



Fachbereich 3
Mathematik - Informatik
Universität Bremen

Klonmanagement

Klonerkennung für
eingebettete Systeme

Diplomarbeit am Fachbereich Mathematik und Informatik

Prof. Dr. R. Koschke

Arbeitsgruppe Softwaretechnik

Universität Bremen

von

cand. inform.

Bernhard Johannes Berger

Tag der Anmeldung: 30. Juli 2007

Tag der Abgabe: 20. Dezember 2007

Danksagung

Ich bedanke mich bei allen, die mich bei meiner Diplomarbeit unterstützt haben. Natürlich vor allem bei meiner Familie, Catharina und Mara Sophie, die in dieser Zeit viel auf mich verzichten mussten. Darüber hinaus danke ich auch meinen Betreuern bei Bosch, sowie allen mit denen ich dort in dieser Zeit zu tun hatte. Nicht zu vergessen, mein Betreuer der Universität, Herr Professor Koschke und auch ein Dank an die, die ich vergessen habe.

Inhaltsverzeichnis

1	Einleitung	1
1.1	Hintergrund	1
1.2	Aufgabenstellung	3
1.3	Aufbau der Arbeit	3
2	Grundlagen	4
2.1	Klone	4
2.1.1	Klonaufbau	5
2.1.2	Klontyp	6
2.2	Klonerkennung	8
2.2.1	Abstraktionsebene	8
2.2.2	Granularität	9
2.2.3	Verfahren	10
2.3	Softwaremetriken	14
2.3.1	Skalenniveau	15
2.3.2	Verwendete Metriken	18
2.4	Refactorings	19
3	Analyse	20
3.1	Bauhaus	20
3.1.1	Intermediate Language	21
3.1.2	Resource Flow Graph	22
3.1.3	Klonerkennungstools	23
3.1.4	Visualisierung	24
3.1.5	Zusätzlich verwendete Programme	30
3.2	Anwendungsfälle	30
3.3	Probleme	31
3.3.1	Klonerkennung	31
3.3.2	Visualisierung	33
3.3.3	Entfernbarkeit	33
3.3.4	Toolkette	34
3.4	Aufgabenstellung	34
4	Umsetzung	36
4.1	Repräsentation der Klonklassen	36
4.2	Darstellung als <i>RFG</i>	39

4.3	Bewertung der Klonklassen	41
4.3.1	Verwendete Metriken	43
4.3.2	Verwendete Filter	46
4.4	Metrikboxen	47
5	Auswertung	50
5.1	Statistik	50
5.2	Ergebnisraumminimierung	52
5.3	Einsparpotential der Klonklassen	53
6	Fazit	61
6.1	Überblick	61
6.2	Interpretation der Ergebnisse	62
6.3	Schwachstellen	63
6.4	Verbesserungen	65
A	ICC-Syntax	66

Abbildungsverzeichnis

2.1	Beispielklonpaare für Typ 1 und 2	6
2.2	Beispielklonpaare für Typ 3 und 4	7
2.3	Klontypen	8
2.4	Aufbau der Skalenniveaus	15
3.1	Erstellen eines <i>RFG</i>	22
3.2	<i>GraVis</i> - Hauptfenster	25
3.3	<i>GraVis</i> - Graphfenster	26
3.4	<i>GraVis</i> - Metrikansicht	27
3.5	<i>GraVis</i> - Clone Pair Format Ansicht	28
3.6	<i>GraVis</i> - Kloninformationen importiert aus einer <i>gxl</i> -Datei	29
3.7	<i>GraVis</i> - Markierter Typ 2 Klon	30
3.8	Verwendete Programme von Bauhaus	35
4.1	Darstellung einer Klonklasse mit fünf Fragmenten	37
4.2	Hierarchischer Klonklassenaufbau	38
4.3	Klonklassenstruktur	38
4.4	Darstellung der Klonklasse mit Hilfe eines <i>RFG</i>	40
4.5	<i>RFG</i> mit Kloninformationen	42
4.6	Metrikbox - Anzahl an <i>IML</i> -Knoten und Anzahl an Parametern	48
4.7	Metrikbox - Anzahl an Fragmente und Anzahl an Parametern	48
4.8	Metrikbox - Anzahl an Fragment und durchschnittliche Anzahl an <i>IML</i> -Knoten	49
5.1	Häufigkeit der Klonklassen vor der Filterung	51
5.2	Häufigkeit der Klonklassen nach der Filterung	52
5.3	Auswirkung der einzelnen Filterungen	54
5.4	<i>Number of different Types</i> der Klonklassen	55
5.5	<i>Number of Parameters</i> der Klonklassen	56
5.6	<i>Number of Nodes</i> gegen <i>Number of Fragments</i> abgetragen	56
5.7	<i>Number of Nodes</i> in Relation zu <i>Number of Fragments</i> und <i>Number of Parameters</i>	57
5.8	<i>Average Number of Lines</i> in Relation zu <i>Average Number of Nodes</i>	58
5.9	Metrikwerte	59

Tabellenverzeichnis

2.1	Klassifizierung von diff	10
2.2	Klassifizierung von dup	12
2.3	Klassifizierung von CloneDR™	13
2.4	Klassifizierung von duplex	13
2.5	Klassifizierung des metrikbasierten Ansatzes	14
2.6	Beschreibung der Nominalskala	16
2.7	Beschreibung der Ordinalskala	16
2.8	Beschreibung der Intervallskala	17
2.9	Beschreibung der Rationalskala	17
2.10	Beschreibung der Absolutskala	17
3.1	Vergleich der wichtigsten Eigenschaften der Klonerkennungstools	32
5.1	Verteilung der Klonklassen vor der Filterung	51
5.2	Anzahl der gefilterten Klonklassen	52
5.3	Verteilung der Klonklassen nach der Filterung	53
5.4	Metrikwerte der ersten entfernten Klonklassen	60

1 Einleitung

*Wir lernen die Menschen nicht kennen,
wenn sie zu uns kommen;
wir müssen zu ihnen gehen,
um zu erfahren, wie es mit ihnen steht.*

Johann Wolfgang von Goethe

Dieses Kapitel soll den Hintergrund, die Aufgabenstellung und die Motivation, die zu dieser Diplomarbeit geführt haben, erklären. Zunächst wird im ersten Abschnitt das Umfeld und die Motivation dargelegt und im zweiten Abschnitt dann das Ziel der Arbeit beschrieben. Der Schluss legt den Aufbau der Diplomarbeit dar, um einen Überblick über dieses Dokument zu geben.

1.1 Hintergrund

Je mehr unser alltägliches Leben von Software abhängt, desto größer wird die Zahl der Softwaresysteme, die gewartet oder an neue Anforderungen angepasst und weiterentwickelt werden müssen. Empirische Studien haben gezeigt, dass die Kosten für die Erstentwicklung der Software nur einen kleinen Teil der Gesamtkosten, die für eine Software aufgewendet werden, ausmacht (unter anderem: [Zelkowitz u. a., 1979](#); [Erlikh, 2000](#)). In diesem Weiterentwicklungsprozess wird mit zunehmendem Alter und Größe der Software die Wartbarkeit ein immer schwerwiegenderes Problem, gerade in den Fällen, in denen bei der Planung des Systems diese Weiterentwicklung nicht antizipiert wurde.

Als Grund für diese hohen Kosten wird unter anderem Softwareerosion genannt, die durch schlechte Planung, mangelnde Möglichkeit der langfristigen Planung und hohem Zeitdruck entsteht. Dieser Begriff fasst eine ganze Reihe von Alterungserscheinungen des Quellcodes und seiner Struktur, also seiner inheränten Softwarearchitektur, zusammen. Um diesem Prozess entgegenzuwirken, wurden verschiedene Analysen entwickelt und nach geeigneten Gegenmaßnahmen gesucht. Hierzu gehört die Validierung der Softwarearchitektur, Metrikerhebung aber auch Klonerkennung. Diese und weitere Maßnahmen sollen die mittel- und langfristige Stabilität und Wartbarkeit der Software gewährleisten.

Durch Klone, also Quelltextkopien, kann sich Wissen über Implementierungsdetails im gesamten System verteilen, aber natürlich können auch Fehler in den kopierten Bereichen

dupliziert und somit über das ganze System verstreut werden, was die Fehleranfälligkeit des Systems steigert. Zudem wird die Codebasis künstlich vergrößert, was das Verständnis des Gesamtsystems erschwert und somit zu einer längeren Einarbeitungszeit in fremden Code führt. Die Auswirkungen von Klonen auf die Wartbarkeit und die Zuverlässigkeit wurde in [Monden u. a. \(2002\)](#) untersucht.

Heutige Klonerkennungsverfahren liefern eine große Anzahl an potentiellen Klonen, also vermeintlich „kopierte“ Quellcodeabschnitte, als Ergebnismenge. Diese Ergebnismenge ist so groß, dass eine manuelle Inspektion aller gemeldeten Treffer nicht in angemessener Zeit realisierbar ist. Allerdings ist eine computerunterstützte Analyse aller Kandidaten nicht durchgängig möglich, außerdem lassen sie sich nicht immer entfernen. Durch ein rigoroses Entfernen aller vorhandenen Klone würden zudem exzessive Tests notwendig werden, in denen geprüft werden müsste, ob sich bei der Entfernung der Klone nicht Fehler in das System eingeschlichen haben.

Diese Diplomarbeit entstand in Kooperation mit der Forschungsabteilung der Robert-Bosch GmbH in Schwieberdingen. Bei der Robert Bosch GmbH existieren Softwaresysteme, die bereits über mehrere Jahre weiterentwickelt wurden, so dass sich das System sowie dessen Architektur ständig an die neuen Anforderungen, wie neue Hardware oder Technologien, anpassen musste.

Als zu analysierendes Softwaresystem für die vorliegende Arbeit diente eine Variante der Steuerungssoftware für die Engin Control Unit, kurz ECU, von Dieselmotoren. Bei diesem System handelt es sich um ein eingebettetes System, was Auswirkungen auf nichtfunktionale Anforderungen wie Laufzeit und Speicherverbrauch der Software hat. Die Software ist komplett in der Programmiersprache C geschrieben und entspricht dem ISO-Standard [ISO/IEC 9899:1990 \(1990\)](#) und muss zudem die Richtlinien des Misra-C Standards ([Misra \(2004\)](#)) einhalten. Zu den Unterschieden zwischen Open- und Closedsource-Systemen gibt es eine Reihe von Untersuchungen, unter anderem in [Paulson u. a. \(2004\)](#).

Die Aufgabe einer ECU ist es, im laufenden Betrieb ständig Daten verschiedener Sensoren zu erfassen und auf Grund des Ist-Zustands und des Soll-Zustands eine geeignete Korrektur mit Hilfe von Aktoren vorzunehmen. Da nicht für jeden unterstützten Motor eine eigene Steuerungssoftware programmiert werden soll, zeichnet sich die Codebasis durch eine hohe Varianz aus, die zum Teil mit Hilfe des C-Präprozessors umgesetzt ist. Die analysierte Variante umfasst circa 340.000 Anweisungs- und 350.000 Kommentarzeilen und existiert seit mehr als zehn Jahren. Das System wird fortlaufend weiterentwickelt, mit neuen Funktionalitäten versehen oder um neue Zielplattformen erweitert und ist somit hinreichend alt und verändert, um Anzeichen von Softwareerosion zu enthalten.

Um diesen Alterungserscheinungen der Software vorzubeugen und Einhalt zu gebieten, sollen nun verschiedene Maßnahmen ergriffen werden. Hier lassen sich natürlich alle bekannten Techniken wie Architekturvalidierung, Metriken oder Klonerkennung verwenden. Wie bereits erwähnt, liefert die Klonerkennung lediglich eine Liste von potentiellen

Klonkandidat, die bei manueller Inspektion eine ganze Reihe von falsch Positiven enthält. Die gefundenen Ergebnisse sind aber nicht unbedingt dafür geeignet, sie mit Hilfe eines Refactorings zu entfernen. Ziel dieser Diplomarbeit ist es, eine Verbesserung dieser Unzulänglichkeiten herbeizuführen.

1.2 Aufgabenstellung

Da der Weg alle Klone zu entfernen in bestehenden Altsystemen nicht gangbar ist, soll eine Möglichkeit geschaffen werden, die Klonproblematik zu kontrollieren. Hier setzt die Diplomarbeit an und analysiert, wie dies geschehen kann. Natürlich kann im Rahmen dieser Arbeit nicht das gesamte Themengebiet des Klonmanagements, das Inhalt der Diplomarbeit ist, abgedeckt werden. Ziel ist es, Lösungsansätze aufzuzeigen und die Wirksamkeit einiger Maßnahmen zu untersuchen. Das Thema Klonmanagement wurde unter anderem schon von [Lague u. a. \(1997\)](#), [Giesecke \(2003\)](#) und [Chiu und Hirtle \(2007\)](#) bearbeitet.

Es wird versucht, einem Wartungsprogrammierer die Möglichkeit zu geben, effizient herauszufinden, wo in dem von ihm zu verbessernden System Klone enthalten sind, wo sich die anderen Kopien befinden und ihm darüber Auskunft geben, ob er einen Klon entfernen kann. Hierfür wird zunächst analysiert, wie Bauhaus für diesen Einsatzzweck verwendet werden kann und wo noch Unzulänglichkeiten existieren, zu deren Lösung im Rahmen dieser Diplomarbeit Ansätze aufgezeigt werden sollen.

1.3 Aufbau der Arbeit

Nach der Einleitung in diesem Kapitel werden zunächst in Kapitel 2 die Grundlagen geliefert, die notwendig sind, um die Arbeit thematisch einordnen und verstehen zu können. Im anschließenden Kapitel 3 folgt ein Überblick über Bauhaus und die in diesem Kontext bereits existierenden Möglichkeiten, die gestellte Aufgabe zu lösen. Außerdem werden hier mögliche Schwachstellen identifiziert. Die dafür entworfenen Lösungsansätze werden in Kapitel 4 beschrieben, um danach die Ergebnisse in Kapitel 5 der Arbeit auszuwerten und beurteilen zu können, welche Auswirkungen diese Verbesserungen haben. In einem abschließenden Fazit in Kapitel 6 werden die Ergebnisse zusammengefasst und beurteilt. Zudem werden mögliche Weiterentwicklungen und neu aufgetretene Fragen angesprochen sowie Ideen für deren Lösung geliefert.

2 Grundlagen

*Seht, da kommt er schleppend wieder!
Wie ich mich nur auf dich werfe,
gleich, o Kobold, liegst du nieder.
Krachend trifft die glatte Schärfe.
Wahrlich! brav getroffen!
Seht, er ist entzwei!
Und nun kann ich hoffen,
und ich atme frei!*

*Wehe! wehe!
beide Teile
stehn in Eile
schon als Knechte
völlig fertig in die Höhe!
Helft mir, ach! ihr hohen Mächte!*

Johann Wolfgang von Goethe

In diesem Kapitel werden zunächst die grundlegenden Begrifflichkeiten von Softwareklonen sowie die bekanntesten Klonerkennungsverfahren eingeführt. Es folgt eine Erläuterung von Metriken und den zu Grunde liegenden Skalen sowie der in der Arbeit verwendeten Standardmetriken. In einem abschließenden Abschnitt werden noch Refactorings vorgestellt, um damit eine ausreichende Wissensgrundlage zu geben, die für das Verständnis der nachfolgenden Kapitel notwendig ist.

2.1 Klone

Schlägt man im Duden den Begriff „Klon“ nach, so findet man die Definition, dass ein Klon ein durch Klonen entstandenes Lebewesen ist. Unter Klonen versteht man die Herstellung von genetisch identischen Kopien von Lebewesen durch ungeschlechtliche Vermehrung. Diese Definition stammt aus der Biologie und lässt sich leider nicht so einfach und vor allem präzise in der Informatik wiederfinden. Hier gibt es für den Begriff des (Software-)Klons keine einheitliche Definition und so fasst Ira Baxter seine Definition relativ allgemein.

Clones are segments of code that are similar according to some definition of similarity.

Ira Baxter, 2002 (zitiert nach: [Falke u. a., 2008](#))

Die Ähnlichkeitsdefinition wird hierbei nicht weiter ausgeführt, was einen großen Interpretationsspielraum lässt, der im Folgenden gefüllt werden soll, damit es hier nicht zu Missverständnissen kommen kann. Diese Definition wird in verschiedenen wissenschaftlichen Arbeiten verwendet unter anderem in [Bellon \(2007\)](#). Dieser Artikel vergleicht die Ergebnisse verschiedener Klonerkennungstools und die im Folgenden wiedergegebene Terminologie wurde dort von allen Tools verwendet.

2.1.1 Klonaufbau

Da Klone bisher nur informell als Quelltextkopie bezeichnet wurden, soll im Folgenden definiert werden, was innerhalb dieser Diplomarbeit als Klon bezeichnet wird. Darüber hinaus wird der Zusammenhang verschiedener geklonter Bereiche erläutert und das ihnen zu Grunde liegende Ähnlichkeitsmaß.

Codefragment

Bei einem Codefragment handelt es sich um einen Quelltextabschnitt. Dieser definiert sich durch den Namen der Datei, in der sich das Fragment befindet, Anfangs- und Endzeile sowie Anfangs- und Endspalte des Codefragments. Hierdurch wird ein Quelltextbereich beschrieben, der mit Hilfe dieser fünf Daten eindeutig zu identifizieren ist. Zudem kann man auf Basis dieser Daten prüfen, ob zwei Codefragmente gleich sind, was dann der Fall ist, wenn alle fünf Daten identisch sind.

Klonpaar

Ein Tupel von zwei Codefragmenten ergeben zusammen ein Klonpaar. Im Bauhauskontext werden die Begriffe Klon und Klonpaar häufig synonym verwendet, was zu Verwirrung bei der Benutzung von Bauhaus führen kann. Diese Paare haben neben den beiden Codefragmenten zusätzlich noch einen Klontyp, der die Art des Paares beschreibt. Näheres hierzu in Abschnitt [2.1.2](#). Klonpaare sind sowohl symmetrisch als auch transitiv. Dies bedeutet, dass, wenn es ein Klonpaar $CP_1 = (CF_1, CF_2)$ gibt, existiert auch das Klonpaar $CP_X = (CF_2, CF_1)$. Durch die Codefragmente ergibt sich die Transitivität. Wenn also die Klonpaare $CP_1 = (CF_1, CF_2)$ und $CP_2 = (CF_2, CF_3)$ vorhanden sind, so gibt es auch ein Klonpaar $CP_3 = (CF_1, CF_3)$.

Klonklassen

Während ein Klonpaar immer ein Paar von Codefragmenten umfasst, kann eine Klonklasse eine beliebige Anzahl an Codefragmenten beinhalten, die größer als eins ist. Somit handelt es sich hierbei um ein allgemeineres Konzept, da auch Klonpaare durch Klonklassen dargestellt werden können. Diese Klonklassen haben neben ihren Codefragmenten ebenfalls einen Klontypen, um so die Ähnlichkeit anzugeben.

2.1.2 Klontyp

Wie bereits erwähnt, beschreibt der Klontyp die Art eines Klonpaares oder einer Klonklasse und entspricht somit dem Ähnlichkeitsmaß in der bereits genannten Klondefinition. In diesem Kontext beschreibt der Klontyp die syntaktische Ähnlichkeit der Codefragmente. Insgesamt wird zwischen vier verschiedenen Klonklassen unterschieden, die hier beschrieben werden.

Typ-1

Bei einem Klon diesen Klontyps handelt es sich um eine exakte Kopie des Quelltextes. In dieser Kopie wurden lediglich Whitespaces oder Kommentare geändert, die bei den meisten Programmiersprachen keinen Einfluss auf das Programmverhalten haben¹. Ein Beispiel ist in Abbildung 2.1a zu sehen.

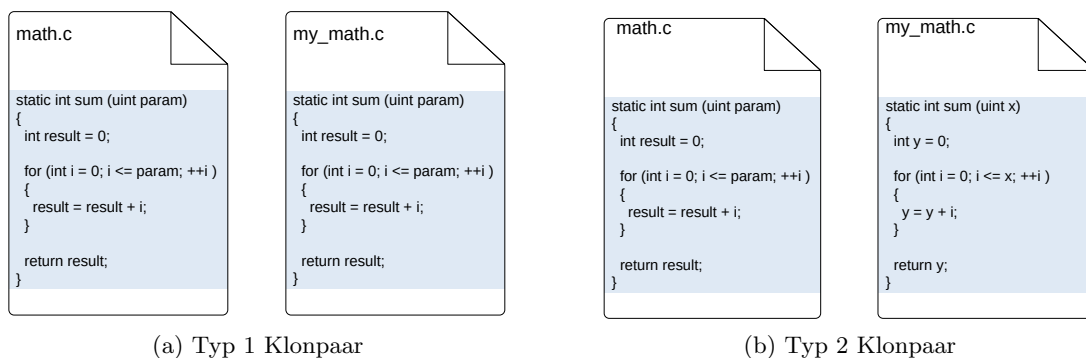


Abbildung 2.1: Beispielklonpaare für Typ 1 und 2

Typ-2

Bei diesem Klontyp handelt es sich im Grunde um einen Typ 1 Klon, bei dem nach dem Kopieren Bezeichner konsistent umbenannt wurden. Formal gesprochen existiert eine

¹Ausgenommen Sprachen wie Haskell, Python oder Whitespace.

bijektive Abbildung, die jedem Bezeichner aus einem Codefragment einen Bezeichner des anderen Codefragments zuordnet und umgekehrt. Ein solcher Klon ist in Abbildung 2.1b dargestellt. Manchmal werden hierzu auch noch die Codefragmente gezählt, in denen die Umbenennung nicht konsistent durchgeführt wurde. Hierdurch können Fehler ausgeblendet werden, die beim Umbenennen der Variablen passiert sind und so zu einer inkonsistenten Umbenennung geführt haben. In diesen Fällen ist die erwähnte Abbildung nicht bijektiv.

Typ-3

Bei Typ 3 Klonen handelt es sich um einen Verbund von mehreren Klonen der beiden vorhergegangenen Typen. Hierbei wird erlaubt, dass zwischen den zwei Klonen Code liegen darf, der nicht identisch ist. Damit lassen sich Klone aufspüren, die leicht modifiziert wurden, was zum Beispiel durch eine Fehlerbehebung passieren kann. Ein Klonpaar diesen Typs ist in Abbildung 2.2a dargestellt.

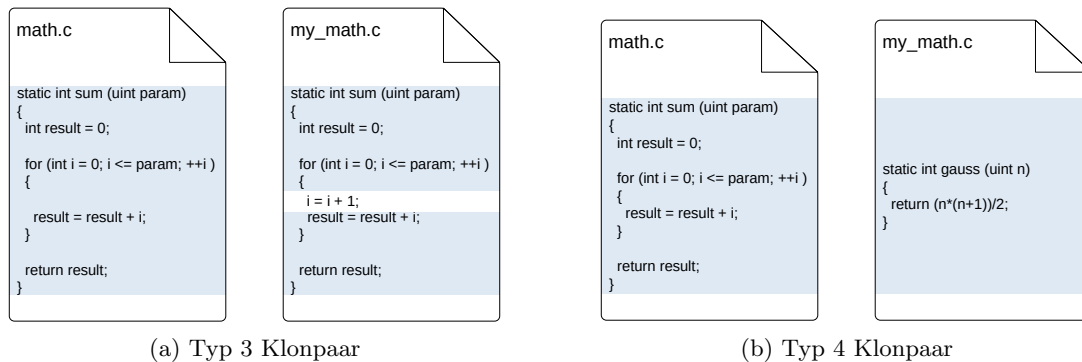


Abbildung 2.2: Beispielklonpaare für Typ 3 und 4

Typ-4

Dieser Klontyp beschreibt semantische Klone, die nur noch ihrem Verhalten nach gleich sind. Diese Klone zu identifizieren ist höchstens in einfachen Fällen möglich. Ein Beispiel für einen Typ 4 Klon, welcher allerdings nicht ohne weiteres identifiziert werden kann, ist in Abbildung 2.2b abgebildet.

Zusammenhang der Klontypen

Schon der Definition der Klontypen ist zu entnehmen, dass sie in Zusammenhang stehen. Die Klontypen verbindet eine Teilmengenbeziehung, die in Abbildung 2.3 dargestellt ist.

$$Typ1 \subset Typ2 \subset Typ3 \subset Typ4$$

Die bei Klonen vom Typ 2 geforderte Abbildung ist bei Klonen vom Typ 1 die Identitätsabbildung, die jedem Bezeichner sich selbst zuweist. Eine solche Zuordnung besteht auch bei den Typ 3 Klonen, nur dass hier Bereiche existieren, die nicht zugeordnet sind. Bei Typ 4 Klonen kann eine solche Zuordnung nicht mehr vorgenommen werden, da hier keine strukturelle Ähnlichkeit mehr besteht.

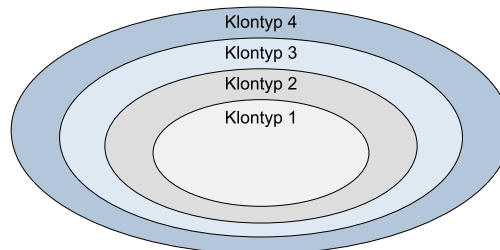


Abbildung 2.3: Klontypen

2.2 Klonerkennung

Nachdem die Definition von Klonen geliefert wurde, werden jetzt die verschiedenen Techniken, um Klone in Quelltexten zu identifizieren, vorgestellt. Die Verfahren unterscheiden sich hinsichtlich der erkannten Klontypen, der Programmiersprachenunabhängigkeit, dem Aufwand und damit der Dauer, der Abstraktionsebene und der Granularität. Die letzten beiden Punkte werden im Folgenden erläutert.

