Institut für Softwaretechnologie
Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Fachstudie Nr. 167

Marktanalyse statischer Codeanalysewerkzeuge für Java

Marius Bauer Michael Nistor
Albert Ziegenhagel

Studiengang:	Softwaretechnik
Prüfer:	Prof. Dr. Stefan Wagner
Betreuer:	Dipl.-Ing. Jan-Peter Ostberg
begonnen am:	29. Mai 2012
beendet am:	30. November 2012
CR-Klassifikation:	D.2.8, D.2.4

Inhaltsverzeichnis

1 Einleitung **9**
- 1.1 Begriffe . . . 9
- 1.2 Fokus der Fachstudie . . . 9
  - 1.2.1 Wichtige Anforderungen . . . 10
  - 1.2.2 Nicht berücksichtigte Aspekte . . . 10
- 1.3 Gliederung . . . 10

2 Ablauf der Fachstudie **11**
- 2.1 Phasen . . . 11
- 2.2 Zeitlicher Verlauf . . . 12

3 Marktüberblick **15**

4 Nutzungsszenario **29**

5 Bewertungskriterien **31**
- 5.1 K.O. Kriterien . . . 31
  - 5.1.1 Erweiterbarkeit . . . 31
  - 5.1.2 Updates . . . 32
  - 5.1.3 Plattform . . . 32
  - 5.1.4 Funktionsumfang . . . 32
- 5.2 Nicht relevante Kriterien . . . 33
  - 5.2.1 Art/Architektur . . . 33
- 5.3 Weitere Kriterien . . . 33
  - 5.3.1 Community/Support . . . 33
  - 5.3.2 Dokumentation . . . 34
  - 5.3.3 Benutzerfreundlichkeit . . . 34
  - 5.3.4 Blacklisting . . . 35
  - 5.3.5 Kosten . . . 35
  - 5.3.6 Lizenz . . . 35
  - 5.3.7 Performance . . . 36
  - 5.3.8 Quellcode vs. Bytecode . . . 36

6 Evaluation **37**
6.1 Voruntersuchung 37
6.2 Bewertungsschema 39
6.3 Werkzeuge 42
6.3.1 Checkstyle 42
6.3.2 ConQAT 46
6.3.3 FindBugs 50
6.3.4 JLint 53
6.3.5 Moose 55
6.3.6 PMD 58
6.3.7 Sonar 61
6.3.8 Understand 64
6.4 Resultat 67

7 Empfehlung **69**

Abbildungsverzeichnis

2.1 Ablauf der Fachstudie mit Phasen und Meilensteinen. Legende: Einar = Einarbeitung in das Thema , Rech = Recherche nach auf dem Markt befindlichen statische Codeanalysewerkzeuge , Info = Erste Analyse der Werkzeuge , Vorb = Vorbereitung auf Zwischen- bzw. Endvortrag , Vortr = Zwischen- bzw. Endvortrag Bewer = Erstellung des Bewertungsschemas , Vor = Voruntersuchung der gefundenen Werkzeuge , break = Pause Analy = Evaluation , Korr = Korrekturphase , Abgab = Abgabe der Fachstudie 13

6.1 Ausschnitt der Konfigurationsdatei für die Sun Coding Conventions 43
6.2 Screenshot nach der Ausführung von CheckStyle im Eclipse-Plug-in 44
6.3 XML Ausgabe von CheckStyle 45
6.4 Der aus Blocks (Funktions-Bausteinen) aufgebaute Analyse-Graph und eine Liste der verfügbaren Blocks 47
6.5 Grafische Ansicht der Analyse-Konfiguration in Eclipse mit Bearbeitungsmöglichkeiten 48
6.6 HTML-Ausgabe einer Clone-Detection-Analyse mit Visualisierung der gefundenen Codeduplikate als Tree Map 49
6.7 Eingabe-Konfiguration von FindBugs 51
6.8 Blacklisting Möglichkeiten 51
6.9 Grafische Ansicht der GUI mit den Resultaten nach der Analyse 52
6.10 Startbildschirm der Anwendung mit dem VM Kontextmenü 56
6.11 Anzeige der Resultate aus der generierten MSE Datei 56
6.12 Ruleset, welches die richtige Verwendung von clone()-Implementierungen prüft 59
6.13 Standard HTML Report von PMD 60
6.14 Blacklisting in Sonar 61
6.15 Der Inhalt einer Konfigurationsdatei, um die Untersuchung zu starten 62
6.16 Anzeige während des Scans 63
6.17 Hauptseite der Resultate, die nach einem Scan angezeigt werden kann 63
6.18 Konfigurationsdialog zum Auswählen von Checks, welche über der Codebasis ausgeführt werden sollen 65
6.19 Treemap, welche das Verhältnis von Codezeilen zu Kommentarzeilen anzeigt . 66

Tabellenverzeichnis

6.1 Ergebnis der Voruntersuchung anhand der K.O.-Kriterien 39
6.2 Punktzahl für jeden Wert auf der Bewertungsskala 40
6.3 Gewichtung der Bewertungskriterien 41
6.4 Endnote anhand der erreichten normierten Punktzahl 41
6.5 Resultat der Evaluation mit Bewertung der Kriterien 67

Kurzfassung

Im Rahmen dieser Fachstudie werden statische Codeanalysewerkzeuge für Java untersucht und bewertet. Die Grundlage der Bewertung entsteht durch Absprache zwischen den Autoren dieser Fachstudie und Herr Ostberg. Codeanalysewerkzeuge welche die grundlegenden Anforderungen überstehen, werden einer genaueren Evaluation unterzogen. Am Ende wird auf Grundlage der Evaluation eine Empfehlung für ein Codeanalysewerkzeug ausgesprochen.

Abstract

In this *Fachstudie* statistical code analysis tools for Java will be studied and evaluated. The basis of the evaluation is derived through an agreement with the authors of this report an Mr. Ostberg. Code analysis tools which outlast these basic requirements will be evaluated thoroughly. At the end of the report a recommendation for one code analysis tool will be issued based on the evaluation.

1 Einleitung

Die Qualität von Software wird immer wichtiger und soll möglichst während dem gesamten Entwicklungszyklus einer Software sichergestellt sein. Dafür werden heutzutage eine große Anzahl von Software-Werkzeugen eingesetzt, die den Entwickler dabei unterstützen sollen. Während der Implementierung werden deswegen u. a. statische Codeanalysewerkzeuge verwendet, die häufig fest in den Entwicklungsprozess integriert sind.

Allerdings gibt es eine Vielzahl von Codeanalysewerkzeugen am Markt, kommerzielle wie Open Source, die über unterschiedlichste Eigenschaften verfügen. Diese reichen vom angebotenen Funktionsumfang, über die Benutzbarkeit, bis hin zur Qualität der Ausgabe. Diese Fachstudie soll deswegen einen Überblick über die am Markt befindlichen Codenanalysewerkzeuge bieten sowie diese evaluieren und bewerten.

1.1 Begriffe

Statische Codeanalyse Der Begriff statische Codeanalyse umfasst die Untersuchung des Quell- oder Bytecodes, ohne den Code auszuführen. Wird der Code für die Analyse ausgeführt, spricht man von dynamischer Codeanalyse. Statische Codeanalyse umfasst z. B. die Messung von Umfangs- und Qualitätsmetriken, die Analyse des Datenflusses oder die Konformität der Softwarearchitektur mit dem Architekturentwurf.

Erweiterbarkeit Im Kontext der Fachstudie ist für die Erweiterbarkeit nicht notwendigerweise die Verfügbarkeit des Quellcodes erforderlich. Auch eine spezifizierte und dokumentierte Schnittstelle oder ein Plug-in-System eines, z. B. kommerziellen, Codeanalysewerkzeugs erfüllt den Begriff der Erweiterbarkeit.

1.2 Fokus der Fachstudie

Das Nutzungsszenario der Fachstudie im universitären Bereich, legt den Fokus der Marktanalyse nicht auf den produktiven Einsatz eines Codeanalysewerkzeuges, sondern auf die Möglichkeiten der Forschung und Weiterentwicklung. Dies hat Auswirkungen auf die an das Codeanalysewerkzeug gestellten Anforderungen.

1.2.1 Wichtige Anforderungen

Aus dem Fokus der Fachstudie ergeben sich die folgenden wichtigen Anforderungen: Erweiterbarkeit, Qualität und Umfang der Dokumentation, Aktivität der Community und Supportleistungen sowie Benutzerfreundlichkeit und Qualität der Ausgabe.

1.2.2 Nicht berücksichtigte Aspekte

Aus dem selben Grund finden folgende Untersuchungsaspekte keine oder nur geringe Berücksichtigung: Quantität und Qualität der Befunde und Ergebnisse, Integrierbarkeit in den Entwicklungsprozess und die Entwicklungsumgebung sowie die Performanz der Untersuchung.

1.3 Gliederung

Die Fachstudie ist in folgender Weise gegliedert:

Kapitel 2 – Ablauf der Fachstudie: Enthält die Beschreibung der einzelnen Phasen der Fachstudie und einen Terminplan mit allen Meilensteinen.

Kapitel 3 – Marktüberblick: Gibt eine möglichst vollständige Auflistung und Kurzbeschreibung aller am Markt vorhandenen statischen Codeanalysewerkzeuge für Java.

Kapitel 4 – Nutzungsszenario: Beschreibung des Szenarios, in dem das in der Fachstudie empfohlene Codeanalysewerkzeug später eingesetzt wird.

Kapitel 5 – Bewertungskriterien: Definition und Beschreibung der K.O.- und Bewertungskriterien, inklusive Gewichtung und Bewertungsskala für jedes einzelne Bewertungskriterium.

Kapitel 6 – Evaluation: Enthält zum einen die Definition des Bewertungsschemas, anhand dessen die Platzierung der evaluierten Codeanalysewerkzeuge ermittelt wird, und zum anderen die Voruntersuchung aller Codeanalysewerkzeuge aus der Marktübersicht entsprechend der K.O.-Kriterien. Die so ermittelten Codeanalysewerkzeuge werden anschließend detailliert auf die hin Bewertungskriterien untersucht und bewertet.

Kapitel 7 – Empfehlung: Abschließend wird auf Grundlage der Evaluation eine Empfehlung für ein oder mehrere Codeanalysewerkzeuge ausgesprochen.

2 Ablauf der Fachstudie

In diesem Kapitel wird der Ablauf der Fachstudie vorgestellt, die im Zeitraum von sechs Monaten vom 29.05.2012 bis 30.11.2012 durchgeführt wurde.

2.1 Phasen

Beginn der Fachstudie und Projektplan

Die Fachstudie begann am 29.05.2012 mit einem Kick-off-Meeting. Dipl.-Ing. Jan-Peter Ostberg stellte uns als Vertreter des Instituts für Softwaretechnologie die Problemstellung hinsichtlich der Auswahl eines statischen Codeanalysewerkzeugs für Java vor, dessen Lizenzierung eine Erweiterung im universitären Rahmen erlaubt. In einem späteren Analysegespräch konnten wir uns anhand vorbereiteter Fragen einen detaillierten Überblick von den Anforderungen an das auszuwählende Codeanalysewerkzeug verschaffen. Im Anschluss an das Kick-off-Meeting und das Analysegespräch planten wir die Phasen und Meilensteine der Fachstudie.

Recherche zum Marktüberblick

Als Erstes verschafften wir uns einen möglichst vollständigen Überblick der am Markt vorhandenen statischen Codeanalysewerkzeuge für Java. Zu jedem Werkzeug listeten wir relevante Informationen wie die Lizenz, den Entwicklungsstand und die Kernaussagen der Hersteller auf.

Analyse der Bewertungskriterien

Anhand der detaillierten Anforderungen entwickelten wir in Absprache mit Herr Ostberg Bewertungskriterien sowie deren Relevanz. Einige der Kriterien müssen auf jeden Fall erfüllt werden und wurden deswegen als K.O.-Kriterien eingestuft. Auch die Gewichtung der Kriterien erfolgte in Absprache mit Herr Ostberg. Am Ende dieser Phase hielten wir am 09.08.2012 einen Zwischenvortrag, in dem die bisherigen Ergebnisse, die Marktübersicht und die Bewertungskriterien, vorgestellt wurden.

Voruntersuchung anhand der K.O.-Kriterien

Im Anschluss an die Definition der Bewertungskriterien untersuchten wir alle während der Recherche gefundenen Codeanalysewerkzeuge auf die K.O.-Kriterien, um die für die ausführliche Evaluation geeigneten Werkzeuge herauszufiltern.

Definition eines Bewertungsschemas

Zeitgleich definierten wir ein Bewertungsschema, in dem die erreichbaren Punkte und die Umrechnung in eine Endnote für die zu untersuchenden Werkzeuge festgelegt wurden.

Evaluation der Werkzeuge

Anhand der vorher ausgearbeiteten Bewertungskriterien, evaluierten wir die nach der Voruntersuchung verbleibenden Codeanalysewerkzeuge. Die Bewertung der Kriterien erfolgte sowohl in textlicher als auch in tabellarischer Form.

Erarbeitung der Empfehlung

Auf Grundlage der Evaluation sprachen wir anschließend eine Empfehlung für eines bzw. mehrere Codeanalysewerkzeuge aus, da sich kein eindeutiger Sieger feststellen ließ.

Abgabe und Ende der Fachstudie

Die Übergabe der fertigen Fachstudie an Herr Ostberg, am 30.11.2012, und der Abschlussvortrag, der am 29.11.2012 gehalten wurde, bildeten das Ende der Fachstudie.

2.2 Zeitlicher Verlauf

Das Gantt-Diagramm in Abbildung 2.1 zeigt den zeitlichen Ablauf der in Abschnitt 2.1 beschriebenen Phasen und die erreichten Meilensteine M1 Marktübersicht (12.07.2012), M2 Definition der Bewertungskriterien (02.08.2012), M3 Zwischenvortrag (09.08.2012), M4 Evaluation (01.11.2012), M5 Empfehlung (08.11.2012), M6 Abschlussvortrag (29.11.2012) und M7 Abgabe (30.11.2012).

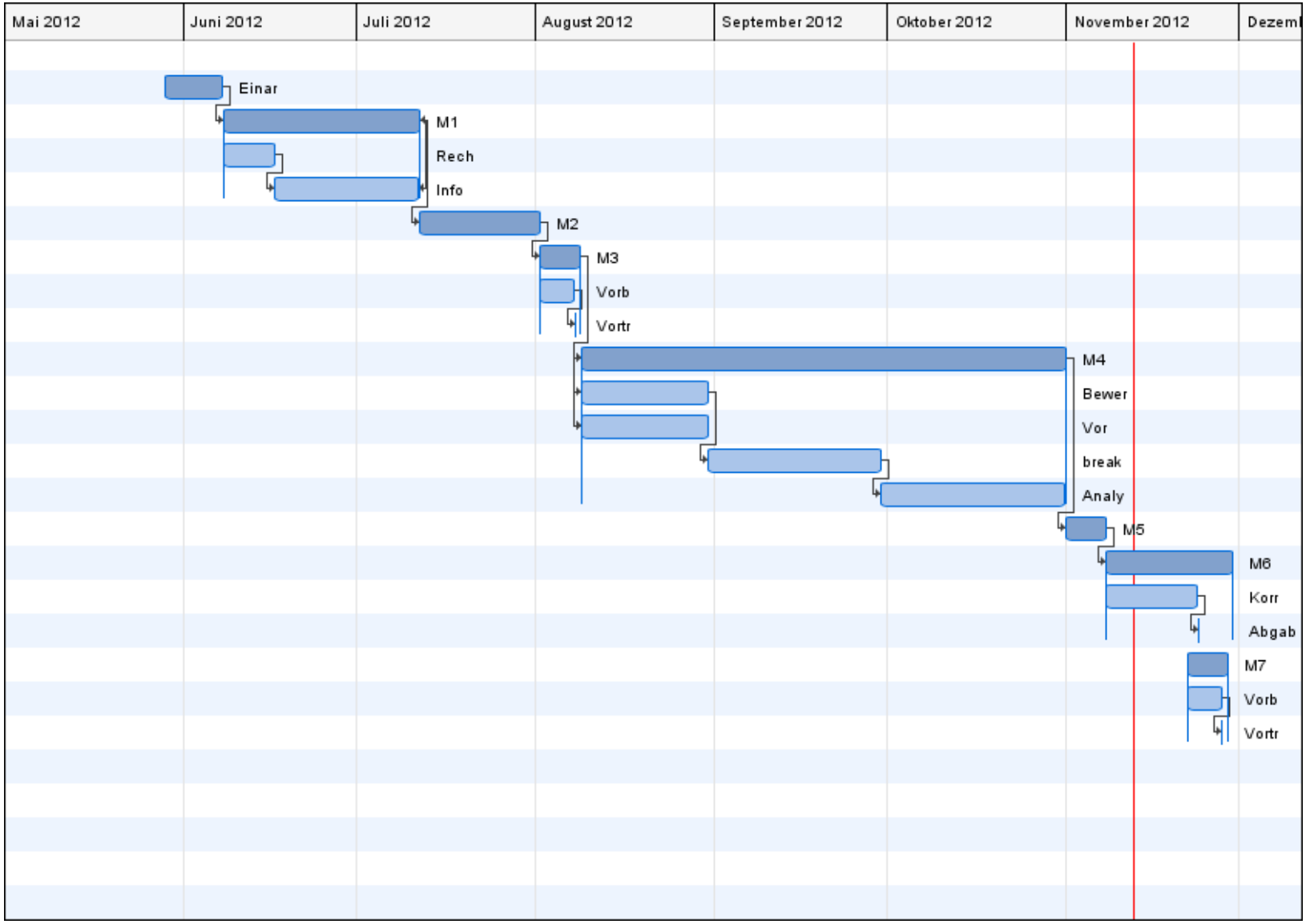


Abbildung 2.1: Ablauf der Fachstudie mit Phasen und Meilensteinen. Legende: Einar = Einarbeitung in das Thema , Rech = Recherche nach auf dem Markt befindlichen statische Codeanalysewerkzeuge , Info = Erste Analyse der Werkzeuge , Vorb = Vorbereitung auf Zwischen- bzw. Endvortrag , Vortr = Zwischen- bzw. Endvortrag Bewer = Erstellung des Bewertungsschemas , Vor = Voruntersuchung der gefundenen Werkzeuge , break = Pause Analy = Evaluation , Korr = Korrekturphase , Abgab = Abgabe der Fachstudie

3 Marktüberblick

In diesem Kapitel werden die derzeit verfügbaren statische Codeanalysewerkzeuge vorgestellt, die mindestens JAVA unterstützen. Die verschiedenen Tools werden stichwortartig beschrieben. Es wird keine Garantie auf Richtigkeit gegeben, da die hier angegebenen Informatione von den Herstellern und/oder Vertreiber stammen. Viele Hersteller kommerzieller Tools geben keine öffentlichen Informationen über die derzeitige Versionsnummer heraus. In diesem Fall, wird die jeweilige Versionsart von uns als unbekannt deklariert.

