Otto-von-Guericke-Universität Magdeburg

Fakultät für Informatik
Institut für Technische und Betriebliche Informationssysteme

# Masterarbeit

## Migrating Cloned Software Products: A Better Mousetrap for Finding Similar Code

Verfasser:

Philipp Müller

30. April 2018

Betreuer:

Prof. Dr. rer. nat. habil. Gunter Saake,
Dr.-Ing. Sandro Schulze,
Dipl.-Inf. Wolfram Fenske

Universität Magdeburg
Fakultät für Informatik
Postfach 4120, D–39016 Magdeburg
Germany

**Philipp, Müller:**
*Migrating Cloned Software Products: A Better Mousetrap for Finding Similar Code*
Masterarbeit, Otto-von-Guericke-Universität Magdeburg, 2018.

# Danksagung

Ich bedanke mich bei allen Personen, die mich im Rahmen dieser Masterarbeit unterstützt haben und mir die Anfertigung dieser Masterarbeit ermöglicht haben. Ich bedanke mich zunächst bei Prof. Dr. Gunter Saake, welcher es mir ermöglicht hat, meine Masterarbeit innerhalb seiner Arbeitsgruppe zu schreiben. Ein besonders großer Dank geht an Wolfram Fenske, welcher mich während der Anfertigung dieser Masterarbeit hervorragend betreut hat. Er stand mir stets mit seinen Ideen und Wissen zur Seite und hat mit seinen konstruktiven Kritiken einen wichtigen Anteil zur Verbesserung meiner Masterarbeit beigetragen. Des Weiteren bedanke ich mich bei Dr.-Ing. Sandro Schulze, welcher sich bereit erklärt hat, die wichtige Rolle des Zweitgutachters zu übernehmen. Abschließend bedanke ich mich bei all jenen, die mich während der Anfertigung dieser Masterarbeit moralisch unterstützt haben.

# Inhaltsverzeichnis

**Inhaltsverzeichnis** iii

**Abbildungsverzeichnis** vii

**Tabellenverzeichnis** ix

**Listings** xii

**Verzeichnis der Abkürzungen** xiii

**1 Einleitung** 1

**2 Grundlagen** 5

2.1 Clone-And-Own . . . . . . . . . . . . . . . . . . . . . . . . . . . . 5

2.1.1 Vorteile von Clone-And-Own . . . . . . . . . . . . . . . . . . . 5

2.1.2 Nachteile von Clone-And-Own . . . . . . . . . . . . . . . . . . 6

2.2 Softwareproduktlinien . . . . . . . . . . . . . . . . . . . . . . . . 7

2.2.1 Feature-Modellierung . . . . . . . . . . . . . . . . . . . . . . . 8

2.2.2 Implementierungstechniken für SPL . . . . . . . . . . . . . . . 9

2.2.3 Umstieg von Clone-And-Own zu Softwareproduktlinien . . . . . . 12

2.3 Codeklone . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 13

2.3.1 Definition . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 13

2.3.2 Typen von Codeklonen . . . . . . . . . . . . . . . . . . . . . . 14

2.3.3 Codeklonerkennung . . . . . . . . . . . . . . . . . . . . . . . . 15

2.3.3.1 Arten der Klonsuche . . . . . . . . . . . . . . . . . . . 15

2.3.3.2 Textbasierte Verfahren . . . . . . . . . . . . . . . . . . 16

2.3.3.3 Modellbasierte Verfahren . . . . . . . . . . . . . . . . . 17

2.3.3.4 Weitere Verfahren . . . . . . . . . . . . . . . . . . . . . 18

2.3.4 Entfernung von Codeklonen . . . . . . . . . . . . . . . . . . . . . 18
2.4 Zusammenfassung . . . . . . . . . . . . . . . . . . . . . . . . . . . . 19

**3 Anforderungen und Konzept 21**
3.1 MBCC Framework Macumba . . . . . . . . . . . . . . . . . . . . . . 21
3.1.1 Architektur . . . . . . . . . . . . . . . . . . . . . . . . . . . . 21
3.1.2 Workflow . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 23
3.1.3 Bedeutung der Metrik bei der Variabilitätsanalyse . . . . . . . . . 28
3.2 Anforderungen . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 31
3.3 Konzept . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 33
3.3.1 Anpassung und Erweiterung des MBCC Frameworks . . . . . . . 33
3.3.2 Aggregation der Vergleichsdaten . . . . . . . . . . . . . . . . . . . 34
3.4 Zusammenfassung . . . . . . . . . . . . . . . . . . . . . . . . . . . . 37

**4 Implementierung 39**
4.1 N-weiser Vergleich im MBCC Framework . . . . . . . . . . . . . . . . 39
4.2 Import der Vergleichsdaten . . . . . . . . . . . . . . . . . . . . . . . 40
4.3 Aufbereitung der Vergleichsdaten . . . . . . . . . . . . . . . . . . . . 42
4.3.1 Datenmodell der Vergleichsdaten . . . . . . . . . . . . . . . . . . 42
4.3.2 Berechnung der Ähnlichkeiten . . . . . . . . . . . . . . . . . . . . 44
4.4 Zusammenfassung . . . . . . . . . . . . . . . . . . . . . . . . . . . . 46

**5 Evaluation 49**
5.1 Fragestellung . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 49
5.2 Fallstudie . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 50
5.3 Durchführung . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 51
5.4 Verteilung der Ähnlichkeiten . . . . . . . . . . . . . . . . . . . . . . 53
5.5 Diskussion . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 59
5.5.1 Qualitativer Vergleich zwischen modellbasierter und textbasierter Codeklonerkennung . . . . . . . . . . . . . . . . . . . . . . . . . 59
5.5.2 Quantitativer Vergleich zwischen modellbasierter und textbasierter Codeklonerkennung . . . . . . . . . . . . . . . . . . . . . . . 67
5.5.3 Einfluss der Metriken auf die Ergebnisse . . . . . . . . . . . . . . 70
5.5.4 Zusammenführung von Codeklonen . . . . . . . . . . . . . . . . . 76

5.5.5 Threats to Validity . . . . . . . . . . . . . . . . . . . . . . . . . 77
5.6 Zusammenfassung . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 78

**6 Verwandte Arbeiten 81**

**7 Zusammenfassung und Ausblick 83**

**Literaturverzeichnis 87**

# Abbildungsverzeichnis

2.1 Kostenvergleich zwischen Produktvarianten einer SPL und Einzelsystemen [PBvdL05] . . . . . 7
2.2 Beispiel für ein Feature-Diagramm [TKB+14] . . . . . 9
2.3 Beispiel für Präprozessor Anweisungen [TKB+14] . . . . . 10
2.4 Verschiedene Codeklontypen im Programmcode [FMS+17] . . . . . 15

3.1 Architektur des MBCC Frameworks (entnommen aus [Tie16]) . . . . . 22
3.2 Workflow des MBCC Frameworks (entnommen aus [Tie16]) . . . . . 24
3.3 Ausschnitt aus einem HTML-Report . . . . . 27
3.4 Ablauf zur Berechnung der Klassenähnlichkeit . . . . . 36

4.1 Prozess zum Import der Vergleichsdaten . . . . . 41

5.1 Histogramm für alle Vergleiche auf Methodenebene . . . . . 55
5.2 Histogramm für alle Vergleiche über den Vergleichswert 0,5 auf Methodenebene . . . . . 55
5.3 Histogramm für alle Vergleiche auf Klassenebene . . . . . 57
5.4 Histogramm für alle Vergleiche über den Vergleichswert 0,5 auf Klassenebene 57
5.5 Histogramm für alle Vergleiche auf Packageebene . . . . . 58
5.6 Histogramm für alle Vergleiche über den Vergleichswert 0,5 auf Packageebene . . . . . 58
5.7 Histogramm für alle Vergleiche auf Projektebene . . . . . 59

# Tabellenverzeichnis

5.1 Statistik zu den Projekten der ApoGames-Reihe . . . . . . . . . . . . . . . 51
5.2 Konfusionsmatrix zur Ähnlichkeit aller Projekte für alle Metriken . . . . 56
5.3 Kategorisierung der 100 betrachteten Methodenvergleiche . . . . . . . . . 68
5.4 Erkennungsrate von CPD für die 100 kategorisierten Methodenvergleiche 68
5.5 Precision, Recall und F-Measure für die Ergebnisse der angepassten Variabilitätsanalyse . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 69

# Listings

3.1 Methode onCreateApp aus ApoClock.java . . . 26
3.2 Methode onCreate aus ApoSnake.java . . . 26
3.3 Auszug aus einem JSON-Report . . . 28
3.4 Konfiguration für Dateien, Namensräumen und Klassifizierer . . . 29
3.5 Konfiguration für Parameter . . . 30
3.6 Konfiguration für Methoden . . . 30
3.7 Programmvariante 1 (angelehnt an[Tie16]) . . . 31
3.8 Programmvariante 2 (angelehnt an [Tie16]) . . . 31

4.1 XSL-Stylesheet zur Umwandlung in die gewünschte Struktur . . . 42
4.2 SQL-Anweisung zur Berechnung der Anzahl der Methoden pro Klasse . . 44
4.3 SQL-Anweisung zur Festlegung der relevanten Tupel für die Berechnung der Klassenähnlichkeit . . . 45
4.4 SQL-Anweisung zur Berechnung der Klassenähnlichkeit . . . 46

5.1 verwendete Gewichtung innerhalb der Methodensignatur . . . 52
5.2 Qualitativer Vergleich zwischen onBackButtonPressed (ApoClockMenu) und onBackButtonPressed (ApoMonoMenu) . . . 60
5.3 Qualitativer Vergleich zwischen getAllLevelsSorted (ApoDiceUserlevels) und getAllLevelsSorted (MyTreasureUserLevels) . . . 61
5.4 Qualitativer Vergleich zwischen getIBackground (ApoEntity) und getIBackground (ApoEntity) . . . 62
5.5 Qualitativer Vergleich zwischen setEditor (ApoDicePanel) und setEditor (ApoMonoPanel) . . . 63
5.6 Qualitativer Vergleich zwischen setSolved (ApoLevelChooserButton) und setBUse (ApoEntity) . . . 64
5.7 Qualitativer Vergleich zwischen canBeTested (ApoMonoEditor) und getLevelString (ApoEntity) . . . 65

5.8 Vergleich zwischen contains (ApoClockEditorClockStats) und isOpaque (ApoEntity) . . . . . 71
5.9 Vergleich zwischen touchedDragged (ApoClockPuzzleGame) und onFinish (ApoDice) . . . . . 72
5.10 Vergleich zwischen render (ApoButton) und render (ApoEntity) . . . . . 72
5.11 Vergleich zwischen setButtonFunction (ApoMonoPanel) und touchedButton (ApoSnakeEditor) . . . . . 73

# Verzeichnis der Abkürzungen

| | |
|---|---|
| **AOP** | Aspekt-orientierte Programmierung |
| **AST** | Abstract Syntax Tree |
| **CPD** | Copy/Paste Detector |
| **CSV** | Comma-separated values |
| **DOP** | Delta-orientierte Programmierung |
| **DSL** | Domain Specific Language |
| **FOP** | Feature-orientierte Programmierung |
| **FST** | Feature Structure Tree |
| **EMF** | Eclipse Modeling Framework |
| **GUI** | Graphical User Interface |
| **HTML** | Hypertext Markup Language |
| **JaMoPP** | Java Model Parser and Printer |
| **JSON** | JavaScript Object Notation |
| **M2M** | Model-to-Model |
| **MBCC** | Model-Based Code Comparison |
| **PDG** | Program Dependency Graph |
| **PHP** | Hypertext Preprocessor |
| **SQL** | Structured Query Language |
| **SPL** | Software-Produktlinie |
| **SPLE** | Software Product Line Engineering |

**XML** Extensible Markup Language

**XSLT** XSL Transformation

# 1 Einleitung

Bei der Entwicklung von kommerziellen Softwareprodukten wird häufig bereits vorhandener Programmcode aus ähnlichen und bereits bewährten Softwareprodukten kopiert und auf die Wünsche des Kunden angepasst. Dieses Prinzip wird auch als Clone-And-Own bezeichnet und ermöglicht eine kostengünstige Entwicklung von Softwareprodukten, da die Wiederverwendung von bereits vorhandenen Programmcode weniger Aufwand in der Entwicklung und beim Testen, im Vergleich zur kompletten Neuentwicklung, bedeutet [CN01]. Des Weiteren ist es möglich, die neu erstellten Softwareprodukte unabhängig von den bereits vorhandenen Softwareprodukten, aus denen der Programmcode entnommen wurde, weiterzuentwickeln, um somit den Anforderungen des Kunden gerecht zu werden [DB07]. Daraus resultiert jedoch ein großes Problem von Clone-and-Own, nämlich der nachträgliche Wartungs- und Weiterentwicklungsaufwand zur Behebung von Fehlern bei wiederverwendeten Programmcodekomponenten [GFGP06]. Wenn sich ein Fehler bei einer wiederverwendeten Komponente herausstellt, so muss der Programmcode bei allen Softwareprodukten, welche diese Komponente ebenfalls teilen, korrigiert werden. Mit wachsender Anzahl an Softwareprodukten kann dies zu einem erheblichen Problem bei der Wartung führen, da die historisch gewachsenen Veränderungen des Programmcodes nicht immer einfach ersichtlich sind und die Gefahr besteht, dass ein nachträglich gefundener Fehler nicht in allen Softwareprodukten korrigiert wird.

Um dem entgegenzuwirken eignet sich die Nutzung von Software-Produktlinien. Eine Software-Produktlinie (SPL) ist eine Menge von Softwareprodukten, welche sich einen gemeinsamen Programmcode teilen, welcher durch eine gemeinsame Plattform entwickelt und verwaltet wird [CE00] [ABKS13]. Somit teilen sich alle Produktvarianten der Software-Produktlinie eine gemeinsame Basis. Bei Änderungen oder Erweiterungen am Programmcode müssen im Gegensatz zum Clone-and-Own nur an einer zentralen Stelle im Programmcode Änderungen vorgenommen werden. Dadurch lassen sich die erhöhten Wartungskosten bei der Korrektur von Fehlern für die verschiedenen Softwareprodukte deutlich senken [RCC13].

Der nachträgliche Wechsel von Clone-and-Own zu Software-Produktlinien erfordert jedoch ein gut durchdachtes Konzept, um die damit verbundenen Herausforderungen bewältigen zu können. Dazu gehört das Identifizieren aller Features, welche die Unterschiede und Gemeinsamkeiten in allen Produktvarianten kennzeichnen [CE00] [ABKS13]. Außerdem muss in jedem Softwareprodukt ermittelt werden, welcher Programmcode genau welches Feature implementiert, sodass die verschiedenen Varianten zur Implementierung eines Feature dann zusammengefasst werden können. Als Ergebnis soll nur noch eine zentrale Variante existieren, welche als Basis für alle Instanzen der Software-Produktlinie genutzt werden soll [FTS13].

Die Ermittlung der Gemeinsamkeiten und Unterschiede zwischen zwei Softwareprodukten ist für die Zusammenführung in eine SPL ein wichtiger Schritt und kann für große Softwareprodukte aufgrund des hohen Aufwands nicht manuell durchgeführt werden. Als Alternative gibt es dafür automatisierte Verfahren, welche die Softwareprodukte nach geklonten Programmcode durchsuchen und somit den Entwickler beim Auffinden und Zusammenführen des geklonten Programmcodes unterstützen. Zur Erkennung von Codeklonen gibt es verschiedene Ansätze, welche ihre Stärken und Schwächen haben.

Ein relativ neuer Ansatz ist die modellbasierte Codeklonerkennung. Zur Untersuchung der modellbasierten Codeklonerkennung hat die Technische Universität Braunschweig einen Forschungsprototypen entwickelt. Bei diesem Forschungsprototypen handelt es sich um ein Framework, welches einen Vergleich von Softwareprodukten anhand deren Programmcodestrukturen vornimmt. Dazu werden Metamodelle aus der Quellcodedatei extrahiert und auf Ähnlichkeit überprüft, wodurch sich auch Ähnlichkeiten in den Programmcodestrukturen feststellen lassen [Tie16] [WTS+16]. Dieses Framework wurde als "Model-Based Code Comparison Framework (MBCC Framework) Macumba" benannt und die Ermittlung der Ähnlichkeit von zwei Vergleichsobjekten wird im weiteren Verlauf dieser Arbeit als "Variabilitätsanalyse" bezeichnet. Im Gegensatz zu textbasierten Verfahren wird bei der Variabilitätsanalyse des MBCC Frameworks nicht ermittelt, ob zwischen zwei Vergleichsobjekten ein Codeklon vorliegt, sondern wie ähnlich sich zwei Vergleichsobjekte sind. Dieser Ähnlichkeitsgrad gibt an, zu wie viel % zwei verschiedene Vergleichsobjekte gleich sind.

Für die Wissenschaft und die Industrie ist die Frage, ob die modellbasierte Codeklonerkennung eine gute Alternative zu anderen Verfahren der Codeklonerkennung ist, von wichtiger Bedeutung. Wenn die modellbasierte Codeklonerkennung eine gute Alternative ist, würde sich durch deren Anwendung sicherlich Aufwand und somit auch Geld bei der Zusammenführung von geklonten Programmcode zu einer SPL sparen lassen.

**Ziel dieser Arbeit**

Das Ziel meiner Arbeit ist zu zeigen, wie die Arbeit von Entwicklern, welche eine Zusammenführung von durch Clone-And-Own entstandenen Programmcodes zu einer SPL vornehmen, mit Hilfe der Variabilitätsanalyse des MBCC Frameworks erleichtert werden kann. Dafür ist es notwendig, prototypische Anpassungen am MBCC Framework vorzunehmen, da die Ergebnisse der Variabilitätsanalyse für mein Vorhaben noch nicht ausreichen und verwendbar sind. Anhand einer geeigneten Fallstudie werden die Ergebnisse der Variabilitätsanalyse genauer untersucht und ausführlich mit den Ergebnissen eines textbasierten Verfahrens zur Codeklonerkennung verglichen. Bei der Untersuchung der Ergebnisse der Variabilitätsanalyse sollen außerdem Stärken und Schwächen der Variabilitätsanalyse bei der Erkennung von Codeklonen ermittelt werden.

**Gliederung dieser Arbeit**

Als erstes werden in Kapitel 2 die Grundlagen erklärt. In den Grundlagen werden die Begriffe Clone-And-Own, SPL und Codeklone sowie Möglichkeiten zur

Erkennung von Codeklonen vorgestellt. In Kapitel 3 wird das bereits erwähnte MBCC Framework vorgestellt. Darauf aufbauend erfolgt die Definition der Anforderungen und im Konzept wird vorgestellt, wie ich die definierten Anforderungen erfüllen werde. Die Implementierung meiner im Konzept angekündigten Anpassungen werden in Kapitel 4 ausführlich vorgestellt. In Kapitel 5 erfolgt die Evaluation und zur Beantwortung von aktuellen Forschungsfragen erfolgt eine ausführliche Diskussion der Ergebnisse. Anschließend wird in Kapitel 6 ein Überblick zu verwandten Arbeiten, die sich mit der Evaluation und dem Vergleich von Codeklonerkennungsverfahren befassen, gegeben. Abschließend erfolgt in Kapitel 7 eine Zusammenfassung der Ergebnisse und es werden noch offene Themen für zukünftige Arbeiten genannt.

**Ergebnisse dieser Arbeit**

Für die durchgeführte Fallstudie wurde festgestellt, dass Codeklone mit der modellbasierten Codeklonerkennung besser erkannt wurden, im Vergleich zur textbasierten Codeklonerkennung. Dazu wird auch gezeigt, welchen Einfluss verschiedene Metriken bei der Ermittlung des Ähnlichkeitsgrades von zwei Vergleichsobjekten haben. Außerdem wird eine begründete Empfehlung formuliert, ab welchem Ähnlichkeitsgrad die Zusammenführung von zwei Vergleichsobjekten sinnvoll ist, um die Arbeit eines Entwicklers bei eben jener genannten Zusammenführung zu unterstützen.

# 2 Grundlagen

In diesem Kapitel beschreibe ich die Grundlagen dieser Arbeit. Dazu werde ich zunächst den Begriff Clone-And-Own erläutern. Es wird erläutert, welche Vorteile und Nachteile die Nutzung von Clone-And-Own mit sich bringt. Als Alternative zur Überwindung der Nachteile von Clone-And-Own erkläre ich anschließend das Prinzip von Softwareproduktlinien. Dazu wird erklärt wie SPLs aus den Features aufgebaut sind und wie sich die Beziehungen von Features mit der Feature-Modellierung ausdrücken lassen. Außerdem werden verschiedene Ansätze zur Implementierung von SPLs vorgestellt. Es wird zudem verdeutlicht, warum eine Umstellung von Clone-And-Own nach Softwareproduktlinien sinnvoll ist. Ein Hauptaspekt bei der Umstellung stellt die Codeklonerkennung dar, welche auch das Hauptthema dieser Arbeit ist. Daher werde ich abschließend verschiedene Verfahren zur Erkennung von Codeklonen vorstellen und ich gehe darauf ein, wie diese Klone entfernt werden können.

## 2.1 Clone-And-Own

In der Softwareentwicklung gibt es für die Schaffung neuer Softwareprodukte mehrere Wege. Eine Situation ist zum Beispiel, dass ein neues Softwareprodukt entwickelt werden soll, welches in seinen Funktionalitäten dem eines schon vorhandenen Softwareproduktes ähnelt. In dieser Situation liegt es nahe, die gesuchten Funktionalitäten aus dem vorhandenem Softwareprodukte für das neue Softwareprodukt zu übernehmen. Statt einer kompletten Neuentwicklung wird der Programmcode, der die gewünschte Funktionalität umsetzt, kopiert und für das neue Softwareprodukt wiederverwendet und eventuell angepasst. Dieses Vorgehen zur Softwareentwicklung wird als Clone-And-Own bezeichnet. Clone-And-Own wird somit als Softwareentwicklungsverfahren zur Erzeugung von Produktvarianten durch Wiederverwendung bereits vorhandener Softwareprodukte definiert [DB07]. In der Industrie häufig angewendet, wird dazu bereits funktionierender und getesteter Programmcode kopiert und die Kopie den Wünschen der Kunden entsprechend angepasst. In der Softwareentwicklung wird dieses Vorgehen auch als Forking bezeichnet, da das geklonte Produkt als Verzweigung (engl. fork) unabhängig vom Ursprungsprodukt existiert und weiterentwickelt wird [RKBC12].

### 2.1.1 Vorteile von Clone-And-Own

Aus der Begriffserklärung lassen sich bereits die Vorteile ableiten. Dazu gehören die niedrigen Entwicklungskosten, da der kopierte Programmcode nicht komplett neu entwickelt werden muss. Die Wiederverwendung hat außerdem den Vorteil, dass der wiederverwendete Programmcode bereits getestet wurde und die Testläufe bestanden hat. Hiermit können erneute Testaufwände reduziert werden, wodurch sich Zeit und Geld

sparen lässt. Die Entwickler können somit schnell auf die Anforderungen von Kunden reagieren. Somit ist auch eine schnellere Markteinführung der neuen Produktvariante möglich, im Vergleich zur kompletten Neuentwicklung [KG06]. Ein weiterer Vorteil ist, dass die neue Variante unabhängig von bestehenden Varianten weiterentwickelt werden kann. Somit besteht keine Gefahr, unbeabsichtigt Fehler in den existierenden Produkten zu verursachen.

### 2.1.2 Nachteile von Clone-And-Own

Die Nutzung von Clone-And-Own wird jedoch langfristig eher Nachteile mit sich bringen. Es kann nämlich vorkommen, dass wiederverwendeter Programmcode doch fehlerhaft ist oder er aufgrund anderer geänderter Funktionalitäten angepasst werden muss. Dies hat zur Folge, dass in sämtlichen Programmvarianten der duplizierte Programmcode geändert werden muss. Das Berichtigen in allen Programmvarianten ist somit zeitintensiv und erfordert einen höheren Wartungsaufwand [DB07], welcher sich durch die Nutzung von SPL verringern lässt.

Darüber hinaus müssen in sämtlichen Programmvarianten die Änderungen getestet werden. Dies ist notwendig, da nicht auszuschließen ist, dass in einzelnen Programmvarianten durch die Änderung des duplizierten Programmcodes neue Fehler entstehen, was ebenfalls einen höheren Wartungsaufwand nach sich zieht [RKBC12] [KG06]. Des Weiteren besteht die Möglichkeit, dass der fehlerhafte Programmcode nicht in allen Programmvarianten korrigiert wird, da nicht immer klar ist, in welchen Produktvarianten der fehlerhafte Programmcode verwendet wurde. Somit würden Fehler in bestimmten Produktvarianten weiterhin bestehen bleiben [RC07].

In Abbildung 2.1 werden die Kosten für die Entwicklung von Produktvarianten durch die Entwicklung neuer Einzelsysteme mit der Entwicklung durch SPL verglichen. Es lässt sich erkennen, dass die Kosten bei der Entwicklung von Einzelsystemen durch die Nutzung von Clone-And-Own mit zunehmender Anzahl von Produktvarianten sehr hoch werden können. Die Kosten bei der Erzeugung neuer Produktvarianten sind zwar relativ niedrig, da der kopierte Programmcode nicht neu entwickelt und getestet werden muss. Jedoch sind die Wartungskosten mit zunehmender Anzahl an Systemen dafür relativ hoch, da wie bereits beschrieben, der Aufwand zur Behebung von Fehlern in jedem kopierten System vorgenommen werden muss. Mit zunehmender Anzahl der Produktvarianten ist die Nutzung von SPL als Alternative daher empfehlenswert. Die Kosten von SPL sind bei geringer Anzahl an Produktvarianten noch höher als bei Clone-And-Own, da ein Mehraufwand bei der Erstentwicklung geleistet werden muss. Dies ändert sich jedoch bereits nach drei Systemen, da die Gesamtheit der Wartungskosten bei Clone-And-Own höher werden als bei SPL. Im nächsten Abschnitt wird daher der Begriff SPL sowie der Prozess zur Etablierung einer SPL, dem Software Product Line Engineering (SPLE), näher erklärt.

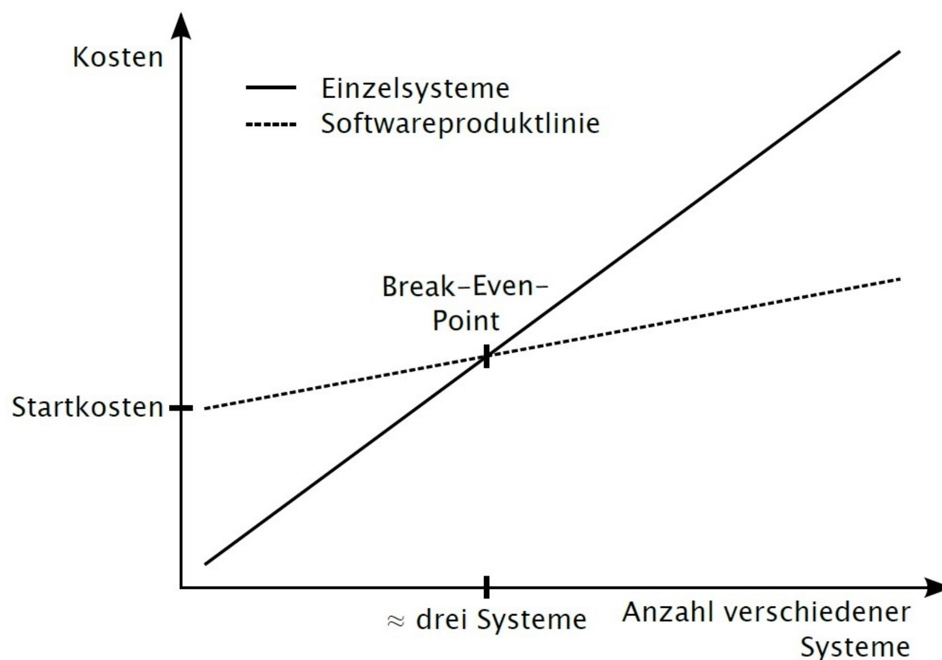



Abbildung 2.1: Kostenvergleich zwischen Produktvarianten einer SPL und Einzelsystemen [PBvdL05]

## 2.2 Softwareproduktlinien

Die Entwicklung neuer Software durch die Wiederverwendung von bereits existierender Software stellt für Entwicklern eine Alternative dar, als die komplette Neuentwicklung von Software. Wie in Abschnitt 2.1 vorgestellt, ist eine einfache Möglichkeit zur Wiederverwendung von Software die Nutzung von Clone-And-Own. Die Nachteile von Clone-And-Own überwiegen jedoch mit wachsender Anzahl an Produktvarianten, weshalb die Nutzung von SPLs als Lösung für dieses Problems sinnvoll ist. Bei einer SPL handelt es sich um eine Menge von Software-Produkten mit gemeinsamen Eigenschaften, die auf ein bestimmtes Marktsegment (Domäne) zugeschnitten sind. Alle Produkte einer SPL teilen sich bestimmte Kernkomponenten und werden somit auf einer gemeinsamen Basis entwickelt. Damit vereint eine SPL zum einen die Variabilität innerhalb einer Domäne und zum anderen die Wiederverwendung auf Basis einer gemeinsamen Plattform [PBvdL05].

Die Gemeinsamkeiten und Unterschiede der Produkte einer SPL stellen dabei die Features dar. Ein Feature ist eine Eigenschaft oder ein für den Nutzer sichtbares Verhalten eines Softwaresystems [KCH+90] [ABKS13]. Eine SPL besteht aus mehreren Features und durch die Nutzung von Features wird die Variabilität der SPL beschrieben. Durch die Beschreibung von Features mit Hilfe der Feature-Modellierung soll zudem die Kommunikation mit allen Beteiligten (Entwickler, Kunden) vereinfacht werden. Die Einzelheiten zur Feature-Modellierung werden in Abschnitt 2.2.1 erklärt.

Für jedes Produkt einer SPL werden bereits vorhandene Features ausgewählt und

zusammengesetzt. Die Auswahl und die Kombination der Features wird als Konfiguration bezeichnet. Mit einer spezifischen Konfiguration können somit die Anforderungen des Kunden und der Nutzer individuell erfüllt werden [PBvdL05]. Durch die individuelle Auswahl der Features lässt sich zudem die Entwicklungszeit neuer Produkte reduzieren, da bereits vorhandene Features keinen Mehraufwand in der Entwicklung bedeuten. Neue Produkte können dadurch früher auf den Markt gebracht werden, weil sie nicht komplett neu entwickelt werden müssen. Außerdem ist der Aufwand bei der Behebung von Fehlern geringer als bei Clone-And-Own, da Änderungen nur an einer Stelle im Programmcode vorgenommen werden müssen, damit sie für alle Produktvarianten gelten. Im Gegensatz dazu muss eine Änderung bei Clone-And-Own mehrmals vorgenommen werden, da für jede Produktvariante einzeln die Änderung realisiert werden muss.

In den nachfolgenden Abschnitten wird dargestellt, wie bei der Etablierung von SPLs vorgegangen wird. Dazu gehört sowohl die abstrakte Beschreibung einer SPL durch ein Feature-Modell sowie die Möglichkeiten eine SPL zu implementieren.

### 2.2.1 Feature-Modellierung

Features stellen die Konfigurationsoptionen dar und ermöglichen die Konfigurierung und Erzeugung der verschiedenen Produktvarianten einer SPL [PBvdL05]. Die Erzeugung einer Produktvariante erfolgt durch die Auswahl der gewünschten Features. Diese Auswahl ist jedoch nicht beliebig, da es sowohl gültige als auch ungültige Auswahlen einer Teilmenge von Features gibt. Eine Teilmenge von Features wird als Konfiguration bezeichnet. Nur eine gültige Konfiguration ermöglicht eine erfolgreiche Kompilierung zu einer funktionsfähigen Produktvariante. Ob eine Konfiguration gültig ist, lässt sich mit einem Feature-Modell bestimmen. Die Feature-Modellierung ermöglicht das Erstellen eines Feature-Modells. In Feature-Modellen werden die Features und ihre Beziehungen untereinander dargestellt und beschreiben die möglichen Kombinationen von Features für eine gültige Konfiguration [KCH+90] [ABKS13] [CE00]. Feature-Modelle beschreiben die Beziehungen und Abhängigkeiten der Features auf einer abstrakten Ebene und sind somit unabhängig von der konkreten Implementierung [CE00].

Die Darstellung eines Feature-Modells lässt sich mit Hilfe eines Feature-Diagramms darstellen [KCH+90]. Feature-Diagramme sind hierarchische Bäume und jedem übergeordneten Feature, welche als Eltern-Feature bezeichnet werden, können weitere untergeordnete Features zugewiesen werden, welche als Kind-Feature oder Subfeature bezeichnet werden. Ein Subfeature kann dabei erst für die Konfiguration ausgewählt werden, wenn auch deren Eltern-Feature ausgewählt wurde [CE00].

In Abbildung 2.2 ist ein Feature-Diagramm abgebildet. Die Knoten beinhalten Features und ein Feature ist entweder obligatorisch, optional oder alternativ. Darüber hinaus sind Features entweder abstrakt oder konkret. Die Auswahl eines abstrakten Features hat keinen direkten Einfluss auf die Implementierung und dem Quellcode. Durch die Auswahl eines abstrakten Features wird es jedoch möglich, deren Subfeatures auszuwählen, welche wiederum abstrakt oder auch konkret sein können. Die konkreten Features legen wiederum fest, wie das Produkt zusammengesetzt wird und sind im Feature-Diagramm immer die Blätter des hierarchischen Baums [TBK09]. Die Kanten

zwischen den Knoten stellen die Abhängigkeiten zwischen den Features dar. Ein Feature kann zudem mehrere Subfeatures haben und die Auswahl der Subfeatures erfolgt entweder alternativ (nur genau ein Subfeature darf ausgewählt werden) oder als Oder-Option (mindestens ein Subfeature muss ausgewählt werden) [Bat05]. Bei der Auswahl eines Features müssen zudem alle obligatorischen Subfeatures dieses Features ausgewählt werden. So ist in Abbildung 2.2 das Feature *Hello World* abstrakt und wird durch die beiden obligatorischen und konkreten Subfeatures *Hello* und *World* realisiert. Zusätzlich gibt es noch das optionale und abstrakte Subfeature *Feature*. Dieses Subfeature bietet eine alternative Auswahl zwischen den beiden konkreten Subfeatures *Wonderful* und *Beautiful*. Somit bietet dieses Feature-Diagramm drei Möglichkeiten, diese sind *Hello World*, *Hello Wonderful World* und *Hello Beautiful World*. Nicht in Abbildung 2.2 enthalten ist eine Oder-Option. Diese wird ähnlich wie eine Alternative dargestellt, mit dem Unterschied, dass der Kreisausschnitt zwischen den Kanten dunkel gefärbt ist.

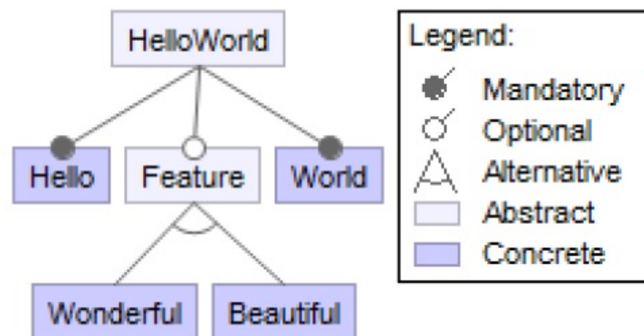



Abbildung 2.2: Beispiel für ein Feature-Diagramm [TKB+14]

Des Weiteren besteht die Möglichkeit, dass Abhängigkeiten zwischen den Features durch sogenannte *Cross-Tree Constraints* ausgedrückt werden. Dies sind aussagenlogische Ausdrücke, welche sich nicht durch die Baumstruktur darstellen lassen und werden für gewöhnlich unter dem Feature-Diagramm aufgeführt [TKB+14]. Ein Beispiel ist, dass wenn ein Feature "A" ausgewählt wird, so darf das Feature B nicht ausgewählt werden und umgekehrt. Der entsprechende aussagenlogische Ausdruck dafür ist dieser: $\neg(A \wedge B)$. Ein weiteres Beispiel wäre, dass die Auswahl von "A" die Auswahl von "B" impliziert. Der entsprechende aussagenlogische Ausdruck dafür ist dieser: $A \Rightarrow B$.

