Fraunhofer Institut
Experimentelles
Software Engineering

Werkzeuge zur Ermittlung von Software-Produktmetriken und Qualitätsdefekten

Studie zu Software-Messwerkzeugen 2005

Autoren:
Jörg Rech
Sebastian Weber

IESE-Report Nr. 108.05/D
Version 1.0
29. November 2005

Eine Publikation des Fraunhofer IESE

Das Fraunhofer IESE ist ein Institut der Fraunhofer-Gesellschaft.
Das Institut transferiert innovative Software-Entwicklungstechniken, -Methoden und -Werkzeuge in die industrielle Praxis. Es hilft Unternehmen, bedarfsgerechte Software-Kompetenzen aufzubauen und eine wettbewerbsfähige Marktposition zu erlangen.

Das Fraunhofer IESE steht unter der Leitung von
Prof. Dr. Dieter Rombach (geschäftsführend)
Prof. Dr. Peter Liggesmeyer
Fraunhofer-Platz 1
67663 Kaiserslautern

Abstract

Heutzutage haben große Softwaresysteme einen Grad an Kompliziertheit erreicht, der sie über unserer Fähigkeit stellt, sie schnell und einfach zu entwickeln oder zu pflegen. Dies erhöht die Notwendigkeit an Software-Organisationen, Systeme mit hoher Qualität zu entwickeln oder existierende dahingehend zu überarbeiten.

Um die Qualität ihrer Softwareprodukte zu verbessern, verwenden Organisationen häufig Maßnahmen wie die Softwaremessung. Software Produktmetriken dienen dabei der quantitativen Messung von Eigenschaften der Softwaresysteme um deren Qualität oder Herstellungsaufwand abzuschätzen und Qualitätsdefekte aufzuspüren. Subsysteme mit einer schlechten Qualität können dann systematisch angegangen werden, um die Qualität des gesamten Softwaresystems zu verbessern.

Dieser Report stellt einige Messwerkzeuge zur Ermittlung von Software Produktmetriken für die Programmiersprache Java vor und untersucht deren Einsatz zur Entdeckung von Qualitätsdefekten wie Code Smells, Antipatterns oder Design Flaws.

Schlagworte: software metrics, software product quality, software quality assurance, software quality measurement, software quality tool, code smells, anti-patterns, design flaws

Inhaltsverzeichnis

Table of Figures

Table of Tables

1 Einführung

Die Softwareindustrie hat den Ruf kostspielige und qualitativ minderwertige Softwaresysteme zu produzieren. Heutzutage haben große Softwaresysteme einen Grad an Kompliziertheit erreicht, der sie über unserer Fähigkeit stellt, sie schnell und einfach zu entwickeln oder zu pflegen. Dies erhöht die Notwendigkeit an Software-Organisationen, Systeme mit hoher Qualität zu entwickeln oder existierende dahingehend zu überarbeiten.

Um die Qualität ihrer Softwareprodukte zu verbessern, verwenden Organisationen häufig Maßnahmen zur Qualitätssicherung wie die Softwaremessung oder Refaktorierung. Es ist wichtig, dass die einzelnen Produkte gewissen Qualitätsstandards entsprechen und dahingehend untersucht werden, um Aussagen über die Qualität von Produkten machen zu können oder sie bezüglich einzelner Qualitätsaspekte zu vergleichen. Mit metrikbasierten Qualitätsanalyse kann die Qualität eines Produktes objektiv erfasst und analysiert werden.

Während eines Softwareprojektes entstehen nun zahlreiche Produkte wie Anforderungen, Entwürfe, Testfälle oder Quellkode. In diesem Report fokussieren wir einzig und alleine auf den Quellkode und schränken die Untersuchung der Werkzeuge weiterhin auf die Programmiersprache Java ein. Die beiden Kerntechniken, für die Messwerkzeuge eingesetzt werden sollen, sind:

- Die *Softwaremessung* verwendet bei Softwaresystemen so genannte Softwaremetriken um den Quellkode zu charakterisieren und damit die Qualität quantitativ einschätzen zu können. Subsysteme oder Klassen mit einer schlechten Qualität können dann systematisch angegangen werden, um die Qualität des gesamten Softwaresystems zu verbessern.
- Bei der *Refaktorierung* wird das Softwaresystem untersucht und überflüssige Konstrukte aus dem Quellkode entfernt, ohne dessen Funktionalität zu verändern. Dabei werden häufig auch spezifische Qualitätsdefekte wie Code Smells gesucht und mit spezifischen Refaktorierungstechniken entfernt. Heutzutage ist die Entdeckung von Qualitätsdefekten aber immer noch manuell zu erledigen, obwohl einige mittels Softwaremetriken zu entdecken sind.

1.1 Softwarequalität

Ein Qualitätsmodell definiert Kriterien (bzw. Qualitätsaspekte) unterschiedlichster Granularität zur Bewertung der Qualität eines Softwareproduktes oder Softwareprozesses. Heutzutage existieren verschieden Rahmenwerke für Quali-

tätsmodelle wie ISO-9126, die dazu verwendet werden können, für ein Softwareprodukt ein konkretes Qualitätsmodel zu entwickeln. Qualitätsmodelle variieren insb. in Bezug auf den Fokus, welches die wichtigsten Qualitätsaspekte für ein Softwareprodukt in einem Unternehmen oder einer Domäne darstellen. Beispielsweise sind Qualitätseigenschaften wie Performance oder Zuverlässigkeit im Automobilbereich wichtiger als Portabilität.

Ein Qualitätsmodell stellt mit all seinen Qualitätsaspekten und Verfeinerungen von Qualitätsaspekten einen gerichteten azyklischen Graphen dar. Knoten repräsentieren dabei Qualitätsaspekte oder Softwaremetriken und Kanten repräsentieren Beziehungen oder Verrechnungsformeln zwischen Qualitätsaspekten und Softwaremetriken. Ausgehend von der Wurzel bis hin zu den Blättern werden die Qualitätsaspekte immer spezifischer. Die letzte Stufe der Konkretisierung bilden die Blätter in Form von Softwaremetriken. Kapitel 2 geht im Detail auf die Verwendung von Metriken ein. Abbildung 1 zeigt die Zusammenhänge in einem Qualitätsmodell. Eine Methode zur produkt- oder domänenspezifischen Verfeinerung stellt die Goal-Question-Metric (GQM) Methode dar. Dabei werden Ziele (Qualitätsaspekte) immer weiter verfeinert und mit Hypothesen unterfüttert, bis man zu spezifischen messbaren Softwaremetriken gelangt. Durch die Verfeinerungsmethode wird ebenfalls die Interpretationsmethode festgelegt.

Wenn nun ein Qualitätsmodell festgelegt ist und entschieden wurde, welche Qualitätsaspekte für das Softwaresystem relevant und wichtig sind, müssen konkrete Indikatoren oder Metriken ausgewählt werden, um den Ausgangszustand und die Veränderung nach einer qualitätsverbessernden Maßnahme zu ermitteln.

Wie in Abbildung 1 dargestellt kann ein Qualitätsaspekt wie Wartung (links im Bild) in feinere Qualitätsaspekte, Entwurfheuristiken und letztendlich Softwaremetriken (rechts im Bild) verfeinert werden. Dadurch wird die Ermittlung von konkreten Metriken ermöglicht, welche dazu verwendet werden können die Wartbarkeit (insb. basierend auf Wissen aus anderen Projekten) abzuschätzen.

Bei der Messung und anschließenden Interpretation werden aber meist subjektive, nicht fassbare Qualitätsaspekte wie z. B. Wartbarkeit, Zuverlässigkeit oder Erweiterbarkeit mit konkreten Messergebnissen in Beziehung gesetzt. Das Problem besteht darin, dass verschiedene Menschen von einer Qualitätsanforderung unterschiedliche Vorstellungen haben.

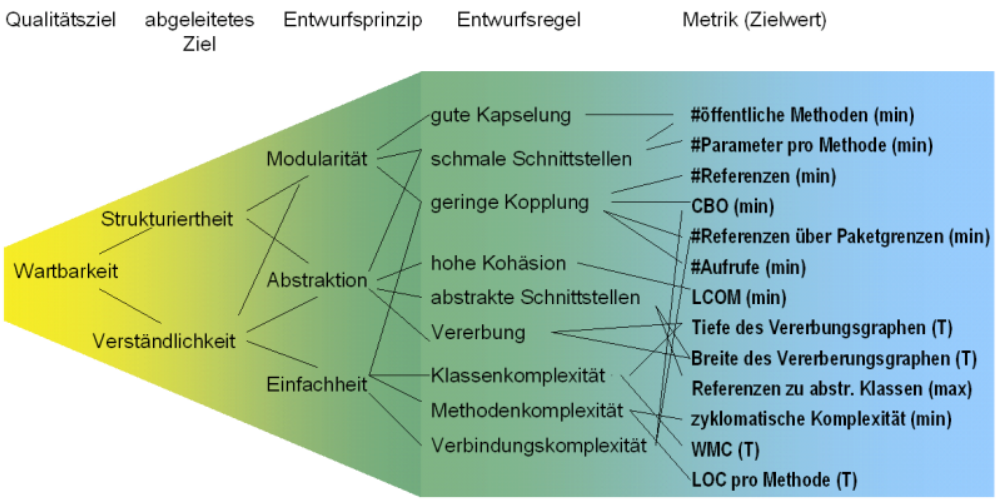


Abbildung 1 Verfeinerung eines Qualitätsmodells für Wartbarkeit bis auf spezifische Metriken (basierend auf http://www.software-kompetenz.de/?11134)

Die erste Hürde bei der Softwaremessung ist es also, einen allgemeinen Konsens über die Begrifflichkeit und die Art der Messung dieser Qualitätsbegriffe zu erreichen und ein produkt- oder domänen-spezifisches Qualitätsmodel zu erstellen. Erst dann können Metriken definiert und Schwellenwerte zur Identifikation von Anomalien festgelegt werden.

Eine weitere Herausforderung ergibt sich durch die Festlegung von Metriken zur Einschätzung eines bestimmten Qualitätsziels. Nur langjährige Praxiserfahrung eines Entwicklungsteams, ein zuverlässiger Industrie-Benchmark, oder eine produkt-spezifische Erfahrungsdatenbank stellt in einem akzeptablen Rahmen sicher, dass eine wohlfundierte und valide Beziehung zwischen den Qualitätsaspekten und den gewählten Metriken hergestellt werden kann. Ansonsten besteht die Gefahr, dass die erhobenen Daten überhaupt keine Qualitätsbewertung erlauben und falsch interpretiert werden.

1.2 Ziele der Studie

Seit den Anfängen des Software Engineering sind sehr viele Softwaremetriken, Sätze von Metriken, und Messwerkzeuge entstanden, um die Software charakterisieren zu können. Dabei können die Metriken zu unterschiedlichen Zwecken eingesetzt werden. Im Folgenden werden die drei wesentlichen Ziele beschrieben, die in dieser Studie untersucht wurden.

1.2.1 Ziel$_1$: Monitoring und Reporting der Softwarequalität

Ein häufig auftretendes Ziel der Softwaremessung ist die Charakterisierung zum Zwecke des Bestimmung von Qualitätseigenschaften, der Prognose von Kosten sowie des Vergleiches zweier (oder mehrerer) Softwareprodukte. Dabei wird das System in einem Snapshot oder einem längeren Zeitraum beobachtet und über die Messwerte ein Report für einen höheren Verantwortlichen (z.B. Manager) erstellt. Im Mittelpunkt steht dabei die Interpretation der Messwerte zu den einzelnen Metriken, um von Ihnen handlungs- oder entscheidungsrelevante Informationen abzuleiten. Im Folgenden werden die Ausprägungen beim Monitoring und Reporting weiter ausgeführt:

- **Kostenabschätzung für die Entwicklung**: Basierend auf Metriken zu Produkten aus frühen Entwicklungsphasen (z.B. Function Points bei Anforderungen) werden die Entwicklungskosten abgeschätzt. In agilen Entwicklungsmethoden mit vielen Iterationen oder bei Entwicklung mit Prototypen können Codemetriken auf die ersten Systemversionen angewandt werden, um die Kosten abzuschätzen. Ein Messwerkzeug müsste also basierend auf diesen frühen, möglicherweise unfertigen Systemversionen, die Frage beantworten können, wie teuer die weitere Entwicklung des Softwareproduktes werden wird.
- **Kostenabschätzung für die Wartung**: Nach der Entwicklung des Softwareproduktes stellt sich die Frage, wie sich die Wartung- und Erweiterungkosten entwickeln. Basierend auf einer lauffähigen und ggf. ausgelieferten Systemversion sollten diese Kosten nun mittels verschiedener Metriken abschätzbar sein. Ein Messwerkzeug sollte also basierend auf einem lauffähigen System die Wartungskosten abschätzen können.
- **Risikoabschätzung**: Neben der Abschätzung der Kosten für die Entwicklung oder Wartung eines Softwaresystems sollten ebenfalls Risiken identifizierbar sein, die während eines solchen Projektes auftreten könnten. Ein Messwerkzeug sollte also basierend auf einer frühen oder fertigen Systemversion Risiken abschätzen können (die ggf. schon in anderen ähnlichen Projekten aufgetreten sind).
- **Qualitätsabschätzung / Benchmarking**: Die Abschätzung der Qualität eines Softwareproduktes kann dazu dienen Probleme und Potentiale zu identifizieren. Neben der Abschätzung der Wartbarkeit sind hier auch die anderen Qualitätsmerkmale interessant, wie bspw. die Wiederverwendbarkeit, Testbarkeit, Performance oder Zuverlässigkeit des Systems. Ein Messwerkzeug sollte also die Frage beantworten können, wie gut (relativ zu einem Standardprodukt oder Benchmark) das Softwaresystem aufgebaut ist.
- **Qualitätsgutachten**: Die Erstellung eines Qualitätsgutachtens über ein Softwaresystem kann dazu dienen, externe Software (z.B. Verwendung von OSS / Third-Party Quellkode, Einkauf von Firmen inkl. Software, Outsourcing / Subcontracting) zu kontrollieren. Neben dem Vergleich der Qualität mit ei-

nem vorgegebenen Qualitätsprofil ist hier auch der Vergleich mit einer Architektur denkbar (d.h. „Entspricht der geliefert Kode dem vorgegebenen Entwurf?“). Dies dient der Reduktion des Aufwandes zur Ermittlung von Schwachstellen in der Software sowie zur Auswahl eines externen Softwareproduktes (d.h. COTS-Auswahl). Ein Messwerkzeug sollte es also ermöglichen die Qualität eines Softwaresystems basierend auf spezifischen Metriken zu messen und die mit einem vorgegebenen Qualitätsprofil oder -standard zu vergleichen.

- **Vergleichsgutachten**: Bei Streitigkeiten über das Copyright oder Patentverletzungen kann es vorkommen, dass ein Report über die Ähnlichkeit zweier Softwaresysteme benötigt wird. Um ein vergleichendes Gutachten zwischen zwei Softwaresystemen zu erstellen, kann ein Messwerkzeug verwendet werden, um Plagiate (z.B. Algorithmen oder Architekturen) zu entdecken.

Weiterhin ist auch der zeitliche Verlauf dieser Abschätzungen während eines Projektes oder über mehrere Projekte hinweg interessant. Dies kann dazu verwendet werden um die Fragen zu beantworten wie sich die Kosten, der Aufwand, die Risiken oder die Qualität der Produkte entwickelt.

Diese Informationen oder Abschätzungen können dann vom Management verwendet werden, um Entscheidungen zu fundieren und größere Maßnahmen einzuleiten oder die Strategie des Qualitätsmanagements zu steuern.

1.2.2 Ziel$_2$: Entdeckung von Anomalien

Neben der Erstellung eines Profils über ein Softwaresystem zum Zwecke der Erstellung eines Reports über die Qualität, der Entwicklungskosten oder der Risiken werden Metriken auch dazu verwendet Probleme im Softwaresystem direkt zu finden und zu beseitigen.

Eine Art der Entdeckung von Problemen in Softwaresystemen ist die Identifizierung von Anomalien in den Metrikdaten. Ist beispielsweise eine Klasse im Softwaresystem extrem groß (d.h. die Lines of Code übersteigen bei weitem den durchschnittlichen Wert) so steigt die Wahrscheinlichkeit, dass dort schlecht programmiert wurde und eine spätere Wartung komplizierter oder gar unmöglich wird. Potentiell lohnt es sich also für den Entwickler die Stellen des Softwaresystems zu untersuchen in denen anomale Metrikdaten gemessen werden.

Was eine Anomalie darstellt hängt vom jeweiligen Referenzmodell ab. Dabei kann zwischen dem System, den Unternehmenssystemen, den Domänsystemen und allgemeinen Systemen unterschieden werden. Eine Anomalie im Referenzmodell des Systems ist also eine „extreme“ Abweichung von den Metrikdaten in dem vermessenen System. Bei dem Referenzmodell bzgl. einer Domäne würde man Anomalien in Bezug auf „normale“ Metrikdaten von Softwaresystemen

einer Domäne (z.B. Informationssysteme oder spezifischer WIKI-Systeme) ermitteln. Anomalien beziehen sich weiterhin sowohl auf einzelne Metriken (z.B. Lines of Code) als auch auf Kombinationen von Metriken.

Im Mittelpunkt der Anomalieanalyse steht dabei die Interpretation der Messwerte in Relation zu einem vorgebenden Mittel- oder Schwellwert. Im Folgenden werden die Ausprägungen bei der Anomalieanalyse weiter ausgeführt:

- **Mittelwertbasierte Anomalieanalyse**: Anomalien entstehen durch eine größere Abweichungen vom Mittelwert (oder Median, Modus, etc.) relativ zum gewählten Referenzmodel und werden umso stärker oder gravierender je weiter sie davon entfernt sind. Bei der Untersuchung der Anomalien werden dann alle Abweichungen vom Mittelwert – wie gering auch immer – betrachtet.
- **Schwellwertbasierte Anomalieanalyse**: Anomalien entstehen durch die Über- oder Unterschreitung eines oder zweier Schwellwerte, die fest vorgegeben (d.h. absolut) sind oder relativ (d.h. als ε-Wert) um eine Standardwert (z.B. Mittelwert) herum liegen.

Liegen die Messungen hinsichtlich der im Qualitätsmodell definierten Metriken vor, so werden diese nach Anomalien bzgl. des Referenzmodells untersucht. Die Identifizierung von Anomalien, insb. basierend auf mehreren Metriken, ist kein trivialer Vorgang. Nicht die Erhebung von Messdaten sondern deren Interpretation stellt ein Problem dar, da einige Metriken auch immer mit sinnhaften Ausnahmen behaftet sind. Beispielsweise haben abstrakte Methoden in Java keine Methodenrümpfe, da diese erst noch in erbenden Klassen ausformuliert werden müssen. Ein Messwert von „0" bei Lines of Code (LOC) ist also hier normal (d.h. stellt keine Anomalie dar) und sollte von der weiteren Analyse und Interpretation ausgeschlossen werden.

Wie der Nutzer des Messwerkzeuges dann mit den Anomalien umgeht, bleibt ihm meist selbst überlassen. Entweder optimiert er den Quellkode dahingehend, dass die Metrikwerte sich dem Mittelwert annähern bzw. auf die „gute" Seite des Schwellwertes wandern, oder er behält den Quellkode bei (z.B. da er aus Performance-Gründen nicht weiter zu optimieren ist).

1.2.3 Ziel$_3$: Entdeckung von Qualitätsdefekten

Zusätzlich zur Entdeckung von Anomalien, die eine Ausnahme von den „normalen" Metrikwerten darstellen können Metriken auch dazu eingesetzt werden, um spezifische Qualitätsdefekte (d.h. Code smells, Anti-patterns, Design Flaws, etc) zu entdecken. Qualitätsdefekte stellen häufig wiederkehrende Probleme oder Anomalien dar, welche die Qualität der Software mindern. Kapitel 3 geht genauer auf diese Art von Defekten in Softwaresystemen ein.

Diese Qualitätsdefekte werden während der Refaktorierung oder dem Reengineerings eines Softwaresystems benötigt, um die Softwarequalität zu verbessern. Bisher ist die Entdeckung von Qualitätsdefekten meist eine manuelle Aufgabe, die nur in einigen einfachen Szenarien automatisierbar ist. Ist ein Qualitätsdefekt identifiziert, besteht die Möglichkeit mittels konkreter Refaktorierungen (d.h. Code-Transformationen) diese zu beseitigen. Ein Messwerkzeug, welches Qualitätsdefekte findet, sollte also den Entwickler direkt mit einer Lösung versorgen oder automatische Refaktorierungen anstoßen.

1.2.4 Weitere Ziele von Softwaremetriken

Neben den oben genannten Zielen gibt es noch weitere, welche wünschenswerterweise durch den Einsatz von Softwaremetriken erreichbar sein sollten.

- **Testplanung**: Metriken sollten die Entdeckung potentiell fehleranfälliger Komponenten unterstützen, um die gezielte Erstellung und Durchführung von modularen Regressionstests sowie die Integration von build-in Tests präziser auf Schwachstellen zu fokussieren.
- **Reengineering-Analysen**: Metriken sollten zur Idenifikation von Ähnlichkeiten zwischen mehreren Softwareprodukten verwendet werden. Dies ermöglicht weiterhin die Erstellung eines Reports über die Struktur eines schlecht- oder undokumentierten Softwaresystems zur Ermittlung von Subsystemen, Architekturebenen oder Schwachstellen.
- **Migrationsevaluierungen**: Solch eine Evaluation dient der beantwortung der Frage „Wie stark hat sich die aktuelle Architektur der Zielarchitektur angenähert?“. Die Ermittlung relevanter Metriken während des Projektverlaufes dient dabei der frühzeitigen Vorwarnung.

Abschließend sollte noch darauf hingewiesen werden, dass eine Messung nur so gut wie das verwendete Werkzeug sein kann. In einem realen Softwareprojekt werden Metriken programmunterstützt erhoben, da dieser Vorgang sehr gut automatisiert werden kann. Ist das Messwerkzeug fehlerfrei, schleichen sich keine Fehler bei der Metrikerhebung ein. Dann ist die Analyse der Ergebnisse bezüglich der zu erreichenden Qualitätsziele der eigentliche Knackpunkt. Wenn das Messwerkzeug schon falsche Daten erhebt bzw. die Interpretation von Metriken zwischen verschiedenen Messwerkzeugen nicht vergleichbar ist, wird die Analyse der Messwerte nicht fehlerfrei verlaufen.

1.3 Strukturierung der Werkzeugbeschreibungen

Zur Analyse der Werkzeuge, welche die Messung der Softwarequalität ermöglichen, wird folgende Struktur verwendet.

- Eine *Übersicht* über die Herkunft und Bezugsinformationen des Werkzeuges
- Beschreibung der *Ziele,* für welche das Werkzeug entwickelt (und beworben) wird.
- Liste aller *Metriken,* die vom Werkzeug berechnet werden können.
- Liste aller *Funktionen* und Features, die im Werkzeuge realisiert wurden
- Szenario $Ziel_1$: Welche Qualitäten sind für ein Reporting messbar?
- Szenario $Ziel_2$: Werden Anomalien entdeckt oder Hinweise auf Probleme generiert?
- Szenario $Ziel_3$: Werden Qualitätsdefekte ermittelt bzw. konkrete Refaktorierung vorgeschlagen?

2 Softwaremetriken für Qualitätsanalysen

Wie bereits im vorherigen Kapitel erwähnt, werden zu Beginn eines Softwareprojekts Qualitätsziele bzgl. der Qualitätsaspekte festgelegt, die durch eine Reihe von Fragen systematisch verfeinert werden. Diese Fragen werden durch relevante Softwaremetriken mit quantitativen Daten (Messwerten) beantwortet.

Messwerte stellen dabei die Zuordnung einer Zahl zu einer Entität (z.B. einer Klasse) basierend auf einer Einheit (d.h. eine Softwaremetrik wie Lines of Code) dar. Die Softwaremetriken und die dazu gemessenen Messwerte repräsentieren hier spezielle Attribute dieser Entität, die für eine weitere Interpretation (z.B. Aufwandsschätzung) verwendet werden können [1].

Wird zu Projektstart bei einem Wartungsprojekt (d.h. ein Softwaresystem existiert und wird erweitert) mit diesem Qualitätsmodell und den spezifizierten Metriken ein Report erstellt, kann dieser als Baseline verwendet werden. Durch diese Vorgehensweise kann sowohl die Veränderung der Softwarequalität ermittelt, als auch eine Aufwandsschätzung durchgeführt werden.

Dieses Kapitel geht genauer auf Softwaremetriken und deren Analyse ein die zur Identifikation von Problemen herangezogen werden. Weiterhin dient die Definition der Metriken als Referenzbasis um diese bei den Messwerkzeugen zu referenzieren.