AgileJ StructureViews

- Reverse Engineering Tool zum Erstellen von UML Diagrammen.
- Eclipse-Plug-in
- Kostenlose Version für nicht kommerzielle Zwecke erhältlich

Untersuchte Version
1.7.10

Lizenz
kommerziell

Webseite
agilej.com

Bauhaus Suite

- Untestützung von: C, C++, C#, Java und Ada
- Erkennung von Stilverstöße, toten Code, Metrikausreißer, duplizierten Code, zyklische Abhängigkeiten und Architekturverstößen
- Hohe Anpassbarkeit bzw. Erweiterbarkeit.
- Optimierung auf Simplizität

Untersuchte Version
unbekannt

Lizenz
kommerziell

Webseite
axivion.com

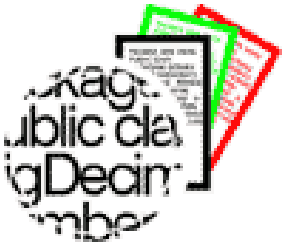
BugScout

- Unterstützung von: Java, PHP, ASP and C#
- Als SaaS in der Cloud
- Erkennung von Sicherheitslücken
- Benutzerdefinierte Sicherheitsregeln
- Überprüfung des Codes auf 'best practices'

Untersuchte Version
Unbekannt
Lizenz
kommerziell
Webseite
buguroo.com

Checkstyle

- Große Ähnlichkeiten zu PMD
- Benutzung von Regeln
- Erkennt duplizierten Code, Richtigkeit von JavaDoc Kommentaren, Zeichenabstände...
- Plug-ins für gängige Entwicklungsumgebungen vorhanden
- Hoher Konfigurationsgrad



Untersuchte Version
5.6 (18.09.2012)

Lizenz
LGPL

Webseite
checkstyle.sourceforge.net

Classycle

- erkennt zyklische Abhängigkeiten
- Erstellt HTML-Berichte

Untersuchte Version
1.4 (10.04.2011)

Lizenz
BSD License

Webseite
classycle.sourceforge.net

Clirr

- Überprüft Bibliotheken und Binär-Dateien auf Kompatibilität zu älteren Versionen

Untersuchte Version
0.6 (27.09.2005)

Lizenz
Apache Software License

Webseite
sourceforge.net/projects/clirr

Condenser

- Clone-Erkennung und automatische Korrektur
- Untersucht Java Quellcodeverzeichnisse

Untersuchte Version
1.06 (30.12.2002)

Lizenz
MIT License

Webseite
condenser.sourceforge.net

ConQAT

- Integrierte Visualisierung der Ergebnisse
- Aggregation von Qualitätsmetriken zur schnellen Übersicht der Qualität
- Clone-Erkennung
- Architektur-Konformitäts-Analyse
- Integration von anderen Codeanalysewerkzeugen wie z.B. FindBugs und PMD
- Erweiterbares Framework (Plug-in Architektur)
- Trendanalyse zur Messung der Qualität über die Zeit
- Kostenpflichtiger Support verfügbar
- Tutorials und ausführliche Dokumentation verfügbar

Untersuchte Version
2011.9 (30.09.2011)

Lizenz
Apache License 2.0

Webseite
conqat.org

Coverity SAVE

- Datenflussanalyse
- Erkennt Sicherheitslücken
- Fehlererkennung
- Parellele Codeanalyse
- Inkrementelle Analyse, d.h. nur geänderter Code wird analysiert

Untersuchte Version
unbekannt

Lizenz
kommerziell

Webseite
coverity.com

Dependecy Finder

- Erkennt Abhängigkeiten
- Objektorientierte Metriken

Untersuchte Version
1.2.1 (29.11.2010)

Lizenz
BSD License

Webseite
sourceforge.net/projects/depfind

Dependometer

- Unterstützung für: Java, C++ und C#
- Erzeugt einen ausführlichen HTML Report
- Erkennt Zyklen
- Erkennt Verstöße gegen die Architektur
- Errechnet Metriken

Untersuchte Version
1.2.5 (28.02.2011)

Lizenz
VPL 1.0.2

Webseite
source.valtech.com

devKing add-on for checKing QA

- Plug-in für die Webapplikation checkKing QA
- Unterstützt Java, JSP, JavaScript, HTML and XML
- Erstellt Berichte
- Source Code wird auf die Einhaltung einer Vielzahl von Regeln überprüft
- Automatische Fehler Präventionsmaßnahmen

Untersuchte Version
unbekannt

Lizenz
kommerziell

Webseite
optimyth.com

DevPartner

- Qualität und Komplexität des Codes
- Erkennung von Speicherlecks
- Thread-Analyse und Erkennung von Dead-Locks
- Code-Überdeckung
- Unterstützt mehrere Sprachen

Untersuchte Version
unbekannt

Lizenz
kommerziell

Webseite
microfocus.com

DMS Software Reengineering Toolkit

- Analysiert direkt den Quellcode
- Komplette Refaktoringkette vom Codeparsen zur Qualitätsanalyse über Restrukturierung bis hin zur automatischen Codegenerierung und Portierung
- Generischer Aufbau in Modulen für leichte Erweiterbarkeit
- Module müssen einzeln zusammengestellt und gekauft werden

Untersuchte Version
unbekannt

Lizenz
kommerziell

Webseite
www.semdesigns.com

DoctorJ - Java analyzer

- ▷ Untersucht *.java Dateien
- ▷ Erkennt Rechtschreibfehler und andere häufige Fehler

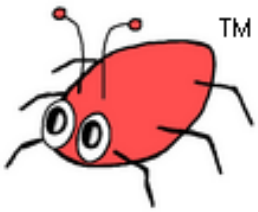
Untersuchte Version
5.1.2 (22.08.2006)

Lizenz
LGPL v2

Webseite
sourceforge.net/projects/doctorj

FindBugs

- ▷ Analysiert Bytecode
- ▷ Fehlererkennung
- ▷ Auch als Plug-in für Eclipse



Untersuchte Version
2.0.1 (12.07.2012)

Lizenz
LGPL

Webseite
findbugs.sourceforge.net

HP Fortify Static Code Analyzer (SCA)

- ▷ Erkennt Sicherheitslücken in Software
- ▷ Schneller Scan des Source Codes

Untersuchte Version
unbekannt

Lizenz
kommerziell

Webseite
fortify.com

Imagix 4d

- für C,C++ und Java Quellcode
- Erzeugt unterschiedlichste Diagramme auf verschiedenen Abstraktionsebenen und hebt Beziehungen/Abhängigkeiten vor
- Findet 20 verschiedene potentielle Fehlerquellen im Code
- Erstellt Codemetriken und analysiert Flussdiagramme
- Unterstützt auch das Erstellen von Dokumentation

Untersuchte Version
7.3.2

Lizenz
kommerziell (Testversion erhältlich)

Webseite
imagix.com

JarAnalyzer

- Analysiert jar-Dateien
- Erkennt Abhängigkeiten zwischen zwei oder mehreren jar-Dateien
- Graphische Ausgabe der Ergebnisse

Untersuchte Version
1.2 (31.07.2006)

Lizenz
BSD Licence

Webseite
kirkk.com

JCSC

- Erkennt Abweichungen vom Coding-Standard

Untersuchte Version
0.98.1 (07.07.2005)

Lizenz
GPL

Webseite
jcsc.sourceforge.net

JDepend

- ▷ Untersucht Java Quellcodeverzeichnisse
- ▷ Erstellt Metriken über Klassen/Interfaces/Packages/... und ihre Abhängigkeiten untereinander

Untersuchte Version
2.9.1 (xx.xx.2010)

Lizenz
BSD License

Webseite
clarkware.com

JLint

- ▷ Untersucht Java Bytecode
- ▷ Durchführen einer Datenflussanalyse und Erstellen eines Lock-Graphen
- ▷ Erkennen von Fehlern
- ▷ Erkennen von Synchronisationsproblemen und Widersprüchen

Untersuchte Version
3.1.2 (11.01.2011)

Lizenz
GPL

Webseite
jlint.sourceforge.net

jTest

- ▷ Datenflussanalyse
- ▷ Berechnen von Code-Metriken
- ▷ Untersucht den Source-Code

Untersuchte Version
unbekannt

Lizenz
kommerziell

Webseite
parasoft.com

Kalistick

- ▷ Cloud-basierte Anwendung
- ▷ Aufzeichnung von Test-'footprints'

Untersuchte Version
unbekannt

Lizenz
kommerziell

Webseite
kalistick.com

Klocwork Insight

- Erkennt Sicherheitslücken im Code
- Graphische Darstellung der Softwarearchitektur
- Unterstützung für C/C++, Java und C#

Untersuchte Version
unbekannt

Lizenz
kommerziell

Webseite
klocwork.com

LDRA Testbed

- Erkennt Abweichungen vom Coding-Standard

Untersuchte Version
unbekannt

Lizenz
kommerziell

Webseite
ldra.com

Macker

- Erkennt Abweichungen von Architektur-Regeln

Untersuchte Version
0.4.2 (03.11.2003)

Lizenz
GPL

Webseite
sourceforge.net/projects/macker

Moose

- Datenanalyseprogramm
- Analysiert Java, C++, XML, ...
- Erstellt Datenmodelle und kann diese analysieren
- Berechnet Metriken
- Erweiterbar und konfigurierbar

Untersuchte Version
4.6 (xx.02.2012)

Lizenz
BSD License und MIT License

Webseite
moosetechnology.org

PMD

- Unterstützt Java, C, C++ und PHP
- Plug-ins für alle gängigen Entwicklungsumgebungen (eclipse, jDeveloper, NetBeans...)
- Benutzung von statischen Regeln
- Erkennung von möglichen Bugs (leere try/catch blöcke), totem code und unnötigen Schleifen
- Neue Regeln lassen sich relativ leicht schreiben
- Zur Erkennung von duplizierten Code lässt sich die Erweiterung CPD benutzen

Untersuchte Version
5.0.0 (01.05.2012)

Lizenz
BSD-style License

Webseite
/pmd.sourceforge.net

ProjectCodeMeter

- Berechnet Code-Metriken
- Berechnet Teamproduktivität und Kosten
- Untersucht Code auf Einhaltung von Coding-Standards
- Vergleicht berechnete Statistiken mit Marktdurchschnittswerten

Untersuchte Version
1.23

Lizenz
kommerziell (Testversion erhältlich)

Webseite
projectcodemeter.com

QJ-Pro

- Erkennt Abweichungen vom Coding-Standard
- Fehlererkennung

Untersuchte Version
2.2.0 (22.03.2005)

Lizenz
GPL

Webseite
qjpro.sourceforge.net/

ResourceMiner

- Grafische Visualisierung der Code-Struktur als Abhängigkeitsbaum

Untersuchte Version
unbekannt

Lizenz
kommerziell

Webseite
resourceminer.nu

Rational AppScan Source Edition

- Erkennt Sicherheitslücken im Code
- Datenflussanalyse
- Spezialisiert auf Code-Sicherheit

Untersuchte Version
unbekannt

Lizenz
kommerziell

Webseite
ibm.com

Sonar

- Untersucht die Software-Architektur
- Clone-Erkennung
- Überprüfung der Komplexität
- Erkennt potentielle Fehler
- Überprüft die Einhaltung von Richtlinien
- Graphische Berichtsausgabe über eine Website
- Viele Erweiterungen vorhanden
- Integration von anderen Codeanalysewerkzeugen wie z.B. PMD, CheckStyle oder FindBugs

Untersuchte Version
3.3 (24.10.2012)

Lizenz
LGPL v3.0

Webseite
sonar.codehaus.org

SonarGraph (SonarJ)

Untersuchte Version
6.0 (01.07.2010)

Lizenz
kommerziell

Webseite
hello2morrow.com

- ▷ Analysiert kompilierte Klassen und Quellcode
- ▷ Graphische Strukturausgabe des zu untersuchenden System
- ▷ Vergleich zwischen Model und Code möglich: zeigt Unterschiede/Fehler auf
- ▷ Plug-ins für eclipse und intelliJ verfügbar
- ▷ Zwei Versionen verfügbar: Architektur und Qualität
- ▷ Weitreichende Statistiken

Soot

Untersuchte Version
2.5.0 (22.01.2012)

Lizenz
LGPL

Webseite
sable.mcgill.ca

- ▷ Generiert vier verschiedene Zwischencodes für die einfachere Analyse
- ▷ Aufruf-Graph-Analyse

Sotoarc/Sotograph

Untersuchte Version
unbekannt

Lizenz
kommerziell

Webseite
hello2morrow.com

- ▷ Grafische Visualisierung der Code-Struktur als hierarchischen Baum
- ▷ Möglichkeit die Software-Architektur grafisch zu spezifizieren
- ▷ Erkennt Abweichungen von der Architektur
- ▷ Auch als Plug-in für Eclipse

TattleTale

- Scannt jar-Dateien
- erkennt Abhängigkeiten
- Bericht wird als HTML-Datei exportiert
- Erkennt/Entfernt 'black-listed' APIs , Klassen Standorte...

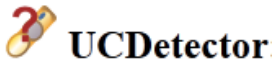
Untersuchte Version
1.2.0 Beta2 (17.02.2012)

Lizenz
LGPL v2.1

Webseite
jboss.org

UC Detector

- Erkennt toten Code
- Erkennt unnötige Public-Deklarationen
- Auch als Eclipse-Plug-in



Untersuchte Version
1.10.0 (05.04.2012)

Lizenz
Eclipse Public License v1.0

Webseite
ucdetector.org

Understand

- Komplette IDE mit Code Editor
- Kann verschiedenste Codemetriken erzeugen: Codezeilen, Code-Coupling, ...
- Codeverifikation gegen 'Coding-Standards' und Programmierrichtlinien (z.B. Effective C++) oder eigen definierte Standards
- Kann Klassen/Objekt Abhängigkeiten aufzeigen
- Für Java, Ada, C/C++, C# Fortran, JOVIAL, Pascal, PL/M, VHDL, Cobol, Web-Sprachen, Phython (bei Java jedoch nicht alles. z.b. Generics nicht)
- Erzeugt unterschiedliche Graphen (z.b. UML)

Untersuchte Version
3.0.638 (02.11.2012)

Lizenz
kommerziell (Testversion erhältlich)

Webseite
scitools.com

Veracode Static Analysis

- Unterstützt C, C++, Java, .NET bytecode, PHP...
- Untersuchung des Byte Codes
- Erkennung von Gefährdungen durch Libraries, APIs, Compileroptimierungen und 3rd party Komponenten
- Erkennung von Sicherheitslücken

Untersuchte Version
unbekannt

Lizenz
kommerziell

Webseite
veracode.com

Yasca

- Unterstützung von: Java, C/C++, HTML, JavaScript, ASP, ColdFusion, PHP, COBOL und weitere Sprachen
- Erkennt Sicherheitslücken
- Fehlererkennung
- Erkennung von Performanz-Problemen
- Überprüfung des Codes auf 'best practices'
- Integration von anderen Codeanalysewerkzeuge wie z.B. FindBugs, PMD und JLint

Untersuchte Version
2.21 (01.11.2010)

Lizenz
BSD Licence, GPL v2, LGPL v2

Webseite
sourceforge.net/projects/yasca

4 Nutzungsszenario

In unserer Fachstudie gibt es keinen externen Kunden. Kunde und Betreuer sind hier äquivalent. Es wird ein Programm gesucht, welches aus der großen Menge der statischen Codeanalyswerkzeuge für Java heraussticht. Auf Basis der Empfehlung dieser Fachstudie, wird die SE2 Abteilung des Instituts für Softwaretechnologie der Universität Stuttgart, sich auf ein statisches Codeanalysewerkzeug festlegen. Es ist geplant an dem empfohlenen Codeanalysewerkzeug sukzessive Erweiterungen durchzuführen, um eine möglichst hohe Anpassung zu erreichen, um es für zukünftige Aufgaben optimal einzusetzen.

