Institut für Softwaretechnologie
Abteilung Software Engineering
Universität Stuttgart
Universitätsstraße 38

Diplomarbeit Nr. 3651

Inkonsistente Klone und Fehler in Software

Kamer Kaya

Studiengang:	Informatik
Prüfer:	Prof. Dr. rer. nat. Stefan Wagner
Betreuer:	M. Sc. Asim Abdulkhaleq
begonnen am:	01. April 2014
beendet am:	01. Oktober 2014

CR-Klassifikation: D.2.5, D.2.7, D.2.9, D.2.12, H.2.1, H.2.4

Inhaltsverzeichnis

Danksagung

Ich möchte mich an dieser Stelle bei allen bedanken, die mich während meiner Arbeit und während des Studiums unterstützt haben.

An erster Stelle geht ein besonderer Dank an meine Familie, die mich während meines Studiums mit Sorgsamkeit und hohem Motivationsaufwand unterstützt haben.

Besonders meinem Betreuer Asim Abdulkhaleq gilt ein Dank, der sich immer Zeit für meine Arbeit genommen hat und mich über die sechs Monate mit viel Motivation und guten Ideen betreut hat.

Ebenfalls möchte ich mich bei meinem Referenten Herrn Prof. Dr. Stefan Wagner bedanken, der mir die Arbeit ermöglicht hat und stets für kritische Situationen der Diplomarbeit immer ansprechbar war.

Ein besonderer Dank gilt an meinem Betreuer Alexander Paar bei der TWT GmbH, der mir eine große Unterstützung in der Einarbeitungsphase und der Analysephase war. Weiterhin möchte ich mich beim Herrn Thorsten Scheibler bedanken, der in Zusammenarbeit mit der Universität Stuttgart die Diplomarbeit ermöglichte. An die Mitarbeiter der TWT GmbH die für Fragen immer offen waren bedanke ich mich ebenfalls.

Zu guter Letzt möchte ich mich bei meinen Freundinnen Zeynep Öztürk, Duygu Söylemez und Hatice Aydeniz bedanken, die während dem Studium geduldig mit mir waren und immer ein offenes Ohr für mich hatten.

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

CF	Codefragment
AST	Abstrakter Syntaxbaum
PDG	Program Dependence Graph
ConQat	Continuous Quality Assessment Toolkit
ERM	Entity-Relantionship-Modell
SQL	Structured Query Language
C	Alle Klongruppen
IC	Inkonsistente Klongruppen
BIC	Fehlerhafte Inkonsistente Klongruppen
KF	Fehlerbehobene inkonsistente Klongruppen
MIS	Modifizierte inkonsistente Klonstellen
ZMIS	Zeitgleich modifizierte inkonsistente Klonstellen
FK	Fehlerhafte Klonklassen

Begriffsverzeichnis

ConQat	ConQat ist ein Werkzeug zur kontinuierlichen Software-Qualitätskontrollanalyse. ConQat wird in dieser Arbeit als Klonerkennungswerkzeug eingesetzt.
Klone	Klone sind redundante Quelltextabschnitte in einem Code, die an mehreren Stellen eines Softwaresystems vorkommen.
Inkonsistente Klone	Redundante Quelltextabschnitte, die mit Modifikationen versehen sind.
Gap	Ungleiche Abschnitte in den Inkonsistenten Klonen werden Gap genannt.
Versionskontrolle	Unter Versionskontrolle hingegen versteht man den Prozess der Verwaltung mehrerer Versionen einer Information
Versionskontrollsystem	Tools die das Automatisieren der Versionskontrolle ermöglichen, werden Versionskontrollsysteme genannt.
Mercurial	Mercurial ist ein plattformunabhängiges, verteiltes Versionskontrollsystem.
Issue-/Bug-Tracking System	Ein Issue-/Bug-Tracking-System ist ein Werkzeug, mit dem diverse Aufgaben in einem Projekt bearbeitet werden.
FogBugz	Das webbasierte System FogBugz ist ein Projektmanagement-system sowie ein Issue-/Bug-Tracking-System, welche umfangreiche Funktionalitäten für Entwicklerteams anbietet.
Kiln	Kiln ist ein webbasiertes System für das Quellcodehosting von Mercurial.
TortoiseHg	Das Tool TortoiseHg ist ein einfach zu bedienendes Frontend und steht mit ihrer graphischen Oberfläche zur Verfügung, um die Benutzung von Mercurial ohne Kommandozeilenbefehle durchzuführen.
Bug	Ein Bug ist ein fehlverhalten in einem Softwaresystem.

Kurzfassung

Softwareklone in einem System erfordern eine hohe Vorsicht im Entwicklungszyklus eines Softwareprojekts. Viele Forscher sind der Ansicht, dass Klone vor allem inkonsistente Klone die Ursache diverser Fehler in Softwaresystemen sind, die sich unbemerkt einschleichen und nicht nachverfolgt werden können. Vor allem die Auswirkungen der inkonsistenten Klone liegen im Interesse vieler Forschungsarbeiten. Jedoch liegen die Forschungsergebnisse der Studien weit auseinander. Im Rahmen dieser Diplomarbeit werden die Auswirkungen der inkonsistenten Klone in einem Softwaresystem analysiert. Des Weiteren analysiert diese Arbeit auf empirischer Basis im Rahmen eines Studiendesigns den Zusammenhang der Inkonsistenten und Fehlern in Softwaresystemen. Die Studie wurde auf drei Industriesystemen durchgeführt und ergab als Resultat, dass Entwickler über fast alle Klonstellen einer Klonklasse informiert sind und diese bei Bedarf zu 58%-92% zeitgleich modifizieren. Es sind lediglich 3%-33% der inkonsistenten Klonklassen fehlerbehaftet und stellen somit eine geringe Gefahr für die Softwareentwicklung. Die umfangreiche Analyse gab den Beschluss, dass die Inkonsistenzen im Vergleich zu exakten Klonen mindestens weniger als die Hälfte einen Fehler verursachen. Weiterhin wiederlegt die Studie, dass durch das Klonen aus Bibliotheken, Klone eine erheblich geringe Anzahl an Fehler darstellen und nach bis zu vier Jahren Klonzeit keinen einzigen Fehler in der gesamten Revisionshistorie verursacht haben. Die Ergebnisse dieser Arbeit beweisen, dass Entwickler bewusst Klonen und dass es durch das bewusste Klonen keinen erhöhten Zusammenhang zwischen inkonsistente Klone und Fehler gibt.

1 Einleitung

Dieses Kapitel verschafft einen Überblick über das Thema der Diplomarbeit. Zunächst schafft der erste Teil einen Überblick über das Umfeld und darauf wird die Motivation der Arbeit dargestellt. Anschließend wird das Ziel der Arbeit beschrieben. Der letzte Abschnitt veranschaulicht den Aufbau der Diplomarbeit, um einen Gesamtüberblick über das Forschungsthema zu verschaffen.

1.1 Überblick

Ein wichtiger Bestandteil der Softwareentwicklung ist die Wartung. Fälschlicherweise werden der Aufwand und die Kosten für die Wartung im Gegensatz zur Implementierung unterschätzt. Empirische Studien haben wiederlegt, dass die Kosten für die Wartung und Entwicklung ein wichtiger Aspekt sind und bis zu 80% der Gesamtkosten und des Aufwandes betragen [1]. Die Ursache für die hohen Kosten ist, dass mit zunehmendem Alter durch die fehlende Planung für die Weiterentwicklung sowie hohem Zeitdruck und mangelnde Möglichkeiten für die langfristige Planung, der Softwareumfang steigt. Forscher haben lange versucht diese Wartungskosten zu minimieren. Für die Verbesserung der Entwicklung gab es einige Arbeiten, welche die Prozessmodelle sowie die Werkzeug- und Sprachunterstützung verbessert haben, um den schlechten Eigenschaften des Codes entgegenzuwirken, die sich negativ auf die Wartungskosten auswirken. Daraus resultierend können die Wartungskosten reduziert werden. Schlechte Wartbarkeit eines Softwaresystems kann oft zu einem schlechten Code führen, der schwer fehleranfällig, verstehbar und veränderbar ist [2], [3]. Ein wichtiger Faktor für einen schlechten Code können Code-Duplizierungen oder Klone sein. Klone sind kopierte Quelltexte, die vor allem durch Copy&Paste entstehen, die über die Implementierungsdetails im gesamten System verteilt werden. Fowler [4] betrachtet das Klonen als „bad smells“ und als den wichtigsten Indikator für die schlechte Wartbarkeit. Durch das Klonen besteht die Gefahr, dass Fehler in den kopierten Codeabschnitten stillschweigend repliziert werden, ohne dass es der klonende Entwickler bemerkt. Speziell inkonsistente Klone lassen vermuten, dass Fehler eingeführt und nicht behoben werden. Das Klonen hat also den Nachteil, dass sich die Fehleranfälligkeit des Systems erhöht und dass sich der Codeumfang vergrößert, welches das Verständnisproblem des Systems erschwert. Welchen Einfluss Klone auf die Wartbarkeit und Zuverlässigkeit haben, untersuchen Monden u.a. [5] in einer umfangreichen Studie. Des Weiteren gab es eine empirische Arbeit zur Analyse der Anzahl der Fehler für inkonsistente Klone, die durch das Ändern des geklonten Codeabschnitts entstehen [6]. Es wurden ebenfalls viele Forschungsarbeiten für das automatische Finden bzw. auch für das automatische Ändern von Klonen gewidmet, um der Fehleranfälligkeit des geklonten Codes entgegenzuwirken [7, 8, 9]. Gleichzeitig

präsentiert eine andere Forschungsgruppe Beweise dafür, dass es keinen erhöhten Zusammenhang zwischen Klonen und Fehler gibt [10].

1.2 Motivation

Es gibt diverse Ansätze und Ergebnisse zu Fehlern in Klone, die sich zum Teil widersprechen. Prinzipiell besteht die Annahme, dass allein durch den Größenzuwachs unnötige Aufwände entstehen, aber speziell die inkonsistenten Klone lassen vermuten, dass auch Fehler eingeführt oder nicht behoben werden.

Beispielsweise analysiert die empirische Studie, an der Forschungsgruppen aus Industrie und Open Source beteiligt sind, dass bei unbewussten inkonsistenten Klonen, jeder zweiter Klon einen Fehler enthält [6]. Eine andere Studie von Rahman, Bird und Dvanbu [10] konnte aber keinen erhöhten Zusammenhang zwischen Klonen und Fehlern finden. Es gibt also verschiedene Ansätze und verschiedene Ergebnisse zu Fehlern in Klone.

Aus diesem Anlass entstand die Diplomarbeit in Kooperation mit der Entwicklungsabteilung der TWT GmbH in Stuttgart Vaihingen. Bei der TWT GmbH existieren Softwaresysteme, die über Jahre hinweg entwickelt werden und aus langen Revisionshistorien bestehen. Die Architektur der Systeme wurde kontinuierlich an neue Anforderungen wie sich ändernde Kundenwünsche und neue Technologien angepasst.

Es wurden von der TWT GmbH drei Softwaresysteme, welche Kundenprojekte für größere Firmen sind, für die Analyse zur Verfügung gestellt. Die Systeme bestehen aus einer hohen Anzahl von Revisionen, die als Basis für die Arbeit dienen.

1.3 Zielstellung

Das Ziel der Arbeit ist es, Hinweise dafür zu finden, ob inkonsistente Klone Fehler verursachen. Im Zusammenhang dazu, soll ermittelt werden, unter welchen Umständen Fehler durch Inkonsistente entstehen. Hierzu soll die empirische Basis für den Zusammenhang zwischen inkonsistenten Klonen und Fehlern erhöht werden, um eine genauere Aussage machen zu können. Dabei sollen die vorhandenen Studien auf Systemen der TWT GmbH repliziert werden. Die Analyse der Studie soll in einem Studiendesign erfasst werden.

1.4 Aufbau der Diplomarbeit

Die Diplomarbeit beginnt mit der Einleitung, die einen Überblick über das gesamte Thema verschafft. Anschließend werden der Hintergrund der Diplomarbeit, die Motivation und das Ziel beschrieben. Die Grundlagen erfolgen in Kapitel 2 und umfassen eine Einführung in die Termini und Definitionen sowie in die unterschiedlichen Klonerkennungsverfahren im Detail. Das darauf folgende Kapitel 3 befasst sich mit dem bisherigen Forschungsstand bzw. den Forschungsarbeiten. Hierbei werden die Gründe und Folgen des Klonens geschildert und verschiedene Arbeiten zur Erkennung inkonsistenter Klone und Bugs dargestellt. Die relevanten Werkzeuge und Softwaresysteme für die Arbeit, wie ConQat, Mercurial und FogBugz werden im vierten Kapitel beschrieben. In Kapitel 5 beginnt der eigentliche Teil der Diplomarbeit, das Studiendesign. Das Studiendesign mit dem Repository Mining wird erklärt und die Forschungsarbeit durchgeführt sowie die Forschungsergebnisse beschrieben, um die Ergebnisse in Kapitel 6 auszuwerten und zu beurteilen. In einem abschließenden Fazit im siebten Kapitel werden die Ergebnisse zusammengefasst und beurteilt. Zudem werden weitere potenzielle Weiterentwicklungsbereiche dargestellt und auf zusätzliche offene Forschungsfragen angedeutet sowie Ideen für deren Lösungen geliefert.

2 Grundlagen Softwareklone

Bei der Betrachtung diverser Studien wird schnell festgestellt, dass verschiedene Definitionen zum Begriff Klon existieren. Genauso sind unterschiedliche Ansichten zu den Subbegriffen des Klons vorhanden. Daher gibt dieses Kapitel eine detaillierte Einführung in die zugrundeliegende Materie, die eine Voraussetzung für das Verständnis der Arbeit ist. Das Kapitel beschreibt die verschiedenen Klonarten und Klontypen sowie den Aufbau der Klone, deren Forschungsgebiet sehr breit gefächert ist. Dementsprechend spiegelt sich das auch in den unterschiedlichen Klonerkennungsansätzen wieder. Nach einer ausführlichen Beschreibung des allgemeinen Klonerkennungsprozesses für inkonsistente Klone werden die wichtigsten und bekanntesten Klonerkennungsverfahren im Detail beschrieben.

2.1 Definition Klone

Die Definition des Klonens lässt sich in der Softwareentwicklung nicht präzise und ohne weiteres festlegen. Daher umfasst das Kapitel zum besseren Verständnis der Klone, beginnend mit einer allgemeinen Definition, detaillierte Definitionen zu verschiedenen Klonarten.

2.1.1 Allgemeine Definition

Die allgemeine Definition von Ira Baxter dient als Grundlage für das Verständnis der Klone.

> *“Clones are segments of code that are similar according to some definition of similarity”* [11].

Laut dieser Definition kann es verschiedene Begriffe der Ähnlichkeit geben. Diese könnten auf Text, der lexikalischen und syntaktischen Struktur oder auf der Semantik basieren. Sie gelten ebenfalls als ähnlich, wenn sie dasselbe Muster haben [12].

Eine weitere wichtige Definition ist die von E. Juergens u.a. [6], welche die Klone in exakte und inkonsistente Klone gliedert. Zunächst ist es wichtig zu wissen, dass sie ein Code als eine Sequenz von Units bezeichnet, die zum Beispiel Bezeichner, normalisierte Statements oder Zeilen sein können.

Der Grund für die Normalisierung ist, dass beim Vergleich von Codeabschnitten die Kommentare und die Benennung der Literale durch die Normalisierung nicht betrachtet werden, sondern lediglich die Codezeilen verglichen werden können.

2.1.2 Definition Exakte Klone

Ein exakter Klon ist ein fortlaufender Substring eines Codes, der mindestens zweimal in dem (normalisierten) Code erscheint [6]. Dies spiegelt das Copy&Paste Verfahren wieder und ist somit die syntaktische Definition des Klons.

2.1.3 Definition Inkonsistente Klone

Juergens u.a. [6] macht eine detaillierte Definition zu inkonsistenten Klonen. Demnach ist ein Substring s vom Code ein inkonsistenter Klon, wenn es einen anderen Teilstring t des Codes derart gibt, dass ihre Bearbeitungsdistanz unter einem gegebenem Schwellenwert ist und dass t keine signifikante Überlappung mit s hat.

2.1.4 Definition Falsch-Positive Klone

Falsch-Positive Klone sind Codefragmente, welche von einem Klonerkennungstool als ein Klon erkannt wurden, die jedoch keinen Klon darstellen [17]. Durch das Festlegegen eines Schwellenwerts für die Mindestklonlänge kann die Anzahl der Falsch-Positive reduziert werden. Da diese Art der Klone sich lediglich durch eine manuelle Analyse der Klonergebnisse erkennen und beseitigen lassen, ist die optimale Wahl des Schwellenwerts wichtig. Der Grund für die Ausgabe von Falsch-Positiven kann bspw. ähnliche, sich wiederholende Strukturen in der Syntax eines Codeabschnitts sein. Viele Klonerkennungswerkzeuge erkennen bereits Falsch-Positive wie bspw. Array-Listen, die sich in den Literalen-Token und Komma-Token mit unterschiedlichen Werten wiederholen [17]. Weitere Beispiele aus einer Vielzahl von Falsch-Positiven sind Case-Anweisungen, import- und „#include“-Anweisungen, sowie Setter- und Getter-Methoden. Die Codefragmente eines Klonpaars mit ähnlicher Struktur, die sich jedoch in den Bezeichnern stark unterscheiden, werden ebenfalls als Falsch-Positive betrachtet.

2.2 Klonaufbau

Bisher wurden Klone lediglich als kopierte bzw. redundante Codeabschnitte bezeichnet. Für das Verständnis der empirischen Arbeit wird im Folgenden der Aufbau eines Klons definiert. Zunächst wird die Definition des Begriffs „Codefragment“ festgelegt, da dieser Begriff immer in Verbindung mit dem Begriff Klon genannt wird. Anschließend erfolgen weitere Begriffsdefinitionen, die für den Aufbau eines Klons relevant sind. Darüber hinaus wird die Beziehung zwischen den Klonen geschildert.

2.2.1 Codefragmente

Bei der Analyse von Klonen werden Codefragmente miteinander verglichen. Ein Codefragment ist ein Quelltextabschnitt, die zum einen den Namen der Klondatei und zum anderen sowohl die Anfangszeile als auch die Endzeile des geklonten Quelltextabschnitts enthält. Mittels dieser Daten lässt sich ein geklonter Quelltextbereich eindeutig identifizieren. Des Weiteren dienen diese Informationen als Grundlage für die Klonanalyse um bspw. zu prüfen, ob zwei Codefragmente tatsächlich Klone voneinander sind.

2.2.2 Klonpaar

Ein Paar von syntaktisch und strukturell ähnlichen Codefragmenten werden nach Kapser et al. [16] als ein Klonpaar bezeichnet. Einer dieser Codefragmente ist die Kopie des anderen. Das Klonpaar ist der kleinste gemeinsame Nenner für die Beschreibung eines Duplikats.

2.2.3 Klonklasse

Eine Klonklasse enthält mindestens zwei Codefragmente, welche dieselbe bzw. ähnliche Funktionalität beschreiben und an unterschiedlichen Stellen im Quelltext erscheinen. Klonklassen werden ebenfalls als Klongruppen bezeichnet. Jürgens et al. [6] stellen Klongruppen als einen zusammenhängenden Graphen dar. Dabei wird ein Codefragment durch einen Knoten verdeutlicht. Kanten zwischen den Knoten existieren erst dann, wenn eine Klonbeziehung zwischen den Codefragmenten besteht. Wenn alle Klone einer Klongruppe exakte Klone sind, wird von einer exakten Klongruppe gesprochen. Eine Klongruppe welche mindestens ein inkonsistentes Klonpaar enthält, wird als inkonsistente Klongruppe bezeichnet. Die Abbildungen 2.1 und 2.2 stellen eine exakte und inkonsistente Klonkasse bzw. Klongruppe als einen Graphen dar, wobei CF als Abkürzung für Codefragment steht.

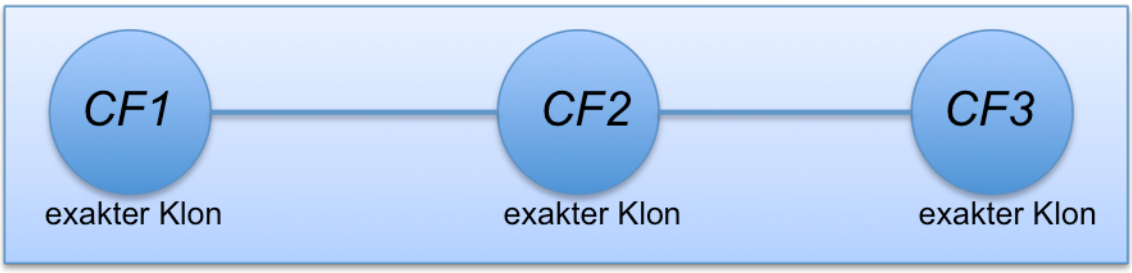

Abbildung 2.1: Exakte Klongruppe

Abbildung 2.2: Inkonsistente Klongruppen

2.3 Klontypen

Kopierte Codeabschnitte werden als Klone bezeichnet. Die Klontypen werden je nach der Ähnlichkeit der Codeabschnitte in vier Klontypen unterschieden. Die Klontypen werden wiederum in textuelle Ähnlichkeit, d.h. ähnlicher Quellcode und funktionale Ähnlichkeit, d.h. lediglich ähnliche Funktionalität ohne textuelle Ähnlichkeit, unterschieden. Die vier Klontypen werden wie folgt definiert:

- **Typ-1-Klone:** Identische Codeabschnitte bis auf Kommentare und Whitespaces.
- **Type-2-Klone:** Syntaktisch identische Codeabschnitte, die sich in den Kommentaren, Literalen, Bezeichnern und im Layout unterscheiden.
- **Type-3-Klone:** Kopierte Codeabschnitte mit weiteren Modifikationen wie Anpassen, Hinzufügen oder Löschen von Codezeilen, Bezeichnern, Kommentaren und Layout.
- **Type-4-Klone:** Codeabschnitte die semantisch ähnlich sind, sich jedoch syntaktisch unterschieden.

Die ersten drei Klontypen lassen sich zur textuellen Ähnlichkeit kategorisieren. Die semantisch ähnlichen Klone gehören zur Kategorie der funktionalen Ähnlichkeit.

2.3.1 Typ-1-Klone

Der Typ-1-Klon ist die exakte Kopie eines Codeabschnitts ohne weitere Modifikationen. In der Kopie handelt es sich um identische Quelltextabschnitte, die sich lediglich in den abstrahierten Kommentaren und Whitespaces (neue Zeilen, Blanks, etc.) unterscheiden (siehe Abbiildung 2.3). Diese Art der Klone kann bspw. durch Inlining von Hand entstehen.

```
int a = 2;
int b = 2 * a;
String name = “Anna”;
```

```
int a = 2; // Kommentar
int b = 2 * a ; // Kommentar
String name = “Anna”;
```

Abbildung 2.3: Typ-1 Klonpaar

2.3.2 Typ-2-Klone

Bei Typ-2-Klonen handelt es sich um syntaktisch identische Kopien der Codefragmente. Die Typ-2-Klone umfassen Typ-1-Klone. Bei dieser Art des Klonens werden neben dem Abstrahieren der Whitespaces und Kommentare auch die Bezeichner bzw. Literale umbenannt. Typ-2-Klone entstehen in der Regel durch die Wiederverwendung einer Funktion, bei der die Bezeichner konsistent auf den Quelltext angepasst werden. Dies ist bspw. bei generischen Funktionen der Fall, bei dem die Anpassung von Hand durchgeführt wird.

Baker, S. [12] macht eine weitere Gliederung der Klone in sog. parametrisierte Klone, die eine Untermenge der Typ-2-Klone sind. Formal bedeutet dies, dass es eine bijektive Abbildung zwischen den zwei Codefragmenten gibt, die jedem Bezeichner eines Codefragments einen Bezeichner des anderen Codefragments zuordnet und umgekehrt.

Werden die Bezeichner im kopierten Quelltext für alle Bezeichner konsistent umbenannt, so spricht man von einem konsistent umbenannten Typ-2-Klon, welches vor allem bedeutet, dass die Codefragmente sich in der Semantik nicht unterscheiden. Wird die Umbenennung im Gegensatz hierzu nicht konsistent durchgeführt, so spricht man von Typ-2-Klonen mit inkonsistenter Umbenennung. Der Grund hierfür können Fehler beim Umbenennungsvorgang sein. In diesem Fall besteht die Gefahr, dass sich die Klone unbeabsichtigt in der Semantik unterscheiden. In Abbildung 2.4 sind Klone mit konsistenter und inkonsistenter Umbenennung dargestellt.

Ursprünglicher Quellcode:

```
int a, b;
for (a=0, b=50;
      a <= b;
     a=a+3; b=b-4)
{….}
```

Konsistente Umbenennung:

```
int c, d;
for (c=0, d=50;
      c <= d;
     c=c+x; d=d-4)
{….}
```

Inkonsistente Umbenennung:

```
int c, d;
for (c=0, d=50;
      c <= c;
     c=d+3; d=d-4)
{….}
```

Abbildung 2.4: Typ-2 Klonpaare mit konsistenter und inkonsistenter Umbenennung

2.3.3 Typ-3-Klone

Die Typ-3-Klone, auch inkonsistente Klone genannt, umfassen Typ-1 und Typ-2-Klone. Hierbei handelt es sich um einen der beiden Klone, der z.B. durch Anpassen, Hinzufügen oder Löschen von Codezeilen modifiziert wurde. Typ-3-Klone werden auch inkonsistente Klone, bzw. Gaps [13], ungleiche Abschnitte, genannt. Dieser Typ des Klons entsteht, wenn eine bestehende Funktionalität im Softwarelebenszyklus weitere Funktionalitäten in der Kopie benötigt und dementsprechend angepasst wird [13], [14]. Daraus kristallisiert sich, dass der Typ-3-Klon sich von den vorherigen beiden Klontypen unterscheidet.