2.1 Klassifikation von Metriken

Metriken können anhand von verschiedenen Kriterien klassifiziert werden. Im Folgenden werden einige, größtenteils unabhängige, Dimensionen zur Unterscheidung von Metriken beschrieben. Eine Möglichkeit besteht z.B. darin, eine Aufteilung bezüglich des zu messenden Objektes vorzunehmen. Im Software Engineering sind dabei die beiden folgenden Ausprägungen dominant:

- **Produktmetriken** sind Maße des Softwareproduktes, die während des kompletten Entwicklungsprozesses erhoben werden. Komplexität des Softwaredesigns oder Anzahl der Seiten der Dokumentation sind Beispiele für diese Kategorie.
- **Prozessmetriken** sind Maße für den Softwareentwicklungsprozess, wie z.B. gesamte Entwicklungszeit oder durchschnittlicher Grad des Entwicklungsstands.

Da wir in diesem Report nur auf Produktmetriken fokussieren, die man aus dem Quellkode erheben kann, werden weitere Unterscheidungsmerkmale bei Prozessmetriken nicht weiter erläutert. Die Produktmetriken werden weiterhin nach dem verwendeten Strukturierungsparadigma unterteilt und stellen somit ebenfalls eine historische Unterteilung dar:

- **Traditionelle Metriken**: Traditionelle Metriken wurden ursprünglich dafür entworfen, den Anforderungen imperativer Programmiersprachen gerecht zu werden. Kritiker sehen in ihnen für den Einsatz in objektorientierten Umgebungen keine ausreichende Berücksichtigung von objektorientierten Konzepten.
- **Objektorientierte Metriken**: Mit zunehmender Verbreitung der Objektorientierung entstanden vermehrt objektorientierte Metriken. Diese Metriken verfolgen das Ziel, auf die Besonderheiten von objektorientierten Strukturen einzugehen. Allerdings existiert auch in diesem Bereich keine eindeutige Definition, was traditionelle von objektorientierten Metriken unterscheidet. Ebenso ist unklar, welche traditionellen Metriken sinnvoll in objektorientierten Umgebungen eingesetzt werden können.
- **Aspektorientierte Metriken:** Ein aktueller Trend ist die aspektorientierte Softwareentwicklung, bei der zur Zeit der Kompilierung zentral verwaltete Aspekte in den Quellkode eingewebt werden. Metriken für diese Sprachart werden zur Zeit entwickelt.

Eine weitere Trennung erfolgt nach der Art, wie die Daten der Metriken erhoben werden. Dabei unterscheidet man in subjektive (bzw. qualitative) und objektive (bzw. quantitative) Metriken.

- **Objektive Metriken** führen immer zum selben Ergebnis, wenn verschiedene Betrachter die Auswertung vornehmen. In den Bereich objektive Metriken ist Lines of Code (LOC) zu zählen.
- **Subjektive Metriken** führen dagegen nicht immer zum gleichen Ergebnis. Verschiedene Beobachter messen in der Regel unterschiedliche Werte.

Man unterscheidet zwischen direkten und indirekten Metriken bzw. primitiven und berechneten Metriken (computed metrics).

- **Direkte Metriken** stehen in keinerlei Beziehung zu anderen Attributen. Somit erfolgt die Messung unabhängig von anderen Maßen. Diese Metriken werden direkt ausgewertet. Ein Beispiel wäre LOC.
- **Indirekte Metriken** setzen die Messung ein oder mehrerer Attribute voraus. Sie werden also aus den Ergebnissen anderer Metriken berechnet.

Abschließend sind im Folgenden noch Metriken aufgeführt, die das Produkt auf unterschiedliche Merkmale hin untersuchen:

- **Größenmetriken**: Die Größe des Softwaresystems hängt von vielen Skalierungsfaktoren ab. Somit existieren viele Metriken, die eine Dimension des Softwaresystems abzählen. Ein weit verbreiteter Ansatz ist die Zählung der Zeilen (Lines of Code, LOC), Attribute, Methoden oder Klassen.
- **Strukturmetriken**: Metriken, welche die Struktur des Softwaresystems bestimmen, sind bspw. Kopplungs- und Kohäsionsmetriken. Sie können für eine statische Analyse herangezogen werden, d.h. eine Untersuchung des statischen Aufbaus eines Softwaresystems auf die Erfüllung vorgegebener Kriterien. Die statische Analyse dient zur Bewertung von Qualitätsmerkmalen wie Lesbarkeit, Erweiterbarkeit oder Testbarkeit.
- **Ablaufsmetriken**: Metriken, die den Ablauf des Softwaresystems charakterisieren, können dynamisch während der Ausführung ermittelt werden. Dazu gehören z.B. Ausführungszeit einer Methode oder die Anzahl der Zugriffe auf ein Attribut.
- **Änderungsmetriken**: Die Änderungen eines Softwaresystems während seiner Entwicklung oder Wartung werden mit Metriken wie Anzahl neuer Klassen oder Anzahl von Modifikationen gemessen. Weiterhin kann auch der verlauf anderer Metriken (z.B. LOC) untersuch und analysiert werden. Diese Metriken können aus Versionierungssystemen (z.B. CVS) ermittelt werden, um Aussagen zur Qualität zu treffen oder eine Prognose über das weitere Wachstum abzugeben.

Alle diese Metriken können natürlich auch noch nach weiteren Merkmalen gruppiert werden. Beispielsweise werden sie danach unterschieden, welches Ziel mit Ihnen verfolgt wird und was man mit Ihnen eigentlich messen will (wie z.B. Komplexität oder Produktivität).

2.2 Softwaremetriken

Über die Jahre wurden viele verschiedene Metriken entwickelt, die häufig auf spezifische Szenarien oder Umgebungen zugeschnitten sind. Leider sind oft keine eindeutigen Definitionen von Metriken angegeben, wodurch diese – obwohl sehr ähnlich – in einigen Messwerkzeugen leicht abweichend implementiert worden sind. Dies ist ein weiterer Grund, warum die spätere Analyse der Metriken häufig nicht eindeutig ist und allgemein gültig in allen Softwareprojekten eingesetzt werden können.

Stattdessen haben sich über die Zeit bestimmte Produktmetriken etabliert, welche im Anschluss an diesen Abschnitt vorgestellt werden. Dazu sind in Tabelle 1 einige wichtige Charakteristiken der objekt-orientierten Sprachen aufgeführt die bei den Definitionen der Metriken Verwendung finden.

Tabelle 1 Wichtige objektorientierte Konzepte

Klasse	Klassen sind die „Baupläne" für die Erzeugung von konkreten Objekten. Klassen definieren die allgemeine Struktur und das Verhalten.
Attribut	Attribute legen die Struktur eines Objekts durch Werte fest.
Objekt	Eine Instanz einer Klasse. Jedes Objekt hat einen eigenen Zustand und ist aufgrund der Identität eindeutig. Objekte bieten Methoden an, um den Zustand zu ändern.
Methode	Eine Operation auf einem bestimmten Objekt. Methoden werden in der Klassendefinition beschrieben.
Vererbung	Durch Vererbung werden Klassenhierarchien festgelegt. Kindklassen erben Charakteristiken von übergeordneten Klassen, können diese jedoch um eigene erweitern.
Nachricht	Ein Objekt stellt durch Messages (Nachricht) anfragen an andere Objekte, um Operationen auf diesen Objekten auszuführen.
Kopplung	Die Kopplung (Coupling) eines Objekts X von einem Objekt Y besteht dann, wenn X Nachrichten an Y sendet.
Kohäsion	Kohäsion (Cohesion) ist der Grad der Abhängigkeiten zwischen Methoden einer Klasse.

Es gibt eine Reihe von Metriken, welche in den untersuchten Messwerkzeugen ermittelt werden. In den nachfolgenden Sektionen werden diese Metriken vorgestellt und aus den Werkzeugbeschreibungen referenziert.

- **LOC (Lines of Code)**: Diese Metrik bestimmt die Anzahl der Programmzeilen, wobei meist die Kommentarzeilen nicht mitgezählt werden. Die Messung ist dabei mit Problemen wie bspw. der Sprachabhängigkeit (eventuell in unterschiedlichen Sprachen unterschiedliche Anzahl von Zeilen notwendig) verbunden.
- **CC (Cyclomatic Complexity)**: Die zyklomatische Komplexität geht auf McCabe zurück und charakterisiert die Kontrollflussstruktur eines Softwaresystems. Sie misst die Anzahl der linear unabhängigen Wege durch den Kontrollflussgraphen. Ziel dieses Maßes ist das Bestimmen der Komplexität eines Algorithmus in einer Methode. Vereinfacht gesprochen berechnet CC ebenfalls die Anzahl der benötigten Testfälle, um die Methode vollständig zu testen. Stellt man den Algorithmus durch ein Kontrollflußdiagramm dar, so ergibt sich der Wert durch die Anzahl der Kanten abzüglich der Knoten plus zwei.
- **WMC (Weighted Methods per Class)**: Die Metrik ermittelt die gewichtete Methoden pro Klasse zur Bestimmung der Komplexität einer Klasse. Gewichtete Methoden, da bspw. Zugriffsmethoden auf Attribute (Get-/Set-Methoden) geringer bewertet und für die Komplexität einer Klasse nicht oder nur schwach einberechnet werden. Eine aufwändigere Variante berechnet die Gewichte der Methoden über die Komplexität mittels CC. WMC kann als Richtwert herangezogen werden, wie viel Zeit und Aufwand notwendig ist, die betrachtete Klasse zu testen und zu überwachen.

- **DIT (Depth of Inheritance Tree)**: Tiefe des Vererbungsbaums ist ein Maß, welches die Anzahl der Superklassen für eine betrachtete Klasse zählt. Die Tiefe der Vererbungshierarchie wird über den längsten Pfad durch die Vererbungshierarchie bis zur betrachteten Klasse bestimmt. In Java ist der längste Pfad als der Weg definiert, der von der universellen Klasse Object hin zur betrachteten Klasse (oder Interface) führt. Mit zunehmender Tiefe des Baums steigt die Komplexität, weil die Anzahl der vererbten Methoden und Attribute steigt. Allerdings steigt dabei der Grad der Wiederverwendbarkeit des Softwaresystems. Je mehr Methoden vererbt werden, desto schwieriger und fehleranfälliger sind Vorhersagen über das Verhalten des Softwaresystems zu treffen.
- **NOC (Number of Children)**: Die Anzahl der Kinder ist äquivalent zur Anzahl der Subklassen für eine bestimmte Klasse. Je höher die Anzahl der Subklassen, desto höher ist der Grad der Wiederverwendbarkeit des Softwaresystems. Allerdings ist dies auch mit einer Steigerung der Komplexität verbunden, weil beispielsweise die Fehleranfälligkeit steigt und somit auch der Aufwand für Testen und Debuggen. Nimmt die Anzahl der Subklassen zu, so erhöht sich das Risiko für eine fehlerhafte Abstraktion der Superklasse. Allgemein kann NOC als Indikator angesehen werden, der den Einfluss der betrachteten Klasse auf das Softwaredesign mißt. Möchte man im Softwaredesign keine Klassen mit zu hohem Einflussfaktor haben, so sollten die NOC-Werte der Klassen keinen zu großen Wert annehmen.
- **CBO (Coupling between Objects):** Anzahl der (Objekt-)Klassen, die mit dem betrachteten Modul in Beziehung stehen. In Java sind Objekte von einer Klasse X abhängig, wenn sie Instanzen von X anlegen und/oder Methoden von X benutzen. In diesen Bereich fällt auch Message Passing, d.h. auch Methodenparameter, die vom Typ X sind, werden von diesem Maß erfasst. Mit diesem Maß werden aber keine Vererbungsbeziehungen betrachtet. Eine Klasse mit sehr vielen Abhängigkeiten mit anderen Klassen weist eine schlechte Wiederverwendbarkeit auf. Solche Klassen sind sehr anfällig für Veränderungen, die sich auf weite Teile des Softwaresystems auswirken. Unabhängige Klassen sind sehr viel leichter in anderen Softwareprojekten einsetzbar. Außerdem steigt die Komplexität des Softwaresystems mit steigendem Grad von Abhängigkeiten.
- **RFC (Response for a Class)**: Das Antwortverhalten einer Klasse ergibt sich aus der Summe aller Methoden die potentiell ausgeführt werden könnten, wenn ein Objekt der Klasse auf eine eingehende Nachricht reagiert. Je höher die Anzahl der Methoden, die aufgerufen werden können, desto größer ist die Komplexität der betrachteten Klasse. Der Aufwand für Testen und Debuggen steigt, weil dabei sehr viele Abhängigkeiten zwischen Methoden betrachtet werden müssen. Dies hängt mit sinkender Codeverständlichkeit und erhöhter Fehleranfälligkeit zusammen.

- **LCOM (Lack of Cohesion on Methods)**: Eine Kohäsionsmetrik, die bestimmt wie stark zusammenhängend die Methoden einer Klasse und deren Instanzvariablen sind. LCOM berechnet sich durch die Anzahl der Methodenpaare, die keine gemeinsamen Instanzvariablen haben, abzüglich der Anzahl der Methodenpaare, die solche besitzen. Klassen mit einer hohen Kohäsion (hoher LOCM-Wert) stellen ein Indiz für eine gutes Design der Klassenhierarchie dar. Klassen mit niedriger Kohäsion erhöhen die Wahrscheinlichkeit von Fehlern während des Entwicklungprozesses aufgrund einer erhöhten Komplexität. Solche Klassen sollten in mehrere Klassen aufgesplittet werden.

2.3 Interpretation von Metriken

Das Erheben von Metriken ist nutzlos, wenn diese Maße nicht interpretiert werden. Ein Problem der Metriken liegt dabei darin, dass es oft nicht eindeutig ist, welche Aussagekraft die erhobenen Werte haben. Oder anders ausgedrückt: Es ist oft nicht klar, wie die Ergebnisse zu interpretieren sind. Dabei stellt die Interpretation von Metriken ein schwieriges Problem dar, welches durch folgende Punkte näher beschrieben wird:

- **Subjektivität der Erhebung:** Obwohl in unserer Studie Werkzeuge untersucht wurden, die objektive Poduktmetriken ermitteln, ist nicht auszuschließen, dass die Interpretation der Metriken von Werkzeug zu Werkzeug variiert. Die Problematik bei der Definition von Softwaremetriken wird am Beispiel von LOC deutlich. Hier stellt sich die Frage, ob Kommentarzeilen oder Leerzeilen mitgezählt werden sollen und was dabei exakt gemessen wird. Desweitern ist auch nicht genau definiert was eine Zeile darstellt da neben den Zeilen die durch „newline“ Zeichen beendet werden wird oft auch die Anzahl von Semikolons „;“ ermittelt.
- **Subjektivität der Auswertung:** Die Auswertung einer Metrik und letztendlich die Entscheidung, was aus den Messwerten interpretiert wird ist von Unternehmen zu Unternehmen unterschiedlich. Es muss klar definiert werden, was beim Überschreiten eines Schwellwertes eines, oder einer Gruppe von Metriken zu tun ist. Wenn beispielsweise eine Metrik zwischen den Werten 0 und 1 liegen kann, wie ist dann ein Wert von 0,7 zu deuten?
- **Vergleichbarkeit**: Gleiche Werte in unterschiedlichen Softwaresystemen können oft nicht miteinander verglichen werden. Der Grad der Ähnlichkeit kann aufgrund vieler externer und projektabhängiger Einflussfaktoren nur sehr schwer definiert werden. Es ist nahezu unmöglich, Metrikergebnisse in unterschiedlichen Rahmenbedingungen (unterschiedliche Umgebung, Programmiersprache, etc.) miteinander zu vergleichen.

Wie in Tabelle 2 dargestellt wird bspw. versucht, die Metriken mit Qualitätsaspekten wie Wartbarkeit oder Komplexität in Verbindung zu setzen [2].

Tabelle 2 Metriken und Qualitätsindikatoren (aus [2])

Metrics	Objective	Development Effort	Testing Effort	Understandability	Maintainability	Reuseability
Complexity	↓		↓	↑	↑	
Size (LOC)	↓		↓	↑	↑	
Comments %	↑	↓	↓	↑	↑	
WMC	↓	↓			↑	↑
RFC	↓		↓			↑
LCOM	↓	↓		↑	↑	↑
CBO	↓		↓	↑	↑	↑

Metriken sind also kein Allheilmittel, sie dienen vielmehr als Indikatoren in Entwicklungsprozessen, um frühzeitig Fehler und Designmängel auszumachen und zu beseitigen.

Eine sinnvolle Interpretation der Metriken kann dadurch erfolgen, dass Histogramme, Diagramme oder Graphen erzeugt werden, um Ausreißer zu identifizieren. Wenn in einem Softwaredesign einzelne Klassen weit oberhalb oder unterhalb der Durchschnittswerte liegen, so passt diese Tatsache nicht in das Gesamtbild. Diese Klassen bieten sich an, analysiert zu werden, um herauszufinden, warum diese Klassen vom Durchschnitt so weit entfernt sind. Ausreißer sind Indikatoren für Designfehler und sollten korrigiert oder zumindestens untersucht werden.

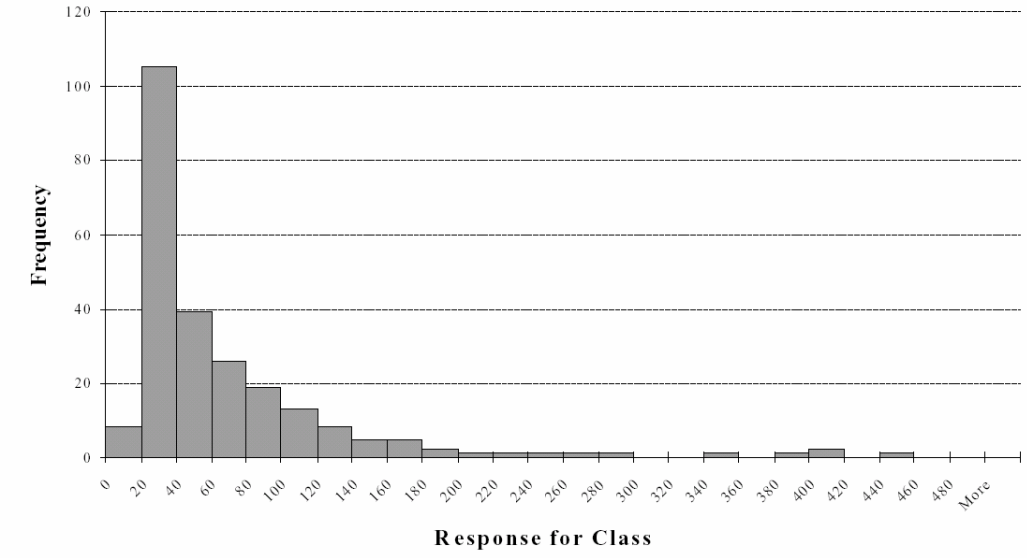


Abbildung 2 RFC-Histogramm für ein Software Projekt (aus [2])

Abbildung 2 zeigt am Beispiel von RFC eine Möglichkeit, wie eine Anomalieanalyse von Metriken für ein Softwareprojekt aussehen könnte. Die Abbildung

zeigt, dass es einige wenige Klassen gibt, die 200 Methoden und mehr aufrufen. Wie bereits erwähnt, steigt die Komplexität, Verständlichkeit und Fehleranfälligkeit mit zunehmendem RFC-Wert. Der Aufwand für Testen und Debuggen steigt ebenfalls. Aufgrund dieser Erkenntnis sollten die Entwickler versuchen, diese Ausreißer zu eliminieren, indem sie die RFC-Werte der betroffenen Klassen deutlich senken. Am Ende dieses Prozesses sollten alle Klassen weitestgehend ähnliche Werte aufweisen. Entsprechend kann eine äquivalente Analyse auf anderen Metriken erfolgen. Ein gutes Metrikprogramm sollte die Maße erheben, die Ergebnisse visualisieren und eventuell Bewertungen und Lösungsvorschläge für die Entwickler anbieten.

Die wichtigste Erkenntnis ist, dass die absoluten Metrikwerte wenig Aussagekraft haben und dass Werte in unterschiedlichen Umgebungen nicht oder sehr schwer miteinander verglichen werden können. Vielmehr sollten die Entwickler vorab planen, in welche Wertebereiche die Maße für die einzelnen Klassen liegen sollten. Wichtig ist aber vor Allem die wichtigsten Qualitätsaspekte im Hinterkopf zu behalten und diese möglichst zu beachten. Ein nach den Metriken optimiertes System kann trotzdem unverständlich und somit schlecht wartbar sein. Die Anomalieanalyse der Metriken identifiziert nur Konstrukte die außerhalb der abgesteckten Werte liegen. Ziel ist es nun, in einer weiteren Optimierungsphase diese Klassen zu überarbeiten, bis die gewünschten Ergebnisse erzielt werden. Allerdings gibt es auch *tradeoffs* da die Optimierung einiger Metriken andere Metriken eventuell in negativer Weise beeinflussen. Einen hohen DIT-Wert ist mit erhöhter Komplexität verbunden, führt allerdings aber auch zu einem hohen Grad an Wiederverwendbarkeit.

3 Qualitätsdefekte

Qualitätsdefekte stellen Probleme in Softwaresystemen dar, welche nicht die Funktionalität allein sondern insb. andere Qualitäten, wie die Wartbarkeit, Wiederverwendbarkeit oder Performance, vermindern. Zu den Qualitätsdefekten gehören Code Smells (Bad smells in Code"), Anti-patterns, Design Flaws (bzw. Entwurfsmängel), negative Design Charakteristiken und neuerdings auch Architektursmells. Diese Defekte sind nicht durch die üblichen Testverfahren aber teilweise über Metriken entdeckbar.

Qualitätsdefekte stellen konkrete und wiederkehrende Probleme dar, welche zumindestens bei Code Smells, durch spezifische Code-Transformationen (Refaktorierungen) beseitigt werden können. Dadurch stellen Qualitätsdefekte Probleme dar, welche teilweise durch Metriken entdeckt und mittels konkreter Handlungsanweisungen beseitigt werden können.

In diesem Kapitel werden einige Qualitätsdefekte vorgestellt und Metriken vorgeschlagen, mit denen man diese entdecken kann. Dabei werden aus Platzgründen nur Code Smells näher erläutert. Ein Messwerkzeug, welches dazu eingesetzt werden soll, diese zu entdecken, sollte also zumindest diese Metriken berechnen und ggf. die Qualitätsdefekte direkt ermitteln und darstellen.

3.1 Code Smells

Der Begriff Code Smells zielt auf Stellen im Quell-Code ab, die möglicher Weise Probleme nach sich ziehen können. Code Smells sind als Indikatoren für Refaktorierungsmaßnahmen zu verstehen. Die Problematik bei der Refaktorierung (engl. Refactoring) besteht nicht in der Durchführung. Vielmehr ist entscheidend zu erkennen, welche Refaktorierungsaktivitäten in welchem Umfang notwendig sind.

Fowler beschrieb in seinem Buch zusammen mit Kent Beck zuerst so genannte „Bad smells in code" (Fowler, 1999), welche seit dem um einige weitere erweitert wurden. Roock und Lippert haben diese Code Smells um Architektur Smells erweitert welche teilweise aus dem Gebiet der Design Flaws stammen (Roock & Lippert, 2004). Entwurfsmängel wurden von Whitemire (als Design Flaws) (Whitmire, 1997) und Riel (als Design Characterisitics) (Riel, 1996) beschreiben.

Tabelle 3 Liste der Code Smells (Bad Smells in Code)

Code smell	Metric
Alternative classes with different interfaces	N/A
Comments	Lines on Documented Code (LODC)und Lines of Non-Documented Code (LOC)
Data class	Number of Public Attributes (NOPA), Number of Attributes (NOA) und Number of Methods (NOM)
Data clumps	N/A
Divergent Change	N/A
Duplicated code	N/A
Feature envy	Number of external Accesses (NOEA) und Number of internal Access (NOIA) pro Methode
Inappropriate Intimacy	N/A
Incomplete library class	N/A
Large class	Number of Methods, Number of Attributes, Lines of Code (LOC) auf Klassenbasis
Long method	Lines of Code (LOC) auf Methodenbasis
Lazy class	Number of Methods (NOM), Number of Attributes (NOA), Number of Users (NOU)
Long parameter list	Number of Parameter (NOP)
Message chains	Depth of Delegation (DOD) pro methode bzw. Delegation
Middle man	Number of Delegates (NOD) und Lines of Code (LOC) auf Methodenbasis
Parallel inheritance hierarchies	N/A
Primitive obsession	Number of Primitive Types (NOPT) und ggf. Number of Attributes (NOA) um das Verhältnis zu bestimmen
Refused bequest	Number of Method Users (NOMU) und Number of Attribute Users (NOAU) ggf. mit der unterscheidung intern und extern.
Shotgun surgery	N/A
Speculative generality	Number of Invocations (NOIV) Number of Instantiations (NOIS)
Switch statements	Existence of Switch (EOS)
Temporary field	N/A
Indecent Exposure	N/A

Code Smells sind potentielle Kandidaten für eine Refaktorierung. In diesem Kapitel wird der Code Smells Katalog von Fowler aufgegriffen und Mechanismen beschreiben, mit denen Code Smells identifiziert werden können. Genauer gesagt wird hier untersucht, ob einzelne Code Smells mit Metriken entdeckt werden können. Aufbauend auf diesem Kapitel werden in Kapitel 4 verschiedene Messwerkzeuge untersuchen. Für die Qualitätssicherung ist der Punkt zu klären, wie mächtig die einzelnen Tools sind und ob sie zur Identifikation von Code Smells herangezogen werden können.