5 Bewertungskriterien

In diesem Kapitel werden die Kriterien beschrieben, welche bei der Bewertung der einzelnen Codeanalysewerkzeuge in Betracht gezogen wurden. Für jedes Kriterium wurde dabei gemeinsam mit dem Betreuer eine Gewichtung erarbeitet, um die untersuchten Codenalysewerkzeuge genau nach den Anforderungen qualifizieren zu können. Für eine Erklärung der hier verwendeten Skalen siehe Kapitel 6.2.

5.1 K.O. Kriterien

Die folgenden Kriterien sind unbedingt notwendig und führen, wenn nicht erfüllt, zum bedingungslosen Ausschluss eines Codenalysewerkzeuges von den weiteren Untersuchungen.

5.1.1 Erweiterbarkeit

Beschreibung Erweiterbarkeit beschreibt die Möglichkeit ein Codeanalysewerkzeug den eigenen Anforderungen entsprechend anzupassen und neu benötigte Funktionalität hinzuzufügen.

Erweiterbarkeit kann unterschiedlich stark gegeben sein und geht deshalb mit anderen Bewertungskriterien wie Dokumentation oder Projektaktivität einher.

Dieses Kriterium distanziert sich jedoch von der Abwägung über die Ausprägung der Erweiterbarkeit und beschäftigt sich nur mit der Tatsache, ob Erweiterbarkeit überhaupt oder gar nicht gegeben ist.

Ist beispielsweise der Quellcode eines zu untersuchenden Codeanalysewerkzeuges erhältlich oder lassen sich Plug-ins in dieses einbinden, so ist die Erweiterbarkeit im Sinne dieses Kriteriums erfüllt, egal wie schwer der Quellcode zu verstehen oder wie mächtig die Plug-in-Schnittstelle ist.

Gewichtung Eine Erweiterbarkeit im Sinne dieses Kriteriums ist zwingend für ein weiteres in Betracht ziehen des untersuchten Codenalysewerkzeuges notwendig.

Skala $\oplus$ / $\ominus$

5.1.2 Updates

Beschreibung Codeanalysewerkzeuge werden häufig von kommerziell orientierten Firmen oder aber auch von Programmierern in ihrer Freizeit entwickelt. Gerade bei letzteren passiert es nicht selten, dass die Entwicklung anfangs extrem voranschreitet, mit der Zeit jedoch stark zurückgeht oder gar völlig aufgegeben wird. Aber auch kommerzielle Projekte ereilt oft das Schicksal, dass diese nicht mehr lukrativ genug sind oder aus anderen Gründen nicht weiterentwickelt werden.

Die Aktivität im Bezug auf Updates eines Softwareprojekts beschreibt, ob dieses noch weiterentwickelt wird und wenn ja, in welchem Maße. Bei einem aktiven Projekt muss davon ausgegangen werden können, dass neue Versionen mit Fehlerbehebungen oder Erweiterungen veröffentlicht werden.

Gewichtung Codenalysewerkzeuge, bei welchen die letzte veröffentlichte Version länger als sechs Monate zurückliegt, werden von der weiteren Bewertung ausgeschlossen.

Skala $\oplus$ / $\ominus$

5.1.3 Plattform

Beschreibung Codeanalysewerkzeuge kommen auf Computern mit unterschiedlichsten Architekturen und Betriebssystemen zum Einsatz. Jedoch ist nicht jedes Codeanalysewerkzeug auf jeder dieser Plattformen lauffähig. Dieses Kriterium beschäftigt sich mit den, vom zu untersuchenden Codeanalysewerkzeugen unterstützten, Plattformen.

Gewichtung Die Codeanalysewerkzeuge müssen mindestens auf Microsoft Windows 7 lauffähig sein und werden, wenn dies nicht erfüllt wird, von der weiteren Bewertung ausgeschlossen.

Skala $\oplus$ / $\ominus$

5.1.4 Funktionsumfang

Beschreibung Gerade im Bereich der statischen Codeanalyse ist der Funktionsumfang der Werkzeugen breit gefächert. Es gibt Codeanalysewerkzeuge, welche nur auf eine ganz besondere Art von Analyse spezialisiert sind und andere, welche gleich ein ganzes Arsenal an verschiedener Funktionalität mit sich bringen. Bei diesem Kriterium geht es darum die unterschiedlichen Funktionalitäten eines Codeanalysewerkzeuges herauszufinden und zu zählen.

Gewichtung Da die analysierten Codeanalysewerkzeuge mit neuer Funktionalität erweitert werden sollen, reicht eine Spezialisierung auf einzelne Funktionalitäten nicht aus. Ein Codeanalysewerkzeug, das nicht mindesten drei unterschiedliche Funktionalitäten bietet, wird von der weiteren Bewertung ausgeschlossen.

Skala $\oplus$ / $\ominus$

5.2 Nicht relevante Kriterien

5.2.1 Art/Architektur

Beschreibung Codeanalysewerkzeuge gibt es sowohl als alleinstehende Programme, als auch als Plug-ins für gängige Entwicklungsumgebungen wie z. B. Eclipse.

Manchmal sind auch Kombinationen möglich. Das Codeanalysewerkzeug kann alleinstehend als Konsolenanwendung verwendet werden, aber auch in einer IDE eingebunden sein, um so besseres visuelles Feedback und eine höhere Produktivität zu erzielen.

Dieses Kriterium soll die Codeanalysewerkzeuge in eine dieser Klassen einteilen.

Gewichtung Dieses Kriterium ist für den Betreuer von **keiner** Bedeutung.

5.3 Weitere Kriterien

5.3.1 Community/Support

Beschreibung Ein weiterer Teil des Aktivitätskriteriums beschreibt die Zugänglichkeit zu Informationen. Bei einem aktiven Projekt im Bezug auf die Community ist es möglich Informationen über die Software zu erhalten, um diese zu verwenden oder zu erweitern. Dieser Teil des Kriteriums setzt nicht unbedingt eine aktive Weiterentwicklung der Software voraus, sondern bezieht sich auf eine aktive Community und angebotene Supportleistungen.

Gewichtung Die Aktivität im Bezug auf die Community ist von **hoher** Wichtigkeit und sollte insbesondere aus Sicht der Erweiterbarkeitsmöglichkeiten, die sich aus einer hohen Aktivität ergeben, betrachtet werden.

Skala $\oplus$ / $\odot$ / $\ominus$

5.3.2 Dokumentation

Beschreibung Dokumentation beinhaltet alle Dokumente, die Auskunft über das zu untersuchende Codeanalysewerkzeug (nicht jedoch der Bedienung dieses) liefern. Hierbei kann es sich um externe Dokumente wie eine Spezifikation, Entwurfsdokumente oder API-Beschreibungen handeln, aber auch um ausgeprägte Quellcodekommentare.

Nicht gemeint sind Dokumente wie zum Beispiel das Handbuch, das zwar üblicherweise auch der Dokumentation zugeordnet wird, nach der Definition der hier behandelten Kriterien, jedoch zur Benutzerfreundlichkeit beiträgt und deshalb zu diesem Kriterium zugeordnet wird.

Ein wichtiger Punkt dieses Kriteriums ist nicht nur das Vorhandensein einer Dokumentation, sondern auch die Qualität dieser. Hierbei ist vor allem zu beachten, ob die Dokumentation auch den aktuellen Stand der Software beschreibt und nicht bereits veraltet ist. Dokumentation, die nicht mehr die momentane Umsetzung zeigt, ist nicht von Nutzen und darf nicht positiv in die Bewertung einfließen.

Gewichtung Die Dokumentation ist von **hoher** Bedeutung. Auch diese sollte insbesondere dahingehend untersucht werden, dass der Quellcode (sofern vorhanden) leicht verstanden werden kann oder dass Schnittstellen für Plug-ins oder andere Anbindungen zur Erweiterbarkeit gut beschrieben sind.

Skala $\oplus$ / $\odot$ / $\ominus$

5.3.3 Benutzerfreundlichkeit

Beschreibung Benutzerfreundlichkeit beschreibt wie einfach ein Benutzer das Codeanalysewerkzeug verwenden kann und wie wohl er sich dabei fühlt. Dies beinhaltet einerseits wie gut er sich intuitiv in der Umgebung des Codeanalysewerkzeuges oder dessen Eingabemöglichkeiten zurecht findet, aber auch wie gut die Funktionalität und dessen Verwendung, z. B. in einem externen Handbuch, beschrieben ist.

Auch die Aufbereitung der Ergebnisse und Verständlichkeit dieser für den Benutzer fließt in dieses Kriterium ein.

Gewichtung Die Benutzerfreundlichkeit ist von **hoher** Bedeutung.

Skala $\oplus$ / $\odot$ / $\ominus$

5.3.4 Blacklisting

Beschreibung Blacklisting beschreibt die Funktionalität einmal gefundene Fehler markieren zu können, so dass diese bei einem weiteren Durchlauf des Codeanalysewerkzeuges nicht mehr bemängelt werden.

Hierdurch ist es möglich Befunde, welche man nicht beheben möchte aus den Resultaten zu streichen, um eine bessere Übersicht mit einem Fokus auf die tatsächlich kritischen Mängel zu gewährleisten. Ein Codeanalysewerkzeug welches diese Funktionalität bietet, wird jedoch einem ohne diese Funktionalität vorgezogen.

Gewichtung Das Vorhandensein von Blacklisting ist von **geringer** Bedeutung.

Skala $\odot$ / $\ominus$

5.3.5 Kosten

Beschreibung Handelt es sich nicht um ein Open Source Codeanalysewerkzeug, so ist dies meist damit verbunden, dass das Codeanalysewerkzeug käuflich erworben werden muss. Die dabei entstehenden Kosten sollen in diesem Kriterium bewertet werden. Zu beachten ist hier auch, ob Testversionen, kostenlose Versionen für die nicht kommerzielle Nutzung oder andere Vergünstigungen z.B. durch eine EDU Version, erhältlich sind.

Ebenso kann es Fälle geben, in denen bestimmte Leistungen, wie beispielsweise der Support oder angebotene Plug-ins, kostenpflichtig sind. Solche Faktoren fließen ebenfalls in dieses Kriterium mit ein.

Gewichtung Die Kosten eines Codeanalysewerkzeuges werden **mittel** gewichtet.

Skala $\oplus$ / $\odot$ / $\ominus$

5.3.6 Lizenz

Beschreibung Es gibt verschiedene Lizenzierungen für Codeanalysewerkzeuge. Diese regeln oft (gerade bei Open Source Werkzeugen) wie man mit Änderungen oder Erweiterungen der Software umgehen soll. Hierbei wird festgelegt, ob man diese veröffentlichen darf, wenn ja in welchem Maße und mit welchen Einschränkungen. Das Ziel dieses Kriteriums ist es zu bewerten, ob die Lizenzierung ein Problem bei der Erweiterungen des Codeanalysewerkzeuges darstellt oder nicht.

Gewichtung Die Lizenz wird nur **gering** gewichtet.

Skala $\oplus$ / $\odot$ / $\ominus$

5.3.7 Performance

Beschreibung Statische Codeanalyse beinhaltet oft aufwendige Prozesse, die eine hohe Komplexität und damit lange Laufzeiten mit sich bringen. Außerdem sind die Codebasen welche durch solche Werkzeuge untersucht werden, nicht selten sehr groß und erschweren dadurch dieses Problem. Im Zusammenhang mit Continuous Integration, in der jede Nacht, oder gar bei jedem Commit von Änderungen in ein Quellcodeverwaltungsprogramm, Tests und statische Codeanalysen ablaufen sollen, ist die effiziente Implementierung und damit einhergehende Performance oft ein Punkt, der in Betracht gezogen werden muss.

Gewichtung Die Performance ist von **geringer** Bedeutung. Das Codeanalysewerkzeug wird vor allem zu wissenschaftlichen Zwecken eingesetzt, bei welchen die Laufzeiten nur eine untergeordnete Rolle spielen.

Skala $\oplus$ / $\odot$ / $\ominus$

5.3.8 Quellcode vs. Bytecode

Beschreibung Java-Quellcode wird zuerst in Bytecode kompiliert, welcher dann von einer Virtuellen Maschine interpretiert und ausgeführt wird. Codeanalysewerkzeuge können entweder den Quellcode oder den kompilierten Bytecode untersuchen um ihre Analysen durchzuführen. Codeanalysewerkzeuge, die Quellcode anstatt Bytecode analysieren, sind vorzuziehen.

Gewichtung Die Unterscheidung ob ein Codeanalysewerkzeug Quell- oder Bytecode analysiert ist von **geringer** Bedeutung.

Skala $\odot$ / $\ominus$

6 Evaluation

6.1 Voruntersuchung

Die gefundenen Codeanalysewerkzeuge werden einer Voruntersuchung unterzogen, um diejenigen auszusortieren, die gegen mindestens ein K.O.-Kriterien verstoßen. Dies dient in erster Linie dazu, die Teilnehmerzahl, die für eine genauere Evaluation in Frage kommen, zu verringern. Die gekürzte Liste enthält darum nur noch Codeanalysewerkzeuge, die alle K.O.-Kriterien erfüllen. Die Vorauswahl geschah mit Hilfe einer Matrix. Sie stellt dar, inwiefern die K.O.-Kriterien von den Codeanalysewerkzeuge erfüllt wurden. Wenn mindestens ein K.O.-Kriterium nicht erfüllt wird, wird bei dem betreffenden Codeanalysewerkzeuge auf eine weitere Untersuchung anderer K.O.-Kriterien verzichtet.

Erklärung der Legende

Erfüllt: $\oplus$	Alle Anforderungen des Kriteriums sind erfüllt.
Nicht erfüllt: $\ominus$	Mindestens eine Anforderung des Kriteriums ist nicht erfüllt.
Nicht überprüft: ?	Das Kriterium wurde im Rahmen der Fachstudie nicht überprüft.

Erklärung der Kriterien

Erweiterbarkeit	Das Codeanalysewerkzeug ist ohne größere Probleme erweiterbar
Updates	Das Codeanalysewerkzeug befindet sich in aktiver Entwicklung
Funktionsumfang	Das Codeanalysewerkzeug bietet mindestens drei verschiedene Funktionen
Plattform	Das Codeanalysewerkzeug läuft mindestens unter Microsoft Windows 7

Name	Erweiterbarkeit	Updates	Funktionsumfang	Plattform
AgileJ StructureViews	⊖	?	?	?
Bauhaus Suite	⊕	⊕	⊕	⊕
BugScout	⊖	?	?	?
Checkstyle	⊕	⊕	⊕	⊕
Classycle	⊕	⊕	⊖	?
Clirr	⊕	⊖	?	?
Condenser	⊕	⊖	?	?
ConQAT	⊕	⊕	⊕	⊕
Coverity SAVE	⊖	?	?	?
Dependency Finder	⊕	⊖	?	?
Dependometer	⊕	⊖	?	?
devKing add-on for checKing QA	⊖	?	?	?
DevPartner	⊖	?	?	?
DMS Software Reengineering	⊖	?	?	?
DoctorJ - Java analyzer	⊕	⊖	?	?
FindBugs	⊕	⊕	⊕	⊕
HP Fortify Static Code Analyzer (SCA)	⊖	?	?	?
Imagix 4d	⊖	?	?	?
JarAnalyzer	⊕	⊖	?	?
JCSC	⊕	⊖	?	?
JDepend	⊕	⊖	?	?
JLint	⊕	⊕	⊕	⊕
jTest	⊖	?	?	?
Kalistick	⊖	?	?	?
Klocwork Insight	⊖	?	?	?
LDRA Testbed	⊖	?	?	?

Name	Erweiterbarkeit	Updates	Funktionsumfang	Plattform
Macker	⊕	⊖	?	?
Moose	⊕	⊕	⊕	⊕
PMD	⊕	⊕	⊕	⊕
ProjectCodeMeter	⊖	?	?	?
QJ-Pro	⊕	⊖	?	?
ResourceMiner	⊖	?	?	?
Rational AppScan Source Edition	⊖	?	?	?
Sonar	⊕	⊕	⊕	⊕
SonarGraph	⊖	?	?	?
Soot	⊕	⊕	⊖	?
Sotoarc/Sotograph	⊖	?	?	?
TattleTale	⊕	⊕	⊖	?
UC Detector	⊕	⊕	⊖	?
Understand	⊕	⊕	⊕	⊕
Veracode Static Analysis	⊖	?	?	?
Yasca	⊕	⊖	?	?

