ist Institut für Softwaretechnik

UNIVERSITÄT KOBLENZ · LANDAU
Fachbereich 4: Informatik

Diplomarbeit

Möglichkeiten der Interoperabilität zwischen bestehenden Metrik-Tools für COBOL und den Entwicklerwerkzeugen der Debeka

vorgelegt von:

Arne Baldauf

abaldauf@uni-koblenz.de

Koblenz, im Dezember 2010

Betreuer:

Prof. Dr. Jürgen Ebert (ebert@uni-koblenz.de)

Daniel Bildhauer (dbildh@uni-koblenz.de)

Judith Haas (juhaas@uni-koblenz.de)

Erklärung

Ich versichere, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Mit der Einstellung dieser Arbeit in die Bibliothek bin ich einverstanden. Der Veröffentlichung dieser Arbeit im Internet stimme ich zu.

Ort, Datum	Unterschrift

Inhaltsverzeichnis

1 Abstract

Im Rahmen dieser Diplomarbeit wird erforscht, wie bestehende Tools zur Berechnung von Softwaremetriken über COBOL-Quelltexte mit den vorhandenen Entwicklerwerkzeugen der Debeka Versicherungsgruppe interoperabel gemacht werden können.

Der Text richtet sich primär an Leser aus dem Bereich der Informatik. Zum Verständnis wird ein Wissensstand vergleichbar dem eines Informatikers mit abgeschlossenem Bachelorstudium oder Vordiplom vorausgesetzt.

Die Gliederung dieser Arbeit orientiert sich an den Phasen des Softwarelebenslaufs. Zunächst findet eine Einführung in die für die Arbeit relevanten Themengebiete COBOL, Softwaremetriken und das Forschungsprojekt Cobus statt, in dessen Rahmen auch diese Diplomarbeit fällt. Außerdem werden die Ziele der Arbeit im Detail erläutert.

Der darauf folgenden Teil der Arbeit beschreibt die Durchführung und Ergebnisse der Recherche nach existierenden Tools zur Berechnung von Softwaremetriken für COBOL. Dieser Teil endet mit einer Bewertung und der Auswahl von SofAudit, COBOL Flow Graph Manipulator (FGM) und Continuous Quality Assessment Toolkit (ConQAT) als geeignete Tools.

Die drei Programme werden im nächsten Teil der Arbeit genauer analysiert und getestet. Dabei wird auch untersucht, worin teils abweichende Ergebnisse zwischen den Tools begründet sind. Es stellt sich heraus, dass FGM in der vorliegenden Version nicht zum Realisieren der geforderten Interoperabilität geeignet ist.

Die Speicherung und Weiterverarbeitung von berechneten Metrikwerten in den Entwicklerwerkzeugen der Debeka macht ein einheitliches Datenformat erforderlich, welches im fünften Teil der Arbeit als XML-basiertes Format entworfen und spezifiziert wird.

Im darauf folgenden Teil wird die eigentliche Software entworfen, welche in einem Hauptprogramm mit je einer universellen Plug-In-Schnittstelle für Metrik-Tools beziehungsweise den Export der Metrikdaten resultiert. Es werden auch die Plug-Ins für SofAudit und ConQAT sowie für den Export der Metrikdaten in das eigene Format und in CSV-Dateien entworfen und realisiert.

Der vorletzte Teil der Arbeit beschreibt die Verwendung und Konfiguration der entwickelten Software.

Im letzten Teil erfolgt die Bewertung der Arbeit anhand der Zielsetzung und den Anforderungen. Dabei wird klar, dass zwar die geforderte Interoperabilität erreicht wurde, die existierenden Metrik-Tools für die Ziele der Forschungsprojekts Cobus aufgrund der vorhandenen Funktionalität und vor allem der mangelnden Erweiterbarkeit eher ungeeignet sind.

Die eigentliche Arbeit wird gefolgt von den Anhängen, in denen sich zusätzliche Informationen befinden. Der Anhang beinhaltet auch die CD mit den entwickelten und verwendeten Programmen sowie den digitalen Versionen aller erstellten Dokumente und Protokolle. Den Abschluss bildet das Literaturverzeichnis.

2 Kontext und Ziel der Arbeit

In diesem Kapitel wird zunächst eine Übersicht über die für das allgemeine Verständnis relevanten Themen geschaffen. Darüber hinaus findet eine kurze Einführung in das Forschungsprojekt Cobus statt, in dessen Rahmen diese Diplomarbeit durchgeführt wird. Als letzter Teil des Kapitels folgt die eigentliche Zielsetzung der Arbeit.

2.1 Cobus

Seit November 2009 betreibt das Institut für Softwaretechnik[1] der Universität Koblenz gemeinsam mit der Debeka Versicherungsgruppe[2] das Forschungsprojekt *COBOL-Bestandsanalyse und -Sanierung (Cobus)*[3]. Der Abschluss des Projektes ist bis Oktober 2011 geplant.

Einige der derzeit von der Debeka betriebenen Softwaresysteme sind von der Versicherungsgruppe selbst in COBOL entwickelt worden. Ein Großteil dieser Systeme betreffen die Kerngeschäftstätigkeit und sind teilweise seit mehr als 20 Jahren im Einsatz. Die zugehörige Codebasis dieser Systeme ist umfangreich. Es fand auch und findet immer noch eine stetige Weiterentwicklung und Wartung der Software statt. Die Relevanz dieser Software für die Debeka und der betriebene Entwicklungsaufwand spiegeln sich auch in der Tatsache wider, dass ein eigenes Plug-In zur COBOL-Programmierung für die Entwicklungsumgebung Eclipse[4] entwickelt wurde. Daneben existieren auch die eigens entwickelte Datenabfragesprache DQL samt zugehörigem COBOL-Präprozessor und einige weitere Tools.

Ziel des Forschungsprojektes Cobus ist es (siehe [Ebert2009]), „eine umfassende Bestandsanalyse und -bewertung des COBOL-basierten Softwaresystems der Debeka durchzuführen. Auf Grundlage dieser Analyse und Bewertung werden anschließend geeignete Maßnahmen abgeleitet, mit denen die Qualität des Systems optimiert und die langfristige Weiterentwickelbarkeit sichergestellt werden kann. Den letzten Teil des Projekts bildet die praktische Anwendung auf einen ausgewählten Teilbereich des Softwaresystems. Dadurch soll die Wirksamkeit der Maßnahmen bewiesen werden“.

2.2 COBOL

Die erste Version der Programmiersprache *Common Business Oriented Language (COBOL)* wurde bereits 1960 von einer Arbeitsgruppe der *Conference on Data System Languages (CODASYL)* entwickelt. Es handelte sich dabei primär um eine offene Sprachspezifikation. COBOL wurde seitdem zu keiner Zeit von einer einzelnen Firma entwickelt und verwaltet, wie dies etwa bei

[1] http://www.uni-koblenz-landau.de/koblenz/fb4/institute/IST
[2] http://www.debeka.de/
[3] http://www.uni-koblenz-landau.de/koblenz/fb4/institute/IST/AGEbert/projekte/cobus
[4] http://www.eclipse.org/

Java[5] oder den .NET-Sprachen[6] der Fall ist. Da CODASYL nicht mehr existiert, unterliegt die Standardisierung von COBOL inzwischen der Zuständigkeit des *American National Standards Institute (ANSI)*[7].

Mit 50 Jahren hat COBOL ein für die IT-Branche sehr hohes Alter. Bei zusätzlicher Betrachtung der hohen Medienpräsenz wesentlich jüngerer Programmiersprachen ist somit die weit verbreitete Meinung, dass COBOL in der heutigen Zeit eine nur noch geringe Relevanz besäße, verständlich. Jedoch wird für das Jahr 1997 ein Anteil von 80% COBOL-Code am damals weltweit existierenden Quellcode (gemessen in Codezeilen) geschätzt und für das Jahr 1999 immerhin ein Anteil von 50% am neuentwickelten Quellcode für Software in betrieblichen Schlüsselsystemen (siehe [Brown1999]). Aber auch jüngere Artikel belegen, dass COBOL nach wie vor eine wichtige und auch aktiv genutzte Programmierprache ist (siehe [Mitchell2006] und [Zotow2009]).

Auf technische Details soll an dieser Stelle nicht weiter eingegangen werden, eine genauere Einführung in die Programmiersprache liefert zum Beispiel [Coughlan2002].

2.3 Metriken

Metriken dienen in der Softwareentwicklung dazu, Aussagen über bestimmte Eigenschaften eines Softwaresystems zu treffen. Diese Eigenschaften müssen in messbaren Größen, das heisst in numerischen Werten, vorliegen.

Metriken werden dabei differenziert nach:

- Größenmaßen (typischerweise mit absoluten Werten - zum Beispiel die Anzahl der Quelltextzeilen),
- Dichtemaßen (üblicherweise mit prozentualen Werten - zum Beispiel der Anteil der Kommentarzeilen),
- Strukturmaßen (zum Beispiel die mittlere Verschachtelungstiefe von Schleifen oder Bedingungen).

In den meisten Fällen können die Metrikwerte dabei mittels Analyse des Quelltextes automatisiert erzeugt werden. Nur in wenigen Fällen - wie etwa bei der Berechnung des Anteils fehlerhaft programmierter Anweisungen[8] - ist dies nicht möglich.

Einen detaillierten Einstieg zum Thema Metriken bietet unter anderem [Sneed2008-1].

[5] `http://java.sun.com/`, entwickelt durch Sun Microsystems (wurde 2009 durch Oracle übernommen).
[6] `http://www.microsoft.com/NET/`, entwickelt durch Microsoft.
[7] `http://www.ansi.org/`
[8] Hiermit sind logisch oder semantisch falsche Anweisungen und nicht Syntaxfehler gemeint.

2.4 Bestehende Techniken am Institut für Softwaretechnik

Im Rahmen der Forschung der Arbeitsgruppe von Prof. Dr. Ebert am Institut für Softwaretechnik sind eine Reihe von Tools und Techniken zur Entwicklung, Wartung und Analyse von Softwaresystem entstanden. Die meisten dieser Tools und Techniken arbeiten nach dem Prinzip der Extraktion, Transformation und Anfrage. Sie folgen dabei einem graphbasierten Ansatz bei der Repräsentation der zu verarbeitenden Softwaresysteme. Graphen ermöglichen einen variablen Grad der Abstraktion über die extrahierten Daten. Ein weiterer Vorteil besteht darin, dass Graphen sehr effizient traversiert werden können.

Bei den verwendeten Graphen handelt es sich um *TGraphen*[9], welche ebenfalls von der Arbeitsgruppe entwickelt wurden. TGraphen sind typisiert, attributiert, gerichtet und geordnet. Die konkrete Ausprägung wird mittels eines Graphschemas festgelegt. Eine detaillierte Einführung zu TGraphen findet sich in [Ebert1995].

2.5 Ziel der Arbeit

Im Rahmen des Projektes Cobus sollen COBOL-basierte Softwaresysteme auch mittels Softwaremetriken bewertet werden.

Das Ziel der Diplomarbeit ist es, die Eignung bestehender Metrik-Software für COBOL hinsichtlich der Interoperabilität mit den bestehenden Entwicklerwerkzeugen der Debeka zu bewerten. Dazu soll zunächst eine Übersicht über die existierenden Metrik-Tools für COBOL erstellt werden. Im nächsten Schritt sollen aus dieser Übersicht drei oder vier Systeme ausgewählt werden, die hierfür am besten geeignet sind. Diese sollen dann im Detail untersucht, verglichen und bewertet werden.

Darüber hinaus soll für jedes dieser Tools aus der engeren Auswahl eine funktionsfähige Implementation erstellt werden, welche die Interoperabilität mit den Entwicklerwerkzeugen der Debeka realisiert. Die Implementation soll dabei bereits eine Auswahl von Metriken unterstützen und darüber hinaus so konzipiert sein, dass eine Erweiterung um neue Metriken ermöglicht wird. Weitere Metriken könnten dabei, basierend auf dem Vorgehensmodell der *Goal/Question/-Metric*-Methode (siehe [Basili1992]), im Rahmen von Cobus - nicht jedoch im Rahmen dieser Arbeit - definiert werden.

Die Anforderungen an die zu entwickelnde Software hängen zum Teil auch von den Ergebnissen der Recherche nach existierenden und der detaillierten Untersuchung einer Auswahl geeigneter Metrik-Tools ab. Deswegen werden die Anforderungen nicht an dieser Stelle, sondern erst unmittelbar vor dem Entwurf in Kapitel 6 aufgeführt.

[9] `http://www.uni-koblenz-landau.de/koblenz/fb4/institute/IST/AGEbert/MainResearch/Graphentechnologie/TGraphen`

3 Recherche nach bestehenden Metrik-Tools für COBOL

In diesem Kapitel werden die im Rahmen der Recherche ermittelten Metrik-Tools für COBOL vorgestellt. Zunächst werden die dabei gestellten Anforderungen (Abschnitt 3.1) und zu recherchierenden Kriterien (Abschnitt 3.2) definiert. Darauf folgt eine Beschreibung der einzelnen Tools (Abschnitt 3.3) und eine Gesamtübersicht (Abschnitt 3.4). Im Anschluss daran folgt eine vorläufige Bewertung der Programme.

3.1 Anforderungen

Bei der Suche nach geeigneten Tools müssen die folgende Anforderungen erfüllen, um in die Liste der potenziell brauchbaren Systeme aufgenommen zu werden:

- Das Tool besitzt Funktionalitäten zur Berechnung von Metriken (direkt oder indirekt über Plug-Ins, Zusatzmodule oder ähnliches)
- Das Tool kann COBOL-Quelltexte verarbeiten oder es besteht die Möglichkeit, diese Funktion hinzuzufügen (direkt oder indirekt über Plug-Ins, Zusatzmodule, Programmierung, etc.)

Diese Anforderungen ergeben sich unmittelbar aus der Zielsetzung der Diplomarbeit.

3.2 Weitere Kriterien

Zwecks der späteren Entscheidung für eine engere Auswahl von Tools werden - soweit dies möglich ist - zusätzlich die unten genannten Kriterien und Eigenschaften der Tools erhoben und aufgelistet. Ein Teil dieser Kriterien wurde aufbauend auf den zuvor genannten Anforderungen und der Zielsetzung dieser Arbeit definiert. Der andere Teil wurde im Gespräch mit den betreuenden Personen dieser Arbeit und den Entwicklern der Debeka erhoben. Die Markierung (I) bezeichnet rein informative und (R) für die spätere Auswahl relevante Kriterien. Diese Unterteilung wurde im Gespräch mit den betreuenden Personen festgelegt.

- (I) Hersteller
- (I) Leistungsumfang (d.h. weitere Funktionalitäten neben der Metrikberechnung)
- (I) Existenz einer Programmierschnittstelle (API)
- (I) Art der internen Arbeitsweise (etwa zeilenbasiert, baumbasiert, graphbasiert, etc.)
- (I) Unterstützte Betriebssysteme und weitere Voraussetzungen
- (I) Unterstützte COBOL-Versionen und -Erweiterungen
- (I) Bedienungsmöglichkeiten: GUI / Konsole
- (I) Verarbeitung nur von einzelnen Codedateien oder ganzen Softwaresystemen möglich
- (I) Projekt- und Firmenreferenzen
- (R) Lizenzmodell und eventuelle Kosten
- (R) Verfügbarkeit einer Demoversion (nur relevant bei kommerzieller Software)

- (R) Aktivität des Projektes (d.h. es erfolgt noch eine Weiterentwicklung)
- (R) Möglichkeit zur Definition eigener Metriken
- (R) Existenz von Export- und Importfunktion sowie dabei mögliche Datenformate.
- (R) Liste der vom Tool unterstützten Metriken
- (R) Visualisierungsmöglichkeit der Ergebnisse
- (I) Performanz und Ressourcenverbrauch - Dieser Punkt wird hier zwar bereits aufgeführt, findet im Folgenden aber noch keine Anwendung, da der Aufwand hierfür zu groß und auch nicht bei allen Tools eine Messung überhaupt möglich wäre.

3.3 Tools

An dieser Stelle werden die einzelnen Tools vorgestellt. Neben einer Beschreibung der Tools hinsichtlich der Kriterien aus Abschnitt 3.2 liegt dabei der Fokus auf den Besonderheiten der jeweiligen Software. Zusätzlich findet sich zu jedem Programm eine tabellarische Übersicht über dessen Eigenschaften.

Die Tools wurden im Internet recherchiert, wobei stets nach „COBOL“ in Kombination mit einem oder mehreren der folgenden Schlüsselbegriffe gesucht wurde:

- Metrik / Metriken
- statische Analyse
- Codeanalyse
- Softwareanalyse
- Codequalität
- Qualitätssicherung
- Qualitätsanalyse

Darüber hinaus wurde auch die jeweils äquivalente englische Bezeichnung verwendet. Aus den Suchergebnissen wurden nur diejenigen Treffer weiter untersucht, welche auf ein Tool (und nicht nur eine Dienstleistung oder anderes) verwiesen, und bei denen das Tool den Anforderungen aus Abschnitt 3.1 entsprach.

3.3.1 AREDIS / Application Miner (QUINTEC GmbH)

Der *AREDIS / Application Miner*[10] (AREDIS / AM) wurde von der QUINTEC Informationstechnologie GmbH als Suite zur Softwareanalyse entwickelt. Das Tool kann neben COBOL auch Quellcode der Sprachen PL/1, Assembler, C, C++, C#, Java, JCL und Visual Basic verarbeiten.

Neben der statischen Analyse von Quellcode beherrscht der Application Miner auch die Verarbeitung von strukturierten Dokumenten, Projektplänen, Datenbankstrukturen und -inhalten

[10] `http://www.quintec.de/74.html` und `http://www.quintec-it.de/pages/03_Produkte/01_AREDIS_AM.htm`

und bringt hierfür Parser für SQL, DL/1, CICS und Präprozessorsprachen mit. Darüber hinaus besitzt das Tool Funktionen zur Laufzeitanalyse. Neben der Analyse zum Zwecke des Reengineerings liegen die Einsatzgebiete auch in der Projektsteuerung, der Nachdokumentation, dem Codereview sowie der Überprüfung der Einhaltung von Code- und Architekturkonventionen. Als Teil der Analysefunktionalität kann das Tool auch Metriken berechnen.

AREDIS / AM arbeitet nach dem Prinzip von Extraktion, Transformation und Anfrage. Die extrahierten Daten werden dabei in einem Repository gespeichert, auf welchem die Analysefunktionen aufsetzen. Quellcode wird dabei intern als abstrakter Syntaxbaum[11] repräsentiert. Die Ergebnisse können in tabellarischer und grafischer Form ausgegeben werden, ein Export in XML-, XMI- und relationalen Datenformaten ist möglich.

Die Bedienung des Application Miners erfolgt mittels einer GUI, eine Integration in Eclipse und einige CASE[12]-Tools ist möglich.

Das Tool kann etwas mehr als 50 Metriken über COBOL-Code berechnen, eine genaue Liste hat der Hersteller bislang noch nicht zur Verfügung gestellt, jedoch eine grobe Einteilung aufgezeigt:

- Größenmaße (u.a. LoC, Function Points)
- Komplexitätsmaße (u.a. McCabe)
- Qualitätsmaße (u.a. hinsichtlich Namens- und Programmiervorgaben)

Eine Möglichkeit zur Definition eigener Metriken ist nicht vorhanden, der Hersteller bietet aber die Möglichkeit einer Implementierung auf Anfrage an. Eine API ist bei diesem Programm vom Hersteller nicht für die Benutzer zugänglich gemacht.

Der Application Miner kann Quellcode in folgenden COBOL-Versionen verarbeiten:

- COBOL-60
- COBOL-74
- COBOL-85
- COBOL-2002

Quintec verweist in den Kundenreferenzen[13] vor allem auf Unternehmen aus dem Finanzsektor (u.a. Deutsche Bank, Deutsche Börse, Hypo Vereinsbank, Landesbank Baden-Württemberg, Sparkassen Informatik) und aus der Informationstechnik (u.a. Deutsche Telekom, IBM, Philips, Siemens), sowie auf diverse Behörden (u.a. Bundesamt für Güterverkehr, Bundesverwaltungsamt, Bundesamt für Wehrtechnik und Beschaffung, Logistkamt der Bundeswehr) und einige weitere größere und kleinere Unternehmen (u.a. Bayer, Deutsche Post, DEUTZ, VW).

AREDIS / AM liegt aktuell in Version 1.6 vom Mai 2009 (Stand: März 2010) vor. Die Extraktoren und Analysetools sind lauffähig auf IBM z/OS, Linux, Unix, Sun Solaris und Microsoft

[11] Im Englischen auch als *abstract syntax tree (AST)* bezeichnet.
[12] CASE: Computer-Aided Software Engineering
[13] `http://www.quintec-it.de/pages/07_Unternehmen/05_Kunden.htm`

Windows, die Visualisierungsoberfläche ausschließlich unter Windows ab Version 2000. Eine Unternehmenslizenz mit Unterstützung für eine Programmiersprache kostet 10.000 €, der Hersteller stellt leider keine Demoversion zur Verfügung. Eine Übersicht der relevanten Kriterien findet sich in Tabelle 1.

Tool	AREDIS / Application Miner (AREDIS / AM)
Hersteller	QUINTEC Informationstechnologie GmbH
Lizenz und Preis	kommerziell - ab 10.000 €
Version und Datum	1.6 (Mai 2009)
Referenzen	Finanzsektor, IT, Behörden, sonstige (haupts. Großunternehmen)
Demoversion verfügbar	nein
Benutzerdefinierte Metriken	nein (nur durch den Hersteller auf Anfrage)
API	nein
Export / Import	ja (XML, XMI, relational) / nein
Arbeitsweise	baumbasiert
Systemvoraussetzungen	z/OS, Linux, Unix, Solaris, Windows
COBOL-Versionen	COBOL-60, -74, -85, -2002
Bedienung	GUI, Eclipse-Integration, CASE-Tool-Integration
Verarbeitung ganzer Systeme	ja
Visualisierung	ja

Tabelle 1: Übersicht zu AREDIS / Application Miner

3.3.2 AUDITOR für COBOL (CC GmbH)

Unter der Bezeichnung *AUDITOR für COBOL*[14] bietet die CC GmbH ein Tool zur Berechnung von Softwaremetriken an. Der Schwerpunkt liegt dabei vor allem auf der Qualitätssicherung. Es existieren auch Varianten des AUDITORs für ABAP, C, C++, Java, Natural und PL/1. Metriken können sowohl über einzelne Programme oder Quelldateien wie auch über ganze Softwaresysteme berechnet werden.

Neben der Programmanalyse durch Metrikberechnung und der Erstellung von Metrikreports liegt die Hauptaufgabe des Programms in der Unterstützung von Entwicklungs- und Wartungsprozessen. Dabei kann über eine Auswahl von Metriken und zugehöriger Schwellwerte ein eigener Regelsatz definiert werden, auf dessen Einhaltung der AUDITOR prüft.

Während der Analyse kann von diesem Tool ein Repository erzeugt werden, so dass spätere Metrikberechnungen auf dieser Basis ohne erneute Verarbeitung von Quellcode möglich sind. Die Ergebnisse und Reports der Analyse können als HTML- oder XML-Dateien ausgegeben werden. Eine Visualisierung der Analyseergebnisse ist vom Programm selbst aus nicht möglich, die Ausgabe im XML-Format wurde aber zur Vereinfachung der Visualisierung mittels externer Tools eingeführt.

Die Bedienung des Tools kann sowohl über eine GUI als auch über einen parametrierten Aufruf eines Konsolenprogramms erfolgen. Zusätzlich steht ein Plug-In zur Integration in Eclipse zur

[14] `http://www.cc-ag.de/index.php?id=1069&L=0`

Verfügung. Mittels des Plug-Ins kann, neben der Anzeige der Analyseergebnisse, auch über eine Liste mit gefundenen Verletzungen des Regelsatzes direkt die jeweils relevante Stelle im Quellcode angewählt werden.

Der AUDITOR kann über COBOL-Quellcode etwa 830 verschiedene Metriken berechnen. Eine Liste der Metriken findet sich in [CC2007], wobei manche der Einträge - unter gleicher Beschreibung und ohne erkennbare Unterschiede - mehrfach aufgeführt sind. Darüber hinaus ist auch die Möglichkeit zur Definition eigener Metriken gegeben. Dazu ist das Einfügen des für die Berechnung nötigen COBOL-Codes in entsprechend bereitgestellte Codefragmente möglich.

CC verweist in den Kundenreferenzen[15] vor allem auf Unternehmen aus dem Finanzsektor (u.a. Bank of Canada, BHW, Commerzbank, DekaBank, Dresdner Bank), aus dem Versicherungssektor (u.a. HUK-Coburg, Rechtschutzversicherung, R+V, Shelter Insurance) und aus der Informationstechnik (u.a. ALLDATA Systems, First Data Resources, ivv-Informationsverarbeitung für Versicherungen, TUI Info Tec), sowie auf einige weitere (größere wie mittlere) Unternehmen und auf ein paar Behörden (u.a. Mercedes, REWE, Bundesamt für den Zivildienst). Darüber hinaus werden Partnerschaften mit u.a. IBM, Oracle und Sun angegeben.

AUDITOR für COBOL liegt aktuell in Version 6.1.1 vom Juni 2009 (Stand: März 2010) vor. Das Programm ist lauffähig unter MS DOS, IBM z/OS, Microsoft Windows ab Version 3.x, Unix, AIX und allen Java-unterstützten Plattformen. Das Eclipse-Plug-In benötigt darüber hinaus Eclipse ab Version 3.0. Eine Unternehmenslizenz für eine beliebige Anzahl von Installationen kostet 64.000 €. Eine Demoversion ist ebenfalls verfügbar - dabei handelt es sich bewusst um eine ältere Version, welche in der Anzahl der Metriken und der Zahl der maximal verarbeiteten Codezeilen limitiert ist. Eine Übersicht der relevanten Kriterien findet sich in Tabelle 2.

Tool	AUDITOR für COBOL
Hersteller	CC GmbH
Lizenz und Preis	kommerziell - 64.000 €
Version und Datum	6.1.1 (Juni 2009)
Referenzen	Finanzsektor, Versicherungen, IT, sonstige
Demoversion verfügbar	ja
Benutzerdefinierte Metriken	ja
API	ja
Export / Import	ja (XML, HTML) / nein
Arbeitsweise	unbekannt
Systemvoraussetzungen	z/OS, Unix, Windows, DOS, AIX, Java-unterstützte Systeme
COBOL-Versionen	COBOL-60, -74, -85, -2002
Bedienung	GUI, Konsole, Eclipse-Plug-In
Verarbeitung ganzer Systeme	ja
Visualisierung	nein

Tabelle 2: Übersicht zu AUDITOR für COBOL

[15] http://www.cc-ag.de/index.php?id=1022&L=0

3.3.3 COBOL Flow Graph Manipulator (pro et con GmbH)

Beim *COBOL Flow Graph Manipulator*[16] (FGM) von der pro et con Innovative Informatikanwendungen GmbH handelt es sich um ein Tool für Analyse, Reengineering und Redokumentation von Softwaresystemen. Zwar lassen sich auch einzelne Programme oder Module verarbeiten, primär ist das Programm jedoch für ganze Systeme gedacht.

Das Programm ist in der Lage, Kontroll- und Datenflüsse zu analysieren und visualisieren. Anhand der Kontrollflussanalyse kann toter (d.h. nicht erreichbarer) Code identifiziert und sowohl im Code als auch in der Graphansicht hervorgehoben werden. Des Weiteren können Aufrufgraphen berechnet und dargestellt werden. Darüber hinaus können Metriken über das analysierte Softwaresystem berechnet werden. Die Ergebnisse aus allen genannten Analyseschritten können anschließend in einem Bericht zusammengefasst ausgegeben und so zur Systemdokumentation hinzugefügt werden.

Der FGM arbeitet nach dem Prinzip von Extraktion, Transformation und Anfrage. Während der Extraktion wird der Code intern in einen Graph umgewandelt und in einem Repository abgelegt. Zusätzlich werden weitere Informationen wie etwa Aufrufstruktur, Datenbank- und Dateibenutzung analysiert und in einer SQL-Datenbank gespeichert. Die beiden Datenspeicher können dabei optional auch auf einem eigenen Server liegen. Auf den Inhalten dieser Speicher setzen anschließend alle zuvor genannten Analysefunktionen des FGM auf. Die Ergebnisse der Analyse können in tabellarischer Form und - wo möglich - als Graph dargestellt oder als csv[17]-/html- beziehungsweise bmp-/vcg[18]-Datei exportiert werden.

Die Bedienung des FGM erfolgt ausschließlich mittels einer GUI, welche als Eclipse Rich Client[19] realisiert ist.

Der Flow Graph Manipulator kann folgende Metriken berechnen:

- Halstead-Metriken
- McCabe-Metrik
- Verschachtelungstiefe
- Lines of Code

Mittels einer eigenen Abfragesprache (FGM-LANguage) können Anfragen an die Repositories gestellt werden. Neben einer Bibliothek mit bereits vorgefertigten Skripten können auch mittels der vorhandenen API eigene Skripte erstellt werden. Hiermit bestünde eventuell auch die Möglichkeit zur Realisierung eigener Metriken oder weiterer Exportformate. Auf jeden Fall bietet der Hersteller aber die Möglichkeit einer Implementierung weiterer Metriken auf Anfrage an.

[16] `http://www.proetcon.de/de/products/fgm_cob_descr_d.html`

[17] Comma Separated Value (CSV) ist ein einfaches Dateiformat für Tabellen.

[18] Visualization of Compiler Graphs (VCG) ist ein Dateiformat zur Graphdarstellung, siehe `http://rw4.cs.uni-sb.de/users/sander/html/gsvcg1.html`.

[19] `http://wiki.eclipse.org/index.php/Rich_Client_Platform`

Der FGM kann Quellcode in folgenden COBOL-Versionen und -Dialekten verarbeiten:

- COBOL-85
- IBM COBOL LE370
- IBM OS/VS COBOL
- IBM OS/VS COBOL II
- NonStop COBOL85 und SCREEN COBOL
- MicroFocus COBOL

pro et con verweist in den Kundenreferenzen[20] vor allem auf Unternehmen aus der Informationstechnik (u.a. Allen Systems Group, Amadeus Germany, msg systems, T-Systems), alle weiteren Unternehmen bilden ein recht breites Spektrum ab (u.a. adept consult, Arbeiterwohlfahrt, Bank-Verlag, Berufsakademie Sachsen, Cosmos Direkt, Flughafen Frankfurt/Main, MAN Nutzfahrzeuge).

Der Flow Graph Manipulator liegt aktuell in Version 3.0 vom Juni 2009 (Stand: Juni 2010) vor und läuft auf Microsoft Windows ab Version 2000. Für das Datenbankserversystem wird MySQL Classic ab Version 3.23 benötigt. Eine Einzelplatzlizenz kostet 9.750 €, eine Demoversion ist ebenfalls verfügbar. Eine neue Version 3.0 befindet sich derzeit in der Entwicklung und soll neben Quellcode auch weitere Entwurfsdokumente analysieren können, siehe [Beier2009]. Eine Übersicht der relevanten Kriterien findet sich in Tabelle 3.

Tool	COBOL Flow Graph Manipulator (FGM)
Hersteller	pro et con Innovative Informatikanwendungen GmbH
Lizenz und Preis	kommerziell - ab 9.750 €
Version und Datum	3.0 (Juni 2010)
Referenzen	IT, sonstige
Demoversion verfügbar	ja
Benutzerdefinierte Metriken	ja
API	ja
Export / Import	ja (CSV, HTML, BMP, VCG) / nein
Arbeitsweise	graphbasiert
Systemvoraussetzungen	Windows, MySQL Classic
COBOL-Versionen	COBOL-85; MicroFocus COBOL; NonStop COBOL85, SCREEN COBOL; IBM COBOL LE370, OS/VS COBOL, OS/VS COBOL II
Bedienung	Eclipse-GUI
Verarbeitung ganzer Systeme	ja
Visualisierung	ja

Tabelle 3: Übersicht zu COBOL Flow Graph Manipulator

[20] `http://www.proetcon.de/de/company/references_d.html`

3.3.4 Continous Quality Assessment Toolkit (Technische Universität München)

Das *Continous Quality Assessment Toolkit*[21] (ConQAT) wurde an der Technischen Universität München entwickelt. Das Programm dient der kontinuierlichen Qualitätskontrolle bei der Entwicklung von Softwaresystemen. Die Entwickler haben dabei einen Schwerpunkt auf die Konzeption als modulares und leicht erweiterbares System gesetzt. Die Verarbeitung von Quellcode in ABAP, ADA, C, C++, C#, COBOL, Java, Visual Basic, PL/1 und PL/SQL ist möglich.

Ursprünglich war ConQAT als variables Kontrollinstrument konzeptioniert, welches sich auch ohne Programmierkenntnisse konfigurieren lässt. Daneben ist inzwischen aber auch die Klon-Erkennung[22] als Anwendungsgebiet hinzugekommen. Als drittes wurden Funktionen zur Architekturanalyse und zur Überprüfung der Architekturkonformität von Systemen hinzugefügt. Dabei werden zur Bewertung jeweils Metriken berechnet, was bei diesem Tool jedoch eher als Hilfsmittel und nicht als Hauptzweck geschieht. Die Ergebnisse aller Funktionen werden als HTML- und XML-Dateien ausgegeben, wobei teilweise auch eine Visualisierung möglich ist.

Die zentrale Instanz in der Arbeitsweise des Tools ist die *ConQAT Engine*. Diese steuert - entsprechend der gegebenen Konfiguration - die sogenannten Prozessoren. Bei den Prozessoren handelt es sich um diejenigen Module und Bibliotheken, welche die eigentliche Analyse durchführen. Dabei können Prozessoren verkettet arbeiten, also etwa ein Prozessor *C* verarbeitet mit den Ausgaben der Prozessoren *A* und *B*. Bei der Verarbeitung eines Softwaresystems wird von ConQAT absichtlich *kein* zentrales Repository aufgebaut. Dies wird damit begründet, dass die dabei stattfindende Transformation in eine andere Repräsentationsform (wie etwa einen Graph) bestimmte Analysefunktionen erschweren oder unmöglich machen würde. Braucht einer der Prozessoren also etwa eine Graphrepräsentation als Eingabe, so muss diesem ein entsprechender Prozessor vorgeschaltet werden. Damit nicht für jeden Analysedurchlauf eine komplette erneute Berechnung nötig ist, sind zur Leistungssteigerung Caching-Funktionen vorhanden.

Die Bedienung von ConQAT erfolgt mittels einer auf Eclipse aufsetzenden GUI, wobei ein darauf angepasstes und entsprechend vorkonfiguriertes Eclipse bereits Bestandteil der Windows- und der MacOS-Version ist. Darüber hinaus lässt sich das Paket (etwa auf anderen Plattformen) auch in ein reguläres Eclipse über dessen integrierte Paketinstallationsfunktionalität einbinden. Des Weiteren lässt sich die ConQAT Engine auch als parametriertes Kommandozeilenprogramm ausführen und so etwa auch automatisiert in Build-Systemen verwenden.

ConQAT kann folgende Metriken berechnen:

- LoC: Lines of code.
- Clone LoC: Cloned lines of code.
- Units: Number of units contained in the analyzed files. Typically, units are statements.

[21] `http://conqat.cs.tum.edu/`

[22] Klon-Erkennung bezeichnet die Erkennung von kopierten (und danach gegebenenfalls leicht modifizierten), somit also identischen oder sehr ähnlichen, Abschnitten im Quellcode.

- Clone Units: Cloned units, i.e., cloned statements.
- RFSS: Redundancy free source statements. That is, the number of statements from a hypothetical system from which all cloning is perfectly removed.
- UnitCoverage: Probability that an arbitrarily chosen statement is part of a clone.
- Clone Count: Number of clones.

Da die APIs mittels der Quelltexte und der ausführlichen Javadoc-basierten Dokumentation zugänglich gemacht sind und mit dem modularen Konzept der Prozessoren eine Möglichkeit für eigene Erweiterungen besteht, könnten über eigene Prozessoren auch Funktionen zur Berechnung weiterer Metriken hinzugefügt werden.

ConQAT kann Quellcode in folgenden COBOL-Versionen verarbeiten:

- COBOL-60
- COBOL-74
- COBOL-85

Die TU München stellt zu ConQAT keine Kunden- oder Projektreferenzen zur Verfügung.

ConQAT liegt aktuell in Version 2.6 vom Februar 2010 (Stand: März 2010) vor, benötigt Java 1.6 und läuft auf allen von dieser Java-Version unterstützten Plattformen. Für einige Analysekomponenten wird darüber hinaus das Program *dot* aus dem GraphViz-Paket[23] und für einige das Microsoft .NET Framework 2.0[24] benötigt. Die Einbindung in Eclipse ab Version 3.5 ist möglich. Das Programm ist kostenlos und quelloffen unter der Apache License 2.0[25] verfügbar. Eine Übersicht der relevanten Kriterien findet sich in Tabelle 4.

Tool	Continous Quality Assessment Toolkit (ConQAT)
Hersteller	Technische Universität München
Lizenz und Preis	Apache License 2.0 - kostenlos
Version und Datum	2.6 (Februar 2010)
Referenzen	unbekannt
Demoversion verfügbar	nein (da Vollversion frei verfügbar)
Benutzerdefinierte Metriken	ja
API	ja
Export / Import	ja (XML, HTML) / nein
Arbeitsweise	teilweise zeilen-, baum- und graphbasiert
Systemvoraussetzungen	Alle Java-unterstützten Plattformen, GraphViz(optional), .NET Framework 2.0 (optional)
COBOL-Versionen	COBOL-60, -74, -85
Bedienung	Eclipse-GUI und -Integration, Konsole
Verarbeitung ganzer Systeme	ja
Visualisierung	ja

Tabelle 4: Übersicht zu ConQAT

[23] `http://www.graphviz.org/`
[24] `http://www.microsoft.com/germany/msdn/netframework/`
[25] `http://www.apache.org/licenses/LICENSE-2.0`

3.3.5 DMS COBOL Source Code Metrics (Semantic Designs, Inc.)

DMS COBOL Source Code Metrics[26] ist Teil des *DMS Reengineering Toolkit* von Semantic Designs, Inc.. Das DMS Reengineering Toolkit ist eine Suite von Programmen zur Analyse, Modifikation, Übersetzung und Generierung von Softwaresystemen. Das Toolkit bringt dabei Unterstützung für diverse Programmiersprachen mit, von den Metrik-Tools existieren neben der COBOL-Version auch noch Varianten für C#, Java, VBScript und Logix5000.

Die Programme des DMS Toolkit bauen intern auf einer modularen Struktur von Komponenten aus der Compilertechnik auf. Je nach Programm kommt dabei eine Kombination aus Parsern, Analysatoren, Transformierern und Formatierern zum Einsatz. Dabei sind die einzelnen Komponenten so flexibel, dass sie oftmals in mehreren der Programme Verwendung finden. So dient etwa der COBOL-Parser als Grundlage für alle Programme zur Verarbeitung von COBOL-Code oder das stets gleiche Metrikberechnungsmodul setzt je nach Sprache auf einem anderen Parser auf. Dies zeigt jedoch nur den Aufbau und die gemeinsame Grundstruktur der Programme, DMS COBOL Metrics ist ein in sich geschlossenes Programm.

DMS COBOL Metrics arbeitet also nach dem Prinzip von Extraktion, Transformation und Anfrage. Die Möglichkeit zur Zwischenspeicherung der extrahierten Daten als Repository ist dabei nicht gegeben. Die Metriken werden sowohl auf die Gesamtmenge analysierten Codes als auch aufgeteilt auf Datei-, Division-, Section- und Paragraph-Ebene[27] berechnet. Die Ergebnisse können im Text- oder XML-Format ausgegeben werden, wobei in letzterem Fall die Struktur und die Formatierung festgelegt werden können. Eine Visualisierung der Ergebnisse wird nicht unterstützt.

Die Bedienung erfolgt über eine GUI. Über diese werden aber nur die eigentlichen, im Hintergrund arbeitenden Kommandozeilenprogramme gesteuert.

DMS COBOL Metrics kann die folgenden Metriken berechnen:

- Source Lines of Code (SLOC)
- Noncommented Source Lines of Code
- Comment Lines
- Number of Methods
- Decision Density
- Cyclomatic Complexity
- Maximum loop nesting
- Maximum conditional nesting
- Halstead measures (volume, difficulty and effort)
- Software Engineering Institute (SEI) Maintainability Index

Die Definition eigener Metriken durch den Benutzer ist nicht möglich. Eine für den Benutzer zugängliche API existiert nicht.

[26] `http://www.semanticdesigns.com/Products/Metrics/COBOLMetrics.html`
[27] Division, Section und Paragraph dienen der Unterteilung und Strukturierung innerhalb einer COBOL-Datei.

Das Tool kann Quellcode in folgenden COBOL-Versionen verarbeiten:

- COBOL-85
- IBM VS COBOL II
- IBM Enterprise COBOL

Semantic Designs bietet keine expliziten Verweise auf Kundenreferenzen, die Webseite mit Ankündigungen [28] lässt aber im Wesentlichen auf Kunden aus dem US-Militär und der Rüstungsindustrie schließen (u.a. US Navy, US Air Force, Northrop Grumman). Darüber hinaus finden sich einige allgemein eher unbekannte Firmen aus der IT, welche aber nicht Kunden, sondern Partner sind.

DMS COBOL Metrics liegt aktuell in Version 1.1.20 vom Januar 2010 (Stand: April 2010) vor. Das Programm läuft ausschließlich auf Microsoft Windows ab Version 2000. Der Hersteller bietet (jedoch nur auf Anfrage) eine Demoversion an. Weitergehende Informationen - insbesondere zum Preis einer Lizenz - sind beim Hersteller angefordert worden, es erfolgte jedoch keine Reaktion außer der Bereitstellung der Demo. Eine Übersicht der relevanten Kriterien findet sich in Tabelle 5.

Tool	DMS COBOL Source Code Metrics
Hersteller	Semantic Designs, Inc.
Lizenz und Preis	Kommerziell - Preis unbekannt
Version und Datum	1.1.20 (Januar 2010)
Referenzen	Militär, Rüstungsindustrie, IT
Demoversion verfügbar	ja
Benutzerdefinierte Metriken	nein
API	nein
Export / Import	ja (XML, Text) / nein
Arbeitsweise	baumbasiert
Systemvoraussetzungen	Windows
COBOL-Versionen	COBOL-85; IBM VS COBOL II, Enterprise COBOL
Bedienung	GUI, Konsole
Verarbeitung ganzer Systeme	ja
Visualisierung	nein

Tabelle 5: Übersicht zu DMS COBOL Source Code Metrics

3.3.6 Eclipse Metrics Plug-In (Frank Sauer)

Das *Eclipse Metrics Plug-In*[29] wurde von Frank Sauer entwickelt. Wie der Name bereits nahelegt, handelt es sich hierbei um ein Plug-In zur Metrikberechnung für Eclipse. Dabei wird ausschließlich Quellcode in Java zur Analyse unterstützt. Eine Berechnung ist sowohl über einzelne Klassen als auch über Pakete („Java Packages“) oder Projekte möglich.

[28] `http://www.semanticdesigns.com/Announce/`
[29] `http://metrics.sourceforge.net/`

Als einzige weitere Funktion (neben der Berechnung von Metriken) kann das Plug-In auch die Abhängigkeiten zwischen Paketen analysieren. Die dabei berechneten Abhängigkeitsgraphen können graphisch dargestellt werden. Die berechneten Metriken werden tabellarisch dargestellt. Zu jedem berechneten Wert wird auch der Durchschnittswert, die Standardabweichung und das Maximum angezeigt. Der für das Maximum relevante Code kann direkt aus dem Plug-In heraus angezeigt werden. Ein Export der berechneten Metriken im XML-Format ist möglich.

Die Bedienung des Plug-Ins erfolgt über die Eclipse-GUI. Alternativ dazu ist eine Steuerung über die Kommandozeile möglich.