Im Folgenden werden nun einige Code Smells vorgestellt und Metriken angegeben, die bei Ihrer Entdeckung behilflich sind. Dabei werden zuerst einige einfache Code Smells betrachtet, bevor komplizierte beschrieben werden.

3.2 Long Method

Bei der objektorientierten Programmierung sollte darauf geachtet werden, dass Methoden nicht zu lang sind. Je länger die Methode, desto schwieriger ist es für einen Entwickler sie zu verstehen. Die Verständlichkeit und Lesbarkeit wird also von der Länge einer Methode negativ beeinflusst und wirkt sich somit negativ auf die Wartbarkeit und Testbarkeit aus. Des Weiteren benötigt eine kurze verständliche Methode prinzipiell weniger Kommentierung als eine lange. Die Namensgebung der Methode und die Wahl der Variablennamen tragen dabei zur Verständnis bei. Ein weiterer Vorteil kurzer Methoden hinsichtlich langer Methoden besteht darin, dass ein Entwickler nicht ständig hin- und herscrollen muss. Eine kurze Methode kann vollständig angezeigt werden. Auch die Identifizierung von Fehlern funktioneller Art wird dadurch vereinfacht.

Die offensichtlichste Metrik für lange Methoden ist die Anzahl der Zeilen (LOC) pro Methode. Dagegen ist die Frage, was eine lange Methode ausmacht, noch nicht beantwortet. Dies muss entweder von der Projektleitung vor Beginn der Coding-Phase festgelegt oder über einen Schwellwert bspw. auf Projektbasis ermittelt werden. Überschreitet eine Methode diesen Schwellwert so handelt es sich um einen Ausreißer und eine Anomalie liegt vor.

3.3 Large Class

Sehr große Klassen deuten auf ein schlechtes objekt-orientiertes Design hin. Je größer der Umfang einer Klasse, desto wahrscheinlicher ist die Existenz von Code-Duplikaten. In großen Klassen ist der Aufbau und das Zusammenspiel der Methoden meist weniger gut sichtbar als in kompakten Klassen. Es gelten prinzipiell die selben Punkte wie bei Long Method (Kapitel 3.2). Da es auch Klassen geben kann die groß sein müssen spielt die Erfahrung und der spezifische Kontext bei der Festlegung von maximalen Höchstwerten von Klassen eine große Rolle. Beispielsweise weisen GUI-Klassen von Grund auf einen größeren Umfang auf da dort alle Zustände und Reaktionsmethoden implementiert werden.

Meist geht die Größe der Klasse mit der Anzahl der Attribute und Methoden einher wodurch sich Number of Attributes und Number of Methods als Metriken anbieten. Hier sei aber zu beachten das durch Vererbung die Summe aller Methoden oder Attribute in einer Klasse noch ansteigen kann.

3.4 Long Parameter List

Methoden mit sehr großen Parameterlisten sind nicht im Sinne der Objektorientierung. Eine Methode sollte nur so viele Parameter besitzen, wie sie für die Lösung der Aufgabe benötigt. Es sollten nur Daten übergeben werden, die sich die Methode nicht selbst beschaffen kann oder deren Beschaffung sehr kosten-

intensiv sind (z.B. Datenbankanfragen). Die anderen Daten kann die Methode durch lokale Variablen, Instanzvariablen oder durch Aufruf von Methoden auf anderen Objekten erlagen. Weiterhin können viele primitive Werte in neuen Datenobjekten gekapselt und an die Methode übergeben werden. Ein Beispiel wäre ein Adress-Objekt, welches aus Postleitzahl, Straße und Hausnummer besteht. Ändert sich die Struktur der Anwendung, indem z. B. ein Feld Hausnummerzusatz eingeführt wird, so muss die Methode nicht verändert werden, da das Adress-Objekt als Parameter übergeben wird. Lange Parameterlisten sind für einen Entwickler schwerer zu verstehen, schwerer zu benutzen und werden häufiger inkonsistent.

Die Metrik, die hier benötigt wird um diesen Code Smell zu entdecken, wird als Number of Parameters bezeichnet.

3.5 Divergent Change

Divergent Change in einer Klasse liegt dann vor, wenn für unterschiedliche Arten von Erweiterungen immer die gleiche Klasse geändert werden muss. Wenn also beispielsweise das System an eine neue Datenbankversion, Präsentationssicht, etc. angepasst wird und dabei jedes Mal ein und dieselbe Klassen geändert werden muss spricht man von Divergent Change. Die Erweiterbarkeit lässt zu Wünschen übrig da man anscheinend unterschiedliche Konzepte in dieser Klasse integriert hat und für eine Art von Erweiterung möglichst nur eine Klasse ändern sollte.

Dieser Code Smell ist nur sehr schwer eindeutig durch Metriken zu entdecken. Insbesondere benötigt man hier Änderungsmetriken, die anzeigen das es Klassen gibt die für die unterschiedlichsten Erweiterungen geändert werden. Grundsätzlich sind hier Regeln besser angebracht um diesen Code Smell zu entdecken.

3.6 Shotgun Surgery

Shotgun Surgery stellt das Gegenteil zu Divergent Change (Kapitel 3.5) dar. Im Gegensatz zu Divergent Change sind bei Shotgun Surgery mehrere Klassen von einer spezifischen Art von Änderung betroffen. Wenn also beispielsweise das System an eine neue Datenbankversion angepasst wird und dabei jedes Mal die beiden gleichen Klassen geändert werden, spricht man von Shotgun Surgery. Die Erweiterbarkeit lässt zu Wünschen übrig, da man pro Erweiterung möglichst wenig (optimalerweise nur eine Klasse) am System ändern sollte. Durch Shotgun Surgery steigt die Fehleranfälligkeit, da der Entwickler bei einer Erweiterung vergessen kann, Änderungen an allen Klassen vorzunehmen.

Shotgun Surgery ist ebenfalls nur sehr schwer eindeutig durch Metriken zu entdecken. Insbesondere benötigt man hier Änderungsmetriken die Anzeigen das es Klassen in Gruppen geändert werden. Grundsätzlich sind hier Regeln besser angebracht um diesen Code Smell zu entdecken.

3.7 Feature Envy

Ein schlechtes objekt-orientiertes Design kann sich auch dadurch zeigen, dass sich Methoden in einer „falschen" Klasse befinden. In solch einem Fall greift eine Methode mehr auf externe Klassen zu als auf die eigene Klasse, in der sie sich befindet. Dieses Code Smell zeigt sich sehr deutlich, wenn eine Methode mehrere Methoden eines anderen Objektes aufruft, um lokale Variablen zu initialisieren. Die Methode holt sich also hauptsächlich Daten für die Berechnung von einem anderen Objekt.

Metriken zur Entdeckung von Ferature Envy sind Number of external Accesses im Verhältnis zu Number of internal Accesses. Zugriffe (engl. Access) sind dabei Methodenaufrufe oder Instanziierungen von Klassen. Allerdings sind bei der Auswertung Ausreißer mit Umsicht zu behandeln. In bestimmten Klassen ist es unerlässlich, dass Methoden fast ausschließlich sehr viele Aufrufe auf anderen Objekten aufrufen. Ein Beispiel für diesen Fall wäre eine Klasse die ein Facade Pattern realisiert.

3.8 Data Clumps

Unter Data Clumps sind Anhäufungen von logisch zusammengehörigen Daten zu verstehen, die an mehreren Stellen im Code in derselben Konstellation auftreten. Beispielsweise ein Adress-Cluster welcher an mehreren Stellen im System vorkommt und bei einer Änderung mehrmals geändert werden muss. In einem guten objekt-orientierten Design werden alle Teilobjekte, die eine Adresse ausmachen, in einem Adress-Objekt gekapselt. Dadurch verringert sich Code-Redundanz und Umfang der Klasse. Zusätzlich wird die Klasse verständlicher.

Solche mehrfach auftretende Datenanhäufungen sind schwer durch ein Tool zu erkennen, da es sich selten um vollständig identische Clone handelt. Ein Messwerkzeug müsste also das parallel auftreten derselben Attribute entdecken. Grundsätzlich sind hier Regeln besser angebracht, um diesen Code Smell zu entdecken.

3.9 Duplicated Code

Analog zum mehrfachen Auftreten von Daten im Quellkode können auch Funktionalitäten häufiger im System auftauchen da die Entwickler aus Performanz-

gründen diese z.B. mittels Copy & Paste kopiert und modifiziert haben. Das Problem bei dupliziertem Quellkode (bzw. Code Clones) ist das Änderungen an der Funktion in mehreren – potentiell unbekannten Stellen – nachgeholt werden müssen.

Auch solche mehrfach auftretende Funktionsanhäufungen sind schwer durch ein Tool zu erkennen (insb. wenn Sie leicht modifiziert wurden), da es sich selten um vollständig identische Kopien handelt. Ein Messwerkzeug müsste also das auftreten derselben Funktions-Konstruktion in unterschiedlichen Methoden entdecken. Grundsätzlich sind hier Regeln besser angebracht, um diesen Code Smell zu entdecken.

3.10 Switch Statement

Switch Statements sollten in einem objektorientierten Design keine Verwendung finden, da man dies (insb. bei Verwendung von Datenobjekten) über die Benutzung der Polymorphie realisieren kann. Häufig gehen mit Switch Statements Code-Duplizierung einher, da diese häufiger im Code vorkommt und bei Änderungen mehrmals im Code geändert werden muss.

Ein Messwerkzeug sollte Klassen, die Switch Statements verwenden, melden. Der Entwickler sollte bei der Analyse dann entscheiden, ob das Switch-Konstrukt bleiben kann oder durch Refaktorierung eliminiert werden muss.

3.11 Primitive Obsession

Primitive Datentypen wie „int“ oder „char“ bilden die Grundbausteine, auf denen „echte“ Objekte wie Integer oder String aufbauen. Obwohl primitive Datentypen einfach zu verwenden sind und in Spezialfällen auch effizienter sind, sollten komplexere Datentypen als Objekte mit eigenen Funktionen realisiert werden.

Als Metrik kann man hier insb. Number of Primitive Data verwenden und versuchen, Häufungen von primitiven Datentypen durch eigene neue Objekte zu ersetzten.

3.12 Parallel Inheritance Trees

Parallele Vererbungshierarchien bilden einen Spezialfall von Shotgun Surgery dar. Sie treten auf, wenn bei einer Spezialisierung einer Klasse (d.h. Ableitung einer Subklasse) in einem Vererbungsbaum auch in anderen Vererbungsbäumen Subklassen abgeleitet werden müssen.

Dieser Code Smell ist ebenfalls nur sehr schwer eindeutig durch Metriken zu entdecken. Insbesondere benötigt man hier Änderungsmetriken, die anzeigen das Klassen parallel geändert werden. Grundsätzlich sind hier Regeln besser angebracht, um diesen Code Smell zu entdecken.

3.13 Lazy Class

Klassen sollten genug Funktionalität besitzen und von anderen Objekten gebraucht werden, damit sich der Aufwand ihrer Erzeugung und ihrer Wartung lohnt. Dieser Code smell entsteht durch eine Überspezialisierung oder im Laufe einer Refaktorierung.

Metriken um Lazy classes zu entdecken sind u. A. Number of Methods, Number of Attribute oder Number of Users (bzw. Using Classes). Unterschreiten die Anzahl der Attribute, Methoden, oder Benutzern einen zu definierenden Schwellwert, so steigt die Wahrscheinlichkeit das dies eine Lazy Class darstellt.

3.14 Speculative Generality

Häufig möchte sich ein Entwickler einige Optionen für spätere Erweiterungen offenhalten. Werden allerdings diese Vorhaben nie umgesetzt, existiert womöglich einige komplizierte Klassen oder Konstruktionen, die nicht benötigt werden und nur das Verstehen behindern. Dies drückt sich beispielsweise durch nie verwendete Parameter aus oder durch abstrakte Klassen, die wenig Arbeit verrichten. Das Resultat dieser spekulativen Erweiterbarkeit ist eine schwer zu verstehende Klasse.

Eine Möglichkeit Speculative Generality zu identifizieren ist die Identifikation von totem Code (d.h. Quellkode der nie ausgeführt wird) mittels dynamische Metriken wie Number of Invocations oder number of Instantiations. Andere Arten dieses Code Smells, wie beispielsweise unbenutzte Parameter, können besser mittels spezifischer Regeln und Analysen ermittelt werden.

3.15 Temporary Field

Wenn Instanzvariablen nur in bestimmten Fällen gesetzt werden, trägt das nicht unbedingt zur Verständlichkeit der Klasse bei. Meistens werden diese Instanzvariablen aus Gründen der Bequemlichkeit als temporäre Variablen für einen Algorithmus verwendet, der sich womöglich über mehrere Methoden erstreckt. Ansonsten sind diese Variablen für die Klasse nicht weiter von Bedeutung. Der Entwickler möchte dadurch der Verwendung einer großen Anzahl von Parameterlisten in Methoden entgegenwirken. Allerdings trägt diese Art der Verwen-

dung von Instanzvariablen nicht zur Lesbarkeit bei, da ein Leser in der Regel davon ausgeht, dass alle Instanzvariablen für ein Klasse von Bedeutung sind.

Temporary Field ist nur sehr schwer eindeutig durch Metriken zu entdecken. Insbesondere benötigt man hier Metriken, die die Nutzung und Initialisierung (z.B. in den Konstruktoren) von Instanzvariablen charakterisieren. Grundsätzlich sind hier aber Regeln besser angebracht, um diesen Code Smell zu entdecken.

3.16 Message Chains

Eine Klasse sollte immer von so wenig anderen Klassen wie möglich abhängen, da bei Änderungen dann nur wenige andere Klassen zu ändern sind. Message Chains entsprechen nicht diesem Prinzip und werden deshalb als Code Smells angesehen. Dabei handelt es sich z.B. um Objektaufrufe der Form ((obj.getWorld(X)).getContinent(Y)).getCountry(Z). Die aufrufende Klasse navigiert über eine bestimmte Anzahl von Aufrufen zum gewünschten Objekt einer anderen Klasse. Bei der Verwendung solcher Konstrukte ist die aufrufende Klasse abhängig von der Art der Navigation, was zu häufigen Änderungen im Code führen kann.

Das Grundprinzip nennt sich „Law of Demeter" und schlägt vor, dass eine Klasse nur mit seinen direkten Nachbarn Nachrichten austauschen soll. Als Metrik müsste ein Werkzeug die „Zugriffstiefe" ermitteln, die dann ab einem spezifischen Schwellwert untersucht wird.

3.17 Middle Man

Dieser Code Smell bezeichnet Klassen die (zu) viele Aufräge (Methodenaufrufe) an andere Klassen delegiert. Obwohl dies bei speziellen Konstruktionen wie dem Facade Pattern gewünscht ist, kann es häufig den Code einfach nur komplizierter machen.

Eine Möglichkeit einen Middle Man zu identifizieren ist die Identifikation von Methoden, die im Verhältnis zu Ihrer Länge viele Delegationen verwenden (insb. Methoden nur mit einer Zeile und einer Delegation). Dazu muss ein Messwerkzeug also zumindest Metriken wie Number of Delegations und LOC für Methoden berechnen.

3.18 Inappropriate Intimacy

Verkapselung und Information Hiding ist ein wichtiges Konzept in der Objektorientierung. Interne Details werden vor der Außenwelt verborgen und Zugriffe sollten nur über Schnittstellen erfolgen, um spätere Änderungen nicht auf zu

Benutzende Klassen auszuweiten. Einige Klassen greifen aber trotzdem sehr stark auf andere Klassen zu oder sind durch Vererbung sehr stark miteinander verwoben.

Metriken um Inappropriate Intimacy festzustellen, sind selten da die Anzahl der Zugriffe auf eine spezifische andere Klasse ermittelt werden und somit potentiell für jede andere Klasse im System eine Metrik existieren müsste.

3.19 Alternative Classes with Different Interfaces

Einige Klassen haben zwar die gleiche oder eine ähnliche Aufgabe aber besitzen keine gemeinsame Superklasse oder benutzen ein gemeinsames Interface. In diesem Fall sollten sie angepasst werden, um einerseits die Polymorphie und andererseits die Verständlichkeit zu unterstützen.

Um diesen Code Smell zu entdecken, muss die funktionelle oder strukturelle Ähnlichkeit zwischen zwei Klassen bestimmt werden, was heutzutage nicht über Messwerkzeuge abgedeckt wird.

3.20 Incomplete Library Class

Bei der Verwendung von Softwarebibliotheken kann es vorkommen, dass eine Methode oder eine Klasse nicht genau das bereitstellt, was man gerade benötigt. In diesem Fall kann häufig die Bibliothek nicht geändert werden, und deshalb muss ein Wrapper um diese Klasse gestülpt werden.

Dieser Code Smell wird eigentlich immer dann gefunden, wenn der Entwickler das Problem hat, die Bibliothek zu benutzen und keine geeignete Klasse findet.

3.21 Data Class

Dabei handelt es sich um Klassen, die ausschließlich als Datencontainer fungieren und keine Funktionalität implementieren. Sie besitzen nur Felder sowie Set- und Get-Methoden, da Instanzvariablen nie öffentlich zugänglich sein sollten. Typischerweise werden diese Klassen zu stark von anderen Klassen modifiziert und Funktionalitäten, um die Daten zu verändern, sind auf diese verteilt.

Um eine Data Class zu entdecken, benötigt man ein Messwerkzeug, welches öffentliche Attribute entdeckt und das das Verhältnis zwischen Attributen und Methoden (insb. ohne Set-/Get-Methoden) bestimmen kann.

3.22 Refused Bequest

In einem schlechten Design kann es Subklassen geben, die einige der geerbten Methoden und Felder überhaupt nicht verwenden. Noch problematischer ist es, wenn die Subklasse zwar Funktionalität wiederverwendet aber mit der Schnittstelle der Superklasse nicht einverstanden ist. In solch einem Fall sollte die Vererbungshierarchie noch einmal überdacht werden. Bei dem Einsatz von Vererbung wird Wiederverwendung meist das Hauptanliegen sein. Dabei wird es so gut wie immer vorkommen, dass Teile der geerbten Funktionalität für die Subklasse nicht notwendig wäre. Diese Tatsache muss akzeptiert werden, ansonsten müsste auf Vererbung verzichtet werden. Erst wenn eine Subklasse so gut wie keine Funktionalität der Superklasse nutzt, liegt ein Code Smell vor.

Aus den Messdaten sollte also ersichtlich sein, wie viele Methoden die Subklassen von Ihren Superklassen selbst verwenden oder von anderen verwendet werden. Metriken sind also Number of Method users und Number of Attribute Users auf jeder ebene der Vererbung. Insbesondere sollte aus der Analyse ersichtlich sein, wenn eine Subklasse Methoden oder Felder der Superklasse niemals verwendet.

3.23 (Superfluous) Comments

An sich stellen Kommentare kein Problem dar, sondern sind wünschenswert, da mit ihnen der Quellkode erklärt wird. Ist das Verhältnis zwischen Kommentaren zu Quellkode sehr hoch (viel mehr Kommentare als Kode), stellt sich die Frage, ob der Quellkode wirklich so schlecht selbsterklärend ist oder ob überflüssige Kommentare das Verständnis behindern. Häufig ist gut strukturierter und verständlicher Code selbsterklärend und Kommentare weisen auch auf schlechten Code hin. Wenn ausführlicher Kommentar zum Verständnis eines logischen Code-Blocks notwendig ist, dann sollte der Code refaktoriert werden. Dabei sollten Variablen- und Methodennamen so gewählt werden, dass die Bedeutung des Codes mit vermittelt wird.

Ein Messwerkzeug muss hier Methoden mit einem sehr hohen Verhältnis von Kommentar zur eigentlichen Funktionalität identifizieren. Ab einem Verhältnis, über einem Schwellwert (z.B. 2 Komentarzeilen zu 1 LOC), sollte der Code inspiziert werden.

4 Messwerkzeuge zur Statischen Softwareanalyse

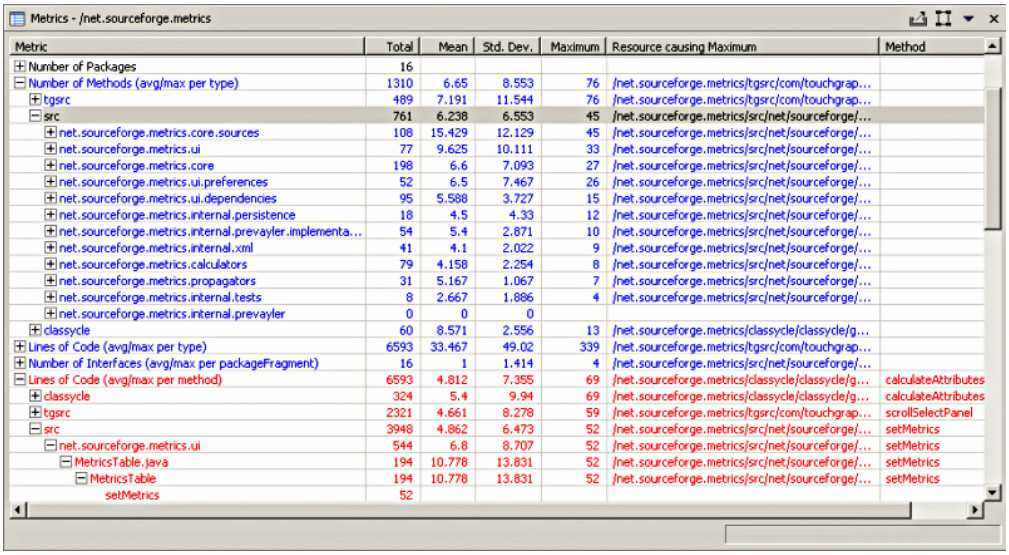
Messwerkzeuge für die statische Softwareanalyse unterstützen den Entwicklungsprozess, um frühzeitig Fehlerquellen aufzudecken. Diese Werkzeuge sind ausgerichtet, um den Aufwand zur Erkennung und Behebung von Designmängeln zu verringern.

Es sei bereits hier erwähnt, dass diese Tools nicht für eine vollständige statische Softwareanalyse herangezogen werden können, da jedes auf eine spezielle Aufgabe ausgerichtet ist. Häufig werden zwar Softwaremetriken berechnet, allerdings stehen keine oder wenig ausreichende Funktionen zur Interpretation und Verbesserung der Codequalität zur Verfügung.

In der vorliegenden Arbeit wird der Schwerpunkt auf statische Softwaremetriken gelegt und in diesem Kapitel werden nun einige Messwerkzeuge vorgestellt. Wir untersuchten im Wesentlichen freie Messwerkzeuge wie eclipse Metrics, TeamInABox oder jDepend. Als kommerzielles Werkzeug führen wir zusätzlich noch Code Pro Studio von Instantiations auf.

4.1 Metrics Eclipse Plugin

Bei diesem Messwerkzeug handelt sich um ein Plugin für die eclipse IDE, welches von http://metrics.sourceforge.net/ bezogen werden kann. Es dient zur statischen Analyse von Eclipse-Projekten. Das Programm stellt eine Reihe von Metriken zur Verfügung, mit deren Hilfe es möglich sein soll, Eclipse-Projekte auf Qualitäteigentschaften zu untersuchen. Nachdem Java-Projekte ihr Interesse an einer Metrikanalyse bekundet haben (in den Projekteigenschaften), wird der Analysevorgang durch das Kompilieren des Projekts angestoßen. Dabei stehen dem Benutzer zwei Sichten zur Verfügung. Wie in Abbildung 3 dargestellt, wird nach Beendigung der Berechnungen die Ergebnissicht mit den gemessenen Daten gefüllt. Die Ergebnisse der Analyse werden dabei tabellarisch in einer Baumstruktur dargestellt.