Tabelle 6.1: Ergebnis der Voruntersuchung anhand der K.O.-Kriterien

6.2 Bewertungsschema

Vorgehen zur Bestimmung des Evaluationsergebnisses für ein Werkzeug

Nachdem alle Kriterien für ein Codeanalysewerkzeug bewertet wurden, werden die erreichten Punktzahlen mit ihren Gewichtung multipliziert und anschließend aufsummiert. Die sich daraus ergebende Gesamtpunktzahl wird auf eine Skala von 0 bis 100% normiert und das Codeanalysewerkzeug bekommt anhand der normierten Punktzahl eine Endnote zugewiesen.

Die normierte Punktzahl P_W für ein Codeanalysewerkzeug W is definiert durch

$$P_W = \frac{1}{P_{max}} \sum_i g_i p_i$$

Hierbei ist

i	ein Kriterium aus Kapitel 5.3,
g_i	die Gewichtung des Kriteriums i,
p_i	die Punktzahl des Kriteriums i anhand der Evaluationsergebnisse,
P_{max}	die maximal erreichbare Punktzahl.

Die Punktzahlen, Gewichtungen und Endnoten können mit Hilfe der nachfolgenden Tabellen ermittelt werden.

Punktzahlen

Für jedes Kriterium kann ein Codeanalysewerkzeug zwischen 0 und 2 Punkten erhalten. Dafür wird es auf der Stufe der Ordinalskala eingeordnet, die das Codeanalysewerkzeug für das entsprechende Kriterium erfüllt.

Ergebnis	Punktzahl	Bedeutung im Allgemeinen
⊕	2 Punkte	Das Kriterium wird vollständig erfüllt.
⊙	1 Punkt	Das Kriterium wird teilweise erfüllt.
⊖	0 Punkte	Das Kriterium wird nicht erfüllt.

Tabelle 6.2: Punktzahl für jeden Wert auf der Bewertungsskala

Gewichtungen

Alle möglichen Gewichtungen befinden sich auf der Ordinalskala $[0, 5]$. Die Gewichtung $g_i = 0$ repräsentiert die irrelevanten Kriterien, die für die Evaluation keine Rolle spielen. Die Gewichtung der einzelnen Kriterien erfolgte in Abstimmung mit Herr Ostberg.

Kriterium i	g_i
Community/Support	5
Dokumentation	5
Benutzerfreundlichkeit	4
Kosten	3
Lizenz	2
Blacklisting	1
Performance	1
Quellcode vs. Bytecode	1
Art/Architektur	0

Tabelle 6.3: Gewichtung der Bewertungskriterien

Endnoten

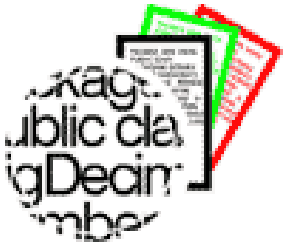
Den normierten Punktzahlen werden Endnoten zugewiesen.

P_W **[%]**	**Endnote**
100-96	1+
95-91	1
90-86	1-
85-81	2+
80-76	2
75-71	2-
70-66	3+
65-61	3
60-56	3-
55-51	4+
50-46	4
45-0	5

Tabelle 6.4: Endnote anhand der erreichten normierten Punktzahl

6.3 Werkzeuge

6.3.1 Checkstyle



Untersuchte Version
5.6 (18.09.2012)

Lizenz
GNU Lesser General Public License v2

Webseite
checkstyle.sourceforge.net

CheckStyle ist ein Kommandozeilenwerkzeug zur statischen Codeanalyse, das einst entwickelt wurde, um Codelayouts gegen gängige Standards zu überprüfen und auf Abweichungen hinzuweisen. Diese Funktionalität war auch namensgebend für das Codeanalysewerkzeug. Seit einer großen Architekturänderungen in Version 3.0 sind jedoch auch andere Tests verfügbar gemacht worden, wie zum Beispiel das Erkennen von Codeduplikaten, Klassendesignproblemen (Sichtbarkeit von Klassenmembern, Final-Deklarationen, Throw-Deklarationen, uvm.) oder fehlerbehafteten Ausdrücken. Dennoch liegt das Augenmerk der meisten Checks immer noch auf Dingen, die man hauptsächlich dem "Stil" von Quellcode zuschreiben würde. CheckStyle analysiert den Quellcode von Javaprogrammen.

Community/Support: Es existieren vier Mailinglisten für CheckStyle. Eine für offizielle Ankündigungen, eine für Commits in das Versionsverwaltungssystem, eine für Benutzerfragen und eine für Entwicklerfragen. Allgemein sind diese Mailinglisten nur sehr schwach in Benutzung und die meisten Einträge der letzten Zeit (besonders in der Entwicklermailingliste) sind von Diskussion unter den Entwicklern selbst.

Dokumentation: Die Dokumentation von CheckStyle ist in einem guten Zustand. Auf der Webseite ist eine Referenzliste mit allen Verfügbaren Checks vorhanden. Jeder einzelne Check wird dabei kurz beschrieben (oft mit einem Codebeispiel), enthält eine Liste mit den verfügbaren Parametern, eine Angabe über das entsprechende Paket und welchem Modul der Test zugeordnet ist. Auch der Quellcode ist gut dokumentiert. Eine API Referenz ist ebenfalls auf der Webseite verfügbar, welche alle Klassen und ihre Methoden erklärt. Es ist auch ein Tutorial vorhanden, dass beschreibt, welche Schritte man verfolgen muss, um neue Checks in CheckStyle einzufügen. Ein Entwurfsdokument, dass die gesamte Architektur von CheckStyle übersichtlich darstellt, ist jedoch nicht vorhanden.

Benutzerfreundlichkeit: CheckStyle wurde hauptsächlich entwickelt, um es direkt in den Buildprozess einzubetten, aber nicht um manuell Codechecks z.B. direkt während der Entwicklung einer Software vom Entwickler selbst durchzuführen. Deshalb besitzt CheckStyle selbst keine grafische Oberfläche, sondern ist ein reines Kommandozeilenwerkzeug, welches über eine XML-Datei konfiguriert wird.

```
<module name="Checker">
    <!--
        If you set the basedir property below, then all reported file
        names will be relative to the specified directory. See
        http://checkstyle.sourceforge.net/5.x/config.html#Checker

        <property name="basedir" value="${basedir}"/>
    -->

    <!-- Checks that a package-info.java file exists for each package.     -->
    <!-- See http://checkstyle.sf.net/config_javadoc.html#JavadocPackage -->
    <module name="JavadocPackage"/>

    <!-- Checks whether files end with a new line.                        -->
    <!-- See http://checkstyle.sf.net/config_misc.html#NewlineAtEndOfFile -->
    <module name="NewlineAtEndOfFile"/>

    <!-- Checks that property files contain the same keys.         -->
    <!-- See http://checkstyle.sf.net/config_misc.html#Translation -->
    <module name="Translation"/>

    <!-- Checks for Size Violations.                    -->
    <!-- See http://checkstyle.sf.net/config_sizes.html -->
    <module name="FileLength"/>

    <!-- Checks for whitespace                               -->
    <!-- See http://checkstyle.sf.net/config_whitespace.html -->
    <module name="FileTabCharacter"/>

    <!-- Miscellaneous other checks.                   -->
    <!-- See http://checkstyle.sf.net/config_misc.html -->
    <module name="RegexpSingleline">
       <property name="format" value="\s+$"/>
       <property name="minimum" value="0"/>
       <property name="maximum" value="0"/>
       <property name="message" value="Line has trailing spaces."/>
    </module>
```

Abbildung 6.1: Ausschnitt der Konfigurationsdatei für die Sun Coding Conventions

Die XML-Konfigurationsdatei ist simpel und übersichtlich aufgebaut. Die einzelnen Module, welche die durchzuführenden Checks repräsentieren, sind als XML-Knoten gestaltet, die wiederum Kindknoten für ihre Parameter besitzen (siehe Abbildung 6.1).

Der Konsolenaufruf von CheckStyle gestaltet sich sehr einfach. Es werden die zu verwendende Konfiguration und die zu untersuchende Datei(en) als Parameter übergeben. Zusätzlich kann eine Ausgabedatei und das Ausgabeformat spezifiziert werden.

Unabhängig von CheckStyle selbst wurde auch ein Eclipse-Plug-in entwickelt, dass es erlaubt CheckStyle einfach über Eclipse auszuführen und die Funde direkt in der IDE im Quellcode darzustellen (siehe Abbildung 6.2). Außerdem erlaubt das Eclipse-Plug-in die Konfigurationen für CheckStyle einfach über Dialoge in den Präferenzen eines Eclipse-Projekts zu modifizieren und den eigenen Anforderungen projektspezifisch anzupassen.

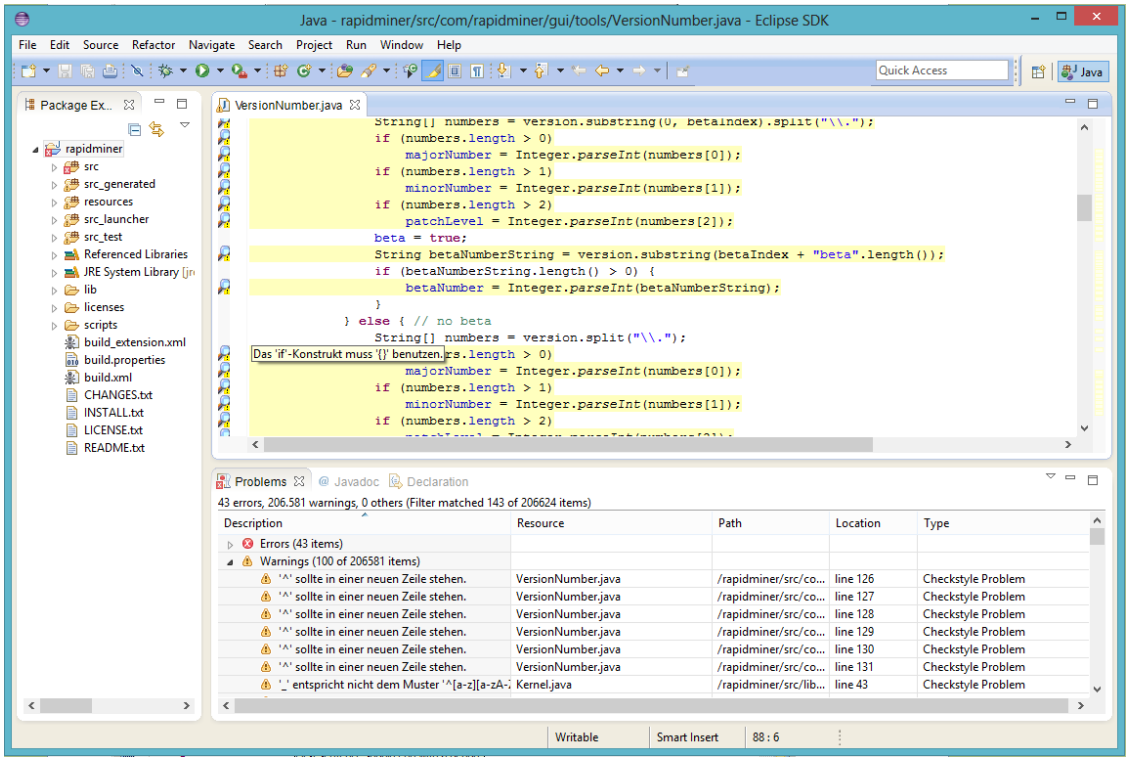


Abbildung 6.2: Screenshot nach der Ausführung von CheckStyle im Eclipse-Plug-in

Eine Integration für Ant-Buildtasks ist ebenfalls vorhanden.

Ausgabe: Nutzt man CheckStyle ohne das Eclipse-Plug-in, so sind zwei unterschiedliche Ausgabeformate möglich: eine Plain-Text-Ausgabe und eine XML-Ausgabe. Wird eine Ausgabedatei spezifiziert, so wird die Ausgabe dort hinein geschrieben. CheckStyle gibt in diesem Fall keinerlei Informationen auf die Standardausgabe aus. Ist keine Ausgabedatei spezifiziert, so wird die gesamte Ausgabe auf die Konsole geschrieben. Die XML Ausgabe ist für die maschinelle Verarbeitung geeignet. Die Befunde werden nach Dateien sortiert übersichtlich abgebildet (siehe Abbildung 6.3).

Kosten: CheckStyle ist Open Source und somit vollkommen kostenfrei.

```
<?xml version="1.0" encoding="UTF-8"?>
<checkstyle version="5.6">
<file name="C:\Users\alzi\Documents\Fachstudie\CheckStyle\testcode\rapidminer\.classpath">
<error line="3" column="1" severity="error" message="File contains tab characters (this is the first i
</file>
<file name="C:\Users\alzi\Documents\Fachstudie\CheckStyle\testcode\rapidminer\.project">
<error line="3" column="1" severity="error" message="File contains tab characters (this is the first i
</file>
<file name="C:\Users\alzi\Documents\Fachstudie\CheckStyle\testcode\rapidminer\build.properties">
</file>
<file name="C:\Users\alzi\Documents\Fachstudie\CheckStyle\testcode\rapidminer\build.xml">
<error line="2" column="1" severity="error" message="File contains tab characters (this is the first i
<error line="27" severity="error" message="Line has trailing spaces." source="com.puppycrawl.tools.che
<error line="38" severity="error" message="Line has trailing spaces." source="com.puppycrawl.tools.che
<error line="402" severity="error" message="Line has trailing spaces." source="com.puppycrawl.tools.ch
<error line="589" severity="error" message="Line has trailing spaces." source="com.puppycrawl.tools.ch
<error line="594" severity="error" message="Line has trailing spaces." source="com.puppycrawl.tools.ch
<error line="611" severity="error" message="Line has trailing spaces." source="com.puppycrawl.tools.ch
<error line="652" severity="error" message="Line has trailing spaces." source="com.puppycrawl.tools.ch
<error line="653" severity="error" message="Line has trailing spaces." source="com.puppycrawl.tools.ch
<error line="658" severity="error" message="Line has trailing spaces." source="com.puppycrawl.tools.ch
<error line="663" severity="error" message="Line has trailing spaces." source="com.puppycrawl.tools.ch
</file>
<file name="C:\Users\alzi\Documents\Fachstudie\CheckStyle\testcode\rapidminer\build_extension.xml">
<error line="0" severity="error" message="Datei endet nicht mit einem Zeilenumbruch." source="com.pupp
<error line="2" column="1" severity="error" message="File contains tab characters (this is the first i
<error line="294" severity="error" message="Line has trailing spaces." source="com.puppycrawl.tools.ch
<error line="353" severity="error" message="Line has trailing spaces." source="com.puppycrawl.tools.ch
</file>
```

Abbildung 6.3: XML Ausgabe von CheckStyle

6.3.2 ConQAT

Untersuchte Version
2011.9 (30.09.2011)

Lizenz
Apache License 2.0

Webseite
conqat.org

Die von der CQSE GmbH und TU München entwickelte Open Source Software ConQAT ist ein statisches Codeanalysewerkzeug, welches unter der Apache Lizenz 2.0 veröffentlicht ist. ConQAT unterstützt nativ verschiedene gängige Programmiersprachen, wie z.B. Java und C#. Die beinhalteten Funktionen reichen vom Erstellen von Qualitätsmetriken, über das Erkennen von Codeduplikaten bis zur Architektur-Konformitäts-Analyse. Die Software ist als Plug-in Framework konzipiert, was sie erweiterbar macht. Außerdem kann es mit anderen verbreiteten Codeanalysewerkzeugen wie z.B. PMD oder FindBugs integriert werden. Da es sich um Open Source Software handelt, fallen für Lizenzen keine Kosten an, allerdings ist der Support durch die CQSE GmbH kostenpflichtig. Da die Software in Java geschrieben ist und außerdem in Eclipse integriert ist, ist sie plattformunabhängig und es werden Downloads für Windows, Mac OS X und Linux angeboten. Es wurde bereits eine neue Version für das Ende des Jahres 2012 angekündigt und die neue ConQAT 2012.9 Engine wurde bereits auf der ICSM 2012 Konferenz als Pre-Release veröffentlicht. ConQAT führt die Analyse auf Quellcodebasis durch und unterstützt Blacklisting von Codeduplikaten.

Community/Support: Auf der ConQAT Webseite ist eine Liste mit Benutzern der Software aus dem akademischen Bereich veröffentlicht. Inwieweit es sich dabei um eine aktive Community handelt konnte ist nicht klar. Support wird von der CQSE GmbH geleistet und ist kostenpflichtig.

Dokumentation: Die ConQAT Webseite bietet Zugriff auf die umfangreiche Dokumentation, die neben dem Handbuch ConQAT Book und der API ConQATDoc auch zahlreiche Tutorials, Screenshots und Demos beinhaltet. Außerdem sind auch zahlreiche Publikationen über ConQAT und solche die ConQAT verwenden aufgeführt.

In dem Online-Dokument ConQATDoc werden alle Funktions-Bausteine (Blocks) beschrieben. Der Quellcode ist nach einem einheitlichen Styleguide geschrieben und ausreichend kommentiert. Eine Spezifikation oder ein Entwurfsdokument, das über ConQATDoc hinaus geht, gibt es jedoch nicht, was den Einstieg in die Weiterentwicklung der Software schwierig macht. Zumindest eine Beschreibung des Plug-ins-Systems wäre notwendig, um neue Plug-ins für ConQAT entwickeln zu können.

