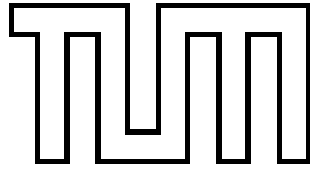


FAKULTÄT FÜR INFORMATIK

DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

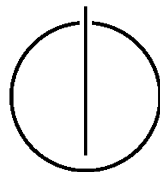
Lehrstuhl IV

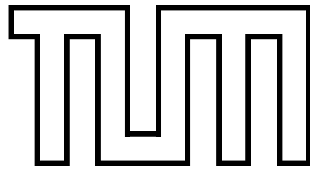
Software & Systems Engineering

Bachelorarbeit in Informatik

Lesbarkeitsanalyse basierend auf Kontroll- und Datenflussgraphen

Martin Klenk





Technische Universität München

Fakultät für Informatik

Lesbarkeitsanalyse basierend auf Kontroll- und Datenflussgraphen

Readability Analysis Based on Control and Data Flow Graphs

Martin Klenk

Bachelorarbeit in Informatik

Aufgabensteller: Prof. Dr. Dr. h.c. Manfred Broy

Betreuer: Benjamin Hummel

Abgabedatum: 15.08.2010

Erklärung

Ich versichere, dass ich diese Bachelorarbeit selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

München, den 15. August 2010

....

Martin Klenk

Abstract

Betrachtet man den gesamten Lebenszyklus einer Software, so stellt man fest, dass etwa 70% der gesamten Kosten auf Wartungsarbeiten entfallen. Die dabei zeitintensivste Aktivität stellt das Lesen und Erschließen von Quelltexten durch die mit der Wartung beschäftigten Entwickler dar. Damit stellt die Erhaltung und Verbesserung der Lesbarkeit von Quelltexten gleichzeitig einen wichtigen Qualitätsaspekt wie eine Voraussetzung dafür dar, dass Projekte im Budget abgeschlossen werden können.

In dieser Arbeit wird zunächst ein Modell entwickelt, wie Lesbarkeit und Verständlichkeit von Quelltexten evaluiert werden kann. Die Basis dieses Modells bilden dabei Kontroll- und Datenflussgraphen von einzelnen Methoden. Es werden eine prototypische Implementierung sowie die dabei verwendeten Algorithmen und Methoden vorgestellt und die Ergebnisse von verschiedenen Testläufen evaluiert.

Zum Schluss werden weitere Verwendungsmöglichkeiten und Erweiterungen vorgestellt, aber auch die Grenzen des Modells aufgezeigt.

Inhaltsverzeichnis

1 Einleitung **1**

1.1 Motivation . . . 1

1.2 Etablierte Metriken zur Messung von Lesbarkeit . . . 2

1.3 Problem . . . 2

1.4 Zielsetzung dieser Arbeit . . . 3

1.5 Ansatz . . . 3

2 Verwandte Arbeiten **5**

3 Lösungsansatz **7**

3.1 Definition von Verständniskontexten . . . 7

3.2 Modellbildung . . . 9

4 Implementierung **17**

4.1 Eingesetzte Technologien . . . 17

4.2 Allgemeine Vorgehensweise . . . 18

4.3 Lösungsverfahren . . . 18

5 Auswertung und Ergebnisse **27**

5.1 Ziele und untersuchte Fragestellungen . . . 27

5.2 Allgemeine Beobachtungen . . . 28

5.3 Detaillierte Auswertung . . . 28

6 Zusammenfassung **39**

7 Ausblick **41**

A Kontrollflussanweisungen im Graphen **43**

A.1 do...while-Schleife . . . 43

A.2 while-Schleife . . . 44

A.3 for-Schleife . . . 44

A.4 for-each-Schleife . . . 45

A.5 if-then-else-Verzweigung . . . 45

A.6 switch-Verzweigung . . . 46

A.7 synchronized-Block . . . 46

B Beispiel für eine LP-Beschreibung **49**

Kapitel 1

Einleitung

Der Lebenszyklus von Software ist ein mehrstufiger Prozess. ISO/IEC 12207 etwa beschreibt einen Rahmen für Softwarelebenszyklen [Sin95]. Die Primäraktivitäten bestehen darin aus Beschaffung (Acquisition), Lieferung (Supply), Entwicklung (Development), Betrieb (Operation) und Wartung (Maintainance). Insbesondere bei langfristig eingesetzten Systemen entfällt der größte Aufwand auf Wartung und Weiterentwicklung des Systems. Typische Aktivitäten hierbei sind das Aufspüren und Beheben von Fehlfunktionen sowie Anpassungen an neue oder geänderte Anforderungen. Die meisten der Aktivitäten erfordern vom eingesetzten Entwickler ein tiefes Verständnis für den Quellcode, den er bearbeiten muss.

1.1 Motivation

Da Wartungsarbeiten an Softwaresystemen einen beträchtlichen Kostenfaktor darstellen, wurde bereits viel Forschungsarbeit geleistet, um die Frage, wie die Wartbarkeit von Software nachhaltig verbessert werden kann, zu beantworten. Nach Boehm [BB01] fallen bis zu 70% der Kosten, die im Softwarelebenszyklus entstehen, in der Wartung des Systems an. Deimel [Dei85] stellte fest, dass eine der wichtigsten und aufwändigsten Einzelaktivitäten dabei das Lesen und Verstehen von Quelltext ist. Damit eine Software nachhaltig wartbar bleibt ist es notwendig, die Lesbarkeit von Quelltext permanent zu überwachen und zu verbessern. Andernfalls nehmen die Kosten mit der Laufzeit des Projekts stetig zu und sorgen im schlimmsten Fall zum Scheitern des Projekts.

1.2 Etablierte Metriken zur Messung von Lesbarkeit

Zur Implementierung einer effizienten und effektiven Qualitätsüberwachung einer Software werden Softwaremetriken eingesetzt [Kan02]. Ziel einer Softwaremetrik ist es, einzelne Qualitätsaspekte einer Software messbar und damit kontrollierbar zu machen. Da Lesbarkeit von Code ein entscheidender Faktor für den Erfolg eines Softwareprojekts ist, wurden hierfür bereits eine Reihe an Metriken entwickelt, die diesen Aspekt erfassen. Buse [BW08] etwa entwickelt eine solche Metrik. Hierbei wurden zunächst von 120 Testpersonen 100 Codeausschnitte bewertet. Anschließend wurden einzelne, automatisch messbare Eigenschaften von Code identifiziert und die Ausschnitte auf diese Eigenschaften hin analysiert. Analysiert wurden dabei beispielsweise die Anzahl an Leerzeilen oder der Anteil an Kommentaren. Die hierbei gewonnenen Daten wurden als Basisdaten in ein wissensbasiertes System eingespeist. Diese trainierte Software konnte damit mit einer Genauigkeit von 80% bestimmen, ob es sich um gut lesbaren Code handelt oder nicht. Ein solcher auf maschinellem Lernen aufbauender Ansatz ist laut Buse vielversprechend, da Lesbarkeit lediglich als zufälliges, nicht an einzelnen Faktoren festzustellendes Produkt in der Entwicklung entsteht. Ähnliche Ansätze sind seit einigen Jahrzehnten ebenfalls für natürliche Sprache im Einsatz. Das amerikanische Verteidigungsministerium etwa schreibt Auftragnehmern eine gewisse Mindestpunkzahl im Flesh-Kincaid Grade Level [Fle48] vor. Einen anderen Weg versuchen die beiden Metriken “Live Variables” und “Variable Spanning” zu gehen. Diese versuchen, Lesbarkeit und Verständnis von Code anhand der Komplexität des Datenflusses zu bestimmen [DG79].

1.3 Problem

Verfahren, wie von Buse [BW08] beschrieben, basieren auf der Klassifikation anhand einer Reihe von Parametern. Welche Parameter jedoch zu welcher Bewertung führen, ist im Einzelnen nicht nachvollziehbar. Daher ist es damit nicht möglich, einzelne Schwachstellen festzustellen und diese zu verbessern. Entsprechende Verbesserungsvorschläge, etwa die Durchführung von Refactorings [Fow99], können nicht automatisch erzeugt werden.

Der Ansatz von “Live Variables” und “Variable Spanning” soll in dieser Arbeit weiterverfolgt werden. Eine Rückführung der Messergebnisse auf den Code soll möglich sein. Bei diesen beiden Ansätzen handelt es sich lediglich um deskriptive Verfahren. Eine direkte Ableitung von Verbesserungen ist auch hierbei nicht möglich. Dennoch ist ein Vorteil dieser Verfahren, dass die Berechnung nachvollziehbar ist.

Dieser Vorteil soll im Rahmen dieser Arbeit genutzt werden, um auf Basis dieser Verfahren eine Methode zur automatischen Optimierung von Code zu entwickeln. Die hierzu notwendigen Methoden werden in dieser Arbeit entwickelt und getestet.

1.4 Zielsetzung dieser Arbeit

Diese Arbeit verfolgt einen Ansatz ähnlich zu “Live Variables” oder “Variable Spanning” zur Evaluation von Lesbarkeit und Verständlichkeit von Quelltexten. Lesbarkeit soll sich dabei nicht ausschließlich auf das reine Lesen des Quelltextes beschränken, sondern auch das Erschließen der Semantik des Codes berücksichtigen. Dieser Ansatz soll beispielhaft implementiert werden und damit erste Analysen durchgeführt werden. Das Modell soll eine direkte Rückführung von gewonnenen Erkenntnissen auf den Quelltext erlauben, und somit neben der reinen Analyse auch als Basis für eine proaktive Unterstützung des Entwicklers dienen können.

1.5 Ansatz

Im ersten Schritt wird in dieser Arbeit ein Modell entworfen, welches eine Eigenschaft, welche zum Verständnis von Code beiträgt, abbildet. Das Modell ist dabei anwendbar auf einzelne Methoden. Die Basis hierfür stellen Graphen dar, welche aus Daten- und Kontrollfluss einer Methode extrahiert werden können. Es wird eine Bewertungsmethode entworfen, sowie eine Strategie, durch die die Bewertung automatisch verbessert werden kann. Die Strategie basiert darauf, anhand des gewonnenen Graphen durch Vertauschungen von Statements eine optimale Reihenfolge der Statements zu erreichen und damit die Lesbarkeit der Methode zu erhöhen.

Kapitel 2

Verwandte Arbeiten

In der Arbeit von Dunsmore und Gannon [DG79] wird eine Methode vorgestellt, durch welche versucht wird, das Verständnis von Code durch die Anzahl der Variablen, welche im aktuellen Statement verwendet werden, zu modellieren. Hierzu wird für ein Statement eine Menge von Variablen definiert, die sog. „live variables“ (LV). Eine Variable wird zwischen der ersten und letzten Referenz innerhalb einer Prozedur „live“ genannt. In der Arbeit wird beobachtet, dass ein Entwickler, um den Quelltext zu verstehen, sich jener Variablen bewusst sein muss, welche aktuell in der Menge der „live variables“ enthalten sind. Die Methode, welche in der Arbeit vorgestellt wird, zählt in jedem Statement die Anzahl der Variablen, welche entweder vor und in dem Statement oder nach und in dem Statement oder vor und nach dem Statement oder nur in dem Statement aktiv sind. Anschließend wurde eine Studie durchgeführt welche untersuchte, wie sich eine unterschiedliche Anzahl von LVs per Statement (LV/S) auf Verständlichkeit und Änderbarkeit auswirken. Hierzu wurden zunächst allgemeine Fragen zum Programmverständnis gestellt und anschließend drei Anweisungen gegeben, was an dem Programm geändert werden soll. Es ergaben sich dabei zwar ähnliche Werte im Programmverständnis für unterschiedliche LV/S, dies kann jedoch auf die Art der Fragestellung zurückgeführt werden. Bei den Modifikationen schnitten die Teilnehmer, welche ein Programm mit niedrigen LV/S verändern sollten, wesentlich besser ab und setzten die Anforderungen mit deutlich weniger Veränderungen um als die andere Gruppe.

In [XSZC00] wird festgestellt, dass, obwohl LV nicht für objektorientierte Sprachen entwickelt wird, dennoch einfach auf objektorientierte Sprachen angewendet werden kann.

Raymond [Ray91] beschäftigt sich mit der allgemeinen Lesbarkeit von Quelltext und entwickelt eine Reihe an prototypischen Quelltextansichten, welche anschließend evaluiert werden. Eine interessante Erkenntnis dieser Arbeit ist, dass Code insbesondere dann effizient lesbar ist, wenn die relevanten Teile des Codes so eng wie möglich gruppiert sind. Dies wird in der Arbeit durch verschiedene Darstellungsformen, etwa unterschiedlich große Schriftgrößen versucht umzusetzen. Alle Darstellungsformen haben das Ziel, die

sog. „locality“ (Lokalität) zu erhöhen, wobei diese sich in der Arbeit nicht ausschließlich auf Code sondern ebenfalls auf die damit verknüpften Dokumentationsartefakte bezieht. Eine hohe Lokalität ist dann gegeben, wenn die relevanten Teile des Codes sowohl physisch als auch konzeptionell möglichst nahe beieinander auftreten [Ray91].

Eine Reihe weiterer Arbeiten beschäftigt sich mit dem Einfluss, welchen Lesbarkeit und Verständlichkeit von Code auf Wartungsarbeiten hat. Deimel [Dei85] etwa schlägt vor, dass auf Grund der Wichtigkeit von Lesbarkeit bereits in der Ausbildung Zeit darauf verwendet wird, Lesen von Code zu trainieren.

Der Fokus dieser Arbeit liegt auf der Optimierung der Lesbarkeit von Code auf Basis dieser Erkenntnisse. Es wird dabei versucht, die Lokalität innerhalb von Methoden zu optimieren, dabei jedoch nicht auf die Veränderung der Darstellungsform gesetzt. Stattdessen wird die Struktur des Codes selbst optimiert.

Kapitel 3

Lösungsansatz

Dieser Abschnitt beschreibt zunächst das Modell sowie die dafür notwendigen Annahmen und Definitionen. Anschließend werden die Kernkonzepte dargestellt, die die Basis für den Lösungsansatz bilden. Zum Schluss wird eine detaillierte Beschreibung des Problems geliefert, welches gelöst werden soll.