2.2.1 Abstraktionsebene

Die Abstraktionsebene gibt an, auf welcher Datenbasis die Klonerkennung arbeitet. Je mehr sie vom Originalquelltext abstrahiert, desto präzisere Aussagen können getroffen werden, da das Wissen über das Programm zunimmt und der Algorithmus weniger darauf angewiesen ist, dass sich die Syntax der Quelltexte gleicht. Allerdings kann hierdurch die Entfernung des entsprechenden Klons schwerer werden.

Ein Teil der Abstraktionsebenen sind vom Compilerbau und dem thematisch verwandten Gebiet der Softwareanalyse bekannt und werden im Folgenden aufgezählt.

1. (Quell-) Text
2. Token
3. Syntax (-baum)
4. Programmabhängigkeit
5. Metriken

Die Ebenen eins bis vier nehmen in ihrer Abstraktheit zu, da sie aufeinander aufbauen und für die Konstruktion einer Ebene die jeweils darunter liegende Ebene benötigt wird. Die letzte Abstraktionsebene der Metriken passt nicht in dieses Schema, da sie Merkmale auf allen der davor liegenden Ebenen oder auf noch anderen Datenbasen messen können.

Um Typ 2 Klone zu erkennen, muss die Klonerkennung zumindest auf der Tokenebene durchgeführt werden, da erst hier zuverlässig erkannt werden kann, welche Quelltextelemente Bezeichner sind und welche nicht. Der Hauptnachteil der höheren Abstraktionsebenen ist, dass mit zunehmender Abstraktion der algorithmische Aufwand für die Analyse steigt und damit auch eine längere Zeit in Anspruch nimmt. Dies führt zu einer langsameren Klonerkennung, was gerade bei großen Systemen spürbar wird. Zudem sinkt die Programmiersprachenunabhängigkeit, da mit zunehmenden Wissen über das Programm auch das Wissen über die Programmiersprache steigen muss, weil sich die Semantik verschiedener Programmiersprachen doch stark unterscheiden kann.

2.2.2 Granularität

Die Granularität einer Klonerkennung bezeichnet die kleinste mögliche Einheit, die bei der Klonerkennung berücksichtigt wird. Je feiner die Granularität des Verfahrens, desto exakter kann das Verfahren, allerdings mit steigendem Aufwand, arbeiten. Da im Prinzip jede Einheit einer Granularitätsstufe mit jeder anderen Einheit verglichen werden muss, ergibt sich bei n Einheiten ein Aufwand von $O(n^2)$. Dass quadratische Algorithmen für große Datenmengen sehr langsam werden, ist allgemein bekannt und wenn man sich den Quelltext von Softwaresystemen anschaut, so kann man feststellen, dass es sich hier im Allgemeinen um große Datenmengen handelt. Allerdings ist eine zu feine Granularität auch nicht notwendig, da die Qualität der Ergebnisse dadurch ab einem gewissen Punkt nicht mehr steigt.

1. Zeichen
2. Zeilen
3. Anweisung
4. Funktion
5. Modul

Die hier aufgezählten Granularitätsstufen werden üblicherweise für die Klonerkennung verwendet. Natürlich könnte man auch noch weitere Stufen definieren, was aber nicht unbedingt sinnvoll ist.

2.2.3 Verfahren

Nachdem die verschiedenen Abstraktionsebenen und Granularitätsstufen erklärt wurden, sollen nun für die verschiedenen Ebenen Referenzimplementierungen angegeben und deren Vor- und Nachteile dargestellt werden.

Textuell

Die einfachste Methode um nach kopiertem Quelltext zu suchen ist auf der Textebene, wo jede Zeile mit jeder anderen Zeile verglichen wird. Sofern sich mehrere gleiche Zeilen in Folge finden lassen, handelt es sich um einen Klon. Solche Algorithmen finden Verwendung bei Tools wie GNU diff², dessen Ziel es ist, den Unterschied zwischen zwei Textdateien herauszufinden.

Die Ähnlichkeit bei diesem Verfahren entspricht dem Verhältnis zwischen gleichen und unterschiedlichen Zeilen einer Datei. Dieses Verfahren eignet sich lediglich zur Lokalisation von kopierten Dateien und bedarf einiger „Handarbeit“, da das Programm nicht für diese Aufgabe konzipiert ist und man das Programm manuell für alle Paare von Dateien aufrufen muss.

diff	
Granularität	Modul
Abstraktionsebene	Text
erkannte Klontypen	Typ 1
Klonart	Klonpaare
Programmiersprachenunabhängigkeit	++
Aufwand	+

Tabelle 2.1: Klassifizierung von diff

Tokenstrom

Das wohl bekannteste Verfahren, dass Klone auf dem Tokenstrom erkennt, wurde von Brenda Baker ([Baker, 1995](#)) entwickelt und arbeitet auf Zeilenbasis. Die erste Implementierung des Ansatzes ist ihr Klonerkennungstool *dup*. Bei diesem Verfahren wird für die Erzeugung des Tokenstroms ein Lexer für die zu analysierende Sprache benötigt, der bei den meisten Programmiersprachen mit Hilfe von Lexergeneratoren relativ einfach zu erzeugen ist.

Der zu analysierende Tokenstrom wird nun auf gleiche Tokensequenzen untersucht, die einem Klon entsprechen. Die Klonerkennung wird über einen so genannten Suffixbaum realisiert, dieses Konzept ist in anderem Kontext auch unter dem Namen *Trie* bekannt.

²<http://www.gnu.org/software/diffutils>

Um den Suffixbaum zu erstellen, wird der Quelltext eingelesen und zu jeder Zeile aus dem Tokenstrom ein Parameter-String, oder kurz P-String, generiert, der dann im Suffixbaum gespeichert wird.

Bei der Generierung des P-Strings wird die Struktur der Token einer Zeile auf einen Funktor abgebildet. Dieser Funktor enthält keine konkreten Bezeichner mehr, da von ihnen abstrahiert wird. Jeder dieser Funktoren ist eindeutig, so dass alle Zeilen mit der gleichen Struktur auf den selben Funktor abgebildet werden. Die konkreten Bezeichner einer Zeile werden als Parameterliste für den Funktor dargestellt. So können gleiche Zeilen dadurch erkannt werden, dass sie den selben Funktor haben. Das folgende Beispiel stammt aus dem Skript der Veranstaltung Softwareengineering ([Koschke, 2006](#)).

$$x = x + y; \rightarrow (P = P + P; x, x, y) \rightarrow (\alpha xxy)$$

Hier ist die Umwandlung einer Quellcodezeile in den dazugehörigen Funktor und dessen Parameterliste zu sehen. In diesem Beispiel würden alle weiteren Zeilen, die die Form $P = P + P;$ haben, auf den Funktor α abgebildet.

Um nun von den Bezeichnern zu abstrahieren, werden diese in den Parameterlisten durch den Wert ersetzt, die die *prev*-Funktion erzeugt. Diese gibt an, wann das letzte Mal ein Parameter vorgekommen ist. Hierbei werden alle dazwischenliegenden Funktoren und Parameter gezählt. Ist ein Parameter noch nicht vorgekommen, so wird ihm die Zahl Null zugeordnet.

Mit den P-Strings und der *prev*-Funktion kann nun der Suffixbaum erstellt werden. Hierbei wird zunächst der P-String für den zu analysierenden Quelltext inklusive einem Eingabeendezeichen in den Suffixbaum eingetragen. An jeder Kante wird dabei ein Funktor und die *prev*-Werte der Parameterliste annotiert. Es wird solange der erste Funktor und die erste Parameterliste des P-Strings entfernt und das Ergebnis erneut in den Suffixbaum übertragen, bis die komplette Eingabe abgearbeitet ist. Klone können nun innerhalb des Suffixbaumes gefunden werden, indem man alle Wege von der Wurzel sucht, die mehrere Blätter haben. Dadurch, dass auf diesem Weg alle Funktoren und Parameter gleich sind, handelt es sich um Klone vom Typ 1 oder Typ 2. Abschließend muss noch geprüft werden, ob die Parameter gleich sind. Sollte dies der Fall sein, so handelt es sich um einen Typ 1 Klon.

Typ 3 Klone werden in einem Nachbearbeitungsschritt zusammengefasst. Hierfür werden mehrere gleiche Klone, die hintereinander liegen, zu einem Typ 3 Klon zusammengefasst. Ein Nachteil dieses Verfahrens ist die Tatsache, dass es zeilenbasiert arbeitet und sich durch einfache Umformatierungen täuschen lässt. Dies tritt vor allem in Projekten auf, in denen kein Styleguide existiert oder dieser nicht strikt befolgt wird.

dup	
Granularität	Zeile
Abstraktionsebene	Token
erkannte Klontypen	Typ 1 - Typ 3
Klonart	Klonpaare
Programmiersprachenunabhängigkeit	+
Aufwand	++

Tabelle 2.2: Klassifizierung von dup

Abstrakter Syntaxbaum

Ein Programm, das Klonerkennung auf Basis des abstrakten Syntaxbaums (AST) durchführt, ist *CloneDR*³, das von Ira D. Baxter entwickelt wird. Das Verfahren wird in [Baxter u. a. \(1998\)](#) vorgestellt.

Bei dem zu Grunde liegenden Verfahren wird zunächst zu dem analysierenden Quelltext ein abstrakter Syntaxbaum erzeugt. Um alle Klone zu finden, müsste jeder Teilbaum mit jedem Teilbaum auf Gleichheit oder Ähnlichkeit geprüft werden. Da dieser quadratische Aufwand vermieden werden soll, wird der Baum partitioniert. Hierbei wird für jeden Teilbaum ein Hashwert berechnet. Haben Teilbäume den gleichen Hashwert, so ist dies ein Indiz dafür, dass es sich hierbei um einen Klon handeln kann. Hierfür werden nun mit Hilfe einer Ähnlichkeitsfunktion alle Teilbäume mit gleichem Hashwert verglichen. Da bei der Hashwertberechnung die Bezeichner ignoriert werden, können mit diesem Verfahren ebenfalls Klone vom Typ 2 gefunden werden. Durch einen Nachbearbeitungsschritt werden nun mehrere Klone in Anweisungssequenzen gesucht und zusammengefasst, um nur maximale Klone zu melden. Hierbei können auch Typ 3 Klone identifiziert werden.

Der Vorteil bei diesem Ansatz ist, dass auf Grund des Wissens, das man dem AST entnehmen kann, die Ergebnisse präziser als die der anderen Tools sind. So können zum Beispiel kommutative Operatoren berücksichtigt werden und syntaktische Einheiten, also ganze Anweisungen, verglichen werden. Zudem werden bei diesem Verfahren nur syntaktisch vollständige Klone gemeldet. Ein nicht unüberwinderlicher Nachteil dieses Verfahrens ist, dass neben einem Lexer für die zu analysierende Sprache auch ein Scanner existieren muss. Hierdurch wird der Aufwand für die Klonerkennung höher und die Implementierung schwieriger.

Würde man eine perfekte Hashfunktion finden, die einen Teilbaum auf eine natürliche Zahl abbildet, so ließe sich ein Vergleich innerhalb der einzelnen Partitionen vermeiden. Eine solche Funktion ist schwer zu finden und der Computer arbeitet im Normalfall aus Performancegründen nicht mit allen natürlichen Zahlen, sondern nur mit einer Teilmenge. Dies kann dazu führen, dass zwei verschiedene Teilbäume auf den gleichen Hashwert abgebildet werden.

³<http://www.semdesigns.com/Products/Clone>

CloneDr™

Granularität	Anweisung
Abstraktionsebene	Syntaxbaum
erkannte Klontypen	Typ 1 - Typ 3
Klonart	Klonklassen
Programmiersprachenunabhängigkeit	—
Aufwand	—

Tabelle 2.3: Klassifizierung von CloneDr™

Program Dependence Graph

Bei der Klonerkennung mit Hilfe eines *PDG*, wie es Jens Krinke in seinem Tool *duplix* implementiert, werden neben den Informationen aus dem abstrakten Syntaxbaum noch Informationen über die Abhängigkeiten des Programms berücksichtigt. Hierdurch wird der Daten- und Kontrollfluss des Programms bei der Klonerkennung in Betracht gezogen und die Qualität der Ergebnisse noch weiter verbessert. In diesem PDG werden nun ähnliche Teilgraphen gesucht, die Klone entsprechen. Nähere Informationen zu dem Verfahren befinden sich in [Krinke \(2001\)](#), wo er sein Vorgehen vorstellt.

Wie bei dem Verfahren auf Basis des abstrakten Syntaxbaum ist auch hier der Aufwand zum Erzeugen der Analysegrundlage zeitintensiv. Hinzu kommt noch das weitaus aufwändigere Suchen nach Klone. Ein weiterer Nachteil besteht darin, dass die Analysen zur Erzeugung des abstrakten Syntaxbaums und der Bestimmung der Programmabhängigkeiten nicht programmiersprachenunabhängig sind, was die Klonerkennung für verschiedene Programmiersprachen erschwert.

duplix

Granularität	Anweisung
Abstraktionsebene	Programmabhängigkeit
erkannte Klontypen	Typ 1 - Typ 3
Klonart	Klonpaare
Programmiersprachenunabhängigkeit	—
Aufwand	—

Tabelle 2.4: Klassifizierung von duplix

Metrikbasiert

Während die bisher erläuterten Klonerkennungsverfahren ihre Informationen aus dem Quelltext oder Strukturen, die die Syntax und Semantik des Quelltextes enthalten, ableiten, geht der metrikbasierte Ansatz, der in [Mayrand u. a. \(1996\)](#) vorgestellt wird und den sie in ihrem Programm verfolgen, von bereits abstrahierten Informationen aus.

Bei diesem Verfahren werden zunächst für verschiedene Quelltextabschnitte Metriken erhoben und aus ihnen anschließend berechnet, ob zwei Quelltextabschnitte gleich oder ähnlich sind. Dieser Ansatz basiert auf der Annahme, dass bei gleichem Quelltext eine Metrik den gleichen Wert liefert und bei ähnlichem Code auch einen ähnlichen Metrikwert. Diese Annahme lässt sich ohne weiteres befürworten, allerdings ist bei der Klonerkennung nicht bekannt, ob zwei Quelltextabschnitt gleich oder ähnlich sind und so ist nicht immer klar, ob gleiche oder ähnliche Kennzahlen auf gleichen oder ähnlichen Code hindeuten.

Metrikbasierter Ansatz

Granularität	Funktion
Abstraktionsebene	Metriken
erkannte Klontypen	Typ 1 - Typ 3
Klonart	Klonpaare
Programmiersprachenunabhängigkeit	–
Aufwand	++

Tabelle 2.5: Klassifizierung des metrikbasierten Ansatzes

Zusammenfassung

In einem Vergleich aller bisher genannten Klonerkennungsverfahren ([Bellon, 2007](#)) stellt sich heraus, dass es kein Klonerkennungsstool und damit keinen Ansatz gibt, der besser geeignet ist Klone zu finden, als die anderen. Jedes Verfahren zeigt in diesem Vergleich sowohl Stärken als auch Schwächen. In Bezug auf das Entfernen von Klonen hat das Verfahren von Baxter den Vorteil, dass es nur syntaktisch vollständige Klone liefert. Die Ergebnisse sind aber auch abhängig von dem analysierten Projekt, so dass man für jedes Projekt und jede Aufgabe entscheiden muss, welches Verfahren die meisten Vorzüge hat.

2.3 Softwaremetriken

Was eine Softwaremetrik ist, wurde bereits durch das Institute of Electrical and Electronics Engineers (kurz IEEE) definiert und soll eine allgemeine Beschreibung für alle Softwaremetriken liefern.

Eine Softwaremetrik ist eine Funktion, die eine Software-Einheit in einen Zahlenwert abbildet.

(IEEE Standard 1061, 1992)

2.3.1 Skalenniveau

Nach der Definition des IEEE wird eine Software-Einheit auf einen Zahlenwert abgebildet. Diesen Zahlen liegt eine Ordnung zu Grunde, die Skalenniveau genannt wird und gibt an, welche Aussagekraft die Zahlen haben und welche Operationen auf ihnen ausgeführt werden können. Zwischen den Skalen existiert eine Ordnung, die in Abbildung 2.4 dargestellt ist. Die Nominalskala ist die Skala mit den wenigsten Möglichkeiten, während die Absolutskala die meisten Operationen erlaubt. Jede Skala erbt alle Eigenschaften, die die Skalen vor ihr schon hatten. Hierdurch werden die Skalen immer mächtiger und erhalten mehr Aussagekraft. Definitionen der Skalenniveaus lassen sich in vielen Büchern, wie zum Beispiel [Fahrmeir u. a. \(2007\)](#), finden.

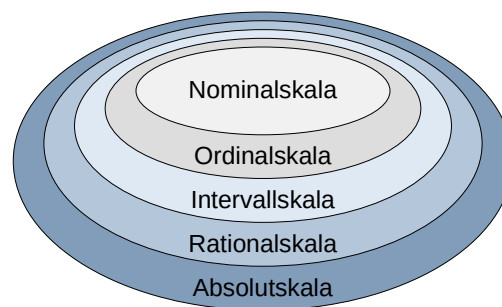


Abbildung 2.4: Aufbau der Skalenniveaus

Die Softwaremetriken als solche lassen sich wiederum weiter aufteilen, je nachdem, welche Eigenschaften sie messen. Fenton und Pfleeger ([Fenton und Pfleeger, 1998](#)) teilen sie in Prozessmetriken, Produktmetriken und Ressourcenmetriken ein, die sich ihrerseits wieder weiter unterteilen lassen. Unter den Produktmetriken finden sich die, die Software messen, während die anderen nur das Umfeld erfassen. Innerhalb der Produktmetriken wird hier noch zwischen den internen und externen unterschieden. Im Rahmen der Diplomarbeit werden einige interne Standardproduktmetriken verwendet, die später erläutert werden.

Nominalskala

Die Nominalskala ist das niedrigste Skalenniveau und ermöglicht lediglich die Entscheidung über Gleich- oder Ungleichheit. Neben dieser Gleichheitsrelation existieren keine weiteren Relationen, wie zum Beispiel eine Ordnungsrelation. Statistisch lässt sich nur aussagen, welches Maß wie häufig vorhanden ist. Die verschiedenen Werte der Skala werden Kategorien genannt.

Die Beispiele in Tabelle 2.6 zeigen, dass sich auf den gemessenen Werten keine weitere Operationen als die Identität durchführen lassen.

Nominalskala	ungeordnete Kategorien
Bedingungen	Symmetrie, Reflexivität, Transitivität
Operationen	Identität
Statistik	Häufigkeit
Beispiel	Studienort : Bremen $\neq$ Stuttgart Name : Peter $\neq$ Paul Wohnort : Bremen = Bremen

Tabelle 2.6: Beschreibung der Nominalskala

Ordinalskala

Bei einer Ordinalskala gibt es zu den Eigenschaften der Nominalskala noch zusätzlich eine Ordnungsrelation. Die Aussage dieser Relation hängt von der gemessenen Metrik ab, erlaubt aber immer einen Vergleich der gemessenen Werte. Statistisch gesehen lässt sich über die Kategorien der Median bilden, um so den im Mittel gemessenen Wert zu bestimmen.

Wie aus dem Beispiel in Tabelle 2.7 zu entnehmen ist, lässt sich aus den gemessenen Werten weder ein Verhältnis noch ein Abstand der Werte entnehmen.

Ordinalskala	geordnete Kategorien
Bedingungen	Konnexivität, Transitivität
Operationen	Vergleich der Kategorien
Statistik	Median
Beispiel	Temperatur: kalt $\leq$ warm $\leq$ heiß

Tabelle 2.7: Beschreibung der Ordinalskala

Intervallskala

Über der Ordinalskala steht die Intervallskala, in der einfache Rechenoperationen auf den gemessenen Werten erlaubt sind. Diese Operationen sind die Addition und die Subtraktion, die es erlauben, den Abstand zwischen Metrikwerten sinnvoll zu bestimmen. Dieser Abstand entspricht der „Ungleichheit“ zweier Metrikwerte. Der Nullpunkt als auch die Abstände zwischen den Werten sind hierbei willkürlich. Für diese Skala lässt sich der Mittelwert sowie die Standardabweichung berechnen.

Das für uns wohl bekannteste Beispiel einer solchen Skala ist die Temperatur in °C, wie sie in Tabelle 2.8 als Beispiel angeführt ist. Hier ist der Nullpunkt der Skala, also 0°C, willkürlich auf den Gefrierpunkt und 100°C auf den Siedepunkt von Wasser gelegt und die Temperaturspanne dazwischen in 100 äquidistante Abschnitte eingeteilt.

Intervallskala	Zahlen
Operationen	Addition, Subtraktion
Statistik	Mittelwert, Standardabweichung
Beispiel	Temperatur: $40^{\circ}\text{C} - 20^{\circ}\text{C} = 20^{\circ}\text{C}$

Tabelle 2.8: Beschreibung der Intervallskala

Rationalskala

Die vorletzte Skala in der Ordnung ist die Rationalskala, die anders als die Intervallskala einen absoluten Nullpunkt besitzt, der durch das gemessene Phänomen gegeben ist. Mit Hilfe der Multiplikation und Division, die zu den bisher genannten Operation hinzugekommen, lassen sich hier Verhältnisse von Merkmalswerten bilden.

Beispiele für Rationalskalen sind das Alter einer Person oder die Temperatur gemessen in $^{\circ}\text{K}$, wie in Tabelle 2.9 angeführt. Hier ergibt sich der absolute Nullpunkt, also 0°K durch die physikalische Gegebenheit, dass es keine Temperatur gibt, die niedriger ist.

Rationalskala	Zahlen mit absolutem Nullpunkt
Operationen	Multiplikation, Division
Statistik	geometrisches Mittel, Korrelation
Beispiel	Temperatur: $40^{\circ}\text{K} \cdot 2 = 80^{\circ}\text{K}$

Tabelle 2.9: Beschreibung der Rationalskala

Absolutskala

Die höchste Skala in dieser Ordnung ist die Absolutskala, die durch die gemessene Eigenschaft *natürlich* gegeben ist. Diese natürliche Gegebenheit lässt sich nicht anders beschreiben. Hierzu gehört die Anzahl der Elemente in einer Menge. Diese Eigenschaft lässt sich auf keine andere Art und Weise sinnvoll abbilden.

Absolutskala	natürlich gegeben
Operationen	absoluter Vergleich
Beispiel	Anzahl aller Menschen

Tabelle 2.10: Beschreibung der Absolutskala

2.3.2 Verwendete Metriken

In diesem Abschnitt sollen die Standardmetriken vorgestellt werden, die im Rahmen der Arbeit verwendet werden, um verschiedene Eigenschaften der zu analysierenden Software zu erfassen.

Lines of Code

Bei der Metrik *Lines of Code* oder kurz LOC handelt es sich um die Anzahl der Zeilen in einem bestimmten Bereich. Diese Metrik kann für verschiedene Quelltexteinheiten wie Modul, Funktion oder Anweisungsblock erhoben werden. Der Metrik liegt eine Absolutskala zu Grunde, da es sich hierbei um die Anzahl der Zeilen aus der Menge aller Zeilen einer Datei handelt.

McCabe Complexity

Die McCabe Komplexität drückt aus, wie komplex ein Quelltextbereich ist. Diese Metrik beruht auf dem Kontrollflussgraph und errechnet sich aus diesem durch $McCabe(G) = \#Kanten(G) - \#Knoten(G) + 2$. Da die Erzeugung eines Kontrollflussgraphen einen hohen Erstellungsaufwand bedeuten kann, gibt es noch einen einfacheren Weg, um zu diesem Ergebnis zu kommen $McCabe(G) = \#Binärverzweigungen + 1$. Auch dieser Metrik liegt eine Absolutskala zu Grunde.

Die dargestellte Metrik ist einfach zu berechnen, berücksichtigt jedoch nicht, wie komplex die einzelnen Anweisungen sind. Sie dient aber im Allgemeinen als recht guter Hinweis dafür, wie schwer ein Quelltextabschnitt zu verstehen ist, da eine große Zahl an Schleifen und Bedingungen die Verständlichkeit erschweren.

Nesting

Der Nesting-Wert eines Quelltextabschnittes ist seine maximale Schachtelungstiefe. Viele Programmiersprachen erlauben es, Blöcke zu definieren, die in anderen enthalten sind. Meist werden sie im Kontext von bedingten Ausdrücken oder Schleifen verwendet. Bei dieser Metrik wird nun analysiert, wie tief der Quelltext geschachtelt ist. Dies ist eine nützliche Information, vor allem im Zusammenhang mit der bereits genannten McCabe Komplexität, da man hierdurch entscheiden kann, ob die Verzweigungen nacheinander kommen oder ineinander liegen. Denn gerade letzteres sorgt dafür, dass ein Programmabschnitt schwer zu verstehen ist. Dieser Metrik liegt eine Rationalskala zu Grunde.

LIC, LEC und LCC

Die Werte *Lines of internal copied Code* LIC, *Lines of external copied Code* LEC und *Lines of copied Code* LCC können am besten gemeinsam betrachtet werden, da ihre Bedeutungen ähnlich sind. Während LIC angibt, wie viele Zeilen einer Datei innerhalb dieser kopiert wurden, gibt LEC an, wie viele Zeilen nach außerhalb der Datei kopiert wurden. Die letzte Metrik in dieser Sammlung, LCC, gibt an, wie viele Zeilen der Datei insgesamt vom Klonen betroffen sind.

