UNIVERSITÄT
KOBLENZ · LANDAU
Fachbereich 4: Informatik

Corpus Engineering

Bachelorarbeit

zur Erlangung des Grades eines Bachelor of Science
im Studiengang Informatik

vorgelegt von

David Klauer

Erstgutachter:	Prof. Dr. R. Lämmel Institut für Softwaretechnik
Zweitgutachter:	Ekatarina Pek Institut für Softwaretechnik

Koblenz, im Mai 2011

Erklärung

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

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

. . . . . .

(Ort, Datum) (Unterschrift)

INHALTSVERZEICHNIS

ABBILDUNGSVERZEICHNIS

KAPITEL 1

EINFÜHRUNG

1.1 Motivation

Die Analyse von Software benötigt einige wichtige Voraussetzungen, vor allem wenn es um eine große Zahl an Softwareprojekten geht. So wurde in der Arbeitsgruppe Softwaresprachen das Projekt *API–Analysis 2.0* gestartet. Ziel ist es die Verwendung und Wiederverwendung von Software und Softwarebibliotheken, sogenannten APIs, zu messen.

Dazu wird nicht nur geprüft, wie sich die Projekte beim Vorgang des Kompilierens verhalten. Die verschiedenen Compilation Units der Softwareprojekte selbst werden ebenfalls durchleuchtet. Aus den so gesammelten Fakten soll unter anderem festgestellt werden inwieweit bestehende Projekte selbst Softwarebibliotheken nutzen, respektive die Projekte solche Softwarebibliotheken und -schnittstellen für weitere Programme bereitstellen.

Um die Projekte zu analysieren, werden sie in Gruppen in sogenannte Corpora organisiert. In diese werden Softwareprojekte, bei denen von einer gewissen Größe und Codequalität ausgegangen werden kann, zusammengetragen und gemeinsam durchleuchtet. Dazu werden eine Reihe selbstentwickelter Tools genutzt, die Fakten über diese Softwareprojekte aufstellen und ausgeben.

1.2 Zentrale Aufgabenstellung

Angesichts der Heterogenität von Softwareprojekten zusammen mit der Menge an Projekten, die analysiert werden, ist es wichtig ein sicheres Rahmenwerk zu haben, in dem diese Analysen stattfinden können. Dazu zählt ein möglichst automatisierter Ablauf aller zu erledigender Aufgaben, wie das Kompilieren der Projekte und die anschließende Extraktion der Fakten, etc.

Weiterhin ist es unerlässlich, die Ergebnisse der einzelnen angewandten Tools gegeneinander abzugleichen. Nur so lässt sich prüfen, wie exakt diese Ergebnisse sind bzw. wie genau die Tools arbeiten.

Dazu soll als Teil dieser Thesis ein Tool implementiert werden, welches automatisiert die Ergebnisse aller Tools vergleicht und prüft, inwiefern sich diese Ergebnisse decken. Anschließend sollen die Fakten der faktextrahierenden Tools genauer untersucht werden. Es soll bestimmt werden, inwieweit diese übereinstimmen oder von einander abweichen.

Die Ergebnisse sollen dazu genutzt werden, um die eingesetzten Tools, in Hinsicht auf Umfang und Richtigkeit der von ihnen erzeugten Fakten zu verbessern.

1.3 Aufbau

Der Aufbau dieser Thesis orientiert sich an den Abläufen und Tools, die im Rahmen von *API–Analysis 2.0* entwickelt wurden. So wurden als Grundlage der Analysen einige Corpora angelegt, die teilweise in ähnlicher Form bereits existieren. Weiterhin wurden Tools implementiert, die Informationen über die Projekte liefern.

Der Aufbau und die Bestandteile eines Corpus werden in Kapitel 2 beschrieben. Ein Corpus besteht aus einer Reihe von Software-Projekten, sowie einigen Zusätzen.

Wenn alle Projekte zusammengetragen worden sind, wird ein BuildAnalyzer angewandt, mehr dazu im Abschnitt 3.1. Dieser sammelt Informationen über die verschiedenen Compilation Units der einzelnen Projekte, welche im nächsten Schritt von den Faktextraktoren genutzt werden.

Die Faktextraktoren, siehe Abschnitt 3.2, verwenden die Informationen des BuildAnalyzer. Sie untersuchen die Projekte und sollen den exakten Aufbau der

Projekte in Fakten ausgedrückt wiedergeben. Ausgedrückt werden die Fakten als Prolog-Fakten.[1] Insgesamt stehen drei verschiedene Faktextraktoren zur Verfügung. Jeder einzelne hat einen eigenen Ansatz die vorliegenden Projekte eines Corpus zu analysieren. Die Ergebnisse können also in verschiedener Hinsicht variieren.

Der FactComparator, siehe Kapitel 4, der als Teil dieser Thesis implementiert wurde, hat das Ziel die Vollständigkeit der erzeugten Fakten zu messen und weiterhin die Ergebnisse der einzelnen Faktextraktoren gegenseitig zu vergleichen, um eventuelle Fehler zu erkennen. Ein Teilziel war es weiterhin, die Resultate des FactComparator zu nutzen, um eventuelle Fehler in den Faktextraktoren auszubessern oder um sie präziser zu machen.

Im Kapitel 5 sind die Ergebnisse des FactComparator zusammengetragen. Diese Ergebnisse zu analysieren ist auch Teil dieses Kapitels. Weiterhin wird beschrieben, wie diese Ergebnisse bereits genutzt wurden und zu einer Verbesserung der Faktextraktoren geführt haben.

1.4 Grundlegende Begriffe

Die grundlegenden Begriffe werden in diesem Abschnitt zusammen aufgelistet, um ihre Bedeutung und Verwendung klarzustellen.

API
: ist die Kurzform für „Application Programming Interface“, zu deutsch etwa „Schnittstelle für Anwendungsprogramme“. Eine API ist im Grunde eine Programmbibliothek, welche Funktionen und Methoden für andere Anwendungen zur Verfügung stellt.

ASM
: wird in dieser Thesis als Kurzform und Synonym für den ASM-Based-FactExtractor verwendet.

BuildAnalyzer
: ist ein Programm, welches den Status von Projekten in einem Corpus vor und nach dem Build-Prozess analysiert, sowie Input-Daten für die Faktextraktoren aus diesen Daten generiert. Näheres in Abschnitt 3.1.

[1] http://en.wikipedia.org/wiki/Prolog#Rules_and_facts

Build-Prozess
beschreibt den Vorgang des Kompilierens eines Software-Projektes. Im Falle dieser Thesis, das Kompilieren von Java-Programmen.

Bytecode-Dateien
fassen Class- und Jar-Dateien zusammen.

Class-Dateien
bezeichnen in dieser Thesis die maschinenlesbaren Dateien eines Programms. Die Bezeichnung leitet sich aus der Programmiersprache Java ab, in der Programme in Dateien mit der Endung `.class` kompiliert werden.

Compilation Unit (CU)
ist der Überbegriff eines Programmteils, in Java etwa eine *Class* oder ein *Interface*.

Corpus
bezeichnet hier eine Sammlung von Java-Projekten, die so oder in ähnlicher Form bereits existieren. Als Beispiel sei hier der Corpus *JavaShape* genannt, der in [BFN+06] bereits definiert ist, was den Umfang und die Auswahl der Java-Projekte angeht.

Faktenbasis
wird hier als Begriff für eine Sammlung von Fakten in Form einer Datei verwendet. In den meisten Fällen handelt es sich hierbei um Prolog-Fakten.

FactComparator (FC)
ist die Bezeichnung eines Tools, welches im Rahmen dieser Thesis entwickelt wurde, um die Fakten die vom BuildAnalyzer und den Faktextraktoren zu analysieren und zu vergleichen.

Faktextraktor
ist die Verallgemeinerung der einzelnen Faktextraktoren, die genutzt werden, also der Begriff der den Recoder-Based-FactExtractor und den ASM-Based-FactExtractor zusammenfasst.

Jar-Dateien
sind eine Form in die Java den gegebenen Quell-Code der Source-Dateien

kompiliert. Meist umfasst eine Jar-Datei viele Compilation Units, im Gegensatz zu den Class-Dateien, die jeweils nur eine umfassen.

RECODER

bezeichnet ein Java-Framework, welches an der Universität Karlsruhe entwickelt wurde. Mittlerweile wird es von der Linnareaus University in Växjö, Schweden, weiterentwickelt. [2]

Repository

ist die Bezeichnung für einen Ablageort für Projekte. Mittels Versionierung lassen sich Änderungen an Software hier nachverfolgen, um beispielsweise einzelne Programmteile einzeln zu aktualisieren.

RFE

wird in dieser Thesis ebenfalls als Kurzform und Synonym für den Recoder-Based-FactExtractor verwendet.

Source-Dateien

sind der für Menschen lesbare Teil eines Java-Programms. Aus ihnen werden die maschinenlesbaren Class- oder Jar-Dateien kompiliert.

[2] Quelle: Webseite des RECODER Projektes, aufgerufen am 03.05.2011
http://sourceforge.net/apps/mediawiki/recoder/index.php?title=Recoder_History

KAPITEL 2

DER CORPUS

Wie bereits beschrieben, werden die zu analysierenden Java-Projekte in sogenannten Corpora organisiert. Die Auswahl der Projekte ist dabei freigestellt. Damit ein Corpus übersichtlich bleibt, bietet es sich an, Software-Projekte zu kategorisieren. Einen Corpus etwa für Software-Projekte, die sich mit XML befassen und einen anderen für Datenbanken-Projekte.

Eine weitere Möglichkeit besteht darin, bereits existierende Sammlungen von Software-Projekten aufzugreifen und aus ihnen einen Corpus zu erstellen. Als Beispiel sei hier der Corpus *JavaShape* genannt. Die Auswahl der Projekte orientiert sich dabei an [BFN+06]. Insgesamt beinhaltet dieser Corpus, der für diese Thesis verwendet wird, momentan 54 Projekte. Welche das genau sind, ist aus den CSV-Tabellen im Anhang A, beispielsweise Tabelle A.1, ersichtlich.

Ein weiterer so zusammengestellter Corpus umfasst eine Auswahl von Java-Projekten der Apache-Software-Foundation.[1] Der sogenannte Corpus *Apache* beinhaltet dabei Projekte, die, Stand Juni 2010, im öffentlichen Subversion-Repository vorhanden waren.[2] Gewählt wurden ausschließlich die Projekte, die in Java implementiert wurden und über die Informationen zum Kompiliervorgang vorlagen. Zu den so gewonnenen 23 Projekten zählen unter anderem: `abdera`, `axis2`, `cassandra`, `log4j` und `maven`.

[1] Webseite der Apache Software Foundation:
http://www.apache.org/

[2] Das öffentliche Software-Repository des Apache Software Foundation:
http://svn.apache.org/repos/asf/

2.1 Bestandteile eines Corpus

Bei früheren Forschungen traten einige Probleme auf. So ließen sich einige Projekte nach einem Build-Prozess nicht mehr in den Ausgangszustand zurücksetzen. Zu diesem Zweck wurde eine Methode entwickelt, welche die „cleanen“ Projekte, also solche in ihrem Ausgangszustand, etwa frisch nach dem Herunterladen, in einem separaten Verzeichnis vorhält. Soll ein Projekt kompiliert, werden, wird es in ein dafür vorgesehenes Verzeichnis kopiert. So kann garantiert werden, dass zu jedem Zeitpunkt an dem ein Projekt gebaut werden soll, die Bedingungen herrschen, die auch beim ersten Kompilieren vorgeherrscht haben. Damit sind die Ergebnisse reproduzierbar.

Viele Projekte benötigen zum Kompilieren mehr als nur ihre eigenen mitgelieferten Source-Dateien, zum Beispiel weitere Frameworks und Bibliotheken dritter. Würden diese einfach in die Verzeichnisse der Projekte eingefügt, wäre der Originalzustand nicht mehr gegeben. So wurde entschieden, eine gemeinsame Sammlung aller von einem beliebigen Projekt benötigten Bibliotheken, bzw. APIs, anzulegen, der sogenannte API-Pool, mehr dazu in Abschnitt 2.2.

Soll ein Corpus, wie er hier genutzt wird, erstellt werden, wird eine bestimmte Grundstruktur der Verzeichnisse, wie sie in Abbildung 2.1 aufgezeigt wird, benötigt.

```
corpus-root/
 |- builds/
 |- deltas/
 |- sources/
```

Abbildung 2.1: Verzeichnisstruktur eines initialen Corpus

Im Verzeichnis `sources` befindet sich für jedes Projekt eines Corpus ein eigenes Projektverzeichnis. Dieses hält die entsprechenden Source-Dateien des Projekts im Originalzustand bereit. Weiterhin liefert eine XML-Datei pro Projekt unter anderem Informationen über Herkunft der Projekte, also wo sie heruntergeladen wurden, wann das Projekt geladen wurde und in welcher Version das Projekt vorliegt. Benötigte ein Projekt zum Bau externe APIs oder sonstige Dateien, so ist dies ebenfalls in der Projekt-XML kommentiert.

Wenn für ein Projekt weitere Dateien als die mitgelieferten Source-Dateien nötig sind, so wird im Verzeichnis `deltas` für dieses Projekt ein Python-Skript `patchit.py` bereitgestellt. Dieses hat die Aufgabe, den Projekten die fehlenden APIs zukommen zu lassen. Ein Beispiel eines solchen Skripts zeigt Abbildung 2.2.

```
1  #!/usr/bin/python
2  # -*- coding: utf-8 -*-
3
4  import shutil
5
6  descriptor = "ant"
7  path_to_apipool = "../../../apipool/"
8  path_dest = "../../builds/" + descriptor + "/"
9  cs = []
10 cs.append(("junit3.8.2.jar", "./lib/"))
11 for file, location in cs:
12     shutil.copy(path_to_apipool + file, path_dest + location)
```

Abbildung 2.2: Beispiel eines `patchit.py`-Skript anhand des `ant`-Projekts

Ein Build-Skript kopiert die ursprünglichen Source-Dateien (aus `sources`) in das Verzeichnis `builds` und wendet die in `deltas` vorliegenden Skripte an. Anschließend werden die einzelnen Projekte gebaut. So ist sichergestellt, dass die Projekte alle benötigten Bibliotheken vor Ort haben und sie kompiliert bzw. gebaut werden können.

Weitere Verzeichnisse, wie etwa `facts` und `logs` werden vom Recoder-Based-FactExtractor erstellt und genutzt. Hier finden sich die extrahierten Fakten, sowie Log-Dateien der einzelnen Ausführungen. Der ASM-Based-FactExtractor benötigt ein Verzeichnis `configs`, indem verschiedene Konfigurationsdateien abgelegt werden, sowie ein Verzeichnis `facts_ASMTool`. Dieses wird als Ausgabeort des Tools verwendet.

