Berechnung einer Sequenz von Editieroperationen zwischen Codefragmenten eines Typ3 Klonpaares

Diplomarbeit

Dimitri Tichanow

Matrikelnummer: 1946985

6.07.2009

Universität Bremen

Fachbereich Mathematik / Informatik
Studiengang Informatik

1. Gutachter: Prof. Dr. Rainer Koschke
2. Gutachter: Prof. Dr. Christian Freksa

Erklärung

Ich versichere, die Diplomarbeit ohne fremde Hilfe angefertigt zu haben. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen entnommen sind, sind als solche kenntlich gemacht.

Bremen, den 6.07.2009

. . . . . .
(Dimitri Tichanow)

Danksagung

Ich bedanke mich bei allen, die mich bei der Erstellung meiner Diplomarbeit unterstützt haben. Insbesondere danke ich meinen Betreuern Herrn Professor Dr. Koschke, Herrn Raimar Falke und Frau Rebecca Tiarks für die gute Betreuung und Zusammenarbeit. Ich danke auch Herrn Bernhard Berger, mit dem ich zahllose interessante Diskussionen geführt habe und der immer ein offenes Ohr für meine Probleme hatte. Darüber hinaus danke ich den Mitarbeitern der Axivion GmbH und der gesamten AG Softwaretechnik. Ebenso danke ich meiner Familie und Anna die mich in jeder Form unterstützt haben und in dieser Zeit viel auf mich verzichten mussten.

INHALTSVERZEICHNIS

KAPITEL 1

Einleitung

Inhalt

In dem ersten Abschnitt des Kapitels wird ein kurzer Einblick in die Problematik der Softwareentwicklung und Softwarewartung gegeben. In dem darauf folgenden Abschnitt wird die Motivation und die Aufgabenstellung beschrieben. Der letzte Abschnitt beschreibt schliesslich die Struktur des Dokuments.

1.1 Hintergrund

Heutzutage wäre die Forschung und Wirtschaft ohne Computer und Softwaresysteme nicht denkbar. Alles wird mit Hilfe von Computern und Softwaresystemen erledigt, seien es einfache alltägliche Tätigkeiten oder komplexe Produktionsprozesse. Da alles dem Wandel unterliegt, werden alte Systeme angepasst und erweitert, oder es werden neue Softwaresysteme entwickelt.

Die Neuentwicklung und vor allem die Weiterentwicklung und Anpassung der Softwaresysteme ist jedoch enorm aufwändig. Nach Boehm in [Boehm, 1981] beansprucht die Neuentwicklung eines Softwaresystems lediglich 20% des Gesamtaufwandes, der Rest wird in die Fehlerbehebung, Anpassung und Weiterentwicklung der Software investiert. Fjedstad und Hamlen in [Fjedstad u. a., 1979] fanden heraus, dass Wartungsprogrammierer ca. 50% ihrer Zeit allein mit der Analyse beschäftigt sind, bevor sie eine Änderung vornehmen und testen können (siehe auch [Koschke, 2007]). Es ist also außerordentlich wichtig bei der Entwicklung von Softwaresystemen auf Aspekte wie Änderbarkeit und Wartbarkeit zu achten, denn wartbare Software ist günstiger in der Weiterentwicklung.

In einer Studie in [Koschke u. a., 2008] zum Thema: Identifikation und Analyse von Softwareklonen werden einige interessante Fakten bezüglich der Entwicklung und Weiterentwicklung von Softwaresystemen geschildert. Dort wird beschrieben, dass die „copy-paste-modify“ Methode für Quelltextdateien, Funktionen und Quelltextfragmente eine weit verbreitete Methode zur praktischen Wiederverwendung in der Softwareentwicklung ist. Man kopiert ein Fragment des Quellcodes, fügt ihn an einer benötigten Stelle ein und passt diesen möglicherweise zusätzlich an. Diesen Vorgang nennt man „code cloning“ und die Kopien des Quellcodes werden als Klone bezeichnet. In nahezu allen Softwaresystemen sind solche Codeklone vorhanden.

Folgen solcher „copy & paste-Programmierung“ sind unter anderem: Fehlerfortpflanzung und schwierigere Fehlerkorrektur, höherer Aufwand für das Testen und Wartung, unnötige Vergrößerung der Quellcodebasis, höherer Aufwand beim Verstehen des Systems und erschwerte Änderbarkeit der Software. Die in der Studie beschriebenen empirischen Untersuchungen zeigen, dass abhängig von der Größe des Softwaresystems, zwischen 10% und 20% des Quellcodes geklont sein könnten, in Ausnahmefällen sogar bis zu 50%. Diese Studien zeigen also, dass die Softwaresysteme oft einen großen Anteil an dupliziertem Quellcode beinhalten, welches folglich die Softwarewartung und Änderung signifikant beeinträchtigen kann. Die Identifikation, Analyse und Behebung der Auswirkungen der Softwareklone ist ein praxisbezogenes Problem und es wird aktiv auf diesem Gebiet geforscht.

Mit dieser Problematik beschäftigt sich das „Bauhaus“-Projekt, welches aus der Zusammenarbeit der Universitäten Stuttgart und Bremen entstand. „Bauhaus“ ist eine Ansammlung von Softwarewerkzeugen, die dem Wartungsingenieur seine Arbeit erleichtern soll (siehe [Plödereder u. a., 2006]). Da diese Diplomarbeit im Rahmen des „Bauhaus“-Projektes entstand, wird im Kapitel Grundlagen 2 genauer darauf eingegangen.

1.2 Aufgabenstellung

Um die negativen Auswirkungen der Softwareklone durch geeignete Maßnahmen beheben zu können, ist es wichtig die Unterschiede zwischen den Kopien zu kennen. Wurde das eingefügte Gegenpart des Originals verändert? Wenn ja, welche Änderungen wurden vorgenommen? Wurden Strukturen hinzugefügt, gelöscht oder geändert? Dem Wartungsingenieur bleibt nichts anderes übrig, als sich die Klonfragmente anzuschauen und die Unterschiede manuell zu analysieren, da in „Bauhaus“ bisher keine Ansätze existieren, die genau diese Aufgabe automatisch erledigen. Erst nach diesem Schritt ist er in der Lage, wenn es möglich ist, einen Refactoringschritt durchzuführen. Im Abschnitt 2.1.4 wird der Begriff „Refactoring“ näher erläutert.

Die Aufgaben dieser Arbeit sind wie folgt unterteilt:

- **Einarbeiten in die Materie**
  Dabei soll Wissen über „Bauhaus“, die Zwischendarstellung IML und die im „Bauhaus“ integrierten Analysewerkzeuge erarbeitet werden. Desweiteren soll nach bereits existierenden Ansätzen zur Lösung der Problemstellung gesucht werden.

- **Auswahl geeigneter Ansätze**
  In diesem Schritt sollen die existierenden Ansätze näher untersucht werden. Es soll evaluiert werden, ob die Ansätze verwendet werden könnten und wie. Ein besonderes Augenmerk soll dabei auf Typ3 2.1.3 Klone gelegt werden.

- **Implementierung eines Systems**
  In diesem Schritt soll eines der geeigneten Ansätze implementiert werden. Dabei soll die Berechnung und Ausgabe der Klonunterschiede an den Benutzer im Vordergrund stehen. Für die Ausgabe an den Benutzer soll eine geeignete Form entwickelt werden.

- **Test und Evaluation des implementierten Systems**
  Hierfür werden mit Hilfe von „Bauhaus“-Werkzeugen aus der, in realen Softwaresystemen, vorhandenen Klonmenge Stichproben entnommen. Für diese Stichproben sollen dann die Unterschiede berechnet, ausgegeben und auf Qualität untersucht werden.

Falls vorhanden, sollen die Verbesserungsmöglichkeiten aufgezeigt und Lösungsvorschläge gemacht werden.

Die Implementierung muss:

- **In der Programmiersprache Ada95 erfolgen**
  Da der größte Teil von „Bauhaus“ in Ada95 implementiert ist und eine Integration der in Ada95 neu implementierten Werkzeuge weitgehend automatisiert wurde, soll das System in Ada95 implementiert werden. Dabei soll diese Programmiersprache in einem für die Implementierung des Ansatzes benötigtem Umfang erlernt werden.
- **Auf der in Bauhaus vorhandenen Zwischendarstellung IML aufbauen**
  Die Zwischendarstellung IML 2.2.1 ist sehr mächtig und mit wenigen Ausnahmen wird diese Datenstruktur von allen „Bauhaus“-Werkzeugen zu Analyse verwendet. Die Implementierung des Ansatzes soll diese Struktur ebenfalls nutzen.

1.3 Aufbau der Arbeit

Nach der Einleitung und Beschreibung der Aufgabenstellung in diesem Kapitel werden in dem Kapitel 2 die Grundlagen geliefert, die notwendig sind, um die Arbeit thematisch einordnen und verstehen zu können. Außer der Grundlagen beinhaltet das Kapitel 2 auch eine Beschreibung der „Bauhaus“-Suite und der Intermediate Language. In anschließendem Kapitel 3 werden die grundlegende Idee zu der Lösung der Problemstellung, ein Überblick über die verwandten Arbeiten und schließlich ein geeigneter Lösungsansatz und seine Erweiterungen vorgestellt und ausführlich beschrieben. In Kapitel 4 wird die Umsetzung/Implementierung eines Ansatzes samt Erweiterungen vorgestellt. Die Beschreibung der Evaluation des implementierten Systems wird in Kapitel 5 vorgestellt. In einem abschließenden Fazit in Kapitel 6 werden die Ergebnisse zusammengefasst und beurteilt. Zusätzlich werden mögliche Verbesserungsmöglichkeiten aufgezeigt, Ideen für deren Beseitigung geliefert sowie Weiterentwicklungen angesprochen.

1.4 Anforderungen an den Leser

In dieser Arbeit wird versucht, die zum Verständnis benötigten Grundlagen detailliert zu erläutern. Der Leser soll in der Lage sein mathematische Notation verstehen zu können. Kenntnisse in der Graphentheorie sind von Vorteil, werden jedoch nicht zwingend benötigt. Kenntnis der allgemeiner programmiersprachlicher Strukturen und der Programmiersprache Ada95 sind äußerst hilfreich. Weiterhin erleichtern Kenntnisse bezüglich der IML und des „Bauhaus“-Projektes das Verständnis.

KAPITEL 2

Grundlagen

Inhalt

In diesem Kapitel werden die für das Verstehen der Arbeit benötigten Grundlagen beschrieben. Zunächst werden die Begriffe der Softwareklone erläutert. Dabei werden der Aufbau und die Typen der Klone näher betrachtet. Es wird auch ein kurzer Einblick in die Thematik der Behebung von Folgen der Klone gegeben. Im zweiten Abschnitt wird auf das „Bauhaus“-Projekt, die Zwischendarstellung IML und seine Analysewerkzeuge eingegangen. In den darauf folgenden Abschnitten werden die Datenstrukturen der Graphen und Bäume eingeführt. Auch einige Algorithmen auf diesen Datenstrukturen, die in den späteren Kapiteln ihre Verwendung finden, werden hier beschrieben.

2.1 Softwareklone

In dem vorherigen Kapitel wurde der Begriff der Softwareklone eingeführt. Leider ist es nicht möglich eine exakte Definition dieses Begriffes zu geben, da es keine scharfe einheitliche Definition in der Literatur gibt. Zusammengefasst handelt es sich bei Softwareklonen um Quellcodefragmente, welche sich in einer oder anderen Form ähnlich sind (siehe [Baxter, 1998], [Kamiya u. a., 2002]). Dabei werden mit dem Begriff der Ähnlichkeit alle Grenzen offen gehalten. Im Folgenden werden Begriffe definiert, die das Aufbau der Softwareklone beschreiben. Desweiteren wird erläutert wie die Softwareklone anhand ihrer Ähnlichkeit über die Klontypen kategorisiert werden.

2.1.1 Codefragment

Die Quelltextkopien werden als Codefragmente bezeichnet und umfassen in den meisten Fällen mehrere Quelltextzeilen. In solchen Fällen ist es notwendig die Start- und Endzeile des Fragments zu kennen. Diese Information wird in dem Konzept der *„source location“*, im Weiteren als SLOC bezeichnet, zusammengefasst. Eine SLOC beinhaltet den absoluten Pfad zu der Datei, in der sich das Codefragment befindet, sowie den Dateinamen und eine Zeilen- und Spaltennummer. Durch Start-SLOC und die End-SLOC wird ein Bereich im Quelltext eingegrenzt, in dem sich der Codefragment befindet. Die Angabe der Start bzw. End-SLOC identifiziert somit eindeutig ein Codefragment.

2.1.2 Klonpaar

Spricht man von dem Klonen, so gehört zu einem Original stets seine Kopie. Die Codefragmente des Originals und der Kopie ergeben zusammen ein Tupel, welches Klonpaar genannt wird. Im Folgenden wird der Begriff Klonpaar synonym für den Begriff Softwareklon bzw. Klon verwendet. Jedes Klonpaar hat einen bestimmten *Typ* X mit $X \in \{1, 2, 3, 4\}$. Die einzelnen Kategorien werden im nächsten Abschnitt des Kapitels 2.1.3 näher betrachtet. Eine Zusammenfassung der Begriffe SLOC, Codefragment und Klonpaar ist in der Abbildung 2.1 dargestellt. Berger beschreibt in [Berger, 2007] treffend die mathematischen Eigenschaften der Klonpaartupel. Diese gelten jedoch nur für Typ1 und Typ2 Klone.

> Klonpaare sind sowohl symmetrisch als auch transitiv. Dies bedeutet, dass, wenn es ein Klonpaar $CP_1 = (CF_1, CF_2)$ gibt, existiert auch das Klonpaar $CP_X = (CF_2, CF_1)$. Durch die Codefragmente ergibt sich die Transitivität. Wenn also die Klonpaare $CP_1 = (CF_1, CF_2)$ und $CP_2 = (CF_2, CF_3)$ vorhanden sind, so gibt es auch ein Klonpaar $CP_3 = (CF_1, CF_3)$.

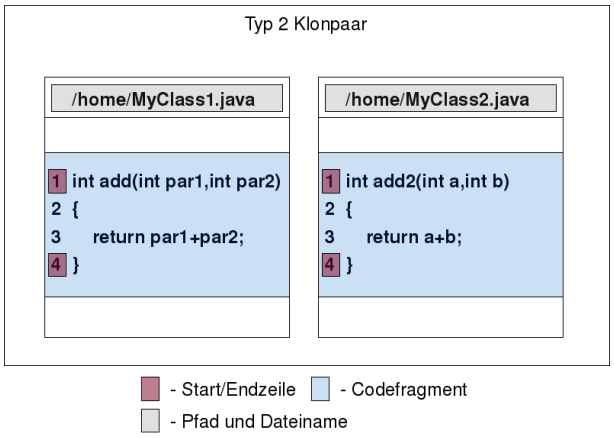

Abbildung 2.1: Zusammenspiel der Begriffe

2.1.3 Klontypen

Die folgende Einteilung der Klone in vier verschiedene Kategorien stammt von Rainer Koschke (in [Koschke, 2007]). Diese Einteilung präzisiert die Definition, da nun auch die Ähnlichkeit der Codefragmente in einem Klon definiert wird. Jede einzelne Kategorie umfasst ausschließlich Klone, die ein bestimmten Ähnlichkeitstyp aufweisen.

Typ1
Ein Klonpaar vom Typ1 weist zwei Codefragmente auf, die absolut gleich zu einander sind. Bei diesen Codefragmenten handelt es sich also um Eins zu Eins Kopien. Die Kopie des Originals unterlag nach dem Einfügen keinen Veränderungen. In der Abbildung 2.2 ist ein Klonpaar von diesem Typ dargestellt.

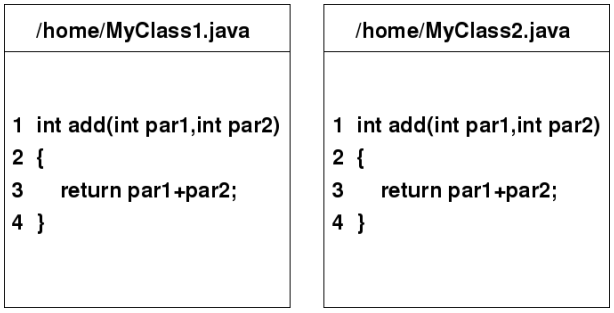

Abbildung 2.2: Beispiel für ein Klonpaar vom Typ1

Typ2
Ein Klonpaar vom Typ2 weist zwei Codefragmente auf, die sich nur in der Namensgebung der Variablen unterscheiden. Bei diesen Codefragmenten handelt es sich um syntaktisch gleiche Kopien, also im Grunde um Typ1 Klone. In der Kopie des Originals wurden jedoch Namen der Variablen konsistent geändert. Dies bedeutet, dass zu jedem Bezeichner im Original ein entsprechender Bezeichner in der Kopie existiert und umgekehrt. Ein solches Klonpaar ist in der Abbildung 2.3 zu sehen.

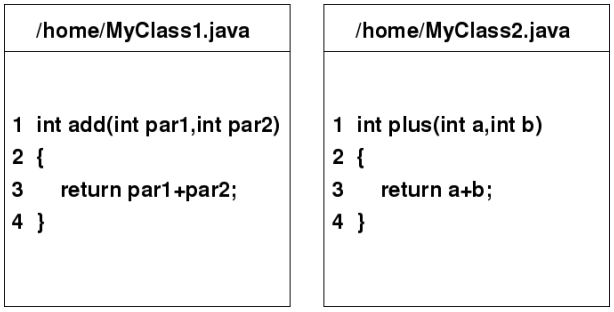

Abbildung 2.3: Beispiel für ein Klonpaar vom Typ2

Typ3
Während die ersten beiden Kategorien eher unscheinbar sind und bis auf die Bezeichner nur syntaktisch gleiche Klonpaare umfassen, ist die Typ3 Kategorie viel interessanter. In diese Kategorie werden Klonpaare eingeordnet, die sich unter Umständen stark voneinander unterscheiden.

Bei den Codefragmenten eines Typ3 Klonpaares handelt es sich um Kopien, deren syntaktische Struktur, also Aufbau des Codefragmente, voneinander abweicht. Die Kopie des Originals wurde nach dem Einfügen modifiziert. Ein Beispiel solcher Modifizierung ist das Einfügen neuer Zeilen um die Funktionalität zu erweitern. Die Abbildung 2.4 zeigt ein Klonpaar vom Typ3.

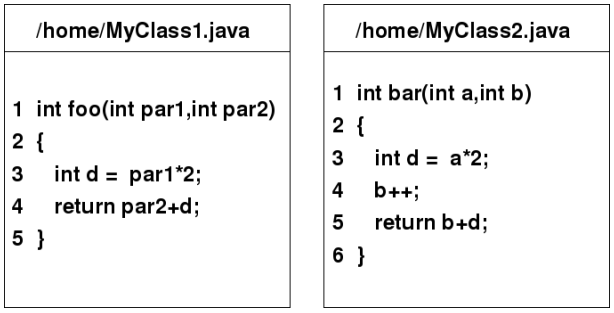

Abbildung 2.4: Beispiel für ein Klonpaar vom Typ3

Die meisten Klonerkennungswerkzeuge mit Ausnahme der metrikbasierten Verfahren führen nach der Identifikation der Klone im Quellcode einen separaten Schritt durch, um die Typ3 Klone zu erkennen. Dabei werden Typ1 oder Typ2 Codefragmente und die dazwischen liegenden Codefragmente zu größeren Klonen zusammengefasst. Die Länge des eingeschlossenen Codefragmentes kann dabei gleich sein oder darf einen vom Benutzer definierten Schwellwert nicht übersteigen. Die Zusammensetzung der Typ3 Klonpaare ist in der Abbildung 2.5 dargestellt.

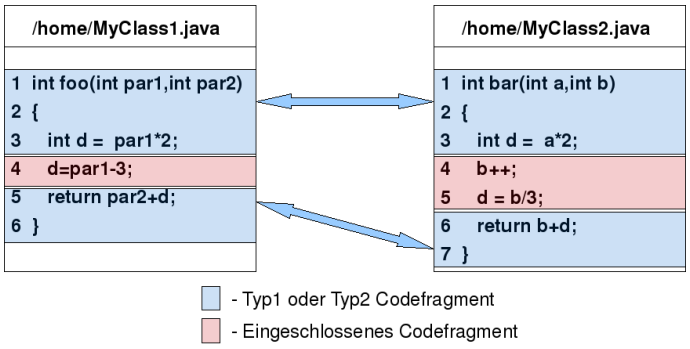

Abbildung 2.5: Zusammensetzung eines Typ3 Klonepaares

Typ4

Ein Klonpaar vom Typ4 weist zwei Codefragmente auf, die sich semantisch nicht voneinander unterscheiden. Es handelt sich hierbei um semantisch äquivalente Codefragmente mit vollständig unterschiedlicher Syntax. Dies bedeutet, dass in den beiden Codefragmenten die selbe Funktionalität auf syntaktisch unterschiedliche Weise realisiert wurde. Ein Beispiel eines Typ4 Klonpaares ist in der Abbildung 2.6 zu sehen.

/home/MyClass1.java	/home/MyClass2.java
1 void foo() 2 { 3 int b = 0; 4 b = b+1 5 }	1 void bar() 2 { 3 int c = 0; 4 c++; 5 }

Abbildung 2.6: Beispiel für ein Klonpaar vom Typ4

2.1.4 Refactoring

Es gibt Stellen im Quelltext, welche die Wartbarkeit eines Softwaresystems beeinträchtigen. Solche Stellen werden „bad smells“ oder einfach „schlechte Gerüche“ genannt. Nach [Fowler, 1999] sind Klone (duplizierter Code) die Nummer Eins in der Rangliste der „bad smells“. Außer dupliziertem Code gibt es vieles mehr, das die Qualität des Quelltextes und somit die Verständlichkeit, Änderbarkeit und Wartbarkeit des Gesamtsystems negativ beeinflusst. Dazu gehören neben Klonen unter Anderem auch lange Methoden, große Klassen und lange Parameterlisten.

Es ist leicht nachvollziehbar, dass solche „schlecht riechenden“ Quelltextstellen nach Möglichkeit geändert werden müssen. Den Prozess der Behebung eines „bad smells“ nennt man Refactoring. Fowler definiert diesen Begriff wie folgt:

> Refactorings sind semantikerhaltende, restrukturierende Code-Transformationen für objektorientierte Programme (zur Verbesserung der Wartbarkeit) [Koschke, 2007]

Neben einer langen Liste von möglichen Quelltextschwachstellen, die Fowler identifiziert, werden zu jedem „bad smell“ auch mindestens ein Refactoringschritt zu Behebung der Folgen vorgeschlagen. Dazu gehören mitunter auch Methodenzusammensetzung, Bewegung der Eigenschaften zwischen Klassen, Organisation von Daten und Vereinfachung bedingter Ausdrücke (mehr in [Koschke, 2007]). Die Auswirkungen der Typ1 oder Typ2 Klone lassen sich relativ einfach beheben, mögliche Optionen dazu wären beispielsweise der „extract method“-Ansatz oder die Extraktion des Klon mittels der Präprozessor-Makros.

Bei Typ3 Klonen ist ein Refactoring viel schwieriger. Bei den meisten Ansätzen wird versucht einen Typ3 Klon auf einen Typ1 oder Typ2 Klon zu reduzieren, um diesen entfernen zu können. Meist müssten zunächst die unterschiedlichen Quelltextbereiche der Klonfragmente in neue Funktionen ausgelagert werden, um diese aus dem Klonkontext heraus aufzurufen. Hierdurch wird der Typ3 Klon wieder auf einen Typ2 oder Typ1 Klon reduziert, welcher dann durch zum Beispiel „extract method“ entfernt werden kann. Bevor die unterschiedlichen Quelltextbereiche der Klonfragmente in neue Funktionen ausgelagert werden können, muss man diese identifizieren.

2.2 Bauhaus

Das „Bauhaus„-Projekt wurde vom Fraunhofer Institut für Experimentelles Software Reengineering und der Universität Stuttgart im Jahr 1996 ins Leben gerufen. Inzwischen wird es als gemeinsames Forschungsprojekt nur an der Universität Bremen von der Arbeitsgruppe Softwaretechnik und von der kommerziellen Ausgründung Axivion GmbH weiterentwickelt. “Bauhaus„ ist eine Ansammlung von Softwarewerkzeugen, die einen Wartungsprogrammierer beim Analysieren, Verstehen und Weiterentwickeln von Software unterstützen soll.

Mit Hilfe von den im “Bauhaus„ integrierten Werkzeugen (siehe Abschnitt 2.2.2) hat der Wartungsprogrammierer die Möglichkeit z.B. verschiedene Metriken über das zu wartende Softwaresystem zu erheben, Klone in dem System aufzuzeigen, die Architektur des Systems zu validieren und vieles mehr. Der Wartungsprogrammierer kann also die potentiellen Problemstellen des Systems identifizieren und wenn möglich beheben.

Um ein Softwaresystem analysieren zu können, muss das System zunächst in eine geeignete Zwischendarstellung gebracht werden. Für “Bauhaus„ wurde eine besondere Zwischendarstellung entwickelt. Diese wird Intermediate Language, kurz IML, genannt. Eine detaillierte Beschreibung der Intermediate Language befindet sich im Abschnitt 2.2.1. In der Abbildung 2.7 wird grob dargestellt, wie die IML generiert und weiterverwendet wird. Von den Quelltextdateien des zu analysierenden Systems ausgehend wird mit Hilfe der „Frontends“ zu Erzeugung von IML eine IML-Darstellung der einzelnen Einheiten generiert. Um eine Darstellung eines gesamten Systems zu erhalten, müssen die einzeln erzeugten IML-Dateien mit dem „IML-Linker“ zu einer systemweiten IML-Datei zusammengefasst werden. Auf der erzeugten Systemiml-Datei können dann schließlich Analysen von „Bauhaus“ durchgeführt werden.

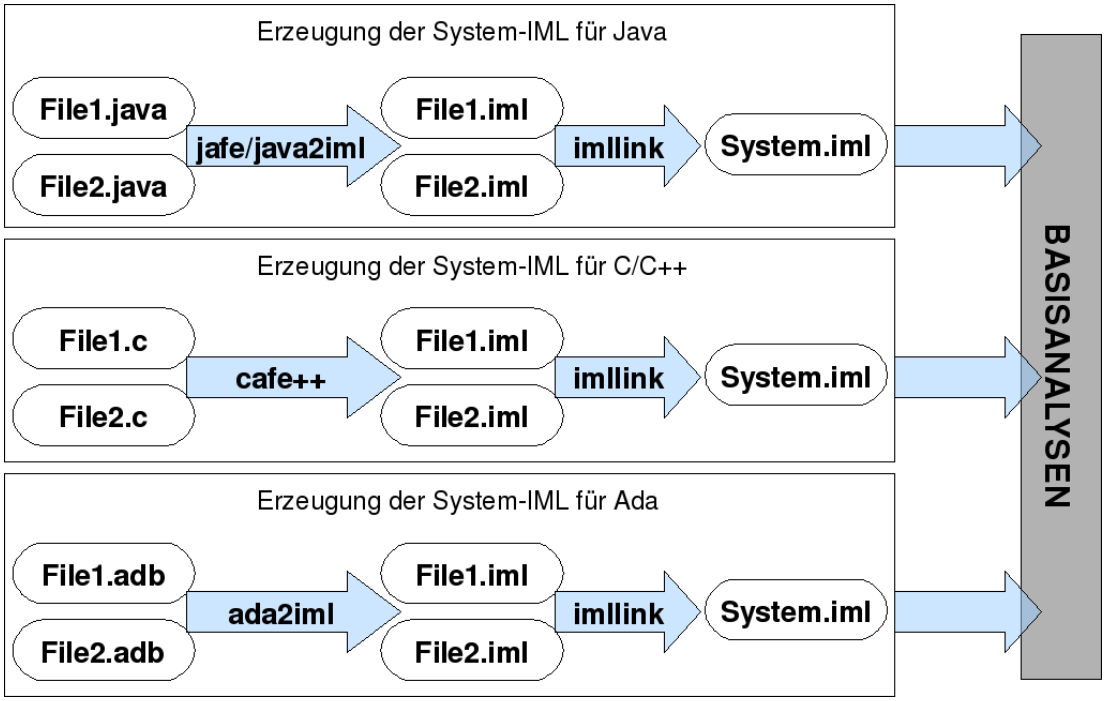

Abbildung 2.7: Generierung einer IML-Datei für ein Softwaresystem

2.2.1 Intermediate Language

Wie bereits beschrieben, ist die IML eine grundlegende Datenstruktur für die in „Bauhaus“ integrierten Analysewerkzeuge. Die IML ist in der Lage Softwaresysteme, die in Programmiersprachen C/C++, Java und Ada implementiert sind, einheitlich darzustellen. Für die Überführung des Quelltextes in die IML-Darstellung sind Werkzeuge wie jafe/java2iml für Java Systeme, cafe++ für C/C++ Systeme und ada2iml für Ada Systeme implementiert worden.

Bei dieser Datenstruktur handelt es sich um einen abstrakten Semantikgraphen, der die logische Erweiterung des abstrakten Syntax Baumes darstellt. Wie in Kapitel 2.3.3.2 geschildert, bildet ein AST nur die Struktur eines Programms ab, während Semantikgraphen erheblich mehr Informationen in sich tragen. Die Kanten eines Semantikgraphen verweisen nicht nur auf die Operanden eines Knotens, sondern auch dessen Typknoten, eventuelle Initialisierungsknoten oder andere Knoten, die weitere semantische Aspekte darstellen.

Ein arithmetischer Operator z.B. besteht aus einem Knoten, der die Operation selbst darstellt wie *Arithmetic_Add*, *Arithmetic_Substract*, *Multiply* oder *Divide* Knoten. Die Kanten *Left_Operand* und *Right_Operand* verweisen auf die Operanden der arithmetischen Operation. Sowohl von den Operations- als auch von den Operandenknoten gehen Kanten aus, die auf einen *Typknoten* verweisen. Dieser bestimmt um was für eine Operation oder um welche Operanden es sich handelt. Ein stark vereinfachtes Beispiel der IML-Darstellung einer Initialisierung einer Variablen der eine Differenz zweier Integer zugewiesen wird, ist in der Abbildung 2.9 des Abschnitts 2.2.2 zu sehen.

Die Knoten der IML sind in Kategorien eingeteilt.

- Die Kategorie *Hierarchical_Unit* stellt ganze Systeme, einzelne Klassen, Methoden, Konstruktoren oder Destruktoren dar.
- Die Kategorie *Value* stellt eine Anweisung beziehungsweise ein Ausdruck dar. Diese Kategorie hat viele Unterkategorien wie *Sequence*, *Operator*, *Loop_Statement* und andere, welche die syntaktische Aspekte einer Programmiersprache abbilden.
- Die Kategorie *Symbol_Nodes* ist in zwei Unterkategorien *T_Node* und *O_Node* aufgeteilt. Die Unterkategorie *T_Node* stellt einzelne Datentypen dar, während die *O_Node* Unterkategorie die Datenwerte repräsentiert. Durch diese Knoten werden Variablen oder Parameter von Methoden dargestellt.

Alle Knoten aus der Kategorie *Value* enthalten über die *Its_Type* Kante ein Verweis auf einen *T_Node*. Ein *Operator* Knoten hat beispielsweise ein Verweis auf einen *T_Node*, der den Typ des Ergebnisses darstellt. Desweiteren besitzt jeder IML-Knoten Attribute wie *Id*, *Sloc*, *Parent* und *Artificial*. Das Attribut *Id* ist eine eindeutige Nummer des Knotens, anhand derer der Knoten in dem IML-Graph identifiziert werden kann. Das Attribut *Sloc* wurde bereits im Abschnitt 2.1.1 eingeführt. Dieser gibt die Position des entsprechenden Quelltextequivalents, der von dem IML-Knoten abgebildet wird. Das Attribut *Artificial* gibt an, ob der IML-Knoten einen Quelltextequivalent besitzt oder nicht.

Die IML Datenstruktur wurde sprachübergreifend konzipiert. Allerdings weisen manche Programmiersprachen auch spezifische Konstrukte, die sich nur schwer verallgemeinern lassen. Alle Konstrukte, die für die jeweilige Programmiersprache spezifisch sind, wurden mit einzelnen IML-Knoten modelliert.

Solche Knoten tragen den Namen der Sprache im Präfix der Knotenbezeichnung. Beispielsweise gibt es in der IML für „Records“, „Arrays“ und „Asserts“ in Ada, extra IML-Knoten *Ada_Record_Create*, *Ada_Array_Create* und *Ada_Assert*. Eine vollständige Übersicht aller IML-Knotenklassen wurde in [Schober, 2007] gegeben.

2.2.2 Werkzeuge

Neben den bereits beschriebenen Frontends zu Erzeugung der IML-Zwischendarstellung existieren in „Bauhaus“ zahlreiche Werkzeuge zu Analyse von Softwaresystemen. Man kann mit Hilfe der Tools zum Beispiel Klone erkennen, Architekturen der Systeme validieren, Zyklen im Aufrufgraph erkennen und verschiedenste Metriken über ein System erheben. Die Anwendung dieser Werkzeuge unterstützt einen Wartungsprogrammierer, da mit Hilfe der Werkzeuge viele verscheidende Problemstellen eines Softwaresystems aufgedeckt werden können. Für diese Diplomarbeit sind jedoch meist nur die Klonerkennungstools von Bedeutung.

In der „Bauhaussuite“ sind insgesamt vier Tools zur Klonerkennung vorhanden, die unterschiedliche Ansätze verfolgen, um Klone zu erkennen. Dabei werden die in [Baker, 1995] und [Baxter, 1998] vorgestellten Verfahren implementiert und zum Teil auch kombiniert, um so die Vorteile beider Ansätze zu verbinden. Berger beschreibt in [Berger, 2007] die einzelnen Klonerkennungstools. Diese Beschreibung wird hier wiederverwendet.

clones
Das Tool clones implementiert den Ansatz von Baker und versucht Klone mit Hilfe eines „Suffixtrees“ auf Basis eines Tokenstroms zu finden. Dieses Tool erkennt Klone in Systemen, die in Programmiersprachen Ada, C, Cobol, C++, C#, Java und Visual Basic implementiert wurden und beherrscht verschiedene Ausgabeformate, in denen die Ergebnisse gespeichert werden können. Das verfahren von Baker wird in clones um einige Möglichkeiten wie z.B. der Ergebnisfilter erweitert, um in der Menge der aufgedeckten Klone uninteressante Klonpaare auszublenden. Dieses Klonerkennungstool hat den Vorteil, dass es den Quelltext der Programmiersprachen C und C++ vor dem Präprozessieren analysiert und damit den Quelltext in seiner Originalversion verwendet. Dies ist im Besonderen bei variantenreicher Software sinnvoll, da Varianten in der Programmiersprache C häufig mit Hilfe des Präprozessors abgebildet werden.

ccdiml
Das Tool ccdiml implementiert den Ansatz von Baxter und versucht die Klone mit Hilfe eines abstrakten Syntax Baumes zu finden. Der AST des Systems wird aus der IML extrahiert, diese dient also als eine Analysegrundlage für ccdiml. Es werden Sprachen wie Ada, C, C++ und Java unterstützt, da zum aktuellen Zeitpunkt nur für diese Sprachen Frontends zu Erzeugung von IML-Darstellung existieren. Im Gegensatz zu clones verarbeitet ccdiml den Programmtext erst nach dem Präprozessieren. Die Teile, die auf Grund des Präprozessors entfernt werden, werden somit nicht berücksichtigt.

cpdetector
Das Tool cpdetector versucht die Ansätze von Baker und Baxter miteinander zu kombinieren. Hierfür wird eine IML geladen, die dann in einen Tokenstrom umgewandelt wird. Das Programm läuft die syntaktischen Kanten ab und generiert für jeden Knoten ein Token. Dieses Token entspricht dem Knotentyp und ist für alle Knoten des gleichen Typs eindeutig. Mit diesem Verfahren sollen die Vorteile der verschiedenen Ansätze kombiniert werden. Das ist zum einen die syntaktische Abgeschlossenheit der Klone und zum anderen die Geschwindigkeit der Klonerkennung.

clast
Das Tool clast verwendet den gleichen Ansatz wie cpdetector nur mit dem Unterschied, dass hierfür als Datenbasis nicht die IML sondern ein eigens hierfür erzeugter AST genutzt wird. Dies hat den Vorteil, dass zum einen mehr Programmiersprachen unterstützt werden können und dass hier die Konstrukte nicht auf einen sprachübergreifenden Kern abgebildet werden, bei dem Detailinformationen über die Syntax verloren gehen. Die Frontends für die verschiedenen Sprachen müssen natürlich erstellt werden, was aber mit den verwendeten Tools lex und yacc ein eher geringer Aufwand im Vergleich zum Erstellen eines IML-Frontends ist. Eine genauere Beschreibung des Verfahrens wird in [Falke u. a. 2008] gegeben.

Visualisierung der IML

cobra
Das Tool „cobra“ ist ähnlich wie Browser aufgebaut und erlaubt das navigieren im Inneren des IML-Graphen. Alle Attribute der IML-Klassen werden berücksichtigt. Attribute die auf andere Knoten verweisen werden als Links dargestellt (blau markierter, klickbarer Text), während andere Attribute als einfacher Text (Name und Wert) dargestellt werden. Man kann auch bestimmte Bereiche des IML-Graphen selektieren und anschauen. Die Menüleiste des IML-Navigators erleichtert das Navigieren. In der Abbildung 2.8 ist die graphische Oberfläche des Tools dargestellt.

