

**Studiengang:** Informatik

**Prüfer:** Prof. Dr. Erhard Plödereder

**Betreuer:** Dr. Rainer Koschke

**begonnen am:** 30. März 2002

**beendet am:** 13. September 2002

**CR-Klassifikation:** D.2.7

Diplomarbeit Nr. 1998

# **Vergleich von Techniken zur Erkennung duplizierten Quellcodes**

Stefan Bellon

Institut für Informatik  
Universität Stuttgart  
Breitwiesenstraße 20-22  
D-70565 Stuttgart



# **Vergleich von Techniken zur Erkennung duplizierten Quellcodes**

Stefan Bellon  
sbellon@sbellon.de

13. September 2002



# Inhaltsverzeichnis

<b>1. Einleitung</b>	<b>7</b>
1.1. Hintergrund der Diplomarbeit	7
1.2. Aufgabenstellung	7
1.3. Anmerkungen	8
1.4. Überblick	8
<b>2. Vorfeld</b>	<b>9</b>
2.1. Definition von Klontyp	9
2.2. Definition von Klonpaar	10
2.3. Definition von Codefragment	12
2.4. Ansätze zur Klonerkennung	12
2.4.1. Baker	12
2.4.2. Baxter	14
2.4.3. Kamiya	15
2.4.4. Krinke	16
2.4.5. Merlo	18
2.4.6. Rieger	19
2.5. Spezifikation von Kommunikationsformaten für Klone	20
2.6. Datenbank-Schema zur Verwaltung der Klondaten	21
2.7. Festlegung der Programmiersprachen im Experiment	24
<b>3. Durchführung</b>	<b>27</b>
3.1. Test-Phase	27
3.2. Erkenntnisse und Änderungen	28
3.3. Haupt-Phase	29
3.4. Konfiguration der Werkzeuge	30
3.4.1. Baker	30
3.4.2. Baxter	30
3.4.3. Kamiya	31
3.4.4. Krinke	33
3.4.5. Merlo	33
3.4.6. Rieger	34
3.5. Probleme bei der Durchführung	34
3.5.1. Baker	34
3.5.2. Baxter	34

## Inhaltsverzeichnis

3.5.3.	Kamiya . . . . .	37
3.5.4.	Krinke . . . . .	37
3.5.5.	Merlo . . . . .	37
3.5.6.	Rieger . . . . .	38
<b>4.</b>	<b>Auswertung</b>	<b>39</b>
4.1.	Erzeugen der Referenzmenge mittels Orakel . . . . .	39
4.2.	Berechnung der Daten zur späteren Analyse . . . . .	44
4.2.1.	Definition von Overlap und Contained . . . . .	44
4.2.2.	Definition von Good und OK . . . . .	45
4.2.3.	Bedeutung von Good und OK . . . . .	46
4.2.4.	Zuordnung von Kandidaten zu Referenzen . . . . .	48
<b>5.</b>	<b>Ergebnisanalyse</b>	<b>51</b>
5.1.	Definition der in der Ergebnisanalyse verwendeten Begriffe . . . . .	51
5.1.1.	Kandidaten . . . . .	51
5.1.2.	Referenzen . . . . .	52
5.1.3.	FoundSecrets . . . . .	54
5.1.4.	Rejected . . . . .	54
5.1.5.	TrueNegatives . . . . .	55
5.1.6.	Recall und Precision . . . . .	55
5.1.7.	Klongrößen . . . . .	56
5.1.8.	Klonverteilung . . . . .	57
5.1.9.	Verschiedenes . . . . .	57
5.2.	Auswertung nach Projekten . . . . .	58
5.2.1.	weltab . . . . .	60
5.2.2.	cook . . . . .	66
5.2.3.	snns . . . . .	72
5.2.4.	postgresql . . . . .	78
5.2.5.	netbeans-javadoc . . . . .	83
5.2.6.	eclipse-ant . . . . .	89
5.2.7.	eclipse-jdtcore . . . . .	96
5.2.8.	j2sdk1.4.0-javax-swing . . . . .	102
5.2.9.	Zusammenfassung . . . . .	108
5.3.	Abschließende Bewertung der Werkzeuge . . . . .	109
5.3.1.	Vorbemerkungen . . . . .	110
5.3.2.	Baker . . . . .	111
5.3.3.	Baxter . . . . .	112
5.3.4.	Kamiya . . . . .	114
5.3.5.	Krinke . . . . .	115
5.3.6.	Merlo . . . . .	117
5.3.7.	Rieger . . . . .	119
<b>6.</b>	<b>Kritischer Rückblick</b>	<b>121</b>

<b>7. Zusammenfassung</b>	<b>123</b>
<b>A. Revalidieren des Experiments</b>	<b>127</b>
A.1. Systemvoraussetzungen . . . . .	127
A.2. Installation . . . . .	128
<b>B. Implementierte Hilfsprogramme</b>	<b>129</b>
B.1. Programme zur Aufbereitung der Quelldateien der Projekte . . . . .	129
B.1.1. textclean . . . . .	129
B.1.2. codenormalize . . . . .	130
B.2. Programme zur Auswertung und Evaluierung . . . . .	131
B.2.1. cconvert . . . . .	131
B.2.2. clones . . . . .	132
<b>C. Dateiformate</b>	<b>135</b>
C.1. Klonpaar-Format . . . . .	135
C.1.1. BNF . . . . .	135
C.1.2. Beispiel . . . . .	135
C.2. Klonklassen-Format . . . . .	136
C.2.1. BNF . . . . .	136
C.2.2. Beispiel . . . . .	136
C.3. Auswertungsdaten . . . . .	137
C.3.1. DTD für die Auswertung im XML-Format . . . . .	137
C.3.2. Definitionen und Erklärungen . . . . .	139
<b>D. Kurzbeschreibung der analysierten Projekte</b>	<b>145</b>
D.1. Test-Phase . . . . .	145
D.1.1. bison . . . . .	145
D.1.2. wget . . . . .	145
D.1.3. EIRC . . . . .	145
D.1.4. spule . . . . .	145
D.2. Haupt-Phase . . . . .	146
D.2.1. weltab . . . . .	146
D.2.2. cook . . . . .	146
D.2.3. snns . . . . .	146
D.2.4. postgresql . . . . .	146
D.2.5. netbeans-javadoc . . . . .	146
D.2.6. eclipse-ant . . . . .	146
D.2.7. eclipse-jdtcore . . . . .	147
D.2.8. j2sdk1.4.0-javax-swing . . . . .	147
<b>Abkürzungsverzeichnis</b>	<b>149</b>
<b>Glossar</b>	<b>151</b>

*Inhaltsverzeichnis*

<b>Verzeichnis elektronischer Quellen</b>	<b>153</b>
<b>Literaturverzeichnis</b>	<b>155</b>

# 1. Einleitung

## 1.1. Hintergrund der Diplomarbeit

Copy&Paste ist noch immer das vorherrschende Programmierparadigma, wenn es um Wiederverwendung von Code geht. Dabei kopiert der Programmierer ein Stück Code an eine andere Stelle (und modifiziert die Kopie möglicherweise leicht), um dort eine ähnliche Funktionalität zu erreichen. Durch häufiges Copy&Paste leidet jedoch die Wartbarkeit des Systems. Ein Fehler muss eventuell an vielen Stellen korrigiert und eine Änderung an vielen Stellen vorgenommen werden. Allerdings ist in den seltensten Fällen dokumentiert, wohin ein Stück Code kopiert wurde.

In der Literatur wurden eine Reihe von Techniken zur Entdeckung so genannter Klone (also Code-Stücke, die sich aus Copy&Paste ergaben) vorgeschlagen. Jedoch ist bis dato unklar, welche der Techniken unter welchen Umständen die bessere ist. Die wichtigsten Wissenschaftler auf diesem Gebiet haben sich nun zusammengetan, um die verschiedenen Techniken quantitativ und qualitativ zu vergleichen.

## 1.2. Aufgabenstellung

Das Ziel dieser Diplomarbeit war der qualitative und quantitative Vergleich der verschiedenen Ansätze zur Erkennung von Klonen, namentlich der Techniken von:

- Baker et al. (siehe [33, 34, 35])
- Baxter et al. (siehe [36])
- Kamiya et al. (siehe [40])
- Krinke (siehe [46])
- Merlo et al. (siehe [43, 47, 48])
- Rieger et al. (siehe [37])

Die genannten Wissenschaftler haben sich an dem Vergleich beteiligt. Alle Aktivitäten sind in enger Zusammenarbeit mit dem Betreuer und den beteiligten Wissenschaftlern erfolgt. Die Aufgaben der vorliegenden Arbeit waren wie folgt:

- Erarbeitung des Vorgehens eines quantitativen Vergleiches der Ergebnisse verschiedener Techniken zur Erkennung von Klonen

## 1. Einleitung

- Auswahl von Java- und C-Systemen, auf die die Techniken angewandt werden sollen; eventuell deren Präparierung, um Tests zur Konfidenz über den Recall einzubauen
- Spezifikation des Formats der Resultate
- Versenden der ausgewählten Systeme sowie die Entgegennahme der Ergebnisse
- Quantitative und qualitative Analyse der Ergebnisse; insbesondere sollen anhand der Ergebnisse die speziellen Stärken und Schwächen einzelner Techniken sowie deren Komplementarität herausgearbeitet werden
- Konzeption und Implementierung von Werkzeugen zur Unterstützung der Messung und Evaluation, sofern erforderlich

Es war nicht Aufgabe der Arbeit, die Techniken selbst auf die selektierten Systeme anzuwenden. Diesen Teil haben die beteiligten Wissenschaftler erledigt.

In einem Vorexperiment anhand kleinerer Systeme wurde das Messverfahren und das Format zur Abgabe der Resultate evaluiert und verbessert. Im eigentlichen Hauptexperiment wurden dann umfangreichere Systeme untersucht.

### 1.3. Anmerkungen

Da die Diskussionen mit den verschiedenen Teilnehmern in Englisch erfolgt sind und somit die einzelnen Definitionen und Spezifikationen ursprünglich in Englisch entstanden sind, sind im Folgenden einige Begriffe im Englischen belassen und wurden nicht ins Deutsche übersetzt.

Des Weiteren wird bei der Trennung der Dezimalstellen von den ganzen Zahlen durchgängig die internationale Schreibweise verwendet, d. h. es wird ein Punkt anstelle eines Kommas gesetzt.

### 1.4. Überblick

In Kapitel 2 auf der nächsten Seite werden Definitionen und Spezifikationen präsentiert, die im Vorfeld des eigentlichen Experiments gemacht werden mussten. Kapitel 3 auf Seite 27 beschäftigt sich mit der Durchführung der Test- und Haupt-Phase. Das Vorgehen bei der Auswertung der gesammelten Daten wird in Kapitel 4 auf Seite 39 erläutert. Kapitel 5 auf Seite 51 befasst sich dann mit der Analyse der bei der Auswertung angefallenen Ergebnisse. Abschließend wird in Kapitel 6 auf Seite 121 kritisch auf das Experiment zurückgeblickt, und in Kapitel 7 auf Seite 123 werden noch einmal alle Ergebnisse kurz zusammengefasst.

## 2. Vorfeld

OBI-WAN: I have to admit that without the clones it would have not been a victory.

YODA: Victory! Victory you say? Master Obi-Wan, not victory. The shroud of the dark side has fallen, begun the clone war has.

*(Star Wars II – Attack of the Clones)*

Als Erstes müssen einige Begriffe, Vorgehensweisen und Ziele dieser Arbeit definiert und spezifiziert werden. Dazu ist es notwendig, die Fähigkeiten und Besonderheiten der einzelnen Ansätze zur Klonerkennung zu betrachten. Hierbei soll allerdings der Schwerpunkt nicht auf die Technik selbst gelegt werden, sondern auf die individuellen Fähigkeiten der einzelnen Werkzeuge und deren Auswirkungen im Vergleich mit den anderen Ansätzen. Um sich mit den einzelnen Techniken selbst vertraut zu machen, wird auf die jeweilige Beschreibung in anderen Veröffentlichungen verwiesen.

Da jedem der teilnehmenden Werkzeuge eine andere Arbeitsweise zu Grunde liegt, jedes Werkzeug verschiedene Schwerpunkte legt sowie unterschiedliche Bedürfnisse hat, muss zur Durchführung des Experiments ein gemeinsamer Nenner gefunden werden, damit die zu sammelnden Daten letztendlich auch vergleichbar sind und Schlüsse daraus gezogen werden können. Hierzu wurde im Vorfeld eine rege Diskussion der Teilnehmer auf der eigens hierfür eingerichteten Mailing-Liste geführt.

Im Folgenden sollen nun die einzelnen Ansätze näher betrachtet und wichtige Begriffe für das gesamte Experiment definiert werden. Die Reihenfolge orientiert sich an der logischen Abhängigkeit der einzelnen Definitionen voneinander.

Die Begriffe Klon und Codefragment können für das Experiment erst nach dem näheren Betrachten der verschiedenen Ansätze definiert werden. Sie hängen stark von den Fähigkeiten der einzelnen Werkzeuge ab. Zum Verständnis beim Lesen werden sie hier jedoch vor den Ansätzen behandelt.

### 2.1. Definition von Klontyp

Die Kopie eines Programmfragments (im Weiteren auch mit Codefragment bezeichnet) wird Klon genannt.

## 2. Vorfeld

Die folgenden verschiedenen Typen von Klonen werden im Verlauf des Experiments berücksichtigt werden:

### Typ 1: Exakte Kopie

- Keinerlei Veränderung an der Kopie (bis auf Whitespace und Kommentare)
- z. B. Inlining von Hand

### Typ 2: Kopie mit Umbenennungen (parametrisierte Übereinstimmung)

- Bezeichner werden in der Kopie umbenannt
- z. B. „Wiederverwendung“ einer Funktion, generische Funktion von Hand

### Typ 3: Kopie mit weiteren Modifikationen

- Code der Kopie wird abgeändert, nicht nur Bezeichner
- z. B. „Erweiterung“ einer Funktion

Dieser Einteilung in die verschiedenen Klontypen liegt die Definition von [44] zu Grunde.

## 2.2. Definition von Klonpaar

Will man Klone miteinander vergleichen, so muss man sich zuerst einmal einig darüber werden, was ein Klon überhaupt ist. Da hier jeder Teilnehmer etwas andere Vorstellungen hat, muss eine Definition von Klon gefunden werden, die für jedes Werkzeug akzeptabel ist. Die erste Eigenschaft, die es zu definieren gilt, ist der Typ der Gleichheit. Diese wurde den Teilnehmern fest vorgegeben (siehe Abschnitt 2.1 auf der vorherigen Seite) und orientiert sich an Koschkes Definition. Des Weiteren kann man aber einen Klon selbst (abgesehen von seinem Typ) auf verschiedene Arten definieren. Während der Diskussion mit den Teilnehmern haben sich drei Arten, einen Klon zu definieren, herauskristallisiert.

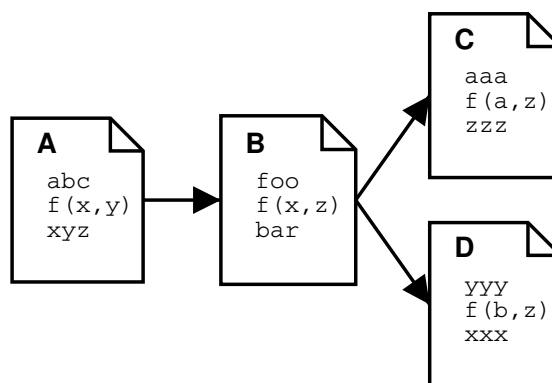


Abbildung 2.1: Beispiel eines Klon bestehend aus vier Codefragmenten

Anhand des Beispiels aus Abbildung 2.1 auf der vorherigen Seite sollen die verschiedenen Arten erläutert werden. Hierbei stelle man sich vor, dass ein Codefragment aus einer Datei A in eine Datei B kopiert wurde und von dort aus weiter nach C und D. Es wurden in jedem Schritt jeweils kleine Änderungen vorgenommen. Folgende drei Möglichkeiten, dies zu beschreiben, haben sich in der Diskussion ergeben:

- **Klonpaar**  
Als Klon wird jeweils ein Paar von zwei Codefragmenten betrachtet, wobei das eine Codefragment aus dem anderen durch Kopieren und eventuell Modifizieren hervorgegangen ist. Die Reihenfolge der zwei Codefragmente jedoch ist willkürlich. Bei dieser Definition kommt man um die Problematik der Transitivität bei Typ-3-Klonen herum, da man Klone nur paarweise betrachtet.  
Beispiel: (A, B) und (B, C) und (B, D) sind jeweils Klonpaare.
- **Klonklasse**  
Als Klon wird jeweils eine Menge von Codefragmenten betrachtet. Diese Codefragmente sind untereinander in irgend einer Art und Weise durch Kopieren und gegebenenfalls Modifizieren entstanden. Es existiert weder eine Ordnung in der Menge, noch können Aussagen über Transitivität gemacht werden, die für alle Werkzeuge gelten.  
Beispiel: (A, B, C, D) bilden eine Klonklasse.  
Werkzeuge, die nur Typ-1- und Typ-2-Klone erkennen und mit Klonklassen arbeiten, betrachten diese als transitiv. Werkzeuge, die zusätzlich noch Typ-3-Klone erkennen, unterscheiden sich hier und betrachten diese nicht unbedingt als transitiv. Aus diesem Grund gibt es letztlich noch:
- **Klonrelationenmenge**  
Diese Darstellung ist eine Mischung aus Klonpaar und Klonklasse. Dabei wird ein Codefragment in Relation zu einem oder mehreren anderen Codefragmenten gesetzt. Es wird jedoch keinerlei Aussage über die Beziehung der anderen in Relation gesetzten Codefragmente untereinander getroffen. Anders ausgedrückt, es wird aus einer Anzahl von Klonpaaren ein gemeinsames Codefragment „ausgeklammert“.  
Beispiel: B(A, C, D) ist eine Klonrelationenmenge, es existiert keine Aussage über die Beziehungen von A, C und D untereinander.

Baxter hat sich als einziger starker Befürworter für die Klonklasse herausgestellt. Alle anderen Teilnehmer haben sich für das Klonpaar ausgesprochen. Merlo hätte die Relationenmenge vorgezogen, ist aber auch mit der Wahl des Klonpaares als Vergleichsbasis einverstanden.

Schließlich war auch Baxter mit der Wahl des Klonpaares als gemeinsamer Nenner einverstanden, wollte aber seine Ergebnisse nicht als Paare, sondern als Menge einsenden. Aus diesem Grund wurden zwei Formate zum Einsenden der Resultate nötig (siehe Abschnitt 2.5 auf Seite 20 und die Abschnitte C.1 auf Seite 135 sowie C.2 auf Seite 136).

### 2.3. Definition von Codefragment

Die Definition eines Klonpaares alleine reicht noch nicht aus. Man muss sich ebenfalls Gedanken um die nächstkleinere Einheit in der obigen Definition, dem Codefragment, machen. Dies gibt die Granularität an, mit der Klone erkannt und verglichen werden können.

Hier unterscheiden sich die Fähigkeiten der einzelnen Werkzeuge wieder stark. Während einige Teilnehmer die Ergebnisse in einer sehr feinen Granularität liefern können – Baxter, Kamiya, Krinke und Merlo (für die Programmiersprache Java) können auch Spalteninformationen ausgeben – gibt es andere Werkzeuge, die lediglich auf Zeilen operieren, wodurch eine Codezeile die kleinste Einheit vorgibt: Baker und Rieger, sowie Merlo für die Programmiersprache C (siehe Abschnitt 2.4).

Nach einiger Diskussion auf der Mailing-Liste waren sich alle Teilnehmer darüber einig, dass es keinen großen Vorteil bringt, bei einem quantitativen Vergleich Spalteninformationen mitzubetrachten. Aus diesem Grund wurde das Tupel (Dateiname, Startzeile, Endzeile) als Definition für ein Codefragment gewählt.

Die Frage, welchen Umfang ein Klonpaar haben muss, ist offen gelassen. Das bedeutet, dass sowohl Klonpaare, die lediglich aus einer Anweisungssequenz bestehen, ebenso verglichen werden wie auch Klonpaare, die aus kompletten Dateien bestehen. Die Zwischenformen wie Funktionen, Methoden sowie Klassen und Strukturen sind selbstverständlich auch am Vergleich beteiligt. Das einzig einschränkende Kriterium der Codefragmente ist die Minimalgröße: Jedes Codefragment, das am Vergleich teilnehmen kann, muss mindestens sechs Zeilen im Quellcode umfassen. Beim Bilden der Referenzmenge (siehe Abschnitt 4.1 auf Seite 39) werden allerdings nur Klonpaare berücksichtigt, die durch Makros oder Funktionen ersetzt werden können oder Sequenzen solcher Klonpaare.

### 2.4. Ansätze zur Klonerkennung

In den nächsten sechs Abschnitten werden die einzelnen Ansätze, die den Werkzeugen der Teilnehmer zu Grunde liegen, vorgestellt. Dabei werden sowohl die Techniken selbst kurz erläutert als auch weitere für das Experiment relevante Details erwähnt.

#### 2.4.1. Baker

Das Verfahren von Brenda S. Baker, das sie in ihrem Programm `Dup` implementiert hat, arbeitet mit tokenbasiertem, zeilenorientiertem Pattern Matching. Dabei wird jede Codezeile in einen so genannten P-String umgewandelt, der die Struktur der Zeile sowie die Bezeichner umfasst. Das eigentliche Matching wird mittels eines Suffix-Trees realisiert.

Sei  $\Sigma$  ein endliches Alphabet, dessen Symbole Konstanten darstellen, und  $\Pi$  ein zu  $\Sigma$  disjunktes Alphabet, dessen Symbole Parameter darstellen, dann bezeichnet P-String eine Kette in  $(\Sigma \cup \Pi)^*$ . Kann man durch eine konsistente Umbenennung der Symbole aus  $\Pi$  einen P-String  $S_1$  in einen P-String  $S_2$  überführen, so werden  $S_1$  und  $S_2$  auch P-Match genannt. Hierbei müssen alle Symbole aus  $\Sigma$  unverändert bleiben.

Des Weiteren gibt es eine Funktion  $\text{prev} : (\Sigma \cup \Pi)^* \rightarrow (\Sigma \cup \mathbb{N})^*$ , so dass jedes erstmalige Vorkommen eines Symbols aus  $\Pi$  durch eine 0 und jedes weitere Vorkommen durch

die Anzahl von Positionen nach links zum letztmaligen Vorkommen ersetzt wird. Die ganzen Zahlen werden in der Darstellung geklammert, um Mehrdeutigkeiten bei mehrstelligen Zahlen zu vermeiden. Sei z. B.  $\Sigma = \{\alpha, \beta\}$  und  $\Pi = \{u, v, x, y\}$ , so gilt:  $\text{prev}(\alpha xy \beta x \alpha y x) = \alpha(0)(0)\beta(3)\alpha(4)(3) = \text{prev}(\alpha uv \beta u \alpha v u)$ .

Wenn  $S = b_1 \dots b_n$  ein P-String ist, dann ist P-Suffix wie folgt definiert:  $\text{p-suffix}(S, i) = b_i \dots b_n$  für  $i \leq n$  und der Leerstring sonst.

Ein P-Suffix-Tree zu  $S$  ist nun der Compacted Trie aller P-Suffixe für  $S$ , wobei ans Ende von  $S$  noch ein eindeutiges Endsymbol angefügt wird.

Anhand dieses P-Suffix-Trees lassen sich nun P-Matches ablesen, da der Pfad von der Wurzel zu einem Knoten allen Blättern unterhalb dieses Knotens gemeinsam ist.

Um dieses Verfahren nun auf Programmcode anwenden zu können, muss dieser erst in eine andere Struktur überführt werden. Hierzu werden Typen bestimmter Anweisungen durch so genannte Funktoren ersetzt und die Variablen hinten angehängt. So könnte diese Struktur für  $x = y; z(x);$  z. B. wie folgt aussehen:  $\alpha x y \beta z x$ , wobei  $\alpha$  für eine Zuweisung einer Variablen an eine andere und  $\beta$  für einen Funktionsaufruf mit einem Parameter und ohne Rückgabewert steht. Dies lässt sich nun in einen P-String und dann in einen prev-String umwandeln:  $\alpha(0)(0)\beta(0)(4)$ .

Das genaue Verfahren ist in [33, 34, 35] beschrieben.

Da die Bezeichner parametrisiert verglichen werden, erkennt dieses Verfahren somit nicht nur den Klontyp 1, sondern auch den Klontyp 2 und kann den gefundenen Klone den entsprechenden Typ zuordnen. Frühere Versionen von Dup konnten auch Klone des Typs 3 erkennen. Allerdings ist diese Funktionalität nicht mehr gewartet worden und konnte daher für den Vergleich nicht mehr aktiviert werden.

Der Ansatz ist invariant gegen das Einfügen von Leerzeichen, Leerzeilen und Kommentaren. Des Weiteren ist er weitgehend programmiersprachenunabhängig, da lediglich eine Regel für die Erkennung der Token notwendig ist.

Dup meldet Klonpaare, die aus je zwei Codefragmenten bestehen. Ein Codefragment ist ein Tupel (Dateiname, Startzeile, Endzeile).

Die zu Grunde gelegte Klonrelation ist paarweise symmetrisch. Dup meldet nicht alle transitiven Klonpaare ebenfalls als Klone, da die Codefragmente eines Klonpaares immer maximal sein müssen, d. h. für ein Codefragment ist die Startzeile so klein wie möglich und die Endzeile so groß wie möglich. Somit kann ein zu zwei anderen Klonpaaren transitives Klonpaar in Wirklichkeit noch größer als diese und daher nicht mehr transitiv zu ihnen sein. Hier wird nur das größere Klonpaar und nicht das kleinere, welches transitiv zu den anderen beiden ist, von Dup geliefert. Eventuell werden auch Klonpaare gemeldet, deren Codefragmente sich überlappen. Dies ist dann der Fall, wenn der eigentliche Klon eine kleinere Einheit eines Codefragments ist (z. B. tritt dies bei case-Blöcken innerhalb von switch-Anweisungen oder langen if-else-if-Kaskaden auf, wobei sich case-Blöcke oder then-Teile der if-Anweisung überlappen, indem sie sich in beiden Codefragmenten wiederfinden). Da Dup jedoch immer maximale Klone liefert, ist die Ausgabe das sich überlappende Klonpaar (siehe [13]).

Die Eingabe, die Dup erwartet, ist präprozessierter Quellcode. Anders ausgedrückt: Präprozessor-Direktiven werden von Dup ignoriert. Daher müssen sie bereits vorher bearbeitet worden sein. Somit werden weder Header-Dateien eingebunden, noch werden Makros ex-

## 2. Vorfeld

pandiert, wenn dies nicht explizit mit dem Aufruf eines Präprozessors geschieht (siehe [14]). Des Weiteren wird der komplette Code analysiert, den Dup zu sehen bekommt. Sind darin noch `#ifdef`-Direktiven enthalten, werden sie und der von ihnen umklammerte Code mit-analysiert.

Dup ist in C geschrieben und umfasst lediglich 4K SLOC (Source Lines Of Code). Momentan läuft es unter UNIX, ist aber problemlos auf anderen Architekturen lauffähig (siehe [12]).

### 2.4.2. Baxter

CloneDR<sup>TM</sup><sup>1</sup> von Ira D. Baxter hingegen basiert auf Sub-Tree-Matching im AST (Abstract Syntax Tree). Hierbei werden Teilbäume des AST miteinander auf Gleichheit bzw. Ähnlichkeit verglichen.

Zuerst wird aus dem zu untersuchenden Quelltext ein AST gebildet. Ein paarweiser Vergleich aller Sub-Trees wäre zu aufwendig. Daher werden die Sub-Trees, sofern sie eine gewisse Größe übersteigen, zuerst mittels einer Hash-Funktion in unterschiedliche Buckets verteilt. Diese Hash-Funktion muss allerdings so angelegt sein, dass Sub-Trees mit kleinen Änderungen in die gleichen Buckets gelangen und nur bei größeren Änderungen die Sub-Trees in unterschiedliche Buckets gehasht werden. Baxter wendet hier eine Hash-Funktion an, welche die Blätter der Sub-Trees nicht berücksichtigt.

Der nächste Schritt ist, alle Sub-Trees in einem Bucket miteinander auf Gleichheit bzw. Ähnlichkeit (siehe Abschnitt 3.4.2 auf Seite 30) zu vergleichen. In diesem Schritt wird nun eine Klon-Menge aufgebaut, in der sich alle Klone befinden. Wenn ein neuer Klon hinzugefügt wird, werden alle Sub-Trees dieses Klons aus der Menge entfernt. Dies hat den Sinn, dass nur der maximale Klon und nicht noch Teilklone von ihm gemeldet werden. AST-Knoten, die kommutative Operatoren darstellen, vergleichen ihre Kind-Knoten in allen kommutativen Reihenfolgen.

Mit bisheriger Methode werden keine Sequenzen von aufeinander folgenden Klonen erkannt. Daher ist ein weiterer Schritt nötig, der diese Sequenzen aufspürt. Aus diesem Grund wird zu jedem AST-Knoten, der Wurzel einer Sequenz sein kann, eine Liste verwaltet, die Hash-Werte seiner Kind-Knoten beinhaltet. Der Algorithmus vergleicht nun Listen gleicher Länge (d. h. gleicher Anzahl von Kind-Knoten), angefangen bei der kleinsten Länge bis hin zur größten Länge. Dabei werden die Sub-Trees gleicher Sequenzlänge wieder gehasht und in Buckets abgelegt. Sobald eine Klonsequenz gefunden ist, werden die Teilklone aus der Klonmenge entfernt.

Da nun weitere Sequenz-Klone in der Menge vorhanden sind, müssen noch einmal die Eltern-Knoten von bereits erkannten Klonen betrachtet werden. Sind diese nun ein Klon, werden sie der Klonmenge hinzugefügt und die Kinder gelöscht.

Eine genauere Beschreibung dieses Vorgangs kann man [36] entnehmen.

Baxters Ansatz kann Klone vom Typ 1 und Typ 2 erkennen, da er sowohl von den Bezeichnern abstrahieren, als auch Teilbäume erfassen kann, die „ähnlich genug“ sind. Der Klontyp wird dem Klon entsprechend zugeordnet. CloneDR<sup>TM</sup> erkennt Klone, bei denen ein

---

<sup>1</sup>Trademark von Semantic Designs, Inc., <http://www.semdesigns.com/>

Parameter leer ist, als Typ 2, wobei sich darüber streiten lässt, ob dies nicht einem Typ-3-Klon entspricht.

Da der Quellcode mittels eines Parsers in einen AST überführt wird, ist auch hier Invarianz gegenüber dem Einfügen von Leerzeichen, Leerzeilen und Kommentaren gegeben. Die Programmiersprachenunabhängigkeit jedoch ist schwieriger zu erreichen: Es muss ein Parser für die entsprechende Programmiersprache vorhanden sein.

CloneDR™ meldet Klonklassen, die aus zwei oder mehr Codefragmenten bestehen. Da CloneDR™ nicht zeilenorientiert arbeitet, sondern die einzelnen Teilbäume des AST genauer auf den Quellcode abbilden kann, ist es hier möglich, ein Codefragment als Tupel aus (Dateiname, Startzeile, Startspalte, Endzeile, Endspalte) anzugeben.

Die Klonrelation der einzelnen Codefragmente untereinander ist ebenfalls paarweise symmetrisch. CloneDR™ bildet immer die größte Klonklasse und verwirft kleinere Klonklassen, die komplett in größeren enthalten sind. Die Klonklassen sind garantiert nicht überlappend, d. h. ein Codefragment, welches in einer Klonklasse vorkommt, kommt in keiner weiteren vor. Ebenfalls nicht überlappend sind die Codefragmente innerhalb einer Klonklasse.

Die Eingabe für CloneDR™ kann aus präprozessiertem oder unprozessiertem Quellcode bestehen. Solange Makros „gutartig“ strukturiert sind, können sie wie normale Sprachsyntax geparsed werden. Sollten sich daraus Folgeprobleme ergeben, so wird das Makro expandiert oder im schlimmsten Fall eliminiert. Diese Probleme werden in einer Log-Datei abgelegt, so dass sie manuell betrachtet und eventuell nachbearbeitet werden können. Ein Nebeneffekt dieser Vorgehensweise ist, dass der komplette Source analysiert wird und nicht, wie bei einem AST-Ansatz zu vermuten wäre, nur die durch `#ifdef`-Direktiven ausgewählte Konfiguration nach dem Präprozessorlauf (siehe [16]).

Da für die zu analysierende Sprache ein Parser vorhanden sein muss, ist man bei der Analyse limitiert auf die vorhandenen Parser: Für die Programmiersprache C muss es ANSI C sein, Dialekte wie z. B. GNU C können nicht vollständig analysiert werden.

Zusätzlich verfügt CloneDR™ über die Möglichkeit, Klone zu visualisieren oder sie gar durch Funktionen oder Makros automatisch zu ersetzen. Dies erklärt auch, warum CloneDR™ keinerlei Überlappungen meldet: Es wäre sonst keine Ersetzung durch Funktionen oder Makros möglich.

CloneDR™ ist in einer eigenen, parallelen Programmiersprache PARLANSE geschrieben und nutzt mehrere CPUs aus, wenn vorhanden. Die Größe des gesamten Systems umfasst circa 230K SLOC und die verwendete Architektur ist Intel x86. Momentan läuft es ausschließlich unter Windows NT/2000, aber laut Baxter wäre eine Portierung nach GNU/Linux durchführbar (siehe [15]).

### 2.4.3. Kamiya

Toshihiro Kamiya verfolgt mit seinem Programm `CCfinder` auch den tokenbasierten Ansatz. Allerdings wird die Eingabe vor dem Vergleichen der Tokens auf verschiedene Arten transformiert. Diese Transformationen sind sprach- und anwendungsabhängig.

Der eigentliche Vorgang der Klonerkennung besteht aus drei Phasen:

Als erstes wird eine lexikalische Analyse entsprechend der Programmiersprache des zu untersuchenden Quelltextes durchgeführt. Die Tokens aller Dateien werden hintereinander

## 2. Vorfeld

gehängt, so dass eine Tokensequenz entsteht. Hierbei werden Token für Whitespace und Kommentare entfernt.

Der zweite Schritt besteht aus den Transformationen. Diese sind je nach Programmiersprache unterschiedlich. Die in diesem Experiment benutzten Transformationen werden in Abschnitt 3.4.3 auf Seite 31 aufgelistet. Nach diesen Token-Transformationen werden noch Typen, Variablen und Konstanten durch spezielle Tokens ersetzt, so dass z. B. Codefragmente mit unterschiedlichen Variablennamen auch als Klonpaare erkannt werden können.

Im letzten Schritt werden aus allen Sub-Strings der transformierten Tokensequenz mittels eines Suffix-Trees gleiche Paare ermittelt.

In [40] findet sich eine detailliertere Erklärung des Verfahrens.

Eine Klassifizierung der gefundenen Klone in die drei Klontypen findet bei `CCFinder` jedoch nicht statt. Laut Kamiya erkennt `CCFinder` die Klontypen 1 und 2 und teilweise auch den Typ 3, wenn die Veränderung klein genug ist.

Auch dieser tokenbasierte Ansatz ist invariant gegenüber Einfügen und Entfernen von Leerzeichen, Leerzeilen und Kommentaren.

Bei den von `CCFinder` gemeldeten Klonen handelt es sich um Klonpaare. Die zu Grunde liegenden Codefragmente werden durch das Tupel (Dateiname, Startzeile, Startspalte, Endzeile, Endspalte, Tokenanzahl) bestimmt. Die Tokenanzahl ist jedoch auf jeden Fall anders als bei anderen Werkzeugen, die mit Tokens arbeiten, da es sich bei Kamiya um die Tokens nach den Transformationen handelt. Die Codefragmente, aus welchen die zurückgelieferten Klonpaare bestehen, sind immer maximal.

Für Kamiyas Ansatz ist die Klonrelation eine Äquivalenzrelation. Es gilt bei ihm auch die Transitivität. Wenn also (A,B) und (B,C) zwei Klonpaare sind, so ist auch (A,C) ein Klonpaar, welches ebenfalls gemeldet wird.

Die Eingabe für `CCFinder` kann entweder präprozessierter oder unprozessierter Quellcode sein. Bei unprozessiertem Code werden Präprozessor-Direktiven ignoriert und Makros in nicht expandierter Form ohne Sonderbehandlung in Tokens gewandelt. Des Weiteren wird der gesamte Quellcode analysiert und nicht eine bestimmte Konfiguration, die durch `#ifdef`-Direktiven ausgewählt wurde (siehe [21]).

`CCFinder` ist in Microsoft Visual C++ Version 6 geschrieben und umfasst 15K SLOC. Das verwendete Betriebssystem ist Microsoft Windows 2000 in der japanischen Version (siehe [20]).

### 2.4.4. Krinke

Jens Krinke benutzt in seinem Werkzeug `Duplix` einen PDG (Program Dependence Graph) zum Identifizieren von Klonen (siehe dazu auch [38, 41]). Das zu untersuchende System wird als PDG dargestellt und `Duplix` identifiziert ähnliche Teilgraphen und gibt diese entsprechend als Klone aus.

Der PDG, den Krinke tatsächlich benutzt, ist ein erweiterter PDG, der sowohl Ähnlichkeiten mit einem AST und einem traditionellen PDG aufweist. Was Anweisungen und Ausdrücke betrifft, entsprechen die Knoten des PDG fast denen des AST. Variablen und Prozeduren haben darüber hinaus spezielle Knoten. Die Knoten können mit einer Klasse (z. B.

Anweisung, Ausdruck, Prozeduraufruf etc.), einem Operator (z. B. Binärausdruck, Konstante etc.), der die Klasse näher beschreibt, sowie einem Wert (z. B. „+“, „-“, Konstantenwerte, Variablennamen etc.), der den Operator näher spezifiziert, attribuiert sein.

Zusätzlich zu den Daten- und Kontrollabhängigkeiten hat der PDG, den Krinke benutzt, noch weitere Kanten: Wertabhängigkeiten (diese beschreiben den Datenfluss von berechneten Werten) und Referenzabhängigkeiten (ein berechneter Wert wird in einer Variablen gespeichert). Diese verschiedenen Arten von Kanten werden als Attribut der Kante dargestellt.

Sei  $V$  die Menge der Knoten und  $E$  die Menge der Kanten eines Graphen. Ein Pfad ist nun definiert als endliche Sequenz  $v_0, e_1, v_1, e_2, v_2, \dots, e_n, v_n$  mit  $v_i \in V$  und  $e_i \in E$ . Die Länge des Pfades sei  $n$ .

Das Ziel ist nun, ähnliche Teilgraphen zu erkennen. Zwei Graphen  $G$  und  $G'$  werden als ähnlich erkannt, wenn für jeden Pfad  $p = v_0, e_1, v_1, e_2, v_2, \dots, e_n, v_n$  im einen Graphen ein gleich langer Pfad  $p' = v'_0, e'_1, v'_1, e'_2, v'_2, \dots, e'_n, v'_n$  im anderen Graphen existiert und die Attribute von Knoten und Kanten entlang der zwei Pfade  $p$  und  $p'$  identisch sind. Eine weitere Bedingung ist, dass beide Graphen jeweils einen einzigen Startknoten ( $v_0$  und  $v'_0$ ) für alle Pfade haben.

Eine genaue Beschreibung des Vorgehens findet sich in [46].

Krinke betont, dass sein Ansatz nicht die typischen Klontypen 1, 2 oder 3 findet, sondern eigentlich einen weiteren Typ 4: „ähnlicher Code“. Da kein anderer Ansatz diesen Typ unterstützt und er auch als Typ 3 mit größeren Modifikationen gesehen werden kann, werden Kringes Klone im Folgenden alle als Typ-3-Klone betrachtet.

Durch das Aufbauen eines Graphen wird auch hier auf einer Ebene gearbeitet, die weit genug vom Quelltext entfernt ist, um Einfügen von Leerzeichen, Leerzeilen und Kommentaren als invariant zu betrachten.

Duplix meldet die Klone als Paare von Codefragmenten zurück. Intern ist eine Granularität von Knoten des Abhängigkeitsgraphen vorhanden. Diese werden wieder auf den Quellcode abgebildet. Daher ist es möglich, Codefragmente als Tupel von (Dateiname, Startzeile, Startspalte, Endzeile, Endspalte) zu betrachten.

Zu bemerken ist, dass Duplix in der Lage ist, Codefragmente mit Lücken zu melden, d. h. ein Klonpaar muss nicht notwendigerweise aus zwei Codefragmenten bestehen, die jeweils „am Stück“ sind, sondern diese Codefragmente können Lücken haben, welche nicht mit dem anderen Codefragment übereinstimmen. Dies hat z. B. Sinn, wenn ein Codefragment in der Entwicklung eines Systems an eine andere Stelle kopiert und in der Mitte eine große Menge von Code eingefügt wird. Duplix erkennt den gleichen Teil am Anfang und am Ende und die nicht kopierte Lücke in der Mitte.

Eine weitere Besonderheit in Kringes Ansatz ist, dass ein Codefragment nur in einem Klonpaar vorkommen kann, selbst wenn es an mehrere Stellen kopiert wurde. Es wird nur der beste Treffer gemeldet, andere mögliche Klonpaare eines Codefragments werden verworfen. Aus diesem Grund muss man sich nicht über Transitivität der Klonpaare Gedanken machen. Die Symmetrieeigenschaft gilt für die zu Grunde gelegte Klonrelation.

Die Eingabe für Duplix muss unprozessiert erfolgen, da Duplix selbst mit speziellen Header-Dateien ausgestattet ist, um somit die Analyse überhaupt durchführen zu können. Duplix ist nur in der Lage, in ANSI C geschriebene Programme zu analysieren. Außerdem

## 2. Vorfeld

werden Makros ausnahmslos expandiert, was zur Folge hat, dass Makros ab einer gewissen Größe und mit einer entsprechenden Struktur als Klone eingestuft werden (siehe [25]).

`DupliX` ist in C++ geschrieben und umfasst komplett 51K SLOC, wobei die Klonerkennung lediglich 1300 SLOC umfasst, der Rest ist Teil der Infrastruktur des Werkzeuges. Es läuft momentan unter GNU/Linux (siehe [24]).

### 2.4.5. Merlo

Der Ansatz, den Ettore Merlo in `CLAN` implementiert hat, beruht auf dem Vergleich von Metriken über Funktionen oder Codeblöcken. Hierbei werden die Metrikerwerte einer Reihe von verschiedenen Metriken für die Funktionen oder Codeblöcke berechnet und verglichen. Ergeben sich gleiche oder zumindest sehr ähnliche Werte, nimmt man an, dass der zu Grunde liegende Code identisch oder ähnlich ist.

Wendet man Metriken auf gleichen Code an, so ergeben sich folgerichtig die gleichen Metrikerwerte. Man hofft nun, dass bei geeigneter Wahl der Metriken auch der Umkehrschluss stimmt, dass also gleiche Metrikerwerte auf gleichen Code hindeuten. Merlo geht über die exakten Kopien hinaus und kann von ähnlichen Metrikerwerten auf ähnlichen Code schließen.

Intern wird der Quelltext von einem Parser zuerst in einen AST überführt. Dieser wird in eine interne Repräsentation übersetzt. Diese Repräsentation beinhaltet Kontrollfluss- und Datenflussinformationen. Die einzelnen Metriken verwenden nun die Knoten dieser internen Repräsentation und werten sie aus.

Der Quelltext wird je nach Analysemodus in Funktionen oder Codeblöcke aufgeteilt und diese werden dann paarweise miteinander verglichen: Die Werte verschiedener Metriken werden nacheinander berechnet. Je nach Abweichung der Metrikerwerte voneinander findet dann eine Klassifizierung in die Klontypen statt, oder es kann erkannt werden, dass kein Klon vorliegt. `CLAN` verfügt über eine Vielzahl unterschiedlicher Metriken. Diejenigen, welche im Experiment verwendet wurden, sind im Abschnitt 3.4.5 auf Seite 33 aufgelistet.

Der Ansatz der Metriken wird in [42, 43, 47, 48] ausgiebig erläutert.

Anhand der verschiedenen Metrikerwerte ist `CLAN` in der Lage, die gefundenen Klone in die Kategorien Typ 1, Typ 2 und Typ 3 einzuteilen.

Je nach Art und Kombination der angewendeten Metriken lassen sich verschiedene Arten von Klonen besser oder schlechter erkennen. Im folgenden Experiment wurde keine Metrik bezüglich des Quellcodelayouts verwendet, so dass auch hier Invarianz gegenüber Veränderung von Leerzeichen, Leerzeilen und Kommentaren vorliegt.

`CLAN` liefert entweder Klontypen (für die Klontypen 1 und 2) oder Paare (für alle Typen 1 – 3) von Codefragmenten zurück. Für die Programmiersprache C werden die Codefragmente durch das Tupel (Dateiname, Startzeile, Endzeile) bestimmt, für die Programmiersprache Java steht das Tupel (Dateiname, Startzeile, Startspalte, Endzeile, Endspalte, Tokenanzahl) zur Verfügung. Dies liegt an der Fähigkeit der zwei verwendeten, verschiedenen Parser. Die Tokenanzahl ist wiederum kein geeignetes Maß, da die Definition von Token bei den Teilnehmern nicht eindeutig ist.

Die zu Grunde liegende Klonrelation ist symmetrisch, aber nicht transitiv. Bei symmetrischen Paaren wird eines der beiden zufällig ausgewählt und ausgegeben.

Die Eingabe für CLAN kann entweder präprozessiert oder unprozessiert sein. Bei unprozessiertem Quellcode können Probleme beim Parsen auftreten, wenn Präprozessor-Direktiven an bestimmten Stellen auftreten oder wenn durch Makros die Anzahl der öffnenden oder schließenden Klammern beeinflusst wird.

Eine Besonderheit von Merlos Ansatz ist, dass der Klontyp 3 noch feiner untergliedert werden kann: Es können Untertypen bestimmt werden, je nach Art und Ausmaß der Modifikation des Klons.

Ebenfalls erwähnenswert ist die Tatsache, dass CLAN über eine Visualisierungsmöglichkeit verfügt: Gefundene Klonpaare können in zwei HTML-Dateien ausgegeben werden. Wenn man zwei Browserfenster parallel geöffnet hat und die Bildlaufleisten synchron bewegt, sieht man so die gefundenen Klonpaare gegenübergestellt und farblich hervorgehoben (siehe [27]).

CLAN wird unter GNU/Linux entwickelt und umfasst insgesamt etwa 41K SLOC. Hierbei umfasst der in Python geschriebene C-Parser etwa 4K SLOC, der Java-Parser 2K SLOC zusätzlichen Code (es wird ein frei verfügbarer Java-Parsergenerator [9] und eine frei verfügbare Java-Grammatik [5] benutzt), der Visualisierungsteil 19K SLOC und der eigentliche Klonerkennungsteil 16K SLOC (siehe [26]).

### 2.4.6. Rieger

Matthias Rieger verfolgt mit seinem Werkzeug Duploc einen zeilenorientierten, stringbasierten Ansatz, der mit Hilfe von Pattern-Matching auf visualisiertem Code arbeitet. Nach simplen Anfangstransformationen werden exakte Kopien in einem Diagramm markiert, mit dessen Hilfe Klone erkannt werden.

Um nahezu vollkommen unabhängig von der Programmiersprache des zu untersuchenden Quelltextes zu sein, werden in der Anfangstransformation lediglich Kommentare und Whitespace-Zeichen entfernt. Die Zeilenanzahl wird dabei aber nicht verändert. Diese Transformation ist derart trivial, dass sie für eine neue Programmiersprache innerhalb von Minuten implementiert werden kann, sofern diese andere Regeln für Kommentare oder erlaubte Whitespace-Zeichen besitzt.

Als nächstes wird eine Matrix angelegt, deren zwei Dimensionen dem Quelltext entsprechen, d. h. für die Suche nach Klonen innerhalb einer 1000 Zeilen langen Datei wird eine Matrix von  $1000 \times 1000$  benötigt. Die Auflösung der Matrix beträgt demnach eine Zeile. Nun wird jede Zeile mit jeder verglichen. Sind sie identisch, so wird in der Matrix an der entsprechenden Position ein Punkt gesetzt. Um diesen Algorithmus zu optimieren, werden die Zeilen vor dem paarweisen Vergleich mittels einer Hash-Funktion in verschiedene Buckets verteilt. Gleiche Zeilen landen zwingend in gleichen Buckets. Auf diese Weise kann man den Algorithmus um einen Faktor (die Anzahl der Buckets) beschleunigen.

In der Matrix gibt es auffällige Muster der Punkte. Diagonalen im Winkel von  $45^\circ$  deuten auf exakte Kopien hin. Unterbrochene Diagonalen ergeben sich, wenn Teile der Kopien verändert wurden. Sind die Diagonalen an einer Stelle vertikal oder horizontal versetzt, so wurde an der entsprechenden Stelle im Quelltext Text eingefügt oder entfernt. Ein weiteres interessantes Muster besteht aus regelmäßig wiederkehrenden kleineren Diagonalen, die in Rechtecken angeordnet sind. Dies ist z. B. typisch für `break`-Anweisungen innerhalb von `switch`-Blöcken. Ungewollte einzelne Punkte werden vor der weiteren Analyse entfernt.

## 2. Vorfeld

Mit einem entsprechenden Pattern-Matching-Algorithmus wird nun nach oben erwähnten Linienmustern gesucht.

Das genaue Vorgehen ist in [37] erklärt.

`Duploc` findet Klone vom Typ 1 und 2 sowie bestimmte Typ-3-Klone, sofern die Modifikation gering genug ist. Eine Kategorisierung der gefundenen Klone in die Klontypen ist momentan noch nicht implementiert.

Bei den gemeldeten Klonen handelt es sich um Paare von Codefragmenten. Da `Duploc` zeilenorientiert arbeitet, bestimmt das Tupel (Dateiname, Startzeile, Endzeile) ein Codefragment. Intern verwendet `Duploc` eine Darstellung als Klonklasse, weshalb auch bei einer Klonklasse mit  $N$  Codefragmenten  $(N - 1) \cdot N/2$  Klonpaare zurückgemeldet werden.

Die zurückgelieferten Codefragmente können wie bei Krinkes Programm (siehe Abschnitt 2.4.4 auf Seite 16) ebenfalls Lücken enthalten.

Die verwendete Klonrelation legt Symmetrie und Transitivität zu Grunde. Es handelt sich also um eine Äquivalenzrelation (Reflexivität ist ja generell sowieso vorhanden).

Die Eingabe für `Duploc` kann entweder präprozessiert oder unprozessiert erfolgen. Bei der Analyse von präprozessiertem Code ist es sogar möglich, die hereingezogenen Header-Dateien von der Analyse auszunehmen, und die Zeilennummern werden auf die Originalzeilennummern vor dem Präprozessorlauf abgebildet (siehe [30]).

`Duploc` ist mit VisualWorks Smalltalk Version 2.5 und 3.0 entwickelt worden und läuft unter Microsoft Windows, GNU/Linux (und anderen Unixen) und Apple MacOS. Es ist vom Autor unter der GNU General Public License veröffentlicht und erhältlich (siehe [31]).

## 2.5. Spezifikation von Kommunikationsformaten für Klone

Um über Klonpaare effizient kommunizieren zu können, muss ein einheitliches Format spezifiziert werden, das zum Beschreiben von Klonpaaren benutzt wird. Da insbesondere Baxter seine Klone nicht als Klonpaare, sondern als Klonklassen einsenden will, liegt es nahe, zwei Formate zu spezifizieren: eines für Klonpaare und eines für Klonklassen. Da die Datenbank intern zum Vergleich auf Klonpaaren arbeitet und das Auswertungsprogramm `clones` (siehe Abschnitt B.2.2 auf Seite 132) dieses Format auch zum Import der Daten benötigt, muss das Klonklassen-Format einfach in das Klonpaar-Format umgewandelt werden können. Dies geschieht mit Hilfe von `cconvert` (siehe Abschnitt B.2.1 auf Seite 131). Aus diesem Grund sind zwei recht simple ASCII-Dateiformate spezifiziert, die sich einfach einlesen und wieder ausgeben lassen.

Die beiden Formate sind in den Abschnitten C.1 auf Seite 135 und C.2 auf Seite 136 näher spezifiziert und durch jeweils ein Beispiel veranschaulicht.

