbungAttribut	Unterklasse1unvollstaendigeVererbungAttribut	attribut1	/testcode/QBenchCppTestcode/unvollstaendig
2	OberklasseunvollstaendigeVererbungAttribut	Unterklasse3unvollstaendigeVererbungAttribut	attribut2	/testcode/QBenchCppTestcode/unvollstaendig
2	OberklasseunvollstaendigeVererbungAttribut	Unterklasse2unvollstaendigeVererbungAttribut	attribut2	/testcode/QBenchCppTestcode/unvollstaendig
3	AehnlichAberVerwandt1	AehnlichAberVerwandt2	name	/testcode/QBenchCppTestcode/signaturaehnli
4	AehnlichAberVerwandt1	AehnlichAberVerwandt2	age	/testcode/QBenchCppTestcode/signaturaehnli
5	OberklasseAttributueberdeckung	Unterklasse2	ichWerdeAuchUeberdeckt	/testcode/QBenchCppTestcode/attributueberd
5	OberklasseAttributueberdeckung	Unterklasse1	ichWerdeAuchUeberdeckt	/testcode/QBenchCppTestcode/attributueberd
6	OberklasseAttributueberdeckung	Unterklasse3	transitivueberdeckt	/testcode/QBenchCppTestcode/attributueberd
7	OberklasseAttributueberdeckung	Unterklasse2	sogarIchWerdeUeberdeckt	/testcode/QBenchCppTestcode/attributueberd
7	OberklasseAttributueberdeckung	Unterklasse1	sogarIchWerdeUeberdeckt	/testcode/QBenchCppTestcode/attributueberd

Abbildung 3.7: Ausschnitt Detail-Seite (Qualitätsindikator *Attributüberdeckung*)

Für den Qualitätsindikator *duplizierter Code* implementiert die Klasse `Clones_Details_Page` eine Sonderform der Detail-Seite. Die Anpassung der Detailseite beschränkt sich im Wesentlichen darauf, dass die generierten Hyperlinks nicht auf „Code-Seiten“ sondern auf „Codevergleichs-Seiten“ (vgl. Abschnitt 3.4.4) verweisen.
Aufgrund der oben erwähnten Notwendigkeit andere Hyperlinks auf der Detailseite zu generieren, wurde auch die sonst zur Speicherung von Tabellen verwendete Datenstruktur (`HTML_Table`) durch eine angepasste (`HTML_Clones_Table`) ersetzt.

3.4.4 Code-Seiten

Wie Eingangs erwähnt, bieten die einzelnen Verletzungen die Möglichkeit direkt in den betroffenen Quellcode zu springen. Hierfür wird die jeweils betroffene Code-Datei unter Verwendung des Open Source Tools *webcpp*[4] als HTML formatiert und mit Syntaxhighlighting und Zeilennummern versehen. Die Code-Seiten sind jeweils an mehreren Stellen auf verschiedene Weise verlinkt. Der Link auf der Dateiangabe führt beispielweise nur in die entsprechende Datei. Der Link auf der Zeilennummer hingegen führt direkt zur angegebenen Nummer.
Die Generierung der Code-Seiten funktioniert natürlich nur, wenn das Tool *webcpp* installiert ist und der Quellcode in den im RFG angegebenen Pfaden liegt.

Auch bei den Code-Seiten ist für den Qualitätsindikator *duplizierter Code* ein Sonderfall zu berücksichtigen. Um die Nachvollziehbarkeit von Verletzungen gegen diesen Indikator sicherzustellen, werden die duplizierten Code-Segmente auf einer Codevergleichs-Seite (vom Typ `Code_Comparison_Page`) gegenübergestellt. Diese stellen nur die jeweils betroffenen Code-Stellen (anstatt der ganze Datei) nebeneinander dar, so dass ein direkter Vergleich möglich ist. Aus Gründen der Lesbarkeit wird zusätzlich auf die Zeilennummerierung der Code-Stücke verzichtet.
Ein Beispiel für die Codevergleichs-Seiten findet sich in Abbildung 3.8.

[4] http://webcpp.sourceforge.net/

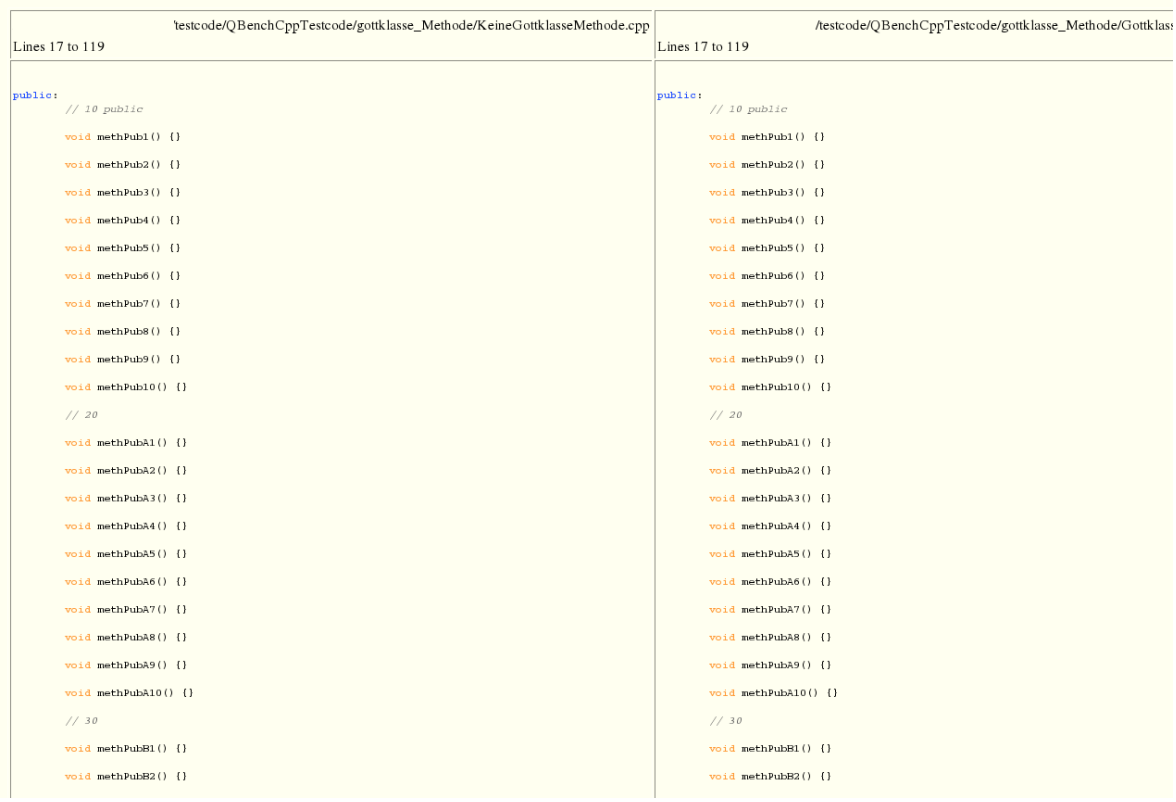


Abbildung 3.8: Beispiel Ausschnitt Codevergleichs-Seite

3.5 Testfälle

Dieser Abschnitt stellt dar, auf welche Weise die korrekte Funktionsweise des implementierten Werkzeugs nachgewiesen werden soll. Zu diesem Zweck werden die einzelnen Testszenarien erläutert.
Es werden zwei verschiedene Testmethoden angewandt. Zum Einen wird ein Funktionstest durchgeführt, bei dem für *C++* und *Java* jeweils Testcode, der die einzelnen Indikatoren gezielt verletzt, analysiert wird. Zum Anderen wird ein Stresstest durchgeführt, bei dem jeweils ein System relevanter Größe für *C++* und *Java* analysiert wird um sicherzustellen, dass die verwendeten Datenstrukturen und Algorithmen geeignet sind, entsprechende Systeme zu verarbeiten.

3.5.1 Funktionale Tests

Der für die funktionalen Tests eingesetzte Code umfasst pro Sprache (*C++* und *Java*) jeweils rund 15.000 Code-Zeilen. In diesen sind pro Indikator in der Regel 1 bis 2 Positiv- und Negativ-Beispiele gezielt implementiert.
Stellenweise verursachen gezielt implementierte Verletzungen eines Qualitätsindikators gleichzeitig Verletzungen eines anderes Indikators. So verursachen beispielsweise die mindestens 51

Attribute – von denen einige als `private` deklariert sind – eines Testfalls für den Qualitätsindikator *Gottklasse (Attribut)* eine entsprechende Anzahl Verletzungen des Indikators *tote Attribute*, weil sie nicht verwendet werden.
Dieser Mangel an Funktionalität führt über den gesamten Testcode verteilt zu einer Vielzahl zusätzlicher Verletzungen. Da der mit der Behebung dieses Umstandes verbundene Aufwand den zeitlichen Rahmen dieser Arbeit überschreiten würde, kann die Anzahl der Testfälle nicht angegeben werden.
Für Indikatoren wie *nachlässige Kommentierung* wurde kein spezieller Testcode implementiert, da es bei einer systemweiten Metrik, die das Verhältniss zwischen Code- und Kommentarzeilen angibt, keine Positiv- und Negativ-Beispiele gibt. Der entsprechende Testfall ergibt sich in diesem Fall aus dem gesamten Testcode.
Es ist anzunehmen, dass durch die oben erwähnte Kombination aus Positiv- und Negativ-Beispielen eine hohe Testabdeckung erreicht wird. Da der genaue Grad der Testabdeckung aufgrund der nicht bekannten Anzahl der Testfälle nicht ermittelbar ist, besteht die Möglichkeit, dass keine vollständige Abdeckung erreicht wird.

Der Testcode für *Java* entstand im Kontext des QBench-Projekts und wurde in der vorliegenden Form freundlicherweise von der Firma SQS[5] zur Verfügung gestellt. Der Testcode für *C++* wurde in Teilen ebenfalls von SQS erstellt, war jedoch zum Zeitpunkt der Übergabe noch nicht fertiggestellt und wurde deshalb in Absprache mit SQS entsprechend erweitert.

3.5.1.1 Testergebnisse für *C++*

Tabelle 3.2 gibt einen Überblick über die Testwerte für *C++*. Nach Qualitätsindikatoren aufgeschlüsselt ist jeweils die Anzahl der gezielt implementierten Verletzungen, gefolgt von der Anzahl der erwarteten und der Werte, die eine Vermessung mit dem Bauhaus-Release Version 5.5.0 ergibt, angegeben. Die Anzahl der erwarteten Verletzungen ergibt sich aus der manuellen Überprüfung einer Vermessung des Testcodes. Die ermittelten Werte werden um fehlende Verletzungen erhöht bzw. um falsche positive Treffer reduziert. Bei Indikatoren mit sehr vielen Verletzungen erfolgt die manuelle Überprüfung auf Stichprobenbasis. Zusätzlich wird die Plausibilität der Ergebnisse dadurch erhärtet, dass der speziell für den jeweiligen Indikator geschriebene Testcode seperat vermessen und gegen die gezielt implementierten Verletzungen abgegelichen wird. Dieser Schritt wird nur im Falle von Abweichungen gesondert dokumentiert.

[5] http://www.sqs.de/

Tabelle 3.2: Testwerte für *C++*

Indikator	Anzahl Verletzungen		
	gezielt implementiert	erwartet	mit Bauhaus 5.5.0
allgemeine Parameter	3	3	2
Attributüberdeckung	4	7	7
ausgeschlagenes Erbe (Implementierung)	3	4	4
ausgeschlagenes Erbe (Schnittstelle)	2	7	7
Datenkapselaufbruch	2	5	5
duplizierter Code	96	407	311
falsche Namenslänge	22	34	34
Generationskonflikt	2	2	2
Gottdatei	2	2	2
Gottklasse (Attribut)	1	1	1
Gottklasse (Methode)	3	3	3
Gottmethode	4	7	7
Gottpaket	2	2	2
halbherzige Operationen	7	61	61
heimliche Verwandtschaft	3	4	4
Identitätsspaltung	3	3	2
Importchaos	1	1	1
Importlüge	3	1	1
informelle Dokumentation	N/A	822	822
Interface-Bypass	5	9	9
Klässchen	3	109	109
Klasseninzest	8	10	7
Konstantenregen	1	1	1
Labyrinthmethode	2	2	2
lange Parameterliste	2	2	2
maskierende Datei	1	38	38
nachlässige Kommentierung	N/A	7134	7134
Namensfehler	8	916	916
Objektplacebo (Attribut)	6	6	6
Objektplacebo (Methode)	2	2	2
Paketchen	1	35	35
Pakethierarchieaufbruch	1	1	1
Polymorphieplacebo	1	1	1
potentielle Privatsphäre (Attribut)	2	138	138
potentielle Privatsphäre (Methode)	2	19	19

Tabelle 3.2: Testwerte für *C++*

Indikator	Anzahl Verletzungen		
	gezielt implementiert	erwartet	mit Bauhaus 5.5.0
pränatale Kommunikation	2	2	2
Risikocode	4	4	4
signaturähnliche Klassen	1	5392	5392
simulierte Polymorphie	1	1	1
späte Abstraktion	3	5	5
tote Attribute	2	100	100
tote Implementierung	2	2	2
tote Methoden	1	58	58
überbuchte Datei	1	17	17
unfertiger Code	6	27	21
unvollständige Vererbung (Attribut)	2	4	4
unvollständige Vererbung (Methode)	2	2	2
verbotene Dateiliebe	1	10	10
verbotene Klassenliebe	1	1	1
verbotene Methodenliebe	1	1	1
verbotene Paketliebe	1	2	2
versteckte Konstantheit	1	16	16

Aus Tabelle 3.2 geht hervor, dass für 47 Qualitätsindikatoren die erwarteten Werte ermittelt werden. Lediglich in 5 Fällen kommt es zu Abweichungen. Im Folgenden werden die Gründe für Abweichungen von der erwarteten Verletzungszahl betrachtet. Zusätzlich werden die Ursachen der Verletzungszahlen bei Qualitätsindikatoren genannt, die sich durch eine besonders hohe Differenz zwischen den gezielt implementierten Verletzungen und den erwarteten Verletzungen auszeichnen.

Für den Qualitätsindikator *allgemeine Parameter* wird eine implementierte Verletzung nicht gefunden, weil die verwendete Bauhaus-Version in manchen Fällen `static-cast`s nicht korrekt abbildet.
Für den Qualitätsindikator *duplizierter Code* werden durch einen Bug in *clones* 48 duplizierte Code-Zeilen nicht als Typ-1-Klon klassifiziert und deshalb nicht erkannt. Daraus ergibt sich ein Fehlwert von 96 Zeilen zum erwarteten Ergebnis.
Die Abweichung bei der Verletzung des Indikators *Identitätsspaltung* resultiert daraus, dass die Klasse `Util` nur einmal im RFG enthalten ist. Die Ursachen sind noch nicht geklärt.
Die Implementierung des Qualitätsindikators *Importlüge* verzichtet – wegen des fast sicheren Auftretens von falsch-positiv Meldungen – bewusst auf die Berücksichtigung von Includes aus Bibliotheken. Entsprechend kann nur eine der drei implementierten Verletzungen erkannt

werden.
Die Abweichung von der erwarteten Verletzungsanzahl für den Indikator *Klasseninzest* resultiert aus fehlenden `Extends`-Kanten zwischen den Klassen `IKlasseMain` und `IKlasseErbt1` bzw. `IKlasseErbt2`.
Der sehr hohe Wert für den Qualitätsindikator *Namensfehler* beruht auf der Tatsache, dass die meisten Methodennamen im Testcode mit einem Kleinbuchstaben beginnen statt mit einem Großbuchstaben, wie in der Indikator-Definition gefordert. Zusätzlich sind in der Treffermenge zwei Verletzungen in der Datei *.cafe++.def* enthalten, die erst seit Version 5.5.0 im RFG auftaucht.
Da der Testcode nur sehr wenige Zugriffe auf Attribute enthält, aber beispielsweise für den Indikator *Gottklasse (Attribut)* sehr viele Attribute deklariert werden mussten, kommt es für den Qualitätsindikator *potenzielle Privatsphäre (Attribut)* zu sehr vielen Treffern, die jedoch plausibel erscheinen. Diese Erklärung ist in Bezug auf Methoden analog für den Indikator *potenzielle Privatsphäre (Methode)* zutreffend. Auch die Verletzungszahlen der Indikatoren *tote Attribute* und *tote Methoden* werden durch diesen Umstand stark angehoben.
Im vorliegenden Testcode für *C++* sind sehr viele Klassen enthalten, die nur wenige Methoden enthalten, die sich in ihrer Signatur häufig gleichen. Insbesondere der Testcode für den Indikator *Gottpaket* enthält pro Paket mehr als 50 Klassen, deren Methoden einfach kopiert wurden und dadurch identische Signaturen haben. Weil jeweils Klassenpaare gezählt werden und somit bereits vier Klassen mit identischen Methodensignaturen als sechs Verletzungen gezählt werden, schaukelt sich die Verletzungszahl für den Qualitätsindikator *signaturähnliche Klassen* zu extrem hohen Werten auf, die jedoch durchaus plausibel erscheinen.
Die Verletzungszahl des Indikators *unfertiger Code* ist um 6 Funde zu niedrig. Diese liegen in einer Datei, die leider nicht im RFG auftaucht. Der Grund liegt vermutlich darin, dass diese – abgesehen von den Kommentaren – keinen weiteren Code enthält.