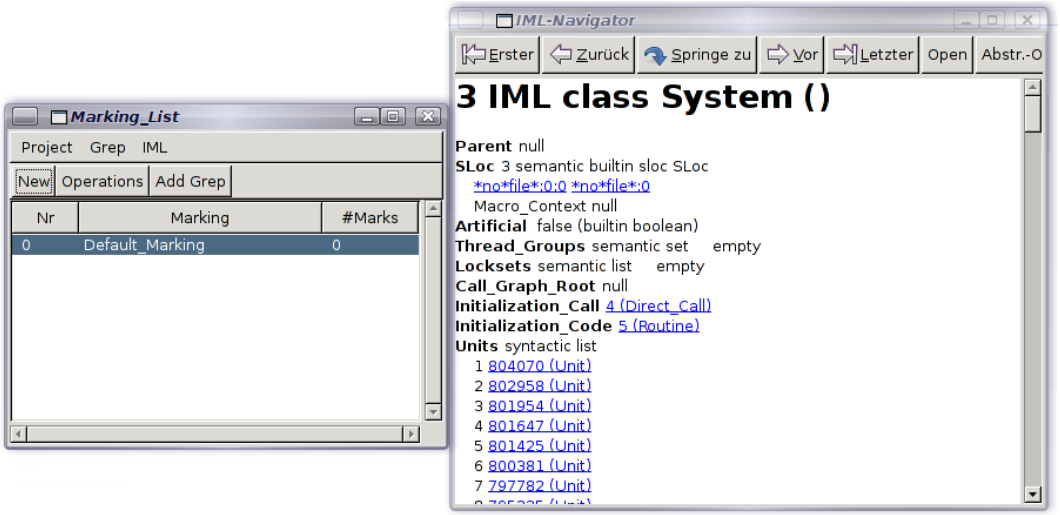

Abbildung 2.8: Graphische Oberfläche von Cobra

iml2dot[1]
Das Tool „iml2dot“ transformiert einen IML-Graph in eine .dot Repräsentation. Dieses Format kann von Linux Tools dot und dotty gelesen werden. Iml2dot benutzt IML-Reflektion um einen Graph im .dot Format zu generieren. Alle Attribute der IML-Klassen werden berücksichtigt. Attribute die auf andere Knoten verweisen, werden als Kanten dargestellt, während andere Attribute als Text innerhalb der Knoten dargestellt werden. Typknoten werden gelb dargestellt, Typkanten sind gestrichelt. Syntaktische Kanten werden fettgedruckt gezeichnet und haben höhere Priorität, deshalb bildet der abstrakte Syntax Baum die Basis des Graphen. Einen Ausschnitt einer .dot Darstellung eines IML-Graphen kann man in der Abbildung 2.9 sehen.

[1] Quelle: https://cube.tz.axivion.com/dokuwiki/doku.php?id=bauhaus:iml2dot2

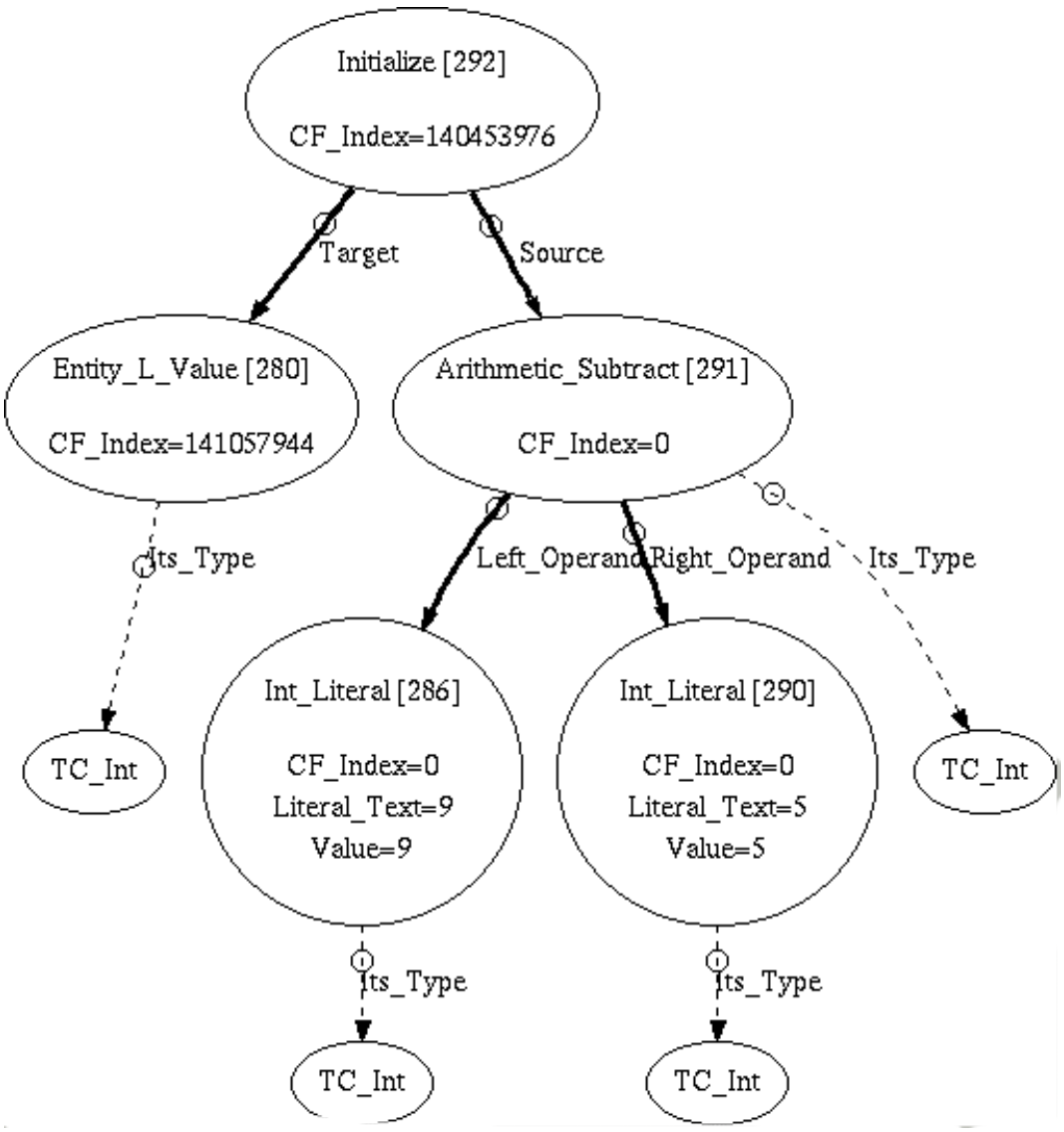

Abbildung 2.9: Beispiel eines IML-Graphen im .dot Format

2.3 Datenstrukturen

In den nachfolgenden Abschnitten werden verschiedene Datenstrukturen verwendet. Es handelt sich hierbei um Listen, Prioritätswarteschlangen, Graphen, Bäume und abstrakte Syntax-Bäume. In diesem Abschnitt des Kapitels werden diese Datenstrukturen definiert. Im Folgenden geht es nicht darum, diese Begriffe vollständig zu erläutern, sondern die Notation der Konzepte, die in dieser Arbeit ihre Verwendung finden, einzuführen. Es wird davon ausgegangen, dass der Leser mit Graphentheorie und Datenstrukturen vertraut ist.

2.3.1 Listen und Prioritätswarteschlangen

Listen und Prioritätswarteschlangen gehören wie z.B. Arrays zu trivialen, grundlegenden Datenstrukturen. Deshalb wird hier nur eine kurze Beschreibung dieser Datenstruktur gegeben.

Bei einer *Liste* handelt es sich um eine Menge von Knoten, die anwendungsspezifische Informationen in sich tragen. Es gibt einfach und doppelt verkettete Listen. Bei einfach verketteten Listen beinhaltet jeder Knoten, bis auf den letzten, zusätzlich einen Zeiger auf seinen Nachfolger. Bei doppelt verketteten Listen hat jeder Knoten neben dem Zeiger auf seinen Nachfolger auch einen Zeiger auf seinen Vorgänger. Eine Ausnahme bilden der erste Knoten, der keinen Vorgänger hat und der letzte Knoten, der keinen Nachfolger aufweist.
Haben Knoten Attribute, die eine Rangordnung zwischen den Knoten ermöglichen, so können Listen sortiert werden. Eine *Prioritätswarteschlange* ist an sich eine sortierte Liste, in der die Knoten nach ihrer Priorität sortiert sind. Das Einfügen der Knoten in die Warteschlange geschieht gemäß ihrer Priorität/Ordnung untereinander. Bei der Entnahme eines Knotens zur Weiterverarbeitung wird immer ein Knoten mit größter Priorität geliefert.

2.3.2 Graphen

Graphen sind Datenstrukturen, mit denen sich unterschiedliche Sachverhalte modellieren und visualisieren lassen. Alle Definitionen der Graphen beinhalten Knoten und Kanten. Knoten sind eindeutig unterscheidbare Objekte und beinhalten Informationen. Kanten verbinden Knoten und drücken Beziehungen zwischen diesen aus.

2.3.2.1 Definitionen

Definition 2.1 (Gerichteter Graph) *Ein gerichteter Graph $G = \{V, E\}$ besteht aus einer endlichen Menge V von Knoten und einer Menge $E \subseteq V \times V$ von Kanten. Der Grad des Graphen $n = |V|$, wird mit der Anzahl der Knoten in der Menge V angegeben. Die Größe des Graphen $m = |E|$ wird mit der Anzahl der Kanten in der Menge E angegeben. Knoten v und w einer Kante $e = (v, w)$ heißen inzident mit der Kante e, wobei v als Quelle und w als Ziel der Kante bezeichnet werden. Existiert eine Kante $e = (v, w)$, welche die Knoten v und w verbindet, so heißen die Knoten adjazent.*

Definition 2.2 (Grad eines Knoten in einem gerichteten Graph) *Jeder Knoten des gerichteten Graphen hat einen „indegree“ und einen „outdegree“.*

- *Unter dem Begriff „indegree“ eines Knotens v, $indegree(v) = |\{e \in E \mid e = (x, v)\}|$, versteht man die Anzahl der Kanten, die den Knoten v als Ziel haben.*

- *Der Begriff „outdegree“ eines Knotens v, $outdegree(v) = |\{e \in E \mid e = (v, x)\}|$, gibt die Anzahl der Kanten an, die den Knoten v als Quelle haben.*

- *Der Grad eines Knotens in einem gerichteten Graph besteht aus der Summe von „indegree“ und „outdegree“, $degree(v) = indegree(v) + outdegree(v)$.*

Definition 2.3 (Ungerichteter Graph) *Ein Graph $G = \{V, E\}$ ist ungerichtet wenn gilt: $(v, w) \in E \Rightarrow (w, v) \in E \ \forall v, w \in V$. Die mit einer Kante inzidenten Knoten in einem ungerichteten Graph heißen Endknoten. Solche Knoten sind in der durch die Kante dargestellten Beziehung gleichberechtigt. Der Grad eines Knotens v in einem ungerichteten Graph $degree(v) = |\{e \in E \mid e = (v, x) \vee e = (x, v)\}|$ gibt die Anzahl der Kanten an, mit denen v inzident ist.*

Die meisten Anwendungen in der Informatik verwenden gewichtete/markierte Graphen. Bei gewichteten Graphen wird jeder Kante ein Gewicht zugeordnet.

Die markierten Graphen haben meist eine Markierung an den Knoten. Auch verschiedene Kombinationen der Konzepte sind möglich. Ein Graph kann also an Knoten und an Kanten neben dem Gewicht auch eine Markierung sog. „Label“ haben.

Definition 2.4 (Gewichte und Markierungen) *Bei den Gewichten handelt es sich meist um Zahlen. Eine Funktion weight* $: E \rightarrow \mathbb{R}$*, die jeder Kante* $e \in E$ *eine Zahl zuordnet, nennt man Gewichtungsfunktion. Markierungen können auch andere Datentypen wie Zeichenketten involvieren.*

Definition 2.5 (Pfad) *Ein Pfad von einem Knoten* v_i *zu einem Knoten* v_j *ist eine alternierende Sequenz* $(v_i, e_{i+1}, v_{i+1}, e_{i+2}, ..., v_{j-1}, e_j, v_j)$ *von Knoten und Kanten, sodass* $e_k = (v_{k-1}, v_k)$ *für* $k \in \{i+1, ..., j\}$*, in der kein Knoten und keine Kante wiederholt vorkommt.*

•*Die Anzahl der Kanten in einem Pfad p nennt man* ***Länge*** *des Pfads; es gilt* $length(p) = l \mid l \in \mathbb{N} \ \wedge l > 0$. Man merkt, dass in der Definition des Pfads die Knoten in der alternierenden Sequenz redundant sind. Deshalb werden im folgenden Verlauf der Arbeit die Knoten eines Pfads ausgelassen.

•*Ein Pfad ist somit eine Abfolge von Kanten, in der keine Kante wiederholt vorkommt.*

Definition 2.6 (Distanz) *Sei* $p = ((v_1, v_2), (v_2, v_3), ..., (v_{n-1}, v_n))$ *ein Pfad vom Knoten* $v_1 \in V$ *zum Knoten* $v_n \in V$ *in einem gewichteten Graph, dann wird die Summe der Gewichte der Kanten im Pfad* $dist(p) = \sum_{i=1}^{n-1} weight((v_i, v_{i+1}))$ *als* ***Distanz*** *bezeichnet.*

2.3.2.2 Darstellung

Für die Darstellung der Graphen wird meist eine graphische Repräsentation statt der mengentheoretischen Definition gewählt. Bei dieser werden die Knoten als dicke Punkte, Kreise, Rechtecke oder Ähnliches dargestellt und die Kanten werden als Linien oder Pfeile zwischen zwei Knoten gezeichnet. In der Abbildung 2.10 sind einige Beispiele für unterschiedliche Graphen dargestellt.

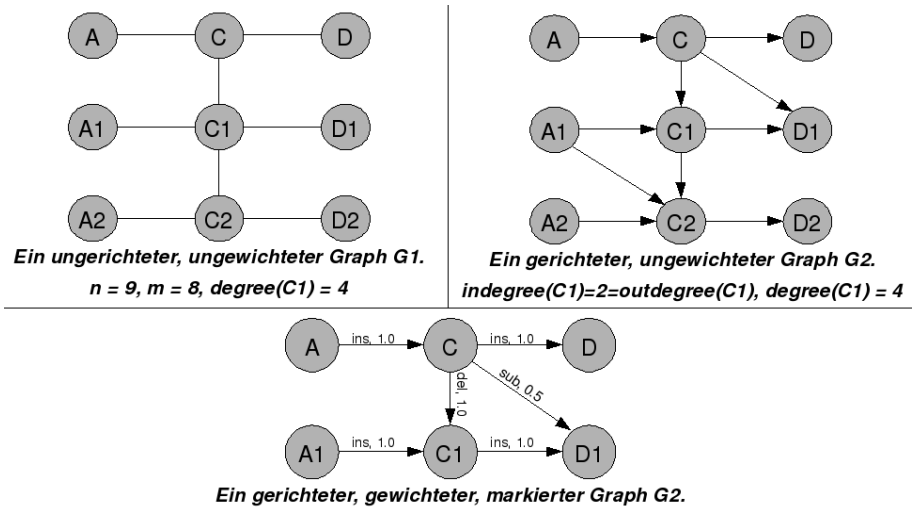

Abbildung 2.10: Beispiele für Graphen

2.3.2.3 Dijkstra's kürzeste Wege

Eine der wichtigsten Problemstellungen für gewichtete Graphen ist das Finden „kürzester Wege/Pfade“ in einem gerichteten gewichteten Graph. Ein kürzester Weg/Pfad wird wie folgt definiert:

Definition 2.7 (kürzester Weg/Pfad) *Sei $G = \{V, E\}$ ein gerichteter Graph mit gewichteten Kanten, weiterhin sei P die Menge aller Pfade vom Knoten $v \in V$ zu dem Knoten $w \in V$. Ein Pfad p wird „kürzester Pfad“ genannt, wenn gilt: $dist(p) \leq dist(p')\ \forall p' \in P$. Es gibt also keine „Alternativstrecke“ $p' \in P$ zu p mit geringeren Kosten.*

Kürzeste Wege sind nicht eindeutig. So kann es zwischen zwei Knoten eines Graphen mehrere Pfade mit gleicher Distanz geben. Abhängig von der Topologie des Graphen kann es auch vorkommen, dass keine Wege zwischen bestimmten Knoten existieren.

Dijkstra's Algorithmus zum Finden kürzester Wege wurde im Jahr 1959 veröffentlicht und nach seinem Erfinder Edsger Wybe Dijkstra benannt. Dieser Algorithmus ist einer der bekanntesten Graphenalgorithmen und findet seinen Platz in zahlreichen Anwendungen. Er basiert auf einer iterativen Erweiterung einer Menge von „billig“ erreichbaren Knoten und kann daher als ein auf dem Greedy-Prinzip basierender Algorithmus aufgefasst werden. Allerdings funktioniert dieser Algorithmus nur für Graphen mit nichtnegativen Kantengewichten. Die Pseudocode-Notation 1 beschreibt die Funktionsweise des Algorithmus.

Pro Knoten wird im Attribut „dist“ ein Wert abgespeichert, der für den Startknoten den Wert 0 Enthält und nach Ablauf des Verfahrens den korrekten Distanzwert zum Startknoten enthalten soll. Während der Berechnung enthält dieses Attribut Zwischenwerte. So ist die Distanz am Anfang unendlich. Der Algorithmus berechnet also die Distanz aller Knoten zum Startknoten. Mit der Angabe des Endknotens kann man anschließend einen kürzesten Pfad extrahieren.
Eine ausführliche Beschreibung mit Anwendung des Verfahrens an einem Beispielgraphen und der Beweis der Optimalität des Algorithmus kann man in [Algorithmen und Datenstrukturen] finden.

Algorithm 1 Pseudocode-Notation für Dijkstra's kürzeste Wege Algorithmus.

```
function DIJKSTRA(G, s)                ▷ Eingabe: Eingabe: Graph G mit Startknoten s.
    for all Knoten u ∈ V − s do
        u.dist := ∞;
    end for
    s.dist := 0;
    PriorityQueue Q := V;
    while ¬ isEmpty(Q) do
        u := extractMinimal(Q);
        for all v ∈ ZielKnotenAusgehenderKanten(u) ∩ Q do
            if u.dist + weight((u, v)) < v.dist then
                v.dist := u.dist+weight((u,v));
                adjustiere Q an neuen Wert v.dist;
            end if
        end for
    end while
end function
```

2.3.3 Bäume

Eine weitere für diese Arbeit wichtige Datenstruktur ist ein Baum. In der Informatik finden Bäume zahlreiche Anwendungsformen, z.B. wird die Anordnung von Dateien im Dateisystem eines Computers häufig in Form eines Baumes dargestellt. Die syntaktische Dekomposition eines Programms kann als ein *„abstrakter Syntax Baum“*, kurz AST, dargestellt werden. Abstrakte Syntax Bäume werden in Kapitel 2.3.3.2 behandelt.

2.3.3.1 Definitionen

Definition 2.8 (Baum) *Ein gerichteter Graph $G = \{V, E\}$ heißt Baum, wenn der Graph keine zyklischen Pfade aufweist und einen ausgezeichneten Knoten r besitzt. Diesen Knoten nennt man Wurzel des Baumes.*

Definition 2.9 (Tiefe) *Für alle Knoten $v \in V$ muss ein eindeutiger Pfad p von der Wurzel r bis zu dem Knoten v existieren. Die Länge diesen eindeutigen Pfades gibt die Tiefe des Knotens im Baum an; es gilt: $depth(v) = length(p) \mid p$ ist eindeutiger $Pfad$ von r zu v.*

Definition 2.10 (Elternknoten, Kind und Geschwisterknoten) *Existiert eine Kante $e = (v, w) \mid e \in E$, so heißt v Elternknoten von w, $parent(w) = v$ und analog dazu heißt w Kindknoten von v. Die Menge $C \subseteq V \mid \forall\, c \in C\ \exists\, e = (v, c) \in E$ stellt die Menge aller Kinder des Knotens v dar. Der Wurzelknoten hat kein Elternknoten. Knoten, die keine Kinder aufweisen, nennt man **Blätter**. Zwei Knoten v und w heißen **Geschwisterknoten** falls gilt: $parent(v) = parent(w)$.*

Definition 2.11 (Geordnete Bäume) *Ein geordneter Baum ist ein Baum, in dem die relative Reihenfolge der Kinder für jeden „Nichtblattknoten“ fest ist. Spielt die Reihenfolge der Kinder eines Knotens keine Rolle, so heißt ein solcher Baum ungeordnet.*

Ein Beispiel für ein Baum und die im Vorfeld eingeführten Begriffe ist in der Abbildung 2.11 dargestellt. Die Knoten des Baumes sind nach deren „preorder“ Reihenfolge durchnummeriert, es handelt sich somit um einen geordneten Baum. Auf die „preorder“ Traversierung eines Baumes wird im Abschnitt 2.3.3.3 eingegangen.

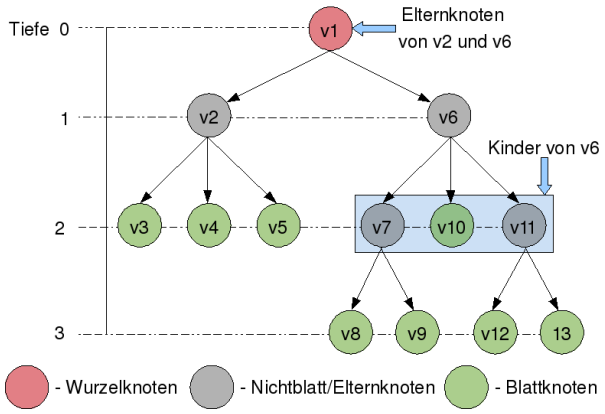

Abbildung 2.11: Beispiel für ein Baum

2.3.3.2 Abstrakte Syntax Bäume

Wie bereits erwähnt, handelt es sich bei einem abstrakten Syntax Baum (*Abstract Syntax Tree*/**AST**) um einen Baum, der die syntaktische Representation eines Quelltextes wiederspiegelt. Bei dem Kompilieren eines Programmtextes wird unter Anderem ein AST erzeugt. Dafür wird der Quelltext zunächst mit Hilfe eines Lexers in ein Tokenstrom überführt. Der Parser baut aus dem Tokenstrom einen Parsebaum auf, der später zu einem abstrakten Syntax Baum wird. Für diese Prozesse muss eine Grammatik vorliegen, welche die Strukturen der Programmiersprache definiert. Der abstrakte Syntax Baum bildet dann eine Grundlage für weitere Schritte in Richtung Darstellung des Quelltextes in einer Maschinensprache. Die Abbildung 2.12 zeigt ein Quelltextfragment, der die Zuweisung einer Differenz an eine Variable beschreibt, sowie seine Darstellung in Form eines ASTs.

Jeder abstrakte Syntax Baum ist geordnet, da die Reihenfolge der Kinder eines Knoten eine große Rolle spielt. Würde man im Beispiel aus der Abbildung 2.12 die Kinder der Subtraktion tauschen, hätte dies gravierende Folgen für das Endergebnis.

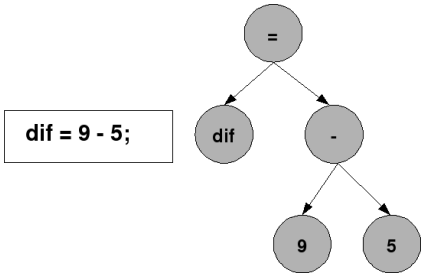

Abbildung 2.12: Beispiel für einen AST

2.3.3.3 Preorder Traversierung

Um einen Baum auslesen zu können, muss man systematisch alle Knoten des Baumes abarbeiten. Dieses Abarbeiten der Knoten wird Traversierung genannt (vom engl. traversal). Hierfür existieren einige Ansätze wie z.B preorder, postorder und andere. In diesem Abschnitt wird die preorder Traversierung eines Baumes erläutert.
Bei der preorder Traversierung eines Baumes wird der Wurzelknoten zuerst abgearbeitet, danach folgt die Abarbeitung der Unterbäume, welche die Kinder des Wurzelknotens als Wurzel haben. Diese Unterbäume werden von links nach rechts rekursiv abgearbeitet. In der Abbildung 2.11 sind die Knoten des Baumes in preorder Reihenfolge durchnummeriert. Eine Pseudocode-Notation 2 beschreibt die Funktionsweise des Algorithmus.

Algorithm 2 Pseudocode-Notation für preorder Traversierung von Bäumen

```
function Preorder_Traversal_Rec(v)                    ▷ Eingabe: Knoten v
    verabeiteKnoten(v);

    for all Knoten u ∈ children(v)left to right do
        Preorder_Traversal_Rec(u);
    end for
end function
```

Durch preorder Traversierung eines Baumes kann man diesen durch z.B Abspeichern der Knoten in einer Liste linearisieren. Eine Linearisierung des ASTs aus Abbildung 2.12 durch preorder Traversierung würde folgendes liefern: (=, dif, - , 9, 5)

KAPITEL 3

Lösungsansatz

Inhalt

In diesem Abschnitt der Arbeit wird der Lösungsansatz erläutert. Zunächst wird die grundlegende Idee zu der Lösung der Problemstellung beschrieben. Im zweiten Abschnitt wird ein Überblick über die verwandten Arbeiten gegeben, damit der Leser einschätzen kann, wie sich das Problem in das Forschungsfeld einordnet. Im letzten Abschnitt wird schließlich ein geeigneter Lösungsansatz und seine Erweiterungen vorgestellt und ausführlich beschrieben.

3.1 Idee

Softwareklone entstehen durch Kopieren und Einfügen eines Codefragements. Nach dem Einfügen können an der Kopie Änderungen vorgenommen werden. Eine Änderung nennt man Operation. Dabei unterscheidet man zwischen den grundsätzlichen Operationen: Löschen einer Struktur, Hinzufügen einer Struktur oder Ersetzen einer Struktur bzw. eines Bezeichners durch eine Andere. Die Operation des Ersetzens bzw. des Umbenennens der Bezeichner nennt man auch Substitution. Es sind auch andere komplexere Operationen, wie z.B. Verschieben einer Struktur an eine andere Position oder Permutation der Strukturen, denkbar. Allerdings lassen sich die komplexen Operationen in mehrere grundlegende Operationen zerlegen. So handelt es sich beim Verschieben einer Struktur um nichts anderes als Löschen der Struktur von ihrer ursprünglichen Position und dem Einfügen an einer neuen Position.

Betrachtet man das Beispiel 2.4 aus dem Abschnitt 2.1.3 so sieht man, dass in der Kopie neben der Umbenennung einiger Bezeichner auch eine Codezeile hinzugefügt wurde. Die Kopie wurde also nach dem Einfügen durch eine Abfolge an Operationen editiert. Die Änderungen bzw. Operationen werden im Folgenden als **Editieroperationen** bezeichnet. Eine Abfolge von Editieroperationen wird **Editiersequenz** genannt.

Um die Unterschiede zwischen den Codefragmenten eines Typ3 Klonpaares aufzuzeigen, kann man eine Editiersequenz berechnen.

Die Sequenz beschreibt genau welche Änderungen an der Kopie vorgenommen wurden und somit auch wie die Codefragmente sich unterscheiden. Die Anzahl der Editieroperationen in der Sequenz kann als Ähnlichkeitsmaß zwischen den Codefragmenten verwendet werden. Je weniger Änderungen vorgenommen wurden, desto ähnlicher sind sich die Codefragmente. Die Anzahl der Editieroperationen in einer Editiersequenz wird im Folgenden **Editierdistanz** genannt (siehe Definition 3.6 und 3.7). Eine Editierdistanz wird somit aus der Editiersequenz berechnet.

Stellt man die Codefragmente eines Klonpaares als abstrakte Syntax Bäume dar, so kann man das Problem auf die Berechnung der Editiersequenz bzw. der Editierdistanz zwischen zwei geordneten Bäumen abbilden. In der Abbildung 3.1 sind abstrakte Syntax Bäume der Codefragmente eines Typ3 Klonpaares vereinfacht dargestellt. Es handelt sich um drei Zuweisungen in dem ersten Codefragment und zwei Zuweisungen in dem zweiten. Man sieht, dass die zweite Zuweisung in der Kopie nicht vorhanden ist und gelöscht werden müsste, um die Codefragmente anzugleichen. Dies spiegelt sich auch in den abstrakten Syntax Bäumen wieder. Die beiden ASTs gleichen sich weitgehend bis auf den mittleren Teilbaum im T_1, der im Baum T_2 nicht vorhanden ist. Man muss also eine Editiersequenz berechnen, die das Löschen des mittleren Teilbaums vorschlägt, um T_1 in T_2 zu transformieren. Durch eine geeignete Interpretation und Ausgabe der Editiersequenz können dann die Unterschiede der Codefragmente eines Klonpaares deutlich gemacht werden.

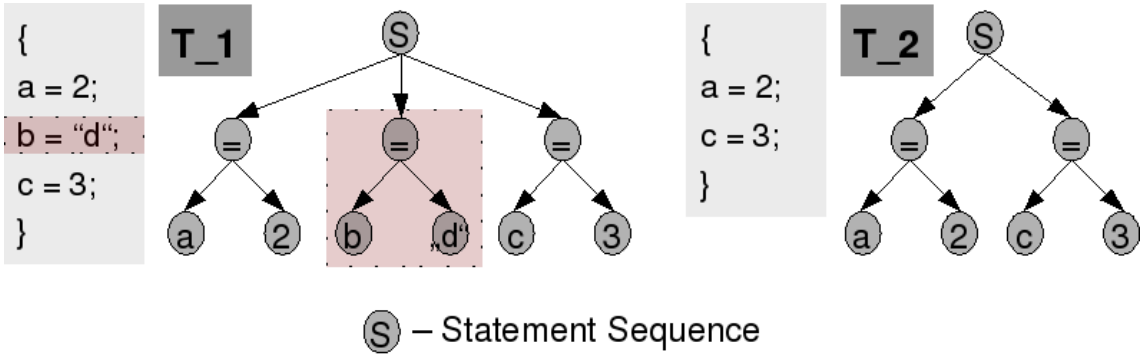

Abbildung 3.1: Vereinfachte ASTs der Codefragmente eines Typ3 Klonpaares

Ein geeigneter Ansatz zur Berechnung einer Editiersequenz, die einen Baum in einen anderen transformiert, wird im Abschnitt 3.3 beschrieben.

3.2 Verwandte Arbeiten

Wladimir Levenstein führte in [Levenstein, 1966] einen Algorithmus zum Berechnen einer Editierdistanz zwischen zwei Zeichenketten ein. Als Editierdistanz bezeichnet er die minimale Anzahl an Operationen in einer Editiersequenz, die eine Zeichenkette in eine andere überführt. Als Operationen gelten Einfügen, Löschen oder Ersetzung von Zeichen.

Es existiert eine Vielzahl an Algorithmen zu Berechnung der Editierdistanz zwischen geordneten Bäumen. Im Jahr 1977 veröffentlichte Selkow in [Selkow, 1977] den ersten Ansatz zur Lösung der Problemstellung. Dieser Ansatz bildet eine Grundlage für die von Tai in [Tai, 1979] präsentierten Ansätze und später auch für die von Zhang und Shasha entwickelten Algorithmen (vorgestellt in [Zhang und Shasha, 1989],[Zhang und Shasha, 1990] und [Zhang, 1995]).

Die Ansätze von Tai, Zhang und Shasha machen keine Einschränkungen bei der Definition die elementaren Grundoperationen. Somit ist bei diesen Ansätzen möglich einen „Nichtblattknoten“ zu löschen oder einzufügen. Dabei werden beim Löschen eines solchen Knoten seine Kinder an seinen Elternknoten angehängt. Beim Einfügen wird ein Teil der Kinder des neuen Elternknoten zu Kindern des eingefügten Knoten. Bezogen auf Operationen auf abstrakten Syntax Bäumen sind solche Operationen nicht zulässig, denn diese würden bei Anwendung sinnlose falsche abstrakte Syntax Bäume erzeugen. In dem Ansatz von Selkow ist das Löschen und Einfügen von Knoten auf Blattknoten eingeschränkt. Die Editiersequenz/Editierdistanz mit solchen eingeschränkten Editieroperationen wird auch $1-degree$ Editiersequenz genannt. In [Yang 1991] wird eine Vorgehensweise vorgestellt, wie die Unterschiede zweier Programme auf syntaktischer Ebene berechnet und dargestellt werden können. Dabei werden „parse-trees“ mit einem „tree-matching“ Algorithmus verglichen.

Einige effiziente Algorithmen zum Vergleichen von Strings verwenden einen Editiergraph-Ansatz, um die Unterschiede zu berechnen. Diese wurden in [Myers, 1986] und [Wu u.a. 1990] vorgestellt. In [Chawathe, 1999] wird dieser Ansatz zum ersten mal verwendet, um eine günstigste Editiersequenz (in der Arbeit als Editierskript bezeichnet) zwischen geordneten markierten Bäumen zu berechnen. Ziel und Kontext der Arbeit von Chawathe ist, einen Ressourcen schonenden Algorithmus zu entwickeln, der die Unterschiede zwischen hierarchisch strukturierten Daten (wie z.B Quellcode) berechnet. Dieser Ansatz wird in dieser Arbeit aufgegriffen und verwendet.

3.3 Editiergraph Ansatz

Der Algorithmus zum Berechnen einer Editiersequenz bzw. Editierdistanz beantwortet die Frage, welche bzw. wieviele Editieroperationen (Hinzufügen, Löschen und Ersetzen von Knoten) notwendig sind, um einen Baum in einen anderen zu transformieren. In [Valiente, 2002] wird ein Ansatz zum Berechnen einer Editiersequenz für geordnete Bäume mit Hilfe eines Editiergraphen vorgestellt. Die nachfolgenden Abschnitte beschreiben diesen Ansatz und die in dieser Arbeit entwickelten Erweiterungen. Im Abschnitt 3.3.1 werden die notwendigen Begriffe definiert und das Verfahren ausführlich erläutert. Der Abschnitt 3.3.2 enthält einen Beispiel zur Veranschaulichung der Vorgehensweise. Im Abschnitt 3.3.4 und 3.3.5 werden letztendlich die Erweiterungen beschrieben.

3.3.1 Definitionen

Definition 3.1 (elementare Editieroperation) *Seien $T_1 = (V_1, E_1)$ und $T_2 = (V_2, E_2)$ geordnete Bäume. Als elementare Editieroperationen an Bäumen T_1 und T_2 gelten die Folgenden:*

- ***„Löschen“ (deletion)*** *<u>eines Blattknoten</u> $v \in V_1$ aus dem Baum T_1, im Folgenden notiert als $v \rightarrow \lambda$ oder (v, λ).*

- ***„Hinzufügen“ (insertion)*** *eines Knoten $w \in V_2$ aus dem Baum T_2, <u>als ein Blattknoten</u> in den Baum T_1, im Folgenden notiert durch $\lambda \rightarrow w$ oder (λ, w)*

- ***„Ersetzung“ (substitution)*** *eines Knotens $v \in V_1$ in dem Baum T_1 durch ein Knoten $w \in V_2$ aus dem Baum T_2, im Folgenden notiert durch $v \rightarrow w$ oder (v, w).*

Das Löschen und Hinzufügen ist somit nur auf Blattknoten (siehe Abschnitt 2.3.3) beschränkt. Um ein Elternknoten löschen zu können, müssen also zunächst alle seine Kindknoten gelöscht werden. Ebenso kann ein Elternknoten nur mit seinem gesamten Unterbaum eingefügt werden.

Definition 3.2 (Transformation) *Eine Transformation eines Baumes T_1 in den Baum T_2 ist eine Editiersequenz $S \subseteq (V_1 \cup \{\lambda\}) \times (V_2 \cup \{\lambda\})$ von elementaren Editieroperationen. Durch das Anwenden der Operationen nach der Reihenfolge, in der sie in der Sequenz vorkommen, wird der Baum T_1 in den Baum T_2 überführt.*

Nicht jede Sequenz von elementaren Editieroperationen stellt eine valide Transformation zwischen zwei geordneten Bäumen dar. Einerseits müssen die Lösch- und Einfügeoperationen von unten nach oben erfolgen, um sicherzustellen, dass tatsächlich nur Blattknoten entfernt oder hinzugefügt werden. Andererseits muss bei einer validen Transformation die Reihenfolge der Eltern und Geschwisterknoten eingehalten werden, um sicherzustellen, dass das Ergebnis der Transformation tatsächlich ein geordneter Baum ist. In einer validen Transformation eines geordneten Baumes T_1 in einen geordneten Baum T_2 muss der Elternknoten eines Knoten aus T_1, der durch einen Knoten aus T_2 substituiert wird, durch den Elternknoten aus T_2 substituiert werden. Weiterhin muss die relative Reihenfolge der Geschwisterknoten bei der Substitution eingehalten werden. Die zweite Anforderung nennt sich „Mapping“ und wird wie folgt definiert:

Definition 3.3 (Mapping) *Seien $T_1 = (V_1, E_1)$ und $T_2 = (V_2, E_2)$ geordnete Bäume, $r_1 \in V_1$ und $r_2 \in V_2$ Wurzelknoten der Bäume T_1 bzw. T_2, $W_1 \subseteq V_1$ und $W_2 \subseteq V_2$. Ein Mapping M von T_1 zu T_2 ist eine Zuordnung $M \subseteq W_1 \times W_2$, sodass gilt:*

- $(r_1, r_2) \in M$ *wenn* $M \neq \emptyset$
- $(v, w) \in M$ *nur wenn* $(parent(v), parent(w)) \in M \ \forall v \in W_1 \wedge \ w \in W_2$
- $(v_1, w_1), (v_2, w_2) \in M \ \forall v_1, v_2 \in W_1 \wedge w_1, w_2 \in W_2$ *gdw.* v_1 *ein Geschwisterknoten links von* v_2 *und* w_1 *ein Geschwisterknoten links von* w_2 *sind*

Für alle Zuordnungen $(v, w) \in M \mid v \in V_1 \wedge w \in V_2$ gilt $depth(v) = depth(w)$. Eine Transformation ist somit dann valide, wenn Einfüge- und Entfernoperationen nur auf Blattknoten erfolgen und die Substitutionsoperationen ein Mapping darstellen.

