Karlsruhe Institute of Technology

Identifikation von Test-Klonen

Bachelorarbeit
von

Johann Böhler

An der Fakultät für Informatik
Institut für Programmstrukturen
und Datenorganisation (IPD)

Erstgutachter:	Prof. Dr. Walter F. Tichy
Zweitgutachter:	Prof. Dr. Ralf H. Reussner
Betreuender Mitarbeiter:	Dipl. Inform.-Wirt Mathias Landhäußer

Bearbeitungszeit: 01.12.2012 – 01.04.2013

Ich versichere wahrheitsgemäß, die Arbeit selbstständig angefertigt, alle benutzten Hilfsmittel vollständig und genau angegeben und alles kenntlich gemacht zu haben, was aus Arbeiten anderer unverändert oder mit Abänderungen entnommen wurde.

Karlsruhe, 28.03.2013

.....

(**Johann Böhler**)

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1. Einleitung

In dieser Arbeit werden Möglichkeiten aufgezeigt Test-Klone innerhalb von Quelltexten automatisiert identifizieren zu können. Als Test-Klone werden dabei idealerweise sowohl semantisch als auch syntaktisch ähnliche Testfälle bezeichnet.

Der automatisierte Ablauf, des hier vorgestellten Verfahrens, ermöglicht den Vergleich vieler Quelltexte ohne erhöhten manuellen Aufwand. Da die manuelle Untersuchung von Testfällen zum Zweck der Klonfindung sehr aufwendig und schnell unübersichtlich wird, kann das Verfahren hier Abhilfe schaffen. Die Ergebnisse der Untersuchung können vielfältig genutzt werden. Auf Grundlage der Test-Klone könnten beispielsweise weitere Testfälle erzeugt werden. Dazu muss untersucht werden, ob ein Zusammenhang zwischen der zu testenden Funktionalität und den geklonten Testfällen besteht. Im günstigsten Fall lässt sich ein kausaler Zusammenhang ableiten und Testfälle können, bei Erkennung eines bestimmten Musters, generiert werden. Weiterhin kann untersucht werden, ob es möglich ist aus den gefundenen Test-Klonen Schablonen zu erzeugen, deren Anpassung oder Veränderung synchron auf die entsprechenden Testfälle die als Klone identifiziert wurden übertragen werden kann. Ebenfalls denkbar ist auch die Erstellung einer Datenbank von Schablonen, aus der geeignete Testfälle für konkrete Probleme ausgewählt werden können. Auf jeden Fall können die Ergebnisse ansonsten zu analytischen Zwecken und der statistischen Datenerhebung genutzt werden.

Motiviert durch eine Untersuchung bei der aufgezeigt wurde, dass unter acht quelloffenen Softwareprojekten im Durchschnitt 8% und bis zu 42% aller Testfälle Klone sind, wird in dieser Arbeit ein Verfahren vorgestellt, mit Hilfe dessen eine automatische Erkennung von Test-Klonen möglich ist [eT12]. Basierend auf einem Werkzeug, dass die Idee des hier vorgestellten Verfahrens implementiert, wurden Testfälle aus *xUnit*-Rahmenwerken für die objektorientierte Programmiersprache *Java* untersucht [Ora11]. Dabei wurden bei manchen Software-Projekten bis zu $80,91\%$ der Testfälle als Klone anderer identifiziert. In jedem der untersuchten Projekte konnte mindestens ein Klonpaar ausgemacht werden.

2. Grundlagen

In diesem Kapitel werden wichtige Begriffe eingeführt und erklärt, deren Verständnis für den weiteren Kontext notwendig ist.

Es werden Grundkenntnisse in Programmiersprachen vorausgesetzt, da die Themen anschaulich an Quelltextausschnitten erörtert werden.

2.1. Testrahmen

Ein Testfall überprüft, bei einer vorgegebenen Menge an Eingabedaten, das Verhalten einer Softwarekomponente. Üblicherweise wird dabei eine bestimmte Funktionalität in einem konkreten Kontext validiert. Eine, im englischen, „*Unit*“ oder ein klar umgrenzter Programmteil wird hierbei auch als eine zu testende Einheit verstanden. Ein Testrahmen stellt dabei Möglichkeiten zur Überprüfung des Verhaltens zur Verfügung. Er hat zudem die Funktion diese Testfälle gesammelt, ob manuell angestoßen oder selbstständig initiiert, auszuführen. Die Ausführung kann beliebig oft wiederholt werden. Die Möglichkeit der Durchführung von Regressionstests hilft bei der Bereinigung von Fehlern, welche bei der Weiterentwicklung entstandenen sind [GHK+98].

JUnit ist ein solcher Testrahmen und gehört zu der Familie der *xUnit*-Testrahmen [BGS12] [Mes07]. Dieser Testrahmen wird in *Java* Software-Projekten eingesetzt. *Java* ist derzeit eine der meist verwendeten objektorientierten Programmiersprachen [TIO12].

Ein *xUnit*-Testfall hat im Allgemeinen folgenden Aufbau:

1. Vorbereitende Maßnahmen (Initialisierung der Eingabedaten)
2. Aufruf der zu testenden Funktionalität
3. Überprüfung des erzeugten Zustands

Diese Struktur wird auch der Aufbau der drei „A“s (*arrange, act, assert*) genannt und wurde in dieser Form von Bill Wake formuliert [Rai04]. Ein Testfall dieser Art ist in Quelltextausschnitt 2.1 zu sehen, in dem jeder entsprechende Schritt durch einen Kommentar gekennzeichnet wurde.

Zum besseren Verständnis dieses Beispiels soll der formelle Aufbau eines *JUnit*-Testfalls erläutert werden. In der aktuellen Version werden Testfälle in dem *JUnit*-Testrahmen

```java
public class FerrariTest() {

   @Test
   public void testAcceleration() {
      // Initialisierung der Eingabedaten (arrange)
      Car ferrari = new Ferrari();

      // Aufruf der zu testenden Funktionalität (act)
      ferrari.accelerate();

      // Überprüfung des neuen Zustands (assert)
      Assert.assertTrue(ferrari.getSpeed() > 0);
   }
}
```

Quelltextausschnitt 2.1: Beschleunigung eines Ferraris

```java
@Test(expected=OutOfFuelException.class)
public void testDrive() throws Exception {

   Car car = new Ferrari();
   while (car.getFuel() >= 0) {
      // fahre bis der Tank leer ist
      car.drive();
   }
}
```

Quelltextausschnitt 2.2: Testfall mit Ausnahmeerwartung

durch die Annotation `@Test` gekennzeichnet. Zu jeder *Java*-Klasse existiert in der Regel genau eine Test-Klasse, die eine Menge an Testfällen enthält. Üblicherweise werden die Namen der Test-Klassen aus dem angehängten Suffix „*Test*“ und dem Namen der zu testenden Klasse zusammengesetzt. Instanzen dieser Klasse werden hierbei als Testobjekte bezeichnet. Eine weitere Konvention besagt, dass die Methodennamen der Testfälle das Präfix „*test*“ tragen sollten [GB07]. Diese Konvention ist historisch bedingt entstanden, da in früheren Versionen von *JUnit* nur derartig bezeichnete Testfälle als jene betrachtet und ausgeführt wurden.

Jeder Testfall kann bei Ausführung vor- und nachgelagerte Methodenaufrufe nach sich ziehen. Entsprechende Methoden werden zum Beispiel mit den Annotationen `@Before` oder `@After` versehen. In diesen können die wiederholte Initialisierung bestimmter Eingabedaten, oder weitere, für eine Sammlung von Testfällen, allgemeingültige Abfolgen von Anweisungen umgesetzt werden.

So genannte Zusicherungen, oder Methoden die den erzeugten Zustand überprüfen, werden in *JUnit* über statische Methoden der `Assert` Klasse ausgeführt. Dies ist in Quelltextausschnitt 2.1 zu sehen. Hier wird ein *Ferrari* beschleunigt und es findet eine anschließende Überprüfung der Geschwindigkeit des Fahrzeugs statt. Ist diese größer 0 km/h, so gilt der Testfall als bestanden.

Weiterhin existieren Testfälle, die das Auftreten einer Ausnahme bei Interaktion mit dem Testobjekt erwarten und überprüfen. Die erwartete Ausnahmeklasse kann im `expected` Attribut der `@Test` Annotation definiert werden. Dies ist in dem Quelltextausschnitt 2.2 zu sehen. Hier fährt ein Fahrzeug sinnbildlich so lange bis der Tank leer ist. Anschließend

```
public class LamborghiniTest() {

   @Test
   public void testAcceleration() {
      Car lamborghini = new Lamborghini();
      lamborghini.accelerate();
      Assert.assertTrue(lamborghini.getSpeed() > 0);
   }
}
```

Quelltextausschnitt 2.3: Beschleunigung eines Lamborghinis

muss die Instanz einer `OutOfFuelException` Ausnahme auftreten. Tritt diese nicht auf, so schlägt der Testfall bei Ausführung fehl.

Die weiteren, in dieser Ausarbeitung vorgestellten, Quelltextbeispiele können in der dargestellten Form dem *JUnit*-Testrahmen entstammen. Es sei jedoch zu erwähnen, dass auch andere, sehr ähnliche Testrahmen für die Programmiersprache *Java* zur Verfügung stehen. Als ein bekannter Vertreter ist an dieser Stelle *TestNG* erwähnenswert [Beu12]. Dieser Testrahmen verwendet einen ähnlichen Terminus und besitzt eine vergleichbare Funktionalität.

2.2. Test-Klon

Um auf die Definition eines Test-Klons überzuleiten, lässt sich der Testfall aus Quelltextausschnitt 2.1 auch auf andere Fahrzeuge übertragen. In dem Quelltextausschnitt 2.3 wird beispielsweise die Instanz eines *Lamborghini* Fahrzeugs beschleunigt. Beide Quelltextausschnitte weisen, sowohl syntaktisch als auch semantisch, eine hohe Ähnlichkeit auf. Die Beschleunigung der beiden Fahrzeuge wirkt sich analog auf der deren Geschwindigkeit aus. Lässt man Kommentare, die Bezeichnungen der Variablen und den Typ der Fahrzeuge außen vor, sind sie sogar identisch. Der Definition nach handelt es sich hierbei um Test-Klone.

2.3. Hilfsmittel zur Lösung

Test-Klone, oder Klone innerhalb von Quelltexten allgemein, können über verschiedene Verfahren ausfindig gemacht werden. Diese Verfahren werden in dem Kapitel 3 vorgestellt und bedienen sich einer Reihe von Datenstrukturen und Algorithmen, welche hier vorgestellt werden.

2.3.1. Längste gemeinsame Teilsequenz

Das Problem der längsten gemeinsamen Teilsequenz ist elementar bei der Auswertung der Ähnlichkeit zwischen zwei Sequenzen. Ein Verfahren, welches dieses Problem löst, sucht die längste gemeinsame Teilsequenz aus Elementen die in mehreren, zu vergleichenden Sequenzen vorkommen. Die längste gemeinsame Teilsequenz lässt sich in polynomieller Zeit finden. Es handelt sich dabei um ein Problem, dass der Klasse der NP-harten Probleme zugeordnet werden kann.

Am Beispiel von zwei Zeichenketten ist dieses Problem sehr einfach vorzustellen. Seien die folgenden zwei Zeichenketten gegeben:

Zeichen	S	K	T	P	A	U	W
Gewicht	1	2	1	1	1	1	1

Zeichen	K	S	T	A	X	J	U
Gewicht	2	1	1	1	1	1	1

Abbildung 2.1.: Gewichtete Zeichenketten

- T*estk**lone***
- M*elone*n

Bei einer Untersuchung der längsten gemeinsamen Teilsequenz würde die Zeichenkette *elone* entstehen, die in den beiden Beispielen entsprechend ihres Vorkommens durch eine kursive Schrift markiert ist. Die längste gemeinsame Teilsequenz entsteht, indem die Sequenz gemeinsamer Zeichen in der Reihenfolge ihres Vorkommens extrahiert wird. Die Endung *ne* beider Zeichenketten ist ebenfalls eine gültige, gemeinsame Teilsequenz, aber in diesem Beispiel nicht die längste. Oft wird in diesem Zusammenhang auch von der gewichtsmäßig schwersten Teilsequenz gesprochen. In dem ersten Beispiel wurde jedes Zeichen uniform gewichtet. Es ist jedoch auch möglich Elemente einer Sequenz unterschiedlich zu gewichten. Dies ist zum Beispiel in den Zeichenketten in der Abbildung 2.1 der Fall. Hier sind `STAU` und `KTAU` zwei gültige, gleichlange Teilsequenzen. Letztere hat jedoch, aufaddiert nach der Wertigkeit der vorkommenden Zeichen, das größere Gewicht und ist deshalb die gesuchte Teilsequenz.

2.3.2. Levenshtein-Distanz

Die Levenshtein-Distanz ist ein Maß, mit dem Zeichenketten verglichen werden können [Lev66]. Sie gibt an wie viele Lösch-, Einfüge- und Ersetz-Operationen notwendig sind, um eine Zeichenkette in die jeweils andere zu überführen. Je geringer dieser Wert ausfällt, desto ähnlicher sind sich die zu vergleichenden Zeichenketten.

Um die Zeichenkette „Testklone“ in „Melonen“ zu überführen, wären folgende Operationen notwendig:

1. Ersetze `T` durch `M`
2. Ersetze `s` durch `l`
3. Lösche `t`
4. Lösche `k`
5. Füge `n` an das Ende der Zeichenkette an

Die Levenshtein-Distanz beträgt für dieses Beispiel fünf.

Der Einsatzzweck der Levenshtein-Distanz ist vergleichbar mit der Suche nach der längsten gemeinsamen Teilsequenz.

2.3.3. Kosinus-Ähnlichkeit

Die Ähnlichkeit zweier Vektoren kann über die Kosinus-Ähnlichkeit abgeschätzt werden. Die Berechnung ist durch folgende Formel definiert:

$$\cos(\theta) = \frac{\vec{a} \cdot \vec{b}}{\|\vec{a}\|\|\vec{b}\|} \tag{2.1}$$

Lassen sich Merkmale von Texten numerisch in Form von Vektoren abbilden, so kann dieses Maß als Bewertungskriterium eingesetzt werden. Wie solche Vektoren erstellt werden können, wird später erklärt.

```
1 return (argument + 5) * 2;
```

Quelltextausschnitt 2.4: Rückgabe einer Summe

2.3.4. Abstrakter Syntaxbaum

Ein abstrakter Syntaxbaum überführt die in einem Dokument vorkommenden, charakteristischen Elemente in eine Baumstruktur [Jon03]. Quelltexte können durch diese Form in ihrer exakten, syntaktischen Struktur wiedergegeben werden [Pin05].

Abstrakte Syntaxbäume finden hauptsächlich Anwendung im Compilerbau, in dem sie eine Zwischendarstellung vor der Ausführung repräsentieren. Sie können allerdings auch zum Zweck der Klon-Erkennung eingesetzt werden. Entscheidend ist die Granularität der Abstraktion der Elemente. In einem Quelltext kann beispielsweise ein Operator sowohl durch einen eigenen Knotentyp repräsentiert, als auch zusammengefasst mit anderen Operatoren abstrahiert werden.

Für die Programmiersprache *Java* stehen mehrere Rahmenwerke zur Verfügung, die aus einem Quelltext einen abstrakten Syntaxbaum ableiten können. Das *Eclipse JDT* Rahmenwerk ist ein solches und orientiert sich dabei stark an den definierten Begriffen der, von *Oracle* herausgegebenen, Sprachspezifikation für *Java* [SJA+12] [Ora11]. An dieser Notation orientiert sich auch die folgende Abbildung eines abstrakten Syntaxbaumes.

Die Darstellung eines abstrakten Syntaxbaumes ist relativ unübersichtlich, deshalb soll an dieser Stelle nur ein kleiner Einblick gegeben werden. Bestehe die Rückgabe einer Funktion, wie in dem Quelltextausschnitt 2.4 gezeigt, aus der Summe einer Variable und einer Konstanten die mit einer weiteren Konstanten multipliziert wird, so könnte der entsprechende abstrakte Syntaxbaum wie in Abbildung 2.2 aussehen.

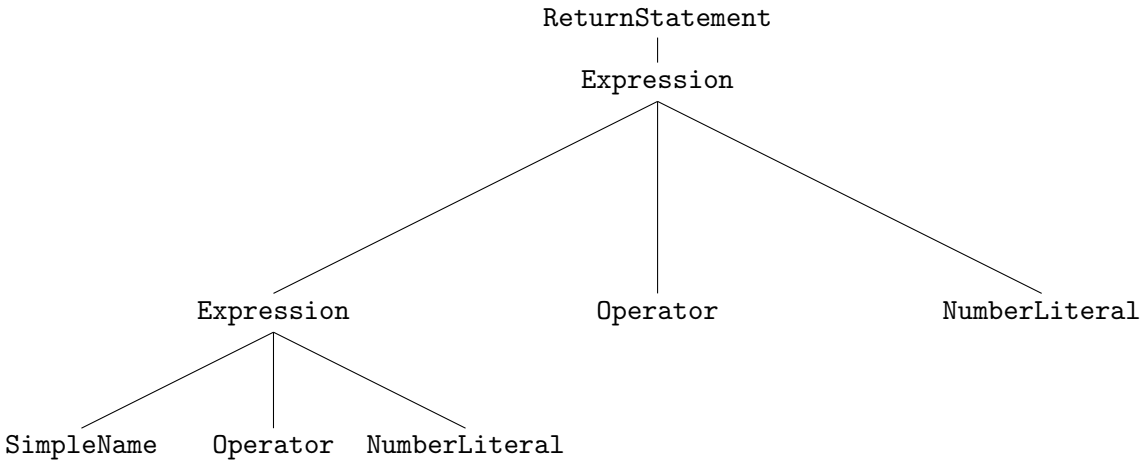

Abbildung 2.2.: Abstrakter Syntaxbaum

3. Verwandte Arbeiten

Nach gründlicher Recherche konnte keine direkt verwandte Untersuchung zur Identifikation von Test-Klonen gefunden werden. Allerdings ist das gestellte Problem eng mit der Plagiatssuche innerhalb von Quelltexten, sowie der Erkennung von Codewiederholung verwandt. Bei diesen Problemen ist es ebenfalls das Ziel syntaktisch oder semantisch ähnliche Quelltextteile zu finden. Im Rahmen dieses Kapitels werden einige bekannte Verfahren vorgestellt. Diese lassen sich in mehrere Kategorien einteilen, die im Folgenden näher beschrieben werden [CDR09] [RCK09]. Die Verfahren unterscheiden sich vor allem in der Darstellungsform der zu vergleichenden Daten. Eine Übersicht der Verfahren, sowie deren jeweilige Vor- und Nachteile, finden sich in Tabelle 3.1 wieder.

3.1. Metrische Analyse

Bei der metrischen Analyse wird üblicherweise jedem Vergleichskandidaten aus einer Menge von Quelltexten ein Vektor zugeordnet, der dessen Merkmale abbildet. Die Koordinaten eines Vektors können zum Beispiel die Anzahl der Anweisungen, Verzweigungen und Schleifen abbilden [DLS81]. Die Vektoren, oder einzelne Koordinaten eines Vektors, können entweder auf absolute Gleichheit oder eine relative Ähnlichkeit untersucht werden, um zu einem Ergebnis in einem Vergleich zu kommen.

Untersucht man für die Quelltextbeispiele aus Tabelle 3.2 die Anzahl der Schleifen und Bedingungsanweisungen, so erhält man die jeweils nebenan abgebildeten Vektoren.

Metriken lassen sich im Allgemeinen schnell erstellen und auswerten, weshalb sie häufig eingesetzt werden. Allerdings lässt das Ergebnis einer solchen Auswertung nur erahnen, ob eine tatsächliche Ähnlichkeit zwischen zwei Vergleichskandidaten besteht. Ein häufiger Anwendungsfall der Metriken ist die Reduzierung der Menge der zu vergleichenden Daten für im Allgemeinen aufwendigere, aber genauere Verfahren.

3.2. Textbasierte Analyse

Plagiate können textbasiert gefunden werden. Hierbei findet ein Vergleich zwischen zwei, in der Regel unveränderten, Texten statt. Mit unverändert ist dabei im Speziellen die Beibehaltung der äußeren Form eines Textes gemeint. Die Übereinstimmung zwischen zwei Texten kann anschließend, beispielsweise durch die *Levensthein-Distanz*, ermittelt werden.