Auffällig ist, dass eine Vergleichsmessung der Lines of Code mit dem Programm `sloccount` um rund 5000 Zeilen niedriger liegt, als die über den RFG ermittelte Zahl. Ein Blick in die `FILE`-View des RFG enthüllt, dass offenbar nicht alle Dateien der *C++-Standardbibliothek* ausgefiltert wurden. Diese Diskrepanz führt unter Umständen zu einer leicht besseren QBL-Einstufung aufgrund der durch die Normalisierung zu positiven Einzel-Ergebnisse, die sich bei größeren Systemen sehr schnell relativiert. Da sich dieses Phänomen auf alle Messungen gleichmäßig auswirkt, ist die Vergleichbarkeit der Messungen nach wie vor gewährleistet. Aus diesem Grund wird auf eine aufwändige Fehlersuche und -behebung an dieser Stelle verzichtet.

Fazit

Lediglich für fünf Qualitätsindikatoren stimmt das Ergebniss nicht mit den Erwartungen überein. Diese Abweichungen konnten allesamt durch kleine Fehler im Bauhaus Release 5.5.0 erklärt werden, die voraussichtlich in zukünftigen Versionen behoben werden. Die Ergebnisse der restlichen Indikatoren erscheinen durchweg plausibel und sind größtenteils durch manuelle Überprüfung im Quellcode nachvollziehbar korrekt.

Dies lässt den Schluß zu, dass das implementierte Werkzeug zur automatisierten Ermittlung des Code-Quality-Index korrekt arbeitet.

3.5.1.2 Testergebnisse für *Java*

Tabelle 3.3 gibt einen Überblick über die Testwerte für *Java*. Nach Qualitätsindikatoren aufgeschlüsselt ist jeweils die Anzahl der gezielt implementierten Verletzungen, gefolgt von der Anzahl der mit Bauhaus ermittelten Verletzungen angegeben. Da schon diese Zahlen deutliche Diskrepanzen zwischen den implementierten und gefundenen Verletzungen aufweisen, wird auf die Angabe der „erwarteten“ Verletzungen verzichtet.
Die Zahlen bzgl. der gezielt implementierten Verletzungen wurden freundlicherweise größtenteils zusammen mit dem Testcode von SQS zur Verfügung gestellt. Einige wenige wurden modifiziert, da sie sich auf eine veraltete Version bezogen und somit nicht mehr zutreffend waren. Für weitere Analysen wäre hier zu berücksichtigen, dass möglicherweise nicht alle Änderungen entdeckt worden sind.
Um den Testcode mit `jafe` übersetzen zu können, musste der Testcode für den Indikator *Polymorphieplacebo* auskommentiert werden. Die darin implementierten drei Verletzungen können also nicht gefunden werden.
Zur Übersetzung des Quellcodes in IML und RFG werden die Bauhaus-Tools `jafe` und `iml2rfg` verwendet. Da `jafe` nicht in den Release-Versionen enthalten ist, sondern selbst übersetzt werden muss, ist zum Nachvollziehen dieser Vermessung – neben dem Zugriff auf das Bauhaus-Repository der Universität-Stuttgart – zusätzlich der Zugriff auf das Repository der Axivion GmbH nötig, da sowohl das `Source-Metrics-Plugin` als auch die IML-Scripting-Schnittstelle ausschließlich dort entwickelt werden.

Tabelle 3.3: Testwerte für *Java*

Indikator	Anzahl Verletzungen	
	gezielt implementiert	mit Bauhaus ermittelt
allgemeine Parameter	2	2
Attributüberdeckung	4	3
ausgeschlagenes Erbe (Implementierung)	1	10
ausgeschlagenes Erbe (Schnittstelle)	3	0
Datenkapselaufbruch	3	2
duplizierter Code	N/A	718
falsche Namenslänge	$8 < 2$ & $8 > 50$	23
Generationskonflikt	1	4
Gottdatei	2	0
Gottklasse (Attribut)	1	1
Gottklasse (Methode)	3	3
Gottmethode	4	7
Gottpaket	2	0

Tabelle 3.3: Testwerte für *Java*

Indikator	Anzahl Verletzungen	
	gezielt implementiert	mit Bauhaus ermittelt
halbherzige Operationen	2	3
heimliche Verwandtschaft	3	2
Identitätsspaltung	3	73
Importchaos	4	13
Importlüge	4	0
informelle Dokumentation	N/A	1080
Interface-Bypass	5	9
Klässchen	5	265
Klasseninzest	8	29
Konstantenregen	1	0
Labyrinthmethode	3	0
lange Parameterliste	2	2
maskierende Datei	2	4
nachlässige Kommentierung	N/A	0,0
Namensfehler	16	60
Objektplacebo (Attribut)	4	0
Objektplacebo (Methode)	1	0
Paketchen	1	0
Pakethierarchieaufbruch	3	0
Polymorphieplacebo	3	3
potentielle Privatsphäre (Attribut)	2	143
potentielle Privatsphäre (Methode)	1	17
pränatale Kommunikation	2	1
Risikocode	3	29
signaturähnliche Klassen	4	0
simulierte Polymorphie	3	0
späte Abstraktion	2	4
tote Attribute	6	498
tote Implementierung	0	0
tote Methoden	2	17
überbuchte Datei	2	16
unfertiger Code	7	0
unvollständige Vererbung (Attribut)	2	0
unvollständige Vererbung (Methode)	2	0
verbotene Dateiliebe	2	9
verbotene Klassenliebe	1	3
verbotene Methodenliebe	2	2

Tabelle 3.3: Testwerte für *Java*

Indikator	Anzahl Verletzungen	
	gezielt implementiert	mit Bauhaus ermittelt
verbotene Paketliebe	3	0
versteckte Konstantheit	4	16

Wie deutlich zu sehen ist, werden für viele Indikatoren überhaupt keine Verletzungen ermittelt, obwohl gezielt Verletzungen implementiert worden sind. Diese Unstimmigkeiten sind vor allem durch Unterschiede zwischen den Programmrepräsentationen für *C++* und *Java* begründet. Eine Überprüfung der Verletzungszahlen bei den Indikatoren für die Verletzungen ermittelt wurden ergibt, dass die meisten Ergebnisse zutreffen, sich allerdings in Details vereinzelt Unstimmigkeiten ergeben. Für *Risikocode* beispielsweise wird regelmäßig ein fehlendes `break` zu viel verzeichnet. Es ist anzunehmen, dass sich der `default`-Zweig nicht genauso wie in *C++* von einem normalen `Case`-Zweig unterscheidet.
Verletzungen für *tote Attribute* verweisen stellenweise auf falsche Code-Stellen. Meistens werden in diesen Fällen Attribute fälschlicherweise einer Unterklasse zugeordnet. Einige dieser Fälle tauchen mehrfach in der Detailliste auf. Die einzig logische Schlußfolgerung angesichts der Tatsache, dass dieser Indikator für *C++* problemlos funktioniert ist, dass der RFG in diesem Zusammenhang fehlerbehaftet ist.
Die nicht gefundenen Treffer in allen Indikatoren, die sich auf Pakete beziehen, resultieren daraus, dass in der Kombination `jafe/iml2rfg` offenbar keine `Package`-Knoten generiert werden. Eine kurze Vergleichsmessung mit einem mit `java2rfg` generierten RFG ergab hier jedoch korrekte Ergebnisse.

Fazit

Die oben exemplarisch geschilderten Ergebnisse zeigen sehr deutlich, dass es offensichtlich eine ganze Reihe von Unterschieden zwischen den Programmrepräsentationen für *C++* und *Java* gibt. Zusätzlich verschärft wird dieses Problem dadurch, dass sich schon der durch `iml2rfg` – aus einer mit `jafe` erzeugten IML – generierte RFG deutlich von einem mit `java2rfg` generierten RFG unterscheidet.
Insbesondere die Repräsentation von Vererbungsbeziehungen inklusive der Zuordnung von vererbten Artefakten zu den jeweiligen Klassen scheint fehlerbehaftet zu arbeiten.
Vor diesem Hintergrund kann aktuell nicht damit gerechnet werden, für *Java*-basierte Systeme eine verlässliche und konsistente Datenbasis zu erhalten. Aus diesem Grund erscheinen weitere Tests – wie die Vermessung von realen Systemen zur Evaluation oder als Stresstest – wenig sinnvoll, weil im Fehlerfall nur schwer nachvollziehbar ist, ob der Fehler ein Programmierfehler ist oder durch eine fehlerhafte Datenbasis ausgelöst wurde.

3.5.2 Stresstest

Als Stresstest für *C++* wurde ein reales Softwaresystem aus dem Bereich der öffentlichen Verwaltung ausgewählt. Dieses beinhaltete nach einer umfangreichen Code-Bereinigung noch ca. 407.000 (davor ca. 700.000) Zeilen Visual-C++ Code. Als Stresstest ist ein solches relativ großes System deshalb geeignet, weil insbesondere die IML für große Systeme sehr viel Platz im Speicher beanspruchen kann und eine Analyse zumindest der IML-basierten Qualitätsindikatoren möglicherweise schlicht daran scheitert, dass nicht genug Arbeitsspeicher zur Verfügung steht. Insbesondere bei 32-Bit Systemen ist hier schnell die technisch maximal adressierbare Grenze von 4 Gigabyte erreicht.
Aus rechtlichen Gründen können zu dem ausgewählten System nur eingeschränkte Angaben gemacht werden. Die Ergebnisse können nur in anonymisierter Form und ohne Detaillisten abgedruckt werden und sind – zusammen mit den Parametern der Vermessung – in Abschnitt 4.2.9 dargelegt.

KAPITEL 4

Evaluation

Dieses Kapitel befasst sich mit der Evaluierung der in Kapitel 3 vorgestellten Implementierung. Zu diesem Zweck wird zunächst das Laufzeitverhalten der Implementierung betrachtet, gefolgt von Vermessungen verschiedener Softwaresysteme.
Die Betrachtung des Laufzeitverhaltens konzentriert sich vor allem auf die Frage, welche Analyseschritte besonders zeitaufwendig sind. Zusätzlich erfolgt eine Betrachtung der Laufzeiten der verschiedenen Vermessungen in Relation zur Größe der jeweiligen Systeme.
Die Vermessungen dienen dem grundsätzlichen Nachweis, dass das implementierte Werkzeug für die Vermessung von realen Softwaresystemen eingesetzt werden kann und eine solche Vermessung zur Ermittlung der technischen Qualität sinnvoll ist.
Durch die Vermessung mehrerer Versionen des selben Softwaresystems (QT-Bibliothek) wird darüber hinaus gezeigt, dass eine kontinuierliche Überwachung der technischen Qualität über den gesamten Produktzyklus eines Softwaresystems wünschenswert und sinnvoll erscheint.
Aufgrund der unbefriedigenden Test-Ergebnisse bei der Vermessung des Java-Testcodes werden im Rahmen dieses Kapitels ausschließlich *C++*-Systeme herangezogen.
Die weitgehende Beschränkung der vermessenen Systeme auf Open Source Software ist zum Einen sinnvoll, um eine maximale Nachvollziehbarkeit und Reproduzierbarkeit der Ergebnisse sicher zu stellen und zum Anderen notwendig, weil die Verfügbarkeit des Quellcodes die Grundvoraussetzung für eine Vermessung ist.

Mit Ausnahme des anonymen Systems werden sämtliche Vermessungen sowie die Ermittlung der Laufzeiten auf einem handelsüblichen PC durchgeführt. Dessen relevanten Eckdaten sind:

Prozessor: Intel Pentium Core 2 Duo E6600 (2 × 2,4Ghz)

RAM: 3 Gigabyte PC5400/DDR2-667

Massenspeicher-Schnittstelle: SATA II

Betriebssystem: Linux – Kubuntu 7.04

Bauhaus-Version: Axivion Bauhaus-Suite 5.5.0

Die genannten Hardware-Komponenten beeinflussen die Geschwindigkeit von Berechnungen

sowie die Ladezeiten von IML und RFG. Die genannte Bauhaus-Version stellt die Reproduzierbarkeit der Ergebnisse sicher.

Kapitelinhalt

4.1 Laufzeitverhalten

Dieser Abschnitt betrachtet das Laufzeitverhalten der implementierten Lösung.
Zunächst wird am Beispiel des *C++*-Testcodes ermittelt, wie zeitaufwendig die einzelnen Metriken sind.
Anschließend werden die Laufzeiten der Vermessungen in Abschnitt 4.2 gegenüber gestellt, um zu überprüfen, ob eine Beziehung zwischen der Laufzeit der Analyse und der Systemgröße besteht.
Abschließend erfolgt eine Betrachtung des Speicherbedarfs.

4.1.1 Laufzeiten der Indikatoren

In diesem Abschnitt werden die Laufzeiten der einzelnen Metriken sowie der Phasen des Programmablaufs am Beispiel des *C++*-Testcodes analysiert. Dabei geht es vornehmlich darum herauszufinden, wie die Laufzeiten zustande kommen und welche Reduzierungspotenziale bestehen.
Die Laufzeiten der einzelnen Programmphasen sind in Tabelle 4.1 aufgeschlüsselt.

Tabelle 4.1: Laufzeiten der Programmphasen (am Beispiel des *C++*-Testcodes)

Vorgang	Laufzeit (in Sekunden)
Berechnung IML-basierter Metriken	60,31
Berechnung RFG-basierter Metriken	5.54
Auswertung der Metriken	6.93
Generierung des HTML-Reports	14.10
Gesamtlaufzeit	87.64

Wie aus dieser Tabelle hervorgeht, nimmt die Berechnung der IML-basierten Metriken die mit Abstand meiste Zeit in Anspruch. Dies liegt vor allem daran, dass die IML wesentlich mehr Informationen enthält als der RFG und hier Bibliotheken, die im RFG ausgefiltert werden können, enthalten sind und somit mit verarbeitet werden müssen.
Der im Vergleich zur Berechnung der RFG-basierten Metriken und der Auswertung recht hohe Zeitaufwand für die Generierung des HTML-Reports resultiert vor allem aus der Verwendung des Tools `webcpp`. Dieses wird per Systemaufruf für jede Datei einzeln aufgerufen, wodurch ein vergleichsweise hoher Overhead entsteht.
Für eine deutliche Reduzierung der Laufzeit erscheint damit die Berechnung der IML-basierten Metriken der erfolgversprechendste Ansatzpunkt.
Wie aus Tabelle 4.2 hervorgeht, zeichnet sich die Mehrzahl der Metriken, unabhängig von der zugrunde liegenden Programmrepräsentation, durch eine sehr geringe Laufzeit aus.
Die drei großen Ausreißer – ausnahmslos IML-basierte Metriken – sind *Objektplacebo (Attribut)*, *signaturähnliche Klassen* und *tote Implementierung*, die alleine für fast die Hälfte der Gesamtlaufzeit verantwortlich zeichnen.
Beim Indikator *tote Implementierung* entsteht die hohe Laufzeit daraus, dass jede einzelne `return` und `throw` Anweisung betrachtet werden muss. Derartige Anweisungen sind bereits

in der Standardbibliothek – von der ein nicht unerheblicher Teil durch die Inkludierung von `iostream` in der IML enthalten ist – in großer Zahl enthalten. Optimierungspotenziale sind hier nicht erkennbar.
Die Laufzeit von *Objektplacebo (Attribut)* entsteht vor allem aus der schieren Anzahl der lesenden und schreibenden Zugriffe auf Variablen sowohl im Testcode als auch in der Standardbibliothek. Eine offensichtliche Optimierungsmöglichkeit ist auch hier nicht zu erkennen.
Beim Indikator *signaturähnliche Klassen* entsteht die hohe Laufzeit vor allem durch den Vergleich der Methodensignaturen jeder Klasse mit den Methodensignaturen jeder anderen im System enthaltenen Klasse. Da die Anzahl der Klassen auch durch die inkludierten Teile der Standardbibliothek deutlich erhöht wird, kommt es hier zwangsweise zu hohen Laufzeiten. Eine offensichtliche Möglichkeit zur Optimierung wäre, Klassen der Standardbibliothek auszufiltern, um somit die Anzahl der zu vergleichenden Klassen zu reduzieren. Bei großen Systemen mit vielen Klassen und Methoden würde der so gewonnene Effekt jedoch zwangsweise an Bedeutung verlieren. Erfolgversprechender erscheint die Verwendung eines schnelleren Algorithmus bei der Auswahl der zu vergleichenden Klassen und dem Vergleich als solchen. Die Umsetzung dieser Überlegung bleibt einer eventuellen Weiterentwicklung des implementierten Werkzeugs vorbehalten, weil der zeitliche Rahmen dieser Arbeit einen solchen Versuch nicht gestattet.