Das Metrics Plug-In kann die folgenden Metriken berechnen:

- Number of Classes
- Number of Children
- Number of Interfaces
- Number of Overridden Methods
- Number of Methods
- Number of Static Methods
- Number of Fields
- Number of Attributes
- Number of Packages
- Number of static Attributes
- Number of Parameters
- Depth of Inheritance Tree
- Nested Block Depth
- Total lines of code
- Method lines of code
- Specialization Index
- McCabe Cyclomatic Complexity
- Weighted Methods per Class
- Lack of Cohesion of Methods
- Afferent Coupling
- Efferent Coupling
- Instability
- Abstractness
- Normalized Distance from Main Sequence

Eine Beschreibung der einzelnen Metriken findet sich in [Henderson1996], [Martin1994] und [Martin2002]. Eine Möglichkeit zur Definition eigener Metriken ist nicht vorhanden. Da das Plug-In jedoch als Open Source frei verfügbar ist, ist somit auch die API zugänglich und eine Erweiterung mittels Programmierung neuer Funktionen denkbar.

Frank Sauer stellt zum Eclipse Metrics Plug-In keine Kunden- oder Projektreferenzen zur Verfügung.

Das Eclipse Metrics Plug-In liegt aktuell in Version 1.3.6 vom Juli 2005 (Stand: April 2010) vor, so dass eine aktive Weiterentwicklung fraglich ist. Das Plug-In benötigt Eclipse ab Version 3.1. Es steht quelloffen unter der Common Public License (CPL) 1.0[30] zur kostenlosen Verfügung. Eine Übersicht der relevanten Kriterien findet sich in Tabelle 6.

Tool	Eclipse Metrics Plug-In
Hersteller	Frank Sauer
Lizenz und Preis	CPL 1.0 - kostenlos
Version und Datum	1.3.6 (Juli 2005)
Referenzen	unbekannt
Demoversion verfügbar	nein (da Vollversion frei verfügbar)
Benutzerdefinierte Metriken	nein
API	ja
Export / Import	ja (XML) / nein
Arbeitsweise	unbekannt
Systemvoraussetzungen	Eclipse ab Version 3.1
COBOL-Versionen	keine
Bedienung	GUI, Konsole
Verarbeitung ganzer Systeme	ja
Visualisierung	ja

Tabelle 6: Übersicht zum Eclipse Metrics Plug-In

3.3.7 Generische Umgebung zum Programmverstehen (Universität Koblenz-Landau)

Im Rahmen der Forschung des Instituts für Softwaretechnik an der Universität Koblenz-Landau ist die *Generische Umgebung zum Programmverstehen*[31] (GUPRO) entwickelt worden. Die hierzu gehörigen Werkzeuge arbeiten nach dem Prinzip der Extraktion, Transformation und Anfrage. Die interne Repräsentation der extrahierten Daten erfolgt in graphbasierter Form, genauer gesagt als TGraphen. Hierbei handelt es sich um typisierte, attributierte, gerichtete und geordnete Graphen.

An dieser Stelle muss auch kurz auf die Versionen und die Gesamtstruktur der zugehörigen, historisch gewachsenen Programme eingegangen werden. Zunächst einmal muss grundsätzlich zwischen dem ursprünglichen (dem eigentlichen) GUPRO und den Nachfolgetools unterschieden werden. Das ursprüngliche GUPRO wurde in C++ realisiert und nur für diese Version existiert eine Oberfläche zur zentralen Bedienung aller zugehörigen Tools. Eines hiervon ist der Extraktor für C++-Quellcode - der einzige realisierte Extraktor für diese Version. Die Kernkomponente von GUPRO ist das Graphenlabor (GraLab), welches die Funktionen zur Erstellung und Bearbeitung von Graphen bereitstellt. Rund um GraLab findet sich eine Sammlung von weiteren Programmen wie etwa Optimierer oder Graphkonverter. Mit der *Graph Repository Query Language* (GreQL) existiert eine eigene Sprache zur Formulierung von Anfragen an

[30] `http://www.opensource.org/licenses/cpl1.0.php`

[31] `http://www.uni-koblenz-landau.de/koblenz/fb4/institute/IST/AGEbert/projekte/abgprojekte/GUPRO`

Graphen. Die Graph eXchange Language (GXL) dient als eigene Sprache zum Austausch von Graphdaten.

Die Nachfolgeprogramme von GUPRO sind in Java realisiert worden. Als Kern fungiert nun JGraLab, die Sprache GreQL2 ist ebenfalls eine Weiterentwicklung. Auch rund um JGraLab existieren bereits diverse weitere Tools, allerdings wurde bislang keine zentrale Benutzeroberfläche wie im ursprünglichen GUPRO implementiert. Bislang existieren nur Oberflächen für einzelne Teilfunktionen, etwa eine Weboberfläche zum Stellen von GreQL2-Anfragen. Grundsätzlich sind alle Tools über die Konsole aufrufbar, wie dies auch bei der alten Version der Fall ist. Der bisher einzige Extraktor kann Java-Quelltexte verarbeiten.

Die von (J)GraLab erzeugten Graphen können als Datei gespeichert werden und so als Repository für eine spätere Weiterverwendung dienen. Mittels Definition eines Graphschemas kann JGraLab die entsprechenden Klassen generieren, so dass sich der Grad der Granularität, also der Abstraktion über den Code, variieren lässt. Sowohl im Fall des C++- als auch des Javaextraktors wird eine sehr feingranulare Repräsentation erzeugt, welche jeweils alle Informationen aus dem Quellcode enthält. Dabei können in beiden Fällen nicht nur einzelne Dateien, sondern auch ganze Softwaresysteme verarbeitet werden.

Mangels eines entsprechenden Extraktors können bislang also keine COBOL-Systeme analysiert werden. Ebenso ist derzeit keine explizite Funktionalität zur Berechnung von Metriken vorhanden, wobei sich dies mittels entsprechender GreQL(2)-Anfragen realisieren ließe.

Als Software aus dem akademischen Umfeld existieren zu GUPRO, (J)GraLab, etc. Referenzen auf den Einsatz in Forschungsprojekten. Dabei waren häufiger auch Partner aus der Wirtschaft (u.a. pro et con, Debeka Gruppe, Amadeus Germany, SAP) und aus öffentlichen Einrichtungen (u.a. Bundesministerium für Bildung und Forschung, Deutsches Zentrum für Luft- und Raumfahrt) beteiligt.

GUPRO läuft unter Microsoft Windows, MacOS, Linux und Unix, die Entwicklung wurde 2005 weitestgehend beendet. JGraLab und die zugehörigen Programme laufen auf allen von Java unterstützten Plattformen und befinden sich nach wie vor in einer steten Weiterentwicklung. Die Tools stehen zu Forschungs- und nichtkommerziellen Einsatzzwecken quelloffen unter der GNU General Public License 2 (GPLv2)[32] zur Verfügung, zu kommerziellen Zwecken muss eine Lizenz erworben werden. Eine Übersicht der relevanten Kriterien findet sich in Tabelle 7.

3.3.8 Interstage Software Quality Analyzer (Fujitsu, Ltd.)

Der *Interstage Software Quality Analyzer*[33] (SQA) wurde von Fujitsu, Ltd. entwickelt. Die Software ist Bestandteil der Application and Service Management Suite, welche eine Reihe von Tools zur technischen und organisatorischen Unterstützung über den gesamten Software-Lebenszyklus

[32] `http://www.gnu.org/licenses/old-licenses/gpl-2.0.html`
[33] `http://www.fujitsu.com/global/services/software/interstage/sqa/`

Tool	Generische Umgebung zum Programmverstehen (GUPRO)
Hersteller	Universität Koblenz-Landau
Lizenz und Preis	GPLv2 - kostenlos (zu Forschungszwecken)
Version und Datum	GUPRO: 2005; JGraLab: Build 1709 (September 2009) -oder- aktuelle Entwicklerversionen
Referenzen	IT, öffentliche Einrichtungen
Demoversion verfügbar	nein (da Vollversion frei verfügbar)
Benutzerdefinierte Metriken	ja
API	ja
Export / Import	ja / ja
Arbeitsweise	graphbasiert
Systemvoraussetzungen	Alle Java-unterstützten Plattformen
COBOL-Versionen	keine
Bedienung	GUI (GUPRO), Konsole (JGraLab und neuere)
Verarbeitung ganzer Systeme	ja
Visualisierung	nein

Tabelle 7: Übersicht zu GUPRO

beinhaltet. Die Suite ist wiederum Teil der Instage-Produktlinie, unter der weitere Produkte für das Management von Geschäftsprozessen, Finanzdaten und IT-Infrastrukturen geführt werden.

Der SQA selbst dient der statischen Analyse von Quellcode mit dem Hauptaugenmerk auf der Bewertung von Qualitätsmerkmalen. Dabei können Softwaresysteme und Programme in Assembler, COBOL, Java, NATURAL und PL/1, Code in den Datenbanksprachen ADABAS, IMS, SQL und UDS, sowie CICS-, UTM- und JCL-Code (Job Control Language) analysiert werden.

Das Tool kann Abhängigkeiten und Struktur der verarbeiteten Quellen analysieren. Darüber hinaus können Verstöße gegen definierbare Programmier- und Qualitätsrichtlinien, problematischer und fehlerhafter Code aufgedeckt werden.

Der SQA ist unterteilt in zwei wesentliche Teile. Zum einen der Server, der die eigentliche Analyse durchführt und Ergebnisse, Reports und Arbeitsdaten speichert. Zum anderen die Clientsoftware, welche auf den Server zugreift und dessen Funktionen und Daten steuert bzw. abfragt. Die Ergebnisse können visualisiert und als HTML-, XML- oder CSV-Daten exportiert werden.

Die Bedienung erfolgt über eine als Eclipse-Plug-In realisierte GUI. Des Weiteren ist die Steuerung der grundlegenden Funktionen über die Kommandozeile ebenfalls möglich.

Der SQA kann folgende Metriken berechnen:

- Source Lines
- Function Points
- Extended Cyclomatic Complexity
- Percent of Comments
- Binary Decisions

- Afferent Couplings
- Efferent Couplings
- Distinct Operators
- Distinct Operands
- Total Operators
- Total Operands

Es besteht zusätzlich die Möglichkeit zur Definition eigener Metriken. Da diese auf Basis der zuvor genannten Metriken berechnet werden müssen, sind die Möglichkeiten hierzu jedoch recht eingeschränkt. Eine API ist nicht für den Benutzer zugänglich. Zwar ist die Möglichkeit zum Stellen von Anfragen in einer SQL-ähnlichen Sprache möglich, jedoch beschränkt sich der Funktionsumfang hierbei auf die Auswahl von Codeabschnitten und die zuvor genannten Metriken.

Der SQA kann Quellcode in folgenden COBOL-Versionen und -Erweiterungen verarbeiten:

- COBOL-60
- COBOL-74
- COBOL-85
- COBOL-2002 (jedoch ohne Objektorientierung)
- BS2000
- DEC
- Fujitsu
- Microfocus
- Tandem

Fujitsu bietet keine Verweise auf Kundenreferenzen, sondern lediglich eine Webseite mit Partnerfirmen[34] aus der IT-Branche (u.a. Infor, Software AG, IDS Scheer).

Interstage Software Quality Analyzer benötigt für den Server einen Microsoft Windows Server ab Version 2003, für den Client sind alle von der Ecplise-Plattform unterstützten Systeme möglich. Eine Demoversion wird auf Anfrage bereitgestellt, jedoch ist auch diese bereits kostenpflichtig. Von Fujitsu wurden Informationen zur aktuellen Versionsnummer nicht zur Verfügung gestellt, Angaben zum Preis der Demo- und Vollversion sowie der internen Arbeitsweise wurden ausdrücklich verweigert. Eine Übersicht der relevanten Kriterien findet sich in Tabelle 8.

3.3.9 IQ Developers Edition (McCabe Software)

McCabe Software liefert mit *IQ Developers Edition*[35] eine Software mit Hauptaugenmerk auf der Analyse der Sicherheit, Komplexität und Qualität von Softwaresystemen. IQ kann Quellcode

[34] `http://www.fujitsu.com/global/services/software/interstage/partners/InterstagePartners.html`

[35] `http://www.mccabe.com/iq_developers.htm`

Tool	Interstage Software Quality Analyzer (SQA)
Hersteller	Fujitsu, Ltd.
Lizenz und Preis	Kommerziell - Preis unbekannt
Version und Datum	unbekannt
Referenzen	unbekannt
Demoversion verfügbar	ja (kostenpflichtig)
Benutzerdefinierte Metriken	ja
API	nein
Export / Import	ja / nein
Arbeitsweise	unbekannt
Systemvoraussetzungen	Windows Server ab Version 2003 / Eclipse
COBOL-Versionen	COBOL-60, -74, -85, -2002 (ohne Objektorientierung); BS2000, DEC, Fujitsu, Microfocus, Tandem
Bedienung	GUI, Konsole
Verarbeitung ganzer Systeme	ja
Visualisierung	ja

Tabelle 8: Übersicht zu Interstage Software Quality Analyzer

in Ada, ASM86, C, C#, C++.NET, C++, COBOL, FORTRAN, JAVA, JSP, Perl, PL1, Visual Basic und Visual Basic.NET analysieren.

Der Schwerpunkt des Tools liegt in der Berechnung von Metriken über den Quellcode und ihrer Auswertung hinsichtlich der oben genannten Punkte. Unterstützend dazu können auch Kontrollflussdiagramme erzeugt und dargestellt, Datenfluss analysiert und redundanter Code aufgedeckt werden. Neben IQ Developers existiert IQ Test Team Edition, welche der Vorbereitung, Durchführung und Auswertung von Tests dient. Des Weiteren gibt es IQ Enterprise Edition, welche die Funktionen der beiden zuvor genannten Tools vereint und erweiterte Reportfunktionen, Änderungsanalyse, Reengineeringfunktionen und Statistiken über den Zeitverlauf von Projekten bietet.

IQ kann die durch die Analyse erzeugten Daten in einem Repository vorhalten. Die genaue Arbeitsweise des Tools ist nicht bekannt, die Funktionalitäten zur Berechnung von Kontroll- und Datenflüssen legen aber ein graph- oder zumindest baumbasiertes Vorgehen nahe. Die Analyseergebnisse können visualisiert und exportiert werden, die berechneten Metriken können dabei - gesondert von den Reports - im XML-Format gespeichert werden.

Die Bedienung des Tools erfolgt über eine GUI. Die eigentlichen Analyseprogramme können daneben auch über die Konsole aufgerufen werden. Für die Analyse von Java-Code besteht darüber hinaus die Möglichkeit der Einbindung in Eclipse über ein Plug-In.

IQ kann die folgenden Metriken berechnen:

- Cyclomatic Complexity Metric (v(G))
- Actual Complexity Metric (ac)
- Module Design Complexity Metric (iv(G))
- Essential Complexity Metric (ev(G))

- Pathological Complexity Metric (pv(G))
- Design Complexity Metric (S0)
- Integration Complexity Metric (S1)
- Object Integration Complexity Metric (OS1)
- Global Data Complexity Metric (gdv(G))
- Data Complexity Metric (DV)
- Tested Data Complexity Metric (TDV)
- Data Reference Metric (DR)
- Tested Data Reference Metric (TDR)
- Maintenance Severity Metric (maint_severity)
- Data Reference Severity Metric (DR_severity)
- Data Complexity Severity Metric (DV_severity)
- Global Data Severity Metric (gdv_severity)
- Percent Public Data (PCTPUB)
- Access to Public Data (PUBDATA)
- Percent of Unoverloaded Calls (PCTCALL)
- Number of Roots (ROOTCNT)
- Fan-in (FANIN)
- Maximum v(G) (MAXV)
- Maximum ev(G) (MAXEV)
- Hierarchy Quality(QUAL)
- Depth (DEPTH)
- Lack of Cohesion of Methods (LOCM)
- Number of Children (NOC)
- Response For a Class (RFC)
- Weighted Methods Per Class (WMC)
- Halstead Program Length
- Halstead Program Volume
- Halstead Program Level and Program Difficulty
- Halstead Intelligent Content
- Halstead Programming Effort
- Halstead Error Estimate
- Halstead Programming Time
- Line Count Software Metrics
- Lines of Code
- Lines of Comment
- Lines of Mixed Code and Comments
- Lines Left Blank

McCabe bietet keine Verweise auf Kundenreferenzen, sondern lediglich eine Webseite mit Partnerfirmen[36] aus der IT-Branche (u.a. InSource, Software Quality Associates).

Das Tool läuft auf allen von Java unterstützten Plattformen. Weitergehende Informationen hinsichtlich Existenz einer API, unterstützter COBOL-Versionen, aktuellem Versionsstand, Systemvoraussetzungen, Preis einer Lizenz und Möglichkeit einer Demoversion sind beim Hersteller angefordert worden, es erfolgte jedoch keinerlei Reaktion. Eine Übersicht der relevanten Kriterien findet sich in Tabelle 9.

Tool	IQ Developers Edition
Hersteller	McCabe Software
Lizenz und Preis	Kommerziell - Preis unbekannt
Version und Datum	unbekannt
Referenzen	unbekannt
Demoversion verfügbar	unbekannt
Benutzerdefinierte Metriken	unbekannt
API	unbekannt
Export / Import	ja (XML) / unbekannt
Arbeitsweise	unbekannt (wahrscheinlich baum- oder graphbasiert)
Systemvoraussetzungen	Alle Java-unterstützten Plattformen
COBOL-Versionen	unbekannt
Bedienung	GUI, Konsole, Eclipse-Plug-In (nur für Analyse von Java)
Verarbeitung ganzer Systeme	ja
Visualisierung	ja

Tabelle 9: Übersicht zu IQ

3.3.10 JDepend (Clarkware Consulting, Inc.)

JDepend[37] wurde durch Clarkware Consulting, Inc. entwickelt. Das Tool dient der Qualitätskontrolle von Java-Quellcode und arbeitet hauptsächlich auf Package-Ebene, auch wenn beliebig viele Packages in einem Analysedurchgang verarbeitet werden können.

Der Funktionsumfang von JDepend umfasst ausschließlich die Berechnung von Metriken mit Hinsicht auf die Bewertung von Erweiterbarkeit, Wartbarkeit und Wiederverwendbarkeit von Packages. Das Tool verarbeitet die Quellen intern auf Text- bzw. Zeilenbasis. Die Anzeige der Ergebnisse erfolgt auf einer einfachen Textbasis oder in Baumform, wobei letztere lediglich die Struktur von Packages und Subpackages widerspiegelt. Die Resultate können im Text- oder XML-Format exportiert werden. Beim XML-Export kann die Formatierung über ein XSLT-Stylesheet festgelegt werden, als Beispiel liegt auch ein zur Graphviz-kompatiblen DOT-Formatierung dienendes Stylesheet bei.

[36] `http://www.mccabe.com/partners.htm`
[37] `http://clarkware.com/software/JDepend.html`

Die Bedienung erfolgt ausschließlich über die Kommandozeilenparameter beim Aufruf, lediglich die Ausgabe in Baumform öffnet dabei ein eigenes Oberflächenfenster zur Ausgabe der Ergebnisse.

JDepend kann die folgenden Metriken berechnen:

- Number of Classes and Interfaces
- Afferent Couplings (Ca)
- Efferent Couplings (Ce)
- Abstractness (A)
- Instability (I)
- Distance from the Main Sequence (D)
- Package Dependency Cycles

Eine Möglichkeit zur Definition eigener Metriken ist nicht vorhanden. Da das Tool jedoch als Open Source frei verfügbar ist, ist somit auch die API zugänglich und eine Erweiterung mittels Programmierung neuer Funktionen denkbar.

Clarkware Consulting stellt zu JDepend keine Kunden- oder Projektreferenzen zur Verfügung.

JDepend liegt aktuell in Version 2.9 vom Dezember 2004 (Stand: April 2010) vor. Da das Tool darüber hinaus auch über die Navigation von der Startseite der Clarkware-Website nicht erreichbar ist, ist eine aktive Weiterentwicklung fraglich. JDepend benötigt mindestens Java 1.5 und läuft auf allen von dieser Java-Version unterstützten Plattformen. Es steht quelloffen unter der BSD License [38] zur kostenlosen Verfügung. Eine Übersicht der relevanten Kriterien findet sich in Tabelle 10.

Tool	JDepend
Hersteller	Clarkware Consulting, Inc.
Lizenz und Preis	BSD - kostenlos
Version und Datum	2.9 (Dezember 2004)
Referenzen	unbekannt
Demoversion verfügbar	nein (da Vollversion frei verfügbar)
Benutzerdefinierte Metriken	nein
API	ja
Export / Import	ja (Text, XML) / nein
Arbeitsweise	text- / zeilenbasiert
Systemvoraussetzungen	Java ab Version 1.5
COBOL-Versionen	keine
Bedienung	Konsole, GUI (nur Ergebnisanzeige)
Verarbeitung ganzer Systeme	ja
Visualisierung	nein

Tabelle 10: Übersicht zu JDepend

[38] `http://www.opensource.org/licenses/bsd-license.php`

3.3.11 Mia-Quality (Mia-Software)

Mia-Quality[39] wurde durch Mia-Software entwickelt und ist Teil der Suite Mia-Insight. Mia-Quality kann Softwaresysteme in COBOL und Natural analysieren, andere Module der Insight-Suite unterstützen auch weitere Programmiersprachen.

Mia-Quality dient dabei der Bewertung der Codequalität und der Kontrolle der Einhaltung von definierbaren Richtlinien. Das Modul Mia-Discovery aus der Insight-Suite kann Komponentenstrukuren, -beziehungen und Ressourcenbenutzung innerhalb des analysierten Softwaresystems aufdecken. Das dritte Modul Mia-Mining hat das Ziel der Analyse von Kontroll- und Datenfluss und der Suche nach unerreichbarem Code.

Mia-Quality kann die Analyseergebnisse in tabellarischer wie auch in graphischer Form ausgeben, die Bedienung erfolgt mittels einer GUI.

Das Tool kann die folgenden Metriken berechnen:

- Halstead-Metriken
- McCabe-Metriken
- McClure Complexity
- Knot-Count (Number of GO TO crossovers)
- Number of control variables
- Number of complex module
- Number of paragraph activations (others than done by PERFORM statements)
- Number of hierarchical levels
- Number of root segments
- Number of illicit GO TO statements
- Number of comments lines
- Number of recursive paths
- Number of dead-code lines
- Number of arithmetic instructions
- Number of program terminaison instructions
- Number of files handling instructions
- Number of sort or merge instructions
- Number of COPY statements used by the program
- Number of PERFORM statements
- Number of MOVE statements
- Number of GOTO statements
- Number of CALL statements
- Number of IF statements
- Number of lines
- Number of lines for data

[39] `http://www.mia-software.com/en/products/mia-insight/mia-quality/`

- Number of lines for instructions
- Dead code density
- PERFORM statements density
- Connection statements density
- Average number of instructions per module
- Hierarchical complexity
- Average number of modules per level

Eine Möglichkeit zur Definition eigener Metriken besteht nicht.

Mia-Quality kann Quellcode in folgenden COBOL-Versionen und -Erweiterungen verarbeiten:

- COBOL-74
- COBOL-85
- Microfocus

Mia-Software verweist in den Kundenreferenzen[40] vor allem auf Unternehmen aus dem Finanzsektor (u.a. Banque Populaire, Crédit du Nord, ING Belgium, Société Générale), aus dem Versicherungswesen (u.a. AXA, CNP Assurances, MAAF, MMA), aus der Luft- und Raumfahrt (u.a. Airbus, EADS, Sukhoi, Thales) und aus dem Verkehrswesen (u.a. Air France, General Motors, SNCF) sowie einige Behörden (u.a. Europäisches Parlament, französisches Verteidigungs-, Verkehrs- und Justizministerium).

Weitere Informationen zur Art der internen Arbeitsweise des Programms, Exportfunktionen, zur Verfügbarkeit einer API, zum aktuellen Versionsstand, Systemvoraussetzungen, zum Preis einer Linzenz und zur Möglichkeit einer Demoversion sind beim Hersteller angefordert, von diesem aber nicht zur Verfügung gestellt worden. Eine Übersicht der relevanten Kriterien findet sich in Tabelle 11.

3.3.12 Rational Asset Analyzer (IBM Corporation)

Der *Rational Asset Analyzer*[41] (RAA) von IBM ist ein Tool zur Analyse von Code bestehender Softwaresysteme. Dabei ist die Verarbeitung von Quellcode in COBOL, PL/1 und Java möglich.

Das Haupteinsatzgebiet der Software liegt dabei in der Dokumentation, dem Programmverstehen und der Änderungsverfolgung. Ziel ist eine Unterstützung der Entwickler bei Einarbeitung, Wartung und Wiederverwendung von Software. Das Hauptaugenmerk liegt dabei - wenn auch nicht ausschließlich - auf Software für Mainframesysteme. Als Teil der Analysefunktionen können auch Metriken über die Quellcodes berechnet werden. Neben dem üblichen Zugriff auf Code über das Dateisystem kann der RAA auch auf die Softwareverwaltungssysteme CVS, PVCS und IBM Rational ClearCase zugreifen.

[40] `http://www.mia-software.com/en/company/references/`
[41] `http://www.ibm.com/software/awdtools/raa/`

Tool	Mia-Quality
Hersteller	Mia-Software
Lizenz und Preis	Kommerziell - Preis unbekannt
Version und Datum	unbekannt
Referenzen	Finanzsektor, Versicherungen, Luft-/Raumfahrt, Verkehrswesen, Behörden
Demoversion verfügbar	unbekannt
Benutzerdefinierte Metriken	nein
API	unbekannt
Export / Import	unbekannt
Arbeitsweise	unbekannt
Systemvoraussetzungen	unbekannt
COBOL-Versionen	COBOL-74, -85; Microfocus
Bedienung	GUI
Verarbeitung ganzer Systeme	ja
Visualisierung	ja

Tabelle 11: Übersicht zu Mia-Quality

Die bei der Analyse generierten Daten werden durch den RAA in einem Repository in Form einer DB2-Datenbank gespeichert. Die Ergebnisreports können sowohl in textbasierter als auch in grafischer Form visualisiert ausgegeben werden.

Die Bedienung des RAA kann über eine reguläre, Eclipse-basierte GUI erfolgen. Des Weiteren ist ein Zugriff über eine webbasierte Oberfläche möglich. Darüber hinaus können die Funktionen über den bereitgestellten Webservice genutzt werden. Ein Zugriff auf die Daten im Repository ist über die SQL-Schnittstelle ebenfalls möglich.

IBM stellt zum Rational Asset Analyzer keine Kunden- oder Projektreferenzen zur Verfügung.

Rational Asset Analyzer liegt aktuell in Version 5.5.1 vom November 2009 (Stand: April 2010) vor. Die Software ist lauffähig auf Microsoft Windows ab Version 2000, zusätzlich wird Open Object Rexx[42] ab Version 3.1.2, IBM DB2 ab Version 8.1, sowie Java 1.6 benötigt. Eine Einzelbenutzerlizenz kostet 2.060 USD, der Hersteller bietet außerdem eine Demoversion an. Weitere beim Hersteller angefragte Informationen hinsichtlich der internen Arbeitsweise, dem Vorhandensein einer API, den unterstützten COBOL-Versionen, möglichen Exportfunktionen und einer Übersicht der unterstützten Metriken blieben unbeantwortet. Eine Übersicht der relevanten Kriterien findet sich in Tabelle 12.

3.3.13 Roadmap for COBOL / Engine for COBOL (RainCode)

Roadmap for COBOL[43] wurde durch RainCode entwickelt. Das Tool dient der Analyse und Dokumentation von Softwaresystemen. Roadmap setzt dabei auf der *Raincode Engine for COBOL* auf. Neben der COBOL-Version existieren auch Varianten von Roadmap für Ada, Informix 4GL

[42] `http://www.oorexx.org/`
[43] `http://www.raincode.com/cobolroadmap.html`

Tool	Rational Asset Analyzer (RAA)
Hersteller	IBM Corporation
Lizenz und Preis	Kommerziell - Preis 2.060 USD
Version und Datum	5.5.1 (November 2009)
Referenzen	unbekannt
Demoversion verfügbar	ja
Benutzerdefinierte Metriken	unbekannt
API	unbekannt
Export / Import	unbekannt
Arbeitsweise	unbekannt (wahrscheinlich baum- oder graphbasiert)
Systemvoraussetzungen	Windows ab Version 2000, Open Object Rexx, IBM DB2 ab Version 8.1, Java 1.6
COBOL-Versionen	unbekannt
Bedienung	GUI, Weboberfläche, Webservice
Verarbeitung ganzer Systeme	ja
Visualisierung	ja

Tabelle 12: Übersicht zu Rational Asset Analyzer

und PL/1. Von der Engine existiert zusätzlich zu den genannten Sprachen auch eine Version für C und C++.

Roadmap besitzt zum einen Funktionalitäten zur Berechnung von Metriken und zur Erzeugung entsprechender Metrikreports. Zum anderen sind Funktionalitäten zum Generieren von Quellcode mit Zusatzinformationen vorhanden. Dazu wird aus jeder Codedatei ein HTML-Äquivalent erzeugt. Darin werden die Zeilen nummeriert, der Quellcode wird farbig formatiert (Schlüsselwörter, Kommentare, eingebettete SQL-Anweisungen, etc.) und jede Variable, Funktion, etc. mit einem Link zur jeweiligen Definition belegt. Die Benutzung von weiteren Ressourcen im Code, wie etwa Dateien, Datenbanken auch Copybooks[44], wird ebenfalls mit einem Link belegt. Dieser verweist in diesem Fall auf zusätzliche Informationen über die jeweilige Ressource.

Alle von Roadmap erzeugten Metrikreports werden - wie auch die „angereicherten“ Codedateien - im HTML-Format ausgegeben. Dabei werden die berechneten Metriken in tabellarischer wie auch in graphischer Form erzeugt. Die Bedienung des Tools erfolgt über eine GUI.

Die Engine selbst dient vor allem der automatisierten Verarbeitung von Code. Haupteinsatzgebiet sind die Berechnung von Metriken, die Überprüfung von Richtlinien, die Anwendung von Updates/Patches und die Qualitätsprüfung. Diese Aktionen können zum Beispiel durch Build-Skripte oder Versionsverwaltungssysteme ausgelöst werden. Die Engine arbeitet dabei intern auf einer baumbasierten Repräsentation des Codes und erlaubt auch die Steuerung und Modifikation von Code mittels einer eigenen Skriptsprache. Die Bedienung der Engine erfolgt über die Konsole.

Roadmap kann die folgenden Metriken berechnen:

[44] Ein Copybook in COBOL kann in etwa mit einer Funktionsbibliothek in modernen Programmiersprachen verglichen werden.

- Number of lines
- Number of blank lines
- Number of code lines
- Number of comment lines
- Number of mixed code and comment lines
- Cyclomatic Complexity
- Maximum nesting level
- Number of statements
- Number of CICS statements
- Number of different CICS maps
- Number of different CICS queues
- Number of different CICS files
- Number of SQL statements
- Number of different SQL tables

Darüber hinaus ist die Möglichkeit zur Berechnung benutzerdefinierter Metriken gegeben.

Der Hersteller spricht von einer Unterstützung „beinahe aller COBOL-Dialekte und -Versionen“, führt jedoch keine genaue Liste auf.

RainCode stellt zu Roadmap und Engine for COBOL keine Kunden- oder Projektreferenzen zur Verfügung.

Roadmap und Engine for COBOL liegen aktuell jeweils in Version 1.83 vom Februar 2010 (Stand: Juni 2010) vor. Die Programme sind lauffähig unter Microsoft Windows ab Version 2000, Linux und Unix. Der Hersteller bietet zwar generierte Resultate beider Produkte als Download an, echte Demoversionen sind jedoch nicht erhältlich. Weitere Informationen zur Verfügbarkeit einer API, einer Demoversion und zum Preis einer Lizenz sind beim Hersteller angefordert, von diesem aber nicht zur Verfügung gestellt worden. Die Preise für die reine Engine liegen je nach unterstützter Programmiersprache zwischen 20.000 € und 26.500 € und sind somit zumindest ein Indiz für den Preisrahmen der COBOL-Variante. Eine Übersicht der relevanten Kriterien findet sich in Tabelle 13.

3.3.14 SmartDoc (Allen Systems Group, Inc.)

SmartDoc[45] ist Teil der *Existing Systems Workbench* (ESW) von Allen Systems Group, Inc.. Die ESW ist eine Softwaresammlung zum Reengineering und zur Wartung bestehender, in COBOL entwickelter Softwaresysteme.

SmartDoc dient der Erzeugung von Reports zur Dokumentation bestehender Software. Dazu können Diagramme und Berichte zur Programmstruktur, zum Kontrollfluss, zur Aufrufstruktur, zu Fehlerquellen und zu berechneten Metriken erzeugt werden. Dabei sind Metriken jedoch nur ein Teil- und nicht der Schwerpunkt. Die ESW beinhaltet darüber hinaus noch

[45] `http://www.asg.com/products/product_details.asp?code=DCX`

Tool	Roadmap for COBOL / Engine for COBOL
Hersteller	RainCode
Lizenz und Preis	Kommerziell - Preis unbekannt
Version und Datum	1.83 (Februar 2010)
Referenzen	unbekannt
Demoversion verfügbar	unbekannt
Benutzerdefinierte Metriken	ja
API	unbekannt
Export / Import	ja (HTML) / nein
Arbeitsweise	baumbasiert
Systemvoraussetzungen	Windows ab Version 2000, Linux, Unix
COBOL-Versionen	unbekannt (wahrscheinlich alle gängigen)
Bedienung	GUI, Konsole
Verarbeitung ganzer Systeme	ja
Visualisierung	ja

Tabelle 13: Übersicht zu Roadmap for COBOL

weitere Tools zur technischen Projektplanung, Qualitätsprüfung, zum Programmverständnis, Editieren, Testen und Debugging.

SmartDoc kann die Analysedaten in einem Repository speichern und hierüber auch auf Daten von anderen Tools der ESW zugreifen oder für diese verfügbar machen.

ASG verweist in den Kundenreferenzen[46] auf breites Spektrum von Unternehmen (u.a. Air France, American Express, Coca Cola, Daimler Chrysler, Deutsche Telekom, Toyota, Verizon).

Weitergehende Informationen hinsichtlich Vorhandensein einer API, unterstützter COBOL-Versionen, berechneter Metriken, Arbeitsweise, Exportfunktionen, Bedienungsweise, aktuellem Versionsstand, Systemvoraussetzungen, Preis einer Lizenz und Möglichkeit einer Demoversion sind beim Hersteller angefordert worden, es erfolgte jedoch keinerlei Reaktion. Eine Übersicht der relevanten Kriterien findet sich in Tabelle 14.

3.3.15 SofAudit (SES Software-Engineering Service GmbH)

Von der SES Software-Engineering Service GmbH stammt *SofAudit*[47]. Das Tool dient der Analyse von Softwaresystemen und kann neben COBOL auch Code in den Programmiersprachen Assembler, PL/1, APS, CSP, DELTA, ABAP, Natural, C, C++, C# und Java, sowie in diversen Datenbeschreibungssprachen (DLI, DDL, ADABAS, SQL, BMS, MFS, HTML, XML und IDL) und einigen Prozesskontrollsprachen (VMS-, MVS- und UNIX-JCL) analysieren.

Haupteinsatzgebiet von SofAudit ist die Berechnung von Metriken und Statistiken sowie die Überprüfung der Einhaltung bestimmter Richtlinien. Dabei kann eine Auswahl aus Richtlinien und Metriken getroffen werden. Zusätzlich können Gewichtungen für jede Metrik und

[46] `http://www.asg.com/about_us/clients.asp`
[47] `http://www.soring.hu/tools.html#SofAudit`

Tool	SmartDoc
Hersteller	Allen Systems Group, Inc.
Lizenz und Preis	Kommerziell - Preis unbekannt
Version und Datum	unbekannt
Referenzen	Sonstige (haupts. Großunternehmen)
Demoversion verfügbar	unbekannt
Benutzerdefinierte Metriken	unbekannt
API	unbekannt
Export / Import	unbekannt
Arbeitsweise	unbekannt
Systemvoraussetzungen	unbekannt
COBOL-Versionen	unbekannt
Bedienung	unbekannt
Verarbeitung ganzer Systeme	ja
Visualisierung	unbekannt

Tabelle 14: Übersicht zu SmartDoc

Schwellwerte für einige der Richtlinien festgelegt werden. Das Tool erzeugt pro analysierte Quelldatei jeweils einen Richtlinien- und Metrikreport, darüber hinaus einen Metrikreport über alle Dateien.

Die Ergebnisse der Richtlinien- und Metrikanalyse können im Textformat exportiert werden, der Gesamtmetrikreport darüber hinaus auch im XML-Format. Eine Visualisierung der Ergebnisse ist nicht möglich.

Die Bedienung des Tools erfolgt über eine GUI, welche zur Analyse aber sprachspezifische Tools aufruft. Bei diesen handelt es sich um Konsolenprogramme, die auch ohne die zentrale GUI einzeln aufgerufen werden können.

SofAudit kann die folgenden Metriken berechnen:

- Data Complexity (Chapin Metric)
- Data Flow Complexity (Elshof Metric)
- Data Access Complexity (Card Metric)
- Interface Complexity (Henry Metric)
- Control Flow Complexity (McCabe Metric)
- Decisional Complexity (McClure Metric)
- Branching Complexity (Sneed Metric)
- Language Complexity (Halstead Metric)
- Average Program Complexity
- Degree Of Modularity
- Degree Of Portability
- Degree Of Testability
- Degree Of Reusability
- Degree Of Convertibility

- Degree Of Flexibility
- Degree Of Conformity
- Degree Of Maintainability
- Average Program Quality

Darüber hinaus können 61 weitere Metriken berechnet werden. Bei diesen handelt es sich um Größenmaße, also etwa um die Zählung bestimmter Codeelemente und Ähnliches. Eine Möglichkeit zur Definition eigener Metriken ist nicht vorhanden, der Hersteller bietet aber die Möglichkeit einer Implementierung auf Anfrage an. Eine API ist nicht zugänglich.

SofAudit kann Quellcode in folgenden COBOL-Versionen verarbeiten:

- COBOL-74
- COBOL-85
- COBOL-96

SES verweist in den Kundenreferenzen[48] auf ein breites Spektrum von Unternehmen (u.a. ESPRIT, Wella, Siemens, Hamburg-Mannheimer Versicherungen, Alldata, Lufthansa).

SofAudit liegt aktuell in Version 5 vom Oktober 2009 (Stand: Juni 2010) vor. Das Programm ist lauffähig unter Microsoft Windows; da die Analysatoren allerdings 16-Bit-Programme sind, laufen diese unter 64-Bit-Betriebssystemen nicht mehr. Der Hersteller bietet keine Demoversion an, der Preis einer Lizenz ist nicht bekannt. Eine Übersicht der relevanten Kriterien findet sich in Tabelle 15.

Tool	SofAudit
Hersteller	SES Software-Engineering Service GmbH
Lizenz und Preis	Kommerziell - Preis unbekannt
Version und Datum	5 Rel. 1009 (Oktober 2009)
Referenzen	Sonstige
Demoversion verfügbar	nein
Benutzerdefinierte Metriken	nein (nur durch den Hersteller auf Anfrage)
API	nein
Export / Import	ja (Text, XML) / nein
Arbeitsweise	unbekannt
Systemvoraussetzungen	Windows
COBOL-Versionen	COBOL-74, -85, -96
Bedienung	GUI, Konsole
Verarbeitung ganzer Systeme	ja
Visualisierung	nein

Tabelle 15: Übersicht zu SofAudit

[48] `http://www.soring.hu/references.html`

3.3.16 Xpediter (Compuware Corporation)

In der Reihe *Xpediter*[49] führt die Compuware Corporation eine Reihe von Tools zur Unterstützung der Entwicklung und Wartung von Softwaresystemen. Die Analyse von Quellcode in COBOL, PL/1, C und Assembler ist möglich.

Der *Program Analyzer* dient dabei der Analyse der Programm- und Datenstruktur. *Xpediter/-DevEnterprise* ist zur Nachverfolgung von Änderungen und zur Qualtitätssicherung gedacht. Dazu verwenden beide Tools unter anderem über den Quellcode berechnete Metriken. Weitere Programme aus der Xpediter-Suite dienen dem Testen, Debugging und der Laufzeitanalyse.

Compuware verweist in den Kundenreferenzen[50] vor allem auf Unternehmen aus der Informationstechnik (u.a. Sun, Software Innovation Finland, Raindrop Information Systems, E.ON IS), alle weiteren Unternehmen bilden ein recht breites Spektrum ab (u.a. Austrian Airlines, China Telecom, Detroit Medical Center, Grand Valley State University, New Hanover Health Network, RHB Bank, West Bend Mutual Insurance Company).

Weitergehende Informationen hinsichtlich Vorhandensein einer API, unterstützter COBOL-Versionen, berechneter Metriken, Arbeitsweise, Exportfunktionen, Bedienungsweise, aktuellem Versionsstand, Systemvoraussetzungen, Preis einer Lizenz und Möglichkeit einer Demoversion wurden beim Hersteller angefordert. Die Antworten wurden jedoch mit Verweis auf eine dort herrschende hohe Arbeitsauslastung verweigert. Eine Übersicht der relevanten Kriterien findet sich in Tabelle 16.

Tool	Xpediter
Hersteller	Compuware Corporation
Lizenz und Preis	Kommerziell - Preis unbekannt
Version und Datum	unbekannt
Referenzen	IT, sonstige
Demoversion verfügbar	unbekannt
Benutzerdefinierte Metriken	unbekannt
API	unbekannt
Export / Import	unbekannt
Arbeitsweise	unbekannt
Systemvoraussetzungen	unbekannt
COBOL-Versionen	unbekannt
Bedienung	unbekannt
Verarbeitung ganzer Systeme	unbekannt
Visualisierung	unbekannt

Tabelle 16: Übersicht zu Xpediter

[49] http://www.compuware.com/solutions/xpediter.asp
[50] http://www.compuware.com/d/case_studies.asp

3.3.17 Yet Another Source Code Analyzer (Michael V. Scovetta)

Yet Another Source Code Analyzer[51] (YASCA) wurde durch Michael V. Scovetta entwickelt. Das Programm ist grundsätzlich für die Quellcode-Analyse gedacht und unterstützt laut Entwickler unter anderem ASP, C, C++, COBOL, Coldfusion, HTML, Java, JavaScript, .NET-Sprachen und PHP.

YASCA selbst dient als zentrales Steuerprogramm, die eigentlichen Analysefunktionen werden über Plug-Ins realisiert. Bei den Plug-Ins handelt es sich dabei um PHP-, GREP- oder PMD[52]-Skripte. Die PMD-basierten Plug-Ins sind dabei, wie PMD selbst, rein auf die Analyse von Javacode beschränkt. Bei allen anderen mitgelieferten Plug-Ins gibt es zum einen solche, die weitere Analyseprogramme ausführen und lediglich kleinere Berechnungen oder Reformatierungen auf die dabei generierten Ausgaben anwenden. Die restlichen Plug-Ins sind eher einfacher Art und arbeiten meist mittels GREP direkt auf den zu analysierenden Quellcodes. Das einzige COBOL-spezifische Plug-In fällt unter die letztere Kategorie und überprüft auf Vorhandensein von GETMAIN- ohne zugehörige FREEMAIN-Aufrufe (was potentiell Speicherlecks verursachen könnte).

Die derzeit unterstützten Analyseprogramme sind:

- FindBugs
- PMD
- JLint
- JavaScript Lint
- PHPLint
- CppCheck
- ClamAV
- RATS
- Pixy

Unter den genannten Programmen kann keines COBOL-Code verarbeiten. Die Ausgabeformate hängen vom verwendeten Plug-In und Analyseprogramm ab, meist handelt es sich jedoch um Text- oder XML-Formate. Die Bedienung von YASCA erfolgt ausschließlich mittels Kommandozeilenparametern beim Aufruf. Keines der Plug-Ins kann Metriken berechnen. Dies zu Implementieren wäre zwar aufgrund der offenen Struktur durchaus möglich, die Möglichkeiten dabei sind aber aufgrund der derzeit vorhandenen Skripte und Programme eher eingeschränkt.

Michael V. Scovetta stellt zu YASCA keine Kunden- oder Projektreferenzen zur Verfügung.

YASCA liegt aktuell in Version 2.1 vom September 2009 (Stand: April 2010) vor, läuft auf Microsoft Windows ab Version 2000 und benötigt für PMD-Plug-Ins Java ab Version 1.4. Das

[51] `http://www.scovetta.com/yasca.html`
[52] `http://pmd.sourceforge.net/`

Programm ist kostenlos und quelloffen unter der BSD License verfügbar. Eine Übersicht der relevanten Kriterien findet sich in Tabelle 17.