Definition 3.4 (valide Transformation) *Eine Sequenz von elementaren Editieroperationen wird als valide Transformation $E \subseteq (V_1 \cup \{\lambda\}) \times (V_2 \cup \{\lambda\})$ bezeichnet, wenn gilt:*

- (v_j, λ) *tritt in E vor* (v_i, λ) *auf* $\forall (v_j, \lambda), (v_i, \lambda) \in E \cap V_1 \times \{\lambda\}$, *sodass Knoten* v_j *unterhalb des Knoten* v_i *in* T_1 *liegt*
- (λ, w_i) *tritt in E vor* (λ, w_j) *auf* $\forall (\lambda, w_i), (\lambda, w_j) \in E \cap \{\lambda\} \times V_2$, *sodass Knoten* w_j *unterhalb des Knoten* w_i *in* T_2 *liegt*
- $E \cap V_1 \times V_2$ *ist ein Mapping von* T_1 *zu* T_2

Für jedes Paar geordnete Bäume existiert immer eine Transformation, die den ersten in den zweiten überführt. Die einfachste Transformation wäre, alle Knoten des ersten Baumes zu löschen und die Knoten des zweiten Baumes einzufügen.

Es scheint, dass die Substitution als elementare Operation nicht notwendig ist, um einen Baum in einen anderen zu transformieren. Dennoch ist die Substitution der Knoten sehr wichtig, denn sie ist notwendig, um die kürzeste oder allgemein die günstigste Transformation zu finden. Um eine solche günstige Transformation zu berechnen, ist es notwendig die einzelnen Editieroperationen zu gewichten.

Definition 3.5 (Kosten einer Editieroperation) *Die Kosten einer Editieroperation werden durch eine Funktion cost* $: V_1 \cup V_2 \cup \{\lambda\} \times V_1 \cup V_2 \cup \{\lambda\} \rightarrow \mathbb{R}$ *beschrieben. Für alle* $v, w, z \in V_1 \cup V_2 \cup \{\lambda\}$ *gilt:*

•$cost(v, w) \geq 0$

•$cost(v, w) = 0$ *gdw.* $v = w$

•$cost(v, w) = cost(w, v)$

•$cost(v, w) \leq cost(v, z) + cost(z, w)$

Die erste Bedingung stellt sicher, dass die Kosten nichtnegativ sind. Die zweite ist die Symmetrie der Kostenfunktion. Die dritte und vierte Bedingung sind als Dreiecksungleichung bekannt. Nun können die Kosten einer Transformation und der Begriff Editierdistanz definiert werden.

Definition 3.6 (Kosten einer Transformation) *Sei* $E \subseteq (V_1 \cup \{\lambda\}) \times (V_2 \cup \{\lambda\})$ *eine Transformation. Die Kosten der Transformation sind:* $cost(E) = \sum_{(v,w)\in E} cost(v, w)$ *die Summe der Kosten der einzelnen Editieroperation.*

Definition 3.7 (Editierdistanz) *Allgemein wird Editierdistanz zwischen zwei geordneten Bäumen* T_1 *und* T_2 *durch:*
$dist(T_1, T_2) = min\{cost(E) \mid E$ *ist eine valide* $Transformation$ *von* T_1 *in* $T_2\}$ *dargestellt.*
In dieser Arbeit wird die Editierdistanz durch die Anzahl der elementaren Operationen einer Transformation mit minimalen Kosten dargestellt.

Ein Ansatz, die günstigste Transformation zwischen zwei geordneten Bäumen zu berechnen, besteht darin, mit Hilfe der Knoten der Bäume, einen Graph aufzubauen, der Editiergraph genannt wird. Dadurch kann das Problem des Berechnens einer validen Transformation zwischen zwei Bäumen auf das Problem des Findens eines kürzesten Weges in dem Editiergraph reduziert werden. Dabei wird nach einem Pfad von dem oberen linken zu dem unteren rechten Graphknoten gesucht.

Definition 3.8 (Editiergraph) *Seien* $T_1 = (V_1, E_1)$ *und* $T_2 = (V_2, E_2)$ *geordnete Bäume. Der Editiergraph hat einen Knoten der Form* vw *für jedes Paar von Knoten* $v \in \{v_0\} \cup V_1$ *und* $w \in \{w_0\} \cup V_2$*, wo* $v_0 \notin V_1$ *und* $w_0 \notin V_2$ *„Dummyknoten“ sind. Desweiteren hat ein Editiergraph Kanten der Form:*

•*Vertikale Kante* $(v_i w_j, v_{i+1} w_j) \in E$ *gdw.* $depth(v_{i+1}) \geq depth(w_{j+1})$

•*Diagonale Kante* $(v_i w_j, v_{i+1} w_{j+1}) \in E$ *gdw.* $depth(v_{i+1}) = depth(w_{j+1})$

•*Horizontale* $(v_i w_j, v_i w_{j+1}) \in E$ *gdw.* $depth(v_{i+1}) \leq depth(w_{j+1})$

•$(v_i w_{n_2}, v_{i+1} w_{n_2}) \in E$ *und* $(v_{n_1} w_j, v_{n_1} w_{j+1})$ *für* $0 \leq i < n_1$ *und* $0 \leq j < n_2$ *für* $0 \leq i < n_1$ *und* $0 \leq j < n_2$*, wo die Knoten der Bäume nach deren preorder Reihenfolge durchnummeriert sind und* n_1 *bzw.* n_2 *die Nummer des letzten Knoten im Baum* T_1 *bzw.* T_2 *ist.*

In einem Editiergraph von zwei geordneten Bäumen T_1 und T_2 repräsentiert eine vertikale Kante der Form $(v_i w_j, v_{i+1} w_j)$ das Löschen des Knoten v_{i+1} aus dem Baum T_1. Eine diagonale Kante der Form $(v_i w_j, v_{i+1} w_{j+1})$ repräsentiert die Ersetzung des Knoten v_{i+1} aus dem Baum T_1 durch den Knoten w_{j+1} aus dem Baum T_2. Eine horizontale Kante der Form $(v_i w_j, v_i w_{j+1}) \in E$ repräsentiert das Einfügen des Knoten w_{j+1}. Der Knoten $w_{j+1} \in T_2$ wird als am weitesten rechts liegender Kindknoten des Knoten $v_i \in T_2$ eingefügt.

Das Fehlen horizontaler und diagonaler Kanten stellt sicher, dass sobald der Pfad eine vertikale Kante traversiert, welche die Entfernung eines Knoten v repräsentiert, kann er nur durch traversieren weiterer vertikaler Kanten erweitert werden. Dies bedeutet, dass alle Knoten im Unterbaum mit v als Wurzel ebenfalls gelöscht werden müssen. Das Fehlen vertikaler und diagonaler Kanten stellt sicher, dass sobald der Pfad eine horizontale Kante traversiert, welche das Einfügen eines Knoten w repräsentiert, kann er nur durch traversieren weiterer horizontaler Kanten erweitert werden. Dies bedeutet, dass alle Knoten im Unterbaum mit w als Wurzel ebenfalls eingefügt werden müssen.

Eine valide Transformation zwischen zwei geordneten Bäumen T_1 und T_2 stimmt also mit einem Pfad in dem Editiergraph der Bäume von dem oberen linken zu dem unteren rechten Graphknoten überein. Die Übereinstimmung ist dadurch gegeben, dass die Substitutionskanten entlang eines solchen Pfades ein Mapping des Baumes T_1 zu dem Baum T_2 festlegen. Existiert also ein Pfad P von dem oberen linken zu dem unteren rechten Graphknoten in einem Editiergraph der Bäume $T_1 = (V_1, E_1)$ und $T_2 = (V_2, E_2)$, dann bildet die Menge $M = \{(v_{i+1}, w_{j+1}) \in V_1 \times V_2 \mid (v_i w_j, v_{i+1} w_{j+1}) \in P\}$ ein Mapping von T_1 zu T_2. Umgekehrt gibt es ein Mapping $M \subseteq V_1 \times V_2$, dann gibt es auch einen Pfad P von dem oberen linken zu dem unteren rechten Graphknoten in einem Editiergraph der Bäume T_1 und T_2, sodass $\{(v_{i+1}, w_{j+1}) \in V_1 \times V_2 \mid (v_i w_j, v_{i+1} w_{j+1}) \in P\} = M$.

Die Berechnung der Editiersequenz und der Editierdistanz zwischen zwei geordneten Bäumen kann nun auf das Problem des Findens kürzester Wege in dem Editiergraph der Bäume reduziert werden. Die Kanten stellen elementare Editieroperationen dar und stellen sicher, dass nur valide Transformationen berechnet werden. Jede Kante wird bei der Umsetzung mit dem Gewicht der repräsentierten Operation gewichtet und mit dem Namen der Operation benannt. Das Finden der kürzesten Wege kann mit Hilfe des in 2.3.2.3 vorgestellten Algorithmus von Dijkstra erledigt werden.

3.3.2 Beispiel

Ein Beispiel einer Transformation ist in der Abbildung 3.2 dargestellt. Die Transformation besteht aus folgenden Operationen:
$\{(v_1, w_1), (v_2, w_2), (v_3, \lambda), (v_4, \lambda), (v_5, w_3), (\lambda, w_4), (\lambda, w_5), (\lambda, w_6), (\lambda, w_7)\}$ und beinhaltet das Löschen der Knoten v_3 und v_4, Ersetzung der Knoten v_1, v_2, v_5 in dem Baum T_1 durch Knoten w_1, w_2, w_3 aus dem Baum T_2 und Hinzufügen der Knoten w_4, w_5, w_6,w_7 aus dem Baum T_2 in den Baum T_1. Die Schritte der Ersetzung wurden der Einfachheit halber in der Abbildung ausgelassen. Das zugrundeliegende Mapping, das die Substitutionen zwischen den Knoten der Bäume darstellt, ist in der Abbildung 3.3 zu sehen.

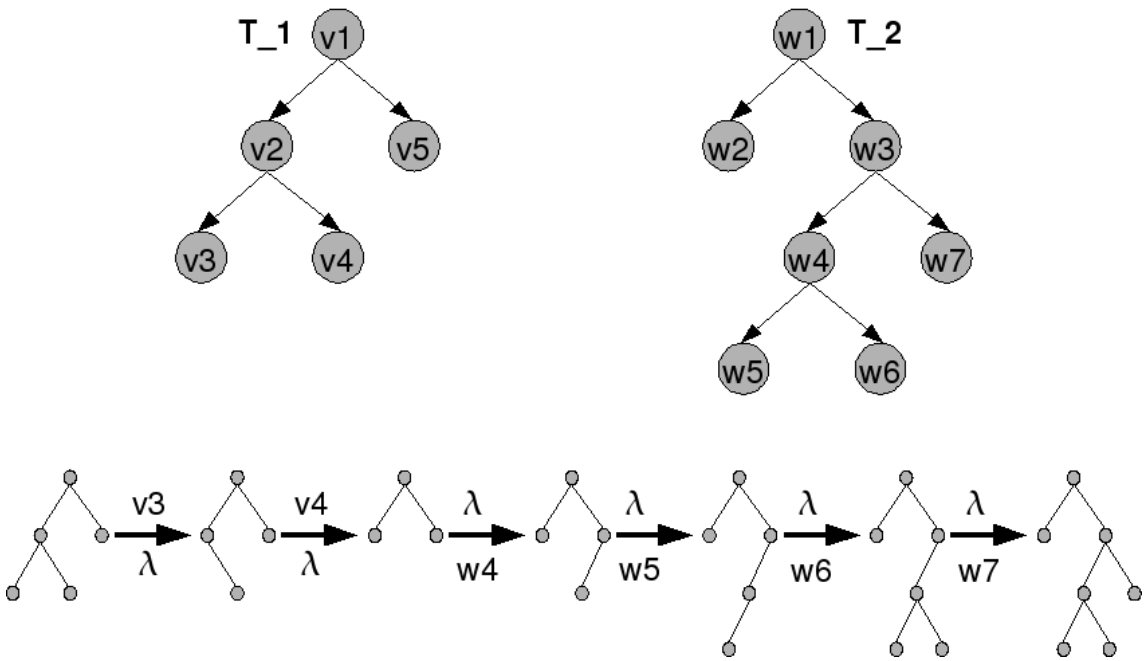

Abbildung 3.2: Vereinfachtes Beispiel einer Transformation.(Quelle: [Valiente, 2002])

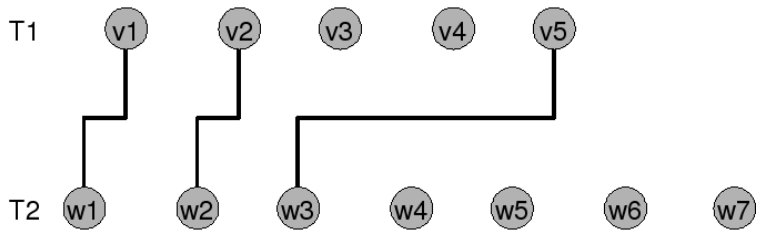

Abbildung 3.3: Mapping der Bäume T_1 zu T_2 aus Abbildung 3.2. (Quelle: [Valiente, 2002])

Die Abbildung 3.4 zeigt einen Editiergraph für Bäume T_1 und T_2 aus der Abbildung 3.2. In der Zeichnung sind die Knoten des Editiergraphen in einem rechteckigen Gitter angeordnet. Die Graphknoten werden aus den Knoten der Bäume gebildet. Diese sind nach der Reihenfolge, in der sie beim preorder Traversieren der Bäume besucht werden, geordnet. Die Kanten des Graphen sind alle von links nach rechts bzw. von oben nach unten gerichtet. Auf das Darstellen der Richtung der Kanten wurde aufgrund der Eindeutigkeit und der Einfachheit halber verzichtet. Die hervorgehobenen Kanten bilden einen kürzesten Weg von dem oberen linken zu dem unteren rechten Graphknoten. Dieser repräsentiert eine valide günstigste Transformation von $T_1 = (V_1, E_1)$ nach $T_2 = (V_2, E_2)$ aus Abbildung 3.2. Die Kosten der Editieroperationen, die durch die Kanten des Graphen dargestellt werden, sind in diesem Beispiel wie folgt festgelegt: $cost(v, w) = 0, cost(\lambda, w) = 1 = cost(v, \lambda)\ \forall v \in V_1\ und\ w \in V_2$. Die Substitution der Knoten hat Kosten von 0, das Hinzufügen und Löschen von Knoten hat Kosten von 1. Die Transformation der Bäume besteht aus folgenden Operationen: $[(v_1, w_1), (\lambda, w_2), (v_2, w_3), (v_3, w_4), (\lambda, w_5), (\lambda, w_6), (v_4, w_7), (v_5, \lambda)]$. Es werden Knoten $w_2, w_5, w_6 \in V_2$ in den Baum T_1 hinzugefügt, der Knoten $v_5 \in V_1$ wird gelöscht und die Knoten $v_1, v_2, v_3, v_4 \in V_1$ durch die Knoten $w_1, w_3, w_4, w_7 \in V_2$ ersetzt.

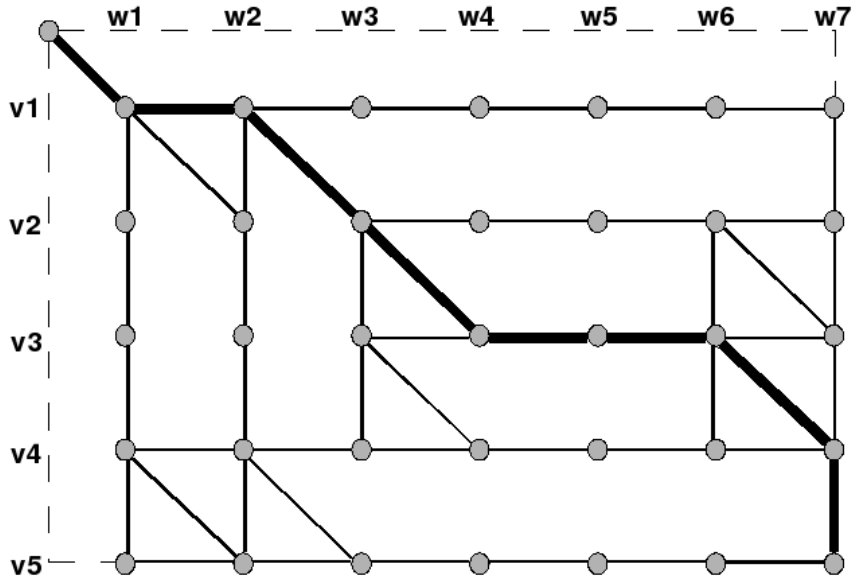

Abbildung 3.4: Editiergraph der Bäume aus der Abbildung 3.2.(Quelle: [Valiente, 2002])

3.3.3 Komplexität

Der beschriebene Algorithmus zum Finden der günstigsten Transformation zwischen zwei geordneten Bäumen, basierend auf der Suche nach einem kürzesten Weg in einem Editiergraph der Bäume, hat eine Laufzeitkomplexität von $O(n_1 n_2)$. Wobei n_1 bzw. n_2 die Anzahl der Knoten in den Bäumen T_1 bzw. T_2 darstellen. Zur Berechnung werden zusätzlich $O(n_1 n_2)$ Speicher benötigt. Die Laufzeit und Speicherkomplexität ist also, unter der Annahme, dass das Konstruieren von Knoten und Kanten in konstanter Zeit abläuft und konstant viel zusätzlichen Speicher benötigt, im „worst case“ quadratisch (Beweis siehe [Valiente, 2002] S.68).

3.3.4 Erweiterung für markierte Bäume

Der beschriebene Ansatz ist fähig, eine Editiersequenz zwischen zwei geordneten markierten Bäumen zu berechnen. Die Kriterien für das Einfügen und das Löschen von Knoten sind ausreichend. Ist an einer bestimmten Stelle im ersten Baum ein Knoten vorhanden, der im zweiten Baum an der selben Position nicht präsent ist, muss er gelöscht werden. Analog ist an einer bestimmten Stelle im zweiten Baum ein Knoten vorhanden, der im ersten Baum an der selben Position nicht präsent ist, muss er hinzugefügt werden. Das Kriterium für die Substitution der Knoten ist allerdings zu allgemein. Maßgebend für die Substitution, wie für das Einfügen und das Löschen, ist die Position der Knoten in den Bäumen. Eine diagonale Kante wird nur dann erzeugt, wenn die Tiefe der Knoten in den Bäumen gleich ist ($(v_i w_j, v_{i+1} w_{j+1}) \in E$ gdw. $depth(v_{i+1}) = depth(w_{j+1})$). Dies hat zur Folge, dass ein Knoten durch einen anderen ersetzt wird, sobald die Position der beiden Knoten in den jeweiligen Bäumen gleich ist, unabhängig davon, welche Information die Knoten beinhalten. Anwendungsbezogen ist dies unter Umständen nicht zulässig. Denn neben der Position im Baum spielt die in den Knoten gespeicherte Information eine entscheidende Rolle. Man muss die Knoten also nach ihrer Substituierbarkeit differenzieren.

Dass eine Substitutionskante erzeugt werden muss, steht außer Frage, somit kann eine ungewünschte Substitutionen nur durch das hohe Gewichten der Kante vermieden werden.

Beim Gewichten der Kante muss die in den Knoten enthaltene Information miteinbezogen werden. Abhängig vom Gewicht kann diese Kante dann im Pfad auftreten oder nicht, denn zu der Substitution zweier Knoten existiert immer eine Alternative. Anstatt einen Knoten v durch w zu ersetzen, kann man auch den Knoten v löschen und den Knoten w an seiner Position einfügen. Die Gewichte der Einfüge- und Löschkanten müssen gleich und konstant sein. Sie dienen als ein Orientierungswert für das Gewichten der Substitution.

Eine Funktion soll vor der Vergabe eines Gewichtes an eine Substitutionskante die Informationen in den Knoten vergleichen. Sind die Informationen beider Knoten gleich oder zumindest ähnlich zu einander, so muss das Gewicht der Substitutionskante **niedriger** als die Summe der Gewichte der Einfüge- und Löschkanten gesetzt werden. Weichen die Informationen beider Knoten stark voneinander ab, so muss das Gewicht der Substitutionskante **höher** als die Summe der Gewichte der Einfüge- und Löschkanten gesetzt werden.

Sei $equal\_or\_similar : v \times w \rightarrow \{TRUE, FALSE\}\ \forall v \in V_1\ und\ w \in V_2$ eine Funktion, welche die in den Knoten der Bäume $T_1 = (V_1, E_1)$ und $T_2 = (V_2, E_2)$ enthaltene Information auf ihre Gleichheit oder Ähnlichkeit zueinander prüft.
Abhängig vom Resultat des Vergleichs gilt also:

- $weight((v_i w_j, v_i w_{j+1})) > weight((v_i w_j, v_{i+1} w_{j+1})) < weight((v_i w_j, v_{i+1} w_j))$ gdw. $equal\_or\_similar(v_{i+1}, w_{j+1}) = TRUE$
- $weight((v_i w_j, v_{i+1} w_{j+1})) > weight((v_i w_j, v_{i+1} w_j)) + weight((v_i w_j, v_i w_{j+1}))$ gdw. $equal\_or\_similar(v_{i+1}, w_{j+1}) = FALSE$

Wie eine solche Funktion für die Zwecke dieser Arbeit implementiert werden kann, wird im Abschnitt 4.3.2 (Unterabschnitt 4.3.2.3) in Kapitel 4 beschrieben.

3.3.5 Postprocessing

Betrachtet man die Editiersequenz im Zusammenhang mit der Darstellung der Unterschiede zwischen Klonen, so werden Codefragmente für die Berechnung dieser Sequenz durch abstrakte Syntax Bäume repräsentiert. Auf diesen Bäumen wird dann, wie beschrieben, die Editiersequenz berechnet. Allerdings werden die abstrakten Syntax Bäume, schon bei relativ kleinen Codefragmenten, recht groß. Dementsprechend kann die Editiersequenz sehr viele Operationen beinhalten. Um dieser Problematik entgegen zu wirken, kann die berechnete Editiersequenz in einem separaten Schritt nachbearbeitet werden. Das Nachbearbeiten nennt man auch Postprcessing. Die möglichen Schritte des Postprocessing werden nun an einem Beispiel erläutert.

In der Abbildung 3.5 sind zwei geordnete Bäume dargestellt. Die Knoten der Bäume beinhalten, neben dem Label, auch Information in Form von Strings (dargestellt durch je zwei Buchstaben im Inneren der Kreise).
Die Transformation des Baumes T_1 in den Baum T_2 sieht wie folgt aus:
$\{(v_1, w_1), (v_2, w_2), (v_3, w_3), (v_4, w_4), (v_5, \lambda), (v_6, \lambda), (v_7, \lambda)\}$. Die Knoten $v_1, v_2, v_3, v_4 \in T_1$ werden durch die Knoten $w_1, w_2, w_3, w_4 \in T_2$ ersetzt und die Knoten $v_5, v_6, v_7 \in T_1$ werden gelöscht. Man sieht, dass die Information in den zu Substitution vorgeschlagenen Knoten v_1, v_2, v_3, v_4, die durch w_1, w_2, w_3, w_4 ersetzt werden sollen, die gleiche ist.

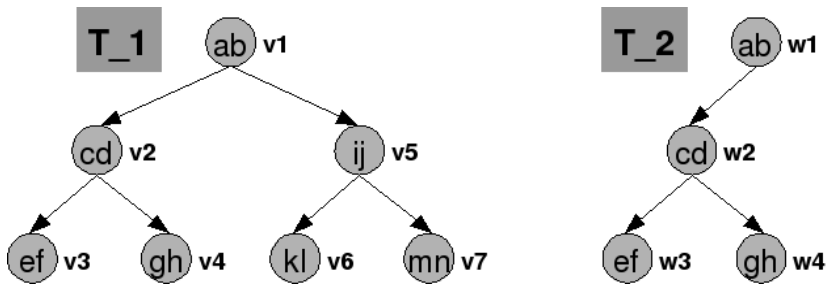

Abbildung 3.5: Beispiel der Bäume T_1 und T_2 zur Erläuterung des Postprocessing

In diesem Fall kann man also die ersten vier Operationen der Editiersequenz weglassen. Desweiteren beschreiben die letzten drei Operationen der Editiersequenz das Löschen des Teilbaumes mit v_5 als Wurzel. Die Einschränkungen auf das Löschen und Einfügen von Blattknoten bei der Berechnung einer Editiersequenz mit Hilfe des Editiergraphen besagen, dass wenn ein „Nichtblattknoten“ gelöscht werden muss, so muss der gesamte Teilbaum bis runter zu den Blattknoten ebenfalls gelöscht werden. Es ist also sichergestellt, dass wenn das Löschen eines „Nichtblattknoten“ in der Editiersequenz vorkommt, mit den darauf folgenden Operationen auch der gesamte Teilbaum gelöscht wird. Das selbe gilt für das Einfügen von „Nichtblattknoten“. Im Sinne des Postprocessing kann diese Eigenschaft benutzt werden, um die Anzahl der Operationen weiter zu reduzieren. In unserem Beispiel kann man also sagen, es wird der Teilbaum mit v_5 als Wurzel gelöscht. Beim Postprocessing der Editiersequenz wird das Löschen und Einfügen somit auf Teilbäume ausgeweitet. Hierdurch wird die Einschränkung beseitigt, dass das Löschen und das Einfügen nur auf Blättern möglich ist. Die Editiersequenz wurde somit von ursprünglich sieben Operationen auf nur eine Operation eingeschränkt.

Zusammengefasst kann man durch Postprocessing die Einschränkungen des Ansatzes beheben und die hohe Anzahl der Operationen in der Editiersequenz maßgeblich einschränken. Die Umsetzung und ausführliche Beschreibung der einzelnen Schritte des Postprocessing, bezogen auf das Berechnen der Unterschiede zwischen Codefragmenten, wird im Abschnitt 4.3.2 (Unterabschnitt 4.3.2.5) in Kapitel 4 erfolgen.

KAPITEL 4

Implementierung

Inhalt

In diesem Kapitel wird die Implementierung des Tools **„treedist“** beschrieben, das im Rahmen dieser Diplomarbeit entstanden ist. Im Abschnitt 4.1 wird die Gewinnung und Struktur der zur Eingabe verwendeten Daten beschrieben. Im Abschnitt 4.2 findet die Beschreibung der Bestandteile und deren Zusammenspiel in dem implementierten System statt. Anschließend werden im Abschnitt 4.3 die Datenstrukturen und Funktionen der einzelnen Module ausführlich erläutert. Besonders interessant für den Leser dürften die Unterabschnitte 4.3.2.3 und 4.3.2.5 des Abschnitts 4.3.2, Abschnitte 4.3.4 und 4.3.5 sein. Dort wird die Umsetzung der in Kapitel 3 genannten Erweiterungen des Editiergraph-Ansatzes ausführlich erläutert, Teile der IML-Hierarchie vorgestellt und die Idee, sowie Funktionen zur Erzeugung der Ausgabe an den Benutzer beschrieben. Die Erläuterung der einzelnen Module beinhaltet, neben der ausführlichen textuellen Beschreibung, auch Auszüge aus dem Quellcode. Die Implementierung erfolgte in der Programmiersprache Ada95. Die Kenntnisse dieser Programmiersprache unterstützen stark das Verstehen des Kapitels, sind jedoch nicht zwingend notwendig.

4.1 Auswahl eines Klonerkennungstools

Um die Typ3 Klone in einem System analysieren zu können, müssen diese zunächst identifiziert werden. In Kapitel 2.2.2 wurden mehrere Klonerkennungstools vorgestellt, die verschiedene Verfahren implementieren und kombinieren. Jedes Verfahren hat seine Vor- aber auch Nachteile. Das Ziel in diesem Kontext ist, dass die Ergebnisse helfen sollten, die gemeldeten Typ3 Klone während Wartungsarbeiten zu beseitigen. Das Tool zu Gewinnung der Klondaten muss somit fähig sein, die Typ3 Klone im System zu identifizieren. Desweiteren müssen die Klonepaare in Form von abstrakten Syntax Bäumen vorliegen.

In [Berger, 2007] (Abschnitt 3.3.1) werden die einzelnen „Bauhaus“-Tools zu Erkennung der Klone miteinander verglichen. Seine Untersuchung zeigt, dass für die gestellte Aufgabe und gewählter Lösungsansatz nur *ccdiml* als Klonerkennungstool in Frage kommt. Für die Klonerkennung mittels ccdiml spricht, dass hier die syntaktische Abgeschlossenheit garantiert ist und so die Durchführung von Refactorings vereinfacht wird. Außerdem können die identifizierten Klonpaare nur von *ccdiml* direkt in der IML annotiert werden, was das Extrahieren der abstrakten Syntax Bäume der Klonpaare ermöglicht.

4.1.1 Annotation der Klone in der IML

Für die Annotation der Klonpaare in der IML existiert ein eigenes Konzept. Dieser wird durch die IML Klasse Namens $Clone\_Info$ beschrieben. In der Abbildung 4.1 wird die Hierarchie dieser Klasse dargestellt. Die abstrakten Klassen sind gelb markiert, konkrete Klassen sind weiß.

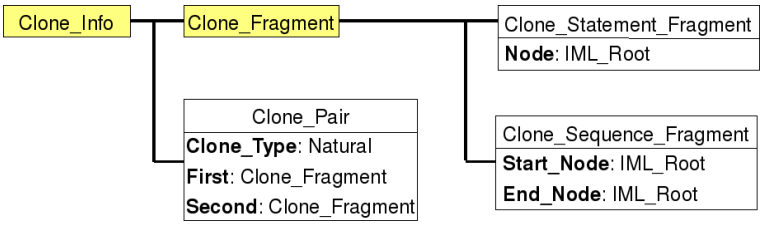

Abbildung 4.1: Konzept der Clone_Tool_Info

Ein Klonpaar wird durch die Klasse $Clone\_Pair$ modelliert. Diese enthält neben dem Attribut $Clone\_Type$, das den Typ eines Klonpaares repräsentiert, auch Attribute $First$ und $Second$. $First$ ist ein Verweis auf das Codefragment des Originals und $Second$ ist dementsprechend ein Verweis auf das Codefragment der Kopie. Die Codefragmente der Klone werden durch die Klasse $Clone\_Fragment$ modelliert. Klassen $Clone\_Statement\_Fragment$ und $Clone\_Sequence\_Fragment$ leiten von dieser ab. $Clone\_Statement\_Fragment$ modelliert einfache Anweisungen und enthält nur ein Verweis auf einen IML Knoten, der die Wurzel des ASTs der Anweisung darstellt. Die Klasse $Clone\_Sequence\_Fragment$ repräsentiert ganze Anweisungssequenzen und enthält Attribute $Start\_Node$ und End_Node. $Start\_Node$ und End_Node sind Verweise auf die Wurzelknoten der ASTs der ersten und der letzten Anweisung der Sequenz. Alle Anweisungen liegen somit innerhalb eines Bereiches, der durch den $Start\_Node$ und End_Node eingegrenzt wird. Die Wurzelknoten der ASTs der Anweisungen innerhalb eines solchen Bereichs haben alle den selben Elternknoten. In der Abbildung 4.2 wird durch ein vereinfachtes Beispiel dargestellt, wie ein Klonpaar in der IML modelliert wird.
Das Klonerkennungstool ccdiml erzeugt nach der Analyse der IML einen $Clone\_Info$ Knoten, der wiederum eine Liste von Verweisen auf $Clone\_Pair$ Knoten enthält. Diese Liste beinhaltet also Klonpaare sämtlicher Klonkandidaten, die im analysierten System gefunden wurden.

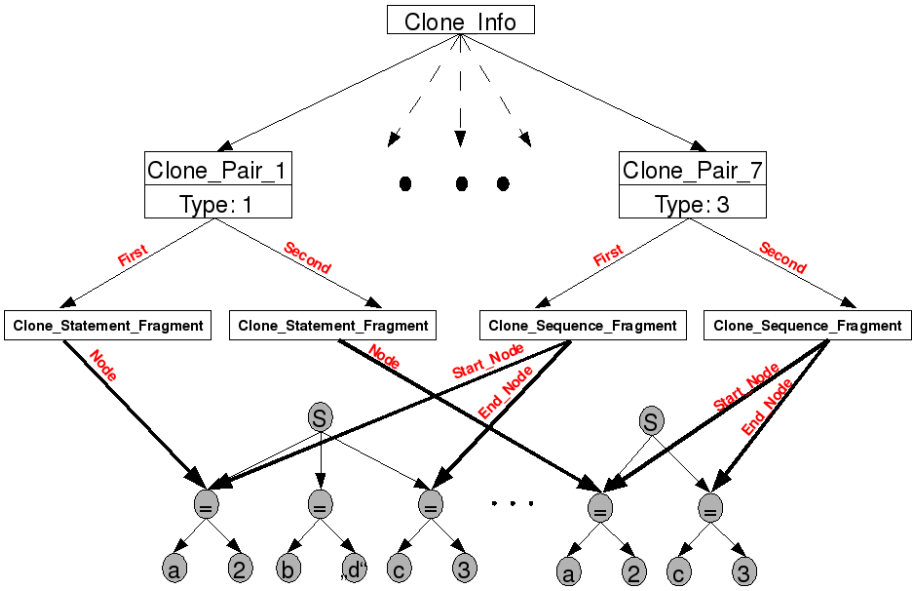

Abbildung 4.2: Vereinfachte Darstellung eines Klonpaares in der IML durch *Clone_Pair* Klasse

4.2 Architektur

Die Architektur des Systems ist relativ einfach aufgebaut. Es handelt sich um einen Substill der „call and return“ Architektur. Diese Art der Architektur nennt sich „Hauptprogramm-Unterprogramm-Architektur“ (*main program and subroutine architecture*). Es gibt einen Hauptkontroll- und Datenfluss. Jede Komponente in der Hierarchie bekommt sukzessive die Kontrolle und Daten von ihrer übergeordneten Komponente, gibt diese jeweils an untergeordnete Komponenten weiter und anschließend wird die Kontrolle wieder an die ursprünglich aufrufende Komponente nach oben in der Hierarchie zurückgegeben (Quelle: [Hauswirth 2003]).

Das implementierte System besteht aus insgesamt sechs Komponenten, im folgenden Module genannt, die unterschiedliche Aufgaben übernehmen. Um die Berechnung durchzuführen, müssen zunächst die Typ3 Klone aus der IML-Datei extrahiert werden. Für diese Aufgabe ist das Modul **„clone_utils“** verantwortlich. Jedes einzelne Klonpaar wird extrahiert und in ein Paar abstrakter Syntax Bäume umgewandelt, die dann auf Unterschiede untersucht werden können. Vor der Berechnung der Unterschiede werden die Quellcodefragmente eines Klopaares mit Hilfe des Moduls **„file_handling_utils“** in der Konsole ausgegeben. Hierfür wird der Pfad zu der Quellcodedatei, sowie Start- und Endzeile der jeweiligen Codefragmente an das Modul übergeben. Zu Berechnung der Unterschiede werden die abstrakten Syntax Bäume eines Klonpaares an das Modul **„edit_graph“** übergeben. In diesem Modul wird ein Editiergraph der ASTs aufgebaut und eine Transformation berechnet, die dann in eine Editiersequenz umgewandelt und nachverarbeitet wird. Beim Aufbauen des Editiergraphen werden die abstrakten Syntax Bäume der Codefragmente mit Hilfe des **„iml_tree_utils“** Moduls, durch preorder Traversierung linearisiert. Beim Gewichten der Substitutionskanten des Editiergraphen wird das **„iml_class_tag_comparator“** Modul verwendet, das die Nachbildung der IML-Klassenhierarchie in Form einer statischen Datenstruktur enthält. Die postprocessierte Editiersequenz wird schließlich an das **„output“** Modul weitergereicht, um eine Benutzerausgabe zu erzeugen.

In der Abbildung 4.3 ist die Architektur des Systems in Form eines Blockdiagramms mit Kontroll- und Datenflusskomponenten dargestellt.