Verfahren	**Vorteile**	**Nachteile**
Metrische Analyse	+ Sehr performant + Guter Vorfilter um Vergleichskandidaten für aufwendigere Vergleichsverfahren auszuschließen	– Die direkten Ergebnisse sind sehr unpräzise
Textbasierte Analyse	+ Performant + Kopierte Quelltextteile werden schnell erkannt	– Große Auswirkung auf das Ergebnis bei unterschiedlicher Formatierung der Vergleichskandidaten – Semantisch ähnliche Kandidaten, die sich nur in der Typisierung unterscheiden, werden nicht erkannt
Lexikalische Analyse	+ Performant + Formatierung der Quelltexte beeinflusst das Ergebnis nicht + Abstrahierte Typisierung und Normalisierung	– Semantik kann bei Abstraktion verloren gehen oder wird nicht berücksichtigt
Syntaktische Analyse	+ Formatierung der Quelltexte beeinflusst das Ergebnis nicht + Die abstrahierte Repräsentation behält den Inhalt der Kandidaten bei und kann flexibel eingesetzt werden	– Hoher Speicherbedarf – Laufzeiterwartung ist, im Vergleich zur lexikalischen Analyse, etwas höher
Semantische Analyse	+ Semantisch, aber nicht zwingend syntaktisch, ähnliche Quelltexte können erkannt werden	– Hohe Laufzeiterwartung durch Vergleichsoperationen auf Graphen

Tabelle 3.1.: Übersicht der Vor- und Nachteile verwandter Arbeiten

```
@Test
public void testRefuelCarPool() {
   Car[] carPool;
   for (Car car : carPool) {
      while (car.getFuel() < 30) {
         car.refuel();
      }
      Assert.assertTrue(car.getFuel() == 30);
   }
}
```

$$\vec{m} = \begin{pmatrix} 2 \\ 0 \end{pmatrix}$$

```
@Test
public void testUsableCarsInPool() {
   Car[] carPool;
   for (Car car : carPool) {
      if (car.getFuel() == 0) {
         Assert.assertFalse(car.isFuelled());
      }
   }
}
```

$$\vec{m} = \begin{pmatrix} 1 \\ 1 \end{pmatrix}$$

Tabelle 3.2.: Beispiele für Vektoren bei einer metrischen Analyse

```
for (int i = 0; i < 10; i++)
```

```
for (
int i = 0;
i < 10;
i++)
```

Abbildung 3.1.: Vergleich von Schleifenrümpfen bei unterschiedlicher Konfiguration im NICAD-Verfahren

Es handelt sich bei diesem Verfahren nicht grundsätzlich um eine auf Quelltexte spezialisierte Lösung, sondern um eines, dass allgemein Texte sequentiell, zum Beispiel zeilenweise, untereinander vergleicht.

Es ist anzumerken, dass die meisten Vertreter der textbasierten Analyse sehr anfällig gegenüber unterschiedlichen Formatierungen sind. Beispielsweise wirken sich zusätzliche oder fehlende Leerzeichen und Zeilenumbrüche, die in Quelltexten nicht zwingend einen Einfluss auf das Verhalten zur Laufzeit zur Folge haben, unmittelbar auf das Ergebnis einer textbasierten Analyse aus.

Eine Ausnahme stellt das *NICAD* Verfahren dar [RC08]. Dieses soll exemplarisch als Vertreter dieser Gruppe vorgestellt werden. Bei diesem Verfahren wird ein Quelltext nach festgelegten Regeln formatiert, um unter den zu vergleichenden Kandidaten eine identische Form zu gewährleisten. Ein resultierender Vorteil ist die Möglichkeit die Anweisungen, die anschließend üblicherweise zeilenweise dargestellt werden, unabhängig von der ursprünglichen Formatierung vergleichen zu können.

Zuerst werden alle Sequenzen, die auf Grund einer konfigurierten Mindestgröße potentiell Teil eines Klonpaares sein können, identifiziert. Diese Anweisungsblöcke werden anschlie-

ßend, entsprechend vorhandener Regeln, formatiert. Kommentare, überflüssige Leerzeichen und Zeilenumbrüche werden dabei in der extrahierten Form weggelassen.

Die formatierten Anweisungen werden zeilenweise, zum Beispiel mit dem aus *Unix*-Derivaten bekanntem `diff` Werkzeug, verglichen [oC84]. Wie viel Information in einer Zeile enthalten ist, beziehungsweise wie detailliert der Vergleich stattfinden soll, ist eine Frage der Konfiguration. Ein Schleifenrumpf kann eine einzeilige Anweisung sein. Allerdings können Start-, Endwert und Schrittweite auch zeilenweise ausgewertet werden, wie in einem Vergleich in Abbildung 3.1 zu sehen ist. Der offensichtliche Vorteil der zweiten Konfiguration ist die Möglichkeit eine Übereinstimmung zweier Schleifenrümpfe prozentual abzuschätzen. Sind zwei Schleifenrümpfe, abgesehen von deren Schrittweite, identisch, so ist die Ähnlichkeit im Vergleich zur ersten Darstellungsweise $> 0\%$.

Im nächsten Schritt werden Anweisungen normalisiert. Da dies ein Merkmal einer *lexikalischen Analyse* ist, wird auf diesen Schritt an dieser Stelle nicht näher eingegangen.

Zuletzt wird die längste gemeinsame Teilsequenz gesucht. Die Übereinstimmung von zwei Texten kann prozentual anhand der Länge dieser Sequenz und der Summe der Zeilen abgeschätzt werden.

3.3. Lexikalische Analyse

Bei einer lexikalischen Analyse wird der Quelltext in logische Einheiten zerlegt und normalisiert. Die Einheiten stellen meist eine abstrahierte Form des ursprünglichen Textes dar. Beispielsweise können Literale unabhängig von ihrem konkretem Wert auf die gleiche Art und Weise ersetzt werden. Das heißt die Zahlen 0 und 1 oder die Zeichenketten „apfel“ und „banane“ können durch eine identische Darstellung repräsentiert werden. Die Formatierung des Quelltextes spielt bei der lexikalischen Analyse keine weitere Rolle, da diese bei der Transformation, oder Abstraktion, verworfen wird. Zwei wesentliche Vorteile, im Vergleich zu einem rein textbasierten Verfahren, kommen somit zum Vorschein: Es ist nicht notwendig die Formatierung der abstrahierten Daten zu berücksichtigen. Zum anderen wird somit eher eine relative Ähnlichkeit, unabhängig von der Typisierung und der Bezeichnungen und nicht vorwiegend eine Gleichheit ermittelt. Dieses Verfahren stellt somit die Brücke zwischen einem rein textbasierten Vergleich und der, im Späteren vorgestellten, syntaktischen Analyse dar. Bekannte Vertreter von Werkzeugen, die eine Plagiatserkennung in Quelltexten ermöglichen, sind unter anderem *JPlag* und *Moss* [MKK] [Aik11]. Beide implementieren die Idee der lexikalischen Analyse.

JPlag konnte in einer aktuellen Untersuchung als ein sehr gut geeignetes Werkzeug zur Erkennung ähnlicher Quelltexte ausgemacht werden [WWKM12]. In dem Verfahren werden Plagiate nach Ablauf von zwei internen Phasen erschlossen [PPM02].

1. Der Quelltext wird eingelesen und in textuelle Merkmale oder Bausteine zerlegt. Leerzeichen, Kommentare und gegebenenfalls weitere Elemente, finden keine Entsprechung in der abstrahierten Form des Quelltextes.
2. Sequenzen von abstrahierten Merkmalen werden anschließend paarweise verglichen. Dabei wird festgestellt, ob ein Kandidat anteilig eine Sequenz des jeweils Anderen enthält. Hier kommt der „*Greedy String Tiling*“ Algorithmus zum Einsatz. In diesem Algorithmus wird eine gefundene Teilsequenz, bis zu ihrer Maximallänge, sukzessiv, bei gefundener Übereinstimmung, erweitert. Dem Grundsatz nach darf jede Sequenz dabei nur einmal als Plagiat zugeordnet werden, sodass bereits erschlossene Paarungen markiert und von der weiteren Verarbeitung ausgeschlossen werden. Die mehrfache Verwendung einer Sequenz ist somit ausgeschlossen und jede Sequenz kann höchstens einer ähnlichen Entsprechung zugeordnet werden.

Die Laufzeit ist, mitsamt einigen vorhandenen Optimierungen, im Durchschnitt relativ gering. Dabei wird gegen eine hohe Anzahl an potentiellen Störfaktoren, die eine Plagiatserkennung behindern können, vorgegangen.

3.4. Syntaktische Analyse

Eine syntaktische Analyse findet auf Grundlage von abstrakten Syntaxbäumen statt. Dabei wird meist nicht jeder Knoten eines Syntaxbaums mit dem seines Vergleichskandidaten auf Gleichheit oder Ähnlichkeit überprüft. Stattdessen ist es, wie im *CloneDr*-Verfahren, üblich bestimmte Knoten vor einem Vergleich zu einem Teilbaum zu aggregieren und diesem zum Beispiel einen Hash-Wert zuzuordnen, sodass ein vereinfachter Vergleich möglich ist [BYM+98]. Eine gute Hash-Funktion muss in diesem Fall jedoch jedes Element innerhalb dieser Zusammenfassung, oder zumindest elementare Teile dieser, berücksichtigen.

Im *CloneDr* Verfahren werden n Knoten gehasht und auf etwa $\frac{1}{10}n$ Behälter verteilt. Sehr wichtig ist dabei, dass ähnliche, aber nicht identische, Elemente oder Knoten in einem gemeinsamen Behälter enthalten sind. Anschließend werden nur Elemente innerhalb eines gemeinsamen Behälters detaillierter untereinander verglichen. Es ist hierbei wichtig, dass kleine, eher unbedeutende Knoten von der Hash-Funktion unberücksichtigt bleiben. Dies kann in der Programmiersprache *Java* beispielsweise ein `this`-Präfix sein. Kann eine Variable innerhalb ihres Geltungsbereichs optional mit einem `this`-Präfix versehen werden, so sollte dieser Variable in beiden Fällen derselbe Hash-Wert zugeordnet werden.

Die Ähnlichkeit zweier Teilbäume lässt sich anschließend über folgende Formel prozentual abschätzen:

$$\ddot{A}hnlichkeit = \frac{2 \cdot S}{(2 \cdot S + L + R)} \cdot 100\% \tag{3.1}$$

S Anzahl gemeinsamer Elemente
L Anzahl Elemente die nur Teil des linken Baumes sind
R Anzahl Elemente die nur Teil des rechten Baumes sind

Zusätzlich wird ein Schwellenwert für die Masse der Teilbäume definiert, unter der keine Vergleiche stattfinden. Die Masse bezeichnet die konfigurierbare Gewichtung der Teilbäume und somit den wertmäßigen Beitrag zum Gesamtergebnis. Besonders kurze Sequenzen, oder gar einzelne Befehle, können so verworfen werden und werden nur als Teil eines größeren Aggregats betrachtet. Ein Vergleich findet demnach nur auf größeren, zusammenhängenden Teilbäumen statt.

3.5. Semantische Analyse

In allen bisher vorgestellten Verfahren fand ein Vergleich von Quelltexten rein auf syntaktischer Ebene statt. Bei einer semantischen Analyse geht es hingegen darum Anweisungen zwischen denen eine Datenabhängigkeit besteht aus Quelltexten zu extrahieren und in einen Graph zu überführen [Kri03]. Das übergreifende Ziel der semantischen Analyse ist die Abstraktion des Quelltextes, um die Analyse von unterschiedlich formulierten, semantischen Klonen zu erleichtern, oder Teilzusammenhänge zu finden.

In den hier erstellten Graphen wird die Datenverarbeitung, oder der Datenfluss, innerhalb eines Programms abgebildet. Eine Abhängigkeit zwischen Anweisungen besteht beispielsweise zwischen der Deklaration einer Variablen und deren weiteren Verwendung oder Modifikation in einem Programm. Sogenannte Kontrollpunkte stellen dabei Knoten dar, an denen Verzweigungen auftreten können. Bedingungsanweisungen und Schleifen sind unter anderem Kontrollpunkte. Die Knoten innerhalb eines Graphen sind nur dann durch eine

```
1  @Test
2  public void testAcceleration() {
3      // Teil von G1
4      Car ferrari = new Ferrari();
5
6      // Teil von G2
7      long start = System.currentTimeMillis();
8
9      // Teil von G1
10     ferrari.accelerate();
11
12     // Teil von G2
13     long finish = System.currentTimeMillis();
14     LOGGER.log("acceleration␣took␣" + (finish - start) + "ms");
15
16     // Teil von G1
17     Assert.assertTrue(ferrari.getSpeed() > 0);
18 }
```

Quelltextausschnitt 3.1: Beschleunigung eines Ferraris und Laufzeitmessung

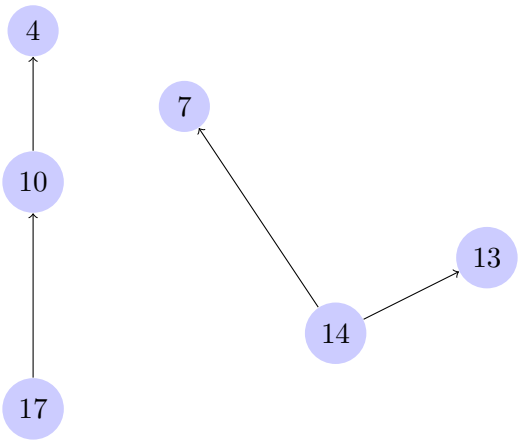

Abbildung 3.2.: Graphen, die eine Datenabhängigkeit aufzeigen

Kante verbunden, wenn eine tatsächliche Abhängigkeit besteht.

Betrachtet man den bereits vorgestellten Testfall 2.1, in dem ein Ferrari beschleunigt wurde und die in Quelltextausschnitt 3.1 um eine Ausgabe erweiterte Modifikation, so weisen diese eine offensichtliche Ähnlichkeit auf. Allerdings enthält das zweite Beispiel zusätzliche Anweisungen, um die Laufzeit auszuwerten. Die Identifikation dieser Testfälle als Klone würde, mit den bislang vorgestellten Mitteln, zu Problemen führen. Angenommen die Anweisungen aus dem Quelltextausschnitt 2.1 können durch einen zusammenhängenden Graph *G1* repräsentiert werden, der auch Teil der Graphen-Menge aus dem Testfall 3.1 ist, so lässt sich eine Teilbeziehung herstellen. Das zweite Beispiel besteht hierbei aus zwei unverbundenen Graphen *G1* und *G2*, wobei *G2* die Laufzeitmessung enthält. Die entsprechenden Kommentare im Quelltextausschnitt markieren dabei die Zugehörigkeit zu den entsprechenden Graphen. Zwischen beiden Graphen in Abbildung 3.2 besteht keine direkte Datenabhängigkeit. Die Bezeichnungen der Knoten in der Abbildung referenzieren die Zeilennummern aus dem Quelltextausschnitt 3.1.

Untersuchungen auf Grundlage von Deckard beschäftigen sich mit der semantischen Ana-

lyse und können als Vertreter dieser Gruppe genannt werden [JMS07] [GJS08].

Die Durchführung einer semantischen Analyse hat, auf Grund der notwendigen Vergleichsoperationen auf Graphen, eine sehr hohe Laufzeit. Eine semantische Analyse kann dabei statisch oder dynamisch erfolgen. Bei einer statischen Analyse werden alle Ausführungszweige eines Quelltextes einbezogen. Bei einer dynamischen Analyse wird, bei ausreichenden Eingabedaten, nur ein bestimmter Ausführungszweig simuliert, aus dem ein repräsentativer Graph erzeugt werden kann.

4. Analyse und Entwurf

Aufbauend auf den vorgestellten Verfahren aus dem vorangehenden Kapitel werden werden potentielle Lösungsverfahren zur Identifikation von Test-Klonen diskutiert. Gesucht werden Testfälle die syntaktisch eine hohe Ähnlichkeit aufweisen und idealerweise auch einen semantischen Zusammenhang besitzen. Der Zusammenhang muss dabei nicht durch identische Testobjekte gegeben sein. Die Typisierung steht unter Anderem außen vor. Die gemeinsame Semantik ist, vor allem zwecks der in der Einleitung formulierten Ziele, zur Klonverwertung gewünscht. In dem hier entwickelten Verfahren findet deshalb ein Vergleich auf generalisierten oder normalisierten Ausschnitten statt. Einige bekannte Verfahren zur Klon-Erkennung sind aus dem vorherigen Kapitel bekannt und werden hier verwendet.

Die konkrete Klassifizierung eines Test-Klons hängt stark von dem Verwendungszweck der Ergebnisse ab. Das im Folgenden vorgestellte Verfahren ist auf eine syntaktische Ähnlichkeitsbewertung ausgelegt. Semantische Kriterien zur Klassifizierung sind jedoch ebenfalls Teil der Bewertung.

Eine elementare Frage ist die Feststellung, wann Test-Klone potentiell auftreten können. Zu jedem Testobjekt können mehrere Testfälle existieren, die bei unterschiedlicher Initialisierung einen Zustand überprüfen. Theoretisch können Testfälle, wie in Quelltextausschnitt 4.1 zu sehen ist, unterschiedlich formuliert werden, obgleich sie eine äquivalente Semantik aufweisen. Allerdings ist bei der Analyse aufgefallen, dass dies eher die Ausnahme darstellt. Bestehen zu einer Schnittstelle beispielsweise mehrere Implementierungen, die unabhängig von einander getestet werden, so müssen in diesem Fall mehrere Testfälle eine hohe semantische Ähnlichkeit aufweisen. Die Schlussfolgerung kann gezogen werden, da die Schnittstelle ein bestimmtes, zu erwartendes Verhalten definiert und alle Testfälle dieses überprüfen. Wurden diese Implementierungen von exakt einem Programmierer angefertigt, so kann außerdem eine hohe syntaktische Ähnlichkeit gegeben sein, da diese dazu neigen Probleme auf eine ähnliche Art und Weise wiederholt zu lösen. Dies definiert die erste Annahme 4.1, von der ausgegangen wird.

Annahme 4.1 (Syntaktische Ähnlichkeit in genau einem Software-Projekt) *Innerhalb genau eines Software-Projekts wird ein klar definierter Programmierstil eingehalten. Test-Klone weisen deshalb eine hohe syntaktische Ähnlichkeit auf.*

```java
@Test
public void testIfCarColorIsRed() {
    Car car;
    if (car.getColor() == Color.RED) {
        Assert.assertTrue(car instanceof Ferrari);
    } else {
        Assert.assertFalse(car instanceof Ferrari);
    }
}

@Test
public void testSwitchCarColorIsRed() {
    Car car;
    switch (car.getColor()) {
    case Color.RED:
        Assert.assertTrue(car instanceof Ferrari);
        break;
    default:
        Assert.assertFalse(car instanceof Ferrari);
    }
}
```

Quelltextausschnitt 4.1: Äquivalente Testfälle bei unterschiedlicher Formulierung

```java
@Test
public void testBrakes() {
    Car ferrari = new Ferrari();
    ferrari.brake();
    Assert.assertTrue(ferrari.getSpeed() == 0);
}
```

Quelltextausschnitt 4.2: Abbremsen eines Ferraris

Wird ein Programm von einem kleinen Personenkreis erzeugt, so neigt dieser dazu einen gemeinsamen Programmierstil anzunehmen, oder sich auf einen zu einigen. Werden Testfälle gänzlich mit leichten Anpassungen kopiert, so muss in der Regel ebenfalls eine hohe syntaktische Ähnlichkeit gegeben sein. Es wird angenommen, dass zumindest im Rahmen genau eines Software-Projekts Probleme auf eine ähnliche Art und Weise gelöst werden und deshalb auch Überschneidungen unter den Testfällen auftreten. Diese können als Test-Klone identifiziert werden. Hier ist sowohl eine syntaktische, als auch eine semantische, Ähnlichkeit zu erwarten.

Eine weitere Intention ist es, ausschließlich Testfälle als Klone zu bezeichnen, die sich grundsätzlich nur anhand ihrer Eingabedaten unterscheiden, da dadurch die semantische Ähnlichkeit forciert wird. Ein Indikator für eine äquivalente Testlogik sind die verwendeten Zusicherungen. Diese können, bei der Suche nach der längsten gemeinsamen Teilsequenz, entsprechend höher gewichtet werden. In Bezug auf den Quelltextausschnitt 2.1, indem ein Ferrari beschleunigt wurde, kann ein weiterer Testfall existieren, der dessen Bremsleistung testet. Dies ist in Quelltextausschnitt 4.2 zu sehen. Die Testfälle unterscheiden sich hier in ihren Zusicherungen, die einen Rückschluss auf die Testlogik geben. Dies führt zu der Annahme 4.2.

Annahme 4.2 (Zusicherungen als semantisches Unterscheidungskriterium) *Zusicherungen definieren die Testlogik eines Testfalls und können als Unterscheidungskriterium bei der Test-Klon Identifikation eingesetzt werden.*

Zusicherungen lassen sich durch die genauere Untersuchung von Befehlszeilen und dem Wissen über die verwendete Schnittstelle, die diese definieren, identifizieren.

Aus den hier aufgestellten Annahmen kann ein geeignetes Verfahren abgeleitet werden. Vorgeschlagen wird ein sehr generisches Verfahren, um die Identifikation von Klonen zu ermöglichen. Dieses ist sequentiell in der Abbildung 4.1 zu sehen und wird im Folgenden näher beschrieben. Eingehend wird eine Menge von Quelltexten in vier Schritten untersucht und eine Liste der Test-Klone wird anschließend ausgegeben.

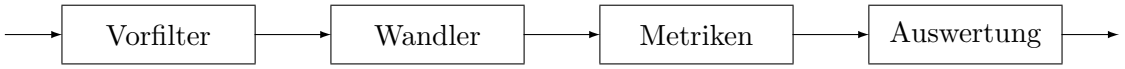

Abbildung 4.1.: Sequentieller Verfahrensablauf

1. **Vorfilter**

   In diesem Schritt werden potentielle Vergleichskandidaten identifiziert. Sollen, wie in dieser Arbeit, Testfälle untersucht werden, so versucht man diese aus der Menge der untersuchten Daten zu extrahieren.

