Entwicklung einer Beschreibungssprache für Programmierrichtlinien

Diplomarbeit
von
Carsten Kruse

November 2002

vorgelegt am
Lehrstuhl für Programmiersprachen und Übersetzerkonstruktion
Prof. Dr. Michael Hanus
Institut für Informatik und Praktische Mathematik
Christian-Albrechts-Universität zu Kiel

Ich danke Herrn Prof. Dr. Michael Hanus für die Betreuung und seine Bereitschaft, vorliegende Arbeit zu bewerten, der Firma PPI Financial Systems GmbH für die Unterstützung der Arbeit und die Bereitstellung des Arbeitsplatzes. Herrn Dr. Christian Friberg und Herrn Dr. Wolfgang Ripken danke ich für die Betreuung in der Firma PPI. Ich danke auch Frank Huch für die kritischen Diskussionen sowie meinen Eltern, meiner Schwester und Kerstin Lindae für ihre redaktionellen Hilfestellungen.

Inhaltsverzeichnis

1 Einleitung **1**
1.1 Konkrete Problemstellung 2
1.2 Aufbau der Arbeit 2

2 Grundlegende Begriffe **4**
2.1 Bäume 4
2.2 Reguläre Ausdrücke 6
2.3 Grammatiken 7
2.4 Syntaxbäume 9
2.4.1 Konkrete Syntax 11
2.4.2 Abstrakte Syntax 11
2.5 Übersetzer 12
2.5.1 Phasen der Übersetzung 12
2.5.2 Generatoren 13

3 Programmierrichtlinien **15**
3.1 Katalog von Anforderungen an den Quellcode 15
3.2 Erhöhung der Wartbarkeit 16
3.3 Klassifizierung von Programmierrichtlinien 16
3.3.1 Formatierungsrichtlinien 16
3.3.2 Reihenfolge von Deklarationen 16
3.3.3 Namenskonventionen 17
3.3.4 Verbot/Gebot von Redundanzen 18
3.4 Überblick über bestehende Systeme 19
3.4.1 JIndent 20
3.4.2 Emacs 21
3.4.3 Centaur 22
3.4.4 Lint 23
3.4.5 Bewertung der Systeme 24

4 Automatisierte Prüfung **26**
4.1 Konzeption der Anwendung 26
4.2 Programmrepräsentation 29

4.2.1 Globale Festlegungen 29
4.2.2 Abstrakter Syntaxbaum (AST) 31
4.3 Baummuster für den AST 34
4.3.1 Definitionen 35
4.3.2 Beispiele für Überdeckungen 37
4.3.3 Priorisierung von Baummustern 39
4.4 Prüfung auf Einhaltung von Programmierrichtlinien 40
4.4.1 Programmierrichtlinien als Baummuster 40
4.4.2 Warnmeldungen und Transformationen 41
4.5 Anwendung auf die Programmierrichtlinienklassen 42
4.5.1 Formatierungsregeln 42
4.5.2 Reihenfolge von Deklarationen 47
4.5.3 Namenskonventionen 48
4.5.4 Verbot / Gebot von Redundanzen 49
4.5.5 Mischformen 49

5 Beschreibungssprache 50
5.1 Existierende Sprachen 50
5.1.1 Formalismus von JIndent 50
5.1.2 Pretty Printing Combinators von Haskell 51
5.1.3 PPML und Box 54
5.1.4 Bewertung der Ansätze 58
5.2 Aufbau der Beschreibungssprache CCML 59
5.2.1 Globaler Anteil 60
5.2.2 Regeln 61

6 Implementierung 75
6.1 Übersicht 75
6.2 Front-End 76
6.2.1 Parser 76
6.2.2 Programmrepräsentation 77
6.3 Back-End 78
6.3.1 Einlesen des XML Baumes 78
6.3.2 CCML Parser 80
6.3.3 Prüfung und Transformation 82
6.3.4 Printer 85
6.4 Demonstration der Anwendung 85

7 Analyse und Ausblick 90
7.1 Neue Ansätze dieser Arbeit 90
7.1.1 Vergleich mit programmiersprachengebundenen Konzepten 90
7.1.2 Erweiterung der programmiersprachenungebundenen Ansätze 91

7.2 Ausdrückbare Richtlinien 91
7.3 Ausblick 92
7.3.1 Hinzufügen von „syntaktischem Zucker“ 92
7.3.2 Verbot/Gebot von Redundanzen 93
7.3.3 Reihenfolge von Deklarationen 94

8 Fazit 96

Tabellenverzeichnis 97

Abbildungsverzeichnis 98

Literaturverzeichnis 100

A Grammatik der Beschreibungssprache 102

B Abstrakte Syntax für Java 105
B.1 Globale Umgebung 105
B.2 Deklarationen 105
B.3 Klassen/Interface Elemente 106
B.4 Anweisungen 107
B.5 Ausdrücke 109
B.6 Hilfsproduktionen 109
B.7 Terminale 110

C Beispielspezifikation für Java 112

D Hinweise zur Anwendung 119

Eidestattliche Erklärung 120

Kapitel 1

Einleitung

Um große Software-Projekte zu einem erfolgreichen Abschluß führen zu können, ist eine bezüglich der Zeit, Qualität und Kosten zielorientierte Entwicklung notwendig. Hinter dem Begriff Software Engineering verbirgt sich der gezielte Einsatz von Beschreibungsmitteln, Methoden und Werkzeugen für die systematische Entwicklung von Software. Ein Software Engineering Prozess untergliedert klassischerweise die Software Entwicklung in vier Phasen, beginnend mit einer Analyse und Definitionsphase über eine Entwurfsphase und anschließender Implementierungsphase bis hin zu einer Testphase.

Ein konkretes Ziel in der Implementierungsphase, das durch Software Engineering verfolgt wird, ist durch gute Lesbarkeit der Programme eine leichte Einarbeitung in eigene oder fremde Programme zu ermöglichen, um somit die Wartung und Wiederverwendung von Programmcode zu erleichtern. Eine Methode, um vorgenannte Ziele zu erreichen, ist die Formulierung eines Anforderungskataloges an den Quellcode in Form von sogenannten Programmierrichtlinien.

Die über Programmierrichtlinien gestellten Anforderungen an den Quellcode können in unterschiedliche Kategorien eingeteilt werden. Neben Programmierrichtlinien, die die Formatierung eines Programmquelltextes spezifizieren, existieren beispielsweise Richtlinien in Form von Namenskonventionen für eine aussagekräftige Namensgebung. Darüberhinaus gibt es eine Vielzahl von weiteren Festlegungen, die in Form von Programmierrichtlinien getroffen werden können.

Um eine automatisierte Prüfung und gegebenenfalls Korrekturen gemäß Programmierrichtlinien vornehmen zu können, wird in dieser Arbeit, die von der Firma PPI Financial Systems[1] unterstützt wird, eine Beschreibungssprache für Programmierrichtlinien entwickelt. Über eine Spezifikation von Programmierrichtlinien mittels der Beschreibungssprache ist eine Anwendung zur Prüfung und Korrektur von Quellcode parametrisierbar, die im Rahmen einer Fallstudie entworfen und implementiert wird.

[1] Pape und Partner Informationssysteme (PPI) Financial Systems GmbH

1.1 Konkrete Problemstellung

Ein Großteil der Softwareentwicklung der Fima PPI Financial Systems besteht aus Projektarbeiten. Konkret bedeutet dies, individuell auf die Anforderungen des Kunden zugeschnitte Problemlösungen zu erarbeiten. Damit geht der Verkauf des Quellcodes einher.

Als Teil der Projektanforderungen stellt der Kunde Anforderungen an den Quellcode in Form von Programmierrichtlinien, um diesen nach eigenen Gesichtspunkten möglichst gut wartbar und wiederverwendbar zu halten. Diese Anforderungen variieren von Projekt zu Projekt. Es existieren keine Standards, die in Abhängigkeit von der Programmiersprache ideale Programmierrichtlinien für ideal wartbaren Quellcode definieren.

Die Umstellung von Entwicklern auf einen anderen Programmierstil für ein neues Projekt ist nicht unproblematisch, zumal der einzelne Entwickler in der Regel ebenfalls einen individuellen Programmierstil bevorzugt. Zudem genügt durch die Nutzung von Entwicklungsumgebungen automatisch generierter Programmcode im allgemeinen nicht den Projektanforderungen im Sinne der Programmierrichtlinien. Wünschenswert wäre also automatisierte Prüfung und Korrektur anhand spezifizierter Programmierrichtlinien.

Die Entwicklung einer Beschreibungssprache, die eine Spezifikation von Programmierrichtlinien ermöglicht, ist vorrangige Aufgabe dieser Arbeit. Bei der Entwicklung der Beschreibungssprache wird insbesondere auf Unabhängigkeit von der zugrundeliegenden Programmiersprache, für die die Programmierrichtlinien spezifiziert werden sollen, Wert gelegt.

In der bereits angesprochenen Fallstudie wird anhand der Programmiersprache Java die praktische Umsetzung der vorgestellten Konzepte demonstriert. Zur Realisierung einer Anwendung für eine andere Programmiersprache ist aufgrund der Unabhängigkeit der Beschreibungssprache von der Programmiersprache nur ein kleiner programmiersprachenabhängiger Teil der vorgestellten Anwendung zu ersetzen.

1.2 Aufbau der Arbeit

Zunächst werden in Abschnitt 2 für diese Arbeit relevante, grundlegende Begriffe der Informatik zusammengefaßt. Im Anschluß wird in Abschnitt 3 auf die Programmierrichtlinien im einzelnen eingegangen, die im Rahmen dieser Arbeit behandelt werden sollen. Dabei wird auch ein Überblick über existierende Systeme gegeben, die verwandte Aufgaben lösen, und deren Lösungsansätze diskutiert.

Abschnitt 4 stellt theoretische Konzepte der zu entwickelnden Anwendung, der genutzten Programmdarstellung und der Prüfung auf Einhaltung von Programmierrichtlinien vor. Nachdem in Abschnitt 5 zunächst die Beschreibungsformalismen der existierenden Systeme vorgestellt werden, wird darauffolgend eine eigene Beschreibungssprache und deren Einbettung in das theoretische Konzept aus Abschnitt 4 vorgestellt. Die Implementierung der Anwendung wird schließlich in Abschnitt 6 erörtert, und damit die praktische Umsetzbarkeit der zuvor diskutierten theoretischen Konzepte gezeigt.

Außer einer Analyse der Ausdrucksfähigkeit der vorgestellten Beschreibungssprache wird in Abschnitt 7 ein Ausblick auf die Integration weiterer Beschreibungsmöglichkeiten in das vorgestellte Konzept beschrieben. Abschließend wird in Abschnitt 8 ein Fazit gezogen.

Kapitel 2

Grundlegende Begriffe

Zunächst soll auf Grundlagen aus der Informatik eingegangen werden, die insbesondere für das Gebiet des Compilerbaus relevant sind. Dabei wird hier für späteres Referenzieren auf diese Grundlagen eine Notation eingeführt, da in der Literatur die Notationen von Autor zu Autor variieren. Die im folgenden vorgestellte Notation ist in Anlehnung an [App98] gewählt.

2.1 Bäume

Die grundlegende Datenstruktur, der sich in dieser Arbeit bedient wird, ist der Baum. Im folgenden werden einige Begriffe eingeführt, die im vollen Umfang erst später bei der Prüfung von Programmierrichtlinien benögt werden. Begonnen wird mit dem Begriff des Attributs, wobei diese Definition erst bei der Definition von Baummustern (auf denen das Konzept der Prüfung auf Einhaltung von Programmierrichlinien beruht) vollständig genutzt wird.

Definition 2.1.1 (Attribut)
Ein *Attribut* ist ein Tupel (n, v) mit n Attributname und v Attributwert. Ein Attributwert v ist entweder eine Zeichenkette oder ein Tupel $(f, p_1, \ldots, p_l)$, $l \geq 0$, wobei f eine $l + 1$-stellige boolsche Funktion mit Parametern p_i, $1 \leq i \leq l$ ist. Für die Funktionsanwendung von f mit den Parametern p_i, $1 \leq i \leq l$ auf ein Argument w gilt: $f(p_1, \ldots p_l, w) \in \{true, false\}$.

Bevor der Begriff des geodneten Baumes in Definition 2.1.3 eingeführt wird, werden zunächst Knoten definiert. Bäume bestehen aus Knoten, die als Verzweigungspunkte angesehen werden können, zwischen denen es eine Vorgänger/Nachfolger-Beziehung gibt.

Definition 2.1.2 (Knoten)
Ein *Knoten* ist ein Tupel (n, A), wobei n der Knotenname und A eine Menge von Attributen ist. Insbesondere kann die Attributmenge auch die leere Menge sein, und ϵ ist der leere Knotenname.

Notation **($Name(k)$ und $Attr(k)$)**
Sei k ein Knoten.
Dann bezeichne $Name(k)$ den Namen des Knotens und $Attr(k)$ die Attributmenge von k.

Definition 2.1.3 (Geordneter Baum)
Ein *geordneter Baum* t ist ein Tupel (r, T), wobei r ein Knoten ist, der als Wurzel von t bezeichnet wird. $T = (t_1, \ldots, t_n)$, $n \geq 0$ ist ein Tupel von geordneten Bäumen. Der geordnete Baum t_i liegt links von t_j, falls $i < j$ gilt. Die Wurzeln der geordneten Bäume $t_1, \ldots, t_n$ sind die direkten Nachfolger von r, während r direkter Vorgänger der Wurzeln der geordneten Bäume $t_1, \ldots, t_n$ ist. Knoten, die keinen direkten Nachfolger haben, heißen Blätter. Alle anderen Knoten werden als innere Knoten bezeichnet.

Im folgenden wird die Menge der Teilbäume eines Baumes t definiert. Durch die Punkte (i) und (ii) der Definition 2.1.4 werden alle geordneten Bäume, die einen beliebigen Knoten von t als Wurzel und alle Knoten ab dieser Wurzel bis zu den Blättern enthalten, als Teilbäume ausgezeichnet. Nach (iii) sind auch alle geordneten Bäume Teilbäume, die aus einem Teilbaum konstruiert werden, bei denen ein beliebiger Knoten keine Nachfolger mehr hat. Im Unterschied zu (ii) müssen also nicht alle Knoten bis zu den Blättern im Teilbaum enthalten sein.

Definition 2.1.4 (Teilbaum)
Sei $t = (r, T)$ ein geordneter Baum. Die Menge M der *Teilbäume* von t ist die kleinste Menge von geordneten Bäumen mit den folgenden Eigenschaften:

(i) $t \in M$

(ii) Falls $(r', T') \in M \Rightarrow T' \subseteq M$

(iii) Falls $s \in M$ mit $s' = (r', T')$ ist ein Teilbaum von s $\Rightarrow \mathrm{s}\binom{s'}{s''} \in M$, wobei $s'' = (r', \emptyset)$ und $\mathrm{s}\binom{s'}{s''}$ bedeutet, daß in s der Teilbaum s' durch den Teilbaum s'' substituiert wird.

Das folgende Beispiel soll die Definition des Teilbaums etwas transparenter machen.

Beispiel 2.1.5
Der geordnete Baum t in Abbildung 2.1 enthält verschiedene Teilbäume.

- Nach (i) aus Defintion 2.1.4 ist t ein Teilbaum von t.
- Alle Bäume mit einer Wurzel, die direkter Nachfolger von r ist, mit den jeweiligen nachfolgenden Bäumen sind nach (ii) Teilbäume von t (z.B. der Baum bestehend nur aus Knoten a oder der Baum der Knoten j,k,l und m).

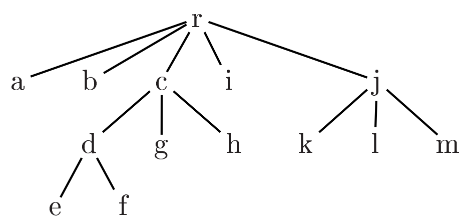

Abbildung 2.1: Beispielbaum t (ohne Attribute)

- Rekursiv betrachtet nach (ii), sind alle Bäume des Tupel T' aller Teilbäume $t' = (r', T')$ von t wiederum Teilbäume (z.B. der Baum mit den Knoten d, e und f)
- Rekursiv betrachtet nach (iii), sind alle Bäume Teilbäume, auch wenn sie nicht vollständig bis zu den Blättern alle Knoten enthalten. Von einem Knoten eines Teilbaumes müssen entweder alle oder aber kein Nachfolger im Teilbaum enthalten sein. Beispiele sind dafür der Baum mit den Knoten c, d, g und h, der Baum bestehend nur aus r oder der Baum mit allen Knoten ohne die Knoten k,l und m).

2.2 Reguläre Ausdrücke

Eine formale Sprache (also z.B. eine Programmiersprache) wird in der Informatik als eine Menge von Zeichenreihen verstanden. Die Zeichenreihen bestehen ihrerseits aus Zeichen, die einem Alphabet angehören.

Definition 2.2.1 (Alphabet)
Ein Alphabet ist definiert als eine endliche Menge von Zeichen. Zur Bezeichnung wird ein Großbuchstabe verwendet, vorzugsweise A.

Um die (möglicherweise unendlich vielen) Zeichenreihen einer formalen Sprache mit einer endlichen Beschreibung spezifizieren zu können, bedient man sich regulärer Ausdrücke. Ein regulärer Ausdruck steht dabei für eine Menge von Zeichenreihen. Nach der Defintion von regulärer Ausdrücke werden im Beispiel 2.2.3 reguläre Ausdrücke und die zugehörigen Mengen von Zeichenreihen, die durch die jeweiligen Ausdrücke beschrieben werden, gezeigt.

Definition 2.2.2 (Reguläre Ausdrücke)
Sei A ein Alphabet.
Dann ist die Menge der *regulären Ausdrücke* R(A) wie folgt definiert:

- Falls a $\in$ A $\Rightarrow$ a $\in$ R(A) (Symbole)

- $\epsilon \in$ R(A) (leerer regulärer Ausdruck)
- Falls $\alpha, \beta \in$ R(A) $\Rightarrow \alpha|\beta \in$ R(A) (Alternative)
- Falls $\alpha, \beta \in$ R(A) $\Rightarrow \alpha \cdot \beta \in$ R(A) (Konkatenation)
- Falls $\alpha \in$ R(A) $\Rightarrow \alpha^\star \in$ R(A) (Iteration)

Zur eindeutigen Zuordnung der Operatoren sind reguläre Ausdrücke klammerbar. Alle Elemente aus R(A) sind *reguläre Ausdrücke* über dem Alphabet A.

Beispiel 2.2.3
Nachfolgend stehen einige Beispiele für reguläre Ausdrücke über dem Alphabet A ={a,b}:

- a · b · c repräsentiert die Menge { "abc"}
- (a · b) | ϵ repräsentiert die Menge {"ab",""}
- ab* repräsentiert die unendliche Menge {"a", "ab", "abb", ...}

Notation
Zu den aus Definition 2.2.2 stammenden Operatoren werden zur Vereinfachung der Notation weitere Operatoren und Schreibweisen eingeführt. Diese Operatoren sind durch die elementaren Operatoren aus Definition 2.2.2 ausdrückbar und gehören somit nicht zur minimalen Menge von Operatoren zur Definition von regulären Ausdrücken.