3.1 Definition von Verständniskontexten

Verschiedene Arbeiten haben bereits eine Reihe von Messgrößen vorgeschlagen, mit denen die Lesbarkeit von Code abgebildet werden kann. Manche Arbeiten schlagen hierfür etwa die Verwendung von angepassten Werkzeugen vor, die durch Vergrößerung von Schrift oder Hervorhebung von besonders relevanten Codestücken eine verbesserte Lesbarkeit erreichen [Ray91]. Auch Kommentaranzahl, Variablenbenennung, Einrückung und die verwendete Programmiersprache selbst haben einen großen Einfluss auf die Lesbarkeit von Code sowie den damit verbundenen Wartungsaufwand.

Damit ein Programmierer oder auch ein Reviewer beim Lesen von Code die einzelnen Statements versteht, ist es notwendig, das Statement in den gesamten Zusammenhang einordnen zu können. [Dei85] etwa stellt fest, dass das größte Problem für weniger erfahrene Programmierer beim Lesen von Code das Erkennen dieser Zusammenhänge und das Bilden von Sinneinheiten darstellt. In einem Programm stellen die Einheiten, welche verstanden werden müssen, die entsprechenden Einheiten der verwendeten Programmiersprache dar. Dies können beispielsweise Verzweigungen, Methodennamen, Bezeichner, Schleifen aber auch Schlüsselwörter oder Codeblöcke sein.

Ein erster Schritt, durch welchen das Verständnis der einzelnen Einheiten verbessert werden kann, kann eine sinngemäße Einrückung von Quelltextzeilen oder auch eine gute Wahl der Bezeichner und Variablennamen sein.

Das Verständnis der einzelnen Bezeichner ist notwendig zum Verständnis von Code, allerdings nicht hinreichend um den Code in seinem Kontext richtig einordnen zu können. Algorithmen etwa bestehen aus vielen Zeilen Code, die erst als Ganzes die entsprechenden Berechnungen korrekt durchführen. Auch zur Implementierung einer Benutzeroberfläche ist es notwendig, zu wissen auf welches Panel etwa welcher Knopf gezeichnet werden soll. In jedem Fall muss ein Entwickler, der Code verstehen oder sogar verändern soll, sich diese Informationen erarbeiten, bevor er seinen Aufgaben nachkommen kann.

Dies wird umso schwieriger, je überlagerter einzelne, sinngemäß zusammengehörige Blöcke sind. Der Entwickler muss einen unnötig großen Verständniskontext aufbauen, damit die entsprechenden Anweisungen verstanden werden können.

Im Folgenden wird ein Modell konstruiert, dessen Ziel es ist, derartige Verschränkungen zu erfassen und zu optimieren. Hierzu wird eine optimale Reihenfolge für die einzelnen Anweisungen einer Methode berechnet, welche die Verschränkungen minimiert und damit die Größe des Verständniskontextes minimiert.

Das Modell versucht die folgenden beiden Anforderungen abzubilden und umzusetzen:

- Die Lokalität des Codes soll möglichst groß sein, um gute Lesbarkeit des Codes zu erreichen. Dies wurde bereits in [Ray91] bestätigt.
- Der Code soll minimale Verschränkungen aufweisen. Wenn zwei unabhängige Codestücke sich überschneiden, dann wird dadurch die Lokalität verschlechtert. Dementsprechend ist Code, in dem viele Verschränkungen auftreten, schlechter lesbar und kann weniger gut verstanden werden. Diese Annahme wurde bereits in [DG79] anhand einer Studie überprüft.

abzubilden und umzusetzen. Damit das Modell entsprechend angewendet werden kann, müssen außerdem noch eine Reihe technischer Annahmen getroffen werden, welche eine Abbildung von Quellcodecode ermöglichen:

- Es werden die Dateneinheiten analysiert, welche durch Variablen repräsentiert werden.
- Um eine Verbesserung zu erreichen, werden zulässige Veränderungen am Code vorgenommen. Die Veränderungen am Code werden auf der Ebene von Statements durchgeführt. Ein Veränderungsschritt ist die Vertauschung zweier Statements.
- Eine Vertauschung soll lediglich die Struktur, nicht aber die Semantik der Methode verändern. Seiteneffekte, beispielsweise Ein- und Ausgabe, werden nicht beachtet.
- Ruft eine Methode eine andere Methode auf, so ist nicht bekannt, welche Variablen in der aufgerufenen Methode verändert werden.
- Verzweigungen im Kontrollfluss durch Exceptions werden nicht beachtet.

- Ein Codeblock, der n Statements enthält, wird im übergeordneten Block als n einzelne Statements in der originalen Reihenfolge behandelt.

Mit diesen Vorgaben kann nun mit der Konstruktion eines Modells begonnen werden.

3.2 Modellbildung

Die Anforderungen müssen auf ein formales Modell abgebildet werden, welches als Basis für die Optimierung dient. Im Kern des Modells steht die Konstruktion eines Datenflussgraphen pro Methode. Jeder Knoten in diesem Datenflussgraphen entspricht dabei einer Zeile Code. Aus dem Graphen lässt sich ableiten, welche Statements vertauscht werden dürfen und welche in ihrer Reihenfolge unverändert bleiben müssen.

3.2.1 Live Variables und Variable Spanning als Grundlage

Die Grundlage für die weiteren Überlegungen stellen die Erkenntnisse aus der Live Variables Metrik sowie die Spannweite von Variablen dar. Als Spanning wird der Abstand zwischen der ersten Verwendung und der letzten Verwendung einer Variablen bezeichnet. Folgender Codeausschnitt verdeutlicht dies:

```
int a = 23; // Beginn Verwendung von a, Position 1
int b = 42; // Beginn Verwendung  von b, Position 2
a = a*a;    // Letzte Verwendung von a, Position 3
//... n stmts
b = b*b;    // Letzte Verwendung von b, Position n+4
```

Die Spannweite berechnet sich für dieses Snippet für die Variable a als

$$span(a) = 3 - 1 = 2$$

und für die Variable b als

$$span(b) = n + 4 - 2 = n + 2$$

Live Variables und Variable Spanning bildet die Anforderungen bereits in einfacher und nachvollziehbarer Weise ab, muss aber noch erweitert werden, damit eine Optimierung der Größe stattfinden kann. Vergleich 1 stellt beispielhaft einen kleinen Codeausschnitt mit gleicher Semantik aber unterschiedlichen Live Variables und Spanning dar. Wie hier erkennbar ist, ist die Lokalität im rechten Snippet wesentlich höher als im linken Snippet. Besonders deutlich ist dies beim Spanning erkennbar. Deklaration, Initialisierung und Verwendung der beiden Variablen stehen näher zusammen; dadurch ist der Zweck der beiden Variablen klarer erkennbar.

Auf Grund der einfacheren Umsetzbarkeit wird als Basis dieser Arbeit **Variable Spanning** als zu optimierende Größe ausgewählt. Eine ähnliche Implementierung wäre allerdings auch für Live Variables denkbar.

`int a = 23; // 1`
`int b = 42; // 2`
`a = a*a;    // 2`
`//... n stmts`
`b = b*b;    // 1`
	`int a = 23; // 1`
	`a = a*a;    // 1`
	`//... n stmts`
	`int b = 42; // 1`
	`b = b*b;    // 1`

$\sum_{live} = 6$ $span(a) = 2$ $span(b) = n + 2$	$\sum_{live} = 4$ $span(a) = 1$ $span(b) = 1$

Vergleich 1: Eine einfache Gegenüberstellung von zwei Codesnippets mit unterschiedlicher Anzahl an Live Variables und unterschiedlichem Spanning. Die Zahl neben jeder Zeile gibt die Anzahl an Live Variables an.

3.2.2 Generelle Vorgehensweise

Die generelle Vorgehensweise, um zu einer möglichst geringen Verschränkung zu gelangen, besteht darin, so lange erlaubte Vertauschungen von Statements vorzunehmen, bis ein minimales Spanning erreicht wird. Die erlaubten Vertauschungen lassen sich dabei aus einem erweiterten Datenflussgraphen bestimmen, aus dem die Abhängigkeiten abgelesen und damit die erlaubten Vertauschungen generiert werden können.

In dem Graphen ist nicht ausschließlich der Datenfluss der Methode repräsentiert, sondern ebenfalls strukturelle Abhängigkeiten. Ein Beispiel für eine strukturelle Abhängigkeit ist etwa die atomare Behandlung eines Codeblocks oder die Abbildung einer Verzweigung im Kontrollfluss.

Im Folgenden soll die Gewinnung des Graphen aus dem Code einer Methode beschrieben werden.

3.2.3 Abbildung von Methoden auf Abhängigkeitsgraphen

Um die Abhängigkeiten zwischen verschiedenen Statements zu repräsentieren, wird ein Graph $D = (V, E)$ mit Knoten V und Kanten E generiert. Ein solcher Graph kann für jede Methode berechnet werden. Eine Methode setzt sich aus einer Reihenfolge von Statements zusammen. Zu jedem dieser Statements $S = \{s_0, s_1, \ldots, s_n\}$ existiert ein Knoten im Graphen. Die Menge der Knoten des Graphen sind also $V = \{v_0, v_1, \ldots, v_n\}$.

Ein Statement kann auf eine beliebige Zahl an Variablen zugreifen. Die Datenabhängkeiten entstehen durch eben diese Zugriffe. Es wird zwischen zwei verschiedenen Arten von Zugriffen unterschieden: Lesen und schreiben. Für diese Zugriffe werden zwei Funktionen definiert, die für einen Knoten v die entsprechende Menge betroffener Variablen berechnen. Sei die Menge der Variablen $VARS$. Die Funktion $w : V \mapsto \mathcal{P}(VARS)$ liefert

die Menge der im Knoten $v \in V$ geschriebenen Variablen. Analog liefert die Funktion $r : V \mapsto \mathcal{P}(VARS)$ die Menge der gelesenen Variablen. Die Funktionen können ebenfalls als Attribute des Knotens interpretiert werden.

Es gilt also

- $w(v_k) = \{var_0, var_1, \ldots, var_i\}$ gdw. in Anweisung k den Variablen var_1, var_2, ..., var_i ein neuer Wert zugewiesen wird, es sich dabei um ein Objekt handelt auf dem eine Methode aufgerufen wird, eine lokale Variable deklariert wird oder eine Klassenvariable, eine Objektvariable oder ein Argument der Methode zum ersten Mal verwendet wird. Methodenaufrufe werden in die Menge der geschriebenen Variablen aufgenommen, da nicht entschieden werden kann, ob ein Methodenaufruf nur lesend auf Eigenschaften des Objekts zugreift oder diese auch verändert.

- $r(v_k) = \{var_0, var_1, \ldots, var_i\}$ gdw. in Anweisung k ausschließlich lesend auf die Variablen var_1 , var_2, ..., var_i zugegriffen wird.

Nun sollen die Kanten des Datenflussgraphen beschrieben werden. Informell wird von jedem Statement, welches eine Variable beschreibt zu jedem Statement, welches anschließend auf diese Wert zugreift, eine Kante im Graphen eingefügt. Diese Kanten werden in der Menge E_1 beschrieben. Formal gilt

$$E_1 = \left\{ (v_i, v_k) \;\middle|\; \begin{array}{ll} & k > i \\ \wedge & w(v_i) \cap r(v_k) = CVars \neq \emptyset \\ \wedge & \exists var \in CVars : \forall t | i < t < k : var \notin w(v_t) \end{array} \right\}$$

Außerdem gibt es wenn in einem Knoten v_i einer Variablen ein neuer Wert zugewiesen wird, eine Kante von jedem Knoten, der den vorherigen Wert der Variablen gelesen hat. Diese Kanten werden in der Menge E_2 beschrieben. Es gilt also

$$E_2 = \left\{ (v_i, v_k) \;\middle|\; \begin{array}{ll} & k > i \\ \wedge & r(v_i) \cap w(v_k) = CVars \neq \emptyset \\ \wedge & \exists var \in CVars : \forall t | i < t < k : var \notin w(v_t) \end{array} \right\}$$

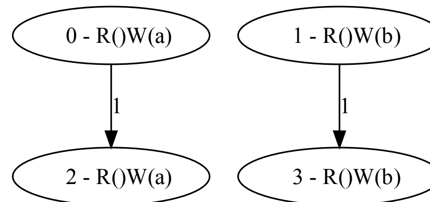
Weitere Kanten werden benötigt, um die Struktur der Sprache zu repräsentieren, damit die gültigen Transformationen auch im Sinne der Sprachdefinition gültig sind. So darf beispielsweise ein If-Then-Else-Konstrukt nicht durch andere Statements geteilt werden. Eine komplette Beschreibung, welche Kanten zusätzlich eingefügt werden, findet sich in Anhang A. Die Menge dieser „künstlichen“ Kanten sei E_3.

