UNIVERSITÄT LEIPZIG

Wirtschaftswissenschaftliche Fakultät
Institut für Wirtschaftsinformatik
Professur Softwareentwicklung
Betreuer: Prof. Dr. Ulrich Eisenecker

„Empirische Untersuchung der Eignung von Code-Clones für den Nachweis der Redundanz als Treiber für die Evolution von Programmierkonzepten“

Masterarbeit

zur Erlangung des akademischen Grades

Master of Science – Wirtschaftsinformatik

vorgelegt von: Harnisch, Björn Ole
Matrikelnummer: 3722194
Email-Adresse: ole.harnisch@gmail.com
Telefonnummer: +491728276077
Anschrift: Gohliser Str. 41
04155 Leipzig

Leipzig, den 9. Januar 2018

Abstract

Bei der Entwicklung von Programmen werden durch Entwickler regelmäßig Code-Clones durch das Kopieren von Quellcode erzeugt. In dieser Arbeit wird ein Ansatz zur automatisierten Messung dieses duplizierten Codes mit Hilfe von Clone-Detection-Tools über mehrere Versionen von verschiedenen Software-Produkten gezeigt. Anhand der Historien von Code-Clones werden Einflüsse auf die Redundanzen dieser Software empirisch gemessen. Damit wird eine Grundlage für den Beweis, dass die Entwicklung von Programmiersprachen zu einem dominanten Teil durch Redundanzreduzierung getrieben wird, geschaffen.

Inhaltsverzeichnis

1 Einleitung

1.1 Problemstellung

Das zentrale Werkzeug in der Softwareentwicklung sind die verwendeten Programmiersprachen und genauso wie Werkzeuge anderer Branchen werden sie stetig weiterentwickelt. In den Versionen der Programmiersprachen werden oftmals viele verschiedene neue Komponenten (oder *Features*) eingeführt und es ist schwer vorherzusagen, welches dieser Features sich in der Praxis durchsetzen wird. Ein häufig versprochener Vorteil, den neue Programmiersprachen-Features mit sich bringen, ist die Reduzierung von Redundanz [1]–[4]. Unterschiedliche Arbeiten konnten bereits die redundanzreduzierende Fähigkeiten von Features nachweisen [5], [6]. Viele der eingeführten Features mit redundanzreduzierenden Eigenschaften setzen sich dabei in ähnlicher Art und Weise in unterschiedlichen Programmiersprachen durch (vgl. Kapitel 2.1). Diese ähnliche Durchsetzungskraft an Klassen von Features, folgend auch Programmierkonzepte genannt, lässt eine Evolution auf höherer Abstraktionsebene vermuten: Die Ebene der Programmierkonzepte. Dennoch gibt es nach wie vor keine empirische Bestätigung dafür, dass die Redundanz ein maßgeblich treibender Faktor der Evolution sein könnte.

Ein erster Schritt für den Nachweis des Treibers der Redundanzreduzierung ist das Messen von Redundanz in Software. Der Einfluss einer Änderung der Programmiersprache auf diese Redundanz muss zudem nachverfolgt werden können. Darüber hinaus sollte eine Untersuchung der Größen für verschiedene Programmiersprachen und Features gleichermaßen anwendbar sein, um die Redundanzreduzierung als Treiber für eine größere Gesamtheit an Programmierkonzepten nachzuweisen.

1.2 Zielsetzung

Diese Arbeit ist als eine Vorarbeit zur Bestätigung folgender **Ausgangsthese** zu verstehen [7]:

„Redundanz ist ein zentraler Treiber für die Evolution von Programmierkonzepten.“

Dafür soll Redundanz gemessen und ausgewertet werden. Für eine allgemeingültige Messung von Redundanz sollen die sogenannten *Code-Clones* verwendet werden. Diese repräsentieren Duplikate innerhalb des Quellcodes und können mit Clone-Detection-Tools erhoben werden. Code-Clones bieten damit eine Kennzahl, die unabhängig von den verwendeten Programmierkonzepten und weitestgehend auch unabhängig von der Programmiersprache anwendbar ist. Würde ein signifikanter Einbruch an Code-Clones nach der Einführung eines Programmierkonzepts nachgewiesen werden, könnte dies als redundanzreduzierende Eigenschaft interpretiert werden.

Das führt zu der **Hypothese**, die durch eine empirische Untersuchung in dieser Arbeit bestätigt werden soll:

„Die $\overbrace{\textit{Messung und Auswertung von Code-Clones}}^{\text{I}}$ *kann* $\overbrace{\textit{als Methode zur Bestätigung}}^{\text{II}}$ *der*
$\overbrace{\textit{Reduzierung von Redundanz}}^{\text{III}}$ *als* $\overbrace{\textit{Treiber für Evolution von Programmierkonzepten}}^{\text{IV}}$
verwendet werden.“

Werden die Bestandteile weiter aufgegliedert, stellen sich zunächst verschiedene Fragen zur Definition, die hier geklärt oder durch Forschungsfragen spezifiziert werden sollen.

Der erste Teil der Hypothese wirft die Frage auf, wie ein automatisierter Prozess zur Erhebung und Auswertung nach spezifischen Fragestellungen aussehen könnte (**Forschungsfrage 1**). Der zweite Teil impliziert, dass mittels des Analyseprozesses aus Teil I ein weitläufiger Beweis der Ausgangsthese möglich ist. Der Beweis selbst ist aber nicht Teil dieser Arbeit. Hier soll lediglich evaluiert werden, ob der Beweis prinzipiell mit den gegebenen Werkzeugen möglich ist und wie er auszuführen wäre (**Forschungsfrage 2**). Im Rahmen von Teil III muss zunächst geklärt werden, was Redundanz in Software ist und was für Folgen eine Reduzierung hätte und wie sie herbeigeführt werden könnte (**Forschungsfrage 3**). Teil IV wirft die Frage nach einer Begriffsklärung von Programmierkonzepten auf (**Forschungsfrage 4**). Weiterhin wird im letzten Teil (Teil IV) zum ersten Mal die Evolution von Programmierkonzepten erwähnt. Der Begriff der Evolution soll dabei eine Analogie zu den Ausdrücken der Evolution von Software [8] oder Programmiersprachen [9] darstellen und soll einerseits die Entwicklung von neuen Programmierkonzepten umfassen als auch andererseits die Akzeptanz von Benutzern, also die Durchsetzung, in der entsprechenden Programmiersprache, in die ein solches Programmierkonzept integriert wird. Welche Treiber es für diese Evolution gibt und welche Nachweisansätze eventuell dafür existieren (**Forschungsfrage 5**), wären offene Fragen zur Unterstützung der Ausgangsthese und von Interesse als Quellen für den

Aufbau des Analyseprozesses. Damit sollen mögliche Konkurrenten identifiziert werden und allgemeine Verfahren für Nachweise für die eigene Anwendung evaluiert werden.

Der Eigenanteil und Mehrwert dieser Arbeit liegt dabei insbesondere bei der Beantwortung von Forschungsfragen 1 und 2, der Entwicklung eines Prozesses zur Erhebung von Code-Clones aus Softwarequellen und einer Auswertung der Ergebnisse als Vorbereitung und Ansatzerarbeitung für den Beweis der Ausgangsthese. Die übrigen Fragen sind daher eher als Teilziele der Fragen 1 und 2 zu verstehen.

1.3 Vorgehensweise

Ein erster Ansatz zur Bestätigung der Hypothese war eine empirische Analyse der Java Generics [10] als ein umgesetztes Programmierkonzept. Nach ausgiebiger Literaturrecherche stellte sich jedoch heraus, dass diese Fragestellung bereits in einer anderen Arbeit, der von Parnin et al. [5], empirisch nachgewiesen wurde. Dieser Ansatz ist jedoch sehr ursachenorientiert und der reale Einfluss wird wenig gemessen (genauer in Kapitel 2.2.3). Ein weiterer Nachteil zur Argumentation der Ausgangsthese ist, dass mit der Analyse, die in der Arbeit vorgenommen wurde, nicht ohne weiteres die Analyse eines anderen Programmierkonzepts möglich ist. Deswegen wurde das zentrale Thema der vorliegenden Arbeit umorientiert auf eine Untersuchung der Eignung von Code-Clones für die Untersuchung unterschiedlicher Programmierkonzepte für einen allgemeineren Nachweis von Redundanzreduzierung.

Die Erhebung und Auswertung der Code-Clones findet in einem Analyseprozess statt, der sich in einen Erhebungsprozess und die Auswertung nach verschiedenen Fragestellungen aufteilt. Für die Erhebung wurden für eine automatisierte Ausführung der Prozesse Programme in Python [11] geschrieben. Außerdem wurden die Clone-Detection-Tools Simian [12] und CCFinderX [13] für die Messung von Code-Clones verwendet. Die Auswertungen wurden mit Hilfe der statistischen Sprache R [14] entwickelt.

Neben der automatisierten Analyse mit Pythonscripts und unterstützender Software wurden an vielen Stellen der Arbeit Literaturrecherchen nach dem Vorbild von Brocke et al. [15] durchgeführt. Dabei sind zunächst Eingrenzungen der Zielstellungen der Recherchen mit Hilfe von Fragen und der Taxonomie von Cooper [16] vorgenommen worden. Anschließend wurden Begriffe in einem Begriffsbaum aufgegliedert (ein Begriffsbaum zu allgemeinen Begriffen findet sich in Abbildung 18 auf S. V im Anhang), um durch Abdeckung von Synonymen und Teilbegriffen die vielfältigen Suchanfragen formulieren zu können. Die Suchanfragen wurden auf verschiedenen Suchplattformen wie Google Scholar [17], IEEExplore [18] und Ebsco [19] gestellt. Die Paper der Ergebnisse wurden auf Relevanz geprüft und mittels Zotero [20] in Kategorien ihrer Zugehörigkeit für Wiederauffindbarkeit sortiert. Eine weiterführende Beschreibung des Vorgehens bei der Literaturrecherche findet sich im Anhang anhand des Beispiels einer Suche zu vergleichbaren Arbeiten auf Seite V.

Der Hauptteil der Arbeit unterteilt sich in die Beschreibung des Erhebungsprozesses (Kapitel 3) und die Auswertungen der erhobenen Daten (Kapitel 4). Die dafür notwendigen Definitionen und Vorbetrachtungen werden in Kapitel 2 durchgeführt. Dort werden Programmierkonzepte, deren Treiber und die Auswahl von Quellen für die Analysen erläutert. Abschließend findet in Kapitel 5 eine Nachbetrachtung statt, in der in einer Fehlerbetrachtung verschiedene Risiken der Validität und Erweiterungsmöglichkeiten für weiterführende Arbeiten untersucht werden.

2 Vorbetrachtung

2.1 Programmierkonzepte

2.1.1 Definition

Als Grundlage für die Programmierung werden in der Praxis verschiedene Programmiersprachen verwendet. Jede dieser Programmiersprachen hat dabei unterschiedliche Eigenschaften und Funktionalitäten. Allerdings werden viele dieser Funktionalitäten in verschiedenen Programmiersprachen gleichermaßen umgesetzt, zum Beispiel:

- Lambdaausdrücke in Java [2], Scala [21], C# [22] etc.
- Generische Ausdrücke in Ada [23], C++ [24] und Java [1]
- Variablen-Deklaration in den meisten gängigen Programmiersprachen

Fasst man nun diese Funktionalitäten einzelner Programmiersprachen mit gleichen Funktionalitäten anderer Programmiersprachen zusammen, kann diese Klasse von Funktionalitäten als Programmierkonzept bezeichnet werden. Ein Programmierkonzept ist somit eine programmiersprachenübergreifende Funktionalität [7].

Ein Element der Ausgangsthese von Seite 1 ist, dass die Evolution von Programmiersprachen auf der Ebene der Programmierkonzepte stattfindet. Das heißt vor der Einführung eines Features in eine Sprache findet die Umwandlung eines Programmierkonzepts in ein Feature statt. Ein Programmierkonzept ist somit gleichermaßen eine abstraktere Formulierung eines Features. Die Evolution einer Programmiersprache geht damit entweder mit der Umsetzung eines vorhandenen Programmierkonzepts oder mit der Entwicklung eines neuen Programmierkonzepts einher.

2.1.2 Programmierkonzepte in Java

Die empirische Untersuchung der vorliegenden Arbeit hat das Ziel der Analyse des Einflusses von Programmierkonzepten auf die Redundanz in Software. Als untersuchte Programmiersprache wurde im Rahmen der prototypischen Untersuchung zunächst Java ausgewählt. In Java wurden durch die Einführung verschiedener Features tiefgreifende Änderungen vorgenommen, deren Verwendung eine Redundanzminderung vermuten lässt. Außerdem ist Java weit verbreitet und es gibt zahlreiche, frei zugängliche Projekte[1] zur Auswahl für eine Analyse.

Damit aber konkrete Zeitpunkte in den Analysen untersucht werden können, müssen zunächst wichtige Versionen in der Sprache ausgemacht werden.

Version	*Datum der Veröffentlichung*	*Features*
J2SE 1.4	06.02.2002	RegEx; Non-blocking I/O; XML Parser; Java Web; Exception Chaining
J2SE 5.0	30.09.2004	Generics; Annotationen; Autoboxing; Enumerationen; Varargs; erweiterte for-Schleife; statische imports; Scanner Klasse
Java SE 6	23.12.2006	Scripting Language Support; Jaxb 2.0
Java SE 7	07.07.2011	Dynamic language support; kürzere Genericsdefinition; Strings in switch
Java SE 8	18.03.2014	Lambdas; Streams; Methodenreferenz; wiederholende Annotationen
Java SE 9	21.09.2017	Jigsaw; JLink; Variable Handles

Tabelle 1: Java-Versionen mit eingeführten Features mit Redundanzverminderungspotenzial (Daten aus: [28], [29])

In Tabelle 1 sind die jüngsten Versionen von Java aufgeführt mit dem Datum ihrer Einführung sowie den eingeführten Features, die für diese Arbeit von Relevanz sein könnten, weil bei ihnen redundanzvermindernde Eigenschaften und größerer Einfluss vermutet werden. Die Tabelle erreicht dabei keine Vollständigkeit, weil sie auf subjektiver Einschätzung basiert. Dennoch kann gefolgert werden, dass die Versionen von Java 5 und 8 wahrscheinlich größeren Einfluss haben, weil dort mehrere Sprachelemente eingeführt wurden, deren jeweiliges Ziel laut Entwicklern [1], [2] eine Redundanzminderung war. Prinzipiell sollte aber jede der genannten Versionen einen größeren Einfluss auf die Redundanz haben, weil dies größere Versionen sind, die neue Funktionalitäten mit sich bringen. Im Gegensatz dazu beinhalten Versionen zwischen den genannten eher Fehlerbehebungen oder neue Bibliotheken. Deswegen können Punkte mit starken Rückgängen von Redundanz nach der Einführung dieser Versionen vermutet werden. Aus diesem Grund werden die Zeitpunkte der Versionen in den Analysen näher beleuchtet.

---

[1] Allein auf der Distributions-Plattform *github* gab es im Jahr 2014 222.852 aktive Java-Repositories, eine Vielzahl davon mit Open-Source-Lizenzen [25], [26]. Bezogen auf Änderungsanträge wurde Java sogar als drittaktivste Sprache für Open-Source-Projekte im Jahr 2017 ermittelt [27].

Als Beispiele für Features werden in dieser Arbeit vermehrt die sogenannten Java Generics und Lambdaausdrücke verwendet. Beide versprechen eine hohe Redundanzverminderung, sind aber dennoch zeitlich spät genug eingeführt worden, um durch eine Repository-Analyse erfasst werden zu können.

```
 2
 3 public class Test {
 4
 5     public static void main(String[] args) {
 6         Maxer c=new Maxer();
 7         double d1 = 42.0, d2 = 22.0/7;
 8         System.out.println(c.maximize(d1,d2));
 9
10         int c1 = 13, c2 = 21/3;     ;
11         System.out.println(c.maximize(c1,c2));
12
13     }
14
15
16 }
17
18 class Maxer{
19     double maximize(double a,double b) {
20         System.out.println("Now the variables are comparing:");
21         if (a>=b){
22             return a;
23         }else{
24             return b;
25         }
26     }
27
28     int maximize(int a,int b) {
29         System.out.println("Now the variables are comparing:");
30         if (a>=b){
31             return a;
32         }else{
33             return b;
34         }
35     }
36
37 }
```

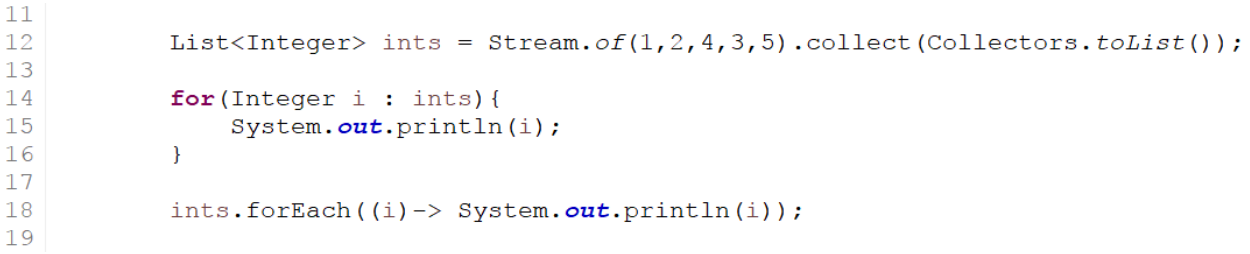
a) *Implementierung ohne generische Typen*

```
 2
 3 public class Test {
 4
 5     public static void main(String[] args) {
 6         Maxer c=new Maxer();
 7         double d1 = 42.0, d2 = 22.0/7;
 8         System.out.println(c.maximize(d1,d2));
 9
10         int c1 = 13, c2 = 21/3;     ;
11         System.out.println(c.maximize(c1,c2));
12
13     }
14
15
16 }
17
18 class Maxer{
19     public  <T extends Comparable<T>> T maximize(T a,T b) {
20         System.out.println("Now the variables are comparing:");
21         if (a.compareTo(b)>0){
22             return a;
23         }else{
24             return b;
25         }
26     }
27
28 }
29
```

b) *Implementierung mit generischen Typen*

Abbildung 1: Java-Klassen zur Maximierung einer Variable aus zweien.

Generics sind Ausdrücke, die in einer mit einem Typsystem ausgestatten Sprache Variablen für bestimmte Typen darstellen können, um Ausdrücke zu verallgemeinern, ohne die Typsicherheit zu gefährden (vgl. [10]). Dies kann beispielsweise verwendet werden, um allgemeinere Klassen zu erstellen, die bei der Instanziierung zu einem Objekt auf einen bestimmten Typ heruntergebrochen werden (zum Beispiel wird eine allgemeine Liste zu einer String-Liste). Des Weiteren können auch Methoden für Typen mit bestimmten Eigenschaften, wie zum Beispiel die Vergleichbarkeit der Elemente dieses Typs (*Comparable*), erstellt werden. In Abbildung 1 b ist ein solcher Ausdruck zu sehen. Im Vergleich zu einer Programmierung ohne Generics können bestimmte Ausdrücke, beispielsweise in Zeilen 19 bis 35 in Abbildung 1 a zu sehen, verkürzt werden zu einem Ausdruck, wie in Abbildung 1 b in den Zeilen 19 bis 26 erkennbar. Damit Redundanz wirklich reduziert wird, muss es Aufrufe mit unterschiedlichen Typen geben. Würde in diesem Beispiel der Aufruf immer nur mit *int*-Variablen stattfinden, wäre ein Methode mit einer Typisierung für *int,* wie in Abbildung 1 a, gemessen an der Anzahl der verwendeten Elemente (*tokens*) sogar kürzer.



```
11
12         List<Integer> ints = Stream.of(1,2,4,3,5).collect(Collectors.toList());
13
14         for(Integer i : ints){
15             System.out.println(i);
16         }
17
18         ints.forEach((i)-> System.out.println(i));
19
```

Abbildung 2: Eine for-Schleife und ein Lambdaausdruck mit gleicher Funktion.

Die Lambdaausdrücke besitzen ein ähnliches Einsparungspotenzial, denn Lambdaausdrücke oder allgemeiner auch „anonyme Funktionen“ sind Funktionen, die ohne Namen aufgerufen werden können [30] und haben damit die Möglichkeit, ohne zusätzlichen Code bestimmte Ausdrücke durch eine verkürzte Darstellung zu ersetzen [31]. In Abbildung 2 ist zu sehen, wie eine Schleife (Zeile 14-16) ersetzt durch einen Lambdaausdruck (Zeile 18) aussehen könnte. Damit werden in diesem Fall zwei Zeilen gespart. Prinzipiell ist genauso wie bei den Generics ein wichtiger Grund für die Einführung, dass Redundanz eingespart werden kann.

2.2 Treiber für die Entwicklung von Programmierkonzepten

Die Umsetzung eines Programmierkonzepts in eine Programmiersprache sowie die darauffolgende Akzeptanz und Verwendung des entstehenden Features durch die Anwender der Programmiersprache könnte als Evolution des Programmierkonzepts bezeichnet werden. Eigenschaften, die eine solche Evolution vorantreiben, werden folgend als Treiber bezeichnet.

Die Einführung eines Programmierkonzeptes in eine Programmiersprache geht in der Regel mit dem Versprechen der Verbesserung bestimmter Eigenschaften des Codes einher. Solche Verbesserungen könnten als Treiber des Programmierkonzepts verstanden werden, da es Argumente für die Benutzer der Sprache sind, ein entsprechendes Feature zu verwenden. Gleichermaßen waren diese Argumente vermutlich auch für die Entwickler der Programmiersprachen die ausschlaggebenden Punkte für die Umsetzung des Programmierkonzepts in ein Feature. Diese Annahmen sind Grundlage für die Untersuchungen dieser Arbeit zu Treibern von Programmierkonzepten und dienen einer vereinfachten Ermittlung von potenziellen Treibern.

2.2.1 Arten der Treiber von Programmierkonzepten

Ein Ziel dieser Arbeit war es vor allem, den Treiber der Reduzierung von Redundanz im Code zu analysieren und dabei insbesondere zu untersuchen, mit welcher Metrik dieser Treiber als zentraler Treiber nachgewiesen werden könnte. Aber es gibt auch andere Treiber, denen jeweils unterschiedliche Metriken zugehörig sind. Eine Teilfrage dieser Arbeit war es zunächst, mögliche Treiber zu identifizieren, um etwaige Konkurrenten für die Redundanz vergleichsvorbereitend zu identifizieren oder eventuell Vorgehensweisen beim Nachweis von Treibern nachzuahmen (Forschungsfrage 5).

Zur Identifizierung der Treiber wurde einerseits eine Literaturrecherche zu Treibern, Akzeptanz und Entwicklung von Programmiersprachen und Features im Allgemeinen und Speziellen durchgeführt wie in Kapitel 1.3 beschrieben. Andererseits wurden Patchnotes verschiedener Programmiersprachenversionen analysiert, um Hinweise über versprochene Verbesserungen der eingeführten Features zu erfahren. Ergebnisse dieser Untersuchungen finden sich in Tabelle 2.