Benutzerfreundlichkeit: Das ConQAT Book bezieht sich zwar auf eine ältere Version der Software, weswegen einige Detailss veraltet sind, aber liefert trotzdem eine detaillierte Beschreibung aller Funktionen der Software. Die Schwäche des Handbuchs, die Verwendung der Software nur knapp zu beschreiben, wird durch die ausführlichen Tutorials auf der

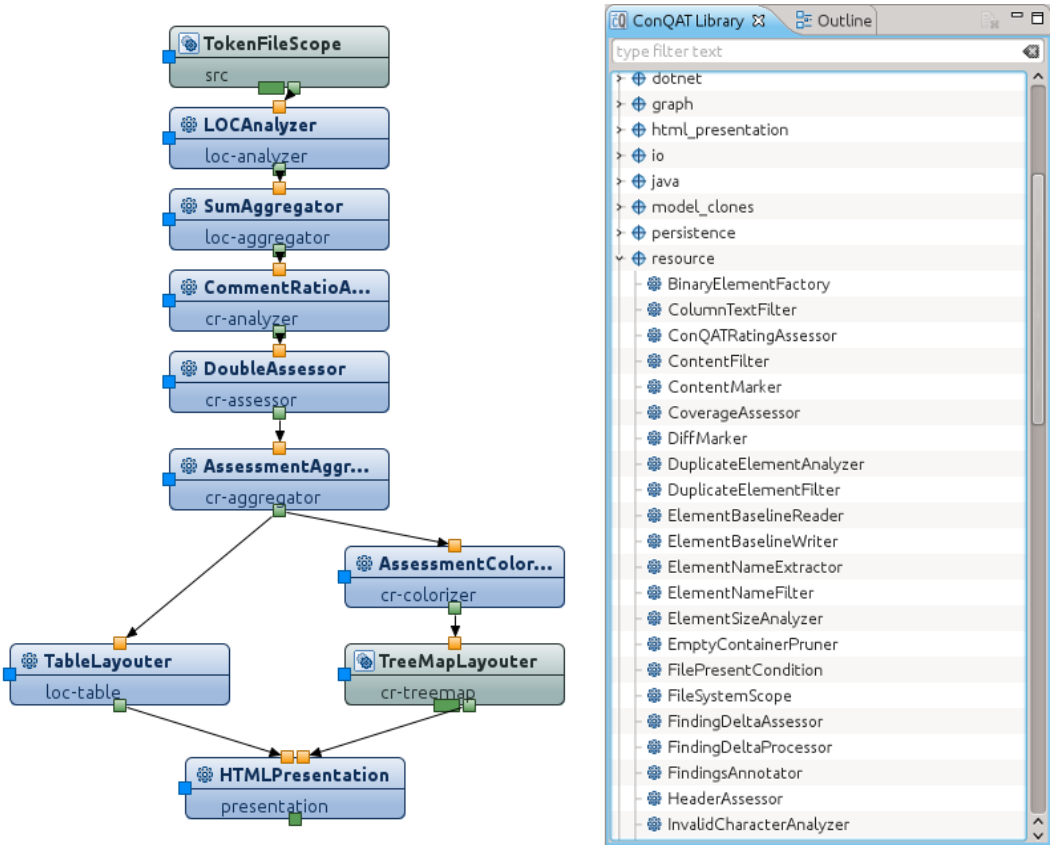


Abbildung 6.4: Der aus Blocks (Funktions-Bausteinen) aufgebaute Analyse-Graph und eine Liste der verfügbaren Blocks

Webseite ausgeglichen. Auch die bei der Installation beinhalteten Beispiele sind für den ersten Einstieg von großer Hilfe. Einzig die Installationsanleitung ist sowohl im Handbuch, als auch auf der Webseite äußerst knapp ausgefallen, weswegen eine korrekte Installation bei evtl. auftretenden Problemen einige Zeit in Anspruch nehmen kann.

Durch die Integration von ConQAT in Eclipse ist der Einstieg für jeden der mit Eclipse vertraut ist leicht. Trotzdem kann allein die Vielzahl an Funktionen den Benutzer überfordern. Auch das Erstellen einer Analyse-Konfiguration als Graph mit Hilfe von Funktions-Bausteinen (Blocks) ist durchaus komplex und erschwert den Einstieg (siehe Abbildung 6.4). Durch die gute Dokumentation der Bausteine, den mitgelieferten Beispiels-Konfigurationen sowie den ausführlichen Schritt-für-Schritt Tutorials, ist dies jedoch keine unüberwindbare Hürde. Wer einmal das Prinzip verstanden hat, kann schnell und effizient komplexe Analysen entwerfen und auch direkt die Weiterverarbeitung und grafische Aufbereitung der gewonnenen Daten spezifizieren. Die Bausteine decken nämlich nicht nur Analysemethoden, sondern auch Verarbeitung sowie Visualisierung der angefallenen Daten ab. Auch die zum Analyse-Graph gehörende Analyse-Konfiguration ist übersichtlich gestaltet und leicht zu konfigurieren (siehe Abbildung 6.5).

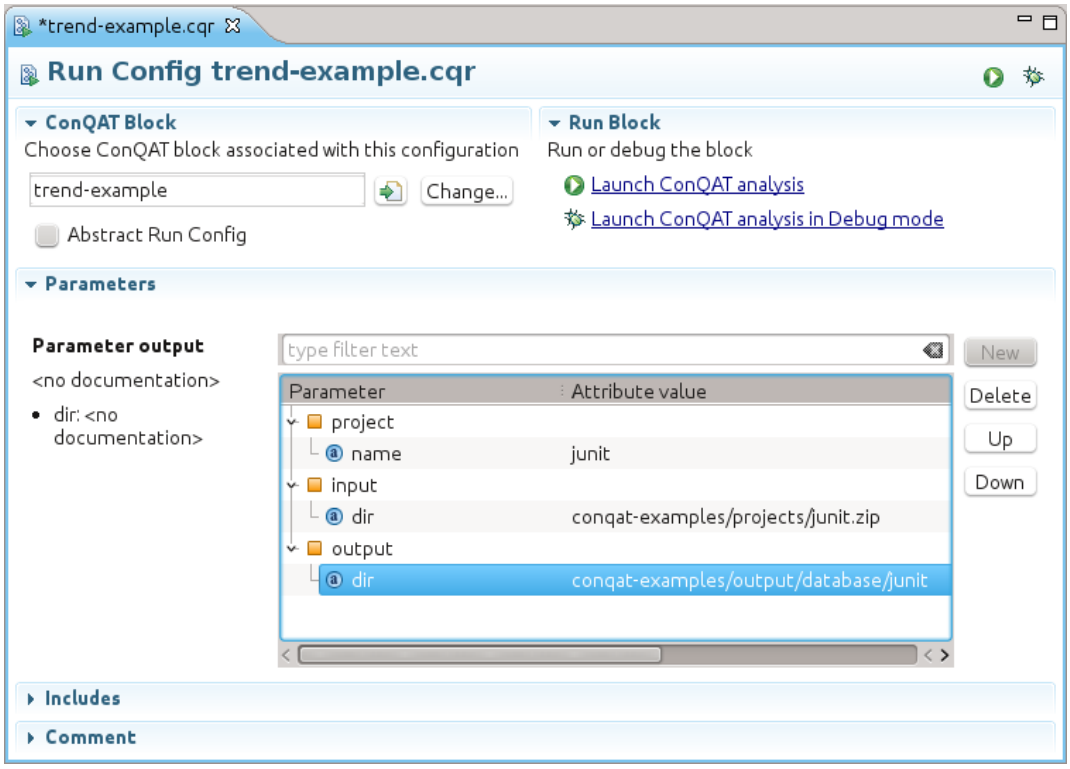


Abbildung 6.5: Grafische Ansicht der Analyse-Konfiguration in Eclipse mit Bearbeitungsmöglichkeiten

Ausgabe: Die Ausgabe der bei einer Analyse erhobenen Daten erfolgt als HTML. Die Ausgabe ist klar strukturiert und bietet eine interaktive Exploration der Daten (siehe Abbildung 6.6). Es gibt eine Übersicht der Analysedaten, Detailansichten der einzelnen Daten mit verschiedenen Visualisierungen (z. B. Tabellen und Tree Maps) sowie verschiedene Informationen zur verwendeten Analyse-Konfiguration und der Ausführung. Im Detail sind dies ein Log der Ausführung mit Informationen und evtl. aufgetretenen Fehlern, der verwendete Analyse-Graph, detaillierte Ausführungszeiten (aufgeteilt nach den einzelnen Analyseschritten) und eine Auflistung aller verwendeten Pakete der ConQAT Engine.

Kosten: Da es sich um Open Source Software handelt fallen keine Lizenzkosten an, allerdings ist der Support kostenpflichtig.

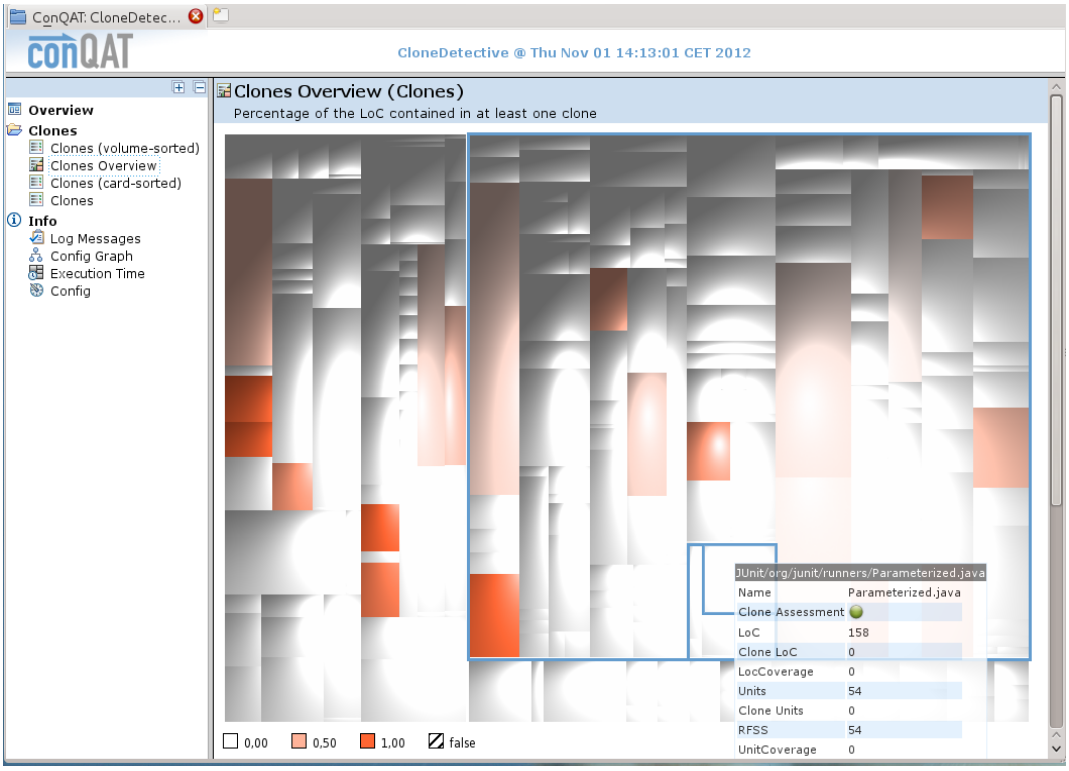
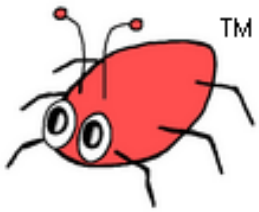


Abbildung 6.6: HTML-Ausgabe einer Clone-Detection-Analyse mit Visualisierung der gefundenen Codeduplikate als Tree Map

6.3.3 FindBugs



Untersuchte Version
2.0.1 (12.07.2012)

Lizenz
LGPL

Webseite
findbugs.sourceforge.net

FindBugs ist eines der bekanntesten Codeanalysewerkzeuge in der statischen Codeanalyse heutzutage. Das ursprünglich von der Universität von Maryland entwickelte Tools kann heute bereits mehr als eine Million registrierte Downloads verzeichnen und zeichnet sich durch seine sehr hohe Fehlererkennungsrate aus. FindBugs gehört zu der Sorte der statischen Codeanalysewerkzeuge, die den Bytecode untersuchen. Der Quellcode wird nur benötigt, wenn man mit der GUI von FindBugs arbeitet und farbliche Markierungen der Fehler-Standorte wünscht. FindBugs unterstützt Blacklisting.

Community/Support: Aufgrund der großen Bekanntheit des Codeanalysewerkzeuges, findet man im gesamten Netz Informationen über FindBugs. Der Quellcode ist aufgrund der Lizenz frei zugänglich und es gibt zahlreiche Tutorials in allen möglichen Sprachen. Das Benutzerhandbuch ist neben englischer Sprache, sogar ins Japanische übersetzt worden. Es existieren zwei Mailinglisten. Die eine ist lediglich für Ankündigungen seitens der Entwickler. Die andere Mailingliste ist für Diskussionen rund um FindBugs geeignet.

Dokumentation: Es existieren aktuelle API Beschreibungen. Sämtliche Dokumente werden gewartet und sind auf einem aktuellen Stand. Bei neuen Releases der Software dauert es im Normalfall stets etwas Zeit, bis die Dokumente nachgezogen werden. Dieser Aufgabe wird jedoch regelmäßig nachgekommen. Für jede neue Version des Tools seit Beginn der Entwicklung existiert zudem ein sehr ausführliches ChangeLog.

Benutzerfreundlichkeit: FindBugs kann mittels Kommandozeile, GUI oder auch als Plug-in für gängige Entwicklungsumgebungen benutzt werden. Die Benutzung des Codeanalysewerkzeuges ist auch ohne größere Einarbeitung möglich. Falls dennoch Probleme auftauchen, kann eines der vielen Tutorials betrachtet werden. Die Entwickler selbst stellen ausführliche Dokumente auf englische (und japanische) Sprache bereit, die einem weiterhelfen. Eine FAQ Sektion deckt die meisten auftauchenden Probleme. Die GUI lässt sich intuitiv bedienen. Gefundene Bugs werden in verschiedenen Kategorien übersichtlich aufgelistet. Der Benutzer hat selbst die Möglichkeit die Sortierung vorzunehmen. Jede Bug Art wird in der GUI genau auf englisch beschrieben. Der Standort wird genau angegeben. Falls auch der Quellcode der Software zum analysieren übergeben wurde, werden die betroffenen Stellen zudem farblich markiert. Auch ohne Quellcode wird die Stelle durch vier Textzeilen genau beschrieben. Blacklisting ist ebenfalls innerhalb der GUI bequem möglich.