Der Typ-3-Klon ist also ursprünglich ein Typ-1 oder Typ-2-Klon, welche durch die Modifizierung unterbrochen wird und daher sich von dem ursprünglichen Codefragment nicht nur in den Bezeichnern, Literalen und Kommentaren unterscheidet, sondern auch unähnliche Anweisungsteile enthält, wie in Abbildung 2.5 ersichtlich wird.

Wichtig für den Typ-3-Klon ist die minimale Klonlänge, also der Schwellenwert für die Ähnlichkeit der Klonabschnitte sowie das Gap Ratio, welches bestimmt um wie viele Codezeilen sich ein Klonpaar unterscheiden darf [15]. Lediglich ein optimales Verhältnis der festgelegten Werte liefert ein optimales Ergebnis der erkannten Typ-3-Klone. Andernfalls erfolgt eine ungünstige Verteilung von Klonabschnitten, welche sich über mehrere Klonklassen verteilen oder es werden die Klone nicht erkannt,sondern nur Teilabschnitte [15]. Ein weiterer Nachteil der ungünstig ausgewählten Schwellenwerte spiegelt sich in der Anzahl der Falsch-Positiven, welche sich erheblich erhöhen. Dies führt bei der Klonerkennung zu einer relativ schlechten Genauigkeit der Klonergebnisse.

```
int a, b;
for (a=0, b=50;
      a <= b;
     a=a+3; b=b-4)
{….}
```

```
int a, b;
for (a=0, b=50;
      a = a+1;
      a <= b;
     a=a+3; b=b-4)
{….}
```

Abbildung 2.5: Typ-3 Klonpaar

2.3.4 Typ-4-Klone

Bei den Typ-4-Klonen handelt es sich um semantisch ähnliche Klone, die zwar dieselbe Funktionalität darstellen, jedoch syntaktisch unterschiedlich sind und somit nicht ohne weiteres als Klon erkannt werden. Beispielsweise sind die for-Schleifen, welche ebenfalls als while-Schleifen dargestellt werden können, ein Klon des Typs 4. Das Inkrement Operator „++“ in einigen Programmiersprachen kann durch die ausgeschriebene Schreibweise ersetzt werden und ist ebenfalls ein Beispiel für ein Typ-4-Klon. Die Bestimmung der semantischen Ähnlichkeit bei syntaktischer Verschiedenheit ist fast unmöglich. Daher wird dieser Klontyp in der Literatur eher selten erwähnt und hat wenig praktischen Nutzen. Die Abbildung 2.6 stellt zwei syntaktisch verschiedene jedoch semantisch gleiche Codefragmente dar.

```
int a (int betrag, int niedrig) {
if (niedrig {
      return (betrag *120)/100;
} else {
      return (betrag *150)/100;
}
}
```

```
int b (int betrag, int niedrig) {
      int prozent;

      if (niedrig) {
         prozent = 20;
      } else {
         prozent = 50;
}
return betrag +
      (prozent * betrag) /100;
}
```

Abbildung 2.6: Typ-4 Klonpaar

2.3.5 Zusammenhang der Klontypen

Die verschiedenen Klontypen stehen in der Mengenbeziehung im Zusammenhang. Das Mengenverhältnis zwischen den Klontypen lässt sich formal wie in Abbildung 2.7 dargestellt folgendermaßen erfassen:

$$\textbf{Typ1} \subset \textbf{Typ2} \subset \textbf{Typ3} \subset \textbf{Typ4}$$

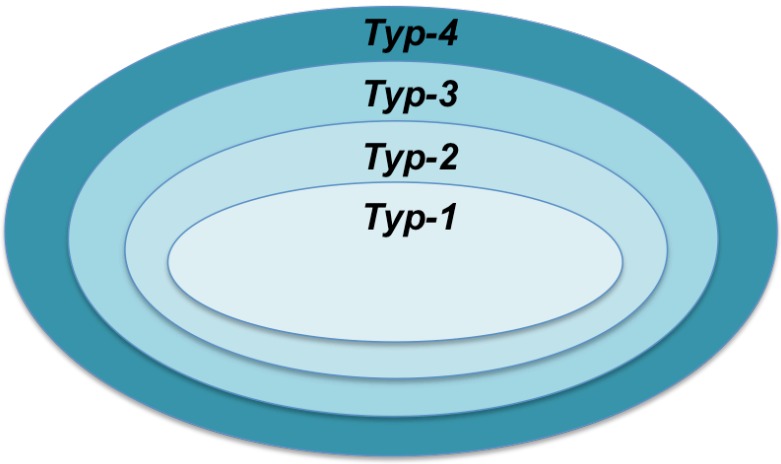

Abbildung 2.7: Mengenbeziehung der Klontypen

Mit steigendem Klontyp wächst die Anzahl der möglichen Codeklone. Der Grund hierfür ist, dass je höher der Klontyp ist, desto mehr Freiraum ist für die Abweichungen zwischen den geklonten Codeabschnitten vorhanden.

In Abbildung 2.7 sind die Mengen der Klontypen dargestellt. Die Menge der Typ-2-Klone umfasst die Menge der Typ-1-Klone, welche die Identitätsabbildung eines geklonten Codeabschnitts ist. Bei den Typ-2-Klonen existieren im Vergleich zu Typ-1-Klonen Bezeichner, die konsistent bzw. inkonsistent umbenannt wurden.

Die Identitätsabbildung wird durch den Typ-3-Klon erweitert, in dem die Unterschiede bspw. nicht nur in den Bezeichnern und im Layout vorkommen, sondern sich komplette Bereiche unterscheiden. Bei den Typ-4-Klonen kann, auf Grund fehlender syntaktischer Ähnlichkeit, keine Zuordnung von geklonten Codeabschnitten vorgenommen werden.

2.4 Klonerkennungsprozess inkonsistenter Klone

Es gibt verschiedene Ansätze zur Klonerkennung die auf unterschiedlichen Klonerkennungswerkzeugen basieren. Gemeinsames Ziel der Klonerkennungswerkzeuge ist es, die Codefragmente miteinander zu vergleichen und mit möglichst wenig Aufwand möglichst viele Klone zu erkennen. Um den Aufwand für die Klonsuche zu reduzieren, existieren Klonerkennungsprozesse, welche in Unterstützung der Klonwerkzeuge umgesetzt werden.

Roy u.a. [18] schaffen einen Gesamtüberblick über die wichtigsten Schritte in einem Klonerkennungsprozess. Sie stellen einen generischen Klonerkennungsprozess vor, die eine Menge von Schritten enthält, welche in der Regel von einem Klonerkennungstool bearbeitet werden. Hierbei befassen sie sich über die Token-Ebene hinaus mit weiteren möglichen Klonerkennungsansätzen und vergleichen diese miteinander. Der vorgestellte generische Klonerkennungsprozess ermöglicht das Vergleichen und Bewerten von Klonerkennungswerkzeugen in Anlehnung auf ihren zugrundeliegenden Mechanismus für die einzelnen Klonerkennungsschritte und ihrer Höhe der Unterstützung für diese Schritte.

Der Fokus dieser Diplomarbeit liegt auf inkonsistente Klone. Daher befasst sich dieses Kapitel mit einem Klonerkennungsprozess, welcher speziell für die Erkennung von inkonsistenten Klonen entworfen wurde, der auf Token-Ebene arbeitet und im Allgemeinen ausreichend für die Suche nach Copy&Paste Codefragmenten ist und zugleich effizient ist [6]. Für diesen Klonerkennungsprozess entworfene Algorithmen und Filter wurden als Teil von CloneDetective [19] implementiert, die auf ConQat basieren (hierzu mehr in Kapitel 4.1) und in der Lage sind inkonsistente Klone zu erkennen.

Die Basisschritte für die Erkennung von inkonsistenten Klonen des vorgestellten Verfahrens beruhen ebenfalls auf den Schritten des allgemeinen Klonerkennungsprozesses [18]. Der Unterschied liegt darin, dass das Klonerkennungstool als eine Pipeline organisiert ist. Die Abbildung 2.8 schildert die einzelnen Schritte für die Klonerkennung.

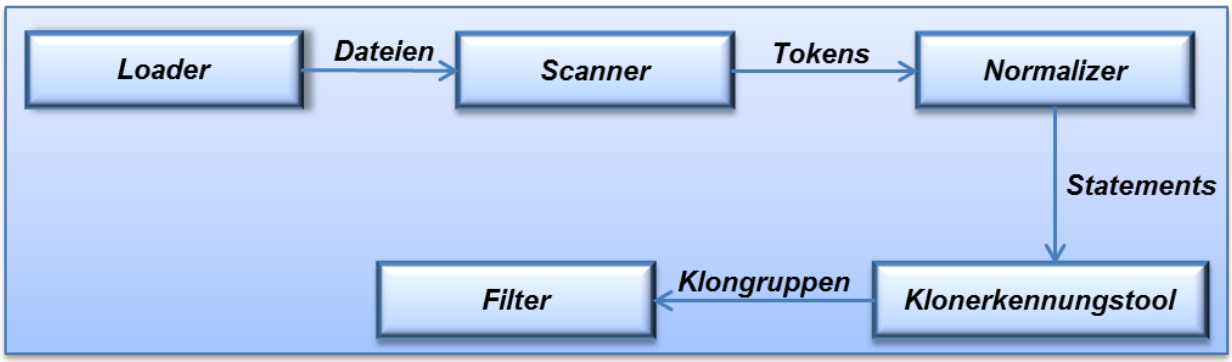

Abbildung 2.8: Klonerkennungsprozess [6]

2.4.1 Vorverarbeitung

Vor der Vorverarbeitung werden zunächst der zu analysierende Code, bzw. die Daten durch den Loader (also aus dem Speicher) ausgelesen. Anschließend muss der Quellcode unterteilt und die Quelle und die Vergleichseinheiten bestimmt werden [18]. Im Klonerkennungsprozess, welcher für inkonsistente Klone entworfen wurde, basiert die Vergleichseinheit auf Tokens. Daher werden die Daten nach dem Auslesen mit Hilfe des Scanners in Tokens aufgeteilt [6].

Der nächste Schritt ist das Entfernen von uninteressanten Teilen des Quellcodes. Diese setzten sich aus generiertem Code, die Falsch-Positive verursachen können, Kommentaren und eingebettetem Code aus anderen Programmiersprachen zusammen. Letztlich setzt sich der Code aus aufgeteilten Tokens zusammen, welche durch die Tokenisierung erfolgt.

2.4.1.1 Normalisierung

Die Normalisierung ist ein wichtiger Bestandteil der Vorverarbeitungsphase. Die Normalisierung stellt aus den Tokens, die aus einmaligen Schlüsselwörtern, Bezeichnern und Operatoren bestehen, wieder Statements zusammen. Dieser Schritt ermöglicht eine bessere Anpassung der Normalisierung und hilft Klone zu vermeiden, die innerhalb von Statements beginnen und enden.

Die Normalisierung beseitigt die Unterschiede in der Benennung der Bezeichner, konstanten Werten und Literalen, so dass sie beim Vergleich von Statements nicht relevant sind. In Abbildung 2.9 ist ein Beispiel für die Normalisierung eines Codeabschnitts. In dem nicht normalisierten Codeabschnitt erfolgt das Umändern einer UTF-8-Datei in eine UTF-16-Datei. Es ist zu erkennen, dass die Bezeichner zu „id“ und die Literale zu „lit“ normalisiert wurden, so dass der Klonerkennungsalgorithmus diese als einen Klon erkennen kann.

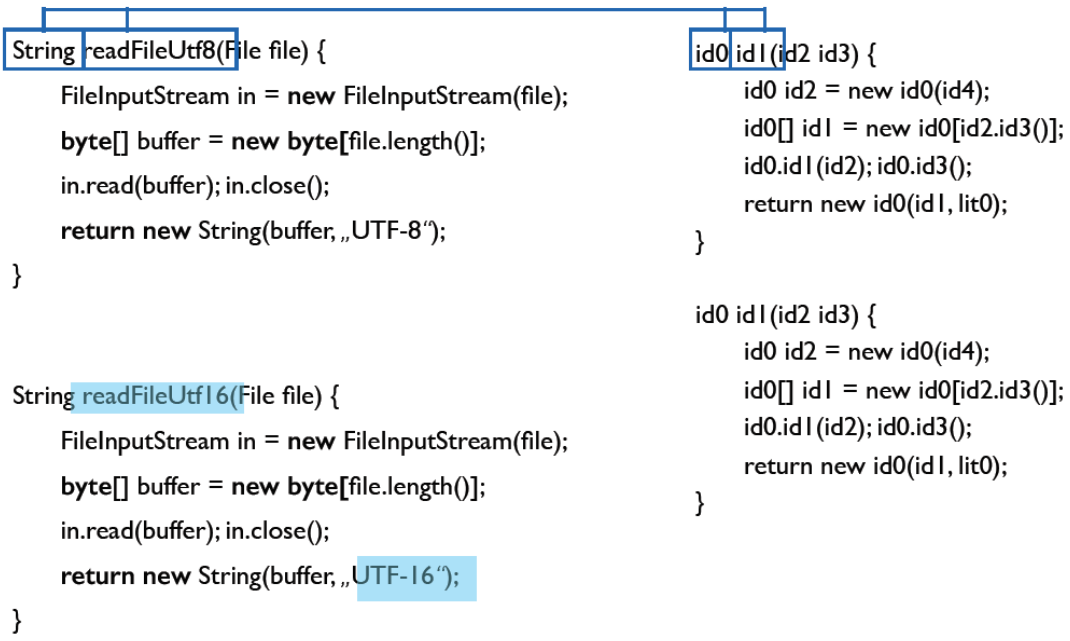

Abbildung 2.9: Normalisierungsbeispiel [20]

2.4.2 Erkennungsalgorithmus

Als nächstes wird die aus den Statements gebildete Sequenz an den Klonerkennungsalgorithmus übergeben. Der speziell für diesen Klonerkennungsprozess entwickelte Algorithmus konstruiert aus dem Quelltext ein Suffix-Baum und führt anschließend, basierend auf einer Bearbeitungsdistanz, für jedes mögliche Suffix eine Klonsuche durch [6]. Anders ausgedrückt sollen für die Klonsuche gemeinsame Teilstrings in der Sequenz gefunden werden, die von allen Datenströmen gebildet wurden, in denen die Teilstrings nicht exakt identisch sein müssen, aber eine durch einen Schwellenwert begrenzte Bearbeitungsdistanz erfüllen.

Als Ergebnis liefert der Erkennungsalgorithmus Klongruppen, welche aus Klonen bestehen, die aus den Sequenzen (normalisierten Tokens) ermittelt wurden.

2.4.3 Nachverarbeitung und Filterung

Zuletzt müssen die Klongruppen überarbeitet bzw. gefiltert werden. Um Speicherplatz zu sparen erfolgt die Filterung relativ früh. Das Ziel ist es, den Speicher nicht für nicht relevante Klongruppen zu verwenden. Das Eliminieren von Klongruppen, deren Klone sich nicht überlappen und Gruppen, deren Klone in anderen Klongruppen enthalten sind, ist ein wesentlicher Bestandteil des Filterns.

Des Weiteren bietet das Filtern neben einer absoluten Grenze für die Anzahl der Inkonsistenten auch eine relative Grenze an. Das ermöglicht das Ausfiltern von Klonen,

in denen die Anzahl der inkonsistenten Klone im Verhältnis zur Länge des Klons einen bestimmten Betrag übersteigt [6].

Der Nachverarbeitungsschritt bietet die Gelegenheit, Falsch-Positive durch eine manuelle Analyse oder automatisierte Heuristiken auszufiltern [18]. Die Ergebnisse des vorgestellten inkonsistenten Klonerkennungsvorgangs werden ebenfalls über eine HTML-Seite visuell in einem Tree Map (mehr dazu in Kapitel 4.1) dargestellt, so dass das manuelle Filtern von Falsch-Positiven beschleunigt werden kann.

2.5 Ansätze zur Klonerkennung

Nach dem in die Begrifflichkeiten und Definitionen der Klone eingeführt wurden, stellt dieses Kapitel Techniken und Verfahren zur Erkennung von Klonen in Codefragmenten vor. Die Ansätze zur Klonerkennung unterscheiden sich zunächst in der Granularität des berücksichtigten Wissens und der Abstraktionsebene der Analyse [21]. Nach deren Erklärungen werden die unterschiedlichen Klonerkennungsverfahren vorgestellt.

2.5.1 Granularität

Unter Granularität versteht man die „Größe“ der Einheiten, die bei der Klonerkennung verglichen werden müssen. Die Granularität bestimmt die Anzahl der potentiell notwendigen Vergleiche im Klonerkennungsvorgang.

Zur Klonerkennung werden üblicherweise folgende Granularitätsstufen verwendet [21]:

- Zeichen
- Zeilen
- Anweisungssequenzen
- Funktionen
- Module

Für ein exaktes Verfahren wird die Granularität verfeinert, in dem kleinere Codefragmente gebildet werden und somit die Anzahl der zu vergleichenden Einheiten steigt. Jeder dieser Einheiten der Granularitätsstufe wird mit jeder anderen Einheit verglichen. Somit steigt also die Zahl der potentiellen Vergleiche und damit verbunden auch der Aufwand. Die steigende Granularität eines Verfahrens führt zu einem quadratischen Aufwand der Vergleiche.

Werden zwei fünfstellige Codefragmente durch ein zeilenorientiertes Verfahren miteinander verglichen, so sind 25 Vergleiche durchzuführen, da jede Zeile des einen Codefragments mit jeder Zeile des anderen Codefragments verglichen wird. Im Vergleich dazu würden tokenorientierte oder zeichenorientierte Verfahren mit feinerer Granularität in der Regel deutlich höhere Vergleiche durchführen.

2.5.2 Abstraktionsebene

Unter Abstraktionsebene versteht man die „Art“ der Einheiten, die zu vergleichen sind. Demzufolge gibt die Abstraktionsebene die Datenbasis an, auf der die Klonerkennung stattfindet.

Zur Klonerkennung werden üblicherweise folgende Abstraktionsebenen unterschieden [21]:

- Text
- Token
- Syntax
- Semantik
- Metriken

Damit ein Codefragment als Klon erkannt werden kann, muss sie zunächst auf die geeignete Art, bspw. durch Normalisierung, abstrahiert werden. Mit zunehmender Abstrahierung können präzisere Aussagen getroffen werden, da die Programmiersprachenunabhängigkeit sinkt. Gleichzeitig steigt jedoch mit zunehmender Abstraktionsebene die Laufzeit.

Damit beispielsweise ein Verfahren Typ-2-Klone erkennen kann, setzt ein Klonerkennungsverfahren mindestens auf der Tokenebene an, da auf dieser Ebene die Bezeichner in einem Quelltext von dem Rest des Quelltextes erkannt bzw. unterschieden werden können.

2.5.3 Textbasierte Klonerkennung

Die textbasierte Klonerkennung ist die bisher einfachste Methode zur Klonerkennung. Textbasierte Ansätze beruhen auf dem Vergleichen von Zeilen bzw. Anweisungssequenzen. Wenn mehrere gleiche aufeinanderfolgende Zeilen als ähnlich erkannt werden, gibt das Klonerkennungstool Klonpaare mit ihrer maximal möglichen Länge zurück. Der zu vergleichende Quelltext wird üblicherweise nicht aufbereitet bzw. nur gering aufbereitet. Beispielsweise werden Whitespaces und Kommentare entfernt. Dieses Verfahren hat den Nachteil, dass sie nicht robust zu Veränderungen, zum Beispiel zu strukturellen Änderungen an den Anfangs- und Endzeilen eines Codefragments, führt. Der Vorteil hingegen besteht darin, dass auf Grund fehlender Aufbereitung, bzw. Transformation oder Normalisierung, keine sprachspezifischen Eigenschaften verwendet werden und das Verfahren infolgedessen sprachunabhängig ist.

Es gibt Studien mit verschiedenen Ansätzen für die Umsetzung des textbasierten Verfahrens, die sich lediglich im Algorithmus zum Vergleich der Codezeilen

unterscheiden. Johnson, J.H. [22] macht bspw. einen Zeichenkettenvergleich mittels Hashwerten. Ducasse S. u.a. [23] führen dagegen einen zeilenweisen Vergleich über Dot-Plots durch. Die Abbildung 2.10 zeigt ein Beispiel für einen Vergleich über Dot-Pots.

Ein bekanntes Klonerkennungstool für das textbasierte Verfahren ist GNU diff, dessen Ziel das Ermitteln von Unterschieden zwischen zwei Dateien ist. Dieses Verfahren ermittelt lediglich die Unterschiede bzw. die Gleichheit zweier Dateien.

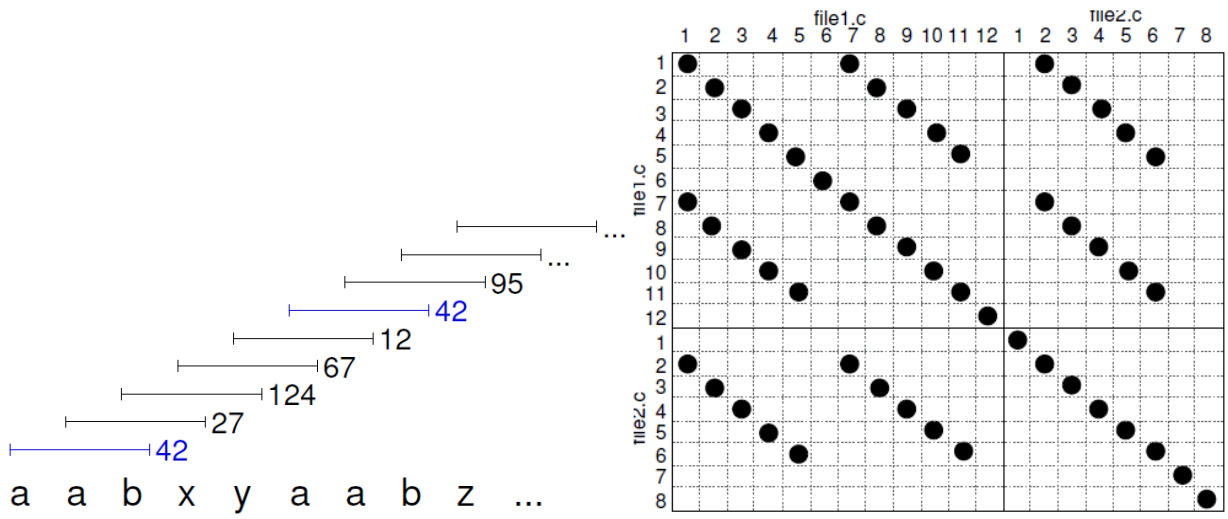

Abbildung 2.10: Zeilenweiser Vergleich über Dot-Plots [24]

2.5.4 Tokenbasierte Klonerkennung

Im Gegensatz zur textbasierten Klonerkennung ist das tokenbasierte Verfahren fortgeschrittener und komplexer. Der Grund hierfür sind die in der ersten Phase der Klonerkennung verwendeten komplizierten Transformationsalgorithmen, um eine Zeichenfolge (auch *tokenstream* genannt) aus dem Quelltext zu konstruieren. Dies erfolgt durch die Anwendung einer lexikalischen Analyse (auch *tokenization* genannt) auf den Quelltext. Dieser Vorgang hat zur Folge, dass die tokenbasierten Techniken sprachabhängig werden. Der wichtige Vorteil dieser Technik ist, dass sowohl exakte als auch ähnliche Klone gefunden werden können, da eben durch die Transformation ähnliche Konstrukte angeglichen werden.

Auch für dieses Verfahren gibt es einige Studien mit verschiedenen Ansätzen, die sich ebenfalls in der zweiten Phase des Klonerkennungsprozesses, nämlich im Algorithmus der Klonerkennung durch Auswertung der Zeichenfolgen unterscheiden.

Das bekannteste Verfahren der tokenbasierten Klonerkennung ist die von Baker, B. [14,12]. Es handelt sich um ein zeilenorientiertes Verfahren, das Klone auf der Zeichenfolge (tokenstrom) erkennt. Dieses Verfahren kommt in ihrem Tool dup zum Einsatz, welches ein tokenbasiertes Pattern-Matching verwendet.

Bei diesem Verfahren wird zunächst für jede Codezeile, durch ein Lexem für die zu analysierende Sprache ein Parameter-String erzeugt (auch P-String genannt).

Ein Parameter String besteht aus Nichtparametersymbolen und Parametersymbolen. Während ein eindeutiges Nichtparametersymbol (sog. Funktor) die Struktur einer Zeile enthält, besteht dagegen ein Parametersymbol aus den Variablen, die in der jeweiligen Codezeile verwendet wurden. Der Funktor repräsentiert die Struktur einer Codezeile eindeutig, so dass Codezeilen mit identischer Struktur auf den gleichen Funktor abgebildet werden. Das folgende Beispiel von Koschke [24] bildet eine Codezeile auf ein Parameter-String ab.

Die

Codezeile: x = x + y

wird abgebildet auf den

Parameter- String: (P = P + P; x,x,y)

der dargestellt wird mit dem

Funktor: αxxy

Die Codezeile wurde in den dazugehörigen Funktor α und dessen Parameterliste umgewandelt. Demnach würden alle Zeilen, welche die Form P = P + P haben, auf den Funktor α abgebildet werden. Die erzeugten und konkatenierten P-Strings aller Codezeilen, die den Programmcode repräsentieren, werden anschließend in einen P-Suffix-Baum übertragen. Der quadratische Aufwand für die Vergleiche wird durch die Verwendung des P-Suffix-Baums vermieden. Aus dem P-Suffix-Baum können sowohl die Position als auch die Länge und Anzahl der Klone direkt ermittelt werden.

Vor dem Aufbau des Suffix-Baums wird für jeden P-String die prev-Funktion codiert, um von den Bezeichnern zu abstrahieren. Hierbei werden den Bezeichnern je nach ihrem Vorkommen Werte in der Parameterliste vergeben. Wenn ein Bezeichner das erste Mal in der Parameterliste vorkommt, erhält sie die Zahl 0. Für jedes weitere Vorkommen erhält sie die Zahl des relativen Abstands zum vorherigen Vorkommen.