Aus den Definitionen und Beispielen sollten die zwei Formate weitestgehend selbsterklärend sein. Einige Punkte bedürfen allerdings noch expliziter Erwähnung:

- Klone, deren Typ das Werkzeug nicht bestimmen kann, sollen als Klone vom Typ 0 gemeldet werden.
- Die Start- und Endzeilen sind beide als *inklusiv* zu betrachten. D. h., wenn die Startzeile  $n$  und die Endzeile  $m$  ist, so handelt es sich um  $m-n+1$  Codezeilen.

## 2.6. Datenbank-Schema zur Verwaltung der Klondaten

- Ein gültiger Dateiname ist als relativer Pfad zur entsprechenden Datei, startend im Väterverzeichnis des zu untersuchenden Projektes, zu betrachten (im Test-Projekt spule wäre z. B. `spule/src/server/ClientConnection.java` ein korrekter Dateiname). Die Projekte sind in den Abschnitten 3.1 auf Seite 27 und 3.3 auf Seite 29 aufgelistet.
- Da nur Klone im wirklichen Quellcode-Teil der Projekte berücksichtigt werden sollen, werden alle Klone, denen der Teilstring `/src/` im Dateinamen fehlt, verworfen. Alle Dateien der zu untersuchenden Projekte befinden sich in entsprechenden Unterverzeichnissen. Eventuell vorhandene andere Verzeichnisse enthalten Bibliotheken, die nicht mitanalysiert werden. Nur das Unterverzeichnis `/src/` ist jeweils von Interesse.
- Es gilt eine minimale Codefragmentgröße von sechs Zeilen Quellcode. Alle Klonpaare, die mindestens ein Codefragment enthalten, das kleiner als sechs Zeilen Quellcode ist, werden verworfen und somit nicht berücksichtigt, auch wenn das zweite Codefragment eventuell das Limit erfüllt.
- In diesem Experiment interessieren nur Klonpaare innerhalb eines Projekts. Klone über Projekte hinweg sollen nicht analysiert und berücksichtigt werden. Aus diesem Grund kann und soll zum Einsenden der Klone pro Projekt genau eine Datei verwendet werden, in der die gefundenen Klone im definierten Format aufgelistet sind.

Die komplette Spezifikation der zwei Formate samt Beispielen und obigen Anmerkungen wurden den Teilnehmern auf der eigens für dieses Experiment eingerichteten Homepage (siehe [18]) in Englisch zur Verfügung gestellt und zusätzlich per E-Mail auf der Mailing-Liste veröffentlicht.

## 2.6. Datenbank-Schema zur Verwaltung der Klondaten

Zum Verwalten und Auswerten der Klondaten der Teilnehmer wird eine Datenbank benötigt. Während der Planungsphase hat sich recht schnell herauskristallisiert, dass die Speicherung und Verwaltung der anfallenden Klondaten am besten mit Hilfe eines RDBMS (Relational DataBase Management System) zu bewerkstelligen ist. Da nur freie Implementierungen, die sowohl auf der Intel x86-Architektur als auch auf Sun Sparc laufen, in Frage kamen, wurde die Wahl eingeschränkt auf die zwei Systeme MySQL und PostgreSQL. Beim Experimentieren mit MySQL waren dessen Grenzen, was die Unterstützung von verschiedenen Konstrukten der SQL (Structured Query Language) betrifft (insbesondere verschachtelte SELECT-Anweisungen), recht schnell erreicht. Da PostgreSQL diese SQL-Konstrukte beherrscht und zusätzlich Erfahrungen mit diesem RDBMS aus meiner Studienarbeit her vorhanden waren, fiel die Entscheidung recht schnell auf das zwar langsamere, aber an Funktionalität reichere PostgreSQL.

Anfangs war die Verwaltung der Klone mittels eines Datenbank-Schemas in reiner 3. Normalform vorgesehen. Ein vereinfachtes Entity-Relationship-Diagramm dieser Datenbank ist in Abbildung 2.2 auf der nächsten Seite zu sehen.

## 2. Vorfeld

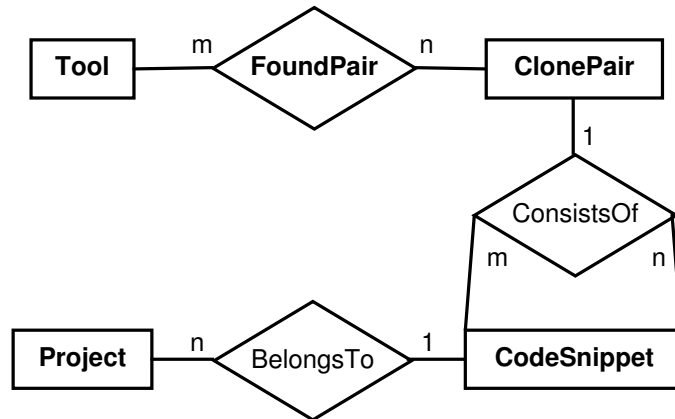


Abbildung 2.2: Vereinfachtes Entity-Relationship-Diagramm der ursprünglichen Datenbank

Die Tabellen enthielten im Einzelnen folgende Attribute:

- Tool(No, Name, Authors, Voluntary, Active)  
In dieser Tabelle wurden Name des Werkzeuges sowie die Namen der Autoren gespeichert. Dies diente zur Identifizierung der Klonpaare im Experiment. Weiterhin waren zwei Flags vorhanden, die angaben, ob es sich um die „Kür“-Teilnahme (siehe Abschnitt 3.4 auf Seite 30) eines Werkzeuges handelt und ob das Werkzeug überhaupt ausgewertet werden soll.
- Project(No, Name, Active)  
Diese Tabelle speicherte die Namen der zu analysierenden Projekte und in einem Flag den Status, ob das Projekt überhaupt ausgewertet werden soll.
- CodeSnippet(No, Project\_No, FileName, FromLine, ToLine)  
In dieser Tabelle wurden die Codefragmente, wie sie spezifiziert worden sind, abgelegt. Zusätzlich wurde noch auf das Projekt referenziert. Dies wäre zwar über den Dateinamen möglich gewesen, aber dann nicht von dem RDBMS selbst zu erledigen.
- ClonePair(No, CodeSnippet\_No1, CodeSnippet\_No2, isReference)  
Einem Klonpaar wurden zwei Codefragmente zugeordnet und der Status, ob es sich dabei um ein gültiges, korrektes Klonpaar (im Folgenden Referenz genannt) handelt oder nicht.
- FoundPair(Tool\_No, ClonePair\_No, Type)  
Alle von den Teilnehmern gefundenen Klonpaare (im Folgenden Kandidaten genannt) wurden zusammen mit dem Klontyp, den das jeweilige Werkzeug dem Klonpaar zugeordnet hat, in dieser Tabelle abgelegt.

Die Gleichheit eines Klonpaares war über die Tabelle ClonePair definiert. Dies wiederum setzte voraus, dass beide Codefragmente des Klonpaares exakt identisch sein mussten. Wie

sich jedoch beim ersten Test-Experiment herausgestellt hat, traf dies auf so gut wie keine Klonpaare zu.

Nach näheren Untersuchungen der Umstände stellte sich heraus, dass dieser Ansatz nicht praxistauglich ist. So kam es z. B. relativ häufig vor, dass ein Werkzeug das Codefragment eines Klonpaares eine Zeile früher enden oder beginnen lies als ein anderes Werkzeug. Häufig war dies bei öffnenden und schließenden geschweiften Klammern der Fall. Abhilfe sollte das Normieren des Quellcodes (siehe Abschnitt 3.2 auf Seite 28) schaffen.

Allerdings wurde dennoch klar, dass es fragwürdig ist, die Gleichheit von Klonpaaren über die exakte Gleichheit der zwei Codefragmente zu definieren.

Aus diesem Grund wurde immer deutlicher, dass ein Ansatz, die Überdeckung von Kandidaten und Referenzen als Maß für die Gleichheit zweier Klonpaare zu nehmen, der bessere Weg ist. Diese Vorgehensweise wurde bereits in [45] in einem anderen Zusammenhang diskutiert, lässt sich aber problemlos auf den Vergleich von Klonpaaren anwenden.

Ein großes Problem bei der Umstellung auf diese Vorgehensweise stellte die Geschwindigkeit von PostgreSQL bei obigem Datenbank-Schema dar: Die Abfragen für die Berechnung der notwendigen Überdeckungen stellten sich als zu zeitaufwändig heraus, da zu oft das Kartesische Produkt von Tabellen gebildet werden musste. Als Konsequenz daraus wurde fortan auf die Redundanzfreiheit der 3. Normalform verzichtet und das Schema umgestellt. In Abbildung 2.3 ist ein vereinfachtes Entity-Relationship-Diagramm des neuen Schemas dargestellt.

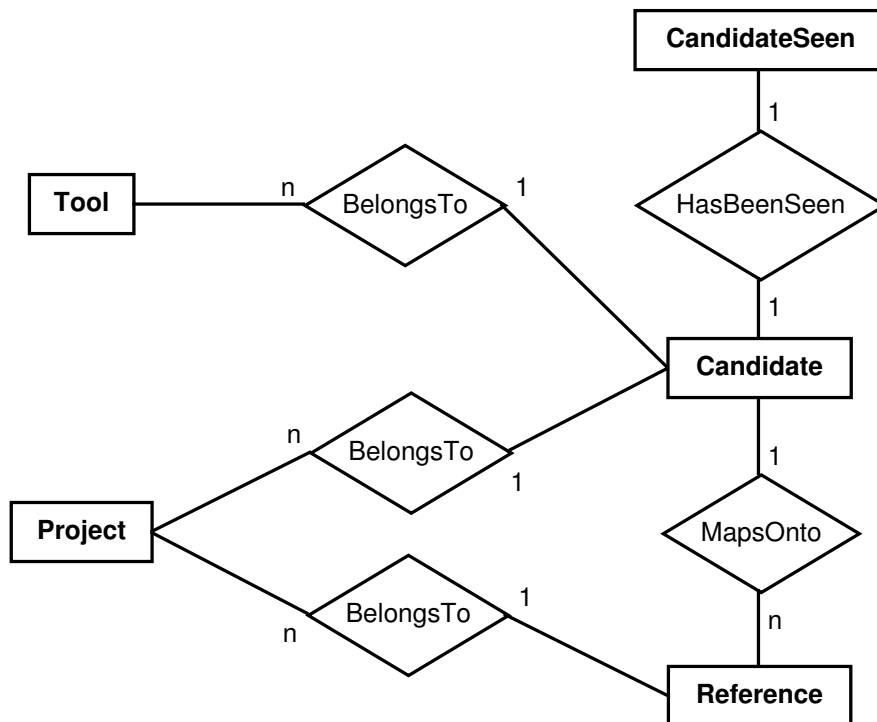


Abbildung 2.3: Vereinfachtes Entity-Relationship-Diagramm der neuen Datenbank

## 2. Vorfeld

Die Tabellen enthalten im Einzelnen folgende Attribute:

- Tool(No, Name, Authors, Voluntary, Active)  
Diese Tabelle ist identisch mit der aus dem alten Schema.
- Project(No, Name, Active, Lang, Size)  
Diese Tabelle ist bis auf die Attribute Lang und Size, die jeweils Programmiersprache und Größe der Projekte angeben, identisch mit der aus dem alten Schema.
- Reference(No, Project\_No, FileName1, FromLine1, ToLine1, FileName2, FromLine2, ToLine2, Type)  
Diese Tabelle beinhaltet die Referenzen. Hierbei werden die zwei Codefragmente, der Verweis auf das Projekt und der Klontyp der Referenz gespeichert.
- Candidate(No, Tool\_No, Project\_No, FileName1, FromLine1, ToLine1, FileName2, FromLine2, ToLine2, Type, Reference\_No, OK, Good)  
Alle von den Teilnehmern vorgeschlagenen Kandidaten samt Verweis auf den Teilnehmer und das Projekt werden in dieser Tabelle abgelegt. Die Felder Reference\_No, OK und Good werden erst zur Auswertung benötigt. Auf sie soll später im Abschnitt 4.2.4 auf Seite 48 eingegangen werden.
- SeenCandidate(No)  
Diese Tabelle verwaltet die Nummern der Kandidaten, die vom Schiedsrichter bereits betrachtet und bewertet worden sind. Dies wird im Abschnitt 4.1 auf Seite 39 näher erläutert. Anstelle dieser Tabelle wäre auch ein Boolesches Feld in der Tabelle Candidate möglich. Da aber die verwendete Version von PostgreSQL einen Fehler hat, der Tabellen mit Booleschen Feldern mit der Zeit immer langsamer werden lässt, ist die Lösung mit dieser weiteren Tabelle aus Geschwindigkeitsgründen gewählt.

## 2.7. Festlegung der Programmiersprachen im Experiment

Wie bereits in der Aufgabenstellung (siehe Abschnitt 1.2 auf Seite 7) erwähnt, sollen Projekte, die in den Programmiersprachen Java und C geschrieben sind, auf Klone analysiert werden. Diese Wahl liegt darin begründet, dass anhand der anfallenden Daten eventuell auch Aussagen über die Klonhäufigkeit in unterschiedlichen Programmiersprachen gemacht werden können. Insbesondere ist von Interesse, ob eine Programmiersprache, die objektorientiertes Programmieren unterstützt, weniger anfällig für das Klonen ist als eine Programmiersprache, die lediglich prozedurale Programmierung unterstützt.

Für Java müssen keine weiteren Spezifikationen mehr getroffen werden, da alle Teilnehmer (bis auf Krinke, der kein Java analysieren kann) mit Java voraussichtlich keine Probleme haben werden. Auch müssen Pakete, von denen die Projekte abhängen und die mit `import` eingebunden werden, für die einzelnen Teilnehmer nicht mitgeliefert werden.

Da es verschiedene C-Dialekte gibt und üblicherweise der Quellcode zuerst von einem Präprozessor bearbeitet wird und erst danach der Code entsteht, den der Compiler selbst sieht, müssen hier noch einige Dinge spezifiziert werden:

## 2.7. Festlegung der Programmiersprachen im Experiment

- `#include`  
Wenn das Werkzeug den Inhalt der zu inkludierenden Dateien benötigt, darf es die Dateien inkludieren. Die Analyse der Klone soll jedoch auf das Hauptsystem beschränkt bleiben. Klone in den inkludierten Dateien werden nicht berücksichtigt. Außerdem müssen die Original-Zeilenummern beibehalten werden.
- `#ifdef, #ifndef, ...`  
Hier unterscheiden sich die Werkzeuge: Textbasierte Verfahren werden den kompletten Quelltext analysieren, ebenso Baxters CloneDR™, welches einen AST aufbaut. Wird zuerst ein Präprozessor über den Quellcode laufen gelassen, so wie es z. B. bei Krinke der Fall ist, so wird nur der Zweig der aktuellen Konfiguration analysiert. Aus diesem Grund ergeben sich hier leichte Unterschiede bezüglich des Codes, den die einzelnen Werkzeuge als Grundlage zur Klonanalyse hernehmen. Dies lässt sich aber nicht verhindern, da auf der anderen Seite kein präprozessierter Code als Ausgangsbasis genommen werden kann, weil Krinke unbedingt selber präprozessieren muss.
- Makro-Behandlung  
Laut einhelliger Meinung auf der Mailing-Liste, sollte die Klonanalyse – soweit möglich – den Quellcode in der Art und Weise betrachten, in der der Programmierer den Code beim Klonen auch sieht. Das bedeutet, dass Makros möglichst nicht expandiert werden sollen. Lediglich wenn das Werkzeug den Quelltext parsen muss und zur Auflösung von Konflikten das Makro expandieren muss, ist dies erlaubt. Inkonsistenzen, was z. B. die Zeilennummerierung betrifft, treten nicht auf, da Makros immer in einer Zeile expandiert werden: die Anzahl der Zeilen ändert sich durch Makro-Expansion nicht.
- Inline-Assembler und Präprozessor-Direktiven, die nicht in Abschnitt B.1.2 auf Seite 130 aufgelistet sind  
Das Werkzeug `codenormalize`, welches den Quellcode der Projekte aufbereitet, bevor sie den Teilnehmern ausgehändigt werden, stellt sicher, dass keine Präprozessor-Direktiven in den Dateien vorhanden sind, die nicht im ANSI C Standard spezifiziert sind. Ebenso wird garantiert, dass keine GNU-spezifischen Inline-Assembler-Anweisungen in den Projekten vorkommen.
- Präprozessor-Direktiven, die in Abschnitt B.1.2 auf Seite 130 aufgelistet sind  
Dies sind die im ANSI C Standard spezifizierten Direktiven, welche die Werkzeuge der Teilnehmer laut eigenen Aussagen verstehen und entweder ignorieren oder korrekt zu interpretieren wissen. Diese werden daher nicht aus den Projekt-Quelldateien entfernt.

Diese Punkte wurden auf der Mailing-Liste diskutiert und von den einzelnen Teilnehmern akzeptiert.

## 2. Vorfeld

## 3. Durchführung

YODA: Begun, this clone war has.  
(*Star Wars II – Attack of the Clones*)

In diesem Kapitel werden die Projekte der Test- und der Haupt-Phase vorgestellt. Erkenntnisse, die im Laufe der Test-Phase gemacht wurden, und die Änderungen, die aus den Erkenntnissen resultieren, werden beschrieben. Des Weiteren wird auf die jeweilige Konfiguration der Werkzeuge im Experiment eingegangen. Abschließend folgt eine Zusammenfassung der Probleme, welche die Teilnehmer beim Analysieren der Projekte zu bewältigen hatten.

### 3.1. Test-Phase

Nachdem mit den einzelnen Teilnehmern die Definitionen und Spezifikationen des Experiments und seines Verlaufs abgeklärt worden waren, wurden vier Projekte als Test-Systeme ausgewählt, um zu testen, wie praktikabel die Durchführung des Experiments nach den festgesetzten Regeln ist. Für die Programmiersprache C wurden die Projekte bison und wget sowie für die Programmiersprache Java die Projekte EIRC und spule ausgewählt. Die Größen der einzelnen Systeme sind in Abbildung 3.1 aufgeführt. Sie sind relativ klein gewählt, da sie nur dazu dienen, den generellen Ablauf zu testen. Auch eine Auswertung soll nur zum Testen des Verfahrens und nicht der Ergebnisse wegen durchgeführt werden. Einige zusätzliche Informationen zu den Projekten sind in Abschnitt D.1 auf Seite 145 vorhanden.

Projekt	Sprache	Projektgröße
bison	C	19K SLOC
wget	C	16K SLOC
EIRC	Java	8K SLOC
spule	Java	10K SLOC

Abbildung 3.1: Projekte der Test-Phase

## 3.2. Erkenntnisse und Änderungen

Wie bereits im Abschnitt 2.6 auf Seite 21 erwähnt wurde, waren die Ergebnisse des ersten Test-Experiments enttäuschend: Aufgrund der Organisation der Datenbank und der Definition der Gleichheit von Klonpaaren konnten keine interessanten Aussagen getroffen werden.

Abhilfe sollte eine Normierung des Codes bringen. Hierbei wird mit dem Hilfsprogramm `codenormalize` (siehe Abschnitt B.1.2 auf Seite 130) der Quellcode transformiert. Zeilen, die lediglich öffnende oder schließende geschweifte Klammern enthalten, werden gelöscht und die darin enthaltene(n) Klammer(n) werden an die vorige Zeile angehängt. Hierbei müssen Kommentare (für C und Java), mehrzeilige Makrodefinitionen (für C) sowie mehrzeilige String-Konstanten (für C) berücksichtigt werden. Des Weiteren werden Leerzeilen entfernt. Glücklicherweise ist die Syntax von C und Java ähnlich genug, dass dies ein Werkzeug für beide Sprachen erledigen kann.

In Abbildung 3.2 ist ein Beispiel zu sehen, wie Quellcode von der Original-Version in die normierte Form transformiert wird.

Original:	Normiert:
<pre>while (c1) {     if (c2)     {         yes();     }     else     {         no();     } }</pre>	<pre>while (c1) {     if (c2) {         yes(); }     else {         no(); } }</pre>

Abbildung 3.2: Code vor der Normierung und danach

Da kein Werkzeug Layout-Informationen benutzt, insbesondere Merlo sein `CLAN` im Experiment ohne Layout-Metriken analysieren lässt, stellt dies keinen allzu großen Eingriff in die Codestruktur dar. Werkzeuge, die zeilenweise auf dem Text arbeiten, sollten hier auch keine Probleme bekommen, da bei einem potentiellen Klon eines Codefragments ja die gleiche Normierung vorgenommen wird, sofern er vorher gleich aussah. Es gibt jedoch die Möglichkeit, dass an einer Stelle bereits vom Autor normierter Quellcode vorliegt, an einer anderen nicht. Die Normierung macht diese beiden Codefragmente nun identisch (sofern der eigentliche Code ein Klon ist). Hier finden zeilenbasierte Werkzeuge mit der Normierung mehr Klone als ohne. Allerdings wird zum einen C-Code in der Praxis wohl nie so formatiert, wie es das Resultat der Normierung ergibt, so dass dieser Fall wohl nicht auftritt. Zum anderen wäre es alternativ möglich gewesen, einen Pretty Printer über den Quellcode der Projekte laufen zu lassen. Dies wird häufig praktiziert und liefert diesen Aspekt betreffend das gleiche Resultat. Aus diesen genannten Gründen ist das Verwenden der Nor-

mierung nicht als irreguläre Beeinflussung der Ergebnisse zu Gunsten bestimmter Werkzeuge zu betrachten. Für Ansätze, die den Code zuerst parsen, macht diese Normierung auch keinen Unterschied aus. Daher kann sie durchgeführt werden, ohne das Ergebnis des Experiments zu verfälschen.

Eine weitere wichtige Erkenntnis der ersten Test-Runde ist, dass im Haupt-Experiment auf Quellcode aus GNU-Projekten verzichtet werden sollte. Diese enthalten zu viele Sprachkonstrukte, die nicht ANSI C, sondern GNU C sind und somit von einigen Teilnehmern (Baxter und Krinke) nicht oder nur mit hohem Aufwand und Modifikationen an den Projekten analysiert werden können. Ursprünglich war das Referenzsystem der einzelnen Projekte GNU/Linux, d. h. `configure` wurde auf GNU/Linux ausgeführt und dies war somit die Konfiguration für die bedingte Kompilierung in den Quelldateien. Aufgrund der Erkenntnisse über die Probleme der GNU-Spezifika ist für die Haupt-Phase Sun Sparc mit dem Betriebssystem Solaris als Referenzplattform gewählt worden. Der Sun C Compiler verfügt unter anderem über die Option `-xc`, die beim Kompilieren auf strikte Kompatibilität mit dem ANSI C Standard testet. Somit kann für die Projekte der Haupt-Phase, die in der Programmiersprache C geschrieben sind, sichergestellt werden, dass sie von den Teilnehmern parsbar sind.

### 3.3. Haupt-Phase

Für die Haupt-Phase wurden insgesamt acht Systeme ausgewählt. Vier davon sind in der Programmiersprache C geschrieben, die restlichen vier in Java. Die Größen der Systeme wurden so gewählt, dass sie beginnend von 11K SLOC bis zu 235K SLOC reichen und jeweils den Bereich dazwischen abdecken. Unterschiede in der Systemgröße wie auch Unterschiede zwischen den zwei Programmiersprachen sollen so ebenfalls auffindig gemacht werden können.

Die einzelnen Systeme und ihre jeweilige Größe in SLOC nach der Normierung (siehe Abschnitt 3.2 auf der vorherigen Seite) sind in Abbildung 3.3 ersichtlich. Einige zusätzliche Informationen zu den Projekten sind in Abschnitt D.2 auf Seite 146 vorhanden.

Projekt	Sprache	Projektgröße
weltab	C	11K SLOC
cook	C	80K SLOC
sns	C	115K SLOC
postgresql	C	235K SLOC
netbeans-javadoc	Java	19K SLOC
eclipse-ant	Java	35K SLOC
eclipse-jdtcore	Java	148K SLOC
j2sdk1.4.0-javax-swing	Java	204K SLOC

Abbildung 3.3: Projekte der Haupt-Phase

Der Zeitraum, der den Teilnehmern zur Analyse der Projekte zur Verfügung stand, wurde auf vier Wochen festgesetzt und auf der Homepage (siehe [18]) veröffentlicht.

### 3. Durchführung

Nach den zwei Test-Läufen im Vorfeld sollten eigentlich keine Probleme in der Haupt-Phase mehr auftreten. Die Projekte sollten für alle Teilnehmer parsbar sein und analysiert werden können. Sollten dennoch wider Erwarten Probleme auftreten, so sollten die Teilnehmer versuchen, diese mit minimalen Änderungen zu beheben und zusammen mit den Analyse-Ergebnissen eine genaue Erklärung samt Begründung der Änderungen einschicken.

## 3.4. Konfiguration der Werkzeuge

Einige der Werkzeuge können anhand von Parametern in ihren Analysefähigkeiten variiert werden. So ist es möglich, unterschiedliche Schwellwerte zu setzen, unterschiedliche Metriken anzuwenden etc. Damit solche Modifikationsmöglichkeiten, die das Werkzeug einem bietet, entsprechend bewertet werden können, darf jeder Teilnehmer zwei Einsendungen pro Projekt tätigen: Die erste ist der Pflicht-Teil und muss mit denselben Einstellungen des Werkzeuges gemacht werden, mit denen die Test-Phase absolviert wurde. Der zweite „Kür“-Teil ist freiwillig und kann mit veränderten Einstellungen des Werkzeuges geschehen. Hier können die Teilnehmer die Qualitäten ihrer Werkzeuge „tunen“ und somit das Ergebnis aus dem Pflicht-Teil verbessern.

Von dieser Möglichkeit haben allerdings nur zwei Teilnehmer Gebrauch gemacht: Kamiya und Merlo. In der Analyse der Ergebnisse in Kapitel 5 auf Seite 51 sind diese „Kür“-Einsendungen mit dem Kürzel „(vol.)“ für englisch „voluntary“ versehen. Alle anderen Daten beziehen sich auf die Pflicht-Einsendungen.

Im Folgenden soll nun kurz auf die Einstellungen der jeweiligen Werkzeuge eingegangen werden.

### 3.4.1. Baker

Die einstellbare Größe bei Bakers `Dup` ist die minimale Codefragment-Größe der Klonpaare. Diese ist per Definition des Experiments auf sechs Zeilen eingestellt (siehe [11]). Baker wies darauf hin, dass die Anzahl der „False Positives“ bei zu kleinen Werten stark ansteigt und dass sie normalerweise Werte von 15 aufwärts als sinnvoll erachtet. Zum Vergleichen mit den anderen Werkzeugen wurde jedoch trotzdem der Wert sechs gewählt, da sonst kleinere Klonpaare, welche andere Werkzeuge finden, nicht gefunden würden.

### 3.4.2. Baxter

Bei Baxters `CloneDR™` lassen sich die folgenden Parameter einstellen: Anzahl der Prozessoren, auf denen die Auswertung läuft (zwei im Experiment), Starthöhe der Teilbäume bei der Analyse im AST (eins im Experiment, d. h. auf Ebene der Blätter; zwei würde eine Ebene oberhalb der Blätter bedeuten, usw.), Anteil der notwendigen Ähnlichkeit von Teilbäumen (90 % im Experiment), Anzahl der möglichen Parameter, die bei einem Typ-2-Klon variiert werden können (sechs im Experiment) und minimale Anzahl der AST-Knoten (im Experiment 15 für Java und 18 für C, da die durchschnittliche Anzahl von Knoten pro Zeile

für die Sprachen unterschiedlich ist). Diese Daten sind in [17] nachzulesen. Die Ähnlichkeit von Teilbäumen ist wie folgt definiert:

$$\text{Similarity}(T_1, T_2) = \frac{2 \cdot \text{Shared}(T_1, T_2)}{2 \cdot \text{Shared}(T_1, T_2) + \text{OnlyIn}(T_1) + \text{OnlyIn}(T_2)}$$

$T_1$  und  $T_2$  bezeichne hierbei die zwei zu vergleichenden Teilbäume des AST und Similarity die gewünschte Größe für die Ähnlichkeit. Shared ist die Anzahl der AST-Knoten, die in beiden Teilbäumen vorkommt, OnlyIn dementsprechend die Anzahl der Knoten, die nur in einem Teilbaum, nicht aber im anderen, vorkommt (siehe [36]).

### 3.4.3. Kamiya

Kamiyas `CCFinder` wurde im Experiment mit folgender Konfiguration benutzt:

Die Suche von Klonpaaren geschieht sowohl innerhalb von Dateien als auch dateiübergreifend.

Für die Programmiersprache C werden folgende Transformationen durchgeführt (teilweise sind die Transformationen nutzlos, da sie sich auf C++ beziehen und daher bei C nicht greifen; `CCFinder` ist allerdings tatsächlich mit diesen Einstellungen gelaufen, aus diesem Grund müssen sie der Korrektheit wegen auch aufgeführt werden):

- neglect name space or package tokens  
(e.g. `std::cin` → `cin`).
- complement block for single statement after  
`for()`, `do`, `else`, `if()`, `while()`.
- parameterize numerical/boolean tokens  
(e.g. `100` → `$param`, `true` → `$param`).
- parameterize function-name tokens  
(e.g. `'foo'` `'('` → `'$param'` `'('`).
- parameterize table initializations  
(e.g. `= { 1, 2 }` → `= { $param }`).
- parameterize type keywords  
(e.g. `int` → `$param`).
- parameterize member-name tokens  
(e.g. `.foo` → `.$param`, `'->' 'foo'` → `'->' '$param'`).
- parameterize the other names.
- insert end-of-definition at next of each top level `'}'` and  `';'` .
- parameterize string literals  
(e.g. `"abc"` → `$param`, `'\t'` → `$param`).
- parameterize template arguments  
(e.g. `foo<int, char>` → `foo`).

### 3. Durchführung

- neglect visibility keywords  
(friend, private, protected, and public).

Für die Programmiersprache Java sind andere Transformationen nötig. Hierbei handelt es sich um folgende Transformationen:

- neglect name space or package tokens  
(e.g. `java.lang.Math` → `Math`).
- complement block for single statement after  
`for()`, `do`, `else`, `if()`, `while()`.
- complement a callee token.
- parameterize numerical/boolean tokens  
(e.g. `100` → `$param`, `true` → `$param`).
- parameterize method-name tokens  
(e.g. `'foo'` `'('` → `'$param'` `'('`).
- parameterize table initializations  
(e.g. `= { 1, 2 }` → `= { $tableinit }`).
- parameterize type keywords  
(e.g. `int` → `$param`).
- parameterize member-name tokens  
(e.g. `.foo` → `.$param`, `'->'` `'foo'` → `'->'` `'$param'`).
- parameterize the other names.
- insert end-of-definition at next of each top level `'}'` and  `';'` .
- neglect interface.
- parameterize string literals  
(e.g. `"abc"` → `$param`, `'\t'` → `$param`).
- neglect visibility keywords  
(private, protected, public, and final).

Alle diese Transformationen wurden von Kamiya in [22] erläutert und hier im Englischen belassen, damit durch eine Übersetzung keine Ungenauigkeiten entstehen.

Des Weiteren kann die minimale Anzahl transformierter Token eingestellt werden, und für das Experiment wurde ein Wert von 30 gewählt.

Obige Einstellungen gelten für den Pflicht-Teil. Im „Kür“-Teil hat Kamiya sein `CCFinder` intern umprogrammiert, so dass manche der obigen Transformationen anders arbeiten und weniger uninteressante Klone liefern. Hierzu liegen allerdings keine näheren Informationen vor.

#### 3.4.4. Krinke

Bei Duplix von Krinke lässt sich nur ein Wert ohne Änderung des Quellcodes modifizieren: Die maximale Pfadlänge im PDG (siehe Abschnitt 2.4.4 auf Seite 16). Diese ist im Experiment auf 20 eingestellt. Dies ist gleichzeitig auch der Wert, der in [46] durch Evaluation empfohlen wird. Eine Gewichtung der Knoten des Graphen spielt im Experiment keine Rolle, da dieser Teil für das Experiment umgeschrieben wurde und nur das Sechs-Zeilen-Kriterium relevant ist (siehe auch [23]).

#### 3.4.5. Merlo

Bei Merlos CLAN lassen sich verschiedene Metriken ein- bzw. ausschalten. Jede Metrik kann zusätzlich noch einen Schwellwert zugewiesen bekommen, der die maximale Abweichung des Metrikwertes angibt, bei dem der Klonkandidat noch als Klon anerkannt wird.

Im Experiment wurden ausschließlich die folgenden Metriken verwendet (siehe [29]):

- CALLS  
Anzahl der Funktionsaufrufe.
- LOCALS  
Anzahl der lokalen Variablen.
- NBRANCHES  
Anzahl der verzweigenden Konstrukte.
- NLOOPS  
Anzahl der Schleifen-Konstrukte.
- NONLCALS  
Anzahl nicht lokaler Variablen.
- PARNUM  
Anzahl der Parameter bei Funktionen (darf nicht benutzt werden, wenn man Block-Analysen durchführt).
- STMNT  
Anzahl der Anweisungen, bzw. der Knoten im AST nach dem Parsen.

Für den Pflicht-Teil des Experiments wurden die Metriken, wie sie in Abbildung 3.4 auf der nächsten Seite aufgelistet sind, benutzt. Die Schwellwerte waren dabei jeweils auf 0 gesetzt. Es wurde eine Block-Analyse durchgeführt, d. h. es wurden nicht nur Funktionen paarweise auf Ähnlichkeit untersucht, sondern auch Blöcke von Anweisungen.

Für den „Kür“-Teil wurden zu den Ergebnissen des Pflicht-Teils noch weitere Klonkandidaten hinzugefügt. Diese wurden durch eine Funktionen-Analyse (d. h. nur komplette Funktionen werden auf Ähnlichkeit überprüft) gefunden. Dabei wurde die Metrik PARNUM hinzugenommen. Für das Project cook wurden die gleichen Daten wie im Pflicht-Teil verwendet, für weltab wurde der Schwellwert der ersten Metrik (CALLS) auf 10 gesetzt, die Schwellwerte der anderen Metriken auf 2, für alle anderen Projekte wurde der Schwellwert

### 3. Durchführung

Projekt	Metriken
alle C-Projekte	CALLS LOCALS NONLCALS STMNT NBRANCHES NLOOPS
alle Java-Projekte	STMNT LOCALS NBRANCHES NLOOPS CALLS

Abbildung 3.4: Metriken im Pflicht-Teil

der ersten Metrik (CALLS für C-Projekte, STMNT für Java-Projekte) auf 10 gesetzt, alle weiteren Schwellwerte auf 0 (siehe dazu [28]).

Fälschlicherweise wurde bei den Projekten der Programmiersprache C in der „Kür“ vergessen, die Metrik NLOOPS mit in den Vergleich einzubeziehen. Dies wurde von Merlo erst lange nach dem Test bemerkt. Eventuell ließe sich bei den Projekten der Programmiersprache C noch ein besseres Ergebnis erzielen, wenn man diese Metrik ebenfalls berücksichtigt.

#### 3.4.6. Rieger

Es sind leider keine Informationen über Parameter des Werkzeuges `Duploc` bekannt.

### 3.5. Probleme bei der Durchführung

Trotz der Test-Phase des Experiments sind bei einigen Teilnehmern Probleme in der Haupt-Phase aufgetreten. In den nächsten Abschnitten sollen diese Fälle für jeden einzelnen Teilnehmer beleuchtet werden.

#### 3.5.1. Baker

Baker meldete keinerlei Probleme mit den acht Projekten.

#### 3.5.2. Baxter

Baxter bemerkte generell, dass die durchgeführten Tests auf Kompatibilität mit dem ANSI C Standard nicht ausreichend gewesen sind. Wenn man mit den mitgelieferten `#defines` eine spezielle Konfiguration der Systeme auswählt, so ist diese ANSI C kompatibel. Da aber für sein `CloneDRTM` z. B. alle Arme dieser Präprozessor-Direktiven syntaktisch korrekt sein müssen, stellen somit inaktive Arme, welche K&R-Stil enthalten, ein Problem dar. Baxter hat aus diesem Grund eine Liste von `#defines` erstellt, mit denen er diese für ihn

„böartigen“ Konstrukte beseitigt. Aus Platzgründen kann hier nicht die vollständige Liste abgebildet werden, aber einige dieser #defines sind in Abbildung 3.5 aufgelistet.

Doch auch das zusätzliche Einfügen dieser #defines hat nicht ausgereicht, um die Projekte cook und postgresql für Baxter parsbar zu machen. Es mussten in cook 49 Stellen und in postgresql 13 Stellen im Quellcode angepasst werden. Auch hier ist es nicht möglich, alle 62 Fälle aufzuzählen, aber einige einzelne Beispiele sollen stellvertretend für die anderen in den Abbildungen 3.6 auf dieser Seite bis 3.8 auf der nächsten Seite angeführt werden.

```
#define va_arg(x,y)
#define il8n(x) (x)
#define YY_BREAK ;
#define KERNEL_STANDARD (kernal_standard)
#define foreach(x,y)
#define ACL_MODE_STR ,
```

Abbildung 3.5: Beispiele für von Baxter benötigte #defines

```
cook/src/common/error.c: line 228
Simulating ANSI varargs with nonANSI (K&R) syntax
void
#if defined(__STDC__) && __STDC__
error_raw(char *s, ...)
#else
error_raw(s sva_last)
    char *s;
    sva_last_decl
#endif
replaced by
//void
#if defined(__STDC__) && __STDC__
void error_raw(char *s, ...)
#else
void error_raw(s sva_last)
//    char *s;
//    sva_last_decl
#endif
```

Abbildung 3.6: Beispiel für Änderungen an cook

### 3. Durchführung

```
cook/src/cook/archive.c, line 1177
statement label found on implicit block formed by preprocessor conditional
done:
#ifdef DEBUG
    errno_hold = errno;
    trace(("return %d; /* errno = %d */\n", result, errno_hold));
replaced by
done: ;
#ifdef DEBUG
    errno_hold = errno;
    trace(("return %d; /* errno = %d */\n", result, errno_hold));
```

Abbildung 3.7: Beispiel für Änderungen an cook

```
postgres/src/backend/bootstrap/bootparse.c Line 481
Preprocessor conditional around macro invocation... can't parse it
#ifdef !YYPURE
    YY_DECL_VARIABLES
#endif /* !YYPURE */
replaced by
//#ifdef !YYPURE
//YY_DECL_VARIABLES
//#endif /* !YYPURE */
```

Abbildung 3.8: Beispiel für Änderungen an postgresql

Abschließend sollen alle bei Baxter aufgetretenen Probleme kurz zusammengefasst und je ein Beispiel dafür genannt werden:

- `if`-Statements in Präprozessor-Conditionals, aber „then“-Teil außerhalb (z. B. `cook/src/common/ac/sys/utsname.c`, Zeilen 49 – 51)
- Generell K&R-Syntax (z. B. `cook/src/common/sub/expr_gram.gen.c`, Zeilen 80 – 82)
- Nachbilden von ANSI `varargs` im K&R-Stil (z. B. Abbildung 3.6 auf der vorherigen Seite)
- Label vor implizitem Block durch Präprozessor-Conditional (z. B. Abbildung 3.7)
- `case`-Label einziges Element innerhalb eines Präprozessor-Conditionals (z. B. `cook/src/cook/fingerprint/subdir.c`, Zeilen 339 – 341)
- `do`-Schleife über `case`-Labels hinweg (z. B. in `postgres/src/backend/access/hash/hashfunc.c`, Zeilen 98 – 117)
- Präprozessor-Conditional um Makro-Aufruf (Abbildung 3.8)

### 3.5. Probleme bei der Durchführung

- CloneDR™ kann keine 64bit Datentypen handhaben (z. B. postgresql/src/backend/commands/sequence.c, Zeilen 28 – 36)
- Inkorrekte Handhabung der Präprozessor-Direktive #error (z. B. postgresql/src/backend/libpq/pqformat.c, Zeilen 58 – 60)
- Assembler-Code in „totem“ Präprozessor-Zweig (z. B. postgresql/src/backend/storage/lmgr/s\_lock.c, Zeilen 70 – 87)
- Illegale Syntax innerhalb von #if 0 (z. B. postgresql/src/backend/utils/adt/datetime.c, Zeilen 86 – 97)
- Endlosschleife von CloneDR™ bei den Dateien (postgresql/src/backend/parser/gram.c und postgresql/src/backend/utils/fmgrtab.c) von postgresql (infolgedessen sind diese zwei Dateien nicht analysiert worden)

#### 3.5.3. Kamiya

Kamiya meldete keinerlei Probleme mit den acht Projekten.

#### 3.5.4. Krinke

Krinke meldete ein Problem mit `weltab`, da es aus mehreren Dateien besteht, die alle eine Funktion `main()` enthalten, sein `Duplix` aber nur eine Funktion `main()` pro System versteht. Aus diesem Grund musste er erst sein Werkzeug anpassen. Bei `snns` konnte er das Sub-System `xgui` nicht analysieren, da er noch keine eigenen Header-Dateien, die er ja benötigt (siehe Abschnitt 2.4.4 auf Seite 16), für die X-Anbindung erstellt hat. `postgresql` war zu groß, so dass er es nicht analysieren konnte. Des Weiteren stellte Krinke fest, dass die Projekte `cook` und `snns` Eigenschaften besitzen, die bei `Duplix` eine extrem hohe Laufzeit bewirken. Woran das genau liegt, konnte er jedoch noch nicht sagen. Wegen all dieser Gründe hat ihm das Zeitlimit von den gesetzten vier Wochen nicht gereicht. Er konnte zwar eine Abgabe zum korrekten Abgabezeitpunkt tätigen, allerdings war diese dermaßen sub-optimal, dass ein Vergleich nichts gebracht hätte. Aus diesem Grund hat er sein Werkzeug überarbeitet und eine verbesserte Abgabe nach insgesamt acht Wochen geliefert.

#### 3.5.5. Merlo

Giuliano Antonioli, der für Ettore Merlos `CLAN` den C-Parser wartet, meldete Probleme mit den Quellen zu `weltab`, da dort noch C-Code im K&R-Stil vorkommt. Dies hatte der Sun C Compiler, der zur Verifikation der strikten Kompatibilität mit dem ANSI C Standard verwendet wurde (siehe Abschnitt 3.2 auf Seite 28), ohne Warnung akzeptiert und daher ist es in der Haupt-Phase ausgewählt worden. Aber Antonioli scheint es möglich gewesen zu sein, seinen Parser anzupassen. Auch die anderen Teilnehmer hatten ansonsten mit dem C-Dialekt von `weltab` keine Probleme, so dass es nicht von der Bewertung ausgenommen werden muss.

### *3. Durchführung*

#### **3.5.6. Rieger**

Rieger hatte Probleme, die zwei großen Systeme postgresql und j2sdk1.4.0-javax-swing zu analysieren, da ihm verfügbare Rechenzeit und/oder Speicherplatz ausgegangen sind.

## 4. Auswertung

OBI-WAN: Your clones are very impressive. You must be very proud.

*(Star Wars II – Attack of the Clones)*

In diesem Kapitel wird erläutert, wie die Menge der Referenzklone gebildet wird. Einige kleine Beispiele von Codefragmenten, die nicht in die Referenzmenge übernommen werden sollen, werden gezeigt. Für die spätere Analyse der Daten müssen Kandidaten auf Referenzen abgebildet werden. Dieser Vorgang wird abschließend behandelt.

### 4.1. Erzeugen der Referenzmenge mittels Orakel

Bisher wurde ein Punkt völlig ausgeklammert: Die Menge der Referenzen, mit denen man die eingesandten Kandidaten der Teilnehmer vergleichen will! Hierzu wurden verschiedene Ansätze diskutiert:

- Vereinigung aller Kandidaten  
Diese Variante ist sehr einfach zu realisieren. Allerdings ist zu erwarten, dass man extrem viele „False Positives“ mit in die Menge bekommt, da ja jeder Kandidat als korrekter Klon angesehen wird. Somit würde u. a. der Wert der Precision (siehe Abschnitt 5.1.6 auf Seite 55) unbrauchbar, weil er immer 1 ergeben würde. Diese Variante wäre zu wählen, wenn alle Werkzeuge perfekte Klonpaare liefern würden. Dies ist in der Realität natürlich nicht der Fall. Ein weiterer Nachteil ist, dass viele fast identische Kandidaten mehrfach vorhanden wären, da sie bei dem einen Werkzeug eine Zeile früher beginnen oder enden als bei dem anderen etc.
- Schnittmenge aller Kandidaten  
Mit diesem Ansatz erwartet man, dass die Anzahl der „False Positives“ drastisch sinkt. Allerdings sinkt auch die Anzahl der „True Positives“, da nicht jedes Werkzeug jedes Klonpaar findet.
- Menge aller Kandidaten, die  $N$  Werkzeuge gemeinsam finden  
Dies wäre ein Kompromiss der ersten beiden Varianten, der allerdings ebenso die Nachteile dieser Möglichkeiten vereint. Und welches  $N$  wäre zu wählen?

#### 4. Auswertung

- Manuelles Bilden der Referenzmenge

Es bleibt letztendlich nur das manuelle Bilden der Referenzmenge. Dies ist allerdings ein sehr zeitaufwendiges Verfahren, da die einzelnen Kandidaten einer nach dem anderen betrachtet und klassifiziert werden müssen. Allerdings hat man dann eine reine Referenzmenge, die frei von „False Positives“ ist und dennoch „True Positives“ enthält.

In dieser Arbeit wurde die letzte Alternative gewählt. Das Bilden der Referenzmenge ist mit dem Programm clones (siehe Abschnitt B.2.2 auf Seite 132) relativ einfach möglich. Dem Schiedsrichter werden die Kandidaten der einzelnen Teilnehmer in zufälliger Reihenfolge präsentiert. Das Programm legt Wert darauf, dass der Anteil der bereits bewerteten Kandidaten pro Tupel (Projekt, Teilnehmer) gleichmäßig wächst, d. h., es ist nicht möglich, dass von einem Projekt oder Teilnehmer fast alle Kandidaten betrachtet werden und von einem anderen so gut wie keine.

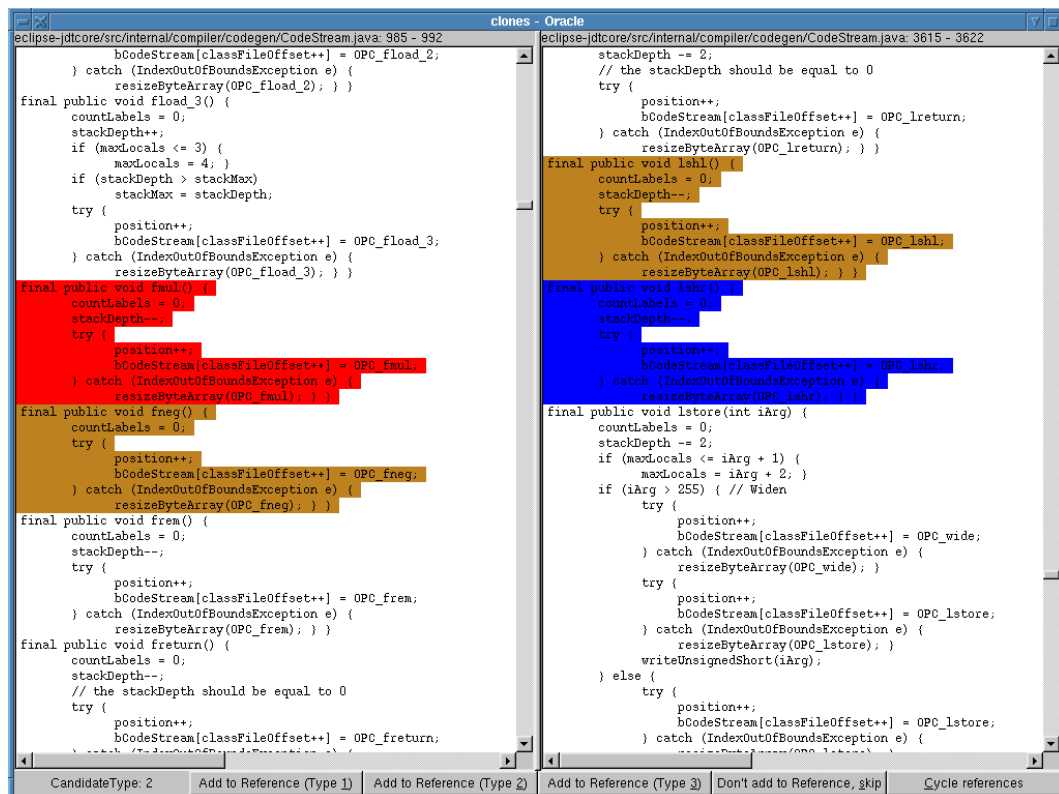


Abbildung 4.1: Bilden der Referenzmenge

Dem Schiedsrichter wird das erste Codefragment des Klonpaares in der linken Fensterhälfte präsentiert und rot hinterlegt. Das zweite Codefragment des Klonpaares erscheint in der rechten Fensterhälfte blau hinterlegt. Sind bereits Klonpaare in der unmittelbaren Umgebung des aktuellen Kandidaten in der Referenzmenge vorhanden, so werden diese

#### 4.1. Erzeugen der Referenzmenge mittels Orakel

in anderen Farben hinterlegt. Dabei bekommen die zwei zum Klonpaar gehörenden Codefragmente die gleiche Farbe. Im Beispiel in Abbildung 4.1 auf der vorherigen Seite ist außer dem aktuellen Kandidaten eine Referenz zu sehen, die bereits aufgenommen wurde. Am oberen Fensterrand werden Dateiname sowie Start- und Endzeile der jeweiligen Codefragmente angezeigt. Am unteren Bildschirmrand ist der vom Werkzeug des Teilnehmers vorgeschlagene Klontyp des Kandidaten zu sehen. Rechts daneben sind die Schaltflächen, mit denen der Kandidat in die Referenz aufgenommen werden kann (dabei wird gleichzeitig der Referenz der entsprechende Typ zugeordnet) oder übergangen werden kann. Im Falle, dass bereits andere Referenzen im Fenster angezeigt werden, gibt es zusätzlich noch eine Schaltfläche, um bei Überlappungen die verschiedenfarbigen Hinterlegungen zyklisch zu vertauschen.

Der Schiedsrichter, der die Bewertung der Kandidaten vornimmt, muss nun im Einzelfall entscheiden, ob ein gültiger Klon vorliegt, der in die Referenzmenge übernommen werden soll oder nicht. Diese Entscheidung ist in vielen Fällen offensichtlich, aber es gibt ebenso auch Fälle, in denen eine Entscheidung schwer fällt.

Im Folgenden sollen einige fiktive Beispiele für Codefragmente von Klonpaaren gegeben werden, welche im Experiment nicht als Bestandteil von gültigen Klonen anerkannt und somit nicht in die Referenzmenge aufgenommen worden wären.

```
    }
    return 0;
}

/*****
 * function xyz
 * description of function xyz...
 *****/
int
```

Abbildung 4.2: Beispiel eines ungültigen Codefragments

Vor allem tokenbasierte/zeilenbasierte Werkzeuge melden unter anderem Klonpaare, die aus Codefragmenten wie z. B. dem aus Abbildung 4.2 bestehen. Hierbei besteht der Anfang des Codefragments aus den letzten Zeilen einer Funktion und das Ende des Codefragments besteht aus der ersten Zeile der folgenden Funktion. Dies ist in keinem Fall Bestandteil eines sinnvollen Klons und daher nicht in die Referenzmenge zu übernehmen. Abbildung 4.3 auf der nächsten Seite zeigt einen weiteren Vertreter dieser Kategorie.

Des Weiteren sind nur maximale Klonpaare in die Referenzmenge zu übernehmen. D. h., wenn sich beide Codefragmente weder nach vorn noch nach hinten vergrößern lassen und auch sonst ein korrekter Klon vorliegt, so wird er übernommen. Sobald sich aber mindestens ein Codefragment in einer Richtung vergrößern lässt, ohne den Klontyp zu verletzen, wird diese Vergrößerung vorgenommen und der vergrößerte Klon ist in die Referenzmenge aufzunehmen. Dieser wird dann von dem Kandidaten des Teilnehmers nur noch teilweise überdeckt.

