

# Analyse von statischem Coupling in Source- und Bytecode mithilfe eines selbst implementierten Tools anhand der objekt-orientierten Sprache Java

Bachelorarbeit

Hannah S. Fischer und Yannick Josuttis

15. September 2021

CHRISTIAN-ALBRECHTS-UNIVERSITÄT ZU KIEL  
INSTITUT FÜR INFORMATIK  
ARBEITSGRUPPE SOFTWARE ENGINEERING

Betreut durch: Priv.-Doz. Dr. Henning Schnoor  
Dr.-Ing. Reiner Jung



### **Eidesstattliche Erklärung**

Hiermit erklären wir an Eides statt, dass wir die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet haben.

Kiel, 15. September 2021

---



# Abstract

Heutzutage ist die Wartbarkeit und Wiederverwendbarkeit von Software ein entscheidendes Kriterium für eine erfolgreiche Entwicklung und Weiterentwicklungen. Coupling ist dabei ein wichtiges Messinstrument, um die Herausforderungen zu meistern. In diesem Prozess stellt Sourcecode das Produkt des Softwareentwicklers dar, welches jedoch nur einer Abstraktion des tatsächlich kompilierten Codes entspricht. Deshalb ist gerade die Unterscheidung des Couplings in Source- und Bytecode von entscheidender Bedeutung, da Bytecode dem tatsächlich ausgeführten Code entspricht. Für diese Zwecke haben wir ein Analysetool entwickelt, welches das Source- und Bytecodecoupling analysieren kann. Diese Arbeit stellt wichtige Unterscheidungen zwischen der Source- und Bytecodeanalyse dar und evaluiert diese anhand von drei bekannten Softwareprojekten.



# Inhaltsverzeichnis

<b>1</b>	<b>Einleitung</b>	<b>1</b>
1.1	Motivation . . . . .	1
1.2	Ziele . . . . .	2
1.2.1	Z1: Coupling Metriken für Java Source- und Bytecode festlegen . . .	2
1.2.2	Z2: Analysetool für statisches Coupling für Java entwickeln . . . . .	2
1.2.3	Z3: Evaluierung von Messergebnissen verschiedener Projekte . . . . .	2
1.3	Aufbau . . . . .	2
<b>2</b>	<b>Grundlagen und Technologien</b>	<b>5</b>
2.1	Coupling und Metriken . . . . .	5
2.2	Byte Code Engineering Library . . . . .	6
2.3	JavaParser und SymbolSolver . . . . .	6
2.4	Abstract Syntax Tree . . . . .	6
2.5	JavaFX . . . . .	7
<b>3</b>	<b>Auswahl und Definition von Coupling Metriken</b>	<b>9</b>
3.1	Method-to-Method Coupling . . . . .	9
3.2	Package Coupling . . . . .	10
3.3	Inheritance Coupling . . . . .	10
3.4	Import Coupling . . . . .	12
3.5	Field Coupling . . . . .	12
3.6	Eine weiterführende Metrik: Structural Debt Index . . . . .	13
<b>4</b>	<b>Besonderheiten der Source- und Bytecodeanalyse</b>	<b>15</b>
4.1	Versteckte Funktionalität . . . . .	15
4.1.1	String-Operationen . . . . .	15
4.1.2	Try-Catch-Blöcke . . . . .	17
4.1.3	Lambda Funktionen . . . . .	18
4.1.4	Importe . . . . .	19
4.1.5	Versteckte Klassen . . . . .	19
4.2	Abhängigkeiten von Superklassen . . . . .	20
4.3	Fehler bei Namensauflösungen . . . . .	20
4.3.1	Methodenketten . . . . .	20
4.3.2	Exceptions . . . . .	21
4.4	Generische Typen . . . . .	21

## Inhaltsverzeichnis

<b>5 Funktionsweise des Analysetools</b>	<b>23</b>
5.1 Anwendung des JavaParsers und SymbolSolvers . . . . .	23
5.2 Anwendung der BCEL . . . . .	26
5.3 Filter für projektinterne Funktionen . . . . .	28
5.4 Implementierung der Metriken . . . . .	28
5.5 Sammlung von Metadaten . . . . .	29
<b>6 Evaluierung</b>	<b>31</b>
6.1 Ziele und Erwartungen . . . . .	31
6.2 Analyse: Mockito . . . . .	32
6.3 Analyse: Guava . . . . .	33
6.4 Analyse: Netflix Hollow . . . . .	35
6.5 Allgemeine Auffälligkeiten der Analyse der Projekte . . . . .	37
6.6 Vergleich von Source- und Bytecode Ergebnissen . . . . .	38
<b>7 Quick Start Guide</b>	<b>41</b>
<b>8 Verwandte Arbeiten</b>	<b>45</b>
<b>9 Fazit und Ausblick</b>	<b>47</b>
9.1 Fazit . . . . .	47
9.2 Ausblick . . . . .	48
<b>Bibliografie</b>	<b>49</b>



# Einleitung

## 1.1. Motivation

In der heutigen Zeit wird Software immer größer und komplexer. Zusätzlich wird Software immer häufiger wiederverwendet, angepasst und integriert. Dadurch ist Qualität ein immer wichtiger werdender Aspekt. Qualität kann hierbei anhand von Wartbarkeit, Wiederverwendbarkeit/Änderbarkeit und Lesbarkeit bewertet werden. Zwei entscheidende Messinstrumente hierfür sind Coupling und Kohäsion. Wir wollen uns mit Coupling auseinandersetzen. Genauer gesagt konzentrieren wir uns auf statisches Coupling. Das Ziel bei Coupling besteht generell darin, die Anzahl der Verbindungen gering zu halten, um so die Qualität in den genannten Punkten zu Verbessern.

Wir wollen sowohl anhand der objekt-orientierten Programmiersprache Java das statische Coupling des Sourcecodes als auch des Bytecodes betrachten. Die Betrachtung beider ist deshalb interessant, da Sourcecode eher die Perspektive der Entwickler und Bytecode die, der Java Virtual Machine wiedergibt. Die bereits genannten Aspekte für Qualität werden von Entwicklern im Sourcecode umgesetzt und wahrscheinlich in den seltensten Fällen im Bytecode. Aus diesen Gründen ist es interessant das statische Coupling von Source- und Bytecode gegenüber zu stellen und insbesondere, ob es Ungleichgewichte zu Gunsten oder Ungunsten einer Seite kommt.

Für das Messen von statischem Coupling im Java-Sourcecode gibt es zurzeit kaum Tools bzw. sind diese nicht zugänglich oder meist kostenpflichtig. Hinzu kommt, dass die Tools zur Sourcecodeanalyse häufig intern auf den Bytecode zurückgreifen und somit keine 'richtige' Sourcecodeanalyse durchführen. Da wir die Source- und Bytecodeanalysen miteinander vergleichen wollen, ist dies von essentieller Bedeutung. Deshalb haben wir zu diesem Zweck ein eigenes Tool, welches statisches Coupling von Java-Code analysieren kann, entwickelt. Um eine bessere Vergleichbarkeit von Source- und Bytecode zu ermöglichen, haben wir uns dazu entschieden, ebenfalls die Bytecodeanalyse selber zu implementieren. Dafür nutzen wir, wie viele andere Analysetools, die Bytecode Engineering Library von Apache Commons.

## 1. Einleitung

### 1.2. Ziele

#### 1.2.1. Z1: Coupling Metriken für Java Source- und Bytecode festlegen

Coupling lässt sich auf verschiedene Arten messen. Unser Ziel ist es, für unsere Zwecke geeignete Metriken festzulegen. Das heißt, wir wollen Metriken verwenden, die für die objekt-orientierte Sprache Java geeignet sind. Darüber hinaus müssen die Metriken sowohl für Source- als auch Bytecode sinnvoll angewendet werden können.

#### 1.2.2. Z2: Analysetool für statisches Coupling für Java entwickeln

Das Hauptziel besteht darin, ein Analysetool zu entwickeln, welches das statische Coupling von Software, die mit der objekt-orientierten Sprache Java formuliert wurde, zu messen. Dabei soll das Coupling von Source- und von Bytecode analysiert werden.

#### 1.2.3. Z3: Evaluierung von Messergebnissen verschiedener Projekte

Als weiteres Ziel wollen wir unser Analysetool auf verschiedene Java-Projekte anwenden. Dabei betrachten wir verschiedene Coupling Metriken, um eine Aussage über den Grad des Couplings treffen zu können. Des Weiteren ist es unser Ziel, die Ergebnisse weiter darauf zu analysieren, inwiefern verschiedene Arten von Coupling miteinander korrelieren, um Abhängigkeiten zwischen unterschiedlichen Metriken feststellen zu können. Darüber hinaus wollen wir untersuchen, ob es Unterschiede zwischen Coupling im Source- und Bytecode gibt, die zum Beispiel durch mögliche Compileroptimierungen entstehen könnten.

### 1.3. Aufbau

Die dargestellten Themen in dieser Ausarbeitung und die Implementierung des Analyse-tools, teilen sich wie folgt auf: Hannah S. Fischer hat sich mit den konzeptuellen Aspekten sowie der Programmierung unterstützender Programmstrukturen und dem Testen des Programms beschäftigt. Yannick Josuttis hat sich mit den Unterschieden der Source- und Bytecodeanalyse, den grundlegenden Implementierungen der Tool-Spezifischen Aspekten sowie der Gestaltung des Userinterfaces auseinandergesetzt. Die Evaluierung der Ergebnisse aus den Analysen ist gemeinschaftlich erarbeitet.

In Kapitel *Grundlagen und Technologien* wird eine Einführung in das Coupling gegeben sowie die verwendeten Tools für die spätere Implementierung und die damit in Zusammenhang stehenden Strukturen vorgestellt. Im darauf folgenden Kapitel *Auswahl und Definition von Coupling Metriken* werden die von dem Analysetool verwendeten Metriken eingeführt und definiert. Das Kapitel *Besonderheiten der Source- und Bytecodeanalyse* beschreibt die Unterschiede und Spezialfälle, die während der Implementierung aufgefallen sind. Das

### 1.3. Aufbau

dann folgende Kapitel *Funktionsweise des Analysetools* stellt die Kernimplementierungen und die Einbindung der verwendeten Technologien des Analysetools dar. Im Kapitel *Evaluierung* werden die Ergebnisse der Analysen von drei Projekten diskutiert und die Zusammenhänge zwischen diesen, sowie die Unterschiede der Ergebnisse des Source- und Bytecodes ausgewertet. Im *Quick Start Guide* werden Hinweise und Hilfen für die Verwendung des Analysetools gegeben. Verwandte Arbeiten werden in *Kapitel 8* diskutiert. *Kapitel 9* zieht ein Fazit und nennt mögliche weiterführende Arbeiten.



# Grundlagen und Technologien

## 2.1. Coupling und Metriken

Coupling bezeichnet die Beziehung zwischen zwei oder mehreren Komponenten innerhalb einer Software und beschreibt damit den Grad der Abhängigkeit dieser untereinander. Es stellt somit eine Möglichkeit dar, die Qualität und Komplexität einer Software zu messen. Dabei kann zwischen vier verschiedenen Arten von Coupling für objekt-orientierte Sprachen unterschieden werden [„A survey on software coupling relations and tools“]:

Strukturelles Coupling wird auch als statisches Coupling bezeichnet, da es die statischen Zusammenhänge in einer Software betrachtet. Dabei werden beispielsweise Methodenaufrufe und Vererbungsbeziehungen zwischen Klassen oder auch Zugriffe auf Variablen aus anderen Klassen als Coupling eingestuft.

Dynamisches Coupling betrachtet ähnlich wie das strukturelle die Aufrufe zwischen den Klassen, jedoch erst zur Laufzeit. Somit können auch Beziehungen zwischen Komponenten gemessen werden, welche durch eine statische Analyse nicht gefunden werden können. Ein Beispiel dafür sind Verbindungen mit komplexeren Vererbungsbeziehungen, die durch Polymorphismus und dynamischer Bindung auftreten können [„Comparing Static and Dynamic Weighted Software Coupling Metrics“].

Eine weitere Art, Beziehungen zwischen Klassen einer Software zu analysieren, ist semantisches Coupling, wobei konzeptuelle Zusammenhänge betrachtet werden. Diese werden anhand von Kommentaren und Indikatoren ermittelt.

Als logisches Coupling oder auch als evolutionäres Coupling werden Beziehungen bezeichnet, welche durch ähnliche Änderungsmuster in einer Release-History auftreten. Ziel dieses Couplings ist es, Klassen zu identifizieren, welche ein vergleichbares Änderungsverhalten aufweisen.

Um das Coupling einer Software messen zu können, werden Tools verwendet, welche anhand von Metriken den Grad der Abhängigkeit zwischen zwei Modulen bestimmen können [„Tool for Measuring Coupling in Object-Oriented Java Software“]. Eine bekannte Metrik ist *Coupling between Objects* (CBO), welche von Chidamber und Kemerer erstmals eingeführt wurde [„Towards a Metrics Suite for Object Oriented Design“]. Sie stellt den Grad der Interaktion eines Objektes mit einem anderen dar. Angelehnt an diese Metrik lassen sich noch weitere spezifischere Metriken definieren, welche wir in Kapitel 3 einführen und genauer beschreiben.