Der P-Suffix-Baum wird durch den P-String und der prev-Funktion zu den Suffixen erstellt. Der P-String eines Suffixes sowie die prev-Werte zu den Suffixen werden auf eine Kante eines Suffix-Baumes eingetragen, welche mit der Eingabeendzeichen $ enden.

Zunächst wird der P-String des Suffixes auf eine Kante in ein Suffixbaum übertragen, welches mit dem Eingabeendzeichen $ endet (siehe Abbildung 2.11). Die komplette Suffixeingabe wird abgearbeitet, indem der Funktor, beginnend mit dem ersten, und die dazugehörige Parameterliste des P-Strings entfernt werden. Das Ergebnis wird anschließend erneut in den P-Suffixbaum eingetragen, bis die Eingabe abgearbeitet ist. Nun können alle Klone ausgehend von der Wurzel über die Kanten gefunden werden.

Das folgende Beispiel von Koschke und Simon [21] schildert eine übersichtliche Darstellung, die zu einem Suffix S den P-Suffix-Baum mit allen prev(S_i) graphisch darstellt, wobei $S_i = s_i s_{i+1} ... s_n\$$ das i´te Suffix von S ist.

Suffix S = αyβyαxαx

➔ prev(S_1) = α0β2α0α2$
➔ prev(S_2) = 0β2α0α2$
➔ prev(S_3) = β0α0α2$
➔ prev(S_4) = 0α0α2$
➔ prev(S_5) = α0α2$
➔ prev(S_6) = 0α2$
➔ prev(S_7) = α0$
➔ prev(S_8) = 0$
➔ prev(S_9) = $

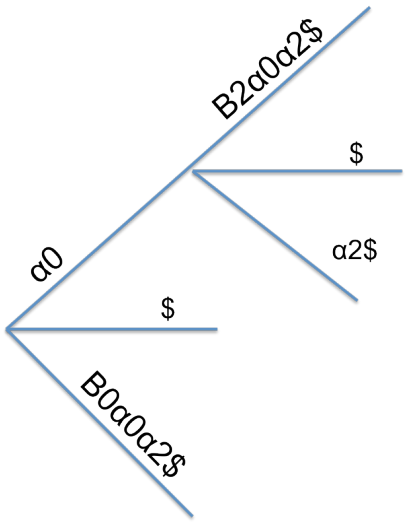

Abbildung 2.11: P-Suffix-Baum zum Suffix S

Für eine schnelle Klonerkennung eignet sich dieses Verfahren ausgesprochen gut und liefert ansprechende Ergebnisse [25]. Das Verfahren ermöglicht das Finden von konsistent umbenannten Typ-1 und Typ-2-Klonen. Typ-3-Klone werden durch einen separaten Schritt am Ende erkannt [26]. Hierzu werden mehrere gleiche Typ-1 und Typ-2-Klone die hintereinander vorkommen zu einem Typ-3-Klon zusammengefasst. Dieses Verfahren hat den Vorteil, dass es durch die niedrige Abstraktionsebene programmiersprachenunabhängig ist. Der Nachteil hingegen ist, dass durch die Bearbeitung auf Zeilenbasis, kleine Umformatierungen zu falschen Ergebnissen führen.

2.5.5 Abstrakter Syntaxbaum

Dieser Ansatz der Klonerkennung basiert auf abstrakten Syntaxbäumen (AST). Ein bekanntes Tool für diesen Ansatz der Klonerkennung ist das von Ira D. Baxter [27] entwickelte Programm CloneDR™.

Bei diesem Verfahren wird zunächst zu jedem Quellcode ein AST erstellt. Alle Teilbäume werden gegeneinander abgeglichen und auf Gleichheit und Ähnlichkeit geprüft (siehe Abbildung 2.12). Mit der Absicht den quadratischen Aufwand zu vermeiden, werden die zu vergleichenden Bäume partitioniert. Die Partitionierung der Bäume erfolgt durch Hash-Funktionen [27]. Nun werden lediglich Teilbäume innerhalb einer gemeinsamen Partition verglichen. Teilbäume, die denselben Hashwert haben, deuten möglicherweise auf einen Klon [24]. Das Prüfen der Teilbäume auf Gleichheit und Ähnlichkeit findet mit Hilfe einer Ähnlichkeitsfunktion statt. Die Partitionierung und der Vergleich der AST ignoriert Bezeichner, sodass die Erkennung der Typ-2-Klone realisierbar ist. Um die maximale Anzahl der Klone zu finden, werden in einem Nachbearbeitungsschritt Klone, die aus mehreren Anweisungen bestehen gesucht und zusammengefasst. Resultierend aus diesem separaten Schritt am Ende können Typ-3-Klone erkannt werden.

Die Berücksichtigung der kommutativen Operatoren gehört zu den wichtigsten positiven Merkmalen dieses Verfahrens. Ein weiterer Vorteil ist, dass ganze Anweisungen also syntaktische Einheiten verglichen werden können. Das Verfahren liefert als Ergebnis syntaktisch vollständige Klone. Der Nachteil hingegen ist, dass das Verfahren auf Grund des syntaxbasierten AST-Matchings relativ aufwändig ist. Für das Erstellen der AST wird ein Parser für jede Programmiersprache benötigt, der dazu führt, dass dieses Verfahren der Klonerkennung weniger programmiersprachenunabhängig ist.

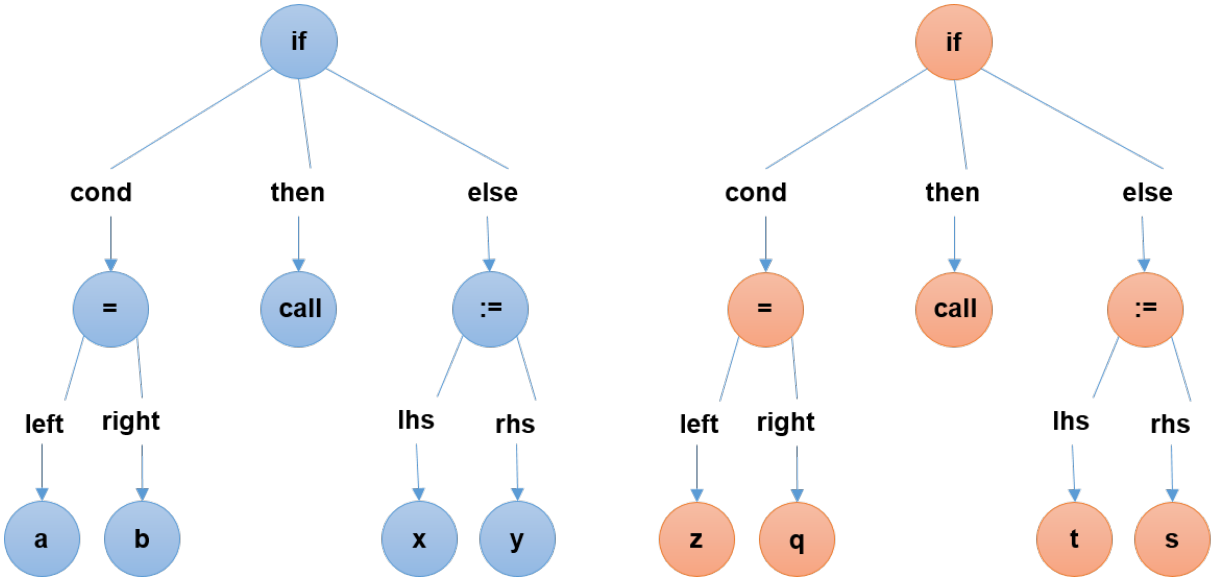

Abbildung 2.12: Abstrakter Syntaxbaum [24]

2.5.6 Graphbasierte Klonerkennung

Das graphbasierte Klonerkennungsverfahren auch Programm Dependence Graph (PDG) genannt, kann als eine Erweiterung der baumbasierten Technik betrachtet werden. Der Unterschied dieses Verfahrens liegt darin, dass durch den PDG die syntaktische Struktur des Quellcodes und der Datenfluss dargestellt werden können, die wiederum bei der Klonerkennung verwendet werden.

Krinke, J. [28] hat für diesen Ansatz der Klonerkennung einen Tool namens duplix entwickelt, der mit Hilfe eines PDGs sowohl Informationen aus dem AST als auch über die Abhängigkeiten des Programms berücksichtigt. Die Klonerkennung erfolgt nun durch die Suche nach ähnlichen Teilgraphen, welche die Klone darstellen.

Eine besondere Eigenschaft dieser Technik ist, dass sie, durch die Verwendung des PDGs, ein gutes Ergebnis für die Erkennung der falschen Codeduplikate liefert. Ein weiterer positiver Aspekt besteht darin, dass in der Anzahl der nicht identifizierten Codeduplikate ein guter Wert erzielt werden kann.

Der Nachteil hingegen liegt im zeitintensiven Aufwand für das Erzielen der Analysegrundlage. Die Suche nach Klonen ist ebenfalls aufwändiger im Vergleich zu anderen Verfahren. Des Weiteren besteht der Nachteil darin, dass das Verfahren nicht programmiersprachenunabhängig ist. Dies wird bei der Analyse der zu erzeugenden ASTs und Ermittlung der Programmabhängigkeiten erkannt. Die mangelnde Programmiersprachenunabhängigkeit erschwert die Klonerkennung für verschiedene Programmiersprachen.

2.5.7 Metrikbasierte Klonerkennung

Die Umsetzung des metrikbasierten Klonerkennungsverfahrens findet sich in der Technik von Mayrand u.a. [29] wieder. Die bisher vorgestellten Verfahren leiten die Informationen aus dem Quellcode oder Strukturen ab, welche sowohl die Syntax als auch die Semantik des Quellcodes enthalten. Das Verfahren von Mayrand geht von bereits abstrahierten Informationen aus. In diesem Verfahren werden verschiedene Metriken für verschiedene Codefragmente erhoben. Anschließend wird aus den Metriken berechnet, ob zwei Codefragmente gleich oder ähnlich sind. Hierzu gibt eine Vergleichsfunktion eine Vorgabe, für welche Ausprägungen der gewählten Merkmale die zu vergleichenden Codefragmente als gleich oder ähnlich zu bewerten sind. In diesem Fall lassen sich gleiche Codefragmente in Typ-1-Klone und ähnliche Codefragmente in Typ-2-Klone kategorisieren.

Die Granularität von Mayrand [29], also die erkannten Klontypen, kategorisieren sich in gleich, ähnlich, verschieden und sind von den jeweils gewählten Vergleichsfunktionen abhängig. Die Abstraktionsebene wird ebenfalls durch die, von der Vergleichsfunktion verwendeten Metrik bestimmt.

Mayrand gliedert in folgende Vergleichsmetriken:

- Name
- Layout
- Anweisungen
- Kontrollfluss

Diese Vergleichsmetriken werden zum einen über den Quelltext, zum anderen über den AST als auch über den Kontrollflussgraphen erhoben.

Da dieser Ansatz auf der Annahme beruht, dass sowohl bei einem gleichen Quellcode eine Metrik ebenfalls den gleichen Wert liefert, als auch bei ähnlichem Quellcode eine Metrik einen ähnlichen Wert liefert, kann diese Annahme ohne weiteres akzeptiert werden. Der Nachteil liegt jedoch darin, dass bei der Klonerkennung nicht erkannt werden kann, ob zwei Quellcodes gleich oder ähnlich sind. Daraus resultiert die Unklarheit in der Beurteilung, ob gleiche oder ähnliche Kennzahlen auch dementsprechend auf gleiche oder ähnliche Quellcodes hindeuten.

3 Forschungsstand

Redundante Codestellen bzw. Code-Klone werden in einem Softwaresystem nicht ausgeschlossen. Klone in Systemen sind jedoch unterschiedlichen Gründen zurückzuführen. In einigen Klonfällen ist das Klonen sogar berechtigt. Klone in Softwaresystemen sind jedoch mit positiven sowie negativen Auswirkungen und Problemen verbunden. Es wurden zahlreiche Studien der Erkennung von Klonen speziell für inkonsistente Klone gewidmet, um der Ursache der Klone auf die Spur zu kommen. Für einen besseren Rückschluss der Auswirkungen der Klone, wurden ebenfalls zahlreiche Studien entwickelt, um die Fehler in Klonen zu analysieren und den Zusammenhang derer mit den Klonen zu ermitteln. Dieses Kapitel befasst sich mit allen diesen Themen im Detail und schafft einen Gesamtüberblick über Wissenschaftlichen Arbeiten, die dieses Thema behandeln.

3.1 Gründe des Klonens

Das Copy&Paste-Verfahren eines Entwicklers erzeugt bereits Klone. Es gibt verschiedene Gründe für das Entstehen eines Softwareklons. Diese werden nach Koschke [39] wie folgt kategorisiert:

Entwicklungsstrategie

Durch das Copy-Paste-Verfahren werden bestehende Funktionalitäten in den Codefragmenten für eine neue Funktionalität dupliziert. Das bedeutet also, wenn eine bereits bestehende Funktionalität in gleicher oder ähnlicher Form im Software-Projekt an einer anderen Stelle erforderlich ist, wird diese an die entsprechende Stelle kopiert und wenn nötig verändert und modifiziert. Der Klon fungiert also als Template für neue Funktionalitäten.

Wartungsvorteile

Ein weiterer Vorteil besteht darin, dass bei der Wiederverwendung von bestehenden Funktionalitäten die Wartung erleichtert wird, da es ich bei dem geklontem Codefragment um einen bewährten Quellcode handelt. Des Weiteren reduziert sich durch das Kopieren eines bewährten Quellcodes die Fehlerwahrscheinlichkeit im Quelltext. Ein besonders wichtiger Punkt für die Entwicklung ist die Unabhängigkeit zwischen den Projektdateien bzw. Komponenten, die durch das Kopieren erzielt werden kann. Die unabhängige Wartung der Komponenten ist daraus resultierend möglich.

Überwindung von Einschränkungen

Eine weitere Ursache für das Entstehen von Klonen liegt in den eingeschränkten und verschiedenen Abstraktionsmechanismen einiger Programmiersprachen. Kim et al. [48] hat bspw. in einer Studie Programmierer während der Entwicklung beobachtet. Die Entwickler mussten in vielen Fällen klonen, weil die verwendete Programmiersprache nicht aussagekräftig war. Wenn also für ein bestimmtes Problem keine angemessene Möglichkeit zur Abstraktion besteht, wird dieses durch die Verwendung von Klonen behoben. Weitere Einschränkungen im Entwicklungszyklus, die das Klonen von Entwicklern begründen, sind mangelnde Entwicklungswerkzeuge, unzureichende Kenntnisse und damit verbunden auch fehlendes Problembewusstsein des Entwicklers sowie Zeitdruck.

Die Verwendung von firmeninternen sowie programmiersprachenabhängigen Bibliotheken ist eine unvermeidbare Ursache für Klone, die nicht notwendigerweise einen negativen Einfluss haben, da lediglich ein erforderliches Protokoll implementiert wird.

3.2 Folgen des Klonens

Laut Studien enthalten 5-25% der Softwaresysteme redundante Codestellen [21, 40, 41], welche sich negativ im Entwicklungszyklus des Software-Projekts auswirken. Durch das Klonen vergrößert sich nämlich der Codeumfang, sodass der Aufwand zum Verstehen des Quelltextes erheblich steigt [17] und damit verbunden nimmt der Testaufwand für das Software-Projekt zu, welcher wiederum für höhere Kosten sorgt [39]. Des Weiteren dauert die Kompilierungszeit der Datei länger. Der größere Codeumfang führt auch zu einem erhöhten Wartungsaufwand eines Systems und somit zu höheren Wartungskosten [2, 11, 5]. Die höheren Wartungskosten werden verursacht durch den erhöhten Aufwand für das Ändern eines geklonten Codefragments. Der Grund hierfür ist, dass Änderungen an einem geklonten Codefragment an allen Klonstellen einer Klongruppe modifiziert bzw. angepasst werden müssen. Die Gefahr besteht darin, dass bei unbewusstem Klonen die Änderungen nicht an allen Klonstellen angepasst werden und die Entwicklung der geklonten Codefragmente unabhängig voneinander erfolgt [6]. Eine unabhängige Weiterentwicklung kann durch das fehlende Verständnis des Entwicklers über das Systems entstehen.

Im Gegensatz dazu gibt es jedoch Studien, die gegenteilige Ergebnisse liefern, wie bspw. dass Klone bewusst erstellt werden, um die Produktivität der Entwickler zu erhöhen [46]. Eine weitere Studie hat bewiesen, dass Entwickler bewusst Klonen, da sie sich an die verschiedenen Klonstellen erinnern und bei einer Änderung diese an allen Klonstellen durchführen [47].

3.3 Inkonsistente Klone und Fehler

Laut Studien enthalten 5-25% der Softwaresysteme redundante Codestellen [21, 40, 41], welche sich negativ im Entwicklungszyklus des Software-Projekts auswirken. Durch das Klonen vergrößert sich nämlich der Codeumfang, sodass der Aufwand zum Verstehen des Quelltextes erheblich steigt [17] und damit verbunden nimmt der Testaufwand für das Software-Projekt zu, welcher wiederum für höhere Kosten sorgt [39]. Des Weiteren dauert die Kompilierungszeit der Datei länger. Der größere Codeumfang führt auch zu einem erhöhten Wartungsaufwand eines Systems und somit zu höheren Wartungskosten [2, 11, 5]. Die höheren Wartungskosten werden verursacht durch den erhöhten Aufwand für das Ändern eines geklonten Codefragments. Der Grund hierfür ist, dass Änderungen an einem geklonten Codefragment an allen Klonstellen einer Klongruppe modifiziert bzw. angepasst werden müssen. Die Gefahr besteht darin, dass bei unbewusstem Klonen die Änderungen nicht an allen Klonstellen angepasst werden und die Entwicklung der geklonten Codefragmente unabhängig voneinander erfolgt [6]. Eine unabhängige Weiterentwicklung kann durch das fehlende Verständnis des Entwicklers über das Systems entstehen.

Im Gegensatz dazu gibt es jedoch Studien, die gegenteilige Ergebnisse liefern, wie bspw. dass Klone bewusst erstellt werden, um die Produktivität der Entwickler zu erhöhen [46]. Eine weitere Studie hat bewiesen, dass Entwickler bewusst Klonen, da sie sich an die verschiedenen Klonstellen erinnern und bei einer Änderung diese an allen Klonstellen durchführen [47].

Des Weiteren befassen sich Juergens et. al. [6] mit der Erkennung und den Auswirkungen der inkonsistenten Klone und Fehlern und stellen fest, dass beim unbewussten inkonsistenten Klonen, jeder zweite Klon einen Fehler verursacht.

Es gibt auch zahlreiche wissenschaftliche Beiträge, die positive Rückschlüsse zu inkonsistenten Klonen liefern. Beispielsweise hat Krinke [45] in seiner Studie bewiesen, dass bei einer konsistenten sowie inkonsistenten Änderung von Klonen nur 50% der Klongruppen einer konsistenten Änderung unterzogen wurde. Außerdem wurde festgestellt, falls eine Klongruppe bereits inkonsistent gewesen ist, diese auch inkonsistent bleibt, da nur ein minimaler Anteil der inkonsistenten Klone durch spätere Änderungen im Laufe der Entwicklung konsistent wird.

Die Studie von Rahman [10] analysiert die Beziehung zwischen Klonen und Fehleranfälligkeit. Zum einen haben sie erkannt, dass die große Mehrheit der Fehler nicht signifikant mit Klonen verbunden sind und zum anderen, dass geklonte Codefragmente weniger fehleranfällig als nicht-geklonte Codefragmente sind [10].

4 Werkzeugunterstützung und Software Systeme

4.1 ConQat

Das folgende Kapitel befasst sich mit dem Tool ConQat, welches in dieser Arbeit zur Klonerkennung verwendet wird. Nach einem kurzen Überblick in die ConQat- Details erfolgt die Beschreibung des Designs und der Architektur. Zuletzt wird die Klonerkennung mittels ConQat im Detail beschrieben.

4.1.1 Überblick zu ConQat

Das Tool Continuous Quality Assessment Toolkit, genannt ConQat, ist ein Werkzeug zur kontinuierlichen Software- Qualitätskontrollanalyse. Die Softwarequalität, welche einen bemerkenswerten Einfluss auf die Wartung und Weiterentwicklung hat, wird in der Entwicklung oft vernachlässigt. Die automatisierte Überwachung diverser Qualitätskriterien ist für die Durchführung von kosteneffizienten und kontinuierlichen Qualitätssicherungsmaßnahmen erforderlich. Ausgehend von diesem Problem wurde ConQat an der Technischen Universität München im Jahre 2007 für den effizienten Aufbau von Qualitätskontroll-Dashboards gegründet. Diese Qualitätsdashboards werden für das Planen und Steuern von IT-Projekten eingesetzt und schaffen einen Überblick über qualitätsrelevante Kriterien in einem Projekt. Hierunter sind ebenfalls Qualitätsdashboards für Klone, bzw. für die Klonerkennung, zu finden, die einen bemerkenswerten Einfluss auf die Qualität einer Software haben können.

Das besondere an ConQat ist, dass es in Zusammenarbeit mit der TU München und der CQSE GmbH kontinuierlich weiterentwickelt und als Open-Source-Software kostenlos angeboten wird.

4.1.2 Design und Architektur

Um den verschiedenen und umfangreichen Qualitätsanforderungen gerecht zu werden, fokussiert sich das Design von ConQat auf die Erweiterbarkeit und Flexibilität. Deshalb wurde ConQat [49] als ein Plug-In Architektur entworfen, welche das Hinzufügen oder Entfernen von Analysemodulen zur Ladezeit ermöglicht. Das ConQat beruht auf einem Pipes&Filter orientiertem Konzept, welches durch ein Netzwerk verschiedener Prozessoren strukturiert ist [49]. Diese Prozessoren sind das zentrale Element von ConQat und wurden in Java implementiert, welche jeweils für eine gewidmete Analyse verantwortlich sind. Die Prozessoren implementieren sehr unterschiedliche Funktionen und arbeiten, indem sie mehrere Inputs akzeptieren und lediglich einen einzigen Output produzieren [50]. Das Output von ConQat, also die Ergebnisse, werden als XML-Dateien und HTML-Seiten ausgegeben. Das besondere an ConQat ist, dass die

Ergebnisse ebenfalls als Graphiken, bspw. Treemap, oder Trends dargestellt werden können. Die ConQat Architektur verfügt über eine Driver Komponente, welche für die Konfiguration des Prozessornetzwerkes und der Weitergabe von Informationen zwischen Prozessoren verantwortlich ist.

Wie auch aus der Abbildung 4.1 zu entnehmen ist, können Prozessoren auf externe Daten, wie das Dateisystem oder auf Datenbanken, entweder direkt oder über einen der mitgelieferten Bibliotheken und Caches zugreifen.

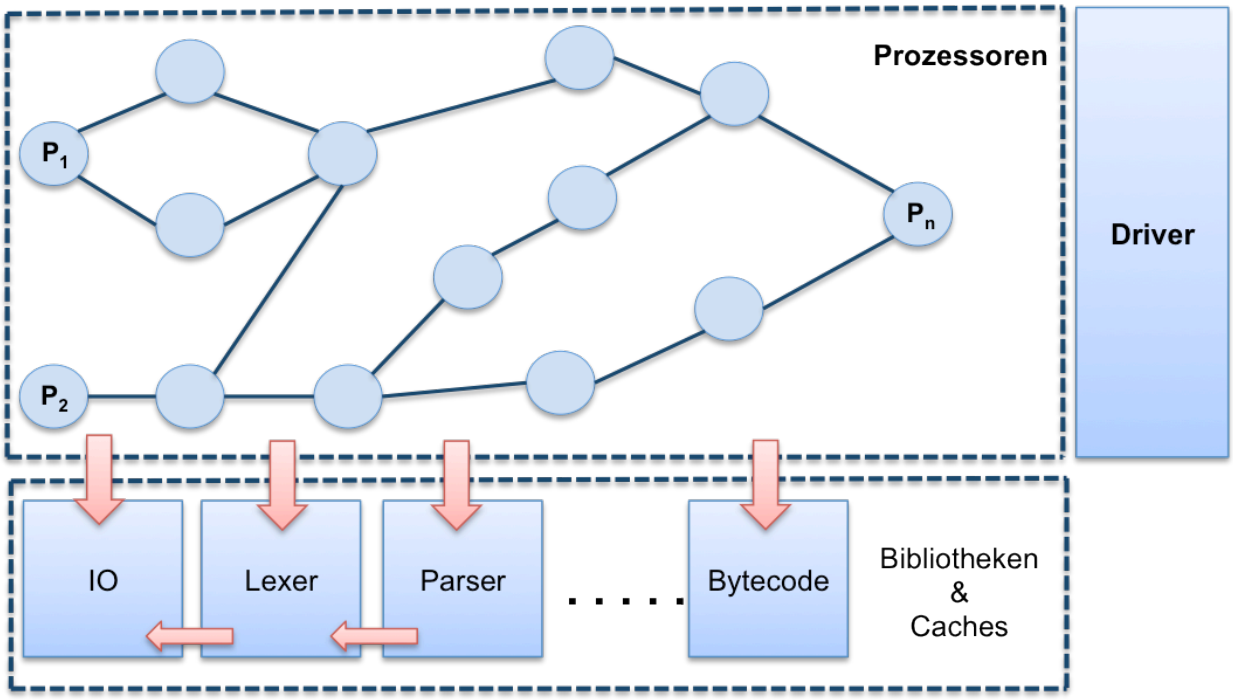

Abbildung 4.1: ConQat Architektur

Die Architektur von ConQat stellt einen leistungsfähigen Konfigurationsmechanismus dar, da die Prozessoren auf vielfältiger Weise miteinander verbunden werden können. Auf Grund der uneingeschränkten Funktionalität der Prozessoren kann ConQat eine Vielfalt von Faktoren, welche die Wartbarkeit oder andere Qualitätsaspekte beeinflussen, bewältigen.