- $\alpha\beta = \alpha \cdot \beta$ für $\alpha, \beta \in$ R(A)
- $\alpha^+ = \alpha\alpha^\star$ für $\alpha \in$ R(A) (Iteration ein oder mehrere Male)
- $\alpha? = \alpha|\epsilon$ für $\alpha \in$ R(A) (Optionales Auftreten)
- [a-z] = a | b | ...| z für a,b, ..., z $\in$ A (Aufzählung)

2.3 Grammatiken

Zur Beschreibung der Syntax von formalen Sprachen dienen sogenannte Grammatiken. Um die syntaktische Struktur von Programmiersprachen festzulegen, eignet sich die spezielle Form der kontextfreien Grammatik. Bevor dieser Begriff formal definiert wird, soll zunächst der Begriff des Tokens geklärt werden.

Definition 2.3.1 (Token)
Ein *Token* ist eine Folge von Zeichen, die eine lexikalische Einheit bildet.

Beispiel 2.3.2
Token einer Programmiersprache können wie folgt aussehen:

- if (Schlüsselwort)
- myVariable (Bezeichner)
- 42 (Literal)
- == (Operator)
- ...

Definition 2.3.3 (Kontextfreie Grammatik)
Eine *kontextfreie Grammatik* G ist ein Tupel (T, N, S, P) mit

- T: Menge von Terminalsymbolen
- N: Menge von Nichtterminalsymbolen
- S: Startsymbol
- P: Menge von Produktionsregeln der Form
  $\mathrm{A} \rightarrow \mathrm{B}_1 \dots \mathrm{B}_\mathrm{n}$; mit $\mathrm{n} \geq 0$, $\mathrm{A} \in \mathrm{N}$ und $\mathrm{B}_1, \dots, \mathrm{B}_\mathrm{n} \in \mathrm{T} \cup \mathrm{N}$

Anstatt zwischen Terminal- und Nichtterminalsymbolen zu trennen, kann man eine kontextfreie Grammatik auch als eine Menge von Symbolen und Produktionen der obigen Form auffassen. Dann sind die Terminalsymbole genau die Symbole, die nicht auf der linken Seite einer Produktionsregel auftauchen. Die Terminalsymbole sind Token gemäß Definition 2.3.1 und werden üblicherweise mit regulären Ausdrücken beschrieben. Zur Trennung von lexikalischen Einheiten innerhalb eines Programmquelltextes dienen Folgen von sogenannten Whitespace Zeichen. Zu den Whitespace Zeichen gehören bei Java beispielsweise Leerzeichen, Tabulatoren sowie Zeilen- und Seitenumbrüche. Zudem ist es möglich mit sogenannten Kommentaren Erläuterungen in den Programmquelltext einzufügen. Whitespace (Leerraum) wie auch Kommentare sind ebenfalls mit regulären Ausdrücken beschreibbar. Da aber über Umfang und Auftreten von Whitespace und Kommentaren im allgemeinen kein Einfluß auf den Programmablauf genommen wird, werden sie häufig auch als Nontokens oder Ignored Tokens bezeichnet.

Beispiel 2.3.4
Eine einfache mehrdeutige (siehe Beispiel 2.4.3) Grammatik für arithmetische Ausdrücke ist G = (T, N, S, P) mit

$\mathrm{T} = \{ 0, 1, \dots, 9, +, \star \}$,
$\mathrm{N} = \{ \mathrm{E}, \mathrm{Num}, \mathrm{Digit} \}$,
$\mathrm{S} = \{ \mathrm{E} \}$ und

P = {
  E → E '+' E ;
  E → E '⋆' E ;
  E → Num ;
  Num → Digit ;
  Num → Digit Num ;
  Digit → '0' ;
  Digit → '1' ;
  ...
  Digit → '9' ;
}.

Notation (EBNF)
Im weiteren wird zur Angabe von Grammatiken die *Extended Backus-Naur Form (EBNF)* nach [ISO96] (mit kleinen Veränderungen s.u.) verwendet. Dabei haben die einzelnen Elemente folgende Bedeutung:

- Nichtterminalsymbole beginnen mit einem Großbuchstaben.
- Terminalsymbole beginnen mit einem Kleinbuchstaben. Alternativ kann ein Terminalsymbol mit nur einer zulässigen Belegung durch die konkrete Belegung in einfachen Anführungszeichen notiert werden.
- Alternativen werden durch den Alternativ-Operator '|' getrennt.
- Durch Einfassen in die eckige Klammern '[' und ']' wird optionales Auftreten der dazwischenstehenden Elemente gekennzeichnet.
- Durch Einfassen in die geschweifte Klammern '{' und '}' wird eine beliebig häufige Wiederholung (insbesondere auch Fehlen) der eingeklammerten Elemente notiert.
- Durch Einfassen in die runden Klammern '(' und ')' können Elemente gruppiert werden.
- Eine Produktionsregel wird in der folgenden Form notiert (die spitzen Klammern stehen für Metanotation):
  <Nichtterminal> = { <Nichtterminal> | <Terminal> } ;
- Auf den Konkatenationsoperator Komma ',' aus [ISO96] wird verzichtet.

2.4 Syntaxbäume

Ein Satz gehört zu der von einer Grammatik G = (T, N, S, P) erzeugten Sprache, wenn es, mit dem Startsymbol beginnend, eine Ableitung gibt, die diesen Satz erzeugt. Der Satz $3 + 4 \star 5$ liegt in der Sprache, die von der Grammatik aus Beispiel 2.3.4 erzeugt wird, da folgende Ableitung möglich ist:

$$
\begin{array}{ll}
 & \mathrm{E} \\
\rightarrow & \mathrm{E} + \mathrm{E} \\
\rightarrow & \mathrm{E} + \mathrm{E} \star \mathrm{E} \\
\rightarrow & \mathrm{E} + \mathrm{E} \star \mathrm{Num} \\
\rightarrow & \mathrm{E} + \mathrm{E} \star \mathrm{Digit} \\
\rightarrow & \mathrm{E} + \mathrm{E} \star 5 \\
\rightarrow & \mathrm{E} + \mathrm{Num} \star 5 \\
\rightarrow & \mathrm{E} + \mathrm{Digit} \star 5 \\
\rightarrow & \mathrm{E} + 4 \star 5 \\
\rightarrow & \mathrm{Digit} + 4 \star 5 \\
\rightarrow & 3 + 4 \star 5
\end{array}
$$

Bei dieser Ableitung wurde immer das am weitesten rechts stehende Nichtterminal weiter expandiert. Man spricht hier von einer *rightmost derivation*. Analog gibt es auch eine *leftmost derivation*.

Definition 2.4.1 (Ableitungsbaum)
Indem jedes Symbol einer Ableitung mit dem Symbol verbunden wird, aus dem es abgeleitet wurde, erhält man einen *Ableitungsbaum*. Die Wurzel des Ableitungsbaumes ist das Startsymbol der Grammatik.

Ein Ableitungsbaum erfüllt die Definition 2.1.3 eines Baumes, wobei die Wurzel des Ableitungsbaumes das Startsymbol der Grammatik ist. Ein Teilbaum des Ableitungsbaumes entspricht der Expansion eines Nichtterminals, falls die Wurzel des Teilbaums mindestens einen direkten Nachfolger hat. Hat die Wurzel keinen Nachfolger, repräsentiert der Teilbaum ein Terminal und ist Blatt des Ableitungsbaumes.

Definition 2.4.2 (Mehrdeutige Grammatik)
Eine Grammatik heißt genau dann *mehrdeutig*, wenn sie für einen Satz (zwei) verschiedene Ableitungsbäume hat.

Beispiel 2.4.3
Die Grammatik aus Beispiel 2.3.4 ist mehrdeutig, da für den Satz $3 + 4 \star 5$ zwei verschiedene Ableitungsbäume existieren.

Die Bäume aus Abbildung 2.2 zeigen verschiedene Interpretationen für den Satz $3 + 4 \star 5$. Nach dem linken Baum wird zuerst die Addition und danach die Multiplikation ausgeführt, womit das Ergebnis 35 wäre. Nach dem rechten Baum werden die Operationen im umgekehrter Reihenfolge ausgeführt,

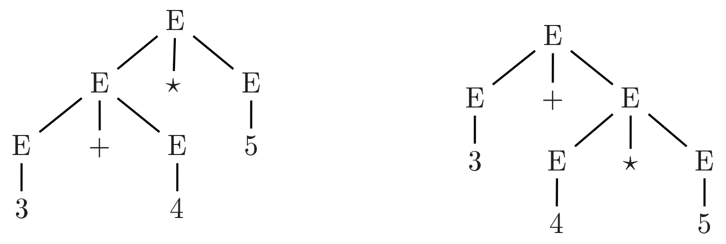

Abbildung 2.2: Zwei verschiedene Ableitungsbäume für $3 + 4 \star 5$

was ein Ergebnis von 23 ergibt. Um eine eindeutige Interpretation von Programmen zu ermöglichen, wird die Grammatik einer Programmiersprache so konzipiert, daß einem Programm eindeutig ein Ableitungsbaum zugeordnet werden kann.

2.4.1 Konkrete Syntax

Die Grammatik einer Programmiersprache definiert die konkrete Syntax der Programmiersprache.

Definition 2.4.4 (Konkreter Syntaxbaum)
Ein Ableitungsbaum, der genau ein Blatt für jedes Token und einen inneren Knoten für jede während der Ableitung benutzte Grammatikregel hat, heißt *konkreter Syntaxbaum*.

2.4.2 Abstrakte Syntax

Im allgemeinen enthält ein konkreter Syntaxbaum viel Redundanz. Zum einen kann die Grammatik Redundanz durch Produktionen enthalten, die einzig dem Zweck dienen, die Grammatik in eine bestimmte Form zu bringen, um die Definition eines Parsers (siehe 2.5.2) einfacher zu ermöglichen. Zum anderen werden über Grammatikregeln auch Operatorprioritäten ausgedrückt, die dann auch aus der Struktur des Ableitungsbaumes ablesbar sind. Ein dritter Punkt sind Redundanzen wie z.B. bei der Regel AddExp $\rightarrow$ Exp '+' Exp ;. Schreibt man einen Übersetzer (siehe 2.5) für eine konkrete Programmiersprache, so weiß der Übersetzer, daß ein AddExp aus zwei (variablen) Ausdrücken und dem Symbol '+' besteht. Es gibt also keine Notwendigkeit, im Syntaxbaum des Programmes das Symbol '+' als ein Blatt zu speichern.

Definition 2.4.5 (Abstrakte Syntax)
Die *abstrakte Syntax* einer Programmiersprache ist in Abhängigkeit der Aufgabe die, um die Redundanzen für die jeweilige Aufgabe bereinigte, konkrete Syntax der Programmiersprache.

Wichtig bei Definition 2.4.5 ist die Betonung der Abhängigkeit von der jeweiligen Aufgabe. Es gibt nicht *die* abstrakte Syntax zu einer Programmiersprache. Die in dieser Arbeit beschriebene abstrakte Syntax der Programmiersprache Java (siehe Anhang B) unterscheidet sich in vielen Punkten von der abstrakten Syntax, die ein Übersetzer für Java-Programme nutzen wird.

Definition 2.4.6 (Abstrakter Syntaxbaum (AST))
Ein Ableitungsbaum gemäß einer abstrakten Syntax zu einer Aufgabe heißt *abstrakter Syntaxbaum (AST)*.

2.5 Übersetzer

2.5.1 Phasen der Übersetzung

Ein Übersetzer (auch Compiler) übersetzt ein Programm aus einer Sprache L_1 in eine Sprache L_2. Dabei kann auch $L_1 = L_2$ gelten. Die Übersetzung gliedert sich bei modernen Übersetzern in mehrere Phasen. Eine erste Unterteilung findet man in der vereinfachten Darstellung in Abbildung 2.3.

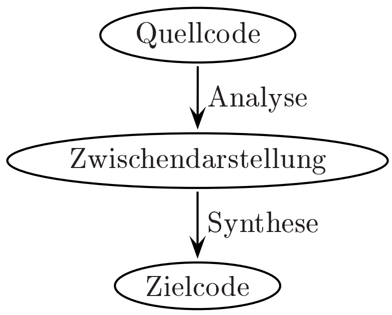

Abbildung 2.3: Unterteilung eines Übersetzers in Analyse und Synthese

Zunächst wird in der Analysephase der Quellcode der Sprache L_1 in eine Zwischendarstellung übersetzt. Diese Zwischendarstellung nutzend, wird im Anschluß in der Synthesephase in Abhängigkeit von der Aufgabenstellung Zielcode in der Sprache L_2 erzeugt. Analyse wie auch Synthese gliedern sich wieder in mehrere Phasen auf. Im folgenden soll nur die Analyse betrachtet werden, da in dieser Arbeit später eine eigene Synthese erarbeitet wird. Die Analyse gliedert sich, wie in Abbildung 2.4 gezeigt, auf.

Die Wandlung des Quellcodes in eine Tokenfolge - die sogenannte lexikalische Analyse - übernimmt der sogenannte Lexer oder Scanner. Die Tokenklassen können über reguläre Ausdrücke spezifiziert werden. Whitespace und Kommentare werden im allgemeinen während der lexikalischen Analyse verworfen; sie werden nicht in die Tokenfolge aufgenommen. Die syntaktische

Analyse aufgrund der zuvor erzeugten Tokenfolge vollzieht der sogenannte Parser anhand der Produktionsregeln der Grammatik der Sprache. Bei der semantischen Analyse erhalten die Knoten des zuvor erstellten abstrakten Syntaxbaumes beispielsweise Typinformationen, falls die zugrundeliegende Sprache getypt ist.

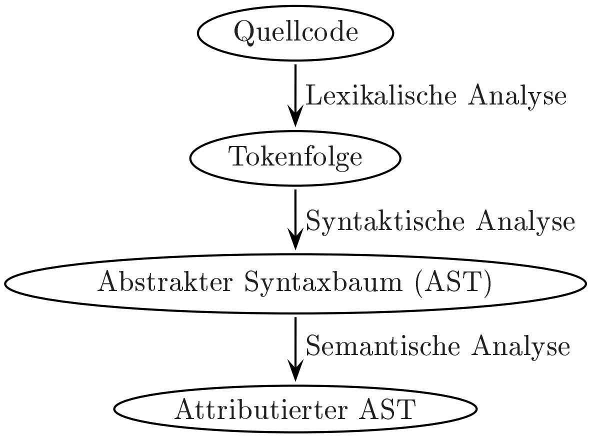

Abbildung 2.4: Phasen der Analyse

Wichtig für diese Arbeit sind lexikalische und syntaktische Analyse. Eine Darstellung des Quellprogrammes als abstrakter Syntaxbaum dient als Arbeitsgrundlage, um die gewünschten Prüfungen und Transformationen durchführen zu können.

2.5.2 Generatoren

Die Übersetzerkonstruktion ist ein Gebiet der Informatik, das sehr gut erforscht ist. Für die Konstruktion von Übersetzern gibt es in vielen Teilen standardisierte Methoden. Für einige Teilaufgaben ist eine Implementierung von Hand nicht notwendig; es existieren Werkzeuge, die gemäß einer Spezifikation den Quellcode für die jeweilige Anwendung generieren. So gibt es insbesondere Scanner- und Parsergeneratoren, die zu einer gegebenen Spezifikation einen Scanner bzw. Parser erzeugen.

Die Syntax der Spezifikationen variiert je nach verwendetem Generator. Im allgemeinen kann man die Aussage treffen, daß ein Scannergenerator Tokenspezifikationen mittels regulärer Ausdrücke benötigt, während der Parsergenerator (der implizit den Scanner miterzeugt) neben der Tokenspezifikation noch die Spezifikation der der Programmiersprache zugrundeliegenden Grammatik benötigt.

Bekannte Vertreter solcher Generatoren sind Lex und Yacc als Scanner bzw. Parsergenerator unter Unix. Ein neuerer Parsergenerator (hervorgegangen aus einer Master's Thesis [Gag98]) ist SableCC [1], der zu einer gegebenen Spezifikation Javacode erzeugt.

[1] siehe auch: http://www.sablecc.org

Kapitel 3

Programmierrichtlinien

In diesem Kapitel sollen Programmierrichtlinien grundlegend eingeführt und deren Relevanz in der professionellen Softwareentwicklung untermauert werden. Dazu werden zunächst in Abschnitt 3.1 Programmierrichtlinien als ein Katalog von Anforderungen an den Quellcode definiert. Im Anschluß wird in Abschnitt 3.2 auf das Ziel der Erhöhung der Wartbarkeit des Quellcodes auf der einen Seite und die Schwierigkeiten bei der Sicherstellung der Einhaltung auf der anderen Seite eingegangen. Um einen Überblick über die verschiedenartigen Anforderungen zu bekommen, die durch Programmierrichtlinien an den Quellcode gestellt werden können, wird in Abschnitt 3.3 eine Kategorisierung in Programmierrichtlinienklassen vorgenommen. Abschließend werden bereits verfügbare Werkzeuge für die Softwareentwicklung betrachtet, die Entwickler unterstützen, von Programmierrichtlinien geforderte Standards einzuhalten.

3.1 Katalog von Anforderungen an den Quellcode

Programmierrichtlinien sind ein Katalog von Anforderungen an den Quellcode eines Programmes. Dabei werden Aussehen und Auftreten von Elementen der Programmiersprache in einem Programm reglementiert. Solche Regeln gibt es für unterschiedlichste Programmiersprachen, unabhängig davon, ob diese aus dem imperativen, funktionalen oder logischen Bereich stammen.

Für eine Programmiersprache gibt es nicht genau den Satz von Konventionen, der den einzig guten Programmierstil spezifiziert. Vielmehr variieren die Vorstellungen von „schönen" Programmen nahezu beliebig. So stellt die Firma Sun Microsystems im World Wide Web ein zwanzigseitiges Dokument [Mic99] von Programmierrichtlinien zur Verfügung. Diese unterscheiden sich in vielen Punkten deutlich von den firmeninternen Richtlinien der Firma PPI Financial Systems.

3.2 Erhöhung der Wartbarkeit

Oberstes Ziel bei der Formulierung von Programmierrichtlinien ist eine Erhöhung der Wartbarkeit des Quellcodes. Über die Formulierung der Anforderungen an den Quellcode soll ein immer gleiches Auftreten und Aussehen von Konstrukten der Programmiersprache erreicht werden.

Im Sinne der Arbeit bei der Firma PPI Financial Systems unterscheiden sich die Anforderungen an den Quellcode von Projekt zu Projekt. Das Sicherstellen der Einhaltung von Programmierrichtlinien ist aus zweierlei Gründen schwierig:

- Die Entwickler müssen sich bei jedem neuen Projekt umstellen und haben dazu noch einen eigenen bevorzugten Programmierstil.
- Automatisch generierter Programmcode (von Entwicklungswerkzeugen wie z.B. JBuilder) genügt in der Regel nicht den gestellten Anforderungen.

Da der Verkauf des Quellcodes Bestandteil der Projektarbeiten ist, ist eine automatisierte Prüfung auf Einhaltung von Programmierrichtlinien äußerst wünschenswert.

3.3 Klassifizierung von Programmierrichtlinien

Über Programmierrichtlinien können verschiedenste Anforderungen an den Quellcode formuliert werden. Die nachfolgende Auflistung ist nicht als abschließender Katalog zu verstehen; es sollen hier die im Hinblick auf diese Arbeit betrachteten Klassen von Anforderungen vorgestellt werden.

