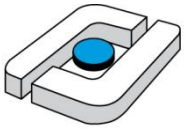

Hochschule Osnabrück
University of Applied Sciences

Bachelorarbeit

zur Erlangung des akademischen Grades

Bachelor of Science

an der

Hochschule Osnabrück

im Fachbereich Ingenieurwissenschaften und Informatik

im Studiengang Technische Informatik

Analyse der Einsatzmöglichkeiten von QS-Werkzeugen zur statischen Quellcodeanalyse von Java

Autor:	Jens Attermeyer
Matrikel-Nr.:	343671
Erstprüfer:	Prof. Dr. Stephan Kleuker
Zweitprüfer:	Prof. Dr. Frank M. Thiesing
Abgabe am:	25. Mai 2011

Kurzfassung

Auf allen Ebenen des Software-Entwicklungsprozesses gibt es verschiedene Möglichkeiten zur Qualitätssicherung. Beim Testen der zu erstellenden Software und unter Umständen auch schon während der Implementierung gibt es u.a. die statischen Testverfahren. Eines dieser Testverfahren ist die statische Quellcodeanalyse, die durch Unterstützung unterschiedlicher Werkzeuge durchgeführt werden kann. Mit Hilfe der statischen Quellcodeanalyse kann der Quellcode z.B. auf die Einhaltung von Programmierrichtlinien überprüft werden. Auch eine Syntaxprüfung und eine Überprüfung bestimmter Metriken oder das Auffinden möglicher Fehler, wie z.B. Null-Pointer-Dereferenzierungen ist möglich.

In der vorliegenden Arbeit werden einige dieser Werkzeuge analysiert. Dabei sollen die Werkzeuge auf die Einfachheit der Installation und der Benutzung hin untersucht werden. Außerdem soll die Möglichkeit einer Integration der Werkzeuge zur statischen Quellcodeanalyse in den Build-Prozess der Werkzeuge Ant und/oder Maven geprüft werden.

Als Analyseergebnis wird eine sinnvolle Kombination der betrachteten Werkzeuge aufgezeigt.

Abstract

Different possibilities of quality management are to be found at all levels of software development processes. Among others, there are static testing procedures for testing the developing software and, possibly, for testing it during implementation. A particular testing procedure is the static source code analysis which can be carried out with the help of different tools. By means of the static source code analysis the source code can be checked for the adherence of programming guidelines. Additionally a syntax check as well as a check for different metrics or detecting potential mistakes like the null-pointer-dereferencing is possible.

In the research paper at hand some of the mentioned tools are to be analysed. The intention is to examine the tools concerning the simplicity of installation and usage. Moreover the possibility of integrating the tools for the static source code analysis into the build-process of the tools Ant and/or Maven is to be checked.

As a result of analysis a meaningful combination of the tools considered will be illustrated.

Danksagung

An dieser Stelle möchte ich mich bei Prof. Dr. Stephan Kleuker bedanken, der mich während meiner Bachelorarbeit betreut und umfangreich unterstützt hat. Auch möchte ich mich bei meinem Zweitprüfer Prof. Dr. Frank M. Thiesing bedanken.

Ein ganz besonderer Dank gilt meinen Eltern, die mich nicht nur finanziell, sondern auch moralisch immer unterstützt haben.

Danken möchte ich auch meinem Kommilitonen Alexander Krieger für das Lesen meiner Bachelorarbeit und die konstruktive Kritik.

Inhalt

1 Einleitung

In den folgenden Abschnitten erfolgt eine kurze Einführung in die Thematik, es wird das Ziel dieser Arbeit benannt und der Aufbau der Bachelorarbeit wird erläutert.

1.1 Einführung in die Thematik

In der Software-Entwicklung sollte die Qualitätssicherung eine große Rolle spielen. Dabei sollten die einzelnen Entwicklungsprozesse einer gewissen Qualität unterliegen. Auch die in dieser Arbeit betrachtete Produktqualität der zu entwickelnden Software sollte den Bestimmungen der ISO 9126 – Norm [ISO 9126] entsprechen, die folgende Qualitätsmerkmale definiert:

- Funktionalität
- Zuverlässigkeit
- Benutzbarkeit
- Effizienz
- Änderbarkeit
- Übertragbarkeit

Das Qualitätsniveau der einzelnen Merkmale muss zunächst durch sogenannte Qualitätsanforderungen festgelegt werden. Diese Anforderungen können dann durch bestimmte Tests überprüft werden.

Es gibt dynamische und statische Testverfahren. Die dynamischen Tests beruhen auf einer Ausführung des erstellten Softwaresystems bzw. einzelner Teile dieses Systems (z.B. einzelne Klassen) mit ausgewählten Testdaten. Im Gegensatz dazu wird in den statischen Testverfahren die Software nicht ausgeführt, sondern einer Analyse unterzogen.

Bei den statischen Testverfahren gibt es zum Einen die strukturierten Gruppenprüfungen, welche auch als Review bezeichnet werden und auf der menschlichen Analysefähigkeit beruhen [SpTi10] S. 81ff und zum Anderen die statische Analyse. Bei einem Review werden die verschiedenen Dokumente (z.B. Quellcodes oder Unified-Modeling-Language (UML) – Diagramme) die während des Software-Entwicklungsprozesses entstehen von mehreren Personen geprüft. Da dieses Verfahren sehr aufwändig sein kann, können zur Vorbereitung eines solchen Reviews die Dokumente auch vorab durch bestimmte Werkzeuge geprüft werden. Diese sollten ein Review aber nicht ersetzen.

In dieser Arbeit sollen verschiedene Werkzeuge zur statischen Quellcodeanalyse von Java-Quellcode analysiert werden. Die Ziele dieser Analyse werden im nächsten Kapitel näher erläutert.

1.2 Ziele der Arbeit

In dieser Arbeit werden der Java-Compiler und fünf verschiedene Werkzeuge, mit denen eine werkzeuggestützte statische Quellcodeanalyse durchgeführt werden kann, analysiert. Es wurden Werkzeuge ausgewählt, die unterschiedliche Ziele verfolgen. Aufgrund von Weiterentwicklungen der Werkzeuge kommt es mittlerweile aber auch zu einigen Überschneidungen der Funktionalität dieser Werkzeuge.

Es sollen folgende Punkte betrachtet werden:

- Einarbeitungsaufwand in die Werkzeuge
- Typische Einsatzszenarien
- Installation der Werkzeuge
- Benutzung der Werkzeuge
- Verwendung der Werkzeuge mit den Build-Managementwerkzeugen Ant und/oder Maven

Als Analyseergebnis steht eine Empfehlung zur sinnvollen Kombination der betrachteten Werkzeuge.

1.3 Aufbau der Arbeit

Zunächst wird in dem Grundlagenkapitel (2) ein Überblick über die statischen Testverfahren in der Software-Entwicklung gegeben und beschrieben, wie diese ablaufen (Kap. 2.1). Danach wird die statische Quellcodeanalyse mittels QS-Werkzeugen näher erläutert (Kap. 2.2).
In Kapitel 3 geht es um die Analyse der ausgewählten Werkzeuge. Zunächst werden die Kriterien, unter denen die Analyse durchgeführt werden soll, beschrieben (Kap. 3.1). Im Anschluss daran wird die Analyse für jedes Werkzeug durchgeführt und erläutert (Kap. 3.2 – 3.7).
In Kapitel 4 folgt als Analyseergebnis eine Empfehlung zur sinnvollen Kombination der verschiedenen Werkzeuge.
Kapitel 5 fasst die wesentlichen Punkte dieser Arbeit als Abschluss noch einmal zusammen.

2 Grundlagen

In den folgenden beiden Kapiteln werden die Statischen Testverfahren erläutert. In Kap. 2.1 geht es zunächst um die manuellen Testverfahren und in Kap. 2.2 um die statische Analyse mittels Werkzeuge, wobei hier nur die Analyse des Quellcodes betrachtet wird.

2.1 Statische Testverfahren

Im Gegensatz zu den dynamischen Testverfahren kommt es bei den statischen Testverfahren nicht zu einer Ausführung der zu testenden Programme. Das statische Testverfahren ist ein analytisches Verfahren, mit dem alle Dokumente die in der Softwareentwicklung entstehen (z.B. UML-Diagramme oder der Quellcode) analysiert werden können [SpTi10] S. 81ff.

Das Ziel solcher Tests ist es, Fehler oder Auffälligkeiten, wie Abweichungen von Dokumentationsvorlagen, die schnell zu einem Fehlverhalten der zu entwickelnden Software führen können, in den Dokumenten zu entdecken. Je früher diese Tests im Software-Entwicklungsprozess eingesetzt werden, desto einfacher und kostengünstiger ist die Behebung eines entdeckten Fehlers.

Diese Analysen können manuell, d.h. durch Personen oder durch entsprechende Werkzeuge durchgeführt werden.

Eine manuelle Prüfung ist eine strukturierte Gruppenprüfung und wird auch als Review bezeichnet. Hierbei wird die menschliche Analysefähigkeit genutzt. Es gibt vier verschiedene Review-Arten, die sich z. B. im Grad der Formalität oder der Intention unterscheiden:

- Walkthrough
- Inspektion
- Technisches Review
- Informelles Review

Diese Review-Arten haben alle die gleiche grundlegende Vorgehensweise:

- Planung
- Einführung
- Vorbereitung
- Review-Sitzung
- Überarbeitung
- Nachbereitung

Die manuelle Prüfung kann sehr aufwändig werden, da die zu prüfenden Dokumente oft sehr umfangreich sein können. Außerdem werden Ressourcen eingesetzt, welche Kosten verursachen. Daher sollte eine solche Prüfung nach Möglichkeit durch geeignete Werkzeuge ergänzt werden. Diese können eine sehr gute Vorarbeit zu den Reviews leisten.

2.2 Statische Quellcodeanalyse mittels QS-Werkzeugen

Eine werkzeuggestützte statische Analyse kann bei Dokumenten die einer formalen Struktur unterliegen durchgeführt werden. Dies ist z.B. beim Quellcode von Java der Fall. Mit Hilfe dieser Quellcodeanalyse können folgende Punkte überprüft werden:

- *Syntaxverletzungen:* Der Quellcode kann auf die Einhaltung der korrekten Syntax der Programmiersprache überprüft werden. Diese Überprüfung wird z.B. auch durch einen Compiler durchgeführt.

- *Nichteinhaltung von Programmierrichtlinien:* Der Quellcode kann auf die Einhaltung bestimmter Programmierrichtlinien, wie z.B. festgelegte Einrückungen oder Namenskonventionen, überprüft werden.

- *Kontrollflussanomalien:* Kontrollflussanomalien in einem Programm, wie z.B. nicht erreichbare Anweisungen oder Sprünge aus Schleifen, können durch die Erstellung und Auswertung eines Kontrollflussgraphen entdeckt werden. Ein Kontrollflussgraph ist ein gerichteter Graph, bei dem die Knoten die Anweisungen und die Kanten die Zweige eines Programms darstellen [LIG09] S. 277.

- *Datenflussanomalien:* Datenflussanomalien können durch eine Datenflussanalyse aufgedeckt werden. Es wird der Gebrauch jeder Variablen überprüft. Eine Variable kann entweder definiert oder undefiniert sein und sie kann referenziert werden:
  - definiert (d): Der Variablen wurde ein Wert zugewiesen.
  - referenziert (r): Der Wert der Variablen wird gelesen.
  - undefiniert (u): Die Variable hat keinen definierten Wert.

  Dadurch lassen sich drei Arten von Anomalien unterscheiden:
  - ur-Anomalie: Ein undefinierter Wert wird gelesen.
  - du-Anomalie: Die Variable erhält einen Wert, der aber ungültig wird, ohne dass er vorher gelesen wurde.
  - dd-Anomalie: Die Variable erhält ein zweites Mal einen gültigen Wert, ohne dass der erste Wert verwendet wurde.

- *Ermittlung von Metriken:* Der Quellcode kann mit verschiedenen Metriken überprüft werden. Wird eine Maßzahl ermittelt, die nicht der Vorgabe entspricht, wird durch die entsprechenden Werkzeuge eine Fehlermeldung erzeugt. Es gibt *technische Metriken* und *nicht-technischen Metriken*. Die technischen Metriken lassen sich durch Werkzeuge bestimmen. Beispiele sind *Anzahl Klassen*, *Anzahl Methoden*, *Anzahl Quellcodezeilen* oder Metriken zur Komplexität [FLE07].

3 Analyse von QS-Werkzeugen zur statischen Quellcodeanalyse

In diesem Kapitel werden zunächst die Analysekriterien nach denen die QS-Werkzeuge analysiert werden beschrieben und gewichtet (Kap. 3.1). Anschließend wird die Analyse der QS-Werkzeuge einschließlich des Java-Compilers vorgestellt (Kap. 3.2 – 3.7).

3.1 Analysekriterien

Die QS-Werkzeuge sollen nach folgenden Kriterien analysiert werden:

1. *Typische Einsatzszenarien*: Hier werden die Werkzeuge daraufhin analysiert, wonach sie den Quellcode analysieren (mögliche Bugs, Programmierrichtlinien, Metriken etc.).
2. *Einarbeitungsaufwand in die Werkzeuge*: Es soll betrachtet werden, ob z.B. eine Dokumentation zu den Werkzeugen vorhanden ist, mit deren Hilfe ein einfacher, schneller Umgang mit dem Werkzeug möglich wird.
3. *Installation der Werkzeuge*: Der Installationsprozess der Werkzeuge soll betrachtet werden. Dabei wird auch geprüft ob eine hilfreiche Installationsanleitung vorhanden ist.
4. *Benutzung der Werkzeuge*: Hier werden die Möglichkeiten der Benutzung der Werkzeuge untersucht, sind z.B. Plug-Ins für Entwicklungsumgebungen vorhanden oder lässt sich das Werkzeug ausschließlich über eine Konsole bedienen.
5. *Verwendung der Werkzeuge mit Ant und/oder Maven*: Es wird analysiert, ob sich die Werkzeuge auch in den Build-Prozess mit Ant und/oder Maven integrieren lassen.

Die Kriterien sind so gewichtet, dass die typischen Einsatzszenarien im Vordergrund stehen. Anhand dieser soll als Analyseergebnis eine mögliche, sinnvolle Kombination der Werkzeuge gezeigt werden. Eine mögliche längere Einarbeitungszeit oder eine umständliche Installation eines Werkzeugs kann einen Einsatz dieses Werkzeugs dennoch rechtfertigen, solange am Ende eine bessere Softwarequalität gegeben ist. Diese Kriterien werden aber nicht so hoch gewichtet, da die Installation oder die Einarbeitung nur einmalig durchgeführt werden müssen und deshalb kein Ausschlusskriterium sein sollten.

Mit Ant können automatisiert ausführbare Programme, aus den einzelnen Quellcode-Dateien, erstellt werden. Für diesen Prozess wird eine XML-Datei benötigt, in der die Reihenfolge der einzelnen Schritte zur Erzeugung des ausführbaren Programms beschrieben wird. Diese Schritte werden in einer speziellen Skriptsprache definiert, die im Falle von Ant aus einzelnen XML-Elementen besteht. Standardmäßig heißt diese XML-Datei *build.xml*.
Maven ist ein weiteres Build-Management-Werkzeug. Wie Ant benötigt auch Maven eine XML-Datei, in der die einzelnen Schritte zur Erstellung des ausführbaren Programms beschrieben werden. Diese heißt standardmäßig *pom.xml*, wobei pom für *Projekt Object Model* steht.

3.2 Der Java-Compiler und die statische Quellcodeanalyse

Ein Compiler ist in zwei Hauptphasen gegliedert, die *Analysephase* und die *Synthesephase*. Die Analysephase des Compilers führt eine statische Quellcodeanalyse des zu kompilierenden Quelltextes durch. Diese Analyse erfolgt in drei Schritten [AHO08]:

- *Lexikalische Analyse:* Diese Analyse wird durch den sogenannten Scanner durchgeführt, welcher den eingelesenen Quellcode in zusammenhängende Token, wie z.B. Schlüsselwörter, Bezeichner, Zahlen und Operatoren, zerteilt.

- *Syntaktische Analyse:* Diese Analyse wird durch den sogenannten Parser durchgeführt. Dabei werden die Token der lexikalischen Analyse in einen Syntaxbaum überführt und es wird geprüft ob die Syntax der der Quellsprache entspricht

- *Semantische Analyse:* Die semantische Analyse überprüft z.B. ob eine Variable vor ihrer Nutzung deklariert wurde oder ob die Typkonsistenz bei Ausdrücken gegeben ist.

Treten bei den Analysen durch den Compiler Fehler auf, werden sie ausgegeben. Diese können dann ausgewertet und im Quellcode behoben werden.

Der Java-Compiler *javac* bietet eine Vielzahl an Optionen. Es gibt die Standard-Optionen und die Nicht-Standard-Optionen. Zu den Nicht-Standard-Optionen gehört die Option *–Xlint* [JW10]. Hierdurch können weitere, folgende Warnungen ausgegeben werden:

- *-Xlint:all* : Bei Angabe dieser Option werden alle folgenden, möglichen Warnungen berücksichtigt.
- *-Xlint:cast* : Diese Option warnt bei redundanten Casts.
- *-Xlint:deprication* : Diese Option warnt bei der Nutzung von Methoden aus dem SDK, die veraltet sind und ersetzt wurden.
- *-Xlint:divzero* : Diese Option warnt vor einer Division durch 0.
- *-Xlint:empty* : Diese Option warnt vor einer if-Abfrage ohne einen Anweisungsblock.
- *-Xlint:unchecked* : Diese Option warnt, falls ungeprüfte oder unsichere Operationen verwendet werden.
- *-Xlint:fallthrough* : Diese Option warnt, falls keine break-Anweisung in einem case-Statement vorhanden ist.
- *-Xlint:path* : Diese Option warnt bei nicht vorhandenen Pfaden bzw. Verzeichnissen.
- *-Xlint:serial* : Diese Option warnt, falls keine serialVersionUID bei einer serialisierbaren Klasse definiert wurde.
- *-Xlint:finally* : Diese Option warnt, falls ein finally-Block nicht normal beendet werden kann.
- *-Xlint:overrides* : Diese Option warnt, falls eine Methode überschrieben werden soll, die Parameter aber nicht übereinstimmen.