### 2.2.2 Implementierungstechniken für SPL

Nach der abstrakten Beschreibung durch ein Feature-Modell geht es nun um die Möglichkeiten die Features auf der Implementierungsebene zu instanziieren. Zur Errichtung einer SPL gibt es mehrere Implementierungstechniken, welche sich jedoch in zwei Kategorien einteilen lassen. Diese Kategorien sind nämlich die annotationsbasierten Verfahren und die kompositionsbasierten Verfahren [KAK08] [ABKS13].

### Annotationsbasierte Implementierungstechniken

Im Gegensatz zu den kompositionsbasierten Implementierungstechniken basieren die annotationsbasierten Implementierungstechniken darauf, dass die Features nicht in physisch getrennten Codefragmenten aufgeteilt werden. Es gibt stattdessen eine gemeinsame Codebasis, welche die Implementation jedes Features beinhaltet, was auch als "150% Modell" bezeichnet wird. Annotationen markieren im Quellcode, welche Codefragmente die entsprechenden Features implementieren. Die Erzeugung von Produktvarianten geschieht, indem die Annotationen geändert oder neue hinzugefügt werden [FTS13]. Dies geschieht mit Hilfe von Präprozessoren. Präprozessoren sind zudem unabhängig von der gewählten Programmiersprache einfach zu verstehen und zu konfigurieren. Bekannte Präprozessoren sind zum Beispiel der C-Präprozessor für die Programmiersprache C und der Java-Präprozessoren Munge [KA09].

```
class HelloWorld {
  void print() {
    //#if Hello
    System.out.print("Hello");
    //#endif
    //#if Beautiful
    System.out.print("␣beautiful");
    //#endif
    //#if Wonderful
//@   System.out.print(" wonderful");
    //#endif
    //#if World
    System.out.print("␣world!");
    //#endif
  }
  static void main(String[] args) {
    new HelloWorld().print();
  }
}
```

Abbildung 2.3: Beispiel für Präprozessor Anweisungen [TKB+14]

In Abbildung 2.3 sei eine Implementierung dargestellt. Ein Block, welcher zu einem Feature gehört, startet mit "if" und endet mit "endif". Wenn die Features "Hello", "Beautiful" und "World" ausgewählt sind, kann der Präprozessor den Programmcode von dem Feature "Wonderful" entfernen oder auskommentieren.

### Kompositionsbasierte Implementierungstechniken

Zu den kompositionsbasierten Verfahren gehören die Feature-orientierte Programmierung (FOP), die Delta-orientierte Programmierung (DOP) und die Aspekt-orientierte Programmierung (AOP). Bei kompositionsbasierten Verfahren werden die Features einer SPL in eigenständige und physisch voneinander getrennte Codefragmente aufgeteilt. Diese Aufteilung vermeidet bestimmte Probleme annotationsbasierter Verfahren. Dazu

gehört die Vermeidung von verwickelten und zerstreutem Programmcode, wodurch dessen Wiederverwendbarkeit erhöht wird [AK09] [FTS13].

Verglichen mit annotationsbasierten Ansätzen erfordern kompositionsbasierte Ansätze mehr Lernaufwand und die Implementierung ist nicht unabhängig von der gewählten Programmiersprache, wie es bei dem Einsatz von Präprozessoren der Fall ist [KA09]. Zum Beispiel kann bei der AOP AspectJ nur für Java genutzt werden und AspectC nur für C [FTS13].

#### Feature-orientierte Programmierung

Die FOP ist eine Erweiterung der Objekt-orientierten Programmierung. Bei ihr sind die Klassen in Features zerlegt und jedes Feature implementiert eine bestimmte Funktionalität. Dabei kann ein Feature mehrere Methoden von unterschiedlichen Klassen beinhalten und die Zusammenfügung der Features ermöglichen die automatische Erzeugung von Programmen. Auf der Implementierungsebene werden die Methoden, die zu einem Feature gehören, als zusammenhängende Einheit betrachtet. Diese zusammenhängende Einheit wird als Feature-Modul bezeichnet. Die Feature-Module erweitern optional das Basis-Programm, sodass ein individuelles neues Programm nach Wunsch des Kunden zusammengestellt werden kann [Pre97]. Die Implementierung von Feature-Modulen wird in FOP durch verschiedene Werkzeuge unterstützt. Zu diesen Werkzeugen gehören zum Beispiel AHEAD [BSR03] und FeatureHouse [ALMK10] [AKL13]. FeatureHouse wird nach der DOP zum besseren Verständnis noch vorgestellt.

#### Delta-orientierte Programmierung

Eine Erweiterung der FOP ist die DOP. Bei der DOP gibt es ein Kern-Modul, welches eine Standardapplikation ist. Dazu gibt es noch eine Menge von Delta-Modulen und jedes Delta-Modul kann Felder, Methoden und Klassen hinzufügen, aber auch entfernen. Delta-Modulen ist es möglich, die Superklasse einer existierenden Klasse zu ändern. Durch eine Konfiguration ist es möglich, ein Kern-Modul und mehrere Delta-Module auszuwählen und zu einem Softwaresystem zusammenzufügen [SBDT10].

#### FeatureHouse

Bei FeatureHouse handelt es sich um ein sprachunabhängiges Framework mit einer Sammlung von Programmen zur Komposition von Feature-Modulen. Feature-Module werden dabei durch einen Feature Structure Tree (FST) repräsentiert. Bei den FST handelt es sich um einen vereinfachten Abstract Syntax Tree (AST). Der Unterschied zum AST besteht darin, dass FST nur Details zur Implementierung enthält. Dazu gehören auch die Informationen, die für die Spezifikation der modularen Struktur eines Softwareartefakts notwendig sind. Dadurch kann mit einem FST jede Art von Softwareartefakten mithilfe einer hierarchischen Struktur dargestellt werden [ALMK10] [AKL13].

So kann FST zum Beispiel zur Repräsentation eines Java-Artefakts genutzt werden. Jedes Package, jede Klassen, jede Methoden und dergleichen werden jeweils als ein Knoten im FST dargestellt. Jeder Knoten besitzt dabei zwei Eigenschaften. Diese

Eigenschaften sind der Name und der Typ, welcher dem strukturellen Element des Softwareartefakts entspricht. Nicht enthalten sind die Statements und Expressions innerhalb einer Methode. Das Feature wird durch den Wurzelknoten des FST repräsentiert und die inneren Knoten stellen jeweils ein Strukturelement dar. Zu diesen Strukturelementen gehören beispielsweise Klassen und Methoden. Dabei ist es möglich, dass jeder innere Knoten weitere innere Knoten unter sich hat. Die Blätter des FST beinhalten die Strukturelemente der jeweils höheren inneren Knoten. Zu diesen Strukturelementen gehören wiederum die Methodenrümpfe, Feldinitialisierungen und mehr.

Mit der Superimposition der FSTs erfolgt dann die Komposition von Softwareartefakten. Die Superimposition ist der Prozess zur Generierung eines Produkts. Dieser Prozess zur Generierung eines Produkts wird als Feature-Komposition bezeichnet. Der Prozess der Superimposition setzt die Softwareartefakte durch das Verschmelzen ihrer dazugehörigen Strukturen zusammen. In objektorientierten Sprachen zählen zum Beispiel Klassen, Interfaces, Methoden und Felder zu diesen Strukturen. Die Vereinigung der Strukturen erfolgt durch die gleiche Signatur, welche sich aus dem Namen und Typ bildet [ALMK10] [AKL13].

Ausgedrückt wird die Superimposition mithilfe des mathematischen Verknüpfungsoperator über einer Menge von Features $F$ [ALMK10].

$$\bullet : F \times F \rightarrow F$$

Anhand von definierten Regeln erfolgt die Komposition von FSTs. Bei der Komposition werden die Knoten vereinigt, welche die gleiche Signatur aufweisen. Dieser Vorgang wird ausgehend vom Wurzelknoten rekursiv für alle Knoten der FSTs durchgeführt. Das Ergebnis einer Komposition zweier FSTs ist ein neuer FST mit den Eigenschaften aller Knoten beider FSTs. So wird beispielsweise durch die Komposition von zwei Java-Programmen auch wieder ein syntaktisch korrektes Java-Programm erzeugt [AKL13].

Die Aneinanderreihung von Kompositionen auf einer Menge von FSTs bildet somit eine Programmvariante.

$$v = F_1 \bullet F_2 \bullet \ldots \bullet F_n$$

### 2.2.3 Umstieg von Clone-And-Own zu Softwareproduktlinien

Der Umstieg von Clone-And-Own zur SPL bei Softwaresystemen stellt beim SPLE keine triviale Aufgabe dar und sollte daher mit einem methodischen Ansatz durchgeführt werden. Ein Beispiel für solch einen methodischen Ansatz ist die Bottom-Up-Adoption von SPLs. Bottom-Up-Adoption erfordert eine sorgfältige Analyse der zu vereinenden Softwareartefakte und wird in drei Hauptaufgaben aufgeteilt [MZB+15].

#### Identifizierung und Analyse von Features

Die erste Aufgabe von Bottom-Up-Adoption umfasst die Identifizierung und Analyse von Features, indem die zu prüfenden Softwareartefakte analysiert werden. Die durch Analyse identifizierten Features repräsentieren optionale Funktionalitäten über

alle geprüften Softwareartefakte. Andere Herangehensweisen zur Featureanalyse erlauben die Entdeckung von *Constraints*, welche die Beziehungen zwischen Features spezifizieren. Nach der Analyse der Features und deren *Constraints* kann daraus ein Feature-Modell mit einer geeigneten Beschreibungsstruktur der Features erarbeitet werden [MZB+15].

**Lokalisierung von Features**

Die zweite Aufgabe von Bottom-Up-Adoption ist die Lokalisierung von Features. Hierbei wird in den einzelnen Varianten geprüft, durch welche konkrete Implementation von Softwareartefakten die Features realisiert wurden. Im Gegensatz zur Identifizierung von Features, sind die Features bei der Lokalisierung schon bekannt. Darüber hinaus ist auch bekannt, in welchen Varianten die Features enthalten sind und in welchen nicht [MZB+15].

**Reengineering**

Reengineering ist die abschließende Phase bei der Bottom-Up-Adoption von SPLs. Hier werden die Varianten der Softwareartefakte zusammengeführt. Dazu gehört die Extraktion von wiederverwendbaren Funktionalitäten. Diese Funktionalitäten sollen dem Feature-Modell entsprechend dann von einem Feature realisiert werden [MZB+15].

Die Identifizierung und Lokalisierung von Features stellt bei der Umstellung von Clone-And-Own zu einer SPL eine wichtige Herausforderung dar. Welche geklonten Programmcodefragmente sich zu einem Feature zusammenfassen lassen ist ein Schlüsselthema dieser Arbeit. Die Codeklonerkennung als Werkzeug zur Bestimmung von geklonten Programmcodefragmenten wird daher eingehend im nachfolgendem Abschnitt erläutert.

## 2.3 Codeklone

In den vorherigen Abschnitten wurde gezeigt, dass der Wechsel von Clone-And-Own zu einer SPL besonders bei einer großen Anzahl an Projekten vorteilhaft ist. Um einen solchen Wechsel jedoch zu ermöglichen, ist die Erkennung von Codeklonen wichtig. In diesem Abschnitt wird deshalb näher darauf eingegangen, worum es sich bei Codeklonen handelt und wie sich Codeklone in Typen einteilen lassen. Abschließend wird darauf eingegangen, welche Methoden es zur Codeklonerkennung gibt und wie diese funktionieren.

### 2.3.1 Definition

Bei einem Codeklone handelt es sich um ein Programmcodefragment, welches im Quellcode mehrfach in identischer oder ähnlicher Form vorkommt. Ein Programmcodefragment wird dabei als eine Sequenz von Quellcodezeilen bezeichnet. Als Codeklone werden sowohl semantisch, wie syntaktisch gleiche Programmcodefragmente gezählt [RCK09] [RC07]. Codeklone entstehen meist durch das Kopieren bereits vorhandener Programmcodefragmente und deren Einfügen an anderer Stelle im Quellcode.

Hierbei wird noch zwischen zwei Arten des Klonens unterschieden. Die erste Art ist das Klonen kleinerer Programmteile, wie einzelne Funktionen oder Klassen. In dieser Arbeit geht es um die zweite Art, dem Klonen im großen Stil. Bei dieser Art werden ganze Systeme, beziehungsweise Subsysteme oder Programme, geklont. Besonders bei Clone-And-Own zur Entwicklung von Produktvarianten werden ganze Programme zum Teil mehrmals geklont, wodurch eine viel größere Menge geklonten Codes entsteht [Kos14].

Das Vorhandensein von Codeklonen wird generell als ein Faktor betrachtet, der die Wartung des Programmcodes erschwert. So muss zum Beispiel bei der Änderung eines Programmcodefragments geprüft werden, ob sämtliche Kopien dieses Programmcodefragments ebenfalls geändert werden müssen [HKI08]. So eine Prüfung ist jedoch zeitaufwändig und fehleranfällig, da die Erkennung von Codeklonen nicht trivial ist. Oft ist zudem nicht dokumentiert, an welchen Stellen Programmcode geklont wurde und es werden nicht immer alle Codeklone gefunden. Eine manuelle Suche ist zudem unzumutbar für sehr große Systeme, da der zeitliche Aufwand einfach zu groß wäre. Die Alternative einer automatisierten Suche nach Codeklonen ist jedoch nicht perfekt, falls Änderungen an den Kopien durchgeführt wurden [MD08]. Dazu kommt noch, dass es bei inkonsequenten Anpassungen zu einem unerwarteten Programmverhalten kommen kann, was die Wartung des Programmcodes ebenfalls erschwert [JDHW09]. Deshalb wird das Vorhandensein von Codeklonen in der Regel als ein Indikator für schlechte Softwarequalität angesehen [KSNM05]. In Studien hat sich gezeigt, dass die Codeklonrate in Softwaresystemen zwischen 7 und 23 Prozent liegt [Bak95] [BYM+98] [LLMZ04].

### 2.3.2 Typen von Codeklonen

Codeklone lassen sich in vier verschiedene Typen definieren [RCK09]. Die Definition erfolgt aufgrund der Art der Ähnlichkeit von Programmcodefragmenten, wobei die ersten drei Typen eine textuelle Ähnlichkeit umfassen und der vierte Typ für semantische Ähnlichkeit ist.

#### Type-I

Type-I-Klone umfassen alle Arten von identischen Codefragmenten und werden daher auch als exakte Kopien (exact clones) bezeichnet. Die einzigen erlaubten Unterschiede bei dieser Art von Codeklonen sind Leerzeichen, Zeilenumbrüche und Tabulatoren. Des Weiteren werden Kommentare bei dieser Art der Codeklone ebenfalls nicht berücksichtigt.

#### Type-II

Typ-II-Klone beinhalten alle Type-I-Klone und werden als Beinahe-Klone (parameter-substituted clone) bezeichnet. Sie umfassen alle syntaktisch identischen Klone, erlauben aber Unterschiede in den Bezeichnern von Feldern, Methoden, Klassen und so weiter.

#### Type-III

Type-III-Klone (near-miss clone oder auch gapped clone) beinhalten alle Type-II-

Klone und erlauben darüber hinaus strukturelle Änderungen am Programmcode. Dazu zählt das Verändern, Hinzufügen oder Löschen von Anweisungen.

**Typ-IV**

Die Type-IV-Klone sind die letzte Art und sie umfassen ausschließlich semantische Ähnlichkeiten zwischen Programmcodefragmenten. Type-IV-Klone führen eine identische Operation aus, sind aber auf verschiedene Arten implementiert worden.

In Abbildung 2.4 ist ein Beispiel für die Typen I bis III zu sehen. Die rot markierten Stellen zeigen Type-I-Klone, da diese Stellen bei dem kopierten Programmcode (b) identisch mit dem originalen Programmcode (a) sind. Die gelb markierte Stelle ist ein Type-II-Klon, wo in (b) lediglich eine Namensänderung vorgenommen wurde und die beiden Stellen syntaktisch identisch sind. Die grün markierte Stelle ist ein Type-III-Klon, bei dem hier zusätzlich zum ansonsten identischen Programmcode noch eine kleine Änderung in (b) hinzugefügt wurde.

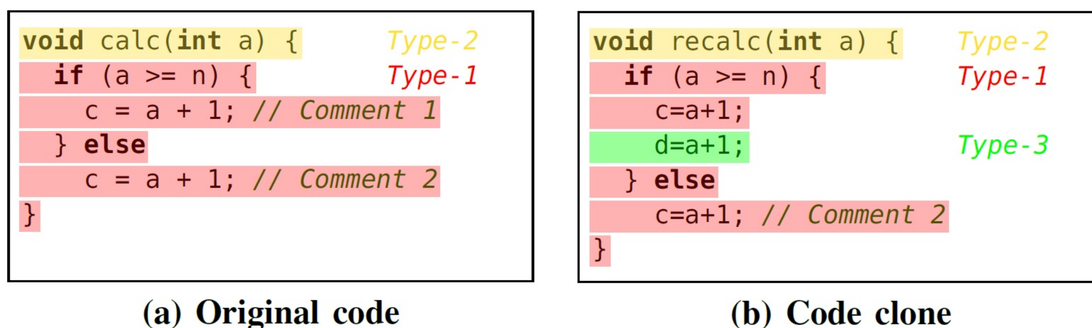



Abbildung 2.4: Verschiedene Codeklontypen im Programmcode [FMS+17]

### 2.3.3 Codeklonerkennung

Die Gründe zur Anwendung von Codeklonerkennung sind vielfältig. In dieser Arbeit liegt der Fokus darin, dass die Codeklonerkennung als Vorbereitung zur Wartung und Zusammenführung von Programmcode genutzt wird. Weitere Anwendungsgebiete sind die Aufdeckung von Softwarelizenzverstößen, das Finden von Plagiaten, die Kompression von Programmcode, die Suche in Programmcode und andere Gebiete [Kos14]. In diesem Abschnitt werden die Arten der Klonsuche und die Techniken der Codeklonerkennung vorgestellt.

#### 2.3.3.1 Arten der Klonsuche

Die Codeklonerkennung lässt sich in drei Arten einteilen, welche im Folgenden vorgestellt werden. Dabei handelt es sich um die Fragmentsuche, die Intra-System Klonsuche und die Inter-System Klonsuche [Kos14].

**Fragmentsuche**

Die Fragmentsuche zielt auf die Suche eines bestimmten geklonten Codefragments ab. Dieser Typ der Suche wird vielfältig genutzt. Dazu gehört die Suche nach Codefragmenten bei denen ein Fehler korrigiert werden muss. Des Weiteren wird mit der Fragmentsuche nach funktionierendem Programmcode gesucht, der schon eine gewünschte Funktionalität umsetzt, damit dieser wieder verwendet werden kann. Außerdem können damit auch Update-Anomalien vermieden werden, wenn ein Programmierer bereits an einem Codefragment arbeitet. Update-Anomalien entstehen, falls zwei Entwickler gleichzeitig und unabhängig am gleichen Programmcode arbeiten und Veränderungen durchführen. Zur Vermeidung von Update-Anomalien wird der betreffende Programmcode gesperrt, der auch an anderer Stelle genutzt wird [Kos14].

**Intra-System Klonsuche**

Beim zweiten Typ handelt es sich um die Intra-System Klonsuche, also nach geklonten Codefragmenten innerhalb eines Systems. Dies wird in erster Linie durchgeführt, um Möglichkeiten für das Zusammenführen von Programmcode zu finden. So kann redundanter Programmcode zusammengeführt werden [Kos14].

**Inter-System Klonsuche**

Der dritte Typ ist die Inter-System Klonsuche, also die Suche nach geklonten Codefragmenten zwischen mehreren Systemen. Im Gegensatz zur Intra-System Klonsuche sind Codeklone innerhalb eines Systems nicht von Interesse [Kos14]. Relevant für diese Arbeit ist die Inter-System Klonsuche.

Zur Erkennung von Codeklonen gibt es verschiedene Ansätze. Ein trivialer Ansatz besteht darin, dass die Suche nach Codeklonen manuell durchgeführt wird. Dazu wird der Programmcode angeschaut und es wird versucht nachzuvollziehen, woher bestimmte Programmcodefragmente stammen und worin Übereinstimmungen mit anderen Programmcodefragmenten bestehen. Wie in Abschnitt 2.3.1 schon erwähnt, ist dies jedoch nicht für sehr große Systeme durchführbar. Eine Alternative bieten automatisierte Verfahren zur Erkennung von Klonen. Diese werden nachfolgend, mit der Beschreibung ihrer Vor- und Nachteile, erklärt.

#### 2.3.3.2 Textbasierte Verfahren

Als erstes Verfahren zur Codeklonerkennung werden die textbasierten Verfahren vorgestellt. Bei textbasierten Verfahren werden die Zeichenketten des Programmcodes miteinander verglichen. Für diesen Zweck eignet sich eine Vorbehandlung der Zeichenketten, indem zum Beispiel Leerzeichen, Zeilenumbrüche und dergleichen entfernt werden sowie eine einheitliche Umwandlung auf Kleinbuchstaben vor dem Vergleich.

**Vergleich von Programmcodezeilen**

Ein einfaches textbasiertes Verfahren wird durch den Vergleich einzelner Programmcodezeilen vorgenommen. Dies wird als zeilenbasierte Technik bezeichnet und wenn unterschiedliche Programmcodezeilen identisch sind, werden sie als Codeklone erkannt [HKI08]. Der Vorteil von zeilenbasierter Codeklonerkennung liegt darin, dass sie einfach zu implementieren ist und dass die erkannten Codeklone einfach zusammengefasst werden können. Jedoch werden so Codeklone nicht erkannt, wenn sie unterschiedliche Identifikatoren benutzen, wie Namen von Funktionen, Variablen und Methoden. Textbasierte Verfahren erkennen außerdem schlecht Codeklone, wenn ein unterschiedliches Format des Programmcodes vorliegt. [HKI08].

**Token-basierte Technik**

Ein weiteres textbasiertes Verfahren ist die Token-basierte Technik. Hierbei werden die Zeichenketten anhand lexikalischer Regeln, wie Abstände und Formatierung vorbehandelt und in einzelne Tokens aufgeteilt. Bei diesen Tokens handelt es sich um die kleinsten syntaktischen Einheiten der zugrunde liegenden Programmiersprache. Dazu zählen beispielsweise Zeichenketten, wie "abc" oder Klammern. Anschließend erfolgt ein Vergleich der Tokensequenzen und identische Tokensequenz mit einer vorher bestimmten Mindestlänge werden als Codeklone erkannt. Somit ist der Token-basierte Ansatz deutlich robuster als reguläre, textbasierte Ansätze. Wenn Tokensequenzen verschiedener Programmteile miteinander identisch sind, so werden sie als Codeklone erkannt [HKI08] [RCK09]. Die tokenbasierten Verfahren haben den Vorteil, dass sie auch dann noch Codeklone erkennen, wenn ein anderes Format zwischen den zu vergleichenden Programmcodefragmenten genutzt wird. Sie haben zudem kein Problem damit, wenn die Identifikatoren unterschiedlich sind [HKI08].

Eine Schwäche von Token-basierten Techniken ist jedoch, dass Codeklone über Methodengrenzen hinweg erkannt werden, sodass ein erkannter Codeklone am Ende einer Methode anfängt und am Anfang der nachfolgenden Methode aufhört. Codeklone dieser Art sind bei der Zusammenführung von Programmcode und somit beim Umstieg von Clone-And-Own zu SPLs problematisch, da es sich streng genommen um zwei Codeklone handelt, die im Programmcode lediglich untereinander stehen. Bei Codeklonen, welche die Mindestlänge von Tokensequenzen nur erreichen, indem sie über Methodengrenzen hinweg gehen, handelt es sich somit um false-positives.

#### 2.3.3.3 Modellbasierte Verfahren

Ein relativ neuer Ansatz zum Vergleich von Programmcodefragmenten und zur Codeklonerkennung sind die modellbasierten Verfahren. Bei den modellbasierten Verfahren erfolgt der Vergleich anhand der Programmcodestrukturen. Dazu werden Metamodelle aus der Quellcodedatei extrahiert und auf Ähnlichkeit überprüft, wodurch sich auch Ähnlichkeiten in den Programmcodestrukturen feststellen lassen [Tie16] [WTS+16]. Das genaue Prinzip dieser Verfahren wird an einem konkreten implementierten Beispiel in Abschnitt 3.1 vorgestellt.

Der Vorteil dieser Verfahren liegt darin, dass sie im Vergleich zu textbasierten Verfahren die Grenzen von Programmstrukturen, wie Methoden, Funktionen und ähnlichem erkennen. So erkennen tokenbasierte Verfahren Codeklone, welche am Ende einer Methode beginnen und am Anfang der darauf folgenden Methode aufhören. Die Anzahl solcher false-Positives lässt sich durch die Verwendung von modellbasierten Verfahren möglicherweise verringern, was im Laufe dieser Arbeit untersucht werden soll.

#### 2.3.3.4 Weitere Verfahren

Neben den textbasierten und modellbasierten Verfahren zur Codeklonerkennung gibt es noch weitere Verfahren. Diese Verfahren werden jedoch nicht ausführlich in dieser Arbeit thematisiert, sondern werden nur kurz erklärt. Für ein tiefgründigeres Wissen wird daher auf die jeweiligen Quellen verwiesen.

**Program Dependency Graph**

Ein Graphen-basierter Ansatz ist der Program Dependency Graph (PDG). Bei diesem Ansatz werden die Programmabhängigkeiten in einem Graphen dargestellt. Anschließend werden die Teilgraphen auf gleiche Strukturen hin untersucht. Wenn eine gleiche Struktur in den verschiedenen Teilgraphen gefunden wird, so handelt es sich um einen Codeklone [RCK09] [Kri01].

**Abstract Syntax Tree**

Ein ähnliches Verfahren wie PDG sieht die Verwendung von abstrakten Syntaxbäumem vor, welche im Englischen als Abstract Syntax Tree (AST) bezeichnet werden. Statt eines Abhängigkeitsgraphen wird der Programmcode bei AST-basierten Techniken in einen Syntax-Baum überführt und Teilbäume mit der gleichen Struktur werden als Codeklone erkannt [BYM+98].

**Metrik-basierte Verfahren**

Die Metrik-basierten Verfahren messen verschiedene Metriken einer bestimmten Einheit, wie einer Funktion, Methode oder Klasse des Softwaresystems. Einheiten, welche dann eine ähnliche Metrik vorweisen, werden als Codeklone erkannt [HKI08].

**Hybride Verfahren**

Darüber hinaus gibt es noch hybride Verfahren, welche aus zwei oder mehr der genannten Techniken bestehen. So lassen sich beispielsweise AST-basierte Techniken mit PDG kombinieren [RCK09].

### 2.3.4 Entfernung von Codeklonen

Der Umstieg von Clone-And-Own zu SPL erfordert auch ein Entfernen und Zusammenführen der Codeklone. So müssen identifizierte Codeklone zu einer gleichen Version zusammengefasst werden, sodass sie in der SPL nur noch als ein Feature existieren.

Dies geschieht durch Refactoring und erfolgt im Anschluss nach der Identifizierung von Codeklonen. Refactoring ist ein Prozess zur Veränderung eines Softwaresystems, ohne dass deren Funktionalität geändert wird, um die Sicht des Nutzers nicht zu ändern. Es wird jedoch die interne Struktur verbessert, mit dem Ziel, die Wartbarkeit des Systems zu erleichtern. Somit wird es für die Entwickler einfacher, den Quellcode zu verstehen sowie ihn anzupassen und zu erweitern [FBB+99]. Die Änderungen am Quellcode durch Refactoring müssen jedoch korrekt durchgeführt werden, da es sonst zu Fehlern im Softwaresystem kommen kann. Eine systematisch geplante und methodische Vorgehensweise für das Refactoring ist somit notwendig. Zwei dieser Vorgehensweisen werden dazu kurz erklärt.

**Pull-Up-Method**

Ein Refactoring ist zum Beispiel Pull-Up-Method. Die Idee hinter Pull-Up-Method besteht darin, dass gleiche oder identische Methoden, die in zwei oder mehr Subklassen existieren, in die Superklasse verschoben werden. Falls die Methode nicht in allen Subklassen exakt identisch ist, so wird sie vorher zu einer gemeinsamen Version zusammengefasst. Nachdem die Methode in der Superklasse verankert wurde, wird sie in allen Subklassen entfernt [FBB+99].

**Extract Method**

Ein weiteres Refactoring-Verfahren ist Extract Method. Bei Extract Method wird eine Methode mit mehreren Funktionalitäten in mehrere neue Methoden aufgeteilt. Die neuen Methoden werden dabei so aufgeteilt, dass sie möglichst wenig Funktionalitäten beinhalten. Die extrahierten Funktionalitäten werden anschließend aus der ursprünglichen Methode entfernt [Sch13].

Das Refactoring sei an dieser Stelle nur kurz erwähnt und wird nicht näher ausgeführt, da das Refactoring von Programmcodevarianten in dieser Arbeit nicht im Fokus liegt. Der Fokus dieser Arbeit liegt auf der Erkennung von Gemeinsamkeiten zwischen Programmvarianten. Aus der Erkennung von Gemeinsamkeiten zwischen Programmvarianten werden jedoch Informationen gewonnen, auf denen ein anschließendes Refactoring aufbaut. Für weiteres tiefgreifenderes Wissen wird daher zum Beispiel auf [FBB+99],[Opd92],[SAK10],[SJF11] und [Sch13] verwiesen.

## 2.4 Zusammenfassung

In diesem Kapitel wurden die Grundlagen zu dieser Arbeit beschrieben. In Abschnitt 2.1 wurde dazu das Prinzip hinter Clone-And-Own vorgestellt sowie deren Vorteile und Nachteile von Clone-And-Own erläutert. Als Alternative zu Clone-And-Own wurde in Abschnitt 2.2 der Begriff der SPL vorgestellt. Dazu wurde an einem Beispiel die Feature-Modellierung erklärt und es wurden verschiedene Möglichkeiten zur Implementierung von SPL vorgestellt. Zur Durchführung des Umstiegs von Clone-And-Own zu SPL wurden abschließend Methoden zur Codeklonerkennung vorgestellt und das Refactoring wurde in diesem Zusammenhang noch kurz erwähnt.

# 3 Anforderungen und Konzept

Im vorherigen Kapitel wurde gezeigt, dass für die Migration von verschiedenen Produktvarianten zu einer SPL ein methodisches Vorgehen wichtig ist. Ein wichtiger Teil der Migration ist hierbei der Vergleich der Quellcodedateien der verschiedenen Produktvarianten, um Codeklone zu finden. Zur Codeklonerkennung gibt es mehrere Verfahren, welche ihre Vorteile und Nachteile haben. Hierbei werde ich den relativ neuen Ansatz der modellbasierten Codeklonerkennung genauer untersuchen. Es soll herausgearbeitet werden, wo die Stärken und wo die Schwächen dieses Verfahrens liegen. In diesem Kapitel werde ich das MBCC Framework Macumba, welches einen modellbasierten Vergleich zwischen Quellcodedateien ermöglicht, vorstellen. Dazu werde ich den Workflow und die Architektur des MBCC Frameworks beschreiben, damit deren Funktionsweise nachvollziehbar ist. Anschließend werde ich an einem Beispiel erklären, wie das Ergebnis eines Vergleichs zwischen zwei Quellcodedateien beim MBCC Framework aussieht. Das MBCC Framework soll genutzt werden, um eine Inter-System Klonsuche zwischen mehr als zwei Produktvarianten vorzunehmen. Dazu werde ich zunächst die Anforderungen definieren, anhand derer ich die Untersuchung vornehmen werde. Im Konzept werde ich abschließend diskutieren, wie die definierten Anforderungen erfüllt werden sollen.

## 3.1 MBCC Framework Macumba

Das MBCC Framework Macumba wurde im Rahmen eines dreimonatigen Projekts an der TU Braunschweig entwickelt. Das Framework ermöglicht die sprachunabhängige sowie sprachübergreifende Identifizierung von Unterschieden zwischen Programmvarianten durch Analyse der Metamodelle der Quellcodedateien. Dazu wird eine Variabilitätsanalyse auf Basis eines modellbasierten Ansatzes durchgeführt [Tie16]. Zur Erstellung dieser Metamodelle wird dafür zunächst ein Parser-Plug-in genutzt. Dem Parser wird dazu eine Konfiguration, in welcher alle Programmpfade der Quellcodedateien enthalten sind, übergeben. Anschließend parst jeder Parser die zu verarbeitenden Quellcodedateien in ein Metamodell. Danach erfolgt die Variabilitätsanalyse durch das Solver-Plug-in, indem dieser die Metamodelle vergleicht. Die Ähnlichkeit zwischen zwei Objekten wird dabei immer mit einem Wert zwischen null und eins beschrieben, wobei die eins für 100% Übereinstimmung steht und null für keine Übereinstimmung. Abschließend werden die Ergebnisse des Solvers noch in einem Report aufgewertet, damit die Ergebnisse für den Nutzer lesbar sind. Eine ausführliche Beschreibung zur genauen Funktionsweise und der einzelnen Komponenten wird in den folgenden Abschnitten 3.1.1 und 3.1.2 erklärt.

### 3.1.1 Architektur

Die Architektur des MBCC Frameworks basiert auf der Eclipse Plug-in Development Environment. Durch diese Architektur können bereits bestehende Eclipse Plug-ins, wie

zum Beispiel die typischen Eclipse Plug-ins Console, Navigator und Progress Monitor sowie die verschiedenen Editoren (z. B. Java Editor oder C++ Editor), genutzt werden, um Quellcode anzuzeigen und zu bearbeiten. Des Weiteren haben Entwickler die Möglichkeit eigene Eclipse Plug-ins zu entwickeln, welche für das MBCC Framework genutzt werden können [Tie16].

Die grundlegende Architektur des MBCC Frameworks sei in Abbildung 3.1 dargestellt. Die Hauptkomponente für die Benutzeroberfläche (Englisch: Graphical User Interface, kurz GUI) ist im oberen Teil der Abbildung in der Farbe Hellgrün dargestellt und wird parallel zur Laufzeit bereitgestellt. Weitere grün hervorgehobene Bestandteile sind typische Eclipse Plug-ins, wie die Console, der Navigator und der Progress Monitor sowie die verschiedenen Editoren (z. B. Java Editor oder C++ Editor), welche zur Anzeige und Bearbeitung von Quellcode wiederverwendet werden. Weiterhin ermöglicht es den Entwicklern eigene Eclipse Plug-ins zu entwickeln, die dann wiederum von dem MBCC Framework zur Laufzeit verwendet werden können [Tie16].