2. **Wandler**

   Die Vergleichskandidaten können in diesem Schritt normalisiert oder verändert werden, um den Ablauf der folgenden Schritte zu erleichtern oder zu verbessern.

3. **Metriken**

   Verschiedene Metriken können frühzeitig Paare, welche mit hoher Wahrscheinlichkeit keine Ähnlichkeit aufweisen werden, bei geringer Laufzeit ausschließen.

4. **Finale Auswertung**

   Die übrig gebliebenen Vergleichskandidaten werden mit einem geeigneten Algorithmus verglichen.

Im Folgenden werden die einzelnen Schritte näher beschrieben.

4.1. Vorfilter

Ein Vorfilter hat die Aufgabe potentielle Vergleichskandidaten zu identifizieren. Um Testfälle analysieren zu können, müssen diese zuerst erkannt werden. Wie dies konkret geschehen kann ist von dem verwendeten Testrahmen abhängig. Exemplarisch steht hierfür die `@Test` Annotation aus *JUnit*, mit der Testfälle gekennzeichnet werden.

Zu beachten ist, dass der restliche, gefilterte Quelltext nicht zwangsläufig verworfen werden darf. Dies würde dazu führen, dass man beispielsweise im späteren Verlauf keinen Zugriff mehr auf vor- oder nachgelagerte Methodenaufrufe hätte. Es ist wichtig diese Daten separat zu verwalten.

Abschließend können Testfälle anhand ihrer Länge verworfen werden. Beispielsweise sind

```
1  @Test
2  public void testDriveFerrari() {
3      Driver driver = new Driver("Jürgen"); // Fahrer
4      Car car = new Ferrari(); // Fahrzeug
5      driver.drive(car);
6      Assert.assertTrue(car.hasMoved());
7  }
8
9  @Test
10 public void testDriveLamborghini() {
11     Car car = new Lamborghini(); // Fahrzeug
12     Driver driver = new Driver("Thomas"); // Fahrer
13     driver.drive(car);
14     Assert.assertTrue(car.hasMoved());
15 }
```

Quelltextausschnitt 4.3: Einfluss der Abfolge von Befehlszeilen

leere Testfälle, abgesehen von der gegebenenfalls gewünschten, statistischen Berücksichtigung, in aller Regel bei der Klonsuche nicht von Interesse.

4.2. Wandler

Ein Wandler soll den Quelltext so modifizieren, dass der Vergleich auf Ähnlichkeit vereinfacht wird. Hier sind der Kreativität nahezu keine Grenzen gesetzt, sodass im Folgenden größtenteils potentielle Optimierer jeweils nur kurz vorgestellt werden.

4.2.1. Normalisierung der Befehlszeilensequenz

Generell kann die Sortierung der Befehlszeilen zweier Vergleichskandidaten die Suche nach der längsten gemeinsamen Teilsequenz behindern. In Quelltexten kann dieser Fall, wie in den Testfällen aus dem Quelltextausschnitt 4.3 zu sehen ist, bei der Deklarationen von Variablen auftreten.

Obwohl die Reihenfolge der Initialisierung des Fahrzeugs und des Fahrers keine Rolle auf die Methodenaufrufe der Zeilen fünf und dreizehn hat, spielt diese, bei der Suche nach der längsten gemeinsamen Teilsequenz beider Testfälle, eine wichtige Rolle. Lässt man die Typen außen vor, so unterscheidet sich die Initialisierung beider Objekte durch die übergebenen Parameter. Die Konstruktoraufrufe sind sich, nach den hier verwendeten Bewertungskriterien, deshalb nicht ähnlich. Daher besteht eine längste gemeinsame Teilsequenz auch nur jeweils aus drei der vier Anweisungen beider Methodenrümpfe. Dabei ist entweder die Initialisierung des Fahrzeugs, oder die des Fahrers in der längsten gemeinsamen Teilsequenz enthalten.

Ein Regelwerk zur Anordnung der Anweisungen kann diesem Problem Abhilfe schaffen. So können Deklarationen stets anhand ihrer Erstverwendung sortiert werden. Dies ist im ersten Testfall aus dem Quelltextausschnitt der Fall. Hier wird in Zeile fünf eine Funktionlität des Fahrers `driver` aufgerufen und das Fahrzeug `car` dient dabei als Argument. Die Deklaration des Fahrers ist entsprechend früher anzusetzen.

4.2.2. Normalisierung äquivalenter Methodenaufrufe

Methodensignaturen können sich in der Anordnung der Parameter, oder gar in der Anzahl der Parameter unterscheiden, jedoch funktionsgleich sein. Die Umstrukturierung der Reihenfolge der Parameter ist komplex und erfordert gegebenenfalls ein manuelles Eingreifen.

```
1 public void info(String message) {
2    log(message, Level.INFO);
3 }
4
5 public void log(String message, Level level) {
6    // Implementierung
7 }
```

Quelltextausschnitt 4.4: Weiterleitung eines Methodenaufrufs

Bei der Erkennung von funktionsgleichen Methoden, die sich in der Parameteranzahl unterscheiden, gibt es mindestens einen Fall bei dem auf einfache Art und Weise eine Äquivalenzzuordnung getroffen werden kann. Unter der Annahme, dass wie in Quelltextausschnitt 4.4, zwei Methoden oder Funktionen existieren, von der eine nur einen Aufruf zu einer überdefinierten Methode kapselt, können Aufrufe auf diese Methoden normalisiert werden. Die Methode `info` kann somit durch einen Methodenaufruf von `log` ersetzt werden, indem der Aufruf um den fehlenden Parameter ergänzt wird.

Gerade zwischen verschiedenen Rahmenwerken, in denen solche weiterleitenden Methodenaufrufe nicht auftreten, kann es jedoch ein Problem darstellen eine äquivalente Funktionalität von Methoden zu erkennen. In den Schnittstellen der Testrahmen *JUnit* und *TestNG* liegen logisch gleiche Möglichkeiten zur Zustandsüberprüfung vor. Diese können sich jedoch stark in der jeweiligen Implementierung unterscheiden und deshalb nicht generalisiert werden. Hier wäre gegebenenfalls eine manuelle Zuordnung nötig.

Im Folgenden werden zwei Signaturen aus den `Assert` Klassen von *JUnit* und *TestNG*:

1. **JUnit**
   `assertEquals(Object expected, Object actual)`
2. **TestNG**
   `assertEquals(Object actual, Object expected)`

Es wird nun vorausgesetzt, dass eine manuelle Äquivalenzzuordnung der Methoden erfolgt ist. In beiden Rahmenwerken existieren weitere, sich äquivalent verhaltende Methoden, die um den zusätzlichen Parameter `message` erweitert sind. Dieser Parameter setzt die auszugebende Nachricht beim Scheitern eines Testfalls.

1. **JUnit**
   `assertEquals(String message, Object expected, Object actual)`
2. **TestNG**
   `assertEquals(Object actual, Object expected, String message)`

Sollte, wie in Quelltextausschnitt 4.4, auch hier innerhalb der Testrahmen eine Aussage über eine äquivalente Funktionalität getroffen werden können, so kann diese anschließend übergreifend auf beide Rahmenwerke angewendet werden.

```java
List<Integer> initialList;

@Test
public void testSortSequence() {
   List<Integer> input = Collections.copyOf(initialList);
   List<Integer> output = new ArrayList<Integer>();
   while (!input.isEmpty()) {
      output.add(getMinimum(input));
   }
   assertTrue(output.size() == initialList.size());
   // hier wird kann getestet werden ob die Ausgabe sortiert ist
}

@Test
public void testRandomSequence() {
   List<Integer> input = Collections.copyOf(initialList);
   List<Integer> output = new ArrayList<Integer>();
   while (!input.isEmpty()) {
      output(getRandom(input));
   }
   assertTrue(output.size() == initialList.size());
}

private static Integer getMinimum(List<Integer> list) {
   Integer minimum = Integer.MAX_VALUE;
   for (Integer element : list) {
      if (element < minimum) {
         minimum = element;
      }
   }
   list.remove(minimum);
   return minimum;
}

private static Integer getRandom(List<Integer> list) {
   int index = Math.random() * list.size();
   Integer result = list.get(index);
   list.remove(index);
   return result;
}
```

Quelltextausschnitt 4.5: Äquivalenter Testfallaufbau bei unterschiedlicher zu testender Funktionalität

4.2.3. Einschieben externer Sequenzen

Testfälle bestehen, im Allgemeinen, aus wenigen Anweisungen. Je kleiner die Menge der Anweisungen ist, desto schwieriger fällt die Beurteilung der Ähnlichkeit, beziehungsweise auch die Beurteilung der gemeinsamen Testlogik. Eine Möglichkeit diese Menge zu vergrößern, ist es Testfälle um weitere Quelltextsequenzen, die zu diesen in Relation stehen, zu erweitern.

Zusätzlich können vor- und nachgelagerte Methodenaufrufe in den Vergleich der Testfälle involviert werden. Soweit vorhanden, vergrößert man damit die zu vergleichende Sequenzlänge. Dies kann einen positiven Einfluss auf das Gesamtergebnis haben. Allerdings ist die Relevanz der eingeschobenen Sequenzen für den konkreten Testfall, beispielsweise durch eine Untersuchung der Datenabhängigkeit, zu bestimmen. Hier kann eine semantische

Analyse erforderlich sein. Außerdem muss das Verhalten für den Fall festgelegt werden, falls zwei Testfälle jeweils vor- und nachgelagerte Methodenaufrufe besitzen oder nicht.

Noch interessanter ist die Frage, ob Methodenaufrufe tatsächlich durch das Einschieben der entsprechenden Befehlssequenz, bis zu einer gewissen Rekursionstiefe, ersetzt werden sollten. Damit würde man einen Teil der zu testenden Funktionalität in die zu vergleichenden Testfälle übertragen. Gerade wenn man Testfälle auf Basis der gefunden Klone automatisch erzeugen oder ergänzen möchte, könnte dies ein wesentlicher Bestandteil sein, um funktionsfähige Produkte garantieren zu können. Konkret sollen die Vorteile dieser Einschiebung an einem Beispiel, welches in dem Quelltextausschnitt 4.5 auf Seite 22 definiert ist, erläutert werden. Gegeben seien zwei Testfälle, die zu Beginn eine hohe syntaktische Ähnlichkeit aufweisen. In einem Testfall wird eine Liste von Zahlen sortiert, in dem Anderen wird sie zufällig neu angeordnet. In beiden Testfällen wird zugesichert, dass Ein- und Ausgabeliste exakt die gleiche Anzahl an Elementen besitzen. In diesem Fall möchte man, je nach Anwendungsfall, eine Unterscheidung treffen, ob es sich bezüglich dieser Testfälle um Test-Klone handelt. Möchte man den Testfall für die Sortierung um eine Zusicherung bezüglich der Anordnung der Listenelemente ergänzen, könnte man dies nicht synchron auf den zweiten Testfall übertragen. Würde man allerdings in beiden Testfällen die Funktionen, die eine Liste sortieren oder zufällig neu anordnen, einschieben, so könnte man diese Differenzierung treffen. Sollte eine Unterscheidung, wie an dem erläuterten Beispiel, nicht notwendig sein, da keine synchrone Modifikation von Testfällen als Ziel ausgegeben wurde, gilt es auch eine Einschiebung von Methodenrümpfen zu verhindern. Es sollte dem Anwender frei überlassen werden dieses Verhalten zu konfigurieren.

4.3. Metriken

Metriken können schnell zu überprüfende Paarungen ausschließen. Da die Anzahl der notwendigen Vergleiche zwischen Testfällen exponentiell zu deren Anzahl ansteigt, besteht eine Notwendigkeit dieses Zwischenschritts. Die Anzahl der zu vergleichenden Paare lässt sich über den Binominialkoeffizienten berechnen. Ottenstein hat bereits 1976 vorgeschlagen, dass man Quelltexte anhand der Anzahl unterschiedlicher Operatoren, Operanden und deren Vorkommen in der Summe unterscheiden könnte [Ott76].

Donaldson ergänzte diese Arbeit 1981 um weitere, von ihm vorgeschlagene Kriterien [DLS81]. Mit der Weiterentwicklung der Programmiersprachen zieht er ebenfalls die Anzahl folgender Strukturen in Betracht:

1. Summe der vorkommenden Variablen
2. Summe der vorkommenden Unterroutinen
3. Summe der vorkommenden Eingabeaufforderungen
4. Summe der vorkommenden Bedingungsanweisungen
5. Summe der vorkommenden Schleifen
6. Summe der vorkommenden Zuweisungen
7. Summe der vorkommenden Methodenaufrufe
8. Gesamtsumme der Ergebnisse aus 2 bis 7

Eine Metrik, die hier vorgestellt wird, orientiert sich an den Werten aus dieser Auflistung.

4.3.1. Aufzählende Metriken

Als aufzählende Metriken werden hier Methoden bezeichnet, welche die Anzahl von signifikanten Merkmalen zwischen zwei oder mehreren Quelltextauszügen vergleichen. Vergleicht man n Merkmale, so lässt sich aus diesen ein mehrdimensionaler Vektor bilden. Die Kosinus-Ähnlichkeit liefert anschließend ein Maß zur Beurteilung der relativen Ähnlichkeit.

Bei leeren oder kurzen Testfällen kann nicht zwingend sicher gestellt werden, dass kein Nullvektor auftritt. Für diesen Fall sollte eine Sonderbehandlung existieren. Sei es die Einführung eines konstanten Wertes als Teil des Vektors oder auch die automatisierte Zurückweisung von Paaren in denen ein Nullvektor vorkommt.

4.3.2. Hierarchiebezogene Metriken

In einem weiteren Ausschlussverfahren können Kandidaten auf Grund ihrer Hierarchiezuordnung, oder auch deren Typ, akzeptiert oder abgewiesen werden. In *Java* können folgende Kriterien zur Bewertung zwischen zwei Kandidaten gewählt werden:

- Der Paketname
- Die Quelltextdatei des Testfalls
- Die Klasse des Testfalls

Eine Entscheidung kann je nach Übereinstimmung der einzelnen Kriterien getroffen werden.

4.3.3. Merkmalbezogene Metriken

Testfälle können anhand ihrer Signatur und äußeren Merkmale klassifiziert werden. Zu der Signatur von Testfällen gehören in der Programmiersprache *Java* auch Annotationen, die zum Teil Auswirkungen auf das konkrete Verhalten eines Testfalls haben. So ist in der Regel zu unterscheiden, ob ein Testfall nach dem Schema der drei „*A*“s aufgebaut ist oder eine Ausnahme behandelt.

Die hier vorgestellte Metrik soll Testfälle klassifizieren und zu einer Zulassung eines Vergleichs oder deren Ausschlusses führen. Abgesehen von dieser Klassifizierung kann auch der Einsatz von Zusicherungen zur weiteren Unterscheidung in einem Testfall berücksichtigt werden. Dabei kann untersucht werden, ob Testfälle gleiche Zusicherungen enthalten und das Vorkommen dieser in der Reihenfolge äquivalent ist.

In früheren Versionen von *JUnit* wurde, anstatt der Definition einer Ausnahmeerwartung in der Annotation, die sogenannte `fail()` Methode verwendet. Diese erzeugte ihrerseits eine Ausnahme um den Ausführungszweig zu validieren. Diese ist zwar in aktuellen Versionen immer noch anwendbar, aber deren Verwendung ist eher unüblich. Fährt man, wie in dem Quelltextausschnitt 2.2 auf Seite 4, erneut ein Fahrzeug bis der Tank leer ist, könnte die Syntax wie in Beispiel 4.6 formuliert werden. Wird die `fail()` Methode aufgerufen, so gilt der Test als fehlgeschlagen. Die Unterscheidung dieser zwei Ansätze wirft die Frage nach der Notwendigkeit einer Normalisierung auf. Innerhalb eines Projektes können nicht unterschiedliche Versionen von *JUnit*- Testrahmen verwendet werden. Dies hängt mit der Einbindung der Klassen in den Ausführungskontext zusammen, was an dieser Stelle nicht näher erläutert wird. Allerdings ist die direkte Konsequenz, dass dieser Mechanismus nur notwendig wäre, wenn eine Identifikation von Test-Klonen unter mehreren Projekten stattfindet. Eine empirische Auswertung über die Relevanz solch einer Untersuchung soll später

```
@Test
public void testDrive() {
   Car car = new Lamborghini();
   try {
      while (car.getFuel() >= 0) {
         // fahre bis der Tank leer ist
         car.drive();
      }
      fail();
   } catch (OutOfFuelException e) {
      // Test ist erfolgreich
   }
}
```

Quelltextausschnitt 4.6: Testfall mit fail() Funktionalität

erfolgen. Ob eine Identifikation dieser Testfalltypen innerhalb dieser Metrik zu behandeln ist, oder dies eher Aufgabe einer Normalisierung innerhalb des Schrittes der Wandler sein sollte, ist ebenfalls eine wichtige Fragestellung.

4.4. Finale Auswertung

Bis zu diesem Schritt wurden Testfälle identifiziert und gegebenenfalls normalisiert. Außerdem konnten Kombinationen dieser, innerhalb der Verarbeitung der Metriken, bereits von einem Vergleich ausgeschlossen werden. Nun gilt es mit einem geeigneten Verfahren Testfälle auf eine relative Ähnlichkeit zu untersuchen.

Grundsätzlich lassen sich Quelltexte der Programmiersprache *Java* auf mindestens zwei Arten untersuchen. Einerseits kann ein Vergleich auf Basis der Quelltexte stattfinden, andererseits kann der *Java Bytecode* als Grundlage für die Analyse herangezogen werden. Der *Bytecode* ist eine abstrahierte Form des Quelltextes, die von der JIT Kompiliereinheit in Maschinensprache übersetzt wird. Diese Assembler ähnliche Form des Quelltextes ist bei der Auswertung größerer Zusammenhänge entsprechend komplex, da diese aufwendig zusammengesetzt werden müssen. Im Folgenden wird von einem Vergleich rein auf der Quelltextebene gesprochen.

Die metrische Analyse wurde bereits in die sequentielle Verarbeitung aufgenommen. Eine abschließende textbasierte Analyse könnte hier vermutlich keine zufriedenstellenden Ergebnisse liefern, da direkte Plagiate nicht zu erwarten sind. In Betracht kommen eine lexikalische, eine syntaktische und eine semantische Analyse. Aufgrund des hohen Abstraktionsgrades einer lexikalischen Analyse und der Komplexität einer semantischen Analyse, die bezüglich der Annahmen nicht notwendig ist, wird in diesem Verfahren eine syntaktische Analyse vorgeschlagen. Diese hat mehrere Vorteile. Da die Repräsentation der Quelltexte, abgesehen von der Formatierung, in einem abstrakten Syntaxbaum vollständig ist, kann der Inhalt, in jedem der bisher vorgestellten Verarbeitungsschritte des Verfahrens, effizient genutzt werden. Dabei müssen nicht mehrere Abbilder der Quelltexte zur Informationserhaltung verwaltet werden. Eine Metrik kann beispielsweise bestimmte Strukturen in einem abstrakten Syntaxbaum über das Besucher-Entwurfsmuster auf elegante Art und Weise analysieren.

Der Vergleich innerhalb der Daten von abstrakten Syntaxbäumen findet auf Befehlszeilenebene statt. Dabei wird von Bezeichnern abstrahiert und Literale werden generalisiert. Das heißt, die Bezeichnungen von Variablen oder Methodenaufrufen werden in einem Vergleich nicht berücksichtigt. Zeichenketten, Zahlen sowie weitere Literale müssen so weit wie

möglich als gleich angesehen werden. Das `null`-Element kann in diesem Zusammenhang gesondert betrachtet werden, da dieses auch eine besondere Bedeutung besitzt. Kommentare, die durchaus Teil eines abstrakten Syntaxbaums sein können, müssen bei einem Vergleich gänzlich verworfen werden.

Bei einer geforderten prozentualen Ähnlichkeit werden sequentiell folgende Merkmale überprüft.

1. Ist die Anzahl und die Reihenfolge der Zusicherungen, in den beiden zu vergleichenden Testfällen gleich, wird der Vergleich zugelassen.
2. Wird die Mindestmasse der Befehlszeilen beider Testfälle nicht erreicht, so besteht eine 0% Ähnlichkeit.
3. Unterscheiden sich die zu vergleichenden Testfälle in ihrer Masse derart, dass die geforderte Ähnlichkeit nicht erreicht werden kann, wird abgebrochen.
4. Die längste gemeinsame Teilsequenz kann zur Bestimmung der relativen Ähnlichkeit herangezogen werden. Die entsprechende Formel 3.1 wurde bereits angegeben und kann auf Seite 13 nachgeschlagen werden. Wird eine vollständige Übereinstimmung gefunden, so wird die Untersuchung beendet.
5. Entferne die erste Befehlsanweisung aus dem niedriger gewichteten Testfall und führe die sequentielle Überprüfung ab Schritt eins erneut aus.