Tool	Yet Another Source Code Analyzer (YASCA)
Hersteller	Michael V. Scovetta
Lizenz und Preis	BSD - kostenlos
Version und Datum	2.1 (September 2009)
Referenzen	unbekannt
Demoversion verfügbar	nein (da Vollversion frei verfügbar)
Benutzerdefinierte Metriken	ja (siehe Text)
API	ja
Export / Import	ja (Text, XML) / nein
Arbeitsweise	unterschiedlich (je nach Plug-In)
Systemvoraussetzungen	Windows, Java
COBOL-Versionen	unbekannt
Bedienung	Konsole
Verarbeitung ganzer Systeme	unterschiedlich (je nach Plug-In)
Visualisierung	nein

Tabelle 17: Übersicht zu YASCA

3.4 Übersicht

Tabelle 18 zeigt eine Übersicht der recherchierten Tools und listet die im Abschnitt 3.1 genannten, entscheidungsrelevanten Eigenschaften auf. Einträge mit einem Fragezeichen bedeuten, dass diese Eigenschaft nicht in Erfahrung gebracht werden konnte.

3.5 Bewertung und Auswahl der Tools

An dieser Stelle erfolgt eine Bewertung und vorläufige Auswahl aus den bisher recherchierten Tools anhand der Kriterien aus Abschnitt 3.2.

Das erste relevante Kriterium ist der Preis für eine Lizenz. Grundsätzlich besteht zwar die Möglichkeit des Erwerbs einer Software im Rahmen von Forschung und Lehre, die Preise der hier genannten Tools liegen dafür jedoch zu hoch. Eine zeitlich und funktional nicht zu weit eingeschränkte Demoversion wäre ebenfalls geeignet, wurde aber von keinem der Hersteller angeboten.

Eine Ausnahme bei diesem Kriterium bildet *SofAudit*, da diese Software der Universität Koblenz für Forschung und Lehre als Vollversion zur freien Verfügung steht. Die zweite Ausnahme bildet der *COBOL Flow Graph Manipulator*, da diese Software vom Hersteller pro et con der Debeka als Vollversion bereitgestellt wurde und somit im Rahmen des Forschungsprojekts Cobus für diese Diplomarbeit ebenfalls zur Verfügung steht.

Daneben bleiben noch die lizenzkostenfreien Tools, so dass sich die erste Auswahl auf folgende Software beschränkt:

Tool	Lizenzmodell, Preis)	Demoversion	Aktives Projekt	Anzahl Metriken	Benutzerdef. Metriken	Export/Import	Visualisierung
AREDIS / Application Miner	kommerziell, ab 10.000 €	nein	ja	50	ja	ja (XML, XML relational) / nein	ja
AUDITOR	kommerziell, ab 64.000 €	ja	ja	830	ja	ja (XML, HTML) / nein	nein
COBOL Flow Graph Manipulator	kommerziell, ab 9.750 €[4]	ja	ja	10	ja	ja (CSV, HTML, BMP, VCG) / nein	ja
Continous Quality Assessment Toolkit	Open Source, kostenlos	-[1]	ja	7	ja	ja (XML, HTML) / nein	ja
DMS COBOL Source Code Metrics	kommerziell, ?	ja	ja	12	nein	ja (XML, Text) / nein	nein
Eclipse Metrics Plug-In	Open Source, kostenlos	-[1]	fraglich	24	nein	ja (XML) / nein	ja
Generische Umgebung zum Programmverstehen	Open Source, kostenlos	-[1]	ja	0	ja	ja / ja	nein
Interstage Software Quality Analyzer	kommerziell, ?	ja[3]	?[2]	11	ja	ja / nein	ja
IQ Developers Edition	kommerziell, ?	?	?[2]	42	?	ja (XML) / ?	ja
JDepend	Open Source, kostenlos	-[1]	fraglich	7	nein	ja (Text, XML) / nein	nein
Mia-Quality	kommerziell, ?	?	?[2]	45	nein	? / ?	ja
Rational Asset Analyzer	kommerziell, ab 2.060 $	ja	ja	?	?	? / ?	ja
Roadmap / Engine for COBOL	kommerziell, ?[5]	nein	ja	14	ja	ja (HTML) / nein	ja
SmartDoc	kommerziell, ?	?	?[2]	?	?	? / ?	?
SofAudit	kommerziell, ?[4]	nein	ja	79	nein	ja (Text, XML) / nein	nein
Xpediter	kommerziell, ?	?	?[2]	?	?	? / ?	?
Yet Another Source Code Analyzer	Open Source, kostenlos	-[1]	ja	0	ja	ja (Text, XML) / nein	nein

Tabelle 18: Übersicht der recherchierten Tools

[1] Eine Demoversion ist nicht verfügbar, aufgrund der kostenlosen Vollversion aber auch irrelevant
[2] Aktivität ist sehr wahrscheinlich
[3] Die Demoversion ist ebenfalls kostenpflichtig
[4] Die Vollversion steht dem Institut für Softwaretechnik bereits kostenlos zur Verfügung
[5] Der Preis liegt vermutlich zwischen 20.000 € und 26.500 € (siehe Text zum Tool)

- COBOL Flow Graph Manipulator
- Continous Quality Assessment Toolkit
- Eclipse Metrics Plug-In
- Generische Umgebung zum Programmverstehen
- JDepend
- SofAudit
- Yet Another Source Code Analyzer

Aus dieser Auswahl fallen - aufgrund der mangelnden Fähigkeit COBOL-Code zu verarbeiten - das *Eclipse Metrics Plug-In*, die *Generische Umgebung zum Programmverstehen* und *JDepend* heraus. Grundsätzlich wäre eine Implementierung der hierzu notwendigen Funktionalität möglich, der geschätzte Aufwand hierfür liegt allerdings für den Rahmen dieser Arbeit zu hoch.

GUPRO / (J)GraLab wurde am Institut für Softwaretechnik entwickelt, somit ist sehr viel Erfahrung mit diesem Tool vorhanden und ein problemloser Zugang zu jeglicher Art von Quellcode und Dokumentation gewährleistet. Da sich darüber hinaus derzeit bei der Debeka ein auf ANTLR[53] und JGraLab basierender Extraktor für COBOL in der Entwicklung befindet, könnte dieses Tool zu einem späteren Zeitpunkt möglicherweise wieder mit in die Auswahl aufgenommen werden.

Yet Another Source Code Analyzer fällt aufgrund der sehr geringen Eignung für COBOL-Code und Metriken ebenfalls aus der Auswahl.

Die Auswahl besteht nun lediglich noch aus den drei Tools *COBOL Flow Graph Manipulator*, *Continous Quality Assessment Toolkit* und *SofAudit*. Hiermit ist nun keine weitere Eingrenzung mittels der übrigen Kriterien mehr notwendig, so dass die Auswahl vorläufig feststeht. Diese drei Tools werden im folgenden Kapitel detailliert untersucht.

[53] `http://www.antlr.org/`

4 Analyse der ausgewählten Metrik-Tools

Nachdem am Ende des vorangegangenen Kapitels eine engere Auswahl aus den recherchierten Tools getroffen wurde, werden die darin verbliebenen Metrik-Tools SofAudit, FGM und ConQAT in diesem Kapitel näher untersucht. Dazu wird zunächst auf jedes der Tools hinsichtlich Installation, Ausführung, Bedienung, Analysevorgang, Analyseergebnissen, benutzerdefinierte Metriken und Ausführung ohne Benutzerinteraktion eingegangen. Anschließend wird eine Menge existierenden COBOL-Codes durch die Programme analysiert und die berechneten Ergebnisse ausgewertet und verglichen.

4.1 SofAudit

SofAudit von der SES Software-Engineering Service GmbH wurde bereits in Abschnitt 3.3.15 vorgestellt. Für die detaillierte Analyse in diesem Kapitel kommt SofAudit in Version 5 (Rel. 1009) vom Oktober 2009 zum Einsatz.

4.1.1 Installation und Ausführung

Eine Installationsroutine besitzt diese Software nicht, die Archivdatei mit allen zum Programm gehörigen Dateien kann an einem beliebigen Pfad auf einem der lokalen Laufwerke entpackt werden. Anschließend ist ein Start des Hauptprogramms ohne weitere vorbereitende Aktionen möglich. Das Hauptprogramm besitzt eine graphische Benutzeroberfläche (graphical user interface - GUI), welche in Abbildung 1 gezeigt ist.

Ein Ausführen des Hauptprogramms ist auch von einem Netzwerkpfad möglich. Dies führt jedoch dazu, dass die während der Analyse im Hintergrund aufgerufenen Scannerprogramme nicht starten können. Da dieses Problem auch dann auftritt, wenn der Netzwerkpfad über einen Laufwerksbuchstaben lokal eingebunden wird, dürfte dies in den unter Windows für Netzlaufwerke geltenden Sicherheitsbeschränkungen begründet sein.

Ein Start des Hauptprogramms ist auch unter den 64Bit-Varianten von Windows[54] möglich. Jedoch können auch hier die Scanner nicht ausgeführt werden. Dies liegt darin begründet, dass es sich dabei - im Gegensatz zum 32Bit-Hauptprogramm - noch um 16Bit-Programme handelt. Betriebssysteme mit 64Bit können zwar 32Bit-Programme ausführen, jedoch keine 16Bit-Programme. Die Ausführung Letzterer ist bei 32Bit-Betriebssystemen noch möglich. Diese Software kann also ausschließlich auf den 32Bit-Versionen von Windows betrieben werden.

[54] Diese Varianten existieren von den Windows-Versionen XP, Server 2003, Vista, Server 2008, 7 und Server 2008 R2.

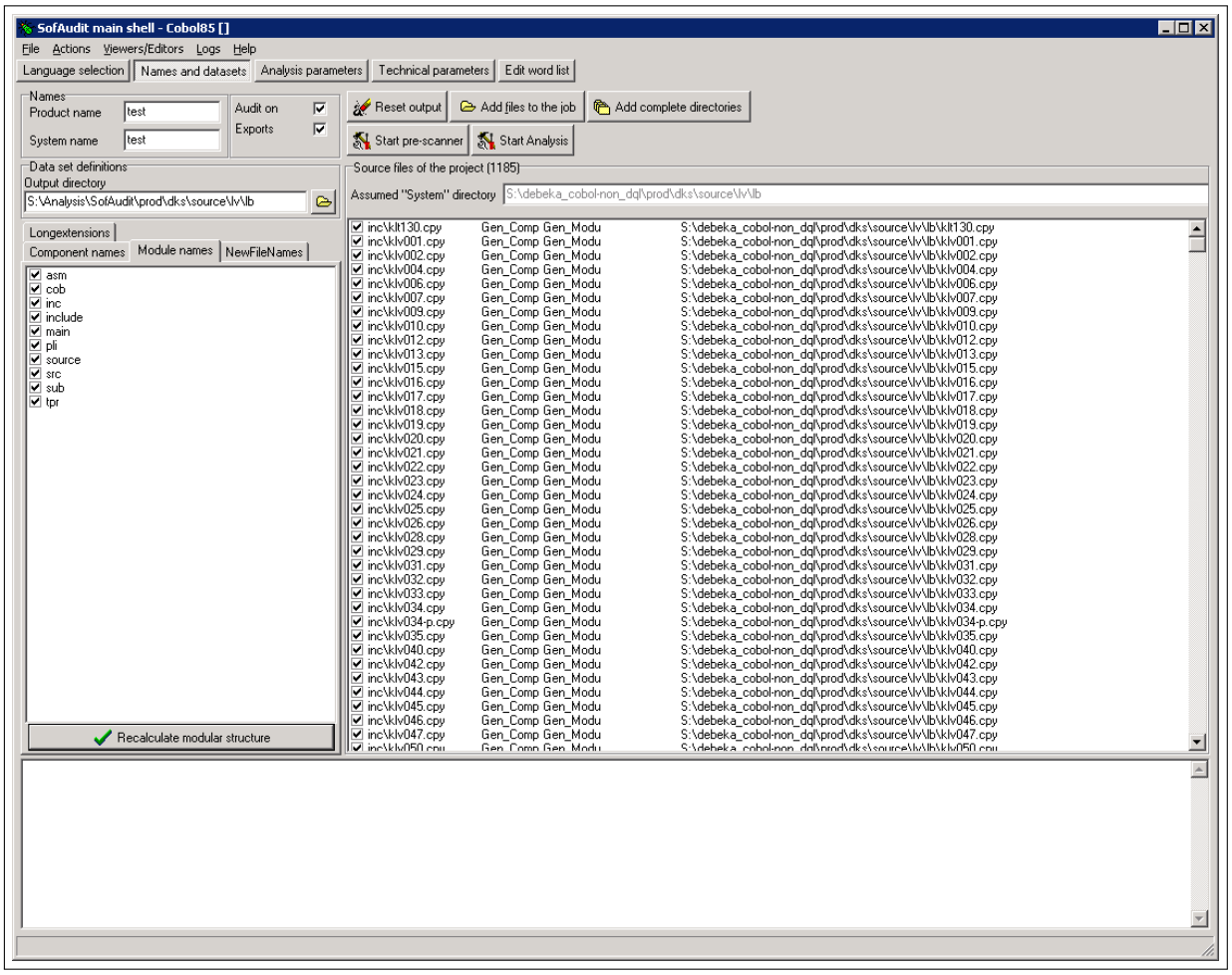

Abbildung 1: Die graphische Benutzeroberfläche (GUI) von SofAudit.

4.1.2 Bedienung

Nach dem Start der GUI wird typischerweise ein neues Projekt angelegt[55]. Dabei muss als erstes die zu verarbeitende Programmiersprache und ein Ausgabeverzeichnis gewählt werden. Optional kann eine Auswahl aus einer Liste von Metriken getroffen werden, standardmäßig sind alle verfügbaren ausgewählt. Zusätzlich kann auch eine Auswahl aus einer Liste von zu prüfenden Programmierrichtlinien ausgewählt werden, hierbei können für einige der Richtlinien auch Schwellwerte konfiguriert werden.

Die zu analysierenden Codedateien können einzeln oder verzeichnisweise hinzugefügt werden. Dabei gilt allerdings eine feste Obergrenze von maximal 9999 Dateien. Copybooks können ebenfalls analysiert werden, werden jedoch genauso wie normale Quellcodedateien verarbeitet. Ein Auflösen einer Copy-Referenz aus einer Codedatei und Berechnung der Metriken inklusive der entsprechenden Copystrecken geschieht demzufolge nicht. Dies wird in der Do-

[55] Ein Öffnen eines bestehenden Projektes ist ebenfalls möglich, würde an dieser Stelle aber nicht alle Schritte des Arbeitsablaufs beschreiben.

kumentation wie folgt begründet: *„Im Fall eines prozeduralen Programms in [...] COBOL [...] gibt es auch Copy- [...] Bibliotheken. Diese Bibliotheken sind zusammen mit dem System, zu dem sie gehören, zu verarbeiten. Sie sollten sich auf derselben Ebene wie die Komponenten befinden. SoftAudit berücksichtigt diesen Code nur einmal, so dass die Code-Messung nicht verfälscht wird.“* (siehe [Sneed2008-2], S. 21).

4.1.3 Analysevorgang

Beim Start des Analysevorgangs kopiert das Hauptprogramm zunächst die Quellcodedateien verzeichnisweise in jeweilige Unterordner des Ausgabeverzeichnisses, wobei die Dateien und Verzeichnisse umbenannt werden. Das Namensschema besteht dabei aus dem Präfix *MOD* (für Verzeichnisse) bzw. *SRC* (für Dateien), gefolgt von einer für den jeweiligen Typ fortlaufenden Nummer. Darüber hinaus wird eine Konfigurationsdatei *last.xml* im Ausgabeverzeichnis generiert, welche Referenzen auf die zu analysierenden Dateien (jeweils im Original und in der Kopie im Ausgabeverzeichnis) auflistet und auch alle weiteren Konfigurationswerte beinhaltet. Dann startet das Hauptprogramm den eigentlichen, sprachspezifischen Analysator und gibt diesem per Aufrufparameter die Konfigurationsdatei mit.

Das Analyseprogramm speichert die Ergebnisse für die Metriken und die Richtlinienprüfung in einem jeweils eigenen Verzeichnis im Ausgabeordner (*Metrics* und *Defis*) ab. Dabei ist die darin jeweils enthaltene Verzeichnisstruktur eine Kopie der Struktur des Eingabeverzeichnisses. Die einzelnen .MET- und .DEF-Dateien - mit den Metrik- beziehungsweise Richtlinenwerten - befinden sich darin im entsprechenden Pfad wie die jeweils zugehörigen Quellcodedateien im Eingabeverzeichnis. Eine Ausnahme bilden die zu Copybooks gehörigen .MET- und .DEF-Dateien, diese werden jeweils in einem Ordner *Gen_Modu* abgelegt. Zusätzlich wird - nur für die Metriken - auch noch eine .MET-Datei und eine XML-Datei mit den berechneten Werten über das gesamte Projekt generiert. Mit Ausnahme dieser einen genannten XML-Datei handelt es sich bei allen anderen Ergebnisdateien um formatierte Textdateien, Listing 11 in Anhang A zeigt eine .MET-Datei.

Nach dem Ende der Analyse werden die kopierten Codedateien und einige weitere Verzeichnisse mit temporären Dateien im Ausgabeordner gelöscht, so dass dieser anschließend im Wesentlichen nur noch die zuvor genannten Dateien mit den Ergebnissen enthält. Dies geschieht jedoch nicht mehr durch den Analysator, sondern wird wiederum vom Hauptprogramm durchgeführt. Abbildung 2 zeigt ein Beispiel der Verzeichnisstruktur des Ausgabeverzeichnisses.

4.1.4 Analyseergebnisse

Die Ergebnisse können anschließend in der GUI angezeigt werden. Dabei findet jedoch keine Visualisierung und keine weitere Formatierung statt, es werden lediglich die generierten Textdateien in einem integrierten Viewer dargestellt.

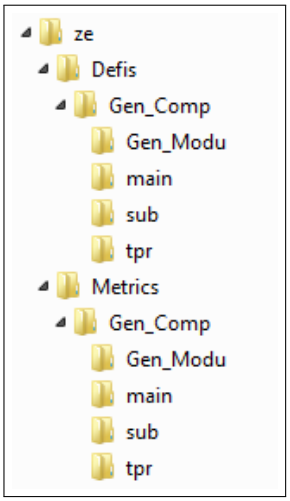

Abbildung 2: Struktur eines Ausgabeverzeichnisses von SofAudit.

4.1.5 Benutzerdefinierte Metriken

SofAudit bietet die Möglichkeit, eine Gewichtung für die einzelnen Metriken zu definieren. Darüber hinaus können Schwellenwerte für die Prüfung der Programmierrichtlinien festgelegt werden. Die Metriken und deren Berechnungsalgorithmen sind fest in die Software einprogrammiert. Eine Möglichkeit zur Definition eigener Metriken ist nicht vorhanden.

4.1.6 Ausführen ohne Benutzerinteraktion

Die sprachspezifischen Analyseprogramme beziehen alle benötigten Informationen aus einer Konfigurationsdatei, sind jeweils für sich allein vollständig lauffähig und legen die berechneten Ergebnisse in Dateien ab. Mittels Generieren der XML-basierten Konfigurationsdatei und Aufruf des Analyseprogramms ist somit ein automatisches Ausführen ohne Benutzerinteraktion möglich. Mittels Parsen der Ergebnisdateien können anschließend die berechneten Werte ausgelesen und weiterverarbeitet werden.

4.2 Flow Graph Manipulator

Der COBOL Flow Graph Manipulator (FGM) von der pro et con Innovative Informatikanwendungen GmbH wurde bereits in Abschnitt 3.3.3 vorgestellt. Für die detaillierte Analyse in diesem Kapitel kommt Flow Graph Manipulator (FGM) in Version 3.0 vom Juni 2010 zum Einsatz.

4.2.1 Installation und Ausführung

Die Software wird mittels einer Installationsroutine installiert. Dabei fällt auf, dass die sonst übliche Wahl des Programmpfades hier nicht möglich ist. Nach Abschluss der Installation ist

ein Start über die angelegte Programmverknüpfung möglich. Abbildung 3 zeigt die GUI des Programms.

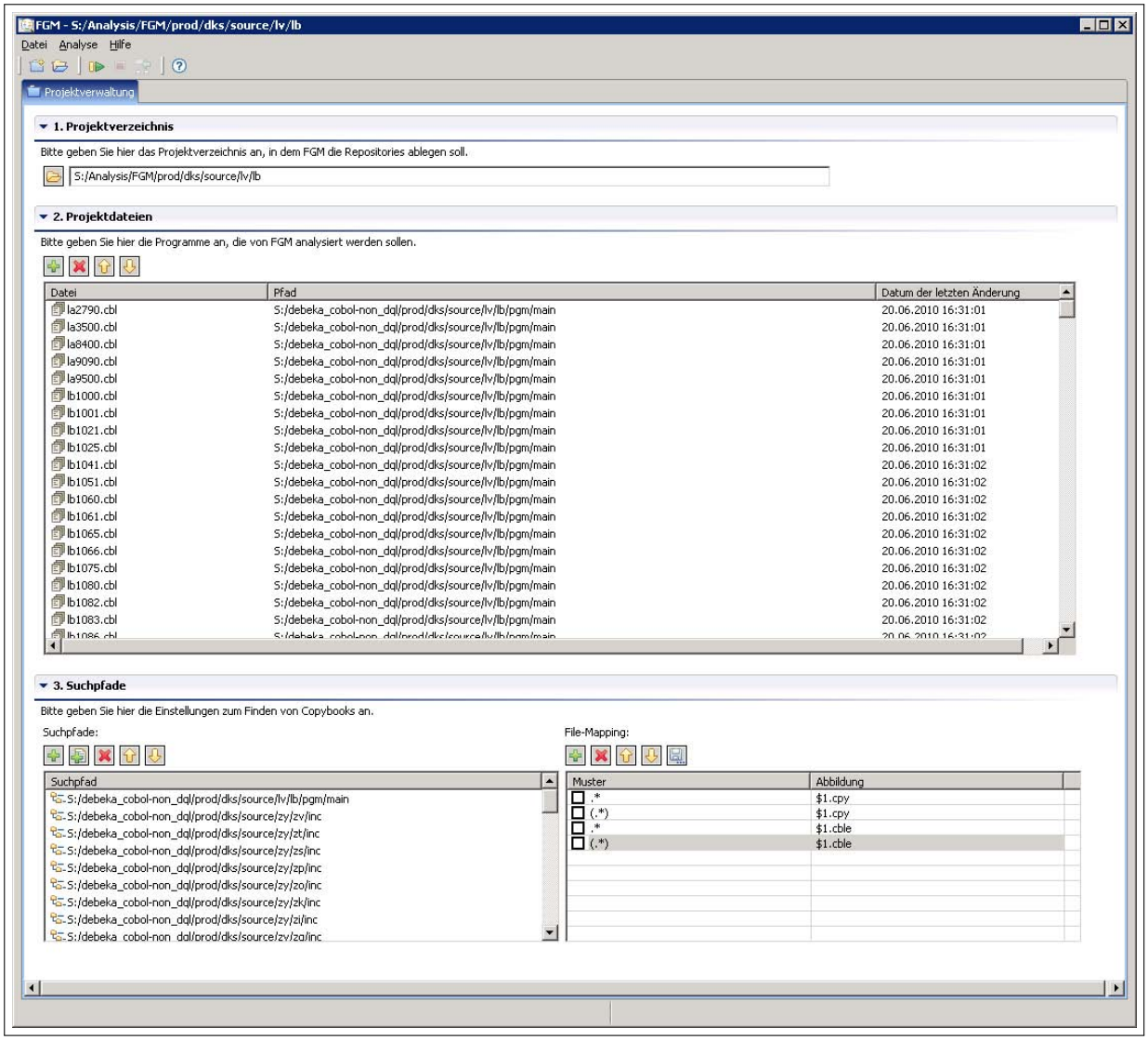

Abbildung 3: Die graphische Benutzeroberfläche (GUI) von Flow Graph Manipulator.

Die dabei aufgerufene .exe-Datei ist jedoch lediglich ein kleiner Starter für das Hauptprogramm mit seiner auf Eclipse basierten Oberfläche und zwei Hintergrundprogramme, den FGM-Server und einen MySQL-Server. Falls die für Eclipse notwendige Java-Laufzeitumgebung auf dem ausführenden Rechner nicht installiert sein sollte, so zeigt der Starter allerdings keine Fehlermeldung, sondern bleibt für ein bis zwei Minuten beim angezeigten Startlogo hängen. Nach dieser Zeit blendet zwar das Logo aus, der eigentliche Prozess des Starters terminiert jedoch nicht.

Auch beim Ausführen des Starters unter einem 64Bit-Betriebssystem zeigt sich dieses Verhalten - selbst wenn eine Java-Laufzeitumgebung vorhanden ist. Mithilfe eines eher zufällig entdeck-

ten Eintrages aus einer Logdatei wurde ersichtlich, dass der Starter von einem falschen Programmverzeichnis ausgeht und dementsprechend die beiden Hintergrundprogramme nicht starten kann[56]. Startet man zu diesem Zeitpunkt die beiden Hintergrundprogramme manuell vom korrekten Pfad, so gelangt man schließlich doch auf die eigentliche Benutzeroberfläche. Eine von hier gestartete Analyse schlägt jedoch fehl - mit einem Hängen des Programms und wiederum ohne eine Fehlermeldung. Der Grund hierfür dürfte ebenfalls die falsche Pfadannahme beim Starten des eigentlichen Analysatorprogramms sein. Diese Software sollte also ausschließlich auf den 32Bit-Versionen von Windows betrieben werden.

4.2.2 Bedienung

Nach dem Start der GUI werden Projekte als jeweils eigener Eclipse-Workspace angelegt oder geladen. Der Workspace-Pfad entspricht dabei auch dem Ausgabeverzeichnis, in dem die Analyseergebnisse gespeichert werden. Neben dem Hinzufügen von Codedateien (ein Hinzufügen von Verzeichnissen ist nicht möglich) sind für jedes Projekt nur zwei weitere Optionen einstellbar. Zum einen können Suchpfade angegeben werden, in welchen FGM nach Copybooks sucht - das Tool löst diese auf und bindet sie bei der Analyse in die jeweiligen Codedateien ein. Zum anderen können sogenannte *File-Mappings* festgelegt werden. Diese sind notwendig, da Copybooks in COBOL-Code nur mit dem Dateinamen referenziert werden, die eigentliche Datei aber üblicherweise noch die Erweiterung *.cpy* besitzt. Mit den Mappings kann festgelegt werden, unter welchem Namensschema Copybooks gesucht werden müssen.

4.2.3 Analysevorgang

Zur eigentlichen Analyse startet auch FGM ein Hintergrundprogramm. Dieses berechnet die Metriken für jede einzelne Codedatei, jedoch nicht Metriken über das Gesamtprojekt. Dem Hintergrundprogramm wird dabei eine generierte Konfigurationsdatei *project.xml* per Aufrufparameter übergeben. Der Analysator generiert zu jeder Codedatei eine Logdatei mit eventuellen Fehlermeldungen, eine *.pap*-Datei mit nur maschinenlesbarem Inhalt sowie eine *.dfg*-Datei. Letztere ist in mehrere Abschnitte unterteilt und enthält neben einer Graphrepräsentation der analysierten Codedatei auch die Ergebnisse der Metrikberechnungen.

Auch bei der Analyse zeigte sich der FGM etwas anfällig für Fehler. So sorgte ein falsch eingestelltes File-Mapping für entsprechende Fehlermeldungen (nicht gefundene Copybook-Dateien) in den Analyseergebnissen. Nach der Korrektur der Fehlerursache und einer erneuten Analyse wurden die gleichen Meldungen jedoch immer noch angezeigt - auch nach Löschen aller generierten Dateien im Ausgabeverzeichnis. Möglicherweise ist die Ursache hierfür in der Spei-

[56] Beim FGM handelt es sich um eine 32Bit-Software. Der Starter versucht im beschriebenen Fall jedoch, den Pfad für 64Bit-Programme zu verwenden (*C:\Program Files*). Unter Windows-Systemen mit 32Bit wäre dieser Pfad korrekt, unter 64Bit-Systemen lautet der korrekte Pfad für 32Bit-Programme jedoch *C:\Program Files (x86)*.

cherung von Daten im MySQL-Server im Hintergrund zu suchen, da nach dem Anlegen eines neuen Projektes mit absolut identischen Einstellungen dieser Fehler nicht mehr aufrat.

4.2.4 Analyseergebnisse

Die Ergebnisse der Analyse können anschließend in tabellarischer und graphischer Form (Balkendiagramm) für jede Codedatei in der GUI angezeigt werden. Die Tabellen können von dort aus als CSV-Dateien exportiert werden. Abbildung 4 zeigt als Beispiel eine Darstellung mit Balkendiagrammen.

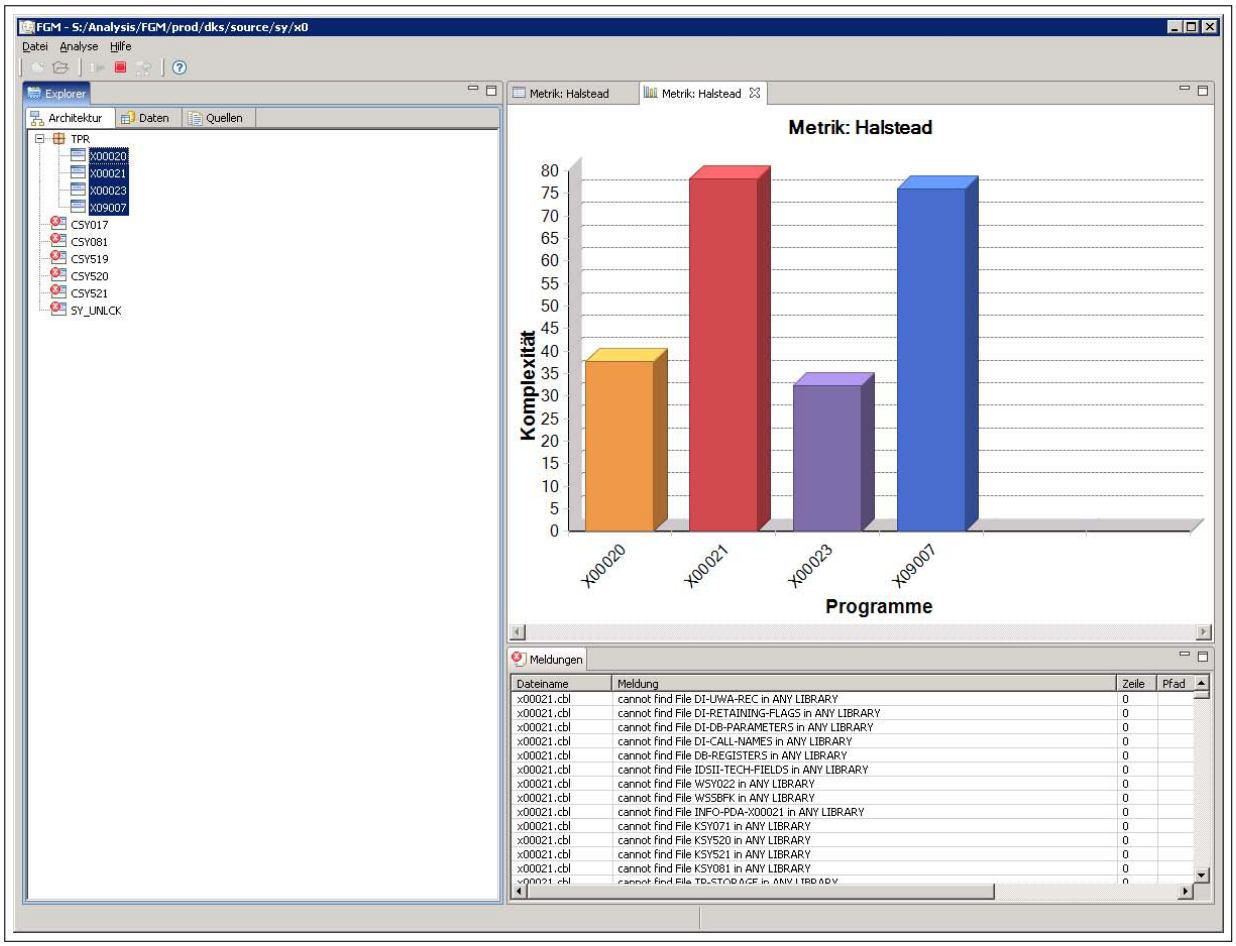

Abbildung 4: Graphische Darstellung von FGM-Analyseergebnissen als Balkendiagramm.

4.2.5 Benutzerdefinierte Metriken

Die Metriken und deren Berechnungsalgorithmen sind fest in die Software einprogrammiert. Zu den vorhandenen Metriken sind keinerlei Optionen oder einstellbare Parameter zugänglich. Eine Möglichkeit zur Definition eigener Metriken ist beim FGM nicht vorhanden.

4.2.6 Ausführen ohne Benutzerinteraktion

Zur Benutzung des von der GUI im Hintergrund verwendeten Analyseprogramms existiert keine Dokumentation und durch reines Ausprobieren war dieses auch nicht verwendbar. Nach Anfordern von weiterer Dokumentation über die Abfragesprache FGM-LAN und den Schnittstellen zu deren externen Verwendung wurde jedoch klar, dass in der aktuellen Version 3.0 des Programms diese Sprache nur noch intern im Programm verwendet wird und keinerlei Schnittstellen mehr offengelegt sind. Vor diesem Hintergrund ist eine Verwendung des Tools zum automatisierten Ausführen ohne Benutzerinteraktion nicht möglich.

4.3 ConQAT

Das Continous Quality Assessment Toolkit (ConQAT) von der Technischen Universität München wurde bereits in Abschnitt 3.3.4 vorgestellt. Für die detaillierte Analyse in diesem Kapitel kommt ConQAT in Version 2.7 vom August 2010 zum Einsatz.

4.3.1 Installation und Ausführung

Eine Installationsroutine besitzt diese Software nicht, die Archivdatei mit allen zum Programm gehörigen Dateien kann an einem beliebigen Pfad auf einem der lokalen Laufwerke entpackt werden. Bei diesem Paket handelt es sich um ein Eclipse 3.5, welches neben den Standardkomponenten auch bereits alle ConQAT-Bibliotheken beinhaltet. Alternativ hierzu lässt sich ConQAT auch in ein bereits vorhandenes Eclipse integrieren. Dies geschieht am einfachsten über die in Eclipse enthaltene Online-Installationsfunktion unter Verwendung des ConQAT-Installationsrepositories[57].

Wie bei allen Eclipse-basierten Anwendungen wird zum Ausführen zusätzlich eine installierte Java-Laufzeitumgebung benötigt. Ein Teil der Analysefunktionen benötigt zusätzlich auch noch das Program *Dot* aus dem GraphViz-Paket[58] oder das .NET Framework von Microsoft[59] - letzteres allerdings nur zur Analyse von C#-Code. Nach dem Start des Programms erfolgt die Bedienung über die GUI, welche in Abbildung 5 gezeigt ist.

Das Ausführen des Programms ist unter 32Bit- und 64Bit-Systemen möglich. Es fällt allerdings auf, dass bei diesem Paket im normalen Auslieferungszustand häufiger Fehlermeldungen mit NullPointer-Exceptions bei dem Versuch von Konfiguration und Analyse von Projekten auftreten. Dies geschieht sowohl bei neu erstellten Projekten als auch bei den beiliegenden Beispielprojekten. Abhilfe für dieses Verhalten schafft das Einschalten der automatischen Aktualisierung des Workspace (über die Menüpunkte *Window - Preferences - General - Workspace - Refresh automatically*).

[57] `http://www4.in.tum.de/~ccsm/conqat_update_site/`
[58] `http://www.graphviz.org/`
[59] `http://www.microsoft.com/net/`

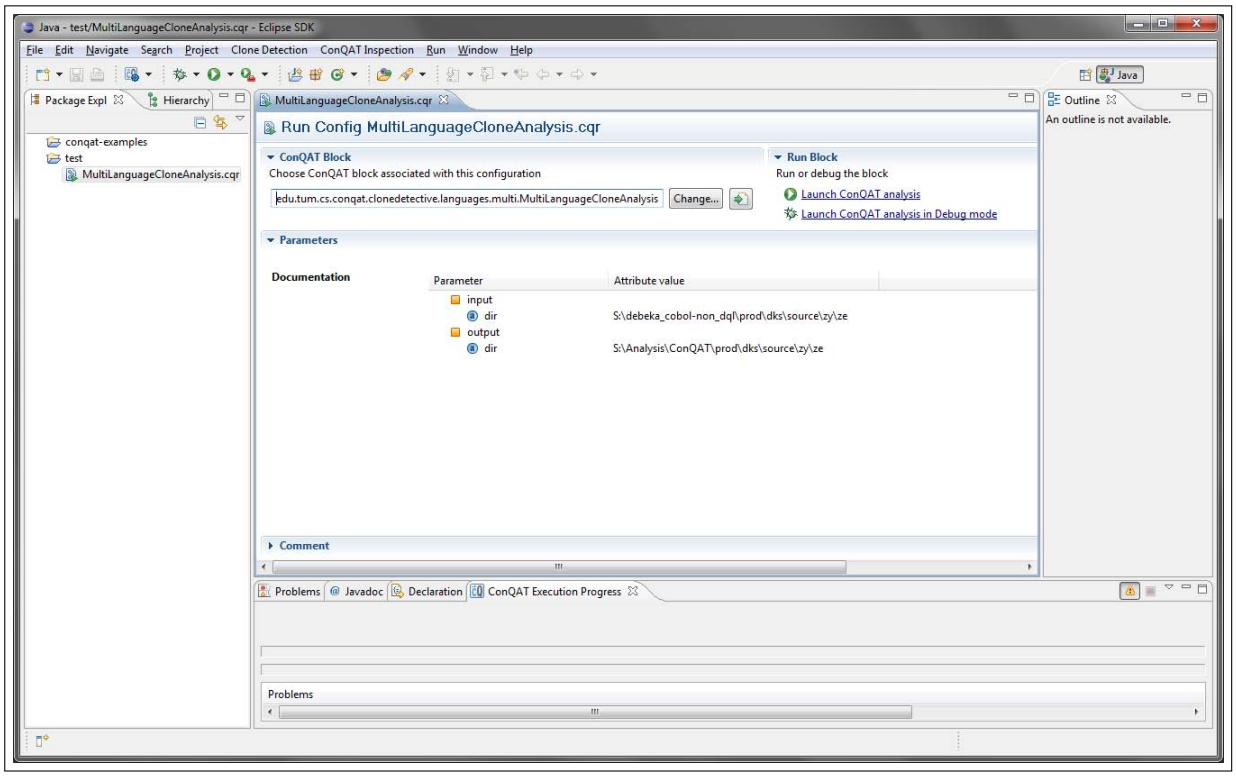

Abbildung 5: Die graphische Benutzeroberfläche (GUI) von ConQAT.

4.3.2 Bedienung

Nach dem Start der GUI kann Eclipse-typisch ein neuer Workspace angelegt oder ein bestehender weiterverwendet werden. Dabei fällt auf, dass einige der im offiziellen Handbuch (siehe [Deißenböck2010]) beschriebenen Beispielprojekte in der Software seit einigen Versionen nicht mehr unter dem angegebenen Namen vorhanden sind - offensichtlich wurde dieser Teil des Handbuchs nicht entsprechend aktualisiert.

Zum Ausführen einer Analyse muss zunächst eine sogenannte *Run Configuration* einem Projekt im Workspace hinzugefügt oder eine bereits bestehende geöffnet werden. Beim Anlegen einer Run Configuration muss genau ein *ConQAT Block* aus der Liste aller vorhandenen Blocks ausgewählt werden. Über die Auswahl des Blocks wird festgelegt, welche Analysefunktionalität zum Einsatz kommt. Die in der Liste angebotenen Blocks zeigen allerdings, dass keine sprachspezifischen Funktionalitäten für COBOL angeboten werden. Neben einigen sprachspezifischen Blocks für Abap, C# und Java sind auch einige wenige generische Blocks vorhanden, welche auch für COBOL verwendet werden können.

Ein Block ist eine XML-Datei, welche die für die Funktionalität notwendigen Java-Klassen festlegt, welche unter ConQAT als *Processor* bezeichnet werden. Darüber hinaus definiert der Block die Abhängigkeiten, Ausführungsreihenfolge, Datenfluss und die konfigurierbaren Parameter der Processor-Klassen. Ein Block kann darüber hinaus auch andere Blocks einbinden. Die

graphische Darstellung eines Blocks zeigt die Abbildung 6. Unter den verfügbaren Processor-Klassen von ConQAT sind jedoch ebenfalls keine COBOL-spezifischen vorhanden.

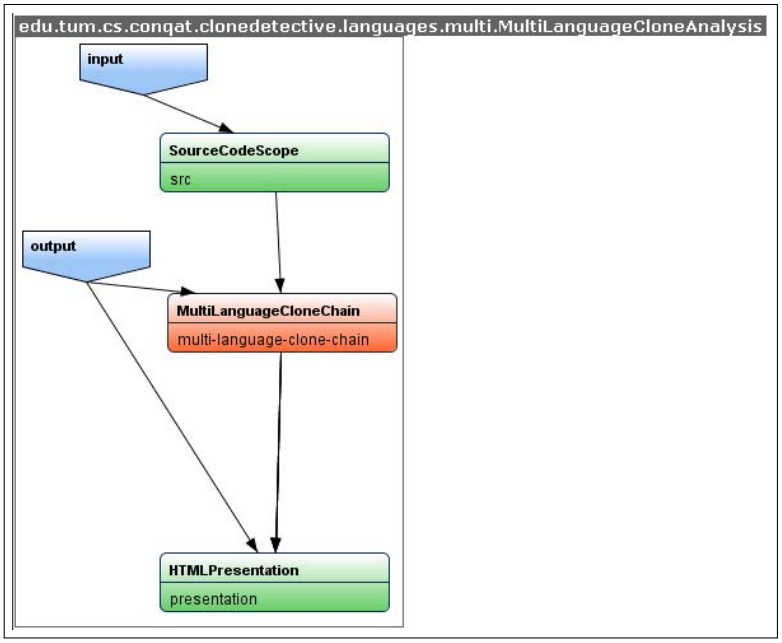

Abbildung 6: Zusammensetzung eines ConQAT Blocks aus Processor-Klassen (*SourceCodeScope*, *HTMLPresentation*), Blocks (*MultiLanguageCloneChain*) und Parametern (*input*, *output*).

Nach dem Anlegen einer Run Configuration müssen noch die notwendigen Parameter - wie etwa Ein- und Ausgabeverzeichnis - konfiguriert werden. Dies geschieht über die ConQAT-Oberfläche. Welche Paremeter notwendig sind, hängt dabei vom verwendeten Block ab. Anschließend kann der Analysevorgang gestartet werden.

4.3.3 Analysevorgang

Die eigentliche Analyse läuft im gleichen Prozess wie das Hauptprogramm ab und zeigt währenddessen Status- und Fehlermeldungen in der Eclipse-internen Konsole an, wie es auch beim Ausführen von Java-Programmen üblich ist. Darüber hinaus wird auch eine ConQAT-eigene Fortschrittsanzeige dargestellt, welche vor allem für den Fall mehrerer parallel laufender Analysen einen Überblick verschafft.

Welche Daten und welches Ausgabeformat während der Analyse erzeugt werden, hängt von dem verwendeten Block ab. Alle der mit ConQAT ausgelieferten Blocks erzeugen HTML-Dateien mit den Ergebnissen, ein Teil davon zusätzlich auch noch eine XML-Datei.

4.3.4 Analyseergebnisse

Die Ergebnisse der Analyse können anschließend in der Eclipse-GUI angezeigt werden. Dabei werden neben der tabellarischen Darstellung die Ergebnisse oft - dies hängt wiederum vom verwendeten Block ab - auch als Treemap[60] angezeigt. Darüber hinaus können auch zusätzliche Informationen wie etwa Fehlerausgaben, Ausführungsdauer und die in Abbildung 6 gezeigte Blockstruktur dargestellt werden.

Abbildung 7 zeigt die Ergebnisse einer Klonanalyse. Über die Links auf der linken Seite können die verschiedenen zuvor genannten Informationen ausgewählt werden. Die Treemap auf der rechten Seite visualisiert die Ergebnisse. Dabei entspricht jedes Rechteck in der Treemap einer Codedatei, die Fläche ist proportional zu der Größe (Lines of Code) der Codedatei. Die Färbung jedes Rechtecks ist abhängig von der Klonbewertung - je höher die Bewertung der Codedatei, desto dunkler die Färbung des Rechtecks. Die Treemap in der Abbildung wird zum Teil von einer weiteren Anzeige überlagert, diese zeigt weitere Detailinformationen zu der jeweiligen Codedatei an, wenn sich der Mauscursor über einem der Felder der Treemap befindet.