Mit diesen drei Mengen lässt sich die Menge E beschreiben als

$$E = E_1 \cup E_2 \cup E_3$$

Wie bereits erwähnt werden Blöcke lediglich als ein Knoten betrachtet. Für den Knoten, der einen Block repräsentiert, wird als gelesene/geschriebene Variablen eine Vereinigung aller in diesem Block gelesenen/geschriebenen Variablen angenommen. Der auf diese Art und Weise gewonnene Graph modelliert alle oben angeführten Anforderungen.

```
simpleMethod() {
  int a = 23;
  int b = 42;
  a = a * a;
  b = b * b;
}
```



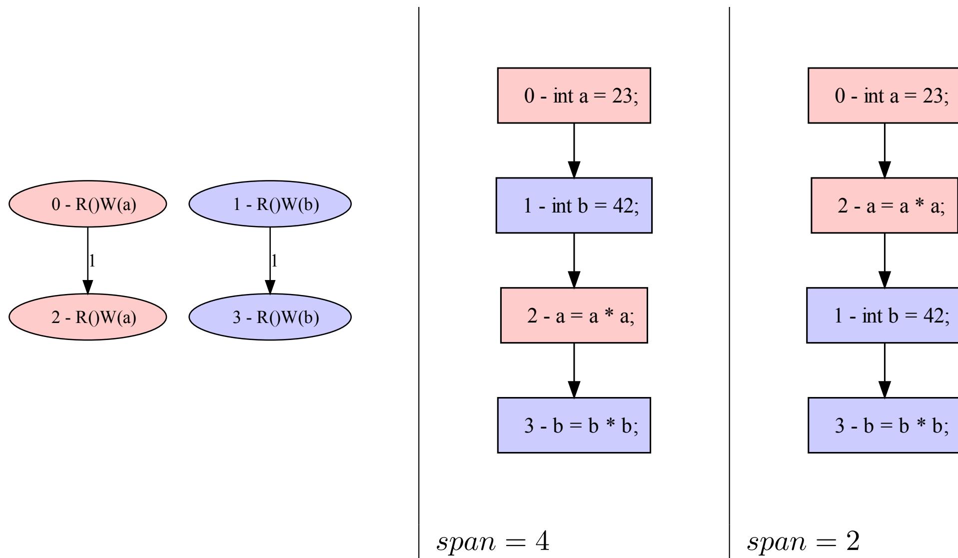
Graph 1: Eine einfache Methode mit dem dazugehörigen Datenflussgraphen. N_i entspricht dem Statement i, $R(\ldots)$ ist die Menge der in dem Statement gelesenen Variablen, $W(\ldots)$ die Menge der in dem Statement geschriebenen Variablen. Das Gewicht an den Kanten gibt die Anzahl an Variablen an, die für die Abhängigkeit verantwortlich sind.

Graph 1 stellt die Erzeugung der Graphen an einem Beispiel dar. Wie in dem Graphen bereits zu erkennen ist, besteht eine Abhängigkeit zwischen den beiden Statements 1 und 3 sowie den beiden Statements 2 und 4. Die beiden Gruppen wurden jeweils auf Grund der Verwendung einer gemeinsamen Variablen gebildet.

Der Graph enthält damit nun alle Informationen, welche zur Berechnung einer optimalen Reihenfolge benötigt werden. Führt ein Pfad von v_i zu v_j, dann muss v_i in der berechneten Transformation vor v_j stehen. Gibt es keinen Pfad von v_i zu v_j, dann können die beiden Statements, die durch diese Knoten repräsentiert werden, vertauscht werden.

Das Ziel der Methode soll sein, die Statements so neu anzuordnen, dass das Spanning der Variablen minimiert wird. Dabei muss die Semantik der Methode erhalten bleiben. Die Bedingungen für gültige Anordnungen lassen sich aus dem erzeugten Datenflussgraphen ablesen. Übertragen auf den Kontrollflussgraphen wird damit versucht, Zusammenhangskomponenten im Datenflussgraphen auf den Kontrollflussgraphen zu übertragen und die zur Zusammenhangskomponente gehörigen Statements so lokal wie möglich zu halten.

Graph 2 verdeutlicht die Idee anhand der Beispielmethode. Zunächst wird der Datenflussgraph der Methode berechnet. In diesem Graphen sind die Zusammenhangskomponenten jeweils in der selben Farbe eingefärbt. Im originalen Kontrollflussgraphen ist erkennbar, dass sich die beiden Zusammenhangskomponenten überschneiden, wodurch ein sehr hohes und unerwünschtes Spanning erreicht wird. Nach der Optimierung stehen die Knoten, welche zu den Zusammenhangskomponenten gehören, kompakt. Dadurch wird das Spanning deutlich reduziert.



Graph 2: Datenflussgraph und dazugehöriger Kontrollflussgraph der Beispielmethode vor und nach der Optimierung

Mit dem selben Prinzip lassen sich auch komplexere Methoden analysieren und optimieren. Selbst wenn keine getrennten Zusammenhangskomponenten im Datenflussgraphen gefunden werden, lässt sich dennoch durch eine geschickte Anordnung eine Verbesserung erreichen. Graph 3 stellt ein Beispiel für eine etwas komplexere Methode dar.

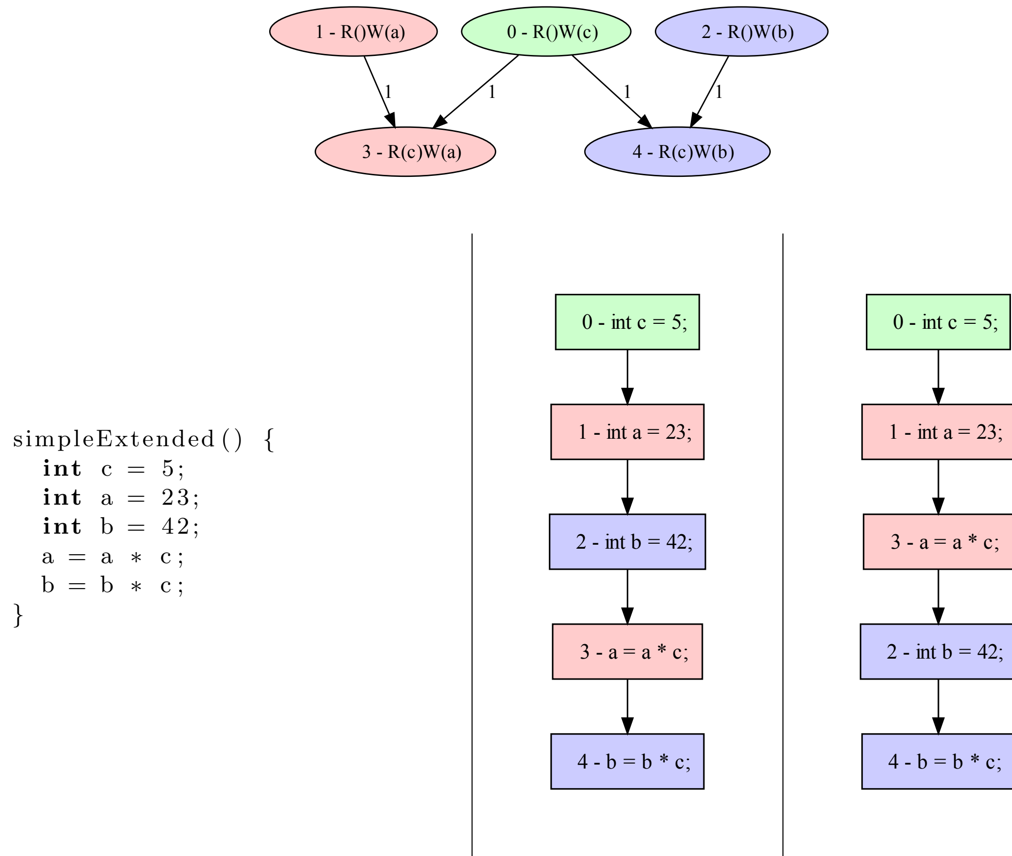
3.2.4 Definition einer auswertbaren Kostenfunktion

Um eine Methode zu optimieren ist es notwendig, die zu optimierende Kostenfunktion formal zu definieren. Im vorherigen Teil wurde das Spanning von Variablen als Kostenfunktion gewählt, dies soll nun etwas abgeändert werden und anschließend in der Optimierung eingesetzt werden.

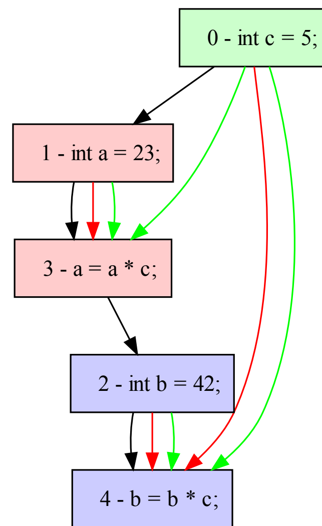
Beim der Berechnung des Variable Spanning wird lediglich die gesamte Lebensspanne einer Variablen betrachtet. Zur Bewertung der Kosten soll jedoch auch die Verteilung der Statements innerhalb der Lebensspanne beachtet werden. Außerdem sollen lokale Optimierungen durchgeführt werden.

Als Basis für die Berechnung der Kosten dient wieder der Datenflussgraph der Methode. Informell werden die Kanten des Datenflussgraphen in den Kontrollflussgraphen projeziert und deren Länge gemessen. Im Unterschied zum klassischen Variable Spanning wird also nicht lediglich der Abstand zwischen erster und letzter Verwendung der Variablen gemessen. Graph 4 stellt die beiden Ansätze zur Verdeutlichung gegenüber.

Durch die Anpassung der Optimierung wird versucht, die Verwendungen einer Variablen



Graph 3: Der Datenflussgraph sowie nicht optimierter und optimierter Kontrollflussgraph einer erweiterten Methode.



Graph 4: Der Kontrollflussgraph (schwarz) mit Kanten für Variable Spanning (rot) sowie den Kanten des Datenflussgraphen (grün). Die Knotenfärbung entspricht den aus dem Datenflussgraphen gewonnenen Gruppen.

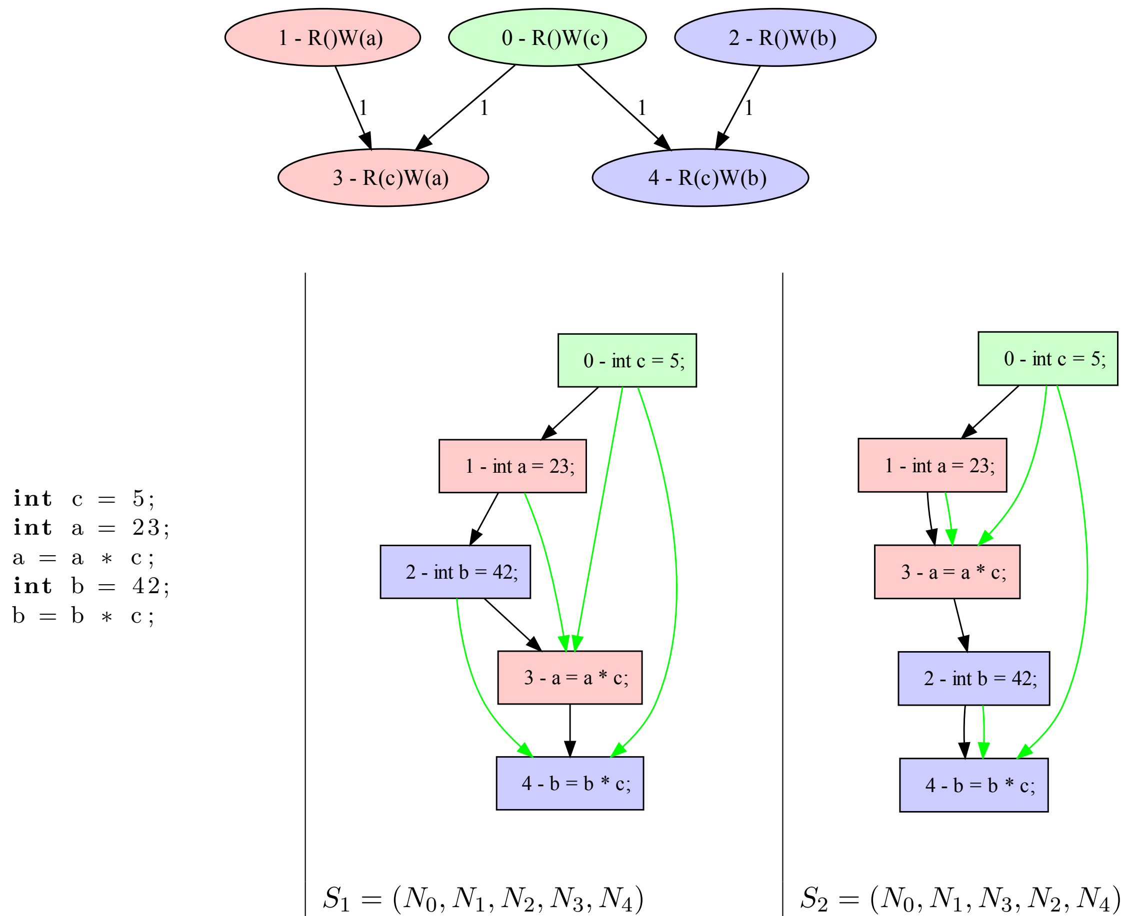
so nah wie möglich an die vorhergehende Definition der Variablen zu ziehen. Dies wird beim klassischen Variable Spanning nicht in dieser Form erreicht. Zur Auswertung muss die verwendete Kostenfunktion nun noch formal definiert werden:

Sei $\pi : S^n \mapsto S^n$ eine Permutation der Statements S. Sei außerdem $pos : S^n \times V \mapsto \mathbb{N}$ eine Funktion, die die Position des Statements in der neuen Permutation zuweist. Außerdem sei $cost : S^n \mapsto \mathbb{N}$ die Funktion, die einer Permutation ihre Kosten zuweist. Dann ist

$$cost(S_p) = \sum_{(v_{i_1}, v_{i_2}) \in E} pos(S_p, v_{i_2}) - pos(S_p, v_{i_1})$$

für eine Permutation S_p der Statements. *cost* ist die Funktion, die minimiert werden soll. Wird eine minimale Lösung gefunden, so stellt die dazu berechnete Permutation eine optimale Reihenfolge der Statements bezüglich der gewählten Kriterien dar. Diese Lösung kann direkt verwendet werden, um die Methode zu transformieren. Im Folgenden wird die im Rahmen dieser Arbeit erarbeitete prototypische Implementierung beschrieben, die diese Optimierung durchführt.