Die volle Ausbaustufe eines Corpus auf den alle Tools angewendet wurden, ist in Abbildung 2.3 dargestellt.

```
corpus-root/
 |- buildAnalyzer/
     |- configForFAs/
 |- builds/
 |- configs/
 |- deltas/
 |- factComparator/
     |- images/
     |- out/
 |- facts/
     |- rfe/
     |- errors/
 |- facts_ASMTool/
 |- logs/
 |- sources/
     |- project1/
     |- project1.xml
     |- ...
```

Abbildung 2.3: Verzeichnisstruktur eines voll ausgebauten Corpus

2.2 Der API-Pool

Für die Corpora steht ein separater API-Pool zur Verfügung. Jegliche APIs die von einem der Projekte benötigt werden und nicht in ihren Auslieferungszuständen vorhanden sind, werden hier bereitgestellt. Aktuell sind dies 73 APIs. Teilweise befinden sich APIs mehrfach, jedoch in unterschiedlichen Versionen im Pool. Der aktuell komplette API-Pool wird in Tabelle 2.1 aufgeführt.

CCTools.jar	compiereRoot.jar	junit-4.8.2.jar
CInstall.jar	db2.jar	junit.jar
CSTools.jar	excalibur-datasource-1.1.1.jar	junit3.8.2.jar
Compiere.jar	excalibur-instrument-1.0.jar	jython-2.1.jar
CompiereInstall.jar	excalibur-logger-1.1.jar	logkit-1.2.jar
Tidy.jar	excalibur-pool-1.2.jar	midpapi10.jar
antlr.jar	htmllexer-2.0-20060923.jar	objenesis.jar
avalon-framework-4.1.4.jar	htmlparser-2.0-20060923.jar	oracle.jar
bsf-2.4.0.jar	ibmWAS.jar	postgreSQL.jar
bsh-2.0b4.jar	j2ee.jar	recoderKey.jar
bshclient.jar	jCharts-0.7.5.jar	serializer-2_9_1.jar
cldcapi10.jar	jai_codec.jar	servlet.jar
commons-codec-1.3.jar	jai_core.jar	soap.jar
commons-codec-1.4.jar	jakarta-oro-2.0.8.jar	sqlServer.jar
commons-collections-3.2.1.jar	javacc-3.2.jar	sqlj.jar
commons-collections-3.2.jar	javacc.jar	sqljDB2.jar
commons-httpclient-3.1.jar	jboss.jar	swtgraphics2d.jar
commons-io-1.4.jar	jdbc2_0-stdext.jar	xalan_2_7_1.jar
commons-jexl-1.1.jar	jdic.jar	xercesImpl-2_9_1.jar
commons-lang-2.4.jar	jdom-1.1.jar	xml-apis-2_9_1.jar
commons-logging-1.0.2.jar	jfreechart-1.0.8-swt.jar	xmlgraphics-commons-1.3.1.jar
commons-logging-1.1.1.jar	jmf.jar	xpp3_min-1.1.4c.jar
commons-net-1.4.1.jar	jorphan.jar	xstream-1.3.1.jar
commons-primitives-1.0.jar	js_rhino1_6R5.jar	
compiereApps.jar	jta1.0.1.jar	

Tabelle 2.1: Die 73 APIs des Pools

KAPITEL 3

TOOLS

Dieses Kapitel soll dazu dienen, einen kurzen Überblick über die verwendeten Tools, welche im Rahmen der Forschungsarbeiten entwickelt wurden, zu verschaffen. Ziel der Bemühungen ist es, soviel Informationen aus den gegebenen Java-Projekten zu ziehen wie möglich. In den Abschnitten dieses Kapitels werden die einzelnen Tools, sowie deren Arbeitsweisen näher erläutert.

Zunächst sei jedoch der komplette Ablauf und die Verkettung der einzelnen Tools grob zu skizzieren:

1. Ausführen des BuildAnalyzer, siehe Abschnitt 3.1, zum Sammeln von grundlegenden Informationen über die Java-Projekte, bzw. deren Compilation Units

2. Aufbereiten dieser Informationen für die Faktextraktoren

3. Ausführen des Recoder-Based-FactExtractors, siehe Abschnitt 3.2.1, mit den von BuildAnalyzer gelieferten Ergebnissen

4. Ausführen des ASM-Based-FactExtractors, siehe Abschnitt 3.2.2, mit den von BuildAnalyzer gelieferten Ergebnissen

3.1 BuildAnalyzer

Der BuildAnalyzer ist ein Tool, welches in einen Corpus, der Form wie sie in Kapitel 2 beschrieben wird, integriert, das Ziel hat, das Verhalten der einzelnen Projekte während des Build-Prozesses zu analysieren und zu dokumentieren.[1] Es wird geprüft, inwiefern sich Java-Projekte in Class- oder in Jar-Dateien kompilieren. Weiterhin wird analysiert, ob während des Build-Prozesses externe APIs oder zumindest Teile von diesen heruntergeladen und genutzt werden. Ebenso wird protokolliert, wenn sich Projekte während des Kompilierens selbstständig die neuesten Versionen aus öffentlichen Software-Repositories beziehen. Und schließlich wird berechnet, zu welchen Teilen die kompilierten Ergebnisse die gegebenen Source-Dateien abdecken.

Als sinnvolles Nebenprodukt liefert der BuildAnalyzer Daten über die Compilation Units der einzelnen Java-Projekte. Diese Daten nutzen die Faktextraktoren als Konfigurationsdateien. Durch sie erfahren die Faktextraktoren, welche Java-Dateien, welche Class-Dateien und welche Jar-Dateien von den verschiedenen Java-Projekten genutzt werden. Alle hier aufgeführten Dateien werden von den Faktextraktoren dann genauer analysiert.

Der grobe Ablauf über die Schritte des BuildAnalyzer soll hier vereinfacht aufgeführt werden:

- Alle Source-Dateien der einzelnen Java-Projekte im gegebenen Corpus werden bezogen
- In einem ersten Durchlauf werden Daten über den Status der Java-Projekte vor Ausführung des Build-Prozess gesammelt
- Der Build-Prozess wird intern für jedes der Java-Projekte ausgeführt
- Während des Build-Prozesses wird protokolliert, wenn Java-Projekte sich aktualisieren bzw. wenn so vorgesehen, fehlende Teile geladen werden
- In einem zweiten Durchlauf werden erneut Daten über den Status der einzelnen Java-Projekte, diesmal nach dem Build-Prozess, gesammelt

[1] Die Source-Dateien zum BuildAnalyzer findet man hier:
https://svn.uni-koblenz.de/laemmel/apianalysis/src/buildsAnalyzer/

- Die Daten der beiden Durchläufe werden verglichen
- Es wird berechnet, inwiefern die Bytecode-Dateien (sowohl Class- als auch Jar-Dateien) die anfangs gegebenen Java-Dateien abdecken
- Die Konfigurationsdateien für die verschiedenen Faktextraktoren werden aus den gesammelten Daten erstellt

Gespeichert werden die zwischendurch ermittelten Daten in einer SQL-Datenbank. Aus dieser heraus werden zum Schluss die Konfigurationsdateien für die genutzten Faktextraktoren erstellt. Hauptbestandteil dieser Konfigurationsdateien sind die Compilation Units der zu analysierenden Java-Projekte. Die Abbildungen 3.1, 3.2, 3.3 und 3.4 zeigen die Konfigurationsdateien für ein Demo-Projekt `proc`.

```
1 src/org/gui/Overview.java
2 src/org/proc/Book.java
3 src/org/proc/Buyable.java
4 src/org/proc/Consumer.java
5 src/org/proc/Producer.java
6 src/org/proc/Product.java
7 src/org/proc/School.java
8 src/org/Program.java
```

Abbildung 3.1: Konfigurationsdatei für RFE

```
1 java(['src', 'org', 'gui'], 'Overview').
2 java(['src', 'org', 'proc'], 'Book').
3 java(['src', 'org', 'proc'], 'Buyable').
4 java(['src', 'org', 'proc'], 'Consumer').
5 java(['src', 'org', 'proc'], 'Producer').
6 java(['src', 'org', 'proc'], 'Product').
7 java(['src', 'org', 'proc'], 'School').
8 java(['src', 'org'], 'Program').
```

Abbildung 3.2: Konfigurationsdatei für den FC in Bezug auf RFE

```
build/org/gui/Overview.class
build/org/gui/Overview$1.class
build/org/proc/Book.class
build/org/proc/Buyable.class
build/org/proc/Consumer.class
build/org/proc/Producer.class
build/org/proc/Producer$1.class
build/org/proc/Product.class
build/org/proc/Product$PID.class
build/org/proc/School.class
build/org/Program.class
```

Abbildung 3.3: Konfigurationsdatei für ASM

```
class(['src', 'org', 'gui'], 'Overview').
class(['src', 'org', 'gui'], 'Overview$1').
class(['src', 'org', 'proc'], 'Book').
class(['src', 'org', 'proc'], 'Buyable').
class(['src', 'org', 'proc'], 'Consumer').
class(['src', 'org', 'proc'], 'Producer').
class(['src', 'org', 'proc'], 'Producer$1').
class(['src', 'org', 'proc'], 'Product').
class(['src', 'org', 'proc'], 'Product$PID').
class(['src', 'org', 'proc'], 'School').
class(['src', 'org'], 'Program').
```

Abbildung 3.4: Konfigurationsdatei für FC in Bezug auf ASM

3.2 Faktextraktoren

Die Faktextraktoren dienen dazu so viel wie möglich über einzelne Java-Projekte zu erfahren. Sie durchleuchten die gegebenen Compilation Units etwa der Source-Dateien, oder der kompilierten Bytecode-Dateien. Hierbei werden alle vorgefundenen Daten in Prolog-Fakten ausgedrückt in eine Faktenbasis gespeichert.

Da der Faktextraktor, der auf ein Java-Compiler-Plugin aufsetzt, momentan einige Probleme hat und nicht lauffähig ist, werden in dieser Thesis lediglich die beiden anderen verwendet und erläutert.

3.2.1 Recoder-Based-FactExtractor

Der Recoder-Based-FactExtractor, kurz auch RFE, wurde im Rahmen eines Projektpraktikums implementiert.

Kernstück des Recoder-Based-FactExtractor ist ein Java-Framework mit dem Namen *RECODER*. Daher leitet sich auch die Bezeichnung dieses Faktextraktors ab. Die RECODER-Bibliothek wurde ursprünglich unter dem Namen *COMPOST*, the COMPOsition SysTem, an der Universität Karlsruhe entwickelt. Im Jahr 2001 wurde der Name zu RECODER geändert. Seit 2007 wird es an der Linnareus University in Växjö, Schweden, weiter betreut und entwickelt. [2]

Einfach ausgedrückt ist RECODER ein Tool, welches Java-Source-Dateien parsed und intern in einem Meta-Modell vorhält. Dieses Meta-Modell kann auf mehrere Arten genutzt werden. Zum Beispiel lassen sich komplexe Java-Projekte um Funktionen erweitern, indem das Meta-Modell um diese Funktionen erweitert wird und anschließend werden durch unparsen die erweiterten Java-Source-Dateien generiert. [3]

Der Recoder-Based-FactExtractor nutzt das Meta-Modell auf eine andere ebenfalls vorgesehene Art und Weise. Um mit dem Meta-Modell Programme zu ändern, analysiert RECODER die Java-Source-Dateien nicht nur einzeln. Vielmehr müssen alle zu einem Java-Projekt gehörenden Dateien angegeben sein, damit alle internen Abhängigkeiten komplett aufgelöst werden können. So werden nicht nur ihre individuellen Strukturen und Komponenten durchleuchtet, sondern eben

[2] Quelle: http://sourceforge.net/apps/mediawiki/recoder/index.php?title=Recoder_History aufgerufen am 03.05.2011

[3] Quelle: http://recoder.sourceforge.net/index_old.html aufgerufen am 03.05.2011

auch komplexe Abhängigkeiten können nachvollzogen werden. So lassen sich nicht nur Aussagen über existierende Klassen, Felder, Methoden machen. Ebenfalls wird es möglich Methodenaufrufe genau zu bestimmen und in Fakten auszudrücken. Wie diese Fakten genau aussehen wird im Unterabschnitt 3.2.3 an einem Beispiel verdeutlicht.

3.2.2 ASM-Based-FactExtractor

Als weiterer Faktextraktor wurde der ASM-Based-FactExtractor, kurz ASM, entwickelt. Der ASM-Based-FactExtractor nutzt, wie der Recoder-Based-FactExtractor, ebenfalls ein bereits existierendes Framework als Grundlage. Das namensgebende Framework „ASM ist ein universell einsetzbares Framework zur Manipulation und Analyse von Java Bytecode.“[4]

Im Rahmen von *API–Analysis 2.0* wird der ASM-Based-FactExtractor für mehrere Zwecke eingesetzt. Zum einen analysiert er Projekte, die nicht als Quell-Dateien, sondern nur als Bytecode vorliegen, zum anderen können mit dem ASM-Based-FactExtractor die als Jar-Dateien vorliegenden APIs im API-Pool analysiert werden.

Gesteuert wird der ASM-Based-FactExtractor über Eingabeparameter und Konfigurationsdateien. Zur Auswahl stehen mehrere, verschiedene Modi. So können beispielsweise ausgewählte APIs respektive Projekte, die es zu untersuchen gilt, explizit angegeben werden, oder etwa der ASM-Based-FactExtractor analysiert alle Projekte eines Corpus, für die vom BuildAnalyzer Konfigurationsdateien vorgegeben werden.

Die Ausgabe der Fakten geschieht wie beim Recoder-Based-FactExtractor in Form von Prolog-basierten Faktenbasen. Der Aufbau dieser Faktenbasen, sowie die Signatur der einzelnen Fakten ist denen des Recoder-Based-FactExtractors angeglichen. Die Signaturen der Fakten werden im nächsten Abschnitt näher erläutert.

3.2.3 Die generierten Fakten

Wie bereits mehrfach erwähnt, werden alle analysierten Programmteile als Prolog-Fakt ausgedrückt in den Faktenbasen abgelegt. Die Signatur der einzelnen

[4]Quelle: `http://asm.ow2.org/` aufgerufen am 03.05.2011

Faktentypen wurde im Rahmen dieser Thesis verbessert. So wurde unter anderem der Typ `field` um das Attribut `FieldType` erweitert.