#### 4. Auswertung

```
catch( ... ) {
    rollback();
    error();
}

void next_function() {
    if( condition ) {
        error();
        return;
    }
}
```

Abbildung 4.3: Beispiel eines ungültigen Codefragments

Andersherum, wenn ein Kandidat eines Teilnehmers zu groß ist, da er offensichtlich am Anfang oder am Ende Teile beinhaltet, die nicht kopiert wurden, so wird der entsprechend verkleinerte Kandidat dann in die Referenzmenge übernommen.

Dies trifft insbesondere auch dann zu, wenn der Klon eine komplette Funktion oder Struktur beinhaltet, aber nicht „rechtzeitig“ endet. Dies ist im fiktiven Beispiel in Abbildung 4.4 verdeutlicht. In diesem Beispiel wäre das Codefragment, das nur aus der Klasse `this_one` besteht, zweifelsohne ein korrekter Klon, nicht aber das vorgeschlagene Codefragment in dieser Größe. Daher ist es in verkleinerter Form in die Referenzmenge zu übernehmen.

```
// one or more lines
}

public class this_one {
    // completely cloned class
}

public class next_one {
    // one or more lines
}
```

Abbildung 4.4: Beispiel eines ungültigen Codefragments

Ein letztes Beispiel zeigt, dass unbalancierte Hälften von Blöcken in extremem Ausmaß kein gültiger Klon sein können. In Abbildung 4.5 auf der nächsten Seite sind lauter `else`-Teile von `if-then-else`-Anweisungen vorhanden. Diese alleine genommen ergeben keine sinnvolle Einheit und sind nicht in die Referenzmenge zu übernehmen.

Des Weiteren werden folgende Konstruktionen nicht als Klon anerkannt:

- Teile von großen Array-Initialisierungen: Meist ist es voneinander unabhängig, dass innerhalb zwei verschiedener Array-Initialisierungen z.B. hundertmal der gleiche Wert „2“ steht. Dies als Klon zu erkennen und dann gegebenenfalls sogar durch ein Makro zu ersetzen, wäre gefährlich, sofern die Arrays nichts miteinander zu tun ha-

#### 4.1. Erzeugen der Referenzmenge mittels Orakel

```
        } else {
            return 1;
        }
    } else {
        return 2;
    }
} else {
    return 3;
}
} else {
    return 4;
}
} else {
    return 5;
}
} else {
    return 6;
}
```

Abbildung 4.5: Beispiel eines ungültigen Codefragments

ben. Eine Veränderung des Makros für ein Array würde das andere mitverändern, was nicht erwünscht ist.

- Tabellenähnliche Initialisierungen (z. B. Initialisierungen vieler Elemente eines Arrays in Folge): Prinzipiell gilt hier das Gleiche (vor allem bei Quelltext, in dem viele GUI-Elementen angelegt werden). Im Einzelfall kann natürlich ein erkennbar gültiger Klon akzeptiert werden.
- Sequenzen von `#include`-Direktiven in C bzw. `import`-Anweisungen: In Java hat es von vorn herein keinen Sinn, diese als Klone betrachten zu wollen, da man sie nicht ersetzen könnte. In C wäre es möglich, gemeinsame `#include`-Direktiven in eine Header-Datei zu verschieben und diese dann einzubinden. Allerdings macht dies den Quelltext nicht übersichtlicher, sondern verschleiert die benutzten Bibliotheken. Daher ist es nicht sinnvoll, dies als Klon anzuerkennen.
- Sich überlappende Codefragmente: Diese können ebenfalls nicht durch Makros oder Funktionen ersetzt werden und sind daher entweder anzupassen, so dass sie sich nicht mehr überlappen (sofern möglich), oder zu verwerfen.
- Codefragmente, die mit Kommentar beginnen oder enden: In solchen Fällen ist der Kommentar am Anfang oder am Ende zu entfernen. Kommentar ist nicht Bestandteil der Funktion des Codes. Es ergibt keinen Sinn, ihn in ein Makro auszulagern. Des Weiteren verlieren Werkzeuge, die den Quelltext zuerst präprozessieren müssen, die Kommentare durch den Präprozessor sowieso.

## 4. Auswertung

Zusammenfassend kann gesagt werden, dass nur Klonpaare aus sinnvoll zusammenhängenden Codefragmenten als korrekte Klone eingestuft werden. Dies trifft u. a. auch zu, wenn sich zwei oder mehrere Funktionen oder Strukturen komplett in einem Codefragment befinden, ohne dass sich davor oder danach noch Teile der vorigen oder nächsten Funktion oder Struktur befinden. Im Extremfall können auch komplette Dateien als Klon akzeptiert werden.

### 4.2. Berechnung der Daten zur späteren Analyse

Zur Analyse der Daten müssen nun Überdeckungsgrade von Kandidaten und Referenzen berechnet werden. Die grundsätzliche Idee hierfür ist [45] entnommen. Allerdings wurden dort einzelne „Objekte“ miteinander verglichen, wobei in diesem Experiment ein „Objekt“, nämlich ein Klonpaar, aus zwei Teilen besteht: den zwei Codefragmenten. Aus diesem Grund müssen die Definitionen aus [45] etwas erweitert werden.

Zuerst sollen zwei Definitionen auf Ebene der Codefragmente eingeführt werden und dann darauf aufbauend zwei Definitionen auf Ebene der Klonpaare. Im Anschluss daran werden die Definitionen an einem kleinen Beispiel veranschaulicht. Mit diesen Definitionen wird die Zuordnung von Kandidaten zu den Referenzen dann durchgeführt.

Im Folgenden wird *CP* (für englisch ClonePair) als Abkürzung für ein Klonpaar und *CS* (für englisch CodeSnippet) als Abkürzung für ein Codefragment verwendet. Dies geschieht, weil bei der Diskussion mit den internationalen Wissenschaftlern auch die englischen Definitionen verwendet wurden.

#### 4.2.1. Definition von Overlap und Contained

##### Overlap

Mit *overlap* ist der Anteil zweier Codefragmente gemeint, zu dem sie sich überdecken (also ihre Schnittmenge), bezogen auf die Vereinigungsmenge.

Bezeichnet  $\text{lines}(CS_1)$  die Zeilen des ersten Codefragments und  $\text{lines}(CS_2)$  die Zeilen des zweiten Codefragments, so lässt sich  $\text{overlap}(CS_1, CS_2)$  mathematisch wie folgt ausdrücken:

$$\text{overlap}(CS_1, CS_2) = \frac{|\text{lines}(CS_1) \cap \text{lines}(CS_2)|}{|\text{lines}(CS_1) \cup \text{lines}(CS_2)|}$$

Oder anders ausgedrückt gilt für  $\text{overlap}(CS_1, CS_2)$ , sofern sich die beiden Codefragmente in derselben Datei befinden:

$$\text{overlap}(CS_1, CS_2) = \frac{\max(0, \min(CS_1.End, CS_2.End) - \max(CS_1.Start, CS_2.Start) + 1)}{\max(CS_1.End, CS_2.End) - \min(CS_1.Start, CS_2.Start) + 1}$$

Befinden sich die Codefragmente nicht in derselben Datei, so gilt:

$$\text{overlap}(CS_1, CS_2) = 0$$

### Contained

Mit contained ist der Anteil eines Codefragments gemeint, zu dem es in einem anderen Codefragment enthalten ist.

Bezeichnet  $\text{lines}(CS_1)$  die Zeilen des ersten Codefragments und  $\text{lines}(CS_2)$  die Zeilen des zweiten Codefragments, so lässt sich  $\text{contained}(CS_1, CS_2)$  mathematisch wie folgt ausdrücken:

$$\text{contained}(CS_1, CS_2) = \frac{|\text{lines}(CS_1) \cap \text{lines}(CS_2)|}{|\text{lines}(CS_1)|}$$

$\text{contained}(CS_1, CS_2) = 0.5$  bedeutet z. B., dass die Hälfte des ersten Codefragments auch im zweiten enthalten ist.

Oder anders ausgedrückt gilt für  $\text{contained}(CS_1, CS_2)$ , sofern sich die beiden Codefragmente in derselben Datei befinden:

$$\text{contained}(CS_1, CS_2) = \frac{\max(0, \min(CS_1.End, CS_2.End) - \max(CS_1.Start, CS_2.Start) + 1)}{CS_1.End - CS_1.Start + 1}$$

Befinden sich die Codefragmente nicht in derselben Datei, so gilt:

$$\text{contained}(CS_1, CS_2) = 0$$

#### 4.2.2. Definition von Good und OK

Für die nun folgenden Definitionen ist es von essenzieller Bedeutung, dass die Codefragmente  $CS_1$  und  $CS_2$  eines Klonpaares einer Ordnung unterliegen. Diese ist wie folgt definiert:

$$\begin{aligned} CS_1 < CS_2 \iff & (CS_1.Dateiname < CS_2.Dateiname) \vee \\ & (CS_1.Dateiname = CS_2.Dateiname \wedge \\ & CS_1.Start < CS_2.Start) \vee \\ & (CS_1.Dateiname = CS_2.Dateiname \wedge \\ & CS_1.Start = CS_2.Start \wedge \\ & CS_1.End < CS_2.End) \end{aligned}$$

Für ein Klonpaar  $CP$  muss demnach immer gelten:

$$CP.CS_1 < CP.CS_2$$

Die nun folgenden zwei Definitionen werden anhand eines Beispiels im nächsten Abschnitt veranschaulicht. Dieses Beispiel ist in Abbildung 4.6 auf Seite 47 dargestellt.

#### 4. Auswertung

##### Good

Der Good-Wert zweier Klonpaare  $CP_1$  und  $CP_2$  wird wie folgt berechnet:

$$\text{good}(CP_1, CP_2) = \min(\text{overlap}(CP_1.CS_1, CP_2.CS_1), \\ \text{overlap}(CP_1.CS_2, CP_2.CS_2))$$

Man beachte, dass `overlap` nicht den Überlappungsgrad der Codefragmente eines Klonpaares berechnet, sondern den Anteil an Überlappung zwischen einem Codefragment des einen Klonpaares und einem Codefragment des anderen Klonpaares.

Zwei Klonpaare  $CP_1$  und  $CP_2$  werden als `Good-Match(p)` bezeichnet, wenn für ein  $p \in [0, 1]$  gilt:

$$\text{good}(CP_1, CP_2) \geq p$$

##### OK

Der OK-Wert zweier Klonpaare  $CP_1$  und  $CP_2$  wird wie folgt berechnet:

$$\text{ok}(CP_1, CP_2) = \min(\max(\text{contained}(CP_1.CS_1, CP_2.CS_1), \\ \text{contained}(CP_2.CS_1, CP_1.CS_1)), \\ \max(\text{contained}(CP_1.CS_2, CP_2.CS_2), \\ \text{contained}(CP_2.CS_2, CP_1.CS_2)))$$

Man beachte, dass `contained` nicht den Anteil an Übereinstimmung der Codefragmente eines Klonpaares berechnet, sondern den Anteil an Übereinstimmung zwischen einem Codefragment des einen Klonpaares und einem Codefragment des anderen Klonpaares.

Zwei Klonpaare  $CP_1$  und  $CP_2$  werden als `OK-Match(p)` bezeichnet, wenn für ein  $p \in [0, 1]$  gilt:

$$\text{ok}(CP_1, CP_2) \geq p$$

#### 4.2.3. Bedeutung von Good und OK

Bei der Analyse der Ergebnisse (siehe Kapitel 5 auf Seite 51) wird jede Berechnung mit zwei Kriterien durchgeführt: Einmal mit dem Good-Wert und einmal mit dem OK-Wert.

Dies hat folgenden Grund: zwei Klonpaare, welche `OK-Match(p)` sind, überlappen sich zumindest in eine Richtung um mindestens den Anteil  $p$ . Dies heißt konkret, dass mindestens  $p \cdot 100\%$  des einen Klonpaares im anderen enthalten sind. Dabei kann das eine Klonpaar aber viel größer sein als das andere, so dass diese Aussage qualitativ an Wert verliert.

Mit dem `Good-Match(p)` ist dies anders: Dort beziehen sich die  $p \cdot 100\%$  auf die Vereinigung der beiden Klonpaare. Sollte nun also ein Klonpaar wesentlich größer sein als das andere, wird sich kein `Good-Match(p)` mehr ergeben.

Das kleine Beispiel in Abbildung 4.6 auf der nächsten Seite soll die vorangegangenen Definitionen und Erklärungen veranschaulichen.

## 4.2. Berechnung der Daten zur späteren Analyse

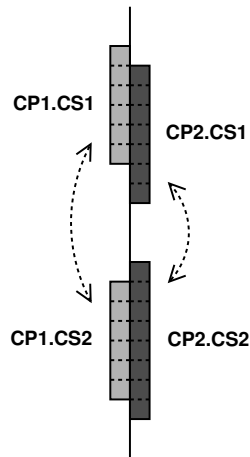


Abbildung 4.6: Beispiel für Überlappungen zweier Klonpaare

Die senkrechte Linie in der Bildmitte stellt den linearen Quellcode dar. Oben ist die erste Zeile und unten die letzte Zeile der Datei (befinden sich die Codefragmente des Klonpaares in unterschiedlichen Dateien, so stelle man sich in der Mitte eine Unterbrechung der senkrechten Linie vor). Die Codefragmente der beteiligten Klonpaare sind durch die grau gefüllten Rechtecke dargestellt. Ein durch eine gepunktete Linie getrennter Block soll eine Zeile darstellen. Auf der linken Seite sieht man das erste Klonpaar  $CP_1$ . Im Beispiel soll dies nun einen Kandidaten darstellen. Auf der rechten Seite ist das zweite Klonpaar, in diesem Falle eine Referenz. Das jeweils erste Codefragment  $CS_1$  wurde irgendwann einmal nach  $CS_2$  kopiert (oder umgekehrt) und eventuell verändert. Dies symbolisieren die gepunkteten Pfeile. Man erkennt, wie das erste Codefragment des Kandidaten früher beginnt und endet als das der Referenz. Das zweite Codefragment hingegen ist völlig im Codefragment der Referenz enthalten.

Der Good-Wert berechnet sich nun wie folgt:

$$\text{good}(CP_1, CP_2) = \min\left(\frac{5}{8}, \frac{6}{8}\right) = \frac{5}{8} < 0.7$$

Damit ist das Beispiel kein Good-Match(0.7). Hiermit erklärt sich nun auch die Verwendung von  $\min$ : Es reicht nicht, wenn sich eines der beiden Codefragmente des Kandidaten gut mit dem der Referenz überdeckt, wenn sich das andere nur schlecht überdeckt. Daher wird das Codefragment als Kriterium genommen, das sich schlechter überdeckt.

Betrachtet man den OK-Wert, ergibt sich folgendes Bild:

$$\text{ok}(CP_1, CP_2) = \min\left(\max\left(\frac{5}{6}, \frac{5}{7}\right), \max\left(\frac{6}{6}, \frac{6}{8}\right)\right) = \frac{5}{6} > 0.7$$

Somit liegt mit dem Beispiel ein OK-Match(0.7) vor. Im Gegensatz zur Berechnung des Good-Wertes werden beim OK-Wert noch zwei  $\max$ -Funktionen benötigt. Dies kommt daher, dass  $\text{overlap}$  symmetrisch ist,  $\text{contained}$  aber nicht. Aus diesem Grund müssen beim

#### 4. Auswertung

OK-Wert beide Richtungen berücksichtigt werden und die bessere wird dann gewertet. Weiterhin müssen aber beide Codefragmente „gut genug“ treffen. Dies wird wiederum mit der min-Funktion erreicht.

Des Weiteren kann man sich anschaulich klar machen, dass für zwei Klonpaare  $CP_1$  und  $CP_2$  immer gilt:  $ok(CP_1, CP_2) \geq good(CP_1, CP_2)$ .

Zusammenfassend kann also gesagt werden, dass das Suchen nach OK-Match( $p$ ) dann Sinn hat, wenn es darum geht, maximal viele Klonpaare zu finden, die aber manuell verifiziert, nachgebessert und verfeinert werden müssen. Ein Good-Match( $p$ ) hingegen ist geeignet zur automatischen Weiterverarbeitung (z. B. Ersetzen der Klone durch Makro- oder Funktionsaufrufe mit einmaliger Definition des Makros bzw. der Funktion) der gefundenen Klonpaare.

##### 4.2.4. Zuordnung von Kandidaten zu Referenzen

Beim Start des Zuordnungsvorgangs aus dem Hauptmenü des Hilfsprogramms `clones` (siehe Abschnitt B.2.2 auf Seite 132) heraus muss man den Parameter  $p$  für obige Good-Match- und OK-Match-Bewertung angeben. Im Experiment wird hier  $p = 0.7$  gewählt, da dies der in [45] vorgeschlagene Wert ist. Dieser Wert ist vor allem dadurch motiviert, dass im Experiment unter anderem auch herausgefunden werden soll, inwieweit die Werkzeuge untereinander gleiche Klone entdecken. Daher sollte er nicht allzu hoch gewählt werden. Will man hingegen nur „sehr gute“ Treffer berücksichtigen, muss  $p$  höher angesetzt werden.

Zu Beginn ist keinem der Kandidaten eine Referenz zugeordnet. Daraufhin erfolgt das Zuordnen von Referenzen zu den Kandidaten. Hierbei wird nach dem Algorithmus in Abbildung 4.7 verfahren.

```
for r in References'range loop
  for c in Candidates'range loop
    ok_max := ok(c, getBestReferenceOf(c));
    good_max := good(c, getBestReferenceOf(c));
    ok := ok(c, r);
    good := good(c, r);
    if better then
      setBestReferenceOf(c, r);
    end if;
  end loop;
end loop;
```

Abbildung 4.7: Zuordnungsalgorithmus

Die „magische“ Funktion `better` lässt sich am einfachsten mit Abbildung 4.8 auf der nächsten Seite und den folgenden Erläuterungen beschreiben. In der Abbildung ist in x-Richtung der Good-Wert und in y-Richtung der OK-Wert aufgetragen. Der Startpunkt eines Pfeils entspricht dem Wertetupel  $(good\_max, ok\_max)$  und der Endpunkt eines Pfeils entspricht dem Wertetupel  $(good, ok)$ . Da immer  $ok(CP_1, CP_2) \geq good(CP_1, CP_2)$  gilt, ist das hellrot einge-

färbte Dreieck niemals mit so einem Tupel erreichbar. Ein OK-Match( $p$ ) liegt in beiden grau hinterlegten Bereichen vor, ein Good-Match( $p$ ) liegt im dunkelgrau hinterlegten Bereich vor. Ziel ist es also, mit dem Tupel ( $good, ok$ ) möglichst in den dunkelgrauen oder zumindest in den hellgrauen Bereich zu gelangen.

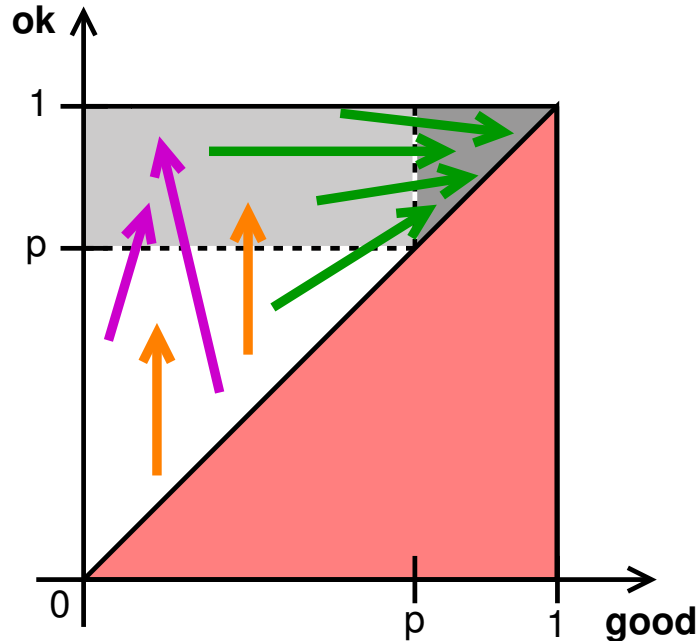


Abbildung 4.8: Graphische Darstellung der Funktion better

Für die Funktion better gilt nun, dass die aktuell betrachtete Referenz  $r$  besser für den Kandidaten  $c$  geeignet ist als die bisher beste Referenz  $getBestReferenceOf(c)$ , wenn mindestens eine der drei folgenden Bedingungen zutrifft:

- $(good \geq p \wedge good > good\_max)$   
 Good-Wert steigt und befindet sich oberhalb der Schranke  $p$ . In der Abbildung ist dies durch grüne Pfeile dargestellt.  
 Motivation: Eine Referenz, die ein Good-Match( $p$ ) ist, hat Vorrang. Gibt es eine Referenz mit einem besseren Good-Wert, so hat diese Vorrang.
- $(good = good\_max \wedge ok > ok\_max)$   
 Good-Wert bleibt gleich und OK-Wert steigt. Die orangenen Pfeile in der Abbildung sind Beispiele hierfür.  
 Motivation: Wenn der Good-Wert gleich ist, hat eine Referenz mit höherem OK-Wert Vorrang.
- $(ok \geq p \wedge ok\_max < p)$   
 OK-Wert steigt von unter  $p$  auf über  $p$  an. Die violetten Pfeile in der Abbildung veranschaulichen dies.

#### 4. Auswertung

Motivation: Es hat eine Referenz Vorrang, deren OK-Wert die Schwelle  $p$  nach oben überschreitet. Solange der Good-Wert sowieso unterhalb der Schwelle  $p$  liegt, kann er dabei auch sinken. Man beachte, dass ein Good-Wert, der bereits oberhalb von  $p$  ist, durch diese Bedingung nicht unter  $p$  sinken kann!

Legt man den Schwerpunkt allerdings auf einen möglichst hohen Good-Wert, so sollte diese Bedingung entfernt werden. Da es für das Experiment nur von Interesse ist, ob ein Good-Match( $p$ ) vorliegt oder nicht, der konkrete Wert aber nicht interessiert, kann diese Bedingung verwendet werden, um eventuell noch einige OK-Match( $p$ ) zu „gewinnen“.

Aus Geschwindigkeitsgründen ist nicht nur der Wert `getBestReferenceOf(c)` eines Kandidaten  $c$  in seinem Datensatz in der Datenbank gespeichert, sondern auch die Werte `ok_max` und `good_max` sind in der Tabelle `Candidate` gespeichert (siehe Abschnitt 2.6 auf Seite 24).

## 5. Ergebnisanalyse

YODA: Blind we are, if creation of this clone army  
we could not see.

*(Star Wars II – Attack of the Clones)*

Nun werden zuerst einige Begriffe eingeführt und definiert, auf die dann später bei der Analyse der Ergebnisse zurückgegriffen wird. Die Ergebnisanalyse selbst findet zuerst für die einzelnen Projekte des Experiments und danach für die Teilnehmer statt.

### 5.1. Definition der in der Ergebnisanalyse verwendeten Begriffe

In den folgenden Definitionen wird  $T$  als Variable für ein Werkzeug (Tool),  $P$  als Variable für ein Projekt (Project) und  $\tau$  als Variable für einen Klontyp (Type) verwendet.

Die Definitionen werden hier mit obigen Variablen versehen, damit klar wird, wie sich die Werte errechnen. In den Abschnitten 5.2 auf Seite 58 und 5.3 auf Seite 109 werden diese Variablen weggelassen, da aus dem Kontext des Abschnitts hervorgeht, um welches Projekt und welchen Teilnehmer es sich handelt. Der Klontyp kann aus den Schraffuren der Diagramme abgelesen werden.

Die Variable  $P$  bezeichnet immer ein Projekt des Experiments oder „insgesamt“, sofern der jeweilige Wert für alle Projekte berechnet werden soll.

Die Variable  $T$  bezeichnet immer ein teilnehmendes Werkzeug oder „insgesamt“, sofern der jeweilige Wert für alle Werkzeuge berechnet werden soll.

Die Variable  $\tau$  bezeichnet immer einen der drei Klontypen oder „insgesamt“, sofern der jeweilige Wert unabhängig vom Klontyp berechnet werden soll. Für  $\tau$  gibt es noch den Fall, „unbestimmt“ einzusetzen, um den Wert für die Menge der unbestimmten Klontypen zu berechnen. Dies ist nicht immer sinnvoll. In den folgenden Definitionen wird daher explizit erwähnt, wenn der Wert auch für „unbestimmt“ berechnet werden kann.

#### 5.1.1. Kandidaten

Bei den Kandidaten handelt es sich um die von den einzelnen Teilnehmern eingesendeten Klonpaare. Je nach Fähigkeit des Werkzeuges werden die Kandidaten mit Klontypen behaftet. Diese Einteilung der Werkzeuge wird berücksichtigt und in den Diagrammen der nächsten Abschnitte ist anhand der Schraffur der jeweilige Klontyp zu erkennen. Kandidaten von

## 5. Ergebnisanalyse

Werkzeugen, welche diese Einteilung nicht vornehmen können, werden als „unbestimmt“ geführt.

$Candidates(P,T,\tau)$  bezeichnet die Anzahl der Kandidaten in einem Projekt  $P$ , die das Werkzeug  $T$  mit einem bestimmten Klontyp  $\tau$  entdeckt. Für den Klontyp kann hier auch „unbestimmt“ eingesetzt werden.

### 5.1.2. Referenzen

In den Diagrammen, welche die Anzahl der von Kandidaten getroffenen Referenzen veranschaulichen, wird ebenfalls der Klontyp durch die Schraffur kenntlich gemacht. Hierbei handelt es sich jedoch nicht um den Klontyp, den das Werkzeug dem Kandidaten zuordnet, sondern um den Klontyp der Referenz, die von dem Kandidaten getroffen wird. Daher gibt es hier keine „unbestimmten“ Klonpaare. Daraus folgt auch, dass nicht die Anzahl der Kandidaten aufgetragen ist, die eine Referenz treffen, sondern die Anzahl der getroffenen Referenzen. Diese kann unter Umständen niedriger sein, wenn mehrere Kandidaten die gleiche Referenz überdecken.

Des Weiteren werden in diesem Diagramm sowohl die Werte für OK-Match(0.7) als auch die Werte für Good-Match(0.7) aufgetragen. Hieran lässt sich erkennen, ob das Werkzeug vermehrt Kandidaten liefert, die sich in derselben Größenordnung der Referenz befinden oder ob sie viel zu groß oder viel zu klein sind.

Die Anzahl der Klonpaare in der Referenzmenge wird durch den Balken „Orakel“ angegeben.

$OKReferences(P,T,\tau)$  ist die Menge der Referenzen vom Typ  $\tau$ , die im Projekt  $P$  vom Werkzeug  $T$  als OK-Match(0.7) überdeckt werden. Entsprechend ist  $GoodReferences(P,T,\tau)$  die Menge von Referenzen vom Typ  $\tau$ , die im Projekt  $P$  vom Werkzeug  $T$  als Good-Match(0.7) überdeckt werden. Für das Werkzeug  $T$  lässt sich hier auch „Orakel“ einsetzen. In den folgenden Definitionen sei  $References(P,T,\tau)$  ein Synonym für  $OKReferences(P,T,\tau)$  bzw.  $GoodReferences(P,T,\tau)$  (je nachdem, ob der Wert im Rahmen einer OK- oder einer Good-Auswertung benutzt wird; siehe Abschnitt 4.2.3 auf Seite 46).

Ein weiterer interessanter Aspekt ist die Anzahl der Referenzen, die von mehreren Werkzeugen erkannt werden. In den folgenden Abschnitten werden diese mehrfach gefundenen Referenzen tabellarisch aufgelistet. Hierbei sind jedoch die zwei „Kür“-Abgaben von Kamiya und Merlo nicht berücksichtigt, da die Ergebnisse sonst verfälscht wären: Sobald einer der beiden eine Referenz trifft, würde sie doppelt gezählt, wenn sie in der „Kür“ ebenfalls gefunden wird.

Eine genauere Aufschlüsselung dieser Mehrfachfunde ist durch eine Matrix von gemeinsam erkannten Referenzen möglich. Zusätzlich zu dieser Matrix der Schnittmenge der gefundenen Referenzen wird noch eine Matrix verwendet, welche die Differenz der von den einzelnen Werkzeugen gefundenen Referenzen wiedergibt. Von den verworfenen Kandidaten ( $RejectedCandidates(P,T,\tau)$ ) werden ebenfalls Schnitt und Differenz in zwei Matrizen dargestellt.

In Abbildung 5.1 auf der nächsten Seite ist ein Beispiel einer solchen Schnittmengenmatrix abgebildet.

### 5.1. Definition der in der Ergebnisanalyse verwendeten Begriffe

	Baxter	Kamiya	Kamiya (vol.)	Krinke	Merlo	Merlo (vol.)	Rieger
Baker	$x_{12}$	$x_{13}$	$x_{14}$	$x_{15}$	$x_{16}$	$x_{17}$	$x_{18}$
Baxter	-	$x_{23}$	$x_{24}$	$x_{25}$	$x_{26}$	$x_{27}$	$x_{28}$
Kamiya	-	-	$x_{34}$	$x_{35}$	$x_{36}$	$x_{37}$	$x_{38}$
Kamiya (vol.)	-	-	-	$x_{45}$	$x_{46}$	$x_{47}$	$x_{48}$
Krinke	-	-	-	-	$x_{56}$	$x_{57}$	$x_{58}$
Merlo	-	-	-	-	-	$x_{67}$	$x_{68}$
Merlo (vol.)	-	-	-	-	-	-	$x_{78}$

Abbildung 5.1: Schnitt der gefundenen Referenzen (bzw. verworfenen Kandidaten) zwischen den Werkzeugen

Für die Matrix der Schnitte der gefundenen Referenzen gilt (für  $i < j$ , sonst undefiniert):

$$x_{ij} = |\text{References}(P, \text{Tool}(i), \tau) \cap \text{References}(P, \text{Tool}(j), \tau)|$$

Für die Matrix der Schnitte der verworfenen Kandidaten gilt (für  $i < j$ , sonst undefiniert):

$$x_{ij} = |\text{RejectedCandidates}(P, \text{Tool}(i), \tau) \cap \text{RejectedCandidates}(P, \text{Tool}(j), \tau)|$$

	Baker	Baxter	Kamiya	Kamiya (vol.)	Krinke	Merlo	Merlo (vol.)	Rieger
Baker	-	$x_{12}$	$x_{13}$	$x_{14}$	$x_{15}$	$x_{16}$	$x_{17}$	$x_{18}$
Baxter	$x_{21}$	-	$x_{23}$	$x_{24}$	$x_{25}$	$x_{26}$	$x_{27}$	$x_{28}$
Kamiya	$x_{31}$	$x_{32}$	-	$x_{34}$	$x_{35}$	$x_{36}$	$x_{37}$	$x_{38}$
Kamiya (vol.)	$x_{41}$	$x_{42}$	$x_{43}$	-	$x_{45}$	$x_{46}$	$x_{47}$	$x_{48}$
Krinke	$x_{51}$	$x_{52}$	$x_{53}$	$x_{54}$	-	$x_{56}$	$x_{57}$	$x_{58}$
Merlo	$x_{61}$	$x_{62}$	$x_{63}$	$x_{64}$	$x_{65}$	-	$x_{67}$	$x_{68}$
Merlo (vol.)	$x_{71}$	$x_{72}$	$x_{73}$	$x_{74}$	$x_{75}$	$x_{76}$	-	$x_{78}$
Rieger	$x_{81}$	$x_{82}$	$x_{83}$	$x_{84}$	$x_{85}$	$x_{86}$	$x_{87}$	-

Abbildung 5.2: Differenz der gefundenen Referenzen (bzw. verworfenen Kandidaten) zwischen den Werkzeugen

In Abbildung 5.2 ist ein Beispiel einer solchen Differenzenmatrix abgebildet.

Für die Matrix der Differenzen der gefundenen Referenzen gilt (für  $i \neq j$ , sonst undef.):

$$x_{ij} = |\text{References}(P, \text{Tool}(i), \tau) \setminus \text{References}(P, \text{Tool}(j), \tau)|$$

Für die Matrix der Differenzen der verworfenen Kandidaten gilt (für  $i \neq j$ , sonst undef.):

$$x_{ij} = |\text{RejectedCandidates}(P, \text{Tool}(i), \tau) \setminus \text{RejectedCandidates}(P, \text{Tool}(j), \tau)|$$

Diese Matrizen werden in den folgenden Abschnitten aus Platzgründen nicht abgebildet. Es werden aber Auffälligkeiten dieser Matrizen erwähnt werden. Anhand dieser Matrizen,

## 5. Ergebnisanalyse

die Schnittmengen oder Differenzen von gefundenen Referenzen oder verworfenen Kandidaten jeweils zweier Teilnehmer darstellen, kann also unter anderem festgestellt werden, ob Werkzeuge ähnlich „falsche“ Kandidaten oder komplett verschiedene gute Referenzen melden.

### 5.1.3. FoundSecrets

Um die Konfidenz der Ergebnisse zu stärken, sind in die einzelnen Projekte von Hand Klone eingebaut worden, um zu testen, inwieweit diese von den Teilnehmern entdeckt werden. Abbildung 5.3 ist eine Übersicht über die versteckten Klonpaare.

Projekt	Typ 1	Typ 2	Typ 3	Summe
weltab	5	6	7	18
cook	1	1	1	3
sns	1	0	1	2
postgresql	0	2	0	2
netbeans-javadoc	4	6	6	16
eclipse-ant	0	2	1	3
eclipse-jdtcore	2	0	1	3
j2sdk1.4.0-javax-swing	0	1	2	3
Summe	13	18	19	50

Abbildung 5.3: Übersicht der versteckten Klone

Die etwas seltsame Aufteilung der Klone auf die acht Projekte hat einen einfachen Grund: Das „Einbauen“ der versteckten Klonpaare war zeitaufwendiger als geplant. Daher befinden sich in den Projekten, die zuerst präpariert wurden, mehr Klone als in den später präparierten Projekten.

FoundSecrets ist nun die Anzahl der versteckten Referenzen, die von den Teilnehmern entdeckt werden.

### 5.1.4. Rejected

Es gibt Kandidaten, die vom Schiedsrichter bewertet wurden, die aber bei der Auswertung auf keine Referenzen treffen. Diese Kandidaten werden  $\text{RejectedCandidates}(P,T,\tau)$  genannt. Dies unterscheidet sich erheblich von den  $\text{FalsePositives}(P,T,\tau)$ : Bei den  $\text{FalsePositives}(P,T,\tau)$  können unter Umständen noch Kandidaten dabei sein, die beim Bewerten eine Referenz erzeugen würden. Da sie aber nicht bewertet wurden, treffen sie auf keine Referenz. Daher ist das Maß der  $\text{FalsePositives}(P,T,\tau)$  nicht ideal, wenn nicht alle Kandidaten bewertet wurden. Dies ist im Experiment aus Zeitgründen nicht möglich gewesen. Daher bezieht sich  $\text{RejectedCandidates}(P,T,\tau)$  nur auf diejenigen Kandidaten, die bewertet wurden.

Es sind jedoch weniger die absoluten Zahlen von Interesse, als vielmehr diese Werte bezogen auf die Anzahl der bewerteten Kandidaten ( $\text{SeenCandidates}(P,T,\tau)$ ). Dieser Anteil

## 5.1. Definition der in der Ergebnisanalyse verwendeten Begriffe

wird nun  $\text{Rejected}(P,T,\tau)$  genannt:

$$\text{Rejected}(P, T, \tau) = \frac{|\text{RejectedCandidates}(P, T, \tau)|}{|\text{SeenCandidates}(P, T, \tau)|}$$

Wie man sieht, werden bei der Aufschlüsselung in die einzelnen Klontypen jeweils die Werte  $\text{RejectedCandidates}(P,T,\tau)$  und  $\text{SeenCandidates}(P,T,\tau)$  der einzelnen Typen benutzt, um  $\text{Rejected}(P,T,\tau)$  zu berechnen. Somit gibt es in diesem Diagramm wieder die Kategorie „unbestimmt“.

Der Balken „insgesamt“ entspricht der Betrachtung der Klontypen, wenn man sie nicht aufschlüsselt. Allerdings gilt zu berücksichtigen, dass dieser Wert sich nicht aus dem arithmetischen Mittel der anderen Balken berechnen lässt, da die Anzahl der Kandidaten des jeweiligen Typs mit eingeht und diese in der Regel unterschiedlich ist.

Generell ist zu diesem Diagrammtyp zu bemerken, dass Balken nur für die Klontypen vorhanden sind, für die der Teilnehmer bei diesem Projekt Kandidaten eingesandt hat. Bei einem Teilnehmer, der nur Klone von unbestimmtem Typ meldet, wird somit kein Balken bei den Typen 1 bis 3 erscheinen. Und ein Teilnehmer, der nur die Typen 1 und 2 klassifizieren kann und meldet, wird keinen Balken bei den unbestimmten Typen und Typ 3 haben.

### 5.1.5. TrueNegatives

$\text{TrueNegativeReferences}(P,T,\tau)$  ist die Menge der Referenzen, die von keinem Kandidaten getroffen werden. Allerdings ist auch hier wieder weniger der Absolutwert als vielmehr das Verhältnis zur Menge aller vorhandener Referenzen eines Projektes und Klontyps ( $\text{References}(P,\tau)$ ) von Interesse. Dieses Maß soll nun  $\text{TrueNegatives}(P,T,\tau)$  genannt werden:

$$\text{TrueNegatives}(P, T, \tau) = \frac{|\text{TrueNegativeReferences}(P, T, \tau)|}{|\text{References}(P, \tau)|}$$

$\text{TrueNegativeReferences}(P,T,\tau)$  berücksichtigt den vom Werkzeug eingestuften Klontyp. Sollte eine Referenz nicht mit einem Kandidaten des gleichen Typs überdeckt werden, so ist sie in den  $\text{TrueNegativeReferences}(P,T,\tau)$  für dieses Werkzeug enthalten.

Balken mit einem Wert von 1 bedeuten in diesem Diagramm, dass der Teilnehmer keinen einzigen Kandidaten dieses Klontyps gefunden hat. Ähnlich zum vorigen Diagrammtyp heißt dies folgerichtig, dass manche Teilnehmer für bestimmte Klontypen immer Balken der Höhe 1 haben. Da es überdies keine Klone von unbestimmtem Typ in den Referenzen gibt, werden diese Balken nicht angezeigt. Dies gilt es natürlich beim Betrachten des Diagramms zu berücksichtigen. Diese Balken wurden aber dennoch nicht weggelassen, da sonst der Eindruck entstehen könnte, das jeweilige Werkzeug könne alle Klone dieses Typs erkennen.

### 5.1.6. Recall und Precision

#### Recall

Unter  $\text{Recall}(P,T,\tau)$  versteht man den Anteil der Referenzen, die von Kandidaten abgedeckt sind, bezogen auf die Menge aller vorhandenen Referenzen eines Projektes und Klontyps

## 5. Ergebnisanalyse

(References( $P, \tau$ )). Recall( $P, T, \tau$ ) ist somit ein Maß dafür, wie viele der vorhandenen Klone ein Werkzeug entdeckt.

Mathematisch ausgedrückt heißt dies:

$$\text{Recall}(P, T, \tau) = \frac{|\text{References}(P, T, \tau)|}{|\text{References}(P, \tau)|}$$

Für  $\tau$  kann hier wieder „unbestimmt“ eingesetzt werden.

### Precision

Unter Precision( $P, T, \tau$ ) versteht man den Anteil der Referenzen, die von Kandidaten abgedeckt sind, bezogen auf die Kandidaten. Precision( $P, T, \tau$ ) ist somit ein Maß dafür, wie groß die Menge „sinnvoller“ Kandidaten eines Werkzeuges ist.

Mathematisch ausgedrückt heißt dies:

$$\text{Precision}(P, T, \tau) = \frac{|\text{References}(P, T, \tau)|}{|\text{Candidates}(P, T, \tau)|}$$

Für  $\tau$  kann hier wieder „unbestimmt“ eingesetzt werden.

Bei Candidates( $P, T, \tau$ ) wird, wie man sieht, der Typ genommen, der das Werkzeug dem Kandidaten gibt. Bei References( $P, T, \tau$ ) hingegen wird der Typ genommen, den die Referenz beim Bewerten durch den Schiedsrichter erhalten hat. Dies hat zur Folge, dass Werkzeuge, welche nur Kandidaten vom Typ „unbestimmt“ abgeben, nur den Balken für die Precision insgesamt haben, aber keine Aufteilung in die einzelnen Typen.

### 5.1.7. Klongrößen

Es stellt sich außerdem auch immer wieder die Frage, wie groß die Kandidaten eigentlich sind, die von den Werkzeugen erkannt werden.

Die Größe eines Klonpaares  $CP$  mit den zwei Codefragmenten  $CS_1$  und  $CS_2$  ist im Experiment auf zwei Arten definiert:

- Für die Berechnung der maximalen Klongröße (MaxReferenceSize( $P, T, \tau$ ) und MaxCandidateSize( $P, T, \tau$ )):

$$\text{size}(CP) = \max(\text{size}(CP.CS_1), \text{size}(CP.CS_2))$$

- Für die Berechnung der durchschnittlichen Klongröße, Abweichung und Varianz (AvgReferenceSize( $P, T, \tau$ ), AvgCandidateSize( $P, T, \tau$ ), StdDevReferenceSize( $P, T, \tau$ ), StdDevCandidateSize( $P, T, \tau$ ), VarianceReferenceSize( $P, T, \tau$ ) und VarianceCandidateSize( $P, T, \tau$ )):

$$\text{size}(CP) = \frac{\text{size}(CP.CS_1) + \text{size}(CP.CS_2)}{2}$$

## 5.1. Definition der in der Ergebnisanalyse verwendeten Begriffe

Und wie bisher gilt natürlich:

$$\text{size}(CS) = CS.End - CS.Start + 1$$

Diese Definitionen sind motiviert durch die Tatsache, dass für die maximale Klongröße in der Tat auch das maximale Codefragment von Interesse ist, während für Durchschnittsberechnungen das arithmetische Mittel der zwei Codefragmente sinnvoller ist.

$\text{MaxCloneSize}(P,T,\tau)$  wird in den Abschnitten 5.2 auf der nächsten Seite und 5.3 auf Seite 109 zusammenfassend für  $\text{MaxReferenceSize}(P,T,\tau)$  und  $\text{MaxCandidateSize}(P,T,\tau)$  benutzt. Entsprechendes gilt für  $\text{AvgCloneSize}(P,T,\tau)$  und  $\text{StdDevCloneSize}(P,T,\tau)$ .

In dieser Arbeit werden hier nur die Werte für  $\tau = \text{„insgesamt“}$  berücksichtigt. Das Programm `clones` (siehe Abschnitt B.2.2 auf Seite 132) berechnet diese Werte aber auch für die drei Klontypen und für „unbestimmt“.

### 5.1.8. Klonverteilung

Die Klonverteilung gibt an, zu welchem Anteil die gefundenen Referenzen aus Codefragmenten bestehen, die innerhalb einer Datei kopiert wurden ( $\text{InnerFileReferences}(P,T,\tau)$  und  $\text{InnerFileCandidates}(P,T,\tau)$ ) und zu welchem Anteil sie aus solchen bestehen, die über Dateien hinweg kopiert wurden ( $\text{AcrossFileReferences}(P,T,\tau)$  und  $\text{AcrossFileCandidates}(P,T,\tau)$ ). Es werden sowohl die absoluten Werte dieser Referenzen angegeben als auch der Anteil in Prozent.

Die Idee zu dieser Auswertung ist [32] entnommen.

In den Abschnitten 5.2 auf der nächsten Seite und 5.3 auf Seite 109 wird  $\text{InnerFilePairs}(P,T,\tau)$  zusammenfassend für  $\text{InnerFileReferences}(P,T,\tau)$  und  $\text{InnerFileCandidates}(P,T,\tau)$  verwendet. Entsprechendes gilt auch für  $\text{AcrossFilePairs}(P,T,\tau)$ .

In dieser Arbeit werden hier nur die Werte für  $\tau = \text{„insgesamt“}$  berücksichtigt. Das Programm `clones` (siehe Abschnitt B.2.2 auf Seite 132) berechnet diese Werte aber auch für die drei Klontypen und für „unbestimmt“.

### 5.1.9. Verschiedenes

Zuletzt sollen noch einige interessante, aber bisher nicht erwähnte Werte aufgelistet werden. Es handelt sich hierbei um die Anzahl der Referenzen, die nur von dem jeweiligen Werkzeug  $T$  erkannt werden, nicht aber von den anderen Werkzeugen ( $\text{OnlyPairs}(P,T,\tau)$ ); des Weiteren die Anzahl der Referenzen, die von dem jeweiligen Werkzeug  $T$  als einziges nicht erkannt werden ( $\text{OnlyButOnePairs}(P,T,\tau)$ ) und die Anzahl der Kandidaten, bei denen sich die zwei Codefragmente überlappen ( $\text{OverlappingCandidates}(P,T,\tau)$ ).

In dieser Arbeit werden hier nur die Werte für  $\tau = \text{„insgesamt“}$  berücksichtigt. Das Programm `clones` (siehe Abschnitt B.2.2 auf Seite 132) berechnet diese Werte aber auch für die drei Klontypen und für „unbestimmt“.

## 5.2. Auswertung nach Projekten

Die Auswertung der Daten ist zum einen unterteilt in die im Abschnitt 4.2.2 auf Seite 45 definierten Good-Match( $p$ )- und OK-Match( $p$ )-Werte, wobei  $p = 0.7$  gewählt ist. Dies bedeutet, dass im Folgenden alle Werte, welche mit Good beschriftet sind, sich auf das Kriterium Good-Match(0.7) beziehen, alle Werte, die mit OK beschriftet sind, beziehen sich dementsprechend auf OK-Match(0.7). Ist weder Good noch OK vermerkt, so handelt es sich um einen davon unabhängigen Wert.

Zum anderen findet eine Unterteilung nach dem Anteil der bewerteten Kandidaten statt. Ursprünglich war vorgesehen, einen Auswertungslauf durchzuführen, wenn 5 % der Kandidaten bewertet worden sind und einen zweiten Auswertungslauf, wenn weitere 5 % (also insgesamt 10 %) bewertet worden sind. Dies war in Anbetracht der großen Anzahl von Kandidaten nicht möglich (siehe Abbildung 5.4). Die erste Auswertung ist nun nach 1 %, die zweite Auswertung nach 2 % erfolgt. Für die Bewertung des ersten Prozentes habe ich 44 Stunden benötigt, das zweite Prozent konnte ich in 33 Stunden bewerten. Dies liegt daran, dass man Übung beim Bewerten bekommt und es somit mit der Zeit schneller geht. Der Sinn dieser zwei Auswertungen liegt darin, Rückschlüsse daraus ziehen zu können, inwieweit der Umstand, dass nur so wenige Kandidaten bewertet werden, die Ergebnisse verfälscht. Bei allen Werten wird der Auswertungszeitpunkt (1 % oder 2 %) vermerkt, ansonsten ist der Wert unabhängig davon.

Projekt	Kandidaten	bewertet	Referenzen	Ausbeute (%)
weltab	13901	280	252	90.00
cook	27122	544	402	73.90
snns	66331	1329	903	67.95
postgresql	59114	1182	555	46.95
netbeans-javadoc	7860	159	55	34.59
eclipse-ant	2440	51	30	58.82
eclipse-jdtcore	92905	1856	1345	72.47
j2sdk1.4.0-javax-swing	56262	1127	777	68.94

Abbildung 5.4: Referenzen und Kandidaten der Projekte (nach 2 %)

In Abbildung 5.4 sind pro Projekt die Anzahl aller eingesendeten Kandidaten, die Anzahl der davon bewerteten und die Anzahl der hiervon in die Referenzmenge übernommenen Kandidaten (dies entspricht der Anzahl von Referenzen) aufgelistet. Die letzte Spalte setzt die zwei vorletzten Spalten in Relation zueinander:

$$\text{Ausbeute} = \frac{\text{Referenzen}}{\text{bewertet}} \cdot 100\%$$

Als erstes fällt auf, dass die Anzahl der Referenzen und auch die Anzahl der Kandidaten sich nicht streng proportional zur Größe der Projekte verhält (siehe dazu auch Abbildung 5.87 auf Seite 108). Der Wert Ausbeute gibt grob an, zu welchem Anteil beim Bewerten aus einem Kandidaten eine Referenz hervorgegangen ist. Hier sind projektspezifische

## 5.2. Auswertung nach Projekten

Unterschiede zu erkennen, die sich allerdings weder auf Projektgröße, noch auf Programmiersprache zurückführen lassen. Es handelt sich hierbei um die Art der Klone, die in dem jeweiligen Projekt vorkommen und wie gut diese von den Werkzeugen erkannt werden. Es lässt sich anhand dieser wenigen Projekte auch nicht pauschal feststellen, in welcher Programmiersprache mehr geklont wird.

Projekt	Rejected	bzgl. Kand. (%)	TrueNeg.Ref.	bzgl. Ref. (%)
weltab	33	11.79	3	1.19
cook	151	27.76	2	0.50
sns	352	26.49	4	0.44
postgresql	685	57.95	31	5.59
netbeans-javadoc	107	67.30	5	9.09
eclipse-ant	23	45.10	0	0.00
eclipse-jdtcore	624	33.62	28	2.08
j2sdk1.4.0-javax-swing	405	35.94	14	1.80

Abbildung 5.5: Verworfenen Kandidaten (Rejected) und nicht gefundene Referenzen (TrueNegatives) nach 2 % und OK-Auswertung

Projekt	Rejected	bzgl. Kand. (%)	TrueNeg.Ref.	bzgl. Ref. (%)
weltab	119	42.50	26	10.32
cook	265	48.71	58	14.43
sns	755	56.81	141	15.61
postgresql	886	74.96	183	32.97
netbeans-javadoc	133	83.65	18	32.73
eclipse-ant	28	54.90	2	6.67
eclipse-jdtcore	1167	62.88	395	29.37
j2sdk1.4.0-javax-swing	756	67.08	96	12.36

Abbildung 5.6: Verworfenen Kandidaten (Rejected) und nicht gefundene Referenzen (TrueNegatives) nach 2 % und Good-Auswertung

In den Abbildungen 5.5 und 5.6 soll exemplarisch der Unterschied von OK-Match(0.7) und Good-Match(0.7) veranschaulicht werden. Man sieht deutlich, dass die Anzahl der nicht akzeptierten Kandidaten und die Anzahl der nicht gefundenen Referenzen stark ansteigt, wenn man als Maß das strengere Good-Kriterium und nicht das schwächere OK-Kriterium zu Grunde legt. Wird Good-Match(0.7) betrachtet, so ist die Zahl der Kandidaten, die eine Referenz „gut genug“ treffen, wesentlich geringer. Dadurch werden auf der einen Seite mehr Kandidaten verworfen, auf der anderen Seite gibt es mehr Referenzen, die von keinem Kandidaten abgedeckt werden. Da die Kandidaten, die dem Good-Kriterium entsprechen, die qualitativ besseren darstellen, soll in den folgenden Abschnitten jeweils nur die Good-Auswertung betrachtet werden, und nur an ausgewählten Stellen wird ein Vergleich zu der OK-Auswertung gezogen.

## 5. Ergebnisanalyse

Bemerkenswert ist die sehr hohe Rate von „schlechten“ Kandidaten (Rejected) bezüglich der Good- und OK-Kriterien im Projekt netbeans-javadoc und die erstaunlich niedrige Rate im Projekt weltab. Bei beiden Projekten handelt es sich jeweils um die kleinsten Projekte der jeweiligen Programmiersprache im Experiment. Auch wenn man die anderen Projekte der zwei Programmiersprachen miteinander vergleicht, fällt auf, dass bei den vorgeschlagenen Kandidaten der Projekte in Java der Anteil der „schlechten“ Kandidaten wesentlich höher ist als bei den Projekten in C. Es scheint also so, als ob Klone in C besser erkannt werden können als in Java. Wahrscheinlicher ist jedoch, dass es sich hierbei um eine Eigenart der ausgewählten Projekte handelt.

Auch bei der Anzahl der nicht gefundenen Referenzen (TrueNeg.Ref.) ist Ähnliches zu beobachten. Bis auf das außerordentlich gute Ergebnis im Projekt eclipse-ant werden die Referenzen in den C-Projekten zu einem höheren Anteil erkannt als in den Java-Projekten. Die Tatsache, dass überhaupt so viele Referenzen nicht erkannt werden, wo doch die Referenzen mittels Orakel aus den Kandidaten gebildet werden, liegt daran, dass der Schiedsrichter einen „schlechten“ Kandidaten nicht als Referenz übernimmt, sondern ihn verbessert und erst dann in die Referenzmenge aufnimmt (siehe Abschnitt 4.1 auf Seite 39). Die Folge davon ist, dass diese Referenz von dem Kandidaten, aus dem sie hervorgegangen ist, nicht notwendigerweise mehr „gut genug“ überdeckt wird.