## 2. Grundlagen und Technologien

### 2.2. Byte Code Engineering Library

Die *Byte Code Engineering Library* (BCEL)<sup>1</sup> von Apache Commons ermöglicht es Java Klassendateien zu analysieren, zu erstellen und zu manipulieren. Dabei werden die Klassen als Objekte dargestellt und beinhalten symbolische Informationen, wie Methoden, Felder und Bytecode-Befehle, der jeweiligen Klasse [*Apache Commons BCEL*]. Damit stellt die BCEL ein Tool dar, welches zur statischen Analyse von Java Klassendateien verwendet werden kann. Wir wollen die BCEL nutzen, um Aufrufe im Bytecode zwischen Klassen zählen und benennen zu können. Dadurch ermöglicht diese Library das Zählen von statischem Coupling in Java-Klassendateien.

### 2.3. JavaParser und SymbolSolver

Um Aufrufe im Sourcecode zwischen Klassen zählen und benennen zu können, nutzen wir den *JavaParser* <sup>2</sup>. Dieser stellt eine Library zur Verfügung, welche es ermöglicht, Java-Sourcecode in Objekte zu parsen, um anhand dessen einen Abstract Syntax Tree (AST) zu erstellen. Auch werden Mechanismen bereitgestellt, die es erlauben durch den AST zu navigieren und mit den Objekten zu interagieren [*JavaParser: Visited*]. Der SymbolSolver<sup>2</sup> stellt ein zusätzliches Tool dar, welches vom JavaParser bereitgestellt wird. Dadurch wird eine Funktion zur Verfügung gestellt, welche es ermöglicht Beziehungen zwischen Knoten aus einem AST oder auch Verbindungen über mehrere AST zu finden und Referenzen bzw. Namen aufzulösen [*JavaParser: Visited*].

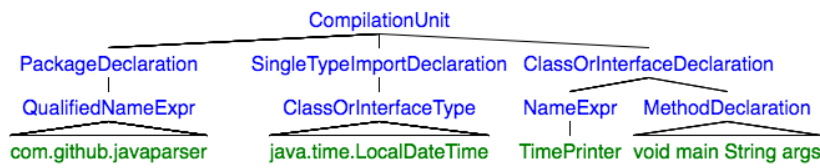
### 2.4. Abstract Syntax Tree

Ein Abstract Syntax Tree (AST) beschreibt in der Informatik einen Baum, welcher auf abstrakte Art und Weise die syntaktische Struktur von Sourcecode darstellt. Dabei repräsentiert die Wurzel des AST die Datei des Sourcecodes und wird im JavaParser beispielsweise als *CompilationUnit* bezeichnet. Jeder Knoten wiederum stellt ein Element der Datei dar. Die Kinder der Wurzel stellen Packages, Importe und Klassen- oder Interfacedeklarationen dar. Die Nachfahren repräsentieren immer spezifischer werdende Elemente einer Datei. Ein Ast beschreibt zum Beispiel ein Statement [*JavaParser: Visited*]. Die Abbildung 2.1 zeigt ein Beispiel eines AST.

---

<sup>1</sup><https://github.com/apache/commons-bcel>

<sup>2</sup><https://github.com/javaparser/javaparser>



Quelle: *JavaParser: Visited*

Abbildung 2.1. AST des JavaParsers

## 2.5. JavaFX

JavaFX [*JavaFX*] war bis JDK 11 ein Bestandteil der Java-Core-Implementierung. Seitdem wurde es als eigenständiges Modul weiterentwickelt. JavaFx ermöglicht eine möglichst einfache Gestaltung von Userinterfaces. Es kann für Desktop-Applikationen, mobile und eingebettete Systeme genutzt werden. Wir nutzen JavaFx für das grafische Userinterface, um möglichst einfach Dateipfade auszuwählen und die Ergebnisse der Analyse in interaktiven Tabellen anzeigen zu lassen. Daneben kann der User auch Metriken an- und abwählen und zwischen den Ergebnissen von Source- und Bytecode wechseln.





# Auswahl und Definition von Coupling Metriken

Um Coupling messen zu können, werden Metriken benötigt, die verschiedene Arten von Beziehungen zwischen Komponenten einer Software beschreiben. Bevor eine Couplinganalyse durchgeführt werden kann, muss entschieden werden, auf Basis welcher Metriken dies umgesetzt werden soll. Im Folgenden wollen wir Metriken definieren, die in diesem Projekt als Grundlage der Couplinganalyse verwendet werden.

## 3.1. Method-to-Method Coupling

Method-to-Method Coupling beschreibt, dass eine Methode einer Klasse eine Methode einer anderen Klasse aufruft. Bei der Definition orientieren wir uns an der von Chidamber und Kemerer eingeführten Metrik *Coupling between Objects (CBO)*. Diese besagt, CBO ist eine Zählweise von nicht durch Vererbungsbeziehung entstandenen Paaren zu anderen Klassen und beschreibt somit, dass Coupling zwischen zwei Objekten entsteht, sobald diese miteinander agieren [„Towards a Metrics Suite for Object Oriented Design“]. Da wir statisches Coupling betrachten, kann man klassischerweise nicht von Objekten sprechen, sondern eher von Aufrufen zwischen Klassen bzw. Klasseninstanzen. In einer späteren Arbeit erweitern Chidamber und Kemerer ihre Definition dahingehend, dass auch durch Vererbungsbeziehung entstandenes Coupling zwischen Objekten mit eingeschlossen wird [„A metrics suite for object oriented design“]. Das bedeutet, auch Methodenaufrufe in eine Superklasse werden als Coupling gedeutet. Auch wir beziehen Coupling zwischen vererbten Klassen mit ein.

Die Metrik Method-to-Method Coupling beinhaltet Methodenaufrufe auf Klasseninstanzen, statische Aufrufe und Aufrufe auf Schlüsselwörtern wie *super* oder *this*, wobei wir letzteres nicht als Coupling zählen, da es einem Aufruf in die aktuelle Klasse entsprechen würde und somit eine Kohäsion darstellt. Konstruktoraufrufe werden nicht als Method-to-Method Coupling gezählt, da diese keinem Aufruf auf einer Klasseninstanz entsprechen, sondern diese erzeugen. Werden Methodenaufrufe aus einer inneren Klasse getätigt oder wird eine Methode aus einer inneren Klasse aufgerufen, zählen wir dies als Coupling zu oder ausgehend von ihrer äußeren Klasse. Aufrufe zwischen äußeren und ihren inneren Klassen werden hingegen nicht gezählt, da wir diese als ein zusammenhängendes Modul

### 3. Auswahl und Definition von Coupling Metriken

interpretieren. Betrachten wir Aufrufe von Methoden, die durch ein Interface zur Verfügung gestellt werden: Einfache Aufrufe dieser Methoden werden als Coupling zu der Klasse gezählt, die diese implementiert. Wird diese Methode jedoch auf einem Objekt, welches als Interface gecasted wurde, aufgerufen, stellt dies hingegen ein Coupling zum Interface dar.

Listing 3.1 zeigt die verschiedenen Methodenaufrufe, die als Method-to-Method Coupling zu den jeweiligen Klassen, in denen die aufgerufenen Methoden implementiert wurden, gezählt werden.

**Listing 3.1.** Beispielaufrufe für Method-to-Method Coupling

```
1  public class M2MExample extends SuperClass{
2      private Type var;
3      private void method(){
4          var.method2();           // Methodenaufruf auf einer Variablen
5          AnotherClass.method3(); // statischer Methodenaufruf
6          super.method4();        // Methodenaufruf zur Superklasse
7      }
8  }
```

### 3.2. Package Coupling

Die Metrik Package Coupling definieren wir auf Grundlage der bereits eingeführten Metrik Method-to-Method Coupling. Sie zählt Verbindungen zwischen Packages, die durch Method-to-Method Coupling entstanden sind. Das bedeutet, es wird eine Verbindung zwischen diesen registriert, sobald ein Coupling zwischen zwei Klassen aus unterschiedlichen Packages durch einen Methodenaufruf gezählt wurde. Dabei wird das Coupling zu dem Package gezählt, in dem sich die aufgerufene Klasse befindet. Das heißt, dass die Couplings der Subpackages nicht zu denen eines möglichen übergeordneten Packages dazugezählt werden. Sind in einem übergeordneten Package keine Klassen definiert, so kann dieses Package auch kein Coupling erzeugen oder von einer Klasse aus einem anderen Package aufgerufen werden. Diese Zählweise wurde gewählt, um die Verbindungen besser nachvollziehen zu können und die Komplexität der Zählung zu verringern.

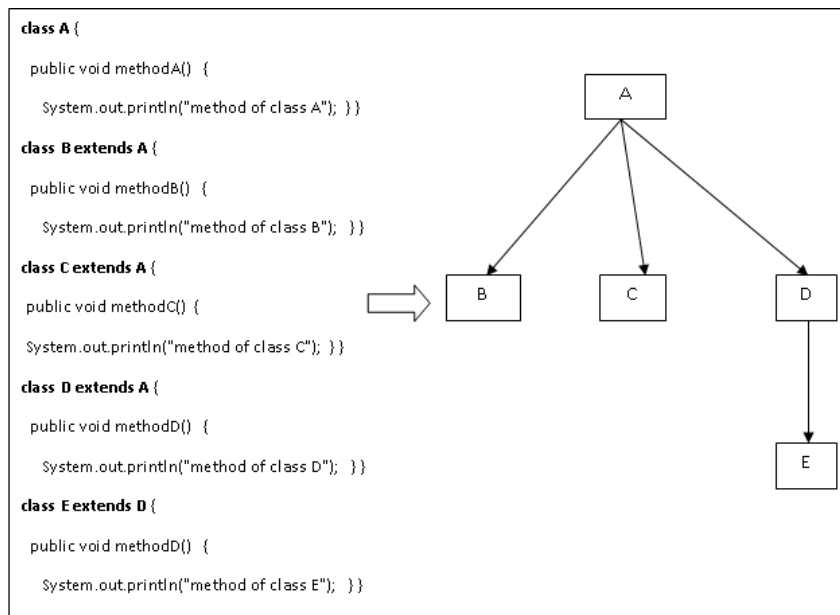
Die hier definierte Metrik dient dazu, logische Einheiten innerhalb einer Software darzustellen, um somit eine Betrachtungsweise der Beziehungen von Klassen auf einer anderen Ebene zu ermöglichen. Dies soll als Analyse des Couplings innerhalb der Software und in Bezug auf dessen Struktur fungieren.

### 3.3. Inheritance Coupling

Für die Metrik Inheritance Coupling orientieren wir uns an einer Arbeit von Bidve und Sarasu, die sich ebenfalls mit dem Messen von Coupling in objekt-orientierten Program-

### 3.3. Inheritance Coupling

miersprachen beschäftigt haben. Sie beschreiben Inheritance Coupling durch Beziehungen, die durch konkrete Vererbungsbeziehungen entstehen [„Tool for Measuring Coupling in Object-Oriented Java Software“]. Auch wir definieren unsere Metrik über solche Beziehungen. Inheritance Coupling beschreibt also Verbindungen, die durch die Schlüsselwörter *extends* und *implements* entstehen und zählt somit ein Coupling zwischen der Subklasse und ihrer Superklasse. Da im Bytecode diese Schlüsselwörter nicht existieren, müssen wir an dieser Stelle eine andere Art der Zählweise für die Bytecodeanalyse wählen. Dafür kann auf die von der BCEL zur Verfügung gestellten Funktion der Abfrage der Superklassen und der verwendeten Interfaces zurückgegriffen werden. Im Gegensatz zu Bidve und Sarasu, die zusätzlich die Rückverbindung, also Beziehung zwischen der Superklasse und der Subklasse zählen, berücksichtigt unsere Metrik diese nicht. Dies hat den Grund, dass in der Sprache Java die Subklasse zwar ihre Superklasse kennt und die Attribute und Methoden dieser verwenden kann, andersherum jedoch nicht.



Quelle: „Tool for Measuring Coupling in Object-Oriented Java Software“

**Abbildung 3.1.** Klassenhierarchie und Vererbungsbeziehungen

Inheritance Coupling beinhaltet des Weiteren nur direkte Vererbungsbeziehungen und zählt keine transitiven Verbindungen. Abbildung 3.1 zeigt eine Klassenhierarchie, die durch Vererbungsbeziehungen aufgebaut ist. In diesem Fall zählt Inheritance Coupling eine Verbindung zwischen den Klassen A und B, zwischen den Klassen A und C und zwischen den Klassen A und D, da diese in einer direkten Vererbungsbeziehung zueinanderstehen. Es wird jedoch kein Coupling zwischen den Klassen A und E gezählt, da diese nur eine

### 3. Auswahl und Definition von Coupling Metriken

indirekte Vererbungsbeziehung untereinander besitzen.

#### 3.4. Import Coupling

Die Metrik Import Coupling orientiert sich ebenfalls, wie das Inheritance Coupling, an der Arbeit von Bidve und Sarasu, welche jegliche Referenzen und Aufrufe ausgehend von einer Klasse als Import Coupling zählen [„Tool for Measuring Coupling in Object-Oriented Java Software“].