Je nach Masse der Testfälle kann es erwünscht sein unterschiedliche Strategien bei einem Vergleich einzusetzen. Vorerst soll nur eine Strategie, oder eine konkrete Implementierung, verwendet werden. Allerdings kann davon ausgegangen werden, dass je größer die Masse eines Testfalls ist, desto mehr kann generalisiert und abstrahiert werden, ohne einen negativen Einfluss auf das Gesamtergebnis zu bewirken. Dies ist ebenfalls in der Analyse erkannt worden.

Zusammenfassend lässt sich sagen, dass der Aufbau von Testfällen analysiert wurde, um aus den Erkenntnissen ein geeignetes Verfahren abzuleiten. Dieses untersucht in vier Schritten, die zum Teil Unterschritte enthalten können, Quelltexte. Zuerst werden Testfälle einer bestimmten Form identifiziert, gegebenenfalls modifiziert und paarweise von Metriken auf eine Ähnlichkeit untersucht. Abschließend findet eine syntaktische Analyse mit dem Ziel der Test-Klon Identifikation statt. Die Implementierung dieses Verfahrens wird im nächsten Kapitel vorgestellt.

5. Implementierung

Als Primärziel dieser Ausarbeitung wurde die Klonuntersuchung von Testfällen aus dem *JUnit*-Testrahmen ausgewählt [BGS12]. Eine Test-Klon Identifikation kann für die Programmiersprache *Java* mit dem sogenannten *Java unit test clone investigator*, oder kurz *Jutci*, durchgeführt werden. Dieses Werkzeug wurde im Rahmen dieser Arbeit entworfen und implementiert. Es setzt die Idee des bereits vorgestellten Verfahrens aus dem vorherigen Kapitel um. Das Werkzeug wurde mit Hilfe der Entwicklungsumgebungen *IntelliJ IDEA* und *Eclipse* implementiert [Jet12] [Fou12]. Es stellt dem Anwender eine grafische *Swing*-Oberfläche zur Verfügung, aus der einzelne Quelltextdateien oder Ordner eines Dateisystems, in denen sich solche befinden, ausgewählt werden können. Diese Oberfläche ist in Abbildung 5.1 zu sehen. In dieser sieht man, dass fünf Projekte für eine projektübergreifende Auswertung hinzugefügt wurden. Nach Auswahl der zu vergleichenden Datenmenge, kann die sequentielle Verarbeitung mit dem Ziel der Test-Klon Identifikation gestartet werden.

Die Konfiguration der Anwendung wird in einer `.properties` Datei verwaltet. Auf diese kann über eine, mit dem *Singleton*-Entwurfsmuster implementierte, Schnittstelle zugegriffen werden. In dieser werden alle Einstellungen, insbesondere diejenigen der einzelnen Verfahrensschritte, verwaltet. Ist die Konfiguration nicht vorhanden, so wird eine im Quelltext definierte Grundkonfiguration verwendet.

Den Kern der Implementierung bildet die `JDT/Core` Software-Bibliothek der *Eclipse Foundation* [SJA+12]. Mit deren Hilfe können zu den ausgewählten Quelltextdateien abstrakte Syntaxbäume erzeugt werden. Neben anderen verfügbaren Bibliotheken, welche einen abstrakten Syntaxbaum erzeugen können (zum Beispiel der `Javaparser` [GI10]), zeichnet sich das `JDT/Core` Paket vor allem durch Aktualität und der stetigen Weiterentwicklung aus. In der Regel wurde bisher alle sechs Wochen ein Meilenstein aus der Projektplanung erreicht und die entsprechenden Quelltexte der Bibliothek wurden anschließend veröffentlicht. Aus diesem Umstand kann die Syntax der aktuellen *Java*-Sprachspezifikation von der Bibliothek erkannt und verarbeitet werden.

Es wurde versucht bei der Implementierung die Schichten des Modells, der Präsentation und der Steuerung klar voneinander zu trennen, um eine erleichterte Erweiterbarkeit zu erzielen. Dabei wurde weitestgehend versucht von verwendeten Strukturen aus externen Abhängigkeiten zu abstrahieren, um einen Austausch dieser zu ermöglichen. So konnte,

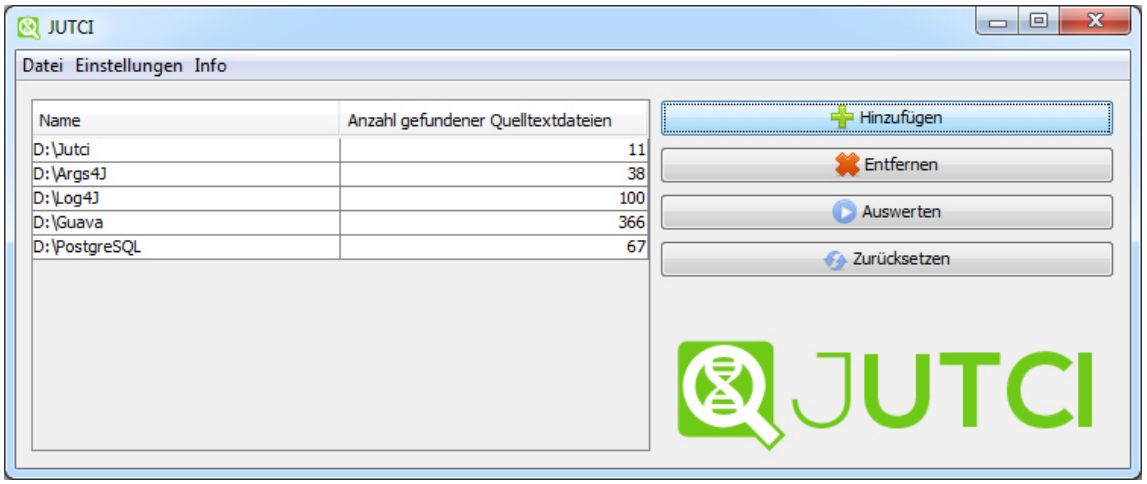

Abbildung 5.1.: Der Hauptbildschirm von Jutci

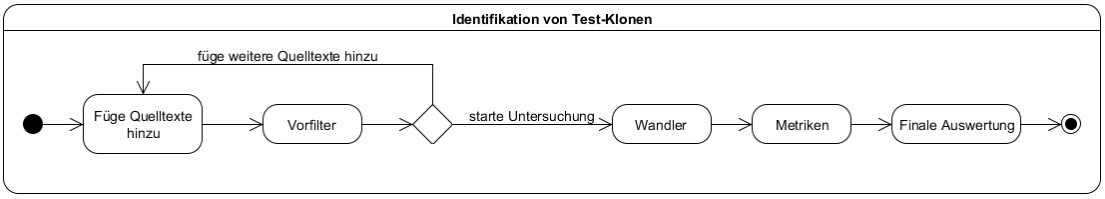

Abbildung 5.2.: UML-Aktivitätsdiagram des implementierten Verfahrens

bei einer Untersuchung mehrerer Industrieprojekte vor Ort, die Anwendung in wenigen Minuten derart erweitert werden, dass sehr viele Submodule automatisiert untersucht werden konnten, ohne die grafische Oberfläche benutzen zu müssen. Diese Erweiterung deckte in etwa 150 Zeilen die Ein- und Ausgabe, sowie die automatisierte Untersuchung in unterschiedlichen Konfigurationen ab.

Innerhalb der bekannten Verfahrensschritte findet weitestgehend eine parallele Verarbeitung der Eingabedaten statt. Diese wurde standardmäßig nur für den Schritt der Wandler, in denen unter anderem Normalisierungen stattfinden können, deaktiviert, um Fehlerfälle, in denen mehrere Wandler identische Passagen editieren, zu vermeiden. Die Verwaltung eines synchronisierten Zugriffs auf die Ressourcen wäre an dieser Stelle zu aufwendig. Jedem implementierten Wandler wurde zudem eine Ausführungspriorität zugeordnet. Ob der Methodenrumpf einer im Testfall aufgerufenen Methode zuerst eingeschoben wird und anschließend Zusicherungen normalisiert werden, oder dies genau in umgekehrter Reihenfolge abläuft, spielt eine große Rolle. In letzterem Fall findet eine Normalisierung der gegebenenfalls vorhandenen, eingeschobenen Zusicherungen nicht statt.

Als Besonderheit sei zu erwähnen, dass die einzelnen Schritte des sequentiellen Verfahrens durch eine abstrakte Klassendefinition vorgegeben sind. Alle Vorfilter, Wandler und Metriken, welche die entsprechenden abstrakten Klassen erweitern, werden über die `Reflections`-Bibliothek eingebunden [Ron12]. Das heißt, dass jede im Klassenpfad vermerkte Implementierung dynamisch in den Kontext der Anwendung eingebunden wird. Kein weiterer Vermerk ist hierzu notwendig.

Im Folgenden werden die Implementierungsdetails der einzelnen Verfahrensschritte, in einer zu dem vorherigen Kapitel analogen Struktur, gründlich erklärt. Das vorgeschlagene, sequentielle Lösungsverfahren ist in Abbildung 5.2 zu sehen.

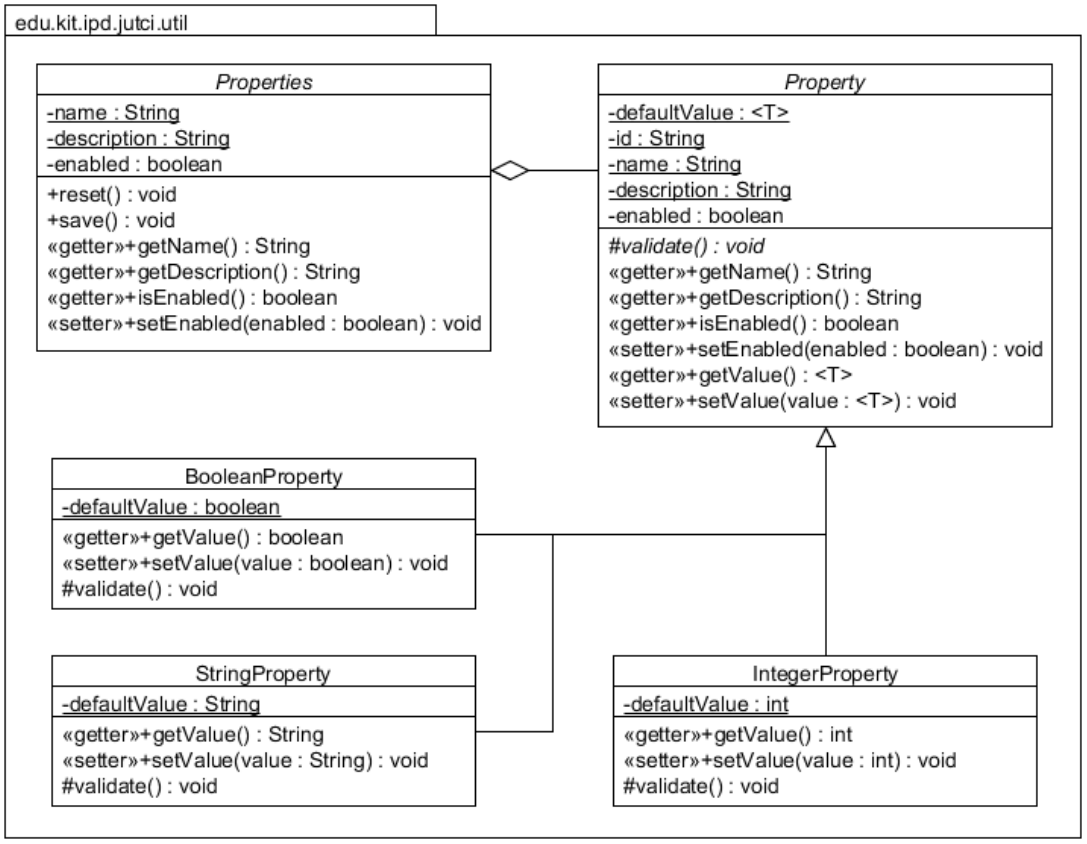

Abbildung 5.3.: Klasse der Konfiguration, die alle Implementierungen der Verfahrensschritte verwenden

Jeder Verfahrensschritt erbt die Funktionalität, der in Abbildung 5.3 dargestellten, abstrakten `Properties`-Klasse. In dieser Klasse werden alle konfigurierbaren, durch die Klasse `Property` repräsentierten, Eigenschaften verwaltet. Zu sehen ist, dass ein ganzer Verfahrensschritt, oder Eigenschaften dessen, mittels des Boolschen Werts `enabled` aktiviert oder deaktiviert werden können.

Jeder Eigenschaft wird ein Standard-Wert zugeordnet, der bis zu einer expliziten Veränderung verwendet wird. Beim Setzen des Wertes einer Eigenschaft, wird die Methode `validate()` aufgerufen, die den zulässigen Wertebereich überprüft.

Die Eigenschaften der Implementierungen der einzelnen Verfahrensschritte werden in den entsprechenden *UML*-Klassendiagrammen nur durch den Platzhalter „. . . “ angedeutet.

Soll eine neue Klasse angelegt werden, benötigt diese einen Namen und eine Beschreibung. Es können dabei Schlüsselbegriffe verwendet werden, die von einer Hilfs-Klasse zur Lokalisierung übersetzt werden. Verfügbare Eigenschaften lassen sich in variabler Anzahl dem Konstruktor der Superklasse übergeben. Von dieser werden sie anschließend verwaltet.

5.1. Vorfilter

In erster Linie sollen in diesem Verarbeitungsschritt Testfälle aus einer Menge von Quelltexten identifiziert werden. Diese können durch prägnante Merkmale unterschieden werden. Ein Filter erhält dabei als Eingabe ein Objekt, welches eine Methode aus dem Quelltext repräsentiert und dieses analysiert. Über einen Boolschen Rückgabewert wird signalisiert,

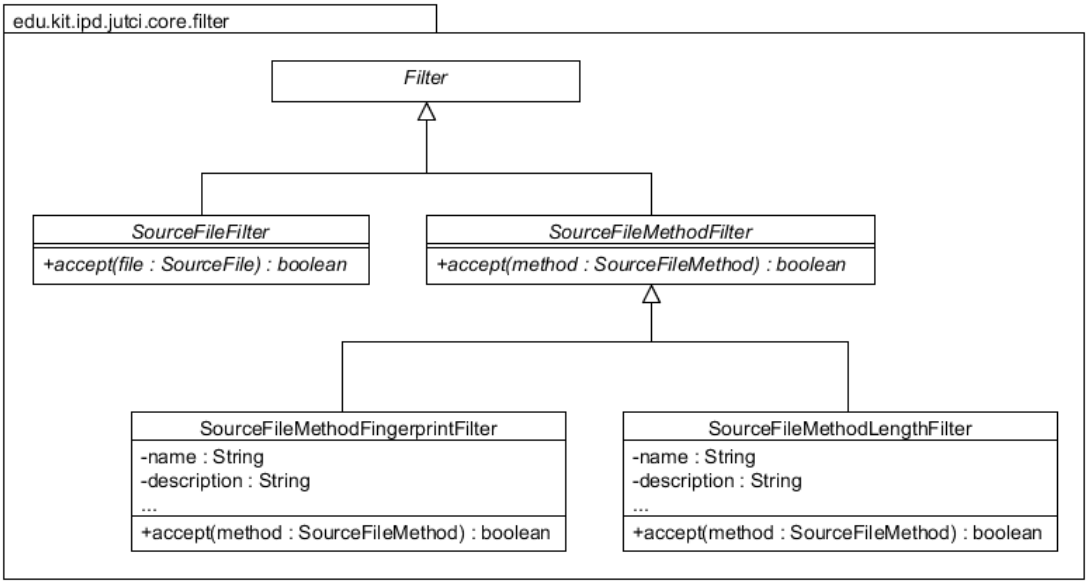

Abbildung 5.4.: UML-Klassendiagramm zu Vorfiltern

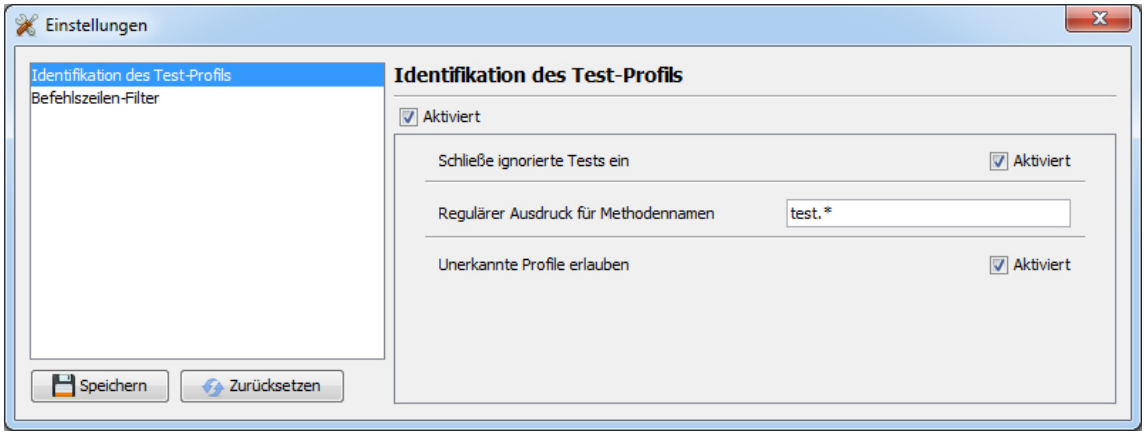

Abbildung 5.5.: Vorfilter-Konfiguration im Jutci

ob eine Methode zur weiteren Analyse oder Verarbeitung freigegeben wird.

Es existieren zwei Kriterien anhand derer Testfälle erkannt werden:

- Es liegt eine `@Test` Annotation vor, die Paketzugehörigkeit dieser kann aufgelöst und einem Testrahmen zugeordnet werden.
- Die Bezeichnung der Testfälle folgt einer bestimmten Konvention anhand derer, diese über einen regulären Ausdruck erkannt werden können.

Diese Merkmale sind hinreichend, um Testfälle von *JUnit 4* und *TestNG 6*, sowie älteren Versionen der Testrahmen, welche die angesprochene Konvention der Bezeichnung einhalten, zu erkennen. Die Erkennung von annotationsbasierten Testrahmen kann zudem anhand einer *XML* Konfiguration definiert werden. Dies wird in dem Quelltextausschnitt 5.1 exemplarisch für den Testrahmen *JUnit 4* gezeigt. Wie man in diesem Beispiel sieht, trägt die Konfiguration den eindeutigen Bezeichner `junit4`, der durch das Attribut `id` festgelegt wird. Jede verfügbare Methodenannotation ist mitsamt der optional verfügbaren

```
<test-framework id="junit4"   name="JUnit␣4">
   <annotations scope="org.junit">
      <annotation id="before" name="Before" stage="BEFORE" />
      <annotation id="test" name="Test" stage="TEST">
         <field id="test.expected" name="expected" />
         <field id="test.timeout" name="timeout" />
      </annotation>
      <annotation id="after" name="After" stage="AFTER" />

      <annotation id="ignore" name="Ignore">
         <field id="ignore.value" name="value" />
      </annotation>
   </annotations>

   <exception annotation="test" ref="test.expected" />
   <ignore annotation="ignore" />
</test-framework>
```

Quelltextausschnitt 5.1: Abstrakte XML Repräsentation von JUnit

Felder definiert. Ist das Attribut `stage` deklariert, so signalisiert dies, dass entsprechend annotierte Methoden innerhalb der Testfall-Ausführung angestoßen werden. Dieses Attribut, sowie die Reihenfolge innerhalb der *XML*-Definition, gibt zudem den Zeitpunkt der Ausführung an. Das Attribut `stage` kann folgende Werte annehmen:

BEFORE Dieses Merkmal gibt an, dass es sich bei der annotierten Methode um einen dem Testfall vorgelagerten Methodenaufruf handelt.

TEST Dieses Merkmal signalisiert, dass mit dieser Annotation die konkreten Testfälle versehen werden.

AFTER Dieses Merkmal gibt an, dass es sich bei der annotierten Methode um einen dem Testfall nachgelagerten Methodenaufruf handelt.

Die Elemente `exception` und `ignore` definieren wie eine Ausnahmeerwartung in Testfällen deklariert wird und ob bestimmte Testfälle bei der Ausführung ignoriert werden sollen. Ob vom Testrahmen ignorierte Testfälle bei der Klon-Erkennung analysiert werden sollen, ist eine Frage der Konfiguration. Weitere Details sind der *XML* Schema Definition im Anhang auf Seite 61 zu entnehmen.

Das Konfigurationsfenster für diesen Filter wird in Abbildung 5.5 dargestellt. Unter diesen Einstellungen wurden Testfälle des *JUnit 3* Testrahmens untersucht.

Ist die Erkennung von annotationsbasierten Testrahmen aktiviert, unerkannte Profile also unzulässig, so wird für jede Methode aus dem abstrakten Syntaxbaum analysiert, ob eine Annotation vorliegt und diese mit dem festgelegten Namen aus dem `name` Attribut der *XML* Konfiguration übereinstimmt, wobei eine volle Referenzauflösung anhand des Paketnamens, der durch das Attribut `scope` festgelegt ist, erfolgt.