In den folgenden Abschnitten soll nun jeweils ein Projekt nach dem anderen betrachtet, die Ergebnisse der Werkzeuge bei diesem Projekt verglichen und Auffälligkeiten erwähnt werden.

### 5.2.1. weltab

weltab ist mit seinen 11K SLOC das kleinste Projekt der Programmiersprache C im Experiment. Es besteht aus 39 .c Dateien, wobei 37 davon eine eigene main() Funktion haben. Dies ist auch ein Grund, warum Krinke Probleme damit hatte und Duplix umgeschrieben werden musste (siehe Abschnitt 3.5.4 auf Seite 37).

**1. Kandidaten.** In Abbildung 5.7 auf der nächsten Seite ist zu erkennen, wie unterschiedlich die Anzahl der eingesandten Kandidaten zwischen den einzelnen Teilnehmern ist.

Interessant ist, dass Kamiyas „Kür“-Abgabe, welche ja im Gegensatz zur Pflicht-Abgabe „schlechte“ Kandidaten eliminiert, hier keinen Unterschied zur Pflicht-Abgabe macht.

**2. Referenzen.** Die großen Unterschiede in der Anzahl der gemeldeten Kandidaten lässt eine Betrachtung der tatsächlich überdeckten Referenzen umso interessanter erscheinen. In Abbildung 5.8 auf Seite 62 sind sowohl die getroffenen Referenzen nach der 1%-Auswertung als auch die nach der 2%-Auswertung zum Vergleich dargestellt. Wie man erkennen kann, ändert sich – grob gesagt – lediglich ein Faktor, der allen Werten gemein ist. Da dies (mit wenigen Ausnahmen) bei allen weiteren Projekten gleich ist, werden lediglich hier im Projekt weltab noch beide Auswertungen gegenübergestellt. In fast allen anderen Projekten werden nur noch die 2%-Daten abgebildet.

Wie bereits im Abschnitt 5.2 auf Seite 58 erwähnt, sinkt die Anzahl der getroffenen Referenzen, wenn man Good-Match(0.7) anstelle von OK-Match(0.7) als Kriterium verwendet. Dies ist sehr anschaulich in Abbildung 5.8 auf Seite 62 zu erkennen. Bei Krinke ist dieser

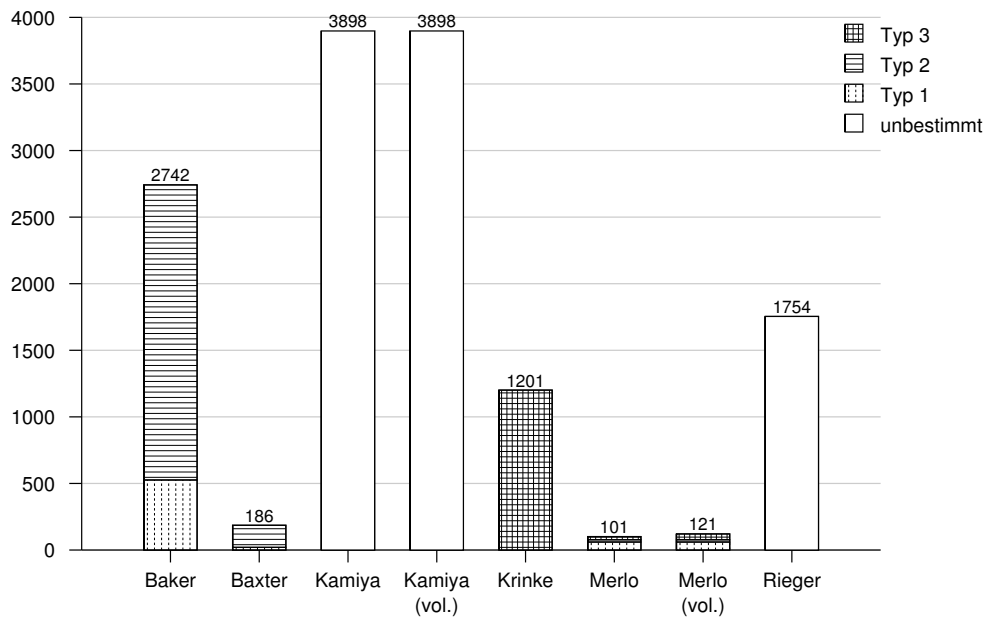


Abbildung 5.7: Anzahl der Kandidaten für das Projekt weltaab

Effekt am stärksten zu beobachten. Dies liegt daran, dass er ja Klone mit Löchern in der Mitte erkennt. Da dies in der Spezifikation des Experiments nicht berücksichtigt ist, meldet er den kompletten Klon. Daraus folgt, dass er nach dem OK-Kriterium eine Überdeckung mit vielen Referenzen hat. Betrachtet man allerdings Good-Match(0.7), so ist der overlap wegen der Löcher nicht „gut genug“.

In Abbildung 5.9 auf Seite 63 sind die Häufigkeiten der mehrfach erkannten Referenzen dargestellt. Es ist zu erkennen, dass es keine einzige Referenz in weltaab gibt, die von allen sechs Werkzeugen erkannt wird. Auf der anderen Seite gibt es 48 Referenzen, welche jeweils nur ein Teilnehmer findet. Der Schwerpunkt liegt bei doppelt und dreifach erkannten Referenzen.

In den Matrizen zu weltaab fällt auf, dass gemäß des Kriteriums Good-Match(0.7) zwischen zwei Kandidaten bei Baker insgesamt neun Kandidaten verworfen werden, die auch bei Rieger verworfen werden. D. h. dass von Baker und Rieger neun gleiche, „schlechte“ Kandidaten in der Bewertung als kein Klon erachtet wurden. Des Weiteren fällt auf, dass Baker und Kamiya 138 Referenzen gemeinsam finden und Baker, Kamiya und Rieger immerhin noch 82 Referenzen. Alle anderen entdecken nur 14 oder weniger Referenzen gemeinsam. Baxter, Krinke und Merlo finden recht wenige Klone, welche die anderen nicht auch finden. Baker, Kamiya und Rieger finden wesentlich mehr Klone, die andere Teilnehmer nicht finden (dies ist aus o. g. gemeinsam erkannten Referenzen auch zu folgern).

**3. FoundSecrets.** Von den 18 in weltaab versteckten Klonpaaren werden zwei Klone überhaupt nicht entdeckt. Dabei handelt es sich um zwei Strukturen, denen Elemente hinzuge-

## 5. Ergebnisanalyse

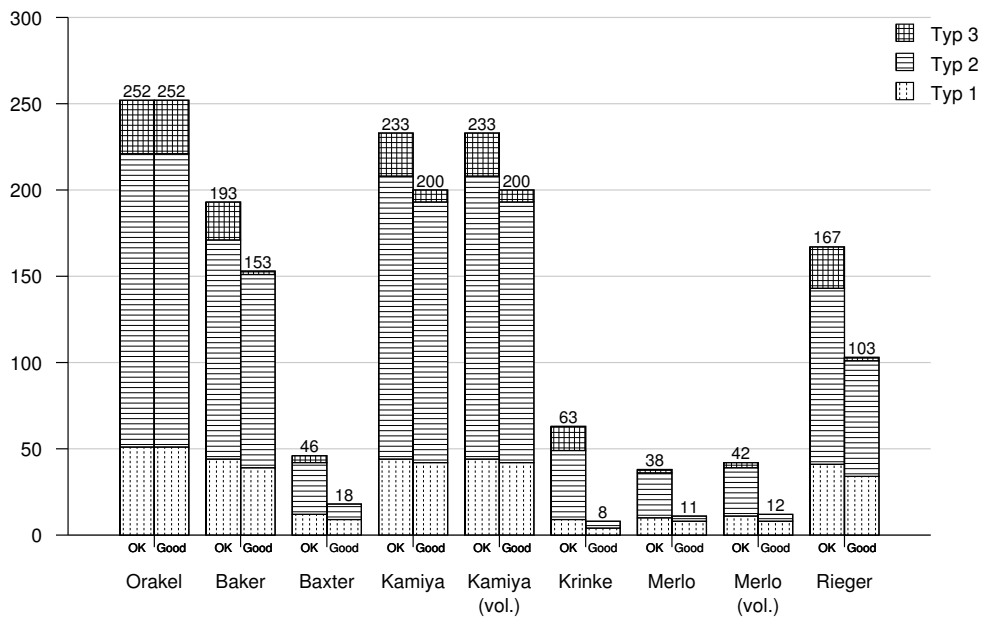
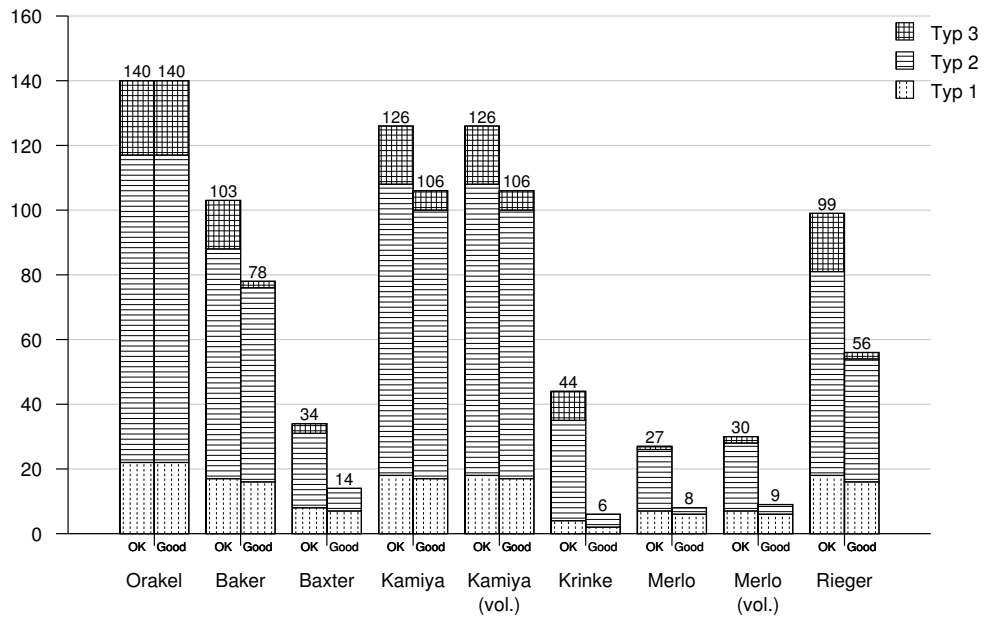


Abbildung 5.8: Getroffene Referenzen für das Projekt weltab  
(oben für 1 %, unten für 2 % bewerteter Kandidaten)

<i>N</i>	Referenzen
1	48
2	96
3	77
4	3
5	2

Abbildung 5.9: *N*-fach gefundene Referenzen von weltab (2 %, Good)

fügt und entfernt wurden. Eine genaue Auflistung der gefundenen Klonpaare ist in Abbildung 5.10 zu finden.

Baker	Baxter	Kamiya	Kamiya (vol.)	Krinke	Merlo	Merlo (vol.)	Rieger
6	7	10	10	0	3	3	6

Abbildung 5.10: FoundSecrets im Projekt weltab (2 %, Good) von 18

**4. Rejected.** Die zu Beginn für alle Projekte erwähnten Werte für verworfene Kandidaten werden hier für das Projekt weltab in die Werte der Teilnehmer aufgegliedert. Dies ist in Abbildung 5.11 zu sehen.

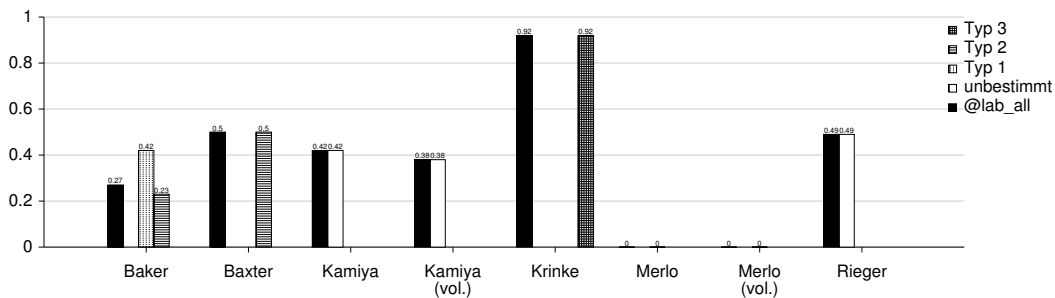


Abbildung 5.11: Rejected für das Projekt weltab (2 %, Good)

Auffällig ist nun zum einen, dass selbst bei der strengeren Good-Auswertung kein Kandidat von Merlo verworfen wurde (die Balken mit der Beschriftung „0“ sind vorhandene Balken, im Gegensatz zu leeren Plätzen im Diagramm). Zum anderen fällt auf, dass so gut wie jeder Kandidat von Krinke, der betrachtet wurde, auf keine Referenz trifft.

**5. TrueNegatives.** Nun liegt es nahe, zu schauen, wie viele der Referenzen von den Teilnehmern gefunden werden. Insbesondere interessiert, ob Merlo mit seinen hochwertigen Kandidaten auch möglichst alle Referenzen abdeckt.

In Abbildung 5.12 auf der nächsten Seite sieht man nun deutlich, dass Merlo weit davon entfernt ist, alle Klone zu erkennen. Genau genommen erkennt er nur rund 4 bzw. 5 % aller Klone. Bis auf Krinke und Baxter erkennen alle anderen Werkzeuge mehr Klonpaare. Die

## 5. Ergebnisanalyse

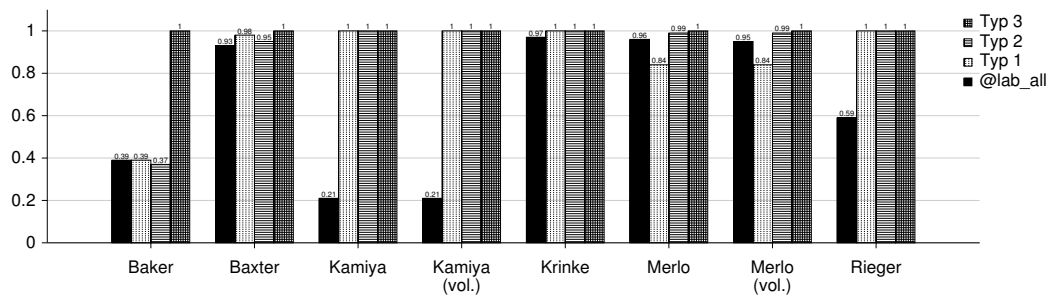


Abbildung 5.12: TrueNegatives für das Projekt weltab (2 %, Good)

Erkennung von Typ-1-Klonen scheint überdies bei Merlo deutlich besser zu sein als die Erkennung von Typ-2- und Typ-3-Klonen. Bei Baker und Baxter halten sich die Erkennung von Klonen vom Typ 2 und 3 in etwa die Waage.

**6. Recall und Precision.** In Abbildung 5.13 auf der nächsten Seite sind die Werte für Recall und Precision auf die jeweiligen Teilnehmer und Klontypen aufgesplittet dargestellt.

Es ist recht deutlich zu erkennen, dass Werkzeuge, die einen großen Recall haben, dafür mit niedrigerer Precision arbeiten. Werkzeuge, welche dafür eine hohe Precision haben, liefern einen niedrigeren Recall.

Die besonders hohe Precision von Baxter bei den Typ-1-Klonen lässt sich dadurch erklären, dass hier nur wenige Kandidaten eingesendet wurden und diese auch noch recht gut auf Referenzen treffen. Insgesamt ist seine Precision aber niedriger als die von Merlo, da Baxters Typ-2-Klone, die überwiegen, eine schlechtere Quote liefern.

**7. Klongrößen.** Für weltab ergeben sich die Klongrößen nach Abbildung 5.14 auf der nächsten Seite.

Bis auf Baker und Rieger, deren maximale Klone bei über 300 Zeilen liegen, befinden sich die größten Klone aller anderen Teilnehmer erstaunlich nahe beieinander. Betrachtet man jedoch die durchschnittliche Klongröße, sieht man recht schnell, dass die gefundenen Referenzen ansonsten unterschiedliche Größen haben, da die Durchschnittswerte stärker gestreut liegen. Kamiya hat mit knapp 16 Zeilen die kleinste Durchschnittsgröße und Krinke mit 122 Zeilen die größte. Baxter, Merlo und Rieger liegen wieder sehr nahe beieinander.

**8. Klonverteilung.** In Abbildung 5.15 auf Seite 66 sind die Vorkommnisse von gefundenen Referenzen, die innerhalb einer Datei bzw. über eine Datei hinweg geklont sind, aufgelistet. Der hohe Anteil über Dateien hinweg lässt sich recht einfach erklären: Wie bereits erwähnt, besteht weltab aus 39 Dateien, von denen 37 eine eigene `main()` Funktion besitzen. Viele dieser Dateien sind Klone voneinander.

**9. Verschiedenes.** Wie man in Abbildung 5.16 auf Seite 66 erkennen kann, finden Baker, Baxter, Krinke und Rieger jeweils einige Referenzen, die kein anderes Werkzeug entdeckt. Allerdings gibt es auch eine Referenz, die bis auf Krinke alle anderen Werkzeuge gemeinsam erkennen.

## 5.2. Auswertung nach Projekten

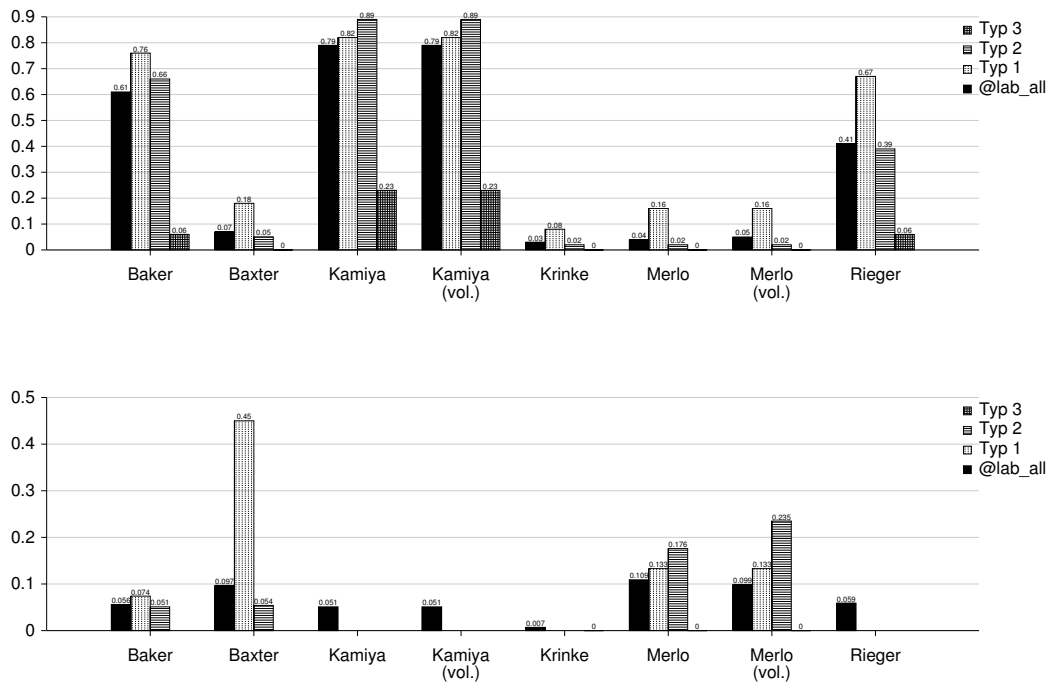


Abbildung 5.13: Recall (oben) und Precision (unten) für das Projekt weltab (2 %, Good)

Teilnehmer	MaxCloneSize	AvgCloneSize	StdDevCloneSize
Baker	322	28.03	63.00
Baxter	193	45.72	62.16
Kamiya	192	15.92	22.65
Kamiya (vol.)	192	15.92	22.65
Krinke	166	122.05	52.16
Merlo	189	50.27	68.67
Merlo (vol.)	189	58.17	70.95
Rieger	385	45.16	96.64
Orakel	391	39.47	86.74

Abbildung 5.14: Größen der Codefragmente von weltab (2 %, Good)

## 5. Ergebnisanalyse

Teilnehmer	InnerFilePairs	in %	AcrossFilePairs	in %
Baker	15	9.80	138	90.20
Baxter	3	16.67	15	83.33
Kamiya	19	9.50	181	90.50
Kamiya (vol.)	19	9.50	181	90.50
Krinke	0	0.00	8	100.00
Merlo	1	9.09	10	90.91
Merlo (vol.)	1	8.33	11	91.67
Rieger	7	6.80	96	93.20
Orakel	24	9.52	228	90.48

Abbildung 5.15: Gefundene Referenzen innerhalb von Dateien und über Dateien hinweg von weltab (2 %, Good)

Teilnehmer	OnlyPairs	OnlyButOnePairs	OverlappingCandidates
Baker	2	0	0
Baxter	2	0	0
Kamiya	0	0	82
Kamiya (vol.)	0	0	82
Krinke	2	1	0
Merlo	0	0	0
Merlo (vol.)	0	0	0
Rieger	5	0	4

Abbildung 5.16: Weitere Werte von weltab (2 %, Good)

Von den 3898 Kandidaten, die Kamiya eingesendet hat, sind 82 davon sich selbst überlappende Klonpaare, die somit von vornherein ausgeschlossen sind, da solche Klone nicht in die Referenzmenge übernommen wurden. Rieger hat auch vier Überlappungen; alle anderen Teilnehmer liefern jedoch überlappungsfreie Kandidaten.

### 5.2.2. cook

Das zweitkleinste C-Projekt im Experiment ist cook mit 80K SLOC. Es besteht aus 295 .c Dateien. Da es doch um einiges größer ist als weltab, liegt es nahe, dass mehr Klone darin vorhanden sind und entdeckt werden.

**1. Kandidaten.** Abbildung 5.17 auf der nächsten Seite zeigt, dass die Teilnehmer in der Tat mehr Kandidaten als im Projekt weltab finden.

Es erstaunt allerdings, dass Kamiya als einziger Teilnehmer weniger Kandidaten als bei weltab findet. Ansonsten sind die Verhältnisse der eingesandten Kandidaten der Teilnehmer untereinander weitgehend gleich.

Auch im Projekt cook bestehen keine außerordentlichen Unterschiede zwischen der Auswertung nach 1 % bewerteter Kandidaten und der Auswertung nach 2 %. Bei allen Teil-

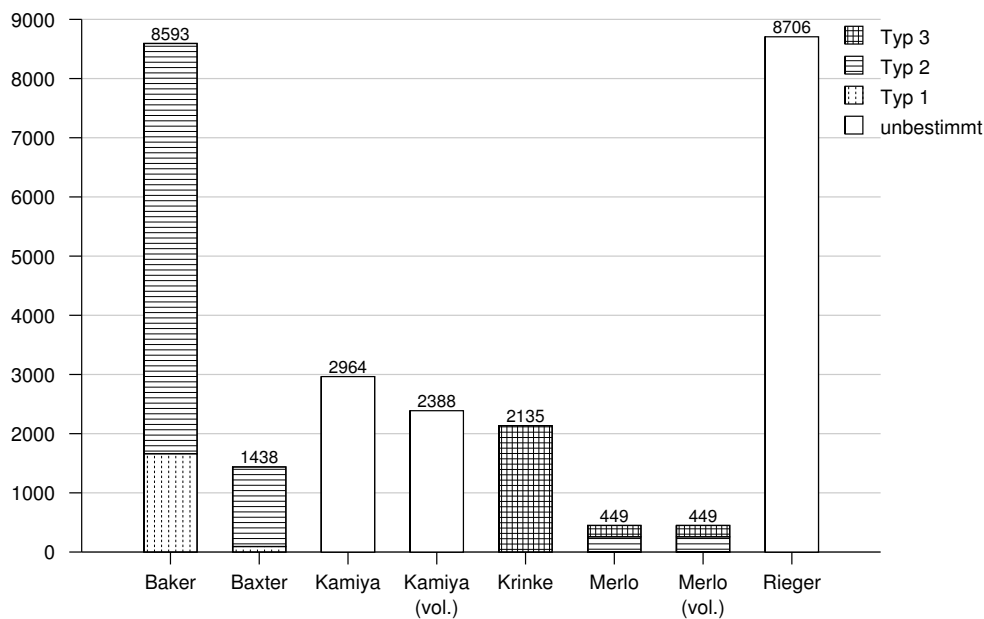


Abbildung 5.17: Anzahl der Kandidaten für das Projekt cook

nehmern entwickelt sich die Zahl der getroffenen Referenzen fast linear, wie dies auch bei weltab beobachtet werden kann. Eine leichte Abnahme ist zu verzeichnen, da – je mehr Referenzen vorliegen – die Teilnehmer mit ihren Kandidaten immer häufiger auf schon vorhandene Referenzen treffen, die nicht mehr hinzugefügt werden müssen. Im Weiteren soll daher auch hier nur die Auswertung mit 2 % und Good-Match(0.7) berücksichtigt werden.

**2. Referenzen.** In Abbildung 5.18 auf der nächsten Seite werden die gefundenen Referenzen der Teilnehmer einander gegenüber gestellt. Baker und Rieger, welche die weitaus meisten Kandidaten eingesendet haben, treffen auch die größte Anzahl von Referenzen. Bemerkenswert ist, wie Merlo mit nur einem Fünftel so vielen Kandidaten wie Krinke sogar mehr Referenzen trifft als dieser.

Abbildung 5.19 auf der nächsten Seite zeigt die Anzahl mehrfach gefundener Klone an. Im Gegensatz zu weltab existieren in cook wesentlich mehr Referenzen, die nur von einem Teilnehmer entdeckt werden.

Betrachtet man wieder die Matrizen, um nach Gemeinsamkeiten zu suchen, fällt auf, dass Baker und Rieger mit 122 gemeinsam erkannten Referenzen und mit neun gemeinsam erkannten „falschen“ Kandidaten die größte Übereinstimmung haben. Interessant ist weiterhin, dass von den 180 von Rieger gefundenen Referenzen nur drei ebenfalls von Merlo gefunden werden.

**3. FoundSecrets.** In cook sind vor dem Experiment je ein Klon von jedem Typ von Hand eingebaut worden. Es werden alle versteckten Referenzen von mindestens einem Werkzeug gefunden. Die Aufteilung ist in Abbildung 5.20 auf der nächsten Seite zu sehen.

## 5. Ergebnisanalyse

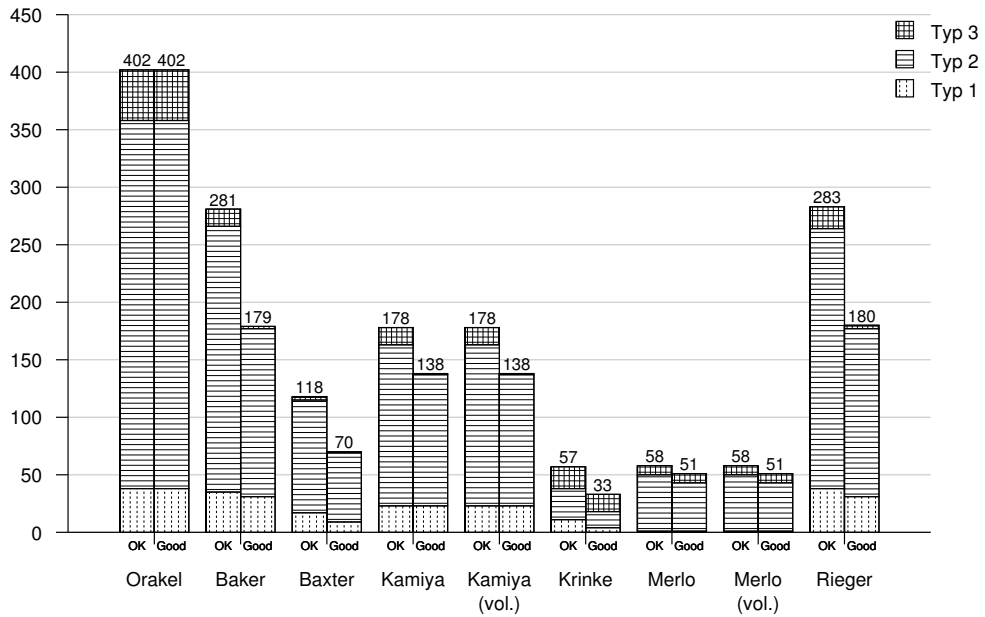


Abbildung 5.18: Getroffene Referenzen für das Projekt cook (2 %)

<i>N</i>	Referenzen
1	150
2	105
3	70
4	14
5	5

Abbildung 5.19: *N*-fach gefundene Referenzen von cook (2 %, Good)

Baker	Baxter	Kamiya	Kamiya (vol.)	Krinke	Merlo	Merlo (vol.)	Rieger
2	1	2	2	1	1	1	1

Abbildung 5.20: FoundSecrets im Projekt cook (2 %, Good) von 3

**4. Rejected.** Die beim Bewerten verworfenen Kandidaten sind ins Verhältnis zu der Gesamtzahl der bewerteten Kandidaten der einzelnen Teilnehmer gesetzt. Dies ist in Abbildung 5.21 dargestellt.

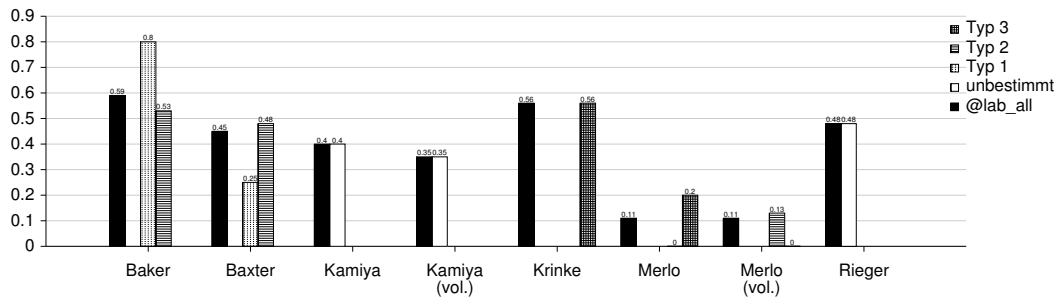


Abbildung 5.21: Rejected für das Projekt cook (2 %, Good)

Auffällig ist, dass sich die Werte von Baxter, Kamiya und Rieger kaum von denen des Projektes weltab unterscheiden. Krinke schneidet hier deutlich besser ab, dennoch wurde mehr als jeder zweite Kandidat von ihm beim Bewerten verworfen. Baker hat die schlechteste Ausbeute, Merlo wie bereits bei weltab die beste.

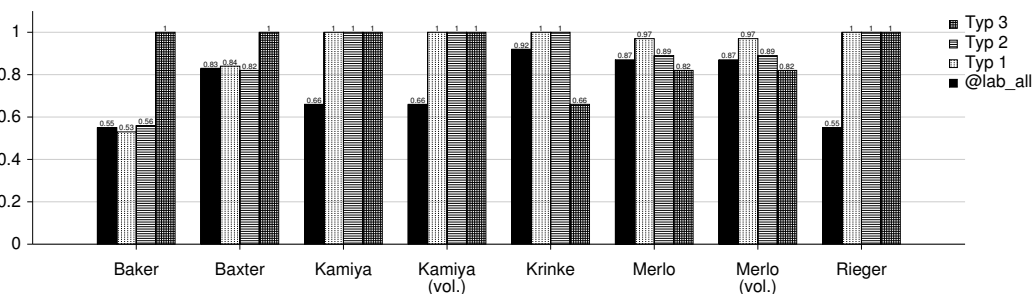


Abbildung 5.22: TrueNegatives für das Projekt cook (2 %, Good)

**5. TrueNegatives.** Beim Anteil der nicht gefundenen Referenzen, den Abbildung 5.22 illustriert, sind Krinke, Merlo und Rieger nahezu gleich wie im ersten Projekt. Kamiya hingegen findet im Verhältnis nur sehr viel weniger Referenzen als im Vorprojekt. Dies scheint die Konsequenz davon zu sein, dass er als einziger Teilnehmer bei diesem Projekt weniger Kandidaten eingesendet hat als bei weltab.

**6. Recall und Precision.** Die bisher für cook gefundenen Erkenntnisse sollen nun durch die bekannten Werte von Recall und Precision in Abbildung 5.23 auf der nächsten Seite bestätigt werden.

## 5. Ergebnisanalyse

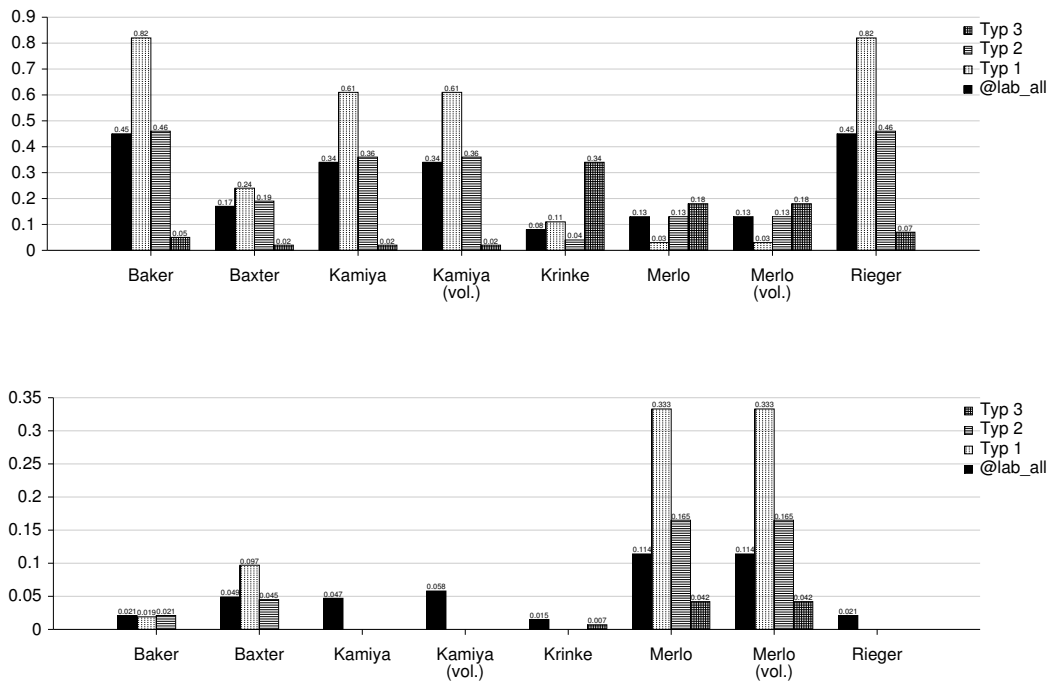


Abbildung 5.23: Recall (oben) und Precision (unten) für das Projekt cook (2 %, Good)

Baker und Rieger haben mit den meisten Kandidaten auch den höchsten Recall. Kamiyas Recall-Werte sind aufgrund oben erwähnter Tatsache auch niedriger als im letzten Projekt. Interessant ist jedoch, dass Merlo bei weltab einen relativ hohen Recall bei Typ-1-Klonen hat und dieser zu Typ-3-Klonen hin abnimmt. Hier ist genau das Gegenteil der Fall: Der Recall bei seinen Typ-1-Klonen ist extrem schlecht, dafür ist er bei Typ-2- und Typ-3-Klonen wesentlich besser.

Bei der Precision zeigt sich wieder das gehabte Bild: Je größer der Recall, desto kleiner die Precision und umgekehrt. Dies wird gerade wieder bei Merlo deutlich, der bei den Typ-1-Klonen mit jedem dritten Klonpaar eine Referenz trifft.

**7. Klongrößen.** Bei den Klongrößen in Abbildung 5.24 auf der nächsten Seite erstaunen die extrem großen Unterschiede der maximalen Klone. Im Vergleich zu weltab sind die Durchschnittsgrößen relativ nahe beieinander, wohingegen sich die Standardabweichungen doch auch wieder stärker unterscheiden. Baxter und Merlo finden demnach bei cook fast ausschließlich kleine Klone, während Baker viel größere Referenzen findet.

**8. Klonverteilung.** Die Aufteilung der gefundenen Referenzen, die innerhalb von Dateien und über Dateien hinweg auftreten, ist in Abbildung 5.25 auf der nächsten Seite dargestellt.

## 5.2. Auswertung nach Projekten

Teilnehmer	MaxCloneSize	AvgCloneSize	StdDevCloneSize
Baker	352	30.46	77.84
Baxter	28	9.37	4.24
Kamiya	207	18.85	23.95
Kamiya (vol.)	207	18.86	23.95
Krinke	99	17.75	25.23
Merlo	26	10.04	3.39
Merlo (vol.)	26	10.04	3.39
Rieger	109	10.43	8.88
Orakel	645	22.41	60.70

Abbildung 5.24: Größen der Codefragmente von cook (2 %, Good)

Teilnehmer	InnerFilePairs	in %	AcrossFilePairs	in %
Baker	40	22.35	139	77.65
Baxter	21	30.00	49	70.00
Kamiya	32	23.19	106	76.81
Kamiya (vol.)	32	23.19	106	76.81
Krinke	2	6.06	31	93.94
Merlo	8	15.69	43	84.31
Merlo (vol.)	8	15.69	43	84.31
Rieger	36	20.00	144	80.00
Orakel	80	19.90	322	80.10

Abbildung 5.25: Gefundene Referenzen innerhalb von Dateien und über Dateien hinweg von cook (2 %, Good)

## 5. Ergebnisanalyse

Der Anteil der Klone innerhalb von Dateien ist bei cook höher als bei weltab. Es fällt jedoch auf, dass Krinke auch hier extrem wenig Klonpaare innerhalb von Dateien findet.

Teilnehmer	OnlyPairs	OnlyButOnePairs	OverlappingCandidates
Baker	32	0	243
Baxter	16	0	0
Kamiya	0	0	262
Kamiya (vol.)	0	0	193
Krinke	21	2	0
Merlo	0	0	0
Merlo (vol.)	0	0	0
Rieger	46	1	165

Abbildung 5.26: Weitere Werte von cook (2 %, Good)

**9. Verschiedenes.** Wie bei weltab zeigt sich auch bei cook in Abbildung 5.26, dass Kamiya und Merlo keine Referenzen finden, die kein anderer Teilnehmer entdeckt. Alle anderen finden einige Klonpaare als einzige.

Baxter, Krinke und Merlo melden auch hier keine sich überlappenden Kandidaten. Interessant ist, dass Kamiyas verbesserte Implementierung im „Kür“-Lauf tatsächlich einen großen Teil dieser unerwünschten Kandidaten entfernt.

### 5.2.3. snns

Das zweitgrößte der C-Projekte ist snns mit insgesamt 115K SLOC und 141 .c Dateien. 43K SLOC davon befinden sich in einem Sub-System, welches auf X-Bibliotheken aufbaut. Dieses kann von Krinke nicht analysiert werden, da er noch keine speziellen Header-Dateien für X-Bibliotheken erstellt hat.

**1. Kandidaten.** Von den Verhältnissen der eingesendeten Kandidaten der Teilnehmer untereinander zeigt sich wieder ein ähnliches Bild wie bereits bei weltab. Gegenüber cook sind von Baker, Baxter und Merlo in etwa gleich viele Kandidaten vorhanden. Rieger findet weniger Kandidaten, bei Kamiya und Krinke allerdings explodieren die Kandidaten förmlich: Obwohl das Projekt nicht einmal doppelt so groß ist wie cook, werden etwa sechsmal so viele Kandidaten gefunden (siehe Abbildung 5.27 auf der nächsten Seite).

**2. Referenzen.** In Abbildung 5.28 auf der nächsten Seite zeigt sich erneut, dass Baker, Kamiya und Rieger die meisten Referenzen finden. Schön zu sehen ist in diesem Projekt (wie auch schon andeutungsweise im letzten), wie qualitativ gut Merlos Kandidaten sind, welche überhaupt eine Referenz treffen: Diese überlappen gleich so gut, dass es fast immer ein Good-Match(0.7) ist. Er hat in seinen zwei Abgaben hier nur sehr wenige Klonpaare, die ein OK-Match(0.7), aber kein Good-Match(0.7) sind. Dies liegt darin begründet, dass er hauptsächlich ganze Funktionen als Klone findet. Diese sind dann, sofern sie überhaupt mit einer Referenz überlappen, gleich sehr gute Matches.

In Abbildung 5.29 auf Seite 74 erkennt man, dass ein großer Teil aller Referenzen von jeweils nur einem Teilnehmer gefunden wird. Aber auch die Menge der Referenzen, die

## 5.2. Auswertung nach Projekten

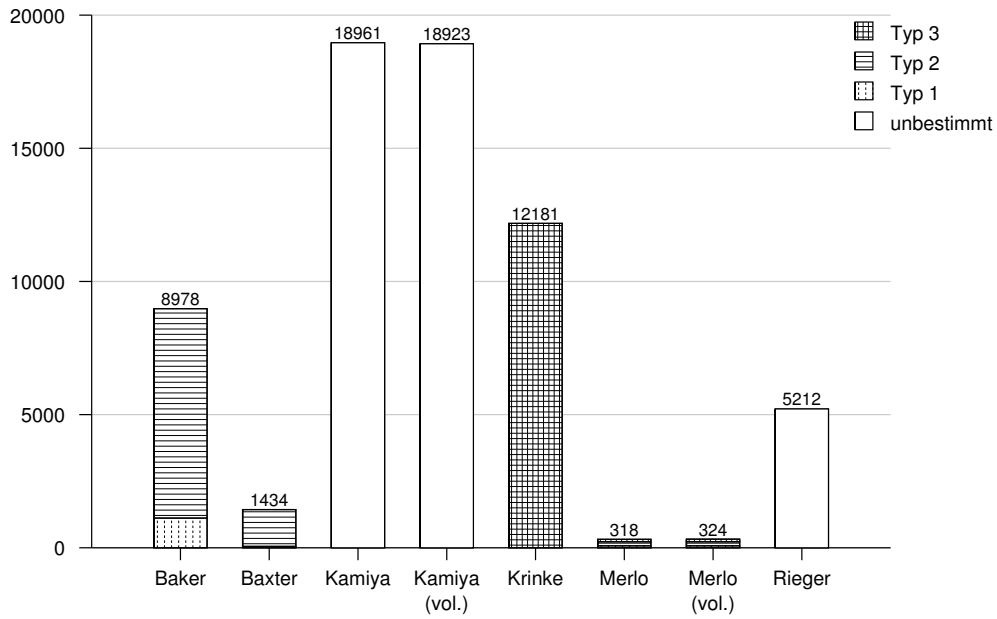


Abbildung 5.27: Anzahl der Kandidaten für das Projekt snns

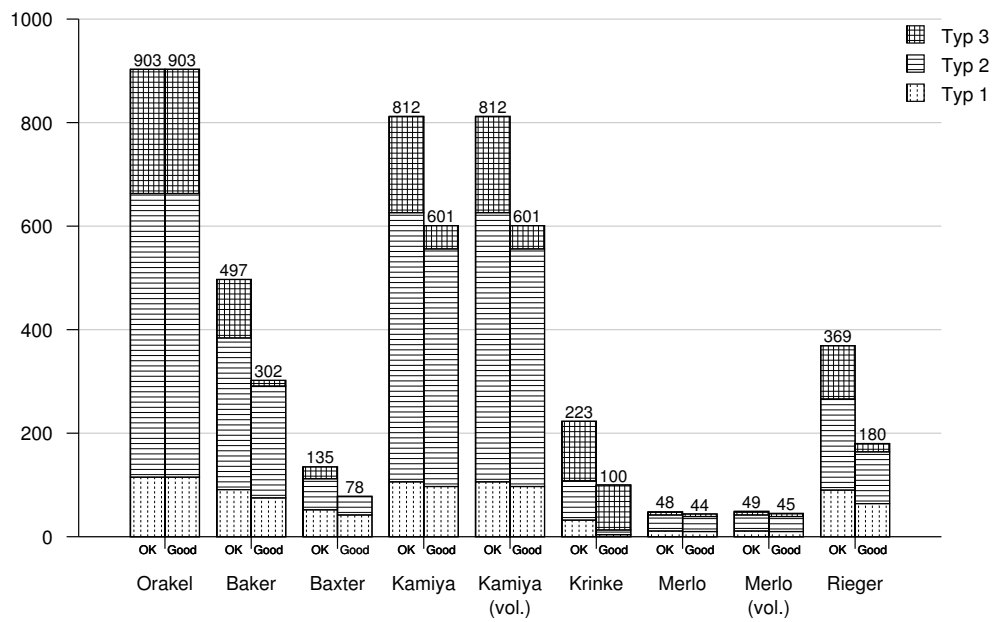


Abbildung 5.28: Getroffene Referenzen für das Projekt snns (2 %)

## 5. Ergebnisanalyse

<i>N</i>	Referenzen
1	419
2	190
3	110
4	35
5	6
6	1

Abbildung 5.29: *N*-fach gefundene Referenzen von snns (2 %, Good)

zwei Werkzeuge finden, ist größer als bei den zwei ersten Projekten. Es gibt sogar eine Referenz, die von allen Teilnehmern gefunden wird.

Betrachtet man die Matrizen, so fällt auf, dass Baker und Kamiya 261 Referenzen gemeinsam entdecken. Baker, Kamiya und Rieger erkennen immerhin noch 130 Referenzen gemeinsam. Mit 67 Klonen zwischen Baxter und Kamiya ist bereits ein größerer Abstand vorhanden. Alle anderen entdecken nur unter 50 gemeinsame Klonpaare. Baxter und Krinke sowie Baxter und Merlo sogar nur jeweils vier. Des Weiteren erkennt Merlo nur vier Referenzen, die Kamiya nicht auch erkennt.

**3. FoundSecrets.** In snns ist jeweils ein Typ-1- und ein Typ-3-Klon versteckt worden. Diesmal werden beide Klone gefunden (siehe Abbildung 5.30). Beide Klone befinden sich in den von Krinke analysierten Sub-Systemen, so dass die Tatsache, dass er ein Sub-System von snns nicht analysieren konnte, hier keine Rolle spielt.

Baker	Baxter	Kamiya	Kamiya (vol.)	Krinke	Merlo	Merlo (vol.)	Rieger
1	1	2	2	0	1	1	1

Abbildung 5.30: FoundSecrets im Projekt snns (2 %, Good) von 2

**4. Rejected.** Baker und Krinke liegen, was verworfene Kandidaten betrifft, zwischen ihren Ergebnissen von *wel*tab und *cook*. Bei Kamiya und Rieger musste wieder in etwa jeder zweite Kandidat verworfen werden. Dies war bei den ersten zwei Projekten nahezu identisch. Wie man in Abbildung 5.31 auf der nächsten Seite aber sieht, haben Baxter und Merlo die beste Ausbeute. Die zwei Balken mit Höhe 1 bei den Typ-1-Klonen von Baxter und Merlo sind darauf zurückzuführen, dass von beiden jeweils nur ein Kandidat bewertet worden ist. Daher ist das Ergebnis für die Typ-1-Klone der beiden nicht repräsentativ. Die gute Ausbeute erkennt man am guten Abschneiden der Typ-2-Klone bei Baxter und dem exzellenten Ergebnis der Typ-2- und Typ-3-Klone bei Merlo. Daraus resultiert dann auch das gute Gesamtergebnis der zwei Teilnehmer.

**5. TrueNegatives.** Die Anzahl der Referenzen, die ein Teilnehmer nicht findet, bezogen auf die Gesamtzahl der Referenzen, ist in Abbildung 5.32 auf der nächsten Seite veranschaulicht.

Merlo findet die wenigsten Referenzen, Kamiya zwei Drittel und damit die meisten. Baxter und Krinke finden nur jede 10. Referenz, Rieger immerhin doppelt so viele. Allerdings

## 5.2. Auswertung nach Projekten

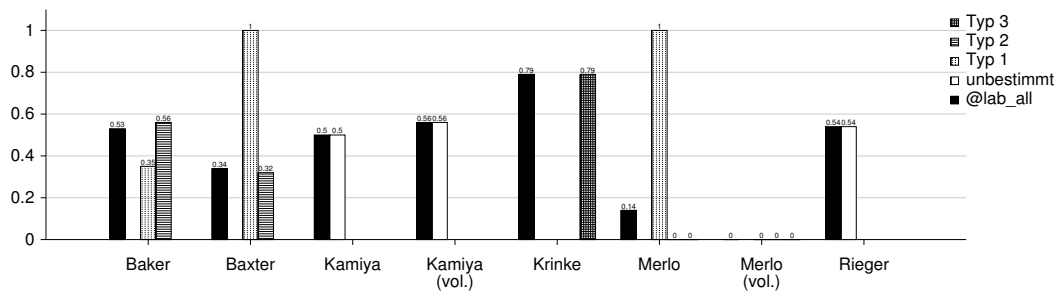


Abbildung 5.31: Rejected für das Projekt snns (2 %, Good)

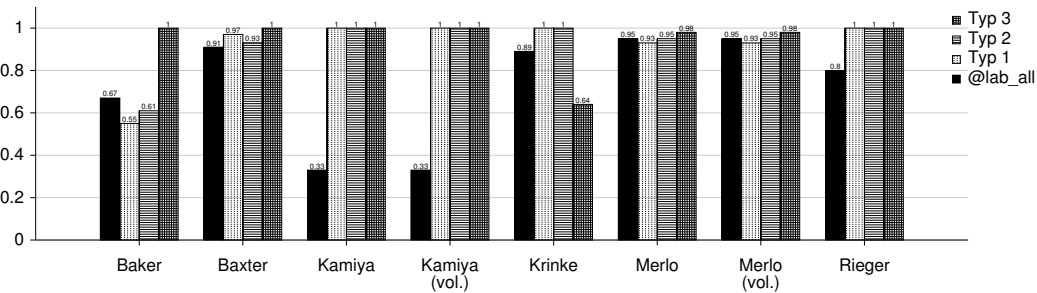


Abbildung 5.32: TrueNegatives für das Projekt snns (2 %, Good)

ist Riegers Ausbeute bei den zwei vorigen Projekten erheblich besser. Baker fand anteilig in weltab und cook ebenfalls mehr Referenzen, dennoch liegt sie bei snns mit einem Drittel gefundener Referenzen hinter Kamiya auf dem zweiten Platz.

**6. Recall und Precision.** Anhand der Werte für Recall und Precision in Abbildung 5.33 auf der nächsten Seite lassen sich bisherige Erkenntnisse wieder bestärken.

An der Aufschlüsselung der zwei Drittel von Kamiya gefundenen Referenzen erkennt man seine Schwächen im Bereich der Typ-3-Klone. Berücksichtigt man jedoch, dass er diese Klontypen eigentlich nur in sehr eingeschränktem Maße erkennen kann, so ist der Recall beachtlich! Ganz im Gegensatz zu Kamiya hat Krinke seine Stärke gerade im Erkennen dieser Klone vom Typ 3. Sein Recall von über einem Drittel erkannter Referenzen dieses Typs wird von keinem anderen Teilnehmer überboten. Bei Baker, Rieger und Baxter ist die Erkennung von Referenzen des Typ 1 am ausgeprägtesten. Auffällig ist jedoch Baxters schlechtes Abschneiden insgesamt. Nur Merlo findet noch weniger Klone.

Abbildung 5.33 auf der nächsten Seite zeigt, dass Baxter eine außerordentlich hohe Precision bei Typ-1-Klonen hat: Von 52 Kandidaten dieses Typs sind alle 52 je ein OK-Match(0.7) mit Referenzen. Immerhin treffen noch 42 Kandidaten auf Referenzen, wenn man Good-Match(0.7) betrachtet. Jedoch hat wieder einmal Merlo mit Abstand die höchsten Precision-Werte, wenn man alle Klontypen gemeinsam betrachtet.