Möglicher Treiber	*Beschreibung*	*Kontext der Erwähnung*	*Quelle*	*Durch Redundanz-reduzierung beeinflusst?*
Lesbarkeits-steigerung	Das Programmierkonzept sorgt für intuitiv besser lesbaren Code für den Entwickler.	Java Methoden-referenzen	[2]	bedingt
Fehlerprävention	Durch das Programmierkonzept können Fehler in der Ausführung vermieden werden.	Typsystem	[32, S. 263]	nein
Verringerter Implementierungs-aufwand	Das Programmierkonzept verringert den Aufwand der Implementierung bestimmter Ausdrücke.	Java Erweiterte for-Schleife	[1]	ja
Verringerte Fehleranfälligkeit	Das neue Programmierkonzept ist weniger fehleranfällig bei der Erstellung als das vorher für den betreffenden Use Case verwendete Konstrukt.	Java Typsichere Enumerationen	[1]	nein
Verringerter Wartungsaufwand	Durch das Programmierkonzept entsteht weniger Aufwand in der Wartung des Quellcodes an Verwendungsstellen.	Java Annotations	[1]	ja
Geringerer Wortaufwand	Auch „verbosity“ – Das Programmierkonzept reduziert in der Regel im Kontext von Boilerplate-Code unnötige Elemente aus der Syntax.	Java Generics	[33]	ja

Gesteigerte Konsistenz	Das Programmierkonzept vereinheitlicht verschiedene Verhaltensmuster.	Python eingebaute breakpoints	[34]	nein
Einfachheit	Das Programmierkonzept steigert die Erlernbarkeit bestimmter Aspekte der Sprache.	Python eingebaute breakpoints	[34]	bedingt
Erweiterte Möglichkeiten	Das Programmierkonzept führt zu einer neuen Möglichkeit der Ausführung, die mit den vorigen Sprachelementen nicht rekonstruierbar war.	Python try/except/else/finally-Block	[35]	nein

Tabelle 2 Mögliche Treiber für die Entwicklung von Programmierkonzepten

Die Tabelle wurde im Rahmen dieser Arbeit zusammengetragen, stellt nur einen Auszug dar und erhebt somit keinen Anspruch auf Vollständigkeit. Des Weiteren sind die Treiber mitunter nur positive Eigenschaften der Features. Außerdem sind die Treiber nicht atomar und nicht zwangsläufig konkurrierend, weswegen ein Treiber auch einen anderen Treiber positiv beeinflussen kann. Zur Unterstreichung des hier untersuchten Treibers der Redundanzreduzierung wurden die dadurch beeinflussten Treiber markiert. In der letzten Spalte steht ein „ja“, wenn eine Redundanzreduzierung als Nebeneffekt auch diesen Vorteil bringt. Für Lesbarkeit ist ein „bedingt“ enthalten, weil die Lesbarkeit durch weniger irrelevante Information sowohl verbessert als auch verschlechtert werden kann. Gleiches gilt für die Einfachheit.

2.2.2 Reduzierung von Redundanz in Software

Im Folgenden soll die Redundanz in Software näher beschrieben werden, deren Reduzierung als zentraler Treiber für die Einführung und Akzeptanz von Programmierkonzepten vermutet wird.

2.2.2.1 Arten von Redundanz in Software

Je nach Fachgebiet gibt es verschiedene Ansätze für die Definition von Redundanz. Im Duden wird Redundanz als „das Vorhandensein von eigentlich überflüssigen, für die Information nicht notwendigen Elementen“ [36] definiert. Diese Definition findet sich ähnlich auch in der Informationstheorie wieder als „überflüssiger Teil einer Nachricht ohne wesentliche neue Inform[ation] …“ [37]. In Definitionen der Linguistik oder Kommunikationstheorie wird Redundanz hingegen nicht ausschließlich als unnötige Information, sondern explizit als gedoppelte oder mehrfach genannte Information definiert [38].

Betrachtet man Redundanz nicht nur als die Mehrfachnennung von Information, sondern auch als unnötige Elemente ohne Informationsgehalt, so gibt es dafür auch im Quellcode von Software Vertreter. Ein Beispiel dafür ist der sogenannte *Boilerplate-Code.* Boilerplate-Code ist die Bezeichnung für Code, der überflüssigerweise in einer Programmiersprache beim Einsatz bestimmter Aufrufe erzeugt werden muss [33], mit anderen Worten überflüssige Elemente von Informationen. Andere Vertreter für überflüssige Information im Code sind toter oder unerreichbarer Code, also Abschnitte im Code, die entweder in keiner Ausführung benötigt werden oder gar nicht erst aufgerufen werden können. Ursprung dieser sind in der Regel schlechte Entwicklungspraktiken oder mangelnder Überblick über die Software. Diese Abschnitte können durch unterschiedliche Werkzeuge erkannt und beseitigt werden (z.B. [39]–[41]).

Eine Alternative ist die Betrachtung von Redundanz ausschließlich als Dopplung oder Mehrfachnennung. Folgend wird diese Art zur logischen Aufgliederung in zwei verschiedene Varianten der Skalierungen aufgeteilt. Einerseits könnten sich Module oder Softwarepakete mit verschiedenen Funktionspaletten funktional überschneiden (folgend: **funktionale Redundanz**). Andererseits können sich vereinzelte Abschnitte des Codes semantisch oder syntaktisch doppeln (folgend: **interne Redundanz**). In dieser Arbeit liegt der Fokus in erster Linie auf interner Redundanz. Funktionale Redundanz ist aber mitunter Ursache für starke interne Redundanz in Softwareprojekten (vgl. Kapitel 4.4) und wird deswegen ebenfalls betrachtet.

Zur Messung von interner Redundanz in Software kann zum Beispiel die Definition von Code-Clones zugezogen werden. Mittels Clone-Detection können Code-Clones im Quellcode in ihrer Anzahl und Größe gemessen werden [42].

2.2.2.2 Code-Clones

Zur Klärung des Begriffs der Code-Clones werden zunächst sogenannte *Code-Fragmente* als jegliche Art von zusammenhängenden Code-Zeilen definiert. Code-Fragmente können eindeutig durch den beinhaltenden Dateinamen, die Start- und die Endzeile identifiziert werden. [42, S. 471]

Ein Code-Fragment zählt als *Clone* zu einem anderen Fragment, wenn die Fragmente Ähnlichkeit nach einer Ähnlichkeitsfunktion aufweisen. Solch eine Ähnlichkeitsfunktion variiert je nach Clone-Typ (siehe unten). [42, S. 471]

Mit anderen Worten sind *Code-Clones* syntaktische, semantische oder funktionale Duplikate von zusammenhängendem Quellcode innerhalb einer Software.

Dabei gibt es verschiedene Typen von Code-Clones. Die Typen 1-3 wurden erstmals in [43] erwähnt und wurden dann in [44], [45] um Typ-4 erweitert (vgl. [42, S. 472]). Mittlerweile wird die Unterteilung in diese Typen aber weitestgehend als Standard in verschiedensten Arbeiten akzeptiert (vgl. [46]–[50]).

Die vier unterschiedlichen Typen sind [42, S. 472]:

- **Typ-1**: Identische Code-Fragmente abgesehen von Leerzeilen, Formatierungen und Kommentaren
- **Typ-2**: Syntaktisch identische Fragmente abgesehen von Leerzeilen, Formatierungen, Kommentaren, Identifizieren, Literalen und Typen
- **Typ-3**: Kopierte Fragmente mit weiteren Modifikationen wie zusätzliche, gestrichene oder leicht geänderte Aufrufe zusätzlich zu geänderten Leerzeilen, Formatierungen, Kommentaren, Identifizieren, Literalen und Typen
- **Typ-4**: Zwei oder mehr Code-Fragmente, die die gleichen Algorithmen ausführen, aber syntaktisch unterschiedlich implementiert sind.

Typ-4-Clones werden dabei auch in aktuelleren Arbeiten häufig vernachlässigt (z.B. [46], [49]). Abgesehen von Typ-4 beinhaltet jeder Typ auch die Typen niedrigerer Stufe. In der Regel kann ein Clone-Detection-Tool (Clone-Detection-Tool) Typen bis zu einer bestimmten Stufe identifizieren, also einschließlich aller darunterliegenden. Davon gibt es allerdings auch vereinzelte Ausnahmen (vgl. [42, S. 484]). Die Betrachtung unterschiedlicher Typen ist deswegen interessant, weil damit klare, qualitative Grenzen unterschiedlicher Clone-Detection-Tools ausgemacht werden können.

Programmierkonzepte beeinflussen vermutlich nicht jede Art von Clone-Typ gleichermaßen, weswegen eine Berücksichtigung der Typen auch eine Rolle bei der Auswahl des Clone-Detection-Tools spielt (vgl. Kapitel 3.3). Welche Art von Programmierkonzepten welchen Einfluss auf unterschiedliche Clone-Typen hat, ist aber eine Untersuchung, die in dieser Form in dieser Arbeit nicht stattfindet, weil hier zunächst lediglich geprüft werden soll, ob die Einführung von Programmierkonzepten überhaupt einen Einfluss auf Code-Clones hat.

Die Messung von Code-Clones, folgend auch *Clone-Detection* genannt, findet dabei in der Regel statisch statt (vgl. [42]). Für diese Arbeit ist darüber hinaus aber auch eine dynamische Analyse über verschiedene Versionen von Software notwendig. Dafür werden folgend verschiedene Herangehensweisen vorgestellt.

In der Untersuchung von Clone-Evolution wird sich mit dem Nachverfolgen von Code-Clones und deren Veränderungen über verschiedene Versionen einer Software während der Softwareerstellung und der Softwareevolution, also während Aktivitäten der Erweiterung, Instandhaltung und Wartung [8, S. V ff.], beschäftigt (vgl. [46]). Dazu gibt es bereits verschiedene Untersuchungen zahlreicher Autoren zur Findung von effizienten Algorithmen, so zum Beispiel Göde et al. [46] und Duala-Ekoko et al. [51]. Diese Arbeiten stellen Ansätze vor, mit denen Code-Clones über verschiedene Versionen hinweg identifiziert werden können und liefern somit Ansätze, die zu Teilen in der vorliegenden Arbeit verwendet werden konnten.

Im Gegensatz dazu stehen Arbeiten, in denen Code-Clones nicht nachverfolgt, sondern in mehreren Versionen der Software separat erhoben wurden, wie beispielsweise in den Arbeiten von Goon et al. [52] und Lavoie und Merlo [53]. Diese Arbeiten haben einen sehr ähnlichen Aufbau zu der empirischen Untersuchung, die hier durchgeführt wurde und inspirierten damit einen Großteil der verwendeten Ansätze für den Prozessablauf.

2.2.2.3 Folgen von Redundanz in Software

In Studien hat sich herausgestellt, dass interne Redundanz, insbesondere in Form von Code-Clones, erhöhten Wartungsaufwand und -kosten durch Mehrfachwartung nach sich zieht [54]. Darüber hinaus bergen Code-Clones bei inkonsistenter Anpassung auch eine erhöhte Gefahr der Einführung von Fehlern oder beim Duplizieren selbst eine Fortsetzung von Fehlern. Auch Erweiterungen und Modifikationen werden durch interne Redundanz erschwert, weil oft unterschiedliche Anpassungspunkte beachtet werden müssen [55, S. 7–8]. Funktionale Redundanz hat darüber hinaus den Nachteil, dass die unnötige Abdeckung von Funktionen womöglich zusätzliche Kosten für die Implementierung oder den Einkauf aufgeworfen hat.

Funktionale Redundanz kann mitunter auch positive Folgen haben. So kann durch funktionale Dopplung zum Beispiel Ausfallsicherheit erreicht werden oder wie in einem Best-Of-Breed-Ansatz [56] der beste Vertreter seiner Art identifiziert werden. Unter Umständen wird solche Redundanz absichtlich eingeführt. Ein Beispiel dafür ist die N-

Versionen-Programmierung, bei der verschiedene funktionell gleiche Softwareversionen erstellt werden, um eine Fehlertoleranz aufzubauen [57].

Auch für die interne Redundanz hat sich herausgestellt, dass es durchaus Muster gibt, die beispielsweise durch erhöhten Wiedererkennungswert positive Auswirkungen auf die Quellcodequalität haben (vgl. [58]). Außerdem können mit modernen Clone-Tracking-Tools auch die negativen Folgen der Mehrfachwartung oder -anpassung durch gezieltes Aufspüren der Clones umgangen werden [51].

Trotz der potentiellen positiven Folgen wird im Rahmen dieser Arbeit davon ausgegangen, dass neue Programmierkonzepte eine Redundanzreduzierung gegenüber einer -steigerung präferieren, weil bis zu einem bestimmten Maß die positiven Effekte überwiegen.

2.2.3 Ansätze für den Nachweis von Redundanzreduzierung als Treiber

In diesem Kapitel werden verschiedene Arbeiten vorgestellt, die sich mit Ansätzen für den Nachweis von Redundanzreduzierung als Treiber für die Evolution von Programmierkonzepten, ähnlich dem Ziel dieser Arbeit, beschäftigen. Dafür wurden Beiträge zur Analyse von Redundanz, zu Nachweisen von Gründen für Akzeptanz verschiedener Features und zu Redundanzreduzierung in einer grundlegenden Literaturrecherche (vgl. Kapitel 1.3) ermittelt und auf Ähnlichkeiten zu Zielen dieser Arbeit geprüft.

Für den Nachweis von Redundanz in Software gibt es bereits verschiedene Ansätze. Dabei ist unter anderem das Messen von Code-Clones oder Boilerplate-Code (vgl. Kapitel 2.2.2.1) zu nennen. Zur Messung von Code-Clones gibt es verschiedene Arbeiten unter dem Namen „Clone Detection“ (vgl. Kapitel 3.3.1). Auch die Messung von Code-Clones über verschiedene Versionen von Software-Repositorys wurde bereits in verschiedenen Arbeiten analysiert [51], [52].

Ansätze zum Beweisen der Gründe für die Akzeptanz oder die Ablehnung durch Benutzer gibt es für verschiedene Features in Programmiersprachen. In der Arbeit von Malloy und Power [59] wurde beispielsweise die Ablehnung von Python 3 empirisch untersucht. Dabei wurden Metriken zur Verwendung und Akzeptanz eingeführt. In der Arbeit von Capek et al. [60] wird eine Forschung vorgestellt, die sich damit beschäftigt, allgemeine Metriken für die Akzeptanz von C#-Features zu finden. Teilweise wird auch nur der ideale oder potenzielle Einfluss eines neuen Features einer Programmiersprache untersucht, so zum Beispiel in der Arbeit von Fuhrer et al. [61]. Dort wurde geprüft, wie viele der fehleranfälligen Casts in Java durch Generics ersetzt werden könnten.

Für den Nachweis von Redundanzreduzierung als Treiber für die Entwicklung oder Akzeptanz von Programmierkonzepten ist in der Literatur vor allem die Methode der Syntaxanalyse aufgefallen. In der Arbeit von Parnin et al. [5] werden dabei insbesondere Java Generics als Programmierkonzept betrachtet. Durch Syntaxbaumanalysen wird hier versionsübergreifend nachverfolgt, wo und wie oft Java Generics ihren Einsatz finden. Anhand der Art und Weise des Einsatzes kann hochgerechnet werden, wie viele Code-Zeilen dadurch gespart wurden. Dafür wurden pro generischer Methode oder Klasse die verschiedenen instanziierenden Typen, die im Einsatz bei den untersuchten Softwarequellen waren, gezählt. Mit der Anzahl der jeweils verwendeten Typen kann durch Multiplikation mit den verwendeten Code-Zeilen für den Ausdruck abgeschätzt werden, wieviel Code ohne Generics hätte mehrfach geschrieben werden müssen.

Im Endeffekt konnte damit gezeigt werden, dass Java Generics zu einer Reduktion von Code-Duplikaten führen. Ein Vorteil dieses Ansatzes ist vor allem, dass die Code-Reduzierung exakt einer Ursache, den Java Generics, zugeordnet werden kann. Andererseits basieren die Zahlen nur auf Annahmen. So würde zum Beispiel ein ineffizienter Ansatz, wie die Instanziierung veralteter Typen, in der Hochrechnung nicht auffallen. Außerdem wäre eine Umstellung auf ein anderes Programmierkonzept für eine breitflächige Untersuchung sehr aufwändig, da die jeweiligen Syntaxelemente sowie ihre Hochrechnungsmethoden geändert werden müssten.

Zusammenfassend gibt es viele Ansätze für den Nachweis von Redundanz in Software. Auch eine Redundanzreduzierung kann nachgewiesen werden. Darüber hinaus gibt es verschiedene Studien über die Akzeptanz und Auswirkung der Einführung einzelner Programmierkonzepte. Es wurde aber kein Ansatz für den Nachweis gefunden, dass Redundanz übergreifend für die Evolution verschiedener Programmierkonzepte ein ausschlaggebender Treiber ist. Dies kann zwar allgemein anhand der bestätigten Nachweise für Programmierkonzepte vermutet werden, aber für eine Bestätigung wäre eine Vergleichsmethode vonnöten, die Redundanz unabhängig vom untersuchten Programmierkonzept messen und nachweisen kann. Dafür soll in dieser Arbeit die Messung von Code-Clones verwendet werden. Außerdem soll testweise geprüft werden, ob ursachenorientierte Ansätze, wie der von Parnin et al [5], mit dem effektorientierten Ansatz der Messung von Code-Clones vereinbar ist (vgl. Kapitel 4.4).

2.3 Auswahl Software Repositories für die Analysen

Die Basis der empirischen Untersuchung dieser Arbeit bilden verschiedene Software-Repositories. Ein Repository ist dabei ein Behälter von Dateien eines Software-Projekts über mehrere Versionen des Projekts (vgl. z.B. [62]). Aus den Repositories können, wie in Kapitel 3 beschrieben, verschiedene Kennzahlen ausgelesen werden. Die Auswahl der Software-Repositories, fortan auch Softwarequellen genannt, ist ausschlaggebend für die Analyse, da die Arten der gewählten Softwarequellen maßgeblich die Ergebnisse der Analysen beeinflussen.

Dabei ist zu beachten, dass eine reine Mindestanzahl an Softwarequellen nicht ausreicht, um der empirischen Untersuchung zu Aussagekraft zu verhelfen. Es muss berücksichtigt werden, dass es unterschiedliche Arten von Softwarequellen gibt, die jeweils unterschiedliche Eigenschaften aufweisen, die auch die Ergebnisse der hier vorliegenden Untersuchung beeinflussen.

2.3.1 Arten von Software Repositories

Zur Klassifizierung von Software gibt es viele verschiedene Ansätze. Die wahrscheinlich häufigste Unterscheidung ist die zwischen *freier* und *proprietärer* Software. Freie oder „Open-Source-“ Software bedeutet dabei, dass der Quellcode frei verfügbar für die Verwendung oder Modifizierung ist [63], [64]. Bei proprietärer oder „Closed-Source“-Software hingegen liegt ein Eigentumsanspruch vor und es muss in der Regel für die Verwendung eine Lizenz käuflich erworben werden. Diese darf weder modifiziert noch weiter verbreitet werden [64].

Andere Unterscheidungen von Software, wie beispielsweise die eingesetzte Domäne, wurden zunächst nicht betrachtet, weil dadurch kein Einfluss auf die Entwicklung der Redundanz vermutet wurde.

Als Grundlage für die Analyse von Software kann auch ein sogenannter Korpus verwendet werden. Ein Korpus ist eine Sammlung von Informationen, in der Regel in Form von Texten, und wird zum Beispiel häufig in der Linguistik für die Analysen von Mustern einer Sprache verwendet [65]. Für die Analysen von Programmiersprachen gibt es auch solche Korpora, ein Beispiel dafür ist der *Qualitas Corpus* [66]. Dieser wird wie andere

Korpora kuratiert und besteht aus verschiedenen Open-Source-Softwarequellen. Der *Qualitas Corpus* hat den Vorteil, dass Ergebnisse der Analysen leicht mit anderen Arbeiten verglichen werden können, weil der Ursprung der Analysen zwangsläufig gleich ist. Außerdem soll der *Qualitas Corpus* zu einer Zeitersparnis führen, weil weniger Zeit zum Ausfindigmachen einer repräsentativen Größe an Softwarequellen benötigt wird [66, S. 345].

2.3.2 Anforderung an Software Repositories

In diesem Kapitel soll erläutert werden, auf welche Kriterien bei der Auswahl von Softwarequellen für die empirische Untersuchung geachtet wurde. Die empirische Untersuchung hat dabei keineswegs den Anspruch, allgemeingültig zu sein. Allerdings sollen durch die Auswahl der Softwarequellen realistische Vertreter für die Untersuchung gewählt werden, deren Entwicklung nicht zu unüblich in der Praxis ist.

Zunächst wurden dafür Mindestanforderungen aufgestellt, die zwangsläufig für die Art der empirischen Untersuchung erforderlich sind:

- Die Software sollte **mindestens 15.000** Code-Zeilen umfassen, weil eine Code-Dopplung unter diesem Wert nicht groß genug ist, um durch Programmiersprachenentwicklung beeinflusst zu werden[2].
- Die Software sollte verschiedene **Versionen** umfassen. Diese Versionen sollten sowohl vor als auch nach der Einführung eines Programmierkonzepts in die verwendete Programmiersprache liegen, um eine Entwicklung nachvollziehen zu können.

---

[2] Dies stellte sich beim Projekt des Eclipse Checkstyle Plugins ([67], [68]) heraus, wo die SLOC teilweise auf 13.000 fielen und dabei nur noch 16 Clones gefunden wurden, die dann nur noch wenig Einblick in die Historie des Projekts geben.

Weiterhin wurden folgende Kriterien festgelegt, um die Softwarequellen untereinander vergleichbar zu machen:

- Die Softwarequellen sollten auf der gleichen Programmiersprache basieren, damit die zeitliche Entwicklung der Sprache die Softwarequellen gleichermaßen betrifft. Hierfür wurde **Java** als Programmiersprache gewählt.
- Die Softwarequellen sollen den gleichen Zeitrahmen überschneiden, um zu gewährleisten, dass sie die gleichen Programmiersprachversionen enthalten. Als Zeitrahmen wurde dafür **2003 bis 2015** gewählt, um sowohl die Einführung von Java 5 (Java Generics) als auch Java 8 (Lambdaausdrücke) abzudecken. In diesem Zeitrahmen sollte eine aktive Entwicklung stattfinden, was sich zum Beispiel durch Commits äußert.

Diese Einschränkungen bilden zunächst ein einzelnes Szenario ab. Für eine weitergefasste und allgemeingültige Analyse müssten diese und eventuell folgende Restriktionen gelockert werden.

Abschließend wurden weitere Kriterien festgelegt, um aus den verbleibenden Softwarequellen auszuwählen und den Bearbeitungsaufwand zu minimieren:

- Die Softwarequellen sollen **Open-Source**-Projekte sein, um Probleme bei Lizensierung und Veröffentlichung der Ergebnisse zu vermeiden.
- Als Versionsverwaltungssystem wurde **Git** gewählt, um eine einheitliche Versionsabarbeitung zu ermöglichen (vgl. Kapitel 3.2).
- Als Softwarequellen wurden in erster Linie Software-Projekte gewählt, die bereits **in ähnlichen Arbeiten untersucht** wurden. Dies sollte das Risiko der Umsetzung minimieren, Ergebnisse vergleichbar machen und die Möglichkeit der Anreicherung um Daten aus anderen Arbeiten ermöglichen.

Das Ergebnis der Eingrenzungen war die Auswahl der Repositories folgender Projekte als Softwarequellen, jeweils mit der Bezeichnung, wie sie in dieser Arbeit referenziert werden:

- **Checkstyle** [41]: Checkstyle ist ein Softwareentwicklungsprogramm zur Unterstützung der Durchsetzung von Formatierungs- und Programmgestaltungsstandards.
- **EclipseJDT** [69]: Die Eclipse Java Development Tools sind eine Kernkomponente einer Entwicklungsumgebung für die Entwicklung mit Java.
- **Findbugs** [39]: Findbugs ist ein Programm zur statischen Analyse von Java-Code auf Programmfehler [70]. Seit September 2017 wird Findbugs in anderem Repository unter dem Namen SpotBugs weitergeführt [71].
- **SpringFramework** [72]: Das Spring Framework ist ein Framework zur Erstellung von Betriebsanwendungen.
- **SquirrelSQL** [73]: Der SQuirreL SQL Client ist ein graphisches Java-Programm zur Betrachtung und Bearbeitung von Datenbanken mit SQL.

Die Eingrenzung der Softwarequellen birgt allerdings Fehlerquellen, weil nun sehr eingegrenzte Typen von Software betrachtet werden. Freie Software besitzt zum Beispiel eine andere Entstehungshistorie und Entwickler gehen anders mit der Einführung neuer Sprachelemente um als Entwicklungsteams in proprietärer Software.

In der Auswahl der Softwarequellen wurde sich gegen den Qualitas Corpus entschieden, weil dieser zu wenige Versionen besitzt und damit eine Historie nicht gut genug abzubilden gewesen wäre. Außerdem hat der Qualitas Corpus nicht den Vorteil, dass bereits Arbeiten zur Adaption von Programmierkonzepten dafür vorliegen.

