**Gottfried Wilhelm**
**Leibniz Universität Hannover**
**Fakultät für Elektrotechnik und Informatik**
**Institut für Praktische Informatik**
**Fachgebiet Software Engineering**

# Statische Codeanalyse zur Ermittlung des Testbedarfs von langlebiger Software

**Bachelorarbeit**

im Studiengang Informatik

von

**Patrick Liedtke**

**Prüfer: Prof. Dr. Joel Greenyer**
**Zweitprüfer: Prof. Dr. Kurt Schneider**
**Betreuer: Dipl.-Inform. Stefan Gärtner**

**Hannover, 28.08.2014**

## Zusammenfassung

In Unternehmen wird Software oft über viele Jahre hinweg genutzt und gleichzeitig an geänderte Bedürfnisse angepasst. Diese ständigen Anpassungen führen zu einer Degenerierung der Struktur der Anwendung und erschweren zukünftige Änderungen, auch hinsichtlich der Abschätzung des zu erwartenden Testbedarfs. Statische Codeanalysen sollen bei der Abschätzung helfen, können bei vielen dynamischen Konstrukten aber keine Ergebnisse ermitteln.
In dieser Arbeit wird ein Konzept vorgestellt mit dem einige dieser dynamischen Konstrukte teilweise aufgelöst werden können und so die statische Analyse erweitern. Betrachtet werden die Auflösung von Interfacevariablen zu konkreten Implementierungstypen und Aufrufe einer Methode oder Eigenschaft von abgeleiteten Klassen. Eine prototypische Umsetzung für die Sprache Java wird mit einer zur Zeit im Betrieb befindlichen Software der Dirk Rossmann GmbH und eingigen OpenSource-Programmen getestet.

## Abstract

Companies often use the same software for many years, while adapting it for any changed requirements. This continuous adaptation leads to degeneration of the software, making it harder to implement future changes and to estimate the amount of required test effort. Static code analysis is helpful in estimating the test effort, but it fails to give proper results on many dynamic features of managed code.
This thesis will present a concept to partially solve some of these dynamic features and therefore improve the static analysis. It will address the resolution of interface variables to their concrete implementation types, as well as calling a method or property of an extended class. A prototype will be implemented for the Java language and tested on current operating software from the Dirk Rossmann GmbH, as well as a few open source tools.

# Inhaltsverzeichnis


**1. Einleitung** **1**
1.1. Problemstellung . . . . . 1
1.2. Ziele der Arbeit . . . . . 2
1.3. Gliederung . . . . . 2

**2. Grundlagen** **3**
2.1. Softwareartefakte . . . . . 3
2.2. Abhängigkeiten im Quellcode . . . . . 3
2.3. Abhängigkeitsbaum . . . . . 4
2.4. Statische und Dynamische Codeanalyse . . . . . 4
2.4.1. Statische Analyse . . . . . 5
2.4.2. Dynamische Analyse . . . . . 5
2.5. Existierende Arbeiten . . . . . 6

**3. Verfahren zur Abschätzung des Testbedarfs** **7**
3.1. Vorraussetzungen und Einschränkungen . . . . . 7
3.1.1. Programmiersprache Java . . . . . 8
3.1.2. Bytecode und Quelltext . . . . . 9
3.1.3. Statische Analyse . . . . . 9
3.1.4. Keine Auswertung von Code . . . . . 9
3.2. Erstellung des Abhängigkeitsbaum . . . . . 9
3.3. Detailanalyse des Abhängigkeitsbaum . . . . . 10
3.3.1. Analyse von konkreten Typen hinter Interfaces . . . . . 11
3.3.2. Erkennung von vererbten Artefakten . . . . . 12
3.4. Aussagekraft des Abhängigkeitsbaums und Metriken . . . . . 12

**4. Erstellung und Verbesserung des Abhängigkeitsbaum** **14**
4.1. Erstellung des Abhängigkeitsbaums . . . . . 14
4.1.1. Werkzeuge zur Bestimmung der Abhängigkeiten . . . . . 14
4.1.2. Erstellung des Baumes . . . . . 19
4.1.3. Klassenbeziehungen bestimmen . . . . . 20
4.2. Verbesserungen im Abhängigkeitsbaum . . . . . 21
4.2.1. Aufrufe von Methoden einer Vaterklasse . . . . . 22
4.2.2. Auflösen von Interfacetypen . . . . . 23
4.3. Parsing von Java-Quelltext . . . . . 26
4.3.1. Genereller Ablauf . . . . . 27
4.3.2. Variablen . . . . . 27
4.3.3. Erstellung von Objekten mittels `new`-Operator . . . . . 28
4.3.4. Casts . . . . . 28

4.3.5. Funktionsaufrufe . . . . . . . . . . . . . . . . . . . . . . . . . . . 28
4.3.6. Ternäre Operatoren . . . . . . . . . . . . . . . . . . . . . . . . . 30
4.3.7. Primitive Datentypen . . . . . . . . . . . . . . . . . . . . . . . . 30
4.3.8. Erweiterte `For`-Schleife . . . . . . . . . . . . . . . . . . . . . . 31
4.3.9. Überprüfung der im Quelltext gefundenen Typen . . . . . . . . . 31
4.4. Metriken anhand des Abhängigkeitsbaum berechnen . . . . . . . . . . . 31

**5. Prototypische Implementierung 34**
5.1. Speicher- und Laufzeitprobleme . . . . . . . . . . . . . . . . . . . . 34
5.1.1. Optimierung des Prototypen hinsichtlich des Evaluationsziels . . 34
5.1.2. Limitierung der Häufigkeit einer Abhängigkeit . . . . . . . . . . . 35
5.1.3. Auswirkungen auf die Ergebnisse . . . . . . . . . . . . . . . . . 36
5.2. Behandlung von Zyklen . . . . . . . . . . . . . . . . . . . . . . . . . 36
5.3. Grafische Darstellung der Ergebnisse . . . . . . . . . . . . . . . . . . 36

**6. Evaluation 38**
6.1. Aufbau . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 38
6.2. Ergebnisse . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 41
6.3. Diskussion . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 46
6.3.1. Interpretation . . . . . . . . . . . . . . . . . . . . . . . . . . . 46
6.3.2. Korrektheit der Ergebnisse . . . . . . . . . . . . . . . . . . . . 47
6.3.3. Vollständigkeit der Ergebnisse . . . . . . . . . . . . . . . . . . 47

**7. Fazit und Ausblick 48**
7.1. Fazit . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 48
7.2. Ausblick . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 48

**A. Testmethoden zur Auswahl eines Werkzeugs 53**

**B. Beschreibung zur Nutzung des Prototypen 56**
B.1. Prototyp zur statischen Analyse . . . . . . . . . . . . . . . . . . . . . 56
B.2. Prototyp zur Visualisierung . . . . . . . . . . . . . . . . . . . . . . . 58

**C. Inhalt der CD 59**

# 1. Einleitung

## 1.1. Problemstellung

Viele Unternehmen benutzen speziell an die eigenen Bedürfnisse angepasste Software. Diese Software wird oft ein einziges mal entworfen und entwickelt, und anschließend im Lauf der Zeit an neue oder geänderte Bedürfnisse innerhalb des Unternehmens angepasst. Je nach Größe der geänderten Anforderungen ändert sich die Grundfunktionalität der Software nicht oder nur minimal, so dass häufig keine Zeit zum vollständigen Testen der implementierten Änderungen eingeplant wird. Oftmals wird außerdem die Dokumentation der Software nicht verwaltet und angepasst. Diese minimalen Änderungen der Anforderungen (und damit der Software) wiederholen sich und resultieren schließlich irgendwann in einer Degenerierung der Software [15]. Damit ist die langsame Entartung des Designs durch nachträgliche Änderungen der Software gemeint. Trotzdem wird solche Software häufig als Produktivsystem weitergenutzt. Minimale Änderungen lassen oft den Aufwand für eine korrekt ins Design der Software eingebettete Änderung unverhältnismäßig hoch erscheinen. Stattdessen werden bestehende Klassen oder Funktionen erweitert und verhalten sich infolgedessen nicht mehr wie ursprünglich vorgesehen. Durch fehlende Dokumentationen lassen sich diese Änderung später nur schwer nachvollziehen. Durch fehlende Testfälle ist außerdem die korrekte Funktionalität der einzelnen geänderten Methoden und Klassen nicht mehr gewährleistet. Diese Degenerierung der Software führt zu immer unübersichtlicheren Auswirkungen von neuen Änderungen innerhalb einer oder mehrerer Methoden auf den Rest der Software. Soll beispielsweise eine Methode angepasst werden, die laut ursprünglicher (bei der erstmaligen Entwicklung angelegten) Dokumentation der Software lediglich die Ausgabe der Daten beeinflusst, so kann diese Methode nach Jahren der Anpassungen plötzlich noch ganz andere Teile des Programms aufrufen um Änderungen an der Oberfläche mit minimalem Aufwand zu ermöglichen.
Soll nun eine weitere Änderung an dieser Software vorgenommen werden, so kann man sich nicht mehr auf das ursprüngliche Design und die ursprüngliche Dokumentation verlassen. Eine Aufwandsanalyse ist nötig (siehe Kapitel 2.2), um die zu erwartende Tragweite der Änderung innerhalb des Programms zu ermitteln und eine Einschätzung zum Aufwand des anschließenden Testens zu ermitteln. Je nach Ergebnis der Analyse kann es für ein Unternehmen einfacher zu entscheiden sein, ob der Nutzen der Änderungen im Verhältnis zum Aufwand der kompletten Implementierung (inklusive Testen) steht. Da dem Unternehmen nur begrenzt Programmierer und Zeit zur Verfügung stehen, handelt es sich hierbei auch um ein betriebswirtschaftliches Interesse. Die vielfache Notwendigkeit einer solchen Analyse beschreibt Harry M. Sneed in seiner Arbeit unter [11] ausführlich.
Eine solche Aufwandsanalyse lässt sich mit der Analyse des vorhandenen Quelltex-

tes ermöglichen. Dabei gibt es die Möglichkeiten der statischen und der dynamischen Analyse (siehe Kapitel 2.4), also der Analyse des geschriebenen (aber nicht interpretierten) Codes oder der Analyse des ausgeführten Codes. Da die Ergebnisse dieser Analysen als Indikator für den zu erwartenden Testaufwand dienen, beeinflussen sie die Einschätzung ob die geplanten Änderungen überhaupt in akzeptabler Zeit durchführbar sind. Deswegen besteht hier ein großes Interesse einer möglichst genauen Analyse. Deren Ergebnisse können dann in einem Abhängigkeitsbaum (siehe Kapitel 2.3) dargestellt werden, der dann einen Hinweis auf den zu erwartenden Aufwand gibt.

## 1.2. Ziele der Arbeit

Das wesentliche Ziel dieser Arbeit ist es, die vorhandenen Möglichkeiten der statischen Code-Analyse zu verbessern und so die Aussagekraft der in Kapitel 1.1 angesprochenen Aufwandsanalyse zu erhöhen. Damit soll eine zuverlässigere Abschätzung des zu erwartenden Testaufwands erreicht werden als bisher mit bekannten Werkzeugen der statischen Analyse. Bisherige Werkzeuge betrachten meist jedes Artefakt einzeln und können deshalb nur begrenzte Aussagen treffen. In dieser Arbeit wird die Idee untersucht, diese einzelnen Ergebnisse untereinander in Beziehung zu setzen um damit die Aussagekraft zu erhöhen. Die gewonnenen Erkenntnisse werden mit Hilfe der Dirk Rossmann GmbH direkt an einer im Einsatz befindlichen Software getestet.

## 1.3. Gliederung

In Kapitel 2 werden zunächst die nötigen Grundlagen für das Verständnis dieser Arbeit vermittelt. In Kapitel 3 wird allgemein dargestellt, wie sich aus der Abhängigkeitsanalyse eines Softwareartefaktes Informationen zum Bewerten des Testaufwands ermitteln lassen. Dabei wird beschrieben, wie die Ergebnisse bisheriger Werkzeuge analysiert und weiterverarbeitet werden sollen. Anschließend wird ein Überblick über die Aussagekraft des Ergebnisses vermittelt. Außerdem werden einige Einschränkungen beschrieben, die sich auf die Analyse und die Aussage dieser Arbeit auswirken.
Kapitel 4 veranschaulicht dann die Umsetzung des Ansatzes aus Kapitel 3, und begründet dabei die gewählten Methoden. Außerdem wird detailiert auf ausgewählte Probleme der statischen Analyse eingegangen und jeweils eine mögliche Herangehensweise zur Verbesserung dieser erklärt. Einschränkungen, die durch die Implementierung des während dieser Arbeit entwickelten Programms entstanden sind werden in Kapitel 5 beschrieben und die Auswirkungen auf die Ergebnisse erklärt. Kapitel 6 evaluiert die vorgestellten Konzepte anhand einer von der Dirk Rossmann GmbH genutzten und zu Testzwecken zur Verfügung gestellten Software, sowie ergänzend an einigen ausgesuchten OpenSource-Projekten. Die Ergebnisse werden interpretiert und diskutiert. Kapitel 7 gibt schließlich ein abschließendes Fazit und beschreibt weiterführende Möglichkeiten, die auf dieser Arbeit aufbauen könnten.

# 2. Grundlagen

## 2.1. Softwareartefakte

In dieser Arbeit wird sich oft auf „Softwareartefakte“ (oder kurz „Artefakte“) bezogen. Der Begriff ist nicht eindeutig definiert [14], hier sind damit einzelne Einheiten eines Programms gemeint, die von anderen Artefakten aufgerufen oder genutzt werden können. Beispiele für Softwareartefakte sind Packages, Methoden, Klassen- und Membervariablen oder auch der `static{}` Bereich einer Klasse. Lokale Variablen einer Methode lassen sich im Gegenzug nicht von anderen Artefakten aufrufen, deshalb zählen sie explizit nicht als eigenständiges Artefakt. Die von solchen lokalen Variablen erzeugten Abhängigkeiten werden durch die Methode repräsentiert in welcher sie deklariert wurden.

## 2.2. Abhängigkeiten im Quellcode

Die Beziehung von einzelnen Softwareartefakten untereinander lassen sich mittels Abhängigkeiten darstellen. Dabei gibt es zwei verschiedene Arten: Eingehende und ausgehende Abhängigkeiten.
Ausgehende Abhängigkeiten treten auf, wenn ein Artefakt ein anderes Artefakt „benutzt“, also beispielsweise eine Methode aufruft, eine Klasse erweitert oder ein Interface implementiert [16]. Im Zusammenhang mit der Ermittlung des Testaufwands zeigen ausgehende Abhängigkeiten also alle Artefakte an, die mitgetestet werden müssen. Eingehende Abhängigkeiten entstehen im Gegenzug immer dann, wenn ein Artefakt von einem anderen genutzt wird. Jede Abhängigkeit zwischen zwei Artefakten kann deshalb sowohl als eingehend, als auch als ausgehend betrachtet werden.
Möchte man beispielsweise eine Methode `m()` vollständig testen, so muss neben der eigentlichen Methode auch jeder Funktionsaufruf und jede Instanz oder Instanziierung eines Objekts innerhalb dieser Methode getestet werden. Diese Funktionsaufrufe (inkl. Instanziierungen) stellen jeweils eine Abhängigkeit zu der zu testenden Methode dar. Ruft man in der Methode `m()` eine Methode `calledMethod()` auf, so besitzt `m()` eine Abhängigkeit zu `calledMethod()`.
Ein vollständiger Test einer Methode erfordert also einen vollständigen Test aller Abhängigkeiten dieser Methode. Diese Abhängigkeiten können ihrerseits ebenfalls wieder Abhängigkeiten besitzen, die dann ebenfalls relevant für einen vollständigen Test der Ausgangsmethode `m()` sind (sogenannte indirekte Abhängigkeiten). In diesem Beispiel könnte also `calledMethod()` wiederum Abhängigkeiten zu anderen Artefakten besitzen, welche dann wiederum in den Tests für `m()` berücksichtig werden

müssen.

```
1  public class ClassA {
2     public void m() {
3        ...
4        ClassB b = new ClassB();
5        b.calledMethod();
6        ...
7     }
8  }
9
10 public class ClassB {
11    public void calledMethod() {
12       ClassC.nextMethod();
13       ...
14    }
15 }
```

Listing 2.1: Verschiedene Formen von Abhängigkeiten

Listing 2.1 verdeutlicht die Beziehung der Methoden `m()` und `calledMethod()` untereinander. Außerdem sieht man, dass `calledMethod()` seinerseits wieder eine neue Methode `ClassC.nextMethod()` aufruft, die somit eine indirekte Abhängigkeit für `m()` darstellt.

## 2.3. Abhängigkeitsbaum

Die jeweils einzeln für ein Softwareartefakt ermittelten Abhängigkeiten lassen sich in eine Baumstruktur umwandeln, welche dann alle von einem Startartefakt erreichbaren Abhängigkeiten enthält. Dieses Startartefakt ist das *Wurzelelement* des Baumes. Seine direkten Nachfolger entsprechen den in der Analyse ermittelten direkten Abhängigkeiten. Jede dieser Abhängigkeiten kann wiederrum eigene Abhängigkeiten besitzen, die dann als Kinder an den jeweiligen Knoten im Abhängigkeitsbaum angehängt werden. Dies wird fortgeführt bis entweder kein Blatt mehr eigene Abhängigkeiten besitzt, oder die Blätter einen Zyklus bilden, also bereits als einer der eigenen Vorgänger im Baum vorhanden sind. Dieser Fall muss erkannt werden, da der Abhängigkeitsbaum sonst unendlich groß werden würde.
Abbildung 2.3 zeigt einen Abhängigkeitsbaum anhand des Beispiels aus Kapitel 2.2, mit `m()` als Wurzelelement. Zu beachten ist hier die zusätzliche Abhängigkeit von `ClassA` zum Konstruktor von `ClassB`.

## 2.4. Statische und Dynamische Codeanalyse

Codeanalyse lässt sich in zwei Kategorien aufteilen: Statisch und dynamisch. Beide Verfahren haben ihre jeweiligen Vor- und Nachteile. Im Folgenden werden die beiden

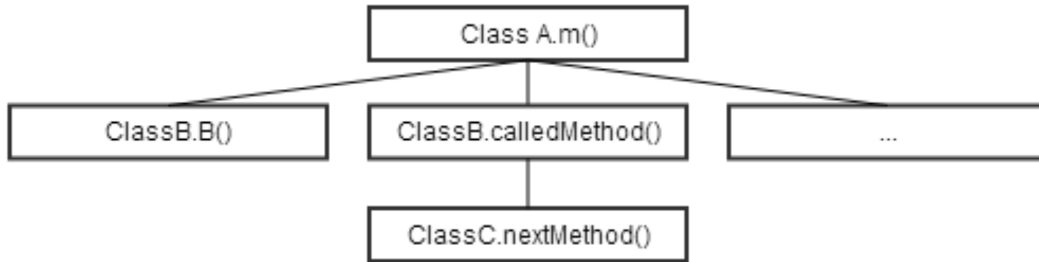


Abbildung 2.1.: Aufrufbaum der Methode m()

Methoden im Hinblick auf eine Abhängigkeitsanalyse untersucht.

### 2.4.1. Statische Analyse

Die statische Analyse wird auf dem Quelltext selbst, oder auf dem kompilierten Zwischencode (beispielsweise im Falle von Java) angewendet. Der zu untersuchende Code wird dabei nicht ausgeführt. Die statische Analyse untersucht *den gesamten* Code, hat also nicht das Pfadproblem der dynamische Analyse [12]. Außerdem kann die Analyse ohne Vorbereitung genutzt werden. Da der zu untersuchende Code nicht ausgeführt werden muss gibt es auch keine notwendigen Anpassungen.
Ein Nachteil der statischen Analyse sind dynamische Elemente im Code, die eigentlich erst zur Laufzeit aufgelöst werden. Dazu gehören je nach betrachteter Programmiersprache beispielsweise Polymorphismus, Reflections und Interfaces [13]. Die tatsächlichen Typen hinter diesen Elementen können sich zur Laufzeit sogar ändern, je nachdem über welchen Ausführungspfad sie aufgerufen werden.