4.1.3 Klonerkennung mit ConQat

In Kapitel 2.5 wurden verschiedene Ansätze zur Klonerkennung beschrieben. ConQat führt die Klonerkennung Token-basiert mittels Syntaxbäumen durch. Die Analyse der TWT-Systeme nach konsistenten und inkonsistenten Klonen erfolgt in dieser Arbeit mit ConQat. ConQat hat durch die kontinuierliche Weiterentwicklung der Funktionalitäten einen hohen Bekanntheitsgrad erreicht [19, 52, 53] und wird in vielen Studien bzw. Forschungsgruppen ebenfalls als Basistechnologie zur Klonerkennung eingesetzt [6, 51].

Da die in dieser Arbeit zu analysierenden Systeme ausschließlich Java Projekte sind, wird lediglich die Klonerkennung für Java Projekte beschrieben. ConQat unterstützt die Klonerkennung für Textdokumente, welche in [51] ihre Anwendung erhalten hat sowie für graphenbasierte Modelle und für Quelltexte. Da ConQat ein plattformunabhängiges Tool ist, unterstützt es die Sprachen ABAP, Java, C#, C/C++, ADA, Visual Basic, PL1 und PL/SQL.

Für die Erkennung exakter Klone wird das ConQat Konfigurationsblock „clonedetection-example.junit.cqr“ ausgewählt, um die Klonerkennung für den Quelltext zu konfigurieren. Im Konfigurationsfenster wird die minimale Klonlänge gewählt. Hier hat sich laut Studien eine minimale Klonlänge von 10 als optimal erwiesen. Anschließend wird unter „input“ der Ordner gewählt, welche den zu analysierenden Quelltext enthält. Im letzten Schritt wird unter „output“ der Ordner festgelegt, in dem die Klonergebnisse zu speichern sind. Nach diesen Angaben wird durch „Launch ConQat analysis“ die Klonanalyse durchgeführt, siehe hierzu Abbildung 4.2.

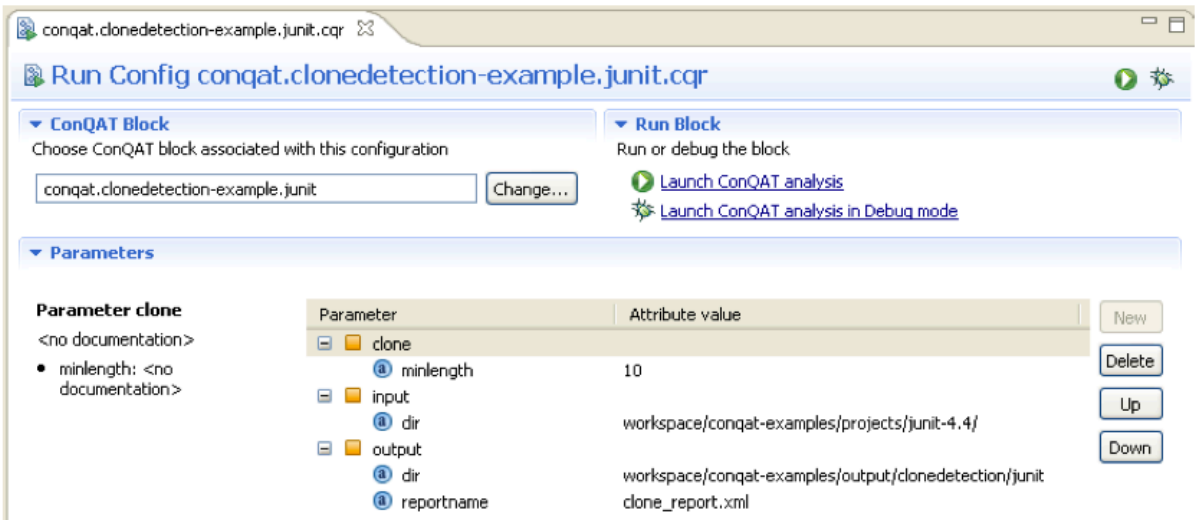

Abbildung 4.2: Klonerkennung- Konfiguration für exakte Klone

Für die Analyse von inkonsistenten Klonen, in ConQat „gapped Clones“ genannt, erfolgt die Klonanalyse ähnlich wie in Abbildung 4.2 zu entnehmen ist. Hierzu wird das Konfigurationsblock „JavaGappedCloneAnalysis.cqr“ gewählt. Diese feinere Form der Klonanalyse erfordert die Angabe einer „gap ratio“ und einer maximalen Fehleranzahl. Das „gap ratio“ gibt an, um wie viele Codezeilen sich ein Klonpaar unterscheiden darf. Es hat sich laut Studien ein gap ratio von 0.25 bewährt, d.h. bei einem 8 Zeilen Code im Klonpaar dürfen sich lediglich 2 Codezeilen unterscheiden. Des Weiteren hat sich eine Fehleranzahl von 10 etabliert. Die Abbildung 4.3 zeigt das Konfigurationsfenster für inkonsistente Klone.

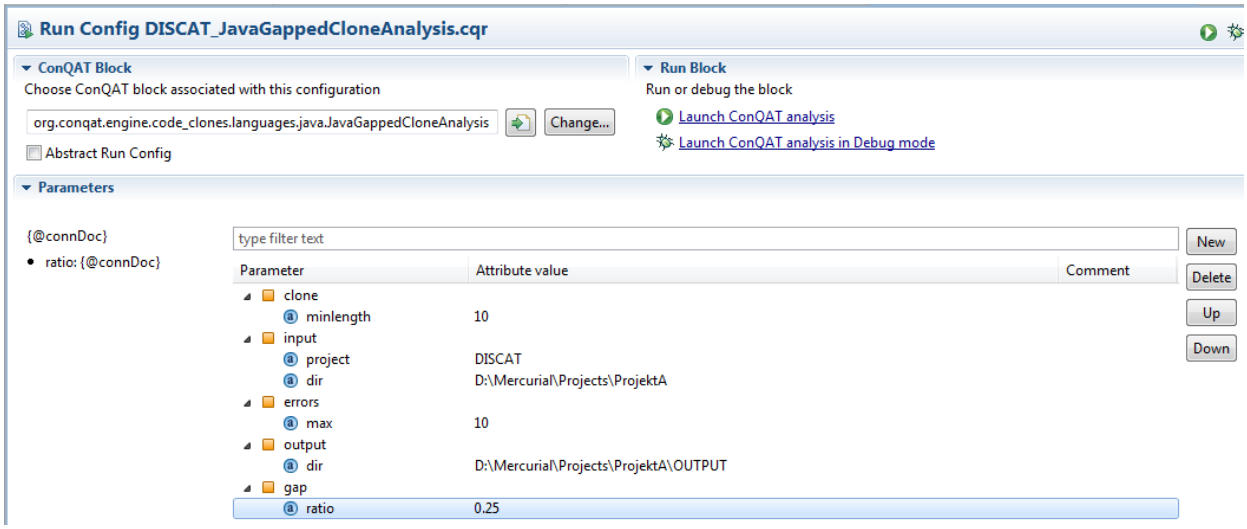

Abbildung 4.3: Klonerkennung- Konfiguration für inkonsistente Klone

In beiden Verfahren wird im Ausgabeordner ein HTML Dokument mit „index.html“ konfiguriert, welches die graphische Darstellung der Klonerkennungsergebnisse, wie in Abbildung 4.4 dargestellt, enthält.

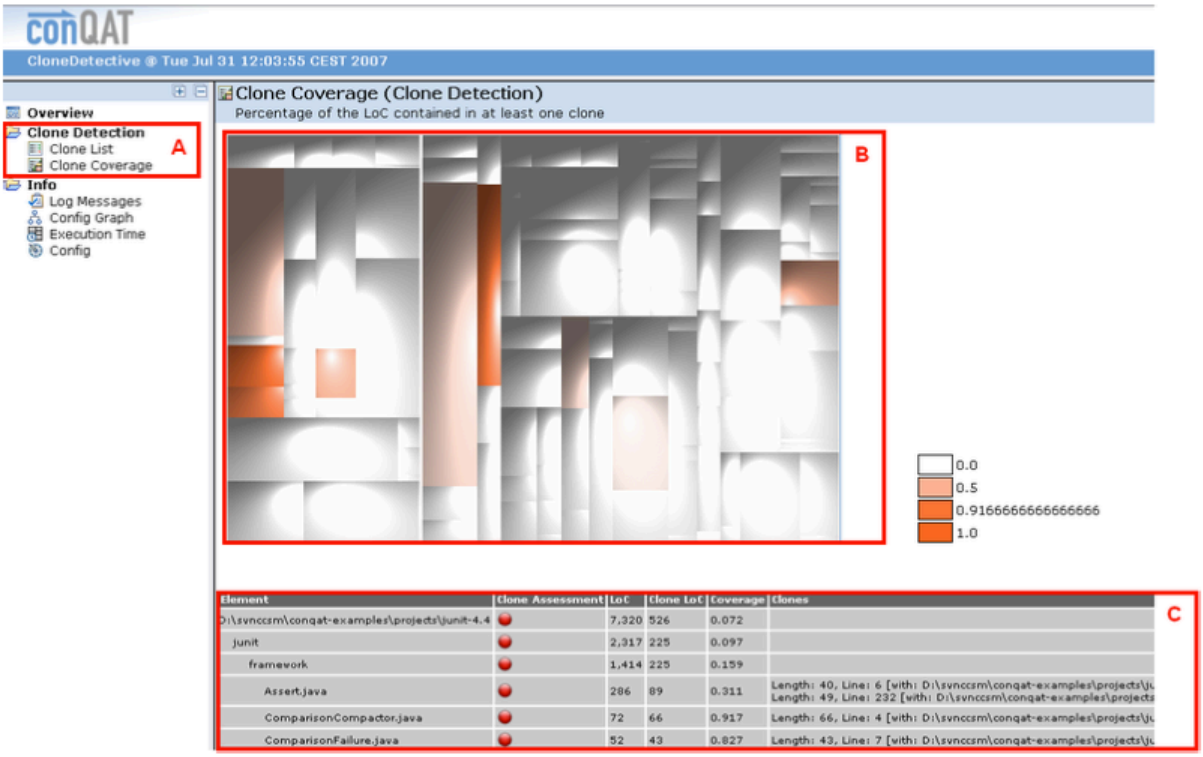

Abbildung 4.4: Übersicht zu den Klonerkennungsergebnissen

Das besondere an ConQat ist die Kloninspektionssicht, welche im festgelegten Ausgabeordner durch die Klonanalyse auf effizienter Art und Weise erstellt wird. Diese Sicht ist die Voraussetzung für die Überprüfung für Falschpositive und für die

Bewertung der Ergebnisse. Sie ermöglicht verschiedene Einsichten zu einem geklonten Code. Es werden explizit die Klonklassen, Klondateien, Klonlängen etc. angegeben, wie aus Abbildung 4.5 zu entnehmen ist.

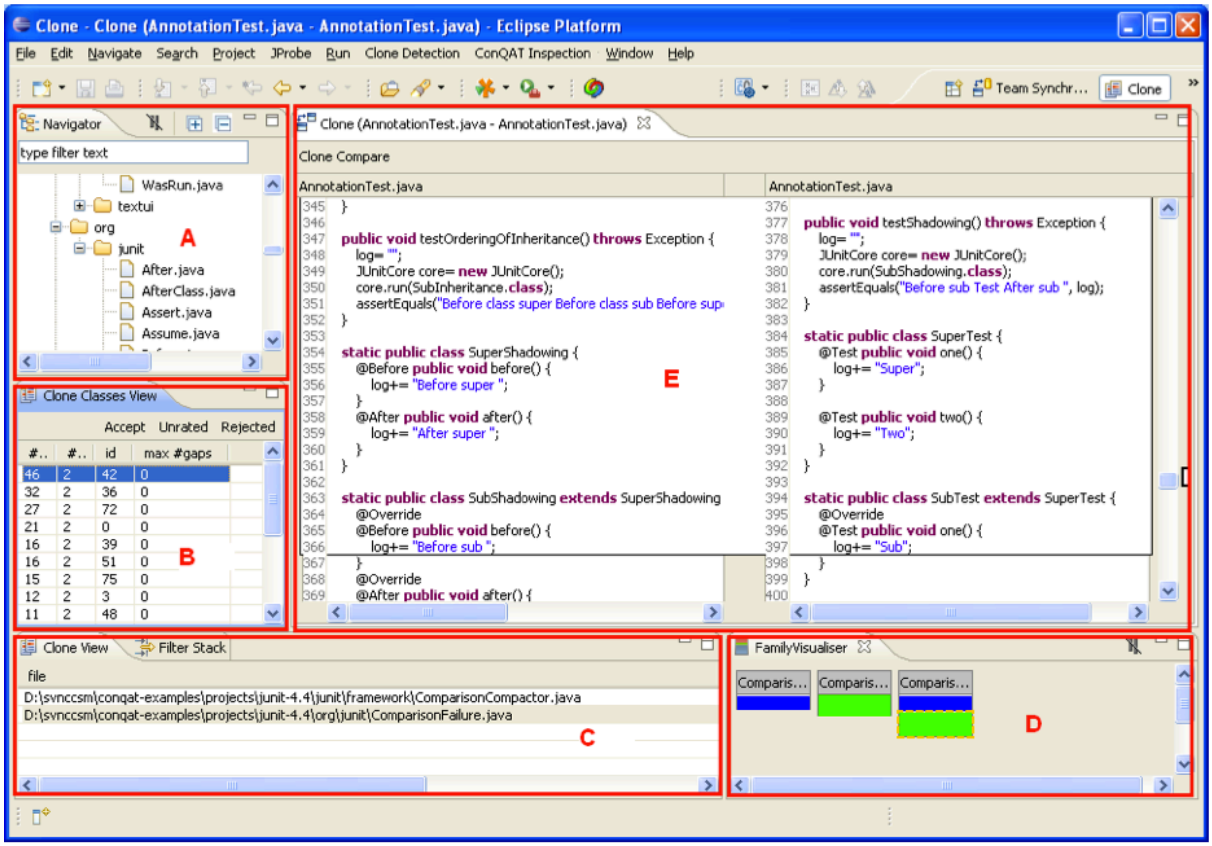

Abbildung 4.5: Klonerkennungsperspektiven

4.2 Mercurial

Mercurial ist ein plattformunabhängiges, verteiltes Versionskontrollsystem. Es verwaltet kleine sowie größere Projekte und stellt einfache und intuitive Schnittstellen zur Verfügung. Bevor ein Versionskontrollsystem überhaupt zum Einsatz kommt, sollte man wissen, was eine Versionskontrolle ist und was für Vorteile ihre Verwendung mit sich bringt. Daher behandelt dieses Kapitel zunächst die Versionskontrolle. Anschließend werden die Struktur sowie die Funktionen von Mercurial detailliert und graphisch beschrieben. Zuletzt befasst sich das Kapitel mit dem für Mercurial zugeschnittenen Frontend – TortoiseHg.

4.2.1 Versionskontrolle

Eine Version ist ein Stand einer Software-Einheit, die durch das Ändern bzw. das Verbessern der Einheit entsteht [31]. Unter Versionskontrolle hingegen versteht man den Prozess der Verwaltung mehrerer Versionen einer Information [30]. Das ist der manuelle Vorgang, wenn eine Änderung an einer Datei auf einer Versionsnummer vorgenommen wird und diese unter einem neuen Namen gespeichert wird und eine Nummer enthält, welche höher als die Versionsnummer ist, auf der die Änderung durchgeführt wurde. Durch die Versionskontrolle können praktisch jede Art von Dateien nachverfolgt werden. Die manuelle Verwaltung von zahlreichen Versionen kann Fehler verursachen. Der Grund hierfür ist, dass mehrere Entwickler an einem Projekt beteiligt sind und zum Teil zeitgleich Änderungen vornehmen. Das kann zu Konflikten in den Versionen führen. Daher ist eine automatisierte Versionskontrolle erforderlich, welche im nächsten Kapitel beschrieben wird.

4.2.2 Funktionen Mercurial

Tools die das Automatisieren der Versionskontrolle ermöglichen, werden Versionskontrollsysteme genannt. Es gibt viele Versionskontrollsysteme, die keine Probleme in der Handhabung von großen Projekten haben. Diese bewältigen problemlos Projekte, an dem Tausende von Entwickler gleichzeitig arbeiten, welche aus einer großen Anazhl von Dateien bestehen [30]. Versionskontrollsysteme ermöglichen das Protokollieren jeglicher Art von Änderungen an einer Datei über die Zeit hinweg. Somit kann zu jedem Zeitpunkt auf verschiedene Versionen sowie Änderungen zugegriffen werden.

Wie bereits genannt ist Mercurial ein plattformunabhängiges, schnelles, leichtgewichtiges und verteiltes Versionskontrollsystem, die für eine einfache und effiziente Verwaltung von großen verteilten Projekten entwickelt wurde. Mercurial wird größtenteils in Python geschrieben. Die Anwendung von Mercurial erfolgt größtenteils über die Kommandozeile, beginnend mit „hg“. Dieses Kapitel gibt eine kurze Einführung über die Funktion des verteilten Versionskontrollsystems sowie in die effektive Nutzung von Mercurial durch die grundlegenden Funktionen.

Verteilte Versionsverwaltung

Bei der verteilten Versionsverwaltung hat jeder Entwickler, im Gegensatz zur zentralen Versionsverwaltung, ein eigenes Repository. Das Repository aus dem Server wird lokal auf den eigenen Arbeitsspeicher kopiert, d.h. geklont. Dies hat den Vorteil, dass falls ein Server beschädigt wird, das Repository von einem beliebigen Entwickler ausgewählt und wieder hergestellt werden kann [32]. Da lokal auf dem eigenen Repository weiterentwickelt wird, ist die Versionsgeschichte dementsprechend verteilt. Ein weiterer Vorteil hierbei ist, dass die Änderungen lokal verfolgt werden können ohne

sich zum Hauptserver zu verbinden. Des Weiteren ermöglicht die verteilte Versionsverwaltung das simultane Arbeiten mehrerer Entwickler an derselben Version, ohne dass Konflikte überhaupt entstehen können.

Sich widersprechende Versionen werden durch mehrere Zweige in der Versionsgeschichte dargestellt, die durch Weiterentwicklung zu einer gemeinsamen Version zusammengefasst werden.

Weshalb unter zahlreichen verteilten Versionskontrollsystemen ausgerechnet Mercurial genutzt werden sollte, begründet B. O'Sullivan [30] wie folgt:

- Mercurial ist leicht zu lernen und einfach zu bedienen.
- Mercurial ist leichtgewichtig.
- Mercurial ermöglicht eine hohe Skalierbarkeit.
- Mercurial ist einfach anzupassen.

Die Abbildung 4.6 stellt die verschiedenen Funktionen in Mercurial dar. In einem verteilten Versionskontrollsystem sowie Mercurial werden folgende Begriffe verwendet:

Repository:

➔ Ein Repository ist ein zentrales Archiv. Diese umfasst in einer Baumstruktur alle Versionen von verschiedenen Dateien sowie ihre Logdateien.

Master Repository:

➔ Das Haupt-Repository ein einem Unternehmen, das den aktuellsten Stand eines Softwaresystems enthält.

Working Directory:

➔ Das lokale Arbeitsverzeichnis eines Entwicklers wird als Working Directory bezeichnet.

Des Weiteren werden verschiedene Funktionen in Mercurial verwendet. Die Tabelle 4.1 schafft einen kurzen Einblick auf die wichtigsten Funktionen in Mercurial und beschreibt sie explizit.

Tabelle 4.1: Funktionen in Mercurial

Funktion	Beschreibung
Clone	Durch die Clone-Funktion wird der ausgewählte Stand des Master Repositorys 1:1 auf das Working Directory kopiert.
Commit	Die Commit-Funktion aktualisiert das lokale Arbeitsverzeichnis mit den Änderungen aus dem Working Directory. Durch Commit legt Mercurial eine neue Revision an.
Update	Durch die Update-Funktion wird der aktuelle Stand des Master Repositorys in das Working Directory übertragen. Das heißt neu hinzugekommene Revisionen werden in das Working Directory hinzugefügt, so dass sich dieser auf dem Zustand des Master Repositorys befindet.
Merge	Durch die Merge-Funktion werden simultane Entwicklungszweige zusammengeführt.
Pull	Die Pull-Funktion zieht die Daten aus fremden Repositorys in das eigene Working Directory.
Push	Durch die Push-Funktion werden die Änderungen bzw. Dateien aus dem eigenen Working Directory in ein fremdes Repository übertragen.
Serve	Die Serve-Funktion startet das Master Repository-Server, um anderen die Pull-, Push- und Clone-Funktion zu ermöglichen

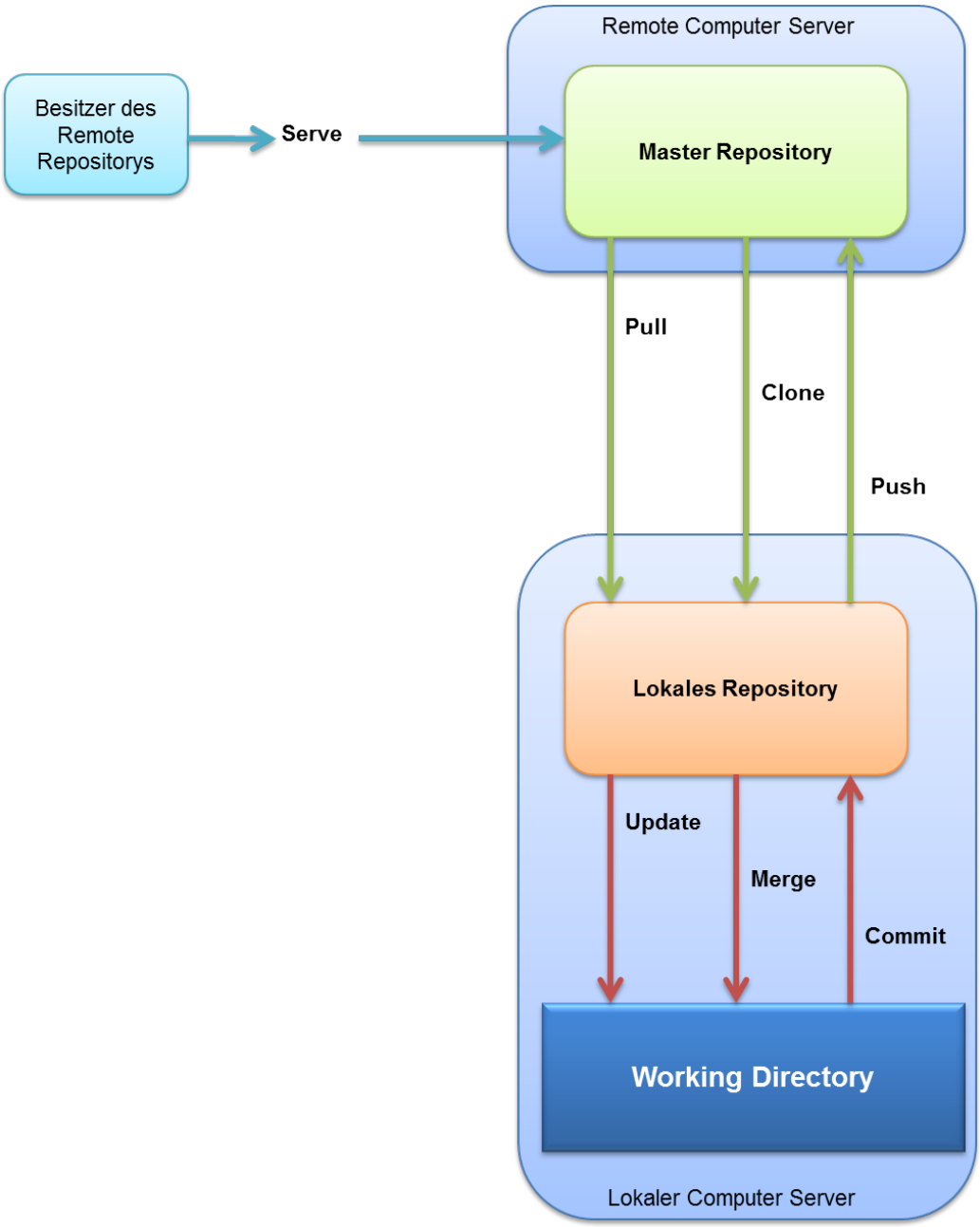

Abbildung 4.6: Die Funktionen von Mercurial

4.2.3 TortoiseHg

Das Tool TortoiseHg ist ein einfach zu bedienendes Frontend und steht mit ihrer graphischen Oberfläche für Microsoft Windows zur Verfügung, um die Benutzung von Mercurial ohne Kommandozeilenbefehle durchzuführen. Die Revisionshistorien der einzelnen Projekte und Dateien können über das Arbeitsverzeichnis in einer graphischen Oberfläche übersichtlich dargestellt werden. Des Weiteren können mit Hilfe von TortoiseHg alle Funktionen von Mercurial ausgeführt werden und die Revisionshistorie wird in Form einer Baumstruktur sehr übersichtlich dargestellt.

4.3 FogBugz

Das webbasierte System FogBugz ist ein Projektmanagementsystem sowie ein Issue-/Bug-Tracking-System, welche umfangreiche Funktionalitäten für Entwicklerteams anbietet.

Ein Issue-/ Bug-Tracking-System ist ein Werkzeug, mit dem diverse Aufgaben in einem Projekt, wie das Bearbeiten von Kundenanfragen und Entwicklervorschlägen, welche in Tickets bzw. Fällen (Cases) angelegt und einzelnen Personen zugewiesen sind, verwaltet werden. Diese Fälle können Bugs, Features, Scheduled Items sein und werden mit Prioritätslevel belegt. FogBugz verfolgt alle Fälle und Tickets an einem zentralen Ort, damit in der Entwicklungshistorie nichts vergessen wird [34].

Des Weiteren können Fehler sehr leicht und schnell in FogBugz über das FogBugz Screenshot- Tool, per E-Mail oder über den Browser erfasst werden. Dadurch kann auch jeder Fall in kürzester Zeit bearbeitet und im Falle eines Fehlers behoben bzw. gefixt werden. Der Bearbeiter eines Cases aktualisiert anschließend den Status seines Fortschritts. Dadurch werden fertige Features wieder an den Projektleiter zurückgesendet.