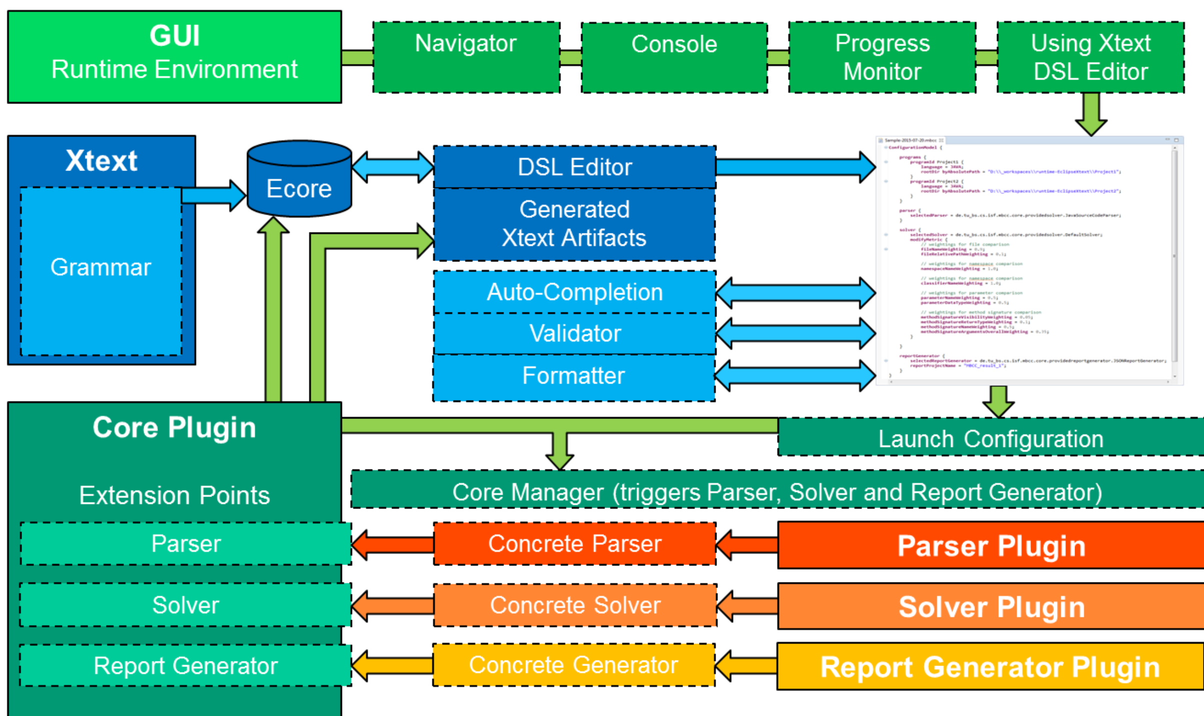



Abbildung 3.1: Architektur des MBCC Frameworks (entnommen aus [Tie16])

Des Weiteren verwendet das MBCC Framework einige Plug-ins, die mit Hilfe des Xtext-Open-Source Frameworks erstellt wurden, welches auf dem Eclipse Modeling Framework (EMF) [Vog] basiert. Die dazugehörigen Komponenten, welche in diesem Absatz beschrieben werden, sind in Abbildung 3.1 in Blau und Hellblau dargestellt. Zu Beginn der Entwicklung des MBCC Frameworks wurde zunächst eine Grammatik für Java in der Sprache Xtend erstellt. Anschließend wurde ein Ecore Metamodell zur Verwendung eines modellbasierten Ansatzes sowie die dazugehörigen Xtext-Artifakte, wie Klassen, Schnittstellen, Abhängigkeiten und mehr, mit Hilfe von Xtext generiert. Außerdem wurde auch ein Editor zur Verwendung einer domänenspezifischen Sprache

(Englisch: Domain Specific Language, kurz DSL) erstellt, um die Konfiguration einer Analyse direkt in Eclipse bearbeiten zu können. Darüber hinaus stellt das Xtext-Framework für den DSL Editor grundlegende Module, wie die Autovervollständigung, einen Validierer sowie einen Formatierer, bereit, die bei Erweiterung des Metamodells beziehungsweise der domänenspezifischen Sprache entsprechend angepasst werden müssen, um die gewünschte Funktionalität zu erzielen [Tie16].

Die Anbindung des sogenannten Core Plug-ins, welches zur Abwicklung des eigenen Workflows verwendet wird, ist im unteren linken Teil der Abbildung 3.1 dargestellt. Der genaue Ablauf des Workflows wird hier noch nicht erklärt. Dies geschieht in Abschnitt 3.1.2, da es notwendig ist, zunächst alle Komponenten der Architektur zu beschreiben. Der Workflow wird gestartet, indem die sogenannte Launch Configuration innerhalb des DSL Editors aufgerufen wird. In Abbildung 3.1 ist erkennbar, wie die Launch Configuration Bestandteil des Core Plug-ins ist. Bei Ausführung der Launch Configuration wird der Core Manager ausgeführt, welcher dann den Workflow koordiniert. Im unteren rechten Teil der Abbildung sind die Plug-Ins für den Parser, Solver und Report Generator dargestellt. Über den entsprechenden Schnittellen greift der Core Manager parallel zur Laufzeit auf die Dienste dieser Plug-Ins zu. Welche konkreten Implementierungen (Concrete Parser, Concrete Solver, Concrete Generator) dabei aufgerufen werden, hängt von der Konfiguration ab. Falls keine explizite Konfiguration für die Nutzung eines Solvers angegeben wird, so wird nach aktuellem Stand des Frameworks eine Default-Konfiguration für das Solver Plug-in ausgeführt. Anders ist dies jedoch bei den Plugins des Parsers und des Report Generators, welche derzeit jeweils eine konkrete Auswahl der Implementierung benötigen. Die detailierte Funktionsweise der eben genannten Komponenten (Parser, Solver und Report Generator) wird später im Abschnitt 3.1.2 erklärt. Aktuell unterstützt das MBCC Framework die Programmiersprache Java. Falls weitere Sprachen, wie C++ oder C, unterstützt werden sollen, so muss in erster Linie ein entsprechender Parser für die zu unterstützende Sprache bereitgestellt werden, welcher die Inhalte der jeweiligen Dateien in das MBCC Framework eigene Metamodell parst. Schlussendlich ermöglicht das Core Plug-in den über Extension Points angebundenen Plug-ins einen kontrollierten Zugriff auf das Modell, welches die Instanz des Ecore Metamodells abbildet. So ist das Parser Plug-in beispielsweise dazu berechtigt, das Modell zu modifizieren, während das Report Generator Plug-in lediglich Leseberechtigungen erhält [Tie16].

### 3.1.2 Workflow

In diesem Abschnitt wird der Ablauf des Workflows beschrieben, damit nachvollziehbar ist, wie ein Bericht mit den Ergebnissen der Variabilitätsanalyse entsteht. Dazu wird erklärt, welche Schritte abgearbeitet werden und wie die Module des Frameworks miteinander interagieren. Außerdem wird beschrieben, wie die Metriken das Ergebnis und den Ablauf der Variabilitätsanalyse beeinflussen.

Das Framework soll die Unterschiede und Gemeinsamkeiten, die zwischen Programmvarianten bestehen, mittels einer Variabilitätsanalyse ermitteln, wie es in Abbildung 3.2 dargestellt wird. Dazu wird Quellcode der verschiedenen Programmvarianten eingelesen und anschließend auf Gemeinsamkeiten und Unterschiede hin

analysiert. Die Ergebnisse der Analyse werden in einem für die Anwender verwertbaren Bericht zusammengestellt, auf dessen Grundlage entschieden werden kann, ob und wie die Programmvarianten in eine gemeinsame Softwareproduktlinie überführt werden.

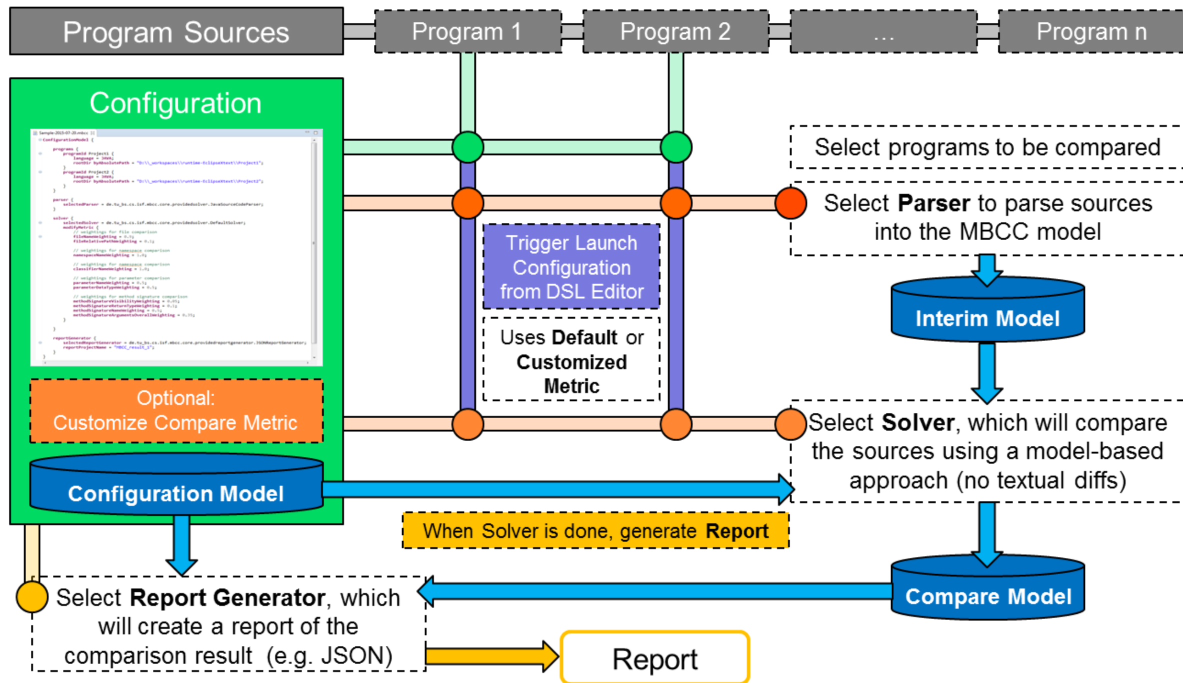



Abbildung 3.2: Workflow des MBCC Frameworks (entnommen aus [Tie16])

Im oberen Teil der Abbildung stellt der Anwender zunächst den Quellcode der zu analysierenden Programmvarianten bereit. Im nächsten Schritt muss eine Konfiguration erstellt werden, die definiert, welche Programmvarianten verglichen werden sollen und welche konkreten Parser, Report Generator und Solver genutzt werden sollen. In der Konfiguration wird ebenfalls festgelegt, welche Metrik verwendet werden soll. Der Einfluss der Metrik auf das Ergebnis der Variabilitätsanalyse wird in Abschnitt 3.1.3 erläutert.

Mit Hilfe des DSL Editors wird die Konfiguration auf ihre Gültigkeit überprüft und validiert und im Configuration Model gespeichert. Anschließend wird der Workflow über die Launch Configuration des Frameworks (mittlerer Teil der Abbildung 3.2) gestartet. Beim Start wird zunächst die Konfiguration vom Core Manager eingelesen, damit dieser weiss, welche Module (Parser, Solver und Report Generator) er zur Laufzeit bereitstellen muss. Anschließend werden die jeweiligen Module nacheinander ausgeführt [Tie16].

Als erstes Modul wird der Parser ausgeführt. Dem Parser werden zuerst die in der Konfiguration erfassten Programmpfade zum Quellcode der Programmvarianten übergeben. Jeder Parser parst dann die durch ihn verarbeitbaren Dateien in das Interim Model. Das Interim Model ist ein vereinfachtes Metamodell, welches sich auf die gängigsten Sprachelemente vieler Programmiersprachen beschränkt. Durch

die Vereinfachung werden bewusst einige sprachspezifische Features verworfen. Auf diese Art sollte es möglich sein, eine Variabilitätsanalyse von Programmvarianten für verschiedene Programmiersprachen abzubilden. So soll beispielsweise sowohl der Vergleich von einem Java Programm zu einem anderen Java Programm als auch der Vergleich von einem C++ Program zu einem anderen C++ Programmen möglich sein. Darüber hinaus sollte die Möglichkeit einer sprachübergreifenden Variabilitätsanalyse nicht ausgeschlossen werden, um zum Beispiel Programmvarianten auf Basis von Java und C++ Quellcode vergleichen zu können [Tie16].

Die Arbeitsweise einer konkreten Parser-Implementierung wird am Java Source Code Parser kurz beschrieben. Der verwendete Java Source Code Parser basiert auf dem Java Model Parser and Printer (JaMoPP). JaMoPP setzt wie das MBCC Framework auf dem Eclipse Modeling Framework auf und verwendet ein eigenes Ecore Metamodell. Der Parser liest Quellcode ein und transformiert diesen in ein JaMoPP-eigenes Modell [Hei09]. Der Java Source Code Parser führt anschließend eine Model-to-Model (M2M) Transformation aus, um die Inhalte des JaMoPP-Modells in das Metamodell des Programmcodes zu überführen [Tie16].

Die Variabilitätsanalyse durch das Solver Modul (zu Deutsch Löser bzw. Problemlöser) geschieht, nachdem alle Dateien in das Interim Model geparst wurden. Die konkrete Implementierung eines Solvers umfasst einen Vergleichsalgorithmus und hängt immer von der jeweilig verwendeten Metrik ab. Zu Beginn liest der Solver das Interim Model und das Configuration Model ein. Das Configuration Model umfasst dabei die Konfiguration der Vergleichsanalyse. In der Konfiguration wird definiert, welche Programme miteinander verglichen werden sollen, wo der entsprechende Quellcode abgelegt wurde und in welcher Programmiersprache der Quellcode vorliegt. Wenn der Nutzer die Verwendung einer speziellen Metrik wünscht, so muss er diese in der Konfiguration modifizieren. Falls keine Modifizierung der Konfiguration vorliegt, wird eine vordefinierte Standardmetrik verwendet. Alle Vergleichsaufgaben werden iterativ bearbeitet. Die Ergebnisse des Solvers werden dabei im Compare Model festgehalten. Das Compare Model lässt sich als Datenstruktur speichern, in der die Ergebnisse der Variabilitätsanalyse durch den Solver zwischengespeichert werden. Durch einfache Map-Strukturen werden die ermittelten Vergleichswerte als Wert der Map und die im Interim Model referenzierte Instanz als Schlüssel abgelegt. Nach Abschluss aller Vergleichsoperationen gibt der Solver zur weiteren Koordinierung des Workflow eine Rückmeldung an den Core Manager weiter [Tie16].

Der Core Manager ruft anschließend den Report Generator auf, damit dieser die Ergebnisse des Solvers aufbereiten kann. Der Report Generator generiert einen Bericht aus der Datenstruktur und den darin enthaltenden Ergebnissen der Variabilitätsanalyse des Compare Model. In diesem Bericht werden die Daten der Analyse visualisiert und wahlweise im HTML-Format oder JSON-Format ausgegeben. Abschließend gibt das Report-Generator-Modul eine Rückmeldung an den Core Manager, dass der Vorgang abgeschlossen ist und beendet somit den Workflow [Tie16].

Als Beispiel für einen Vergleich im HTML-Report und im JSON-Report wird der Vergleich zwischen zwei Methoden betrachtet. Aus dem Projekt ApoClock wird dafür die Methode `onCreateApp` von der Datei ApoClock.java genommen. Als Vergleichsme-

Listing 3.1: Methode onCreateApp aus ApoClock.java

```
protected void onCreateApp( ) {
  BitsLog.setLogType(BitsLog.TYPE_NONE);
  BitsApp.sWantFullscreen = true;
  BitsApp.sOrientationMode = BitsApp.ORIENTATION_PORTRAIT;
  BitsApp.sGameWidth = 480;
  BitsApp.sGameHeight = 640;
  BitsApp.sWantTitleBar = false;
  BitsApp.sMaxCirclePoints = 180;
//    BitsApp.sMaxFPS = 60;
  BitsApp.sMaxUpdate = 100;
  BitsApp.sMaxTouchPointer = 3;
//    BitsApp.sSleepMode = BitsApp.SLEEP_MODE_OFF;
  ConnectivityManager cm = (ConnectivityManager)this.getSystemService(Context.
      CONNECTIVITY_SERVICE);
  ApoClock.ni = cm.getActiveNetworkInfo();
  ApoClock.settings = this.getSharedPreferences(ApoClockConstants.PREFS_NAME, 0);
  BitsGame.getInstance().addScreen(new ApoClockPanel(1));
}
```

Listing 3.2: Methode onCreate aus ApoSnake.java

```
protected void onCreate( ) {
  BitsLog.setLogType(BitsLog.TYPE_NONE);
  BitsGame.sWantFullscreen = true;
  BitsGame.sOrientationMode = BitsGame.ORIENTATION_PORTRAIT;
  BitsGame.sGameWidth = 480;
  BitsGame.sGameHeight = 640;
  BitsGame.sWantTitleBar = false;
  BitsGame.sMaxRenderCommands = 1000;
  BitsGame.sMaxImageCount = 10;
  BitsGame.sMaxCirclePoints = 180;
  BitsGame.sMaxFPS = 60;
  BitsGame.sMaxTouchPointer = 3;
  BitsGame.sSleepMode = BitsGame.SLEEP_MODE_OFF;
  BitsGame.sWantMusic = false;
  BitsGame.sWantSound = false;
  BitsGame.sMaxFontCount = 3;
  ConnectivityManager cm = (ConnectivityManager)this.getSystemService(Context.
      CONNECTIVITY_SERVICE);
  ApoSnake.ni = cm.getActiveNetworkInfo();
  ApoSnake.settings = this.getSharedPreferences(ApoSnakeConstants.PREFS_NAME, 0);
  BitsGame.getIt().addScreen(new ApoSnakePanel(1));
}
```

thode wird dafür `onCreate` von der Datei ApoSnake.java aus dem Projekt ApoSnake genommen. Beide Projekte gehören der ApoGames-Reihe an. Die ApoGames-Reihe wird später in Abschnitt 5.2 näher erläutert, da die ApoGames-Reihe als Eingabedaten für die spätere Untersuchung genutzt werden. In Listing 3.1 wird die Methode `onCreate` abgebildet und in Listing 3.2 wird die Methode `onCreate` von der Datei ApoSnake.java abgebildet.

Es lässt sich gut erkennen, dass beide Methoden vom Aufbau her ähnlich sind. Ein Unterschied liegt jedoch darin, dass statt `BitsApp` in ApoClock.java, `BitsGame` in ApoSnake.java verwendet wird.

In Abbildung 3.3 wird der Ausschnitt aus dem HTML-Report gezeigt, in dem die Methoden `onCreate` und `onCreateApp` verglichen werden. Der Vergleichswert wird

hierbei aus Komponenten berechnet. Zum einen ist dies der Vergleichswert für “Signature Matching” mit 83,33%. Beim “Signature Matching” wird die Ähnlichkeit der Signatur zwischen beiden Methoden gemessen. Dazu gehört der Vergleich zwischen Methodenname, Rückgabetyp, Parametertypen und so weiter. Mit “Body Matching” wird die Ähnlichkeit des Methodenrumpfs zwischen beiden Methoden gemessen. Da nur zwei Zeilen übereinstimmen, beträgt die Übereinstimmung der beiden Methoden bei “Body Matching” nur 11% . Zwischen den Werten für “Signature Matching” und “Body Matching” wird ein gewichteter Mittelwert gebildet und mit “Total Matching” beziffert. Die Gewichtung bei der Berechnung des Mittelwerts lässt sich durch die Konfiguration einstellen. Eine detaillierte Vorstellung zur Berechnung der Ähnlichkeit zwischen zwei Methoden erfolgt in Abschnitt 3.1.2 Für das Beispiel wurde eine Gewichtung von 0,5 für beide gewählt, sodass der Wert für “Total Matching” 47% $(0,5 \cdot 83,33\% + 0,5 \cdot 11\%)$ beträgt.

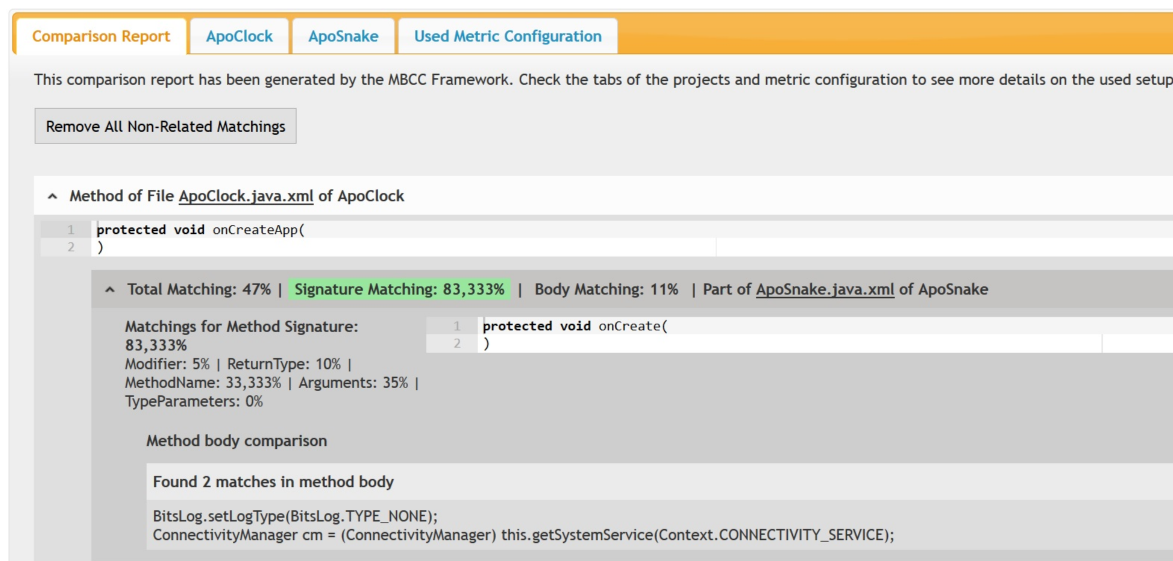



Abbildung 3.3: Ausschnitt aus einem HTML-Report

In Listing 3.3 ist der Ausschnitt des JSON-Reports mit dem gleichen Vergleich, wie im HTML-Report zu sehen. Der JSON-Report ist schwerer zu lesen als der HTML-Report, jedoch geben beide die Vergleichswerte wieder. Die Informationen zur Methode `onCreateApp` sind im oberen Teil des Listings und die Informationen zur Methode `onCreate` im unteren Teil.

Hier steht der Vergleichswert immer nach der Variable `comparison_result` und für den Vergleich zwischen `onCreateApp` und `onCreate` steht hier ein Wert von 0,833. Dies entspricht mit 83,33% Übereinstimmung dem Wert für die Signaturähnlichkeit zwischen den beiden Methoden aus dem HTML-Report.

Listing 3.3: Auszug aus einem JSON-Report

```
...
  }, {
    "parent_program" : "ApoClock",
    "parent_namespace" : "net.apogames.apoclock$",
    "parent_classifier" : "ApoClock",
    "belongs_to" : "INSTANCE",
    "name" : "onCreateApp",
    "data_type" : "void",
    "visibility" : "PROTECTED",
    "arguments" : [ ],
    "compare_values" : [ {
...
    }, {
      "parent_program" : "ApoSnake",
      "parent_namespace" : "net.apogames.aposnake$",
      "parent_classifier" : "ApoSnake",
      "belongs_to" : "INSTANCE",
      "name" : "onCreate",
      "data_type" : "void",
      "visibility" : "PROTECTED",
      "arguments" : [ ],
      "comparison_result" : 0.8333333333333333
    }, {
...
```

### 3.1.3 Bedeutung der Metrik bei der Variabilitätsanalyse

Die Verwendung von Softwaremetriken wird im Allgemeinen für das Messen bestimmter Eigenschaften verwendet, wie die statische Analyse eines Softwaresystems. Dazu können alle erdenklichen messbaren Eigenschaften bestimmt werden, die mittels Funktionen berechnet werden. Für den Vergleich von Programmvarianten kann zum Beispiel die Gesamtanzahl der Methoden, Parameter oder Dateien einer Programmvariante oder die Anzahl ähnlicher Methoden, Statements und mehr berechnet werden. Beim MBCC Framework spielen Metriken besonders bei der Ermittlung des Ähnlichkeitswerts zwischen zwei Elementen eine Rolle. Der Gesamtwert für die Ähnlichkeit zwischen zwei Objekten erfolgt durch die Aggregation der einzelnen Teilmetriken. Mit Teilmetriken ist die Ähnlichkeit zwischen den kleinsten Objektelementen gemeint. Die Gewichtung der Metriken lässt sich dabei in der Konfiguration verändern [Tie16]. Welche Kriterien bei der Ermittlung der Ähnlichkeit der jeweiligen Softwareartefakte eine Rolle spielen und wie sich die Gewichtung verändern lässt, wird im Folgenden erklärt.

#### Vergleich von Dateien, Namensräumen und Klassifizierer

Für die Berechnung der Ähnlichkeit von Dateien, Namensräumen und Klassifizierer wird der Name der genannten Objekte miteinander verglichen. Bei Dateien kann je nach gewählter Gewichtung auch noch der relative Pfad der Programmvarianten in die Berechnung mit einbezogen werden. In Listing 3.4 ist ein kommentierter Auszug aus der Konfiguration als Beispiel für die Metrik von Dateien, Namensräumen und Klassifizierer abgebildet. In der Summe muss die Gewichtung immer eins betragen.

Die Ähnlichkeit von zwei Dateien berechnet sich aus dem gewichteten Mittelwert der Teilmetriken. Im Folgenden wird die Berechnung für die Ähnlichkeit von zwei Dateien $d_1$ und $d_2$ kurz erklärt. Für diese Berechnung gibt es fünf Variablen:

Listing 3.4: Konfiguration für Dateien, Namensräumen und Klassifizierer

```
// Metrik Dateien
// Gewichtung des Dateinamen
fileNameWeighting = 0.9;
// Gewichtung des relativen Pfads der Dateien
fileRelativePathWeighting = 0.1;

// Metrik Namensraeume
// Gewichtung des Namens von Namensraeumen
namespaceNameWeighting = 1.0;

// Metrik Klassifizierer
// Gewichtung des Namens von Klassifizierer
classifierNameWeighting = 1.0;
```

- Ähnlichkeitswert zwischen $d_1$ und $d_2$ ($d$)
- Ähnlichkeitswert zwischen den Namen von $d_1$ und $d_2$ ($n$)
- Ähnlichkeitswert zwischen den relativen Pfaden von $d_1$ und $d_2$ ($r$)
- Gewichtung des Ähnlichkeitswerts zwischen den Dateinamen ($n_g$; fileNameWeighting in der Konfiguration)
- Gewichtung des Ähnlichkeitswerts zwischen den relativen Pfaden ($r_g$; fileRelativePathWeighting in der Konfiguration)

Die Formel zur Berechnung der Ähnlichkeit von zwei Dateien wäre somit:

$$d = n * n_g + r * r_g$$

Angenommen der Name von $d_1$ und $d_2$ ist gleich, womit $n = 1{,}0$ ist. Der relative Pfad ist jedoch komplett unterschiedlich, womit $r = 0{,}0$ ist. In Listing 3.4 ist $n_g$ mit 0,9 und $r_g$ mit 0,1 gewichtet. Die Werte in die Formel eingesetzt, ergibt:

$$d = 1{,}0 * 0{,}9 + 0{,}0 * 0{,}1 = 0{,}9$$

Die Ähnlichkeit zwischen $d_1$ und $d_2$ beträgt somit 0,9 (90%). Die Ähnlichkeit der Inhalte von Dateien, Namensräumen und Klassifizierern, wie zum Beispiel die Ähnlichkeit von Methoden, spielen bei der Ermittlung des Vergleichswertes keine Rolle. Die Formeln zur Berechnung aller anderen Vergleichsobjekte ist ähnlich und wird deshalb nicht explizit vorgestellt.

### Vergleich von Parametern

Beim Vergleich von Parametern untersucht der Algorithmus die jeweilige Namensgebung und den Typ der zu vergleichenden Parametern. In Listing 3.5 ist ein kommentierter Auszug aus der Konfiguration für die Gewichtung beim Vergleich von Parametern angegeben. Für dieses Beispiel wurde eine Gewichtung von 0,5 sowohl für den Parameternamen als auch für den Parametertyp gewählt.

Listing 3.5: Konfiguration für Parameter

```
1  // Metrik Parameter
2  // Gewichtung des Parameternamen
3  parameterNameWeighting = 0.5;
4  // Gewichtung des Parametertyps
5  parameterDataTypeWeighting = 0.5;
```

Listing 3.6: Konfiguration für Methoden

```
1  // Metrik Methodensignatur
2  // Gewichtung des Modifiers
3  methodSignatureOverallModifierWeighting = 0.05;
4  // Gewichtung des Rueckgabetyps
5  methodSignatureReturnTypeWeighting = 0.1;
6  // Gewichtung des Methodennamens
7  methodSignatureNameWeighting = 0.2;
8  // Gewichtung der Uebergabeparameter
9  methodSignatureArgumentsOverallWeighting = 0.35;
10 // Gewichtung der Typparametern
11 methodSignatureTypeParametersWeighting = 0.3;
12
13 // Metrik Methoden
14 // Gewichtung der Methodensignatur
15 methodTotalSignatureWeighting = 0.6;
16 // Gewichtung des Methodenrumpfs
17 methodTotalBodyWeighting = 0.4;
```

### Vergleich von Methoden

Der Vergleich von Methoden besteht aus zwei Teilen. Der erste Teil der Ähnlichkeitsbestimmung umfasst den Vergleich der Methodensignaturen und der zweite Teil umfasst den Vergleich der Methodenrümpfe.

Zur Ermittlung des Vergleichswert für die Methodensignatur erfolgt eine gewichtete Teilauswertung der Bestandteile Modifier, Rückgabetyp, Methodenname, Übergabeparameter und den Typparametern. In Listing 3.6 ist ein kommentierter Auszug aus der Konfiguration für die Gewichtung beim Vergleich von Methoden abgebildet. Dabei kann auch die Gewichtung für die Bestandteile der Methodensignatur verändert werden. Wichtig zu erwähnen ist, dass wenn zwei Methoden ohne Typparameter miteinander verglichen werden, dann wird der Ähnlichkeitswert für Typparameter auf 1,0 gesetzt [Tie16].

Bei der Ermittlung des Vergleichswertes für den Methodenrumpf wird geprüft, welche und wie viele Statements übereinstimmen, beziehungsweise welche und wie viele Statements nicht übereinstimmen. Die Statements mit Übereinstimmungen (matching statements) werden den Statements, zu denen keine Übereinstimmung gefunden wurde (non-related statements), gegenübergestellt. Anhand der Methoden aus den Listings 3.7 und 3.8 soll eine beispielhafte Berechnung des Ähnlichkeitswertes erfolgen. Die Methode aus Listing 3.7 besitzt insgesamt sieben Statements im Methodenrumpf (siehe Nummerierung in den Kommentaren) und die Methode aus Listing 3.8 besitzt vier Statements. Die Bezugsgröße zur Berechnung entspricht immer der Anzahl der

Statements der Methode mit der größeren Anzahl an Statements. In diesem Beispiel wäre die Bezugsgröße 7, da die Methode aus Listing 3.7 mit insgesamt sieben Statements mehr hat als die Vergleichsmethode. Die Anzahl der übereinstimmenden Statements beträgt drei, die Statements `stmt1;`, `if (cond)` und `stmt3;`, welche in beiden Methoden vorkommen. Die Anzahl der Nicht-Übereinstimmungen berechnet sich aus der Bezugsgröße minus der Anzahl der Übereinstimmungen, was in diesem Beispiel vier ist. Somit ergibt sich für dieses Beispiel eine Ähnlichkeit von 42,86% (entspricht 3/7) für den Methodenrumpf. Dagegen stimmen 57,14% (entspricht 4/7) nicht überein. Die Ähnlichkeit zwischen der Methodensignatur beträgt für dieses Beispiel 100% (1,0), da die Methodensignaturen gleich sind [Tie16].

Nachdem die Werte für die Ähnlichkeit von Methodenrumpf und Methodensignatur berechnet wurden, wird aus diesen beiden Werten der gewichtete Mittelwert gebildet. In Listing 3.6 wurde zum Beispiel eine Gewichtung von 0,4 für den Methodenrumpf und eine Gewichtung von 0,6 für die Methodensignatur gewählt. Somit ergibt sich ein Ähnlichkeitswert von rund 0,77 (0,6 * 1,0 + 0,4 * 0,4286) für diesen Methodenvergleich.

Listing 3.7: Programmvariante 1 (angelehnt an[Tie16])

```
1  void methodA () {
2    stmt1;       // 1 matching
3    stmt2;       // 2 non-related
4    if (cond)    // 3 matching
5    {
6      stmt3;     // 4 matching
7    }
8    else         // 5 non-related
9    {
10     stmt4;     // 6 non-related
11   }
12   stmt5;       // 7 non-related
13   // stmt6;    // - non-related
14 }
```

Listing 3.8: Programmvariante 2 (angelehnt an [Tie16])

```
1  void methodA () {
2    stmt1;       // 1 matching
3    // stmt2;    // - non-related
4    if (cond)    // 2 matching
5    {
6      stmt3;     // 3 matching
7    }
8    // else      // - non-related
9    // {
10   // stmt4;    // - non-related
11   // }
12   // stmt5;    // - non-related
13   stmt6;       // 4 non-related
14 }
```

## 3.2 Anforderungen

Die Überführung von Programmvarianten, die durch Clone-And-Own entstanden sind, zu einer SPL ist eine umfassende Aufgabe, zu der ein methodisches Vorgehen notwendig ist. Eine zentrale Aufgabe ist dabei die Inter-System Klonsuche über alle Programmvarianten hinweg. Mit einer geeigneten Codeklonerkennung wird festgestellt, welche Programmcodefragmente an welchen Stellen der Programmvarianten kopiert und wiederverwendet wurden. Dazu werde ich in dieser Arbeit den relativ neuen Ansatz einer modellbasierten Codeklonerkennung näher untersuchen und evaluieren. Für diese Untersuchung werde ich das vorgestellte MBCC Framework nutzen, welches eine modellbasierte Variabilitätsanalyse über Programmvarianten ermöglicht. Bei der Untersuchung werde ich ausarbeiten, wo die Stärken und Schwächen des MBCC Frameworks bei der Erkennung von Codeklonen bei einer Inter-System Klonsuche liegen und ich werde ausarbeiten, wie die Ergebnisse des MBCC Frameworks einen Entwickler bei der Zusammenführung von geklonten Programmcode unterstützten kann. In diesem Abschnitt werde ich die Anforderungen definieren, um eine Inter-System Klonsuche mit dem MBCC Framework durchführen zu können und wie sich die Ergebnisse dieser

Analyse geeignet darstellen lassen.

Dazu gilt es zunächst, geeignete Eingabedaten zu bestimmen, die einer Inter-System Klonsuche über mehrere Programmvarianten hinweg gerecht werden. Ein wichtiges Kriterium für die Eingabedaten ist, dass diese durch die Anwendung von Clone-And-Own entstanden sind, da die Zusammenführung von Programmvarianten, die aus Clone-And-Own entstanden sind, zu einer SPL als Hintergrundmotivation zu dieser Arbeit steht.

Für die Inter-System Klonsuche über mehrere Programmvarianten ist es notwendig, dass das MBCC Framework einen Vergleich über mehrere Programmvarianten unterstützt. Dazu werde ich prüfen, inwiefern ein N-weiser von N Programmvarianten bereits vom MBCC Framework unterstützt wird. Wenn notwendig, werde ich Anpassungen oder Erweiterungen im MBCC Framework vornehmen, damit ein N-weiser Vergleich ermöglicht wird.

Ebenfalls wichtig ist die Art der Darstellung der Ergebnisse der Vergleichsanalyse. Hierfür werde ich prüfen, inwiefern die bereits vorhandenen Reports genutzt werden können oder ob eine andere Art der Darstellung genutzt werden soll. Für eine andere Darstellung könnten die Daten aus dem HTML-Report oder dem JSON-Report verwendet werden, was geprüft wird.

Die Sicht des Entwicklers, welcher die Projekte zu einer SPL zusammenführen soll, ist hier am wichtigsten, weshalb es sinnvoll wäre, dem Entwickler zu zeigen, in welchen Teilen sich die Projekte ähneln. Dazu sollte es möglich sein, die Ähnlichkeit auf verschiedenen Granularitätsebenen einzusehen. Dazu gehört die Ähnlichkeit auf:

- Projektebene (wie ähnlich sind sich zwei Projekte)
- Packageebene (wie ähnlich sind sich zwei Packages aus zwei unterschiedlichen Projekten)
- Klassenebene (wie ähnlich sind sich zwei Klassen aus zwei unterschiedlichen Projekten)
- Methodenebene (wie ähnlich sind sich zwei Methoden aus zwei unterschiedlichen Projekten)

Die Ähnlichkeit zweier Methoden ist die niedrigste Granularitätsstufe und wird bereits vom MBCC Framework unterstützt. Ebenfalls unterstützt wird ein Vergleich von Dateien, also den Klassen der Programmvarianten und der Vergleich von Namensräumen, also den Packages der Programmvarianten. Wie jedoch aus Abschnitt 3.1.3 hervorgeht, wird beim MBCC Framework für den Vergleich von Dateien lediglich die Namensgebung und der relative Pfad als Vergleichskriterium herangezogen. Für Namensräume wird sogar nur die Namensgebung als Vergleichskriterium herangezogen. Die Inhalte von Dateien und Namensräumen werden jedoch nicht betrachtet. Zur Berechnung der Ähnlichkeit auf Klassenebene, Packageebene und auch Projektebene sollen jedoch deren Inhalte für die Berechnung der Ähnlichkeit verwendet werden, da eine spätere Zusammenführung zu einer SPL über die Zusammenführung der Funktionalitäten erfolgen würde. Deshalb ist es erforderlich, dass für die Berechnung der Ähnlichkeit auf den höheren Ebenen eine Aggregation über die jeweils darunter liegende Ebene erfolgen muss. So soll sich die

Ähnlichkeit auf Klassenebene aus der Ähnlichkeit auf Methodenebene berechnen und so weiter. Zur Durchführung der Aggregation muss daher eine geeignete Aggregationsmethode bestimmt werden, um einen akkuraten Vergleich auf den einzelnen Ebenen gewährleisten zu können. Die herausgearbeitete Aggregationsmethode wird anschließend für die Vergleichsanalyse implementiert.