### 2.4.2. Dynamische Analyse

Die dynamische Analyse kennzeichnet sich durch eine Analyse des Codes *während seiner Ausführung*. Es wird also mitprotokolliert was ausgeführt wird. Auf diese Weise lassen sich zuverlässig alle genutzten Typen bestimmen, denn spätestens bei der Ausführung muss jeder Typ eindeutig identifiziert sein.
Damit diese Analyse ausgeführt werden kann muss der Code jedoch instrumentalisiert werden [13], um Zwischenergebnisse bei der Ausführung abfangen zu können. Dies ist bei Software wie in 1.1 beschrieben ein Problem, da diese Software meist als Produktivsystem genutzt wird und ein Ausfall für das Unternehmen nicht tragbar ist. Ein weiteres Problem entsteht bei der Ausführung selber: Der ausgeführte Code wird detailliert untersucht und gewünschte Informationen werden mitprotokolliert. *Nicht ausgeführter* Code hingegen wird überhaupt nicht betrachtet. Um eine vollständige Analyse eines Programms zu erhalten wäre es also nötig *alle* vorhandenen Pfade des Programms zu berücksichtigen [3, S.2]. Die Ermittlung aller möglichen Pfade und

die anschließende Überprüfung ob wirklich alle Pfade ausgeführt worden sind stellt offensichtlich einen enormen Aufwand dar.

## 2.5. Existierende Arbeiten

Die Idee dynamische Eigenschaften einer Sprache bei der statischen Analyse zu verbessern ist nicht neu. Trotzdem sind nur wenige Arbeiten in diesem Bereich zu finden. Bisherige Ansätze wie in [5] konzentrieren sich beispielsweise auf die Auflösung von Reflection mittels verbesserter Stringanalyse. Andere Ansätze versuchen dynamische Metriken mit statischer Analyse zu berechnen [3] um so von den Vorteilen der dynamischen Analyse zu profitieren. Dabei nutzen sie aber andere Methoden als die in dieser Arbeit vorgestellten. Anstatt an einer bestimmten Stelle im Code die vorhandenen Typen und Abhängigkeiten zu konkretisieren werden alle möglichen Kombinationen möglicher konkreter Vorkommen berechnet, beispielsweise alle Implementierungstypen zu einem Interface. Außerdem arbeiten all diese Ansätze auf Klassenebene. Eine Untersuchung zur Auswertung einer statischen Analyse (auf Methodenebene) mithilfe von Beziehungen unter den einzelnen Ergebnissen ist mir zur Zeit nicht bekannt.

# 3. Verfahren zur Abschätzung des Testbedarfs

Zur Abschätzung des Testaufwands eines Softwareartefakts (Klasse, Methode, Package, etc) müssen alle anderen aufgerufenen oder genutzten Artefakte ermittelt werden. Deshalb wird eine Abhängigkeitsanalyse des zu untersuchenden Softwareartefakts durchgeführt.
Das bisher gängige Konzept der statischen Analyse, nämlich Klasse für Klasse einzeln auf referenzierte Typen und Methoden zu untersuchen, soll in dieser Arbeit erweitert und eine Beziehung der einzelnen Klassen untereinander mit in die Analyse einbezogen werden. Dazu wird zusätzlich zu der gängigen Analyse vom Benutzer ein *Startelement* (in dieser Arbeit auch „Wurzelelement“ genannt) bestimmt, von welchem aus die Abhängigkeiten in einer Baumstruktur betrachtet werden (siehe Kapitel 3.2). Diese Betrachtung erlaubt zusätzliche Informationen bei der Analyse zu verwenden, beispielsweise kann die aufrufende Methode bei der Analyse einer Methode für einige Zwecke mit in Betracht gezogen werden (beispielsweise in Kapitel 3.3). Die ermittelten Ergebnisse gelten an dieser Stelle natürlich nicht mehr global, da sich die Beziehung der einzelnen Abhängigkeiten je nach gewähltem Wurzelelement ändern. Zur Ermittlung des Testaufwands an einer lokalen Stelle im Code ist eine solche Ansicht der Abhängigkeiten aber unter Umständen zielführender als eine globale Betrachtung, wenn dadurch sonst unentdeckte Abhängigkeiten ermittelt werden können.
Abbildung 3.1 zeigt schematisch die einzelnen Schritte des Konzept. Aus den vorhandenen Code-Dateien (sowohl Class- als auch Java-Dateien) wird zunächst mittels existierender Werkzeuge ein Abhängigkeitsbaum generiert. Dieser wird anschließend nach Verbesserungsmöglichkeiten durchsucht und die gefundenen Knoten in einer Detailanalyse mittels der Methoden in Kapitel 3.3 erneut untersucht. In den fertigen Baum fließen zum Schluss noch Informationen aus vorhandenen Testreports mit ein, die einen Überblick über die Testabdeckung geben. Die einzelnen Schritte der Detailanalyse und der Errechnung der Testabdeckung mittels der Testreports wird ab Kapitel 3.2 beschrieben.

## 3.1. Vorraussetzungen und Einschränkungen

Diese Arbeit orientiert sich an einer Software der Dirk Rossmann GmbH und richtet sich in ihren Anforderungen primär nach dieser. Außerdem werden einige allgemeine Einschränkungen bei der Untersuchung gemacht, um den Umfang der Arbeit auf einige Kernbereiche zu reduzieren.

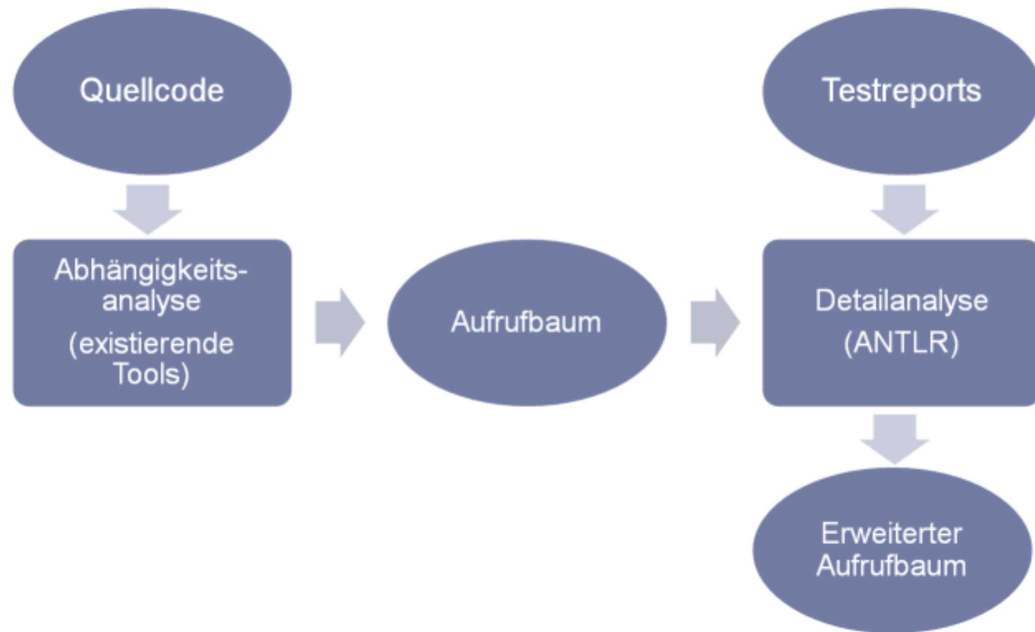


Abbildung 3.1.: Schematische Darstellung des Konzepts

### 3.1.1. Programmiersprache Java

In dieser Arbeit wird lediglich die Programmiersprache Java in der Version 1.4 berücksichtigt, da die zu testende Software in dieser Sprache und dieser Version geschrieben wurde. Das bedeutet, dass Sprachkonzepte aus höheren Javaversionen wie beispielsweise Generics nicht unterstützt werden. Außerdem werden Reflections nicht berücksichtigt, da diese mit einer Vielzahl an möglichen Aufrufen zu komplex für diese Arbeit sind und sich ohnehin nicht immer mit statischer Codeanalyse auflösen lassen [1, S. 53] [5].

Das Konzept von lokalen Klassen innerhalb anderer Klassen wird nur teilweise beachtet. Diese inneren Klassen werden als Abhängigkeiten angezeigt, allerdings nicht im Optimierungsprozess berücksichtigt. Der Grund dafür liegt in der geringen zu erwartenden Ergebnisverbesserung bezüglich der von der Dirk Rossmann GmbH zur Verfügung gestellten Software, da lokale Klassen so gut wie nicht genutzt werden. Außerdem bedeutet eine Nichtberücksichtigung von lokalen Klassen eine deutliche Reduzierung der Komplexität bei der Analyse des Java-Codes (siehe Kapitel 4.3).

Konzepte von anderen Programmiersprachen, welche nicht in Java vorhanden sind werden in dieser Arbeit ebenfalls nicht betrachtet.

### 3.1.2. Bytecode und Quelltext

Für diese Arbeit muss der Code der Software mindestens als Bytecode in Class-Dateien vorliegen, für einige Analysen auch als Quelltext in Java-Dateien. Erstere können sich dabei in einem Container wie beispielsweise einer Jar-Datei befinden. Die Class-Dateien werden zur erstmaligen Erkennung der Abhängigkeiten mittels des Programms DependencyFinder benötigt, die Quelltext-Dateien dienen später der Verbesserung der erkannten Abhängigkeiten. Fehlen einzelne Java-Dateien, so werden diese bei der späteren Analyse übersprungen.
Außerdem müssen die Java-Dateien in einer ihren Packages entsprechenden Verzeichnisstruktur liegen und dürfen nicht auf verschiedene Projekte verteilt sein, sie müssen also das gleiche Wurzelverzeichnis haben. Beispielsweise muss die Klasse `de.unihannover.se.Class1` in einem Verzeichnis „%wurzel%/de/unihannnover/se/“ liegen. Entsprechend muss eine Klasse `de.unihannover.se.Class2` in dem selben Verzeichnis liegen, inklusive gleicher „%wurzel%“ vor der Packagestruktur.

### 3.1.3. Statische Analyse

In dieser Arbeit wird eine statische Codeanalyse einer dynamischen vorgezogen, da es sich bei der zu untersuchenden Software um ein Produktivsystem handelt. Eine dynamische Analyse würde Änderungen im Quelltext selbst benötigen. Außerdem ist bei einer dynamischen Codeanalyse nicht garantiert, dass alle Verzweigungen und Aufrufe auch wirklich ausgeführt werden, wodurch Abhängigkeiten übersehen werden können [3, S.2]. Gerade zur Abschätzung von benötigten Testfällen sind aber alle möglichen Abhängigkeiten in allen Pfaden (Verzweigungen innerhalb einer Methode) für den Benutzer interessant. Die Nachteile einer statischen Codeanalyse, beispielsweise der ungenauen Erkennung von Typen hinter Interfaces, sollen in dieser Arbeit mittels nachträglicher Analyse möglichst minimiert werden.

### 3.1.4. Keine Auswertung von Code

In dieser Arbeit werden lediglich Zuweisungen, die Rückschlüsse auf den konkreten Typ einer Variablen zulassen berücksichtigt (siehe 4.3). Es gibt keine Unterscheidung von Verzweigungen im Code, also keine `if`- oder `switch`-Statements. Ebenso werden keine Operatoren wie `instaceof` oder Schleifen und andere Kontrollstrukturen ausgewertet.

## 3.2. Erstellung des Abhängigkeitsbaum

Die Abhängigkeiten von einzelnen Softwareartefakten untereinander werden oft als Indikator zur Einschätzung des Designs von Software genutzt [3], beispielsweise zur Bestimmung der Kopplung der Klassen untereinander. Deswegen gibt es bereits frei

verfügbare Werkzeuge, welche versuchen die Abhängigkeiten von Klassen und Paketen mittels statischer Codeanalyse zu ermitteln und zu Metriken zusammenzufassen. Speziell für Java entwickelte Werkzeuge werden im Abschnitt 4.1.1 im Hinblick auf die Anforderungen dieser Arbeit untersucht. Genutzt wird schließlich das OpenSource-Programm DependencyFinder[1], da die angebotenen Features sehr gut mit den Anforderungen dieser Arbeit zusammenpassen.
Auf Basis des Ergebnisses dieses Programms wird ein Abhängigkeitsbaum des gesamten zu untersuchenden Projektes erstellt, der die Grundlage für den nächsten Schritt stellt.

Da die gefundenen Abhängigkeiten je nach Größe des Projektes sehr zahlreich werden können, erhält der Benutzer die Möglichkeit einer Filterung aller Einträge mittels regulärer Ausdrücke. Unabhängig davon werden jedoch Artefakte aus den Paketen `java.*` und `javax.*` automatisch herausgefiltert und nicht mit in die Ergebnisse einbezogen. Einträge aus diesen beiden Paketen sind sehr zahlreich vertreten, da sie die Grundfunktionalität der Sprache Java repräsentieren. Bei der Bestimmung des Testaufwands können sie jedoch außer Acht gelassen werden, da Klassen aus diesen Paketen ohnehin nicht getestet werden.

Die von DependencyFinder ermittelten Abhängigkeiten liegen in einer einzigen Baumstruktur vor, welche alle Abhängigkeiten von allen Softwareartefakten enthält. Dieser Baum wird zunächst in eine lineare Datenstruktur umgewandelt, da der Urpsrungsbaum Informationen enthält die nicht benötigt werden, oder sogar korrigiert werden müssen (siehe Kapitel 4.1.1). Außerdem müsste der komplette Baum durchsucht werden, wollte man ein bestimmtes Artefakt untersuchen.
Nun kann von jeder beliebigen Klasse oder Methode der Software ein sogenannter Abhängigkeitsbaum erstellt werden. Das Wurzelelement ist das gewählte Artefakt selbst (beispielsweise genau eine Methode einer Klasse), die direkten Kindknoten repräsentieren die direkten Abhängigkeiten des Artefakts. Da jede Abhängigkeit selbst wieder über Abhängigkeiten verfügen kann, wird nun eine Abhängigkeitsanalyse für jeden der Kindknoten durchgeführt. So entsteht ein Baum, der alle zu dieser Zeit bekannten direkten und indirekten Abhängigkeiten enthält und als Knoten repräsentiert, ausgehend von dem zunächst ausgewählten Artefakt. Aus diesem Baum kann nun eine erste Abschätzung des Testaufwands mittels der Größe des Baumes oder seiner Tiefe erfolgen.

## 3.3. Detailanalyse des Abhängigkeitsbaum

Die bisher ermittelten Abhängigkeiten unterliegen den Einschränkungen des zur Ermittlung genutzten Werkzeugs, in diesem Fall DependencyFinder (siehe Abschnitt 4.1.1). Unter anderem gibt es also noch keine Informationen darüber welcher konkrete Implementierungstyp für ein Interface genutzt wird, und welche Methoden einer Vaterklasse aufgerufen werden. Um diese Einschränkungen zu umgehen, folgt nun eine eigene

[1] depfind.sourceforge.net/

statische Codeanalyse, mit der genau diese beiden Einschränkungen verbessert werden sollen.

### 3.3.1. Analyse von konkreten Typen hinter Interfaces

Durch die Angabe eines Wurzelartefaktes sind neue Informationen vorhanden, die zur Bestimmungszeit der Abhängigkeiten durch DependencyFinder noch nicht zugänglich waren. Erstellt man aus den in 3.2 ermittelten Abhängigkeiten einen Baum ist beispielsweise bekannt, welche Methode von welcher Methode aufgerufen wird (für ein gegebenes Wurzelartefakt des Abhängigkeitbaums), ohne das der Code dafür ausgeführt werden muss. Dadurch lassen sich Informationen von einem Artefakt in ein anderes überführen. Gibt es beispielsweise zwei Methoden `a()` und `b(TypeA)`, von denen man weiß das `b(TypeA)` von `a()` aufgerufen wird und dabei Parameter entgegennimmt, so können diese Parameter in `a()` genauer untersucht werden. Das ist beispielsweise dann sinnvoll, wenn der Typ eines Parameters in der Signatur ein Interface oder eine vererbende Klasse ist (hier beispielsweise `TypeA`). Zur Abschätzung des Testbedarfs ist es nicht sinnvoll solche Typen zu berücksichtigen, da sich der Aufwand je nach tatsächlichem Implementierungstypen stark unterscheiden kann. Für den Interfacetypen selbst endet der Abhängigkeitsbaum an dem Punkt des Methodenaufrufes sogar, denn da die Methode im Interface nicht implementiert ist gibt es auch keine weiteren Abhängigkeiten. Ähnliches gilt für abstrakte Klassen.
An dieser Stelle ist also eine Konkretisierung erwünscht, die jedoch nur mithilfe des Codes in `b(TypeA)`nicht möglich ist. Kennt man aber die aufrufende Methode, so kann man den Parameter untersuchen und versuchen, den genauen Typ des Objektes zu identifizieren. Dies wird in Abschnitt 4.2.2 ausführlicher beschrieben. Die Methode wird für alle Knoten des Baumes ausgeführt, bei denen man sich eine Verbesserung hinsichtlich der Genauigkeit des tatsächlichen Typs verspricht, also nicht nur Methoden mit entsprechenden Signaturen, sondern beispielsweise auch Member- und Klassenvariablen.
Werden durch diese Methode einzelne Knoten des Abhängigkeitsbaums konkretisiert (also beispielsweise ein Interfacetyp auf einen konkreten Implementierungstyp zurückverfolgt), so werden die entsprechenden Knoten im Baum durch die neuen Knoten ersetzt. Diese neuen Knoten könnten nun ihrerseits wieder neue Abhängigkeiten haben. Eine erneute Analyse mittels DependencyFinder ist aber nicht nötig, da die Abhängigkeiten zu jedem Artefakt in der linearen Datenstruktur liegen, die zu Beginn der Erstellung des Baumes angelegt wurde. Allerdings muss für jeden neuen Knoten die hier besprochene Optimierung durchgeführt werden, da auch die diese wiederum Abhängigkeiten haben könnten, die konkretisiert werden müssten.

Natürlich gibt es verschiedene Fälle, in denen eine solche Konkretisierung einfach nicht möglich ist. Ein Beispiel hierfür zeigt Listing 3.1. In Fällen wie diesen werden alle in Frage kommenden Implementierungstypen gespeichert und als *mögliche* Implementierung in den Abhängigkeitsbaum hinzugefügt. Diese Alternativen im Baum sollen dem Benutzer zumindest eine Einschränkung der möglichen Implementierungstypen für diese Stelle im Code bieten.

```
1  public void doSth(int useImplOne) {
2     InterfaceType interf;
3     if(useImplOne > 0)
4        interf = new Impl1();
5     else
6        interf = new Impl2();
7     doSthElse(interf);
8  }
9  public void doSthElse(InterfaceType param) {
10    ...
11 }
```

Listing 3.1: Nicht eindeutig bestimmbarer Typ

### 3.3.2. Erkennung von vererbten Artefakten

Eine weitere Einschränkung betrifft die Nutzung von Artefakten in einer Klasse `B`, die in einer Vaterklasse `A` implementiert wurden. Bisherige statische Analysen betrachten beide Klassen alleine für sich. Dadurch können solche Artefakte in Klasse `B` nicht aufgelöst werden, es wird lediglich der Methodenname aber keine Implementierung gefunden. Auch an dieser Stelle endet der Abhängigkeitsbaum an einem solchen Knoten. Setzt man die beiden einzelnen Analysen von Klasse `A` und Klasse `B` miteinander in Beziehung, so lassen sich die Abhängigkeiten in Klasse `B` durch die tatsächlichen Abhängigkeiten in `A` ersetzen und der Abhängigkeitsbaum kann weiter entwickelt werden. Dafür werden Informationen bezüglich der Beziehungen der einzelnen Klassen und Interfaces untereinander benötigt. Diese Informationen lassen sich ebenfalls aus den Abhängigkeiten ableiten, das genaue Vorgehen wird in Abschnitt 4.1.3 beschrieben. Die genaue Vorgehensweise ist in Abschnitt 4.2.1 beschrieben.

## 3.4. Aussagekraft des Abhängigkeitsbaums und Metriken

An dieser Stelle ist die Erstellung des Abhängigkeitsbaums abgeschlossen. Im Folgenden werden nun aus den gefundenen Abhängigkeiten Informationen gewonnen, die einen tatsächlichen Überblick über den zu erwartenden Testaufwand liefern. Der Abhängigkeitsbaum selbst stellt dabei die ermittelte Menge aller bekannten zu testenden Softwareartefakte dar. Je nach Art der Artefakte ergeben sich aber Unterschiede im zu erwartenden Testaufwand. Besteht der Baum beispielsweise aus 50% Klassenvariablen, bei denen maximal eine Überprüfung des zugewiesenen Wertes nötig ist, kann der zu erwartende Testaufwand als eher gering angesehen werden. Besteht die Mehrheit der Knoten allerdings aus Methoden die aufgerufen werden, so kann ein deutlich höherer Aufwand erwartet werden. Ziel ist also nun, Informationen zu gewinnen, die Rückschlüsse auf den zu erwartenden Testaufwand zulassen.