3 Erhebungsprozess für die Analyse von Software auf Clones

Um die anfangs aufgestellte Hypothese zu bestätigen, wurde eine empirische Untersuchung durchgeführt. Ziel waren in erster Linie eine Datenerhebung und die Vorbereitung einer statistischen Auswertung (vgl. Kapitel 4).

In diesem Kapitel werden zunächst der Ursprung und der Inhalt des Ansatzes für die durchgeführte empirische Untersuchung beschrieben. Anschließend werden die einzelnen Bereiche des Prototyps in Unterkapiteln näher erklärt.

3.1 Aufbau des Erhebungsprozesses

3.1.1 Lösungsansatz

Zur Bestätigung der Vermutung, dass Redundanz ein Treiber für die Entwicklung von Programmierkonzepten ist (vgl. Kapitel 1.2), soll mittels einer empirischen Untersuchung der Code-Clone-Anteil verschiedener Softwarequellen vor und nach der Einführung eines Programmierkonzepts in eine Programmiersprache gemessen werden. Weil der Zeitpunkt der Einführung sich nicht genau festlegen lässt, wird großflächig die Historie der Software verwendet. Als Idee für die Abarbeitung großer Mengen von Daten bot sich vor allem ein sogenannter Tool-Chain-Ansatz an. Dabei geht es in erster Linie darum, vorhandene Tools einzusetzen, die in ihren Bereichen bereits erprobt und durch Spezialisierung leistungsfähig sind [74, S. 371–372]. Ähnliche Ansätze für die Auswertung und Interpretation von Code-Clones wurden bereits in verschiedenen ähnlichen Arbeiten umgesetzt (vgl. [51], [52], [75]).

Vorteil eines solchen Ansatzes sind wesentlich geringere Risiken der Funktionalität und der Kredibilität, da die verwendeten Mechanismen einzeln bereits funktionell und wissenschaftlich erprobt sind. Alternativ könnten der komplette Prozessfluss oder große Teile dessen auch selbst implementiert werden, was ein performanteres, passenderes aber auch schwerer umzusetzendes Programm zur Folge hätte.

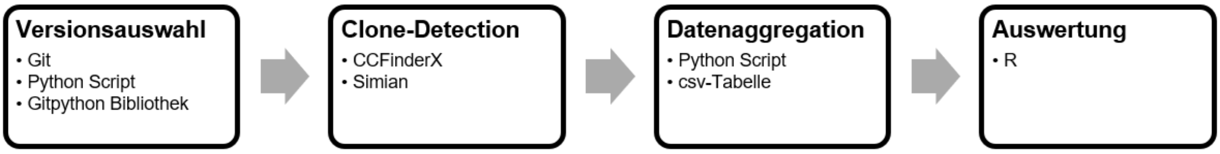


Abbildung 3: Bereiche des Analyseprozesses und verwendete Technologien.

Für diese Arbeit wurde eine solche Tool-Chain als Vorbild genommen. Die zentrale Komponente, die Clone-Detection, wird durch etablierte Tools umgesetzt, ein Großteil des restlichen Prozesses wurde selbst implementiert. Die Teilaufgaben, die für den gesamten Prozess benötigt werden, lassen sich in vier größere Bereiche einteilen. In Abbildung 3 sind die vier Bereiche und die verwendeten Tools oder Sprachen zu sehen. Wie der Name „Tool-Chain“ impliziert, sind auch hier die einzelnen Schritte linear miteinander verkettet. Diese grobe Struktur findet sich auch in der Umsetzung der Programme wieder.

3.1.2 Prozessteuerung

Der eigentliche Erhebungsprozess umfasst die ersten drei Bereiche: Versionsauswahl, Clone-Detection und Datenaggregation. Die Auswertung gehört zwar auch zum Analyseprozess, spielt aber prozessual eine besondere Rolle (siehe unten). Für die einzelnen Bereiche des Erhebungsprozesses und andere besondere Sinneinheiten wurden separate Module angelegt (siehe Abbildung 4; im CD-Anhang Ordner *Erhebungsprozess*). Der modulare Aufbau soll eine spätere Erweiterung oder Abänderung von Teilbereichen erleichtern. Gesteuert wird der Prozessfluss durch ein Pythonscript (*ProcessControl*), welches die jeweiligen Module der einzelnen Bereiche aufruft. In Abbildung 5 ist vereinfacht in einem Aktivitätsdiagramm dargestellt, wie der Erhebungsprozess abläuft. Dort wird über die Versionen einer zu untersuchenden Software iteriert. In einer Iteration wird eine Softwareversion eingestellt, anschließend werden Metriken des Quellcodes berechnet und schlussendlich in einem einheitlichen Datenformat zusammengefasst.

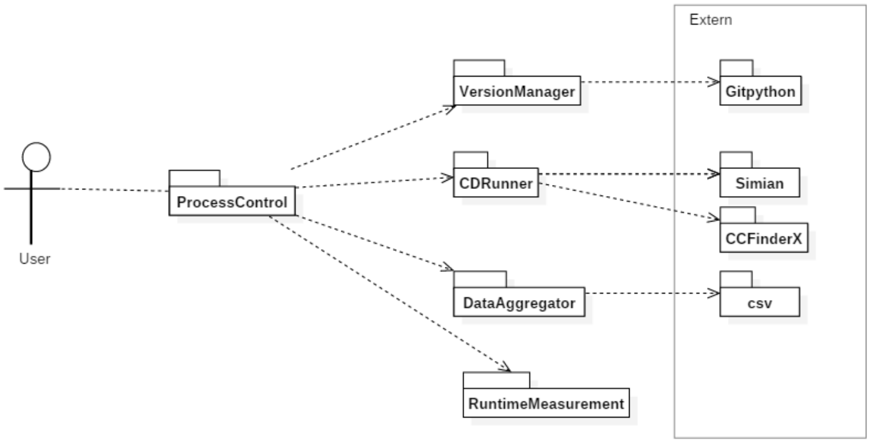


Abbildung 4: Paketdiagramm der wichtigsten Module des Erhebungsprozesses.

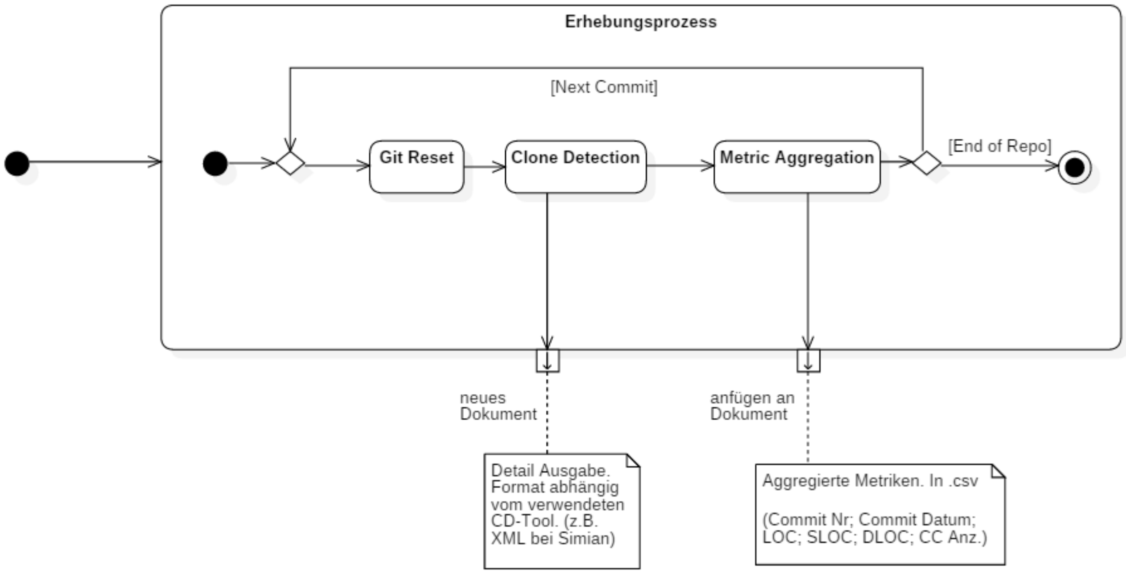


Abbildung 5: Aktivitätsdiagramm des Erhebungsprozesses

Der Auswertungsschritt ist losgelöst von der Prozessteuerung. Da die Auswertungen auf den Daten aller Iterationen einer Ausführung aufsetzen und Auswertungen oft stark individualisiert sind, werden sie manuell aufgerufen.

3.2 Umgang mit Versionierung

3.2.1 Allgemein

Weil die Historie einer Software betrachtet werden soll, bietet sich eine Analyse der einzelnen Versionen an. Zur Versionskontrolle werden in der Praxis verschiedene

Systeme eingesetzt. Die jeweiligen Versionskontrollsysteme (kurz VKS) haben dabei verschiedene Stärken und Schwächen und somit auch Einsatzgebiete. [76, S. 6–8]

In dieser Arbeit wird vor allem das VKS Git betrachtet, weil dort durch dezentrale Repositories ein einfacherer Zugriff auf die komplette Versionshistorie möglich ist. Git gilt auch als schneller im Vergleich zu anderen VKS, vor allem beim Zurücksetzen auf vergangene Versionen. Darüber hinaus nehmen Git-Repositories weniger Festplattenspeicher in Anspruch. [77, S. 39 ff.] Dies sind nützliche Eigenschaften, weil eine Vielzahl an Software-Repositorys betrachtet werden soll und die Laufzeit dafür das größte Problem darstellt (vgl. Kapitel 3.3.3). Außerdem gibt es viele Open-Source-Projekte mit Git-Repository, weil Git ursprünglich dafür entwickelt worden war und es seitdem an Popularität in diesen Bereichen gewonnen hat [76], [77]. Open-Source-Projekte bieten sich aufgrund ihres leichten Zugangs und unproblematischer Rechtverhältnisse für die Software-Analysen an (vgl. Kapitel 2.3).

Durch das Erheben von Daten aus Software Repositories fällt die Untersuchung in das Forschungsfeld *mining software repositories* (kurz MSR). In diesem Feld geht es um die Analyse von Informationen, die aus der Auswertung von Software Repositories gewonnen werden können [78]. Aufgrund der Ähnlichkeiten des Vorgehens können auch Forschungsergebnisse aus anderen Arbeiten verwendet werden, um die Kennzahlen mit zusätzlichen Informationen anzureichern.

Für das automatisierte Laden der Software-Versionen wurde ein weiteres Pythonscript verwendet. Der Umgang mit Git wurde durch die Bibliothek *Gitpython* [79] erleichtert. Vor der Ausführung wird eines der nach den Kriterien aus Kapitel 4.2 ausgewählten Software Repositories geklont. In der Ausführung wird das Repository einer Softwarequelle zunächst auf den *origin/master* (Anfangszustand des Hauptentwicklungszweigs) hart zurückgesetzt, um etwaige vorangegangene Änderungen durch das Programm oder einen Benutzer vollständig zu verwerfen. Anschließend wird schrittweise auf einen zurückliegenden Commit zurückgesetzt. Ein Commit stellt dabei den Stand einer bestimmten Version der Software dar [80]. Ein Commit besitzt eine Vielzahl an Metadaten, die jeweils ausgelesen werden können, wie zum Beispiel Datum, Autor oder eine Markierung (engl. *tag*). Für die einfachste Fassung der Untersuchung, das Ermitteln einer Historie von Code-Clones für eine Software, ist lediglich das Datum einschließlich Uhrzeit für die Einordnung im Zeitstrahl von Relevanz und wird für die Datenaggregation gespeichert.

3.2.2 Commit-Filter

Bei der Ausführung hat sich gezeigt, dass nur ein geringer Prozentsatz an Commits notwendig ist, um einen historischen Einblick zu erlangen (vgl. Kapitel 4.3.2). Auf die Laufzeit hat dies aber trotzdem großen Einfluss, weil die Clone-Detection im aktuellen Stand keine Unterschiede (vgl. Kapitel 3.3.3) betrachtet, sondern pro Version die komplette Software analysiert. Dadurch steigt die Laufzeit für die Analysen für jeden betrachteten Commit deutlich an. Allein im Szenario aus Kapitel 4.3.2 dauerte die Ausführung bei der Untersuchung von jedem 50. Commit 719 Sekunden, bei jedem zehnten Commit schon 2743 Sekunden und die Analyse von jedem Commit wurde nach etwa fünf Stunden unterbrochen. Das Ergebnis der drei Analysen zeigt jedenfalls bis auf leichtes Rauschen sehr ähnliche Graphen.

Aus diesem Grund bietet es sich an, die Commits zu filtern, um ohne wesentlichen Informationsverlust, die Laufzeit zu verringern. Für die Auswahl der relevanten Commits gibt es verschiedene Ansätze. Denkbar wären zum Beispiel Filterungen anhand von Metadaten der Commits, wie beispielsweise veränderte Code-Zeilen oder anhand des Datums. Code-Zeilen sind dabei aber zu unspezifisch, um beispielsweise Tests oder Nicht-Source-Code auszuschließen und ein weiter Präprozessor wäre notwendig, was wiederum das ursprüngliche Ziel der Laufzeitverringerung nicht unterstützt. Anhand des Datums auszuwählen ist in den meisten Projekten aufgrund der unregelmäßigen Entwicklung mit zu viel Informationsverlust oder -überlagerung verbunden.[3]

In der Arbeit von Goon et al. [52] wurde mit dem *„git log --first-parent“*-Befehl gefiltert, um nur den jeweils ersten Commit nach einem *merge* (Zusammenführung von Bearbeitungszweigen) zu verwenden. Dies ist zwar eine inhaltsorientiertere Möglichkeit der Filterung, erwies sich aber als zu inkonsistent, weil nicht in jedem Software-Projekt regelmäßig Zweige und *merges* verwendet werden und deshalb die Daten sehr zerstückelt werden. In anderen Arbeiten, wie unter anderem von Parnin et. al. [5], werden gar keine Filter verwendet. Dies funktioniert aber nur, weil durch andere Analysemethoden kürzere Laufzeiten auftreten und kleinere Datenmengen produziert werden.

---

[3] Wenn beispielsweise in einer Versionshistorie jede Woche ein Commit betrachtet wird und in einer Woche 100 Commits gemacht werden und die nächsten fünf Wochen hingegen gar kein Commit gemacht wird, dann werden 99 Commits trotz eventueller Code-Clone-Verminderung nicht betrachtet und stattdessen ein und derselbe Commit sechs Mal in den Datensatz aufgenommen.

Nach der Implementierung verschiedener dieser Ansätze hat sich herausgestellt, dass ein einfaches Überspringen einer festen Zahl an Commits in den meisten Fällen das beste Verhältnis aus Laufzeitverringerung und dem Verlust relevanter Informationen bietet. Daraus resultierend wurden Erhebungsprozesse mit jeweils zehn übersprungenen Commits durchgeführt. Um im Nachhinein eine höhere Abdeckung zu erreichen, wurde teilweise in erneuter Ausführung eine andere Schrittweite wie neun oder vier übersprungene Commits (im Gegensatz zu zehn oder 20) mit anderer Abdeckung ausgewählt. Die daraus entstehenden Kennzahlen lassen sich problemlos in der Datenaggregation und -auswertung zu den bestehenden eingliedern.

3.3 Clone-Detection

Sobald die Software auf die richtige Version gebracht ist, gilt es, die Code-Clones innerhalb der Software aufzuspüren. Das Finden von Code-Clones innerhalb einer Software wird in der Literatur in der Regel als *Clone-Detection* bezeichnet. [13], [42], [81] Aufgrund des häufigen Einsatzes in der Praxis haben sich dafür verschiedene Herangehensweisen herausgebildet und es gibt eine Vielzahl an Tools, die eine solche Funktion übernehmen können. [42, S. 470–471]

Folgend werden die Unterschiede verschiedener Clone-Detection-Tools im Hinblick auf die Auswahl für den Erhebungsprozess vorgestellt und anschließend wird gezeigt, wie Clone-Detection in der Untersuchung angewendet wurde.

3.3.1 Arten und Vertreter

In diesem Kapitel sollen die Unterschiede der verschiedenen Clone-Detection-Tools analysiert werden, um eine Auswahl für diese oder eine andere, darauf aufsetzende Untersuchung zu erleichtern. Die meisten Clone-Detection-Tools basieren auf einem von fünf Ansätzen: text-, token-, baum- oder graphbasierte Ansätze oder Hybride dieser [42], [82].

Im Folgenden werden die Ansätze dieser häufigsten Techniken näher beschrieben ([42], [82]).

Im string- oder auch textbasierten Ansatz wird der untersuchte Code nicht in ein anderes Format übersetzt. Die Extraktion von ungewollten Zeichen, die Segmentierung sowie der Vergleich der Segmente findet im Format des ursprünglichen Sourcecodes statt. Dafür werden in neueren Ansätzen vor allem die Hashwerte der Segmente verglichen und durch Angleichungen können mit statistischen Annäherungen sogar Typ-3-Clones approximiert werden (z.B. [83]).

Die sogenannten lexikalischen oder *tokenbasierten* Verfahren übersetzen ähnlich wie ein Compiler die Strukturen in aufeinanderfolgende Tokens. In der Regel werden Identifizierer und Literale aussortiert und die verbleibenden syntaktischen Elemente können sehr performant über Hashvergleiche auf Duplikate geprüft werden. Damit lassen sich sehr einfach Typ-2-Clones identifizieren (z.B. [13]).

In den syntaxbasierten Methoden soll der Code auf eine Ebene abstrahiert werden, so dass vorwiegend die syntaktischen Eigenschaften erhalten bleiben. Eine Möglichkeit dafür sind die sogenannten *baumbasierten* Verfahren, in denen in der Regel ein abstrakter Syntaxbaum (*abstract syntax tree*, kurz AST) aufgestellt wird. Mit Hilfe dieser Bäume können relativ performant Vergleichsverfahren ausgeführt werden, indem Unterbäume miteinander verglichen werden (z.B. [84]).

Eine semantikorientierte Methode sind die *graphbasierten* Clone-Detections. In diesen wird ein Programmabhängigkeitsgraph (*programm dependency graph*, kurz PDG) aufgestellt, der Informationen über den Kontroll- und Datenfluss enthält und damit auch semantisches Hintergrundwissen behält. Das kann vor allem qualitative Vorteile bergen, wie das Erkennen von Code-Clones höheren Typs (z.B. [85]).

Darüber hinaus gibt es auch *Hybride*, die Kombinationen der verschiedenen anderen Techniken darstellen. Dort haben sich vor allem die metrik-basierenden Ansätze hervorgetan, die aus PDG oder AST Metriken generieren und mit diesen Duplikate finden (z.B. [86]).

Seit der Einführung dieser Klassifizierungen wurden aber auch Clone-Detection-Tools vorgestellt, die sich in keine der vorgenenannten Kategorien einordnen lassen, zum Beispiel ein auf Deep-Learning basierender Ansatz [87] entwickelt im Jahr 2017. Diese Arten von Clone-Detection-Tools wurden zunächst nicht betrachtet, weil sie noch nicht genug in anderen Arbeiten erprobt wurden.

Für die Evaluation und den Vergleich von Clone-Detection-Tools wurden bereits verschiedenste Arbeiten angefertigt [42], [81], [82]. Diese liefern eine gute Grundlage für die Auswahl eines Tools für den Erhebungsprozess. Die Arbeit von Roy et al [42] ist dabei von den untersuchten schon rein von der Anzahl der betrachteten Clone-Detection-Tools und den verwendeten Kriterien am umfangreichsten. In der Arbeit werden vor allem die

Grundeigenschaften verglichen, wie zum Beispiel die Sprachunterstützung oder externe Abhängigkeiten [42, S. 477–483]. Darüber hinaus wird aber auch verglichen, was für Arten von Clones mit welcher Präzision erkannt werden [42, S. 483–490]. Anhand dieser und Kriterien anderer Arbeiten kann eine Vorauswahl für die empirische Untersuchung getroffen werden, in dem Mindestanforderungen an das Clone-Detection-Tool aufgestellt werden. Die für diese empirische Untersuchung (ohne Erweiterungen) gesetzten Anforderungen können einschließlich Begründung der Tabelle 3 entnommen werden. Die aufgezeigten Anforderungen und Ausprägungen entstammen Roy et al [42]. Allerdings sind die meisten Arbeiten zur Evaluation von Clone-Detection-Tools relativ alt und spiegeln damit nur bedingt den aktuellen Stand der Technik wieder.

Anforderung	***Ausprägung***	***Begründung***
Unterstützte Programmiersprache	Java	Java ist eine weit verbreitete Sprache und hat sich auch gerade in den letzten zehn Jahren stark weiterentwickelt (vgl. Kapitel 2.1.2). Für eine prototypische Untersuchung ist Java deshalb geeignet, weil durch Einführung von Programmierkonzepten eine Beeinflussung von Code-Clones erwartet werden könnte.
Verfügbarkeit	Open Source	Ein Open Source Clone-Detection-Tool soll verwendet werden, um Publikation und Bezug sowie einen späteren Austausch zu erleichtern.
Benutzungsoberfläche	Konsole	Zur automatisierten Anbindung in die Tool-Chain sollte das Clone-Detection-Tool über einen Konsolenbefehl ausführbar sein.
Clone Type	Min. Typ 2	Typ-1- und Typ-2-Clones werden von nahezu allen Tools erkannt [42, S. 480]. Als Mindestanforderung für ein Clone-Detection-Tool kann dies also durchaus gefordert werden.

Tabelle 3: Anforderungen an Clone-Detection-Tool für die empirische Untersuchung

Mit den Anforderungen aus Tabelle 3 konnte der Pool der auswählbaren Clone-Detection-Tools zwar zunächst eingedämmt werden, aber eine Auswahl allein daraus ist noch nicht möglich. Als ausschlaggebendes Kriterium war es nun sehr wichtig, ein Clone-Detection-Tool anzubinden, welches bereits in einem ähnlichen Kontext erprobt wurde. Dadurch, dass das Tool bereits verwendet wurde, soll das Risiko einer Implementierung minimiert werden. Diese Verwendung sollte dabei relativ aktuell sein und einen ähnlichen Untersuchungsaufbau besitzen, daher wurden Arbeiten der letzten fünf Jahre betrachtet, die sich mit Code-Clone-Analysen von Softwareversionshistorien beschäftigen. Um einen solchen Vertreter zu finden, wurde eine Literaturrecherche, wie in Kapitel 1.3 beschrieben, durchgeführt. Dies führte zu einer Auswahl der beiden Clone-Detection-Tools Simian und CCFinderX wie in Kapitel 3.3.2 beschrieben.

3.3.2 Eigene Verwendung

In der Tool-Chain wurden sowohl Simian als auch CCFinderX eingesetzt. Durch die Verwendung zweier Clone-Detection-Tools sollte überprüft werden, ob das verwendete Clone-Detection-Tool einen großen Einfluss auf die Ergebnisse hat.