## 5. Ergebnisanalyse

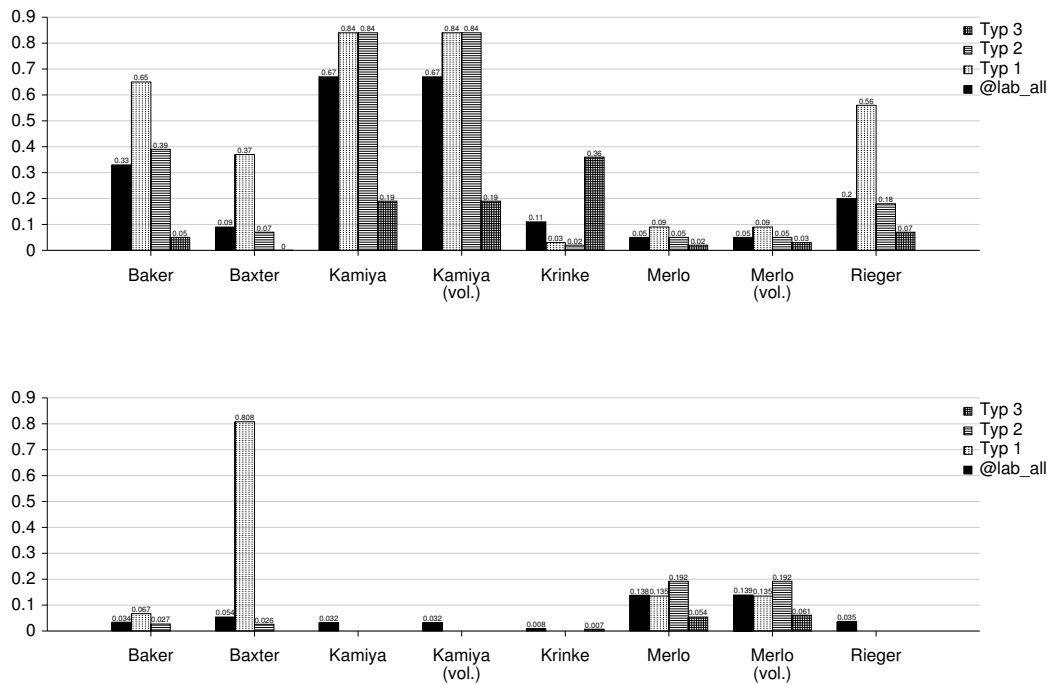


Abbildung 5.33: Recall (oben) und Precision (unten) für das Projekt snns (2 %, Good)

**7. Klongrößen.** An den Größen der gefundenen Referenzen in Abbildung 5.34 auf der nächsten Seite fällt auf, dass Baker und Rieger immer mit die größten Klone finden. Betrachtet man allerdings die Durchschnittswerte, so findet Krinke, wie bereits bei cook, mit Abstand die größten Referenzen. Dies liegt daran, dass er gerade Typ-3-Klone sucht und diese sehr oft aus großen Codefragmenten bestehen. Er fällt auch mit seiner großen Standardabweichung aus dem Rahmen. Die der anderen Teilnehmer liegt im Bereich von 20 – 30, seine bei 77 Zeilen.

**8. Klonverteilung.** Betrachtet man die Verteilung der Klonpaare auf gleiche oder unterschiedliche Dateien in Abbildung 5.35 auf der nächsten Seite, so fällt auf, dass bei snns die Verteilung der Klone eher innerhalb von Dateien vorhanden ist. Bis auf Kamiya und Merlo, die noch etwas mehr dateiübergreifende Klonpaare finden, ist bei allen anderen Teilnehmern ein deutliches Gefälle hin zu Klonen innerhalb der gleichen Datei zu sehen.

**9. Verschiedenes.** Wie man in Abbildung 5.36 auf Seite 78 sieht, gibt es wieder Klone, die alle Teilnehmer bis auf Krinke und Rieger finden. Andererseits gibt es auch wieder Klonpaare, die nur ein Teilnehmer findet. Nur Kamiya und Merlo im Pflicht-Teil finden keine Klone, die kein anderer findet.

## 5.2. Auswertung nach Projekten

Teilnehmer	MaxCloneSize	AvgCloneSize	StdDevCloneSize
Baker	239	13.16	19.02
Baxter	130	16.30	19.51
Kamiya	149	18.66	24.76
Kamiya (vol.)	149	18.66	24.76
Krinke	349	122.19	77.57
Merlo	95	20.88	19.14
Merlo (vol.)	95	20.84	18.93
Rieger	218	20.80	31.05
Orakel	1064	22.30	48.20

Abbildung 5.34: Größen der Codefragmente von snns (2 %, Good)

Teilnehmer	InnerFilePairs	in %	AcrossFilePairs	in %
Baker	210	69.54	92	30.46
Baxter	56	71.79	22	28.21
Kamiya	281	46.76	320	53.24
Kamiya (vol.)	281	46.76	320	53.24
Krinke	57	57.00	43	43.00
Merlo	22	50.00	22	50.00
Merlo (vol.)	22	48.89	23	51.11
Rieger	139	77.22	41	22.78
Orakel	439	48.62	464	51.38

Abbildung 5.35: Gefundene Referenzen innerhalb von Dateien und über Dateien hinweg von snns (2 %, Good)

## 5. Ergebnisanalyse

Teilnehmer	OnlyPairs	OnlyButOnePairs	OverlappingCandidates
Baker	24	0	335
Baxter	8	0	0
Kamiya	0	0	1129
Kamiya (vol.)	0	0	1120
Krinke	81	5	0
Merlo	0	0	0
Merlo (vol.)	1	0	0
Rieger	21	1	254

Abbildung 5.36: Weitere Werte von snns (2 %, Good)

Betrachtet man die sich überlappenden Klone, erhält man wieder das gleiche Ergebnis: Baxter, Krinke und Merlo liefern ausschließlich sich nicht überlappende Kandidaten, die anderen haben auch Überlappungen in ihren Einsendungen. Kamiyas optimierte Version von CCFinder hat hier allerdings nur wenige der sich überlappenden Klone eliminiert.

### 5.2.4. postgresql

Mit 235K SLOC ist postgresql das größte der Projekte im Experiment und besteht aus 322 .c Dateien. Da Krinke und Rieger es aufgrund seiner Größe nicht analysieren konnten, fehlt der Vergleich zu diesen zwei Teilnehmern in diesem Abschnitt.

**1. Kandidaten.** In Abbildung 5.37 auf der nächsten Seite fällt auf, dass die Menge der eingesandten Kandidaten der Teilnehmer von den Größenverhältnissen untereinander her ähnlich ist wie beim Projekt snns. Alle Teilnehmer finden etwas mehr Klone, Merlo verdreifacht sogar die Anzahl seiner Kandidaten. Insgesamt werden aber – gemessen an der Größe des Projektes – weniger Kandidaten gefunden als in snns.

**2. Referenzen.** Betrachtet man die in Abbildung 5.38 auf der nächsten Seite dargestellte Anzahl von Referenzen, so fällt auf, dass postgresql fast nur halb so viele Klone enthält wie snns, obwohl es fast doppelt so groß ist. Im Vergleich fällt nun auf, dass Baker und Kamiya in etwa nur halb so viele Referenzen treffen wie im Projekt snns, was proportional zur Gesamtzahl der Referenzen wäre. Baxter trifft hier etwas mehr, aber Merlo steigert seine getroffenen Referenzen auf fast das Doppelte.

Da sowohl Krinke als auch Rieger postgresql nicht analysiert haben, fällt in Abbildung 5.39 auf Seite 80 auf, dass die vier verbleibenden Teilnehmer immerhin 11 Referenzen gemeinsam entdecken. Die Anzahl der Referenzen, die nur von einem Werkzeug gefunden werden, ist mit 201 wieder am höchsten.

Schaut man sich die Matrizen für dieses Projekt an, so fällt wieder auf, dass Baker und Kamiya viele Klone gemeinsam erkennen, nämlich 121 Stück. Baker und Merlo hingegen finden nur 15 Referenzen gemeinsam.

**3. FoundSecrets.** In postgresql sind zwei Klone vom Typ 2 versteckt worden. Beide werden, wie Abbildung 5.40 auf Seite 80 belegt, von den Teilnehmern entdeckt.

## 5.2. Auswertung nach Projekten

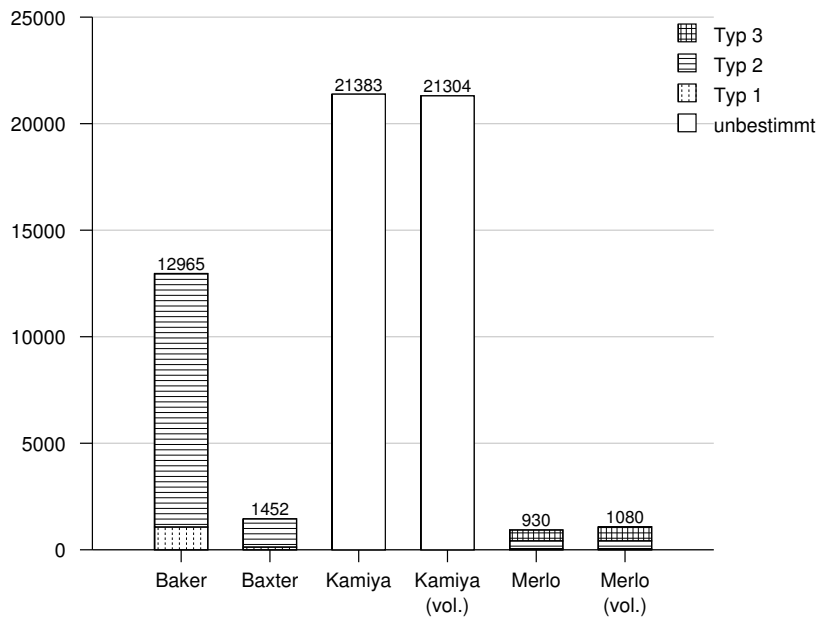


Abbildung 5.37: Anzahl der Kandidaten für das Projekt postgresql

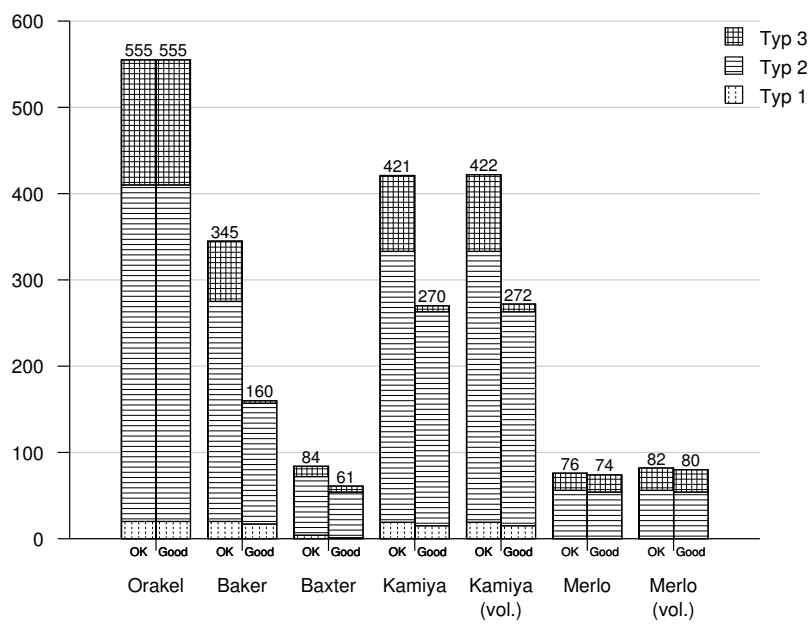


Abbildung 5.38: Getroffene Referenzen für das Projekt postgresql (2 %)

## 5. Ergebnisanalyse

<i>N</i>	Referenzen
1	201
2	136
3	16
4	11

Abbildung 5.39: *N*-fach gefundene Referenzen von postgresql (2 %, Good)

Baker	Baxter	Kamiya	Kamiya (vol.)	Merlo	Merlo (vol.)
1	0	2	2	1	1

Abbildung 5.40: FoundSecrets im Projekt postgresql (2 %, Good) von 2

**4. Rejected.** In Abbildung 5.41 erkennt man, dass extrem viele Kandidaten von Baker und Kamiya beim Bewerten verworfen worden sind. Diese Zahlen sind hier in der Tat repräsentativ, da es sich um 348 von 427 betrachteten Kandidaten bei Kamiya und um 192 von 259 betrachteten Kandidaten bei Baker handelt und somit nicht nur um einige wenige „Ausreißer“. Bei Baxter musste nur jeder fünfte Kandidat verworfen werden. Bei Merlo gar nur jeder 10. bzw. jeder 20. Kandidat. Bedenkt man, dass es sich hierbei hauptsächlich um Typ-3-Klone handelt, so zeigt dies, wie qualitativ „sauber“ die erkannten Klone von Merlo sind.

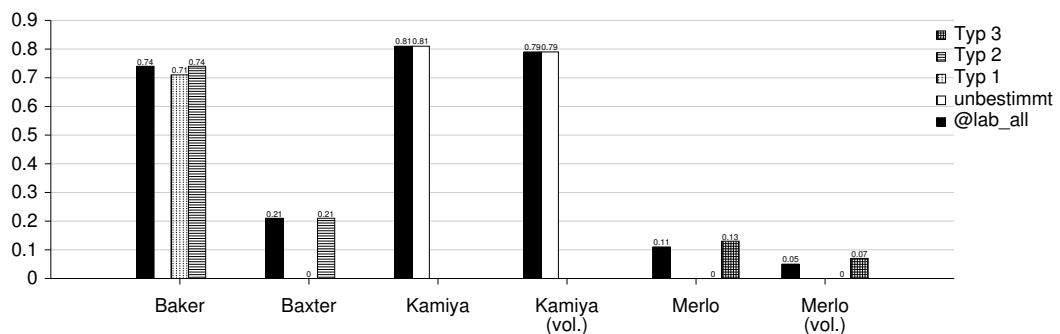


Abbildung 5.41: Rejected für das Projekt postgresql (2 %, Good)

**5. TrueNegatives.** Trotz seines schlechten Ergebnisses, was die verworfenen Kandidaten betrifft, schafft es Kamiya, jede zweite Referenz zu entdecken, wie Abbildung 5.42 auf der nächsten Seite zeigt. Baker, von der 26 % aller bewerteten Kandidaten verworfen werden mussten, findet immerhin noch 29 % aller Referenzen. Baxter und Merlo liegen mit jeweils nur etwas über 10 % erkannter Referenzen zurück.

**6. Recall und Precision.** Die Tendenz der vorigen Werte bestätigt sich wieder einmal in den Werten für Recall und Precision (siehe Abbildung 5.43 auf der nächsten Seite).

## 5.2. Auswertung nach Projekten

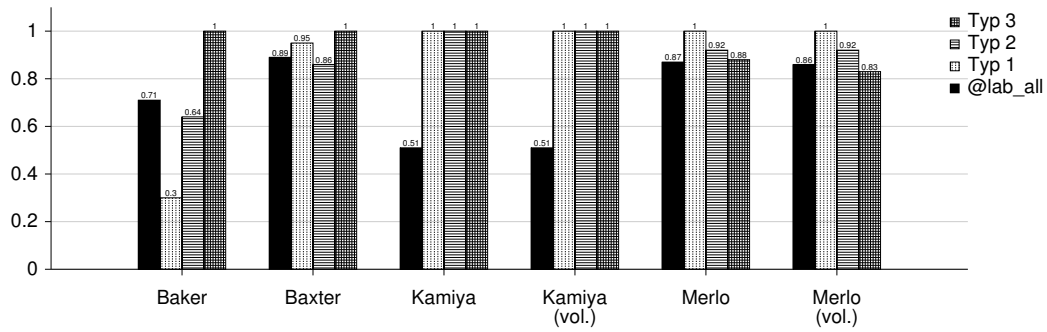


Abbildung 5.42: TrueNegatives für das Projekt postgresql (2 %, Good)

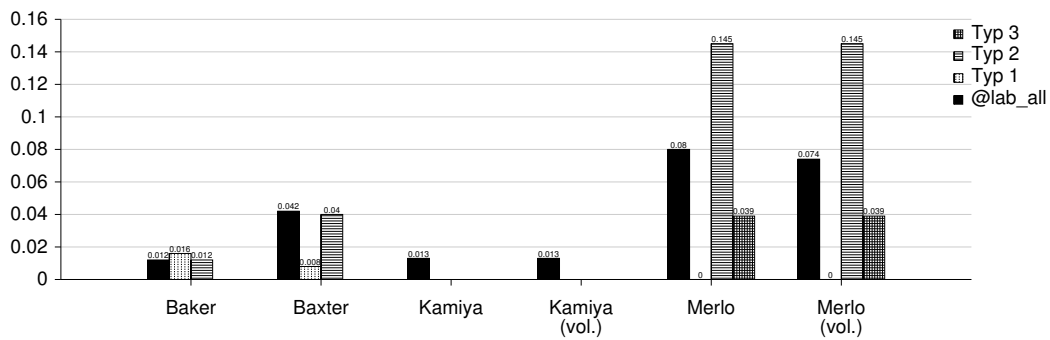
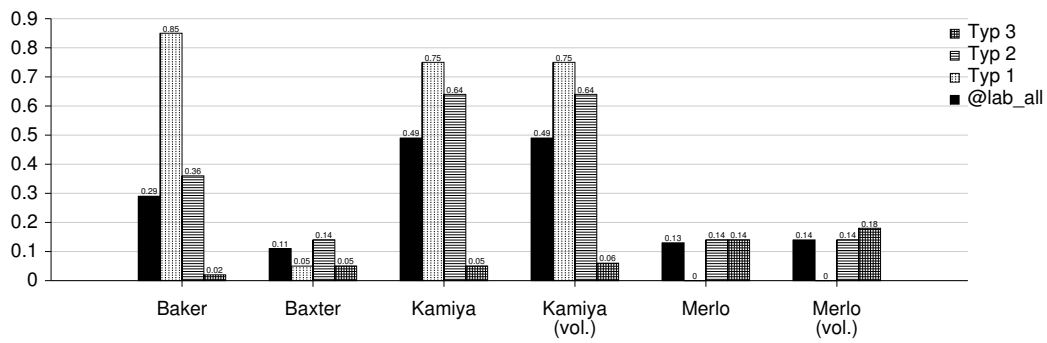


Abbildung 5.43: Recall (oben) und Precision (unten) für das Projekt postgresql (2 %, Good)

## 5. Ergebnisanalyse

Trotz der Tatsache, dass 80 % von Kamiyas Kandidaten verworfen werden mussten, entdeckt er, wie bereits erwähnt, noch jede zweite Referenz. Betrachtet man nur Typ-1-Klone, so findet er drei Viertel aller Klone dieses Typs und noch knapp zwei Drittel aller Klone vom Typ 2. Bakers Erkennung vom Typ 1 ist allerdings noch ausgeprägter, sie findet 85 %. Baxter und Merlo finden nur etwas mehr als jeden 10. Klon. Es muss jedoch bemerkt werden, dass, auf Typ-3-Klone bezogen, Merlo am besten abschneidet. Er findet 14 % in seinem Pflicht-Teil und 18 % aller Typ-3-Referenzen in der „Kür“. Allerdings war er nicht in der Lage, auch nur eine einzige exakte Kopie zu entdecken.

Schaut man sich die Werte der Precision an, so fällt auf, dass Merlos Typ-2-Kandidaten die höchste Precision erbringen. Baxter erreicht nur eine halb so hohe Precision wie Merlo, sie ist aber immerhin noch knapp viermal so hoch wie bei Baker und Kamiya. Diese brauchen immerhin 77 bzw. 83 Kandidaten, um eine Überdeckung mit einer Referenz zu erreichen.

Teilnehmer	MaxCloneSize	AvgCloneSize	StdDevCloneSize
Baker	323	14.00	27.77
Baxter	44	11.40	7.47
Kamiya	171	13.59	13.50
Kamiya (vol.)	171	13.63	13.47
Merlo	91	13.14	11.69
Merlo (vol.)	91	12.86	11.29
Orakel	322	14.08	21.01

Abbildung 5.44: Größen der Codefragmente von postgresql (2 %, Good)

**7. Klongrößen.** Trotz der relativ großen maximalen Klongröße bei Baker und Kamiya sind die Klone von postgresql im Durchschnitt um die 13 Zeilen lang, wie man in Abbildung 5.44 sieht. Bei der durchschnittlichen Größe sind sich die Teilnehmer erstaunlich einig. Die Standardabweichung spiegelt hier die maximalen Klongrößen wider, da die Durchschnittsgrößen bereits sehr klein sind. Im Vergleich mit den anderen C-Projekten scheint postgresql die kleinsten geklonten Einheiten zu haben.

Teilnehmer	InnerFilePairs	in %	AcrossFilePairs	in %
Baker	141	88.13	19	11.88
Baxter	55	90.16	6	9.84
Kamiya	220	81.48	50	18.52
Kamiya (vol.)	222	81.62	50	18.38
Merlo	58	78.38	16	21.62
Merlo (vol.)	64	80.00	16	20.00
Orakel	470	84.68	85	15.32

Abbildung 5.45: Gefundene Referenzen innerhalb von Dateien und über Dateien hinweg von postgresql (2 %, Good)

**8. Klonverteilung.** In Abbildung 5.45 auf der vorherigen Seite sieht man auch sehr deutlich, dass die vorhandenen Klone fast alle Klone sind, deren Kopie innerhalb derselben Datei liegt. Auch hier sind sich die Teilnehmer wieder relativ einig.

Die Klongrößen und die Verteilung der Klone auf die Dateien lassen darauf schließen, dass es in postgresql einige Dateien gibt, in denen eine große Anzahl von geklonten Funktionen auftreten. Z. B. sind in der Datei `postgresql/src/backend/nodes/equalfuncs.c` eine große Anzahl von Vergleichs-Funktionen definiert, die von den Teilnehmern zum Teil als geklont angesehen werden.

Teilnehmer	OnlyPairs	OnlyButOnePairs	OverlappingCandidates
Baker	34	9	513
Baxter	20	4	0
Kamiya	0	0	1029
Kamiya (vol.)	2	0	1006
Merlo	0	0	0
Merlo (vol.)	6	0	0

Abbildung 5.46: Weitere Werte von postgresql (2 %, Good)

**9. Verschiedenes.** Auch in Abbildung 5.46 zeigt sich das nun bisher vertraute Bild. Kamiya und Merlo finden weniger und im Pflicht-Teil gar keine Klone, die sonst niemand zweites findet. Es gibt jedoch neun Referenzen, die Baxter, Kamiya und Merlo alle finden, aber Baker nicht.

Baxter und Merlo liefern keine sich überlappenden Klone, Baker und Kamiya hingegen tun dies: jeder 25. bzw. 21. Kandidat hat sich überlappende Codefragmente.

### 5.2.5. netbeans-javadoc

Mit 19K SLOC ist netbeans-javadoc das kleinste Projekt der Programmiersprache Java im Experiment und besteht aus 101 Dateien. Der Vergleich mit Krinke fehlt in diesem Abschnitt, da er mit Duplix keine Java-Projekte analysieren kann.

**1. Kandidaten.** An der Anzahl der eingesendeten Kandidaten in Abbildung 5.47 auf der nächsten Seite fällt zunächst die große Diskrepanz zwischen Kamiya und den anderen Teilnehmern auf. Weiter fällt auf, dass Kamiyas Modifikation an `CCFinder` viele der Kandidaten zu eliminieren scheint. Aber dennoch sind es über viermal so viele wie z. B. von Baker. Baxter meldet mit nur 33 Kandidaten so wenig wie in keinem anderen Projekt im Experiment.

**2. Referenzen.** Obwohl das Projekt fast doppelt so groß ist wie `weltab`, sind nur gut ein Fünftel so viele Referenzen darin enthalten, wie man in Abbildung 5.48 auf der nächsten Seite erkennen kann. Der erste Eindruck suggeriert nun, dass dies an der Programmiersprache liegen muss.

Äußerst auffällig ist auch die Tatsache, dass Kamiya mit seiner Zweiteinsendung sogar etwas besser ist als mit seinem Pflicht-Teil, obwohl er nur etwas mehr als ein Viertel so viele

## 5. Ergebnisanalyse

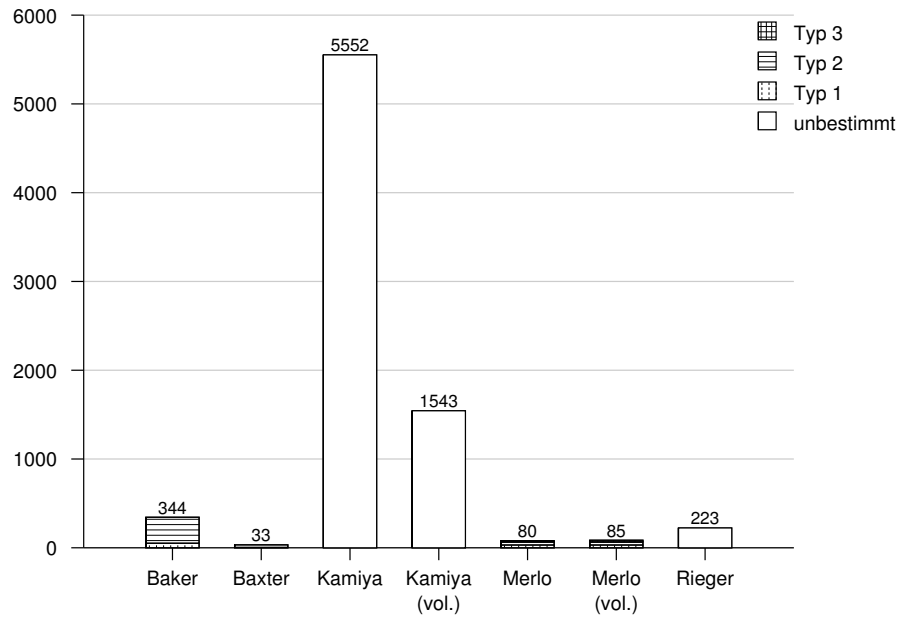


Abbildung 5.47: Anzahl der Kandidaten für das Projekt netbeans-javadoc

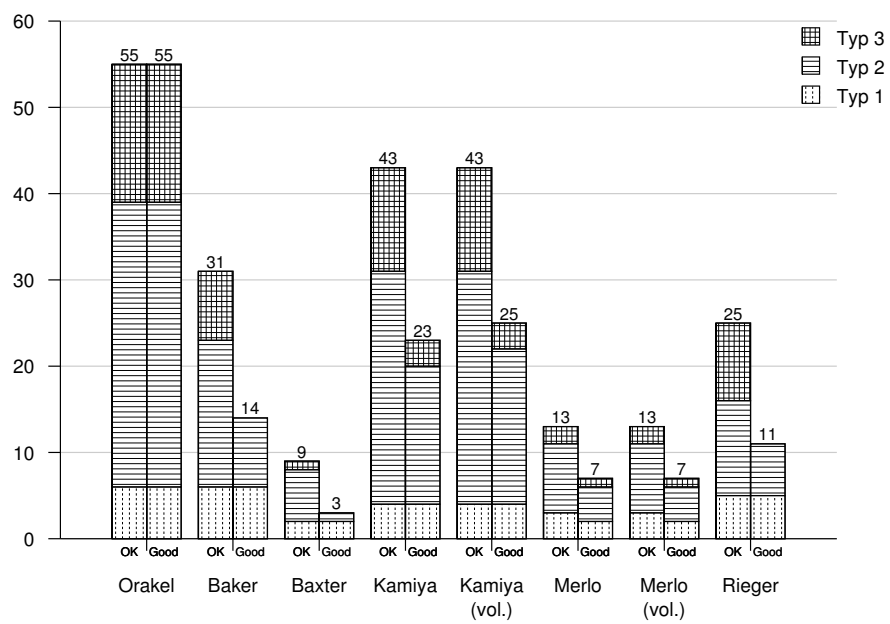


Abbildung 5.48: Getroffene Referenzen für das Projekt netbeans-javadoc (2 %)

Kandidaten in seiner „Kür“ hat. Dies heißt, dass seine verbesserte Variante von `CCFinder` in der Tat 4009 höchstwahrscheinlich allesamt uninteressante Kandidaten entfernt hat. Zumindest hat sich keiner dieser Kandidaten mit einer Referenz überdeckt. Merlo hingegen hat mit den fünf weiteren Kandidaten seiner „Kür“-Einsendung keinen Treffer gelandet.

<i>N</i>	Referenzen
1	23
2	8
3	2
4	2
5	1

Abbildung 5.49: *N*-fach gefundene Referenzen von `netbeans-javadoc` (2 %, Good)

In Abbildung 5.49 ist die Anzahl der von mehreren Werkzeugen gefundenen Referenzen dargestellt. Der eindeutige Schwerpunkt liegt auch hier wieder auf einfach gefundenen Referenzen. Aber es wird eine Referenz von allen fünf Teilnehmern entdeckt, die das Projekt analysiert haben.

Da es bei `netbeans-javadoc` relativ wenig Klone gibt, bringt die Betrachtung der Matrizen aus Schnittmengen und Differenzen keine besonders interessanten Erkenntnisse. Baker und Kamiya sowie Baker und Rieger haben jeweils acht Referenzen gemeinsam gefunden. Die Tatsache, dass Kamiya extrem viele Kandidaten eingesendet hat, relativiert die Feststellung, dass bei ihm neun Kandidaten verworfen wurden, die auch bei Rieger nicht akzeptiert wurden, und fünf, die auch bei Baxter verworfen wurden.

**3. FoundSecrets.** In `netbeans-javadoc` sind insgesamt 16 Klone versteckt worden. Vier davon sind vom Typ 1 und jeweils sechs vom Typ 2 und Typ 3. Fünf von den 16 Klonpaaren werden nicht gefunden. Dabei handelt es sich um einen Typ-2-Klon, der innerhalb einer Datei kopiert ist und aus einer Sequenz von Anweisungen mit veränderten Parametern besteht. Drei weitere nicht gefundene Klone sind ebenfalls innerhalb einer Datei kopiert und vom Typ 3. Dabei handelt es sich jeweils um eine kopierte Funktion, Struktur und Anweisungssequenz mit kleinen Veränderungen. Der letzte nicht gefundene Klon ist auch vom Typ 3 und besteht aus einer Sequenz von Anweisungen, die in eine andere Datei kopiert und leicht verändert ist. Die Auflistung der gefundenen Klonpaare ist in Abbildung 5.50 zu sehen.

Baker	Baxter	Kamiya	Kamiya (vol.)	Merlo	Merlo (vol.)	Rieger
6	2	5	5	2	2	4

Abbildung 5.50: FoundSecrets im Projekt `netbeans-javadoc` (2 %, Good) von 16

**4. Rejected.** Da, um die Schwelle von 2 % bewerteter Kandidaten zu erhalten, bei Baxter nur ein einziger Kandidat bewertet werden musste und bei Merlo jeweils zwei Kandidaten, sind die Werte für Baxter und Merlo in der Abbildung 5.51 auf der nächsten Seite nicht repräsentativ. Bei Rieger wurden fünf Kandidaten bewertet, bei Baker sieben und bei Kamiya 31 bzw. 111. Zumindest die Werte für Kamiya können daher durchaus als repräsentativ angesehen werden, aber auch die von Baker und Rieger zeigen eine gewisse Tendenz.

## 5. Ergebnisanalyse

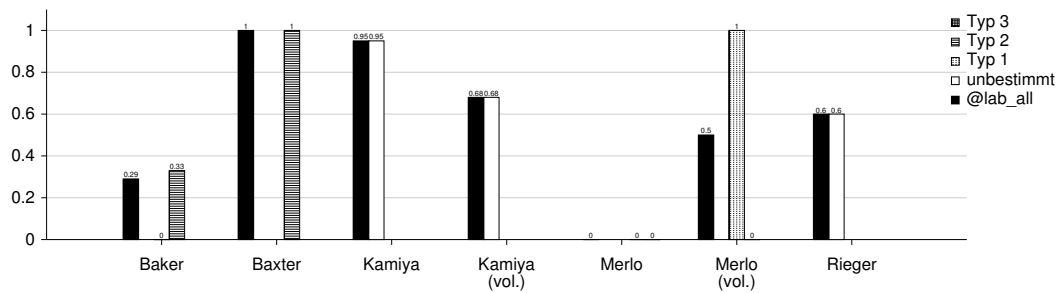


Abbildung 5.51: Rejected für das Projekt netbeans-javadoc (2 %, Good)

Man erkennt, wie Kamiyas Ausbeute durch seine überarbeitete Version von CCFinder besser wird: statt 95 % verworfener Kandidaten wurden nur noch 68 % nicht akzeptiert.

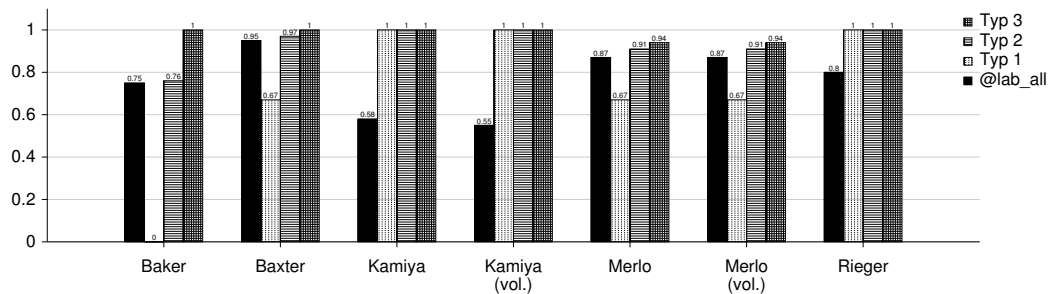


Abbildung 5.52: TrueNegatives für das Projekt netbeans-javadoc (2 %, Good)

**5. TrueNegatives.** Abbildung 5.52 ist wieder unabhängig von der Anzahl der bewerteten Kandidaten. Daher können hier wieder alle Teilnehmer in die Betrachtung einbezogen werden.

Auffällig ist, dass Baker alle sechs in netbeans-javadoc vorkommenden Referenzen vom Typ 1 findet. Kamiya findet wieder einmal den größten Anteil an Klonen, diesmal ist es aber Baxter, der den kleinsten Teil findet. Das liegt natürlich sehr stark daran, dass er mit nur 33 Kandidaten keine 55 Referenzen mit Good-Match(0.7) überdecken kann.

**6. Recall und Precision.** Wie bereits erwähnt, sieht man in Abbildung 5.53 auf der nächsten Seite noch einmal sehr deutlich, dass Baker alle Referenzen vom Typ 1 findet. Auch Rieger hat fünf davon gefunden, Kamiya immerhin noch vier, Baxter und Merlo nur noch jeweils zwei. Da es 33 Referenzen vom Typ 2 und 16 Referenzen vom Typ 3 gibt, sind deren Werte etwas aussagekräftiger. Generell scheinen die Typ-3-Klone in netbeans-javadoc schwieriger zu entdecken zu sein als die von den anderen Projekten. Vom Typ 2 erkennt Kamiya die Hälfte, die anderen Teilnehmer nur ein Viertel und weniger. Besonders niedrig ist hier Baxters Recall.

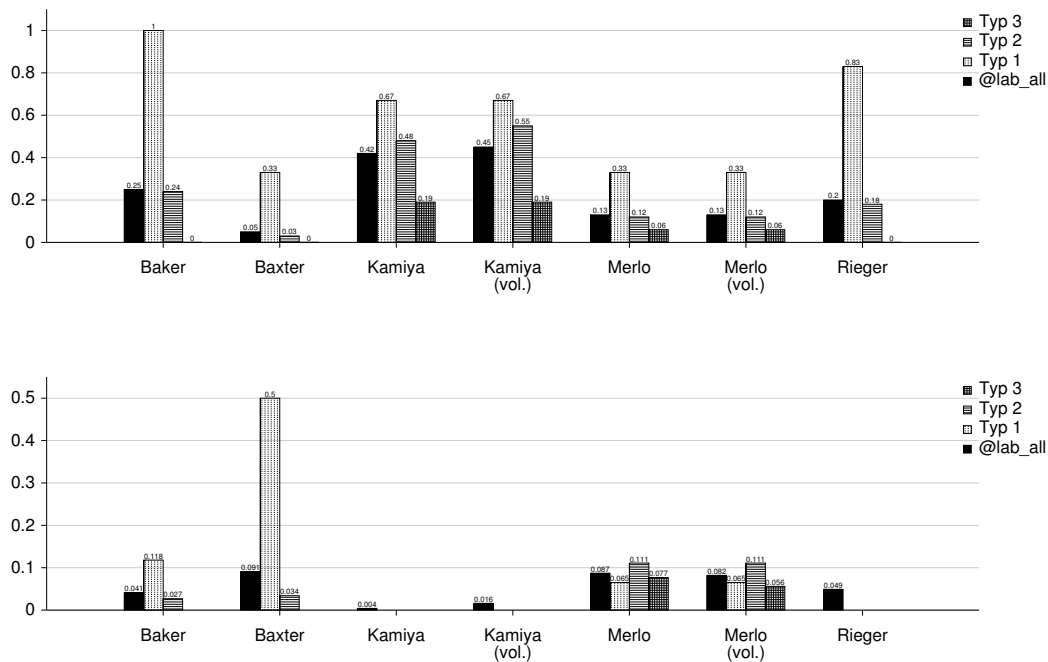


Abbildung 5.53: Recall (oben) und Precision (unten) für das Projekt netbeans-javadoc (2 %, Good)

Der besonders hohe Wert bei Baxters Precision für Typ-1-Klone liegt daran, dass er nur vier Kandidaten vom Typ 1 hat, von denen zwei jeweils eine Referenz treffen. Betrachtet man die Precision, ohne sie in die Klontypen aufzuschlüsseln, so sind es wieder Baxter und Merlo, die am wenigsten Kandidaten brauchen, um eine Referenz zu treffen. An Kamiyas Precision-Wert von 0.004 lässt sich noch einmal deutlich erkennen, wie „schlecht“ die Kandidaten seines unmodifizierten CCFinder hier sind: Auf eine Überdeckung mit einer Referenz kommen 250 Kandidaten ohne Überdeckung.

**7. Klongrößen.** In Abbildung 5.54 auf der nächsten Seite sieht man, wie bei allen Teilnehmern bis auf Merlo die maximale Klongröße sich etwa im gleichen Rahmen bewegt. Auch bei der Durchschnittgröße sind sich die meisten einig, nur Baxter findet im Schnitt größere Referenzen als die anderen Teilnehmer.

**8. Klonverteilung.** Die Frage, ob die Codefragmente der gefundenen Referenzen überwiegend in derselben Datei geklont sind oder über Dateien hinweg, lässt sich mit Abbildung 5.55 auf der nächsten Seite beantworten. Man sieht wieder deutlich, dass es viel weniger Klone innerhalb von Dateien gibt und mehr Klone über Dateien hinweg. Baxters 100 % sind allerdings aufgrund der niedrigen Anzahl gefundener Referenzen nicht repräsentativ.

## 5. Ergebnisanalyse

Teilnehmer	MaxCloneSize	AvgCloneSize	StdDevCloneSize
Baker	41	13.93	11.10
Baxter	43	23.67	17.82
Kamiya	48	16.69	10.75
Kamiya (vol.)	48	16.96	10.40
Merlo	22	13.00	5.05
Merlo (vol.)	22	13.00	5.05
Rieger	41	15.64	10.76
Orakel	122	22.95	24.14

Abbildung 5.54: Größen der Codefragmente von netbeans-javadoc (2 %, Good)

Teilnehmer	InnerFilePairs	in %	AcrossFilePairs	in %
Baker	5	35.71	9	64.29
Baxter	0	0.00	3	100.00
Kamiya	10	43.48	13	56.52
Kamiya (vol.)	10	40.00	15	60.00
Merlo	3	42.86	4	57.14
Merlo (vol.)	3	42.86	4	57.14
Rieger	1	9.09	10	90.91
Orakel	17	30.91	38	69.09

Abbildung 5.55: Gefundene Referenzen innerhalb von Dateien und über Dateien hinweg von netbeans-javadoc (2 %, Good)

Teilnehmer	OnlyPairs	OnlyButOnePairs	OverlappingCandidates
Baker	2	0	13
Baxter	1	1	0
Kamiya	0	0	358
Kamiya (vol.)	1	0	240
Merlo	0	0	0
Merlo (vol.)	0	0	0
Rieger	1	0	8

Abbildung 5.56: Weitere Werte von netbeans-javadoc (2 %, Good)

**9. Verschiedenes.** Betrachtet man die Referenzen, die nur ein Teilnehmer allein erkennt, so fällt wieder auf, dass Merlo und Kamiya im Pflicht-Teil keine Referenzen als einzige erkennen. Baxter hat es überdies versäumt, eine Referenz zu erkennen, die alle anderen finden.

Baxter und Merlo sind die einzigen, die wieder keine sich überlappenden Kandidaten melden (siehe Abbildung 5.56 auf der vorherigen Seite).

### 5.2.6. eclipse-ant

Das mit 35K SLOC in 178 Dateien zweitgrößte Java-Projekt im Experiment ist eclipse-ant. Auch hier ist der Vergleich mit Krinke nicht möglich, da er mit DupliX keine Java-Programme analysieren kann.

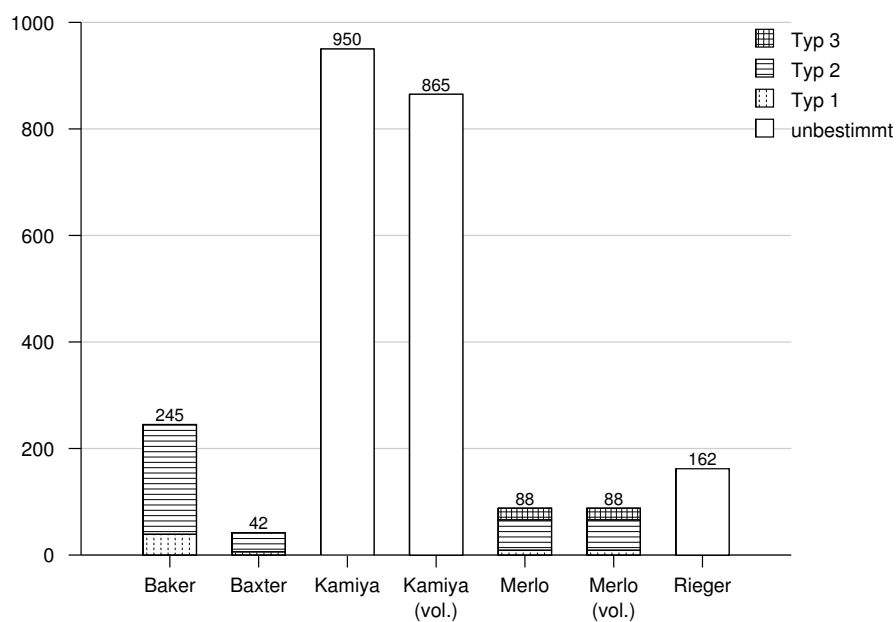


Abbildung 5.57: Anzahl der Kandidaten für das Projekt eclipse-ant

**1. Kandidaten.** Als erstes fällt einem in Abbildung 5.57 ins Auge, dass in diesem Projekt so wenig Kandidaten eingesendet wurden, wie in keinem der anderen Projekte. Kamiya hat mit nur knapp 1000 Kandidaten zwar noch die meisten in diesem Projekt, aber doch am wenigsten im Vergleich zu seinen anderen Einsendungen. Wie man schon fast erwartet, liegen Baker und Rieger auf den Plätzen zwei und drei. Baxter hat mit nur 42 – wie schon bei netbeans-javadoc – die wenigsten Kandidaten eingesendet.

**2. Referenzen.** Für dieses Projekt müssen die Daten der Auswertung nach 1 % sowie die Daten der Auswertung nach 2 % gezeigt werden. Insgesamt handelt es sich nämlich um eine so niedrige Anzahl von Referenzen (siehe Abbildung 5.58 auf der nächsten Seite), dass sich hier – im Gegensatz zu den anderen Projekten – Schwankungen ergeben.

## 5. Ergebnisanalyse

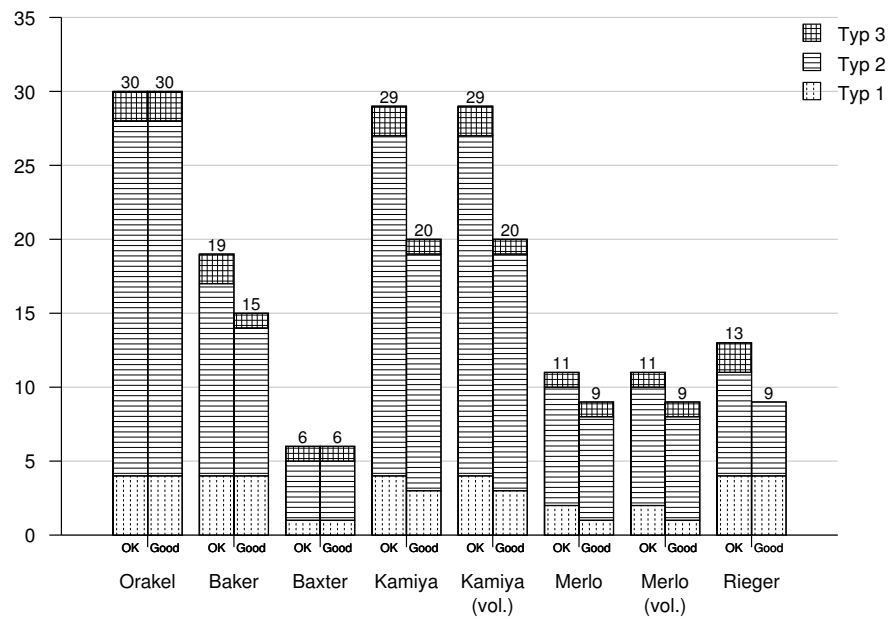
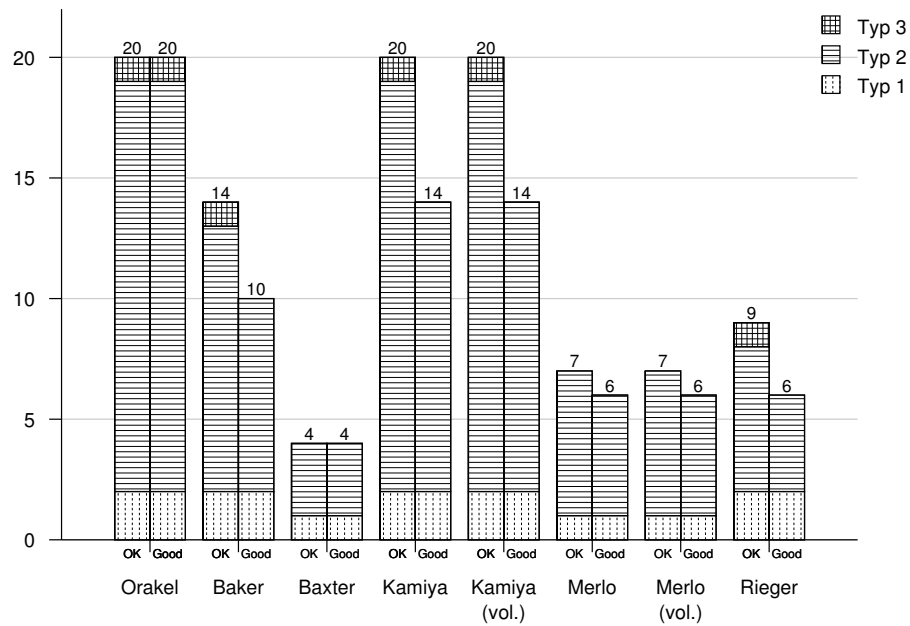


Abbildung 5.58: Getroffene Referenzen für das Projekt eclipse-ant  
(oben für 1 %, unten für 2 % bewerteter Kandidaten)

Auffällig ist, dass Kamiya mit OK-Match(0.7) nach 1 % alle Referenzen überdeckt und nach 2 % alle bis auf eine. Bei eingehender Betrachtung fällt ebenso auf, dass sich die Werte nach 2 % fast allesamt proportional aus den Werten nach 1 % mit einer Multiplikation von 1.5 ergeben. Auf den ersten Blick scheinen also bereits bei einer so niedrigen Anzahl von Referenzen die Daten repräsentativ zu sein.

Die Häufigkeiten der mehrfach gefundenen Referenzen, die man in Abbildung 5.59 vorfindet, zeigen sogar, dass selbst bei so wenig Referenzen die gesamte Sparte vorhanden ist: Von Referenzen, die nur ein Werkzeug findet, bis zu Referenzen, die alle Teilnehmer erkennen, ist alles vorhanden. Auch der „übliche“ Schwerpunkt bei einfach erkannten Referenzen ist zu erkennen.

<i>N</i>	Referenzen
1	14
2	4
3	5
4	3
5	2

Abbildung 5.59: *N*-fach gefundene Referenzen von eclipse-ant (2 %, Good)

Sieht man sich die Matrizen an, sind allerdings doch einige Dinge recht auffällig: Baker und Kamiya finden immerhin 11 Referenzen gemeinsam. Bei der geringen Anzahl ist dies ein extrem hoher Anteil. Baker und Rieger entdecken zusammen noch acht Referenzen. Gemessen an der Tatsache, dass Rieger insgesamt nur neun Referenzen trifft, heißt dies folglich, dass er nur eine Referenz erkennt, die Baker nicht auch findet.

**3. FoundSecrets.** Diesmal sind vom Schiedsrichter zwei Klonpaare vom Typ 2 und ein Klonpaar vom Typ 3 versteckt worden. Dieser Typ-3-Klon wird von keinem Teilnehmer mit einem Good-Match(0.7) gefunden. Bis auf Baxter und Merlo finden die anderen diesen Klon jedoch mit einem OK-Match(0.7) mit totaler Überdeckung. Es handelt sich bei dem Klon um eine Klasse von 83 Zeilen Länge, bei deren Kopie einige Methoden fehlen. Sie ist dadurch nur noch 64 Zeilen lang. Die Teilnehmer finden einzelne kopierte Methoden, aber nicht die ganze Klasse. So sind auch die Resultate der OK-Match(0.7) und Good-Match(0.7) zu erklären. In Abbildung 5.60 wird die Anzahl der von den einzelnen Teilnehmern gefundenen restlichen zwei Referenzen aufgeschlüsselt.

Baker	Baxter	Kamiya	Kamiya (vol.)	Merlo	Merlo (vol.)	Rieger
2	1	1	1	2	1	2

Abbildung 5.60: FoundSecrets im Projekt eclipse-ant (2 %, Good) von 3

**4. Rejected.** Anhand der zwei Graphiken in Abbildung 5.61 auf der nächsten Seite sieht man nun recht deutlich, wie sehr die Daten bei dieser niedrigen Anzahl von Referenzen noch schwanken. Von Baxter ist nur ein einziger Kandidat bewertet worden und dieser wurde nicht verworfen. Bei Merlo wurden in der 1 %-Auswertung jeweils ein Kandidat, in der

## 5. Ergebnisanalyse

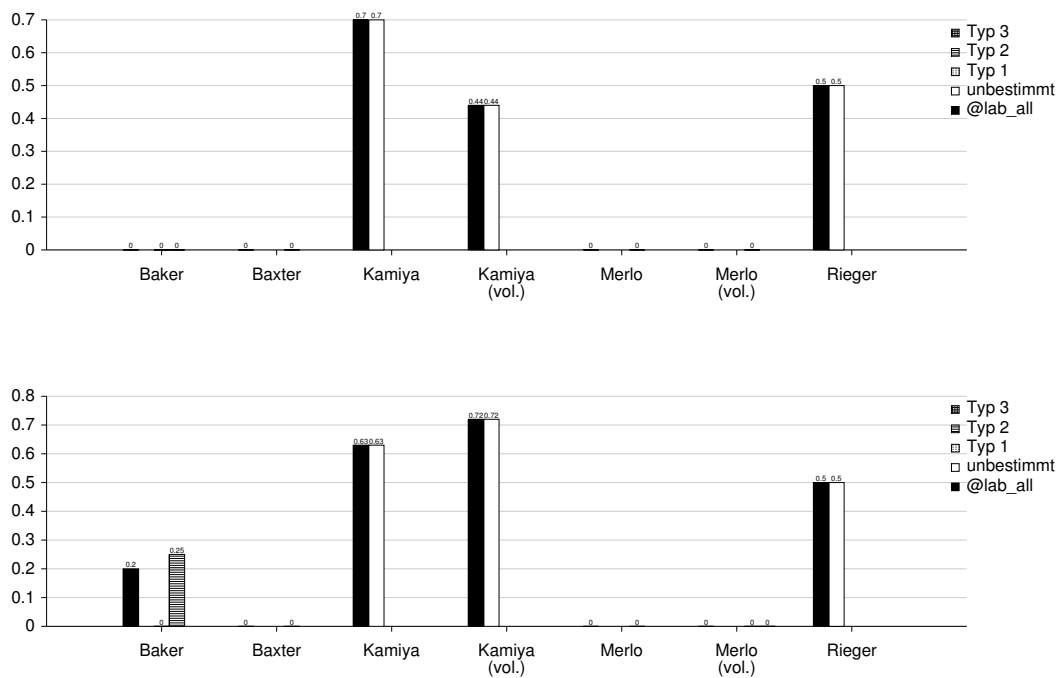


Abbildung 5.61: Rejected für das Projekt eclipse-ant  
(1 %, Good oben und 2 % Good unten)

2%-Auswertung jeweils zwei Kandidaten bewertet und akzeptiert. Bei Rieger waren es zwei bzw. vier bewertete Kandidaten, von denen jeweils einer bzw. zwei verworfen worden sind. Aus diesem Grund ist der Wert von Rieger stabil, dennoch darf dies nicht als repräsentativ angesehen werden. In der 1%-Auswertung sind von Baker drei Kandidaten akzeptiert worden, in der 2%-Auswertung kamen zwei weitere hinzu, von denen einer verworfen wurde. Bei Kamiya sind immerhin zehn bzw. neun Kandidaten in der ersten Auswertung bewertet und 19 bzw. 18 in der zweiten. Dennoch sieht man, dass diese Anzahl zu klein ist, um stabile Ergebnisse zu liefern.