Ein weiterer, konfigurierbarer Filter überprüft die Anzahl der Befehlszeilen eines bereits identifizierten Testfalls. Wird bei der Ausführung die eingestellte Mindestanzahl der Befehlszeilen unterschritten, so werden diese Testfälle verworfen. Dies hat den Vorteil, leere und einzeilige, triviale Testfälle verwerfen zu können.

Die *UML*-Klassendiagramme, zu den Vorfiltern, sind in Abbildung 5.4 zu sehen.

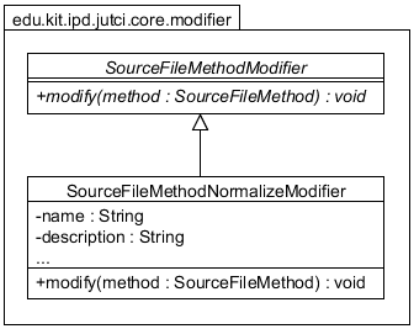

Abbildung 5.6.: UML-Klassendiagramm zu Wandlern

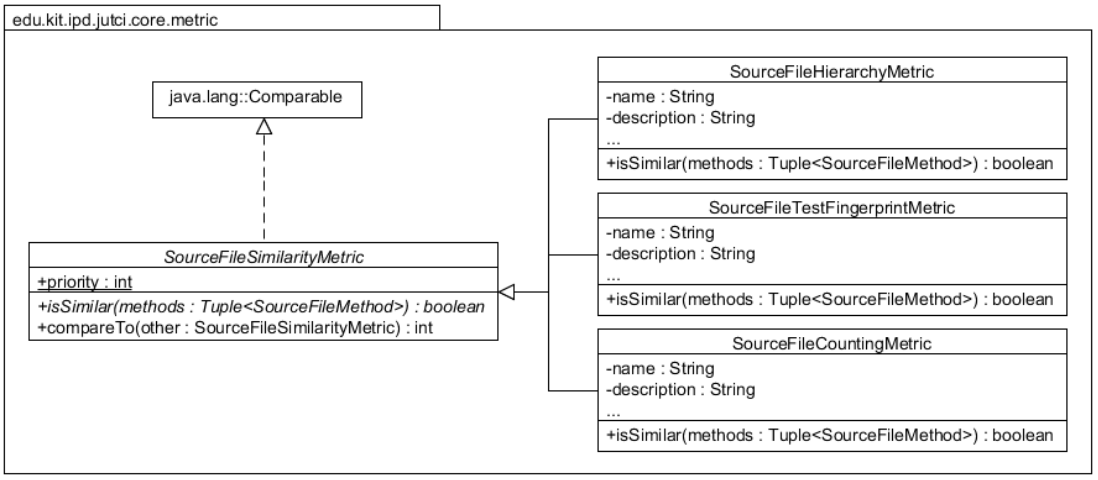

Abbildung 5.7.: UML-Klassendiagramm zu Metriken

5.2. Wandler

Vorerst wurde nur ein Wandler implementiert, der die Zusicherungen unterschiedlicher Testrahmen generalisiert und zur Untersuchung irrelevante Parameter verwirft. Dies ist beispielsweise der Parameter `message`, wie in Abschnitt 4.2.2 auf Seite 20 gezeigt. Die in den Testrahmen *JUnit* und *TestNG* zur Verfügung stehenden Zusicherungen sind, unter Angabe der Indizes relevanter Parameter, in einer *CSV*-Datei definiert worden. Das *UML*-Klassendiagramm der Wandler ist in Abbildung 5.6 zu sehen.

5.3. Metriken

Bei allen Metriken besteht die Eingabe aus jeweils einem Tupel von Testfällen. Dies kann dem *UML*-Klassendiagramm in Abbildung 5.7 entnommen werden. Nach Ausführung aller vorhandenen Metriken für einen Tupel, wird es zur finalen Auswertung zugelassen oder abgewiesen. Dies wird, analog zu den Vorfiltern, durch einen Boolschen Rückgabewert signalisiert. Wird ein Tupel von nur einer Metrik abgewiesen, so wird das Tupel von den anderen Metriken nicht analysiert und das Ergebnis sofort zurückgegeben. Metriken sind deshalb manuell nach ihrer durchschnittlichen Laufzeit priorisiert worden.

Im Rahmen der Implementierung sind drei Metriken entstanden, die sich an denen aus dem Entwurf vorgestellten Metriken orientieren.

5.3.1. Aufzählende Metrik

Diese Metrik kann ab einer konfigurierbaren Mindestanzahl an Befehlszeilen angestoßen werden. Für jeden Testfall wird die Anzahl folgender Merkmale untersucht:

- Anzahl der `for`-Schleifen
- Anzahl der `while`-Schleifen
- Gesamtanzahl der Schleifen
- Anzahl der Bedingungsanweisungen
- Anzahl der Befehlszeilen

Da der Einsatz von Schleifen und Bedingungsanweisungen bei Testfällen, die aus nur drei oder weniger Befehlszeilen bestehen, unüblich ist, besteht das Mindestmaß bevor eine Untersuchung stattfindet, um einen schnellen Ausschluss vor dem Traversieren des abstrakten Syntaxbaumes zu gewährleisten.

Die geforderte Kosinus-Ähnlichkeit ist durch eine prozentuale Angabe zu setzen.

5.3.2. Hierarchiebezogene Metrik

In dieser Metrik können Testfälle, die im gleichen Paket liegen, oder die einer gemeinsamen Quelltextdatei entstammen, ausgeschlossen werden. Um eine projektübergreifende Analyse zu ermöglichen, kann zudem ein Vergleich von Quelltextdateien ausgeschlossen werden, welche zusammengefasst hinzugefügt worden sind.

Der paarweise Vergleich eines Testfalls mit sich selbst wurde zu Testzwecken teilweise aktiviert, sollte aber für gewöhnlich deaktiviert bleiben. Ansonsten existiert zu jedem Testfall ein Test-Klon.

5.3.3. Metrik zur Testklassifizierung

Innerhalb dieser Metrik können Testfälle, die einen unterschiedlichen Testrahmen nutzen, ausgeschlossen werden. Ebenfalls wird hier, falls möglich, unterschieden, ob ein Testfall eine Definition für eine Ausnahmeerwartung trägt oder nicht. Werden diesbezüglich Testfälle identifiziert, die eine unterschiedliche Ausnahme erwarten, können diese ausgeschlossen werden.

Die Ausnahmeerwartung wird rein über die vorliegenden Annotationen untersucht.

5.4. Finale Auswertung

Bei diesem Ausführungsschritt sollen jeweils zwei Testfälle verglichen werden, um zu einem abschließenden Ergebnis zu kommen. Der Vergleich findet, in der konkreten Implementierung, auf Ebene der Befehlszeilen statt. Diese werden in einer vom abstrakten Syntaxbaum getrennten, eigenen, baumähnlichen Struktur verwaltet. Die Elemente dieses Baumes sind in Abbildung 5.8 zu sehen. Die Notwendigkeit dieser Abstraktion entsteht durch die erschwerte Erweiterbarkeit der `JDT/Core` Software-Bibliothek. Dazu gehören unter anderem folgende Gründe:

- Bestimmte Klassen können aufgrund ihrer Deklaration nicht erweitert werden.
- Die Sichtbarkeit von Klassen, Feldern und Methoden, auf die ein Zugriff gewünscht wäre, ist auf das Paket, oder gar gänzlich nur auf den Geltungsbereich der Klasse, eingeschränkt.

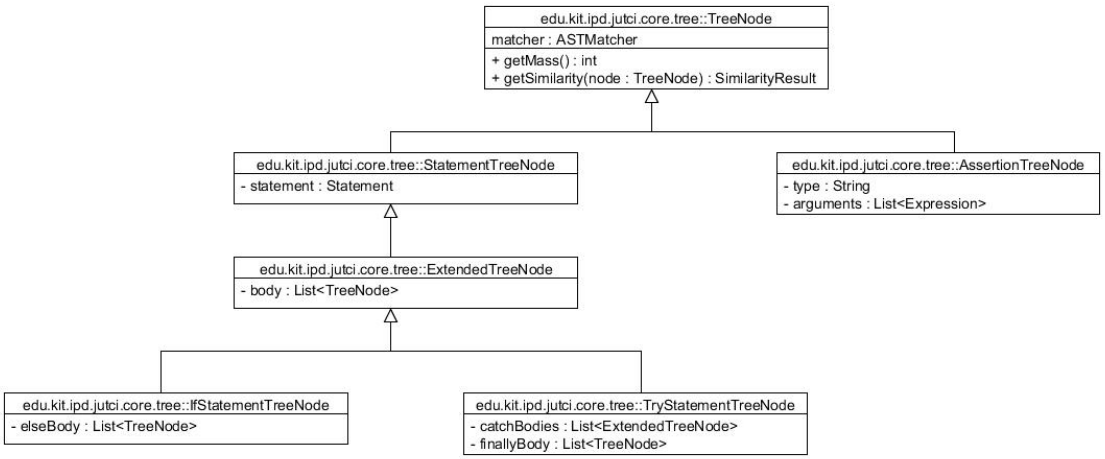

Abbildung 5.8.: Elemente der abstrahierten Teilbäume im abstrakten Syntaxbaum

Zwar existiert beispielsweise eine Komponente, die Elemente des abstrakten Syntaxbaumes vergleicht, allerdings überprüft diese von einer angegebenen Wurzel aus rekursiv den gesamten Teilbaum, also auch große Befehlsblöcke. Dabei wird zudem auf eine Gleichheit der Teilbäume und nicht auf Ähnlichkeit geprüft. Dieses Verhalten musste abstrahiert werden. Hierfür konnte die angesprochene Klasse `ASTMatcher` angepasst werden. Ansonsten wurde der bereits angesprochene Algorithmus bei der Suche nach der längsten gemeinsamen Teilsequenz eingesetzt.

Es werden zwei prozentuale Ähnlichkeits-Schwellen definiert. Zum einen die Schwelle, ab der Test-Klone automatisch erkannt werden sollen und zum anderen eine Schwelle ab der, der Anwender manuell über die Ähnlichkeit von zwei Testfällen entscheiden kann. Gerade für die Evaluation ist dies sehr hilfreich. Auf diese Weise können alle Testfallkombinationen, die eine gewisse Ähnlichkeit aufweisen, in einem vorher definierten Intervall manuell untersucht werden.

Ist die Untersuchung abgeschlossen, kann der Anwender die gefundenen Klone analysieren. Dazu steht der in Abbildung 5.9 dargestellte Dialog bereit. Auf der linken Seite der Abbildung kann der Anwender innerhalb einer hierarchischen Baumstruktur die bidirektional verfügbaren Klonpaare sehen und auswählen. Die Quelltextausschnitte der Auswahl sind auf der rechten Seite zu sehen. Die als Klone identifizierten Testfallpaare lassen sich anhand ihrer vollständigen Signatur in einer *CSV* Datei speichern. Damit können diese Daten in anderen Anwendungen verarbeitet werden.

Die Implementierung wurde genutzt, um das vorgeschlagene Verfahren an ausgewählten Projekten einzusetzen. Die Evaluation basiert auf dem Einsatz des *Jutci*.

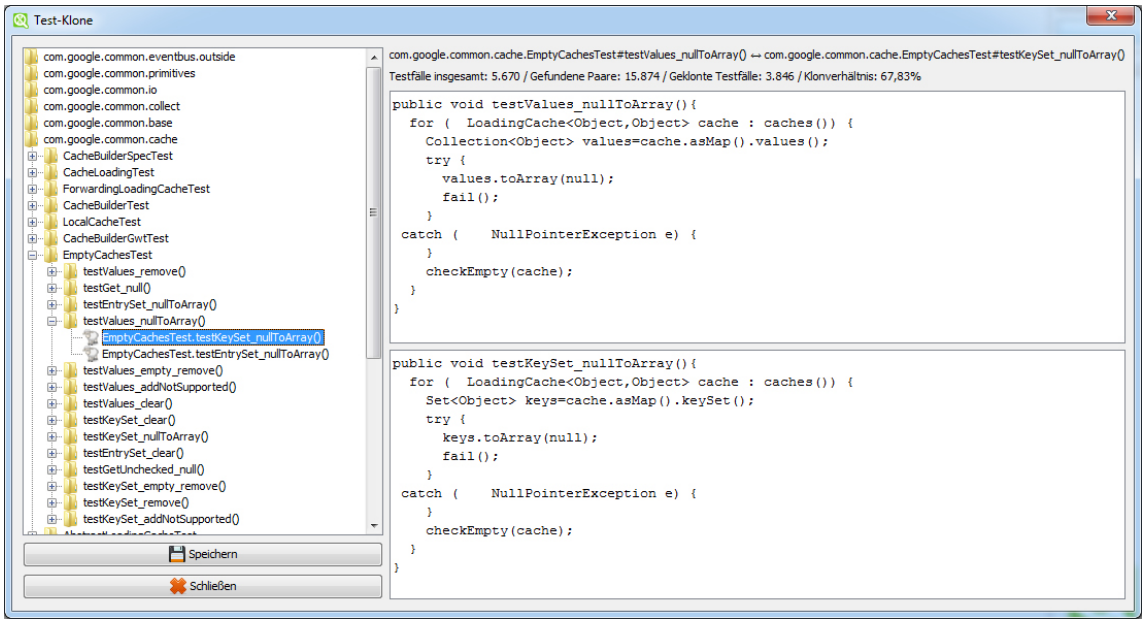

Abbildung 5.9.: Darstellung der Untersuchungsergebnisse im Jutci

6. Evaluation

Im Rahmen der Evaluation wurden mehrere quelloffene-, sowie proprietäre Software Projekte, auf verschiedene Aspekte bei der Identifikation von Test-Klonen untersucht. Eine Untersuchung fand dabei projektintern, nur innerhalb der Quelltexte eines Projekts, sowie projektübergreifend statt. Der konkrete Ablauf der Evaluation wird in Abschnitt 6.1 beschrieben. Es folgt zunächst eine kurze, zusammengefasste Auflistung der Untersuchungsziele:

- Allgemeine Validierung der Funktionalität des vorgestellten Verfahrens bei Ausführung der Implementierung unter verschiedenen Konfigurationen.
- Untersuchung des Anteils von Testfällen, die mindestens einen Klon besitzen, in Relation zu der Gesamtheit aller untersuchten Testfälle.
- Analyse der Grenzfälle, an denen das vorgestellte Verfahren scheitert. Also an denen Test-Klone entweder überhaupt nicht oder fälschlicherweise als solche identifiziert werden.
- Vergleich der erzielten Ergebnisse auf Basis der quelloffenen Bibliotheken zu denen aus proprietären Industrieprojekten.
- Beurteilung der Relevanz der Untersuchungsergebnisse.
- Ermittlung von Kennziffern, wie zum Beispiel der Präzision, Ausbeute und dem F-Maß.

Es wurden Projekte unterschiedlicher Größe aus unterschiedlichen semantischen Umgebungen ausgewählt. Die Projekte sollten dabei einen gewissen Bekanntheitsgrad haben und möglichst von voneinander unabhängigen Entwicklern betreut werden. Damit soll eine Spezialisierung des vorgestellten Verfahrens auf eine bestimmte Problemmenge, wie zum Beispiel einen bestimmten Programmierstil, verhindert werden.

Eine Auflistung der untersuchten Projekte findet sich in Tabelle 6.1 wieder. Abgesehen vom Namen des Projekts repräsentieren die weiteren Spalten dieser Tabelle die Attribute einer *Maven* Abhängigkeit [Fou13a]. Somit wird der jeweilig untersuchte Entwicklungsstand eindeutig referenziert.

Zur Evaluation wurden mehrere Kennziffern ermittelt, mit Hilfe derer sich die Ergebnisse beurteilen lassen.

Name	`groupId`	`artifactId`	`version`
Derby	org.apache.derby	derby	10.8.3.0
Log4J	org.apache.logging.log4j	log4j	2.0-beta3
Commons Email	org.apache.commons	commons-email	1.3
Commons IO	org.apache.commons	commons-io	2.5-SNAPSHOT
Commons Logging	commons-logging	commons-logging	1.1.2-SNAPSHOT
Guava	org.google.guava	guava	14.0-SNAPSHOT
JBoss Web	org.jboss.web	jbossweb	7.7.0-SNAPSHOT
Args4J	args4j	args4j	2.0.23-SNAPSHOT
PostgreSQL JDBC	postgresql	postgresql	9.2-1002

Tabelle 6.1.: Liste untersuchter Softwareprojekte

T_g Testfälle ins*g*esamt

K_m Anzahl der *m*anuell identifizierten Test-Klon-Paare

K_a Anzahl der *a*utomatisch identifizierten Test-Klon-Paare

K_i Anzahl derjenigen Testfälle, zu denen mindestens ein Klon identifiziert werden kann. Diese Testfälle sind *i*ndividuell unterscheidbar

K_q Die Test-Klon*q*uote definiert durch $\frac{K_i}{T_g}$

r_+ Anzahl der richtig positiven Test-Klone

f_+ Anzahl der falsch positiven Test-Klone

f_- Anzahl der falsch negativen Test-Klone

Richtig positive Test-Klone sind dabei diejenigen Testfallpaare, die von dem Werkzeug erkannt wurden und sich ebenfalls bei der manuellen Untersuchung als Klone herausgestellt haben. Falsch positive sind Testfallpaare, die zwar automatisch erkannt worden sind, aber eigentlich keine Klone darstellen. Letztlich sind falsch negative Test-Klone diejenigen, die von dem Werkzeug *Jutci* übergangen worden sind, obwohl es sich bei jenen Testfällen um Klone handelt. Aus den oben genannten Merkmalen lassen sich zudem folgende Kennziffern ableiten:

Präzision $p = \frac{r_+}{r_+ + f_+}$

Ausbeute $a = \frac{r_+}{r_+ + f_-}$

F-Maß $F_1 = 2 \cdot \frac{p \cdot a}{p + a}$

Die Präzision p gibt den Anteil der richtig positiv erkannten Klone an der Gesamtheit der als positiv erkannten Klone an. Sie ist ein geeignetes Maß um festzustellen, wieviele der gefunden Klone auch tatsächlich Klone sind. Die Ausbeute a wiederum gibt den Anteil der richtig positiv klassifizierten Klone an der Gesamtheit der tatsächlichen Klone an. Aus ihr lässt sich ablesen, welcher Anteil der tatsächlichen Klone gefunden wurde. Beide Werte werden hier in Prozent angegeben.

Bei dem F-Maß F_1 handelt es sich um das harmonische Mittel aus Präzision und Ausbeute. Aus diesem Wert, der in dem Intervall $[0, 1]$ liegt, lässt sich die Gesamteffektivität der Auswertung ablesen. Präzision und Ausbeute werden gleichmäßig verrechnet.

Das Untersuchungswerkzeug wurde innerhalb der in Tabelle 6.2 vorgestellten Umgebung ausgeführt. Eine Untersuchung dauerte auf dieser, in der Regel, nur wenige Sekunden. Die im Folgenden vorgestellten Ergebnisse wurden auf dieser Architektur erstellt.

Betriebssystem	Windows 7 Professional, 64-Bit
Laufzeitumgebung	Java SE Runtime Environment (build 1.7.0_017-b11), 64-Bit
Prozessor	Intel i7-2630QM
Arbeitsspeicher	8,00 GB

Tabelle 6.2.: Die Testumgebung

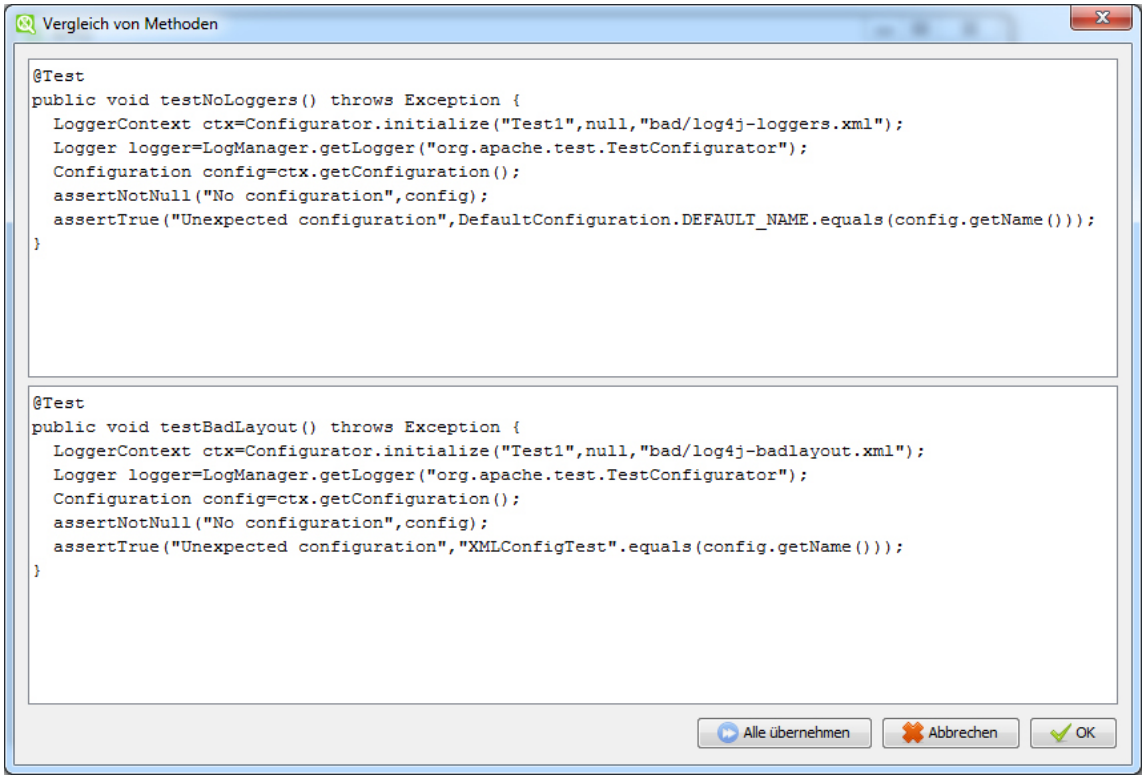

Abbildung 6.1.: Vergleich zweier Testfälle

6.1. Vorgehen

Wie bereits angesprochen, lassen sich in dem Werkzeug *Jutci* zwei Schwellenwerte konfigurieren. Zum einen die Schwelle, oder prozentuale Ähnlichkeit zweier Kandidaten, ab der sie automatisch als Test-Klone markiert werden und zum anderen die Schwelle ab der ein manuelles Eingreifen, oder eine manuelle Überprüfung, erforderlich wird. Solch eine manuelle Überprüfung wird durch ein Dialogfenster wie in Abbildung 6.1 umgesetzt. Die Ähnlichkeit kann hier manuell validiert oder abgewiesen werden. Alternativ können alle, manuell zu überprüfenden Paare übersprungen und als Klone übernommen werden.