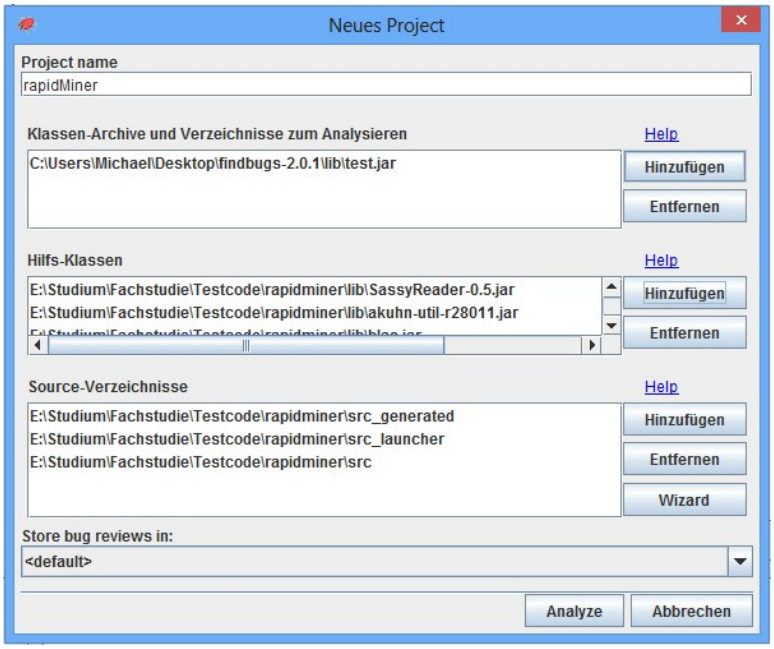


Abbildung 6.7: Eingabe-Konfiguration von FindBugs

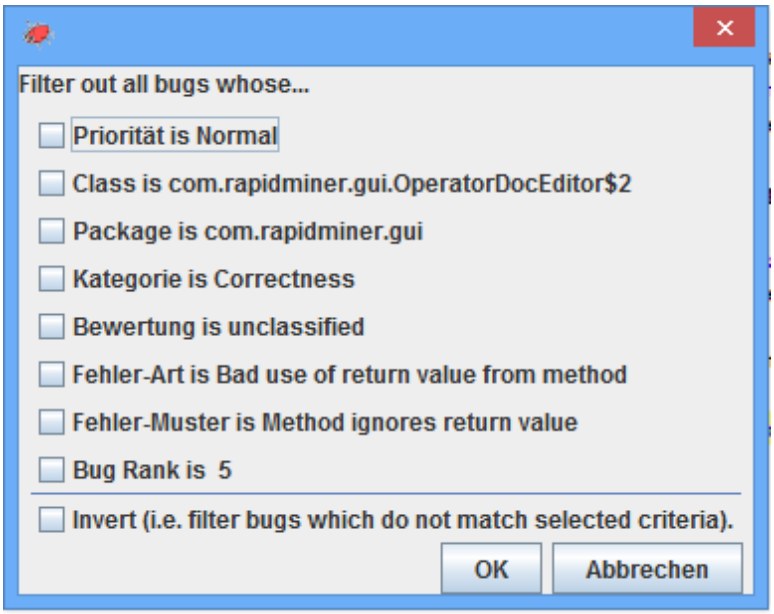


Abbildung 6.8: Blacklisting Möglichkeiten

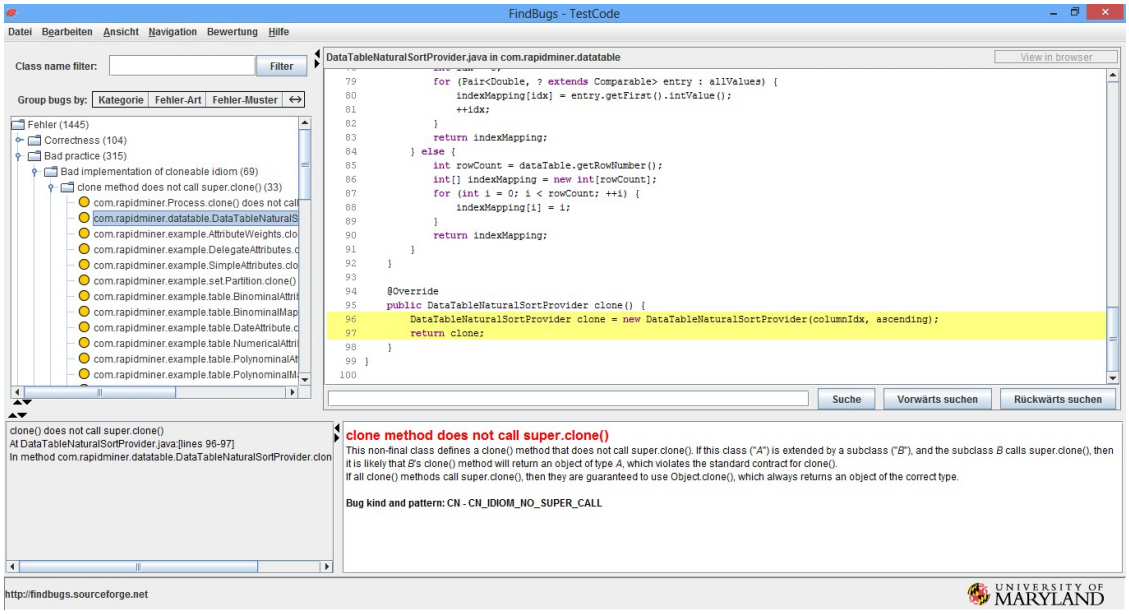


Abbildung 6.9: Grafische Ansicht der GUI mit den Resultaten nach der Analyse

Ausgabe: Die Ausgabe ist in HTML oder XML möglich. Die durch die Konsolenanwendung erzeugte XML-Dateien, lassen sich über die GUI einlesen. Diese XML-Dateien werden dort sehr übersichtlich dargestellt.

Kosten: FindBugs zählt zu den Open Source Codeanalysewerkzeugen und bietet keine Form von kommerziellem Support und/oder Ähnlichem an.

6.3.4 JLint

Untersuchte Version
3.1.2 (11.01.2011)

Lizenz
GPL

Webseite
jlint.sourceforge.net

Bei JLint handelt es sich um eine, unter der GPL veröffentlichte, Open Source Software, die seit 2002 auf SourceForge entwickelt wird. Da es sich bei GPL um eine copyleft-Lizenz handelt, dürfen auch Änderungen/Erweiterungen an der Software nur unter GPL/LGPL veröffentlicht werden. Dabei scheint es kein festes Entwicklerteam zu geben und die Aktivität im Projekt scheint stark zu schwanken. JLint untersucht Java Programme per statischer Code Analyse des Bytecodes auf Fehler, Synchronisationsprobleme und Widersprüche im Code. Es werden unter anderem Race-Conditions und Deadlocks erkannt. Dazu wird eine Datenflussanalyse durchgeführt und ein Lock-Graph erstellt. Blacklisting wird nicht unterstützt. Auch scheint die Architektur nicht speziell auf Erweiterbarkeit ausgelegt.

Community/Support: Obwohl es sich um ein Open Source Projekt handelt, das schon lange läuft, gibt es keine aktive Community. Auf der Projektseite auf SourceForge ist nur geringe Aktivität zu verzeichnen. Fragen im Forum scheinen nur sporadisch von einem der Entwickler beantwortet zu werden, im BugTracker gemeldete Fehler werden nicht bearbeitet und es gibt auch keine Mailingliste. Außerdem wird ausdrücklich darauf hingewiesen, dass es keinerlei Support von Seiten der Entwickler gibt, da es sich um ein nicht kommerzielles Projekt handelt.

Dokumentation: Es gibt keinerlei Software-Dokumente wie z. B. Spezifikation oder Entwurf die bei der Weiterentwicklung der Software hilfreich sein könnten. Da auch der Quellcode komplett unkommentiert ist, dürfte eine Erweiterung der Software sich als äußerst schwierig darstellen.

Benutzerfreundlichkeit: Bei diesem Punkt ist vorweg zu sagen, dass die Software nicht in der Praxis getestet werden konnte, da eine Installation nicht möglich war. Alle Einschätzungen beruhen deswegen alleine auf den Beschreibungen des Handbuchs und sonstigen Informationsquellen über die Software.

Dadurch dass der Quellcode selbst kompiliert werden muss, fällt der Einstieg schwer. Dies wird dadurch verstärkt, dass es zwar eine Anleitung dazu gibt, diese jedoch äußerst knapp gehalten ist und praktisch keinerlei Informationen zu evtl. auftretenden Fehler enthält. Da beim Kompilieren der Software Fehler auftraten, die sich zwar teilweise beheben ließen, aber eben nicht vollständig, konnte die Software nicht ausgeführt werden.

Als einziges Dokument gibt es ein Handbuch, dass die grundsätzliche Funktionsweise von JLint erläutert und alle Funktionen auflistet und beschreibt. Die möglichen Kommando-

zeilenparameter werde ausführlich beschrieben, so dass auch ein mit der Kommandozeile unerfahrener Benutzer sich schnell zurecht findet.

Bei JLint handelt es sich um ein reines Kommandozeilenprogramm. Durch die gute Erklärung der möglichen Parameter im Handbuch stellt dies jedoch kein Problem dar. Zuerst muss das zu analysierende Java-Programm mit einem beliebigen Java-Compiler in Bytecode übersetzt werden und anschließend als Parameter an JLint übergeben werden.

Ausgabe: Die Ausgabe erfolgt ausschließlich auf die Konsole. Über die Qualität der Ausgabe kann leider keine Aussage getroffen werden, da weder über das Format, noch über den Inhalt, im Handbuch eine Aussage getroffen wird.

Kosten: Da es sich um Open Source Software handelt, fallen keine Lizenzkosten an. Auch gibt es sonst keine kommerziellen Leistungen von Seiten der Entwickler.

6.3.5 Moose

Untersuchte Version
4.6 (xx.02.2012)

Lizenz
BSD License und MIT License

Webseite
moosetechnology.org

Die erste Version von Moose wurde bereits im Jahr 1996 veröffentlicht. Mittlerweile ist man bei der Version 4.7 angelangt. Da diese Version jedoch noch nicht final ist (und sich unter Windows nicht korrekt laden ließ), wurde die vorige Version getestet. Der Hauptnutzen der Software liegt darin, Entwicklern und Ingenieuren zu helfen, größere Softwareprojekte durch eine große Anzahl an Visualisierungen verständlicher zu machen. Dabei wird der Quellcode untersucht. Blacklisting wird seitens Moose nicht unterstützt. An dieser Stelle muss die auffallend schlechte Performance des Produktes erwähnt werden. Die meisten Schritte bei der Analyse der Software benötigen Einiges an Geduld. Auf dem benutzten Testrechner war es nicht möglich das Codeanalysewerkzeug effizient zu benutzen. Die Kernkomponente der Software befindet sich unter der BSD und MIT License. Dies trifft jedoch nicht auf alle benötigten Komponenten zu. Der empfohlene Java Parser befindet sich beispielsweise unter einer kommerziellen Lizenz.

Community/Support: Es existiert eine Mailingliste, die aktiv benutzt wird.

Dokumentation: Die Dokumente sind leider nicht auf dem aktuellsten Stand. Es existieren viele Informationen über die Hintergründe von Moose. Die Qualität der eigenen Dokumente ist jedoch nicht völlig zufrieden stellend und auch nicht aktuell. Es existiert ein extra Internetauftritt, der unter http://www.themoosebook.org erreichbar ist und sehr viele Informationen über die Geschichte und die Benutzung von Moose liefert. Ebenfalls liefert diese Dokumentation einiges an Hintergrundwissen, beispielsweise wie die benutzten Algorithmen arbeiten. Leider finden sich keine Informationen darüber, von wann der beschriebene Stand ist. Zum Teil sind deshalb Informationen falsch. Kommentare unter den jeweiligen Einträgen helfen einem bei Fehler meist weiter. Diese werden jedoch überwiegend von den Entwicklern nicht beachtet. Außerdem muss erwähnt werden, dass einige Bereiche lediglich ein “TODO“ als Inhalt anzeigen.

Benutzerfreundlichkeit: Moose punktet leider nicht in Sachen Benutzerfreundlichkeit. Dies fängt beim Start der Software an. Ohne die oben genannte Dokumentation ist die Benutzung unmöglich. Fehler in der Dokumentation werden bereits hier schon offenbart. Wenn der Start geglückt ist, kommt ein erstes Gefühl des Verlorenseins auf. Eine integrierte Hilfe-Funktion liefert einem außer Lizenz Informationen und ein Verweis auf den Internetauftritt keine Hilfe. Das einzige verfügbare Kontextmenü stammt von der virtuellen Maschine, in der Moose geladen wird. Nach dem Laden einer XML ähnlichen Datei, erhält man eine große Auflistung von Textzeilen. Die eigentliche Komplexität des Codeanalysewerkzeuges kommt spätestens hier negativ zur Geltung.

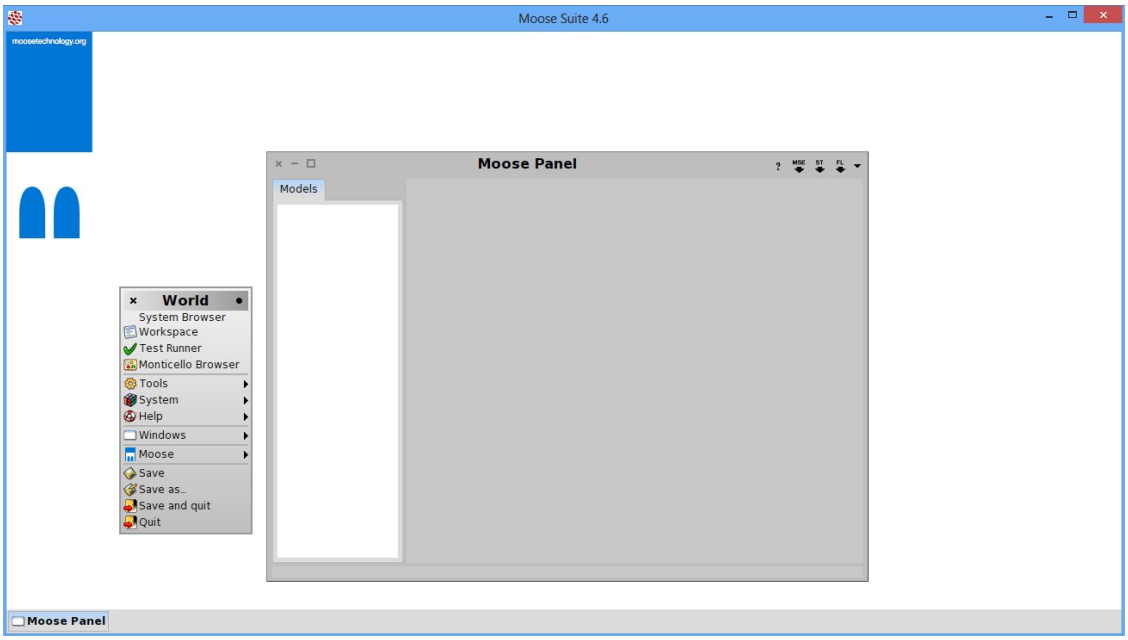


Abbildung 6.10: Startbildschirm der Anwendung mit dem VM Kontextmenü

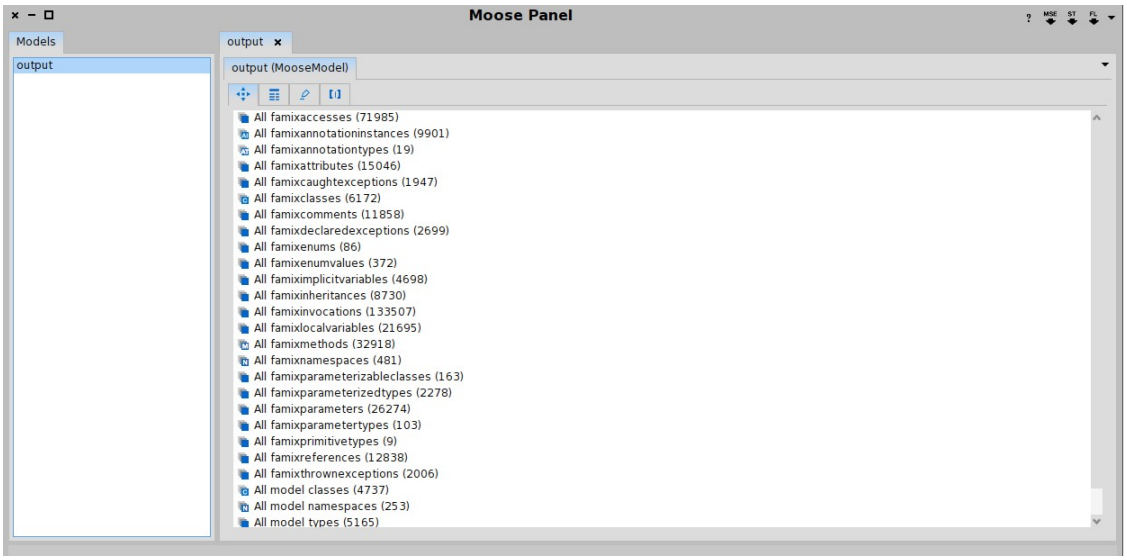


Abbildung 6.11: Anzeige der Resultate aus der generierten MSE Datei

Ausgabe: Die meisten Ergebnisse lassen sich visuell innerhalb der GUI darstellen. Diese liegen in einer sehr unübersichtlichen textuellen Form vor. Visualisierungen lassen sich durch längere Ladenzeiten über nicht wirklich intuitive Vorgänge erzeugen.

Kosten: Es existieren derzeit keine kommerziellen Angebote von Moose. Der Support ist ebenfalls kostenlos.

6.3.6 PMD

Untersuchte Version
5.0.0 (01.05.2012)

Lizenz
Eigene "BSD ähnlich"

Webseite
pmd.sourceforge.net

PMD ist ein statisches Codeanalysewerkzeug für Java, das aber auch JavaScript, XML, XSL und JSP unterstützt. Es arbeitet sehr ähnlich wie CheckStyle, unterstützt jedoch wesentlich mehr verschiedene Tests, die auch eher tatsächliche Fehler und nicht nur den “Codingstyle“ analysieren. Es sind weiterhin Tests (in PMD "Rules", im Folgenden Regeln genannt) vorhanden, die Stellen im Code finden, die auf schlechte Performance hindeuten (z.B. String-, StringBuffer- und StringBuilder-Nutzung). Es existieren Plug-ins für unzählige Entwicklungsumgebungen, von denen einige jedoch nicht auf dem aktuellen Stand (sowohl von Seiten der IDE, also auch von Seiten PMDs) sind. Neue Regeln können entweder in Java geschrieben werden, in dem eine entsprechende Klasse abgeleitet wird, in der die neue Art der Überprüfung durchgeführt werden soll oder auch als XPath-Ausdrücke. PMD untersucht den Java Quellcode und unterstützt Blacklisting auf verschiedenste Arten und Weisen.

Community/Support: PMD hat eine Mailingliste und ein Forum. Das Forum scheint nicht sehr aktiv zu sein (etwa zwei bis vier Beiträge pro Monat), jedoch erhält man auf gestellte Fragen, meisten noch am selben Tag eine Antwort. Die Mailingliste wird zwar frequentierter verwendet, ist jedoch eher für interne Angelegenheiten der PMD Entwickler vorgesehen. Ein kostenpflichtiger Support ist nicht vorhanden.