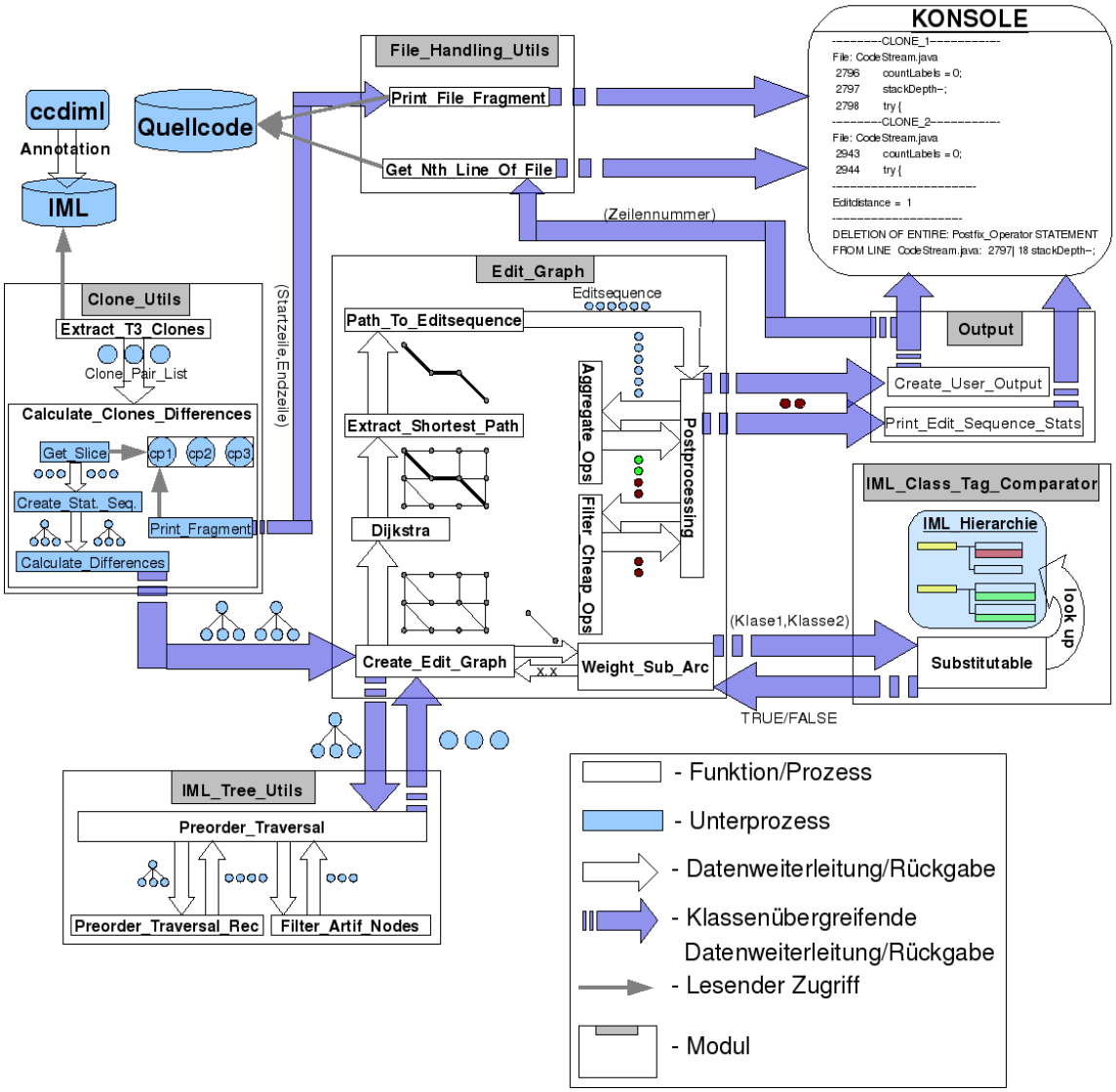

Abbildung 4.3: Architektur

4.3 Module

In diesem Abschnitt werden die Aufgaben und Funktionsweise der implementierten Module beschrieben. Besonderes Augenmerk liegt hier auf den Sachverhalten, die für IML spezifisch sind. Es wird jedoch darauf verzichtet, die Funktionalität der verwendeten „Bauhaus“-Bibliotheken ausführlich zu beschreiben.

4.3.1 „clone_utils“ Modul

Das Modul *clone_utils* ist für die Extraktion der Klonpaare aus der IML verantwortlich. Voraussetzung hierfür ist, dass die Klonepaare identifiziert und in dem im Abschnitt 4.1.1 beschrieben Format annotiert wurden. Jedes einzelne Klonpaar wird extrahiert und in ein Paar abstrakter Syntax Bäume umgewandelt, die dann auf Unterschiede untersucht werden können.

Das Einlesen der IML ist relativ einfach und kann mit Hilfe der von „Bauhaus“ bereitgestellten Bibliotheken *IML_Graphs* und *IML.IO* erledigt werden. In dem eingelesenen IML-Graph kann nun nach dem *Clone_Info* Knoten gesucht werden, um mit seiner Hilfe die Klonpaare aus der Liste der Kandidaten zu extrahieren. Dabei muss darauf geachtet werden, dass nur Typ3 Klone extrahiert werden. Diese Aufgabe wird von der Funktion **Extract_T3_Clones** des Moduls übernommen. Die Funktion extrahiert alle vorhanden Klonpaare vom Typ3, speichert sie in einer Liste und gibt diese Liste letztendlich zurück.

Nach der Extraktion der Typ3 Klonpaare müssen die Bäume der Klone für den nachfolgenden Vergleich extrahiert werden. Wie bereits beschrieben, enthält jedes *Clone_Pair* zwei Verweise. Es handelt sich entweder um *Clone_Statement_Fragment* Knoten oder um *Clone_Sequence_Fragment* Knoten. Der erste und der zweite *Clone_Statement_Fragment* Knoten enthalten nur Verweise auf zwei IML Knoten, welche die Wurzelknoten der abstrakten Syntax Bäume beider Anweisungen darstellen. Die beiden Wurzelknoten zusammen mit deren Unterbäumen können direkt für den Vergleich genutzt werden.
Bei *Clone_Sequence_Fragments* müssen die Bäume erst konstruiert werden, da es sich hierbei nur um Verweise auf Start und Endknoten zweier Anweisungssequenzen handelt. Die Konstruktion der Bäume der Anweisungssequenzen wird von zwei Funktionen des Moduls übernommen. Die erste Funktion heißt **Get_Nodes_Between_Start_And_End** und ist für die Extraktion der Knoten in dem Bereich zwischen dem Start- und Endknoten verantwortlich. Es werden alle Wurzelknoten der Teilbäume, die in diesem Bereich liegen, in einer Liste gespeichert und zurückgegeben. Hierdurch entsteht eine Menge an losen Teilbäumen, die jedoch zu einer Anweisungsequenz gehören. Um diese zusammenzusetzen, wird ein künstlicher *Statement_Sequence* Knoten erzeugt und alle Teilbäume aus der Menge werden als Kindknoten and diesen angehängt. Diese Aufgabe wird von der Funktion **Create_Statement_Sequence** übernommen. Bei dieser Prozedur geht die Verbindung zum eigentlichen Elternknoten der Teilbäume verloren und muss nach der Berechnung der Unterschiede wieder hergestellt werden. Hierfür muss der Elternknoten zwischengespeichert werden. Die Wiederherstellung der Beziehung ist notwendig, da die herausgelösten Teilbäume zusammen mit den Elternknoten zu anderen Klonpaaren gehören können.

In der Abbildung 4.4 werden die Schritte an einem allgemeinen Beispiel verdeutlicht. Die Prozedur **Calculate_Clones_Differences** führt diese einzelnen Schritte aus. Als Eingabe bekommt sie eine mit Hilfe von **Extract_T3_Clones** extrahierte Liste aller im System vorhandenen Typ3 Klonpaare. Jedes einzelne Klonpaar wird verarbeitet. Dabei wird wie oben beschrieben vorgegangen. Handelt es sich bei einem Klonpaar um zwei *Clone_Statement_Fragment*, so werden die verwiesenen Bäume direkt an den Vergleichsalgorithmus weitergeleitet. Stößt man auf ein Paar von *Clone_Sequence_Fragments*, so werden zunächst die zu vergleichenden Bäume wie beschrieben konstruiert, an den Vergleichsalgorithmus weitergeleitet und schließlich wird der ursprüngliche Zustand wiederhergestellt.
Um Bäume zu vergleichen und eine Ausgabe zu erzeugen, werden Funktionen aus dem **edit_graph** Modul aufgerufen, das im folgenden Abschnitt 4.3.2 beschrieben wird.

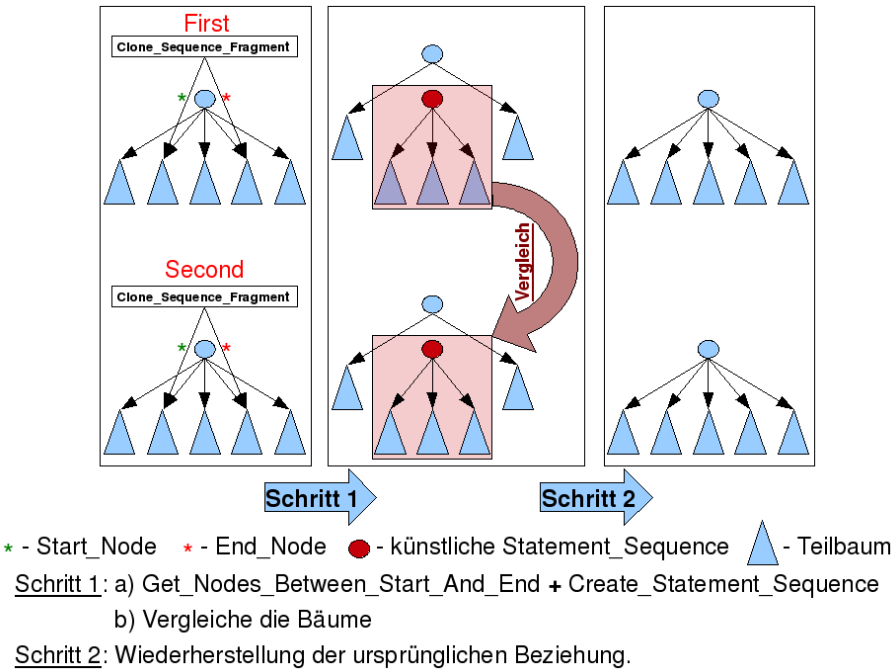

Abbildung 4.4: Schritte bei Extraktion der Bäume aus *Clone_Sequence_Fragment*

4.3.2 „edit_graph“ Modul

Das edit_graph Modul ist für das Vergleichen zweier abstrakter Syntax Bäume und das Postprocessing verantwortlich. In dem Abschnitt 2.3.2 beschriebene Datenstruktur Graph, Dijkstra's Algorithmus zum Suchen kürzester Wege (siehe Abschnitt 2.3.2.3) und der im Abschnitt 3.3 dargestellte Editiergraph Ansatz, zusammen mit seinen Erweiterungen, treffen hier aufeinander. Im Folgenden wird die Implementierung der Datenstrukturen und Algorithmen, sowie deren Zusammenspiel bei der Problemlösung erläutert.

4.3.2.1 Struktur des Editiergraphen

Knoten des Editiergraphen
Ein Graphknoten hat in dem Editiergraph Ansatz die Form vw, wobei v ein Knoten des ersten und w ein Knoten des zweiten Baums ist. Die Bäume für den Vergleich werden aus der IML extrahiert. Es handelt sich also bei den Baumknoten um IML-Knoten. Die beiden IML-Knoten und weitere Informationen werden in der Datenstruktur *Node_Pair* zusammengefasst. Das Listing 4.1 zeigt das *Node_Pair* Konzept, das durch ein Ada-Record beschrieben wird.

Listing 4.1: Definition des *Node_Pair* Konzeptes

```
type Node_Pair_Rec is record
    ------------------------------------------
    Node_1 : Storables.Storable; --v
    Preorder_Number_Of_1: Natural;
    Depth_Of_1: Natural;
    Primary_Label_Of_1: Ada.Strings.Unbounded.Unbounded_String;
    Secondary_Label_Of_1: Ada.Strings.Unbounded.Unbounded_String;
```

```
    ----------------------------------------
    Node_2 : Storables.Storable; --w
    Preorder_Number_Of_2: Natural;
    Depth_Of_2: Natural;
    Primary_Label_Of_2: Ada.Strings.Unbounded.Unbounded_String;
    Secondary_Label_Of_2: Ada.Strings.Unbounded.Unbounded_String;
end record;
type Node_Pair is access all Node_Pair_Rec;
```

Die Attribute *Node_1* und *Node_2* beinhalten die IML-Knoten des ersten und des zweiten Baumes. Attribute *Preorder_Number_Of_1* und *Preorder_Number_Of_2* speichern die preorder Reihenfolge des ersten und des zweiten Knoten in den entsprechenden Bäumen.
Die Attribute *Depth_Of_1* und *Depth_Of_2* repräsentieren die Tiefe des ersten und des zweiten Knoten in den entsprechenden Bäumen. Durch die preorder Nummer und die Tiefe eines Knoten wird seine Position in einem Baum beschrieben.
Die Attribute *Primary_Label_Of_1* und *Secondary_Label_Of_1* bzw. *Primary_Label_Of_2* und *Secondary_Label_Of_2* speichern eine Beschreibung des ersten bzw. des zweiten IML-Knoten. Der primäre Label enthält lediglich den Bezeichner der IML-Klasse in Form eines Strings und gibt an, um welche Knotenart es sich handelt. Der sekundäre Label speichert ebenfalls in Form eines Strings die Attribute der zugehörigen Klasse und kann auch semantische Informationen beinhalten. Eine Variable kann in dem IML-Graph durch einen Knoten der Klasse *Entity_L_Value* dargestellt werden. In diesem Fall würde der primäre Label einen „Entity_L_Value“ String beinhalten. Der sekundäre Label würde unter Anderem die Identifikationsnummer des Knoten im Graph und den Namen der Variablen speichern.

Die Funktion **New_Node_Pair** bekommt als Eingabe zwei IML-Knoten, die preorder Nummern der Knoten und die Tiefe der Knoten in den entsprechenden Bäumen. Diese Funktion erzeugt für die Rückgabe ein *Node_Pair* Objekt. Ein solches Objekt ist eines der wichtigsten Bestandteile des *Graph_Node* Objektes.

Die preorder Nummer und die Tiefe der Knoten spielen eine tragende Rolle beim Erstellen der Graphkanten und werden mit Hilfe des Moduls „iml_tree_utils“ (siehe 4.3.3) berechnet. Die beiden Labels haben für die Gewichtung der Graphkanten eine große Bedeutung und werden vor allem für die Entscheidung, ob ein Knoten durch einen anderen ersetzt werden kann, hinzugezogen. Die preorder Traversierung eines Baumes wurde im Abschnitt 2.3.3.3 beschrieben. Die Tiefe eines Knoten in einem Baum wurde im Abschnitt 2.3.3 definiert. Das Listing 4.2 zeigt das *Graph_Node* Konzept, das durch ein Ada-Record beschrieben wird.

Listing 4.2: Definition des *Graph_Node* Konzeptes

```
type Graph_Node_Rec is record
   ID : Natural;
   Value : Node_Pair;
   Distance: Float;
   Shortest_Path_Predecessor_ID : Integer;
end record;
type Graph_Node is access all Graph_Node_Rec;
```

Ein Graphknoten des Editiergraphen beinhaltet Attribute *ID*, *Value*, *Distance* und *Shortest_Path_Predecessor_ID*. Das Attribut *ID* ist eine eindeutige Identifikationsnummer des Graphknoten in dem Editiergraph und wird von der Funktion **Generate_Node_Id** erzeugt.

Dieses soll neben der Identifikation des Knoten auch die Suche nach einem Graphknoten erleichtern. Das Attribut *Value* beinhaltet das oben beschriebene *Node_Pair* mit allen für die Berechnung benötigten Informationen über die IML-Knoten. Das Attribut *Distance* speichert die Entfernung des Graphknoten zu dem ausgezeichneten ersten Knoten des Graphen. Das Attribut *Shortest_Path_Predecessor_ID* repräsentiert die Identifikationsnummer des Knoten, der auf dem kürzesten Weg im Graph vor dem aktuellen Knoten liegt. Die zwei letzten Attribute werden bei der Suche nach einem kürzesten Weg von dem ersten zu dem letzten Knoten im Graph verwendet.

Die Funktion **New_Graph_Node** bekommt als Eingabe einen *Node_Pair* Objekt und erzeugt für die Rückgabe ein *Graph_Node* Objekt. Das Attribut *Distance* wird bei der Erzeugung des Graph-Knoten auf 999999.0 gesetzt und symbolisiert ∞. Hierdurch kann der erste Schritt bei der Implementierung des Algorithmus zum Finden kürzester Wege ausgelassen werden. Das Attribut *Shortest_Path_Predecessor_ID* wird bei der Erzeugung des Graph-Knoten auf 0 gesetzt und symbolisiert, dass dieser Knoten noch keinen Vorgänger auf einem kürzesten Weg aufweist. Mit Hilfe der Funktion **Add_Node** kann ein Graph-Knoten in die Menge der Knoten eines Graphen eingefügt werden. Dabei wird zur Sicherheit überprüft, ob ein solcher Knoten nicht bereits vorhanden ist. Die Gesamtmenge der Knoten in einem Graph kann nach bestimmten Knoten durchsucht werden, um diese zum Beispiel durch eine Graph-Kante zu verbinden. Die Suche nach einem Knoten wird von den Funktionen **Get_Node_By_Id** oder **Get_Graph_Node_By_Tree_Nodes_Preorder** übernommen. Die erste Funktion sucht anhand der Identifikationsnummer nach einem Knoten. Die zweite Funktion benötigt die preorder Nummern der IML-Knoten, die im *Node_Pair* Objekt in jedem Graph-Knoten mitgeführt werden.

Kanten des Editiergraphen
Das Listing 4.3 zeigt das *Graph_Edit_Arc* Konzept, das durch ein Ada-Record beschrieben wird. In dem Editiergraph Ansatz sind Kanten gerichtet, markiert und gewichtet. Jede Kante des Graphen repräsentiert eine Editieroperation (siehe Definition 3.8).

Listing 4.3: Definition des *Graph_Edit_Arc* Konzeptes

```
type Graph_Edit_Arc_Rec is record
    ID: Natural;
    Operation: Ada.Strings.Unbounded.Unbounded_String;
    Source: Graph_Node;
    Target: Graph_Node;
    Costs: Float;
 end record;
 type Graph_Edit_Arc is access all Graph_Edit_Arc_Rec;
```

Das Attribut *ID* ist, analog zu der *ID* der Graphknoten, eine eindeutige Identifikationsnummer der Graphkante in dem Editiergraph und wird von der Funktion **Generate_Arc_Id** erzeugt. Das Attribut *Operation* ist die Markierung der Kante in Form eines Strings. Dieses Attribut kann die Werte „Ins“, „Del“ und „Sub“ annehmen, die durch die Kante repräsentierte Editieroperation beschreiben. Das Attribut *Source* beinhaltetet einen Graphknoten, der die Quelle der Kante darstellt. Analog beinhaltet das Attribut *Target* einen Graphknoten, der das Ziel der Kante darstellt. Durch diese zwei Attribute ist die Richtung der Kante eindeutig bestimmt. Das Attribut *Costs* beinhaltet das Gewicht der Kante, das die Berechnung eines kürzesten Weges im Graph maßgeblich beeinflusst. Die Kosten der Kanten für die Operationen Einfügen und Löschen sind konstant. Die Kosten der Kanten für die Operation der Substitution müssen, abhängig von der Information der IML-Knoten, bestimmt werden.

Die Berechnung der Substitutionkosten wird von der Funktion **Weight_Sub_Arc** erledigt. Die grundlegenden Überlegungen, zu der Umsetzung einer solchen Funktion, wurden im Abschnitt 3.3.4 geliefert. Die Details zu Implementierung werden in 4.9 ausführlich behandelt.

Die Funktion **New_Graph_Edit_Arc** erzeugt ein *Graph_Edit_Arc* Objekt und gibt diess zurück. Hierfür ist die Angabe des Quell- und Zielknoten notwendig. Die Funktion **Get_Arc** ist für das Suchen von Graph-Kanten in der Gesamtmenge der Kanten eines Graphen verantwortlich. Dabei wird gezielt nach Kanten gesucht, die eine bestimmte Quelle und ein bestimmtes Ziel haben. Falls ein solches *Graph_Edit_Arc* Objekt existiert, wird dieses zurückgegeben. Mit Hilfe der Funktionen **Get_Outgoing_Arcs** und **Get_Incoming_Arcs** kann man nach den ausgehenden und den eingehenden Kanten eines Knoten suchen. Die erste Funktion wird bei der Implementierung des Algorithmus zum Suchen kürzester Wege im Graph verwendet.

Datenstruktur des Editiergraphen
Wie im Abschnitt 2.3.2 definiert, besteht ein Graph aus einer Menge von Knoten und eine Menge von Kanten. In einem Editiergraph sind zusätzlich der erste (oben links in der Darstellung) und der letzte (unten rechts in der Darstellung) Knoten von Bedeutung. Deshalb werden diese beiden Knoten, zusätzlich zu der Gesamtmenge der Knoten, in die Definition der Struktur mitaufgenomen. Die Datenstruktur des Graphen wird durch ein Ada-Record beschrieben (siehe Listing 4.4).

Listing 4.4: Definition des *Graph* Konzeptes

```
package Graph_Nodes is new Lists (Graph_Node);
package Graph_Arcs is new Lists (Graph_Edit_Arc);

type Graph is record
  All_Graph_Nodes : Graph_Nodes.List := Graph_Nodes.Create;
  All_Graph_Arcs : Graph_Arcs.List := Graph_Arcs.Create;
  First_Node : Graph_Node;
  Last_Node: Graph_Node;
end record;
```

Die Attribute *All_Graph_Nodes* und *All_Graph_Arcs* beinhalten Listen mit Graphknoten und Graphkanten. Die Attribute *First_Node* und *Last_Node* speichern, wie oben beschrieben, die beiden ausgezeichneten Graphknoten. Diese beiden Knoten sind Start- und Endknoten bei der Suche nach einem Weg im Editiergraph.

4.3.2.2 Konstruktion des Editiergraphen

Die Konstruktion des Graphen geschieht nach den im Abschnitt 3.3 erläuterten Vorschriften und ist im Funktionsinneren in mehrere Schritte unterteilt. Um die Knoten des Editiergraphen zu konstruieren, werden zwei geordnete Bäume benötigt. Diese werden an die Funktion **Create_Graph_From_Trees**, in Form von zwei IML-Knoten, übergeben. Die beiden IML-Knoten sind Wurzelknoten der zu vergleichenden abstrakten Syntax Bäume der Codefragmente. Die IML-Knoten der Bäume werden nach deren preorder Reihenfolge in einer Liste gespeichert, um die Bäume zu linearisieren. Hierfür wird die Funktion **Preorder_Traversal** des Moduls **„iml_tree_utils“** aufgerufen. Die genaue Beschreibung der Funktionen dieses Moduls wird im Abschnitt 4.3.3 gegeben. Zu Konstruktion der Graphknoten und der Graphkanten muss die Länge (Anzahl der Knoten im Baum) der beiden Bäume bekannt sein. Das Listing 4.5 zeigt die ersten zur Konstruktion des Editiergraphen benötigten Schritte.

Listing 4.5: Linearisierung der IML-Bäume

```
Tree_1_Node_List :=
    IML_Tree_Utils.Preorder_Traversal(Tree_Root_1);
Tree_2_Node_List :=
    IML_Tree_Utils.Preorder_Traversal(Tree_Root_2);
N1 :=
    IML_Tree_Utils.Node_List.Length(Tree_1_Node_List);
N2 :=
    IML_Tree_Utils.Node_List.Length(Tree_2_Node_List);
```

Nach der Linearisierung und Berechnung der Länge der Bäume können nun die Knoten des Editiergraphen erzeugt werden. Die Variablen $N1$ und $N2$ stellen die Länge der Bäume dar. Die Listen *Tree_1_Node_List* und *Tree_2_Node_List* beinhalten alle IML-Knoten der jeweiligen IML-Bäume. Es werden insgesamt $N1 \times N2$ Graphknoten mit Hilfe der beiden Listen erzeugt. Das Listing 4.6 zeigt die einzelnen Schritte der Erzeugung von Graphknoten. Das Aufbau eines Graphknoten und die zur Konstruktion verwendeten Funktionen wurden bereits in 4.3.2.1 erläutert. An dieser Stelle werden zur Erzeugung benötigte Informationen geliefert.

Listing 4.6: Erzeugen der Graphknoten

```
for I in 0 .. N1-1 loop
 for J in 0 .. N2-1 loop
  Current_IML_Node_1:=
      IML_Tree_Utils.Node_List.Get_Nth(
                          Tree_1_Node_List,I);
  Current_IML_Node_2:=
      IML_Tree_Utils.Node_List.Get_Nth(
                          Tree_2_Node_List,J);
  Tree_Node_1_Depth :=
       IML_Tree_Utils.Get_Nodes_Depth2(
                                Current_IML_Node_1);
  Tree_Node_2_Depth :=
       IML_Tree_Utils.Get_Nodes_Depth2(
                            Current_IML_Node_2);
  Add_Node(Return_Graph,
           New_Graph_Node(
              New_Node_Pair(Current_IML_Node_1,
                            Current_IML_Node_2,
                            I,J,
                            Tree_Node_1_Depth,
                            Tree_Node_2_Depth)));
 end loop;
end loop;
```

Nach der Konstruktion der eigentlichen Knoten wird noch ein Dummyknoten erzeugt. Dieser wird mit dem ersten erzeugten Graphknoten durch eine Substitutionkante verbunden. Der Dummyknoten wird als *First_Node* des Graphen betrachtet. Der letzte erzeugte Graphknoten wird als *Last_Node* des Graphen betrachtet.

Für die Erzeugung der Kanten müssen die in der Definition 3.8 gemachten Einschränkungen beachtet werden. Der Letzte Punkt der Definition besagt, dass ein Editiergraph Kanten der Form: $(v_i w_{n_2}, v_{i+1} w_{n_2}) \in E$ und $(v_{n_1} w_j, v_{n_1} w_{j+1})$ für $0 \leq i < n_1$ und $0 \leq j < n_2$ aufweist. Diese Kanten verlaufen in der Darstellung am äußeren Rand des Gitters. Hierdurch wird sichergestellt, dass zumindest der einfachste mögliche Pfad zwischen dem ersten und dem letzten Knoten des Graphen existiert. Betrachtet man diesen Pfad als eine Transformation zwischen den beiden Bäumen, so werden alle Knoten des ersten Baumes gelöscht und alle Knoten des zweiten Baumes in den ersten eingefügt. Es wird somit sichergestellt, dass immer eine Transformation zwischen zwei geordneten Bäumen existiert. Listing 4.7 zeigt die Erzeugung dieser äußeren Kanten. Die Graphknoten werden in zwei Schleifen, entlang des Randes in der Darstellung des Graphen, durch eine vertikale bzw. horizontale Kante verbunden. Da es sich hierbei um Einfüge- und Löschkanten handelt, werden die Kosten dieser auf 1.0 gesetzt.

Listing 4.7: Erzeugen der äußeren Graphkanten

```
for I in 0 .. N1-2 loop
 New_Graph_Edit_Arc(
     Return_Graph,
     del,
     Get_Graph_Node_By_Tree_Nodes_Preorder(
                         Return_Graph, I, N2-1),
     Get_Graph_Node_By_Tree_Nodes_Preorder(
                         Return_Graph, I+1, N2-1),
     1.0);
end loop;
for J in 0 .. N2-2 loop
 New_Graph_Edit_Arc(
     Return_Graph,
     ins,
     Get_Graph_Node_By_Tree_Nodes_Preorder(
                             Return_Graph, N1-1, J),
     Get_Graph_Node_By_Tree_Nodes_Preorder(
                             Return_Graph, N1-1, J+1),
     1.0);
end loop;
```

Nach der Konstruktion der äußeren Kanten müssen die inneren Kanten erzeugt werden. Solche Kanten haben die in 3.8 durch die ersten drei Punkte definierte Form. Listing 4.8 zeigt die Erzeugung der inneren Kanten.

Listing 4.8: Erzeugen der inneren Graphkanten

```
for I in 0 .. N1-2 loop
 for J in 0 .. N2-2 loop
  Current_IML_Node_1 :=
          IML_Tree_Utils.Node_List.Get_Nth(
                        Tree_1_Node_List,I+1);
  Current_IML_Node_2 :=
          IML_Tree_Utils.Node_List.Get_Nth(
                        Tree_2_Node_List,J+1);
  Tree_Node_1_Depth :=
       IML_Tree_Utils.Get_Nodes_Depth2(
                      Current_IML_Node_1);
  Tree_Node_2_Depth :=
      IML_Tree_Utils.Get_Nodes_Depth2(
                     Current_IML_Node_2);

  if Tree_Node_1_Depth >= Tree_Node_2_Depth then
   New_Graph_Edit_Arc(
       Return_Graph,
       del,
       Get_Graph_Node_By_Tree_Nodes_Preorder(
                              Return_Graph, I,J),
       Get_Graph_Node_By_Tree_Nodes_Preorder(
                            Return_Graph, I+1, J),
       1.0);
  end if;
  if Tree_Node_1_Depth = Tree_Node_2_Depth then
   New_Graph_Edit_Arc(
       Return_Graph,
       sub,
       Get_Graph_Node_By_Tree_Nodes_Preorder(
                              Return_Graph, I, J),
       Get_Graph_Node_By_Tree_Nodes_Preorder(
                              Return_Graph, I+1, J+1),
       Weight_Sub_Arc(
          Get_Graph_Node_By_Tree_Nodes_Preorder(
                            Return_Graph, I+1, J+1)));
  end if;
  if Tree_Node_1_Depth <= Tree_Node_2_Depth then
   New_Graph_Edit_Arc(
       Return_Graph,
       ins,
       Get_Graph_Node_By_Tree_Nodes_Preorder(
                              Return_Graph, I,J),
       Get_Graph_Node_By_Tree_Nodes_Preorder(
                              Return_Graph, I,J+1),
       1.0);
  end if;
 end loop;
end loop;
```

In zwei ineinander verschachtelten Schleifen wird jeder IML-Knoten des ersten Baumes, kombiniert mit jedem Knoten des zweiten Baums, betrachtet. Die Variablen I und J stellen die preorder Nummer der IML-Knoten im ersten und zweiten Baum dar. Um zu entscheiden, welche Graphkante zu erzeugen ist, werden die Tiefen der IML-Knoten mit preorder Nummern $I+1$ und $J+1$ verglichen. Abhängig von dem Resultat des Vergleichs wird der Graphknoten, welcher die IML-Knoten I und J enthält, mit einem benachbarten Knoten verbunden. Ist die Tiefe des $I+1$ten IML-Knoten größer als die Tiefe des $J+1$ten IML-Knoten, so wird eine horizontale Kante zum benachbarten Graphknoten, mit IML-Knoten $I+1$ und J als Inhalt, erzeugt. Ist die Tiefe dieser Knoten gleich, so wird eine diagonale Kante zum benachbarten Graphknoten, mit IML-Knoten $I+1$ und $J+1$ als Inhalt, erzeugt. Im letzten Fall ist die Tiefe des $I+1$ten IML-Knoten kleiner als die Tiefe des $J+1$ten IML-Knoten. Hier wird eine vertikale Kante zum benachbarten Graphknoten, mit IML-Knoten I und $J+1$ als Inhalt, erzeugt. Die horizontalen und vertikalen Kanten haben Gewicht von **1.0**, da es sich um Einfüge- und Löschkanten handelt. Das Gewicht der diagonalen Kanten werden, wie bereits beschrieben, durch die Funktion **Weight_Sub_Arc** bestimmt (siehe 4.9).

4.3.2.3 Gewichtung der Substitutionskanten

Wie bereits erwähnt, wird die Funktion **Weight_Sub_Arc** dazu verwendet, die diagonalen Substitutionskanten zu gewichten. Hierfür werden die beiden in dem Zielgraphknoten der Kante enthaltenen Labels näher betrachtet.
Eine Substitutionskante hat die Form: $(vw, v_{i+1}w_{i+1})$ und repräsentiert die Ersetzung des Knoten v_{i+1} aus dem ersten Baum durch Knoten w_{i+1} aus dem zweiten Baum. Es handelt sich hierbei um IML-Knoten, die bestimmte Strukturen des Quellcodes modellieren. Diese dürfen jedoch nicht immer substituiert werden (siehe Abschnitt 3.3.4). Die beiden zu betrachtenden Knoten und deren Beschreibung in Form von Labels (primäres Label und sekundäres Label) sind im Zielgraphknoten einer Substitutionskante gespeichert. Die Entscheidung über die Substituierbarkeit zweier unterschiedlichen IML-Klassen wird mit Hilfe des Moduls **„iml_class_tag_comparator“** und seiner Funktion **Substitutable** getroffen. Dieses Modul wird im Abschnitt 4.3.4 beschrieben. Die Ungleichungen aus dem Abschnit 3.3.4 müssen nun wie folgt präzisiert werden:

- $w_1 = weight(vw, v_{i+1}w_{i+1}) < 1.0$ gdw.
  $Primary\_Label\_Of\_w_{i+1} \neq Primary\_Label\_Of\_v_{i+1}$ und
  $Substitutable(Primary\_Label\_Of\_2, Primary\_Label\_Of\_1) = TRUE$

- $w_1' = weight(vw, v_{i+1}w_{i+1}) > weight((v_iw_j, v_{i+1}w_j)) + weight((v_iw_j, v_iw_{j+1}))$ gdw.
  $Primary\_Label\_Of\_w_{i+1} \neq Primary\_Label\_Of\_v_{i+1}$ und
  $Substitutable(Primary\_Label\_Of\_2, Primary\_Label\_Of\_1) = FALSE$

- $w_2 = weight(vw, v_{i+1}w_{i+1}) < w_1$ gdw.
  $Primary\_Label\_Of\_w_{i+1} = Primary\_Label\_Of\_v_{i+1}$ und
  $Secondary\_Label\_Of\_w_{i+1} \neq Secondary\_Label\_Of\_v_{i+1}$

- $w_3 = weight(vw, v_{i+1}w_{i+1}) < w_2$ gdw.
  $Primary\_Label\_Of\_w_{i+1} = Primary\_Label\_Of\_v_{i+1}$ und
  $Secondary\_Label\_Of\_w_{i+1} = Secondary\_Label\_Of\_v_{i+1}$

Der Zielgraphknoten einer Substitutionskante dient als Eingabe für die Gewichtungsfunktion. Listing 4.9 zeigt wie ein Gewicht anhand der Labels berechnet wird.

Listing 4.9: Gewichtung der Substitutionskanten

```
Targets_Node_Pair := Target_Graph_Node.Value;
 if (Targets_Node_Pair.Primary_Label_Of_2 /=
     Targets_Node_Pair.Primary_Label_Of_1) then
   if IML_Class_Tag_Comparator.Substitutable(
          Targets_Node_Pair.Primary_Label_Of_2,
          Targets_Node_Pair.Primary_Label_Of_1) then
     Weight:=0.9;
   else
     Weight:=Float(10*(Node_Subtree_Length +
                       Sub_Node_Subtree_Length));
   end if;
 elsif (Targets_Node_Pair.Primary_Label_Of_2 =
        Targets_Node_Pair.Primary_Label_Of_1)
   and (Targets_Node_Pair.Secondary_Label_Of_2 =
        Targets_Node_Pair.Secondary_Label_Of_1) then
     Weight:=0.1;
 elsif (Targets_Node_Pair.Primary_Label_Of_2 =
        Targets_Node_Pair.Primary_Label_Of_1)
   and (Targets_Node_Pair.Secondary_Label_Of_2 /=
        Targets_Node_Pair.Secondary_Label_Of_1) then
     Weight:=0.5;
 end if;
return Weight;
```

Als ein Orientierungswert für das Gewicht der Substitutionskante dienen das Gewicht der Lösch- und Einfügekanten. Dieser ist immer konstant und liegt bei 1.0. Die beiden *Primary_Label* Felder des Graphknoten enthalten Klassennamen der IML-Knoten. Die beiden *Secondary_Label* Felder der Graphknoten enthalten Beschreibung der IML-Knoten (siehe Beschreibung des *Node_Pair* Konzeptes in 4.3.2.1). Es gibt insgesamt also mehrere Fälle wie ein Gewicht der Substitution ausfallen kann.

- **Fall1**
  $(Primary\_Label\_Of\_2 = Primary\_Label\_Of\_1) \wedge$
  $(Secondary\_Label\_Of\_2 = Secondary\_Label\_Of\_1)$.
  Dies bedeutet, dass die durch IML-Knoten modellierten Konstrukte absolut gleich sind. Die Tatsache der absoluten Gleichheit zweier IML-Knoten deutet darauf hin, dass es sich bei den abgebildeten Quelltextstrukturen um Typ1 Klone handelt. Diese Stelle des Coderagmentes wurde nach dem Einfügen nicht angepasst. Eine solche Substitution ist sehr günstig und wird mit **0.1** gewichtet.

- **Fall2**
  $(Primary\_Label\_Of\_2 = Primary\_Label\_Of\_1) \wedge$
  $(Secondary\_Label\_Of\_2 \neq Secondary\_Label\_Of\_1)$.
  Dies bedeutet, dass die durch IML-Knoten modellierten Konstrukte zwar in der selben Klasse liegen, sich jedoch leicht unterscheiden. Zwei Variablen werden zum Beispiel durch eine *Entity_L_Value* IML-Klasse dargestellt. Der sekundäre Label würde die Bezeichner der Variablen beinhalten. Es handelt sich in diesem Beispiel also um zwei Variablen mit unterschiedlichen Bezeichnern. Die Tatsache der starken Ähnlichkeit zweier IML-Knoten deutet darauf hin, dass es sich bei den abgebildeten Quelltextstrukturen um Typ2 Klone handelt.

An dieser Stelle des Coderagmentes wurden nach dem Einfügen lediglich die Bezeichner verändert. Eine solche Substitution ist ebenfalls günstig und wird mit **0.5** gewichtet.