Die beiden oben beschriebenen Faktextraktoren nutzen das gleiche Schema, um ihre gesammelten Daten auszudrücken. In Abbildung 3.5 ist eine Liste der Fakttypen mitsamt ihren Attributen aufgeführt. Im weiteren werden dann die einzelnen Fakttypen erläutert und klargestellt, was die Attribute repräsentieren. Die Abbildung 3.6 zeigt an Ausschnitten des Projekts `hsqldb` beispielhaft, wie die Prologfakten selbst aussehen.

```
1 package(ID , QualifiedPackageName).
2 class(ID, QualifiedPackageName, ClassName, SuperClass,
      Implements, Modifiers).
3 field(ID, ContainingClassName, FieldType, FieldName, Modifiers).
4 method(ID, ContainingClassName, MethodName, Modifiers,
      ReturnType, ArgumentsList).
5 methodCall(ID, ContainingMethodName, CalledMethodName,
      ArgumentsList).
6 localVariable(ID, ContainingMethodName, VariableType,
      VariableName, Modifiers).
7 interface(ID, QualifiedPackageName, InterfaceName, ExtendsList,
      Modifiers).
```

Abbildung 3.5: Liste der Faktentypen und deren Inhalte

Das Attribut `ID`, welches in jeder der verschiedenen Typen zu finden ist, wird von Faktextraktoren unterschiedlich besetzt. Es dient der Identifikation von Fakten bzw. soll durch dieses Attribut ein Fakt einem Projekt zugeordnet werden können, sollte es zwei gleiche Fakten in Faktenbasen unterschiedlicher Projekte geben.

1. `package` wird für jedes gefundene Java-Package angelegt.
   - `QualifiedPackageName` enthält die Pakethierarchie als Prologliste
2. `class` wird für jede gefundene Klasse angelegt.
   - `QualifiedPackageName` enthält die Pakethierarchie als Prologliste
   - `ClassName` enthält den Bezeichner der Klasse
   - `SuperClass` enthält den qualifizierten Bezeichner der Super-Klasse als Prologliste

- `Implements` enthält den qualifizierten Bezeichner aller implementierten Interfaces als Liste von Prologlisten
- `Modifiers` enthält alle Modifier als Prologliste

3. `field` wird für jedes gefundene Feld angelegt.
   - `ContainingClassName` enthält den qualifizierten Bezeichner der Klasse, in der sich das Feld befindet, als Prologliste
   - `FieldType` enthält den qualifizierten Bezeichner des Feldtyps als Prologliste
   - `FieldName` enthält den Bezeichner des Feldes
   - `Modifiers` enthält alle Modifier als Prologliste
4. `method` wird für jede gefundene Methode angelegt.
   - `ContainingClassName` enthält den qualifizierten Bezeichner der Klasse, in der sich die Methode befindet, als Prologliste
   - `MethodName` enthält den Bezeichner der Methode
   - `Modifiers` enthält alle Modifier als Prologliste
   - `ReturnType` enthält den qualifizierten Bezeichner des Rückgabetyps als Prologliste
   - `ArgumentsList` enthält alle qualifizierten Bezeichner aller Argumente als Liste von Prologlisten
5. `methodCall` wird für jeden gefundenen Methodenaufruf angelegt.
   - `ContainingMethodName` enthält den qualifizierten Bezeichner der Methode, in der sich der Methodenaufruf befindet, als Prologliste
   - `CalledMethodName` enthält den qualifizierten Bezeichner der aufgerufenen Methode
   - `ArgumentsList` enthält alle qualifizierten Bezeichner aller Argumente als Liste von Prologlisten
6. `localVariable` wird für jede gefundene lokale Variable angelegt.

- `ContainingMethodName` enthält den qualifizierten Bezeichner der Methode, in der sich die lokale Variable befindet, als Prologliste
- `VariableType` enthält den qualifizierten Bezeichner des Variablentyps als Prologliste
- `VariableName` enthält den Bezeichner der Variable
- `Modifiers` enthält alle Modifier als Prologliste

7. `interface` wird für jedes gefundene Interface angelegt.
   - `QualifiedPackageName` enthält die Pakethierarchie als Prologliste
   - `InterfaceName` enthält den Bezeichner des Interfaces
   - `ExtendsList` enthält den qualifizierten Bezeichner aller erweiterten Interfaces als Liste von Prologlisten
   - `Modifiers` enthält alle Modifier als Prologliste

```
1 package('hsqldb', ['org','hsqldb']).
2 class('hsqldb', ['org','hsqldb'], 'BaseMemoryNode',
     ['org','hsqldb','Node'], [], ['abstract']).
3 field('hsqldb', ['org','hsqldb','BaseMemoryNode'],
     ['org','hsqldb','Node'], 'nLeft', ['protected']).
4 method('hsqldb', ['org','hsqldb','BaseMemoryNode'],
     'isFromLeft', [], ['boolean'], []).
5 methodCall('hsqldb',
     ['org','hsqldb','BaseMemoryNode','isFromLeft'],
     ['org','hsqldb','BaseMemoryNode','isRoot'], []).
6 localVariable('hsqldb',
     ['org','hsqldb','BaseMemoryNode','isFromLeft'],
     ['org','hsqldb','Node'], 'parent', []).
7 interface('hsqldb', ['org','hsqldb'], 'GrantConstants',
     [['java','lang','Object']], ['public']).
```

Abbildung 3.6: Liste der Faktentypen und deren Inhalte am Beispiel von `hsqldb`

KAPITEL 4

FACTCOMPARATOR

4.1 Ziele des FactComparator

Die Projekte in einem Corpus sollen so genau wie möglich durchleuchtet werden. Tools zu diesem Zweck wurden im Kapitel 3 bereits vorgestellt. Jedes der Tools liefert Daten bzw. Fakten. Doch steht noch nicht fest, inwieweit diese Daten und Fakten den gegebenen Projekten entsprechen. Und weiterhin, ob tatsächlich alle Bereiche der Projekte durch die Faktextraktoren analysiert wurden. Zu diesem Zweck wurde als Teil dieser Thesis ein Tool unter dem Namen „FactComparator“ entwickelt.

Welche Aufgaben soll der FactComparator also erfüllen?

- Die Vollständigkeit der Fakten berechnen, die die Faktextraktoren liefern
- Die Faktenbasen der Faktextraktoren untereinander vergleichen.
  - Werden die gleichen Compilation Units gefunden?
  - Werden die gleichen Felder, Methoden und Methodenaufrufe erkannt?
  - Wenn es Unterschiede gibt, wie sehen diese aus?
- Alle erhobenen Informationen, beispielsweise in Form von Tabellen, wiedergeben

Welche Ziele sollen durch der Einsatz des FactComparators erreicht werden?

- Verbesserungsmöglichkeiten der Faktextraktoren aufzeigen, um etwa die Vollständigkeit der Analysen zu maximieren
- Eine Angleichung der Ausgabe und Art der Fakten, welche von den Faktextraktoren geliefert werden

4.2 Aufbau des FactComparator

Der FactComparator besteht aus einer Reihe von Python- und Prolog-Skripten, welche zusammen die eben erwähnten Aufgaben ausführen. Die Prolog-Skripte dienen dem Umgang mit den Faktenbasen. Diese müssen eingelesen werden, damit mit den Fakten gearbeitet werden kann. Mehr dazu im Unterabschnitt 4.2.2 dieses Kapitels.

Die Python-Skripte kümmern sich unter anderem darum, dass die richtige Umgebung für den Einsatz des FactComparator geschaffen wird. Die Konfiguration des FactComparator erfolgt über XML-Dateien, die beim Starten über eine Konsole als Argument mitgegeben werden. Näheres dazu folgt im Unterabschnitt 4.2.1.

Nun soll erstmal auf den generellen Aufbau und die Verzeichnishierarchie des FactComparator eingegangen werden, siehe Tabelle 4.1. Im Hauptverzeichnis des FactComparator befinden sich anfangs lediglich die Python-Skripte, sowie ein Ordner `Prolog` mit den Prolog-Skripten und ein Ordner `StarterXMLs`, welcher die XML-Konfigurationsdateien der verschiedenen Corpora enthält.

```
corpus-root/
 |- factComparator/
     |- Prolog/
     |- StarterXMLs/
         |- corpus_root.xml
         |- ...
```

Abbildung 4.1: Initiale Verzeichnisstruktur des FactComparator

4.2.1 Der Python-Part

Das Hauptskript `factComparator.py` ist in erster Linie dafür zuständig, alle einzelnen Arbeitsschritte zu organisieren und nacheinander auszuführen.

Ein erster Schritt besteht darin, die Informationen zum Corpus, der untersucht werden soll, einer Konfigurationsdatei zu entnehmen. Zusammen mit dem `dirs.py`-Skript werden dann das Arbeitsverzeichnis `tmp`, das Ausgabeverzeichnis `out`, als auch ein `backup` und `logs` Verzeichnis erstellt. Diese dienen der Ablage von Ergebnissen und Log-Dateien früherer Durchläufe des FactComparator. Im Arbeitsverzeichnis und Ausgabeverzeichnis werden vorsorglich für jedes Projekt des Corpus ein Ordner mit der Projektbezeichnung als Namen erstellt.

Sind diese Vorkehrungen getroffen, beginnt der FactComparator damit die Daten der in Kapitel 3 vorgestellten Tools zu sammeln. Sie werden in das Arbeitsverzeichnis `tmp` kopiert. Ist dies geschehen, befinden sich die Faktenbasen des Recoder-Based-FactExtractor und des ASM-Based-FactExtractor, sowie die Konfigurationsdateien des BuildAnalyzer sowohl für die einzelnen Faktextraktoren als auch den FactComparator, an einem Ort.

Nun werden die verschiedenen Prolog-Skripte nacheinander ausgeführt. Dazu prüft das Hauptskript, welche Ergebnisse der einzelnen Tools vorliegen und dementsprechend, welche Ergebnisse verglichen werden müssen. Das Ausführen des Prolog-Skripts geschieht durch das Erstellen eines Konsolenaufrufes von SWI-Prolog. Er enthält Informationen darüber, welcher Teil des Prolog-Part aufgerufen und welches Goal abgearbeitet werden soll.

Wurden alle Schritte im Prolog-Part ausgeführt, erstellt `charts.py` mit Hilfe von „pygooglechart“, siehe [Kas11], zu allen Projekten und sämtlichen Vergleichen Venn Diagramme. [1] Dabei ist „pygooglechart“ selbst ein Python-Skript, welches auf die „Google Chart Tools“ zurückgreift.[2] Die „Google Chart Tools“ erlauben es, Diagramme verschiedenster Art mittels Parameter in einer Webseitenanfrage via URL zu generieren. Die korrekte Generierung dieser URL und der Parameter, sowie das Herunterladen der Diagramme aus den vorgegebenen Daten übernimmt „pygooglechart“. Die Venn Diagramme, zu finden auf der CD im

[1] Informationen und eine Definition zu Venn Diagrammen findet man unter: http://www.combinatorics.org/Surveys/ds5/VennWhatEJC.html#whatis

[2] Die Google Chart Tools erreicht man unter: http://code.google.com/intl/de-DE/apis/chart/

Anhang B, spiegeln wieder, inwieweit sich die Ergebnisse der Tools decken.

Die Verzeichnishierarchie nach einem kompletten Durchlauf des FactComparators wird von Tabelle 4.2 beschrieben.

```
corpus-root/
 |- factComparator/
     |- Prolog/
     |- StarterXMLs/
         |- corpus_root.xml
         |- ...
     |- backup/
     |- images/
         |- project1/
         |- ...
     |- logs/
     |- out/
         |- project1/
         |- ...
     |- tmp/
         |- project1/
         |- ...
```

Abbildung 4.2: Komplette Verzeichnisstruktur des FactComparator

4.2.2 Der Prolog-Part

Wie bereits erwähnt, finden sich die Prolog-Pakete im `Prolog`-Verzeichnis des FactComparator. Sobald alle Vorbereitungen, wie das Erstellen der Verzeichnisse und das Sammeln der Daten und Faktenbasen der einzelnen Tools getroffen worden sind, werden diese verschiedenen Prolog-Pakete der Reihe nach ausgeführt.

`CompareCompilationUnits.pro` erlaubt es, die Compilation Units der einzelnen Tools zu vergleichen. Mit `CompareFactbases.pro` werden die Inhalte der einzelnen Compilation Units verglichen. Dabei werden die Signaturen der einzelnen Fakten, wie die von Feldern und Methoden, auf Unterschiede geprüft. `CsvTableCreator.pro` berechnet verschiedene Tabellen aus den Daten die das

`CompareCompilationUnits.pro`-Paket liefert. Wie diese Pakete genau aufgerufen werden und wie sie ineinander greifen, schildert der Abschnitt 4.3.

Sowohl `CompareCompilationUnits.pro` als auch `CompareFactbases.pro` greifen dabei auf das Paket `EnhanceAndLoadFacts.pro` zurück. Mit ihm lassen sich die Faktenbasen der verschiedenen Tools laden. Auch ist es möglich, nur bestimmte Faktentypen zu laden.

4.2.3 Die XML-Konfigurationsdatei

Die XML-Konfigurationsdateien, als Beispiel siehe Abbildung 4.3, liefern dem FactComparator Informationen über den zu untersuchenden Corpus. Wo befindet sich dieser und welche Projekte gilt es zu untersuchen. Das XML-Tag `<corpus>` gibt das Hauptverzeichnis des jeweiligen Corpus an. Da Prolog im Hauptskript über einen Konsolenaufruf gestartet wird, benötigt man das XML-Tag `<swipl>`. Es zielt auf die ausführbare Datei von SWI-Prolog, einer Prolog-Umgebung unter GNU-Lizenz.[3] Die Pfadangabe im `<asmDir>`-Tag verweist auf ein Verzeichnis mit den Faktenbasen des ASM-Based-FactExtractor. Das `<projects>`-Tag beinhaltet die Liste der Projekte in einem Corpus, deren Bezeichner jeweils in ein separates `<project>`-Tag abgelegt sind. Alle soeben beschriebenen Tags befinden sich im XML-Root-Tag `<starter>`.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <starter>
3      <corpus>/java/corpus_demo/</corpus>
4      <swipl>/opt/local/bin/swipl</swipl>
5      <asmDir>
6          /java/corpus_demo/facts_ASMTool/05-05-2011-13-43-12/
7      </asmDir>
8      <projects>
9          <project>proc</project>
10     </projects>
11 </starter>
```

Abbildung 4.3: Beispiel einer XML-Konfigurationsdatei

[3] http://www.swi-prolog.org/

4.2.4 Die Aufrufparameter

Über die Aufrufparameter der Konsole wird dem FactComparator die XML-Konfigurationsdatei mitgeteilt. Außerdem können bestimmte Programmelemente und Programmabschnitte durch diese Parameter ausgelassen werden. Eine Liste der verschiedenen Parameter wird in Abbildung 4.4 gezeigt, ebenso ist dort ein Beispielaufruf unter „Usage:“ zu sehen.

```
Usage: [sudo] Python factComparator.py -f Corpus-XML-File [options]

Compare facts between buildAnalyer, Recoder-based-Factextractor and ASM-based-
Factextractor. Use option-flags to turn of some parts of the factComparator,
e.g. when you do not want to create charts...