Bei der Definition dieser Metrik müssen wir eine Unterscheidung für die Source- und Bytecodeanalyse vornehmen, da das Coupling hier nicht auf derselben Grundlage gezählt werden kann. In der Sourcecodeanalyse kann Import Coupling anhand der Importe im Sourcecode, welche mittels des Schlüsselwortes *import* implementiert sind, erkannt und gezählt werden. Somit lassen sich die Verbindungen der Klassen in Bezug auf diese Metrik leicht zählen. In der Bytecodeanalyse hingegen können Importe nicht mehr anhand des Schlüsselwortes *import* spezifiziert werden, da dieses im Bytecode nicht mehr existiert. Stattdessen wird das Coupling mittels Referenzen ohne Dopplungen gezählt, welche im Bytecode ablesbar sind.

Beim Import Coupling für die Sourcecodeanalyse ist zusätzlich zu beachten, dass nicht im Code verwendete, jedoch hinzugefügte Importe nichtsdestotrotz als Verbindung gezählt werden. Diese Metrik zählt hingegen kein Import Coupling ausgehend von einem Interface. Das bedeutet, sind in einem Interface Importe hinzugefügt, werden diese nicht als Coupling registriert.

#### 3.5. Field Coupling

Die Metrik Field Coupling orientiert sich nicht an Aufrufen, wie zum Beispiel die Metrik Method-to-Method Coupling, sondern zählt die Vorkommen von Feldern, ähnlich wie das Inheritance Coupling, welches anhand der Vorkommen von Schlüsselwörtern, wie beispielsweise *extends*, die Vererbungen zählt. Die Metrik Field Coupling registriert also eine Verbindung zu einer Klasse, sobald ein Feld vom Typ dieser Klasse in einer anderen Klasse definiert wird.

Auch in dieser Metrik beziehen wir Felder, die vom Typ einer inneren Klasse sind oder in einer inneren Klasse definiert wurden, auf ihre äußere, denn auch für diese Metrik werden äußere und ihre inneren Klassen als ein Modul betrachtet. Wird in einer Klasse ein Feld vom Typ eines Interfaces definiert, wird dies als Coupling zu diesem Interface gezählt. Wird hingegen ein Feld in einem Interface definiert, wird dies nicht als Coupling gezählt.

Diese Metrik dient dazu, das Coupling einer Software nicht nur anhand von Aufrufen zu betrachten. Das ist sinnvoll, da die Verwendung von Feldern, ähnlich wie Vererbungsbeziehungen, eine Erweiterung der Klasse durch Methoden ermöglicht, die durch das Feld zur Verfügung gestellt werden.

### 3.6. Eine weiterführende Metrik: Structural Debt Index

Die Metrik *Structural Debt Index* (SDI) entstammt dem statischen Code-Analysetool *Sonargraph* [hello2morrow] und berechnet die Anzahl der Abhängigkeiten, die aufgelöst werden müssten, um alle zyklischen Verbindungen zwischen Packages zu entfernen. Diese Metrik gibt somit einen Richtwert dafür an, wie aufwendig eine Bereinigung von zyklischen Abhängigkeit der Packages wäre. Da Sonargraph keine genauen Angaben dazu macht, wie sie ihre Metriken bzw. den Structural Debt Index genau berechnen, definieren wir den Structural Debt Index in Anlehnung an die Arbeit „Technical Debt Indexes Provided by Tools: A Preliminary Discussion“ wie folgt:

Der Structural Debt Index kann für jeden einzelnen Zyklus  $j$  bestimmt werden.

$$SDI_j = 10L + \sum_{i=1}^L dependencies(link_i) \quad (3.1)$$

Dabei gibt  $L$  die Größe des Zyklus an und  $dependencies(link_i)$  die Anzahl der Abhängigkeiten für die  $i$ -te Verbindung des Zyklus. Der Structural Debt Index ergibt sich somit aus der Summer dieser.

$$SDI = \sum_{j=1}^{Number\ of\ Cycles} SDI_j \quad (3.2)$$



# Besonderheiten der Source- und Bytecodeanalyse

Um Source- und Bytecode besser miteinander vergleichen zu können, müssen wir die Unterschiede zwischen diesen betrachten. Hierbei kann man Sourcecode als Abstraktion von Bytecode verstehen. Diese Abstraktion ist sinnvoll, da der Java-Compiler aus dem Sourcecode den Bytecode generiert. Dadurch kann es bei einem direkten Vergleich zwischen Sourcecode und dem dazu entsprechenden Kompilat zu Unterschieden kommen.

In diesem Kapitel betrachten wir einige Unterschiede von Source- und Bytecode, die während der Bearbeitung gefunden worden sind und sich auf Java 14 beziehen. Somit repräsentieren die hier vorgestellten Fälle eine sehr relevante Teilmenge, es existieren aber noch weitere. Zu den meisten hier vorgestellten Fällen sind JUnit Tests<sup>1</sup> vorhanden, die die richtige Funktionsweise als Teil des Analysetools sicherstellen.

## 4.1. Versteckte Funktionalität

In bestimmten Fällen verstecken sich hinter der Abstraktion von Sourcecode-Funktionen mehrere Funktionsaufrufe im entsprechenden Bytecode, die nicht direkt ersichtlich sind. Dies ist meist dann der Fall, wenn diese auf Java-internen Funktionen beruhen.

### 4.1.1. String-Operationen

Werden bei Java Strings miteinander konkateniert, versteckt sich dahinter ein größerer Aufruf, als in dem Sourcecode ersichtlich. Hierfür schauen wir uns einen Ausschnitt für Sourcecode an, in dem ein String konkateniert und dann auf der Standardconsole ausgegeben wird:

Listing 4.1. ConcatHelloWorld

```
1 public static void main(final String[] args) {  
2     String str = "Hello_";  
3     str += "World";
```

<sup>1</sup><https://git.informatik.uni-kiel.de/hs/bachelorprojekt-fischer-josuttis/-/tree/master/coupling-analysis/lib/src/test/java/examples>

#### 4. Besonderheiten der Source- und Bytecodeanalyse

```
4   System.out.println(str);  
5   }
```

Das entsprechende Gegenstück im Bytecode sieht folgendermaßen aus:

**Listing 4.2.** ConcatHelloWorld Bytecode

```
public static void main(java.lang.String[]);  
Code:  
0: ldc          #16          // String Hello  
2: astore_1  
3: new          #18          // class java/lang/StringBuilder  
6: dup  
7: aload_1  
8: invokestatic #20          // Method java/lang/String.valueOf:(Ljava/  
   lang/Object;)Ljava/lang/String;  
11: invokespecial #26          // Method java/lang/StringBuilder."<init  
   >":(Ljava/lang/String;)V  
14: ldc          #29          // String World  
16: invokevirtual #31          // Method java/lang/StringBuilder.append  
   :(Ljava/lang/String;)Ljava/lang/StringBuilder;  
19: invokevirtual #35          // Method java/lang/StringBuilder.  
   toString:()Ljava/lang/String;  
22: astore_1  
23: getstatic    #39          // Field java/lang/System.out:Ljava/io/  
   PrintStream;  
26: aload_1  
27: invokevirtual #45          // Method java/io/PrintStream.println:(  
   Ljava/lang/String;)V  
30: return
```

Auffällig ist zunächst, dass der Bytecode deutlich länger ist, welches zum einen an den Speicher- und Ladebefehlen liegt, zum anderen an den 'versteckten' Methoden, die Java-intern aufgerufen werden, wie bei Byte 3, 11, 16 und 19 dargestellt. Insbesondere bei 3 fällt auf, dass ein *java.lang.StringBuilder* initialisiert und auf diesem dann bei 16 *append* und anschließend *toString* bei 19 aufgerufen wird.

Ähnlich verhält es sich bei Kontrollstrukturen, wie *foreach-loops*. Hierbei wird intern der *java.util.Iterator* aufgerufen. Da wir uns nur für die Unterschiede von projektinternem Coupling interessieren, wird dies nicht als Unterschied zwischen Source- und Bytecode gezählt.



### 4.1.2. Try-Catch-Blöcke

Verwenden wir Try-Catch-Blöcke im Sourcecode, ist das eine Abstraktion von dem tatsächlichen Bytecode. Als Beispiel sehen wir uns diesen Codeabschnitt an.

**Listing 4.3.** Try-Catch-Block Sourcecode

```

1  try {
2      i = 0;
3  } catch (final IllegalStateException e) {
4      i = 1;
5  } finally {
6      System.out.println(i);
7  }
```

Das entsprechende Gegenstück im Bytecode sieht wie folgt aus:

**Listing 4.4.** Try-Catch-Block Bytecode

```

// try block
0: iconst_0
1: istore_1
2: goto      28
// catch block
5: astore_2
6: iconst_1
7: istore_1
// finally block after catch
8: getstatic  #16          // Field java/lang/System.out:Ljava/io/
    PrintStream;
11: iload_1
12: invokevirtual #22        // Method java/io/PrintStream.println:(I)V
15: goto      35
// finally block (exception not caught)
18: astore_3
19: getstatic  #16          // Field java/lang/System.out:Ljava/io/
    PrintStream;
22: iload_1
23: invokevirtual #22        // Method java/io/PrintStream.println:(I)V
26: aload_3
27: athrow
// finally block (no exception)
28: getstatic  #16          // Field java/lang/System.out:Ljava/io/
    PrintStream;
31: iload_1
```

#### 4. Besonderheiten der Source- und Bytecodeanalyse

```
32: invokevirtual #22                // Method java/io/PrintStream.println:(I)V
35: return
Exception table:
   from    to  target  type
    0       2     5     Class java/lang/IllegalStateException
    0       8    18     any
```

Im Sourcecode wird die Methode *println* nur einmal im finally-Block aufgerufen, im Bytecode hingegen sind drei Aufrufe zu finden und zwar bei Byte 12, 23 und 32. Dies liegt an dem Exceptionhandling von Java und wie finally-Blöcke behandelt werden. Im unteren Abschnitt von Listing 4.4 befindet sich eine Exception table. Aus dieser kann abgelesen werden, von welchem Byte zu welchem Byte, eine bestimmte Exception gefangen werden soll. In diesem Beispiel wird von 0 bis 2 eine *IllegalStateException* gefangen und von 0 bis 8 jegliche andere Exceptions. Unter *target* kann man ablesen, zu welcher Stelle im Bytecode gesprungen werden soll. Die Abschnitte sind hier, anders als im Sourcecode geparkt worden. Von 0-2 befindet sich der try-Block, von 5-7 der catch-Block und von 8-15 dessen finally-Block, der, nach dem Fangen der Exception, ausgeführt wird. Es wird weiter unterschieden: Wenn eine andere Exception, als die *IllegalStateException*, geworfen wurde, wird ein anderer finally-Block ausgeführt (von 18-27). Kommt es zu keiner Exception, wird von 28-35 wiederum ein anderer finally-Block ausgeführt. Somit entstehen drei Methodenaufrufe im Bytecode aus einem Methodenaufruf im Sourcecode. Ist dieser Aufruf eine projektinterne Verbindung, welche vom Analysetool gefunden wird, befinden sich im Bytecode auch dementsprechend mehr Verbindung bzw. mehr statisches Coupling.

Ähnlich ist auch das Konzept von Try-Catch-Blöcken mit Klassen, die das Interface *Autoclosable* implementieren.

##### 4.1.3. Lambda Funktionen

Der JavaParser kann Abhängigkeiten von Parametern aus Lambda-Funktionen bzw. anonymen Klassen, die in dem Funktionsblock verwendet werden, auflösen. Als Beispiel hierfür verwenden wir einen *java.util.Comparator*. Das Auflösen ist hier sowohl als Lambda Ausdruck, wie in Listing 4.5 gezeigt, als auch durch die direkte Implementierung der anonymen Klasse, wie in Listing 4.6, möglich.

Listing 4.5. Lambda Comperator

```
1 public static void main(final String[] args) {
2     final List<Integer> rndNumbers = List.of(7, 3, 1337, 42);
3     rndNumbers.sort( (x,y) -> x.compareTo(y));
4 }
```

Listing 4.6. Anonymous Comperator

```

1  public static void main(final String[] args) {
2      final List<Integer> rndNumbers = List.of(7, 3, 1337, 42);
3      rndNumbers.sort(new Comparator<Integer>() {
4          @Override
5              public int compare(final Integer x, final Integer y) {
6                  return x.compareTo(y);
7              }
8          });
9  }

```

Der JavaParser stößt dabei manchmal jedoch an seine Grenzen und kann in komplexen und nicht leicht reproduzierbaren Beispielen die Verbindungen nicht richtig auflösen. Dies ist immer dann der Fall, wenn Variablen, wie  $x$  und  $y$  aus den vorherigen Beispielen, nicht aufgelöst werden können und Aufrufe davon abhängen. Interessant ist im Gegensatz zum Sourcecode, dass die BCEL im Bytecode den Methodenaufruf, der in dem funktionalen Interface definiert worden ist, an dieser Stelle findet, ihn jedoch nicht richtig zuordnen kann.

#### 4.1.4. Importe

Im Sourcecode können Importe deklariert werden, die im Bytecode dann zu dem entsprechenden Import gelinkt werden. Bytecode hat somit generell keine Importdeklaration. Deshalb können auch nur die Abhängigkeiten zu anderen Klassen bestimmt werden. Zusätzlich kommen noch versteckte Importe hinzu, die im Sourcecode ebenfalls nicht explizit aufgeführt werden müssen. Es ist z.B. im Sourcecode nicht notwendig Klassen desselben Packages zu importieren. Aus diesen Gründen kann es zu unterschiedlichem Coupling im Source- und Bytecode kommen.