Im Anhang (S. 53) befindet sich der Quellcode zweier Java-Klassen, um die Warnungen des Compilers durch die *–Xlint* – Option zu zeigen.

Die Analysen des Compilers sind sehr hilfreich, bieten aber nicht den Umfang der im Folgenden vorgestellten QS-Werkzeuge.

3.3 Checkstyle

In diesem Abschnitt soll das QS-Werkzeug Checkstyle [CS] genauer betrachtet werden. Dabei wird zunächst beschrieben was Checkstyle ist und zu welchem Zweck es verwendet werden kann. Außerdem wird auf die Installation und die Benutzung des Werkzeugs eingegangen. Als letztes soll geprüft werden, ob sich Checkstyle auch in Verbindung mit den Build-Management-Werkzeugen Ant und/oder Maven verwenden lässt. Eine mögliche Einbindung des Werkzeugs in den Build-Prozess wird beschrieben.

3.3.1 Beschreibung und typische Einsatzszenarien

Checkstyle ist ein Open Source Programm, dass der Lesser GNU Public License unterliegt und in Java programmiert ist. Die ursprüngliche Hauptfunktion von Checkstyle war Java-Quellcode auf die Einhaltung bestimmter Programmierrichtlinien zu untersuchen. Seit der Version 3 können von Checkstyle auch Klassendesign-Probleme, duplizierter Code oder auch verschiedene (mögliche) Bugs gefunden werden. Es wird die Version 5.3 von Checkstyle betrachtet.

Verwendet werden kann Checkstyle über die Kommandozeile, als Plug-In für verschiedene integrierte Entwicklungsumgebungen (IDE) (siehe 3.3.3) oder im Build-Management mit Ant oder Maven (siehe 3.3.4).

Checkstyle kann über eine XML-Datei mit benutzerspezifischen Einstellungen konfiguriert werden. Diese XML-Datei kann verschiedene Module, welche für die unterschiedlichen Checks stehen, enthalten. Zurzeit stehen 131 Module bzw. Checks zur Verfügung. Zusätzlich können eigene Checks in Java programmiert und der XML-Datei als Modul hinzugefügt werden. Die vorhandenen Standardchecks sind in 15 Kategorien eingeteilt:

Annotations:	Diese Checks überprüfen den Quellcode im Zusammenhang mit Annotationen, z.B. wird geprüft ob die *java.lang.Override*-Annotation vorhanden ist, falls das *{@inheritDoc}* Javadoc-Tag vorhanden ist.
Block Checks:	Diese Checks überprüfen den Quellcode im Zusammenhang mit einzelnen Codeblöcken, z.B. wird nach leeren Blöcken gesucht.
Class Design:	Diese Checks überprüfen den Quellcode im Zusammenhang mit dem Klassendesign, z.B. sollte eine Klasse, die nur private Konstruktoren enthält, als *final* deklariert werden.
Coding:	Diese Checks überprüfen den Quellcode im Zusammenhang mit der Codierung, z.B. wird nach leeren Anweisungen gesucht, oder ob lokale Variablen oder Methodenparameter die ihren Wert nie ändern als *final* deklariert sind.
Duplicate Code:	Dieser Check überprüft den Quellcode auf duplizierten Code.
Headers:	Diese Checks überprüfen die Quellcode-Dateien, ob sie mit einem bestimmten Header beginnen.
Imports:	Diese Checks überprüfen den Quellcode im Zusammenhang mit Import-Anweisungen z.B. werden ungenutzte Paket-Importe gefunden.

Javadoc Comments:	Überprüft den Quellcode im Zusammenhang mit Javadoc-Kommentaren. Z.B. wird geprüft ob jede Variable mit einem Javadoc-Kommentar versehen ist.
Metrics:	Überprüft die definierten Klassen auf die Einhaltung bestimmter Metriken, wie z.B. die *zyklomatische Komplexität*.
Missellaneus:	Unter diesem Punkt sind verschiedene Überprüfungen zusammengefasst. Z.B. kann geprüft werden, ob eine Quellcode-Datei am Ende einen Zeilenumbruch enthält.
Modifiers:	Diese Checks beziehen sich auf die Verwendung der Modifier in Java. Z.B. kann überprüft werden, ob die Angabe der Modifier die Reihenfolge, wie sie in der Java Sprachspezifikation angegeben ist, einhält.
Naming Conventions:	Diese Checks überprüfen den Quellcode auf die Einhaltung festgelegter Namenskonventionen. Diese Konventionen können durch Reguläre Ausdrücke festgelegt werden.
Regexp:	Hierrüber können verschiedene Reguläre Ausdrücke bestimmt werden, auf die der Quellcode überprüft werden soll. Z.B. kann nach Verwendungen von „System.out.println“ gesucht werden.
Size Violations:	Diese Checks überprüfen den Quellcode auf die Einhaltung bestimmter Größenbeschränkungen. Z.B. kann der Quellcode auf die Einhaltung bestimmter Zeilenlängen überprüft werden.
Whitespace:	Diese Checks überprüfen den Quellcode auf die Verwendung von Leerzeichen.

Auf eine detaillierte Beschreibung der unterschiedlichen Checks wird aufgrund der Vielzahl an dieser Stelle verzichtet und auf die Projektseite von Checkstyle [CS] verwiesen.

3.3.2 Installation

Zur Verwendung von Checkstyle über die Kommandozeile oder mit Ant muss zunächst die Datei *checkstyle-5.3-bin.zip* von der Projektseite heruntergeladen werden. Danach wird diese Datei in ein beliebiges Verzeichnis entpackt. Am einfachsten kann Checkstyle dann verwendet werden, wenn der Dateipfad von *checkstyle-5.3-all.jar* der Umgebungsvariablen PATH hinzugefügt wird.

Um Checkstyle mit der IDE Eclipse verwenden zu können, wir das Checkstyle-Plug-In über den üblichen Weg der Installation von Eclipse-Plug-Ins installiert. Der im Folgenden beschriebene Installationsvorgang setzt voraus, dass Eclipse bereits installiert ist und gilt für die Version 3.6.2 (Helios):

- Wähle *Help* => *Install New Software* => Unter “Work with” auf *Add…* klicken
- *Name*: “Checkstyle”, *Location*: “http://eclipse-cs.sourceforge.net/update
- Wähle *Ok* => Häkchen setzten bei “Checkstyle“ => *Next* => *Next*
- Der Lizenzvereinbarung zustimmen
- Wähle *Finish* und zum Schluss *Restart Now*

Für Maven existiert ebenfalls ein Checkstyle-Plug-In, auf welches unter 3.3.4 näher eingegangen wird.

Es existieren außer einem Plug-In für Eclipse und Maven noch Checkstyle-Plug-Ins für:

- IntelliJ
- NetBeans
- BlueJ
- tIDE
- Emacs JDE
- jEdit
- Vim editor
- Krysalis Centipede
- Sonar
- QALab
- Borland Jbuilder

Diese Plug-Ins werden in dieser Arbeit nicht betrachtet.

3.3.3 Benutzung

Beispiel einer Konfigurationsdatei

Als erstes soll anhand eines kleinen Beispiels der Aufbau einer Konfigurationsdatei für Checkstyle gezeigt werden, in der verschiedene bereits in Checkstyle realisierte Überprüfungsmöglichkeiten zu einer eigenen Prüfungsvariante kombiniert werden .

```
1  <?xml version="1.0" encoding="UTF-8"?>
2 <!DOCTYPE module PUBLIC "-//Puppy Crawl//DTD Check Configuration 1.3//EN"
3         "http://www.puppycrawl.com/dtds/configuration_1_3.dtd">
4
5  <module name="Checker">
6    <module name="TreeWalker">
7      <module name="JavadocVariable"/>
8      <module name="CyclomaticComplexity"/>
9        <property name="max" value="${comp}"/>
10     </module>
11     <module name="RequireThis">
12     <module name="MethodeLenght"/>
13       <property name="tokens" value="CTOR_DEF"/>
14       <property name="max" value="${mlength}"/>
15     </module>
16   </module>
17 </module>
```

Zeile 5: Jede Konfiguration beginnt mit dem XML-Element für ein Modul *Checker*. Die Unterelmente von Checker sind die Module, die zur Überprüfung des Quellcodes verwendet werden.

Zeile 6: Das *TreeWalker*-Modul erhält eine Java-Datei und erstellt daraus mit Hilfe des ANTLR Parsers [ANTLR] einen abstrakten Syntaxbaum (AST) (Abb. 3.1). Die Knoten dieses Baums haben bestimmte Token (in Abb. 3.1 in der Spalte *Type*). Durch diesen AST wird traversiert und die entsprechenden Untermodule, die sich für ein bestimmtes Token registriert haben, werden aufgerufen.

Zeile 7: Dieses Modul überprüft ob eine Variable mit einem Javadoc-Kommentar versehen ist.

Zeile 8: Dieses Modul überprüft die „Zyklomatische Komplexität“ im Rumpf eines Konstruktors, einer Methode, eines Static-Initalisieres oder eines Instanz-Initialisierers.

Zeile 9: Über die Eigenschaft *max* wird die Obergrenze für die „Zyklomatische Komplexität“ festgelegt. In diesem Fall kann der Wert für *max* über eine Kommandozeilen-Property oder über eine Ant-Checkstyle-Task-Property festgelegt werden.

Zeile 11: Dieses Modul überprüft ob bei dem Zugriff auf eine Instanzvariable oder eine Methode des aktuellen Objekts explizit über „this.“ zugegriffen wird.

Zeile 12: Dieses Modul überprüft eine Methode oder einen Konstruktor auf die Einhaltung einer bestimmten Zeilenlänge.

Zeile 13: Über das Property *tokens* kann ein bestimmtes Token aus dem AST gewählt werden, für das die Überprüfung durchgeführt werden soll. Hier wird also explizit festgelegt, dass nur Konstruktoren überprüft werden sollen. Ist dieses Property nicht angegeben wird der Defaultwert verwendet, welcher festlegt, dass sowohl Methoden als auch Konstruktoren auf ihre Länge überprüft werden.

Zeile 14: Dieses Property legt die maximale Zeilenanzahl eines Konstruktors fest.

Wie das Beispiel zeigt, gibt es zu den einzelnen Modulen auch Properties, mit denen die Eigenschaften der Module eingestellt werden können.
Zu dem Beispiel befindet sich im Anhang auf Seite 56 ein passendes Beispielprogramm (*KonfigBeispiel.java*).

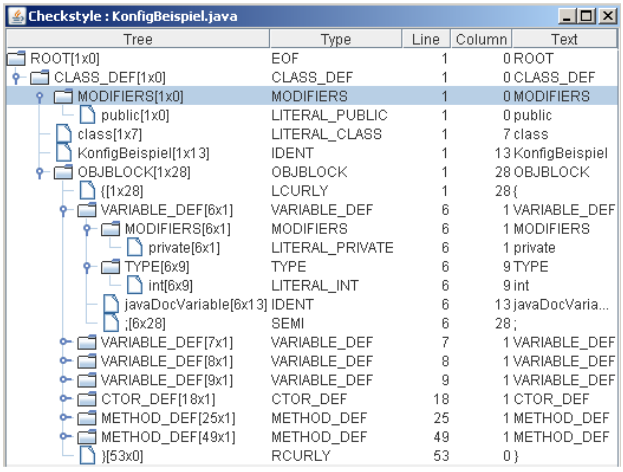

Abb. 3.1: Abstract Syntax Tree (AST) der KonfigBeispiel.java-Datei

Benutzung über die Kommandozeile

Checkstyle kann über die Kommandozeile wie folgt aufgerufen werden:

```
java   [-D<property>=<value>]           \
       com.puppycrawl.tools.checkstyle.Main    \
       -c <configurationFile>     \
       [-f <format>][-p <propertiesFile>]      [-o <file>]   \
       [-r <dir>] javafile
```

Checkstyle überprüft die als `javafile` angegebenen Java-Quellcodedateien und gibt mögliche Fehlermeldungen der Überprüfung auf der Standardausgabe aus. Die Bedeutungen der Parameter sind Folgende:

- `-D<property>=<value>`: Über diesen Parameter können die erweiterten Property-Values, wie im Beispiel der Konfigurationsdatei auf Seite 10, *comp* und *mlength*, gesetzt werden.
- `-f`: Hiermit wird das Format der der Fehlermeldungen angegeben. Mögliche Formate sind „plain“ oder „xml“. Der Defaultwert ist plain.
- `-p`: Über diesen Parameter kann eine Properties-Datei angegeben werden, welche Werte für die erweiterten Property-Values spezifiziert. Wird eine solche Datei verwendet und gleichzeitig der `-D<property>=<value>` - Parameter, wird nur die Properties-Datei berücksichtigt.
- `-o`: Über diesen Parameter kann eine Datei angegeben werden, in die die Fehlermeldungen geschrieben werden sollen.
- `-r`: Über diesen Parameter kann ein Ordner angegeben werden in dem sich die zu überprüfenden Java-Dateien befinden. Es werden dann alle im angegebenen Ordner und dessen Unterordner befindlichen Java-Dateien geprüft.

Um das Beispielprogramm *KonfigBeispiel.java* gegen die Konfigurationsdatei *KonfigBeispiel.xml* über die Kommandozeile zu testen, kann man also folgendes eingeben:

```
java   –Dcomp=5 com.puppycrawl.tools.checkstyle.Main –c /
       KonfigBeispiel.xml –p KonfigBeispiel.properties –f xml –o /
       KonfigBeispiel_Fehler.xml –r src/
```

Hier wird nun die im Ordner *src* befindliche Java-Datei *KonfigBeispiel.java* überprüft. Mögliche Fehler die bei der Überprüfung auftauchen werden im XML-Format in die *KonfigBeispiel_Fehler.xml* – Datei geschrieben, welche neu angelegt wird, falls sie noch nicht existiert. In der Properties-Datei (*KonfigBeispiel.properties*) werden die Werte für die Properties angegeben. Fehlt in der Properties-Datei die Angabe „comp = 5“ und wird der Parameter „-Dcomp=5“ angegeben, erscheint die Fehlermeldung „Property ${comp} has not been set“, was zeigt, dass bei Angabe des Parameters „-p“ und einer Properties-Datei der Parameter „-Dcomp=5“ ignoriert wird.
Die Beispiel-Dateien befinden sich im Anhang auf Seite 56.

Zum Anzeigen eines AST wie in Abbildung 3.1 wird das Graphical User Interface (GUI) von Checkstyle wie folgt aufgerufen:

```
java com.puppycrawl.tools.checkstyle.gui.Main src/KonfigBeispiel.java
```

Benutzung mit Eclipse

Um Checkstyle mit Eclipse nutzen zu können, muss das Checkstyle Plug-In zunächst wie unter 3.2 beschrieben, installiert worden sein. Durch Auswahl von *Window => Preferences* kann man zu folgendem Dialog gelangen:

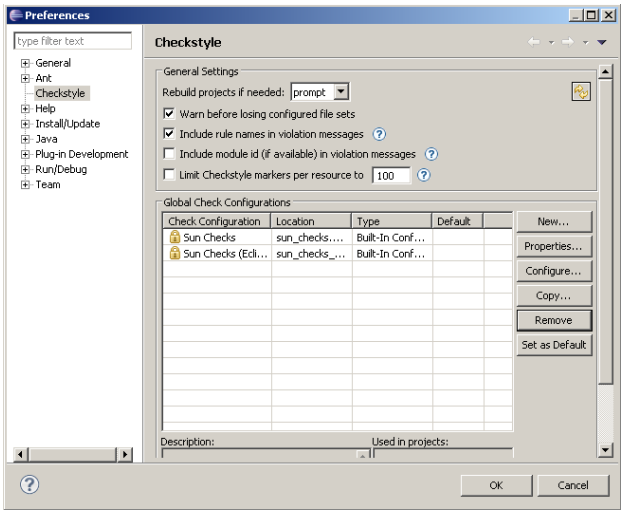

Abb. 3.2: Dialog für Checkstyle-Einstellungen

Wie in Abb. 3.2 zu sehen ist, werden standardmäßig die Checks von Sun mit dem Checkstyle-Plug-In eingebunden. Diese könnten über *Configure* an die eigenen Bedürfnisse angepasst werden, was hier aber nicht betrachtet wird.
Über *New*... erhält man folgenden Dialog über den neue Checks definiert werden können:

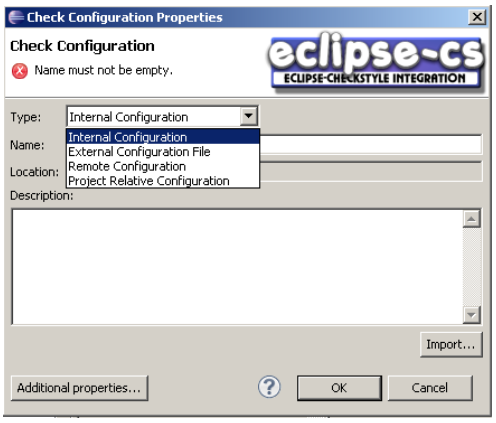

Abb. 3.3: Dialog zur Erstellung eines neuen Checks

Wie in Abb. 3.3 zu sehen, stehen für den Typ einer Check-Konfiguration vier Möglichkeiten zur Auswahl:

- *Internal Configuration*: Interne Konfigurationen werden in den Metadaten von Eclipse gespeichert.
- *External Configuration*: Dieser Typ wird verwendet, falls eine außerhalb des Eclipse-Workspaces vorhandene Checkstyle-Konfiguration verwendet werden soll.
- *Remote Configuration*: Dieser Typ wird verwendet, falls die Checkstyle-Konfiguration auf einem Webserver bereitgestellt werden soll.
- *Projekt Relativ Configuration*: Dieser Typ wird verwendet, falls die Checkstyle-Konfiguration für ein bestimmtes Projekt genutzt wird.