Um eine Liste der manuell gefundenen Klone zu finden, wurde die Schwelle zur automatischen Erkennung auf 100% gesetzt. Bei dieser Schwelle werden Klone syntaktisch gleichen Aufbaus identifiziert. Sie müssen die gleiche Anzahl und die gleiche Struktur von Befehlszeilen aufweisen.

Abhängig von der Projektgröße wurde die Grenze, ab der eine manuelle Validierung erforderlich wird, auf zwischen 30% und 50% Ähnlichkeit gesetzt. Die Paare im Ähnlichkeitsintervall $[30\% - 50\%, 100\%)$ wurden manuell geprüft. Die entsprechenden Ergebnisse und verworfenen Paarungen wurden anschließend stichprobenartig kontrolliert, ohne dass eine Abweichung festgestellt werden konnte.

Da im Großteil der vorgestellten Projekte der Testrahmen *JUnit 3* eingesetzt wird, wurden Methoden, die potentielle Testfälle repräsentieren, anhand des regulären Ausdrucks „`test[.]*`“ ausgewählt.

Untersucht wurden alle Paare von Testfällen, auch diejenigen, die innerhalb der gleichen Quelltextdateien definiert wurden. Eine Normalisierung der Zusicherungen innerhalb der Testfälle fand dabei falls möglich statt.

Die Ergebnisse der manuellen Klonsuche wurden im `CSV` Format gespeichert, wobei jede Zeile ein durch ein Semikolon getrenntes Paar von Klonen enthält. Das `diff` Werkzeug wurde anschließend verwendet, um die Unterschiede zwischen den Ergebnissen der Klonsuche unterschiedlicher Konfiguration miteinander zu vergleichen.

Die Schwellenwerte wurden nach der manuellen Klonsuche auf identische Werte gesetzt, um ein manuelles Eingreifen auszuschließen. Sie wurden schrittweise, beginnend bei 50%, erhöht.

Die konkrete Konfiguration für die einzelnen Testrahmen kann im Anhang, im Abschnitt B auf Seite 62, entnommen werden. Eine Abweichung von dieser wird im Folgenden explizit genannt.

6.2. Projektinterne Auswertung

Im Folgenden werden die Ergebnisse für die einzelnen Projekte vorgestellt. Zusätzlich wird auf Grenzfälle, die bei der Untersuchung aufgefallen sind, hingewiesen. Die vorgestellten Quelltextausschnitte orientieren sich dabei an den originalen Auszügen, wurden aber zwecks der Anschaulichkeit zum Teil abgewandelt, gekürzt oder mit zusätzlichen Kommentaren versehen.

Allgemein konnte festgestellt werden, dass die Präzision bei ansteigenden Schwellenwerten der Ähnlichkeit sehr hoch ist, während die Ausbeute dabei absinkt. In Abbildung 6.2 werden Präzision und Ausbeute aus den Untersuchungen aller Schwellenwerte in Relation gestellt. Die Größe der Kreise gibt dabei Aufschluss über die Anzahl der Testfälle, zu denen Test-Klone gefunden werden konnten. Dies soll veranschaulichen, dass durchaus noch Optimierungsbedarf besteht, um die richtige Balance zu finden. Wie solche Optimierungen aussehen können, wird ebenfalls Teil der Diskussion sein.

6.2.1. Log4J

T_g	108
K_m	48
K_i	44
K_q	40, 47%
Testrahmen	*JUnit 4*

Log4J ist ein bekanntes Rahmenwerk zur Protokollierung für die Programmiersprache*Java* und wurde von der *Apache Software Foundation* entwickelt [Fouc].

Die Testfälle innerhalb des Projekts lassen sich zwar mit dem Testrahmen *JUnit 4* ausführen, allerdings ist deutlich erkennbar, dass diese in einer früheren Version implementiert worden sind. Die Testfälle nutzen nicht die aktuelle Schnittstelle. Die Zusicherungen können deshalb zum Teil nicht erkannt werden.

Wie in den Eckdaten des Projekts zu sehen ist, ist das Verhältnis der manuell gefundenen Klone zu der Gesamtanzahl der untersuchten Kombinationen nicht sonderlich hoch. In diesem Fall konnten bei der manuellen Untersuchung viele Test-Klone gefunden werden die

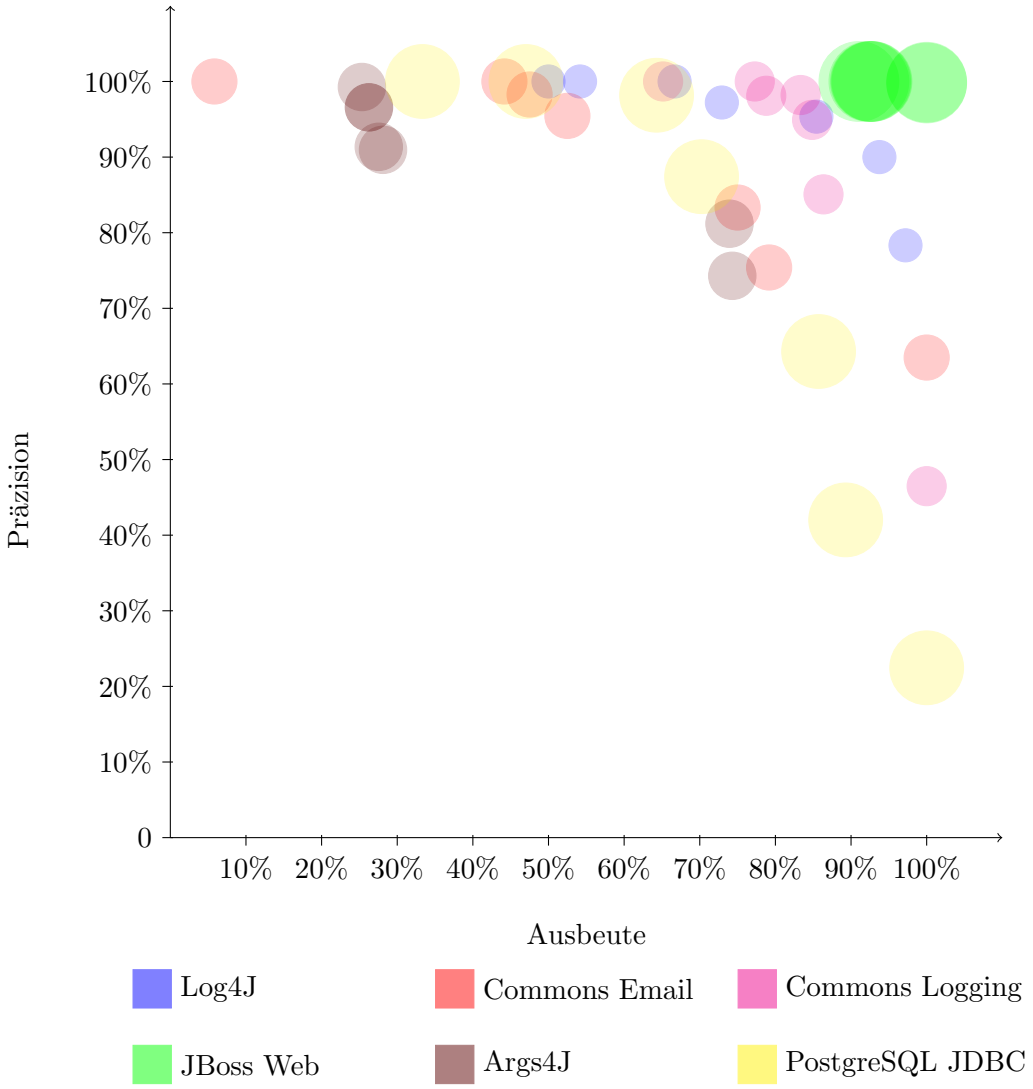

Abbildung 6.2.: Relation von Präzision und Ausbeute in den Untersuchungen

eine ähnliche Logik aufweisen, sich allerdings in den Eingabedaten signifikant unterscheiden. Sind die Testfälle an sich relativ kurz, unterscheiden sich jedoch in der Initialisierung der Eingabedaten erheblich, so ist eine korrekte Identifikation sehr schwer. Dies ist in Quelltextausschnitt 6.1 zu sehen. In dem angesprochenen Beispiel wird die Konfiguration des Testobjekts auf unterschiedliche Art und Weise initialisiert.

Ähnlichkeitsschwelle	K_a	r_+	f_+	f_-	p	a	F_1
50%	60	47	13	1	78,33%	97,92%	.87
60%	50	45	5	3	90%	93,75%	.92
70%	43	41	2	7	95,35%	85,42%	.9
80%	36	35	1	13	97,22%	72,92%	.83
90%	32	32	0	16	100%	66,67%	.8
95%	26	26	0	22	100%	54,17%	.7
100%	24	24	0	24	100%	50%	.67

Tabelle 6.3.: Zusammenfassung der Ergebnisse für Log4J

```
1  @Test
2  public void testFromStream() throws Exception {
3      InputStream is = new FileInputStream(FILE);
4      ConfigurationSource source = new ConfigurationSource(is, FILE);
5      LoggerContext ctx = Configurator.init(null, source);
6      Logger logger = LogManager.getLogger("TestConfigurator");
7      Configuration config = ctx.getConfiguration();
8      // ... Testlogik
9  }
10
11 @Test
12 public void testByName() throws Exception {
13     LoggerContext ctx = Configurator.init("-config", null, null);
14     Logger logger = LogManager.getLogger("TestConfigurator");
15     Configuration config = ctx.getConfiguration();
16     // ... Testlogik
17 }
```

Quelltextausschnitt 6.1: Unterschiedliche Intialisierung bei ähnlicher Logik

6.2.2. Commons Email

T_g	118
K_m	120
K_i	60
K_q	50,48%
Testrahmen	*JUnit 3*

Commons Email ist eine Schnittstelle, deren Ziel es ist das Versenden von elektronischen Briefen zu vereinfachen [Fou13b].

Wie bereits in Abschnitt 6.2.1 zu *Log4J* angesprochen, treten auch hier ähnliche Grenzfälle auf, welche die Ausbeute mit ansteigender Ähnlichkeitsschwelle verringern. Die Ausbeute sinkt drastisch ab einer geforderten Ähnlichkeit von 80%. Hier unterscheiden sich die Eingabedaten, obwohl sie eine klare, semantische Ähnlichkeit aufweisen, syntaktisch derart, dass ein Vergleich fehl schlägt. Ein weiterer, recht häufig auftretender Fall ist in Quelltextausschnitt 6.2 abgebildet. Hier werden die Eingabedaten in einem Fall innerhalb des Testfalls initialisiert, in dem Anderen werden diese aus einer externen Quelle bezogen. Ein Einschub oder eine Referenzauflösung findet dabei nicht statt.

Ein interessantes Beispiel sind die in Quelltextausschnitt 6.3 gezeigten Testfälle. In einem Beispiel werden elektronische Postadressen iterativ zu einer Liste hinzugefügt, auch wenn die Schnittstelle eine äquivalente Funktion besitzt, mit der man eine Sammlung von Adressen setzen kann. In dem Anderen werden diese über solch eine Funktion festgelegt. Die Eingabedaten und die Zusicherungen sind dabei selbst ohne eine Generalisierung nahezu identisch. Der Verdacht, es könne sich dabei um direkt kopierte und leicht adaptierte Testfälle handeln, liegt sehr nahe. Betrachtet man die originalen Auszüge dieser Passagen, findet man sogar eine identische äußere Form vor. Eine Vermutung lautet, dass die zu testende Schnittstelle im Laufe der Zeit erweitert, aber nur ein Testfall angepasst wurde und die Testfälle sich deshalb unterscheiden. Dies wäre höchst interessant, denn eines der formulierten Ziele ist die synchrone Anpassung von Test-Klonen.

Ähnlichkeitsschwelle	K_a	r_+	f_+	f_-	p	a	F_1
50%	189	120	69	0	63,49%	100%	.78
60%	126	95	31	25	75,4%	79,17%	.77
70%	108	90	18	30	83,33%	75%	.79
80%	66	63	3	57	95,45%	52,5%	.68
90%	58	57	1	63	98,28%	47,5%	.64
95%	53	53	0	67	100%	44,17%	.61
100%	7	7	0	113	100%	5,83%	.11

Tabelle 6.4.: Zusammenfassung der Ergebnisse für Commons Email

```
1  @Test
2  public void testGetSetDebug(){
3     boolean[] tests = {true, false};
4     for (int i = 0; i < tests.length; i++) {
5        email.setDebug(tests[i]);
6        assertEquals(tests[i], email.isDebug());
7     }
8  }
9
10 @Test
11 public void testSetSubject(){
12    for (int i = 0; i < testCharsValid.length; i++) {
13       email.setSubject(testCharsValid[i]);
14       assertEquals(testCharsValid[i], email.getSubject());
15    }
16 }
```

Quelltextausschnitt 6.2: Häufig auftretender Fall an dem die Erkennung fehlschlägt

```
@Test
public void testAddReplyTo() {
   List<InternetAddress> expected;
   expected.addAll(EMAIL)

   // Hier der Unterschied
   for (int i=0; i < EMAIL.length; i++) {
      email.addReplyTo(EMAIL[i]);
   }

   assertEquals(expected.size(), email.getReplyTo().size());
   assertEquals(expected, email.getReplyTo());
}

@Test
public void testAddBccArray() {
   List<InternetAddress> expected;
   expected.addAll(EMAIL);

   // Hier der Unterschied
   email.addBcc(EMAIL);

   assertEquals(expected.size(), email.getBccAddresses().size());
   assertEquals(expected, email.getBccAddresses());
}
```

Quelltextausschnitt 6.3: Äquivalente Testfälle die eine unterschiedliche Schnittstellen-Funktionalität nutzen

6.2.3. Commons Logging

T_g	76
K_m	66
K_i	52
K_q	78,78%
Testrahmen	*JUnit 3*

Commons Logging ist ein Rahmenwerk, dass als Brücke zwischen Implementierungen zum Zweck der Protokollierung dient [Fou13c]. Es abstrahiert von einem Rahmenwerk wie zum Beispiel *Log4J* und stellt eine einheitliche Schnittstelle für ähnliche Bibliotheken zur Verfügung.

Auffallend bei der Untersuchung ist, dass viele der gefundenen Test-Klone sehr kurz sind und wieder eine ausgelagerte Testlogik besitzen. Syntaktisch handelt es sich zwar eindeutig um Klone und diese wurden hier auch entsprechend bezeichnet, allerdings würde der Einschub der entsprechenden Logik das Ergebnis aus Tabelle 6.5 stark verändern.

Ebenso grenzwertig ist dadurch auch die Ausbeute. In Quelltextausschnitt 6.4 sieht man zwei sehr ähnliche Testfälle. Allerdings werden diese ab einem Schwellenwert von 51% der Ähnlichkeit nicht mehr als Test-Klone identifiziert, da sie sich in den Parametern der Methodenaufrufe leicht unterscheiden. Um diesen Sonderfall zu behandeln wäre eine automatische Normalisierung äquivalenter Methodenaufrufe notwendig. Dies wurde im Rahmen der Analyse bereits vorgestellt.

Ähnlichkeitsschwelle	K_a	r_+	f_+	f_-	p	a	F_1
50%	142	66	76	0	46,48%	100%	.63
60%	67	57	10	9	85,07%	86,36%	.86
70%	59	56	3	10	94,92%	84,85%	.9
80%	56	55	1	11	98,21%	83,33%	.9
90%	53	52	1	14	98,11%	78,79%	.87
95%	51	51	0	15	100%	77,27%	.87
100%	43	43	0	23	100%	65,15%	.79

Tabelle 6.5.: Zusammenfassung der Ergebnisse für Commons Logging

```
1  @Test
2  public void testDebug1() {
3     Logger log = LogFactory.getLogger(Test.class);
4     log.debug("test")
5  }
6
7  @Test
8  public void testDebug2() {
9     Logger log = LogFactory.getLogger(Test.class.getName());
10    log.debug("test")
11 }
```

Quelltextausschnitt 6.4: Leichte Unterscheidung in den Argumenten

6.2.4. JBoss Web

T_g	131
K_m	982
K_i	106
K_q	80,91%
Testrahmen	*JUnit 3*

JBoss Web ist ein auf *Tomcat* basierender, leichtgewichtiger Web Server [JBo12] [Foua]. Auf diesem Server lassen sich mehrere Technologien, wie *JSP* und *PHP*, beherbergen.

Das Fazit aus der in Tabelle 6.6 vorgestellten Untersuchung fällt relativ kurz aus. Sowohl die Präzision als auch die Ausbeute ist sehr hoch. Das vorgestellte Verfahren funktioniert hier sehr gut. Die Ausbeute sinkt aus dem Grunde, da ein `null`-Wert innerhalb der Argumente eines Funktionsaufrufs nicht äquivalent zu einem konstantem Wert betrachtet wird.

Ähnlichkeitsschwelle	K_a	r_+	f_+	f_-	p	a	F_1
50%	984	982	2	0	99,8%	100%	1
60%	983	982	1	0	99,9%	100%	1
70%	911	911	0	71	100%	92,77%	.96
80%	910	910	0	72	100%	92,67%	.96
90%	909	909	0	73	100%	92,57%	.96
95%	907	907	0	75	100%	92,36%	.96
100%	894	894	0	88	100%	91,04%	.95

Tabelle 6.6.: Zusammenfassung der Ergebnisse für JBoss Web

6.2.5. Args4J

T_g	97
K_m	558
K_i	63
K_q	64,94%
Testrahmen	*JUnit 3*

Args4J ist eine Bibliothek die eingesetzt werden kann um Kommandozeilenargumente bequem einzulesen und objektorientiert auswerten zu können [Kaw13]. Zusätzlich können bestimmte Nachrichten wie eine Benutzeranleitung im *Unix*-System anhand entsprechender Konventionen ausgegeben werden.

Wie Tabelle 6.7 zu entnehmen ist, ist die Präzsion der Ergebnisse relativ hoch und wird deshalb nicht weiter erörtert. Die Ausbeute sinkt allerdings beachtlich, da Zusicherungen nicht ausreichend normalisiert werden. In der aktuellen Implementierung wird eine deutliche Unterscheidung zwischen Zusicherungen, die einen Wahrheitswert überprüfen und denen die eine Gleichheit zwischen zwei Objekten zusichern, gemacht. Allerdings lassen sich beide in der Regel in die jeweils andere Schreibweise überführen, wie es in Quelltextausschnitt 6.5 zu sehen ist. Solch eine Normalisierung würde die Ergebnisse der Untersuchung extrem aufwerten, denn diese äquivalente Schreibweise wird exzessiv eingesetzt, sodass Klone die aus drei Befehlszeilen bestehen und sich nur in der Zusicherung unterscheiden, ab einer Ähnlichkeitsschwelle von 66% nicht als solche erkannt werden.

```
1 Assert.assertTrue("a".equals("a"));
2 Assert.assertEquals("a", "a");
```

Quelltextausschnitt 6.5: Äquivalente Zusicherungen

Ähnlichkeitsschwelle	K_a	r_+	f_+	f_-	p	a	F_1
50%	537	399	138	138	74,3%	74,3%	.74
60%	489	397	92	140	81,19%	73,93%	.77
70%	166	151	15	386	90,96%	28,12%	.43
80%	162	148	14	389	91,36%	27,56%	.42
90%	146	141	5	396	96,58%	26,26%	.41
95%	146	141	5	396	96,58%	26,26%	.41
100%	137	136	1	401	99,27%	25,33%	.4

Tabelle 6.7.: Zusammenfassung der Ergebnisse für Args4J

6.2.6. PostgreSQL JDBC

T_g	440
K_m	168
K_i	98
K_q	64,94%
Testrahmen	*JUnit 3*

PostgreSQL ist ein weit verbreitetes relationales Datenbanksystem. Für die Programmiersprache *Java* existiert ein so genannter *JDBC*-Treiber, über den beispielsweise Datenbanken über eine definierte Schnittstelle angesprochen und manipuliert werden können [HCJJ12] [Ora]. Die Testfälle für diesen Treiber wurden hier untersucht.