## 3.3 Konzept

Nachdem ich im vorherigen Abschnitt die Anforderungen definiert habe, gilt es in diesem Abschnitt zu zeigen, wie die Anforderungen erfüllt werden sollen. Dazu werde ich ein Konzept ausarbeiten und vorstellen, um zu zeigen, welche Vorgehen bei der Erfüllung der Anforderungen genutzt werden und auch zu begründen, warum diese Vorgehen genutzt werden. Ich werde darauf eingehen, welche Anpassungen und Erweiterungen ich am MBCC Framework vornehmen werde, um mit dem MBCC Framework eine Inter-System Klonsuche durchführen zu können. Anschließend erkläre ich, welche Darstellung ich zur Präsentation der Ergebnisse aus der Variabilitätsanalyse nutzen werde und wie ich mit dieser Darstellung eine statistische Analyse der Ergebnisse durchführen werde. Abschließend werde ich noch erklären, welche Aggregationsmethode für die Berechnung der Ähnlichkeit für die verschiedenen Granularitätsebenen genutzt wird. Die Berechnung wird zudem an einem Beispiel näher erläutert.

### 3.3.1 Anpassung und Erweiterung des MBCC Frameworks

Das MBCC Framework soll einen N-weisen Vergleich zwischen N verschiedenen Projekten vornehmen. Die Zusammenführung von geklonten Programmvarianten steht im Kontext dieser Arbeit und somit wird auch nur eine Inter-System Klonsuche durchgeführt. Es sollen also nur Quellcodedateien zwischen den verschiedenen Programmen auf Gemeinsamkeiten untersucht werden. Die Analyse soll also nicht zwischen Quellcodedateien stattfinden, die zum selben Programm gehören.

Hierzu werden die Quellcodedateien aus einer Programmvariante zu einem Projekt zusammengefasst und ein Vergleich zwischen zwei Quellcodedateien soll nur stattfinden, wenn diese aus zwei unterschiedlichen Projekten stammen. Ein Vergleich zwischen zwei Methoden soll zudem nur einmal durchgeführt werden, da die Ähnlichkeit von zwei Methoden symmetrisch ist.

Als nächstes wird erklärt, wie ich die Daten aus dem Vergleich der Programmvarianten analysieren werde. Hierfür werde ich das Ergebnis des JSON-Reports nutzen, da sich die strukturierten Daten der JSON-Datei gut zur weiteren Verarbeitung eignen. Damit die Daten des JSON-Reports auch genutzt werden können, werde ich am JSON-Report eine Änderung vornehmen. Wie in Abschnitt 3.1.2 bei der Erklärung des JSON-Reports beschrieben, betrachtet der JSON-Report für die Ähnlichkeit von Methoden nur die Ähnlichkeit der Methodensignatur. Ähnlich wie beim HTML-Report soll bei der Erzeugung des JSON-Reports ein gewichteter Mittelwert aus der Ähnlichkeit der Methodensignatur und der Ähnlichkeit des Methodenrumpfes für die Ähnlichkeit zwischen zwei Methoden gebildet werden.

Die Daten des JSON-Reports sind bereits strukturiert, somit bietet es sich an, die Daten zur Analyse in eine relationale Datenbank zu importieren. Durch die Auslagerung der Analyse in eine relationale Datenbank kann zudem ein Problem mit der Skalierung in Eclipse umgangen werden. Dieses Problem trifft bei der Analyse von mehreren großen Projekten auf. Das MBCC Framework erfordert einen großen Arbeitsspeicher und es war nicht möglich, alle fünf Projekte aus der Fallstudie, welche die Eingabedaten stellen, auf einmal zu vergleichen. Es ist jedoch möglich, die Analyse immer mit zwei Projekten durchzuführen. Die Ergebnisse aller Analysen werden dann am Ende in der relationalen Datenbank zusammengeführt, sodass ein Überblick über alle Projekte auf einmal möglich ist. Ein weiterer Vorteil bei der Nutzung einer Datenbank besteht darin, dass die Vergleichsdaten aus der Datenbank für statistische Analysen und für die Erstellung von Reports genutzt werden können.

Mit Hilfe der relationalen Datenbank werde ich mehrere Auswertungen mit der Structured Query Language (SQL) erstellen. Diese Auswertungen sollen verschiedene Statistiken zur Ähnlichkeit auf den einzelnen Ebenen (Projektebene, Packageebene, Klassenebene und Methodenebene) wiedergeben. Anhand dieser Auswertungen werde ich Stärken und Schwächen des MBCC Frameworks, bei der Erkennung von Codeklonen, ermitteln. Außerdem wird es dem Entwickler, welcher die Programmvarianten zuammenführen soll, mit diesen Auswertungen ermöglicht, zu sehen, an welchen Stellen und in welchem Umfang sich die Programmvarianten ähneln. Des Weiteren soll ein geeigneter Client genutzt werden, um auf die Analyseergebnisse, welche auf der relationalen Datenbank ausgeführt werden, zuzugreifen.

### 3.3.2 Aggregation der Vergleichsdaten

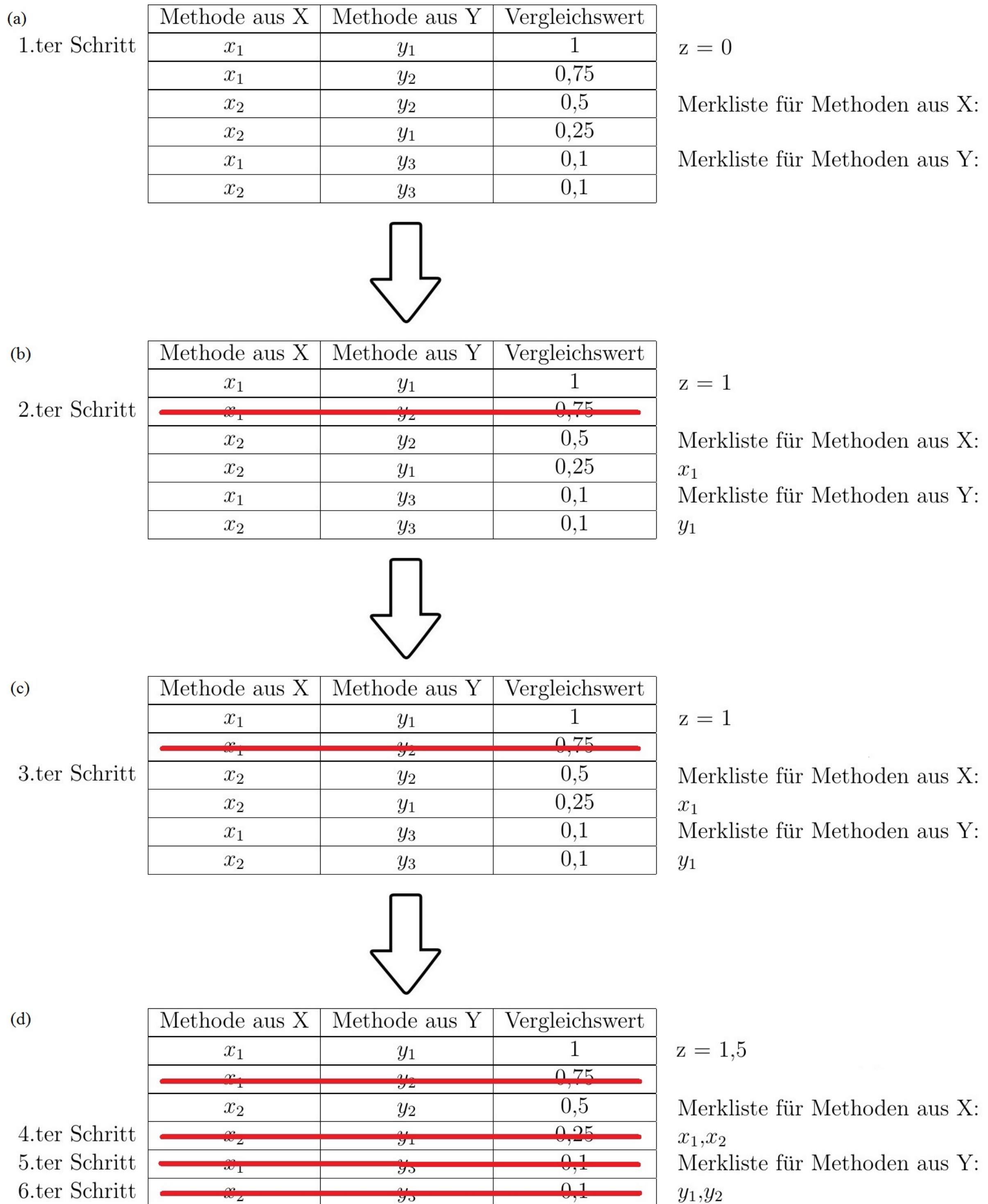
In diesem Abschnitt werde ich vorstellen, wie die von mir ausgearbeitete Aggregation der Ergebnisdaten aus der Variabilitätsanalyse des MBCC Framework durchgeführt wird. Wie in Abschnitt 3.2 beschrieben, können die Ergebnisse der Variabilitätsanalyse nicht vollständig genutzt werden, da für die Dateiähnlichkeit und Namensraumähnlichkeit nicht die Inhalte und somit die Funktionalitäten verglichen werden. Zur Berechnung der Ähnlichkeit der verschiedenen Granularitätsebenen werden deshalb immer die Daten der jeweils niedrigeren Ebene aggregiert. So wird die Ähnlichkeit auf Klassenebene aus der Ähnlichkeit der Methodenebene berechnet und so weiter. Die Daten zur Methodenähnlichkeit sind dabei schon vorhanden.

Neben den Ergebnisdaten zur Methodenähnlichkeit gibt es auch Ergebnisdaten zur Ähnlichkeit von Parametern. Parameter sind zwar auch Teil der Klassen, werden aber nicht für die Berechnung der Ähnlichkeit auf Klassenebene hinzugezogen. Dies liegt daran, dass die Zusammenführung von Funktionalitäten im Vordergrund bei der Überführung von durch Clone-And-Own entstandene Produktvarianten zu einer SPL steht. Die Namen von Parametern und der Typen sind für die Zusammenführung von Funktionalitäten eher uninteressant. Hier sind die Statements der Methoden und somit auch die Ähnlichkeit von Methoden zueinander entscheidener für die Bestimmung der Ähnlichkeit von Klassen. Die Berechnung zur Ähnlichkeit von zwei Klassen erfolgt somit auf Basis der Ähnlichkeit der enthaltenen Methoden.

Zur Beschreibung der allgemeinen Berechnung ist die Klasse X aus dem Projekt A sowie die Klasse Y aus dem Projekt B gegeben. Die Klasse X beinhaltet die Methoden $x_1$, $x_2$, ..., $x_n$ und die Klasse Y beinhaltet die Methoden $y_1$, $y_2$, ..., $y_n$. Die Berechnung auf Klassenebene soll folgendermaßen ablaufen. Die Ähnlichkeit einer Klasse X aus Projekt A zur Klasse Y aus Projekt B geschieht, indem der Durchschnitt der höchsten Vergleichswerte jeder Methode aus X mit einer Methode aus Y gebildet wird. Die Auswahl der Vergleichswerte soll dabei *greedy* erfolgen. Die Vergleiche werden anhand des Vergleichswertes in absteigender Reihenfolge betrachtet. Anfangs wird der höchste Vergleichswert zwischen zwei Methoden aus X und Y genommen. Die hierfür genommenen Methoden werden als x (Methode aus der Klasse X) und y (Methode aus der Klasse Y) bezeichnet. Dieser Wert wird mit dem zweithöchsten Vergleichswert addiert. Der zweithöchste Vergleichswert darf jedoch nicht zwischen x und einer anderen Methode aus Y, beziehungsweise aus y und einer anderen Methode aus X sein, da sowohl x als auch y als Methoden in der Summierung enthalten sind. Dies wird solange fortgesetzt, bis alle Vergleiche der Klassen X und Y betrachtet wurden. Der aufsummierte Wert wird dann durch die Anzahl der Methoden der Klasse, welche die höhere Anzahl an Methoden hat, geteilt. Der so entstandene Wert zwischen null bis eins gibt dann die Ähnlichkeit der Klassen X und Y wieder. Dieses Vorgehen wird solange wiederholt bis jede Klasse aus Projekt A mit jeder Klasse aus Projekt B verglichen wurde. Sollte es weitere Projekte geben, so wird dieses Vorgehen für alle Projekte durchgeführt, sodass jede Klasse aus einem Projekt mit allen Klassen aus einem anderen Projekt verglichen wird. Aufgrund der Symmetrie der Vergleiche (die Ähnlichkeit von X zu Y ist exakt gleich zur Ähnlichkeit von Y zu X) wird jeder Vergleich zwischen zwei Klassen nur einmal durchgeführt. Die Berechnung zur Ähnlichkeit auf Packageebene und Projektebene erfolgt analog und wird deshalb nicht näher vorgestellt.

Anhand von Abbildung 3.4 wird der Ablauf zur Berechnung der Ähnlichkeit zwischen zwei Klassen an einem Beispiel erklärt. Die Tabellen in der Abbildung haben drei Spalten und in einer Zeile steht jeweils ein Vergleich einer Methode aus X mit einer Methode aus Y. In der ersten und zweiten Spalte stehen die Methoden aus X und Y, die miteinander verglichen werden. In der dritten Spalte steht der Vergleichswert, der aus dem Vergleich der beiden Methoden aus X und Y resultiert. Die Zeilen der Tabellen sind dabei so angeordnet, dass die Vergleiche in absteigender Reihenfolge in Bezug auf den Vergleichswert sortiert sind. Klasse X aus dem Projekt A beinhaltet die zwei Methoden $x_1$ und $x_2$. Klasse Y aus dem Projekt B beinhaltet die drei Methoden $y_1$ und $y_2$ und $y_3$. Die Vergleichswerte liegen im Bereich 0 bis 1, sodass ein Vergleichswert von 1 eine Übereinstimmung von 100% bedeutet. Rechts von den Tabellen steht noch die Variable z, welche ein Zwischenspeicher ist, indem die Vergleichswerte aufaddiert werden. Außerdem stehen dort noch die Merklisten, in denen festgehalten wird, welche Methoden X und Y schon bei der Berechnung mit einbezogen wurden. Anfangs sind die Merklisten leer und die Variable z hat den initialen Wert null.

Im ersten Schritt (Teil (a) der Abbildung 3.4) wird der Vergleich zwischen $x_1$ und $y_1$ betrachtet, da dieser Vergleich den höchsten Vergleichswert hat. Der Vergleichswert von 1 wird zu z addiert. Zudem werden beide Methoden jeweils in den Merklisten vermerkt. Im zweiten Schritt (Teil (b) der Abbildung 3.4) wird der Vergleich zwischen $x_1$ und $y_2$



(a) 1.ter Schritt

| Methode aus X | Methode aus Y | Vergleichswert |
|---|---|---|
| $x_1$ | $y_1$ | 1 |
| $x_1$ | $y_2$ | 0,75 |
| $x_2$ | $y_2$ | 0,5 |
| $x_2$ | $y_1$ | 0,25 |
| $x_1$ | $y_3$ | 0,1 |
| $x_2$ | $y_3$ | 0,1 |

z = 0

Merkliste für Methoden aus X:

Merkliste für Methoden aus Y:

(b) 2.ter Schritt

| Methode aus X | Methode aus Y | Vergleichswert |
|---|---|---|
| $x_1$ | $y_1$ | 1 |
| ~~$x_1$~~ | ~~$y_2$~~ | ~~0,75~~ |
| $x_2$ | $y_2$ | 0,5 |
| $x_2$ | $y_1$ | 0,25 |
| $x_1$ | $y_3$ | 0,1 |
| $x_2$ | $y_3$ | 0,1 |

z = 1

Merkliste für Methoden aus X: $x_1$

Merkliste für Methoden aus Y: $y_1$

(c) 3.ter Schritt

| Methode aus X | Methode aus Y | Vergleichswert |
|---|---|---|
| $x_1$ | $y_1$ | 1 |
| ~~$x_1$~~ | ~~$y_2$~~ | ~~0,75~~ |
| $x_2$ | $y_2$ | 0,5 |
| $x_2$ | $y_1$ | 0,25 |
| $x_1$ | $y_3$ | 0,1 |
| $x_2$ | $y_3$ | 0,1 |

z = 1

Merkliste für Methoden aus X: $x_1$

Merkliste für Methoden aus Y: $y_1$

(d) 4.ter Schritt, 5.ter Schritt, 6.ter Schritt

| Methode aus X | Methode aus Y | Vergleichswert |
|---|---|---|
| $x_1$ | $y_1$ | 1 |
| ~~$x_1$~~ | ~~$y_2$~~ | ~~0,75~~ |
| $x_2$ | $y_2$ | 0,5 |
| ~~$x_2$~~ | ~~$y_1$~~ | ~~0,25~~ |
| ~~$x_1$~~ | ~~$y_3$~~ | ~~0,1~~ |
| ~~$x_2$~~ | ~~$y_3$~~ | ~~0,1~~ |

z = 1,5

Merkliste für Methoden aus X: $x_1$,$x_2$

Merkliste für Methoden aus Y: $y_1$,$y_2$

Abbildung 3.4: Ablauf zur Berechnung der Klassenähnlichkeit

betrachtet. Die Methode $x_1$ ist jedoch schon vermerkt, da der Vergleich von $x_1$ und $y_1$ in die Berechnung eingeflossen ist. Damit kann der Vergleich zwischen $x_1$ und $y_2$ nicht mehr in die Berechnung einfließen, obwohl dieser Vergleich den zweithöchsten Vergleichswert besitzt. Im dritten Schritt (Teil (c) der Abbildung 3.4) wird der Vergleich zwischen $x_2$ und $y_2$ betrachtet. Beide Methoden sind in den Merklisten noch nicht vermerkt, sodass der Vergleichswert zu z addiert wird. Anschließend werden beide Methoden in den Merklisten vermerkt. Die Schritte vier bis sechs (Teil (d) der Abbildung 3.4.) bringen keine weitere Veränderung mit sich, da immer mindestens eine Methode schon vermerkt ist. Der Wert aus z wird abschließend durch die Anzahl der Methoden der Klasse mit der höheren Anzahl an Methoden geteilt. Im Beispiel ergibt sich so ein Wert von 1,5/3 = 0,5. Somit sind sich die Klassen X und Y zu 50% ähnlich.

Die Vergleichswerte der einzelnen Ebenen werden danach statistisch ausgewertet. Aus wissenschaftlicher Sicht bietet sich dazu die Verwendung von Histogrammen an. Auch die Verteilung, wie sich die Ähnlichkeit einer Ebene aus der jeweils niedrigeren Ebene zusammensetzt, ist dafür interessant. Eine geeignete Darstellung für den Entwickler, welcher die Projekte zu einer SPL zusammenfassen soll, wäre die Auflistung aller relevanten Ähnlichkeiten. Mit solch einer Auflistung kann der Entwickler feststellen, für welche Methoden, Klassen etc. eine Zusammenführung sinnvoll ist.

## 3.4 Zusammenfassung

In diesem Kapitel wurde das MBCC Framework Macumba, welches eine modellbasierte Variabilitätsanalyse zum Vergleich von Programmvarianten durchführen kann, vorgestellt. Dazu wurde zunächst die Architektur und der Workflow des MBCC Frameworks vorgestellt. Anhand der Metriken wurde zudem gezeigt, welche Daten für die Variabilitätsanalyse betrachtet werden. Danach wurden die Anforderungen definiert, die für die Untersuchung eines modellbasierten Klonerkennungsverfahrens erfüllt sein müssen. Für die Untersuchung soll das MBCC Framework genutzt werden. Deshalb bezogen sich die Anforderungen in erster Linie auf dieses. Im Konzept wurde dann erklärt, wie ich die definierten Anforderungen erfüllen werde. Dazu wurden zunächst die Eingabedaten, anhand derer eine Variabilitätsanalyse mit dem MBCC Framework vorgenommen wird, bestimmt. Zudem wurde beschrieben, welche Anpassungen und Erweiterungen ich am MBCC Framework vornehmen werde. Abschließend wurde noch der Ablauf zur Berechnung der Ähnlichkeit auf Klassenebene beispielhaft vorgestellt.

# 4 Implementierung

In diesem Kapitel werde ich beschreiben, wie ich die im Konzept vorgestellten Lösungsansätze umgesetzt habe. Ich beschreibe zunächst, wie der N-weise Vergleich von N Programmvarianten beim MBCC Framework abläuft und welche Anpassungen ich vorgenommen habe. Wie in Abschnitt 3.3.1 beschrieben, werde ich eine relationale Datenbank für die Darstellung der Ergebnisse aus der Variabilitätsanalyse und zur statistischen Analyse der Ergebnisse aus der Variabilitätsanalyse nutzen. Als relationale Datenbank werde ich MySQL nutzen und ich werde den Importprozess beschreiben, mit dem ich die Vergleichsdaten in die MySQL-Datenbank überführen werde. Abschließend wird erklärt, wie die Vergleichsdaten in der MySQL-Datenbank aufbereitet werden und wie ich in der MySQL-Datenbank die Aggregation der Vergleichsdaten, zur Ermittlung der Ähnlichkeit auf den verschiedenen Granularitätsebenen, durchführen werde.

## 4.1 N-weiser Vergleich im MBCC Framework

Der N-weise Vergleich von N Programmvarianten ist eine wichtige Vorbedingung, um eine Inter-System Klonsuche mit dem MBCC Framework vornehmen zu können. In diesem Abschnitt wird daher dargelegt, wie ein solcher Vergleich abläuft und welche Anpassungen ich vorgenommen habe.

Vor dem Vergleich werden alle zu vergleichenden Programmvarianten mittels der Konfiguration des MBCC Framework jeweils als ein Projekt definiert. Für das MBCC Framework ist ein Projekt eine Sammlung von Softwareartefakten. Die Konfiguration kann so angepasst werden, dass für jede zu vergleichende Programmvariante genau ein Projekt in der Konfiguration definiert wird. Darüber hinaus ist es jedoch auch möglich, eine Programmvariante über mehrere Projekte zu verteilen. Die Verteilung einer Programmvariante über mehrere Projekte hinweg ist für besonders große Programmvarianten wichtig, da wie in Abschnitt 3.3.1 beschrieben, das MBCC Framework viel Arbeitsspeicher erfordert. Der Bedarf an Arbeitsspeicher wächst dabei mit der Größe und der Anzahl der zu vergleichenden Projekte. Durch das Aufteilen einer Programmvariante in mehrere Projekte kann dieses Problem umgangen werden, indem immer nur ein Teil aller Projekte auf einmal verglichen wird und die Ergebnisse der einzelnen Vergleiche am Ende zusammengeführt werden. Die im Konzept vorgestellte Herangehensweise, dass ein Vergleich zwischen zwei Methoden nur einmal durchgeführt werden soll, wird vom MBCC Framework bereits durchgeführt, welches nachfolgend kurz erklärt wird.

Für den Vergleich werden zu Beginn zwei Listen initialisiert, diese werden als “linke Liste” und als “rechte Liste” bezeichnet. In der linken Liste werden alle Elemente aufgeführt, die schon miteinander verglichen wurden und in der rechten Liste sind alle

Elemente, die noch zu vergleichen sind. Zu Beginn des Vergleichs ist die linke Liste leer und in der rechten Liste sind alle zu vergleichenden Elemente aus allen Projekten enthalten. Elemente sind dabei alle zu vergleichenden Objekte (Methoden, Klassen und so weiter).

Im ersten Schritt wird das erste Element aus der rechten Liste entfernt und in die linke Liste gespeichert. Beim zweiten Schritt wird aus der rechten Liste wieder das erste Element ausgewählt, dieses Element wird als "rechtes Element" bezeichnet. Das rechte Element wird in diesem Schritt jedoch nicht aus der rechten Liste entfernt, da dieser Schritt erst im nächsten Durchlauf des Algorithmus durchgeführt wird. Im dritten Schritt wird das rechte Element mit allen Elementen in der linken Liste verglichen.

Für eine Inter-System Klonsuche soll ein Vergleich zwischen zwei Elementen nur stattfinden, wenn sie zu unterschiedlichen Projekten gehören. Als Anpassung habe ich deshalb eine Kontrolle eingeführt, die vor jedem Vergleich stattfindet. So wird vor jedem Vergleich kontrolliert, ob der Name des Projekts, zu dem das rechte Element gehört, mit dem Namen des Projekts, zu dem das aktuell gewählte Element aus der linken Liste gehört, übereinstimmt. Ein Vergleich findet nur statt, wenn die Projektnamen unterschiedlich sind, ansonsten nicht. Außerdem findet ein Vergleich zwischen zwei Elementen nur statt, wenn sie auch zur gleichen Kategorie gehören. So wird eine Methode nur mit einer anderen Methode verglichen, jedoch nicht mit einer Klasse. Nachdem das rechte Element mit allen Elementen aus der linken Liste verglichen wurde, beginnt der Algorithmus wieder mit dem ersten Schritt. Dies wird solange wiederholt, bis alle Elemente in der linken Liste sind, womit auch die rechte Liste leer ist. Somit wurden alle Elemente genau einmal miteinander verglichen.

## 4.2 Import der Vergleichsdaten

In diesem Abschnitt werde ich vorstellen, wie der Importprozess der Daten aus dem JSON-Report von mir realisiert wurde. Die Auswahl der Technologien zur Realisierung des Importprozesses erfolgte so, dass möglichst viele Schritte des Importprozesses als Erweiterung des MBCC Framework realisierbar sind. Bei der Auswahl des Datenbanksystems habe ich darauf geachtet, dass dieses frei für die wissenschaftliche Verwendung zur Verfügung steht und ohne Erwerbung einer kommerziellen Lizenz nutzbar ist. Zudem sollen alle notwendigen Funktionen zur Aufbereitung der Vergleichsdaten bereitgestellt sein. Daher habe ich MySQL als relationale Datenbank ausgewählt, da MySQL als freie Open Source Version verfügbar ist. MySQL ist auf allen gängigen Betriebssystemen, wie Unix, Mac OS X, Linux und Windows lauffähig [MyS]. Als Client zur Verwaltung der MySQL-Datenbank nutze ich phpMyAdmin. Bei phpMyAdmin handelt es sich um ein Freeware Softwaretool, geschrieben in Hypertext Preprocessor (PHP) und es ermöglicht die Administration einer MySQL Datenbank [php]. Der Dateiimport über phpMyAdmin unterstützt kein JSON-Format, somit müssen die Vergleichsdaten erst in ein Format umgewandelt werden, welches von phpMyAdmin unterstützt wird. Ein geeignetes Format zum Import von Daten in MySQL bietet das Dateiformat für Comma-separated values (CSV). Der Vorteil von CSV liegt darin, dass die tabellarisch angeordnete Datenstruktur der Struktur einer in MySQL gespeicherten Tabelle entspricht.

Bevor jedoch der Importprozess beschrieben wird, wird noch kurz beschrieben, wie die Vergleichsergebnisse des JSON-Report von mir angepasst wurden. Wie in Abschnitt 3.1.2 dargestellt, wird für den JSON-Report nur der Wert für die Signaturähnlichkeit zwischen zwei Methoden für die Ermittlung der Methodenähnlichkeit genutzt. Die Ähnlichkeit des Methodenrumpfes, wie es beim HTML-Report der Fall ist, wird beim JSON-Report nicht zur Ermittlung der Methodenähnlichkeit genutzt. Der JSON-Report wird deshalb von mir so angepasst, dass sowohl die Ähnlichkeit der Signatur als auch die Ähnlichkeit des Rumpfes zweier Methoden als gewichteter Mittelwert zur Ermittlung der Ähnlichkeit zwischen zwei Methoden genutzt wird. Des Weiteren wird der JSON-Report von mir so angepasst, dass dieser nur noch die Ergebnisse der Methodenähnlichkeit umfasst, da nur diese für die Aufbereitung in der MySQL-Datenbank genutzt werden. Die Ähnlichkeiten der Namensräume und so weiter fallen somit aus dem Bericht raus.

Zu Beginn des Importprozesses müssen die Daten des JSON-Reports in das CSV-Format umgewandelt werden. Bei dieser Umwandlung wird noch ein Zwischenschritt getätigt, da die Datenstruktur des JSON-Reports von der tabellarisch angeordneten Datenstruktur für CSV abweicht. Dieser Zwischenschritt umfasst die Umwandlung von JSON zur erweiterbaren Auszeichnungssprache Extensible Markup Language (XML). Die Struktur der XML-Daten wird anschließend mit XSL Transformation (XSLT) so verändert, dass diese der tabellarisch angeordneten Datenstruktur bei CSV entsprechen. Die umstrukturierten XML-Daten werden anschließend in das CSV-Format überführt.

In Abbildung 4.1 wird der Prozess zum Import der Vergleichsdaten des JSON-Reports in die MySQL-Datenbank dargestellt. Der erste Schritt ist die Umwandlung des JSON-Reports in das XML-Format. Zur Umwandlung des JSON-Reports wurde Macumba um diese Funktion erweitert. Mit den Funktionen der Klasse `JSONtoXML` wird der JSON-Report in das XML-Format überführt. Die Umwandlung erfolgt mit der Funktion `org.json.XML.toString()`.

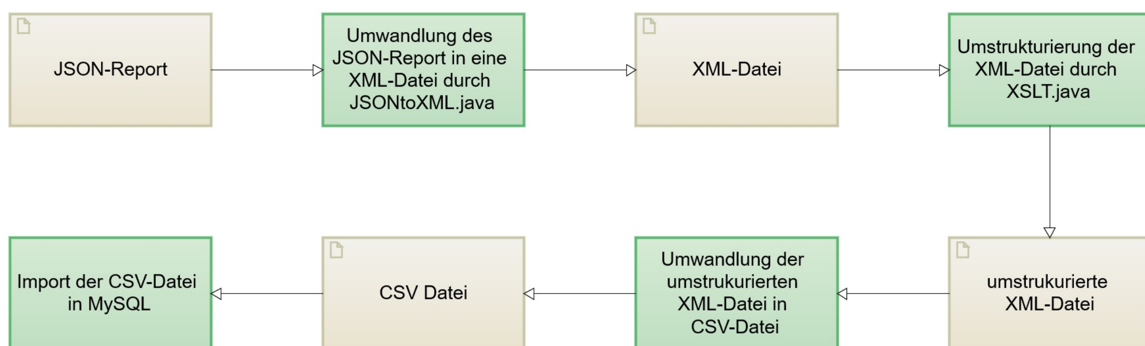



Abbildung 4.1: Prozess zum Import der Vergleichsdaten

Anschließend werden die XML-Dateien mit XSL Transformation (XSLT) umgestellt. Die Umstellung erfolgt mit der Klasse `XSLT`. Das dafür verwendete XSL-Stylesheet wird in

Listing 4.1: XSL-Stylesheet zur Umwandlung in die gewünschte Struktur

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

    <xsl:template match="/">
        <ROWSET>
    <xsl:for-each select="//comparison_result">
      <ROW>
        <type><xsl:value-of select="name(../..)"/></type>
        <parent_program><xsl:value-of select="../../parent_program"/></parent_program>
        <parent_namespace><xsl:value-of select="../../parent_namespace"/></
            parent_namespace>
        <parent_classifier><xsl:value-of select="../../parent_classifier"/></
            parent_classifier>
        <parent_name><xsl:value-of select="../../name"/></parent_name>
        <comparison_parent_program><xsl:value-of select="../parent_program"/></
            comparison_parent_program>
        <comparison_parent_namespace><xsl:value-of select="../parent_namespace"/></
            comparison_parent_namespace>
        <comparison_parent_classifier><xsl:value-of select="../parent_classifier"/></
            comparison_parent_classifier>
        <comparison_name><xsl:value-of select="../name"/></comparison_name>
        <comparison_result><xsl:value-of select="."/></comparison_result>
      </ROW>
    </xsl:for-each>
  </ROWSET>
    </xsl:template>
</xsl:stylesheet>
```

Listing 4.1 aufgeführt. Daran lässt sich auch die gewünschte Struktur erkennen, sodass ein Vergleich alle Informationen beider Vergleichsobjekte enthält sowie den Vergleichswert zwischen diesen beiden Objekten. So werden für jeden Vergleich (umschlossen mit dem Tag `ROW`) alle wichtigen Informationen, wie `type`, `parent_program` und so weiter festgehalten. In Abschnitt 4.3 erfolgt die genaue Erklärung, welche Informationen in den jeweiligen Tags stehen, da die Tags bereits dem Datenmodell in der MySQL-Datenbank entsprechen.

Die so entstandene XML-Datei "XML_for_CSV" wird dann letztendlich durch die Klasse `XMLtoCSV` in eine CSV-Datei umgewandelt und die CSV-Datei wird in die MySQL-Datenbank importiert.

## 4.3 Aufbereitung der Vergleichsdaten

In diesem Abschnitt werde ich darstellen, wie ich die Daten zur Vergleichsanalyse in der MySQL-Datenbank aufbereite. Die Aufbereitung der Vergleichsdaten ist notwendig, damit verschiedene Analysen der Vergleichsdaten durchgeführt werden können. Die Ergebnisse dieser Analysen werden in der Evaluation diskutiert.

### 4.3.1 Datenmodell der Vergleichsdaten

Nachdem die Vergleichsdaten des JSON-Reports als CSV-Datei vorliegen, wird die CSV-Datei zur Aufbereitung der Daten in die MySQL-Datenbank importiert. Das Datenmodell entspricht einer Tabelle, die alle relevanten Informationen aus dem JSON-Report

beinhaltet. Daher gibt es auch nur eine Basistabelle. Die Erklärung zu den Spalten ist wie folgt:

- type - gibt an, um welchen Vergleich es sich handelt (aktuell werden nur noch Methodenvergleiche aus dem JSON-Report übernommen)
- parent_program - Name des Projekts, aus dem das erste Vergleichsobjekt stammt
- parent_namespace - gibt den Namespace des ersten Vergleichsobjekts an (den Namen des Package zu dem das Objekt gehöhrt)
- parent_classifier - Name der Vergleichsdatei, in der sich das erste Vergleichsobjekt befindet
- parent_name - Name des ersten Vergleichsobjekts
- parent_datatyp - Datentyp des ersten Vergleichsobjekts
- parent_arguments - eine Liste aller Argumente und deren Datentypen vom ersten Vergleichsobjekts
- comparison_parent_program - Name des Projekts, aus dem das zweite Vergleichsobjekt stammt
- comparison_parent_namespace - wie parent_namespace für das zweite Vergleichsobjekt
- comparison_parent_classifier - wie parent_classifier für das zweite Vergleichsobjekt
- comparison_name - wie parent_name für das zweite Vergleichsobjekt
- comparison_datatyp - wie parent_datatyp für das zweite Vergleichsobjekt
- comparison_parent_arguments - wie parent_arguments für das zweite Vergleichsobjekt
- comparison_result - Vergleichswert der beiden Vergleichsobjekte zwischen null und eins

Zwar sind die Daten auf diese Weise nicht normalisiert und sie sind redundant gespeichert, dies ist jedoch für die weitere Aufbereitung der Daten unproblematisch. Ich habe bewusst ein einfaches Modell mit nur einer Tabelle gewählt, denn im Gegensatz zu üblichen Datenbanken werden an den Daten keine weiteren Veränderungen mehr vorgenommen. So werden typische Datenbankoperationen, wie das Ändern, Löschen und Hinzufügen von Zeilen nach dem Import der Daten nicht durchgeführt. Außerdem müssen keine zusätzlichen `JOIN`-Operationen durchgeführt werden, da das Datenbankschema nur eine Basistabelle umfasst. Für die Datenaufbereitung werden aus dieser Basistabelle weitere Tabellen gebildet. Diese zusätzlichen Tabellen haben neben den bereits vorhandenen Informationen aus der Basistabelle weitere Informationen, die aus den bereits vorhandenen Informationen berechnet werden. Man könnte auch Views benutzen. Es ist jedoch sinnvoll, die Ergebnisse jeder `SELECT`-Anweisung zu speichern, um deren erneute Berechnung zu vermeiden, da einige `SELECT`-Anweisungen mehr als einmal verwendet werden. Ein weiterer Vorteil dieser Implementierungsstrategie besteht darin, dass die Anweisungen relativ klein und übersichtlich bleiben, im Vergleich zu den Anweisungen bei der Nutzung von Views, da bei der Verwendung von Views möglicherweise Sub-`SELECT`-Anweisungen notwendig sind.

Listing 4.2: SQL-Anweisung zur Berechnung der Anzahl der Methoden pro Klasse