Hier wird nun an einem Beispiel der Umgang mit dem Typ *Internal Configuration* gezeigt, in dem die gleiche Konfiguration wie in dem Einführungsbeispiel dieses Kapitels auf Seite 10 erstellt wird.

Der Dialog aus Abb. 3.3 wird folgendermaßen gefüllt:

Typ: Internal Configuration
Name: KonfigBeispiel

Anschließend wird die Eingabe mit *OK* bestätigt.

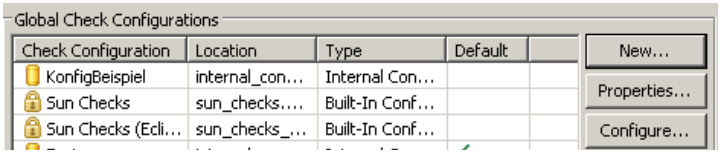

Abb. 3.4: Übersicht vorhandener Check-Konfigurationen

Wie in Abb. 3.4 zu sehen, steht nun die neue Check-Konfiguration „KonfigBeispiel“ zur Verfügung, muss aber noch mit Inhalt gefüllt werden. Dies geschieht durch die Auswahl von *Configure*... und man erhält folgenden Dialog:

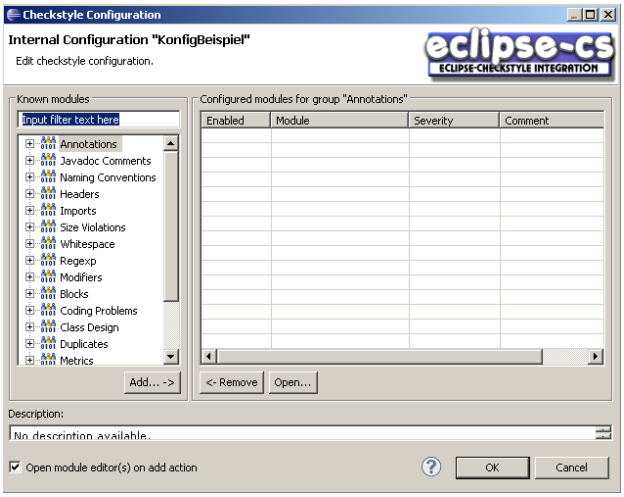

Abb. 3.5: Dialog zur Erstellung einer Checkstyle-Konfiguration

Durch Auswahl der Checks auf der linken Seite unter „Known modules“ und einem Klick auf *Add...* werden diese der Checkstyle-Konfiguration hinzugefügt. Zum Schluss mit *OK* bestätigen.

Durch einen Rechtsklick auf den Projektordner, welcher die zu prüfenden Java-Dateien enthält, kann man den Menüeintrag *Properties* wählen und gelangt zu folgendem Dialog:

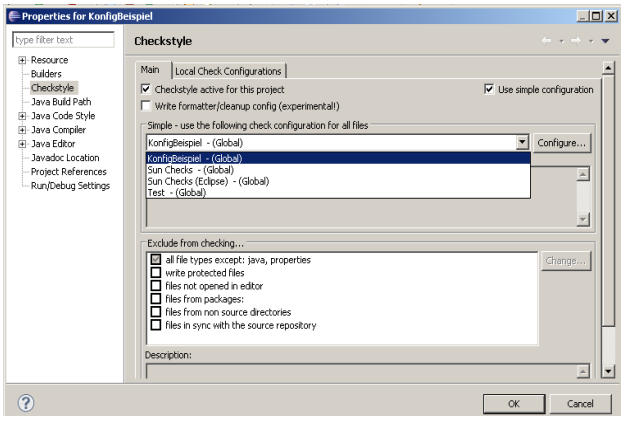

Abb. 3.6: Dialog zur Auswahl einer Konfiguration für ein Projekt

Durch die Auswahl der zuvor erstellten Checkstyle-Konfiguration „KonfigBeispiel“ wie in Abb. 3.6 zu sehen und der Auswahl von „Checkstyle active for this project“, kann das Beispielprogramm überprüft werden. Dies führt dazu, dass mögliche Fehlermeldungen direkt im Editor angezeigt werden (Abb. 3.7).

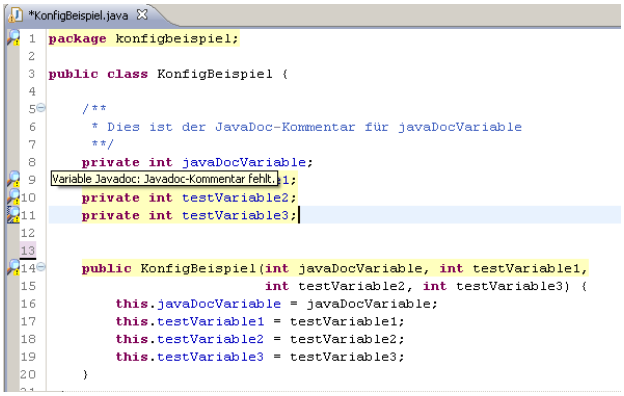

Abb. 3.7: Eclipse-Editor mit Hervorhebung der von Checkstyle gefundenen Fehler

Dies soll einen Einstieg in Benutzung von Checkstyle mittels Eclipse zeigen. Für eine ausführliche Beschreibung wird auf [CSE] verwiesen.

3.3.4 Verwendung mit Ant und/oder Maven

Ant

Checkstyle lässt sich durch eine *taskdef* –Deklaration in der von Ant benötigten build.xml-Datei in den Build-Prozess integrieren. Der Dateipfad von checkstyle-5.3-all.jar sollte dabei in der PATH-Variablen angegeben sein.

Um die KonfigBeispiel.java-Datei von Checkstyle in Verbindung mit Ant zu prüfen sieht die build.xml-Datei wie folgt aus:

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <project name="Ant" basedir=".">
3     <taskdef resource="checkstyletask.properties"
4            classpath="path/to/checkstyle-5.3-all.jar/>
5
6     <target name="checkstyle">
7           <checkstyle config="KonfigBeispiel.xml"
properties="KonfigBeispiel.properties">
8                 <fileset dir="src" includes="*.java"/>
9                 <formatter type="xml"
toFile="KonfigBeispiel_Fehler.xml"/>
10          </checkstyle>
11    </target>
12</project>
```

In Zeile 3 und 4 befindet sich die *taskdef*-Deklaration. In Zeile 6 wird ein Ziel (target) für das Ant-Skript definiert, welches die Überprüfung der Java-Dateien mittels Checkstyle ausführt. In Zeile 7 ist angegeben, welche Konfigurationsdatei (KonfigBeispiel.xml) und Propertiesdatei (KonfigBeispiel.properties) Checkstyle verwenden soll. In Zeile 8 wird Checkstyle mitgeteilt, welche Dateien (*.java, alle Javadateien) geprüft werden sollen und in welchem Ordner sie sich befinden (src). In Zeile 9 wird das Format der Fehlerausgabe (xml)

und die Datei (KonfigBeispiel_Fehler.xml), in die die Fehlermeldungen geschrieben werden sollen, festgelegt.

Dieses Beispiel zeigt wie Checkstyle im Zusammenhang mit Ant verwendet werden kann. Es existieren noch weitere mögliche Parameter für Checkstyle, die in der build.xml-Datei angegeben werden können. Darauf wird an dieser Stelle nicht näher eingegangen, sondern auf [CS] verwiesen.

Maven

Für Maven existiert ein Checkstyle-Plug-In, welches zum Standardumfang von Maven gehört. Es kann über die Kommandozeile oder durch Hinzufügen in das Projektmodell (POM) genutzt werden.

Um die KonfigBeispiel.java-Datei mittels Checkstyle im Zusammenhang mit Maven zu prüfen kann dies über die Kommandozeile wie folgt geschehen:

```
mvn checkstyle:checkstyle -Dcheckstyle.input.file=src/KonfigBeispiel.java
                          -Dcheckstyle.output.file=checkstyleFehler.xml
                          -Dcheckstyle.output.format=xml
```

Über `-Dcheckstyle.input.file=src/KonfigBeispiel.java` wird die zu überprüfende Javadatei angegeben und über `-Dcheckstyle.output.file=checkstyleFehler.xml` und `-Dcheckstyle.output.format=xml` wird die Fehlerausgabedatei und das Ausgabeformat angegeben. Die von Checkstyle verwendete Konfigurationsdatei muss in diesem Fall *checkstyle.xml* heißen und im aktuellen Verzeichnis liegen.

Die zweite Möglichkeit Checkstyle mittels Maven zu verwenden ist es, das Checkstyle-Plug-In in der pom.xml-Datei hinzuzufügen. Der folgende Ausschnitt einer solchen pom-Datei zeigt dies.

```
<project>
  ...
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-checkstyle-plugin</artifactId>
        <version>2.6</version>
        <configuration>
          <configLocation>checkstyle.xml</configLocation>
        </configuration>
      </plugin>
  ...
</project>
```

Auch hier existieren weiter Einstellmöglichkeiten für Checkstyle. Für weitere Details wird auf [CSM] verwiesen.

3.4 FindBugs

In diesem Abschnitt soll das QS-Werkzeug FindBugs [FB] genauer betrachtet werden. Dabei wird zunächst beschrieben was FindBugs ist und zu welchem Zweck es verwendet werden kann. Außerdem wird auf die Installation und die Benutzung des Werkzeugs eingegangen. Als letztes soll geprüft werden, ob sich FindBugs auch in Verbindung mit den Build-Management-Werkzeugen Ant und/oder Maven verwenden lässt und eine mögliche Einbindung des Werkzeugs in den Build-Prozess wird beschrieben.

3.4.1 Beschreibung und typische Einsatzszenarien

FindBugs ist ein Open Source Programm und wurde an der University of Maryland entwickelt. Es wurde in Java geschrieben und unterliegt der Lesser GNU Public License. FindBugs dient zum Aufspüren möglicher Bugs, also Fehler im Quellcode. Hierzu nutzt es sogenannte *Bug Pattern*. Also Muster, die ein Anzeichen dafür sind, dass dieser Teil des Programms zu einem Fehlverhalten führen kann. FindBugs untersucht dabei den Java-Bytecode und greift dazu auf die *Byte Code Engineering Library* [BCEL] zurück. Es besteht die Möglichkeit, dass die möglicherweise fehlerhaften Stellen im Quellcode angezeigt werden.

Genutzt werden kann FindBugs über die Kommandozeile, eine eigene GUI, mit Ant und Maven oder als Plug-In für verschiedene Entwicklungsumgebungen. Als Beispiel für ein solches Plug-In wird in dieser Arbeit die Integration in Eclipse näher betrachtet.

In der Version 1.3.9 von FindBugs, welche in dieser Arbeit betrachtet wird, kann der Bytecode nach 369 Bug Pattern untersucht werden. Hierzu existieren 369 sogenannte Detectoren, welche in folgende 9 Kategorien eingeteilt sind:

Bad practice:	Diese Detectoren entdecken Verstöße gegen Regeln der „guten Praxis“, wie z.B. allgemein bekannte Namenskonventionen.
Correctness:	Es werden Codestellen mit offensichtlichen Fehlern gefunden, wie z.B. eine Klasse die das Serializable-Interface implementiert, aber keine serialVersionUID definiert.
Experimental:	Diese Kategorie enthält Detectoren, die noch nicht vollständig sind und an denen momentan noch gearbeitet wird.
Internationalization:	Es werden Codestellen gefunden, welche Probleme bei der Verarbeitung von z.B. internationalen Sonderzeichen verursachen können.
Malicious code vulnerability:	Es können Codestellen gefunden werden, an denen eine Verletzbarkeit durch bösartigen Code besteht. Z.B. sollte eine finalize() Methode nicht public sein.
Multithreaded correctness:	Es werden Codestellen mit offensichtlichen Fehlern im Zusammenhang mit Multithreading gefunden, z.B. wenn die Methode notify() anstatt notifyAll() verwendet wird.
Performance:	Es werden Codestellen gefunden, die die Performance verschlechtern, wie z.B. der Aufruf der toString()-Methode an einem String.

Security:	Es werden Codestellen gefunden, die ein Sicherheitsrisiko darstellen, z.B. Passwörter für eine Datenbankverbindung, die fest in den Code geschrieben wurden.
Dodgy:	Es werden fragwürdige, verwirrende Codestellen gefunden, z.B. wird eine Variable sich selbst zugewiesen.

Auf eine genauere Beschreibung der unterschiedlichen Detectoren wird aufgrund der Vielzahl an dieser Stelle verzichtet und auf die Projektseite von FindBugs [FB] verwiesen.

3.4.2 Installation

Um FindBugs über die FindBugs-GUI, die Kommandozeile oder als Ant-Task verwenden zu können, ist es am einfachsten die Binärdistribution im *tar*- oder *zip*-Format von der Projektseite [FB] herunter zu laden. Anschließend kann diese Datei in einen beliebigen Ordner entpackt werden.

Um FindBugs mit der IDE Eclipse verwenden zu können, wird das FindBugs-Plug-In, wie unter 3.3.2 für das Checkstyle-Plug-In bereits beschrieben, installiert. Eine genaue Installationsbeschreibung für FindBugs befindet sich im Anhang auf Seite 57.

Für Maven existiert ebenfalls ein FindBugs-Plug-In, auf welches unter 3.4.4 näher eingegangen wird.

3.4.3 Benutzung

In dem Ordner *findbugs-1.3.9/bin* stehen zwei Skripte zum Starten von Findbugs zur Verfügung. Für Windows ist dies *findbugs.bat* und für Unixsysteme *findbugs*. Über diese Skripte kann FindBugs über die Kommandozeile genutzt werden oder die eigene GUI geöffnet werden. Am einfachsten ist dies unter Windows, indem die PATH-Variable um den Pfad zu diesem Skript ergänzt wird.

Bis auf die Benutzung über die Kommandozeile gilt für alle anderen Benutzungsmöglichkeiten, dass sogenannte *Filter Files* verwendet werden können. Hierdurch ist es z.B. möglich, bestimmte Klassen für bestimmte Überprüfungen auszuschließen. Diese Filter werden in Form von XML-Dateien definiert. Um hierrüber näheres zu erfahren, wird auf die Seite [FB]=>Manual=>“Filter Files“ verwiesen.

Als Beispiel zur Demonstration von FindBugs dient das Java-Programm „FindBugs_Demo.java“ (siehe Anhang Seite 57)

Benutzung über die Kommandozeile

Um Findbugs über die Kommandozeile zu nutzen, wird folgendes Kommando verwendet:

```
findbugs –textui Classarchivdatei oder einzelne Classdatei
```

Als `Classarchivdatei` können folgende Formate verwendet werden: *jar*, *war*, *ear*, *zip* oder *sar*.
Das o.a. Kommando ist das Minimum um FindBugs über die Kommandozeile ausführen zu können. Zusätzlich steht noch eine Reihe von möglichen Kommandozeilen-Optionen zur Verfügung. Auf eine Auflistung dieser Optionen wird an dieser Stelle verzichtet und stattdessen auf die Seite [FB] =>Manual => „Running FindBugs“ verwiesen.

Das Beispielprogramm *FindBugs_Demo.java* kann über folgendes Kommando durch FindBugs analysiert werden:

```
findbugs -textui path/to/FindBugs_Demo.class
```

Benutzung über die FindBugs-GUI

Die Benutzung über die FindBugs-GUI soll durch eine Analyse des JUnit-Jar-Archivs gezeigt werden.

Durch die Ausführung des FindBugs-Skripts *findbugs.bat* ohne Parameter wird folgende FindBugs-Oberfläche geöffnet:

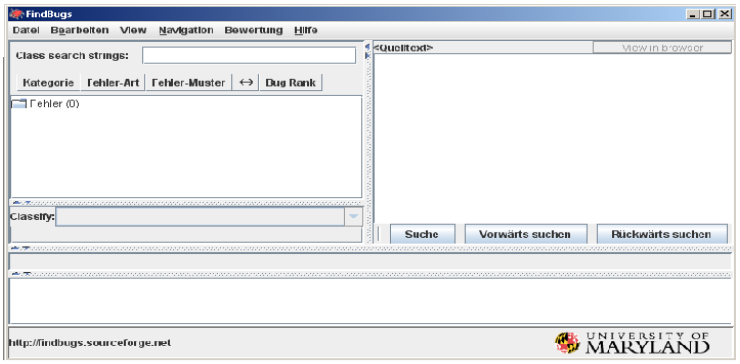

Abb. 3.8: FindBugs - GUI

Ein neues FindBugs-Project wird wie folgt angelegt:

Datei => Neues Projekt

Es erscheint folgendes Eingabefenster:

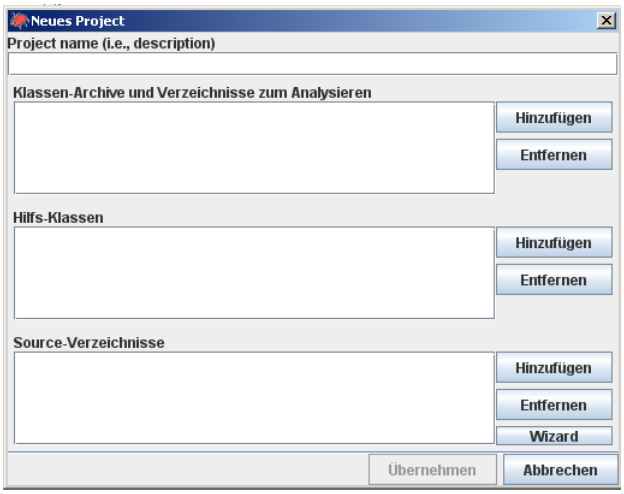

Abb. 3.9: Dialog zum Erstellen eines neuen FindBugs-Projekts

Unter *Project name* wird zunächst ein Projektname vergeben, z.B. „JUnit Analyse“. Als nächstes werden unter *Klassen-Archive und Verzeichnisse zum Analysieren* die zu analysierenden Klassen-Archive hinzugefügt, z.B. „junit-dep-4.8.1.jar“. Damit mögliche Bugs auch im Quellcode angezeigt werden können, kann man unter *Source-Verzeichnisse* die Verzeichnisse mit den Quellcodes hinzufügen, z.B. „junit-4.8.1-src.jar“. Sollten die zu analysierenden Klassendateien andere Klassen referenzieren, welche nicht in den unter *Klassen-Archive und Verzeichnisse* angegebenen Klassenarchiven oder Verzeichnissen enthalten sind, können diese unter *Hilfs-Klassen* hinzugefügt werden.

Wurde ein Projektname vergeben, alle benötigten Archive, Verzeichnisse und Dateien hinzugefügt und mit *Übernehmen* bestätigt, wird die Analyse durchgeführt. Abb. 3.4.3 zeigt die FindBugs-Oberfläche nach der Analyse von „junit-dep-4.8.1.jar“.

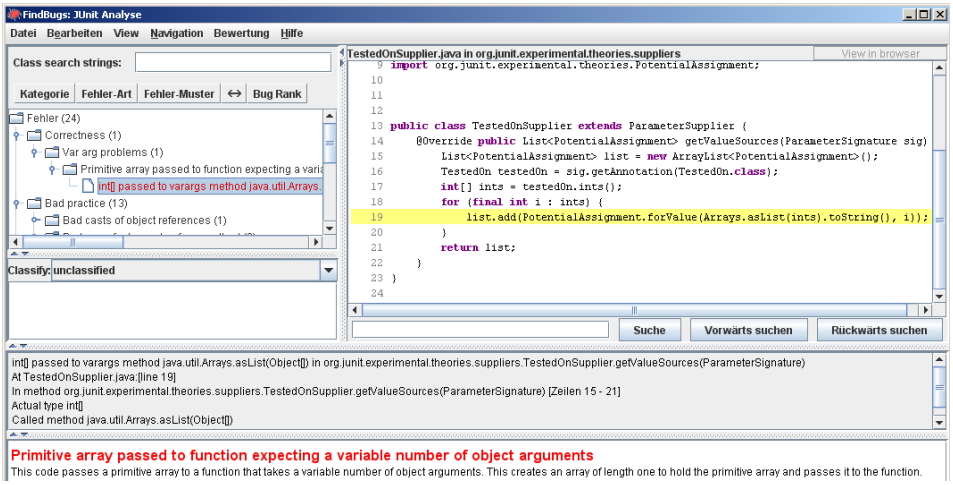

Abb. 3.10: FindBugs - GUI nach einer Analyse

In Abb. 3.10 ist in der oberen linken Hälfte die Auflistung der gefundenen, möglichen Bugs zu sehen. Diese Auflistung ist als Baumstruktur aufgebaut. In der oberen rechten Hälfte wird die passende Stelle, zu dem in dem Baum gewählten möglichen Bug, im entsprechenden

Quellcode markiert. Ganz unten in der Abbildung ist eine ausführliche Fehlerbeschreibung zu sehen.

Benutzung mit Eclipse

Um FindBugs mit Eclipse nutzen zu können, muss das FindBugs-Plug-In zunächst wie unter 3.4.2 beschrieben, installiert worden sein. Durch Auswahl von *Window => Preferences* kann man zu folgendem Dialog gelangen:

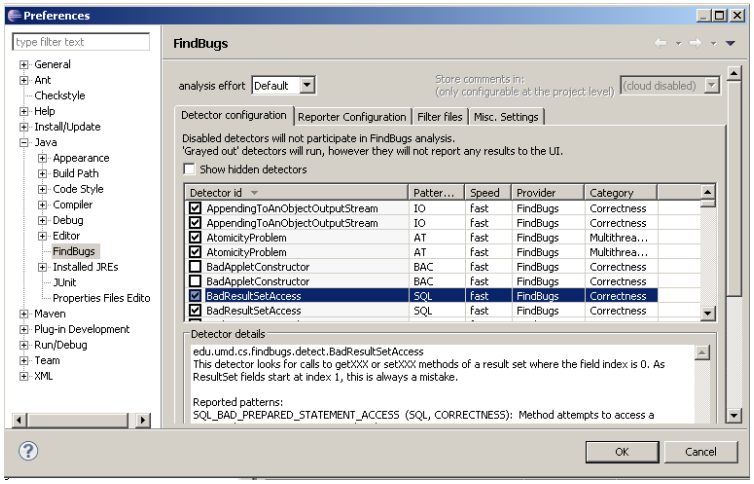

Abb. 3.11: Einstellungen-Dialog für FindBugs

Es können verschiedene Einstellungen für FindBugs vorgenommen werden. Z.B. können unter *Detector configuration* verschiedene Detectoren ausgewählt werden, die bei einer Überprüfung verwendet werden sollen. Unter *Reporter Configuration* kann z.B. festgelegt werden, welche Kategorien von Bugs angezeigt werden sollen. Unter *Filter files* können XML-Filter-Dateien ausgewählt werden. Und unter *Misc. Settings* können weitere Einstellungen vorgenommen werden, z.B. können eigene Detectoren hinzugefügt werden.
Um eine Analyse mit FindBugs vorzunehmen, klickt man mit der rechten Maustaste in dem *Package Explorer* von Eclipse auf ein Projekt, ein einzelnes Paket oder eine einzelne Java-Datei. Es erscheint ein Auswahlmenü, das u.a. den Punkt *Find Bugs* enthält. Über diesen Menüpunkt kann nun die Überprüfung gestartet werden (Abb. 3.12)

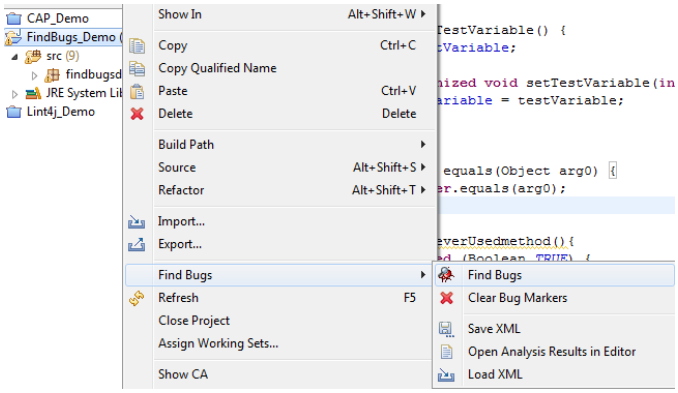

Abb. 3.12: Kontextmenü um FindBugs zu starten

Nach der Überprüfung werden die Ergebnisse in der FindBugs-Perspektive angezeigt, wenn diese über *Window => Open Perspective => Other => FindBugs* ausgewählt wurde:

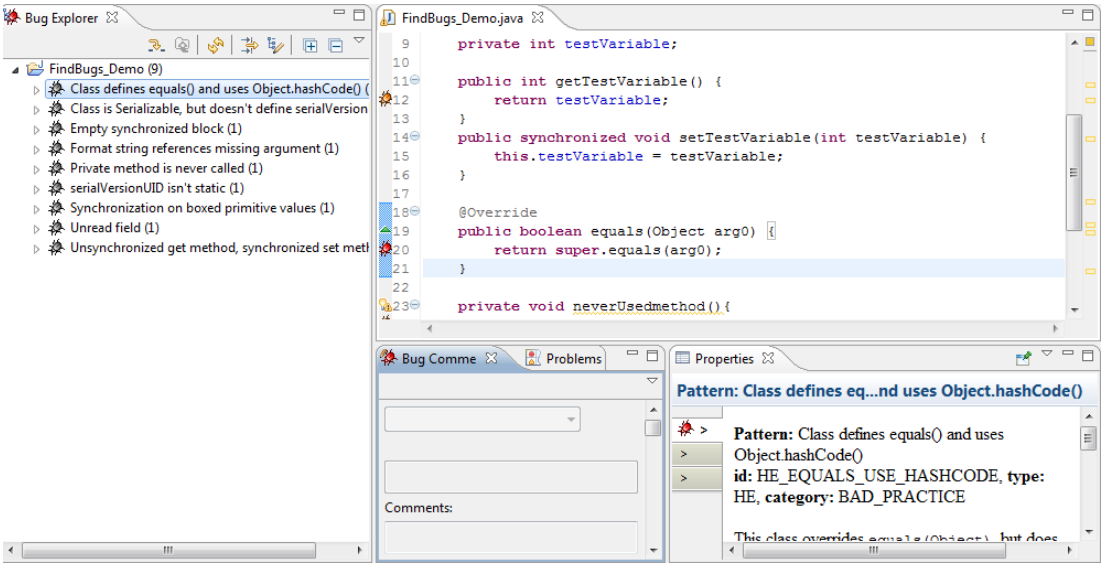

Abb. 3.13: FindBugs-Perspektive in Eclipse

Im linken Bereich werden die gefundenen, möglichen Bugs angezeigt. Durch einen Doppelklick auf diese wird die entsprechende Stelle im Quellcode angezeigt (rechte Seite). Außerdem werden mögliche Bugs im Quellcode auch durch die „Käfer“ am linken Rand des Editors angezeigt. Unten links ist eine genauere Beschreibung des Bugs gegeben.

3.4.4 Verwendung mit Ant und/oder Maven

Ant

FindBugs lässt sich durch eine *taskdef* –Deklaration in der von Ant benötigten build.xml-Datei in den Build-Prozess von Ant integrieren.

Anhand der folgenden XML-Datei soll beispielhaft die Verwendung von FindBugs mit Ant gezeigt werden.

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <project name="FindBugs-Ant" basedir=".">
3      <property name="findbugs.home"
4                value="path/to/findbugsinstallation"/>
5
6      <taskdef name="findbugs"
7               classname="edu.umd.cs.findbugs.anttask.FindBugsTask"
8               classpath="${findbugs.home}/lib/findbugs-ant.jar"/>
9
10     <target name="findbugs">
11           <findbugs home="${findbugs.home}"
12                     output="html"
13                     outputFile="FindBugs_Demo.html">
14                <class location="path/to/FindBugs_Demo.class"/
15           </findbugs>
16     </target>
17 </project>
```

In Zeile 3 wird eine Property angelegt, die den Pfad zur FindBugs-Installation enthält. In Zeile 6-8 erfolgt die *taskdef*-Deklaration. Hier wird festgelegt, welche Klasse genutzt werden soll, um FindBugs auszuführen und wo diese zu finden ist. In Zeile 10 wird ein Ziel für Ant definiert, welches die Ausführung von FindBugs beinhaltet. Mit dem Attribut `home` wird der Pfad zur FindBugs-Installation mitgeteilt. Die Attribute `output` und `outputFile` legen in diesem Fall fest, dass die FindBugs-Meldungen im HTML-Format in die Datei FindBugs_Demo.html geschrieben werden. Über das XML-Element `class location` wird der Pfad zu der class-Datei angegeben, welche überprüft werden sollen.

Das gezeigte Beispiel soll als ein kurzes Einführungsbeispiel dienen und zeigen, wie FindBugs mit Ant verwendet werden kann. Es existiert noch eine Reihe weiterer Parameter die für eine FindBugs-Task angegeben werden können. Darauf wird an dieser Stelle nicht näher eingegangen, sonder auf [FB]=>Manual=>“Using the FindBugs Ant task“ verwiesen.

Maven

Für Maven existiert ein FindBugs-Plug-In. Dieses kann auf folgende Weise in eine POM-Datei hinzugefügt werden:

```
<project>
  ...
   <reporting>
    <plugins>
      <plugin>
        <groupId>org.codehaus.mojo</groupId>
        <artifactId>findbugs-maven-plugin</artifactId>
        <version>2.3.2</version>
      </plugin>
    </plugins>
  </reporting>
       ...
</project>
```

Es können verschiedene Einstellungen für FindBugs vorgenommen werden. Auf diese Einstellmöglichkeiten wird nicht näher eingegangen, sondern auf [FBM] verwiesen.

3.5 PMD

Dieses Kapitel beschreibt das QS-Werkzeug PMD [PMD] und nennt die typischen Einsatzszenarien dieses Werkzeugs. Es wird gezeigt wie PMD installiert wird und welche Möglichkeiten der Benutzung es gibt. Zum Schluss wird beschrieben, wie PMD in Verbindung mit den Build-Management-Werkzeugen Ant und Maven verwendet werden kann.

3.5.1 Beschreibung und typische Einsatzszenarien

PMD ist ein in Java geschriebenes Open Source Programm zur statischen Quellcodeanalyse. Es unterliegt der *Berkeley Software Distribution* – Lizenz. Für PMD existiert kein offiziell ausgeschriebener Name. Es überprüft den Java-Quellcode auf potentielle Probleme, wie mögliche Bugs, ungenutzten, suboptimalen oder unnötig komplizierten Code. Außerdem existiert mit dem *Copy/Paste-Detector* (CPD) eine Erweiterung zum Aufspüren von dupliziertem Code.

PMD kann über die Kommandozeile verwendet oder in die Build-Prozesse von Ant und Maven integriert werden. Außerdem existieren 15 Plug-Ins für verschiedene IDEs. Es wird in Kap. 3.5.3 die Verwendung des Plug-Ins für Eclipse näher betrachtet.

PMD überprüft den Quellcode anhand verschiedener Regeln (Rules). In der Version 4.2.5, welche in dieser Arbeit betrachtet wird, stehen insgesamt 257 Regeln, die in 24 sogenannte *Rulesets* gegliedert sind, zur Verfügung. Folgende Rulesets existieren:

Android:	Enthält Regeln im Zusammenhang mit dem Android SDK.
Basic:	Enthält Basisregeln, die z.B. leere catch-Blöcke oder leere if-Anweisungen enthalten.
Braces:	Enthält Regeln bezüglich der Verwendung von Klammern. Z.B. sollen if- oder while-Anweisungen von Klammern eingeschlossen sein.
Code Size:	Enthält Regeln zur Überprüfung von einzelnen „Code-Größen“, wie z.B. lange Parameterlisten oder lange Methoden, bezogen auf die Zeilenzahl.
Clone:	Enthält Regeln bezüglich der clone()-Methode, z.B. sollte die clone()-Methode implementiert werden, wenn die Klasse das Cloneable-Interface implementiert.
Controversial:	Dieser Regelsatz enthält Regeln, die kontrovers diskutiert werden können. Z.B. ob es in einer Methode nur eine return-Anweisung geben darf.
Coupling:	Durch diese Regeln kann die Kopplung zwischen Objekten begrenzt werden, indem z.B. die Anzahl der import-Anweisungen begrenzt wird.
Design:	Durch diese Regeln können fragwürdige Codedesignstellen gefunden werden, wie z.B. sehr tief verschachtelte If-Anweisungen.
Finalizers:	Diese Regeln beziehen sich auf die Verwendung der finalize()-Methode und sich daraus ergebene, mögliche Probleme, wie z.B. eine leere finalize()-Methode oder der explizite Aufruf der finalize()-Methode.
Import Statements:	Diese Regeln beziehen sich auf import-Anweisungen und finden z.B. ungenutzte Imports oder duplizierte Imports.
J2EE:	Dieser Regelsatz bezieht sich auf das „Java Enterprise Edition“-Framework und enthält z.B. Regeln bezüglich Namenskonventionen.

Javabeans:	Dieser Regelsatz enthält zwei Regeln bezüglich JavaBeans. JavaBeans müssen serialisierbar sein, somit müssen Member-Variablen transient oder static sein oder getter- und setter-Methoden haben. Außerdem muss eine SerialVersionUID vorhanden sein.
JUnit Tests:	Diese Regeln beziehen sich auch Probleme, die im Zusammenhang mit JUnit-Tests auftreten können. Z.B. sollten JUnit-Tests Assert-Anweisungen enthalten.
Logging (Java):	Diese Regeln finden fragwürdige Verwendungen des Loggers. Z.B. wenn in einer Klasse mehr als ein Logger definiert wird.
Logging (Jakarta):	Diese Regeln finden fragwürdige Codestellen bei der Nutzung des Jakarta-Logging-Frameworks.
Migrating:	Enthält Regeln über die Migration von einer JDK-Version zu einer anderen. Es sollte z.B. überlegt werden eine Hashtable gegen die neuere Map zu tauschen.
Naming:	Dieser Regelsatz enthält Regeln bezüglich der Namensvergabe. Z.B. können Variablen mit sehr kurzen Namen (ein Buchstabe) oder auch mit sehr langen Namen gefunden werden. Außerdem gibt es Regeln bezüglich Namenskonventionen.
Optimizations:	Diese Regeln befassen sich mit Optimierungen. Z.B. sollte zur String-Verknüpfung anstatt += ein StringBuffer-Objekt und dessen append()-Methode genutzt werden.
Strict Exceptions:	Dieser Regelsatz enthält Regeln bezüglich der Verwendung von Exceptions. Z.B. sollten keine NullPointerExceptions geworfen werden, da dies verwirrend ist, da diese normalerweise von der VM geworfen werden.
Strings:	Diese Regeln beziehen sich auf die Verwendung von Strings. Es wird z.B. ein überflüssiger Aufruf der toString()-Methode an einem String gefunden.
Sun Security:	Hier werden zwei Regeln definiert. Es sollte z.B. nicht ein internes Array einer Klasse aus einer Methode zurückgegeben werden, sondern eine Kopie. Ebenso sollte ein als Parameter übergebenes Array nicht direkt gespeichert werden, sondern eine Kopie.
Unused Code:	Es werden nicht verwendete private Datenfelder oder Methoden und nicht verwendete lokale Variablen gefunden.
Java Server Pages:	Dieser Regelsatz enthält Regeln bezüglich Java Server Pages. Z.B. sollten Style-Informationen in einer CSS-Datei und nicht in einer JSP enthalten sein.