- **Fall3**
  $Primary\_Label\_Of\_2 \neq Primary\_Label\_Of\_1$.
  Dies bedeutet, dass es sich um zwei unterschiedliche Strukturen handelt, die unter Umständen nicht substituierbar sind. Ein Beispiel hierfür wäre ein $If\_Statement$ und C_For_Loop, die eine „if"-Anweisung und eine „for"-Schleife modellieren. Offensichtlich unterscheiden sich beide Konstrukte bezüglich der Sprachsemantik stark voneinander und dürfen nicht ohne Weiteres substituiert werden. Es gibt aber auch Strukturen, die zwar durch unterschiedliche IML-Klassen modelliert werden, sich jedoch ähnlich sind. Die binären Operatoren $<$ und $>$ werden durch IML-Klassen $Less\_Than$ und $Greater\_Than$ modelliert. Diese sind, von der Semantik her, sehr ähnlich zueinander. Obwohl die Klassenbezeichner unterschiedlich sind, können diese IML-Knoten als substituierbar betrachtet werden. Somit müssen alle IML-Klassen in einem separaten Schritt auf ihre Substituierbarkeit untersucht werden, bevor ein Gewicht berechnet werden kann. Zeigt die Untersuchung, dass die Klassen substituierbar sind, so wird das Gewicht auf **0.9** gesetzt. Ist die Substitution nicht möglich, so wird das Gewicht auf $10 * (N1 + N2)$ gesetzt, wobei $N1$ und $N2$ die Längen der Bäume unter dem ersten und dem zweiten IML-Knoten sind. Durch die Multiplikation mit 10 wird sichergestellt, dass eine solche Substitution aus der Editiersequenz der Klone ausgeschlossen wird. Es wird somit in diesem Fall günstiger, den gesamten ersten Baum zu löschen und den gesamten zweiten Baum einzufügen.

Die in der Implementierung gewählten Gewichte erfüllen zwar die oben beschriebenen Ungleichungen, sind jedoch beliebig gewählt. Eine falsche Wahl der Gewichte würde eine suboptimale Editiersequenz liefern. Somit muss bei der Evaluation des Systems besonders auf die Optimalität der Editiersequenz geachtet werden.

4.3.2.4 Berechnung der Transformation

Um eine Transformation zwischen zwei IML-Bäumen zu berechnen wird zuerst, wie oben beschrieben, ein Editiergraph aufgebaut. In diesem Graph gilt es einen kürzesten Weg zwischen dem ersten und dem letzten Graphknoten zu finden. Der in 2.3.2.3 vorgestellte Algorithmus zum Finden kürzester Wege kommt an dieser Stelle zum Einsatz. Dieser wird durch die Funktion **Shortest_Paths** mit Hilfe des Pakets $Priority\_Queues\_Unbounded$ implementiert. Als Eingabe werden der Startknoten und der gesamte Graph übergeben. Diese Funkion berechnet die Distanz jedes einzelnen Graphknoten zu dem Startknoten. Pro Knoten wird auch das $Shortest\_Path\_Predecessor\_ID$ Attribut gesetzt. Nach der Annotation der Graphknoten muss der eigentliche Pfad extrahiert werden. Diese Aufgabe wird von der Funktion **Extract_Shortest_Path_From_First_To_Last** übernommen. Listing 4.10 zeigt die Extraktion eines kürzesten Weges in einem Editiergraph.

Listing 4.10: Extraktion des kürzesten Pfades aus dem annotierten Graph

```
Path : Graph_Arcs.List :=
                Graph_Arcs.Create;
First_Node : Graph_Node :=
                My_Graph.First_Node;
Last_Node: constant Graph_Node :=
                My_Graph.Last_Node;
```

```
    Current_Node : Graph_Node;
    Predecessor_ID : Natural;
    Path_Arc : Graph_Edit_Arc;
begin
   Shortest_Paths(My_Graph, First_Node);
   Path_Arc :=
     Get_Arc(My_Graph,
             Last_Node.Shortest_Path_Predecessor_ID,
             Last_Node.ID);
   Current_Node := Last_Node;
   Predecessor_ID :=
     Current_Node.Shortest_Path_Predecessor_ID;
   while Predecessor_ID >= 0 and
         Current_Node /= My_Graph.First_Node
   loop
     Path_Arc :=
            Get_Arc(My_Graph, Predecessor_ID,
                    Current_Node.ID);
     Graph_Arcs.Attach(Path_Arc,Path);
     Current_Node := Path_Arc.Source;
     Predecessor_ID :=
        Current_Node.Shortest_Path_Predecessor_ID;
   end loop;
 return Path;
```

Nach der Annotation des Graphen durch die **Shortest_Paths** Funktion wird der Pfad extrahiert. Ausgehend von dem letzten Graphnoten werden die Kanten zu den Graphknoten mit der *Shortest_Path_Predecessor_ID* verfolgt und in einer Liste gespeichert. Nach dem erreichen des ersten Graphknoten ist die Extraktion des Pfades abgeschlossen und die Liste wird zurückgegeben. Um eine der Definition 3.4 nach valide Transformation zu erzeugen, müssen die Kanten am Anfang der Rückgabeliste angefügt werden, um richtige Reihenfolge zu erhalten.

4.3.2.5 Postprocessing

Der extrahierte Pfad stellt eine Transformation zwischen abstrakten Syntax Bäumen der beiden Codefragmente eines Klonpaares dar. Er besteht aus Graphkanten, die jedoch nun redundante Information beinhalten. Eine Graphkante enthält zwei Graphknoten, die wiederum jeweils zwei *Node_Pair* Objekte enthalten. Die durch die Graphkante repräsentierte Operation bezieht sich nur auf die IML-Knoten im *Node_Pair* des Zielgraphknoten. Lediglich bei den Graphkanten, welche die Einfügeoperationen darstellen, ist auch die Quelle der Kante interessant, jedoch auch nur der erste IML-Knoten des *Node_Pair* Objektes der Quellgraphknoten. Die für die Berechnung der Transformation benötigten Information wie preorder Nummer, Tiefe der IML-Knoten, beide Labels der Knoten, Distanzattribut der Graphknoten und andere sind nach der Berechnung nicht länger von Bedeutung. Deshalb werden zwei neue Konzepte *Simple_Op* und *Edit_Sequence* eingeführt, welche nur die für das Postprocessing und die Ausgabe notwendige Daten enthalten. Das Listing 4.11 zeigt wie die Struktur dieser Konzepte aussieht.

Listing 4.11: Definition des *Simple_Op* Konzeptes

```
type Simple_Op_Rec is record
  Operation: Ada.Strings.Unbounded.Unbounded_String;
  Costs: Float;
  Parent_Node: IML_Roots.IML_Root;
  Node : IML_Roots.IML_Root;
  Sub_Node : IML_Roots.IML_Root;
  Mark: Natural;
end record;
type Simple_Op is access all Simple_Op_Rec;

package Edit_Sequence is new Lists (Simple_Op);
```

Ein *Simple_Op* Objekt repräsentiert eine elementare Editieroperation auf zwei IML-Knoten. Es werden die drei definierten elementaren Operationen durch nur ein Konzept abgebildet. Das Feld *Operation* speichert den Bezeichner der Operation in Form eines Strings („ins“, „del“ oder „sub“). Das Attribut *Costs* beinhaltet die Kosten einer Editieroperation. Das Gewicht einer Graphkante repräsentiert die Kosten der durch die Kante abgebildeten Operation. Die Kosten betragen somit 1.0 bei Einfüge- und Löschoperationen. Bei einer Substitution sind die Kosten unterschiedlich (siehe 4.9). Das Attribut *Parent_Node* speichert ein IML-Knoten des ersten Baumes, der nur bei der Einfügeoperation gesetzt wird. Dieser fungiert als ein Elternknoten beim Einfügen des IML-Knoten aus dem zweiten Baum. Bei dem Modellieren einer Lösch- oder Substitutionsoperation wird dieses Attribut auf *null* gesetzt.

Muss ein IML-Knoten gelöscht, eingefügt oder durch einen anderen ersetzt werden, so wird dieser IML-Knoten im Feld *Node* referenziert. Dieser beinhaltet also, bei allen drei möglichen Editieroperationen, einen Wert. Das Feld *Sub_Node* speichert einen IML-Knoten aus dem zweiten Baum, der einen IML-Knoten aus dem ersten Baum ersetzen soll. Beim modellieren der Einfüge- und Löschoperation wird dieses Attribut auf *null* gesetzt. Das Attribut *Mark* vom Typ *Natural* ist eine Markierung. Diese wird erst beim späteren Postprocessing relevant. Sie wird später dazu verwendet, Operationen auf „Nichtblattknoten“ auszuzeichnen. Eine *Edit_Sequence* ist somit eine Liste von *Simple_Op* Objekten.

Die Umwandlung eines Pfades in eine Editiersequenz wird mit Hilfe der Funktion **Arc_List_To_Edit_Sequence** durchgeführt. Diese Funktion bekommt als Eingabe eine Liste von *Graph_Edit_Arc* Objekten und gibt eine Liste von *Simple_Op* Objekten zurück. Listing 4.12 zeigt einen wesentlichen Ausschnitt dieser Funktion. Die Liste der Graphkanten wird sequenziell durchlaufen. Für jede Kante wird ein neues *Simple_Op* Objekt erzeugt und am Ende einer Rückgabeliste eingehängt.

Listing 4.12: Umwandlung eines Pfades in eine Sequenz von Operationen

```
if Current_Arc.Operation = ins then
   New_Op := new Simple_Op_Rec'(
                ins , Current_Arc.Costs,
                IML_Roots.IML_Root(Sources_Node_Pair.Node_1),
                IML_Roots.IML_Root(Targets_Node_Pair.Node_2),
                null,0);
   Edit_Sequence.Attach(Result_Edit_Sequence,New_Op);
end if;
if Current_Arc.Operation = del  then
   New_Op := new Simple_Op_Rec'(
                del, Current_Arc.Costs,
                null,
                IML_Roots.IML_Root(Targets_Node_Pair.Node_1),
                null,0);
   Edit_Sequence.Attach(Result_Edit_Sequence,New_Op);
end if;
if Current_Arc.Operation = sub then
   New_Op := new Simple_Op_Rec'(
                sub, Current_Arc.Costs,
                null,
                IML_Roots.IML_Root(Targets_Node_Pair.Node_1),
                IML_Roots.IML_Root(Targets_Node_Pair.Node_2),0);
   Edit_Sequence.Attach(Result_Edit_Sequence,New_Op);
end if;
```

Um aus den Informationen einer Graphkante eine *Simple_Op* zu erzeugen, müssen zunächst die *Node_Pair* Objekte des Quell und Zielgraphknoten, sowie das *Operation* und *Costs* Attribut gelesen werden. Werte der *Operation* und *Costs* Attribute der Graphkante können sofort durch gleichnamige Attribute im *Simple_Op* übernommen werden.
Die Variablen *Sources_Node_Pair* und *Targets_Node_Pair* bekommen die *Node_Pair* Objekte des Quellknoten und des Zielknoten einer Graphkante zugewiesen. Die darin enthaltenen IML-Knoten werden für das Erzeugen des *Simple_Op* Objektes verwendet. Handelt es sich um eine vertikale Graphkante, die ein „ins“ im Feld *Operation* trägt, wird ein neues *Simple_Op* Objekt erzeugt. Das Attribut *Parent_Node* übernimmt den Wert des Attributes *Node_1* von dem *Sources_Node_Pair*. Das Attribut *Node* übernimmt den Wert des Attributes *Node_2* von dem *Target_Node_Pair*. Das Attribut *Sub_Node* wird auf *null* gesetzt. Handelt es sich um eine horizontale Graphkante, die ein „del“ im Feld *Operation* trägt, wird ein neues *Simple_Op* Objekt erzeugt. Die Attribute *Parent_Node* und *Sub_Node* werden auf *null* gesetzt. Das Attribut *Node* übernimmt den Wert des Attributes *Node_1* von dem *Target_Node_Pair*. Trifft man auf eine diagonale Graphkante, die ein „sub“ im Feld *Operation* trägt, wird ein neues *Simple_Op* Objekt erzeugt. Das Attribut *Parent_Node* wird auf *null* gesetzt. Die Attribute *Node* und *Sub_Node* bekommen Werte der Attribute *Node_1* und *Node_2* von dem *Target_Node_Pair*. Bei allen drei möglichen Kombinationen der Werte in einem *Simple_Op* Objekt wird dem Attribut Mark bei der Erzeugung der Wert 0 zugewiesen. Die Umwandlung eines Pfades in eine Editiersequenz erlaubt eine vollständige Abstraktion von der eigentlichen Berechnung. Die Editiersequenz ist eine genaue Abbildung des Pfades bezüglich der Reihenfolge der Graphkanten, enthält nur die notwendigen Daten und stellt eine Grundlage für die weiteren Schritte dar.

Wie bereits im Abschnitt 3.3.5 beschrieben, kann eine Editiersequenz bezüglich der Anzahl der enthaltenen Operationen recht groß ausfallen. An einem kurzen Beispiel wird erläutert weshalb ein Postprocessingschritt nach der Berechnung der Transformation notwendig ist. Eines der zahlreichen Klone, an denen die Implementierung getestet wurde, ist in der Abbildung 4.5 dargestellt. Dieser wurde vom Klonerkennungstool „ccdiml“ im System „Ant“ von Eclipse gefunden. Die Berechnung der Transformation zwischen abstrakten Syntax Bäumen der beiden Codefragmente lieferte eine Editiersequenz, die aus 32 Editieroperationen bestand. Diese Editiersequenz behandelt jeden einzelnen IML-Knoten der Bäume und würde den Benutzer mit Informationen überlasten. Viele dieser Operationen sind für den Benutzer von geringer Bedeutung.

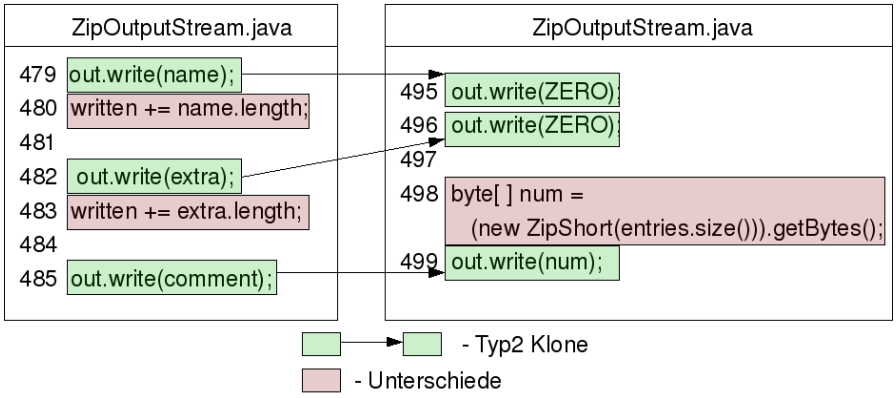

Abbildung 4.5: Ein Typ3 Klonpaar aus dem System „Eclipse-Ant“

Betrachtet man die beiden Codefragmente genau, so erkennt man, dass der erste Codefragment mit Leichtigkeit in den zweiten Codefragment transformiert werden kann. Löscht man die Zeilen 480 und 483 aus dem ersten Fragment und fügt die Zeile 498 aus dem zweiten Fragment in den ersten ein, so erhält man ein Typ2 Klonpaar.
Die Codefragmente würden sich somit nur durch die Bezeichner der Variablen unterscheiden. Erwartungsgemäß sind also nur 3 statt 32 Operationen für die Transformation der abgebildeten Codefragmente des Klonpaares notwendig. In dieser Arbeit ist das Ziel des Postprocessing, die berechnete Editiersequenz bezüglich der Anzahl der Operationen einzuschränken.

Das Postprocessing besteht aus zwei wesentlichen Schritten. Im ersten Schritt wird nach bestimmten Mustern in der Sequenz gesucht, um wichtige Operationen zu markieren. Die Markierung kann durch das Setzen des Attributes *Mark* in einem *Simple_Op* Objekt erfolgen. Die für den Benutzer besonders interessante Operationen werden mit einer 3 markiert, während uninteressante Operationen mit der Zahl 0 markiert werden. Es sind insgesamt fünf Fälle zu unterscheiden.

- **Fall 1** Substitution von Knoten mit gleichen IML-Klassen.
  Knoten mit gleichen IML-Klassen modellieren, bis auf die Bezeichner, gleiche Konstrukte aus dem Quellcode. Solche Strukturen im Quellcode deuten darauf hin, dass es sich bei diesen Konstrukten um Typ1 oder Typ2 Klone handelt. Die Operationen der Ersetzung von Strukturen durch gleiche oder sehr ähnliche Strukturen stellt die Gemeinsamkeiten der beiden Codefragmente eines Klonpaares dar. Solche Operationen sind uninteressant und werden mit der Zahl 0 markiert. In der Abbildung 4.6 sind stark vereinfachte abstrakte Syntaxbäume zweier „if“-Anweisungen dargestellt.

Die grün markierten Knoten sind typische Repräsentanten solcher uninteressanten Substitutionen vor allem im Bereich der Blattknoten. Die rot gekennzeichneten Knoten werden für den zweiten Fall bedeutsam.

- **Fall 1.2** Substitution von Nichtblattknoten.
  Im Fall 1 werden alle Operationen zur Substitution von IML-Knoten mit gleichen IML-Klassen als unwichtig angesehen. Jedoch könnten einige davon von Interesse sein. Substitution von Nichtblattknoten bedeutet, dass Konzepte gleich sind. Ein Beispiel hierfür wären zwei Zuweisungen eines Wertes an eine Variable. Interessant ist aber auch wie der zugewiesene Wert entsteht. Werden die Werte unterschiedlich berechnet, so muss die rechte Seite der ersten Zuweisung an die rechte Seite der zweiten Zuweisung angepasst werden. Diese kann weitere interessante Operationen enthalten, welche die vorgehende Substitution ebenfalls interessant machen könnten. Solche Substitution werden mit der Zahl 3 markiert, jedoch wurde bisher keine geeignete Verwendung für sie gefunden. Die Substitution von „Nichtblatt-IML-Knoten“ mit gleichen IML-Klassen wird trotz der Markierung nicht weiterverwendet.

- **Fall 2** Substitution von Nichtblattknoten mit unterschiedlichen IML-Klassen.
  Eine solche Substitution kann nur dann in der Editiersequenz vorkommen, wenn die beiden von der Operation betroffenen IML-Knoten semantisch ähnliche Strukturen im Quellcode darstellen. Nur in solchen Fällen wird die entsprechende Graphkante so gewichtet, dass ein Weg durch diese Kante in einem Editiergraph günstig ist. Dies bedeutet, dass an dieser Stelle im Quellcode eine semantische Änderung vorgenommen wurde. Solche Operationen sind besonders interessant und werden mit der Zahl 3 markiert. Ein Beispiel hierfür wären die binären logischen Operatoren $>$ und $<$ in den Bedingungen zweier „if“-Anweisungen $if(a > c)$ und $if(a < c)$. Die Abbildung 4.6 zeigt stark vereinfachte abstrakte Syntax Bäume der beiden Anweisungen. Man sieht, dass die Knoten, welche die logischen Operatoren modellieren (rot markiert), sich an gleichen Position in den ASTs befinden und sprachsemantisch sehr ähnlich zueinander sind. In einem solchen Fall ist die Substitution möglich und sinnvoll.

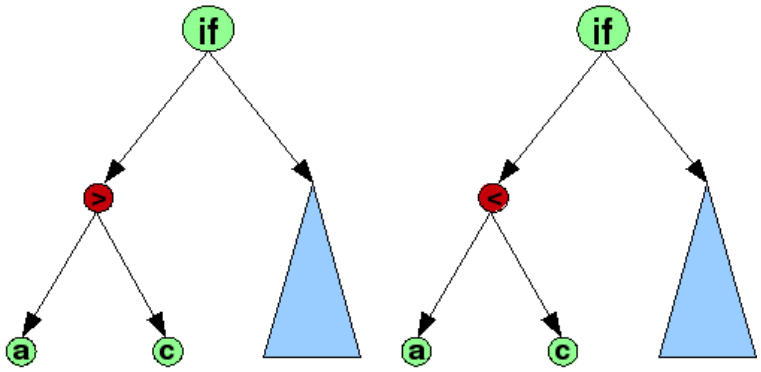

Abbildung 4.6: Beispiel ASTs zweier strukturell ähnlicher „if“-Anweisungen mit semantischen Unterschieden

- **Fall 3** Löschen oder Einfügen von Nichtblattknoten.
  Das Löschen oder Einfügen eines Nichtblattknoten hat zur Folge, dass alle Knoten, die sich unter diesem befinden, ebenfalls gelöscht oder engefügt werden müssen. Es ist somit nicht notwendig alle nachfolgenden Operationen zu betrachten.

In diesem Fall werden die Einfüge- und Löschoperationen auf Unterbäume eingeführt. Alle nachfolgenden Operationen, die das Löschen oder Einfügen der Knoten im Unterbaum darstellen, werden nicht betrachtet. Eine solche Operation wird mit der Zahl 3 markiert. Eine mit 3 markierte Einfüge- oder Löschoperation bedeutet, bezogen auf die Unterschiede zweier Codefragmente, dass ganze Ausdrücke eingefügt oder gelöscht werden. In der Abbildung 4.7 sind zwei Anweisungssequenzen mit deren ASTs dargestellt. Die Editiersequenz würde neben der Substitution der grün markierten Knoten (Fall 1) auch das Löschen der fünf rot markierten Knoten beinhalten. In diesem Fall wird nur die Operation zum Entfernen des rot markierten Zuweisungsknoten in die postprocessierte Sequenz aufgenommen. Diese Operation kann als Löschen der gesamten zweiten Zeile der ersten Anweisungssequenz interpretiert werden. Die Operationen zum Löschen der übrigen rot markierten Knoten sind somit nicht mehr relevant.

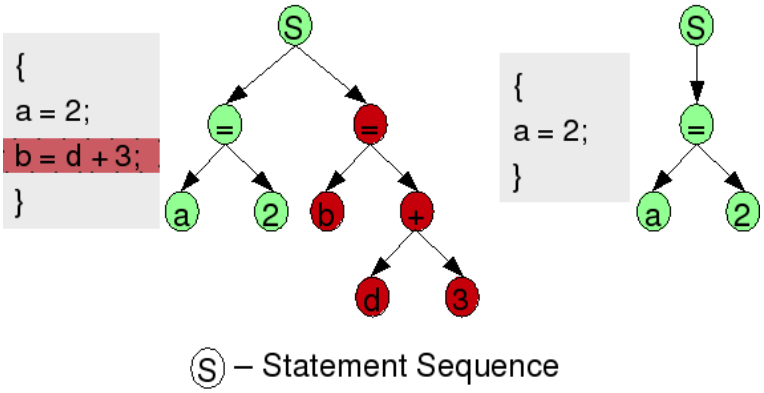

Abbildung 4.7: Beispiel ASTs. Löschen eines Unterbaums.

- **Fall 4** Löschen oder Einfügen von Blattknoten.
  Diese beiden Operationen sind nach wie vor die grundlegenden Operationen einer Editiersequenz. Viele Operationen zum Löschen oder Einfügen von Blattknoten werden nicht betrachtet, da sie meist zu ganzen Unterbäumen gehören, die gelöscht oder eingefügt werden. Es kann jedoch nicht ausgeschlossen werden, dass auch einzelne Blattknoten eingefügt oder gelöscht werden können. Kommt eine solche Operation vor, wird sie mit der Zahl 0 markiert. Diese Operationen sind durchaus wichtig, jedoch muss zwischen dem Einfügen bzw. Löschen von Nichtblattknoten und einfachen Blattknoten unterschieden werden.

Der erste Schritt des Postprocessing, der die oben beschriebene Fallunterscheidung und Markierung der Operationen übernimmt, wird in der Funktion **Aggregate_Ops** durchgeführt. Als Eingabe wird eine Liste von *Simple_Op* Objekten übergeben, die sequentiell durchlaufen wird. Nach der Differenzierung und Markierung werden die *Simple_Op* Objekte in einer neuen Liste gespeichert, die letztendlich zurückgegeben wird. In dem Listing 4.13 wird die Behandlung der Operationen zur Substitution von Knoten dargestellt.

Listing 4.13: Postprocessing der Substitution

```
if Current_Op.Operation = sub then
  ...
  ...
  Sub_Length:=
     Iner_Sequence_Length + Iner_Sequence_Length_2;
-- sub nonleaf
    if Sub_Length>2 then
      Current_Op.Mark:=3;
      Edit_Sequence.Attach(Result_Edit_Sequence,Current_Op);
      Outer_Index := Outer_Index+1;
--sub imlclass1 != imlclass2
    elsif Current_Op.Costs>=1.0 then
      Current_Op.Mark:=3;
      Edit_Sequence.Attach(Result_Edit_Sequence,Current_Op);
      Outer_Index := Outer_Index+1;
--sub imlclass1 == imlclass2 or leaf
    else
      Current_Op.Mark:=0;
      Edit_Sequence.Attach(Result_Edit_Sequence,Current_Op);
      Outer_Index := Outer_Index+1;
    end if;
else
    Outer_Index := Outer_Index+1;
end if;
```

Um festzustellen, ob Nichtblattknoten substituiert werden, muss die Größe (Anzahl der Knoten) der Bäume mit den zu substituierenden Knoten als Wurzel berechnet werden. Handelt es sich um Blattknoten, so ist die Größe der beiden Bäume jeweils 1. Anhand der Summe beider Baumgrößen kann nun entschieden werden, ob es sich um Blatt- oder Nichtblattknoten handelt. Ist die Summe größer als 2, so handelt es sich eindeutig um Substitution von Nichtblattknoten. Um die benötigte Summe zu berechnen, werden die Bäume mit den im Attribut *Node* und *Sub_Node* des *Simple_Op* Objektes gespeicherten IML-Knoten ausgelesen und linearisiert. Bei der Linearisierung werden beide Bäume preorder traversiert.
Dabei werden die Knoten nach deren preorder Reihenfolge in einer Liste gespeichert. Die Längen dieser Listen stellen die Größe der beiden Bäume dar. Die Summe der Längen wird der Variablen *Sub_Length* zugewiesen. Die ersten beiden Zweige der „if“-Anweisung decken Fall 1.2 und Fall 2 ab. Der erste Fall wird im „else“-Zweig behandelt.

In dem Listing 4.14 wird die Behandlung der Operationen zum Einfügen und Löschen von Knoten dargestellt. In der dargestellten „if“-Anweisung werden die Fälle 3 und 4 behandelt.

Listing 4.14: Postprocessing der Einfüge und Löschoperationen

```
if Current_Op.Operation = ins or
   Current_Op.Operation = del then
 Iner_Sequence_Length :=0;
 Subtree:=
    IML_Tree_Utils.Node_List.Create;
 Subtree:=
    IML_Tree_Utils.Preorder_Traversal(Current_Op.Node);
 Iner_Sequence_Length:=
    IML_Tree_Utils.Node_List.Length(Subtree);
--ins/del nonleaf
    if Iner_Sequence_Length>1 then
       Current_Op.Mark:=3;
       Edit_Sequence.Attach(Result_Edit_Sequence,Current_Op);
       Outer_Index:=Outer_Index+Iner_Sequence_Length;
    end if;
--ins/del leaf
    if Iner_Sequence_Length=1 then
       Current_Op.Mark:=0;
       Edit_Sequence.Attach(Result_Edit_Sequence,Current_Op);
       Outer_Index := Outer_Index+1;
    end if;
end if;
```

Wenn ein IML-Knoten eingefügt oder gelöscht wird, muss festgestellt werden, ob es sich um ein Nichtblattknoten handelt. Hierzu wird, ähnlich wie bei der Markierung von Substitutionen, der Baum, mit dem zu löschenden bzw einzufügenden IML-Knoten als Wurzel, linearisiert. Anhand der Länge der Liste kann festgestellt werden, ob ein Blatt oder ein Nichtblattknoten eingefügt oder gelöscht werden muss. Übersteigt die Größe des Baums den Wert 1, so wird ein Nichtblattknoten gelöscht oder eingefügt. Gleicht die Größe des Baumes dem Wert 1, so handelt es sich um das Einfügen oder Löschen eines Blattknoten. Die Operation zum Löschen oder Einfügen eines Nichtblattknoten wird mit dem Wert 3 markiert. Die Operation zum Löschen oder Einfügen eines Blattknoten erhält die Zahl 0 als Markierung. Die Länge der IML-Knotenliste in den Fällen der Nichtblattknoten gibt auch an, wie viele nachfolgende Operationen ausgelassen werden können. Bei Operationen auf Nichtblattknoten kann der Zeiger auf die Operation, die als nächstes verarbeitet werden soll, um die gesamte Länge dieser Knotenliste verschoben werden.

Im ersten Schritt des Postprocessing wurde die Editiersequenz bezüglich der Anzahlt der Operationen eingeschränkt. Es wurde aber auch mit Hilfe der Markierung eine Unterscheidung in wichtige und unwichtige Operationen durchgeführt. Im zweiten Schritt werden nun die unwichtigen Operationen ausgefiltert. Die im ersten Schritt nachbearbeitete Editiersequenz wird ein weiteres mal sequentiell durchlaufen, um jede Operation noch einmal zu betrachten. Es wird, wie auch im ersten Schritt, eine neue Editiersequenz erzeugt. In dieser Liste werden nun alle Operationen gespeichert, die eine Relevanz haben. Im Listing 4.15 werden die Kernkomponenten der Funktion **Filter_Cheap_Ops** dargestellt. Diese Funktion führt den zweiten Schritt des Postprocessing durch.

Listing 4.15: Filterfunktion des Postprocessing

```
while Edit_Sequence.More(Iterator) loop
  Current_Op := Edit_Sequence.Current(Iterator);

--filter uninteressting subs
  if (Current_Op.Mark = 0 and
      Current_Op.Operation = sub) then
    Filtered_Counter := Filtered_Counter+1;

--inherit ins/del ops
  elsif (Current_Op.Operation = ins or
         Current_Op.Operation = del)
  then
    Edit_Sequence.Attach(Result_Sequence,Current_Op);

--filter uninteressting nonleaf subs
  elsif (Current_Op.Mark = 3 and
         Current_Op.Operation = sub) and
        (Storables.Get_External_Tag(
         Storables.Storable(Current_Op.Node).all) =
         Storables.Get_External_Tag(
         Storables.Storable(Current_Op.Sub_Node).all))
  then
     Filtered_Counter := Filtered_Counter+1;

--inherit interessting subs
  elsif (Current_Op.Mark = 3 and
         Current_Op.Operation = sub) and
        (Storables.Get_External_Tag(
         Storables.Storable(Current_Op.Node).all) /=
         Storables.Get_External_Tag(
         Storables.Storable(Current_Op.Sub_Node).all))
  then
     Edit_Sequence.Attach(Result_Sequence,Current_Op);
  end if;
 Edit_Sequence.Next(Iterator);
end loop;
```

Der erste Zweig der *if*-Anweisung der Funktion lässt alle Operationen zur Substitution aus, welche die Markierung 0 tragen (siehe Fall 1 in 4.3.2.5). Hier werden also die Typ1 bzw. Typ2 Bestandteile der Typ3 Codefragmente ausgefiltert. Die Einfüge- und Löschoperationen werden im zweiten Zweig der *if*-Anweisung behandelt. Diese Operationen deuten auf die Veränderungen der Struktur der Bäume und somit der Quelltextstrukturen. Sie sind für die Darstellung der Unterschiede zwischen Codefragmenten sehr wichtig und werden in die neue Editiersequenz übernommen (siehe Fall 3 und 4 in 4.3.2.5). Im dritten und im vierten Zweig der *if*-Anweisung werden die mit einer 3 markierten Substitutionen behandelt.
Die Operationen zur Substitution von Nichtblattknoten mit gleichen IML-Klassen werden ausgefiltert (siehe Fall 1.2 in 4.3.2.5). Die Operationen zur Substitution von IML-Knoten mit ungleichen IML-Klassen werden in die neue Editiersequenz übernommen (siehe Fall 2 in 4.3.2.5).

In der Abbildung 4.8 werden die einzelne Schritte des Postprocessing an einem Beispiel erläutert. Es handelt sich um zwei vereinfachte abstrakte Syntax Bäume zweier „if“-Anweisungen, die ein Typ3 Klonpaar bilden. Es wird dargestellt wie die berechnete Editiersequenz durch das Postprocessing verarbeitet wird. Die rot markierten Bereiche der Codefragmente sind auch in den ASTs sichtbar und stellen die Unterschiede dar. Die grün markierten Knoten in den ASTs bilden die Gemeinsamkeiten der Codefragmente ab. Nach dem Postprocessing bleiben nur die Operationen über, die durch Anwendung genau diese Unterschiede beheben würden.

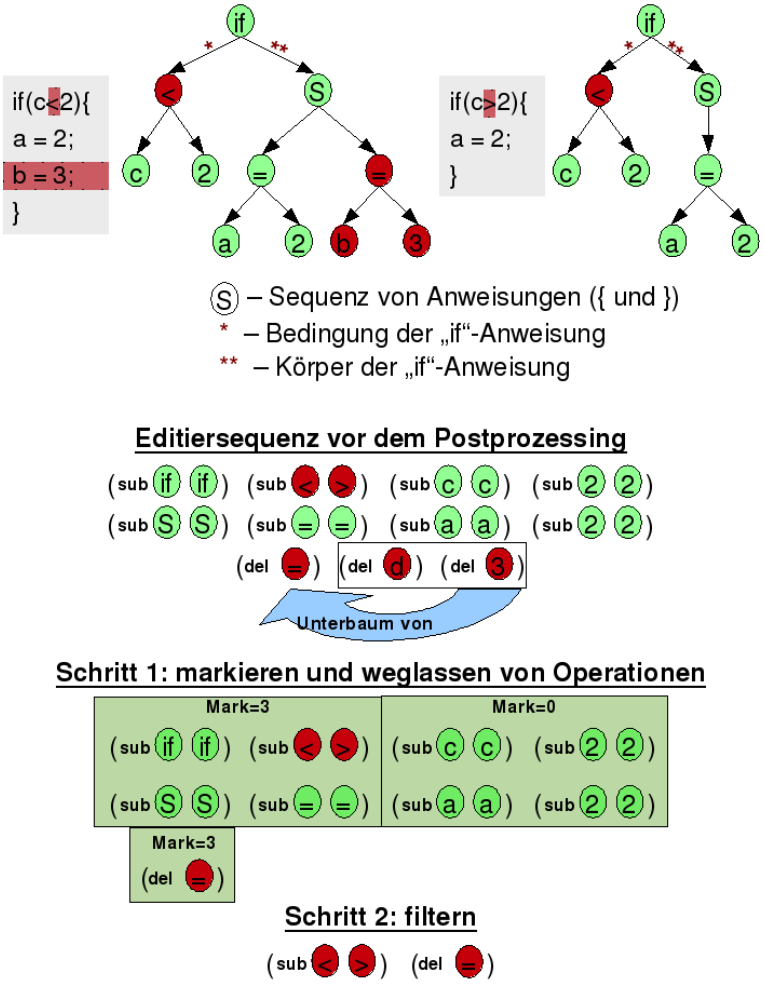

Abbildung 4.8: Einzelne Schritte des Postprocessing am Beispiel

4.3.3 „iml_tree_utils“ Modul

Das **iml_tree_utils** Modul enthält Funktionen zur Berechnung der Informationen, die für das Aufbauen eines Editiergraphen benötigt werden. Um ein Editiergraph zweier abstrakten Syntax Bäume aufzubauen, müssen die Bäume durch einer preorder Traversierung linearisiert werden. Desweiteren benötigt man zum Aufbau auch die Tiefe der einzelnen Knoten im jeweiligen Baum. Diese beiden Aufgaben werden von den Funktionen **Preorder_Traversal_Recursive**, **Preorder_Traversal** und **Get_Nodes_Depth2** dieses Moduls übernommen.

Die ersten beiden Funktionen sind für das Auslesen und Linearisieren der Bäume verantwortlich. Die dritte Funktion berechnet die Tiefe der Knoten.

Das Listing 4.16 zeigt wie der im Abschnitt 2.3.3.3 eingeführte Algorithmus zum preorder traversieren eines Baums implementiert wurde. Um den abstrakten Syntax Baum von einem IML-Knoten ausgehend auszulesen, werden die syntaktischen Kanten im IML-Graphen verfolgt. Dies geschieht mit Hilfe des im „Bauhaus“ vorhandenen Modul *IML_Roots*, welches für das Verarbeiten von IML-Knoten implementiert wurde.

Listing 4.16: Extraktion eines ASTs aus der IML und Preorder Traversierung

```
procedure Preorder_Traversal_Recursive
   (Node            : in  IML_Roots.IML_Root;
    Traversal_List  : in out Node_List.List;
    Visited         : in out Node_List.List)
is
    Children : IML_Roots.Class_Vector renames
               IML_Roots.Syntactic_Children (Node);
    Succ     : IML_Roots.IML_Root;
begin
  if Node_List.IsInList (Visited, Node) then
    return;
  else
    Node_List.Attach (Visited, Node);
  end if;
  for I in reverse Children'Range loop
    Succ := Children(I);
    Preorder_Traversal_Recursive(
                Succ, Traversal_List, Visited);
  end loop;
    Traversal_List :=
         Node_List.Attach (Node,Traversal_List);
end Preorder_Traversal_Recursive;
```

Die zu der syntaktischen Dekomposition gehörenden IML-Knoten werden nach deren preorder Reihenfolge in einer Liste gespeichert. Aus dem IML-Graphen extrahierten abstrakten Syntax Bäume stellen die Struktur eines Codefragmentes dar. Jedoch hat nicht jeder IML-Knoten einen Äquivalent im Quellcode. Manche Quellcodestrukturen werden mit Hilfe von künstlich erzeugten Knoten repräsentiert. Zum Beispiel wird durch die IML-Knoten *Begin_Of_Lifetime* und *End_Of_Lifetime* der Lebenszyklus einer Variablen modelliert. Diese Knoten tauchen überall dort auf, wo Variablen deklariert werden. Sie haben keinen Äquivalent im Quellcode und sind deshalb immer künstlich. In der Abbildung 4.9 ist ein AST einer *Java* Methode dargestellt.