Eine besondere Eigenschaft von FogBugz ist die facettenreiche Suchoption. Es können die gesamte Fallliste aber auch Wiki- und Kundenbeiträge gesucht werden.

Das Projektmanagementsystem von FogBugz bietet diverse Funktionalitäten für die Entwickler eines Projektes an. Folgende Managementaufgaben sind mit dem Einsatz von FogBugz möglich [33]:

- Das Erfassen der Aufgaben mit Fällen und Unterfällen (bzw. Cases und Subcases)

  → In FogBugz wird jede Aufgabe (also Issue) und jeder Fehler (also Bug) mit einem Fall (also Case bzw. Subcase) verbunden. Diese werden von dem verantwortlichen bzw. zum Fall eingetragenen Entwickler bearbeitet.

  → Ein Fehlereintrag in FogBugz kann in der Case-Liste, wie in Abbildung 4.7, durch den roten Käfer erkannt werden.

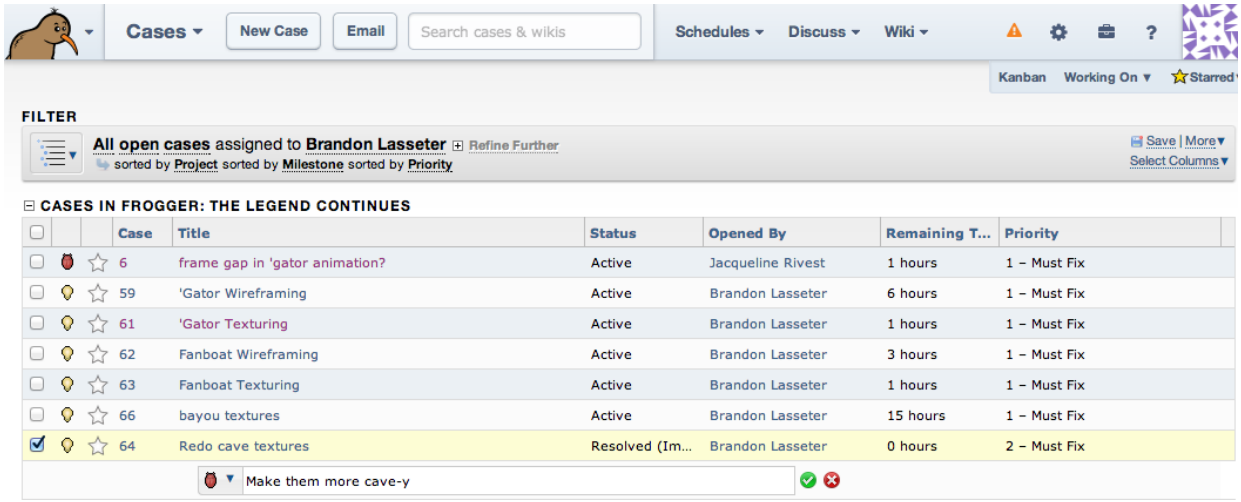

Abbildung 4.7: Bugeintrag in FogBugz [33]

- Das Erstellen von Meilensteinen

  → Üblicherweise wird für einen Projekt ein Projektplan erstellt, welcher auf Meilensteinen basiert. FogBugz ermöglicht das Hervorheben von wichtigen Terminen in den Meilensteinen sowie das Erstellen von Meilensteinen.

- Visualisierung von Falldaten
  → FogBugz bietet eine große Vielfalt an graphischer Darstellung bzw. Auswertung von Falldaten.

- Kontrolle der Änderungen über die Entwicklungszeit
  → FogBugz speichert Snapshots über die Revisionshistorie eines Falls, einschließlich der Details zum Fall.

- Zusammenarbeit mit Teamkollegen
- Treffen von Fristen
- Kontrolle über Ihre Projekte
- Integration mit Versionskontrolle

4.4 Kiln

Kiln ist ein webbasiertes System für das Quellcodehosting von Git und Mercurial. Git ist ebenfalls wie Mercurial ein Versionskontrollsystem. Des Weiteren bietet Kilnden Entwicklern durch die Leistungen von Git und Mercurial eine Einfachheit bei der Entwicklung von Softwareprojekten und zudem eine ausgezeichnete Gelegenheit Softwarecodes zu verwalten und den größten Nutzen daraus zu ziehen.

Bei der Nutzung von Mercurial bzw. Git stellt der Kiln-Server den zentralen Punkt des Versionskontrollsystems dar. Dabei werden die Daten zentral abgelegt. Mit anderen Worten hat also Kiln eine zweifache Funktion. Zum einen dient er als Datenspeicher

und zum anderen als Verteilknoten für den Quelltext sowie für weitere projektrelevante Dateien. Dies ermöglicht eine verteilte Softwareentwicklung in Teams.

Es sprechen viele aussagekräftige Argumente für die Nutzung von Kiln. Einige dieser lauten wie folgt [36]:

1) Kiln ermöglicht die synchrone Entwicklung der Entwickler und unterstützt die Entwickler bei der gemeinsamen Entwicklung.

➔ Kiln enthält jeden Stand des Softwareprojektes. Des Weiteren können Entwickler eines Teams gegenseitig in die Änderungen des Quelltexts einsehen.

2) Kiln bietet eine verteilte Versionskontrolle eines Softwareprojekts an.

➔ Es können mehrere Entwickler an einem Code arbeiten, so dass unterschiedliche Quelltexte (also Branches) entstehen, und diese dann durch Kiln zusammengeführt werden (Merge) (siehe Abbildung 4.8).

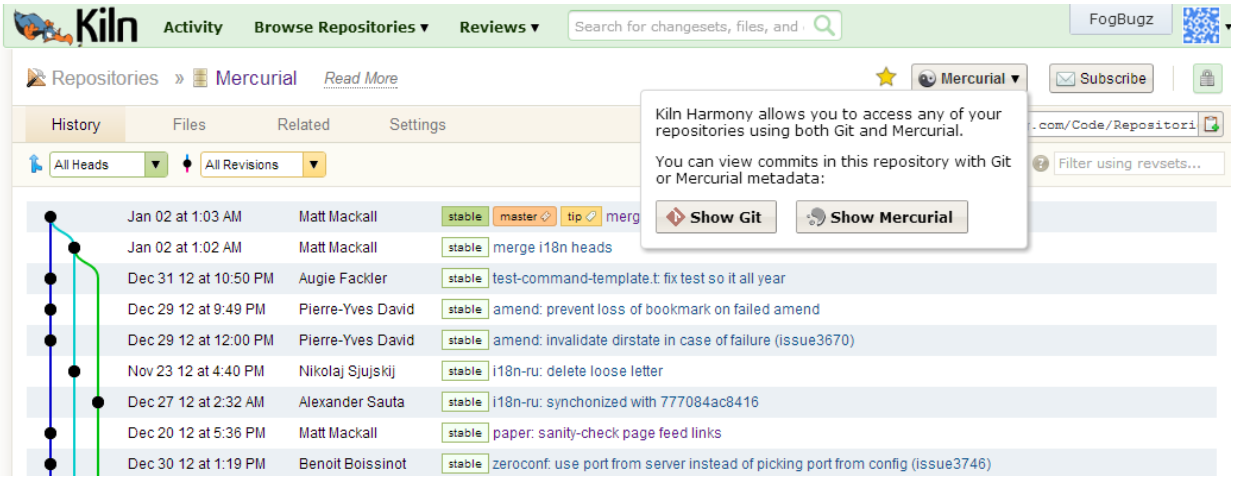

Abbildung 4.8: Verteilte Versionskontrolle in Kiln [36]

3) Kiln ermöglicht die Integration von FogBugz

➔ Die Integration von FogBugz geschieht über denselben Login wie Kiln. Mit anderen Worten, Kiln und FogBugz teilen sich ein Login. Durch einen gemeinsamen Login ist ein reibungsloser und gemeinsamer Arbeitsablauf verfügbar. Der Vorteil hierbei ist, dass Entwickler beim Entwickeln nicht zwischen zwei Systemen wechseln müssen.

➔ Mit Kiln und FogBugz können Fehler gemeldet und behoben werden, in dem der Code weiterentwickelt wird. Bei der Fehlerbehebung besteht die Möglichkeit alle Codestellen eines Bereichs zu aktualisieren, in dem der Fehler vorhanden ist.

4) Kiln ermöglicht eine übersichtliche Organisierung von Projekten und Repositorys
➔ In Kiln werden die Repository Management Seiten sowie die Projekteinstellungsseiten auf einer einzigen Seite dargestellt, so dass die Navigation einfach durchzuführen ist und die Seiten leicht zu lesen sind.

5) Kiln bietet eine äußerst leistungsfähige Codesuche.
➔ Über das integrierte Suchfeld besteht die Möglichkeit sehr schnell auf „Changesets“ (über die „ChangesetID“), Dateien und Codes zuzugreifen. Die Suchoption in Kiln ist sehr effektiv bei der Suche nach Problemcodes, die behoben werden müssen, sowie bei der Suche nach guten und nützlichen Codestellen, die wiederverwendet werden können [37].

5 Studiendesign

In diesem Kapitel wird das Herzstück der Arbeit präsentiert. Um einen besseren Eindruck auf die mögliche Fehlerträchtigkeit der inkonsistenten Klone zu erhalten wurde ein Studiendesign entwickelt. Dieses Kapitel umfasst das Studiendesign zur Analyse der TWT-Systeme. Zunächst werden die Studienobjekte mit faktischen Daten beschrieben. Das darauf folgende Unterkapitel beschreibt die Forschungsfragen. Anschließend wird geschildert, wie die notwendigen Daten für die Analyse aufbereitet werden. Mit diesen Daten erfolgt die Durchführung der Studie mittels einer Datenbankanwendung. Das letzte Unterkapitel beschreibt für jede Forschungsfrage die Vorgehensweise der Datenanalyse.

5.1 Studienobjekte

Die Studie wird auf den Systemen der TWT GmbH durchgeführt. Als Quellen für die Softwaresysteme wurden drei Projekte gewählt, die in Entwicklung sind und über eine lange Entwicklungshistorie verfügen. Alle Systeme werden ausschließlich in Java durch verschiedene Teams und mit verschiedenen Funktionalitäten entwickelt. Die Anzahl der Systeme und die Beteiligung mehrerer Entwickler an einem Projekt führen zu besseren Analyseergebnissen der Studie. Des Weiteren sind die Systeme bereits im Einsatz und werden kontinuierlich weiterentwickelt und angepasst. Aus Datenschutzgründen werden die Namen der Systeme nicht genannt und erhalten eine Bezeichnung von A bis C. Detaillierte Informationen zu den Systemen, wie Alter und Codezeilen, sind in Tabelle 5.1 erfasst.

TWT steht für Technisch-Wissenschaftlicher Transfer und stellt die rasche Umsetzung wissenschaftlicher Expertise in technologisch anspruchsvolle Produkte und Dienstleistungen in den Geschäftsfeldern Information & Engineering Technologies dar. Das Portfolio umfasst die Software-, Produkt-, und Prozessentwicklung sowie die technische Beratung und Industrieforschung. An den Standorten Stuttgart, München, Friedrichshafen und Ingolstadt entwickelt die TWT GmbH seit 1986 als Technologiepartner der Branchen Automotive, Aerospace, Healthcare und Energy, für ein breites Spektrum an Kunden, eine ganzheitliche und maßgeschneiderte Lösung.

Die in dieser Studie untersuchten Systeme A bis C wurden für verschiedene Unternehmen aus der Automobilindustrie entwickelt und stellen wesentlich verschiedene Funktionalitäten dar. Die Systeme werden seit vier bis fünf Jahren von insgesamt 25 bis 30 Entwicklern entwickelt und gewartet. Des Weiteren nutzen alle Systeme dieselbe firmeninterne Bibliothek.

Tabelle 5.1: Informationen zu den analysierten Systemen

System	*Organisation*	*Sprache*	*Größe (kLOC)*	*Revision*	*Alter*	*Entwickler*
A	Automobil-industrie	Java	253	2740	4 Jahre	10
B	Automobil-industrie	Java	332	1622	5 Jahre	5
C	Automobil-industrie	Java	454	2181	4 Jahre	10

5.2 Forschungsfragen

Das zugrundeliegende Problem, das wir analysieren, ist der Zusammenhang zwischen inkonsistenten Klonen und ihren Fehlern. Dieses Problem wirft einige Fragen auf, welche sich durch detaillierte Analysen beantworten lassen und ein besseres Verständnis schaffen. Die Studie beruht auf drei Hauptfragen die sich zum Teil durch die Untergliederung in weiteren Fragen beantworten lassen.

Forschungsfrage 1: *Enthalten Systeme inkonsistente Klone?*

An erster Stelle muss geklärt werden, ob die zur Analyse stehenden Systeme inkonsistente Klone enthalten. Hier steht jedoch der Anteil der inkonsistenten Klone im Vordergrund. Es wird also ermittelt, ob die inkonsistenten Klone signifikant höher sind als die restlichen Klone des Systems. Das Verhältnis der inkonsistenten Klone zu exakten Klonen ist also eine Analyse, welche die Analyseergebnisse unterstützen soll.

Forschungsfrage 2: Können inkonsistente Klone Indikatoren für Fehler sein?

Nachdem die inkonsistenten Klone in den realen Systemen ermittelt wurden, wird festgestellt, ob sie eine Verantwortung für Fehler tragen. Hierzu wird die Revisionshistorie der Datei in Betracht gezogen, ob sich im Laufe der Zeit in den inkonsistenten Klonen Fehler bilden. Für ein umfangreicheres Verständnis lässt sich diese Frage durch das Beantworten weiterer Fragen rückschließen. Jede Unterfrage dient als Antwortbaustein zur Hauptfrage. Wichtig ist hierbei, den Zusammenhang der inkonsistenten Klone mit einem Issue-Tracking-System zu analysieren sowie die inkonsistenten Klone zu analysieren, die keinen Bezug zu einem Issue-Tracking-System haben. Daraus lassen sich informative faktische Daten ermitteln. In diesem Zusammenhang ist es von Bedeutung die Inkonsistenten auf Fehlerkorrektur zu analysieren sowie die Gründe für Fehler durch Entwicklerbefragung zu ermitteln.

Die Hauptfrage lässt sich durch folgende Unterfragen beantworten:

2.1) Ist die große Mehrheit der inkonsistenten Klone als Fehler erfasst?

Die Frage die hier beantwortet werden soll ist, ob die erkannten inkonsistenten Klone überhaupt in dem verwendeten Issue-Tracking-System als ein Fehler erfasst sind. Hier ist der Anteil der gekennzeichneten Inkonsistenzen wichtig. Daraus lässt sich nämlich ermitteln, wie viele der gesamten inkonsistenten Klone überhaupt Fehler enthalten, die erkannt und zum Beheben im Issue-Tracking-System erfasst sind.

2.2) Werden Fehler an einem geklontem Code konsistent an allen geklonten Codes einer Klongruppe behoben?

Nachdem die Fehler in den inkonsistenten Klonen ermittelt wurden, soll analysiert werden, ob Fehler, die in dem verwendeten Issue-Tracking-System erfasst sind, in allen Codes einer Klongruppe behoben wurden. Daraus werden Ergebnisse erzielt, die besagen, ob Fehler, trotz dessen dass sie in einem Issue-Tracking-System zur Behebung erfasst sind, weiterhin noch eine Gefahr darstellen oder falls ein Fehler in einem Klon einmal erkannt wurde, keine Gefahr mehr für das System darstellt und sich von der Kategorie der gefährlichen Klone ablöst.

2.3) Spielt die Größe der inkonsistenten Klone eine Rolle für die Häufigkeit der Fehler?

Nach dem der Zusammenhang der Fehler und Inkonsistenzen ermittelt wurde, ist es wichtig festzustellen, unter welchen Kontextbedingungen Fehler gegeben sind. Mit der Annahme, dass der Code doppelt so lang ist, die Wahrscheinlichkeit, dass ein Fehler doppelt so häufig eintreten kann, ist es interessant festzustellen, dass ein inkonsistenter Klon mit größerem Codeumfang, mehr Fehler einbringen kann.

2.4) Was ist der Zusammenhang zwischen Inkonsistenten und Fehlern?

Diese Frage unterscheidet sich von den vorherigen Fragen. Nachdem die Fehler in den Inkonsistenzen ermittelt wurden, wird festgestellt, welche Vorgehensweisen beim Klonen einen Fehler verursachen. Hierzu wird das Klonverhalten des Entwicklers analysiert und unter welchen Umständen ein Klon zu einem Fehler führt. Des Weiteren werden die inkonsistenten Klone analysiert, die über die gesamte Revisionshistorie keine Fehler enthalten.

Forschungsfrage 3: Wie viele Type-1-Klone mit einem Fehler werden durch das Modifizieren für die Fehlerbehebung zu einem Typ-3-Klon ohne Fehler?

Nachdem analysiert wurde, ob Klone tatsächlich Indikatoren für Fehler sind, ist es interessant festzustellen, ob Typ-1-Klone, die mit einem Issue-Tracking-System verbunden sind und entwickelt werden, um den Fehler zu beheben, zu einem Typ-3-Klon werden und somit keinen Fehler mehr enthalten. Durch diesen Vorgang wird ermittelt, ob bewusst entwickelte inkonsistente Klone einen Beitrag zur Fehlerbehebung leisten.

5.3 Datensammlung und Konfigurationssysteme

Die Analyse der Projekte auf Klone und Fehler erfordert viele Schritte und das Beachten vieler Details, die bewusst durchzuführen sind. Das Kapitel beschreibt die Konfigurationsschritte der Analyse, die erforderlich waren, um zuverlässige Ergebnisse erzielen zu können.

Gemein haben alle Schritte die Datenbasis. Das Überprüfen auf Klone auf jeder Version jeder Datei ist nicht machbar. Stattdessen wurde das Projekt in der Revisionshistorie zu einem bestimmten Zeitpunkt im Entwicklungszyklus und auf einer bestimmten Version mit der Update-Funktion gespeichert, das Snapshot genannt wird. Nach dem Update stehen lediglich die Daten bis zum festgelegten Zeitpunkt im Verzeichnis für die Analyse zur Verfügung. Bei allen Projekten wurde ungefähr ein zwei bis drei Jahre früherer Entwicklungsstand als Datenbasis ausgewählt. Der Grund hierfür ist, dass nach der Klonermittlung eine größere Datenbasis in der Revisionshistorie zur Verfügung steht, um die inkonsistenten Klone in der gesamten Revisionshistorie bis zum Zeitpunkt der Analyse auf Fehler zu untersuchen bzw. auf Weiterentwicklung und Fehlerbehebung zu prüfen.

5.3.1 Klondaten aus ConQat

Die Klonerkennung wird mit dem bereits in Kapitel 4 vorgestellten Klonanalysewerkzeug ConQat auf dem ausgewählten Snapshot durchgeführt. Alle Projekte wurden in Java geschrieben. Infolgedessen wurde der ConQat-Block „JavaGappedCloneAnalysis.cqr“ zur Erkennung inkonsistenter Klone auf den drei Objekten durchgeführt. Der Algorithmus für inkonsistente Klone, bzw. gapped Clones, wurde von Juergens et. al. entwickelt [6].

Der inkonsistente Klonerkennungsansatz wurde mit konservativen und liberalen Klonerkennungsparametern durchgeführt. Dies sollte die Ausrichtung der Studie auf eine bestimmte Klonerkennungsparametereinstellung reduzieren, um das Systemverhalten zu verstehen und wie die Klone mit größerem Freiraum, also kleinerem Parameter für die minimale Klonlänge, unähnlich geworden sind.

Für den liberalen Klonerkennungsansatz wurde für die minimale Klonlänge (Minlength) 10 Statements festgelegt, d.h. die Klone müssen mindestens 10 Zeilen lang sein. Für

das „gap ratio“, d.h. das maximale inkonsistente Klonverhältnis, wurde ein Parameter von 0,25 und für die maximale Fehleranzahl ein Parameter von 10 festgelegt. Ein „gap ratio“ gibt an, um wie viele Codezeilen sich ein Klonpaar unterscheiden darf. Beispielsweise dürfen sie bei einem gap ratio von 0,25 und acht Zeilen Code in einem Klonpaar maximal zwei Codezeilen unterscheiden. Die Klonerkennung für den liberalen Ansatz (genannt Runtime) betrug zwischen 62 Sekunden bis 294s. Die Tabelle 5.2 enthält wichtige Informationen zum Klonerkennungsergebnis. Die Definitionen zu den restlichen Begriffen auf Tabelle 5.2 lauten wie folgt:

kLOC: Anzahl der Codezeilen (in Tausend)

Clone LOC: Anzahl der geklonten Codezeilen

Clone Count: Anzahl der Klone

Tabelle 5.2: Klonerkennung mit liberalem Ansatz

Project	Minlength	Error	Gap Ratio	Runtime	kLOC	Clone LOC	Clone Count
A	10	10	0,25	58s	253	25.443	981
B	10	10	0,25	58s	332	49.200	1.545
C	10	10	0,25	112s	454	47.800	2.244

Des Weiteren wurde eine Klonerkennung mit denselben Parametern durchgeführt. Die minimale Klonlänge wurde auf 15 erhöht. Dies führt zu erheblich niedrigeren Klonergebnissen.

Für diese Studie fiel die Entscheidung auf eine konservative Klonanalyse. Daher wurde die minimale Klonlänge erneut erhöht und auf 20 festgesetzt.

Die Tabelle 5.3 zeigt die Klonergebnisse für den konservativen Ansatz. Im Vergleich zum liberalen Ansatz und der manuellen Analyse der Klone ist deutlich zu erkennen, dass der konservative Ansatz erheblich bessere Klonergebnisse liefert.

Tabelle 5.3: Klonerkennung mit konservativem Ansatz

Project	Minlength	Error	Gap Ratio	Runtime	kLOC	Clone LOC	Clone Count
A	20	10	0,25	52s	253	7.600	143
B	20	10	0,25	42s	332	17.700	352
C	20	10	0,25	97s	454	15.600	382

Die erfassten Klonkandidaten wurden dann manuell gelesen, um Falsch-Positive zu entfernen. Es wurden also Codefragmente, welche von ConQat als Klon erkannt wurden, jedoch keine semantische Beziehung hatten, aussortiert. Für die weitere Analyse wurden die restlichen Klonkandidaten als Basis genommen. Diese Klonkandidaten werden in ConQat mit weiteren Informationen in Klonklassen gegliedert ausgegeben.

Es wurden aus dem ConQat Output die Klonklassen mit dem Dateinamen (wobei es sich hier um die Dateipfade handelt), die Anfangszeile, die Endzeile und die maximal möglichen Gaps extrahiert und in Excel exportiert. In der Excel-Liste sind unter jeder Klonklasse die Klondateien mit den extrahierten Dateien erfasst. Diese wir mit Excel-Verweisen und Funktionen so umgestaltet, dass zu jeder Klondatei die Klonklasse angegeben wird, in der sie enthalten ist, sowie die oben genannten Daten wie Anfangszeile, etc. Der Grund hierfür ist, dass die Dateien in dem Versionsverwaltungssystem auf Klone analysiert werden und diese als Datenbasis dienen. Des Weiteren finden diese Daten später in einer Datenbank Anwendung. Daher ist dieses Format der Datenliste relevant.

Eine wichtige Information ist, dass in diesem Schritt noch keine Trennung in der Handhabung zwischen inkonsistenten und konsistenten Klongruppen gemacht wird. Diese werden während der Auswertung in der Datenbank beachtet.

5.3.2 Daten aus Mercurial

In dieser Studie wird die gesamte Revisionshistorie des Projektes zur Analyse betrachtet. Daher muss für jede Datei in der sich ein Klon befindet, sei es ein konsistenter oder inkonsistenter Klon, die gesamte Revisionshistorie aus Mercurial ermittelt werden. Infolgedessen dienen die aufbereiteten Ergebnisse der Klondaten aus ConQat in diesem Schritt als Datenbasis. Aus dieser Liste werden nämlich lediglich die Dateipfade eines Projekts, in der Klone enthalten sind, in eine Textdatei gespeichert.

Es wurde in Python ein Skript geschrieben (siehe Abbildung 5.1), das für jede in der Textdatei enthaltene Datei, die gesamte Revisionshistorie aus Mercurial ermittelt und in eine Textdatei speichert. Die Revisionshistorie besteht aus „Changesets“, welche durch Committs entstanden sind. Zu jedem Changeset wird die lokale und eindeutige „ChangesetID“ angeben. Des Weiteren sind zu jedem „Changeset“ der Benutzer, der das Committ ausgelöst hat, der Zeitpunkt des „Committs“, sowie die „Branch“ und der Parent des Committs und eine Beschreibung des Committs erfasst. Für die Analyse sind jedoch die ChangesetID, der Benutzer, der Zeitpunkt sowie die Beschreibung des Changesets relevant. Das Skript wird in einer Python-Kommandozeile, für jedes einzelne Projekt, wie in Abbildung 5.2 dargestellt ausgeführt.