3.3.1 Formatierungsrichtlinien

Die größte zu behandelnde Klasse ist die Klasse der Formatierungsrichtlinien. Im wesentlichen werden hier Festlegungen zur Verteilung von Whitespace (Leerraum), der im beliebigen Umfang zwischen den lexikalischen Token im Programmquelltext auftreten kann, getroffen. Eine genauere Aufteilung zeigt die Übersicht in Tabelle 3.1, die zu den einzelnen Kategorien Beispiele für Programmierrichtlinien in Java enthält.

3.3.2 Reihenfolge von Deklarationen

Diese Klasse von Programmierrichtlinien beschäftigt sich mit Reihenfolgen der Deklaration von z.B. Variablen oder Klassenelementen (bei objektorientierten Programmiersprachen). Gängig ist beispielsweise die Forderung (bei getypten Programmiersprachen), Variablendeklarationen in einer definierten Ordnung nach Typen zu sortieren. Semantisch macht das für das

	Klasse	Beispiel einer Richtlinie
1.	**Verteilung von Leerraum**	
	Leerzeilen	Anzahl der Leerzeilen zwischen zwei Methoden
	Gebot von Leerzeichen	Ein Leerzeichen vor und hinter einem binären Operator
	Verbot von Leerzeichen	Kein Leerzeichen vor einem Semikolon oder einem Komma
	Gebot von Zeilenumbrüchen	Nur eine Anweisung pro Zeile
	Verbot von Zeilenumbrüchen	Kein Zeilenumbruch nach schließender geschweifter Klammer vor einem 'else'
2.	**Einrückung**	
	Einrückeinheit	Größe einer Einrückeinheit
	Einrückungszeichen	Verbot von Tabulatorzeichen
3.	**Maximale Zeilenlänge**	
	Angabe der max. Zeilenlänge	Max. Zeilenlänge ist 80 Zeichen
	Definition von optionalen Zeilenumbrüchen	Umbruch nach einem Komma, Parameter untereinander

Tabelle 3.1: Formatierungsrichtlinien

Programm keinen Unterschied, solange es keine Querbezüge zwischen den Variablen gibt. Das ist der Fall, wenn eine Variable bei ihrer Deklaration initialisiert und in einer weiteren Deklaration diese Variable zur Initialisierung verwendet wird.

Unproblematischer ist z.B. das Sortieren von Methoden einer Java-Klasse nach Sichtbarkeit. Diese Vertauschungen verändern in keinem Fall die Semantik des Programmes. Üblicherweise wird genau eine solche Sortierung von Klassenelementen nach Art und Sichtbarkeit gefordert.

3.3.3 Namenskonventionen

Die in Rahmen einer Programmiersprache vergebbaren Bezeichner können sich auf unterschiedliche Elemente der Programmiersprache beziehen. Des weiteren kann man z.B. Variablenbezeichner auch je nach Typ der Variable in Klassen einteilen. Weit verbreitet sind Konventionen, die vorschreiben, nach welchem Muster Bezeichner benannt werden müssen. Gängig ist beispielsweise die sogenannte *ungarische Notation*, entwickelt von dem Ungarn Charles Simonyi (siehe [CS91]), für C, C++ und weitere Programmiersprachen. Mittels ungarischer Notation ist der Typ einer Variablen aus dem Namen über ein Prefix ablesbar (siehe Tabelle 3.2).

Bei der Programmiersprache Java sind Namenspräfixe nach ungarischer

Type	Namenspräfix	Beschreibung
global	g	globale Variable
pointer	p	Zeiger
array	a	Feld
struct	t	Instanz einer Struktur
function	x	Funktion, nur sinnvoll zusammen mit p als px
void	v	nichts bzw. unspezifiziert
char	c	ein Byte = ein Zeichen
uchar	uc	ein Byte = 0...255
schar	sc	ein Byte = -127...128
string	sz	0 terminierter String
bool	b	TRUE oder FALSE
ushort	us	zwei Byte= 0...65535
sshort	ss	zwei Byte= -32767...32768
uint	ui	vier Byte= 0...4294967296 (evtl. auch zwei)
sint	si	vier Byte= 2147483647...-2147483648 (evtl. auch zwei)
ulong	ul	vier Byte= 0...4294967296 (evtl. auch acht)
slong	sl	vier Byte= 2147483647...-2147483648 (evtl. auch acht)
float	f	Fließkomma, einfache Genauigkeit
double	d	Fließkomma, meist doppelte Genauigkeit

Tabelle 3.2: Namenspräfixe bei ungarischer Notation

Notation aufgrund eines übersichtlicheren Typsystems zumeist nicht erwünscht. Statt dessen gibt es Richtlinien, die fordern, daß Klassennamen groß- und Methodennamen kleingeschrieben werden, sowie sich innerhalb eines Namens anschließende Wörter mit einem Großbuchstaben beginnen.

3.3.4 Verbot/Gebot von Redundanzen

Der Konstruktor der direkten Basisklasse wird bei Java über das Schlüsselwort super aufgerufen und kann nur als erste Anweisung im Konstruktor stehen. Fehlt diese Anweisung wird sie vom Java Compiler ergänzt. Die Angabe ist also redundant.

Der explizite Aufruf des Konstruktors der direkten Basisklasse ist nur ein Beispiel für Redundanz. Es existieren Richtlinien, die im Falle solcher redundanter Elemente die explizite Angabe fordern oder verbieten. Im Beispiel des Aufrufs des Konstruktors der Basisklasse, kann eine Forderung

Type	**Richtlinie**	**Beispiel**
Klasse	Hauptwort beginnend mit Großbuchstaben	Circle, Vector
Schnittstelle	wie Klassenname (kein Präfix I !)	
Methode	Umfassen ein Verb beginnend mit Kleinbuchstaben; innere Wörter beginnen mit Großbuchstaben	getProperty, setColor
Variable	Beginnen mit Kleinbuchstaben; innere Wörter beginnen mit Großbuchstaben	drawColor
Schleifenvariable	Vorrangig i, j, k, m und n	
Konstanten	Nur Großbuchstaben; innere Wörter durch Unterstrich abtrennen	MAX_WIDTH

Tabelle 3.3: Beispiele für Namenskonventionen in Java

darin bestehen, daß der Aufruf explizit angegeben werden muß, oder daß die explizite Angabe verboten ist.

Ein weiteres Beispiel für die Forderung von Redundanz, kann die Programmierrichtlinie sein, daie bei einem „if-Statement" im „then"-Teil stets einen Block fordert. Bei Programmen der Programmiersprache Java ist bei einer einzelnen Anweisung im „then"-Teil die Angabe von Blockklammern um diese Anweisung redundant.

3.4 Überblick über bestehende Systeme

Nachdem die für diese Arbeit relevanten Programmierrichtlinien vorgestellt wurden, soll zunächst ein Überblick über bestehende Systeme gegeben werden, deren Anliegen die Erhöhung der Codesicherheit ist. Im wesentlichen lassen sich die Systeme in die Kategorien Pretty Printer bzw. Code Checker einordnen. Ein Pretty Printer dient zur Formatierung von Quellcode anhand gewisser Regeln. Ein Code Checker hingegen führt Codeanalysen durch, die über die Prüfungen des Compilers hinausgehen.

Für Pretty Printer existiert neben einer Vielzahl von programmiersprachengebundenen Ansätzen auch eine Reihe von Ansätzen, einen Pretty Printer unabhängig von einer Programmiersprache zu definieren. Bereits 1980 präsentiert D.C. Oppen in [Opp80] Algorithmen zur Realisierung eines Pretty Printers, der unabhängig von der zugrundeliegenden Programmiersprache ist. Dabei beschreibt Oppen eine Trennung in ein programmiersprachenabhängiges Front-End (erzeugt einen abstrakten Syntaxbaum) und ein programmiersprachenunabhängiges Back-End (beinhaltet den Pretty Printer, der als Arbeitsgrundlage den abstrakten Syntaxbaum nutzt). Diese Trennung findet sich auch in der vorliegenden Arbeit wieder (siehe Abschnitt 4.1). Die Ideen von Oppen verarbeitend, entwickeln A. Conchon und E. Morcos-Chounet in [EMC86] den Formalismus PPML[1] (siehe Abschnitt 5.1.3) zur Spezifikation von Formatierungsregeln für Pretty Printer, da der Pretty Printer von Oppen im Hinblick auf die Formatierungen nicht parametrisierbar ist. PPML fließt als Spezifikationssprache für einen Pretty Printer in das Centaur-System ein (siehe Abschnitt 3.4.3).

Eine programmiersprachenabhängige Realisierung eines Pretty Printers, der kommerziell vertrieben wird, ist JIndent für die Programmiersprache Java. Neben dem Pretty Printer stellt JIndent weitere Funktionen bereit, die auch andere Programmierrichtlinien als reine Formatierungsregeln umsetzen (siehe Abschnitt 3.4.1). Die Formalismen, die JIndent benutzt, um Programmierrichtlinien zu beschreiben, werden in Abschnitt 5.1.1 näher betrachtet.

Zwischen den beiden vorhergenannten Pretty Printern liegt im Sinne der Abstraktion von der Programmiersprache der Editor Emacs. Dieser bietet dem Programmierer die Möglichkeit, Programmquelltexte zu formatieren, wobei versierte Benutzer auf die Art der Formatierung Einfluß nehmen können. Wieweit bei Emacs die Mechanismen zur Abstraktion von der Programmiersprache gehen, und es somit ermöglicht wird, für eine neue Sprache einen Pretty Printer zu definieren, wird in Abschnitt 3.4.2 diskutiert.

Neben den vorgenannten Beispielen wird in Abschnitt 3.4.4 der Code Checker Lint vorgestellt, womit auch eine Abgrenzung zu den in den vorhergehenden Abschnitten genannten Programmierrichtlinien getroffen werden soll. Programmierrichtlinien, die Analysen ähnlich wie bei Lint bedingen, sind nicht Bestandteil dieser Arbeit.

3.4.1 JIndent

JIndent[2] von der deutschen Firma Software & Solutions ist nicht nur Pretty Printer für Java Programme, sondern ermöglicht auch die Spezifikation ausgewählter Programmierrichtlinien der Klasse Verbot/Gebot von Redundanzen. Ca. 160 Parameter, die durch den Benutzer gesetzt werden können, steuern im wesentlichen die Formatierung des Programmcodes (siehe Abschnitt 5.1.1). Dabei hat jeder Parameter eine genau definierte Wirkung

[1] Pretty Printing Meta Language
[2] http://www.jindent.com

für eine spezielle Stelle in der Java-Syntax. Es können also Programmierrichtlinien im Sinne der 160 Parameter spezifiziert werden, durch die die gängigen Java-Formatierungen erzielt werden können. Ebenfalls können mit JIndent Richtlinien umgesetzt werden, die Elemente im Sinne von Gebot bzw. Verbot von Redundanzen einfügen oder entfernen. So kann durch Setzen bestimmter Parameter gefordert werden, daß im „then"- und „else"-Teil einer „if"-Anweisung immer ein Block stehen muß. Gegebenenfalls werden also Blockklammern um eine Einzelanweisung im „then"- bzw. „else"-Teil einer „if"-Anweisung ergänzt oder entfernt.

Das Hauptanliegen von JIndent ist die Umsetzung von Formatierungsrichtlinien für Java. Diese können nur dann umgesetzt werden, wenn sie sich im Rahmen eines gängigen Programmierstiles befinden. Einzelne spezielle Verbote bzw. Gebote von Redundanzen sind auch mit JIndent spezifizierbar. Die Reihenfolge von Deklarationen sowie Namenskonventionen können mit diesem Tool nicht geprüft werden.

3.4.2 Emacs

Emacs[3] ist ein freier, erweiterbarer Echtzeit-Editor, der aus dem Gnu-Projekt stammt. Nützliche, unterstützende Eigenschaften von Emacs bei der Softwareentwicklung sind das farbige Hervorheben von syntaktischen Elementen (Syntax Highlighting) und automatische Einrückung.

Eine Abstraktionsebene, die hier eingeführt wurde, ist die des Syntaxtables (siehe [BL02]), die im folgenden beschrieben wird. Hierauf aufsetzend, muß das Syntax Highlighting und die automatische Einrückung für die jeweilige Programmiersprache per Hand implementiert werden. Allerdings existieren für die meisten in Lehre und Praxis verwendeten Sprachen bereits frei verfügbare Implementierungen.

Syntaxtables ordnen jeden Buchstaben des ASCII-Zeichensatzes einer der syntaktischen Klassen der Tabelle 3.4. Bei jedem Buchstaben sind neben dieser Zuordnung zur syntaktischen Klasse auch die Flags aus Tabelle 3.5 setzbar. Im wesentlichen dienen die Flags zur Spezifikation zweier alternativer Kommentarstile, die mit einer Sequenz aus zwei Zeichen beginnen, wobei das erste Zeichen der kommentareinleitenden Sequenz beider Stile identisch sein muß. Die Anfangsfolge des Alternativstils wird definiert, wenn ein Zeichen gleichzeitig 'b' (alternativer Kommentarstil) und '2'-Flag (für das zweite Zeichen einer Kommentaranfangssequenz) hat. Im Falle von C++ sieht eine gültige Zuordnung der Zeichen für die Kommentarstile wie folgt aus[4]:

'/'	.124b
'*'	.23

[3] http://www.gnu.org/software/emacs/emacs.html
[4] vgl.[BL02], S.715f

Name	Kodierung	Beschreibung
Whitespace	’ ’ oder ’-’	Whitespace Zeichen
Word	’w’	Buchstaben (z.B. für Variablennamen)
Symbol	’_’	Symbole, die in Variablennamen verwendet werden dürfen
Punctation	’.’	Operatoren
Open Parenthesis	’(’	Öffnende Klammer (Blockanfang)
Close Parenthesis	’)’	Schließende Klammer (Blockende)
String Quote	’”’	Stringbegrenzung
Escape	’\’	Anfang einer Escape-Sequenz
Character Quote	’/’	Anfang einer Einzeichen Escape-Seq.
Paired Delimiter	’$’	Bereichsmarkierung (z.B. Math Mode in Latex)
Expression Prefix	’/”	Zeichen von vorangestellten syntaktischen Operatoren (z.B. ’ für quote in LISP)
Comment Starter	’<’	Kommentaranfangszeichen
Comment Ender	’>’	Kommentarendezeichen
Inherit	’@’	Geerbtes Verhalten vom allgemeineren Syntax Table

Tabelle 3.4: Kodierung der syntaktischen Klassen bei Syntaxtables

newline >b

Damit erhält man folgende Kommentarstile:

- Stil a: Beginnt mit /∗ und endet mit ∗/
- Stil b: Beginnt mit // und endet mit einem Zeilenumbruch

Mit dem aktuellen Syntaxtable werden die Parsing- und die Word-Motion-Funktionen von Emacs gesteuert. Diese dienen zum Erkennen von lexikalischen Einheiten (Token), geklammerten Ausdrücken und zur Klassifikation von komplizierteren syntaktischen Einheiten. Diese Funktionen werden im allgemeinen zur Implementierung von Funktionen wie Syntax Highlighting und automatische Einrückung genutzt.

3.4.3 Centaur

Ziel des Centaur-Systems [BCD$^+$89], entstanden aus dem EG-Projekt GIPE[5] (1984-1989), ist die automatische Generierung von Programmierentwicklungsumgebungen aus einer syntaktischen und semantischen Spezifikation einer Programmiersprache. Eine Entwicklungsumgebung beinhaltet einen

[5] Generation of Interactive Programming Environments

Flag	Beschreibung
'1'	Erstes Zeichen einer 2-Zeichen Kommentaranfangsfolge
'2'	Zweites Zeichen einer 2-Zeichen Kommentaranfangsfolge
'3'	Erstes Zeichen einer 2-Zeichen Kommentarendefolge
'4'	Zweites Zeichen einer 2-Zeichen Kommentarendefolge
'b'	Alternativer (zweiter) Kommentarstil b

Tabelle 3.5: Flags bei Syntaxtables

in graphische Entwicklungsoberflächen integrierten Interpreter sowie einen Debugger. Des weiteren wird ein Pretty Printer aufgrund einer Spezifikation in der Beschreibungssprache PPML [EMC86] generiert. PPML ermöglicht die Definition von Formatierungen eng an der abstrakten Syntax der jeweiligen Programmiersprache.

Mit dem Centaur-System ist es also möglich einen Pretty Printer für eine beliebige Programmiersprache zu definieren, indem die Formatierungen über die Beschreibung durch PPML spezifiziert werden. In Abschnitt 5.1.3 werden die Mechanismen von PPML genauer beschrieben und im weiteren die Ideen von PPML in der eigenen Sprachentwicklung verarbeitet.

3.4.4 Lint

Aus dem Bereich der Code Checker stammt das Unix Werkzeug Lint (siehe [Joh78]). Es dient zur weitergehenden Sicherstellung von Codesicherheit für C Programme über den Compiler hinaus, die aus Gründen der Laufzeit des Compilers nicht vom C Compiler selber vorgenommen wird. Im einzelnen kümmert sich Lint um folgende Aspekte:

- Variablen und Funktionen, die nie verwendet werden
- Typinkonsistenzen, z.B. bei Zuweisungen
- Nicht erreichbare Programmfragmente
- Verwendung von Variablen ohne vorherige Zuweisung
- Schleifen, die nicht nach dem letzten Statement verlassen werden können oder in die nicht beim ersten Statement hineingesprungen werden kann
- Ungültige Rückgaben einer Funktion
- Ungewöhnliche Typumwandlungen
- Konstrukte, die abhängig von der Maschinenarchitektur unterschiedlich interpretiert werden.

- Veraltete Schreibweisen (z.B. =- statt -=)
- Diverse legale aber sonderbare Ausdrücke (z.B. *p++;)

Lint gibt bei Verletzung der genannten Punkte Warnungen aus, aber nimmt keine Programmtransformationen vor. Sinnvollerweise wird Lint innerhalb großer C Projekte eingesetzt, um frühzeitig potentielle Fehlerquellen zu detektieren.

3.4.5 Bewertung der Systeme

Die vier vorgestellten Systeme stehen mehr oder weniger im Zusammenhang mit dem Ziel dieser Arbeit, mittels einer Spezifikation von Programmierrichtlinien in einer programmiersprachenunabhängigen Beschreibungssprache automatisierte Prüfungen auf Einhaltung der Programmierrichtlinien vornehmen zu können. Im folgenden wird eine Bewertung der Konzepte bezüglich der Verwendbarkeit in dieser Arbeit vorgenommen.

JIndent behandelt von allen Systemen die meisten der vorgestellten Programmierrichtlinien, ist aber programmiersprachenabhängig. Dazu kommt das Problem, daß Formatierungen von JIndent im wesentlichen gemäß gängiger Programmierstile umgesetzt werden. Entspricht eine Formatierungsrichtlinie nicht einem der gängigen Programmierstile, ist es möglich, daß die Programmierrichtlinie mit der Parameterisierung von JIndent nicht mehr formulierbar ist. Die Bewertung der Konzepte der Formalismen, die JIndent zugrunde liegen, wird in Abschnitt 5.1.4 vorgenommen, nachdem nochmals näher die Formulierung der Parameter betrachtet wurde.

Die Abstraktionsmechanismen von einer konkreten Programmiersprache, die Emacs mit den Syntaxtables bietet, stoßen bei der Nutzung durch beliebige Programmiersprachen schnell an ihre Grenzen. Das wird schon beim Syntax-Highlighting der Programmiersprache Haskell deutlich, da Haskell nicht die Restriktion erfüllt, daß die beiden Kommentarstile mit einer mehrere Zeichen umfassenden Kommentaranfangssequenz mit demselben Zeichen beginnen. Als Resultat gerät Emacs bei der Klassifizierung von Strings und Kommentaren in einem Haskell Programm in bestimmten Situationen durcheinander und trifft falsche Zuordnungen. Die Konzepte von Emacs sind damit für diese Arbeit nicht nutzbar.