Folgendes Beispiel illustriert die Funktionsweise der Kostenfunktion:



Die Kanten des Kontrollflussgraphen (schwarz) geben die Ausführungsreihenfolge vor, die Kanten des Abhängigkeitsgraphen (grün) stellen die Abhängigkeiten zwischen den Statements dar. Die Kostenfunktion lässt sich aus den Abhängigkeitsgraphen aufstellen als

$$cost(S_p) = \underbrace{p(N_3) - p(N_0)}_{(N_0,N_3)} + \underbrace{p(N_4) - p(N_0)}_{(N_0,N_4)} + \underbrace{p(N_3) - p(N_1)}_{(N_1,N_3)} + \underbrace{p(N_4) - p(N_2)}_{(N_2,N_4)}$$

was den Längen der Kanten entspricht. Wird die in den Beispielgraphen dargestellte Permutation ausgewertet, dann ergibt sich

$$cost(S_1) = (3 - 0) + (4 - 0) + (3 - 1) + (4 - 2) = 11$$

$$cost(S_2) = (2 - 0) + (4 - 0) + (2 - 1) + (4 - 3) = 8$$

mit dem gewünschten Ergebnis, dass S_2 die bessere Ausführungsreihenfolge darstellt.

Kapitel 4

Implementierung

Im Folgenden werden die im Rahmen dieser Arbeit verfolgten Ansätze zur Implementierung der Optimierung vorgestellt und diskutiert. Die Herangehensweise wird ebenso beschrieben wie die zur Umsetzung notwendigen Formulierungen und Transformationen des Problems. Zunächst werden zwei Ansätze vorgestellt, die eine kombinatorische Lösung berechnen. Anschließend wird das Problem als lineares Programm formuliert und mit Hilfe von verfügbaren Solvern gelöst.

4.1 Eingesetzte Technologien

Die verwendeten Technologien sind bei allen hier vorgestellten Lösungsstrategien ähnlich. Als Zielsprache der Analyse wurde Java ausgewählt. Für Java existieren bereits eine Reihe an gut dokumentierten und leistungsfähigen Tools, welche den technischen Teil der Implementierung deutlich vereinfachen. Die implementierten Algorithmen wurden in das an der TU München entwickelte Analyseframework ConQAT integriert. Hierdurch kann auf Funktionen, etwa zum Einlesen von Code, zurückgegriffen werden. Des Weiteren können mit Hilfe von ConQAT übersichtliche Ausgabeseiten erstellt werden, die eine Evaluation der Ergebnisse wesentlich vereinfachen.

Als Basis für den Aufbau der Graphen wird ein abstrakter Syntaxbaum der Methode benötigt. Für Java existieren eine Reihe an Compilern, die diese Funktion zur Verfügung stellen. In dieser Arbeit wurde der Compiler der Eclipse Foundation verwendet, der ebenfalls in den weit verbreiteten Java Development Tools zum Einsatz kommt. Auf Grund der relativ großen Verbreitung existiert für diesen Compiler eine ausreichend ausführliche Dokumentation.

4.2 Allgemeine Vorgehensweise

Der Verarbeitungsprozess kann auf einem hohen Level in folgende Schritte untergliedert werden:

- Suchen und Einlesen des zu untersuchenden Codes
- Erzeugen eines Syntaxbaumes für den Code
- Annotation der Statements mit den Lese- und Schreibmengen und Berechnung des Abhängigkeitsgraphen
- Ausführen der Optimierung auf Basis des Abhängigkeitsgraphen
- Erzeugen der Ausgabe

Die ersten beiden sowie der letzte Schritt sollen im Rahmen dieser Arbeit nicht weiter interessieren, da diese bereits implementiert sind. Zur Annotation der Statements liefert der Eclipse-Compiler bereits eine Reihe von Funktionen. Berechnet werden die Mengen durch Traversierung des Syntaxbaumes. Aus der Liste der annotierten Statements kann durch Hinzufügen der oben beschriebenen Kanten der Abhängigkeitsgraph aufgebaut werden. Der Fokus liegt im Folgenden auf der Ausführung der Optimierung.

4.3 Lösungsverfahren

In diesem Abschnitt werden die verschiedenen Lösungsverfahren vorgestellt und diskutiert. Es wird auf Probleme eingegangen, die im Rahmen der Implementierung aufgetreten sind und die Grenzen der vorgestellten Verfahren erörtert. Zum Schluss wird das letztlich ausgewählte Verfahren ausführlich besprochen.

4.3.1 Generelle Schwierigkeiten bei der Implementierung

Da das zu lösende Optimierungsproblem kombinatorischer Art ist, ist die Implementierung einer Lösungsstrategie nicht trivial. Bevor mit der Implementierung begonnen wurde, wurden daher eine Reihe von Instanzen betrachtet. Auf dieser Basis wurde anschließend mit der Implementierung begonnen. Hierbei wurde angenommen, dass die Anzahl an möglichen Kombinationen ausreichend klein ist, was sich in der Praxis für die meisten Methoden auch bestätigt hat. Einige Methoden konnten jedoch auf Grund der großen Anzahl an Kombinationen mit keinem der evaluierten Verfahren komplett optimiert werden.

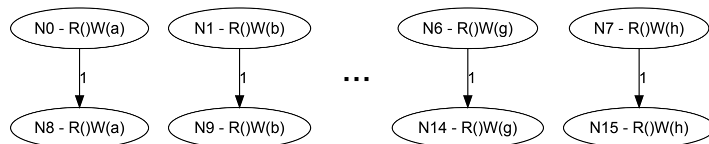
4.3.2 Verworfene Lösungsstrategien

Im Rahmen dieser Arbeit wurden eine Reihe an Lösungsstrategien entwickelt und auf Grund nicht hinreichender Leistungsfähigkeit verworfen.

Aufzählung von Permutationen

Die als erstes evaluierte Lösungsstrategie versucht, durch ein Aufzählen aller gültigen Permutationen auf Basis des Graphen und der anschließenden Berechnung der Kosten zu einem optimalen Ergebnis zu kommen. Dieser Ansatz ist die naheliegendste Lösung und wurde unter der Annahme implementiert, dass die Graphen ausreichend klein sind. Gibt es wenige Möglichkeiten, den Graphen zu traversieren, dann liefert dieses Verfahren in sehr kurzer Zeit eine optimale Lösung. Ein Vorteil dieses Verfahren ist, dass neben dem minimalen und dem realen Wert auch der maximale Wert der Methode berechnet werden kann.

```
permutableTest() {
  int a = 0;
  int b = 0;
  int c = 0;
  int d = 0;
  int e = 0;
  int f = 0;
  int g = 0;
  int h = 0;
  a = a * a;
  b = b * b;
  c = c * c;
  d = d * d;
  e = e * e;
  f = f * f;
  g = g * g;
  h = h * h;
}
```



Graph 5: Ein konstruiertes Beispiel für eine Methode für die durch Aufzählen der Permutationen keine optimale Lösung berechnet werden konnte.

Ein Problem dieser Methode ist, dass es nur auf eine Teilmenge der Methoden angewendet werden kann. Methoden, die viele verschiedene Permutationen und damit viele verschiedene Ausführungsreihenfolgen enthalten, sind im Rahmen der Optimierung die interessanteren. Die Anzahl der Permutationen dieser Methoden ist jedoch zu hoch, als dass durch ein reines Aufzählen eine Lösung gefunden werden kann. Eine Methode mit n Statements verfügt über $\mathbb{O}(n!)$ Permutationen. Selbst für kleine n ist diese Zahl bereits

zu groß. Ein konstruiertes, sehr einfaches Beispiel ist in Graph 5 dargestellt. Bereits für dieses Beispiel konnte keine Lösung mehr berechnet werden.

Auf Grund der mangelhaften Leistungsfähigkeit wurde der Ansatz verworfen.

Anwendung einer Greedy-Heuristik

Um für alle untersuchten Methoden zumindest ein Ergebnis zu erreichen, wurden eine Greedy-Heuristik implementiert. Die Heuristik findet einfache Verbesserung und kann erweitert werden, damit auch weitere Verbesserungen gefunden werden können. Das Prinzip der Heuristik basiert darauf, aus einer Ausführungsreihenfolge durch eine Vertauschung eine günstigere Reihenfolge zu erzeugen. Hierzu werden, beginnend mit der Ausgangsreihenfolge, alle möglichen Vertauschungen bewertet. Anschließend wird die Vertauschung, die die größte Verbesserung bringt, gewählt und durchgeführt. Dies wird so lange iteriert, bis keine Verbesserungen mehr gefunden werden.

Mit dieser Heuristik kann für einfache Graphen eine optimale Lösung gefunden werden, allerdings lässt sich nicht zuverlässig überprüfen, ob die gefundene lokale Lösung auch global optimal ist. Hierzu müsste eine untere Schranke berechnet werden, was mit dieser Methode nicht möglich ist. Des weiteren lassen sich einfache Beispiele konstruieren, für die diese Heuristik keine guten Ergebnisse liefert.

Das Problem der Heuristik ist, dass immer nur einzelne Statements vertauscht werden können. Häufig müssen jedoch Gruppen von Statements vertauscht werden. Diese werden von der Heuristik nicht erkannt. Eine mögliche Erweiterung der Heuristik wäre die Berechnung von Vorausschaukosten. Hierzu könnten, anstatt nur eine einzelne Vertauschung zu betrachten, k Vertauschungen betrachtet und bewertet werden. Für sinnvolle Werte von k ist der Aufwand zur Berechnung der Vorausschau bereits so hoch, dass relativ einfache Methoden nicht mehr zuverlässig optimiert werden können.

Für $k = 1$ lässt sich sehr leicht ein erweiterbares Gegenbeispiel konstruieren. Es müssen im Code immer Blöcke der Länge $k + 1 = 2$ gebildet werden.

```
int i = 2;
i = i * i;
int b = 3;
b = b * b;
i = i - 2;
i = i * i;
```

Um eine Verbesserung der Kosten zu erreichen, müssen mindestens zwei Statements vertauscht werden. Anderenfalls werden die Kosten lediglich an eine andere Stelle verschoben.

Ein Vergleich der Verfahren hat ergeben, dass mit den im Folgenden vorgestellten Verfahren durchweg bessere Ergebnisse erzielt werden können, weshalb dieser Ansatz ebenfalls verworfen wurde.

4.3.3 Einsatz eines ILP-Solvers

Da durch die beiden beschriebenen einfache Ansätze keine zufriedenstellende Ergebnisse erreicht werden konnten, wurde ein Adapter zur Verwendung von ILP (Integer Linear Program)-Solvern implementiert. Ein ILP ist ein mathematisches Programm eines ganzzahligen Optimierungsproblems. Durch die Verwendung eines Solvers kann auf sehr leistungsfähige Lösungsverfahren für derartige Probleme zurückgegriffen werden, wodurch eine wesentlich höhere Performance erwartet werden kann. Zunächst wurden die Bedingungen und die Kostenfunktion neu formuliert, um ein solches Programm aus dem Abhängigkeitsgraphen zu erzeugen. Anschließend wurden mit diesen Programmen die frei verfügbaren Solver LPSolve, Symphony und SCIP sowie der kommerzielle Solver cplex evaluiert. Pro Probleminstanz wurde ein Zeitrahmen von maximal 5 Minuten vorgegeben. Wurde bis zu diesem Zeitpunkt keine Lösung berechnet, dann wurde der als obere Schranke der Lösung berechnete Wert verwendet. Diese Werte wurden verglichen und auf Basis des Vergleichs eine Entscheidung gefällt.

4.3.4 Notwendige Neuformulierung des Problems

Damit eine Methode mit einem ILP-Solver optimiert werden kann, müssen die Bedingungen und die Kostenfunktion als ILP formuliert werden. Im Allgemeinen haben ILPs die Form $\min\{cx|Ax \leq b\}$. x ist der Vektor, der die Variablen enthält, c ist ein Vektor, der die Koeffizienten für die Kostenfunktion enthält, A enthält die Koeffizienten für die Ungleichungen, b enthält die rechten Seiten der Ungleichungen. Bei einer Programmierung ist darauf zu achten, dass Ungleichungen der Form $a \neq b$ nicht möglich sind, diese jedoch benötigt werden, um sicherzustellen, dass es sich bei der Lösung um eine Permutation handelt. Um dies zu realisieren wird eine Matrix verwendet. Die Einträge der Matrix stellen die Zuweisung von Statements auf Zeilen dar. Die Einschränkungen werden als Ungleichungen unter Verwendung der Variablen formuliert.

Sei

$$P = \begin{pmatrix} x_{1,1} & \cdots & x_{1,n} \\ \vdots & \ddots & \vdots \\ x_{n,1} & \cdots & x_{n,n} \end{pmatrix} \in \{0,1\}^{n\times n}$$

eine Matrix. Ist ein Eintrag

$$x_{i,j} = 1$$

dann wird Statement i der j-ten Stelle der Ausführungsreihenfolge zugewiesen.

Zur Durchführung der Optimierung müssen die Einschränkungen aus dem Graphen in diese Form übersetzt werden. Es werden drei Arten von Einschränkungen unterschieden.

Einmalige Zuordnung von Statements an Positionen

Da eine Permutation einer Methode berechnet werden soll, muss jedes Statement genau einer Position zugewiesen sein. Mit den Variablen der Matrix P lassen sich die folgenden beiden Bedingungen formulieren:

$$loa_j : \sum_{i=1}^{n} x_{i,j} = 1 \quad \forall j \in \{1, \ldots, n\}$$

Informell ausgedrückt besagen diese Ungleichungen, dass jede Zeile nur ein einziges Mal verwendet werden darf.

$$soa_i : \sum_{j=1}^{n} x_{i,j} = 1 \quad \forall i \in \{1, \ldots, n\}$$

Diese Ungleichungen sorgen dafür, dass ein Statement nur ein einzelnes Mal einer Zeile zugewiesen werden darf. Damit ist sichergestellt, dass die berechnete Lösung in jedem Fall eine Permutation der Methode darstellt.

Bestimmung der Position eines Statements

Aus der Belegung der Matrix P kann die Position des Statements s_i in der Permutation berechnet werden:

$$pos(s_i) = \sum_{j=1}^{n} j x_{i,j}$$

Die Positionsfunktion wird in der generierten Formulierung nicht gesondert betrachtet. Stattdessen wird die expandierte Summe an deren Stelle eingefügt.