#### 4.1.5. Versteckte Klassen

Für manche Konstrukte im Sourcecode ist es für Java notwendig zusätzliche Klassen im Bytecode zu erstellen. Eine dieser versteckten Klassen sind innere Klassen, also solche, die innerhalb einer Klasse definiert wurden. Das heißt, wenn Klasse A eine innere Klasse B beinhaltet, wird eine Klassendatei für Klasse A und zusätzlich eine Klassendatei für Klasse B erstellt. Der Name dieser Klasse besteht aus Klasse A und Klasse B, die mit einem \$ separiert werden. Eine andere dieser versteckten Klassen entsteht durch anonyme Implementierung, wie in Abschnitt 4.1.3 beschrieben. Anders als bei den inneren Klassen, wird hier für jede anonyme Implementierung eine eigene Klasse erstellt, die den Namen der Klasse trägt, in der sie erstellt wurde, gefolgt von einem \$ und einer Ganzzahl. Da wir die Java-Klassen und deren Kompilate miteinander vergleichen wollen, werden die versteckten Klassen ihrer äußeren Klasse zugeordnet.

#### 4. Besonderheiten der Source- und Bytecodeanalyse

### 4.2. Abhängigkeiten von Superklassen

Ein interessanter Aspekt zwischen Source- und Bytecodeanalyse ist die Superklassenbeziehung, da Superklassenaufrufe klassischerweise erst zur Laufzeit ermittelt werden. Mit dem JavaParser ist man im Sourcecode in der Lage, die richtige Zuordnung von dem Methodenaufruf zu der richtigen Superklasse zu finden. Das heißt, wird eine Methode per *super.methodName()* oder *methodName()* aufgerufen, kann die entsprechende Superklasse, in der diese Methode implementiert ist, zugeordnet werden. Bei der Bytecodeanalyse mit BCEL wird hingegen nicht immer die richtige Klasse gefunden. Es gibt in Java die Möglichkeit, Methoden aus den Superklassen zu überschreiben. Damit würde man die Möglichkeit verlieren, diese Methoden aus den Superklassen zu verwenden. Deshalb gibt es das Schlüsselwort *super*, mit dessen Hilfe zusätzlich auch die Methoden aus der Superklasse verwendet werden können. Dabei handelt es sich aber um zwei unterschiedliche Arten von invoke-Instruktionen im Bytecode. Wird eine Methode überschrieben oder in der Klasse implementiert, wird ein *invokevirtual* ausgeführt. Der Aufruf mit *super* hingegen erzeugt ein *invokedynamic* zur ersten übergeordneten Superklasse, welcher zur Laufzeit aufgelöst werden muss. Somit findet die BCEL ein statisches Coupling zu dieser Klasse. Ist die aufgerufene Methode jedoch beispielsweise in der Superklasse der Superklasse implementiert, entsteht somit fälschlicherweise ein Coupling zur ersten übergeordneten Superklasse. Der Aufruf ohne das Schlüsselwort *super* erzeugt im Bytecode ein Coupling zur eigenen Klasse, welches wir nicht zählen, wie in Abschnitt 5.3 beschrieben wird.

### 4.3. Fehler bei Namensauflösungen

#### 4.3.1. Methodenketten

Wenn Verbindungen, wie z.B. in Abschnitt 4.1.3 beschrieben, nicht aufgelöst werden können, besteht die Möglichkeit von kaskadierenden Fehlern. Das bedeutet, sobald ein Teil einer Methodenkette nicht aufgelöst werden kann, sind alle Folgeglieder, die in Abhängigkeit zu diesem stehen, ebenfalls nicht auflösbar. Somit entsteht durch einen einzigen Auflösungsfehler eine ganze Reihe von Fehlern.

Listing 4.7. Method chain

```
1 funcA(funcB(funcC(var).funcD()), funcE());
```

In Listing 4.7 ist eine solche Methodenkette abgebildet. Die Methode *funcA* ist von den Methoden *funcB* und *funcE* abhängig, wobei *funcB* wiederum eine Abhängigkeit zu *funcC* bzw. *funcD* aufweist. Die Methode *funcA* kann damit nur dann richtig aufgelöst werden, wenn ihre Subkomponenten ebenfalls aufgelöst werden können.

### 4.3.2. Exceptions

Der JavaParser bzw. der SymbolSolver kann in manchen Fällen Namen nicht auflösen. Dies kann verschiedenen Ursachen zu Grunde liegen. Wird ein Name versucht aufzulösen, und dies schlägt fehl, wirft der JavaParser im Normalfall eine `UnsolvedSymbolException` oder `UnsupportedOperationException`. Letztere wird von dem JavaParser geworfen, wenn zum Auflösen des Namens interne Operationen des JavaParsers notwendig sind, die noch nicht implementiert wurden. Zum Beispiel ist es nicht möglich, Verbindungen, die im Zusammenhang mit Annotationen stehen, aufzulösen. Allerdings kommt es in einigen Fällen zu undokumentierten Fehlern. Einige dieser Fehler können eine `RuntimeException` oder einen `Stackoverflow` verursachen. Wenn Namen nicht aufgelöst werden können, in dessen Zusammenhang jedoch noch weitere Auflösungen stehen, wie in Abschnitt 4.3.1 beschrieben, kann bei dem davon abhängigen Methodenaufruf eine `RuntimeException` folgen. In Spezialfällen kann es auch zu einem `Stackoverflow` kommen. Alle Exceptions, die während des Durchlaufs des Analysetools gefunden werden, werden wie in Abschnitt 5.5 dargelegt, vermerkt.

## 4.4. Generische Typen

Generische Typen bzw. Klassen, die generische Typen verwenden, eignen sich sehr gut, um Code zu verallgemeinern. Will man hierbei statisches Coupling messen, würde in manchen Fällen Coupling zu abstrakten Typen gefunden werden und nicht zu den tatsächlichen Klassen. Dies liegt daran, dass sich die Aufrufe erst zur Laufzeit, also auf den Objekten selbst, feststellen lassen. Bei dieser Art des Couplings zur Laufzeit handelt es sich dann nicht mehr um statisches, sondern um dynamisches Coupling, welches wir hier allerdings nicht betrachten wollen. Das bedeutet, dass generische Abhängigkeiten nicht immer von dem Analysetool gefunden werden können. Ein einfaches Beispiel dafür ist der generische Typ eines Feldes, da dieser erst zur Laufzeit festgelegt wird. Einfache Methodenaufrufe, die einen generischen Typen als Parameter erwarten, können hingegen aufgelöst werden, da der Klassen- und Methodenname hierfür bekannt ist.



# Funktionsweise des Analysetools

In diesem Kapitel wollen wir die allgemeine Funktionsweise und die Implementierung der Kernkomponenten sowie Teile der Struktur des Analysetools betrachten. Im Allgemeinen wird in einem Durchlauf sowohl eine Bytecodeanalyse als auch im Anschluss eine Sourcecodeanalyse durchgeführt. Dies kann aber je nach Anwendungsfall angepasst werden. Den Verarbeitungsprozess kann man dabei in vier Phasen einteilen. In der ersten Phase werden die Daten für die Analyse erfasst. Für den Bytecode werden dafür alle Klassen-Dateien und für den Sourcecode die Javafiles in einem vorher spezifizierten Verzeichnis gesammelt. Die zweite Phase befasst sich mit dem Parsen von Source- und Bytecode. Zusätzlich werden für das Parsen des Sourcecodes in dem Projekt verwendete externe Bibliotheken benötigt, welches näher in Abschnitt 5.1 beschrieben wird. In der dritten Phase findet die tatsächliche Verarbeitung bzw. die Berechnung der Metriken statt. Diese werden hierbei nacheinander berechnet. In der vierten Phase werden die entstandenen Ergebnisse in Dateien im CSV-Format persistiert. Dieses Analysetool ist im GitLab<sup>1</sup> zugänglich.

## 5.1. Anwendung des JavaParsers und SymbolSolvers

Wie in Abschnitt 2.3 beschrieben, verwenden wir den JavaParser für das Bestimmen des Couplings im Sourcecode. Um mit dem Sourcecode arbeiten zu können, müssen die Java-Dateien (Sourcecode) in *com.github.javaparser.ast.CompilationUnits* umgewandelt werden. Dafür werden diese aus einem vordefinierten Hauptdateiverzeichnis gesammelt und in *CompilationUnits* umgewandelt, wie die Zeilen 3-5 in Listing 5.1 darstellen. Anschließend wird der SymbolSolver konfiguriert. Dieser ist notwendig, um später die Namen im Sourcecode aufzulösen (resolve). Dafür werden *TypeSolver* verwendet, diese werden in drei verschiedene Kategorien aufgeteilt: *JavaParserTypeSolver*, *ReflectionTypeSolver* und *JarTypeSolver*.

Listing 5.1. Parse to CompilationUnits

```
1 public static List<Optional<CompilationUnit>> getAllCompilationUnits() {  
2     /* ... */  
3     final String[] patterns = { "**/*.java" };
```

<sup>1</sup><https://git.informatik.uni-kiel.de/hs/bachelorprojekt-fischer-josuttis/-/tree/master/coupling-analysis>  
(Commit-Id: 622dfe668d302b03d46d22dafd3c0a8356c0219c)

## 5. Funktionsweise des Analysetools

```
4      final List<String> sourceList = Arrays
5          .asList(DirectoryTool.filesScannedInDirectory(DirectoryTool.
6              getSourceCodeInputRoot(), patterns));
7
8      /* ... */
9      setupSymbolSolver();
10     /* ... */
11     // Parse all files.
12     final List<Optional<CompilationUnit>> cus = sourceList
13         .stream()
14         .map(FunctionHelper
15             .handleExceptionFunctionWithWrapper(source -> StaticJavaParser.
16                 parse(new File(source))))
17         .collect(Collectors.toList());
18     return cus;
19 }
```

Der *JavaParserTypeSolver* ist für das Auflösen von Verbindungen zwischen internen Projektverbindungen des Sourcecodes zuständig. Der *ReflectionTypeSolver* wird benötigt, um Verbindungen zu Java build-in-functions aufzulösen. Die *JarTypeSolver* sind dafür zuständig auch Verbindungen zu externen Libraries aufzulösen. Auch wenn das Analysetool nur projektinterne Verbindungen betrachten soll, ist der *JavaParserTypeSolver* für diese Zwecke nicht ausreichend. Das liegt daran, dass die Verbindungen sowohl von build-in-functions, als auch den externen Libraries abhängen können. Somit ist es unerlässlich, sie mit einzubinden. Ein vereinfachtes Beispiel für die Notwendigkeit des *ReflectionTypeSolver* ist der Aufruf *method(Integer.parseInt("1"))* und für den *JarTypeSolver* der Aufruf *method(biMap.get("one"))*, wobei das Objekt *biMap* in diesem Fall von der Library Guava stammt. In beiden Fällen hängt der Methodenaufruf von dem Argument ab, welches zum einen eine build-in-function und zum anderen einen Aufruf von einem Objekt aus einer Library darstellt. Dieses Problem wird in Abschnitt 4.3.1 näher betrachtet. Der *JavaParser* stellt den *CombinedTypeSolver* zur Verfügung, mit Hilfe dessen sich mehrere Solver kombinieren lassen. Die Anwendung der TypeSolver ist in Listing 5.2 zu finden.

Damit können die Java-Dateien zu den *CompilationUnits* geparkt werden (Listing 5.1 Zeile 10-14).

**Listing 5.2.** Setup Symbol Solver

```
1  private static void setupSymbolSolver() {
2
3      final List<TypeSolver> parseTypeSolvers = new ArrayList<>();
4      parseTypeSolvers.add(new JavaParserTypeSolver(DirectoryTool.
5          getSourceCodeInputRoot()));
6
7      // Add the ReflectionTypeSolver for resolve internal connections of java
```



## 5.1. Anwendung des JavaParsers und SymbolSolvers

```
7 // in-build-functions.
8 parseTypeSolvers.add(new ReflectionTypeSolver());
9
10 // Add external Library files.
11 /* ... */
12 libSources
13     .stream()
14     // Create and collect JarType Solvers form given libraries and map
15     // to the corresponding library name
16     .map(FunctionHelper.handleExceptionFunction(lib -> {
17         parseTypeSolvers.add(new JarTypeSolver(lib));
18         return lib.substring(lib.lastIndexOf("/") + 1, lib.length());
19     }))
20     .collect(Collectors.toList());
21 /* ... */
22 });
23 // Setup java symbol solver
24 final TypeSolver typeSolver = new CombinedTypeSolver(parseTypeSolvers);
25 final JavaSymbolSolver symSolv = new JavaSymbolSolver(typeSolver);
26 // Sets the Symbol Resolver globally
27 StaticJavaParser.getConfiguration().setSymbolResolver(symSolv);
28 }
```

Der JavaParser implementiert das Visitor Design Pattern. Das bedeutet, es kann ein eigener Visitor implementiert werden, der bei dem Besuchen des AST (siehe Abschnitt 2.4) bestimmte Funktionalitäten zur Verfügung stellt, wenn ein bestimmter Knoten des AST besucht wird. Der JavaParser stellt zusätzlich auch eine andere Möglichkeit zur Verfügung, die ebenfalls in unserem Analysetool zur Verwendung kommt. Damit können wir in den CompilationUnits nach allen oder nach dem ersten Auftreten eines Knotens suchen. In diesem Beispiel wird mit nur einer Zeile Code nach allen Klassen- und Interface-Deklarationen gesucht.