Dokumentation: Auf der Webseite von PMD ist eine ausführliche Liste über die vorhanden Rulesets existent. Diese ist sehr ähnlich zu der von CheckStyle aufgebaut. Jede einzelne Regel wird kurz erklärt und ein Beispiel dazu aufgeführt. Es ist auch ein Tutorial über das Erweitern von PMD, in Form von neuen Regeln, vorhanden. Es existieren außerdem einige andere Informationen, wie zum Beispiel Guidelines zum Aufstellen von guten Regeln oder weiterführende Informationen für PMD Entwickler. Eine API Referenz, welche aus den Quellcodekommentaren generiert wurde ist ebenfalls vorhanden, sie ist jedoch an vielen Stellen unvollständig oder unzureichend.

Benutzerfreundlichkeit: PMD selbst ist als Kommandozeilentool konzipiert. Es existieren jedoch Plug-ins für sehr viele IDEs (Eclipse, EMacs, JEdit, ...) oder Buildsysteme (Ant, Maven, ...). Viele Plug-ins (darunter auch das Eclipse-Plug-in) sind jedoch nicht auf dem aktuellen Stand und können somit nur eingeschränkt werden. Die letzte stabile Version des Eclipse-Plug-ins unterstützt beispielsweise nur Eclipse in der Version 3.2.6 und ist vier Jahre alt. Eine neue Version für PMD 5.0 und Eclipse 4 (aka Juno) ist jedoch gerade in Arbeit, konnte aber für diese Fachstudie nicht untersucht werden. Das Ausführen über die Konsole gestaltet sich etwas kompliziert. Rulesetdateien, welche in XML geschrieben sind

und Verweise auf Java Klassen haben, die die entsprechenden Tests implementieren, sind standardmäßig direkt in den Resourcen der PMD JAR-Datei verbaut und sind nicht immer leicht zu verstehen. Externe Dateien lassen sich jedoch auch angeben. Dennoch sind die Parameter eher unübersichtlich und schlecht dokumentiert.

```
<?xml version="1.0"?>

<ruleset name="Clone Implementation"
    xmlns="http://pmd.sourceforge.net/ruleset/2.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://pmd.sourceforge.net/ruleset/2.0.0 http://pmd.sourceforge.net/ruleset_2_0_0.xsd"
    xsi:noNamespaceSchemaLocation="http://pmd.sourceforge.net/ruleset_2_0_0.xsd">
  <description>
The Clone Implementation ruleset contains a collection of rules that find questionable usages of the clone() method.
  </description>

    <rule name="ProperCloneImplementation"
        language="java"
            since="1.4"
        message="Object clone() should be implemented with super.clone()"
        class="net.sourceforge.pmd.lang.rule.XPathRule"
         externalInfoUrl="http://pmd.sourceforge.net/rules/java/clone.html#ProperCloneImplementation">
    <description>
Object clone() should be implemented with super.clone().
    </description>
    <priority>2</priority>
    <properties>
        <property name="xpath">
            <value>
                <![CDATA[
//MethodDeclarator
[@Image = 'clone']
[count(FormalParameters/*) = 0]
[count(../Block//*[
    (self::AllocationExpression) and
    (./ClassOrInterfaceType/@Image = ancestor::
ClassOrInterfaceDeclaration[1]/@Image)
  ])> 0
]
                ]]>
            </value>
        </property>
```

Abbildung 6.12: Ruleset, welches die richtige Verwendung von clone()-Implementierungen prüft

Ausgabe: PMD unterstützt viele verschiedene Ausgabeformate. Außer einer Plain-Text Ausgabe sind vor allem HTML und XML hervorzuheben. Eine interessante Möglichkeit ist es direkt XSLT Transformationen auf die generierten XML Dateien auszuführen, um so eine Formatierung nach eigenen Vorlieben zu erhalten.

Kosten: PMD ist Open Source und somit komplett kostenfrei.

PMD report

Problems found

#	File	Line	Problem
1	C:\Users\alzi\Documents\Fachstudie\PMD\testcode\rapidminer\src\com\rapidminer\Process.java	310	Object clone() should be implemented with super.clone()
2	C:\Users\alzi\Documents\Fachstudie\PMD\testcode\rapidminer\src\com\rapidminer\Process.java	310	clone() method should throw CloneNotSupportedException
3	C:\Users\alzi\Documents\Fachstudie\PMD\testcode\rapidminer\src\com\rapidminer\datatable\DataTableNaturalSortProvider.java	94	clone() method should be implemented only if implementing Cloneable interface
4	C:\Users\alzi\Documents\Fachstudie\PMD\testcode\rapidminer\src\com\rapidminer\datatable\DataTableNaturalSortProvider.java	95	Object clone() should be implemented with super.clone()
5	C:\Users\alzi\Documents\Fachstudie\PMD\testcode\rapidminer\src\com\rapidminer\datatable\DataTableNaturalSortProvider.java	95	clone() method should throw CloneNotSupportedException
6	C:\Users\alzi\Documents\Fachstudie\PMD\testcode\rapidminer\src\com\rapidminer\datatable\DataTableSortProvider.java	33	clone() method should be implemented only if implementing Cloneable interface
7	C:\Users\alzi\Documents\Fachstudie\PMD\testcode\rapidminer\src\com\rapidminer\datatable\DataTableSortProvider.java	33	clone() method should throw CloneNotSupportedException
8	C:\Users\alzi\Documents\Fachstudie\PMD\testcode\rapidminer\src\com\rapidminer\example\AbstractAttributes.java	44	clone() method should be implemented only if implementing Cloneable interface
9	C:\Users\alzi\Documents\Fachstudie\PMD\testcode\rapidminer\src\com\rapidminer\example\AbstractAttributes.java	45	clone() method should throw CloneNotSupportedException
10	C:\Users\alzi\Documents\Fachstudie\PMD\testcode\rapidminer\src\com\rapidminer\example\Attribute.java	71	clone() method should be implemented only if implementing Cloneable interface
11	C:\Users\alzi\Documents\Fachstudie\PMD\testcode\rapidminer\src\com\rapidminer\example\Attribute.java	71	clone() method should throw CloneNotSupportedException
12	C:\Users\alzi\Documents\Fachstudie\PMD\testcode\rapidminer\src\com\rapidminer\example\AttributeDescription.java	78	clone() method should be implemented only if implementing Cloneable interface
13	C:\Users\alzi\Documents\Fachstudie\PMD\testcode\rapidminer\src\com\rapidminer\example\AttributeDescription.java	79	Object clone() should be implemented with super.clone()
14	C:\Users\alzi\Documents\Fachstudie\PMD\testcode\rapidminer\src\com\rapidminer\example\AttributeDescription.java	79	clone() method should throw CloneNotSupportedException
15	C:\Users\alzi\Documents\Fachstudie\PMD\testcode\rapidminer\src\com\rapidminer\example\AttributeRole.java	67	clone() method should be implemented only if implementing Cloneable interface

Abbildung 6.13: Standard HTML Report von PMD

6.3.7 Sonar

Untersuchte Version
3.3 (24.10.2012)

Lizenz
LGPL v3.0

Webseite
sonar.codehaus.org

Sonar ist ein unter der LGPL veröffentlichtes mehrbenutzerfähiges statisches Codeanalysewerkzeug, welches webbasierend ist und daher aus drei Komponenten besteht. Die erste Komponente entspricht einer Datenbank, in der die Resultate der Analyse gespeichert werden. Sonar bietet Unterstützung für alle gängige Datenbanken Arten an. Die zweite Komponente ist ein WebServer, in der die Ergebnisse aufbereitet untersucht werden können. Außerdem wird hier die Konfiguration der Software vorgenommen. Die dritte Komponente ist eine Konsolenanwendung, die für die Untersuchung des Quellcodes zuständig ist. In den letzten zwei Jahren gab es mehr als 15 veröffentlichte Versionen. Die Entwickler werben mit einer Downloadzahl von über 100.000. Sonar benutzt standardmäßig die sehr bekannten statischen Codeanalysewerkzeuge CheckStyle, FindBugs und PMD als Plug-ins ab Werk. Es wird sowohl der Bytecode als auch der Quellcode analysiert. Blacklisting wird von Sonar unterstützt.

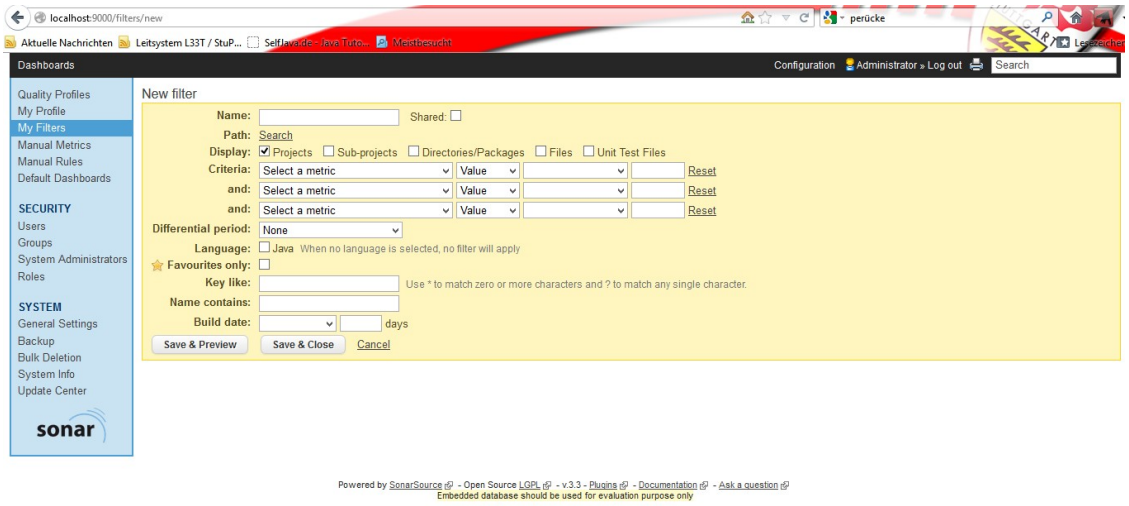


Abbildung 6.14: Blacklisting in Sonar

Community/Support: Es existieren Mailinglisten für Benutzer und für Entwickler. Diese werden sehr aktiv benutzt. Pro Monat kommen einige Hundert neue Nachrichten hinzu. Es wird zwar auch bei der nicht kommerziellen Version Support seitens der Entwickler angeboten, dieser unterscheidet sich aber aller Wahrscheinlichkeit nach stark von den Supportleistungen der kostenpflichtigen Versionen.

Dokumentation: Die Dokumente sind auf einem aktuellen Stand und es existiert eine API. Es existieren Tutorials, die für die Konfigurations- bzw. Einrichtungsaufgaben notwendig sind, um die Software erfolgreich zu benutzen. Eine ausführliche FAQ Sektion sowie Hilfestellungen für Entwickler runden das Ganze ab.

Benutzerfreundlichkeit: Wenn man zum ersten Mal mit der Software in Berührung kommt, wird man nach kurzer Zeit feststellen, dass die Konfiguration und Einrichtung mehr Zeit bedarf, als man denkt. Das Werkzeug wirbt damit, dass es einzig und allein vier kleine Schritte sind, bis man Resultate erkennen kann. Diese Schritte sind jedoch etwas komplizierter. Das Eclipse-Plug-in ist erst benutzbar, nachdem man sein Projekt vollständig durch eine Konsolenanwendung analysiert hat. Um die Konsolenanwendung jedoch zu benutzen, müssen erst Einstellungen im benutzten Betriebssystem vorgenommen werden und diverse Konfigurationsdateien von Hand erstellt werden. Es existiert keine GUI, die einem die Arbeit abnimmt. Nach der Konfiguration und einem erfolgreichem Durchlauf, kann man die erzeugten Resultate ansehen. Das Eclipse-Plug-in ist daher nicht im Stande die Konfigurations- bzw. Einrichtungs-Aufgaben zu übernehmen. Es vergeht daher etwas an Zeit, bis man eine lauffähige Software hat. Sobald man jedoch die Einarbeitungszeit hinter sich hat, ist die Software einfach zu bedienen und die Resultate werden einem sehr ausführlich angezeigt. Da die Ergebnisse in einer Datenbank gespeichert werden ist ein Vergleich mit mehreren erzielten Ergebnisse sehr gut möglich. Ebenso einigt sich dieses Codeanalysewerkzeug sehr gut, wenn mehrere Menschen am gleichen Projekt arbeiten.

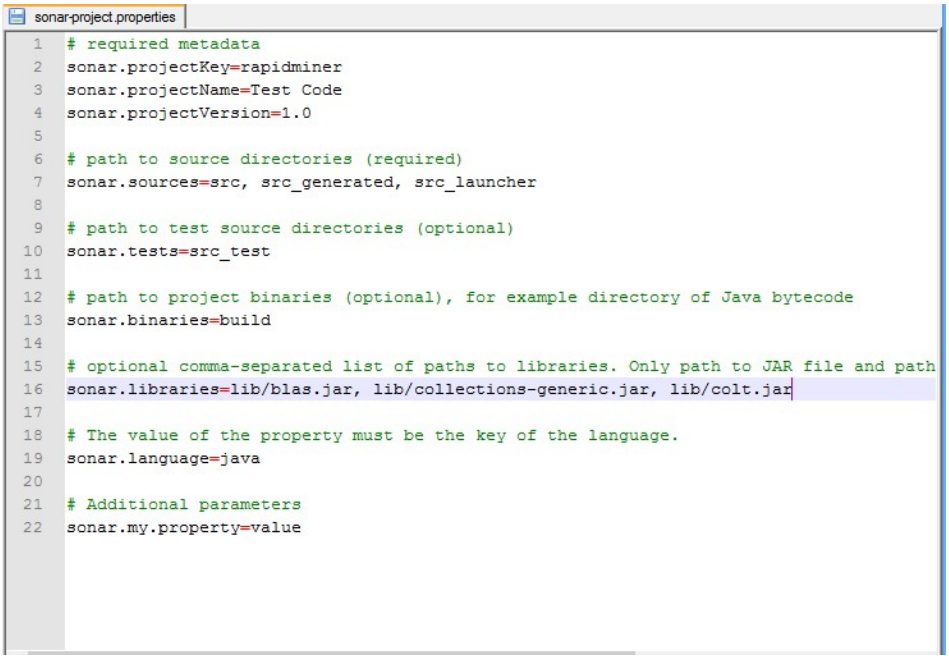


Abbildung 6.15: Der Inhalt einer Konfigurationsdatei, um die Untersuchung zu starten

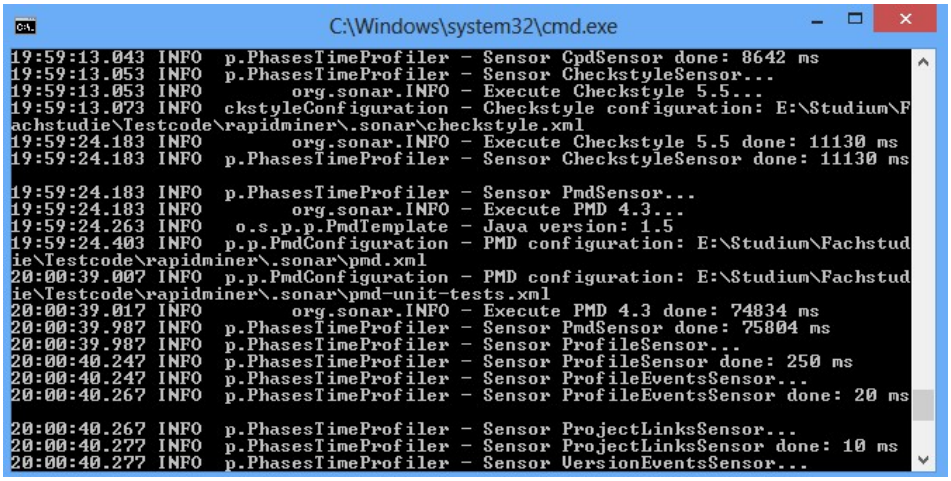


Abbildung 6.16: Anzeige während des Scans

Ausgabe: Die Resultate der Analyse werden in einer definierten Datenbank gespeichert. Sonar selbst liefert bereits eine Datenbankart mit. Mittels Webinterface können die Resultate eingesehen werden. Die Visualisierung erfolgt hierbei überwiegend textuell. Es gibt ausführliche Statistiken, die alle leicht verständlich sind. Jedoch fehlen imposante Graphiken, um die Resultate eindrucksvoller aussehen zu lassen.