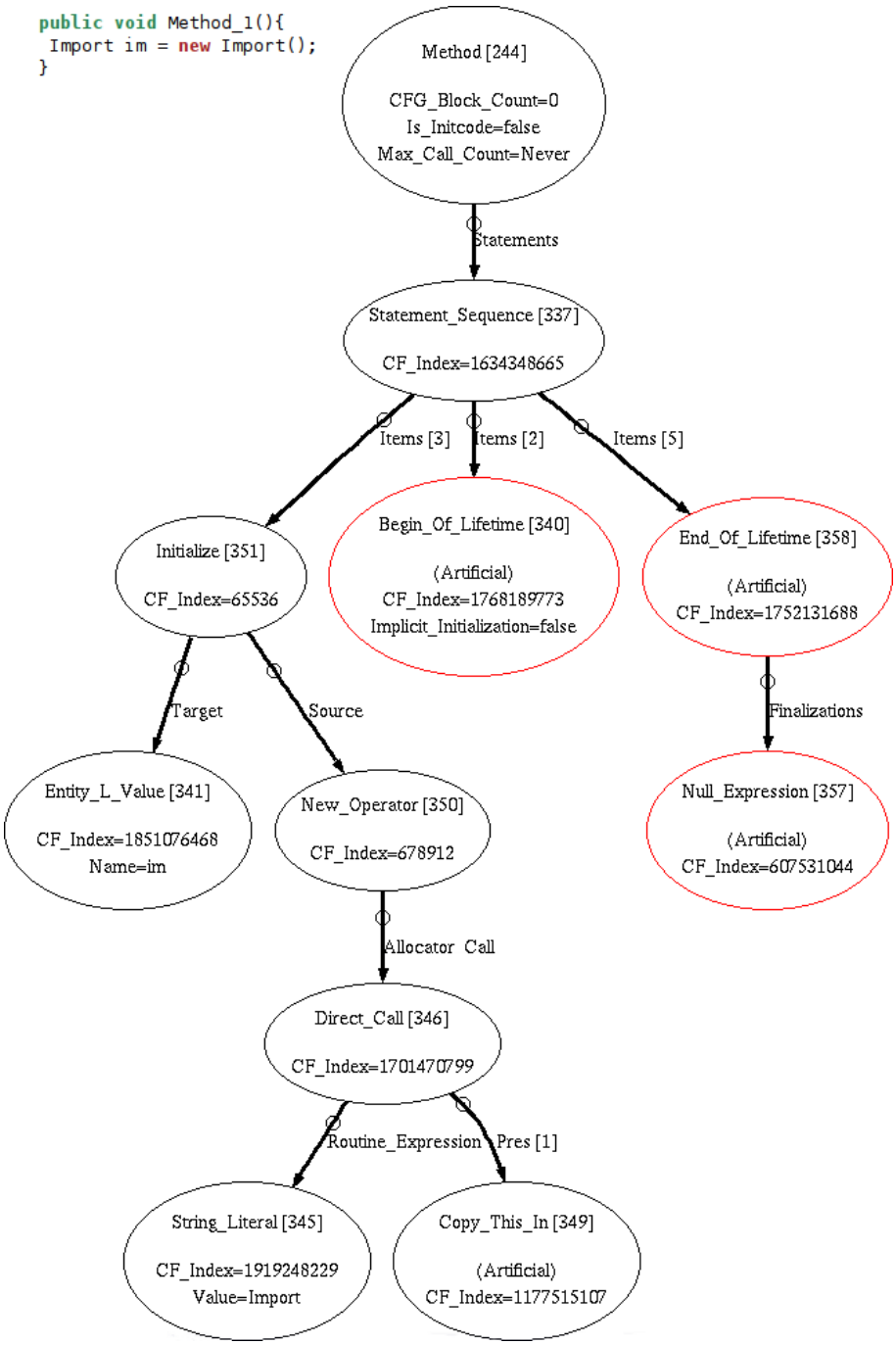

Abbildung 4.9: Aus dem IML-Graphen extrahierter AST mit künstlichen Knoten.

Der Rumpf der Methode besteht aus nur einer Zeile: *Imoprt im = new Import*();. Dabei wird mit Hilfe des *new* Operators ein neues Objekt einer Variablen zugewiesen. Es wird also eine Variable instantiiert. Dieser Baum wurde direkt aus einem IML-Graphen extrahiert und enthält viele künstlichen Knoten (rote Umrandung in der Abbildung). Diese künstlichen Knoten sind für die Berechnung der Editiersequenz unerheblich und müssen nach dem Linearisieren aus der Liste entfernt werden.

Die Funktion **Preorder_Traversal** erleichtert die Handhabung der oben beschriebenen rekursiven Funktion. Die für das rekursive Traversieren benötigte Strukturen werden initialisiert, anschließend wird die rekursive Funktion aufgerufen. Nach dem Linearisieren werden auch die künstlichen Knoten ausgefiltert. Das Listing 4.17 zeigt die Implementierung der Funktion.

Listing 4.17: Filterung der künstlichen IML-Knoten

```
function Preorder_Traversal
  (Node: in  IML_Roots.IML_Root )
return Node_List.List
is
  Traversal_List  :  Node_List.List:=Node_List.Create;
  Visited         :  Node_List.List:=Node_List.Create;
  First_Node      :  IML_Roots.IML_Root;
begin
  Preorder_Traversal_Recursive(
        Node, Traversal_List,Visited);
  Remove_Artificial(Traversal_List);
  return Traversal_List;
end Preorder_Traversal;
```

Zum Entfernen der künstlichen Knoten wird die generische Funktion **Gen_DeleteItems** des „Bauhaus“-Paketes **Lists** von der Funktion **Remove_Artificial** überschrieben. Diese verwendet die Funktion **Is_Artificial**, um zu erkennen, ob ein Knoten künstlich ist oder nicht. In jedem IML-Knoten ist das Attribut *Artificial* vom Typ „boolean“ vorhanden. Dieses gibt an, ob ein Knoten einen Äquivalent im Quellcode aufweist oder nicht. Die Funktion **Is_Artificial** liest einfach den Wert dieses Attributes aus und gibt diesen zurück.

Die letzte Funktion des Moduls ist für das Berechnen der Tiefe eines Knoten im Baum verantwortlich. Das Listing 4.18 zeigt wie die Funktion **Get_Nodes_Depth2** implementiert wird.

Listing 4.18: Berechnung der Tiefe der IML-Knoten im AST

```
function Get_Nodes_Depth2(Node: in IML_Roots.IML_Root)
return Natural
is
  Current_Node: IML_Roots.IML_Root;
  Depth: Natural := 0;
begin
  Current_Node:=Node;
  while Current_Node/= IML_Roots.No_IML_Root loop
    if not IML_Roots.Get_Artificial(Current_Node) then
       Depth:=Depth+1;
    end if;
```

```
    Current_Node := IML_Roots.Get_Parent (Current_Node);
  end loop;
return Depth;
end Get_Nodes_Depth2;
```

Um die Tiefe zu berechnen, werden von einem Eingabeknoten ausgehend die Kanten zu den Elternknoten verfolgt. Solange ein Elternknoten existiert, gibt es auch eine Kante. In einer Schleife zählt man, deshalb, solange die Kanten, bis der Wurzelknoten des Baumes erreicht wurde. Der Wert des Zählers stellt am Ende die gesuchte Tiefe des Knoten dar. Die Kanten zu den künstlichen Elternknoten werden nicht mitgezählt.

4.3.4 „iml_class_tag_comparator“ Modul

Das Modul **iml_class_tag_comparator** wird für die Gewichtung der Substitutionskanten verwendet (siehe 4.9). Steht eine Substitution von IML-Knoten an, muss zunächst entschieden werden ob, diese zulässig ist. Um diese Entscheidung treffen zu können, werden die IML-Klassen der Knoten betrachtet. Sind die Klassen verschieden, bedeutet dies, dass die abgebildeten programmiersprachliche Strukturen unterschiedlich sind. In solchen Fällen muss die Syntax und auch die Semantik dieser Strukturen näher betrachtet werden. Die Klassenhierarchie der IML beinhaltet Klassen zum Modellieren vieler möglichen Anweisungen in verschiedenen Programmiersprachen.

Allgemein besteht die IML-Klassenhierarchie aus **abstrakten** und **konkreten** Klassen. Sowohl abstrakte, als auch konkrete Klassen werden abgeleitet und können dabei um Attribute erweitert werden. Bei den Attributen wird zwischen **syntaktischen** und **semantischen** unterschieden. Falls mehrere konkrete Klassen von einer Klasse ableiten, so sind die von den Klassen dargestellte programmiersprachlicher Strukturen zu einander ähnlich. Es kommt jedoch vor, dass eine Klasse beim Ableiten durch einen syntaktischen Attribut erweitert wird. Es gilt folgendes:

- Ein abstrakter Syntax Baum besteht nur aus Knoten der konkreten Klassen.
- Konkrete Klassen mit der gleichen Menge an syntaktischen Attributen modellieren Quellcodestrukturen mit gleicher oder ähnlicher Syntax.

Darauf basiert eine einfache Überlegung, die eine Kategorisierung der Klassen erlaubt und letztendlich die Entscheidung über Substituierbarkeit von IML-Knoten ermöglicht.

Alle konkreten Klassen, die eine Oberklasse nicht durch syntaktische Attribute erweitern, können substituiert werden. Klassen, die beim Ableiten die Oberklasse um ein syntaktischen Attribut erweitern, sind von der Substitution ausgeschlossen.

In der Abbildung 4.10 sind zwei kleine Ausschnitte aus der Klassenhierarchie der IML abgebildet. Dargestellt sind zwei abstrakte Klassen, die von weiteren konkreten Klassen erweitert werden.

Die konkreten Klassen *Return_With_Value* und *Return_Without_Value* leiten von der abstrakten Klasse *Return_Statement* ab. Die konkrete Klasse *Return_With_Value* erweitert die abstrakte Klasse *Return_Statement* um einen syntaktischen Attribut, während die konkrete Klasse *Return_Without_Value* keine Erweiterungen beinhaltet.

Betrachtet man die durch *Return_With_Value* und *Return_Without_Value* repräsentierten programmiersprachlichen Konstrukte, so handelt es sich hierbei um unterschiedliche Konzepte. Beide stellen den letzten Schritt einer „Berechnung“ dar. Sie befinden sich somit in der selben Kategorie. Jedoch wird in einem Fall ein Wert mitgeliefert (Erweiterung durch das Attribut *Expression*) und im anderen Fall wird die „Berechnung“ einfach abgebrochen. Die beiden konkreten Klassen dürfen somit nicht substituiert werden.

Die zweite abstrakte IML-Klasse in der Abbildung ist *Literal*. Diese wird von konkreten Klassen *Boolean_Literal*, *Char_Literal*, *Int_Literal* und anderen erweitert. Allerdings werden bei der Erweiterung keine neuen syntaktischen Attribute eingeführt. Die repräsentierten porgrammiersprachlichen Konstrukte sind stets Literale. Die Typen dieser sind zwar unterschiedlich, jedoch handelt es sich immer um stark ähnliche Konstrukte. Sie dürfen somit substituiert werden.

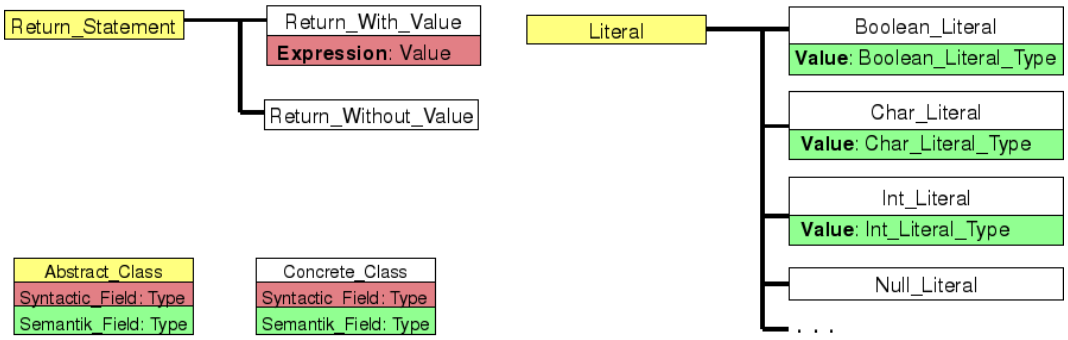

Abbildung 4.10: Ausschnitt aus der Klassenhierarchie der IML

Kategorisierung der IML-Klassen

In diesem Modul wurde ein für die Aufgabe relevanter Teil der IML-Klassenhierarchie in einer statischen Datenstruktur abgespeichert. Dabei wurden die IML-Klassen in Kategorien eingeteilt. Insgesamt wurden die Klassen in 15 Kategorien eingeteilt. Im Folgenden werden die Kategorien bzw. Unterkategorien und deren Klassen beschrieben. Dabei wird innerhalb einer Kategorie zwischen substituierbaren und nicht substituierbaren Klassen unterschieden. Klassen verschiedener Kategorien bzw. Unterkategorien dürfen nicht substituiert werden. **Einige Klassen werden gesondert behandelt** (siehe 4.3.4).

Kategorie 1 „Sequence“
Die Abbildung 4.11 zeigt die Klassen der Kategorie „Sequence“. Diese beinhaltet zwei Unterkategorien *Statement_Sequence* und *Ada_Choise*. Die konkrete Klasse *Synchronized_Sequence* der Kategorie *Sequence* ist von der Substitution durch andere konkreten Klassen dieser Kategorie ausgeschlossen, da diese die Oberklasse durch ein syntaktischen Attribut erweitert. Die Klasse *Tryall_Finally* der Unterkategorie *Statement_Sequence* ist mit der selben Begründung ebenfalls von der Substitution durch andere konkreten Klassen dieser Unterkategorie ausgeschlossen. Die restlichen Klassen der Kategorie *Sequence*, bis auf die Klasse *Postfix_Operator* (**siehe Sonderbehandlung** 4.3.4), sowie alle Klassen in der Unterkategorie *Ada_Choice* dürfen gegeneinander substituiert werden.

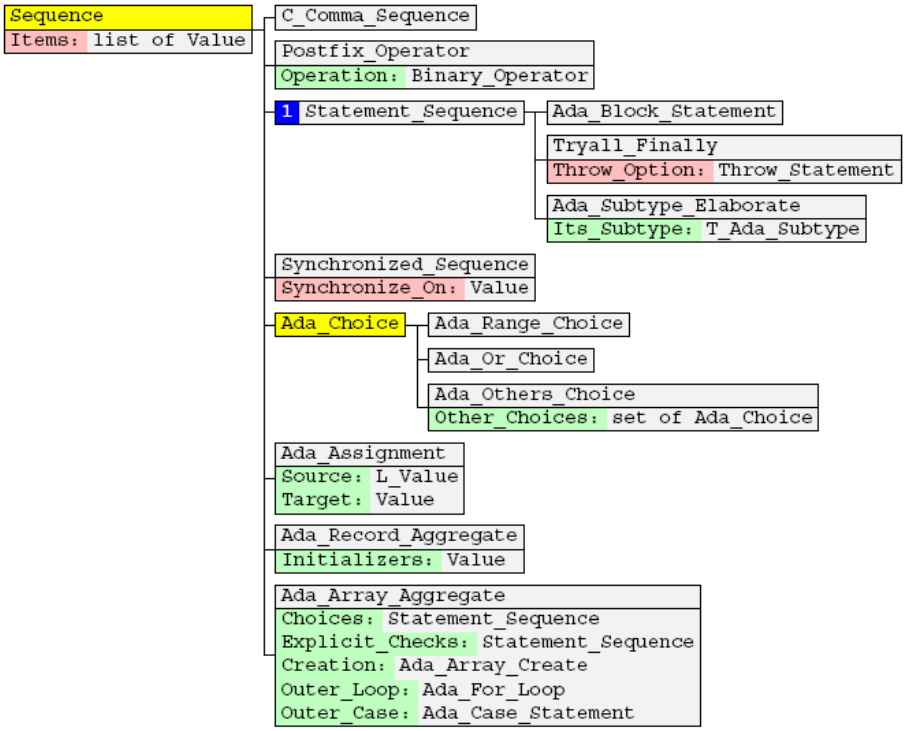

Abbildung 4.11: Klassen der Kategorie Sequence

Kategorie 2 „Literal“

Die Abbildung 4.12 zeigt die Klassen der Kategorie „Literal“. Diese Kategorie enthält keine Unterkategorien. Keine der konkreten Klassen dieser Kategorie erweitert die Oberklasse um ein syntaktischen Attribut. Somit dürfen hier alle Klassen gegeneinander substituiert werden. Die Klassen dieser Kategorie sind zusätzlich ein **Teil einer Sonderbehandlung** (siehe 4.3.4).

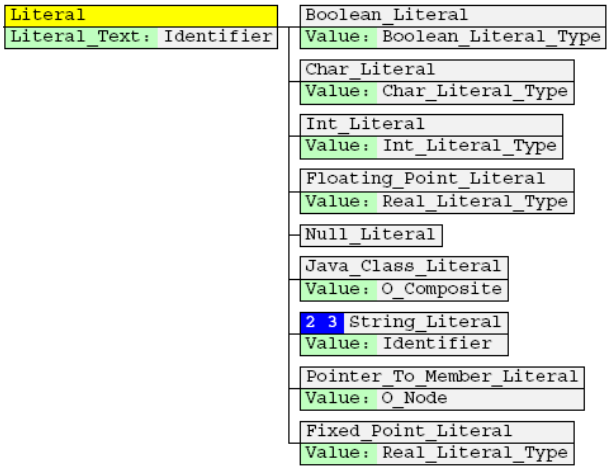

Abbildung 4.12: Klassen der Kategorie Literal

Kategorie 3 „Common_Subexpression“
Die Abbildung 4.13 zeigt die Klassen der Kategorie „Common_Subexpression“. Diese Kategorie enthält keine Unterkategorien. Keine der konkreten Klassen dieser Kategorie erweitert die Oberklasse um ein syntaktischen Attribut.
Somit dürfen hier alle Klassen gegeneinander substituiert werden.

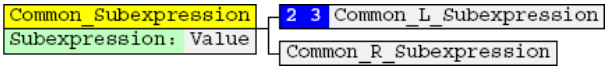

Abbildung 4.13: Klassen der Kategorie Common_Subexpression

Kategorie 4 „Unary_LR_Operator“
Die Abbildung 4.14 zeigt die Klassen der Kategorie „Unary_LR_Operator“. Diese Kategorie enthält eine Unterkategorie *LR_Conversion*. Keine der abgeleiteten konkreten Klassen in der Kategorie selbst, sowie in der Unterkategorie, enthält eine syntaktische Erweiterung. Die Substitution ist somit, innerhalb der Kategorie und innerhalb der Unterkategorie, erlaubt.

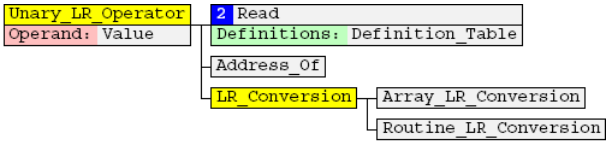

Abbildung 4.14: Klassen der Kategorie Unary_LR_Operator

Kategorie 5 „Pre_Call_Operation“
Die Abbildung 4.15 zeigt die Klassen der Kategorie „Pre_Call_Operation“. Keine der konkreten Klassen dieser Kategorie erweitert die Oberklasse um ein syntaktischen Attribut. Somit dürfen hier alle Klassen gegeneinander substituiert werden.

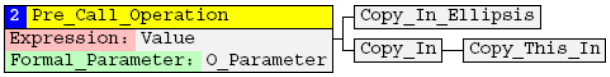

Abbildung 4.15: Klassen der Kategorie Pre_Call_Operation

Kategorie 6 „Unconditional_Branch“
Die Abbildung 4.16 zeigt die Klassen der Kategorie „Pre_Call_Operation“. Diese Kategorie ist in vier folgende Unterkategorien aufgeteilt: *Throw_Statement*, *Exit_Statement*, *Goto_Statement* und *Return_Statement*. Die konkreten Klassen innerhalb der ersten drei Kategorien dürfen substituiert werden, da keine syntaktische Erweiterung vorhanden ist. Die beiden Klassen innerhalb der Kategorie *Return_Statement* dürfen nicht gegeneinander getauscht werden, da die Klasse *Return_With_Value* die Oberklasse um einen syntaktischen Attribut erweitert.

Kategorie 7 „Static_Member_Selection“
Die Abbildung 4.17 zeigt die Klassen der Kategorie „Static_Member_Selection“. Diese Kategorie enthält keine Unterkategorien. Keine der konkreten Klassen dieser Kategorie erweitert die Oberklasse um ein syntaktischen Attribut. Somit dürfen hier alle Klassen gegeneinander substituiert werden. Diese Kategorie ist eigentlich eine Unterkategorie der Kategorie „Etity_L_Value“. Die konkrete Oberklasse „Etity_L_Value“ dieser Kategorie **ist Teil einer Sonderbehandlung** (siehe 4.3.4).

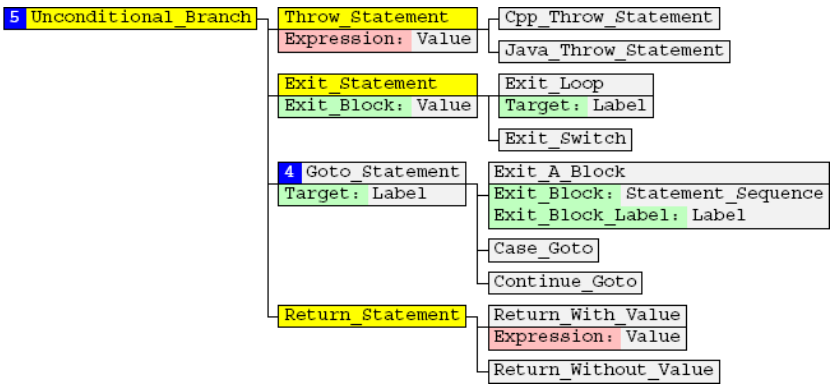

Abbildung 4.16: Klassen der Kategorie Unconditional_Branch

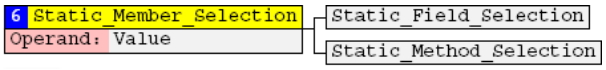

Abbildung 4.17: Klassen der Kategorie Static_Member_Selection

Kategorie 8 „Label“
Die Abbildung 4.18 zeigt die Klassen der Kategorie „Label“. Diese Kategorie enthält keine Unterkategorien. Keine der konkreten Klassen dieser Kategorie erweitert die Oberklasse um ein syntaktischen Attribut. Somit dürfen hier alle Klassen gegeneinander substituiert werden.

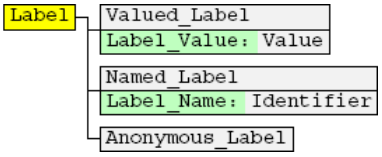

Abbildung 4.18: Klassen der Kategorie Label

Kategorie 9 „Assignment“
Die Abbildung 4.19 zeigt die Klassen der Kategorie „Assignment“. Diese Kategorie enthält keine Unterkategorien. Keine der konkreten Klassen dieser Kategorie erweitert die Oberklasse um ein syntaktischen Attribut. Somit dürfen hier alle Klassen, bis auf die Klasse *Postfix_Operator* (**siehe Sonderbehandlung** 4.3.4) gegeneinander substituiert werden.

Kategorie 10 „Operator“
Die Abbildung 4.20 zeigt die Klassen der Kategorie „Operator“.
Die Modellierung dieser Kategorie weicht von der Spezifikation der IML ab. Insgesamt enthält diese Kategorie sowohl in der Modellierung, als auch in der Spezifikation, zwei Unterkategorien *Binary_Operator* und *Unary_Operator*. Diese sind jedoch in der Modellierung, **anders wie in der Spezifikation der IML**, in keine weiteren Unterkategorien unterteilt. Die Unterkategorie *Binary_Operator* enthält weitere Unterkategorien *Add_Operator* und *Substract_Operator*. Diese wurden in der Modellierung ausgelassen.
Hierdurch wird ermöglicht, dass z.B. die Klasse *Arithmetic_Add* durch die Klasse *Arithmetic_Substract* substituiert werden darf. Die Unterkategorie *Unary_Operator* enthält eine Unterkategorie *Value_Conversion*, die ebenfalls eine Unterkategorie *Explicit_Conversion* enthält.

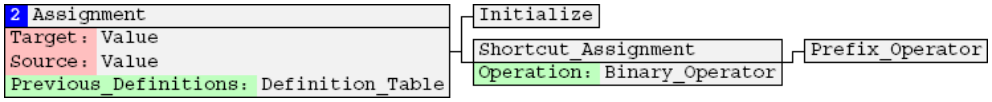

Abbildung 4.19: Klassen der Kategorie Assignment

Die konkreten Klassen der Unterkategorie *Explicit_Conversion* wurden als zu der Unterkategorie *Value_Conversion* zugehörend modelliert. Hierdurch wird ermöglicht, dass z.B. die Klasse *Static_Cast* durch die Klasse *Implicit_Conversion* substituiert werden darf. Die restlichen Klassen innerhalb der beiden Unterkategorien *Binary_Operator* und *Unary_Operator* können gegeneinander substituiert werden.

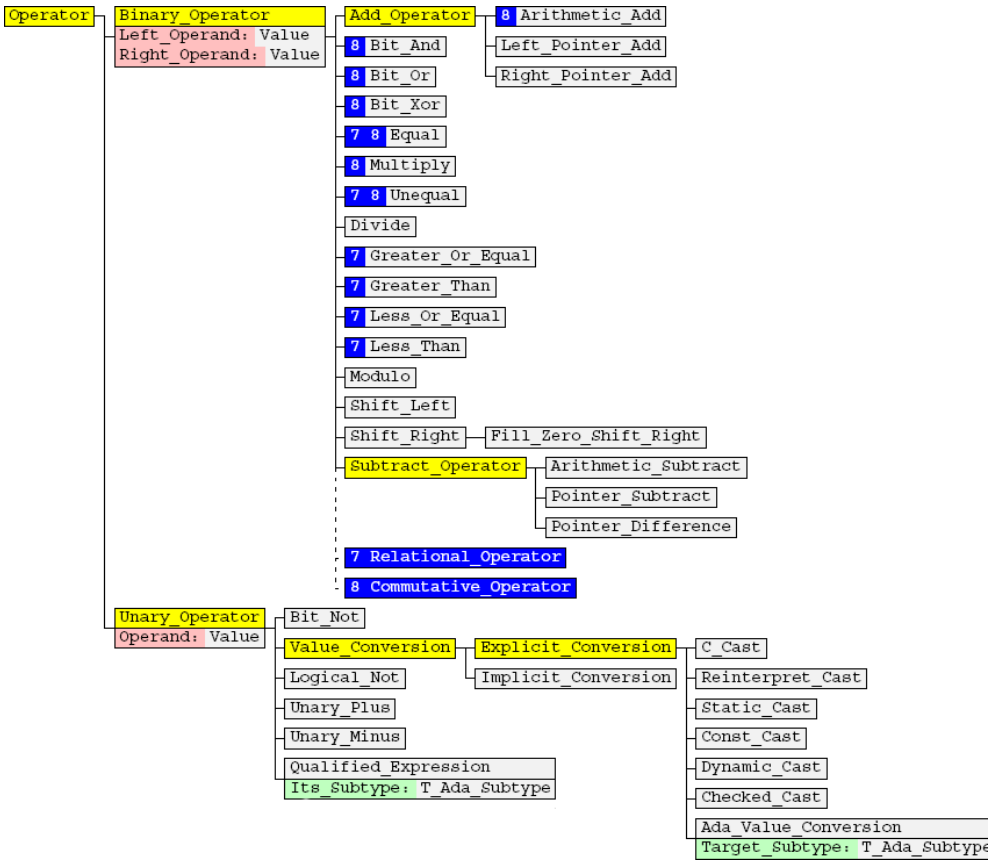

Abbildung 4.20: Klassen der Kategorie Operator

Kategorie 11 „Member_Selection“

Die Abbildung 4.21 zeigt die Klassen der Kategorie „Member_Selection“. Diese Kategorie enthält keine Unterkategorien. Keine der konkreten Klassen dieser Kategorie erweitert die Oberklasse um ein syntaktischen Attribut. Somit dürfen hier alle Klassen gegeneinander substituiert werden.

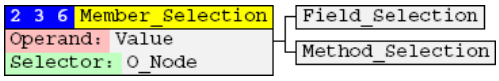

Abbildung 4.21: Klassen der Kategorie Member_Selection

Kategorie 12 „Pointer_To_Member_Selection“
Die Abbildung 4.22 zeigt die Klassen der Kategorie „Pointer_To_Member_Selection“. Diese Kategorie enthält keine Unterkategorien. Keine der konkreten Klassen dieser Kategorie erweitert die Oberklasse um ein syntaktischen Attribut. Somit dürfen hier alle Klassen gegeneinander substituiert werden.

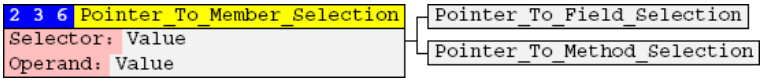

Abbildung 4.22: Klassen der Kategorie Pointer_To_Member_Selection

Kategorie 13 „Routine_Call“
Die Abbildung 4.23 zeigt die Klassen der Kategorie „Routine_Call“. Diese Kategorie beinhaltet eine Unterkategorie *Multiple_Callees*. Die Klasse *Virtual_Call* darf nicht durch andere Klassen dieser Unterkategorie substituiert werden. Die restlichen Klassen der Unterkategorie und der Kategorie selbst sind substituierbar.

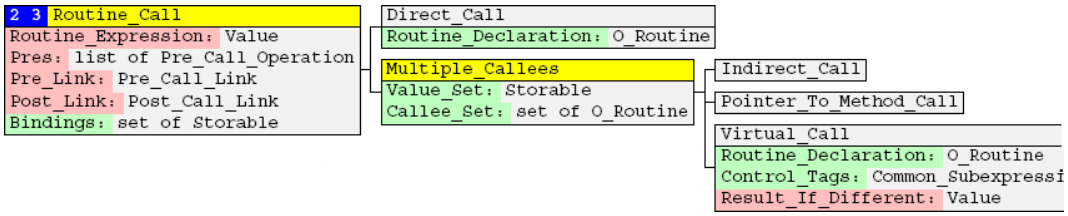

Abbildung 4.23: Klassen der Kategorie Routine_Call

Kategorie 14 „Conditional“
Die Abbildung 4.24 zeigt die Klassen der Kategorie „Conditional“. Diese Kategorie enthält keine Unterkategorien. Die Klasse *Assert* erweitert die Oberklasse um einen syntaktischen Attribut und ist somit von der Substitution ausgeschlossen. Die restlichen Klassen der Kategorie können gegeneinander substituiert werden.

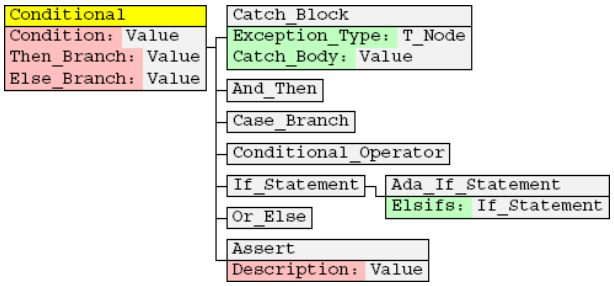

Abbildung 4.24: Klassen der Kategorie Conditional

Kategorie 15 „Loop_Statement“
Die Abbildung 4.25 zeigt die Klassen der Kategorie „Loop_Statement“. Diese Kategorie enthält keine Unterkategorien. Alle Klassen der Kategorie können gegeneinander substituiert werden, da keine die Oberklasse um einen syntaktischen Attribut erweitert.

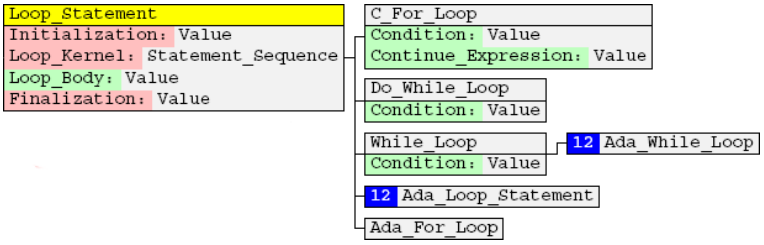

Abbildung 4.25: Klassen der Kategorie Loop_Statement

Sonderbehandlung
Die Sonderbehandlung beinhaltet IML-Klassen, die ähnliche programmiersprachliche Strukturen modellieren, jedoch durch die Hierarchie der IML zu unterschiedlichen Kategorien gehören und somit nicht gegeneinander substituierbar sind. Eine gesonderter Vergleich der IML-Klassenbezeichner lässt folgende zusätzliche Substitutionen zu:

- Substitution von *Postfix_Operator* durch *Prefix_Operator* und umgekehrt. Diese beiden Klassen liegen in unterschiedlichen Kategorien, modellieren jedoch ähnliche programmiersprachliche Strukturen und sollten deshalb substituierbar sein.
- Substitution der Klasse *Entity_L_Value* durch Klassen der Kategorie *Literal* und umgekehrt. Die Klasse *Entity_L_Value* modelliert Bezeichner. Die Klassen der Kategorie *Literal* modellieren Werte. Bezeichner sollen durch Werte substituierbar sein. Zum Beispiel kann auf der rechten Seite einer Zuweisung im ersten Codefragment eines Klonpaares ein Bezeichner stehen und im zweiten Codefragment ein Wert. Es wäre in diesem Fall sinnvoll den Bezeichner durch den Wert zu ersetzen, statt den Bezeichner zu Löschen und den Wert einzufügen.

4.3.5 „output“ Modul

Das Output Modul ist eine Ansammlung von Prozeduren, die für die Ausgabe verantwortlich sind. Neben zahlreichen Funktionen zur Ausgabe von Debugging-Informationen enthält das Modul auch Funktionen zu Interpretation und Ausgabe der Editiersequenz an den Benutzer. Prozeduren *Print_Graph_Stats*, *Print_Arc_Sequence_Stats* und *Print_Edit_Sequence_Stats* sind für die Erhebung und Ausgabe von Statistiken, wie z.B. Anzahl der Knoten und der Kanten in dem Editiergraph oder Anzahl der Kanten im Pfad bzw. Anzahl der Operationen in der Editiersequenz, verantwortlich. Die Prozedur *Create_User_Output* erzeugt aus der Editiersequenz eine auf Quellcode basierende Benutzerausgabe.

Die berechnete Editiersequenz besteht aus Editieroperationen (*Simple_Op* Objekte), die auf IML-Knoten operieren. Man könnte also mühelos die Information über die Veränderungen in der Struktur eines IML-Baums ausgeben. Zum Beispiel *„Löschen eines Knoten X aus dem Baum_1“* oder *„Löschen eines Unterbaumes mit Y als Wurzel aus dem Baum_1“* oder *„Einfügen eines Unterbaumes mit W als Wurzel aus dem Baum_2, als Kind von V aus dem Baum_1“*. Jedoch sind solche Informationen für einen Benutzer wenig hilfreich, da er nicht mit abstrakten Syntax Bäumen, sondern lediglich mit dem Quellcode eines Systems arbeitet. Somit sollte die Ausgabe in einem für den Benutzer nachvollziehbaren Format geschehen. Anstatt mit IML-Knoten zu arbeiten wird die Ausgabe auf Quellcodestrukturen abgebildet.

Hierfür können zum Einen die Klassenbezeichner der IML-Knoten und zum Anderen die Zeilen des Quellcodes verwendet werden. Jeder IML-Knoten beinhaltet einen SLOC Attribut, das die genaue Postion des simulierten programmiersprachlichen Konstruktes darstellt. Die an der gegebenen Stelle vorhandene Struktur kann zur Ausgabe verwendet werden. Anhand der im Postprocessing gemachten Markierung der Knoten kann entsprechend unterschieden werden, ob es sich beim Einfügen und Löschen um einfache Variablen bzw. Literale handelt oder komplexere Strukturen eingefügt bzw. gelöscht werden müssen. Ist eine Operation mit einer 1 markiert, so handelt es sich um einen Blattknoten und somit um einen Literal oder einen Bezeichner. Eine mit 3 markierte Operation deutet auf das Einfügen oder Löschen von komplexeren Strukturen bis hin zu den ganzen Zeilen.