Diese drei Werte geben einen Überblick, wie viel von einer einzelnen Datei durch Kopieren entstanden ist. Dabei muss beachtet werden, dass $LIC(M) + LEC(M) \geq LCC(M)$ gilt. Dies kommt daher, dass sowohl LIC als auch LEC Mengen von Quelltextzeilen eines Moduls M sind, die sich teilweise überschneiden können und die gemeinsamen Elemente bei der Vereinigung der beiden Mengen zusammenfallen. Dieser Metrik liegt eine Absolutskala zu Grunde.

2.4 Refactorings

Martin Fowler führt in seinem Buch Refactoring (Fowler, 1999) eine Reihe von so genannten *Bad Smells*, also schlechten Gerüchen, von Software ein. Mit diesem Begriff bezeichnet er Schwachstellen im Quelltext, die häufig im Code aufzufinden sind und negative Auswirkungen auf die Codequalität haben. Diese negativen Auswirkungen sind hier hauptsächlich Lesbarkeit, Verständlichkeit und Änderbarkeit der Software. Ein Teil dieser *Bad Smells* beziehen sich explizit auf objektorientierte Paradigmen, wie zum Beispiel *Large Class*, *Data Class* oder *Lazy Class*, so dass sie nicht in eine imperative Programmiersprache wie C übertragbar sind. Es existieren aber auch *Bad Smells* die sich durchaus übertragen lassen, wie zum Beispiel *Comments*, *Long Method* oder *Duplicated Code*. Unter dem letzten Begriff fasst er Klone zusammen und setzt diesen schlechten Geruch auf den ersten Platz seiner Aufzählung.

Neben den Problemen stellt Fowler jeweils auch mindestens eine Lösung vor, die ausführlich erläutert werden. Diese Lösungsschemata nennt er Refactorings und sie sollen dem Entwickler bei seiner täglichen Arbeit helfen. Einige dieser Refactorings sind in modernen Entwicklungsumgebungen wie Eclipse⁴ oder IntelliJ⁵ integriert und können einfach und vollautomatisch ausgeführt werden.

Da Fowler auch zum *Bad Smell Duplicated Code* Lösungsansätze vorschlägt, bietet dies eine gute Grundlage um nachzuvollziehen, welche Klone sich entfernen lassen und auf welche Art dies möglich ist.

⁴Im Internet zu finden unter <http://www.eclipse.org>

⁵Im Internet zu finden unter <http://www.intellij.com>

3 Analyse

Mit dem Wissen wächst der Zweifel.

Johann Wolfgang von Goethe

In diesem Kapitel soll untersucht werden, wie mit Bauhaus Klone erkannt und vor allem anschließend visualisiert werden können. Die Visualisierung ist zentraler Bestandteil, wenn man Klone konsequent beseitigen und im Besonderen überwachen möchte. Hierfür gibt es zunächst eine Einführung in die Bauhaus-Suite und der für die Lösung dieser Aufgabe relevanten Programme und Datenformate in Abschnitt 3.1. Anschließend werden in Abschnitt 3.2 die potentiellen Anwendungsfälle der Software identifiziert, um daraus in Abschnitt 3.3 die Tools, die zu deren Bearbeitung notwendig sind, abzuleiten und die Entscheidung für ein bestimmtes Tool bei mehreren Alternativen zu begründen. Abschließend soll auf Basis der Anwendungsfälle die Aufgabenstellung in Abschnitt 3.4 noch einmal wiedergegeben und detaillierter spezifiziert werden.

3.1 Bauhaus

Bauhaus ist eine Softwarewerkzeugkiste, die bei der Analyse und dem Verstehen von Software hilfreich ist. Sie ist für Windows, Solaris und Linux verfügbar und wird seit 1996 entwickelt. Das Bauhaus-Projekt wurde von der Universität Stuttgart¹ und dem Fraunhofer Institut für Experimentelles Software-Engineering² gegründet, wird aber inzwischen auch an der Universität Bremen³ und durch die Axivion GmbH⁴ weiterentwickelt.

Bauhaus unterstützt eine Reihe von Quellcodeanalysen, wie zum Beispiel Klonerkennung, Architekturvalidierung, Metrikerhebung und Zyklenerkennung im Aufrufgraph, um so auf potentielle Problemstellen der Software hinzuweisen und Stellen zu identifizieren, an der Softwareerosion Einhalt geboten werden kann. Durch die vorhandene Anbindung an Python lassen sich aber auch eigene Analysen oder die Validierung eigener Richtlinien einfach erstellen und durchführen.

¹<http://www.uni-stuttgart.de>

²<http://www.iese.fraunhofer.de>

³ <http://www.uni-bremen.de>

⁴<http://www.axivion.com>

Um ein Programm analysieren zu können, muss man den Herstellungsprozess der zu untersuchenden Software dahingehend anpassen, dass beim Übersetzen der Quelltext nicht an den Compiler sondern an das Analyse-Tool *cafeCC* übergeben wird. Die Aufrufparameter müssen hierfür im Allgemeinen nicht geändert werden, da es die Parameter der gängigen Compiler versteht. *cafeCC* erstellt nun aus dem Quelltext eine Bauhaus-interne Zwischendarstellung, genannt Intermediate Language (*IML*). Näheres hierzu findet sich in Abschnitt 3.1.1.

Da die *IML* immer das ganze Programm mit all seinen Programmkonstrukten darstellt und dies sehr schnell unübersichtlich wird und für weitere Analysen eine zu große Datenmenge ergäbe, kann aus der *IML* ein Resource Flow Graph (*RFG*) erzeugt werden. Damit werden unwichtige Details abstrahiert und nur noch die in- und externen Abhängigkeiten des Programms darstellt. Mehr zum *RFG* findet sich im Abschnitt 3.1.2.

Welche Möglichkeiten Bauhaus zur Klonerkennung bietet und welche Verfahren hier implementiert sind, wird in Abschnitt 3.1.3 ausgeführt. Mit diversen weiteren Kommandozeilen-Tools und der grafischen Oberfläche *GraVis* lassen sich die bereits erwähnten Analysen auf der *IML* und dem *RFG* durchführen. Eine kurze Erläuterung von *GraVis* erfolgt in Abschnitt 3.1.4. Weitere verwendete Tools und das Zusammenspiel aller Programme werden in Abschnitt 3.1.5 erläutert.

3.1.1 Intermediate Language

Bei der Programmmzwischendarstellung *IML* handelt es sich um einen annotierten Syntaxbaum. Der Begriff des Baumes ist in diesem Kontext allerdings nicht ganz treffend, da es sich auf Grund der syntaktischen und semantischen Kanten hier eigentlich um einen normalen Graphen handelt.

Um einen annotierten Syntaxbaum zu erhalten, wird im Allgemeinen zunächst der Quelltext in einen Tokenstrom umgewandelt und anschließend durch einen Parser in einen Parsebaum überführt, der dann in einen abstrakten Syntaxbaum (*AST*) umgewandelt wird. Während der Parsebaum noch strikt nach den Ableitungsregeln der Sprachgrammatik aufgebaut ist, abstrahiert der *AST* von diesen. So werden zum Beispiel rekursive Regeln der Grammatik im *AST* in Listen überführt oder überflüssige Knoten aus dem Baum entfernt. Im nächsten Schritt wird der *AST* in einen annotierten Syntaxbaum überführt. In diesem Schritt kommen zu den syntaktischen nun auch noch semantische Informationen, welche durch Kanten innerhalb des annotierten Syntaxbaums repräsentiert werden, die aus dem Baum einen allgemeinen Graphen werden lassen. In diesem annotierten Syntaxbaum hat man zusätzlich Informationen wie zum Beispiel Datentypen oder Sprungziele. Natürlich lassen sich hier noch beliebige andere Informationen erfassen.

Auf Basis dieser syntaktischen und semantischen Informationen lassen sich verschiedene Analysen durchführen wie Pointeranalyse, Daten- und Kontrollflussanalyse, Klonerkennung und Metrikerhebung. Sofern sich diese Analysen programmiersprachenunabhängig gestalten lassen besteht die Möglichkeit, diese auch auf allen unterstützten Sprachen

durchzuführen, da in der *IML* die Sprachkonstrukte der verschiedenen Sprachen auf eine allgemeine sprachunabhängige Darstellung abgebildet werden. Zur Zeit gibt es Analysefrontends für die Sprachen Ada, C, C++ und Java. Für diese Sprachen lassen sich auch entsprechende Analysen entwickeln.

3.1.2 Resource Flow Graph

Der *RFG* ist die zentrale Informationsquelle für *GraVis* und kann entweder aus der *IML* wie im Fall von Ada, C, C++ oder Java oder direkt aus einem Bytecode wie zum Beispiel bei Java oder .NET gewonnen werden.

Dem *RFG* liegt ein Graph zu Grunde, in dem sowohl die Knoten als auch die Kanten, welche immer gerichtet sind, frei typisiert werden können. Diese Typen lassen sich zur Laufzeit definieren und sofort verwenden. Da in einem *RFG* im Allgemeinen viele Informationen erfasst werden, gibt es das Konzept der Sicht (*View*). Eine *View* wird durch ihren Namen identifiziert und enthält eine Teilmenge aller Knoten und Kanten im *RFG* und stellt einen bestimmten Aspekt des Systems dar. Häufig werden *RFG*-basierte Analysen auf Basis einer *View* durchgeführt und erzeugen wiederum eine neue *View*, die das Ergebnis der Analyse enthält.

Neben den Knoten und Kanten kann ein *RFG* auch Attribute enthalten. Diese können dem Graph, den *Views*, den Kanten oder den Knoten zugewiesen werden, um auf diesem Weg weitere Informationen zu erfassen. Die Attribute können wiederum frei definiert werden, wobei sie von einem der fünf gegebenen Datentypen Togglewert, boolescher Wert, Integerzahl, Fließkommazahl oder String sein müssen. Mit diesem Konzept lassen sich beispielsweise Metriken oder Verweise auf den Quelltext annotieren, was zum Beispiel bereits beim Erstellen eines *RFG* gemacht wird.

Erzeugt man einen *RFG* aus einer der oben beschriebenen Quellen, so werden alle globalen Programmelemente wie Pakete, Klassen, Variablen etc., die die Struktur des Programms darstellen, auf Knoten und die Abhängigkeiten zwischen ihnen auf Kanten abgebildet. Wenn dies möglich ist, werden an die Knoten und Kanten auch die entsprechenden Quelltextposition annotiert, so dass man immer zu dem Quelltext springen kann, der zu diesem Objekt geführt hat.

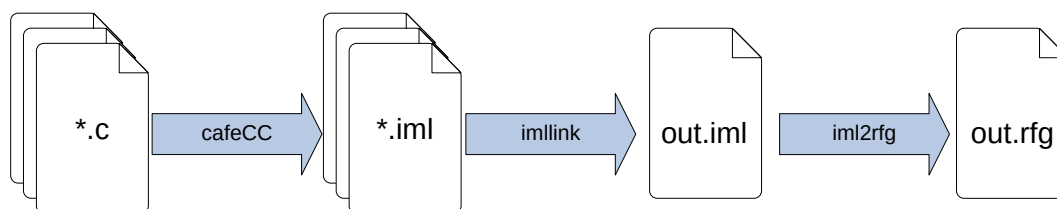


Abbildung 3.1: Erstellen eines *RFG*

Um einen *RFG* für ein C- oder C++-Programm zu erzeugen, muss man zunächst, wie in Abbildung 3.1 dargestellt die Quellcodedateien mit *cafeCC* übersetzen. Hieraus ergeben sich mehrere *IML*-Dateien, die mit dem Tool *imllink3* zu einer *IML* zusammengelinkt werden, die die Informationen des gesamten Programms enthält. Dieses Vorgehen unterscheidet sich nicht von dem eines normalen Compilers. Die gelinkte *IML* kann nun mit dem Programm *iml2rfg* in einen *RFG* überführt werden, auf dem weitere Analysen durchgeführt werden können oder den man sich mit *GraVis* anzeigen lassen kann.

3.1.3 Klonerkennungstools

Im Rahmen von Bauhaus gibt es insgesamt vier verschiedene Klonerkennungstools, die die Verfahren von Baker und Baxter, die in Abschnitt 2.2.3 erläutert wurden, implementieren und zum Teil sogar kombinieren, um so die Vorteile beider Verfahren miteinander zu verbinden.

clones

Das Tool *clones* implementiert den Ansatz von Baker und versucht Klone auf Basis des Tokenstroms zu finden. Es unterstützt die Programmiersprachen Ada, C, Cobol, C++, C#, Java und Visual Basic und beherrscht verschiedene Ausgabeformate, in denen die Ergebnisse gespeichert werden können. Es wurde gegenüber dem vorgestellten Verfahren um einige Möglichkeiten, wie Ergebnisfilter, erweitert, um in der gemeldeten Klonmenge uninteressante Klonpaare auszublenden.

Dieses Klonerkennungstool hat den Vorteil, dass es den Quelltext der Programmiersprachen C und C++ vor dem Präprozessieren analysiert und damit den wirklichen Quelltext in seiner Originalversion verwendet. Dies ist im Besonderen bei variantenreicher Software sinnvoll, da Varianten in der Programmiersprache C häufig mit Hilfe des Präprozessors abgebildet werden.

ccdimpl

ccdimpl implementiert den Ansatz von Baxter, also die Klonerkennung auf dem *AST*. Als Analysegrundlage dient die *IML*, die alle benötigten Informationen hierfür enthält. Da es auf der *IML* beruht, werden auch nur die Sprachen unterstützt, für die die *IML* generiert werden kann. Dies sind zum aktuellen Zeitpunkt Ada, C, C++ und Java.

Im Gegensatz zu *clones* verarbeitet *ccdimpl* den Programmtext erst nach dem Präprozessieren. Damit werden Klone nur für eine bestimmte Präprozessorkonfiguration erkannt. Teile, die auf Grund des Präprozessors entfernt werden, können nicht berücksichtigt werden.

cpdetector

Im Gegensatz zu den beiden vorherigen Tools wird bei *cpdetector* versucht die Ansätze von Baker und Baxter zu kombinieren. Hierfür wird eine *IML* geladen, die dann in einen Tokenstrom umgewandelt wird. Das Programm läuft die syntaktischen Kanten ab und generiert für jeden Knoten ein Token. Dieses Token entspricht dem Knotentyp und ist für alle Knoten des gleichen Typs eindeutig. Mit diesem Verfahren sollen die Vorteile der verschiedenen Ansätze kombiniert werden. Das ist zum einen die syntaktische Abgeschlossenheit der Klone und zum anderen die Geschwindigkeit der Klonerkennung.

clast

Das Tool *clast* verwendet den gleichen Ansatz wie *cpdetector* nur mit dem Unterschied, dass hierfür als Datenbasis nicht die *IML* sondern ein eigens hierfür erzeugter *AST* genutzt wird. Dies hat den Vorteil, dass zum einen mehr Programmiersprachen unterstützt werden können und dass hier die Konstrukte nicht auf einen sprachübergreifenden Kern abgebildet werden, bei der Detailinformationen über die Syntax verloren gehen. Die Frontends für die verschiedenen Sprachen müssen natürlich erstellt werden, was aber mit den verwendeten Tools *lex* und *yacc* ein eher geringer Aufwand im Vergleich zum Erstellen eines *IML*-Frontends ist. Eine genauere Beschreibung des Verfahrens und die Ergebnisse sind in [Falke u. a. \(2008\)](#) zusammengefasst.

3.1.4 Visualisierung

Bei *GraVis* handelt es sich um eine grafische Oberfläche, die in der Lage ist, einen *RFG* darzustellen und zudem die Möglichkeit bietet, auf diesem Graphen Operationen und Analysen durchzuführen. In Abbildung 3.2 sieht man das Hauptfenster von *GraVis*, in dem bereits ein *RFG* geöffnet ist. Man erkennt die verschiedenen *Views*, die in dem geladenen *RFG* enthalten sind und wie viele Knoten und Kanten sie enthalten. Durch einen Doppelklick auf den Namen einer *View* öffnet sich ein neues Fenster, welches ihren Inhalt darstellt. Enthält die *View* Kanten vom in der Hierarchiespalte angegebenen Typ, so werden die Knoten hierarchisch dargestellt. Dies erleichtert gerade bei großen Systemen die Übersicht, da man sich zunächst einen groben Überblick verschaffen und dann weiter hineinzoomen kann.

Das Hauptfenster erlaubt auch, einzelne *Views* zu sperren, so dass sie nicht geändert werden können, bis sie wieder entsperrt werden. Darüber hinaus kann einer *View* eine Rolle zugewiesen werden, was vor allem bei der Architekturanalyse von Interesse ist. Die letzten vier Spalten zeigen an, wie viele Knoten und Kanten in der entsprechenden *View* markiert und selektiert sind. Ein markiertes Objekt ist immer in allen *Views* markiert, was hilfreich ist, wenn man den Zusammenhang zwischen verschiedenen *Views* erkennen will. Eine Selektion bezieht sich immer nur auf eine bestimmte *View*, die allerdings mehrfach geöffnet sein kann sowie deren assoziierten Metrikansichten.

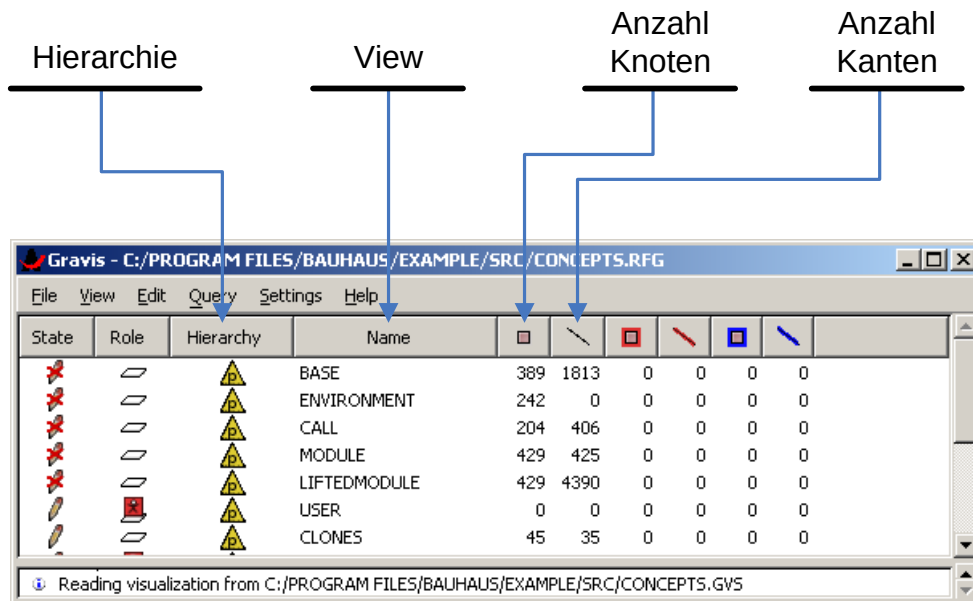


Abbildung 3.2: GraVis - Hauptfenster

Öffnet man eine *View*, so wird deren Inhalt grafisch in einer Graphansicht dargestellt, wie in dem Screenshot in Abbildung 3.3 zu sehen ist. Hier ist eine Modulansicht geöffnet, deren Kanten geliftet wurden. Dies sorgt dafür, dass in hierarchischen Ansichten die Kanten von nicht sichtbaren Knoten entlang der Hierarchie nach oben propagiert werden. Hierbei werden Kanten des gleichen Typs nur einmal dargestellt, so dass die Anzahl der Kanten abnimmt und man einen groben Überblick über die Abhängigkeiten im System bekommt. Öffnet man einen hierarchischen Knoten, so werden alle gelifteten Kanten, die von diesem Knoten ausgehen oder in ihm münden ausgeblendet und die Kanten der nächsten Hierarchieebene dargestellt.

Zunächst sind in der *View* eine Reihe von Knoten enthalten, von denen einer geöffnet ist. Die durch einen Slash gekennzeichneten Knoten stellen Verzeichnisknoten dar, während Knoten, die durch ein Dateisymbol abgebildet werden, für einzelne Module des Systems stehen, in diesem Fall für Dateien. Das Verzeichnis mit dem Namen *src* ist in dieser Abbildung geöffnet und man sieht die dort enthaltenen Knoten. Die Farben der Kanten zwischen den Knoten stehen für die verschiedenen Kantentypen, wie zum Beispiel Aufrufe, Variablenverwendung oder Typkanten.

Die Attribute, die an Knoten oder Kanten im *RFG* annotiert sind, lassen sich auf verschiedene Art und Weise visualisieren. Zum einen ist es möglich, die Werte an das betroffene Objekt zu annotieren wie in dem gezeigten Beispiel die Namen der Verzeichnisse und Module. Zum anderen kann durch die Integer-Attribute die Stärke einer Kante beeinflusst werden. Toggle-Werte lassen sich außerdem durch kleine Icons anzeigen, die über dem Knoten eingeblendet werden.

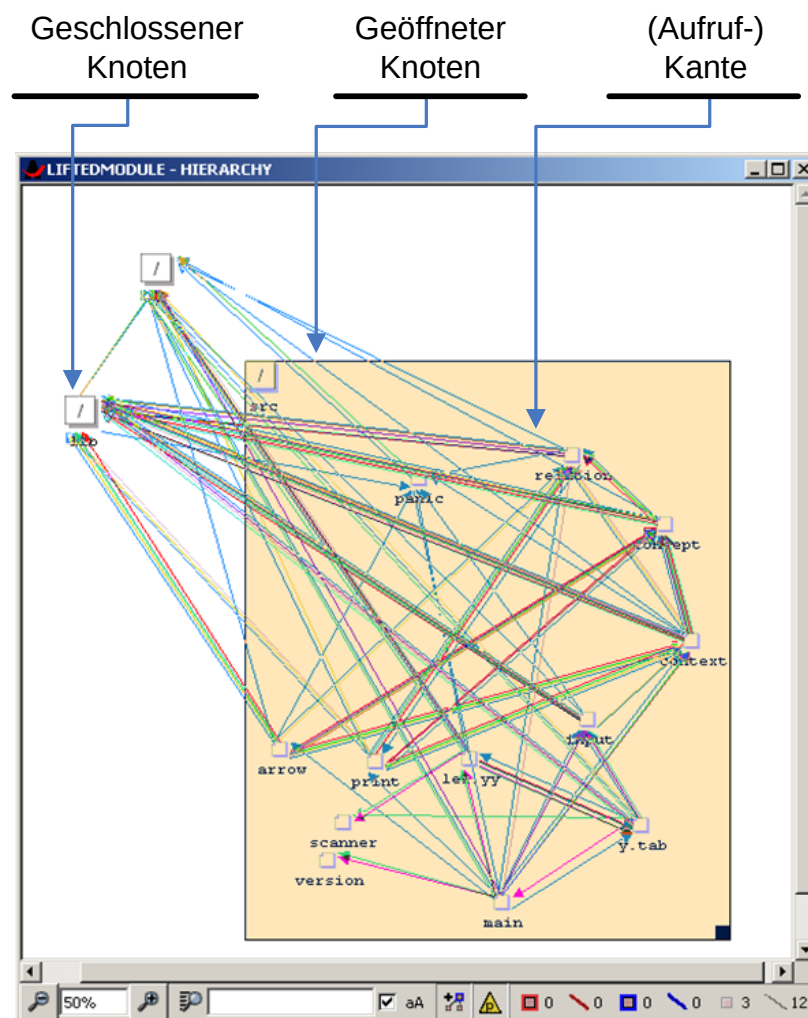


Abbildung 3.3: *GraVis* - Graphfenster

Zu einer *View* lassen sich auch in einem Metrikfenster die Werte von Integer- und Float-attributen der Knoten der *View* anzeigen. Mit diesem Hilfsmittel lässt sich gut nach Programmelementen suchen, die auffällige Attributwerte haben oder die Korrelation von Attributwerten überprüfen.

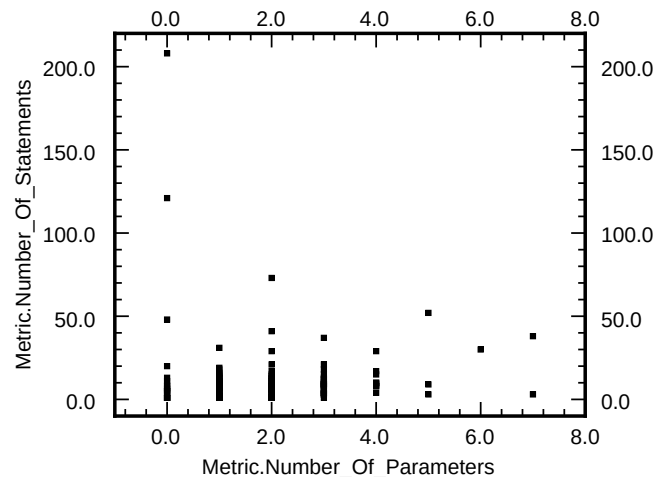


Abbildung 3.4: *GraVis* - Metrikansicht

Im Beispiel in Abbildung 3.4 sieht man alle Funktionen aus einem Aufrufgraphen als Knoten in der Metrikbox. Auf der Abszisse wurden die Anzahl der Parameter der Funktion abgetragen und auf der Ordinate die Anzahl der Anweisungen in der Funktion. Nun kann man versuchen, in diesem Ergebnis interessante Knoten zu identifizieren. Diese Knoten kann man dann in der Metrikansicht auswählen wobei sie automatisch in der zugehörigen *View* selektiert werden, in der man dann die Knoten im Kontext ihrer Umgebung betrachten kann.