Metric	Total	Mean	Std. Dev.	Maximum	Resource causing Maximum	Method
Number of Packages	16					
Number of Methods (avg/max per type)	1310	6.65	8.553	76	/net.sourceforge.metrics/tgsrc/com/touchgrap...	
tgsrc	489	7.191	11.544	76	/net.sourceforge.metrics/tgsrc/com/touchgrap...	
src	761	6.238	6.553	45	/net.sourceforge.metrics/src/net/sourceforge/...	
net.sourceforge.metrics.core.sources	108	15.429	12.129	45	/net.sourceforge.metrics/src/net/sourceforge/...	
net.sourceforge.metrics.ui	77	9.625	10.111	33	/net.sourceforge.metrics/src/net/sourceforge/...	
net.sourceforge.metrics.core	198	6.6	7.093	27	/net.sourceforge.metrics/src/net/sourceforge/...	
net.sourceforge.metrics.ui.preferences	52	6.5	7.467	26	/net.sourceforge.metrics/src/net/sourceforge/...	
net.sourceforge.metrics.ui.dependencies	95	5.588	3.727	15	/net.sourceforge.metrics/src/net/sourceforge/...	
net.sourceforge.metrics.internal.persistence	18	4.5	4.33	12	/net.sourceforge.metrics/src/net/sourceforge/...	
net.sourceforge.metrics.internal.prevayler.implementa...	54	5.4	2.871	10	/net.sourceforge.metrics/src/net/sourceforge/...	
net.sourceforge.metrics.internal.xml	41	4.1	2.022	9	/net.sourceforge.metrics/src/net/sourceforge/...	
net.sourceforge.metrics.calculators	79	4.158	2.254	8	/net.sourceforge.metrics/src/net/sourceforge/...	
net.sourceforge.metrics.propagators	31	5.167	1.067	7	/net.sourceforge.metrics/src/net/sourceforge/...	
net.sourceforge.metrics.internal.tests	8	2.667	1.886	4	/net.sourceforge.metrics/src/net/sourceforge/...	
net.sourceforge.metrics.internal.prevayler	0	0	0			
classycle	60	8.571	2.556	13	/net.sourceforge.metrics/classycle/classycle/g...	
Lines of Code (avg/max per type)	6593	33.467	49.02	339	/net.sourceforge.metrics/tgsrc/com/touchgrap...	
Number of Interfaces (avg/max per packageFragment)	16	1	1.414	4	/net.sourceforge.metrics/src/net/sourceforge/...	
Lines of Code (avg/max per method)	6593	4.812	7.355	69	/net.sourceforge.metrics/classycle/classycle/g...	calculateAttributes
classycle	324	5.4	9.94	69	/net.sourceforge.metrics/classycle/classycle/g...	calculateAttributes
tgsrc	2321	4.661	8.278	59	/net.sourceforge.metrics/tgsrc/com/touchgrap...	scrollSelectPanel
src	3948	4.862	6.473	52	/net.sourceforge.metrics/src/net/sourceforge/...	setMetrics
net.sourceforge.metrics.ui	544	6.8	8.707	52	/net.sourceforge.metrics/src/net/sourceforge/...	setMetrics
MetricsTable.java	194	10.778	13.831	52	/net.sourceforge.metrics/src/net/sourceforge/...	setMetrics
MetricsTable	194	10.778	13.831	52	/net.sourceforge.metrics/src/net/sourceforge/...	setMetrics
setMetrics	52					

Abbildung 3 Ergebnissicht beim eclipse metrics plugin

Die Anzeigereihenfolge der Metriken, sowie die Farbgebung der Tabellenwerte (inkl. Schwellwerte), können in den Einstellungen des Plugins festgelegt werden. Dadurch ist für einen Benutzer schnell ersichtlich, ob Resourcen (Pakete, Klassen, ...) existieren, die sich außerhalb des erlaubten Wertebereichs befinden. Außerdem wird diejenige Resource angezeigt, die den Höchstwert für das betrachtete Maß aufweist. Durch einen Doppelklick gelangt man direkt an die betroffene Stelle im Quellcode. So gesehen handelt es sich dabei um eine Anomalie, die es zu untersuchen gilt. Damit das Programm Warnungen für Werte außerhalb des zulässigen Bereichs anzeigt, muss in den Programmeinstellungen die dafür vorgesehne Option aktiviert werden.

Ein weiterer wichtiger Punkt in den Programmeinstellungen stellt „*Safe Ranges*“ dar. Hier können die zulässigen Wertebereiche (Min und Max) für alle Metriken definiert werden. Nur wenige Metriken haben Standardwerte, da jedes Projekt seine eigenen Eigenschaften aufweist und spezielle Schwellwerte angepasst werden müssen. Werte außerhalb des gesteckten Bereichs führen zu Warnungen. Zusätzlich kann in der Spalte „*Hint for fix*“ mögliche Lösungsvorschläge gegeben werden, falls der erlaubte Bereich nicht eingehalten wird. Das Programm rät dem Benutzer beispielsweise bei LOC, dass bei einer Überschreitung des maximalen Werts, eine Refaktorierung in Form von „Extract Method“ das Problem lösen könnte. Somit ist es am Anfang der Implementierungsphase möglich, die Wertebereiche für alle Metriken zu definieren. Bei jedem Kompilierungsvorgang wird die Ergebnissicht aktualisiert, und der Benutzer bekommt potentielle Kandidaten für eine Refaktorierung angezeigt. Somit kann die Softwarequalität fortlaufend während der Implementierung überwacht und verbessert werden.

4.1.1 Verwendete Metriken

Es werden sowohl einfache als auch komplexere *Metriken* berechnet. Vertreter von einfachen Metriken sind `Anzahl aller Klassen und Interfaces`, `Anzahl abstrakter Klassen`, `Anzahl der Instanzvariablen` oder `Lines of Code`. Eine Auflistung aller verfügbaren Metriken zeigt Tabelle 4.

Tabelle 4 Verwendete Metriken in Metrics Eclipse Plugin

Verwendete Metriken	
Number of Classes	Number of Children
Number of Interfaces	Depth of Inheritance Tree
Number of Overridden Methods	Number of Fields
Lines of Code	Specialization Index
Mc Cabe's Cyclomatic Complexity	Weighted Methods per Class
Lack of Cohesion in Method	Afferent Coupling
Efferent Coupling	Instability
Abstractness	Normalized Distance from Main Sequence

Das Plugin berechnet die `Anzahl der Kinder` für eine Klasse, wobei Kindklassen alle direkten Subklassen einer Klasse darstellen. Zusätzlich gilt eine Klasse als Kindklasse eines oder mehrerer Interfaces, wenn sie das Interface bzw. die Interfaces implementiert. Weitere Metriken sind `Anzahl überschriebener Methoden` (NORM), `Tiefe der Vererbungshierarchie` (DIT) und der Spezialisierungsgrad einer Klasse. Letzteres benötigt zur Berechnung `Afferent Coupling`, was sich durch die Anzahl der Klassen berechnet, die sich außerhalb eines Pakets befinden und sich auf Klassen innerhalb dieses Pakets beziehen. Dagegen beschreibt `Efferent Coupling` die Anzahl der Klassen, die sich innerhalb eines Pakets befinden und von Klassen außerhalb dieses Pakets abhängig sind. Komplexe Metriken werden ebenfalls berechnet, wie z. B. McCabe's `Cyclomatic Complexity` oder `Lack of Cohesion in Methods`.

4.1.2 Weitere Features

Weiterhin besteht die Möglichkeit Klassen- bzw. Paketabhängigkeiten grafisch darzustellen. Dazu gibt es eine eigene Sicht „*Dependency Graph View*“. Auf der Projekt-Website wird genau beschrieben, wie dieser Graph (siehe Abbildung 4) zur Verminderung von Abhängigkeiten verwendet und somit zur Verbesserung der Projektqualität genutzt werden kann.

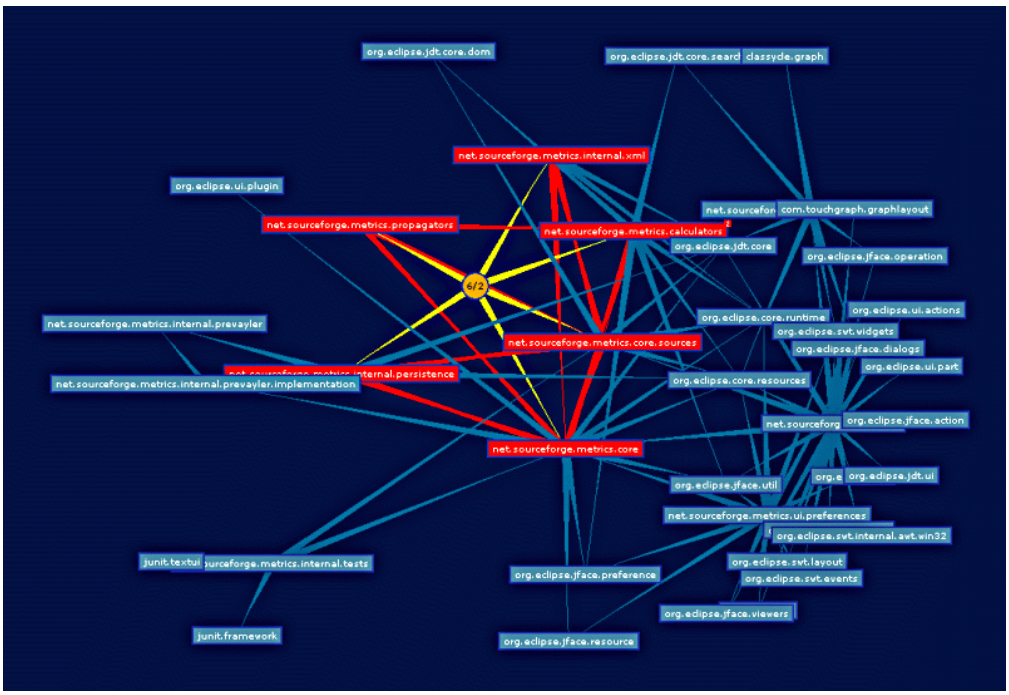


Abbildung 4 Darstellung von Klassen- und Paketabhängigkeiten

Der Dokumentation zu diesem Plugin sind so gut wie keine Informationen darüber zu entnehmen, welche Aussagen bezüglich der Qualität des Software-Systems auf Basis der einzelnen Metriken gemacht werden kann. Zwar können zulässige Wertebereiche und Hinweise zur Reduzierung dieser Werte gemacht werden, allerdings muss dies explizit durch den Benutzer getan werden. Es bedarf also einer Recherche des Benutzers nach den verwendeten Metriken, um Schlussfolgerungen bezüglich Software-Qualität machen zu können. Nur mit entsprechendem Wissen können dann die Wertebreeiche für alle Metriken für ein Projekt bestimmt werden.

Weiterhin besteht die Möglichkeit die erhobenen Maße in eine *XML-Datei* zu exportieren. Ansonsten existieren keine weiteren Exportfunktionen. Der Import von Werten ist nicht möglich.

4.1.3 Fazit

Alles in allem macht das Plugin einen guten Eindruck. Es fügt sich nahtlos in die Eclipse-Workbench ein. Über zwei Sichten sowie zwei Einstellungsmasken kann eine Analyse von Eclipse-Projekten durchgeführt werden. Die zulässigen Wertebereiche für die Metriken können für jedes Projekt individuell bestimmt werden. Dieses Tool kann zur Unterstützung der Implementierungsphase. Es bietet sich besonders dann an, wenn die Entwicklung in Eclipse erfolgt, so dass nicht meh-

rere voneinander unabhängie Programme verwendet werden müssen. Eine Analyse des Projekts erfolgt automatisch bei jedem Kompiliervorgang.

Die Szenarien, die in Sektion 1.2 ausführlicher beschrieben sind, wurden folgendermaßen bewertet:

- Z1 (Qualitätsreporting): Die berechneten Werte der Metriken können in XML exportiert werden und mittels eigens herzustellender XSL-Transformatoren in andere Formate umgewandelt werden.
- Z2 (Anomalieanalyse): Die Identifikation von Anomalien wird gut unterstützt. Es können eigene Schwellwerte festgelegt werden um Anomalien zu entdecken. Relative Anomalien bzgl. eines berechneten Wertes wie dem arithmetischen Mittel oder des Median sind nicht möglich.
- Z3 (QD-Entdeckung): Qualitätsdefekte werden nicht explizit erkannt. Die Angabe der Verbesserungsvorschläge kann Refaktorierungen beinhalten und somit den Entwickler bei einigen Qualitätsdefekten wir „Long Method" unterstützen.

4.2 TEAMINABOX Eclipse Metrics Plugin

Dieses Plugin für die eclipse IDE verfolgt das Ziel, dem Entwickler während des Entwicklungsprozesses Hilfsmittel zur Verfügung zu stellen, damit dieser einen ständigen Überblick über den Zustand des Softwaresystems hat. Im Quellcode werden kritische Bereiche gekennzeichnet und erläutert (Abbildung 5).

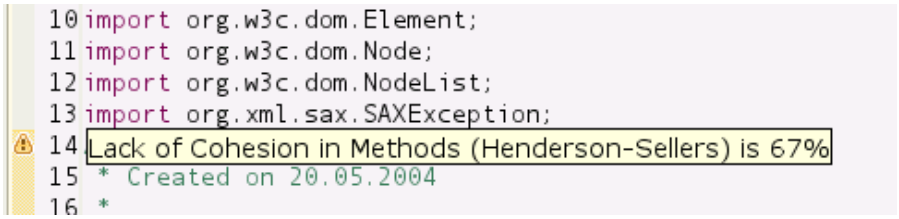


Abbildung 5 Metrik-Werte werden direkt im Quellcode angezeigt

Dabei werden in der Task-Liste von Eclipse Warnungen angezeigt, wenn die Schwellwerte der Metriken überschritten werden. Auch hier müssen Projekte für die Betrachtung explizit in den Projekteinstellungen ausgewählt werden. Es besteht die Möglichkeit einzelne Java-Dateien eines ausgewählten Pakets aus der Betrachtung auszuschließen. Wie in Abbildung 6 dargestellt, können die maximal erlaubten Höchstwerte für die Metriken definiert werden. Werden diese Werte überschritten, so wird dies in der Task-Liste und direkt im Quellcode der betroffenen Ressource vermerkt.

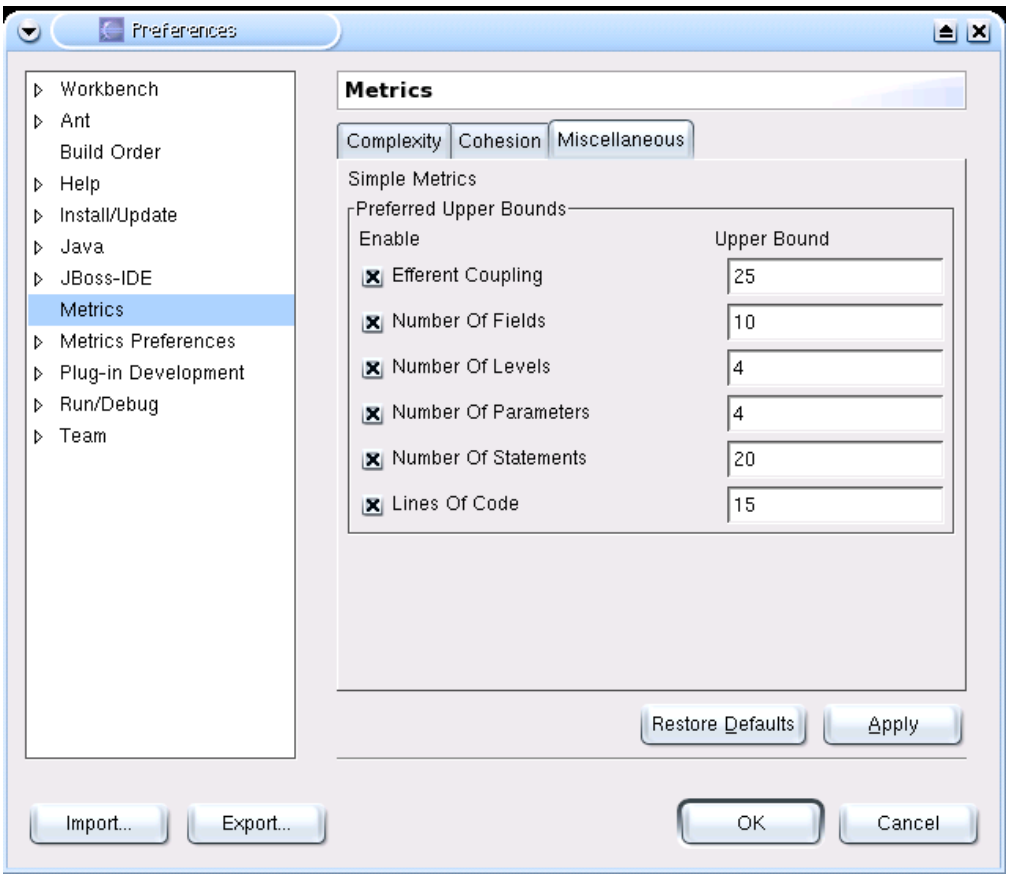


Abbildung 6 Festlegen der Schwellwerte für die einzelnen Metriken

Wie bereits beim Metrics Eclipse Plugin (siehe Sektion 4.1) wird eine Metrik-Analyse durch einen Kompiliervorgang gestartet. Es werden hauptsächlich Metriken auf Methodenbasis verwendet. Die Dokumentation beschreibt, welche Aussagekraft die einzelnen Metriken haben. Es wird erklärt, wie die Werte zu interpretieren sind und dieses Wissen zur Code-Verbesserung herangezogen werden kann.

Das Auffinden von Anomalien ein weit aufwändigeres Unterfangen. Hier werden die erhobenen Maße nicht tabellarisch aufgelistet, sondern müssen aus der Task-Liste oder aus dem Quell-Code zeitaufwändig extrahiert werden.

4.2.1 Verwendete Metriken

Teaminabox unterstützt eine Reihe von Metriken welche in Tabelle 5 aufgelistet sind. McCabe's Cyclomatic Complexity kann zur Bewertung der Komplexität einer Methode herangezogen werden. Die Summe aller zyklomatischen Komple-

xitäten aller Methoden werden auch in der Berechnung des Wertes für eine Klasse berücksichtigt.

Efferent Couplings für eine Klasse gibt die Anzahl aller Klassen und Schnittstellen an, auf die sich diese Klasse in irgendeiner Form bezieht. Das beinhaltet Vererbung, implementierte Schnittstellen, Parameter in Methodensignaturen, Variablendeklarationen sowie geworfene und abgefangene Ausnahmen. Ein hoher Wert deutet darauf hin, dass die betrachtete Klasse eventuell in mehrere Klassen aufgesplittet werden sollte oder verkleinert werden solte. Er gibt die Stabilität einer Klasse zu alle seiner abhängigen Klassen und Schnittstellen an.

Tabelle 5 Verwendete Metriken in TEAMINABOX Metric Plugin

Verwendete Metriken	
Mc Cabe's Cyclomatic Complexity	Efferent Coupling
Lack of Cohesion in Method	Lines of Code in Method
Number of Fields	Number of Levels
Number of Parameters	Number of Statements
Weighted Methods per Class	

Durch Lack of Cohesion in Methods soll dem Entwickler gezeigt werden, ob eine Klasse eine einfache oder eine mehrfache Abstraktion repräsentiert. Wenn eine Klasse mehr als eine Abstraktion darstellt, dann sollte ein Refactoring in mehrere kleinere Klassen erfolgen, wobei jede dieser neuen Klassen eine einfache Abstraktion repräsentiert. Hier bedeutet ein niedriger Wert eine gute Kohäsion.

Number Of Levels gibt die maximale Anzahl von Schachtelungen in einer Methode an. Eine hohe Schachtelung deutet auf eine hohe Komplexität hin und führt somit zu geringerer Verständlichkeit. Häufig, aber nicht immer, arbeiten solche Methoden auf der untersten Abstraktionsstufe oder auf verschiedenen Abstraktionsstufen, was auch zu Verminderung der Verständlichkeit führen kann. Methoden mit hoher Schachtelung können vereinfacht werden, indem beispielsweise private Methoden angelegt werden und somit Code ausgelagert und wieder verwendet wird.

Das Werkzeug berechnet auch einfache Metriken wie z. B. Lines of Code in Method oder Number Of Fields.

4.2.2 Fazit

Das Plugin bietet neben dem bereits beschriebenen Funktionsumfang nur noch wenige zusätzliche Features. Die erhobenen Metrikwerte können nach HTML und CSV exportiert werden. Ansonsten gibt es keine weiteren Funktionen. Eine grafische Repräsentation der Metrik-Werte sucht man vergebens.

Da es sich hierbei um ein Eclipse-Plugin handelt kann es am ehesten mit Metrics Eclipse Plugin (Sektion 4.1) verglichen werden. Sieht man vom unterschiedlichen Umfang der unterstützten Metriken ab, so kann prinzipiell dieselbe Qualitätsanalyse durchgeführt werden. Allerdings gestaltet sich dieser Vorgang bei TEAMINABOX wesentlich aufwändiger, weil die Resultate nicht in kompakter Form auf einen Blick präsentiert werden. Eine Verbesserung dieser Situation stellt der Export in eine HTML-Darstellung dar. Danach kann der Benutzer eine Untersuchung der erhobenen Daten vornehmen.

Die Szenarien, die in Sektion 1.2 ausführlicher beschrieben sind, wurden folgendermaßen bewertet:

- Z1 (Qualitätsreporting): Die berechneten Werte der Metriken können in HTML und CSV exportiert werden und somit in den großen Analysewerkzeugen (z.B. Excel oder Statistika) verwendet werden.
- Z2 (Anomalieanalyse): Das Programm bietet wenig Unterstützung für das entdecken von Anomalien. Eine Möglichkeit besteht darin, vor Beginn der Implementierung in den Einstellungen die maximalen Höchstwerte für die einzelnen Metriken zu definieren. In der Analysephase werden alle Wertüberschreitungen entdeckt und können als Anomalien angesehen werden, die es gilt weiter zu untersuchen.
- Z3 (QD-Entdeckung): Qualitätsdefekte werden nicht explizit erkannt. Es werden keine expliziten Vorschläge für mögliche Refaktorierungen gemacht.

4.3 JDepend

JDepend ist eine eigenständige Java-Applikation, die unter BSD Lizenz steht. Das Tool zielt darauf ab Entwickler bei ihrer Arbeit zu unterstützen, indem Metriken erhoben werden, die als Anhaltspunkte zu möglichen Refaktorierungsmaßnahmen herangezogen werden können.

Design-Qualität kann durch Erweiterbarkeit, Wiederverwendbarkeit und Wartbarkeit beschrieben werden. Alle diese Aspekte werden auch von internen Paketabhängigkeiten beeinflusst. An diesem Punkt setzt JDepend an. Die Erweiterbarkeit eines Software-Designs ist vom Grad der Unabhängigkeit bezüglich der Implementierungsdetails abhängig. Je unabhängiger Designs von den Details der Implementierung sind, desto höher ist der Grad der Erweiterbarkeit. Ähnlich verhält es sich mit der Wartbarkeit. Wartbarkeit verbessert sich, wenn Änderungen an der Software einfach umgesetzt werden können, ohne viele Veränderungen und Anpassungen an anderen Teilen des Projekts vornehmen zu müssen.

JDepend benötigt als Eingabe kompilierte Systeme als class-Dateien oder jar-Archive. Die erhobenen Werte können durch eine textuelle Benutzerschnittstel-

le, grafische Benutzerschnittstelle oder XML-Schnittstelle (Abbildung 7) dargestellt werden. Der XML-Output kann dann weiterverarbeitet werden.

```
- <JDepend>
  - <Packages>
    - <Package name="epayment.adapters">
      - <Stats>
          <TotalClasses>2</TotalClasses>
          <ConcreteClasses>2</ConcreteClasses>
          <AbstractClasses>0</AbstractClasses>
          <Ca>0</Ca>
          <Ce>4</Ce>
          <A>0</A>
          <I>1</I>
          <D>0</D>
        </Stats>
        <AbstractClasses> </AbstractClasses>
      - <ConcreteClasses>
          <Class>ABCGatewayAdapter</Class>
          <Class>XYZGatewayAdapter</Class>
        </ConcreteClasses>
```

Abbildung 7 XML-Output von JDepend

Ziel des Softwaresystems ist die bessere Integration der erhobenen Maße für andere Tools zu bieten. Die grafische Swing-Schnittstelle ist auf den ersten Blick gewöhnungsbedürftig da die Informationen sehr kompakt dargestellt sind. Wie in Abbildung 8 dargestellt handelt sich um eine Baumstruktur, die beliebig aufgeklappt werden kann.