Wie in Abschnitt 3.4 einleitend erwähnt, wurde das System Lint zur Abgrenzung vorgestellt. Die Prüfungen, die Lint vornimmt, dienen zweifelsohne zur Erhöhung der Codesicherheit. Damit verfolgt Lint ein Ziel, das von anderer Seite auch durch die vorgestellten Programmierrichtlinien erreicht werden soll. Idealerweise werden in einem Softwareentwicklungsprozess beide Arten von Prüfungen vorgenommen, um möglichst hohe Codesicherheit zu erreichen.

Dem Ziel der Arbeit, über einen programmiersprachenunabhängigen Formalismus die Formulierung von Programmierrichtlinien zu ermöglichen, um dann mittels einer automatisierten Prüfung und Transformation programmierrichtlinienkonformen Programmquelltext zu erhalten, kommt das Centaur-System am nächsten. Hier wird über Spezifikationen eine Programmierentwicklungsumgebung für eine beliebige Programmiersprache generiert. Von besonderem Interesse ist die Generierung des, in diese Entwicklungsumgebung integrierten, Pretty Printers. Um das Verhalten des Pretty Printers zu spezifizieren, wird sich der Spezifikationssprache PPML (siehe Kapitel 5.1.3) bedient. PPML dient als Grundlage der in dieser Arbeit zu entwickelnden Beschreibungssprache für Programmierrichtlinien (siehe Kapitel 5.1.4).

Kapitel 4

Automatisierte Prüfung

Zunächst wird in Abschnitt 4.1 die Konzeption der Anwendung zur automatisierten Prüfung auf Einhaltung von Programmierrichtlinien vorgestellt. Darauffolgend wird in Abschnitt 4.2 auf die Programmrepräsentation eingegangen, die die Schnittstelle zwischen programmiersprachenabhängigen Front-End und programmiersprachenunabhängigen Back-End bildet. Nach der Einführung von grundlegenden Definitionen in Abschnitt 4.3 wird im Anschluß in Abschnitt 4.4 das theoretische Konzept der Prüfung auf Einhaltung von Programmierrichtlinien vorgestellt. Abschließend wird in Abschnitt 4.5 die Umsetzung des theoretischen Konzeptes im Hinblick auf die in Abschnitt 3.3 vorgestellten Klassen von Programmierrichtlinien diskutiert.

4.1 Konzeption der Anwendung

Die Kästen mit durchgezogener Linie in Abbildung 4.1 zeigen die einzelnen Teile der Anwendung. Fett gedruckt in gestrichelten Kästen sind die Ein- und Ausgaben der Anwendung, während die nicht fett gedruckten gestrichelten Kästen für Zwischendarstellungen des zu prüfenden Programmquellcodes und die Spezifikation der Grammatik als Eingabe für den Parsergenerator stehen.

Insgesamt teilt sich die Anwendung in ein programmiersprachenabhängiges Front-End und ein programmiersprachenunabhängiges Back-End. Das Front-End umfaßt lediglich den Parser zur Wandlung des Quellcodes in die Zwischendarstellung. Zum Back-End, das die Zwischendarstellung des Quellcodes als Eingabe hat, gehört neben Prüfung und Transformation auch der Printer. Die einzelnen Elemente der Anwendung werden im folgenden erklärt.

Programmquelltext

Eine Eingabe der Anwendung ist das auf Einhaltung von Programmierrichtlinien zu prüfende Programm.

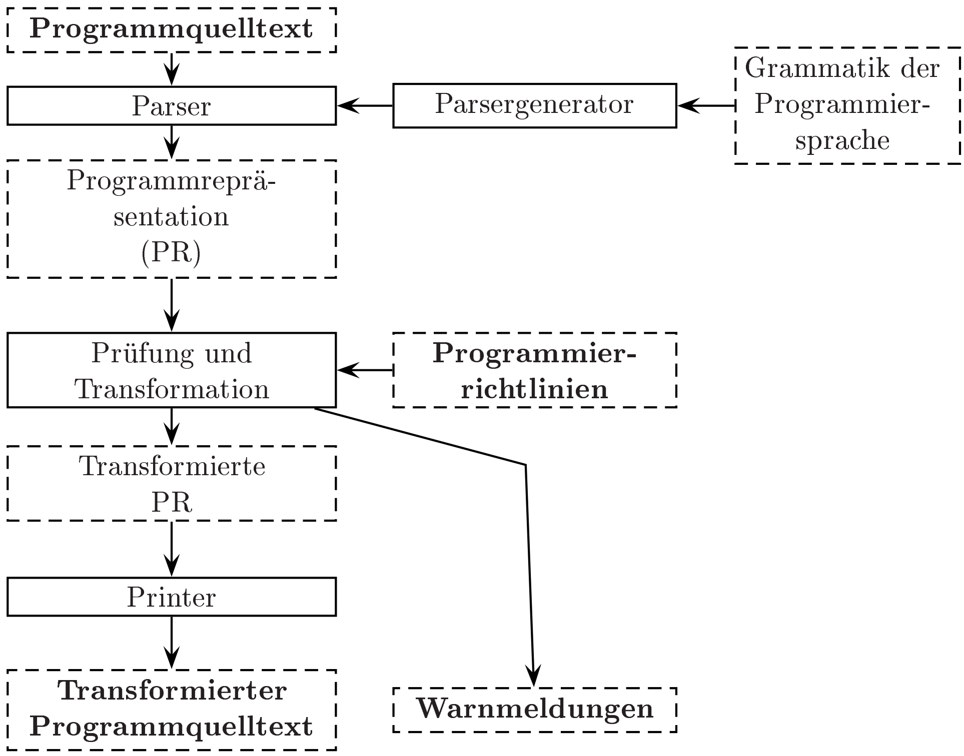

Abbildung 4.1: Ablauf der Anwendung

Parser

Zur Prüfung von Programmierrichtlinen ist eine rein lexikalische Analyse des Programmquelltextes unzureichend, da lexikalische Einheiten (Token) in verschiedenen Kontexten auftreten können. Um diese unterscheiden zu können, ist eine syntaktische Analyse des Quelltextes notwendig. Zur syntaktischen Analyse von Programmen dient ein Parser (siehe 2.5).

Parsergenerator und Grammatik

Der Parser kann mit Hilfe eines Parsergenerators (siehe 2.5.2), der als Eingabe eine Spezifikation der Syntax der Programmiersprache benötigt, generiert werden. Anders als bei der Konstruktion eines Compilers für die jeweilige Programmiersprache, bei der Whitespace-Zeichen und Kommentare während der lexikalischen Analyse übergangen werden, ist es hier wichtig Formatierungen und Kommentare zu bewahren. Also muß die Spezifikation für den Parsergenerator so erweitert werden, daß der Parser Whitespace-Zeichen und Kommentare erkennt und behält.

Programmrepräsentation

Die Ausgabe eines Parsers zu einem syntaktisch korrekten Programmquelltext ist im allgemeinen ein abstrakter Syntaxbaum (siehe Definition 2.4.6). Die Programmrepräsentation, die der Parser hier liefert, ist ein um globale Festlegungen zur Speicherung von Formatierungsfestlegungen erweiterter abstrakter Syntaxbaum. Die genaue Definition der Programmrepräsentation folgt in Abschnitt 4.2

Programmierrichtlinien

Als Eingabe für das Prüfungs- und Transformationsmodul ist eine Konfigurationsdatei anzugeben, die die Spezifikation der Programmierrichtlinen enthält. Die Formulierung der Programmierrichtlinien erfolgt eng an der abstrakten Syntax in der Beschreibungssprache, die in Kapitel 5 vorgestellt wird.

Transformation und Prüfung

Das Prüfungs- und Transformationsmodul prüft auf Einhaltung der in der Konfigurationsdatei spezifizierten Programmierrichtlinien und transformiert gegebenenfalls gemäß der Richtlinien. Ob transformiert werden soll, ist zum einen abhängig von der Vorgabe des Benutzers und zum anderen von der Art der Programmierrichtlinie (siehe Abschnitt 4.5).

Warnmeldungen

Eine Art der Ausgabe der Anwendung sind Warnmeldungen, die auf Verletzungen von Programmierrichtlinien hinweisen. Eine Warnmeldung umfaßt Angaben zur Art der Verletzung und zur Position im Quelltext.

Transformierte Programmrepräsentation

Soll ein Quelltext automatisch zu den Programmierrichtlinien konform korrigiert werden, wird die Programmrepräsentation transformiert.

Printer und Transformierter Programmquelltext

Der Printer wandelt die transformierte Programmrepräsentation in einen Programmquelltext um. Im Prinzip handelt es sich hierbei um eine Umkehrung des Parsers und deswegen spricht man auch von einem Unparser (vgl. [MvdB96]). Im Unterschied zu einem Pretty Printer enthält ein Printer keinerlei Vorgaben, wie bestimmte Elemente der Programmrepräsentation zu formatieren sind. Die Formatierungen sind in der Programmrepräsentation kodiert.

4.2 Programmrepräsentation

Der Parser wandelt einen Programmquelltext in eine Programmrepräsentation um. Diese beinhaltet neben dem abstrakten Syntaxbaum einen globalen Teil, in dem Festlegungen zu Formatierungen gespeichert sind. Im folgenden wird die Programmrepräsentation durch eine Grammatik in EBNF-Notation definiert (siehe Tabelle 4.2). Eine Implementierung (vgl. Kaptiel 6) kann die Darstellung des Baumes beispielsweise mit XML 1.0 nach [Wor00] realisieren.

```
ProgramRepresentation
  = Globals AST ;
```

Abbildung 4.2: EBNF-Notation der Programmrepräsentation

4.2.1 Globale Festlegungen

Programmiersprachen erlauben eine Menge von Zeichen(-folgen) (Whitespace) zum Trennen von lexikalischen Einheiten. Des weiteren bieten die meisten höheren Programmiersprachen Möglichkeiten, in Form von sogenannten Kommentaren Anmerkungen in einem Programmquelltext einzufügen.

Die globalen Festlegungen dienen zur Definition der Whitespace-Zeichen und Kommentarstile der zugrundeliegenden Programmiersprache. Die Definition für Globals in EBNF-Notation ist wie in Abbildung 4.3. Dabei wird zwischen Whitespace-Zeichen zur horizontalen und zur vertikalen Trennung und nach Block- und Zeilenendekommentaren unterschieden. Eine CharSeq besteht aus einer nichtleeren Sequenz von beliebigen Zeichen. Die Reihenfolge der Definition der Elemente der Klassen HWhitespace und VWhitespace ist wichtig für die spätere Interpretation und Ersetzung von Leerraum, da die Beschreibungssprache nur horizontale und vertikale Komposition unterscheidet. Die textuell erste auftretene Definition ist jeweils (bei HWhitespace und VWhitespace) die bei der Umsetzung von horizontaler und vertikaler Komposition bei Formatierungsregeln relevante Definition.

4.2.1.1 Horizontale Trennung

Das gebräuchlichste Zeichen zur horizontalen Trennung ist das Leerzeichen. Daneben erlauben viele Programmiersprachen auch Tabulatoren zur horizontalen Trennung, die jedoch im Sinne von Programmierrichtlinien aufgrund von unterschiedlicher Darstellung in unterschiedlichen Editoren häufig verboten werden. Vorsicht ist bei Programmiersprachen geboten, bei denen

```
Globals
   = {Whitespace} {Comments} ;

Whitespace
   = {HWhitespace} {VWhitespace} ;

HWhitespace
   = 'hWhitespace' 'value' '=' '"' CharSeq '"';

VWhitespace
   = 'vWhitespace' 'value' '=' '"' CharSeq '"';

Comments
   = {BlockComment} {EolComment} ;

BlockComment
   = 'blockComment' 'begin' '=' '"' CharSeq '"' 'end' '=' '"' CharSeq '"';

EolComment
   = 'eolComment' 'begin' '=' '"' CharSeq '"' end' '=' '"' CharSeq '"';
```

Abbildung 4.3: EBNF-Notation von Globals und Whitespace

horizontale Trennzeichen semantiktragend sein können (z.B. Offside-Rule in Haskell). Bei der Formulierung von Formatierungsrichtlinien muß also Sorge getragen werden, daß die Semantik des Programmes nicht verändert wird.

4.2.1.2 Vertikale Trennung

Die vertikale Trennung wird im wesentlichen über Zeilenumbrüche realisiert. Für Zeilenumbrüche existieren unterschiedliche Kodierungen, was auch in der verschiedenartigen Kodierung der Zeilenumbrüche in unterschiedlichen Betriebssystemklassen begründet ist. Tabelle 4.1 gibt dazu einen Überblick.

Betriebssystemklasse	**Kodierung**	**Beschreibung**
meiste Unix-Derivate	\n	Linefeed
MS-Dos, Windows	\r\n	Carriage Return + Linefeed
Mac-OS	\r	Carriage Return

Tabelle 4.1: Zeilenumbrüche bei verschiedenen Betriebsystemklassen

Neben den Zeilenumbrüchen kann es weitere Zeichen zur vertikalen Trennung geben. So erlaubt beispielsweise Java auch den Seitenumbruch (Formfeed) als weiteres vertikales Trennzeichen.

4.2.1.3 Blockkommentare

Neben einer evtl. mehrere Zeichen umfassenden Zeichenfolge, die den Kommentar einleitet, hat ein Blockkommentar auch eine Zeichenfolge, die den Kommentar abschließt. Mit Ausnahme der Kommentarendefolge sind dazwischen in der Regel alle zulässigen Eingabesymbole der Programmiersprache in beliebiger Reihenfolge erlaubt, die die Programmiersprache zulässt. Die Klasse BlockComment bietet die Möglichkeit eine beliebige Anzahl von Blockkommentarstilen zu definieren, indem die Kommentaranfangs- sowie die Kommentarendekennung spezifiziert werden.

4.2.1.4 Zeilenendekommentare

Zeilenendekommentare können als eine spezielle Form der Blockkommentare angesehen werden, die ebenfalls mit einer evtl. mehrere Zeichen umfassenden Kommentaranfangskennung beginnen und mit einem Zeilenumbruch enden. Die Trennung der Definitionen von Zeilenende- und Blockkommentaren liegt in der unterschiedlichen Interpretation dieser Klassen im Sinne von Formatierungen begründet, worauf allerdings erst später eingegangen werden soll (siehe Kapitel 4.5.1.1).

4.2.1.5 Klassifizierung für Java

Die Tabelle 4.2 nimmt eine Klassifizierung für Whitespace der Programmiersprache Java vor, während Tabelle 4.3 eine Klassifizierung der Java-Kommentarstile zeigt.

4.2.2 Abstrakter Syntaxbaum (AST)

Der abstrakte Syntaxbaum repräsentiert den syntaktisch analysierten Programmquelltext. Die Definition, der im Rahmen dieser Arbeit entstandenen abstrakten Syntax der Programmiersprache Java, ist Anhang B zu entnehmen. Der im folgenden beschriebene abstrakte Syntaxbaum erfüllt Definition 2.4.6.

Eine Restriktion für die abstrakte Syntax ist das Verbot von Iterationen, da die später vorgestellte Beschreibungssprache keine Beschreibungsmöglichkeiten für Iterationen vorsieht. Diese Restriktion stellt insofern keine Einschränkung dar, da eine Iteration in der abstrakten Syntax in eine Produktion mit zwei Alternativen ohne Iteration umsetzbar ist (Äquivalenz von EBNF und BNF).

Während die Blätter des abstrakten Syntaxbaumes die Terminale der abstrakten Syntax repräsentieren, entsprechen die inneren Knoten den Nichtterminalen. Nichtterminalen sowie auch Terminalen werden im AST Attri-

Name	**Zeichen**	**Beschreibung**
HWhitespace		
Leerzeichen	' '	Leerzeichen sind häufig die einzig erlaubten horizontale Trennungszeichen im Sinne von Programmierrichtlinien
Tabulator	\t	Tabulatoren werden teilweise zum Einrücken verwendet
VWhitespace		
Zeilenumbruch	\r \n \r\n	Java akzeptiert alle drei voranstehenden Versionen des Zeilenumbruchs
Seitenumbruch	\f	Der Seitenumbruch (Formfeed) findet selten in Java als Whitespace Zeichen Anwendung

Tabelle 4.2: Klassifizierung von Whitespace und Kommentaren für Java

bute angeheftet. Die Attribute sind Paare (n, v), wobei n der Attributname und v der Attributwert ist (siehe Definition 2.1.1). Dabei haben Terminale Attribute für

- die konkrete Belegung (Attributname „value"),
- die Zeilennummer des ersten Zeichens des Terminals im Programmquelltext (Attribut „line"),
- die Spaltennummer des ersten Zeichens des Terminals im Programmquelltext (Attribut „column"),
- voranstehenden Whitespace und Kommentare (Attribut „whitespace").

Nichtterminale haben lediglich ein Attribut „whitespace" für den voranstehenden Whitespace sowie die Kommentare. Allgemein besteht ein Knoten im AST aus dem Knotennamen und Attributen.

4.2.2.1 Whitespace und Kommentare im AST

Entgegen gängiger Praxis im Compilerbau, bei der Whitespace und Kommentare bereits während der lexikalischen Analyse verworfen werden, ist beim Überprüfen von Formatierungen eine Repräsentation des Whitespaces im AST unverzichtbar. Whitespace und Kommentare können an beliebiger

Name	Beginn	Ende	Beschreibung
BlockComment			
Normaler Kommentar	/*	*/	Evtl. mehrzeiliger Kommentar für Erläuterungen
JavaDoc Kommentar	/**	*/	Evtl. mehrzeiliger Kommentar zur Erstellung von Dokumentation mit JavaDoc
EolComment			
Zeilenendekommentar	//	\r\n \r \n	Wird mit einem Zeilenumbruch abgeschlossen

Tabelle 4.3: Klassifizierung von Kommentaren für Java

Stelle im Programmquelltext in beliebigem Umfang auftreten. Daher wird in der vorliegenden Arbeit jedem Knoten des abstrakten Syntaxbaumes das Attribut „whitespace" angeheftet, in dem vor dem syntaktischen Element liegender Whitespace und Kommentare abgelegt werden. Bei der Zuordnung des Whitespaces inklusive der Kommentare zu den Knoten wird wie folgt vorgegangen:

1. Der Whitespace und die Kommentare werden dem nachfolgenden Terminal zugeordnet.
2. Ist ein Terminal der linkeste Nachfolger eines Nichtterminals, werden Whitespace und Kommentare dem umschließenden Nichtterminal zugeordnet.
3. Ist ein Nichtterminal der linkeste Nachfolger eines weiteren Nichtterminals, werden Whitespace und Kommentare dem umschließenden Nichtterminal zugeordnet.

Die getroffene Zuordnung ist sinnvoll, da so die Prüfung von Teilbäumen beim Durchlaufen des abstrakten Syntaxbaumes erleichtert wird. Bei einer Zuordnung des Whitespaces und der Kommentare nur zu den Terminalen müßte z.B. bei der Prüfung der Einrückung eines Statements, bis zum ersten Terminal im Teilbaum des Statements im AST abgestiegen werden, um die Prüfung der Formatierung vornehmen zu können.