Neben der Programmstruktur lassen sich auch die Ergebnisse der Klonerkennung mit *GraVis* anzeigen. Eine Möglichkeit hierfür ist es eine *cpf*-Datei zu öffnen. Nachdem die Datei geladen wurde öffnet sich ein neues Fenster, in dem alle Klonpaare, wie in Abbildung 3.5 gezeigt, aufgelistet sind.

In den ersten beiden Spalten ist das erste Klonfragment mit seinem Dateinamen und der Anfangs- und Endzeile aufgelistet. Die dritte und vierte Spalte enthält die selben Informationen, allerdings für das zweite Klonfragment. Die fünfte Spalte enthält den Typ des Klonfragments und zuletzt kommen zwei Spalten, die seine Länge angeben. Einmal durch die Anzahl der enthaltenen Quellcodezeilen und in der letzte Spalte, wenn diese Information vorhanden ist, die Anzahl der enthaltenen Token. Die Anzeige lässt sich nach einer beliebigen Spalte sortieren. Die Sortierung nach den Dateinamen erleichtert die Suche nach Klone in einer bestimmten Datei, während eine Sortierung nach der Länge schnell zeigt, wo besonders große Klone vorhanden sind.

Klonfragment 1
Klonfragment 2
Typ
Länge

File (1)	Lines (1)	File (2)	Lines (2)	Typ	#Lines	#Tokens
lex.yy.c	[723-734]	lex.yy.c	[1110-1121]	1	12	0
lex.yy.c	[721-734]	lex.yy.c	[1080-1093]	3	14	0
list.c	[106-121]	input.c	[93-105]	3	16	0
main.c	[152-162]	y.tab.c	[1165-1176]	3	12	0
main.c	[149-168]	y.tab.c	[1165-1186]	3	22	0
relation.c	[122-135]	relation.c	[143-154]	2	14	0
set.c	[535-546]	set.c	[633-646]	3	14	0
set.c	[282-307]	set.c	[316-340]	2	26	0
set.c	[545-558]	set.c	[594-607]	1	14	0
set.c	[584-595]	set.c	[633-646]	3	14	0
set.c	[444-453]	set.c	[512-521]	1	10	0
set.c	[441-453]	set.c	[511-521]	3	13	0
set.c	[430-442]	set.c	[498-508]	3	13	0
set.c	[282-307]	set.c	[348-372]	2	26	0
set.c	[316-340]	set.c	[348-372]	2	25	0
set.c	[416-425]	set.c	[467-478]	3	12	0
set.c	[256-269]	set.c	[384-397]	2	14	0
set.c	[416-427]	set.c	[736-746]	3	12	0
set.c	[228-243]	set.c	[256-272]	3	17	0
set.c	[430-440]	set.c	[498-507]	1	11	0
set.c	[535-558]	set.c	[584-607]	2	24	0
set.c	[228-241]	set.c	[384-397]	2	14	0
set.c	[228-241]	set.c	[256-269]	2	14	0
y.tab.c	[1176-1188]	y.tab.c	[1322-1331]	3	13	0

Abbildung 3.5: *GraVis* - Clone Pair Format Ansicht

Neben dem *cpf*-Format können einige Tools die Ergebnisse noch als Graphformate speichern. Diese lassen sich dann in *GraVis* importieren und darstellen, wie in Abbildung 3.6 geschehen.

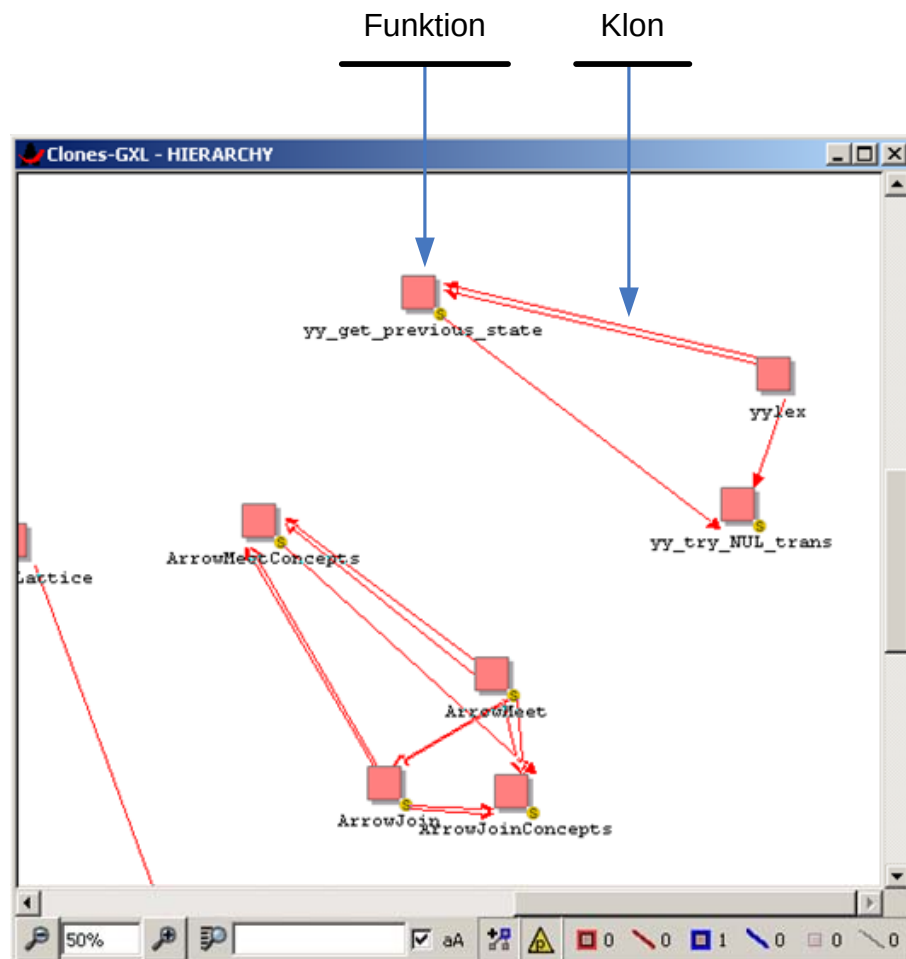


Abbildung 3.6: *GraVis* - Kloninformationen importiert aus einer *gxl*-Datei

Bei dieser Vorgehensweise werden die Klone als Kanten zwischen zwei Knoten dargestellt. Die Knoten repräsentieren hierbei Funktionen, in denen ein Klonfragment enthalten ist. Die Kanten sind auch hier gerichtet, was den Eindruck erwecken könnte, dass der Klon von einer Funktion in die andere geklont wurde. Diese Aussage kann jedoch nicht getroffen werden und der Graph sollte als ungerichteter Graph interpretiert werden.

Klickt man in der *cpf*-Ansicht auf einen Eintrag oder in dem importierten Graphen auf eine Kante, so bekommt man die beiden Klonfragmente nebeneinander in einem Quellcodefenster angezeigt. Die geklonten Quellcodeabschnitte werden hierbei entsprechend ihres Types eingefärbt. In der Standardeinstellung sind Typ 1 Klone rot, Typ 2 Klone gelb und Typ 3 Klone grün. Um die Übersicht zu vereinfachen gibt es für die

3.2. Anwendungsfälle

Klontypen jeweils drei Farbabstufungen. Die jeweils hellste Stufe markiert den gesamten Klonbereich, die mittlere geänderte Bezeichner oder Operatoren und die dunkelste Farbe zeigt hinzugekommene Bereiche an. Ein markierter Typ 2 Klon ist in Abbildung 3.7 zu sehen. So lassen sich bei Typ 2 und Typ 3 Klonen schnell die Unterschiede der Fragmente finden. Die geänderten und hinzugekommenen Markierungen sind für alle Klontypen vorhanden, da Kommentare auch in Typ 1 Klonen geändert werden können.

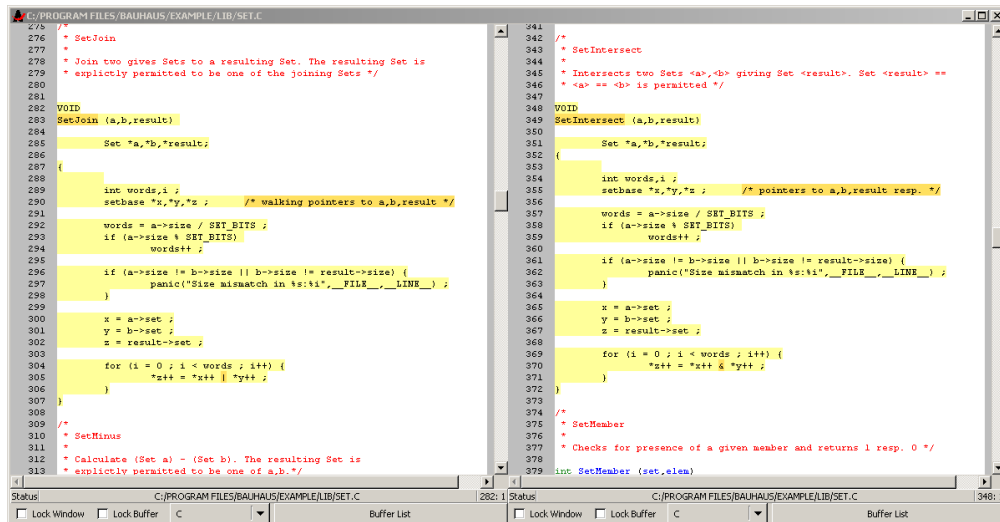


Abbildung 3.7: GraVis - Markierter Typ 2 Klon

3.1.5 Zusätzlich verwendete Programme

Neben den genannten Programmen wird die Bauhaus-Scriptingschnittstelle verwendet, die es erlaubt, durch ein Python-Programm direkt auf alle internen Daten von Bauhaus zuzugreifen, ohne jedoch in den Quelltext von Bauhaus eingreifen zu müssen. Hierdurch kann eine Lösung geschaffen werden, die unabhängig von der verwendeten Bauhaus-Version ist, so lange die Scriptingschnittstelle stabil bleibt.

3.2 Anwendungsfälle

Im Gespräch mit meinem Betreuer haben sich zwei potentielle Anwendungsfälle, in denen das Ergebnis dieser Arbeit verwendet werden kann, herauskristallisiert. Diese sollen nun dargelegt werden, da sie einen großen Einfluss auf die zu erstellende Software haben.

Der erste Anwendungsfall ist im Rahmen der Fehlerbeseitigung innerhalb der Software zu sehen. Sollte bei Tests, Reviews oder im Feld ein Fehler in der Software gefunden werden, ist es für die Beseitigung von Interesse, ob der Bereich, in dem der Fehler liegt, durch

Klonen entstanden ist oder in einer ähnlichen Form noch einmal im Quelltext auftaucht. Sollte dies der Fall sein, könnte es sein, dass der Fehler auch in dem anderen Kontext auftritt, was durch ein geeignetes Klonmanagement vermieden werden kann und in einem insgesamt stabileren und fehlerunanfälligeren System resultiert. Den Wartungsprogrammierer interessieren in diesem Fall allerdings nur die Klonklassen, von denen ein Fragment in einem fehlerhaften Modul enthalten ist. Dies erfordert eine Sicht auf die Ergebnisse der Klonerkennung, die es dem Wartungsprogrammierer ermöglicht, auf eine vertraute Art und Weise in dem System zu navigieren um so schnell die für ihn interessanten Bereiche zu finden. Allerdings werden bei diesem Vorgehen Klone meist gar nicht oder nur sehr lokal entfernt, da es für die einzelnen Teile des Systems unterschiedliche Zuständige gibt und dieses Vorgehen entsprechend koordiniert werden muss.

Der zweite Anwendungsfall findet sich innerhalb des Weiterentwicklungsprozesses der Software wieder. Die Software wird Schritt für Schritt weiterentwickelt, um so an neue externe und interne Anforderungen angepasst zu werden. Innerhalb eines Weiterentwicklungsschritts gibt es verschiedene Phasen, in denen die Änderung geplant und anschließend umgesetzt wird. Ziel ist es innerhalb einer dieser Phasen, in der eine allgemeine Wartung der Softwarearchitektur vorgenommen wird, systematisch Klone zu finden und zu entfernen, um so die Qualität des Gesamtsystems zu verbessern. Hierbei werden auch Klone entfernt, die über Modul oder Subsystemgrenzen hinausreichen. Neben einer globalen Sicht auf das System ist es für diesen Anwendungsfall von Interesse zu sehen, zwischen welchen Komponenten der Software besonders viel geklont wurde, um hier gezielt mit geeigneten Maßnahmen einschreiten zu können.

Für beide Anwendungsfälle ist es wichtig eine Möglichkeit zu haben, die dem Nutzer automatisiert anzeigt, ob ein Klon entfernt werden kann, da gerade in der Programmiersprache C die Möglichkeiten Klone zu entfernen beschränkt sind. Neben der Entfernbarkeit ist auch wichtig, welche Auswirkungen ein anschließendes Refactoring auf die nichtfunktionalen Anforderungen hat.

3.3 Probleme

Nachdem die verschiedenen Programme, die im Rahmen von Bauhaus existieren, vorgestellt wurden, sollen nun die Schwachstellen dieser Tools, die eine effiziente Lösung dieser Aufgabe erschweren, aufgezeigt werden.

3.3.1 Klonerkennung

Für die Klonerkennung existieren wie in Abschnitt 3.1.3 beschrieben vier unterschiedliche Tools, die verschiedene Verfahren implementieren und kombinieren. Jedes Verfahren hat seine Vor- aber auch Nachteile, die eine Entscheidung für ein Verfahren oder Tool erschweren. Zu diesem Schluss kommt auch die Untersuchung in [Burd und Bailey \(2002\)](#), die die Ergebnisse verschiedener Klonerkennungstools analysiert. Das Ziel in diesem

Kontext war, dass die Ergebnisse dazu verwendet werden sollten, die gemeldeten Klone während Wartungsarbeiten zu beseitigen.

Für ein rein tokenbasiertes Verfahren, wie es *clones* implementiert, spricht die Tatsache, dass in dem System viele Präprozessordirektiven enthalten sind, die vereinzelt sogar große Teile des Quellcodes ausblenden. Ein Nachteil ist allerdings, dass viele gemeldete Klone nicht syntaktisch abgeschlossen sind, das heißt Anfang und Ende liegen in verschiedenen Schachtelungsebenen oder gar in verschiedenen Funktionen. Solche Klone lassen sich nicht durch Refactorings entfernen, da hierfür der Abschnitt syntaktisch abgeschlossen sein muss. Zudem soll die *IML* weiter analysiert werden, was bedeutet, dass die Ergebnisse im Anschluss auf die *IML* abgebildet werden müssen. Neben einem hohen zeitlichem Aufwand spricht dagegen, dass viele Klone hier nicht korrekt abgebildet werden können, da sie zum Beispiel teilweise oder ganz in einem Bereich liegen, der durch den Präprozessor entfernt wurde. Die Option *-scope*, die die syntaktische Abgeschlossenheit garantieren soll, hat bei einer Vielzahl der Dateien Probleme, so dass die Ergebnisse nicht ausreichend genau waren.

Für die Klonerkennung mittels *ccdimpl* spricht, dass hier die syntaktische Abgeschlossenheit garantiert ist und so die Durchführung von Refactorings vereinfacht wird. Außerdem lassen sich die Ergebnisse direkt in der *IML* annotieren, was einen Datenverlust durch nicht ausreichend genaue Quellcodepositionen in der *IML* verhindert. Gegen dieses Verfahren spricht die Tatsache, dass viele Quelltextabschnitte nicht berücksichtigt werden können, da sie durch den Präprozessor entfernt werden und der Klonerkennung nicht zur Verfügung stehen.

Mit *cpdetector* wird die Geschwindigkeit des tokenbasierten Verfahrens mit der Möglichkeit der syntaktischen Abgeschlossenheit kombiniert. Dies sind zwei Punkte, die für dieses Tool sprechen. Allerdings werden hierdurch auch keine Präprozessoranweisungen berücksichtigt und es bestehen keine Möglichkeiten, die Ergebnisse in die *IML* zu übertragen. Die gleichen Vor- und Nachteile gelten auch bei *clast*.

	Analysiert Präprozessor	Syntaktische Abgeschlossenheit	Speichert Ergebnisse in <i>IML</i>
<i>clones</i>	✓	✓ ⁵	
<i>ccdimpl</i>		✓	✓
<i>cpdetector</i>		✓	
<i>clast</i>		✓	

Tabelle 3.1: Vergleich der wichtigsten Eigenschaften der Klonerkennungstools

In Tabelle 3.1 sind noch einmal vergleichend die Eigenschaften der verschiedenen Tools zusammengefasst. Gegen *cpdetector* und *clast* spricht, dass diese Tools weder Präprozessoranweisungen noch das Abspeichern der Daten in der *IML* unterstützen und die anderen Tools mehr der gewünschten Funktionalitäten bieten. Für *clones* spricht zwar die

⁵Funktioniert nicht bei allen Dateien problemlos

Geschwindigkeit der Klonerkennung und dass es bei der Analyse die Präprozessoranteile berücksichtigt, allerdings können auch hier die Ergebnisse nicht in der *IML* gespeichert werden. Somit spricht für *ccdiml*, dass es als einziges Tool seine Ergebnisse in der *IML* speichert, auch wenn es keine Bereiche berücksichtigt, die durch Präprozessoranweisungen auskommentiert sind. Trotz diesen Nachteils soll im Rahmen dieser Arbeit *ccdiml* zur Klonerkennung verwendet werden, da sich die Ergebnisse am besten weiterverarbeiten lassen. Lösungsmöglichkeiten für die fehlende Präprozessorunterstützung werden in Kapitel 6 geliefert.

Wenn man sich die Ergebnisse der verschiedenen Klonerkennungstools anschaut, so sind überall Klone vorhanden, die zum einen nicht geeignet sind um sie durch ein Refactoring zu entfernen und zum anderen zwar Klone im theoretischen Sinn sind, deren Entfernung aber nicht sinnvoll ist. Zu der letzten Gruppe gehören Programmiermuster, die zwar häufiger vorkommen aber in jedem Kontext eine andere Bedeutung haben. Einen solchen Klon zu entfernen würde nicht zur Verständlichkeit des Quellcodes beitragen.

3.3.2 Visualisierung

Die existierenden Visualisierungsmöglichkeiten der Ergebnisse mit Hilfe von *GraVis* sind für die genannten Anwendungsfälle aus verschiedenen Gründen nicht ausreichend. In beiden Darstellungsvarianten werden lediglich Klonpaare visualisiert, was bei einem großen System mit einer großen Anzahl an Klonkandidaten schnell unübersichtlich wird. Bei der listenbasierten Ansicht lassen sich zwar gut Dateien identifizieren, die besonders stark von Klonen betroffen sind, wenn man dort nach Dateinamen sortieren lässt, es ist aber schwer nachzuvollziehen, wie es in einem bestimmten Subsystem beziehungsweise Verzeichnis aussieht. Das gleiche gilt für die graphenbasierte Darstellung, bei der man nicht einmal auf Dateiebene runterschauen kann sondern nur einzelne Funktionen sieht. Mit geeigneten Visualisierungsmöglichkeiten, wie sie in Beckwermert (2006) zusammengefasst sind, lassen sich durchaus interessante Informationen entnehmen. Weitere Möglichkeiten sind in Tairas u. a. (2006) beschrieben. Auch hier werden alle Visualisierungen auf Basis von Klonpaaren vorgenommen, was bei großen Klonklassen unübersichtlich werden kann.

3.3.3 Entfernbarekeit

Ziel der Klonerkennung in diesem Kontext ist es unter anderem die Klone zu entfernen. Wenn man hierfür nach geeigneten Wegen in der Programmiersprache C sucht, so ergeben sich im Vergleich zu den Vorschlägen von Fowler und den in Fanta und Rajlich (1999) genannten relativ wenige Möglichkeiten. Klone vom Typ 1 und 2 können mit Hilfe des Refactorings *Extract_Method* entfernt werden, nur dass hier keine Methode, sondern eine Funktion extrahiert wird. Die zweite Möglichkeit ist es, den Klon mittels eines Präprozessormakros zu extrahieren. Dies ist allerdings nicht zu empfehlen, da hier der Klon zwar auf Quelltextebene eliminiert wird aber im Binärcode nach wie vor vorhanden

ist. Ein solches Vorgehen wird schon in dem bereits erwähnten Tool *CloneDR*TM umgesetzt, um so die Wartbarkeit der Software zu verbessern.

Bei Typ 3 Klonen ist ein Refactoring noch schwieriger, wenn nicht sogar in vielen Fällen fast unmöglich. Bei allen Lösungsstrategien wird versucht, einen Typ 3 Klon auf einen vom Typ 1 oder 2 zu reduzieren, um diesen entfernen zu können. Der trivialste Fall ist der, dass zwischen dem eigentlich geklonten Bereich und dem neu eingefügten Bereich keine Daten- und Kontrollabhängigkeit besteht, so dass der Bereich vor oder hinter den Klon geschoben werden kann und sich so ein reiner Typ 1 oder 2 Klon ergibt. Ist dies nicht möglich, so müssten zunächst die unterschiedlichen Quelltextbereiche der Klonfragmente in neue Funktionen ausgelagert werden, die die gleiche Signatur haben, um diese aus dem Klonkontext heraus aufzurufen. Hierdurch wird der Typ 3 Klon wieder auf einen Typ 2 oder Typ 1 Klon reduziert, der mit dem oben genannten Refactoring entfernt werden kann.

In den meisten Fällen ist dies jedoch leider nicht möglich, somit werde ich mich im Folgenden nur auf Klone vom Typ 1 oder 2 konzentrieren, da diese einfach zu entfernen sind und entfernbare Typ 3 Klone auf diese Fälle reduziert werden können.

3.3.4 Toolkette

Die verwendeten Tools und deren Zusammenarbeit werden in Abbildung 3.8 schematisch dargestellt. Der Quelltext wird mit Hilfe von *cafeCC* in eine *IML* überführt und gelinkt. Auf dieser *IML* führt *ccdimpl* seine Klonerkennung aus und speichert die Ergebnisse ebenfalls in dieser. Aus der *IML* wird auch der *RFG* mit dem Tool *iml2rfg* generiert. Anschließend kann man aus der *IML* mit der Scriptingschnittstelle eine *icc*-Datei erzeugen und diese dann wieder mit dem Scripting in den *RFG* importieren. Auf diesem *RFG* werden nun Analysen durchgeführt, die mit der Scriptingschnittstelle realisiert sind. Der *RFG* lässt sich natürlich zu jedem Zeitpunkt mit *GraVis* darstellen, um einen Überblick zu bekommen.

3.4 Aufgabenstellung

Die Software, die im Rahmen dieser Diplomarbeit erstellt wird, soll einen Nutzer bei den in Abschnitt 3.2 beschriebenen Aufgaben unterstützen. Im Wesentlichen ergeben sich drei Aufgabengebiete, das ist zum einen eine kompakte Datenrepräsentation, eine geeignete und intuitive Darstellung und zuletzt eine geeignete Bewertungsfunktion, die vorhersagt, ob sich eine Klonklasse für ein Refactoring eignet, ohne die nichtfunktionalen Anforderungen zu stark negativ zu beeinflussen.

Das erste Ziel, die Reduzierung der darzustellenden Informationen, soll durch eine Kombination verschiedener Maßnahmen erreicht werden, um so möglichst gute Ergebnisse zu erzielen. Zum einen sind, wie bereits beschrieben, nicht alle gemeldeten Klone interessant,

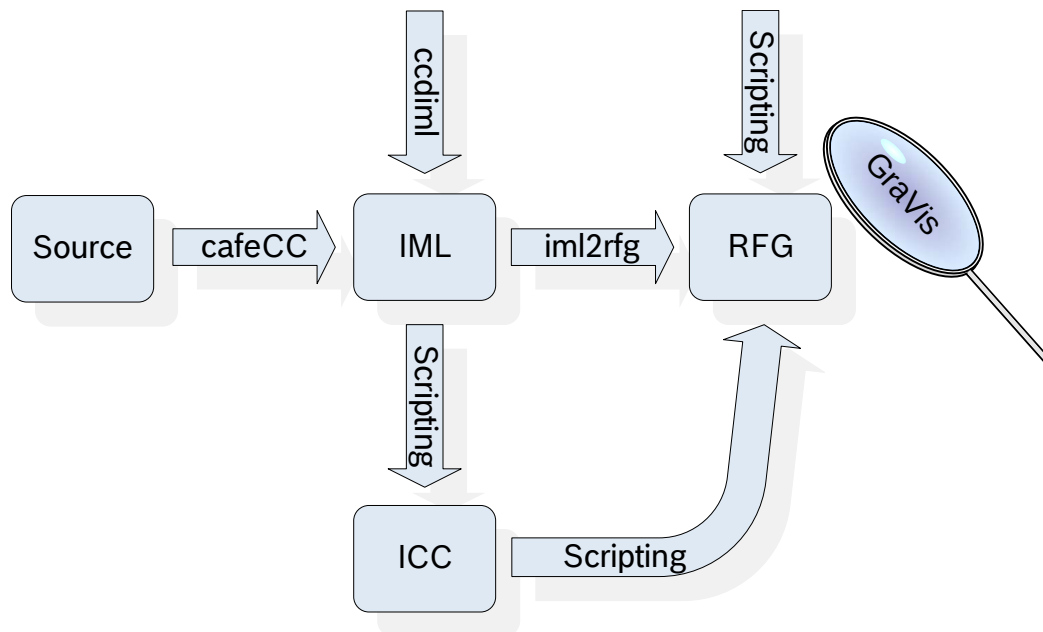


Abbildung 3.8: Verwendete Programme von Bauhaus

so dass die uninteressanten Fälle mit Hilfe von Filtern entfernt werden können. Zum anderen sind Klonfragmente unter Umständen mehrfach vorhanden, so dass hier ohne einen Informationsverlust Daten reduziert werden können.