Schon auf Klassenebene hat sich gezeigt, dass Metriken einen klaren Hinweis auf den zu erwartenden Aufwand geben können [6]. Dafür kann man den Baum selbst betrachten und aus seiner Struktur Informationen direkt gewinnen. Beispielsweise repräsentiert die Tiefe des Baums die maximale Aufruftiefe der Software, ausgehend vom untersuchten Wurzelelement des Baumes. Auch die Größe des Abhängigkeitsbaums kann bereits als Metrik genutzt werden, denn sie stellt die Anzahl der zu testenden Artefakte dar. Hier muss allerdings berücksichtigt werden, dass es Zweige gibt die Alternativen darstellen. Diese müssen also bei der Zählung gesondert betrachtet werden.
Mithilfe des in dieser Arbeit erstellten Baumes können nun nicht nur Metriken über Klassen, sondern auch über Methoden oder einen bestimmten Aufrufpfad des untersuchten Projekts erstellt werden. So ist es denkbar komplexere Metriken wie beispielsweise McCabe zu nutzen um Pfade mit maximaler Komplexität zu finden. Die Berechnung auf Bytecode gestaltete sich jedoch zu aufwendig zu implementieren, so dass in dieser Arbeit auf andere Metriken zurückgegriffen wird.
Die prototypische Implementierung dieses Konzeptes erstellt beispielsweise eine Metrik zur Ermittlung der relativen Testabdeckung über den gesamten Abhängigkeitsbaum, die für jeden Knoten neu berechnet wird (die Dirk Rossmann GmbH verfügt nämlich bereits über Testfälle für viele Methoden der zu untersuchenden Software). Außerdem können die Anzahl der *unterschiedlichen* Klassen und Methoden im Baum bestimmt werden. Die Bestimmung der Metriken wird in Kapitel 4.4 genauer beschrieben.

# 4. Erstellung und Verbesserung des Abhängigkeitsbaum

Der Abhängigkeitsbaum ist das zentrale Element für die Abhängigkeitsanalyse in dieser Arbeit. Dieses Kapitel beschreibt die Erstellung, nachträgliche Verbesserung und Interpretation des Baumes.

## 4.1. Erstellung des Abhängigkeitsbaums

Die Darstellung von Abhängigkeiten von Softwareartefakten ist bereits mittels vieler frei verfügbarer Werkzeuge und für verschiedene Programmiersprachen möglich. In dieser Arbeit wird lediglich die Umsetzung für Software in der Sprache Java besprochen.

### 4.1.1. Werkzeuge zur Bestimmung der Abhängigkeiten

Für Java-Anwendungen gibt es bereits kostenlose und quelloffene Programme, die mit unterschiedlichen Ansätzen versuchen die Abhängigkeiten von Softwareartefakten untereinander zu ermitteln. Zu den bekanntesten zählen JDepend[1], DependencyAnalyzer[2], Dependometer[3] und DependencyFinder[4]. Sie alle ermöglichen das Auslesen der Abhängigkeiten einer Klasse und deren Methoden, sind also geeignet um Informationen für einen Abhängigkeitsbaum zu ermitteln. Alle diese Werkzeuge arbeiten auf dem Bytecode der Java-Anwendung. Der Grund dafür ist der Aufbau dieser Dateien, der ein komfortables Auslesen aller Eigenschaften einer Klasse ermöglicht [1, S. 26ff]. Abhängigkeiten die im Bytecode nicht angegeben sind fehlen jedoch, weshalb die nachträgliche Analyse aus Kapitel 3.3 benötigt wird.
Die Vorgehensweise der einzelnen Werkzeuge unterscheidet sich, so dass sie auf ihre Eignung für die Anforderungen dieser Arbeit hin untersucht werden müssen.

#### Anforderungen

Die oberste Anforderung ist die Vollständigkeit und Korrektheit der erkannten Abhängigkeiten. Außerdem sollte das Werkzeug entweder leicht erweiterbar sein, oder zumindest die Ergebnisse in einer Form gegeben sein, die eine einfache Weiterverarbei-

[1] http://clarkware.com/software/JDepend.html
[2] http://www.dependency-analyzer.org/
[3] http://source.valtech.com/display/dpm/Dependometer
[4] http://depfind.sourceforge.net/

tung ermöglicht.
Schon vor einem möglichen Testlauf dieser Programme zeigte sich, dass Dependometer nicht zu dieser Arbeit passt. Das Programm ist zwar in der Lage Abhängigkeiten zu erkennen, vergleicht diese aber mit einer durch den Benutzer vorgegebenen Struktur und meldet dann mögliche Verstöße. Diese Struktur muss zunächst erstellt und später bei Anpassungen gewartet werden, und würde deswegen die nötige Arbeit zur Abschätzung des Testbedarfs erhöhen. Auch JDepend erwies sich schnell als nicht nutzbar, denn das Programm erlaubt es nur Abhängigkeiten zwischen einzelnen Packages zu bestimmen und ist damit für das Konzept dieser Arbeit viel zu ungenau, als das sich eine weitere Analyse lohnen würde.
Übrig bleiben also die Programme DependencyAnalyzer und DependencyFinder.

Zum Testen der Erkennungsraten der einzelnen Programme wurden einige Testmethoden geschrieben, die dann von jedem dieser Programme untersucht wurden. Der Code befindet sich im Anhang A, die einzelnen Tests stellen mögliche Formen der Abhängigkeiten dar.

| Test\Name | DependencyAnalyzer | DependencyFinder |
|---|---|---|
| Instanziierung | Ja | Ja |
| Typ in Signatur | Ja | Ja |
| Cast | Ja | Ja |
| Rückgabewert in Interfacetyp | Ja[5] | Nein |
| Vererbte Methoden | Nein[6] | Ja |
| Membervariable | Ja | Ja |

Tabelle 4.1.: Ergebnisse des Konzeptes dieser Arbeit

Beide Werkzeuge konnten in etwa die gleichen Abhängigkeiten erkennen. Der einzige große Unterschied besteht in der Gliederung der Ergebnisse. DependencyAnalyzer ermittelt die Abhängigkeiten nur auf Klassenebene. Für das Konzept dieser Arbeit sind Abhängigkeiten auf Methodenebene sehr viel förderlicher, so dass DependencyFinder hier geeigneter ist.

#### DependencyFinder

Der DependencyFinder nimmt als Eingabe eine beliebige Anzahl an Class-Dateien (auch in Containern wie beispielsweise JAR) und bei Bedarf auch aus verschiede-

[5] DependencyAnalyzer baut auf Klassenebene einen Abhängigkeitsbaum ähnlich wie in dieser Arbeit auf. Deswegen wird aus dem Test nicht deutlich ob immer alle Typen angegeben werden die eine Abhängigkeit zu dem untersuchten Interface haben, oder nur die Typen die auch wirklich Rückgabewert einer Methode sein können.

[6] Da die Abhängigkeiten nur auf Klassenebene angezeigt werden lassen sich keine einzelnen Methoden testen

nen Verzeichnissen. Dies stellte sich bei der Dirk Rossmann GmbH vor Ort ebenfalls als vorteilhaft heraus, da die einzelnen Projekte dort mittels Apache Maven [7] erstellt werden und sich deshalb Teile der Class-Dateien in anderen Verzeichnissen befinden können. Aus diesen Class-Dateien ermittelt DependencyFinder dann drei unterschiedliche Arten von Abhängigkeiten:

1. Feature zu Feature - Abhängigkeiten von einem Artefakt (Methode, Klassenvariable, Enumeration oder der statische Bereich einer Klasse, aber keine anderen Klassen oder Packages) zu einem anderen Artefakt (mit denselben Einschränkungen)
2. Feature zu Klasse - Wenn ein Objekt dieser Klasse in dem untersuchten Feature genutzt wird oder durch das untersuchte Feature repräsentiert wird (bspw. Klassenvariable)
3. Klasse zu Klasse - Implementiert eine Klasse A ein Interface B, oder erbt eine Klasse A von einer anderen Klasse C, so entsteht diese Art von Abhängigkeit

Artefakt und Feature dürfen hier nicht vermischt werden. Ein Artefakt meint die in 2.1 beschriebene Definition, ein Feature ist eine Teilmenge davon und meint einen Unterteil einer Klasse, der von sich aus Abhängigkeiten zu anderen Artefakten (also Features, Klassen und Packages) erzeugen kann.
Alle Artefakte enthalten immer den vollqualifizierenden Namen. Wird in einer Abhängigkeit eine Methode `test` der Klasse `ClassA` im Package `de.unihannover.se` gefunden, die als Parameter ein Objekt vom Typ String erwartet, so lautet der vollständige Name dieses Artefakts (bzw. Feature):
`de.unihannover.se.ClassA.test(java.util.String)`.
Hier wird deutlich, dass *jeder* Typ, auch Parameter in einer Methode, vollständig erhoben werden.
Abhängigkeiten von Packages zu anderen Packages oder von Klassen zu Packages werden aus den drei oben aufgelisteten Abhängigkeitsformen abgeleitet. Gibt es beispielsweise eine Klasse `A` im Package `X` und eine abhängige Klasse `B` im Package `Y`, so entsteht automatisch eine Abhängigkeit des Packages `X` zum Package `Y`. Ähnliches gilt auch für Abhängigkeiten von einem Feature zu einem anderen, denn da ein Feature immer ein Element einer Klasse ist wird auch immer eine Abhängigkeit der beiden beinhaltenden Klassen impliziert.
Die ermittelten Abhängigkeiten treten in zwei Varianten auf: Eingehende und Ausgehende Abhängigkeiten. Eine ausgehende Abhängigkeit eines Artefakts bedeutet, dass dieses Artefakt abhängig von einem anderen Element ist. Ein Beispiel dafür wäre ein Methodenaufruf, in diesem Fall ist die aufrufende Methode von der aufgerufenen Methode abhängig (beispielsweise von ihrem Rückgabewert). Eine eingehende Abhängigkeit eines Artefakts bedeutet dementsprechend, dass dieses Artefakt von einem anderen gebraucht wird. Jede Abhängigkeit kann also sowohl als eingehend, als auch als ausgehend betrachtet werden, je nachdem von welchem Artefakt aus. Folgende Abbildung demonstriert dies an einem Beispiel.

[7] http://maven.apache.org/

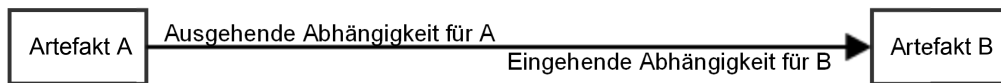


Abbildung 4.1.: Abhängigkeit aus Sicht beider Artefakte

Diese Abhängigkeiten werden von DependencyFinder automatisch für jede gefundene Klasse, jede gefundene Methode und jedes gefundene weitere Artefakt erstellt. Für jede Abhängigkeit gibt es zum Schluss also auch mindestens zwei Einträge, einmal aus Sicht des einen und einmal aus Sicht des anderen Artefakts. Darüber hinaus bietet das Programm noch die Möglichkeit eine Reihe von Metriken bezüglich der gefundenen Abhängigkeiten zu berechnen, diese sind an dieser Stelle aber uninteressant, da das Ergebnis noch optimiert werden soll. Außerdem wird zu jeder Abhängigkeit ein Flag „confirmed=true“ angelegt. Dieses wird gesetzt, wenn das abhängige Artefakt in einer Class-Datei gefunden und analysiert wurde. Referenzen auf andere Artefakte, also beispielsweise auf Klassen die aus einem anderen Projekt stammen und deshalb nicht in der zu untersuchenden Menge der Class-Dateien liegen, werden entsprechend als „confirmed=false“ gekennzeichnet.
Bei der Analyse der Abhängigkeiten mittels DependencyFinder sind die Einschränkungen des Programms und die negative Erkennung in den eigenen Testfällen zu berücksichtigen (siehe Tabelle 4.1). Darüber hinaus wurden noch weitere Abhängigkeitsformen getestet um mögliche weitere nicht erkannte Abhängigkeitsformen zu identifizieren. Insgesamt gibt es sechs Fälle in denen falsche oder gar keine Abhängigkeiten erkannt werden:

1. Als „final“ deklarierte Konstanten einer Klasse erzeugen oft keine Abhängigkeit zu ihrer definierenden Klasse, wenn sie einen primitiven Datentyp oder einen Stringtyp darstellen. Der Javacompiler trägt solche Konstanten in eine globale Tabelle, den *Constant_Pool* ein und löscht dabei die Zugehörigkeit zu einer Klasse [8].
   Ob eine solche Konstante bzw. die dadurch entstehende Abhängigkeit interessant für den Testaufwand ist kann diskutiert werden, hier wird diese Einschränkung jedoch nicht verbessert.

2. Objekte können in Java sowohl Variablen ihres eigenen Typs, als auch dem Typ einer Vaterklasse zugeordnet werden. DependencyFinder überprüft nicht, ob eine Variable eines bestimmten Typs auch andere Objekte aufnehmen könnte, diese anderen Objekte stellen aber natürlich neue Abhängigkeiten dar, die nun nicht gefunden werden können. Listing 4.1 zeigt dies an einem Beispiel:

```
1   class A {
2      public void doSth() { ... }
3   }
4
5   class B extends A {
6      public void doSth() { ... }
7   }
8
9   class X {
10     public void execute(A obj) {
11        obj.doSth(); //ist obj vom Typ A oder B?
12     }
13  }
```

Listing 4.1: Beispiel zur Uneindeutigkeit von Typen der Vaterklasse

Hier erkennt DependencyFinder für Methode *X.execute()* lediglich Abhängigkeiten zu Klasse A, obwohl auch ein Objekt von Klasse B als Parameter übergeben werden könnte.

3. Ein ähnliches Problem ergibt sich mit Interfaces und auch abstrakten Klassen. DependencyFinder analysiert jede Class-Datei einzeln und nutzt keine Informationen einer Klasse zur genaueren Bestimmung in anderen Klassen. Folglich wird zwar der Interfacetyp erkannt, da aber keine Implementierungen bekannt sind können auch keine weiteren Abhängigkeiten erkannt werden. Dies ist eine der größten Einschränkungen des Werkzeugs, da die Erstellung des Abhängigkeitsbaumes an dieser Stelle faktisch abbricht (siehe Abschnitt 4.1.2). Dieses Problem wird in Abschnitt4.2.2 analysiert und nach Möglichkeit eingeschränkt.

4. Reflections werden nicht unterstützt. Da bei Reflections die konkrete Klasse erst zur Laufzeit bekannt ist gibt es keine Chance die Typinformation statisch aus der Class-Datei herauszulesen. Da Reflections sehr komplex und teilweise unmöglich mittels statischer Codeanalyse aufzulösen sind, werden sie in dieser Arbeit nicht weiter betrachtet [1, S.53].

5. Generische Typinformationen werden nicht erkannt. Beispielsweise würde `new ArrayList<GenClass>();` eine Abhängigkeit zu der Klasse „java.util.ArrayList“ erstellen, jedoch nicht zu der generischen Information, in diesem Fall die Klasse „GenClass“. Die Behandlung von Generics wird in dieser Arbeit aufgrund der Einschränkung zur maximalen Java-Version 1.4 aber ohnehin nicht betrachtet [9].

6. Methoden, die eine Klasse `Child` von einer Klasse `Parent` erbt aber nicht überschreibt, existieren in der Class-Datei der Klasse `Child` nicht, hier existiert nur ein Eintrag im Constant_Pool der Class-Datei [8] [7]. Wird eine solche Methode in der Klasse `Child` aufgerufen, so erzeugt DependencyFinder eine Abhängigkeit zu einer Methode in der `Child`-Klasse, die aber eigentlich gar nicht existiert (da ja die Methode der Klasse `Parent` ausgeführt wird). Solche Abhängigkeiten werden mit dem Flag *unconfirmed* gespeichert. Damit zeigt das Programm, dass eine solche Methode aufgerufen wird. Da die Methode aber nicht in der Class-Datei gefunden werden konnte, können auch keine weiteren Abhängigkei-

ten gefunden werden. Dies wird in Abschnitt4.2 genauer erklärt und nachträglich korrigiert.

Außerdem werden generell keine Abhängigkeiten zu lokalen Variablen ermittelt, diese werden beim Kompilieren durch den Compiler eliminiert.

Eine weitere wichtige Einschränkung bei der späteren Auswertung der Ergebnisse des Programms ensteht durch das weglassen von doppelten Abhängigkeiten innerhalb eines Artefakts. Wenn beispielsweise in einer Methode mehrere Objekte einer Klasse erstellt werden, so wird trotzdem nur eine Abhängigkeit ermittelt. [7] Das Gleiche gilt für alle anderen Artefakte, zwei Methodenaufrufe von zwei unterschiedlichen Objekten einer Klasse in der gleichen Methode erzeugen nur eine einzige Abhängigkeit. Diese Einschränkung ist später bei der nachträglichen Optimierung des Abhängigkeitsbaumes wichtig.

### 4.1.2. Erstellung des Baumes

Sind alle Abhängigkeiten ermittelt, so bietet DependencyFinder die Möglichkeit die Ergebnisse in einer XML-Datei zu speichern und damit die Möglichkeit der Weiterverarbeitung. Für diese Arbeit erfolgt die Weiterverarbeitung zunächst ebenfalls mithilfe von DependencyFinder. Da dieser quelloffen ist, können von DependencyFinder gegebene Klassen genutzt werden, um die XML-Datei einzulesen und zunächst in eine Baumstruktur zu transformieren. Dieser Baum enthält nun alle oben aufgezählten Abhängigkeiten und kann mittels eines Visitors [8] abgearbeitet werden. Um ein bestimmtes Artefakt und seine Abhängigkeiten zu bekommen müsste man also den gesamten Baum durchlaufen und nach dem entsprechenden Artefakt suchen. Da dies in der späteren Optimierung öfter der Fall sein wird, wird der Baum zunächst in eine teillineare Datenstruktur entsprechend der Abbildung 4.2 umgewandelt. In der Abbildung nicht mehr dargestellt sind die Knoten C1 - C3. Diese werden ebenfalls in der Liste gespeichert, da sie aber keine Abhängigkeiten besitzen würden sie in der Abbildung lediglich als freie Knoten zu sehen sein.

Innerhalb dieser Struktur unterscheiden wir die einzelnen Artefakte noch nach ihrer Art. Abhängigkeiten von Klassen zu anderen Artefakten werden getrennt von Abhängigkeiten von anderen Artefakten gespeichert. Dies ermöglicht später eine einfachere Untersuchung, beispielsweise in Kapitel 4.1.3. In dieser linearen Struktur lässt sich nun einfach für jedes beliebige Artefakt ein Abhängigkeitsbaum erstellen. Dazu wird das Wurzelelement aus der Liste herausgesucht und alle bekannten *ausgehenden* Abhängigkeiten als Kindknoten an dieses angehängt. Danach sucht man jeden der Kindknoten ebenfalls in der linearen Datenstruktur und verfährt analog. Dies wird wiederholt bis alle Knoten abgearbeitet sind und die Blätter des Baumes selbst nicht mehr über ausgehende Abhängigkeiten verfügen. Nun folgen die einzelnen Detailanalysen mit denen der Abhängigkeitsbaum verbessert werden soll. Für die Methoden zur Verbesserung des Abhängigkeitsbaumes sind allerdings Informationen über die Bezie-

[8]Umsetzung des Visitor-Pattern, siehe http://www.oodesign.com/visitor-pattern.html

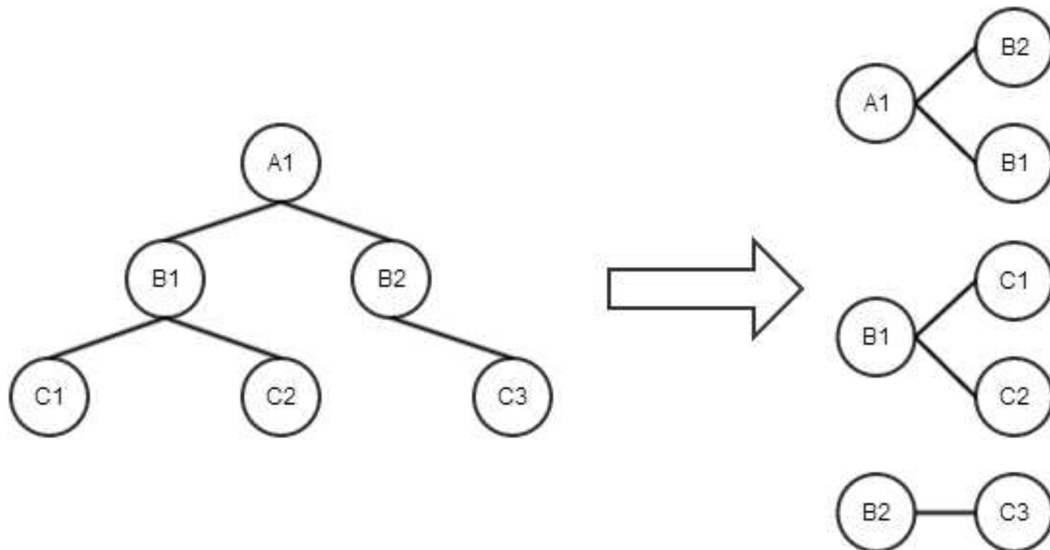


Abbildung 4.2.: Umformung der Baumstruktur

hung der einzelnen Klassen und Interfaces untereinander nötig. Diese Informationen werden ebenfalls aus den Abhängigkeiten ermittelt.