```
1  CREATE TABLE all_methodnames_count
2
3  SELECT
4  `projectname`,
5  `filename`,
6  COUNT(`methodname`) AS methodcount
7  FROM
8  `all_methodnames`
9  GROUP BY
10 `projectname`,
11 `filename`
```

### 4.3.2 Berechnung der Ähnlichkeiten

Die Berechnung der Ähnlichkeiten für alle Ebenen erfolgt komplett in der MySQL-Datenbank. Dadurch ist es nicht notwendig, die Berechnung mit einer anderen Programmiersprache durchzuführen, weil die Daten für die Berechnung sonst immer aus der MySQL-Datenbank exportiert werden müssten und die Ergebnisse wieder importiert werden müssten. Die Ähnlichkeit auf Methodenebene kann sofort aus der Basistabelle `all_method_comparisons` herausgelesen werden, da der Vergleichswert für den Vergleich von zwei Methoden in der Spalte `comparison_result` bereits vorliegt. Nachfolgend werden nicht alle Teilschritte detailliert beschrieben. Die zur Nachvollziehung wichtigsten Schritte werden anhand der aufgeführten Listings erklärt.

Zur Berechnung der Ähnlichkeit auf Klassenebene wird zuerst ermittelt, wie viele Methoden jede einzelne Klasse umfasst. Dazu wird ein `COUNT()` über alle Methodennamen für jede Klasse durchgeführt. In Listing 4.2 wird dazu über die Spalten `projectname` (beinhaltet den Namen des Projekts zu dem die Klasse gehört) und `filename` (Name der Klasse) eine Gruppierung mit `GROUP BY` durchgeführt.

Für jedes Tupel wird die Anzahl der Methoden pro Klassen hinzugefügt. Ein Tupel umfasst immer zwei Methoden aus zwei unterschiedlichen Klassen, weshalb für beide Klassen die Anzahl der Methoden notwendig ist. Die Spalte `leftmethodcount` gibt dabei die Anzahl der Methoden der ersten Klasse an, die in `filename` steht, welche sozusagen auf der linken Seite der Tabelle steht. Gleiches gilt für die Spalte `rightmethodcount` und die zweite Klasse, aus der die Vergleichsmethode stammt, `comparison_filename`, welche auf der rechten Seite der Tabelle stehen. Zudem werden die Daten so angeordnet, dass auf der linken Seite immer die Klasse mit der geringeren Anzahl an Methoden steht, im Vergleich zur anderen Klasse.

Im nächsten Schritt erfolgt die eigentliche Auswahl der Methodenvergleiche für die Ähnlichkeit von Klassen. Die dazugehörige SQL-Anweisung ist in Listing 4.3 abbgebildet. Die Tabelle `method_comparison_left` wird mit `ORDER BY` sortiert, sodass alle Methodenvergleiche zwischen zwei Klassen untereinander stehen, in absteigender Reihenfolge der Vergleichswerte. Diese Sortierung ist notwendig, damit die richtige Auswahl der Vergleichswerte erfolgt. Wie im Konzept vorgestellt, wird als erstes immer der Methodenvergleich mit dem höchsten Vergleichswert für die Berechnung zur Ähnlichkeit zwischen zwei Klassen vorgenommen. Die beiden Methoden, zu denen dieser Vergleichs-

wert gehört, werden dabei in einer Art Liste vermerkt, sodass kein Vergleichswert mehr genommen wird, bei dem die gemerkten Methoden beteiligt sind. Die Realisierung der Merklisten erfolgt mit den temporären Variablen `@list_left_temp` und `@list_left`. Die Variablen sollen sich in einer fortlaufenden Zeichenkette die bereits ausgewählten Methoden der linken Seite merken. Für die rechte Seite machen dies die Variablen `@list_right_temp` und `@list_right`. Für die temporären Variablen wird die sortierte Tabelle von oben nach unten durchgegangen. Mit folgender Anweisung wird festgestellt, ob sich die obere Zeile in den Spalten `filename` und `comparison_filename` von der aktuellen Zeile unterscheidet: `IF(@last_data = ‘filename‘ AND @last_datac = ‘comparison_filename‘, @list_left, ’’)`. Falls ja, wird die Merkliste für die bereits verwendeten Methoden geleert, andernfalls wird sie weiter fortgeführt. Die Spalte `@is_max` stellt ein Flag dar, mit der festgestellt wird, ob der Vergleichswert dieser Zeile für die Berechnung der Klassenähnlichkeit genutzt werden soll. Mit `LIKE CONCAT(’%’,‘methodname‘,’|%’)` wird geprüft, ob die linke Methode schon in der Merkliste für die linken Methoden enthalten ist. Analog erfolgt auch die Prüfung für die rechte Methode. Wenn sowohl die linke als auch die rechte Methode noch nicht in den Merklisten enthalten sind, so wird die Zeile mit `@is_max` geflagt und beide Methoden werden in die jeweilige Merkliste `@list_left` beziehungsweise `@list_right` aufgenommen.

Listing 4.3: SQL-Anweisung zur Festlegung der relevanten Tupel für die Berechnung der Klassenähnlichkeit

```
1  CREATE TABLE method_comparison_notice
2  
3  SELECT
4  t.*,
5  @list_left_temp := IF(@last_data = ‘filename‘ AND @last_datac = ‘comparison_filename‘,
       @list_left, ’’) AS list_left_temp,
6  @list_right_temp := IF(@last_data = ‘filename‘ AND @last_datac = ‘comparison_filename‘,
       @list_right, ’’) AS list_right_temp,
7  @is_max := IF(@list_left_temp LIKE CONCAT(’%’,‘methodname‘,’|%’) OR @list_right_temp
       LIKE CONCAT(’%’,‘comparison_methodname‘,’|%’), 0, 1) AS is_max,
8  @list_left := IF(@is_max = 1, CONCAT(@list_left_temp,‘methodname‘,’|’),@list_left_temp)
       AS list_left,
9  @list_right := IF(@is_max = 1, CONCAT(@list_right_temp,‘comparison_methodname‘,’|’),
       @list_right_temp) AS list_right,
10 @last_data := ‘filename‘,
11 @last_datac := ‘comparison_filename‘
12 FROM
13 ‘method_comparison_left‘ t,
14 (SELECT
15   @is_max := 0,
16   @list_left := ’’,
17   @list_right := ’’,
18   @last_data := ’’,
19   @last_datac := ’’,
20   @list_left_temp := ’’,
21   @list_right_temp := ’’
22 ) r
23 ORDER BY
24 ‘projectname‘,
25 ‘filename‘,
26 ‘comparison_projectname‘,
27 ‘comparison_filename‘,
28 ‘comparison_result‘ desc
```

In Listing 4.4 wird gezeigt, wie die Ähnlichkeit von zwei Klassen abschließend berechnet wird. Dazu wird die Tabelle `method_comparison_notice` gruppiert und es wird für jede Kombination von Klassen aus zwei unterschiedlichen Projekten der Vergleichswert

Listing 4.4: SQL-Anweisung zur Berechnung der Klassenähnlichkeit

```
CREATE TABLE file_comparison_greedy

SELECT
`projectname`,
`filename`,
`comparison_projectname`,
`comparison_filename`,
SUM(`comparison_result`) / MAX(`rightmethodcount`) AS comparison_result
FROM
`method_comparison_notice`
WHERE
`is_max` = 1
GROUP BY
`projectname`,
`filename`,
`comparison_projectname`,
`comparison_filename`
```

gebildet. Hierfür werden nur Zeilen genommen, die mit `@is_max` geflagt sind. Der aufsummierte Vergleichswert `comparison_result` wird durch die Anzahl der Methoden der rechten Klasse geteilt, da diese Klasse mehr oder zumindest gleich viele Methoden wie die linke Klasse hat. Dadurch gehen indirekt die übrig gebliebenen Methoden der rechten Klasse mit null als Wert in die Aufsummierung ein, was durchaus Sinn ergibt. So können sich zum Beispiel zwei Klassen nur zu maximal 50% ähneln, wenn eine Klasse doppelt so viele Methoden hat, wie die andere Klasse.

Somit stehen in der Tabelle `file_comparison_greedy` alle Vergleichswerte zwischen zwei Klassen aus zwei unterschiedlichen Projekten. Die Berechnungen für die Ähnlichkeit zwischen zwei Packages (aus zwei unterschiedlichen Projekten) aus der Klassenähnlichkeit sowie die Berechnung zur Ähnlichkeit zwischen Projekten aus der Packageähnlichkeit erfolgt analog und wird deshalb nicht zusätzlich erklärt.

## 4.4 Zusammenfassung

In diesem Kapitel wurde beschrieben, wie der N-weise Vergleich zwischen N Programmvarianten beim MBCC Framework abläuft. Um eine Inter-System Klonsuche zu garantieren, habe ich eine Anpassung vorgenommen. Durch diese Anpassung am MBCC Framework werden zwei Vergleichsobjekte vom gleichen Projekt nicht miteinander verglichen. Außerdem wurde der Prozess für den Import der Vergleichsdaten aus dem JSON-Report hin zur MySQL-Datenbank in seinen einzelnen Schritten beschrieben. Dazu habe ich den JSON-Report so angepasst, dass nur Methodenvergleiche aufgelistet werden, da nur Methodenvergleiche für die weitere statistische Analyse benötigt werden. Bei der Ermittlung der Ähnlichkeit von zwei Methoden habe ich eine Anpassung vorgenommen, sodass sowohl die Ähnlichkeit der Methodensignatur als auch die Ähnlichkeit des Methodenrumpfes zur Ermittlung der Ähnlichkeit von zwei Methoden genutzt werden, da vorher nur die Ähnlichkeit der Methodensignatur genutzt wurde. Anschließend habe ich den Importprozess vorgestellt. Beim Importprozess wird der JSON-Report in eine CSV-Datei umgewandelt und die CSV-Datei wird in die MySQL-Datenbank importiert. Abschließend wurde dargestellt, wie ich die Vergleichsdaten in der MySQL-Datenbank

aufbereitet habe, sodass ich die Vergleichsdaten für die statistische Analyse nutzen kann. Anhand der SQL-Anweisungen wurde dazu beispielhaft erklärt, wie sich die Ähnlichkeit von Klassen aus der Ähnlichkeit ihrer Methoden berechnen lässt.

# 5 Evaluation

In diesem Kapitel erfolgt die Evaluation des in Abschnitt 3.3 vorgestellten Konzepts. Ziel dieser Evaluation ist, zu zeigen, wie die Ergebnisse der Variabilitätsanalyse des MBCC Frameworks einem Entwickler bei der Zusammenführung von durch Clone-And-Own entstandenen Programmcodes zu einer SPL unterstützen können. Dazu werde ich für die ausgewählte Fallstudie die Ergebnisse der Variabilitätsanalyse mit den Ergebnissen einer textbasierten Codeklonerkennung vergleichen. Außerdem werde ich diskutieren, ab welchem Ähnlichkeitswert ein Codeklon vorliegt und welchen Einfluss die gewählte Metrik beim Aufspüren bestimmter Codeklone hat.

## 5.1 Fragestellung

Das Ziel dieser Evaluation ist es, zu zeigen, inwiefern sich die Variabilitätsanalyse des MBCC Framework für eine Inter-System Klonsuche zwischen mehreren Programmvarianten eignet und wie deren Ergebnisse für die Zusammenführung von verschiedenen durch Clone-And-Own entstandenen Programmvarianten genutzt werden können. Außerdem wird diskutiert, wie verlässlich die Ergebnisse der Variabilitätsanalyse sind. Im Fokus der Evaluierung steht dabei die Untersuchung der Ergebnisse anhand der Eingabedaten, die in Abschnitt 5.2 vorgestellt werden. Die Evaluation wird sich dabei auf drei Forschungsfragen konzentrieren, welche nachfolgend aufgeführt und erklärt werden.

Frage 1: *Wie gut ist die Codeklonerkennung der modellbasierten Variabilitätsanalyse des MBCC Frameworks im Vergleich zu einer textbasierten Codeklonerkennung?*

Mithilfe der ersten Forschungsfrage soll geklärt werden, ob die modellbasierte Codeklonerkennung des MBCC Frameworks, im Vergleich zu einer textbasierten Codeklonerkennung, Codeklone besser oder schlechter erkennt. Dazu werden die Ergebnisse der Variabilitätsanalyse mit den Ergebnissen eines textbasierten Codeklonerkennungstools gegenübergestellt. Zur Gegenüberstellung wird FeatureIDE genutzt, welches ein open-source Framework für feature-orientierte Softwareentwicklung basierend auf Eclipse ist [TKB+14]. FeatureIDE bietet ein optionales Plugin, den Copy/Paste Detector (CPD), zum Erkennen von Codeklonen [CPD]. Zur Beantwortung dieser Forschungsfrage wird als erstes ein qualitativer Vergleich durchgeführt. Als zweites wird zur Beantwortung dieser Forschungsfrage ein quantitativer Vergleich der Codeklonerkennung durchgeführt. Für den quantitativen Vergleich werden die Messgrößen Precision, Recall und F-Measure für beide Codeklonerkennungsverfahren ermittelt und miteinander verglichen. Für den qualitativen und quantitativen Vergleich werden für verschiedene Ähnlichkeitswerte zufällige Methodenvergleiche ausgewählt. Für diese Methodenvergleiche wird bewertet, ob deren Zusammenführung sinnvoll ist. Die Bewertung wird anhand der Ermittlung,

ob es sich bei den zu vergleichenden Methoden um Codeklone handelt, erfolgen. Mit CPD wird für die Methoden mit Codeklonen ermittelt, ob dieser Codeklon von CPD erkannt wird. Darüber hinaus sollen Ursachen ermittelt werden, warum ein Codeklon erkannt wurde oder warum kein Codeklon erkannt wurde.

Frage 2: *Welchen Einfluss haben verschiedene Metriken bei der Ermittlung der Ähnlichkeit von zwei Methoden?*

Mit der zweiten Forschungsfrage soll geklärt werden, wie sich die Ergebnisse bei der Verwendung von verschiedenen Metriken, wie sie in Abschnitt 3.1.3 vorgestellt wurden, ändern. Insbesondere möchte ich untersuchen, ob bestimmte Metriken nützlichere Ergebnisse hervorbringen als andere. Die Nützlichkeit der Ergebnisse hängt davon ab, ob ein Entwickler, welcher die verschiedenen Programmvarianten zu einer SPL zusammenfassen soll, mehr zusammenfassbare Methoden findet. Dazu wird die Variabilitätsanalyse mit verschiedenen Metriken durchgeführt. Insbesondere wird dabei darauf eingegangen, wie die Größe von Methoden (also die Anzahl der Statements pro Methode) das Ergebnis bei der Verwendung von verschiedenen Metriken beeinflusst. Zur Beantwortung dieser Frage wird die Variabilitätsanalyse mit drei Metriken durchgeführt. Einer ausgeglichenen Metrik, in der Methodensignatur und Methodenrumpf gleich gewichtet sind und zwei Metriken, in denen jeweils ein Kriterium deutlich höher gewichtet ist. Mit jeder Metrik werden die Ähnlichkeitswerte auf jeder Granularitätsebene (Methodenebene, Klassenebene, Packageebene und Projektebene) gemessen und miteinander verglichen. Außerdem wird anhand ausgewählter qualitativer Vergleiche gezeigt, ob bestimmte Metriken sich für das Aufspüren von bestimmten Codeklonen besser eignen als andere Metriken.

Frage 3: *Ab welchem Ähnlichkeitsgrad ist die Zusammenführung von geklonten Programmelementen sinnvoll?*

Zur Beantwortung der dritten Forschungsfrage wird eine Empfehlung ausgesprochen, ab welchem Ähnlichkeitsgrad eine Zusammenführung von Objekten sinnvoll erscheint. Dazu wird anhand des qualitativen Vergleichs und anhand der quantitativen Ergebnisse (Precision, Recall und F-Measure), welche bei der Beantwortung der ersten Frage durchgeführt wurden, festgestellt, ab welchem Ähnlichkeitsgrad ein Codeklon vorliegt und es wird diskutiert ob eine Zusammenführung ab diesem Ähnlichkeitsgrad sinnvoll erscheint.

## 5.2 Fallstudie

In diesem Abschnitt werden die Eingabedaten, mit denen ich mein Konzept zur Untersuchung der Variabilitätsanalyse des MBCC Framework prüfen werde, vorgestellt. Für die modellbasierte Variabilitätsanalyse sollen mehrere Projekte aus der ApoGames-Reihe als Fallstudie genutzt werden. Bei der ApoGames-Reihe handelt es sich um eine von Dirk Aporius in Java entwickelte Sammlung von Spielen für die Plattform Android [Apo]. Aus dieser Sammlung von Spielen werden insgesamt fünf Projekte miteinander verglichen. Dazu gehören die Projekte ApoClock, ApoDice, ApoMonoAndroid, ApoSna-

ke und myTreasureAndroid. Diese Projekte sind geeignet, da sie durch die Anwendung von Clone-And-Own entstanden sind. Es kann somit angenommen werden, dass zwischen den Projekten viele geklonte Programmcodefragmente existieren, deren Funktionalitäten sich gut für die Zusammenführung in eine SPL eignen. Daher soll die modellbasierte Variabilitätsanalyse des MBCC Frameworks mit diesen Eingabedaten durchgeführt werden. In Tabelle 5.1 wird für die fünf genannten Projekte der ApoGames-Reihe aufgeführt, wie viele Java-Dateien (entspricht der Anzahl der Klassen) jedes Projekt hat sowie die gesamte Anzahl an Leerzeilen, Kommentarzeilen und Programmcodezeilen für jedes Projekt. Die Messung der Zahlen erfolgte mit dem Programm CLOC [CLO].

| Projekt | Dateien | Leerzeilen | Kommentarzeilen | Programmcodezeilen |
|---|---|---|---|---|
| ApoClock | 28 | 862 | 422 | 3584 |
| ApoDice | 19 | 537 | 381 | 2504 |
| ApoMonoAndroid | 24 | 971 | 417 | 6446 |
| ApoSnake | 19 | 580 | 401 | 2946 |
| myTreasureAndroid | 27 | 1069 | 742 | 5322 |

Tabelle 5.1: Statistik zu den Projekten der ApoGames-Reihe

## 5.3 Durchführung

In diesem Abschnitt zeige ich, wie ich die in Abschnitt 5.2 vorgestellte Fallstudie durchgeführt habe. Dazu zeige ich zunächst, wie ich die Eingabedaten mit der Variabilitätsanalyse des MBCC Frameworks analysiert habe. Danach zeige ich, wie ich die Ergebnisse der Variabilitätsanalyse mit der MySQL-Datenbank aufbereitet habe. Abschließend erkläre ich noch, welche Schritte ich zur Beantwortung der Forschungsfragen vorgenommen habe.

Bei der Variabilitätsanalyse des MBCC Frameworks wurde eine große Menge an Arbeitsspeicher benötigt. Deshalb war es notwendig, die Analyse der fünf ApoGames-Projekte aufzuteilen. Anstatt dass alle fünf Projekte auf einmal mit der Variabilitätsanalyse des MBCC Framework verglichen wurden, werden immer zwei Projekte in einem Schritt verglichen, sodass bei fünf Projekten insgesamt 10 Schritte notwendig sind. In jedem Schritt wird der JSON-Report für jeden Vergleich von zwei Projekten erstellt und über den Importprozess in die MySQL-Datenbank importiert. Die Aufspaltung der Variabilitätsanalyse in einzelne Schritte ist dabei unproblematisch, da das Ergebnis zwischen zwei Projekten sich nicht ändert, wenn noch weitere Projekte miteinander verglichen werden. Lediglich der Importprozess muss für jeden Schritt einzeln durchgeführt werden, sodass alle Analysevorgänge zwischen den Projekten in einer Tabelle der MySQL-Datenbank gespeichert sind. Nachdem alle Vergleichsdaten in der MySQL Tabelle vorhanden sind, werden die einzelnen Skripte zur Aufbereitung der Daten nacheinander ausgeführt.

Dieser Vorgang wird für insgesamt drei Varianten von Metriken zur Bestimmung der Ähnlichkeit von zwei Methoden durchgeführt. Die drei Metrikvarianten wurden

Listing 5.1: verwendete Gewichtung innerhalb der Methodensignatur