4.3.5 Benutzerdefinierte Metriken

Über die Möglichkeit zum Erstellen eines neuen Blocks lassen sich auch eigene Metriken definieren. Ein neuer Block kann in einem Projekt im Workspace direkt mithilfe der GUI angelegt werden. Dem Block können dann - ohne die Notwendigkeit des Programmierens - weitere existierende Blocks und Processorklassen hinzugefügt werden. Der Block wird dabei ähnlich wie in Abbildung 6 in Diagrammform dargestellt und die Komponenten können einfach mittels Drag & Drop sowie Kontextmenü bearbeitet werden. Des Weiteren besteht auch die Möglichkeit eigene Processorklassen in Java zu entwickeln und hierüber die Definition eigener Metriken vorzunehmen.

4.3.6 Ausführen ohne Benutzerinteraktion

Neben der Eclipse-basierten Variante existiert auch die *ConQAT Engine*. Mittels dieser können keine eigenen Blocks oder Run Configurations ertellt, bestehende Run Configurations jedoch ausgeführt werden. Hierzu gibt es eine Stapelverarbeitungsdatei, welche ihrerseits im Wesentlichen aber eine Java-Umgebung mit der entsprechenden ConQAT-Klasse startet. Da die ConQAT Engine komplett in Java realisiert ist, kann alternativ zur Stapelverarbeitungsdatei auch ein Import in eigene Programme erfolgen und die Verarbeitung einer Run Configuration von dort gestartet werden. Ein Ausführen ohne Benutzerinteraktion ist also auf zwei Arten möglich. Der Export der Metrikdaten erfolgt bei den meisten Blocks in HTML-Dateien, diese sind zum Einlesen der berechneten Werte aber eher ungeeignet. Alternativ bringt ConQAT aber

[60] `http://www.cs.umd.edu/hcil/treemap-history/`

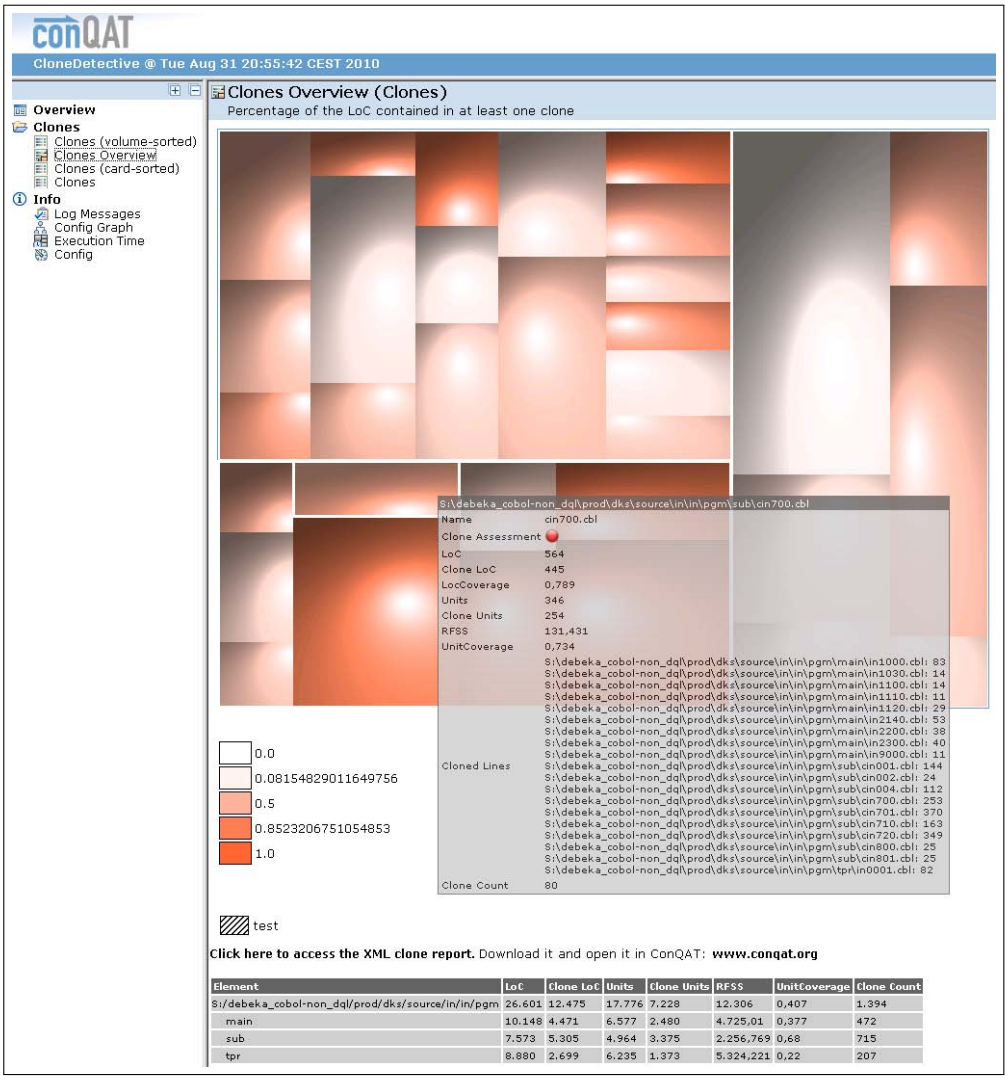

Abbildung 7: Graphische Darstellung von ConQAT-Analyseergebnissen als Treemap.

bereits einen Processor für den XML- (XMLFileWriter) und CSV-Export (CSVFileWriter) mit, gegen die sich in den meisten Blocks der HTML-Export austauschen lässt. Die modifizierten Blocks wiederum lassen sich in den entsprechenden Run Configurations verwendet werden. Die Möglichkeit zum Einlesen und Weiterverarbeiten der Metrikergebnisse ist somit realisierbar, aufgrund der besseren Struktur und Maschinenlesbarkeit ist hierbei der XML-Export die bevorzugte Variante.

4.4 Vorbereitung des COBOL-Quellcodes zur Analyse

Für Tests wurden von der Debeka größere Mengen COBOL-Quellcode bereitgestellt[61], alles in allem etwa 530MB in knapp 15000 Dateien.

Dieser Code enthält jedoch teilweise Anweisungen in der *Debeka Query Language* (DQL), welche in der vorhandenen Form kein gültiger COBOL-Code ist und somit von den Metrik-Tools nicht oder nicht korrekt verarbeitet werden kann. Bei DQL handelt es sich um eine Datenabfragesprache mit einer SQL-ähnlichen Syntax. Zweck dieser Sprache ist die Schaffung einer Abstraktionsebene über die eigentlichen Anfragen an die konkrete Datenbank (siehe [Haas2010-1], S.10-12).

Um dies zu beheben, wurde von der Debeka auch deren DQL-Präprozessor zur Verfügung gestellt. Dieser erzeugt aus einer Eingabedatei mit DQL-Code eine Ausgabedatei mit entsprechend modifiziertem, reinem COBOL-Code.

Zum Zweck der Verarbeitung des gesamten zur Verfügung gestellten Codes durch den Präprozessor wurde zunächst das Hilfsprogramm *UnDQL* geschrieben. Dieses verarbeitet ganze Verzeichnisbäume und wendet den DQL-Präprozessor auf alle darin enthaltenen Codedateien an. Bei dem Tool handelt es sich um ein in Java entwickeltes Kommandozeilenprogramm, die Quellen sind als komplettes Eclipse-Projekt im SVN-Repository dieser Arbeit verfügbar[62].

Nach Anwendung des Tools auf den gesamten zur Verfügung stehenden Quellcode war dieser nun etwa 550MB groß und konnte zum Testen verwendet werden.

4.5 Tests und Analyseergebnisse

Zwecks Übersichtlichkeit sollen an dieser Stelle im Wesentlichen nur die Ergebnisse von zwei kleinen Programmen aus dem verfügbaren Quellcode vorgestellt werden.

Dabei handelt es sich zum einen um das Programm zur *Steuerung der Displaytechnik*, welches sich im Quellcodepaket unter *prod/dks/source/sy/x0* befindet. Es besteht aus lediglich 4 Codedateien ohne Copybooks. Zwar werden im Code sehr wohl Copystrecken referenziert, diese befinden sich jedoch nicht unterhalb von *prod/dks/source/sy/x0*. Während der Analyse dieses Programms werden den Metrik-Tools keinerlei Copybooks verfügbar gemacht, so dass diese auf keinen Fall aufgelöst werden. Ziel hiervon ist die Vergleichbarkeit der Metrikwerte unter Ausschluss der unterschiedlichen Handhabung von Copystrecken durch die einzelnen Tools.

Zum anderen wird das Programm zur *Masseneinspeicherung Tabelle ZT1210* verwendet, welches sich im Quellcodepaket unter *prod/dks/source/zy/ze* befindet. Es besteht aus 6 Codedateien und

[61] Dieser Quellcode ist im SVN-Repository unter `https://svn.uni-koblenz.de/agebert/cobus-nda/debeka_cobol.tar.gz` verfügbar, aber nicht öffentlich zugänglich.

[62] `https://svn.uni-koblenz.de/gup/re-group/trunk/project/cobus-metrics-interop/src/UnDQL`

4 Copybooks. Dieses Programm dient dem Vergleich der Metrikwerte unter Beachtung der unterschiedlichen Handhabung von Copystrecken durch die einzelnen Tools.

Darüber hinaus wird auch die Verarbeitungsgeschwindigkeit getestet. Hierzu wird der Teil des Quellcodepakets unter *prod/dks/source/lv/lb* analysiert. Dieser beinhaltet 751 Codedateien und 434 Copybooks, welche zusammen 47MB groß sind.

4.5.1 SofAudit

Die Tabellen 19 und 20 zeigen die von SofAudit berechneten Metrikwerte für die beiden in Abschnitt 4.5 genannten Programme *Steuerung der Displaytechnik* und *Masseneinspeicherung Tabelle ZT1210*. Der Spaltenbezeichner steht dabei für den Namen der Codedatei. Da die Copybooks wie einzelne Codedateien behandelt werden und somit nicht in die anderen Codedateien einfließen, sind deren Metrikwerte ebenfalls aufgeführt.

In den Tabellen werden nicht alle berechneten Metrikwerte aufgeführt, sondern hauptsächlich die Komplexitäts- und Qualitätsmetriken. Ein Großteil der Quantitätsmetriken (etwa Anzahl der Schleifen, Variablen, Konstanten etc.) wurde an dieser Stelle weggelassen um die Übersichtlichkeit nicht zu beeinträchtigen. Listing 11 im Anhang A zeigt den Inhalt einer von SofAudit generierten Metrikdatei. Aus dieser lässt sich auch eine Übersicht über alle Metriken ablesen, welche von SofAudit berechnet werden können.

Die Verarbeitung von 47MB Code dauert rund 1:20 Minuten. Die Gesamtwerte dieser Berechnung finden sich in Listing 11 im Anhang A.

4.5.2 Flow Graph Manipulator

Die Tabellen 21 und 22 zeigen die vom FGM berechneten Metrikwerte für die beiden in Abschnitt 4.5 genannten Programme *Steuerung der Displaytechnik* und *Masseneinspeicherung Tabelle ZT1210*. Der Spaltenbezeichner steht dabei für den Namen der Codedatei. Dabei handelt es sich nur um die jeweils vorhanden .cbl-Dateien. Da FGM keine Metriken über die Summe der Codedateien berechnet, ist auch keine entsprechende Spalte in den beiden Tabellen enthalten. Copybooks werden von FGM aufgelöst und nicht als einzelne Codedateien verarbeitet, somit fließt ihr Code in Tabelle 22 in die Ergebnisse der Codedateien ein. Da die Werte in Tabelle 21 ohne die Zugriffmöglichkeit auf Copybooks berechnet wurden, spiegeln diese Ergebnisse die Werte der reinen Codedateien wider.

Da der FGM nur eine überschaubare Menge von Metriken berechnet, sind diese vollständig in den Tabellen aufgeführt.

Die Verarbeitung von 47MB Code dauert rund 110 Minuten. Diese Zeit wird fast vollständig von der Analyse / Extraktion der Codedateien benötigt. Der Anteil für die Berechnung der verschiedenen Graphen und des sonstigen Applikationwissens am Ende der Analyse ist dagegen sehr gering.

	x00020	x00021	x00023	x09007	-gesamt-
# Zeilen	900	786	897	500	3083
# Codezeilen	661	541	584	328	2114
# Kommentarzeilen	235	231	294	166	926
# Anweisungen	494	439	450	286	1669
# Sections	39	13	20	7	79
# Paragraphs	115	37	58	19	229
# Datenstrukturen/Objekte	10	32	44	9	95
# CALL-Anweisungen	0	32	10	1	43
# PERFORM-Anweisungen	43	39	30	3	115
# IF-/CASE-Anweisungen	31	53	44	5	133
# Schleifenanweisungen	0	5	0	0	5
# GOTO-Anweisungen	0	8	5	1	14
# Kontrollanweisungen ges.	74	106	79	12	271
Chapin	0,313	0,275	0,248	0,326	0,295
Elshof	0,777	0,752	0,613	0,258	0,636
Card	0,300	0,300	0,300	0,300	0,300
Henry	0,145	0,470	0,344	0,235	0,297
McCabe	0,704	0,691	0,642	0,545	0,675
McClure	0,141	0,255	0,265	0,040	0,180
Sneed	0,195	0,915	0,483	0,064	0,434
Halstead-Komplexität	0,080	0,149	0,255	0,509	0,193
⌀ Komplexität	0,331	0,475	0,390	0,284	0,376
Modularität	0,960	0,871	0,960	0,737	0,960
Portabilität	0,960	0,960	0,960	0,960	0,960
Testbarkeit	0,791	0,693	0,717	0,960	0,792
Wiederverwendbarkeit	0,224	0,362	0,274	0,392	0,283
Konvertierbarkeit	0,735	0,655	0,532	0,891	0,677
Flexibilität	0,093	0,453	0,571	0,746	0,083
Konformität	0,230	0,608	0,593	0,001	0,269
Wartbarkeit	0,564	0,561	0,616	0,641	0,550
⌀ Qualität	0,569	0,645	0,652	0,666	0,571

Tabelle 19: Analyseergebnisse von SofAudit für das Programm *Steuerung der Displaytechnik*

4.5.3 ConQAT

Die Tabellen 23 und 24 zeigen die von ConQAT berechneten Metrikwerte für die beiden in Abschnitt 4.5 genannten Programme *Steuerung der Displaytechnik* und *Masseneinspeicherung Tabelle ZT1210*. Der Spaltenbezeichner steht dabei für den Namen der Codedatei. Da die Copybooks wie einzelne Codedateien behandelt werden und somit nicht in die anderen Codedateien einfließen, sind deren Metrikwerte ebenfalls aufgeführt.

Die Tabellen enthalten alle Metriken, welche sich mittels der mit ConQAT mitgelieferten Analysefunktionen berechnen lassen.

Die Verarbeitung von 47MB Code dauert - abhängig von der verwendeten Analysefunktionalität - zwischen 15 Sekunden und drei Minuten. Im Fall der Klonanalyse - hierbei dürfte es sich um die aufwändigste Analysefunktionalität handeln - brach das Programm nach etwa sechs

Minuten Ausführungsdauer mit einem Stapelüberlauf (Stack Overflow) ab. Ein schrittweises Erhöhen der maximalen Stapelgröße für die Java-VM bis 16MB bewirkte keine Verbesserung, bei Werten darüber hinaus startete Eclipse selbst nicht mehr. Ein Ausführen der Run Configuration mit der konsolenbasierten ConQAT-Engine war mit einem auf 32MB erhöhten Wert möglich und dauerte dabei knapp sechs Stunden.

	zze500.cbl	zze501.cbl	ze8000.cbl	zzc500.cbl	zt1100.cbl	zt1210.cbl	kze500.cpy	kze501.cpy	wze500.cpy	wze501.cpy	-gesamt-
# Zeilen	2641	342	1063	166	2372	769	127	35	222	35	7772
# Codezeilen	1796	190	664	78	1495	382	95	16	146	14	4876
# Kommentarzeilen	852	147	387	88	844	364	32	19	76	21	2830
# Anweisungen	1453	146	353	57	966	265	95	16	116	13	3480
# Sections	20	5	13	2	30	22	0	0	0	0	92
# Paragraphs	59	14	38	5	89	65	0	0	0	0	270
# Datenstrukturen/Objekte	44	9	17	9	25	20	1	1	1	1	128
# CALL-Anweisungen	21	3	1	3	6	5	0	0	0	0	39
# PERFORM-Anweisungen	149	12	51	14	60	43	0	0	0	0	329
# IF-/CASE-Anweisungen	248	11	30	3	124	21	0	0	0	0	437
# Schleifenanweisungen	12	1	15	0	24	0	0	0	0	0	52
# GOTO-Anweisungen	58	8	11	0	61	9	0	0	0	0	147
# Kontrollanweisungen ges.	476	32	97	17	263	73	0	0	0	0	958
Chapin	0,219	0,260	0,301	0,195	0,288	0,263	0,300	0,300	0,300	0,300	0,052
Elshof	0,712	0,517	0,672	0,500	0,828	0,490	0,960	0,960	0,960	0,960	0,655
Card	0,300	0,300	0,845	0,300	0,300	0,300	0,300	0,300	0,300	0,300	0,845
Henry	0,106	0,449	0,731	0,960	0,152	0,533	0,300	0,300	0,300	0,300	0,263
McCabe	0,672	0,727	0,721	0,850	0,645	0,776	0,300	0,300	0,300	0,300	0,680
McClure	0,352	0,201	0,375	0,113	0,386	0,175	0,300	0,300	0,300	0,300	0,339
Sneed	0,523	0,620	0,516	0,960	0,462	0,675	0,300	0,300	0,300	0,300	0,529
Halstead-Komplexität	0,080	0,899	0,370	0,920	0,080	0,306	0,300	0,300	0,300	0,300	0,155
⌀ Komplexität	0,370	0,496	0,566	0,599	0,392	0,439	0,382	0,382	0,382	0,382	0,439
Modularität	0,556	0,960	0,951	0,960	0,521	0,960	0,500	0,500	0,500	0,500	0,753
Portabilität	0,960	0,960	0,241	0,960	0,960	0,960	0,500	0,500	0,500	0,500	0,923
Testbarkeit	0,601	0,749	0,576	0,744	0,615	0,710	0,500	0,500	0,500	0,500	0,645
Wiederverwendbarkeit	0,475	0,391	0,345	0,714	0,289	0,329	0,040	0,960	0,960	0,960	0,379
Konvertierbarkeit	0,604	0,571	0,520	0,742	0,625	0,634	0,121	0,093	0,323	0,346	0,560
Flexibilität	0,597	0,590	0,722	0,793	0,561	0,655	0,500	0,500	0,500	0,500	0,267
Konformität	0,611	0,506	0,583	0,754	0,095	0,535	0,926	0,937	0,991	0,923	0,480
Wartbarkeit	0,599	0,571	0,534	0,597	0,512	0,599	0,555	0,648	0,654	0,647	0,532
⌀ Qualität	0,625	0,662	0,559	0,783	0,522	0,672	0,455	0,579	0,616	0,609	0,567

Tabelle 20: Analyseergebnisse von SofAudit für das Programm *Masseneinspeicherung Tabelle ZT1210*

	x00020	x00021	x00023	x09007
# Zeilen	900	786	897	500
# Codezeilen	661	541	584	328
# Leerzeilen	0	0	0	0
# Kommentarzeilen	138	197	221	80
# leere Kommentarzeilen	101	48	80	92
# Anweisungen	303	318	267	269
⌀ Verschachtelung	3,0132	3,81761	3,46067	3,02974
McCabe	4,0	60,0	45,0	6,0
Halstead-Länge	2028	1400	1043	1576
Halstead-Vokabular	346	226	229	156
Halstead-Volumen	17105,4	10948,3	8176,29	11481,8
Halstead-Komplexität	37,6331	78,0645	32,2537	75,8588
Halstead-Aufwand	643731,0	854670,0	263715,0	870995,0

Tabelle 21: Analyseergebnisse des FGM für das Programm *Steuerung der Displaytechnik*

	cze500	cze501	ze8000	zec500	zt1100	zt1210
# Zeilen	2641	342	1063	166	2372	769
# Codezeilen	1770	190	664	78	1495	382
# Leerzeilen	0	0	0	0	0	0
# Kommentarzeilen	562	80	236	65	569	261
# leere Kommentarzeilen	309	72	163	23	308	126
# Anweisungen	1021	115	251	47	727	204
⌀ Verschachtelung	4,2478	3,25217	3,60558	3,61702	4,49931	3,29902
McCabe	261,0	13,0	46,0	4,0	149,0	22,0
Halstead-Länge	5442	919	154	473	8962	772
Halstead-Vokabular	603	219	66	155	387	263
Halstead-Volumen	50262,4	7145,03	930,837	3441,61	77039,1	6206,05
Halstead-Komplexität	101,704	60,4686	10,0	40,2735	163,136	23,0042
Halstead-Aufwand	5111870,0	432050,0	9308,37	138606,0	1256780000000,0	142765,0

Tabelle 22: Analyseergebnisse des FGM für das Programm *Masseneinspeicherung Tabelle ZT1210*

	x00020	x00021	x00023	x09007	-gesamt-
# Zeilen	900	786	897	500	3083
# Codezeilen	661	539	584	328	2112
Kommentarzeilenanteil	0,344	0,460	0,452	0,249	-
# Klon-Zeilen	289	268	184	40	781
# Units	558	502	526	309	1895
# Klon-Units	200	117	46	24	387
# RFSS	414	432	503	297	1646
UnitCoverage	0,358	0,233	0,087	0,078	0,204
# Klone	29	13	4	2	48

Tabelle 23: Analyseergebnisse von ConQAT für das Programm *Steuerung der Displaytechnik*

	cze500.cbl	cze501.cbl	ze8000.cbl	zec500.cbl	zt1100.cbl	zt1210.cbl	kze500.cpy	kze501.cpy	wze500.cpy	wze501.cpy	-gesamt-
# Zeilen	2641	342	1063	166	2372	769	127	35	222	35	7772
# Codezeilen	1770	190	664	78	1495	382	95	16	146	14	4850
Kommentarzeilenanteil	0,340	0,482	0,418	0,598	0,425	0,604	0,309	0,605	0,303	0,624	-
# Klon-Zeilen	569	171	235	0	1662	643	0	0	0	0	3280
# Units	1712	177	627	74	1406	317	95	16	146	14	4584
# Klon-Units	345	75	185	0	1054	299	0	0	0	0	1958
# RFSS	1512	139	497	74	605	156	95	16	146	14	3254
UnitCoverage	0,202	0,424	0,295	0	0,750	0,943	0	0	0	0	0,427
# Klone	26	5	16	0	96	11	0	0	0	0	154

Tabelle 24: Analyseergebnisse von ConQAT für das Programm *Masseneinspeicherung Tabelle ZT1210*

4.5.4 Vergleich

Tabelle 25 vergleicht die Werte derjenigen Metriken, welche von mindestens zwei der Tools berechnet werden. Der Zeilenbezeichner steht dabei für den Namen der Codedatei. Es wäre zu erwarten gewesen, dass die verschiedenen Tools pro Codedatei und Metrik jeweils identische Ergebniswerte hätten berechnen müssen.

	# Zeilen (SofAudit)	# Zeilen (FGM)	# Zeilen (ConQAT)	# Codezeilen (SofAudit)	# Codezeilen (FGM)	# Codezeilen (ConQAT)	# Anweisungen (SofAudit)	# Anweisungen (FGM)	McCabe (SofAudit)	McCabe(FGM)	Halstead-Komplexität (SofAudit)	Halstead-Komplexität (FGM)
x00020	900	900	900	661	661	661	494	303	0,704	4,0	0,080	37,6331
x00021	786	786	786	541	541	539	439	318	0,691	60,0	0,149	78,0645
x00023	897	897	897	584	584	584	450	267	0,642	45,0	0,255	32,2537
x09007	500	500	500	328	328	328	286	269	0,545	6,0	0,509	75,8588
cze500	2641	2641	2641	1796	1770	1770	1453	1021	0,672	261,0	0,080	101,704
cze501	342	342	342	190	190	190	146	115	0,727	13,0	0,899	60,4686
ze8000	1063	1063	1063	664	664	664	353	251	0,721	46,0	0,370	10,0
zec500	166	166	166	78	78	78	57	47	0,850	4,0	0,920	40,2735
zt1100	2372	2372	2372	1495	1495	1495	966	727	0,645	149,0	0,080	163,136
zt1210	769	769	769	382	382	382	265	204	0,776	22,0	0,306	23,0042

Tabelle 25: Vergleich der Analyseergebnisse

Anzahl Zeilen: Die Werte dieser Metrik werden von allen drei Tools identisch berechnet.

Anzahl Codezeilen: Die Werte dieser Metrik werden von allen drei Tools im Wesentlichen identisch berechnet. In einem Fall berechnet ConQAT zwei Codezeilen weniger als FGM und SofAudit, in einem anderen Fall berechnet SofAudit 26 Codezeilen mehr als die anderen beiden Tools.

Anzahl Anweisungen: SofAudit und FGM berechnen bei dieser Metrik für alle Codedateien jeweils voneinander abweichende Werte. Da abweichende Werte auch bei denjenigen Codedateien auftreten, für die FGM während der Analyse keinen Zugriff auf Copybooks hatte, kommt eine Begründung mit der unterschiedlichen Auflösung von Copystrecken nicht in Frage. Im Diagramm in Abbildung 8 werden die berechneten Werte über die Codedateien aufgetragen. Zwecks einfacherer Vergleichbarkeit für den Betrachter ist die Folge von Werten pro Tool jeweils als Linie dargestellt. Der Wertebereich beider Folgen

ist zwar nicht identisch, liegt aber in vergleichbaren Größenordnungen. Die Form des Verlaufs beider Linien ist gleichartig und die Minima und Maxima beider Wertefolgen treten über den jeweils gleichen Codedateien auf. Ausgehend von den zuvor genannten Gründen ist eine Korrelation der Werte von SofAudit und FGM gegeben.

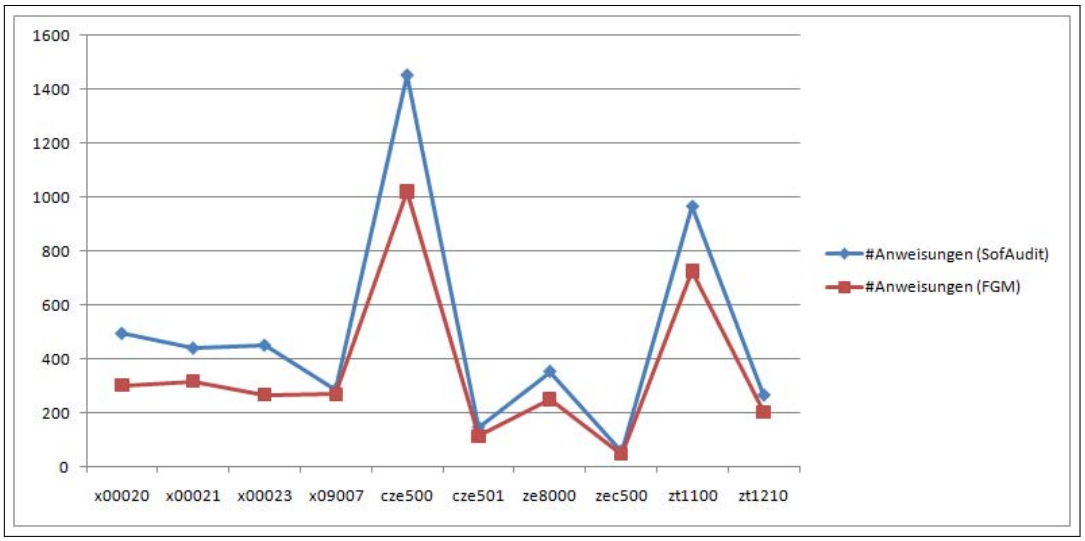

Abbildung 8: Diagramm der berechneten Metrikwerte für die Anzahl der Anweisungen von SofAudit und FGM.

McCabe: SofAudit und FGM berechnen bei dieser Metrik für alle Codedateien jeweils voneinander abweichende Werte. Da abweichende Werte bei allen Codedateien auftreten, scheidet die unterschiedliche Behandlung von Copystrecken durch die Tools - wie schon zuvor bei der Anzahl der Anweisungen - als Grund aus. In der Abbildung 9 werden die berechneten Wertefolgen als Liniendiagramme über die Codedateien aufgetragen. Abgesehen von der erheblichen Abweichung des Wertebereichs der Diagramme ist auch die Form des Verlaufs beider Linien unterschiedlich. Darüber hinaus treten die Minima und Maxima beider Wertefolgen über unterschiedlichen Codedateien auf. Ausgehend von den zuvor genannten Gründen ist keine Korrelation der Werte von SofAudit und FGM für die McCabe-Metrik erkennbar.

Halstead-Komplexität: SofAudit und FGM berechnen bei dieser Metrik für alle Codedateien jeweils voneinander abweichende Werte. Da abweichende Werte bei allen Codedateien auftreten, scheidet auch hier die unterschiedliche Behandlung von Copystrecken durch die Tools als Grund aus. In der Abbildung 10 werden die berechneten Wertefolgen als Liniendiagramme über die Codedateien aufgetragen. Wieder zeigt sich eine erheblichen Abweichung des Wertebereichs der Diagramme und Unterschiede in der Form des Verlaufs beider Linien. Ebenso treten die Minima und Maxima beider Wertefolgen über unterschiedlichen Codedateien auf. Ausgehend von den zuvor genannten Gründen ist

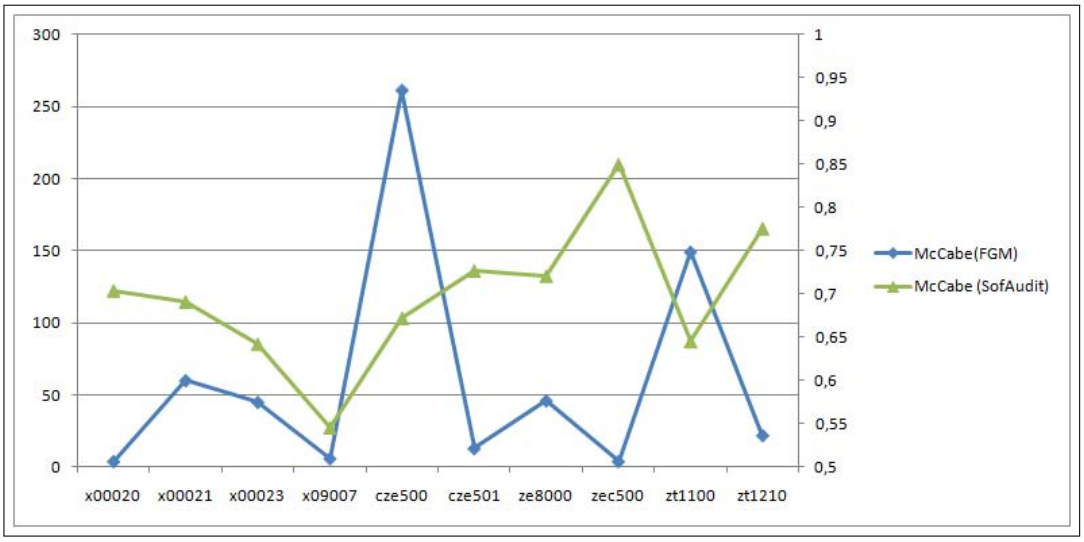

Abbildung 9: Diagramm der berechneten Metrikwerte für die McCabe-Metrik von SofAudit (Wertebereich: rechte Y-Achse) und FGM (Wertebereich: linke Y-Achse).

keine Korrelation der Werte von SofAudit und FGM für die Halstead-Komplexität erkennbar.

Tests mit anderem Quellcode haben die hier genannten Ergebnisse bestätigt. Eine Analyse der Abweichungen in den Ergebnissen folgt im nächsten Abschnitt.

4.6 Analyse der abweichenden Ergebnisse

Da die von den Tools berechenbaren Metrikwerte zur Bewertung von Software dienen sollen, ist eine Untersuchung der abweichenden Ergebnisse unerlässlich.

4.6.1 Anzahl echter Codezeilen

Bei den berechneten Werten für die Anzahl echter Codezeilen treten nur vereinzelt Abweichungen auf. Obwohl diese Abweichungen nur als geringfügig einzustufen sind, sollen die Gründe hierfür dennoch untersucht werden.

Wie in Tabelle 25 gezeigt, wird für die Datei *X00021.CBL* eine Anzahl von 539 echten Codezeilen von ConQAT berechnet. Sowohl SofAudit als auch FGM berechnen hier 541 Codezeilen. Durch Unterteilung der analysierten Datei in immer kleinere Codefragmente stellte sich heraus, dass die Ursache für diese Differenz in den beiden in Listing 1 gezeigten Codezeilen liegt. ConQAT interpretiert diese Zeilen weder als Code noch als Kommentar. SofAudit und FGM hingegen interpretieren diese Zeilen als Code. Eine Untersuchung aller anderen Codedateien aus Tabelle 25 ergab, dass entsprechende Anweisungen darin nicht vorhanden waren. Durch

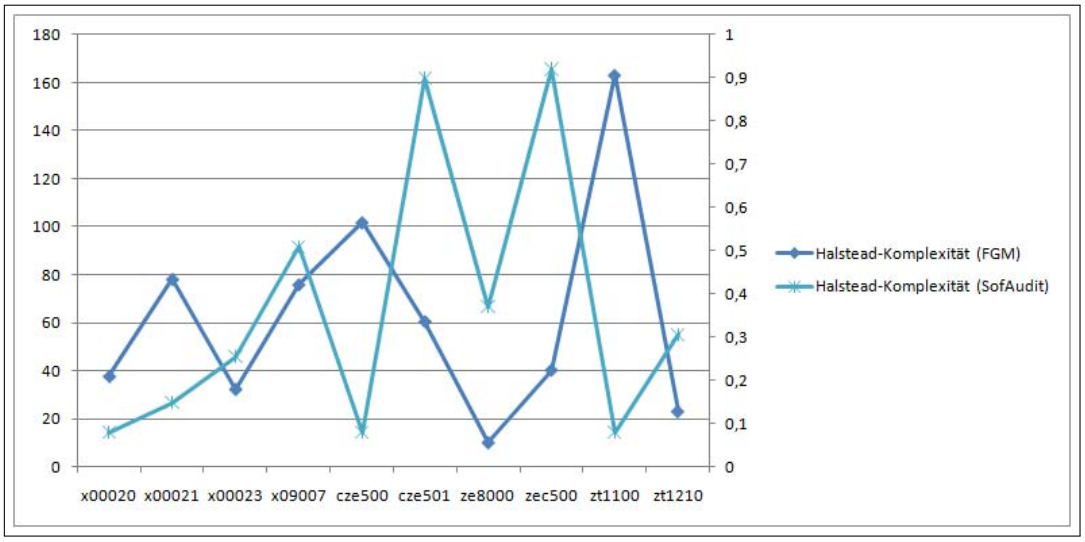

Abbildung 10: Diagramm der berechneten Metrikwerte für die Halstead-Komplexität von SofAudit (Wertebereich: rechte Y-Achse) und FGM (Wertebereich: linke Y-Achse).

Analyse weiterer Dateien mit entsprechenden Codezeilen oder durch Hinzufügen in beliebige Dateien können entsprechende Abweichungen in den Ergebnissen reproduziert werden.

```
1 AUTHOR. FISCHER
2 DATE-WRITTEN. 10-03-87.
```

Listing 1: Ausschnitt aus X00021.CBL

In Tabelle 25 wird auch gezeigt, dass SofAudit für die Datei *CZE500.CBL* eine Anzahl von 1796 echten Codezeilen berechnet. Sowohl FGM als auch ConQAT berechnen hier 1770 Codezeilen. Eine Unterteilung der analysierten Datei in kleinere Codefragmente brachte in diesem Fall keinen Aufschluss. Das Entfernen aller Kommentarzeilen hat zur Folge, dass SofAudit anschließend ebenfalls 1770 echte Codezeilen berechnet. Durch schrittweises Entfernen von Zeilen mit auskommentiertem Code wurde letztendlich klar, dass jede Zeile mit einer auskommentierten *COPY*-Referenz von SofAudit fälschlicherweise als Codezeile interpretiert wird. Zusätzlich wird eine solche Zeile aber auch als Kommentarzeile mitgezählt. Andere auskommentierte Zeilen - etwa solche mit *MOVE*-, *IF*- oder *PERFORM*-Anweisungen - wurden hingegen korrekterweise ausschließlich als Kommentarzeilen interpretiert. Eine Untersuchung aller anderen Codedateien aus Tabelle 25 ergab, dass entsprechende Zeilen darin nicht vorhanden waren. Durch Analyse weiterer Dateien mit entsprechenden Codezeilen oder durch Hinzufügen in beliebige Dateien können entsprechende Abweichungen in den Ergebnissen reproduziert werden.

4.6.2 Anzahl der Anweisungen

Die Berechnung der Anzahl der Anweisungen soll anhand des einfachen Codes aus Listing 2 untersucht werden. Dieser Code beinhaltet vier Anweisungen[63].

```
1  IDENTIFICATION DIVISION.
2  PROGRAM-ID. A.
3  DATA DIVISION.
4  WORKING-STORAGE SECTION.
5  01  W-FELD    PIC X(10).
6  PROCEDURE DIVISION.
7  A000-STEUERUNG SECTION.
8     MOVE "HELLO" TO W-FELD.
9     DISPLAY W-FELD.
10    MOVE "WORLD" TO W-FELD.
11    DISPLAY W-FELD.
```

Listing 2: COBOL-Testprogramm mit vier Anweisungen

SofAudit berechnet für diesen Code sechs Anweisungen. Durch Löschen einzelner Zeilen zeigt sich, dass die letzten vier Zeilen als einzelne Anweisungen interpretiert werden. Darüber hinaus wird auch die Variablendeklaration in Zeile 5 als Anweisung interpretiert. Ein Löschen von Zeilen mit Section-, Division- oder Program-Id-Definitionen stellt keinen korrekten COBOL-Code dar und hat ein Ergebnis von null Anweisungen zur Folge. Deshalb kann nicht bestimmt werden, aus welcher Zeile die sechste Anweisung resultiert.

FGM berechnet für diesen Code eine Anzahl von null Anweisungen. Dies macht einen weiteren Test mit einem anderen Code notwendig, welcher in Listing 3 gezeigt ist und acht Anweisungen beinhaltet[64].

FGM berechnet für diesen zweiten Code die korrekte Anzahl von acht Anweisungen. Wird eine weitere DISPLAY-Anweisung hinzugefügt, so erhöht sich bei diesem Code auch der berechnete Wert um eins. Dies ist unabhängig davon, ob die Anweisung innerhalb oder außerhalb der *IF*- oder *PERFORM*-Anweisungen eingefügt wird.

SofAudit berechnet für den zweiten Code 13 Anweisungen. Durch das Löschen und Einfügen entsprechender Zeilen zeigt sich, dass die *IF*- und *PERFORM*-Anweisungen doppelt gezählt werden. Rechnet man dies mit der Anzahl der eigentlichen Anweisungen und den beiden Variablendeklarationen aus der fünften und sechsten Codezeile zusammen, ergibt dies eine Summe von zwölf. Wie schon beim vorherigen Testcode kann auch hier nicht bestimmt werden, aus welcher Zeile die überzählige Anweisung resultiert.

Die Tests mit den Quellcodes zeigen, dass beide Programme nicht ganz fehlerfrei bei der Berechnung der Anweisungen funktionieren. Die Werte können aber durchaus als Grundlage einer Abschätzung der jeweiligen Größenordnung dienen.

[63] Die vier Anweisungen befinden sich in den letzten vier Zeilen von Listing 2: Jeweils zwei *MOVE*- und zwei *DISPLAY*-Anweisungen.

[64] Die Anweisungen befinden sich in den letzten elf Zeilen von Listing 3: zwei *MOVE*-, vier *DISPLAY*-Anweisungen sowie jeweils eine *IF*- und *PERFORM*-Anweisung.

```
1  IDENTIFICATION DIVISION.
2  PROGRAM-ID. A.
3  DATA DIVISION.
4  WORKING-STORAGE SECTION.
5  01  W-FELD      PIC X(10).
6  01  W-NUMMER    PIC 9(2).
7  PROCEDURE DIVISION.
8  A000-STEUERUNG SECTION.
9     MOVE "HELLO" TO W-FELD.
10    DISPLAY W-FELD.
11    MOVE "WORLD" TO W-FELD.
12    DISPLAY W-FELD.
13    IF  W-FELD = "WORLD"
14       DISPLAY "!"
15    END-IF.
16    PERFORM VARYING W-NUMMER FROM 1 BY 1
17     UNTIL W-NUMMER > 10
18       DISPLAY "NR.=" W-NUMMER
19    END-PERFORM.
```

Listing 3: COBOL-Testprogramm mit acht Anweisungen

4.6.3 McCabe

Die Untersuchung der Ergebnisabweichungen bei der Berechnung der zyklomatischen Komplexitätszahl (McCabe-Metrik) erfolgt an dieser Stelle zunächst anhand eines Vergleichs der zugrunde liegenden Berechnungsvorschriften von FGM und SofAudit. Die hierfür notwendigen Informationen sind bei beiden Programmen im Handbuch enthalten.

Für FGM wird die Berechnungsvorschrift für die McCabe-Metrik wie folgt beschrieben (siehe [proetcon2009], S.116):
„Sei G der Kontrollflussgraph eines Moduls. Ist G zusammenhängend, so ist die zyklomatische [...] Zahl v von G:

$$v = e - n + 1,$$

wobei e die Anzahl der Kanten (edges) und n die Anzahl der Knoten (nodes) ist. Besteht G aus p Komponenten, so ist

$$v = e - n + 2p.$$

Die zyklomatische Zahl oder McCabe-Metrik kann als Anzahl linear unabhängiger Wege durch das Programm interpretiert werden.“

Dies entspricht der allgemein verbreiteten Definition der McCabe-Metrik, wie sie unter anderem auch in [McCabe1976] auf Seite 308 aufgeführt ist. Per Definition handelt es sich bei den Werten der McCabe-Metrik immer um einen ganzzahligen Wert, was bei den bislang von FGM

berechneten Werten der Fall ist. Die für die Quellcodes aus Listing 2 und 3 berechneten Werte von eins beziehungsweise vier sind korrekt.

Für SofAudit wird die Berechnungsvorschrift für die McCabe-Metrik mit folgender Formel angegeben (siehe [Sneed2008-2], S.61):

$$CtrlFlowCompl = \frac{Branches - (Selections + Cases + Loops)}{Branches}$$

Die Bezeichner stehen dabei für folgende Werte:

- *Branches*: Anzahl der Verzweigungen des Kontrollflusses
- *Selections*: Anzahl der *IF*-Anweisungen
- *Cases*: Anzahl der *CASE*-Anweisungen
- *Loops*: Anzahl der Schleifen (*UNTIL*, *WHILE*)

Der hierbei berechnete Wert basiert auf der Anzahl all jener Anweisungen, welche Verzweigungen im Kontrollfluss verursachen. Die Ergebnisse sind auf den Wertebereich zwischen null und eins skaliert. Die Berechnungsvorschrift weicht wesentlich von der zuvor bei FGM gezeigten ab, womit es sich bei dem von SofAudit berechneten Wert auch nicht um die McCabe-Metrik im eigentlichen Sinne handelt.

4.6.4 Halstead-Komplexität

Die Untersuchung der Ergebnisabweichungen bei der Berechnung der Halstead-Komplexität erfolgt an dieser Stelle zunächst anhand eines Vergleichs der zugrunde liegenden Berechnungsvorschriften von FGM und SofAudit. Die hierfür notwendigen Informationen sind bei beiden Programmen im Handbuch enthalten.