Options:
  -h, --help            show this help message and exit
  -f CORPUS_XML, --file=CORPUS_XML
                        xml-file containing corpus-dir, swipl-dir, asmDir and
                        project-list
  -x, --cleanup         flag just to delete all created folders
  -n, --no-copy         flag if you want the old tmp-folder to remain
                        unchanged from previous runs and no copying should be
                        applied
  -c, --no-charts       flag if you don't want to create charts
  -b, --no-factbases    flag to avoid factbase comparation
  -q, --quiet           flag to run swipl in silent-mode
```

Abbildung 4.4: Liste der Aufrufparameter

4.3 Schritte der Informationsgewinnung

Die Informationsgewinnung erfolgt im FactComparator stufenweise. Die Ergebnisse werden zur weiteren Verwendung in Dateien zwischengespeichert. Dabei werden die Zwischenergebnisse früherer Stufen von den darauf folgenden genutzt.

Im ersten Schritt werden Compilation Units verglichen und in verschiedene Kategorien sortiert. Insgesamt liegen drei Tools vor, der BuildAnalyzer, der Recoder-Based-FactExtractor und der ASM-Based-FactExtractor. Zusammen mit der Tatsache, dass der ASM-Based-FactExtractor, sowohl Class- als auch Jar-Dateien untersucht, ergeben sich alles in allem fünf verschiedene Konstellationen, die es zu betrachten gilt:

- BuildAnalyzer <-> Recoder-Based-FactExtractor
- BuildAnalyzer <-> ASM-Based-FactExtractor (jeweils Class-Dateien)
- BuildAnalyzer <-> ASM-Based-FactExtractor (Jar-Dateien)
- Recoder-Based-FactExtractor <-> ASM-Based-FactExtractor (Class-Dateien)
- Recoder-Based-FactExtractor <-> ASM-Based-FactExtractor (Jar-Dateien)

Als nächstes werden anhand der gelieferten Resultate Tabellen berechnet. Sie sollen zeigen, inwiefern sich die Faktenbasen der Tools decken. Hier zeigt sich, ob tatsächlich alle vom BuildAnalyzer vorgegeben Compilation Units von den Faktextraktoren untersucht wurden.

Anschließend werden die Faktenbasen der beiden Faktextraktoren verglichen. Dabei werden die Felder, die Methoden, sowie die Methodenaufrufe der Compilation Units betrachtet, die von beiden Faktextraktoren untersucht wurden. Es werden sowohl die Menge der Faktentypen pro Compilation Unit ausgegeben als auch alle Unterschiede in den Fakten dokumentiert. Da, wie gehabt, zwei Faktextraktoren vorliegen und der ASM-Based-FactExtractor zwei Modi hat, sind hier noch die folgenden Vergleiche notwendig:

- Recoder-Based-FactExtractor <-> ASM-Based-FactExtractor (Class-Dateien)
- Recoder-Based-FactExtractor <-> ASM-Based-FactExtractor (Jar-Dateien)

Als letzter Schritt werden die bereits beschriebenen Venn Diagramme zur Visualisierung der Schnittmengen erstellt.

Die Ergebnisse, sowie die Erläuterungen zu ihnen, finden sich im Kapitel 5 dieser Thesis.

KAPITEL 5

ERGEBNISSE

5.1 Beschreibung der Ergebnisse

Die Arten der Ergebnisse und aus welchen Schritten sie entstanden sind, sind in Abschnitt 4.3 nachzulesen. Hier sollen nun die Ergebnisse am Beispiel des kleinen Test-Programms `proc` vorgestellt werden.[1] Die Interpretation der Ergebnisse erfolgt im nächsten Abschnitt (5.2). Die Konfigurationsdateien der Faktextraktoren zu diesem Programm wurden bereits in Abschnitt 3.1 gezeigt. Anhand dieser Dateien produzieren die Faktextraktoren ihre Faktenbasen.

Der FactComparator vergleicht in seiner ersten Stufe die Compilation Units dieser Faktenbasen mit den vom BuildAnalyzer gelieferten Fakten über die zu analysierenden Compilation Units, siehe etwa Abbildung 3.2. Als Ergebnis werden in dieser Stufe zunächst sogenannte „rough"-Tabellen erstellt. Diese beinhalten ausschließlich Werte über die verschiedenen Kategorien, in welche die verglichenen Compilation Units einsortiert werden.

- Menge der Compilation Units die von Vergleichskandidat 1 respektive 2, erkannt wurden (BA, RFE, ASM)
- Menge über die Zahl eventueller innerer Klassen (BA_inner, RFE_inner, ASM_inner)

[1] Die Quell-Dateien von `proc` befinden sich im Anhang

- Menge aller Compilation Units, die von beiden Vergleichskandidaten gefunden wurden (both)
- Menge an Compilation Units, die lediglich von einem der Vergleichskandidaten untersucht wurde (onlyBA, onlyRFE, onlyASM)

project	BA	RFE	both	onlyBA	onlyRFE	RFE_inner
proc	8	11	8	0	3	3

Tabelle 5.1: Vergleich der Compilation Units zwischen BA und RFE

project	BA	ASM	both	onlyBA	onlyASM	BA_inner	ASM_inner
proc	11	11	11	0	0	3	3

Tabelle 5.2: Vergleich der Compilation Units zwischen BA und ASM

project	RFE	ASM	both	onlyRFE	onlyASM	RFE_inner	ASM_inner	RFE_innerF
proc	11	11	11	0	0	3	3	0

Tabelle 5.3: Vergleich der Compilation Units zwischen RFE und ASM

Aus diesen ersten „rough“-Tabellen berechnen sich anschließend die „clean“-Tabellen. Bei diesen wurden eventuelle Unreinheiten herausgerechnet. In Tabelle 5.3 ist zusätzlich zu den oben beschriebenen Werte noch eine weiterer Wert RFE_innerF (entspricht RFE_innerFailed) zu finden. Wie sich herausgestellt hat, gibt es beim Recoder-Based-FactExtractor einige Probleme bei der Erkennung von inneren Klassen. Diese werden fälschlicherweise als eigenständige Compilation Units erkannt und führen somit zu einer Verfälschung des Ergebnisses. RFE_innerF ergibt sich aus dem Abgleich der Ergebnisse, die ausschließlich in den Faktenbasen von RFE gefunden wurden, mit denen, die ausschließlich als innere Klassen in ASM vorhanden waren.

Als Zusatz zu den bereinigten Werten wird in den „clean“-Tabellen noch die relative Deckungsgleichheit zwischen RFE und ASM (siehe „RFE∩ASM %“) respektive die Vollständigkeit der Faktenbasen zwischen BA und RFE/ASM (siehe etwa „BA∩RFE %“) angegeben.

Zur Visualisierung der Vollständigkeiten wurden Venn Diagramme herangezogen. Basierend auf den Ergebnissen der „clean“-Tabellen ergeben sich für `proc` die Abbildungen 5.1, 5.2 und 5.3

project	BA	RFE	RFE_inner	RFE_innerF	BA∩RFE	BA∩RFE %	onlyBA	onlyRFE
proc	8	11	3	0	8	100	0	0

Tabelle 5.4: „clean“-Tabelle für BA und RFE

project	BA	ASM	BA_inner	ASM_inner	BA∩ASM	BA∩ASM %	onlyBA	onlyASM
proc	11	11	3	3	11	100	0	0

Tabelle 5.5: „clean“-Tabelle für BA und ASM

project	RFE	ASM	RFE_inner	RFE_innerF	ASM_inner	RFE∩ASM	RFE∩ASM %	onlyRFE	onlyASM
proc	11	11	3	0	3	11	100	0	0

Tabelle 5.6: „clean“-Tabelle für RFE und ASM

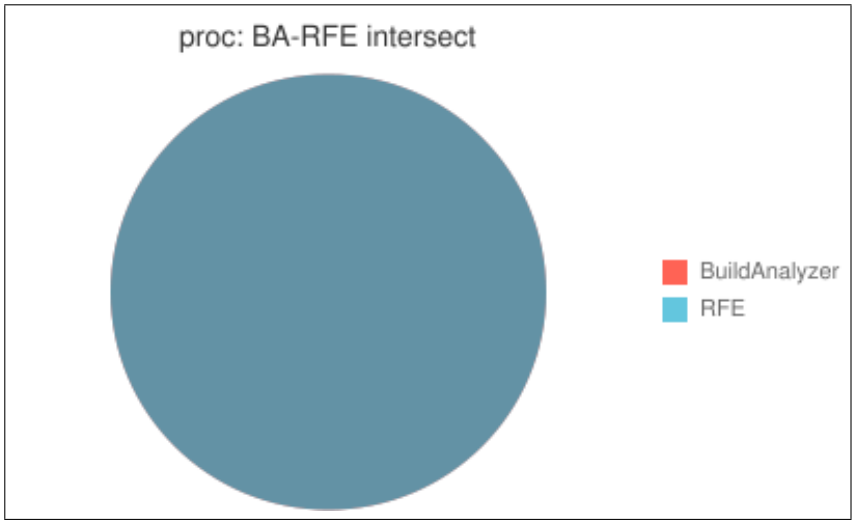

Abbildung 5.1: Venn Diagramm für `proc` zu BA und RFE

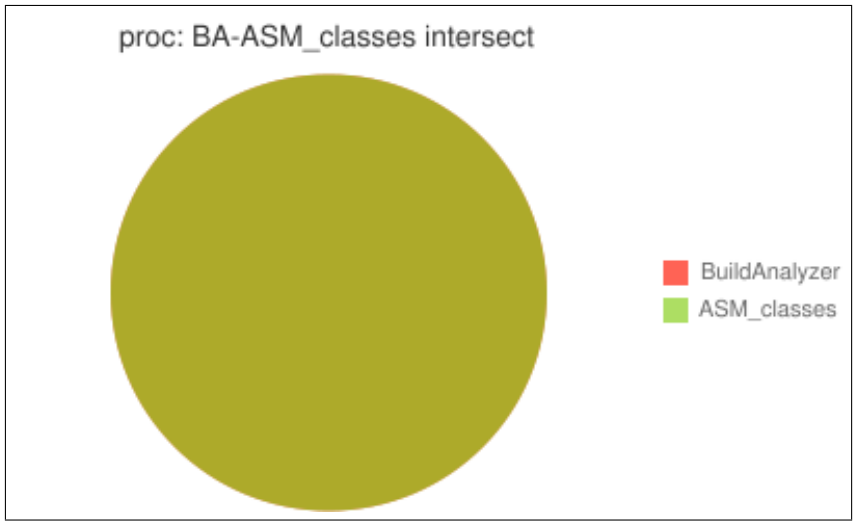

Abbildung 5.2: Venn Diagramm für `proc` zu BA und ASM

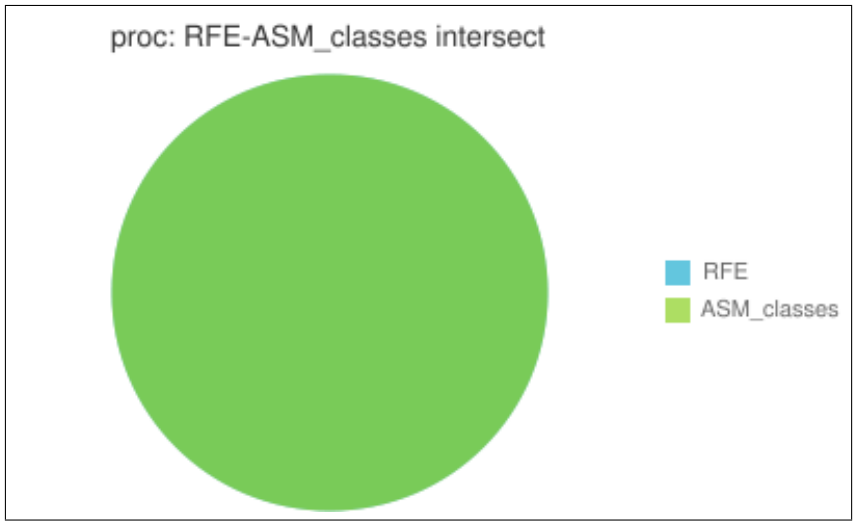

Abbildung 5.3: Venn Diagramm für `proc` zu RFE und ASM

Da für `proc` die Java-Dateien ausschließlich in Class-Dateien kompiliert werden, sind die Tabellen und Diagramme, welche auf den Jar-Teil des ASM-Based-FactExtractor eingehen, hier nicht aufgeführt. Für den Corpus *JavaShape* finden sich diese jedoch neben den hier gezeigten Ergebnissen im Anhang an den entsprechenden Stellen.

In einem weiteren Schritt berechnet der FactComparator, wieviele Felder, Methoden und Methodenaufrufe sich in einer Compilation Unit vom Recoder-Based-FactExtractor und ASM-Based-FactExtractor befinden und stellt diese Werte in einer CSV-Tabelle einander gegenüber. Pro analysiertem Softwareprojekt wird eine solche „compare"-Datei angelegt. Für das `proc`-Beispiel findet sich diese Datei in Tabelle 5.7.

Eventuelle Unterschiede zwischen den Faktenbasen der beiden Faktextraktoren werden im letzten Schritt des FactComparators dokumentiert. Sie dienen hauptsächlich dem Aufdecken von Fehlern oder Unstimmigkeiten in der Ausgabeform der Fakten. Ein Ausschnitt einer solchen „diff"-Datei zeigt Tabelle 5.8.[2]

[2]diff: von difference (engl. für Unterschied)

project	RFE∩ASM CUs	# RFE Felder	# ASM Felder	# RFE Methoden	# ASM Methoden	# RFE Methoden-aufrufe	# ASM Methoden-aufrufe
proc	[org,gui,'Overview']	5	4	3	3	20	20
proc	[org,gui,'Overview$1']	1	3	1	2	3	4
proc	[org,proc,'Book']	0	0	2	2	2	6
proc	[org,proc,'Consumer']	2	2	5	5	5	11
proc	[org,proc,'Producer']	1	1	2	2	5	6
proc	[org,proc,'Producer$1']	0	1	1	2	1	2
proc	[org,proc,'Product']	1	1	3	4	1	3
proc	[org,proc,'Product$PID']	2	3	4	5	8	14
proc	[org,proc,'School']	0	0	1	1	1	2
proc	[org,'Program']	0	0	1	2	6	7
proc	[org,proc,'Buyable']	0	0	1	1	0	0

Tabelle 5.7: Faktenbasis-Vergleichstabelle zwischen RFE und ASM

project	RFE∩ASM CUs	Fundort: RFE oder ASM ?	Fakten-typ	Fakten-signatur
proc	'Overview'	rfe	field	rfe(field(proc,[org,gui,'Overview'],['long[]'],test,[public]))