```
methodSignatureOverallModifierWeighting = 0.05;
methodSignatureReturnTypeWeighting = 0.1;
methodSignatureNameWeighting = 0.5;
methodSignatureArgumentsOverallWeighting = 0.35;
methodSignatureTypeParametersWeighting = 0.0;
```

bewusst stark unterschiedlich gewählt, um deutlich zu zeigen, wie sie das Ergebnis beeinflussen. Diese drei Varianten umfassen deshalb ein Verhältnis von Methodensignaturähnlichkeit zu Methodenrumpfähnlichkeit von 70:30, 50:50 und 30:70. Im weiteren Verlauf der Evaluation werden die verschiedenen Metriken mit "70:30-Metrik", "50:50-Metrik" und "30:70-Metrik" bezeichnet, wobei die erste Zahl für die Gewichtung der Methodensignatur steht und die zweite Zahl für die Gewichtung des Methodenrumpfes steht. Dabei setzt sich die Metrik für die Methodensignatur für alle drei Varianten, wie in Listing 5.1 einsehbar, zusammen. Diese Metrik aus Listing 5.1 entspricht der Voreinstellung für die Gewichtung der Methodensignatur, falls in der Konfiguration des MBCC Frameworks keine benutzerspezifische Gewichtung angegeben wird. Der Wert für `methodSignatureTypeParametersWeighting` wird auf 0,0 gelassen, da die Gewichtung des Typparameters das Ergebnis sonst verfälscht. Wie in Abschnitt 3.1.3 beschrieben, wird für Methoden ohne Typparameter immer 100% Ähnlichkeit für den Typparameter verrechnet, falls Methoden ohne Typparameter miteinander verglichen werden. In der Fallstudie gibt es jedoch keine Methoden mit Typparameter, weshalb der Ähnlichkeitswert zwischen zwei Methoden einen gewissen Wert (abhängig von der gewählten Metrik) niemals unterschreiten würde, wenn `methodSignatureTypeParametersWeighting` höher als 0,0 gewichtet wird.

Zur Beantwortung der Forschungsfragen werden verschiedene `SELECT`-Anweisungen mit MySQL formuliert, um daraus aussagekräftige Statistiken erstellen zu können. Die Ergebnisse der `SELECT`-Anweisungen werden in Histogrammen zusammengefasst (siehe Abschnitt 5.4). Detaillierte Ergebnisse, wie der Ähnlichkeitswert zwischen zwei Methoden, werden in Abschnitt 5.4 nicht vorgestellt. Detaillierte Ergebnisse werden bei Bedarf in der Diskussion (Abschnitt 5.5) näher untersucht.

Für die erste Forschungsfrage werden mehrere Methodenvergleiche anhand ihres Ähnlichkeitswertes der 50:50-Metrik ausgewählt, welche für den qualitativen Vergleich zwischen CPD und MBCC Framework genutzt werden. Die Ergebnisse des qualitativen Vergleichs werden anschließend mit den Messgrößen Precision, Recall und F-Measure zusammengefasst.

Bei der Auswahl der Methodenvergleiche wurde darauf geachtet, dass die Methodenvergleiche möglichst unterschiedliche Ähnlichkeitswerte haben. Die Auswahl der Methodenvergleiche erfolgt anhand des Vergleichswertes zwischen zwei Methoden für die 50:50-Metrik (Verhältnis von Methodensignatur zu Methodenrumpf ist 50:50), da die 50:50-Metrik immer den Mittelwert zwischen allen drei Metriken bildet. Die Ergebnisse der anderen beiden Metriken werden ebenfalls erwähnt und kurz erläutert. Eine umfassende Diskussion zur Beeinflussung des Ergebnisses durch verschiedene Metriken erfolgt jedoch erst beim Beantworten der zweiten Forschungsfrage. Für die Auswahl der

Methodenvergleiche werden zehn Bereiche festgelegt, in denen sich die Vergleichswerte befinden sollen. Die Spannweite der Bereiche beträgt jeweils 10%, sodass die Spannweite $S$ des ersten Bereichs bei $1,0 \geq S > 0,9$ liegt, für den zweiten Bereich bei $0,9 \geq S > 0,8$ und so weiter. Aus jedem Bereich werde ich zehn Methodenvergleiche zufällig auswählen, sodass für den Vergleich insgesamt 100 Methodenvergleiche betrachtet werden.

Bei der textbasierten Codeklonerkennung CPD in FeatureIDE werden Zeichenketten in Tokens zerlegt. Damit zwei Programmcodeausschnitte als ähnlich gelten, müssen sie eine gewisse Anzahl gleicher Tokens beinhalten, wie es in Abschnitt 2.3.3.2 vorgestellt wurde. Die Anzahl gleicher Tokens für die Bestimmung von ähnlichen Programmcode kann manuell vom Nutzer bestimmt werden. Zur Durchführung des Vergleichs wird die Anzahl der gleichen Tokens auf 20 gesetzt. Somit wird sichergestellt, dass Codeklone eine gewisse Länge haben und nicht zu kurz sind. Codeklone, deren Länge unter 20 Tokens liegt, werden jedoch nicht als Codeklon erkannt.

Zur Beantwortung der zweiten Forschungsfrage wird die Variabilitätsanalyse mit drei verschiedenen Metriken durchgeführt und verglichen. Der Vergleich erfolgt anhand der schon vorgestellten 70:30-, 50:50- und 30:70-Metriken. Die bei der ersten Forschungsfrage ausgewählten Methodenvergleiche für den qualitativen Vergleich zwischen CPD und MBCC Framework werden zur Beantwortung der dritten Frage wiederverwendet. Für ausgewählte Methodenvergleiche wird eine Diskussion zu den Ergebnissen der verschiedenen Metriken erfolgen, um daraus typische Eigenschaften für die verschiedenen Metriken abzuleiten.

Abschließend werden die qualitativen und quantitativen Vergleiche der ersten und zweiten Forschungsfrage zur Beantwortung der dritten Forschungsfrage dahingehend bewertet. Dazu zählt die Ermittlung, ab welchem Ähnlichkeitswert ein Codeklon mit großer Sicherheit vorliegt und ab welchem Ähnlichkeitswert eine Zusammenführung von Objekten sinnvoll erscheint. Die Ergebnisse der Messgrößen Precision, Recall und F-Measure werden dazu näher betrachtet.

## 5.4 Verteilung der Ähnlichkeiten

In diesem Abschnitt werde ich die Ergebnisse der Variabilitätsanalyse mit den Eingabedaten der ApoGames-Reihe vorstellen. Die Verteilung der Ähnlichkeiten werde ich für jede Granularitätsebene, beginnend von der niedrigsten Ebene, einzeln vorgestellen. Aufgrund des Umfangs der Fallstudie werden die Ergebnisse der Variabilitätsanalyse zusammenfassend in Histogrammen präsentiert. Die Histogramme mit den Ergebnissen der Variabilitätsanalyse sind von der Form her alle gleich. Auf der X-Achse der Histogramme sind die Klassen eingetragen. Jede Klasse steht für einen Bereich, in dem der Vergleichswert der Variabilitätsanalyse zwischen zwei Objekten liegt. Angegeben wird dieser Bereich mit *"obere Grenze"* $\geq a >$ *"untere Grenze"*, wobei die Variable $a$ für den Vergleichswert der Variabilitätsanalyse zwischen zwei Objekten steht. Für jede Klasse gibt es drei verschiedenfarbige Balken. Sie repräsentieren das Ergebnis für die verschiedenen Metriken, also der Gewichtung und somit dem Verhältnis von Methodensignatur zu Methodenrumpf. Die Y-Achse steht für die Anzahl der Elemente

pro Klasse, also die Anzahl der Vergleiche mit einem Vergleichswert, der in dem jeweiligen Bereich liegt. Wichtig zu erwähnen ist dabei, dass für Klassen ohne Methode kein Ähnlichkeitswert berechnet werden kann. Klassen ohne Methode sind im Ergebnis somit nicht enthalten.

**Ergebnis auf Methodenebene**

Auf Methodenebene wird ermittelt, wie ähnlich sich zwei Methoden aus zwei unterschiedlichen Projekten sind. Die Ergebnisse auf Methodenebene liegen bereits aus der Variabilitätsanalyse des MBCC Frameworks vor. In Abbildung 5.1 wird das Ergebnis der 922.352 Methodenvergleiche in einem Histogramm gezeigt. Im Bereich 0 bis 0,10 liegen sehr viele Vergleiche, weshalb Unterschiede zwischen den vorderen Klassen schwer zu erkennen sind. Daher wird in Abbildung 5.2 das Ergebnis für die ersten fünf Klassen (Bereiche mit einem Wert über 0,5) extra angezeigt.

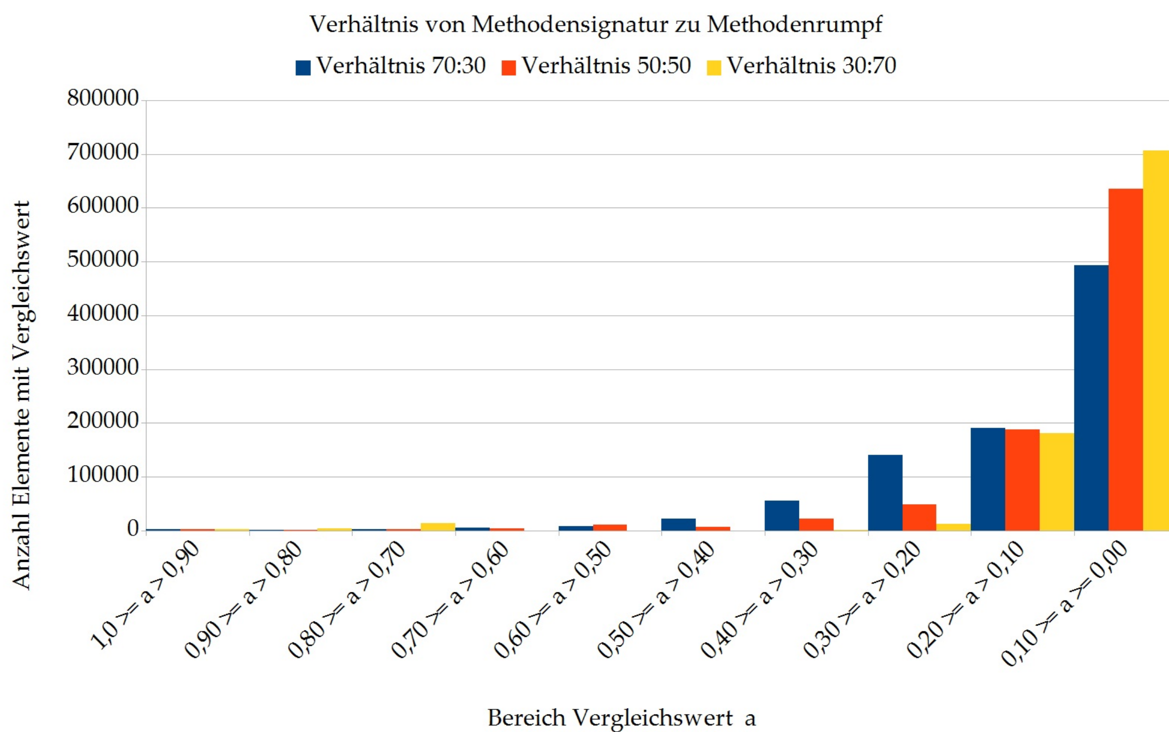



Abbildung 5.1: Histogramm für alle Vergleiche auf Methodenebene

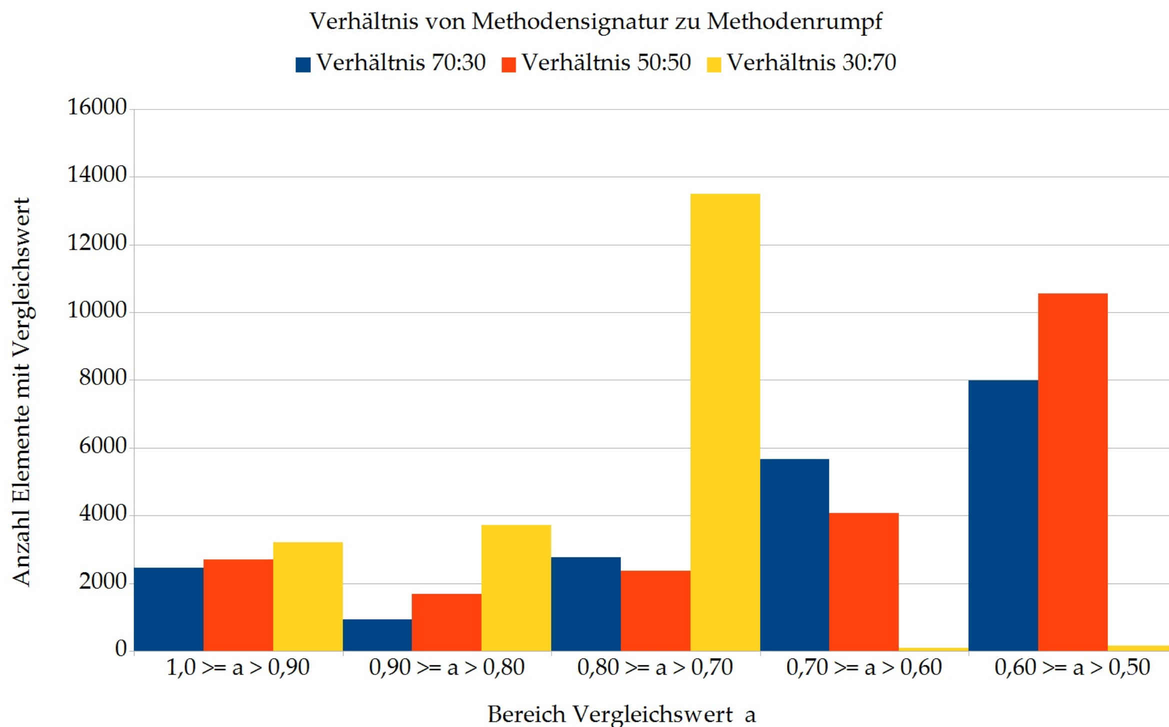



Abbildung 5.2: Histogramm für alle Vergleiche über den Vergleichswert 0,5 auf Methodenebene

### Ergebnis auf Klassenebene

Die Ergebnisse auf Klassenebene geben an, wie ähnlich sich zwei Klassen aus zwei unterschiedlichen Projekten sind. Die Ermittlung der Ähnlichkeit auf Klassenebene erfolgt aus den Ergebnissen der Ähnlichkeit auf Methodenebene. In Abbildung 5.3 wird das Ergebnis der 5134 Klassenvergleiche in einem Histogramm gezeigt. Im Bereich 0 bis 0,10 liegen sehr viele Vergleiche, weshalb Unterschiede zwischen den vorderen Klassen schwer zu erkennen sind. Daher wird in Abbildung 5.4 das Ergebnis für die ersten fünf Klassen (Bereiche mit einem Wert über 0,5) extra angezeigt.

### Ergebnis auf Packageebene

Durch die Ergebnisse auf Packageebene wird ermittelt, wie ähnlich sich zwei Packages aus zwei unterschiedlichen Projekten sind. Die Ermittlung der Ähnlichkeit auf Packageebene erfolgt aus den Ergebnissen der Ähnlichkeit auf Klassenebene. In Abbildung 5.5 wird das Ergebnis der 249 Packagevergleiche in einem Histogramm gezeigt. Im Bereich 0 bis 0,10 liegen sehr viele Vergleiche, weshalb Unterschiede zwischen den vorderen Klassen schwer zu erkennen sind. Daher wird in Abbildung 5.6 das Ergebnis für die ersten fünf Klassen (Bereiche mit einem Wert über 0,5) extra angezeigt.

### Ergebnis auf Projektebene

Die Ähnlichkeit auf Projektebene gibt an, wie ähnlich sich zwei Projekte sind. Die Ermittlung der Ähnlichkeit auf Packageebene erfolgt aus den Ergebnissen der Ähnlichkeit auf Packageebene. In Abbildung 5.7 wird das Ergebnis der 10 Projektvergleiche zwischen den Projekten ApoClock, ApoDice, ApoMonoAndroid, ApoSnake und myTreasureAndroid in einem Histogramm gezeigt. Tabelle 5.1 zeigt eine Konfusionsmatrix, zu welchen Vergleichen die einzelnen Werte gehören. Die Konfusionsmatrix ist symmetrisch, weil der Vergleich zweier Projekte symmetrisch ist. In jeder Zelle stehen dabei drei Vergleichswerte. Sie spiegeln den Vergleichswert für jede durchgeführte Metrik (Verhältnis Methodensignatur zu Methodenrumpf) wieder. In jeder Zelle steht links der Wert für die 70:30-Metrik, in der Mitte steht der Wert für die 50:50-Metrik und rechts steht der Wert für die 30:70-Metrik. Der höchste Wert jeder Zelle ist grün hervorgehoben. Aus der Konfusionsmatrix kann abgelesen werden, welche Metrik für jeden Vergleich den höchsten Ähnlichkeitswert hat. Die Zellen auf der Diagonalen sind leer, da bei der durchgeführten Inter-System Klonsuche ein Vergleich zwischen einem Projekt mit sich selbst nicht vorgenommen wird.

| | ApoClock | ApoDice | ApoMonoA. | ApoSnake | myTreasureA. |
|---|---|---|---|---|---|
| ApoClock | - | 0,47 /0,46/0,45 | 0,43 /0,41/0,39 | 0,45 /0,44/0,43 | 0,30 /0,28/0,26 |
| ApoDice | 0,47 /0,46/0,45 | - | 0,60 /0,58/0,56 | 0,89 /0,88/0,87 | 0,28 /0,26/0,25 |
| ApoMonoA. | 0,43 /0,41/0,39 | 0,60 /0,58/0,56 | - | 0,59 /0,57/0,56 | 0,40 /0,38/0,36 |
| ApoSnake | 0,45 /0,44/0,43 | 0,89 /0,88/0,87 | 0,59 /0,57/0,56 | - | 0,29 /0,28/0,26 |
| myTreasureA. | 0,30 /0,28/0,26 | 0,28 /0,26/0,25 | 0,40 /0,38/0,36 | 0,29 /0,28/0,26 | - |

Tabelle 5.2: Konfusionsmatrix zur Ähnlichkeit aller Projekte für alle Metriken

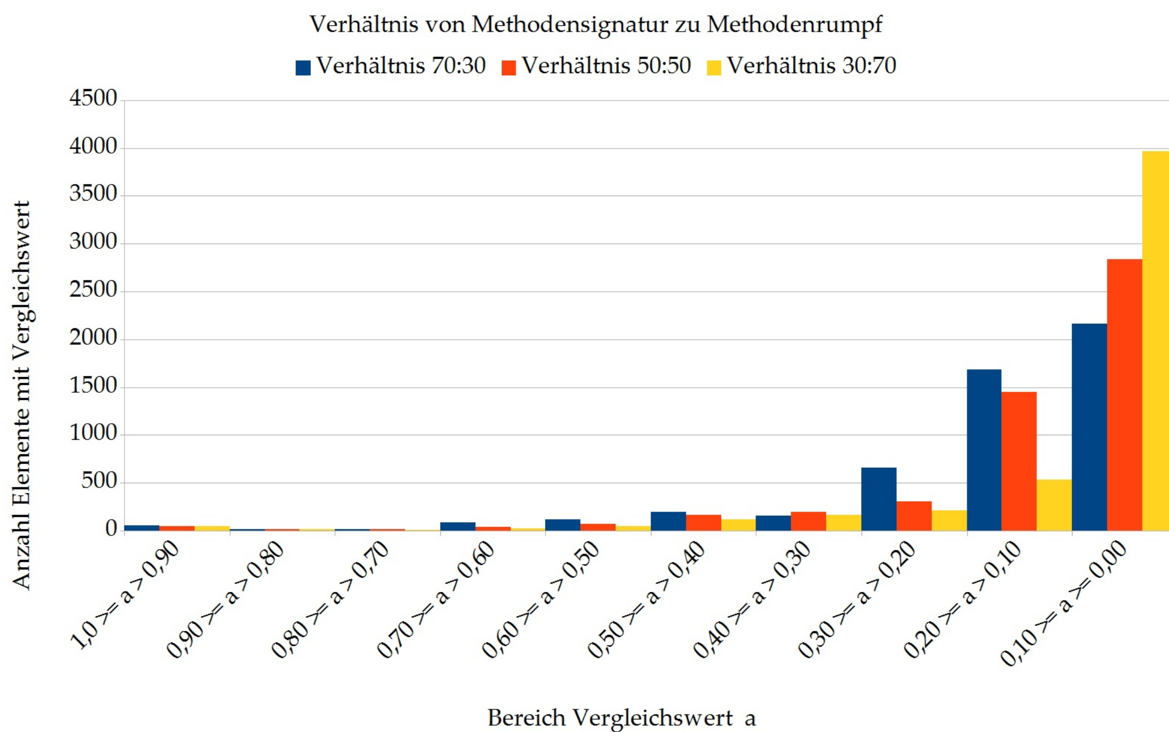



Abbildung 5.3: Histogramm für alle Vergleiche auf Klassenebene

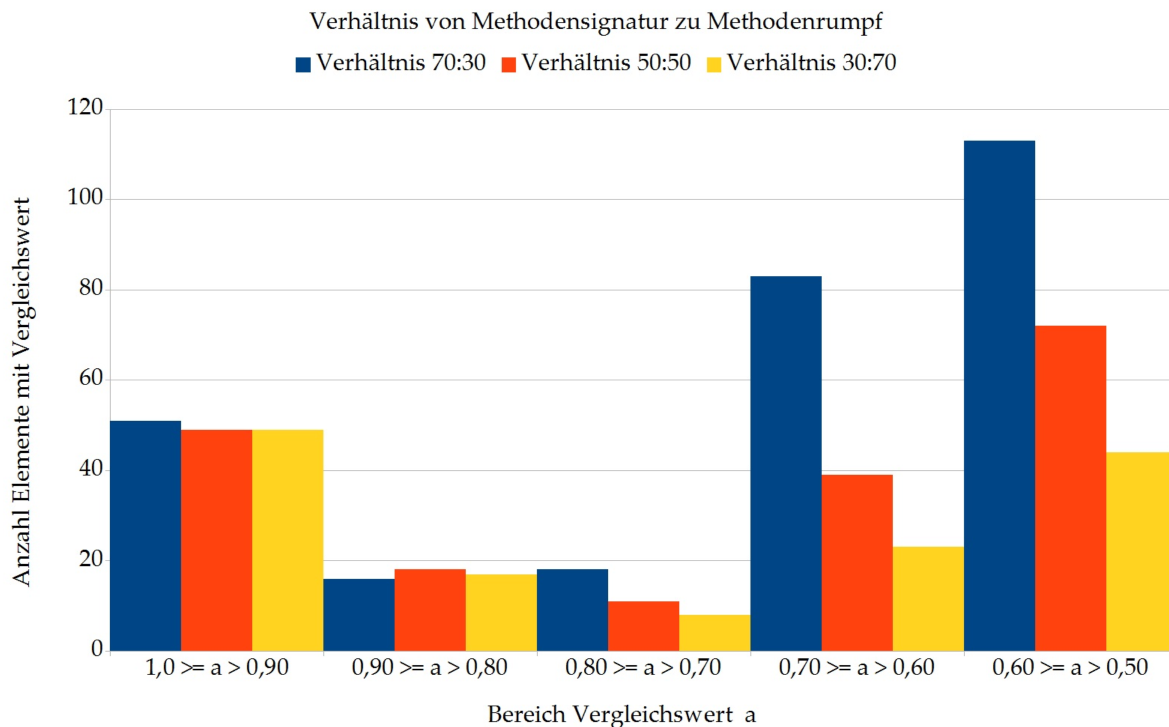



Abbildung 5.4: Histogramm für alle Vergleiche über den Vergleichswert 0,5 auf Klassenebene

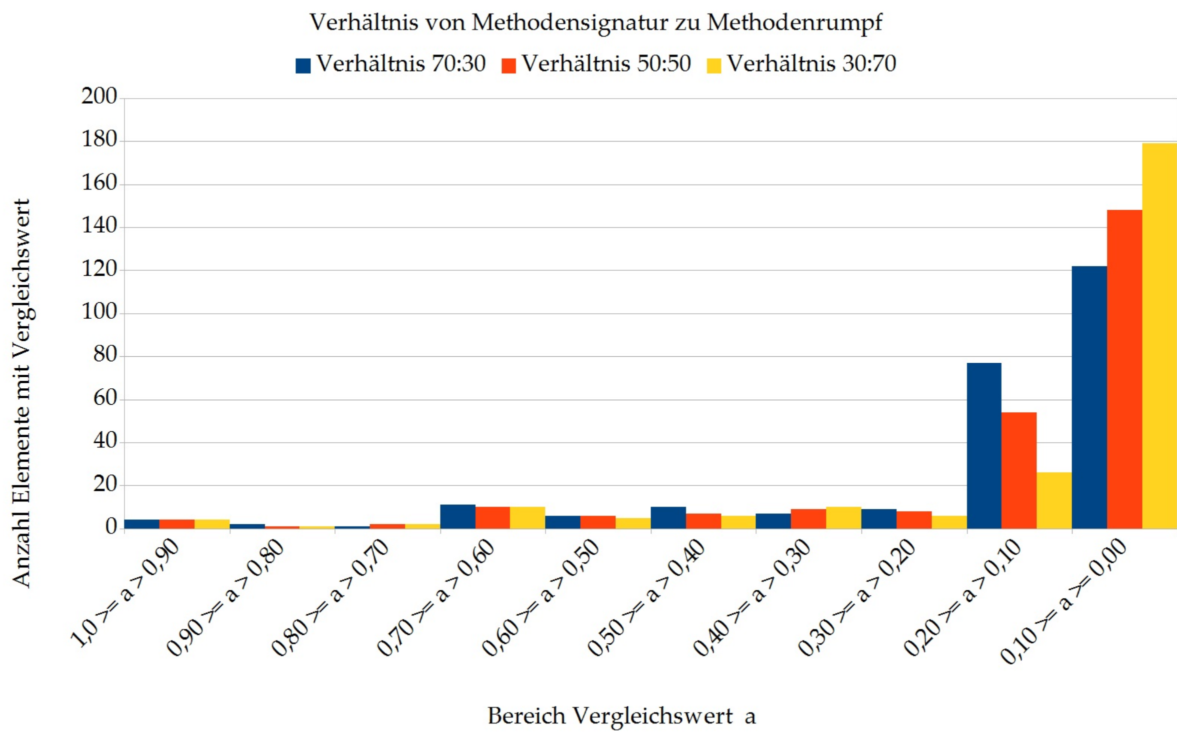



Abbildung 5.5: Histogramm für alle Vergleiche auf Packageebene

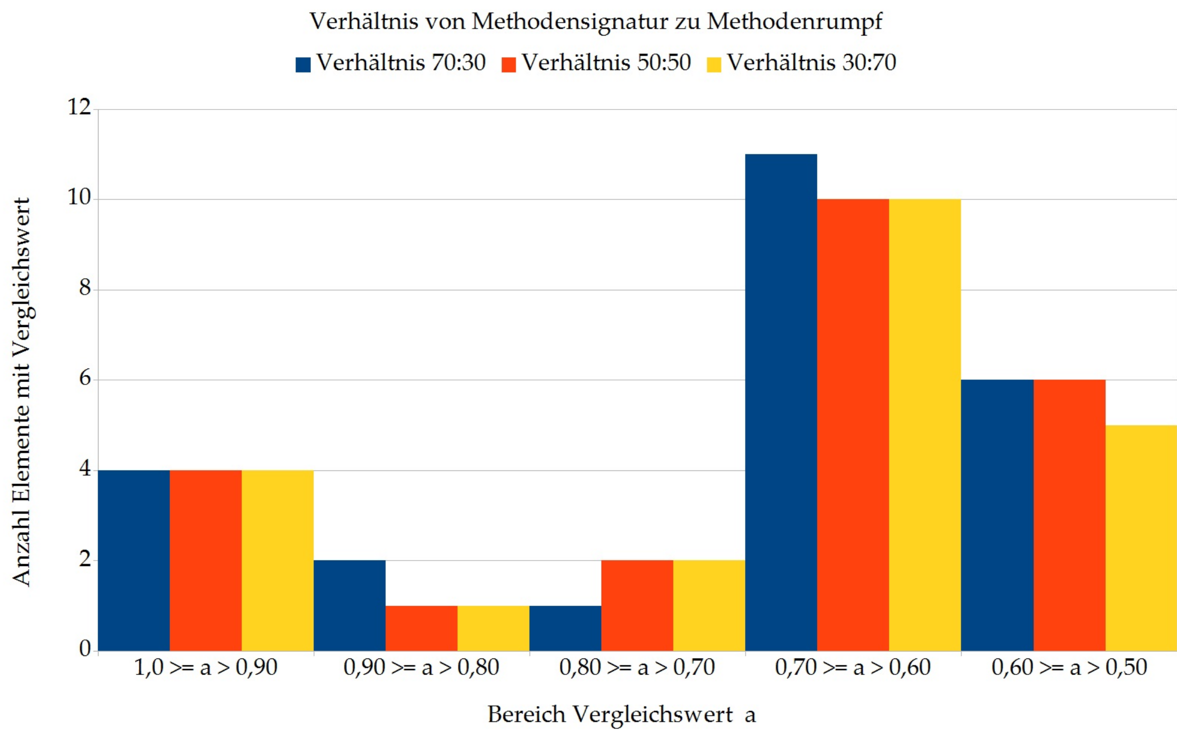



Abbildung 5.6: Histogramm für alle Vergleiche über den Vergleichswert 0,5 auf Packageebene

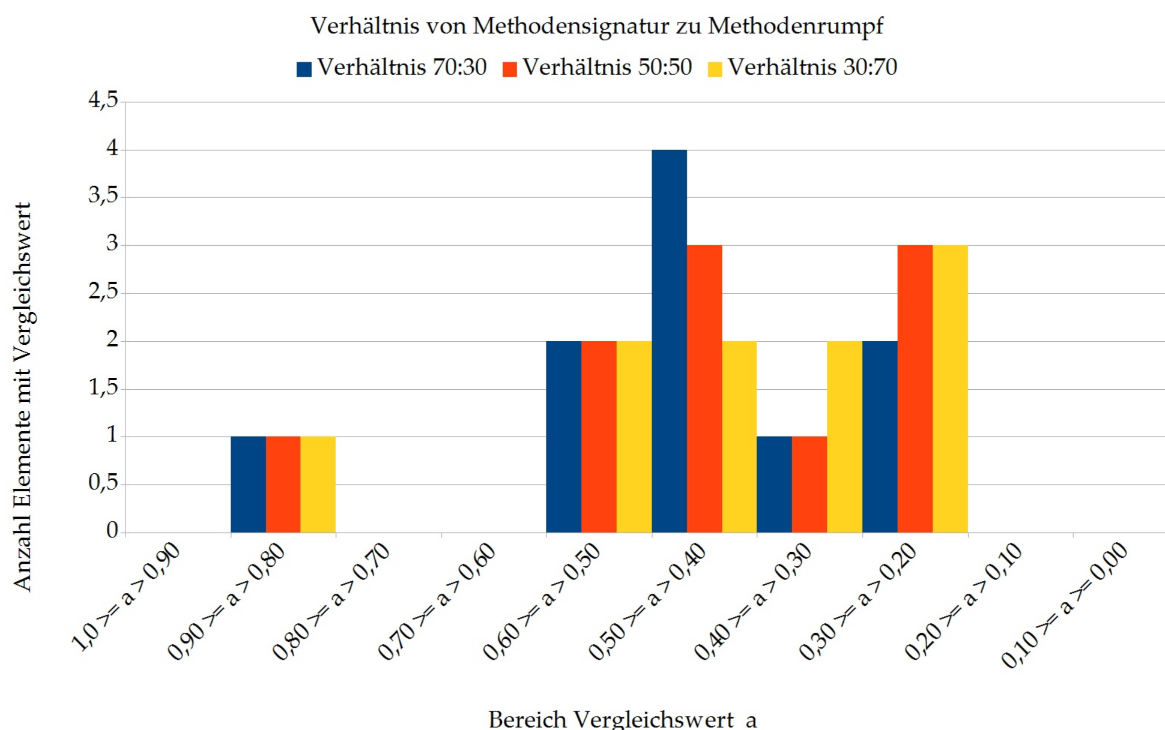



Abbildung 5.7: Histogramm für alle Vergleiche auf Projektebene

## 5.5 Diskussion

In diesem Abschnitt werde ich die in Abschnitt 5.4 vorgestellten Ergebnisse diskutieren und interpretieren. Als Diskussionsleitfaden dienen dafür die in Abschnitt 5.1 formulierten Forschungsfragen, welche ich im Zuge der Diskussion beantworten werde. Für jede Forschungsfrage werden die relevanten Ergebnisse zur Beantwortung der einzelnen Fragen herangezogen. Zusätzlich werden noch aussagekräftige detaillierte Auswertungen, wie der Vergleich von zwei Methoden und denn dazugehörigen Vergleichswerten, zur Durchführung der Diskussion genutzt.

### 5.5.1 Qualitativer Vergleich zwischen modellbasierter und textbasierter Codeklonerkennung

Ein Kriterium zur Beantwortung der ersten Forschungsfrage *“Wie gut ist die Codeklonerkennung der modellbasierten Variabilitätsanalyse des MBCC Frameworks im Vergleich zu einer textbasierten Codeklonerkennung?”* ist der qualitative Vergleich zwischen der textbasierten Codeklonerkennung in FeatureIDE und der modellbasierten Variabilitätsanalyse des MBCC Frameworks.

Bei der detaillierten Betrachtung der 100 ausgewählten Methodenvergleiche wird darauf geachtet, ob ein Codeklon vorliegt oder nicht und ob die Zusammenführung dieses Codeklons sinnvoll ist. Für jeden Methodenvergleich wird der Ähnlichkeitswert vom MBCC Framework ermittelt und kontrolliert, ob CPD für den jeweiligen Methoden-

Listing 5.2: Qualitativer Vergleich zwischen onBackButtonPressed (ApoClockMenu) und onBackButtonPressed (ApoMonoMenu)

```
/*
 * Methode onBackButtonPressed aus der Klasse ApoClockMenu (Projekt ApoClock)
 */
  public void onBackButtonPressed() {
    BitsGame.getInstance().finishApp();
  }
/*
 * Methode onBackButtonPressed aus der Klasse ApoMonoMenu (Projekt ApoMonoAndroid)
 */
  public void onBackButtonPressed() {
    BitsGame.getInstance().finishApp();
  }
```

vergleich einen Codeklon erkennt oder nicht. Die Entscheidung ob die Zusammenführung von zwei verglichenen Methoden sinnvoll ist oder nicht, hängt davon ab, ob ein Codeklon zwischen beiden Methoden vorliegt oder nicht. Ob ein Codeklon zwischen beiden Methoden vorliegt oder nicht, wird von mir anhand der Kriterien für die verschiedenen Typen von Codeklonen (siehe Abschnitt 2.3.2) festgestellt. Ein Codeklon muss somit die Kriterien von mindestens einem der vier Typen erfüllen, damit dieser als Codeklon gewertet wird. Bei den 100 ausgewählten Methodenvergleiche wurde jedoch kein Codeklon des vierten Typs gefunden, weshalb sich die Betrachtung auf die Typen I bis III konzentriert.

Eine Aufführung aller 100 vorgenommenen Methodenvergleiche und deren Diskussion wird an dieser Stelle nicht erfolgen. Von diesen 100 Methodenvergleiche werden jedoch nachfolgend einige wenige genauer vorgestellt und am Ende werden die Erkenntnisse aus allen Methodenvergleichen zusammengefasst. Die miteinander verglichenen Methoden werden immer in einem Listing aufgeführt. Unterschiedlicher Programmcode zwischen den Methoden ist in den Listings rot markiert.

In Listing 5.2 wird ein Type-I-Klon gezeigt. Zwischen den Methoden `onBackButtonPressed` aus der Klasse ApoClockMenu und `onBackButtonPressed` aus der Klasse ApoClockMenu liegt ein Type-I-Klon vor, da beide Methoden identisch sind. Das Ergebnis der Variabilitätsanalyse des MBCC Frameworks liegt für alle drei Metriken bei 1,0. Des Weiteren erkennt CPD für dieses Beispiel einen Codeklon. Dies liegt jedoch daran, dass sowohl nach als auch vor den Methoden noch weiterer identischer Programmcode steht, welcher in dem Listing jedoch nicht abgebildet ist. Der von CPD erkannte Codeklon geht somit über die Grenzen der Methoden hinaus. Würden beide Methoden für sich stehen, so würde CPD für diese beiden Methoden keinen Codeklon erkennen, da die Länge beider Methoden 17 Tokens beträgt und somit unter der gewählten Mindestanzahl von 20 Tokens liegt. Damit CPD auch kleinere Type-I-Klone erkennt, müsste die Mindestanzahl heruntergesetzt werden.

Type-I-Klone werden von der Variabilitätsanalyse des MBCC Frameworks immer erkannt und unabhängig von der gewählten Metrik mit einem Ähnlichkeitswert von 1,0 bewertet. Die Variabilitätsanalyse des MBCC Frameworks ist für die Erkennung von Type-I-Klonen somit stabiler, da die Erkennungsrate von CPD von der gewählten Mindestanzahl an Tokens abhängt. Type-I-Klone eignen sich für die Zusammenführung

Listing 5.3: Qualitativer Vergleich zwischen getAllLevelsSorted (ApoDiceUserlevels) und getAllLevelsSorted (MyTreasureUserLevels)

```
1  /*
2   * Methode getAllLevelsSorted aus der Klasse ApoDiceUserlevels (Projekt ApoDice)
3   */
4    private String[] getAllLevelsSorted() {
5      if (this.userlevels.getLevels().size() <= 0) {
6        return null;
7      }
8      int size = this.sortByUpload.size();
9      String[] levels = new String[size];
10     for (int level = 0; level < levels.length; level++) {
11       String curLevel = this.userlevels.getLevels().get(this.sortByUpload.get(level));
12       BitsLog.d(level‘‘, curLevel);
13       levels[level] = curLevel;
14     }
15     return levels;
16   }
17 /*
18  * Methode getAllLevelsSorted aus der Klasse MyTreasureUserLevels (Projekt
       myTreasureAndroid)
19  */
20   private String[] getAllLevelsSorted() {
21     if (this.userlevels.getLevels().size() <= 0) {
22       return null;
23     }
24     int size = this.sortByUpload.size();
25     String[] levels = new String[size];
26     for (int level = 0; level < levels.length; level++) {
27       String curLevel = this.userlevels.getLevels().get(this.sortByUpload.get(level));
28       levels[level] = curLevel;
29     }
30     return levels;
31   }
```

besonders gut, da keine Anpassungen am Programmcode vorgenommen werden müssen.

In Listing 5.3 werden die Methoden `getAllLevelsSorted` (aus der Klasse ApoDiceUserlevels und dem Projekt ApoDice) mit der Methode `getAllLevelsSorted` (aus der Klasse MyTreasureUserLevels Projekt myTreasureAndroid) aufgeführt. Bei diesen beiden Methoden handelt es sich um einen Type-III-Klon, da eine Methode eine zusätzliche Anweisung in Zeile 12 hat. Die Variabilitätsanalyse des MBCC Frameworks ergab dabei folgende Ähnlichkeitswerte: 0,97 für die 70:30-Metrik, 0,95 für die 50:50-Metrik und 0,92 für die 30:70-Metrik für das Verhältnis Methodensignatur zu Methodenrumpf. Die Ergebnisse der Variabilitätsanalyse sind für diesen Vergleich plausibel, da die Methodensignatur in beiden Methoden identisch ist und die Abweichung beim Methodenrumpf bei nur einem Statement liegt. Den höchsten Vergleichswert hat somit auch die Metrik, bei der die Methodensignatur mit 70% am höchsten gewichtet ist.

Die textbasierte Codeklonerkennung CPD in FeatureIDE hat für diesen Vergleich zwei Codeklone erkannt. Beide Codeklone gehen dabei über die Methodengrenze hinaus, da sowohl vor als auch nach den Methoden weiterer identischer Programmcode steht, welcher in dem Listing jedoch nicht abgebildet ist. Der erste Codeklon endet vor der rot markierten Zeile und der zweite Codeklon beginnt nach der rot markierten Zeile. Type-III-Klone lassen sich zusammenführen, in dem vorher Anpassungen vorgenommen werden. Die Anpassung von Type-III-Klonen kann mit dem Refactoring “Extract

Listing 5.4: Qualitativer Vergleich zwischen getIBackground (ApoEntity) und getIBackground (ApoEntity)

```
/*
 * Methode getIBackground aus der Klasse ApoEntity (Projekt ApoDice)
 */
  public BitsImage getIBackground() {
    return this.iBackground;
  }
/*
 * Methode getIBackground aus der Klasse ApoEntity (Projekt ApoMonoAndroid)
 */
  public BitsGLImage getIBackground() {
    return this.iBackground;
  }
```

Method” durchgeführt werden. Mit “Extract Method” werden die unterschiedlichen Funktionalitäten aus den Methoden herausgezogen und separat umgesetzt. Die identischen Funktionalitäten beider Methoden werden anschließend zusammengefasst.

Für das Beispiel in Listing 5.3 ist das Ergebnis des MBCC Frameworks besser nachvollziehbar, da hier nur ein Codeklon mit einem sehr hohen Ähnlichkeitswert erkannt wird. Das Ergebnis von CPD ist hier nicht optimal, da für eine Methode gleich zwei Codeklone erkannt werden, welche auch noch über die Methodengrenzen hinausgehen.

In Listing 5.4 werden die Methoden `getIBackground` (aus der Klasse ApoEntity und dem Projekt ApoDice) mit der Methode `getIBackground` (aus der Klasse ApoEntity Projekt ApoMonoAndroid) aufgeführt. Bei diesen beiden Methoden handelt es sich um einen Type-II-Klon, da die Bezeichnung des Rückgabewertes anders ist. Die Variabilitätsanalyse des MBCC Frameworks ergab dabei folgende Ähnlichkeitswerte: 0,93 für die 70:30-Metrik, 0,95 für die 50:50-Metrik und 0,97 für die 30:70-Metrik für das Verhältnis Methodensignatur zu Methodenrumpf. Die Ergebnisse der Variabilitätsanalyse sind für diesen Vergleich plausibel, da der Methodenrumpf in beiden Methoden identisch ist und es eine kleine Abweichung bei der Methodensignatur gibt. Den höchsten Vergleichswert hat somit auch die Metrik, bei der der Methodenrumpf mit 70% am höchsten gewichtet ist.

Type-II-Klone lassen sich mit kleinen Anpassungen zusammenfassen. Vor der Zusammenfassung muss lediglich die Bezeichnung aller Elemente vereinheitlicht werden.

Die textbasierte Codeklonerkennung CPD in FeatureIDE hat für den Vergleich in Listing 5.4 keinen Codeklon erkannt, weil die Methoden nur 12 Tokens lang sind. Dadurch erreichen die identischen Programmcodefragment nicht die erforderliche Anzahl an Mindesttokens von 20, um von CPD als Codeklon erkannt zu werden. Damit CPD hier einen Codeklon erkennt, müsste die Anzahl an Tokens auf 10 oder geringer heruntergesetzt werden. Ein Heruntersetzen der Mindestanzahl an Tokens hätte jedoch zur Folge, dass mehr Codeklone, welche über die Methodengrenzen hinausgehen, erkannt werden, wie es bei den Methoden in Listing 5.3 der Fall ist.

Listing 5.5: Qualitativer Vergleich zwischen setEditor (ApoDicePanel) und setEditor (ApoMonoPanel)

```
/*
 * Methode setEditor aus der Klasse ApoDicePanel (Projekt ApoDice)
 */
  protected final void setEditor(boolean bSolvedLevel) {
    if (super.getModel() != null) {
      super.getModel().close();
    }

    super.setModel(this.editor);

    this.setButtonVisible(ApoDiceConstants.BUTTON\underline{\ }EDITOR);

    this.editor.setLevelSolved(bSolvedLevel);
    super.getModel().init();
  }
/*
 * Methode setEditor aus der Klasse ApoMonoPanel (Projekt ApoMonoAndroid)
 */
  protected final void setEditor(boolean bUpload) {
    if (super.getModel() != null) {
      super.getModel().close();
    }

    super.setModel(this.editor);

    this.setButtonVisible(ApoMonoConstants.BUTTON\underline{\ }EDITOR);
    this.editor.setUploadVisible(bUpload);

    super.getModel().init();

    this.changeAdViewVisibility(AdView.INVISIBLE);
    this.musicPlayer.setMenu(true);
  }
```

In Listing 5.5 werden die Methoden `setEditor` aus der Klasse ApoDicePanel und `setEditor` aus der Klasse ApoMonoPanel aufgeführt. Bei diesen beiden Methoden handelt es sich um einen Codeklon des zweiten und dritten Types. Ein Codeklon des zweiten Typs liegt vor, da in Zeile 11 und 26 die Bezeichung einer Variablen anders ist (`ApoDiceConstants` wird zu `ApoMonoConstants`). Ein Type-III-Klon liegt vor, da es zusätzliche Anweisungen gibt, wie in Zeile 31 und 32. Die Variabilitätsanalyse des MBCC Frameworks ergab dabei folgende Ähnlichkeitswerte: 0,85 für die 70:30-Metrik, 0,75 für die 50:50-Metrik und 0,65 für die 30:70-Metrik für das Verhältnis Methodensignatur zu Methodenrumpf.

Codeklone, die sowohl Type-II-Klone als auch Type-III-Klone sind, können mit größeren Anpassungen zusammengefasst werden. Es muss sowohl eine Anpassung mit "Extract Method" als auch eine vereinheitlichte Bezeichnung der Elemente vorgenommen werden, bevor die Methoden zusammengefasst werden können.

CPD erkennt für diese beide Methoden aus Listing 5.5 einen Codeklon in den Zeilen 5 bis 9 und 20 bis 24, da nur diese beiden identischen Abschnitte die Mindestanzahl von 20 Tokens erreichen. Wie in diesem Beispiel gibt es Methoden, bei denen mehrere kleine Änderungen über die gesamte Methode verteilt vorgenommen wurden. Bei diesen Methoden handelt es sich dann um Type-II-Klone oder Type-III-Klone oder um beides.

Listing 5.6: Qualitativer Vergleich zwischen setSolved (ApoLevelChooserButton) und setBUse (ApoEntity)

```
/*
 * Methode setSolved aus der Klasse ApoLevelChooserButton (Projekt ApoSnake)
 */
  public void setSolved(boolean bSolved) {
    this.bSolved = bSolved;
  }
/*
 * Methode setBUse aus der Klasse ApoEntity (Projekt myTreasureAndroid)
 */
  public void setBUse(boolean bUse) {
    this.bUse = bUse;
  }
```

Die Ergebnisse der Variabilitätsanalyse des MBCC Frameworks sind im Vergleich zu den Ergebnissen von CPD für solche Fälle plausibel. Die Variabilitätsanalyse liefert für jede Metrik einen hohen Ähnlichkeitswert. Der Unterschied zwischen der 70:30-Metrk und der 30:70 ist mit 0,2 jedoch nicht gering. Der Grund für diesen Unterschied liegt darin, dass die Unterschiede zwischen beiden Methoden vorrangig im Methodenrumpf sind, weshalb die Metrik mit der höchsten Gewichtung der Methodensignatur den höchsten Ähnlichkeitswert hat. CPD erkennt hier nur einen Codeklon, wenn die identischen Abschnitte groß genug sind. Falls die identischen Abschnitte jedoch zu kurz sind, wird kein Codeklon erkannt.

In Listing 5.6 ist ein weiteres Beispiel eines Type-II-Klons zwischen der Methode `setSolved` aus der Klasse ApoLevelChooserButton und der Methode `setBUse` aus der Klasse ApoEntity abgebildet. Obwohl es nur Änderungen in den Bezeichnern gibt und die Methoden ansonsten identisch sind, liegt das Ergebnis der Variabilitätsanalyse für die 70:30-Metrik bei 0,47, für die 50:50-Metrik bei 0,33 und für die 30:70-Metrik bei 0,2. Besonders der Wert für die 30:70-Metrik ist hier sehr gering im Vergleich zum Methodenvergleich in Listing 5.4, wo es sich auch um einen Type-II-Klon handelt. Der Wert für die 30:70-Metrik ist beim Beispiel in Listing 5.6 besonders niedrig, weil die Übereinstimmung des Methodenrumpfes mit 0,0 bewertet wird(keine Übereinstimmung).

Obwohl der Ähnlichkeitswert bei allen Metriken unter 0,5 liegt, können auch diese beiden Methoden mit kleinen Anpassungen zusammengefasst werden. So muss nur der Methodenname und der Parametername vereinheitlicht werden.

CPD findet für Listing 5.6 keinen Codeklon, da die identischen Abschnitte zu klein sind. Sowohl die Variabilitätsanalyse des MBCC Frameworks als auch CPD haben Probleme kleine Type-II- oder Type-III-Klone als Codeklone zu erkennen.

Ein letztes Beispiel für den qualitativen Vergleich wird der Vergleich zwischen der Methode `canBeTested` aus der Klasse ApoMonoEditor und der Methode `getLevelString` aus der Klasse ApoSnakeEditor in Listing 5.7 gezeigt. Bei diesen beiden Methoden handelt es sich nicht um einen Codeklon, weshalb eine Zusammenfassung dieser Methoden, aufgrund ihrer Unterschiede, nicht sinnvoll ist. Zur besseren Lesbarkeit

Listing 5.7: Qualitativer Vergleich zwischen canBeTested (ApoMonoEditor) und getLevelString (ApoEntity)