Eine ausführliche Beschreibung aller Regeln ist unter [PMD] zu finden.

3.5.2 Installation

Zur Verwendung von PMD über die Kommandozeile oder als Ant-Task kann man über [PMD] auf die Downloadseite von PMD gelangen. Hier steht die Datei *pmd-bin-4.2.5.zip* zum Download zur Verfügung. Nach dem Download kann diese Datei in einen beliebigen Ordner entpackt werden.

Die Installation des PMD-Eclipse-Plug-Ins erfolgt wie bereits unter 3.3.2 für das Checkstyle-Plug-In beschrieben. Eine konkrete Installationsbeschreibung für das PMD-Plug-In befindet sich im Anhang auf Seite 58.

Ein Plug-In für das Build-Management-Tool Maven existiert ebenfalls. Hierauf wird in Kap. 3.5.4 näher eingegangen.

3.5.3 Benutzung

Um Regeln, die bei der Überprüfung mit PMD verwendet werden sollen, auszuwählen kann ein sogenanntes RuleSet in Form einer XML-Datei definiert werden. Im Folgenden ist ein solches RuleSet (RuleSet_demo.xml) gezeigt, welches als Beispiel für die Benutzung von PMD dient.

```
1 <?xml version="1.0"?>
2 <ruleset name="RuleSet_demo"
3    xmlns="http://pmd.sf.net/ruleset/1.0.0"
4    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5    xsi:schemaLocation="http://pmd.sf.net/ruleset/1.0.0
6                        http://pmd.sf.net/ruleset_xml_schema.xsd"
7  xsi:noNamespaceSchemaLocation="http://pmd.sf.net/ruleset_xml_schema.xsd">
8
9  <rule ref="rulesets/basic.xml/EmptyIfStmt"/>
10 <rule ref="rulesets/basic.xml/EmptySynchronizedBlock"/>
11
12 <rule ref="rulesets/design.xml/AvoidDeeplyNestedIfStmts">
13    <properties>
14          <property name="problemDepth" value="2"/>
15    </properties>
16 </rule>
17
18 <rule ref="rulesets/naming.xml/MethodNamingConventions"/>
19
20 <rule ref="rulesets/codesize.xml/ExcessiveParameterList">
21     <properties>
22        <property name="minimum" value="5"/>
23     </properties>
24 </rule>
25</ruleset>
```

Das Hauptelement der XML-Datei ist das `ruleset`-Element. Als Unterelemente werden dann die einzelnen Regeln als `rule`-Elemente, die bei einer Quellcodeüberprüfung berücksichtigt werden sollen, angeben. Eine Regel wird über den `ref`-Parameter ausgewählt. Im Beispiel werden immer einzelne Regeln ausgewählt. Sollen z.B. alle Regeln des Naming-Rulesets bei ein Überprüfung berücksichtigt werden, kann dies über folgende Definition geschehen: `<rule ref="rulesets/naming.xml/>`, es wird also auf eine explizite Angabe einer einzelnen Regel verzichtet. In dem obigen Beispiel werden aus dem Basic-Ruleset die Regeln „EmptyIfStmt“ und „EmptySynchronizedBlock“ (Zeile 9 und 10) ausgewählt. Aus dem Design-RuleSet wird die Regel „AvoidDeeplyNestedIfStmts“ verwendet und über das Property „problemDepth“ wird eine maximale Tiefe der If-Verschachtelung von 2 festgelegt (Zeile 12-16). Aus dem Naming-Ruleset wird die Regel „MethodNamingConventions“ gewählt (Zeile 18), welche besagt, dass ein Methodenname mit einem kleinen Buchstaben beginnen sollte und keine Unterstriche im Namen vorkommen sollen. In Zeile 20-24 wird

aus dem Code Size-Ruleset die Regel „ExcessiveParameterList“ ausgewählt und über die Property „minimum“ wird festgelegt, dass die Parameterliste einer Methode auf vier Parameter begrenzt wird.

Benutzung über die Kommandozeile

Die Benutzung von PMD über die Kommandozeile kann durch folgendes Kommando geschehen:

```
pmd filename|jar- oder zip-Datei mit Quellcodedateien|Verzeichnis
      report format ruleset file
```

PMD wird in diesem Fall über das Skript pmd.bat ausgeführt. Der erste Paramater ist entweder eine einzelne Java-Datei, eine jar- oder zip-Datei, die die Quellcode-Dateien enthält, oder es wird ein Verzeichnis mit den Quellcode-Dateien angeben. Wird ein Verzeichnis als Parameter verwendet, werden auch die Java-Dateien in dessen Unterverzeichnissen überprüft. Der zweite Parameter `report format` legt das Format der durch PMD erzeugten Meldungen fest. Es stehen die Formate „text“, „xml“, „html“ und „nicehhtml“ zur Verfügung. Der dritte Parameter gibt ein RuleSet an, welches zur Überprüfung verwendet werden soll.

Um also als Beispiel die Java-Datei „PMDDemo.java“ (siehe Anhang Seite 58) mit der oben gezeigten „RuleSet_demo.xml“-Datei zu überprüfen und die Ausgabe in eine html-Datei zu schreiben, kann folgendes Kommando verwendet werden:

```
pmd PMDDemo.java html RuleSet_demo.xml > PMD_Ausgabe.html
```

Es können noch weitere Parameter übergeben werden. Hierzu wird auf [PMD] verwiesen. Die drei oben genannten Parameter müssen als Minimum angeben werden.

Benutzung mit Eclipse

Um PMD in Verbindung mit Eclipse nutzen zu können, muss das PMD-Plug-In wie unter 3.5.2 beschrieben installiert worden sein. Ist dies der Fall kann man über *Window => Preferences* zu dem folgenden Dialog kommen:

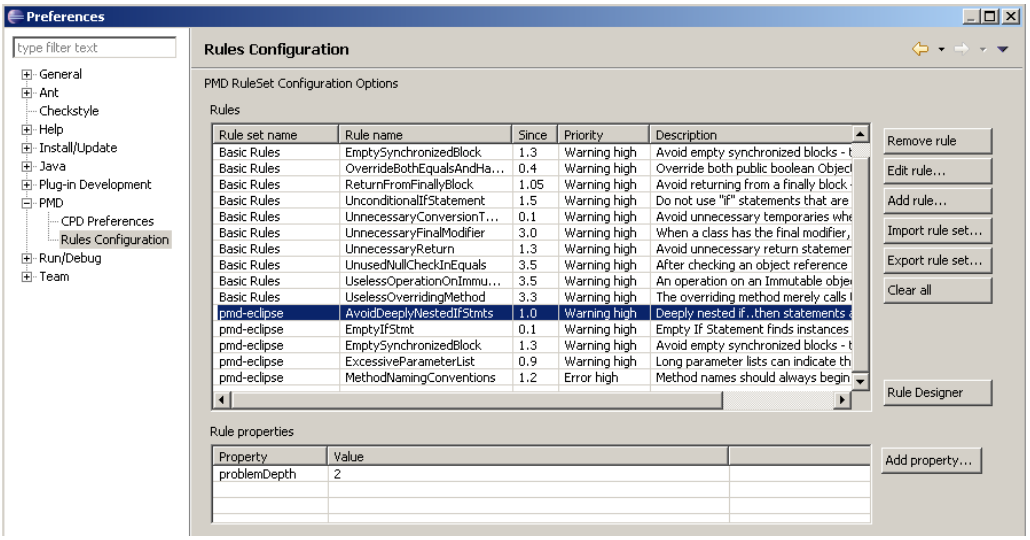

Abb. 3.14: Dialog zur Konfiguration von PMD Regeln unter Eclipse

In Abb. 3.14 ist der Dialog zur Auswahl von Regeln, die von PMD für eine Überprüfung von Java-Quellcode verwendet werden sollen. In dem mittleren Bereich unter „Rules“ kann man die momentan verwendeten Regeln sehen. Die unteren fünf Regeln sind die aus dem Eingangsbeispiel der „RuleSet_demo.xml“- Datei. Sie wurden über den Button *Import rule set...* und Auswahl der XML-Datei hinzugefügt. Damit bei einer Überprüfung nur diese fünf Regel beachtet werden, kann man die anderen Regeln entweder über den Button *Remove rule* einzeln entfernen werden oder man klickt vor dem Import der Regeln aus der XML-Datei auf den Button *Clear all*.

Eine Überprüfung des Quellcodes kann dann über folgendes Kontextmenü durchgeführt werden:

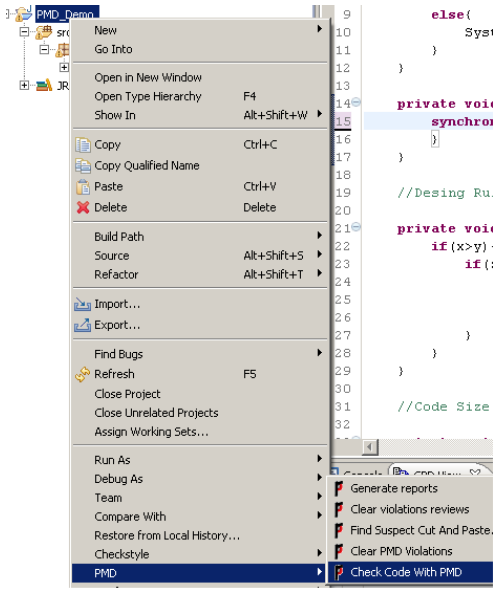

Abb. 3.15: Kontextmenü zum Ausführen von PMD

Die PMD-Meldungen werden daraufhin im Eclipse-Editor angezeigt:

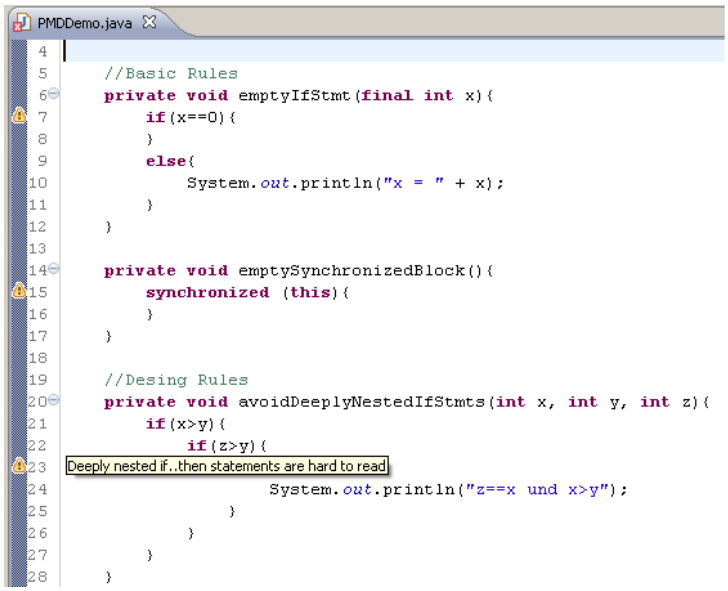

Abb. 3.16: PMD-Meldungen im Eclipse-Editor

Zu erkennen sind die Meldungen an den gelben Dreiecken mit den Rufzeichen. Befindet sich der Mauszeiger auf einem dieser Dreiecke wird die konkrete Meldung angezeigt. Außerdem können über *Window => Show View => Others* auch vier eigene Views von PMD betrachtet werden:

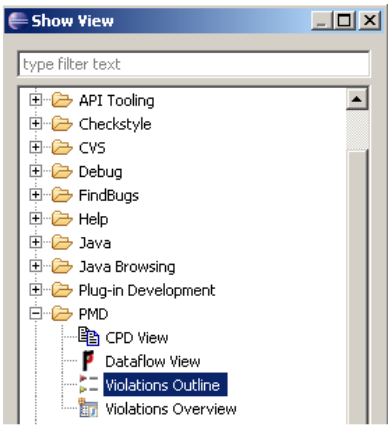

Abb. 3.17: PMD – Views

Die Views „Violations Outline" und „Violations Overview" zeigen ebenfalls die PMD-Meldungen an. Die „CPD View" ist eine View für den Copy-/Paste – Detector. Mit der „Dataflow View" kann ein Datenflussgraph für eine ausgewählte Methode angezeigt werden.

3.5.4 Verwendung mit Ant und/oder Maven

Ant

Durch eine *taskdef*-Deklaration in der von Ant verwendeten build.xml-Datei kann PMD in den Ant-Build-Prozess eingebunden werden. Im Folgenden ist eine solche build-Datei gezeigt.

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <project name="Ant_PMD" basedir=".">
3
4      <property name="pmd.home" value="C:/Programme/Java/pmd-4.2.5"/>
5
6      <taskdef name="pmd" classname="net.sourceforge.pmd.ant.PMDTask"
7                          classpath="${pmd.home}/lib/pmd-4.2.5.jar"/>
8
9      <target name="pmd">
10           <pmd>
11                 <ruleset>RuleSet_demo.xml</ruleset>
12                 <fileset dir="src" includes="*.java"/>
13                 <formatter type="html" toFile="PMD_Meldungen.html"/>
14           </pmd>
15     </target>
16</project>
```

In Zeile 4 wird zunächst eine Property definiert, die den Pfad zur PMD-Installation enthält. Zeile 6-7 enthält die *taskdef*-Deklaration. Hier wird ein Name, welche Klasse für die Überprüfung und wo sich diese Klasse befindet angegeben. In Zeile 9-14 wird dann ein Ant-Ziel definiert. Über das XML-Element `ruleset` wird ein Regelsatz, nach dem der Quellcode überprüft werden soll, angegeben. Das Element `fileset` legt fest welche Dateien überprüft werden sollen und wo sich diese befinden. Schließlich wird über `formatter` das Ausgabeformat und eine Datei in die die Meldungen geschrieben werden sollen angegeben.

Für weitere Einstellmöglichkeiten und Parameter bezüglich PMD und Ant wird auf [PMD] verwiesen.

Maven

Für Maven existiert ein PMD-Plug-In. Dieses kann auf folgende Weise in eine POM-Datei hinzugefügt werden:

```
<project>
   ...
   <reporting>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-pmd-plugin</artifactId>
      </plugin>
    </plugins>
  </reporting>
  ...
</project>
```

Es können verschiedene Einstellungen für PMD vorgenommen werden. Auf diese Einstellmöglichkeiten wird nicht näher eingegangen, sondern auf [PMDM] verwiesen.

3.6 Lint4j

In diesem Kapitel wird das QS-Werkzeug Lint4j [LJ] betrachtet. Es folgt zunächst eine Beschreibung von Lint4j und die typischen Einsatzszenarien werden genannt (Kap. 3.6.1). Danach wird der Installationsprozess beschrieben (Kap. 3.6.2), bevor in Kap. 3.6.3 auf die verschiedenen Benutzungsmöglichkeiten von Lint4j eingegangen wird. In Kap. 3.6.4 wird abschließend die Verwendung von Lint4j im Zusammenhang mit den Build-Management-Werkzeugen Ant und Maven erläutert.

3.6.1 Beschreibung und typische Einsatzszenarien

Der Name „Lint4j“ steht für „Lint for Java“. Lint4j überprüft Java-Quell- und Bytecode. Es findet z.B. Probleme im Zusammenhang mit Threads, Locking und Performance. Dies geschieht u.a. durch die Analyse eines Datenflussgraphen oder eines Lock-Graphen.

Lint4j kann über die Kommandozeile oder als Ant-Task verwendet werden. Außerdem gibt es Plug-Ins für Eclipse und Maven.

Die Probleme, die von Lint4j entdeckt werden können, sind in folgende elf Kategorien eingeteilt:

Architectual problems:	In dieser Kategorie werden Probleme bzgl. der Architektur gefunden. Z.B. eine abstrakte Klasse, die keine abstrakten Methoden definiert oder eine Klasse, die ein Interface implementiert, welches schon von einer Oberklasse implementiert wurde.
Contracts defined in the Java core API:	In dieser Kategorie werden Probleme im Zusammenhang mit der Klasse java.lang.Object behandelt. Z.B. ein expliziter Aufruf der finalize()-Methode oder es wird die clone()-Methode überschrieben, ohne dass das Cloneable-Interface implementiert wird.
Java Language Constructs:	Hier werden Probleme gefunden, wie z.B. return-Anweisungen in einem finally-Block, leere catch-Blöcke oder die Verwendung von switch ohne default-Anweisung.
Immature code:	In dieser Kategorie überprüft Lint4j den Code auf Methode und die Nutzung bestimmter Klassenvariablen, die üblicherweise in der Entwicklungsphase zum debuggen genutzt werden, wie z.B. System.err oder System.out.
Performance:	Hier werden Codestellen gefunden, die zu Performance-Einbußen führen können. Z.B. die Nutzung des „+=“-Operators zur Zusammensetzung von Strings anstatt der Verwendung der append()-Methode eines StringBuffers.
Code portability:	Hier werden Codestellen gefunden, die die Portabilität des Codes einschränken, wie z.B. hartcodierte Newline-Zeichen.

The Serialization and Externalization contracts:	Diese Kategorie beschäftigt sich mit Problemen bzgl. Serialisierung und Externalisierung. Z.B. wenn eine Klasse die das Externalizable-Interface implementiert keinen Default-Konstruktor besitzt.
Suspicious coding and likely bugs:	In dieser Kategorie geht es um verdächtige Codestellen und mögliche Bugs. Z.B. werden Zuweisungen, die keinen Effekt haben, wie x=x gefunden.
Java Synchronization and Scalability:	Diese Kategorie beschäftigt sich mit Problemen im Bereich Synchronisation und Skalierbarkeit. Z.B. werden Codefragmente gefunden die mehr als drei Sperren zur selben Zeit enthalten.
Coding patterns that impact readability and code size:	In dieser Kategorie werden return-Anweisungen in void-Methoden gefunden und eine if-Abfrage bestehend aus zwei Bedingungen, wobei die erste auch noch den Negations-Operator verwendet. Eine solche if-Abfrage ist schwer lesbar.
Violations of the EJB specification:	In dieser Kategorie geht es um Verletzungen der EJB Spezifikation für EJB 2.1. Z.B. wird entdeckt, wenn eine Bean-Klasse keine Default-Konstruktor besitzt.

Eine ausführliche Auflistung aller Probleme, die von Lint4j gefunden werden können, wird auf [LJ] verwiesen.

3.6.2 Installation

Zur Installation von Lint4j steht unter [LJ] ein Archiv für Unix-Systeme und ein Archiv für Windows-Systeme zum Download bereit. Hier soll die Installation der Windows-Version betrachtet werden. Das Archiv *lint4j-0.9.1.zip* wird zunächst heruntergeladen und in ein beliebiges Verzeichnis entpackt. Es empfiehlt sich die Umgebungsvariable PATH um einen Eintrag des Pfades zu dem Ordner *lint4j-0.9.1/bin* zu ergänzen. Hierdurch ist es nun möglich Lint4j über die im bin-Ordner befindliche Batch-Datei über die Kommandozeile zu nutzen.

Die Installation des Eclipse-Plug-Ins für Lint4j erfolgt auf die gleiche Weise wie die Installation des Checkstyle-Plug-Ins (siehe 3.2.2). Die ausführliche Installationsanleitung des Lint4j-Plug-Ins befindet sich im Anhang auf Seite 59.

Auf das Maven-Plug-In wird in Kap. 3.6.4 näher eingegangen.

3.6.3 Benutzung

Die Benutzung von Lint4j soll an einem Beispiel (Lint4j_Demo.java) gezeigt werden. In diesem Programm sind insgesamt sieben kritische Codestellen, die von Lint4j gefunden werden. Das Lint4j_Demo.java-Programm befindet sich im Anhang auf Seite 59.

Benutzung über die Kommandozeile

Ist der Pfad zu dem Ordner *lint4j.0.9.1/bin* wie unter 3.6.2 beschrieben der Umgebungsvariablen PATH hinzugefügt worden, kann Lint4j über folgendes Kommando verwendet werden:

```
lint4j [-J vmoptions] [-v level] -sourcepath path[;path]* [-classpath
path[;path]*] [-exclude packagename]* [-class class[:class]*] [packagename |
filename]+
```

Die Bedeutung der Kommandozeilen-Parameter:

`-J:` Über diesen Parameter werden Optionen für die virtuelle Maschine übergeben, z.B. –Xmx um die Speichergröße zu erhöhen. Wird der Parameter –J angegeben, muss er an erster Stelle stehen. (optional)

`-v:` Über diesen Parameter wird der Umfang der Meldungen von Lint4j eingestellt. Der Bereich geht von 1-5. Der Defaultwert ist 3. (optional)

`-sourcepath:` Über diesen Parameter werden die jar-Dateien oder die Verzeichnisse, die die Quell- und class-Dateien enthalten, welche überprüft werden sollen angegeben. Mehrere Dateien oder Verzeichnisse können über ein ; getrennt angegeben werden.

`-classpath:` Über diesen Parameter werden zip-Dateien, jar-Dateien oder Ordner angeben, die Klassen enthalten, die von den Source-Dateien referenziert werden. Diese referenzierten Klassen werden dann bei der Überprüfung nicht berücksichtigt. (optional)

`-exclude:` Über diesen Parameter werden Pakete angegeben, die von der Überprüfung ausgenommen werden sollen. (optional)

`-class:` Über diesen Parameter werden die Klassennamen von den Klassen angegeben, welche überprüft werden sollen.

Das Beispielprogramm Lint4j_Demo.java kann durch folgendes Kommando getestet werden, falls eine Umgebungsvariable für Lint4j vorhanden ist:

```
lint4j -v 5 -sourcepath src lint4jdemo.*
```

Dabei befinden sich im Ordner „src“ die Dateien *Lint4j_Demo.java* und *Lint4j_Demo.class*. *lint4jdemo* ist das Paket, in dem sich die Java-Datei befindet.

Benutzung mit Eclipse

Über einen Rechtsklick auf ein Projekt im Projektexplorer öffnet sich ein Kontextmenü. Wählt man aus dem Menü den Eintrag „Properties“ erhält man folgenden Dialog, sofern das Eclipse-Plug-In für Lint4j wie unter3.6.2 beschrieben installiert wurde:

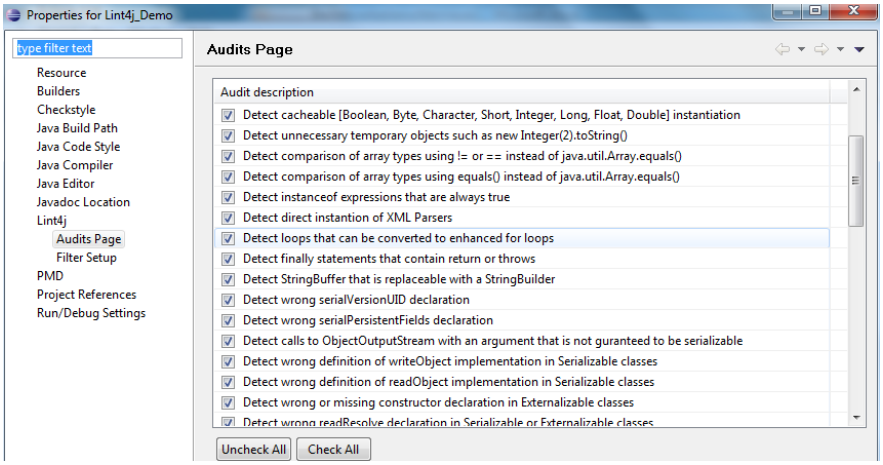

Abb. 3.18: Auswahl-Dialog Lint4j

Über den Dialog, wie er in Abb. 3.18 dargestellt ist, kann ausgewählt werden, welche Probleme von Lint4j gefunden werden sollen.

Die eigentliche Überprüfung wird über einen Rechtsklick auf das zu prüfenden Projekt im Package-Explorer und Auswahl von *Lint4j => Audit* gestartet (siehe Abb. 3.19). Dabei werden die im Auswahl-Dialog, wie in Abb. 3.18 zu sehen, verwendet.

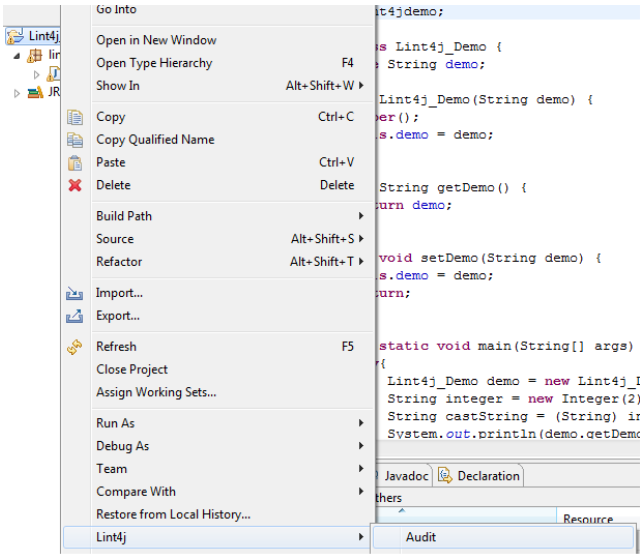

Abb. 3.19: Start von Lint4j in Eclispe

Die von Lint4j gefundenen Probleme werden dann an den entsprechenden Stellen im Quellcode im Editor und in der Eclipse-View *Problems* angezeigt (siehe Abb. 3.20).

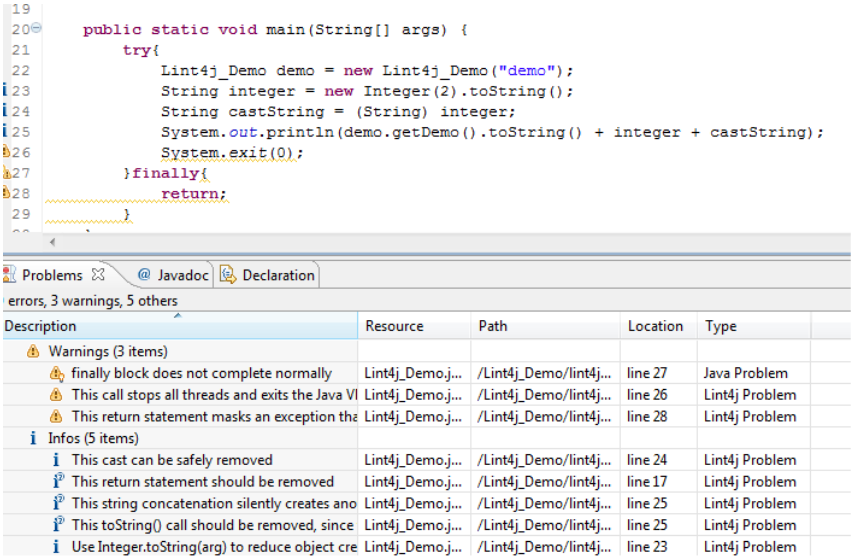

Abb. 3.20: Anzeige der von Lin4j gefunden Probleme in Eclipse

3.6.4 Verwendung mit Ant und/oder Maven

Durch eine *taskdef*-Deklaration in der von Ant verwendeten build.xml-Datei kann Lint4j in den Ant-Build-Prozess eingebunden werden. Im Folgenden ist eine solche build-Datei für das Beispielprogramm *Lint4j_Demo.java* gezeigt.

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <project name="Lint4j_Demo" basedir=".">
3
4     <property name="distdir"
5               location="C:/Users/Jens/Programme/Java/lint4j-0.9.1"/>
6     <property name="src.path"
7               location="C:/Users/Jens/BA_QSWerkzeuge/Lint4j/Ant/src"/>
8
9     <taskdef name="lint4j"
10             classname="com.jutils.lint4j.ant.Lint4jAntTask">
11          <classpath>
12                <pathelement location="${distdir}/jars/lint4j.jar"/>
13          </classpath>
14    </taskdef>
15
16    <target name="lint4j_demo">
17      <lint4j>
18          <sourcepath>
19            <pathelement path="${src.path}"/>
20          </sourcepath>
21
22          <formatters>
23              <formatter type="xml" toFile="lint4j_demo_ausgabe.xml"/>
24          </formatters>
25      </lint4j>
26    </target>
27 </project>
```

In den Zeilen 4,5,6 und 7 werden zunächst zwei Properties definiert. Die erste Property enthält den Pfad zur Lint4j-Installation. Die zweite Property enthält den Pfad zu der

*Lint4j_Demo.*java- und *Lint4j_Demo.class- Datei*. Zeile 9-14 enthält die *taskdef*-Deklaration. Hier wird der Name der Klasse, welche für die Überprüfung verwendet wird, angegeben. In Zeile 16-26 wird das Ant-Ziel für Lint4j definiert. Hier wird über das `sourcepath`- Element angeben, wo sich die zu überprüfenden Dateien befinden. Über das `formatters`-Element wird das Ausgabeformat von Lint4j und eine Datei in die die Meldungen geschrieben werden sollen, festgelegt.

Für weitere optionale Parameter, die für Lint4j im Zusammenhang mit Ant im Build-File angeben werden können wird auf [LJ] verwiesen.

Maven

Das Lint4j-Plug-In für Maven kann durch Hinzufügen von folgendem Code in die von Maven verwendete pom.xml-Datei installiert werden:

```
<dependency>
  <groupId>lint4j</groupId>
  <artifactId>jutils-lint4j-plugin</artifactId>
  <version>1.3.1</version>
  <type>plugin</type>
</dependency>
```

Der Report-Abschnitt in der pom.xml muss wie folgt ergänzt werden:

```
<report>jutils-lint4j-plugin</report>
```

In die *maven.properties*-Datei muss folgende Zeile hinzugefügt werden:

```
maven.repo.remote=http://www.ibiblio.org/maven,http://www.jutils.com/maven
```

3.7 CAP

In diesem Kapitel wird das QS-Werkzeug *Code Analyse Plugin* (CAP) [CAP] vorgestellt. In Kap. 3.7.1 wird zunächst das Werkzeug beschrieben und erläutert zu welchem Zweck es eingesetzt werden kann. In Kap. 3.7.2 wird gezeigt, wie CAP installiert wird und in Kap. 3.7.3 wird die Benutzung erklärt.

3.7.1 Beschreibung und typische Einsatzszenarien

Das Werkzeug *Code Analyse Plugin* wurde an der Hochschule Reutlingen von Johannes Schneider und Matthias Mergenthaler entwickelt. Bei CAP handelt es sich um ein Plug-In für Eclipse. Aus diesem Grund kann es auch nicht, wie es bei den in den Kap. 3.3 -3.6 vorgestellten Werkzeugen der Fall war, über die Kommandozeile oder im Zusammenhang mit Ant oder Maven verwendet werden. In dieser Arbeit wird die zurzeit aktuelle Version 1.2.0 betrachtet. Diese steht unter der *GNU General Public License*.

Durch eine Quellcode-Analyse mittels CAP werden Kennzahlen ermittelt, durch die Schwächen in der Software-Architektur erkannt werden. Durch eine Auswertung der erkannten Schwächen und einer anschließenden Überarbeitung des Software-Designs kann die Wartbarkeit und Wiederverwendbarkeit von Java-Paketen erhöht werden. Zur Darstellung der von CAP ermittelten Kennzahlen bietet das Plug-In eine eigene Eclipse-Perspektive.

Im Folgenden werden die von CAP ermittelten Kennzahlen erläutert:

Abkürzung	Name	Bedeutung
CC	Concrete Class Count	Anzahl der konkreten Klassen in einem Paket
AC	Abstract Class (and Interface) Count	Anzahl der abstrakten Klassen und Interfaces in einem Paket
A	Abstractness (0-1)	(Anzahl abstrakter Klassen)/(Anzahl aller Klassen) Je höher der Wert, desto größer der Anteil an Interfaces und abstrakter Klassen im Paket. A=0: Keine abstrakten Klassen oder Interfaces A=1: Nur abstrakte Klassen oder Interfaces
Ca	Afferent Couplings	Afferente (eingehende) Kopplungen (used by) Anzahl der Pakete, die von diesem Paket abhängig sind bzw. dieses Paket benutzen.
Ce	Efferent Couplings	Efferente (ausgehende) Kopplung (depending on) Anzahl der Pakete, von denen dieses Paket selbst abhängig ist bzw. die dieses Paket selbst benutzt.

Abkürzung	Name	Bedeutung
I	Instability (0-1)	(Ce)/(Ce+Ca) Verhältnis der ausgehenden Kopplungen zu der Summe der aller Kopplungen I=0: komplett stabil. Keine Abhängigkeiten von anderen Paketen. Unempfindlich gegen Änderungen an anderen Paketen. I=1: komplett instabil. Paket reagiert sehr empfindlich auf Änderungen in anderen Paketen.
D	Distance from the Main Sequence (0-1)	Gibt den Abstand von der idealisierten Hauptsequenz D=A+I-1 an und damit die Balance zwischen Abstraktheit und Stabilität. Optimale Pakete sind vollständig abstrakt und stabil oder konkret und instabil.
C	Cycles	Zyklische Abhängigkeiten von Pakete

3.7.2 Installation

Die Installation von CAP erfolgt auf die gleiche Weise wie die Installation anderer Eclipse-Plug-Ins, die schon in vorherigen Kapiteln beschrieben wurden. Eine konkrete Installationsanleitung für CAP befindet sich im Anhang auf Seite 60.

CAP ist von zwei Plug-Ins abhängig. Dieses sind *JFreeGraph* und *GEF*, welche ebenfalls über den üblichen Weg der Eclipse-Plug-In – Installation installiert werden können.

3.7.3 Benutzung

Durch einen Rechtsklick auf ein Projekt im Projektexplorer kann aus dem daraufhin erscheinenden Kontextmenü der Eintrag „Show CA“ gewählt werden. Man erhält einen Dialog zur Auswahl des Projekts welches von CAP analysiert werden soll. Nach der Analyse wird automatisch in die CAP-Perspektive (Abb. 3.21) gewechselt.