proc	'Overview'	asm	methodCall	methodCall('Overview.class',[org,gui,'Overview','<init>'],[org,gui,'Overview',pack],[])
proc	'Overview'	asm	methodCall	methodCall('Overview.class',[org,gui,'Overview','<init>'],[org,gui,'Overview',setVisible],[[boolean]])
proc	'Overview'	rfe	methodCall	methodCall(proc,[org,gui,'Overview','<init>'],[java,awt,'Window',pack],[])
proc	'Overview'	rfe	methodCall	methodCall(proc,[org,gui,'Overview','<init>'],[java,awt,'Window',setVisible],[[boolean]])
proc	'Overview$1'	asm	field	asm(field('Overview$1.class',[org,gui,'Overview$1'],[org,proc,'Product'],'val$p',[default,final]))
proc	'Overview$1'	asm	field	asm(field('Overview$1.class',[org,gui,'Overview$1'],[org,gui,'Overview'],'this$0',[default,final]))
proc	'Overview$1'	asm	methodCall	methodCall('Overview$1.class',[org,gui,'Overview$1','<init>'],[java,lang,'Object','<init>'],[])
proc	'Overview$1'	asm	methodCall	methodCall('Overview$1.class',[org,gui,'Overview$1',actionPerformed],[org,gui,'Overview',repaint],[])
proc	'Overview$1'	rfe	methodCall	methodCall(proc,[org,gui,'Overview$1',actionPerformed],[java,awt,'Component',repaint],[])
proc	'Book'	asm	methodCall	methodCall('Book.class',[org,proc,'Book',buy],[java,lang,'StringBuilder','<init>'],[])
proc	'Book'	asm	methodCall	methodCall('Book.class',[org,proc,'Book',buy],[java,lang,'StringBuilder',append],[[java,lang,'Object']])
proc	'Book'	asm	methodCall	methodCall('Book.class',[org,proc,'Book',buy],[java,lang,'StringBuilder',append],[[java,lang,'String']])
proc	'Book'	asm	methodCall	methodCall('Book.class',[org,proc,'Book',buy],[java,lang,'StringBuilder',toString],[])
proc	'Producer$1'	asm	field	asm(field('Producer$1.class',[org,proc,'Producer$1'],[org,proc,'Producer'],'this$0',[default,final]))
proc	'Producer$1'	asm	method	method('Producer$1.class',[org,proc,'Producer$1'],'<init>',[default],[void],[[org,proc,'Producer'],[int],[int]])
proc	'Producer$1'	asm	methodCall	methodCall('Producer$1.class',[org,proc,'Producer$1','<init>'],[org,proc,'Product','<init>'],[[int],[int]])
proc	'Product'	asm	methodCall	methodCall('Product.class',[org,proc,'Product','<init>'],[java,lang,'Object','<init>'],[])
proc	'Product'	asm	methodCall	methodCall('Product.class',[org,proc,'Product','<init>'],[java,lang,'Object','<init>'],[])
proc	'Product'	rfe	methodCall	methodCall(proc,[org,proc,'Product','<init>'],[org,proc,'Product','PID','<init>'],[[int],[int]])
proc	'Product$PID'	asm	field	asm(field('Product$PID.class',[org,proc,'Product$PID'],[org,proc,'Product'],'this$0',[default,final]))
proc	'Product$PID'	asm	methodCall	methodCall('Product$PID.class',[org,proc,'Product$PID',compareTo],[org,proc,'Product$PID',getGid],[])
proc	'Product$PID'	rfe	methodCall	methodCall(proc,[org,proc,'Product$PID',compareTo],[org,proc,'Product','PID',getGid],[])
proc	'School'	asm	methodCall	methodCall('School.class',[org,proc,'School','<init>'],[org,proc,'Consumer','<init>'],[])
proc	'Program'	asm	method	method('Program.class',[org,'Program'],'<init>',[public],[void],[])
proc	'Program'	asm	methodCall	methodCall('Program.class',[org,'Program','<init>'],[java,lang,'Object','<init>'],[])

Tabelle 5.8: Ein Ausschnitt aus der Faktenbasis-Vergleichstabelle zwischen RFE und ASM

5.2 Interpretation der Ergebnisse

Die Überschneidungen der einzelnen Resultate der Tools lassen sich am besten durch die Venn Diagramme visualisieren. Dabei sind verschiedene Kategorien zu erkennen. Einige Projekte können beispielsweise nicht kompiliert werden, sie liefern somit keine Ergebnisse, weder vom BuildAnalyzer, noch von einem der Faktextraktoren. Was sich in den Tabellen durch eine Zeile mit ausschließlich Nullen präsentiert, spiegelt sich in den Venn Diagrammen als weiße Fläche wieder, siehe hierzu etwa ein Venn Diagramm des Programms `drjava` Abbildung 5.4. Weitere Vertreter dieser Art sind zum Beispiel `glassfish`, `hibernate` und `jag`.

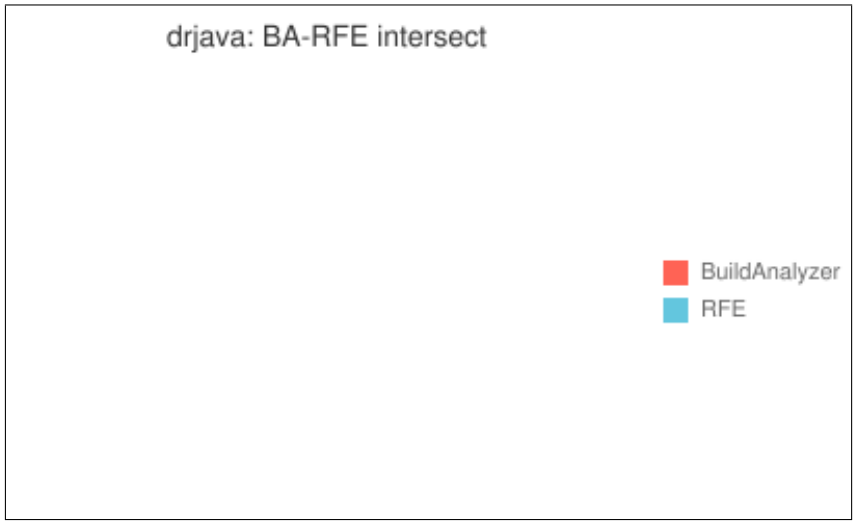

Abbildung 5.4: `drjava`: Beispiel für ein nicht kompilierbares Projekt

Dem gegenüber stehen die meisten Projekte, die nahezu komplette Deckungsgleichheit erreichen. Zu unterscheiden sind hier grundsätzlich zwei Typen. Solche Projekte wie `aglets`, `antlr`, `derby` oder `sequoia`, die sich in Class-Dateien kompilieren, stehen etwa `fitlibrary`, `ireport` oder `jung` gegenüber, die jeweils in Jar-Dateien kompilieren. Als Beispiel für erstere sei hier `sequoia` aufgeführt, siehe Abbildung 5.5. Abbildung 5.6 zeigt mit `ireport` einen Vertreter letzterer.

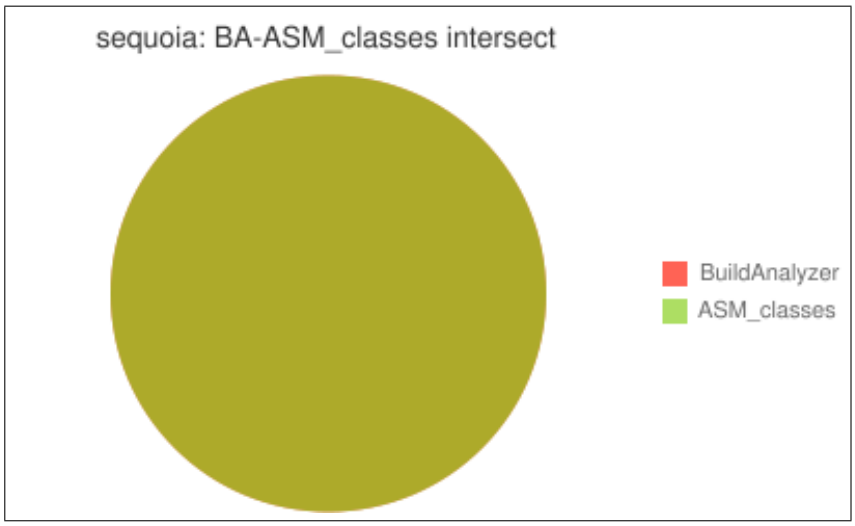

Abbildung 5.5: `sequoia`: Beispiel für Projekt, welches sich in Class-Dateien kompiliert

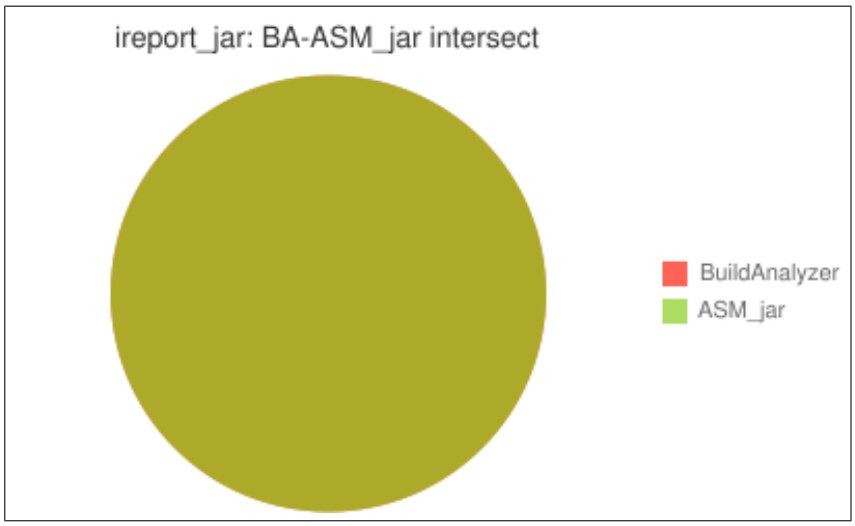

Abbildung 5.6: `ireport`: Beispiel für Projekt, welches sich in Jar-Dateien kompiliert

Projekte wie `hsqldb`, `jchempaint` oder `jetty` kompilieren zwar sowohl in Class- als auch Jar-Dateien, jedoch ist eine deutliche Tendenz in eine Richtung zu erkennen. Am Beispiel von `hsqldb` geht diese, wie aus Abbildung 5.7 ersichtlich, weg von Class-Dateien. `jchempaint` in derselben Abbildung hingegen kompiliert nur zu kleinen Teilen in Jar-Dateien. Die Diagramme der jeweils anderen Kompilierungsform sind nahezu Deckungsgleich.

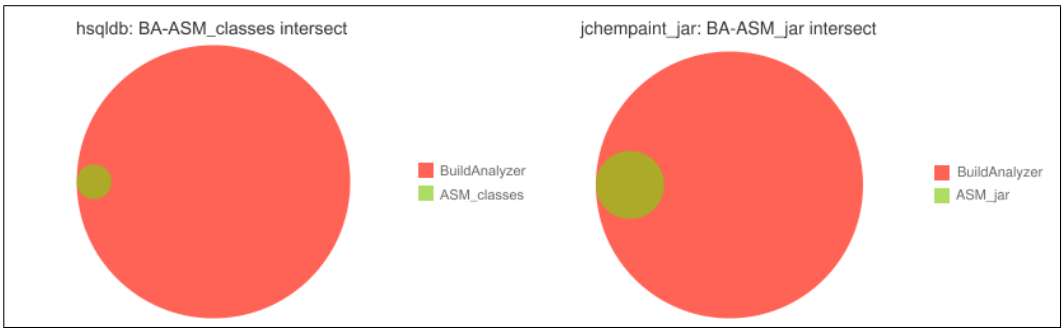

Abbildung 5.7: `hsqldb` und `jchempaint`: Beispiele für die teilweise Kompilierung in eine Richtung

Im Gegensatz zum ASM-Based-FactExtractor, welcher den Bytecode analysiert, wird der Recoder-Based-FactExtractor ausschließlich auf die Source-Dateien angewendet. Lässt sich ein Projekt nicht kompilieren, so sind die Venn Diagramme des Recoder-Based-FactExtractors ebenfalls leer. Bei allen anderen Projekten zeichnet sich eine sehr hohe Deckungsgleichheit mit den Ergebnissen des Build-Analyzer ab, wie Abbildung 5.8 aber auch die Tabellen im Anhang A belegen.

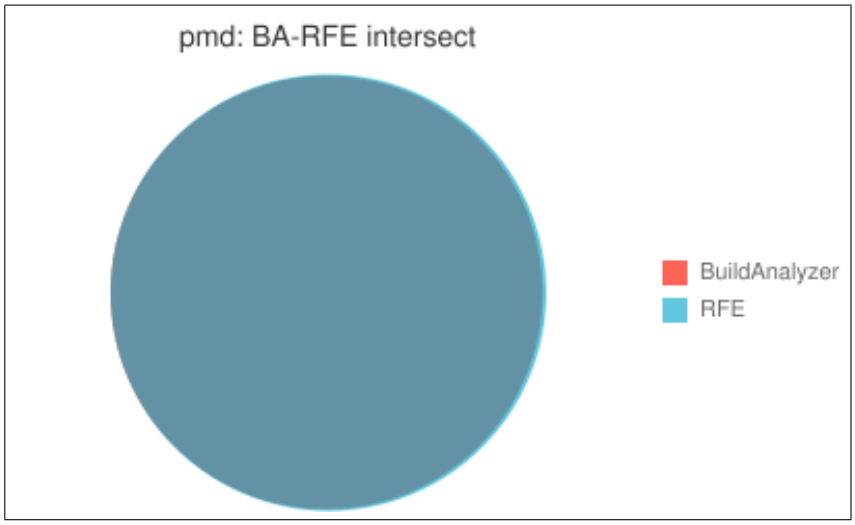

Abbildung 5.8: `pmd`: Beispiel für eine hohe Deckungsgleichheit zwischen BA und RFE

Die Unterschiede in den „diff“- und „compare“-Dateien, welche für den Corpus *JavaShape* ebenfalls auf der CD im Anhang B zu finden sind, lassen sich zwei verschiedenen Ursachen zuschreiben.

Hauptursache für die Unterschiede ist der Java-Compiler. Dieser ist bemüht beim Kompilieren den Quellcode in einen so effektiv wie möglichen Bytecode zu transferieren. Dabei kann es vorkommen, dass neue lokale Variablen auftauchen oder Paketabhängigkeiten bereits aufgelöst werden. Da der Recoder-Based-FactExtractor auf die Analyse der Quell-Dateien abzielt, bekommt dieser von den Optimierungen des Java-Compilers nichts mit. Der ASM-Based-FactExtractor, der nach dem Kompilieren entweder auf die Class- oder die Jar-Dateien angewendet wird, analysiert folglich den optimierteren Bytecode.

Weitere Unterschiede kommen daher, dass der ASM-Based-FactExtractor Methodenaufrufe über weitere Schritte auflöst, diese Möglichkeit hat der Recoder-Based-FactExtractor nicht.

Findet der RFE beispielsweise einen Aufruf der Form `PrintStream.println()`, so wird dieser als Prologfakt wie folgt ausgedrückt:

- `['java','io','PrintStream','println'],[['java','lang','String']]`

Beim ASM wird dieser Aufruf noch weiter aufgelöst und es lassen sich zusätzlich

die folgenden Methodenaufrufe finden:[3]

- `['java','lang','StringBuilder','<init>'],[]`
- `['java','lang','StringBuilder','append'],[['java','lang','Object']]`
- `['java','lang','StringBuilder','append'],[['java','lang','String']]`
- `['java','lang','StringBuilder','toString'],[]`

[3]Alle aufgeführten Codefragmente sind jeweils die `CalledMethodName` und `ArgumentsList` eines „methodCall“ Fakts

KAPITEL 6

FAZIT

6.1 Zusammenfassung

Das Ziel dieser Thesis war es, einen funktionierenden Corpus mitsamt automatisierter Anwendung der verschiedenen Tools sicher am Laufen zu haben. Trotz einiger Schwierigkeiten bei den einzelnen Tools funktioniert mittlerweile der eingerichtete Nightly-Build wieder. Hier werden die vorhandenen Projekte der einzelnen Corpora der Reihe nach kompiliert, um auf diese danach die Tools anzuwenden. Die Ergebnisse der Nightly-Builds können unter http://empirical.uni-koblenz.de/~nightlyrunner/nightlies/ abgerufen werden.

Der BuildAnalyzer liefert die Daten für die Faktextraktoren. Diese extrahieren mittlerweile sehr zuverlässig und mit einer hohen Übereinstimmung der Compilation Units die Projekte. Auch stimmen die extrahierten Fakten der verschiedenen Faktextraktoren in weiten Teilen überein, so dass beide verlässlich zu weiteren Analysen genutzt werden können.