Untrennbarkeit von Blöcken

Damit die Statements innerhalb eines Codeblocks nicht mit Statements außerhalb eines Codeblocks vertauscht werden können, müssen auch hierfür Gleichungen definiert werden. Informell kann dies dadurch erreicht werden, dass der Abstand zwischen zwei aufeinander folgenden Statements innerhalb eines Blocks auf eins festgelegt wird. Die entsprechenden Ungleichungen lassen sich wie folgt formulieren: Gehören die Statements $s_k, \ldots, s_l$ zu einem Block dann gilt für diese Statements:

$$blk_i : pos(s_i) - pos(s_{i+1}) = 1 \quad \forall i \in \{k, \ldots, l-1\}$$

Damit ist die Untrennbarkeit von Blöcken sichergestellt. Hier ist wie bereits beschrieben darauf zu achten, dass die Positionsfunktion durch die echten Summen ersetzt werden.

Zur Optimierung von Blöcken wird ein eigenes Problem formuliert. Die Kosten für eine Methode setzt sich als Summe über die Kosten der Blöcke zusammen.

Formulierung der Abhängigkeiten

Die Abhängigkeiten zwischen zwei Statements können durch eine einfache Ungleichung formuliert werden. Muss Statement s_i nach Statement s_j ausgeführt werden, dann kann dies durch die Ungleichung

$$dep_i : pos(s_j) - pos(s_i) \leq -1$$

in der berechneten Lösung berücksichtigt werden.

Berechnung des Ausgangswertes

Der eingesetzte Solver kann auch zur Berechnung des Ausgangswertes verwendet werden. Hierzu müssen lediglich die Positionen der einzelnen Statements bereits vor dem Lösungsvorgang auf den entsprechenden Wert fixiert werden. Dies geschieht zur Festlegen der Werte der Matrix P:

$$x_{i,j} = \begin{cases} 1 & i = j \\ 0 & i \neq j \end{cases}$$

Informell steht also in der berechneten Lösung das i-te Statement an der i-ten Stelle.

Formulierung der Kostenfunktion

Die zu minimierende Funktion muss ebenfalls für den ILP-Solver definiert werden. Die Definition unterscheidet sich kaum von der bereits vorgestellten Definition. Ausgangspunkt sind auch hierbei die Kanten E des Abhängigkeitsgraphen, deren Länge im resultierenden Kontrollflussgraphen minimiert werden soll. Die Kostenfunktion kann also definiert werden als:

$$cost : \sum_{(s_i, s_j) \in E} pos(s_j) - pos(s_i)$$

Auch hier ist wieder darauf zu achten, dass in einer ILP-Formulierung keine Funktionen auftreten dürfen und stattdessen die Positionsfunktion expandiert werden muss. Die zu minimierende Funktion stellt zusammen mit den beschriebenen Ungleichungen ein komplettes ILP dar, welches in jedem Fall mindestens eine Lösung, die Ausgangsreihenfolge, hat. Als Format für die Beschreibung und als Eingabe für die Löser wurde, aufgrund der im Vergleich zu MPS wesentlich besseren Lesbarkeit, das LP-Format gewählt. Eine beispielhafte Beschreibung einer Probleminstanz findet sich im Anhang B.

4.3.5 Getestete Solver

Die Formulierungen wurden programmatisch aus den berechneten Graphen erzeugt. Mit diesen Formulierungen wurde die Auswertung der ausgewählten Solver durchgeführt.

LPSolve 5.5

Zunächst wurde der frei verfügbare Solver LPSolve[1] evaluiert. Dieser zeichnet sich durch eine einfache Benutzung durch die Mitlieferung einer integrierten Entwicklungsumgebung aus. In den Tests war LPSolve leistungsfähiger als die bereits beschriebenen Strategien, allerdings wurde auch hier das Zeitlimit von 5 Minuten häufig überschritten. Dies ist darauf zurückzuführen, dass LPSolve nach wenigen Vorberechnungen eine Strategie anwendet, deren Performance mit der des ersten beschriebenen Algorithmus vergleichbar ist.

Symphony 5.2.0

Der zweite freie Solver, der getestet wurde, war Symphony[2]. Die Ergebnisse, die mit Symphony erzielt werden konnten, ähnelten jenen von LPSolve. Auch hier konnte eine bessere Performance verglichen mit den vorgestellten Algorithmen festgestellt werden. Die Ergebnisse waren jedoch ebenfalls nicht befriedigend, da die Marke von 5 Minuten oft überschritten wurde und die berechnete obere Schranke dann meist noch über dem Ausgangswert der Methode lag.

cplex 8.000

Der dritte Solver der untersucht wurde, ist cplex[3]. cplex ist ein kommerzieller Solver, der auf den Rechnern der Fakultät für Informatik zur Verfügung steht. Erste Tests lieferten sehr vielversprechende Ergebnisse, allerdings konnten auch hier für einige mittelgroße Instanzen nur nach weit mehr als 5 Minuten eine Lösung gefunden werden. Teilweise konnte auch überhaupt keine Lösung gefunden werden. Die mit cplex erzielten Ergebnisse waren bereits verwendbar, allerdings stammt cplex 8.000 aus dem Jahr 2002. Seitdem wurden in diesem Bereich noch große Fortschritte gemacht, weshalb ein weiterer, neuerer Solver getestet wurde.

SCIP 12.0.0

Der letzte getestete Solver SCIP[4] (**S**olving **C**onstraint **I**nteger **P**rograms) wird vom Konrad-Zuse-Zentrum für Informationstechnik Berlin entwickelt und ist für akademische Anwender kostenlos verfügbar. Benchmarks[5] zeigen, dass es sich bei SCIP um den mit Abstand leistungsfähigsten nicht-kommerzielle ILP-Solver handelt. Da mit SCIP

[1] http://lpsolve.sourceforge.net/5.5/
[2] http://branchandcut.org/
[3] http://www-01.ibm.com/software/integration/optimization/cplex-optimizer/
[4] http://scip.zib.de/
[5] http://plato.asu.edu/ftp/milpf.html

die meisten Probleme in akzeptabler Zeit gelöst werden konnten wurde dieser Solver ausgewählt, um die im nächsten Teil der Arbeit vorgestellten Analysen durchzuführen. Allerdings ist auch hier zu beachten, dass nicht alle Probleme gelöst werden konnten. Dies ist dann in den Lösungen vermerkt. Eine nicht lösbare Instanz lässt im Allgemeinen weniger auf die Größe der Instanz als viel mehr auf die geringen Zusammenhänge innerhalb einer Methode schließen.

4.3.6 Integration des Solvers

Da es sich bei der Implementierung lediglich um einen Prototypen handelt, wurde auf die Generierung eines Interfaces für den gewählten Solver verzichtet. Stattdessen wird aus ConQAT heraus eine LP-Datei generiert, welche als Eingabe für den Solver dient. Dieser wird anschließend über das Command Line Interface aufgerufen. Die dabei berechneten Ergebnisse werden in einer weiteren Datei gespeichert, welche, nachdem der Solver terminiert hat, von ConQAT zur weiteren Verarbeitung eingelesen und verarbeitet werden. Die Lösungsdatei enthält alle interessanten Informationen. Neben unterer und oberer Schranke enthält sie außerdem Informationen über die Laufzeit, die Zahl der Variablen und die entsprechende Belegung der Variablen. Falls ein Zeitlimit erreicht wurde, ist dies ebenfalls erkennbar und die aktuell beste obere Schranke kann verwendet werden.

Kapitel 5

Auswertung und Ergebnisse

In diesem Kapitel werden die bei der Berechnung entstandenen Ergebnisse auf drei exemplarisch ausgewählten Projekten präsentiert. Zur Berechnung der Ergebnisse wird die im vorherigen Kapitel beschriebene Integration eines ILP-Solvers verwendet.

5.1 Ziele und untersuchte Fragestellungen

Ziel der Auswertungen war es herauszufinden, wie gut das Modell auf echten Code angewandt funktioniert. Dabei waren folgende Fragestellungen von Interesse:

- Wie groß ist der Anteil an Methoden, der sich durch dieses Modell optimieren lässt?
- Sind die berechneten Optimierungen gültig?
- Wie gut sind die berechneten Optimierungen im Bezug auf Lesbarkeit?
- Wie aussagekräftig sind Absolutwerte?
- Wie aussagekräftig sind Relativwerte?
- Korrelieren die gemessenen Ergebnisse mit bisherigen Erfahrungen?

Des Weiteren galt es zu untersuchen, wo weitere Optimierungen im Modell notwendig sind und welche weiteren Eigenschaften von Code ausgehend von dem vorgestellten Modell untersucht werden können.

5.2 Allgemeine Beobachtungen

Allgemein ist zu beobachten, dass die berechneten Kosten sehr stark schwanken. Insbesondere bei Methoden, deren Aufgabe die Implementierung einer graphischen Benutzeroberfläche ist, sind die Kosten sehr hoch, da die Methoden im Allgemeinen sehr lang und wenig verschachtelt sind.

Ein relativ großer Teil der Methoden erreicht Kosten der Höhe 0, was darauf zurückzuführen ist, dass es sich bei diesen Methoden um einfache Getter oder Setter handelt, die aus lediglich einer Zeile Code bestehen und daher keine Überlappungen zulassen.

Des Weiteren ist auffällig, dass viele Methoden bereits über eine fast optimale Verteilung der Statements verfügen, obwohl Entwickler nicht explizit auf eine solche Verteilung achten. Auch hier fällt auf, dass, je komplexer die Methode ist, desto näher die erreichte Punktzahl am Optimum ist. Methoden, die etwa zum Laden von Testdaten oder zum Aufbau einer graphischen Oberfläche dienen, weichen auch hier stärker von den berechneten Optima ab. Dies kann möglicherweise damit begründet werden, dass es sich bei einer Benutzeroberfläche nicht um einen einzelnen Themenbereich handelt und beispielsweise verschiedene Buttons oder Textfelder im Allgemeinen nicht direkt miteinander interagieren.

5.3 Detaillierte Auswertung

Zur Auswertung wurde ein Dashboard erzeugt, welches die Analyseergebnisse übersichtlich zusammenfasst. Des Weiteren wurde für exemplarische Klassen und Methoden die Entstehung der Punktzahl anhand des Codes nachvollzogen. Einzelne Beispiele sollen die errechneten Verbesserungen verdeutlichen. Teilweise entsteht durch eine Optimierung auch subjektiv schlechter verständlicherer Code. Hierfür werden ebenfalls Beispiele angeführt. Die statistische Auswertung wurde mit Hilfe eines Exports der Daten in Excel durchgeführt.

5.3.1 Auswahl der Beispielanalysen

Als exemplarische Analysen wurden JUnit, JabRef sowie der Kern von ConQAT ausgewählt. JUnit ist dabei Stellvertreter für OpenSource-Projekte mit sehr großer Verbreitung und dezentraler Entwicklung. Bei JabRef handelt es sich um ein Werkzeug, welches zur Verwaltung von Literaturdatenbanken entwickelt wurde. Bei ConQAT handelt es sich um das in dieser Arbeit verwendete Analyseframework selbst, bei dessen Entwicklung großer Wert auf Sicherstellung einer hohen Codequalität liegt. Interessant ist hierbei auch, wie sich die Analyseergebnisse im Vergleich zu den anderen hier vorgestellten und analysierten Projekten verhalten.

5.3.2 JabRef

JabRef[1] stellt das erste analysierte Projekt dar. Zur Analyse wurde die Version 2.6 der Software verwendet. Bei JabRef konnten insgesamt 3391 Methoden analysiert werden. Von diesen 3391 Methoden konnten 84 aufgrund von Zeitbeschränkungen nicht komplett analysiert werden. Für 1415 Methoden lagen die errechneten Kosten bei null, für 1471 Methoden wurde eine bereits optimale Reihenfolge der Statements festgestellt. Die statistischen Ergebnisse sind in folgender Tabelle dargestellt.

Messgröße	**JabRef**	**JabRef Tests**
Anzahl Methoden	3205	292
Methoden m. Score 0	$1440(44,93\%)$	$47(16,10\%)$
Methoden m. Verhältnis 1	$1279(39,91\%)$	$220(75,34\%)$
Methoden m. Verhältnis > 1	$486(15,16\%)$	$25(8,56\%)$
Maximalwert	589 (real), 577 (optimal)	320 (real), 311 (optimal)
Maximaler Verbesserungsfaktor	$5,21$	$1,72$

Allgemeine Beobachtungen

Bei der Analyse der Methoden konnten eine Reihe an interessanten Beobachtungen gemacht werden. In bestimmten Arten von Code wurden besonders hohe Werte erreicht. So wurden bei einer absteigenden Sortierung der Ergebnisse nach dem Verhältnis $score_{real}/score_{opt}$ in den obersten Zeilen überproportional viele Methoden gefunden, welche Code zur Konstruktion von Benutzerinterfaces beinhalten. Dies kann mit Einschränkungen damit erklärt werden, dass diese Art von Code sehr viele Seiteneffekte erzeugt, welche Abhängigkeiten erzeugen, die bei der implementierten Analyse nicht gefunden werden. Außerdem handelt es sich bei den Methoden zum Großteil um sehr lange Methoden.

Des Weiteren fällt auf, dass im mit ANTLR generierten Code des BibTEX-Lexers und des BibTEX-Parsers ebenfalls hohe Werte erreicht werden. Dies spricht dafür, dass die Vorgehensweise, die zu verhältnismäßig niedrigen Werten führt, in Codegeneratoren nicht implementiert ist.