3.3.2.1 Simian

Simian [12] wurde ausgewählt, weil es bereits in einem ähnlichen Projekt verwendet wurde. In der Arbeit zu ClonEvol [75] wurde eine Tool-Chain beschrieben, in der Code-Clones in verschiedenen Versionen betrachtet werden können. Diese beiden Aspekte waren auch Ziel der Untersuchung dieser Arbeit und die Veröffentlichung ClonEvols liegt im selbst gesetzten Rahmen von fünf Jahren.

```
cd srcPath
simian-2.5.6 -includes=**/*.java -excludes=**/*Test.java -formatter=xml
threshold=6 > targetPath
```

Programmtext 1: Konsolenaufruf des Tools Simian zur Clone-Detection

In Programmtext 1 ist zu sehen, wie ein Aufruf innerhalb des Erhebungsprozesses aussieht. Dabei wird die Software des *srcPath* analysiert und eine Ausgabe im *targetPath* generiert. Der Parameter *includes* spezifiziert dabei die Ordnerstrukturen, die betrachtet werden sollen und welche Dateiänderungen relevant sind. In diesem Fall findet eine Analyse von Java-Code[4] statt. Dafür wählt ein Präprozessor alle Dateien mit den gewünschten Endungen aus den Unterordnern des Pfades aus. Mit *excludes* werden hier noch Testfälle ausgeschlossen, denn diese neigen dazu, viele Dopplungen zu enthalten, die aber weniger Einfluss auf die Code-Qualität haben. Das Ausschließen von Testfällen ist relativ weit verbreitet (z.B. [52, S. 48]) und wurde deswegen zumindest in ersten Ausführungen ebenfalls angewendet. Bei späterer Betrachtung hat sich jedoch gezeigt, dass die Exklusion von Tests vernachlässigt werden kann, weil mit dieser Untersuchung keine Code-Qualität untersucht werden soll, sondern die Entwicklung der Mächtigkeit einer Programmiersprache, die unter anderem auch Einfluss auf Testfälle hat. In Java konnten vermutlich durch Annotationen oder Testhilfebibliotheken wie Mockito [88] oder TestNG [89] Code-Duplikationen in den Testfällen vermindert werden. Abgesehen davon

---

[4] Die gewählte Programmiersprache hat im kompletten Analyseprozess lediglich auf das Clone-Detection-Tool einen Einfluss und Simian unterstützt verschiedene Sprachen, wie zum Beispiel auch C++ oder C#.

ist der *excludes* Befehl auch nicht mächtig genug, um beispielsweise das Vorkommen von Test-Annotationen oder variierende Klassennamen wie zum Beispiel „ProcessFlowTest2.java“ durch Regex-Ausdrücke zu filtern.

Des Weiteren kann über den *threshold*-Parameter die Mindestanzahl an Zeilen, die für eine Identifikation als Code-Clone notwendig sind, angegeben werden. Die Unterschiede der Variation sind in Kapitel 4.3.3 näher beschrieben.

Simian ließ sich relativ leicht in den Prozessfluss integrieren und sein Einsatz war sehr robust und wenig fehleranfällig. In den Laufzeitanalysen aus Kapitel 4.3.1 zeigte sich jedoch, dass die Laufzeit für größere Projekte stark zunimmt. Dies und der Nachteil, dass lediglich Typ-1- und Typ-2-Clones gefunden werden können [42, S. 487], sprechen gegen einen Einsatz in größer skalierten Projekten.

3.3.2.2 CCFinderX

Als zweites Clone-Detection-Tool sollte für einen stärkeren Kontrast ein anderer Ansatz gewählt werden. Den stärksten Kontrast zum textbasierten Ansatz in Skalierbarkeit und Genauigkeit bilden dabei tokenbasierte Ansätze, die sehr gut skalieren, dafür aber nicht an die hohe Genauigkeit der textbasierten herankommen [82, S. 155]. CCFinderX [13], [90] folgt einem solchen tokenbasierten Ansatz, ist in Benchmarks mit populären Clone-Detection-Tools als konkurrenzfähig bestätigt [91, S. 1162 ff.] und findet dank seiner effizienten Arbeitsweise in vielen, auch aktuellen, Arbeiten seinen Einsatz. Beispielsweise in der empirischen Untersuchung der Arbeit von Goon et al. [52] wurden mit CCFinderX Code-Clones von Open-Source-Projekten in ihrer Versionshistorie gemessen. Dies ähnelt stark der in dieser Arbeit angestrebten empirischen Untersuchung, weswegen CCFinderX als für diesen Kontext erprobt gewertet und damit für die Implementierung ausgewählt wurde.

```
cd interimPath
ccfx d java -d directoryPath -n interimPath -o (interimPath + filename)
ccfx m (interimPath + filename) -w -o (targetPath + filename)
```

Programmtext 2: Analyseaufruf einer Software mit CCFinderX und Metrikberechnung

In Programmtext 2 sind die für die Auswertung einer Software im Erhebungsprozess verwendeten Aufrufe aufgezeigt. Dabei werden drei verschiedene Pfade verwendet: Der *directoryPath* zeigt auf das zu untersuchende Softwareverzeichnis; der *interimPath* dient der Speicherung aller Zwischenergebnisse von CCFinderX und *targetPath* ist der Zielpfad für die Detailausgabe der Clone-Detection. Der Bezeichner *filename* dient sowohl der eindeutigen Identifizierung des Ergebnisses als auch der Zuordnung der Zwischenschritte zu diesem, falls ein Fehler aufgetreten sein sollte.

Mit *ccfx d* werden der Präprozessor (in diesem Fall für Java) und die eigentliche Clone-Detection aufgerufen. Mit dem Befehl *ccfx m* wird im Anschluss eine Datei mit Metriken aus der intermediären Datei generiert (vgl. Anhang S. XI). In diesen Metriken sind LOC (lines of code), SLOC (*source lines of code*) CLOC (*cloned lines of code*) und eine Zuordnung zur ursprünglichen Klasse (*file identificator*, FID) enthalten. In Anlage S. XI ist eine Detailausgabe des CCFinderX zu sehen, die den hier beschriebenen und generierten Metriken entspricht.

In Kapitel 6.3.3 schneidet CCFinderX im Vergleich mit Simian tatsächlich besser in Laufzeit und Anzahl der gefundenen Clones ab. Für einen klaren Vergleich der Güte reichen die Ergebnisse allerdings nicht aus. Dennoch zeigt der Einsatz der beiden Clone-Detection-Tools hier, dass es tatsächlich beträchtliche Unterschiede zwischen den verwendeten Mechanismen gibt und deswegen eine Auswahl geeigneter Clone-Detection-Tools für eine empirische Untersuchung größerer Datenmengen umso entscheidender für die Validität der Forschungsergebnisse ist. Vorerst konnte lediglich festgestellt werden, dass sich CCFinderX grundsätzlich besser für die Datenerhebung als Simian eignete.

Bei der Bearbeitung unterschiedlicher Versionen einer Software trat allerdings öfter ein Problem mit dem Präprozessor von CCFinderX auf. Scheinbar ließen sich bestimmte Zwischenrechnungen beim Versionsübergang nicht zurücksetzen. Dies sorgte teilweise für fehlerhafte Ergebnisse entlang der Versionshistorie. Zur Behebung dieses Problems wurden speziell bei der Ausführung von CCFinderX zusätzliche Bereinigungen für das Repository und andere zwischenspeichernde Ordner ausgeführt. Aus Gründen der Verlässlichkeit und Ausführungsrobustheit wurden die Datenerhebungen für die Auswertungen in Kapitel 4 weitestgehend mit Simian durchgeführt.

Bei einer Ausweitung der vorliegenden empirischen Untersuchung oder ähnlichen Untersuchungen sollte CCFinderX dennoch berücksichtigt werden, da er durch lineare Laufzeitkomplexität und gute Skalierung für große Systeme immer noch ein leistungsstarkes Tool darstellt [13]. Sollte sich das Problem also mit späteren Versionen von CCFinderX legen, ist es mit großer Wahrscheinlichkeit eine bessere Alternative zu Simian.

3.3.3 Laufzeitproblem und Lösungsansätze

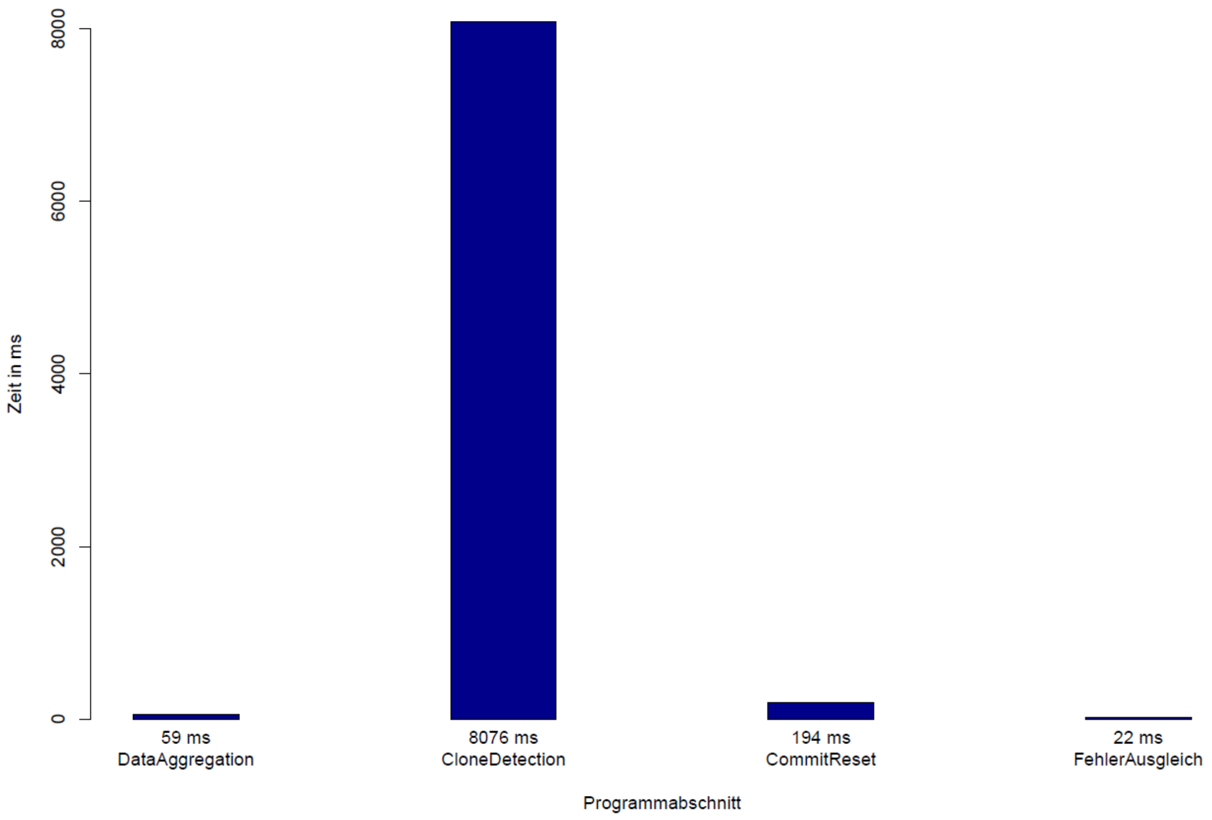


Abbildung 6: Laufzeiten des Datenerhebungsprozesses je Iteration aufgeschlüsselt nach Programmabschnitten

In Abbildung 6 sind die Laufzeiten der einzelnen Programmabschnitte zu erkennen. Grundlage für die Zeitmessung war die Datenerhebung des Checkstyle-Projekts mit etwa 120.000 SLOC (ohne weitere Dateifilterung) durch Simian mit einem Grenzwert von mindestens sechs Zeilen für einen Code-Clone. Die Laufzeiten sind gemittelt über die jüngsten 200 Commits (Stand: 26.11.2017) und wurden mit einem Computer, wie in Anlage S. IX beschrieben, berechnet. Am Diagramm ist zu erkennen, dass die Clone-Detection mit Abstand die meiste Zeit in Anspruch nimmt. In kleineren Projekten ist der Unterschied zwar weitaus geringer (800ms bei 15.000 SLOC), aber dennoch bildet die Clone-Detection im äußerst zeitintensiven Analyseprozess einen Flaschenhals, weswegen gerade hier effiziente Algorithmen wünschenswert sind. Das *multi-threading* von CCFinderX [90] ist dabei nur ein kleiner Schritt, weil dadurch die Laufzeit im optimalen Fall nur maximal durch die Anzahl der reellen und virtuellen Kerne geteilt werden kann (vgl. Kapitel 4.3.1 Laufzeiten). Betrachtet man im Gegensatz dazu beispielsweise nur ein Delta der Versionen für die Clone-Detection, würde der Analyseraum der Clone-Detection auf die wenigen Code-Zeilen eines Commits und den davon tangierten Stellen im Programm schrumpfen, anstatt hunderttausende Zeilen des kompletten Projekts zu betrachten. In der Arbeit [51] wurden solche Ansätze umgesetzt, was zeigt, dass eine effiziente Historienanalyse durchaus denkbar ist. Der Analyseraum würde sich in den meisten betrachteten Projekten um ein Tausendfaches verringern und damit vermutlich auch die Laufzeit der Clone-Detection. Leider eigneten sich diese Algorithmen nicht für die Verwendung in der vorliegenden empirischen Untersuchung.

In dem Feld der Clone-Evolution (z.B. [46], [48], [49], [51]) geht es speziell um die Nachverfolgung von Code-Clones über Softwareversionen hinweg. Ziel ist dabei in der Regel die Untersuchung der Eigenschaften von Code-Clones. Mit dem Nachverfolgen von Code-Clones bietet sich aber auch eine neue Möglichkeit als Alternative zum Eliminieren von Code-Clones, weil dadurch zum Beispiel das große Problem des zusätzlichen Wartungsaufwands [92] minimiert werden könnte. Aber auch für die Untersuchung der Entwicklung von Programmierkonzepten könnte diese Technik eingesetzt werden, weil so der Lebenslauf eines Code-Clones bis zu seiner Eliminierung nachverfolgt werden kann. Wenn ein Code-Clone sich lange in einer Software hält, aber dann doch eliminiert wird, spricht das eventuell für eine Änderung der Umstände, wie beispielsweise der Einführung eines neuen Programmierkonzepts. Solch eine Art der Untersuchung könnte durch eine zielgerichtete Analyse leistungstechnische oder qualitative Vorteile mitbringen.

Abschließend ist zu erwähnen, dass der aktuelle Untersuchungsaufbau für größere Projekte eher ungeeignet ist. Für weiterführende Analysen sollten entweder der Untersuchungsraum durch die Betrachtung von Unterschieden zwischen Commits verringert werden oder gänzlich andere Clone-Detection-Tools, wie SourcererCC [91], CloneWorks [93] oder CCLearner [87], eingebunden werden, die durch neue Ansätze zwar größeres Umsetzungsrisiko bergen, aber dafür bessere Skalierung versprechen.

3.4 Datenaggregation

Die aus Clone-Detection-Tools entstehenden Daten, sind in der Regel sehr große Detailaufnahmen[5]. Das liegt daran, dass solch ein Tool als Nebenprodukt auch viele Informa-

---

[5] Beispielsweise im EclipseJDT-Projekt mit etwa 200.000 LOC entstanden Detailausgaben von bis zu 11 MB Größe. Auf 3.500 untersuchten Commits entstanden Daten im Umfang von 16,3 GB für eine Analyse.

tionen produziert, die im Rahmen der Untersuchung, abgesehen von Erweiterungsmöglichkeiten (vgl. Kapitel 5.2), zweitrangig sind. So wird zum Beispiel beschrieben, in welchen Zeilen und Dateien sich Clones befinden, was aber innerhalb der hier vorgenommenen Auswertung des gesamten Projekts nicht betrachtet wurde. Ein weiteres Problem ist, dass unterschiedliche Clone-Detection-Tools unterschiedliche Formate für die Erstellung solcher Dateien haben.

Wegen dieser Probleme werden in jedem Erhebungsschritt die wichtigsten Metriken aus der jeweiligen Datei ausgelesen und in einem einheitlichen Datenformat zusammengetragen.

Im ersten Schritt wurden Kennzahlen von Interesse identifiziert. Für die ersten Ausführungen des Analyseprozesses wurden grundlegende Kennzahlen gewählt, die erstens mit wenig Aufwand den Detailergebnissen der meisten Clone-Detection-Tools entnommen werden können (um diesbezüglich einen späteren Austausch möglich zu machen) und zweitens die Entwicklung von Code-Clones über eine Historie hinweg repräsentieren. Aus diesen Anforderungen folgte die Auswahl dieser Kennzahlen für die aggregierende Datei:

- Commit-Datum: Zeitpunkt des Commits für die Erstellung einer Historie
- **SLOC**: *source lines of code* sind die vom Clone-Detection-Tool als für die Analyse relevant erachteten Code-Zeilen (ausgeschlossen sind Leerzeichen, Kommentare etc.).
- **DLOC**: *duplicated lines of code* sind alle Code-Zeilen, die mehrfach im untersuchten Verzeichnis (innerhalb einer gesetzten Reichweite) vorkommen.

Aus DLOC und SLOC wird später ein Verhältnis der duplizierten Zeilen zu den signifikanten Code-Zeilen berechnet. Statt des Code-Clone-Zeilenverhältnisses hätte unter den oben genannten Anforderungen auch die reine Anzahl von Code-Clones gereicht. Ohne ein Verhältnis zur Größe der untersuchten Software ist aber diese Kennzahl allein weniger aussagekräftig oder vergleichbar. Für andere Fragestellungen wurden in dieser Datei später auch LOC (Lines of Code), Anzahl an Code-Clones und Commit-Nummer erfasst.

Die dafür notwendigen Daten werden einerseits aus den Metadaten der Commits gewonnen und andererseits aus den Detailausgaben der Clone-Detection-Tools aggregiert.

In Simian ist eine solche Detailausgabe eine XML-Datei, wie im Programmtext 3 Anhang S. X zu sehen. Die für die aggregierte Tabelle benötigten Daten können unverändert der *summary* entnommen werden. Bei CCFinderX hingegen ist es eine TSV-Datei (siehe Programmtext 5 Anhang S. XI) mit den Kennzahlen verteilt auf die jeweiligen zugrundeliegenden Dateien. In einem Script für die Datenaggregation werden SLOC, LOC und die dort genannten CLOC (cloned lines of code; in der Tabelle dann DLOC) aus allen Dateien für das gesamte Projekt aufaddiert.

Die gesammelten Daten aus Commit und Detailausgabe werden dann in einer CSV-Datei gespeichert (vgl. Programmtext 6 Anhang S. XII). Mit jeder untersuchten Version der Software kommt eine weitere Zeile hinzu und so liegen alle für die Auswertung benötigten Kennzahlen zentralisiert an einem Ort. Die Daten-Tabelle ist für alle Commits nur wenige KB groß und kann problemlos für weitere Analysen oder die Erstellung von Graphen verwendet werden.

4 Auswertung der Messungen

4.1 Vorgehensweise der Auswertung

In diesem Kapitel wird die Auswertung der Daten aus Kapitel 3 beschrieben. Hierfür wurde die statistikorientierte Programmiersprache R [14] verwendet. Mit dieser wurden jeweils die aus dem Erhebungsprozess generierten Dateien, wie zum Beispiel CSV-Dateien mit Laufzeiten oder mit Ergebnissen der Clone-Detection, eingelesen. Anschließend wurden daraus für die folgenden Untersuchungen Kennzahlen und Grafiken generiert.

Abgesehen von den vergleichenden Untersuchungen aus Kapitel 4.3 wurden für die Erhebung einheitliche Einstellungen verwendet. Als Clone-Detection-Tool wurde Simian mit einem Schwellwert von sechs Zeilen eingesetzt. Für die Commit-Filterung wurde mindestens jeder 20. Commit[6] betrachtet. Abgesehen von Vergleichen der Clone-Detection-Tools wurde außerdem der *„excludes“*-Filter von Simian angewendet, um Testklassen zumindest weitestgehend ausschließen zu können. Darüber hinaus wurde, wenn nicht anders angegeben, von Projekten jeweils nur ein Kernmodul betrachtet, um gewollte Dopplungen von Modulen beispielsweise durch Varianten oder Produktlinien (vgl. [94, S. 3]) oder funktionale Redundanz (vgl. Kapitel 2.2.2.1) zu ignorieren. Für die Berechnungen wurde ein PC mit den im Anhang auf Seite IX beschriebenen Konfigurationen verwendet.

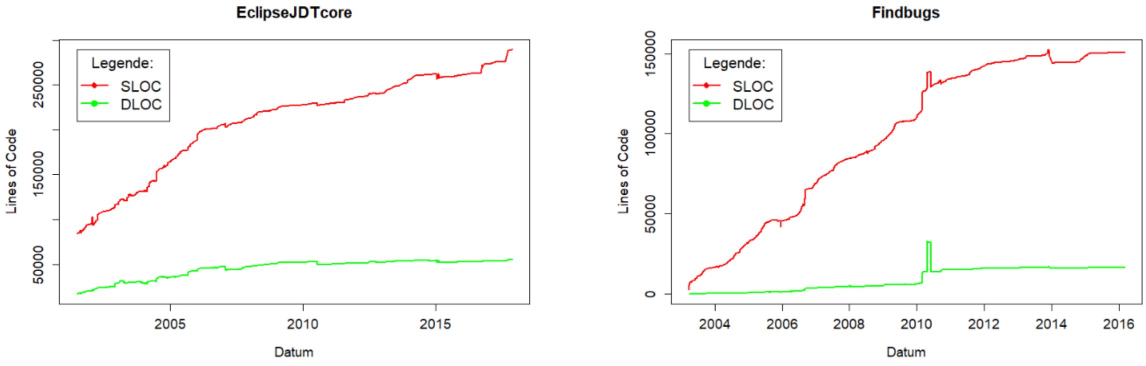
Die verwendeten R-Scripts, die für die Erstellung von Graphen oder Berechnungen eingesetzt wurden, finden sich im CD-Anhang im Ordner *Auswertungsscripts*. Die dafür verwendeten Daten liegen in den Ordnern zu *Clone-Historien* oder *Laufzeiten*.

4.2 Betrachtung von Code-Clone-Historien

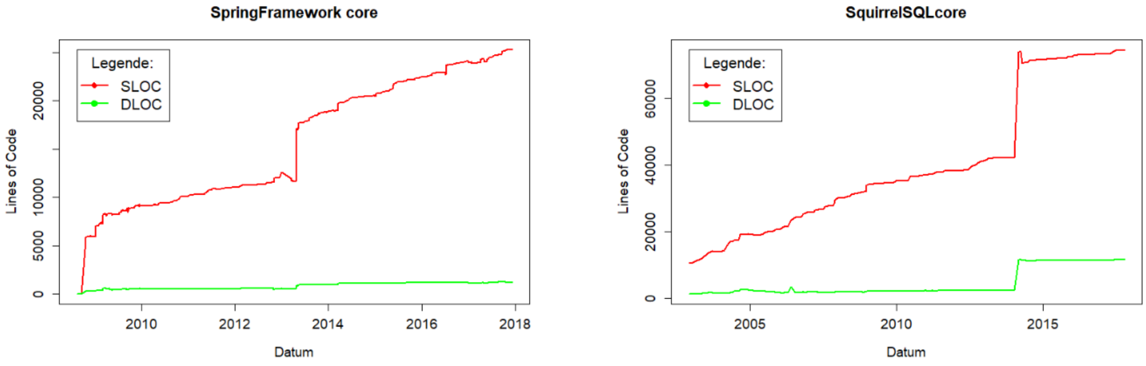
Ein großes Ziel dieser Arbeit war vor allem eine historische Darstellung von Code-Clones in einer untersuchten Software. Die dafür benötigten Daten wurden, wie in Kapitel 3 beschrieben, erhoben.

---

[6] Ist ein Commit fehlerhaft und damit nicht auswählbar gewesen, wurde mit Schrittweite eins jeder Commit getestet und der nächste verwendbare Commit gesucht.



a) EclipseJDT Kernmodul *b) Findbugs*



c) SpringFramework Kernmodul *d) SquirrelSQL Kernmodul*

Abbildung 7: Graphen signifikanter und duplizierter Programmierzeilen als Ergebnis einer Clone-Detection

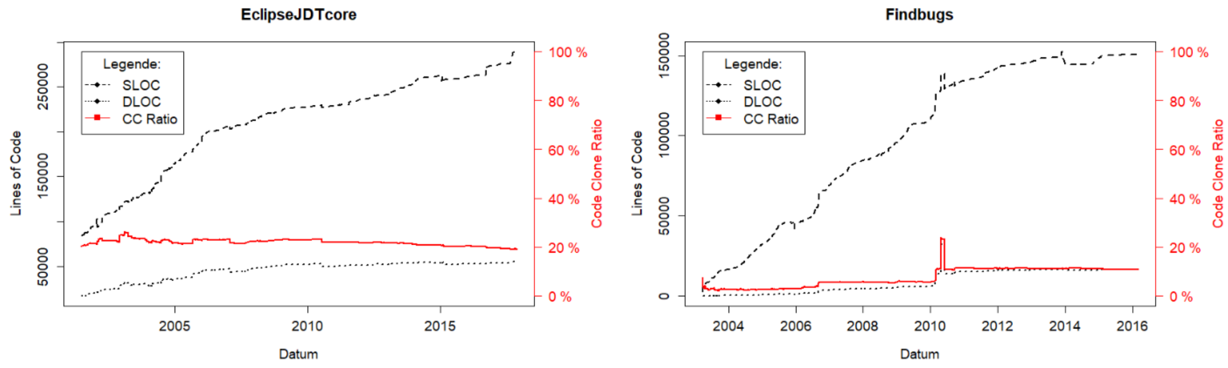
In Abbildung 7 können verschiedene Graphen betrachtet werden, die die reine Veränderung der Zeilenzahlen zeigen. SLOC sind dabei die *source lines of code*, also die Programmzeilen, die vom Clone-Detection-Tool für die Analysen als signifikant betrachtet werden (Ausschluss von Freizeilen, Kommentaren etc.). DLOC sind die *duplicated lines of code*, also (signifikante) Zeilen, die mehrfach auftreten. Dabei wird jedes Vorkommen gezählt, das heißt wenn ein bestimmter Code-Abschnitt *X* zweimal vorkommt, dann entspricht DLOC für *X* der Anzahl der Zeilen von *X* multipliziert mit zwei[7].