Die Ausgabe des Skripts wird in eine Excel-Datei exportiert und mit Excel-Verweisen und Funktionen umgeschrieben. Die Ausgabe listet nämlich zu jeder Datei, die in der vorherigen Textdatei erfasst war, die „Changesets“ mit den genannten Informationen untereinander auf. Als Datenbasis benötigen wir jedoch eine Liste, die jede ChangesetID in eine Zeile erfasst und die dazugehörigen restlichen Informationen wie Benutzer, Dateiname, Beschreibung etc. in den Spalten derselben Zeile erfasst. Diese Struktur ist für die Datenbankanwendung erforderlich.

```
# Starten mit
# import hg
# hg.run("src.txt", "log.txt")

import os, subprocess

def run(listFilePath, logFilePath):
  log = ""
  f = file(listFilePath)
  for line in f.readlines():
    log += hglog(line.strip(), logFilePath)

  f.close()
  f = file(logFilePath, "w")
  f.write(log)
  f.close()

def hglog(srcFilePath, logFilePath):
  log = ""
  log += "%s:\n\n" % srcFilePath
  log += subprocess.check_output(["hg", "log", srcFilePath])
  log += "\n\n\n"
  return log;
```

Abbildung 5.1: Skript für die Ausgabe der Revisionshistorie der Klondateien

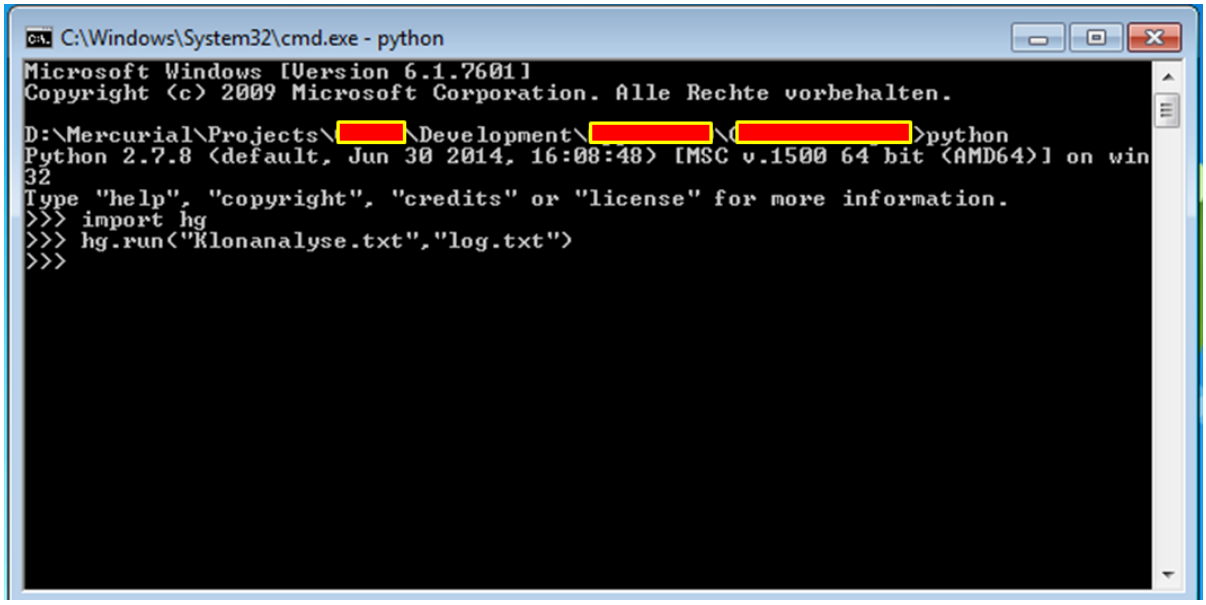

Abbildung 5.2: Skript in Python durch Mercurial-Export ausführen

5.3.3 Daten aus FogBugz und Kiln

Nachdem die gesamte Revisionshistorie zu den Klondaten ermittelt wurde, müssen noch die Fehler in den Klondaten ermittelt werden. Als Anmerkung – die Revisionshistorie der Klondaten besteht aus ChangesetIDs mit zusätzlichen Informationen wie in Kapitel 5.3.2 erläutert. Bei der Fehlersuche hat zunächst die Art der Klonklassen, ob inkonsistent oder konsistent, keine Bedeutung. Es müssen nämlich die Fehler für alle geklonten Codefragmente ermittelt werden.

In Kapitel 4.3 und 4.4 wurden die webbasierten Issue- bzw. Bug- Tracking-Systeme ausführlich beschrieben. Codestellen die bearbeitet werden müssen und bearbeitet wurden bzw. einen Fehler enthalten oder deren Fehler behoben wurden, werden in FogBugz mittels Cases (Fällen) festgehalten. Jeder Case hat eine eindeutige Case-Nummer. Wird ein Quelltext ausgehend von einem Case modifiziert, wird bei der Commit-Message eine Case-Nummer als Referenz angegeben. Dadurch enthält jedes Changeset, bei dem ein Case bearbeitet wurde, einen Case-Eintrag. Dieser Case kann, entsprechend des Cases, ein Feature oder Bug-Eintrag sein, welcher in Kiln über die ChangesetID gefunden werden kann. Zusammengefasst bedeutet dies, dass ChangesetIDs in Kiln geprüft werden, um Fehler (also Bug- Einträge) zu ermitteln.

Für diese Studie wurde jede ChangesetID der Revisionshistorie einer Klondatei in Kiln gesucht. Anschließend wurde geprüft, ob zu dieser ChangesetID ein Case-Eintrag besteht. Die Liste mit den Revisionshistorien wurde mit den referenzierten Case-Nummern, wie Feature-Nummer oder Bug-Nummer erweitert. Durch diesen Analysevorgang wurden alle Klondateien ermittelt, die modifiziert wurden, um einen Fehler zu beheben. Ob jedoch die modifizierten Zeilen mit den Klonzeilen übereinstimmen wird in der Studiendatenauswertung in Kapitel 5.4 analysiert.

Es besteht die Möglichkeit den Prozess zur Analyse von Case- Einträgen mit einer Kiln-API zu automatisieren. Die API gibt aus der Repository eines Projekts, beginnend vom letzten Changeset des Repositorys, 100 Changesets sowie die Case-Einträge in den Changesets zurück. Die API lautet wie folgt [38]:

Api/{version}/Repo/{ixRepo}/History (GET)

also:

https://XXXXXXX.de/fogbugz/kiln/Api/1.0/Repo/100538/History?revOldest=ae5d2dcb44a7&nChangesetLimit=100&token=h6jejas66etdc177ulq7m3l9hpisc

Hierbei ist 100538 die Repository-Nummer des Projekts und ae5d2dcb44a7 die letzte ChangesetID im Repository des Projekts, ChangesetLimit=100 ist die höchste Anzahl der Changesets, die zurückgegeben wird.

Für diese Studie benötigen jedoch die Case-Daten für alle Changesets. Das bedeutet, dass die API erweitert werden müsste, um die Informationen für die gesamte Revisionshistorie zu erhalten. Der Prozess für die Erweiterung der Kiln-API ist jedoch nicht im Rahmen dieser Diplomarbeit und wurde aus diesem Grund nicht durchgeführt.

5.4 Studiendatenauswertung

Bisher erfolgte lediglich das Zusammenstellen der Datenbasis für den eigentlichen Analyseschritt. Ein wichtiger Punkt ist an dieser Stelle anzumerken. Der inkonsistente Klonanalyseansatz wurde auf zwei verschiedenen, zu einem relativ aktuellen und bis zu zwei Jahre früheren, Entwicklungsständen durchgeführt. Nach dem ersten Analyseschritt konnte für den aktuellen Entwicklungsstand eine höhere Anzahl von Fehlereinträgen ermittelt werden. Jedoch lag das Problem darin, dass die Weiterentwicklung der Klondateien nach der Klonanalyse so gering war, dass sehr wenige Daten in der Revisionshistorie für die Analyse auf Weiterentwicklung und Fehlerbehebung zur Verfügung standen. Aus diesem Grund wurde für alle Projekte ein zwei bis drei Jahre alter Entwicklungsstand zur Klonerkennung verwendet und die darauf folgende Analyse auf einer höheren Datenbasis durchgeführt.

Dieses Kapitel beschreibt die Auswertung der Studiendaten, die mittels einer Datenbankanwendung erfolgt. Zunächst wird der Aufbau der Datenbank beschrieben. Anschließend erfolgt die Darstellung der ERM-Diagramme. Abschließend werden die SQL-Abfragen für das Beantworten der Forschungsfragen beschrieben.

5.4.1 Studiendatenvorbereitung und ERM-Diagramme

Die Datenbasis für die Analyse erfolgt in Unterstützung einer Datenbank. Hierzu wurde die Software MS ACCESS 2013 verwendet. Es wurde zunächst eine neue Datenbank erstellt. Folgende Daten wurden anschließend als Tabellen in die Datenbank importiert:

1. Klonklassen aus ConQat (eindeutig, ohne Duplikate)
   Die Klonklassen werden aus den Ergebnissen des inkonsistenten Klonerkennungsansatzes aus ConQat ermittelt. Die Klonklassen aus Kapitel 5.3.1 werden in eine separate Tabelle gespeichert. Anschließend werden Duplikate entfernt. Daraus resultiert eine eindeutige Liste der Klonklassen.

2. Klondateien aus ConQat (eindeutig, ohne Duplikate)
   Die Klondateien werden wie die Klonklassen aus den ConQat-Ergebnissen ermittelt.

3. Basis_SQL
   Diese Daten sind die in Kapitel 5.3.3 ermittelten Daten aus FogBugz und Kiln. Diese enthält zu allen Klondateien die gesamte Revisionshistorie bis zum Zeitpunkt der durchgeführten Klonanalyse. Die Revisionshistorie besteht wiederum zum einen aus den Changesets mit den Informationen wie Benutzer, Datum, Summary und KlondateiID und zum anderen aus den Cases die Bugs und Features enthalten.

4. Beziehungstabelle
   Es wird eine Beziehungstabelle für die Datenauswertung erstellt. Der Grund hierfür ist, dass zwischen den Daten aus ConQat und den Daten aus FogBugz eine n:m-Beziehung besteht. Die Beziehungstabelle wird als Zwischentabelle verwendet, um die n:m-Beziehung zu beheben. Genau aus diesem Grund wurden auch die Klondateien und Klonklassen in separate Tabellen gespeichert und mit der Beziehungstabelle verknüpft. In Abbildung xxx ist die Beziehung der Tabellen durch ein ERM-Diagramm graphisch dargestellt.

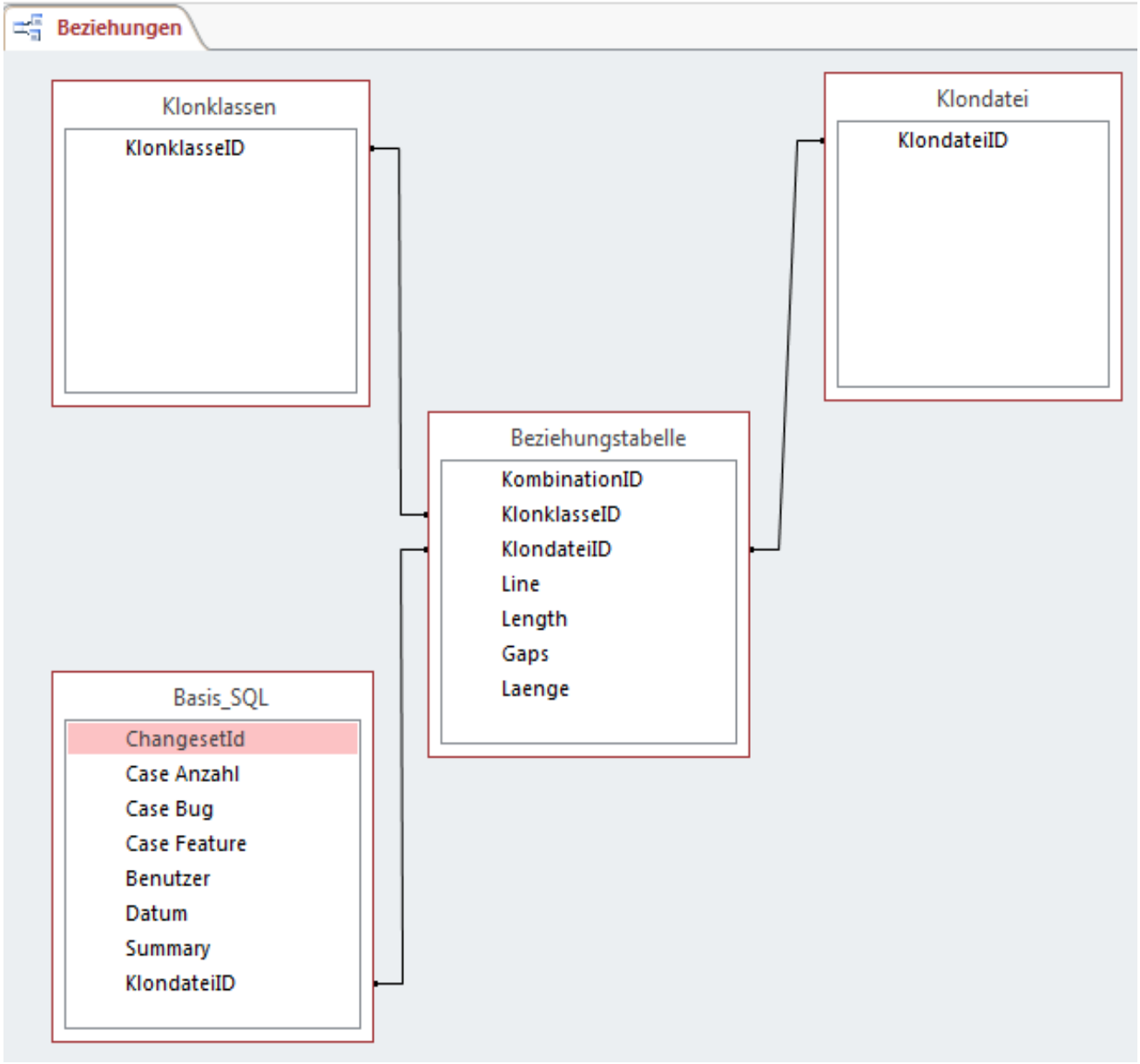

Abbildung 5.3: ERM-Diagramm für die Datenauswertung

5.4.2 SQL-Abfragen

Die Auswertung der Daten erfolgte durch die Ausführung von SQL-Abfragen. Die SQL-Abfragen, welche für das Beantworten der Forschungsfragen erforderlich waren, lauten wie folgt:

1. Als erstes wurden die inkonsistenten Klonklassen mit folgender SQL-Abfrage ermittelt:

```
SELECT [BEZIEHUNGSTABELLE].KLONKLASSEID,
COUNT ([BEZIEHUNGSTABELLE].KLONKLASSEID) AS ANZAHLKLONDATEI

FROM BEZIEHUNGSTABELLE

WHERE ([BEZIEHUNGSTABELLE].GAPS>0

GROUP BY [BEZIEHUNGSTABELLE].KLONKLASSEID;
```

Abbildung 5.4: SQL-Abfrage für inkonsistente Klonklassen

Hierbei steht „GAP“ für die Anzahl der Inkonsistenten in einer Klondatei. Mit GAP>0 werden lediglich die inkonsistenten Klonklassen aufgerufen. Die Abfrage liefert somit alle inkonsistenten Klonklassen mit der Anzahl der Klonstellen des Klons, die in dieser Klonklasse enthalten sind. Die Anzahl der inkonsistenten Klonklassen wird dann von MS ACCESS an der unteren Leiste ausgegeben. Das Ergebnis wird wie in Abbildung 5.5 ausgegeben.

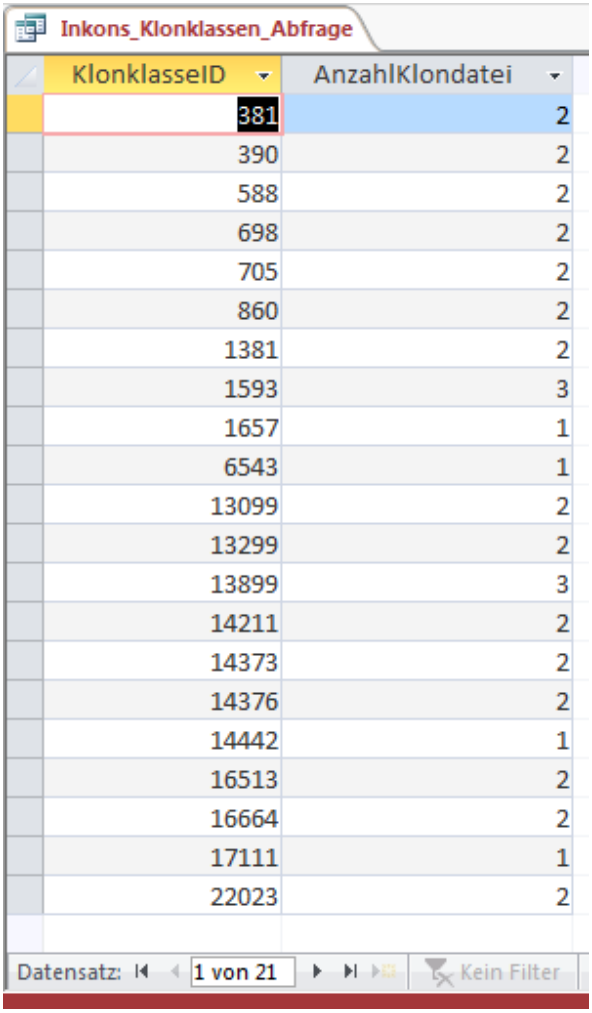

Abbildung 5.5: Abfrageresultat zu Inkonsistenten Klonklassen

2. Abfrage um die gesamte Revisionshistorie der Dateien aufzulisten:

```
SELECT [BEZIEHUNGSTABELLE].KLONKLASSEID,
BEZIEHUNGSTABELLE.KLONDATEIID, BEZIEHUNGSTABELLE.GAPS,
BEZIEHUNGSTABELLE.LINE, BEZIEHUNGSTABELLE.LENGTH,
BASIS_SQL.[CASE ANZAHL], BASIS_SQL.[CASE BUG],
BASIS_SQL.CHANGESETID

FROM ((KLONDATEI INNER JOIN BEZIEHUNGSTABELLE ON
KLONDATEI.KLONDATEIID = BEZIEHUNGSTABELLE.KLONDATEIID)
INNER JOIN KLONKLASSEN ON BEZIEHUNGSTABELLE.KLONKLASSEID =
KLONKLASSEN.KLONKLASSEID) INNER JOIN BASIS_SQL ON
BEZIEHUNGSTABELLE.KLONDATEIID = BASIS  SQL.KLONDATEIID;
```

Abbildung 5.6: Abfrage Revisionshistorie einer Datei

Hierbei wird mit Hilfe der Beziehungstabelle, zu allen Klondateien einer Klonklasse die gesamte Revisionshistorie mit den Daten aus der Basis_SQL ausgegeben. Das Ergebnis der Abfrage sieht wie in Abbildung 5.7 aus.

RevHistorieZuKlonklassen

KlonklasseID	KlondateiID	Gaps	Line	Length	Case Anzahl	Case Bug	ChangesetId
1705	D:\Mercurial\Projects\Discat X\Development\App	0	98	321	0	0	54a74b9a88ea
1705	D:\Mercurial\Projects\Discat X\Development\App	0	98	321	0	0	4b172dcaf884
705	D:\Mercurial\Projects\Discat X\Development\App	5	681	870	0	0	16ee333f2bd5
705	D:\Mercurial\Projects\Discat X\Development\App	5	681	870	0	0	c9e96d79c286
705	D:\Mercurial\Projects\Discat X\Development\App	5	902	1086	0	0	16ee333f2bd5
705	D:\Mercurial\Projects\Discat X\Development\App	5	902	1086	0	0	34410a06652e
705	D:\Mercurial\Projects\Discat X\Development\App	5	902	1086	0	0	c8aec68fb8bf
705	D:\Mercurial\Projects\Discat X\Development\App	5	902	1086	0	0	f2ba777a9f90
705	D:\Mercurial\Projects\Discat X\Development\App	5	902	1086	0	0	09179567d0e1
705	D:\Mercurial\Projects\Discat X\Development\App	5	902	1086	0	0	8c5b8207c5b4
705	D:\Mercurial\Projects\Discat X\Development\App	5	902	1086	0	0	904d2d3cc8e2
705	D:\Mercurial\Projects\Discat X\Development\App	5	902	1086	0	0	19097302ef5f
705	D:\Mercurial\Projects\Discat X\Development\App	5	902	1086	0	0	501471aab2b1
705	D:\Mercurial\Projects\Discat X\Development\App	5	902	1086	0	0	fe7e23da0fce
705	D:\Mercurial\Projects\Discat X\Development\App	5	902	1086	0	0	d9bb4e252ddc
6457	D:\Mercurial\Projects\Discat X\Development\App	0	569	695	2	2886	3834eb31fc88
6457	D:\Mercurial\Projects\Discat X\Development\App	0	569	695	0	0	d3c1cbd1ee07
6457	D:\Mercurial\Projects\Discat X\Development\App	0	569	695	0	0	7c19e9093f3f
6457	D:\Mercurial\Projects\Discat X\Development\App	0	569	695	0	0	94d72d2cae22
6457	D:\Mercurial\Projects\Discat X\Development\App	0	569	695	0	0	48b2b42a0687
6457	D:\Mercurial\Projects\Discat X\Development\App	0	569	695	0	0	1469a6a16445
6457	D:\Mercurial\Projects\Discat X\Development\App	0	569	695	0	0	a0cf2afadf6f

Datensatz: 1 von 1174 Kein Filter Suchen

Abbildung 5.7: Abfrageresultat zur Revisionshistorie zu Klonklassen

3. Abfrage, um Klonklassen zu ermitteln die einen Case-Bug-Eintrag haben:

```
SELECT [BEZIEHUNGSTABELLE].KLONKLASSEID,
COUNT ([BEZIEHUNGSTABELLE].KLONKLASSEID) AS
ANZAHLKLONKLASSEID

FROM (KLONKLASSEN INNER JOIN (KLONDATEI INNER JOIN
BEZIEHUNGSTABELLE ON KLONDATEI.KLONDATEIID =
[BEZIEHUNGSTABELLE].KLONDATEIID) ON KLONKLASSEN.KLONKLASSEID
= [BEZIEHUNGSTABELLE].KLONKLASSEID) INNER JOIN BASIS_SQL ON
[BEZIEHUNGSTABELLE].KLONDATEIID=BASIS_SQL.KLONDATEIID

WHERE (BASIS_SQL.[CASE BUG])>0
GROUP BY [BEZIEHUNGSTABELLE].KLONKLASSEID;
```

Abbildung 5.8: Abfrage für fehlerhafte Klonklassen

Durch diese Abfrage werden alle Klonklassen ermittelt, sowohl konsistente als auch inkonsistente, in denen ein Fehler behoben wurde. Diese Abfrage ist erforderlich für die Analyse der inkonsistenten Klone auf Fehler.

4. Für die Forschungsfrage 3 wird eine SQL-Abfrage für Case-Bug- Einträge in konsistenten Klonklassen erzeugt, da analysiert wird, ob fehlerhafte Typ-1-Klone im Laufe der Entwicklung zu fehlerfreien inkonsistenten Klonklassen werden. Die Abfrage lautet wie folgt:

```
SELECT [BEZIEHUNGSTABELLE].KLONKLASSEID,
COUNT ([BEZIEHUNGSTABELLE].KLONKLASSEID) AS
ANZAHLKLONKLASSEID

FROM (KLONKLASSEN INNER JOIN (KLONDATEI INNER JOIN
BEZIEHUNGSTABELLE ON KLONDATEI.KLONDATEIID =
[BEZIEHUNGSTABELLE].KLONDATEIID) ON KLONKLASSEN.KLONKLASSEID
= [BEZIEHUNGSTABELLE].KLONKLASSEID) INNER JOIN BASIS_SQL ON
[BEZIEHUNGSTABELLE].KLONDATEIID=BASIS_SQL.KLONDATEIID

WHERE (((BEZIEHUNGSTABELLE.GAPS)=0) AND ((BASIS_SQL.[CASE
BUG])>0))

GROUP BY [BEZIEHUNGSTABELLE].KLONKLASSEID;
```

Abbildung 5.9: Abfrage fehlerhafte inkonsistente Klonklassen

Nun stehen alle Daten zur weiteren Analyse bereit. Um nun den fehlerhaften Code in einer Version festzustellen, wird die Version r für eine Datei in der ein Fehler gefunden wurde festgehalten. Da das Repository eines Projektes sich auf einem älteren Stand befindet, werden der Revisionshistorie mit der Pull- Funktion in Mercurial neue Änderungen hinzugefügt, die jeweils durch eine Commit-Message in die Revisionshistorie aufgenommen werden und somit eine ChangesetID erhalten. In Tortoise besteht die Möglichkeit für jede Datei die Revisionshistorie einzusehen. Somit kann für jede Datei der inkonsistenten Klonklasse die gesamte Revisionshistorie betrachtet und die Entwicklung überprüft werden. Des Weiteren kann festgestellt werden, ob während der Entwicklung in den inkonsistenten Klondateien ein Fehler behoben wurde. Somit können Rückschlüsse über die Fehlerhaftigkeit der inkonsistenten Klonklassen gemacht werden.

Wie die Datenanalyse mit den aufbereiteten Daten durchgeführt wird, ist in Kapitel 5.5 detailliert beschrieben.

5.5 Datenanalyse

Zur Beantwortung der Forschungsfragen wurde in Anlehnung auf die Studien von Juergens et al. [6] und Rahman et al [10] ein Ansatz zur Datenanalyse entwickelt. Dieses Kapitel stellt die Datenanalyse für das Beantworten der verschiedenen Forschungsfragen vor.

In der Datenanalyse werden unterschiedliche Mengen von Klongruppen untersucht, um die Forschungsfragen zu beantworten. Die Unterschiede in den Definitionen der Klongruppenmengen basieren auf der Vielfältigkeit der Fragen. Die Hauptmenge enthält alle Klongruppen C, die zweite grundlegende Menge sind die inkonsistenten Klongruppen IC. Des Weiteren existieren die Mengen der erkannten Fehler in inkonsistenten Klonen mit BIC. Die unabhängigen Variablen in der Studie sind das Entwicklungsteam, die Programmiersprache, die funktionelle Domäne, das Alter und die Größe der Systeme.

Datenanalyse zur Forschungsfrage 1:

Die Forschungsfrage 1 untersucht die Existenz der inkonsistenten Klone auf den produktiven TWT Systemen, welche bereits im Einsatz sind. Die Analyse dieser Frage erfolgt wie in der Studie von Jürgens u.a. [6]. Zunächst wird auf den jeweiligen Systemen der Klonanalyseansatz für inkonsistente Klone mit ConQat durchgeführt. Die Ergebnisse werden manuell geprüft, um die Falschpositiven zu eliminieren. Anschließend wird das Verhältnis der inkonsistenten Klone zu den gesamten Klonen aus den Resultaten der SQL- Abfragen mit $|IC|/|C|$ berechnet. Abbildung 1 stellt die Mengen der beiden Klongruppen dar.