### 4.1.3. Klassenbeziehungen bestimmen

DependencyFinder ermittelt für jede Class-Datei eigenständig Abhängigkeiten, aber keine genauen Informationen über die Beziehung zwischen Klassen oder Interfaces selbst. Diese Hierarchien (oder im Falle von Interfaces Implementierungen) lassen sich jedoch aus den gefundenen Abhängigkeiten auf *Klassenebene* (s. Kapitel 4.1.1) ermitteln.
Es gibt zwei für die Optimierung interessante Informationen:

#### Implementierungen von Interfaces

Zunächst sind sämtliche Implementierungen eines bestimmen Interfaces in allen durchsuchten Class-Dateien interessant. Dazu ist zunächst zu ermitteln, welche der untersuchten Class-Dateien überhaupt Interfaces enthalten, denn DependencyFinder lässt diesbezüglich nur wenig Rückschlüsse zu. Es gibt allerdings eine Eigenschaft von Java, die hier bei der Unterscheidung hilfreich ist: Der Konstruktor. In Java haben alle Klassen Konstruktoren, selbst wenn der Programmierer keinen expliziten Konstruktor angibt. In diesem Fall wird ein Standardkonstruktor erstellt, der auch in die Class-Datei mit kompiliert wird. Dieser Konstruktor ist also ein Artefakt, welches DependencyFinder mit in die Ausgabe schreibt. Untersucht man nun die Abhängigkei-

ten einer Klasse, so kann man die Feature-Abhängigkeiten nach einer „Methode“ untersuchen die den Klassennamen trägt. Diese Abhängigkeit besteht, da bei jeder Instanziierung der Konstruktor automatisch aufgerufen wird. Dabei muss noch beachtet werden, dass der Konstruktor beliebig viele Parameter entgegennehmen kann, denn der Entwickler kann anstatt des impliziten, vom Compiler erstellen Konstruktor natürlich auch einen eigenen definieren, der dann beliebig viele Parameter entgegen nimmt. Wird ein solcher Konstruktor also gefunden, wird dies als Zeichen einer konkreten Klasse interpretiert. Diese Methode ist theoretisch nicht absolut zuverlässig, denn in einem Interface könnten durchaus Methoden deklariert werden, die auf einen Konstruktor schließen lassen würden. Beispielsweise könnte ein Interface `Iface` eine Methode `public void Iface()` vorschreiben. Der Rückgabetyp dient in Java nicht als Kriterium zur Unterscheidung von Methoden, deshalb wird er auch bei der Abhängigkeitsanalyse nicht berücksichtigt. Die Abhängigkeiten für eine Methode `public void Iface()` und einen Konstruktor `public Iface()` sehen also gleich aus. Sollen solche Fälle ebenfalls abgefangen und ausgefiltert werden, so muss die entsprechende Java-Datei durchsucht und ausgewertet werden (beispielsweise per ANTLR). Für diese Arbeit wurde darauf verzichtet, da dass Deklarieren von diesen „Konstruktor“-Methoden in Interfaces gegen allgemeine Konventionen verstößt und in den untersuchten Projekten nicht vorhanden war.

#### Klassenhierarchien bestimmen

Klassenhierarchien werden von DependencyFinder nicht selbst bestimmt. Betrachtet man jedoch die Abhängigkeiten von Klasse zu Klasse, so kann man diese Informationen wiederherstellen, denn auf dieser Ebene stellt jede Abhängigkeit eine Abhängigkeit in der Klassenhierarchie. Die ausgehende Seite der Abhängigkeit erbt dabei die eingehende Seite. Gibt es beispielsweise eine Abhängigkeit von der Klasse `B` zur Klasse `A`, so bedeutet dies, dass `B` von `A` geerbt hat. Gleiches gilt für Interfaces. In der Dokumentation des DependencyFinder wird zwar daraufhin gewiesen das ebenfalls Abhängigkeiten bei der Nutzung einer Klasse als Typ einer Variablen entstehen, dies sind jedoch die sogenannten impliziten Abhängigkeiten (da sie eigentlich die Abhängigkeit eines Features zu einer Klasse darstellen).

Mit diesen zusätzlichen Informationen können nun einige Punkte des Abhängigkeitsbaums verbessert werden.

## 4.2. Verbesserungen im Abhängigkeitsbaum

Der Abhängigkeitsbaum enthält nun alle Informationen, die vom DependencyFinder zur Verfügung gestellt wurden. Durch die in Kapitel 4.1.1 aufgezählten Einschränkungen ist dieser Baum jedoch unvollständig, bzw. teilweise sogar falsch.
Da ein vollständiges Lösen aller dieser Probleme ein erheblicher Aufwand wäre, wurden die einzelnen Punkte auf ihre vermutliche Auswirkung hinsichtlich Genauigkeit und

Vollständigkeit analysiert. Punkt 1 (finale Konstanten einer Klasse) und Punkt 5 (Generische Typen) wurden dabei als vernachlässigbar eingestuft, da Konstanten für die zu erstellenden Testfälle uninteressant sind und Generics durch die gewählte Java-Version 1.4 generell nicht unterstützt werden. Reflections (Punkt 4) werden ebenfalls nicht weiter verfolgt, da eine vollständige Abdeckung aller möglichen Formen von Reflections den Rahmen dieser Arbeit sprengen würde [1, S.53] und lediglich eine Teillösung uninteressant für das Ergebnis dieser Arbeit wäre. Als wichtig wurde Punkt 2 (Erkennung von Kindobjekten in Variablen vom Typ der Vaterklasse), Punkt 3 (tatsächliche Implementierungstypen in Interfaceobjekten) und Punkt 6 (Aufrufe von Methoden einer Vaterklasse) erachtet. In all diesen Fällen endet der Abhängigkeitsbaum aufgrund mangelnder Informationen über die tatsächliche aufgerufene Methode. Punkt 2 und 6 beziehen sich auf Artefakte einer Vaterklasse, Punkt 3 auf Artefakte einer Klasse die ein bestimmtes Interface implementiert hat. Werden die Vater- bzw. Implementierungsklassen gefunden, so kann der Abhängigkeitsbaum weiter entwickelt werden. Die einzelnen Punkte werden noch einmal genauer ab Kapitel 4.2.1 erklärt und ihre Optimierung erläutert. Für die Optimierung selbst sind Informationen über die Beziehung der Klassen und Interfaces untereinander erforderlich. Diese wurden bereits in Kapitel 4.1.3 aus den vorhandenen Informationen von DependencyFinder extrahiert.

### 4.2.1. Aufrufe von Methoden einer Vaterklasse

Folgendes Beispiel soll den Fehler im Abhängigkeitsbaum verdeutlichen:

```
1   class Parent {
2      public void doPapaStuff() {
3         ...
4      }
5   class Child extends Parent {
6      public void executeSomething() {
7         ...
8         this.doPapaStuff();
9      }
10     /* Hier ist keine Methode doPapaStuff() implementiert */
11  }
```

Listing 4.2: Aufrufe von geerbten Methoden

Obwohl ein Objekt vom Typ Child keine Methode `doPapaStuff()` besitzt, kann diese dennoch aufgerufen werden. Das liegt daran, dass die Java-VM diese Methode erst zur Laufzeit auflöst und bis dahin lediglich ein Symbol im Constant_Pool der Class-Datei speichert [7] [8].
DependencyFinder erkennt nun in der Methode `executeSomething()` einen Aufruf einer Methode `Child.doPapaStuff()`, findet aber keine Methode mit diesem Namen. Stattdessen wird eine Abhängigkeit mit Namen `Child.doPapaStuff()` angelegt und mit dem Flag *unconfirmed* versehen. Dieses Flag signalisiert dem Benutzer, dass hier zwar auf eine Methode referenziert werden soll, diese Methode vom

DependencyFinder jedoch nicht gefunden wurde. Der Grund dafür ist egal, auch bei fehlenden Class-Dateien eines Typs auf den referenziert wird ist das *unconfirmed* -Flag gesetzt (beispielsweise sind quasi immer alle Abhängigkeiten zu Features in den Packages `java` *unconfirmed*, denn deren Class-Dateien befinden sich quasi nie im Projektverzeichnis).

Wird beim Erstellen des Abhängigkeitsbaums ein Feature angefordert, bei welchem das *unconfirmed* -Flag gesetzt ist, so gibt es dafür also zwei Möglichkeiten: Entweder das Feature ist tatsächlich nicht auffindbar (weil beispielsweise in einer externen, nicht untersuchten Bibliothek) oder es handelt sich um oben beschriebene Einschränkung. Da im ersten Fall jeder Versuch der Verbesserung automatisch fehlschlägt wird zwischen den Fällen nicht unterschieden.
Zunächst wird ermittelt, ob die Klasse zu welcher das Feature gehört von einer anderen Klasse geerbt hat. Dazu werden die oben beschriebenen Hierarchieinformationen ausgewertet. Ist eine Vaterklasse gefunden worden, so wird versucht das gleiche Feature in dieser zu finden. Gleiches Feature bedeutet hier, dass der exakt gleiche, vollqualifizierende Name inklusiver gleicher etwaiger Parameter in der gleichen Reihenfolge in der Vaterklasse gesucht wird. Wird kein solches Feature gefunden, oder ist das gefundene Feature wieder mit dem Flag *unconfirmed* versehen, so wird die gefundene Vaterklasse ihrerseits wieder auf eine Vaterklasse untersucht und die Untersuchung dort wiederholt. Das wird solange gemacht bis keine Vaterklasse mehr vorhanden ist oder ein entsprechendes Feature in einer der Vaterklassen gefunden wurde (und dieses nicht mit dem *unconfirmed*Flag versehen ist). Im negativen Ergebnis (alle Vaterklassen erfolglos durchsucht) kann der Abhängigkeitsbaum an dieser Stelle nicht verbessert werden. Im positiven Ergebnis kann das neue gefundene Feature nun wieder nach dem bekannten Muster in den Baum eingefügt werden und die neuen Abhängigkeiten ermittelt werden.
An dieser Stelle kann man sicher sein dass kein falsches Element eingefügt wird, solange die gleichen Regeln wie beim Überschreiben eines Features in Java eingehalten werden und die Hierarchieinformationen korrekt sind. Andernfalls könnte die die JVM von Java selbst nicht mehr unterscheiden ob ein Element überschrieben wurde oder nicht.

### 4.2.2. Auflösen von Interfacetypen

Als zweiten Schritt der nachträglichen Optimierung wird versucht Interfacetypen auf konkrete Implementierungstypen zu reduzieren. Interfaces können als Typ in Methodensignaturen oder Member- bzw. Klassenvariablen auftauchen. Da es keine konkrete Implementierung der im Interface definierten Methoden gibt, kann DependencyFinder auch keine Abhängigkeiten aus einem Methodenaufruf dieser Interfacetypen generieren. Je nach Implementierungstyp der zur Laufzeit aufgerufen wird, unterscheiden sich außerdem die aufgerufenen Abhängigkeiten. Auch wenn Interfacetypen die Möglichkeit bieten alle Objekte von implementierenden Klassen aufzunehmen, so kann es dennoch sein das für einen bestimmten Ablaufpfad des Programms immer nur ein konkreter Implementierungstyp genutzt wird. Mit der Erstellung eines Abhängigkeits-

baumes für ein bestimmtes Artefakt ist der Aufrufweg zu solch einer Methode eindeutig bestimmt. Selbst wenn die Methode innerhalb des Baumes mehrmals auftauchen sollte, gibt es für jeden Aufruf nur den Weg über die direkten Vorgänger bis hin zur Wurzel. Dieser Pfad kann nun untersucht werden. Dabei wird zwischen Interfacetypen in Signaturen von Methoden, und Interfacetypen als Klassen- bzw. Membervariablen unterschieden.

**Interfacetypen in Methodensignaturen**

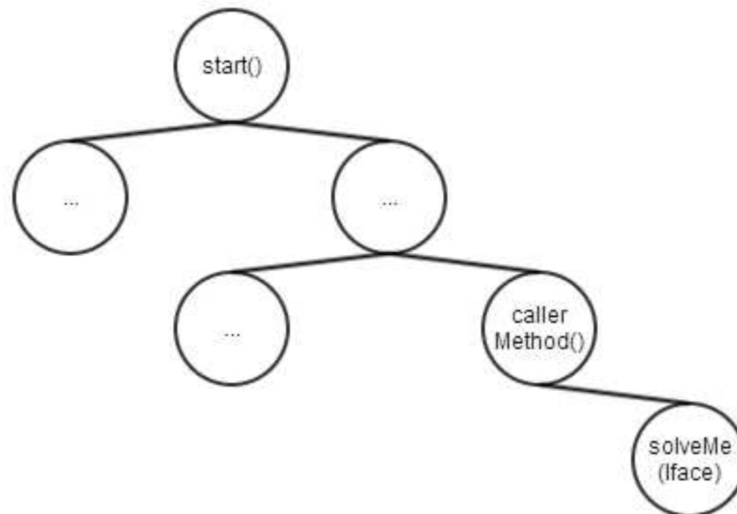


Abbildung 4.3.: Aufrufpfad einer Methode

Gegeben sei der Teilaufrufbaum in Abbildung 4.3. Der Einfachheit wegen enthält dieser Ausschnitt nur Methoden, keine anderen Artefakte als Abhängigkeit. Das Wurzelelement ist hier die Methode `start()`, ausgehend von dieser werden einige andere Methoden aufgerufen, bis irgendwann die Methode `solveMe()` erreicht wird. Diese erwartet einen Parameter vom Typ `Iface`, bei welchem es sich in diesem Beispiel um ein Interface handeln soll. Die Methode `solveMe()` selbst erlaubt keinerlei Rückschlüsse auf den tatsächlichen Implementierungstypen. In der direkt davor gelegenen Methode `callerMethod()` gibt es jedoch einen Aufruf der Methode, in welchem ein Parameter übergeben werden muss. Dieser Parameter ist ein Java-Ausdruck, welchen man auswerten und möglicherweise zurückverfolgen kann. Bei diesem Ausdruck kann es sich um eine Variable, einen Funktionsaufruf oder einen `new`-Operator (bzw. einen String oder primitiven Typ wie ein Integer oder ein Double) handeln. Das Zurückverfolgen dieser Ausdrücke selbst wird in 4.3 beschrieben. Wird ein konkreter Typ durch dieses Verfahren ermittelt, so wird als erstes überprüft ob dieser Typ tatsächlich das entsprechende Interface implementiert. Dazu werden die Daten aus 4.1.3 genutzt. Implementiert der gefundene Typ das untersuchte Interface, so wird davon ausgegangen das eine *mögliche* konkrete Implementierung gefunden wurde. Durch die Einschrän-

kungen bei der Zurückverfolgung kann zum einen keine hundertprozentige Korrektheit gewährleistet werden, zum anderen ist es auch möglich, dass mehrere mögliche Typen für ein Interfaceparameter gefunden werden. Ein Beispiel dafür ist das Factory-Pattern. Wird eine Zuweisung von einer Factory zu einer Variable gefunden, so sind prinzipiell alle Typen in dieser Factory mögliche Kandidaten für den gesuchten Implementierungstyp.

#### Interfacetypen in Member- oder Klassenvariablen

Member- oder Klassenvariablen können ebenfalls vom Typ eines Interfaces sein. Wird diese Variable in einer Methode genutzt oder von einer Methode zurückgegeben und dann genutzt, so entsteht wieder eine Abhängigkeit eines Interfaces. Um dieses Interface zu verfolgen müssen eigentlich zwei Zuweisungsmethoden überprüft werden: Das direkte Zuweisen, beispielsweise mittels `obj.var = type;` und das Zuweisen über eine Methode. In dieser Arbeit wird zur Vereinfachung nur die zweite Variante überprüft. Dazu werden zunächst alle Methoden der Klasse ermittelt, die der Klassen-/Membervariablen etwas zuweisen. In jeder gefundenen Methode wird überprüft, ob es einen Parameter vom Typ der Interfacevariablen gibt und ob dieser Parameter letztlich der Variablen zugewiesen wird. Am einfachsten ist dies bei einfachen Getter-Methoden umgesetzt. Alle Methoden auf die diese Kriterien zutreffen werden gespeichert und der abhängige Parameter wird markiert. In den aufrufenden Methoden des Abhängigkeitsbaumes wird dann nach einem Aufruf einer solchen Methode gesucht. Gibt es einen Treffer, so wird der markierte Parameter wie oben bereits erklärt verfolgt. Für ausführliche Erklärungen zur Verfolgung des Parameter im Javacode sei wieder auf 4.3 verwiesen.

#### Einschränkungen

Beide Methoden bieten keine hundertprozentige Korrektheit, auch nicht falls ein Typ gefunden wird. Es ist möglich, dass in der aufrufenden Methode mehr als ein Objekt von der Klasse mit der Interfacevariablen erstellt wird. Wie in 4.1.1 erläutert, werden in diesem Fall die Abhängigkeiten zusammengefasst. Sollten ähnliche Methoden von beiden Objekten aufgerufen werden, so gibt es keine Möglichkeit die Abhängigkeit im Abhängigkeitsbaum einem dieser Objekte zuzuweisen. Das wird dann problematisch, wenn beide Objekte unterschiedliche Implementierungstypen für die Interfacevariable nutzen. In einem solchen Fall werden die Implementierungstypen *beider* Objekte als Alternativen in den Abhängigkeitsbaum eingepflegt. Für die Abschätzung des Testbedarfs kann es zu einigen falschen Einträgen kommen, wie folgendes Beispiel zeigt:

```
1  ...
2  ClassType obj1 = new ClassType();
3  ClassType obj2 = new ClassType();
4  obj1.setInterfaceVariable(new Impl1());
5  obj2.setInterfaceVariable(new Impl2());
6  //uses Interfacemethod i1()
7  obj1.methodThatUsesInterfaceVariable();
8  //uses Interfacemethod i2()
9  obj2.anotherMethodThatUsesInterfaceVariable();
10 ...
```

Listing 4.3: Doppelte Erkennung von Implementierungstypen

`ClassType` sei der Name der Klasse, die eine Variable mit dem Typen eines Interfaces hat. Diese Variable kann über `setInterfaceVariable()` gesetzt werden und wird in den Methoden `methodThatUsesInterfaceVariable()` und `anotherMethodThatUsesInterfaceVariable()` genutzt um jeweils *verschiedene* Methoden des Interfaces aufzurufen (`i1()` bzw `i2()`). In diesem Codebeispiel wird die erste Methode nur von `obj1` aufgerufen, nicht von `obj2`. Gleiches gilt umgekehrt für die zweite Methode. Folglich müsste vom Typen `Impl1()` eigentlich nur die Methode `i1()` als Abhängigkeit hinzugefügt werden, entsprechend für `Impl2()` nur die Methode `i2()`. Bei der Analyse dieser Arbeit werden jedoch sowohl `Impl1` als auch `Impl2` als mögliche Implementierungstypen gefunden und *alle* Abhängigkeiten für jeden dieser Typen als Alternativen in den Abhängigkeitsbaum eingetragen. Es ergeben sich hier also die zusätzlichen Abhängigkeiten der Methode `i2()` aus der Klasse `Impl1` und der Methode `i1()` aus der Klasse `Impl2`. Mit einer genaueren Analyse des Java-Codes könnten sich diese falschen Abhängigkeiten herausfiltern lassen.
Die Konstruktion des obigen Listings erfordert allerdings sehr spezielle Umstände und dürfte eher selten auftreten. Ähnliche Fälle lassen sich beispielsweise mittels des Operators `instanceof` generieren, dieser wird durch die Codeanalyse ebenfalls nicht ausgewertet.