**Listing 5.3.** Find in CompilationUnit

```
1 final List<ClassOrInterfaceDeclaration> coi = cu.findAll(
    ClassOrInterfaceDeclaration.class);
```

Da wir nun in der Lage sind, Knoten in dem AST zu besuchen bzw. zu finden, wollen wir diese Funktionalität erweitern, um Coupling zu identifizieren. Um statisches Coupling für Method-to-Method Coupling zu identifizieren, muss nach Methodenaufrufen gesucht werden. Das entsprechende Gegenstück im JavaParser stellt die Klasse *MethodCallExpr* dar. In Abschnitt 3.1 wurde das Method-to-Method Coupling definiert. Um die Klasse (Callee), auf dem die Methode aufgerufen worden ist, zu finden, ist es zuerst notwendig,

## 5. Funktionsweise des Analysetools

zu ermitteln, zu welcher entsprechenden Methodendeklaration die Methode gehört. Wie in Listing 5.4 abgebildet, kann mit *resolve* auf der *MethodCallExpr* die Zuordnung gefunden werden. Diese, bzw. das Bestimmen der Zugehörigkeit zu der richtigen Methodendeklaration, übernimmt der *SymbolSolver*, den wir bereits aufgesetzt haben. Anschließend wird der *qualified name* bestimmt. Der qualifizierte Name der Methode besteht aus dem, des deklarierenden Typs, gefolgt von einem Punkt und dem Namen der Methode. Also enthält der qualifizierte Name die Verbindung zu der aufgerufenen Klasse. Ein Vorteil, die Beziehung über den *qualified name* zu bestimmen, ist, dass gleichnamige Klassen über den Paketnamen als zusätzlichen Indikator differenziert werden können.

Listing 5.4. Resolve name

```
1 public void getMethodFullName(final MethodCallExpr mce, final String className,
2     final String methodName) {
3     try {
4         // At this point the SymbolSolver is called.
5         final ResolvedMethodDeclaration rmd = mce.resolve();
6         String name = rmd.getQualifiedName();
7         // ...successfully resolved...
```

## 5.2. Anwendung der BCEL

Ähnlich wie bei dem *JavaParser* in Abschnitt 5.1 parsen wir den Bytecode zu einem *Java-Objekt*, mit dem wir interagieren können. Die BCEL (siehe Abschnitt 2.2) stellt dafür den *org.apache.bcel.classfile.ClassParser* zur Verfügung. Wir erhalten nach dem Parsen eine *org.apache.bcel.classfile.JavaClass*. Wichtig ist, dass jede *JavaClass* zu dem *org.apache.bcel.Repository* hinzugefügt werden muss, sonst kann es dazu führen, dass die Verbindungen zwischen den verschiedenen Klassen nicht aufgelöst werden können. Dieser Ablauf ist in Listing 5.5 zu finden.

Listing 5.5. Parsing java classes

```
1 private void collectJavaClass(final String pathToClassFile) {
2
3     final ClassParser parser = new ClassParser(pathToClassFile);
4     /* ... */
5     final JavaClass javaClass = parser.parse();
6     // Caching java classes
7     // this is necessary to resolve dependencies between classes later.
8     Repository.addClass(javaClass);
9     addClass(javaClass);
10    /* ... */
```

```
11 | }
```

Mit dem `JavaClass` Objekt kann nun gearbeitet werden. Zum Beispiel können für das Method-to-Method Coupling (siehe Abschnitt 3.1) alle Methoden der `JavaClass` abgerufen werden. In diesen Methoden suchen wir dann mit dem `MethodVisitor` und einer gegebenen Strategie nach Methodenaufrufen.

Listing 5.6. Search in all Methods

```
1 public void calculateCoupling(final JavaClass javaClass) {
2
3     final ConstantPoolGen constantPoolGen = new ConstantPoolGen(javaClass.
4         getConstantPool());
5     final Method[] methodDeclarations = javaClass.getMethods();
6
7     for (final Method md : methodDeclarations) {
8         final MethodVisitor mv = new MethodVisitor(md, javaClass.getClassName(),
9             constantPoolGen, strategy);
10        mv.visitMethod();
11    }
```

Das ist möglich, da die BCEL das Visitor Design Pattern implementiert. Um z.B. die einzelnen Methodenteile im Bytecode besuchen zu können, erweitert der `MethodVisitor` den `org.apache.bcel.generic.EmptyVisitor`. Damit können z.B. die verschiedenen `invoke`-Befehle des Bytecodes wie `visitINVOKESTATIC`, `visitINVOKESPECIAL`, `visitINVOKEVIRTUAL` und `visitINVOKEDYNAMIC` besucht werden. In unserem Code rufen diese Methoden wiederum `visitINVOKEMethodCall` auf. Mit Hilfe des Konstantenpools kann dann der Klassenname, sowie der Methodenname zu der `invokeInstruction` gefunden werden. Anschließend wird überprüft, ob diese Methode ignoriert werden soll, wie in Abschnitt 5.3 beschrieben. Ist das nicht der Fall, wird dieses Coupling hinzugefügt.

Listing 5.7. Visiting invoke method calls

```
1 public void visitINVOKEMethodCall(final InvokeInstruction invokeInstruction) {
2
3     final String classTo = invokeInstruction.getClassName(constantPoolGen);
4     final String methodName = invokeInstruction.getMethodName(constantPoolGen);
5
6     // Make sure that we do not count internal parts, e.g. control structures.
7     if (!MethodToMethodCoupling.isIgnored(classTo, methodName)) {
8         registerCoupling(classTo, methodName);
9     }
```

### 5.3. Filter für projektinterne Funktionen

Da wir nur an projekt-internen Daten interessiert sind, gibt es zwei Möglichkeit die ungewollten Abhängigkeiten herauszufiltern. Zum einen gibt es die Möglichkeit, eine konkrete Liste mit Klassen zu erstellen, die nicht Teil des Projektes sind, also eine Blacklist. Das hat den Vorteil, dass man sehr feingranular entscheiden kann, was nicht gezählt werden soll. Zum anderen gibt es die Möglichkeit, eine Whitelist zu definieren, die genau die Verbindungen durchlässt, die darin definiert sind. In unserem Projekt verwenden wir die zweite Variante, da wir direkt bei dem Parsen der Klassen bzw. Java-Dateien die Namen auf die Liste setzen können und somit automatisch diejenigen Klassen hinzufügen, die Teil des Projektes sind. Auch das Testen ist über eine Whitelist einfacher zu gestalten, weil man sich auf bestimmte Teile beschränken kann. Im Endresultat des Analysetools befindet sich, wie in Abschnitt 5.5 beschrieben, eine Statistik, über die Anzahl der Verbindungen, die jeweils für Source- und Bytecode gefunden worden sind. Damit eine möglichst gute Vergleichbarkeit erreicht wird, sollten diese möglichst dicht zusammen oder am besten identisch sein. Da es aber Unterschiede zwischen Source- und Bytecode gibt, wie unter anderem in Abschnitt 4.1 beschrieben, könnten diese das Ergebnis verzerren. Deshalb gibt es neben der Whitelist noch eine Ignorelist. Mit Hilfe dieser Liste werden z.B. Iterator-Aufrufe oder interne Casts ausgeschlossen. Des Weiteren werden alle selbstbezogenen Couplings (Self-Connections) herausgefiltert, weil es sich bei diesen Verbindungen nicht mehr um Coupling, sondern um Kohäsion handelt.

### 5.4. Implementierung der Metriken

Die Berechnung der Metriken kann auf verschiedene Arten realisiert werden. Man könnte z.B. eine einzige Klasse erstellen, die für die Berechnung jeder Metrik eine entsprechende Funktion zur Verfügung stellt. Da wir aber, je nach Bedarf die Möglichkeiten haben wollen die Metriken an- und abzuwählen, haben wir uns dazu entschieden, ähnlich des Strategy Design Patterns, dass jede Berechnung der Metrik in einer eigenen Klasse realisiert wird. Alle Klassen implementieren das gemeinsame Interface *ICoupling* und erben von der abstrakten Klasse *ACoupling*, die Funktionen zur Verfügung stellt, die alle Metriken benötigen. Jede einzelne Metrik speichert dabei ihre gesammelten Informationen selber. Der *CouplingMonitor* ist dafür zuständig, diese Metriken zu verwalten und aufzurufen. Eine weitere Designentscheidung, die getroffen wurde, ist, dass das Interface *ICoupling* sowohl eine Funktion für die Berechnung des Couplings für Bytecode als auch eine für Sourcecode enthält. Das bedeutet, dass jede Metrik beide Funktionalitäten zur Verfügung stellt. Diese Designentscheidung wurde getroffen, weil die Berechnung des Couplings sehr ähnlich gestaltet ist und damit unnötige Parallelstrukturen verhindert werden können. Es besteht dennoch weiterhin die Möglichkeit, dort ggf. die Funktionalitäten für Weiterentwicklungen wieder zu trennen. Diese Struktur ist in Abbildung 5.1 veranschaulicht.

## 5.5. Sammlung von Metadaten

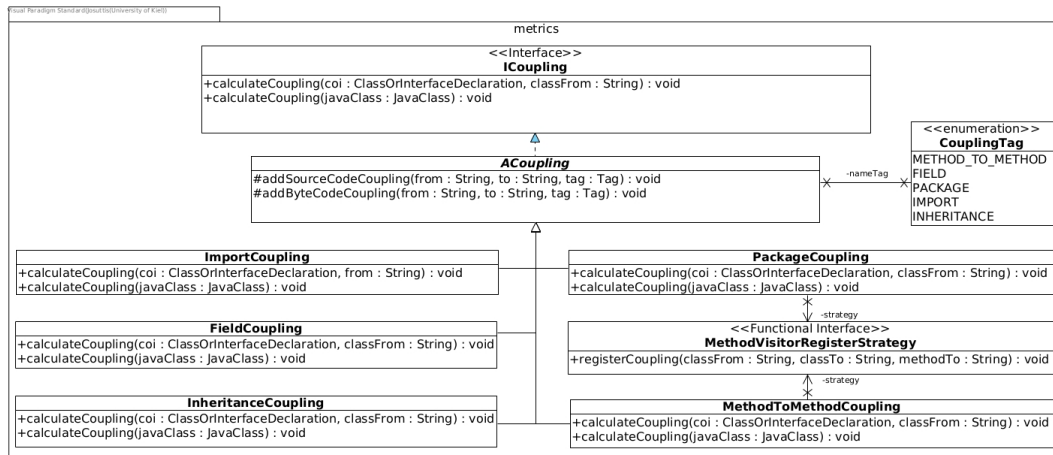


Abbildung 5.1. UML-Diagram metrics

## 5.5. Sammlung von Metadaten

Das Analysetool sammelt Metadaten zu jeder Metrik. Diese bestehen aus einer Anzahl von Verbindungen, die aufgelöst werden konnten, der Menge der projektinternen und selbstbezogenen Verbindungen, sowie der Fehlermenge. Letztere werden so vermerkt, dass nicht nur das Aufkommen eines Fehlers gezählt wird, sondern auch bei welcher Metrik dieser entstanden ist. Zusätzlich wird die dazugehörige Zeile und Spalte der Klasse angegeben, in welcher der Fehler aufgetreten ist. Neben diesen Metadaten wird der gesamte Verlauf des Analysetools in einer Datei persistiert.



# Evaluierung

In diesem Kapitel wollen wir uns verschiedene Projekte mit unserem Analysetool anschauen. Die hier betrachteten Projekte sind Gradle oder Maven Projekte. Da sowohl Gradle als auch Maven die kompilierten Dateien in einer Ordnerstruktur ablegen und externe Libraries in den jeweiligen Repositories gespeichert werden, können somit die benötigten Daten für unser Analysetool einfach abgefragt werden.

In Abschnitt 6.2, Abschnitt 6.3 und Abschnitt 6.4 betrachten wir jeweils ein Projekt und analysieren interne und projektübergreifende Aspekte. Anschließend werden in Abschnitt 6.5 allgemeine Auffälligkeiten erörtert und in Abschnitt 6.6 die Source- und Bytecode-Ergebnisse gegenübergestellt.

Die Ergebnisse der mittels des Analysetools analysierten Projekte sind im Git-Repository<sup>1</sup> zu finden.

## 6.1. Ziele und Erwartungen

Einer der Gründe, weshalb wir unterschiedliche Projekte analysieren, ist es, praktische Anwendungsbeispiele darzustellen. Des Weiteren erhalten wir so eine Rückmeldung, ob die Ergebnisse, die unser Analysetool erzeugt, sinnvoll sind.

Für jedes der Projekte werden in den folgenden Abschnitten die Analyseergebnisse dargestellt. Dabei ist ein weiteres Ziel herauszufinden, ob die verschiedenen Metriken untereinander Indikatoren für Coupling darstellen. Auch wollen wir den Zusammenhang von Source- und Bytecode betrachten. Da ein direkter Vergleich nicht immer möglich ist, wie unter anderem in Kapitel 4 beschrieben, ist ein Ziel der Analyse, deren Unterschiede in Bezug auf die Metriken aufzuzeigen.