Tabelle 4.2: Laufzeiten der einzelnen Metriken (am Beispiel des *C++*-Testcodes)

Qualitätsindikator	verstrichene Zeit (in Sekunden)
allgemeine Parameter	2,13
Attributüberdeckung	0,17
ausgeschlagenes Erbe (Implementierung)	0,05
ausgeschlagenes Erbe (Interface)	0,10
Datenkapselaufbruch	0,10
duplizierter Code	1,66
falsche Namenslänge und Namensfehler	0,36
Generationskonflikt	0,33
Gottklasse (Attribut)	0,10
Gottklasse (Methode)	0,13
halbherzige Operationen	0,09
heimliche Verwandtschaft	0,06
Identitätsspaltung	0,04
Importchaos	0,05
Importlüge	3,39
informelle Dokumentation	0,09
Interface-Bypass	0,09
Klasseninzest	0,11
Konstantenregen	0,04
Labyrinthmethode	0,03
maskierende Datei	0,11
Objektplacebo (Attribut)	14,45
Objektplacebo (Methode)	1,76
Paketchen und Gottpaket	0,06
Pakethierarchieaufbruch	0,06
Polymorphieplacebo	0,23
potenzielle Privatsphäre (Attribut)	0,10
potenzielle Privatsphäre (Methode)	0,03
pränatale Kommunikation	0,11
Risiko Code	0,35
signaturähnliche Klassen	12,21
simulierte Polymorphie	0,004
späte Abstraktion	0,02
tote Attribute	0,04
tote Implementierung	15,89
tote Methoden	0,03
überbuchte Datei	0,11
unvollständige Vererbung (Attribut)	0,12
unvollständige Vererbung (Methode)	0,38
verbotene Dateiliebe	0,02
verbotene Klassenliebe	0,06
verbotene Methodenliebe	0,05
verbotene Paketliebe	0,03
versteckte Konstantheit	0,02

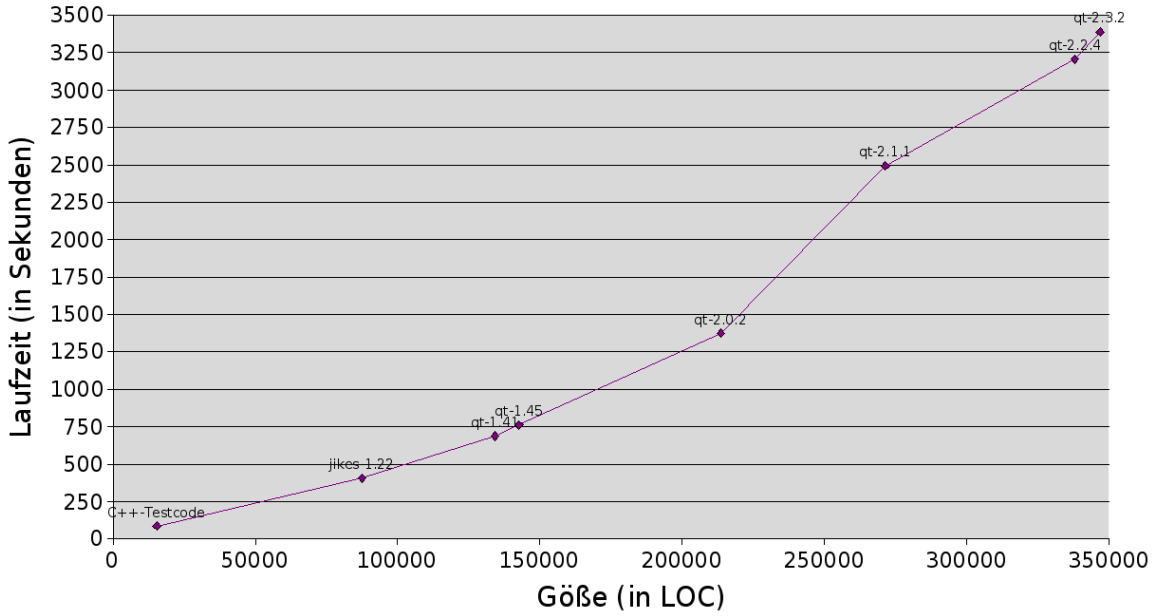
4.1.2 Laufzeiten der Vermessungen

Dieser Abschnitt betrachtet in Tabelle 4.3 die Laufzeiten der Vermessungen in Abschnitt 4.2 mit dem Ziel, einen möglichen Zusammenhang zwischen der Laufzeit und der Systemgröße (in Lines of Code) aufzudecken. Um dies zu erleichtern, werden die Laufzeiten in Abbildung 4.1 einander grafisch gegenüber gestellt.

Tabelle 4.3: Laufzeiten der Vermessungen

System	Lines of Code	Laufzeit (in Sekunden)
C++-Testcode	15.394	87,64
jikes-1.22	87.474	407,95
qt-1.41	134.152	687,97
qt-1.45	142.614	761,72
qt-2.0.2	213.500	1375,17
qt-2.1.1	271.329	2.492,99
qt-2.2.4	337.914	3.207,10
qt-2.3.2	346.748	3.389,05

Abbildung 4.1: Laufzeiten der Vermessungen



Obwohl die ersten Messungen den Verdacht nahelegen, dass ein linearer Zusammenhang zwischen der Systemgröße und der Analyselaufzeit besteht, zeigen die deutlichen Sprünge zwischen *qt-2.0.2* und *qt-2.1.1* sowie *qt-2.2.4* und *qt-2.3.2*, dass es zumindest noch weitere Faktoren gibt, die Einfluss auf die Laufzeit nehmen.
Betrachtet man die Laufzeiten der einzelnen Metriken sowie die Ursachen für die wenigen besonders hohen Laufzeiten, drängt sich geradezu die Annahme auf, dass die Struktur des untersuchten Systems einen entscheidenden Einfluss auf die Laufzeit der Vermessung hat. Ein

System, in dem die einzelnen Artefakte nicht einmal die Voraussetzungen für den Verdacht einer Verletzung erfüllen, wird zwangsweise schneller vermessen werden, als ein System vergleichbarer Größe mit deutlich mehr Verletzungen. Es muss während der Vermessung schlicht wesentlich weniger Code ausgeführt werden und auch die Auswertung und das Generieren des Ergebnis-Reports nehmen natürlich weniger Zeit in Anspruch, je weniger Verletzungen berücksichtigt werden müssen.

Es muss also festgehalten werden, dass sich die Laufzeit einer Analyse nicht zuverlässig anhand der Systemgröße vorhersagen lässt, weil auch die Wartbarkeit – im Sinne des Code-Quality-Index – des Systems eine große Rolle spielt, die ja gerade durch die Analyse bestimmt werden soll.
Inwieweit eine zuverlässige Vorhersage der Laufzeit auf Grundlage der Systemgröße innerhalb gewisser Toleranzen zumindest näherungsweise möglich ist, wäre auf Basis einer größeren Vergleichsbasis zu ermitteln.

4.1.3 Speicherbedarf

Der Speicherbedarf der Implementierung ist ebenso wie die Laufzeit von verschiedenen Faktoren abhängig.
Den mit Abstand größten Speicherbedarf verursacht in der Regel das Laden der IML- und RFG-Dateien, die jeweils komplett im Speicher gehalten werden müssen. Insbesondere die IML nimmt selbst bei relativ kleinen Dateigrößen sehr viel Platz im Arbeitsspeicher in Anspruch. Die IML des *C++*-Testcodes beispielsweise belegt auf der Festplatte 148,7 MB. Nach dem Laden der Datei mit dem Scripting-Interface benötigt sie im Arbeitsspeicher gute 700 MB.
Der maximale Speicherbedarf der Vermessung des Testcodes liegt bei etwas mehr als 850 MB und darin ist neben der IML auch noch der Speicherbedarf für den RFG enthalten.
Die Daten, die im Zuge der Metrikerhebung vorgehalten werden, können in diesem Zusammenhang also offenbar weitgehend als vernachlässigbar gelten. Dies liegt vor allem daran, dass es sich größtenteils um Zeiger auf Objekte in IML oder RFG handelt, die nicht viel Speicher belegen.
Die IML für das zu Testzwecken untersuchte System *qt-1.41* nimmt – obwohl das System deutlich mehr Lines of Code beinhaltet als der Testcode – 52 MB auf der Festplatte und knapp 300 MB im Arbeitsspeicher in Anspruch. Ein eindeutiger Zusammenhang zwischen der Systemgröße und der Größe der resultierenden IML scheint also ebenfalls nicht zu bestehen.

Ein weiterer Posten ist das Python-Dictionary, in dem die Ergebnisse und Detaillisten gesammelt werden. Seine Größe hängt vor allem von der Anzahl der im System ermittelten Verletzungen ab, da sich hieraus der Umfang der Detaillisten ergibt. Unter Umständen kann ein System mit sehr vielen Verletzungen den Speicherbedarf an dieser Stelle also zusätzlich

in die Höhe treiben.

Zusammenfassend muss festgestellt werden, dass – wie schon die Laufzeit – auch der Speicherbedarf einer Analyse in hohem Maße vom Umfang der IML und der vorab nicht bekannten Anzahl der Verletzungen abhängt.

4.2 Vermessungen

Im Zuge dieses Abschnitts werden verschiedene Softwaresysteme vermessen, um – durch gefundene Verletzungen und daraus resultierenden QBL-Einstufungen – nachzuweisen, dass das entwickelte Tool einen praktischen Mehrwert bietet und einen Beitrag zur Erkennung von Optimierungspotenzialen hinsichtlich der Wartbarkeit von Softwaresystemen leisten kann.

Zur Vermessung wurden der alternative Open Source Java-Compiler jikes in Version 1.22 sowie mehrere Versionen der Open Source Qt-Klassenbibliothek ausgewählt. Die Ergebnisse dieser Auswahl zeigen, dass die Sinnhaftigkeit der Vermessung eines Softwaresystems nicht von seinem Typ oder seinem Verwendungszweck abhängt. Weiterhin wird offensichtlich, dass eine kontinuierliche Überwachung der Wartbarkeit insbesondere bei Software – die beständig weiterentwickelt wird – sinnvoll erscheint, um Aufwand und Kosten der Weiterentwicklung möglichst gering zu halten.
Das zuletzt vermessene anonyme System diente vor allem als Stresstest und wird gesondert betrachtet, weil die Vermessung mit anderen Versionen sowohl von Bauhaus als auch des Tools zur Bestimmung des Code-Quality-Index durchgeführt wurde.
Aufgrund der wenig befriedigenden Testergebnisse für *Java* wurden ausschließlich in *C++* geschriebene Systeme ausgewählt.
Im Folgenden werden, um eine bessere Lesbarkeit zu erreichen, jeweils nur die Verletzungszahlen tabellarisch aufgelistet und keine Screenshots der Übersichtstabellen abgedruckt.

4.2.1 generelles Vorgehen

Um Wiederholungen zu vermeiden wird hier kurz das generelle Vorgehen bei der Vermessung der ausgewählten Softwaresysteme dargelegt.
In der Regel reicht es aus, den Build-Vorgang dahingehend anzupassen, dass der Bauhaus-Compiler cafeCC statt dem ursprünglich vorgesehenen Compiler verwendet wird. Dies kann etwa durch Modifikationen des Makefiles geschehen oder durch einen entsprechenden Parameter beim Aufruf von make. Vor dieser Anpassung ist es ratsam, das zu untersuchende System einmal normal zu übersetzen, um sicherzustellen, dass alle Pfade und Bibliotheken korrekt eingerichtet sind und eventuelle Fehler ausschließlich auf Bedienungsfehler oder Bauhaus-Spezialitäten zurückzuführen sind.
Sollten nach der Anpassung des Build-Prozesses Fehlermeldungen auftreten, ist zunächst zu

überprüfen, ob sich das Problem möglicherweise durch das Setzen zusätzlicher Parameter für den Bauhaus-Compiler lösen lässt. In den meisten Fällen rühren derartige Probleme daher, dass der Build-Prozess auf einen bestimmten Compiler optimiert wurde und dieser stellenweise anderes arbeitet als cafeCC.
Sollte sich das Problem auch so nicht lösen lassen, muss die entsprechende Code-Stelle überprüft und gegebenenfalls modifiziert werden. Für *qt-1.41* wurde beispielsweise eine Methode auskommentiert, die in einer Compiler-spezifischen Prä-Prozessor-Anweisung deklariert wurde. Dieses Vorgehen kann das Ergebniss unter Umständen leicht verfälschen. So würde etwa das Entfernen des einzigen Zugriffs auf ein privates Attribut eine Verletzung des Qualitätsindikators *tote Attribute* hervorrufen, die im ursprünglichen Code nicht vorhanden war. In Anbetracht des Umfangs der untersuchten Systeme sind solche Ausnahmefälle jedoch zu vernachlässigen.
Die Lizenz der frühen QT-Versionen verbietet eine Veröffentlichung von modifizierten Versionen, daher wird – um lizenzrechtliche Schwierigkeiten zu vermeiden – auf eine genaue Beschreibung der erfolgten Änderungen verzichtet. Die Änderungen ergeben sich jedoch aus den Fehlermeldungen von cafeCC und sind damit bei Bedarf leicht reproduzierbar und nachvollziehbar.

Im Zuge dieser Arbeit wird der Bauhaus-Compiler cafeCC grundsätzlich mit dem Parameter „-B“ (siehe [AXI07a, Seite 117]) aufgerufen, um nach dem Verschieben der Programmrepräsentationen oder des Quellcodes nicht neu übersetzen zu müssen.
Da die Vermessungen unter Linux durchgeführt werden, erfolgen die Aufrufe von `iml2rfg` und des implementierten Tools mit den Parametern `-lib /usr` bzw. `--lib /usr`, um die C++-Standardbibliotheken von der Untersuchung auszuschließen. Weiterhin wird angenommen, dass Pakete über Verzeichnisse organisiert sind. Deshalb wird das Vermessungstool über die Konfigurationsdatei entsprechend konfiguriert.

Wie sich im Zuge der nachfolgenden Vermessungen herausstellte, arbeitet der Indikator *tote Implementierung* offensichtlich fehlerhaft. Erste oberflächliche Analysen des Problems legen die Vermutung nahe, dass insbesondere `inline` Methoden und Funktionen sowie `if`-Anweisungen, die auf eine Umschließung ihrer bedingten Anweisung mit geschweiften Klammern verzichten, in der IML so modelliert werden, dass einer Kontrollflußunterbrechung weitere Anweisungen nachfolgen. Eine genauere Analyse muss an dieser Stelle aus Zeitgründen unterbleiben. Da eine Wiederhohlung der Vermessungen sehr viel Aufwand verursachen würde und die QBL-Einstufungen – also die Endergebnisse – durch den Fehler nicht beeinflusst werden, wird darauf verzichtet. Statt dessen wird im Zuge der Ergebnisbetrachtung, der Indikator *tote Implementierung* – obgleich er der Vollständigkeit halber abgedruckt wird – außen vor gelassen.