4.2.2.2 Beispiel für ein AST-Fragment

Zur Veranschaulichung soll zum Programmausschnitt aus Abbildung 4.4 der korrespondierende abstrakte Syntaxbaum angegegeben werden. Zum einen wird der zugehörige AST als Baum gezeigt (Abbildung 4.5), zum anderen wird eine XML-Darstellung des ASTs in Abbildung 4.6 angegeben. Mit Ausnahme des Attributs „value" bei den Terminalknoten wird in der Baumdarstellung auf die Angabe sämtlicher Attribute verzichtet.

```
if ( a == 1 ) {
    ...
}
```

Abbildung 4.4: Programmfragment

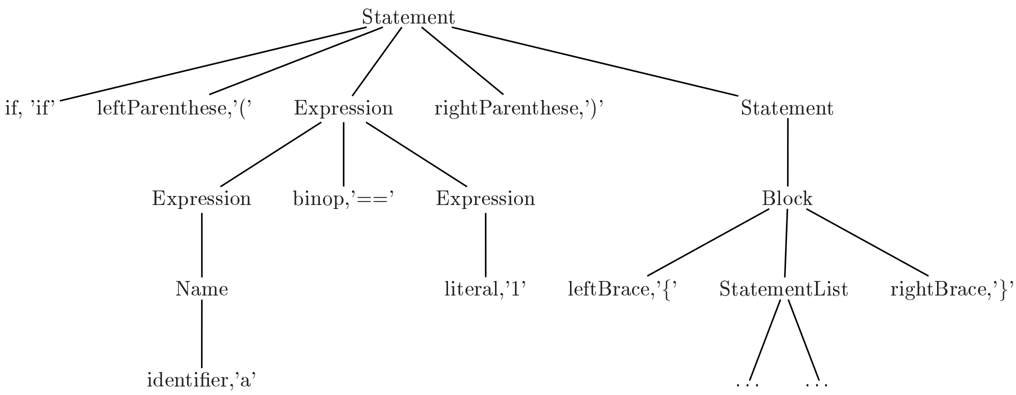

Abbildung 4.5: Baumdarstellung

Die XML-Darstellung (Abbildung 4.6) beinhaltet sämtliche Attribute unter der Annahme, daß das Terminal 'if' in Zeile 15 und Spalte 8 steht. In dieser Darstellung wird auch der Transfer des Inhaltes des Attributes „whitespace" von einem Terminal auf das äußerste direkt umschließende Nichtterminal deutlich. Der Leerraum vor dem Terminal identifier, 'a' wird nicht diesem Terminal zugeordnet, sondern dem äußeren Nichtterminal Expression, welches nicht linkester Nachfolger eines weiteren Nichtterminals ist. Eine ähnliche Situation findet sich beim Nichtterminal Statement, zu dem das „whitespace"-Attribut des Terminals if, 'if' übertragen wird.

4.3 Baummuster für den AST

Im folgenden soll ein Konzept vorgestellt werden, den abstrakten Syntaxbaum mit sogenannten Baummustern zu überdecken. Dazu werden im Abschnitt 4.3.1 zunächst die Begriffe Baummuster und Überdeckung definiert und im Anschluß in Abschnitt 4.3.2 Beispiele für Überdeckungen gezeigt.

```
<Statement whitespace="        ">
  <if value="if" whitespace="" line="15" column="8">
  <leftParenthese value="(" whitespace=" " line="15" column="11">
  <Expression whitespace=" ">
   <Expression whitespace="">
     <Name whitespace="">
       <identifier value="a" whitespace="" line="15" column="13">
     </Name>
   </Expression>
   <binop value="==" whitespace=" " line="15" column="15">
   <Expression whitespace=" ">
     <literal value="1" whitespace="" line="15" column="18">
   </Expression>
  </Expression>
  <rightParenthese value=")" whitespace=" " line="15" column="20">
  <Statement whitespace=" ">
   <Block whitespace="">
     <leftBrace value="{" whitespace="" line="15" column="22">
     <StatementList whitespace="\n      ">
       ...
     </StatementList>
     <rightBrace value="}" whitespace="" line="17" column="8">
   </Block>
  </Statement>
</Statement>
```

Abbildung 4.6: XML-Darstellung des AST mit allen Attributen

Dabei wird auf Mehrdeutigkeiten in Form von Überschneidungen von Baummustern hingewiesen, die im Abschnitt 4.3.3 aufgelöst werden sollen.

4.3.1 Definitionen

Zunächst soll der Begriff des Baummusters eingeführt werden. Ein Baummuster ist ein geordneter Baum mit speziellen Anforderungen für die Blätter. Mit Hilfe von Baummustern sollen später Teilbäume des abstrakten Syntaxbaumes überdeckt werden. Die Baummuster beinhalten dann (siehe Abschnitt 4.4) die Programmierrichtlinien, deren Einhaltung (im AST) geprüft werden soll.

Definition 4.3.1 (Baummuster)
Ein *Baummuster* ist ein geordneter Baum, wobei für zwei paarweise verschiedene Blätter k, k' gilt: $Name(k) \neq Name(k')$ oder $Name(k) = Name(k') =$

ϵ.

Im folgenden wird auf die Definition von Überdeckungen eines Teilbaumes oder Baummusters mit einem Baummuster hingearbeitet. Dazu werden nacheinander Attributüberdeckung, Knotensemiüberdeckung und Knotenüberdeckung definiert, mit deren Hilfe dann der Begriff Überdeckung eingeführt werden kann.

Definition 4.3.2 (Attributüberdeckung)
Seien $a = (n, v)$ und $a' = (n', v')$ Attribute.
Dann heißt a' *Attributüberdeckung* von a, falls $n = n'$ und

- $v = v'$ oder
- $f'(p_1, \ldots p_l, v) = true$, falls $v' = (f', p_1, \ldots p_l)$.

Die Schreibweise $a' \rhd a$ bedeutet a' ist Attributüberdeckung von a.

Eine Attributüberdeckung liegt dann vor, wenn bei gleichem Attributnamen die Attributwerte entweder auch gleich sind (falls die Attributwerte beide Zeichenketten oder beide Tupel sind) oder die Funktionsanwendung auf den Attributwert des zu überdeckenden Attributs *true* ergibt (falls Attributwert des überdeckenden Attributs Tupel ist). Ein Knoten überdeckt einen anderen Knoten im Sinne einer Knotensemiüberdeckung, wenn alle Attribute des ersten Knotens Attributüberdeckung eines Attributs des zweiten Knoten sind.

Definition 4.3.3 (Knotensemiüberdeckung)
Seien $k = (t, A)$ und $k' = (t', A')$ Knoten.
Dann heißt k' *Knotensemiüberdeckung* von k, falls $\forall a' \in A' \exists a \in A$: $a' \rhd a$.

Im Gegensatz zur Knotensemiüberdeckung muß für eine Knotenüberdeckung zusätzlich auch der Knotenname gleich sein.

Definition 4.3.4 (Knotenüberdeckung)
Seien $k = (t, A)$ und $k' = (t', A')$ Knoten.
Dann heißt k' *Knotenüberdeckung* von k, falls k' Knotensemiüberdeckung von k und $Name(k') = Name(k)$.

Schließlich ist ein Baummuster eine Überdeckung eines Teilbaums oder eines anderen Baummusters, wenn die Struktur der beiden geordneten Bäume gleich ist und für alle korrespondierenden inneren Knoten eine Knotenüberdeckung und alle korrespondierenden Blätter eine Knotensemiüberdeckung vorliegt.

Definition 4.3.5 (Überdeckungen)
Sei $b = (r, T)$ mit $T = (b_1, \ldots, b_n), n \geq 0$ Teilbaum oder Baummuster, und

sei $b' = (r', T')$ mit $T' = (b'_1, \ldots, b'_n)$ Baummuster.
Dann heißt b' *Überdeckung* von b

- falls $n = 0 \Rightarrow r'$ ist Knotensemiüberdeckung von r.
- falls $n > 0 \Rightarrow r'$ ist Knotenüberdeckung von r und $\forall i$ mit $1 \leq i \leq n$: b'_i ist Überdeckung von b_i.

4.3.2 Beispiele für Überdeckungen

Im folgenden werden Beispiele für Baummuster und die Überdeckung von Teilbäumen in einem AST (Ausschnitt) gezeigt. In Abbildung 4.7 ist ein Baummuster zu sehen, in dem die Knoten mit dem leeren Typ ϵ ein „value" Attribut mit Wert 'if', '(' oder ')' enthalten. Diese Knoten dienen zur Überdeckung von Terminalen. Die Blätter lit und id haben keine Attribute und dienen zur Überdeckung von Terminalen die einen konkreten Wert aus einem Wertebereich (mit mehr als einem Wert) annehmen können. Alle inneren Knoten haben einen nichtleeren Typ und keine Attribute. Allgemein steht die Notation $n : n'$ für eine Knotensemiüberdeckung eines Knotens k mit $Name(k) = n$ durch einen Knoten k' mit $Name(k') = n'$.

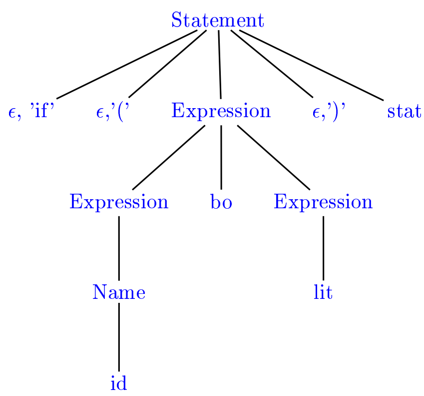

Abbildung 4.7: Baummuster

In Abbildung 4.8 umfaßt die blaue Linie den Teilbaum in dem AST Ausschnitt aus Kapitel 4.2.2, den das Baummuster aus Abbildung 4.7 überdeckt. Ebenfalls blau markiert sind die Elemente des Baummusters. Eine weitere Überdeckung eines anderen Teilbaumes mit einem anderen Baummuster in demselben AST Ausschnitt ist in Abbildung 4.9 zu sehen. Wieder umfaßt die blaue Linie den überdeckten Teilbaum, und die blau eingefärbten Elemente bestimmen die Elemente des Baummusters.

Neben der Überdeckung aus Abbildung 4.8 sind weitere Überdeckungen mit Baummustern für diesen AST Ausschnitt denkbar. Ein weiteres Beispiel

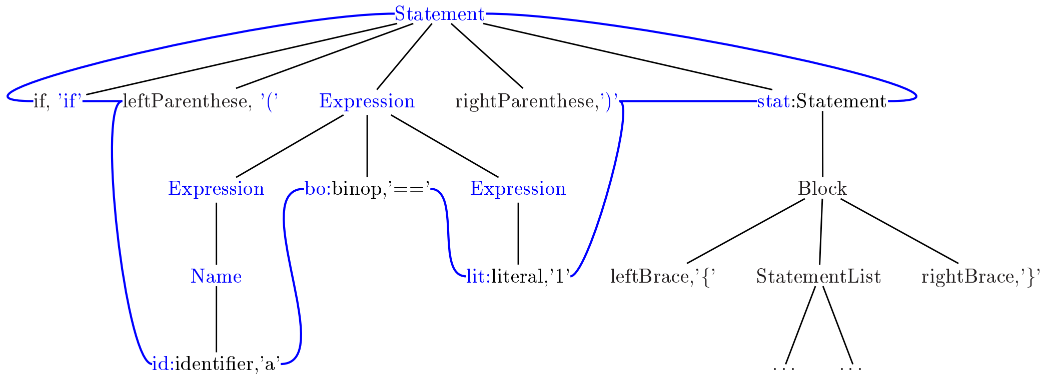

Abbildung 4.8: Überdeckung mit einem Baummuster

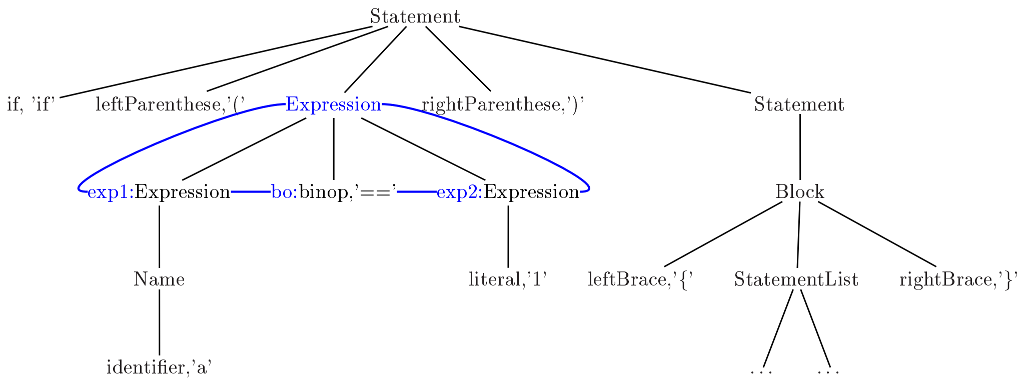

Abbildung 4.9: Eine weitere mögliche Überdeckung

zeigt Abbildung 4.9. Spätestens bei der Betrachtung dieser dritten möglichen Überdeckung fällt auf, daß sich die gezeigten überdeckten Teilbäume überschneiden. Bei der Prüfung von Teilbäumen auf die Einhaltung von Programmierrichtlinien (siehe Abschnitt 4.4), ist eine eindeutige Zuordnung eines Knotens zu einer Regel und somit zu einer Überdeckung sinnvoll, um eine Eindeutigkeit in der Zuordnung von Programmierrichtlinien zu einem Teilbaum zu erzwingen[1].

[1]Hier wird vorgegriffen, da die Formulierung von Programmierrichtlinien mittels Baummustern erst nachfolgend diskutiert wird. Durch Abschnitt 4.5 wird die Notwendigkeit der eindeutigen Zuordnung deutlich, da sonst durch die Anwendung mehrerer Programmierrichtlinien einer Klasse auf einen Teilbaum Situationen entstehen können, in denen die Erfüllung aller Programmierrichtlinien für den Teilbaum nicht möglich ist.

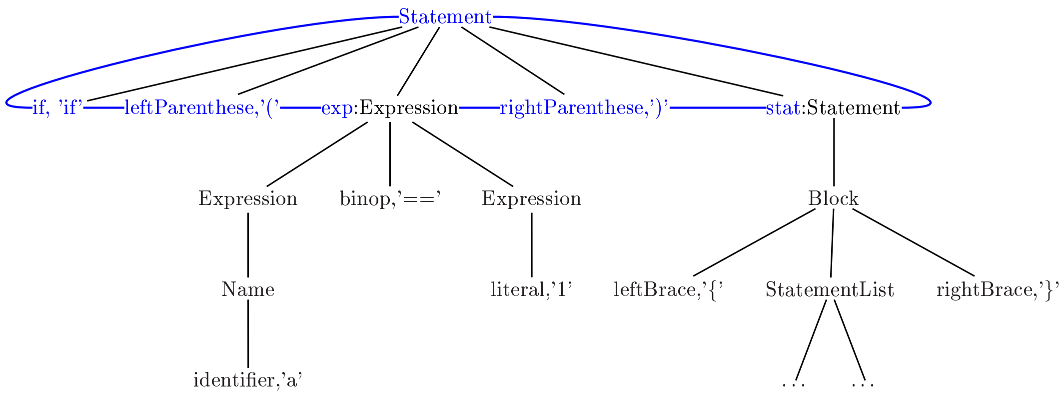

Abbildung 4.10: Eine dritte mögliche Überdeckung

4.3.3 Priorisierung von Baummustern

Die überdeckten Teilbäume aus den Abbildungen 4.8 – 4.10 überschneiden sich in unterschiedlicher Art und Weise. Weitere, als die durch die Beispiele gezeigten und im folgenden diskutierten, Unterteilungen sind zur eindeutigen Auswahl eines Baummusters nicht notwendig.

1. Die überdeckten Teilbäume aus Abbildung 4.8 und Abbildung 4.10 haben dieselbe Wurzel.

2. Die Wurzel des überdeckten Teilbaumes aus Abbildung 4.9 ist innerer Knoten des überdeckten Teilbaumes aus Abbildung 4.8.

3. Die Wurzel des überdeckten Teilbaumes aus Abbildung 4.9 ist Blatt des überdeckten Teilbaumes aus Abbildung 4.10.

Um die Mehrdeutigkeit aus 1. aufzulösen, wird die Reihenfolge bei der Prüfung, ob ein Baummuster eine Überdeckung eines Teilbaumes ist, als relevant angesehen. Das erste Baummuster mit Wurzel r', das einen der möglichen Teilbäume mit Wurzel r überdeckt (wobei insbesondere auch r' Knotenüberdeckung von r ist) wird gewählt. Betrachtet man nochmals die Baummuster aus den Abbildungen 4.8 (Baummuster A) und 4.10 (Baummuster B), stellt man fest, daß Baummuster A einen spezielleren Fall abdeckt als Baummuster B. Anders formuliert bedeutet dies, daß, falls das Baummuster B in der Reihenfolge vor Baummuster A steht, es nie zu einer Überdeckung durch Baummuster A kommen wird, da zuvor bereits immer Baummuster B paßt. Also sollten speziellere Baummuster in der Reihenfolge weiter vorne stehen als allgemeinere, da es sonst nie zu einer Überdeckung durch die spezielleren Muster kommt.

Um die Mehrdeutigkeit nach 2. aufzulösen, wird die Festlegung getroffen, daß der abstrakte Syntaxbaum gemäß einer Top-Down-Analyse durchlaufen wird. Als Konsequenz ergibt sich für obige Beispiele, daß in der Mehrdeutigkeit nach 2. das Baummuster aus Abbildung 4.8 Priorität vor dem Baummuster aus Abbildung 4.9 besitzt. Für die Begründung dieser Festlegung betrachte man nochmals die Abbildungen 4.8 (Baummuster A) und 4.9 (Baummuster B). Durch Baummuster B werden alle Teilbäume mit Wurzel Expression und der Form ,,Expression binop Expression" überdeckt, während ein solcher Teilbaum nur dann ein Teil des von Baummuster A überdeckten Teilbaumes ist, wenn er in einem bestimmten Kontext auftritt. Durch die Festlegung der Top-Down-Analyse werden also speziellere Baummuster allgemeineren Baummustern vorgezogen.

Die Auflösung einer Überschneidung nach 3 findet wie folgt statt: Falls die Wurzel eines überdeckten Teilbaumes A Blatt eines weiteren überdeckten Teilbaumes B ist, dann ist die Überdeckung mit beiden Baummustern zulässig. Ein durch einen Knoten überdecktes Blatt eines Teilbaumes kann also gleichzeitig Wurzel eines anderen überdeckten Teilbaumes sein.

4.4 Prüfung auf Einhaltung von Programmierrichtlinien

4.4.1 Programmierrichtlinien als Baummuster

Die Idee, die der Prüfung auf Einhaltung von Programmierrichtlinien zugrunde liegt, fußt auf einer Überdeckung des abstrakten Syntaxbaumes mit Baummustern. Eine Regel besteht aus zwei Baummustern nach Definition 4.4.1. Dabei überdeckt das erste Baummuster die Teilbäume, für die die Programmierrichtlinie geprüft werden soll. Das zweite Baummuster beinhaltet dann die eigentlichen Programmierrichtlinien, z.B. in Form von weiteren gesetzten Attributen im Vergleich zum ersten Baummuster.

Definition 4.4.1 (Regel)
Seien b_1 und b_2 Baummuster.
Sei r_1 Wurzel von b_1 und r_2 Wurzel von b_2.
Das Tupel (b_1, b_2) heißt *Regel*, falls

- r_1 Knotenüberdeckung von r_2 ist oder
- r_1 Knotensemiüberdeckung von r_2 und $|r_1|$=$|r_2| = 1$ ist.