Das zweite Ziel ist eine effiziente Präsentation der Ergebnisse, die dem Nutzer einen gezielteren Umgang mit den gemeldeten Klonen bietet. Hierfür soll der in Bauhaus vorhandene *RFG* genutzt werden. Dieser hat den Vorteil, dass er auf Grund seines Aufbaus über eine hohe Flexibilität verfügt und in *GraVis* eine große Anzahl an Layoutalgorithmen vorhanden sind und Graphen auf Wunsch auch hierarchisch darstellen kann.

Die letzte Aufgabe ist eine automatisierte Beurteilung der einzelnen Klonklassen bezüglich der Entfernbareit mit Hilfe eines Refactorings. Das Ergebnis soll dem Benutzer als Hinweis dienen, ob eine Klonklasse entfernt werden kann ohne die nichtfunktionalen Anforderungen zu beeinflussen oder wie stark die Beeinflussung ist. Hier kann dann abgewogen werden, ob ein Refactoring für einen bestimmten Fall sinnvoll ist.

4 Umsetzung

*Es ist nicht genug, zu wissen, man muss auch anwenden;
es ist nicht genug, zu wollen, man muss auch tun.*

Johann Wolfgang von Goethe

In diesem Kapitel werden die Lösungsideen und deren Umsetzung beschrieben. In Abschnitt 4.1 wird eine abgewandelte Form der Klonklassendarstellung und im darauf folgenden Abschnitt 4.2 die Visualisierung der Klonklassen mit Hilfe eines *RFGs* erläutert. Abschnitt 4.3 erklärt die Ideen zur Bewertung der Klonklassen. Abschließend beschreibt Abschnitt 4.4 Visualisierungsmöglichkeiten, die bei der Auswertung der Ergebnisse hilfreich sind.

4.1 Repräsentation der Klonklassen

In Abschnitt 2.1.1 wurden die gängigen Arten Klone zu speichern vorgestellt und in Abschnitt 3.1.4 die Visualisierungsmöglichkeiten erläutert, die im Rahmen von Bauhaus existieren. Hierbei kann festgestellt werden, dass die Speicherung und Darstellung der Ergebnisse meistens als Klonpaare erfolgt. Dies hat allerdings bei großen Systemen mit einer großen Anzahl an Klonkandidaten den Nachteil, dass die Datenmenge steigt und zum Teil Daten mehrfach vorhanden sind, was das Verständnis erschwert und die Übersichtlichkeit beeinträchtigt.

Ziel ist es, die Ergebnisse der Klonerkennung möglichst übersichtlich und kompakt grafisch darzustellen. Hierbei ist eine Darstellung als Klonklassen interessant, da sich hierdurch mehrere Vorteile gegenüber einer Darstellung als Klonpaare ergibt, wenn man die Klonpaare nicht auf Funktionsebene, wie beim *gxl*-Import, sondern auf Basis der Klonfragmente visualisiert. Bei einer Klonklasse mit mehr als zwei Klonfragmenten würde eine Darstellung mittels Klonpaaren zu einer hohen Kantenzahl führen, da die Codefragmente vollständig vernetzt wären und sich für eine Klonklasse mit n Klonfragmenten insgesamt $\frac{n \cdot (n-1)}{2}$ Klonkanten ergeben, wie in Abbildung 4.1 auf der linken Seite dargestellt.

Diese hohe Anzahl an Kanten führt schnell zu einer unübersichtlichen Darstellung, die die ohnehin schon hohe Komplexität noch weiter steigert. Stellt man hingegen eine Klonklasse als eigenständiges Objekt dar und verbindet diese mit den Klonfragmenten, so ergeben sich bei n Fragmenten auch nur n Kanten, wie in Abbildung 4.1 auf der rechten Seite

4.1. Repräsentation der Klonklassen

zu sehen. Zudem wird bei dieser Visualisierung die Klonklasse explizit dargestellt im Gegensatz zu der vorherigen Darstellung, wo der Nutzer diese aus den dargestellten Klonkanten ableiten muss.

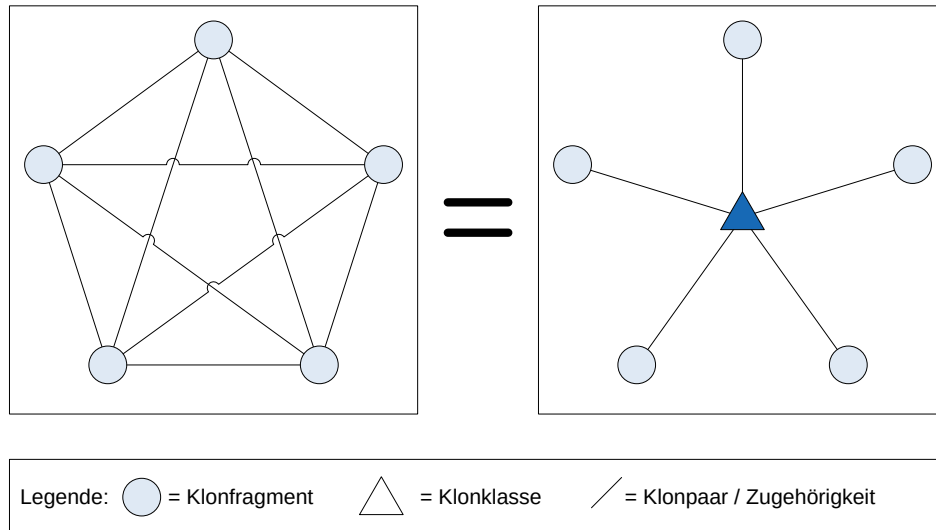


Abbildung 4.1: Darstellung einer Klonklasse mit fünf Fragmenten. Links als Klonpaare, rechts als Klonklasse.

In dieser Darstellungsform können allerdings noch immer zu viele Kanten auftreten. Dies tritt immer dann auf, wenn ein Klonfragment sich in mehreren Klonklassen befindet. Betrachtet man dies theoretisch, so kommt man zu noch mehr Einsparpotenzial. Ein Klonfragment ist eindeutig über seine Daten identifizierbar und sollte in dem Zielsystem nur einmal abgebildet sein. Es gibt also unter den Klonfragmenten keine zwei Fragmente, die den selben Quelltextbereich beschreiben. Existieren nun zwei Klonklassen, in denen das selbe Fragment enthalten ist, so folgt daraus, dass diese Klonklassen in einer Beziehung zueinander stehen. Diese kann aus den Klontypen der beiden Klassen abgeleitet werden. Handelt es sich bei beiden um Typ 1 oder Typ 2 Klassen, so können sie auf Grund der Transitivität zu einer Klasse vereinigt werden. Ist eine der Klassen vom Typ 1 und die andere vom Typ 2, so kann man daraus eine Klonklassen-Hierarchie ableiten.

Abbildung 4.2 zeigt eine Klonklasse vom Typ 2 (in der Abbildung links) und eine Klonklasse vom Typ 1 (in der Abbildung rechts). Die Klonklasse vom Typ 2 muss immer größer als die Klonklasse vom Typ 1 sein, da sie auf jeden Fall alle Elemente der Typ 1 Klasse und zusätzlich Fragmente, die zu einer Typ 2 Klasse führen enthält. Im linken Graphen ist dargestellt, wie das Ergebnis aussehen würde, wenn man die Klonklassen aus der Klonerkennung importiert und rechts ist die vereinfachte hierarchische Ansicht aufgezeichnet.

Insgesamt ergibt sich die in Abbildung 4.3 dargestellte Struktur. Eine Klonklasse vom Typ 2 (C_1^2 in der Abbildung) kann außer einzelnen Klonfragmenten (F_7 und F_8) auch

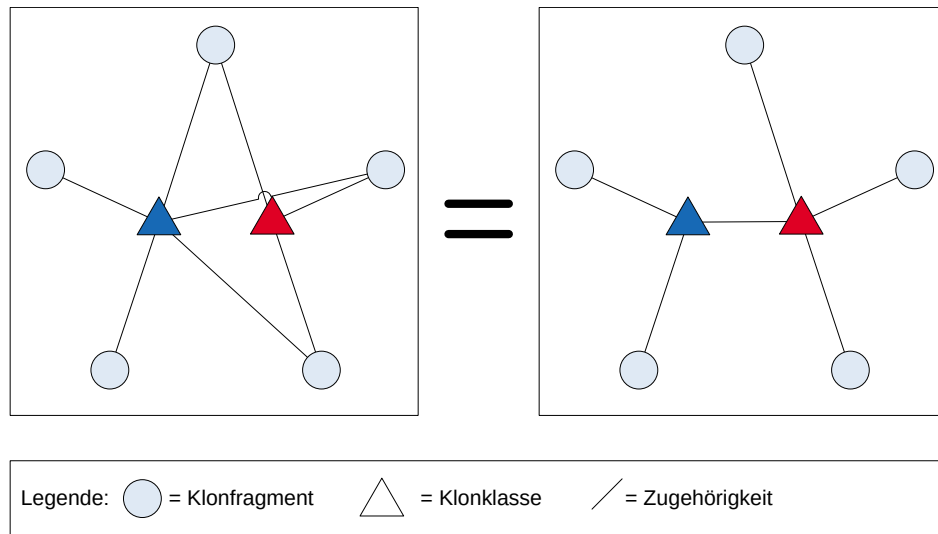


Abbildung 4.2: Hierarchischer Klonklassenaufbau

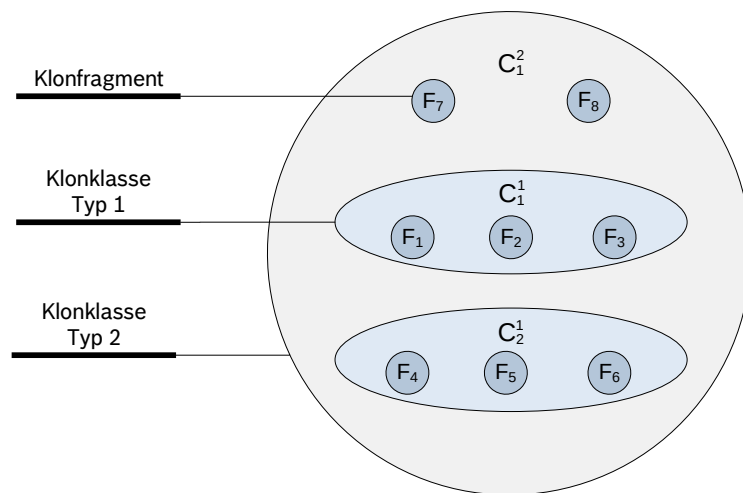


Abbildung 4.3: Klonklassenstruktur

Klonklassen vom Typ 1 (C_1^1 und C_2^1) enthalten, die wiederum aus einzelnen Fragmenten bestehen. Eine Typ 1 Klasse muss immer zwei Fragmente beinhalten, während eine des Typs 2 nur voraussetzt, dass zwei Elemente enthalten sind. Tiefere Verschachtelungen als die hier aufgeführten kann es auf Grund des Aufbaus der Klonfragmente und Klonklassen nicht geben. Durch diese Darstellung sind zwar nicht mehr alle Fragmente einer Typ 2 Klonklasse mit der Klasse verbunden, allerdings lassen sie sich schnell finden, indem man den Kanten zu den benachbarten Typ 1 Klassen folgt und dann deren Fragmente hinzunimmt.

Da das Programm *ccdimpl* verwendet wird, welches keine Klonklassen sondern nur Klonpaare melden kann, müssen die Klonklassen aus den Paaren erzeugt werden. Dies geschieht mit dem Skript *link_clonefile.py*, welches zwei Parameter erwartet. Der erste Parameter ist eine *IML*, die bereits Kloninformationen enthält und der zweite Parameter ist die Ausgabedatei. Das Format für diese Datei ist an die von *cpf* und *ccf* ¹ angelehnt und in Anhang A spezifiziert. Um zu validieren, dass keine Klonpaare bei diesem Schritt verloren gehen, wurde das Skript *check_icc.py* geschrieben, das aus den Klonklassen einer *icc*-Datei wieder Klonpaare generiert und testet, ob diese auch alle in der *IML* enthalten sind. Dieses war bei allen getesteten Dateien der Fall.

4.2 Darstellung als RFG

Die im letzten Abschnitt entstandene Datenstruktur muss nun noch grafisch dargestellt werden. Hierfür soll der von Bauhaus unterstützte *RFG* verwendet werden, um die in Bauhaus existierenden Analysen und Daten wiederverwenden und die Visualisierung mit Hilfe von *GraVis* vornehmen zu können.

Hierzu werden die Klonklassen und die Klonfragmente als unterschiedliche Knotentypen importiert und die Beziehung zwischen ihnen als Kante, wie im vorherigen Abschnitt beschrieben, dargestellt. Da die Kanten hier gerichtet sind, ergibt sich dadurch auch gleich die Richtung der Zugehörigkeit, was das Verständnis vereinfacht. Ein einfacher Import, also alle Objekte in die oberste Ebene der *View* einzufügen, ist bei großen Datenmengen nicht besonders hilfreich und übersichtlich. Vielmehr soll die Dateistruktur, die sich aus dem aus der *IML* erzeugten *RFG* ergibt, verwendet werden, um die Klonfragmente in das zugehörige Modul einzuordnen. Hierdurch entsteht eine hierarchische Ordnung, die dem Nutzer schon aus dem Projekt bekannt ist, da die Aufteilung des Systems in die verschiedenen Subsysteme über Verzeichnisse abgebildet wird. Somit kann man bei der Navigation in der *View* sein Wissen über den Aufbau der Software dazu verwenden, bestimmte Module wiederzufinden und dort gezielt nach Klonen zu suchen, was eine der Anforderungen an das System war. Zudem erleichtert dies dem Nutzer den Umgang mit der Software, da er die Struktur schon kennt.

¹<http://www.bauhaus-stuttgart.de/clones>

Klonklassen noch nach den Subsystemen benannt, die sie betreffen. Eine Klonklasse, die Klonfragmente in den Subsystemen A , B und C hat, erhält den Namen $A\#B\#C$ und ist damit eindeutig identifizierbar.

Diese Ideen lassen sich in der von mir gewählten Visualisierung zum Teil wiederfinden. Die Kohäsion lässt sich bestimmen, wenn man in ein Modul oder Subsystem hinein navigiert und sich die Klonklassenknoten anschaut, die innerhalb dieses Teilbaums liegen. Die Kopplung sind die Klonklassen, die mit ihren Kanten in die einzelnen Subsysteme hineinzeigen. So erhält man auch hier zunächst einen groben visuellen Überblick, welche Teile des Systems besonders stark betroffen sind und kann dann in das System hineinschauen.

In Abbildung 4.4 ist ein Beispiel der Visualisierung gegeben. Hier sind insgesamt drei Subsysteme gegeben. Im Subsystem A gibt es eine Klonklasse vom Typ 1 mit zwei Klonfragmenten. Des Weiteren existiert hier ein Klonfragment, das zu einer Klonklasse vom Typ 2 auf globaler Ebene gehört. Zu dieser Klasse gehört noch ein Klonfragment in Subsystem B und eine Typ 1 Klonklasse mit zwei weiteren Klonfragmenten in Subsystem C. Zu Beginn sind die Subsysteme kollabiert und sie werden als Knoten dargestellt, so dass man sehen kann, dass eine Typ 2 Klonklasse existiert, die die beiden Subsysteme A und B betrifft. Bei weiterem Interesse kann man diese öffnen und sieht die weiteren Details. Durch diese Darstellungsweise erhält man zu Beginn einen Systemüberblick und kann dann je nach Interesse und Zielsetzung innerhalb des Systems navigieren. Die resultierende Ansicht ist in Abbildung 4.5 dargestellt. Hierbei sind die Icons für die Verzeichnisse und Module aus dem Bauhaus-Standard genommen. Die Klonfragmente werden als gelbe Kreise dargestellt und die Klonklassen als farbige Rauten. In der Abbildung ist eine Klonklasse und die ausgehenden Kanten selektiert (blau eingefärbt) und man erkennt, wo sich die zugehörigen Klonfragmente befinden. Diese Vorgehensweise entspricht der, aus [Shneiderman \(1996\)](#) oft zitierten Vorgehensweise: *Overview first, zoom and filter, then details on demand*.

Um diese Ansicht erzeugen zu können, müssen die Daten der Klonerkennung in den *RFG* importiert werden. Hierfür steht das Skript *import_view.py* zur Verfügung. Zum Ausführen dieses Skriptes muss ein *RFG* geladen sein, in dem eine neue *View* mit dem Namen *Clones* erzeugt wird. Der Name der *View* kann in der Datei *config.py* geändert oder eine Abfrage eingebaut werden.

4.3 Bewertung der Klonklassen

Ein weiteres Ziel, das in Abschnitt 3.4 definiert wurde, ist die Beurteilung der Klonklassen nach ihrer Entfernbarekeit. Für diese Aufgabe werden zunächst für die einzelnen Klonfragmente und Klonklassen verschiedene Metriken berechnet, um mit ihnen unterschiedliche Informationen über den betroffenen Quelltext zu erfassen. Aus diesen Informationen wird dann in einem Auswertungsschritt eine Möglichkeit geschaffen, mit der man beurteilen kann, wo sich ein Refactoring nicht oder nur minimal negativ auf die nichtfunktionalen

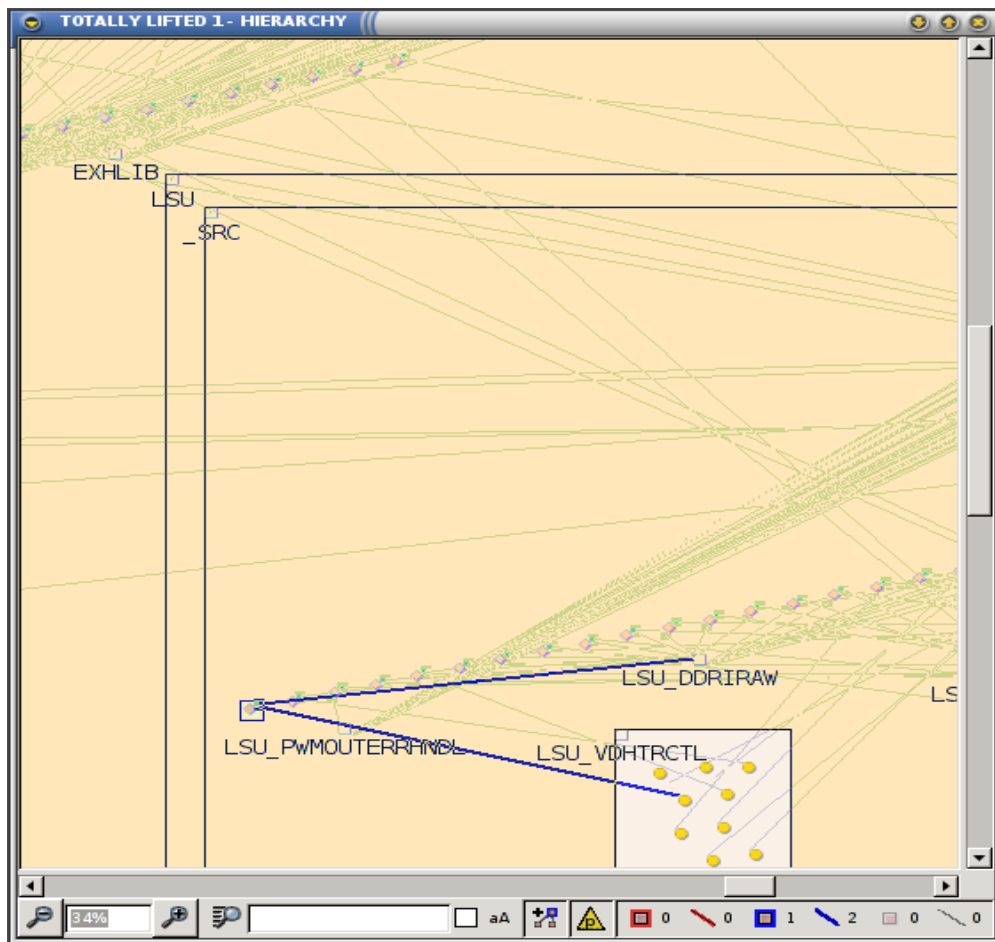


Abbildung 4.5: *RFG* mit Kloninformationen

Anforderungen auswirkt, um so die Aufmerksamkeit des Nutzers auf besonders interessante Klone zu lenken. Im Folgenden werden die Informationen beschrieben, die erhoben wurden. Neben diesen interessanten Klonklassen werden hier auch besonders uninteressante bestimmt, die als Folge dieser Tatsache aus der Ergebnismenge eliminiert werden, um so die Übersicht über das System zu vereinfachen. Diese Filter werden im Anschluss an die Metriken beschrieben. Um dieses Skript aus *GraVis* ausführen zu können, muss erst ein *RFG* und dessen dazugehörige *IML* geöffnet werden, damit alle für das Skript benötigte Informationen vorhanden sind.

4.3.1 Verwendete Metriken

Die verwendeten Standardmetriken wurden bereits in Abschnitt 2.3.2 beschrieben. Sie werden hier noch einmal aufgeführt, um ihre Verwendung zu begründen. Für die bisher nicht erläuterten Metriken werden die noch fehlenden Definition geliefert.

Lines of Code

Die Metrik *LOC* berechnet die Zeilen, die durch ein Klonfragment betroffen sind. Diese Information wird für die Metrik *Number of Lines* benötigt.

Lines of External Copied Code

Die Metriken *LIC*, *LEC* und *LCC* wurden bereits erläutert und implementiert, um die Suche nach Modulen und Verzeichnissen zu erleichtern, die besonders stark von Klonen betroffen sind. Hierfür werden die Werte für die einzelnen Module sowie rekursiv für die einzelnen Verzeichnisse berechnet. Eine effiziente Art der Visualisierung der Ergebnisse dieser Metriken mit Hilfe einer Treemap wird in Beckwermert (2006) beschrieben.

Zudem kann man diese Metriken als Hinweis auf die Klon-kohäsion und -kopplung nehmen. Die Kohäsion entspricht der *LIC* und die Kopplung dem *LEC*. Allerdings sollten diese Werte immer im Verhältnis der *LOC* des Moduls oder Subsysteme betrachtet werden, um so einer Überbewertung der Ergebnisse vorzubeugen.

McCabe Complexity

Die McCabe Komplexität soll für zwei verschiedene Aspekte verwendet werden. Zum einen um Klonklassen aufzufinden, die nach dieser Metrik besonders komplex und damit schwer zu verstehen sind. Die Verständlichkeit des Systems soll durch eine Extraktion der Abschnitte verbessert werden. Zum anderen soll abgeschätzt werden, wie viel Assembler-code aus einem Klonfragment entsteht. Aus Verzweigungen sowie Schleifen entstehen im

Allgemeinen mehr Assembleranweisungen als zum Beispiel aus einer Zuweisung. Dies liegt an der auszuwertenden Bedingung und den Sprüngen, die durchgeführt werden müssen.

Diese Metrik wird für alle Klonfragmente berechnet und anschließend auf der Ebene der Klonklassen der Mittelwert für alle zugehörigen Fragmente berechnet. Hierbei ist anzumerken, dass theoretisch alle Fragmente einer Klonklasse vom Typ 1 oder Typ 2 die selbe McCabe Komplexität besitzen müssen, da sie auf Grund der Tatsache, dass sie Klone sind, strukturell gleich sein und somit die gleichen Anweisungen beinhalten müssen.

Number of Clone Fragments

Bei dieser Metrik handelt es sich um eine einfache Größenmetrik, die für eine Klonklasse angibt, wie viele Klonfragmente in ihr enthalten sind. Es handelt sich also um eine Metrik mit einer Absolutskala als Grundlage.

Einige Metrikwerte auf Klonklassenebene fassen Metriken der Klonfragmente lediglich zusammen. In einigen Fällen kann es interessant sein, diese Werte im Verhältnis zu der Anzahl der Klonfragmente zu sehen, um so den Durchschnitt der Metrikwerte abschätzen zu können. Ein Beispiel hierfür ist die Metrik *Number of Lines*.

Nesting

Ähnlich wie die McCabe Komplexität soll Nesting dazu verwendet werden besonders komplexe Klonklassen zu identifizieren. Auch hier wird für die Klonklassen das Mittel der Metrikwerte der enthaltenen Klonklassen berechnet.

Number of Files

Mit der Metrik *Number of Files* wird gemessen, wie viele Dateien von einem Klon betroffen sind. Mit diesem Maß kann man beurteilen, wie weit eine Klonklasse verteilt ist.