**5. TrueNegatives.** Wie man in Abbildung 5.62 auf der nächsten Seite sehen kann, sind die Werte der TrueNegatives um einiges stabiler und repräsentativer. Dies liegt daran, dass es hier nicht auf die Anzahl bewerteter Kandidaten ankommt, sondern auf die Größe und den Umfang der Referenzmenge.

Bei Baker, Kamiya und Rieger sind die Werte stabil. Bei Baxter erscheinen die Daten zunächst fehlerhaft, da sich der Gesamtanteil nicht verändert, wenn sowohl der Anteil für Typ 1 als auch der Anteil für Typ 2 steigt und der für Typ 3 gleich bleibt. Dies hat aber seine Richtigkeit und lässt sich dadurch erklären, dass zwei Kandidaten, die mit unterschiedli-

## 5.2. Auswertung nach Projekten

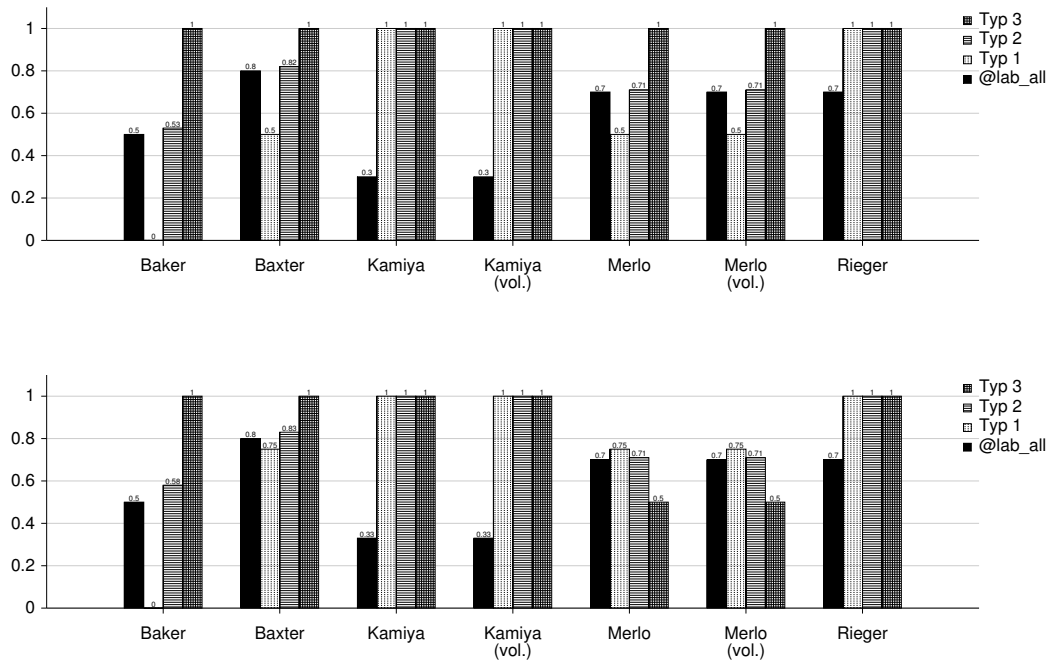


Abbildung 5.62: TrueNegatives für das Projekt eclipse-ant  
(1 %, Good oben und 2 %, Good unten)

chem Typ eingeschendet sind, die gleiche Referenz mit Good-Match(0.7) überdecken. Beim Gesamtanteil werden diese nur einmal gewertet, da es sich um die gleiche Referenz handelt. Bei der Aufschlüsselung in die verschiedenen Typen werden sie jedoch bei jedem Typ gewertet. Dies gilt prinzipiell für alle Graphiken dieses Typs und ist beabsichtigt. In diesem Fall ist es nur äußerst auffällig und könnte fälschlicherweise als Fehler der Auswertung angesehen werden.

Zuletzt fällt allerdings bei Merlo auf, dass beim Bewerten von 1 % zu 2 % ein Typ-1-Klon hinzugekommen ist, den Merlo nicht findet, dafür aber ein Typ-3-Klon, den er findet. Dies reicht bei der geringen Anzahl von Referenzen bereits aus, um diesen Unterschied zu machen.

**6. Recall und Precision.** Auch die Werte von Recall und Precision unterliegen dieser starken Schwankung. Da sich jedoch aus den Werten für die 1 %-Auswertung außer der Tatsache, dass sie nicht repräsentativ sind, keine weiteren Erkenntnisse gewinnen lassen, werden in Abbildung 5.63 auf der nächsten Seite nun wieder ausschließlich die Diagramme für die 2 %-Auswertung gezeigt.

## 5. Ergebnisanalyse

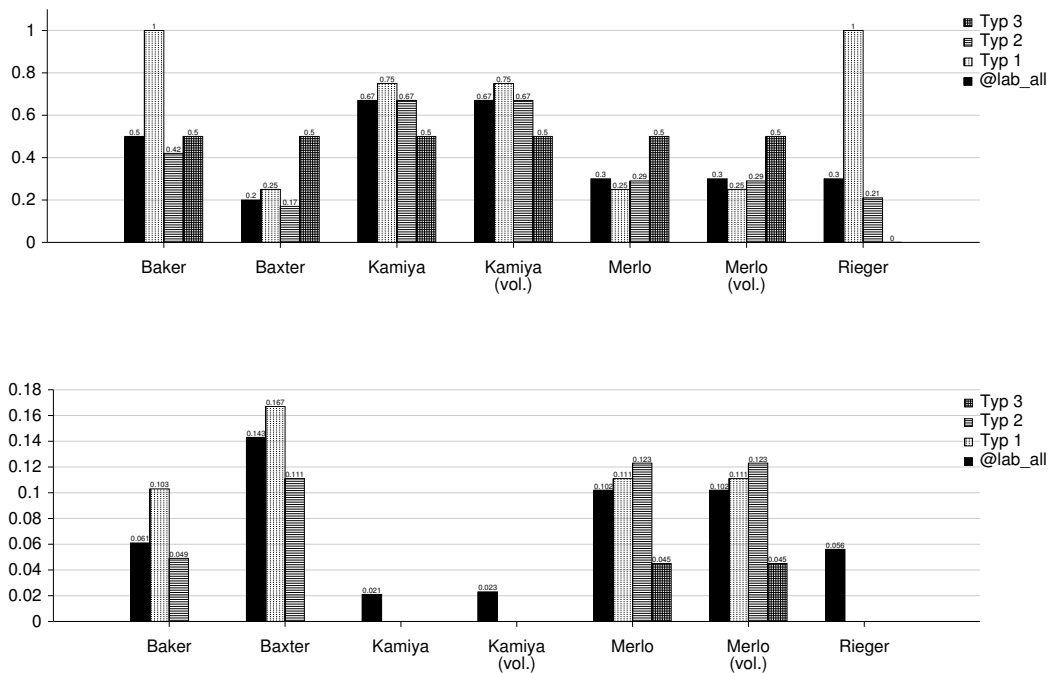


Abbildung 5.63: Recall (oben) und Precision (unten) für das Projekt eclipse-ant (2 %, Good)

Baker und Rieger schaffen es, alle vier Klone vom Typ 1 zu finden. Kamiya findet drei davon, hat aber mit insgesamt zwei Drittel gefundener Referenzen wieder den höchsten Recall. Verglichen mit den anderen Projekten hat Merlo in eclipse-ant seinen höchsten Recall.

Aufgrund ihrer geringen Anzahl von Kandidaten führen Baxter und Merlo wieder die Rangliste der höchsten Precision-Werte an.

**7. Klongrößen.** Die Betrachtung der Klongrößen in Abbildung 5.64 auf der nächsten Seite macht deutlich, dass es sich bei den gefundenen Klonpaaren in eclipse-ant hauptsächlich um kleine Klone handelt. Die Werte für die Durchschnittsgröße und die Standardabweichung hängen bei der niedrigen Anzahl an Referenzen zu stark von der maximalen Größe ab, um ihnen eine große Bedeutung beizumessen.

**8. Klonverteilung.** Auch die Verteilung der gefundenen Referenzen in Abbildung 5.65 auf der nächsten Seite ist alles andere als repräsentativ. Es liegt aber näher, eine 50:50 Verteilung zu vermuten, als anzunehmen, dass eine der beiden Arten deutlich häufiger vertreten wäre.

**9. Verschiedenes.** Eine überraschende Auffälligkeit in Abbildung 5.66 auf Seite 96 stellt der Umstand dar, dass nun auch Merlo sich überlappende Klonpaare meldet. In allen 5 Fällen handelt es sich jedoch um Teile einer if-else-if-Kaskade, wobei jeweils die Zeilen,

## 5.2. Auswertung nach Projekten

Teilnehmer	MaxCloneSize	AvgCloneSize	StdDevCloneSize
Baker	25	11.19	5.18
Baxter	15	9.50	3.21
Kamiya	50	14.75	11.84
Kamiya (vol.)	50	14.75	11.84
Merlo	15	8.89	2.76
Merlo (vol.)	15	8.89	2.76
Rieger	25	11.78	6.42
Orakel	83	15.30	14.41

Abbildung 5.64: Größen der Codefragmente von eclipse-ant (2 %, Good)

Teilnehmer	InnerFilePairs	in %	AcrossFilePairs	in %
Baker	7	46.67	8	53.33
Baxter	3	50.00	3	50.00
Kamiya	5	25.00	15	75.00
Kamiya (vol.)	5	25.00	15	75.00
Merlo	6	66.67	3	33.33
Merlo (vol.)	6	66.67	3	33.33
Rieger	4	44.44	5	55.56
Orakel	13	43.33	17	56.67

Abbildung 5.65: Gefundene Referenzen innerhalb von Dateien und über Dateien hinweg von eclipse-ant (2 %, Good)

## 5. Ergebnisanalyse

die `else if` enthalten, zu beiden Codefragmenten gemeldet sind. D. h. die Endzeile des einen Codefragments ist die Startzeile des zweiten.

Teilnehmer	OnlyPairs	OnlyButOnePairs	OverlappingCandidates
Baker	2	0	21
Baxter	0	0	0
Kamiya	0	0	122
Kamiya (vol.)	0	0	114
Merlo	0	0	5
Merlo (vol.)	0	0	5
Rieger	0	2	12

Abbildung 5.66: Weitere Werte von `eclipse-ant` (2 %, Good)

Aufgrund der Tatsache, dass in `eclipse-ant` in der Tat extrem wenige Klone zu sein scheinen und die Menge der eingesendeten Kandidaten in diesem Projekt am geringsten ist, wäre es eventuell in Erwägung zu ziehen, für dieses Projekt eine Auswertung mit höherem Anteil an bewerteten Kandidaten vorzunehmen, d. h. die Bewertung zum Beispiel fortzusetzen bis 50 % aller Kandidaten vom Schiedsrichter bewertet sind.

### 5.2.7. `eclipse-jdtcore`

Bei `eclipse-jdtcore` handelt es sich um das zweitgrößte der Java-Projekte. Es besteht aus 148K SLOC auf 741 Dateien verteilt. Auch hier muss wieder auf einen Vergleich mit `Krinke` verzichtet werden.

**1. Kandidaten.** Betrachtet man die Anzahl der Kandidaten in Abbildung 5.67 auf der nächsten Seite, so fällt auf, dass im Vergleich zu den zwei bisherigen Java-Projekten immens viele Klone vorhanden zu sein scheinen. Darin sind sich alle Teilnehmer einig. Ein weiterer interessanter Punkt ist, dass Rieger, der bisher immer die gleiche Größenordnung an Kandidaten wie Baker eingesendet hat, im Vergleich zu den anderen Teilnehmern extrem wenig zu finden scheint. Des Weiteren sticht ins Auge, dass Merlo diesmal auch sehr viele Kandidaten findet. Kamiya, der wie immer die meisten Kandidaten in den Projekten meldet, eliminiert in seiner „Kür“-Variante immerhin ein Viertel seiner Kandidaten des Pflicht-Teils.

**2. Referenzen.** Abbildung 5.68 auf der nächsten Seite stellt sich bei näherer Betrachtung als höchst interessant heraus. Es fällt wieder auf, dass Kamiyas „Kür“-Optimierung extrem gut funktioniert. Nur ein `Good-Match(0.7)` zu viel ist eliminiert worden, ansonsten hat er nur uninteressante Kandidaten aus der Menge entfernt. Baker, die dann im Vergleich zu Kamiyas Zweiteinsendung mehr Kandidaten meldet, trifft auch etwas mehr Referenzen als er. Allerdings sind Merlos Daten hier unübertroffen: Zum einen schafft er es, mit weniger als der Hälfte der Kandidaten wie Baker mehr Referenzen zu überdecken, zum anderen ist die Qualität seiner Kandidaten sehr gut. Dies sieht man an dem geringen Anteil, den er von der OK-Auswertung zur Good-Auswertung verliert. Dies sind nur um 29 %. Qualitativ besser ist

## 5.2. Auswertung nach Projekten

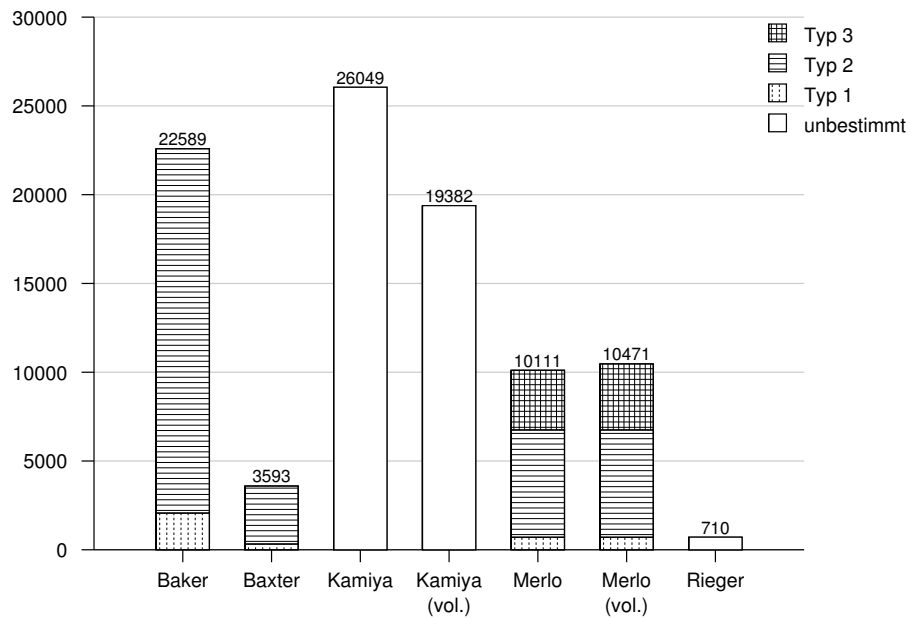


Abbildung 5.67: Anzahl der Kandidaten für das Projekt eclipse-jdtcore

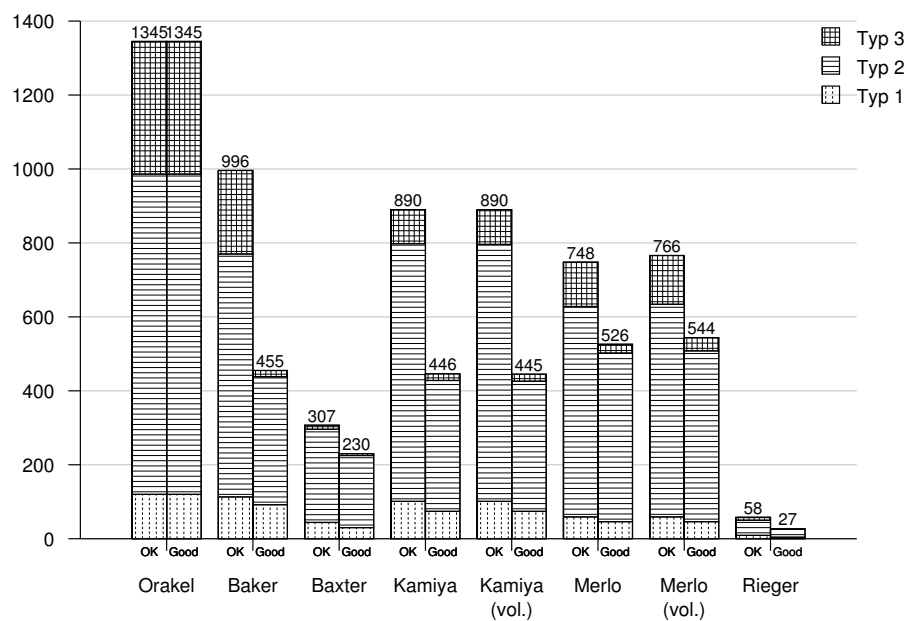


Abbildung 5.68: Betroffene Referenzen für das Projekt eclipse-jdtcore (2 %)

## 5. Ergebnisanalyse

hier nur Baxter, der nur 25 % weniger Good-Match(0.7)-Treffer als OK-Match(0.7)-Treffer hat. Rieger überdeckt mit seinen Kandidaten die geringste Anzahl von Referenzen.

$N$	Referenzen
1	388
2	359
3	162
4	23

Abbildung 5.69:  $N$ -fach gefundene Referenzen von eclipse-jdtcore (2 %, Good)

An der in Abbildung 5.69 dargestellten Anzahl an Referenzen, die von den Kandidaten mehrerer Werkzeuge überdeckt werden, fällt auf, dass selbst 162 Referenzen noch von drei Werkzeugen erkannt werden. Die einfach und doppelt gefundenen Referenzen kommen auch in eclipse-jdtcore am häufigsten vor. Obwohl noch 23 Kandidaten von vier Teilnehmern gemeinsam entdeckt werden, wird kein einziger von allen gefunden. Dies liegt wohl auch daran, dass Rieger in diesem Projekt besonders schlecht abschneidet.

In den Matrizen findet sich die Information, dass Baker und Kamiya insgesamt 303 Referenzen gemeinsam entdecken, Baker und Merlo immerhin noch 186. Des Weiteren wurden sieben Kandidaten bei Baker verworfen, welche auch bei Kamiya bei der Bewertung nicht akzeptiert wurden.

**3. FoundSecrets.** Von den drei versteckten Klonen ist einer ein Typ-3-Klon und dieser wird von keinem Teilnehmer mit einem Good-Match(0.7) gefunden. Bis auf Baxter entdecken ihn die anderen aber bei Betrachtung mit OK-Match(0.7). Es handelt sich hierbei wieder um eine komplette Klasse, der ein Feld und alle Bezüge darauf in der Kopie entfernt wurden. Die anderen zwei Klonpaare (beide vom Typ 1) werden, wie in Abbildung 5.70 dargestellt, gefunden.

Baker	Baxter	Kamiya	Kamiya (vol.)	Merlo	Merlo (vol.)	Rieger
2	0	1	1	1	1	0

Abbildung 5.70: FoundSecrets im Projekt eclipse-jdtcore (2 %, Good) von 3

**4. Rejected.** Als erstes fällt in Abbildung 5.71 auf der nächsten Seite wieder auf, wie Kamiya seinen extrem hohen Anteil an verworfenen Kandidaten im Pflicht-Teil in der „Kür“ etwas senken kann. Trotz der Unterschiede in der Anzahl der eingesendeten Kandidaten bei Baker und Rieger, ist beiden wieder gemeinsam, dass nur etwa ein Drittel der Kandidaten nicht verworfen wurde. Auffällig bei Baker ist, dass Typ-1-Klone deutlich häufiger akzeptiert wurden als Typ-2-Klone. Dies ist auch bei Merlos Typ-3-Kandidaten zu erkennen: Sie mussten viel öfter verworfen werden als seine Kandidaten vom Typ 1 und Typ 2. Baxter hat hier eindeutig die beste Ausbeute, denn bei ihm wurde nur jeder 10. Kandidat nicht akzeptiert.

**5. TrueNegatives.** Die zu erwartende Tatsache, dass Rieger den höchsten Anteil an nicht gefundenen Referenzen hat, wird in Abbildung 5.72 auf der nächsten Seite deutlich be-

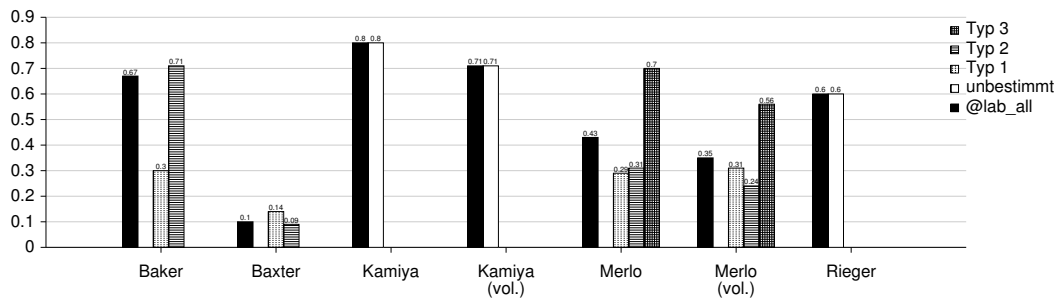


Abbildung 5.71: Rejected für das Projekt eclipse-jdtcore (2 %, Good)

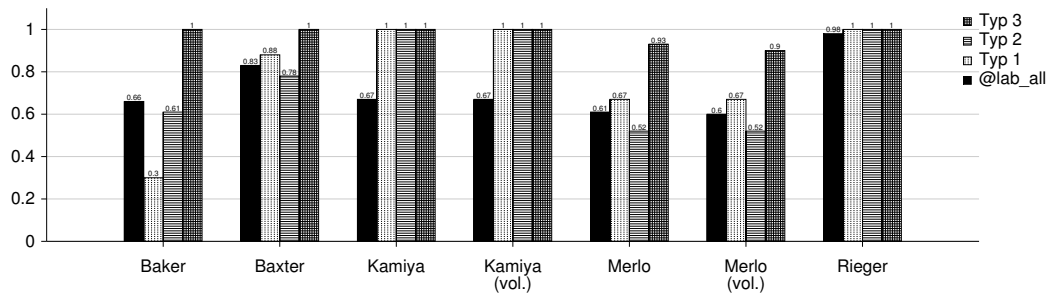


Abbildung 5.72: TrueNegatives für das Projekt eclipse-jdtcore (2 %, Good)

stätigt. Bis auf Baxter, der 83 % aller Referenzen nicht findet, bewegen sich die anderen Teilnehmer zwischen zwei Dritteln und 60 %. Schränkt man die Betrachtung auf Referenzen vom Typ 1 ein, so verpasst Baker mit ihren gemeldeten Typ-1-Kandidaten gar nur 30 %. Fast jeder zweite Kandidat vom Typ 2, den Merlo liefert, trifft auf eine Referenz vom Typ 2.

**6. Recall und Precision.** Den Werten für den Recall in Abbildung 5.73 auf der nächsten Seite kann man nochmals entnehmen, dass alle Teilnehmer bis auf Baxter und Rieger 33 bis 40 % erreichen. Unterschiede ergeben sich aber wieder einmal in der Erkennung der einzelnen Typen. Baker und Kamiya erkennen exakte Kopien besonders gut, während Merlo das bessere Werkzeug hat, wenn es um das Auffinden von parametrisierten Kopien in eclipse-jdtcore geht.

Die große Anzahl gemeldeter Kandidaten beschert Baker und Kamiya wieder eine sehr niedrige Precision. Sie landen im Schnitt nur bei jedem 50. Kandidaten eine Überdeckung mit einer Referenz. Die Aussage, dass Baker eher Kandidaten vom Typ 1 und Merlo eher Kandidaten vom Typ 2 findet, wird dadurch bestärkt, dass sowohl Recall als auch Precision bei den jeweiligen Typen der Teilnehmer höher ist als bei den anderen Typen. Das heißt, Baker findet mehr Typ-1-Klone im Projekt als Klone vom Typ 2. Auch ist ihre Trefferquote bei ihren Kandidaten vom Typ 1 höher als die bei Typ 2. Für Merlo gilt Entsprechendes für die genau umgekehrte Typ-Konstellation.

## 5. Ergebnisanalyse

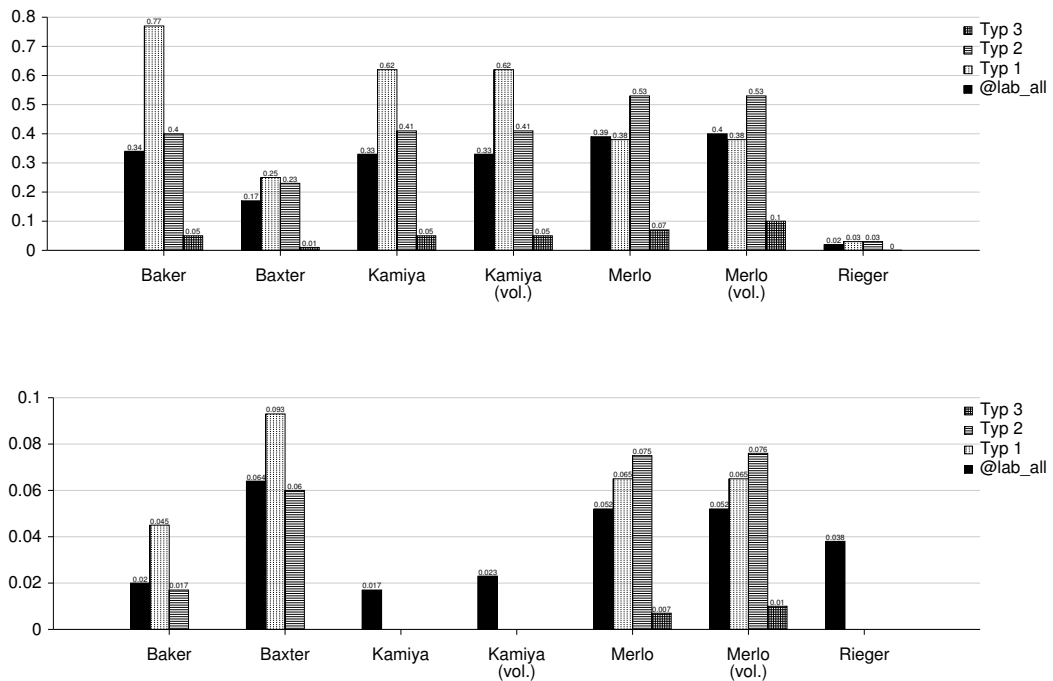


Abbildung 5.73: Recall (oben) und Precision (unten) für das Projekt eclipse-jdtcore (2 %, Good)

**7. Klongrößen.** Bis auf Rieger finden alle anderen Teilnehmer ihren größten Klon jenseits der 200-Zeilen-Marke, wie man aus Abbildung 5.74 auf der nächsten Seite entnehmen kann. Auch die Durchschnittsgrößen sind relativ nahe beisammen. Nur Kamiya findet im Schnitt größere Referenzen. Sowohl seine größte gefundene Referenz als auch seine Standardabweichung ist höher als die der Konkurrenten.

**8. Klonverteilung.** In eclipse-jdtcore herrschen wieder Klonpaare vor, deren beide Codefragmente sich innerhalb derselben Dateien befinden. Dies kann man deutlich in Abbildung 5.75 auf der nächsten Seite erkennen. Je nach Teilnehmer handelt es sich um gut zwei Drittel bis fast vier Fünftel aller Referenzen. Eine Erklärung für diese Auffälligkeit könnte unter anderem der Quelltext in `eclipse-jdtcore/src/internal/compiler/codegen/CodeStream.java` sein. Diese Datei besteht fast ausschließlich aus voneinander geklonten Funktionen.

**9. Verschiedenes.** Baker findet, wie in Abbildung 5.76 auf der nächsten Seite zu sehen ist, eine erstaunlich hohe Anzahl von Referenzen, die außer ihr niemand anderes findet. Auch Kamiya und Merlo finden mit ihren „Kür“-Teilen Klone als einzige. Die hohe An-

## 5.2. Auswertung nach Projekten

Teilnehmer	MaxCloneSize	AvgCloneSize	StdDevCloneSize
Baker	213	15.84	18.51
Baxter	219	14.97	16.55
Kamiya	345	21.21	28.42
Kamiya (vol.)	327	21.42	28.28
Merlo	206	13.76	11.87
Merlo (vol.)	206	13.65	11.71
Rieger	32	15.19	7.48
Orakel	267	15.53	17.72

Abbildung 5.74: Größen der Codefragmente von eclipse-jdtcore (2 %, Good)

Teilnehmer	InnerFilePairs	in %	AcrossFilePairs	in %
Baker	333	73.19	122	26.81
Baxter	157	68.26	73	31.74
Kamiya	330	73.99	116	26.01
Kamiya (vol.)	329	73.93	116	26.07
Merlo	408	77.57	118	22.43
Merlo (vol.)	425	78.13	119	21.88
Rieger	1	3.70	26	96.30
Orakel	920	68.40	425	31.60

Abbildung 5.75: Gefundene Referenzen innerhalb von Dateien und über Dateien hinweg von eclipse-jdtcore (2 %, Good)

Teilnehmer	OnlyPairs	OnlyButOnePairs	OverlappingCandidates
Baker	71	0	300
Baxter	26	0	0
Kamiya	0	0	1519
Kamiya (vol.)	1	0	1108
Merlo	0	0	24
Merlo (vol.)	17	0	24
Rieger	10	21	11

Abbildung 5.76: Weitere Werte von eclipse-jdtcore (2 %, Good)

## 5. Ergebnisanalyse

zahl an Referenzen, die alle anderen Teilnehmer bis auf Rieger finden, lässt sich mit hoher Wahrscheinlichkeit wieder damit erklären, dass er im Vergleich zu den anderen sehr wenig Kandidaten meldet. Doch immerhin sind 10 seiner 27 gefundenen Kandidaten Klonpaare, die sonst niemand erkennt.

Wieder ist Baxter der einzige, der ausschließlich sich nicht überlappende Kandidaten liefert. Die 24 Kandidaten von Merlo, welche sich überlappen, sind wieder `if-else-if`-Konstrukte, bei denen die letzte Zeile des ersten und die erste Zeile des zweiten Codefragments identisch sind.

Abschließend ist zu bemerken, dass alles in allem der Verdacht nahe liegt, dass Rieger hier keine „normale“ Analyse durchführen konnte, sondern dass ihm eine Ressource (entweder Speicherplatz oder Rechenzeit) ausgegangen ist. Es gibt viele Indizien dafür: Er findet im Verhältnis zu Baker viel weniger Klone als sonst, die maximale Klonegröße ist bei ihm viel kleiner als üblich, er findet fast keine Klonpaare innerhalb einer Datei und außerdem hatte er Probleme, die zwei anderen großen Projekte `postgres` und `j2sdk1.4.0-javax-swing` zu analysieren. Und da `eclipse-jdtcore` sogar mehr Klone enthält als `j2sdk1.4.0-javax-swing`, ist dieser Verdacht wohl nicht ganz unbegründet.

### 5.2.8. j2sdk1.4.0-javax-swing

Zum Schluss soll nun das größte der Java-Projekte beleuchtet werden. `j2sdk1.4.0-javax-swing` umfasst 538 Dateien und summiert sich auf 204K SLOC. Von Krinke fehlen wieder die Vergleichswerte, da er kein Java analysieren kann, und auch von Rieger sind hier keine Ergebnisse vorhanden, da er dieses Projekt aufgrund seiner Größe nicht analysieren kann.

**1. Kandidaten.** In Abbildung 5.77 auf der nächsten Seite sind die Kandidaten der sechs in diesem Projekt teilnehmenden Werkzeuge aufgetragen. Abgesehen vom fast schon üblichen Verhältnis der eingesandten Kandidatenmenge der Teilnehmer zueinander, fällt auf, dass Baxter fast nur Kandidaten vom Typ 2 findet. Seine 53 Kandidaten vom Typ 1 gehen bei 3713 Kandidaten vom Typ 2 fast unter. Dem entgegengesetzt meldet Merlo nämlich nahezu gleiche Anteile der drei Klontypen.

**2. Referenzen.** Sofort fällt ins Auge, dass der in Abbildung 5.78 auf der nächsten Seite bei Baxter als fast nicht existent erwartete Balken für Typ-1-Klone doch recht groß ist: Er findet 94 Referenzen vom Typ 1, obwohl er nur 53 Kandidaten mit dem Typ 1 liefert. Dies zeigt, dass er hier mit seiner Kategorisierung der Klonpaare in die Klontypen falsch liegt und viele Kandidaten fälschlicherweise als Typ 2 benennt, obwohl es sich in der Tat um Typ-1-Klone handelt. Außerdem liefern nur drei seiner als Typ 1 gemeldeten Kandidaten eine Überdeckung bei `Good-Match(0.7)`. Des Weiteren fällt auf, dass Merlo sehr wenige Referenzen vom Typ 3 findet, obwohl er in etwa gleich viele jeden Typs liefert. Das Bemerkenswerte an `j2sdk1.4.0-javax-swing` scheint zu sein, dass es im Vergleich zu den anderen Projekten sehr wenig Klone vom Typ 3 beinhaltet und obwohl es größer als `eclipse-jdtcore` ist, insgesamt doch nur die Hälfte an Referenzen hat.

Anhand Abbildung 5.79 auf Seite 104 sieht man, dass der große Anteil der Referenzen in `j2sdk1.4.0-javax-swing` von zwei Werkzeugen getroffen wird. Selbst der Anteil, den drei Teilnehmer finden, ist noch größer als der, der nur einfach erkannt wird.

## 5.2. Auswertung nach Projekten

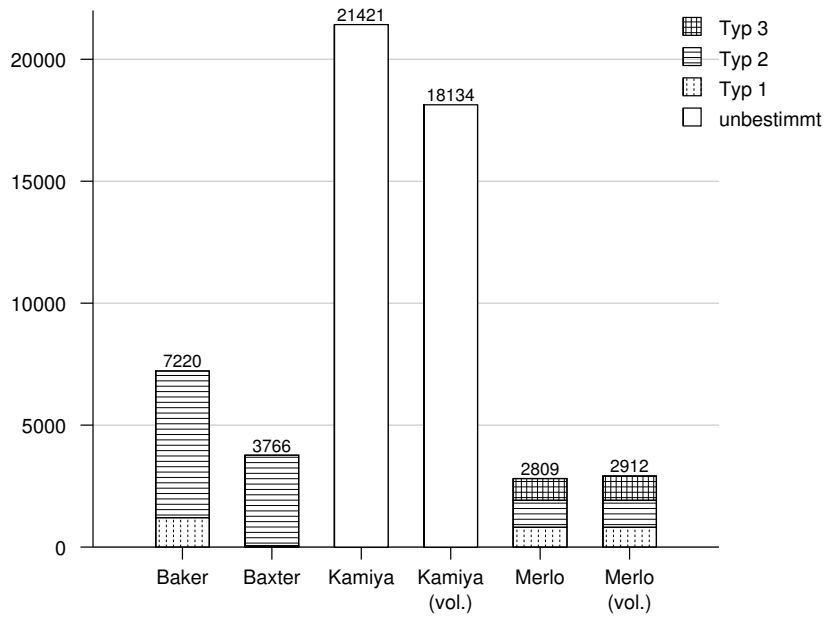


Abbildung 5.77: Anzahl der Kandidaten für das Projekt j2sdk1.4.0-javax-swing

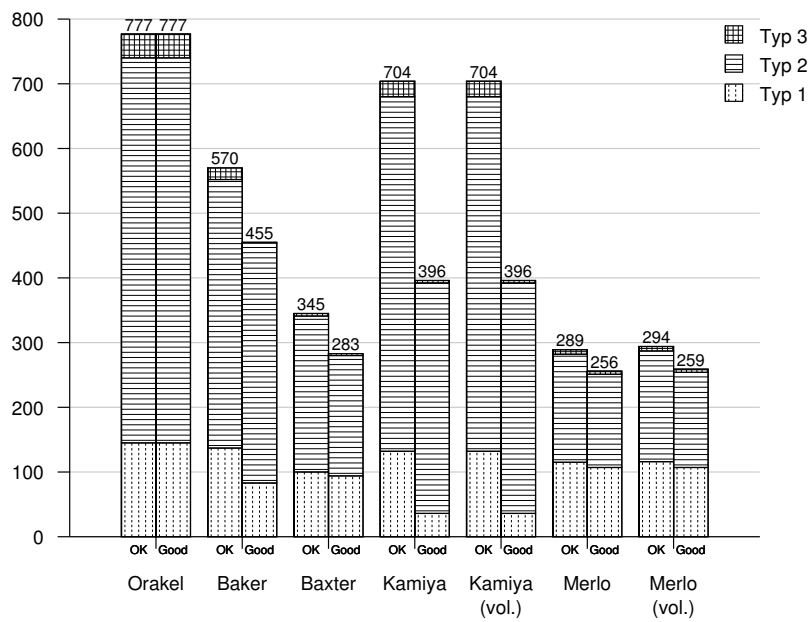


Abbildung 5.78: Getroffene Referenzen für das Projekt j2sdk1.4.0-javax-swing (2 %)

## 5. Ergebnisanalyse

$N$	Referenzen
1	177
2	306
3	187
4	10

Abbildung 5.79:  $N$ -fach gefundene Referenzen von j2sdk1.4.0-javax-swing (2 %, Good)

Eine Betrachtung der Matrizen zeigt, dass Baker und Kamiya 309 Referenzen gemeinsam entdecken, Baker und Baxter noch 169 und Baker, Baxter und Merlo zusammen noch 147 Referenzen. Bei den Kandidaten, die bei zwei Teilnehmern verworfen wurden, haben Baker und Kamiya 14 bei beiden nicht akzeptierte Kandidaten.

**3. FoundSecrets.** Bei den drei versteckten Referenzen handelt es sich um zwei vom Typ 3 und einen Klon vom Typ 2. Einer der zwei Typ-3-Klone wird nur mit OK-Match(0.7), nicht aber mit Good-Match(0.7) erkannt. Die anderen zwei Klonpaare werden von den Werkzeugen der Teilnehmer gemäß Abbildung 5.80 entdeckt.

Baker	Baxter	Kamiya	Kamiya (vol.)	Merlo	Merlo (vol.)
1	2	0	0	1	1

Abbildung 5.80: FoundSecrets im Projekt j2sdk1.4.0-javax-swing (2 %, Good) von 3

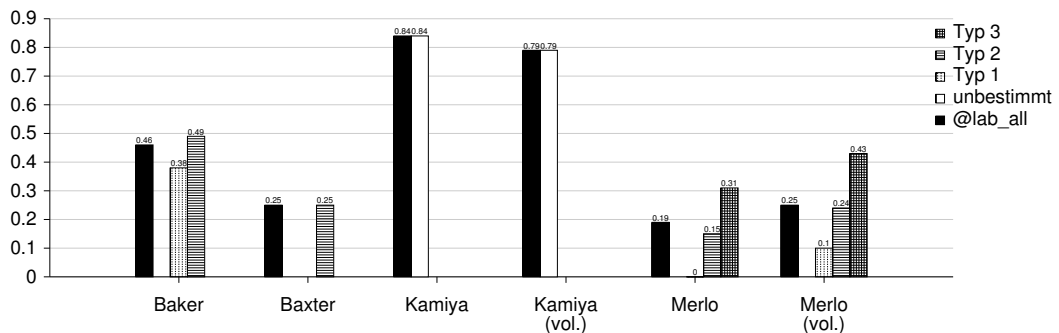


Abbildung 5.81: Rejected für das Projekt j2sdk1.4.0-javax-swing (2 %, Good)

**4. Rejected.** In Abbildung 5.81 sind wieder die Anzahl der Kandidaten, die bewertet worden sind, aber sich nicht bei Good-Match(0.7) mit einer Referenz überdecken, im Verhältnis zu der Anzahl betrachteter Kandidaten pro Teilnehmer dargestellt. Von Kamiyas Referenzen wurden bis auf etwa ein Fünftel alle Kandidaten verworfen. Bei Baker waren es etwas weniger als die Hälfte, die nicht akzeptiert werden konnten. Bei Baxter und Merlo ist nur jeder vierte Kandidat als Klonpaar nicht akzeptabel. Betrachtet man wieder nur die Typ-1-Klone, so ist keiner von Merlos 11 betrachteten Kandidaten im Pflicht-Teil zu verwerfen gewesen.

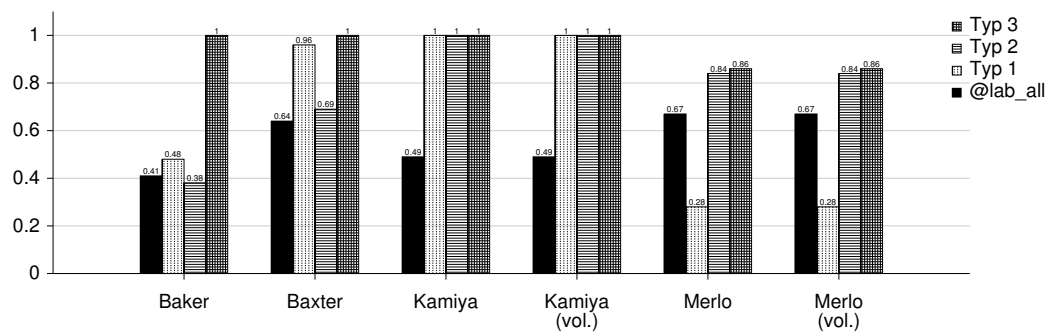


Abbildung 5.82: TrueNegatives für das Projekt j2sdk1.4.0-javax-swing (2 %, Good)

**5. TrueNegatives.** Abbildung 5.82 zeigt, dass die Teilnehmer immerhin ein Drittel bis drei Fünftel aller vorhandenen Referenzen finden. Je nach Klontyp ergeben sich wieder Unterschiede. Vor allem Baxters Wert für die Typ-1-Klone spiegelt wider, dass er nur 53 Kandidaten dieses Typs meldet und nur drei davon wirklich auf Referenzen vom Typ 1 treffen (das Diagramm setzt ja die Kandidaten, die sich mit Referenzen überdecken, in Relation zu der Gesamtzahl der Referenzen des jeweils gleichen Typs). Sein tatsächlicher Recall an Typ-1-Klonen, die er ja meistens falsch kategorisiert hat, ist daher erst in Abbildung 5.83 auf der nächsten Seite zu sehen. Weiterhin erkennt man, dass Merlos Kandidaten des Typs 1 fast drei Viertel aller Referenzen dieses Typs finden.

**6. Recall und Precision.** In Abbildung 5.83 auf der nächsten Seite sieht man nun, dass Baxter dennoch einen hohen Recall an Klonen vom Typ 1 hat. Dies ist darauf zurückzuführen, dass er eben doch viele dieser Referenzen findet, sie „nur“ falsch kategorisiert hat. Dennoch findet Merlo noch mehr: 74 % aller Referenzen vom Typ 1 entdeckt sein Werkzeug. Äußerst interessant ist, wie niedrig allgemein der Recall des 3. Klontyps ausfällt. Nicht nur, dass j2sdk1.4.0-javax-swing wenig Klone dieses Typs zu haben scheint, es sieht so aus, als seien sie auch noch besonders schwierig zu entdecken. Baker und Kamiya finden die meisten parametrisierten Kopien.

Der völlig „irre“ Wert in der Abbildung für Baxters Precision bei den exakten Kopien kommt aus bereits erwähnter Anomalie, dass er nur 53 seiner Kandidaten als Typ 1 kategorisiert, aber im Endeffekt 94 Referenzen dieses Typs findet. Daraus ergibt sich  $\frac{94}{53} \approx 1.774$ .

Sieht man von diesem „Ausreißer“ ab, so ergibt sich das bereits gewohnte Bild. Merlo erreicht die größte Precision, Kamiya die niedrigste. Bei Merlo sind die Werte für die 1. und 2. Klontypen wieder überdurchschnittlich, lediglich beim Typ 3 ist die Ausbeute diesmal extrem gering. Dies bestärkt noch einmal den Verdacht, dass das Projekt j2sdk1.4.0-javax-swing wenig Klone dieses Typs hat und diese dann auch noch schwieriger als sonst zu entdecken sind. Aus diesem Grund ist es schade, keinen Vergleichswert mit einem anderen Werkzeug zu haben, das auch Klone vom 3. Typ erkennt (wie z. B. Duploc oder Duplix).

**7. Klongrößen.** Einen äußerst interessanten Anblick bietet auch Abbildung 5.84 auf Seite 107. Die Teilnehmer mit den größeren Recall-Werten haben im Schnitt auch größere Referenzen gefunden. Dies liegt wohl vor allem auch an den Dateien im Verzeichnis

## 5. Ergebnisanalyse

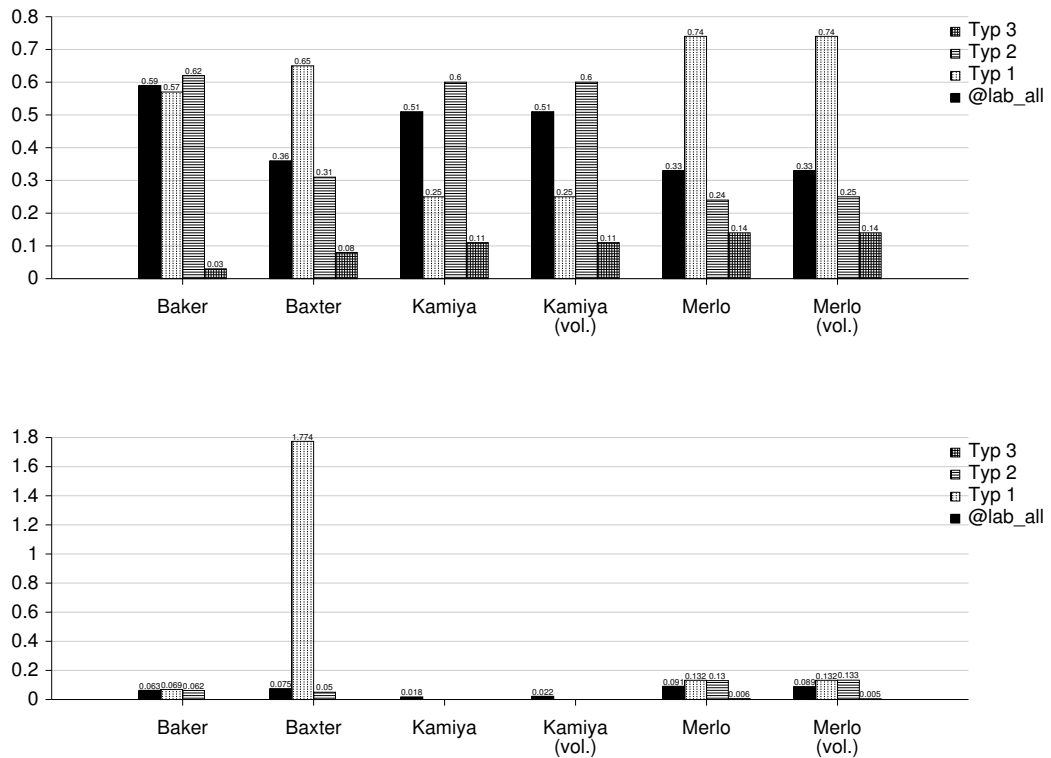


Abbildung 5.83: Recall (oben) und Precision (unten) für das Projekt j2sdk1.4.0-javax-swing (2 %, Good)

j2sdk1.4.0-javax-swing/src/plaf/multi/. Alle 32 Dateien in diesem Verzeichnis sind Klone voneinander. Entweder die kompletten Dateien sind paarweise Typ-2-Klone, oder zumindest große Teile daraus lassen sich als Klone vom Typ 2 betrachten. Kleinere Methoden daraus sind auch als Typ 1 zu entdecken. So lassen sich Merlos geringe Werte sowohl bei der maximalen Klongröße als auch bei durchschnittlicher Größe und Standardabweichung erklären.

**8. Klonverteilung.** Abbildung 5.85 auf der nächsten Seite bestärkt die eben erwähnte Auffälligkeit: In j2sdk1.4.0-javax-swing wird wesentlich mehr über Dateien hinweg geklont. Die Teilnehmer sind sich hier auch erstaunlich mit einem Anteil von 80 % – 90 % einig. Lediglich Merlo findet etwas mehr Klone innerhalb von Dateien. Das liegt daran, dass er, wie schon gesehen, kleinere Klonpaare findet, und dies sind dann auch des Öfteren Methoden innerhalb einer Klasse. Die anderen Teilnehmer finden oft ganze Klassen oder zumindest mehrere Methoden direkt hintereinander über Dateien hinweg geklont.

**9. Verschiedenes.** Auch diesmal sind es hauptsächlich Baker und Baxter, die einige Referenzen finden, die sonst niemand findet (siehe Abbildung 5.86 auf der nächsten Seite).

## 5.2. Auswertung nach Projekten

Teilnehmer	MaxCloneSize	AvgCloneSize	StdDevCloneSize
Baker	326	59.45	54.13
Baxter	183	48.77	60.18
Kamiya	326	71.39	50.57
Kamiya (vol.)	326	67.96	46.45
Merlo	51	9.25	4.95
Merlo (vol.)	51	9.24	4.92
Orakel	309	43.10	50.17

Abbildung 5.84: Größen der Codefragmente von j2sdk1.4.0-javax-swing (2 %, Good)

Teilnehmer	InnerFilePairs	in %	AcrossFilePairs	in %
Baker	68	14.95	387	85.05
Baxter	54	19.08	229	80.92
Kamiya	47	11.87	349	88.13
Kamiya (vol.)	47	11.87	349	88.13
Merlo	60	23.44	196	76.56
Merlo (vol.)	60	23.17	199	76.83
Orakel	137	17.63	640	82.37

Abbildung 5.85: Gefundene Referenzen innerhalb von Dateien und über Dateien hinweg von j2sdk1.4.0-javax-swing (2 %, Good)

Aber Baxter gelingt es in j2sdk1.4.0-javax-swing nicht, eine ganze Reihe von Klonen zu entdecken, die alle anderen Teilnehmer finden. Auch ist er neben Krinke der einzige, der im gesamten Experiment nicht einen sich überlappenden Kandidaten gemeldet hat. Dies hängt damit zusammen, dass sein CloneDR™ die Klone, die es findet, automatisch durch Makros oder Funktionen ersetzen kann. Dies funktioniert natürlich nur, wenn keine Überlappungen vorhanden sind. Merlo meldet 22 sich überlappende Kandidaten. Es handelt sich hierbei wieder – wie bei den letzten beiden Projekten – um `if-else-if`-Konstrukte, wobei jeweils der Teil mit dem `else if` in beiden Codefragmenten vorkommt.

Teilnehmer	OnlyPairs	OnlyButOnePairs	OverlappingCandidates
Baker	33	1	197
Baxter	36	32	0
Kamiya	0	0	965
Kamiya (vol.)	0	0	857
Merlo	0	0	22
Merlo (vol.)	1	0	22

Abbildung 5.86: Weitere Werte von j2sdk1.4.0-javax-swing (2 %, Good)

### 5.2.9. Zusammenfassung

Ein interessanter Wert, den es beim Vergleich aller Projekte miteinander zu betrachten gilt, ist die Anzahl der Referenzen pro 1K SLOC. In Abbildung 5.87 sind die Werte für alle acht Projekte aufgelistet. Es gilt zu beachten, dass dies nur eine untere Schranke der tatsächlichen Klonhäufigkeit darstellt: Bewertet man mehr als nur 2 % der Kandidaten, wird die Anzahl der Referenzen auf jeden Fall noch ansteigen und damit auch die Anzahl der Referenzen pro Zeilen.