Für FGM wird die Berechnungsvorschrift für die Halstead-Komplexität wie folgt beschrieben (siehe [proetcon2009], S.115-116):
Seien in einer Sourcecode-Komponente:
n_1*: Anzahl unterschiedlicher Operatoren,*
n_2*: Anzahl unterschiedlicher Operanden,*
N_1*: Gesamtzahl der verwendeten Operatoren,*
N_2*: Gesamtzahl der verwendeten Operanden.*
Programmschwierigkeit D:

$$D = \frac{n_1}{2} * \frac{N_2}{n_2}$$

Dies entspricht der allgemein verbreiteten Definition der Halstead-Komplexität, wie sie unter anderem auch in [Halstead1979] aufgeführt ist. Da aber auch Halstead selbst nicht genau definiert hat, welche Codeelemente als Operatoren und Operanden zu interpretieren sind, konnten die von FGM für die beiden Beispielprogramme berechneten Werte nicht nachvollzogen

werden. Zur detaillierten Analyse der berechneten Werte wurde zusätzlich der Quellcode aus Listing 4 verwendet. Grund hierfür ist die Tatsache, dass die Unterscheidung von Operatoren und Operanden für den menschlichen Betrachter einfacher ist, wenn es sich bei diesen auch wirklich um solche aus mathematischen Berechnungen handelt.

Mittels manueller Berechnung der Ergebnisse, Vergleich mit den von FGM berechneten Werten, Zuhilfenahme der von FGM berechneten Halstead-Implementierungslänge[65] und des Halstead-Vokabulars[66], sowie dem sukzessiven Hinzufügen der einzelnen Codezeilen zu dem Code aus Listing 4 konnten die Berechnungen nachvollzogen werden.

FGM zählt für n_1 nicht nur die eigentlichen Operatoren wie etwa *ADD* und *DIVIDE* mit, sondern auch Schlüsselwörter wie etwa *TO*, *FROM*, *GIVING* und darüber hinaus auch die Klammern in der letzten *COMPUTE*-Anweisung[67]. Bei der Zahl der unterschiedlichen Operanden n_2 werden neben den Variablen auch alle fest codierten Zahlen mitgezählt, dabei mehrfach vorkommende Zahlenwerte wie mehrfach referenzierte Variablen aber nur einmal gezählt. Zu n_2 addiert FGM stets noch eins hinzu[68]. Auch zu der Gesamtzahl verwendeter Operanden N_2 addiert FGM stets noch eins hinzu[69]. Der wahrscheinlichste Grund für dieses Verhalten dürfte die Vermeidung von Fehlern durch eine Division durch null sein, falls eine analysierte Codedatei leer sein sollte oder keine Anweisungen enthält.

```
1  IDENTIFICATION DIVISION.
2  PROGRAM-ID. A.
3  DATA DIVISION.
4  WORKING-STORAGE SECTION.
5  01  W-NUMMER    PIC 9(2).
6  PROCEDURE DIVISION.
7  A000-STEUERUNG SECTION.
8     ADD 1 TO W-NUMMER.
9     ADD 1 2 3 4 5 6 7 8 9 10 GIVING W-NUMMER.
10    SUBTRACT 2 FROM W-NUMMER.
11    SUBTRACT 1 2 3 4 5 FROM 15 GIVING W-NUMMER.
12    MULTIPLY 3 BY W-NUMMER.
13    DIVIDE 4 INTO W-NUMMER.
14    COMPUTE W-NUMMER = (W-NUMMER + 1 - 2) * 3 / 4.
```

Listing 4: COBOL-Testprogramm mit mathematischen Anweisungen

Mit dieser Berechnungsweise ist auch die Halstead-Komplexität von 21,5769 für den Code aus Listing 4 und 12,2778 als Wert[70] für den Code aus Listing 3 nachvollziehbar. Für den Code aus Listing 2 wäre allerdings ein Wert von 1,25 zu erwarten[71], die Abweichung dürfte aber - wie bei der berechneten Anzahl von null Anweisungen - in einem Fehler von FGM begründet liegen.

[65] Halstead-Implementierungslänge *N*: $N = N_1 + N_2$
[66] Halstead-Vokabular *n*: $n_1 + n_2$
[67] Für den Code aus Listing 4 ergibt sich somit $n_1 = 17$.
[68] Für den Code aus Listing 4 ergibt sich eigentlich $n_2 = 12$, FGM rechnet mit $n_2 = 13$.
[69] Für den Code aus Listing 4 ergibt sich eigentlich $N_2 = 32$, FGM rechnet mit $N_2 = 33$.
[70] Für den Code aus Listing 3 ist nach der Berechnungsweise von FGM $n_1 = 13$; $n_2 = 9$; $N_2 = 17$.
[71] Für den Code aus Listing 2 ist nach der Berechnungsweise von FGM $n_1 = 2$; $n_2 = 4$; $N_2 = 5$.

Für SofAudit wird die Berechnungsvorschrift für die Halstead-Komplexität mit folgender Formel angegeben (siehe [Sneed2008-2], S.61):

$$LangCompl = \frac{StatementTypes * 4}{ProcStatements} * \frac{DataUsed * 2}{DataReferences}$$

Die Bezeichner stehen dabei für folgende Werte:

- *StatementTypes*: Anzahl verwendeter Anweisungstypen
- *ProcStatements*: Anzahl prozeduraler Anweisungen
- *DataUsed*: Anzahl referenzierter Datenelemente
- *DataReferences*: Anzahl Referenzen auf Datenelemente

Der hierbei berechnete Wert basiert sinngemäß auf den gleichen Informationen aus dem Quelltext wie die eigentliche Halstead-Komplexität. Da die hier angewendete Berechnung sich aber doch wesentlich von der eigentlichen Halstead-Definition unterscheidet und der von SofAudit berechnete Wert zwischen null und eins skaliert ist, handelt es sich hierbei nicht um die Halstead-Komplexität im eigentlichen Sinne.

4.6.5 Weitere Metriken

Die Untersuchungen der von SofAudit verwendeten Berechnungsvorschriften für die McCabe- und Halstead-Metrik legt die Vermutung nahe, dass SofAudit auch bei der Berechnung der weiteren Komplexiätsmetriken von den allgemein verbreiteten Algorithmen abweicht und diese auf den Wertebereich zwischen null und eins skaliert. Eine weitergehende Untersuchung hierzu findet sich in Anhang B.

5 Entwurf eines Metrikdatenformates

Jedes der in Kapitel 4 untersuchten Tools verwendet ein eigenes Datenformat. Das Ziel ist die Interoperabilität dieser - und später möglicherweise auch weiterer - Tools mit der bei der Debeka verwendeten Software. Die Interoperabilität wird wesentlich erleichtert, wenn die durch die Metrik-Tools generierten Ergebnisse in einer einheitlichen Darstellung vorliegen oder in eine solche transformiert werden. Darüber hinaus wird ein Vergleich der von den verschiedenen Tools berechneten Werte vereinfacht. Der Entwurf eines universellen Datenformats zum Speichern und Austausch von Softwaremetrikdaten erfolgt in diesem Kapitel.

5.1 Begriffsklärung

Zunächst ist es angebracht, die einzelnen Begriffe zu klären und voneinander abzugrenzen.

Eine *Metrik* legt fest, welche Eigenschaft eines Messobjektes gemessen wird.

Das *Messobjekt* ist das Artefakt, über welches gemessen wird. Bei Softwaremetriken kann es sich etwa um Quellcode, Programme, Datenbestände oder auch Entwurfsdokumente handeln. Im Fall der hier zum Einsatz kommenden Tools handelt es sich beim Messobjekt jedoch stets um eine beliebige Menge Quellcode. Das kann etwa eine Datei, ein Verzeichnis mit mehreren Dateien oder ein Projekt (eine Zusammenstellung aus den zuvor genannten Elementen) sein. Damit kann aber auch ein beliebiger Codeabschnitt gemeint sein, welcher nicht an die üblichen Grenzen der Dateisysteme gebunden ist.

Die *Ausprägung* eines Messobjektes bezeichnet ein Messobjekt in einem konkreten Zustand. Wird zum Beispiel eine Codedatei geändert, so handelt es sich vor und nach der Änderung zwar um das gleiche Messobjekt, aber um eine jeweils andere Ausprägung. Im Bereich der Softwaremetriken wird die Ausprägung auch häufig als *Version* bezeichnet.

Ein *Metrikwert* ist das berechnete Ergebnis einer bestimmten Metrik über eine konkrete Ausprägung eines Messobjektes. Wichtig ist dabei, dass der Wert *nur* für die konkrete Ausprägung gültig ist. Es ist durchaus üblich auch den Wert als *Metrik* zu bezeichnen. Im Rahmen dieser Arbeit soll aber zugunsten der Eindeutigkeit stets die Unterscheidung zwischen Metrik und Metrikwert zur Anwendung kommen.

5.2 Darstellung eines Metrikwertes

Es ist notwendig, die Eigenschaften eines einzelnen Metrikwertes zu betrachten. Einige Metriken stehen durchaus im Zusammenhang zueinander, beschreiben dabei aber dennoch unterschiedliche Eigenschaften eines Messobjektes. Als Beispiel kann an dieser Stelle die durchschnittliche und die maximale Verschachtelungstiefe einer Codedatei dienen. Jeder der beiden Werte beschreibt genau genommen eine andere Eigenschaft und sollte somit einer eigenen Metrik zugehörig sein.

Bei den Werten der meisten Metriken handelt es sich um numerische Werte. Bei den Größenmaßen sind ganzzahlige Werte üblich, wie etwa die Anzahl der Codezeilen („Lines of Code“ - LoC) oder die Anzahl der definierten Variablen. Bei den Werten der Dichtemaße handelt es sich üblicherweise um reelle Zahlen im mathematischen Sinne, also um Gleitkommazahlen im Sinne der Informatik – wie etwa dem Verhältnis von echten zu gesamten Codezeilen. Bei den Strukturmaßen sind beide Arten vertreten, etwa die mittlere (Gleitkommawert) und maximale (ganzzahliger Wert) Verschachtelungstiefe.

Darüber hinaus ist es aber auch möglich, dass Metriken mit nicht-numerischen Werten zum Einsatz kommen. Möglich sind hierbei zum Beispiel Enum-Typen oder boolesche Werte zur Darstellung verschiedener Skalarniveaus, etwa die Einteilung in Komplexitätsklassen (*hoch*, *mittel*, *niedrig*) oder Erfüllung einer Eigenschaft (*true*, *false*). Ideal wäre jedoch die Möglichkeit zur Speicherung jedes beliebigen Datentyps.

Die Werte *einer* konkreten Metrik sind also stets von *einem* spezifischen Datentyp. Darüber hinaus besitzt jeder Wert eine Einheit, welche auch immer für *eine* konkrete Metrik spezifisch ist - zum Beispiel „Textzeilen“ für die Metrik „Lines of Code“.

5.3 Datenformat zur Speicherung von Metriken in einer Datei

Nachdem im letzten Abschnitt die Möglichkeiten zur Repräsentation eines Metrikwertes erläutert wurden, sollen nun die Bedingungen für die Speicherung in einer Datei erörtert werden.

Das Datenformat sollte dabei nach Möglichkeit folgende Eigenschaften erfüllen:

- Eignung zur Datenspeicherung
- Gute Strukturierbarkeit
- Einfache Erweiterbarkeit
- Flexibilität hinsichtlich speicherbarer Datentypen
- Maschinenverarbeitbarkeit, jedoch ohne die Lesbarkeit durch Menschen unnötig zu erschweren
- Verwendung verbreiteter und allgemein anerkannter Standards
- Verarbeitung durch bestehende Software der Debeka mit geringen Aufwand realisierbar (Interoperabilität)

Diese Eigenschaften werden im Wesentlichen durch die Extensible Markup Language (XML)[72] erfüllt, welche weithin Anwendung findet und allgemein akzeptiert ist. Für XML spricht darüber hinaus auch, dass das Zentrale Debeka Repository (Zeder) bereits entsprechende Verarbeitungsfunktionalitäten besitzt. Da XML aber nur die grundlegenden Elemente und die Syntax festlegt, muss eine genauere Struktur zur Speicherung von Metriken in Form eines XML-Schemas definiert werden.

[72] `http://www.w3.org/XML`

Metrikbezeichner. Da Werte zu mehreren Metriken in einer Datei möglich sein sollten, muss zu jedem Wert auch ein eindeutiger Metrikbezeichner angegeben werden. Dazu eignet sich am besten ein Bezeichner in Form einer Zeichenfolge, in welcher ein beschreibender, eindeutiger Name der jeweiligen Metrik gespeichert wird. Es wäre aus technischer Sicht zwar ein numerischer Bezeichner (in Form einer Metriknummer) oder ein Bezeichner in Form eines Kürzel ebenfalls denkbar. Dies würde jedoch die Lesbarkeit für menschliche Betrachter verschlechtern. Sollten die Verwendung von Kurzbezeichnern dennoch notwendig sein, so sollte zwecks Vermeidung von Redundanzen nur der Kurzbezeichner gespeichert werden. Eine Zuordnung des vollständigen Bezeichners zum jeweiligen Kürzel sollte separat an einer anderen Stelle gespeichert werden.

Die Speicherung des Bezeichners sollte in Form eines XML-Attributwertes erfolgen. Die Alternative – eine Speicherung über den Bezeichner des XML-Elementes – hätte den Nachteil, dass für jede neue Metrik auch das XML-Schema angepasst werden müsste.

Metrikwerte. Die Speicherung der Metrikwerte sollte in deren String-Darstellung erfolgen. Zwar lässt XML über die Schemata die Definition einer Typeinschränkung zu, im Endeffekt wird aber immer in Form eines Strings gespeichert. Hinsichtlich der Flexibilität bezüglich der Speicherung von Metrikwerten beliebigen Typs sollte auch über das XML-Schema hierfür keine Einschränkung definiert werden.

Eine explizite Speicherung des Datentyps und der Einheit zu jedem Wert ist nicht sinnvoll, da Beides implizit über den jeweiligen Metrikbezeichner festgelegt ist.

Metrikdefinitionen. Eine Speicherung von Bezeichner, Beschreibung bzw. Langform des Bezeichners und Einheit zu jeder Metrik ist in den Daten dennoch nicht vermeidbar, damit diese auch zu einem späteren Zeitpunkt oder für andere Programme verfügbar sind. Die Speicherung von diesen Angaben getrennt von den berechneten Metrikwerten vermeidet Redundanzen oder Widersprüche, erhöht die Änderbarkeit und verringert das Datenvolumen.

Messobjekte und Ausprägungen. Zusätzlich muss gespeichert werden, auf welches Messobjekt sich die Metrikwerte beziehen. Bei allen bislang genauer untersuchten Tools ist die einzelne Quellcodedatei die kleinste Einheit, welche als Messobjekt verwendet werden kann. Darüber hinaus können aber auch Verzeichnisse oder Projekte als Messobjekt dienen. Auch hier sollte im Idealfall die Möglichkeit gewahrt bleiben, beliebige andere Messobjekte verwenden zu können, ohne das hierfür zugrunde liegende Schema ändern zu müssen. Dies ist möglich, indem der Typ des Messobjektes als XML-Attributwert gespeichert wird. Die eigentliche Referenz auf das Messobjekt sollte einem möglichst einheitlichen Schema entsprechen. Denkbar hierfür wäre etwa die Verwendung von URN[73] oder URI[74]. Da bei der Debeka das Versions-

[73] URN: Uniform Resource Name, siehe `http://tools.ietf.org/html/rfc2141`
[74] URI: Uniform Resource Identifier, siehe `http://tools.ietf.org/html/rfc3986`

verwaltungssystem SVN[75] im Einsatz ist (siehe [Haas2010-1], S. 21), erfüllt die SVN-Adresse des Messobjekts diese Voraussetzung und ist auf beliebigen Systemen stets einheitlich. Es ist allerdings auch denkbar, dass Metriken lokal auf einem Entwicklercomputer auf nicht unter der Versionsverwaltung stehendem Code berechnet wird. In diesem Fall ist eine Zuordnung nur temporär und lokal möglich und der Pfad des Objektes ist hierfür ausreichend.

Neben dem Bezug auf das Messobjekt muss aber auch der Bezug auf die Ausprägung gespeichert werden, hierfür kann die SVN-Versionsnummer verwendet werden. Im Fall von lokal modifiziertem Code kann dies durch entsprechende Markierung der Versionsnummer verdeutlicht werden. Im Fall von nicht versionierten Code kann die Zuordnung wieder nur temporär und lokal erfolgen und es erfolgt keine Angabe.

Um Fälle abbilden zu können, in denen ein Messobjekt Teil eines anderen Messobjektes ist, sollte die Möglichkeit der Verschachtelung im Speicherformat gegeben sein. Ein Beispiel einer XML-Struktur, in welcher alle zuvor genannten Punkte berücksichtigt wurden, findet sich in Listing 5. Das zugehörige XML-Schema findet sich in Listing 6. Abbildung 11 zeigt eine Entsprechung des XML-Schemas als UML-Klassendiagramm. Die Klassen entsprechen dabei XML-Nodes, die Klassennamen den Nodenamen und die Klassenattribute den XML-Attributen.

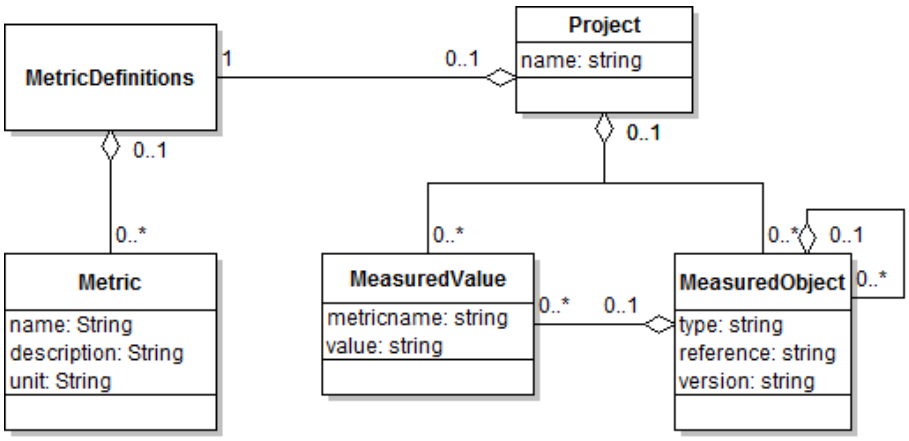

Abbildung 11: UML-Diagramm des XML-Schemas zur Speicherung von Metrikdaten.

[75] SVN: Subversion, siehe `http://subversion.apache.org/`

```
<?xml version="1.0" encoding="utf-8"?>

<project name="Example project" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="metricsschema.xsd">

  <metricdefinitions>
    <metric name="loc" description="Total lines of code" unit="lines" />
    <metric name="avgnestdepth" description="Average nesting depth" unit="depth" />
    <metric name="compclass" description="Complexity classification" unit="complexity" />
    <metric name="y2kcompl" description="Y2k compliancy" unit="compliancy" />
  </metricdefinitions>

  <measuredobject type="project" reference="svn://my.server/repo/trunk/exproj">

    <measuredobject type="directory" reference="svn://my.server/repo/trunk/exproj/src">

      <measuredobject type="file"
          reference="svn://my.server/repo/trunk/exproj/src/component1.cbl" version="123">
        <measuredvalue metricname="loc" value="123" />
        <measuredvalue metricname="avgnestdepth" value="4.56" />
        <measuredvalue metricname="compclass" value="high" />
        <measuredvalue metricname="y2kcompl" value="true" />
      </measuredobject>

      <measuredobject type="file"
          reference="svn://my.server/repo/trunk/exproj/src/component2.cbl" version="123">
        <measuredvalue metricname="loc" value="42" />
        <measuredvalue metricname="avgnestdepth" value="1.23" />
        <measuredvalue metricname="compclass" value="low" />
        <measuredvalue metricname="y2kcompl" value="true" />
      </measuredobject>

      <measuredvalue metricname="loc" value="165" />
      <measuredvalue metricname="avgnestdepth" value="3.51" />
      <measuredvalue metricname="compclass" value="medium" />
      <measuredvalue metricname="y2kcompl" value="true" />
    </measuredobject>

    <measuredobject type="file" reference="svn://my.server/repo/trunk/exproj/mainprogram.cbl"
        version="123">
      <measuredvalue metricname="loc" value="77" />
      <measuredvalue metricname="avgnestdepth" value="3.33" />
      <measuredvalue metricname="compclass" value="medium" />
      <measuredvalue metricname="y2kcompl" value="false" />
    </measuredobject>

    <measuredvalue metricname="loc" value="242" />
    <measuredvalue metricname="avgnestdepth" value="3.04" />
    <measuredvalue metricname="compclass" value="medium" />
    <measuredvalue metricname="y2kcompl" value="false" />
  </measuredobject>

  <measuredvalue metricname="loc" value="242" />
  <measuredvalue metricname="avgnestdepth" value="3.04" />
  <measuredvalue metricname="compclass" value="medium" />
  <measuredvalue metricname="y2kcompl" value="false" />
</project>
```

Listing 5: Beispiel einer XML-Struktur mit Metrikdaten

```
<?xml version="1.0" encoding="utf-8"?>

<xs:schema attributeFormDefault="unqualified" elementFormDefault="qualified"
    xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xs:element name="project">
    <xs:complexType>
      <xs:sequence minOccurs ="0" maxOccurs ="unbounded">
        <xs:element name="metricdefinitions" type="metricdefinitionsType" maxOccurs="1"
            minOccurs="1" />
        <xs:element name="measuredobject" type="measuredobjectType" minOccurs="0"
            maxOccurs="unbounded" />
        <xs:element name="measuredvalue" type="measuredvalueType" minOccurs="0"
            maxOccurs="unbounded" />
      </xs:sequence>
      <xs:attribute name="name" type="xs:string" use="optional" />
    </xs:complexType>
  </xs:element>

  <xs:complexType name="metricdefinitionsType">
    <xs:sequence>
      <xs:element name="metric" minOccurs="0" maxOccurs="unbounded">
        <xs:complexType>
          <xs:attribute name="name" type="xs:string" />
          <xs:attribute name="description" type="xs:string" />
          <xs:attribute name="unit" type="xs:string" />
        </xs:complexType>
      </xs:element>
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name="measuredobjectType">
    <xs:sequence>
      <xs:element name="measuredobject" type="measuredobjectType" minOccurs="0"
          maxOccurs="unbounded" />
      <xs:element name="measuredvalue" type="measuredvalueType" minOccurs="0"
          maxOccurs="unbounded" />
    </xs:sequence>
    <xs:attribute name="type" type="xs:string" use="required" />
    <xs:attribute name="reference" type="xs:string" use="required" />
    <xs:attribute name="version" type="xs:string" use="optional" />
  </xs:complexType>

  <xs:complexType name="measuredvalueType">
    <xs:attribute name="metricname" type="xs:string" use="required" />
    <xs:attribute name="value" type="xs:string" use="required" />
  </xs:complexType>

</xs:schema>
```

Listing 6: XML-Schema zur Speicherung von Metrikdaten

5.4 Datenformate bestehender Tools

Die Entwicklung eines eigenen Schemas wurde gewählt, weil die Schemata der existierenden Tools einige wesentliche Nachteile besitzen. Im Anhang C findet sich ein XML-Beispiel (Listing

12) und das XML-Schema (Listing 13) des Eclipse Metrics Plug-Ins[76]. Dieses war als Anregung und Vergleichsmöglichkeit bei der Entwicklung hilfreich, besitzt aber nichtsdestotrotz einige Nachteile:

- Einige Bestandteile sind Java-spezifisch (etwa das Attribut *package*), im Idealfall sollte das Schema aber für beliebige Sprachen eingesetzt werden können.
- Die Speicherung von Mittel-, Maximal- und mittlerem Abweichungswert sollte der klaren Trennung und der Einfachheit halber nicht in einer, sondern besser in drei eigenen Metriken erfolgen.
- Das Schema könnte generischer und kompakter sein.
- Zu jeder Metrik wird sowohl ein Kürzel als auch die komplette Bezeichnung gespeichert. Dies stellt eine Redundanz dar und vermindert die Lesbarkeit.

Im Anhang C wird darüber hinaus in Listing 14 auch ein Beispiel des ebenfalls XML-basierten Datenformats des XMLFileWriter-Processors von ConQAT gezeigt. Die Struktur dieses Datenformates ist dem hier entwickelten insofern ähnlich, als dass die Struktur möglichst einfach gehalten wurde und die Verschachtelung bzw. Aggregation von Messobjekten ebenfalls abgebildet wird. Das von ConQAT gewählte Format besitzt aber folgende Nachteile gegenüber dem entwickelten:

- Eine Speicherung der Version des Messobjektes erfolgt nicht.
- Eine Speicherung der Metrikdefinition mit Bezeichner, Beschreibung und Einheit erfolgt nicht.
- Eine Speicherung des Typs des Messobjektes erfolgt nicht.

[76] `http://metrics.sourceforge.net/`

6 Entwurf der Software

In Kapitel 4 wurden die Möglichkeiten zur automatisierten Ausführung der Metrik-Tools beschrieben und in Kapitel 5 wurde das Datenformat für die Metrikdaten spezifiziert. In diesem Kapitel wird der darauf aufbauende Entwurf der Software vorgestellt. Dazu werden zunächst die Anforderungen vollständig zusammengetragen und darauf aufbauend die Abläufe sowie die benötigten Funktionseinheiten des Programms entworfen.

6.1 Anforderungen an die Software

Die Anforderungen ergeben sich vor allem aus der Zielsetzung der Arbeit sowie der Umgebung und den Randbedingungen, unter welchen die Software ausgeführt werden soll. Darüber hinaus ergeben sich auch Anforderungen aus den verwendeten Metrik-Tools und dem bereits spezifizierten Datenformat zur Speicherung der Metrikwerte.

1. Die Ausführung der Software soll eingebettet in die Entwicklungswerkzeuge der Debeka möglich sein:
   (a) Die Ausführung des Programms muss als Konsolenanwendung ohne GUI erfolgen.
   (b) Der vollständige Ablauf des Programms ohne manuellen Eingriff des Benutzers muss möglich sein.
   (c) Die Steuerung und Konfiguration des Programms muss über Parameter oder Konfigurationsdateien möglich sein.
   (d) Die Ausführung der Metrik-Tools muss ebenfalls automatisch erfolgen.
   (e) Für gegebenenfalls auftretende Fehler oder Statusmeldungen sollte eine Loggingfunktion vorhanden sein.
2. Das Programm muss alle Metrik-Tools aus der engeren Auswahl (SofAudit, ConQAT) gleichermaßen unterstützen.
3. Das Programm soll die Metrikdaten im spezifizierten Datenformat und zusätzlich im CSV-Format speichern können:
   (a) Die berechneten Metrikwerte müssen aus den Datenformaten der Tools extrahiert werden.
   (b) Die berechneten Metrikwerte müssen gegebenenfalls transformiert und im jeweiligen Datenformat gespeichert werden.
4. Das Programm sollte möglichst einfach erweiterbar sein:
   (a) Die Einbindung zusätzlicher Metrik-Tools zu einem späteren Zeitpunkt soll möglichst einfach realisierbar sein.

(b) Die Unterstützung weiterer Metriken soll möglichst einfach realisierbar sein.

(c) Der Export der Daten in andere Datenformate soll möglichst einfach realisierbar sein.

6.2 Ausführungsablauf und Funktionen

Aus den Anforderungen ergibt sich der folgende Ablauf der Programmausführung:

1. Auswertung der übergebenen Parameter:
   (a) Zu analysierende Dateien oder Verzeichnisse
   (b) Zielpfad für die Speicherung der Ergebniswerte
   (c) Die zu verwendende Konfigurationsdatei
   (d) Der Projektname (optional)
2. Einlesen der Einstellungen aus der Konfigurationsdatei:
   (a) Zu verwendender Präprozessor
   (b) Arbeitsverzeichnis des Programms
   (c) Verhaltensweise im Fehlerfall
   (d) Als Codedateien zu verarbeitende Dateitypen
   (e) Zu ignorierende Verzeichnisse (z.B. für SVN-spezifische Ordner)
   (f) Zu verwendende Metrik-Tools
   (g) Zu verwendende Datenformate für den Metrikdatenexport
3. Einlesen der Einstellungen für die einzelnen Metrik-Tools aus jeweils eigenen, spezifischen Konfigurationsdateien
4. Vorbereitung der zu analysierenden Objekte:
   (a) Erfassung der Verzeichnisse und Codedateien
   (b) Kopieren der Codedateien inklusive ihrer Verzeichnisstruktur in einen temporären Unterordner des Arbeitsverzeichnisses
   (c) Anwenden des Präprozessors auf die Codedateien (falls gegeben)
5. Ausführen der einzelnen Metrik-Tools (nach einander), jeweils:
   (a) Durchführen der notwendigen Vorbereitungsschritte (z.B. Erzeugen toolspezifischer Dateien, Verzeichnisse, etc.)
   (b) Ausführen des eigentlichen Metrik-Tools

(c) Extraktion der vom Tool generierten Metrikdaten und Import in den internen Datenbestand des Programms

6. Ausführen des Exportes der Metrikdaten aus dem internen Datenbestand in die verschiedenen Datenformate (nacheinander)

7. Aufräumen des Arbeitsverzeichnisses:

   (a) Löschen der Kopien der Codedateien

   (b) Löschen der erzeugten toolspezifischen Eingabedateien

   (c) Löschen der von den Metrik-Tools erzeugten Ausgabedateien

6.3 Struktur des Hauptprogramms

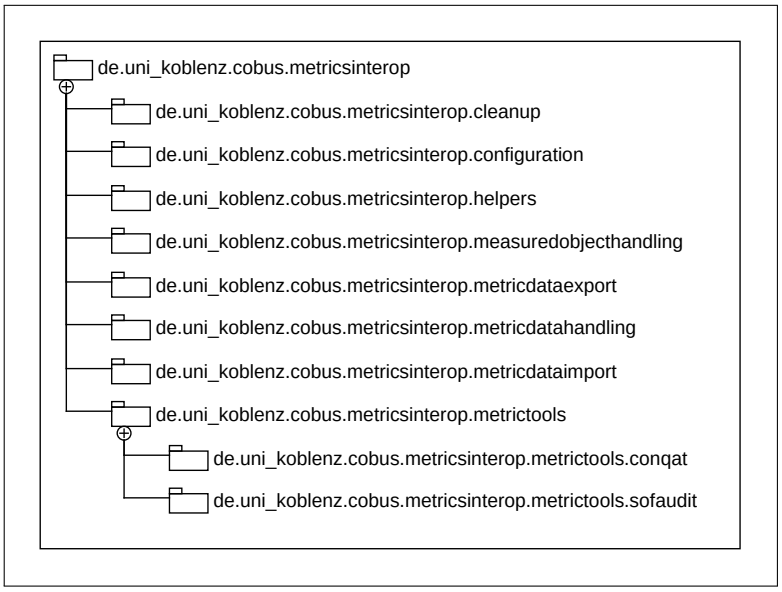

Abbildung 12: Die Packages des Programms.

Aus den Anforderungen und dem Programmablauf ist ersichtlich, dass das Programm in verschiedene Funktionalitäten teilbar ist. Da diese selbst aber zu umfangreich für einzelne Klassen sind, werden die Funktionalitäten zunächst in einzelne Packages aufgeteilt. Abbildung 12 zeigt die verschiedenen Packages. Diese werden im Folgenden genauer beschrieben. Dabei wird für jedes Package auch ein Klassendiagramm gezeigt. Diese enthalten jeweils des Packages einschließlich aller Ableitungen, und Assozationen untereinander. Zusätzlich werden auch die Klassen von außerhalb des Packages gezeigt, zu welchen eine Assozation besteht, bei der die Klasse im Package eine von außerhalb verwendet. Der umgekehrte Fall wird nicht dargestellt. Um die Diagramme halbwegs übersichtlich zu halten, werden nur die Felder und Funktionen dargestellt, welche als *public* deklariert sind oder sich innerhalb des dargestellten Packages

befinden und als *protected* deklariert sind. Rückgabetypen und Parameter sind ebenfalls nicht dargestellt. Ein vollständiges Diagramm findet sich in Anhang E.

Da für die einzelnen Metrik-Tools jeweils sehr spezifische Arbeitsschritte zur Vorbereitung der Ausführung und zur Extraktion der Metrikdaten notwendig sind, wird die Funktionalität hierfür in ein jeweiliges Plug-In ausgelagert. Auch zum Export der Daten in verschiedene Formate sind spezifische Arbeitsschritte notwendig, so dass auch diese Funktionalität über Plug-Ins realisiert wird. Für beide Arten der Plug-Ins wird jeweils eine eigene Schnittstelle definiert, so dass neue Plug-Ins über den Klassennamen instanziiert werden können, ohne dass hierfür Modifikationen am Code des Hauptprogramms notwendig sind.

6.3.1 Package de.uni_koblenz.cobus.metricsinterop

Wie in Abbildung 12 bereits gezeigt, ist dieses Package das Basispackage in der Hierarchie und enthält alle weiteren Unterpackages. Das zugehörige Klassendiagramm ist in Abbildung 13 gezeigt. Das Package enthält die folgende Klasse:

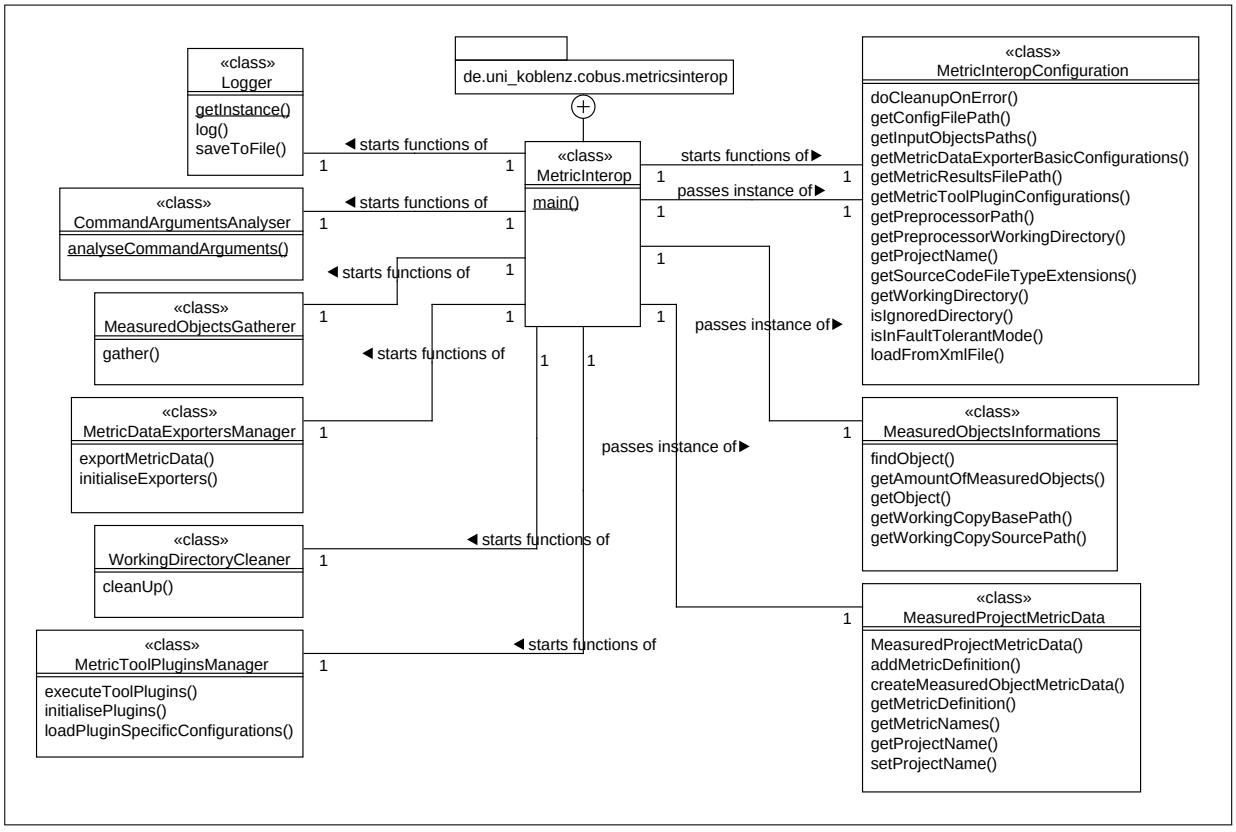

Abbildung 13: Klassendiagramm: Package *de.uni_koblenz.cobus.metricsinterop*.

MetricInterop. Diese Klasse enthält die *main()*-Methode zum Starten des Programms. Die Funktionalität dieser Klasse beschränkt sich auf die Instanziierung aller benötigten Klassen für die eigentliche Programmfunktionalität und das Aufrufen der entsprechenden Methoden.

6.3.2 Package de.uni_koblenz.cobus.metricsinterop.configuration

Dieses Package beinhaltet alle für die Einstellungen des Hauptprogramms benötigte Funktionalitäten. Das zugehörige Klassendiagramm ist in Abbildung 14 gezeigt. Das Package enthält folgende Klassen:

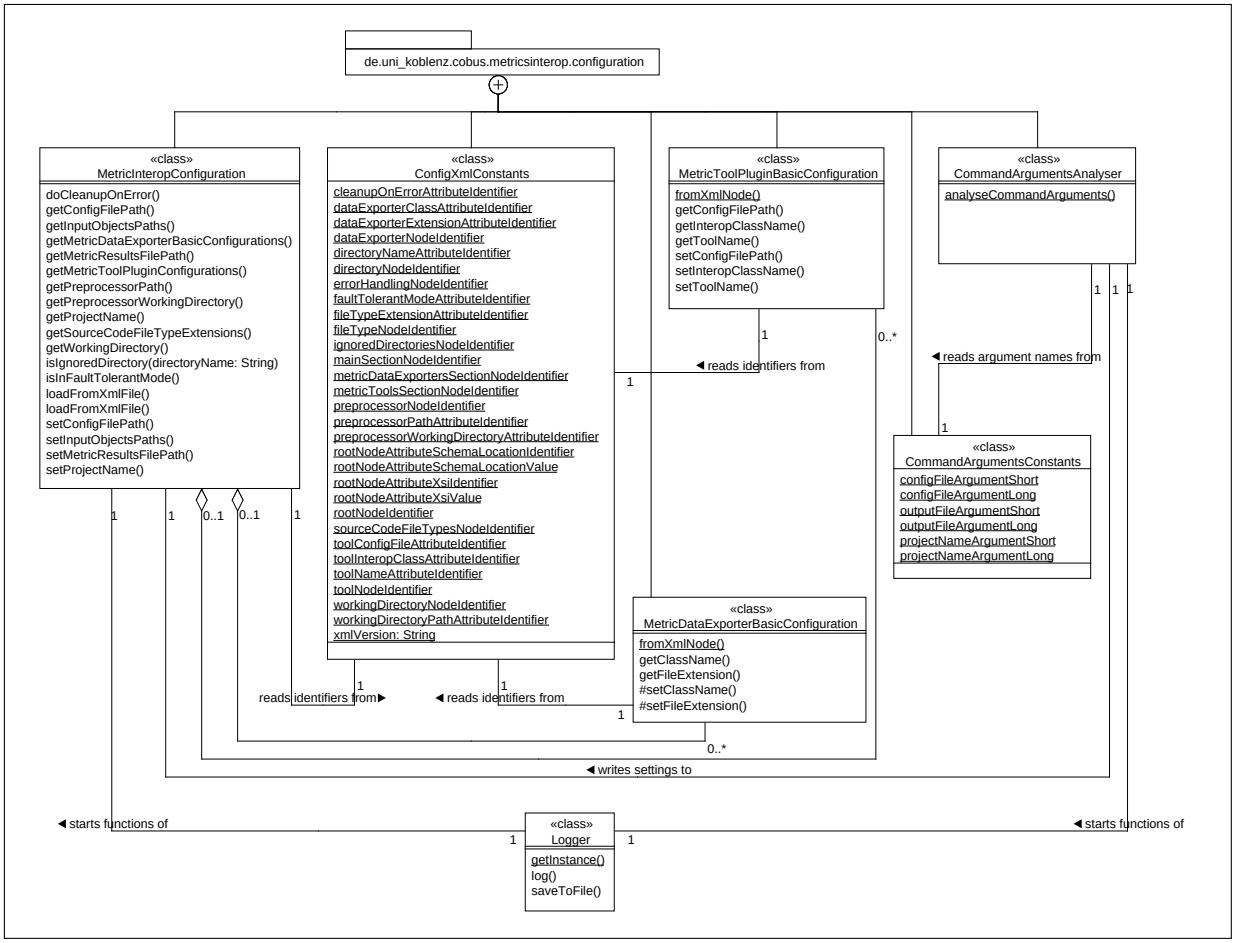

Abbildung 14: Klassendiagramm: Package *de.uni_koblenz.cobus.metricsinterop.configuration*.

MetricInteropConfiguration. Diese Klasse enhält die Funktionalität zum Vorhalten und Abfragen der Programmeinstellungen zur Laufzeit. Des Weiteren kann die Konfiguration aus einer XML-Datei eingelesen werden.

MetricToolPluginBasicConfiguration. Diese Klasse enthält die Funktionalität zum Vorhalten und Abfragen der Basiseinstellungen eines einzelnen Metrik-Tool-Plug-Ins zur Laufzeit. Darüber hinaus können die entsprechenden Einstellungen aus einem entsprechenden XML-Element eingelesen werden, welche Teil der Konfigurationsdatei sind. Instanzen dieser Klasse werden von *MetricInteropConfiguration* vorgehalten und benutzt.

MetricDataExporterBasicConfiguration. Diese Klasse enthält die Funktionalität zum Vorhalten und Abfragen der Basiseinstellungen eines einzelnen Metrikdatenexport-Plug-Ins zur Laufzeit. Darüber hinaus können die entsprechenden Einstellungen aus einem entsprechenden XML-Element eingelesen werden, welche Teil der Konfigurationsdatei sind. Instanzen dieser Klasse werden von *MetricInteropConfiguration* vorgehalten und benutzt.

ConfigXmlConstants. Diese Klasse enthält alle in der Konfigurationsdatei möglichen Namen der XML-Elemente und -Attribute als Konstanten. Sie wird von *MetricInteropConfiguration*, *MetricToolPluginBasicConfiguration* und *MetricDataExporterBasicConfiguration* verwendet. Hierdurch werden Änderungen am XML-Schema der Konfigurationsdatei vereinfacht.

CommandArgumentsAnalyser. Diese Klasse enthält die Funktionalität zum Überprüfen der Argumente, welche dem Programm beim Aufruf übergeben werden. Sind diese gültig, so wird eine neue Instanz von *MetricInteropConfiguration* erzeugt und darin die per Argument übergebenen Einstellungen gespeichert.

CommandArgumentsConstants. Diese Klasse enthält alle als Optionsbezeichner gültigen Aufrufargumente als Konstanten. Hierdurch werden Änderungen erleichtert.

6.3.3 Package de.uni_koblenz.cobus.metricsinterop.measuredobjecthandling

Dieses Package beinhaltet alle Funktionalitäten, um die zu analysierenden Objekte vorzubereiten, Informationen über diese zu sammeln und zur Laufzeit vorzuhalten. Das zugehörige Klassendiagramm ist in Abbildung 15 gezeigt. Das Package enhält folgende Klassen:

MeasuredObjectsInformations. Diese Klasse enthält Funktionalität zum Vorhalten und Abfragen von Informationen über alle zu analysierenden Objekte.