Eine Regel besteht nach Definition 4.4.1 aus zwei Baummustern. Die Prüfung auf Einhaltung der im Muster b_2 einer Regel $p = (b_1, b_2)$ kodierten Programmierrichtlinien läuft ab, indem, falls ein Teilbaum t von b_1 überdeckt wird, geprüft wird, ob auch b_2 t überdeckt:

(i) b_2 überdeckt ebenfalls $t \Rightarrow p$ wurde in t eingehalten

(ii) b_2 überdeckt nicht $t \Rightarrow p$ wurde in t verletzt

Die syntaktische und semantische Korrektheit eines gemäß einer Regel transformierten Programmes wird durch obige Definition nicht gewährleistet und liegt in der Obhut desjenigen, der die Regeln formuliert (Ein Beispiel ist das Weglassen von Elementen im zweiten Baummuster einer Regel). Ein Baummuster b_2 einer Regel $p = (b_1, b_2)$ kann mehrere Programmierrichtlinien beinhalten (z.B. eine Formatierung und eine Namenskonvention). In den folgenden Beispielen wird für eine Regel $p = (b_1, b_2)$ eine Interpretation der Regel in Abhängigkeit von der Beziehung zwischen den Baummustern b_1 und b_2 angegeben.

- Wenn $b_1 = b_2$, wird durch b_2 keine zusätzliche Eigenschaft gefordert, und p ist immer erfüllt (Identität).

- Wenn b_1 b_2 überdeckt ($b_1 \neq b_2$), werden durch b_2 zusätzliche Eigenschaften in den durch b_1 überdeckten Teilbäumen geprüft, da b_2 mehr Attribute als b_1 enthält. Diese Situation findet man z.B. bei der Prüfung von Formatierungsregeln und Namenskonventionen.

- Wenn b_1 b_2 nicht überdeckt, dann kann ein von b_1 überdeckter Teilbaum nicht von b_2 überdeckt werden. Die durch Baummuster b_2 formulierte Programmierrichtlinie wird also in allen Teilbäumen verletzt, die von b_1 überdeckt werden. Diese Situation findet man bei Programmierrichtlinien der Klassen Verbot/Gebot von Redundanzen und Reihenfolge von Deklarationen.

4.4.2 Warnmeldungen und Transformationen

Nachdem die Prüfung auf Einhaltung von Programmierrichtlinien ergeben hat, ob die Programmierrichtlinien eingehalten oder verletzt wurden, soll in diesem Abschnitt geklärt werden, wie auf eine mögliche Verletzung einer Programmierrichlinie reagiert werden kann. Im wesentlichen gibt es dabei zwei Möglichkeiten:

- Ausgabe einer Warnmeldung mit Art und Position der Verletzung

- Korrektur der Verletzung, d.h. Transformation des ASTs

Unproblematischer ist die Ausgabe einer einfachen Warnmeldung, mit Art und Position (im Programmquelltext) der Verletzung. Die Position der Verletzung kann über das Terminal ermittelt werden, das der Verletzung am nächsten steht, da die Terminale ihre Position im Quelltext als Attribute „line" und „column" beinhalten.

Bei einer Transformation des ASTs kann es aufgrund des lokalen Charakters der Korrektur zu unerwünschten Nebeneffekten kommen. Beispielsweise hätte die Korrektur eines Variablennamens bei einer Deklaration (bei Verletzung einer Namenskonvention) zur Folge, daß an allen Stellen, an denen die Variable verwendet wird, ebenso eine Umbenennung stattfinden müsste. Innerhalb derselben Quelldatei wäre dies mit entsprechend formulierten Regeln noch realisierbar. Bei einem Zugriff auf diese Variable aus einer anderen Datei (z.B. auf Klassenvariablen in Java) wird das Problem in diesem Rahmen nicht mehr beherrschbar. Die Korrektur von Formatierungen ist unter gewissen Voraussetzungen jedoch möglich (siehe Abschnitt 4.5.1).

4.5 Anwendung auf die Programmierrichtlinienklassen

Bei der Betrachtung der unterschiedlichen Programmierrichtlinienklassen soll zunächst davon ausgegangen werden, daß eine Regel p nur Programmierrichtlinien einer Klasse enthält. Im Anschluß an die Diskussion der einzelnen Programmierrichtlinienklassen wird dann in Abschnitt 4.5.5 auf Mischformen eingegangen, da die Defintion 4.4.1 es zuläßt, in einer Regel gleichzeitig mehrere Programmierrichtlinien unterschiedlicher Klassen zu definieren. So könnten mit einer Regel nach Definition 4.4.1 beispielsweise gleichzeitig Formatierungen und Namenskonventionen (auch für mehrere Elemente) geprüft werden.

4.5.1 Formatierungsregeln

Bei einer Regel $p = (b_1, b_2)$, die die Formatierung eines Konstruktes der Programmiersprache steuert, gilt für die Baummuster b_1 und b_2, daß b_1 b_2 überdeckt. Das Baummuster b_2 enthält im Vergleich zu b_1 nur zusätzliche Attribute „whitespace" und damit ist b_1 Überdeckung von b_2.

Allerdings sind im AST im Attribut „whitespace" zur späteren Rekonstruktion die vor dem Element stehenden Kommentare mitkodiert. Es reicht also nicht aus, zu prüfen, ob der, in den „whitespace"-Attributen des Baummusters b_2 kodierte Leerraum, den Werten der „whitespace"-Attribute der jeweiligen Knoten des ASTs entspricht. Vielmehr muß sich der Leerraum der „whitespace"-Attribute des Baummusters b_2 mit dem formatierungsrelevanten Leerraum der „whitespace"-Attribute der zu überdeckenden Knoten des ASTs decken. Daher hat ein „whitespace"-Attribut zur Prüfung einer Formatierung in einem Baummuster b_2 eine Gestalt wie in Abbildung 4.11. Der Attributwert ist die boolsche Funktion $wsMatch$ mit Parameter p_1, der, den vor dem Element des Baummusters geforderten, Leerraum beinhaltet. Ein Funktionsaufruf $wsMatch(p_1, v)$ hat den Wert $true$ als Rückgabe, falls der aus v extrahierte formatierungsrelevante Leerraum mit p_1 übereinstimmt.

In allen anderen Fällen liefert $wsMatch(p_1, v)$ false.

$(whitespace, (wsMatch, p_1))$

Abbildung 4.11: „whitespace"-Attribut in einem Baummuster

Zur Diskussion des Problems, welcher Leerraum für die Formatierung relevant ist, wird zunächst auf verschiedene Situationen eingegangen, in denen Kommentare auftreten können, und wie dieses Auftreten dann hinsichtlich des formatierungsrelevanten Leerraumes interpretiert werden soll.

4.5.1.1 Interpretation von Kommentaren

Kommentare werden in dieser Arbeit als vorangestellte oder nachgestellte Erläuterungen interpretiert. Nachfolgend wird über die Diskussion von Programmbeispielen, formuliert in der Programmiersprache Java, ein Konzept erarbeitet, den verschiedenen Kommentarstilen Funktionen zuzuordnen. In der Praxis werden diese Rollen für Kommentare nicht durchgängig durchgehalten. Die Annahmen, die jedoch über diese Rollenzuteilung für Kommentarstile getroffen werden, fordern ein bewußtes Kommentieren von Programmen. Dieses vereinbart sich mit der Zielsetzung von Programmierrichtlinien. Dem Betrachter von Programmquellcode wäre somit allein über die Art des Kommentares sofort die Funktion des Kommentares bewußt. Insofern stellt die Forderung des bewußten Kommentierens also keine echte Einschränkung der Anwendbarkeit der in dieser Arbeit vorgestellten Konzepte dar.

Kommentare sind mit den Charakteristiken, die in Abschnitt 4.2.1 beschrieben sind, nach Block- und Zeilenendekommentaren aufteilbar. Für die korrekte Leerraumermittlung ist die Unterscheidung von Block- und Zeilenendekommentaren notwendig, wie im folgenden deutlich wird.

Blockkommentare

Man betrachte die Situation aus Abbildung 4.12 in einem Java-Programm. Dazu sei eine Richtlinie gegeben, die zwischen zwei Methoden genau zwei Leerzeilen fordert. Betrachtet man den Leerraum vor dem Kommentar, wäre diese Richtlinie erfüllt, während mit dem Leerraum nach dem Kommentar oder dem Leerraum vor und nach dem Kommentar die Richtline verletzt würde.

Versteht man Blockkommentare im Sinne von vorangestellten Erläuterungen, liegt die Interpretation nahe, den Blockkommentar semantisch dem nachfolgenden Knoten zuzuordnen. Insofern wäre es sinnvoll, nur den Leerraum vor dem Blockkommentar als relevant anzusehen. Tatsächlich existieren Programmierrichtlinien, die fordern, Blockkommentare dem Kommentierten immer voranzustellen. Mit dieser Interpretation des Blockkommen-

```
public void doSomething() {
    ...
}


/**
Dieser Kommentar beschreibt die nachfolgende Methode.
*/

public void doSomethingElse() {
    ...
}
```

Abbildung 4.12: Beispiel für Whitespace und einen Blockkommentar

tars in Abbildung 4.12 wäre oben angeführte Richtlinie erfüllt. Es soll aber an dieser Stelle nicht ausgeschlossen werden, daß der auf den Blockkommentar folgende Leerraum als relevant bezüglich der Formatierung anzusehen ist. Aus diesen Überlegungen heraus muß es eine Möglichkeit geben, Blockkommentaren eine Rolle als vor- oder nachgestellte Erläuterung zuzuordnen.

Zeilenendekommentare

Zeilenendekommentare finden häufig Verwendung, um direkt nachfolgend eine Erläuterung für das Voranstehende zu geben.

```
public void doSomething() {
    int i; // Zähler
    int j; // noch ein Zähler
    ...
} // Ende der Methode doSomething
```

Abbildung 4.13: Beispiel für Whitespace und Zeilenendekommentare

Abbildung 4.13 zeigt an zwei Positionen eine gängige Verwendung von Zeilenendekommentaren. Der auf die Zeilenendekommentare folgende Leerraum, scheint in dieser Situation im Sinne der Formatierung interessanter, da der Kommentar Voranstehendes erläutert. Problematisch ist, daß das Zeilenende zur Definition des Kommentares gehört. Das wird deutlich, wenn man die Richtlinie betrachtet, daß pro Zeile immer nur eine Anweisung stehen darf. Anders formuliert heißt das, daß zwischen zwei Anweisungen ein Zeilenumbruch stehen muß. Entfernt man den Kommentar inklusive voranstehendem Leerraum, der der Anweisung „int j;" voransteht, aus dem Attribut „whitespace", das den Whitespace und die Kommentare repräsentiert, dann hat das

Attribut keinen Inhalt mehr. Insbesondere fehlt auch der durch die Richtlinie geforderte Zeilenumbruch. Allerdings fordert die Richtlinie keineswegs eine Formatierung gemäß Abbildung 4.14. Insofern ist der abschließende Zeilenumbruch eines Zeilenendekommentares relevant im Sinne der Formatierung.

```
public void doSomething() {
    int i; // Zähler

    int j; // noch ein Zähler
    ...
} // Ende der Methode doSomething
```

Abbildung 4.14: Falsche Richtlinienumsetzung

Neben der Verwendung als nachgestellte Erläuterung werden Zeilenendekommentare auch als vorangestellte Erläuterungen genutzt. Die Entwicklungsumgebung JBuilder erzeugt beispielsweise Programmquelltext, wie in Abbildung 4.15 dargestellt.

```
public class MyFrame extends JFrame {

    // constructs the frame
    public MyFrame() {
        ...
    }
```

Abbildung 4.15: Generierter Code von JBuilder der Firma Borland

Stellt man wieder Überlegungen bezüglich der Relevanz des betrachteten Leerraums an, gelangt man zu dem Schluß, daß in dieser Situation der Leerraum vor dem Zeilenendekommentar der wichtige Leerraum für die Formatierung ist, während der abschließende Zeilenumbruch nicht für die Formatierung relevant ist.

Wird eine einheitliche Kommentierung gefordert, wird ein Zeilenendekommentar entweder die eine oder die andere Funktion haben. Deutlich sollte jedoch geworden sein, daß eine Rollenzuordnung zu einem Kommentarstil notwendig ist, um den formatierungsrelevanten Whitespace extrahieren zu können.

Aufeinanderfolgende Kommentare

Des weiteren kann es zu Situationen kommen, in denen mehr als ein Kommentar im „whitespace"-Attribut eines Knotens vorkommt. Bei gleichartigen Kommentaren kann die zuvor vorgestellte Interpretation beibehalten werden und der Leerraum vor dem ersten bzw. nach dem letzten Kommentar als relevant angesehen werden. Interessant sind Mischformen wie im Beispiel in der Abbildung 4.16.

```
public void doSomething() {
    int i; // Zähler
    ...
} // Ende der Methode doSomething


/**
Dieser Kommentar beschreibt die nachfolgende Methode.
*/

public void doSomethingElse() {
    ...
}
```

Abbildung 4.16: Beispiel für Whitespace, Zeilenende- und Blockkommentar

Mit den gleichen Argumentationen wie in den vorangegangenen Abschnitten ist der für die Formatierung relevante Leerraum, der Leerraum zwischen Zeilenende- und Blockkommentar. Dabei ist wiederum der abschließende Zeilenumbruch des Zeilenendekommentares wichtig für die Formatierung.

4.5.1.2 Extrahieren von formatierungrelevantem Whitespace

Um den formatierungsrelevanten Whitespace eines AST-Knotens zu extrahieren, muß der Whitespace (aus dem Attribut „whitespace") von den Kommentaren und nicht formatierungsrelevanten Whitespace getrennt werden. Vorausstezung dafür ist, daß den verschiedenen Kommentarstilen eine Rolle als vorangestellte oder nachgestellte Erläuertung zugewiesen wurde. Die Tabelle 4.4 erläutert, wie sich der formatierungsrelevante Whitespace bei den verschiedenen Rollenzuweisungen zusammensetzt.

Es wurde bisher lediglich zwischen Block- und Zeilenendekommentaren unterschieden. Bei genauerer Differenzierung, bei der verschiedene Block- bzw. Zeilenendekommentarstile unterschieden werden können, besteht die Möglichkeit jedem Kommentarstil eine Funktion als vor- oder nachgestellte Erläuterung zuzuordnen. Dann kann analog zum System der Tabelle 4.4 der zu extrahierende Leerraum bei unterschiedlichen Kommentarstilen ermittelt werden.

		Zeilenendekommentar	
		nachgestellte Erläuterung	vorangestellte Erläuterung
Blockkommentar	vorangestellte Erläuterung	Der extrahierte Leerraum besteht aus allem Leerraum zwischen letzten Zeilenendekommentar (einschließlich des abschließenden Zeilenumbruchs) und dem ersten Blockkommentar	Der extrahierte Leerraum besteht aus allem Leerraum vor den Kommentaren
	nachgestellte Erläuterung	Der extrahierte Leerraum besteht aus allem Leerraum nach den Kommentaren (einschließlich des abschließenden Zeilenumbruchs, falls der letzte Kommentar ein Zeilenendekommentar ist)	Der extrahierte Leerraum besteht aus allem Leerraum nach dem letzten Blockkommentar und vor dem ersten Zeilenendekommentar

Tabelle 4.4: Kommentarinterpretationen

4.5.2 Reihenfolge von Deklarationen

Die Überprüfung von Programmierrichtlinien, die sich auf Reihenfolgen von Elementen beziehen, ist nur dann sinnvoll, wenn die resultierende Reihenfolge immer noch syntaktisch korrekt ist. Daher wird die Definition von Programmierrichtlinien dieser Klasse eingeschränkt auf Listen mit einer abstrakten Syntax der Formen aus Abbildung 4.17, da in diesen Fällen auch nach Vertauschung der Elemente das Programm syntaktisch korrekt bleibt.

```
AList                    BList
 = A          oder        = B
 | A AList ;              | B t BList ;
```

Abbildung 4.17: Listen zur Vertauschung von Reihenfolgen

Formuliert man eine Regel $p = (b_1, b_2)$ für eine Programmierrichtlinie der Klasse Reihenfolge von Deklarationen, dann wird b_2 nicht von b_1 überdeckt, da in b_2 die Reihenfolge der Elemente vertauscht ist. Die automatische Korrektur gemäß einer solchen Programmierrichtlinie ist als unsicher anzusehen, da zwar die syntaktische Korrektheit erhalten bleibt, trotzdem das Programm nicht mehr übersetz- oder interpretierbar sein muß, wie folgendes Beispiel zeigt:

```
public void methodA() {
      int i = 0;
      String s = "Zähler="+ i;
      ...
}
```

Werden die beiden Anweisungen vertauscht, da gefordert wird, daß erst Variablen vom Typ String und im Anschluß Variablen vom Typ int deklariert werden sollen, wird eine Transformation vorgenommen, durch die das Programm nicht mehr übersetzbar ist.

4.5.3 Namenskonventionen

Wie bei den Formatierungsregeln ist bei einer Namenskonvention, die mit einer Regel $p = (b_1, b_2)$ geprüft wird, das Baummuster b_1 Überdeckung von Baummuster b_2. Dies ist damit begründet, daß Namen als Attribut von Terminalen kodiert sind.

In der abstrakten Syntax für Java aus Anhang B werden Bezeichner durch das Terminal identifier repräsentiert. Im „value"-Attribut eines identifier-Knotens, das die konkrete Belegung enthält, ist der Name des Bezeichners kodiert.

Zur Prüfung von Namensklassen dient im „value"-Attribut eines Knotens im Baummuster b_2 die boolsche Funktion $regexMatch$, die als Parameter einen regulären Ausdruck enthält. Ein Beispiel für einen solchen Knoten k, der Blatt in einem Baummuster b_2 einer Regel $p = (b_1, b_2)$ ist, ist in Abbildung 4.18 zu sehen.

$$k = (id, (regexMatch, "[a-z].*"))$$

Abbildung 4.18: Beispiel für einen Knoten mit Namenskonvention

Für die Formulierung der regulären Ausdrücke ist die Syntax für reguläre Ausdrücke der Klasse java.util.regex.Pattern des JDK[2] 1.4 maßgebend. Dabei stehen neben den Beschreibungsmöglichkeiten nach Abschnitt 2.2 weitere Beschreibungsmöglichkeiten zur Verfügung. Der Punkt im obigen Beispiel repräsentiert alle Eingabezeichen mit Ausnahme von Zeilenumbrüchen. Im

[2] Java Development Kit

Beispiel aus Abbildung 4.18 werden alle Knoten überdeckt in dessen „value"-Attribut zunächst ein Kleinbuchstabe und darauffolgend beliebige Zeichen stehen.

Als Reaktion auf eine Verletzung ist bei Namenskonventionen lediglich die Ausgabe von Warnmeldungen sinnvoll, da eine lokale Korrektur des Namens gemäß der Konvention weite Auswirkungen bis hin zu anderen Programmodulen haben kann.