4.2.2 jikes-1.22

Die Vermessung des alternative Java-Compilers jikes verlief ohne erkennbare Probleme und führte als Ergebnis zu einem Quality-Benchmark-Level von 1.
Eine bessere Einstufung wird augenscheinlich vor allem durch die hohen Werte für die ab QBL 2 relevanten Qualitätsindikatoren verhindert. Bei Indikatoren die erst ab QBL 3 (z.B. *maskierende Datei*) oder 4 (z.B. *Klässchen*) relevant werden sind die Grenzwerte zu einem wesentlich höheren Anteil erfüllt.

Tabelle 4.4: Übersicht jikes-1.22

Erreichter QBL	1
Analysezeit	407,95 Sekunden
Lines of Code	87.474
Dateien	71
Klassen	396

Tabelle 4.5: Verletzungszahlen jikes-1.22

Qualitätsindikator	Verletzungen	Qualitätsindikator	Verletzungen
allgemeine Parameter	60	nachlässige Kommentierung	65272,0
Attributüberdeckung	0	Namensfehler	1163
ausgeschlagenes Erbe (Implementierung)	6	Objektplacebo (Attribut)	0
ausgeschlagenes Erbe (Schnittstelle)	0	Objektplacebo (Methode)	1
Datenkapselaufbruch	454	Paketchen	0
duplizierter Code	910	Pakethierarchieaufbruch	0
falsche Namenslänge	18	Polymorphieplacebo	0
Generationskonflikt	0	potenzielle Privatsphäre (Attribut)	550
Gottdatei	16	potenzielle Privatsphäre (Methode)	300
Gottklasse (Attribut)	2	pränatale Kommunikation	6
Gottklasse (Methode)	9	Risikocode	468
Gottmethode	39	signaturähnliche Klassen	50
Gottpaket	1	simulierte Polymorphie	0
halbherzige Operationen	5	späte Abstraktion	0
heimliche Verwandtschaft	20	tote Attribute	189
Identitätsspaltung	7	tote Implementierung	234
Importchaos	0	tote Methoden	205
Importlüge	0	überbuchte Datei	22
informelle Dokumentation	4270	unfertiger Code	87
Interface-Bypass	140	unvollständige Vererbung (Attribut)	0
Klasseninzest	159	unvollständige Vererbung (Methode)	37
Klässchen	12	verbotene Dateiliebe	49
Konstantenregen	1	verbotene Klassenliebe	9
Labyrinthmethode	0	verbotene Methodenliebe	20
lange Parameterliste	4	verbotene Paketliebe	0
maskierende Datei	4	versteckte Konstantheit	40

Die in Tabelle 4.5 aufgeführten Verletzungszahlen erscheinen nach stichprobenhafter Überprüfung schlüssig. Bei verschiedenen Indikatoren fallen jedoch stellenweise Verletzungen auf, die zwar per Definition korrekt sind, im konkreten Fall allerdings offensichtlich bewusst eingebaut sind oder zumindest keine schädlichen Auswirkungen haben.

Besonders auffällig ist der hohe Wert für *nachlässige Kommentierung*. Betrachtet man den Quellcode der Dateien mit besonders hohen Werten, wird deutlich, dass hier häufig sehr kleine Methoden – in der Datei ast.h beispielsweise rund 80, die verschiedene Typkonvertierungen realisieren – deklariert und implementiert werden. Durch eine sprechende Benennung bedürfen diese kaum einer ausführlichen Kommentierung. So ist fraglich, inwieweit der hohe Wert für *nachlässige Kommentierung* tatsächlich einen Qualitätsmangel darstellt.
Auch der Indikator *Risikocode* zeichnet sich durch eine hohe Zahl von Verletzungen aus, die im Vergleich mit den Grenzwerten ungewöhnlich hoch erscheint. Die folgenden zwei exemplarischen Beispiele Listing 4.1 und Listing 4.2 machen jedoch deutlich, dass auch hier offenbar einige „nicht schädliche“ Verletzungen enthalten sind.

Listing 4.1: platform.h

```
862     {
863     case 10: case 11: case 12: case 13: case 14: case 15:
864          str[i] = U_A - 10 + d;
865          break;
866     default:
867          str[i] = U_0 + d;
868          break;
869     }
```

In Listing 4.1 wird der sogenannte „Fall-Through“-Effekt offensichtlich gezielt genutzt, um eine Wiederholung des Codes für die Fälle 10 bis 15 zu vermeiden.

Listing 4.2: stream.cpp

```
870      {
871      case U_a: case U_A:
872          return 10;
873      case U_b: case U_B:
874          return 11;
875      case U_c: case U_C:
876          return 12;
877      case U_d: case U_D:
878          return 13;
879      case U_e: case U_E:
880          return 14;
```

```
881     case U_f: case U_F:
882         return 15;
883     default:
884         return ch - U_0;
885     }
```

Listing 4.2 verzeichnet eine Reihe fehlender `break`-Anweisungen. Inwieweit dies beabsichtigt ist oder nicht, kann dem Code zwar nicht entnommen werden, ist aber für die Schädlichkeit der vorliegenden Verletzungen ohnehin irrelevant. Da die `return`-Anweisungen den Kontrollfluss unterbrechen, käme eine eventuell nachfolgende `break`-Anweisung nicht mehr zur Ausführung. Auch ein „Fall-Through"-Effekt stellt sich aus diesem Grund nicht ein. Entsprechend liegen hier zwar per Definition klare Verletzungen vor, es sind aber keine schädlichen Auswirkungen zu erwarten.
Die Verletzungen für den Indikator *informelle Dokumentation* erscheinen ebenfalls korrekt. Das bedeutet jedoch nicht zwingend, dass 4270 Methoden unkommentiert sind, sondern lediglich, dass die Kommentierung entweder nicht vor der Methode steht oder nicht mit `/*` eingeleitet wurde. Vielfach findet sich statt dessen ein einleitender Kommentar, der mit `//` eingeleitet wird.
Bei den Ergebnissen für die beiden Indikatoren *potenzielle Privatsphäre* fällt bei der Prüfung der Detaillisten auf, dass einige Verletzungen aufgelistet werden. Dies resultiert daraus, dass sie durch Templatevererbungen mehrfach im RFG enthalten sind.

Zusammenfassend ist festzuhalten, dass es offensichtliche Optimierungspotenziale hinsichtlich der Wartbarkeit von *jikes-1.22* gibt. Diese sind jedoch nicht ganz so groß, wie die Verletzungszahlen auf den ersten Blick suggerieren.

4.2.3 qt-1.41

Bei der hier analysierten QT-Bibliothek handelt es sich um eine Klassenbibliothek zur Programmierung grafischer Benutzeroberflächen. Die Bibliothek zeichnet sich vor allem dadurch aus, dass sie plattformübergreifend ist und Implementierungen in mehreren Programmiersprachen existieren. Die Vermessung der hier vorliegenden *C++*-Implementierung von Version 1.41 verlief ohne erkennbare Probleme und erreichte – wie aus Tabelle 4.6 ersichtlich – den Quality-Benchmark-Level 1.

Tabelle 4.6: Übersicht qt-1.41

Erreichter QBL	1
Analysezeit	687,97 Sekunden
Lines of Code	134.152
Dateien	303
Klassen	412

Die in Tabelle 4.7 abgebildeten Verletzungszahlen erscheinen nach stichprobenhafter Überprüfung durchaus plausibel.

Tabelle 4.7: Verletzungszahlen qt-1.41

Qualitätsindikator	Verletzungen	Qualitätsindikator	Verletzungen
allgemeine Parameter	7	nachlässige Kommentierung	55956.0
Attributüberdeckung	8	Namensfehler	34931
ausgeschlagenes Erbe (Implementierung)	39	Objektplacebo (Attribut)	16
ausgeschlagenes Erbe (Schnittstelle)	200	Objektplacebo (Methode)	5
Datenkapselaufbruch	408	Paketchen	2
duplizierter Code	1232	Pakethierarchieaufbruch	0
falsche Namenslänge	273	Polymorphieplacebo	0
Generationskonflikt	0	potenzielle Privatsphäre (Attribut)	75
Gottdatei	13	potenzielle Privatsphäre (Methode)	234
Gottklasse (Attribut)	0	pränatale Kommunikation	45
Gottklasse (Methode)	7	Risikocode	293
Gottmethode	17	signaturähnliche Klassen	130
Gottpaket	3	simulierte Polymorphie	0
halbherzige Operationen	66	späte Abstraktion	6
heimliche Verwandtschaft	32	tote Attribute	86
Identitätsspaltung	12	tote Implementierung	378
Importchaos	11	tote Methoden	196
Importlüge	0	überbuchte Datei	62
informelle Dokumentation	3310	unfertiger Code	11
Interface-Bypass	178	unvollständige Vererbung (Attribut)	0
Klasseninzest	6	unvollständige Vererbung (Methode)	135
Klässchen	30	verbotene Dateiliebe	101
Konstantenregen	107	verbotene Klassenliebe	19
Labyrinthmethode	0	verbotene Methodenliebe	8
lange Parameterliste	38	verbotene Paketliebe	7
maskierende Datei	41	versteckte Konstantheit	2

Wie bereits bei der Vermessung von *jikes-1.22* (vgl. Abschnitt 4.2.2) fällt bei der Betrachtung der Ergebnisse vor allem der sehr hohe Wert für den Indikator *nachlässige Kommentierung* auf. Dies beruht vor allem darauf, dass in den einzelnen Quellcode-Dateien zum Teil sehr viel auskommentierter Code zu finden ist, der zu einer „Überkommentierung“ des Systems führt. Diese ist aus den dargestellten Werten nicht von einer Unterkommentierung zu unterscheiden, da laut Indikator-Definition absolute Werte angegeben werden.
Der sehr hohe Wert des Indikators *Namensfehler* lässt darauf schließen, dass bei der Entwicklung dieser Software ein anderes Bennenungsschema verwendet wurde, als durch die Indikatordefinition vorgegeben wird.
Ebenfalls diskussionwürdig erscheinen bei näherer Betrachtung die – nach Indikatordefinition korrekten – Ergebnisse für *Importchaos*. Viele der hier monierten mehrfachen Includes

sind durch Präprozessoranweisungen eingeschlossen, so dass jeweils nur eine der inkludierenden Codezeilen zur Ausführung kommen sollte. Eine automatische Erkennung, inwieweit die jeweiligen Zeilen tatsächlich ausgeführt werden, ist allerdings im Rahmen einer statischen Analyse systembedingt nicht möglich.
Generell fällt auf, dass relativ viele Werte deutlich höher sind, als die Schwellenwerte für die einzelnen QBL erwarten lassen würden. So liegt der *Datenkapselaufbruch* mit einem normierten Wert von 3,0413 weit über dem Schwellwert für QBL 2 (1,3700).
Eine mögliche Erklärung hierfür wäre, dass es sich um eine frühe Version der Software handelt, die eher eine Art Meilenstein als eine endgültige Lösung darstellt.

4.2.4 qt-1.45

Berücksichtigt man die gestiegene Zahl an Codezeilen, unterscheiden sich die in den Tabellen (Tabelle 4.8 und Tabelle 4.9) dargestellten Ergebnisse erwartungsgemäß – da es sich nur um einen kleinen Versionssprung handelt – nur unwesentlich von den Ergebnissen für Version 1.41. Auffällig ist lediglich der deutliche Anstieg der Verletzungen beim Qualitätsindikator *duplizierter Code*. Gepaart mit der deutlich angestiegenen Anzahl der Dateien bei fast gleichbleibender Klassenzahl liegt hier die Vermutung nahe, dass der Anstieg auf Umstrukturierungen zurückzuführen ist. Ein Blick in die Verzeichnisstruktur des Systems bestätigt diesen Verdacht. Viele Header-Dateien im Verzeichnis `include` finden sich in einem oder mehreren Unterverzeichnissen von `src` namens- und inhaltsgleich wieder. Offenbar wurden hier – möglicherweise aus Gründen der Kompatibilität – nach der Umstrukturierung der Verzeichnisstruktur die ursprünglichen Header-Dateien nicht entfernt. Für eine eventuelle Weiterentwicklung oder Wartung stellt dies ein Risiko dar, weil vor Änderungen geprüft werden müsste, welche Headerdateien verwendet werden und ob diese auch tatsächlich die korrekte und aktuelle Version enthalten.

Tabelle 4.8: Übersicht qt-1.45

Erreichter QBL	1
Analysezeit	761,72 Sekunden
Lines of Code	142.614
Dateien	343
Klassen	413

Tabelle 4.9: Verletzungszahlen qt-1.45

Qualitätsindikator	Verletzungen	Qualitätsindikator	Verletzungen
allgemeine Parameter	7	nachlässige Kommentierung	61788.0
Attributüberdeckung	8	Namensfehler	34948
ausgeschlagenes Erbe (Implementierung)	39	Objektplacebo (Attribut)	16
ausgeschlagenes Erbe (Schnittstelle)	197	Objektplacebo (Methode)	4
Datenkapselaufbruch	410	Paketchen	1
duplizierter Code	31160	Pakethierarchieaufbruch	0
falsche Namenslänge	281	Polymorphieplacebo	0
Generationskonflikt	0	potenzielle Privatsphäre (Attribut)	85
Gottdatei	13	potenzielle Privatsphäre (Methode)	233
Gottklasse (Attribut)	0	pränatale Kommunikation	43
Gottklasse (Methode)	7	Risikocode	294
Gottmethode	16	signaturähnliche Klassen	119
Gottpaket	3	simulierte Polymorphie	0
halbherzige Operationen	66	späte Abstraktion	6
heimliche Verwandtschaft	32	tote Attribute	84
Identitätsspaltung	12	tote Implementierung	379
Importchaos	11	tote Methoden	196
Importlüge	0	überbuchte Datei	60
informelle Dokumentation	3297	unfertiger Code	11
Interface-Bypass	179	unvollständige Vererbung (Attribut)	0
Klasseninzest	6	unvollständige Vererbung (Methode)	135
Klässchen	31	verbotene Dateiliebe	116
Konstantenregen	107	verbotene Klassenliebe	19
Labyrinthmethode	0	verbotene Methodenliebe	8
lange Parameterliste	38	verbotene Paketliebe	8
maskierende Datei	44	versteckte Konstantheit	2

4.2.5 qt-2.0.2

Die Analyse dieser QT-Version verlief weitgehend problemlos, lediglich das Tool `clones` stürzte während der Analyse mit der Meldung `Segmentation Fault` ab. Entsprechend ist das Ergebniss für den Qualitätsindikator *duplizierter Code* mit größter Wahrscheinlichkeit unvollständig.
Der ermittelte QBL ist, wie schon in den vorhergehenden Versionen, 1.

Tabelle 4.10: Übersicht qt-2.0.2

Erreichter QBL	1
Analysezeit	1375,17 Sekunden
Lines of Code	213.500
Dateien	408
Klassen	649

Tabelle 4.11: Verletzungszahlen qt-2.0.2

Qualitätsindikator	Verletzungen	Qualitätsindikator	Verletzungen
allgemeine Parameter	16	nachlässige Kommentierung	102170.0
Attributüberdeckung	10	Namensfehler	43799
ausgeschlagenes Erbe (Implementierung)	116	Objektplacebo (Attribut)	4
ausgeschlagenes Erbe (Schnittstelle)	339	Objektplacebo (Methode)	22
Datenkapselaufbruch	635	Paketchen	2
duplizierter Code	6302	Pakethierarchieaufbruch	0
falsche Namenslänge	429	Polymorphieplacebo	194
Generationskonflikt	2	potenzielle Privatsphäre (Attribut)	358
Gottdatei	21	potenzielle Privatsphäre (Methode)	334
Gottklasse (Attribut)	1	pränatale Kommunikation	102
Gottklasse (Methode)	15	Risikocode	412
Gottmethode	24	signaturähnliche Klassen	226
Gottpaket	3	simulierte Polymorphie	0
halbherzige Operationen	139	späte Abstraktion	11
heimliche Verwandschaft	67	tote Attribute	147
Identitätsspaltung	25	tote Implementierung	464
Importchaos	19	tote Methoden	257
Importlüge	0	überbuchte Datei	90
informelle Dokumentation	5734	unfertiger Code	44
Interface-Bypass	150	unvollständige Vererbung (Attribut)	3
Klasseninzest	21	unvollständige Vererbung (Methode)	200
Klässchen	44	verbotene Dateiliebe	164
Konstantenregen	44	verbotene Klassenliebe	19
Labyrinthmethode	0	verbotene Methodenliebe	12
lange Parameterliste	168	verbotene Paketliebe	7
maskierende Datei	51	versteckte Konstantheit	35

Im Vergleich zu Version 1.45 fällt vor allem auf, dass die Verletzungszahlen (siehe Tabelle 4.11) in vielen Fällen deutlich angestiegen sind. Setzt man diese jedoch in Relation zur Systemgröße (siehe Tabelle 4.10) wird deutlich, dass sich die Werte in vielen Fällen nur sehr moderat in der zweiten oder dritten Nachkommastelle verschlechtert, stellenweise sogar deutlich verbessert haben. Als Beispiel für eine solche Verbesserung wäre etwa *Objektplacebo (Attribut)* zu nennen.