Wegen der unterschiedlichen Grundlagen für die Zählweise des Import und Inheritance Couplings im Source- und Bytecode erwarten wir in diesen Bereichen eine Abweichung der Ergebnisse. Jedoch vermuten wir eine Korrelation aller Metriken von Source- und Bytecode, welche wir mit Hilfe der Spearman Korrelation berechnen.

Wie in Abschnitt 3.2 bereits beschrieben, sollten Packages logische Einheiten bilden. In Bezug auf die Analyse erwarten wir daher ein höheres Coupling des Method-to-Method Couplings innerhalb eines Packages als paketübergreifend.

---

<sup>1</sup><https://git.informatik.uni-kiel.de/hs/bachelorprojekt-fischer-josuttis/-/tree/master/results>

## 6. Evaluierung

### 6.2. Analyse: Mockito

Mockito [Mockito] ist ein bekanntes Mocking-Framework für JUnit Tests.

#### Konfiguration

Betriebssystem: Windows

Java Version: 1.8.0\_171

Hauptdateiverzeichnis des Sourcecodes: \*/mockito/src/main/java/

Hauptdateiverzeichnis des Bytecodes: \*/mockito/build/classes/java/main/

#### Ergebnisse

Bei der Analyse von Mockito wurden 419 Klassen und 66 Packages in der Bytecodeanalyse und 418 Klassen und 66 Packages in der Sourcecodeanalyse gefunden. Die Klasse *org.mockito.runners.package-info.java* wird von der Sourcecodeanalyse nicht gefunden, sie enthält jedoch auch keine Klassendeklaration, sondern lediglich eine Package-Information und erzeugt somit kein Coupling.

Tabelle 6.1 zeigt fünf Klassen, sortiert nach dem höchsten Method-to-Method Coupling.

**Tabelle 6.1.** Top 5 Method-to-Method Coupling Sourcecode (Bytecode)

Name	Method	Import	Field	Inheritance
1. org.mockito.internal.exceptions.Reporter	119 (119)	21 (38)	0 (0)	0 (0)
2. org.mockito.AdditionalMatchers	84 (84)	8 (11)	0 (0)	0 (0)
3. org.mockito.internal.MockitoCore	82 (82)	30 (35)	0 (0)	0 (0)
4. org.mockito.internal.creation.bytebuddy. InlineDelegateByteBuddyMockMaker	65 (67)	15 (58)	7 (7)	3 (7)
5. org.mockito.Mockito	61 (61)	26 (15)	7 (7)	1 (1)

In der Tabelle 6.2 werden die Korrelationen zwischen Metriken der Bytecodeanalyse dargestellt. Auffällig dabei ist, dass das Method-to-Method Coupling eine geringere Korrelation zum Field und Inheritance Coupling hat, als zum Import Coupling. Dies passt ebenfalls zu den Ergebnissen aus Tabelle 6.1. Die Klassen, die ein hohes Method-to-Method Coupling erzeugen, haben ebenfalls häufig ein hohes Import Coupling, jedoch wenig bis kein Coupling in Bezug auf Field und Inheritance.

Diese Auffälligkeit lässt sich ebenfalls aus den Ergebnissen der Sourcecodeanalyse erkennen. Wie in Tabelle 6.3 ablesbar, ist auch hier die Korrelation zwischen Method-to-Method und Field Coupling und zwischen Method-to-Method und Inheritance Coupling am geringsten, auch wenn nicht ganz so deutlich wie in der Bytecodeanalyse. Das passt ebenfalls zu der Anzahl der entsprechenden Couplings aus Tabelle 6.1.



Eine Begründung für diese Auffälligkeit liegt darin, dass allein durch die Verwendung von wenigen Feldern und Vererbungsbeziehungen der Zugriff auf deutlich mehr Methoden ermöglicht wird.

**Tabelle 6.2.** Korrelationen der Metriken der Bytecodeanalyse

	Import	Field	Inheritance
Method	0.819	0.429	0.347
Import		0.588	0.592
Field			0.578

**Tabelle 6.3.** Korrelationen der Metriken der Sourcecodeanalyse

	Import	Field	Inheritance
Method	0.782	0.363	0.204
Import		0.504	0.366
Field			0.373

Betrachten wir aus der Bytecodeanalyse das Method-to-Method Coupling der Klassen aus demselben Package im Vergleich zu dem Package Coupling. Beispiele hierfür sind das Package *org.mockito.internal.creation.bytebuddy* mit einem Coupling von 165 und die Klassen darin mit einem aufsummierten Method-to-Method Coupling von 255, das Package *org.mockito.internal.invocation* mit einem Coupling von 60 und einem aufsummierten Method-to-Method Coupling der Klassen innerhalb dessen von 79. Als drittes Beispiel betrachten wir das Package *org.mockito.internal.verfication.checkers* mit einem Coupling von 44 und einem Gesamt Method-to-Method Coupling der Klassen von ebenfalls 44. Auffällig an diesen Beispielen ist, dass das Method-to-Method Coupling der Klassen aus demselben Package aufsummiert zwar etwas höher liegt, als das Coupling ihres Packages, jedoch nicht besonders weit auseinanderliegt. In einem Fall ist es sogar gleich. Daraus lässt sich schließen, dass die Aufteilung der Klassen in Packages nicht unbedingt auf Grundlage der Struktur und des Couplings, sondern eher nach Sinn getroffen wurde.

Die Berechnung des Structural Debt Index nach der Definition aus Abschnitt 3.6 der Bytecodeanalyse ergab einen Wert von 5416. Veranschlagt man nun mindestens 6 Minuten Arbeit, wie von Sonargraph angegeben, für jede Verbindung, ergeben sich daraus 541.6 Stunden Arbeit, um alle Verbindungen der Package-Zyklen aufzulösen.

## 6.3. Analyse: Guava

Guava [*Guava*] ist eine von Google entwickelte Sammlung von Java-Libraries für neue Collectiontypes, Graph-Libraries, Hashing und vieles mehr. Es stellt somit eine Erweiterung der Standard Java-Libraries dar.

## 6. Evaluierung

### Konfiguration

Betriebssystem: Windows

Java Version: 1.8.0\_171

Hauptdateiverzeichnis des Sourcecodes: \*/guava/guava/src/

Hauptdateiverzeichnis des Bytecodes: \*/guava/guava/target/classes/

### Ergebnisse

Bei der Sourcecodeanalyse von Guava wurden 557 Klassen und 17 Packages, bei der Bytecodeanalyse 570 Klassen und 17 Packages gefunden. Dabei wurde 13-mal die Klasse *package-info* im Bytecode in verschiedenen Packages gezählt, welche keine Klassendeklaration, sondern lediglich eine Package-Information enthält und somit auch kein Coupling erzeugt. Die Sourcecodeanalyse fand hingegen die Klassen *com.google.common.hash.LockFreeBitArray* und *com.google.common.base.StringConverter*, welche sich jedoch nicht aus der Bytecodeanalyse ergaben. Diese beiden Klassen erzeugen allerdings ein geringes Coupling.

Die Tabelle 6.4 zeigt die fünf Klassen mit dem höchsten Method-to-Method Coupling. Auffällig hierbei ist der enorme Unterschied im Import Coupling. Die Messergebnisse fallen deutlich höher zugunsten der Bytecodeanalyse aus. Dieser Unterschied war beispielsweise bei Mockito nicht so stark.

**Tabelle 6.4.** Top 5 Method-to-Method Coupling Sourcecode (Bytecode)

Name	Method	Import	Field	Inheritance
1. com.google.common.cache.LocalCache	346 (341)	16 (82)	41 (46)	7 (10)
2. com.google.common.collect.TreeRangeSet	199 (197)	1 (60)	6 (14)	5 (11)
3. com.google.common.collect.Maps	187 (188)	8 (101)	10 (13)	12 (30)
4. com.google.common.collect.Multisets	186 (180)	5 (77)	5 (13)	2 (12)
5B. com.google.common.collect. ImmutableRangeSet	61 (163)	26 (53)	7 (6)	1 (6)
5S. com.google.common.collect.Multimaps	156 (140)	4 (107)	10 (10)	13 (21)

Tabelle 6.5 und Tabelle 6.6 zeigen die Korrelationen zwischen den Metriken, sowohl im Bytecode als auch in Sourcecode. Dabei fällt auf, dass die Korrelationen zwischen dem Method-to-Method Coupling und Field und Inheritance Coupling deutlich höher sind als bei Mockito. Ebenfalls signifikant ist der Unterschied der Korrelationen zwischen Import und Inheritance Coupling im Source- und Bytecode. Klassen, die viele Importe besitzen, haben wenig Coupling in Bezug auf Vererbung.

Betrachten wir das Package Coupling der Bytecodeanalyse und vergleichen es mit dem Method-to-Method Coupling von Klassen, die in denselben Packages liegen. Dafür sehen wir uns folgende Beispiele an: Das Package *com.google.common.collect* hat das höchste Coupling mit einem Wert von 1022. Die Klassen, die darin liegen, erzeugen aufsummiert ein

**Tabelle 6.5.** Korrelationen der Metriken der Bytecodeanalyse

	Import	Field	Inheritance
Method	0.866	0.778	0.681
Import		0.692	0.863
Field			0.538

**Tabelle 6.6.** Korrelationen der Metriken der Sourcecodeanalyse

	Import	Field	Inheritance
Method	0.589	0.715	0.553
Import		0.528	0.223
Field			0.426

Method-to-Method Coupling von 4415. Das Package *com.google.common.util.concurrent* hat einen Wert von 335, seine Klassen erzeugen zusammen ein Method-to-Method Coupling von 691. Ein weiteres Package *com.google.common.cache* hat ein Package Coupling von 283 und die Klassen, die darin liegen, ein Method-to-Method Coupling von insgesamt 575. Auffällig daran ist, dass das Coupling der Packages deutlich geringer ausfällt als das Method-to-Method Coupling der Klassen, die innerhalb dieser liegen. Die Tendenz konnte beispielsweise bei der Analyse von Mockito nicht festgestellt werden, denn dort gab es nur geringe bis keine Unterschiede der beiden Metriken.

Das Ergebnis lässt darauf schließen, dass bei der Strukturierung des Programms darauf geachtet wurde, dass Klassen, die sich häufiger aufrufen, in denselben Packages untergebracht wurden. Ein weiterer Hinweis darauf, dass bei der Programmierung unnötiges Coupling vermieden wurde, ist der Structural Debt Index, welcher nach der Definition aus Abschnitt 3.6 einen Wert von 0 ergibt. Da wir für den Index nur projektinterne Package-Zyklen betrachten, ist jedoch nicht auszuschließen, dass Guava einen höheren SDI besitzt, sobald auch projektexterne Verbindungen mit einbezogen werden.

## 6.4. Analyse: Netflix Hollow

Netflix ist ein bekannter Streamingdienst. Die Netflix Hollow [Netflix Hollow] ist eine Java Library, durch die In-Memory-Datensätze von einem Producer an mehrere Consumer weitergegeben werden können, um somit einen leistungsstarken, schreibgeschützten Zugriff zu ermöglichen.

### Konfiguration

Betriebssystem: Linux

Java Version: java-8-openjdk-amd64

## 6. Evaluierung

Hauptdateiverzeichnis des Sourcecodes: \*/hollow/hollow/src/main/java/

Hauptdateiverzeichnis des Bytecodes: \*/hollow/hollow/build/classes/java/main/

### Ergebnisse

Während der Sourcecodeanalyse wurden 464 Klassen und 74 Packages gefunden. Die Bytecodeanalyse ergab 469 Klassen und ebenfalls 74 Packages. Die fünf Klassen, die während der Bytecodeanalyse zusätzlich gezählt wurden, sind:

*com.netflix.hollow.api.sampling.TimeSliceSamplingDirector,*

*com.netflix.hollow.core.read.filter.ResolvedTypeFilter,*

*com.netflix.hollow.core.read.filter.Resolver,*

*com.netflix.hollow.core.read.filter.TypeActions* und

*com.netflix.hollow.core.read.filter.UnresolvedTypeFilter.*

Diese Klassen erzeugen Coupling in allen Metriken, außer im Field Coupling.

Tabelle 6.7 zeigt fünf Klassen, sortiert nach dem höchsten Method-to-Method Coupling. Auffällig an diesen Klassen ist, dass alle fünf wenig bis kein Inheritance Coupling aufweisen. Dies spiegelt sich auch in den Korrelationswerten aus Tabelle 6.8 und Tabelle 6.9 wieder.

**Tabelle 6.7.** Top 5 Method-to-Method Coupling Sourcecode (Bytecode)

Name	Method	Import	Field	Inheritance
1. com.netflix.hollow.api.producer. AbstractHollowProducer	184 (216)	26 (61)	15 (15)	0 (1)
2. com.netflix.hollow.tools.patch.delta. HollowStateDeltaPatcher	150 (151)	27 (48)	3 (20)	0 (0)
3. com.netflix.hollow.tools.filter. FilteredHollowBlobWriter	130 (130)	20 (24)	4 (4)	0 (0)
4. com.netflix.hollow.tools.history.keyindex. HollowHistoryTypeKeyIndex	130 (130)	15 (25)	5 (10)	0 (0)
5. com.netflix.hollow.core.write. HollowMapTypeWriteState	124 (122)	8 (13)	4 (4)	1 (1)

Zwischen den Metriken Method-to-Method und Inheritance gibt es eine sehr geringe Korrelation, sowohl im Bytecode als auch im Sourcecode. Dies ähnelt den Ergebnissen von Mockito, steht jedoch im Gegensatz zu denen von Guava. Besonders auffällig ist hier, die sehr niedrige Korrelation zwischen dem Field Coupling und Inheritance Coupling im Bytecode und der sogar leicht negativen Korrelation dieser im Sourcecode. Des Weiteren besteht eine grundsätzlich sehr geringe Korrelation zwischen dem Inheritance Coupling und allen anderen Metriken.