Bei der Identifikation von Test-Klonen, deren Ergebnisse in Tabelle 6.8 dargestellt sind, konnte festgestellt werden, dass die Testfälle wesentlich ausführlicher als in den bisherig vorgestellten Projekten sind. Im Gegensatz zu diesen enthalten die Testfälle hier auch den Großteil der Logik und diese wurde selten ausgelagert. Deshalb war zu erwarten, dass die Ausbeute bei steigender Ähnlichkeitsschwelle deutlich

Ähnlichkeitsschwelle	K_a	r_+	f_+	f_-	p	a	F_1
50%	748	168	580	0	22,46%	100%	.37
60%	357	150	207	18	42,02%	89,29%	.57
70%	224	144	80	24	64,29%	85,71%	.73
80%	135	118	17	50	87,41%	70,24%	.78
90%	110	108	2	60	98,18%	64,29%	.78
95%	79	79	0	89	100%	47,02%	.64
100%	56	56	0	112	100%	33,33%	.5

Tabelle 6.8.: Zusammenfassung der Ergebnisse für PostgreSQL JDBC

abfallen wird, denn hier ist eine größere Streuung unter den Eingabedaten, die in den Testfällen enthalten sind, erkennbar.

6.2.7. Derby und Guava

T_g	3415
Ø Untersuchungszeit	45 Sekunden
Testrahmen	*JUnit 3*

Tabelle 6.9.: Merkmale im Derby-Projekt

Die untersuchten Bibliotheken *Derby* und *Guava* weisen eine hohe Anzahl von Test-Klonen auf [Foub] [Goo13]. *Apache Derby* ist eine quelloffene, von der *Apache Software Foundation* entwickelte, relationale Datenbank. Dank der eingesetzten Implementierungssprache *Java* ist diese auf vielen Plattformen verwendbar.

In *Guava* sind von *Google* gesammelte Bibliotheken enthalten, die in *Java*-Projekten dieses Unternehmens eingesetzt werden. Diese Bibliotheken enthalten zur *Java*-Klassenbibliothek ergänzende Datenstrukturen, wie beispielsweise Listen mit einer ergänzten Funktionalität, und stellen für viele allgemeine Probleme nützliche Funktionen zur Verfügung. Bei diesen Projekten fällt die manuelle Untersuchung sehr schwer, denn bereits im Vorfeld konnte beobachtet werden, dass die benötigte Zeit der manuellen Beurteilung über die Ähnlichkeit zweier Testfälle bei mehr als 1.000 Kombinationen sehr lange dauert und die Aufmerksamkeit dabei nachlässt. Hier ist die Differenz aus denjenigen Klonen, die bei einer 50% Ähnlichkeit und derjenigen mit einer 100% Ähnlichkeit, also der Menge aus der die Paarungen der Testfälle normalerweise manuell validiert wurden, besonders hoch. Auf eine manuelle Untersuchung die alle Kombinationen abdeckt, wurde deshalb verzichtet und stattdessen nur eine stichprobenartige Untersuchung durchgeführt. Die Ergebnisse wurden anschließend jeweils auf die Gesamtmenge hochgerechnet. Die Hochrechnung für die *Derby* Bibliothek ist in Tabelle 6.11 abgebildet. Die für *Guava* findet sich in Tabelle 6.12 wieder.

Bei beiden Ergebnissen fällt auf, dass die Präzision im Vergleich zu den bisherigen Ergebnissen relativ niedrig und auch die Ausbeute ungewohnt schlecht ist. Zunächst lässt sich festhalten, dass die vorgestellten Untersuchungen nur bei einer geforderten Ähnlichkeit von 50% und 100% stattfanden. Die Diskrepanz beider Ergebnisse ist somit relativ hoch, anstatt fließend überzugehen. Bei einer geforderten Ähnlichkeit von 100% muss die Anzahl und die Reihenfolge der als äquivalent eingestuften Befehlszeilen gleich sein. Bei 50% ist die Beurteilung dementsprechend abgeschwächt. Selbst bei einer 80% Ähnlichkeit verbessert sich allein das Verhältnis aus den gefundenen Test-Klonen nur marginal.

Im Folgenden wird speziell der niedrige Präzisionswert für die Untersuchung der Test-Klone innerhalb der *Derby* Bibliothek erklärt. Hier wurden etwa 55% der gefundenen Testfälle falsch beurteilt. Es konnte festgestellt werden, dass die Testfälle, die als Klone

bezeichnet wurden syntaktisch durchaus Klone sind. Allerdings fielen bei der semantischen Beurteilung drei Hauptgründe zur falschen Beurteilung auf:

- Die gefundenen Testfälle der Klonpaare bestehen aus wenigen Befehlszeilen, die zum größten Teil aus Aufrufen von Hilfsmethoden bestehen. Diese Hilfsmethoden enthalten die Testlogik. Die Zusicherungen sind nicht Teil der als Testfall deklarierten Methoden.
- Die Testfälle folgen einer simplen Struktur und die Wahrscheinlichkeit syntaktische Klone zu finden ist deshalb sehr hoch. Teilweise könnte hier die semantische Bedeutung der Testfälle in Zeichenketten und Bezeichnern von Variablen und Methoden erschlossen werden. Allgemein ist beispielsweise ein Testfall, der die Persistierung von Daten überprüft im Vergleich zu einem Testfall in der diese verworfen werden soll unterschiedlich zu behandeln. In diesen Testfällen konnte aber kein Unterschied, abgesehen von Schlüsselbegriffen wie `commit` und `rollback`, festgestellt werden.
- Zusicherungen können in der vorliegenden Klassenhierarchie zum Teil nicht erkannt werden und werden deshalb als gewöhnliche Methodenaufrufe behandelt. Dies hängt mit der erschwerten Referenzauflösung zusammen.

T_g	5670
Ø Untersuchungszeit	1 Minute
Testrahmen	*JUnit 3*

Tabelle 6.10.: Merkmale im Guava-Projekt

Innerhalb der *Guava* Bibliothek trat dabei das zuerst genannte Problem ebenfalls auf, allerdings in einem wesentlich geringeren Maß.

Die niedrige Ausbeute kommt durch eine Kombination aus allen bisher genannten Problemen, welche sich auf die Ausbeute auswirken, zustande. Da hier die Menge der Testfälle um einiges höher ist, ist auch die Wahrscheinlichkeit, dass jene Probleme auftreten, höher. Auch hier sollen die Gründe für einen derartig extremen Wert innerhalb der Untersuchung der *Guava* Bibliothek aufgezählt werden:

- Die Eingabedaten eines Testfalls sind teilweise extern ausgelagert. Hier müsste eine Normalisierung durch einen Einschub stattfinden.
- Ähnliche Probleme wurden auf unterschiedliche Art und Weise gelöst. Der syntaktische Vergleich schlägt deshalb fehl. Möglicherweise ist dies mit der Anzahl der an diesem Projekt beteiligten Personen, die aktuell mit 25 beziffert ist, zu erklären, denn es ist kein durchgehender Stil in der Formulierung der Testfälle zu erkennen. Vermutet wird hier, dass die Entwicklung, wie bereits in der Projektbeschreibung erwähnt, verteilt stattfand und mehr ein Zusammenschluss verschiedener Teilbereiche, als eine zusammenführende Lösung ist.

Ähnlichkeitsschwelle	K_a	r_+	f_+	f_-	p	a	F_1
50%	19.121	8.400	10.721	0	43,93%	100%	.61
100%	6.105	3.400	2.705	5.000	55,69%	40,48%	.46

Tabelle 6.11.: Zusammenfassung der Ergebnisse für Derby

Ähnlichkeitsschwelle	K_a	r_+	f_+	f_-	p	a	F_1
50%	63.471	30.800	32.671	0	48,52%	100%	.65
100%	9.088	7.538	1.550	23.262	82,29%	24,18%	.37

Tabelle 6.12.: Zusammenfassung der Ergebnisse für Guava

6.3. Projektübergreifende Auswertung

Die angegebenen Projekte wurden auf eine übergreifende Ähnlichkeit untersucht. Dazu wurde eine Auswertung angestoßen, in der ein Vergleich zwischen Quelltextdateien des gleichen Projekts ausgeschlossen wurde. Insgesamt wurden dabei 10.135 Testfälle untersucht, was 51.354.045 Vergleichen entspricht. In der vorgestellten Testumgebung belegten die zur Auswertung notwendigen Datenstrukturen und der Applikationskontext bis zu $3,1GB$ und diese dauerte 2 Minuten und 40 Sekunden.

Die Auswertung ergab, dass unter 446 involvierten Testfällen 3.861 Klonpaare gefunden werden konnten. Dieses entspricht einem Verhältnis von $4,4\%$ der Testfälle, zu denen mindestens ein Test-Klon identifiziert werden kann.

Die gefundenen Test-Klone bestanden unter den Projekten aus maximal drei Befehlsanweisungen, wobei die meisten involvierten Testfälle gar aus nur einer Befehlszeile bestanden. Diese hatten zudem keinen erkennbaren, semantischen Zusammenhang, wobei es zumindest bei den Bibliotheken zur Protokollierung zu erwarten wäre. Die umgekehrte Schlussfolgerung ist, dass unter kurzen Testfällen die Wahrscheinlichkeit größer ist Test-Klone zu finden. Wenn man die Paarungen genauer analysiert stellt man fest, dass die Testfälle gar einen sehr primitiven Aufbau haben. Die meisten bestehen ausschließlich aus Methodenaufrufen mit keinem und bis zu zwei Parametern. Bei Testfällen, die aus mehr als einer Befehlszeile bestehen, waren dies zudem meist dieselben Methodenaufrufe mit variierenden, übergebenenen Parametern.

Es ist anzunehmen, dass auch bei einer größeren Menge von untersuchten Projekten eine ähnliche Bilanz entsteht. Die projektübergreifende Auswertung und Identifikation von Test-Klonen liefert scheinbar keine sinnvoll verwertbaren Ergebnisse.

6.4. Auswertung von Industrieprojekten

Im Rahmen der Evaluation hat sich glücklicherweise die Möglichkeit ergeben, vier Industrieprojekte zu untersuchen, die den Testrahmen *JUnit 4* nutzen. Die Ergebnisse liegen anonymisiert vor und es dürfen keine Grenzfälle aufgezeigt werden. Es lässt sich jedoch erkennen, dass Test-Klone nicht nur innerhalb von öffentlichen Bibliotheken zu finden sind. Es könnte durchaus auch ein wirtschaftlicher Nutzen in Folgeuntersuchungen liegen.

Unter 13.429 Testfällen, was 90.162.306 Vergleichen entspricht, konnten die aus der Tabelle 6.13 zu entnehmenden Daten erstellt werden. Zwischen $45,29\%$ und $87,77\%$ der Testfälle besitzen - unter der Annahme, dass eine hohe Präzision erreicht wurde - dabei mindestens einen Test-Klon. Dieses Ergebnis variiert dabei stark, wenn man die Untersuchung innerhalb der Einzelprojekte betrachtet. Diese sind im Anhang auf Seite 63 zu entnehmen.

Auffällig in der Untersuchung ist der Rückgang der gefundenen Klonpaare um etwa 70% zwischen den Ähnlichkeitsschwellen von 50% und 60%, weshalb dies näher analysiert wurde. In zwei weiteren Ausführungen innerhalb dieses Intervalls wurden dazu jeweils nur Testfälle einer Mindestlänge von drei oder mehr und vier oder mehr Befehlszeilen ausgewertet. Diese Ergebnisse sind den Tabellen 6.14 und 6.15 zu entnehmen. Auch hier ist ein signifikanter Rückgang von etwa 50% festzustellen. Die Vermutung, dass innerhalb dieses Intervalls vor allem relativ kleine Testfälle nicht mehr als Klone identifiziert werden, trifft deshalb nicht zu und kann nicht der Hauptgrund für einen derartig starken Rückgang sein. Eine Erklärung ist zum Beispiel, dass parameterlose Methodenaufrufe häufig auftreten, wobei die restliche Syntax stark variiert. So können Fälle, in denen aus diesen relativ langen gemeinsame Teilsequenzen gebildet werden, vermehrt auftreten.

Ähnlichkeitsschwelle	K_a	K_i	K_q
50%	329.855	11.786	$87,77\%$
60%	97.904	10.236	$76,22\%$
70%	68.054	8.755	$65,19\%$
80%	62.189	7.852	$58,47\%$
90%	57.587	6.675	$49,71\%$
95%	56.788	6.273	$46,71\%$
100%	66.536	6.082	$45,29\%$

Tabelle 6.13.: Zusammenfassung der Ergebnisse aus Industrieprojekten

Ähnlichkeitsschwelle	K_a	K_i	K_q
50%	72.093	7.678	$88,08\%$
60%	36.658	6.967	$79,92\%$

Tabelle 6.14.: Ergebnisse der Untersuchung von Testfällen aus Industrieprojekten mit mindestens drei Befehlszeilen

Ähnlichkeitsschwelle	K_a	K_i	K_q
50%	39.673	5.604	$86,68\%$
60%	20.424	4.940	$76,41\%$

Tabelle 6.15.: Ergebnisse der Untersuchung von Testfällen aus Industrieprojekten mit mindestens vier Befehlszeilen

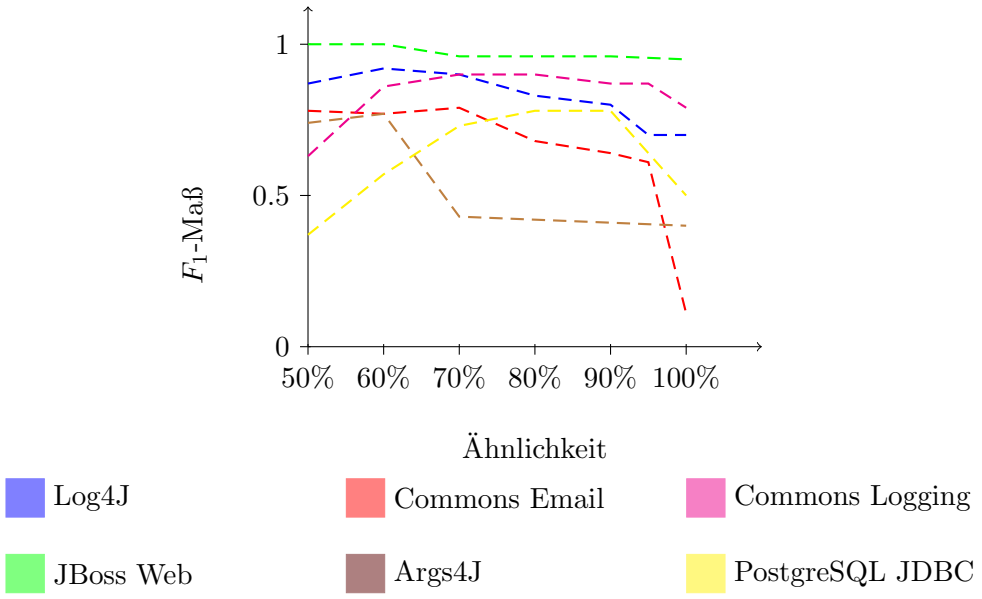

Abbildung 6.3.: F_1-Maß der Untersuchungen

6.5. Fazit

Für die projektinternen Untersuchungen ist die Entwicklung des F_1-Maßes in Abbildung 6.3 zu sehen. Das durchschnittliche F_1-Maß wurde, gewichtet mit der Klonanzahl K_i aus jedem Projekt k, für alle Ähnlichkeitsschwellenwerte ermittelt. Dabei wurde das Maximum von $0,81$ bei einer Ähnlichkeitsschwelle von 60% erreicht. Die Formel zur Berechnung lautet wie folgt:

$$\frac{\sum_{k=0}^{n} K_{ik} \cdot F_{1k}}{\sum_{k=0}^{n} K_{ik}} \tag{6.1}$$

Dies entspricht, äquivalent ermittelt, einer Präzision von $87,01\%$ und einer Ausbeute von $85,35\%$. Mit steigender Projektgröße, wie bei den Bibliotheken *Derby* und *Guava*, oder auch der projektübergreifenden Untersuchung, sinken die Werte dieser Kennziffern. Zwar unterscheiden sich die Testfälle im Aufbau deutlicher, allerdings finden vermehrt einfache, syntaktische Überschneidungen statt, die das Ergebnis negativ beeinflussen. Hier muss das Verfahren derart erweitert werden, sodass die semantische Beurteilung eine größere Rolle einnimmt. Eine Zusammenfassung der Grenzfälle, sowie Vorschläge zur Optimierung, werden im nächsten Kapitel gegeben.

7. Zusammenfassung und Ausblick

Das hier vorgestellte Verfahren dient dazu Klone von Testfällen zu identifizieren. Dabei werden sequentiell vier Schritte auf eine initiale Quelltextmenge ausgeführt, um zu einem Ergebnis zu kommen. Der Quelltext wird auf bestehende Testfälle analysiert und diese werden zum Teil normalisiert oder in eine andere Darstellung überführt. Anschließend wird die Menge der Vergleichskandidaten durch bestimmte Metriken eingeschränkt, bevor die abschließende syntaktische Analyse auf Basis eines abstrakten Syntaxbaumes stattfindet.

Anhand der enstandenen Evaluationsergebnisse kann man sehen, dass die Implementierung dieses Verfahrens bereits eine hohe Präzision bei der Klonfindung erreicht. Dieser Wert hängt jedoch stark von der Klassifikation von Test-Klonen ab. Hier spielt die subjektive Beurteilung, je nachdem ob Form oder Inhalt stärker gewichtet werden, eine ausgeprägte Rolle. Das vordergründige Ziel dieser Arbeit war es Test-Klone automatisiert finden zu können, ohne eine explizite Wertung in eine bestimmte Richtung vorzunehmen. Die verwendete, syntaktische Analyse deutet jedoch bereits darauf hin, dass eine Beurteilung vorwiegend auf Basis der Form der Testfälle stattfindet. Dies steht in Einklang zu den aus dem Kapitel 4 definierten Annahmen über das Auftreten von Test-Klonen, in dem es heißt, dass Test-Klone vorwiegend dann auftreten, wenn sie direkt kopiert und angepasst werden. Der Vergleich der Ergebnisse aus den Untersuchungen, die projektintern und projektübergreifend stattfanden, bestätigen diese Annahme. Je nach Konfiguration der Ähnlichkeitsschwellenwerte, ab denen Testfälle als Klone klassifiziert werden, können unterschiedliche Typen von Klonen identifiziert werden. Ist etwa eine Ähnlichkeit von 70% gefordert, findet man noch viele, sich ausschließlich durch ihre Eingabedaten unterscheidende Klone. Wird hingegen eine Ähnlichkeit von 100% gefordert so spielt die syntaktische Beurteilung eine übergeordnete Rolle. Diese Differenzierung kann je nach formuliertem Ziel einer Folgeuntersuchung unterschiedlich ausfallen.

Es gibt Grenzfälle, in denen hier klassifizierte Klone nicht oder fälschlicherweise erkannt werden. Die Gründe hierfür werden im Folgenden skizziert aufgelistet.

- Die Testlogik ist in Zeichenketten oder bestimmten Eingabewerten verborgen, deren Analyse nicht möglich ist ohne den verarbeitenden Quelltext auf komplizierte Art und Weise dynamisch zu analysieren.
- Besteht ein Testfall aus einer großen Anzahl von Befehlszeilen und wird ein Klon gefunden, so ist die Wahrscheinlichkeit sehr hoch, dass diese eine semantische Ver-

knüpfung haben. Bei kurzen Testfällen wurde festgestellt, dass diese abgesehen von ihrer syntaktischen Änhlichkeit teilweise überhaupt keinen Zusammenhang besitzen. Diese Klone sind, im Sinne der formulierten Anwendungsfälle, eher unerwünscht, auch wenn sie ganz klar als syntaktische Klone zu bezeichnen sind.

- Möglichkeiten der Vererbung innerhalb von Test-Klassen wurden nicht berücksichtigt, was beispielsweise zu Problemen bei der Erkennung und Normalisierung von Zusicherungen geführt hat.
- Das Einschieben der Methodenrümpfe aufgerufener Methoden innerhalb von Testfällen wurde nicht realisiert, da es auch nicht erforderlich ist alle Abhängigkeiten von Testfällen in den Anwendungskontext einzubinden. Das Einschieben hätte einen positiven Effekt auf die semantische Beurteilung.
- Die Reihenfolge oder Sortierung der Befehlsanweisungen innerhalb von Testfällen spielt eine große Rolle bei der Suche nach der längsten gemeinsamen Teilsequenz. Eine semantische Analyse würde diesem Problem entgegenwirken.