Projekt	Referenzen pro 1K SLOC	Kandidaten pro 1K SLOC
weltab	22.91	1263.73
cook	5.03	339.02
sns	7.85	576.79
postgresql	2.36	251.55
netbeans-javadoc	2.89	413.68
eclipse-ant	0.86	69.71
eclipse-jdtcore	9.09	627.74
j2sdk1.4.0-javax-swing	3.81	275.79

Abbildung 5.87: Klone pro 1000 Zeilen Quellcode (2 %)

Betrachtet man die steigenden Größen der Projekte von weltab nach postgresql bei den C-Projekten und von netbeans-javadoc hin zu j2sdk1.4.0-javax-swing bei den Java-Projekten, so fällt auf, dass hier keine Aussage über die Klonhäufigkeit in Abhängigkeit zur Projektgröße getroffen werden kann. Die Menge vorkommender Klone ist vielmehr tatsächlich von der Software-Architektur des jeweiligen Projektes abhängig. Es gibt größere Projekte, die weniger Klone pro SLOC aufweisen, aber auch größere Projekte, die eine größere Häufigkeit von Klonen zeigen.

Es fällt allerdings auf, dass die Projekte, die in Java geschrieben sind, im Schnitt doch eine niedrigere Häufigkeit von Klonen aufweisen als die Projekte, die in C geschrieben sind. Ob dies Zufall ist und in der Auswahl der Projekte begründet liegt oder ob sich diese Tendenz tatsächlich bestätigt, müsste eine eingehendere Untersuchung mit einer größeren Anzahl von Projekten zeigen. Vier Projekte pro Programmiersprache reichen nicht aus, um hier eine allgemein gültige Aussage treffen zu können.

In Abbildung 5.88 auf der nächsten Seite zeigt sich, dass auch die Größe der Klone nicht unbedingt von der Programmiersprache abhängt. Bei den zwei „Ausreißern“ in cook und sns handelt es sich jeweils um zwei von Bison bzw. Yacc generierten Dateien. Alle weiteren Referenzen sind kleiner als 400 Zeilen. Bis auf die zwei „klonarmen“ Projekte netbeans-javadoc und eclipse-ant sind immer Klone von über 250 Zeilen Code vorhanden. Die durchschnittliche Größe bewegt sich ebenfalls unabhängig von der Sprache zwischen etwa 15 und 45 Zeilen.

Wie sich den Abbildungen der vorangegangenen Abschnitten entnehmen lässt, ist auch die Verteilung auf die drei Klontypen nicht sprachspezifisch unterschiedlich. Generell ist der 2. Klontyp immer am häufigsten vertreten. Dies ist noch einmal in Abbildung 5.89 auf der nächsten Seite zusammengefasst.

### 5.3. Abschließende Bewertung der Werkzeuge

Projekt	MaxReferenceSize	AvgReferenceSize
weltab	391	39.47
cook	645	22.41
snns	1064	22.30
postgresql	322	14.08
netbeans-javadoc	122	22.95
eclipse-ant	83	15.30
eclipse-jdtcore	267	15.53
j2sdk1.4.0-javax-swing	309	43.10

Abbildung 5.88: Klongrößen der Referenzen im Vergleich (2 %)

Projekt	Typ 1 (%)	Typ 2 (%)	Typ 3 (%)
weltab	20.24	67.46	12.30
cook	9.45	79.60	10.95
snns	12.74	60.69	26.57
postgresql	3.60	70.27	26.13
netbeans-javadoc	10.91	60.00	29.09
eclipse-ant	13.33	80.00	6.67
eclipse-jdtcore	8.92	64.39	26.69
j2sdk1.4.0-javax-swing	18.66	76.58	4.76

Abbildung 5.89: Anteil der Klontypen (2 %)

Man sieht deutlich, wie Typ 2 immer überwiegt. Bei Typ 1 und Typ 3 gibt es allerdings keine Indizien für irgendeine Abhängigkeit, weder von der Größe der Systeme, noch von der verwendeten Programmiersprache.

Zusammenfassend kann also gesagt werden, dass die Wahl der Programmiersprache an sich keinen automatischen Einfluss auf die Häufigkeit und die Beschaffenheit der Klone im Projekt hat. Es kommt viel mehr darauf an, welche Software-Architekturen und welche Programmierstile die Entwickler der Softwaresysteme wählen. Eine Tendenz, dass in Java weniger geklont wird als in C, scheint vorhanden zu sein. Ob dies allerdings an den OO-Eigenschaften der Sprache liegt, lässt sich ohne genauere Untersuchungen, die den Rahmen dieser Ausarbeitung sprengen würden, nicht sagen.

### 5.3. Abschließende Bewertung der Werkzeuge

Bevor nun auf die einzelnen Teilnehmer eingegangen wird, sollen noch Vergleiche angestellt werden, die bisher in der Auswertung nach Projekten so nicht durchgeführt wurden. Da sich diese Vergleiche auf die Teilnehmer untereinander beziehen, passen sie aber nicht in die darauf folgenden Abschnitte. Aus diesem Grund werden sie nun hier gesondert betrachtet, da sie nicht vorenthalten werden sollen. Im Anschluss werden dann die Ergebnisse der Teilnehmer zusammengefasst.

## 5. Ergebnisanalyse

### 5.3.1. Vorbemerkungen

In Abbildung 5.90 ist die Anzahl der gefundenen Klonpaare eines Werkzeuges zu erkennen, die sich exakt mit einer Referenz überdecken, d. h. für die  $\text{Good-Match}(0.7) = 1$  mit der jeweiligen Referenz gilt. Die Prozentangabe bezieht sich auf die Anzahl aller Referenzen, die der Teilnehmer mit einem Good-Match getroffen hat.

Teilnehmer	Referenzen	in %
Baker	929	50.68
Baxter	471	62.88
Kamiya	596	23.93
Kamiya (vol.)	595	23.84
Krinke	34	4.33
Merlo	726	67.53
Merlo (vol.)	748	67.75
Rieger	211	40.11

Abbildung 5.90: Referenzen, die mit  $\text{good} = 1$  getroffen werden, absolut und prozentual bezogen auf alle Referenzen mit  $\text{Good-Match}(0.7)$  (für 2 %-Auswertung)

Hierbei zeigt sich, dass Krinkes gefundene Referenzen zu über 95 % nicht exakt der getroffenen Referenz entsprechen, sondern größer, kleiner oder verschoben sind. Bei Baker ist jeder zweite Treffer mit einer Referenz gleichzeitig eine exakte Überdeckung mit dieser Referenz. Merlo unterstreicht mit den zwei Dritteln noch einmal die Qualität der Klonpaare, die er meldet.

Projekt	Typ 1	in %	Typ 2	in %	Typ 3	in %	Insgesamt	in %
Baker	285	82.13	1329	98.52	—	—	1614	93.13
Baxter	34	18.09	540	99.26	—	—	574	76.64
Kamiya	—	—	—	—	—	—	—	—
Kamiya (vol.)	—	—	—	—	—	—	—	—
Krinke	—	—	—	—	102	100.00	102	72.34
Merlo	165	94.29	618	83.63	61	95.31	844	86.30
Merlo (vol.)	165	94.29	618	82.51	80	96.39	863	85.70
Rieger	—	—	—	—	—	—	—	—

Abbildung 5.91: Korrekt kategorisierte Referenzen (2 %, Good)

Ein anderer interessanter Aspekt, der bisher nur indirekt und ungenau aus den Graphiken der Abschnitte 5.2.1 auf Seite 60 bis 5.2.8 auf Seite 102 abgelesen werden kann, ist die Güte der Voraussagen des Klontyps bei gefundenen Klonpaaren.

Anhand der Werte aus Abbildung 5.91 lässt sich dies genauer ablesen. Wie bereits bei der Beschreibung mancher Diagramme erläutert, können auch hier nur Werte für diejenigen Teilnehmer genannt werden, deren Werkzeuge überhaupt eine Kategorisierung vornehmen. Daher bleiben bei manchen Teilnehmern einige (oder alle) Spalten leer.

### 5.3. Abschließende Bewertung der Werkzeuge

Die Tatsache, dass Krinke alle Typ-3-Referenzen richtig einstuft, ist recht einfach erklärt: Er stuft alle Kandidaten als Typ 3 ein, daher eben auch diejenigen, die wirklich vom 3. Klontyp sind. In Wirklichkeit stimmt diese Einstufung nur zu gut 72 %, da die anderen seiner Treffer keinen Typ 3, sondern Typ 2 oder gar Typ 1 treffen. Baxters Problem, exakte Kopien zu kategorisieren, zeigt sich auch hier deutlich: Nur 18 % der Referenzen vom Typ 1, die er findet, stuft sein Werkzeug auch als Typ 1 ein. Deutlich besser, nahezu perfekt, ist sein Wert für die Typ-2-Klone. Merlo hat gerade bei diesem Klontyp größere Probleme. Bei Typ 1 und Typ 3 liegt er mit nur ca. 5 % Abweichung immer richtig, bei den Referenzen des Typs 2 allerdings sind 17 % seiner treffenden Kandidaten mit dem falschen Typ versehen. Auf alle Klontypen bezogen, die das Werkzeug kategorisieren kann, liefert Baker die genaueste Einteilung. Nur bei 7 % liegt sie daneben.

#### 5.3.2. Baker

Baker hatte mit ihrem Werkzeug *Dup* keinerlei Probleme mit den Projekten im Experiment. Es kann also recht einfach auf Systeme verschiedener Größe und Programmiersprachen angewendet werden.

Die Ergebnisse, die *Dup* liefert, sind so einzuordnen, dass vorwiegend ein hoher Recall erreicht wird und dabei eine niedrigere Precision in Kauf genommen wird.

*Dup* findet anteilig mehr Referenzen vom Typ 1 als vom Typ 2. Allerdings funktioniert die Kategorisierung der Typ-2-Klone etwas besser, da manche Kandidaten vom Typ 1 fälschlicherweise als Typ 2 eingestuft werden.

Ein Nachteil, wenn man die Ergebnisse des Werkzeugs automatisch weiterverwenden will, ist die Tatsache, dass zwischen 0 und 9 % der gemeldeten Kandidaten (je nach Projekt) sich überlappen. Dies sind somit keine Klone, die sich zur Generalisierung oder zum Ersetzen durch Funktionen oder Makros eignen.

Die Größen der gefundenen Klone bewegen sich im Mittelfeld aller angetretenen Teilnehmer. Sowohl die maximale Klongröße ist mit 352 als „normal“ anzusehen als auch die durchschnittliche Klongröße, die mit 23.26 mitten im Feld von 20 – 25 liegt, welches bei den meisten Teilnehmern vorherrscht.

Im Experiment entdeckt *Dup* 200 Referenzen, die außer Baker kein anderer Teilnehmer findet. Es werden insgesamt 10 Referenzen in den Projekten von allen anderen Teilnehmern gefunden, die Baker nicht trifft.

25 der 200 Referenzen sind vom Typ 1. Dabei handelt es sich fast ausschließlich um Anweisungssequenzen, die unverändert kopiert wurden. Sechs sind vom 3. Klontyp und setzen sich aus Methoden und Anweisungssequenzen zusammen. Die restlichen Referenzen, die nur Baker gefunden hat, sind vom Typ 2 und auch hier handelt es sich zum größten Teil um Sequenzen von Anweisungen.

Bei den 10 nur von Baker nicht gefundenen Referenzen handelt es sich um neun Funktionen von Typ 2 aus *postgresql* und einer Methode vom Typ 2 aus *j2sdk1.4.0-javax-swing*.

Von den 50 in den Projekten versteckten Klonen hat Baker 21 mit *Good-Match(0.7)* überdeckt und 38 mit *OK-Match(0.7)*. Insgesamt sind 13 Klone vom Typ 1 versteckt, wovon Baker 11 findet. Beim Typ 2 findet sie lediglich 8 von 18 versteckten Referenzen. Von den 19 Typ-3-Klonpaaren bleiben bis auf zwei alle anderen unentdeckt.

## 5. Ergebnisanalyse

Es kommt ziemlich häufig vor, dass Baker anstelle eines größeren Klons zwei (oder mehrere) kleinere meldet, die sich dann entweder überlappen oder eine Lücke dazwischen haben, da diese Zeile(n) den großen Klon zu einem Typ 3 machen würde.

Es gibt aber auch eine ganze Menge von Kandidaten, deren Start oder Ende „unsauber“ ist, d. h. um eine oder mehrere Zeilen von der Referenz abweicht.

Projekt	Benutzerzeit (Sec.)	Speicher (MB)
weltab	0.8	6
cook	3.2	18
snns	5.5	31
postgresql	11.7	62
netbeans-javadoc	0.6	6
eclipse-ant	1.1	8
eclipse-jdtcore	9.3	48
j2sdk1.4.0-javax-swing	9.2	50

Abbildung 5.92: Ressourcenverbrauch von Dup

Der Verbrauch von Ressourcen während der Analyse der Systeme ist in Abbildung 5.92 dargestellt. Bei den Zeiten handelt es sich um die Benutzerzeit, d. h. wirklich die Zeit, die ein Anwender von Dup auf die Analyse warten muss. Die Projekte des Experiments wurden unter IRIX mit einer MIPS R10000 mit acht 250 MHz IP27 Prozessoren und 4 GB Hauptspeicher bearbeitet. Allerdings lief Dup nur auf einem der acht Prozessoren (siehe [12]).

Zusammenfassend kann gesagt werden, dass Dup ein Werkzeug ist, mit dem man viele Klone, vor allem exakte Kopien, aber auch parametrisierte Kopien, auffinden kann. Allerdings ist die Anzahl der „False Positives“ doch recht hoch und die Genauigkeit, mit der die Klone gefunden werden, ist auch nicht immer optimal, so dass eine automatische Weiterverwertung der Ergebnisse schwierig ist.

### 5.3.3. Baxter

Baxters CloneDR™ ist dagegen ein sehr empfindliches Werkzeug, was die Art der Eingabe betrifft. Bei der Analyse der C-Quelltexte traten sehr viele Probleme auf, die auf Eigenarten der Programmiersprache C zurückzuführen sind. Die Probleme sind auf die Tatsache zurückzuführen, dass CloneDR in jedem Zweig von Präprozessor-Conditionals korrektes ANSI C erwartet. Oft wählen diese Präprozessor-Conditionals jedoch mit einem Zweig ANSI C und mit dem anderen K&R C oder GNU C aus. Solche Fälle treten jedoch in vielen Systemen auf, so dass das Einsatzgebiet von CloneDR™ sich etwas beschränkt und eingengt darstellt. Bei Java-Code sind diese Nachteile nicht aufgetreten.

Das Werkzeug liefert einen sehr niedrigen Recall, dafür aber eine um so höhere Precision, wie an den Auswertungen der einzelnen Projekte zu sehen ist.

CloneDR™ liefert beim Auffinden von Typ-1-Klonen bessere Ergebnisse als beim Finden von Typ-2-Klonen, wenn man die Werte für Recall und Precision betrachtet. Jedoch werden viele Typ-1-Klonpaare nicht korrekt eingestuft, sondern als Kandidaten vom Typ 2 gemeldet.

### 5.3. Abschließende Bewertung der Werkzeuge

Positiv fällt auf, dass Baxter in keinem einzigen Projekt einen Kandidaten liefert, dessen Codefragmente sich überlappen. Dies ist für Baxter eine notwendige Eigenschaft, da sonst CloneDR™ kein automatisches Ersetzen der gefundenen Klone durch generalisierte Funktionen oder Makros durchführen kann.

Ebenfalls positiv zu erwähnen ist, dass der Unterschied der getroffenen Referenzen zwischen einer Betrachtung mit OK-Match(0.7) und einer Betrachtung mit Good-Match(0.7) oft nur klein ist (mit Ausnahme des Projektes weltab). Dies bedeutet, dass seine Kandidaten die Referenzen generell gut überdecken. Dies sieht man auch an der großen Menge von exakten Überdeckungen (siehe 5.90 auf Seite 110).

Die Durchschnittsgröße der gefundenen Referenzen ist bei Baxter auch im guten Mittelfeld. Mit 22.46 Zeilen liegt er genau in der Mitte der Werte, welche die meisten Werkzeuge als Durchschnittsgröße liefern. Der größte Klon, den CloneDR™ im Experiment geliefert hat, ist 219 Zeilen lang. Dies ist etwas weniger, als die meisten anderen Werkzeuge liefern.

In allen acht Projekten findet Baxter 109 Referenzen, die außer ihm kein zweiter meldet; und es gibt 37 Referenzen, die alle anderen Teilnehmer außer ihm finden.

Von den 109 Referenzen, die nur er findet, sind neun vom Typ 1, wobei es sich ausschließlich um kleine Methoden oder `if-else`-Anweisungen handelt. Sechs weitere sind vom Typ 3. Dabei handelt es sich um Klassen, Methoden, Funktionen sowie Teile von `switch-case`-Anweisungen. Der Rest ist vom Typ 2 und umfasst fast ausschließlich komplette Methoden und Funktionen. Es sind aber auch kopierte Strukturen darunter.

Die 37 Referenzen, die von allen Teilnehmern entdeckt werden, nur von CloneDR™ nicht, teilen sich auf in 11 exakte Kopien von Blöcken und Methoden und 26 parametrisierte Kopien von Blöcken, Methoden und Funktionen.

14 der 50 versteckten Klonpaare werden von CloneDR™ mit Good-Match(0.7) und 18 mit OK-Match(0.7) gefunden. Vom Typ 1 findet er fünf Klone (was 38.5 % entspricht), vom Typ 2 sind es acht (was 44.4 % entspricht) und vom Typ 3 ist es gerade ein Klonpaar (was 5.3 % entspricht), das er findet.

Projekt	Benutzerzeit (Sec.)	Speicher (MB)
weltab	132	135
cook	401	174
sns	10800	628
postgresql	9780	702
netbeans-javadoc	91	118
eclipse-ant	705	321
eclipse-jdtcore	2695	437
j2sdk1.4.0-javax-swing	9780	658

Abbildung 5.93: Ressourcenverbrauch von CloneDR™

In Abbildung 5.93 ist die Benutzerzeit und der Speicherverbrauch von CloneDR™ im Experiment dargestellt. Die Zeiten sind von Hand gemessen, da die intern verwendeten Variablen zur Zeitmessung bereits nach 20 Sekunden überlaufen. Bei den großen Projekten wurde darüber hinaus nur minutengenau gemessen. Der Speicherverbrauch ist der von PAR-

## 5. Ergebnisanalyse

LANSE angegebene Höchstwert während der Analyse. Im Experiment wurde ein Pentium Pro mit 200 MHz unter Windows NT 4.0 verwendet.

Die Klone, die von Baxters CloneDR™ gefunden werden, werden sehr gut getroffen. Die Precision ist gut, d. h. nur wenige seiner Kandidaten sind fehlerhaft, aber er bezahlt das mit einem niedrigen Recall, den er aber für seine Zwecke in Kauf nimmt. Für den Zweck, den CloneDR™ erfüllen soll, ist es jedoch auch nicht wichtig, restlos alle Klone zu identifizieren, daher genügt das Werkzeug völlig den gestellten Anforderungen.

### 5.3.4. Kamiya

Mit keinem der acht Projekte des Experiments hatte CCFinder von Kamiya Probleme. Alles wurde, ohne Modifikationen machen zu müssen, von ihm akzeptiert.

Kamiyas Werkzeug liefert sehr viele Kandidaten. Daraus folgt, dass die Precision sehr niedrig ist: Auf eine getroffene Referenz kommen manchmal bis zu 250 Kandidaten, die uninteressant sind. Auf der anderen Seite liefert das Werkzeug einen besonders hohen Recall.

Vergleicht man die drei Klontypen und das Abschneiden von CCFinder aufgeschlüsselt auf die Typen, so fällt sehr deutlich auf, dass die Klone vom Typ 3, die laut Kamiya sowieso nur in sehr eingeschränktem Maße erkannt werden, in der Tat nicht sehr gut erkannt werden. Sowohl der Recall als auch die Precision vom Typ 3 sind nicht besonders gut. Die anderen zwei Typen werden jedoch relativ gut erkannt. Meist ist der Recall bei Typ 1 besser als bei Typ 2.

Kamiyas Werkzeug kann jedoch keine Einstufung der gefundenen Kandidaten in die drei Klontypen vornehmen.

Es fällt auch auf, dass gerade bei den Projekten in der Programmiersprache Java der Anteil der Kandidaten, die bewertet und verworfen worden sind, besonders hoch ist. Dies könnte bedeuten, dass seine Transformationen für Java noch nicht optimal sind.

CCFinder ist ein Werkzeug, welches einen hohen Anteil an sich überlappenden Klonpaaren liefert. Im Experiment bewegt sich dieser Wert zwischen 2 % und knapp 16 %.

Die durchschnittliche Größe der gefundenen Referenzen ist bei Kamiya mit 23.7 Zeilen wieder genau im Bereich der meisten anderen Werkzeuge auch. Ebenso ist der größte erkannte Klon mit 345 Zeilen im Größenbereich, in dem viele andere Werkzeuge ebenfalls ihren maximalen Klon entdeckt haben.

Betrachtet man die Anzahl der Referenzen, die nur CCFinder findet und sonst kein zweites Werkzeug, und die Anzahl der Referenzen, die Kamiya als einziger nicht findet, obwohl sie alle anderen Teilnehmer melden, so fällt auf, dass beide Werte bei der Pflicht-Abgabe von Kamiya 0 sind.

In seiner „Kür“-Abgabe mit modifiziertem CCFinder findet Kamiya vier Referenzen, die sonst niemand erkennt. Es gibt weiterhin keine Referenzen, die außer ihm alle anderen überdecken. Unter den vier Referenzen ist eine parametrisierte Kopie einer Klasse und drei Typ-3-Klone, die alle drei switch-case-Anweisungen enthalten.

Kamiya erkennt 37 der 50 versteckten Klone mit OK-Match(0.7) und 23 davon mit Good-Match(0.7). Von den 13 versteckten Typ-1-Klonen bleiben nur drei unentdeckt, von den versteckten Typ-2-Klonen findet er 10, und drei der Typ-3-Referenzen werden gefunden.

### 5.3. Abschließende Bewertung der Werkzeuge

Bei den Kandidaten, welche die versteckten Referenzen überdecken, fällt auf, dass häufig zwei Kandidaten zusammen die Referenz ergeben. Vor allem bei den versteckten Klone vom Typ 3 ist dies sehr oft der Fall, da der Teil des Klons vor der eingefügten oder gelöschten Zeile und der Teil danach als zwei separate Klone vom Typ 1 oder Typ 2 erkannt werden.

Insgesamt kann gesagt werden, dass seine „Kür“-Abgabe wesentlich besser ist als die Pflicht-Abgabe. Dies wird vor allem bei Java deutlich, da hier seine Veränderungen wohl besser greifen.

Projekt	Benutzerzeit (Sec.)	Speicher (KB)
weltab	5.5	8056
cook	26.9	14132
sns	20.4	27160
postgresl	33.9	47520
netbeans-javadoc	10.6	8056
eclipse-ant	7.8	12752
eclipse-jdtcore	151.3	44412
j2sdk1.4.0-javax-swing	184.0	44348

Abbildung 5.94: Ressourcenverbrauch von CCFinder

In Abbildung 5.94 sind Benutzerzeit und Speicherverbrauch für die Analyse im Pflicht-Teil dargestellt. Bei der Benutzerzeit handelt es sich um die Zeit, die sich der Rechner im Benutzermodus oder im Systemmodus befindet. Speicher misst den maximalen Verbrauch während des Analyselaufs. Es werden zusätzlich temporäre Dateien angelegt; dieser Speicherbedarf ist hierbei nicht berücksichtigt. Die Werte für den „Kür“-Teil sollten laut Kamiya nicht von den Werten des Pflicht-Teils abweichen. CCFinder lief im Experiment auf einem Pentium 4 mit 1.4 GHz und 512 MB Speicher unter der japanischen Version von Windows 2000.

Sein Werkzeug findet die meisten Klone, produziert dabei aber auch am meisten „False Positives“. Es gilt daher abzuwägen, was einem wichtiger ist. Das Werkzeug ist sehr unempfindlich gegenüber den Eingabedaten.

#### 5.3.5. Krinke

Zuerst muss nochmals erwähnt werden, dass Krinke die doppelte Bearbeitungszeit hatte. Da bei ihm unerwartete Probleme aufgetreten sind (siehe Abschnitt 3.5.4 auf Seite 37), benötigte er acht Wochen anstelle der ausgemachten vier.

Doch auch dann konnte er das Projekt postgresl nicht analysieren, da es zu groß für sein Werkzeug ist. Bei den Projekten cook und sns stellte er eine extrem hohe Laufzeit von Duplix fest, weiß aber noch nicht, worin dies begründet liegt. Es scheint aber so, dass Krinkes Werkzeug für größere Systeme nicht geeignet ist.

Hier spielt unter anderem auch das Manko eine Rolle, dass spezielle Header-Dateien zuerst von Hand erstellt oder bearbeitet werden müssen, damit ein Projekt, welches Bibliotheken einbindet, analysiert werden kann.

## 5. Ergebnisanalyse

Ein weiterer Nachteil von Duplix ist der fehlende Java-Support. Momentan können nur C-Projekte untersucht werden.

Bezüglich der Werte für Recall und Precision muss leider gesagt werden, dass bei Duplix weder ein hoher Recall noch eine hohe Precision erreicht wird. Lediglich für Typ-3-Klone ist ein relativ hoher Recall zu verzeichnen. Am niedrigsten ist der Recall bei den Typ-2-Referenzen.

Krinkes Duplix nimmt keine Einstufung der gefundenen Kandidaten in die verschiedenen Klontypen vor. Da allerdings „ähnlicher Code“ gemeldet wird, lieferte Duplix im Experiment alle Kandidaten als Typ 3 zurück.

Einen deutlichen Unterschied zu den anderen Teilnehmern erkennt man auch beim Betrachten der durchschnittlichen Größe der gefundenen Referenzen. Sie beträgt bei Krinke über 87 Zeilen, bei den anderen Teilnehmern liegt sie zwischen 20 und 25 Zeilen. Dies liegt wieder einmal begründet in der Tatsache, dass Krinke eigentlich Klone mit Löchern in der Mitte meldet, dies aber im Experiment nicht berücksichtigt wird. Die maximale Klongröße liegt mit 349 Zeilen im Rahmen der Werte der anderen Teilnehmer.

Der Unterschied von OK-Match(0.7) zu Good-Match(0.7) ist bei ihm auch so groß wie bei keinem anderen Teilnehmer. Dies liegt daran, dass seine Kandidaten sehr groß sind und daher viele Referenzen überdecken, aber nicht genau genug.

Positiv zu bemerken ist, dass Krinke keine sich überlappenden Kandidaten liefert.

Auch erkennt er in den drei von ihm bearbeiteten Projekten 104 Referenzen, die außer ihm niemand findet. Und acht Referenzen werden von allen anderen Teilnehmern gefunden, nur nicht von ihm.

Die 104 einzig von ihm gefundenen Referenzen teilen sich wie folgt auf: Eine identisch geklonte Anweisungssequenz, neun parametrisierte Kopien (darunter meist Anweisungssequenzen, aber auch Funktionen) sowie 94 Klone vom Typ 3. Diese setzen sich aus größeren Anweisungssequenzen und kompletten Funktionen zusammen.

Wie bereits erwähnt, findet Krinke in den drei von ihm analysierten Systemen acht Referenzen nicht, die alle anderen Teilnehmer gefunden haben. Es handelt sich hierbei ausschließlich um ganze Funktionen kleinerer und mittlerer Größe. Zwei davon sind identische Kopien und die restlichen sechs sind parametrisierte Kopien.

Von den 23 in den Projekten `weltab`, `cook` und `sns` versteckten Klonen hat Duplix nur einen einzigen gefunden (sowohl mit OK-Match(0.7) als auch mit Good-Match(0.7)). Es handelt sich dabei um eine parametrisierte Kopie einer Funktion in eine andere Datei aus dem Projekt `cook`.

Projekt	Benutzerzeit (Sec.)	Speicher (KB)
weltab	36285	13188
cook	882589	12780
sns	228772	64664

Abbildung 5.95: Ressourcenverbrauch von Duplix

Abbildung 5.95 zeigt den Ressourcenverbrauch von Duplix für die drei bearbeiteten Projekte. Die Benutzerzeit ist die Zeit, die für die Vergleiche im PDG benötigt wird, nachdem

der Graph in den Speicher geladen wurde. Beim Speicherverbrauch handelt es sich um die Menge von Speicher, die für das Programm `DupliX` selbst und die Daten des jeweiligen Graphen benötigt werden. Die angegebenen Werte beziehen sich nicht auf den maximalen Speicherverbrauch, sondern geben den Speicherverbrauch bei Programmende an. Während der eigentlichen Analyse wird der Speicherverbrauch noch höher liegen. Dazu liegen jedoch keine Angaben vor. `cook` lief auf einem 1 GHz Dual-Xeon-Rechner, wobei ein Prozessor für die Systemaufgaben von GNU/Linux abgestellt war und der andere Prozessor somit uneingeschränkt die Klonanalyse bearbeiten konnte. Die anderen beiden Projekte sind entweder mit dem gleichen Rechner analysiert worden oder mit einem etwas langsameren. Dies konnte von Krinke nicht mehr genau festgestellt werden.

Krinkes Werkzeug zeigt vor allem bei den Klonen des 3. Typs Potential. Allerdings hat es dafür bei den Typen 1 und 2 derartige Schwächen, dass es für einen ernsthaften Einsatz zum Auffinden und Ersetzen von ungewollten Klonen in großen Softwaresystemen nicht geeignet erscheint. Dazu kommt das Manko, dass es auf die Programmiersprache C fixiert ist.

#### 5.3.6. Merlo

Bis auf ein Problem beim Parsen von C-Dateien hatte Merlo kein Problem mit den Projekten des Experiments.

Das Werkzeug `CLAN` liefert vorwiegend eine sehr hohe Precision und dafür einen niedrigen Recall. Die Tatsache der besonders guten Precision wird untermauert durch den niedrigsten Anteil verworfener Kandidaten, verglichen mit den anderen Teilnehmern (mit der Ausnahme des nicht repräsentativen `netbeans-javadoc` und `eclipse-jdtcore`, wo Baxter besser ist).

Auffällig ist bei Merlo, dass er keine überragende Typerkennung für einen bestimmten Klontyp hat. Je nach Projekt ist es einmal Typ 1, einmal Typ 2 und manchmal auch Typ 3, den er am besten erkennt. Dies scheint wohl eine Eigenart der Metriken zu sein. Interessanterweise ist die Klassifizierung der Kandidaten in die entsprechenden Typen zu einem sehr großen Anteil korrekt. Besonders gut funktioniert die Klassifizierung der Typen 1 und 3.

Noch ein Hinweis auf die gute Qualität der von ihm gefundenen Referenzen ist der Vergleich der Werte bei `OK-Match(0.7)` mit den Werten bei `Good-Match(0.7)`: es sind (bis auf weltab) nur sehr wenige Referenzen, die ein `OK-Match`, aber kein `Good-Match` sind.

Ein weiterer bemerkenswerter Punkt ist die Tatsache, dass er in den C-Projekten überhaupt keine sich überlappenden Klone meldet. In den Java-Projekten überlappen sich bis zu 5.7% der Kandidaten mit sich selbst. Jedoch sind alle diese Überlappungen auf `if-else-if`-Konstrukte zurückzuführen, deren `else if` Zeile sowohl die letzte Zeile des ersten Codefragments als auch die erste Zeile des zweiten Codefragments ist.

Sowohl die durchschnittliche Größe der gefundenen Referenzen als auch die größte von ihm entdeckte Referenz sind im Vergleich mit den anderen Teilnehmern kleiner. Der maximale Klon ist mit 206 Zeilen um über 100 Zeilen kleiner als der der meisten anderen Teilnehmer. Auch eine Durchschnittsgröße von 17.87 liegt nicht mehr im Bereich von 20 bis 25 Zeilen. Dies ist wohl darin begründet, dass Merlo hauptsächlich ganze Funktionen einzeln als Kandidaten liefert. Andere Teilnehmer liefern manchmal mehrere Funktionen am Stück als einen Kandidaten.

## 5. Ergebnisanalyse

In seiner Pflicht-Abgabe kommt keine einzige Referenz vor, die nur er trifft, aber kein anderer. Ebenso wenig überdecken alle anderen eine Referenz, die er übersieht.

Im „Kür“-Teil kann er dies sogar etwas steigern: er entdeckt 25 Kandidaten, ohne dass sie ein anderer Teilnehmer meldet. Weiterhin übersieht er keine Referenz, die alle anderen Teilnehmer entdecken.

Unter obigen 25 Referenzen sind sechs parametrisierte Kopien von Methoden und 19 weitere Klone vom Typ 3. Bei all diesen handelt es sich um komplette Methoden.

Im Gegensatz zur „Kür“-Abgabe Kamiyas ist Merlos „Kür“ keine nennenswerte Verbesserung.

Von den insgesamt 50 versteckten Klonen findet Merlo 18 mit OK-Match(0.7) und 12 mit Good-Match(0.7). Von den 13 Typ-1-Klonen findet er fünf, vom Typ 2 entdeckt er sieben der 18 versteckten Klone und keinen vom Typ 3.

Projekt	Benutzerzeit (Sec.)	Speicher (page faults)
weltab	0.39	509
cook	0.35	471
sns	0.70	633
postgresql	0.92	714
netbeans-javadoc	0.25	440
eclipse-ant	0.21	410
eclipse-jdtcore	3.35	1875
j2sdk1.4.0-javax-swing	2.50	1390

Abbildung 5.96: Ressourcenverbrauch von CLAN

In Abbildung 5.96 ist die Benutzerzeit und der Speicherverbrauch dargestellt. Bei den Zeiten handelt es sich wirklich die Zeit, die ein Anwender von CLAN auf das Ergebnis der Analyse warten muss. Beim Speicherbedarf liegen keine genauen Werte vor. In der Spalte „page faults“ ist angegeben, wie oft ein Speicherseitenfehler vorlag. Wichtig zu wissen ist noch, dass eine Speicherseite auf der Intel x86-Architektur (welche Merlo benutzt) 4 KB entspricht. Somit erhält man durch Multiplikation der Speicherseitenfehler mit der Speicherseitengröße eine Obergrenze des Speicherbedarfs von CLAN. Bei den Werten in der Abbildung ist allerdings nur die Klonsuche und Typbestimmung enthalten, Parsing und Datei-Handling sowie Eingabe und Ausgabe ist nicht inbegriffen. Für das Experiment wurde CLAN auf zwei unter GNU/Linux laufenden Maschinen eingesetzt: einem Intel Pentium 4 mit 1,6 GHz und 256 MB Speicher, auf welchem der C-Parser lief und einem Intel Pentium II mit 400 MHz und 324 MB Speicher, auf welchem der Java-Parser und die Klonerkennung liefen (siehe [26]).

Abschließend ist zu sagen, dass CLAN eine außerordentlich hohe Precision hat und die Typen sehr gut einstuft. Auch werden die Klonegrenzen sehr gut getroffen (wie man an der hohen Anzahl korrekter Treffer aus Abbildung 5.90 auf Seite 110 sehen kann). Will man allerdings möglichst viele Klone im System erkennen, so ist CLAN nicht das Werkzeug der Wahl. Allerdings verfügt CLAN über eine sehr nützliche Visualisierungsmethode, die es ermöglicht, gefundene Klone ohne weitere Hilfsmittel mit jedem HTML-Browser zu betrachten.

### 5.3.7. Rieger

Duploc konnte die zwei Projekte postgresql und j2sdk1.4.0-javax-swing aufgrund ihrer Größe nicht analysieren. Und anhand der Ergebnisse liegt es nahe, zu vermuten, dass auch bei eclipse-jdtcore kein normaler Betrieb des Werkzeugs möglich war. Es scheint so, als ob Duploc nicht für Systeme dieser Größenordnung geeignet ist.

Riegers Werkzeug liefert im Experiment Recall-Werte, die eher als hoch einzuordnen sind; dafür liegen die Precision-Werte im unteren Bereich.

Betrachtet man die Werte für den Recall, so ist zu erkennen, dass Duploc mehr Referenzen vom Typ 1 erkennt als Referenzen vom Typ 2. Den schlechtesten Recall hat Rieger beim 3. Klontyp.

Duploc von Rieger liefert keine Einstufung in die Klontypen. Diese Erweiterung des Werkzeuges ist geplant, aber nicht rechtzeitig zum Experiment fertig gestellt worden.

Die Differenz der gefundenen Referenzen bei OK-Match(0.7) und Good-Match(0.7) ist tendenziell größer als bei den anderen Werkzeugen.

Außerdem liefert Rieger sich überlappende Kandidaten. Je nach Projekt variiert dies von fast keinem bis zu über sieben Prozent.

Mit einem maximalen Klon von 385 Zeilen Länge findet Duploc die größte Referenz im Experiment. Die Durchschnittsgröße der von dem Werkzeug gefundenen Referenzen ist mit 19.83 Zeilen fast noch im Bereich der anderen Werkzeuge zwischen 20 und 25 Zeilen.

Rieger findet 83 Referenzen, die außer ihm niemand überdeckt. Außerdem gibt es 25 Referenzen, die von allen anderen gefunden werden, nur nicht von ihm.

Von den 83 nur von ihm gefundenen Klonen sind acht vom 3. Klontyp, drei sind exakte Kopien und die restlichen 72 sind parametrisierte Kopien. Bei allen drei Typen handelt es sich in der überwiegenden Mehrheit um Anweisungssequenzen. Aber auch komplette Funktionen sind vereinzelt darunter.

Die bereits erwähnten 25 Referenzen, die alle anderen Teilnehmer außer Rieger finden, lassen sich wie folgt aufschlüsseln: Bei acht der Referenzen handelt es sich um exakte Kopien von Methoden, switch-, if-else- oder try-catch-Anweisungen. 16 weitere Klone sind vom Typ 2 und setzen sich ebenfalls aus Methoden, switch- und if-else-Anweisungen sowie Funktionen zusammen. Der noch fehlende letzte Klon ist eine if-Anweisung, deren Kopie vom 3. Klontyp ist. Es handelt sich jeweils um komplette Blöcke, d. h., die Methoden und Anweisungen sind komplett kopiert.

Von 45 versteckten Klonen, die Rieger hätte in seinen sechs analysierten Projekten erkennen können, hat er 26 mit einem OK-Match(0.7) überdeckt und 14 mit einem Good-Match(0.7). Vom Typ 1 hat er acht von 13, vom Typ 2 fünf von 15 und vom Typ 3 nur noch eine von 17 versteckten Referenzen entdeckt.

Informationen zu Rechenzeit und Speicherverbrauch liegen zu Duplix leider nicht vor.

Ein wichtiger Vorteil von Duploc ist allerdings, dass es als einziges Werkzeug im Experiment unter der GNU General Public License steht und somit jedem frei zugänglich ist. Änderungen und Verbesserungen können daher problemlos integriert werden.

Die Tatsache, dass Duploc für größere Systeme nicht geeignet zu sein scheint, ist ein Problem, welches den Einsatz zum Aufspüren von Klonen in mitunter auch großen Softwaresystemen in Frage stellt.

## 5. Ergebnisanalyse

## 6. Kritischer Rückblick

Ein Punkt, der zu Beginn bei der Planung des Zeitbedarfs nicht berücksichtigt wurde, ist die Tatsache, dass eine Kooperation mit anderen Teilnehmern, die in verschiedenen Zeitzonen um den Globus herum beheimatet sind, sehr viel Zeit verschlingt. Da die Teilnehmer sowohl aus den USA und Kanada als auch aus Japan kommen, kann man aufgrund der Zeitverschiebung erst am Folgetag mit einer Beantwortung von E-Mails rechnen. Bedenkt man nun, dass vor allem in der Planungsphase des Experiments ein hoher Kommunikationsbedarf herrscht und viele E-Mails zwischen den einzelnen Teilnehmern ausgetauscht werden müssen, so wird schnell klar, dass der Faktor Zeit hier nicht unterschätzt werden darf.

Ein weiterer Punkt, den man nicht vernachlässigen kann, ist die Abhängigkeit des Schiedsrichters von den Teilnehmern. Das Einsenden der Daten zu den Abgabeterminen hat im Prinzip recht gut funktioniert, da die Teilnehmer sonst mit einer Disqualifikation hätten rechnen müssen. Allerdings war es äußerst mühsam, von manchen Teilnehmern weitere Informationen zu bekommen. So fehlen z. B. immer noch Informationen über Riegers Werkzeug.

Dennoch lief die Kooperation mit den Teilnehmern ansonsten reibungslos; alle anderen Fragen wurden beantwortet und die Diskussionen waren hilfreich und fruchtbar.

Das Experiment selbst betreffend hat sich der erste Test-Lauf als äußerst wichtig erwiesen. Durch ihn wurden die Schwächen des ersten Datenbank-Schemas und der zuerst gewählten Definition von Gleichheit zweier Klone aufgedeckt. Im Nachhinein betrachtet ist nun die Normierung des Codes (siehe Abschnitt 3.2 auf Seite 28) überflüssig, da durch die Berechnung der Überdeckungsgrade von Klon und Referenz dies nicht mehr zwingend notwendig ist. Wie bereits in Abschnitt 3.2 auf Seite 28 erläutert, können zeilenbasierte Werkzeuge durch die Normierung eventuell etwas mehr Klone erkennen als ohne. Jedoch könnte das entsprechende Werkzeug in der Praxis den Quelltext sowieso mit einem Pretty Printer formatieren, so dass die Normierung keinen wirklichen Eingriff darstellt.

## 6. Kritischer Rückblick

## 7. Zusammenfassung

Nachdem nun alle Daten erfasst und ausgewertet sind, hat sich bestätigt, was bereits zu Anfang vermutet werden konnte: „Den“ Sieger oder „die“ Siegerin gibt es nicht. Die verschiedenen Werkzeuge mit ihren unterschiedlichen Ansätzen und Zielen haben alle ihre Stärken und Schwächen.

Man kann die Werkzeuge grob in zwei Kategorien einteilen: Diejenigen, welche einen hohen Recall und dafür eine niedrige Precision liefern und diejenigen, welche einen niedrigen Recall in Kauf nehmen, um dafür eine hohe Precision erreichen zu können.

Baxter und Merlo gehören mit ihren Werkzeugen zu der Kategorie, die eine hohe Precision erreicht. Baker, Kamiya und Rieger liefern dafür einen hohen Recall. Krinke schafft es nur für Klone vom Typ 3, einen hohen Recall zu erreichen.

Mit dem rein tokenbasierten Ansatz, den Baker mit Dup verfolgt, findet sie sehr viele Kandidaten, von denen allerdings ein Großteil die Referenzen nicht qualitativ gut überdeckt. Gemessen an Recall und Precision ist Dup besser geeignet, um exakte Kopien zu erkennen, als um parametrisierte Kopien zu finden. Klone vom Typ 3 werden nur in Ausnahmefällen erkannt. Die Einteilung der gefundenen Kandidaten in die Klontypen funktioniert sehr gut. Ihr Werkzeug hat Stärken, wenn es darum geht, Sequenzen von Anweisungen als Klon zu erkennen, dafür existieren Probleme, die Klone genau einzugrenzen: oft sind Start- und Endezeile der Codefragmente nicht exakt. Von den versteckten Klonen werden 42 % qualitativ gut entdeckt.

Der AST-basierte Ansatz, den Baxter mit seinem CloneDR™ wählt, liefert deutlich weniger Kandidaten, dafür sind diese von guter Qualität, was eine hohe Precision zeigt. Auch bei ihm ist ein Vorsprung der Erkennung des 1. Klontyps im Gegensatz zum 2. Typ vorhanden. Klone vom Typ 3 werden nur in seltenen Fällen entdeckt. Die parametrisierten Kopien werden von CloneDR™ auch meist als solche erkannt. Jedoch werden die exakten Kopien auch häufig fälschlicherweise als Klone des 2. Typs identifiziert. Es zeigen sich keine besonderen Stärken oder Schwächen beim Erkennen von Methoden, Funktionen oder Strukturen. 28 % aller versteckten Klone im Experiment werden von CloneDR™ mit guter Qualität gefunden.

CCFinder von Kamiya benutzt wieder einen tokenbasierten Ansatz mit Eingabetransformationen. Ähnlich wie Baker findet er sehr viele Kandidaten, wobei es sich bei einem großen Teil davon nicht um qualitativ hochwertige Kandidaten handelt. Die Werte von Recall und Precision zeigen, dass Klone vom Typ 1 besser erkannt werden als Klone vom Typ 2. CCFinder erkennt auch einige Typ-3-Klone, allerdings deutlich schlechter als die exakten und parametrisierten Kopien. Kamiyas Werkzeug kann die gefundenen Kandidaten nicht in die Klontypen einteilen. Er erkennt alle Referenzen, die von allen anderen Teilnehmer auch erkannt werden. Besondere Stärken in der Art der gefundenen Klone lassen sich

## 7. Zusammenfassung

aufgrund der niedrigen Anzahl von Referenzen, die er als einziger Teilnehmer findet, nicht ausmachen. Mit 46 % findet er die meisten der versteckten Klone.

Krinke benutzt in seinem `Duplix` den Ansatz, die Kandidaten mittels eines PDG zu finden. Allerdings ist sein Werkzeug nicht für Java geeignet und kann keine großen Systeme analysieren. Des Weiteren müssen für das zu analysierende System zuerst von Hand Header-Dateien erstellt werden. Die Precision ist durchweg äußerst niedrig, der Recall ist lediglich für Klone vom Typ 3 sehr gut. Dies ist laut Krinke sowieso der einzige Typ, den `Duplix` erkennen kann. Das Werkzeug hat Stärken im Auffinden von großen Funktionen und Sequenzen von Anweisungen des 3. Typs. Kleinere Funktionen findet es hingegen selten und macht Fehler bei deren Grenzen. Von den Klonen, die in den Projekten versteckt wurden, die Krinke analysiert hat, findet sein Werkzeug nur einen einzigen, was einem Anteil von 4 % entspricht.

Mit `CLAN` hat Merlo ein Werkzeug im Rennen, das Klone anhand ihrer Metrikwerte aufspürt. `CLAN` liefert, wie bereits weiter oben erwähnt, gute Werte für Precision, dafür aber schlechtere für Recall. Diese Werte zeigen auch, dass Merlos Werkzeug die drei Klontypen relativ gleich gut erkennt: Je nach analysiertem Projekt schwanken die Anteile der drei Klontypen hin und her. `CLAN` nimmt eine Einstufung der gefundenen Kandidaten in die Klontypen vor, die sehr gut ist. Lediglich bei parametrisierten Kopien wird ab und zu nach oben zu Typ 1 oder nach unten zu Typ 3 abgewichen. Referenzen, die alle anderen Teilnehmern erkennen, werden allesamt von Merlo auch gefunden. Die Stärke seines Werkzeuges liegt darin, komplette Funktionen und Methoden zu entdecken; somit sind die Grenzen der Kandidaten, die `CLAN` meldet, meist sauber. Auf der anderen Seite kann es aber auch als Nachteil betrachtet werden, dass Anweisungssequenzen von ihm nur in einem anderen Modus (siehe Erklärung zur „Kür“ in Abschnitt 3.4.5 auf Seite 33) entdeckt werden können. Von den versteckten Klonen werden von Merlos Kandidaten 24 % gut überdeckt.

Rieger geht mit `Duploc` den Weg eines zeilenorientierten Ansatzes, der im Weiteren die Kandidaten per Pattern Matching entdeckt. Das Werkzeug scheint Probleme mit großen Projekten zu haben, so dass diese nicht analysiert werden können. `Duploc` findet meist sehr viele Kandidaten, wobei die Anzahl der tatsächlichen Referenzen wesentlich niedriger liegt. Dies drückt sich in einem hohen Recall und einer niedrigen Precision aus. Wie die Recall-Werte zeigen, ist Riegers Erkennung von Typ-1-Klonen effektiver als die Erkennung von parametrisierten Kopien. Am schlechtesten schneiden bei ihm die Typ-3-Klone ab. Sein Werkzeug liefert noch keine Einstufung der gefundenen Kandidaten in die Typen. Dies ist aber in einer späteren Version geplant. Seine Stärken liegen im Erkennen von Anweisungssequenzen. Dort erkennt er relativ viele Klone, die andere nicht erkennen. Dafür hat er Probleme, einige Kopien bedingter Anweisungen (`if-else`, `switch-case` sowie `try-catch`) zu erkennen, die alle anderen erfolgreich identifizieren können. Von den versteckten Klonen in den sechs von ihm analysierten Systemen findet er 31 %.

Beim Vergleich der Werkzeuge untereinander fällt auf, dass Baker und Kamiya sehr viele Referenzen gemeinsam finden. Auch Rieger findet noch einen Großteil dieser Referenzen. Betrachtet man die Anzahl der gemeldeten Kandidaten, so liegt sie bei Baker und Rieger meist etwa in der gleichen Größenordnung. Ebenso melden diese zwei Teilnehmer viele Kandidaten gemeinsam, welche beim Betrachten verworfen worden sind.

Dies scheint in der Tat darin begründet zu liegen, dass alle drei einen ähnlichen Ansatz

zu Grunde legen: Alle drei starten zeilen- bzw. tokenorientiert bei der Suche nach Klonen.

Im Gegensatz dazu fällt auf, dass die gefundenen Referenzen von Merlo und Rieger, von Baxter und Merlo, von Baxter und Krinke sowie die von Baker und Merlo größtenteils paarweise disjunkt sind. Hierbei handelt es sich immer um zwei völlig verschiedene Ansätze.

Zwischen Kamiya und Merlo gibt es auch Gemeinsamkeiten: Kamiya entdeckt fast alles auch, was Merlo findet. Andersherum gilt dies nicht, da – wie bereits mehrfach erwähnt – Kamiyas Recall deutlich höher ist als der Merlos. Des Weiteren haben beide keine Referenz übersehen, die von jeweils allen anderen Teilnehmern gefunden wird. D. h. beide finden alle „eindeutigen“ Klone.

Außer bei Krinke (und bei Kamiya im Projekt j2sdk1.4.0-javax-swing) ist bei den erkannten Typ-1-Klonen der Unterschied zwischen OK-Match(0.7) und Good-Match(0.7) gering. Dies zeigt, dass exakte Kopien von diesen Teilnehmern recht gut eingegrenzt und erkannt werden.

Bei Typ-3-Klonen steigt die Zahl der gefundenen Referenzen stark an, wenn man das Kriterium vom Good-Match auf OK-Match lockert. Da Klone dieses Typs sowieso manuelle Betrachtung erfordern, wäre es eventuell sinnvoll, hier nur das OK-Match-Kriterium anzulegen.

Einig sind sich alle Werkzeuge beim Verhältnis von gefundenen Klonen innerhalb von Datei zu Kopien über Dateien hinweg. Hier sind keine werkzeugspezifischen Besonderheiten zu sehen.

Nun ist es noch interessant, darüber nachzudenken, ob eine Kombination der verschiedenen Werkzeuge ein besseres Ergebnis liefern könnte.

Zunächst denkt man darüber nach, zwei oder mehrere Werkzeuge „hintereinander zu schalten“, also die Schnittmenge der Kandidaten zu bilden. Dies hat jedoch nur dann Sinn, wenn zum einen die Kandidaten eines Werkzeuges nicht bereits komplett in denen eines anderen Werkzeuges enthalten sind. Denn sonst könnte man gleich ausschließlich das Werkzeug mit der niedrigen Anzahl an Kandidaten hernehmen und würde das gleiche Ergebnis erhalten. Zum anderen wäre es dann nützlich, wenn die Werkzeuge größtenteils die gleichen Referenzen treffen, aber unterschiedliche FalsePositives melden. Dann würde nämlich das Bilden der Schnittmenge die Precision nach dem Schnitt erhöhen.

Im Experiment könnte man hier die Werkzeuge von Kamiya und Baker oder Kamiya und Rieger eventuell gewinnbringend „hintereinander schalten“. Wie bereits erwähnt, liefern Kamiya und Baker sowie Kamiya und Rieger jeweils paarweise sehr viele gemeinsame Referenzen, bei den Rejected sind jedoch wenig gemeinsame Kandidaten dabei.