MeasuredObjectInformations. Diese Klasse enthält die Funktionalität zum Vorhalten und Abfragen von Informationen über ein einzelnes zu analysierendes Objekt. Dazu gehören der Pfad der Originalversion, der Pfad der Kopie im Arbeitsverzeichnis des Programms, die Version und der SVN-Pfad. Diese Informationen müssen vorgehalten werden, um bei der Extraktion

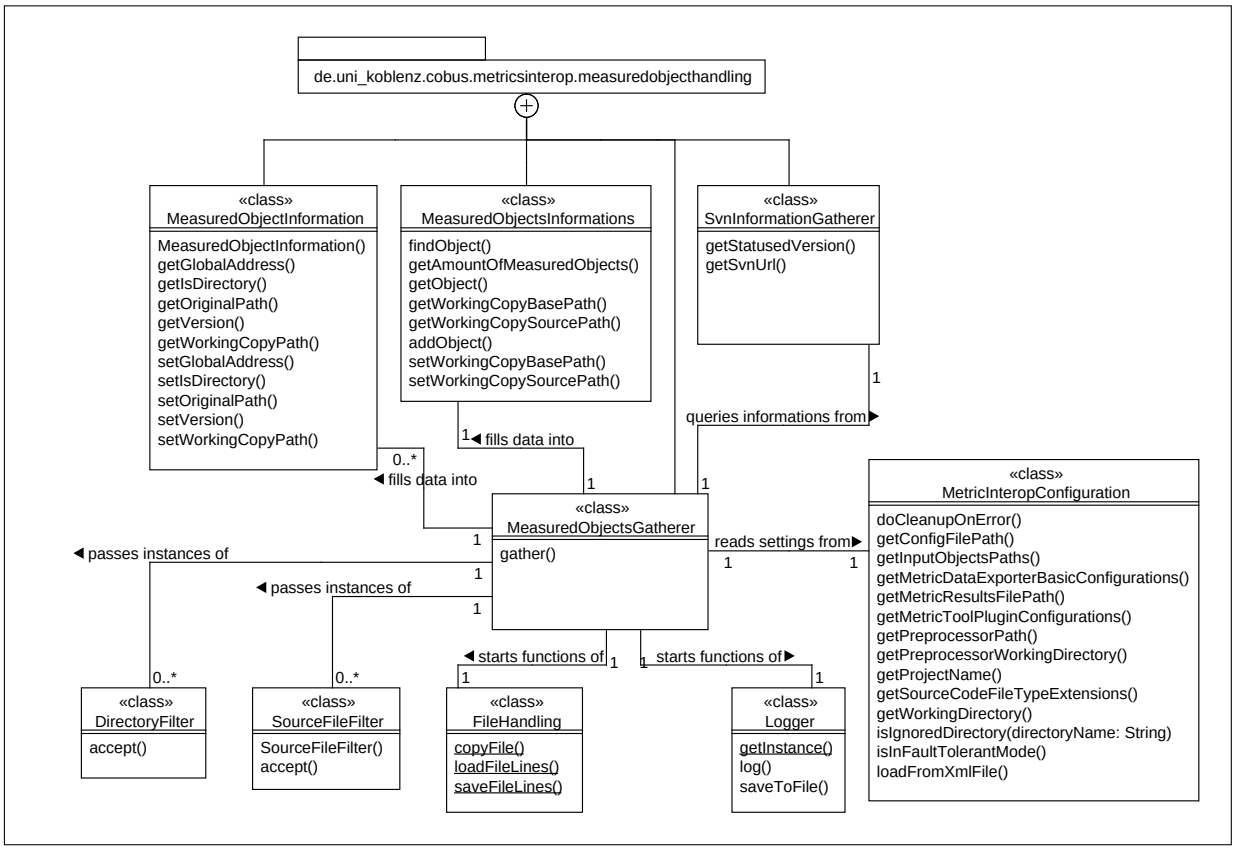

Abbildung 15: Klassendiagramm: Package *de.uni_koblenz.cobus.metricsinterop.measuredobjecthandling*.

der Metrikdaten – welche ja selbst nur den Bezug zur Arbeitskopie besitzen – den Bezug zum Original des zu analysierdenden Objektes wiederherstellen zu können.

MeasuredObjectsGatherer. Diese Klasse enthält die Funktionalität zum Generieren der Daten einer *MeasuredObjectsInformations*-Instanz. Dazu werden alle in einer *MetricInteropConfiguration* gespeicherten Objekte verarbeitet. Neben dem Sammeln von Informationen über die Objekte werden dabei auch die Kopien in einem neuen Unterordner im Arbeitsverzeichnis des Programms angelegt und, sofern in der Konfiguration angegeben, der Präprozessor auf diese angewendet.

SvnInformationGatherer. Diese Klasse enthält die Funktionalität zum Abfragen der SVN-Version und des SVN-Pfades eines zu analysierenden Objektes und wird entsprechend von *MeasuredObjectsGatherer* verwendet. Hierzu wird die SVNKit-Komponente[77] benutzt.

[77] SVNKit (`http://svnkit.com/`) ist quelloffen unter der TMate-Lizenz (`http://svnkit.com/license.html`) verfügbar.

6.3.4 Package de.uni_koblenz.cobus.metricsinterop.metricdatahandling

Dieses Package beinhaltet alle Funktionalitäten, um Metrikdaten zur Laufzeit vorzuhalten. Das zugehörige Klassendiagramm ist in Abbildung 16 gezeigt. Das Package enhält folgende Klassen:

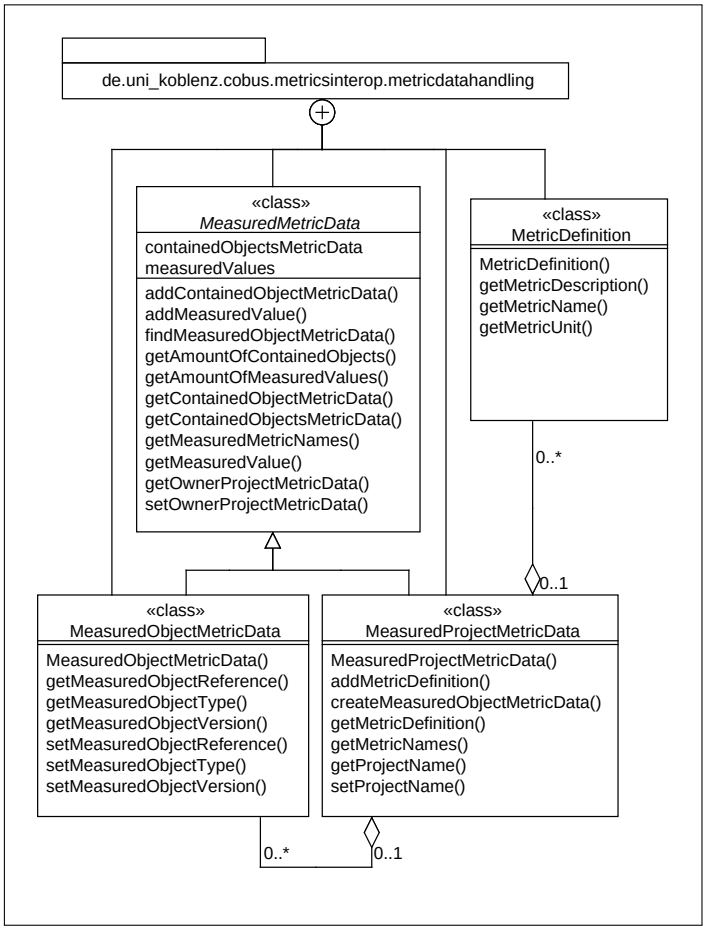

Abbildung 16: Klassendiagramm: Package *de.uni_koblenz.cobus.metricsinterop.metricdatahandling*.

MeasuredMetricData. Diese abstrakte Klasse beinhaltet grundlegende Funktionalität zum Vorhalten von Metrikdaten. Die enthaltene Funktionalität wird sowohl für den Gesamtmetrikdatenbestand als auch für den Metrikdatenbestand einzelner Objekte benötigt.

MeasuredObjectMetricData. Diese Klasse ist von *MeasuredMetricData* abgeleitet und beinhaltet Funktionalität zum Vorhalten von Metrikdaten eines einzelnen Objektes. Zusätzlich können in einer Instanz dieser Klasse auch weitere Instanzen verwaltet werden. Dies bildet den Fall ab, wenn Metrikdaten von weiteren Objekten durch das Objekt der aktuellen Instanz aggregiert werden (etwa Metrikdaten über ein Verzeichnis, welche Metrikdaten über mehrere

darin enthaltene Dateien aggregieren). Objekte dieser Klasse sind immer einem bestimmten Gesamtdatenbestand in Form eines *MeasuredProjectMetricData*-Objektes zugeordnet. Dies kann entweder durch direkte Zugehörigkeit oder indirekt über aggregierende Objekte der Fall sein. Um die Persistenz des Gesamtdatenbestandes sicherzustellen, kann diese Klasse nur über eine Factory-Methode in *MeasuredProjectMetricData* instanziiert werden.

MetricDefinition. Diese Klasse enthält Funktionalität zum Vorhalten der Definitionen der im Datenbestand enthaltenen Metriken. Dies ist notwendig, um zu jeder Metrik über ihren Bezeichner auch die Beschreibung und die Einheit abfragen zu können.

MeasuredProjectMetricData. Diese Klasse ist von *MeasuredMetricData* abgeleitet und enthält die Funktionalität zum Vorhalten aller Metrikdaten. Neben Messdaten über alle im Bestand aggregierten Objekte können hierzu auch Instanzen der *MeasuredObjectMetricData*-Klasse verwaltet werden, welche die Daten der einzelnen aggregierten Objekte beinhalten. Des Weiteren wird in dieser Klasse eine Menge von *MetricDefinition*-Objekten vorgehalten, welche die Definitionen aller in den Daten vorkommenden Metriken darstellt. Die entsprechenden Definitionen müssen einem *MeasuredProjectMetricData*-Objekt hinzugefügt werden, bevor zum jeweiligen Metrikbezeichner gehörige Metrikdaten zum Objekt selbst oder einem der zugehörigen *MeasuredObjectMetricData*-Objekte hinzugefügt werden können.

6.3.5 Package de.uni_koblenz.cobus.metricsinterop.metricdataimport

Dieses Package enthält die folgende Klasse, das zugehörige Klassendiagramm ist in Abbildung 17 gezeigt:

MetricDataXmlImporter. Diese Klasse enthält Funktionalität zum Laden eines Metrikdatenbestandes aus einer XML-Datei des im Rahmen dieser Arbeit entwickelten Schemas. Sie ist zunächst zum Testen der Funktionalität anderer Komponenten entstanden und wird deshalb im fertigen System nicht verwendet. Da die Importfunktionalität vollständig und es denkbar ist, dass diese zu einem späteren Zeitpunkt noch einmal benötigt wird, bleibt die Klasse dennoch bestehen.

6.3.6 Package de.uni_koblenz.cobus.metricsinterop.metricdataexport

Dieses Package beinhaltet die Funktionalität zum Export von Metrikdaten in verschiedene Formate und zum Verwalten der Export-Plug-Ins. Das zugehörige Klassendiagramm ist in Abbildung 18 gezeigt. Das Package enhält folgende Typen:

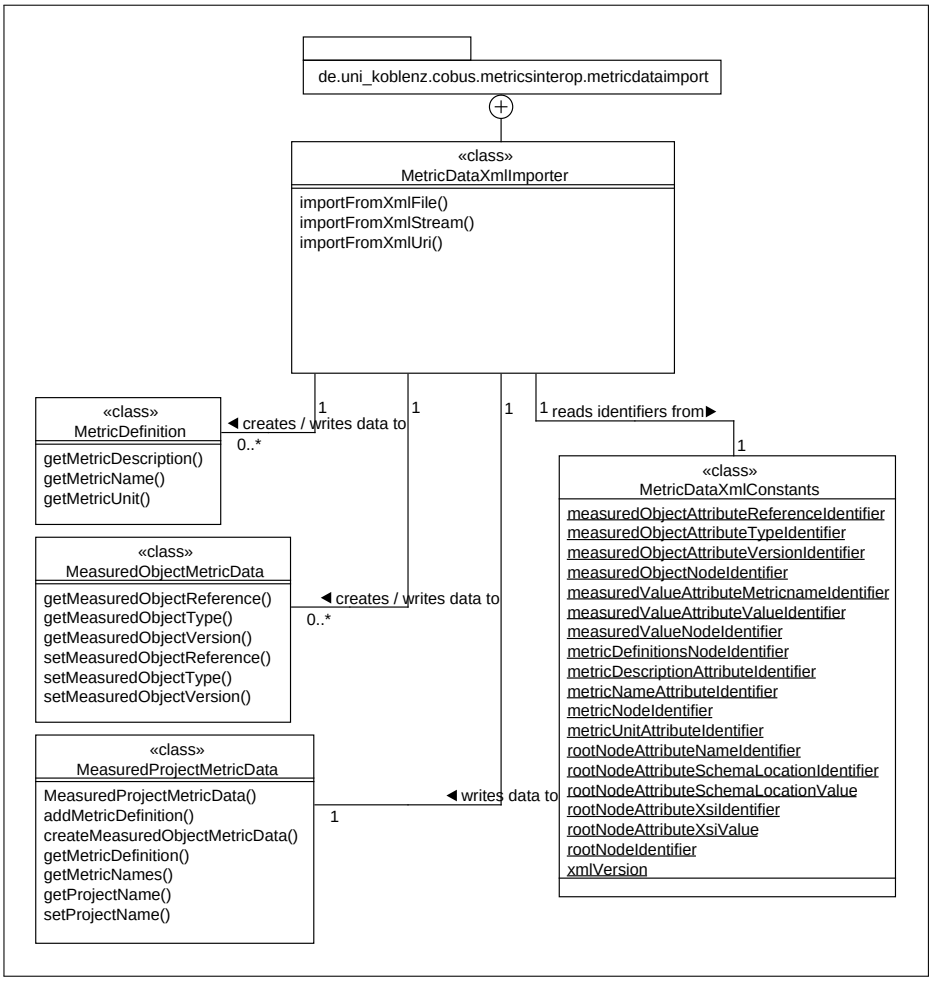

Abbildung 17: Klassendiagramm: Package *de.uni_koblenz.cobus.metricsinterop.metricdataimport*.

MetricDataExporter. Diese Schnittstelle definiert die Funktionen, welche ein Plug-In zum Export von Metrikdaten implementieren muss. Diese wird später in Abschnitt 6.3.11 detailliert beschrieben.

MetricDataExportersManager. Diese Klasse enthält die Funktionalität zum Instanziieren und Vorhalten der in den Einstellungen angegebenen Export-Plug-Ins und dem Aufruf ihrer Funktionen.

MetricDataCsvExporter. Diese Klasse implementiert die Schnittstelle *MetricDataExporter* und ist somit ein Plug-In zum Metrikdatenexport. Mit der enthaltenen Funktionalität kann der Metrikdatenbestand in eine CSV-Datei[78] exportiert werden.

[78] Comma-separated-value, ein einfaches Tabellenformat

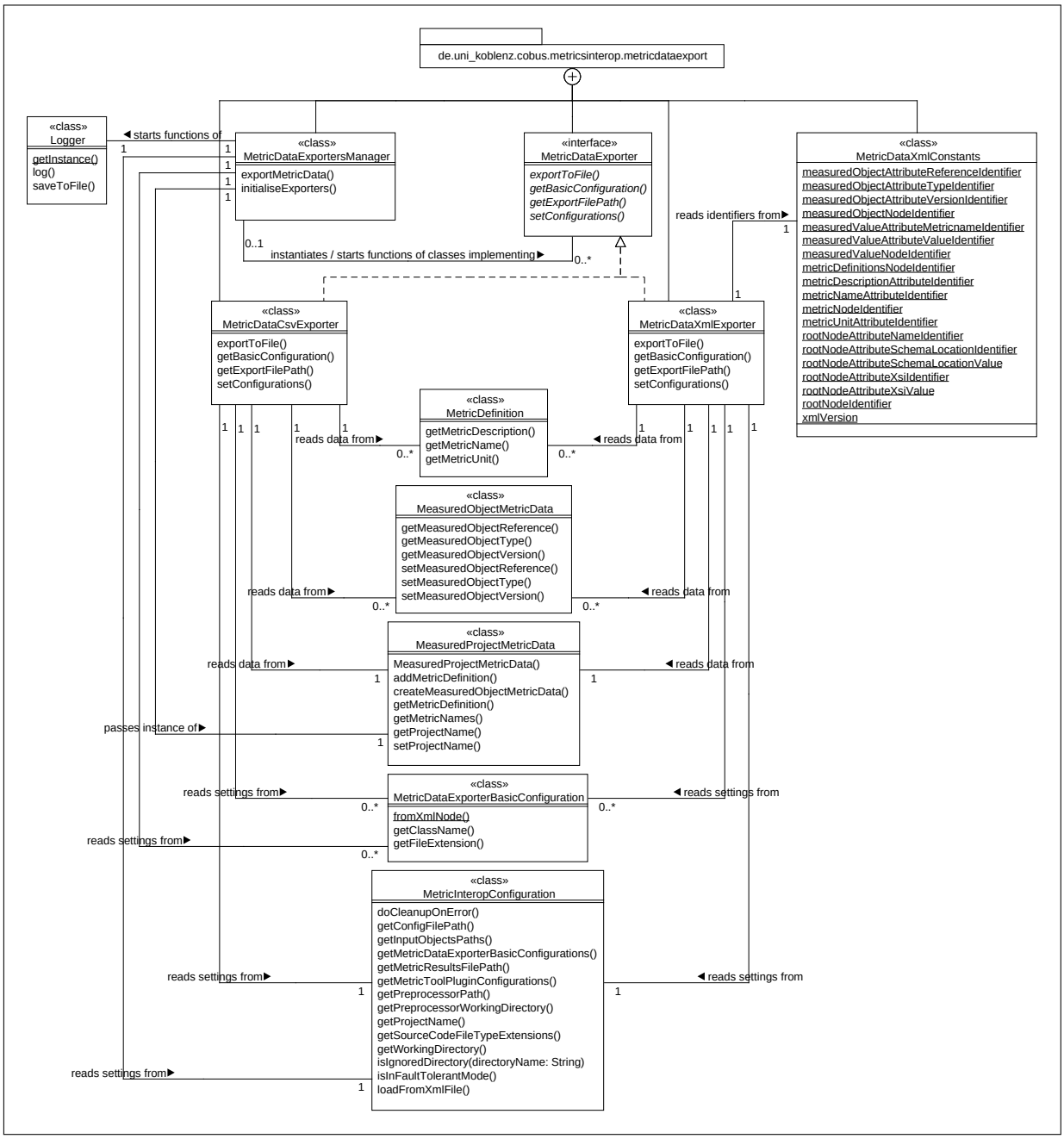

Abbildung 18: Klassendiagramm: Package *de.uni_koblenz.cobus.metricsinterop.metricdataexport*.

MetricDataXmlExporter. Diese Klasse implementiert die Schnittstelle *MetricDataExporter* und ist somit ein Plug-In zum Metrikdatenexport. Mit der enthaltenen Funktionalität kann der Metrikdatenbestand in eine XML-Datei mit dem im Rahmen dieser Arbeit entwickelten Schema exportiert werden.

MetricDataXmlConstants. Diese Klasse enthält alle im XML-Schema vorhandenen Namen der XML-Elemente und -Attribute als Konstanten. Sie wird von *MetricDataXmlExporter* und *MetricDataXmlImporter* und *MetricDataExporterBasicConfiguration* verwendet. Hierdurch werden Änderungen am XML-Schema vereinfacht, falls diese einmal notwendig sein sollten.

6.3.7 Package de.uni_koblenz.cobus.metricsinterop.metrictools

Dieses Package beinhaltet die Funktionalitäten zum Verwalten der Plug-Ins für die Metrik-Tools. Es enthält die im Folgenden beschriebenen Typen. Das zugehörige Klassendiagramm ist in Abbildung 19 gezeigt. Da die realisierten Plug-Ins umfangreich sind, liegen diese jeweils in einem eigenen Package und werden einzeln in den Abschnitten 6.4 und 6.5 beschrieben.

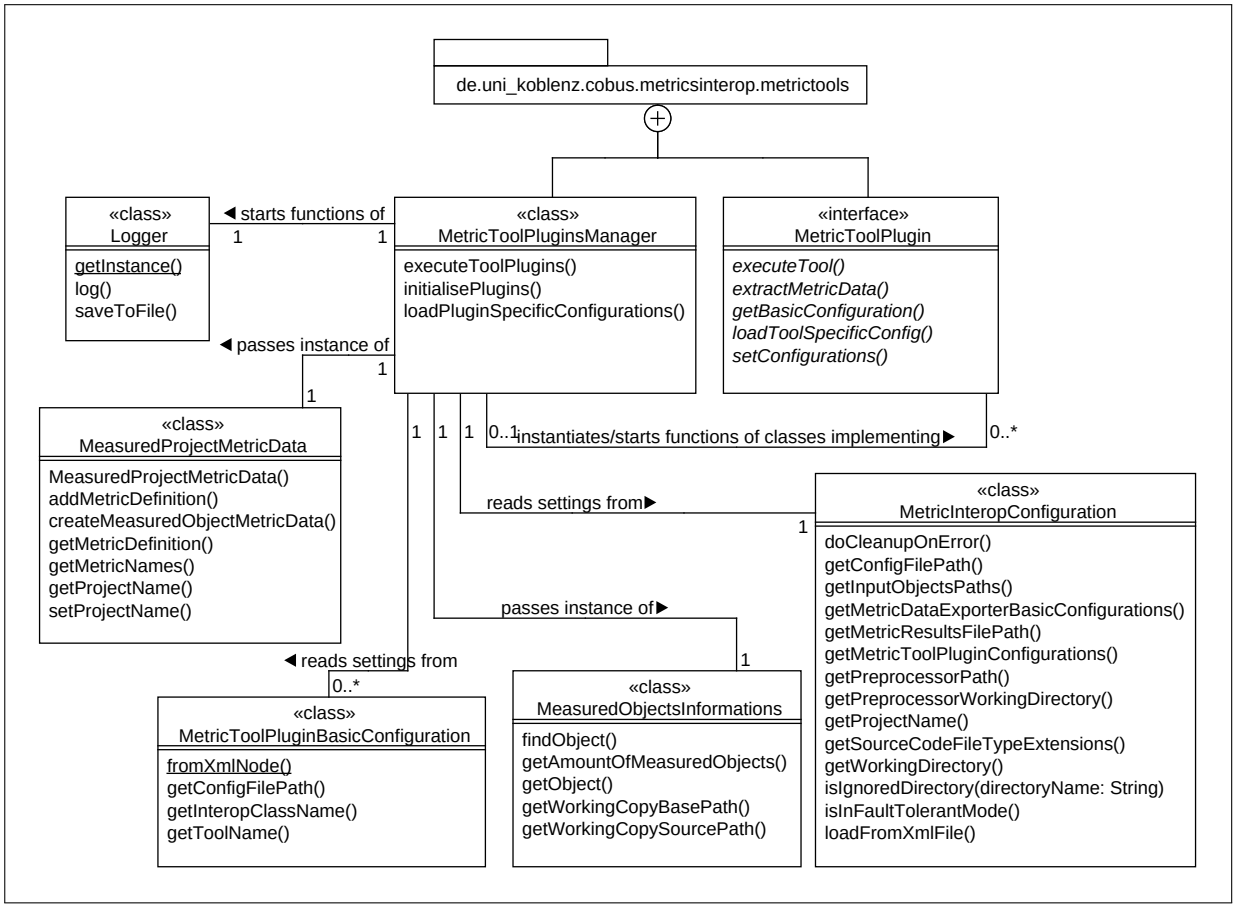

Abbildung 19: Klassendiagramm: Package *de.uni_koblenz.cobus.metricsinterop.metrictools*.

MetricToolPlugin. Diese Schnittstelle definiert die Funktionen, welche ein Plug-In für ein Metrik-Tool implementieren muss. Die Schnittstelle wird später in Abschnitt 6.3.10 detailliert beschrieben.

MetricToolPluginsManager. Diese Klasse enthält die Funktionalität zum Instanziieren und Vorhalten der in den Einstellungen angegebenen Plug-Ins für die Metrik-Tools sowie dem Aufruf ihrer Funktionen.

6.3.8 Package de.uni_koblenz.cobus.metricsinterop.cleanup

Dieses Package beinhaltet die folgende Klasse, das zugehörige Klassendiagramm ist in Abbildung 20 gezeigt:

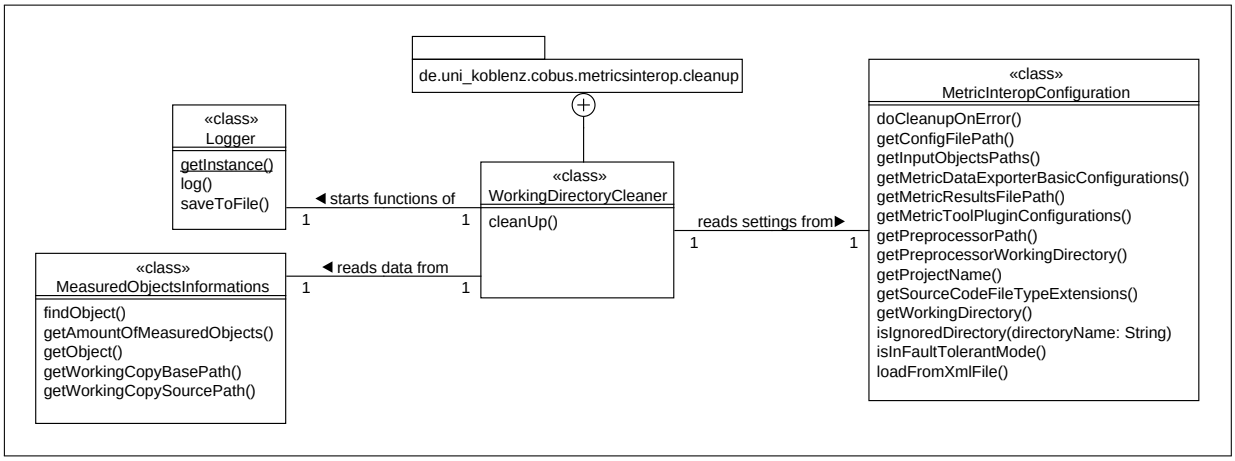

Abbildung 20: Klassendiagramm: Package *de.uni_koblenz.cobus.metricsinterop.cleanup*.

WorkingDirectoryCleaner. Diese Klasse enthält die Funktionalität zum Aufräumen des Arbeitsverzeichnisses. Dabei handelt es sich im Wesentlichen das Löschen aller im Programmablauf angelegten Dateien und Verzeichnisse.

6.3.9 Package de.uni_koblenz.cobus.metricsinterop.helpers

Dieses Package beinhaltet Funktionalitäten, welche an verschiedenen Stellen von den anderen Komponenten benutzt wird. Das zugehörige Klassendiagramm ist in Abbildung 21 gezeigt. Das Package enthält die folgenden Klassen:

DirectoryFilter. Diese Klasse wird zum Filtern von Verzeichnissen mit der *File.listFiles()*-Funktion verwendet.

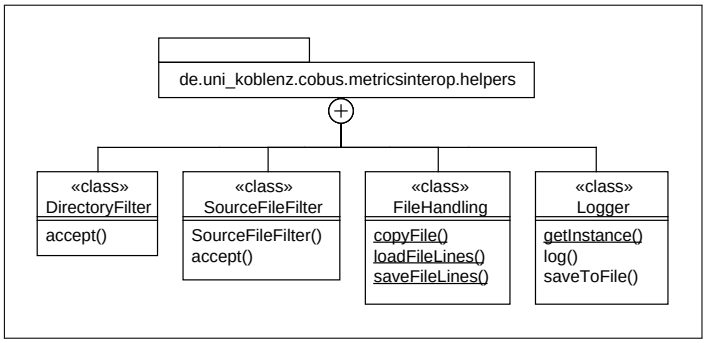

Abbildung 21: Klassendiagramm: Package *de.uni_koblenz.cobus.metricsinterop.helpers*.

SourceFileFilter. Diese Klasse wird zum Filtern von Quellcodedateien mit der *File.listFiles()*-Funktion verwendet. Die Dateitypen sind dabei konfigurierbar.

FileHandling. Diese Hilfsklasse enthält verschiedene Funktionen für Dateien, etwa zum Einlesen aller Zeilen einer Textdatei oder zum Kopieren.

Logger. Diese Klasse enthält die für dieses Programm benötigte Loggingfunktionalität. Da während der gesamten Laufzeit nur eine einzige Instanz benötigt wird, ist diese Klasse nach dem Singleton-Entwurfsmuster implementiert.

6.3.10 Schnittstelle der Plug-Ins für die Metrik-Tools

Um die Verwendung neuer Metritools möglichst einfach und ohne eine Änderung am Code des eigentlichen Programms zu ermöglichen, wurde eine einheitliche Schnittstelle *MetricToolPlugin* für Tool-Plug-Ins definiert. Sofern ein Plug-In diese Schnittstelle implementiert, kann ein *MetricToolPluginsManager*-Objekt dieses über den in den Einstellungen angegebenen Klassennamen instanziieren und verwenden. Abbildung 22 zeigt die Schnittstelle als Diagramm. Die darin definierten Funktionen müssen wie folgt implementiert werden:

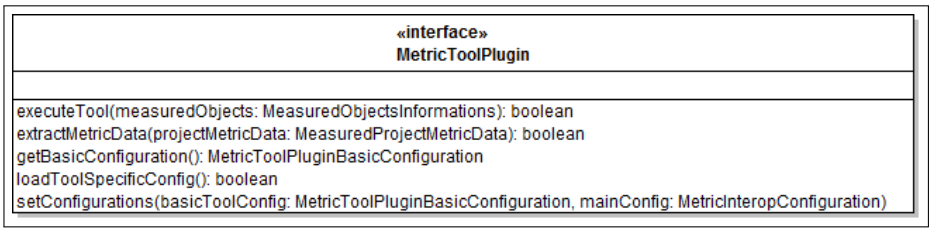

Abbildung 22: Die Schnittstelle für Metrik-Tool-Plug-Ins.

setConfigurations(basicToolConfig: MetricToolPluginBasicConfiguration, mainConfig: MetricInteropConfiguration). Diese Funktion wird vom *MetricToolPluginsManager* direkt nach dem Instanziieren eines Plug-Ins aufgerufen. Die dabei übergebenen Objekte sind die zum Plug-In gehörende Basiskonfiguration und die allgemeine Programmkonfiguration. Beide müssen als interne Referenz gespeichert werden. Die Funktion hat keinen Rückgabewert.

getBasicConfiguration(): MetricToolPluginBasicConfiguration. Diese Funktion muss die intern gespeicherte Referenz auf die zum Plug-In gehörende *MetricToolPluginBasicConfiguration*-Instanz zurückgeben. Falls die Referenz noch nicht gesetzt wurde, muss *null* zurückgegeben werden.

loadToolSpecificConfig(): boolean. Diese Funktion lädt die für das Plug-In spezifischen Einstellungen aus der Konfigurationsdatei. Als Pfad zu dieser Datei muss der in der intern referenzierten *MetricToolPluginBasicConfiguration* angegebene Werte verwendet werden. Nach erfolgreichem Laden muss die Funktion *true* zurückgeben, im Fehlerfall *false*. Falls das Plug-In keine Konfigurationsdatei benötigt, muss ebenfalls *true* zurückgegeben werden.

executeTool(measuredObjects: MeasuredObjectsInformations): boolean. Diese Funktion startet zunächst alle eventuell zur Vorbereitung notwendigen Schritte und anschließend die Ausführung des eigentlichen Metrik-Tools. Aus dem übergebenen *MeasuredObjectsInformations*-Objekt müssen alle für das Tool geeigneten Objekte ausgewählt werden (z.B. nur die Dateien, falls das Tool keine Verzeichnisse verarbeiten kann). Die Ausführung muss synchron erfolgen, der Rücksprung aus der Funktion darf also erst dann erfolgen, wenn das Metrik-Tool beendet ist. Nach erfolgreicher Ausführung muss die Funktion *true* zurückgeben, im Fehlerfall *false*.

extractMetricData(projectMetricData: MeasuredProjectMetricData): boolean. Diese Funktion startet die Extraktion der Metrikdaten aus den generierten Daten des Metrik-Tools und importiert diese in das übergebene *MeasuredProjectMetricData*-Objekt. Vor dem eigentlichen Import müssen dem *MeasuredProjectMetricData*-Objekt zunächst die Definitionen der zu importierenden Metriken hinzugefügt werden. Ob bei dem Import nur eine Auswahl der von dem Tool generierten Metriken importiert werden soll, sollte über die spezifische Konfigurationsdatei des Plug-Ins einstellbar sein. Nach erfolgreichem Import muss die Funktion *true* zurückgeben, im Fehlerfall *false*.

6.3.11 Schnittstelle der Plug-Ins für den Metrikdatenexport

Um den Export von Metrikdaten möglichst einfach - auch ohne eine Änderung am Code des eigentlichen Programms - um neue Dateiformate erweitern zu können, wurde eine einheitliche Schnittstelle *MetricDataExporter* für entsprechende Plug-Ins definiert. Sofern ein Plug-In

diese Schnittstelle implementiert, kann die *MetricDataExportersManager*-Klasse dieses über den in den Einstellungen angegebenen Klassennamen instanziieren und verwenden. Abbildung 23 zeigt die Schnittstelle als Diagramm. Die darin definierten Funktionen müssen wie folgt implementiert werden:

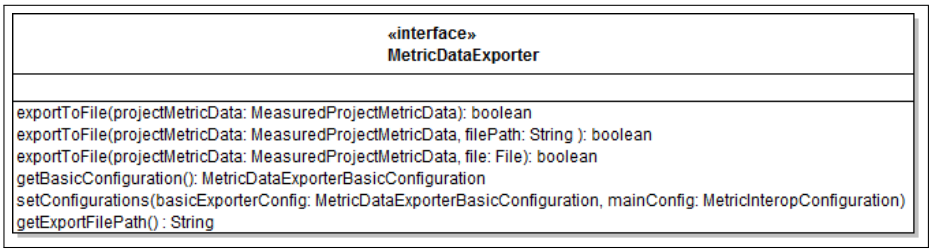

Abbildung 23: Die Schnittstelle für Plug-Ins zum Metrikdatenexport.

setConfigurations(basicExporterConfig: MetricDataExporterBasicConfiguration, mainConfig: MetricInteropConfiguration)). Diese Funktion wird vom *MetricDataExportersManager* direkt nach dem Instanziieren eines Plug-Ins aufgerufen. Die dabei übergebenen Objekte sind die zum Plug-In gehörende Basiskonfiguration und die allgemeine Programmkonfiguration. Beide müssen als interne Referenz gespeichert werden. Die Funktion hat keinen Rückgabewert.

getBasicConfiguration(): MetricDataExporterBasicConfiguration. Diese Funktion muss die intern gespeicherte Referenz auf die zum Plug-In gehörende *MetricDataExporterBasicConfiguration*-Instanz zurückgeben. Falls die Referenz noch nicht gesetzt wurde, muss *null* zurückgegeben werden.

getExportFilePath(): String. Diese Funktion muss aus dem in der intern referenzierten *MetricInteropConfiguration* gespeicherten Ausgabepfad und der in der intern referenzierten *MetricDataExporterBasicConfiguration* gespeicherten Dateierweiterung den kompletten Pfad der Exportdatei konkatenieren und zurückgeben. Falls die Referenzen nicht gesetzt wurden oder ein anderer Fehlerfall auftritt, muss *null* zurückgegeben werden.

exportToFile(projectMetricData: MeasuredProjectMetricData): boolean. Diese Funktion startet den Export der Daten aus der übergebenen *MeasuredProjectMetricData*-Instanz. Als Pfad für die Ausgabedatei muss dabei der Rückgabewert der eigenen *getExportFilePath()*-Funktion verwendet werden. Nach erfolgreichem Export muss die Funktion *true* zurückgeben, im Fehlerfall *false*.

exportToFile(projectMetricData: MeasuredProjectMetricData, filePath: String): boolean. Diese Funktion entspricht der zuvor genannten Variante bis auf die Ausnahme, dass der als *filePath* übergebene Pfad für die Ausgabedatei verwendet werden muss.

exportToFile(projectMetricData: MeasuredProjectMetricData, file: File): boolean. Diese Funktion entspricht der zuvor genannten Variante bis auf die Ausnahme, dass die als *file* referenzierte Datei für die Ausgabedatei verwendet werden muss.

6.4 Struktur des Plug-Ins für SofAudit

Aufgrund des Umfangs wurde das Plug-In für das Metrik-Tool SofAudit als eigenes Package *de.uni_koblenz.cobus.metricsinterop.metrictools.sofaudit* realisiert. Das zugehörige Klassendiagramm ist in Abbildung 24 gezeigt, enhält aufgrund des Umfangs aber nur die Assoziationen innerhalb des Packages. Das Package enthält folgende Klassen:

SofauditToolPluginFacade. Diese Klasse implementiert die *MetricToolPlugin*-Schnittstelle. Die Funktionalität dieser Klasse beschränkt sich auf die Instanziierung aller benötigten weiteren Klassen des Plug-Ins und das Aufrufen der entsprechenden Methoden. Somit handelt es sich hierbei um eine Implementierung des *Facade*-Entwurfsmusters.

SofauditConfiguration. Diese Klasse enhält die Funktionalität zum Vorhalten und Abfragen der spezifischen Einstellungen des Plug-Ins zur Laufzeit. Des Weiteren kann die Konfiguration aus einer XML-Datei eingelesen werden.

SofauditMetricConfiguration. Diese Klasse enthält die Funktionalität zum Vorhalten und Abfragen der Einstellungen einer einzelnen Metrik zur Laufzeit. Neben dem Bezeichner, der Einheit und der Beschreibung der Metrik (zur Verwendung beim Import in den internen Metrikdatenbestand des Hauptprogramms) wird auch die Zeichenfolge gespeichert, anhand welcher die Metrikwerte in den Ergebnisdateien von SofAudit identifiziert werden können. Darüber hinaus können die entsprechenden Einstellungen aus einem entsprechenden XML-Element eingelesen werden, welches ein Teil der spezifischen Konfigurationsdatei des Plug-Ins ist. Objekte dieser Klasse werden von *SofauditConfiguration* vorgehalten und benutzt.

SofauditConfigXmlConstants. Diese Klasse enthält alle in der toolspezifischen Konfigurationsdatei möglichen Namen der XML-Elemente und -Attribute als Konstanten. Sie wird von *SofauditConfiguration* und *SofauditMetricConfiguration* verwendet. Hierdurch werden Änderungen am XML-Schema der Konfigurationsdatei vereinfacht, falls diese notwendig sein sollten.

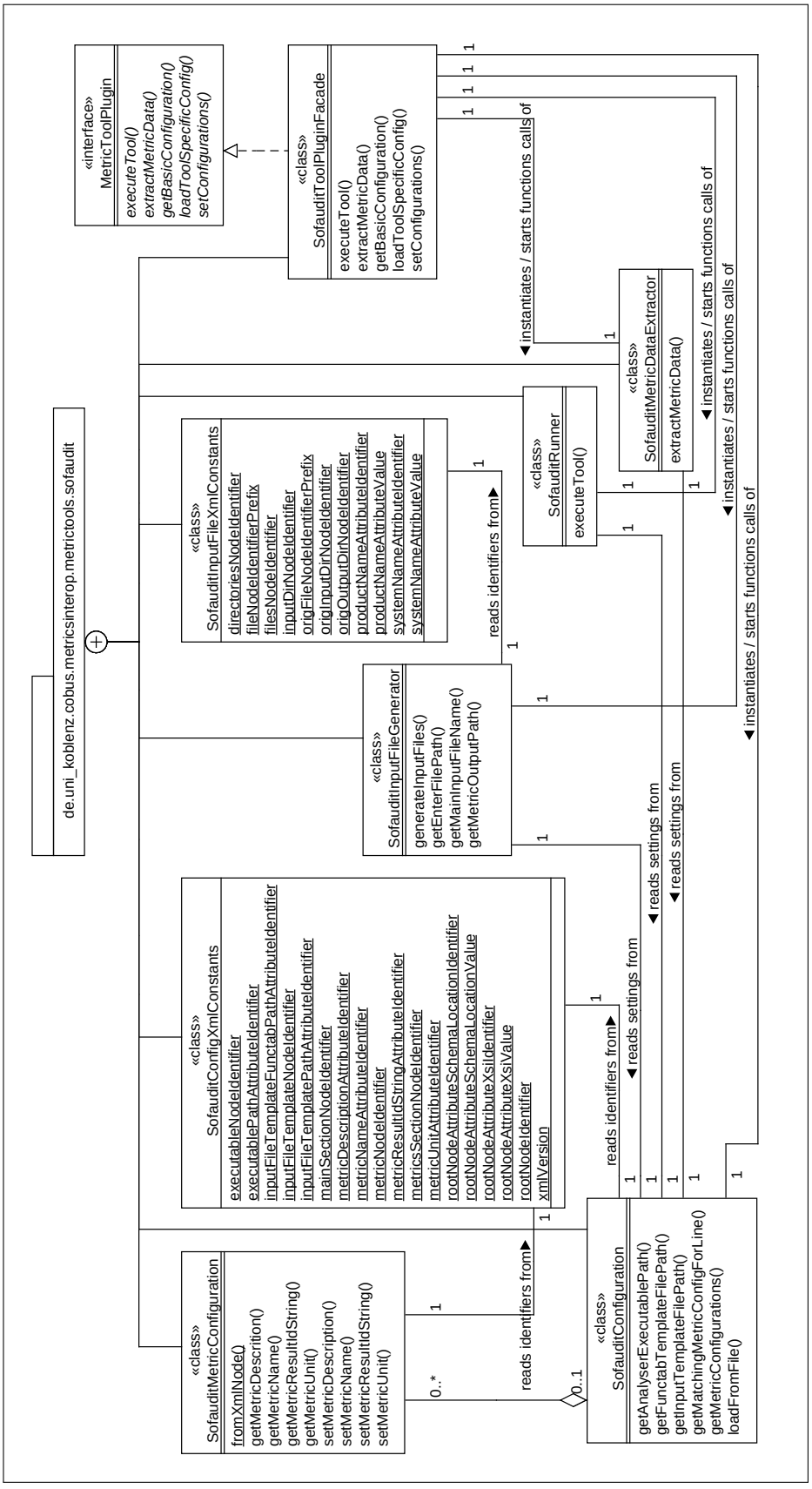

Abbildung 24: Klassendiagramm: Package *de.uni_koblenz.cobus.metricsinterop.metrictools.sofaudit*.

SofauditInputFileGenerator. Diese Klasse enthält die Funktionalität zum Erzeugen der Eingabedateien für eines der Analyseprogramm von SofAudit. Dazu werden entsprechende Vorlagendateien kopiert und modifiziert, welche über die spezifischen Einstellungen des Plug-Ins festgelegt wurden. Darüber hinaus kann die Klasse auch noch eine Reihe von Verzeichnissen erstellen, die von SofAudit benötigt werden.

SofauditInputFileXmlConstants. Diese Klasse enthält alle Namen der relevanten XML-Elemente und -Attribute aus der Haupteingabedatei als Konstanten. Sie wird von *SofauditInputFileGenerator* verwendet. Hierdurch werden Änderungen am XML-Schema der Eingabedatei vereinfacht, falls diese mit einer neueren Programmversion von SofAudit notwendig sein sollten.

SofauditRunner. Diese Klasse enthält die Funktionalität zum Ausführen eines der Analyseprogramme von SofAudit.

SofauditMetricDataExtractor. Diese Klasse enthält die Funktionalität zum Extrahieren der Metrikwerte aus den von SofAudit generierten Ergebnisdateien und zum Import in den Metrikdatenbestand des Hauptprogramms. Dabei kann über die spezifischen Einstellungen des Plug-Ins auch festgelegt werden, dass nur die Werte einer Teilmenge der berechneten Metriken importiert werden.

6.5 Struktur des Plug-Ins für ConQAT

Das Plug-In für das Metrik-Tool ConQAT befindet sich aufgrund des Umfangs in einem eigenen Package *de.uni_koblenz.cobus.metricsinterop.metrictools.conqat*. Das zugehörige Klassendiagramm ist in Abbildung 25 gezeigt, enhält aufgrund des Umfangs aber nur die Assoziationen innerhalb des Packages. Das Package beinhaltet die folgenden Klassen:

ConqatToolPluginFacade Diese Klasse implementiert die *MetricToolPlugin*-Schnittstelle. Die Funktionalität dieser Klasse beschränkt sich auf die Instanziierung aller benötigten weiteren Klassen des Plug-Ins und das Aufrufen der entsprechenden Methoden. Somit handelt es sich hierbei um eine Implementierung des *Facade*-Entwurfsmusters.

ConqatConfiguration Diese Klasse enhält die Funktionalität zum Vorhalten und Abfragen der spezifischen Einstellungen des Plug-Ins zur Laufzeit. Des Weiteren kann die Konfiguration aus einer XML-Datei eingelesen werden.