```
/*
 * Methode canBeTested aus der Klasse ApoMonoEditor (Projekt ApoMonoAndroid)
 */
  private boolean canBeTested() {
    int goal = 0;
    int player = 0;
    int beamer = 0;
    for (int y = 0; y < this.level.length; y++) {
      for (int x = 0; x < this.level[0].length; x++) {
        if (this.level[y][x] == 2) {
          player += 1;
        }
        if (this.level[y][x] == 3) {
          goal += 1;
        }
        if (this.level[y][x] == 11) {
          beamer += 1;
        }
      }
    }
    if ((player == 1) && (goal == 1) && ((beamer == 0) || (beamer == 2))) {
      this.getGame().getButtons()[5].setVisible(true);
      return true;
    }
    this.getGame().getButtons()[5].setVisible(false);
    this.setUploadVisible(false);
    return false;
  }
/*
 * Methode getLevelString aus der Klasse ApoSnakeEditor (Projekt ApoSnake)
 */
  private String getLevelString() {
    String level = "";
    char c = (char)(this.level[0].length + 96);
    level += String.valueOf(c);
    c = (char)(this.level.length + 96);
    level += String.valueOf(c);
    for (int y = 0; y < this.level.length; y++) {
      for (int x = 0; x < this.level[0].length; x++) {
        if (this.level[y][x] >= 10) {
          c = (char)(87 + this.level[y][x]);
          level += String.valueOf(c);
        } else {
          level += String.valueOf(this.level[y][x]);
        }
      }
    }
    return level;
  }
```

werden die Unterschiede zwischen beiden Methoden nicht rot markiert, da ein Großteil nicht übereinstimmt. Im Mittelteil beider Methoden gibt es jedoch eine auffällige Übereinstimmung, welche grün markiert ist.

Die Variabilitätsanalyse des MBCC Frameworks ergab dabei folgende Ähnlichkeitswerte: 0,32 für die 70:30-Metrik, 0,26 für die 50:50-Metrik und 0,20 für die 30:70-Metrik. Da es sich beim Methodenvergleich in Listing 5.6 nicht um einen Codeklon handelt, sind die niedrigen Werte bei der Variabilitätsanalyse auch plausibel.

CPD hat für diese beiden Methoden jedoch einen Codeklon erkannt, nämlich für die grün markierten Abschnitte. Bei dieser Übereinstimmung handelt es sich um den Beginn von zwei verschachtelten `for`-Schleifen. Die Länge der Übereinstimmung ist größer als 20 Tokens, weshalb CPD hier einen Codeklon erkennt. Auch wenn dieser Abschnitt übereinstimmt, sind diese beiden Methoden keine Codeklone, weshalb das Ergebnis von CPD hier falsch ist. Die niedrigen Werte der Variabilitätsanalyse sind hier plausibler. Im vorherigen Beispiel in Listing 5.4 wurde kein Codeklon erkannt, weil die Mindestanzahl an Tokens mit 20 zu hoch war. Würde die Mindestanzahl jedoch heruntergesetzt werden, würden mehr Codeklone wie in Listing 5.7 erkannt, obwohl solche Methoden keine Codeklone sind. Die Ergebnisse der Variabilitätsanalyse des MBCC Frameworks sind für Methodenvergleiche dieser Art, wie in Listing 5.7, stabiler.

Nach Betrachtung aller 100 Methodenvergleiche bin ich zu dem Fazit gekommen, dass die Variabilitätsanalyse sich besser als CPD zum Aufspüren von Codeklonen, welche zusammenführbar sind, eignet.

Die von CPD erkannten Codeklone gehen oft über die Methodengrenzen hinaus oder es wird beim Vergleich von zwei Methoden mehr als ein Codeklon gefunden (siehe Listing 5.3). Die Variabilitätsanalyse hat dieses Problem nicht, da ein Vergleich von zwei Methoden immer nur einen Ähnlichkeitswert hat und dieser unabhängig von benachbarten Methoden ermittelt wird.

Beide Verfahren haben jedoch eine Schwäche bei der Erkennung von Type-II- und Type-III-Klonen, falls es sich um Methoden mit sehr kleinem Methodenrumpf handelt (Listing 5.6). CPD hat hier wieder das Problem, dass die Anzahl an Tokens der identischen Abschnitte nicht die Mindestanzahl an Tokens erreicht. Die Variabilitätsanalyse ergibt für solche Methodenvergleiche einen geringen Ähnlichkeitswert, weil kleine Änderungen, wie die Umbenennung einer Variablen an einer Stelle, bei Methoden mit kleinem Methodenrumpf das Ergebnis mehr beeinflussen als bei Methoden mit großem Methodenrumpf. Die Variabilitätsanalyse ist bei der Erkennung von kleinen Methoden jedoch stabiler als CPD. Bei CPD werden die verglichenen Methoden gar nicht als Codeklon klassifiziert. Bei der Variabilitätsanalyse ist lediglich der Ähnlichkeitswert geringer und ein Entwickler kann bei kleineren Ähnlichkeitswerten immer noch selbst entscheiden, ob es sich um einen Codeklon handelt oder nicht.

### 5.5.2 Quantitativer Vergleich zwischen modellbasierter und textbasierter Codeklonerkennung

Ein weiteres Kriterium zur Beantwortung der ersten Forschungsfrage *“Wie gut ist die Codeklonerkennung der modellbasierten Variabilitätsanalyse des MBCC Frameworks im Vergleich zu einer textbasierten Codeklonerkennung?”* ist der quantitative Vergleich beider Verfahren. Für den quantitativen Vergleich werde ich die Ergebnisse der 100 ausgewählten Methodenvergleiche des qualitativen Vergleichs zusammenfassen. Für den quantitativen Vergleich werde ich die Messgrößen Precision, Recall und F-Measure für beide Verfahren der Codeklonerkennung bestimmen.

In Tabelle 5.3 ist eine Übersicht aller 100 Methodenvergleiche abgebildet. Die Übersicht zeigt, wie ich die 100 Methodenvergleiche kategorisiert habe und wie viele der 100 Methodenvergleich in jede Kategorie gehören. Für den quantitativen Vergleich habe ich zehn Bereiche festgelegt. Diese Bereiche werden mit *“obere Grenze”* $\geq a >$ *“untere Grenze”* bezeichnet, wobei $a$ der Ähnlichkeitswert der Variabilitätsanalyse für diesen Methodenvergleich ist (Ergebnis für die 50:50-Metrik). Für jeden Bereich (erste Spalte) wird aufgelistet, ob und wie viele Codeklone in diesem Bereich von mir gefunden wurden und um welche Art es sich handelt (Spalten 2 bis 7). In der zweiten Spalte ist die Anzahl der zusammenführbaren Type-I-Klone für diesen Bereich festgehalten. In der dritten Spalte ist die Anzahl der zusammenführbaren Codeklone, welche nur Type-II-Klone sind, festgehalten. In der vierten Spalte ist die Anzahl der zusammenführbaren Codeklone, welche nur Type-III-Klone sind, festgehalten. In der fünften Spalte ist die Anzahl der zusammenführbaren Codeklone, welche sowohl Type-II-Klone als auch Type-III-Klone sind, festgehalten. In der sechsten Spalte ist die Anzahl aller zusammenführbaren Codeklone festgehalten (Zusammenfassung der Spalten 2 bis 5). In der siebten Spalte ist die Anzahl der Methodenvergleiche, deren Zusammenführung nicht sinnvoll ist, festgehalten.

Aus Tabelle 5.3 lässt sich ablesen, dass es sich bei allen Methodenvergleichen mit einem Wert über 0,6 bei der Variabilitätsanalyse um zusammenführbare Codeklone handelt. Alle Methodenvergleiche mit einem Wert von 0,2 oder niedriger sind nicht zusammenführbar. In den Bereichen zwischen 0,2 und 0,6 gibt es sowohl zusammenführbare Codeklone als auch Methoden, deren Zusammenführung nicht sinnvoll ist und je niedriger der Vergleichswert ist, desto größer ist die Wahrscheinlichkeit, dass es sich um Methoden handelt, deren Zusammenführung nicht sinnvoll ist.

In Tabelle 5.4 wird gezeigt, wie gut CPD Codeklone erkannt hat. Die Tabelle 5.4 entspricht vom Aufbau her der Tabelle 5.3, mit dem Unterschied, dass in jeder Zelle zwei Werte stehen. Der rechte Wert entspricht der tatsächlichen Anzahl an zusammenführbaren Codeklonen diesen Typs und sie stimmen mit den Werten aus der Tabelle 5.3 überein. Für die Spalten 2 bis 6 gibt der linke Wert für jeden Typ an, wie viele Codeklone von CPD bei zusammenführbaren Methoden auch erkannt wurden (true-positives). In Spalte 7 gibt der linke Wert an, bei wie vielen Methoden, deren Zusammenführung nicht sinnvoll ist, CPD auch keinen Codeklone gefunden hat (true negative). Aus der Tabelle 5.4 lässt sich herauslesen, dass je niedriger der Ähnlichkeitswert aus der Variabilitätsanalyse ist, desto schwerer fällt es CPD Codeklone zwischen den zusammenführbaren Methoden zu finden. Während im Bereich 0,9 bis

| Bereich | Typ des Codeklon | | | | | |
|---|---|---|---|---|---|---|
| | Typ I | Typ II | Typ III | Typ II u. III | Alle Typen | Kein Klon |
| $1{,}0 \geq a > 0,9$ | 2 | 6 | 2 | - | 10 | - |
| $0{,}9 \geq a > 0,8$ | - | 9 | 1 | - | 10 | - |
| $0{,}8 \geq a > 0,7$ | - | 7 | - | 3 | 10 | - |
| $0{,}7 \geq a > 0,6$ | - | 6 | 1 | 3 | 10 | - |
| $0{,}6 \geq a > 0,5$ | - | 3 | - | 3 | 6 | 4 |
| $0{,}5 \geq a > 0,4$ | - | 2 | - | 1 | 3 | 7 |
| $0{,}4 \geq a > 0,3$ | - | 1 | - | 3 | 4 | 6 |
| $0{,}3 \geq a > 0,2$ | - | - | - | 1 | 1 | 9 |
| $0{,}2 \geq a > 0,1$ | - | - | - | - | - | 10 |
| $0{,}1 \geq a \geq 0,0$ | - | - | - | - | - | 10 |
| Alle Bereiche | 2 | 34 | 4 | 14 | 54 | 46 |

Tabelle 5.3: Kategorisierung der 100 betrachteten Methodenvergleiche

| Bereich | Typ des Codeklon | | | | | |
|---|---|---|---|---|---|---|
| | Typ I | Typ II | Typ III | Typ II u. III | Alle Typen | Kein Klon |
| $1{,}0 \geq a > 0,9$ | 2/2 | 3/6 | 2/2 | - | 7/10 | -/- |
| $0{,}9 \geq a > 0,8$ | -/- | 5/9 | 1/1 | -/- | 6/10 | -/- |
| $0{,}8 \geq a > 0,7$ | -/- | 2/7 | -/- | 3/3 | 5/10 | -/- |
| $0{,}7 \geq a > 0,6$ | -/- | 2/6 | 1/1 | 1/3 | 4/10 | -/- |
| $0{,}6 \geq a > 0,5$ | -/- | -/3 | -/- | 2/3 | 2/6 | 3/4 |
| $0{,}5 \geq a > 0,4$ | -/- | -/2 | -/- | 1/1 | 1/3 | 7/7 |
| $0{,}4 \geq a > 0,3$ | -/- | -/1 | -/- | 1/3 | 1/4 | 6/6 |
| $0{,}3 \geq a > 0,2$ | -/- | -/- | -/- | -/1 | -/1 | 8/9 |
| $0{,}2 \geq a > 0,1$ | -/- | -/- | -/- | -/- | -/- | 10/10 |
| $0{,}1 \geq a \geq 0,0$ | -/- | /- | /- | -/- | -/- | 10/10 |
| Alle Bereiche | 2/2 | 12/34 | 4/4 | 8/14 | 26/54 | 44/46 |

Tabelle 5.4: Erkennungsrate von CPD für die 100 kategorisierten Methodenvergleiche

1,0 noch 7 von 10 zusammenführbaren Codeklonen gefunden werden, findet CPD im Bereich 0,6 bis 0,7 nur 4 von 10 zusammenführbaren Codeklonen.

Mit den Ergebnissen aus Tabelle 5.4 lassen sich nun die Messgrößen Precision, Recall und F-Measure berechnen. Precision gibt prozentual an, wie viele der erkannten Codeklone tatsächlich zusammenführbare Codeklone sind. Recall gibt prozentual an, wie viele der tatsächlichen zusammenführbaren Codeklone auch erkannt wurden. F-Measure kombiniert Precision und Recall, um aus beiden Werten das harmonische Mittel zu bilden.

$$Precision = \frac{true\ positives}{true\ positives + false\ positives} = \frac{26}{26+2} \approx 0,93$$

$$Recall = \frac{true\ positives}{true\ positives + false\ negatives} = \frac{26}{26+28} \approx 0,48$$

$$F{-}Measure = 2 * \frac{Precision * Recall}{Precision + Recall} = 2 * \frac{0,93 * 0,48}{0,93 + 0,48} \approx 0,63$$

CPD hat mit 0,93 einen hohen Wert bei Precision, während Recall nur bei 0,48 liegt. Der Wert für Recall würde sich erhöhen lassen, indem die Anzahl der Mindesttokens verringert wird. Dadurch würden mehr tatsächliche zusammenführbare Codeklone auch als solche erkannt werden. Gleichzeitig würde damit aber Precision sinken, da mehr nicht zusammenführbare Methoden als Codeklone erkannt werden.

Als nächstes bestimme ich Precision, Recall und F-Measure für die Ergebnisse der Variabilitätsanalyse des MBCC Frameworks. Die Bestimmung von Precision, Recall und F-Measure für die Ergebnisse der Variabilitätsanalyse kann jedoch nicht auf gleiche Art erfolgen, wie bei CPD. Bei CPD gibt es nur zwei mögliche Ergebnisse, entweder handelt es sich um einen Codeklon ("ja") oder es handelt sich nicht um einen Codeklon ("nein"). Bei der Variabilitätsanalyse wird jedoch ein Ähnlichkeitswert angegeben, der ausdrückt zu wie viel Prozent zwei Methoden übereinstimmen. Deshalb werde ich eine Grenze festlegen, ab welchem Ähnlichkeitswert es sich um einen zusammenführbaren Codeklon handelt. Alle Vergleiche mit einem Ähnlichkeitswert über dieser Grenze werden als zusammenführbare Codeklone klassifiziert (entspricht dem "ja" bei CPD). Alle Vergleiche mit einem Ähnlichkeitswert unter dieser Grenze werden als Methoden, deren Zusammenführung nicht sinnvoll ist, klassifiziert (entspricht dem "nein" bei CPD). Die Festlegung erfolgt jedoch nicht willkürlich, stattdessen werde ich insgesamt zehn Grenzen festlegen und für jede einzelne Grenze Precision, Recall und F-Measure ausrechnen und miteinander vergleichen. Die erste Grenze wird 0,9 sein. Jeder Methodenvergleich mit einem Ähnlichkeitswert über 0,9 wird als zusammenführbare Codeklone klassifiziert ("ja") und jeder Methodenvergleich mit 0,9 oder niedriger wird mit "nein" klassifiziert. Die Grenzen haben einen Abstand von 0,1, sodass die zweite Grenze 0,8 ist und so weiter. Die letzte Grenze ist 0,0 und für diese Grenze werden alle Methodenvergleiche als Codeklone klassifiziert. In Tabelle 5.5 wird für jede Grenze das Ergebnis für Precision, Recall und F-Measure aufgeführt.

| Grenze | Precision | Recall | F-Measure |
|---|---|---|---|
| 0,9 | 1,0 | 0,19 | 0,32 |
| 0,8 | 1,0 | 0,37 | 0,54 |
| 0,7 | 1,0 | 0,56 | 0,72 |
| 0,6 | 1,0 | 0,74 | 0,85 |
| 0,5 | 0,92 | 0,85 | 0,88 |
| 0,4 | 0,82 | 0,91 | 0,86 |
| 0,3 | 0,76 | 0,98 | 0,86 |
| 0,2 | 0,68 | 1,0 | 0,81 |
| 0,1 | 0,6 | 1,0 | 0,75 |
| 0,0 | 0,54 | 1,0 | 0,70 |

Tabelle 5.5: Precision, Recall und F-Measure für die Ergebnisse der angepassten Variabilitätsanalyse

Aus Tabelle 5.5 lässt sich entnehmen, dass je niedriger die Grenze gewählt wird,

desto geringer wird der Wert für Precision und desto höher wird der Wert für Recall, was zu erwarten ist. Je niedriger die Grenze gewählt wird, desto mehr Methodenvergleiche werden als Codeklone klassifiziert. Dadurch werden zwar mehr tatsächliche Codeklone auch richtig klassifiziert, jedoch werden auch mehr Methodenvergleiche als Codeklon klassifiziert obwohl es keine sind.

Die 0,5-Grenze hat mit 0,88 den besten Wert für F-Measure. Wenn die 0,5-Grenze für den quantitativen Vergleich von CPD und der Variabilitätsanalyse des MBCC Frameworks gewählt wird, so ist die Variabilitätsanalyse mit Ausnahme von Precision (0,92 bei MBCC gegenüber 0,93 bei CPD) besser als CPD. Wenn die 0,6-Grenze gewählt wird, so ist die Variabilitätsanalyse sogar in allen Bereichen besser, da Precision hier mit 1,0 am höchsten ist und der Recall mit 0,74 immer noch höher ist als von CPD (Recall für CPD ist 0,48). F-Measure ist bei allen Grenzen außer der 0,9-Grenze und 0,8-Grenze für die Variabilitätsanalyse höher. Der quantitative Vergleich ergibt somit, dass die Variabilitätsanalyse besser zusammenführbare Codeklone als CPD erkennt, deren Zusammenführung sinnvoll ist. Dass die Variabilitätsanalyse besser zusammenführbare Codeklone als CPD erkennt, deckt sich mit den Erkenntnissen aus dem qualitativen Vergleich.

### 5.5.3 Einfluss der Metriken auf die Ergebnisse

Beim qualitativen Vergleich zwischen den Ergebnissen des MBCC Frameworks und den Ergebnissen der textbasierten Codeklonerkennung CPD in FeatureIDE wurden die Abweichungen bei den verschiedenen Metriken schon kurz erläutert. Zur Beantwortung der Forschungsfrage *"Welchen Einfluss haben verschiedene Metriken bei der Ermittlung der Ähnlichkeit von zwei Methoden?"* werde ich die Ergebnisse der verschiedenen Metriken nun ausführlicher diskutieren.

Zunächst wird der Einfluss der verschiedenen Metriken auf Methodenebene betrachtet. Bei den Methodenvergleichen habe ich die Beobachtung gemacht, dass das Ergebnis verschiedener Metriken mal mehr und mal weniger voneinander abweicht. Dies wurde beim qualitativen Vergleich zwischen der Variabilitätsanalyse und CPD in Abschnitt 5.5.2 schon festgestellt und kurz erläutert. Im Folgenden werde ich aussagekräftige Codebeispiele genauer betrachten und ich werde die gemachten Beobachtungen konkretisieren und Aussagen zum Einfluss der Metrik auf das Ergebnis formulieren.

In Listing 5.8 wird der Vergleich der Methode `contains` aus der Klasse ApoClockEditorClockStats (Projekt ApoClock) und der Methode `isOpaque` aus der Klasse ApoEntity (Projekt ApoDice) betrachtet. Für diesen Methodenvergleich haben alle Metriken ein ähnliches Ergebnis. So liegt der Ähnlichkeitswert für die 70:30-Metrik bei 0,26, für die 50:50-Metrik bei 0,33 und für die 30:70-Metrik bei 0,39. An diesem Beispiel habe ich die Feststellung gemacht, dass kurze Statements die gleiche Gewichtung haben, wie lange Statements. Bei diesem Vergleich werden zwei sehr kurze Methoden betrachtet (`contains` mit drei Statements und `isOpaque` mit vier Statements). Von der Logik her sind sie unterschiedlich, da zum Beispiel die `IF`-Statements gut erkennbar anders aufgebaut sind. Trotzdem liegt der Ähnlichkeitswert für den Methodenrumpf bei 0,5. Dies liegt daran, dass ein kurzes Statement die gleiche Gewichtung hat wie ein

Listing 5.8: Vergleich zwischen contains (ApoClockEditorClockStats) und isOpaque (ApoEntity)

```
1  /*
2   * Methode contains aus der Klasse ApoClockEditorClockStats (Projekt ApoClock)
3   */
4    public boolean contains(final float x, final float y) {
5      if ((this.x < x) && (this.x + this.width > x) &&(this.y < y) && (this.y + this.height > y)) {
6        return true;
7      }
8      return false;
9    }
10 /*
11  * Methode isOpaque aus der Klasse ApoEntity (Projekt ApoDice)
12  */
13   public boolean isOpaque(int rgb) {
14
15     int alpha = (rgb >> 24) & 0xff;
16     // red = (rgb >> 16) & 0xff;
17     // green = (rgb >> 8) & 0xff;
18     // blue = (rgb ) & 0xff;
19
20     if (alpha == 0) {
21       return false;
22     }
23
24     return true;
25   }
```

langes Statement. In diesem Beispiel gibt es mit `return false;` und `return true;` zwei kurze Statements. Diese Statements können das Ergebnis für den Methodenrumpf (insbesondere für sehr kurze Methoden) verfälschen, da sie die gleiche Gewichtung wie ein sehr langes Statement mir sehr viel Logik haben. Beim Vergleich von zwei Methoden mit kurzem Methodenrumpf sorgen solche typischen kurzen Statements dafür, dass der Ähnlichkeitswert beim Methodenrumpf größer wird. Um dem entgegen zu wirken, sollte eine Metrik gewählt werden, bei der die Methodensignatur höher gewichtet ist als der Methodenrumpf. Eine hohe Gewichtung des Methodenrumpfes ist für Methoden mit langen Statements zu empfehlen.

Noch deutlicher fällt die in Listing 5.8 gemachte Beobachtung bei dem Methodenvergleich in Listing 5.9 auf. Beide Methoden haben hier eine Ähnlichkeit von 100% im Methodenrumpf, da der Methodenrumpf bei beiden Methoden leer ist. Somit hat die Metrik mit der höchsten Gewichtung des Methodenrumpfes den größten Ähnlichkeitswert im Vergleich zu den anderen beiden Metriken. So liegt der Ähnlichkeitswert für die 30:70-Metrik bei 0,74, bei der 50:50-Metrik bei 0,57 und bei der 70:30-Metrik bei 0,40. Trotz eines hohen Ähnlichkeitswertes bei der 30:70-Metrik ist ein Zusammenführen der Methoden nicht sinnvoll, da die Methodensignaturen unterschiedlich sind.

Für den Methodenvergleich in Listing 5.10 ist der Unterschied beim Ergebnis der verschiedenen Gewichtungen am größten. Das Ergebnis bei der 70:30-Metrik ist 0,7, bei der 50:50-Metrik ist das Ergebnis 0,5 und bei der 30:70-Metrik ist das Ergebnis 0,3. Dies liegt daran, dass die Methodensignatur zu 100% übereinstimmt (bis auf die unterschiedlichen Parameternamen), während im Methodenrumpf gar keine übereinstimmenden Statements vorhanden sind. Trotz übereinstimmender Signatur ist eine Zusammenführung dieser beiden Methoden wegen dem unterschiedlichen

Listing 5.9: Vergleich zwischen touchedDragged (ApoClockPuzzleGame) und onFinish (ApoDice)

```
1  /*
2   * Methode touchedDragged aus der Klasse ApoClockPuzzleGame (Projekt ApoClock)
3   */
4    public void touchedDragged(int x, int y, int oldX, int oldY, int finger) {
5
6    }
7  /*
8   * Methode onFinish aus der Klasse ApoDice (Projekt ApoDice)
9   */
10   protected void onFinish( ) {
11   }
```

Listing 5.10: Vergleich zwischen render (ApoButton) und render (ApoEntity)

```
1  /*
2   * Methode render aus der Klasse ApoButton (Projekt ApoClock)
3   */
4    public void render(BitsGLGraphics g, int changeX , int changeY ) {
5      if ( this.isVisible() ) {
6        super.render(g, changeX , changeY);
7      }
8    }
9
10 /*
11  * Methode render aus der Klasse ApoEntity (Projekt ApoMonoAndroid)
12  */
13   public void render(BitsGLGraphics g, int x, int y) {
14     if ((this.getIBackground() != null) && (this.isBVisible())) {
15       g.drawImage(this.iBackground, (this.getX() + x), (this.getY() + y), (this.getX() +
           x + this.getWidth()), (this.getY() + y + this.getHeight()));
16       if (this.isBSelect()) {
17         g.setColor(255, 0, 0);
18         g.drawRect((int) (this.getX() + x), (int) (this.getY() + y), (int) (this.
             getWidth() - 1), (int) (this.getHeight() - 1));
19       }
20     }
21   }
```

Methodenrumpf wenig sinnvoll. Je höher die Methodensignatur gewichtet ist, desto mehr Methodenvergleiche dieser Art würden einen hohen Ähnlichkeitswert bekommen, obwohl eine Zusammenführung solcher Methoden nicht sinnvoll wäre. In Listing 5.10 wurden die Übereinstimmungen im Quellcode grün markiert, da ein Großteil der Methoden nicht übereinstimmt.

Für den Methodenvergleich in Listing 5.11 ist die in Listing 5.10 gemachte Beobachtung sogar noch deutlicher, weil hier eine kleine Methode mit einer deutlich größeren Methode verglichen wird. Das Ergebnis der 70:30-Metrik liegt für diesen Methodenvergleich bei 0,47, für die 50:50-Metrik bei 0,33 und für die 30:70-Metrik bei 0,20. Es gibt jedoch nur in der Methodensignatur Übereinstimmungen und der Methodenrumpf der zweiten Methode ist um ein Vielfaches länger als der Methodenrumpf der ersten Methode. Daher scheint ein Ähnlichkeitswert von 0,47 bei der 70:30-Metrik immer noch zu hoch und die Ähnlichkeitswerte für die 50:50-Metrik und die 30:70-Metrik sind plausibler. In Listing 5.11 wurden die Übereinstimmungen im Quellcode grün markiert, da ein Großteil der Methoden nicht übereinstimmt.

Listing 5.11: Vergleich zwischen setButtonFunction (ApoMonoPanel) und touchedButton (ApoSnakeEditor)