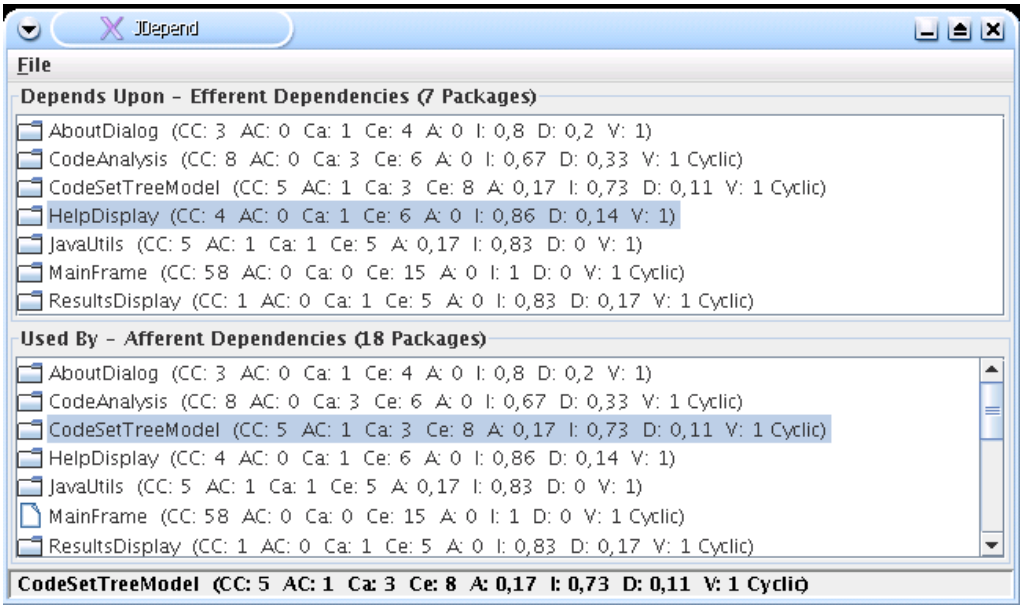


Abbildung 8 Grafische Schnittstelle von JDepend

Die Anzeige ist zweigeteilt in eine Darstellung für die Metrik Efferent Coupling und eine für Afferent Coupling. Hinter jeder Komponente stehen die erhobenen Metriken in Klammer. Die Abkürzungen stehen für:

CC - Concrete Class Count
AC - Abstract Class (and Interface) Count
Ca - Afferent Couplings (Ca)
Ce - Efferent Couplings (Ce)
A - Abstractness (0-1)
I - Instability (0-1)
D - Distance from the Main Sequence (0-1)
V - Volatility (0-1)
Cyclic – Falls das Paket ein Abhängigkeitszyklus beinhaltet

Zur Identifikation von Paketabhängigkeitszyklen ist die textuelle und XML-Schnittstelle vorgesehen. Das Prinzip der Zyklenidentifikation wird im folgenden anhand von zwei Beispielen, wie sie von der textuellen Schnittstelle ausgegeben werden gezeigt. Aus der ersten Ausgabe ist zu entnehmen, dass com.xyz.ejb von com.xyz.servlet abhängt. com.xyz.servlet wiederum ist abhängig von com.xyz.ejb. Durch die zweite Ausgabe zeigt sich, dass com.xyz.client von com.xyz.ejb abhängt, dieses wiederum von com.xyz.servlet, was von com.xyz.ejb abhängt. Somit besteht eine zyklische Abhängigkeitsbeziehung zwischen com.xyz.servlet und com.xyz.ejb.

```
com.xyz.ejb
  |
  | com.xyz.servlet
  |-com.xyz.ejb
com.xyz.client
  |
  |-com.xyz.ejb
  | com.xyz.servlet
  |-com.xyz.ejb
```

Auf der Website von JDepend wird beschrieben, dass JDepend dabei hilft, in einem iterativen Prozess das System so zu refaktorieren, das die Pakete mit geringer Abstraktion von Paketen mit hoher Abstraktionsstufe abhängen. Somit können die Pakete mit hoher Abstraktion unabhängig wiederverwendet werden und sind zusätzlich in hohem Maße erweiterbar. In einem guten Design werden Abhängigkeiten zwischen stabilen Paketen angestrebt. Abhängigkeiten können mit JDepend iterativ bestimmt und dahingegen verändert werden, so dass bessere Software entsteht. Stablile Pakete sollten das Zentrum von lose abhängigen Applikationen sein, so dass die Entwicklungszeit nicht maßgeblich von Software-Veränderungen beeinflusst wird. Stabile Pakete bilden Fassaden zu anderen Subsystemen, so dass Entwicklungsteams parallel in hoher Geschwindigkeit arbeiten können.

4.3.1 Verwendete Metriken

Zu den einfachen Metriken gehören die *Anzahl von Klassen und Schnittstellen*. JDepended unterscheidet dabei nicht zwischen abstrakten Klassen und Schnittstellen. Die Anzahl der konkreten und abstrakten Klassen (und Schnittstellen) in einem Paket geben Aufschluss über die Erweiterbarkeit des Pakets.

Afferent Couplings einer Klasse kann als Maß für die Paketzuständigkeit verstanden werden. Dagegen beschreibt Efferent Couplings die Anzahl von anderen Paketen, von denen Klassen, die sich innerhalb eines betrachteten Pakets befinden, abhängig sind. Die Anzahl dieser Pakete kann als Maß für die Paketunabhängigkeit gesehen werden.

Der Grad der Abstraktheit berechnet sich durch das Verhältnis der Anzahl von abstrakten Klassen (und Schnittstellen) in einem betrachteten Paket von der Gesamtanzahl von Klassen in dem zu analysierenden Paket. Dieser Wert bewegt sich zwischen 0 und 1, wobei eine Abstraktheit von 0 die Existenz eines vollkommen konkreten Pakets bedeutet und ein Wert von 1 ein vollständig abstraktes Paket bedeutet.

Tabelle 6 Erhobene Metriken von JDepend

Verwendete Metriken	
Anzahl Klassen/Interfaces	Abstraktheit
Afferent Coupling	Efferent Coupling
Instabilität	Distance from Main Sequence
Cylces betw. package dependencies	

Weitere Metriken sind Instabilität, Abstand von der Main Sequence und Zyklen zwischen Paketabhängigkeiten, die von JDepend angezeigt werden. Unter Abstand von der Main Sequence versteht man die Abweichung eines Pakets von der idealisierten Linie Abstraktheit + Instabilität = 1. Der Abstand ist ein Indikator für die Paketbalance zwischen Abstraktheit und Stabilität. Ein Paket, das sich exakt auf der main sequence befindet ist optimal ausbalanciert hinsichtlich Abstraktheit und Stabilität. Ideale Pakete sind weder vollständig abstrakt noch vollständig stablil (x = 0, y = 1) bzw. vollständig konkret und vollständig instabil (x = 1, y = 0). Der Ergebnisbereich für dieses Metrik liegt zwischen 0 und 1. D = 0 bedeutet, dass das betrachtete Paket deckungsgleich mit der main sequence ist, wohin gegen D = 1 zeigt, dass das Paket so weit wie möglich davon entfernt ist. Tabelle Tabelle 6 listet alle verwendeten Metriken auf.

4.3.2 Weitere Features

JDepend bietet die Möglichkeiten Third-Party-Paketabhängigkeiten zu identifizieren und zu isolieren. Sobald die Abhängigkeiten zu diesen Paketen bestimmt wurden, können diese Abhängigkeiten durch abstrakte und stabile Pakete kontrolliert werden, da diese die Implementierungsdetails dieser Pakete kapseln.

Pakete, die hohe Kohäsion aufweisen sowie im hohen Maße unabhängig von anderen Paketen sind, können als autonome Module mit ihren eigenen Veröffentlichungsplänen und Versionsnummern veröffentlicht werden. Ein weiteres Feature von JDepend ist die Identifikation von Paketabhängigkeitszyklen. Zyklen verschlechtern die Wiederverwendbarkeit. Dazu kann man die textuelle Benutzerschnittstelle verwenden, die im Detail zeigt, wie der Zyklus zwischen Paketen zustande kommt. Ziel dabei ist, Zyklen zu bestimmen und mit gängigen objektorientierten Softwaretechniken aufzulösen.

JDepend kann durch auch in JUnit-Testfällen verwendet werden. Somit ist eine Automatisierung der Qualitätsanalyse möglich, da nicht zwingend eine visuelle Anzeige und manuelle Auswertung notwendig ist. Mittels Testfällen kann geprüft werden, ob die erhobenen Maße im gewünschten Rahmen liegen. Falls sie außerhalb liegen ist der entsprechende Testfall nicht erfolgreich. JUnit kann auch zum Entdecken von Paketabhängigkeiten und Zyklen verwendet werden.

Die Verwendung von Junit hat den Vorteil, das die Analyse automatisiert ablaufen kann. Allerdings kann das speziell für das Identifizieren von Paketabhängigkeiten und -zyklen aufwändig und umständlich sein. Aus diesem Grund gibt es das Modul Dependencies FitNesse fixture, womit es möglich ist, Paketabhängigkeiten übersichtlich in tabellarischer Form aufzubereiten. Diese Erweiterung verwendet die JDepend-API. Anhand der Abbildung 9 wird das Prinzip kurz erläutert. Das zu untersuchende System hat in diesem Fall drei Module bzw. Pakete: ejb, web und util. Der Benutzer hat sich entschieden, dass Abhängigkeiten zwischen ejb und util sowie web und util erlaubt sind. Dieser Sachverhalt wird duch die beiden Kreuze beschrieben, so dass ejb und web von util abhängen.

Module Dependencies			
	ejb	web	util
ejb			X
web			X
util			

Abbildung 9 Abhängigkeiten der Pakete ejb, web und util

Wird diese Tabelle als fixture set ausgeführt, so zeigen die Farben rot und grün, ob das System die gewünschten Paketabhängigkeiten aufweist. In der Abbildung deutet die grüne Hintergrundfarbe der Zellen, die ein Kreuz beinhalten, daraufhin, dass im System tatsächlich solche Abhängigkeiten zwischen den Paketen bestehen. Dagegen zeigt die rote Hintergrundfarbe der Zelle ejb-web, dass auch zwischen diesen beiden Paketen eine Abhängigkeit besteht, diese aber nicht erwünscht ist. Somit weis der Entwickler, wo er anzusetzen hat.

4.3.3 Fazit

JDepend ist ein sehr brauchbares Messwerkzeug, um die Qualität eines Softwaresystems im Auge zu behalten. Vor allem die tabellarische Darstellung der Paketabhängigkeiten ist auch in der Praxis einsetzbar, um ungewollte Abhängigkeiten zu identifizieren und zu beseitigen. JDepend ist für das Auffinden von Paketabhängigkeiten und Zyklen entwickelt worden. Die JDepend-API bietet die Möglichkeit eigene Erweiterungen zu schreiben, die speziell an bestimmte Bedürfnisse angepasst sind.

Als negativ, könnte die Tatsache gesehen werden, dass es mehrere Anzeigemodi (grafisch, textuel, XML) gibt, um alle Informationen anzuzeigen. Es gibt also nicht eine einfache Schnittstelle, die alle Informationen bereitstellt. Zyklen

beispielsweise, werden zwar in der grafischen Schnittstelle benannt, die Struktur der zyklomatischen Abhängigkeit wird jedoch nur in der textuellen und XML- Schnittstelle aufgezeigt.

Die Szenarien, die in Sektion 1.2 ausführlicher beschrieben sind, wurden folgendermaßen bewertet:

- Z1 (Qualitätsreporting): Die berechneten Werte der Metriken können in XML exportiert werden.
- Z2 (Anomalieanalyse): Das Programm bietet kaum Unterstützung für das entdecken von Anomalien. Messwerte die die maximalen Höchstwerte für die einzelnen Metriken überschreiten werden ermittelt und farblich dargestellt.
- Z3 (QD-Entdeckung): Qualitätsdefekte werden nicht explizit erkannt. Es werden keine expliziten Vorschläge für mögliche Refaktorierungen gemacht.

4.4 CodeAnalyzer

CodeAnalyzer ist eine Swing-Applikation, mit der sehr einfache Metriken erhoben werden können. Bevor eine Analyse eines Projektes erfolgen kann, muss ein Extension Set angegeben werden. Hier hat man u. A. die Wahl zwischen Java, C, C++ oder auch Assembler. Des Weiteren besteht die Möglichkeit neue Extension Sets zu definieren, existierende zu editieren oder auch zu löschen. Extension Sets definieren die zulässigen Dateien an ihren Endungen, die für eine Analyse berücksichtigt werden sollen. Anschließend kann der Benutzer im Code Set Browser ein neuen Code Set anlegen und diesem einen Namen geben. Code Sets können auch geladen oder gespeichert werden. Ein Code Set fungiert als Wurzelknoten, dem Dateien und Verzeichnisse hinzugefügt werden können. Es besteht die Möglichkeit Dateien und Ordner von der Analyse durch Exclude oder Include auszuschließen bzw. wieder aufzunehmen. Nach dem Analysevorgang werden im Ergebnisfenster die Resultate der erhobenen Metriken angezeigt.

4.4.1 Erhobene Metriken

Wie bereits erwähnt handelt es sich um sehr einfache Quellcode-Metriken die in Tabelle 7 aufgelistet sind. Ein Projekt als Ganzes oder einzelne Ausschnitte daraus können auf durchschnittliche Zeilenlänge, Anzahl der Codezeilen, Anzahl der kommentierten Zeilen und Leerzeichen-Zeilen untersucht werden.

Tabelle 7 Verwendete Metriken von CodeAnalyzer

Verwendete Metriken	
Total Files	Total Lines
Avg Line Length	Code Lines
Comment Lines	Whitespace Lines
Code/(Comment+Whitespace) Ratio	Code/Comment Ratio
Code/Whitespace Ratio	Code/Total Lines Ratio
Code Lines per File	Whitespace Line per File

Außerdem zeigt das Programm einige Verhältnisse zu diesen Metriken an. Ein Beispiel wäre das Verhältnis von Quellcode zu Kommentaren. Wie in Abbildung 10 dargestellt wird für jeden ausgewählten Knoten im Code Set Browser eine Statistik und eine grafische Repräsentation im Results-View angezeigt.

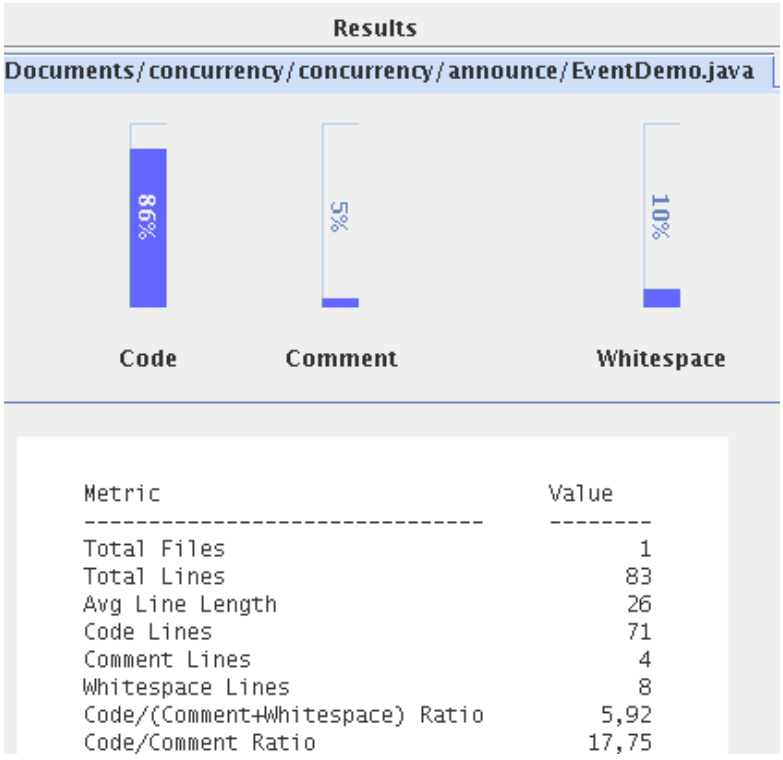
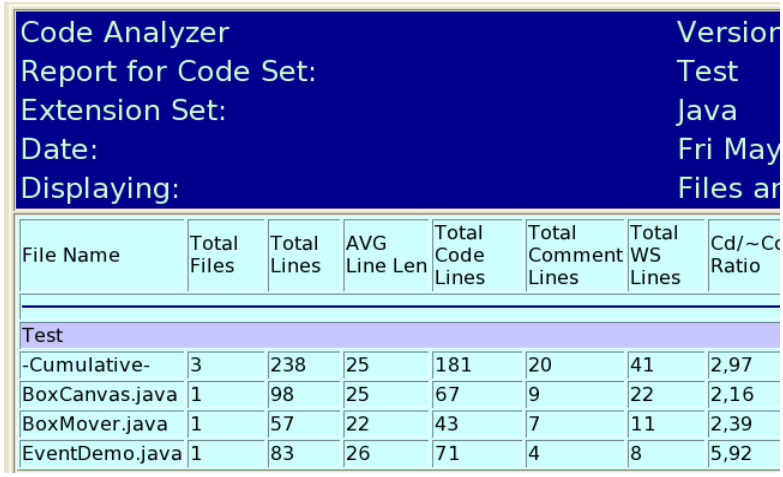


Abbildung 10 Ergebnissicht von CodeAnalyzer

4.4.2 Weitere Features

Extension Sets und Code Sets können abgespeichert werden, um die Einstellungen wieder zu verwenden. Die erhobenen Maße können des Weiteren nach

CSV oder HTML für eine Archivierung der Metrikergebnisse exportiert werden. Abbildung 11 zeigt den Aufbau eines solchen HTML-Reports.



Code Analyzer — Version
Report for Code Set: Test
Extension Set: Java
Date: Fri May
Displaying: Files ar

File Name	Total Files	Total Lines	AVG Line Len	Total Code Lines	Total Comment Lines	Total WS Lines	Cd/~Cc Ratio
Test							
-Cumulative-	3	238	25	181	20	41	2,97
BoxCanvas.java	1	98	25	67	9	22	2,16
BoxMover.java	1	57	22	43	7	11	2,39
EventDemo.java	1	83	26	71	4	8	5,92

Abbildung 11 HTML-Report in CodeAnalyzer

4.4.3 Fazit

Dieses Tool eignet sich nur bedingt für den produktiven Einsatz. Entwickler bekommen einen Überblick davon, wie das Verhältnis zwischen Quellcode und Kommentar aussieht. Für das Werkzeug spricht seine sehr einfache Bedienung. Es handelt sich um ein einziges jar-File, welches ohne vorherige Einstellungen ausgeführt werden kann.

Die Szenarien, die in Sektion 1.2 ausführlicher beschrieben sind, wurden folgendermaßen bewertet:

- Z1 (Qualitätsreporting): Die berechneten Werte der Metriken können in HTML und CSV exportiert werden und somit in den großen Analysewerkzeugen (z.B. Excel oder Statistika) verwendet werden.
- Z2 (Anomalieanalyse): Das Programm bietet keine Unterstützung für das entdecken von Anomalien. Abweichungen der Metriken vom Durchschnittswert müssen manuell ermittelt werden.
- Z3 (QD-Entdeckung): Qualitätsdefekte werden nicht explizit erkannt. Es werden keine expliziten Vorschläge für mögliche Refaktorierungen gemacht.

4.5 CCCC

CCCC ist ein Tool zur Analyse von Quell-Code, das von http://sourceforge.net/projects/cccc bezogen werden kann. Ursprünglich für C

und C++ geschrieben, unterstützt es auch andere Hochsprachen. Wie in Abbildung 12 dargestellt handelt sich bei CCCC um ein Konsolenprogramm, dem Sourcecode-Dateien zu übergeben sind. Führt man das Programm in der Kommandozeile aus, so erzeugt in einem ersten Schritt der verwendete Parser Einträge in eine interne Datenbank.

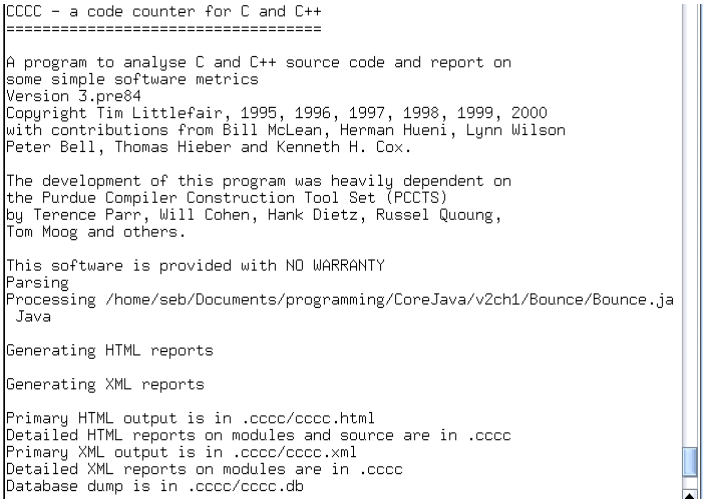


Abbildung 12 Ausführung von CCCC auf der Konsole

In einem zweiten Schritt wird aus diesen Werten der HTML-Report generiert. Aus Anwendersicht, erzeugt dass Programm im aufgerufenen Verzeichnis einen Ordner .cccc, der die Metrikreporte in Form von HTML- und XML-Dateien enthält. Die Datei cccc.html ist standardmäßig die Übersichtsseite, von der aus alle anderen HTML-Dateien erreicht werden können. Zusätzlich wird ein Dump der internen Datenbank angelegt (cccc.db), aus der die HTML- und XML-Zusammenfassungen entstanden sind. Der Kommandozeilenbefehl akzeptiert einige Optionen, mit denen die Standardwerte wie z. B. Dateinamen des HTML-Reports oder Speicherort angepasst werden können. Anhand der Dateiendung wird ein entsprechender Parser ausgewählt, vorausgesetzt die Sprache wird von CCCC unterstützt. Neben C, C++ und Java gibt es auch Unterstützung für Ada95. Der Autor stellt in seinem User-Guide ganz klar heraus, dass es sich hierbei um Freeware handelt. Da CCCC als Quellcode vorliegt, ist es erwünscht, dass Entwickler das Programm an ihre Ansprüche anpassen und für die Qualitätsanalyse von Softwaresystemen verwenden.

4.5.1 Erhobene Metriken

Wie in Abbildung 13 gezeigt werden die Messwerte in Tabellenform dargestellt. Besondere Einträge werden dabei hervorgehoben, um darauf hinzuweisen, dass die erlaubten Werte für diese Kategorie überschritten wurden. Es existieren für jede Metrik zwei Grenzwerte. Überschreitet ein Maß den ersten

Grenzwert, bleibt aber unter dem zweiten, so wird das durch gelben Hintergrund oder kursiver Schrift kenntlich gemacht. Problematisch wird es, wenn das Maß rot hinterlegt ist oder durch fette Schrift hervorgehoben wird. In diesem Fall wurde der zweite Grenzwert auch überschritten. Der Programmierer sollte versuchen, durch Anpassung des Softwaresystems den Wert dieses Maß zu reduzieren. Die Grenzen für die einzelnen Maße können vom Benutzer in Konfigurationsdateien festgelegt werden und durch Verwendung der richtigen Option als Argument cccc übergeben werden. Somit kann ein Entwickler Höchstwerte festlegen, die auf keinen Fall überschritten werden dürfen.

Number of modules	NOM	15	
Lines of Code	LOC	105	7.000
McCabe's Cyclomatic Number	MVG	6	0.400
Lines of Comment	COM	76	5.067
LOC/COM	L_C	1.382	
MVG/COM	M_C	0.079	
Information Flow measure (inclusive)	IF4	20	1.333
Information Flow measure (visible)	IF4v	20	1.333
Information Flow measure (concrete)	IF4c	0	0.000
Lines of Code rejected by parser	REJ	5	

Abbildung 13 Ausschnitt aus dem HTML-Report von CCCC

Ein Metrikreport setzt sich durch mehrere HTML-Tabellen zusammen. Es existieren Tabellen für Metriken auf Projektebene sowie für die einzelnen Module. In Java stellen Module Klassen und Interfaces dar.

Es werden hauptsächlich einfache Maße erhoben die in Tabelle 8 aufgelistet sind. In die Kategorie der einfachen Metriken fallen LOC, Anzahl der Kommentar-Zeilen, Anzahl der Module im Projekt oder auch Anzahl gewichteter Methoden pro Klasse.

Tabelle 8 Verwendete Metriken von CCCC

Verwendete Metriken	
LOC	McCabe's Cyclomatic Compl. (MVG)
Comment Lines (COM)	LOC / COM
MVG / COM	Fan-out, Fan-in
Henry-Kafura/Shepperd measure	Number of Modules
Weighted methods per class	Rejected lines

Komplexere Metriken sind McCabe's Cyclomatic Complexity, eine Art Abhängigkeitsmaß (Fan-out, Fan-in) und Henry-Kafura/Shepperd Maß zur Analyse des Informationsflusses. Die genaue Bedeutung dieser Maße kann dem User-Guide entnommen werden.

4.5.2 Fazit

Es handelt sich um ein einfach zu bedienendes Konsolen-Werkzeug zur Ermittlung von Metriken. Nachteilig ist, dass fast nur einfache Maße erhoben werden. Laut Autor ist die Korrektheit der Maße nicht sichergestellt, außerdem kann es bei der Analyse zu Fehlern und Abstürzen kommen. Die eigentliche Intension dieses Tools ist es jedoch, es als Basis zu verwenden und beliebig weiterzuentwickeln. Das Programm kann somit beliebig an spezielle Anforderungen angepasst werden, indem neue, komplexere Metriken erhoben werden und der HTML-Report an besondere Ansprüche erweitert wird. Außerdem sind weitere Ausgabeformate denkbar. Ebenso ist es denkbar, die XML-Ausgabe des Reports weiterzuverarbeiten. Das Programm kann sehr gut für automatisierte Metrikerhebungen verwendet werden.

Die Szenarien, die in Sektion 1.2 ausführlicher beschrieben sind, wurden folgendermaßen bewertet:

- Z1 (Qualitätsreporting): Die berechneten Werte der Metriken können in HTML und XML exportiert werden und mittels eigens herzustellender XSL-Transformatoren in andere Formate umgewandelt werden.
- Z2 (Anomalieanalyse): Die Identifikation von Anomalien wird gut unterstützt. Es können zwei Schwellwerte festgelegt werden um Anomalien zu entdecken. Relative Anomalien bzgl. eines berechneten Wertes wie dem arithmetischen Mittel oder des Median sind nicht möglich.
- Z3 (QD-Entdeckung): Qualitätsdefekte werden nicht explizit erkannt und es werden keine Angabe von Verbesserungsvorschlägen (Refaktorierungen) gemacht.