Die Ausgabe der Substitutionsoperationen ist einfach zu erledigen. Beim Substituieren ist neben der Position der beiden Konstrukte im Quellcode nur noch interessant, welches Konstrukt durch welches Ersetzt wird. Somit ist es ausreichend diese Informationen auszugeben. Eine solche Ausgabe hat also die folgende Form:

ERSETZEN VON: *IML-Klassenbezeichner_1*
IN *Dateiname_1: Zeilenummer|Spalte Quellcodezeile_1*
*−−**DURCH**−− >:* *IML-Klassenbezeichner_2*
IN *Dateiname_2: Zeilenummer|Spalte Quellcodezeile_2*

Beim Löschen ist neben der Position des zu entfernenden Konstruktes im Quellcode das Konstrukt selbst interessant. Es ist ebenfalls interessant, ob es sich hierbei um eine komplexe oder einfache Struktur handelt. Eine Ausgabe für komplexe Strukturen hat also die folgende Form:

LÖSCHEN DER GANZEN: *IML-Klassenbezeichner* ***ANWEISUNG***
VON ZEILE *Dateiname: Zeilenummer|Spalte Quellcodezeile*

Eine Ausgabe für einfache Strukturen wie Literale oder Bezeichner hat also die folgende Form:

LÖSCHEN VON: *IML-Klassenbezeichner*
VON ZEILE *Dateiname: Zeilenummer|Spalte Quellcodezeile*

Die Ausgabe der Einfügeoperationen gestaltet sich etwas schwieriger. Hier ist es ebenfalls wichtig zu unterschieden, ob es sich um einfache oder komplexe Strukturen handelt. Die Position des Konstruktes im Quellcode, sowie das Konstrukt selbst, müssen ausgegeben werden aber auch die Einfügeposition ist von großer Bedeutung. Ein Knoten wird immer als Kind eines Elternknoten eingefügt. Dieser Elternknoten kann eindeutig bestimmt werden und wird beim Postprocessing im *Parent_Node* Attribut des *Simple_Op* Objektes gespeichert. Zur Ausgabe steht also nur die Position des Konstruktes im Quellcode, das durch das Elternknoten simuliert wird. Da zwischen komplexen und einfachen Konstrukten unterschieden wird, kann ein Konstrukt nach einer Zeile/Anweisung oder in das Innere dieser eingefügt werden. Eine Ausgabe für komplexe Strukturen hat also die folgende Form:

EINFÜGEN DER GANZEN: *IML-Klassenbezeichner_1* ***ANWEISUNG***
AUS *Dateiname_1: Zeilenummer|Spalte Quellcodezeile_1*
*−**INNERHALB|NACH-ZEILE**− >:* *Dateiname_2: Zeilenummer|Spalte Quellcodezeile_2*

Eine Ausgabe für einfache Strukturen wie Literale oder Bezeichner hat also die folgende Form:

EINFÜGEN VON: *IML-Klassenbezeichner_1*
AUS *Dateiname_1: Zeilenummer|Spalte Quellcodezeile_1*
−INNERHALB-ZEILE− >: *Dateiname_2: Zeilenummer|Spalte Quellcodezeile_2*

Die Listings 4.19 und 4.20 enthalten eine Ausgabe aus dem Vergleich der Codefragmente eines Klopaares aus dem System ***Eclipse-Ant***. Zunächst werden noch vor dem Berechnen der Unterschiede die beiden zu vergleichenden Codefragmente des Klonpaares ausgegeben. Dies geschieht unmittelbar nach der Extraktion des Klonpaares aus der IML durch das „clone_utils“ und „file_handling_utils“ Modul. Das Listing 4.19 enthält die Ausgabe der Codefragmente eines Klonpaares.

Listing 4.19: Ausgabe der Codefragmente eines Klonpaares an Benutzer

```
————————————CLONE_1————————————
File: Jvc.java
 91          Commandline cmd = new Commandline();
 92          cmd.setExecutable("jvc");
 93          if (destDir != null) {
 94              cmd.createArgument().setValue("/d");
 95              cmd.createArgument().setFile(destDir); }
 96          // Add the Classpath before the "internal" one.
 97          cmd.createArgument().setValue("/cp:p");
————————————————————————————————
————————————CLONE_2————————————
File: Kjc.java
 94          Commandline cmd = new Commandline();
 95         // generate classpath.
 96          Path classpath = getCompileClasspath();
 97          if (deprecation == true) {
 98              cmd.createArgument().setValue("-deprecation");}
 99          if (destDir != null) {
 100              cmd.createArgument().setValue("-d");
 101              cmd.createArgument().setFile(destDir); }
 102          // generate the clsspath
 103          cmd.createArgument().setValue("-classpath");
————————————————————————————————
```

Nach der Berechnung und dem Postprozessieren der Editiersequenz wird diese, wie oben beschrieben, an den Benutzer ausgegeben. Hier werden ebenfalls die Funktionen aus dem „file_hanling_utils“ Modul aufgerufen. Neben der Ausgabe der Operationen selbst werden noch Statistiken zu der Editiersequenz ausgegeben. Dabei handelt es sich um die Anzahl der Operationen vor und nach dem Postprocessing. Im Listing 4.20 ist eine Ausgabe einer Editiersequenz zu den Codefragmenten aus 4.19 dargestellt.

Listing 4.20: Beispiel der Ausgabe der Editiersequenz an Benutzer

```
————————————————————————
Path_Editdistance =  47
————————————————————————
Editdistance =  3
————————————————————————
```

```
DELETION OF ENTIRE: Virtual_Call STATEMENT FROM LINE
Jvc.java:  92| 13   cmd.setExecutable("jvc");
-----------------------------------------
INSERTION OF ENTIRE: Initialize STATEMENT IN
Kjc.java:  96| 9    Path classpath = getCompileClasspath();
-INTO/AFTER-LINE->:
Jvc.java:  92| 27   cmd.setExecutable("jvc");
-----------------------------------------
INSERTION OF ENTIRE: If_Statement STATEMENT IN
Kjc.java:  97| 9    if (deprecation == true) {
-INTO/AFTER-LINE->:
Jvc.java:  92| 27   cmd.setExecutable("jvc");
-----------------------------------------
```

Eine Ausgabe für das Einfügen oder Löschen einer komplexeren Struktur beinhaltet also ein *DER GANZEN* bzw. *ENTIRE* String. Dies soll andeuten, dass die gesamte Struktur eingefügt oder gelöscht werden soll, auch wenn diese sich über mehrere Zeilen erstreckt. Im Beispiel 4.20 soll eine ganze „if“-Anweisung eingefügt werden. Diese erstreckt sich über die Zeilen 97 und 98, somit sollen alle diese Zeilen als Teil der gesamten „if“-Anweisung miteingefügt werden.

4.3.6 „file_handling_utils“ Modul

Das Modul **„file_handling_utils“** ist sehr einfach und besteht lediglich aus einer Funktion und einer Prozedur. Dieses Modul wird von dem **„output“** und dem **„clone_utils“** Modul ausschließlich dazu verwendet, eine Ausgabe an den Benutzer zu erzeugen. Die Aufgabe dieses Moduls besteht darin, eine Zeile oder ein Fragment einer Quellcodedatei einzulesen und auf der Konsole auszugeben oder an den Aufrufer zurück zu geben. Das Auslesen einer Zeile der Quellcodedatei wird von der Funktion **Get_Nth_Line_Of_File** übernommen.
Die Funktion benötigt als Eingabe den Pfad zu der Quellcodedatei und die Nummer der zu lesenden Zeile. In dem Listing 4.21 wird das Innere der Funktion dargestellt.

Listing 4.21: Ausgabe einer Zeile einer Datei

```
  Result_List  : String_Lists.List;
  Result     : Ada.Strings.Unbounded.Unbounded_String
             := Ada.Strings.Unbounded.To_Unbounded_String ("");
begin
  Result_List:=String_Lists_Serializer.From_File(Path);
  Result:= String_Lists.Get_Nth(Result_List,Line_Num-1);
return Ada.Strings.Unbounded.To_String(Result);
exception
   when CONSTRAINT_ERROR => Result:=
      Ada.Strings.Unbounded.To_Unbounded_String
                              (" NO SRC LINE AVALIABLE");
      return Ada.Strings.Unbounded.To_String(Result);
   when STRING_LISTS.ITEMNOTPRESENT => Result:=
      Ada.Strings.Unbounded.To_Unbounded_String
                              (" NO SRC LINE AVALIABLE");
      return Ada.Strings.Unbounded.To_String(Result);
```

Um eine Zeile einzulesen, werden die im „Bauhaus“ vorhandenen Bibliotheken *String_Lists_Serializer* und *String_Lists* verwendet. Mit Hilfe der ersten wird eine Datei eingelesen und in einer Liste von Strings gespeichert, die von der zweiten Bibliothek zur Verfügung gestellt wird. Jedes Stringelement der Liste beinhaltet eine Zeile der Datei. Somit muss nur noch die benötigte Zeile aus der Liste entnommen und zurückgegeben werden. Im Falle, wenn es unter dem gegebenen Pfad keine Datei existiert, wird eine Exception abgefangen und ein *NO SRC_LINE AVALIABLE* zurückgegeben.

Die Prozedur **Print_File_Fragment** ist für das Auslesen und Ausgeben eines Fragmentes einer Datei verantwortlich. Als Eingabe werden neben dem Pfad zu der Datei auch die Nummern der Start- und Endzeile benötigt. Im Listing 4.22 ist die Umsetzung zu sehen.

Listing 4.22: Ausgabe eines Fragmentes einer Datei

```
begin
 for I in Start_Line .. End_Line loop
   Ada.Text_Io.Put(Integer'Image(I) & " " &
                   Get_Nth_Line_Of_File(Path ,I));
   Ada.Text_Io.New_Line;
end loop;
```

In einer Schleife wird je eine Zeile der Datei zwischen Start- und Endnummer ausgelesen und auf der Konsole ausgegeben. Zum Auslesen der einzelnen Zeilen wird die oben beschriebene Funktion **Get_Nth_Line_Of_File** aufgerufen.

KAPITEL 5

Evaluation

Inhalt

In diesem Kapitel wird die Evaluation des Systems beschrieben. Im Abschnitt 5.1 wird die Vorgehensweise beschrieben. Der Abschnitt 5.2 enthält eine kurze Beschreibung der realen Systeme, die für die Evaluation verwendet wurden. Im Abschnitt 5.3 werden letztendlich die Ergebnisse der Evaluation vorgestellt.

5.1 Vorgehen

Die eigentliche Aufgabe des implementierten Systems besteht darin, die Unterschiede eines Typ3 Klonpaares zu berechnen und in einem geeigneten Format an den Benutzer auszugeben. Es wäre jedoch ohne weiteres möglich die Unterschiede zwischen beliebigen geordneten Bäumen zu berechnen, solange diese aus der IML extrahiert werden können bzw. in dem IML-Format vorliegen. Während der Entwicklungsphase wurde das System an solchen Bäumen getestet, die nicht immer Typ3 Klone repräsentierten. Es wurden unzählige einfache Java Klassen entwickelt, in die IML überführt und mit Hilfe des implementierten Tools verglichen. Diese Regressionstests während der Entwicklung reichen jedoch nicht aus, um die interessanten Fragen bezüglich der Qualität des implementierten Systems zu beantworten. Nach der Implementierungsphase wurde durch eine abschließende Evaluation des Systems versucht, Fehler in der Implementierung zu finden und folgende Fragen zu beantworten:

- Ist die berechnete Editiersequenz korrekt?
- Ist die berechnete Editiersequenz optimal?
- Wie effektiv ist das Postprocessing?
- Wie ist das Laufzeitverhalten des Systems?

Um diese Fragen beantworten zu können, werden Daten benötigt, die eine Analyse ermöglichen.

Optimal wäre eine größere Ansammlung an bereits identifizierten Typ3 Klonen, deren Unterschiede bereits analysiert worden sind. Zum Zeitpunkt der Entstehung dieser Arbeit befand sich eine solche Ansammlung an auf Unterschiede analysierten Typ3 Klonen in der Entwicklung (siehe [Tiarks, Koschke und Falke 2009]). Um die Analyse durchzuführen, wurde eine eigene Ansammlung an Typ3 Klonen erstellt. Hierfür wurden zunächst sechs reale Softwaresysteme auf die Belastung durch Klone untersucht. Bei den Systemen handelt es sich um: Eclipse-Ant, Eclipse JDT-Core, Javax-swing, SNNS, Wget und GNU-Bison (siehe 5.2). Die IML-Graphen der Systeme wurden an ccdiml übergeben. Das Tool wird dabei mit folgenden Parametern aufgerufen: *./ccdiml* ***-mode fine -minlines 5 -outformat iml*** */Pfad/zu/der/-datei.iml*. Die Parameter ***-mode fine*** und ***-minlines 5*** weisen das Tool an, Klone von mindestens fünf Zeilen Länge auf der Anweisungsebene zu suchen. Der Parameter **-outformat iml** weist das Tool an die identifizierten Klonkandidaten, wie in 4.1.1 beschrieben, in der IML zu annotieren. Auf diese Weise identifizierten Typ3 Klone sind bezüglich ihrer Größe und Komplexität gut analysierbar. Die Tabelle 5.1 zeigt die Anzahl der identifizierten Typ3 Klone in dem jeweiligen System.

System	Version	KLOC	Sprache	Anzahl der Typ3 Klone
Eclipse-Ant	Datum 15.2.02	35	Java	135
Eclipse JDT-Core	r3.3	148	Java	22439
Javax-swing	J2SDK1.4.0	204	Java	6062
GNU-Bison	1.32	19	C	564
Wget	1.5.3	16	C	128
SNNS	4.2	115	C	564

Die Anzahl der identifizierten Typ3 Klone ist sehr hoch. Somit ist es nicht möglich, alle Klone auf Unterschiede zu untersuchen. Deshalb wurde entschieden, zufällig Klone aus der vorhandenen Menge zu entnehmen. Für jedes der Systeme wurden zehn Zahlen zwischen 1 und n mit einem Zufallszahlengenerator erzeugt, wobei n die Gesamtanzahl der Typ3 Klone des jeweiligen Systems darstellt. Die auf diese Art entnommenen Stichproben bilden die Basis für weitere Analysen während der Evaluation.

Jedes Typ3 Klonpaar der Stichproben wurde von dem implementierten System auf Unterschiede untersucht. Dabei wurde eine Editiersequenz berechnet, einem Postprocessing unterzogen, interpretiert und ausgegeben. Um festzustellen, ob die Editiersequenz korrekt und optimal ist und somit die ersten zwei Fragen zu beantworten, wurde jede Ausgabe einer genauen Betrachtung unterzogen. Dabei wurde die ausgegebene **„ist“-Editiersequenz** mit der von dem Benutzer erwarteten optimalen **„soll“-Editiersequenz** verglichen.

Bei der Berechnung der Editiersequenz wurden neben der Ausgabe der Codefragmente und der Editiersequenzinterpretation auch Laufzeitdaten erhoben und ebenfalls ausgegeben. Die Auswertung der erhobenen Daten wird im Abschnitt 5.3.3 vorgestellt.

5.2 Systeme

In diesem Abschnitt wird eine kurze Beschreibung der Systeme gegeben, die zu Gewinnung der Eingabedaten für die Evaluation des entwickelten Tools verwendet wurden. Dabei wurden drei Java (Javax-swing, Eclipse-Ant, Eclipse JDT-Core) und drei C/C++ (SNNS, Wget, GNU-Bison) Systeme ausgesucht. Das Erzeugen der IML-Dateien für diese Systeme wurde freundlicherweise von der AG-Softwaretechnik übernommen, da dieser Vorgang sehr komplex ist und viel Erfahrung im Umgang mit den entsprechenden Bauhaus-Werkzeugen benötigt.

Javax-swing
Bei Swing handelt es sich um eine Programmierschnittstelle und Grafikbibliothek zum Programmieren von grafischen Benutzeroberflächen. Swing wurde von Sun Microsystems für die Programmiersprache Java entwickelt. Seit Java-Version 1.2 ist es Bestandteil der Java-Runtime. Swing gehört zu den Java Foundation Classes, die eine Sammlung von Bibliotheken zur Programmierung von grafischen Benutzerschnittstellen bereitstellen. Zu diesen Bibliotheken gehören Java2D, das Accessibility-API, das Drag & Drop-API und das Abstract Window Toolkit.
Swing baut auf dem älteren AWT auf und ist mit den anderen APIs verwoben [1].

Eclipse-Ant
Die Komponente Eclipse-Ant ist entwickelt worden, um die Stärken von Apache-Ant und Eclipse zu verbinden. Apache-Ant ist ein auf Java basierendes Werkzeug zum Kompilieren der in Java entwickelten Systeme. Als ein von Eclipse unabhängiges Tool ist Ant ein Teil des Apache open-source Projektes. Von der Funktionalität her ist dieses Werkzeug dem C/C++ make Vorgang ähnlich. Durch Integration von Ant in Eclipse hat Eclipse-Ant folgende Funktionalität:

- Ant „buildfiles“ aus Eclipse heraus ausführen.
- Zugriff auf Ressourcen und Funktionalität der Entwicklungsumgebung Eclipse aus den Ant „buildfiles“
- Bereitstellen der Graphischen Oberfläche für das beobachten des Ausgabe der laufenden Kompilierungsprozesse
- Bereitstellen der Werkzeuge und Hilfe bei entwickeln der Ant „buildfiles“ (z.B. „buildfile“ Editor, Debugger und andere)

Das Ziel des Tools Eclipse-Ant besteht also darin, Ant als ein Teil der Entwicklungsumgebung Eclipse zu realisieren. Die Entwickler, welche Eclipse benutzen, werden dadurch beim Kompilieren stärker unterstützt, da keine externen Komponenten zum Kompilieren komplexer Systeme mehr benötigt werden [2].

Eclipse JDT-Core
Eclipse JDT-Core ist eine in der Entwicklungstool Eclipse integrierte Java-Infrastruktur. Es beinhaltet:

- Einen inkrementellen Java-Compiler. Dieser Compiler basiert auf der von VisualAge entwickelten Java-Compiler Technologie. Die Besonderheit der Technologie besteht darin, dass diese eine Kompilierung und Debugging des Quellcodes ermöglicht, welches ungelöste Fehler enthält.
- Ein Java-Modell, das eine API zum Navigieren des Java-Element-Baumes bereitstellt. Einen Java-Element-Baum definiert eine Java zentrierte Ansicht eines Projektes. Es visualisiert Elemente wie Paketfragmente, Kompilierungseinheiten, binäre Klassen, Typen, Methoden und Felder.
- Ein Java-Dokument-Modell, das eine API zum Manipulieren der strukturierten Java-Source Dokumente bereitstellt.

[1] http://de.wikipedia.org/wiki/Swing_(Java)
[2] http://www.eclipse.org/eclipse/ant/

- Einen Quellcode Formatierer.
- ...[3].

Snns
SNNS (Stuttgart Neural Network Simulator) ist ein effizienter Simulator neuronaler Netze für Unix Workstations und Parallelrechner, welches in dem Institut für Parallele und Verteilte Systeme an der Universität Stuttgart entwickelt wurde.
Er besitzt eine graphische Oberfläche unter X-Windows, eine umfangreiche Bibliothek moderner neuronaler Lernverfahren und ist portabel und erweiterbar. SNNS wird auch bei externen Kooperationspartnern und über 300 Anwendern weltweit eingesetzt. Anwendungen sind u.a. Zeichen- und Ziffernerkennung, Erkennung von Werkstückbildern, Texturerkennung, Klassifikation von EEG-Signalen, Steuerung autonomer Fahrzeuge, Spracherkennung mit Lippenlesen, Börsenkursprognose, Lastprognose in Stromnetzen, Regelung von Elektronenbahnen in einem Speicherring, Farbrezepturberechnung, Ähnlichkeitsanalyse biologisch aktiver Moleküle und Proteinstrukturvorhersage[4].

Wget
GNU Wget ist ein freies Kommandozeilen-Programm zum Herunterladen von Ressourcen (Dateien, Webseiten, etc) über ein Netzwerk. Zu den unterstützten Protokollen gehören ftp, http und https. Die erste Version stammt aus dem Jahr 1995 und wurde von Hrvoje Niksic geschrieben. Das Programm gibt es sowohl für UNIX und GNU/Linux als auch für OS/2, Windows und für SkyOS. Es steht unter der GNU General Public License und ist Teil des GNU-Projekts. Wget kann einen abgebrochenen Download wieder aufnehmen, komplette Webseiten mit Bildern sowie vollständige Websites herunterladen, z.B. zum Offline-Lesen oder zur Archivierung. Zusätzlich kann man über das Angeben einer IP-Adresse Multipath Routing nutzen, einen Browser vortäuschen, automatisch zufällig lange Pausen einlegen und einiges mehr. Dadurch ist es sowohl als universeller Download-Manager, als auch als Offline-Reader einsetzbar [5].

GNU-Bison
GNU-Bison ist eine Open-Source Weiterentwicklung des Parsergenerators Yacc und ist Teil des GNU-Projektes. Bison konvertiert eine Grammatik einer Programmiersprache in ein C oder C++ Programm. Dieses Programm ist in der Lage die Sequenzen von Token zu parsen, die mit der gegebenen Grammatik konform sind. Solche Parser werden vor allem im Bereich des Compilerbau benötigt[6].

5.3 Auswertung

In diesem Abschnitt werden die Ergebnisse der Arbeit präsentiert und Antworten auf die in 5.1 gestellte Fragen geliefert. Zunächst wird kurz auf die Typ3 Klone der Stichproben eingegangen, welche die Basis der Evaluation bilden. Danach wird die Analyse und ihre Ergebnisse beschrieben. Letztendlich wird die Untersuchung der Laufzeitkomplexität des Systems und Effektivität des Postprocessing vorgestellt.

[3] http://www.eclipse.org/jdt/core/index.php
[4] http://www.informatik.uni-stuttgart.de/kolloquium/zell.html
[5] http://de.wikipedia.org/wiki/Wget
[6] http://en.wikipedia.org/wiki/GNU_bison

5.3.1 Stichproben

Die Typ3 Klone in den entnommenen Stichproben variieren stark in ihrer Größe und Komplexität. Das Listing 5.1 zeigt eines der einfachsten analysierten Typ3 Klonpaare, deren Codefragmente aus lediglich einer bzw. zwei Zeilen bestehen. Die Komplexität der Anweisungen der Codefragmente geht nicht über einfache Zuweisungen und Postfix Operatoren hinaus. Die Codefragmente unterscheiden sich lediglich in einer Zeile, sind also bis auf den Postfix Operator im zweiten Codefragment gleich. Das Listing 5.2 zeigt eines der komplexesten analysierten Typ3 Klonpaare, deren Codefragmente aus mehreren Zeilen bestehen.
Die Anweisungen der beiden Codefragmente beinhalten mehrere „if“-Anweisungen und somit auch Vergleichsoperatoren, Zuweisungen mit Funktionsaufrufen und Rückgabeanweisungen. Die Codefragmente sind bis auf eine „if“-Anweisung und unbedeutenden Unterschieden der Bezeichner bzw. Literale gleich.

Listing 5.1: Das einfachste analysierte Typ3 Klonpaar

```
----------------CLONE_1----------------------
File: CodeStream.java
 2796          countLabels = 0;
 2797          stackDepth--;
----------------CLONE_2----------------------
File: CodeStream.java
 2943          countLabels = 0;
-----------------------------------------------
DELETION OF ENTIRE: Postfix_Operator STATEMENT
FROM LINE  CodeStream.java:  2797| 18        stackDepth--;
-----------------------------------------
```

Listing 5.2: Das komplexeste analysierte Typ3 Klonpaar

```
----------------CLONE_1----------------------
File: ftp-basic.c
 327   nwritten = iwrite (RBUF_FD (rbuf),
                 request, strlen (request));
 328   if (nwritten < 0) {
 329       free (request);
 330       return WRITEFAILED; }
 331   free (request);
 332   /* Get appropriate response.  */
 333   err = ftp_response (rbuf, &respline);
 334   if (err != FTPOK) {
 335       free (respline);
 336       return err; }
 337   if (*respline != '3') {
 338       free (respline);
 339       return FTPRESTFAIL; }
 340   free (respline);
 341   /* All OK.  */
 342   return FTPOK; }
```

```
----------------CLONE_2----------------
File: ftp-basic.c
 351    nwritten = iwrite (RBUF_FD (rbuf),
                 request, strlen (request));
 352    if (nwritten < 0) {
 353        free (request);
 354        return WRITEFAILED; }
 355    free (request);
 356    /* Get appropriate response.  */
 357    err = ftp_response (rbuf, &respline);
 358    if (err != FTPOK) {
 359        free (respline);
 360        return err; }
 361    if (*respline == '5') {
 362        free (respline);
 363        return FTPNSFOD; }
 364    if (*respline != '1') {
 365        free (respline);
 366        return FTPRERR; }
 367    free (respline);
 368    /* All OK.  */
 369    return FTPOK; }
-----------------------------------------
INSERTION OF ENTIRE: If_Statement STATEMENT
IN  ftp-basic.c:  361| 3  if (*respline == '5') {
-INTO/AFTER-LINE->:  ftp-basic.c:  336| 14  return err; }
-----------------------------------------
```

Es wurden insgesamt sechzig Typ3 Klonpaare unterschiedlicher Komplexität, wie in 5.1 beschrieben, extrahiert und während der Evaluationsphase zur Analyse der Systemqualität verwendet. Die Analyse und die Ergebnisse werden als nächstes im Abschnitt 5.3.2 vorgestellt.

Unmittelbar nach der Ausgabe der Codefragmente und vor der Ausgabe der Editiersequenz erfolgt die Ausgabe der Laufzeitdaten. Der Übersicht halber wurden diese Daten in den oben gezeigten Beispielen ausgelassen. Das Listing 5.3 zeigt eine exemplarische Ausgabe der Laufzeitdaten. Es werden alle zur Analyse der Laufzeit notwendige Daten erhoben und ausgegeben.

Listing 5.3: Ausgabe der Laufzeitdaten und Statistik

```
-----------------------------------------
TREE_1 HAS:   18 NODES
TREE_2 HAS:   32 NODES
----------------GRAPH_STATS----------------
Graph has:   577 Nodes
Graph has:   840 Arcs
--DUR.-GRAPH-CONSTRUCT.-(Seconds)--
 0.050463000
-----------------------------------------
---DUR.-PATHEXTRACTION-(Seconds)---
 0.013051000
```

```
-----------------------------------------
---DUR.-POSTPROZESING-(Seconds)----
 0.000184000
-----------------------------------------
---EFFECTIVENESS-POSTPROZESSING----
Path_Editdistance =  37
---------Shortest Path has---------
  5del Arcs
  19ins Arcs
  13sub Arcs
-----------------------------------------
Editdistance =  6
---------Editsequence has----------
  3del Arcs
  3ins Arcs
  0sub Arcs
-----------------------------------------
```

5.3.2 Analyse

Bei der Analyse wurde, wie bereits erwähnt, jede Ausgabe einer genauen Betrachtung unterzogen. Dabei wurde die ausgegebene „ist“-Editiersequenz mit der von dem Benutzer erwarteten korrekten „soll“-Editiersequenz verglichen. Eine Übereinstimmung der „soll“-Editiersequenz mit der „ist“-Editiersequenz belegt somit die Korrektheit der berechneten Editiersequenz. Zusätzlich zu der Korrektheit wurde auf die Optimalität der Editiersequenz geachtet. Die Begriffe Korrektheit und Optimalität werden im Kontext der Editiersequenz als nächstes erläutert.

Eine berechnete Editiersequenz wird nur dann als **korrekt** bewertet, wenn sie mit der „soll“-Editiersequenz übereinstimmt und bei Anwendung der Operationen die Struktur des ersten Codefragment an die Struktur des zweiten Codefragments angleicht. Das Listing 5.4 zeigt eine Ausgabe der Editiersequenz eines Klonpaares aus dem System Javax-Swing. Die „soll“-Editiersequenz würde also das Löschen der gesamten „if“-Anweisung von der Zeile 975 und der der gesamten „if“-Anweisung von der Zeile 977 vorschlagen. Wendet man die Operationen der berechneten Editiersequenz an, so wird die Struktur des ersten Codefragments an die Struktur des zweiten Codefragments angeglichen. Sie stimmt zusätzlich mit der „soll“-Editiersequenz überein und ist somit korrekt.

Listing 5.4: Beispiel einer Ausgabe der korrekten Editiersequenz eines Klonpaares aus Javax-swing

```
-----------------CLONE_1-----------------------
File: CodeStream.java
 974        countLabels = 0;
 975        if (maxLocals <= 3) {
 976              maxLocals = 4; }
 977        if (stackDepth > stackMax)
 978              stackMax = stackDepth;
 979        try {
```

```
————————————CLONE_2————————————
File: CodeStream.java
 3616        countLabels = 0;
 3617        try {
————————————————————————————————
DELETION OF ENTIRE: If_Statement STATEMENT FROM LINE
CodeStream.java:  975| 8  if (maxLocals <= 3) {
————————————————————————————————
DELETION OF ENTIRE: If_Statement STATEMENT FROM LINE
CodeStream.java:  977| 8  if (stackDepth > stackMax)
```

Eine berechnete Editiersequenz wird nur dann als **optimal** bewertet, wenn sie korrekt und zusätzlich (bezüglich der Anzahl und Art der Editieroperationen) die günstigste ist. Das Listing 5.5 zeigt eine Ausgabe der Editiersequenz eines Klonpaares aus dem System SNNS. Eine korrekte Editiersequenz könnte das Ersetzen des „==“-Operators durch „<“-Operator (Angleichen der Bedingung der ersten „if“-Anweisungen der Codefragmente) und anschließend das Löschen der ersten „if“-Anweisung von der Zeile 352 des ersten Codefragments (Angleichen der Anzahl der „if“-Anweisungen der Codefragmente) vorschlagen. Das Anwenden dieser Operationen würde die Struktur der Codefragmente angleichen. Diese Editiersequenz wäre zwar korrekt jedoch nicht die günstigste. Es ist günstiger die erste „if“-Anweisung einfach zu löschen. Die berechnete Editiersequenz schlägt genau diese eine Editieroperation vor und ist somit sowohl korrekt als auch optimal. Eine optimale Editiersequenz kann nur dann berechnet werden, wenn die Gewichte der Substitutionskanten sinnvoll gewählt worden sind. Die Analyse der Optimalität der Editiersequenzen dient also der Überprüfung der Wahl dieser Gewichte.

Listing 5.5: Beispiel einer Ausgabe der optimalen Editiersequenz eines Klonpaares aus SNNS

```
————————————CLONE_1————————————
File: kr_td.c
 350     if (NoOfUnits == 0)
 351        return( KRERR_NO_UNITS );
 352     if (NoOfInParams < 1)
 353        return( KRERR_PARAMETERS );
 354     *NoOfOutParams = 1;
 355     *parameterOutArray = OutParameter;
————————————CLONE_2————————————
File: learn_f.c
 2278     if (NoOfInParams < 1)
 2279        return (KRERR_PARAMETERS);
 2280     *NoOfOutParams = 1;
 2281     *parameterOutArray = OutParameter;
————————————————————————————————
DELETION OF ENTIRE: If_Statement STATEMENT FROM LINE
kr_td.c:  350| 5    if (NoOfUnits == 0)
```

Insgesamt waren **59 von 60** der analysierten Editiersequenzen korrekt und optimal. Bei dem einem übrigbleibenden Klonkandidaten konnte, aufgrund stark komplexer Syntax, keine „soll“-Editiersequenz erstellt werden, sodass eine genaue Analyse der Korrektheit und Optimalität nicht möglich war. Dieses Ergebnis belegt, dass die Gewichte für die Substitutionskanten des Editiergraphen und die Einteilung der IML-Klassen in bestimmte Kategorien angemessen gewählt wurden.

5.3.3 Messung der Laufzeit

Die Berechnung der Unterschiede besteht im Wesentlichen aus drei Schritten. Nach der Extraktion der abstrakten Syntax Bäume wird als erstes der Editiergraph aufgebaut. Nach diesem Schritt wird ein kürzester Weg von dem Start zu dem Endknoten des Graphen extrahiert. Dieser Pfad wird schließlich in eine Editiersequenz umgewandelt, die dann einer Nachbehandlung(Postprocessing) unterzogen wird. Alle drei Phasen der Berechnung nehmen eine gewisse Zeit der gesamten Dauer in Anspruch.

Soft- und Hardware
Die Laufzeit hängt stark von der Soft- und Hardware ab. Alle Messungen wurden auf folgendem System durchgeführt:

- System: Toshiba Satellite M30X Notebook
- Prozessortyp: Intel Pentium M 725, 400 MHz Front Side Bus und L2 Cache mit 2 MB.
- Prozessorgeschwindigkeit: 1.6 GHz.
- Arbeitsspeicher: 512 MB DDR-RAM PC 2700.
- Speicher: 60-GB-S.M.A.R.T.-Festplatte.
- „Bauhaus“ Revision: 26996
- Betriebsystem: Ubuntu 7.10, Gutsy Gibbon

Dauer Grapherzeugung
In der ersten Phase der Berechnung wird also ein Editiergraph erzeugt. Die theoretische Laufzeitkomplexität beim Aufbauen des Editiergraphen liegt im „worstcase“ in der Komplexitätsklasse $O(n^2)$ mit $n = max\{n_1, n_2\} \times k$, wobei n_1 und n_2 die Anzahl der Knoten der beiden abstrakten Syntax Bäume sind und k eine konstante Zeit die zum Erzeugen eines Graphknoten bzw. einer Graphkante benötigt wird (Siehe 3.3.3).

Die Abbildung 5.1 zeigt zusammengefasst das gemessene Laufzeitverhalten des Systems während der Grapherzeugung. Auf der Abszisse ist die Anzahl der erzeugten Graphknoten zu sehen. Auf der Ordinate ist gemessene Zeit in Sekunden dargestellt. Die grüne Linie stellt den Verlauf einer quadratischen Funktion dar.

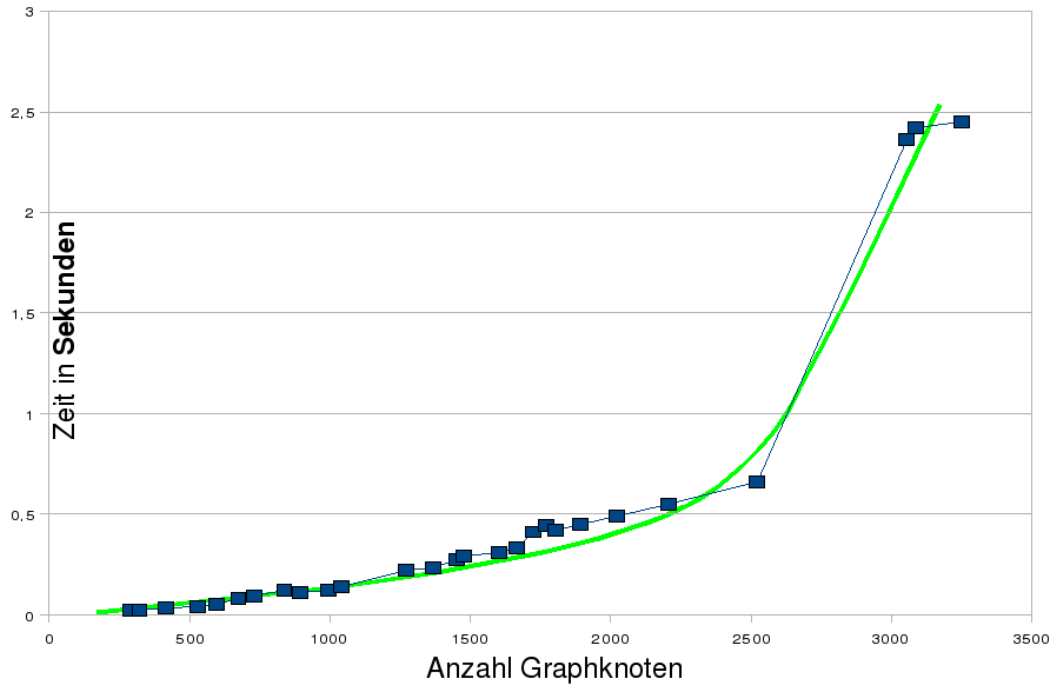

Abbildung 5.1: Dauer der Erzeugung eines Editiergraphen mit bis zu 3500 Knoten

Exemplarische Nachrechnungen decken sich weitgehend mit der Annahme, dass das Laufzeitverhalten des Systems bei der Erzeugung des Editiergraphen im „worstcase“ in der Komplexitätsklasse $O(n^2)$ liegt. Der Verlauf des entstandenen Graphen in der Abbildung 5.1 bestätigt die Annahme zusätzlich.

Dauer Pfadextraktion
In der zweiten Phase der Berechnung wird aus dem erzeugten Editiergraph ein kürzester Weg extrahiert. Die Kernkomponente der Pfadextraktion ist der Algorithmus von Dijkstra. Der Aufwand von Dijkstra's Algorithmus mit einer Prioritätswarteschlange auf der Basis einer verketteten Liste liegt in der Komplexitätsklasse $O(n^2)$. Wobei n die Anzahl der Graphknoten repräsentiert (mehr in [Algorithmen und Datenstrukturen]). Dieser Algorithmus wird dazu verwendet, die Entfernung aller Knoten im Graph zu dem Startknoten zu berechnen und in dem Graph zu annotieren. Zusätzlich wird pro Knoten ein Vorgänger auf dem kürzesten Pfad annotiert. Um einen Pfad zu extrahieren, werden nach der Annotation, von dem Endknoten aus, die referenzierten Vorgängerknoten verfolgt. Diese Prozedur liegt vom Laufzeitverhalten her in der Komplexitätsklasse $O(n)$, da nur endlich viele Referenzierte Graphknoten extrahiert werden müssen. Die allgemeine Laufzeitkomplexität dieser Phase liegt somit in $O(n^2)$

Die Abbildung 5.2 zeigt zusammengefasst das gemessene Laufzeitverhalten des Systems während der Extraktion eines kürzesten Pfades aus dem Editiergraphen. Auf der Abszisse ist die Anzahl der Graphknoten zu sehen. Auf der Ordinate ist gemessene Zeit in Sekunden dargestellt. Die grüne Linie stellt den Verlauf einer quadratischen Funktion dar.