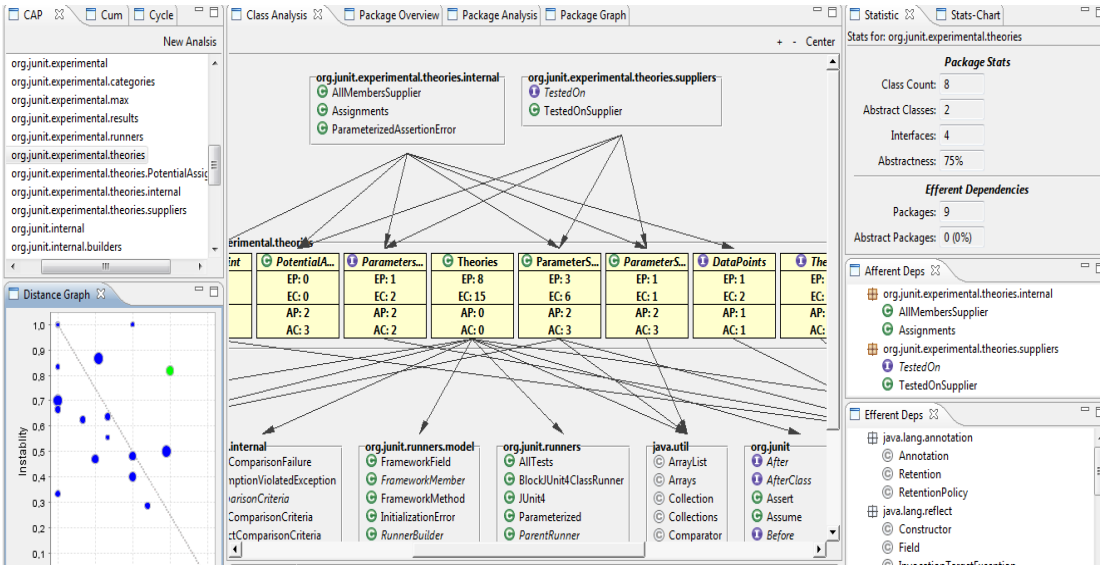

Abb. 3.21: CAP Eclipse-Perspektive

Als Beispiel wurde der Quellcode von JUnit analysiert. In Abb. 3.21 ist sind oben links die im JUnit-Projekt vorhandenen Pakete aufgelistet. Aktuell ist das Paket *org.junit.expermiental.theories* ausgewählt. Unten links ist der *Distance Graph* zu sehen. Die graue Linie ist die Ideallinie. Das aktuell ausgewählte Paket wird als grüner Punkt dargestellt, die übrigen als blaue Punkte. Unten rechts sind unter *Afferent Deps* bzw. *Efferent Deps* die Pakete aufgeführt, die das aktuell ausgewählte Paket nutzen bzw. die Pakete, die vom aktuell ausgewählten Paket genutzt werden. In der Mitte ist eine graphische Übersicht der Abhängigkeiten des ausgewählten Pakets und seinen Klassen bzw. Interfaces zu sehen. In den gelben Rechtecken, welche die Klassen bzw. Interfaces darstellen, haben die Angaben EP, EC, AP und AC folgende Bedeutung:

EP: Anzahl der efferenten Pakete
EC: Anzahl der efferenten Klassen
AP: Anzahl der afferenten Pakete
AC: Anzahl der afferenten Klassen

Oben rechts ist unter *Statistic* eine Statistik zu dem aktuell ausgewählten Paket zu sehen.

4 Analyseergebnisse

In diesem Kapitel werden die Analyseergebnisse der in Kap. 3 vorgestellten Werkzeuge zusammengefasst und verglichen. Es erfolgt zunächst eine tabellarische Gegenüberstellung der Werkzeuge. Anschließend folgt eine Betrachtung der Werkzeuge anhand der in Kap. 3.1 definierten Analysekriterien. Zum Schluss wird dann eine sinnvolle Kombination der Werkzeuge vorgeschlagen.

Tabellarische Gegenüberstellung

Werkzeug	Was wird analysiert?	Anzahl Regeln	Eclipse-Plug-In	Ant
Checkstyle	Quellcode	131	Ja	Ja
FindBugs	Bytecode	369	Ja	Ja
PMD	Quellcode	244	Ja	Ja
Lint4j	Quellcode und Bytecode	89	Ja	Ja

Tab. 1: Gegenüberstellung Werkzeuge 1

Ein deutlicher Unterschied besteht in der Anzahl der vorhandenen Regeln. Checkstyle und PMD analysieren ausschließlich den Java-Quellcode, FindBugs ausschließlich Java-Bytecode und Lint4j sowohl Java-Quellcode als auch Bytecode. Zu allen vier Werkzeugen existiert ein Eclipse-Plug-In und sie lassen sich in Verbindung mit dem Build-Management-Werkzeug Ant verwenden.

Werkzeug	Maven	Benutzung über Kommando-zeile	Eigene GUI	Können eigene Regeln erstellt werden?	Auswahlmöglichkeit vorhandener Regeln
Checkstyle	Ja	Ja	Ja, zum Anzeigen des AST	Ja, mit Java	Über eine XML-Konfigurationsdatei und in Eclipse durch Auswahl einer XML-Konfigurationsdatei oder einem Auswahlmenü
FindBugs	Ja	Ja	Ja	Ja, mit Java	Über eine XML-Filterdatei oder Annotationen und in Eclipse über ein Auswahlmenü
PMD	Ja	Ja	Nein	Ja, mit Java oder XPath	Über eine XML-Konfigurationsdatei, Annotationen oder Kommentare und in Eclipse durch ein Auswahlmenü
Lint4j	Ja	Ja	Nein	?	In Eclipse über ein Auswahlmenü

Tab. 2: Gegenüberstellung Werkzeuge 2

Die Werkzeuge lassen sich außerdem mit dem Build-Management-Werkzeug Maven und über die Kommandozeile verwenden. Eine eigene grafische Oberfläche ist nur für FindBugs vorhanden. Die vorhandenen Regeln in Checkstyle, FindBugs und PMD können durch eigene Regeln ergänzt werden. Diese werden in Form von Java-Programmen erstellt. Im Fall von PMD ist dies auch durch XPath-Ausdrücke möglich. Ob es für Lint4j eine Möglichkeit gibt eigene Regeln zu erstellen, konnte anhand der vorhanden Dokumentation nicht festgestellt werden. Welche von den vorhandenen Regeln jeweils für eine Analyse genutzt werden sollen, können bei allen vier Werkzeugen in Eclipse über ein Auswahlmenü festgelegt werden. Für Checkstyle, FindBugs und PMD können diese auch über XML-Konfigurationsdateien festgelegt werden. FindBugs und PMD bieten zusätzlich die Möglichkeit, eine Auswahl über Annotationen zu treffen. Im durch PMD zu analysierenden Quellcode können außerdem Kommentare hinzugefügt werden, die die kommentierte Stelle von einer Prüfung ausnimmt.

Das Werkzeug CAP wird in den beiden Tabellen nicht aufgeführt, da es sich aufgrund seiner Eigenart nicht direkt mit den anderen Werkzeugen vergleichen lässt.

Installation der Werkzeuge

Die Installation erfolgt bei Checkstyle, FindBugs, PMD und Lint4j auf die gleiche Weise. Es wird eine Distribution des jeweiligen Werkzeugs z.B. in Form einer zip-Datei von der entsprechenden Projektseite heruntergeladen. Anschließend wird diese in ein beliebiges Verzeichnis entpackt. Um das Werkzeug dann am einfachsten zu verwenden, sollte man einen entsprechenden Eintrag in der PATH-Variablen setzten.

Die Installation des jeweiligen Eclipse-Plug-Ins geschieht bei allen vier Werkzeugen über den üblichen Weg der Eclipse-Plug-In – Installation.

Es ergibt sich somit insgesamt, dass die Installation der Werkzeuge auf einfache Weise möglich und der Aufwand gering ist. Keines der Werkzeuge hat bei der Installation einen Vor- oder Nachteil.

Einarbeitungsaufwand und Benutzung der Werkzeuge

Die Einarbeitung ist aufgrund der ausführlichen Beschreibungen der Werkzeuge auf den jeweiligen Projektseiten im Internet gut möglich. Ein Nachteil ist, dass diese Einarbeitung, mit Ausnahme von Lint4j, nur direkt am Rechner möglich ist, da leider keine Manuals in Pdf-Form vorhanden sind.

Der Einarbeitungsaufwand ist bei den einzelnen Werkzeugen recht unterschiedlich. Dies ergibt sich unter anderem aus der Tatsache, dass die QS-Programme sich in der Anzahl der Regeln stark unterscheiden (siehe Tab. 4.1), wenn man diese alle verstehen möchte. Da sich Checkstyle, FindBugs und PMD z.B. über XML-Dateien, bzgl. der zu verwendenden Regeln konfigurieren lassen und diese Möglichkeit genutzt werden soll, erhöht sich der Einarbeitungsaufwand. Dieses bietet aber auch Vorteile gegenüber Lint4j. Sollen die Werkzeuge ausschließlich in Verbindung mit Eclipse genutzt werden, ist der Einarbeitungsaufwand gering, da die Bedienung zum einen sehr intuitiv und zum anderen auf den jeweiligen Projektseiten auch gut beschrieben ist. Lediglich die Benutzung von CAP benötigt einen etwas höheren Einarbeitungsaufwand, da die CAP-Perspektive viele Diagramme und sonstige Auswertungen bietet. Eine Nutzung der Werkzeuge über die Kommandozeile erhöht den Einarbeitungsaufwand ebenfalls, da zu den jeweiligen QS-Programmen mehrere Optionen zur Verfügung stehen. Eine eigene graphische Oberfläche wird nur von FindBugs angeboten. Die Verwendung dieser Oberfläche ist sehr intuitiv und außerdem auf der Manual-Seite von FindBugs ausführlich beschrieben.

Der Einarbeitungsaufwand ist somit für die verschiedenen Werkzeuge unterschiedlich hoch und hängt auch damit zusammen, wie die Werkzeuge genutzt werden sollen. Aufgrund dessen ergeben sich auch hier keine Vor- oder Nachteile für die einzelnen QS-Programme. Gleiches gilt für die Benutzung der Werkzeuge.

Verwendung der Werkzeuge mit Ant und/oder Maven

Eine Verwendung mit Ant und Maven ist bei allen Werkzeugen, mit Ausnahme von CAP, möglich.

Die Integration von Checkstyle, FindBugs, PMD und Lint4j in den Build-Prozess von Ant erfolgt über *taskdef*-Deklarationen. Unterschiedliche Parameter sind für die jeweiligen QS-Programme vorhanden, um diese zu konfigurieren. Eine ausführliche Beschreibung zur Integration und den einzelnen Parametern ist auf den Projektseiten gegeben.

Maven-Plug-Ins sind für die vier im vorherigen Absatz genannten Werkzeuge vorhanden. Die QS-Programme können über die Plug-Ins in den Maven-Build-Prozess eingebunden und über verschieden Parameter konfiguriert werden. Beschreibungen zu den Parametern und wie ein solches Plug-In in eine POM-Datei eingebunden wird stehen auf den Projektseiten zu den Werkzeugen bzw. den Maven-Plug-Ins zur Verfügung.

Insgesamt ergibt sich somit ein kleiner Nachteil für CAP, da sich dieses Werkzeug nicht in einen Build-Prozess einbinden lässt.

Sinnvolle Kombination der Werkzeuge

Bevor ein Vorschlag zur sinnvollen Kombination der Werkzeuge vorgestellt wird, soll zunächst eine tabellarische Übersicht gegeben werden, zu wie viel Prozent ein Werkzeug die Regeln eines anderen Werkzeugs abdeckt. Anschließend erfolgt eine Übersicht der Alleinstellungsmerkmale der QS-Programme.

Es wurde in Microsoft Excel jeweils eine Tabelle mit den für Checkstyle, FindBugs, PMD und Lint4j vorhandenen Regeln erstellt. Anhand einer Auswertung dieser Tabellen konnten die jeweiligen Prozentwerte in Tab. 4.3 ermittelt werden.

	Checkstyle	**FindBugs**	**PMD**	**Lint4j**
Checkstyle	-	2 %	18 %	10 %
FindBugs	6 %	-	10 %	19 %
PMD	27 %	9 %	-	21 %
Lint4j	5 %	6 %	11 %	-

Tab. 3: Übersicht prozentuale Abdeckung von Regeln anderer Werkzeuge

Die Tabelle zeigt, dass Checkstyle z.B. 2 % der Regeln von FindBugs, 18 % der Regeln von PMD und 10 % der Regeln von Lint4j abdeckt usw. Entscheidet man sich dazu nur eins der vier Werkzeuge einzusetzen, zeigt die Tabelle, dass PMD bevorzugt werden sollte, da es insgesamt die größte Anzahl von Regeln anderer QS-Programme beinhaltet. Außerdem können fehlende Regeln durch eigene Programmierung ergänzt werden.
Die folgende Tabelle zeigt, wie viele der Regeln eines Werkzeugs exklusiv von diesem zur Verfügung gestellt werden, d.h. von keinem anderen QS-Programm auf eine ähnliche Weise angeboten werden.

Checkstyle	96
FindBugs	323
PMD	170
Lint4j	58

Tab. 4: Anzahl exklusiver Regeln der Werkzeuge

Im Folgenden werden die Alleinstellungsmerkmale der Werkzeuge genannt, wobei dieses sich immer auf eine zusammengehörige Gruppe von Regeln beziehen.

Checkstyle: Regeln bezüglich

- Annotationen
- Javadoc – Kommentare
- Namenskonventionen (falls diese selbst definiert werden sollen)
- Nutzung von Leerzeichen

FindBugs: Regeln bezüglich

- Bit-Operationen
- Format-Strings
- Null-Pointer und null-Werte

PMD: Regeln bezüglich

- Android
- Java Server Pages
- J2EE
- Logging

Lint4j: Regeln bezüglich

- Enterprise Java Beans

Ein Vorschlag zur sinnvollen Kombination der Werkzeuge auf Basis einzelner Regeln ist aufgrund der Vielzahl von Regeln nicht möglich. Deshalb basiert der Vorschlag auf den individuellen Stärken der einzelnen Werkzeuge.

Um die Einhaltung von Programmierrichtlinien zu überprüfen, wird das Werkzeug Checkstyle empfohlen, da in diesem Bereich schon sehr viele Regeln vorhanden sind. Durch Parameter können die Regeln auf definierte Programmierrichtlinien eingestellt werden. Sollte dennoch die Einhaltung einer Regel aus den definierten Programmierrichtlinien nicht direkt durch Checkstyle entdeckt werden, kann Checkstyle durch eigene sogenannte Checks ergänzt werden.

PMD eignet sich sehr gut um Codestellen zu finden, die man mit „bad practices“ bezeichnen kann. Dieses sind z.B. leere Blöcke und das Fangen von Ausnahmen, ohne dass anschließend etwas geschieht usw..

FindBugs und Lint4j sind besonders geeignet, wenn es um mögliche Bugs im Bezug auf Multithreading geht. Falls die Werkzeuge nicht nur als Plug-In einer Entwicklungsumgebung genutzt werden sollen, sondern z.B. auch im Zusammenhang mit Ant oder Maven, sollte FindBugs bevorzugt werden, da eine explizite Regelauswahl für Lint4j leider nicht gegeben ist. Außerdem bietet FindBugs den Vorteil, dass es um eigene Regeln ergänzt werden kann.

Insgesamt wird empfohlen, die Werkzeuge Checkstyle, FindBugs und PMD als Plug-In einer Entwicklungsumgebung zu nutzen, da mögliche Regelverletzungen direkt angezeigt und behoben werden können. Wird Ant oder Maven als Build-Management-Werkzeug eingesetzt, sollte man die QS-Programme auch in den Build-Prozess einbeziehen, da so eine abschließende Kontrollmöglichkeit besteht, ob auch alle Entwickler einwandfreien Code, bezüglich der spezifizierten Regeln geschrieben haben, oder ob vielleicht doch noch Fehler übersehen wurden.

Um die Werkzeuge sinnvoll nutzen zu können, bedarf es einer individuellen Auswahl von Regeln, die beachtet werden sollen. Dies ist selbstverständlich mit einem hohen Aufwand verbunden, allerdings ist dieser nur einmalig und sollte sich im Laufe der Zeit bezahlt machen.

Das Werkzeug CAP bietet sich an, falls Kennzahlen bezüglich der Softwarearchitektur ermittelt und anschließend bewertet werden sollen. Allerdings setzt die Verwendung dieses Werkzeugs die Nutzung von Eclipse vor aus.

5 Zusammenfassung

Zur Qualitätssicherung in der Softwareentwicklung können u.a. die statischen Testverfahren genutzt werden. Hierzu zählt die statische Quellcodeanalyse. Da eine solche Analyse sehr aufwändig sein kann, sollte sie durch entsprechende Werkzeuge durchgeführt werden. Als Beispiele wurden in dieser Arbeit fünf verschiedene Werkzeuge analysiert. Die Analyse erfolgte anhand der Kriterien: Typische Einsatzszenarien, Einarbeitungsaufwand, Installation, Benutzbarkeit und Verwendung mit Ant und/oder Maven.

Der Einarbeitungsaufwand ist je nach Umfang der Werkzeuge unterschiedlich. Eine Einarbeitung ist aber dank der vorhanden, guten Projektseiten im Internet gut möglich. Auch die Installation und Benutzung ist aufgrund der guten Dokumentation nicht kompliziert. Eine Verwendung im Zusammenhang mit den Build-Management-Werkzeugen Ant und Maven ist, bis auf bei CAP, bei allen analysierten Werkzeugen möglich.

Ein konkreter Vorschlag zur kombinierten Nutzung der Werkzeuge ist aufgrund der Vielzahl von vorhandenen Regeln sehr schwer. Außerdem kommt es auch auf die individuellen Bedürfnisse der Nutzer an. Eine einmalige Auswahl von Regeln und der entsprechenden Werkzeuge anhand der eigenen Bedürfnisse kann recht aufwendig sein, sollte aber anschließend gewinnbringend sein.

Im Rahmen der Erstellung dieser Arbeit wurde das Werkzeug *QAlab* „entdeckt“. Dieses Werkzeug fast die Werkzeuge: Checkstyle, PMD, FindBugs, Cobertura und Simian in einem zusammen. Daher könnte eine Betrachtung dieses Werkzeugs in einer zukünftigen Bachelorarbeit sehr sinnvoll sein.
Die analysierten fünf Werkzeuge sind nur ein kleiner Teil der zur Verfügung stehenden Werkzeuge. Somit könnten in weiteren Arbeiten auch noch andere QS-Werkzeuge untersucht werden.