```
1  /*
2   * Methode setButtonFunction aus der Klasse ApoMonoPanel (Projekt ApoMonoAndroid)
3   */
4   public void setButtonFunction(final String function) {
5     if (super.getModel() != null) {
6       super.getModel().touchedButton(function);
7     }
8   }
9
10 /*
11  * Methode touchedButton aus der Klasse ApoSnakeEditor (Projekt ApoSnake)
12  */
13  public void touchedButton(String function) {
14    if (function.equals(ApoSnakeEditor.BACK)) {
15      this.onBackButtonPressed();
16    } else if (function.equals(ApoSnakeEditor.TEST)) {
17      String levelString = this.getLevelString();
18      BitsLog.d("levelString", levelString);
19      this.getGame().setPuzzleGame(-1, levelString, false);
20    } else if (function.equals(ApoSnakeEditor.UPLOAD)) {
21      this.setLevelSolved(false);
22      this.uploadString = new ApoSnakeString(240, 470, 20, "Uploading␣...", true, 200,
        true);
23
24      Thread t = new Thread(new Runnable() {
25
26        public void run() {
27          ApoSnakeEditor.this.uploadString();
28        }
29      });
30      t.start();
31    } else if (function.equals(ApoSnakeEditor.XMINUS)) {
32      if (this.level[0].length - 1 >= 3) {
33        this.newLevelSize(this.level[0].length - 1, this.level.length);
34      }
35    } else if (function.equals(ApoSnakeEditor.XPLUS)) {
36      if (this.level[0].length + 1 <= 14) {
37        this.newLevelSize(this.level[0].length + 1, this.level.length);
38      }
39    } else if (function.equals(ApoSnakeEditor.YMINUS)) {
40      if (this.level.length - 1 >= 3) {
41        this.newLevelSize(this.level[0].length, this.level.length - 1);
42      }
43    } else if (function.equals(ApoSnakeEditor.YPLUS)) {
44      if (this.level.length + 1 <= 14) {
45        this.newLevelSize(this.level[0].length, this.level.length + 1);
46      }
47    }
48  }
```

Als abschließende Feststellung muss noch erwähnt werden, dass die Gewichtung von Methodensignatur und Methodenrumpf keine große Rolle spielt, falls die Ähnlichkeitswerte für die Methodensignatur und dem Methodenrumpf sehr nah beieinander liegen.

Als Nächstes werden die Gesamtergebnisse der Abbildungen 5.1 bis 5.7 näher betrachtet. In den Abbildungen 5.1 und 5.2 ist das Gesamtergebnis auf Methodenebene zu sehen. Tendenziell lässt sich für alle Metriken ein exponentieller Anstieg beobachten und dass die meisten Methodenvergleiche einen Ähnlichkeitswert von 0,0 bis 0,1 haben. Dabei fällt die 30:70-Metrik auf (gelbe Balken). Die 30:70-Metrik hat sowohl in den ersten drei Bereichen (0,7 bis 1,0) als auch im letzten Bereich (0,0 bis 0,1) die größte Anzahl an Methodenvergleichen im Vergleich zu den anderen beiden Metriken. Während in allen anderen Bereichen die 30:70-Metrik am wenigsten Methodenvergleiche hat. Erklären lässt sich dies durch die vielen Methoden, bei denen der Methodenrumpf leer ist, wie die beiden Methoden in Listing 5.9.

Wenn zwei Methoden mit leerem Methodenrumpf miteinander verglichen werden, stimmen sie im Methodenrumpf zu 100% überein. Für die 30:70-Metrik hat das zur Folge, dass Methoden mit leerem Methodenrumpf einen Ähnlichkeitswert von mindestens 0,7 haben, da der Ähnlichkeitswert von 1,0 für den Methodenrumpf zu 70% gewichtet wird. Dadurch ist auch der Abfall bei der Anzahl an Methodenvergleiche von den Bereichen 0,7 bis 0,8 (13498 Elemente) zu 0,6 bis 0,7 (78 Elemente) für die 30:70-Metrik erklärbar, wie in Abbildung 5.2 zu sehen ist. Ähnliches lässt sich auch für die 50:50-Metrik für die Bereiche 0,5 bis 0,6 (10561 Elemente) und 0,4 bis 0,5 (6464 Elemente) beobachten, da bei der 50:50-Metrik zwei Methoden mit leerem Methodenrumpf einen Ähnlichkeitswert von mindestens 0,5 haben. Die 70:30-Metrik hat von allen Metriken im letzten Bereich (0,0 bis 0,1) die geringste Anzahl an Methodenvergleichen. Dies lässt sich anhand der hohen Gewichtung für die Methodensignatur erklären. Selbst wenn der Methodenrumpf sowie der Methodenname und die Methodenparameter zwischen zwei Methoden komplett unterschiedlich sind, besteht doch die Möglichkeit, dass sie beim Rückgabewert (`int`, `float`, `String` usw.) oder bei der Sichtbarkeit (`public`, `protected` usw.) übereinstimmen. Dies liegt daran, dass die Auswahl an Rückgabewerten und Sichtbarkeit begrenzt ist, welche Teil der Methodensignatur sind. Dadurch gibt es für die 70:30-Metrik im Vergleich zu den anderen beiden Metriken die geringste Anzahl an Elementen im Bereich 0,0 bis 0,1. Aus diesem Grund ist von einer zu hohen Gewichtung des Rückgabewertes und der Sichtbarkeit, im Vergleich zu den anderen Bestandteilen der Methodensignatur, abzuraten.

Die Ergebnisse auf Klassenebene (Abbildungen 5.3 und 5.4) ähneln sich vom Trend her den Ergebnissen auf Methodenebene. Dies ist zu erwarten, da die Ergebnisse auf Klassenebene sich aus den Ergebnissen der Methodenebene aggregieren. Ein Unterschied zu den Ergebnissen auf Methodenebene lässt sich jedoch in den vorderen Bereichen (Abbildung 5.4) erkennen. So hat die 70:30-Metrik in allen vorderen Bereichen die größte Anzahl an Vergleichen, während die 30:70-Metrik die geringste Anzahl hat. Auf Methodenebene war dies noch umgekehrt, hier hatte die 30:70-Metrik für die ersten drei Bereiche (0,7 bis 1,0) die größte Anzahl an Vergleichen. Der Grund dafür liegt daran,

dass für den Vergleich auf Klassenebene immer nur die höchsten Vergleichswerte auf Methodenebene gewählt werden und jede Methode nur einmal für die Ermittlung des Ähnlichkeitswerts zwischen zwei Klassen genommen werden darf, wie es in Abschnitt 3.3.3 erklärt wurde. Selbst wenn es viele Methoden mit leerem Methodenrumpf in beiden zu vergleichenden Klassen gibt, wird eine Methode nur einmal ausgewählt und die anderen hohen Ähnlichkeitswerte zwischen Methoden mit leerem Methodenrumpf werden nicht berücksichtigt. Dadurch sinkt der Einfluss von Methoden mit leerem Methodenrumpf auf das Ergebnis für die Ähnlichkeit von zwei Klassen.

Auch auf Packageebene liegen die Vergleichswerte für die 70:30-Metrik tendenziell am höchsten, wie in den Abbildungen 5.5 und 5.6 ersichtlich ist, da die Ergebnisse auf Packageebene sich aus den Ergebnissen der Klassenebene aggregieren. Auf Projektebene ist in Tabelle 5.1 erkennbar, dass die 70:30-Metrik den höchsten Wert für jeden Vergleich von zwei Projekten hat, während die 30:70-Metrik den geringsten Wert für jeden Vergleich hat. Erklären lässt sich diese Tendenz, weil bei der ausgewählten Fallstudie viel Programmcode durch Clone-And-Own entstanden ist. Es ist anzunehmen, dass die Methodensignatur im Vergleich zum Methodenrumpf eher unverändert blieb. Wie bei dem Methodenvergleich in Listing 5.10 wurden Anpassungen eher im Methodenrumpf durchgeführt als in der Methodensignatur. Dadurch hat die Metrik mit der höchsten Gewichtung bei der Methodensignatur auch den höchsten Ähnlichkeitswert auf Projektebene.

Der Einfluss verschiedener Metriken auf das Ergebnisse der Variabilitätsanalyse lässt sich folgendermaßen zusammenfassen. Eine hohe Gewichtung des Methodenrumpfes ist für Methoden mit kleinem oder leerem Methodenrumpf nicht empfehlenswert, da die Unterschiede in der Methodensignatur nicht groß genug gewichtet werden (siehe Listing 5.8 und 5.9). Umgekehrt gilt jedoch auch, dass eine hohe Gewichtung der Methodensignatur für Methoden mit großem Methodenrumpf nicht gut ist (siehe Listing 5.10). Deshalb wird für Projekte, die viele Methoden mit kleinem Methodenrumpf haben, eine Metrik mit hoher Gewichtung der Methodensignatur empfohlen. Für Projekte, die viele Methoden mit großem Methodenrumpf haben, wird eine hohe Gewichtung des Methodenrumpfes empfohlen. Für Projekte, bei denen die Anzahl an Methoden mit kurzem und langem Methodenrumpf ausgeglichen ist, wird eine ausgeglichene Gewichtung empfohlen.

Eine Alternative zu einer Metrik für alle Methodenvergleiche wäre eine dynamische Gewichtung von Methodensignatur und Methodenrumpf in Abhängigkeit von der Größe des Methodenrumpfes. Bei einer dynamischen Gewichtung würde der Methodenrumpf höher gewichtet, je größer der Methodenrumpf ist. Dadurch würde es keine Nachteile mehr für Methoden mit kleinem oder großem Methodenrumpf geben, da für jede Methodenrumpfgröße immer eine passende Metrik angewendet wird.

Bei der Ermittlung des Ähnlichkeitswertes für den Methodenrumpf sollte zudem die Länge der Statements mit einbezogen werden. Lange Statements sollten eine höhere Gewichtung wie kurze Statements haben, da Übereinstimmungen bei kurzen Statements weniger wichtig bei der Zusammenfassung sind, als Übereinstimmungen bei langen Statements.

Für die drei ausgesuchten Metriken (70:30-Metrik, 50:50-Metrik und 30:70-Metrik) gilt jedoch, dass der Einfluss der Gewichtung auf den höheren Ebenen (Klassenebene, Packageebene und Projektebene) abnimmt. Für die Aggregation auf Klassenebene werden nämlich nur die Methodenvergleiche mit den höchsten Ähnlichkeitswerten ausgewählt. Ein hoher Ähnlichkeitswert erfordert jedoch eine hohe Übereinstimmung in der Methodensignatur und dem Methodenrumpf. Ähnliches gilt für die Aggregation auf Packageebene und Projektebene. Auf Projektebene ist der Unterschied zwischen 30:70-Metrik und 70:30- Metrik sogar so gering, dass der größte Abstand bei einem Projektvergleich nur 4% beträgt (siehe Tabelle 5.2 zwischen den Projekten myTreasureAndroid und ApoMonoAndroid).

### 5.5.4 Zusammenführung von Codeklonen

Zur Beantwortung der dritten Forschungsfrage *"Ab welchem Ähnlichkeitsgrad ist die Zusammenführung von geklonten Programmelementen sinnvoll?"* werde ich die Ergebnisse des quantitativen Vergleichs nutzen. Die Empfehlung, ab welchem Ähnlichkeitsgrad eine Zusammenführung von Methoden sinnvoll ist, erfolgt anhand der Erkennungsrate, ob für diesen Ähnlichkeitsgrad wirklich ein zusammenführbarer Codeklon vorliegt oder nicht. Die Zahlen, die ich zur Beantwortung dieser Frage nutze, stammen aus der Tabelle 5.5, welche die Werte für Precision, Recall und F-Measure für verschiedene Grenzen angibt.

Welche Grenze am besten ist, hängt davon ab, was der Entwickler, welcher die Methoden zusammenführen soll, für Ergebnisse haben möchte. Wenn der Entwickler möglichst alle Codeklone finden möchte und es ihm egal ist, ob andere Methodenvergleiche falsch klassifiziert werden, da deren Zusammenführung nicht sinnvoll ist, so ist die 0,3-Grenze oder gar 0,2-Grenze am besten. Für diese beiden Grenzen werden 98%, beziehungsweise 100%, aller zusammenführbarer Codeklone gefunden (Wert für Recall bei 0,98, bsw. 1,0). Bei diesen Grenzen sind jedoch 24% (0,3-Grenze), beziehungsweise 32%, der Methodenvergleiche nicht für eine Zusammenführung geeignet. Die Aussortierung dieser false-positives würde somit einen entsprechenden Mehraufwand bedeuten.

Wenn es dem Entwickler wichtig ist, dass er möglichst wenig oder keine false-positives aussortieren muss, wäre die 0,6-Grenze am besten geeignet. Für diese Grenze liegt Precision mit 1,0 am höchsten und es werden 74% aller zusammenführbarer Codeklone gefunden (Recall bei 0,74).

Am besten ist die Auswahl einer Grenze, die einen Kompromiss zwischen Aussortierung von false-positives und dem Finden möglichst vieler zusammenführbarer Codeklone vereint. Dies lässt sich mit F-Measure am besten bestimmen, da F-Measure das harmonische Mittel zwischen Precision und Recall bildet. Die Grenze mit dem besten Wert für F-Measure wäre mit 0,88 die 0,5-Grenze. Ab einem Ähnlichkeitsgrad von mehr als 50% ist die Zusammenführung von Methoden somit am sinnvollsten.

### 5.5.5 Threats to Validity

Zum Abschluss der Evaluation werde ich in diesem Abschnitt die Gefahren für die Validität (Threats to Validity) der Evaluation diskutieren. Die Gefahren für die Validität zeigen, wie das Ergebnis meiner durchgeführten Evaluation durch das ausgewählte Vorgehen beeinflusst wurde und ob sich die Ergebnisse auch auf andere Anwendungsgebiete übertragen lassen. Die Gefahren für die Validität unterteilen sich in interne (Internal) und äußere (External) Gefahren.

**Threats to Internal Validity**

Durch das Aufführen der internen Gefahren für die Validität zeige ich, welche Eigenschaften beim Aufbau meiner Evaluation die Ergebnisse der Evaluation beeinflusst haben könnten. Dazu zählt zum einen die Implementierung. Die Motivation zu dieser Arbeit bestand darin, zu prüfen, wie die modellbasierte Variabilitätsanalyse des MBCC Frameworks dazu genutzt werden kann, um die Funktionalitäten von Projekten, die durch Clone-And-Own entstanden sind, zu einer SPL zusammenzuführen. Deshalb wurde in der Implementierung bei der Berechnung zur Ähnlichkeit von zwei Klassen nur die Methodenähnlichkeit herangezogen, da diese die Funktionalitäten eines Programmes umfassen. Die Ähnlichkeit von Parametern einer Klasse wurde nicht herangezogen. Als Folge dieser Entscheidung wird für Klassen, welche keine Methoden beinhalten, kein Ähnlichkeitswert zu allen anderen Klassen gebildet. Klassen ohne Methoden sind somit nicht im Ergebnis der Analyse enthalten und die Parameter einer Klasse haben keinen Einfluss auf das Ergebnis, selbst wenn sie vollkommen unterschiedlich sein sollten. Von den ursprünglichen 117 Klassen der Fallstudie wurden 114 Klassen für die Analyse genutzt. Die drei Klassen ohne Methoden sind `ApoClockConstans`, `ApoDiceConstans` und `ApoSnakeConstans`, welche nur projektspezifische Konstanten enthalten. Der Prozentsatz von Klassen ohne Methoden beträgt 2,66% für die Fallstudie der ApoGames-Reihe und ist somit vernachlässigbar. Sollte die in dieser Arbeit vorgestellte Vorgehensweise auf andere Fallstudien, die viele Klassen ohne Methoden umfasst, angewendet werden, muss diese Vereinfachung bedacht werden.

Ebenfalls muss bedacht werden, dass die Ähnlichkeit zwischen Methodensignaturen nur mit der Metrik aus Listing 5.1 durchgeführt wurde und keiner anderen Metrik. Bei Veränderung dieser Metrik würden sich somit auch die Ergebnisse ändern. Außerdem konnte auf Grund des Umfangs der Fallstudie nur ein kleiner Teil der Ergebnisse zur Ähnlichkeit von Methoden für den qualitativen Vergleich herangezogen werden. So wurden insgesamt nur 100 von 922.352 Methodenvergleichen (entspricht 0,011%) detailliert betrachtet. Die Betrachtung anderer Methodenvergleiche würde das Ergebnis möglicherweise beeinflussen.

Auch das Auswahlverfahren kann eine Rolle bei der Ermittlung des Ergebnisses gespielt haben. Die Methodenvergleiche wurden anhand ihrer Ähnlichkeitswerte für die 50:50-Metrik ausgewählt. Das Ergebnis würde durch Auswahl mit einer anderen Metrik möglicherweise anders sein. Außerdem wurde das Auswahlverfahren so gewählt, dass möglichst viele Methodenvergleiche mit unterschiedlichen Ähnlichkeitswerten ausgewählt werden. Beim vorgestellten Auswahlverfahren aus Abschnitt 5.3 wurden

nur 20% der Methodenvergleiche mit einem Wert von 0,0 bis 0,2 ausgewählt. Wenn das Auswahlverfahren komplett zufällig sein würde, so würden voraussichtlich 89,3% der Methodenvergleiche aus dem Bereich 0,0 bis 0,2 stammen, da insgesamt 823.644 von 922.352 Methodenvergleiche für die 50:50-Metrik einen Ähnlichkeitswert von 0,0 bis 0,2 haben (siehe Abbildung 5.1). Für Methodenvergleiche mit einem Ähnlichkeitswert von 0,0 bis 0,2 muss angenommen werden, dass es sich nicht um Codeklone handelt (siehe Tabelle 5.3). Ein Vorteil der von mir genutzten Auswahlmethodik liegt jedoch daran, dass es sich bei 54% der zufällig ausgesuchten Methoden um Codeklone handelt. Diese 54 Codeklone haben gezeigt, dass die Variabilitätsanalyse Codeklone besser erkennt als CPD. Bei einer komplett zufälligen Auswahl wären voraussichtlich nur 10 Codeklone ausgewählt wurden, wodurch ein weniger repräsentatives Ergebnis beim Vergleich der Variabilitätsanalyse mit CPD sich ergeben hätte.

Eine weitere interne Gefahr stellt die Einstufung der zufällig ausgewählten Methodenvergleiche, welche für den qualitativen und quantitativen Vergleich genutzt wurden, dar. Die Einstufung, ob die Zusammenführung von zwei Methoden sinnvoll ist oder nicht, wurde nur von mir durchgeführt. Diese Einstufung erfolgte jedoch auf einem objektiven Kriterium, nämlich das Kriterium, ob es sich um einen Type-I-Klon, Type-II-Klon oder Type-III-Klon handelt, da sich Typ-II- und Typ-III-Klone nach einer Anpassung des Programmcodes für eine Zusammenführung eignen. Typ-I-Klone können sogar ohne Anpassung zusammengeführt werden, da die Klone identisch sind. Eine vollständige Objektivität kann jedoch nicht gewährleistet werden, da ich auch die Einstufung der Type-II- und Type-III-Klone alleine vorgenommen habe. Die Bewertung von weiteren Personen zur Einschätzung der in den Codebeispielen vorgestellten Methodenvergleiche würde die Objektivität der Einstufung zusätzlich garantieren und den Einfluss auf die Ergebnisse verringern.

**Threats to External Validity**

Bei den externen Gefahren für die Validität geht es darum, zu diskutieren, inwiefern die Erkenntnisse aus der Evaluation auf andere Systeme übertragbar sind. Bei der Fallstudie handelt es sich um in Java geschriebene Android-Spiele. Da nur eine Fallstudie durchgeführt wurde, kann nicht garantiert werden, ob sich die Erkenntnisse aus der Evaluation auch auf andere Programmiersprachen, wie zum Beispiel C++, Python oder Perl, übertragen lassen. Ob die Variabilitätsanalyse des MBCC Frameworks im Vergleich zu CPD für jede Programmiersprache besser ist, kann somit nicht garantiert werden. Ob sich die Erkenntnisse aus der Evaluation auch auf andere Domänen, also nicht nur Android-Spiele, übertragen lässt, ist schwer abzuschätzen. Für eine verlässliche Aussage wäre die Durchführung weiterer Fallstudien mit anderen Domänen notwendig.

## 5.6 Zusammenfassung

In diesem Kapitel wurde die Evaluation anhand der ApoGames-Reihe als Fallstudie vorgenommen. Zur Durchführung wurde die Variabilitätsanalyse des MBCC Frameworks mit drei verschiedenen Metriken durchgeführt und die Ergebnisse wurden zur Beantwortung der Forschungsfragen mit denen einer textbasierten Codeklonerkennung verglichen.

Anhand des qualitativen und des quantitativen Vergleichs wurde gezeigt, dass sich die Variabilitätsanalyse des MBCC Frameworks zum Aufspüren von zusammenführbaren Codeklonen besser eignet als CPD. Außerdem wurde der Einfluss verschiedener Metriken auf das Ergebnis der Variabilitätsanalyse aufgezeigt und es wurde die Empfehlung ausgesprochen, dass ab einem Ähnlichkeitsgrad von mehr als 50% die Zusammenführung von Programmcode sinnvoll ist. Insgesamt zeigt die Evaluation, dass ein Entwickler durch die Ergebnisse der Variabilitätsanalyse zur Zusammenführung von Programmcode zu einer SPL gut unterstützt wird. Es wurden jedoch auch einige Schwächen der Variabilitätsanalyse gezeigt, die noch verbessert werden können. Es hat sich herausgestellt, dass eine hohe Gewichtung der Methodensignatur für den Vergleich von Methoden mit großem Methodenrumpf weniger sinnvoll ist. Im Gegenzug ist es weniger sinnvoll eine Gewichtung des Methodenrumpfes für den Vergleich von Methoden mit einem kleinen Methodenrumpf zu wählen.

# 6 Verwandte Arbeiten

Das Ziel dieser Arbeit ist zu zeigen, wie die Arbeit von Entwicklern, welche eine Zusammenführung von durch Clone-And-Own entstandenen Programmcodes zu einer SPL, mit Hilfe der Variabilitätsanalyse des MBCC Frameworks, erleichtert werden kann. Anhand der ApoGames-Fallstudie wurden die Ergebnisse der Variabilitätsanalyse evaluiert und ausführlich mit den Ergebnissen von CPD verglichen. In diesem Kapitel werden verwandte Arbeiten, die sich mit der Evaluation und dem Vergleich von Codeklonerkennungsverfahren beschäftigen, aufgeführt.

In der Arbeit von Roy et al. [RCK09] wird ein Überblick über verschiedene Codeklonerkennungsverfahren aufgezeigt und wie sich Codeklonerkennungsverfahren im Laufe der Zeit entwickelt haben. Darüber hinaus werden verschiedene Codeklonerkennungsverfahren qualitativ miteinander verglichen. Dabei wurden textbasierte, Token-basierte, Graphen-basierte, Metrik-basierte und AST-basierte Verfahren zur Codeklonerkennung miteinander verglichen. Der Vergleich erfolgte anhand von verschiedenen Szenarien und wie gut die Codeklonerkennungsverfahren bei den verschiedenen Szenarien abschnitten. Außerdem wird ein Schema für eine Klassifizierung vorgestellt und die miteinander verglichenen Codeklonerkennungsverfahren werden daran klassifiziert.

Die Arbeit von Bellon et al. [BKA+07] beschäftigt sich mit dem Vergleich von verschiedenen Codeklonerkennungstools. Anhand von acht großen C- und Java-Programmen wurden für sechs Codeklonerkennungstools, welche textbasierte, Token-basierte, Graphen-basierte, Metrik-basierte und AST-basierte Verfahren umgesetzt haben, die Ergebnisse der Codeklonerkennung quantitativ miteinander verglichen. Beim Vergleich wurden die Messgrößen Precision und Recall ermittelt und es wurden die Laufzeiten der Tools miteinander verglichen.

In der Arbeit von Roy und Cordy [RC09] wird ein Framework zur empirischen Evaluation von Codeklonerkennungstools vorgestellt. Dieses Framework ermöglicht auf effektive Weise die automatische Erhebung der Messgrößen Precision und Recall für reale Systeme. Dieses Framework führt die Prüfung der Codeklonerkennungstools in zwei Phasen durch. In der ersten Phase werden aus einer Codebasis zufällige Codeklone generiert, damit in der zweiten Phase quantitativ ermittelt werden kann, wie gut die Codeklonerkennungstools die generierten Codeklone erkennen.

Schon 2004 haben Rysselberghe und Demeyer [RD04] drei verschiedene Codeklonerkennungsverfahren evaluiert. Bei dieser Evaluation wurde herausgefunden, dass das textbasierte Verfahren für einen ersten Überblick über den geklonten Code am besten geeignet ist. Zum Aufspüren von geklonten Subroutinen eignete sich der Graphen-basierte Ansatz am besten und die Token-basierte Technik am besten in Kombination

mit einem feingranularen Refactoringtool genutzt werden kann.

In der Arbeit von Bruntink et al. [BvDvET05] geht es um die Prüfung, ob die Codeklonerkennung für das Aufspüren von Cross-Cutting Concerns in C-Programmen geeignet ist. Die Evaluation erfolgte anhand von fünf spezifischen Cross-Cutting Concerns, die in einem C-Programm manuell gefunden wurden und ob die Codeklonerkennungsverfahren in der Lage waren diese aufzuspüren.

# 7 Zusammenfassung und Ausblick

Clone-And-Own wird bei der Entwicklung von kommerziellen Softwareprodukten häufig angewendet, um Zeit und Kosten bei der Entwicklung zu sparen. Die neu erstellten Softwareprodukte können so unabhängig von den bereits vorhandenen Softwareprodukten, aus denen der Programmcode entnommen wurde, weiterentwickelt werden. Beim Klonen werden jedoch auch Fehler aus den bereits vorhandenen Softwareprodukten übernommen. Das daraus resultierende Problem ist, dass der nachträgliche Wartungs- und Weiterentwicklungsaufwand zur Behebung von Fehlern bei allen wiederverwendeten Programmcodekomponenten die initialen Anfangskosten schon für eine geringe Anzahl an geklonten Varianten aufwiegen und übertreffen. Als Alternative zu Clone-And-Own eignen sich Software-Produktlinien, welche sich einen gemeinsamen Programmcode teilen, welcher durch eine gemeinsame Plattform entwickelt und verwaltet wird. Durch den gemeinsamen Programmcode müssen Änderungen nur an einer zentralen Stelle im Programmcode vorgenommen werden. Der nachträgliche Wechsel von Clone-and-Own zu Software-Produktlinien erfordert jedoch ein gut durchdachtes Konzept, wozu auch die Identifizierung aller Features und deren Zusammenfassung gehört. Die Zusammenfassung der Features kann durch automatische Codeklonerkennungsverfahren unterstützt werden. Dadurch werden Codeklone erkannt, welche mit Refactoring zusammengefasst werden können. Es gibt jedoch mehrere Codeklonerkennungsverfahren, welche sowohl ihre Stärken als auch Schwächen haben.

In dieser Arbeit habe ich den relativ neuen Ansatz einer modellbasierten Codeklonerkennung näher untersucht. Anhand von einem Forschungsprototypen, dem MBCC Framework, habe ich diese Untersuchung vorgenommen, mit dem Ziel zu zeigen, wie die Ergebnisse der Variabilitätsanalyse einen Entwickler beim Aufspüren von zusammenführbaren Programmcode unterstützen kann. Für ein besseres Verständnis habe ich zunächst die Architektur und Arbeitsweise des MBCC Frameworks näher vorgestellt. Darauf aufbauend habe ich die Anforderungen definiert, um eine Inter-System Klonsuche zwischen mehreren durch Clone-And-Own entstandenen Programmvarianten durchführen zu können, damit das Ergebnis einer solchen Inter-System Klonsuche näher untersuchen werden kann. Zur Untersuchung der Ergebnisse der Variabilitätsanalyse wurde ein Konzept erstellt, welches die Überführung der Ergebnisse aus dem JSON-Report in eine relationale Datenbank umfasst. Die Ergebnisdaten werden in der relationalen Datenbank aufgewertet, sodass eine detaillierte statistische Analyse der Ergebnisse ermöglicht wird. Darüber hinaus ist es möglich benutzerdefinierte Reports zu erstellen, welche dem Entwickler bei der Auswertung der Ergebnisdaten unterstützen können

Die Evaluation erfolgte anhand von fünf Spielen aus der ApoGames-Reihe. Diese fünf Spiele sind durch Anwendung von Clone-And-Own entstanden. Nachdem ich die Ergebnisse der Variabilitätsanalyse zwischen den Spielen der ApoGames-Reihe in

eine MySQL-Datenbank importiert hatte, habe ich mit der Analyse der Ergebnisse angefangen. Dabei habe ich die Ergebnisse der Variabilitätsanalyse mit den Ergebnisses einer textbasierten Codeklonerkennung verglichen. Die textbasierte Codeklonerkennung wurde mit CPD durchgeführt. Bei CPD handelt es sich um ein optionales Plugin für FeatureIDE. Die Codeklonerkennung von CPD wird mit der Token-basierten Technik durchgeführt.

Mit Beantwortung der ersten Forschungsfrage habe ich gezeigt, dass die Variabilitätsanalyse Codeklone im Vergleich zu CPD besser erkennt. Der qualitative Vergleich hat dabei gezeigt, dass CPD unter Umständen mehr als einen Codeklon beim Vergleich von zwei Methoden findet. Die Ergebnisse der Variabilitätsanalyse sind hier plausibler und einfacher zu verstehen, da ein Wert angegeben wird, wie ähnlich sich zwei Vergleichsobjekte sind. Im quantitativen Vergleich hat sich dies bestätigt, da die Variabilitätsanalyse bei der Klassifizierung von Codeklonen bessere Werte für die Messgrößen Precision, Recall und F-Measure hat.

Mit der zweiten Forschungsfrage habe ich untersucht, welchen Einfluss die gewählte Metrik auf die Ergebnisse hat. Dabei habe ich festgestellt, dass es beim Vergleich von Methoden sinnvoller ist, wenn bei Methoden mit einem kleinen Methodenrumpf die Methodensignatur höher gewichtet ist als beim Vergleich von Methoden mit einem großen Methodenrumpf, wo es sinnvoller ist den Methodenrumpf höher zu gewichten. Dies liegt daran, dass schon kleine Änderungen an einer Stelle im Programm, wie das umbenennen einer Variablen, bei Methoden mit kleinem Methodenrumpf einen viel größeren Einfluss auf den Ähnlichkeitswert haben, als bei Methoden mit großem Methodenrumpf.

Durch das Beantworten der dritten Forschungsfrage habe ich die Empfehlung gegeben, dass die Zusammenführung von Methoden mit einem Ähnlichkeitsgrad von mehr als 50% am sinnvollsten ist. Wenn alle Methodenvergleiche mit einem Ähnlicheitsgrad von mehr als 50% als zusammenführbare Codeklone klassifiziert werden, ergibt sich für F-Measure mit 0,88 der beste Wert, im Vergleich zu anderen Grenzen, ab der zwei vergleichende Methoden als zusammenführbare Codeklone klassifiziert werden.

Abschließend habe ich durch Aufführung der Gefahren für die Validität gezeigt, dass die Thematik noch nicht abgeschlossen ist und die Durchführung von weiteren Fallstudien notwendig ist.

Das Ergebnis dieser Arbeit lässt sich wie folgt zusammenfassen: Die modellbasierte Codeklonerkennung lässt sich gut zur Unterstützung eines Entwicklers bei der Zusammenführung von Codeklonen nutzen. Die Ergebnisse der modellbasierten Codeklonerkennung sind plausibler im Vergleich zu einer textbasierten Codeklonerkennung. Für den Vergleich von Projekten mit einer großen Anzahl an Methoden mit kleinem Methodenrumpf sollte eine Metrik mit höherer Gewichtung der Methodensignatur gewählt werden. Für den Vergleich von Projekten mit einer großen Anzahl an Methoden mit großem Methodenrumpf sollte eine Metrik mit höherer Gewichtung des Methodenrumpfes gewählt werden. Ab einem Ähnlichkeitsgrad von mehr als 50% ist die Zusammenführung von Methoden zu empfehlen.

### Ausblick für zukünftige Arbeiten

Für Zukünftige Arbeiten gibt es mehrere Themen und Fragestellungen, die sich aus meiner Arbeit ergeben. Dazu zählt die Frage, ob sich die Ergebnisse meiner Arbeit mit den Ergebnissen anderer durchgeführter Fallstudien bestätigen lassen. Die ApoGames-Fallstudie bezieht sich auf Android-Spiele, die mit Java entwickelt wurden, weshalb die Durchführung von Fallstudien mit anderen Programmiersprachen und Anwendungsdomänen zu bevorzugen ist.

In der Fallstudie habe ich festgestellt, dass die Ergebnisse der Variabilitätsanalyse auch von der gewählten Metrik abhängen. Die Länge des Methodenrumpfes ist bei der Wahl der Metrik für die Methodenvergleiche von wichtiger Bedeutung. Für Methoden mit großem Methodenrumpf ist eine höhere Gewichtung des Methodenrumpfes sinnvoller, während für Methoden mit kleinem Methodenrumpf eine höhere Gewichtung der Methodensignatur sinnvoller ist. Da in einem Projekt jedoch Methoden mit unterschiedlichen Größen des Methodenrumpfes vorkommen, muss bei der Auswahl der Metrik immer ein Kompromiss gemacht werden. Deshalb bietet sich die Nutzung einer dynamischen Metrik an. Eine dynamische Metrik würde für Methoden mit einem großem Methodenrumpf den Methodenrumpf höher gewichten und für Methoden mit kleinem Methodenrumpf die Methodensignatur höher gewichten. Zur Feststellung, ob es sich um eine Methode mit großem oder kleinem Methodenrumpf handelt, könnte das modellbasierte Verfahren der Codeklonerkennung um die Token-basierte Technik erweitert werden. Die Größe des Methodenrumpfes wird anhand der Anzahl der Tokens bestimmt. Je mehr Tokens ein Methodenrumpf umfasst, desto höher sollte der Methodenrumpf für den Vergleich gewichtet werden. Anhand der Tokens könnte auch die Ähnlichkeit für den Methodenrumpf bestimmt werden, in dem die Anzahl der übereinstimmenden Anzahl von Tokens im Verhältnis zur Gesamtzahl an Tokens gesetzt wird.

Ein weiteres Thema ist die Integration des MBCC Frameworks in FeatureIDE, sodass die Ergebnisse von CPD und dem MBCC Framework direkt verglichen werden können. Für meine Arbeit musste ich die Ergebnisse für den gleichen Methodenvergleich immer separat für CPD und MBCC Framework heraussuchen, damit ich die Ergebnisse vergleichen konnte. Eine Integration des MBCC Frameworks in FeatureIDE würde weitere Vergleiche von CPD und MBCC Framework erleichtern.

Ein Problem, welches sich während der Durchführung der Fallstudie gezeigt hat, lag in der Skalierbarkeit. So konnten immer nur zwei von fünf Projekten der ApoGames-Reihe mit dem MBCC Framework auf einmal verglichen werden. Die Ergebnisse von jedem Durchlauf wurden am Ende in der MySQL-Datenbank dann wieder zusammengeführt. Deshalb wäre eine Verbesserung des MBCC Frameworks, sodass auch mehrere große Projekte miteinander verglichen werden können, von Vorteil. So müssten die Vergleiche zwischen den Projekten nicht aufgeteilt und separat durchgeführt werden, nur um die Ergebnisse der Vergleiche am Ende wieder zusammenzuführen. Dadurch könnten große Projekte mit einem Durchlauf des MBCC Frameworks verglichen werden und der Aufwand zur Durchführung des Vergleichs mit dem MBCC Framework würde sich für den Nutzer verringern.

Des Weiteren sollte der Importprozess zum Import der Vergleichsdaten aus dem JSON-Report in die MySQL-Datenbank komplett automatisch durchgeführt werden. Wenn alle Schritte des Importprozesses automatisch ausgeführt werden, bedeutet dies weniger Aufwand für den Nutzer und es könnten Fehler beim Importprozess ausgeschlossen werden. Aktuell muss der Importprozess nach jedem durch das MBCC Framework durchgeführten Vergleich manuell initiiert und durchgeführt werden.

# Literaturverzeichnis


[ABKS13] Apel, S.; Batory, D.; Kästner, C.; Saake, G.: *Feature-Oriented Software Product Lines: Concepts and Implementation.* Springer, Berlin, 2013.

[AK09] Apel, S.; Kästner, C.: An overview of feature-oriented software development. *The Journal of Object Technology*, Band 8(5), S. 49–84, 2009.

[AKL13] Apel, S.; Kästner, C.; Lengauer, C.: Language-independent and automated software composition: The featurehouse experience. *IEEE Trans. Softw. Eng.*, Band 39, Nr. 1, S. 63–79, Januar 2013.

[ALMK10] Apel, S.; Lengauer, C.; Möller, B.; Kästner, C.: An algebraic foundation for automatic feature-based program synthesis. *Sci. Comput. Program.*, Band 75, Nr. 11, S. 1022–1047, November 2010.

[Apo] Aporius, D. `http://apo-games.de/index_android.php` Aufgerufen am 19.03.2018.

[Bak95] Baker, B. S.: On finding duplication and near-duplication in large software systems. In *Proceedings of the Second Working Conference on Reverse Engineering*, WCRE '95, S. 86–. IEEE Computer Society, Washington, DC, USA, 1995.

[Bat05] Batory, D.: Feature models, grammars, and propositional formulas. In *Proceedings of the 9th International Conference on Software Product Lines*, SPLC'05, S. 7–20. Springer-Verlag, Berlin, Heidelberg, 2005.

[BKA+07] Bellon, S.; Koschke, R.; Antoniol, G.; Krinke, J.; Merlo, E.: Comparison and evaluation of clone detection tools. *IEEE Trans. Softw. Eng.*, Band 33, Nr. 9, S. 577–591, September 2007.

[BSR03] Batory, D.; Sarvela, J. N.; Rauschmayer, A.: Scaling step-wise refinement. In *Proceedings of the 25th International Conference on Software Engineering*, ICSE '03, S. 187–197. IEEE Computer Society, Washington, DC, USA, 2003.

[BvDvET05] Bruntink, M.; Deursen, A. v.; Engelen, R. v.; Tourwe, T.: On the use of clone detection for identifying crosscutting concern code. *IEEE Trans. Softw. Eng.*, Band 31, Nr. 10, S. 804–818, Oktober 2005.

[BYM+98] Baxter, I. D.; Yahin, A.; Moura, L.; Sant'Anna, M.; Bier, L.: Clone detection using abstract syntax trees. In *Proceedings of the International Conference on Software Maintenance*, ICSM '98, S. 368–. IEEE Computer Society, Washington, DC, USA, 1998.

[CE00] Czarnecki, K.; Eisenecker, U.: *Generative Programming: Methods, Tools, and Applications.* Addison-Wesley Professional, New York, 2000.

[CLO] CLOC. `https://github.com/AlDanial/cloc` Aufgerufen am 19.03.2018.

[CN01] Clements, P.; Northrop, L.: *Software Product Lines: Practices and Patterns.* Addison-Wesley Professional, 2001.

[CPD] CPD. `http://pmd.sourceforge.net/pmd-4.3.0/cpd.html` Aufgerufen am 19.03.2018.

[DB07] Dalgarno, M.; Beuche, D.: Variant management. *In The British Computer Society. 3rd British Computer Society Configuration Management Specialist Group Conference. Oxford*, S. 6, 2007.

[FBB+99] Fowler, M.; Beck, K.; Brant, J.; Opdyke, W.; Roberts, D.: *Refactoring: Improving the Design of Existing Code.* Addison-Wesley Professional, 1999.

[FMS+17] Fenske, W.; Meinicke, J.; Schulze, S.; Schulze, S.; Saake, G.: Variant-preserving refactorings for migrating cloned products to a product line. In *IEEE 24th International Conference on Software Analysis, Evolution and Reengineering, SANER 2017, Klagenfurt, Austria, February 20-24, 2017*, S. 316–326, 2017.

[FTS13] Fenske, W.; Thüm, T.; Saake, G.: A taxonomy of software product line reengineering. In *Proceedings of the Eighth International Workshop on Variability Modelling of Software-Intensive Systems*, VaMoS '14, S. 4:1–4:8. ACM, New York, NY, USA, 2013.

[GFGP06] Geiger, R.; Fluri, B.; Gall, H. C.; Pinzger, M.: Relation of code clones and change couplings. In *Fundamental Approaches to Software Engineering, 9th International Conference, FASE 2006, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2006, Vienna, Austria, March 27-28, 2006, Proceedings*, S. 411–425, 2006.

[Hei09] Heidenreich, F.: Jamopp: The java model parser and printer., 2009.

[HKI08] Higo, Y.; Kusumoto, S.; Inoue, K.: A metric-based approach to identifying refactoring opportunities for merging code clones in a java software system. *J. Softw. Maint. Evol.*, Band 20, Nr. 6, S. 435–461, November 2008.

[JDHW09] Juergens, E.; Deissenboeck, F.; Hummel, B.; Wagner, S.: Do code clones matter? In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, S. 485–495. IEEE Computer Society, Washington, DC, USA, 2009.

[KA09] Kästner, C.; Apel, S.: Virtual separation of concerns - a second chance for preprocessors. *The Journal of Object Technology*, Band 8(6), S. 59–78, 2009.

[KAK08] Kästner, C.; Apel, S.; Kuhlemann, M.: Granularity in software product lines. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, S. 311–320. ACM, New York, NY, USA, 2008.

[KCH+90] Kang, K. C.; Cohen, S. G.; Hess, J. A.; Novak, W. E.; Peterson, A. S.: Feature-oriented domain analysis (foda) feasibility study. Technical report, Software Engineering Institute, Carnegie-Mellon University, Pittsburgh, Pennsylvania 15213, November 1990.

[KG06] Kapser, C. J.; Godfrey, M. W.: Supporting the analysis of clones in software systems. *Journal of Software Maintenance and Evolution: Research and Practice*, Band 18(2), S. 61–82, 2006.

[Kos14] Koschke, R.: Large-scale inter-system clone detection using suffix trees and hashing. *J. Softw. Evol. Process*, Band 26, Nr. 8, S. 747–769, August 2014.

[Kri01] Krinke, J.: Identifying similar code with program dependence graphs. In *Proceedings of the Eighth Working Conference on Reverse Engineering (WCRE'01)*, WCRE '01, S. 301–. IEEE Computer Society, Washington, DC, USA, 2001.

[KSNM05] Kim, M.; Sazawal, V.; Notkin, D.; Murphy, G.: An empirical study of code clone genealogies. *SIGSOFT Softw. Eng. Notes*, Band 30, Nr. 5, S. 187–196, September 2005.

[LLMZ04] Li, Z.; Lu, S.; Myagmar, S.; Zhou, Y.: Cp-miner: a tool for finding copy-paste and related bugs in operating system code. In *Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, S. 20–20. USENIX Association, Berkeley, CA, USA, 2004.

[MD08] Mens, T.; Demeyer, S.: *Software Evolution.* Springer, 2008.

[MyS] MySQL. `https://www.mysql.com` Aufgerufen am 08.02.2018.

[MZB+15] Martinez, J.; Ziadi, T.; Bissyandé, T. F.; Klein, J.; Le Traon, Y.: Bottom-up adoption of software product lines: A generic and extensible approach. In *Proceedings of the 19th International Conference on Software Product Line*, SPLC '15, S. 101–110. ACM, New York, NY, USA, 2015.

[Opd92] Opdyke, W., F.: *Refactoring object-oriented frameworks.* Dissertation, University of Illinois at Urbana-Champaign, 1992.

[PBvdL05] Pohl, K.; Böckle, G.; Linden, F. J. v. d.: *Software Product Line Engineering: Foundations, Principles and Techniques.* Springer, New York, 2005.

[php] phpMyAdmin. `https://www.phpmyadmin.net` Aufgerufen am 08.02.2018.

[Pre97] Prehofer, C.: Feature-oriented programming: A fresh look at objects. Technical report, Institut für Informatik, Technische Universität München, 80290 München, Germany, 1997.

[RC07] Roy, C. K.; Cordy, J. R.: A survey on software clone detection research. Technical report, School of Computing, Queen's University, Ontario, Canada, 2007.

[RC09] Roy, C. K.; Cordy, J. R.: A mutation/injection-based automatic framework for evaluating code clone detection tools. In *Proceedings of the IEEE International Conference on Software Testing, Verification, and Validation Workshops*, ICSTW '09, S. 157–166. IEEE Computer Society, Washington, DC, USA, 2009.

[RCC13] Rubin, J.; Czarnecki, K.; Chechik, M.: Managing cloned variants: A framework and experience. In *Proceedings of the 17th International Software Product Line Conference*, SPLC '13, S. 101–110. ACM, New York, NY, USA, 2013.

[RCK09] Roy, C. K.; Cordy, J. R.; Koschke, R.: Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Sci. Comput. Program.*, Band 74, Nr. 7, S. 470–495, Mai 2009.

[RD04] Rysselberghe, F. V.; Demeyer, S.: Evaluating clone detection techniques from a refactoring perspective. In *Proceedings of the 19th IEEE International Conference on Automated Software Engineering*, ASE '04, S. 336–339. IEEE Computer Society, Washington, DC, USA, 2004.

[RKBC12] Rubin, J.; Kirshin, A.; Botterweck, G.; Chechik, M.: Managing forked product variants. In *16th International Software Product Line Conference, SPLC '12, Salvador, Brazil - September 2-7, 2012, Volume 1*, S. 156–160, 2012.

[SAK10] Schulze, S.; Apel, S.; Kästner, C.: Code clones in feature-oriented software product lines. In *Proceedings of the Ninth International Conference on Generative Programming and Component Engineering*, GPCE '10, S. 103–112. ACM, New York, NY, USA, 2010.

[SBDT10] Schaefer, I.; Bettini, L.; Damiani, F.; Tanzarella, N.: Delta-oriented programming of software product lines. In *Proceedings of the 14th International Conference on Software Product Lines: Going Beyond*, SPLC'10, S. 77–91. Springer-Verlag, Berlin, Heidelberg, 2010.

[Sch13] Schulze, S.: *Analysis and Removal of Code Clones in Software Product Lines.* Dissertation, Otto-von-Guericke-Universität Magdeburg, 2013.

[SJF11] Schulze, S.; Jürgens, E.; Feigenspan, J.: Analyzing the effect of preprocessor annotations on code clones. In *Proceedings of the 11th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM'11)*, S. 115–124, 2011.

[TBK09] Thum, T.; Batory, D.; Kastner, C.: Reasoning about edits to feature models. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, S. 254–264. IEEE Computer Society, Washington, DC, USA, 2009.

[Tie16] Tiede, M.: Modellbasiertes reverse engineering von variabilität im quellcode. Masterthesis, Technischen Universität Braunschweig, 2016.

[TKB+14] Thüm, T.; Kästner, C.; Benduhn, F.; Meinicke, J.; Saake, G.; Leich, T.: Featureide: An extensible framework for feature-oriented software development. *Sci. Comput. Program.*, Band 79, S. 70–85, Januar 2014.

[Vog] Vogel, L. `http://www.vogella.com/tutorials/EclipseEMF/article.html` Aufgerufen am 20.11.2017.

[WTS+16] Wille, D.; Tiede, M.; Schulze, S.; Seidl, C.; Schaefer, I.: Identifying variability in object-oriented code using model-based code mining. In Margaria, T.; Steffen, B. (Hrsg.): *Leveraging Applications of Formal Methods, Verification and Validation: Discussion, Dissemination, Applications*, S. 547–562. Springer International Publishing, Cham, 2016.

# Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und nur mit erlaubten Hilfsmitteln angefertigt habe.

Magdeburg, den 30. April 2018

Philipp Müller