4.6 Condenser

Condenser ist ein kleines Konsolenwerkzeug, welches Code-Duplikate findet. In der aktuellen Version wird der Anwender nur darauf hingewiesen, zwischen welchen Klassen sich an welchen Stellen Methodenduplikate befinden. In zukünftigen Versionen soll das Programm nicht nur Duplikate finden, sondern auch beseitigen. Um Condenser benutzen zu können, muss Jython auf dem System installiert sein. Jython ist eine vollständige Java-Implementierung von Python, was es ermöglicht, Python auf jeder Java-Plattform laufen zu lassen. Nachdem das Tool heruntergeladen und Jython installiert wurde, müssen noch ein paar Änderungen am CLASSPATH vorgenommen werden. Anschließend kann man ein Verzeichnis, welches Source-Code beinhaltet, auf Duplikate untersuchen. Die Ausgabe liefert die Signaturen der Methoden, die mehrfach vorkommen, und die dazugehörigen Klassen. Abbildung 14 zeigt das Ergebnis einer Analyse, bei der die Methode one() sowohl in der Klasse Blobby als auch in Dummy vorkommt.

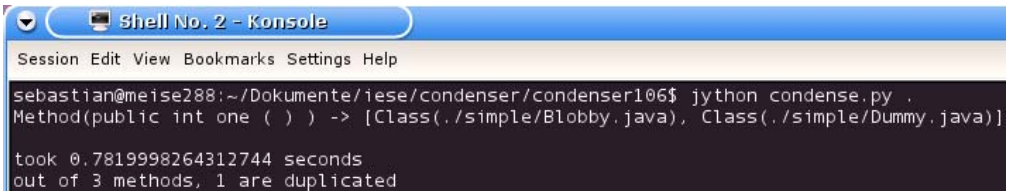


Abbildung 14 Ausgabe des Konsolenprogramms Condenser

4.6.1 Fazit

Ein paar weitere Features wären wünschenswert, um Condenser produktiv einsetzen zu können. Es werden ausschließlich Methoden-Duplikate erkannt. Dagegen wäre es sinnvoll, auch auf gleiche Code-Passagen (z. B. zwei identische Methoden, die sich nur durch ihren Namen unterscheiden) zu prüfen. Noch einen Schritt weiter in den produktiven Einsatzbereich wäre, wenn das Tool ähnlichen Code erkennen würde. Solch eine Situation kommt häufiger vor, dann nämlich, wenn ein Entwickler einfach eine bereits implementierte Methode kopiert, wenige Änderungen vornimmt und z.B. nur die Variablennamen ändert. Im Grunde liegt auch hier Code-Duplizierung vor, was dringend vermieden werden sollte. Ein weiterer Schritt in die richtige Richtung wäre die Integration von Condenser in eine IDE wie beispielsweise Eclipse. Das Markieren von Duplikaten direkt im Quellcode würde Code-Duplizierung frühzeitig eliminieren.

Die Szenarien, die in Sektion 1.2 ausführlicher beschrieben sind, wurden folgendermaßen bewertet:

- Z1 (Qualitätsreporting): Die entdecken Duplikate werden nur auf der Konsole ausgegeben.

- Z2 (Anomalieanalyse): Die Identifikation von Anomalien wird nicht unterstützt. Bei den entdeckten Duplikaten gibt es keine prozentuale Wahrscheinlichkeit mittels der man auf Anomalien schließen könnte.
- Z3 (QD-Entdeckung): Außer Code Duplikaten werden keine Qualitätsdefekte erkannt und es werden keine Angabe von Verbesserungsvorschlägen (Refaktorierungen) gemacht.

4.7 FindBugs

FindBugs ist ein statisches Analysewerkzeug für Java-Bytecode, um Bugs frühzeitig im Entwicklungsprozess ausfindig zu machen. Die Idee hinter diesem Programm basiert auf so genannten bug patterns. Bug patterns sind häufig verwendete Code-Ausdrücke, die aufgrund einer Vielzahl von Aspekten fälschlicherweise Anwendung finden. In vielen Projekten entstehen beispielsweise bug patterns, indem die Entwickler die Verwendung einer API nicht richtig verstanden haben.

FindBugs wird unter der GNU Public License vertrieben und ist somit frei zugänglich. Die Entwickler sprechen von einer beta quality software, die in Zukunft noch weiter entwickelt werden soll. FindBugs benötigt für die Ausführung mindestens Java 1.4. Die Analyse von Java-Projekten ist nicht an eine bestimmte Version gebunden.

In der aktuellen Version existieren vier Arten von Bug-Kategorien. Bestimmte bug patterns untersuchen das Projekt auf Korrektheit. Beispiele für diese Art von Bugs sind z. B. „Call to equals() with null argument“ oder „Finalizer does not call superclass finalizer“. Des Weiteren existieren Bugs, welche Design-Fehler darstellen, da die Möglichkeit eröffnet wird, durch böswillige Aktionen die Stabilität und Sicherheit des Projekts zu gefährden. Ein Beispiel für diese Kategorie wäre u. a. „Field isn't final and can't be protected from malicious code“. Weiterhin existieren Bugs, die die Performance des Gesamtsystems negativ beeinflussen können (z. B. „Private method is never called“ oder „Method concatenates strings using + in a loop“). Projekte werden außerdem auf Multithreading-Aspekte untersucht. „Unsynchronized get method, synchronized set method“ ist ein Bug, der repräsentativ für diese Art von Fehlern steht. Eine vollständige Beschreibung aller Bugs ist der Dokumentation von FindBugs zu entnehmen, die auch online unter
http://findbugs.sourceforge.net/bugDescriptions.html verfügbar ist.

4.7.1 Eigenständige Anwendung

FindBugs kann zur Analyse von jar-, zip-, ear- und war-Archiven verwendet werden. Mit Hilfe der grafischen Benutzeroberfläche werden sehr leicht Java-Archieve für die Analyse ausgewählt (Abbildung 15).

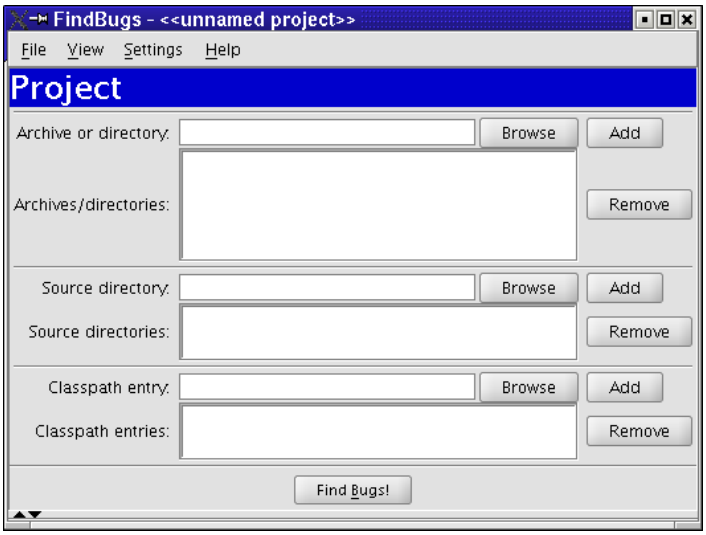


Abbildung 15 Auswahl von Archiven für die Analyse

Wie der Abbildung zu entnehmen ist, können auch Verzeichnisse und einzelne Source-Dateien angegeben werden.

Nach der Auswahl der zur untersuchenden Quellen und dem Anstoßen des Messvorgangs präsentiert das Werkzeug die Ergebnisse in einer Sicht, die durch mehrere Tabs unterteilt wird. Abbildung 16 zeigt wie das Result-Browsing vonstatten geht. Ein Bug-Tree zeigt sehr anschaulich potentielle Fehler. Wählt der Benutzer einen Bug im Baum aus, so bekommt dieser eine detaillierte Beschreibung sowie Verbesserungsvorschläge.

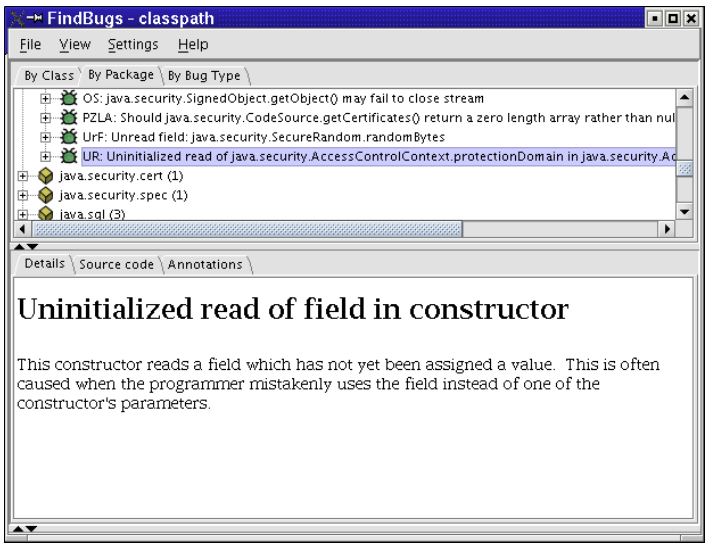


Abbildung 16 Übersichtliche Anzeige der Ergebnisse

FindBugs bietet die Möglichkeiten Bug-Ergebnisse zu speichern und zu laden. Als Speicherungsformat wird XML verwendet. Es werden auch die benutzerdefinierten Anmerkungen mitgespeichert.

4.7.2 Eclipse-Plugin

FindBugs liegt ebenfalls als Eclipse-Plugin vor. Die dazu notwendigen Dateien müssen einfach in das Eclipse-Plugin-Verzeichnis extrahiert werden. Nachdem Start der Eclipse-Workbench kann die Sicht Bug Details gestartet werden. Um eine Bug-Analyse zu starten, muss die FindBugs-Analyse für ein Java-Projekt in den Projekt-Einstellungen über die Kategorie Findbugs aktiviert werden (Abbildung 17).

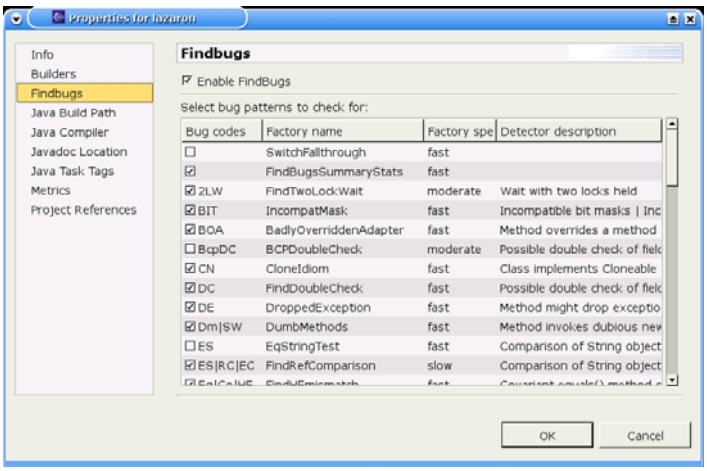


Abbildung 17 Konfiguration von FindBugs in Eclipse

Anschließend wird ein Analysevorgang im Package Explorer über das Kontext-Menü des jeweiligen Projekts (Rechtsklick auf das Projekt) durch den Menü-Punkt Find Bugs angestoßen (Abbildung 18).

Abbildung 18 FindBugs als Eclipse-Plugin

4.7.3 Weitere Features

Weiterhin kann FindBugs in ein Build-Skript für Ant integriert werden. Somit ist es möglich, dass ein Ant-Skript automatisch während des Deployments Find-Bugs ausführt. Dazu muss die build.xml angepasst und task definitions definiert werden.

Der Filter files wird dazu verwendet anzugeben welche Bug reports für Klassen oder Methoden angezeigt werden sollen. In der aktuellen Version 0.8.4 sind Filter nur über das Command Line Interface möglich, allerdings nicht über die grafische Benutzeroberfläche. Mit den Argumenten -exclude excludefilter.xml und -include includefilter.xml muss bei Verwendung von Filtern FindBugs gestartet werden. In einer XML-Datei legt der Benutzer fest, welche Arten von Bug reports für eine Klasse akzeptiert werden sollen. Dies geschieht mit Hilfe von match-Klauseln. Es existieren drei Arten von match-Klauseln: BugCode, Method und Or. BugCode spezifiziert die verwendeten Bug-Kategorien. Es handelt sich hierbei um eine Liste von Bug-Abkürzungen, wobei die einzelnen Einträge durch Kommata abgetrennt werden. Die Or-Klausel wird dazu verwendet, um mehrere Match-Klauseln miteinander zu kombinieren.

4.7.4 Fazit

Das Messwerkzeug FindBugs ist ein ein Eclipse-Plugin welches problemlos in den Softwareentwicklungsprozess eingegliedert werden kann um sogenannte bug patterns zu suchen. So können Codefragmente aufgepürt werden, die häufig Verwendung finden, allerdings nicht empfehlenswert sind und sich negativ auf die Performance oder Sicherheit des Systems auswirken. Nach welchen bug patterns gesucht werden soll, kann in den Einstellungen konfiguriert werden. Durch Export der Bug reports kann die Verbesserung der Codequalität nachgewiesen werden. Es ist durchaus sinnvoll FindBugs in der Implementierungsphase zu verwenden, vor allem wenn als IDE Eclipse verwendet wird.

Die Szenarien, die in Sektion 1.2 ausführlicher beschrieben sind, wurden folgendermaßen bewertet:

- Z1 (Qualitätsreporting): Die berechneten Werte der Metriken können in XML gespeichert werden und mittels eigens herzustellender XSL-Transformatoren in andere Formate umgewandelt werden.
- Z2 (Anomalieanalyse): Die Identifikation von Anomalien wird nicht unterstützt da Bug Pattern entweder vorhanden sind oder nicht.
- Z3 (QD-Entdeckung): Qualitätsdefekte in Form von Bug Patterns werden explizit erkannt und es werden Rationale beschrieben warum diese vorkommen können. Eine konkrete Angabe von Verbesserungsvorschlägen (Refaktorierungen) wird nicht gemacht.

4.8 Java Coding Standard Checker (JCSC)

JCSC ermöglicht Java-Quelldateien gegen selbstdefinierbare Coding-Standards zu validieren. JCSC kann in der Praxis zur Verbesserung der Lesbarkeit eines Projekts eingesetzt werden. Eine Analyse der Sourcen auf Einhaltung von Namenskonventionen ist ebenso möglich, wie das Prüfen auf Konformität eines vordefinierten Coding-Stils (z. B. an welchen Stellen im Code sind Leerzeichen erlaubt). Das Tool erlaubt auch das Prüfen von Struktur des Quellcodes. Somit ist es beispielsweise möglich, zu prüfen, ob die Instanzvariablen zu Beginn der Klassendefinition definiert werden. Außerdem können Ordnungen, beispielsweise nach Sichtbarkeiten, definiert werden. Der Benutzer hat die Möglichkeit den Umfang von Javadoc-Kommentaren zu definieren und genau zu bestimmen, an welchen Stellen im Code Kommentare notwendig sind. Durch die angesprochenen Features erlaubt das Tool die Gewährleistung von Einheitlichkeit bezüglich eines Softwareprojekts, so dass auch die Lesbarkeit und Verfolgbarkeit des Codes merklich zunimmt. Eine weitere Aufgabe sieht das Tool in dem Erkennen von bad smells. Solche Coding-Schwächen können ein erstes Indiz für potentielle Bugs darstellen. Somit identifiziert JCSC schlechte Code-Passagen wie leere Catch-Blöcke.

4.8.1 Komfortabler Rules-Editor

Neben dem eigentlichen Programm existiert noch eine weitere Anwendung zur Verwaltung von Regeln. Der Regel-Editor erlaubt aufgrund einer Swing-Oberfläche eine einfache Verwaltung von Regeln (siehe Abbildung 19). Regelmengen werden zwar durch XML-Dateien beschrieben, allerdings ermöglicht die GUI das Editieren von Regeln ohne XML-Kenntnisse.

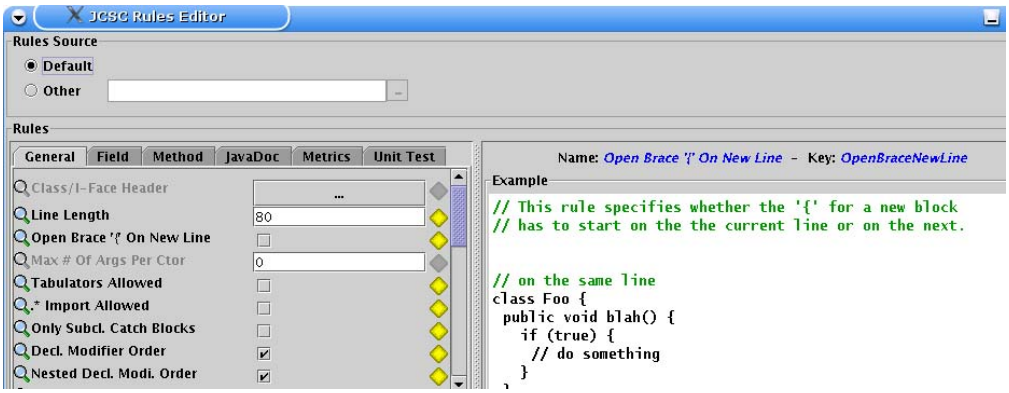


Abbildung 19 Regel-Editor von JCSC

In Abbildung 20 wird ein Ausschnitt aus der mitgelieferten XML datei mit verschiedenen Regeln gezeigt.

```
<?xml version="1.0" encoding="ISO-8859-1"?>.
<?xml-stylesheet href="xsl/rules.xsl" type="text/xsl"?>.
<jcsc version="__0.96__" date="2004-01-21 13:11:33">.
   <rules>.
      <rule name="CtorJavaDocPackage" category="JavaDoc" enabled="true">.
         <type>boolean</type>.
         <value>false</value>.
         <severity>4</severity>.
      </rule>.
      <rule name="MaxNumberOfNCSSPerConstructor" category="Metrics" enabled="true">.
         <type>integer</type>.
         <value>25</value>.
         <severity>4</severity>.
      </rule>.
      <rule name="FieldJavaDocPackage" category="JavaDoc" enabled="true">.
```

Abbildung 20 Regeln werden durch XML definiert

Die Regeln teilen sich in mehrere Kategorien auf und können durch die entsprechenden Tabs editiert werden: General, Field, Method, JavaDoc, Metrics und UnitTest. Zu allen verfügbaren Regeln existieren Erklärungen sowie Java-Beispiele. Für jede Regeln definiert der Benutzer zusätzlich eine Dringlichkeitsstufe (severity), die von Level 1 (notice) bis Level 5 (severe) reicht. Beispielsweise ist die Regel „Allow Public Field“ in der Kategorie Field standardmäßig auf Level 5 gesetzt, was nicht sonderbar verwunderlich ist, da Information Hiding zu den wichtigsten Konzepten der objektorientieren Programmierung gehört.

Ein Benutzer hat die Möglichkeit, die Standardregelmenge als Ausgangsbasis zu nutzen, Änderungen entsprechend der Projektanforderungen vorzunehmen, und diese neue Regelmenge abzuspeichern. Selbstdefinierte Regelmengen können jederzeit in den Editor geladen werden.

4.8.2 Anzeige der Ergebnisse

JCSC erlaubt ein ganzes Projekt zu überprüfen und die resultierende Datei in einem XML-fähigen Browser wie in Abbildung 21 dargestellt anzuzeigen.

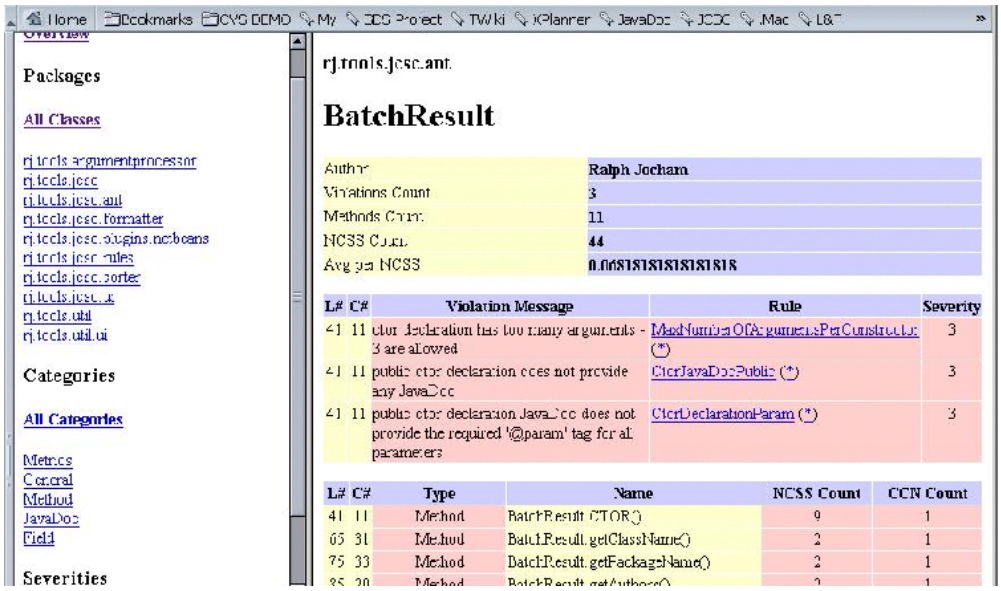


Abbildung 21 Anzeige der Ergebnisse nach einer Analyse im Browser

Vorraussetzung dafür ist Ant 1.5, was während des Build-Vorgangs zusätzlich das Projekt nach der angegebenen Regelmenge überprüft. Aufgrund von Ant ist es möglich, das IDEs Java-Sourcen parsen und die Ergebnisse anzeigen. Wie in Abbildung 22 zu sehen existiert bereits eine frühe Version eines Plugins für die kommerzielle Entwicklungsumgebung IntelliJ IDEA.

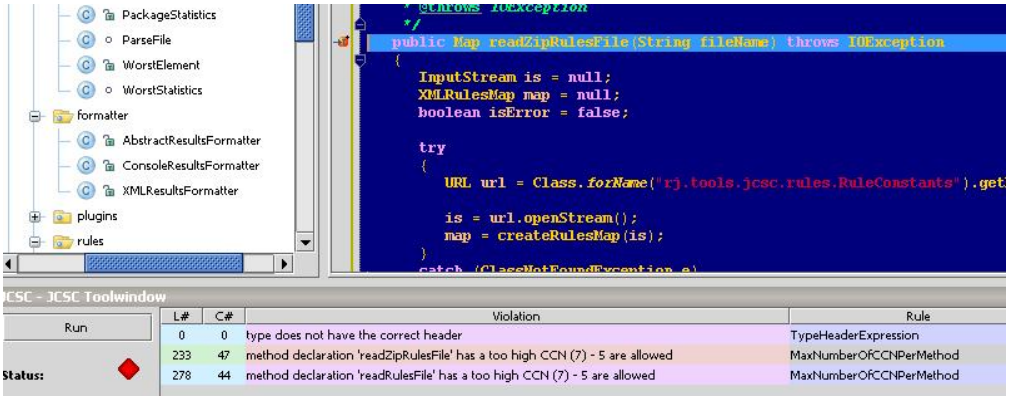


Abbildung 22 JCSC ist IDE-Pluginfähig (hier als Beispiel IntelliJ)

4.8.3 Fazit

Die Verwendung von JCSC in der Implementierungsphase ist zu empfehlen, da der Einhaltung von Code-Konventionen ein hoher Bedeutungsgrad zugerechnet werden sollte. Es erscheint vielleicht auf den ersten Blick fragwürdig, warum die Einhaltung von Konventionen enorm wichtig sein sollte. Eine projektweite, einheitliche Menge von Konventionen erhöht die Lesbarkeit des Codes ungemein. Die verschiedenen Entwickler können sich schneller in Code-Fragmente von Kollegen zurecht finden. Dieser Punkt ist nicht zu unterschätzen, da bei keiner Festlegung von Konventionen, jeder Entwickler seinem eigenen Stil nachgeht. Folglich unterscheiden sich die Softwarekomponentenkomponenten Entwickler zu Entwickler erheblich.

Die Szenarien, die in Sektion 1.2 ausführlicher beschrieben sind, wurden folgendermaßen bewertet:

- Z1 (Qualitätsreporting): Die berechneten Werte der Metriken können in XML gespeichert werden und mittels eigens herzustellender XSL-Transformatoren in andere Formate umgewandelt werden.
- Z2 (Anomalieanalyse): Die Identifikation von Anomalien wird nicht unterstützt da die Regeln entweder zutreffend sind oder nicht.
- Z3 (QD-Entdeckung): Qualitätsdefekte werden höchstens in Form von einigen Regeln explizit erkannt aber konkrete Angabe von Verbesserungsvorschlägen (Refaktorierungen) wird nicht gemacht.