4.5.4 Verbot / Gebot von Redundanzen

Bei der Formulierung einer Programmierrichtlinie in einer Regel $p = (b_1, b_2)$, die Verbot oder Gebot von Redundanzen fordert, ist wie bei der Klasse Reihenfolge von Deklarationen b_1 keine Überdeckung von b_2. Das liegt darin begründet, daß für die Formulierung von Programmierrichtlinien die redundante Elemente fordern bzw. verbieten, optionale syntaktische Elemente in b_2 hinzugefügt oder weggelassen werden (gegenüber b_1). Auf der Ebene der Baummuster ist nicht sicherstellbar, daß das Hinzufügen bzw. Weglassen von Elementen die Semantik des Programmes erhält. Insofern ist es auch in bezug auf diese Klasse von Programmierrichtlinien sinnvoll, auf erkannte Verletzungen von Programmierrichtlinien mit Warnmeldungen zu reagieren, und auf eine automatische Korrektur zu verzichten.

4.5.5 Mischformen

In einer Regel $p = (b_1, b_2)$ können Programmierrichtlinien verschiedener Klassen gleichzeitig formuliert werden. Als Beispiel können in einem Baummuster b_2 Forderungen zu einer Formatierung und mehreren Namenskonventionen geprüft werden, da sich die Forderungen in verschiedenen Attributen der Knoten des Baummusters b_2 ausdrücken lassen.

Weitere Mischformen sind denkbar. Die Einarbeitung von Konzepten zur Formulierung von Programmierrichtlinien der Klassen Verbot/Gebot von Redundanzen und Reihenfolge von Deklarationen werden jedoch erst im Ausblick in Abschnitt 7.3 ausführlicher diskutiert.

Kapitel 5

Beschreibungssprache

In diesem Kapitel wird ein Formalismus vorgestellt, der es ermöglicht, Programmierrichtlinien so zu spezifizieren, daß sie anschließend in Baummuster nach Abschnitt 4.3 umgesetzt werden können. Um diese Umsetzung realisieren zu können, ist die Formulierung der Programmierrichtlinien eng mit der abstrakten Syntax der zugrundeliegenden Programmiersprache verknüpft.

Zunächst wird in Abschnitt 5.1 auf existierende Sprachen eingegangen, die programmiersprachenabhängige Formulierung von Programmierrichtlinien (siehe Abschnitt 5.1.1) bzw. für Teilbereiche der Formulierung von Programmierrichtlinien programmiersprachenunabhängige Lösungen (siehe Abschnitte 5.1.2 und 5.1.3) anbieten. Im Anschluß wird in Abschnitt 5.2 auf Grundlage der zuvor vorgestellten Sprache PPML (siehe 5.1.3) die Beschreibungssprache CCML[1] definiert, mit der es möglich ist, Programmierrichtlinien in enger Verknüpfung mit der abstrakten Syntax der zugrundeliegenden Programmiersprache zu spezifizieren.

5.1 Existierende Sprachen

5.1.1 Formalismus von JIndent

JIndent stellt einen Formalismus bereit, um in der Programmiersprache Java geschriebene Programme gemäß Programmierrichtlinien umzuformatieren. Dieser Formalismus besteht aus ca. 160 Parametern, die den Stil der Formatierungen steuern. Diese Parameter werden in einem sogenannten Property-File benutzerdefiniert gesetzt. Die Abbildungen 5.1 und 5.2 zeigen beispielhaft[2] die Nutzung des Formalisums von JIndent, mit dem das Aussehen von switch-Anweisungen gesteuert wird. Dabei dienen die Parameter indentSize (für generelle Einrückung) und indentCaseFromSwitch (Einrückung des „case" gegenüber dem „switch-") zur Festlegung der Einrückung bei einer

[1] Code Conventions Meta Language
[2] aus [TA02]

switch-case-Anweisung.

```
switch(a) {
    case 1:
        System.out.println("one");
        break;
    case 2:
        System.out.println("two");
        break;
}
```

Abbildung 5.1: Einrückung von „case"

Beide Parameter (indentSize und indentCaseFromSwitch) haben im Beispiel aus Abbildung 5.1 den Wert 4. Damit wird eine Einrückung von vier Zeichen vor dem Schlüsselwort „case" gegenüber dem switch gefordert, während die Einrückung der nachfolgenden Anweisungen gegenüber dem „case" durch den Parameter indentSize ebenfalls auf 4 Zeichen festgelegt wird. Im Beispiel aus Abbildung 5.2 wird im Unterschied zum ersten Beispiel keine Einrückung des „case" gegenüber dem „switch" gefordert (indentSwitchFromCase=0). Von daher stehen hier die Schlüsselwörter „switch" und „case" auf einer Ebene.

```
switch(a) {
case 1:
    System.out.println("one");
    break;
case 2:
    System.out.println("two");
    break;
}
```

Abbildung 5.2: Keine Einrückung von „case"

5.1.2 Pretty Printing Combinators von Haskell

Der erste Entwurf einer „Pretty-printing Library" für die funktionale Programmiersprache Haskell[3] stammt von John Hughes in [Hug95]. Im Vordergrund steht bei Hughes die These, daß durch Nutzung von Bibliotheken bei der Softwareentwicklung die Erstellung von fehlerfreier Software gefördert wird. Dazu bescheibt Hughes einen Prozeß zur Entwicklung von Bibliotheken für Haskell, und anhand einer Fallstudie entsteht die „Pretty-printing

[3] http://www.haskell.org

Library". Diese Bibliothek ist inzwischen ein Standardpaket von Haskell. Eine Variante, implementiert von Simon Peyton Jones (1997), existiert im Glasgow Haskell Compiler (vgl. [Wad98]).

Basierend auf den Überlegungen von Hughes beschreibt Philip Wadler in [Wad98] eine „Pretty-printing" Bibliothek, die das Konzept von Hughes vereinfacht. Beide vorgenannten Ansätze betrachtend, beschreiben Pablo R. Azero und S. Doaiste Swierstra in [PRA98] Verfeinerungen in einer weiteren Bibliothek. Im folgenden wird anhand des Ansatzes von P. Wadler der Aufbau einer „Pretty-printing" Bilbiothek von Haskell erläutert.

5.1.2.1 Aufbau der Bibliothek nach Wadler

In seinem Ansatz [Wad98] definiert Wadler sieben sogenannte Kombinatoren zum Umgang mit der Datenstruktur Doc (siehe Abbildung 5.3), die ein fertig gelayoutetes Dokument repräsentiert.

(<>)	:: Doc -> Doc -> Doc	(Konkatenation)
nil	:: Doc	(leeres Doc)
text	:: String -> Doc	(Wandlung String zu Doc)
line	:: Doc	(Zeilenumbruch)
nest	:: Int -> Doc -> Doc	(Einrückung)
group	:: Doc -> Doc	(Zeilenumbrüche in Doc optional)
pretty	:: Int -> Doc -> String	(Wandlung Doc zu String)

Abbildung 5.3: Kombinatoren zum Umgang mit der Datenstruktur Doc

Der Kombinator <> steht für die Konkatenation zweier Dokumente. Die Konkatenation ist assoziativ und hat das leere Dokument nil als neutrales Element. Des weiteren dient der Kombinator text zur Wandlung eines Strings in ein Doc, und der Kombinator line steht für einen Zeilenumbruch. Wadler setzt an dieser Stelle voraus, daß Zeichen, die einen Zeilenumbruch kodieren, innerhalb eines Strings, der mittels text zu einem Doc gewandelt wird, verboten sind. Damit hat ausschließlich der Kombinator line die Aufgabe Zeilenumbrüche zu kodieren. Der Kombinator nest rückt ein Doc um eine als Parameter zu übergebene Anzahl von Leerzeichen ein. Durch einen group Kombinator werden alle im Doc enthaltenen Zeilenumbrüche optional. Sie werden genau dann (alle) ausgeführt, wenn das Dokument nicht mehr in den noch in der Zeile vorhandenen Platz paßt. Dabei wird die Zeilenbreite dem Kombinator pretty als Parameter mitgeteilt und pretty wandelt das ebenfalls übergebene Doc gemäß der Zeilenbreite in einen (fertig gelayouteten) String zurück.

5.1.2.2 Anwendungsbeispiele für „Pretty-Printing" Kombinatoren

Als Anwendungsbeispiel[4] für die im vorangegangenen Abschnitt vorgestellten Kombinatoren wird im folgenden das Layouten von Instanzen einer Datenstruktur Tree gezeigt. Die Datenstruktur hat einen Konstruktor bezeichnet mit Node, der einen String und eine Liste von Trees als Parameter bekommt. Die Funktionen showTree, showList und showTrees übernehmen das Layouten einer Instanz vom Typ Tree.

```
data Tree              = Node String [Tree]

showTree (Node s ts) = text s <> nest (length s) (showList ts)

showList []            = nil
showList ts            = text "[" <> nest 1 (showTrees ts) <> text "]"

showTrees [t]          = showTree t
showTrees (t:ts)       = showTree t <> text "," <> line <> showTrees ts
```

Abbildung 5.4: Layoutdefinition der Datenstruktur Tree

Die voranstehenden Definitionen der Funktionen showTree, showList und showTrees erzeugen beispielsweise eine Ausgabe wie in Abbildung 5.5, falls eine Funktion length :: String –> Int zur Verfügung steht, die die Länge des übergebenen Strings als Ergebnis liefert.

```
aaa[bbbbb[ccc,
          dd],
    eee,
    ffff[gg,
         hhh,
         ii]]
```

Abbildung 5.5: Gelayouteter Baum

Während mit den Funktionsdefinitionen aus Abbildung 5.4 die Zeilenbreite keinen Einfluß auf das Layout hat, zeigt ein zweites Beispiel unter Benutzung des Kombinators group, wie Layouts in Abhängigkeit der Zeilenbreite variieren. Dazu ist im Gegensatz zur Definition aus Abbildung 5.4 lediglich die Funktion showTree, wie in Abbildung 5.6 gezeigt, zu ändern.

In Abhängigkeit der Zeilenbreite erzielt man mit der neuen Definition von showTree unterschiedliche Ausgaben, da alle Zeilenumbrüche im Wir-

[4] siehe auch [Wad98], Seiten 2ff.

showTree (Node s ts) = group(text s <> nest (length s) (showList ts))

Abbildung 5.6: Geänderte Funktion showTree

kungsbereich eines group-Kombinators nur dann ausgeführt werden, falls die Gruppe nicht mehr in die Zeile paßt. Die gleiche Ausgabe wie in Abbildung 5.5 erhält man bei einer Zeilenbreite von 15 Zeichen. Eine Zeilenbreite von 30 Zeichen führt bei dem gleichen Baum zu einer Ausgabe wie in Abbildung 5.7.

```
aaa[bbbbb[ccc, dd],
    eee,
    ffff[gg, hhh, ii]]
```

Abbildung 5.7: Gelayouteter Baum bei 30 Zeichen Zeilenbreite

5.1.3 PPML und Box

Die Pretty Printing Meta Language PPML wurde in ihrer ursprünglichen Form von E. Morcos-Chounet und A. Conchon entwickelt und 1986 im Artikel [EMC86] veröffentlicht. In bezüglich Notationen und Umfang geringfügig abgewandelter Form fließt PPML als Spezifikationssprache für einen Pretty Printer in das Centaur-System (siehe Kaptitel 3.4.3) ein. Der im Centaur-System implementierte Umfang wird in [Pro94] von 1994 beschrieben (Teil der Centaur Dokumentation). Basierend auf PPML entwickeln M. van den Brand und E. Visser 1996 in [MvdB96] die Sprache Box, die ähnliche Kombinatoren wie PPML bereitstellt. Im Unterschied zum Centaur System muß bei M. van den Brand und E. Visser die Spezifikation des Pretty Printers nicht manuell erstellt werden. Stattdessen wird zu einer Syntaxdefinition einer Programmiersprache automatisch ein Pretty Printer in der Sprache Box generiert. M. de Jonge verfeinert 2000 in ihrer Arbeit [Jon00] die Sprache Box. Dabei verwendet M. de Jonge ihre Definition von Box zur manuellen Definition von Pretty Printern. Sie vereinigt damit aus dem Centaur-System stammende Ideen mit den Ideen von M. van den Brand und E. Visser. Die Stärken des von M. de Jonge entwickelten Systems liegen in der Unabhängigkeit von einem umfassenden System (wie z.B. dem Centaur-System) und in der Möglichkeit der leichteren praktischen Einsetzbarkeit im Gegensatz zum System von M. van den Brand und E. Visser.

Gemeinsam ist oben genannten Arbeiten, daß der Pretty Printer aus einem programmiersprachenabhängiges Front-End und einem programmiersprachenunabhängiges Back-End besteht. In der folgenden Beschreibung der Syntax von PPML soll auf den Teil des Sprachumfangs von PPML aus

[EMC86] eingegangen werden, der für die vorliegende Arbeit interessant ist.

5.1.3.1 Sprachbeschreibung von PPML

Nach [EMC86] ist ein Pretty Printer eine geordnete Menge von Regeln. Eine Regel hat die Form, wie in Abbildung 5.8 gezeigt.

```
<pattern> -> <format>
```

Abbildung 5.8: Regel in PPML

Durch <pattern> wird ein Muster für Teilbäume des abstrakten Syntaxbaums definiert, für die die Formatierungsregel <format> gelten soll. Das Muster wird für eine Produktionsregel der abstrakten Syntax angegeben, die über die Nichtterminale der linken und rechten Seite der Syntaxregel identifiziert wird. Für alle Nichtterminale der rechten Produktionsregelseite kann entweder eine Variable (beginnend mit einem Stern) oder wiederum ein Muster stehen. Abbildung 5.9 zeigt als Beispiele ein einfaches und ein verschachteltes Muster in PPML.

(i) `if(*cond, *stat1, *stat2)` (einfaches Muster)

(ii) `if(eq(*a, *b), *stat1, *stat2)` (verschachteltes Muster)

Abbildung 5.9: Beispiele für Muster in PPML

Im <format>-Teil der Regel stehen zur Formulierung der Formatierungsrichtlinien alle Variablen des Musters und die Terminale, die in den Produktionsregeln der abstrakten Syntax auftreten, zur Verfügung. Diese werden mit sogenannten Kombinatoren, die die Formatierung spezifizieren, verknüpft. Abbildung 5.10 zeigt ein Bespiel für eine vollständige Regeldefinition. Dabei tauchen neben den Terminalen der abstrakten Syntaxregel für eine „if-then-else"-Anweisung und den Variablen aus dem Musterteil in der Formatierungsregel die Kombinatoren v und hv auf. Diesen Kombinatoren wird über die Klammerung in eckigen Klammern ein Wirkungsbereich zugewiesen.

Tabelle 5.1 zeigt eine Übersicht über die in PPML zur Verfügung stehenden Kombinatoren. Während h immer eine horizontale und v immer eine vertikale Komposition der folgenden Elemente fordert, steht hv für optionale horizontale oder vertikale Komposition in Abhängigkeit davon, ob die folgenden Elemente die zulässige Zeilenbreite überschreiten.

PPML enthält weitere Elemente zur Muster- und Formatierungsregelformulierung, die zu einer kompakteren Syntax führen, aber die Ausdrucksfähigkeit der Sprache nicht weiter verändern. Wie eingangs erwähnt, besteht ein

```
if(*cond, *stat1, *stat2)
-> [v [hv "if" *cond]
      [hv "then" *stat1]
      [hv "else" *stat2]
   ]
```

Abbildung 5.10: Beispiel für eine Regel in PPML

Name	Wirkung	Beschreibung
h	horizontal	Horizontale Verknüpfung der nachfolgenden Elemente
v	vertikal	Vertikale Verknüpfung der nachfolgenden Elemente
hv	horizontal oder vertikal	Horizontale Verknüpfung, falls nachfolgende Elemente die maximale Zeilenbreite nicht überschreiten; sonst vertikale Verknüpfung

Tabelle 5.1: Kombinatoren in PPML

Pretty Printer für eine Programmiersprache S aus einer geordneten Menge von Regeln $R(S)$. Der Pretty Printer arbeitet auf einen abstrakten Syntaxbaum t zu einem Programm formuliert in S wie folgt:

- Wähle die erste Regel, deren linke Seite auf t paßt.
- Instanziiere alle Mustervariablen der Regel mit den korrespondierenden Teilbäumen.
- Wende die rechte Seite der Regel an.
- Generiere rekursive Aufrufe für die instanziierten Mustervariablen.

Der beschriebene Algorithmus setzt voraus, daß für alle Elemente der abstrakten Syntax eine Regel in $R(S)$ existieren muß. da sonst rekursive Aufrufe möglicherweise nicht ausgeführt werden können. Des weiteren ist die Reihenfolge der Regeln in $R(S)$ wichtig. Steht eine Regel r_1 mit einer spezielleren linken Seite hinter einer Regel r_2 mit einem allgemeineren Muster auf der linken Seite, wird r_1 nie angewendet.

5.1.3.2 Kombinatoren von Box

Während die grundlegenden Prinzipien von PPML in der Sprache Box weiter verfolgt werden, ändern sich Umfang und Art der Möglichkeiten der Formulierung der Muster und der Formatierungsregeln. Die Muster von M. De

Jonge lassen keine Verschachtelung zu, womit für eine Regel der abstrakten Syntax immer nur eine Regel für den Pretty Printer angegeben werden und nicht zwischen spezielleren Instanzen unterschieden werden kann. Im folgenden werden die Kombinatoren der Sprache Box nach [Jon00] erläutert, die auch die Grundlage für die, in dieser Arbeit entworfenen, Kombinatoren bilden. Eine Neuerung sind Parameter, die den Kombinatoren angeheftet werden können, die die Größe des Leerraums steuern. Tabelle 5.2 zeigt eine Übersicht über die Kombinatoren der Sprache Box.

Name	Parameter	Wirkung	Beschreibung
H	hs	horizontal	Horizontale Verknüpfung der nachfolgenden Elemente
V	vs, is	vertikal	Vertikale Verknüpfung der nachfolgenden Elemente
HV	hs, vs, is	horizontal oder vertikal	Horizontale Verknüpfung, falls nachfolgende Elemente die maximale Zeilenbreite nicht überschreiten; sonst vertikale Verknüpfung
ALT		alternativ	Abhängig von der Zeilenbreite Formatierung nach der ersten oder der zweiten Subbox

Tabelle 5.2: Kombinatoren in Box

Ein Ausdruck in der Sprache Box setzt sich wie folgt zusammen:

- Ein String ist eine elementare Box.
- Durch Verknüpfung von Boxen mit einem Kombinator entsteht wieder eine Box.