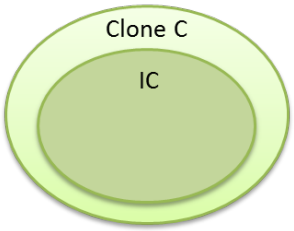

Abbildung 5.10: Menge der gesamten und inkonsistenten Klongruppen

Datenanalyse zur Forschungsfrage 2:

Für die Frage, ob inkonsistente Klone Indikatoren für Fehler sind, wird die Revisionshistorie, ähnlich wie in [10] zur Analyse des inkonsistent geklonten Codes mit einbezogen. Ein Versionsverwaltungssystem, in dieser Studie Mercurial, bietet in der Regel eine sehr umfangreiche Revisionshistorie an. Das Versionsverwaltungssystem enthält demnach die Versionsgeschichte einer Datei, bspw. Informationen über neu hinzugefügte, gelöschte und veränderte Dateien. Des Weiteren gibt sie Informationen über Entwickler, die Änderungen an einer Datei vorgenommen haben. Eine neue Version für eine Datei entsteht durch eine Commit-Message eines Entwicklers, der möglicherweise Änderungen an der Datei vorgenommen hat. Die Version erhält durch einen Commit an einer Datei, eine lokale Versionsnummer, die innerhalb des eigenen Repository gültig ist, sowie eine eindeutige Identifikationsnummer, die ChangesetID genannt wird und Gültigkeit im gesamten Repository hat. Des Weiteren wird neben dem Entwickler ebenfalls der Zeitpunkt des Committ-Eintrags einer Version angegeben.

Diese Studie untersucht die Auswirkungen des Klonens im gesamten Projekt-Lebenszyklus. Daher müssen alle Klone auf allen Versionen gefunden werden, die im Versionsverwaltungssystem committet wurden.

Der Grund, weshalb die Revisionshistorie eines Klons bei der Analyse betrachtet wird, ist die bestehende Möglichkeit, dass ein inkonsistent geklonter Code, zunächst nach dem Klonen keinen Fehler verursacht. Jedoch kann das geklonte Codefragment im Laufe der Entwicklung zu einem fehlerhaften Code werden. Die Abbildung 5.11 zeigt, wie ein Fehlereintrag in einem Issue-Tracking-System für einen inkonsistenten Klon verfolgt und ermittelt wird.

Beispielsweise soll in der Version 1 der Case 1: Impelement_xxx implementiert werden. Dieser wird entwickelt und anschließend committed. In der Version 2 wird festgestellt, dass dieses Codefragment in den Dateien a und b, in den Zeilen 10-20 (Datei a) und 30-50 (Datei b) inkonsistent geklont wurden, die noch keinen Fehler enthalten. Diese inkonsistenten Klone werden ebenfalls weiterentwickelt. In späteren Versionen wird in dem inkonsistenten Klon a in den Zeilen 10-15 ein Fehler erkannt, der in dem Issue-Tracking-System behoben werden muss. Über die Case-Nummer, welche eindeutig für die gesamte Entwicklung ist, kann der Fehler im Issue-Tracking-System gefunden und

gefixt werden. Somit wurde festgelegt, welcher inkonsistenter Code im Laufe der Entwicklungshistorie zu einem inkonsistenten Code wird, der einen Fehler enthält.

Im Allgemeinen kann also ermittelt werden, welcher inkonsistenter Klon, der einen Bezug zu einem Issue-Tracking-System hat, einen Zusammenhang mit einem Fehler hat. Hierzu werden die Ergebnisse der SQL-Abfragen zu allen inkonsistenten Klonklassen als Datenbasis genommen. Für jede Klondatei in einer inkonsistenten Klonklasse wird der beschriebene Fehleranalyseansatz durchgeführt. Aus dieser Vorgehensweise kann ermittelt werden, ob und wann Klone verändert wurden bzw. Fehler im Laufe der Entwicklung verursacht haben.

Während diesem Analyseprozess können viele weitere wichtige Fragen wie die Forschungsfragen 2.1 – 2.4 analysiert werden. Vor allem die Forschungsfrage 2.2, ob Fehler in inkonsistenten Klonen an allen Codefragmenten einer Klongruppe behoben werden, lässt sich anhand dieser Analyse sehr gut beschreiben und beantworten. Da die gesamte Revisionshistorie der Klondateien einer Klonklasse betrachtet wird, kann sehr gut beobachtet werden, ob Fehler in einer Klonklasse an allen Klonstellen behoben werden und ob die Fehlerbehebung zeitgleich durchgeführt wird oder sogar überhaupt nicht betrachtet wird.

Mit dem dargestellten Analyseverfahren kann die zeitliche Spanne zwischen den Klonen ermittelt werden. Wenn eine große zeitliche Spanne zwischen den Klonen liegt, die keinen Fehler enthalten, kann der Klon als ein robuster Klon eingestuft werden. Klone die zeitgleich modifiziert werden, sei es eine Fehlerbehebung oder andere Änderungen, werden als bewusstes Klonen kategorisiert. Im Gegensatz dazu definieren sich Klone, für welche die Fehlerbehebung nicht in allen Klondateien einer Klonklasse durchgeführt wird, als fehlerhafte Klonklassen und demnach als unbewusstes Klonen.

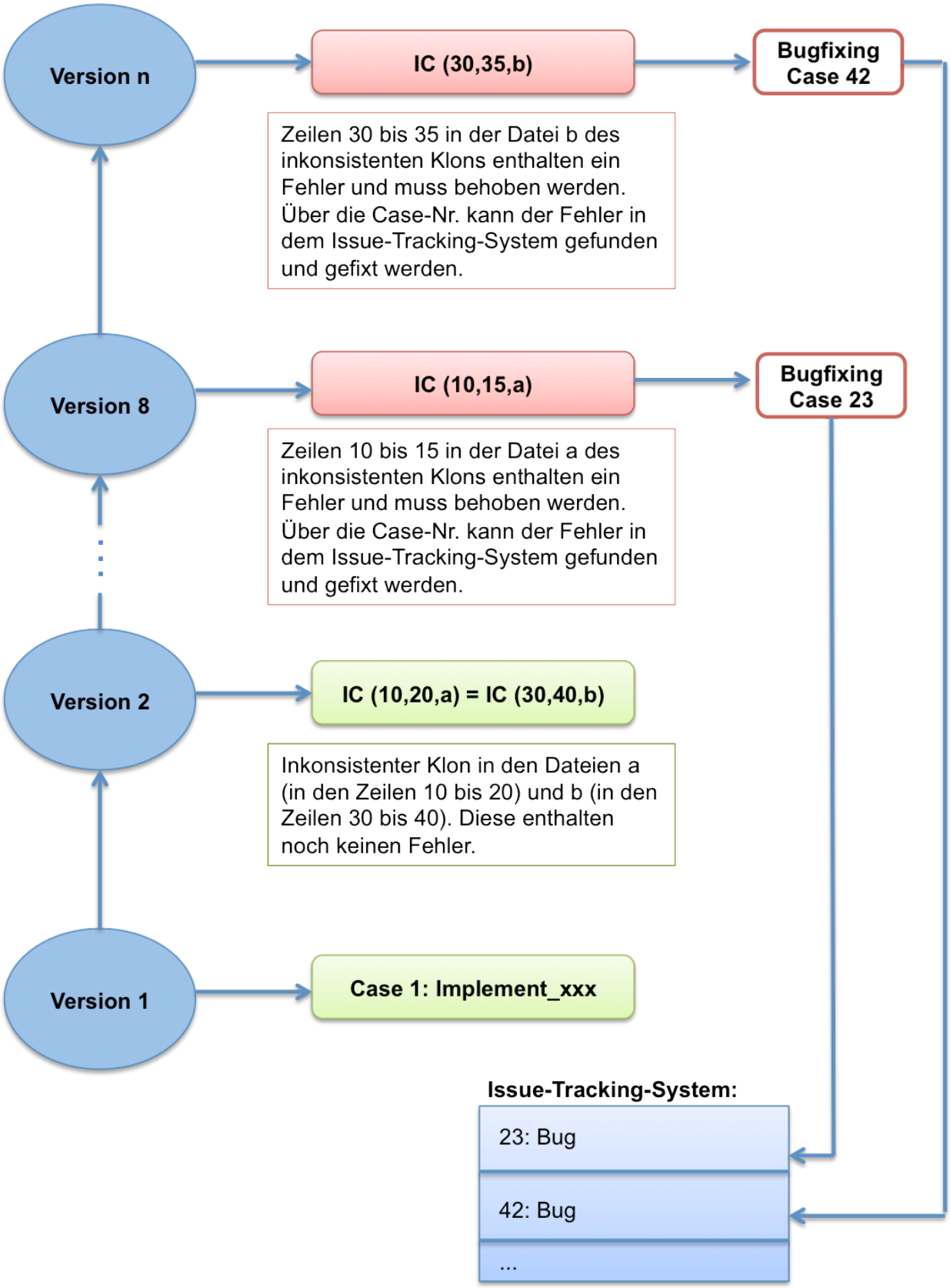

Abbildung 5.11: *Prozess zur Verfolgung eines inkonsistenten Klons über die Revisionshistorie*

Datenanalyse zur Forschungsfrage 2.1

Die Frage, ob die große Mehrheit der inkonsistenten Klone als Fehler erfasst sind, kann im Grunde wie in der *Datenanalyse zur Forschungsfrage 2* ermittelt werden. Für diese Frage werden zunächst die erkannten inkonsistenten Klone in dem Issue-Tracking-System FogBugz und Kiln auf einen Bug-Eintrag geprüft. Dadurch entsteht die Menge der inkonsistenten Klone, die einen Fehler enthalten und auch erkannt wurden.

Zunächst muss aber eine Definition für einen fehlerhaften Code festgelegt werden. Im Grunde kann eine Reihe von Codezeilen, die einen Fehler verursacht haben, als fehlerhafter Code bezeichnet werden. Jedoch ist es schwierig, das tatsächlich schuldhafte Codefragment zu finden. Diese Studie nutzt die Definition eines fehlerhaften Codes aus der Studie von Rahman et al. [10], die besagt, dass ein *buggy code* eine Reihe von Codezeilen ist, die geändert wurden, um einen Fehler zu beheben.

Nachdem der buggy Code gefunden wurde, wird geprüft ob die Codezeilen in denen Änderungen vorgenommen wurden, mit den Codezeilen, in denen vom Klonerkennungstool ConQat ein inkonsistenter Klon gefunden wurde, übereinstimmen. Wenn dies der Fall ist, kann davon ausgegangen werden, dass der inkonsistente Klon im Entwicklungszyklus Fehler verursacht hat.

Das Verhältnis der erkannten Fehler in inkonsistenten Klonen (BIC) wird mit $|BIC|/|IC|$ berechnet.

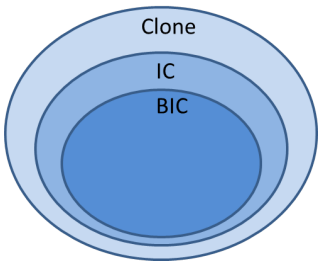

Abbildung 5.12: Menge der erkannten Fehler in Inkonsistenzen im Issue-Tracking-System

Datenanalyse zur Forschungsfrage 2.2

Auf die vorherige Frage aufbauend lässt sich die Frage, ob Fehler an einem Codefragment konsistent an allen inkonsistent geklonten Codes einer Klongruppe behoben wird, durch eine weitere Analyse beantworten. Das Resultat dieser Frage dient als Grundlage bei der Argumentation der Auswirkungen der inkonsistenten Klone.

Die Ergebnisse des Klonerkennungstool ConQat bestehen typischerweise aus einer Reihe von Klongruppen. Jede der Klongruppen enthält Codefragmente, die aneinander ähnlich sind und sich in derselben oder auch in verschiedenen Dateien befinden

können. Demnach enthalten die Klongruppen Informationen über die Dateien in denen sich das geklonte Codefragment befindet, Informationen zu Anfangszeilen und Endzeilen des Klons, die Länge des Klons sowie die maximal mögliche Gap-Anzahl.

Nach der Festlegung der Menge der erkannten Fehler in den Inkonsistenzen (BIC), wird geprüft, ob der Fehler in allen Codefragmenten einer Klongruppe, die diesen Fehlereintrag haben, behoben wurde. Fehler werden in einem Issue-Tracking-System entdeckt und aufgezeichnet und im Laufe der Entwicklung von den Entwicklern behoben. Ein behobener Fehler wird auf eine bestimmte Version im Versionsverwaltungssystem verbunden. Daher wird der Korrekturvorgang für die gesamte Revisionshistorie einer Datei geprüft. Wenn ein Fehler in allen geklonten Codes einer Klongruppe behoben wurde, stellt der Fehler kein Risiko dar. Andernfalls wird das inkonsistente Klonen als risikobehaftet eingestuft.

Wichtig: Während der Analyse wird untersucht, ob die Korrektur der erkannten Fehler zeitgleich oder mit Zeitverzug oder überhaupt nicht erfolgt. Aus diesem Ergebnis wird erkannt, ob Entwickler bewusst klonen.

Ermittelt wird die Menge der inkonsistenten Klone, die an allen Codefragmenten einer Klongruppe keinen Fehler mehr enthalten (KF). Das Verhältnis der Menge der inkonsistenten Klone, welche keinen Fehler mehr enthalten (KF) lässt sich mit |KF|/|BIC| berechnen.

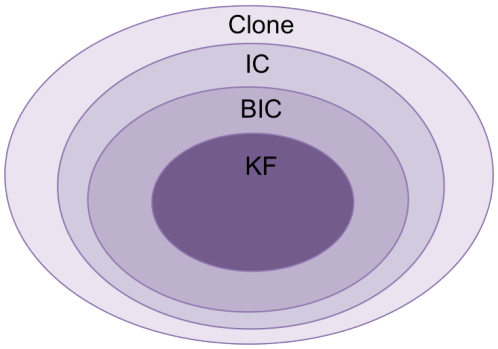

Abbildung 5.13: Menge der inkonsistenten Klongruppen, bei denen an jedem Klon einer Klongruppe der Fehler behoben wurde

Hypothese:

Die Antwort auf diese Frage ist das Hauptergebnis der Studie, weil es die Auswirkungen eines inkonsistenten Klons begründet. Daher wird hieraus eine Hypothese abgeleitet:

Das Verhältnis der Fehler in den Inkonsistenten ist höher als die Fehler in den restlichen Klonen im System.

Datenanalyse zur Forschungsfrage 2.3

Es wichtig die Fehler zu kategorisieren, um ein besseres Verständnis für den Grund der Fehler zu ermitteln, bzw. festzustellen unter welchen Kontextbedingungen ein Fehler gegeben ist. Hierzu wird die gesamte Revisionshistorie eines Inkonsistenten Klons analysiert und festgestellt, ob ein Klon im Entwicklungszyklus modifiziert wurde und ob sich Fehler durch die fehlenden Klonkenntnisse der Entwickler einschleichen. Es ist ebenfalls interessant festzustellen, weshalb inkonsistente Klonklassen trotz langer Entwicklungshistorie keinen Fehler im gesamten Entwicklungszyklus verursachen und erhält aus diesem Grund eine Untersuchung.

Datenanalyse zur Forschungsfrage 2.4

Es wichtig die Fehler zu kategorisieren um ein besseres Verständnis für den Grund der Fehler zu ermitteln, bzw. festzustellen unter welchen Kontextbedingungen ein Fehler gegeben ist. Hierzu wird die gesamte Revisionshistorie eines Inkonsistenten Klons analysiert und festgestellt, ob ein Klon im Entwicklungszyklus modifiziert wurde und ob sich Fehler durch die fehlenden Klonkenntnisse der Entwickler einschleichen. Es ist ebenfalls interessant festzustellen, weshalb inkonsistente Klonklassen trotz langer Entwicklungshistorie keinen Fehler im gesamten Entwicklungszyklus versuchen und erhält aus diesem Grund eine Untersuchung.

Datenanalyse zur Forschungsfrage 3:

Letztlich werden die Auswirkungen der inkonsistenten Klone nicht in negativer sondern in positiver Hinsicht betrachtet. Bisher wurde analysiert, ob die inkonsistenten Klone Fehler verursachen bzw. ob die Fehler in den inkonsistenten an allen Klonstellen behoben wurden. Nun soll hingegen analysiert werden, ob Typ-1-Klone mit einem Fehler durch das Modifizieren für die Fehlerbehebung zu einem Typ-3-Klon ohne Fehler wurden.

Hierzu erfolgt zunächst die Fehleranalyse wie in der Datenanalyse 2.1 für Typ-1-Klone. Als Datenbasis dient das Ergebnis der SQL-Abfrage für exakte Klonklassen die einen Fehlereintrag haben. Klone mit einem Fehlereintrag im Issue-Tracking-System, werden weiterhin zur Analyse unterzogen. Im zweiten Analyseschritt erfolgt das Prüfen des fehlerhaften Typ-1-Klon Codefragments. Es wird geprüft, ob dieser weiterentwickelt wurde und in der Revisionshistorie durch einen „bug-fixing-Eintrag“ behoben wurde. Des Weiteren wird in der Revisionshistorie geprüft, ob es sich nun beim fehlerfreien Codefragment um einen inkonsistenten Klon handelt. Dieser kann in den Ergebnissen des Klonanalysewerkzeugs ConQat geprüft werden. Um sicherzustellen, dass der inkonsistente Klon tatsächlich nur einen positiven Beitrag im Entwicklungszyklus geleistet hat, erfolgt als letzter Schritt das Prüfen des inkonsistenten Klons auf Fehler. Wenn nämlich der inkonsistente Klon keinen Fehler enthält, bzw. keinen weiteren

Fehler in der Revisionshistorie verursacht hat, handelt es sich um einen positiven inkonsistenten Klon.

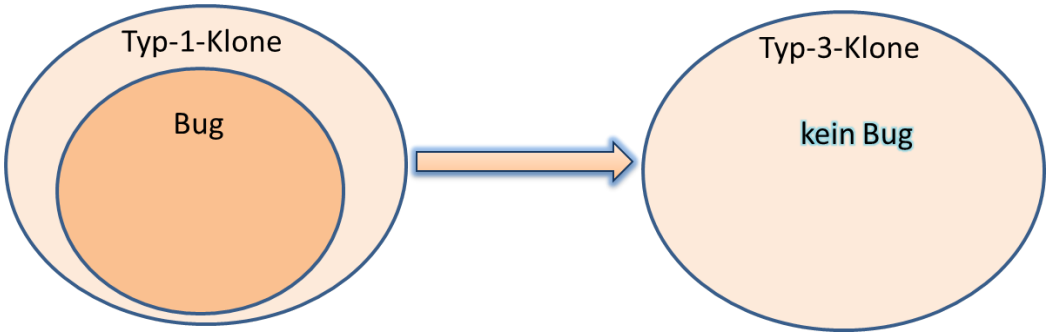

Abbildung 5.14: Typ-1-Klone die einen Fehler enthalten werden zu Typ-3-Klone ohne Fehler

6 Ergebnisse

Die Ergebnisse des Studiendesigns sind für das Beantworten der jeweiligen Forschungsfragen präzise in Tabelle 2 erfasst. Die Tabelle 5.1 in Kapitel 5.1 enthält Informationen zu den Studienobjekten. Unter anderem listet sie die Größe der Projekte in Codezeilen auf. In näherer Betrachtung der Größe der Codezeilen und der Anzahl der Klonergebnissen aus ConQat wird ersichtlich, dass je größer ein Projekt ist, desto höher die Anzahl der Klone sind. Projekt C ist ungefähr doppelt so groß wie Projekt A. Dies spiegelt sich ebenfalls in den Klonergebnissen der Projekte wieder, da Projekt C im Vergleich zu Projekt A doppelt so viele Klonklassen enthält.

Tabelle 6.1: Zusammenfassung der Studienergebnisse

Projekt	**DISCAT**	**FCAD**	**CO2MO**
Klonklassen |C|	37	88	82
Inkonsistente Klonklassen |IC|	21	21	65
Exakte Klonklassen	16	67	17
Inkonsistente Klonstellen |IS|	46	43	146
Modifizierte inkonsistente Klonstellen |MIS|	24	19	67
Verhältnis der modifizierten Klonstellen (in %) |MIS|/|IS|	0,52	0,44	0,45
Zeitgleich modifizierte inkonsistente Klonstellen (in %) |ZMIS|	14	17	62
Verhältnis zeitgleich modifizierte inkonsistente Klonstellen (in %) |ZMIS|/|MIS|	0,58	0,89	0,92
Fehlerhafte Klonklassen |FK|	16	5	37
Fehlerhafte Inkonsistente Klonklassen |BIC|	7	1	2
Verhältnis fehlerhafter inkonsistenter Klonklassen (in %)|BIC|/ |IC|	0,33	0,05	0,03
Fehlerbehobene Inkonsistente Klonklassen |KF|	4	1	0
FF 1 |IC|/|C|(in %)	0,56	0,23	0,79

FF 2.1 ⎪BIC⎪/⎪IC⎪(in %)	0,33	0,05	0,03
FF 2.2 |KF| / |BIC| (in %)	0,57	1,0	0
Durchschnittliche Inkonsistente Klonlänge	60	62	78
Durchschnittliche Fehlerhafte Inkonsistente Klonlänge	50	39	83
Hypothese ⎪BIC⎪/ ⎪FK⎪ (in %)	0,43	0,2	0,05

Aus den Ergebnissen lässt sich schließen, dass die Klonklassen der Projekte, bis auf Projekt B, über die Hälfte etwa 56% - 79% inkonsistent sind. Der Grund weshalb das Projekt B im Verhältnis zu den beiden anderen Projekten weniger inkonsistente Klonklassen enthält ist, dass Codefragmente aus den firmeninternen Bibliotheken exakt geklont wurden, um bestimmte Funktionalitäten wiederzuverwenden. Schlussfolgernd kann die Forschungsfrage 1, ob die Systeme größtenteils inkonsistente Klone enthalten, positiv beantwortet werden.

Aus diesen inkonsistenten Klonen wurde ungefähr die Hälfte 44%-55% weiterentwickelt. Es wurden Codezeilen entfernt, hinzugefügt oder bestehende Klonzeilen, sei es in den Literalen oder Bezeichnern, modifiziert. Daraus resultiert, dass bestehende Klone eine Unterstützung für die Entwickler im Entwicklungszyklus der Programmierer sind. Um zu untersuchen, ob Entwickler überhaupt bewusst Klonen wurde analysiert, ob die Modifizierung an allen Klonstellen einer Klonklasse zeitgleich durchgeführt wurde. Aus den Ergebnissen lässt sich schließen, dass in allen Projekten die Entwickler über fast alle Klonstellen, 58%-92%, informiert sind und diese zeitgleich bei Bedarf modifizieren. Das spiegelt sich auch in den Fehlerergebnissen der inkonsistenten Klone wieder. Infolge des bewussten Klonens sind lediglich 3%-33% der inkonsistenten Klonklassen fehlerbehaftet. Daraus resultierend kann die Forschungsfrage 2, ob inkonsistente Klone Indikatoren für Fehler sein können negativ beantwortet werden, da die Inkonsistenten beim bewussten Klonen verhältnismäßig eine geringe Gefahr für ein Softwaresystem darstellen. Folgernd beweisen die Ergebnisse dieser Studie die Analysewerte der Studie von Rahman u.a. [10], die besagen, dass die aus den Versionskontrollsystemen gewonnen Daten, die auch erfasste Fehler in einem System kennzeichnen, keinen bemerkenswerten Zusammenhang zwischen Klonen und Fehler darstellen.

Da sehr wenige inkonsistente Klonklassen einen Bezug zu einem Fehler haben, lautet die Antwort für die Forschungsfrage 2.1, dass sich lediglich ein sehr geringer Anteil der inkonsistenten Klonklassen, mit 3-33%, als fehlerbehaftet kategorisieren lässt.

Für die Forschungsfrage 2.2 wurde basierend auf den fehlerbehafteten inkonsistenten Klonklassen geprüft, ob die Fehler in einer Klonklasse an allen Klonstellen behoben wurden. Aus den Ergebnissen lässt sich schließen, dass bis auf das Projekt C die zeitgleich modifizierten inkonsistenten Klonstellen und fehlerbehobenen Klonstellen

einen Zusammenhang haben. Bei bewussterem Klonen, also zeitgleich modifizierte Klone, steigt der prozentuale Anteil der an allen Stellen behobenen Fehler in einer Klonklasse mit dem zeitgleich modifizierten Inkonsistenten. In Anbetracht, dass das Projekt C ein Ausreißer ist, kann diese Frage mit 57%-100% der fehlerbehobenen Klonstellen, positiv beantwortet werden.

Die Hypothese der Studie lautet zur Erinnerung wie folgt: „Das Verhältnis der Fehler in den Inkonsistenten ist höher als die Fehler in den restlichen Klonen im System“. Im Durchschnitt enthalten die inkonsistenten Klonklassen, mit 5%-43%, weniger fehlerhafte Klonstellen als die restlichen Klonstellen im System. Daraus kristallisiert sich, dass die Inkonsistenten im Vergleich zu exakten Klonen mindestens weniger als die Hälfte einen Fehler verursachen.

Um die Kontextbedingungen der Fehler in den Inkonsistenten zu bestimmen, wird die Forschungsfrage 2.3 analysiert und um Resultate über den Zusammenhang der Klonlänge der Inkonsistenten und der Fehlerhäufigkeit zu finden. Für diese Analyse ist Projekt C erneut ein Ausreißer. Die Klonlänge der fehlerhaften Inkonsistenten Klonklassen ist kleiner als die durchschnittliche Klonlänge. Die vergleichsweise relativ größeren Klone, bis zu bspw. 206 Zeilen, enthalten dagegen keine Fehler. Daraus resultiert das Ergebnis, dass sich Fehler in kleineren Inkonsistenten Klonen befinden. Für Projekt C gilt das ebenfalls für die eine fehlerbehaftete Klonstelle. Daher kann dieser Beschluss gezogen werden. Folgernd kann die Annahme, dass je größer ein inkonsistenter Klon ist, desto größer die Fehlerwahrscheinlichkeit ist, nicht wiederlegt werden.