## 4.3. Parsing von Java-Quelltext

Um die Typen einer Variable wie in 4.2.2 aufzulösen muss der Quelltext bis zu einem gewissen Maße so interpretiert werden, wie es später auch durch den Compiler geschieht. Eine gleichwertige Erkennung würde einen unverhältnismäßig hohen Aufwand bedeuten. Deswegen werden in der für diese Arbeit implementierten Variante nur einige Teilbereiche unterstützt, die einen Großteil aller möglichen Fälle einer Typzuweisung erkennen sollen. Sämtliche gemachten Einschränkungen dienen der Reduzierung der Komplexität. Dieser Abschnitt gibt einen Überblick über die Analysemethoden, die Nutzung dieser Methoden ist in 4.2.2 beschrieben.
Da es sich bei der Quelltextanalyse im Wesentlichen um Texterkennung handelt, müssen einige Regeln aufgestellt werden um korrekte Ergebnisse zu bekommen. Der ei-

gentliche Java-Code wird zunächst mittels ANTLR geparst und in einen AST gewandelt. Dadurch ist sichergestellt, dass der Quelltext gültiger Java-Code ist. Nun müssen Regeln für einzelne Teile des Quelltextes aufgestellt werden, um die gesuchten Werte ermitteln zu können. Unterstützt werden folgende Ausdrücke:

**Liste 1:** Unterstütze Java-Konstruke

1. Variablen
2. Die Erstellung von Objekten mittels des `new` Operators
3. Casts
4. (statische) Funktionsaufrufe, insbesondere die Erkennung von Rückgabewerten
5. Ternäre Operatoren
6. Primitive Datentypen
7. Erweiterte For-Schleifen

Konkrete Typen werden dabei nur aus den Punkten 2, 3, 6 und 7 gewonnen. Der Rest wird weiterverfolgt bis einer der genannten Punkte erreicht wird.

### 4.3.1. Genereller Ablauf

Die Verfolgung im Quelltext besteht aus einem rekursiven Auflösen von einzelnen Ausdrücken im Quelltext. Angenommen, ein Parameter einer Methode `traceMe()` soll verfolgt werden (beispielsweise weil es sich um einen Interfacetypen handelt), dann wird zunächst die Java-Datei der *aufrufenden* Klasse mittels ANTLR geparst. Im Quelltext wird die aufrufende Methode gesucht, und in dieser Methode der Name „traceMe“. Ist dieser Name gefunden, so werden die mittels Kommata getrennten Parameter einzeln extrahiert und der Parameter, der konkretisiert werden soll untersucht.
Zunächst erfolgt eine Klassifizierung in eine der in Liste 1 aufgelisteten Arten. Anhand dieser Klassifizierung wird nun anhand der ab 4.3.2 beschriebenen Vorgehensweise versucht, einen konkreten Typen zu finden.

### 4.3.2. Variablen

Variablen dienen in Quelltext als Platzhalter von Zwischenergebnissen in Form von primitiven Daten oder Objekten. Wird eine Variable als Parameter verwendet, so wurde ihr an einer vorherigen Stelle im Code ein Wert in der Form `variable = x;` zugewiesen. Die Variable `x` kann dabei selbst einer der in Liste 1 aufgezählten Ausdrücke sein, der dann seinerseits weiterverfolgt wird. Zu unterscheiden sind dabei lokale Variablen (inklusive übergebene Parameter) und Member-/Klassenvariablen. Wird eine Variable gefunden, so wird also entweder der vorherige Teil der Methode nach einer Zuweisung zu einer Variablen mit diesem Namen gesucht (lokale Variable), oder es werden wie in 4.2.2 beschrieben zuweisende Methoden gesucht und in der aufrufenden Methode nach einem Vorkommen dieser Methodennamen gesucht (Member-/Klassenvariable).

Der zuweisende Wert wird wieder klassifiziert und weiter ausgewertet.
Wird einer Variable zweimal etwas zugewiesen (die erste Zuweisung wird also überschrieben), so wird dies nicht berücksichtigt. In diesem Fall werden beide Zuweisungen einzeln betrachtet und aufgelöst, es können also auch zwei konkrete Typen ermittelt werden.

### 4.3.3. Erstellung von Objekten mittels `new`-Operator

Wird ein solcher Operator gefunden (beispielsweise bei der Zuweisung zu einer Variablen), so kann ein konkreter Typ ermittelt werden. Dabei wird der Klassenname hinter dem Schlüsselwort „new“ ermittelt. Diese Klasse stellt den vermuteten Typ dar. An dieser Stelle endet damit auch die Analyse für den betrachteten Ausdruck.

### 4.3.4. Casts

Wird ein Wert an eine Variable übergeben oder als Parameter in einem Methodenaufruf übergeben, so kann dieser gecastet werden, also in einen expliziten Typ umgewandelt werden. Wird ein solcher Cast entdeckt, wird der Zieltyp des Casts als konkreter Typ übernommen. Es spielt keine Rolle von welchem Typ gecastet wurde, darum wird an dieser Stelle die Verfolgung im Code ebenfalls abgebrochen.

### 4.3.5. Funktionsaufrufe

Java unterstützt Polymorphie und das Überladen von Funktionen. Deswegen muss neben dem Namen auch die Anzahl der Parameter und deren jeweiliger Typ bestimmt werden. Dazu wird jeder Parameter extrahiert und einzeln analysiert. Sollte sich einer der Parameter nicht auflösen lassen, so wird die Analyse des Funktionsaufrufes ebenfalls abgebrochen, da nicht sichergestellt werden kann das die richtige Methode gefunden wird.
Konnten alle Parameter aufgelöst werden gibt es drei verschiedene zu betrachtende Funktionsaufrufe:

1. Funktionsaufrufe in der gleichen Klasse
2. Funktionsaufrufe von Objekten
3. Statische Funktionsaufrufe

#### Funktionsaufrufe in der gleichen Klasse

Aufrufe innerhalb einer Klasse entsprechen einem Sprung zum Ende der aufgerufenen Methode. Dort wird der `return` Ausdruck untersucht, denn nach dem `return` kann ebenfalls wieder einer der von dieser Analyse unterstützen Ausdrücke stehen (also eine Variable, ein Funktionsaufruf, etc). Dieser Ausdruck wird ebenfalls verfolgt. Diese

Art von Funktionsaufrufen wird durch das Fehlen einer Objektvariablen erkannt (der Spezialfall `this` wird nachfolgend erläutert).

**Funktionsaufrufe von Objekten:**

Um Funktionsaufrufe von Objekten aufzulösen muss neben der Methodensignatur auch der Typ des Objektes bekannt sein. Darum besteht der erste Schritt hier in der Auflösung der Objektvariablen. Zunächst wird überprüft, ob es sich bei der Variablen um das Schlüsselwort `this` handelt. In diesem Fall wird die Abarbeitung wie bei einem Funktionsaufruf in der gleichen Klasse fortgeführt. Andernfalls wird der Name der Objektvariablen extrahiert und wie eine normale Variable selbst verfolgt. Zu erwarten ist an einer Stelle eine Zuweisung mittels des `new`-Operators. Konnte die Objektvariable keinem *eindeutigen* Typen zugewiesen werden (also auch wenn die Analyse zwei verschiedene in Frage kommenden Typen liefert), so wird die Analyse hier abgebrochen. Andernfalls bestände die Chance eine Methode der falschen Klasse zu untersuchen, alle von dieser Methode ausgehenden Abhängigkeiten wären dann falsch.
Konnte der Typ der Objektvariablen hingegen bestimmt werden, so wird der vollqualifizierende Name bestimmt (siehe 4.3.9), die entsprechende Klasse geladen und die gefundene Methode ab dem `return` untersucht.
Listing 4.4 zeigt ein Beispiel für einen solchen Aufruf in Zeile 7. Um den Funktionsaufruf richtig verfolgen zu können muss zunächst der Typ von `object` bestimmt werden (nicht nötig wenn statt `object` ein `this` die Objektvariable repräsentieren würde). In diesem Beispiel ist die Auflösung trivial. Ist der Typ von `object` bekannt, so wird eine entsprechende Klasse im eigenen Package `se.unihannover.de` und in den importierten Packages `sra.unihannover.de` gesucht. Nur wenn eine passende Klasse gefunden wurde wird die Methode `methodA()` in dieser untersucht und die `return`-Anweisung analysiert.

```
1  class ClassA() {
2    package se.unihannover.de;
3    import sra.unihannover.de.*;
4
5    public void methodA() {
6        ObjType object = new ObjType();
7        SolveMe var = object.methodA();
8        ...
9     }
10 }
```

Listing 4.4: Funktionsaufruf von einem Objekt

**Statische Funktionsaufrufe:**

Die Untersuchung auf einen statischen Funktionsaufruf findet eigentlich während der Untersuchung in 4.3.5 statt. Wurde für die Objektvariable kein Typ gefunden, so be-

steht die Chance eines statischen Aufrufs. In diesem Fall wird der Name der Objektvariablen extrahiert und die importierten Packages auf einen passenden Typen hin untersucht (siehe dazu 4.3.9). Im Beispiel von Listing 4.4 würde also nach einer Klasse `object` gesucht werden, sollte die Auflösung der Variablen scheitern. Es kann hier keinen Konflikt mit der Variante aus Kapitel 4.3.5 geben, denn eine Variable kann nicht den gleichen Namen haben wie ein definierter Typ. Wurde eine passende Klasse gefunden, so wird diese geladen und die entsprechende Methode wieder ab dem `return` Ausdruck untersucht.

Generell gilt: Ist eine Methode gefunden und wird untersucht, so wird explizit nicht der im Quelltext angegebene Rückgabetyp in der Methodensignatur verwendet. Diese Lösung wäre zwar einfach und hätte immer ein Ergebnis, aber der angegebene Rückgabetyp einer Methode entspricht nicht immer dem tatsächlich zurückgegebenen Typen (Beispiel: Interface vs Objekt eines Implementierungstyp, Vaterklasse vs Kindobjekt).

### 4.3.6. Ternäre Operatoren

Obwohl eigentlich keine Kontrollstrukturen in dieser Analyse berücksichtig werden, wurde für ternäre Operatoren eine eigene Klassifikation erstellt. Der Grund waren Probleme bei der Erkennung von Casts, bei denen fälschlicherweise auch viele ternäre Operatoren eingeordnet wurden. Eine eigene Erkennung löste dieses Problem. Trotz der eigenen Klassifikation wird kein Kontrollfluss ausgewertet, das heißt es werden *immer beide Ausdrücke* (sowohl für eine wahre als auch eine falsche Bedingung) untersucht. Diese Ausdrücke gehören wieder einem der in Liste 1 gennanten Punkte an, die Analyse geht hier also weiter. Listing 4.5 zeigt dies an einem Beispiel, sowohl die Methode `methodA()` als auch die Methode `methodB()` werden nacheinander analysiert. Ergibt eine der Analysen einen konkreten Typ, so wird dieser der Variablen `var` zugeordnet. Ergeben beide Analysen einen konkreten Typ, so werden auch beide Typen nacheinander der Variablen `var` zugeordnet. Sämtliche weitere Analyseschritte, beispielsweise der Aufruf der Methode `m()` werden dann für beide Typen durchgeführt.

```
1  public void functionA(String cond, ObjectType obj) {
2      SolveMe var;
3      var = (cond.equals("true") ? obj.methodA() : obj.methodB());
4      var.m();
5  }
```

Listing 4.5: Auflösung eines ternären Operators

### 4.3.7. Primitive Datentypen

Primitive Datentypen werden ohne `new` Operator oder Cast erzeugt, sie werden bei der Deklaration erkannt. Zu den primitiven Datentypen wird in dieser Arbeit auch der Typ

`String` gezählt, denn auch dieser kann ebenfalls mittels einfacher Zuweisung eines Strings erzeugt werden. Wird ein primitiver Datentyp erkannt, so endet die Analyse für den untersuchten Ausdruck.

### 4.3.8. Erweiterte `For`-Schleife

Java bietet die Möglichkeit mit dem `for` Operator eine verkürzte Schreibweise zu nutzen um schnell über Datenstrukturen zu iterieren[9]. Dazu wird anstatt der üblichen Signatur ein Iterationsobjekt initialisiert, welches dann für jeden Durchgang ein entsprechendes Objekt aus einer Liste (oder anderen iterierbaren Struktur) repräsentiert. Aufgrund des Aufbaus des AST ist es notwendig diesen Fall gesondert zu betrachten. Die Deklaration des Iterationsobjektes kann mit den anderen Klassifiezierungen nicht ermittelt werden. Da die Variable außerdem nie mittels `new` initialisiert wird, wird in diesem Fall ausnahmsweise der bei der Deklaration angegebene Typ der Variablen übernommen.

### 4.3.9. Überprüfung der im Quelltext gefundenen Typen

Wird mittels einer der Punkte aus Liste  1 ein konkreter Typ entdeckt, so wird dieser auf seine Existenz überprüft. Dazu werden beim Laden einer Klasse sämtliche Imports und die `package` Anweisung (sofern vorhanden) für die jeweilige Klasse ausgelesen. Da sich die Java-Dateien innerhalb einer den Packages entsprechenden Ordnerstruktur befinden müssen, kann die Existenz eines Typs bestimmt werden indem alle Dateien in Ordnern der importierten Packages überprüft werden. Befindet sich in einem Ordner eine Datei namens „%typ%.java“ (wobei %typ% den gefundenen Typ darstellt), dann ist dies der im Quelltext referenzierte Typ.

## 4.4. Metriken anhand des Abhängigkeitsbaum berechnen

Der Abhängigkeitsbaum selbst ist für eine Aussage zum Testaufwand nur bedingt geeignet. Naturgemäß wird dieser sehr schnell groß und unübersichtlich. Zur Abschätzung des Testaufwands bietet es sich deshalb an diverse Metriken mittels des Baumes zu berechnen, beispielsweise die Größe des Baumes. In dieser Arbeit wurden folgende Metriken implementiert:

1. Größe des Abhängigkeitsbaumes
2. Anzahl der verschiedenen Klassen im Baum
3. Anzahl der Methodenaufrufe im Baum
4. Testabdeckung durch vorhandene Tests

[9]Diese Datenstrukturen müssen das Interface `Iterable` implementieren, siehe auch http://docs.oracle.com/javase/1.5.0/docs/api/java/lang/Iterable.html

Die ersten drei Metriken sind relativ einfach und geben einen schnellen Überblick über die Größe des Baumes und den zu erwartenden Aufwand. Da die Ergebnisse als XML-Struktur gespeichert werden sind außerdem Informationen wie die maximale Tiefe des Baumes oder die Anzahl der Abhängigkeiten bis zu einer bestimmten Ebene leicht zu ermitteln.
Die vierte Metrik, „Testabdeckung durch vorhandene Tests“, beschreibt die relative Abdeckung eines Knotens im Baum inklusive seiner Kinder durch bereits von früheren Aufgaben erstellten Tests. Die Dirk Rossmann GmbH hat für viele Elemente der untersuchten Software bereits Testfälle geschrieben. Da die Abschätzung des Testaufwandes wesentlich von bereits vorhandenen Testfällen beeinflusst wird, werden diese Testfälle untersucht[10] und alle Elemente des Abhängigkeitsbaumes auf ein eventuelles Vorkommen in diesen Testfällen untersucht. Ist ein Knoten im Abhängigkeitsbaum beispielsweise ein Blatt und kommt in den Testfällen vor, so besitzt er eine Testabdeckung von 100%. Angenommen der Vorgänger dieses Knoten ist jedoch nicht getestet und hat ein weiteres Kind mit einer Testabdeckung von 0%, so ist die relative Testabdeckung des Vorgängers

$$\frac{1 \text{ [getestete Knoten]}}{3 \text{ [alle Knoten]}} = 33\%.$$

Abbildung 4.4 vedeutlicht dies an einem Beispiel mit einem vereinfachten Abhängigkeitsbaum. Blaue Knoten signalisieren das Vorhandensein in den Testfällen, rote Knoten sind ungetestet.

Je nach Bedarf können diese Metriken leicht angepasst werden. Außerdem könnten neben diesen aus den Ergebnissen abgeleiteten Abschätzungen auch Metriken während des Erstellen des Baumes berechnet werden. Damit ließen sich dann beispielsweise Informationen wie „der Pfad mit dem höchsten Verzweigungsgrad“ bestimmen.
Nach Erstellung des Abhängigkeitsbaumes werden diese Testfälle nun durchsucht und die gefundenen Softwareartefakte aller Klassen mitprotokolliert. Die gefundenen Aufrufe werden mit dem Abhängigkeitsbaum des eigentlichen Projektes (wieder für ein gegebenes Wurzelelement) abgeglichen, um so eine relative Testabdeckung zu erhalten. Da die Testfälle nicht durchgehend Standard-JUnit-Tests sind, wird bei einem gefundenen Methodenaufruf vereinfachend davon ausgegangen, dass das Ergebnis dieses Aufrufs überprüft wird und das entsprechende Artefakt als getestet markiert werden kann. Der erweiterte Abhängigkeitsbaum wird dann inklusive der Informationen über die Testabdeckung mit den bereits vorhandenen Tests ausgegeben.

[10] Testfälle automatisch zu interpretieren kann ähnlich komplex werden wie die Analyse selbst, da nicht jedes Ergebnis eines jeden Methodenaufruf auch tatsächlich getestet wird. In Absprache mit der Dirk Rossmann GmbH wurde deshalb folgende Vereinfachung angenommen: Alle Artefakte die in Testfällen gefunden worden sind, werden als getestet angesehen. Dabei zählt allein der Aufruf des Artefakts, was mit dem Ergebnis passiert wird nicht beachtet.

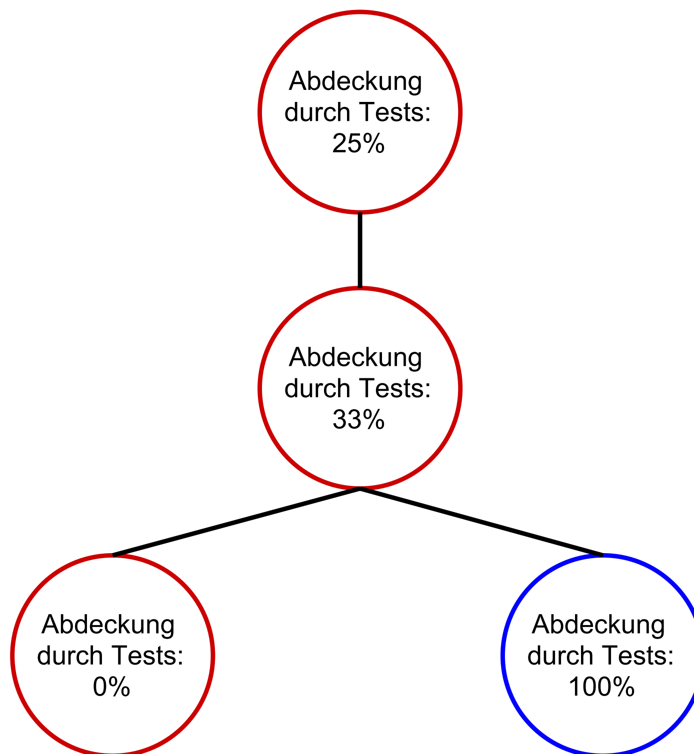


Abbildung 4.4.: Testabdeckung im Abhängigkeitsbaum

# 5. Prototypische Implementierung

Die Implementierung der Konzepte aus Kapitel 3 und 4 wurde prototypisch in Java vorgenommen. Dabei wurden die Konzepte aus wie beschrieben umgesetzt. Das Parsing des Java-Codes in Kapitel 4.3 wurde mithilfe des Programms ANTLR[1] und einer gegebenen Java-Grammatik realisiert. ANTLR erzeugt zunächst einen Abstract Syntax Tree (AST)[2] des zu untersuchenden Quelltextes, der dann durchlaufen wird. Die Umsetzung der Regeln aus Kapitel 4.3 werden dann mithilfe verschiedener Visitors[3] übernommen.

## 5.1. Speicher- und Laufzeitprobleme

Bei einigen Tests des Prototyps zeigten sich Schwierigkeiten bezüglich der Laufzeit und des Speicherverbrauches, wenn einige sehr große Abhängigkeitsbäume erstellt werden sollten. Diese Bäume besaßen schon vor der Optimierung zum Teil deutlich mehr als 500.000 Knoten, so dass ein einzelner Durchlauf eines dieser Bäume (sofern er überhaupt erfolgreich war) bis zu 30 Minuten dauern konnte. Da sich die Größe der Bäume nicht einfach minimieren lässt (die Abhängigkeiten existieren ja tatsächlich) und für die spätere Evaluation mehrere tausend dieser Bäume erstellt und optimiert werden sollten, ist in dem Prototyp dieser Arbeit die im Folgenden beschriebene Vorgehensweise implementiert.

### 5.1.1. Optimierung des Prototypen hinsichtlich des Evaluationsziels

Um die Größe der Bäume verringern zu können aber gleichzeitig zur Evaluation passende Ergebnisse zu erhalten, musste an dieser Stelle die Art der Evaluation berücksichtigt werden. Interessant wären sowohl die Anzahl der Teilbäume im Abhängigkeitsbaum die verbessert werden konnten, als auch eine Einschätzung zur Qualität einer einzelnen Verbesserung. Im ersten Fall hätte der Baum komplett entwickelt werden müssen, Teilknoten die verbessert werden konnten müssten jedoch nicht weiter untersucht werden und der Baum könnte an dieser Stelle enden. Bei einer Aussage zur Qualität der Verbesserungen müssten nicht der gesamte Baum untersucht werden, dafür müssten die einzelnen verbesserten Teilbäume aber vollständig entwickelt werden.