4.9 IBM Structural Analysis for Java (SA4J)

SA4J hat das Ziel, Architekturmängel von Softwaresystemen zu identifizieren. Das Tool befindet sich zwar noch in einem Preview-Status, ist aber schon pro-

duktiv einsetzbar. Laut Hersteller soll es in der Endversion für große Softwaresysteme geeignet sein. IBM empfiehlt dieses Tool während des gesamten Implementierungsprozesses fortwährend zu verwenden, um schon in einer frühen Phase architektonische Schwächen zu finden. Je später Architekturfehler im Entwicklungszyklus entdeckt werden, desto kostspieliger ist es, diese zu beheben. Teilweise ist es dann sogar zu spät, diesen Fehler zu beheben.

SA4J orientiert sich am ISO 9126 Qualitätsmodell und hat das Ziel automatisch Qualitätseigenschaften wie z. B. Wartbarkeit, Stabilität oder auch Erweiterbarkeit zu messen. Im Gegensatz zu den zuvor beschriebenen Werkzeugen, werden Metriken auf hoher Abstraktionsebene erhoben. SA4J analysiert die Beziehungen zwischen den einzelnen Software-Komponenten.

4.9.1 Skeleton

Hauptaugenmerkt legt das Tool auf die Stabilität eines Softwaresystems, das durch ein so genanntes "Skeleton" dargestellt wird. Dabei handelt es sich um die Visualisierung von Abhängigkeitsbeziehungen. Abbildung 23 zeigt die Beziehungen in einem kleinen Softwaresystem. Jedes Viereck stellt entweder eine Klasse, ein Interface oder eine Gruppe dar. Gruppen beinhalten mehrere Objekte, werden allerdings durch ein Kästchen repräsentiert, da bei eine Änderungen alle Objekte dieser Gruppe betroffen sind.

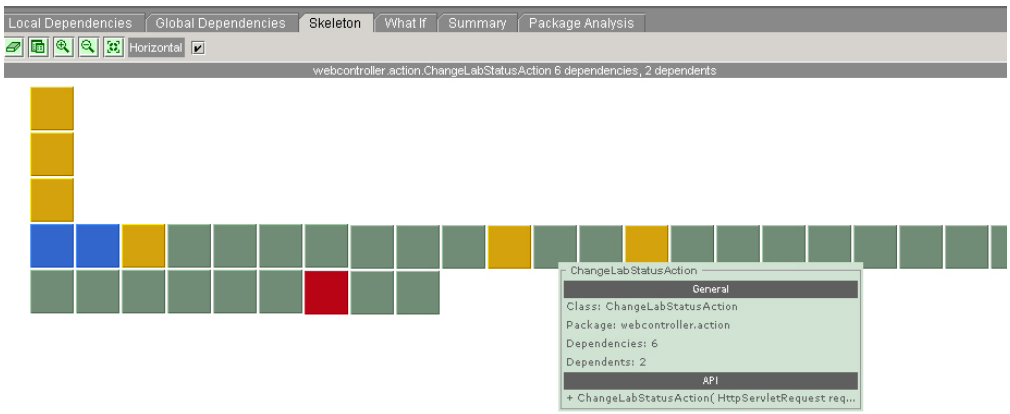


Abbildung 23 Skeleton View in SA4J

Die verschiedenen Abhängigkeitsbeziehungen, die in SA4J unterschieden werden, sind in Abbildung 24 aufgeführt. Die Repräsentation des Softwaresystems erfolgt in verschiedenen Ebenen. Objekte auf der untersten Ebene sind von keinen anderen Objekten im Softwaresystem abhängig. Objekte auf den darauf folgenden Ebenen stehen mit Objekten der darunter liegenden Ebenen in Beziehung. Bewegt der Anwender den Mauszeiger über ein Objekt, so erscheint eine Info-Box, die generelle Informationen über das betrachtete Objekt sowie

dessen API zeigt. Falls das Objekt in Abhängigkeitsbeziehungen verwickelt ist, wird dies in dem Feld Application Analyser dargestellt. Die Architektur des Softwaresystems aus Abbildung 23 ist nicht besonders gut, da die zweite Ebene wesentlich mehr Objekte aufweist, als die erste Ebene.

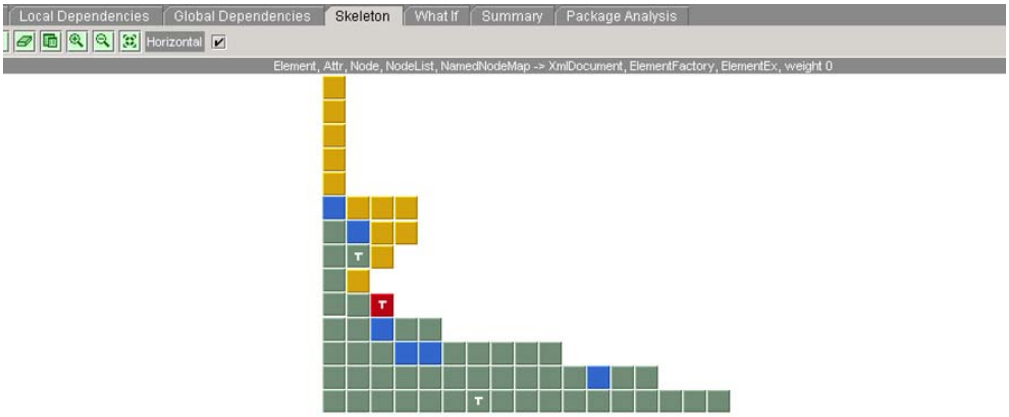


Abbildung 24 Skeleton Visualisierung in SA4J

Stabile Systeme zeigen in der Visualisierung eine pyramidenartige Form auf. Allgemein ist der Grad der Stabilität eines Systems in SA4J sehr schnell ersichtlich, da die Struktur eines gesunden Systems ein "physikalisch stabiles" Gebilde darstellt. Der Aufbau in Abbildung 24 ist zwar nicht optimal, stellt allerdings eine Verbesserung im Vergleich zu dem ersten Softwaresystem (dieses ist nicht stabil und "fällt in sich zusammen") dar. Ein Software-Architekt könnte jetzt versuchen, die Objekte in den Ebenen sieben bis neun zu reduzieren, so dass jede Ebene nicht mehr Objekte als die Ebene darunter besitzt. Ein weiteres Problem stellen die drei mit T gekennzeichneten Tangles (Abhängige Cluster) dar.

In der Skeleton-Analysesicht sind auch die Auswirkungen einer Änderung eines Objekts sichtbar. Wenn der Anwender auf ein Objekt klickt, so werden alle anderen Objekte farblich hervorgehoben, die bei einer Änderung betroffen wären. In Abbildung 24 hat der Anwender auf das rote Kästchen (mittleres T) geklickt. Die gelb markierten Objekte (oberhalb des angeklickten) sind direkt von einer Änderung des roten Objekts betroffen.

4.9.2 Explorer

Der Explorer beschreibt mit Hilfe eines Netzwerks, wie einzelnen Komponenten (Klassen, Schnittstellen, Pakete) miteinander in Beziehung stehen. Abbildung 5 zeigt die Abhängigkeitsbeziehungen, in die das mittlere Paket ejb verwickelt ist.

Abbildung 25 Zugriffsgraph in SA4J

In der Explorer-Sicht kann dabei, ausgehend von einem zentralen Objekt, analysiert werden, von welchen anderen Komponenten es abhängt bzw. welche Komponenten von dem zu untersuchenden Objekt abhängen. Der Benutzer kann ganz einfach per Klick auf ein Objekt durch das Softwaresystem navigieren. Dabei wird das ausgewählte Objekt zentriert, so dass der Anwender von diesem Objekt ausgehend die Abhängigkeitsbeziehungen studieren kann. Es stehen dem Benutzer einige komfortable Funktionen zur Verfügung, um auch in großen Softwaresystemen den Überblick zu wahren. In der Sidebar können bestimmte Objekttypen (Klassen, Schnittstellen, Pakete) und Abhängigkeitsbeziehungen (extends, implements, throws, etc) abgewählt werden. Zusätzlich kann die Zoom-Funktion verwendet werden, um Teilbereiche von sehr großen Netzen auf dem Bildschirm darstellen zu können. Außerdem können Teile einer großen Netzstruktur ausgewählt werden, um sie auf dem Bildschirm lesbar darzustellen. Der Benutzer markiert dazu den gewünschten Bereich per Maustaste. Wenn nun die Maustaste losgelassen wird, wird der selektierte Bereich auf dem Bildschirm dargestellt. Es stehen einige Abhängigkeitsfilter zur Verfügung, mit denen wichtige Fragen ausgehend von einem zentralen Objekt beantwortet werden können:

- Was sind die abhängigen Objekte?
- Von welchen Objekten hängt das zu untersuchende Objekt ab?
- Welche Objekte verwenden das zu untersuchende Objekt?
- Welche Objekte werden verwendet?
- Welche Objekte stehen mit dem zu untersuchenden Objekt in Beziehung?

Wie auch in der Skeleton-Sicht erscheint ein Tooltip mit beschreibenden Informationen (generelle Informationen, gegebenenfalls Informationen zur API, Auflistung der Anti-Patterns falls vorhanden), sobald der Anwender den Mauszeiger über ein Objekt bewegt. Der Benutzer kann einen Snapshots von dem Explorer-Fenstern machen und über previous und next auch zu einem späteren Zeitpunkt vergangene Snapshots erneut anzeigen lassen. Snapshots können auch als Jpeg gespeichert werden.

4.9.3 Lokale und globale Abhängigkeitsbeziehungen

Die verschiedenen Abhängigkeitsbeziehungen können auch tabellarisch für alle Klassen und Schnittstellen angezeigt werden. In der Sicht "Local Dependencies" wird für jede Softwareeinheit (Klasse oder Schnittstelle) der Grad der lokalen Abhängigkeit nach der Art der Beziehung aufgelistet. Die Spalten der Tabelle beziehen sich auf den Typ der Beziehung. Das Tool hebt besondere Abhängigkeitsbeziehungen durch rote Schrift hervor, so dass kritische bzw. markante Bereiche schnell ersichtlich sind und genauer untersucht werden können.

Die Sicht für "Global Dependencies" ist ähnlich aufgebaut. Für jedes Objekt des Projekts wird die Anzahl von Objekten gezählt, zu der eine Abhängigkeit testet (dependencies). Zusätzlich wird für jedes Objekt die Anzahl der davon abhängigen Objekten (dependents) aufgeführt. Die Spalte "Affected" gibt für ein bestimmtes Objekt die Gesamtzahl der notwendigen Änderungen an, wenn alle anderen Objekte geändert werden. Dagegen beschreibt "Affects" aus Sicht eines bestimmten Objekts notwendige Änderungen anderer Klassen, falls an diesem Änderungen vorgenommen werden. In dieser Sicht wird die Stabilität des Gesamtsystems berechnet. Dieser Wert ergibt sich aus der durchschnittlichen Anzahl von betroffenen Objekten, die von Änderungen betroffen sind, falls ein gegebenes Objekt geändert wird (Durchschnitt aller Werte der Spalte "Affected"). Auch hier werden besondere Bereiche durch rote Schrift hervorgehoben. Es handelt sich um hohe Abhängigkeitswerte im Vergleich zum restlichen System. IBM bezeichnet dieses Konzept zur Identifikation von kritischen Bereichen als "heat map". Sowohl in der Sicht "Local Dependencies" als auch "Global Dependencies" können die Werte nach HTML zur Archivierung exportiert werden.

4.9.4 What-if-Analyse

Diese Sicht visualisiert die Auswirkungen von Änderungen im Softwaresystem. Dabei werden Verzeichnisse und Pakete durch Rechtecke dargestellt. Darin befinden sich dazugehörige Klassen und Interfaces, die durch Kreise repräsentiert werden. Zu Beginn sind alle Kreise grün, nur ein ausgewähltes Objekt wird durch rote Farbe hervorgehoben. Auf dieses Objekt bezieht sich die What-if-Analyse. Aus Gründen der Übersichtlichkeit werden keine Paket, Klassen oder

Schnittstellennamen angezeigt. Für ein Objekt erfährt man Genaueres, wenn der Mauszeiger für einen kurzen Augenblick über dieses bewegt wird. Es öffnet sich ein Anzeige, die nähere Informationen zur Herkunft, API und falls vorhanden Anti-Patterns gibt. Es gibt noch weitere Möglichkeiten ein bestimmtes Objekt ausfindig zu machen. Über Navigation -> Find kann nach einem bestimmten Objekt gesucht oder ein bestimmtes Objekt direkt ausgewählt werden. Abbildung 26 zeigt ein Beispiel für eine What-if-Analyse.

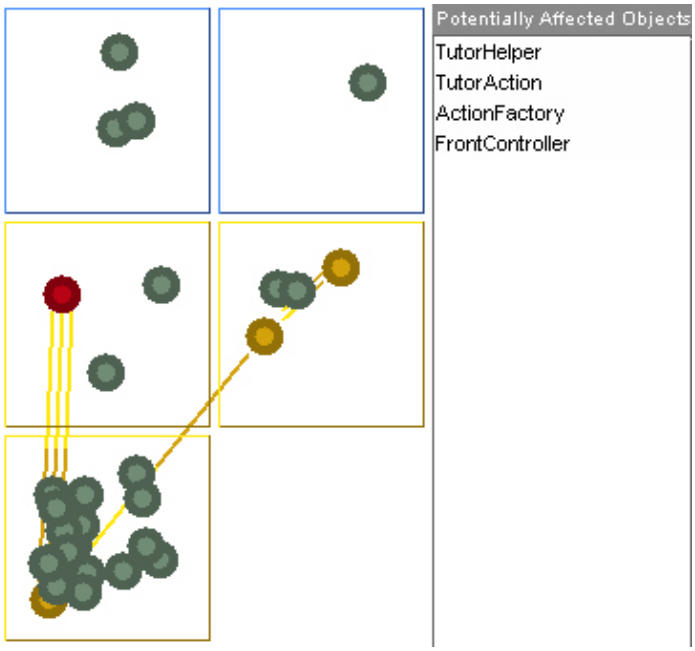


Abbildung 26 Abhängigkeitsgraph in SA4J

Die Analyse erfolgt aus Sicht des rot markierten Objekts. Von diesem Objekt ausgehend durchläuft das Programm das Netz der abhängigen Klassen, die orange hervorgehoben werden. Die Namen der betroffenen Objekte werden nochmals der Reihe nach aufgelistet ("Potentially Affected Objects"). Es besteht die Möglichkeit einen Screenshot einer What-if-Analyse zu machen oder die Liste der betroffenen Objekte nach HTML zu exportieren.

4.9.5 Paket-Analyse

Diese Sicht soll Auskunft über die physikalische Struktur des Softwaresystems geben. Die Analyse soll Aufschlüsse darüber geben, wie gut das System in Pakete partitioniert ist. Es werden die Paket-basierte Metriken Instability, Abstractness und Imbalance berechnet. Abstractness berechnet pro Paket die Anzahl von Schnittstellen und abstrakten Klassen. Instability ergibt sich aus dem Verhältnis dependencies und der Summe von dependencies und dependents

des Pakets. In der Spalte "Bonding" wird das Verhältnis zwischen internen Abhängigkeiten und der Gesamtanzahl aller Abhängigkeiten in dem betrachteten Paket. Ein hoher Wert deutet auf eine keines standalone-Paket hin, das nur ein kleines Interface nach außen bietet. Die Spalte "Link Density" bezeichnet die internen Abhängigkeiten (dependencies und dependents) bezogen auf die Anzahl der Objekte in dem betrachteten Paket. Auch diese Paketübersicht kann als HTML-Seite gespeichert werden.

4.9.6 Summary

Eine Zusammenfassung listet nochmal die wichtigsten Merkmale der Analyse auf. Die Gesamtstabilität des Systems sollte laut IBM um die 90% betragen. Stabile System sollen einen Wert oberhalb von 90% aufweisen. Auf einen Blick werden hier die verschiedenen Abhängigkeitsbeziehung für das Gesamtsystem aufgelistet, gefolgt von den Strukturmustern. Dieser HTML-Report kann auch gespeichert werden.

4.9.7 Fazit

Obwohl es sich um eine Vorabversion handelt, so macht SA4J einen guten Eindruck. Bereits jetzt kann das Programm in der Praxis produktiv eingesetzt werden. Die Analyse-Modi, vor allem What-if und Skeleton sind beeindruckend. Die einzelnen Analysewerkzeuge sind nicht unabhängig voneinander sondern sind sehr gut miteinander integriert. Wenn der Anwender beispielsweise in der Sicht für die lokalen Abhängigkeiten ein Objekt auswählt und anschließend in die What-if-Sicht wechselt, so ist das ausgewählte Objekt auch hier markiert. Hilfreich sind auch die Zoom-Möglichkeiten in den verschiedenen Diagrammen. Bereits in dieser Betaversion stehen schon Hilfefunktionen für die einzelnen Analysewerkzeuge zur Verfügung. In jeder Sicht können die Ergebnisse gespeichert oder exportiert werden. Das Programm zeigt nicht nur die verschiedenen Abhängigkeitsbeziehungen durch unterschiedliche Darstellungen auf, sondern berechnet auch Metriken und bestimmt Strukturmuster. In einer finalen Version sollte neben dem HTML-Export auch noch andere Formate zur Verfügung gestellt werden, insbesondere XML, um eine bessere Weiterverarbeitung der Ergebnisse zu gewährleisten. Bisher können die Ergebnisse nur als HTML-Dateien oder Screenshots archiviert werden. Das Programm ist sehr gut strukturiert. Die Menüführung ist sinnvoll und logisch. Das Programm reagiert flott auf Benutzereingaben, ist also keinesfalls träge. Nur bei sehr großen Projekten kann es zu vereinzelt zu Einfrieren des Programms kommen. Diese Probleme sollten in der endgültigen Version gelöst sein.

Die Szenarien, die in Sektion 1.2 ausführlicher beschrieben sind, wurden folgendermaßen bewertet:

- Z1 (Qualitätsreporting): Die berechneten Werte der Metriken können in HTML und XML exportiert werden und mittels eigens herzustellender XSL-Transformatoren in andere Formate umgewandelt werden.
- Z2 (Anomalieanalyse): Die Identifikation von Anomalien wird gut unterstützt. Es können zwar keine eigene Schwellwerte festgelegt werden aber durch die Visualisierung der Metriken und Abhängigkeiten kann der Nutzer Anomalien leicht entdecken.
- Z3 (QD-Entdeckung): Qualitätsdefekte werden nicht explizit erkannt obwohl es helfen soll einige „strukturellen Antipatterns" zu entdecken. Eine Angabe von Verbesserungsvorschlägen (Refaktorierungen) fehlt aber leider.

4.10 Instantiations - Code Pro Studio for Eclipse

Instantiations hat ein Werkzeug für Java-Entwicklern auf den Markt gebracht, die mit IBM WebSphere Studio oder Eclipse arbeiten. Das Unternehmen hat sich zum Ziel gesetzt, dass hochqualitative Java-Projekte nur einen kurzen Entwicklungsprozess durchleben sollen um somit Kosten einsparen zu können. Code Pro Studio ist kostenpflichtig, so dass für den kommerziellen Gebrauch eine Linzens erforderlich ist. Um sich einen Überblick zu verschaffen, kann nach einer Anmeldung eine zeitlich limitierte Testversion heruntergeladen werden. In diesem Kapitel wird die Testversion von Code Pro Studio Version für Eclipse betrachtet.

Die betrachtete Software besteht aus den Features dreier Komponenten, die auch separat zu einem günstigeren Paket bezogen werden können (aus diesem Grund überschneiden sich teilweise die Features der Komponenten): *Advisor*, *Agility* und *Build*.

- Advisor beschäftigt sich mit der Analyse von Java-Quellcode. In diesen Bereich fallen Code Audits, Klassen- und Paketabhängigkeitsanalyse, Metriken und eine Javadoc-Bearbeitungskomponente.
- Agility besitzt Features wie die Erstellung von Design Patterns, verbesserte Eclipse-Views (z. B. Java+ Perspective oder Java Browsing+ Perspective), neue Wizards, Einstellungsmöglichkeiten der Import- und Exportmöglichkeiten, Editor-Erweiterungen und CVS-Erweiterungen aber auch eine Scheduler-Komponente zusammengefasst.
- Build verfügt über Features wie Deployment-Automation, CVS-Erweiterungen, Ant-Erweiterungen oder auch eine Scheduler-Komponente.

Im Hauptmenü der eclipse Workbench ist ein neuer Menüeintrag hinzugekommen, von dem aus alle Komponenten der Software erreicht werden und eingestellt werden können. Neben einem Button, um bequem zu dem Einstellungsdialog zu gelangen, kann man von hier aus auch die Hilfe anwählen. Außerdem

kann hier zwischen den Views und den Perspektiven gewechselt werden. Allerdings muss man zuvor über Window > Show View bzw. Window > Open Perspective die gewünschten Views und Perspektiven) aktivieren. Auf dem Testsystem (Eclipse 3.0.0 unter Linux) konnten nicht alle Views und Perspektiven korrekt angezeigt werden. Je nachdem in welcher Perspektive man sich befindet, befinden sich in der Workbench-Toolbar neue Icons. So zum Beispiel kann in der Collaboration-Administration-Perspektive ein Wizard für die Erstellung von Java Applets und Design Patterns aufgerufen werden. Außerdem kann ein Speichermonitor eingeblendet werden, der die Speicherauslastung anzeigt. Während dem Test konnte über den Wizard zur Erstellung von Design Patterns keine neuen Klassen angelegt werden; es kam immer wieder zu Fehlermeldungen.

4.10.1 Metriken

Eine Analyse des Quellcodes erfolgt am schnellsten über das Kontext-Menü im Package Explorer. Durch einen Klick auf die rechte Maustaste kann über den Eintrag „CodePro Tools“ ein Werkzeug ausgewählt werden und auf das ausgewählte Element angewendet werden. In einer eigenen Sicht werden die Ergebnisse präsentiert. Abbildung 27 zeigt, wie die Metrik-Ergebnisse angezeigt werden.

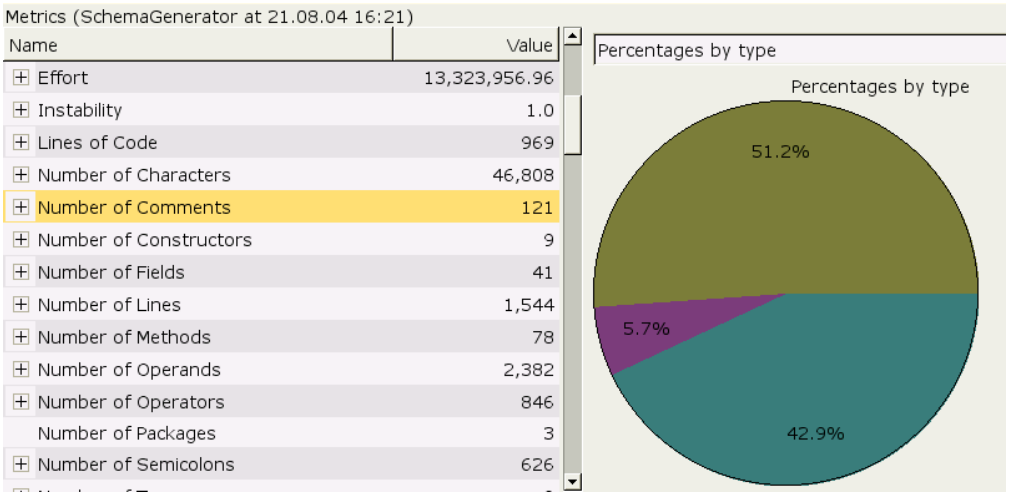


Abbildung 27 Tortengraphiken in Code Pro

Weiterhin unterstützt Code Pro Studio eine Vielzahl von Metriken. Diese sind in mehrere Kategorien eingeteilt. Angefangen mit einfachen Metriken wie LOC (Basic) existieren auch komplexere Metriken, die sich auf die Gruppen Complexity, Dependency, Halstead, Inheritance und Ratio aufteilen. Tabelle 9 bietet einen detaillierten Einblick in den Umfang an Metriken.

Tabelle 9 Tabelle: Verwendete Metriken in CodePro Studio

Verwendete Metriken	
Average Lines of Code Per Method	Average Number of Constructors Per Type
Average Number of Fields Per Type	Average Number of Methods Per Type
Average Number of Parameters	Lines of Code
Number of Characters	Number of Comments
Number of Constructors	Number of Fields
Number of Lines	Number of Methods
Number of Packages	Number of Semicolons
Number of Types	Average Block Depth
Average Cyclomatic Complexity	Weigthed Methods
Abstractness	Afferent Couplings
Distance	Efferent Couplings
Instability	Number of Operands
Number of Operators	Number of Unique Operands
Number of Unique Operators	Program Length
Program Vocabulary	Program Volume
Inheritance	Average Depth of Inheritance Hierarchy
Average Number of Subtypes	Comments Ratio

Metriken können auf einzelne Komponenten wie Projekt, Paket oder Type (Klasse oder Schnittstelle) angewendet werden. Das Plugin bietet die Möglichkeit weitere Metriken hinzuzufügen. Dies geschieht mit den Eclipse-bekannten Extension Points. Metrik-Ergebnisse können als Report in verschiedene Formate wie HTML oder XML exportiert werden. Es stehen verschiedene Reportarten zur Verfügung, außerdem erlaubt das Programm eigene Reportarten zu definieren. Wie sich die einzelnen Metriken berechnen, ist der sehr umfangreichen Code-Pro-Eclipse-Hilfe zu entnehmen.