Wir wollen nun das Package Coupling der Bytecodeanalyse betrachten und es mit dem Method-to-Method Coupling der Klassen, die innerhalb derselben Packages lie-

## 6.5. Allgemeine Auffälligkeiten der Analyse der Projekte

**Tabelle 6.8.** Korrelationen der Metriken der Bytecodeanalyse

	Import	Field	Inheritance
Method	0.776	0.637	0.054
Import		0.643	0.221
Field			0.059

**Tabelle 6.9.** Korrelationen der Metriken der Sourcecodeanalyse

	Import	Field	Inheritance
Method	0.688	0.536	0.057
Import		0.485	0.233
Field			-0.033

gen, vergleichen. Einen Ausschnitt der Analyseergebnisse zeigen folgende drei Packages: *com.netflix.hollow.core.write* besitzt ein Package Coupling von 657. Die Klassen, welche in diesem Package liegen, erzeugen insgesamt ein Method-to-Method Coupling von 756. Das Package *com.netflix.hollow.tools.history* hat ein Coupling von 332 und die Klassen die darin liegen, besitzen insgesamt ein Method-to-Method Coupling von 408. Das Package *com.netflix.hollow.core.index* erzeugt ein Coupling von 299. Die Klassen innerhalb dieses Packages haben aufsummiert ein Method-to-Method Coupling von 373. Aus den Werten lässt sich erkennen, dass das Package Coupling zwar leicht niedriger als das Method-to-Method Coupling seiner Klassen ist, jedoch kein besonders großer Unterschied zwischen den Metriken liegt, wie zum Beispiel bei Guava erkennbar ist.

Aus den daraus resultierenden Erkenntnissen lässt sich schlussfolgern, dass, ähnlich wie bei Mockito, der Hauptfokus bei der Strukturierung der Klassen in Packages nicht auf dem Coupling lag, sondern eher auf Grundlage ihres Aufgabenbereichs geschehen ist.

Auffällig ist hier auch der hohe Structural Debt Index mit einem Wert von 18660. Folgt man den Angaben von Sonargraph, welche mindestens 6 Minuten Arbeitszeit pro Verbindung angeben, die aufgelöst werden muss, würden 1866 Stunden Arbeit benötigt werden, um alle Verbindungen der Package-Zyklen aufzulösen.

## 6.5. Allgemeine Auffälligkeiten der Analyse der Projekte

Aus den drei analysierten Projekten lassen sich allgemeine Indikatoren feststellen. Method-to-Method und Import Coupling korrelieren in allen Projekten stark miteinander. Das bedeutet, dass man Method-to-Method Coupling als Indikator für Import Coupling und umgekehrt sehen kann.

Auffällig ist, dass der Structural Debt Index von Guava bei 0 liegt und dieses Projekt gleichzeitig die größte Anzahl an Vererbungen aufweist. Da der SDI auf Package-Zyklen basiert, folgt daraus, dass Guava keine projektinternen Package-Zyklen besitzt. Da Java

## 6. Evaluierung

nur zyklensfreie Einfachvererbung zulässt, könnte daraus geschlossen werden, dass gezielt Vererbungen als Mittel gegen solche Zyklen genutzt wurden. Im Gegensatz dazu weist Netflix Hollow weniger Inheritance Coupling auf, welches zusätzlich eine niedrige Korrelation mit Method-to-Method Coupling und einen vergleichsweise hohen Structural Debt Index besitzt.

Des Weiteren können wir feststellen, dass in allen Projekten ungefähr die Hälfte des Couplings projektinterne Verbindungen sind.

### 6.6. Vergleich von Source- und Bytecode Ergebnissen

Wir wollen nun die Unterschiede der Analysen des Source- und Bytecodes der verschiedenen Projekte miteinander vergleichen. Dafür betrachten wir die Korrelationen zwischen denselben Metriken der beiden Analysen, um somit erkennen zu können, wie ähnlich das Coupling jeweils im Sourcecode und Bytecode gezählt wurde. Klassen, die nicht von beiden Analysen gefunden wurden, werden für die Korrelationen vernachlässigt.

Tabelle 6.10, Tabelle 6.11 und Tabelle 6.12 stellen jeweils für die drei analysierten Projekte die Korrelationen zwischen denselben Metriken, die Summe des Couplings von projektinternen Verbindungen ohne Selbstverbindungen (und die Summe aller gefundenen Verbindungen), die Differenz des Couplings und die entstandenen Fehlerquoten der Sourcecodeanalyse sowie der Bytecodeanalyse dar.

Im Allgemeinen lässt sich feststellen, dass die Bytecodeanalyse mehr Coupling misst als die Sourcecodeanalyse. Eine Ausnahme stellt das Method-to-Method und das Package Coupling aus dem Projekt Netflix Hollow dar. Einige Gründe dafür, dass Bytecode im Allgemeinen ein höheres Coupling aufweist, sind die in Kapitel 4 beschriebenen Unterschiede. Demnach können wir bei den betrachteten Projekten nicht davon ausgehen, dass nennenswerte Kompileroptimierungen stattgefunden haben, die anhand des Couplings im Bytecode hätten festgestellt werden können.

**Tabelle 6.10.** Vergleich zwischen Source- und Bytecodeanalyse von Mockito

	Method	Import	Field	Inheritance	Package
Korrelationskoeffizient	0.999	0.816	0.852	0.705	0.999
Summe Sourcecode	1923 (5129)	1273 (2028)	173 (584)	228 (356)	1619 (4965)
Summe Bytecode	1951 (5231)	1518 (3945)	194 (681)	269 (406)	1654 (5231)
Differenz	± 28	± 245	± 21	± 41	± 35
Fehlerquote Sourcecode	1.07 %	0.0 %	1.36 %	0.0 %	1.06 %
Fehlerquote Bytecode	0.0 %	0.0 %	0.0 %	4.43 %	0.0 %

Die Fehlerquote der Bytecodeanalyse von Mockito in Tabelle 6.10 im Inheritance Coupling

## 6.6. Vergleich von Source- und Bytecode Ergebnissen

von 4.43% entsteht hauptsächlich dadurch, dass Superklassen in einem Package erwartet werden, welches nicht unter dem erwarteten Pfad existiert.

**Tabelle 6.11.** Vergleich zwischen Source- und Bytecodeanalyse von Guava

	Method	Import	Field	Inheritance	Package
Korrelationskoeffizient	0.987	0.548	0.815	0.960	0.999
Summe Sourcecode	8435 (24505)	1323 (5260)	437 (2640)	484 (1356)	3005 (23617)
Summe Bytecode	8656 (26585)	4188 (14564)	514 (3547)	824 (1784)	3061 (26585)
Differenz	$\pm 221$	$\pm 2865$	$\pm 77$	$\pm 340$	$\pm 56$
Fehlerquote Sourcecode	1.73 %	0.0 %	4.05 %	0.0 %	1.77 %
Fehlerquote Bytecode	0.0 %	0.0 %	0.0 %	0.0 %	0.0 %

Die hohe Fehlerquote des Field Couplings der Sourcecodeanalyse in Tabelle 6.11 lässt sich größtenteils auf fehlerhafte Auflösung von generischen Typen zurückführen, die wir, wie in Abschnitt 4.4 beschrieben, nicht zählen.

**Tabelle 6.12.** Vergleich zwischen Source- und Bytecodeanalyse von Netflix Hollow

	Method	Import	Field	Inheritance	Package
Korrelationskoeffizient	0.989	0.856	0.902	0.822	0.998
Summe Sourcecode	7014 (14460)	2289 (3211)	587 (1688)	266 (334)	5561 (12362)
Summe Bytecode	6771 (13452)	2960 (6428)	627 (1921)	304 (404)	5531 (13452)
Differenz	$\pm 243$	$\pm 671$	$\pm 40$	$\pm 38$	$\pm 30$
Fehlerquote Sourcecode	0.65 %	0.0 %	0.82 %	0.0 %	0.71 %
Fehlerquote Bytecode	0.0 %	0.0 %	0.0 %	0.0 %	0.0 %

Abschließend kann man erkennen, dass es eine starke Korrelation zwischen denselben Metriken der Source- und Bytecodeanalyse gibt, auch wenn diese teilweise hohe Differenzen aufweisen. Insgesamt lässt sich feststellen, dass die Ergebnisse der beiden Analysen nicht stark voneinander abweichen. Im Allgemeinen kann man auf Grundlage unserer Ergebnisse sagen, dass es reichen würde, nur eine der beiden Analysen durchzuführen, um eine Übersicht über das Coupling zu erhalten. Dabei ist eine Analyse des Bytecodes weniger komplex als die des Sourcecodes, da der Aufwand durch das Parsen entfällt. Dennoch konnten einige Unterschiede durch die Sourcecodeanalyse festgestellt und daraus Erkenntnisse gewonnen werden.





## Quick Start Guide

Da es sich bei diesem Projekt um ein Gradle Projekt handelt, muss unter Linux zunächst `./gradlew build` und anschließend `./gradlew run` ausgeführt werden. Unter Windows werden dieselben Befehle mit der Datei `gradlew.bat` ausgeführt. Danach sollte sich das Fenster, wie in Abbildung 7.1 gezeigt, öffnen.

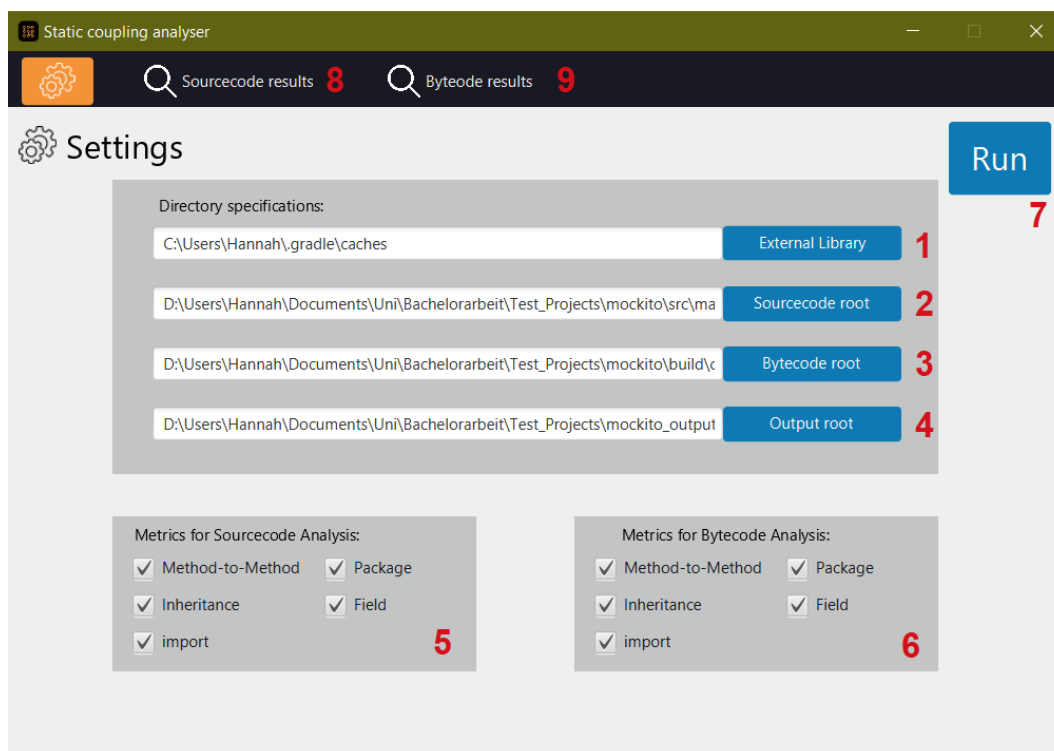


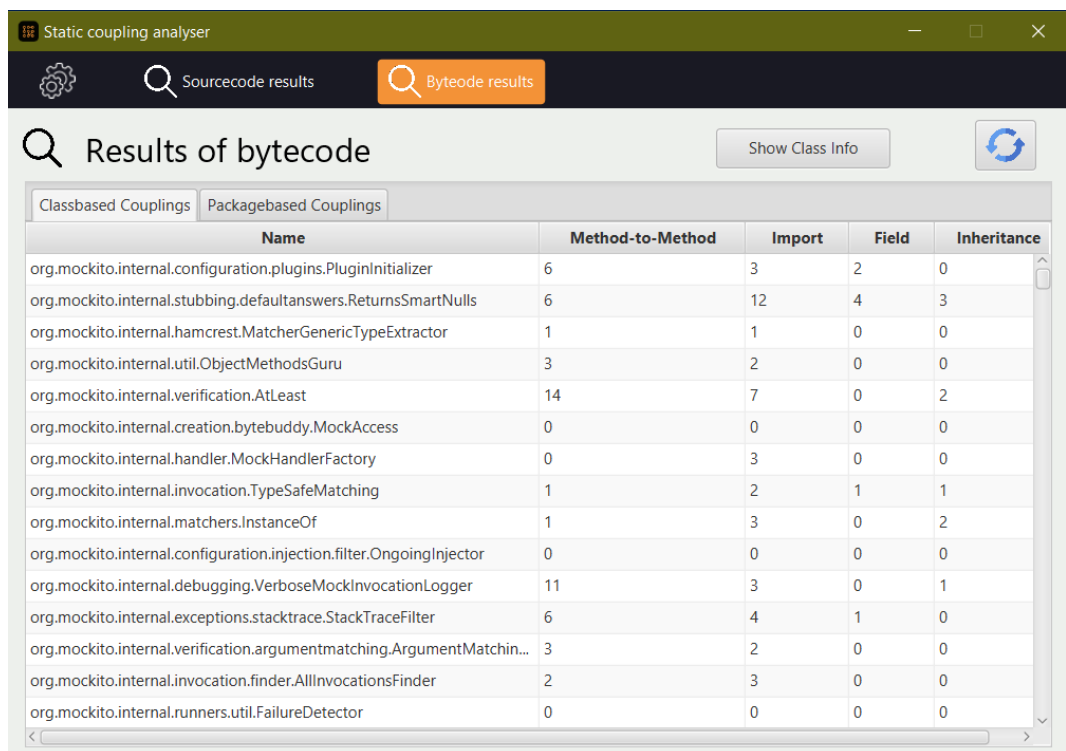
Abbildung 7.1. Einstellungen

In diesem Fenster können alle notwendigen Einstellungen für die Source- und Bytecodeanalyse getätigt werden.