---

[1] http://www.antlr.org

[2] http://c2.com/cgi/wiki?AbstractSyntaxTree

[3] Umsetzung des Visitor-Patterns

Da die Voraussetzungen für die in dieser Arbeit beschriebenen Methoden recht spezifisch sind (siehe 4) ist es das Ziel der Evaluation die gemachten Verbesserungen zu beurteilen, die Häufigkeit der Verbesserungen wird nur sekundär betrachtet.

### 5.1.2. Limitierung der Häufigkeit einer Abhängigkeit

Einzelne Bäume, insbesondere solche mit Methoden als Wurzelelement die früh von der Einstiegsmethode aufgerufen werden, besaßen mehrere 100.000 Knoten. Dieser großen Anzahl an Abhängigkeiten standen meist nur wenige Tausend verschiedene Softwareartefakte gegenüber. Die Abhängigkeitsbäume hatten also an mehreren Stellen die gleichen Abhängigkeiten in sich. Ohne die in Kapitel 4 beschriebenen Methoden entwickeln sich aus diesen Abhängigkeiten immer wieder dieselben Unterbäume, wie in Abbildung 5.1 mit blauen Knoten dargestellt.

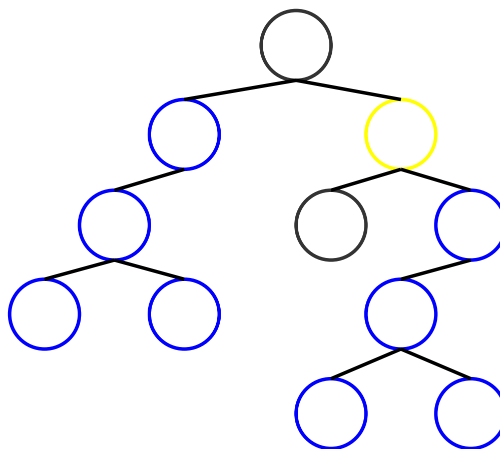

Abbildung 5.1.: Gleiche Teilbäume in einem Abhängigkeitsbaum

Zunächst wurde ein Ansatz zur Ersetzung dieser doppelten Unterbäume durch einzelne Knoten mit entsprechenden Informationen über den ersetzten Teilbaum untersucht. Dieser Ansatz geriet jedoch in Konflikt mit anderen Teilen des Programms, beispielsweise der Zyklenerkennung. Es zeigte sich, dass die Idee zu komplex war um sie in der verbliebenen Zeit umzusetzen. Deshalb wurde für die Implementierung folgende Einschränkung hinzugefügt: Der Abhängigkeitsbaum enthält bei seiner erstmaligen Erstellung jede Abhängigkeit nur ein einziges mal. Dadurch verringert sich die Größe des Baumes enorm. Um die Größe der verbesserten Teilbäume trotzdem richtig einschätzen zu können wird jeder *verbesserte* Teilbaum von dieser Einschränkung ausgenommen, in diesem verbesserten Teilbaum dürfen also alle Abhängigkeiten auftreten, unabhängig davon ob sie schon einmal an anderer Stelle des gesamten Abhängigkeitsbaumes aufgetreten sind. Diese Einschränkungen lassen sich über die Konfiguration des Programms jedoch ein- und ausschalten.

### 5.1.3. Auswirkungen auf die Ergebnisse

Diese Minimierung hat keine Auswirkungen auf die in 4.2.1 beschriebene Methode zur Erkennung von nicht-überschriebenen Artefakten einer Vaterklasse, da die dort gewonnenen Verbesserungen unabhängig vom Abhängigkeitsbaum gewonnen wurden. Es ist also egal ob ein Teilbaum einmal oder mehrmals im Abhängigkeitsbaum steht, die Verbesserung durch diese Methode wäre an beiden Stellen identisch. Durch das Wegstreichen aller zusätzlichen Vorkommen dieses Teilbaumes verringert sich die Anzahl der gemachten Verbesserungen, die Qualität einer einzelnen Verbesserung ändert sich jedoch nicht.
Für die Methode aus 4.2.2 zur Erkennung von Interfacetypen gilt im Grunde das Gleiche, nämlich dass sich die Anzahl der Verbesserungen verringert, die Qualität der durchgeführten Verbesserungen aber gleich bleibt. Ob an einer Stelle im Abhängigkeitsbaum eine Verbesserung stattfinden kann hängt stark vom gewählten Pfad ab. Ein Interfacetyp in einem Teilbaum `X` könnte in einem Pfad des Abhängigkeitsbaums verbessert werden, weil die durch die Vorgängerknoten repräsentierten Methoden passende Zuweisungen enthalten. In einem anderen Zweig `Y` gäbe es aber andere Vorgängerknoten, die Verbesserung kann an dieser Stelle also unter Umständen nicht angewandt werden. Abbildung 5.1 zeigt ein Beispiel dazu, im linken blaugefärbten Teilbaum gibt es lediglich die Wurzel als Vorgänger. Im rechten Teil kommt außerdem der gelbe Knoten als Vorgänger dazu. In beiden Fällen repräsentieren die Vorgängerknoten Methoden die dann die Kinder des Vorgängerknotens aufrufen. im linken Fall ruft das Wurzelelement also direkt den blauen Teilbaum auf, im rechten Teilbaum erst den gelben Knoten, der dann wiederum den rechten Teilbaum aufruft. Befinden sich nun nur im gelben Knoten die gesuchten Zuweisungen, so ändert sich der Erfolg der Verbesserungen je nachdem welcher der beiden Teilbäume entfernt wird. Da der Baum per Tiefendurchlauf erstellt wird würde in diesem Fall der rechte Teilbaum nicht erstellt werden, die Verbesserungen gingen also verloren.

## 5.2. Behandlung von Zyklen

Innerhalb des Abhängigkeitsbaumes kann es zu Zyklen vorkommen, also zu Knoten die irgendwann wieder sich selbst als Abhängigkeit erkennen. Diese Zyklen werden durch Überprüfung aller Vorgänger eines einzufügenden Knotens entdeckt und pauschal mit der Größe 1 bewertet. Die weitere Auflösung des Abhängigkeitsbaumes wird an dieser Stelle nicht fortgeführt, da der Baum sonst unendlich groß werden würde.

## 5.3. Grafische Darstellung der Ergebnisse

Um die Ergebnisse einer bestimmten Analyse (nicht der Evaluation) verständlich darzustellen wurde neben dem eigentlichen Prototyp auch ein Werkzeug in JavaScript erstellt, mit dem die Ergebnisse grafisch dargestellt werden können.

Bei der Erstellung des Tools wurde darauf geachtet auch große Bäume schnell zu verarbeiten. Aus Gründen der Übersicht wird immer nur ein einziger Knoten und deren direkte Nachfolger betrachtet. Außerdem stehen allgemeine Informationen wie eine Übersicht aller genutzten Klassen bzw. Methoden und die Gesamtgröße des untersuchten Baumes zur Verfügung.

# 6. Evaluation

Die Evaluation lässt sich für zwei relevante Fragestellungen ausrichten. Zum Einen ist die Frage „Wieviele Knoten werden in einem Abhängigkeitsbaum verbessert?“ interessant, zum Anderen die Frage „Wie sehr lassen sich einzelne Knoten verbessern?“. Beide Fragestellungen sind relevant für die Abschätzung des Testaufwands. Allerdings würde die pure Anzahl der gemachten Verbesserungen keine Aussage über die Veränderung des Abhängigkeitsbaumes (und damit des zu erwartenden Testaufwands) zulassen. In 5.1.3 wurde bereits angedeutet, dass die Evaluation sich eher an der zweiten Fragestellung orientieren wird. Dafür gibt es mehrere Gründe. Zunächst spielt die Struktur der Software eine große Rolle für das erfolgreiche Verbessern mittels der in 3 vorgestellten Methoden. Sieht man sich beispielsweise das Konzept zur Konkretisierung von Interfacetypen an ( 4.2.2), so sind eindeutige Voraussetzungen vorhanden damit dieses Konzept überhaupt genutzt werden kann (konkret sind ein Interfacetyp und mindestens eine aufrufende Methode gefordert, die dann einen konkreten Implementierungstypen zuweist). Diese Voraussetzungen sind nur an bestimmten Stellen im Projekt gegeben, das Design hat also einen großen Einfluss auf die Anzahl der möglichen Verbesserungen.
Trotz dieser Gründe wird in 6.2 ab und zu auf die Anzahl der Verbesserungen Bezug genommen. Diese Daten sind dann unter der in 5.1.3 gemachten Einschränkungen zur Evaluation zu betrachten.
Das Hauptaugenmerk der Evaluation dient allerdings zur Einschätzung der *verbesserten* Teilbäume (nicht zu verwechseln mit dem gesamten Abhängigkeitsbaum, die getroffenen Aussagen werden aber anhand von Daten getroffen, die auch für den ganzen Abhängigkeitsbaum ermittelt werden können). Am Ende der Evaluation wird dann eine Einschätzung zur Qualität dieser verbesserten Teilbäume getroffen werden können und die Auswirkungen auf die Abschätzung des Testaufwands erklärt.

## 6.1. Aufbau

Die Evaluation erfolgt mithilfe des in dieser Arbeit entwickelten Prototypen. Testobjekte sind sowohl die Software der Dirk Rossmann GmbH, als auch diverse Open-Source-Programme. Letztere werden benötigt, da die Software der Dirk Rossmann GmbH das Spring-Framework[1] nutzt und Interfaces mittels Dependency Injection[2] einen tatsächlichen Implementierungstypen zugewiesen bekommen. Die nötige Konfiguration für diese Dependency Injections liegt in einer XML-Struktur vor, so dass sie von dem hier entwickelten Programm nicht gelesen werden können. Ohne eine Zuweisung von

[1] http://projects.spring.io/spring-framework/
[2] http://www.itwissen.info/definition/lexikon/Dependency-Injection-dependency-injection-DI.html

Implementierungstypen im Code kann jedoch auch keine der in 4.2.2 beschriebenen Methoden getestet werden. An dieser Stelle dienen die Open-Source-Programme als Ersatz. Außerdem lassen sie an einigen Stellen interessante Vergleiche zu und stützen die Vermutung von Zusammenhängen zwischen Eigenschaften einer Software und dem Erfolg der Analyse. An jedes dieser Programme werden die gleichen Anforderungen wie an die Software der Dirk Rossmann GmbH gestellt, insbesondere also zumindest teilweise vorhandener Quelltext und eine maximale Java-Version 1.4.
Die Evaluation selbst besteht aus einem vollständigen Durchlauf der in dieser Arbeit beschriebenen Analyse, inklusive der Nachbearbeitung des Abhängigkeitsbaumes in 4.2. Als Wurzelelement wird *jede Methode von jeder zum jeweiligen Projekt gehörende Klasse* gewählt. Jede Methode stellt also einmal den Startpunkt einer Abhängigkeitsanalyse dar. Der Abhängigkeitsbaum ändert sich für jedes Wurzelelement und damit auch die möglichen Konkretisierungen. Besteht ein Projekt beispielsweise aus 10 Klassen mit jeweils 15 Methoden pro Klasse, so wird die Analyse 150 mal durchgeführt, jeweils von einer anderen Methode als Wurzelelement. Die Ergebnisse sind einzeln betrachtbar, in 6.2 wird aber auch ein Überblick pro Projekt angegeben. Diese Art der Evaluierung stellt sicher, dass trotz der Einschränkungen durch die Implementierungen alle Abhängigkeiten wenigstens einmal untersucht werden. Auf Durchschnitte der Ergebnisse für ihre jeweiligen Projekte wird nach Möglichkeit (aber nicht immer) verzichtet. Auch hier sei wieder auf die Vorraussetzungen verwiesen, die an vielen Stellen nicht gegeben sind und die tatsächlichen Ergebnisse deshalb „verwässern“ würden.
Nach der Analyse werden die entstandenen Bäume nach verbesserten Teilbäumen durchsucht. Diese Teilbäume werden dann ihrerseits wieder auf ihre Größe, die Anzahl der jeweils unterschiedlichen Klassen und die Anzahl der jeweils unterschiedlichen Methoden untersucht. Die Ergebnisse *aller* optimierten Teilbäume dienen dann als Grundlage für die getroffenen Aussagen.

Die zur Evaluation genutzten Projekte sind:

1. Freecol Version 0.5[3] - ein OpenSource Strategiespiel, das seit 2002 entwickelt wird und momentan in Version 0.10.7 bereit steht. Die hier genutzte ältere (lauffähige) Version begründet sich mit der damals genutzten Java-Version 1.4
2. Apache HttpComponents 4.1[4] - ein Framework zur Vereinfachung der Kommunikation mit dem HTTP-Protokoll
3. JHotDraw 6.0 Beta 1[5] - ein Java-Gui Framework
4. Jonas 4.10.9[6] - ein OpenSource-Server, der momentan in Version 5.3 zum Download bereit steht. Auch hier begründet sich die ältere Version mit der genutzten Java-Version.

Bei der Auswahl der Projekte wurde darauf geachtet, dass nicht nur verschieden Größen (hinsichtlich der Anzahl an Klassen und Methoden) gewählt wurden, sondern auch

[3] http://www.freecol.org/
[4] http://hc.apache.org/
[5] http://www.jhotdraw.org/
[6] http://jonas.ow2.org/xwiki/bin/view/Main/

unterschiedliche Anwendungsbereiche abgedeckt werden. Freecol beispielsweise ist als Spiel direkt darauf ausgelegt vom Anwender gestartet und genutzt zu werden. Http-Components hingegen dient ähnlich wie JHotDraw eher anderen Programmierern als Unterstützung in ihren eigenen Projekten, es sind also mehr Interfaces (als Schnittstellen) zu erwarten. Jonas ist ein eigenständiges Programm, das jedoch darauf ausgelegt ist vom Benutzer möglichst einfach und ohne Programmierkenntnisse erweitert werden zu können. Neben einer breiten Auswahl bezüglich der Anzahl der Interfaces ist auch eine relativ weite Abdeckung von Vererbungen vertreten. Ein Überblick über alle Projekte gibt Tabelle 6.1.

| Name | ADIT | #Klassen | #Methoden | #Interfaces |
|---|---|---|---|---|
| Dirk Rossmann GmbH | 0.1686 | 7300 | 528 | 27 |
| Freecol | 0.4439 | 424 | 4402 | 16 |
| HttpComponents | 0.2216 | 167 | 1029 | 50 |
| JHotDraw | 0.9182 | 611 | 6239 | 48 |
| Jonas | 0.8712 | 2406 | 19057 | 136 |

Tabelle 6.1.: Eigenschaften der getesteten Software

Die einzelnen Spalten in Tabelle 6.1 haben folgende Bedeutungen:

1. **ADIT** Eine spezielle Form der „Depth Inheritance Tree“-Metrik. Normalerweise wird diese Metrik (DIT) für eine Klasse als Abstand des Typs zum Ursprungsobject `java.lang.object` genutzt [10], also die gesamte Tiefe des Vererbungsbaums. Allerdings werden viele Klassen aus den Paketen `java.*` und `javax.*` abgeleitet. Wie in 4.2 beschrieben werden in dieser Arbeit aber alle Klassen und Abhängigkeiten aus diesen Paketen ignoriert, denn für gewöhnlich stellen sie keine zu testenden Artefakte dar. In Tabelle 6.1 stellt ADIT nun die *durchschnittliche Vererbungstiefe aller Klassen* im Projekt dar, die *nicht zu den Packages java.* und javax.** gehören. Erbt eine Klasse beispielsweise von überhaupt keiner Klasse explizit, dann wird in diesem Fall ein ADIT von 0 ermittelt, da `java.lang.Object` zwar die Vaterklasse ist, aber nicht berücksichtigt wird. Gleiches gilt, wenn eine Klasse von beispielsweise `java.util.ArrayList` erben sollte, denn auch diese Klasse befindet sich im Package `java`. Wird hingegen von einer beliebigen Klasse außerhalb dieser Packages erweitert, so wird der ADIT für die Kindklasse um 1 erhöht. Im Grunde werden also alle Java-Klassen aus dem Vererbungsbaum herausgeschnitten.
ADIT zeigt also die durchschnittliche Anzahl der erweiterten Klassen (nach obigen Auswahlkriterien) an. Ein Wert von nahe 1 bedeutet beispielsweise, dass durchschnittlich nahezu jede Klasse im Projekt eine Klasse erweitert, die nicht zu den Packages `java` und `javax` gehören. Dies kann als gute Voraussetzung für die Analyse in 4.2.1 interpretiert werden. Ein Wert nahe 0 legt hingegen nahe, dass Vererbungen nur sehr gezielt eingesetzt werden und viele Klassen entweder von nicht betrachteten Klassen erben, oder generell keine andere Klasse erweitern (`java.lang.Object` natürlich ausgenommen). In einem solchen

Fall sind die Chancen für eine erfolgreiche Anwendung von 4.2.1 schlechter, da Vererbung hier eine Vorraussetzung ist.

2. **#Klassen** Die Anzahl der im Projekt referenzierten Klassen, die durch DependencyFinder analysiert wurden. Also alle Klassen, die im Projekt genutzt werden und der Anwendung mindestens als Class-Datei zur Verfügung stehen.
3. **#Methoden** Anzahl der Methoden im Projekt.
4. **#Interfaces** Die Anzahl der Interfaces im Projekt. Das Verhältnis aus Klassen und Interfaces kann als Indikator für die Erfolgswahrscheinlichkeit der Analysemethode in 4.2.2 genutzt werden. Mehr Interfaces bedeuten mehr Typen, die durch die Analyse konkretisiert werden könnten. Die bloße Anzahl ist allerdings kein hinreichendes Kriterium, da die Art wie die Interfacetypen genutzt (und Implementierungstypen zugewiesen) werden genauso entscheidend ist.

## 6.2. Ergebnisse

Für jedes Projekt wurde wie beschrieben ein Analysedurchlauf für jede mögliche Methode als Startmethode durchgeführt. Die absoluten Ergebnisse sind in Tabelle 6.2 angegeben. Auch hier sei wieder der Hinweis auf die Einschränkungen durch die prototypische Implementierung gegeben. Trotzdem lassen sich Tendenzen über die Effektivität der einzelnen Analysemethoden feststellen und ein Eindruck über die Auswirkung der Verbesserungen auf die gefundenen Abhängigkeiten bekommen. Gesondert betrachtet werden muss der Punkt „Analyse mit Interfaceerkennung“ für die Software der Dirk Rossmann GmbH, denn durch die Nutzung von Frameworks lassen sich die Implementierungstypen hinter den Interfacevariablen nicht statisch auflösen.
Die eingetragene Zahl in jeder Zelle gibt die summierte Größe *aller* Abhängigkeitsbaume eines Projektes für die in den Spaltentiteln angegeben Analysemethode(n) an. Alle 4402 erstellen Abhängigkeitsbäume des Programms Freecol haben zusammen beispielsweise nach einer „normalen Analyse“ (also ohne die in dieser Arbeit vorgestellten Verbesserungen) eine Größe von 496.037 Knoten. Nach hinzuziehen der ersten Verbesserungsmethode, der Vererbungserkennung in 4.2.1, erhöht sich dieser Gesamtwert auf 553.073 Knoten (Differenz von 57.036 Knoten).

| Name | A1: reine Analyse | A2: A1 mit Vererbungserkennung | A3: A2 mit Interfaceerkennung |
|---|---|---|---|
| Dirk Rossmann GmbH | 129.753 | 135.041 | 135.041 |
| Freecol | 496.037 | 553.073 | 558.193 |
| HttpComponents | 6.437 | 7.060 | 7.068 |
| JHotDraw | 61.606 | 79.165 | 82.825 |
| Jonas | 438.317 | 525.821 | 526.423 |

Tabelle 6.2.: Erkannte Abhängigkeiten mit unterschiedlichen Analysen

Aus diesen Daten wird bereits deutlich, dass die Verbesserung durch die Methode zur Erkennung von geerbten Softwareartefakten in allen Projekten wesentlich erfolgreicher ist, als die Erkennung von Interfacetypen. Außerdem lassen sich Zusammenhänge zwischen dem ermittelten ADIT-Wert eines Projektes und der Effektivität dieser Methode erkennen. Jonas und JHotDraw haben einen vergleichsweise hohen Wert von 0.8712 und 0.9182, und erreichen eine Vergrößerung der insgesamt gefundenen Abhängigkeiten von 19,97% bzw. sogar 28,5% nur mit der ersten Methode. Die Projekte Freecol, HttpComponents und die Software der Dirk Rossmann GmbH haben einen im Vergleich niedrigen ADIT-Wert, ebenso sind die erreichten Verbesserungen mit dieser Methode deutlich niedriger, nämlich nur etwa 6% (Freecol), 9,5% (HttpComponents) und nur etwa 4% bei der Software der Dirk Rossmann GmbH.