Anhand der Graphen kann eine Korrelation von SLOC und DLOC erkannt werden: Gibt es einen Anstieg der SLOC, steigen in der Regel auch die DLOC. In Abbildung 7 ist dies vor allem in den Graphen a und b zu beobachten, wo SLOC und DLOC stetig ansteigen

---

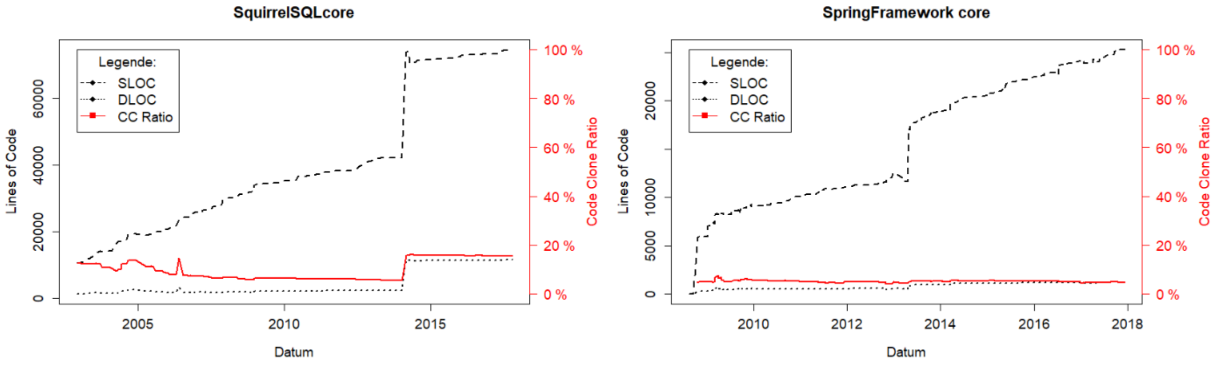
[7] Eine Quelle für diese DLOC-Definition ist in Simian und CCFinderX nicht gegeben, wurde aber im Rahmen der Untersuchung empirisch anhand der Detailausgaben geprüft.

und auch Ausreißer ähnlich verlaufen. Für den Verlauf aus Abbildung 7 a, der Code-Clone-Historie vom EclipseJDT-Kernmodul, lässt sich ein Korrelationskoeffizient nach Pearson [95] in Höhe von 0,985 berechnen. Das beweist zwar keinen kausalen Zusammenhang, legt aber eine positiv lineare Abhängigkeit nahe. Erklärt werden kann dies zum einen dadurch, dass bei mehr Programmcode, der zu Verfügung steht, auch mehr Raum für Code-Clones existiert. Die Korrelation der beiden Kennzahlen variiert je nach Projekt. Zum Beispiel im Projekt Findbugs (Abbildung 7 b) liegt der Korrelationskoeffizient nur noch bei etwa 0,9. Diese Unterschiede könnten auf unterschiedlichen Codequalitätsphilosophien basieren und unterstreichen die Wichtigkeit der Betrachtung verschiedener Softwarequellen als Basis für die Untersuchung. Trotzdem war für alle betrachteten Projekte der Pearson-Korrelationskoeffizient stets zwischen 0,9 und 1 (vgl. Tabelle 4 SLOC:DLOC), was eine allgemeine lineare Korrelation der beiden Größen vermuten lässt. Durch diese Abhängigkeit ist eine Entwicklung der Code-Clones schwer auszumachen, weil Tendenzen maßgeblich durch den Umfang des Projekts sowie dessen Zuwachs beeinflusst werden.



a) EclipseJDT Kernmodul

b) Findbugs



c) SquirrelSQL Kernmodul

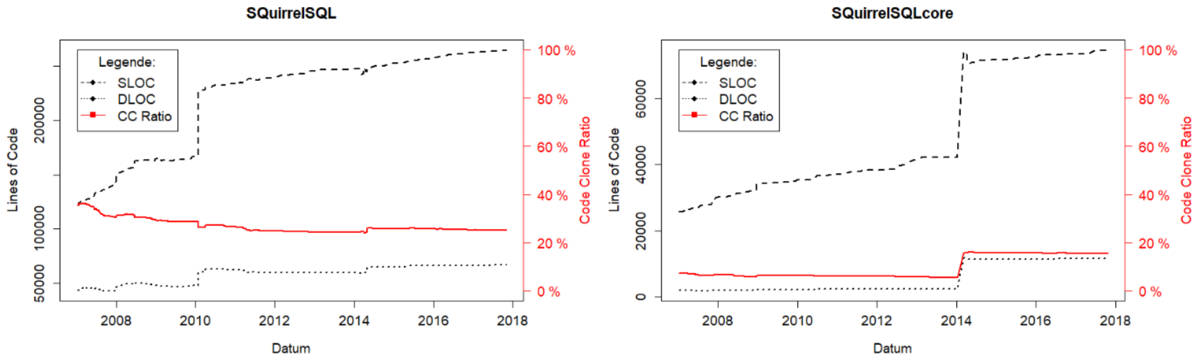
d) SpringFramework Kernmodul

Abbildung 8: Graphen signifikanter und duplizierter Programmierzeilen erweitert um die Code-Clone-Ratio

Projekt	***Pearson SLOC:DLOC***	***Pearson SLOC:CCR***	***Min-Max CCR***
Checkstyle	0,932	0,89	3,7-34,4%
EclipseJDT	0,986	0,603	19-26,5%
EclipseJDT core	0,985	-0,464	19,1-26,3%
Findbugs	0,92	0,9	2,1-23,9%
SpringFramework core	0,985	-0,245	4-7,4%
SquirrelSQL	0.903	-0,743	24,1-36,4%
SquirrelSQL core	0,959	0,801	5,7-16,2%

Tabelle 4: Korrelationskoeffizienten und CCR-Rahmen verschiedener Softwarequellen

Um die direkte Abhängigkeit zwischen SLOC und DLOC zu relativieren, wird in Abbildung 8 ein Verhältnis der beiden Größen eingeführt. Das Code-Clone-Verhältnis (*code clone ratio*, kurz: CCR), wie in Abbildung 8 mit der roten Linie gekennzeichnet, ermittelt sich aus dem Verhältnis von DLOC zu SLOC ($CCR = \frac{DLOC}{SLOC}$). Zwischen der neuen Größe und den SLOC gibt es relativiert über mehrere Projekte keine lineare Abhängigkeit mehr (vgl. Tabelle 4 SLOC:CCR). Aber es scheint, dass auch darüber hinaus eine weitere Abhängigkeit der beiden Größen existiert. Kommen abrupt große Anteile an Code hinzu, wie zum Beispiel in Findbugs im Jahr 2010 oder dem SquirrelSQL-Kernmodul im Jahr 2014 (Abbildung 8 b und c), dann wächst auch der relative Anteil an Code-Clones stark an. Dieser relative Anteil nimmt dann im Verlauf der Zeit wieder ab. Das ist ein Phänomen, was schon bei *C*- und *C*++-Projekten beobachtet werden konnte (vgl. [52, S. 49]). Erklärt werden kann es beispielsweise dadurch, dass ein großes Wachstum an Code unter Umständen mit der Einführung von ähnlichen Modulen einhergeht, die im Laufe der Zeit gegenüber ihrem kopierten Pendant abgeändert werden. Tatsächlich tritt das Phänomen relativ häufig auf, wenn man die Entwicklung ganzer Projekte statt einzelner Module betrachtet, wie in Abbildung 9 zu sehen. Dort sind die Schwankungen aufgrund der Projektgröße weniger stark, dafür jedoch häufiger. Die genaue Ursache der Anstiege kann anhand der reinen Betrachtung von Code-Clones nicht bestimmt werden und wird deswegen in einer Interessenpunkt-Analyse in Kapitel 4.4 näher untersucht.



a) SquirrelSQL Projekt

b) SquirrelSQL Kernmodul

Abbildung 9: Gegenüberstellung des kompletten SquirrelSQL-Projekts gegenüber dem Kernmodul

In der Betrachtung mehrerer Analysen stellte sich heraus, dass sich für die meisten Projekte der Code-Clone-Anteil in einem konstanten Rahmen hält. Dies ist zum Beispiel in Tabelle 4 unter Min-Max zu erkennen, wo jeweils Minimum und Maximum der CCR für die unterschiedlichen Softwarequellen festgehalten sind, wobei dort größere Ausreißer enthalten sind, deren Ursache in Kapitel 4.4 näher analysiert wird. Innerhalb eines festen Zeitrahmens, wie beispielsweise für das SquirrelSQL-Kernmodul die Unterteilung in vor und nach dem Jahr 2014, kann für viele Projekte ein relativ engmaschiger Rahmen für die CCR ausgemacht werden.

Des Weiteren kann in den Graphen aus Abbildung 8 gesehen werden, dass der Code-Clone-Anteil für viele Projekte eine sinkende Tendenz aufweist. Dies könnte als die Fähigkeit, Redundanz zu reduzieren, gedeutet werden. Eine lineare Senkung kann durch den Korrelationskoeffizienten (vgl. Tabelle 4 SLOC:CCR) nicht nachgewiesen werden, was vermutlich an den abrupten Steigungen liegt (vgl. Kapitel 4.4). Bereinigt um diese Ausreißer oder durch eine andere statistische Größe ist ein Nachweis einer sinkenden Tendenz dennoch denkbar. Die Tendenz könnte entweder die Folge einer Senkung der DLOC (Refactoring) oder die Folge von einem stärkeren Wachstum der SLOC als der DLOC (Vorbeugung) sein. Beides könnte unter anderem auf folgende Ursachen zurückgeführt werden:

- Gesteigerte Fähigkeiten der beteiligten Entwickler
- Gesteigerte Mächtigkeit der Sprache
- Einsatz von Clone-Detection-Tools
- Codequalität fördernde Prozesse wie Code Reviews
- Vermehrter Einsatz entwicklungsunterstützender Tools

Um den Einfluss der Programmiersprachenentwicklung darzustellen, lässt sich die Steigung der CCR ohne Ausreißer *m* für einen Zeitabschnitt *t* und eine Softwarequelle *s* durch die folgende Formel abstrahiert darstellen, wobei q_{pst} der Einfluss der Programmiersprache *p* unter der Annahme, dass sich eine Programmiersprache nicht zu redundanterer Syntax entwickelt[8] ist:

$$m_{st} = \frac{\mu_-}{q_{pst} \cdot \mu_+}$$

Die Variable μ_+ entspricht dem gesammelten Einfluss anderer redundanzmindernder Ursachen. Als Gegengröße ist in der Formel μ_- enthalten, also redundanzsteigernde Einflüsse, wie unter anderem die positiven Einflüsse von Code-Clones (vgl. [58]), der allgemeine Entwicklungsprozess, der Code-Clones mit sich bringt, oder unerfahrenere Ent-

---

[8] Diese Annahme wurde getroffen, weil wie in Kapitel 1 beschrieben, diese Arbeit nur einen Ansatz zum Beweis der Hypothese sucht, dass ein Kerntreiber der Entwicklung von Programmierkonzepten die Redundanz ist. Unter neutralerem Standpunkt muss auch beachtet werden, dass sich Features unter Umständen auch durchsetzen, wenn sie mehr Code-Dopplung verursachen, wie zum Beispiel zur Verbesserung der Lesbarkeit.

wickler, die Dopplungen nicht absehen können. Schafft man es also nach anderen Einflüssen μ zu kontrollieren und zu bereinigen, könnte sich daraus ein Einflusswert der Mächtigkeit der Programmiersprache berechnen lassen. Dieser Einflusswert q_{pst} wiederum könnte für unterschiedliche Versionen der Programmiersprache, also Zeitabschnitte *t* im Graphen, berechnet werden. Dazu müssen die analysierten Daten in Einflussbereiche der verschiedenen Programmiersprachenversionen, wie zum Beispiel in Kapitel 4.4 für EclipseJDT gezeigt, aufgeteilt werden und jeweils ein q_{pst} der entsprechenden Version *v* berechnet werden ($q_{pst} = q_{psv}$, wenn t dem Zeitabschnitt des redundanzvermindernden Einflussbereichs der Version entspricht). Wird nun weiterhin für die gleiche Programmiersprachenversion ein Wert aus hinreichend vielen verschiedenen Softwarequellen berechnet, so ließe sich ein direkter redundanzvermindernder Einfluss dieser Programmiersprachenversion q_{pv} berechnen. Um nun auf den Einfluss eines Programmierkonzepts schließen zu können, müssten verschiedene Programmiersprachen *p* und deren Versionen *v*, die dieses Programmierkonzept umsetzen, zur Schätzung verwendet werden. Solch eine hypothetische Rechnung ist nur möglich, weil die Messung von Code-Clones in unterschiedlichen Programmiersprachen und Softwareversionen vorgenommen werden kann und alle Programmierkonzepte gleichermaßen betrachtet werden können. Im Rahmen dieser Arbeit wurde die Rechnung nicht durchgeführt, weil allein die Extraktion der anderen Einflussfaktoren μ das Zeitbudget dieser Masterarbeit überschritten hätte.

4.3 Vergleich unterschiedlicher Konfigurationen

In diesem Kapitel sollen verschiedene Konfigurationen des Versuchaufbaus der empirischen Untersuchung gegenübergestellt werden. Ziel ist dabei ein Abwägen von Alternativen aber auch die Identifizierung von Möglichkeiten bestimmter Konfigurationen für die Beantwortung konkreter Forschungsfragen.

4.3.1 Vergleich unterschiedlicher Clone-Detection-Tools

Für den Vergleich der Clone-Detection-Tools Simian und CCFinderX wurde das Projekt SpringFramework [72] auf Code-Clones untersucht. Es wurde ein kürzerer Abschnitt, nämlich die jüngsten 7.500 Commits (Stand: 13.12.2017) in 50er Schritten, betrachtet. Dabei wurde nur das Kernmodul analysiert. Für Simian wurde ein Schwellwert von sechs Zeilen für eine Dopplung und bei CCFinderX ein Schwellwert von 50 Tokens verwendet. Dies sind zwar beides gängige Werte beim Vergleich unterschiedlicher Clone-Detection-

Tools (vgl. [91, S. 1162], [96, S. 133]), aber die unterschiedlichen Arten von Schwellwerten zeigen schon, dass die beiden Tools nicht zwangsläufig direkt miteinander vergleichbar sind.

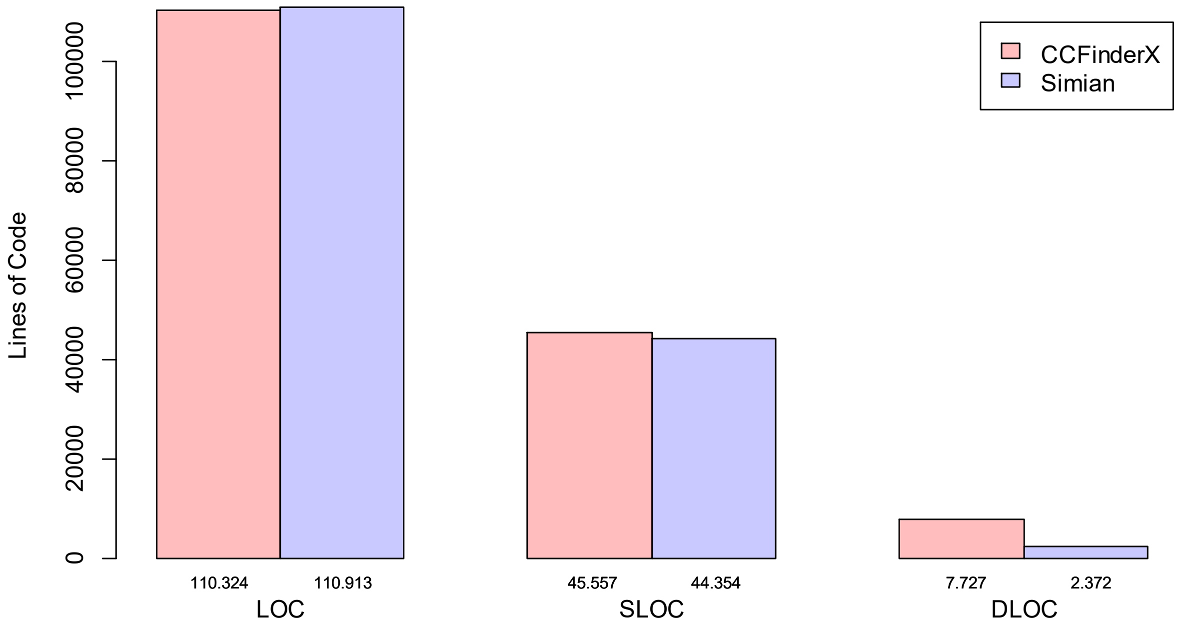


Abbildung 10: Vergleich der Kennzahlen von Simian und CCFinderX in einer Analyse

Für eine Deutung der Kennzahlen sind zunächst in Abbildung 10 die betrachteten Kennzahlen aus einer Version in einem Säulendiagramm für die Clone-Detection-Tools jeweils gegenübergestellt. Die betrachteten Code-Zeilen (LOC) sollten dabei stets gleich sein, da diese eindeutig in den untersuchten Dateien definiert sind. Zwischen den beiden Tools scheinen dennoch Unterschiede in der Auffassung dieser zu existieren, da die Zahlen um 589 abweichen, was zwar im Verhältnis zu 110.000 LOC nicht hoch ist, aber bei solch einer Kennzahl nicht auftreten sollte. Die Ursache dafür ist unbekannt, da beide Clone-Detection-Tools die gleichen Dateien betrachten, es könnte aber daran liegen, dass einer der beiden eventuell die LOC selbst berechnet und dabei Unterschiede in der Auffassung auftreten zum Beispiel bei Zählung der Start- und Endzeile. Dennoch wird dies als eine mögliche Fehlerquelle für die Vergleichbarkeit der beiden Clone-Detection-Tools vermerkt.

Die SLOC unterscheiden sich ein wenig stärker. Hier liegt der Unterschied bei 1203 Code-Zeilen, was etwa 2,6% entspricht. Der Unterschied ist verständlicher, weil es zwar eindeutige Definitionen gibt, diese aber nicht unbedingt gleich verwendet werden, beispielsweise wegen Zählung von logischen oder physischen LOC oder der Formatierung des Quellcodes vor der Zählung (vgl. [97]). Die SLOC geben in der Analyse den Betrachtungsraum des Clone-Detection-Tools vor und sollten in der Lösungspräsentation erwähnt werden, weil sich Code-Duplikate daraus ermitteln und somit auch nur unter Beachtung der SLOC als Größenordnung mit den Ergebnissen anderer Clone-Detection-Tools vergleichen lassen. Bei dem eigentlichen Resultat der Clone-Detection, den *duplicated lines of code*, ist der Unterschied am größten. In der Version aus Abbildung 10 berechnet Simian nur etwa 30,7% der Anzahl an duplizierten Code-Zeilen von CCFinderX.

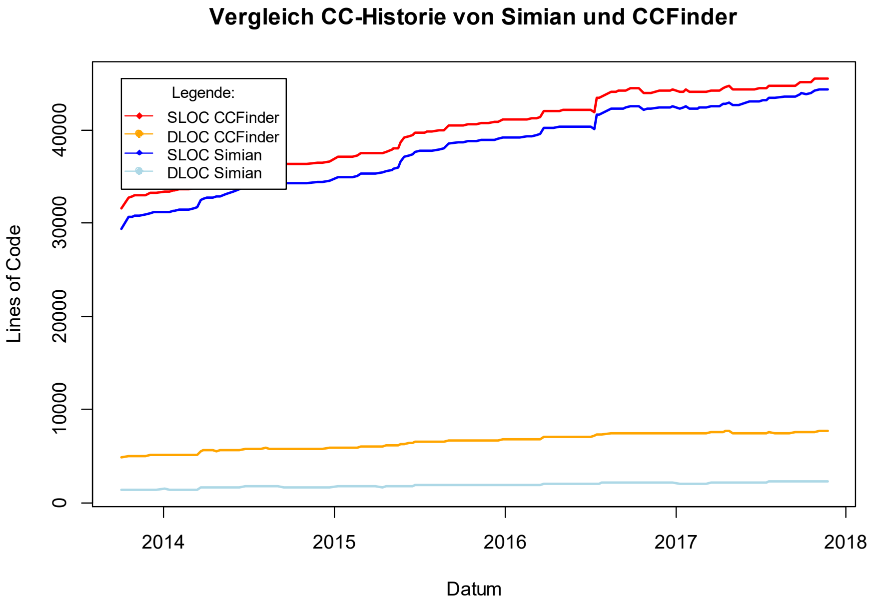


Abbildung 11: Vergleich der Code-Clone-Historie von Clone-Detection-Tools Simian und CCFinderX

In Abbildung 11 sind die Code-Clone-Historien, also die gemessenen Kennzahlen aus allen untersuchten Commits, der beiden Tools zu sehen. Erkennbar ist vor allem, dass trotz der unterschiedlichen Definitionen und Größeneinheiten beide Tools vor allem bei den SLOCs ähnliche Verläufe zeichnen. Um die Ähnlichkeiten der Verläufe in Zahlen zu fassen, wurde zunächst für jedes Datum die Differenz der beiden berechneten SLOC ermittelt. Anschließend wurden von den Differenzen Mittelwert und Standardabweichung berechnet. Der Unterschied zwischen den beiden SLOC-Definitionen liegt in dem hier gezeigten Beispiel bei durchschnittlich 1874 Code-Zeilen und die Standardabweichung liegt bei etwa 277. Der Wert der Standardabweichung ist höher als erwartet, kann aber auch am Graphen nachvollzogen werden, da dort der Abstand zwischen den SLOC-Linien bei weiter zurückliegendem Datum wächst. Gerechnet auf durchschnittlich 40.000 SLOC ist das dennoch ein sehr geringer Wert und zeigt den ähnlichen Verlauf der beiden Graphen. Für eine genauere Analyse der Vergleichbarkeit der beiden Clone-Detection-Tools sollte vermutlich ein anderes Ähnlichkeitsmaß verwendet werden, was den steigenden Abstand berücksichtigen könnte.

Der Unterschied zwischen den beiden Clone-Detection-Tools fiel bei der Untersuchung aus Abbildung 11 für die DLOC größer aus. So ist einerseits der relative Anteil an DLOC gegenüber SLOC für CCFinderX deutlich größer als bei Simian. Darüber hinaus sind auch stellenweise häufiger Wachstum und Abfall vom Code-Clone-Anteil zu erkennen. Mitte April 2017 steigt zum Beispiel beim CCFinderX der Code-Clone-Anteil auf 17,3 % und liegt in den benachbarten Monaten bei 17% und 16,9%, in Simian dagegen ist der Unterschied von 5,1% im April zu den Nachbarmonaten mit 5,07% nur ein Zehntel so groß (vgl. CD-Anhang Code-Clone-Historien). Es scheint also, dass CCFinderX empfindlicher gegenüber Veränderungen ist. Wird für die DLOC-Verläufe die gleiche Ähnlichkeitsanalyse wie für SLOC durchgeführt, ergibt sich für die Differenzen ein arithmetisches Mittel in Höhe von 4.659 bei einer Standardabweichung von 604. Diese Werte sind im Verhältnis zu den untersuchten Größen (DLOCs bei Simian im Durchschnitt 1.922) wesentlich höher und unterstreichen einen starken Unterschied in der Clone-Detection.