Der hohe Wert für *nachlässige Kommentierung* entsteht – wie schon in den Vorgängerversionen – eher durch eine Überkommentierung. In diesem Fall rührt dies weniger daher, dass viel auskommentierter Quellcode in den Dateien enthalten ist. Vielmehr wurde offenbar die Dokumentation der Bibliothek – die naturgemäß recht umfangreich ist – per Doxygen[1] direkt in den Quellcode geschrieben. Ob dies tatsächlich einen Qualitätsmangel darstellt ist zumindest diskutabel.
Besonders auffällig sind die stark angestiegenen Werte im Bereich der potenziellen Privatsphäre und bei *Polymorphieplacebo*. Im Bereich der potenziellen Privatsphäre sind die Ergebnisse wenig überraschend, da eine Bibliothek viele Funktionalitäten bereitstellt, die von ihr selbst nicht verwendet werden. Bei *Polymorphieplacebo* handelt es sich größtenteils um falsch potitive Treffer, die dadurch entstehen, dass die fragliche Methode in der erbenden Klasse mit gleicher Signatur in IML und RFG existiert, ohne dass sie im Quellcode explizit deklariert worden ist. Der Grund für dieses Verhalten ist unklar.
Zusammenfassend kann festgehalten werden, dass sich die Qualität gegenüber der Vorgängerversion nur unwesentlich in Details verändert hat, im Großen und Ganzen aber stabil blieb.

4.2.6 qt-2.1.1

Tabelle 4.12: Übersicht qt-2.1.1

Erreichter QBL	1
Analysezeit	2492,99 Sekunden
Lines of Code	271.329
Dateien	454
Klassen	869

Auch für *qt-2.1.1* steigen die absoluten Verletzungszahlen (Tabelle 4.13) gegenüber der Vorgängerversion deutlich an. In Relation zur Systemgröße bleibt die Qualität jedoch weitgehend stabil. Es sind weder überproportionale Verschlechterungen noch überproportionale Verbesserungen zu verzeichnen.

Lediglich der Qualitätsindikator *duplizierter Code* fällt auf 0 Verletzungen zurück. Dies ist aber nicht auf eine signifikante Verbesserung der Software zurück zu führen, sondern auf einen erneuten – aus einem Speicherzugriffsfehler resultierenden – Absturz von `clones`.

[1] `http://www.stack.nl/~dimitri/doxygen/`

Tabelle 4.13: Verletzungszahlen qt-2.1.1

Qualitätsindikator	Verletzungen	Qualitätsindikator	Verletzungen
allgemeine Parameter	12	nachlässige Kommentierung	135655.0
Attributüberdeckung	32	Namensfehler	54492
ausgeschlagenes Erbe (Implementierung)	132	Objektplacebo (Attribut)	4
ausgeschlagenes Erbe (Schnittstelle)	453	Objektplacebo (Methode)	29
Datenkapselaufbruch	980	Paketchen	2
duplizierter Code	0	Pakethierarchieaufbruch	0
falsche Namenslänge	533	Polymorphieplacebo	230
Generationskonflikt	6	potenzielle Privatsphäre (Attribut)	477
Gottdatei	27	potenzielle Privatsphäre (Methode)	443
Gottklasse (Attribut)	3	pränatale Kommunikation	114
Gottklasse (Methode)	20	Risikocode	650
Gottmethode	36	signaturähnliche Klassen	412
Gottpaket	3	simulierte Polymorphie	0
halbherzige Operationen	216	späte Abstraktion	12
heimliche Verwandschaft	72	tote Attribute	197
Identitätsspaltung	27	tote Implementierung	618
Importchaos	22	tote Methoden	409
Importlüge	0	überbuchte Datei	100
informelle Dokumentation	8042	unfertiger Code	52
Interface-Bypass	226	unvollständige Vererbung (Attribut)	4
Klasseninzest	15	unvollständige Vererbung (Methode)	246
Klässchen	67	verbotene Dateiliebe	210
Konstantenregen	49	verbotene Klassenliebe	22
Labyrinthmethode	0	verbotene Methodenliebe	14
lange Parameterliste	192	verbotene Paketliebe	7
maskierende Datei	60	versteckte Konstantheit	38

4.2.7 qt-2.2.4

Tabelle 4.14: Übersicht qt-2.2.4

Erreichter QBL	1
Analysezeit	3207,1 Sekunden
Lines of Code	337.914
Dateien	510
Klassen	1131

Erwartungsgemäß gestaltet sich das Ergebnis der Vermessung von *qt-2.2.4* ähnlich zu den vorherigen. Die absoluten Werte steigen proportional zur Systemgröße an. Auch der Absturz von `clones` wiederholt sich, weshalb der Wert für *duplizierter Code* vermutlich falsch ist. Die Einstufung der einzelnen Qualitätsindikatoren ändert sich lediglich für *versteckte Konstantheit*. Hier wird nun der Grenzwert für QBL 2 unterboten.

Tabelle 4.15: Verletzungszahlen qt-2.2.4

Qualitätsindikator	Verletzungen	Qualitätsindikator	Verletzungen
allgemeine Parameter	21	nachlässige Kommentierung	166476.0
Attributüberdeckung	35	Namensfehler	65107
ausgeschlagenes Erbe (Implementierung)	180	Objektplacebo (Attribut)	4
ausgeschlagenes Erbe (Schnittstelle)	534	Objektplacebo (Methode)	46
Datenkapselaufbruch	1107	Paketchen	2
duplizierter Code	0	Pakethierarchieaufbruch	0
falsche Namenslänge	641	Polymorphieplacebo	420
Generationskonflikt	7	potenzielle Privatsphäre (Attribut)	677
Gottdatei	40	potenzielle Privatsphäre (Methode)	544
Gottklasse (Attribut)	4	pränatale Kommunikation	138
Gottklasse (Methode)	25	Risikocode	949
Gottmethode	58	signaturähnliche Klassen	590
Gottpaket	4	simulierte Polymorphie	0
halbherzige Operationen	298	späte Abstraktion	16
heimliche Verwandschaft	84	tote Attribute	273
Identitätsspaltung	29	tote Implementierung	752
Importchaos	23	tote Methoden	474
Importlüge	0	überbuchte Datei	121
informelle Dokumentation	10819	unfertiger Code	77
Interface-Bypass	291	unvollständige Vererbung (Attribut)	5
Klasseninzest	39	unvollständige Vererbung (Methode)	245
Klässchen	99	verbotene Dateiliebe	242
Konstantenregen	49	verbotene Klassenliebe	24
Labyrinthmethode	0	verbotene Methodenliebe	17
lange Parameterliste	252	verbotene Paketliebe	17
maskierende Datei	65	versteckte Konstantheit	37

4.2.8 qt-2.3.2

Die meisten der hier abgedruckten Ergebnisse für *qt-2.3.2* unterscheiden sich nur minimal von denen der Vorgängerversion. Diese Unterschiede haben weder Änderungen an der QBL-Einstufung des Gesamtsystems noch an der Einstufung einzelner Indikatoren zur Folge. Da die Unterschiede der Versionen hinsichtlich ihrer Größe ebenfalls sehr klein sind, erscheint dies nicht überraschend. Positiv fällt in dieser Version der Indikator *infor-*

Tabelle 4.16: Übersicht qt-2.3.2

Erreichter QBL	1
Analysezeit	3389,05 Sekunden
Lines of Code	346.748
Dateien	510
Klassen	1156

Tabelle 4.17: Verletzungszahlen qt-2.3.2

Qualitätsindikator	Verletzungen	Qualitätsindikator	Verletzungen
allgemeine Parameter	20	nachlässige Kommentierung	173408.0
Attributüberdeckung	35	Namensfehler	65184
ausgeschlagenes Erbe (Implementierung)	175	Objektplacebo (Attribut)	4
ausgeschlagenes Erbe (Schnittstelle)	550	Objektplacebo (Methode)	50
Datenkapselaufbruch	1123	Paketchen	1
duplizierter Code	0	Pakethierarchieaufbruch	0
falsche Namenslänge	641	Polymorphieplacebo	420
Generationskonflikt	7	potenzielle Privatsphäre (Attribut)	679
Gottdatei	41	potenzielle Privatsphäre (Methode)	645
Gottklasse (Attribut)	4	pränatale Kommunikation	139
Gottklasse (Methode)	25	Risikocode	992
Gottmethode	64	signaturähnliche Klassen	590
Gottpaket	4	simulierte Polymorphie	0
halbherzige Operationen	300	späte Abstraktion	16
heimliche Verwandschaft	87	tote Attribute	292
Identitätsspaltung	30	tote Implementierung	454
Importchaos	25	tote Methoden	485
Importlüge	0	überbuchte Datei	116
informelle Dokumentation	10706	unfertiger Code	77
Interface-Bypass	300	unvollständige Vererbung (Attribut)	5
Klasseninzest	39	unvollständige Vererbung (Methode)	242
Klässchen	93	verbotene Dateiliebe	244
Konstantenregen	49	verbotene Klassenliebe	24
Labyrinthmethode	0	verbotene Methodenliebe	17
lange Parameterliste	272	verbotene Paketliebe	17
maskierende Datei	66	versteckte Konstantheit	37

melle Dokumentation auf, der deutlich bessere Werte als in den Vorgängerversionen zeigt. Da allerdings bereits in den Vorgängerversionen viele Methoden durchaus dokumentiert waren und die Verletzung nur dadurch zustande kam, dass die Dokumentation nicht unmittelbar vor der jeweiligen Methode stand, ist fraglich wieweit dies tatsächlich eine signifikante Verbesserung der Wartbarkeit darstellt.

4.2.9 Anonymes System aus dem Bereich der öffentlichen Verwaltung

Wie bereits in Abschnitt 3.5.2 erwähnt, war die Vermessung dieses Systems vornehmlich als Stresstest gedacht, um nachzuweisen, dass das implementierte Werkzeug auch reale Softwaresysteme signifikanter Größe verkraftet.
Sowohl das Vorgehen als auch die Ergebnisse dieser Vermessung werden im Folgenden – soweit dies ohne Verletzung von Verschwiegenheitsvereinbarungen möglich ist – dargelegt.
Bei dem vermessenen System handelt es sich um ein im produktiven Einsatz befindliches Softwaresystem aus dem Bereich der öffentlichen Verwaltung, mit einem Umfang von rund 700.000 Zeilen Visual-C++ Code. Durch die mitgelieferten Visual-Studio-Projekt-Dateien wurden allerdings nur rund 407.000 Codezeilen verwendet, so dass auch nur diese in IML und RFG überführt wurden. Untersuchungsgegenstand war also faktisch ein System mit 407.000 Codezeilen.

Durchgeführt wurde die Analyse auf einem PC mit Intel-Pentium-4-Prozessor und zwei Gigabyte Hauptspeicher. Als Betriebsystem kam Windows XP-SP2 zum Einsatz. Nachdem ein erster Versuch das System in eine IML und einen RFG zu überführen an den technischen Speichergrenzen des 32-Bit Systems scheiterte, wurde klar, dass es mit der verfügbaren Rechnerkonfiguration unmöglich war, eine IML des Gesamtsystems zu erstellen.
Stattdessen wurden die einzelnen Systemteile in einzelne IMLs überführt, aus denen dann RFGs generiert wurden, die zu einem RFG des Gesamtsystems zusammengelinkt werden konnten.
Es wurde also zeitgleich mit dem Nachweis, dass auch große Systeme analysiert werden können, der Nachweis erbracht, dass auch eine Vermessung auf Basis von mehreren IMLs – die jeweils nur ein Teilsystem repräsentieren – möglich ist. Durch die in Abschnitt 3.2.1 erwähnte Funktionalität zur Unterdrückung von doppelt ermittelten Verletzungen wird eine mögliche Verfälschung der Ergebnisse durch die Verwendung mehrerer IMLs verhindert.
Die Analyse nahm etwa 2 Tage in Anspruch. Diese – insbesondere im Vergleich zu den in Abschnitt 4.1.2 genannten Werten – extrem hohe Laufzeit fiel bereits während der Vermessung auf.
Ein Blick auf den Windows Task-Manager zeigte, dass die Speicherauslastung des Systems mit zeitweise fast vier Gigabyte deutlich über dem physikalisch verbauten Arbeitsspeicher lag und damit durch die Verwendung der Auslagerungsdatei zu einer hohen Festplattenaktivität führte. Die unerwartet hohe Laufzeit resultiert also sehr wahrscheinlich vor allem aus dem exzessiven Gebrauch der Auslagerungsdatei – und damit der vergleichsweise langsamen Festplatte.

Tabelle 4.18: Übersicht anonymes System

Programmiersprache	Visual C++
Codezeilen	407.000
Dateien	1.554
Klassen	13.882

Tabelle 4.18 gibt einen Überblick über die Eckdaten des Systems.
Die Ergebnisse der Analyse sind in Tabelle 4.19 aufgelistet. Die extrem hohe Anzahl an Klassen ergibt sich aus der Zählung aller Knoten vom Typ `Class` im RFG des Gesamtsystems. Ein sehr großer Teil dieser Klassen sind offenbar Template-Instanziierungen.
In Tabelle 4.19 sind die Ergebnisse der Analyse nach Qualitätsindikatoren aufgeschlüsselt wiedergegeben.

Tabelle 4.19: Verletzungszahlen des anonymen Systems

Qualitätsindikator	Verletzungen	Qualitätsindikator	Verletzungen
Attributüberdeckung	6	Namensfehler	54379
ausgeschlagenes Erbe (Implementierung)	16	Objektplacebo (Attribut)	0
ausgeschlagenes Erbe (Schnittstelle)	8	Objektplacebo (Methode)	6
Datenkapselaufbruch	3037	Paketchen	131
duplizierter Code	0	Pakethierarchieaufbruch	0
falsche Namenslänge	547	Polymorphieplacebo	0
Generationskonflikt	0	potenzielle Privatsphäre (Attribut)	80007
Gottdatei	0	potenzielle Privatsphäre (Methode)	6858
Gottklasse (Attribut)	136	pränatale Kommunikation	11
Gottklasse (Methode)	172	Risikocode	2268
Gottmethode	53	signaturähnliche Klassen	14636
Gottpaket	0	simulierte Polymorphie	2845
halbherzige Operationen	22	späte Abstraktion	112
heimliche Verwandtschaft	372	tote Attribute	21820
Identitätsspaltung	204	tote Implementierung	1570
Importchaos	0	tote Methoden	1483
informelle Dokumentation	0	überbuchte Datei	0
Interface-Bypass	1191	unfertiger Code	0
Klässchen	682	unvollständige Vererbung (Attribut)	3
Klasseninzest	37	unvollständige Vererbung (Methode)	387
Konstantenregen	125	verbotene Dateiliebe	0
Labyrinthmethode	0	verbotene Methodenliebe	89
lange Parameterliste	171	verbotene Paketliebe	0
maskierende Datei	0	versteckte Konstantheit	98832
nachlässige Kommentierung	0.0		

Die stichprobenartige Überprüfung dieser Ergebnisse legte den Schluß nahe, dass sie weitestgehend zutreffend sind und im Vergleich zu den bisher dargelegten Vermessungen nicht völlig aus dem Rahmen fallen. Einige Ausreißer – sowohl nach unten als auch nach oben – wie zum Beispiel bei der potenziellen Privatsphäre von Attributen und Methode, resultieren vermutlich vor allem daraus, dass in den Programmrepräsentationen teilweise generierter Code enthalten war, der offenbar einige bibliotheksartige Funktionalitäten bereitstellt, die wenig genutzt werden.
Weil durch einen Bedienungsfehler der Quellcode während der Vermessung nicht zugreifbar war, ist zu vermuten, dass die Verletzungszahlen für *duplizierten Code*, *informelle Dokumentation* und *nachlässige Kommentierung* nicht aus einem besonders sauber implementierten und gut kommentierten System resultieren, sondern schlicht aus dem Mangel an Daten, der dazu führte, dass mögliche Verletzungen nicht erkannt werden konnten. Dies zeigt jedoch, dass die implementierte Lösung auch dann zuverlässig arbeitet, wenn Teile der nötigen Informationen nicht zur Verfügung stehen, so dass auch hieraus ein Erkenntnissgewinn entstand.
Weil das analysierte System nur für eine kurze Zeitspanne zur Verfügung stand, musste die Vermessung mit einer relativ frühen – noch nicht ganz vollständigen – Version des implementierten Werkzeugs durchgeführt werden, so dass sich erwartungsgemäß durch die erwähnte stichprobenhafte Überprüfung der Ergebnisse auch einige fehlerbehaftete Ergebnisse offenbarten, die wertvolle Erkenntnisse für die weitere Optimierung im Hinblick auf die Ergebnisqualität lieferten.
Eine sicherlich interessante Wiederholung der Analyse war aufgrund der zeitlich eingeschränkten Verfügbarkeit des Systems leider nicht möglich.