Die Tabelle 6.3 enthält noch einmal zum direkten Vergleich die absoluten Zahlen der durch die Verbesserungsmethoden gefundenen Abhängigkeiten. Damit lassen sich die beiden genutzten Methoden zur Verbesserung relativ gut vergleichen. Die dritte Spalte zeigt außerdem das größte Verhältnis der Größe eines Teilbaums mit und ohne Verbesserung. Für das Programm Freecol beträgt dieser Wert beispielsweise 383:1. Das bedeutet, dass ein Teilbaum eines Abhänigkeitsbaumes mit den Methoden zur Verbesserung um den Faktor 383 vergrößert werden konnte. Hatte dieser Teilbaum vor der Verbesserung eine Größe von 3 Knoten, so hat er nach der Verbesserung eine Größe von 3 * 383 = 1149 Knoten. Auf absolute Zahlen wurde hier verzichtet, da die Abhängigkeitsbäume für unterschiedliche Startmethoden immer unterschiedlich groß sind und absolute Verbesserungszahlen je nach Ausgangsgröße anders interpretiert werden müssten.

| Name | ØVerbesserung durch A2 | ØVerbesserung durch A3 | maximale Verbesserung | verbesserte Methoden |
|---|---|---|---|---|
| Dirk Rossmann GmbH | 5288 | - | 281:1 | 267 (2.4%) |
| Freecol | 29640 | 6021 | 383:1 | 437 (9,9%) |
| HttpComponents | 623 | 8 | 43:1 | 167 (16.2%) |
| JHotDraw | 17559 | 3660 | 562:1 | 611 (9,8%) |
| Jonas | 87504 | 602 | 376:1 | 2406 (12,6%) |

Tabelle 6.3.: Ergebnisse des Konzeptes dieser Arbeit

Mit diesen Daten lassen sich bereits eingeschränkt Rückschlüsse über die Häufigkeit der Anwendung einer Verbesserung ermitteln. Außerdem kann man bereits eine Vorstellung über die Effektivität einer einzelnen Verbesserung entwickeln, beispielsweise durch die angegebene maximale Verbesserung (also der Bestfall) oder durch Betrachten der absoluten Verbesserungen und der Anzahl der verbesserten Methoden.
Um genauere Aussagen über die Größe der verbesserten Bäume zu bekommen wurden für jedes Projekt alle *verbesserten Teilbäume* untersucht und in Boxplot-Diagramme

eingetragen. Die Daten bestehen also aus *jedem* gefundenen verbessertem Knoten in *jedem* für die Evaluation berechneten Abhängigkeitsbaum. Besitzt ein Abhängigkeitsbaum keine verbesserten Knoten und daraus resultierende Teilbäume, so ist auch kein Teil von ihm in der folgenden untersuchten Menge. Genauso gut kann es vorkommen, dass zwei verschiedene Teilbäume aus einem einzigen Abhängigkeitsgraphen verbessert wurden. In diesem Fall sind auch beide Teilbäume in den Daten der Boxplot-Diagramme vorhanden.

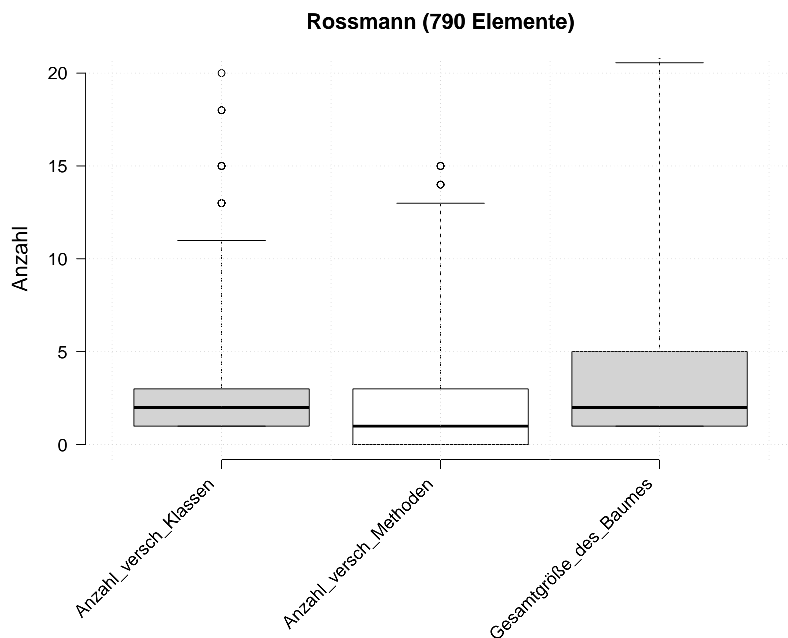


Abbildung 6.1.: Verteilung der verbesserten Teilbäume für die Software der Dirk Rossmann GmbH

In den Diagrammen sind die jeweiligen Anzahlen der unterschiedlichen Klassen und Methoden in den *optimierten* Teilbäumen eingetragen. Pro Teilbaum wird eine Klasse bzw. Methode also nur einmal gezählt, auch wenn sie mehrmals vorkommen sollte. Die Abbildungen zeigen den Median der Daten an, außerdem als Box eingezeichnet die zwei Quartile in denen sich die mittleren 50% der Datenpunkte befinden. Zusätzlich ist eine gestrichelte Altman-Antenne (Whisker) eingezeichnet, die 95% aller Datenpunkte abdeckt. Das heißt insbesondere, dass über dieser Antenne noch 2,5% aller Datenpunkte liegen (entsprechend liegen 2,5% aller Datenpunkte unter der unteren Grenze der Antenne). Außerhalb der Antenne sind schließlich alle Vorkommen von Datenpunkten bei einem Wert mittels Punkten im Diagramm eingezeichnet. Je dunkler der Punkt, desto mehr Daten liegen an der entsprechenden Stelle im Graph. Für alle Diagramme gilt außerdem, dass zur Übersichtlichkeit die Y-Achse nur bis zur Größe 20 gezeichnet wird. Bei allen Grafiken gibt es noch Datenpunkte die weit über diesem Wert liegen, die Abbildung aber zu sehr in die Länge ziehen würden.

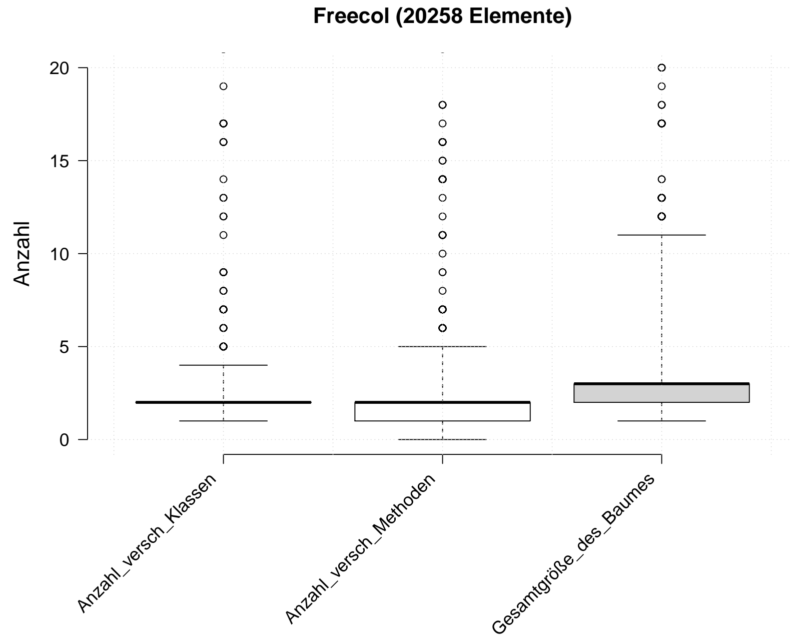


Abbildung 6.2.: Verteilung der verbesserten Teilbäume für Freecol

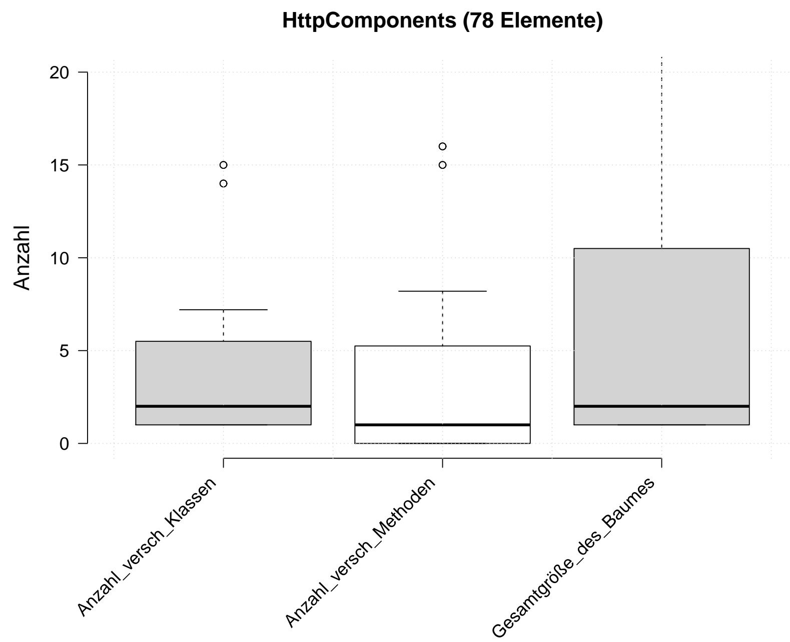


Abbildung 6.3.: Verteilung der verbesserten Teilbäume für HttpComponents

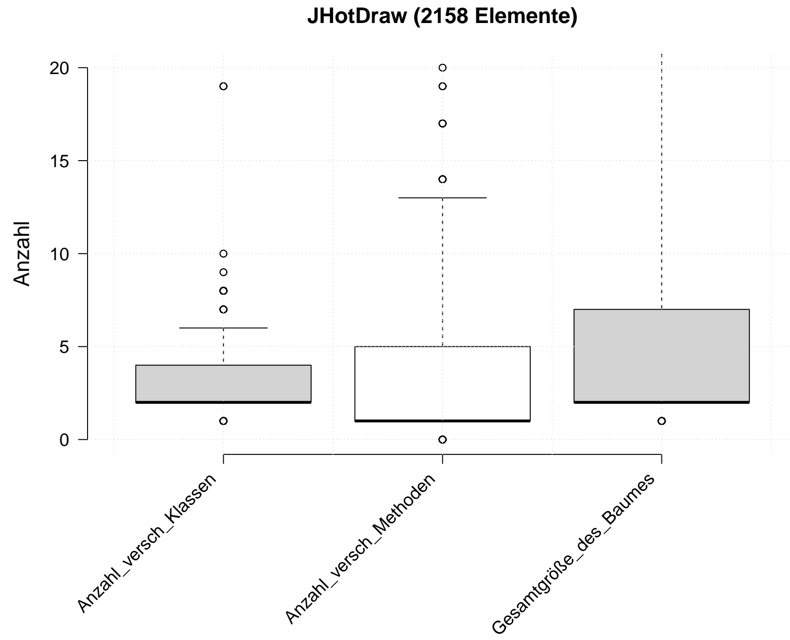


Abbildung 6.4.: Verteilung der verbesserten Teilbäume für JHotDraw

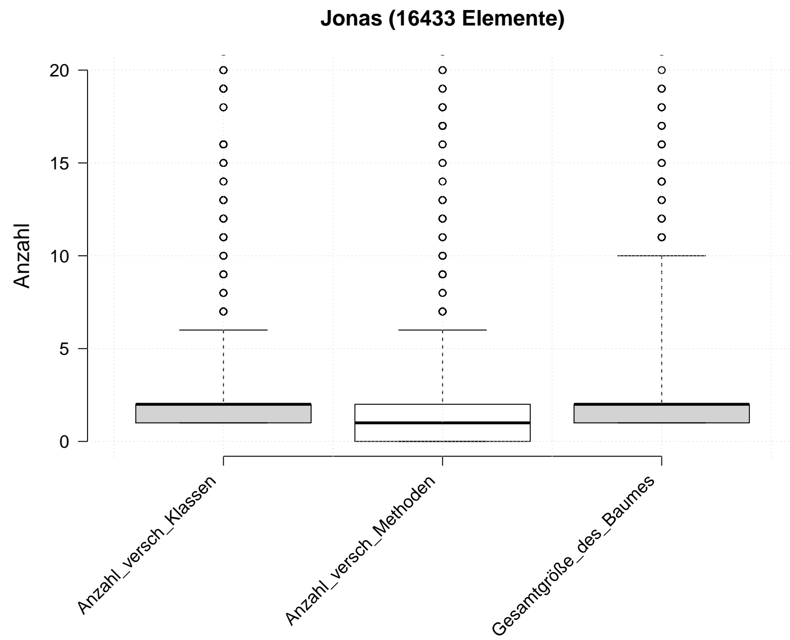


Abbildung 6.5.: Verteilung der verbesserten Teilbäume für Jonas

## 6.3. Diskussion

### 6.3.1. Interpretation

Aus den Tabellen 6.1 und 6.2 lässt sich ein Zusammenhang zwischen der durschnittlichen Vererbung und der Effektivität der Analyse vermuten. Je größer der Wert „ADIT“ in Tabelle 6.1 ist, desto besser funktionierte die Analyse aus 4.2.1. Ein ähnlicher Zusammenhang zwischen der Anzahl der Interfaces im Projekt und der Effektivität der Analyse aus 4.2.2 lässt sich aus den vorhandenen Daten nicht bestätigen. Hier scheint also die Nutzungsart der Interfaces und die Art der Zuweisungen von Implementierungstypen entscheidender zu sein, als die bloße Existenz von vielen Interfaces. Insgesamt geht aus 6.3 hervor, dass zumindest für die getesten Projekte die erste Verbesserungsmethode deutlich effektiver ist als die zweite. Dies kann aber zumindest teilweise auch mit den Einschränkungen in 5.1.3 erklärt werden, da die Auflösung der Interfacetypen auf passende Pfade im Abhängigkeitsbaum angewiesen ist.
Bezüglich der Größe der verbesserten Teilbäume zeigen die Boxplot-Diagramme eine klare Verteilung. Abgesehen von HttpComponents haben mehr als 50% aller Bäume eine Größe von weniger als 10 Knoten. Die Anzahl der unterschiedlichen Klassen bzw. Methoden ist meistens noch kleiner. Auffällig ist die Größe der Boxen im Zusammenhang mit der Anzahl der Datenpunkte. Je mehr Datenpunkte vorhanden sind, desto kleiner werden die Werte in den Quartilen. Auch die Größe der Antennen sinkt mit zunehmender Datenmenge deutlich ab. Eine mögliche Erklärung für diesen Anstieg an kleinen Teilbäumen liefert möglicherweise der Aufbau der Evaluation. Als Beispiel betrachten wir eine Methode `m()`, die von den Methoden `a()` und `b()` aufgerufen wird, also eine Abhängigkeit der beiden Methoden darstellt. Duch die Art der Evaluation wird nun zunächst `m()` selbst analysiert. Angenommen dabei wird ein Teilbaum von `m()` verbessert, so wird dieser verbesserte Teilbaum als Datenpunkt in die Boxplot-Diagramme eingetragen. Durch die Evaluation werden nun aber auch die Methoden `a()` und `b()` analysiert. Da `m()` als Teilbaum bei beiden Methoden vorkommt, wird auch `m()` beide Male mitanalysiert. Die Ergebnisse von `m()` fallen also mindestens drei mal in die Datenmenge der Boxplot-Diagramme und beeinflussen so das Ergebnis stärker, als andere Teilbäume. Das gilt allgemein für Methoden die häufig aufgerufen werden, beispielsweise Getter- und Settermethoden, die vermutlich bei nahezu jeder Nutzung des dazugehörigen Objektes aufgerufen werden. Deshalb sind neben den Quartilen auch die „Ausreißer“ über der Antenne für die Auswertung interessant. Diese zeigen nämlich, dass es durchaus Methoden gibt die von den in dieser Arbeit analysierten Methoden in größerem Maße profitieren. Das zeigen auch die maximalen Verbesserungen aus Tabelle 6.3.
Die Anzahl der Knoten in den gefundenen Verbesserungen kann als Anhaltspunkt für den zu erwartenden Testaufwand verstanden werden, der ohne diese Verbesserungen möglicherweise übersehen worden wäre. Für die vielen kleinen Teilbäume bedeutet dies, dass der zu erwartende Testaufwand nur wenig beeinflusst werden dürfte. Je mehr Knoten im verbesserten Abhängigkeitsbaum vorhanden sind, desto größer ist der Einfluss auf die Abschätzung des Testbedarfs. Daraus lässt sich schließen, dass im Schnitt weniger als die Hälfte der in dieser Evaluation ermittelten Verbesserungen

wirklich relevant für die Abschätzung sind. Die Vorteile die sich aus dem Konzept dieser Arbeit ergeben sind also ebenfalls von der zu analysierenden gewählten Methode abhängig, können dann aber auch signifikant sein.
Die Anzahl der unterschiedlichen Klassen und Methoden erlauben außerdem eine genaure Einschätzung zur Ermittlung des Testaufwands als die simple Größe des Baumes, da doppelte Artefakte für gewöhnlich mit den gleichen Tests überprüft werden können. Diese Informationen sind also eher als Hinweis auf den Testbedarf zu verstehen als die Anzahl aller Knoten im Abhängigkeitsbaum, die in den Boxplot-Diagrammen im Durchschnitt größer war als die Anzahl der unterschiedlichen Methoden.

Insgesamt zeigt sich also, dass ein mittels statischer Analyse erstellter Abhängigkeitsbaum durch eine nachträgliche Analyse im Hinblick auf die bekannten Schwächen zum Teil deutlich verbessert werden kann. Die Abschätzung des Testbedarfs wird damit für jede gewählte Metrik genauer, da mehr Daten zur Verfügung stehen. Hinsichtlich der gewählten Metrik selbst zeigte sich, dass die bloße Größe des Abhängigkeitsbaumes irreführend sein kann. Die Anzahl der verschiedenen genutzten Klassen oder die Anzahl der verschiedenen genutzten Methoden hingegen vermitteln eine deutlichere Aussage über den zu erwartenden Aufwand. Da diese Zahlen nur vom Abhängigkeitsbaum abhängen bedeutet ein verbesserter Abhängigkeitsbaum auch eine genauer berechnete Metrik.

### 6.3.2. Korrektheit der Ergebnisse

In 4.2.2 wurde bereits ein Beispiel für mögliche falsche Einträge im Abhängigkeitsbaum durch die Verfolgung von Interfacetypen gegeben. Die lexikalische Analyse des Java-Codes in dieser Arbeit ist relativ einfach gehalten, deswegen können trotz der benannten Kontrollen bei einem gefunden Typen (Überprüfung der `Import` Anweisungen auf Existenz des gefundenen Typs, Überprüfung ob der Typ eine entsprechende Methode hat und ein entsprechendes Interface implementiert) keine Garantien gegeben werden, dass dieser Typ auch wirklich dem von der JVM eingesetzten Typ entspricht. Bei diversen stichprobenartigen manuellen Überprüfungen konnte jedoch kein falsch Typ ermittelt werden.

### 6.3.3. Vollständigkeit der Ergebnisse

Wie in 5.1.3 erklärt, wurde bei der Implementierung des Prototypen auf die vollständige Erkennung aller Verbesserungen zugunsten einer akzeptablen Laufzeit verzichtet. Insbesondere die gefundenen Ergebnisse der Verfolgung von Interfacetypen stellt deshalb eine untere Grenze der möglichen Erkennungen dar. Würden die Teilbäume mit Interfacetypen öfter im zu untersuchenden Abhängigkeitsbaum auftauchen, so wäre die Chance auf eine mögliche Verbesserung höher. Dies wurde bereits in Kapitel 5.1.3 ausgeführt. Ein größerer oder vollständiger Abhängigkeitsbaum würde also mehr Pfade bieten anhand derer ein Interfacetyp aufgelöst werden könnte.

# 7. Fazit und Ausblick

## 7.1. Fazit

Die in Kapitel 6 gezeigten Daten belegen die Wirksamkeit des in 3 vorgestellten Konzeptes. Die Ergebnisse einer statischen Analyse wurden im Hinblick auf bekannte Limitierungen analysiert und mit anderen Teilergebnissen in Beziehung gesetzt. Dadurch konnte ein zuvor erstellter Abhängigkeitsbaum verbessert werden.