Neben den qualitativen Unterschieden, die bisher genannt wurden, gibt es aber auch Performanz-Unterschiede zwischen den beiden Clone-Detection-Tools. Für die Erstellung der Historie aus Abbildung 11 brauchte Simian im Durchschnitt acht Sekunden pro Commit und CCFinderX war zwar inkonsistenter[9], brauchte im Durchschnitt aber nur 6,4 Sekunden pro Commit (vgl. CD-Anhang im Ordner *Runtimes*). Der erste Durchlauf benötigt bei CCFinderX in der Regel sehr lange (in diesem Fall 55 Sekunden). Ohne diesen Durchlauf liegt die Durchschnittszeit bei nur etwa 6,1 Sekunden. Für größere Analysen mit mehr Commits lohnt sich ein Einsatz von CCFinderX also prinzipiell eher als für kürzere Analysen.

Zusammenfassend schneidet CCFinderX für diese empirische Untersuchung sowohl qualitativ als auch die Laufzeit betreffend besser ab. Allerdings erschweren die in Kapitel

---

[9] Standardabweichung ohne den ersten Durchlauf bei CCFinderX bei 3,1 Sekunden. Bei Simian liegt die Standardabweichung bei etwa 0,25 Sekunden.

3.3.2.2 erwähnten Probleme den verlässlichen Einsatz, weswegen in dieser Arbeit Simian deutlich häufiger für die Berechnung eingesetzt wurde. Für weitere empirische Untersuchungen dieser Art lohnt sich aber eine Evaluation anderer Tools, die in Kapitel 3.3 vorgestellt wurden. Dafür eignen sich auch andere Vergleichsverfahren, wie der Vergleich anhand eines Clone-Detection-Korpus wie der *BigCloneBench* [96].

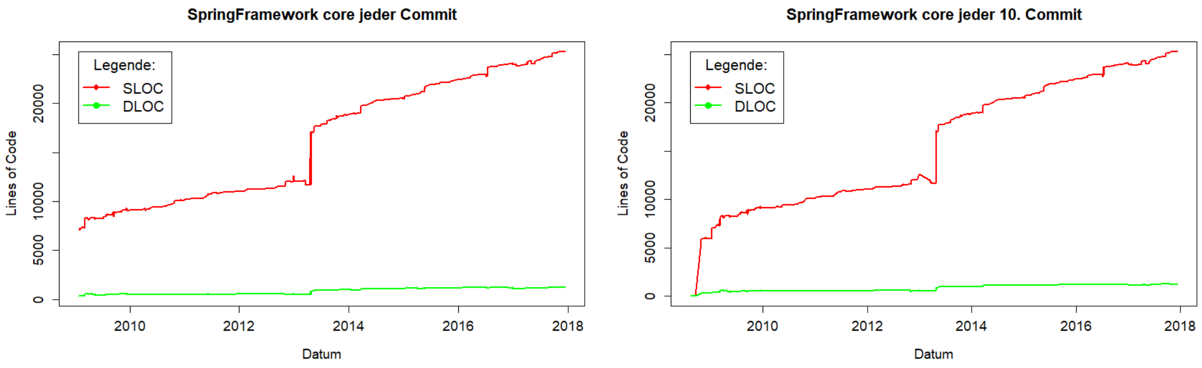
4.3.2 Vergleich unterschiedlicher Commit-Filter

In diesem Kapitel soll der Einfluss der Auswahl der untersuchten Commits einer Software-Historie analysiert werden. Dafür wurden zunächst Commits in festen Abständen betrachtet. Dies soll einen Einblick darauf geben, wieviel Information verloren geht, wenn Commits übersprungen werden.

In den Graphen der Abbildung 12 wurden für das SpringFramework Commit-Filter mit unterschiedlichen Schrittweiten betrachtet. Der Unterschied ist in den Graphen a - c, abgesehen von leichtem Rauschen, kaum erkennbar[10]. Lediglich in Abbildung 12 d gehen Informationen verloren, die eventuell zur Identifizierung eines Interessenpunkts wichtig sind, da zum Beispiel starke Schwankungen nur noch schwer einem Zeitraum zuzuordnen sind. Ursache für den mangelnden Einfluss vieler Commits ist vermutlich, dass oft nur wenige Zeilen verändert werden, und damit kein großer Einfluss auf die Anzahl der duplizierten Code-Zeilen entsteht. Außerdem wird ein weitgefasster Kontext betrachtet: Für die Analyse einer vollständigen Historie werden in der Regel etwa 14 Jahre betrachtet und zehn Commits entsprechen in aktiveren Repositories mitunter nur einem Tag.

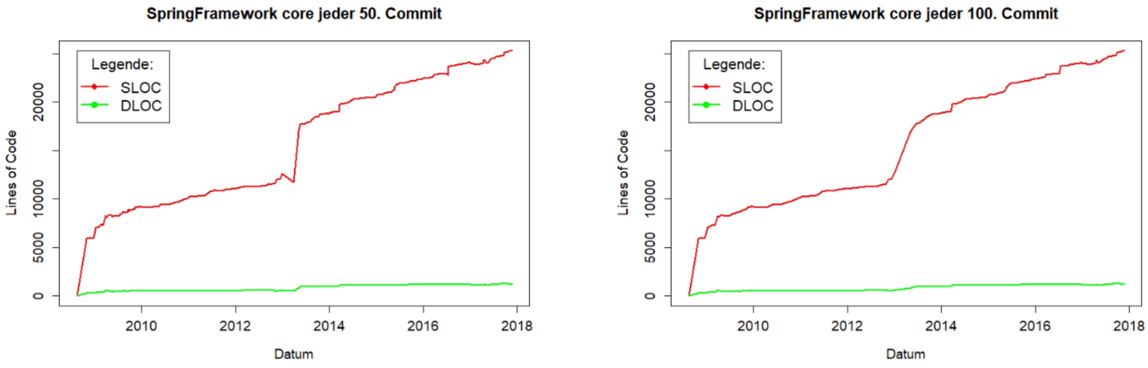
---

[10] Der Durchlauf der Rechnung von Abbildung 12 a, also die Verwendung von jedem Commit, wurde nach fünf Stunden verfrüht abgebrochen und ist deswegen am linken Rand verkürzt.



a) Jeder Commit im Filter (verkürzt)

b) Jeder 10. Commit im Filter



c) Jeder 50. Commit im Filter

d) Jeder 100. Commit im Filter

Abbildung 12: Betrachtung von Code-Clone-Analysen mit unterschiedlichen Commit-Filtern

Obwohl in vielen Repositories trotz starker Reduktion der betrachteten Commits nur wenige Informationen, die für die Analysen aus Kapitel 4.4 benötigt wurden, verloren gehen, hat die Reduktion dennoch starken Einfluss auf die Laufzeit, da eine Iteration im Durchschnitt etwa acht Sekunden benötigt.

4.3.3 Vergleich unterschiedlicher Schwellwerte für die Erkennung

Eine weitere relevante, variable Größe in der Clone-Detection ist der Schwellwert der Clone-Detection-Tools. Dieser gibt an, ab wann ein Code-Clone als solcher zählt. In Simian und ähnlichen textbasierten Clone-Detection-Tools werden dafür Mindestzeilen verwendet. Ein üblicher Wert ist dabei sechs [12], [91, S. 1162], [96, S. 133]. Bei token-basierten Clone-Detection-Tools wie CCFinderX werden Mindesttokens festgelegt, in der Regel liegen die bei 50 [90], [91, S. 1162], [96, S. 133].

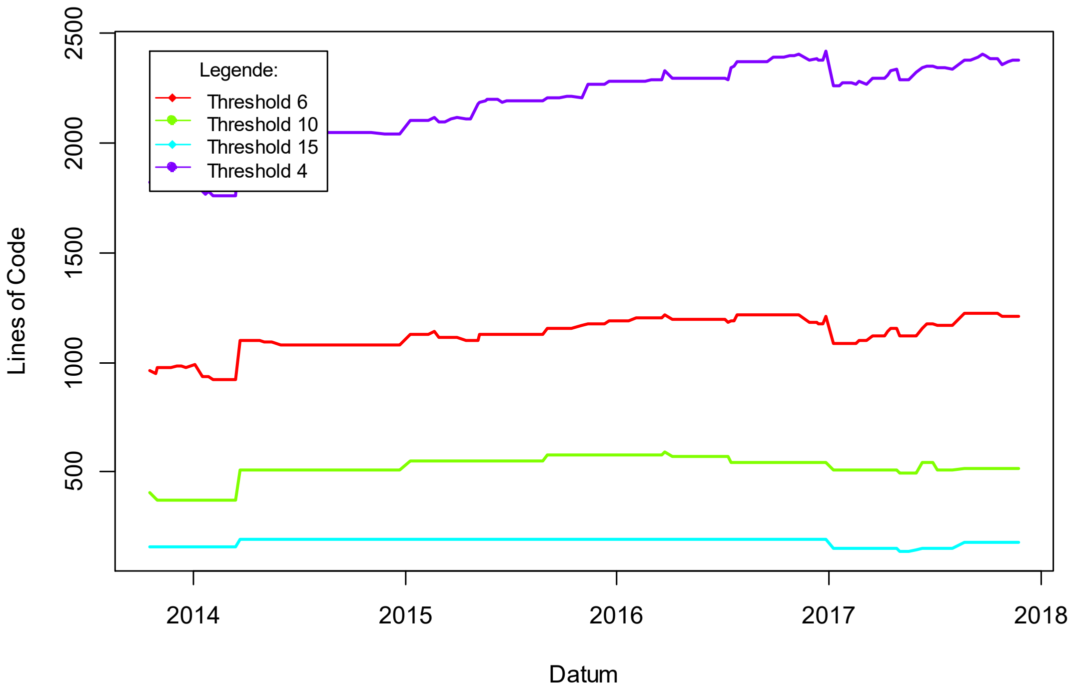


Abbildung 13: Code-Clone-Historien mit unterschiedlichen Schwellwerten

In diesem Kapitel geht es, im Gegensatz zu den vorangegangenen, nicht um eine Abwägung von Alternativen, sondern vielmehr um eine Einordnung der gefundenen und relevanten Code-Clones. In diesem Fall sollen die Größen der untersuchten Code-Clones unterschieden werden, was sich über die Schwellwerte bestimmen lässt. Mit anderen vom Clone-Detection-Tool gegebenen Werten würden sich auch andere Metadaten prüfen oder Clones mit bestimmten Eigenschaften herausfiltern lassen. Hier soll jedoch zunächst gezeigt werden, wie eine solche Klassifizierung aussehen könnte, und was für Ergebnisse sich daraus herauslesen ließen.

Mit Simian kann der Schwellwert über den *threshold*-Parameter geändert werden. In Abbildung 13 ist sichtbar, wie sich eine Code-Clone-Historie mit unterschiedlichen Schwellwerten verhält. Mit niedrigerem Schwellwert werden mehr Code-Clones identifiziert. Des Weiteren scheinen kleinere Code-Clones häufiger zu variieren, was sich durch häufigere Ausreißer in der Kurve äußert. Dies kann aber auch mit der höheren Anzahl an Code-Clones zusammenhängen.

Die Schwellwerte der Clone-Detection-Tools können als Filter für bestimmte Arten an Clones verwendet werden. Für eine weitere Analyse könnten zum Beispiel ausschließlich große Code-Clones betrachtet werden, also eine Datenerhebung mit hohem Schwellwert. Dadurch könnten gezielt Programmierkonzepte wie beispielsweise die Java Generics untersucht werden, wo ein Einsatz größere Mengen von Methoden oder ganze Klassen obsolet machen kann, ergo sehr große Code-Clones entfernt werden können.

4.4 Untersuchung verschiedener Interessenpunkte

In diesem Kapitel ist es das Ziel, Interessenpunkte aus Messungen näher zu untersuchen. Ein Interessenpunkt soll in diesem Kontext eine Auffälligkeit der Datenreihe darstellen. Dafür wird zunächst geklärt, was in diesem Kontext von Interesse ist und wie es objektiv ausgemacht werden könnte. Anschließend werden verschieden Punkte analysiert, wo aufgrund einer Änderung der Umstände eine starke Änderung der Kennzahlen vermutet wird und abschließend werden sonstige Anomalien der Datenreihen näher auf ihre Ursache geprüft.

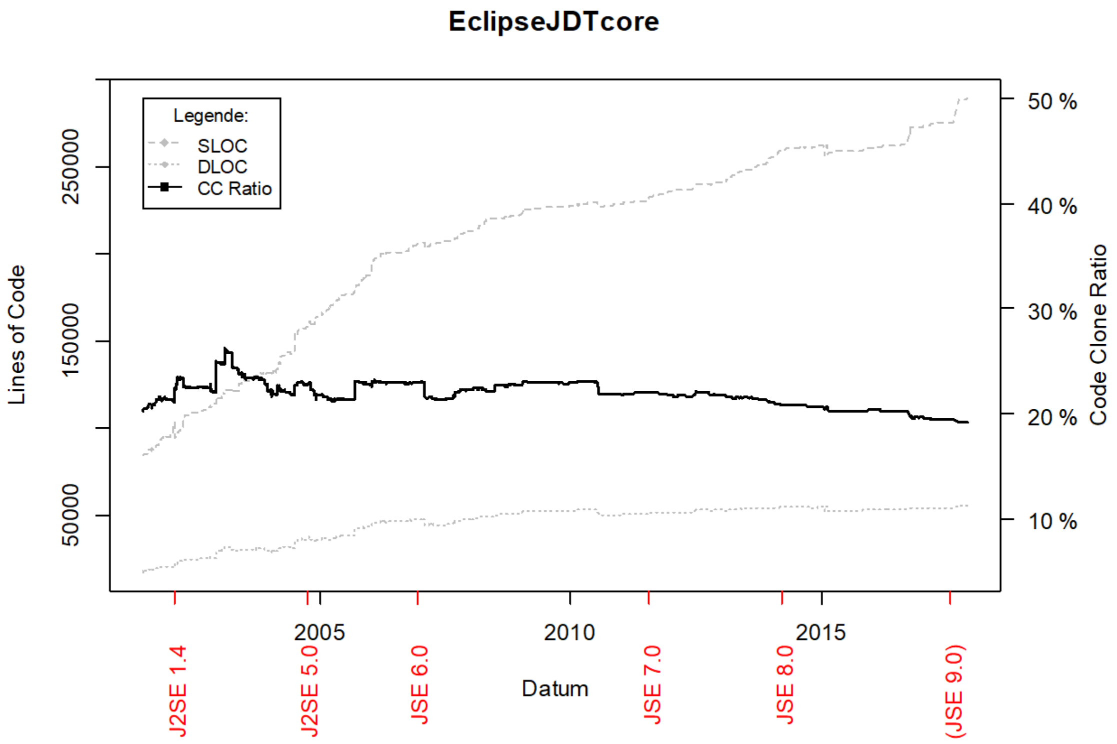


Abbildung 14: Code-Clone-Historie vom EclipseJDT-Kernmodul

Für die Analyse äußerer Umstandsänderungen sollte zunächst ein Graph ohne stärkere Anomalien ausgewählt werden. Dafür wird die in Abbildung 14 gezeigte Code-Clone-Historie des EclipseJDT-Kernmoduls betrachtet. Durch die enorme Größe von nahezu 300.000 Programmzeilen in einem Modul, besitzt diese Sofwarequelle Robustheit und hat weniger extreme Schwankungen zwischen Versionen als kleinere Projekte. Für die

Untersuchung wurden dafür zunächst die Zeitpunkte der Neuerungen der verwendeten Programmiersprache Java im Graphen rot gekennzeichnet (vgl. Kapitel 2.1.2).

In Tabelle 1 auf Seite 6 sind die hier markierten Versionen und einhergehende Neuerungen mit Redundanzverminderungspotenzial zu sehen (vgl. Kapitel 2.1.3). An den markierten Stellen wurde also zunächst ein Rückgang der Code-Clones vermutet. Im Graphen nehmen nach der Veröffentlichung einer neuen Version bei EclipseJDT zwar die Code-Clones ab, diese Tendenz zur Verringerung findet aber über die komplette Historie statt und kann dadurch, dass es keine starken Einbrüche an den eingetragenen Punkten gibt, nicht eindeutig einer Versionsneuerung zugeordnet werden.

Die allgemeine Veröffentlichung einer neuen Version der Programmiersprache ist jedoch noch keine Aussage darüber, wann eine solche Neuerung wirklich eingeführt wurde. Dafür wurde speziell für das Projekt *EclipseJDT* aus Abbildung 14 das Repository näher untersucht, um zu schauen, wann zum Beispiel auf Java 7 umgestellt wurde. Die Umstellung in EclipseJDT begann am 20. Januar 2011 und wurde am 28. Juli komplettiert. Als Entwicklungsumgebung ist EclipseJDT mit der Umstellungszeit eine Ausnahme, da Programmiersprachen-Features nicht nur für die eigene Verwendung benutzt werden, sondern auch im Produkt unterstützt werden müssen. Dennoch sind wahrscheinlich größere Refactorings an bestehendem Code frühestens nach dem 28. Juli vorgenommen worden, was sich ungefähr mit dem Einführungsdatum von Java 7 (07.07.2011) deckt.

Schon oberflächlich ist kaum ein Zusammenhang der Rückgänge relativer Code-Clone-Anteile mit der Einführung oder der Adaption einer neuen Java-Version zu erkennen. Neben der mangelnden Aussagekraft ist eine solche Methode der manuellen Untersuchung der Repositories auf Umstellung auch sehr langwierig und für eine größere Anzahl an Projekten und Versionen eher ungeeignet.

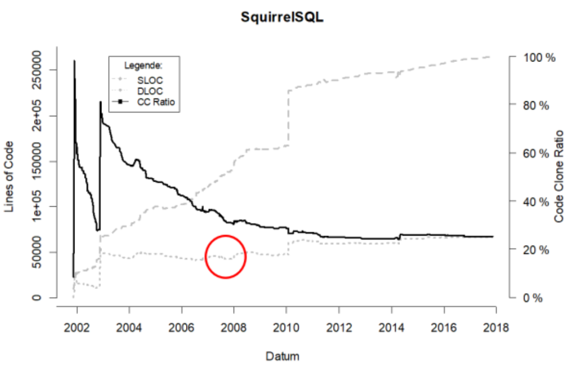


Abbildung 16: Code-Clone-Historie des SquirrelSQL-Projekts

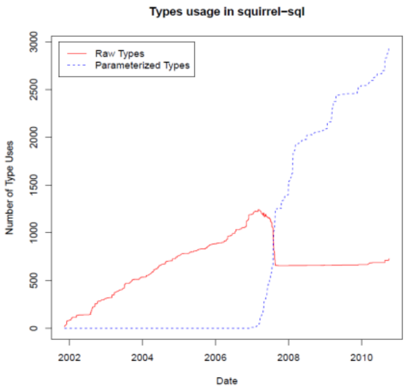


Abbildung 15: Verwendung von Generic-Definitionen und Aufrufen im SquirrelSQL-Projekt [5]

Aus diesem Grund wurden Daten anderer Arbeiten hinzugezogen, um eine Adaption eventuell leichter erkennen zu können. In der Arbeit von Parnin et al. [5] zum Beispiel wurde speziell die Einführung von Java Generics untersucht. Deren Verlauf lässt sich auch anhand eines Graphen nachverfolgen, wie in Abbildung 15 zu sehen ist. Dabei sind *parametrized types* (blau) generische Typ-Definitionen, ein Anstieg zeigt also einen Zuwachs an Generics Verwendungen. Die *raw types* (rot) hingegen stellen generisierbare Typen dar, die durch eine Generic-Definition ersetzt werden könnten. Hier ist also ein Rückgang Indikator für Redundanzminderung. Wenn man nun die Zunahmen der Java Generics mit der Abnahme der Code-Clone-Anteile aus Abbildung 16 für das gleiche Projekt zeitlich vergleicht, so lässt sich kein eindeutiger Zusammenhang aufweisen. Tatsächlich ist aber Mitte 2017, wo von Parnin et al. eine stärkere Überarbeitung festgestellt werden konnte [5, S. 9 unten], ein leichter Rückgang der DLOC um etwa 3.000 Code-Zeilen (in Abbildung 16 rot markiert) erkennbar. Diesem Rückgang konnte allerdings statistisch keine Signifikanz zugeordnet werden, weil allgemein starke Schwankungen der DLOC-Kurve vorliegen. Ein konkreter Zusammenhang müsste also mit mehr Daten über weitere Projekte geprüft werden. Dazu wäre eine Rekonstruktion der Untersuchung von Parnin et al. notwendig.

Folgend werden nun Anomalien im Graphen auf ihre Ursachen untersucht, um dadurch Hinweise auf Ursachen in der Programmiersprachenentwicklung zu finden oder mögliche Fehlerquellen zu identifizieren und auszuschließen.

In Abbildung 8 c auf Seite 38, dem Graphen des SquirrelSQL-Kernmoduls, kann im Januar 2014 ein starker Anstieg der SLOC und DLOC festgestellt werden. Unter näherer

Analyse stellt sich aber heraus, dass zu diesem Zeitpunkt die Ordnerstruktur verändert wurde, sodass andere Module zusätzlich betrachtet wurden. Ab diesem Punkt steigt aber auch der Code-Clone-Anteil enorm an. Bei näherer Untersuchung stellte sich heraus, dass die Ursache weder die Anbindung von Testfällen oder Dopplung ganzer Module durch funktionale Redundanz (vgl. Kapitel 2.2.2.1) ist, was bei den Untersuchungen häufig Ursache mit wenig Aussagekraft eines solchen rapiden Anstiegs war. Stattdessen gibt es ab diesem Zeitpunkt vermehrt Dopplungen durch Dialekte [98] und eigene Datentypen mit ähnlichem Aufbau. Es scheint, als sei dies eine bewusste Entscheidung zu Ähnlichkeit, wie im Code-Clone-Muster „*Forking*" [58, S. 651].

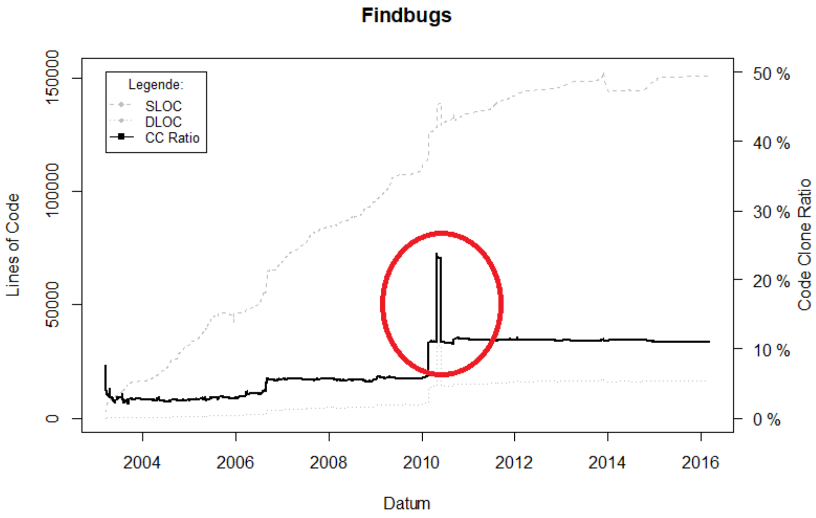


Abbildung 17: Code-Clone-Historie von Findbugs mit markiertem Interessenpunkt

Findbugs enthält ebenfalls eine Auffälligkeit. In Abbildung 17 rot markiert gibt es eine Stelle um den 28. April 2010, an dem der Code-Clone-Anteil enorm ansteigt und anschließend am 2.Juni 2010 wieder abfällt. Ursache dafür ist die aktive Dopplung eines ganzen Plugins vermutlich zu Testzwecken (Code-Clone-Muster „*Experimental Variation*" [58, S. 653]). Erst über einen Monat später wurde das Modul von einem anderen Entwickler als obsolet erkannt und entfernt.