Um die Forschungsfrage 2.4 zu beurteilen, und zwar unter welchen Kontextbedingungen Fehler in Inkonsistenten gegeben sind und Hinweise für die Fehlerträchtigkeit von Klonen zu finden, erfolgten zwei wesentliche Kategorisierungen während der Analyse der Inkonsistenten. Die Klone wurden in bewusste und robuste Klone gegliedert. Das bewusste Klonen bezeichnet das sorgfältige Klonverhältnis eines Entwicklers. Hierunter fallen wie obig genannt Klone, die zeitgleich modifiziert werden und auch nach dem Modifizieren keinen Fehler verursachen.

Die robusten Klone hingegen umfassen inkonsistente Klonklassen, die keinen Fehlereintrag in der gesamten Revisionshistorie haben. Bei der Analyse wurde festgestellt, dass die robusten Klone, auch bis zu 4 Jahren nach dem Klonen weiterentwickelt werden und trotz dessen keinen Fehler erzeugen. Daher werden Klone, die trotz langer Revisionshistorie keinen Fehler darstellen als Klone bezeichnet, die bewusst geklont und modifiziert werden. Ein besonders wichtiger Punkt bei den robusten Klonen ist, dass die Klone größtenteils aus einer Eclipse-Bibliothek, zu 60,74%, und firmeninternen Bibliothek, zu 70%, stammen. Daraus resultiert, dass insgesamt 76% der inkonsistenten Klone für eine Fehlerbehebung nicht modifiziert werden, da ihr ursprünglicher Quelltext aus einer Bibliothek stammt. Daraus kann abgeleitet werden, dass das Klonen aus Bibliotheken eine geringere Gefahr darstellt.

Zuletzt erfolgte die Untersuchung der inkonsistenten Klone auf einen positiven Einfluss auf die Softwareentwicklung. Für die Forschungsfrage 3 wurde untersucht, ob Typ-1-Klone die einen Fehler enthalten, im Laufe der Entwicklungshistorie modifiziert und inkonsistent wurden, um den Fehler zu beheben. Die Analyse ergab, dass sich alle exakten Klone die einen Fehler enthalten zu einem inkonsistenten Klon ohne Fehler entwickelten. Des Weiteren erfolgte die Fehlerbehebung der Typ-1-Klone zeitgleich an allen Klonstellen. Das hängt mit dem bewussten Klonverhältnis der Entwickler zusammen.

Zusammenfassend kann also der Rückschluss gezogen werden, dass bei bewusstem Klonen, also dem zeitgleichen Modifizieren der Klone, die Klone robust sind und somit keine Fehler verursachen. Weiterhin beweist die Studie, dass das Klonen aus Bibliotheken weniger Fehler verursacht. Außerdem sind wie anhand anderer Studien angenommen inkonsistente Klone nicht die Verursacher für Fehler, sondern auch das Mittel zur Fehlerbehebung in den exakten Klonen.

7 Gefahren für die Gültigkeit

In diesem Kapitel werden die Gefahren und die Gültigkeit der Studie bewertet und ausführlich dargestellt.

7.1 Konstruktvalidität

Es wurde die Entwicklungshistorie der Systeme analysiert, um festzustellen, ob die Inkonsistenten wirklich durch Änderungen an einem System entstanden sind. Das Problem besteht jedoch darin, dass Codefragmente durch Kopieren und Modifizieren in einem einzigen Commit eingefügt wurden. Daher wurde die gesamte Revisionshistorie der Industriesysteme manuell bearbeitet, um alle Änderungen an einem Codefragment zu prüfen.

Die Cases, die im Issue-Tracking-System einen Bugeintrag haben, wurden für jedes Projekt als Basis für fehlerhafte Codefragmente verwendet. Ein Case erhält durch den manuellen Vorgang eines Entwicklers einen Bugeintrag. Da das Verhalten der Entwickler nicht immer ordnungsgemäß und vollständig sein muss, kann die in dieser Studie verwendete Datenbasis für Fehler nicht die komplette Menge an Fehler enthalten. Weiterhin ist es möglich, dass es sich bei Cases, die einen Feature-Eintrag haben, um fehlerhafte Codezeilen handelt. Um dieser Gefahr entgegenzuwirken, wurden die Codezeilen für jedes Case geprüft, ob die Änderungen in den inkonsistenten Codezeilen vorgenommen wurden und auf deren Robustheit geprüft, da es keine Möglichkeit auf Fehlerprüfung gab.

Das Prüfen aller inkonsistenten Klonzeilen auf Fehler ist praktisch nicht möglich gewesen, da es eine hohe Entwicklerzeit und Bereitschaft für die Prüfung erfordern würde. Der Nutzen des Entwicklers ist in diesem Fall zu gering, so dass das praktisch nicht umsetzbar ist.

Es ist durchaus möglich, dass es sich bei einigen der Änderungen in der Revisionshistorie einer Klondatei, die einen Bugeintrag haben, tatsächlich nicht um fehlerhafte Codezeilen handelt. Um diesem Problem entgegenzuwirken, wurde ein hoher manueller Aufwand aufgebracht, um die falschen Resultate zu eliminieren.

7.2 Interne Gültigkeit

Die aus ConQat analysierten inkonsistenten Klone wurden auf Falsch-Positive geprüft. Die restlichen Systeme sollten ebenfalls auf inkonsistente Klone und Fehler geprüft werden. Da jedoch die Entwicklerzeit dafür fehlte, wurden die restlichen Klone wie aus ConQat nach dem Prüfen auf Falsch-Positive als Datenbasis genommen. Das führt

dazu, dass die Menge der inkonsistenten Klone sowie die damit verbundenen Fehler, einer kleineren Menge entsprechen, als es möglicherweise tatsächlich ist. Somit kann es zu leichten Abweichungen in der Forschungsfrage 1 und damit verbunden auch in den Werten in der Forschungsfrage 2 führen.

Die Konfigurationsparameter für den Klonerkennungstool ConQat wurde mit einem liberalem und konservativem Ansatz durchgeführt. Der liberale Ansatz liefert höhere Klonergebnisse. Trotzdessen wurde die Klonerkennung mit konservativen Parametern ausgeführt, da bei einem liberalen Ansatz eine höhere Entwicklerbereitschaft erforderlich ist und diese nicht zur Verfügung stand. Bei dem konservativen Ansatz haben wir aber eine kleinere Klonbasis, welche die Studie unterstützt, um präzisere Auswertungen zu machen.

7.3 Externe Gültigkeit

Die Softwaresysteme wurden auf ein neues Issue-Tracking-System umgesetzt. Frühere Codezustände waren nicht ersichtlich und konnten daher nicht zur Analyse unterzogen werden. Demnach ist die Menge der Systeme nicht vollständig repräsentativ. Die Studie wurde lediglich auf drei relativ jungen Industriesystemen durchgeführt, die folglich auch über kleinere Revisionshistorien verfügen. Ein älteres System mit einer längeren Revisonshistorie würde die Resultate dieser Studie konsolidieren. Obwohl alle Systeme in Java geschrieben sind und unterschiedliche Funktionalitäten ausführen, sind die Ergebnisse in den verschiedenen Projekten ziemlich konsistent, nämlich dass es keinen erhöhten Zusammenhang zwischen inkonsistenten Klonen und Fehlern gibt und dass sogar das Klonen aus Eclipse- und firmeninternen Bibliotheken eine Unterstützung für die Entwickler ist.

8 Zusammenfassung und Ausblick

Im Rahmen dieser Diplomarbeit wurde der Zusammenhang der inkonsistenten Klone und Fehler in Softwaresystemen untersucht, um Hinweise für die Fehlerträchtigkeit der Klone zu finden und festzustellen unter welchen Kontextbedingungen sie gegeben sind. Dabei untersucht die Arbeit auf empirischer Basis drei Industriesysteme, in den inkonsistenten Klonfragmenten über die gesamte Revisionshistorie auf Fehler. Um den Zusammenhang der Fehler in den Inkonsistenten zu analysieren, wurde ein Studiendesign entwickelt. Das Studiendesign enthält drei Hauptforschungsfragen, welche als Grundlage für die Analyseergebnisse dienen. Die Klonerkennung erfolgt auf dem Klonerkennungswerkzeug ConQat. Die Analyse wurde auf den, von den Falsch-Positiven bereinigten, Klonergebnissen durchgeführt. Es konnte festgestellt werden, dass die Hälfte der Klonklassen inkonsistent ist. Die Revisionshistorie gab Informationen über die Fehlereinträge in einem Klonfragment. Die Auswertung der Daten wurde mit Hilfe von Datenbanken durchgeführt. Der Einsatz einer Datenbank hat den Vorteil, dass der manuelle Aufwand für die Suche nach den Informationen in den Klonklassen, wie Klondateien, Anfangszeile, Endzeile, Gaps, nicht mehr besteht. Die Datenanalyse ergab, dass die Entwickler größtenteils bewusst Klonen, da 58-92% der Klonklassen an allen Klonstellen gleichzeitig modifiziert wurden. Demnach sind lediglich 3%-33% der inkonsistenten Klonklassen fehlerbehaftet, welches nur einen geringen Anteil darstellt und verdeutlicht, dass die inkonsistenten Klone keinen großen Einfluss auf die Fehlerrate eines Systems haben. Des Weiteren beweist die Studie, dass Klone auch nach 4 Jahren Klonzeit über die gesamte Revisionshistorie keinen Fehler verursachen. Ungefähr drei Viertel der inkonsistenten Klone stammen aus Eclipse- bzw. firmeninternen Bibliotheken, die trotz Modifizierungen zu 76% keinen Zusammenhang zu Fehlern haben. Dies führt zum Beschluss, dass das Klonen aus Bibliotheken eine geringere Gefahr darstellt. Die Studie untersuchte den Einfluss der Länge der inkonsistenten Klone auf die Fehlerhäufigkeit. Die Ergebnisse verdeutlichen, dass Fehler überwiegend in kleinen Inkonsistenten Codefragmenten enthalten sind. Zusätzlich veranschaulicht die Studie, dass inkonsistente Klone nicht wie angenommen Fehler verursachen, sondern auch durch Fehlerbehebungen entstehen. Durch das Modifizieren der Typ-1-Klone zur Fehlerbehebung entstehen nämlich Typ-3-Klone, die über die gesamte Revisionshistorie weiter geklont und modifiziert werden und trotz dessen keinen Fehler verursachen. In Kapitel 7 sind die Gefahren für die Gültigkeit der Studie erfasst, die alle Gefahren und Hürden der Arbeit darstellt, welche zu Abweichungen in den Analysewerten führen können.

8.1 Ausblick

Die Studie nutzte für die Analyse drei relativ junge Industriesysteme. Für bessere und umfangreichere Studienergebnisse sollte eine größere Datenbasis mit mehr Projekten vorliegen und die Studie auf diesen Projekten repliziert werden. Die Datenbasis der Analyse setzt sich aus dem Klonerkennungstool ConQat, aus dem Versionskontrollsystem Mercurial und dem Issue-Tracking-System FogBugz zusammen. Die Datenbasis wurde mit einem hohen manuellen Aufwand aufbereitet. Um die Studie auf einer umfangreicheren Datengrundlage durchzuführen, ist die Automatisierung für die Aufbereitung der Datenbasis erforderlich, siehe Abbildung 8.1. Die Voraussetzung für die Automatisierung ist das Entwickeln eines Tools, welches die erforderlichen Daten zu den Klonergebnissen aus ConQat aus den unterschiedlichen Systemen bezieht und diese als Analysegrundlage aufbereitet und anschließend listet.

Das Tool erhält die Klonergebnisse aus dem Klonerkennungswerkzeug, in unserem Fall aus ConQat. Nach dem Prüfen auf Falsch-Positive werden die Klondateien in Form von Identifikationsschlüssel für die Suche im Issue-Tracking-System aufbereitet. Des Weiteren müssen die Klondateien Informationen über die Anfangszeile, Endzeile und Gaps enthalten. Aus diesen Daten ist nun ersichtlich, ob ein Klon konsistent oder inkonsistent ist und in welchen Zeilen sich der Klon befindet. Anschließend werden die aufbereiteten Daten in das Tool importiert. Das Tool bezieht zum einen Informationen aus einem Versionskontrollsystem und zum anderen aus einem Issue-Tracking-System. Aus dem Versionskontrollsystem werden zu jeder Klondatei die Informationen zu einem Commit, wie ChangesetID, Datum, Zeit, Benutzer, Beschreibung gelistet. Im nächsten Schritt wird zu jeder Klondatei im Issue-Tracking-System die gesamte Revisionshistorie geprüft. Es erfolgt die Analyse, ob die inkonsistenten Klone sich im Laufe der Revisionshistorie verändert haben und in diesem Zusammenhang Fehler entstanden sind. Des Weiteren soll automatisiert geprüft werden, ob die Änderungen an allen Klonstellen einer Klonklasse zeitgleich erfolgt. Für die Aufbereitung dieses Tools können Schnittstellen für Kiln und FogBugz verwendet werden, solang die Systeme auf diesen Systemen gepflegt werden.

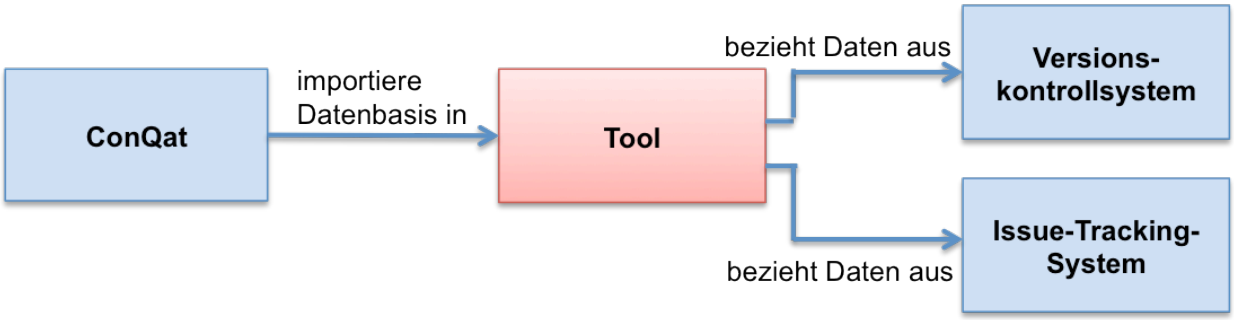

Abbildung 8.1: Tool zur Automatisierung

9 Literatur

[1] G. Alkhatib: The maintenance problem of application software: An empirical analysis, Journal of Software Maintenance: Research and Practice, 4(2), S. 83-104, DOI: 10.1002/smr.4360040203, 1992.

[2] M. Fowler, K. Beck, J. Brant, W. Opdyke, D. Roberts: Refactoring: Improving the Design of Existing Code. 1. Auflage, Addison-Wesley Professional, Massachusetts, 1999, ISBN 0-201-48567-2.

[3] M. V. Mäntylä, C. Lassenius: Subjective evaluation of software evolvability using code smells: An empirical study, Empirical Software Engineering, 11(3), S. 395-431, 2006. Siehe: http://link.springer.com/article/10.1007%2Fs10664-006-9002-8#page-1

[5] A. Monden, D. Nakae, T. Kamiya, S. Sato, K. Matsumoto: Software quality analysis by code clones in industrial legacy software. In Proceedings Eighth IEE Symposium on Software Metrics 2002. IEE, S. 87-94, 2002.

[6] E. Juergens, F. Deissenboeck, B. Hummel, S. Wagner: Do Code Clones Matter?, In ICSE 2009, IEE 31st International Conference on Software Engineering 2009. Canada, IEEE, S. 485-495, 2009.

[7] R. Komondoor, S. Horwitz: Effective, automatic procedure extraction, In Proceedings IWPC 2003 of the *11th IEEE International Workshop on Program Comprehension. Washington, DC, USA. IEEE Computer Society, S. 33-43, 2003.*

[8] Y. Higo, T. Kamiya, S. Kusumoto, K. Inoue: Aries: refactoring support tool for code clone, In ACM SIGSOFT Software Engineering Notes, 30(4), New York.S. 1-4, 2005. Siehe: http://dx.doi.org/10.1145/1082983.1083306

[9] M. Balazinska, E. Merlo, M. Dagenais, B. Lague, K. Kontogiannis: Partial redesign of java software systems based on clone analysis, In Proceedings of the Sixth Working Conference on Reverse Engineering. Washington, DC, USA. IEEE Computer Society, S. 326-336, 1999.

[10] F. Rahman, C. Bird, P. Devanbu: Clones: What is that Smell?, In 7th IEEE Working Conference on Mining Software Repositories (MSR). Cape Town. IEEE Computer Society, S. 72-81, 2010.

[11] R. Koschke: Survey of Research on Software Clones, In Dagstuhl Seminar Proceedings 06301. Universität Bremen, 2007. Siehe: http://drops.dagstuhl.de/opus/volltexte/2007/962

[12] B. S. Baker: A Program for identifying Duplicated Code. Computing Science and Statistics. In Proceedings of the 24th Symposium on the Interface. S. 24:49-57, 1992.

[13] Y. Ueda, T. Kamiya, S. Kusumoto, K. Inoue: On Detection of Gapped Code Clones using Gap Locations. In Proceedings of the Ninth Asia-Pacific Software Engineering Conference. Washington, DC, USA. IEEE Computer Society, S. 327-336, 2002.

[14] B. S. Baker: On Finding Duplication and Near-Duplication in Large Software Systems. In Proceedings of 2nd Working Conference on Reverse Engineering. Los Alamitos, California. IEEE Computer Society, S. 86-95, 1995.

[15] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, E. Merlo: Comparison and Evaluation of Clone Detection Tools. IEEE Transactions on Software Engineering, 33(99), S. 577–591, 2007.

[16] C. Kapser, M. Godfrey: A Taxonomy of Clones in Source Code: The reengineers most wanted list. In Working Conference on Reverse Enginnering. IEEE Computer Society, 2003.

[17] F. Raimer: Erkennung von Falsch-Positiven Softwareklonen mittels Lernverfahren. Dissertation, Universität Bremen, 2014, S. 19-32. ISBN: 978-3-8325-3651-0, Logos Verlag Berlin GmbH, Berlin, 2014.

[18] K. Ch. Roy, J. R. Cordy, R. Koschke: Comparison and Evaluation of Code Clone Detection Techniques and Tools: A Qualitative Approach. Science of Computer Programming, 74(7), S. 470-495, 2009.

[19] E. Juergens, F. Deissenboeck, B. Hummel: CloneDetective – A Workbench for Clone Detection Research. In Proceedings of the 31st International Conference on Software Engineering. Washington, DC, USA. IEEE Computer Sciety, S. 603-606, 2009.

[20] S. Wagner: Vorlesung: Qualitätssicherung und Wartung (QSW), SS2014; Universität Stuttgart

[21] R. Koschke, S. Simon: Vorlesung Software-Reengineering, WS2004/2005, Folien 119-124, Universität Bremen, Universität Stuttgart.

[22] J. H. Johnson: Substring matching for clone detection and change tracking. In: Proceedings International Conference on Software Maintenance. Victoria, BC. IEEE Computer Society, S. 120–126, 1994.

[23] S. Ducasse, M, Rieger, S. Demeyer: A Language Independent Approach for Detecting Duplicated Code. In Proceedings of the IEEE International Conference on Software Maintenance. Oxford. IEEE Computer Society, S. 109–118, 1999.

[24] R. Koschke: Vorlesung Software-Reengineering, WS2009/2010, Folien 22-35, Universität Bremen.

[25] S. Bellon: Vergleich von Techniken zur Erkennung duplizierten Quellcodes. Diplomarbeit, Universität Stuttgart, 2002.

[26] B. S. Baker, R. Giancarlo: Longest Common Subsequence from Fragments via Sparse Dynamic Programming. In Algorithms – ESA ’98. S. 79-80, Springer Verlag Berlin Heidelberg, 1998.

[27] I.D. Baxter, A. Yahin, L. Moura, M. Sant’Anna, L. Bier: Clone Detection Using Abstract Syntax Trees. In International Conference on Software Maintenance. Bethesda, MD. IEEE Computer Society, S. 368–377, 1988.

[28] J. Krinke: Identifying Similar Code with Program Dependence Graphs. In Proceedings Eighth Working Conference on Reverse Engineering. Stuttgart. IEEE Computer Society, S. 301–309, 2001.

[29] J. Mayrand, C. Leblanc, E. M. Merlo: Experiment on the Automatic Detection of Function Clones in a Software System using Metrics. In Proceedings of the International Conference on Software Maintenance. Washington, DC; USA. IEEE Computer Society, S. 244–254, 1996.

[30] Bryn O’Sullivan: Mercurial: The Definite Guide; O’Reilly Media, 2009. Onlinebuch Link: http://hgbook.red-bean.com/read/

[31] S. Wagner: Vorlesung Grundlagen Software-Engineering, SS2014, Universität Stuttgart.

[32] Git Software, Dokumentation: 1.1 Los geht’s - Wozu Versionskontrolle?. Siehe: http://git-scm.com/book/de/Los-geht%E2%80%99s-Wozu-Versionskontrolle%3F, Letzter Zugriff am 28.09.2014.

[33] Fog Creek Software, Dokumentation FogBugz; Projektmanagement. Siehe: http://www.fogcreek.com/fogbugz/features/project-management/, Letzter Zugriff am 28.09.2014.

[34] Fog Creek Software, Dokumentation FogBugz, Issue-Tracking-System. Siehe: http://www.fogcreek.com/fogbugz/features/issue-tracking/, Letzter Zugriff am 28.09.2014.

[35] Fog Creek Software, Dokumentation Kiln. Siehe: https://www.fogcreek.com/kiln/, Letzter Zugriff am 28.09.2014

[36] Fog Creek Software, Dokumentation Kiln, Team-Up. Siehe: http://www.fogcreek.com/kiln/features/team-up/, Letzter Zugriff am 28.09.2014

[37] Fog Creek Software, Dokumentation Kiln. Siehe:

http://www.fogcreek.com/kiln/features/get-organized/, Letzter Zugriff am 28.09.2014

[38] Fog Creek Software, Kiln API: Repositories. Siehe: https://developers.fogbugz.com/default.asp?W166, Letzter Zugriff am 28.09.2014.

[39] R. Koschke: Vorlesung Software-Reengineering, WS2009/2010, Folien 15-18, Universität Bremen.

[40] M. Rieger, S. Ducasse, M. Lanza: Insights into system–wide code duplication. In Proceedings of the 11th Working Conference on Reverse Engineering. IEEE Computing Society, S. 100.109, 2004.

[41] M. Rieger: Effective Clone Detection Without Language Barrier. Inauguraldissertation, Universität Bern, 2005.

[42] L. Jiang, Z. Su, E. Chiue: Context-based detection of clone-related bugs. In Proceedings of the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of the software engineering. New York, USA. S. 55-64. ISBN 978-1-59593-811-4.

[43] Z. Li, S. Lu, S. Myagmar, Y. Zhou: CP-Miner: Finding copy-paste and related bugs in large-scale software code. IEE Transactions on Software Engineering, 32(3), S. 176 – 192, 2006.

[44] E. Juergens, B. Hummel, F. Deissenboeck, M. Feilkas: Static Bug Detection Through Analysis of Inconsistent Clones. In Workshopband SE Konferenz 2008. LNI, GI, 2008.

[45] J. Krinke: A Study of Consistent and Inconsistent Changes to Code Clones. In WCRE 2007 of the 14th Working Conference on Reverse Engineering. Vancouver, BC. IEEE Computing Society, S. 170-178, 2007.

[46] C. Kapser, M. W. Godfrey: Cloning considered harmful. In WCRE ’06 on the 13th Working Conference on Reverse Engineering. Benevento. IEEE Computing Society, S. 19-28, 2006.

[47] S. Thummalapenta, L. Cerulo, L. Avensano, M. D. Penta: An empirical study on the maintenance of source code clones. Empirical Software Engineering, 15(1), S.1-34, 2009.

[48] M. Kim, L. Bergman, T. Lau, D. Notkin: An Ethnographic Study of Copy and Paste Programming Practices in OOPL. In Proceedings of the 2004 International Symposium on Empirical Software Engineering. IEEE Computer Society, S. 83-92, 2004.

[49] F. Deißenböck, T. Seifert: Kontinuierliche Qualitätsüberwachung mit CONQAT: Paper, Institut Software & Systems Engineering, Technische Universität München.

[50] ConQat Tool: Dokumentation ConQat User Guide 2013.10. Siehe: https://www.cqse.eu/download/conqat/conqat-book-2013.10.pdf, Letzter Zugriff am 28.09.2014.

[51] E. Juergens, F. Deissenboeck, M. Feilkas, B. Hummel, B. Schaetz, S. Wagner, Ch. Domann, J. Streit: Can Clone Detection Support Quality Assessments of Requirements Specifications?. In Proceedings of the 32nd ACM/IEE International Conference on Software Engineering (2). New York, USA. S 79-88, 2010. ISBN: 978-1-60558-719-6.

[52] M. Feilkas, D. Ratiu, E. Juergens: The Loss of Architectural Knowledge during System Evolution: An Industrial Case Study. In Proceedings of the 17th IEEE International Conference on Program Comprehension 2009. Vancouver, BC. IEEE Computing Society, S. 188-197, 2009.

[53] M. Metuh: Schaffung einer Basis für die kontinuierliche Qualitätsanalyse. Diplomarbeit, Universität Stuttgart, 2012.

Erklärung

Ich versichere, diese Arbeit selbstständig verfasst zu haben. Ich habe keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommene Aussagen als solche gekennzeichnet. Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines anderen Prüfungsverfahrens. Ich habe diese Arbeit bisher weder teilweise noch vollständig veröffentlicht. Das elektronische Exemplar stimmt mit allen eingereichten Exemplaren überein.

Unterschrift:

Stuttgart, 01.10.2014