Zusammenfassend hat die Vermessung dieses Systems nicht nur wie erwartet den Nachweis erbracht, dass mit der implementierten Lösung auch Softwaresysteme relevanter Größe vermessen werden können und dass dies in Anbetracht der Verletzungszahlen sinnvoll erscheint, sondern sogar noch über den eigentlichen Zweck der Vermessung hinausgehende Erkenntnissgewinne ermöglicht.

4.2.10 Fazit der Vermessungen

Aus den Ergebnissen der Vermessungen verschiedener Softwaresysteme geht klar hervor, dass das entwickelte Tool mit überschaubarem Aufwand in der Lage ist, Optimierungspotenziale hinsichtlich der Wartbarkeit von Softwaresystemen aufzudecken. Die Verletzungszahlen zeigen ebenfalls deutlich, dass diese Potenziale zumindest bei den hier untersuchten Systemen durchaus vorhanden sind.
Es muss allerdings einschränkend berücksichtigt werden, dass die ermittelten Ergebnisse nicht ohne nähere Untersuchung als Handlungsanweisung verwendet werden sollten, weil einige Verletzungen zwar objektiv dem – durch den Qualitätsindikator definierten – Problemmuster entsprechen, jedoch unter Umständen aus gutem Grund verursacht wurden. In solchen Fällen würde sich durch eine Änderung zwar die Verletzungszahl reduzieren, aber es ist nicht zwingend von einer Erhöhung der Wartbarkeit auszugehen. Zusätzlich wurden bei einzelnen Indikatoren in bestimmten Konstellationen Fehldiagnosen festgestellt, die bei einer Beurteilung der Qualität berücksichtigt werden sollten.
Ungeachtet dieser Einschränkungen, können unter Verwendung der konkreten Trefferzahlen schnell besonders kritische Bereiche identifiziert und mittels der Detaillisten konkrete Handlungsempfehlungen abgeleitet werden, deren Umsetzung in der Regel in einer Verbesserung der Wartbarkeit resultiert. Durch die dadurch verkürzten Wartungszyklen kann vermutlich mittelbar auch die Kunden- bzw. Nutzerzufriedenheit gesteigert werden, indem fehlerbereinigte Versionen und neue Funktionalitäten schneller zur Verfügung gestellt werden können.

KAPITEL 5

Diskussion und Reflexion

In den vorangegangenen Kapiteln wurden vor allem Fakten und Ergebnisse präsentiert. Aber bereits dabei deutete sich an, dass sowohl die vorgestellte Implementierung als auch der zugrundeliegende Qualitätsindex stellenweise Schwachstellen aufweisen und keineswegs als Allheilmittel zu betrachten sind.
Dieses Kapitel dient deshalb der kritischen Auseinandersetzung mit den Ergebnissen und Grundlagen dieser Arbeit.
Es werden zunächst die bisherigen Ergebnisse der Arbeit zusammengefasst und im Hinblick auf ihren Umfang und ihre Qualität diskutiert.
Anschließend erfolgt eine kritische Betrachtung des als Grundlage der Arbeit dienenden Code-Quality-Index. Hierbei werden neben der Aussagekraft des Index auch seine Grenzen betrachtet. Insbesondere werden anhand einiger Indikatoren exemplarisch mögliche Schwachstellen aufgezeigt, die sich zum Teil auch im Verlauf der Vermessungen in Kapitel 4 schon angedeutet haben.
Abschließend werden noch einige besondere Herausforderungen aufgezeigt, die sich während der Umsetzung der Arbeit präsentiert haben.

Kapitelinhalt

5.1 Implementierung

Wie aus den vorangegangenen Kapiteln hervorgeht, konnten alle 52 Qualitätsindikatoren erfolgreich – wenn auch in Einzelfällen mit leichten Einschränkungen – implementiert werden. Ausgehend von den Vermessungen in Abschnitt 4.2 kann angenommen werden, dass diese Einschränkungen – zumindest bei *C++*-basierten Systemen – nicht zu einer zu positiven Bewertung der vermessenen Systeme führen.
Da die in Abschnitt 3.5 dokumentierten Testergebnisse für den *C++*-Testcode durchweg positiv ausfallen, muss davon ausgegangen werden, dass die weniger positiven Testergebnisse für den *Java*-Testcode vor allem durch nicht dokumentierte Unterschiede in den Programmrepräsentationsformen entstehen. Trotz dieses Problems liefern immer noch deutlich mehr als die Hälfte der Qualitätsindikatoren verwertbare und weitgehend nachvollziehbare Ergebnisse. Berücksichtigt man den nicht marktreifen Entwicklungsstand der IML für *Java* ist dies ein durchaus positives Ergebniss, das durch zukünftige Weiterentwicklungen sicherlich noch weiter verbessert werden wird.
Auch die erstmals in Abschnitt 2.5.3 skizzierte Möglichkeit, die Vermessung nach dem Code-Quality-Index in einen täglichen Build-Prozess zu integrieren, erscheint angesichts der in Abschnitt 4.2 dokumentierten Laufzeiten der einzelnen Vermessungen durchaus realistisch. Dies trifft umso mehr zu, berücksichtigt man, dass die Vermessungen auf einem handelsüblichen Heim-PC durchgeführt wurden, während ein täglicher Build-Prozess im Umfeld der professionellen Softwareentwicklung üblicherweise auf einem wesentlich leistungsfähigeren Server abgearbeitet wird. Auch der vor allem durch die IML recht hohe Speicherbedarf relativiert sich in diesem Szenario.
Durch die implementierten Funktionalitäten kann die Bauhaus-Suite den Wartungsingenieur nun neben Hilfestellungen beim Programmverstehen und dem Rekonstruieren und Vergleichen von Architekturen zusätzlich dabei unterstützen, Schwachstellen in der Implementierung zu identifizieren und zu beheben. Im Rahmen eines Migrationsprojektes wäre es beispielsweise möglich, die Wartbarkeit des Quellcodes kontinuierlich zu überwachen und gegebenenfalls schnell zu ermitteln wo Änderungsbedarf besteht.
Die zusätzlich zur HTML-Ausgabe implementierte Ausgabe der Verletzungszahlen als XML-Datei erleichtert die Überführung in alternative Ausgabeformate und kann – wie in Anhang B angedeutet – auch zum direkten Vergleich verschiedener Messungen verwendet werden. Auch eine automatisierte Benachrichtigung der Entwickler bei Entstehung neuer Verletzungen wäre auf Basis dieser Ausgabeform denkbar.

Besondere Herausforderung bei der Implementierung

Selbstverständlich traten im Zuge der Implementierung stellenweise auch Schwierigkeiten auf, die es zu lösen galt.
Zumindest die vermutlich größte – in jedem Fall aber zeitaufwendigste – Herausforderung soll hier kurz vorgestellt werden.

Der IML Scripting Guide [AXI07b] beinhaltet zwar eine Dokumentation der verschiedenen in der IML enthaltenen Knoten-Typen, aber er gibt nur wenig Aufschluss darüber, wie ein bestimmter Sachverhalt in der IML modelliert ist. Dies musste in den meisten Fällen aufwendig ermittelt werden, indem der fragliche Sachverhalt in eine IML überführt wurde. Aus der so entstandenen IML musste dann vom fraglichen Ausgangsknoten (beispielsweise einer Klasse oder Methode) ausgehend die Struktur des Graphen ausgegeben werden, um die verwendeten Knoten-Typen und ihre Zusammenhänge zu bestimmen und daraus schließlich einen Algorithmus zu abstrahieren, der den jeweiligen Sachverhalt wiederspiegelt.
Dadurch gestaltete sich die Implementierung der IML-basierten Qualitätsindikatoren sehr aufwendig.
Für zukünftige Arbeiten kann in diesem Zusammenhang die IML-Knotenreferenz aus der Diplomarbeit von Philippe Maurice Schober [Sch07], die zum Zeitpunkt der Implementierung leider noch nicht verfügbar war, eine wertvolle Hilfe darstellen.

5.2 Qualität der Vermessungsergebnisse

Während sich der vorangegangene Abschnitt vor allem auf den Umfang und Nutzen der Implementierung konzentrierte, betrachtet dieser Abschnitt die Qualität der mit der Implementierung ermittelten Messergebnisse.

Zuerst muss man sich die Frage stellen, was ein korrektes Messergebniss überhaupt ausmacht. Betrachtet man alle Verletzungen als korrektes Ergebnis, die nach der jeweiligen Indikatordefinition eine Verletzung darstellen, sind die in Abschnitt 3.5 und Abschnitt 4.2 dokumentierten Messergebnisse sicherlich gut. Es wurden nur wenige „falsche“ Verletzungen ermittelt und vor allem für *C++* konnten auch die meisten tatsächlich vorhandenen Verletzungen bestimmt werden.
Andererseits fanden sich inbesondere in den Vermessungsergebnissen für die QT-Bibliotheken verschiedene Stellen, die zwar per Indikatordefinition eine Verletzung darstellen, aber offensichtlich bewusst und gezielt implementiert worden sind und keine erkennbaren negativen Auswirkungen auf die Wartbarkeit haben. Die Schädlichkeit einer solchen Verletzung ist sicherlich nur schwer argumentierbar. Diese quasi „erwünschten“ Verletzungen sind jedoch auf maschineller Ebene nicht von „unerwünschten“ Verletzungen zu unterscheiden und können somit nur ausgefiltert werden, wenn sie vorab bekannt sind. In diesem Fall könnten die Ergebnisse der Vermessung gegen eine Whitelist abgeglichen und bereinigt werden.
Aber auch abseits von „erwünschten“ Verletzungen ist die Qualität der Ergebnisse nur schwer zu beurteilen. Während fälschlicherweise ermittelte Verletzungen noch relativ leicht erkannt werden können, gestaltet sich die Entdeckung von nicht erkannten Verletzungen ungleich schwieriger. Hier bliebe letztlich nur der Abgleich von Ergebnissen eines manuellen Reviews mit den Ergebnissen der automatisierten Vermessung. Wobei auch dieser Weg fehleranfällig ist, da nicht garantiert werden kann, dass im Rahmen des manuellen Reviews sämtliche Ver-

letzungen erkannt worden sind.

Vor diesem Hintergrund erscheint es ratsam, die ermittelten Messwerte nicht als absolute Werte zu betrachten, sondern als eine Annäherung, auf deren Basis eine genaue Prüfung durchgeführt werden sollte. Man könnte sagen: Es werden keine tatsächlichen Schwachstellen, sondern potenzielle Schwachstellen ermittelt, die auf ihre Relevanz hin überprüft werden sollten.

5.3 Code-Quality-Index

Bislang wurde in dieser Arbeit vor allem die Umsetzung des Code-Quality-Index dokumentiert, ohne seinen Sinn kritisch zu hinterfragen. Nachdem im Zuge der Implementierung und Evaluierung einige Erfahrungen mit dem Index gewonnen werden konnten, wird dies nun nachgeholt.
Eine Betrachtung ausgewählter Indikatoren erfolgt in Abschnitt 5.4.

Die generelle Idee, die technische Qualität eines Softwaresystems anhand eines einzelnen Wertes auszudrücken, ist ohne Frage wünschenswert und dürfte insbesondere für die Managementebene hilfreich sein. Zu diesem Zweck die Klassifizierung in fünf verschiedene Level der deutschen Hotelzertifizierung zu adaptieren, erscheint durchaus nachvollziehbar und zweckmäßig. Allerdings wird dort mit vergleichsweise harten Kriterien gearbeitet, während sich die Kriterien (Schwellwerte) im Falle des Code-Quality-Index mit jedem vermessenen und in das Repository eingepflegte System grundlegend ändern können. Dadurch ist es theoretisch denkbar, dass durch das Einpflegen eines besonders guten Systems mehrere bereits vermessene Systeme deutlich abgewertet und plötzlich um einen oder mehrere QBL schlechter eingestuft werden. Im Vergleich zum gerade eingepflegten System ist diese Einstufung zwar zutreffend und korrekt, aber letztlich erweckt die Abstufung den Eindruck, dass die Qualität der Systeme nachgelassen hat.
Dieses Vorgehen ist in etwa mit einer Prüfungssituation vergleichbar, in der die ersten x Prüflinge nachträglich Notenabzüge hinnehmen müssen, wenn ein Prüfling ein außergewöhnlich gutes Ergebniss erzielt.
Es sollte allerdings beachtet werden, dass dieses Szenario sehr theoretischer Natur ist und in der Praxis – aufgrund der Struktur des Index, der zugrundeliegenden Anzahl von über 120 vermessenen Systemen und der nach QBL abgestuften Relevanz der einzelnen Indikatoren – für gewöhnlich nicht eintreten dürfte.
Der Verwender des Code-Quality-Index sollte dennoch im Auge behalten, dass die Qualitätseinstufung ausschließlich auf Basis des Vergleichs mit der vorhandenen, aber ausreichend breiten, Datenbasis erfolgt. Eine langfristig gleichbleibende Qualitätseinschätzung scheint aber vor dem Hintergrund dieser Möglichkeit nicht sichergestellt zu sein. Eine Integration des Index beispielsweise in Outsourcing-Verträge zur Festlegung der zu liefernden Qualität,

wie indirekt durch [SSM06, Seite 41] suggeriert, beinhaltet somit ein gewisses Risiko, da sich die Einstufung theoretisch jederzeit ändern kann.
Eine Unsicherheit in diesem Zusammenhang entsteht auch dadurch, dass die Definition des Index in [SSM06] keinen Aufschluß darüber gibt, auf welcher Auswahl die 120 initial vermessenen Systeme basieren. Wurden hier vornehmlich gute Systeme vermessen, handelt es sich hauptsächlich um Problemfälle, bei denen eher schwache Ergebnisse zu erwarten waren, oder fand eine repräsentative Auswahl statt?

Einen Schwachpunkt stellt die Klassifizierung nach absoluter Erfüllung aller Voraussetzungen dar. Ein Softwaresystem, das in 51 der 52 Qualitätsindikatoren keinerlei Verletzungen aufweist und somit die Grenzwerte für QBL 5 erfüllt, aber im 52sten Qualitätsindikator nur QBL 1 erreicht, wird unter Umständen als QBL 1 klassifiziert. Bei diesem 52sten Indikator könnte es sich beispielsweise um *Labyrinthmethode* handeln. In diesem Fall stellt sich die Frage: „Erhöht sich der Wartungsaufwand für dieses fiktive System, nur aufgrund einiger Methoden mit komplexem Kontrollfluss, wirklich derart, dass eine höhere Einstufung als QBL 1 nicht gerechtfertigt ist?“
In einem realen Projekt würde dieses Szenario vermutlich dazu führen, dass der fragliche Indikator von der Betrachtung ausgenommen oder modifiziert wird, weil er im konkreten Projektkontext offensichtlich ungeeignet ist.
Dennoch verdeutlicht dieses extreme Beispiel, dass die Einstufung in einen bestimmten Quality-Benchmark-Level nicht in jedem Fall zwingend auf ein schlecht wartbares System hindeutet. Deshalb sollte die QBL-Einstufung als Warnsignal verstanden werden, dessen Ursachen ergründet werden müssen, bevor eine endgültige Beurteilung des untersuchten Softwaresystems erfolgt.