Folgt man der Argumentation von [Jiang u. a. \(2006\)](#), so sind Klonklassen mit einem Wert von eins bei dieser Metrik eher uninteressant, da dies sehr lokale Klone sind, die einfach überblickt werden können.

Number of Functions

Number of Functions gibt die Anzahl der von Klonfragmenten betroffenen Funktionen an. In Kombination mit der Metrik *Number of Files* lässt sich hieraus berechnen, wie viele Funktionen im Schnitt pro Datei von einer Klonklasse betroffen sind.

Number of Lines

Während *Lines of Code* die Anzahl der betroffenen Zeilen auf Klonfragmentebene berechnet, werden diese Werte für die Klonklassen aufaddiert und als *Number of Lines* abgespeichert. Diese Metrik gibt die Gesamtgröße der Klonklasse auf Zeilenbasis an. Hiermit lässt sich ein grober Überblick über die Größe der Klonklasse gewinnen.

Number of Nodes

Nach einigen Versuchen mit der *Number of Lines* Metrik stellte sich heraus, dass die gemeldeten Werte zwar im Sinne der Metrik richtig sind aber nicht den Umfang der Anweisungen wiedergeben, die in den betroffenen Bereichen liegen. Die Ursache hierfür ist in der hohen Kommentardichte und in den bedingten Präprozessoranweisungen zu finden.

Um hier genauere Ergebnisse zu erhalten wurde die Metrik *Number of Nodes* eingeführt, die die Anzahl der betroffenen, nicht künstlichen *IML*-Knoten angibt. Hierdurch lässt sich genauer angeben, wie groß eine Klonklasse ist. Vor allem lassen sich diese Metrikwerte im Vergleich zu den Werten von *Lines of Code* zuverlässiger vergleichen.

Number of Parameters

Um beurteilen zu können, ob ein Refactoring für eine bestimmte Klonklasse sinnvoll und vor allem möglich ist, werden die Metriken *Number Of Parameters* und *Parameter Type is Different* berechnet. Die erste gibt an, wie viele Parameter eine Funktion übergeben bekommt, wenn man die Klonklassen durch ein Refactoring extrahiert. Die zweite Metrik sagt aus, wie viele inkompatible Datentypen existieren.

Um diese Werte zu berechnen, werden zunächst zu jeder Klonklasse die lesenden und schreibenden Zugriffe auf Variablen, Konstanten und Felder aus den betroffenen Bereichen herausgesucht. Hierbei werden die Zugriffe nach ihrer Reihenfolge sortiert, so dass einander entsprechende Zugriffe sich zugeordnet werden können. Anschließend werden diese korrespondierenden Zugriffe auf verschiedene Kriterien geprüft, um herauszufinden, ob sie kompatibel sind und ob sie übergeben werden müssen.

Zwei Zugriffe sind kompatibel, wenn der Typ des lesenden oder schreibenden Ausdrucks gleich ist. Dies lässt sich in der *IML* verhältnismäßig einfach prüfen, da hier die Typen an den Ausdrücken annotiert sind und diese einfach verglichen werden können. Sollten die Typen der Knoten unterschiedlich sein, so hat dies in der Programmiersprache C verheerende Folgen, da hierdurch die Quellcodeabschnitte nicht durch eine Repräsentation ersetzt werden können.

Eine Menge von einander zugeordneten Zugriffen muss unter verschiedenen Umständen als Parameter übergeben werden. Wenn es sich um Literale handelt, deren Wert sich

unterscheidet, muss dieses Literal übergeben werden. Wird ein Parameter oder eine lokale Variable verwendet, so müssen diese immer übergeben werden, da die neue Funktion auf diese nicht zugreifen kann und auch an allen aufrufenden Stellen unterschiedlich ist. Globale Variablen müssen nicht in allen Fällen übergeben werden. Sollten alle Zugriffe auf die gleiche globale Variable erfolgen, so kann die Übergabe entfallen.

4.3.2 Verwendete Filter

Die Anzahl der gemeldeten Klonkandidaten ist sehr hoch und bei einer manuellen Inspektion lassen sich einige uninteressante Klonklassen identifizieren, die relativ viele Klonfragmente enthalten aber für ein Refactoring uninteressant sind. Diese werden nun durch entsprechende Filter automatisch aus der Liste der potenziellen Klone entfernt, um so die Datenmenge zu reduzieren und dem Nutzer die uninteressanten Fälle zu ersparen. Dadurch ist ein effektiveres Arbeiten möglich.

Assembler

Eine verhältnismäßig große Klonklasse ist die, die eine Assembleranweisung enthält. Der Grund hierfür liegt in der Syntax, in der in C Assembleranweisungen notiert werden. Die Anweisungen werden als ein zusammenhängender String angegeben, wobei dieser sich über mehrere Zeilen erstreckt. Damit kommen viele Assembleranweisungen über die minimale Klonlänge von zehn Zeilen, lassen sich aber nicht sinnvoll extrahieren, wenn es sich um Klonklassen des Typs 2 handelt. Typ 1 Klone können im Gegensatz jedoch entfernt werden, weshalb sie nicht ausgefiltert werden.

Assignment

Eine weitere, sehr große Klonklasse ist die, die nur oder zum überwiegenden Teil einfache Zuweisungen enthält. Alle Zuweisungssequenzen sind Typ 2 Klone aber ein sinnvolles Entfernen ist in den allerwenigsten Fällen möglich. Hierfür müsste in der Regel Zuweisungsziel und -quelle übergeben werden, was eine Auslagerung in eine Funktion unsinnig erscheinen lässt, da nur noch die Zuweisungen konstant wären. Zudem gibt es keinen logischen Zusammenhang zwischen den einzelnen Klonfragmenten, was eine Entfernung der Klone sinnvoll erscheinen lassen würde.

Blacklist

Um nicht für jede neu auftauchende uninteressante Klonklasse einen Filter schreiben zu müssen, können mit Hilfe des *Blacklist*-Filters eine Reihe von c-Dateien definiert werden, die unerwünschte Programmuster enthalten. Diese Dateien müssen mit übersetzt und gelinkt werden. Der Filter entnimmt dann alle Klonklassen, die ein Fragment in einer der

angegebenen Dateien enthält, der Menge der gemeldeten Klonklassen. Bei umfangreichen Mustern muss aber beachtet werden, dass sie unter Umständen kleinere Klonfragmente enthalten, die zur Entfernung erwünschter Klonklassen führen. Insgesamt ist hier zu beachten, dass möglichst minimale Beispiele und auch nur eines pro Datei angegeben werden, um so nicht durchaus erwünschte Klonklassen zu treffen.

Empty Function

Auf Grund des Präprozessors gibt es eine Reihe von Funktionen, die verhältnismäßig lang sind aber dennoch in der *IML* keine Anweisung enthalten. Im Quelltext sind hier im Allgemeinen Quellcodeteile enthalten, die auf Grund der Präprozessorkonfiguration entfernt wurden. Diese leeren Funktionen werden alle in einer Klonklasse zusammengefasst, sind aber für ein Refactoring uninteressant und werden aus diesem Grund entfernt.

4.4 Metrikboxen

Mit Hilfe der Metrikboxen lassen sich die Zusammenhänge verschiedener Metrikwerte darstellen und man kann mit ihnen Klonklassen mit interessanten Werten einfach finden. Im folgenden werden einige Metrikboxen abgebildet, um die Vorgehensweise aufzuzeigen. Weitere Metrikboxen werden in Kapitel 5 verwendet, um Klonklassen zu finden, die für die Lösung der gestellten Aufgabe wichtig sind.

In Abbildung 4.6 sieht man die Anzahl der in einer Klonklasse enthaltenen *IML*-Knoten in Relation zu der errechneten Anzahl an Parametern. Wie der Grafik zu entnehmen ist gibt es vergleichsweise wenig Klonklassen, die eine größere Menge von *IML*-Knoten enthalten. Die Anzahl der Parameter reicht von Null bis zu 34. Jedoch sind auch hier die Klonklassen mit einer niedrigen Parameteranzahl in der Mehrheit. Ein direkter Zusammenhang lässt sich aus den gemessenen Werten nicht entnehmen.

In der Metrikbox in Abbildung 4.7 wurde die Anzahl der Fragmente einer Klasse gegen die Anzahl an Parameter abgetragen. Hier ist zu erkennen, dass die Klassen, die viele Parameter für die Übergabe benötigen würden, über wenige Fragmente verteilt sind. Klonfragmente, die häufig kopiert wurden, scheinen hauptsächlich globale Variablen zu verwenden und nur wenige aus dem, den Klon enthaltenen, Kontext.

Interessanterweise kann aus Abbildung 4.8 entnommen werden, dass Klonklassen, die viele Fragmente besitzen, verhältnismäßig wenig *IML*-Knoten enthalten. Dies bedeutet, dass die einzelnen Klonfragmente wenige Anweisungen enthalten, wodurch sie wieder uninteressant für ein Refactoring werden.

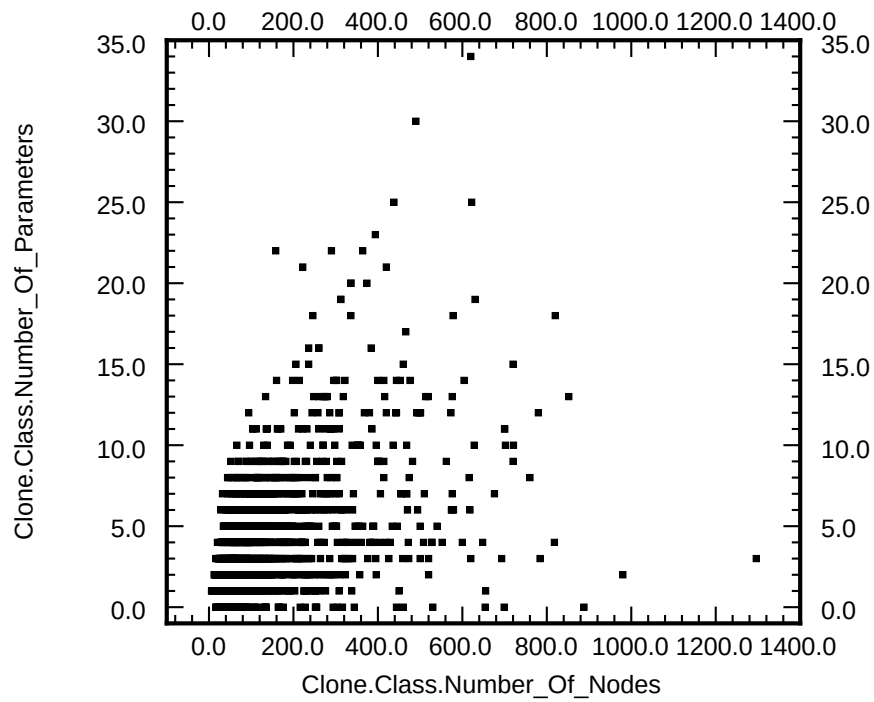


Abbildung 4.6: Metrikbox - Anzahl an *IML*-Knoten und Anzahl an Parametern

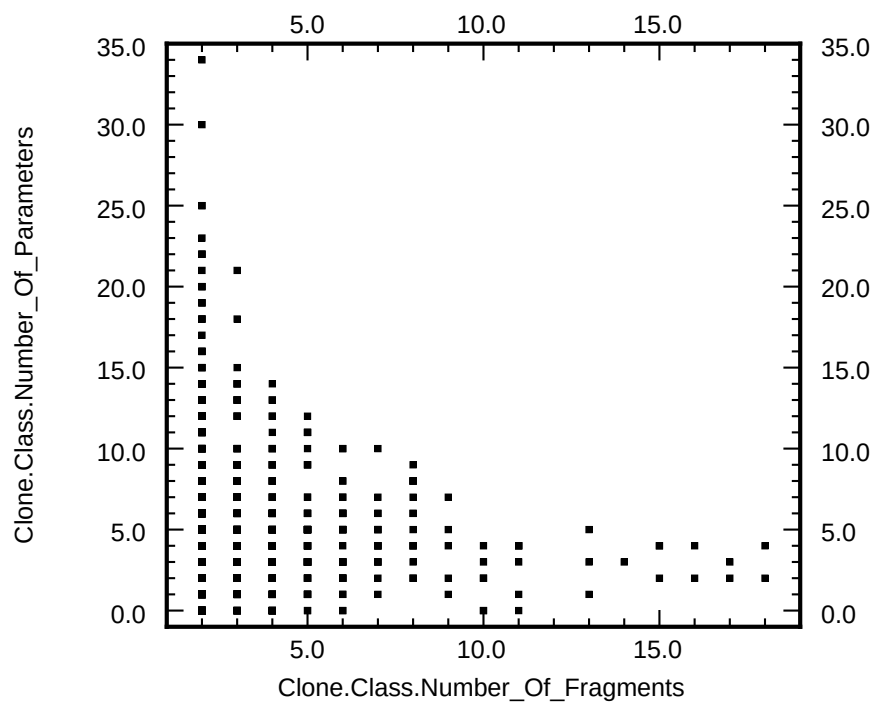


Abbildung 4.7: Metrikbox - Anzahl an Fragmente und Anzahl an Parametern

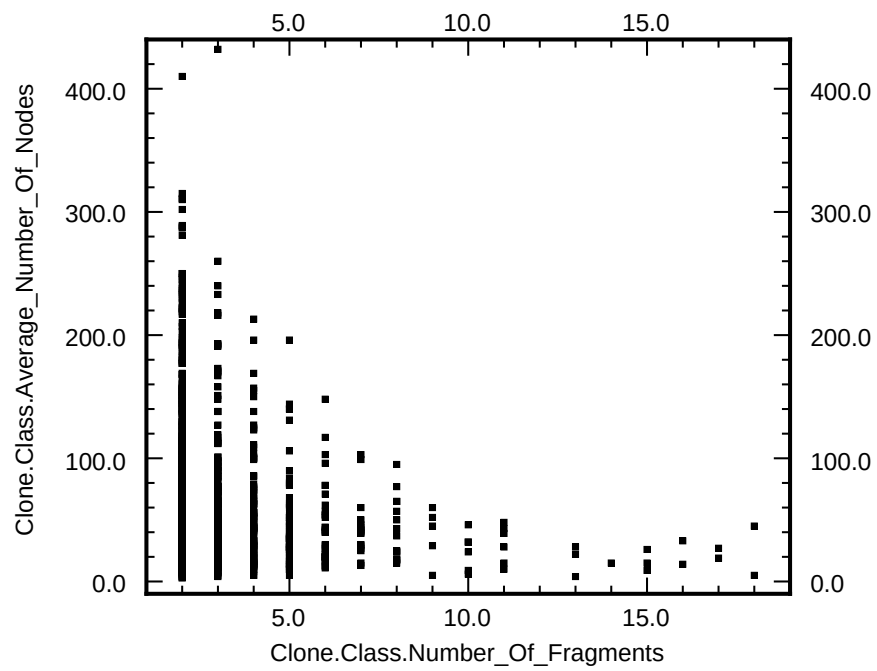


Abbildung 4.8: Metrikbox - Anzahl an Fragment und durchschnittliche Anzahl an *IML*-Knoten

5 Auswertung

Auch aus Steinen, die einem in den Weg gelegt werden, kann man Schönes bauen.

Johann Wolfgang von Goethe

In diesem Kapitel soll beschrieben werden, wie die Wirksamkeit der getroffenen Maßnahmen untersucht wurde und welche Ergebnisse aus ihnen resultieren. Zunächst werden in Abschnitt 5.1 die Ergebnisse der Klonerkennung ohne jegliche Nachbearbeitung vorgestellt, um die Basisdaten meiner Analysen aufzulisten. Anschließend werden in Abschnitt 5.2 die Ergebnisse der im letzten Kapitel vorgestellten Filter untersucht. Im abschließenden Abschnitt 5.3 wird überprüft, was durch Entfernen der einzelnen Klonklassen in Hinblick auf die nichtfunktionalen Anforderungen, der Laufzeit und des Speicherverbrauchs eingespart werden kann.

5.1 Statistik

Der Quelltext, der von mir analysierten Systemvariante besteht insgesamt aus ungefähr 340.000 Anweisungs- und 350.000 Kommentarzeilen. Der verhältnismäßig hohe Anteil an Kommentaren ist zum Teil durch den Misra-Standard bedingt. Die Misra-Regeln sind eine Reihe von Empfehlungen, die eingehalten werden sollen um sicherzustellen, dass der Code korrekt arbeitet. Allerdings bietet er die Möglichkeit, gewisse Regeln weniger streng zu fassen, wenn das Verhalten auf dem verwendeten Compiler sichergestellt werden kann. In solchen Fällen fordert der Standard einen Kommentar, der auf den gewollten Regelverstoß hinweist.

Das Tool *ccdiml* braucht zur Klonerkennung auf dem Gesamtsystem mit den Parametern *-mode fine -type3gap 0* gut 1,5 Stunden und findet 5952 potentielle Klonpaare. Dies ist eine für die Systemgröße verhältnismäßig kleine Zahl. Nachdem die Klonpaare mit dem Skript *link_clonefile.py* zu Klassen zusammengelinkt wurden, ergeben sich 1647 Klonklassen mit insgesamt 4798 Klonfragmenten.

Wie sich die Fragmente vor der Filterung auf die einzelnen Klonklassen verteilen, kann aus Abbildung 5.1 und Tabelle 5.1 entnommen werden. Man kann erkennen, dass für eine Vielzahl der Ergebnisse Klonpaare ausreichen würden, da über 60% der Klonklassen nur aus zwei Fragmenten bestehen.

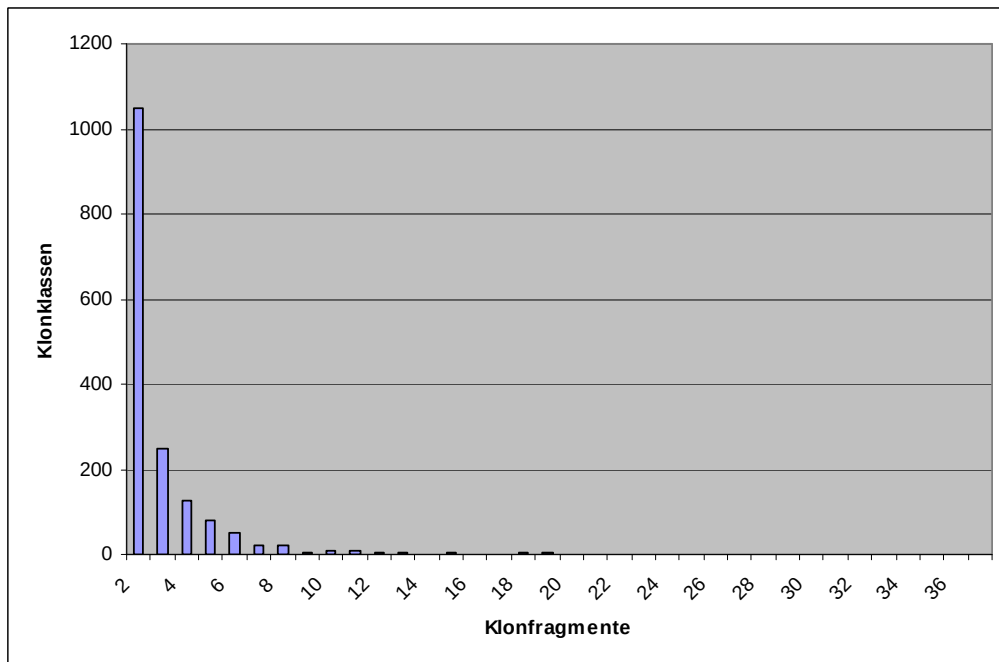


Abbildung 5.1: Häufigkeit der Klonklassen vor der Filterung

Fragmente	2	3	4	5	6	7	8	9	10	11
Anzahl	1050	250	128	78	50	21	19	6	10	7

Fragmente	12	13	14	15	16	17	18	19	20	21
Anzahl	3	4	1	5	2	2	3	3	0	0

Fragmente	22	23	24	25	26	27	28	29	30	31
Anzahl	1	0	2	0	0	0	0	0	0	1

Fragmente	32	33	34	35	36	37
Anzahl	0	0	0	0	0	1

Tabelle 5.1: Verteilung der Klonklassen vor der Filterung

	Assembler	Assignment	Empty Function	Gesamt
Klassen	60	264	1	325
Fragmente	145	1051	13	1209

Tabelle 5.2: Anzahl der gefilterten Klonklassen

5.2 Ergebnisraumminimierung

Die in Abschnitt 4.3.2 aufgeführten Filter sind verschieden wirksam, so dass die Ergebnisse in Tabelle 5.2 aufgeschlüsselt sind. Hier kann man erkennen, dass die Klonklasse der *Assignments* den größten Anteil herausgefiltert hat.

Insgesamt wurden mit Hilfe der Filter 325 Klonklassen mit 1209 Fragmenten entfernt. Dies sind 20% der Klonklassen und 25% der Klonfragmente. Wie in Abbildung 5.2 und Tabelle 5.3 zu sehen ist, sind durch die Filterung Klonklassen aller Größen betroffen, besonders häufig werden jedoch besonders große Klassen entfernt.

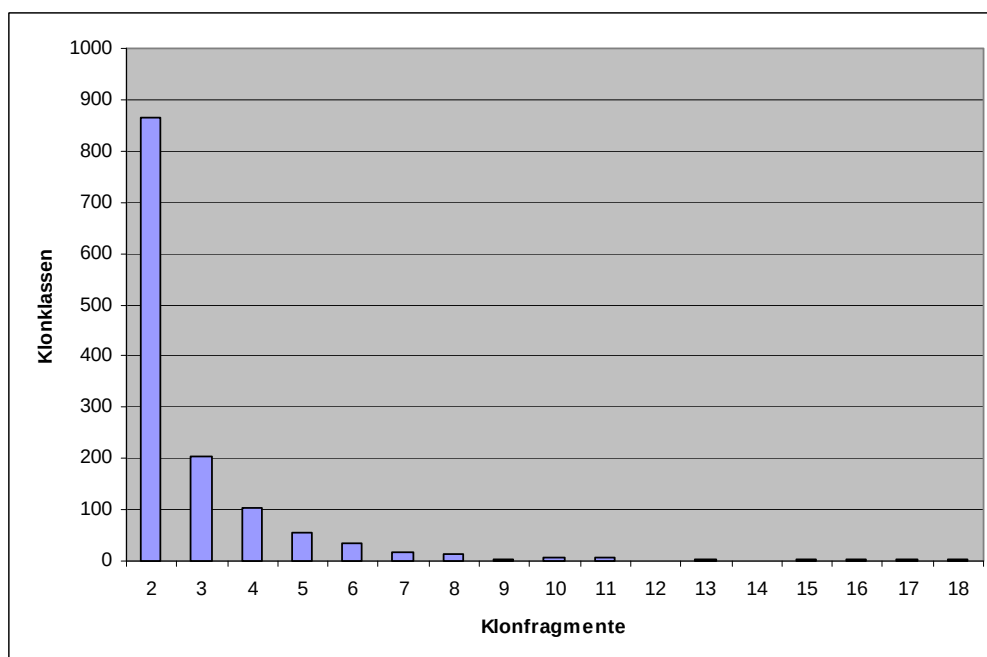


Abbildung 5.2: Häufigkeit der Klonklassen nach der Filterung

Die gefilterten Ergebnisse wurden manuell daraufhin analysiert, ob sie wirklich uninteressant für ein Refactoring sind, um festzustellen, ob hierbei auch interessante Klone entfernt wurden. Insgesamt wurden 110 Klonklassen analysiert, was einem Drittel der entfernten Klonklassen entspricht. Insgesamt wurden Klonklassen aller drei Filter analysiert und es war festzustellen, dass der Filter *Assignment* fünf falsche Treffer hatte, was einer Quote von 5% entspricht.

5.3. Einsparpotential der Klonklassen

Fragmente	2	3	4	5	6	7	8	9	10	11
Anzahl	866	203	104	55	33	16	15	5	6	6

Fragmente	12	13	14	15	16	17	18
Anzahl	0	3	1	3	2	2	2

Tabelle 5.3: Verteilung der Klonklassen nach der Filterung

Diese fünf Klonklassen lassen sich in zwei Kategorien unterteilen. Bei zwei Klassen handelte es sich um Zuweisungssequenzen, die eigentlich hätten refactoriert werden können, da es sich um die Initialisierung von Strukturen mit Standardwerten handelt. Dies ist ähnlich einem Konstruktor in objektorientierten Programmiersprachen und hätte in eine Funktion extrahiert werden können. Die zweite Gruppe der falsch Positiven wurde auf Grund eines Programmierfehlers entfernt, der nach der Auswertung noch behoben wurde. Somit liegt die Quote der falschen Treffer nach der Verbesserung bei 1%.

5.3 Einsparpotential der Klonklassen

Um das Einsparpotential der Klonklassen beurteilen zu können, wurden einige Klonklassen mit Hilfe des Refactorings *Extract Method* entfernt. Davor wurden jedoch verschiedene Informationen über den Objektcode, die Klon-klasse und das -fragment sowie den Quelltext erhoben, um sie dann noch einmal nach dem Refactoring zu messen. Die Differenz der Werte ist somit die Auswirkung, also Ersparnis oder Mehraufwand, des Refactorings.