Im Gegensatz dazu ist im SpringFramework (Abbildung 8, S. 38), wo durch eine Umstrukturierung der Module die SLOC enorm ansteigen (am 23. April 2013), der Code-Clone-Anteil ab diesem Punkt nicht sonderlich höher, weswegen vermutet werden kann, dass der zusätzliche Code von einer ähnlichen Güte ist.

Die Untersuchung der verschiedenen Interessenpunkte hat gezeigt, dass starke Änderungen der Code-Clones vielerlei Ursachen haben können. Eine Zusprechung einer drastischen Änderung der Code-Clone-Anteile zu einer Änderung der Programmiersprache ist äußerst schwierig, von vielen Faktoren abhängig und damit auch schwer automatisiert nachweisbar. Kennzahlen von der kompletten Historie oder Abschnitten abzuleiten, wie in Kapitel 4.2 beschrieben, bietet sich demnach vermutlich eher an. Ansonsten wäre auch eine Kombination beider Ansätze denkbar, da die Erweiterung um zusätzliche Daten durchaus neue Erkenntnisse brachte.

5 Nachbetrachtung

5.1 Fehlerbetrachtung

In diesem Kapitel sollen verschiedene Fehlerquellen analysiert werden. Dazu werden zunächst Fehler betrachtet, die im Laufe der Arbeit eliminiert werden konnten, aber dennoch in ähnlicher Form auftreten könnten und anschließend werden Fehlerquellen aufgezeigt, die eine Gefahr für die Validität der Arbeit darstellen.

Ein großes Problem sind die Testfälle der Software-Projekte. In einigen Arbeiten werden Tests ausgeschlossen ([52, S. 48]), weil diese sich häufig vom normalen Entwicklungsprozess stark unterscheiden und viele Clones produzieren, die aber wenig aussagen. Deswegen wurden in den Ausführungen mit der Hilfe von verschiedenen durch die Clone-Detection-Tools gegebenen Parameter Tests ausgeschlossen. Leider stellte sich heraus, dass die Tests nicht vollständig erfasst wurden. Ursache dafür sind unterschiedliche Formen der Annotationen von Tests[11], die durch die Filter des Clone-Detection-Tools nicht erkannt wurden. Als Schlussfolgerung daraus wurden bei neueren Ausführungen ausschließlich zentrale Elemente, in der Regel der „src“-Ordner des Kernmoduls, analysiert, um dadurch etwaige Testanalysen zu unterbinden.

Durch die Betrachtung einzelner Module wurde auch das Problem der Dopplung von Modulen behoben. In Kapitel 4.4 wurde bereits angedeutet, dass es in Projekten relativ häufig vorkam, dass größere Anstiege von Code-Clone-Anteilen häufig mit der Dopplung ganzer Module zum Beispiel zu Testzwecken zusammenhingen. Diese funktionale Dopplung verzerrt das Bild über duplizierten Code mitunter sehr stark, sodass Einflüsse von Programmiersprachen nur noch schwer hervortreten.

Als Alternative zur Ausschließung von Tests wurden für einzelne Module alle Dateien ungefiltert betrachtet. Dies erhöht zwar die Code-Clone-Anzahl, wurde aber gleichermaßen für miteinander verglichene Analysen gemacht, um dadurch ähnliche Bedingungen zu schaffen. Auch wenn die duplizierten Zeilen dadurch höher sind, sollte eine Entwicklung der duplizierten Zeilen ähnlich verlaufen, weil für produktiven Code immer

---

[11] Tests wurden weitestgehend durch ein „Test“ im Klassennamen gekennzeichnet aber auch teilweise nur durch Methoden- oder Ordnerbezeichnungen. In manchen Fällen wurden sogar ausschließlich Java Annotations für die Identifizierung eines Tests verwendet. Darüber hinaus waren Klassenbezeichnungen nicht immer eindeutig durch das Clone-Detection-Tool erkennbar. So wurde zum Beispiel bei einer Benennung von „AccessRestrictionsTests.java“ statt „AccessRestrictionsTest.java“ in EclipseJDT kein Ausschluss mehr durchgeführt.

etwa der gleiche Anteil an Test-Code hinzugefügt wird. Der Ansatz dieser ganzheitlichen Betrachtung ist im Sinne der Arbeit vielleicht sogar noch besser geeignet. Dadurch kann zum Beispiel auch die Entwicklung von Programmiersprachen im Kontext der Testentwicklung abgedeckt werden.

Die Auswahl der Softwarequellen, wie sie in Kapitel 2.3 beschrieben ist, stellt ein Problem dar, weil durch die eingeschränkte Betrachtung von Softwarequellen zwar ein leichterer Prozessaufbau möglich war, aber dadurch auch nur ein eingeschränktes Bild der Realität gezeichnet wird. Open-Source-Projekte mit Entwicklern aus unterschiedlichen Bereichen haben beispielsweise andere Entwicklungsvorgänge als Entwicklungen innerhalb eines Unternehmens. Dort könnten die Anpassungen an neue Programmiersprachen ganz anders priorisiert sein. Diese Problematik ließe sich in einer Erweiterung der empirischen Untersuchung um weitere Softwarequellen beheben.

Dass den Effekten nicht eindeutig Ursache zuzuordnen sind, ist dagegen ein Problem des Ansatzes selbst. Wenn ausschließlich Code-Clones historisch betrachtet werden, können zwar Rückgänge oder Anstiege der Anteile klar erkannt werden, die Ursachen dafür können aber unterschiedlicher Natur sein. So könnte zum Beispiel einfach ein neuer Entwickler zum Projekt zugestoßen sein und dadurch neue Konzepte in der Bekämpfung von Redundanz mit sich bringen. Die Ursachenforschung ist dabei zwangsläufig abhängig von weiteren Daten. Diese müssen entweder zusätzlich selbst erhoben werden oder aus Ergebnissen anderer Arbeiten angefügt werden. Diese Ansätze bergen jedoch selbst wieder eigene Fehlerquellen.

Außerdem ist die Entwicklung von Code-Clones nicht zwangsläufig eindeutig. So könnte beispielsweise eine gleich bleibende Anzahl an Code-Clones einerseits bedeuten, dass die duplizierten Stellen des Codes nicht berührt wurden oder anderseits, dass gleich viele Code-Clones abgegangen und hinzugekommen sind. Um also eine eindeutige Entwicklung von Code-Clones nachzuvollziehen, müssten die vorhandenen Code-Clones über mehrere Versionen hinweg verfolgt werden. Dies ist mit Ansätzen des Clone-Trackings durchaus möglich [51], wurde aber in dieser Arbeit nicht verwendet.

Die Ergebnisse, die in der durchgeführten empirischen Untersuchung erhoben wurden, sind wegen des Tool-Chain-Ansatzes stark abhängig von den verwendeten Werkzeugen. Unter anderem durch Vergleiche unterschiedlicher Einstellungen und Clone-Detection-Tools sollte der Einfluss relativiert werden. Dennoch sind erzeugte Kennzahlen abhängig von der fehlerfreien Ausführung darunterliegender Technologien. Unterschiede in den

Interpretationen offenbarten sich ansatzweise im Vergleich unterschiedlicher Clone-Detection-Tools in Kapitel 4.3.1. Dort wurden Diskrepanzen in der Zählung der Code-Zeilen sichtbar. Die Unterschiede waren zwar nicht gravierend, zeigen aber, dass entweder durchaus unterschiedliche Semantiken der Kennzahlen vorliegen oder zumindest unterschiedliche Rechnungen durchgeführt werden. Der modulare Aufbau des Programms zur Prozesssteuerung ermöglicht dabei zwar einen leichten Austausch der Werkzeuge, dafür müssen die Werkzeuge aber zunächst auf ihr Fehlerverhalten oder Semantik geprüft werden. Ein Ansatz zur Prüfung des Fehlverhaltens funktioniert zum Beispiel durch die Auswertung bekannter Code-Clones [87, S. 254].

Insgesamt gibt es eine Vielzahl an Fehlerquellen. Unter Berücksichtigung dieser können die Ergebnisse der empirischen Untersuchung jedoch durchaus gewertet werden und mit einer Erweiterung wären eindeutigere Ergebnisse sogar durchaus denkbar.

5.2 Erweiterungsmöglichkeiten

In diesem Kapitel sollen verschiedene Möglichkeiten für die Erweiterung der bestehenden empirischen Untersuchung und Arbeit aufgezeigt werden.

Einerseits könnten in aufbauenden Untersuchungen bestehende Fehlerquellen durch Erweiterung der Untersuchungsbereiche reduziert werden. So könnten beispielsweise neue Softwarequellen aus anderen Kontexten berücksichtigt werden. Eventuell könnte auch ein Korpus, ähnlich dem *Qualitas Corpus* [66], der in einer Untersuchung als Softwarequelle (Kapitel 2.3) nicht gewählt wurde, verwendet werden, um die Qualität entstehender Ergebnisse zu verbessern.

Darüber hinaus könnte die Betrachtung von anderen Programmiersprachen, anderen Zeiträumen oder weiteren Kennzahlen anderer Arbeiten neue Erkenntnisse liefern. Jedes Szenario hat dabei andere Umstände und das Einführen von Programmierkonzepten könnte stärkere Folgen haben. In gleicher Art und Weise könnte auch die Verwendung neuer Technologien Einfluss auf die Ergebnisse haben. So könnte zum Beispiel ein neueres Clone-Detection-Tool verwendet werden, um vielleicht in kürzerer Laufzeit mehr Code-Clones zu finden, deren Beeinflussung eventuell häufiger sichtbar ist.

Eine andere zusätzliche Technologie, die auch mehrfach in der Arbeit erwähnt wurde, ist das Clone-Tracking im Rahmen der Clone-Evolution. Die Nachverfolgung von Code-Clones bietet für die empirische Untersuchung vielerlei Chancen. So könnten eventuell

Ursachen der Eliminierung klarer ausgemacht werden. Damit können Feature-Einführungen als Ursache für Redundanzreduzierung klarer gesammelt werden. Damit könnte ein sowohl effektorientierter Ansatz, wie in dieser Arbeit, mit einem ursachenorientierten Ansatz wie zum Beispiel aus Parnin et al. [5] kombiniert werden.

Darüber hinaus könnten auch, wie bereits in Kapitel 4.3.3 erwähnt, Code-Clones gefiltert werden. Durch die selektive Analyse bestimmter Code-Clones könnten Erkenntnisse über die Art und Weise der Einflüsse bestimmter Maßnahmen gemessen werden. So ist beispielsweise denkbar, dass die Dopplung ganzer Module in der Regel eher größere Code-Clones erzeugt. Diese könnten zum Beispiel im Kontext einer Programmierkonzept-Analyse ignoriert werden.

Abgesehen von der Erweiterung der empirischen Untersuchung könnten auch die Ergebnisse dieser Arbeit weiterverwendet werden. Mit Hilfe des automatisierten Analyseprozesses könnte beispielsweise die Untersuchung der Ausgangsthese aus Kapitel 1.2 durchgeführt werden. Auch eine erweiterte Analyse der bisher erhobenen Daten ist denkbar. Dabei könnten durch Verwendung anderer statistischer Kenngrößen oder Vergleiche mit anderen Daten durchaus neue Erkenntnisse entstehen.

5.3 Schlussbemerkung

Der größte Mehrwert dieser Arbeit ist die Vorbereitung weiterer Analysen auf dem Feld der Untersuchung von Einflüssen von Redundanz. So wurden unter anderem Begriffe der Redundanz in Software und der Programmierkonzepte geklärt. Außerdem wurden erste Analysen zu vorhandenen Treibern von Programmierkonzepten durchgeführt.

Darauf aufsetzend wurde ein automatisierter Prozess zur Erhebung von Code-Clones aus verschiedenen Softwarequellen entwickelt. In dem Kontext wurden Ansätze zur Integration und Auswahl von Clone-Detection-Tools vorgestellt, wobei Simian sich als einfacher zu einzusetzen und konsistenter in der Ausführung herausstellte, CCFinderX aber sowohl durch höhere Qualität im Finden von Clones als auch durch die kürzere Laufzeit eine starke Alternative darstellt. Außerdem wurde ein Ansatz für das Versionsmanagement der Softwarequellen sowie das Zeiteinsparungspotenzial mit wenig Informationsverlust durch Commit-Filterung gezeigt. Die entstehenden Daten wurden durch effiziente Aggregation in ein einheitliches Format gebracht, wodurch Analysen auf dem Gebiet möglich wurden.

Die Auswertung dieser Daten zeigte, dass eine Darstellung der historischen Redundanz von Software durch Code-Clones durchaus möglich ist. Außerdem kann theoretisch aus den erhobenen und eventuell anderen bereichernden Daten ein Einfluss von Programmiersprachen-Features berechnet werden. Die Vermutungen zu den Unterschieden der Clone-Detection-Tools und den Vorteilen der Commit-Filterung konnten anhand der Datenanalyse ebenfalls bestätigt werden. Darüber hinaus zeigte eine Analyse der Schwellwerte eines Clone-Detection-Tools, dass Code-Clones nach ihren Eigenschaften gefiltert werden können. Die Analyse der Code-Clone-Verläufe erweitert um Daten zur Einführung von ausgewählten Features zeigte nur einen kleinen Einfluss auf die duplizierten Zeilen. Darüber hinaus wiesen detailliertere Analysen bestimmter Interessenpunkte vielerlei Fehlerquellen des Ansatzes auf. So können zum Beispiel Ergebnisse leicht durch größere, absichtliche Dopplungen in der Regel zu Testzwecken verzerrt werden.

Dennoch konnte gezeigt werden, dass eine Untersuchung der Redundanzentwicklung prinzipiell möglich ist. Der wahre Mehrwert dieser Arbeit zeigt sich jedoch hoffentlich in einer fortführenden Forschungsarbeit.

Anhang

Vorgehensweise der Literaturrecherchen

In diesem Kapitel soll die Vorgehensweise der Literaturrecherchen anhand der des Beispiels der Suche nach ähnlichen Arbeiten demonstriert werden. Die Literaturrecherche basiert hier stets auf einem Schema von Brocke et al. [15].

Aufbauend darauf wurden die erledigten Vorgänge in die vier Bereiche Reichweitendefinition, Konzeptualisierung des Themas, Literatursuche und Literaturanalyse aufgeteilt.

Eigenschaft	***Ausprägung***			
Fokus	**Ergebnisse**	**Methoden**	Theorien	Anwendung
Ziele	**Integration**	Kritisieren	Herausforderungen	
Perspektive	neutrale Darstellung	**Einnahme einer Position**		
Abdeckung	vollständig	vollständig selektiv	**repräsentativ**	**zentral**
Organisation	historisch	konzeptuell	methodisch	
Zielgruppe	Fachleute	**Wissenschaft**	Praxis	Öffentlichkeit

Tabelle 5: Einordnung eigener Literatursuche in die Taxonomie von Cooper

Für die Reichweitendefinition wurde in erster Linie die Taxonomie von Cooper [16] verwendet. Für das Beispiel der Suche von ähnlichen Arbeiten wurden die Felder, die in Tabelle 5 zu sehen sind, ausgewählt. Die ausgewählten Attribute sind dabei durch Unterstreichung markiert. Der Fokus sollte auf Arbeiten liegen, die Methoden beschreiben, die eventuell für diese Arbeit verwendet werden können, aber auch Ergebnisse waren von Interesse, weil diese zur Anreicherung der eigenen Ergebnisse verwendet werden sollten. Ziel war eine Integration, es sollten also Kompatibilitäten geprüft werden und vor allem Brücken zwischen den Code-Clones und eventuell anderen Themengebieten geschlagen werden. Abgesehen davon wurden vorweg auch Fragen definiert, die durch die Literaturrecherche beantwortet werden sollen. Die Perspektive ist die Einnahme einer Position. Dies war später vor allem bei der Auswahl der Literatur wichtig: Es gibt vielerlei Literatur zum Beispiel für die Beweise anderer Treiber von Programmierkonzepten. Da die Vermutung aber eher auf der Redundanz lag, wurde diese zunächst priorisiert. Die Abdeckung sollte dabei keinesfalls vollständig sein, weil die Literaturrecherche nicht das zentrale Element dieser Arbeit ist. Die Zielgruppe ist die Wissenschaft, weil diese Arbeit in erster Linie als Vorbereiter für eine weitere wissenschaftliche Arbeit gelten soll. Lediglich für die Organisation wurde keine Entscheidung getroffen.

Zur Einordnung des Themas (auch Topikalisierung genannt) wurde neben einer Definitionssammlung zum Auseinanderhalten der Begriffe ein Begriffsbaum angefertigt. Dieser ist in Abbildung 18 Seite VIII zu sehen. Das Ziel des Begriffsbaums war eine Aufschlüsselung von Teilbegriffe und eine Identifizierung von Synonymen für eine klarere Formulierung von Suchanfragen.

Key Words	***Datenbank***	***Ergebnisse (Auszug)***
Code Clone History	IEEExplore GoogleScholar	[52]
Programming Languages Evolution	IEEExplore GoogleScholar	--
Code Clones Version	IEEExplore GoogleScholar	[53], [75]
Programming Language Features Adoption	IEEExplore GoogleScholar	[5]
Code Clone Evolution	IEEExplore GoogleScholar	[46], [49], [51]

Tabelle 6: Dokumentation zur ersten Suche nach ähnlichen Arbeiten

Die Literatursuche bestand zunächst aus einer Journal-Suche, anschließend wurden verschiedene Datenbanken für die *key word search* verwendet. In der Beispiel-Suche nach ähnlichen Arbeiten wurden dabei keine Einschränkungen der Journals gemacht, weil dabei zunächst alle ähnlichen Arbeiten (der Informatik) betrachtet werden sollten, weil noch kein zu tiefes Wissen über die potentiell beteiligten Fachgebiete vorlag. Die verwendeten *key words*, die verwendeten Datenbanken und Ergebnisse, die später verwendet wurden, der ersten Suche zum Beispielthema sind in Tabelle 6 aufgezählt. Die Key Words wurden dabei immer einschließlich ihrer Synonyme und Flexionen verwendet.

Die gefundene Literatur wurde dann in Zotero in unterschiedlichen Ordnern ihren Themengebieten, wie Redundanz, Features, Code-Clones, Clone Evolution, Mining Software Repositories, Software Evolution, Clone-Detection und Methodik allgemein (Softwareentwicklung, Literaturrecherche etc.), zugeordnet. Anschließend wurde Literaturanalysen durchgeführt, um die Relevanz für diese Arbeit zu prüfen und Erkenntnisse zu gewinnen.

Solche Analysen wurden in ähnlicher, wenn auch weniger detaillierter Weise, für Clone-Detection-Tools, Treiber für Programmierkonzepte und Beweisansätze für Treiber durchgeführt. Neben diesen Literaturrecherchen wurden auch viele kleinere Suchen zu Themen mit weniger Abdeckungsrelevanz durchgeführt.

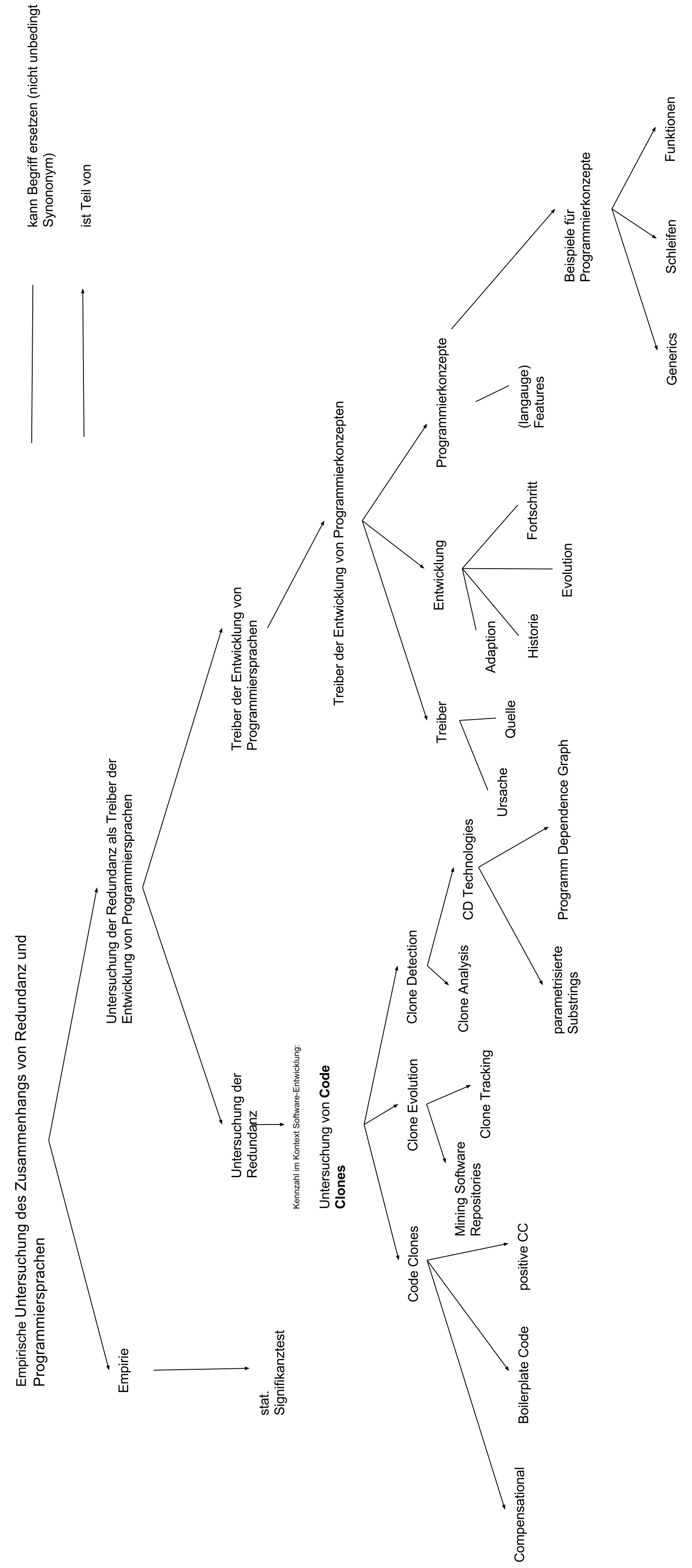


Abbildung 18: Begriffsbaum für Suchen zu ähnlichen Arbeiten.

Verwendete Computerkonfiguration

Die Parameter des für die Rechnungen verwendeten Computers für eine Nachvollziehbarkeit der Ergebnisse:

Betriebssystem*:* *Windows 10*
CPU*:* *AMD Phenom II X4 970 Processor mit vier Kernen je 3,50 GHz*
RAM*:* *8 GB*
Festplatte*:* *SSD mit etwa 500 MB/s Lese-/Schreibgeschwindigkeit*

Beispiele für Dateien

In diesem Kapitel sollen verschiedene Ausgaben gezeigt werden, um zu zeigen, welche Daten analysiert wurden.

Beispiel für Detailausgabe von Simian

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet href="simian.xsl" type="text/xsl"?>
<simian version="2.5.6">
    <check failOnDuplication="true" ignoreCharacterCase="true"
ignoreCurlyBraces="true" ignoreIdentifi-erCase="true" ignoreModifiers="true"
ignoreStringCase="true" threshold="6">
        <set lineCount="6" fingerprint="4ecd289ea151fc6db6503e45eb2f32f0">
            <block sourceFile="C:\Spring Framework\spring-framework\spring-
core\src\main\java\org\springframework\asm\MethodWriter.java"
startLineNumber="2298" endLineNum-ber="2305"/>
            <block sourceFile="C:\Spring Framework\spring-framework\spring-
core\src\main\java\org\springframework\asm\FieldWriter.java"
startLineNumber="296" endLineNum-ber="303"/>
        </set>
        <set lineCount="6" fingerprint="75b55d5f0994c820c7c27ed5a47a0ae2">
            <block sourceFile="C:\Spring Framework\spring-framework\spring-
core\src\main\java\org\springframework\util\CommonsLogWriter.java"
startLineNumber="60" end-LineNumber="70"/>
            <block sourceFile="C:\Spring Framework\spring-framework\spring-
core\src\main\java\org\springframework\util\CommonsLogWriter.java"
startLineNumber="47" end-LineNumber="56"/>
        </set>

            ...