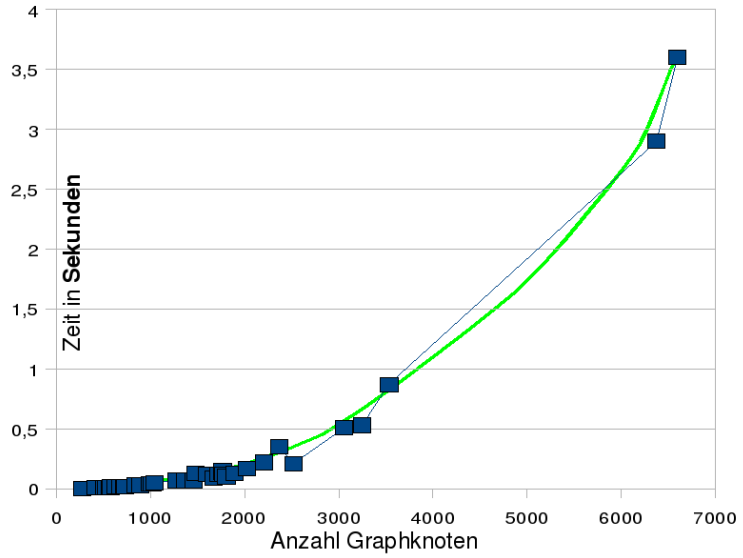

Abbildung 5.2: Dauer der Pfadextraktion

Exemplarische Nachrechnungen decken sich weitgehend mit der Annahme, dass das Laufzeitverhalten des Systems bei der Extraktion eines kürzesten Pfades aus dem Editiergraphen im „worstcase“ in der Komplexitätsklasse $O(n^2)$ liegt. Der Verlauf des entstandenen Graphen in der Abbildung 5.2 bestätigt die Annahme zusätzlich.

Dauer Postprocessing
In der letzten Phase der Berechnung wird der extrahierte Pfad in eine Editiersequenz umgewandelt, die anschließend postprocessiert wird. Die theoretische Laufzeitkomplexität der Postprocessingphase liegt in $O(n)$ ist also linear, wobei n die Anzahl der Editieroperationen einer Editiersequenz darstellt. Die Einordnung in diese Komplexitätsklasse ist dadurch berechtigt, da beim Postprocessing die Editiersequenz mit endlich vielen Editieroperationen drei mal sequentiell durchlaufen wird. Beim ersten Durchlauf wird ein Pfad in eine Editiersequenz umgewandelt. Im zweiten Durchlauf werden die Einfüge- und Löschoperationen auf Teilbäumen identifiziert und uninteressante Operationen werden markiert. Im dritten Durchlauf werden die markierten Operationen ausgefiltert.

Die Abbildung 5.3 zeigt zusammengefasst das gemessene Laufzeitverhalten des Systems während der Umwandlung und dem Nachbearbeiten der Editiersequenz. Auf der Abszisse ist die Anzahl der Editieroperationen einer Editiersequenz zu sehen. Auf der Ordinate ist gemessene Zeit in Millisekunden dargestellt. Die grüne Linie stellt den Verlauf einer linearen Funktion dar.

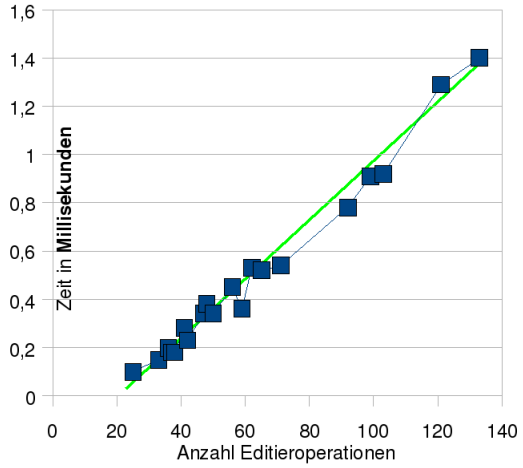

Abbildung 5.3: Dauer der Postprocessing in Millisekunden

Exemplarische Nachrechnungen decken sich weitgehend mit der Annahme, dass das Laufzeitverhalten des Systems während der Postprocessingphase im „worstcase“ in der Komplexitätsklasse $O(n)$ liegt. Der Verlauf des entstandenen Graphen in der Abbildung 5.3 bestätigt die Annahme zusätzlich.

5.3.4 Effektivität des Postprocessing

Das Postprocessing ist wichtig, um die Einschränkungen des Editiergraphansatzes zu beheben. Das eigentliche Ziel ist es, die Anzahl der Editieroperationen zu reduzieren. Schon bei relativ kleinen Typ3 Klonen mit wenigen Unterschieden fiel die Editiersequenz recht groß aus. Durch das Postprocessing wurde die Anzahl jedoch stark eingeschränkt. Die Stärke der Reduktion wurde bei der Evaluation berechnet. Der Reduktionsfaktor hängt zum Einen von der Art und zum Anderen von der Anzahl der Unterschiede ab. Wurden viele schwerwiegende Änderungen an der Kopie vorgenommen, wie z.B. zahlreiches Löschen oder Einfügen von Zeilen/Strukturen, so fällt der Reduktionsfaktor relativ gering aus. Weniger Änderungen ergeben einen großen Reduktionsfaktor. Die Abbildung 5.4 zeigt, für jeweils zehn Stichproben jedes Systems, wie sich die Anzahl der Editieroperation vor und nach dem Postprocessing verhält. Der durchschnittliche Reduktionsfaktor der Anzahl der Editieroperationen für das jeweilige Softwaresystem ist in der Tabelle 5.1 dargestellt. Die Tabelle 5.2 zeigt den durchschnittlichen Reduktionsfaktor für die verschieden Programmiersprachen und den allgemeinen durchschnittlichen Reduktionsfaktor.

System	Reduktionsfaktor
Eclipse-Ant	11
Eclipse JDT-Core	20
Javax-swing	25
GNU-Bison	10
Wget	25
SNNS	24

Tabelle 5.1: Durchschnittlicher Reduktionsfaktor (verschiedene Systeme).

Sprache	Reduktionsfaktor
Java	19
C/C++	20
Gesamt	19

Tabelle 5.2: Durchschnittlicher Reduktionsfaktor insgesamt.

Das Ausweiten der Editieroperationen Löschen und Einfügen auf Teilbäume zusammen mit der Filterung der uninteressanten Operationen schränkt die Editierdistanz, wie erwartet, recht effektiv ein.

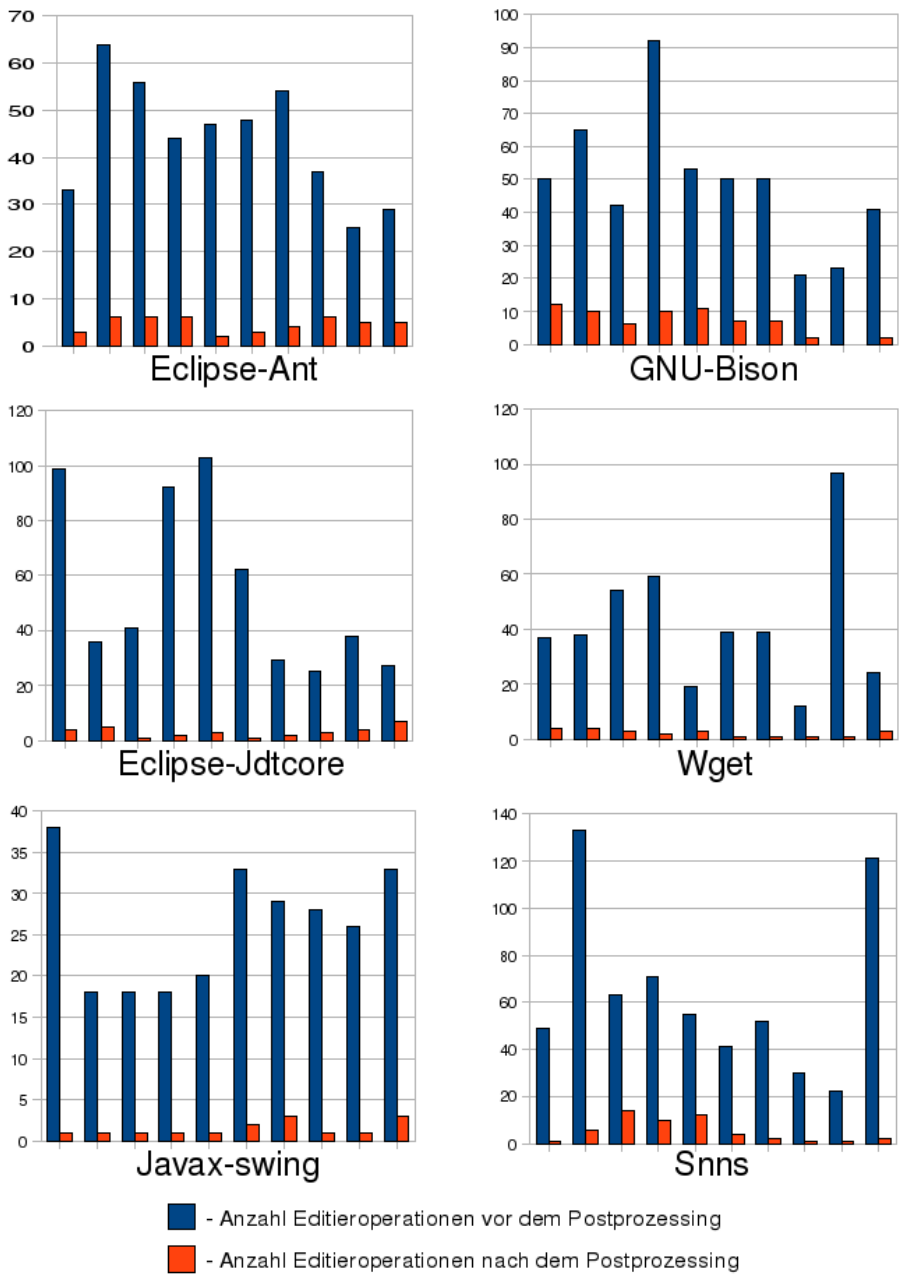

Abbildung 5.4: Reduktion der Editiersequenz

KAPITEL 6

Fazit

Inhalt

In diesem Kapitel wird die gesamte Arbeit abschließend noch einmal betrachtet. In Kapitel 6.1 werden der Kontext der Arbeit, die gestellten Aufgaben und die erreichten Ziele zusammengefasst. Dabei wird auch auf die besonderen Herausforderungen bei der Umsetzung eingegangen. Das Kapitel 6.2 enthält die Beschreibung der Verbesserungsmöglichkeiten des implementierten Systems und der Verbesserungsvorschläge. In Kapitel 6.3 werden mögliche Erweiterungen des Systems vorgestellt.

6.1 Aufgabe und Ergebnisse

Nahezu alle Softwaresysteme sind von Softwareklonen belastet, was negative Auswirkungen auf die Wartbarkeit und Verständlichkeit der Systeme zur Folge hat. Die von Softwareklonen belastete Systeme sind schwer zu pflegen und weiter zu entwickeln. Desweiteren führt das Klonen zu Fehlerfortpflanzung, worunter die Qualität der Softwaresysteme leidet. Es wäre somit vorteilhaft, Klone aus den Systemen zu entfernen. Dies ist jedoch nicht immer einfach zu bewerkstelligen. Die Softwareklone haben verschiedene Ausprägungen. Die Typ3 Klone sind besonders schwer zu entfernen, da die Kopie des originalen Codefragments verändert wurde. Vor dem Beseitigen der Typ3 Klone muss man die Unterschiede, zumindest in der Syntax, genau kennen.

In dem Kontext der Softwarewartung entstand schließlich diese Arbeit. Das ausgeschriebene Thema der Diplomarbeit war „Vergleich von Typ-3 Klonen“. Das Vergleichen von Objekten setzt Unterschiede voraus. Diese Voraussetzung wird von den Codefragmenten eines Typ3 Klonpaares voll und ganz erfüllt. Die Aufgabe wurde somit auf das Berechnen der Unterschiede zwischen den Codefragmenten eines Typ3 Klonpaares ausgeweitet.

Es galt also einen Ansatz zur Berechnung der Unterschiede zu finden und zu implementieren. Dabei musste beachtet werden, dass das implementierte System in die „Bauhaus“-Suite integrierbar sein soll. Im Gegenzug stellt „Bauhaus“ eigene Werkzeuge, Datenstrukturen und Bibliotheken zur Verfügung. Es wurde entschieden, die „Bauhaus“-Zwischendarstellung IML als Grundlage für die Berechnung zu verwenden. Diese wird aus den abstrakten Syntax Bäumen der Softwaresysteme aufgebaut, ist programmiersprachenübergreifend und sehr mächtig. Eines der herausragenden Werkzeuge der „Bauhaus“-Suite zu Identifikation von Klonen eines Softwaresystems ist ccdiml. Dieses Tool verwendet die IML, um die Klone zu identifizieren und bietet die Möglichkeit, diese in der Zwischendarstellung zu annotieren. Diese Eigenschaften erlauben sowohl den Quellcode, als auch die abstrakten Syntax Bäume der Fragmente eines Klonpaares zu extrahieren.

Zur Berechnung der Unterschiede wurde ein Verfahren ausgewählt und implementiert, das mit Hilfe eines Editiergraphen die Editiersequenz zwischen zwei geordneten Bäumen berechnet. Als Ergebnis der Berechnung entsteht eine Editiersequenz, die eine Abfolge von drei unterschiedlichen elementaren Editieroperationen beinhaltet. Dabei handelt es sich um Operationen wie Löschen, Einfügen und Substituieren von Baumknoten. Die Operationen Löschen und Einfügen sind jedoch nur auf Blattknoten erlaubt. Bei Anwendung dieser Operationen aus der Editiersequenz wird der erste Baum in den zweiten transformiert. Somit behebt die Editiersequenz die Unterschiede zwischen den beiden Bäumen. Im Kontext der Typ3 Klone fungieren die abstrakten Syntax Bäume der beiden Codefragmente eines Klonpaares als Eingabe für die Berechnung. Das Verfahren selbst wurde bei der Implementierung dahingehend angepasst und erweitert, dass es auf den aus der IML extrahierten abstrakten Syntax Bäumen arbeiten kann. Die Einschränkungen des Ansatzes werden durch das Postprocessing der berechneten Editiersequenz behoben. Desweiteren hilft das Postprocessing die Anzahl der Operationen zu verkleinern. Für das Postprocessing wurde eine Datenstruktur zur Unterbringung der Editiersequenz entwickelt. Die postprocessierte Editiersequenz wird schließlich interpretiert, um die Unterschiede zwischen den Codefragmenten in Form einer geeigneten Ausgabe dem Benutzer zugänglich zu machen.

Bei der Umsetzung gab es jedoch einige besondere Herausforderungen. So musste eine Programmiersprache von Grund auf neu erlernt werden. Ich habe bis zu dem Zeitpunkt der Implementierungsphase keinerlei Erfahrungen mit der Programmiersprache Ada95 gemacht. Eine neue Programmiersprache innerhalb der so kurzen Zeit im vollen Umfang zu erlernen, ist schlicht nicht möglich. Deshalb lässt die Qualität des Quellcodes des implementierten Systems an manchen Stellen zu wünschen übrig, da versucht wurde komplexe Sachverhalte mit einfachen, bereits bekannten Mitteln zu realisieren. Die IML ist eine sehr komplexe Struktur, die viele Jahre Entwicklungszeit hinter sich hat. Diese Zwischendarstellung zu verstehen, war ebenfalls besonders herausfordernd. Darüber hinaus gestaltete sich die Verwendung der „Bauhaus“-Bibliotheken zur Verarbeitung der IML als schwierig, da diese zum Teil spärlich dokumentiert sind.

Im Großen und Ganzen ist im Rahmen dieser Arbeit ein System entstanden, das recht einfach zu sein scheint und einige Verbesserungsmöglichkeiten aufweist (siehe 6.2), jedoch die gestellten Anforderungen erfüllt. Auch im jetzigen Zustand weist das System viel Potenzial auf und kann vielfältig eingesetzt werden (siehe 6.3).

6.2 Verbesserungsmöglichkeiten

Während der Umsetzungen und der Evaluationsphase sind einige Verbesserungsmöglichkeiten des Systems und des Ansatzes aufgedeckt worden. So stellte zum Beispiel die Anzahl der Operationen in der Editiersequenz, die als Folge der Einschränkungen in dem Editiergraphansatz entstand, ein Problem dar. Diese Verbesserungsmöglichkeit konnte jedoch durch das Implementieren des Postprocessing behoben werden. Einige Probleme konnten aufgrund der zeitlichen Gegebenheiten nicht behoben werden. Diese Verbesserungsmöglichkeiten werden in den nachfolgenden Abschnitten des Kapitels diskutiert.

6.2.1 Einfügeposition

Die elementaren Operationen sind auf Baumknoten definiert. Die Editiersequenz transformiert ein Baum in einen anderen. Das Einfügen eines Knotens erfolgt als das am weitesten rechts liegende Kind eines Elternknoten im ersten Baum. Bezogen auf das Einfügen von Teilbäumen, die ganze Zeilen repräsentieren, ist die Einfügeposition präzise. Zeilen werden stets unterhalb anderer Zeilen eingefügt. Die Ausgabe an den Benutzer gibt in diesem Fall beide Zeilen aus und gibt die Einfügeposition eindeutig an. Bei dem Einfügen von Teilstrukturen innerhalb der Zeilen lässt die Ausgabe an den Benutzer jedoch zu viel Spielraum. Muss eine Struktur innerhalb einer Zeile eingefügt werden, so besagt die Ausgabe an den Benutzer lediglich, dass eine Struktur in die Zeile hinein eingefügt werden soll, jedoch nicht die genaue Position. Um die genaue Position zu bestimmen, müssen zunächst die Kindknoten des Elternknoten, unter dem das Einfügen erfolgen soll, analysiert werden. Die Position des am weitesten rechts liegenden Kindknoten dient dann als Orientierung für das Einfügen der neuen Struktur.

6.2.2 Ausgabe an den Benutzer

Die Ausgabe an den Benutzer verwendet zum Einen die Quellcodezeilen und zum Anderen die Informationen der IML. Es werden IML-Klassenbezeichner verwendet, um auszugeben, welche Quellcodestrukturen gelöscht, eingefügt oder ersetzt werden müssen. Die IML-Klassenbezeichner korrelieren stark mit den simulierten Quellcodestrukturen, sind jedoch nicht jedem Benutzer bekannt. Einige sind nicht intuitiv und nur für Benutzer mit umfangreichen Kenntnissen der IML geeignet. Die Position der Struktur $Zeilennummer \| Spalte$ kann im Zweifelsfall zur Lokalisierung der Struktur benutzt werden, dennoch ist dies auch nicht immer sinnvoll. Die IML-Knoten der Klassen $Field\_Selection$ oder $Virtual\_Call$ modellieren zum Beispiel, im Sinne der Syntax einer objektorientierten Programmiersprache, nur ein Punkt in der Anweisung $Objektname.Feld$ oder die Stelle direkt nach dem Punkt in der Anweisung $Objektname.Methode()$. Eine Kombination zwischen IML-Klassenbezeichnern und einem Beispiel der Syntax der modellierten Struktur könnten an dieser Stelle Abhilfe schaffen.

Die Interpretation und Ausgabe der Editieroperationen erfolgt in der Reihenfolge, in der sie in der Editiersequenz vorkommen. Dies kann unter Umständen zu Verwirrung führen. Das Einfügen einer Zeile **nach** einer bereits zum Löschen freigegebenen Zeile macht wenig Sinn. Als Vorbeugungsmaßnahme könnten die Operationen der Editiersequenz vor der Interpretation sortiert werden. Alle Substitutionen werden vor dem Einfügen und alle Einfügeoperationen vor dem Löschen ausgegeben.

Bei der Berechnung und Ausgabe der Unterschiede an C/C++ Systemen fiel auf, dass teilweise Operationen auf nicht vorhandenen Strukturen vorgeschlagen wurden.

Eine besondere Eigenschaft von C/C++ Programmen sind die Makros. Beim Erzeugen der IML werden vor der Umwandlung des Quellcodes in die Zwischendarstellung die Makros von dem Frontent expandiert. Dies hat zu Folge, dass die abstrakten Syntax Bäume mancher Codefragmente die Strukturen aus den expandierten Makros beinhalten. An den Stellen dieser Strukturen sind im Quellcode jedoch die Makros selbst vorhanden. Bei der Ausgabe der Codefragmente werden diese aus Quellcodedateien ausgelesen, welche die Makros vor der Expandierung durch den Compiler beinhalten. Im Listing 6.1 ist ein Beispiel einer solchen Ausgabe dargestellt. In der Zeile 807 des Zweiten Codefragments wird ein Makro *FREE_MAYBE* verwendet. Dieses wird laut der Definition des Makros bei der Erzeugung der IML in eine *do−while* Anweisung umgewandelt. Somit beinhaltet der AST des Codefragments einen Teilbaum, der diese *do − while* Anweisung modelliert. Es kommt also zu einer falschen Ausgabe der entsprechenden Operation.

Listing 6.1: C/C++ Makros

```
————————————CLONE_1————————————
File: ftp.c
 1211   while (f) {
 1212       struct fileinfo *next = f->next;
 1213       free (f->name);
 1214       if (f->linkto)
 1215        free (f->linkto);
 1216       free (f);
 1217       f = next; } }
————————————CLONE_2————————————
File: url.c
 804   while (l) {
 805       urlpos *next = l->next;
 806       free (l->url);
 807       FREE_MAYBE (l->local_name);
 808       free (l);
 809       l = next; } }
————————————————————————————
DELETION OF ENTIRE: If_Statement STATEMENT FROM LINE
ftp.c:  1214| 7      if (f->linkto)
————————————————————————————
INSERTION OF ENTIRE: Do_While_Loop STATEMENT IN
url.c:  807| 7      FREE_MAYBE (l->local_name);
-/INTO/AFTER-LINE->:
ftp.c:  1215| 14       free (f->linkto);
————————————————————————————
```

Um diesen Umstand zu beheben, muss bei der Ausgabe der Codefragmente vor der Berechnung der Editiersequenz auf Quellcode mit bereits expandierten Makros zurückgegriffen werden. Dieser muss jedoch in einer solchen Form vorliegen. Dies ist somit weniger eine Verbesserungsmöglichkeit des Systems, sondern eher der Eingabedaten.

Eine weitere kleine Verbesserungsmöglichkeit der Benutzerausgabe besteht bei der Interpretation von Operationen auf Blattknoten. Meist handelt es sich um Knoten der Klasse *Entitiy_L_Value*, Klassen der Kategorien *Literal* aber auch Klassen der Kategorie *Label* und andere können als Blattknoten vorkommen.

Bei einer Operation auf einem der Knoten dieser Klassen wäre es neben dem Klassennamen, der die Art der Quellcodestruktur beschreibt, auch die Wertigkeit bzw. der Bezeichner von Interesse. Dies geschieht momentan lediglich für die Variablen, also Knoten der Klasse *Entitiy_L_Value*. Bei der Ausgabe der Operationen auf Knoten der Klasse *Entitiy_L_Value* wird in einem extra Schritt durch das Traversieren der semantischen *Name* Kante der Bezeichner der Variablen ausgelesen und ausgegeben. Bei Operationen auf einer Klasse der Kategorie *Literal*, zum Beispiel Ersetzen einer 3 durch 2.5, wird lediglich ausgegeben, dass ein *Int_Literal* durch ein *Float_Literal* ersetzt werden soll. Die Werte der Literale werden ausgelassen. Somit sollten für alle Klassen, die einen Bezeichner oder einen Wert modellieren, die Werte ausgelesen und mitausgegeben werden.

6.2.3 „class_tag_comparator“

Das Modul „class_tag_comparator“ ist für das Vergleichen zweier IML-Klassenbezeichner, bezüglich deren Substituierbarkeit, implementiert worden. Ein Teil der IML, das für die Modellierung der abstrakten Syntax Bäume relevant ist, wurde in disem Modul in einer statischen Datenstruktur nachgebildet. Diese Datenstruktur beinhaltet alle relevanten Klassennamen der IML-Klassenhierarchie in Form von Strings. Die Klassenhierarchie diente bei der Modellierung als Vorbild. Die Zwischendarstellung IML wird ständig weiterentwickelt. Somit müsste die statische Datenstruktur des Moduls für jede Änderung an der IML-Spezifikation angepasst werden, um die korrekte Funktionsweise des Tools zu gewährleisten. Dies ist eindeutig eine Verbesserungsmöglichkeit der Implementierung. Es kann nicht erwartet werden, dass bei jeder Änderung der IML das System angepasst werden muss. Ein alternativer Ansatz wäre, die „IML-Reflection“ Bibliotheken der „Bauhaus“-Suite für das Vergleichen zu verwenden oder die benötigten Daten direkt aus der Spezifikationsdatei der IML zu lesen. Dies würde die Beständigkeit der Moduls aufbrechen und eine dynamische Funktionsweise erlauben.

6.2.4 IML-Klassen „Field_Selection“ und „Method_Selection“

Die IML-Klassen *Field_Selection* und *Method_Selection* modellieren ein Teil der Anweisung *Objektname.Methodenname*() einer objektorientierten Programmiersprache, also Aufruf einer Methode eines Objektes. Die beiden Klassen liegen in der Kategorie *Member_Selection* und erben somit die Attribute *Operand* und *Selector*. Das Attribut *Operator* verweist auf den *Objektnamen* und das Attribut *Selector* auf den *Methodennamen* in der Anweisung *Objektanme.Methodenname*(). Das Attribut *Selector* ist jedoch semantisch und gehört in der IML somit nicht zu dem abstrakten Syntax Baum des Konstruktes. Deshalb wird er bei dem Berechnen der Editiersequenz nicht betrachtet, obwohl er eindeutig zu der Syntax und somit zu dem AST der Anweisung gehört. Durch das Begehen der semantischen Kante *Selector*, beim Linearisieren der Bäume ,vor dem Aufbauen des Editiergraphen kann dem entgegen gewirkt werden.

6.3 Einsatzgebiete und weiterführende Arbeit

In diesem Kapitel werden potentielle Einsatzgebiete und mögliche Erweiterungen des Systems beschrieben. Die Qualität des Systems, sowie die Anzahl der Einsatzgebiete kann durch die beschriebenen Erweiterungen gesteigert werden.

6.3.1 Verbesserte Ausgabe

Die Unterschiede sind in der Ausgabe sichtbar und der Benutzer kann anhand dieser Entscheidungen treffen. Die Ausgabe ist jedoch verbesserungswürdig, da die Unterschiede lediglich in textueller Form als Operationen angezeigt werden. Neben der Visualisierung der Unterschiede sind, unter Umständen, auch die Gemeinsamkeiten von Interesse. Eine mögliche Erweiterung in diesem Zusammenhang wäre, die Quellcodefragmente des Klonpaares in einem Editor anzuzeigen und eine farbige Markierung der gemeinsamen und der unterschiedlichen Teile vorzunehmen. Ein Beispiel einer solchen Ausgabe ist in der Abbildung 6.1 zu sehen.

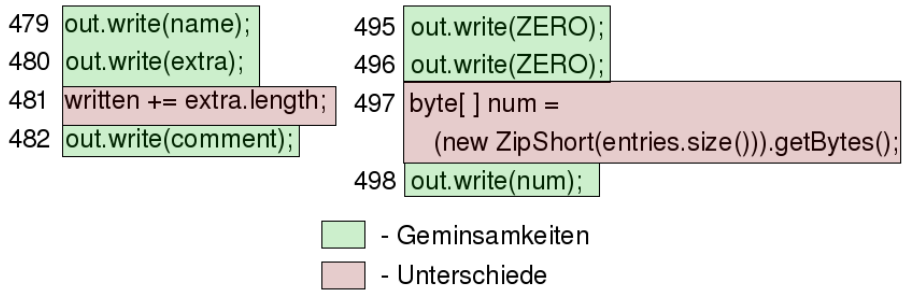

Abbildung 6.1: Erweiterung der Ausgabe

6.3.2 Semantische Interpretation

Das Postprocessing zielt zum Einen darauf ab, die Einschränkungen des Editiergraphansatzes zu beheben und zum Anderen die Anzahl an Editieroperationen durch das Filtern einzuschränken. Beim Beheben der Einschränkungen wird in der Editiersequenz nach Mustern gesucht, die das Löschen oder Einfügen von Teilbäumen darstellen. Diese Vorgehensweise kann auch dazu verwendet werden, weitere Postprocessing Schritte zu realisieren. Das Postprocessing kann dahingehend erweitert werden, dass eine Möglichkeit geschaffen wird, die Editiersequenz semantisch zu interpretieren oder neue Editieroperationen zu erkennen. Eine Operation *Verschieben* kann zum Beispiel daran erkannt werden, dass eine Teilsequenz von Editieroperationen, zunächst das Löschen einer Struktur anweist und eine weitere Teilsequenz, später oder auch früher in der Gesamtabfolge, das Einfügen der selben oder einer ähnlichen Struktur vorschlägt. Mehrere *Verschiebe*-Operationen zusammen könnten als eine Permutation der Zeilen bzw. Strukturen interpretiert werden. Ein stark vereinfachtes Beispiel der möglichen Vorgehensweise ist in der Abbildung 6.2 zu sehen. Es sind viele weitere semantische Interpretationen der Teilsequenzen denkbar.

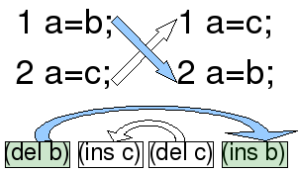

Abbildung 6.2: Erkennung der Permutation von Strukturen

6.3.3 Refactoring

Das entwickelte System kann auch jetzt schon für Refactoring der Typ3 Klone verwendet werden. Jedoch verlangt das Tool zu viel von dem Benutzer. Mit der verbesserten Ausgabe kann die Arbeit der Wartungsprogrammierer zwar weiter erleichtert werden, die eigentliche Entfernung des Klons und die dazugehörige Auswahl des Refactorings muss jedoch immer noch vollständig von dem Benutzer durchgeführt werden. Das implementierte System kann in Richtung semi-automatisches Refactoring weiterentwickelt werden. Die Entfernung der Klone mittels Zusammenführung ist eine der möglichen Optionen. Hierfür werden die gleichen Teile der Klone in eine separate Methode ausgelagert. Die unterschiedlichen Teile werden innerhalb der neuen Methode durch Fallunterscheidung behandelt. Die Unterschiede können bereits jetzt extrahiert werden. Die gemeinsamen Teile sind ebenfalls aus der Editiersequenz, noch vor dem Postprocessing, extrahierbar. Auf die gemeinsamen Fragmente deuten die langen Folgen von Substitutionsoperationen innerhalb der Sequenz. Anhand dieser Informationen kann ein Vorschlag zu Entfernung des Klones gemacht werden.

Es existieren auch weitere Ansätze, die bei dem Vorschlag zu der Zusammenführung der Klone helfen könnten. Einige sind in der Lage den größten gemeinsamen Teilbaum zweier Bäume zu berechnen (siehe [Mäkinen, 1989]). Dieser Teilbaum kann das Grundgerüst der Methode bilden, welche die Zusammenführung der Klone realisiert. Eine Erweiterung des Systems, die mit Hilfe der Kombination des Ansatzes zur Berechnung der Editiersequenz und des Baum-Isomorphie Ansatzes einen Vorschlag zum Entfernen der Klone macht, wäre durchaus denkbar.

6.3.4 Klonerkennung

In dieser Arbeit spielt die Editiersequenz eine wichtige Rolle. Diese wird dazu verwendet, die Unterschiede der Typ3 Klone zu bestimmen und auszugeben. Ein weiteres Einsatzgebiet wäre die Klonerkennung an sich. Aus der Editiersequenz resultiert die Editierdistanz. Um diese zu berechnen, muss entweder die Anzahl der Operationen oder die Summe der Kosten der Operationen bestimmt werden. Dieser Wert kann als eine Metrik für die Ähnlichkeit der Klone eingesetzt werden. Einige Tools der „Bauhaus“-Suite sind in der Lage Typ3 Klone zu identifizieren. Jedoch sind nicht alle Kandidaten für den Benutzer interessant, da viele davon nicht entfernbar sind. Anhand der Editierdistanz könnten die identifizierten Klonpaare gefiltert werden, um so bessere Kandidaten zu liefern.

Die Editiersequenz/Editierdistanz kann auch als ein Metrik basierter Ansatz zur Identifikation der Klone eingesetzt werden. Hierfür muss man jedoch vorher eine geeignete Zerlegung des abstrakten Syntax Baumes des gesamten Systems in Teilbäume finden. Diese Teilbäume können dann miteinander verglichen werden,, um Klone zu finden. Bei Typ1 und Typ2 Klonen würde die Editiersequenz nur aus Substitutionsoperationen bestehen. Enthält die Editiersequenz auch Lösch- oder Einfügeoperationen, handelt es sich um Typ3 Klone.

ABBILDUNGSVERZEICHNIS

LISTINGS

LITERATURVERZEICHNIS

[Algorithmen und Datenstrukturen] G. Saake, K. U. Sattler: Algorithmen und Datenstrukturen. Eine Einführung mit Java. Dpunkt-Verlag, Heidelberg 2004.

[Baxter, 1998] I.D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, L. Bier: Clone Detection Using Abstract Syntax Trees. In: Proceedings of the 14th International Conference on Software Maintenance IEEE CS (Veranst.), 1998.

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

[Bellon, 2007] S. Bellon: Comparison and Evaluation of Clone Detection Tools. In: IEEE Trans. Softw. Eng. 33 2007, – Rainer Koschke and Giulio Antoniol and Jens Krinke and Ettore Merlo. – ISSN 0098-5589

[Berger, 2007] B. Berger: Klonmanagement. Klonerkennung für eingebettete Systeme. Diplomarbeit 2007.

[Boehm, 1981] B. Boehm: Software Engineering Economics. Englewood Cliffs, NJ : Prentice Hall, 1981.

[Chawathe, 1999] S.S. Chawathe: Comparing Hierarchical Data in External Memory. In: Proceedings of the 25th International Conference on Very Large Data Bases, 1999.

[Falke u. a. 2008] R. Falke, P. Frenzel, R. Koschke: Empirical Evaluation of Clone Detection Using Syntax Suffix Trees. Zur Veröffentlichung eingereicht, 2008.

[Fjedstad u. a., 1979] R.K. Fjedstad, W.T. Hamlen: Application Program Maintenance Study: Report to our Respondents. In: Proceedings of the GUIDE 48. Philadelphia, PA, 1979.

[Fowler, 1999] M. Fowler: Refactoring: improving the design of existing code. Boston, MA, USA : Addison-Wesley Longman Publishing Co., Inc., 1999.

[Hauswirth 2003] M. Hauswirth, S. Dustdar, H. Gall: Software-Architekturen für verteilte Systeme: Prinzipien, Bausteine und Standardarchitekturen für moderne Software. Springer-Verlag, Berlin. 2003.

[Kamiya u. a., 2002] T. Kamiya, S. Kusumoto and K. Inoue: CCFinder: A Multilinguistic Token-Based Code Clone Detection System for Large Scale Source Code. IEEE TSE, 2002.

[Koschke u. a., 2008] C. K. Roy, J. R. Cordy und R. Koschke: Identification and Analysis of Software Clones: A Comprehensive Survey. Zur Veröffentlichung eingereicht, 2008.

[Koschke, 2007] R. Koschke: Software-Reengineering / Universität Bremen. Vorlesungsskript 2007.

[Levenstein, 1966] V. Levenstein: Binary Codes Capable of Correcting Deletions, Insertions and Reversals. In: Soviet Physics Doklady 10, 1966.

[Myers, 1986] E. W. Myers: An O(nd) differnce algorithm and its variations. In: Algorithmica 1, 1986.

[Mäkinen, 1989] E. Mäkinen: On the subtree isomorphism problem for ordered Trees. In: Inform. Process. Lett. 32, 1989.

[Plödereder u. a., 2006] R. Aoun, G. Vogel und E. Plödereder: Bauhaus - a Tool Suite for Program Analysis and Reverse Engeneering. In: Reliable Software Technologies - Ada-Europe 2006, Juni 2006.

[Selkow, 1977] S. M. Selkow: The tree-to-tree correction Problem. In: Information Processing Letters 6, 1977.

[Schober, 2007] P. M. Schober: Konzeption und Implementierung eines Interpreters für die sprachenubergreifende Programmrepräsentation IML. Diplomarbeit 2007.

[Tai, 1979] K.-C. Tai: The tree-to-tree correction Problem. In: Journal of the ACM 26, 1979.

[Tiarks, Koschke und Falke 2009] Rebecca Tiarks, Rainer Koschke, Raimar Falke: An Assessment of Type-3 Clones as Detected by State-of-the-Art Tools. Zur Veröffentlichung eingereicht, 2009.

[Valiente, 2002] G. Valiente. Algorithms on trees and graphs. Springer-Verlag, Berlin 2002.

[Wilhelm, 1981] R. Wilheln: Modified tree-to-tree correction problem. In: Inform. Prozess. Lett. 12, 1981.

[Wu u.a. 1990] S. Wu, U. Manber, G. Myers: O(np) sequence comparison algorithm. In: Inform. Prozess. Lett. 35, 1990.

[Yang 1991] W. Yang: Identifying syntaktic differences between two programms. In: Soft. Pract. Exper. 21, 1991.

[Zhang und Shasha, 1989] K. Zhang and D. Shasha: Simple fast algorithms for the editing distance between trees and related problems. In: SIAM Journal on Computing 18, 1989.

[Zhang und Shasha, 1990] K. Zhang and D. Shasha: Fast algorithms for the uni cost editing distance between trees. In: J. Algorithms 11, 1990.

[Zhang, 1995] K. Zhang: Algorithms for the cinstrained editing distance between ordered labeled trees and ralated problems. In: Pattern Recogn. 28, 1995.