Zu den Informationen, die erhoben werden sollten, gehört zunächst die Größe des resultierenden Programms und der betroffenen Objekt- und *IML*-Dateien, um so die absolute Größenänderung messen zu können. Darüber hinaus gehören noch die Metrikwerte der zu entfernenden Klonklasse und deren Fragmente dazu. Unter ihnen sollen nach der Analyse diejenigen bestimmt werden, die positive und negative Auswirkung auf die Änderungen haben. Zuletzt wird noch die Anzahl der Assembleranweisungen der betroffenen Funktionen gemessen, was einen konkreten Hinweis auf das Laufzeitverhalten des resultierenden Codes geben kann. Da die Anzahl der Klonklassen nach dem Import und dem automatischen Filter immer noch sehr hoch ist, werden zunächst noch manuell Gruppen von Klonklassen entfernt. Wie sich die Anzahl der Klonklassen und der Klonfragmente durch die verschiedenen Schritte verringert hat, ist in Abbildung 5.3 zu sehen.

In dieser Abbildung gibt es für jeden Schritt ein Säulenpaar, das die Anzahl der noch enthaltenen Klonklassen (dunkelblau) und Klonpaare (hellblau) darstellt. Unter dem Säulenpaar steht der Schritt, der zu dem Säulenpaar geführt hat. Das erste Paar visualisiert die Situation direkt nach dem Import der Klondaten. In einem ersten Schritt werden die Metriken berechnet und die programmierten Filter angewendet (siehe letzter Abschnitt), was zu der zweiten Säule führt. Anschließend wurden manuell alle Klonklassen gelöscht, bei denen die Anzahl der inkompatiblen Parametertypen ungleich Null ist, da

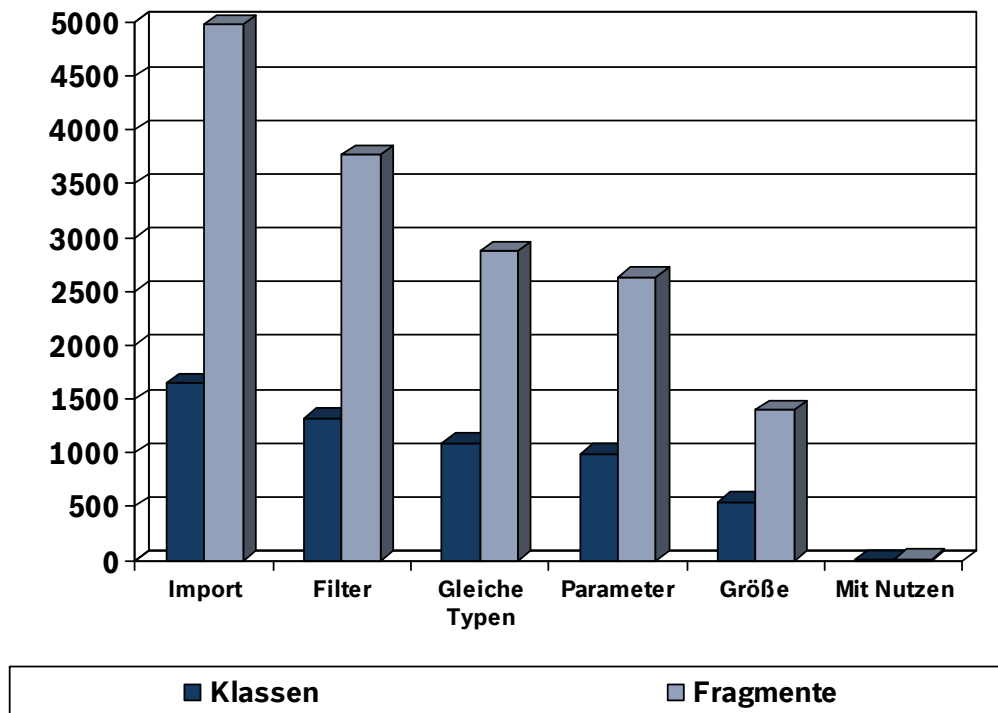


Abbildung 5.3: Auswirkung der einzelnen Filterungen

diese Klonklassen in der Programmiersprache C nicht entfernt werden können und somit für die weitere Betrachtung nicht relevant sind. Die betroffenen Klonklassen sind in Abbildung 5.4 ausgewählt. Die verbleibenden Klassen und Fragmente sind in der dritten Säule von Abbildung 5.3 enthalten. Um die Datenmenge weiter zu reduzieren, wurden nun die Klonklassen mit besonders vielen Parametern entfernt, was zu der Säule vier führt. Die gelöschten Knoten sind in Abbildung 5.5 selektiert. Es ist zu sehen, dass alle Klonklassen ausgewählt wurden, deren *Number of Parameters*-Wert größer als neun ist. Dieser Schwellwert basiert auf empirischen Untersuchungen für den Zielprozessor. Der Compiler generierte den Assemblercode so, dass bis zu acht Funktionsparameter mit Hilfe von Prozessorregistern übergeben werden. Vier dieser Parameter müssen *by Value* und die anderen vier *by Reference* übergeben werden. Wenn die Funktion keinen Rückgabewert besitzt, so kann noch ein zusätzlicher Parameter *by Value* über ein Register übergeben werden, welches sonst für den Rückgabewert reserviert ist. Jeder weitere Parameter wird über den Stack übergeben und erzeugt bei dessen Verwendung mehr Assembleranweisungen, so dass ein negativer Einfluss auf die nichtfunktionalen Anforderungen unwahrscheinlicher wird. In einem letzten Schritt wurden Klonklassen entfernt, die nicht mehr als 40 *IML*-Knoten pro Klonfragment besitzen. Die Klonerkennung arbeitet mit einer minimalen Zeilenanzahl pro Klonfragment, was bei dem hier analysierten Code nicht immer aussagekräftig ist, da unter Umständen viele Kommentare in dem Fragment liegen und so nur sehr wenige Anweisungen vorhanden sind. Der Wert 40 wurde durch

einige Versuche bestimmt und sorgt dafür, dass ungefähr zehn Anweisungen in dem Klonfragment enthalten sind. Die verbleibenden Klonklassen und Klonfragmente sind in dem fünften Säulenpaar enthalten. Insgesamt konnte durch diese Filterung rund ein Drittel der Klonklassen und Fragmente entfernt werden.

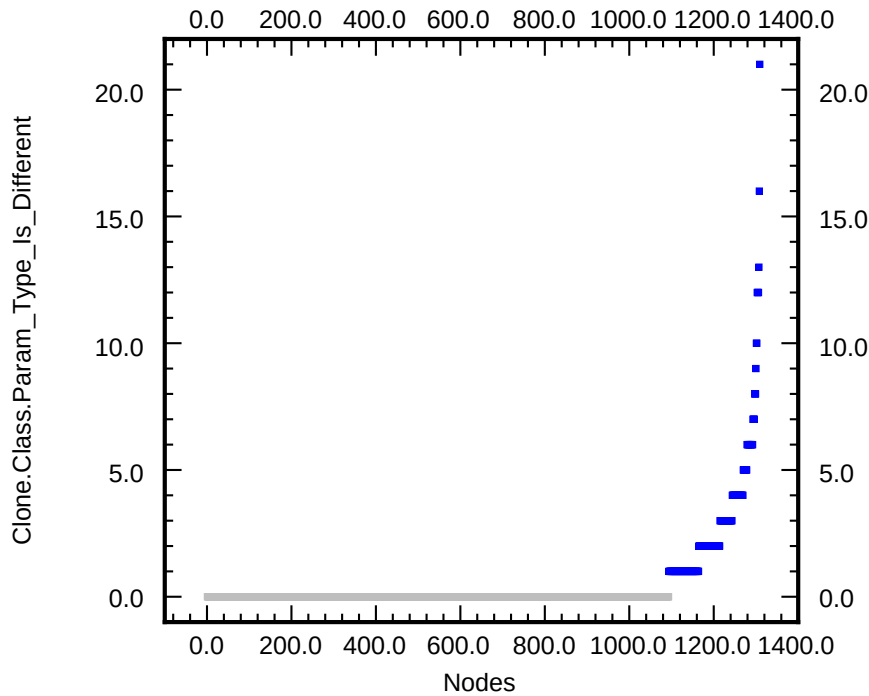


Abbildung 5.4: *Number of different Types* der Klonklassen

Die verbliebenen Klonklassen sind in Abbildung 5.7 abgebildet. Hier wird zum einen die Anzahl der Knoten gegen die Anzahl der Fragmente und zum anderen gegen die Anzahl der Parameter abgetragen. Dies sind aussagekräftige Metrikverhältnisse, da Klonklassen, die eine große Anzahl an *IML*-Knoten besitzen zwar viel Einsparpotenzial ermöglichen, jedoch wird diese dadurch verringert, wenn der Klon über viele Fragmente verteilt ist oder viele Parameter erwartet. Eine hohe Anzahl an Fragmenten ist ineffizient, da mit ihnen in jedem Kontext nur wenig gespart werden kann und an jeder Stelle ein Funktionsaufruf eingefügt werden muss. Dieser Aufruf kostet Assembleranweisungen, Speicher und Zeit. Bei einer großen Anzahl an Parametern ist der Nachteil, dass diese vor dem Aufruf alle in Register kopiert und die alten Register gesichert werden müssen. Je mehr Register betroffen sind, desto länger benötigt der Prozessor zum Retten und Wiederherstellen der Werte.

Bei der ersten Durchsicht der Ergebnisse ist schnell aufgefallen, dass die Metrik *Number of Lines* in Bezug auf die Größe eines Klons nicht sehr aussagekräftig ist. Die Gründe hierfür wurden bereits aufgeführt und daraufhin ist die Metrik *Number of Nodes* eingeführt worden. In Abbildung 5.8 wurde die Mittelwert der *Number of Lines* einer Klonklasse

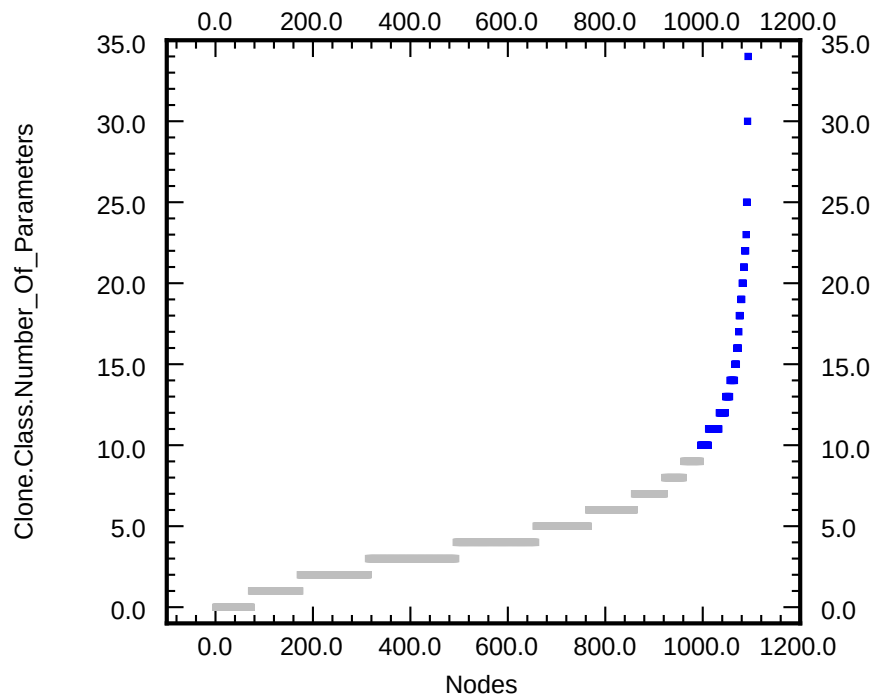


Abbildung 5.5: *Number of Parameters* der Klonklassen

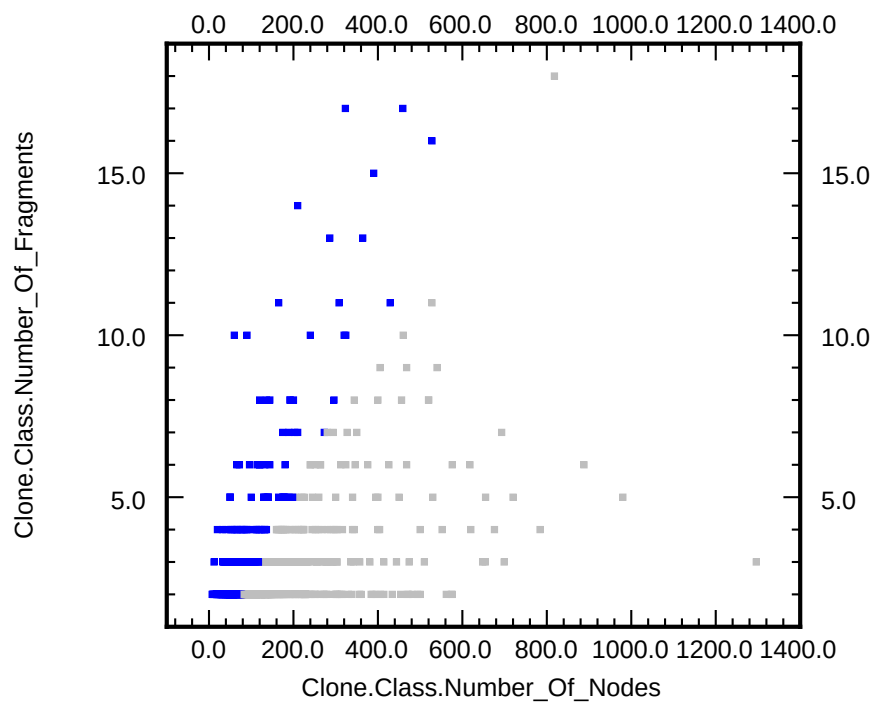


Abbildung 5.6: *Number of Nodes* gegen *Number of Fragments* abgetragen

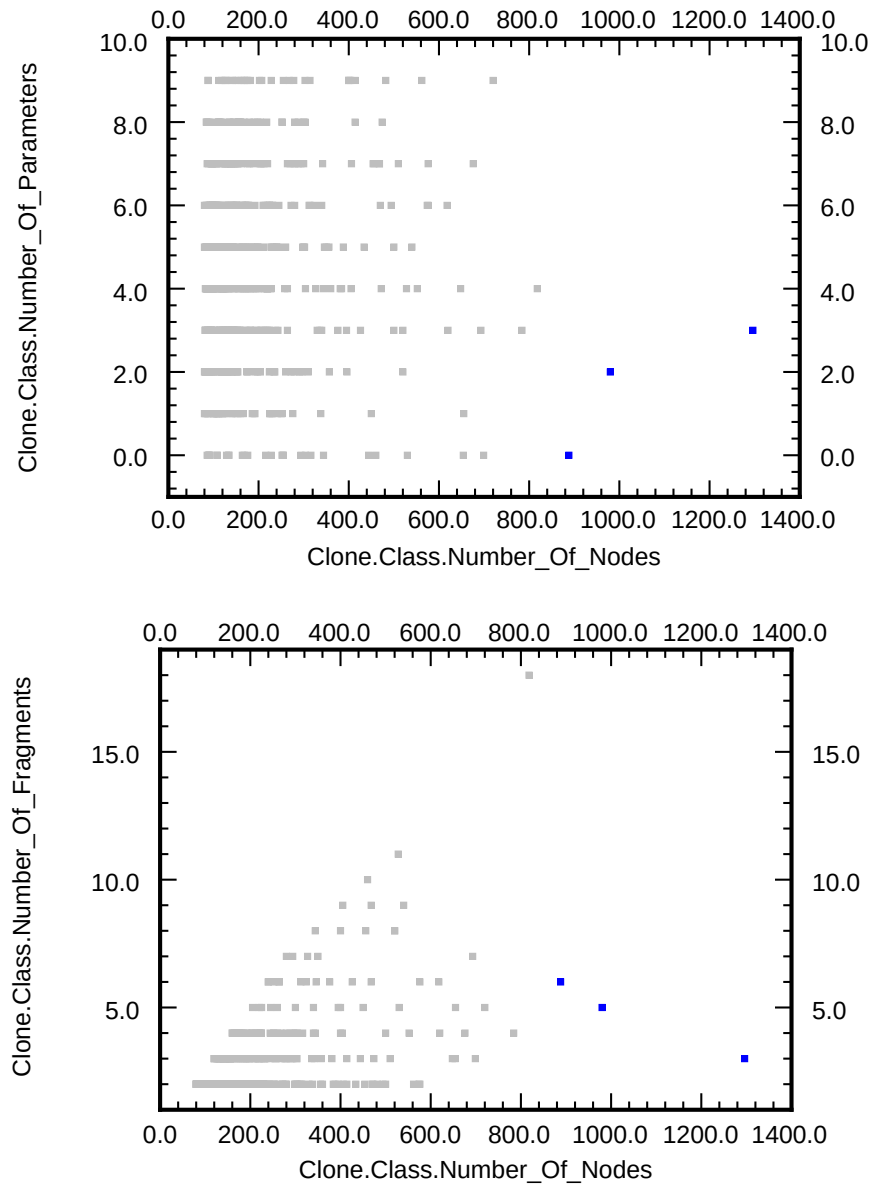


Abbildung 5.7: *Number of Nodes* in Relation zu *Number of Fragments* und *Number of Parameters*

gegen den Mittelwert der *Number of Nodes* einer Klonklasse abgetragen. Wäre in jeder Zeile eine Anweisung enthalten, so müsste sich hier eine Gerade ergeben oder die Punkte zwischen zwei Geraden liegen. Hier ist das Ergebnis jedoch sehr weit aufgefächert und wenn man sich die Zahlen anschaut, so fallen auch Klassen mit verhältnismäßig wenig *IML*-Knoten pro Zeile auf. So enthält der zeilenmäßig größte Klon mit circa 400 Zeilen nur rund 120 Knoten. Dies entspricht nur einem Knoten alle $3,3$ Zeilen. Eine einfache Zuweisung erzeugt in der *IML* schon drei Knoten, was darauf hindeutet, dass sehr wenige Anweisungen in dem betroffenen Abschnitt enthalten sind.

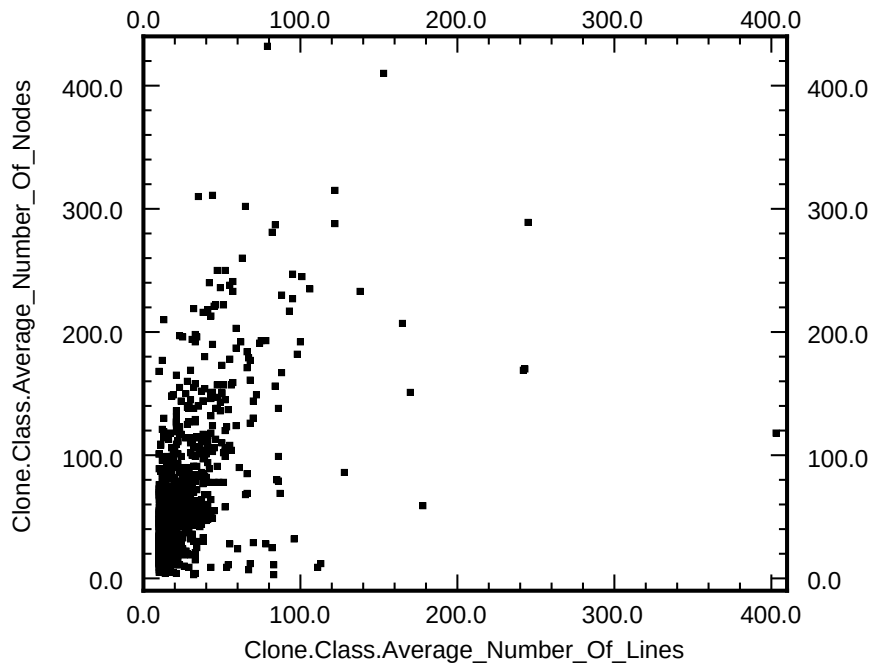


Abbildung 5.8: *Average Number of Lines* in Relation zu *Average Number of Nodes*

Leider stellte sich schnell heraus, dass nicht viele Klonklassen in dem analysierten System enthalten sind, die sich entfernen lassen und dabei auch ein Nutzen für die nichtfunktionale Anforderung der Größe entsteht. Nach bereits vier der interessantesten Klonklassen waren die Auswirkungen schon negativ, so dass insgesamt mehr Speicher für die Funktionsaufrufe verbraucht wurde als durch das Entfernen eingespart werden konnte. Auf Grund dieser geringen Anzahl kann keine zuverlässige Aussage darüber gemacht werden, welche Klonklassen entfernt werden sollten und welche nicht.

Auswertungsdaten

Insgesamt wurden 20 Klonklassen entfernt. Das Kriterium dafür war das Verhältnis der Werte *Number of Nodes* zu *Number of Fragments*. Sind besonders viele Knoten in einem Fragment enthalten, so ist hier die Wahrscheinlichkeit am höchsten, dass eingespart

5.3. Einsparpotential der Klonklassen

werden kann. Die Ergebnisse der ersten sechs Klonklassen sind in Abbildung 5.9 dargestellt. Die Weiteren wurden nicht aufgeführt, da sie alle einen größeren Objektcode zur Folge haben und keine weiteren Erkenntnisse bringen.

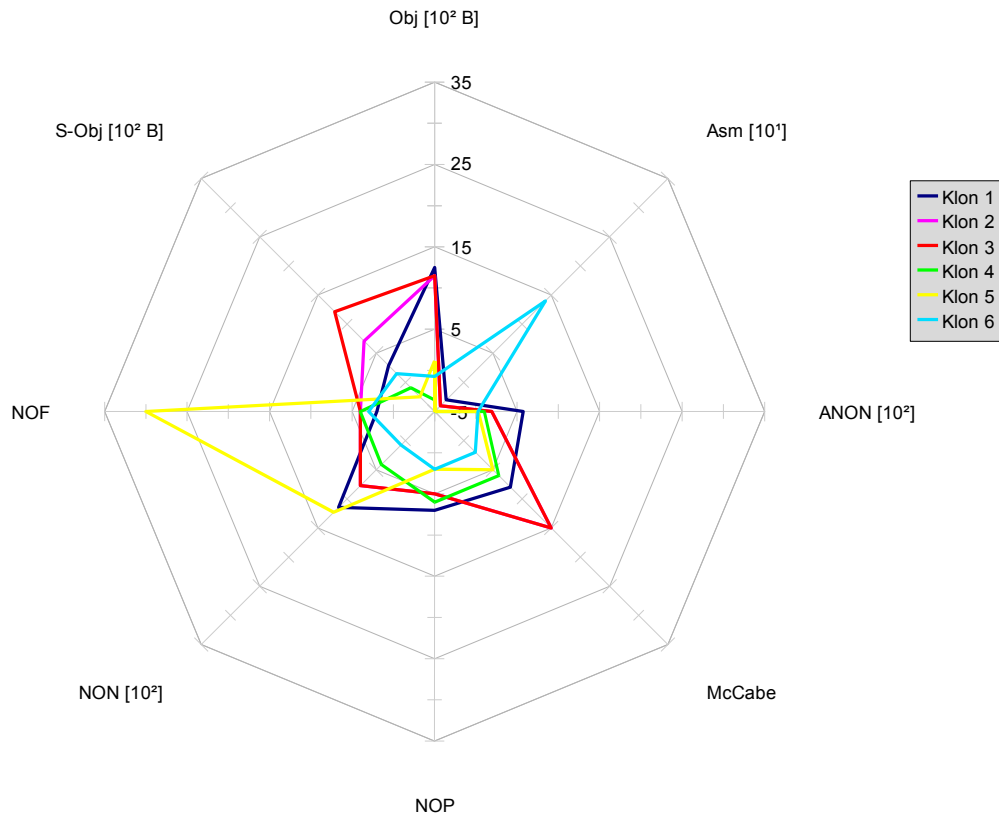


Abbildung 5.9: Metrikwerte

Die Reihenfolge, in der die Klonklassen entfernt wurden, ist durch die Metrik *Average Number of Nodes*, in der Grafik mit *ANON* abgekürzt, festgelegt. Darüber hinaus sind noch die Metrikwerte für die McCabe Komplexität, *Number of Parameters* (kurz *NOP*), *Number of Nodes* (kurz *NON*), *Number of Fragments* (kurz *NOF*), die Objektgröße (kurz *Obj*), die Objektgröße ohne Debuginformationen (kurz *S-Obj*) und die Änderung in den Assembleranweisungen erfasst. Die genauen Werte sind in Tabelle 5.4 enthalten.

Die Werte *Obj*, *S-Obj* und *Asm* geben die Ersparnis in Bezug auf die gemessene Eigenschaft an. Positive Werte bedeuten eine Reduzierung der Größe und sind somit relevant für die nichtfunktionalen Anforderungen. Zu beachten sind hierbei die verschiedenen Wertebereiche der Achsen, die gewählt wurden, um einen erkennbaren Graphen zu erhalten. Es ist auffällig, dass Klon 2 und Klon 3 großteils gleiche Werte aufweisen. Wenn man die beiden Klonklassen analysiert, so sind es Typ 1 Klonklassen, die zur selben Typ 2 Klonklasse gehören, sich aber nicht gemeinsam entfernen lassen. Aus diesem Grund werden die beiden Typ 1 Klassen getrennt gemeldet und müssen auch separat entfernt

	<i>ANON</i>	Obj	S-Obj	<i>NOF</i>	<i>NON</i>	<i>NOP</i>	McCabe	Asm
Klon 1	573	1248	288	2	1146	7	8	-3
Klon 2	193	1148	708	4	772	5	15	-4
Klon 3	193	1148	1212	4	772	5	15	-4
Klon 4	103	-360	-92	4	412	6	6	-27
Klon 5	30	116	-248	30	123	2	5	-15
Klon 6	25	-72	152	3	75	2	2	14

Tabelle 5.4: Metrikwerte der ersten entfernten Klonklassen

werden. Da die Werte gleich sind, ist nur eine der beiden Linien in der Grafik zu sehen, nur bei dem Wert von S-Obj weichen sie voneinander ab.