Viele der Ergebnisse der Untersuchungen im Rahmen dieser Thesis sind direkt in die Faktextraktoren eingeflossen. So wurden etwa die Signaturen der Faktenbasen teilweise überarbeitet, um genauere Informationen zu liefern. Auch wurde der Recoder-Based-FactExtractor an einigen Stellen angepasst, da dieser innere Klassen nicht in einer $-Notation dargestellt hat.[1]

[1] $-Notation: ÄußereKlasse$InnereKlasse: für benannte innere Klassen.
ÄußereKlasse${nummeriert}: für anonyme innere Klassen

6.2 Ausblick

Die wichtigsten Grundlagen für weitergehende Analysen wurden vor dieser Thesis geschaffen und deren Ergebnisse durch den FactComparator bekräftigt. In weiteren Forschungsarbeiten könnten daher weitere Corpora angelegt werden, um die Analysen auf ein breiteres Feld auszuweiten.

Die Entwicklungen an den Tools, auf denen die Faktextraktoren aufbauen, sind noch nicht vollständig abgeschlossen und so sollte die Qualität der Faktextraktoren in Zukunft weiter steigen. Sollten neue Funktionen in den genutzten Frameworks implementiert werden, können diese in Faktextraktoren übernommen werden, oder aber, wenn einige Fehler in den Frameworks behoben werden, die bisher zu Abbrüchen in den Analysevorgängen führen. Somit würde die Zahl der analysierten Projekte weiter steigen.

KAPITEL 7

DANKSAGUNGEN

Zum Abschluss möchte ich allen Beteiligten des *API–Analysis 2.0* Teams für eine gute Zusammenarbeit danken. Insbesondere danke ich meinen Betreuern Prof. Dr. Ralf Lämmel und Ekatarina Pek für die vielen Ideen und Ratschläge. Weiterer Dank geht an Sebastian Jackel, Jan Baltzer, Joachim Pehl und Malte Knauf, für ihre Tools und die Auskünfte über diese.

Ebenfalls möchte ich mich bei meiner Familie und bei meinen Freunden bedanken, die mich bei der Erstellung dieser Arbeit so gut unterstützt haben.

LITERATURVERZEICHNIS

[BFN+06] BAXTER, Gareth ; FREAN, Marcus ; NOBLE, James ; RICKERBY, Mark ; SMITH, Hayden ; VISSER, Matt ; MELTON, Hayden ; TEMPERO, Ewan: Understanding the shape of Java software. In: *Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*. New York, NY, USA : ACM, 2006 (OOPSLA '06). – ISBN 1–59593–348–4, 397–412

[Kas11] KASZUBA, Gerald: *pygooglechart - A complete Python wrapper for the Google Chart API*. http://pygooglechart.slowchop.com, as on 28.04.2011

[LPS11] LÄMMEL, Ralf ; PEK, Ekaterina ; STAREK, Jürgen: Large-scale, AST-based API-usage analysis of open-source Java projects. In: *SAC'11 - ACM 2011 SYMPOSIUM ON APPLIED COMPUTING, Technical Track on "Programming Languages"*, 2011

[Sta10] STAREK, Jürgen: *A large-scale analysis of Java API usage*, Universität Koblenz-Landau, Campus Koblenz, Diplomarbeit, 2010

ANHANG A

CSV-TABELLEN

In diesem Abschnitt des Anhang sind alle vom FactComparator erstellten Tabellen zu finden. Unterteilt in die Unterabschnitte A.1, in dem die „rough“-Tabellen zu finden sind, und A.2, in dem alle „clean“-Tabellen zusammengetragen sind.

A.1 „rough“-Tabellen

project	BA CUs	RFE CUs	Both	OnlyBA	OnlyRFE	RFE_Inner
aglets	276	409	276	0	133	115
ant	748	1092	748	0	344	131
antlr	105	212	105	0	107	51
aoi	367	663	367	0	296	246
argouml	2308	2906	2307	1	599	417
axion	381	437	381	0	56	39
azureus	3262	7588	3262	0	4326	3985
colt	297	592	297	0	295	294
columba	1366	1681	1365	1	316	297
compiere	2500	2519	2310	190	209	145
derby	280	310	278	2	32	21
drjava	0	0	0	0	0	0
fit	59	97	59	0	38	11
fitlibrary	875	1478	875	0	603	361
galleon	282	804	282	0	522	406
ganttproject	519	1077	519	0	558	451
geronimo	4	4	4	0	0	0
glassfish	0	0	0	0	0	0
hibernate	0	0	0	0	0	0
hsqldb	248	334	248	0	86	60
ireport	655	2534	655	0	1879	1835
jag	0	0	0	0	0	0
jaga	0	0	0	0	0	0
james	275	330	275	0	55	41
jasperreports	1389	1711	1348	41	363	159
javacc	0	0	0	0	0	0
jboss	231	306	231	0	75	41
jchempaint	824	882	824	0	58	52
jdk	0	0	0	0	0	0
jdom	125	149	125	0	24	12
jedit	487	1095	487	0	608	389
jetty	322	409	322	0	87	47
jext	0	0	0	0	0	0
jfreechart	492	511	492	0	19	7
jgraph	218	370	218	0	152	116
jhotdraw	0	0	0	0	0	0
jmeter	736	863	736	0	127	81
joggplayer	0	0	0	0	0	0
jrefactory	0	0	0	0	0	0
jtopen	0	0	0	0	0	0
jung	20	34	19	1	15	1
junit	277	579	251	26	328	104
lucene	415	725	415	0	310	142
megamek	1334	1781	1334	0	447	386
netbeans	734	1738	712	22	1026	496
openoffice	0	0	0	0	0	0
pmd	751	914	751	0	163	94
poi	2006	2426	1977	29	449	185
rssowl	0	0	0	0	0	0
sablecc	198	267	198	0	69	55
sandmark	0	0	0	0	0	0
scala	0	0	0	0	0	0
sequoia	173	207	173	0	34	15
tomcat	0	0	0	0	0	0

Tabelle A.1: „rough“-Tabelle zwischen BuildAnalyzer und RFE

project	BA CUs	ASM CUs	Both	OnlyBA	OnlyASM	BA_Inner	ASM_Inner
aglets	409	409	409	0	0	139	139
ant	1096	1045	1045	51	0	366	342
antlr	212	212	212	0	0	107	107
aoi	0	0	0	0	0	0	0
argouml	2777	951	951	1826	0	523	176
axion	0	0	0	0	0	0	0
azureus	0	0	0	0	0	0	0
colt	593	593	593	0	0	299	299
columba	1686	1	1	1685	0	337	0
compiere	7457	694	694	6763	0	511	39
derby	309	309	309	0	0	29	29
drjava	0	0	0	0	0	0	0
fit	96	96	96	0	0	38	38
fitlibrary	0	0	0	0	0	0	0
galleon	0	0	0	0	0	0	0
ganttproject	1088	77	77	1011	0	639	46
geronimo	6	6	6	0	0	0	0
glassfish	0	0	0	0	0	0	0
hibernate	0	0	0	0	0	0	0
hsqldb	338	7	7	331	0	92	0
ireport	0	0	0	0	0	0	0
jag	0	0	0	0	0	0	0
jaga	0	0	0	0	0	0	0
james	342	342	342	0	0	69	69
jasperreports	0	0	0	0	0	0	0
javacc	152	152	152	0	0	17	17
jboss	0	0	0	0	0	0	0
jchempaint	886	559	559	327	0	67	19
jdk	0	0	0	0	0	0	0
jdom	0	0	0	0	0	0	0
jedit	0	0	0	0	0	0	0
jetty	415	333	333	82	0	99	99
jext	0	0	0	0	0	0	0
jfreechart	0	0	0	0	0	0	0
jgraph	186	186	186	0	0	83	83
jhotdraw	0	0	0	0	0	0	0
jmeter	873	399	399	474	0	141	49
joggplayer	0	0	0	0	0	0	0
jrefactory	0	0	0	0	0	0	0
jtopen	10	10	10	0	0	1	1
jung	0	0	0	0	0	0	0
junit	601	404	404	197	0	357	298
lucene	0	0	0	0	0	0	0
megamek	0	0	0	0	0	0	0
netbeans	1791	1791	1791	0	0	1233	1233
openoffice	0	0	0	0	0	0	0
pmd	925	17	17	908	0	177	4
poi	2472	2472	2472	0	0	473	473
rssowl	0	0	0	0	0	0	0
sablecc	0	0	0	0	0	0	0
sandmark	0	0	0	0	0	0	0
scala	0	0	0	0	0	0	0
sequoia	206	206	206	0	0	34	34
tomcat	0	0	0	0	0	0	0

Tabelle A.2: „rough“-Tabelle zwischen BuildAnalyzer und ASM

project	BA CUs	ASM CUs	Both	OnlyBA	OnlyASM	BA_Inner	ASM_Inner
aglets_jar	0	0	0	0	0	0	0
ant_jar	1098	1	1	1097	0	366	0
antlr_jar	0	0	0	0	0	0	0
aoi_jar	662	1051	662	0	389	323	453
argouml_jar	4786	1914	1837	2949	77	1046	349
axion_jar	445	448	445	0	3	64	64
azureus_jar	7942	7599	7594	348	5	4974	4966
colt_jar	0	0	0	0	0	0	0
columba_jar	1689	1701	1689	0	12	337	337
compiere_jar	12550	1359	1357	11193	2	906	50
derby_jar	0	0	0	0	0	0	0
drjava_jar	0	0	0	0	0	0	0
fit_jar	0	0	0	0	0	0	0
fitlibrary_jar	1492	1495	1492	0	3	619	619
galleon_jar	1227	6	6	1221	0	1165	4
ganttproject_jar	1088	1016	1011	77	5	639	595
geronimo_jar	0	0	0	0	0	0	0
glassfish_jar	0	0	0	0	0	0	0
hibernate_jar	0	0	0	0	0	0	0
hsqldb_jar	313	313	313	0	0	91	91
ireport_jar	2547	2540	2537	10	3	1887	1887
jag_jar	0	0	0	0	0	0	0
jaga_jar	0	0	0	0	0	0	0
james_jar	0	0	0	0	0	0	0
jasperreports_jar	3832	1819	1766	2066	53	812	379
javacc_jar	0	0	0	0	0	0	0
jboss_jar	322	105	2	320	103	107	6
jchempaint_jar	887	51	51	836	0	67	13
jdk_jar	0	0	0	0	0	0	0
jdom_jar	76	76	76	0	0	13	13
jedit_jar	1119	1132	1119	0	13	692	700
jetty_jar	416	84	82	334	2	99	0
jext_jar	0	0	0	0	0	0	0
jfreechart_jar	509	4	4	505	0	17	0
jgraph_jar	0	0	0	0	0	0	0
jhotdraw_jar	0	0	0	0	0	0	0
jmeter_jar	816	4	4	812	0	132	1
joggplayer_jar	0	0	0	0	0	0	0
jrefactory_jar	0	0	0	0	0	0	0
jtopen_jar	0	0	0	0	0	0	0
jung_jar	38	38	38	0	0	18	18
junit_jar	397	209	209	188	0	118	59
lucene_jar	736	743	736	0	7	345	345
megamek_jar	1778	1800	1778	0	22	453	453
netbeans_jar	2775	166	5	2770	161	2015	8
openoffice_jar	0	0	0	0	0	0	0
pmd_jar	925	727	727	198	0	177	155
poi_jar	0	0	0	0	0	0	0
rssowl_jar	0	0	0	0	0	0	0
sablecc_jar	285	286	285	0	1	89	89
sandmark_jar	0	0	0	0	0	0	0
scala_jar	0	0	0	0	0	0	0
sequoia_jar	0	0	0	0	0	0	0
tomcat_jar	0	0	0	0	0	0	0

Tabelle A.3: „rough“-Tabelle zwischen BuildAnalyzer und ASM_J

project	RFE CUs	ASM CUs	Both	OnlyRFE	OnlyASM	RFE_inner	ASM_inner	RFE_innerF
aglets	409	409	379	30	30	115	139	18
ant	1092	1045	771	321	274	131	342	204
antlr	212	212	138	74	74	51	107	56
aoi	0	0	0	0	0	0	0	0
argouml	2906	951	901	2005	50	417	176	21
axion	0	0	0	0	0	0	0	0
azureus	0	0	0	0	0	0	0	0
colt	592	593	588	4	5	294	299	1
columba	1681	1	1	1680	0	297	0	0
compiere	2519	694	407	2112	287	145	39	8
derby	310	309	297	13	12	21	29	7
drjava	0	0	0	0	0	0	0	0
fit	97	96	70	27	26	11	38	26
fitlibrary	0	0	0	0	0	0	0	0
galleon	0	0	0	0	0	0	0	0
ganttproject	1077	77	59	1018	18	451	46	15
geronimo	4	6	4	0	2	0	0	0
glassfish	0	0	0	0	0	0	0	0
hibernate	0	0	0	0	0	0	0	0
hsqldb	334	7	7	327	0	60	0	0
ireport	0	0	0	0	0	0	0	0