4.10.2 Analyse von Abhängigkeiten

Code Pro Studio bietet eine Dependency-Komponente, mit der Abhängigkeiten zwischen Projektkomponenten in unterschiedlicher Granularitätsstufen analysiert werden können. Eine Analyse findet auf Projekt-, Paket oder Type-Ebene statt. Das Ergebnis wird in einer Graphenstruktur dargestellt. Abbildung 28 zeigt einen Abhängigkeitsgraphen auf Paketebene.

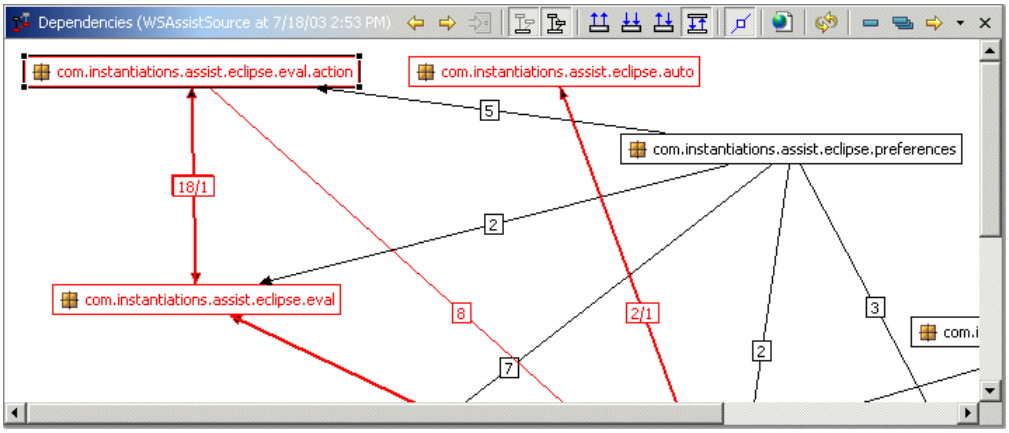


Abbildung 28 Abhängigkeitsgraph in Code Pro

Die Entwickler beabsichtigen mit der graphischen Repräsentation von Abhängigkeiten eine schnellere und effizientere Suche von Designmängeln. Eine aufwendige Suche im Quellcode ist somit nicht mehr notwendig, so dass Kosten aufgrund von Zeiteinsparungen reduziert werden können. Code Pro Studio bietet die Möglichkeit Zyklen zu identifizieren.

4.10.3 Code Audits

Die Audit-Komponente prüft den Quellcode auf eine definierte Audit-Regelmenge und zeigt in einer eigenen Sicht die Regelverstöße an. Ein einfaches Beispiel wäre eine Regel, die prüft, ob Klassennamen immer mit einem Großbuchstaben beginnen (Naming Conventions). Code Pro Studio schlägt dem Benutzer einige Lösungsmöglichkeiten vor, so dass die Audit-Schranke der betrachteten Regel nicht mehr verletzt wird.

Ziel dieser Komponente ist es, Coding-Probleme schon während der Entwicklungszeit zu finden. Ohne das frühzeitige Auffinden solcher Fehler propagieren sich Fehler durch eine Vielzahl von Dateien, was eine anschießende Fehlerkorrektur zu einem aufwendigen Unterfangen macht. Tabelle 10 zeigt eine Auswahl von Audit-Regeln, die sich standardmäßig im Softwareumfang befinden.

Tabelle 10 Bereiche, für die Audit-Regeln existieren

Verwendete Audits	
Exceptions	Import Usage
Inheritance	Javadoc Conventions
Modifier Usage	Performace
Portability	Property Files

Verwendete Audits	
Possible Errors	Semanatic Errors
hreads and Synchronization	XML
EJB	Formatting
Internationalization	Naming Conventions
Spell Checking	Coding Style

4.10.4 Fazit

Es handelt sich um ein kommerzielles Plugin, dass sehr gut in Eclipse integriert ist. Es steht sehr viel Funktionalität zur Verfügung, um die Codequalität eines Softwareprojekts zu verbessern. Allerdings scheint es so, als würden hauptsächlich nur einfache Metriken berechnet werden. Mit Audits können ebenfalls Probleme frühzeitig behoben werden. Somit wird gewährleistet, dass über das komplette Projekt für das Projekt definierten Regeln eingehalten werden.

Die Szenarien, die in Sektion 1.2 ausführlicher beschrieben sind, wurden folgendermaßen bewertet:

- Z1 (Qualitätsreporting): Die berechneten Werte der Metriken können in HTML und XML exportiert werden und mittels eigens herzustellender XSL-Transformatoren in andere Formate umgewandelt werden.
- Z2 (Anomalieanalyse): Die Identifikation von Anomalien wird kaum unterstützt da keine Schwellwerte festgelegt oder problematische Stellen hervorgehoben werden.
- Z3 (QD-Entdeckung): Qualitätsdefekte werden nicht explizit erkannt aber die Regeln gehen in eine ähnliche Richtung entdecken. Eine Angabe von Verbesserungsvorschlägen wird angegeben um die Auslösung der Regeln beim nächsten Durchlauf zu verhindern.

5 Weitere Werkzeuge

Messwerkzeuge gibt es heutzutage wie Sand am Meer. Neben den Messwerkzeugen die in Kapitel 4 vorgestellt wurden exstieren noch viele weitere, die sowohl statische als auch dynamische Metriken berechnen. Dabei sind sowohl freie Messwerkzeuge als auch kommerzielle auf dem Markt vorhanden. Weiterhin gibt es natürlich nicht nur Werkzeuge für die Programmiersprache Java, sondern auch für andere Programmiersprachen wie C++, C oder Fortran. In Rahmen dieser Untersuchung konnten leider nicht alle Messwerkzeuge analysiert werden.

Meist aktuelle Listen von Messwerkzeugen findet man im Internet z.B. auf den Seiten des Testing-FAQs (http://testingfaqs.org/t-static.html) der Newsgroup comp.software.testing, den Seiten von Chris Lott (http://www.chris-lott.org/resources/cmetrics/), den Seiten der Firma VerifySoft (http://www.verifysoft.com/de_overview.html), sowie auf den Seiten des NIST Projektes SAMATE (Software Assurance Metrics and Tool Evaluation) (http://samate.nist.gov/index.php/Tools). Des Weiteren existieren sehr gute Informationen über Softwaremetriken auf den Seiten von Wikipedia sowohl im englischsprachigen (http://en.wikipedia.org/wiki/Software_metrics) als auch deutschsprachigen (http://de.wikipedia.org/wiki/Softwaremetrik) Teil.

In diesem Kapitel werden nun kurz weitere Messwerkzeuge zur statischen sowie zur dynamischen Softwareanalyse vorgestellt.

5.1 Weitere Werkzeuge zur statischen Softwareanalyse

5.1.1 Jmetric

Dies ist ein Messwerkzeug für Java welches im Rahmen von Forschungsaktivitäten an der School of Information Technology der Swinburne University of Technology entstanden ist (http://www.it.swin.edu.au/projects/jmetric/default.htm). Neben der Berechnung unterschiedlichster Metriken werden auch verschiedene Präsentationsformen (d.h. Charts und Diagramme) angeboten.

5.1.2 CodePro Express

CodePro Express ist ein kommerzielles Messwerkzeug für Java der Firma Instantiations (http://www.instantiations.com/codepro/express) welches voll in die eclipse-DIE integriert ist. Neben der Generierung von Reports in HTML und XML

werden auch die unterschiedlichsten Metriken berechnet und in eclipse durch zusätzliche graphische Elemente wie Tortengraphiken dargestellt (http://www.instantiations.com/codepro/ws/docs/features/metrics/metrics.html)

5.1.3 JavaNCSS und JavaNCSS4eclipse

Diese beiden Messwerkzeuge sind OSS Messwerkzeuge für Java und frei im Internet verfügbar (http://www.kclee.de/clemens/java/javancss/, http://sourceforge.net/projects/jncss4eclipse/). JavaNCSS4eclipse stellt dabei ein Plugin für die eclipse IDE dar. JavaNCSS erstellt mehrere Metriken und kann die Informationen in XML exportieren.

5.1.4 Jrefactory

JRefactory ist ein Werkzeug das den Entwickler sowohl bei der Refaktorierung als auch beim Messen des Systems unterstützt und von Chris Seguin als OSS freigegeben ist (http://sourceforge.net/projects/jrefactory/). Es soll laut Webseite eine Metrik-Erweiterung besitzen und Qualitätsdefekte wie Cut&Paste (Code Duplication) erkennen können.

5.1.5 DependencyFinder

Dieses Messwerkzeug ist darauf ausgelegt Abhängigkeiten in Java Quellkode zu entdecken (http://depfind.sourceforge.net/) und ist als OSS frei verfügbar. Momentan werden über 60 Metriken auf verschieden Abstraktionsebenen gemessen (http://depfind.sourceforge.net/Manual.html#OOMetrics).

5.1.6 Code Analyzer

CodeAnalyzer ist ein nicht-kommerzielles Messwerkzeug für C, C++ und Java welches mehrere Zeilenbasierte Metriken berechnet und die Ausgaben nach CSV (Comma Seperated Values) exportiert (http://sourceforge.net/projects/codeanalyze-gpl/).

5.1.7 Dependometer

Ein frei verfügbares Messwerkzeug der Firma Valtech für Java welches die Metriken von Robert C. Martin, Craig Larman und John Lakos berechnen kann (http://sourceforge.net/projects/dependometer/).

5.1.8 Jmove

jMove ist ein OSS Messwerkzeug für Java, welches die Metriken von Robert Martin berechnet und die Ausabe als XMI exportieren kann (http://sourceforge.net/projects/jmove/).

5.1.9 YADA

Yet Another Dependency Analyzer (YADA) ist ein Messwerkzeug für Java welches Abhängigkeiten auf Paket-Ebene berechnet (http://sourceforge.net/projects/eclipse-yada/)

5.1.10 JavaCount

Dieses Messwerkzeug für Java berechnet LOC und die Differenz zwischen zwei Versionen einer Datei (http://csdl.ics.hawaii.edu/Tools/JavaCount/JavaCount.html).

5.1.11 McCabe IQ

Die Firma McCabe hat das kommerzielle Messwerkzeuge McCabe IQ entwickelt welches mehrere Komplexitäts-orientierte und Datenbezogene Metriken berechnet (http://www.mccabe.com/iq_qa.htm).

5.1.12 Sotograph (Software-Tomograph)

Ein sehr großes kommerzielles Messwerkzeug für Java, EJB und C++ das sehr viele Metriken berechnen und visualisieren kann (http://www.software-tomography.com/html/sotograph.htm)

5.1.13 QStudio for Java

Ein plugin für die eclipse IDE welches Metriken für Java Software berechnet und von der Firma QA-Systems vertrieen wird (http://www.qa-systems.com/eclipse)

5.1.14 CMTJava

CMTJava ist ein Messwerkzeug für Java welches auf Komplexitätsmessung ausgelegt ist und die McCabe, Halstead und LOC berechnet (http://www.testwell.fi/cmtjdesc.html)

5.1.15 Krakatau Metrics

Kratakau Metrics ist ein kommerzielles Messwerkzeug für Java undC/C++ von der Firma PowerSoft welches verschiedene Metriken berechnen kann und verschiedene Darstellungsformen wie Kiviatgraphen unterstützt (http://www.powersoftware.com/). Andere Produkte der Firma PowerSoft dienen des Metrikmanagements sowie der Visualisierung.

5.1.16 CodeReports

Das Messwerkzeug für Java der Firma SmartBear Software dient der Vermessung von Softwaresystemen aus Versionierungsystemen und insb. dem Reporting über diese (http://www.codehistorian.com/codereports-overview.php).

5.1.17 JCVSReport

JCVSReport istein Messwerkzeug für Java welches aus dem Versionieungssystem CVS weitere Metriken über die Evolution des Softwaresystems ermitteln kann (http://www.cs.toronto.edu/~james/JCVSReport/).

5.1.18 CodeSonar

Die Firma GrammaTech hat mit CodeSonar ein kommerzielles Messwerkzeug für C/C++ entwickelt welches typische funktionale Defekte entdeckt und Daten nach XML exportieren kann (http://www.grammatech.com/products/codesonar/overview.html).

5.1.19 Jstyle

Ein kleines Meswerkzeug für Java welches auch Code Reviews durchführt und einige Conventionen überprüft (http://www.mmsindia.com/jstyle.html).

5.1.20 SimianUI

Das eclipse-basierte Messwerkzeug für Java der Firma Integility überprüft die Ähnlichkeit von Quellkode um Code Duplikate aufzuspüren (http://www.integility.com/simianui.html).

5.1.21 Code Analysis Plugin - CAP

Ein eclipse-basiertes Messwerkzeug welches die Robert Martin Metriken berechnet, die Abhängigkeiten ermittelt und mehrere Visualisierungstechniken zur Verfügung stellt (http://cap.xore.de).

5.2 Werkzeuge für dynamische Softwareanalysen

5.2.1 Eclipse Profiler

Der Eclipse Profiler ist ein dynamisches Messwerkzeug für Java welches Informationen über die Methodenaufrufe ermittelt und einen Call-Graphen erstellen kann (http://eclipsecolorer.sourceforge.net/index_profiler.html).

5.2.2 Extensible Java Profiler

Ein open-source Projekt welches Java welches Informationen über die Methodenaufrufe ermittelt (http://ejp.sourceforge.net).

5.2.3 jMechanic

Das Messwerkzeug jMechanic berechnet Metriken über die CPU oder Heapnutzung in Java (http://jmechanic.sourceforge.net/).

5.3 Werkzeuge zur Überprüfung von Programmierrichtlinien

5.3.1 JavaChecker

JavaChecker ist ein Werkzeug für Java welches Code-Defekte (inaccurate exceptions handling, hiding defects, style defects, violations of standard usage contracts, synchronization defects) erkennt (https://javachecker.dev.java.net/).

5.3.2 PMD

Das Werkzeug PMD dient der Überprüfung der unterschiedlichsten Richtlinien welche als Regeln abgelegt sind und individuell erweitert werden können (http://pmd.sourceforge.net/). Dabei wird auch Dead Code und Duplicated Code angezeigt.

5.3.3 SimScan

Dieser Similarity Scanner erkennt ähnlichen Quellkode (http://www.blue-edge.bg/download.html).

5.3.4 Checkstyle

Checkstyle, ein Plugin für die eclipse IDE, ist ein Messwerkzeug für Java welches nach Abweichungen von vorgegeben Kodierungsstandards sucht (http://sourceforge.net/projects/eclipse-cs).

5.3.5 Clover

Clover ist ein eclipse Plugin das die Codeabdeckung und andere testbezogene Metriken berechnet sowie einen Export in HTML, PDF oder XML anbietet (http://www.thecortex.net/clover).

5.3.6 Demeter Cop

Dieses eclipse Plugin refaktoriert Java Quellkode automatisch indem es das Gesetzt von Demeter umsetzt (http://demetercop.sourceforge.net/).

5.3.7 Duplication Management Framework

Eine Umgebung zur Ausführung von Algorithmen zur Entdeckung von Code Duplikaten (http://www.sourceforge.net/projects/dupman).

5.3.8 EclipsePro Audit

EclipsePro Audit ist ein Werkzeug der Firma Intantiations das mehr als 700 regeln und Metriken berechnet (http://www.instantiations.com/eclipsepro/audit.htm).

5.3.9 FEAT

Das Feature Analysis and Exploration Tool (FEAT) wurde von Martin Robillard während seiner Dissertation entwicelt und ermittelt einige Eigenarten im Java Quellkode (http://www.cs.ubc.ca/labs/spl/projects/feat/).

5.3.10 Lint und jLint

Ein eclipse Plugin für das Messwerkzeug Lint welches verschiedene Metriken und Datenflussanalysen für Java Softwaresysteme erstellen kann (http://jlint.sourceforge.net/).

5.3.11 Indus

Indus ist ein Messwerkzeug für Java welches einige sehr spezifische Mess- und Analysetechniken wie bspw. points-to-Analysen, escape-Analysen, und Abhängigkeitsanalysen durchführt (http://indus.projects.cis.ksu.edu).

5.3.12 QA-J

QA-J ist ein Messwerkzeug für Java der Firma Programming Research welches auch für andere Programmiersprachen existiert und Programmierrichtlinien überprüft (http://www.programmingresearch.com).

5.4 Weitere Werkzeuge

5.4.1 Creole

Das Plugin für die eclipse DIE welches das Werkzeug Shrimp integriert und die Struktur von Java Quellode visualisiert (http://www.thechiselgroup.org/creole). Diese Visualisierungen können verwendet werden um manuell Probleme zu entdecken.

6 Fazit

Heutzutage gibt es eine Vielzahl von Messwerkzeugen zur Analyse von Java Quellcode. Allerdings konnte sich keines der getesteten Werkzeuge klar als Favorit hervortun. Würde man alle Vorteile der einzelnen Programme zusammenführen, so hätte man ein sehr gutes Analyse-Werkzeug an der Hand.

Beispielsweise überzeugt in Code Studio Pro die Anzeige der Messergebnisse. Allerdings werden dem Anwender nicht die Zusammenhänge der Messergebnisse mit der Softwarequalität deutlich gemacht. Es existiert keine Funktionalität zur Auswertung der Ergebnisse. Es werden keine Graphen angeboten, mit denen Ausreißer identifiziert werden können. Code Pro Studio bietet keine Vorschläge zur Code-Verbesserung an.

Die Analyse der Abhängigkeiten sind in Code Pro Studio und Teaminabox sehr gut gelöst. Eine Graphische Repräsentation ermöglicht eine einfache Inspektion der Zusammenhänge zwischen Systemkomponenten. Außerdem können Zyklen identifiziert werden. Ob diese Graphen in sehr großen Projekten übersichtlich sind, ist fraglich, allerdings ist solch eine Komponente wesentlich effizienter als die Analyse des Quellcodes rein auf Basis von Metriken.

6.1 Defizite heutiger Software Messwerkzeuge

Die meisten Werkzeuge beherrschen die Berechnung der bekannten Metriken wie McCabe, Halstead oder Chidamber-Kemerer. Ein Metrik-Experte oder eine fähige Person mit einem zielorientierten Analyseplan (bspw. Auf Basis von GQM) kann basierend auf diesen Daten die Qualität besser (d.h. quantitativer) abschätzen und einige Problemfälle identifizieren.

Ein Code-Analyse Tool sollte aber nicht nur Metriken berechnen und die Ergebnisse präsentieren, sondern Funktionalität bieten, um diese Werte zu analysieren, zu interpretieren, und zu visualisieren. Wie dies aussehen könnte, haben wir bereits in der Einführung besprochen. Das Tool sollte den Entwickler dabei unterstützen, in wie Weit Aussagen mit Hilfe von Metriken auf Aspekte wie Lesbarkeit oder Komplexität gemacht werden können. Dem Entwickler sollten Vorschläge gemacht werden, welche Änderungsmöglichkeiten existieren, um die Qualität des Softwaresystems zu verbessern. Zusammengefasst haben heutiger Messwerkzeuge defizite in folgenden Bereichen:

- **Interpretationsunterstützung**: Was der Benutzer mit den ermittelten Metriken tun soll, bleibt oft im Dunkeln. Es gibt nur vereinzelt Hinweise darauf ab wann eine Metrik ein Problem darstellt. Die Werkzeuge helfen oft nicht bei der Interpretation der Metrikdaten, und wenn doch, dann gibt es oft nur einen Schwellwert der dazu führt, dass ein Objekt (und eine spezifische Metrik davon) farblich markiert sind.
- **Problembehandlung**: Basierend auf einem Metrikprofil können nur in einfachen Fällen konkrete Maßnahmen zur Beseitigung der Defekte abgeleitet werden. Heutige Messwerkzeuge teilen höchstens das Problem (z.B. zu hohe Kopplung) mit, aber schaffen es nicht, konkrete Handlungsanweisungen vorzuschlagen, wie man das Problem entfernt, geschweige den präventiv verhindern kann.
- **Metrikhandling**: Wenn eine Metrik ein Problem aufzeigt es sich aber herausstellt, das man an dieser Stelle nichts ändern kann (z.B. aus Gründen der Performance oder durch Verwendung einer externen Bibliothek) ist es nicht möglich diese Entscheidung oder Erfahrung in einem der Werkzeuge oder dem Quellkode selbst zu hinterlegen. Bei wiederkehrenden Messungen werden die gleichen (unausweichlichen) Probleme immer und immer wieder aufgefunden.
- **Anomalieentdeckung**: Die Identifikation von Ausreißern bzw. Anomalien in Metrikdaten durch automatische oder visuelle Verfahren (Stichworte Visuelles Data Mining, Visual Analytics) wird nur selten umgesetzt. Eine Anomalieanalyse ist nur mit weiteren Werkzeugen möglich, falls das Messwerkzeug seine Daten in ein Standardformat (z.B. CSV) exportieren kann. Beispielsweise könnten Diagramme und Histogramme in Excel zur Identifikation von Ausreißern verwendet werden.
- **Benchmarking**: Es stehen keine Informatione zum Benchmarking zur Verfügung. Ein Vergleich des Metrikprofils mit anderen (Standard-) softwaresystemen oder selbst Open-Source Softwaresystemen ist nicht möglich. Ob sich das Metrikprofil für die spezifische Anwendungsdomäne, Programmiersprache oder Architekturparadigma in einem „normalen“ Feld bewegt oder startk davon abweicht, kann nicht ermittelt werden.

7 Literatur

[1] Wolfgang H. Janko and Stefan Koch. Software-Metriken. Elektronisch verfügbar unter http://wwwai.wu-wien.ac.at/~koch/lehre/inf-wirt-3-pi-ss-04/metriken/metriken.pdf, Stand: 14.08.2004, 2004.

[2] Dr. Linda H. Rosenberg. Applying and Interpreting Object Oriented Metrics. Elektronisch verfügbar unter http://www.literateprogramming.com/ooapply.pdf, Stand: 14.08.2004, 2004.

[3] Fowler, M. (1999). Refactoring: Improving the Design of Existing Code (1st ed.): Addison-Wesley.

[4] Riel, A. J. (1996). Object-oriented Design Heuristics. Reading, Mass.: Addison-Wesley Pub. Co.

[5] Roock, S., & Lippert, M. (2004). Refactorings in großen Softwareprojekten: Komplexe Restrukturierungen erfolgreich durchführen (in German). Heidelberg: dpunkt Verlag.

[6] Whitmire, S. A. (1997). Object-oriented Design Measurement. New York, NY, USA: John Wiley & Sons.

[7] Wolfgang H. Janko and Stefan Koch. Software-Metriken. Elektronisch verfügbar unter http://wwwai.wu-wien.ac.at/~koch/lehre/inf-wirt-3-pi-ws-03/metriken/metriken.pdf, Stand: 14.08.2005, 2005.

8 Biografien

Dipl.-Inform. Jörg Rech beschäftigt sich seit 1992 mit der professionellen Softwareentwicklung. Als Wissenschaftler und Projektmanager am Fraunhofer Institut für Experimentelles Software Engineering (IESE) war er in verschiedenen Industrie- und Forschungsprojekten wirksam. Er erhielt das Vordiplom und Diplom in Informatik mit dem Nebenfach Elektrotechnik von der Universität Kaiserslautern. Als Mitarbeiter am Lehrstuhl von Prof. Dieter Rombach (AG Software Engineering, AGSE) war er für verschiedene Lehrtätigkeiten verantwortlich. Seine Forschungsinteressen umfassen Wissensentdeckung in Software-Versionierungssystemen, Defektentdeckung, Code Mining, Code Retrieval, Softwareanalyse und Wissensmanagement. Jörg Rech publizierte mehrere wissenschaftliche Fachartikel im Bereich Software Engineering und Wissensmanagement und ist ein Mitglied der Gesellschaft für Informatik (GI).

E-Mail: joerg.rech@iese.fraunhofer.de

Sebastian Weber studiert Informatik an der Universität Kaiserslautern und arbeitet als Hilfswissenschaftler am Fraunhofer Institut für Experimentelles Software Engineering.

E-Mail: s_webe@informatik.uni-kl.de

Dokumenten Information

Titel:	Werkzeuge zur Ermittlung von Software-Produktmetriken und Qualitätsdefekten Studie zu Software-Messwerkzeugen 2005
Datum:	29. November 2005
Report:	IESE-108.05/D
Status:	Final
Klassifikation:	Öffentlich