Detaillierte Auswertung einzelner Methoden

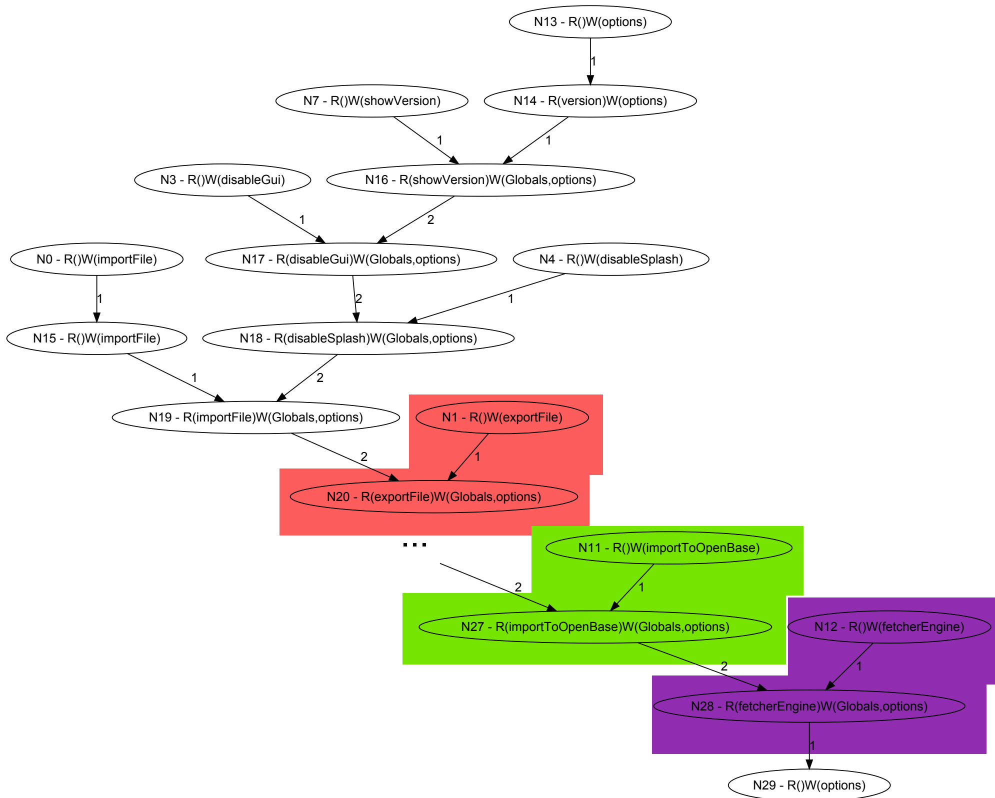
Es konnten einige Methoden aus dem JabRef-Code nachvollziehbar verbessert werden. Folgendes Beispiel erreichte in den Testläufen die maximale relative Verbesserung um den Faktor $5,21$:

```
importFile = new StringOption("");
```

[1] http://jabref.sourceforge.net/

```
exportFile = new StringOption("");
helpO = new BooleanOption();
disableGui = new BooleanOption();
disableSplash = new BooleanOption();
blank = new BooleanOption();
loadSess = new BooleanOption();
showVersion = new BooleanOption();
exportPrefs = new StringOption("jabref_prefs.xml");
importPrefs = new StringOption("jabref_prefs.xml");
auxImExport = new StringOption("");
importToOpenBase = new StringOption("");
fetcherEngine = new StringOption("");

options = new Options("JabRef "); // Create an options repository.
options.setVersion(GUIGlobals.version);

importFile.setDescription("imopoepuoeu"); // Globals.lang);
options.register("version", 'v', Globals.lang("Display version"),
  showVersion);
options
  .register("nogui", 'n',
  Globals.lang("No GUI. Only process command line options."),
  disableGui);

options.register("nosplash", 's',
  Globals.lang("Do not show splash window at startup"), disableSplash);
options.register("import", 'i', Globals.lang("Import file") + ": "
  + Globals.lang("filename") + "[,import format]", importFile);
options.register("output", 'o', Globals.lang("Output or export file")
  + ": " + Globals.lang("filename") + "[,export format]",
  exportFile);
options.register("help", 'h',
  Globals.lang("Display help on command line options"), helpO);
options.register("loads", 'l', Globals.lang("Load session"), loadSess);
options.register("prexp", 'x',
  Globals.lang("Export preferences to file"), exportPrefs);
options.register("primp", 'p',
  Globals.lang("Import preferences from file"), importPrefs);
options.register("aux", 'a', Globals.lang("Subdatabase from aux")
  + ": " + Globals.lang("file") + "[.aux]" + ","
  + Globals.lang("new") + "[.bib]", auxImExport);
options.register("blank", 'b',
  Globals.lang("Do not open any files at startup"), blank);

options.register("importToOpen", '\0',
  Globals.lang("Import to open tab"), importToOpenBase);

options.register("fetch", 'f',
  Globals.lang("Run Fetcher, e.g. \"--fetch=Medline:cancer\""),
  fetcherEngine);

options.setUseMenu(false);
```

Der Abhängigkeitsgraph der Methode sieht wie folgt aus:



Aus dem Abhängigkeitsgraphen wird die Struktur der Methode sehr gut deutlich. Es wird für jede zu setzende Option zunächst ein Optionsobjekt erzeugt, welches den Wert repräsentiert. Anschließend wird dieses Objekt in einer globalen Optionsregistrierung registriert. Das Problem an dieser Methode ist, dass zunächst die Optionsobjekte erzeugt werden und anschließend registriert werden. Um nachzuvollziehen, welchen Standardwert die registrierte Option hat, muss von der Registrierung an die Deklaration gesprungen werden.

Die berechnete Optimierung sieht vor, die Deklarations-Registrierungsblöcke, welche im Graphen blockweise farbig markiert sind, zusammenzufassen. Dabei werden die Deklarationen der Werteobjekte und die Registrierung der Objekte direkt hintereinander geschrieben.

Eine andere Möglichkeit, die Lesbarkeit zu verbessern, wäre etwa, ein Refactoring anzuwenden, welches die Initialisierung der Werteobjekte sowie die Registrierung der Werteobjekte trennt. Ein solches Refactoring wäre etwa die Extraktion zweier Methoden.

Durch die Umstrukturierung wird die Methode selbsterklärender, da erleichternd auch noch bei jeder Registrierung eine textuelle Beschreibung der Methode übergeben wird:

```
options = new Options("JabRef "); // Create an options repository.
```

```
options.setVersion(GUIGlobals.version);

showVersion = new BooleanOption();
options.register("version", 'v', Globals.lang("Display version"),
  showVersion);

disableGui = new BooleanOption();
options.register("nogui", 'n',
    Globals.lang("No GUI. Only process command line options."),
    disableGui);

disableSplash = new BooleanOption();
options.register("nosplash", 's',
  Globals.lang("Do not show splash window at startup"), disableSplash);

importFile = new StringOption("");
importFile.setDescription("imopoepuoeu"); // Globals.lang);
options.register("import", 'i', Globals.lang("Import file") + ": "
  + Globals.lang("filename") + "[,import format]", importFile);

exportFile = new StringOption("");
options.register("output", 'o', Globals.lang("Output or export file")
  + ": " + Globals.lang("filename") + "[,export format]", exportFile);

helpO = new BooleanOption();
options.register("help", 'h',
  Globals.lang("Display help on command line options"), helpO);

loadSess = new BooleanOption();
options.register("loads", 'l', Globals.lang("Load session"), loadSess);

exportPrefs = new StringOption("jabref_prefs.xml");
options.register("prexp", 'x',
  Globals.lang("Export preferences to file"), exportPrefs);

importPrefs = new StringOption("jabref_prefs.xml");
options.register("primp", 'p',
  Globals.lang("Import preferences from file"), importPrefs);

auxImExport = new StringOption("");
options.register("aux", 'a', Globals.lang("Subdatabase from aux")
  + ": " + Globals.lang("file") + "[.aux]" + ","
  + Globals.lang("new") + "[.bib]", auxImExport);

blank = new BooleanOption();
options.register("blank", 'b',
  Globals.lang("Do not open any files at startup"), blank);

importToOpenBase = new StringOption("");
options.register("importToOpen", '\0',
  Globals.lang("Import to open tab"), importToOpenBase);

fetcherEngine = new StringOption("");
```

```
options.register("fetch", 'f',
  Globals.lang("Run Fetcher, e.g. \"--fetch=Medline:cancer\""),
  fetcherEngine);

options.setUseMenu(false);
```

Ähnliche Verbesserungen konnten bei verschiedenen Methoden im JabRef-Code festgestellt werden. Allerdings sind die berechneten Verbesserungen nicht immer direkt umsetzbar, da Abhängigkeiten durch Methodenaufrufe oder Seiteneffekte nicht beachtet werden. Prinzipiell ist das Ergebnis jedoch, dass in Methoden mit einem hohen relativen Verbesserungsfaktor und einem absoluten Wert größer 10 meist eine Restrukturierung zu einer tatsächlichen Verbesserung der Lesbarkeit führt.

5.3.3 ConQAT Core

Der Kern des Analyseframeworks ConQAT[2] wurde als zweites Projekt ausgewählt und mit der beschriebenen Methode untersucht. Analysiert wurde die Version 2.6. Es wurden dabei 728 Methoden des Kerns selbst sowie 240 Methoden von Testfällen untersucht. Es wird erwartet, dass die Ergebnisse für ConQAT auf Grund des Fokus auf qualitativ hochwertigem Code und einem Entwicklungsprozess mit integrierten Qualitätssicherungsmaßnahmen in Form von Reviews gut ausfallen.

Messgröße	**ConQAT**	**ConQAT Tests**
Anzahl Methoden	728	239
Methoden m. Score 0	$344(47,25\%)$	$154(64,44\%)$
Methoden m. Verhältnis 1	$340(46,70\%)$	$83(34,73\%)$
Methoden m. Verhältnis > 1	$44(6,04\%)$	$2(0,84\%)$
Maximalwert	81 (real), 81 (optimal)	84 (real), 76 (optimal)
Maximaler Verbesserungsfaktor	$2,00$	$1,25$

Allgemeine Beobachtungen

Die gesamte Codebase von ConQAT konnte, anders als bei JabRef, in etwa 500 Sekunden analysiert werden. Lediglich eine einzige Methode erreichte den festgelegten Timeout von 300 Sekunden. 47% der Methoden von ConQAT erreichten einen Wert von 0, bei den Tests erreichten sogar knapp 65% der Methoden diesen Wert. Dies ist auf sehr simple Methoden zurückzuführen, bei denen keinerlei Beziehungen zwischen den Statements besteht, oder die lediglich nur aus einem einzigen Statement bestehen. Dies ist beispielsweise bei einfachen Getter- und Setter-Methoden der Fall. Im Core wurde für 46 % ein Wert von 1 berechnet, was einer optimalen Reihenfolge der Statements entspricht. Bei den Tests erreichten knapp 35% diesen Wert. Eine Verbesserung konnte im Core-Code bei

[2] http://conqat.cs.tum.edu/

etwa 6% errechnet werden, was 44 Methoden entspricht. Es konnten bei den Testfällen lediglich bei zwei Methoden Verbesserungen festgestellt werden. Insgesamt wurde beobachtet, dass kürzere Methoden, die per se eine höhere Kohäsion haben, in den Analysen besser abschneiden als Methoden, in denen verschiedene Aufgaben implementiert sind, etwa umfassende Testfälle.

Zusammenfassend kann festgestellt werden, dass dadurch, dass die durchschnittliche Methodenlänge im ConQAT-Code relativ kurz ist, wesentlich bessere Ergebnisse erzielt werden. Wurden weniger gute Ergebnisse erzielt, so ist dies bei einer näheren Analyse darauf zurückzuführen, dass durch Refactorings Methoden extrahiert wurden und die Seiteneffekte und Zugriffe auf Objektvariablen innerhalb dieser Methoden bei der Analyse nicht berücksichtigt werden.

Die Maximalwerte lagen für den Core bei 81 (real) und 81 (optimal), für die Testfälle bei 84 (real) und 76 (optimal). Das größte Verbesserungsverhältnis lag für den Core bei 2, für die Testfälle bei 1,25.

Detaillierte Auswertung einzelner Methoden

Die Arten der Verbesserungen ähneln denen, die bei JabRef gefunden wurde. Es sei hier ein Beispiel für eine Methode angeführt, die deutlich optimiert wurde. Die Optimierung ist jedoch nicht gültig, da hinter den Methodenaufrufen Abhängigkeiten verborgen sind, die nicht im Graphen repräsentiert sind.

Die optimierte Methode:

```
execute(...) {
  File jarFile = new File(getProject().getBaseDir()
    .getName() + ".jar");
  fileCount = 0;
  try {
    addBundles(bundleClosure);
    // add additional libraries and directories
    for (AntElement library : libraries) {
      addJar(new File(library.getName()));
    }
    for (AntElement directory : directories) {
      addDirectory(new File(directory.getName()));
    }

    outStream = new JarOutputStream(new FileOutputStream(jarFile));
    System.out.println("Created " + jarFile.getName() + " from "
      + fileCount + " files.");
    outStream.close();

  } catch (IOException e) {
    throw new BuildException(e);
  }
  jarEntries.clear();
```

```
}
```

Die originale Methode:

```
execute(...) {
  File jarFile = new File(getProject().getBaseDir()
    .getName() + ".jar");
  jarEntries.clear();
  fileCount = 0;
  try {
    outStream = new JarOutputStream(new FileOutputStream(jarFile));
    addBundles(bundleClosure);
    // add additional libraries and directories
    for (AntElement library : libraries) {
      addJar(new File(library.getName()));
    }
    for (AntElement directory : directories) {
      addDirectory(new File(directory.getName()));
    }
    outStream.close();

    System.out.println("Created " + jarFile.getName() + " from "
      + fileCount + " files.");
  } catch (IOException e) {
    throw new BuildException(e);
  }
}
```

Dieses Beispiel verdeutlicht einerseits die Arbeitsweise der Optimierung, andererseits werden hier auch Grenzen der Optimierung deutlich. So wird beispielsweise die Verwendung der Variablen `outStream` so lokal wie möglich gehalten. Allerdings wird diese Variable indirekt in der Methode `addBundles` sowie in der Methode `addDirectory` verwendet. Die hier vorgeschlagene Transformation hätte zur Folge, dass die Variable `outStream` im Methodenaufruf noch nicht initialisiert ist. Ebenso ist die Verschiebung der Verwendung der Variablen `jarEntries` an das Ende der Methode im Sinne der Optimierung zwar korrekt, auf Grund von Seiteneffekten wird hier jedoch auch die Semantik der Methode verändert. Die Methode wird dadurch fehlerhaft.