Interessant für die nichtfunktionalen Anforderungen sind die Werte S-Obj und Asm, da sie angeben, wie sich diese geändert haben. Der erste gibt an, wie sich der Speicherverbrauch im Flash verändert hat, während der zweite anzeigt, wie viele Assembleranweisungen eingespart werden konnten. Die eingesparten Assembleranweisungen variieren von Kontext zu Kontext, da der Compiler durch die Extraktion scheinbar besser oder schlechter optimieren konnte. Die Varianz war aber nicht hoch und es war immer maximal eine einstellige Wertedifferenz. Um die Einsparung zu bestimmen wurde zunächst berechnet wie viele Assembleranweisungen in der Funktion, aus der der Klon entfernt wurde, gespart werden konnte. Hiervon wurde die Anzahl der Anweisungen der neuen Funktion abgezogen und es ergibt sich die Anzahl an Assembleranweisungen, die in dem Kontext durch den Funktionsaufruf hinzugekommen sind. In der Tabelle sind die gemittelten und im Anschluss gerundeten Werte aufgeführt.

Aus den erhobenen Werten lassen sich keine Gesetzmäßigkeiten feststellen. Es kann aber zumindest gesagt werden, dass zwischen keinen der gemessenen Metriken und den untersuchten nichtfunktionalen Anforderungen ein direkter Zusammenhang besteht. Es ist zudem fraglich, ob bei der gewählten Optimierungsstufe eine sichere Vorhersage möglich ist. Wenn man alle analysierten Klonklassen betrachtet, so kann man feststellen, dass das Einsparpotenzial von einer Reihe von Faktoren wie Größe, Anzahl der Fragmente, Komplexität und Anzahl der geänderten Zugriffe abhängt. Bei einer Reihe von Klonklassen mussten auf Grund der Parameterübergaberegeln einige Variablen, die vorher *by Value* verwendet wurden, *by Reference* übergeben werden. Hierdurch muss an jeder Verwendungsstelle das Objekt dereferenziert werden, was einem höheren Aufwand als einen einfachen Zugriff entspricht.

Abschließend lässt sich sagen, dass sich die in dem System vorhandenen Klone nicht dafür eignen mittels des Refactorings *Extract Method* entfernt zu werden, da die einzelnen Fragmente in der Mehrzahl der Fälle zu wenige Anweisungen besitzen und so die nichtfunktionalen Anforderungen negativ beeinflusst werden.

6 Fazit

*Wer fertig ist, dem ist nichts recht zu machen,
ein Werdender wird immer dankbar sein.*

Johann Wolfgang von Goethe

Dieses Kapitel soll noch einmal einen abschließenden Überblick über die Diplomarbeit geben. Eine Zusammenfassung der Aufgaben und der erreichten Ziele findet sich in Abschnitt 6.1. Abschnitt 6.2 enthält eine abschließende Interpretation der Ergebnisse der Auswertung. Im darauf folgenden Abschnitt 6.3 werden Schwachstellen in dem von mir gewählten Ansatz genannt und mögliche Lösungswege aufgezeigt. Weitere Ideen für Verbesserungen der verwendeten Tools, um die Aufgabe besser und genauer lösen zu können, finden sich im abschließenden Abschnitt 6.4.

6.1 Überblick

Ausgeschriebenes Thema der Diplomarbeit war „Klonerkennung für eingebettete Systeme“. Es bestand in der Forschungsabteilung der Robert Bosch GmbH die Vermutung, dass die dort vorhandenen Systeme Klone enthalten. Um diese Frage zu beantworten wurde die Diplomarbeit ausgeschrieben. In der Ausschreibung wurden jedoch keine konkreten Ziele definiert, da zunächst geprüft werden sollte, wie gut die Klonerkennungsverfahren für Software der Automobilbranche arbeiten. In der Anfangsphase meiner Untersuchungen entwickelte sich dann das Thema dieser Arbeit, da bereits nach einer ersten Analyse des Systems festzustellen war, dass die Software Klone enthält, die soweit möglich, beseitigt werden sollten. Nach einem Gespräch mit meinem Betreuer haben wir die Aufgabe „Klonmanagement“ gewählt. Es wurden zwei Anwendungsfälle für ein solches System identifiziert, um die Anforderungen an eine solche Software genauer zu definieren.

Aus den Anwendungsfällen ergab sich zum einen der Bedarf einer kompakten Repräsentation der Ergebnisse sowie eine Bewertung der Klone nach ihrer Entfernbarekeit, um damit den Nutzer bei seiner Arbeit zu unterstützen und uninteressante Klone schon vor der Anzeige auszufiltern. Um die erste Anforderung zu lösen, wurden Klonklassen zur Datenrepräsentation gewählt, die um einige Eigenschaften erweitert wurden, um so die Informationsmenge noch kompakter visualisieren zu können. Diese Anforderung wurde

auch erfolgreich mit dem System gelöst. Somit ist die Darstellung für den Nutzer intuitiver zu bedienen und es werden auch weniger Daten dargestellt. Die zweite Anforderung konnte auf Grund mangelnder geeigneter Klone nur teilweise gelöst werden. Es wurde zunächst nach Möglichkeiten gesucht, wie Klone sinnvoll entfernt werden können, um daraus dann die Bewertungskriterien abzuleiten. In der Programmiersprache C lässt sich nur ein sinnvoller Weg finden und zwar das Refactoring *Extract Method*. Vor dem Hintergrund der nichtfunktionalen Anforderung, der Laufzeit und dem Speicherverbrauch sollten deren wahrscheinliche Änderung der entscheidende Faktor für die Bewertung sein. Um hier zunächst einen Einstieg zu finden, wurden für die Klonfragmente und Klonklassen eine Reihe von Metriken berechnet, unter anderem die Anzahl an Parametern für eine extrahierte Funktion und die Anzahl der inkompatiblen Datentypen dieser Parameter. Mit diesen Daten sollten Klonklassen gefunden werden, bei denen ein Refactoring in C überhaupt möglich und sinnvoll ist. Bei diesem Schritt stellte sich jedoch bald heraus, dass diese Klonklassen in der analysierten Variante des Systems nicht sehr häufig sind, so dass keine ausreichende Datenbasis zustande kam, um eine Bewertungsregel zu finden. Dennoch lassen sich die erhobenen Daten gut dazu nutzen, die Klonklassenmenge noch weiter einzuschränken. Zudem lassen sich aus diesen Daten Ergebnisse über die Aufgabenstellung hinaus finden, die durchaus interessant sind und weitere Entwicklungsziele aufzeigen.

6.2 Interpretation der Ergebnisse

Die Ergebnisse aus Abschnitt 5.3 dürfen nicht ohne ihren Kontext betrachtet werden. Ein vorschneller Schluss aus den Ergebnissen wäre, dass in der analysierten Variante ein Klonmanagement nicht notwendig sei, da scheinbar keine interessanten Klone vorhanden sind. Die Situation stellt sich jedoch anders dar, wenn man mit dem Tool *clones* nach Klonen sucht und diese genauer analysiert. So finden sich zum Beispiel einige Dateien in dem untersuchten System, die bis auf wenige Änderungen identisch sind.

Ein Dateipaar, bei der jede Datei 870 Zeilen umfasst, unterschied sich lediglich in 82 Zeilen, also weniger als 10% Differenz. Die Zeilen sind gleichmäßig über die Dateien verteilt, so dass die gemeldeten Klonpaare von *ccdiml* nur zehn bis zwanzig Zeilen lang sind und vorerst uninteressant erscheinen. Da sich die Zeilen nur durch ein Zeichen unterscheiden liegt der Verdacht nahe, dass die gesamte Datei durch Klonen entstanden ist und ein solches Paar müsste bei Änderungen beobachtet werden, so dass diese konsistent geschehen.

Ein zweites Paar, das sich sehr ähnlich ist, beinhaltet zwei Dateien, die jeweils 645 Zeilen lang sind. Es unterscheiden sich hier nur 34 Zeilen, jedoch gibt es hier Modifikationen der Klonfragmente, so dass neben geänderten auch eingefügte oder gelöschte Zeilen existieren.

Diese beiden Beispiele machen deutlich, dass durchaus Klone existieren aber sich bei einer genaueren Betrachtung herausstellt, dass sie in der Sprache C nicht entfernt werden

können. Betrachtet man diese, intuitiv als Klon erkennbaren Dateien, so fällt auf, dass hier die Programmiersprache C nicht mächtig genug ist, solche Klone zu vermeiden. So operieren die Funktionen in beiden Paaren auf verschiedenen Datenstrukturen, die unterschiedliche Namen haben, deren Aufbau jedoch ähnlich ist. Im ersten Fall sind die Strukturen sogar identisch. Der Grund für die verschiedenen Namen liegt im Misra-Standard, der ein solches Vorgehen empfiehlt, um so eine strenge Typsicherheit in der Sprache C zu erreichen. Mit Hilfe geeigneter Sprachmittel, in diesem Fall Templates, wie sie die Sprachen C++ oder Ada unterstützen, könnten hier auf der Quelltextebene Klone eingespart werden, um auch so die Wartbarkeit zu verbessern, ohne die Sicherheit und Stabilität des Codes negativ zu beeinflussen. In dem zweiten Beispiel lässt sich das Problem nicht mit einem Templatemechanismus lösen, da hier Anweisungen hinzukommen oder wegfallen. Jedoch sind auch hier wiederum die Strukturen, auf denen die Algorithmen operieren sehr ähnlich und haben einen gemeinsamen Kern an gleichen Feldern. Auf diesen Feldern wird in den durch *ccdiml* erkannten Klonen operiert und die „Lücken“ enthalten die Operationen auf den Feldern, die nicht gemeinsam sind. Hier würde ein objektorientierter Ansatz mit Vererbung weiterhelfen, um so aus der kleineren Struktur eine Basisklasse für die größere Struktur zu machen, was in diesem Fall anwendbar wäre. So müsste ein Teil der Algorithmen nur einmal implementiert werden, was Klone vermeidet.

Insgesamt lassen sich die Ergebnisse der Untersuchung dahingehend zusammenfassen, dass in der analysierten Software Klone enthalten sind, die jedoch nicht ohne weiteres entfernt werden können. Der Grund hierfür ist in den meisten Fällen in der Sprache C zu finden, die keine geeigneten Mittel bietet, diese Klone zu vermeiden. Hier würden Aspekte der objektorientierten Programmierung weiterhelfen, um Klone zumindest auf der Quellcodeebene vorzubeugen. Dies führt zu dem Schluss, dass zum einen die vorgenommene automatisierte Bewertung nicht mächtig genug ist, solche Fälle zu identifizieren aber auch, dass der Aspekt der konsistenten Änderungen vor diesem Hintergrund eine stärkere Bedeutung erhält. Ein System, das diese Zielsetzung verfolgt wird in [Toomim u. a. \(2004\)](#) beschrieben. Wenn der Nutzer einen Bereich editiert, der durch klonen entstanden ist, so hat er die Möglichkeit, dass das System die Änderungen konsistent in allen Klonfragmenten übernimmt.

6.3 Schwachstellen

Bereits bei der Planung als auch während der Realisierung sind einige Schwachstellen in der von mir angestrebten Umsetzung aufgefallen. Andere dieser Schwachstellen wurden bewusst in Kauf genommen, um so Analysen zu ermöglichen, die sonst sehr schwer oder gar nicht möglich gewesen wären.

Präprozessor

Durch das verwendete Klonerkennungstool *ccdimpl* ist es leider nicht möglich, Klone zwischen verschiedenen Präprozessorkonfigurationen zu finden. Diese Problematik wurde bereits in Abschnitt 3.3.1 angedeutet.

In der Entwicklungsphase wurden hierfür zwei Lösungswege entwickelt, die diesen Missstand beseitigen könnten. Zum einen besteht die Möglichkeit mit *ccdimpl* verschiedene Präprozessorkonfigurationen zu analysieren, um so Klone zu finden, die in Bereichen liegen, die nur bedingt kompiliert werden. Die Ergebnisse müssten dann so miteinander kombiniert werden, dass bestehende Klonklassen vergrößert werden. Hierdurch werden aber keine Klone zwischen Programmstellen gefunden, die nie zusammen in einer Konfiguration auftauchen können. Um solche Klone zu finden, müsste man ein Tool verwenden, das den Quelltext vor dem Präprozessieren analysiert, wie *clones*. In dessen Ergebnissen könnte man nach Übereinstimmungen mit denen von *ccdimpl* suchen. Klassen, die gemeinsame Fragmente haben, kann man vereinigen, um so die Treffermenge zu vergrößern. Sollte ein gemeldetes Fragment von *clones* eines von *ccdimpl* überdecken, so ist dies ein ausreichendes Kriterium für die Vereinigung. Ein Nachteil dieser Vorgehensweise ist jedoch, dass für die von *clones* gefundenen Fragmente keine Repräsentation in der *IML* vorhanden ist.

Aufteilung der Klonklassen

In der von mir implementierten Software wird in allen Fällen versucht, die gesamte Klonklasse durch eine Funktion zu extrahieren. Hierdurch kann es passieren, dass die *Number of different Parameters* größer als Null ist, die Fragmente aber durchaus bei einer weiteren Unterteilung der Klonklasse in mehrere gleiche Funktionen extrahiert werden könnten. Da bereits für jedes Fragment ein Parametervektor bestimmt wird, die zur Zeit auf Gleichheit geprüft werden, könnte man diese auf Grund der Gleichheit partitionieren, so dass die Klassen weiter unterteilt werden können.

Ergebnisse von Number of Parameters

In manchen Situationen kann der Wert der Metrik *Number of Parameters* zu hoch sein. In der implementierten Analyse werden keine Kontrollflussinformationen berücksichtigt, wodurch einige Variablen als Parameter angenommen werden, die aber keine Bedeutung haben. Ein Beispiel sind Laufvariablen für *for*-Schleifen, die bedeutungslos für den weiteren Programmablauf sind.

Insgesamt gibt es für eine Variable, die in einem Klonfragment verwendet wird, vier verschiedene Fälle. Wird eine Variable in dem Fragment als erstes gelesen, so muss sie als *in*-Parameter übergeben werden. Folgt nach dem Lesen noch ein Schreiben, so muss sie ein *in out*-Parameter sein. Wird eine Variable als erstes beschrieben, so muss sie,

in dem Fall, dass sie nach dem Fragment als erstes gelesen wird, als *out*-Parameter übergeben werden. Wird sie im Anschluss wieder überschrieben, so muss sie gar nicht übergeben werden. Variablen die außerhalb des Klonfragments außer einer Initialisierung weder geschrieben, noch gelesen werden, können komplett aus der umgebenden Funktion entfernt werden.

Diese Untersuchung der Parameter könnte ebenfalls in die Analyse mit aufgenommen werden, um die Ergebnisse zu verbessern. Der zur Zeit gemeldete Wert geht von der pessimistischen Annahme aus, dass alle Parameter sowohl gelesen als auch geschrieben werden und dies auch vor und nach dem Klonfragment geschieht wodurch die Übergabe zwingend erforderlich ist.

Statische Laufzeitbestimmung

Der von mir gewählte Ansatz geht davon aus, dass die Anzahl der ausgeführten Assembleranweisungen linear mit der Laufzeit zusammenhängt. Dies ist bei Prozessoren, die eine CISC-Architektur verwenden, nicht immer der Fall, da hier einzelne Anweisungen auch länger als einen Takt brauchen können. Zudem erschweren Techniken wie Pipelining die Laufzeitvorhersage.

6.4 Verbesserungen

Um die Ergebnisse der Klonerkennung in der von mir analysierten Software zu verbessern gibt es verschiedene Möglichkeiten. Um die Analysen auf der *IML* effektiver zu gestalten könnte man alle Präprozessorconfigurationen parallel in einer *IML* darstellen. Für den allgemeinen Fall ist dies so gut wie unmöglich, da die nicht verwendeten Präprozessorzweige unter Umständen nicht syntaktisch korrekt sind. Dies kann in diesem System jedoch nicht auftreten, da der Misra-Standard dieses ausschließt. Um dies zu erreichen müsste jedoch viel Aufwand in das verwendete Frontend *cafe++* als auch in die Klonerkennung *ccdimpl* gesteckt werden. Jedoch lassen sich auf dieser *IML* nur Klone zwischen verschiedenen Konfigurationen einer Variante analysieren und nicht variantenübergreifend.

A ICC-Syntax

Das von mir verwendete Dateiformat für die erweiterte Klonklassendateien ist stark an die bisher existierenden Formate *cpf* und *ccf*¹ angelehnt. Es ist ein reines ASCII-Dateiformat, dessen Aufbau durch die BNF (Backus Naur Form) in Listing A.1 festgelegt ist. Weitere Informationen zu der Bedeutung sind der Beschreibung in Abschnitt 4.1 zu entnehmen.

```
1 newline := '\n'
2 separator := '\t'
3 digit := '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
4 digit0 := '0' | digit
5 positive := digit [digit0*]
6 natural := digit0 [digit0*]
7 from_line := positive
8 from_node := positive
9 from_file := valid-filename
10 to_line := positive
11 to_node := positive
12 type := '1' | '2' | '3'
13 version := natural '.' natural
14 fragment := from_file separator from_line separator to_line
15             separator from_node separator to_node newline
16 subbody := fragment [fragment+]
17 subclass := '@subclass ' type newline subbody newline
18 body := [fragment*] [subclass*]
19 class := '@class separator type newline body newline
20 file := '@ICP ' version newline [class*]
```

Listing A.1: BNF des ICC-Formats

¹<http://www.bauhaus-stuttgart.de/clones>

Literaturverzeichnis

- [Baker 1995] BAKER, B. S.: On finding duplication and near-duplication in large software systems. In: *WCRE '95: Proceedings of the Second Working Conference on Reverse Engineering*. Washington, DC, USA : IEEE Computer Society, 1995, S. 86. – ISBN 0-8186-7111-4
- [Baxter u. a. 1998] BAXTER, Ira D. ; YAHIN, Andrew ; MOURA, Leonardo ; SANT'ANNA, Marcelo ; BIER, Lorraine: Clone Detection Using Abstract Syntax Trees. In: *Proceedings of the 14th International Conference on Software Maintenance (ICSM)* IEEE CS (Veranst.), 1998, S. 368–377
- [Beckwermert 2006] BECKWERMERT, Felix: Visualisierung von Klonerkennung / Universität Bremen. Oktober 2006. – Diplomarbeit
- [Bellon 2007] BELLON, Stefan: Comparison and Evaluation of Clone Detection Tools. In: *IEEE Trans. Softw. Eng.* 33 (2007), Nr. 9, S. 577–591. – Rainer Koschke and Giulio Antoniol and Jens Krinke and Ettore Merlo. – ISSN 0098-5589
- [Burd und Bailey 2002] BURD, Elizabeth ; BAILEY, John: Evaluating Clone Detection Tools for Use during Preventative Maintenance. In: *SCAM '02: Proceedings of the Second IEEE International Workshop on Source Code Analysis and Manipulation (SCAM'02)*. Washington, DC, USA : IEEE Computer Society, 2002, S. 36. – ISBN 0-7695-1793-5
- [Chiu und Hirtle 2007] CHIU, Andy ; HIRTLE, David: *Beyond Clone Detection*. 2007. – http://www.cs.uwaterloo.ca/~dhirtle/publications/beyond_clone_detection.pdf, 10.12.2007
- [Erlikh 2000] ERLIKH, Len: Leveraging Legacy System Dollars for E-Business. In: *IT Professional* 2 (2000), Nr. 3, S. 17–23. – ISSN 1520-9202
- [Fahrmeir u. a. 2007] FAHRMEIR, Ludwig ; KÜNSTLER, Rita ; PIGEOT, Iris ; TUTZ, Gerhard: *Statistik. Der Weg zur Datenanalyse*. Springer, Berlin, 2007. – ISBN 3540440003
- [Falke u. a. 2008] FALKE, Raimar ; FRENZEL, Pierre ; KOSCHKE, Rainer: *Empirical Evaluation of Clone Detection Using Syntax Suffix Trees*. 2008. – Zur Veröffentlichung eingereicht
- [Fanta und Rajlich 1999] FANTA, Richard ; RAJLICH, Václav: Removing clones from the code. In: *Journal of Software Maintenance* 11 (1999), Nr. 4, S. 223–243. – ISSN 1040-550X

- [Fenton und Pfleeger 1998] FENTON, Norman E. ; PFLEEGER, Shari L.: *Software Metrics: A Rigorous and Practical Approach*. Boston, MA, USA : PWS Publishing Co., 1998. – ISBN 0534954251
- [Fowler 1999] FOWLER, Martin: *Refactoring: improving the design of existing code*. Boston, MA, USA : Addison-Wesley Longman Publishing Co., Inc., 1999. – ISBN 0-201-48567-2
- [Giesecke 2003] GIESECKE, Simon: Clone-based Reengineering für Java auf der Eclipse-Plattform / Universität Oldenburg. 2003. – Diplomarbeit
- [ISO/IEC 9899:1990 1990] INTERNATIONAL ORGANIZATION FOR STANDARDIZATION: *ISO/IEC 9899:1990: Programming languages — C*. Geneva, Switzerland : International Organization for Standardization, 1990
- [Jiang u. a. 2006] JIANG, Zhen M. ; HASSAN, Ahmed E. ; HOLT, Richard C.: Visualizing Clone Cohesion and Coupling. In: *APSEC '06: Proceedings of the XIII Asia Pacific Software Engineering Conference*. Washington, DC, USA : IEEE Computer Society, 2006, S. 467–476. – ISBN 0-7695-2685-3
- [Koschke 2006] KOSCHKE, Rainer: Software-Reengineering / Universität Bremen. 2006. – Vorlesungsskript
- [Krinke 2001] KRINKE, Jens: Identifying Similar Code with Program Dependence Graphs. In: *WCRE '01: Proceedings of the Eighth Working Conference on Reverse Engineering (WCRE'01)*. Washington, DC, USA : IEEE Computer Society, 2001, S. 301. – ISBN 0-7695-1303-4
- [Lague u. a. 1997] LAGUE, Bruno ; PROULX, Daniel ; MAYRAND, Jean ; MERLO, Ettore M. ; HUDEPOHL, John: Assessing the Benefits of Incorporating Function Clone Detection in a Development Process. In: *ICSM '97: Proceedings of the International Conference on Software Maintenance*. Washington, DC, USA : IEEE Computer Society, 1997, S. 314. – ISBN 0-8186-8013-X
- [Mayrand u. a. 1996] MAYRAND, Jean ; LEBLANC, Claude ; MERLO, Ettore: Experiment on the Automatic Detection of Function Clones in a Software System Using Metrics. In: *ICSM '96: Proceedings of the 1996 International Conference on Software Maintenance*. Washington, DC, USA : IEEE Computer Society, 1996, S. 244. – ISBN 0-8186-7677-9
- [Misra 2004] MOTOR INDUSTRY SOFTWARE RELIABILITY ASSOCIATION: *MISRA-C:2004 — Guidelines for the use of the C language in critical systems*. Nuneaton : MIRA Limited, 2004
- [Monden u. a. 2002] MONDEN, Akito ; NAKAE, Daikai ; KAMIYA, Toshihiro ; SATO, Shin ichi ; MATSUMOTO, Ken ichi: Software Quality Analysis by Code Clones in Industrial Legacy Software. In: *METRICS '02: Proceedings of the 8th International Symposium on Software Metrics*. Washington, DC, USA : IEEE Computer Society, 2002, S. 87. – ISBN 0-7695-1339-5
- [Paulson u. a. 2004] PAULSON, James W. ; SUCCI, Giancarlo ; EBERLEIN, Armin: An Empirical Study of Open-Source and Closed-Source Software Products. In: *IEEE Trans. Softw. Eng.* 30 (2004), Nr. 4, S. 246–256. – ISSN 0098-5589

- [Shneiderman 1996] SHNEIDERMAN, Ben: The Eyes Have It: A Task by Data Type Taxonomy for Information Visualizations. In: *IEEE Visual Languages*. College Park, Maryland 20742, U.S.A., 1996, S. 336–343
- [Tairas u. a. 2006] TAIRAS, Robert ; GRAY, Jeff ; BAXTER, Ira: Visualization of clone detection results. In: *eclipse '06: Proceedings of the 2006 OOPSLA workshop on eclipse technology eXchange*. New York, NY, USA : ACM, 2006, S. 50–54. – ISBN 1-59593-621-1
- [Toomim u. a. 2004] TOOMIM, Michael ; BEGEL, Andrew ; GRAHAM, Susan L.: Managing Duplicated Code with Linked Editing. In: *VLHCC '04: Proceedings of the 2004 IEEE Symposium on Visual Languages - Human Centric Computing (VLHCC'04)*. Washington, DC, USA : IEEE Computer Society, 2004, S. 173–180. – ISBN 0-7803-8696-5
- [Zelkowitz u. a. 1979] ZELKOWITZ, Marvin V. ; SHAW, Alan C. ; GANNON, John D.: *Principles of Software Engineering and Design*. Prentice Hall Professional Technical Reference, 1979. – ISBN 013710202X