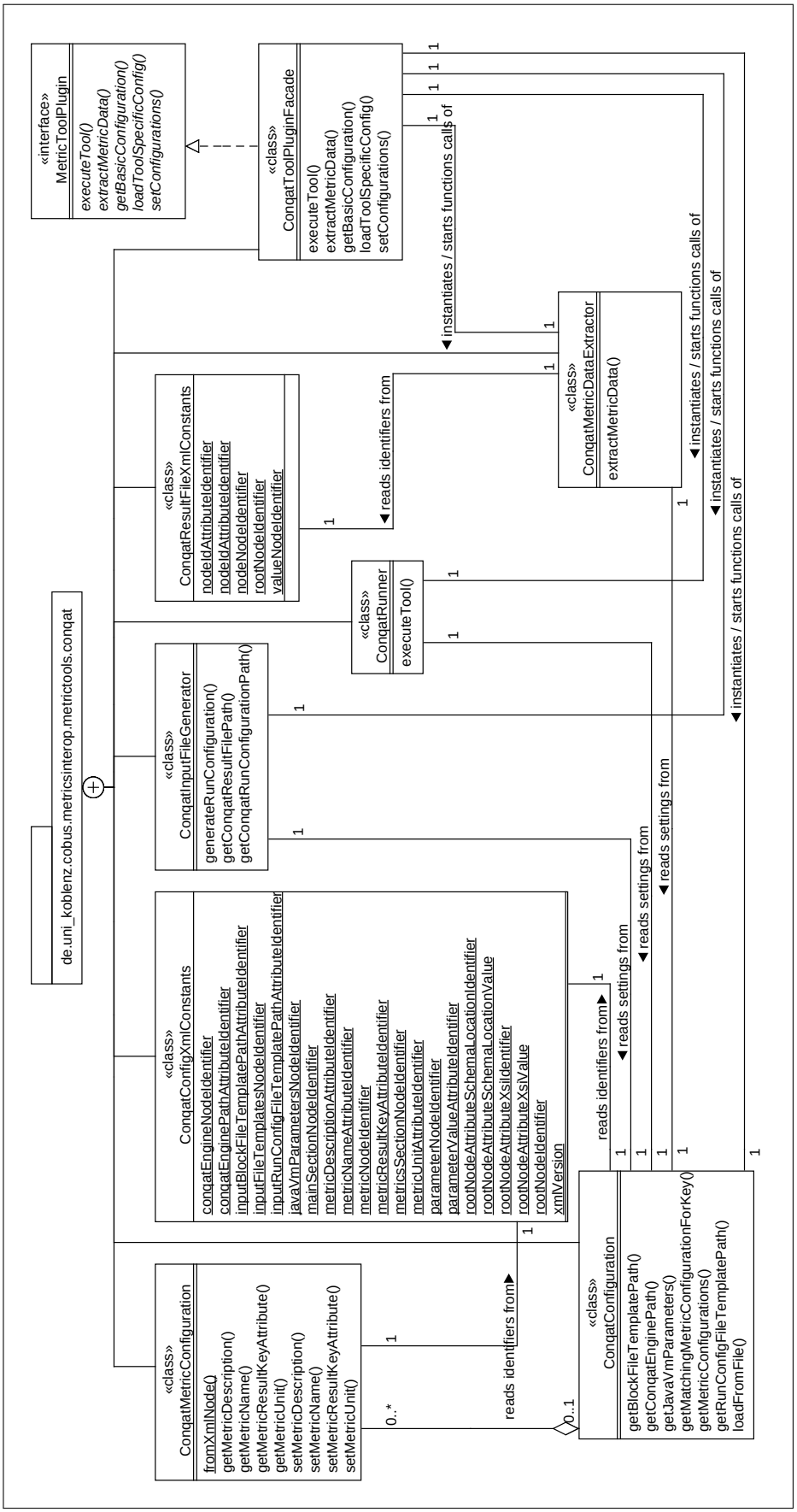

Abbildung 25: Klassendiagramm: Package *de.uni_koblenz.cobus.metricsinterop.metrictools.conqat*.

ConqatMetricConfiguration Diese Klasse enthält die Funktionalität zum Vorhalten und Abfragen der Einstellungen einer einzelnen Metrik zur Laufzeit. Neben dem Bezeichner, der Einheit und der Beschreibung der Metrik (zur Verwendung beim Import in den internen Metrikdatenbestand des Hauptprogramms) wird auch der Metrikbezeichner gespeichert, unter welchem die Metrikwerte in der Ergebnisdatei von ConQAT gespeichert sind. Darüber hinaus können die entsprechenden Einstellungen aus einem entsprechenden XML-Element eingelesen werden, welches ein Teil der spezifischen Konfigurationsdatei des Plug-Ins ist. Objekte dieser Klasse werden von *ConqatConfiguration* vorgehalten und benutzt.

ConqatConfigXmlConstants Diese Klasse enthält alle in der toolspezifischen Konfigurationsdatei möglichen Namen der XML-Elemente und -Attribute als Konstanten. Sie wird von *ConqatConfiguration* und *ConqatMetricConfiguration* verwendet. Hierdurch werden Änderungen am XML-Schema der Konfigurationsdatei vereinfacht, falls diese notwendig sein sollten.

ConqatInputFileGenerator Diese Klasse enthält die Funktionalität zum Erzeugen einer *Run Configuration*-Datei für ConQAT. Dazu wird diejenige Vorlagendatei kopiert und modifiziert, welche über die spezifischen Einstellungen des Plug-Ins festgelegt wurde. Der einer Run Configuration zugeordnete Block - welcher ebenfalls über die Einstellungen festgelegt wurde - wird nicht kopiert, sondern lediglich mit entsprechendem Pfad in der generierten Version referenziert.

ConqatRunner Diese Klasse enthält die Funktionalität zum Ausführen der ConQAT Engine mit einer generierten Run Configuration. Das Ausführen geschieht dabei als eigenständiges Programm in einer neuen Java-Instanz. Dies bietet im Gegensatz zur Ausführung als eingebettetes Programm einige Vorteile:

- Der Ausgabe- und Fehlerstrom kann einfacher eingelesen werden.
- Im Fall eines Abbruchs der ConQAT Engine kann der entsprechende Fehler mitgeloggt werden, ohne dass das Plugin oder Hauptprogramm selbst abbricht. Kritische Fehler wie der in Abschnitt 4.5.3 beschriebene Stapelüberlauf wären beim Ausführen als eingebettetes Programm nicht mittels *try...catch* abfangbar, da die komplette Java-Instanz abbricht.
- Das Plugin kann für die ConQAT Engine benötigte Umgebungsvariablen temporär für die Dauer der Ausführung selbst setzen, ohne dass hierfür bereits bei der Ausführung des Hauptprogramms Einstellungen durch den Benutzer oder das ausführende Skript notwendig sind.
- Die Java-Instanz der ConQAT Engine kann mit eigenen Parametern ausgeführt werden, ohne dass eventuell hiervon hervorgerufene Nebeneffekte auf das Hauptprogramm wirken.

ConqatMetricDataExtractor Diese Klasse enthält die Funktionalität zum Extrahieren der Metrikwerte aus der von ConQAT generierten XML-Ergebnisdatei und zum Import in den Metrikdatenbestand des Hauptprogramms. Dabei kann über die spezifischen Einstellungen des Plug-Ins auch festgelegt werden, dass nur die Werte einer Teilmenge der berechneten Metriken importiert werden.

ConqatResultFileXmlConstants Diese Klasse enthält alle zur Extraktion der Metrikdaten benötigten Namen der XML-Elemente und -Attribute aus der von ConQAT generierten XML-Ergebnisdatei als Konstanten. Sie wird von *ConqatMetricDataExtractor* verwendet. Hierdurch werden Änderungen vereinfacht, falls sich einmal das XML-Schema der ConQAT-Ergebnisdateien ändern sollte.

7 Verwendung der Software

Anhand des Entwurfs aus Kapitel 6 wurde das Hauptprogramm sowie die Plug-Ins für SofAudit, ConQAT, den XML- und CSV-Export implementiert. In diesem Kapitel wird beschrieben, wie das Hauptprogramm und die Plug-Ins verwendet werden. Darüber hinaus wird auch aufgeführt, welche Systemvoraussetzungen hierfür jeweils erfüllt werden müssen.

7.1 Hauptprogramm

Zusammen mit dem Hauptprogramm werden an dieser Stelle auch die Export-Plug-Ins beschrieben, da letztere keine spezifischen Voraussetzungen oder Konfigurationsdateien besitzen.

7.1.1 Systemvoraussetzungen

Zum Ausführen des Hauptprogramms, des CSV-Export-Plug-Ins und des XML-Export-Plug-Ins müssen die folgenden Voraussetzungen erfüllt sein:

- Java Laufzeitumgebung (JRE) ab Version 5[79] (entwickelt und getestet auf Version 6, Update 22 und 23)
- SVNKit 1.3.4 Standalone Version[80] (diese Komponente ist im *lib*-Ordner der Software bereits enthalten)
- Betriebssystem und Hardware: Alle von der Java Laufzeitumgebung unterstützten Plattformen und Systeme
- *svnkit.jar* muss per Classpath erreichbar sein (per Umgebungsvariable oder per Parameter beim Aufruf der Java Laufzeitumgebung)
- *metricsinterop.jar* oder der *bin*-Ordner muss per Classpath erreichbar sein (per Umgebungsvariable oder per Parameter beim Aufruf der Java Laufzeitumgebung)

7.1.2 Aufruf und Parameter

Der Start der Software erfolgt über die Befehlsfolge

```
java jopts de.uni_koblenz.cobus.metricsinterop.MetricInterop options
```

Dabei steht *jopts* für die optionalen Parameter der Java Laufzeitumgebung[81], welche an dieser

[79] `http://www.oracle.com/technetwork/java/javase/downloads/index.html`
[80] `http://svnkit.com/download.php`
[81] Eine Dokumentation der JRE-Parameter ist verfügbar unter `http://download.oracle.com/javase/6/docs/technotes/tools/windows/java.html`.

Stelle nicht beschrieben werden sollen. Die als *options* an die Software übergebenen möglichen Parameter sind:

- `--config` *confpath* oder `-c` *confpath* zum Übergeben des Pfades *confpath* der XML-Konfigurationsdatei. Der Pfad kann dabei eine absolute oder relative Angabe sein. Dieser Parameter wird zwingend und genau einmal benötigt.
- `--output` *outpath* oder `-o` *outpath* zum Übergeben des Pfades *outpath* für die Ausgabedatei(en). Der Pfad kann dabei eine absolute oder relative Angabe sein und sollte das Zielverzeichnis und den Dateinamen, nicht jedoch eine Dateierweiterung beinhalten. Letztere wird über die Einstellungen der einzelnen Datenexport-Plug-Ins in der Konfigurationsdatei festgelegt. Darüber hinaus wird vom Programm selbst auch eine Logdatei nach *outpath*-log.txt gespeichert. Dieser Parameter wird zwingend und genau einmal benötigt.
- `--name` *projectname* oder `-n` *projectname* zum Angeben eines Projektnamens *projectname*, welcher mit in den exportierten Metrikdaten gespeichert wird. Dieser Parameter ist optional und darf höchstens einmal übergeben werden.
- *inpath* zum Übergeben des Pfades *inpath* einer zu analysierenden Datei oder eines Verzeichnisses. Der Pfad kann dabei eine absolute oder relative Angabe sein. Bei Verzeichnissen wird nicht nur der Inhalt, sondern auch der Inhalt in allen Unterordnern analysiert. Dieser Parameter kann beliebig oft, jedoch aber mindestens einmal übergeben werden.

Die Reihenfolge der Parameter ist hierbei irrelevant, sie werden voneinander durch Leerzeichen getrennt. Enthält ein übergebener Wert selbst ein Leerzeichen, so muss dieser in Anführungszeichen übergeben werden. Eine vollständige Aufrufzeile sieht (auf einem Windows-System) wie das folgende Beipiel aus:

```
java de.uni_koblenz.cobus.metricsinterop.MetricInterop ↪
  D:\src\bsp\test.cbl D:\src\bsp\inc ↪
  --config D:\metrics\configs\config.xml ↪
  --output S:\metrics\results\test ↪
  --name "Mein Projektname..."
```

7.1.3 Konfigurationsdatei

Da die Menge der Aufrufparameter zugunsten der Übersichtlichkeit gering gehalten wurde, sind die weiteren Einstellungen mittels einer Konfigurationsdatei festzulegen. Da das Programm vor allem dazu gedacht ist, automatisiert von anderen Entwicklerwerzeugen ausgeführt zu werden, sind in der Konfigurationsdatei all diejenigen Einstellungen enthalten, welche üblicherweise nur einmal festgelegt werden und dementsprechend bei jedem Aufruf identisch bleiben. Die Datei ist eine XML-Datei mit einem festgelegten Schema. Listing 7 zeigt ein

Beispiel einer Konfigurationsdatei, das zugehörige Schema ist in Listing 15 im Anhang D enthalten.

```
<?xml version="1.0" encoding="utf-8"?>
<configuration xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="configschema.xsd">

  <main>
    <preprocessor path="D:\tools\dql-preprocessor\qs0020.exe"
        workingdirectory="D:\tools\dql-preprocessor" />
    <workingdirectory path="D:\temp" />
    <errorhandling faulttolerantmode="false" cleanuponerror="true"/>
    <sourcecodefiletypes>
      <filetype extension=".cbl" />
      <filetype extension=".cob" />
      <filetype extension=".cpy" />
    </sourcecodefiletypes>
    <ignoreddirectories>
      <directory name=".svn" />
      <directory name="_svn" />
      <directory name=".settings" />
    </ignoreddirectories>
  </main>

  <metrictools>
    <tool name="SofAudit" interopclass=
    "de.uni_koblenz.cobus.metricsinterop.metrictools.sofaudit.SofauditToolPluginFacade"
        configfile="D:\metrics\configs\config-sofaudit.xml" />
    <tool name="ConQAT" interopclass=
    "de.uni_koblenz.cobus.metricsinterop.metrictools.conqat.ConqatToolPluginFacade"
        configfile="D:\metrics\configs\config-conqat.xml" />
  </metrictools>

  <metricdataexporters>
    <exporter
        class="de.uni_koblenz.cobus.metricsinterop.metricdataexport.MetricDataXmlExporter"
        fileextension=".xml" />
    <exporter
        class="de.uni_koblenz.cobus.metricsinterop.metricdataexport.MetricDataCsvExporter"
        fileextension=".csv" />
  </metricdataexporters>

</configuration>
```

Listing 7: Beispiel einer XML-Konfigurationsdatei für das Hauptprogramm

In der Datei können die im Folgenden beschriebenen Optionen gesetzt werden:

- Über das *preprocessor*-Element kann der Pfad und das Arbeitsverzeichnis eines auf die Codedateien anzuwendenden Präprozessors konfiguriert werden. Das *workingdirectory*-Attribut ist optional. Soll kein Präprozessor ausgeführt werden, so kann das komplette Element weggelassen werden.
- Über das *workingdirectory*-Element kann das Arbeitsverzeichnis des Programms festgelegt werden. In diesem Verzeichnis erzeugt das Programm ein neues Unterverzeichnis,

in welchem die vorbereiteten Codedateien, Ausgabedateien der Metrik-Tools und sonstige Arbeitsdateien abgelegt werden. Das komplette Element kann auch weggelassen werden, in diesem Fall wird das per Umgebungsvariable angegebene Temporärverzeichnis verwendet.

- Über das *errorhandling*-Element kann das Verhalten des Programms im Fehlerfall festgelegt werden. Ist der fehlertolerante Modus über das Attribut *faulttolerantmode* eingeschaltet, so versucht das Programm im Fehlerfall weiterzuarbeiten, sofern dies beim konkreten Fehler möglich ist. Über das Attribut *cleanuponerror* kann festgelegt werden, ob das Programm bei einem fehlerbedingten Abbruch trotzdem das Arbeitsverzeichnis aufräumen soll. Letztere Einstellung ist unabhängig vom fehlertoleranten Modus, in diesem aber nur dann relevant, wenn der Fehler einen Abbruch des gesamten Programms bedingt. Falls das gesamte Element weggelassen wird, sind standardmäßig die in Listing 7 gezeigten Einstellungen aktiv.

- Mittels der *filetype*-Elemente unterhalb von *sourcecodefiletypes* können die Dateitypen festgelegt werden, die das Programm beim Durchsuchen von Verzeichnissen als zu analysierende Codedateien interpretieren soll. Diese Einstellung hat nur Einfluss auf den Inhalt von Verzeichnissen, die beim Programmaufruf übergeben werden. Werden einzelne Dateien übergeben, so werden diese auch verarbeitet, wenn sie keinem der eingestellten Typen entsprechen. Wird das komplette *sourcecodefiletypes*-Element weggelassen, so gelten für die Dateitypen die in Listing 7 gezeigten Einstellungen als Standard.

- Die in *ignoreddirectories* enthaltenen Elemente legen fest, welche Verzeichnisse anhand ihres Namens nicht nach Codedateien durchsucht werden sollen. Dies kann genutzt werden, um etwa die Ordner mit Arbeitsinformationen von Versionsverwaltungssystemen oder Entwicklungsumgebungen auszuschließen. Diese Ordner könnten Kopien von Codedateien enthalten und somit die generierten Metrikwerte verfälschen. Falls das komplette *ignoreddirectories*-Element weggelassen wird, bleibt diese Filterfunktion ausgeschaltet.

- Über die Einträge im *metrictools*-Element werden die vom Hauptprogramm zu verwendenden Plug-Ins für die einzelnen Metrik-Tools festgelegt. Pro Plug-In muss der Name des Tools (dieser dient nur zu Darstellungszwecken in den Statusmeldungen), der Pfad zur Konfigurationsdatei mit den Plug-In-spezifischen Einstellungen und der Klassenname des Plug-Ins angegeben werden. Der Klassenname muss dabei der vollqualifizierte Name derjenigen Klasse sein, welche die *MetricToolPlugin*-Schnittstelle implementiert. Es ist möglich, das gleiche Plug-In mehrfach mit unterschiedlichen Konfigurationsdateien festzulegen. Im *metrictools*-Element muss mindestens ein Plug-In enthalten sein. Zum erfolgreichen Ausführen müssen die Klassen eines Plug-Ins hierfür schon beim Start des Hauptprogramms via *Classpath* erreichbar sein.

- Die Einträge im *metricdataexporters*-Element legen fest, welche Plug-Ins für den Metrikdatenexport benutzt werden sollen. Pro Plug-In muss die entsprechende Dateinamenerweiterung und der Klassenname angegeben werden. Letzterer muss auch hier der vollqualifizierte Name einer Klasse sein, welche die *MetricDataExporter*-Schnittstelle implementiert. Ebenso müssen die Klassen eines Plug-Ins via *Classpath* beim Start des Hauptprogramms erreichbar sein. Weitere Einstellungen benötigen die Export-Plug-Ins nicht. Es muss mindestens ein Export-Plug-In festgelegt werden.

7.2 Plug-In für SofAudit

Im Folgenden werden die Voraussetzungen und die Einstellungen des Plug-Ins für das Metrik-Tool SofAudit behandelt.

7.2.1 Systemvoraussetzungen

Das Plug-In - für sich alleine genommen - stellt die gleichen Systemvoraussetzungen wie diejenigen, die zuvor in Abschnitt 7.1.1 für das Hauptprogramm aufgeführt wurden. Da das Plug-In eines der Analyseprogramme von SofAudit startet, kommen dadurch bedingt weitere Voraussetzungen bei der Verwendung des Plug-Ins zustande:

- SofAudit muss auf einem lokalen Laufwerk installiert und von den Benutzerrechten her ausführbar sein
- Beim Betriebssystem muss es sich um eine 32Bit-Version von Microsoft Windows handeln

7.2.2 Konfigurationsdatei

Wird das Plug-In verwendet, so benötigt es dazu eine eigene Konfigurationsdatei. Diese ist eine XML-Datei mit einem festgelegten Schema. Listing 8 zeigt ein Beispiel einer Konfigurationsdatei, das zugehörige Schema ist in Listing 16 im Anhang D enthalten.

In der Datei können die im Folgenden beschriebenen Optionen gesetzt werden:

- Mittels des *executable*-Elementes wird der Pfad zu dem zu verwendenden Analyseprogramm von SofAudit festgelegt. Diese Einstellung wird zwingend benötigt.
- Im Element *inputfiletemplate* werden die Pfade zu den Vorlagen für die Eingabedateien des SofAudit-Analyseprogramms angegeben. Das Plug-In erkennt und überschreibt dabei eventuell in den Vorlagen noch vorhandene Referenzen auf Dateien. Als Vorlagen sollten dabei nur Dateien zum Einsatz kommen, die vom GUI-basierten Hauptprogramm von SofAudit erzeugt wurden. Der Grund hierfür liegt darin, dass die Analyseprogramme eine ganz bestimmte Formatierung der Dateien benötigen. Dateiname und

-erweiterung der Vorlagen können beliebig sein, diese werden entsprechend in den generierten Kopien im Arbeitsverzeichnis durch das Plug-In sichergestellt. Die Pfade zu beiden Vorlagendateien werden zwingend benötigt.

- Mittels der *metric*-Elemente im zugehörigen Abschnitt wird festgelegt, für welche Metriken aus den von SofAudit berechneten Ergebnissen die Daten extrahiert und in den Datenbestand des Hauptprogramms importiert werden. Pro Metrik muss der Bezeichner (*name*-Attribut), die Beschreibung (*description*-Attribut) und die Einheit (*unit*-Attribut) angegeben werden. Darüber hinaus wird für jede Metrik derjenige Teil der Zeile benötigt, welcher in den von SofAudit generierten .MET-Dateien unmittelbar vor dem berechneten Wert steht und eine eindeutige Zuordnung der Metrik erlaubt. Im Beispiel aus Listing 8 würden nur die vier dort angegebenen Metriken extrahiert und importiert. Die Werte aller anderen von SofAudit berechneten Metriken blieben außen vor. Es muss mindestens ein *metric*-Element vorhanden sein und für jedes sind alle Attribute erforderlich.

```
1  <?xml version="1.0" encoding="utf-8"?>
2  <configuration xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:noNamespaceSchemaLocation="sofauditconfigschema.xsd">
3
4    <main>
5      <executable path="S:\cobus-metrics-interop\lib\sofaudit5r1009\COBAUDIT\COBAUDIT.EXE" />
6      <inputfiletemplate path="S:\temp\testconfig\sofauditinputtemplate.xml"
           functabpath="S:\temp\testconfig\sofauditfunctabtemplate.wrl" />
7    </main>
8
9    <metrics>
10     <metric name="sa-sourcefiles" description="SofAudit: Analysierte Codedateien"
           unit="Dateien" resultidstring="Number of Sources analyzed               =======>" />
11     <metric name="sa-loc"         description="SofAudit: Zeilen insgesamt"
           unit="Zeilen"  resultidstring="Number of Source Lines in all            =======>" />
12     <metric name="sa-sourceloc"   description="SofAudit: Codezeilen"
           unit="Zeilen"  resultidstring="Number of Genuine Code Lines             =======>" />
13     <metric name="sa-commentloc"  description="SofAudit: Kommentarzeilen"
           unit="Zeilen"  resultidstring="Number of Comment Lines                  =======>" />
14   </metrics>
15
16 </configuration>
```

Listing 8: Beispiel einer XML-Konfigurationsdatei für das SofAudit-Plug-In

7.3 Plug-In für ConQAT

Im Folgenden werden die Voraussetzungen und die Einstellungen des Plug-Ins für das Metrik-Tool ConQAT behandelt.

7.3.1 Systemvoraussetzungen

Das Plug-In - für sich alleine genommen - stellt die gleichen Systemvoraussetzungen wie die zuvor in Abschnitt 7.1.1 für das Hauptprogramm aufgeführten. Da das Plug-In die ConQAT

Engine ausführt, kommt dadurch bedingt eine weitere Voraussetzung bei der Verwendung des Plug-Ins zustande: Die ConQAT Engine muss für den aktuellen Benutzer erreichbar und von den Benutzerrechten her ausführbar sein (diese Komponente ist im *lib*-Ordner der Software bereits enthalten). Da die ConQAT Engine ebenfalls vollständig in Java implementiert wurde, sind deren weitere Systemvoraussetzungen bereits mit denen des Hauptprogramms vollständig abgedeckt.

7.3.2 Konfigurationsdatei

Wird das Plug-In verwendet, so benötigt es dazu eine eigene Konfigurationsdatei. Diese ist eine XML-Datei mit einem festgelegten Schema. Listing 9 zeigt ein Beispiel einer Konfigurationsdatei, das zugehörige Schema ist in Listing 17 im Anhang D enthalten.

```
1  <?xml version="1.0" encoding="utf-8"?>
2  <configuration xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:noNamespaceSchemaLocation="conqatconfigschema.xsd">
3
4    <main>
5      <conqatengine path="S:\cobus-metrics-interop\lib\conqat-2.7" />
6      <inputfiletemplates
           blockfilepath="S:\temp\testconfig\cq-templates\LoCExtendedAnalysis.cqb"
           runconfigfilepath="S:\temp\testconfig\cq-templates\LoCExtendedAnalysis.cqr" />
7      <javavmparameters>
8        <parameter value="-Xss32m" />
9        <parameter value="-Xmx512m" />
10     </javavmparameters>
11   </main>
12
13   <metrics>
14     <metric name="cq-loc"              description="ConQAT: Zeilen insgesamt"
           unit="Zeilen"    resultkeyattribute="LoC" />
15     <metric name="cq-commentratio"     description="ConQAT: Anteil Kommentarzeilen"
           unit="Anteil"    resultkeyattribute="CR" />
16     <metric name="cq-cratioassessment" description="ConQAT: Bewertung Anteil Kommentarzeilen"
           unit="Bewertung" resultkeyattribute="CR-Assessment" />
17   </metrics>
18
19 </configuration>
```

Listing 9: Beispiel einer XML-Konfigurationsdatei für das ConQAT-Plug-In

In der Datei können die im Folgenden beschriebenen Optionen gesetzt werden:

- Mittels des *conqatengine*-Elementes wird der Pfad zur zu verwendenden Version der ConQAT Engine festgelegt. Dabei muss es sich um den Basispfad der ConQAT Engine handeln, in welchem sich die Unterordner *bin*, *lib*, *bundles*, etc. befinden. Diese Einstellung wird zwingend benötigt, da vom Plug-In hieraus die zur Ausführung der Engine benötigte Umgebungsvariable *CONQAT_HOME* und einige zusätzliche Classpath-Werte erzeugt werden.

- Über das Attribut *runconfigfilepath* im Element *inputfiletemplates* wird festgelegt, wo sich die Vorlagendatei zum Generieren der Run Configuration befindet. Die Vorlage der Run Configuration muss dabei gewissen Anforderungen entsprechen, welche in Abschnitt 7.3.3 behandelt werden. Diese Einstellung wird zwingend benötigt.
- Über das Attribut *blockfilepath* im Element *inputfiletemplates* wird festgelegt, wo sich der zur Run Configuration zugehörige Block befindet. Auch der Block muss gewissen Anforderungen entsprechen, welche in Abschnitt 7.3.3 behandelt werden. Diese Einstellung wird zwingend benötigt.
- Mittels der *parameter*-Elemente im Abschnitt *javavmparameters* kann festgelegt werden, welche Argumente[82] beim Start der ConQAT Engine an deren eigene Instanz der Java-Laufzeitumgebung zusätzlich übergeben werden. Es können keine oder beliebig viele dieser Parameter festgelegt werden.
- Mittels der *metric*-Elemente im zugehörigen Abschnitt wird festgelegt, für welche Metriken aus den von ConQAT berechneten Ergebnissen die Daten extrahiert und in den Datenbestand des Hauptprogramms importiert werden. Pro Metrik muss der Bezeichner (*name*-Attribut), die Beschreibung (*description*-Attribut) und die Einheit (*unit*-Attribut) angegeben werden. Darüber hinaus wird für jede Metrik über das Attribut *resultkeyattribute* angegeben, welcher Wert dem *key*-Attribut in der XML-Ergebnisdatei von ConQAT für diese Metrik zugewiesen ist. Dies erlaubt eine eindeutige Zuordnung jedes Wertes zu einer Metrik. Es muss mindestens ein *metric*-Element vorhanden sein und für jedes sind alle Attribute erforderlich.

7.3.3 Anforderungen an Blocks und Run Configuration-Vorlagen

Wie bereits zuvor in Abschnitt 4.3.2 beschrieben, wird mittels eines Blocks beschrieben, welche einzelnen Processor-Klassen verwendet und welche Ein- und Ausgabeparameter hierzu benötigt werden. Damit beim Ausführen des Blocks (mit den Parametern aus einer zugehörigen Run Configuration) die vom Plug-In benötigte XML-Ausgabedatei erzeugt wird, muss im Block hierzu ein XMLFileWriter-Processor enthalten sein. Abbildung 26 zeigt als Beispiel den in Listing 9 referenzierten Block.

Bei einer Run Configuration handelt es sich um Textdatei. In dieser wird ein Block referenziert und für dessen benötigte Parameter werden konkrete Werte festgelegt. Eine vom Plug-In verwendbare Vorlagendatei einer Run Configuration wird am einfachsten erzeugt, indem mittels ConQAT (der Version mit Eclipse-GUI, nicht der Engine) eine neue Run Configuration zum gewünschten Block erstellt wird. Beim entsprechenden Parameter für das Eingabeverzeichnis der Quellcodedateien muss als Wert der Platzhalter %%*inputdirpath*%%, beim Parameter für die

[82] Einen Überblick über die möglichen Parameter gibt es unter `http://download.oracle.com/javase/6/docs/technotes/tools/windows/java.html`.

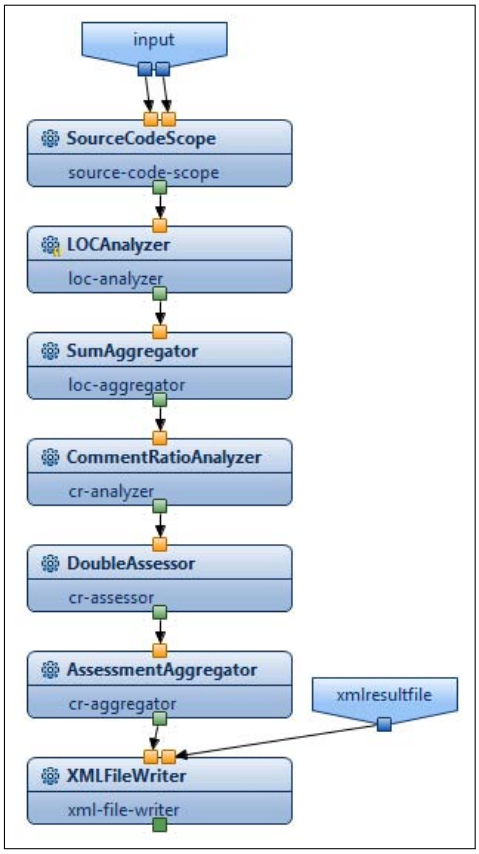

Abbildung 26: Zusammensetzung eines ConQAT Blocks mit einem *XMLFileWriter*-Processor.

XML-Ausgabedatei der Platzhalter *%%xmlresultfilepath%%* eingetragen werden. Alle weiteren Parameter müssen mit konkreten Werten versehen werden. Listing 10 zeigt die Vorlagendatei der Run Configuration für den Block aus Abbildung 26.

```
#!LoCExtendedAnalysis
input.sourcecodedir=%%inputdirpath%%
input.code-language=COBOL
xmlresultfile.path=%%xmlresultfilepath%%
```

Listing 10: Beispiel einer XML-Konfigurationsdatei für das ConQAT-Plug-In

8 Bewertung und Ausblick

Als abschließende Betrachtungen erfolgen in diesem Kapitel eine Bewertung anhand der erfüllten Anforderungen und der Zielsetzung der Arbeit sowie ein Ausblick auf mögliche Weiterentwicklungen der Software und aktuelle Entwicklungen im Rahmen des Forschungsprojekts Cobus.

8.1 Erfüllte Anforderungen

Bevor auf die eigentlichen Anforderungen an die Software eingegangen wird, soll an dieser Stelle noch einmal kurz auf das in Kapitel 5 entworfene Metrikdatenformat eingegangen werden.

Das per Schema definierte, XML-basierte Format ist so konzipiert, dass eine klare und einfache Struktur vorhanden ist. Diese ist aber so flexibel gestaltet, dass Daten beliebiger Metriken und beliebigen Typs gespeichert werden können. Sollte einmal eine Speicherung von Daten anderer als den bisher verwendeten Metrik-Tools erforderlich werden, so kann dies ohne weitere Änderungen am Schema erfolgen. Durch die Basierung auf XML ist die Maschinenverarbeitbarkeit in allen gängigen Programmiersprachen möglich, wobei der Aufwand hierfür aufgrund vorhandener XML-Komponenten wesentlich niedriger ausfällt als bei einem proprietären Format. Gleichzeitig erschwert das gewählte XML-Format die Lesbarkeit entsprechender Dateien für den Menschen nicht zu sehr, wozu auch die Verwendung eines Editors mit Unterstützung für XML-Schemata und Highlighting beitragen kann. Darüber hinaus kann auch das bei der Debeka vorhandene *Zeder*-System bereits XML-Daten verarbeiten.

Im Folgenden wird jede Anforderung aus Kapitel 6 aufgezählt und dabei mit einer Beschreibung hinsichtlich der Erfüllung durch das entworfene und implementierte Softwaresystem versehen:

1. Die Ausführung der Software soll eingebettet in die Entwicklungswerkzeuge der Debeka möglich sein:
   Diese Anforderung wurde erfüllt: Siehe Unterpunkte.
   (a) Die Ausführung des Programms muss als Konsolenanwendung ohne GUI erfolgen.
   Diese Anforderung wurde erfüllt: Das Softwaresystem ist vollständig als Konsolenanwendung realisiert.
   (b) Der vollständige Ablauf des Programms ohne manuellen Eingriff des Benutzers muss möglich sein.
   Diese Anforderung wurde erfüllt: Der Programmablauf erfordert nach dem Start keinerlei Benutzerinteraktion.
   (c) Die Steuerung und Konfiguration des Programms muss über Parameter oder Konfigurationsdateien möglich sein.

Diese Anforderung wurde erfüllt: Die Steuerung des Programms erfolgt vollständig über Aufrufparameter (für die pro Aufruf spezifischen Einstellungen) und Konfigurationsdateien (für die pro System oder Verwendungszwekc spezifischen Einstellungen).

(d) Die Ausführung der Metrik-Tools muss ebenfalls automatisch erfolgen.
Diese Anforderung wurde für ConQAT und SofAudit erfüllt. Aufgrund der in Kapitel 4 beschriebenen Gegebenheiten kann diese Anforderung für FGM nicht erfüllt werden.

(e) Für gegebenenfalls auftretende Fehler oder Statusmeldungen sollte eine Loggingfunktion vorhanden sein.
Diese Anforderung wurde erfüllt: Sämtliche Status- und Fehlermeldungen werden in einer Logdatei gespeichert, darüber hinaus werden darin auch eventuelle Ausgaben der aufgerufenen Metrik-Tools gespeichert.

2. Das Programm muss alle Metrik-Tools aus der engeren Auswahl (SofAudit, ConQAT) gleichermaßen unterstützen.
Diese Anforderung wurde für SofAudit und ConQAT erfüllt. Wie bereits beschrieben, erwies sich FGM als nicht geeignet für die geplante Verwendung.

3. Das Programm soll die Metrikdaten im spezifizierten Datenformat und zusätzlich im CSV-Format speichern können:
Diese Anforderung wurde für erfüllt: Mittels jeweils eines Plug-Ins können die Daten in die genannten Formate gespeichert werden.

   (a) Die berechneten Metrikwerte müssen aus den Datenformaten der Tools extrahiert werden.
   Diese Anforderung wurde erfüllt: Die von den Metrik-Tools generierten Daten werden vom jeweils spezifischen Plug-In extrahiert.

   (b) Die berechneten Metrikwerte müssen gegebenenfalls transformiert und im jeweiligen Datenformat gespeichert werden.
   Diese Anforderung wurde erfüllt: Die Daten werden zur Laufzeit in den internen Datenbestand der Software aufgenommen und dabei jeweils einer Metrik mit Bezeichner, Beschreibung und Einheit sowie dem Messobjekt zugeordnet. Die Speicherung dieses Datenbestandes erfolgt durch Export-Plug-Ins.

4. Das Programm sollte möglichst einfach erweiterbar sein:
Diese Anforderung wurde erfüllt: Siehe Unterpunkte.

   (a) Die Einbindung zusätzlicher Metrik-Tools zu einem späteren Zeitpunkt soll möglichst einfach realisierbar sein.
   Diese Anforderung wurde erfüllt: Das entwickelte Hauptprogramm stellt alle benötigten Basisfunktionen und eine Schnittstelle für Plug-Ins zur Verfügung. In diesen Plug-Ins werden dabei die toolspezifischen Funktionalitäten gekapselt, so dass eine Erweiterung um neue Tools ohne Modifikation des Hauptprogramms ermöglicht wird.

(b) Die Unterstützung weiterer Metriken soll möglichst einfach realisierbar sein.
Diese Anforderung wurde erfüllt: Die Software ist so konzipiert, dass die Daten beliebiger Metriken verarbeitet werden können. Welche Metriken konkret zum Einsatz kommen, ist dabei nur von den verwendeten Tools abhängig. Die Konfiguration der umgesetzten Plug-Ins erlaubt die Auswahl einer Teilmenge der Metriken des jeweiligen Tools beim Import der Daten.

(c) Der Export der Daten in andere Datenformate soll möglichst einfach realisierbar sein.
Diese Anforderung wurde erfüllt: Eine entsprechende Schnittstelle erlaubt die Erweiterung mittels zusätzlicher Plug-Ins für den Export, ohne dass hierfür eine Modifikation des Hauptprogramms nötig ist.

8.2 Bewertung der entwickelten Software und der verwendeten Metrik-Tools

Wie im vorangegangenen Abschnitt detailliert aufgeführt wurde, sind fast alle gestellten Anforderungen erfüllt worden. Die einzige Ausnahme wurde dadurch bedingt, dass sich die aktuelle Version von FGM erst bei der genaueren Analyse als ein rein GUI-basiertes Tool ohne Möglichkeit zur automatisierten Ausführung herausstellte.

Anhand der bei der Recherche zugrunde liegenden Kriterien ist die getroffene Auswahl der Tools dennoch richtig. Diese wurde allerdings auch dadurch wesentlich beeinflusst, dass viele Informationen zu den recherchierten Metrik-Tools nicht in Erfahrung gebracht werden konnten, da viele Hersteller trotz mehrfacher Anfragen keinerlei Reaktion gezeigt haben. Das Verhalten der meisten Hersteller legt die Vermutung nahe, dass vor allem ein Interesse am Erbringen von entsprechenden Dienstleistungen und nicht am einmaligen Verkauf von Lizenzen besteht.

Zeitlich parallel zu dieser Arbeit ist von den am Forschungsprojekt Cobus beteiligten Personen der Debeka und des Instituts für Softwaretechnik ein Katalog mit Metriken entstanden (siehe [Haas2010-2]). Die darin festgelegten Metriken sollen als Grundlage der Analyse und Bewertung des COBOL-basierten Softwaresystems der Debeka dienen.

Beim Vergleich des Metrik-Katalogs mit den Analysefunktionen der recherchierten Tools zeigt sich, dass der Katalog einige Metriken enthält, welche von keinem der Tools berechnet werden können. Die meisten der Programme lassen keine oder nur eine eingeschränkte Erweiterung um neue Metriken zu. Die einzige Ausnahme hierzu stellt ConQAT mit seinem flexiblen und modularen Aufbau dar, bildet mit der vorhandenen Funktionalität aber nur eine sehr geringe Schnittmenge mit dem Metrik-Katalog.

Bei den detailliert untersuchten Tools hat sich auch gezeigt, dass diese teilweise nicht ganz präzise oder nach einer anderen als der für die jeweilige Metrik definierten Berechnungsvorschrift arbeiten. Alles in allem sind die für COBOL existierenden Metrik-Tools also für die Verwendung im Rahmen des Cobus-Projekts eher ungeeignet.

Das eine Ziel dieser Arbeit (siehe dazu auch Kapitel 2.5) war die Bewertung der Eignung bestehender Metrik-Software für COBOL hinsichtlich der Interoperabilität mit den bestehenden Entwicklerwerkzeugen der Debeka. Das andere Ziel war die Erstellung einer entsprechenden funktionsfähigen Implementation für einige ausgewählte Tools, mit der Möglichkeit der Erweiterung um neue Tools und der Unterstützung für weitere Metriken. Die Hauptziele dieser Arbeit sind also erreicht worden.

8.3 Ausblick

Die im Rahmen dieser Arbeit entwickelte Software erfüllt die Anforderungen und ist von diesem Gesichtspunkt her vollständig. Dennoch sollen an dieser Stelle einige Ideen genannt werden, die in einer eventuellen Nachfolgerversion möglich wären.

Es wäre denkbar, eine Funktionalität zum Berechnen weiterer Metriken auf der Basis des programminternen Metrikdatenbestands zu realisieren. Eine solche Funktion müsste dazu die Berechnungsvorschriften aus einer Konfigurationsdatei einlesen, interpretieren und ausführen. Im Programmablauf würde dies nach dem Ausführen und der Extraktion der Daten der Metrik-Tools geschehen, bevor der Datenbestand gespeichert wird.

Es wäre auch denkbar, neben der Analyse von Dateien und Verzeichnissen einer lokalen Arbeitskopie auch eine Unterstützung für den direkten Zugriff auf die Datenbestände eines SVN-Servers oder eines anderen Versionsverwaltungssystems zu realisieren.

Darüber hinaus wäre es denkbar, einen Benutzeroberfläche zum Bearbeiten der Konfigurationsdateien umzusetzen.

Aufgrund der mangelnden Eignung der existierenden Metrik-Tools zum Erreichen der Ziele von Cobus befindet sich mittlerweile ein neues Tool in der Entwicklung. Dieses basiert auf einem mit ANTLR erzeugten COBOL-Parser und der JGraLab-Software des Instituts für Softwaretechnik. Zur Berechnung der Metriken kommen bei diesem Tool GreQL-Anfragen zum Einsatz, so dass hierüber auch eine gute Erweiterbarkeit gegeben ist. Die Entwicklung erfolgt gemeinsam durch die Debeka und das Institut für Softwaretechnik.

Ob dieses neue Metrik-Tool nach der Fertigstellung von der hier entwickelten Software ausgeführt wird, bleibt abzuwarten. Es wäre aber ebenso denkbar, dass Komponenten der hier entwickelten Software in das neue Tool einfließen.