Abkürzungsverzeichnis

AST	Abstract Syntax Tree
CSS	Cascading Style Sheet
GUI	Graphical User Interface
IDE	Integrated Development Environment
POM	Project Object Model
UML	Unified Modeling Language
XML	Extensible Markup Language

Abbildungsverzeichnis

Tabellenverzeichnis

Anhang A

Beispiel zu Kapitel 3.2:

Damit alle Warnungen der Compileroption -Xlint ausgegeben werden, sollte der folgende Code wie folgt übersetzt werden, wobei der Ordner *test* nicht existiert:

javac –Xlint –sourcepath test OverrideTestClass.java JavaCompilerTest.java

JavaCompilerTest.java:

```
package javacompilertest;

import java.io.Serializable;
import java.util.Date;
import java.util.HashSet;
import java.util.Set;

public class JavaCompilerTest extends OverrideTestClass implements
Serializable{

	//keine serialVersionUID,
	//um die Compileroption -Xlint:serial zu testen

	private int testVariable;

	public JavaCompilerTest(int testVariable) {
		this.testVariable = testVariable;
	}

	public int getTestVariable() {
		return testVariable;
	}

	public void setTestVariable(int testVariable) {
		this.testVariable = testVariable;
	}

	//Testfunktion zur Compileroption -Xlint:depricationCheck
	public void depricationCheck(){
		Date abgabedatum = new Date(2011, 5, 25);
	}

	//Testfunktion zur Compileroption -Xlint:empty
	public void emptyTest(){
		if(this.getTestVariable() == 10);
	}
	//Testfunktion zur Compileroption -Xlint:fallthrough
	public void fallthroughTest(){
		switch (this.getTestVariable()) {
		case 1: System.out.println("Case 1");
				break;
		case 2: System.out.println("Case 2");
		case 10: System.out.println("Case 10");
				 break;
		default:
			break;
		}
	}

	//Testfunktion zur Compileroption -Xlint:finally
```

```
	public void finallyTest(){
		try {
			int divison = this.divison(10,0);
		} catch (Exception e) {
			// TODO: handle exception
		}
	}

	private int divison(final int dividend, final int divisor) {
		int quotient = 0;
		try {
			if(divisor == 0){
				throw new ArithmeticException("Division durch 0"
					+ " ist nicht erlaubt!");
			}
			quotient = dividend/divisor;
		}
		finally{
			return quotient;
		}
	}

	//Testfunktion zur Compileroption -Xlint:override
	@Override
	public void overrideTestfunktion(final String... test){

	}

	//Testfunktion zur Compileroption -Xlint:unchecked
	public Set uncheckedTest(){
		final Set testSet = new HashSet();
		testSet.add(10);
		testSet.add(11);
		testSet.add(12);
		return testSet;
	}

	public static void main(String[] args) {
		JavaCompilerTest jct = new JavaCompilerTest(10);
		//Testen der Compileroption -Xlint:cast
		System.out.println("Testvariable = " +
					(int)jct.getTestVariable());

		//Test der Compileroption -Xlint:divzero
		int div = jct.getTestVariable()/0;

		jct.emptyTest();
		jct.fallthroughTest();
		Set test = jct.uncheckedTest();
	}
}
```

OverrideTestClass.java:

```
package javacompilertest;

public class OverrideTestClass {
	public void overrideTestfunktion(final String[] test){

	}
}
```

Ausgabe des Compilers:

```
JavaCompilerTest.java:43: warning: [deprecation] Date(int, int, int) in java.util.Date has been
deprecated
                Date abgabedatum = new Date(2011, 5, 25);
                                   ^
JavaCompilerTest.java:48: warning: [empty] empty statement after if
                if(this.getTestVariable() == 10);
                                          ^
JavaCompilerTest.java:89: warning: overrideTestfunktion(java.lang.String...) in
javacompilertest.JavaCompilerTest overrides overrideTestfunktion(java.lang.String[]) in
javacompilertest.OverrideTestClass; overridden method has no '...'
        public void overrideTestfunktion(final String... test){
                    ^
JavaCompilerTest.java:96: warning: [unchecked] unchecked call to add(E) as a member of the raw
type java.util.Set
                testSet.add(10);
                        ^
JavaCompilerTest.java:109: warning: [divzero] division by zero
                int div = jct.getTestVariable()/0;
                                               ^
JavaCompilerTest.java:22: warning: [serial] serializable class javacompilertest.JavaCompilerTest has
no definition of serialVersionUID

public class JavaCompilerTest extends OverrideTestClass implements Serializable{
    ^
JavaCompilerTest.java:57: warning: [fallthrough] possible fall-through into case
                case 10: System.out.println("Case 10");
                ^
JavaCompilerTest.java:84: warning: [finally] finally clause cannot complete normally
                }
                ^
JavaCompilerTest.java:106: warning: [cast] redundant cast to int
                                                        (int)jct.getTestVariable());
```

Beispieldateien zu Kapitel 3.3.3 und 3.3.4

KonfigBeispiel.java:

```
public class KonfigBeispiel {

	/**
	 * Dies ist der Javadoc-Kommentar für javaDocVariable
	 **/
	private int javaDocVariable;
	private int testVariable1;
	private int testVariable2;
	private int testVariable3;


	public KonfigBeispiel(int javaDocVariable, int testVariable1,
			int testVariable2, int testVariable3) {
		this.javaDocVariable = javaDocVariable;
		this.testVariable1 = testVariable1;
		this.testVariable2 = testVariable2;
		this.testVariable3 = testVariable3;
	}

	private int requireThisUndCyclomaticComplexityTest(){
		int sum=0;
		if(testVariable1 > javaDocVariable){
			for(int i=0;i<10;i++){
				sum += i;
			}
		}
		else if(testVariable2 > javaDocVariable){
			for(int i=0;i<20;i++){
				sum += i;
			}
		}
		else if(testVariable3 > javaDocVariable){
			for(int i=0;i<30;i++){
				sum += i;
			}
		}
		else{
			sum = this.javaDocVariable;
		}

		return sum;
	}

	public static void main(String[] args) {
		KonfigBeispiel kb = new KonfigBeispiel(40, 10, 20, 30);
		System.out.println("Summe = " +
				kb.requireThisUndCyclomaticComplexityTest());
	}
}
```

KonfigBeispiel.properties:

comp=5
mlength=4

Installation des FindBugs-Plug-Ins für Eclipse

- Wähle *Help* => *Install New Software* => Unter "Work with" auf *Add*... klicken
- *Name*: "FindBugs", *Location*: "http://findbugs.cs.umd.edu/eclipse-daily"
- Wähle *Ok* => Häkchen setzten bei "FindBugs" => *Next* => *Next*
- Der Lizenzvereinbarung zustimmen
- Wähle *Finish* und zum Schluss *Restart Now*

Beispieldatei zu Kapitel 3.4.3 und 3.4.4

FindBugs_Demo.java:

```
package findbugsdemo;

import java.io.Serializable;

public class FindBugs_Demo  implements Serializable{

	long serialVersionUID = 1L;

	private int testVariable;

	public int getTestVariable() {
		return testVariable;
	}
	public synchronized void setTestVariable(int testVariable) {
		this.testVariable = testVariable;
	}

	@Override
	public boolean equals(Object arg0) {
		return super.equals(arg0);
	}

	private void neverUsedmethod(){
		synchronized (Boolean.TRUE) {
		}
	}

	public static void main(String[] args) {
		System.out.printf("%s" + "%s", "Es fehlt was!");
	}
}
```

Installation des PMD-Plug-Ins für Eclipse

- Wähle *Help* => *Install New Software* => Unter "Work with" auf *Add*... klicken
- *Name*: "PMD", *Location*: "http://pmd.sf.net/eclipse"
- Wähle *Ok* => Häkchen setzten bei "PMD" => *Next* => *Next*
- Der Lizenzvereinbarung zustimmen
- Wähle *Finish* und zum Schluss *Restart Now*

Beispieldatei zu Kapitel 3.5.3 und 3.5.4

PMDDemo.java:

```
package pmddemo;

public class PMDDemo {

	//Basic Rules
	private void emptyIfStmt(final int x){
		if(x==0){
		}
		else{
			System.out.println("x = " + x);
		}
	}

	private void emptySynchronizedBlock(){
		synchronized (this){
		}
	}

	//Design Rules
	private void avoidDeeplyNestedIfStmts(int x, int y, int z){
		if(x>y){
			if(z>y){
				if(z==x){
					System.out.println("z==x und x>y");
				}
			}
		}
	}

	//Code Size Rules
	private void excessiveParameterList(int param1, int param2, int
					param3, int param4, int param5,
					int param6, int param7, int
					param8, int param9, int param10){

	}

	//Naming Rules

	private void MethodNamingConventions(){
		System.out.println("Methodennamen sollten mit einem kleinen
				Buchstaben beginnen!");
	}

	public static void main(String[] args) {
		PMDDemo pmd = new PMDDemo();
		pmd.emptyIfStmt(5);
		pmd.emptySynchronizedBlock();
		pmd.avoidDeeplyNestedIfStmts(5, 4, 5);
		pmd.excessiveParameterList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
		pmd.MethodNamingConventions();
	}}
```

Installation des Lint4j-Plug-Ins für Eclipse

- Wähle *Help* => *Install New Software* => Unter "Work with" auf *Add*... klicken
- *Name*: "Lint4j", *Location*: "http://www.jutils.com/eclipse-update"
- Wähle *Ok* => Häkchen setzten bei "Lint4j" => *Next* => *Next*
- Der Lizenzvereinbarung zustimmen
- Wähle *Finish* und zum Schluss *Restart Now*

Demo-Programm für Lint4j

Lint4j_Demo.java:

```
package lint4jdemo;

public class Lint4j_Demo {
	private String demo;

	public Lint4j_Demo(String demo) {
		super();
		this.demo = demo;
	}

	public String getDemo() {
		return demo;
	}

	public void setDemo(String demo) {
		this.demo = demo;
		return;
	}

	public static void main(String[] args) {
		try{
			Lint4j_Demo demo = new Lint4j_Demo("demo");
			String integer = new Integer(2).toString();
			String castString = (String) integer;
			System.out.println(demo.getDemo().toString() +
							integer + castString);
			System.exit(0);
		}finally{
			return;
		}
	}
}
```

Installation des CAP-Plug-Ins für Eclipse

- Wähle *Help* => *Install New Software* => Unter “Work with” auf *Add…* klicken
- *Name*: “CAP”, *Location*: “http://cap.xore.de/update"
- Wähle *Ok* => Häkchen setzten bei “CAP“ => *Next* => *Next*
- Der Lizenzvereinbarung zustimmen
- Wähle *Finish* und zum Schluss *Restart Now*

Sind die beiden von CAP benötigten Plug-Ins noch nicht installiert, können diese auch durch zusätzliche Auswahl in dem Dialog, wo das Häkchen bei CAP gesetzt wird, direkt mit installiert werden.

Anhang B

Inhalt der CD:

Ordner	Dateien
Bachelorarbeit_Jens_Attermeyer/ analysierte_Werkzeuge/ Checkstyle/Beispiele/Ant	build.xml KonfigBeispiel.properties KonfigBeispiel.xml
Bachelorarbeit_Jens_Attermeyer/ analysierte_Werkzeuge/ Checkstyle/Beispiele/Ant/src	KonfigBeispiel.java
Bachelorarbeit_Jens_Attermeyer/ analysierte_Werkzeuge/ Checkstyle/Beispiele/Kommandozeile	KonfigBeispiel.bat KonfigBeispiel.xml KonfigBeispiel.properties
Bachelorarbeit_Jens_Attermeyer/ analysierte_Werkzeuge/ Checkstyle/Beispiele/Kommandozeile/src	KonfigBeispiel.java
Bachelorarbeit_Jens_Attermeyer/ analysierte_Werkzeuge/ Checkstyle/Programm	Enthält die Checkstyle-Distribution Version 5.3
Bachelorarbeit_Jens_Attermeyer/ analysierte_Werkzeuge/ Compiler	CompilerTest.bat JavaCompilerTest.java OverrideTestClass.java
Bachelorarbeit_Jens_Attermeyer/ analysierte_Werkzeuge/FindBugs/Beispiele/Ant	build.xml
Bachelorarbeit_Jens_Attermeyer/ analysierte_Werkzeuge/FindBugs/ Beispiele/Ant/src/findbugsdemo	FindBugs_Demo.java
Bachelorarbeit_Jens_Attermeyer/ analysierte_Werkzeuge/FindBugs/ Beispiele/Ant/bin/findbugsdemo	FindBugs_Demo.class
Bachelorarbeit_Jens_Attermeyer/ analysierte_Werkzeuge/FindBugs/ Beispiele/Kommandozeile	FindBugs_Demo.bat
Bachelorarbeit_Jens_Attermeyer/ analysierte_Werkzeuge/FindBugs/ Beispiele/Kommandozeile/src/findbugsdemo	FindBugs_Demo.java
Bachelorarbeit_Jens_Attermeyer/ analysierte_Werkzeuge/FindBugs/ Beispiele/Kommandozeile/bin/findbugsdemo	FindBugs_Demo.class
Bachelorarbeit_Jens_Attermeyer/ analysierte_Werkzeuge/FindBugs/ Programm/	Enthält die FindBugs-Distribution Version 1.3.9
Bachelorarbeit_Jens_Attermeyer/ analysierte_Werkzeuge/Lint4j/ Beispiele/Ant	build.xml
Bachelorarbeit_Jens_Attermeyer/ analysierte_Werkzeuge/Lint4j/ Beispiele/Ant/src/lint4jdemo	Lint4j_Demo.class Lint4j_Demo.java
Bachelorarbeit_Jens_Attermeyer/ analysierte_Werkzeuge/Lint4j/ Beispiele/Kommandozeile	lint4j_demo.bat

Ordner	Dateien
Bachelorarbeit_Jens_Attermeyer/ analysierte_Werkzeuge/Lint4j/ Beispiele/Kommandozeile/src/lint4jdemo	lint4j_demo.class lint4j_demo.java
Bachelorarbeit_Jens_Attermeyer/ analysierte_Werkzeuge/Lint4j/ Programm	Enthält die Lint4j-Distribution Version 0.9.1
Bachelorarbeit_Jens_Attermeyer/ analysierte_Werkzeuge/PMD/Beispiele/Ant	build.xml PMDDemo.java RuleSet_demo.xml
Bachelorarbeit_Jens_Attermeyer/ analysierte_Werkzeuge/PMD/Beispiele/ Kommandozeile	PMDDemo.bat PMDDemo.java RuleSet_demo.xml
Bachelorarbeit_Jens_Attermeyer/ analysierte_Werkzeuge/PMD/Programm	Enthält die PMD-Distribution Version 4.2.5
Bachelorarbeit_Jens_Attermeyer/ schriftliche_Ausarbeitung	BachelorarbeitJensAttermeyer.docx BachelorarbeitJensAttermeyer.pdf

Tab. 5: Inhalt der CD

Pfadangaben in den Dateien müssen angepasst werden!!

Erklärung

Hiermit versichere ich, dass ich meine Bachelorarbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Datum: .....

(Unterschrift)

Quellen und Literaturverzeichnis

[AHO08] Aho, A. et al., *Compiler – Prinzipien, Techniken und Werkzeuge*, 2. Auflage, München, 2008, Pearson Studium

[ANTLR] ANTLR Projektseite: http://www.antlr.org/ (Zugegriffen am 05.04.2011, archiviert mit WebCite® als: http://www.webcitation.org/5xiTNSCU6)

[CAP] CAP Projektseite: http://cap.xore.de/cap.php?show=cap.index (zugegriffen am 04.05.2011, archiviert mit WebCite® als: http://www.webcitation.org/5yQXGcWpP)

[CS] Checkstyle Projektseite: http://checkstyle.sourceforge.net/index.html (Zugegriffen am 04.04.2011, archiviert mit WebCite® als: http://www.webcitation.org/5x0FbvTL0)

[CSE] Checkstyle- Eclipse-Plug-In Projektseite: http://eclipse-cs.sourceforge.net/ (Zugegriffen am 06.04.2011, archiviert mit WebCite® als: http://www.webcitation.org/5xk5CPcfz)

[CSM] Maven-Checkstyle-Plug-In Projektseite: http://maven.apache.org/plugins/maven-checkstyle-plugin/index.html (Zugegriffen am 08.04.2011, archiviert mit WebCite® als: http://www.webcitation.org/5xoROHnH0)

[FB] FindBugs Projekseite: http://findbugs.sourceforge.net/ (zugegriffen am 11.04.2011, archiviert mit WebCite® als: http://www.webcitation.org/5xrnBZOOL)

[FBM] Maven-FindBugs-Plug-In: http://mojo.codehaus.org/findbugs-maven-plugin/ (zugegriffen am 15.04.2011, archiviert mit WebCite® als: http://www.webcitation.org/5xxVZTip3)

[FLE07] Fleischer, A., *Metriken im praktischen Einsatz*, 2007 http://www.sigs-datacom.de/fileadmin/user_upload/zeitschriften/os /2007/03/fleischer_OS_03_07.pdf (Zugegriffen am 30.03.2011, archiviert mit WebCite® als: http://www.webcitation.org/5xZHcP0WD)

[ISO 9126] ISO-Norm 9126: http://de.wikipedia.org/wiki/ISO/IEC_9126 (Zugegriffen am 18.05.2011, archiviert mit WebCite® als: http://www.webcitation.org/5ylyuBlPU)

[JW10] JavaWorld, *javac's –Xlint Options*, 2010 http://www.javaworld.com/community/node/5276 (Zugegriffen am 30.03.2011, archiviert mit WebCite® als: http://www.webcitation.org/5xQHT6J6T)

[LIG09] Liggesmeyer, P., *Software-Qualität – Testen, Analysieren und Verifizieren von Software* 2. Auflage, Heidelberg, 2009, Spektrum Akademischer Verlag

[PMD] PMD Projektseite: http://pmd.sourceforge.net/
(zugegriffen am 18.05.2011, archiviert mit WebCite® als:
http://www.webcitation.org/5ylxEOIyK)

[PMDM] Maven-PMD-Plug-In: http://maven.apache.org/maven-1.x/plugins/pmd/
(zugegriffen am 18.05.2011, archiviert mit WebCite® als:
http://www.webcitation.org/5ylxOkkfn)

[SpLi10] Spillner, A., Linz T., *Basiswissen Softwaretest*, 4. Auflage, Heidelberg, 2010, dpunkt