        <set lineCount="33" fingerprint="9f45d54365e45a953cb961c14738a1ad">
            <block sourceFile="C:\Spring Framework\spring-framework\spring-
core\src\main\java\org\springframework\asm\FieldWriter.java"
startLineNumber="144" endLineNumber="187"/>
            <block sourceFile="C:\Spring Framework\spring-framework\spring-
core\src\main\java\org\springframework\asm\MethodWriter.java"
startLineNumber="520" endLineNumber="563"/>
        </set>
        <summary duplicateFileCount="44" duplicateLineCount="1080"
duplicateBlockCount="116" totalFileCount="341" totalRawLineCount="61127"
totalSignificantLineCount="20493" processingTime="921"/>
    </check>
</simian>
```

Programmtext 3: Verkürzte Ausgabe einer Clone-Detection von Simian

Beispiel für Detailausgabe von CCFinderX

```
ccfxraw0
      pa:db 50
s     2
u     +
t     12
w     f+g+w+
j     +
k     60m
preprocessed_file_postfix     .java.2_0_0_0.default.ccfxprep
pp    +
n     C:/CCFiles/

java
C:/SpringFramework/spring-framework/spring-
core/src\main\java\org\springframework\asm\AnnotationVisitor.java
   ç   C:/SpringFramework/spring-framework/spring-
core/src\main\java\org\springframework\asm\AnnotationWriter.java
ÑC:/SpringFramework/spring-framework/spring-
core/src\main\java\org\springframework\asm\Attribute.java
————————————————————————   ’  C:/SpringFramework/spring-framework/spring-
core/src\main\java\org\springframework\asm\ByteVector.java
….
```

Programmtext 4: Ergebnis einer Clone-Detection mit CCFinderX

```
FID   LOC   SLOC  CLOC  CVRL
1     169   39    0     0
2     371   200   38    0.19
3     255   59    0     0
4     339   166   38    0.228916
5     2498  1657  291   0.175619
6     322   71    18    0.253521
7     1776  679   164   0.241532
8     144   2     0     0
9     75    2     0     0
10    152   35    23    0.657143
11    329   178   124   0.696629
```

Programmtext 5: Auszug aus Metrikausgabe von CCFinderX

Beispiel für aggregierte Daten

```
Commit#;CommitDate;LOC;SLOC;DLOC;#CC
23042;22_11_17__0810;553363;289342;55481;4761
23034;21_11_17__0943;552919;289061;55400;4753
23026;15_11_17__0308;552871;289028;55400;4753
23018;13_11_17__0444;552822;288994;55398;4753
23010;09_11_17__0825;552755;288948;55398;4753
23002;05_11_17__0741;552679;288909;55404;4754
22994;02_11_17__1804;552567;288840;55406;4754
22986;28_10_17__2057;552464;288776;55418;4754
22978;11_10_17__0746;552466;288772;55404;4754
22970;07_10_17__1301;552419;288746;55382;4751
22962;30_09_17__1832;552395;288732;55382;4751
22954;27_09_17__0632;552426;288753;55394;4751
22946;26_09_17__0528;552364;288724;55392;4751
22938;16_08_17__2304;528377;275714;53664;4534
```

Programmtext 6: Auszug aus einer CSV-Datei als Ergebnis einer Datenerhebung

Abbildungsverzeichnis

Tabellenverzeichnis

Programmtextverzeichnis

Abkürzungsverzeichnis

CCR – *code clone ratio* – Code-Clone-Anteil

DLOC – *duplicated lines of code* – duplizierte Code-Zeilen

KB – Kilobyte

LOC – *lines of code* – Zeilen Programmcode

VKS – Versionskontrollsystem

MB – Megabyte

MSR – Mining Software Repositories

SLOC – *source lines of code* – (relevante) Quellcode-Zeilen

Literaturverzeichnis

[1] „J2SE(TM) 5.0 New Features“. [Online]. Verfügbar unter: https://docs.oracle.com/javase/1.5.0/docs/relnotes/features.html. [Zugegriffen: 19-Mai-2017].

[2] „What’s New in JDK 8“. [Online]. Verfügbar unter: http://www.oracle.com/technetwork/java/javase/8-whats-new-2157071.html. [Zugegriffen: 22-Dez-2017].

[3] „What’s New in Python 2.7 — Python 2.7.14 documentation“. [Online]. Verfügbar unter: https://docs.python.org/2.7/whatsnew/2.7.html. [Zugegriffen: 04-Jan-2018].

[4] B. Wagner, „Neues in C# 7 – Leitfaden für C#“. [Online]. Verfügbar unter: https://docs.microsoft.com/de-de/dotnet/csharp/whats-new/csharp-7. [Zugegriffen: 04-Jan-2018].

[5] C. Parnin, C. Bird, und E. Murphy-Hill, „Java Generics Adoption: How New Features Are Introduced, Championed, or Ignored“, in *Proceedings of the 8th Working Conference on Mining Software Repositories*, New York, NY, USA, 2011, S. 3–12.

[6] H. A. Basit, D. C. Rajapakse, und S. Jarzabek, „An Empirical Study on Limits of Clone Unification Using Generics.“, in *SEKE*, 2005, S. 109–114.

[7] U. W. Eisenecker, „Redundancy in Source Programs On The Evolution of Programming Concepts“, 17-Apr-2017. [Vortrag].

[8] H. M. Sneed und R. Seidl, *Softwareevolution: Erhaltung und Fortschreibung bestehender Softwaresysteme*, 1. Aufl. Heidelberg: dpunkt.verl, 2013.

[9] M. James, „The Evolution Of Programming Languages“, *I Programmer*, 22-Juli-2015. [Online]. Verfügbar unter: http://www.i-programmer.info/news/98-languages/8809-the-evolution-of-programming-languages.html. [Zugegriffen: 04-Jan-2018].

[10] „Generic Types Java™ Tutorials“. [Online]. Verfügbar unter: https://docs.oracle.com/javase/tutorial/java/generics/types.html. [Zugegriffen: 03-Jan-2018].

[11] „Welcome to Python.org“, *Python.org*. [Online]. Verfügbar unter: https://www.python.org/. [Zugegriffen: 04-Jan-2018].

[12] „Simian - Similarity Analyser | Duplicate Code Detection for the Enterprise | Features“. [Online]. Verfügbar unter: http://www.harukizaemon.com/simian/features.html. [Zugegriffen: 14-Okt-2017].

[13] T. Kamiya, S. Kusumoto, und K. Inoue, „CCFinder: a multilinguistic token-based code clone detection system for large scale source code“, *IEEE Trans. Softw. Eng.*, Bd. 28, Nr. 7, S. 654–670, Juli 2002.

[14] „R: The R Project for Statistical Computing“. [Online]. Verfügbar unter: https://www.r-project.org/. [Zugegriffen: 28-Dez-2017].

[15] J. Brocke *u. a.*, „Reconstructing the Giant: On the Importance of Rigour in Documenting the Literature Search Process“, in *ECIS 2009 Proceedings*, 2009, Bd. 9, S. 2206–2217.

[16] H. M. M. Cooper, „Organizing knowledge syntheses: A taxonomy of literature reviews“, *Knowl. Soc.*, Bd. 1, Nr. 1, S. 104–126, März 1988.

[17] „Google Scholar“. [Online]. Verfügbar unter: https://scholar.google.de/schhp?hl=de&as_sdt=0,5. [Zugegriffen: 04-Jan-2018].

[18] „IEEE Xplore Digital Library“. [Online]. Verfügbar unter: http://ieeexplore.ieee.org/Xplore/home.jsp. [Zugegriffen: 04-Jan-2018].

[19] „EBSCO Information Services Service Selection Page“. [Online]. Verfügbar unter: http://search.epnet.com/. [Zugegriffen: 04-Jan-2018].

[20] „Zotero | Home“. [Online]. Verfügbar unter: https://www.zotero.org/. [Zugegriffen: 04-Jan-2018].

[21] „Basic Declarations & Definitions“. [Online]. Verfügbar unter: http://scala-lang.org/files/archive/spec/2.12/04-basic-declarations-and-definitions.html. [Zugegriffen: 02-Jan-2018].

[22] B. Wagner, „Lambda-Ausdrücke (C#-Programmierhandbuch)“. [Online]. Verfügbar unter: https://docs.microsoft.com/de-de/dotnet/csharp/programming-guide/statements-expressions-operators/lambda-expressions. [Zugegriffen: 02-Jan-2018].

[23] „Generic Declarations“. [Online]. Verfügbar unter: http://www.adaic.org/resources/add_content/standards/95lrm/ARM_HTML/RM-12-1.html. [Zugegriffen: 02-Jan-2018].

[24] „Templates - C++ Tutorials“. [Online]. Verfügbar unter: http://www.cplusplus.com/doc/oldtutorial/templates/. [Zugegriffen: 02-Jan-2018].

[25] „GitHut - Programming Languages and GitHub“. [Online]. Verfügbar unter: http://githut.info/. [Zugegriffen: 04-Jan-2018].

[26] „Build software better, together“, *GitHub*. [Online]. Verfügbar unter: https://github.com. [Zugegriffen: 04-Jan-2018].

[27] „GitHub Octoverse 2017“, *GitHub Octoverse 2017*. [Online]. Verfügbar unter: https://octoverse.github.com/. [Zugegriffen: 04-Jan-2018].

[28] „Java version history“, *Wikipedia*, 20-Dez-2017. [Online]. Verfügbar unter: https://en.wikipedia.org/w/index.php?title=Java_version_history&oldid=816285858. [Zugegriffen: 04-Jan-2018].

[29] „Oracle Technology Network for Java Developers | Oracle Technology Network | Oracle“. [Online]. Verfügbar unter: http://www.oracle.com/technetwork/java/index.html. [Zugegriffen: 04-Jan-2018].

[30] „Lambdas everywhere“. [Online]. Verfügbar unter: http://dobegin.com/lambda-functions-everywhere/. [Zugegriffen: 06-Jan-2018].

[31] „Lambda Expressions (The Java™ Tutorials)“. [Online]. Verfügbar unter: https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html. [Zugegriffen: 03-Jan-2018].

[32] L. Cardelli, „Type Systems“, *ACM Comput Surv*, Bd. 28, Nr. 1, S. 263–264, März 1996.

[33] R. Lämmel und S. P. Jones, „Scrap Your Boilerplate with Class: Extensible Generic Functions“, in *Proceedings of the Tenth ACM SIGPLAN International Conference on Functional Programming*, New York, NY, USA, 2005, S. 204–215.

[34] „What’s New In Python 3.7 — Python 3.7.0a3 documentation“. [Online]. Verfügbar unter: https://docs.python.org/3.7/whatsnew/3.7.html. [Zugegriffen: 01-Jan-2018].

[35] „What’s New in Python 2.5 — Python 2.7.14 documentation“. [Online]. Verfügbar unter: https://docs.python.org/2/whatsnew/2.5.html. [Zugegriffen: 01-Jan-2018].

[36] „Duden | Re-d-un-danz | Rechtschreibung, Bedeutung, Definition, Synonyme, Herkunft“. [Online]. Verfügbar unter: https://www.duden.de/rechtschreibung/Redundanz. [Zugegriffen: 01-Jan-2018].

[37] „Wortschatz – deu_newscrawl_2011 – Redundanz“. [Online]. Verfügbar unter: http://corpora.uni-leipzig.de/de/res?corpusId=deu_newscrawl_2011&word=Redundanz. [Zugegriffen: 01-Jan-2018].

[38] H. Glück, Hrsg., *Metzler-Lexikon Sprache*, 4., Aktualisierte und überarb. Aufl. Stuttgart: Metzler, 2010.

[39] „findbugs: The new home of the FindBugs project“, 31-Dez-2017. [Online]. Verfügbar unter: https://github.com/findbugsproject/findbugs. [Zugegriffen: 01-Jan-2018].

[40] „UCDetector“. [Online]. Verfügbar unter: http://www.ucdetector.org/. [Zugegriffen: 01-Jan-2018].

[41] „Checkstyle - project“, 15-Dez-2017. [Online]. Verfügbar unter: https://github.com/checkstyle/checkstyle.

[42] C. K. Roy, J. R. Cordy, und R. Koschke, „Comparison and Evaluation of Code Clone Detection Techniques and Tools: A Qualitative Approach“, *Sci Comput Program*, Bd. 74, Nr. 7, S. 470–495, Mai 2009.

[43] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, und E. Merlo, „Comparison and Evaluation of Clone Detection Tools“, *IEEE Trans Softw Eng*, Bd. 33, Nr. 9, S. 577–591, Sep. 2007.

[44] M. Gabel, L. Jiang, und Z. Su, „Scalable detection of semantic clones“, in *2008 ACM/IEEE 30th International Conference on Software Engineering*, 2008, S. 321–330.

[45] R. Komondoor und S. Horwitz, „Using Slicing to Identify Duplication in Source Code“, in *Static Analysis*, 2001, S. 40–56.

[46] N. Göde, „Evolution of Type-1 Clones“, in *2009 Ninth IEEE International Working Conference on Source Code Analysis and Manipulation*, 2009, S. 77–86.

[47] D. Rattan, R. Bhatia, und M. Singh, „Software clone detection: A systematic review“, *Inf. Softw. Technol.*, Bd. 55, Nr. 7, S. 1165–1199, Juli 2013.

[48] R. K. Saha, C. K. Roy, K. A. Schneider, und D. E. Perry, „Understanding the evolution of Type-3 clones: An exploratory study“, in *2013 10th Working Conference on Mining Software Repositories (MSR)*, 2013, S. 139–148.

[49] S. Bazrafshan, „Evolution of Near-Miss Clones“, in *2012 IEEE 12th International Working Conference on Source Code Analysis and Manipulation*, 2012, S. 74–83.

[50] A. Sheneamer und J. Kalita, „A Survey of Software Clone Detection Techniques“, *Int. J. Comput. Appl.*, Bd. 137, S. 1–21, März 2016.

[51] E. Duala-Ekoko und M. P. Robillard, „Tracking Code Clones in Evolving Software“, in *29th International Conference on Software Engineering (ICSE’07)*, 2007, S. 158–167.

[52] A. Goon, Y. Wu, M. Matsushita, und K. Inoue, „Evolution of code clone ratios throughout development history of open-source C and C++ programs“, in *2017 IEEE 11th International Workshop on Software Clones (IWSC)*, 2017, S. 1–7.

[53] T. Lavoie und E. Merlo, „How much really changes? A case study of firefox version evolution using a clone detector“, in *2013 7th International Workshop on Software Clones (IWSC)*, 2013, S. 83–89.

[54] R. Koschke, „Survey of Research on Software Clones“, *Dagstuhl Semin. Proc.*, 2007.

[55] C. K. Roy und J. R. Cordy, „A survey on software clone detection research“, *Queen’s Sch. Comput. TR*, Bd. 541, Nr. 115, 2007.

[56] „Definition Best-of-Breed Erklärung Best-of-Breed“. [Online]. Verfügbar unter: http://www.softselect.de/business-software-glossar/best-of-breed. [Zugegriffen: 01-Jan-2018].

[57] L. Chen und A. Avizienis, „N-VERSION PROGRAMMINC: A FAULT-TOLERANCE APPROACH TO RELIABILITY OF SOFTWARE OPERATION“, in *, Twenty-Fifth International Symposium on Fault-Tolerant Computing, 1995, Highlights from Twenty-Five Years*, 1995, S. 113-.

[58] C. J. Kapser und M. W. Godfrey, „Cloning considered harmful: patterns of cloning in software“, *Empir. Softw. Eng.*, Bd. 13, Nr. 6, S. 645, Dez. 2008.

[59] B. A. Malloy und J. F. Power, „Quantifying the Transition from Python 2 to 3: An Empirical Study of Python Applications“, in *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 2017, S. 314–323.

[60] P. Capek, E. Kral, und R. Senkerik, „Towards an Empirical Analysis of .NET Framework and C# Language Features’ Adoption“, in *2015 International Conference on Computational Science and Computational Intelligence (CSCI)*, 2015, S. 865–866.

[61] R. Fuhrer, F. Tip, A. Kiezun, J. Dolby, und M. Keller, „Efficiently Refactoring Java Applications to Use Generic Libraries“, in *Proceedings of the 19th European Conference on Object-Oriented Programming*, Berlin, Heidelberg, 2005, S. 71–96.

[62] „Git - Ein Git Repository anlegen“. [Online]. Verfügbar unter: https://git-scm.com/book/de/v1/Git-Grundlagen-Ein-Git-Repository-anlegen. [Zugegriffen: 01-Jan-2018].

[63] „GNU: Kategorien freier und unfreier Software“. [Online]. Verfügbar unter: http://www.gnu.org/philosophy/categories.de.html. [Zugegriffen: 01-Jan-2018].

[64] „Open-Source-Software — Enzyklopaedie der Wirtschaftsinformatik“. [Online]. Verfügbar unter: http://enzyklopaedie-der-wirtschaftsinformatik.de/lexikon/uebergreifendes/Kontext-und-Grundlagen/Markt/Open-Source-Software/index.html/?searchterm=open-source. [Zugegriffen: 01-Jan-2018].

[65] „DWDS – Das Wortauskunftssystem zur deutschen Sprache in Geschichte und Gegenwart“. [Online]. Verfügbar unter: https://www.dwds.de/d/korpora. [Zugegriffen: 03-Jan-2018].

[66] E. Tempero *u. a.*, „The Qualitas Corpus: A Curated Collection of Java Code for Empirical Studies“, in *2010 Asia Pacific Software Engineering Conference*, 2010, S. 336–345.

[67] „Eclipse Checkstyle Plugin“. [Online]. Verfügbar unter: http://eclipse-cs.sourceforge.net/#!/. [Zugegriffen: 05-Jan-2018].

[68] „Eclipse Checkstyle Plug-in / Git“. [Online]. Verfügbar unter: https://sourceforge.net/p/eclipse-cs/git/ci/master/tree/. [Zugegriffen: 05-Jan-2018].

[69] „JDT/Core project repository (eclipse.jdt.core)“, 24-Dez-2017. [Online]. Verfügbar unter: https://github.com/eclipse/eclipse.jdt.core. [Zugegriffen: 03-Jan-2018].

[70] „FindBugs™ - Find Bugs in Java Programs“. [Online]. Verfügbar unter: http://findbugs.sourceforge.net/. [Zugegriffen: 01-Jan-2018].

[71] A. Loskutov, „[FB-Discuss] Announcing SpotBugs as FindBugs successor“, 21-Sep-2017. [Online]. Verfügbar unter: https://mailman.cs.umd.edu/pipermail/findbugs-discuss/2017-September/004383.html. [Zugegriffen: 03-Jan-2018].

[72] „spring-framework: Spring Framework“, 19-Dez-2017. [Online]. Verfügbar unter: https://github.com/spring-projects/spring-framework.

[73] „SQuirreL SQL Client Home Page“. [Online]. Verfügbar unter: http://squirrel-sql.sourceforge.net/. [Zugegriffen: 03-Jan-2018].

[74] S. Imran, M. Buchheit, B. Hollunder, und U. Schreier, „Tool Chains in Agile ALM Environments: A Short Introduction“, in *On the Move to Meaningful Internet Systems: OTM 2015 Workshops*, 2015, S. 371–380.

[75] A. Hanjalić, „ClonEvol: Visualizing software evolution with code clones“, in *2013 First IEEE Working Conference on Software Visualization (VISSOFT)*, 2013, S. 1–4.

[76] N. B. Ruparelia, „The History of Version Control“, *SIGSOFT Softw Eng Notes*, Bd. 35, Nr. 1, S. 5–9, Jan. 2010.

[77] R. Somasundaram, *Git: Version Control for Everyone*. Birmingham: Packt Publishing, 2013.

[78] „MSR 2017“. [Online]. Verfügbar unter: http://2017.msrconf.org/#/home. [Zugegriffen: 08-Sep-2017].

[79] „GitPython Documentation — GitPython 2.1.7 documentation“. [Online]. Verfügbar unter: http://gitpython.readthedocs.io/en/stable/index.html. [Zugegriffen: 09-Okt-2017].

[80] „Git - git-commit Documentation“. [Online]. Verfügbar unter: https://www.git-scm.com/docs/git-commit. [Zugegriffen: 05-Dez-2017].

[81] E. Burd und J. Bailey, „Evaluating clone detection tools for use during preventative maintenance“, in *Proceedings / Second IEEE International Workshop on Source Code Analysis and Manipulation*, Los Alamitos, Calif., 2002, S. 36–43.

[82] K. Solanki und S. Kumari, „Comparative study of software clone detection techniques“, in *2016 Management and Innovation Technology International Conference (MITicon)*, 2016, S. 152–156.

[83] S. Ducasse, M. Rieger, und S. Demeyer, „A language independent approach for detecting duplicated code“, in *IEEE International Conference on Software Maintenance, 1999. (ICSM ’99) Proceedings*, 1999, S. 109–118.

[84] I. D. Baxter, A. Yahin, L. Moura, M. Sant’Anna, und L. Bier, „Clone detection using abstract syntax trees“, in *Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272)*, 1998, S. 368–377.

[85] J. Krinke, „Identifying Similar Code with Program Dependence Graphs“, in *Proceedings of the Eighth Working Conference on Reverse Engineering (WCRE’01)*, Washington, DC, USA, 2001, S. 301–309.

[86] N. Davey, P. Barson, S. Field, R. Frank, und D. Tansley, „The development of a software clone detector“, *Int. J. Appl. Softw. Technol.*, S. 219–236, 1995.

[87] L. Li, H. Feng, W. Zhuang, N. Meng, und B. Ryder, „CCLearner: A Deep Learning-Based Clone Detection Approach“, in *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2017, S. 249–260.

[88] „Mockito framework site“. [Online]. Verfügbar unter: http://site.mockito.org/. [Zugegriffen: 15-Dez-2017].

[89] „TestNG - Welcome“. [Online]. Verfügbar unter: http://testng.org/doc/. [Zugegriffen: 15-Dez-2017].

[90] „Tutorial of CLI Tool ccfx“. [Online]. Verfügbar unter: http://www.ccfinder.net/doc/10.2/en/tutorial-ccfx.html. [Zugegriffen: 14-Okt-2017].

[91] H. Sajnani, V. Saini, J. Svajlenko, C. K. Roy, und C. V. Lopes, „SourcererCC: Scaling Code Clone Detection to Big-Code“, in *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, 2016, S. 1157–1168.

[92] E. Juergens, F. Deissenboeck, B. Hummel, und S. Wagner, „Do Code Clones Matter?“, in *Proceedings of the 31st International Conference on Software Engineering*, Washington, DC, USA, 2009, S. 485–495.

[93] J. Svajlenko und C. K. Roy, „CloneWorks: A Fast and Flexible Large-Scale Near-Miss Clone Detection Tool“, in *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, 2017, S. 177–179.

[94] G. Böckle, P. Knauber, K. Pohl, und K. Schmid, *Software-Produktlinien: Methoden, Einführung und Praxis*, 1. Aufl. Heidelberg: dpunkt, 2004.

[95] U. Kamps, „Gabler Lexikon: Pearson Korrelationskoeffizient“. [Online]. Verfügbar unter: http://wirtschaftslexikon.gabler.de/Definition/korrelationskoeffizient.html. [Zugegriffen: 03-Jan-2018].

[96] J. Svajlenko und C. K. Roy, „Evaluating clone detection tools with BigCloneBench“, in *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2015, S. 131–140.

[97] V. Nguyen, S. Deeds-Rubin, T. Tan, und B. W. Boehm, „A SLOC Counting Standard“, in *COCOMO II Forum*, 2007, Bd. 2007, S. 1–16.
[98] „Dialect (Hibernate JavaDocs)“. [Online]. Verfügbar unter: https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/dialect/Dialect.html. [Zugegriffen: 22-Dez-2017].

Eidesstattliche Erklärung

Harnisch, Björn Ole

Name, Vorname

3722194

Matrikelnummer

Ich versichere, dass ich die Masterarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Darüber hinaus versichere ich, dass die elektronische Version der Masterarbeit mit der gedruckten Version übereinstimmt.

Ort, Datum, Unterschrift