Das Vorgehen, zwei Werkzeuge „hintereinander zu schalten“, hätte eventuell auch dann Sinn, wenn die Datenmenge mit dem ersten Werkzeug so stark reduziert werden könnte, dass die Bearbeitung mit dem zweiten Werkzeug überhaupt erst oder wenigstens effizienter möglich wird. Im Experiment hat sich jedoch gezeigt, dass die Werkzeuge, die eine hohe Precision liefern, keine solche „Vorselektion“ aus Geschwindigkeitsgründen nötig haben.

Eine interessante und in der Tat auch nützliche Idee ist es, die gefundenen Kandidaten von Werkzeugen zu vereinigen. Vereinigt man jedoch Kandidaten von Werkzeugen mit niedriger Precision so ist auch dieses Vorgehen kontraproduktiv. Vereinigt man aber die Kandidaten von Werkzeugen mit hoher Precision, so erhält man mehr getroffene Referenzen mit möglichst niedriger Zunahme der FalsePositives.

## 7. Zusammenfassung

Im Experiment könnte man die Kandidaten von Baxter und Merlo vereinigen. Die Hälfte von Baxters getroffenen Referenzen wird auch von Merlo entdeckt und 37 % von den Referenzen, die Merlo findet, werden auch von Baxter gefunden. Eine Vereinigung der Kandidaten würde also die Anzahl der gefundenen Referenzen deutlich erhöhen.

Wenn auch kein eindeutiger „Sieger“ des Experiments ausgemacht werden kann, so können doch Unterschiede und Gemeinsamkeiten zwischen den verschiedenen Ansätzen aufgezeigt werden. *Die* optimale Lösung gibt es nicht. Je nach Ziel ist das eine oder das andere Werkzeug vorzuziehen. Die Ergebnisse können den einzelnen Teilnehmern auch dazu dienen, aufgezeigte Schwächen zu beseitigen und die Effizienz ihrer Werkzeuge zu verbessern.

Zum Abschluss dieser Arbeit sollen die Eigenschaften der Werkzeuge sowie einige der gewonnenen Erkenntnisse über sie in der Tabelle in Abbildung 7.1 einander gegenübergestellt werden.

	Baker	Baxter	Kamiya	Krinke	Merlo	Rieger
<b>Eigenschaften:</b>						
Erkennung der Klontypen	1, 2	1, 2	1, 2, 3	3	1, 2, 3	1, 2, 3
Kategorisierung der Klontypen	1, 2	1, 2	–	3	1, 2, 3	–
Zeilen-/Tokenbasiert	×	–	×	–	–	×
Graphenbasiert	–	×	–	×	–	–
Metrikbasiert	–	–	–	–	×	–
C analysierbar	×	×	×	×	×	×
Java analysierbar	×	×	×	–	×	×
Multiprozessor-Unterstützung	–	×	–	–	–	–
Source erhältlich	–	–	–	–	–	×
<b>Erkenntnisse:</b>						
Konfigurierbarkeit	–	+	+	–	+	?
Robustheit gegenüber Eingabe	+	–	+	+	+	+
Skalierbarkeit	+	+	+	–	+	–
Geschwindigkeit	++	–	+	--	++	?
Speicherverbrauch	+	–	+	+	++	?
Hoher Recall	+	–	+	–	–	+
Hohe Precision	–	+	–	–	+	–
Überlappungsfreie Kandidaten	–	++	--	++	+	–
OnlyPairs	200	109	4	104	25	83
OnlyButOnePairs	10	37	0	8	0	25
FoundSecrets	42 %	28 %	46 %	4 %	24 %	31 %
Rejected	61 %	24 %	71 %	77 %	32 %	51 %
Korrekte Klonkategorisierung	93 %	77 %	–	72 %	86 %	–

Abbildung 7.1: Übersicht der Werkzeuge der Teilnehmer  
(Prozentangaben beziehen sich auf 2 %- und Good-Auswertung)

# A. Revalidieren des Experiments

## A.1. Systemvoraussetzungen

Für das Experiment werden die folgenden Systeme bzw. Programme benötigt. Es ist möglich, dass teilweise auch alternative Komponenten verwendet werden können, jedoch wird dann mit Sicherheit die eine oder andere Änderung nötig sein.

- Klonverwaltung, Bewertung von Kandidaten und Analyse
  - GNU Compiler Collection 2.95.4
  - GNU make 3.79.1
  - C++ STL 4.5.3
  - Qt3 Bibliothek 3.0.4
  - PostgreSQL 7.2.1
  
- Auswertung und Dokumentation
  - tetex 1.0.7
  - GlossTeX 0.4
  - FoilTeX 2.1.3
  - bibtopic 1.0j
  - KOMA-Skript 2.81
  - texdepend 0.92
  - play 1999-03-26
  - sed 3.02
  - Dia 0.90
  - png2eps 2002-05-25
  - pngcheck 1.99.3
  - xsltproc 1.0.18
  - ploticus 2.04

## A.2. Installation

Generell sind die in Abschnitt A.1 auf der vorherigen Seite genannten Programme entsprechend ihrer Dokumentation zu installieren. Falls weitere Aktionen nötig sind, werden sie im Folgenden erläutert:

- PostgreSQL  
Es muss mit `createuser -d -A clones` ein neuer Benutzer angelegt werden, unter welchem die Datenbank alle Daten für das Experiment ablegen kann. Des Weiteren müssen eventuell in der Konfigurationsdatei `pg_hba.conf` Rechte entsprechend angepasst werden.
- Qt3  
Hier ist darauf zu achten, dass Qt3 mitsamt dem Datenbank-Treiber für PostgreSQL-Anbindung (QPSQL7-Treiber) installiert wird (siehe dazu auch [8]). Entweder ist dies in der Bibliothek eingebunden oder über einen Plug-In-Mechanismus gelöst. Wird dies vergessen, erscheint beim Starten von `clones` eine entsprechende Fehlermeldung.
- tetex  
Zum Übersetzen der Dokumentation reicht eventuell der in der Konfigurationsdatei `texmf.cnf` reservierte Speicher nicht aus. Kommt eine derartige Fehlermeldung, so müssen die Werte `hash_extra` und/oder `pool_size` entsprechend vergrößert werden.
- texdepend  
`texdepend` erkennt in der ausgelieferten Version nicht alle Dateinamen im Experiment, da weder `-` noch `\` in Dateinamen erkannt werden. Letzteres ist nötig, um einen Bug im  $\text{\LaTeX}$ -Paket `graphicx` zu umgehen, der es unmöglich macht, Dateien mit mehreren Punkten im Dateinamen, aber ohne Angabe der Extension einzubinden. Aus diesem Grund muss in `texdepend` die Zeichenfolge `[/\w\d.]` in den Zeilen 211 und 227 zu `[/\w\d.\-\\]` erweitert werden (weitere Informationen zu  $\text{\LaTeX}$  siehe auch [39]).

## B. Implementierte Hilfsprogramme

Alle im Folgenden vorgestellten Werkzeuge zur Unterstützung der Aufbereitung der Systeme, zur Erfassung und Konvertierung der Daten sowie zur Auswertung der Ergebnisse sind unter der GNU General Public License (siehe [4] und [1]) veröffentlicht und unter [19] erhältlich.

### B.1. Programme zur Aufbereitung der Quelldateien der Projekte

Die Projekte des Experiments wurden zuerst mit dem Programm `textclean` und anschließend mit dem Programm `codenormalize` bearbeitet, so dass Quelltext, der den Werkzeugen der Teilnehmer vermutlich Probleme bereitet hätte, weitestgehend entfernt wurde oder einzelne illegale Zeichen ersetzt wurden.

#### B.1.1. `textclean`

##### NAME

`textclean` — Bereinigt Eingabe von nicht-ASCII-Werten

##### SYNTAX

`textclean`

##### BESCHREIBUNG

Das Programm `textclean` ist ein Filter, der Daten zeichenweise von `stdin` liest und sie wieder auf `stdout` schreibt. Dabei werden die einzelnen Zeichen unverändert ausgegeben, mit Ausnahme folgender ASCII-Werte:

Eingabe	Ausgabe
1 – 7	' ? '
8	"
9	' ' (variable Anzahl)
10,11,12	' \n '
13	"
14 – 31	' ? '
127 – 255	' ? '

Terminiert die Eingabe nicht mit `' \n '`, so wird dies am Ende der Eingabe noch hinzugefügt.

## B. Implementierte Hilfsprogramme

### B.1.2. codenormalize

#### NAME

codenormalize — Normiert C- und Java-Quelldateien

#### SYNTAX

```
codenormalize [INFILE [OUTFILE]]
```

#### BESCHREIBUNG

Das Programm `codenormalize` ist ein Filter, der Daten zeilenweise von `stdin` oder `INFILE` nach `stdout` oder `OUTFILE` schreibt. Steht „-“ für `INFILE`, so wird ebenfalls von `stdin` gelesen, steht „-“ für `OUTFILE`, so wird ebenfalls auf `stdout` geschrieben.

Alle Zeilen mit Präprozessor-Direktiven, die nicht in folgender Auflistung vorkommen, werden nicht kopiert:

- `#define`
- `#elif`
- `#else`
- `#endif`
- `#error`
- `#if`
- `#ifdef`
- `#ifndef`
- `#include`
- `#pragma`
- `#undef`

Treten `{` oder `}` alleine in einer Zeile auf (Leerzeichen sind ebenfalls erlaubt), so werden sie an die vorige Zeile angehängt. Dabei werden Kommentare `/*, */` und `//` berücksichtigt und die Klammern werden an die syntaktisch korrekte Stelle platziert. Ebenfalls berücksichtigt werden mehrzeilige Makrodefinitionen und mehrzeilige Zeichenketten.

Leerzeilen werden entfernt.

Alle übrigen Zeilen werden unverändert kopiert.

#### FEHLER

Es ist davon auszugehen, dass Fälle konstruierbar sind, in denen `codenormalize` fehlerhaft arbeitet und entweder syntaktisch falschen Code erzeugt oder sich semantische Änderungen ergeben. Während der Tests an allen Projekten im Experiment ist jedoch kein solcher Fall aufgetreten.

## B.2. Programme zur Auswertung und Evaluierung

Abgaben im Klonklassen-Format müssen zuerst in das Klonpaar-Format überführt werden. Dies geschieht mittels des Programms `cconvert`. Daraufhin können sie vom Programm `clones` importiert werden. `clones` ist das Hauptprogramm, welches den Import der Klon-daten in die Datenbank, die Erzeugung der Referenzmenge und die Auswertung der Daten vornimmt.

### B.2.1. `cconvert`

#### NAME

`cconvert` — Konvertiert vom Klonklassen-Format ins Klonpaar-Format

#### SYNTAX

```
cconvert [INFILE [OUTFILE]]
```

#### BESCHREIBUNG

Das Programm `cconvert` ist ein Filter, der Daten zeilenweise von `stdin` oder `INFILE` nach `stdout` oder `OUTFILE` schreibt. Steht „-“ für `INFILE`, so wird ebenfalls von `stdin` gelesen, steht „-“ für `OUTFILE`, so wird ebenfalls auf `stdout` geschrieben.

Mit `cconvert` können Dateien im Format für Klonklassen in das Format für Klonpaare umgewandelt werden. Fehlerhafte Zeilen werden ignoriert und es wird eine Fehlermeldung ausgegeben. Einelementige Klonklassen und Klonklassen, die Codefragmente mehrerer Systeme beinhalten, werden ebenfalls ignoriert und es wird eine Fehlermeldung ausgegeben.

#### FEHLER

Ist eine fehlerhafte Zeile länger als 1000 Zeichen, so werden für diese Zeile mehrere Fehlermeldungen ausgegeben, es wird dennoch mit der nächsten korrekten Eingabezeile fortgefahren.

## B. Implementierte Hilfsprogramme

### B.2.2. clones

#### NAME

clones — Verwaltet das Experiment

#### SYNTAX

clones

#### BESCHREIBUNG

Das Programm `clones` dient zur kompletten Verwaltung des Experiments. Es ist ein Frontend für die Datenbank, welche die Daten der verschiedenen Teilnehmer des Experiments verwaltet. Es benutzt die Qt-Bibliothek (siehe [8]) und ist daher in C++ geschrieben (siehe auch [49]). Folgende Menüpunkte stehen nach dem Start von `clones` zur Verfügung:

##### Projects/Tools (Projekte/Werkzeuge)

Die einzelnen Teilnehmer und Projekte des Experiments können in den Tabellen angelegt, verändert und gelöscht werden. Insbesondere können einzelne Projekte oder Werkzeuge aktiviert oder deaktiviert werden.

##### Import Candidates (Kandidaten-Import)

Die von den einzelnen Teilnehmern vorgeschlagenen Kandidaten können hier importiert werden. Dazu wählt man den entsprechenden Teilnehmer und das entsprechende Projekt in der Matrix aus, gibt die minimale Codefragmentgröße der Klonpaare an und kann dann die Datei im Klonpaar-Format zum Import auswählen. Während des Imports werden auf `stdout` Informationen und Statistiken zum Import ausgegeben. Die Matrix ist ebenfalls von Interesse, da sie den Anteil der bewerteten Kandidaten pro Teilnehmer und pro Projekt anzeigt. Sie wird nach jeder Orakel-Bewertung und beim Programmstart aktualisiert.

##### Import References (Referenzen-Import)

Referenzen können ebenfalls im Klonpaar-Format importiert werden. Dies geschieht nach Auswahl einer zu importierenden Datei. Hierbei werden alle Referenzen importiert; eine minimale Codefragmentgröße gibt es nicht. Während des Imports werden auf `stdout` Informationen und Statistiken zum Import ausgegeben.

##### View Clones (Klone anzeigen)

Sowohl Kandidaten als auch Referenzen können hier visuell betrachtet werden. Mit je einem SQL-Filter kann man die Auswahl näher eingrenzen. Das Klonpaar wird nach Auswahl eines Datensatzes in einem eigenen Fenster angezeigt und farblich hervorgehoben. Dabei hat das Codefragment in der linken Bildschirmhälfte die Farbe rot, während das Codefragment in der rechten Hälfte blau angezeigt wird. Beliebig viele Klonpaare können gleichzeitig geöffnet sein.

### Oracle Candidates (Kandidaten-Orakel)

Die von den Teilnehmern vorgeschlagenen Kandidaten werden dem Schiedsrichter zufällig präsentiert. Dabei wird in Betracht gezogen, dass der Anteil der bewerteten Klonpaare pro Teilnehmer und pro Projekt sich immer im gleichen Maße erhöht (d. h. es kann nicht passieren, dass zufällig nur Kandidaten eines Teilnehmers oder von einem Projekt bewertet werden). Der Schiedsrichter kann die Codefragmente durch Selektion mit der Maus ändern und diesen eventuell veränderten Kandidaten in die Referenzmenge übernehmen oder ignorieren. Dabei muss auch der Klontyp von 1 bis 3 festgelegt werden. Sind in der näheren Umgebung des Kandidaten bereits Referenzen vorhanden, werden diese mit anderen Farben angezeigt. Bei den Referenzen haben beide Codefragmente immer dieselbe Farbe, beim aktuellen Kandidaten sind die Codefragmente rot bzw. blau. Damit Überlappungen erkannt werden können, lassen sich die Referenzen und der aktuelle Kandidat durchrollen.

### Map Candidates (Kandidaten zuordnen)

Als Vorstufe zur eigentlichen Auswertung müssen den Kandidaten Referenzen und deren Ähnlichkeit zugeordnet werden. Hierbei werden zwei Maßstäbe benutzt: Das Good-Kriterium und das OK-Kriterium. Beim OK-Kriterium muss die Referenz zu einem bestimmten Anteil im Kandidat enthalten sein oder umgekehrt. Beim Good-Kriterium muss der sich überlappende Teil von Referenz und vom Kandidaten zum vereinigten Teil der beiden in einem bestimmten Anteil stehen. Dieser Anteil  $p$  für das Good- und OK-Kriterium lässt sich angeben. Zusätzlich kann der Benutzer auswählen, ob die eventuell bereits vorhandenen Mapping-Werte in der Datenbank zuerst gelöscht werden sollen. Dies ist notwendig, wenn das letzte Mapping mit einem anderen Wert für  $p$  erstellt wurde oder wenn die Tabellen der Datenbank von Hand editiert worden sind. Das Löschen dieser Mapping-Werte kann erhebliche Zeit in Anspruch nehmen.

### Evaluate (Auswertung)

Die zuvor den Referenzen zugeordneten Kandidaten werden hier ausgewertet. Die Auswertung besteht aus vielen einzelnen Faktoren und Kriterien. Die Auswertung wird auf `stdout` ausgegeben. Daher sollte vor der Auswertung eventuell `stdout` in eine Datei umgeleitet werden. Die Ausgabe ist im XML-Format, so dass die weitere Verarbeitung (mit XSL-Transformationen) erleichtert wird. Das Format wird in Abschnitt C.3.1 auf Seite 137 genauer erläutert. Auch hier muss wieder der Anteil  $p$  für das Good- und OK-Kriterium angegeben werden.

### Vacuum/Reindex (Datenbank aufräumen)

Da die verwendete Version der PostgreSQL-Datenbank einen Fehler aufweist, der dazu führt, dass Berechnungen mit der Zeit immer langsamer werden, sollte die Datenbank ab und zu mit dieser Funktion bereinigt werden. Dies hat lediglich Auswirkungen auf die Geschwindigkeit, die Ergebnisse bleiben unverändert.

## *B. Implementierte Hilfsprogramme*

### Quit (Beenden)

Das Frontend wird sofort verlassen, ohne weitere Aktionen an der Datenbank durchzuführen.

### DATEIEN

Es muss eine funktionierende Installation einer PostgreSQL-Datenbank mit mindestens der Version 7 vorhanden sein. Des Weiteren muss ein Benutzer `clones` mit dem Passwort `clones` mit Zugriff auf dem lokalen Rechner angelegt sein.

### FEHLER

In seltenen Fällen werden die Codefragmente der Klonpaare beim Anzeigen nicht richtig markiert bzw. alte Markierungen nicht entfernt. Dies ist höchstwahrscheinlich auf einen Fehler der verwendeten Bibliothek Qt 3 zurückzuführen. Eventuell behebt Linken gegen eine neuere Version der Bibliothek Qt 3 dies automatisch.

Des Weiteren werden bei Eingabefeldern keine Überprüfungen durchgeführt. Illegale Eingaben können somit zu unerwarteten Effekten führen.

## C. Dateiformate

### C.1. Klonpaar-Format

#### C.1.1. BNF

Das Format für Klonpaare ist durch folgende BNF (Backus Naur Form) bindend vorgegeben. Es handelt sich hierbei um ein reines ASCII-Format.

```
lineend    := '\n'
separator  := '\t'
digit      := '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
digit0     := '0' | digit
number     := digit [digit0*]
filename1  := valid-filename
fromline1  := number
toline1    := number
filename2  := valid-filename
fromline2  := number
toline2    := number
clonetype  := '0' | '1' | '2' | '3'
line       := filename1 separator fromline1 separator toline1
              filename2 separator fromline2 separator toline2
              separator clonetype lineend
file       := line [line*]
```

#### C.1.2. Beispiel

prj/src/foo.c	12	56	prj/src/bar.c	68	90	1
prj/src/wom.c	34	50	prj/src/bat.c	90	124	2
prj/src/wom.c	69	100	prj/src/bar.c	45	70	1
prj/src/wom.c	69	100	prj/src/foo.c	59	80	1
prj/src/bar.c	45	70	prj/src/foo.c	59	80	1

## C.2. Klonklassen-Format

### C.2.1. BNF

Das Format für Klonklassen ist durch die folgende BNF definiert. Hierbei handelt es sich genau genommen um kein reines ASCII-Format im engeren Sinne, da es das §-Zeichen benutzt, welches in den ASCII-Zeichen nicht definiert ist.

```
lineend    := '\n'
separator  := '\t'
marker     := '§'
digit      := '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
digit0     := '0' | digit
number     := digit [digit0*]
filename   := valid-filename
fromline   := number
toline     := number
clonetype  := '0' | '1' | '2' | '3'
line       := filename separator fromline separator toline lineend
clonelist  := marker separator clonetype newline line line*
file       := clonelist [clonelist*] marker newline
```

### C.2.2. Beispiel

```
§      1
prj/src/foo.c  12      56
prj/src/bar.c  68      90
§      2
prj/src/wom.c  34      50
prj/src/bat.c  90     124
§      1
prj/src/wom.c  69     100
prj/src/bar.c  45      70
prj/src/foo.c  59      80
§
```

## C.3. Auswertungsdaten

### C.3.1. DTD für die Auswertung im XML-Format

Das Datenformat, welches clones (siehe Abschnitt B.2.2 auf Seite 132) bei einer Auswertung erzeugt, ist XML (EXtensible Markup Language). Hier wird nun eine DTD (Document Type Definition) für das ausgegebene XML aufgelistet.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!-- clones.dtd - Document Type Definition of clones XML output -->

<!ELEMENT Evaluation (Project+)>
<!ATTLIST Evaluation Percentage CDATA #REQUIRED>

<!ELEMENT Project (Language,
                   Size,
                   Type+)>
<!ATTLIST Project Name CDATA #REQUIRED>

<!ELEMENT Type (ReferencePairs,
                AllCandidatePairs,
                AllCandidateSeenPairs,
                AllOverlappingCandidates,
                MaxReferenceSize,
                AvgReferenceSize,
                StdDevReferenceSize,
                VarianceReferenceSize,
                InnerFileReferences,
                AcrossFileReferences,
                ProjectCriterion+,
                Tool+)>
<!ATTLIST Type Name CDATA #REQUIRED>

<!ELEMENT ProjectCriterion (AllRejectedPositiveCandidates,
                            AllTrueNegativeReferences,
                            ReferencesFoundByNTools+,
                            ReferenceIntersectionMatrix,
                            ReferenceDifferenceMatrix,
                            RejectedIntersectionMatrix,
                            RejectedDifferenceMatrix)>
<!ATTLIST ProjectCriterion Name (OK|Good) #REQUIRED>

<!ELEMENT Tool (CandidatePairs,
                CandidateSeenPairs,
                MaxCandidateSize,
```

### C. Dateiformate

```
        AvgCandidateSize,  
        StdDevCandidateSize,  
        VarianceCandidateSize,  
        InnerFileCandidates,  
        AcrossFileCandidates,  
        OverlappingCandidates,  
        ToolCriterion+)>  
<!ATTLIST Tool Name CDATA #REQUIRED>  
  
<!ELEMENT ToolCriterion (FoundReferences,  
        FoundReferencesWithCorrectType,  
        FoundReferencesWithHigherType,  
        FoundReferencesWithLowerType,  
        Recall,  
        Precision,  
        OnlyPairs,  
        OnlyButOnePairs,  
        FalsePositiveCandidates,  
        RejectedPositiveCandidates,  
        TrueNegativeReferences,  
        MaxCandidateSize,  
        AvgCandidateSize,  
        StdDevCandidateSize,  
        VarianceCandidateSize,  
        InnerFileCandidates,  
        AcrossFileCandidates,  
        ReferenceOverlapDistribution,  
        ReferenceClassSizeDistribution)>  
<!ATTLIST ToolCriterion Name (OK|Good) #REQUIRED>  
  
<!ELEMENT Language (#PCDATA)>  
<!ELEMENT Size (#PCDATA)>  
<!ELEMENT ReferencePairs (#PCDATA)>  
<!ELEMENT AllCandidatePairs (#PCDATA)>  
<!ELEMENT AllCandidateSeenPairs (#PCDATA)>  
<!ELEMENT AllRejectedPositiveCandidates (#PCDATA)>  
<!ELEMENT AllTrueNegativeReferences (#PCDATA)>  
<!ELEMENT AllOverlappingCandidates (#PCDATA)>  
<!ELEMENT MaxReferenceSize (#PCDATA)>  
<!ELEMENT AvgReferenceSize (#PCDATA)>  
<!ELEMENT StdDevReferenceSize (#PCDATA)>  
<!ELEMENT VarianceReferenceSize (#PCDATA)>  
<!ELEMENT InnerFileReferences (#PCDATA)>  
<!ELEMENT AcrossFileReferences (#PCDATA)>
```

```

<!ELEMENT ReferencesFoundByNTools (#PCDATA)>
<!ATTLIST ReferencesFoundByNTools N CDATA #REQUIRED>
<!ELEMENT ReferenceIntersectionMatrix (#PCDATA)>
<!ELEMENT ReferenceDifferenceMatrix (#PCDATA)>
<!ELEMENT RejectedIntersectionMatrix (#PCDATA)>
<!ELEMENT RejectedDifferenceMatrix (#PCDATA)>
<!ELEMENT CandidatePairs (#PCDATA)>
<!ELEMENT CandidateSeenPairs (#PCDATA)>
<!ELEMENT MaxCandidateSize (#PCDATA)>
<!ELEMENT AvgCandidateSize (#PCDATA)>
<!ELEMENT StdDevCandidateSize (#PCDATA)>
<!ELEMENT VarianceCandidateSize (#PCDATA)>
<!ELEMENT InnerFileCandidates (#PCDATA)>
<!ELEMENT AcrossFileCandidates (#PCDATA)>
<!ELEMENT FoundReferences (#PCDATA)>
<!ELEMENT FoundReferencesWithCorrectType (#PCDATA)>
<!ELEMENT FoundReferencesWithHigherType (#PCDATA)>
<!ELEMENT FoundReferencesWithLowerType (#PCDATA)>
<!ELEMENT Recall (#PCDATA)>
<!ELEMENT Precision (#PCDATA)>
<!ELEMENT OverlappingCandidates (#PCDATA)>
<!ELEMENT OnlyPairs (#PCDATA)>
<!ELEMENT OnlyButOnePairs (#PCDATA)>
<!ELEMENT FalsePositiveCandidates (#PCDATA)>
<!ELEMENT RejectedPositiveCandidates (#PCDATA)>
<!ELEMENT TrueNegativeReferences (#PCDATA)>
<!ELEMENT ReferenceOverlapDistribution (OverlapCount*)>
<!ELEMENT OverlapCount (#PCDATA)>
<!ATTLIST OverlapCount Multiplicity CDATA #REQUIRED>
<!ELEMENT ReferenceClassSizeDistribution (ClassSizeCount*)>
<!ELEMENT ClassSizeCount (#PCDATA)>
<!ATTLIST ClassSizeCount ClonePairs CDATA #REQUIRED>

```

### C.3.2. Definitionen und Erklärungen

Die einzelnen XML-Elemente im vorigen Abschnitt sollen nun kurz erklärt und, wenn nötig, genau definiert werden.

Befinden sich Elemente innerhalb von anderen Elementen, entsprechend der DTD von Abschnitt C.3.1 auf Seite 137, so gelten diese nur für den aktuellen Bereich, auch Kontext genannt. Konkret heißt dies, dass z. B. alle Elemente innerhalb eines `<Type Name="2">` Elements sich nur auf Typ-2-Klone beziehen, alle Elemente innerhalb eines Elementes `<Tool Name="Baker">` beziehen sich nur auf Daten von Baker, alle Elemente innerhalb eines `<ToolCriterion Name="OK">` beziehen sich nur auf die OK-Match(p)-Auswertung etc. Dieser Kontext gilt für alle nun folgenden Elemente:

### C. Dateiformate

#### Language

Programmiersprache des Projektes

#### Size

Größe des Projektes in 1K SLOC

#### ReferencePairs

Anzahl aller Referenzen

#### AllCandidatePairs

Anzahl aller Kandidaten (summiert über Tochter-Elemente hinweg)

#### AllCandidateSeenPairs

Anzahl aller vom Schiedsrichter bewerteten Kandidaten (summiert über Tochter-Elemente hinweg)

#### AllRejectedPositiveCandidates

Anzahl aller vom Schiedsrichter bewerteten Kandidaten, die auf keine Referenz treffen (summiert über Tochter-Elemente hinweg)

#### AllTrueNegativeReferences

Anzahl aller Referenzen, die von keinem Kandidaten gemäß  $OK\text{-Match}(p)$  überdeckt werden (summiert über Tochter-Elemente hinweg)

#### AllOverlappingCandidates

Anzahl aller Kandidaten, bei denen sich das eine Codefragment mit dem anderen überlappt (summiert über Tochter-Elemente hinweg)

#### MaxReferenceSize

Maximale Klongröße der Referenzen (dabei ist die Klongröße eines Klons das größere der zwei Codefragmente)

#### AvgReferenceSize

Durchschnittliche Klongröße der Referenzen (dabei ist die Klongröße eines Klons das arithmetische Mittel der zwei Codefragmente)

#### StdDevReferenceSize

Standardabweichung der Klongrößen der Referenzen (dabei ist die Klongröße eines Klons das arithmetische Mittel der zwei Codefragmente)

#### VarianceReferenceSize

Varianz der Klongrößen der Referenzen (dabei ist die Klongröße eines Klons das arithmetische Mittel der zwei Codefragmente)

#### InnerFileReferences

Anzahl der Referenzen, bei denen beide Codefragmente innerhalb derselben Datei liegen

AcrossFileReferences

Anzahl der Referenzen, bei denen die Codefragmente in anderen Dateien liegen

ReferencesFoundByNTools

Anzahl aller Referenzen, die  $N$  Werkzeuge gemeinsam finden (es werden nur die Pflicht-Abgaben berücksichtigt)

ReferenceIntersectionMatrix

Matrix der Referenzen, welche die Werkzeuge gemeinsam finden (ASCII-Matrix)

ReferenceDifferenceMatrix

Matrix der Referenzen, die ein Werkzeug findet, ein anderes nicht (ASCII-Matrix)

RejectedIntersectionMatrix

Matrix der Kandidaten, die auf keine Referenz treffen, aber von zwei Werkzeugen benannt werden (ASCII-Matrix)

RejectedDifferenceMatrix

Matrix der Kandidaten, die auf keine Referenz treffen und von einem Werkzeug benannt werden, einem anderen aber nicht (ASCII-Matrix)

CandidatePairs

Anzahl aller Kandidaten

CandidateSeenPairs

Anzahl aller vom Schiedsrichter bewerteten Kandidaten

MaxCandidateSize

Maximale Klongröße der Kandidaten (dabei ist die Klongröße eines Klons das größere der zwei Codefragmente)

AvgCandidateSize

Durchschnittliche Klongröße der Kandidaten (dabei ist die Klongröße eines Klons das arithmetische Mittel der zwei Codefragmente)

StdDevCandidateSize

Standardabweichung der Klongrößen der Kandidaten (dabei ist die Klongröße eines Klons das arithmetische Mittel der zwei Codefragmente)

VarianceCandidateSize

Varianz der Klongrößen der Kandidaten (dabei ist die Klongröße eines Klons das arithmetische Mittel der zwei Codefragmente)

InnerFileCandidates

Anzahl der Kandidaten, bei denen beide Codefragmente innerhalb derselben Datei liegen

### C. Dateiformate

#### AcrossFileCandidates

Anzahl der Kandidaten, bei denen die Codefragmente in anderen Dateien liegen

#### FoundReferences

Anzahl der gefundenen Referenzen (entsprechend OK-Match(p) bzw. Good-Match(p))

#### FoundReferencesWithCorrectType

Anzahl der gefundenen Referenzen (gemäß OK-Match(p) bzw. Good-Match(p)), bei denen das Werkzeug des Kontexts den richtigen Klontyp zuordnet

#### FoundReferencesWithHigherType

Anzahl der gefundenen Referenzen (gemäß OK-Match(p) bzw. Good-Match(p)), bei denen das Werkzeug des Kontexts einen zu hohen Klontyp zuordnet

#### FoundReferencesWithLowerType

Anzahl der gefundenen Referenzen (gemäß OK-Match(p) bzw. Good-Match(p)), bei denen das Werkzeug des Kontexts einen zu niedrigen Klontyp zuordnet

#### Recall

Wert für Recall, wobei FoundReferences und ReferencePairs aus demselben Kontext verwendet werden

#### Precision

Wert für Precision, wobei FoundReferences und CandidatePairs aus demselben Kontext verwendet werden

#### OverlappingCandidates

Anzahl der Kandidaten, bei denen sich das eine Codefragment mit dem anderen überlappt

#### OnlyPairs

Anzahl der Referenzen, die nur das Werkzeug des Kontexts findet

#### OnlyButOnePairs

Anzahl der Referenzen, die alle Werkzeuge bis auf das Werkzeug des Kontexts finden

#### FalsePositiveCandidates

Anzahl der Kandidaten, die auf keine Referenz treffen (gemäß OK-Match(p) bzw. Good-Match(p))

#### RejectedPositiveCandidates

Anzahl der vom Schiedsrichter bewerteten Kandidaten, die auf keine Referenz treffen

#### TrueNegativeReferences

Anzahl der Referenzen, die von keinem Kandidaten gemäß OK-Match(p) überdeckt werden

ReferenceOverlapDistribution

Anzahl und Verteilung von mehrfach getroffenen Referenzen

ReferenceClassSizeDistribution

Größe und Häufigkeit von eine Klonklasse bildenden Klonpaaren, die als Referenzen getroffen werden

### *C. Dateiformate*

## D. Kurzbeschreibung der analysierten Projekte

### D.1. Test-Phase

#### D.1.1. bison

bison ist ein in der Programmiersprache C geschriebener Parser-Generator, der kontextfreie Grammatiken in ein C-Programm umwandelt, welches dann die durch die Grammatik angegebene Sprache parsen kann. Es ist unter <http://www.gnu.org/software/bison/> erhältlich. Die im Experiment verwendete Version 1.32 besteht aus 19K SLOC. Das Projekt wurde hauptsächlich von drei Autoren geschrieben, wobei zahllose weitere Personen Verbesserungsvorschläge gemacht haben, die eingearbeitet worden sind.

#### D.1.2. wget

wget ist ein Werkzeug, mit dem Daten über Netzwerke anhand ihres URL (Uniform Resource Locator) heruntergeladen werden können. Es ist in C implementiert, umfasst 16K SLOC und ist unter <http://www.gnu.org/software/wget/> erhältlich. Im Experiment wurde Version 1.5.3 verwendet, die im Großen und Ganzen von vier Autoren geschrieben wurde. Jedoch sind viele Beiträge auch von anderen Entwicklern im Laufe der Zeit gemacht worden.

#### D.1.3. EIRC

EIRC (*Eteria IRC Client*) ist ein in der Programmiersprache Java implementierter IRC-Client. Er ist nur 8K SLOC groß, und im Experiment wurde die von zwei Autoren entwickelte Version 1.0.1 verwendet. Weitere Informationen sind unter <http://sourceforge.net/projects/eirc/> zu finden.

#### D.1.4. spule

spule (*Secure Practical Universal Lecture Evaluator*) ist ein in der Programmiersprache Java implementiertes Client-Server-System zur Durchführung sicherer Vorlesungsumfragen. Es wurde von zwei Autoren entwickelt, und die im Experiment verwendete Version 1.0.0 ist 10K SLOC groß und unter <http://sourceforge.net/projects/spule/> erhältlich.

## D.2. Haupt-Phase

### D.2.1. weltab

weltab ist ein „Election Tabulation System“, das ursprünglich von der Firma Radius Systems Inc. entwickelt wurde und mittlerweile oft zu Software-Reengineering- und Analyse-Zwecken verwendet wird. Es ist 11K SLOC groß und in C geschrieben.

### D.2.2. cook

cook ist ein mächtigerer Ersatz für das traditionelle Werkzeug make. Das System ist von einer Person entwickelt und geschrieben worden. Es umfasst 80K SLOC C-Code und kann von der Seite des Autors unter <http://www.canb.aaug.org.au/~millerp/cook/cook.html> heruntergeladen werden. Im Experiment wurde die Version 2.19 verwendet.

### D.2.3. snns

snns (*Stuttgart Neural Network Simulator*) ist ein in der Programmiersprache C geschriebener Software-Simulator für neuronale Netze, der ursprünglich an der Universität Stuttgart entwickelt wurde, nun aber von der Universität Tübingen weiterentwickelt wird. Das System ist 115K SLOC groß. Informationen über die im Experiment verwendete Version 4.2 sind unter <http://www-ra.informatik.uni-tuebingen.de/SNNS/> zu finden.

### D.2.4. postgresql

postgresql ist eine in C geschriebene freie Implementierung eines RDBMS auf SQL-Basis. Der Schwerpunkt ist große Kompatibilität mit dem SQL-Standard, weniger die Geschwindigkeit. Im Experiment wurde zu Analysezwecken die Version 7.2 verwendet. Diese umfasst 235K SLOC und ist unter <http://www.postgresql.org/> verfügbar. PostgreSQL wird von einem Entwickler-Team von 20 Haupt-Entwicklern und weiteren 50 Personen, die gelegentlich Code beisteuern, entwickelt.

### D.2.5. netbeans-javadoc

netbeans-javadoc ist das Sub-System der Entwicklungsumgebung NetBeans, welcher Javadoc unterstützt. Es ist in Java implementiert, und die Größe dieses Moduls beläuft sich auf 19K SLOC. Insgesamt entwickeln neun Personen an diesem Sub-System, über das man unter <http://javadoc.netbeans.org/> weitere Informationen erhalten kann. Im Experiment ist die Version des Releases 331 verwendet worden.

### D.2.6. eclipse-ant

eclipse-ant ist der Teil der erweiterbaren, universellen Entwicklungsumgebung Eclipse, der das Werkzeug ant zur Generierung unterstützt. Die Größe dieses Sub-Systems umfasst 35K SLOC und ist in Java geschrieben. Nähere Informationen zu Eclipse sind unter

<http://www.eclipse.org/> verfügbar. Für das Experiment wurde ein Code-Snapshot vom 15. Februar 2002 verwendet.

### **D.2.7. eclipse-jdtcore**

eclipse-jdtcore ist der Teil der in Java implementierten, erweiterbaren, universellen Entwicklungsumgebung Eclipse, der den 148K SLOC großen Kern des Plug-In-Systems, den Compiler, beinhaltet. Details sind unter <http://www.eclipse.org/jdt/> zu finden. Für das Experiment wurde ein Code-Snapshot vom 15. Februar 2002 verwendet.

### **D.2.8. j2sdk1.4.0-javax-swing**

j2sdk1.4.0-javax-swing ist der Teil der Java-Entwicklungsumgebung von Sun, welcher die Swing-Bibliotheken beinhaltet. Dieses Sub-System ist 204K SLOC groß und unter anderem von <http://java.sun.com/> erhältlich.

*D. Kurzbeschreibung der analysierten Projekte*

## Abkürzungsverzeichnis

- ANSI..... American National Standards Institute ist eine Organisation, welche Standardisierungen verschiedenster Art administriert und koordiniert.
- ASCII..... American Standard Code for Information Interchange ist das wohl weitgebräuchlichste Verfahren, Zeichen in 7 bit zu codieren.
- AST..... Abstract Syntax Tree bezeichnet den abstrakten Syntaxbaum, der beim Parsen eines Quellcodes im Parser aufgebaut wird.
- BNF..... Backus Naur Form ist eine Notation, um Grammatiken auszudrücken. Es handelt sich dabei um Produktionsregeln, bestehend aus Terminal- und Nichtterminalsymbolen.
- DTD..... Document Type Definition gibt die Syntax an, an die sich eine XML-Datei, welche diese Definition benutzt, zu richten hat.
- GNU..... GNU's Not Unix ist eine freie Re-Implementierung des Betriebssystems UNIX. <http://www.gnu.org/>
- HTML..... HyperText Markup Language ist eine Beschreibungssprache zur Präsentation von Text, Tabellen, Bildern und Querverweisen. Sie wird hauptsächlich im World Wide Web verwendet.
- IRC..... Internet Relay Chat bezeichnet man das Protokoll, um über das Internet in Echtzeit mit anderen Teilnehmern textbasiert zu kommunizieren.
- K&R..... Kernighan&Ritchie sind die Erfinder der Programmiersprache C, und nach ihnen ist der von ihnen erfundene C-Dialekt benannt. Mittlerweile gibt es einen davon abweichenden ANSI C Standard.
- PDG..... Program Dependence Graph steht für Abhängigkeitsgraph. In diesem Graphen werden die Abhängigkeiten einzelner Programmanweisungen und/oder Variablen untereinander abgebildet.
- RDBMS..... Relational DataBase Management System bezeichnet ein relationales Datenbank-System.
- SLOC..... Source Lines Of Code ist eine Maßeinheit der Programmgröße und gibt die Zahl der Quellcodezeilen an.

### *Abkürzungsverzeichnis*

- SQL..... Structured Query Language ist die standardisierte Sprache zur Abfrage von Daten eines RDBMS. <http://www.sql.org/>
- URL..... Uniform Resource Locator beschreibt Ort und Zugriffsprotokoll von Daten über das Internet.
- XML..... EXtensible Markup Language steht für „erweiterbare Beschreibungssprache“ und wurde entworfen, um Daten zu transportieren (siehe [2]).
- XSL..... EXtensible Stylesheet Language steht für „erweiterbare Formatvorlage“ und gibt Regeln an, nach welchen XML-Dateien transformiert werden (siehe [3, 10]).

# Glossar

**Codefragment** bezeichnet einen zusammenhängenden Teil Quellcode, der selektiert werden kann, um ihn z. B. an eine andere Stelle zu kopieren und dadurch zu klonen.

**Copy&Paste** bezeichnet das Kopieren eines Textbereiches von einem Ort an einen anderen.

**Kandidat** ist ein Klonpaar, das von einem Teilnehmer im Experiment eingesandt wurde.

**Kandidatenmenge** bezeichnet die Menge aller Kandidaten

**Klon** bezeichnet allgemein ein Stück duplizierten Quellcodes.

**Klonklasse** bezeichnet eine Menge von zwei oder mehr Codefragmenten, welche untereinander identisch oder ähnlich genug – gemäß den Klontypen – sind.

**Klonpaar** ist ein Paar von zwei Codefragmenten, welche identisch oder ähnlich genug – gemäß den Klontypen – sind.

**MySQL** ist eine freie Implementierung eines RDBMS auf SQL-Basis. Der Schwerpunkt ist große Geschwindigkeit. <http://www.mysql.com/>, siehe auch [6].

**Orakel** bezeichnet den Vorgang, bei dem zufällige Kandidaten manuell betrachtet und bewertet werden und eventuell modifiziert in die Menge der Referenzen aufgenommen werden oder nicht.

**PostgreSQL** ist eine freie Implementierung eines RDBMS auf SQL-Basis. Der Schwerpunkt ist große Kompatibilität mit dem SQL-Standard. <http://www.postgresql.org/>, siehe auch [7].

**Referenz** ist ein Klonpaar, welches manuell als korrektes und gültiges Klonpaar verifiziert wurde.

**Referenzmenge** bezeichnet die Menge aller Referenzen

**Schiedsrichter** bezeichnet die Person, die das Experiment durchführt und speziell die Bewertung der Kandidaten mittels des Orakels vornimmt.

## *Glossar*

## Verzeichnis elektronischer Quellen

- [1] *Deutsche Übersetzung der GNU General Public License*. <http://www.gnu.de/gpl-ger.html>. übersetzt von Katja Lachenmann Übersetzungen im Auftrag der S. u. S. E. GmbH.
- [2] *Extensible Markup Language (XML)*. <http://www.w3.org/XML/>.
- [3] *The Extensible Stylesheet Language (XSL)*. <http://www.w3.org/Style/XSL/>.
- [4] *GNU General Public License*. <http://www.gnu.org/licenses/gpl.html>.
- [5] *JavaCC Grammar Repository*. <http://www.cobase.cs.ucla.edu/pub/javacc>.
- [6] *MySQL Manual*. <http://www.mysql.com/doc/>.
- [7] *PostgreSQL Interactive Documentation*. <http://www.postgresql.org/idocs/>.
- [8] *Qt Reference Documentation*. <http://doc.trolltech.com/3.0/>.
- [9] *WebGain Products: JavaCC*. [http://www.webgain.com/products/java\\_cc](http://www.webgain.com/products/java_cc).
- [10] *XML Path Language (XPath)*. <http://www.w3.org/TR/xpath>.
- [11] BAKER, BRENDA S.: *clones for main experiment*. Persönliche E-Mail, Message-ID: <3CB0F060.4595C55E@research.bell-labs.com>.
- [12] BAKER, BRENDA S.: *Dup technical data*. Persönliche E-Mail, Message-ID: <3D2DC694.6125836C@research.bell-labs.com>.
- [13] BAKER, BRENDA S.: *Questions concerning tool capabilities*. Mailing-Listen-Artikel, Message-ID: <3C45E330.B9423E6F@research.bell-labs.com>.
- [14] BAKER, BRENDA S.: *Results from test experiments*. Mailing-Listen-Artikel, Message-ID: <3C68ABE7.ED85B60A@research.bell-labs.com>.
- [15] BAXTER, IRA D.: *First round results*. Mailing-Listen-Artikel, Message-ID: <D74778365DEFD01187A700A02448802F2B8A0E@PLUTO>.
- [16] BAXTER, IRA D.: *Questions concerning tool capabilities*. Mailing-Listen-Artikel, Message-ID: <D74778365DEFD01187A700A02448802F2A7252@PLUTO>.
- [17] BAXTER, IRA D.: *Starting shot for main round!* Persönliche E-Mail, Message-ID: <D74778365DEFD01187A700A02448802F2D88D6@PLUTO>.

## Verzeichnis elektronischer Quellen

- [18] BELLON, STEFAN: *Detection of Software Clones – Tool comparison experiment*. <http://www.informatik.uni-stuttgart.de/ifi/ps/clones/index.html>. Offizielle, eigens für das Experiment eingerichtete Homepage mit verbindlichen Informationen zur Durchführung.
- [19] BELLON, STEFAN: *Studium-Seite*. <http://www.sbellon.de/studium.html>. Persönliche Homepage des Autors mit während des Studiums erarbeitetem Material.
- [20] KAMIYA, TOSHIHIRO: *Intermediate results and further questions*. Mailing-Listen-Artikel, Message-ID: <005d01c1b19b\$d486b610\$38acdda3@is.aistnara.ac.jp>.
- [21] KAMIYA, TOSHIHIRO: *Questions concerning tool capabilities*. Mailing-Listen-Artikel, Message-ID: <005701c19e8f\$98afbb10\$38acdda3@is.aistnara.ac.jp>.
- [22] KAMIYA, TOSHIHIRO: *Starting shot for main round!* Persönliche E-Mail, Message-ID: <002b01c1cad1\$70d1d560\$38acdda3@EGOIST>.
- [23] KRINKE, JENS: *Duplix-Parameter*. Persönliche E-Mail, Message-ID: <3D3FF0A8.6000504@fmi.uni-passau.de>.
- [24] KRINKE, JENS: *Intermediate results and further questions*. Persönliche E-Mail, Message-ID: <3D04CEA3.30405@fmi.uni-passau.de>.
- [25] KRINKE, JENS: *Questions concerning tool capabilities*. Mailing-Listen-Artikel, Message-ID: <d8hvge2edfk.fsf@fmi.uni-passau.de>.
- [26] MERLO, ETTORE: *architecture info*. Persönliche E-Mail, Message-ID: <Pine.LNX.4.44.0207170909130.7384-100000@d5333-08>.
- [27] MERLO, ETTORE: *clones investigation*. Mailing-Listen-Artikel, Message-ID: <Pine.LNX.4.33.0201201846060.16478-100000@pascal>.
- [28] MERLO, ETTORE: *final run results*. Persönliche E-Mail, Message-ID: <Pine.LNX.4.44.0207051244300.24488-100000@d5333-08>.
- [29] MERLO, ETTORE: *final run results*. Persönliche E-Mail, Message-ID: <Pine.LNX.4.44.0207122020280.30555-100000@d5333-08>.
- [30] RIEGER, MATTHIAS: *Clone Contest Entry*. Persönliche E-Mail, Message-ID: <5.1.0.14.0.20020208192728.009f4eb0@mail.iam.unibe.ch>.
- [31] RIEGER, MATTHIAS: *Duploc Home*. <http://www.iam.unibe.ch/~rieger/duploc/>.

## Literaturverzeichnis

- [32] BAILEY, JOHN und ELIZABETH BURD: *Evaluating Clone Detection Tools for Use during Preventative Maintenance*. (to appear).
- [33] BAKER, BRENDA S.: *A program for identifying duplicated code*. In: *Computer Science and Statistics 24: Proceedings of the 24th Symposium on the Interface*, Seiten 49–57, März 1992.
- [34] BAKER, BRENDA S.: *On Finding Duplication and Near-Duplication in Large Software Systems*. In: WILLS, L., P. NEWCOMB und E. CHIKOFSKY (Herausgeber): *Second Working Conference on Reverse Engineering*, Seiten 86–95, Los Alamitos, California, Juli 1995. IEEE Computer Society Press.
- [35] BAKER, BRENDA S.: *Parameterized Pattern Matching: Algorithms and Applications*. *Journal Computer System Science*, 52(1):28–42, Februar 1996.
- [36] BAXTER, IRA D., ANDREW YAHIN, LEONARDO MOURA, MARCELO SANT’ANNA und LORRAINE BIER: *Clone Detection Using Abstract Syntax Trees*. In: *Proceedings; International Conference on Software Maintenance*, 1998.
- [37] DUCASSE, STÉPHANE, MATTHIAS RIEGER und SERGE DEMEYER: *A Language Independent Approach for Detecting Duplicated Code*. In: *Proceedings of the International Conference on Software Maintenance (ICSM99)*, 1999.
- [38] FERRANTE, J., K. OTTENSTEIN und J. WARREN: *The program dependence graph and its use in optimization*. *ACM Trans. on Prog. Lang. and Sys.*, 9(3):319–349, Juli 1987.
- [39] GOOSSENS, MICHEL, FRANK MITTELBACH und ALEXANDER SAMARIN: *Der  $\LaTeX$ -Begleiter*. Addison-Wesley, Bonn; Paris; Reading, Mass. [u. a.], 1994.
- [40] KAMIYA, TOSHIHIRO, SHINJI KUSUMOTO und KATSURO INOUE: *CCFinder: A Multi-Linguistic Token-based Code Clone Detection System for Large Scale Source Code*. *IEEE Trans. Software Engineering* (to appear).
- [41] KOMONDOOR, RAGHAVAN und SUSAN HORWITZ: *Tool Demonstration: Finding Duplicated Code Using Program Dependences*. *Lecture Notes in Computer Science*, 2028:383ff, 2001.

- [42] KONTOGIANNIS, K.: *Evaluation Experiments on the Detection of Programming Patterns Using Software Metrics*. In: *Proceedings of the 4th Working Conference on Reverse Engineering (WCRE '97)*, 1997.
- [43] KONTOGIANNIS, K., R. DEMORI, M. BERNSTEIN, M. GALLER und ETTORRE MERLO: *Pattern matching for design concept localization*. In: *WCRE '95: Proceedings of the Second Working Conference on Reverse Engineering, (Toronto, Ontario; July 14-16, 1995)*, Seiten 96–103. IEEE Computer Society Press (Order Number PR07111), Juli 1995.
- [44] KOSCHKE, RAINER: *Reengineering*. University of Stuttgart, 2000. Skriptum zur Vorlesung.
- [45] KOSCHKE, RAINER und THOMAS EISENBARTH: *A Framework for Experimental Evaluation of Clustering Techniques*. In: *International Workshop on Program Comprehension*, Seiten 201–210, Limerick, Ireland, Juni 2000. IEEE Computer Society Press.
- [46] KRINKE, JENS: *Identifying Similar Code with Program Dependence Graphs*. In: *Proceedings of the Eighth Working Conference On Reverse Engineering (WCRE'01)*, 2001.
- [47] LAGUÈ, BRUNO, DANIEL PROULX, JEAN MAYRAND, ETTORRE M. MERLO und JOHN HUDEPOHL: *Assessing the benefits of incorporating function clone detection in a development process*. In: *International Conference on Software Maintenance*, Seiten 314–321, 1997.
- [48] MAYRAND, JEAN, CLAUDE LEBLANC und ETTORRE M. MERLO: *Experiment on the Automatic Detection of Function Clones in a Software System using Metrics*. In: *Proceedings of the International Conference on Software Maintenance*, 1996.
- [49] STROUSTRUP, BJARNE: *The C++ Programming Language*. Addison-Wesley, 2000. Special edition, includes bibliographical references and index.

## **Erklärung**

Hiermit versichere ich, diese Arbeit selbstständig verfasst und nur die angegebenen Quellen benutzt zu haben.

---

(Stefan Bellon)