Zusammenfassend, stellt der Code-Quality-Index vor allem ein adäquates Werkzeug dar, um sich möglichen Risiken zu nähern und eine Abschätzung der technischen Qualität zu erhalten. Dies bedeutet aber keineswegs, dass er als Allheilmittel zur Bestimmung der technischen Qualität einer Software eingesetzt werden kann. Dies ist auch, wie auf Seite 67 des Buches „Code-Quality-Management“ [SSM06] ausgeführt, nicht beabsichtigt.
Die Existenz vereinzelter Schwachstellen, die durch extreme, konstruierte Beispiele zum Tragen kommen können, ist bei einer automatischen Analyse, die auf eine Vielzahl von verschiedenen Systemen anwendbar sein soll, nicht zu vermeiden.
Somit bestätigt der Code-Quality-Index, trotz kleiner, systembedingter Schwächen, durchaus den ihm zugedachten Zweck.

5.4 Die Qualitätsindikatoren

Einige der durch den Code-Quality-Index definierten Qualitätsindikatoren erscheinen im Einzelfall durchaus diskussionswürdig. Insbesondere die Grenzwerte, über die eine Verletzung des jeweiligen Indikators bestimmt wird, wirken mangels einer Begründung teilweise beliebig.
Im Folgenden werden deshalb einige exemplarisch ausgewählte Qualitätsindikatoren kritisch betrachtet. Da es wenig Sinn hat Gegenvorschläge zu machen, die nicht wissenschaftlich begründet werden können, dient dieser Abschnitt vor allem als Denkanstoß, inwieweit die jeweiligen Indikatoren in der vorliegenden Form sinnvoll sind und wo gegebenenfalls noch zusätzlicher Forschungsbedarf bestehen könnte, um den jeweiligen Indikator zu optimieren.

5.4.1 falsche Namenslänge

Dass es grundsätzlich nötig ist, an einem bestimmten Punkt eine Grenze zu ziehen steht ausser Frage und so erscheint eine Diskussion darüber, ob die minimale Namenslänge mit zwei oder drei Zeichen festgelegt werden sollte wenig sinnvoll. Allerdings erscheint die maximale Grenze von 50 Zeichen recht beliebig und durchaus diskussionswürdig.
Die Frage, ob ein Bezeichnername mit 40 Zeichen wirklich wesentlich leichter les- und wahrnembar ist als einer mit 50 Zeichen, fällt sicherlich eher in den Bereich der Psychologie und soll hier auch gar nicht beantwortet werden. Betrachtet man jedoch, dass insbesondere in Unix-basierten Betriebssystemen vielfach noch immer mit Zeilenlängen von maximal 80 Zeichen gearbeitet wird, erscheint eine Namenslänge von 50 Zeichen zu hoch.

5.4.2 Gottdatei, -klasse und -paket

Die generelle Problematik der Qualitätsindikatoren *Gottdatei*, *Gottklasse (Attribut bzw. Methode)* und *Gottpaket* wird in der jeweiligen Beschreibung ausgeführt und durch Literaturangaben untermauert. Allerdings werden die in den Definitionen genannten Grenzwerte durch diese nicht zwingend begründet.
Die für den Indikator referenzierten *Code Conventions for the Java Programming Language* [SM99] trifft ihrerseits lediglich die Aussage „Files longer than 2000 lines are cumbersome and should be avoided“. Weshalb die Grenze bei 2000 Zeilen und nicht erst bei 3000 oder bereits bei 1500 festzusetzen ist, geht daraus nicht hervor.
Die Übersichtlichkeit einer Datei (oder auch einer Methode) hängt in starkem Maße davon ab, wie ihr Inhalt formatiert ist. Es ist durchaus möglich, bereits 50 Zeilen Quellcode auf eine Art und Weise zu formatieren, die den Aufwand sie zu lesen und zu verstehen stark erhöht. Das Auffinden bestimmter Stellen wird von modernen Entwicklungsumgebungen in der Regel durch verschiedene Funktionalitäten unterstützt und ist ohne diese Unterstützung auch bei deutlich weniger als 2000 Codezeilen keine schöne Aufgabe.
Vor diesem Hintergrund wird diese Grenze wenig verständlich.

Auch bei den anderen drei Qualitätsindikatoren erscheint es wenig einsichtig, weshalb die Grenze des vertretbaren Aufwands ausgerechnet bei 50 umschlossenen Artefakten überschritten wird.

5.4.3 informelle Dokumentation

Zusätzlich zu der bereits in der Indikatorbeschreibung [SSM06, Seite 222] erwähnten Problematik, dass es unmöglich ist festzustellen, ob der vorhandene formelle Kommentar auch sinnvolle Inhalte enthält, wirft dieser Indikator weitere Fragen auf.
Ohne Frage ist es sinnvoll die Funktionsweise von Methoden zu dokumentieren und eine gut lesbare API-Dokumentation kann grade bei Wartungsarbeiten eine unschätzbare Hilfe sein. Aber in den meisten Fällen sollte eine Spezifikation, was eine Methode tut und gegebenenfalls auch wie sie es tut, schon vor der Implementierung existieren, so dass es keinen Bedarf an einer auf Grundlage von formellen Kommentierungsmechanismen generierten Dokumentation gibt. Da die beiden Dokumentationsformen weitgehend inhaltsgleich wären, bestünde hier vielmehr die Gefahr, dass bei Änderungen nur eine der beiden Dokumentationen angepasst wird. Die Folge könnte sein, dass viel Aufwand investiert werden muss um zu prüfen, welche der beiden Dokumentationen den aktuellen Stand wiederspiegelt. Das Ergebniss dieser Prüfung könnte im Extremfall von Methode zu Methode unterschiedlich ausfallen und so zusätzliche Aufwände nach sich ziehen.

Eine weitere Frage wäre, ob eine nicht formell kommentierte „Getter-“ oder „Setter-“Methode eine ebenso schwerwiegende Verletzung darstellt, wie eine äußerst komplexe Methode. Beim Vergleich verschiedener Softwaresysteme mit unterschiedlichen Anwendungsszenarien kann es dazu kommen, dass ein – ansonsten untadeliges – System mit vielen unkommentierten „Getter-“ und „Setter-“Methoden schlechter eingestuft wird als ein System, in dem zwar alle „Getter-“ und „Setter-“Methoden entsprechend kommentiert sind, die Methoden, die die Hauptfunktionalitäten bereitstellen jedoch völlig unkommentiert sind.

5.4.4 Klässchen

Häufig werden kleine Klassen verwendet um elementare Datentypen, auf denen weitaus komplexere Typen aufbauen, zu realisieren. Ein einfaches Beispiel wäre etwa die einfach verkettete Liste, wie sie von Ottmann und Widmayer in „Algorithmen und Datenstrukturen“ vorgestellt wird [OW02, Seiten 25ff.]. Die eigentliche Liste setzt sich hier aus Instanzen eines Datentyps zusammen, der eine Variable für die Daten und einen Zeiger auf das nächste Listen-Element enthält. Eine entsprechende Klasse bestünde also aus nur zwei Attributen.
Hier pauschal anzunehmen, dass die Klasse keine Daseinsberechtigung hat und in einer größeren integriert werden könnte, erscheint gewagt.

5.4.5 nachlässige Kommentierung

Es erscheint fragwürdig, den optimalen Kommentierungsgrad eines Systems pauschal festzulegen.
Ein System mit eher trivialen Funktionalitäten und sprechenden Bezeichnern wird vermutlich mit einer deutlich niedrigeren Kommentierungsrate als gut verständlich und wartbar empfunden werden, als ein System mit sehr komplexen Funktionalitäten und kryptischen Bezeichnern. Auch die Verwendung oder Nichtverwendung von Frameworks kann diesen Wert sicherlich beeinflussen.
Allerdings ist auf Basis einer automatischen Analyse kaum zu ermitteln, um welchen Typ von System es sich bei dem Analysierten handelt. Es bleibt die Frage, ob ein Verhätltnis von eins zu eins zwischen Code- und Kommentarzeilen wirklich in der Mehrzahl der Fälle zu einem annähernd optimal kommentierten System führt.

5.4.6 potenzielle Privatsphäre (Methode)

Dieser Indikator wies vor allem während der Vermessung der QT-Bibliotheken durchweg hohe Werte auf. Der naheliegende Schluß ist, dass hier viele Methoden als `protected` deklariert wurden, um eine Spezialisierung durch, die Bibliothek verwendenden, Anwendungen zu ermöglichen.
Vor diesem Hintergrund erscheint es sinnvoll, diesen Indikator bei zukünftigen Vermessungen von Bibliotheken und Frameworks außen vor zu lassen.

5.4.7 signaturähnliche Klassen

Es ist anzunehmen, dass Klassen mit vielen „Getter-“ und „Setter-“Methoden sehr schnell Verletzungen dieses Qualitätsindikators verursachen, die nur schwer vermeidbar sind. Hier könnte es sich lohnen zu untersuchen, ob sich dieser Effekt bei Systemen ab einer gewissen Größe selbsttätig auflöst.

5.4.8 Fazit

Die für den Code-Quality-Index definierten Qualitätsindikatoren müssen auf eine Vielzahl von Softwaresystemen mit sehr unterschiedlichen Anwendungsszenarien anwendbar sein, um eine Vergleichbarkeit der technischen Qualität dieser Softwaresysteme zu gewährleisten.
Dass hierbei nicht jedes einzelne Szenario berücksichtigt werden kann, ist selbstverständlich. So verwundert es nicht, dass sich zu einzelnen Qualitätsindikatoren durchaus Gegenbeispiele finden lassen, in denen eine Verletzung keine Einschränkung der Wartbarkeit mit sich bringt. Die stellenweise sehr hohen Grenzwerte für einzelne Verletzungen legen den Schluss nahe, dass sie bewusst hoch angesetzt wurden, um falsch Positive so weit wie möglich auszuschließen.

KAPITEL 6

Zusammenfassung und Ausblick

Die vorangegangenen Kapitel haben sich der Beschreibung der Grundlagen, des Lösungsansatzes und der Umsetzung sowie der Evaluierung und Bewertung der Arbeitsergebnisse gewidmet. Dieses abschließende Kapitel liefert neben einer Zusammenfassung der Ergebnisse einen Ausblick auf mögliche Erweiterungen und Verbesserungen sowohl der Implementierung als auch des zugrundeliegenden Code-Quality-Index.

Die Implementierung und Evaluierung eines Werkzeugs zur automatisierten Ermittlung des im Buch „Code-Quality-Management“ [SSM06] definierten Code-Quality-Index im Kontext des Bauhaus-Projekts konnte im Rahmen dieser Arbeit weitgehend erfolgreich durchgeführt werden.
Für *C++* können alle 52 Qualitätsindikatoren mit minimalen Einschränkungen bestimmt und ausgewertet werden. Für *Java* gilt dies nur für rund die Hälfte der Indikatoren, was vor allem an nicht dokumentierten Unterschieden zwischen den Programmrepräsentationen für *C++* und *Java* liegt. Da es innerhalb des Bauhaus-Projekts beständige Bestrebungen gibt die diesbezügliche Situation zu verbessern, ist damit zu rechnen, dass sich diese Ergebnisse in Zukunft quasi automatisch verbessern werden.
Die Präsentation der Ergebnisse in Form von generierten HTML-Seiten hat sich im Verlauf der Evaluierung als durchaus nützlich erwiesen. Als zukünftige Erweiterung bietet sich eine noch engere Integration der Ergebnisse in den RFG und damit einhergehend eine Visualisierung der Vermessungsergebnisse – beispielsweise im Rahmen einer zukünftigen Diplomarbeit – in Gravis an.
Sofern an der HTML-Ausgabe festgehalten wird, wäre es wünschenswert, in die Detail-Seiten eine kurze Beschreibung des jeweiligen Qualitätsindikators zu integrieren, um die Verständlichkeit zu erhöhen.
Der der Arbeit zugrundeliegende Code-Quality-Index hat sich als geeignet erwiesen, um die Qualität eines Softwaresystems mit vertretbarem Aufwand näherungsweise zu bestimmen. Da es sich um eine automatische Analyse handelt, die mögliche semantische Schwächen anhand von vornehmlich syntaktischen Merkmalen aufdecken soll, kommt es hierbei systembedingt zu gewissen Ungenauigkeiten und Fehlern. So zeigte sich unter anderem, dass sich der Index für die Vermessung von Bibliotheken nur bedingt eignet, da diese aufgrund der bereitgestellten aber nicht genutzten Funktionalität eher schwache Einstufungen erhalten.

Wie die Vermessungen in Kapitel 4 gezeigt haben, wäre in der praktischen Anwendung des Index eine Erweiterung desselben sowie der Implementierung um eine *Whitelist* sinnvoll, die „erlaubte“ Verletzungen enthält. Hierbei müsste allerdings berücksichtigt werden, dass das Nichtbeachten von Verletzungen die Ergebnisse verfälscht und damit die Vergleichbarkeit beeinträchtigen könnte.

Insgesamt werden durch die Ergebnisse dieser Arbeit die Fähigkeiten der Bauhaus-Suite, einen Wartungsentwickler in seinen Arbeitsabläufen zu unterstützen, deutlich erweitert. Durch die Vermessung des zu wartenden Software-Systems erhält er nicht nur eine Abschätzung der internen Qualität des Systems, die gegebenenfalls Einfluss auf das Budget und den Zeitplan nimmt, sondern es werden auch konkrete Risikopotenziale aufgedeckt, aus denen bei Bedarf konkrete Handlungsanweisungen abgeleitet werden können.

ABBILDUNGSVERZEICHNIS

TABELLENVERZEICHNIS

LITERATURVERZEICHNIS

[AXI07a] AXIVION: *Axivion Guide 5.5.* Axivion GmbH, 2007.

[AXI07b] AXIVION: *IML Scripting Guide 5.5.* Axivion GmbH, 2007.

[Bar98] Barnes, John: *Programming in Ada95.* Pearson Education Limited, Harlow, Essex, 2. Auflage, 1998.

[Bec06] Beckwermert, Felix: *Visualisierung von Software-Klonerkennung.* Diplomarbeit, Universität Bremen, Fachbereich 3: Mathematik / Informatik, 2006.

[Bel02] Bellon, Stefan: *Vergleich von Techniken zur Erkennung duplizierten Quellcodes.* Diplomarbeit, Universität Stuttgart, Institut für Informatik, 2002.

[BW84] Basili, Victor R. und David M. Weiss: *A Methodology for Collecting Valid Software Engineering Data.* IEEE Trans. Software Eng., 10(6):728–738, 1984.

[FP96] Fenton, Norman E. und Shari Lawrence Pfleeger: *Software Metrics - A Rigorous & Practical Approach.* International Thomson Computer Press, London, 2. Auflage, 1996.

[Hal77] Halstead, Maurice H.: *Elements of Software Science.* Elsevier, New York, 1977.

[Kna02] Knauss, Markus: *Erweiterung und Generierung der Zwischendarstellung IML für Java-Programme.* Diplomarbeit, Universität Stuttgart, Institut für Informatik, 2002.

[KS03] Karaca, Tahir und Sebastian Setzer: *Erweiterung und Generierung der Zwischendarstellung IML für C++ Programme.* Diplomarbeit, Universität Stuttgart, Institut für Informatik, 2003.

[KV07] Kuipers, Tobias und Joost Visser: *Maintainability Index Revisited - position paper -.* Eleventh European Conference on Software Maintenance and Reengineering (CSMR 2007), März 2007.

[LL07] Ludewig, Jochen und Horst Lichter: *Software Engineering - Grundlagen, Menschen, Prozesse, Techniken.* dpunkt.verlag, Heidelberg, 1. Auflage, 2007.

[McC76] McCabe, T.: *A Software Complexity Meassure.* IEEE Trans. Software Eng., 2(6):308–320, Dezember 1976.

[MRW77] McCall, J. A., P.K. Richards und G.F. Walters: *Factors in Software-Quality*. Rome Air development, Rom, 1977.