Teile dieser Probleme sind eng mit der Auswahl der Bibliothek zur Ableitung von abstrakten Syntaxbäumen verknüpft. Die erschwerte Möglichkeit der Anpassung macht eine Referenzauflösung von Methodenaufrufen oder Variablen extrem schwierig. Ein erstes Ziel wäre es diese, aufgrund mangelnder Alternativen, durch eine eigene, für eine Teilmenge der Probleme geeignete, Implementierung zu ersetzen. Alternativ steht dem eine Einbindung des Verfahrens in die Entwicklungsumgebung *Eclipse* gegenüber. Hier könnte man eine erweiterte Funktionalität des Rahmenwerks nutzen und so gewisse Probleme anders lösen, auch wenn die Abstraktion der Grunddaten weiterhin erforderlich und die zu verwaltende Datenmenge weiterhin extrem hoch wäre. Es folgt eine Auflistung der erweiterten Möglichkeiten durch solch eine Anpassung:

- Das Einschieben externer Sequenzen wird ohne größeren Aufwand möglich sein, was bei einer Vielzahl von falsch positiv erkannten Test-Klonen Abhilfe schaffen würde. Dies gilt für diejenigen Fälle in denen Eingabedaten außerhalb des Testfalls initialisiert, oder die Testlogik ausgelagert vorgefunden wurde.
- Vor allem Zusicherungen, aber auch die Referenzen weiterer, wichtiger Funktionalitäten von Testrahmen könnten schnell und einfach erschlossen werden. Das Problem der komplexen Vererbungshierarchie würde sich, zumindest in der aktuell auftretenden Form, nicht mehr so drastisch auf das Ergebnis auswirken.

Es werden jedoch noch weitere Verbesserungen vorgeschlagen. So könnten die Auswertungen von Bezeichnungen in die semantische Analysen miteinbezogen werden. Ein mit dem Wort `set` beginnender Methodenaufruf könnte so als ein für das Testobjekt Wert setzender Aufruf interpretiert werden. Diese Information kann man nutzen, um zum einen das Testobjekt zu identifizieren, zum anderen um es von der Testlogik ab zu kapseln. Welche Funktionalität getestet wird, kann ebenfalls teilweise aus den Bezeichnung extrahiert werden. Mittlerweile ist der Quelltextauschnitt 2.1, in dem ein *Ferrari* beschleunigt wird, bestens bekannt. Die Testfallbezeichnung `testAcceleration` folgt der Konvention Testfälle mit dem Präfix `test` auszustatten. Das Wort `Acceleration` signalisiert in diesem Fall, dass der Methodenaufruf `accelerate` etwas mit der zu testenden Funktionalität zu tun haben könnte, was in diesem Fall tatsächlich zutrifft. Generell könnte nach einem Methodenaufruf mit der kleinsten Levenshtein-Distanz gesucht werden, um unter Umständen die zu testende Funktionalität zu identifizieren. Ziel ist es die Testfallstruktur der drei „*A*“s zu erkennen, um die Bestandteile dieser Struktur einzeln auswerten zu können.

Um auch die Testfälle, in der eine Ausnahme auftritt, klassifizieren zu können, kann die Signatur der aufgerufenen Methoden untersucht werden. Wird dabei festgestellt, dass eine

Ausnahme die bei einem Methodenaufruf auftreten kann, erwartet wird, so ist die Wahrscheinlichkeit hoch, dass dies diejenige ist in der eine Ausnahme auftreten soll. Auch hier wird die zu testende Funktionalität erschlossen.

Konzeptionell müssen hierzu auch die zu testenden Objekte in die Untersuchung einbezogen werden, was in der aktuellen Implementierung keine Erforderlichkeit darstellt. Diese Entscheidung wurde bewusst getroffen um eine zügige Untersuchung zu ermöglichen, ohne alle Abhängigkeiten einbinden zu müssen. Ein automatisches Erschließen dieser Abhängigkeit ist bei gleich bleibender Anwenderfreundlichkeit schwer zu bewerkstelligen. Deshalb wurde die modulare Einbindung in eine Entwicklungsumgebung als ein Zweig der Fortentwicklung vorgeschlagen.

Mit den gewonnenen Erfahrungen aus Analyse und Evaluation, kann hier eine abschließende Bewertung über den Nutzen dieser Untersuchung erfolgen. Das vorgestellte Verfahren kann in vielerlei Hinsicht optimiert werden, was eine Menge Zeit bedarf. Es wird nicht als abwegig erachtet, nahezu alle tatsächlich vorkommenden Test-Klone innerhalb einzelner Projekte automatisch zu identifizieren. Im Gegensatz dazu hat sich die projektübergreifende Auswertung als uninteressant herausgestellt. Mit dem Ziel der Optimierung müssten allerdings weitere, einsetzbare Strategien entworfen oder zumindest implementiert werden. Dies kann der Einsatz einer semantischen Analyse nach der syntaktischen Analyse sein. Es kann aber auch eine Adaptierung an die Form der Testfälle sein. Je nach Länge der Testfälle, oder dem Vorkommen von vor- und nachgelagerten Methodenaufrufen, könnten unterschiedliche Bewertungskriterien eingesetzt werden. Eine pauschal einsetzbare Strategie, sowie eine eindeutige Systematik der Test-Klone konnte nicht festgestellt werden.

Wie sind die potentiellen Verwendungszwecke, die in der Einleitung genannt wurden, zu bewerten? In einzelnen Projekten, wie zum Beispiel *Log4J*, könnte zum Teil die automatische Generierung von Testfällen durch die hier gewonnen Erkenntnisse möglich sein. Die dort identifizierten Test-Klone unterschieden sich nur durch die Eingabedaten, deren Generierung nicht ausgeschlossen scheint. Dazu müsste in einigen Fällen nur die Vererbungshierarchie untersucht werden. Testfälle in denen sich das Testobjekt nur durch eine unterschiedliche Implementierung einer Schnittstelle unterscheidet, könnten so erzeugt werden. Auch bei *Args4J* könnten Testfälle, wenn auch eine vorherige statische Analyse der eigentlichen Anwendung notwendig wäre, generiert werden. Zum größten Teil ist dieses Ziel, zumindest im Auge des Autors, ein schwieriges Unterfangen. Gerade wenn die Testlogik sich unterscheidet, oder die Eingabedaten nicht nach gewissen Mustern festgelegt werden können, scheint es unmöglich dieses zu erreichen. Auch die potentielle Laufzeit, sollte es sich doch als möglich erweisen, wird als sehr hoch eingeschätzt.

Ebenfalls wird die Erstellung einer Datenbank, aus der schematische Testfälle, die häufig als Test-Klone identifiziert worden sind, ausgewählt werden können, als schwer realisierbar eingestuft. Dafür ist die syntaktische Varianz zwischen den Stilen der Testfälle einzelner Projekte zu groß und die Testobjekte in aller Regel zu unterschiedlich.

Einen nennenswerten Nutzen könnte diese Untersuchung bei der synchronen Modifizierung von Testfällen haben. Sofern es sich bei der Modifikation nicht um eine grundlegende Veränderung der Testlogik handelt, sollte dies möglich sein.

Literaturverzeichnis

[Aik11] Alex Aiken: *Measure of Software Similarity (MOSS)*, März 2011. `http://theory.stanford.edu/~aiken/moss/`.

[Beu12] Cedric Beust: *TestNG*, November 2012. `http://testng.org/doc/index.html`.

[BGS12] Kent Beck, Erich Gamma und David Saff: *JUnit*, November 2012. `http://www.junit.org`.

[BYM+98] Ira D. Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant' Anna und Lorraine Bier: *Clone Detection Using Abstract Syntax Trees*. 1998.

[CDR09] Michel Chilowicz, Etienne Duris und Gilles Roussel: *Syntax tree fingerprinting for source code similarity detection*. In: *Program Comprehension, 2009. ICPC '09. IEEE 17th International Conference on*, Seiten 243 –247, Mai 2009.

[DLS81] John L. Donaldson, Ann Marie Lancaster und Paula H. Sposato: *A plagiarism detection system*. In: *Proceedings of the twelfth SIGCSE technical symposium on Computer science education*, SIGCSE '81, Seiten 21–25, New York, NY, USA, 1981. ACM, ISBN 0-89791-036-2.

[eT12] Mathias Landhäußer und Walter F. Tichy: *Automated Test-Case Generation by Cloning*. In: *Proc. of the 7th International Workshop on Automation of Software Test (AST 2012)*, Juni 2012, ISBN 978-1-4673-1821-1.

[Foua] Apache Software Foundation: *Apache Tomcat*. `http://tomcat.apache.org/`.

[Foub] Apache Software Foundation: *Derby*. `http://db.apache.org/derby/`, Version: 10.8.3.0.

[Fouc] Apache Software Foundation: *Log4J*. `http://logging.apache.org/log4j/2.x/`, Version: 2.0-beta3.

[Fou12] Eclipse Foundation: *Eclipse IDE for Java Developers*, Dezember 2012. `http://www.eclipse.org/downloads/packages/eclipse-ide-java-developers/junosr1`, Version: 4.2.1.

[Fou13a] Apache Software Foundation: *Apache Maven Project*, Februar 2013. `http://maven.apache.org/`.

[Fou13b] Apache Software Foundation: *Commons Email*, Januar 2013. `http://commons.apache.org/proper/commons-email/`, Version 1.3.

[Fou13c] Apache Software Foundation: *Commons Logging*, März 2013. `http://commons.apache.org/proper/commons-logging/`, Version 1.1.2.

[GB07] Erich Gamma und Kent Beck: *JUnit A Cook's Tour*. Seite 7, Januar 2007.

[GHK+98] Todd L. Graves, Mary Jean Harold, Jung Min Kim, Adam Porter und Gregg Rothermel: *An empirical study of regression test selection techniques.* In: *Proceedings of the 20th international conference on Software engineering*, Seiten 188–189. IEEE Computer Society, 1998.

[GI10] Julio Vilmar Gesser und Hendy Irawan: *JavaParser*, Januar 2010. `http://code.google.com/p/javaparser/`, Version: 1.5.

[GJS08] Mark Gabel, Lingxiao Jiang und Zhendong Su: *Scalable Detection of Semantic Clones.* In: *Proceedings of the 30th International Conference on Software Engineering*, Seiten 321–300. ICSE, 2008.

[Goo13] Google: *Guava*, Januar 2013. `http://code.google.com/p/guava-libraries/`, Version: 14.0-SNAPSHOT.

[HCJJ12] Adrian Hall, Dave Cramer, Kris Jurka und Oliver Jowett: *PostgreSQL*, November 2012. `http://jdbc.postgresql.org/`, Version: 9.2-1002.

[JBo12] JBoss: *JBoss Web*, Juli 2012. `http://www.jboss.org/jbossweb`, Version: 7.7.0-SNAPSHOT.

[Jet12] JetBrains: *IntelliJ IDEA*, Dezember 2012. `http://www.jetbrains.com/idea/`, Version: 12.0.1.

[JMS07] Lingxiao Jiang, Ghassan Misherghi und Zhendong Su: *Deckard: Scalable and Accurate Tree-based Detection of Code Clones.* 2007.

[Jon03] Joel Jones: *Abstract Syntax Tree Implementation Idioms.* In: *Proceedings of the 10th Conference on Pattern Languages of Programs (PLoP2003)*, 2003. `http://www.hillside.net/plop/plop2003/Papers/Jones-ImplementingASTs.pdf`.

[Kaw13] Kohsuke Kawaguchi: *Args4J*, Januar 2013. `http://args4j.kohsuke.org/`, Version: 2.0.23-SNAPSHOT.

[Kri03] Jens Krinke: *Advanced Slicing of Sequential and Concurrent Programs.* April 2003.

[Lev66] Vladimir I. Levenshtein: *Binary codes capable of correcting deletions, insertions, and reversals.* In: *Soviet Physics-Doklady*, Band 10, 1966.

[Mes07] Gerard Meszaros: *xUnit test patterns: Refactoring test code.* Addison-Wesley Professional, 2007. `http://books.google.de/books?id=-izOiCEIABQC&dq=xunit`.

[MKK] Guido Malpohl, Emeric Kwemou und Moritz Kroll: *JPlag.* `http://www.jplag.de`.

[oC84] University of California: *Diff*, 1984. `http://linux.die.net/man/1/diff`, `http://src.opensolaris.org/source/xref/onnv/onnv-gate/usr/src/cmd/diff/diff.c`.

[Ora] Oracle: *JDBC.* `http://www.oracle.com/technetwork/java/javase/jdbc/index.html`.

[Ora11] Oracle: *Java Language Specification.* Juli 2011. `http://docs.oracle.com/javase/specs/`.

[Ott76] K. J. Ottenstein: *An algorithmic approach to the detection and prevention of plagiarism.* SIGCSE Bull., 8(4):30–41, Dezember 1976, ISSN 0097-8418. `http://doi.acm.org/10.1145/382222.382462`.

[Pin05] Steffen Pingel: *Abstrakte Syntaxbäume*. 2005.

[PPM02] Lutz Prechelt, Michael Philippsen und Guido Malpohl: *Finding Plagiarisms among a Set of Programs with JPlag*. Journal of Universal Computer Science, 8(11):1016–1038, 2002. `http://jucs.org/jucs_8_11/finding_plagiarisms_among_a/Prechelt_L.pdf`.

[Rai04] J. B. Rainsberger: *JUnit Recipes: Practical Methods for Programmer Testing*. Manning Publications, 2004, ISBN 1932394230.

[RC08] Chancal K. Roy und James R. Cordy: *NICAD: Accurate Detection of Near-Miss Intentional Clones Using Flexible Pretty-Printing and Code Normalization*. In: *Program Comprehension, 2008. ICPC 2008. The 16th IEEE International Conference on*, Seiten 172 –181, june 2008.

[RCK09] Chanchal K. Roy, James R. Cordy und Rainer Koschke: *Comparison and evaluation of code clone detection techniques and tools: A qualitative approach*. Science of Computer Programming, Special Issue on Program Comprehension (ICPC 2008), 74(7):470 – 495, 2009, ISSN 0167-6423. `http://www.sciencedirect.com/science/article/pii/S0167642309000367`.

[Ron12] Ronmamo: *Reflections*, Mai 2012. `http://code.google.com/p/reflections/`, Version: 0.9.8.

[SJA+12] Srikanth Sankaran, Ayushman Jain, David Audel, Frederic Fusier, Jayaprakash Arthanareeswaran, Jerome Lanneluc, Markus Keller, Olivier Thomann, Philippe Mulet, Stephan Herrmann, Satyam Kandula und Walter Harley: *JDT/Core*, August 2012. `http://www.eclipse.org/projects/project.php?id=eclipse.jdt.core`, Version: 3.8.2.v20120814-155456.

[TIO12] TIOBE Software BV: *TIOBE-Index*, November 2012. `http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html`.

[WWKM12] Debora Weber-Wulff, Katrin Köhler und Christopher Möller: *Collusion Detection System Test Report 2012*, November 2012. `http://plagiat.htw-berlin.de/collusion-test-2012/`.

Anhang

A. XML Schema zu annotationsbasierten Testrahmen

Dieses *XML* Schema legt die erlaubte Definition zu annotationsbasierten Testrahmen fest.

```
<?xml version="1.0" encoding="UTF-8" ?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
        elementFormDefault="qualified"
        version="1.0"
        targetNamespace="http://www.kit.edu"
        xmlns:kit="http://www.kit.edu">

   <xsd:element name="test-framework" type="kit:unitTestFramework" />

   <xsd:complexType name="unitTestFramework">
      <xsd:sequence minOccurs="1" maxOccurs="1">
         <xsd:element minOccurs="1" maxOccurs="1" name="annotations" type="kit:unitTestFrameworkAnnotations" />
         <xsd:element minOccurs="1" maxOccurs="1" name="exception" type="kit:exceptionIdentifier" />
         <xsd:element minOccurs="1" maxOccurs="1" name="ignore" type="kit:ignoreIdentifier" />
      </xsd:sequence>
      <xsd:attribute name="name" type="xsd:string" use="required" />
      <xsd:attribute name="id" type="xsd:ID" use="required" />
   </xsd:complexType>

   <xsd:complexType name="unitTestFrameworkAnnotations">
      <xsd:sequence>
         <xsd:element minOccurs="1" maxOccurs="unbounded" name="annotation" type="kit:unitTestFrameworkAnnotation" />
      </xsd:sequence>
      <xsd:attribute name="scope" type="xsd:string" use="required" />
   </xsd:complexType>

   <xsd:complexType name="unitTestFrameworkAnnotation">
      <xsd:sequence>
         <xsd:element minOccurs="0" maxOccurs="unbounded" name="field" type="kit:unitTestFrameworkAnnotationField" />
      </xsd:sequence>
      <xsd:attribute name="id" type="xsd:ID" use="required" />
      <xsd:attribute name="name" type="xsd:string" use="required" />
      <xsd:attribute name="stage" type="kit:unitTestExecutionStage" use="optional" />
   </xsd:complexType>
```

```
35
36    <xsd:complexType name="unitTestFrameworkAnnotationField">
37       <xsd:attribute name="id" type="xsd:ID" use="required" />
38       <xsd:attribute name="name" type="xsd:string" use="required" /
             >
39    </xsd:complexType>
40
41    <xsd:complexType name="ignoreIdentifier">
42       <xsd:attribute name="annotation" type="xsd:IDREF" use="
             required" />
43       <xsd:attribute name="ref" type="xsd:IDREF" use="optional" />
44       <xsd:attribute name="expectedValue" type="xsd:string" use="
             optional" default="true" />
45    </xsd:complexType>
46
47    <xsd:complexType name="exceptionIdentifier">
48       <xsd:attribute name="annotation" type="xsd:IDREF" use="
             required" />
49       <xsd:attribute name="ref" type="xsd:IDREF" use="required" />
50    </xsd:complexType>
51
52    <xsd:simpleType name="unitTestExecutionStage">
53       <xsd:restriction base="xsd:string">
54          <xsd:enumeration value="BEFORE" />
55          <xsd:enumeration value="TEST" />
56          <xsd:enumeration value="AFTER" />
57       </xsd:restriction>
58    </xsd:simpleType>
59 </xsd:schema>
```

B. Unterschungskonfiguration

Hier sind die Konfigurationseinstellungen für die Testrahmen *JUnit 3* und *JUnit 4* aufgelistet. Diese können, wenn die Implementierung mindestens einmal gestartet wurde, im Heimverzeichnis des aktuellen Benutzers im Order `.jutci` unter dem Namen `application.properties` gefunden werden.

B.1. JUnit 3

```
1  filter.fingerprint.enabled=true
2  filter.fingerprint.forceIgnored.value=true
3  filter.fingerprint.methodPattern.value=test.*
4  filter.fingerprint.profileMissing.value=true
5  filter.length.minimum.name.value=1
6  filter.length.name.enabled=true
7  investigator.queue.size=512
8  investigator.termination=300000
9  investigator.threads=64
10 metric.hierarchy.enabled=true
11 metric.hierarchy.file.value=true
12 metric.hierarchy.method.value=false
13 metric.hierarchy.package.value=true
14 metric.hierarchy.repository.value=true
15 metric.test.enabled=true
16 metric.test.varFrameworks.value=true
17 modifier.normalizer.assert.value=true
18 modifier.normalizer.enabled=true
```

B.2. JUnit 4

```
filter.fingerprint.enabled=true
filter.fingerprint.forceIgnored.value=true
filter.fingerprint.methodPattern.value=
filter.fingerprint.profileMissing.value=false
filter.length.minimum.name.value=1
filter.length.name.enabled=true
investigator.queue.size=512
investigator.termination=300000
investigator.threads=64
metric.hierarchy.enabled=true
metric.hierarchy.file.value=true
metric.hierarchy.method.value=false
metric.hierarchy.package.value=true
metric.hierarchy.repository.value=true
metric.test.enabled=true
metric.test.varFrameworks.value=true
modifier.normalizer.assert.value=true
modifier.normalizer.enabled=true
```

C. Auswertung von Industrieprojekten

Es folgt die Auflistung von den Untersuchungsergebnissen aus vier Industrieprojekten.

C.1. Projekt A

$T_g = 6.886$

Ähnlichkeitsschwelle	K_a	K_i	K_q
50%	54.044	5.856	85,04%
60%	22.789	4.953	71,93%
70%	13.066	4.083	59,29%
80%	8.839	3.450	50,10%
90%	4.896	2.458	35,70%
95%	4.113	2.077	30,16%
100%	3.890	1.917	27,84%

C.2. Projekt B

$T_g = 5.478$

Ähnlichkeitsschwelle	K_a	K_i	K_q
50%	178.180	4.894	89,32%
60%	55.158	4.376	79,88%
70%	45.274	3.875	70,74%
80%	44.151	3.657	66,76%
90%	43.529	3.498	63,86%
95%	43.513	3.477	63,47%
100%	43.484	3.446	62,91%

C.3. Projekt C

$T_g = 809$

Ähnlichkeitsschwelle	K_a	K_i	K_q
50%	6.336	734	90,73%
60%	2.354	625	77,26%
70%	1.669	551	68,11%
80%	1.550	519	64,15%
90%	1.542	506	62,55%
95%	1.542	506	62,55%
100%	1.542	506	62,55%

C.4. Projekt D

$T_g = 269$

Ähnlichkeitsschwelle	K_a	K_i	K_q
50%	1.118	210	78,07%
60%	576	172	63,94%
70%	428	143	53,16%
80%	386	133	49,44%
90%	375	128	47,58%
95%	375	128	47,58%
100%	375	128	47,58%