Zunächst müssen die Dateiverzeichnisse gewählt werden. Das mit (1) gekennzeichnete

## 7. Quick Start Guide

Feld ermöglicht es, externe Libraries für die Sourcecodeanalyse hinzuzufügen. Wenn ein Gradle oder Maven Projekt analysiert werden soll, wird hier der jeweilige Cache gewählt. Unter (2) kann das Hauptdateiverzeichnis des Sourcecodes bestimmt werden. Wichtig hierbei ist, dass das richtige Verzeichnis und kein Ober- oder Unterverzeichnis verwendet werden darf. Ein gültiger Pfad wäre `'*/src/main/java'`, nicht richtig hingegen wäre `'*/src/main'`. Unter (3) lässt sich das Hauptdateiverzeichnis des Bytecodes auswählen. Zusätzlich sollte ein Verzeichnis für die Ausgaben bestimmt werden. Dies ist bei dem mit (4) gekennzeichneten Feld möglich. Mit den Checkboxen bei (5) und (6) können verschiedene Metriken an- und abgewählt werden, nach denen der jeweilige Code analysiert werden soll (Hinweis: Im aktuellen Entwicklungsstand wird für den Bytecode immer eine Analyse für Package Coupling durchgeführt, da die Berechnung des SDI diese benötigt). Um die Analyse zu starten, muss der 'Run'-Button angeklickt werden. Die Ergebnisse der Source- und Bytecodeanalyse können unter (8) und (9) eingesehen werden. Bei korrekter Ausführung erhält man beispielsweise folgende Ansicht, wie in Abbildung 7.2 dargestellt:



Name	Method-to-Method	Import	Field	Inheritance
org.mockito.internal.configuration.plugins.PluginInitializer	6	3	2	0
org.mockito.internal.stubbing.defaultanswers>ReturnsSmartNulls	6	12	4	3
org.mockito.internal.hamcrest.MatcherGenericTypeExtractor	1	1	0	0
org.mockito.internal.util.ObjectMethodsGuru	3	2	0	0
org.mockito.internal.verification.AtLeast	14	7	0	2
org.mockito.internal.creation.bytebuddy.MockAccess	0	0	0	0
org.mockito.internal.handler.MockHandlerFactory	0	3	0	0
org.mockito.internal.invocation.TypeSafeMatching	1	2	1	1
org.mockito.internal.matchers.InstanceOf	1	3	0	2
org.mockito.internal.configuration.injection.filter.OngoingInjector	0	0	0	0
org.mockito.internal.debugging.VerboseMockInvocationLogger	11	3	0	1
org.mockito.internal.exceptions.stacktrace.StackTraceFilter	6	4	1	0
org.mockito.internal.verification.argumentmatching.ArgumentMatchin...	3	2	0	0
org.mockito.internal.invocation.finder.AllInvocationsFinder	2	3	0	0
org.mockito.internal.runners.util.FailureDetector	0	0	0	0

Abbildung 7.2. Ergebnisse der Bytecodeanalyse

Die Ergebnisse werden hier tabellarisch dargestellt und lassen sich nach den Attributen auf- und absteigend sortieren. Auch ist es möglich, einen Klassennamen auszuwählen

und mit Hilfe des Buttons 'Show Class Info' eine detaillierte Ansicht zu erhalten, welche Couplings gefunden wurden.

Unter dem ausgewählten Ausgabeverzeichnis ist unter dem Ordner 'outputs' eine 'meta.log'-Datei zu finden, die den Ablauf des Programms sowie eine Übersicht über die Ergebnisse und Fehler enthält, die während der Analyse gefunden wurden. Ein weiterer Ordner 'csv-files' beinhaltet zu jeder Metrik eine CSV-Datei mit den Ergebnissen.



# Verwandte Arbeiten

Angelehnt an die Arbeit von Hasselbring und Schnoor „Comparing Static and Dynamic Weighted Software Coupling Metrics“, die sich mit dem Vergleich von statischem Coupling und gewichtetem dynamischen Coupling beschäftigt haben, haben wir uns mit dem Unterschied von statischem Coupling im Source- und Bytecode beschäftigt. Dafür mussten Metriken definiert werden, für die wir uns an der Arbeit von Bidve und Sarasu [„Tool for Measuring Coupling in Object-Oriented Java Software“] orientiert haben. Diese haben ebenfalls Metriken für die statische Analyse von Coupling definiert, um diese später in einem selbst implementierten Analysetool zu verwenden. Leider war es nicht möglich dieses Tool online zu finden. In einer weiteren Arbeit haben Bidve und Sarasu ihr Tool als Webanwendung weiterentwickelt [„Web Based Tool for Measuring Coupling in Object-Oriented Software Modules“], jedoch ist auch dieses nicht zugänglich.



# Fazit und Ausblick

## 9.1. Fazit

Eines der Ziele war es, Metriken festzulegen, die geeignet für die Sourcecode- und Bytecodeanalyse sind. Dies ist uns größtenteils gelungen, auch wenn die genaue Umsetzung der Metriken für Import und Inheritance Coupling in Source- und Bytecode auf unterschiedliche Art und Weise geschehen ist.

Wir konnten alle Metriken erfolgreich in unserem Analysetool implementieren. Es war das Hauptziel unseres Bachelorprojekts: Ein Analysetool zu entwickeln, welches statisches Coupling von der objekt-orientierten Sprache Java auf Grundlage von Metriken messen kann. Dabei hat sich die Analyse des Sourcecodes als komplexer erwiesen, da die Namensauflösung durch den SymbolSolver des JavaParsers anfangs zu vielen Fehlern führte. Trotz des Mehraufwands haben wir uns dazu entschieden, eine grafische Benutzeroberfläche zu integrieren, welches eigentlich über unsere vorher definierten Ziele hinausging. Bei der Implementierung haben wir darauf geachtet, sinnvoll zu kommentieren und dass Metriken später einfach hinzugefügt werden können, damit unser Tool weiterentwickelt werden kann.

Das dritte Ziel bestand darin, unser Analysetool auf verschiedene Projekte anzuwenden. Wir konnten drei bekannte Projekte auf statisches Coupling analysieren, die zum Teil bis zu etwa 24000 Verbindungen allein in einer Metrik aufgewiesen haben. Weiter war es unser Ziel, die Ergebnisse zwischen dem Source- und Bytecode auszuwerten und miteinander zu vergleichen. Dabei haben wir festgestellt, dass sich die Source- und Bytecodeanalyse nicht signifikant voneinander unterscheiden. Wir haben zusätzlich herausgefunden, dass trotz der Unterschiede, die Werte miteinander korrelieren. Das bedeutet, es ist nur notwendig, eine der beiden Analysen durchzuführen, um eine Übersicht über das Coupling zu erhalten. Die Bytecodeanalyse hat sich dabei als einfacher erwiesen. Ein interessanter Aspekt ist die Korrelation zwischen Method-to-Method und Import Coupling. Dies bedeutet, dass Entwickler einen Eindruck über ihr Method-to-Method Coupling anhand der Anzahl der Importe feststellen können. Ebenfalls interessant ist der Zusammenhang zwischen Vererbungen als Mittel zur Verringerung von Zyklen.

## 9. Fazit und Ausblick

### 9.2. Ausblick

Im Verlauf dieses Projektes haben sich mehrere Aspekte für spätere Arbeiten herauskristallisiert. Zunächst gäbe es die Möglichkeit, das statische Coupling genauer mit dynamischem Coupling zu vergleichen und eventuell auch auf Unterschiede von Source- und Bytecode einzugehen. Ein weiterer interessanter Punkt wäre es herauszufinden, ob andere Java Virtual Machines zu anderen Ergebnissen führen würden. Gerade im Kontext der JVM wäre es zudem spannend andere Programmiersprachen, die auf dieser basieren, zu betrachten und dahingehend zu analysieren, ob ähnliche Ergebnisse feststellbar sind. Auch denkbar wäre es, das Tool um weitere Metriken und Funktionalitäten zu ergänzen, wie beispielsweise eine Darstellung des Couplings durch interaktive Graphen. Da wir uns in dieser Arbeit ausschließlich mit Coupling beschäftigt haben, könnte die Betrachtung von Kohäsion ebenfalls interessant sein. In diesem Zusammenhang könnte auch die Wechselwirkung zwischen Kohäsion und Coupling untersucht werden.



# Literaturverzeichnis

- [*Apache Commons BCEL*]. Apache Commons BCEL. 4. Juni 2021. URL: <http://commons.apache.org/proper/commons-bcel/>. (Siehe Seite 6)
- [„Technical Debt Indexes Provided by Tools: A Preliminary Discussion“] F. Arcelli Fontana, R. Roveda und M. Zanoni. Technical Debt Indexes Provided by Tools: A Preliminary Discussion. In: Okt. 2016, Seiten 28–31. (Siehe Seite 13)
- [„Tool for Measuring Coupling in Object-Oriented Java Software“] V. Bidve und P. Sarasu. Tool for Measuring Coupling in Object-Oriented Java Software. *International Journal of Engineering and Technology* 8 (Apr. 2016), Seiten 812–820. (Siehe Seiten 5, 11, 12, 45)
- [„Web Based Tool for Measuring Coupling in Object-Oriented Software Modules“] V. Bidve, P. Sarasu, S. Pathan und G. Pakle. Web Based Tool for Measuring Coupling in Object-Oriented Software Modules. *International Journal of Intelligent Engineering and Systems* 12 (Aug. 2019), Seiten 201–211. (Siehe Seite 45)
- [„A metrics suite for object oriented design“] S. Chidamber und C. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering* 20.6 (1994), Seiten 476–493. (Siehe Seite 9)
- [„Towards a Metrics Suite for Object Oriented Design“] S. R. Chidamber und C. F. Kemerer. Towards a Metrics Suite for Object Oriented Design. In: *Conference Proceedings on Object-Oriented Programming Systems, Languages, and Applications*. OOPSLA '91. Phoenix, Arizona, USA: Association for Computing Machinery, 1991, Seiten 197–211. URL: <https://doi.org/10.1145/117954.117970>. (Siehe Seiten 5, 9)
- [„A survey on software coupling relations and tools“] E. Fregnan, T. Baum, F. Palomba und A. Bacchelli. A survey on software coupling relations and tools. *Inf. Softw. Technol.* 107 (2019), Seiten 159–178. (Siehe Seite 5)
- [*Guava*]. Guava. 18. Aug. 2021. URL: <https://github.com/google/guava>. (Siehe Seite 33)
- [*hello2morrow*]. hello2morrow. *Empowering Software Craftsmanship*. 11. Aug. 2021. URL: <https://www.hello2morrow.com/products/sonargraph>. (Siehe Seite 13)
- [*JavaFX*]. JavaFX. 19. Aug. 2021. URL: <https://openjfx.io/>. (Siehe Seite 7)
- [*Mockito*]. Mockito. 18. Aug. 2021. URL: <https://github.com/mockito/mockito>. (Siehe Seite 32)
- [*Netflix Hollow*]. Netflix Hollow. 18. Aug. 2021. URL: <https://github.com/Netflix/hollow>. (Siehe Seite 35)
- [„Comparing Static and Dynamic Weighted Software Coupling Metrics“] H. Schnoor und W. Hasselbring. Comparing Static and Dynamic Weighted Software Coupling Metrics. *Computers* 9.2 (2020). URL: <https://www.mdpi.com/2073-431X/9/2/24>. (Siehe Seiten 5, 45)

## Literaturverzeichnis

[*JavaParser: Visited*] N. Smith, D. van Bruggen und F. Tomassetti. *JavaParser: Visited. Analyse, transform and generate your Java code base*. 5. Feb. 2021. (Siehe Seiten 6, 7)