[Mye77] Myers, Glenford J.: *An extension to the cyclomatic measure of program complexity*. SIGPLAN Not., 12(10):61–64, 1977.

[OW02] Ottmann, T. und P. Widmayer: *Algorithmen und Datenstrukturen*. Spektrum Akademischer Verlag GmbH, Heidelberg / Berlin, 4. Auflage, 2002.

[pyt07] *Python (Programmiersprache)*. Wikipedia, letzter Abruf: 16.08 2007. `http://de.wikipedia.org/wiki/Python_%28Programmiersprache%29`.

[qbe07] *QBench-Projekt*. Webseite, letzter Abruf: 03.08 2007. `http://www.qbench.de`.

[Sch07] Schober, Philippe Maurice: *Konzeption und Implementierung eines Interpreters für die sprachübergreifende Programmrepräsentation IML*. Diplomarbeit, Universität Bremen, Fachbereich 3: Mathematik / Informatik, 2007.

[SM99] Sun-Microsystems: *Code Conventions for the Java Programming Language*. Webseite, April 1999. letzter Abruf: 28.09.2007, `http://java.sun.com/docs/codeconv/html/CodeConvTOC.doc.html`.

[Som07] Sommerville, Ian: *Software Engineering 8*. Addison-Wesley, 8. Auflage, 2007.

[SSM06] Simon, Frank, Olaf Seng und Thomas Mohaupt: *Code-Quality-Management - Technische Qualität industrieller Softwaresysteme transparent und vergleichbar gemacht*. dpunkt.verlag, Heidelberg, 1. Auflage, 2006.

[WOA97] Welker, Kurt D., Paul W. Oman und Gerald G. Atkinson: *Development and Applikation of an automated Source Code Maintainability Index*. Journal of Software Maintenance, 9(3):127–159, Mai 1997.

ANHANG A

Schwellwerte

Die in diesem Anhang aufgelisteten Schwellwerte für die Bestimmung des Quality-Benchmark-Level sind Kapitel 10 in [SSM06] entnommen.

Tabelle A.1: QBL Schwellwerte für C++

	Schwellwert für			
Qualitätsindikator	QBL 2	QBL 3	QBL 4	QBL 5
allgemeine Parameter	0,0000	0,0000	0,0000	0,0000
Attributüberdeckung	0,0360	0,0130	0,0040	0,0000
ausgeschlagenes Erbe (Implementierung)	0,1390	0,0360	0,0120	0,0000
ausgeschlagenes Erbe (Schnittstelle)	0,0920	0,0010	0,0000	0,0000
Datenkapselaufbruch	1,3700	0,8680	0,1810	0,0000
duplizierter Code	30,3570	17,3370	15,9690	4,7610
falsche Namenslaenge	0,1220	0,0800	0,0320	0,0000
Generationskonflikt	0,0080	0,0000	0,0000	0,0000
Gottdatei	0,0590	0,0110	0,0010	0,0000
Gottklasse (Attribut)	0,0070	0,0000	0,0000	0,0000
Gottklasse (Methode)	0,0460	0,0310	0,0150	0,0000
Gottmethode	0,0680	0,0440	0,0120	0,0000
Gottpaket	0,0170	0,0100	0,0070	0,0000
halbherzige Operationen	0,3790	0,0000	0,0000	0,0000
heimliche Verwandschaft	0,0840	0,0350	0,0230	0,0000
Identitaetsspaltung	0,0450	0,0030	0,0000	0,0000
Importchaos	0,0000	0,0000	0,0000	0,0000
Importluege	0,0000	0,0000	0,0000	0,0000
informelle Dokumentation	14,2160	14,0410	13,6430	13,6080
Interface-Bypass	1,0070	0,5020	0,1430	0,0000
Klaesschen	0,4080	0,2430	0,1080	0,0000
Klasseninzest	0,0160	0,0030	0,0010	0,0000
Konstantenregen	0,0040	0,0010	0,0000	0,0000
Labyrinthmethode	1,1100	0,2500	0,1410	0,0450
lange Parameterliste	0,0720	0,0460	0,0300	0,0000

Tabelle A.1: QBL Schwellwerte für C++

Qualitätsindikator	Schwellwert für QBL 2	QBL 3	QBL 4	QBL 5
maskierende Datei	2,5410	1,8420	1,1490	0,0000
nachlaessige Kommentierung	370,2640	6,2720	1,4530	0,4910
Namensfehler	22,2680	14,0300	5,2950	1,1460
Objektplacebo (Attribut)	0,9950	0,0100	0,0000	0,0000
Objektplacebo (Methode)	1,1290	0,2780	0,0150	0,0000
Paketchen	0,0170	0,0100	0,0000	0,0000
Pakethierarchieaufbruch	0,0000	0,0000	0,0000	0,0000
Polymorphieplacebo	0,0800	0,0120	0,0000	0,0000
potenzielle Privatsphaere (Attribut)	5,4880	2,2880	1,0660	0,0440
potenzielle Privatsphaere (Methode)	1,2360	0,0000	0,0000	0,0000
praenatale Kommunikation	0,0450	0,0150	0,0070	0,0000
Risikocode	0,1340	0,0370	0,0000	0,0000
signaturaehnliche Klassen	2,4100	0,2380	0,1100	0,0000
simulierte Polymorphie	0,0000	0,0000	0,0000	0,0000
spaete Abstraktion	0,0090	0,0000	0,0000	0,0000
tote Attribute	0,9470	0,5850	0,2580	0,0010
tote Implementierung	0,0000	0,0000	0,0000	0,0000
tote Methoden	2,1610	1,0780	0,3010	0,0210
ueberbuchte Datei	0,2280	0,1340	0,0460	0,0000
unfertiger Code	1,1520	0,4060	0,2310	0,0000
unvollstaendige Vererbung (Attribut)	0,5400	0,1810	0,1380	0,0680
unvollstaendige Vererbung (Methode)	1,2500	0,5420	0,3660	0,0310
verbotene Dateiliebe	0,5610	0,2940	0,0640	0,0070
verbotene Klassenliebe	0,2450	0,1750	0,0870	0,0000
verbotene Methodenliebe	0,0200	0,0080	0,0000	0,0000
verbotene Paketliebe	0,1150	0,0660	0,0000	0,0000
versteckte Konstantheit	0,1360	0,0330	0,0080	0,0000

Tabelle A.2: QBL Schwellwerte für Java

Qualitätsindikator	Schwellwert für QBL 2	QBL 3	QBL 4	QBL 5
allgemeine Parameter	2,2260	1,2010	0,7200	0,0000
Attributueberdeckung	0,0820	0,0200	0,0000	0,0000
ausgeschlagenes Erbe (Implementierung)	0,4160	0,1890	0,1100	0,0000
ausgeschlagenes Erbe (Schnittstelle)	1,0840	0,5100	0,1270	0,0000
Datenkapselaufbruch	0,4040	0,0500	0,0000	0,0000
duplizierter Code	80,4740	25,3920	8,1620	0,0000
falsche Namenslaenge	0,8780	0,1370	0,0410	0,0000
Generationskonflikt	0,0120	0,0000	0,0000	0,0000
Gottdatei	0,0650	0,0250	0,0000	0,0000
Gottklasse (Attribut)	0,0000	0,0000	0,0000	0,0000
Gottklasse (Methode)	0,1080	0,0590	0,0000	0,0000
Gottmethode	0,1030	0,0430	0,0000	0,0000
Gottpaket	0,0800	0,0120	0,0000	0,0000
halbherzige Operationen	0,0280	0,0000	0,0000	0,0000
heimliche Verwandschaft	0,4860	0,2570	0,1730	0,0000
Identitaetsspaltung	0,1220	0,0730	0,0070	0,0000
Importchaos	5,0490	1,3290	0,4280	0,0000
Importluege	2,3210	1,3270	0,6240	0,0000
informelle Dokumentation	19,0120	14,2650	9,8300	0,5920
Interface-Bypass	1,9640	1,1110	0,1770	0,0000
Klaesschen	1,4570	1,0760	0,7630	0,0000
Klasseninzest	0,0550	0,0200	0,0000	0,0000
Konstantenregen	0,3780	0,1330	0,0640	0,0000
Labyrinthmethode	0,7610	0,4950	0,2980	0,0000
lange Parameterliste	0,1620	0,0830	0,0200	0,0000
maskierende Datei	0,0000	0,0000	0,0000	0,0000
nachlaessige Kommentierung	339,5540	199,8100	57,6530	3,7720
Namensfehler	6,1000	3,3560	1,2630	0,0000
Objektplacebo (Attribut)	8,0620	3,8410	0,1500	0,0000
Objektplacebo (Methode)	1,2460	0,1000	0,0240	0,0000
Paketchen	0,2250	0,1250	0,0550	0,0000
Pakethierarchieaufbruch	0,0150	0,0000	0,0000	0,0000
Polymorphieplacebo	0,0080	0,0000	0,0000	0,0000
potenzielle Privatsphaere (Attribut)	1,4610	0,9490	0,3220	0,0000
potenzielle Privatsphaere (Methode)	1,9600	1,1610	0,7350	0,0000
praenatale Kommunikation	0,5500	0,2440	0,1170	0,0000
Risikocode	0,9460	0,4150	0,2160	0,0000

Tabelle A.2: QBL Schwellwerte für Java

Qualitätsindikator	Schwellwert für QBL 2	QBL 3	QBL 4	QBL 5
signaturaehnliche Klassen	1,2840	0,3550	0,0870	0,0000
simulierte Polymorphie	0,7800	0,3730	0,1380	0,0000
spaete Abstraktion	0,0440	0,0140	0,0000	0,0000
tote Attribute	0,5830	0,1630	0,0610	0,0000
tote Implementierung	0,0000	0,0000	0,0000	0,0000
tote Methoden	0,5130	0,2640	0,1130	0,0000
ueberbuchte Datei	0,0370	0,0100	0,0000	0,0000
unfertiger Code	1,0320	0,5530	0,0240	0,0000
unvollstaendige Vererbung (Attribut)	0,6950	0,4160	0,1850	0,0000
unvollstaendige Vererbung (Methode)	1,3660	0,9320	0,3880	0,0000
verbotene Dateiliebe	0,7880	0,3610	0,0790	0,0000
verbotene Klassenliebe	0,8070	0,3380	0,0730	0,0000
verbotene Methodenliebe	0,0440	0,0130	0,0000	0,0000
verbotene Paketliebe	0,2120	0,1210	0,0490	0,0000
versteckte Konstantheit	0,9150	0,5030	0,0900	0,0000

ANHANG B

Durchführung einer Vermessung am Beispiel des *C++*-Testcodes

Dieser Anhang stellt am Beispiel des *C++*-Testcodes dar, wie eine Vermessung durchgeführt wird und erläutert in diesem Zusammenhang auch die Kommandozeilenparameter des implementierten Werkzeugs.
Da die Entwicklung der Bauhaus Suite vornehmlich auf Unix-basierten Betriebssystemen durchgeführt wird, beschränkt sich die folgende Anleitung auf eine Vermessung mit einem Unix-basierten System. Die Vermessung unter Windows durchzuführen ist prinzipiell möglich, allerdings werden hier zusätzliche Schritte zur korrekten Einrichtung der Bauhaus Suite notwendig. Entsprechende Anweisungen sind dem Axivion Guide [AXI07a] zu entnehmen.

Der erste Schritt besteht darin, die notwendigen Anwendungen einzurichten. Dies sind Python, webcpp und die Bauhaus Suite. Im Falle der Bauhaus Suite ist darauf zu achten, dass in der verwendeten Lizenz die zu analysierende Sprache, das IML-Scripting-Interface und das Axivion Source-Metrics-PlugIn enthalten sind.
Die folgende Beschreibung basiert auf Bauhaus-Version 5.5.0. Für neuere Versionen müssen die Befehle möglicherweise angepasst werden.
Das dem Testcode beiliegende Makefile ist bereits für die Übersetzung mit Bauhaus angepasst und verwendet als Compiler den Befehl `cafeCC -B. -no_strip`. Wenn der Speicher eines Rechners zu knapp bemessen ist, kann der Parameter `-no_strip` testweise entfernt werden. Hierdurch gehen allerdings unter Umständen Informationen verloren, die das Vermessungsergebniss beeinflussen können.
Zusätzlich zum Makefile liegt dem Testcode eine Stapelverarbeitungsdatei bei, die die folgenden Befehle enthält:

Listing B.1: Batch-Datei

```
1 make
2 iml_optimizer QBenchCppTestcode.exe QBenchCpp.iml
3 iml2rfg QBenchCpp.iml QBenchCpp.rfg -B. -lib /usr
4 imlmetrics -rfg QBenchCpp.rfg QBenchCpp.iml
5 make clean
```

1. Der Befehl `make` generiert eine IML-Datei namens `QBenchCppTestcode.exe`
2. Dieser Befehl nimmt einige Optimierungen an der zuvor generierten IML vor. Vor allem werden doppelt vorhandene Knoten entfernt und dadurch der Speicherbedarf der IML gesenkt.
3. In einem dritten Schritt erfolgt die Überführung der IML in einen RFG. Als Bibliothekspfad wird das Verzeichnis `/usr` angegeben, um die *C++-Standard-Template-Library* von der Vermessung auszunehmen.
4. Das Programm `imlmetrics` berechnet einige von Bauhaus mitgelieferte Metriken. Der Schalter `-rfg` weist das Programm an, die Ergebnisse der Metriken im angegebenen RFG zu annotieren.
5. Schließlich werden die im Zuge der IML-Generierung erstellten Objektdateien gelöscht. Dieser Schritt hat keinen Einfluss auf die Vermessung, sondern dient lediglich der Einsparung von Speicherplatz auf der Festplatte.

Nach diesen Vorarbeiten kann die eigentliche Vermessung erfolgen, die in diesem Beispiel mit folgendem Befehl angestoßen wird:

```
rfgscript code-quality-analysis.py QBenchCpp.iml QBenchCpp.rfg
    --lib /usr -l cpp -x -s -o ~/ausgabepfad
```

Der Aufruf enthält das Hauptskript `code-quality-analysis.py` sowie die zuvor generierte IML und den RFG. Bei `rfgscript` handelt es sich um den Bauhaus-eigenen Interpreter für die IML- und RFG-Scripting-Schnittstelle. Zusätzlich kann noch eine Reihe von Parametern übergeben werden, deren Bedeutung in der folgenden Aufzählung erklärt ist:

-s Die Annotierungen der Metriken werden im RFG gespeichert und können beispielsweise mit `Gravis` jederzeit eingesehen werden.

-x Zusätzlich zum HMTL-Report wird eine XML-Datei mit Namen `results.xml` erzeugt, die neben den Eckdaten des analysierten Systems die absoluten Verletzungszahlen pro Qualitätsindikator enthält. Sie wird im gleichen Pfad wie der HTML-Report abgelegt.

-o Hiermit kann ein alternativer Pfad für die Ausgabe angegeben werden. Ist dieser Schalter nicht gesetzt wird wird `USERHOME/analysis-output` als Defaultwert verwendet.

-l Wird verwendet um die analysierte Sprache mitzuteilen. Mögliche Werte sind `cpp` und `java`. Wird der Schalter nicht gesetzt, geht das Programm davon aus, dass *C++* analysiert wird.

--lib Wie bereits bei der Überführung der IMl zum RFG kann hierüber ein Verzeichnis definiert werden, in dem sich Bibliothekselemente befinden, die nicht in der Vermessung berücksichtigt werden sollen.
Dieser Schalter kann mehrfach verwendet werden.

Nach erfolgter Durchführung der Vermessung ist der resultierende Bericht im angegebenen Verzeichnis zu finden. Als Einstiegspunkt dient hier üblicherweise die Datei `index.html`.
Um eventuelle Auswirkungen eines Versionswechsels zu einer neueren Bauhaus-Version auf die in Abschnitt 3.5 dokumentierten Testergebnisse schneller ermitteln zu können, liegen den Skripten zwei XML Dateien `expected_results_cpp.xml` und `results.xml` bei, die jeweils die Werte der erwarteten bzw. der mit Bauhaus-Version 5.5.0 ermittelten Werte enthalten. Mit den beiden nachfolgend aufgeführten Befehlen können eventuelle Abweichungen schnell ermittelt werden:

```
diff --context=0 expected_results_cpp.xml ~/analysis_output/results.xml
```

```
diff --context=0 results.xml ~/analysis_output/results.xml
```