Bei der Betrachtung der einzelnen Methoden fällt auf, dass die durchschnittliche Methodenlänge der verbesserten Methoden im Vergleich zu JabRef deutlich niedriger ist. Dies ist Voraussetzung dafür, dass niedrige Werte in der Analyse erreicht werden können.

5.3.4 JUnit

Bei der Analyse von JUnit[3] wurden 692 Methoden sowie 689 Methoden von Testfällen analysiert und ausgewertet. In den Testfällen konnte für 13,94% der Methoden eine

[3] http://junit.org

Optimierung errechnet werden, im restlichen Code lediglich bei 1,88%. Es wurde ein Snapshot der Version 4.9 vom 11. Mai 2010 analysiert.

Messgröße	**JUnit**	**JUnit Tests**
Anzahl Methoden	692	689
Methoden m. Score 0	420(60,96%)	223(32,37%)
Methoden m. Verhältnis 1	259(37,43%)	372(53,99%)
Methoden m. Verhältnis > 1	13(1,88%)	94(13,94%)
Maximalwert	42 (real), 42 (optimal)	105 (real), 58 (optimal)
Maximaler Verbesserungsfaktor	1,70	2,24

Allgemeine Beobachtungen

Ähnlich wie beim ConQAT-Code konnte auch bei JUnit nur in wenigen Klassen überhaupt eine Optimierung berechnet werden. Wurde eine Optimierung berechnet, so war diese meist ungültig, da sie die Semantik der Methode veränderte. Der Hauptgrund für die Ungültigkeit waren Seiteneffekte, die nicht gefundene Abhängigkeiten induzieren.

Detaillierte Auswertung einzelner Methoden

Für folgende Methode wurde eine deutliche Optimierung berechnet, die allerdings aufgrund von Seiteneffekten ungültig ist. Ähnliches wurde bei praktisch allen in JUnit analysierten Methoden festgestellt.

```
StringWriter swin= new StringWriter();
PrintWriter pwin= new PrintWriter(swin);
pwin.println("junit.framework.AssertionFailedError");
pwin.println("␣␣at␣junit.framework.Assert.fail(Assert.java:144)");
pwin.println("␣␣at␣junit.framework.Assert.assert(Assert.java:19)");
pwin.println("␣␣at␣junit.framework.Assert.assert(Assert.java:26)");
pwin.println("␣␣at␣MyTest.f(MyTest.java:13)");
pwin.println("␣␣at␣MyTest.testStackTrace(MyTest.java:8)");
pwin.println("␣␣at␣java.lang.reflect.Method.invoke(Native␣Method)");
pwin.println("␣␣at␣junit.framework.TestCase.runTest(TestCase.java:156)");
pwin.println("␣␣at␣junit.framework.TestCase.runBare(TestCase.java:130)");
pwin.println("␣␣at␣junit.framework.TestResult\$1.protect(TestResult.java
    :100)");
pwin.println("␣␣at␣junit.framework.TestResult.runProtected(TestResult.java
    :118)");
pwin.println("␣␣at␣junit.framework.TestResult.run(TestResult.java:103)");
pwin.println("␣␣at␣junit.framework.TestCase.run(TestCase.java:121)");
pwin.println("␣␣at␣junit.framework.TestSuite.runTest(TestSuite.java:157)");
pwin.println("␣␣at␣junit.framework.TestSuite.run(TestSuite.java,␣Compiled␣
    Code)");
pwin.println("␣␣at␣junit.swingui.TestRunner\$17.run(TestRunner.java:669)");
fUnfiltered= swin.toString();
```

```
StringWriter swout= new StringWriter();
PrintWriter pwout= new PrintWriter(swout);
pwout.println("junit.framework.AssertionFailedError");
pwout.println(" at MyTest.f(MyTest.java:13)");
pwout.println(" at MyTest.testStackTrace(MyTest.java:8)");
fFiltered= swout.toString();
```

Die Optimierung schlägt vor, die beiden Methodenaufrufe `swin.toString()` sowie `swout.toString()` vor die Ausgabe zu ziehen. Werden die beiden Methodenaufrufe jedoch vor der Ausgabe durchgeführt, dann wird jeweils ein leerer String zurückgeliefert anstelle des gewünschten Stacktraces.

5.3.5 Ergebnisse

Die Fragestellungen können nach eingehender Analyse zum Großteil beantwortet werden. Generell ist zu sagen, dass der Anteil der optimierbaren Methoden relativ gering ist. Im Test schwankten die Werte zwischen $0,84\%$ (ConQAT Tests) und $15,16\%$ (JabRef). Eine intensive Wartung des Codes sowie eine entsprechende Reife der Software scheinen Gründe für gute Bewertungen zu sein.

Die berechneten Optimierungen sind lediglich teilweise gültig. Ein Grund hierfür ist, dass einige Abhängigkeiten durch Methoden verborgen werden oder aufgrund von Seiteneffekten nicht messbar sind. Besonders bei JUnit war dieses Phänomen zu beobachten. Hier gab es insgesamt sehr wenige mögliche Optimierungen. Die gefundenen Optimierungen waren meist ungültig. JabRef hingegen zeigt, dass die Methode durchaus zu Verbesserungen führen kann. Hier konnten eine Reihe an gültigen Verbesserungen gefunden und verwendet werden.

Am Beispiel von JabRef lässt sich auch erkennen, dass die berechneten Optimierungen Potential haben, die Lesbarkeit zu verbessern. Allerdings sind die berechneten Ergebnisse immer mit Vorsicht zu betrachten: Häufig muss händisch editiert werden, um eine tatsächliche Verbesserung zu erreichen.

Die berechneten Absolutwerte schwanken sehr stark, sodass eine Aussage auf dieser Basis kaum möglich ist. Allgemein kann gesagt werden, dass bei hohen Absolutwerten

- viele Variablen in der Methode verwendet werden.
- die Methode von mindestens mittlerer Länge ist.

Eine Einschätzung, wie groß die möglichen Verbesserungen sind, lässt sich hieraus jedoch nicht unmittelbar ableiten. Das Verhältnis von realem Wert und optimalem Wert ist hier aussagekräftiger.

Subjektive Erfahrungen bei den Analysen zeigen, dass der JUnit-Code sowie der ConQAT-Code insgesamt verständlicher sind als der JabRef-Code. Dies ist vor allem auf die kürzeren und intuitiver strukturierten Methoden zurückzuführen. Das Ergebnis spiegelt sich in den Ergebnissen wieder, wobei nur ein kleiner Teil des gesamten Codes betrachtet wurde und daher diese Aussage mit Vorsicht zu genießen ist.

Kapitel 6

Zusammenfassung

Diese Arbeit stellt einen Ansatz dar, wie aus Daten- und Kontrollflussgraphen proaktiv Strukturverbesserungen im Bezug auf Lesbarkeit und Verständnis von Code generiert werden können. Das Modell wurde an echten Projekten getestet und die Ergebnisse wurden bewertet. Abschließend kann festgestellt werden, dass der erarbeitete Ansatz bereits einige interessante Erkenntnisse liefert, jedoch auch Grenzen besitzt. Diese werden einerseits durch die mathematische Komplexität der Optimierungen aufgezeigt, andererseits durch schwer messbare Eigenschaften, etwa Seiteneffekte. Zur produktiven Verwendung eines solchen Ansatzes ist weitere Arbeit notwendig. Die Ergebnisse haben jedoch auch gezeigt, dass die Methoden, für die eine hohe Optimierung berechnet wurde, meist Verbesserungen benötigen. Diese könnten etwa in Form von Refactorings wie beispielsweise Extraktion von Methoden stattfinden.

Die generierten optimalen Reihenfolgen der Statements sind hingegen meist nicht eins zu eins umsetzbar. Durch die Komplexität von realem Code, sei es aufgrund von Seiteneffekten oder etwa auch Reflection, wird durch die berechneten Optimierungen meist die Semantik der Methoden verändert. Eine hohe Optimierung korreliert jedoch damit, wie viele unterschiedliche Aufgaben innerhalb einer Methode verarbeitet werden. Sind beispielsweise in einer Methode verschiedene Verfahren zum Vergleich zweier Objekte implementiert, dann äußert sich dies einerseits in hohen Laufzeiten der Optimierungsstrategie und andererseits in hohen Ausgangswerten und hohen relativen Optimierungsfaktoren. Obgleich die berechnete Optimierung hier nicht unmittelbar eine Verbesserung bietet, kann durch eine Aufteilung der Verfahren auf verschiedene Methoden eine strukturelle Verbesserung erreicht werden.

Die Verständlichkeit und Lesbarkeit von Code kann mit dieser Methode nicht in allen Fällen adäquat bewertet werden. So kann die Implementierung komplexer Algorithmen bei entsprechender Strukturierung auch absolut sehr gute Punktzahlen erreichen, selbst wenn der Code nur schwer verständlich ist. Die Quote der False Negatives ist relativ hoch. Bei schlechten relativen Bewertungen liegt jedoch meist wirklich ein strukturelles

Problem vor.

Kapitel 7

Ausblick

Die in dieser Arbeit gewonnenen Erkenntnisse können für verschiedene Zwecke weiterverwendet werden. Die berechneten Abhängigkeitsgraphen etwa stellen eine Möglichkeit dar, Datenabhängigkeiten zwischen verschiedenen Programmteilen zu identifizieren. So wären etwa Algorithmen denkbar, die in diesen Graphen Zusammenhangskomponenten suchen und diese ab einer gewissen Größe dem Benutzer als Vorschlag für ein Method Extraction Refactoring zur Verfügung stellen.

Durch eine weitere Analyse der in dieser Arbeit erzeugten Graphen wäre auch eine gezielte Suche nach verdächtigen Strukturen denkbar, um etwa Qualitätsmängel festzustellen. Ein einfaches Beispiel wäre auch hier die Anzahl der Zusammenhangskomponenten, in die ein Graph zerfällt, wenn einzelne Knoten weggelassen werden. Entstehen dadurch viele Zusammenhangskomponenten, so spricht dies unter Umständen für eine niedrige Kohäsion innerhalb der Methode und kann als Hinweis für die Notwendigkeit für ein Refactoring verwendet werden.

Die Verwendung der Methode zur Lesbarkeitsanalyse kann ebenfalls noch weiter verbessert werden. Die hier vorgestellte Methode kann dabei als Basis dienen. So ist etwa durch die Berechnung der Mengen der Variablen, welche innerhalb aufgerufener Methoden gelesen oder verändert werden, eine Verbesserung hinsichtlich der Gültigkeit der berechneten Reihenfolge möglich. Die bisher erzielten Ergebnisse können zur Erzeugung von Stichproben in einem übergeordneten Qualitätsmanagementprozess dienen, da sich in der Auswertung gezeigt hat, dass schlechte Bewertungen von Methoden meist auf Strukturprobleme hinweisen.

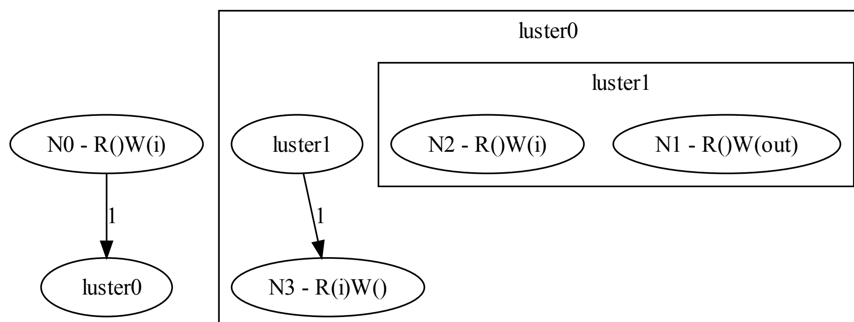
Ebenfalls von Interesse ist die Analyse von neuem Code, der noch nicht den Reifegrad der in dieser Arbeit analysierten Projekte hat. Aus einer solchen Analyse ließen sich bessere Rückschlüsse auf die Leistungsfähigkeit der Methode zur proaktiven Untersützung von Entwicklern ziehen.

Anhang A

Kontrollflussanweisungen im Graphen

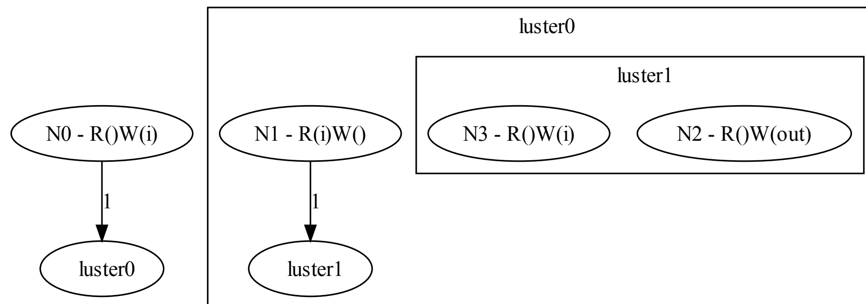
Im Folgenden ist die Abbildung von Kontrollflussstrukturen im Code auf die Abhängigkeitsgraphen dokumentiert. Die Cluster entsprechen logischen Blöcken, welche in der Lösungsstrategie expandiert werden.