A Übersicht über die von SofAudit berechneten Metrikwerte

Das Listing 11 zeigt eine .MET-Datei, welche in dieser Form von SofAudit generiert wird und die Werte aller von SofAudit berechneten Metriken beinhaltet.

```
+--------------------------------------------------------------------+
|     S O F T A N A L   P R O G R A M   M E T R I C   R E P O R T    |
|                                                                    |
|   LANGUAGE: COB85                                 DATE: 12.09.10   |
|                                                                    |
|   MODULE: lb                                      PAGE:        1   |
|                                                                    |
+--------------------------------------------------------------------+
|                  Q U A N T I T Y   M E T R I C S                   |
+--------------------------------------------------------------------+
|            C O D E   Q U A N T I T Y   M E T R I C S               |
|                                                                    |
|       Number of Sources analyzed              =======>     1185    |
|       Number of Source Lines in all           =======>  1128778    |
|       Number of Genuine Code Lines            =======>   790071    |
|       Number of Comment Lines                 =======>   326931    |
|       Number of Major Rule Violations         =======>    24213    |
|       Number of Medium Rule Violations        =======>   245537    |
|       Number of Minor Rule Violations         =======>    51797    |
|                                                                    |
|        S T R U C T U R A L  Q U A N T I T Y  M E T R I C S          |
|                                                                    |
|       Number of Modules                       =======>      751    |
|       Number of Copy/Includes                 =======>      434    |
|       Number of Copy/Include References       =======>    10089    |
|       Number of Entry Points                  =======>      751    |
|       Number of Exit Points                   =======>      749    |
|       Number of Sections/Procedures           =======>    12416    |
|       Number of Labels/Paragraphs/Code Blocks  =======>    38232    |
|       Number of Reusable Code Blocks          =======>     6198    |
|       Number of Data Structures/Objects       =======>    24854    |
|       Number of Reusable Data Objects         =======>    10746    |
|                                                                    |
|             D A T A   Q U A N T I T Y   M E T R I C S              |
|                                                                    |
|       Number of Panels processed              =======>        0    |
|       Number of Reports produced              =======>      161    |
|       Number of Files declared                =======>     1041    |
|       Number of Data Bases accessed           =======>        0    |
|       Number of Data Views selected           =======>        0    |
|       Number of Global Data Structures        =======>    11484    |
|       Number of Local  Data Structures        =======>    13370    |
|       Number of Data Variables declared       =======>   158577    |
|       Number of Global Data Variables         =======>    81325    |
|       Number of Local Data Variables          =======>    77252    |
|       Number of Conditional Variables         =======>     6209    |
|       Number of Data Constants                =======>   334995    |
|       Number of Redefinitions (Unions)        =======>     2919    |
|       Number of Arrays ( Vectors)             =======>     1983    |
|       Number of different Data Types          =======>    59296    |
|       Number of Data Elements referenced      =======>   226697    |
|       Number of Arguments / Input Variables   =======>    64335    |
|       Number of Results / Output Variables    =======>    92170    |
|       Number of Predicates / Conditional Data  =======>     5591    |
```

```
|       Number of Parameters / Function Arguments=======>     5078     |
|       Number of Communication Data Fields      =======>        0     |
|       Number of Database Attributes            =======>    12344     |
|       Number of Linkage Data Fields            =======>       15     |
|       Number of Convertible Data Variables     =======>    57300     |
|       Number of Data-Points                    =======>    66758     |
|                                                                       |
|         P R O C E D U R A L   Q U A N T I T Y   M E T R I C S         |
|                                                                       |
|       Number of Statements                     =======>   622776     |
|       Number of Macro Statements               =======>        0     |
|       Number of Procedural Statements          =======>   470291     |
|       Number of Convertable Statements         =======>   375846     |
|       Number of Input Operations               =======>      408     |
|       Number of Output Operations              =======>     1811     |
|       Number of File & Database Accesses       =======>     4901     |
|       Number of Foreign Modules called         =======>     4511     |
|       Number of Call Statements                =======>    19298     |
|       Number of Perform Statements             =======>    44195     |
|       Number of Selections (If & Case)         =======>    70022     |
|       Number of Loop Statements (Until/While)  =======>     4212     |
|       Number of GOTO Branches                  =======>    17236     |
|       Number of all Control Statements         =======>   135375     |
|       Number of Control Flow Branches          =======>   195440     |
|       Number of Literals in Statements         =======>    50143     |
|       Number of Constants in Statements        =======>   218534     |
|       Number of Test Cases required            =======>    40033     |
|       Number of different Statement Types      =======>    21746     |
|       Number of Function-Points                =======>     8092     |
+-----------------------------------------------------------------------+
/
+-----------------------------------------------------------------------+
|     S O F T A N A L   P R O G R A M   M E T R I C   R E P O R T       |
|                                                                       |
|   LANGUAGE: COB85                                 DATE: 12.09.10      |
|                                                                       |
|   MODULE: lb                                      PAGE:        2      |
|                                                                       |
+-----------------------------------------------------------------------+
|              C O M P L E X I T Y       M E T R I C S                  |
+-----------------------------------------------------------------------+
|       DATA COMPLEXITY (Chapin Metric)          =======>  0.048        |
|       DATA FLOW COMPLEXITY (Elshof Metric)     =======>  0.595        |
|       DATA ACCESS  COMPLEXITY (Card Metric)    =======>  0.825        |
|       INTERFACE COMPLEXITY (Henry Metric)      =======>  0.260        |
|       CONTROL FLOW COMPLEXITY (McCabe Metric)  =======>  0.680        |
|       DECISIONAL COMPLEXITY (McClure Metric)   =======>  0.283        |
|       BRANCHING COMPLEXITY (Sneed Metric)      =======>  0.662        |
|       LANGUAGE COMPLEXITY  (Halstead Metric)   =======>  0.150        |
|                                                                       |
|       AVERAGE  PROGRAM COMPLEXITY              =======>  0.437        |
|                                                                       |
+-----------------------------------------------------------------------+
|                  Q U A L I T Y     M E T R I C S                      |
+-----------------------------------------------------------------------+
|       DEGREE OF MODULARITY                     =======>  0.729        |
|       DEGREE OF PORTABILITY                    =======>  0.945        |
|       DEGREE OF TESTABILITY                    =======>  0.712        |
|       DEGREE OF REUSABILITY                    =======>  0.268        |
|       DEGREE OF CONVERTIBILITY                 =======>  0.554        |
```

```
115 |       DEGREE OF FLEXIBILITY                    =======>  0.429       |
116 |       DEGREE OF CONFORMITY                     =======>  0.505       |
117 |       DEGREE OF MAINTAINABILITY                =======>  0.545       |
118 |                                                                       |
119 |       AVERAGE  PROGRAM QUALITY                 =======>  0.585       |
120 |                                                                       |
121 +-----------------------------------------------------------------------+
```

Listing 11: Beispielübersicht aller von SofAudit berechneten Metrikwerte

B Algorithmen der Komplexitätsmetriken in SofAudit

In diesem Kapitel werden die verwendeten Berechnungsvorschriften der Komplexitätsmetriken von SofAudit auf eine Abweichung von den allgemein verbreiteten Algorithmen untersucht. Die McCabe- und Halstead-Metriken werden dabei ausgelassen, da diese bereits entsprechend in Kapitel 4.6 abgehandelt werden.

Die in den Berechnungsvorschriften enthaltenen Bezeichner stehen dabei für folgende Werte:

- *Files*: Anzahl deklarierter Dateien
- *DBs*: Anzahl Datenbanken, auf welche zugegriffen wird
- *FileAccesses*: Anzahl Dateizugriffe
- *DBAccesses*: Anzahl Datenbankzugriffe
- *DataUsed*: Anzahl referenzierter Datenelemente
- *DataReferences*: Anzahl Referenzen auf Datenelemente
- *Predicates*: Anzahl Prädikate / Bedingungen
- *Results*: Anzahl Funktionsergebnisse / Ausgabevariablen
- *Arguments*: Anzahl Argumente / Eingabevariablen
- *Parameters*: Anzahl Parameter / Funktionsparameter
- *TPOps*: Anzahl der Transportoperationen
- *Includes*: Anzahl der mittels Copy / Include referenzierten Elemente
- *Calls*: Anzahl der Call-Aufrufe
- *ProcStatements*: Anzahl prozeduraler Anweisungen
- *Selections*: Anzahl der *IF*-Anweisungen
- *Cases*: Anzahl der *CASE*-Anweisungen
- *Loops*: Anzahl der Schleifen (*UNTIL*, *WHILE*)

Card. Die Berechnungsvorschrift für die Card-Metrik (Datenzugriffskomplexität) wird im Handbuch von SofAudit mit folgender Formel angegeben (siehe [Sneed2008-2], S.60):

$$AccessCompl = 1 - \frac{Files + DBs}{Files + DBs + FileAccesses + DBAccesses}$$

Die Datenzugriffkomplexität nach Card ist allgemein wie folgt definiert (siehe [Masak2005]):

$$\gamma_i = \delta_i + \beta_i$$

mit

$$\delta_i = \frac{1}{m} * \sum_{j=1}^{m} \phi_i^2(j)$$
$$\beta_i = \frac{1}{m} * \sum_{j=1}^{m} \frac{v_i(j)}{\phi_i(j) + 1}$$

Dabei wird δ als strukturelle Komplexität und β als Datenkomplexität bezeichnet. Die Komponenten der Funktionen sind wie folgt definiert:

- *i*: Das aktuelle Modul
- *m*: Die Anzahl interner Prozeduren
- *v*: Die Anzahl der I/O-Variablen im Modul
- ϕ: Die Anzahl der modulexternen Aufrufe

Da die beiden Berechnungsvorschriften völlig unterschiedlich sind und auch auf unterschiedlichen Daten basieren, sind die von SofAudit berechneten Werten nicht mit denen der eigentlichen Card-Metrik konform.

Chapin. Die Berechnungsvorschrift für die Chapin-Metrik (Datenkomplexität) wird im Handbuch von SofAudit mit folgender Formel angegeben (siehe [Sneed2008-2], S.60):

$$DataCompl = 1 - \frac{DataUsed}{Predicates * 2 + Results * 1,5 + Arguments * 1 + Parameters * 0,5}$$

Von den Erfindern dieser Metrik wurde die Datenkomplexität Q wie folgt definiert (siehe [Chapin1978]):

$$Q = (3 * C + 2 * M + P + \frac{1}{2} * T) * (1 + (\frac{1}{3} * E)^2)$$

Dabei sind die Komponenten der Funktion wie folgt definiert:

- *C*: Eingabedaten, welche als Bedingungen für Verzweigungen oder Entscheidungen verwendet werden
- *M*: Eingabedaten, welche im Programmablauf modifiziert werden
- *P*: Für den Prozess notwendige Eingabedaten
- *T*: Im Programmablauf unveränderte Daten
- *E*: Ausdruck für Verbindungen zwischen Modulen

Die Semantik der zugrundeliegenden Werte beider Berechnungsvorschriften ist weitestgehend identisch. Allerdings setzt SofAudit andere Faktoren zur Gewichtung ein und skaliert darüber hinaus die Werte auf einen Bereich von null bis eins. Somit handelt es sich also nicht um die Chapin-Metrik im eigentlichen Sinne.

Henry. Die Berechnungsvorschrift für die Henry-Metrik (Schnittstellenkomplexität) wird im Handbuch von SofAudit mit folgender Formel angegeben (siehe [Sneed2008-2], S.60-61):

$$LangCompl = \frac{DBAccesses * 8 + TPOps + 8 + Includes * 6 + FileAccesses * 4 + Calls * 2}{ProcStatements}$$

Von den Erfindern dieser Metrik wurde die Komplexität des Informationsflusses *IF(M)* eines Moduls *M* wie folgt definiert (siehe [Henry1981]):

$$IF(M) = length(M) * (fanin(M) * fanout(M))^2$$

Dabei sind die einzelnen Funktionen wie folgt definiert:

- *length(M)*: Ein Maß nach Wahl für den Umfang des Moduls; üblicherweise wird die Anzahl echter Codezeilen verwendet
- *fanin(M)*: Die Anzahl der in M eingehenden Datenflüsse (Parameter; globale, von M gelesene Datenstrukturen)
- *fanout(M)*: Die Anzahl der von M ausgehenden Datenflüsse (Rückgabewerte; globale, von M geschriebene Datenstrukturen)

Da die beiden Berechnungsvorschriften nicht identisch sind, sind die von SofAudit berechneten Werten nicht mit denen der eigentlichen Henry-Metrik konform.

McClure. Die Berechnungsvorschrift für die McClure-Metrik (Entscheidungskomplexität) wird im Handbuch von SofAudit mit folgender Formel angegeben (siehe [Sneed2008-2], S.61):

$$ConditionalCompl = \frac{Selections * 2 + Cases * 2 + Loops * 4}{ProcStatements}$$

Vom Erfinder dieser Metrik wurde die Entscheidungskomplexität *D* wie folgt definiert (siehe [McClure1978]):

$$D = C + V$$

Dabei sind die Komponenten der Funktion wie folgt definiert:

- *C*: Anzahl der Entscheidungsoperationen
- *V*: Anzahl der Variablen, welche als Bedingung in Entscheidungen referenziert werden

SofAudit berechnet die Werte hier ebenfalls auf Basis der Entscheidungsoperationen. Allerdings werden dabei zusätzliche Faktoren zur Gewichtung verwendet und die Bedingungen auf Basis von Variablen außen vor gelassen. Darüber hinaus werden die Werte in einem Verhältnis zur Anzahl der prozeduralen Anweisungen auf einen Bereich von null bis eins skaliert. Somit handelt es sich also nicht um die McClure-Metrik im eigentlichen Sinne.

C Metrikdatenformate anderer Tools

Der Code in Listing 12 zeigt XML-Daten, welche vom Eclipse Metrics Plug-In generiert wurden und die Metrikwerte für das Hilfstool UnDQL[83], SVN-Revision r12205, beinhalten. Im Anschluss daran findet sich das zugehörige Schema in Listing 13. Als letztes zeigt Listing 14 vom XMLFileWriter-Processor von ConQAT erzeugte Daten.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <Metrics scope="UnDQL" type="Project" date="2010-08-09"
       xmlns="http://metrics.sourceforge.net/2003/Metrics-First-Flat">
3     <Metric id = "NORM" description ="Number of Overridden Methods">
4        <Values per = "type" total = "0" avg = "0" stddev = "0" max = "0">
5           <Value name="CobolSourcefileFilter" source ="CobolSourcefileFilter.java" package
                ="(default package)" value ="0"/>
6           <Value name="DirectoryFilter" source ="DirectoryFilter.java" package ="(default
                package)" value ="0"/>
7           <Value name="UnDQL" source ="UnDQL.java" package ="(default package)" value ="0"/>
8        </Values>
9     </Metric>
10    <Metric id = "NOF" description ="Number of Attributes">
11       <Values per = "type" total = "0" avg = "0" stddev = "0" max = "0">
12          <Value name="CobolSourcefileFilter" source ="CobolSourcefileFilter.java" package
                ="(default package)" value ="0"/>
13          <Value name="DirectoryFilter" source ="DirectoryFilter.java" package ="(default
                package)" value ="0"/>
14          <Value name="UnDQL" source ="UnDQL.java" package ="(default package)" value ="0"/>
15       </Values>
16    </Metric>
17    <Metric id = "NSC" description ="Number of Children">
18       <Values per = "type" total = "0" avg = "0" stddev = "0" max = "0">
19          <Value name="CobolSourcefileFilter" source ="CobolSourcefileFilter.java" package
                ="(default package)" value ="0"/>
20          <Value name="DirectoryFilter" source ="DirectoryFilter.java" package ="(default
                package)" value ="0"/>
21          <Value name="UnDQL" source ="UnDQL.java" package ="(default package)" value ="0"/>
22       </Values>
23    </Metric>
24    <Metric id = "NOC" description ="Number of Classes">
25       <Values per = "packageFragment" total = "3" avg = "3" stddev = "0" max = "3">
26          <Value name="(default package)" package ="(default package)" value ="3"/>
27       </Values>
28    </Metric>
29    <Metric id = "MLOC" description ="Method Lines of Code">
30       <Values per = "method" total = "96" avg = "12" stddev = "12,176" max = "35">
31          <Value name="processDirectory" source ="UnDQL.java" package ="(default package)"
                value ="35"/>
32          <Value name="evaluateArguments" source ="UnDQL.java" package ="(default package)"
                value ="30"/>
33          <Value name="processFile" source ="UnDQL.java" package ="(default package)" value
                ="9"/>
34          <Value name="printHelp" source ="UnDQL.java" package ="(default package)" value
                ="8"/>
35          <Value name="main" source ="UnDQL.java" package ="(default package)" value ="7"/>
```

[83] Verfügbar im SVN-Repository dieser Arbeit unter `https://svn.uni-koblenz.de/gup/re-group/trunk/project/cobus-metrics-interop/src/UnDQL`

```
36          <Value name="accept" source ="CobolSourcefileFilter.java" package ="(default
                package)" value ="3"/>
37          <Value name="accept" source ="DirectoryFilter.java" package ="(default package)"
                value ="3"/>
38          <Value name="processDirectory" source ="UnDQL.java" package ="(default package)"
                value ="1"/>
39       </Values>
40    </Metric>
41    <Metric id = "NOM" description ="Number of Methods">
42       <Values per = "type" total = "2" avg = "0,667" stddev = "0,471" max = "1">
43          <Value name="CobolSourcefileFilter" source ="CobolSourcefileFilter.java" package
                ="(default package)" value ="1"/>
44          <Value name="DirectoryFilter" source ="DirectoryFilter.java" package ="(default
                package)" value ="1"/>
45          <Value name="UnDQL" source ="UnDQL.java" package ="(default package)" value ="0"/>
46       </Values>
47    </Metric>
48    <Metric id = "NBD" description ="Nested Block Depth" max ="5" hint ="use Extract-method to
          split the method up">
49       <Values per = "method" avg = "1,875" stddev = "1,053" max = "4">
50          <Value name="processDirectory" source ="UnDQL.java" package ="(default package)"
                value ="4"/>
51          <Value name="evaluateArguments" source ="UnDQL.java" package ="(default package)"
                value ="3"/>
52          <Value name="main" source ="UnDQL.java" package ="(default package)" value ="2"/>
53          <Value name="processFile" source ="UnDQL.java" package ="(default package)" value
                ="2"/>
54          <Value name="accept" source ="CobolSourcefileFilter.java" package ="(default
                package)" value ="1"/>
55          <Value name="accept" source ="DirectoryFilter.java" package ="(default package)"
                value ="1"/>
56          <Value name="printHelp" source ="UnDQL.java" package ="(default package)" value
                ="1"/>
57          <Value name="processDirectory" source ="UnDQL.java" package ="(default package)"
                value ="1"/>
58       </Values>
59    </Metric>
60    <Metric id = "DIT" description ="Depth of Inheritance Tree">
61       <Values per = "type" avg = "1" stddev = "0" max = "1">
62          <Value name="CobolSourcefileFilter" source ="CobolSourcefileFilter.java" package
                ="(default package)" value ="1"/>
63          <Value name="DirectoryFilter" source ="DirectoryFilter.java" package ="(default
                package)" value ="1"/>
64          <Value name="UnDQL" source ="UnDQL.java" package ="(default package)" value ="1"/>
65       </Values>
66    </Metric>
67    <Metric id = "NOP" description ="Number of Packages">
68       <Value value="1"/>
69    </Metric>
70    <Metric id = "CA" description ="Afferent Coupling">
71       <Values per = "packageFragment" avg = "0" stddev = "0" max = "0">
72          <Value name="(default package)" package ="(default package)" value ="0"/>
73       </Values>
74    </Metric>
75    <Metric id = "NOI" description ="Number of Interfaces">
76       <Values per = "packageFragment" total = "0" avg = "0" stddev = "0" max = "0">
77          <Value name="(default package)" package ="(default package)" value ="0"/>
78       </Values>
79    </Metric>
```

```
80    <Metric id = "VG" description ="McCabe Cyclomatic Complexity" max ="10" hint ="use
          Extract-method to split the method up">
81       <Values per = "method" avg = "4,375" stddev = "4,386" max = "14" maxinrange="false">
82          <Value name="evaluateArguments" source ="UnDQL.java" package ="(default package)"
                value ="14" inrange="false"/>
83          <Value name="processDirectory" source ="UnDQL.java" package ="(default package)"
                value ="9"/>
84          <Value name="accept" source ="CobolSourcefileFilter.java" package ="(default
                package)" value ="4"/>
85          <Value name="accept" source ="DirectoryFilter.java" package ="(default package)"
                value ="2"/>
86          <Value name="main" source ="UnDQL.java" package ="(default package)" value ="2"/>
87          <Value name="processFile" source ="UnDQL.java" package ="(default package)" value
                ="2"/>
88          <Value name="printHelp" source ="UnDQL.java" package ="(default package)" value
                ="1"/>
89          <Value name="processDirectory" source ="UnDQL.java" package ="(default package)"
                value ="1"/>
90       </Values>
91    </Metric>
92    <Metric id = "TLOC" description ="Total Lines of Code">
93       <Value value="124"/>
94    </Metric>
95    <Metric id = "RMI" description ="Instability">
96       <Values per = "packageFragment" avg = "1" stddev = "0" max = "1">
97          <Value name="(default package)" package ="(default package)" value ="1"/>
98       </Values>
99    </Metric>
100   <Metric id = "PAR" description ="Number of Parameters" max ="5" hint ="Move invoked method
          or pass an object">
101      <Values per = "method" avg = "1,25" stddev = "0,661" max = "2">
102         <Value name="processDirectory" source ="UnDQL.java" package ="(default package)"
                value ="2"/>
103         <Value name="processDirectory" source ="UnDQL.java" package ="(default package)"
                value ="2"/>
104         <Value name="processFile" source ="UnDQL.java" package ="(default package)" value
                ="2"/>
105         <Value name="accept" source ="CobolSourcefileFilter.java" package ="(default
                package)" value ="1"/>
106         <Value name="accept" source ="DirectoryFilter.java" package ="(default package)"
                value ="1"/>
107         <Value name="evaluateArguments" source ="UnDQL.java" package ="(default package)"
                value ="1"/>
108         <Value name="main" source ="UnDQL.java" package ="(default package)" value ="1"/>
109         <Value name="printHelp" source ="UnDQL.java" package ="(default package)" value
                ="0"/>
110      </Values>
111   </Metric>
112   <Metric id = "LCOM" description ="Lack of Cohesion of Methods">
113      <Values per = "type" avg = "0" stddev = "0" max = "0">
114         <Value name="CobolSourcefileFilter" source ="CobolSourcefileFilter.java" package
                ="(default package)" value ="0"/>
115         <Value name="DirectoryFilter" source ="DirectoryFilter.java" package ="(default
                package)" value ="0"/>
116         <Value name="UnDQL" source ="UnDQL.java" package ="(default package)" value ="0"/>
117      </Values>
118   </Metric>
119   <Metric id = "CE" description ="Efferent Coupling">
120      <Values per = "packageFragment" avg = "0" stddev = "0" max = "0">
121         <Value name="(default package)" package ="(default package)" value ="0"/>
```

```
122         </Values>
123      </Metric>
124      <Metric id = "NSM" description ="Number of Static Methods">
125         <Values per = "type" total = "6" avg = "2" stddev = "2,828" max = "6">
126            <Value name="UnDQL" source ="UnDQL.java" package ="(default package)" value ="6"/>
127            <Value name="CobolSourcefileFilter" source ="CobolSourcefileFilter.java" package
                   ="(default package)" value ="0"/>
128            <Value name="DirectoryFilter" source ="DirectoryFilter.java" package ="(default
                   package)" value ="0"/>
129         </Values>
130      </Metric>
131      <Metric id = "RMD" description ="Normalized Distance">
132         <Values per = "packageFragment" avg = "0" stddev = "0" max = "0">
133            <Value name="(default package)" package ="(default package)" value ="0"/>
134         </Values>
135      </Metric>
136      <Metric id = "RMA" description ="Abstractness">
137         <Values per = "packageFragment" avg = "0" stddev = "0" max = "0">
138            <Value name="(default package)" package ="(default package)" value ="0"/>
139         </Values>
140      </Metric>
141      <Metric id = "SIX" description ="Specialization Index">
142         <Values per = "type" avg = "0" stddev = "0" max = "0">
143            <Value name="CobolSourcefileFilter" source ="CobolSourcefileFilter.java" package
                   ="(default package)" value ="0"/>
144            <Value name="DirectoryFilter" source ="DirectoryFilter.java" package ="(default
                   package)" value ="0"/>
145            <Value name="UnDQL" source ="UnDQL.java" package ="(default package)" value ="0"/>
146         </Values>
147      </Metric>
148      <Metric id = "WMC" description ="Weighted methods per Class">
149         <Values per = "type" total = "35" avg = "11,667" stddev = "12,284" max = "29">
150            <Value name="UnDQL" source ="UnDQL.java" package ="(default package)" value ="29"/>
151            <Value name="CobolSourcefileFilter" source ="CobolSourcefileFilter.java" package
                   ="(default package)" value ="4"/>
152            <Value name="DirectoryFilter" source ="DirectoryFilter.java" package ="(default
                   package)" value ="2"/>
153         </Values>
154      </Metric>
155      <Metric id = "NSF" description ="Number of Static Attributes">
156         <Values per = "type" total = "3" avg = "1" stddev = "1,414" max = "3">
157            <Value name="UnDQL" source ="UnDQL.java" package ="(default package)" value ="3"/>
158            <Value name="CobolSourcefileFilter" source ="CobolSourcefileFilter.java" package
                   ="(default package)" value ="0"/>
159            <Value name="DirectoryFilter" source ="DirectoryFilter.java" package ="(default
                   package)" value ="0"/>
160         </Values>
161      </Metric>
162   </Metrics>
```

Listing 12: Beispiel-XML-Daten des Eclipse Metrics Plug-Ins

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema targetNamespace="http://metrics.sourceforge.net/2003/Metrics-First-Flat"
    xmlns:m="http://metrics.sourceforge.net/2003/Metrics-First-Flat"
    xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="unqualified"
    attributeFormDefault="unqualified">
        <xs:complexType name="MetricType">
                <xs:choice>
                        <xs:element name="Values" type="m:ValuesType"/>
                        <xs:element name="Value" type="m:ValueType"/>
                </xs:choice>
                <xs:attribute name="id" type="xs:string" use="required"/>
                <xs:attribute name="description" type="xs:string" use="required"/>
                <xs:attribute name="max" type="xs:double"/>
                <xs:attribute name="hint" type="xs:string"/>
        </xs:complexType>
        <xs:complexType name="CycleType">
                <xs:sequence>
                        <xs:element name="Package" type="xs:string" maxOccurs="unbounded"/>
                </xs:sequence>
                <xs:attribute name="name" type="xs:string" use="required"/>
                <xs:attribute name="nodes" type="xs:integer" use="required"/>
                <xs:attribute name="diameter" type="xs:integer" use="required"/>
        </xs:complexType>
        <xs:element name="Metrics">
                <xs:complexType>
                        <xs:sequence>
                                <xs:element name="Cycle" type="m:CycleType" minOccurs="0"
                                    maxOccurs="unbounded"/>
                                <xs:element name="Metric" type="m:MetricType" minOccurs="0"
                                    maxOccurs="unbounded"/>
                        </xs:sequence>
                        <xs:attribute name="scope" type="xs:string" use="required"/>
                        <xs:attribute name="type" type="xs:string" use="required"/>
                        <xs:attribute name="date" type="xs:date" use="required"/>
                </xs:complexType>
        </xs:element>
        <xs:complexType name="ValueType">
                <xs:attribute name="name" type="xs:string" use="required"/>
                <xs:attribute name="value" type="xs:double" use="required"/>
                <xs:attribute name="source" type="xs:string"/>
                <xs:attribute name="package" type="xs:string"/>
                <xs:attribute name="inrange" type="xs:boolean" default="true"/>
        </xs:complexType>
        <xs:complexType name="ValuesType">
                <xs:sequence>
                        <xs:element name="Value" type="m:ValueType" minOccurs="0"
                            maxOccurs="unbounded"/>
                </xs:sequence>
                <xs:attribute name="per" type="xs:string"/>
                <xs:attribute name="total" type="xs:string"/>
                <xs:attribute name="avg" type="xs:string"/>
                <xs:attribute name="stddev" type="xs:string"/>
                <xs:attribute name="max" type="xs:string"/>
                <xs:attribute name="maxinrange" type="xs:boolean" default="true"/>
        </xs:complexType>
</xs:schema>
```

Listing 13: XML-Schema des Eclipse Metrics Plug-Ins

```
<?xml version="1.0" encoding="UTF-8"?>
<result xmlns="http://conqat.cs.tum.edu/ns/node" hideRoot="false">
  <key>LoC</key>
  <key>CR</key>
  <key>CR-Assessment</key>
  <node id="S:\debeka_cobol-non_dql\prod\dks\source\in\in">
    <description>S:\debeka_cobol-non_dql\prod\dks\source\in\in</description>
    <value key="LoC">27.788</value>
    <value key="CR" />
    <value key="CR-Assessment">RED</value>
    <node id="S:\debeka_cobol-non_dql\prod\dks\source\in\in\inc">
      <description>inc</description>
      <value key="LoC">1.187</value>
      <value key="CR" />
      <value key="CR-Assessment">RED</value>
      <node id="S:\debeka_cobol-non_dql\prod\dks\source\in\in\inc\kin001.cpy">
        <description>kin001.cpy</description>
        <value key="LoC">47</value>
        <value key="CR">0,463</value>
        <value key="CR-Assessment">GREEN</value>
      </node>
      <node id="S:\debeka_cobol-non_dql\prod\dks\source\in\in\inc\kin002.cpy">
        <description>kin002.cpy</description>
        <value key="LoC">41</value>
        <value key="CR">0,517</value>
        <value key="CR-Assessment">GREEN</value>
      </node>
    </node>
    <node id="S:\debeka_cobol-non_dql\prod\dks\source\in\in\pgm">
      <description>pgm</description>
      <value key="LoC">26.601</value>
      <value key="CR" />
      <value key="CR-Assessment">RED</value>
      <node id="S:\debeka_cobol-non_dql\prod\dks\source\in\in\pgm\main">
        <description>main</description>
        <value key="LoC">10.148</value>
        <value key="CR" />
        <value key="CR-Assessment">YELLOW</value>
        <node id="S:\debeka_cobol-non_dql\prod\dks\source\in\in\pgm\main\in1000.cbl">
          <description>in1000.cbl</description>
          <value key="LoC">832</value>
          <value key="CR">0,222</value>
          <value key="CR-Assessment">YELLOW</value>
        </node>
        <node id="S:\debeka_cobol-non_dql\prod\dks\source\in\in\pgm\main\in1010.cbl">
          <description>in1010.cbl</description>
          <value key="LoC">626</value>
          <value key="CR">0,4</value>
          <value key="CR-Assessment">YELLOW</value>
        </node>
        </node>
      </node>
      <node id="S:\debeka_cobol-non_dql\prod\dks\source\in\in\pgm\sub">
        <description>sub</description>
        <value key="LoC">7.573</value>
        <value key="CR" />
        <value key="CR-Assessment">YELLOW</value>
        <node id="S:\debeka_cobol-non_dql\prod\dks\source\in\in\pgm\sub\cin001.cbl">
          <description>cin001.cbl</description>
```

```
          <value key="LoC">321</value>
          <value key="CR">0,487</value>
          <value key="CR-Assessment">GREEN</value>
        </node>
        <node id="S:\debeka_cobol-non_dql\prod\dks\source\in\in\pgm\sub\cin002.cbl">
          <description>cin002.cbl</description>
          <value key="LoC">495</value>
          <value key="CR">0,472</value>
          <value key="CR-Assessment">GREEN</value>
        </node>
      </node>
      <node id="S:\debeka_cobol-non_dql\prod\dks\source\in\in\pgm\tpr">
        <description>tpr</description>
        <value key="LoC">8.880</value>
        <value key="CR" />
        <value key="CR-Assessment">RED</value>
        <node id="S:\debeka_cobol-non_dql\prod\dks\source\in\in\pgm\tpr\ia0001.cbl">
          <description>ia0001.cbl</description>
          <value key="LoC">3.267</value>
          <value key="CR">0,146</value>
          <value key="CR-Assessment">RED</value>
        </node>
        <node id="S:\debeka_cobol-non_dql\prod\dks\source\in\in\pgm\tpr\ia0009.cbl">
          <description>ia0009.cbl</description>
          <value key="LoC">1.402</value>
          <value key="CR">0,312</value>
          <value key="CR-Assessment">YELLOW</value>
        </node>
      </node>
    </node>
  </node>
</result>
```

Listing 14: Beispiel-XML-Daten (gekürzt) des XMLFileWriter-Processors von ConQAT

D XML-Schemata der Konfigurationsdateien

In diesem Anhang befinden sich die XML-Schemata der Konfigurationsdatei des Hauptprogramms (Listing 15), des Plug-Ins für SofAudit (Listing 16) und des Plug-Ins für ConQAT (Listing 17).

```
<?xml version="1.0" encoding="utf-8"?>
<xs:schema attributeFormDefault="unqualified" elementFormDefault="qualified"
    xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="configuration">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="main" minOccurs="1" maxOccurs="1">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="preprocessor" minOccurs="0" maxOccurs="1">
                <xs:complexType>
                  <xs:attribute name="path" type="xs:string" use="required" />
                  <xs:attribute name="workingdirectory" type="xs:string" use="optional" />
                </xs:complexType>
```

```
14              </xs:element>
15              <xs:element name="workingdirectory" minOccurs="0" maxOccurs="1">
16                <xs:complexType>
17                  <xs:attribute name="path" type="xs:string" use="required" />
18                </xs:complexType>
19              </xs:element>
20              <xs:element name="errorhandling" minOccurs="0" maxOccurs="1">
21                <xs:complexType>
22                  <xs:sequence />
23                  <xs:attribute name="faulttolerantmode" type="xs:boolean" use="optional" />
24                  <xs:attribute name="cleanuponerror" type="xs:boolean" use="optional" />
25                </xs:complexType>
26              </xs:element>
27              <xs:element name="sourcecodefiletypes" minOccurs="0" maxOccurs="1">
28                <xs:complexType>
29                  <xs:sequence>
30                    <xs:element maxOccurs="unbounded" name="filetype" minOccurs="1">
31                      <xs:complexType>
32                        <xs:attribute name="extension" type="xs:string" use="required" />
33                      </xs:complexType>
34                    </xs:element>
35                  </xs:sequence>
36                </xs:complexType>
37              </xs:element>
38              <xs:element name="ignoreddirectories" minOccurs="0" maxOccurs="1">
39                <xs:complexType>
40                  <xs:sequence>
41                    <xs:element maxOccurs="unbounded" name="directory" minOccurs="1">
42                      <xs:complexType>
43                        <xs:attribute name="name" type="xs:string" use="required" />
44                      </xs:complexType>
45                    </xs:element>
46                  </xs:sequence>
47                </xs:complexType>
48              </xs:element>
49            </xs:sequence>
50          </xs:complexType>
51        </xs:element>
52        <xs:element name="metrictools" maxOccurs="1" minOccurs="1">
53          <xs:complexType>
54            <xs:sequence>
55              <xs:element maxOccurs="unbounded" name="tool" minOccurs="1">
56                <xs:complexType>
57                  <xs:attribute name="name" type="xs:string" use="required" />
58                  <xs:attribute name="interopclass" type="xs:string" use="required" />
59                  <xs:attribute name="configfile" type="xs:string" use="optional" />
60                </xs:complexType>
61              </xs:element>
62            </xs:sequence>
63          </xs:complexType>
64        </xs:element>
65        <xs:element name="metricdataexporters" maxOccurs="1" minOccurs="1">
66          <xs:complexType>
67            <xs:sequence>
68              <xs:element maxOccurs="unbounded" name="exporter" minOccurs="1">
69                <xs:complexType>
70                  <xs:attribute name="class" type="xs:string" use="required" />
71                  <xs:attribute name="fileextension" type="xs:string" use="required" />
72                </xs:complexType>
73              </xs:element>
```

```
            </xs:sequence>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

Listing 15: XML-Schema für die Konfigurationsdateien des Hauptprogramms

```
<?xml version="1.0" encoding="utf-8"?>
<xs:schema xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    attributeFormDefault="unqualified" elementFormDefault="qualified"
    xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="configuration">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="main" maxOccurs="1" minOccurs="1">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="executable" minOccurs="1" maxOccurs="1">
                <xs:complexType>
                  <xs:attribute name="path" type="xs:string" use="required" />
                </xs:complexType>
              </xs:element>
              <xs:element name="inputfiletemplate" maxOccurs="1" minOccurs="1">
                <xs:complexType>
                  <xs:attribute name="path" type="xs:string" use="required" />
                  <xs:attribute name="functabpath" type="xs:string" use="required" />
                </xs:complexType>
              </xs:element>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
        <xs:element name="metrics" minOccurs="1" maxOccurs="1">
          <xs:complexType>
            <xs:sequence>
              <xs:element maxOccurs="unbounded" name="metric" minOccurs="1">
                <xs:complexType>
                  <xs:attribute name="name" type="xs:string" use="required" />
                  <xs:attribute name="description" type="xs:string" use="required" />
                  <xs:attribute name="unit" type="xs:string" use="required" />
                  <xs:attribute name="resultidstring" type="xs:string" use="required" />
                </xs:complexType>
              </xs:element>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

Listing 16: XML-Schema für die Konfigurationsdateien des SofAudit-Plug-Ins

```
<?xml version="1.0" encoding="utf-8"?>
<xs:schema xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    attributeFormDefault="unqualified" elementFormDefault="qualified"
    xmlns:xs="http://www.w3.org/2001/XMLSchema">
```

```
  <xs:element name="configuration">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="main" minOccurs="1" maxOccurs="1">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="conqatengine" minOccurs="1" maxOccurs="1">
                <xs:complexType>
                  <xs:attribute name="path" type="xs:string" use="required" />
                </xs:complexType>
              </xs:element>
              <xs:element name="inputfiletemplates" maxOccurs="1" minOccurs="1">
                <xs:complexType>
                  <xs:attribute name="blockfilepath" type="xs:string" use="required" />
                  <xs:attribute name="runconfigfilepath" type="xs:string" use="required" />
                </xs:complexType>
              </xs:element>
              <xs:element name="javavmparameters" maxOccurs="1" minOccurs="1">
                <xs:complexType>
                  <xs:sequence>
                    <xs:element maxOccurs="unbounded" name="parameter" minOccurs="0">
                      <xs:complexType>
                        <xs:attribute name="value" type="xs:string" use="required" />
                      </xs:complexType>
                    </xs:element>
                  </xs:sequence>
                </xs:complexType>
              </xs:element>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
        <xs:element name="metrics" maxOccurs="1" minOccurs="1">
          <xs:complexType>
            <xs:sequence>
              <xs:element maxOccurs="unbounded" name="metric" minOccurs="1">
                <xs:complexType>
                  <xs:attribute name="name" type="xs:string" use="required" />
                  <xs:attribute name="description" type="xs:string" use="required" />
                  <xs:attribute name="unit" type="xs:string" use="required" />
                  <xs:attribute name="resultkeyattribute" type="xs:string" use="required" />
                </xs:complexType>
              </xs:element>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

Listing 17: XML-Schema für die Konfigurationsdateien des ConQAT-Plug-Ins

E Gesamtklassendiagramm

Auf dem Poster in diesem Anhang befindet sich das Klassendiagramm des Hauptprogramms und aller implementierten Plug-Ins. Dabei sind alle Funktionen und Felder enhalten, einschließlich aller Parameter und Typen. Auch sind alle Assoziationen innerhalb der einzelnen Packages dargestellt, lediglich die Assozationen über Packagegrenzen hinweg wurden der Übersichtlichkeit halber weggelassen.

F Software auf CD

Auf der CD in diesem Anhang befinden sich:

- Die Quelltexte der im Rahmen dieser Arbeit entwickelten Software im Unterordner *src*
- Die Dokumentation der Quelltexte (Javadoc) im Unterordner *doc*
- Die Quelltexte (*.tex*) und die generierten Dokumente (*.pdf*) dieser Arbeit, aller entstandenen Teildokumente und der Sitzungsprotokolle - einschließlich der zugehörigen Bilder etc. - im Unterordner *thesis*
- Die .jar-Datei mit der kompilierten Version der Software, SVNKit 1.3.4 und ConQAT Engine 2.7 im Unterordner *lib*
- Die Shellskripte zum Setzen der benötigten Umgebungsvariablen und zum Ausführen der Software im Unterordner *bin*
- Die Beispielkonfigurationsdateien und -vorlagendateien für die Software und die Plug-Ins im Unterordner *examples*

Aufgrund einer entsprechenden Nichtveröffentlichungsvereinbarung mit der Debeka sind die zum Testen verwendeten COBOL-Quelltexte und der DQL-Präprozessor nicht auf dieser CD enthalten. Da es sich bei COBOL Flow Graph Manipulator (FGM) und SofAudit um kostenpflichtige Vollversionen handelt, sind diese aus urheberrechtlichen Gründen ebenfalls nicht auf dieser CD enthalten.

Literatur

[Basili1992] Basili, Victor R. (1992):
Software Modeling and Measurement: The Goal/Question/Metric Paradigm
`http://www.cs.umd.edu/~basili/publications/technical/T78.pdf`
Abruf: 02/2010

[Beier2009] Beier, Anja (2009):
Flow Graph Manipulator (FGM) 3.0: Reverse-Engineering-Werkzeug für komplexe Softwaresysteme
`http://www.proetcon.de/de/company/press/wsr_09_fgm.pdf`
Abruf: 03/2010

[Brown1999] Brown, Gary DeWard (1999): *COBOL: The failure that wasn't*
`http://www.cobolreport.com`
- Der Artikel ist nicht mehr im Netz verfügbar, wird jedoch u.a. in [Coughlan2002] referenziert. -

[CC2007] CC GmbH (2007):
AUDITOR für COBOL: Metrics Reference Manual, Version 1.7.2
-Dieses Dokument wird mit der Demoversion mitgeliefert und ist nicht im Internet verfügbar.

[Chapin1978] Chapin, Ned; Denniston, Susan P. (1978):
Characteristics of a structured program.
ACM SIGPLAN Notices, Version 13 No 5, S. 36-45

[Coughlan2002] Coughlan, Michael (2002): *Introduction to COBOL*
`http://www.csis.ul.ie/cobol/Course/COBOLIntro.htm`
Abruf: 02/2010

[Deißenböck2010] Deißenböck, Dr. Florian; Feilkas, Martin; Heinemann, Lars; Hummel, Benjamin; Jürgens, Elmar (2010):
ConQAT Book, Version 2.6
`http://conqat.cs.tum.edu/download/conqat-book.pdf`
Abruf: 03/2010

[Ebert1995] Ebert, Prof. Dr. Jürgen; Franzke, Angelika (1995):
A Declarative Approach to Graph Based Modeling
in: Mayr, E.; Schmidt, G.; Tinhofer, G.: Graphtheoretic Concepts in Computer Science, S.38-50.
Berlin: Springer, 1995.

[Ebert2009] Ebert, Prof. Dr. Jürgen; Riediger, Dr. Volker; Haas, Judith (2009):
COBOL-Bestandsanalyse und -Sanierung (Cobus)
http://www.uni-koblenz-landau.de/koblenz/fb4/institute/IST/AGEbert/projekte/cobus/
Abruf: 02/2010

[Haas2010-1] Haas, Judith; Doppler, Max (2010):
Anwendungs-,Werkzeuglandschaft und Entwicklungsprozess des Debeka-Kernsystems, Version 2.0
https://svn.uni-koblenz.de/agebert/cobus-project/5_berichte/52_anwendungslandschaft_tools/522_final/system_und_werkzeug_landschaft_v2.0.pdf
Abruf: 08/2010

[Haas2010-2] Haas, Judith (2010):
Metrik-Katalog, Version 1.0
https://svn.uni-koblenz.de/agebert/cobus-project/5_berichte/54_metriken/metrikkatalog_v1.0.pdf
Abruf: 12/2010

[Halstead1979] Halstead, Maurice Howard (1979):
Elements of software science. 3. Auflage
Oxford: Elsevier Science Ltd., 1979. ISBN 978-0444002051

[Henderson1996] Henderson-Sellers, Brian (1996):
Object-Oriented Metrics: Measures of Complexity.
New Jersey: Prentice Hall, 1996. ISBN 978-0132398725

[Henry1981] Henry, Sallie M.; Kafura, Dennis G. (1981):
Software structure metrics based on information flow.
IEEE Transactions on Software Engineering, SE-7(5), S. 510-518

[Martin1994] Martin, Robert C. (1994):
OO Design Quality Metrics.
http://www.objectmentor.com/resources/articles/oodmetrc.pdf
Abruf: 04/2010

[Martin2002] Martin, Robert C. (2002):
Agile Software Development: Principles, Patterns and Practices.
New Jersey: Prentice Hall, 2002. ISBN 978-0135974445

[Masak2005] Masak, Dieter (2005):
Moderne Enterprise Architekturen.
Berlin: Axel Springer Verlag, 2005. ISBN 978-3540229469

[McCabe1976] McCabe, Thomas J. (1976):
A complexity measure.
IEEE Transactions on Software Engineering, SE-2(4), S. 308-320.
`http://www.literateprogramming.com/mccabe.pdf`
Abruf: 10/2010

[McClure1978] McClure, Carma L. (1978):
A Model for Program Complexity Analysis.
3. International Conference on Software Engineering, May 1978, S. 149-157

[Mitchell2006] Mitchell, Robert L. (2006): *Cobol: Not Dead Yet*
`http://www.computerworld.com/s/article/266156/Cobol_Not_Dead_Yet`
Abruf: 02/2010

[proetcon2009] pro et con Innovative Informatikanwendungen GmbH (2009):
Flow Graph Manipulator (Handbuch), Version 2.4
- Dieses Dokument wurde in digitaler Form durch den Hersteller bereitgestellt. -

[Sneed2008-1] Sneed, Harry (2008): *Software Measurement*
Vorlesungunterlagen WS2008/2009, Universität Koblenz
`http://www.uni-koblenz-landau.de/koblenz/fb4/institute/IST/AGEbert/teaching/WS0809/softwaremeasurement`
Abruf: 02/2010

[Sneed2008-2] Sneed, Harry M. (2008):
SoftAudit - System Auditing Werkzeug, Version 1.2
- Dieses Dokument liegt der Software in digitaler Form mit bei. -

[Zotow2009] Zotow, Nikolai (2009): *Vergessene Welt*
iX 12/2009, S.92f; Hannover: Heise Verlag.