Ein Großteil der verbesserten Teilbäume hatte eine vergleichsweise niedrige Größe, die Relevanz dieser Teilbäume für den Testaufwand dürfte eher klein sein. Es wurden aber ebenfalls Verbesserungen mit deutlich größeren Auswirkungen ermittelt. Die Streuung der Ergebnisse lässt sich mit den nötigen Voraussetzungen für eine erfolgreiche Analyse, dem Aufbau der Evaluation und der Struktur der Software erklären. Je nach Eigenschaften des Projektes hinsichtlich der Verwendung von Interfaces und der Häufigkeit und Tiefe der Vererbungsstrukturen kann so eine deutliche Verbesserung des Abhängigkeitsbaumes erzielt werden und die Abschätzung für den zu erwartenden Testbedarf präzisiert werden. Da es keinen Fall gibt in dem die Erkennungsrate verringert wurde ist dieses Konzept zur Verbesserung des Abhängigkeitsbaums zu empfehlen. Mithilfe des Abhängigkeitsbaums lassen sich dann Metriken errechnen, wie beispielsweise die Anzahl der genutzten Klassen oder Methoden. Anhand dieser Informationen kann nun der Testbedarf abgeschätzt werden.

## 7.2. Ausblick

Das in dieser Arbeit vorgestellte Konzept bietet viele Möglichkeiten zur Erweiterung und Verbesserung, vorrangig für die Methode der Erkennung von Interfacetypen. Die Erkennung von abgeleiteten Artefakten hingegen ist relativ einfach und sicher realisierbar gewesen, so dass dort nur wenig Spielraum zur Verbesserung vorhanden sein dürfte. Eine Möglichkeit zur Verbesserung wäre beispielsweise die prototypische Implementierung. Eine speichereffiziente Lösung ermöglicht möglicherweise die vollständige Entwicklung aller Abhängigkeitsbäume. Damit ließen sich dann auch wirklich alle abgeleiteten Abhängigkeiten auflösen, außerdem bieten sich so neue Pfade zur Erkennung an von Interfacetypen an.
Eine Verbesserung der Codeanalyse könnte ebenfalls zu einer genaueren Erkennung beitragen. Die Berücksichtigung des `instanceof` Operators beispielsweise kann als neuer Fall in der Codeanalyse betrachtet werden. Im Zusammenhang mit der Berücksichtigung von Verzweigungen können so an einigen Stellen die Typen hinter den Interfaces eindeutig bestimmt werden. Listing 7.1 zeigt dies an einem Beispiel, bei dem

der übergebene Parameter für die Methode methodA() einem eindeutigen Typen zugeordnet werden kann.

```
1 if (var instanceof TypA)
2     methodA(var);
3 else
4     methodB(var);
```

Listing 7.1: instanceof-Operator zur Typerkennung

Ebenfalls möglich sind weitere Verbesserungen des Konzepts beim Durchsuchen des Abhängigkeitsbaumes. Die hier vorgestellte Lösung überprüft nur die direkten Vorgänger einer Methode im Abhängigkeitsbaum auf Zuweisungen, die zur Erkennung von Interfacetypen dienen können. Denkbar sind aber auch Konstrukte, bei denen die Zuweisung eines konkreten Typs in einem Nachbarknoten passiert. Listing 7.2 gibt hierfür ein Beispiel. Die Berücksichtigung solcher Zuweisungen würde die Erkennung ebenfalls erhöhen.

```
1  public void show() {
2      ObjectA obj = new ObjectA();
3
4      //obj hat eine Setter-Methode setType(ConcreteObj), die einer
           Interface-Membervariablen einen konkreten
           Implementierungstypen zuweist
5      FillerObject filler = new FillerObject();
6
7      //filler nutzt in der Methode fill() den Setter von obj und
           setzt so einen konkreten Typen
8      filler.fill(obj);
9
10     //in der Methode doSth() wird die Interface-Variable genutzt.
           Diese kann aber nicht verfolgt werden, da die Zuweisung
           nicht in einer der aufrufenden Methoden stattfand
11     obj.doSth();
12 }
```

Listing 7.2: Beispiel für ausgelagerte Typzuweisungen

Um auf Anwendungen wie die der Dirk Rossmann GmbH mit einer alternativen Zuweisung von Implementierungstypen reagieren zu können, wäre außerdem die manuelle Vorgabe von bestimmten Implementierungstypen für bestimmte Interfaces denkbar. In diesem Fall würde der Abhängigkeitsbaum nicht nach Zuweisungen durchsucht, sondern die Abhängigkeiten des Interfaces würden direkt auf die Abhängigkeiten des gewählten Implementierungstypen zeigen. Die eigentliche Analyse würde dadurch zwar nicht beeinflusst, das Ergebnis würde für den Benutzer (bei korrekt gewähltem Implementierungstypen) aber noch aussagekräftiger werden. Dieser Ansatz kann außerdem für verschiedene Frameworks mit unterschiedlichen Methoden der Zuweisung von Typen zu Interfaces genutzt werden.

Schließlich gibt es außerdem die Möglichkeit die Aussagekraft des erstellten Abhängigkeitsbaum mittels Metriken zu erhöhen. In dieser Arbeit werden nur relativ ungenaue Informationen bezüglich der Anzahl der verschiedenen Klassen und Methoden angegeben, es ist aber genauso vorstellbar bei der Erstellung des Baumes (oder einem späteren Durchlauf) Metriken wie beispielsweise die McCabe-Komplexität zu berechnen und damit die Abschätzung des Testaufwands noch genauer zu ermöglichen.

# Literaturverzeichnis


[1] Christoph Stoike,
*Konfiguration von Java-Applikationen durch Abhängigkeitsanalyse*,
Christian-Albrechts-Universität zu Kiel, 2007

[2] T.Syst, P.Box, F.Tampere,
*Analyzing Java Software by Combining Metrics and Program Visualization*,
Software Maintenance and Reengineering, Proceedings of the Fourth European, S. 199 - 208, 2000

[3] Yin Liu, Ana Milanova,
*Static Analysis for Dynamic Coupling Measures*,
Proceedings of the 2006 conference of the Center for Advanced Studies on Collaborative research, Article No. 10, 2006

[4] P.Louridas,
*Static Code Analysis*,
Software, IEEE (Volume: 23, Issue: 4), S. 58 - 61, 2006

[5] Jason Sawin,
*Improving the Static Resolution of Dynamic Java Features*,
Springer Verlag, Automated Software Engineering, 2009

[6] Lionel C. Briand, Jürgen Wüst, Hakim Lounis
*Using Coupling Measurement for Impact Analysis in Object-Oriented Systems*,
IEEE International Conference on Software Maintenance, (ICSM ’99) S. 475 - 482, 1999

[7] Jean Tessier,
*DependencyFinder Manual*,
`http://depfind.sourceforge.net/Manual.html`
(abgerufen am 05.08.2014)

[8] Tim Lindholm, Frank Yellin, Gilad Bracha, Alex Buckley,
*The Java Virtual Machine Specification, Java SE7 Edition*,
Prentice Hall, 2013

[9] *Introduction (The Java Tutorials)*,
`http://docs.oracle.com/javase/tutorial/extra/generics/intro.html`
(abgerufen am 06.08.2014)

[10] Shyam R. Chidamber, Chris F. Kemerer,
*Towards a metrics suite for object oriented design*,
Conference proceedings on Object-oriented programming systems, languages and applications, S. 197-211, 1991

[11] H.Sneed,
*Estimating the Costs of a Reengineering Project*,
12th Working Conference on Reverse Engineering, S. 111 - 120, 2005

[12] *Dynamic Analysis vs. Static Analysis*,
`https://software.intel.com/sites/products/documentation/doclib/iss/2013/inspector/lin/ug_docs/GUID-E901AB30-1590-4706-94B1-9CD4736D8D2D.htm`
(abgerufen am 15.08.2014)

[13] Prof. Dr. Rainer Koschke,
*Vorlesung Software-Reengineering*,
`http://www.informatik.uni-bremen.de/st/lehre/re09/dynamische_analyse-1x2.pdf`
(abgerufen am 14.08.2014)

[14] Klaus Krogmann,
*Artefakte*,
`http://ak-traceab.gi.de/fileadmin/artefakte-KlausKrogmann.pdf`
(abgerufen am 14.08.2014)

[15] L. Hochstein, M. Lindvall,
*Combating architectural degeneration: a survey*,
Information and Software Technology, Volume 47, Issue 10, S. 643–656, 2005

[16] Stefan Jungmayr,
*Testability measurement and software dependencies*,
Proceedings of the 12th International Workshop on Software Measurement, S. 179-202, 2002

# A. Testmethoden zur Auswahl eines Werkzeugs

Zunächst wurden einige Klassen und ein Interfaces als Grundlage der Tests geschrieben. Die Klassen `FindMe`, `FindMeChild` und `OrFindMe` stellen dabei die Typen dar die als Abhängigkeit gefunden werden sollen. Zwei von ihnen implementieren das Interface `FindMeInterface`. Damit lassen sich alle Tests durchführen, auch solche in denen auf Vererbung oder bestimmte Implementierungstypen eines Interfaces getestet werden soll.
In allen Methoden ist ein `System.out.println()` untergebracht. Damit soll verhindert werden dass Methoden vom Compiler wegoptimiert werden und so einige Typen gar nicht gefunden werden könnten.

```java
public interface FindMeInterface {
        public void doSth();
}

public class FindMe implements FindMeInterface {
        public void doSth() {
                System.out.println("DoSth␣aufgerufen");
        }

        public void papaMethod() {
                System.out.println("Papa-Methode␣aufgerufen");
        }

        public static FindMeInterface getType(boolean alternative
            ) {
                if (alternative)
                        return new OrFindMe();
                else
                        return new FindMe();
        }
}

public class FindMeChild extends FindMe {

        public void doSth() {
                System.out.println("FindMeChild␣sagt␣Hallo");
        }
}

public class OrFindMe implements FindMeInterface {

        public void doSth() {
                System.out.println("OrFindMe␣sagt␣Hallo");
        }

}
```

Listing A.1: Basis der Testklassen

Die einzelnen Testfälle stehen in Listing A.2. Zur Übersicht gibt ein Kommentar den gesuchten Typen an.

```
1  public class TestCast {
2          //Gesuchter Typ: FindMe
3          public void testCast(FindMeChild obj) {
4                  FindMe var = (FindMe)obj;
5                  var.doSth();
6                  System.out.println("Cast␣Test");
7          }
8  }
9
10 public class TestDerivedClass {
11     //Gesuchter Typ: FindMe.papaMethod()
12         public void testDerivedClass(FindMeChild obj) {
13                 obj.papaMethod();
14                 System.out.println("Papatest␣fertig");
15         }
16 }
17
18 public class TestInstantiate {
19     //Gesuchter Typ: FindMe
20         public void testInstanciate() {
21                 FindMe variable = new FindMe();
22                 System.out.println("FindMe␣Instanziierung");
23                 variable.doSth();
24         }
25 }
26
27 public class TestInterfaceTypes {
28     //Gesuchter Typ: OrFindMe
29         public void testInterface() {
30                 FindMeInterface obj = FindMe.getType(true);
31                 obj.doSth();
32                 System.out.println("testInterface␣fertig");
33         }
34 }
35
36 public class TestMemberVar {
37         FindMe obj = new FindMe();
38         FindMeChild child;
39     //Gesuchter Typ: FindMe und FindMeChild
40         public void testMemberVars() {
41                 this.obj.doSth();
42                 System.out.println("Membervariablen␣fertig");
43         }
44 }
45
46 public class TestSignature {
47     //Gesuchter Typ: FindMe
48         public void testSignature(FindMe obj) {
49                 System.out.println("FindMe␣in␣Signatur");
50                 obj.doSth();
51         }
52 }
```

Listing A.2: Testklassen

# B. Beschreibung zur Nutzung des Prototypen

## B.1. Prototyp zur statischen Analyse

Der Prototyp befindet sich als Eclipse-Projekt auf der CD. Die Anwendung kann einfach gestartet werden und läuft dann abhängig von der angegebenen Konfiguration selbstständig ab. Die Konfiguration erfolt mittels der Datei „config“. Die Konfiguration erfolgt in der Form „Schlüsselwort=Wert“. Die Datei wird zeilenweise ausgelesen, pro Zeile kann es also nur eine Anweisung geben. Einzelne Zeilen können aber mit einem vorgestelltem „//“ auskommentiert werden.
Folgende Optionen stehen zur Verfügung:

1. **xml** Der Name der XML-Datei in der die Zwischenergebnisse von DependencyFinder gespeichert werden.
2. **name** Ein Name der vergeben werden kann. Nur interessant wenn die Anwendung im Evaluationsmodus läuft, denn dann werden einige Dateiennamen mittels dieses Namen erstellt.
3. **calculateFullTree** Wird „true“ angegeben, so wird der Abhängigkeitsbaum komplett berechnet und damit in den meisten Fällen genauer werden. Es gibt aber Fälle, in denen der Abhängigkeitsbaum in dieser Implementierung zu groß wird. In diesem Fall kann der Wert geändert werden und der Baum wird nicht voll entwickelt. Details dazu in Kapitel 5.1.3.
4. **startingMethod** Hier wird die Methode angegeben die als Wurzelelement des Abhängigkeitsbaumes genutzt werden soll. Die Methode muss mit vollqualifizierendem Namen angegeben werden, ebenso eventuelle Parameter. Ein Wert könnte beispielsweise so aussehen:
   `de.unihannover.se.TestClass.MethodA(java.lang.String)`
5. **recompileXml** Wird „true“ angegeben, so werden die Zwischenergebnisse von DependencyFinder bei jedem Durchlauf neu errechnet. Möchte man mehrere Abhängigkeitsbäume aus dem gleichen Projekt berechnen kann der Wert geändert und die Zwischenergebnisse wiederverwendet werden.
6. **classFileSourceDirectory** Das Verzeichnis in dem die *Class-Dateien* liegen die analysiert werden sollen. Unterordner werden ebenfalls durchsucht und alle gefundenen Class-Dateien in die Analyse miteinbezogen. Es kann nur ein Verzeichnis angegeben werden.
7. **sourceDirectory** Das Verzeichnis in dem die Packagestruktur der *Java-Dateien* beginnt. Ist eine Klasse beispielsweise im Package `de.unihannover.se`, so gibt es eine entsprechende Ordnerstruktur „se/unihannover/de“. Der Pfad zu

dieser Struktur muss angegeben werden, die Struktur selbst jedoch nicht. In Eclipce-Projekten ist dies meist das „src“-Verzeichnis.

8. **filter** Hier kann ein regulärer Ausdruck angegeben werden nach dem sämtliche Einträge im Abhängigkeitsbaum gefiltert werden.
   Würde beispielsweise `de\.unihannover\.se\.*` angegeben, würden nur Artefakte aus `de.unihannover.se` oder einem Unterpackage in die Untersuchung einbezogen werden.
9. **mode** Hier kann entweder „evaluate“ oder „checkMethod“ angegeben werden. Mit „evaluate“ werden alle gefundenen Methoden als Wurzelelement genutzt und am Ende eine Statistik ausgegeben. Mit „checkMethod“ wird die unter *startingMethod* angegebene Methode untersucht und die Ergebnisse in der Datei *export* ausgegeben, die dann zur Visualisierung mit dem JavaScript-Werkzeug genutzt werden kann.
10. **export** Der Name der Datei in dem die Ergebnisse ausgegeben werden sollen, sofern der gewählte *mode* „checkMethod“ entspricht.
11. **methodsOnly** Hier muss „true“ angegeben werden, wenn anstatt aller Softwareartefakte (inklusive Klassen- und Membervariablen) nur Methoden im Abhängigkeitsbaum berücksichtigt werden sollen.
12. **calculateTestCoverage** Hier muss „true“ angegeben werden, wenn vorhandene Testfälle geparst werden sollen und die Testabdeckung in den Abhängigkeitsbaum übernommen werden soll
13. **testClassFileSourceDirectory** Das Verzeichnis mit den *Class-Dateien* welche die zu untersuchenden Testfälle beinhalten. Es gelten die gleichen Bedingungen wie für *classFileSourceDirectory*.
14. **testSourceDirectory** Das Verzeichnis mit den *Java-Dateien* die zu den Testfällen gehören. Es gelten die gleichen Bedingungen wie für *sourceDirectory*.
15. **testXml** Die XML-Datei in der die Zwischenergebnisse von DependencyFinder bezüglich den Testfällen gespeichert werden sollen.

## B.2. Prototyp zur Visualisierung

Wurde ein Abhängigkeitsbaum mittels des *mode* „checkMethod“ erstellt, so kann die Datei mit dem in *export* angegebenen Namen mithilfe des Javascript-Tools auf der CD dieser Arbeit visualisiert werden. Dazu muss im Ordner „JavaScript“ die Datei „index.html“ im Browser geöffnet werden. Nun kann die Ergebnisdatei einfach auf die Seite gezogen werden und wird analysiert. Die obere Statusleiste gibt allgemeine Informationen über den Abhängigkeitsbaum an. Die Liste „Abhängigkeiten“ enthält alle Kinder des letzten Elements in der Liste „Vorgänger“, mittels Klicks auf eines der Kinder oder einen der Vorgänger kann durch den Baum navigiert werden. Wurden im Abhängigkeitsbaum Informationen zur Testabdeckung eingezogen, so gibt die Farbe den Grad der Abdeckung an: Grün bedeutet vollständig getestet (inklusive Kinder), Gelb bedeutet dass es sowohl getestete als auch ungetestete Abhängigkeiten gibt, und rot signalisiert einen komplett ungetesteten Zweig. Außerdem wird durch einen grünen Rahmen um einen Eintrag angezeigt, dass genau dieser Knoten in den Testfällen vorkommt. Es ist also auch möglich ein Eintrag mit gelbem Hintergrund und grünem Rahmen angezeigt zu bekommen, nämlich dann wenn ein Knoten getestet wurde aber ungetestete Kinder enthält.

# C. Inhalt der CD

1. Eclipseprojekt mit dem in dieser Arbeit entwickeltem Prototypen
2. JavaScript-Werkzeug zur Visualisierung von Ergebnisdateien aus dem Prototypen inklusive Beispieldatei
3. Die in Anhang A angegebenen Testmethoden
4. Die untersuchten Tools aus Kapitel 4.1.1
5. Diese Arbeit als PDF-Datei

# Abbildungsverzeichnis

2.1. Aufrufbaum einer Beispielmethode . . . . . . . . . . . . . . . . . . . . . . . . 5

3.1. Schematische Darstellung des Konzepts . . . . . . . . . . . . . . . . . . . . . 8

4.1. Abhängigkeiten aus Sicht zweier Artefakte . . . . . . . . . . . . . . . . . . . 17
4.2. Umformung der Baumstruktur . . . . . . . . . . . . . . . . . . . . . . . . . . 20
4.3. Aufrufpfad einer Methode . . . . . . . . . . . . . . . . . . . . . . . . . . . . 24
4.4. Testabdeckung im Abhängigkeitsbaum . . . . . . . . . . . . . . . . . . . . . 33

5.1. Gleiche Teilbäume in einem Abhängigkeitsbaum . . . . . . . . . . . . . . . . 35

6.1. Verteilung der verbesserten Teilbäume der Rossmann Software . . . . . . . 43
6.2. Verteilung der verbesserten Teilbäume für Freecol . . . . . . . . . . . . . . . 44
6.3. Verteilung der verbesserten Teilbäume für HttpComponents . . . . . . . . . 44
6.4. Verteilung der verbesserten Teilbäume für JHotDraw . . . . . . . . . . . . . 45
6.5. Verteilung der verbesserten Teilbäume für Jonas . . . . . . . . . . . . . . . 45

# Listings

2.1. Verschiedene Formen von Abhängigkeiten . . . . . . . . . . 4

3.1. Nicht eindeutig bestimmbarer Typ . . . . . . . . . . 12

4.1. Beispiel zur Uneindeutigkeit von Typen der Vaterklasse . . . . . . . . . . 18
4.2. Aufrufe von geerbten Methoden . . . . . . . . . . 22
4.3. Doppelte Erkennung von Implementierungstypen . . . . . . . . . . 26
4.4. Funktionsaufruf von einem Objekt . . . . . . . . . . 29
4.5. Auflösung eines ternären Operators . . . . . . . . . . 30

7.1. instanceof-Operator zur Typerkennung . . . . . . . . . . 49
7.2. Beispiel für ausgelagerte Typzuweisungen . . . . . . . . . . 49

A.1. Basis der Testklassen . . . . . . . . . . 54
A.2. Testklassen . . . . . . . . . . 55

# Erklärung der Selbstständigkeit

Hiermit versichere ich, dass ich die vorliegende Bachelorarbeit selbständig und ohne fremde Hilfe verfasst und keine anderen als die in der Arbeit angegebenen Quellen und Hilfsmittel verwendet habe. Die Arbeit hat in gleicher oder ähnlicher Form noch keinem anderen Prüfungsamt vorgelegen.

Hannover, den 28.08.2014

______________________________
Patrick Liedtke