A.1 do...while-Schleife



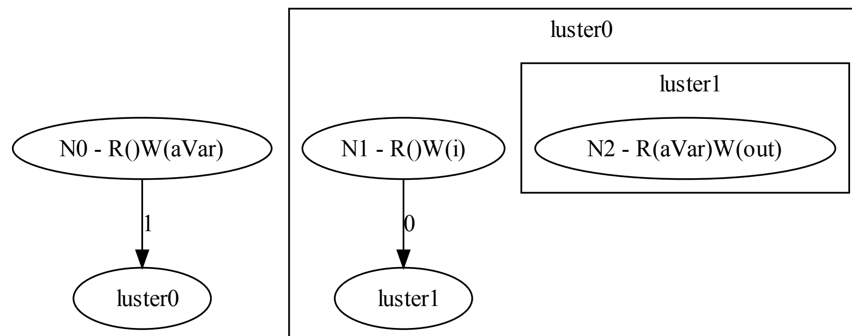

```
int i = 0;
do {
  System.out.println("test");
  i++;
} while (i < 10);
```

A.2 while-Schleife



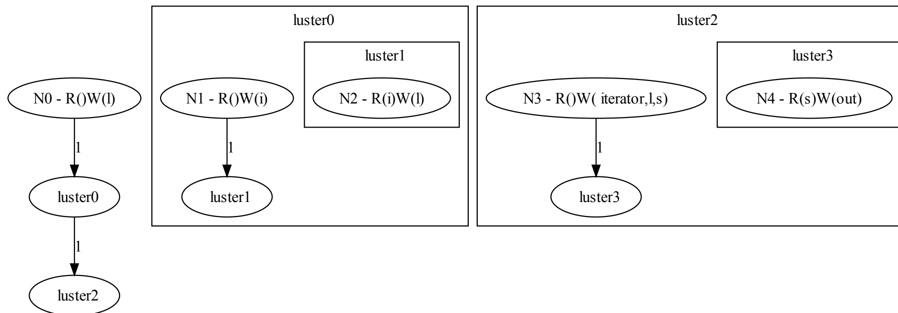

```
int i = 0;
while (i < 10) {
  System.out.println("test");
  i++;
}
```

A.3 for-Schleife



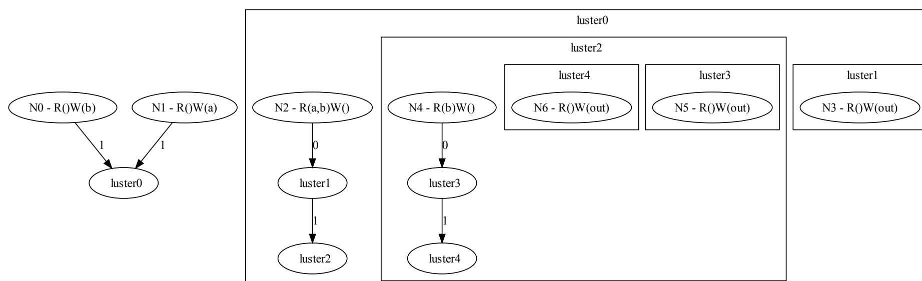

```
int aVar = 5;
for (int i = 0; i < 13; i++) {
  System.out.println(aVar);
}
```

A.4 for-each-Schleife



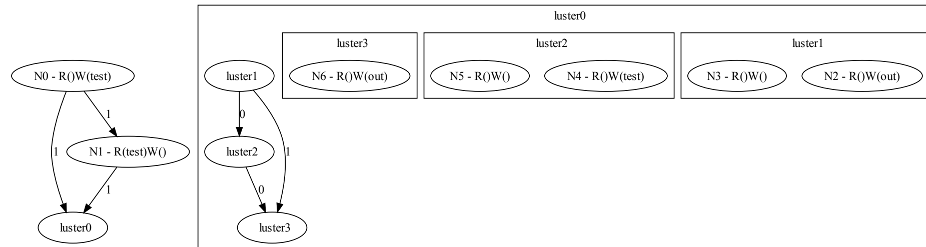

```
List<String> l = new ArrayList<String>();
for (int i = 0; i < 10; i++) {
  l.add("item␣" + i);
}
for (String s : l) {
  System.out.println(s);
}
```

A.5 if-then-else-Verzweigung



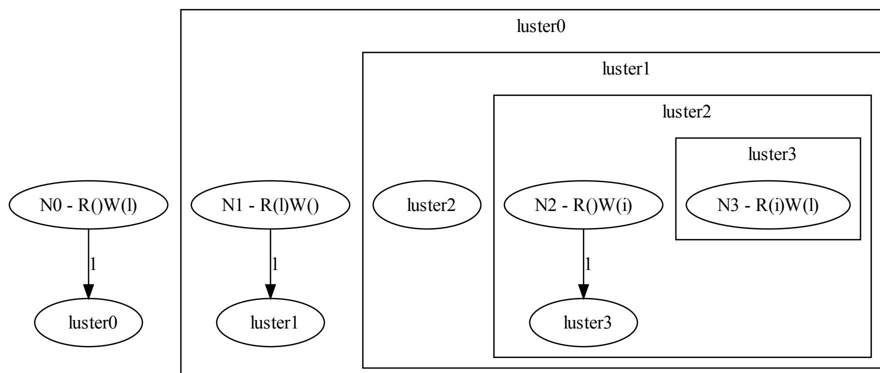

```
boolean b = false;
boolean a = true;
if (b && a) {
  System.out.println("test1");
} else if (b) {
  System.out.println("test2");
} else {
  System.out.println("test3");
}
```

A.6 switch-Verzweigung




```
int test = 3;
switch (test) {
case 1:
  System.out.println("test");
  break;
case 2:
  test += 1;
  break;
default:
  System.out.println("test2");
}
```

A.7 synchronized-Block




```
List<String> l = new ArrayList<String>();
synchronized (l) {
```

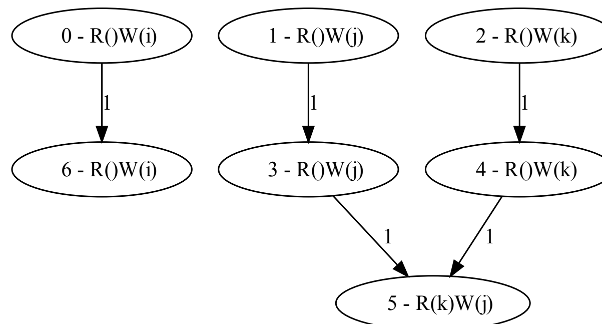
```
  for (int i = 0; i < 10; i++) {
    l.add("item " + i);
  }
}
```

Anhang B

Beispiel für eine LP-Beschreibung

Hier wird ein Beispiel für die Beschreibung eines Optimierungsproblems im LP-Format für eine einfache Methode gezeigt. Der berechnete Wert für diese Methode beträgt 13 (original) bzw. 7 (optimiert).

```
int i = 0;
int j = 0;
int k = 0;
j++;
k++;
j = j + k;
i++;
```



Eine LP-Beschreibung des Problems sieht wie folgt aus:

```
min
  Zielfunktion:
  1 s6l1 + 2 s6l2 + 3 s6l3 + 4 s6l4 + 5 s6l5 + 6 s6l6
    - 1 s0l1 - 2 s0l2 - 3 s0l3 - 4 s0l4 - 5 s0l5 - 6 s0l6
  + 1 s3l1 + 2 s3l2 + 3 s3l3 + 4 s3l4 + 5 s3l5 + 6 s3l6
    - 1 s1l1 - 2 s1l2 - 3 s1l3 - 4 s1l4 - 5 s1l5 - 6 s1l6
  + 1 s4l1 + 2 s4l2 + 3 s4l3 + 4 s4l4 + 5 s4l5 + 6 s4l6
    - 1 s2l1 - 2 s2l2 - 3 s2l3 - 4 s2l4 - 5 s2l5 - 6 s2l6
  + 1 s5l1 + 2 s5l2 + 3 s5l3 + 4 s5l4 + 5 s5l5 + 6 s5l6
    - 1 s3l1 - 2 s3l2 - 3 s3l3 - 4 s3l4 - 5 s3l5 - 6 s3l6
```

```
  + 1 s5l1 + 2 s5l2 + 3 s5l3 + 4 s5l4 + 5 s5l5 + 6 s5l6
    - 1 s4l1 - 2 s4l2 - 3 s4l3 - 4 s4l4 - 5 s4l5 - 6 s4l6

subject to
  dep0: 1 s6l1 + 2 s6l2 + 3 s6l3 + 4 s6l4 + 5 s6l5 + 6 s6l6
        - 1 s0l1 - 2 s0l2 - 3 s0l3 - 4 s0l4 - 5 s0l5 - 6 s0l6 >= 1
  dep1: 1 s3l1 + 2 s3l2 + 3 s3l3 + 4 s3l4 + 5 s3l5 + 6 s3l6
        - 1 s1l1 - 2 s1l2 - 3 s1l3 - 4 s1l4 - 5 s1l5 - 6 s1l6 >= 1
  dep2: 1 s4l1 + 2 s4l2 + 3 s4l3 + 4 s4l4 + 5 s4l5 + 6 s4l6
        - 1 s2l1 - 2 s2l2 - 3 s2l3 - 4 s2l4 - 5 s2l5 - 6 s2l6 >= 1
  dep3: 1 s5l1 + 2 s5l2 + 3 s5l3 + 4 s5l4 + 5 s5l5 + 6 s5l6
        - 1 s3l1 - 2 s3l2 - 3 s3l3 - 4 s3l4 - 5 s3l5 - 6 s3l6 >= 1
  dep4: 1 s5l1 + 2 s5l2 + 3 s5l3 + 4 s5l4 + 5 s5l5 + 6 s5l6 - 1 s4l1
        - 2 s4l2 - 3 s4l3 - 4 s4l4 - 5 s4l5 - 6 s4l6 >= 1

  loa0: s0l0 + s1l0 + s2l0 + s3l0 + s4l0 + s5l0 + s6l0 = 1
  loa1: s0l1 + s1l1 + s2l1 + s3l1 + s4l1 + s5l1 + s6l1 = 1
  loa2: s0l2 + s1l2 + s2l2 + s3l2 + s4l2 + s5l2 + s6l2 = 1
  loa3: s0l3 + s1l3 + s2l3 + s3l3 + s4l3 + s5l3 + s6l3 = 1
  loa4: s0l4 + s1l4 + s2l4 + s3l4 + s4l4 + s5l4 + s6l4 = 1
  loa5: s0l5 + s1l5 + s2l5 + s3l5 + s4l5 + s5l5 + s6l5 = 1
  loa6: s0l6 + s1l6 + s2l6 + s3l6 + s4l6 + s5l6 + s6l6 = 1
  soa0: s0l0 + s0l1 + s0l2 + s0l3 + s0l4 + s0l5 + s0l6 = 1
  soa1: s1l0 + s1l1 + s1l2 + s1l3 + s1l4 + s1l5 + s1l6 = 1
  soa2: s2l0 + s2l1 + s2l2 + s2l3 + s2l4 + s2l5 + s2l6 = 1
  soa3: s3l0 + s3l1 + s3l2 + s3l3 + s3l4 + s3l5 + s3l6 = 1
  soa4: s4l0 + s4l1 + s4l2 + s4l3 + s4l4 + s4l5 + s4l6 = 1
  soa5: s5l0 + s5l1 + s5l2 + s5l3 + s5l4 + s5l5 + s5l6 = 1
  soa6: s6l0 + s6l1 + s6l2 + s6l3 + s6l4 + s6l5 + s6l6 = 1

integers
  s0l0 s0l1 s0l2 s0l3 s0l4 s0l5 s0l6 s1l0 s1l1 s1l2
  s1l3 s1l4 s1l5 s1l6 s2l0 s2l1 s2l2 s2l3 2l4 s2l5
  s2l6 s3l0 s3l1 s3l2 s3l3 s3l4 s3l5 s3l6 s4l0 s4l1
  s4l2 s4l3 s4l4 s4l5 s4l6 s5l0 s5l1 s5l2 s5l3 s5l4
  s5l5 s5l6 s6l0 s6l1 s6l2 s6l3 s6l4 s6l5 s6l6
end
```

Literaturverzeichnis

[AC76] F. E. Allen and J. Cocke. A program data flow analysis procedure. *Commun. ACM*, 19:137–, March 1976.

[BB01] Barry W. Boehm and Victor R. Basili. Software defect reduction top 10 list. *IEEE Computer*, 34:135–137, 2001.

[BW08] Raymond P.L. Buse and Westley R. Weimer. A metric for software readability. In *Proceedings of the 2008 international symposium on Software testing and analysis*, ISSTA '08, pages 121–130, New York, NY, USA, 2008. ACM.

[Dei85] Lionel E. Deimel, Jr. The uses of program reading. *SIGCSE Bull.*, 17:5–14, June 1985.

[DG79] Hubert E. Dunsmore and John D. Gannon. Data referencing: An empirical investigation. *Computer*, 12:50–59, 1979.

[Dij97] Edsger W. Dijkstra. *A Discipline of Programming.* Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition, 1997.

[EM82] James L. Elshoff and Michael Marcotty. Improving computer program readability to aid modification. *Commun. ACM*, 25:512–521, August 1982.

[Fle48] Rudolph Flesch. A new readability yardstick. *Journal of Applied Psychology*, 32(3):221–233, 1948.

[Fow99] Martin Fowler. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley, Boston, MA, USA, 1999.

[GJSB05] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification, Third Edition.* Addison-Wesley Longman, Amsterdam, 3 edition, June 2005.

[Han98] Nuzhat J. Haneef. Software documentation and readability: a proposed process improvement. *SIGSOFT Softw. Eng. Notes*, 23:75–77, May 1998.

[HTC95] A. E. Hatzimanikatis, C. T. Tsalidis, and D. Christodoulakis. Measuring the readability and maintainability of hyperdocuments. *Journal of Software Maintenance*, 7:77–90, March 1995.

[Kan02] Stephen H. Kan. *Metrics and Models in Software Quality Engineering.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[McC78] Carma L. McClure. A model for program complexity analysis. In *Proceedings of the 3rd international conference on Software engineering*, ICSE '78, pages 149–157, Piscataway, NJ, USA, 1978. IEEE Press.

[MMNS83] Richard J. Miara, Joyce A. Musselman, Juan A. Navarro, and Ben Shneiderman. Program indentation and comprehensibility. *Commun. ACM*, 26:861–867, November 1983.

[Ray91] Darrell R. Raymond. Reading source code. In *Proceedings of the 1991 conference of the Centre for Advanced Studies on Collaborative research*, CASCON '91, pages 3–16. IBM Press, 1991.

[Sin95] Raghu Singh. International standard iso/iec 12207 software life cycle processes. Technical report, ISO/IEC, 1995.

[Wey88] Elaine J. Weyuker. Evaluating software complexity measures. *IEEE Trans. Softw. Eng.*, 14:1357–1365, September 1988.

[XSZC00] M. Xenos, D. Stavrinoudis, K. Zikouli, and D. Christodoulakis. Object-oriented metrics - a survey, 2000.