jag	0	0	0	0	0	0	0	0
jaga	0	0	0	0	0	0	0	0
james	330	342	305	25	37	41	69	13
jasperreports	0	0	0	0	0	0	0	0
javacc	0	0	0	0	0	0	0	0
jboss	0	0	0	0	0	0	0	0
jchempaint	882	559	557	325	2	52	19	1
jdk	0	0	0	0	0	0	0	0
jdom	0	0	0	0	0	0	0	0
jedit	0	0	0	0	0	0	0	0
jetty	409	333	273	136	60	47	99	39
jext	0	0	0	0	0	0	0	0
jfreechart	0	0	0	0	0	0	0	0
jgraph	370	186	160	210	26	116	83	56
jhotdraw	0	0	0	0	0	0	0	0
jmeter	863	399	379	484	20	81	49	17
joggplayer	0	0	0	0	0	0	0	0
jrefactory	0	0	0	0	0	0	0	0
jtopen	0	0	0	0	0	0	0	0
jung	0	0	0	0	0	0	0	0
junit	579	404	132	447	272	104	298	223
lucene	0	0	0	0	0	0	0	0
megamek	0	0	0	0	0	0	0	0
netbeans	1738	1791	971	767	820	496	1233	601
openoffice	0	0	0	0	0	0	0	0
pmd	914	17	15	899	2	94	4	1
poi	2426	2472	2097	329	375	185	473	263
rssowl	0	0	0	0	0	0	0	0
sablecc	0	0	0	0	0	0	0	0
sandmark	0	0	0	0	0	0	0	0
scala	0	0	0	0	0	0	0	0
sequoia	207	206	187	20	19	15	34	18
tomcat	0	0	0	0	0	0	0	0

Tabelle A.4: „rough“-Tabelle zwischen RFE und ASM

project	RFE	ASM_J	Both	OnlyRFE	OnlyASM_J	RFE_inner	ASM_J_inner	RFE_innerF
aglets_jar	0	0	0	0	0	0	0	0
ant_jar	1092	1	1	1091	0	131	0	0
antlr_jar	0	0	0	0	0	0	0	0
aoi_jar	663	1024	576	87	448	246	446	42
argouml_jar	2906	1899	1817	1089	82	417	335	64
axion_jar	437	445	415	22	30	39	61	14
azureus_jar	7588	7489	6509	1079	980	3985	4855	251
colt_jar	0	0	0	0	0	0	0	0
columba_jar	1681	1701	1648	33	53	297	337	12
compiere_jar	2519	1358	1154	1365	204	145	50	13
derby_jar	0	0	0	0	0	0	0	0
drjava_jar	0	0	0	0	0	0	0	0
fit_jar	0	0	0	0	0	0	0	0
fitlibrary_jar	1478	1475	1227	251	248	361	599	226
galleon_jar	804	6	4	800	2	406	4	1
ganttproject_jar	1077	1000	777	300	223	451	579	88
geronimo_jar	0	0	0	0	0	0	0	0
glassfish_jar	0	0	0	0	0	0	0	0
hibernate_jar	0	0	0	0	0	0	0	0
hsqldb_jar	334	312	273	61	39	60	90	25
ireport_jar	2534	2535	2479	55	56	1835	1882	36
jag_jar	0	0	0	0	0	0	0	0
jaga_jar	0	0	0	0	0	0	0	0
james_jar	0	0	0	0	0	0	0	0
jasperreports_jar	1711	1808	1529	182	279	159	368	144
javacc_jar	0	0	0	0	0	0	0	0
jboss_jar	306	67	2	304	65	41	6	0
jchempaint_jar	882	51	51	831	0	52	13	0
jdk_jar	0	0	0	0	0	0	0	0
jdom_jar	149	76	67	82	9	12	13	14
jedit_jar	1095	1121	743	352	378	389	689	231
jetty_jar	409	84	84	325	0	47	0	0
jext_jar	0	0	0	0	0	0	0	0
jfreechart_jar	511	4	4	507	0	7	0	0
jgraph_jar	0	0	0	0	0	0	0	0
jhotdraw_jar	0	0	0	0	0	0	0	0
jmeter_jar	863	4	4	859	0	81	1	0
joggplayer_jar	0	0	0	0	0	0	0	0
jrefactory_jar	0	0	0	0	0	0	0	0
jtopen_jar	0	0	0	0	0	0	0	0
jung_jar	34	38	20	14	18	1	18	14
junit_jar	579	182	168	411	14	104	57	5
lucene_jar	725	734	526	199	208	142	336	155
megamek_jar	1781	1797	1732	49	65	386	450	38
netbeans_jar	1738	70	3	1735	67	496	3	0
openoffice_jar	0	0	0	0	0	0	0	0
pmd_jar	914	722	623	291	99	94	150	56
poi_jar	0	0	0	0	0	0	0	0
rssowl_jar	0	0	0	0	0	0	0	0
sablecc_jar	267	286	251	16	35	55	89	14
sandmark_jar	0	0	0	0	0	0	0	0
scala_jar	0	0	0	0	0	0	0	0
sequoia_jar	0	0	0	0	0	0	0	0
tomcat_jar	0	0	0	0	0	0	0	0

Tabelle A.5: „rough“-Tabelle zwischen RFE und ASM_J

A.2 „clean“-Tabellen

project	BA	RFE	RFE_inner	RFE_innerF	BA∩RFE	BA∩RFE %	onlyBA	onlyRFE
aglets	276	409	115	18	276	100	0	0
ant	748	1092	131	204	748	100	0	9
antlr	105	212	51	56	105	100	0	0
aoi	367	663	246	42	367	100	0	8
argouml	2308	2906	417	64	2307	99.9567	1	118
axion	381	437	39	14	381	100	0	3
azureus	3262	7588	3985	251	3262	100	0	90
colt	297	592	294	1	297	100	0	0
columba	1366	1681	297	12	1365	99.9268	1	7
compiere	2500	2519	145	13	2310	92.4	190	51
derby	280	310	21	7	278	99.2857	2	4
drjava	0	0	0	0	0	0	0	0
fit	59	97	11	26	59	100	0	1
fitlibrary	875	1478	361	226	875	100	0	16
galleon	282	804	406	1	282	100	0	115
ganttproject	519	1077	451	88	519	100	0	19
geronimo	4	4	0	0	4	100	0	0
glassfish	0	0	0	0	0	0	0	0
hibernate	0	0	0	0	0	0	0	0
hsqldb	248	334	60	25	248	100	0	1
ireport	655	2534	1835	36	655	100	0	8
jag	0	0	0	0	0	0	0	0
jaga	0	0	0	0	0	0	0	0
james	275	330	41	13	275	100	0	1
jasperreports	1389	1711	159	144	1348	97.0482	41	60
javacc	0	0	0	0	0	0	0	0
jboss	231	306	41	0	231	100	0	34
jchempaint	824	882	52	1	824	100	0	5
jdk	0	0	0	0	0	0	0	0
jdom	125	149	12	14	125	100	0	-2
jedit	487	1095	389	231	487	100	0	-12
jetty	322	409	47	39	322	100	0	1
jext	0	0	0	0	0	0	0	0
jfreechart	492	511	7	0	492	100	0	12
jgraph	218	370	116	56	218	100	0	-20
jhotdraw	0	0	0	0	0	0	0	0
jmeter	736	863	81	17	736	100	0	29
joggplayer	0	0	0	0	0	0	0	0
jrefactory	0	0	0	0	0	0	0	0
jtopen	0	0	0	0	0	0	0	0
jung	20	34	1	14	19	95.0	1	0
junit	277	579	104	223	251	90.6137	26	1
lucene	415	725	142	155	415	100	0	13
megamek	1334	1781	386	38	1334	100	0	23
netbeans	734	1738	496	601	712	97.0027	22	-71
openoffice	0	0	0	0	0	0	0	0
pmd	751	914	94	56	751	100	0	13
poi	2006	2426	185	263	1977	98.5543	29	1
rssowl	0	0	0	0	0	0	0	0
sablecc	198	267	55	14	198	100	0	0
sandmark	0	0	0	0	0	0	0	0
scala	0	0	0	0	0	0	0	0
sequoia	173	207	15	18	173	100	0	1
tomcat	0	0	0	0	0	0	0	0

Tabelle A.6: „clean“-Tabelle zwischen BuildAnalyzer und RFE

project	BA	ASM	BA_inner	ASM_inner	BA∩ASM	BA∩ASM %	onlyBA	onlyASM
aglets	409	409	139	139	409	100	0	0
ant	1096	1045	366	342	1045	95.3467	51	0
antlr	212	212	107	107	212	100	0	0
aoi	0	0	0	0	0	0	0	0
argouml	2777	951	523	176	951	34.2456	1826	0
axion	0	0	0	0	0	0	0	0
azureus	0	0	0	0	0	0	0	0
colt	593	593	299	299	593	100	0	0
columba	1686	1	337	0	1	0.059312	1685	0
compiere	7457	694	511	39	694	9.30669	6763	0
derby	309	309	29	29	309	100	0	0
drjava	0	0	0	0	0	0	0	0
fit	96	96	38	38	96	100	0	0
fitlibrary	0	0	0	0	0	0	0	0
galleon	0	0	0	0	0	0	0	0
ganttproject	1088	77	639	46	77	7.07721	1011	0
geronimo	6	6	0	0	6	100	0	0
glassfish	0	0	0	0	0	0	0	0
hibernate	0	0	0	0	0	0	0	0
hsqldb	338	7	92	0	7	2.07101	331	0
ireport	0	0	0	0	0	0	0	0
jag	0	0	0	0	0	0	0	0
jaga	0	0	0	0	0	0	0	0
james	342	342	69	69	342	100	0	0
jasperreports	0	0	0	0	0	0	0	0
javacc	152	152	17	17	152	100	0	0
jboss	0	0	0	0	0	0	0	0
jchempaint	886	559	67	19	559	63.0926	327	0
jdk	0	0	0	0	0	0	0	0
jdom	0	0	0	0	0	0	0	0
jedit	0	0	0	0	0	0	0	0
jetty	415	333	99	99	333	80.241	82	0
jext	0	0	0	0	0	0	0	0
jfreechart	0	0	0	0	0	0	0	0
jgraph	186	186	83	83	186	100	0	0
jhotdraw	0	0	0	0	0	0	0	0
jmeter	873	399	141	49	399	45.7045	474	0
joggplayer	0	0	0	0	0	0	0	0
jrefactory	0	0	0	0	0	0	0	0
jtopen	10	10	1	1	10	100	0	0
jung	0	0	0	0	0	0	0	0
junit	601	404	357	298	404	67.2213	197	0
lucene	0	0	0	0	0	0	0	0
megamek	0	0	0	0	0	0	0	0
netbeans	1791	1791	1233	1233	1791	100	0	0
openoffice	0	0	0	0	0	0	0	0
pmd	925	17	177	4	17	1.83784	908	0
poi	2472	2472	473	473	2472	100	0	0
rssowl	0	0	0	0	0	0	0	0
sablecc	0	0	0	0	0	0	0	0
sandmark	0	0	0	0	0	0	0	0
scala	0	0	0	0	0	0	0	0
sequoia	206	206	34	34	206	100	0	0
tomcat	0	0	0	0	0	0	0	0

Tabelle A.7: „clean“-Tabelle zwischen BuildAnalyzer und ASM

project	BA	ASM	BA_inner	ASM_inner	BA∩ASM	BA∩ASM %	onlyBA	onlyASM
aglets_jar	0	0	0	0	0	0	0	0
ant_jar	1098	1	366	0	1	0.0910747	1097	0
antlr_jar	0	0	0	0	0	0	0	0
aoi_jar	662	1051	323	453	662	100	0	389
argouml_jar	4786	1914	1046	349	1837	38.3828	2949	77
axion_jar	445	448	64	64	445	100	0	3
azureus_jar	7942	7599	4974	4966	7594	95.6182	348	5
colt_jar	0	0	0	0	0	0	0	0
columba_jar	1689	1701	337	337	1689	100	0	12
compiere_jar	12550	1359	906	50	1357	10.8127	11193	2
derby_jar	0	0	0	0	0	0	0	0
drjava_jar	0	0	0	0	0	0	0	0
fit_jar	0	0	0	0	0	0	0	0
fitlibrary_jar	1492	1495	619	619	1492	100	0	3
galleon_jar	1227	6	1165	4	6	0.488998	1221	0
ganttproject_jar	1088	1016	639	595	1011	92.9228	77	5
geronimo_jar	0	0	0	0	0	0	0	0
glassfish_jar	0	0	0	0	0	0	0	0
hibernate_jar	0	0	0	0	0	0	0	0
hsqldb_jar	313	313	91	91	313	100	0	0
ireport_jar	2547	2540	1887	1887	2537	99.6074	10	3
jag_jar	0	0	0	0	0	0	0	0
jaga_jar	0	0	0	0	0	0	0	0
james_jar	0	0	0	0	0	0	0	0
jasperreports_jar	3832	1819	812	379	1766	46.0856	2066	53
javacc_jar	0	0	0	0	0	0	0	0
jboss_jar	322	105	107	6	2	0.621118	320	103
jchempaint_jar	887	51	67	13	51	5.74972	836	0
jdk_jar	0	0	0	0	0	0	0	0
jdom_jar	76	76	13	13	76	100	0	0
jedit_jar	1119	1132	692	700	1119	100	0	13
jetty_jar	416	84	99	0	82	19.7115	334	2
jext_jar	0	0	0	0	0	0	0	0
jfreechart_jar	509	4	17	0	4	0.785855	505	0
jgraph_jar	0	0	0	0	0	0	0	0
jhotdraw_jar	0	0	0	0	0	0	0	0
jmeter_jar	816	4	132	1	4	0.490196	812	0
joggplayer_jar	0	0	0	0	0	0	0	0
jrefactory_jar	0	0	0	0	0	0	0	0
jtopen_jar	0	0	0	0	0	0	0	0
jung_jar	38	38	18	18	38	100	0	0
junit_jar	397	209	118	59	209	52.6448	188	0
lucene_jar	736	743	345	345	736	100	0	7
megamek_jar	1778	1800	453	453	1778	100	0	22
netbeans_jar	2775	166	2015	8	5	0.18018	2770	161
openoffice_jar	0	0	0	0	0	0	0	0
pmd_jar	925	727	177	155	727	78.5946	198	0
poi_jar	0	0	0	0	0	0	0	0
rssowl_jar	0	0	0	0	0	0	0	0
sablecc_jar	285	286	89	89	285	100	0	1
sandmark_jar	0	0	0	0	0	0	0	0
scala_jar	0	0	0	0	0	0	0	0
sequoia_jar	0	0	0	0	0	0	0	0
tomcat_jar	0	0	0	0	0	0	0	0