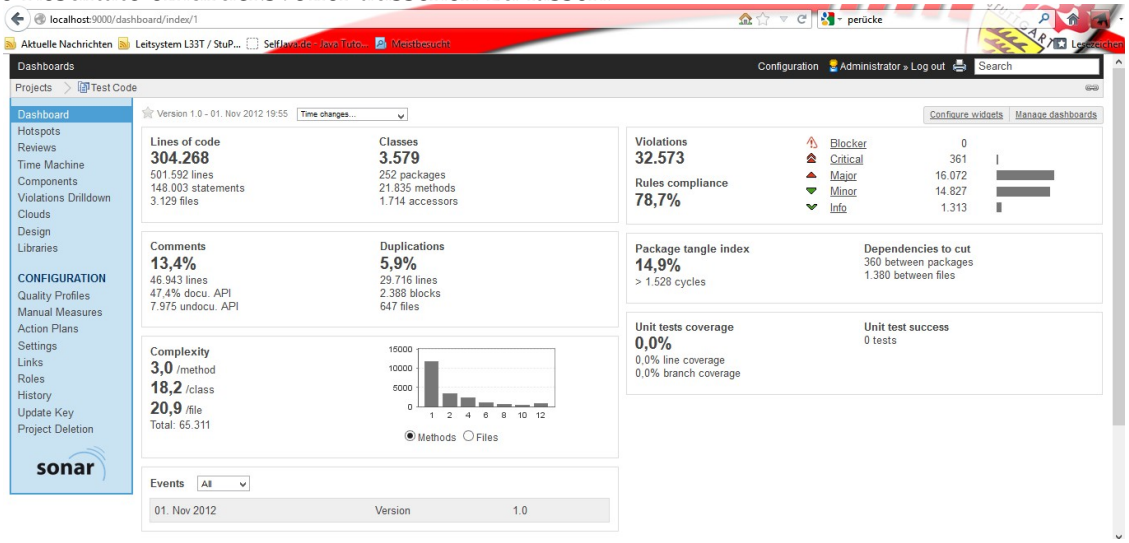


Abbildung 6.17: Hauptseite der Resultate, die nach einem Scan angezeigt werden kann

Kosten: Die eigentliche Sonar Platform ist Open Source. Es gibt jedoch noch eine Professional Edition und eine Enterprise Edition, die kommerziell sind. Diese beinhalten neben einem besseren Support, auch die Unterstützung weiterer (zum Teil exotischer) Programmiersprachen. Die Kosten verlaufen sich im mittleren fünf stelligen Bereich. Die Firma bietet für 2000 Euro einen Installations- und Konfigurationsdienst an. Zudem werden auch Schulungen angeboten, deren Kosten bei mehreren Tausend Euro liegen. Es existieren neben kostenlose Plug-ins auch kommerzielle Plug-ins.

6.3.8 Understand

Untersuchte Version
3.0.638 (02.11.2012)

Lizenz
kommerziell (Testversion erhältlich)

Webseite
scitools.com

Understand ist eine kommerzielle Codeanalyseplattform für viele verschiedene Programmiersprachen, darunter auch Java. Sie kommt mit einer kompletten Entwicklungsumgebung, inklusive Code-Editor, der jedoch stark auf die Analyse und das Reviewing von Codeprojekten, anstatt auf die tatsächliche Entwicklung innerhalb der IDE, ausgelegt ist. Understand enthält dabei unterschiedliche Pakete zum erstellen von Metriken, dem Prüfen von Coding-Standards oder anderen Dingen, die auf unsaubere Programmierung hinweisen. Außerdem ist es Understand möglich Abhängigkeiten zu analysieren und verschiedene Graphen (UML, Kontroll-Fluss, Aufrufhierarchien, ...) zu erstellen. Understand besitzt die Möglichkeit zur Erweiterung der meisten Funktionalitäten, wie dem Messen von Metriken oder eigenen Code-Checks, bis hin zu neuen Grapherstellungs-Algorithmen. Diese Erweiterungen können wahlweise über eine Perl oder eine Phython Schnittstelle gescriptet werden. Eine große Auswahl an existierenden Skripten, welche zu einem großen Teil von Benutzern geschrieben wurden, werden direkt auf der Webseite zum Download angeboten.

Community/Support: Understand verfügt über ein Forum, in dem Fragen sowohl zur Verwendung der Software selbst, als auch zur Erweiterung über Skripte, diskutiert werden können. Das Forum ist sehr aktiv und Fragen werden oft schon nach wenigen Stunden von Mitarbeitern des Herstellers beantwortet. Außerdem ist es möglich Fragen auch an eine Supportadresse per E-Mail zu senden.

Dokumentation: Understand ist ausgiebig dokumentiert. Die Entwicklungsumgebung verfügt über eine Hilfe, die alle Komponenten der Oberfläche erklärt und an Beispielen verdeutlicht. Außerdem existieren auf der Webseite ein FAQ und Videotutorials, in denen einige Features der Software genau erklärt werden. Von den Entwicklern wird auch ein Blog verwaltet, in dem Neuigkeiten über das Projekt angekündigt und vorgestellt werden. Die Perl und Phyton APIs zur Erweiterung besitzen beide sehr lange API Referenzbeschreibungen, welche alle Funktionen oder Klassen der Schnittstelle beschreiben.

Benutzerfreundlichkeit: Understand ist ein großes Framework, dass zwar über eine graphische Benutzeroberfläche verfügt, die an eine IDE erinnert, jedoch ist diese oft unübersichtlich strukturiert und macht einen überladenen Eindruck. Dies führt dazu, dass viele Funktionalitäten nicht intuitiv bedienbar sind und vieles in der Hilfe nachgeschaut werden muss. Dennoch sind alle Dinge über grafische Elemente einstellbar, die man manchmal eben einfach nur suchen muss.

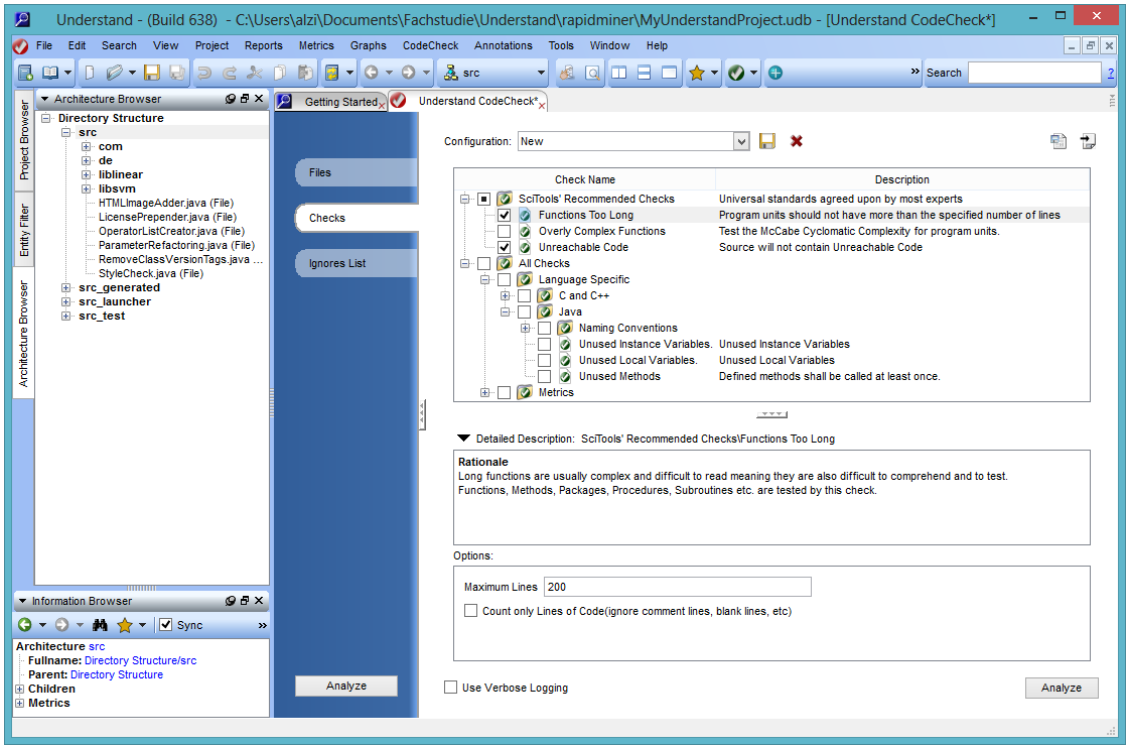


Abbildung 6.18: Konfigurationsdialog zum Auswählen von Checks, welche über der Codebasis ausgeführt werden sollen

Ausgabe: Es werden viele verschiedene Ausgabemöglichkeiten unterstützt, die in der Regel alle innerhalb der grafischen Oberfläche angezeigt werden. Understand ist es dabei möglich Befunde im Code direkt auf die entsprechenden Codezeilen zurückzuspiegeln. Außerdem lassen sich für Metriken und Befunde Treemaps erstellen, welche einen guten Überblick über Schwachstellen der Codebasis liefern. Alle Diagramme, Graphiken und Tabellen lassen sich auch in entsprechende Formate exportieren (meist CSV oder XML).

Kosten: Understand ist kostenpflichtig. Der tatsächliche Preis ist nur auf Anfrage erhältlich. Es steht jedoch eine 16 Tage Testversion zur Verfügung, die innerhalb des Testzeitraums uneingeschränkt genutzt werden kann. Der Support ist nicht an spezielle Supportverträge gekoppelt oder zeitlich begrenzt, kostet also nichts extra.

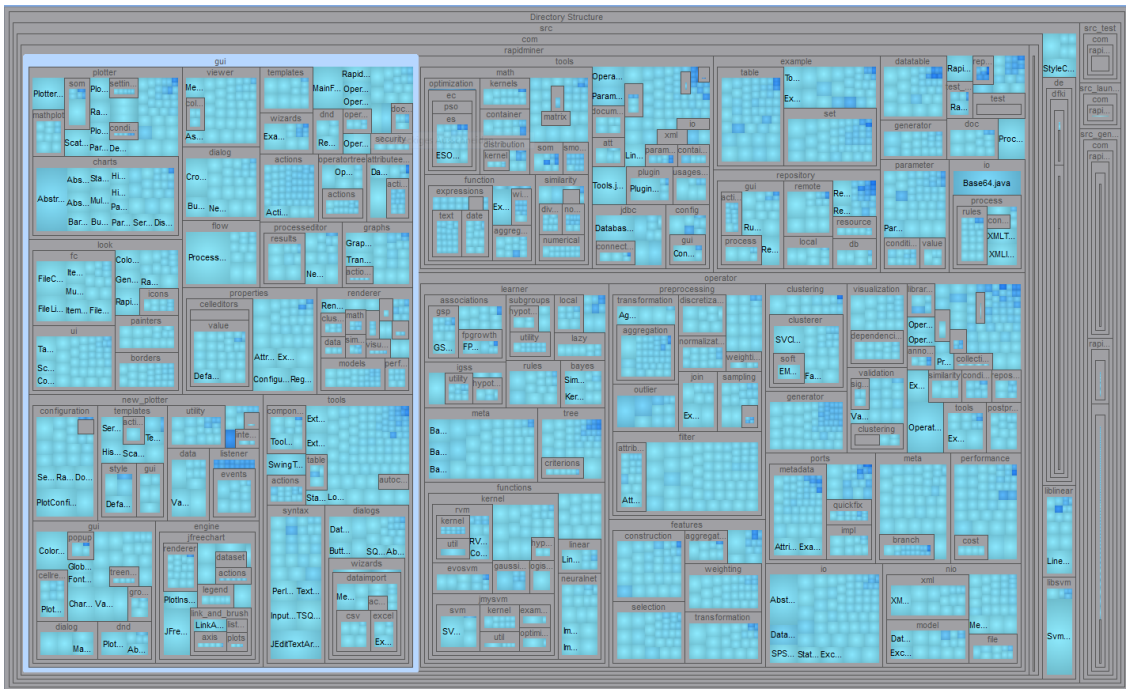


Abbildung 6.19: Treemap, welche das Verhältnis von Codezeilen zu Kommentarzeilen anzeigt

6.4 Resultat

Die detaillierten Beschreibungen der statischen Codeanalysewerkzeuge stellen das Resultat der Evaluationsphase dar. Dieses kann in kompakter Form der nachfolgenden Tabelle entnommen werden.

Kriterium	**CheckStyle**	**ConQAT**	**Find Bugs**	**JLint**	**Moose**	**PMD**	**Sonar**	**Understand**
Community/Support	⊖	⊙	⊕	⊖	⊖	⊙	⊙	⊕
Dokumentation	⊙	⊙	⊙	⊖	⊖	⊙	⊙	⊕
Benutzerfreundlichkeit	⊙	⊕	⊕	⊙	⊖	⊖	⊕	⊙
Kosten	⊕	⊙	⊕	⊕	⊕	⊕	⊙	⊖
Lizenz	⊕	⊕	⊕	⊙	⊙	⊕	⊕	⊕
Blacklisting	⊖	⊕	⊕	⊖	⊖	⊕	⊕	⊕
Performance	⊕	⊕	⊕	⊕	⊖	⊕	⊕	⊙
Quellcode vs. Bytecode	⊕	⊕	⊖	⊖	⊕	⊕	⊕	⊕
Normierte Punktzahl in %	53	71	73	32	23	60	71	75
Endnote	4+	2-	2-	5	5	3-	2-	2-

Tabelle 6.5: Resultat der Evaluation mit Bewertung der Kriterien

7 Empfehlung

Nach unserer Evaluation konnten vier Codeanalysewerkzeuge die Bestnote (2-) erreichen. Dennoch existieren teilweise große Unterschiede zwischen den einzelnen Codeanalysewerkzeugen, so dass eine Bestimmung eines eindeutigen Siegers nicht einfach möglich ist. Viel mehr hängt dies von dem konkreten Einsatzgebiet und den persönlichen Vorlieben den Benutzers ab. Im Folgenden wollen wir auf die Individualitäten der vier Gewinner-Werkzeug eingehen, so dass der Leser nach eigenem Ermessen, das für ihn richtige Codeanalysewerkzeug wählen kann. Vor allem sollte klar sein, dass in unserer Bewertung nicht auf den tatsächlichen Funktionsumfang der Codeanalysewerkzeuge eingegangen wurde, da dieser von Werkzeug zu Werkzeug sehr stark variiert und ein guter Vergleich damit unmöglich wird. Das Hauptaugenmerk lag bei dieser Fachstudie auf anderen Punkten (siehe Abschnitt 5).

Eines der Codeanalysewerkzeuge sticht besonders heraus: **FindBugs**. Denn anders als die übrigen drei Codeanalysewerkzeuge ist FindBugs ein auf ein Gebiet spezialisiertes Einzelwerkzeug und keine komplette Plattform zur statischen Codeanalyse. FindBugs ist nach unseren Kriterien, die vor allem Erweiterbarkeit in den Vordergrund stellen, sehr gut bewertet worden. Der Funktionsumfang ist jedoch durch seine Ausrichtung auf ein Themengebiet eingeschränkt. Bei den anderen Codeanalysewerkzeugen, die eher eine Plattform für verschiedene Funktionalitäten bilden, ist dies nicht der Fall. Dies geht sogar so weit, dass die Codeanalysewerkzeuge ConQAT und Sonar, FindBugs (oder auch CheckSytle und PMD) einbinden können und damit den gesamten Funktionsumfang dessen in sich integrieren.

Sonar bietet einige Besonderheiten, wie dass es auf einem Datenbanksystem aufbaut, in dem es alle Ergebnisse vorhergegangener Tests abspeichert und dass es ein Webinterface liefert, um diese bequem und übersichtlich darzustellen. Dies kann insbesondere dann interessant sein, wenn das Codeanalysewerkzeug produktiv in einem Team eingesetzt wird, so dass jeder Entwickler eine gute Übersicht über die aktuelle Qualität des Codes erhalten kann.

ConQAT punktet besonders im Bereich der Benutzerfreundlichkeit. Hier ist es sehr einfach die auszuführenden Analyseschritte in einem Graphen zusammenzustellen. Auch die Ausgabe wird sehr ordentlich und übersichtlich in vielen unterschiedlichen Formaten dargestellt. Auch eine Trendanalyse darüber, wie sich ein Softwareprojekt im Laufe seines Daseins entwickelt wird einfach unterstützt. Wichtiges Merkmal von ConQAT ist, im Gegensatz zu den anderen beiden Analyseplattformen mit der Bestnote, dass ConQAT sehr gut in Eclipse integriert ist. Dies kann vor allem für einen Entwickler, welcher in dieser IDE auch seine normale Arbeit verrichtet, von Vorteil sein.

Understand, das einzige kommerzielle Codeanalysewerkzeug in der Endauswahl, kommt mit einer eigenen grafischen Oberfläche mit integriertem Editor, die stark an eine IDE

erinnert. Diese ist jedoch nicht auf tatsächliches Entwickeln, sondern auf Tätigkeiten im Rahmen der statischen Codeanalyse ausgelegt. Wichtiger Unterschied ist außerdem, dass man Erweiterungen nicht direkt am Quellcode in Java vornehmen kann (da dieser nicht verfügbar ist), sondern diese in Perl oder Python über die speziell dafür angebotenen APIs vornehmen muss.

Erklärung

Wir versichern, diese Arbeit selbstständig verfasst zu haben. Wir haben keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommene Aussagen als solche gekennzeichnet. Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines anderen Prüfungsverfahrens. Wir haben diese Arbeit bisher weder teilweise noch vollständig veröffentlicht. Das elektronische Exemplar stimmt mit allen eingereichten Exemplaren überein.

_____

«Ort, Datum, Unterschriften»

Marius Bauer, Michael Nistor, Albert Ziegenhagel

Alle URLs wurden zuletzt am 08.11.2012 geprüft.