Tabelle A.8: „clean“-Tabelle zwischen BuildAnalyzer und ASM_J

project	RFE	ASM	RFE_inner	RFE_innerF	ASM_inner	RFE∩ASM	RFE∩ASM %	onlyRFE	onlyASM
aglets	409	409	115	18	139	397	97.066	12	12
ant	1092	1045	131	204	342	975	89.2857	117	70
antlr	212	212	51	56	107	194	91.5094	18	18
aoi	0	0	0	0	0	0	0	0	0
argouml	2906	951	417	21	176	922	31.7275	1984	29
axion	0	0	0	0	0	0	0	0	0
azureus	0	0	0	0	0	0	0	0	0
colt	592	593	294	1	299	589	99.4932	3	4
columba	1681	1	297	0	0	1	0.0594884	1680	0
compiere	2519	694	145	8	39	415	16.4748	2104	279
derby	310	309	21	7	29	304	98.0645	6	5
drjava	0	0	0	0	0	0	0	0	0
fit	97	96	11	26	38	96	98.9691	1	0
fitlibrary	0	0	0	0	0	0	0	0	0
galleon	0	0	0	0	0	0	0	0	0
ganttproject	1077	77	451	15	46	74	6.87094	1003	3
geronimo	4	6	0	0	0	4	100	0	2
glassfish	0	0	0	0	0	0	0	0	0
hibernate	0	0	0	0	0	0	0	0	0
hsqldb	334	7	60	0	0	7	2.09581	327	0
ireport	0	0	0	0	0	0	0	0	0
jag	0	0	0	0	0	0	0	0	0
jaga	0	0	0	0	0	0	0	0	0
james	330	342	41	13	69	318	96.3636	12	24
jasperreports	0	0	0	0	0	0	0	0	0
javacc	0	0	0	0	0	0	0	0	0
jboss	0	0	0	0	0	0	0	0	0
jchempaint	882	559	52	1	19	558	63.2653	324	1
jdk	0	0	0	0	0	0	0	0	0
jdom	0	0	0	0	0	0	0	0	0
jedit	0	0	0	0	0	0	0	0	0
jetty	409	333	47	39	99	312	76.2836	97	21
jext	0	0	0	0	0	0	0	0	0
jfreechart	0	0	0	0	0	0	0	0	0
jgraph	370	186	116	56	83	216	58.3784	154	-30
jhotdraw	0	0	0	0	0	0	0	0	0
jmeter	863	399	81	17	49	396	45.8864	467	3
joggplayer	0	0	0	0	0	0	0	0	0
jrefactory	0	0	0	0	0	0	0	0	0
jtopen	0	0	0	0	0	0	0	0	0
jung	0	0	0	0	0	0	0	0	0
junit	579	404	104	223	298	355	61.3126	224	49
lucene	0	0	0	0	0	0	0	0	0
megamek	0	0	0	0	0	0	0	0	0
netbeans	1738	1791	496	601	1233	1572	90.4488	166	219
openoffice	0	0	0	0	0	0	0	0	0
pmd	914	17	94	1	4	16	1.75055	898	1
poi	2426	2472	185	263	473	2360	97.2795	66	112
rssowl	0	0	0	0	0	0	0	0	0
sablecc	0	0	0	0	0	0	0	0	0
sandmark	0	0	0	0	0	0	0	0	0
scala	0	0	0	0	0	0	0	0	0
sequoia	207	206	15	18	34	205	99.0338	2	1
tomcat	0	0	0	0	0	0	0	0	0

Tabelle A.9: „clean“-Tabelle zwischen RFE und ASM

project	RFE	ASM	RFE_inner	RFE_innerF	ASM_inner	ASM∩RFE	ASM∩RFE %	onlyRFE	onlyASM
aglets	409	409	115	18	139	397	97.066	12	12
ant	1092	1045	131	204	342	975	93.3014	117	70
antlr	212	212	51	56	107	194	91.5094	18	18
aoi	0	0	0	0	0	0	0	0	0
argouml	2906	951	417	21	176	922	96.9506	1984	29
axion	0	0	0	0	0	0	0	0	0
azureus	0	0	0	0	0	0	0	0	0
colt	592	593	294	1	299	589	99.3255	3	4
columba	1681	1	297	0	0	1	100	1680	0
compiere	2519	694	145	8	39	415	59.7983	2104	279
derby	310	309	21	7	29	304	98.3819	6	5
drjava	0	0	0	0	0	0	0	0	0
fit	97	96	11	26	38	96	100	1	0
fitlibrary	0	0	0	0	0	0	0	0	0
galleon	0	0	0	0	0	0	0	0	0
ganttproject	1077	77	451	15	46	74	96.1039	1003	3
geronimo	4	6	0	0	0	4	66.6667	0	2
glassfish	0	0	0	0	0	0	0	0	0
hibernate	0	0	0	0	0	0	0	0	0
hsqldb	334	7	60	0	0	7	100	327	0
ireport	0	0	0	0	0	0	0	0	0
jag	0	0	0	0	0	0	0	0	0
jaga	0	0	0	0	0	0	0	0	0
james	330	342	41	13	69	318	92.9825	12	24
jasperreports	0	0	0	0	0	0	0	0	0
javacc	0	0	0	0	0	0	0	0	0
jboss	0	0	0	0	0	0	0	0	0
jchempaint	882	559	52	1	19	558	99.8211	324	1
jdk	0	0	0	0	0	0	0	0	0
jdom	0	0	0	0	0	0	0	0	0
jedit	0	0	0	0	0	0	0	0	0
jetty	409	333	47	39	99	312	93.6937	97	21
jext	0	0	0	0	0	0	0	0	0
jfreechart	0	0	0	0	0	0	0	0	0
jgraph	370	186	116	56	83	216	116.129	154	-30
jhotdraw	0	0	0	0	0	0	0	0	0
jmeter	863	399	81	17	49	396	99.2481	467	3
joggplayer	0	0	0	0	0	0	0	0	0
jrefactory	0	0	0	0	0	0	0	0	0
jtopen	0	0	0	0	0	0	0	0	0
jung	0	0	0	0	0	0	0	0	0
junit	579	404	104	223	298	355	87.8713	224	49
lucene	0	0	0	0	0	0	0	0	0
megamek	0	0	0	0	0	0	0	0	0
netbeans	1738	1791	496	601	1233	1572	87.7722	166	219
openoffice	0	0	0	0	0	0	0	0	0
pmd	914	17	94	1	4	16	94.1176	898	1
poi	2426	2472	185	263	473	2360	95.4693	66	112
rssowl	0	0	0	0	0	0	0	0	0
sablecc	0	0	0	0	0	0	0	0	0
sandmark	0	0	0	0	0	0	0	0	0
scala	0	0	0	0	0	0	0	0	0
sequoia	207	206	15	18	34	205	99.5146	2	1
tomcat	0	0	0	0	0	0	0	0	0

Tabelle A.10: „clean“-Tabelle zwischen ASM und RFE

project	RFE	ASM	RFE_inner	RFE_innerF	ASM_inner	RFE∩ASM	RFE∩ASM %	onlyRFE	onlyASM
aglets_jar	0	0	0	0	0	0	0	0	0
ant_jar	1092	1	131	0	0	1	0.0915751	1091	0
antlr_jar	0	0	0	0	0	0	0	0	0
aoi_jar	663	1024	246	42	446	618	93.2127	45	406
argouml_jar	2906	1899	417	64	335	1881	64.7281	1025	18
axion_jar	437	445	39	14	61	429	98.1693	8	16
azureus_jar	7588	7489	3985	251	4855	6760	89.088	828	729
colt_jar	0	0	0	0	0	0	0	0	0
columba_jar	1681	1701	297	12	337	1660	98.7507	21	41
compiere_jar	2519	1358	145	13	50	1167	46.3279	1352	191
derby_jar	0	0	0	0	0	0	0	0	0
drjava_jar	0	0	0	0	0	0	0	0	0
fit_jar	0	0	0	0	0	0	0	0	0
fitlibrary_jar	1478	1475	361	226	599	1453	98.3085	25	22
galleon_jar	804	6	406	1	4	5	0.621891	799	1
ganttproject_jar	1077	1000	451	88	579	865	80.3157	212	135
geronimo_jar	0	0	0	0	0	0	0	0	0
glassfish_jar	0	0	0	0	0	0	0	0	0
hibernate_jar	0	0	0	0	0	0	0	0	0
hsqldb_jar	334	312	60	25	90	298	89.2216	36	14
ireport_jar	2534	2535	1835	36	1882	2515	99.2502	19	20
jag_jar	0	0	0	0	0	0	0	0	0
jaga_jar	0	0	0	0	0	0	0	0	0
james_jar	0	0	0	0	0	0	0	0	0
jasperreports_jar	1711	1808	159	144	368	1673	97.7791	38	135
javacc_jar	0	0	0	0	0	0	0	0	0
jboss_jar	306	67	41	0	6	2	0.653595	304	65
jchempaint_jar	882	51	52	0	13	51	5.78231	831	0
jdk_jar	0	0	0	0	0	0	0	0	0
jdom_jar	149	76	12	14	13	81	54.3624	68	-5
jedit_jar	1095	1121	389	231	689	974	88.9498	121	147
jetty_jar	409	84	47	0	0	84	20.5379	325	0
jext_jar	0	0	0	0	0	0	0	0	0
jfreechart_jar	511	4	7	0	0	4	0.782779	507	0
jgraph_jar	0	0	0	0	0	0	0	0	0
jhotdraw_jar	0	0	0	0	0	0	0	0	0
jmeter_jar	863	4	81	0	1	4	0.463499	859	0
joggplayer_jar	0	0	0	0	0	0	0	0	0
jrefactory_jar	0	0	0	0	0	0	0	0	0
jtopen_jar	0	0	0	0	0	0	0	0	0
jung_jar	34	38	1	14	18	34	100	0	4
junit_jar	579	182	104	5	57	173	29.8791	406	9
lucene_jar	725	734	142	155	336	681	93.931	44	53
megamek_jar	1781	1797	386	38	450	1770	99.3824	11	27
netbeans_jar	1738	70	496	0	3	3	0.172612	1735	67
openoffice_jar	0	0	0	0	0	0	0	0	0
pmd_jar	914	722	94	56	150	679	74.2888	235	43
poi_jar	0	0	0	0	0	0	0	0	0
rssowl_jar	0	0	0	0	0	0	0	0	0
sablecc_jar	267	286	55	14	89	265	99.2509	2	21
sandmark_jar	0	0	0	0	0	0	0	0	0
scala_jar	0	0	0	0	0	0	0	0	0
sequoia_jar	0	0	0	0	0	0	0	0	0
tomcat_jar	0	0	0	0	0	0	0	0	0

Tabelle A.11: „clean“-Tabelle zwischen RFE und ASM_J

project	RFE	ASM	RFE_inner	RFE_innerF	ASM_inner	ASM∩RFE	ASM∩RFE %	onlyRFE	onlyASM
aglets_jar	0	0	0	0	0	0	0	0	0
ant_jar	1092	1	131	0	0	1	100	1091	0
antlr_jar	0	0	0	0	0	0	0	0	0
aoi_jar	663	1024	246	42	446	618	60.3516	45	406
argouml_jar	2906	1899	417	64	335	1881	99.0521	1025	18
axion_jar	437	445	39	14	61	429	96.4045	8	16
azureus_jar	7588	7489	3985	251	4855	6760	90.2657	828	729
colt_jar	0	0	0	0	0	0	0	0	0
columba_jar	1681	1701	297	12	337	1660	97.5897	21	41
compiere_jar	2519	1358	145	13	50	1167	85.9352	1352	191
derby_jar	0	0	0	0	0	0	0	0	0
drjava_jar	0	0	0	0	0	0	0	0	0
fit_jar	0	0	0	0	0	0	0	0	0
fitlibrary_jar	1478	1475	361	226	599	1453	98.5085	25	22
galleon_jar	804	6	406	1	4	5	83.3333	799	1
ganttproject_jar	1077	1000	451	88	579	865	86.5	212	135
geronimo_jar	0	0	0	0	0	0	0	0	0
glassfish_jar	0	0	0	0	0	0	0	0	0
hibernate_jar	0	0	0	0	0	0	0	0	0
hsqldb_jar	334	312	60	25	90	298	95.5128	36	14
ireport_jar	2534	2535	1835	36	1882	2515	99.211	19	20
jag_jar	0	0	0	0	0	0	0	0	0
jaga_jar	0	0	0	0	0	0	0	0	0
james_jar	0	0	0	0	0	0	0	0	0
jasperreports_jar	1711	1808	159	144	368	1673	92.5332	38	135
javacc_jar	0	0	0	0	0	0	0	0	0
jboss_jar	306	67	41	0	6	2	2.98507	304	65
jchempaint_jar	882	51	52	0	13	51	100	831	0
jdk_jar	0	0	0	0	0	0	0	0	0
jdom_jar	149	76	12	14	13	81	106.579	68	-5
jedit_jar	1095	1121	389	231	689	974	86.8867	121	147
jetty_jar	409	84	47	0	0	84	100	325	0
jext_jar	0	0	0	0	0	0	0	0	0
jfreechart_jar	511	4	7	0	0	4	100	507	0
jgraph_jar	0	0	0	0	0	0	0	0	0
jhotdraw_jar	0	0	0	0	0	0	0	0	0
jmeter_jar	863	4	81	0	1	4	100	859	0
joggplayer_jar	0	0	0	0	0	0	0	0	0
jrefactory_jar	0	0	0	0	0	0	0	0	0
jtopen_jar	0	0	0	0	0	0	0	0	0
jung_jar	34	38	1	14	18	34	89.4737	0	4
junit_jar	579	182	104	5	57	173	95.0549	406	9
lucene_jar	725	734	142	155	336	681	92.7793	44	53
megamek_jar	1781	1797	386	38	450	1770	98.4975	11	27
netbeans_jar	1738	70	496	0	3	3	4.28571	1735	67
openoffice_jar	0	0	0	0	0	0	0	0	0
pmd_jar	914	722	94	56	150	679	94.0443	235	43
poi_jar	0	0	0	0	0	0	0	0	0
rssowl_jar	0	0	0	0	0	0	0	0	0
sablecc_jar	267	286	55	14	89	265	92.6573	2	21
sandmark_jar	0	0	0	0	0	0	0	0	0
scala_jar	0	0	0	0	0	0	0	0	0
sequoia_jar	0	0	0	0	0	0	0	0	0
tomcat_jar	0	0	0	0	0	0	0	0	0

Tabelle A.12: „clean“-Tabelle zwischen ASM_J und RFE

ANHANG B

DATENTRÄGER

Auf der CD sind die Programmquellen des FactComparator enthalten, sowie alle Venn Diagramme, „compare“- und „diff“-Dateien die von diesem über den Corpus *JavaShape* erzeugt worden sind.