Abbildung 5.11 zeigt die gleiche Regel, die in Abbildung 5.10 mit PPML formuliert wurde, in einer Formulierung in der Sprache Box. Die Musterformulierung wird nach M. de Jonge mittels einer Syntaxregel in SDF nach [HHKR89] vorgenommen (in Abbildung 5.11 links von —).

```
"if" Expression "then" Statement "else" Statement → Statement —
-> V vs=0 [ HV hs=1 vs=0 is=0 ["if" _1]
        HV hs=1 vs=0 is=0 ["then" _2]
        HV hs=1 vs=0 is=0 ["else" _3]
    ]
```

Abbildung 5.11: Beispiel für eine Regel in Box

In der Regelformulierung korrespondieren die Platzhaltersymbole _1, _2 und _3 mit den Nichtterminalen der Syntaxregel des Musters. Steht in einer Formatierungsregel ein Platzhalter, erfolgt an dieser Stelle ein rekursiver Aufruf für den korrespondierenden Teilbaum. Die Aufstellung in Abbildung 5.12 der Kombinatoren zeigt anschaulich die Wirkung der Kombinatoren.

$$\mathsf{H}\ [\boxed{B_1}\ \boxed{B_2}\ \boxed{B_3}] = \boxed{B_1}\ \boxed{B_2}\ \boxed{B_3}$$

$$\mathsf{V}\ [\boxed{B_1}\ \boxed{B_2}\ \boxed{B_3}] = \begin{array}{c}\boxed{B_1}\\ \boxed{B_2}\\ \boxed{B_3}\end{array}$$

$$\begin{aligned}\mathsf{HV}\ [\boxed{B_1}\ \boxed{B_2}\ \boxed{B_3}] = {} & \mathsf{H}\ [\boxed{B_1}\ \boxed{B_2}\ \boxed{B_3}]\ \mathsf{oder}\\ & \mathsf{V}\ [\mathsf{H}\ [\boxed{B_1}\ \boxed{B_2}]\ \boxed{B_3}]\ \mathsf{oder}\\ & \mathsf{V}\ [\boxed{B_1}\ \mathsf{H}\ [\boxed{B_2}\ \boxed{B_3}]]\ \mathsf{oder}\\ & \mathsf{V}\ [\boxed{B_1}\ \boxed{B_2}\ \boxed{B_3}]\end{aligned}$$

$$\mathsf{ALT}\ [\boxed{B_1}\ \boxed{B_2}] = \boxed{B_1}\ \mathsf{oder}\ \boxed{B_2}$$

Abbildung 5.12: Wirkung der Kombinatoren in Box (anschaulich)

Der Kombinator HV versucht, die Zeilenbreite möglichst gut zu nutzen und wählt die erste der angeführten Formatierungen, die in die Zeile paßt. Der Kombinator HV gehört nicht zu einem minimalen Satz von Kombinatoren, da HV durch ALT, V und H ausgedrückt werden kann. Abbildung 5.13 zeigt dies für den HV Kombinator, in dessen Wirkungsbereich drei Elemente liegen.

$$\begin{aligned}\mathsf{HV}\ [\boxed{B_1}\ \boxed{B_2}\ \boxed{B_3}] = {} & \mathsf{ALT}\ [\mathsf{H}\ [\boxed{B_1}\ \boxed{B_2}\ \boxed{B_3}]\\ & \quad \mathsf{ALT}\ [\mathsf{V}\ [\mathsf{H}\ [\boxed{B_1}\ \boxed{B_2}]\ \boxed{B_3}]\\ & \quad\quad \mathsf{ALT}\ [\mathsf{V}\ [\boxed{B_1}\ \mathsf{H}\ [\boxed{B_2}\ \boxed{B_3}]]\\ & \quad\quad\quad \mathsf{V}\ [\boxed{B_1}\ \boxed{B_2}\ \boxed{B_3}]]]]\end{aligned}$$

Abbildung 5.13: Ausdrücken von HV durch ALT, H und V

5.1.4 Bewertung der Ansätze

Der Formalismus, den JIndent nutzt, erlaubt von allen vorgestellten Formalismen die Spezifikation der meisten, der für diese Arbeit relevanten, Programmierrichtlinien. Das Konzept, das hinter der Formulierung der Programmierrichtlinien steht, ist jedoch schwer auf andere Programmiersprachen übertragbar. Für jede Programmiersprache müßte eine vorangehende Analyse die benötigten Parameter bestimmen und ihre genaue Wirkung definieren. Damit ist aber ein programmiersprachenabhängiges Back-End nötig. Da in dieser Arbeit ein auf Unabhängigkeit von der zugrundeliegenden Programmiersprache basierendes Konzept entwickelt werden soll, führen Formalismen nach der Art von JIndent nicht zum Ziel.

Die Pretty Printer Kombinatoren von Haskell erlauben die Formulierung von Formatierungen für beliebige Datenstrukturen. Über den group Kombinator können die Formatierungen in Abhängigkeit von der zur Verfügung stehenden Zeilenbreite gewählt werden, was bei der Formulierung von Formatierungsrichtlinien wichtig ist. Eine Datenstruktur, die einen Programmquelltext repäsentiert, ist der abstrakte Syntaxbaum. Für eine Verfolgung eines Ansatzes in Anlehnung an Haskell's Pretty Printing Combinators müßte also zunächst eine Darstellung der abstrakten Syntax geschaffen werden, für die dann über die Angabe von Funktionen mit den Kombinatoren ein Layout definiert werden könnte. Allerdings kann der Umfang der Grammatik einer realen Programmiersprache dazu führen, daß sich durch die Vielzahl von Funktionen zur Umsetzung der Formatierungsrichtlinien (allein in Beispiel 5.4 werden zur Formatierung eines Baumes mit einem Konstruktor drei sich rekursiv aufrufende Funktionen benötigt) die Formulierung sehr komplex gestaltet.

Ein modularerer Aufbau der Formatierungsrichtlinien wird durch die Formulierung mit PPML erreicht. Dabei wird Teilbäumen, identifiziert über eng an der abstrakten Syntax formulierte Muster, eine Formatierung zugewiesen, die über Kombinatoren formuliert wird. Das Ändern und insbesondere das Hinzufügen einer speziellen Formatierung für bestimmte Konstrukte der Programmiersprache ist einfacher als bei einer Formulierung mit den Pretty Printing Kombinatoren von Haskell.

Bei den bestehenden Implementierungen von Systemen, die sich PPML oder der daraus abgeleiteten Sprachen bedienen, ist es allerdings nicht möglich vorhandene Formatierungen zu behalten. Dieser Aspekt wird bei der Entwicklung der Beschreibungssprache dieser Arbeit berücksichtigt. Insgesamt ist PPML eine gute Grundlage für eine Sprache, die der Beschreibung von Programmierrichtlinien dient. Der Abschnitt 5.2 beschreibt, aufbauend auf PPML und Box, eine Sprache, mit der es möglich ist, Programmierrichtlinien anhand der abstrakten Syntax zu formulieren.

5.2 Aufbau der Beschreibungssprache CCML

Die Beschreibungssprache CCML - Code Convention Meta Language - ist in der vorliegenden Arbeit in Anlehnung an die Sprache PPML entstanden. CCML integriert die ausgeprägten Formalismen von PPML zur Spezifikation von Formatierungen mit neu entwickelten Formalismen zur Beschreibung von weiteren Programmierrichtlinien. Neu geschaffen wurde auch die Möglichkeit Quelltextformatierungen unverändert zu lassen, wozu neben der Erweiterung des Sprachumfanges von PPML auch die Kodierung der originalen Formatierungen im AST notwendig ist (vgl. Abschnitt 4.2.2).

Eine Spezifikation von Programmierrichtlinien in CCML setzt sich aus zwei Teilen zusammen. Im ersten Teil werden globale Festlegungen getroffen. Der anschließende zweite Teil beinhaltet die eigentlichen Programmierrichtlinien, deren Formulierung an Regeln in PPML erinnert. Die Startproduktion der Grammatik einer CCML-Spezifikation findet sich in Abbildung 5.14.

```
CCML-Spec
= Globals Rules ;
```

Abbildung 5.14: Unterteilung der CCML-Spezifikation in zwei Abschnitte

5.2.1 Globaler Anteil

In dem Abschnitt 4.2.1 wurde auf die Problematik der verschiedenen Whitespace-Klassen eingegangen. Den dabei definierten Klassen für horizontale und vertikale Trennung sowie Block- und Zeilenendekommentaren wird im globalen Teil der Programmrepräsentation je eine geordnete Menge von Elementen zugeordnet. Die Zuordnung für Java findet sich in den Tabellen 4.2 und 4.3 wieder.

Die Sprache CCML unterscheidet wie auch PPML bei den Formatierungen lediglich zwischen horizontaler und vertikaler Trennung und trifft keine weiteren Unterscheidungen bezüglich verschiedener Trennzeichen. Das jeweils bezüglich der Ordnung der Menge kleinste Zeichen der Klassen für horizontale und vertikale Trennung, das in der Programmrepräsentation definiert ist, ist das Zeichen, das mit einem durch CCML spezifizierten Trennzeichen korrespondiert. Für eventuell andere auftretende Trennzeichen der genannten Klassen wird eine Abbildung dieser Zeichen auf das bezüglich der Ordung kleinste Zeichen zur Prüfung von Leerraum benötigt. Im globalen Teil einer CCML-Spezifikation können genau diese Abbildungen spezifiziert werden. Als Beispiel kann hier also spezifiziert werden, daß ein Tabulator bei der Prüfung von Leerraum auf vier Leerzeichen abgebildet wird, falls das Leerzeichen das kleinste Zeichen bezüglich der Ordnung der Menge der horizontalen Trennzeichen ist.

Neben den beschriebenen Abbildungen, gibt es im globalen Teil die Möglichkeit, die Zeilenbreite zu spezifizieren. Formatierungen, die diese Zeilenbreite überschreiten, sind unzulässig. Sind alternative Formatierungen angegeben, muß also diejenige gewählt werden, die die zulässige Zeilenbreite nicht überschreitet. Abbildung 5.15 zeigt die Grammatik für den globalen Teil der CCML-Spezifikation.

```
Globals
= 'Globals' {HTranslation} {VTranslation}
  {BlockCommentRole} {EOLCommentRole} [LineWitdhDefiniton] ;

HTranslation
= 'htrans' 'type' '=' Number 'number' '=' Number;

VTranslation
= 'vtrans' 'type' '=' Number 'number' '=' Number;

BlockCommentRole
= 'blockCommentRole' 'type' '=' Number role '=' ('pre' | 'post')

EOLCommentRole
= 'eolCommentRole' 'type' '=' Number role '=' ('pre' | 'post')

LineWitdhDefiniton
= 'linewidth' '=' Number;
```

Abbildung 5.15: Globale Festlegungen in CCML

Schließlich können im globalen Teil einer CCML Spezifikation auch noch Rollenzuweisungen für die verschiedenen Kommentarstile vorgenommen werden. Eine ausführliche Diskussion der Rollenzuweisung zu den Kommentarstilen findet sich in Abschnitt 4.4.

Alle Angaben im globalen Teil der Spezifikation sind optional. Falls keine Angabe erfolgt, wird eins zu eins in bezug auf die Trennzeichen abgebildet, und die Zeilenbreite ist unbegrenzt. Des weiteren wird Blockkomentaren als Voreinstellung die Rolle der vorangestellten und Zeilenendekommentaren die der nachgestellten Erläuterung zugewiesen. Abbildung 5.16 zeigt für Java typische globale Festlegungen unter der Annahme der ebenfalls in der Abbildung gezeigten globalen Festlegungen in der Programmrepräsentation. Die Definition der Übersetzungen in den jeweils ersten Typ besagt, daß ein Tabulator in vier Leerzeichen und ein Formfeed in zwei Zeilenumbrüche gewandelt wird. Des weiteren wird Blockkommentaren die Rolle der vorangestellten Erläuterung und Zeilenendekommentaren die Rolle der nachgestellten Erläuterung zugewiesen, und die maximale Zeilenbreite auf 80 Zeichen festgelegt.

5.2.2 Regeln

Der zweite Abschnitt einer CCML-Spezifikation besteht aus einer beliebigen Anzahl von Regeln. Eine Regel besteht aus bis zu vier Teilen. Abbildung 5.17

(i) Globale Festlegungen in der Programmrepräsentation

```
hWhitespace value="␣"
hWhitespace value="\t"
vWhitespace value="\n"
vWhitespace value="\r"
vWhitespace value="\r\n"
vWhitespace value="\f"
blockComment begin="/*"end="*/"
eolComment begin="//"end="\r\n"
eolComment begin="//"end="\n"
eolComment begin="//"end="\r"
```

(ii) Globale Festlegungen in der CCML-Spezifikation

```
htrans type=1 number=4
vtrans type=3 number=2
blockCommentRole type=0 role=pre
eolCommentRole type=0 role=post
eolCommentRole type=1 role=post
eolCommentRole type=2 role=post
linewidth = 80
```

Abbildung 5.16: Globale Festlegungen für Java

zeigt die Grammatik für den Regelteil.

Ähnlich wie bei PPML besteht eine Regel im wesentlichen aus einem Muster (Pattern) und der Definition der Programmierrichtlinie (Convention) für die Elemente, auf die das Muster paßt. Zusätzlich besteht die Möglichkeit über einen sogenannten Guard das Muster (oder die Konvention s.u.) weiter zu spezialisieren. Ist ein Guard angegeben, kann eine Aktion spezifiziert werden, die im Falle eines nicht erfüllten Guards ausgeführt wird. Die Begriffe Muster und Pattern sowie Konvention und Convention werden im folgenden synonym benutzt.

Die Reihenfolge der Regeldefinition ist relevant bei der Prüfung auf Einhaltung von Programmierrichtlinien. Die erste auf den gerade zu prüfenden Teilbaum passende Regel wird angewendet. Nach Definition 4.4.1 besteht eine Regel $p = (b_1, b_2)$ aus zwei Baummustern. Für einen Teilbaum t des abstrakten Syntaxbaumes des zu prüfenden Programms wird versucht, t mit dem Baummuster b_1 einer Regel p aus der geordneten Menge von Regeln zu überdecken. Wird t von b_1 überdeckt, wird geprüft, ob b_2 t überdeckt. Falls t von b_2 überdeckt wird, sind die in der Regel spezifizierten Programmierrichtlinien erfüllt, sonst werden sie verletzt.

Je nach Formulierung der Regel setzen sich b_1 und b_2 unterschiedlich zusammen. Tabelle 5.3 gibt einen Überblick darüber, welche der Regelteile

```
Rules
= 'Rules' {Rule} ;

Rule
= NoGuardRule
| GuardRule ;

NoGuardRule
= Pattern '->' Convention ';' ;

GuardRule
= Pattern '|' Guard '->' Convention ['|' GuardNotMatchedAction] ';' ;
```

Abbildung 5.17: Globale Festlegungen in CCML

in Abhängigkeit von der Regelformulierung welche Baummuster bestimmen.

Regeltyp	b_1 bestimmt durch	b_2 bestimmt durch
Pattern –> Convention	Pattern	Convention
Pattern | Guard –> Convention	Pattern und Guard	Convention
Pattern | Guard –> Convention | GuardNotMatchedAction	Pattern	Convention und Guard

Tabelle 5.3: Zuordnung der Regelteile zu den Baummustern b_1 und b_2

Im einfachsten Fall enthält die Regel nur Pattern und Convention. Dann ergibt sich b_1 aus dem Pattern und b_2 aus dem Convention Anteil. Schwieriger wird es beim Auftreten eines Guards. Hier wird weiter unterschieden, ob eine GuardNotMatchedAction angegeben wurde oder ob diese fehlt. Fehlt die GuardNotMatchedAction, dann ergibt sich b_1 aus dem Pattern und dem Guard, und das Baummuster b_2 ergibt sich wie zuvor aus dem Convention Anteil. In diesem Fall ist also der Guard als reine Spezialisierung des Patterns zu interpretieren, und die Programmierrichtlinie kommt nur dann zur Prüfung, wenn der zu prüfende Teilbaum den Anforderungen aus Pattern und Guard genügt. Im verbliebenen Fall, daß eine GuardNotMatchedAction angegeben wurde, ist die Interpretation des Guards anders. Der Guard wird in diesem Fall dem Baummuster b_2 zugerechnet, das sich damit aus Convention und eben dem Guard zusammensetzt. Das Baummuster b_1 wird nur durch das Pattern bestimmt. Als GuardNotMatchedAction sind nur

Aktionen zugelassen, die keine Veränderung des abstrakten Syntaxbaumes bedeuten. Zunächst ist nur die Ausgabe von Warnmeldungen vorgesehen, als Ausblick sind aber weitere Funktionen denkbar. Die Warnmeldungen nach der GuardNotMatchedAction werden nur dann ausgegeben, wenn der Guard nicht erfüllt wurde. Technisch bedeutet dies, daß, falls b_2 den von b_1 überdeckten Teilbaum nicht überdeckt, geprüft werden muß, ob der Guard Grund für diese Nichtüberdeckung ist, um entscheiden zu können, ob die GuardNotMatchedAction ausgeführt werden soll. Dazu wird versucht den von b_1 überdeckten Teilbaum mit einem Baummuster b_2' zu überdecken, daß von Pattern und Guard bestimmt wird (es gilt dabei $b_2' \rhd b_2$).

5.2.2.1 Muster (Pattern)

Ein Muster dient zur Identifikation von Teilbäumen im abstrakten Syntaxbaum, wobei die Wurzel eines solchen Teilbaumes ein Nichtterminal der abstrakten Syntax ist.

Definition 5.2.1 (Muster)
Ein Muster m ist ein Tupel (n, M) mit $M = (m_1, \ldots, m_l), l \geq 1$, wobei n Nichtterminalssymbol ist, und m_i ist

- Terminalsymbol 't'
- Mustervariable v
- Ein Muster, eingefaßt in geschweifte Klammern

Die Notationen der Nichtterminalsymbole und Terminalsymbole sind wie in der abstrakten Syntax. Nichtterminalsymbole beginnen mit einem Großbuchstaben und Terminalsymbole enthalten ihren Wert in einfachen Hochkommata. Eine Mustervariable besteht aus einer beliebigen Zeichenkette aus den Buchstaben a-z und A-Z und Ziffern 0-9, beginnend mit einem Buchstaben.

```
Pattern
= nonterminal PatternTail ;

PatternTail
= terminal
 | variable
 | '{' Pattern '}' ;
```

Abbildung 5.18: Grammatik für Muster

In Abbildung 5.18 ist die Syntax der Musterformulierung nochmals in Form einer Grammatik in EBNF-Notation aufgeführt. Die Elemente „nonterminal", ,,terminal" und ,,variable" sind Terminale in der Grammatik von CCML (siehe Anhang A). Die Beispiele (i)-(iii) aus Abbildung 5.19 zeigen verschiedene Muster für ein Statement. Das Beispiel aus (i) zeigt ein Muster, das auf jede Instanz eines ,,if-then-Statements" in Java paßt. Dabei sind 'if', '(' und ')' die Terminalsymbole und cond sowie stat Mustervariablen des Musters.

(i) Statement 'if' '(' cond ')' stat

(ii) Statement 'if' '(' cond ')' {Statement {Block '{' statList '}'}}

(iii) Statement 'if' '(' {Expression {Expression {Name id}} binop exp} ')' {Statement {Block '{' statList '}'}}

Abbildung 5.19: Beispiele für Muster

Das Muster aus (ii) hat anstelle der Mustervariablen stat ein nochmal in sich verschachteltes tieferes Muster. Damit paßt dieses Muster nur auf Instanzen von ,,if-then-Statements", wenn das im ,,then"-Teil folgende Statement ein Block ist. In (iii) ist im Gegensatz zu (ii) zusätzlich ein verschachteltes, tieferes Muster für die Bedingung des ,,if-then-Statements" angegeben.

Würden diese Muster in Regeln einer CCML-Spezifikation auftauchen, dürfte das Muster aus (i) nicht vor den Mustern aus (ii) und (iii) auftreten. Sonst würden die Regeln mit den Mustern (ii) und (iii) nie zur Anwendung kommen, da (i) auf alle Instanzen eines ,,if-then-Statements" paßt.

Interpretation als Baummuster

Nach Tabelle 5.3 bestimmt das Muster (Pattern) (möglicherweise zusammen mit einem Guard) das Baummuster b_1 einer Regel $p = (b_1, b_2)$. Während der Guard ein Baummuster durch Setzen von weiteren Attributen spezialisieren kann, wird die Struktur von b_1 wird durch das Muster bestimmt. Ein Muster $m = (n, M)$ mit $M = (m_1, \ldots, m_l)$ wird in ein Baummuster $b = (r, T)$ mit $T = (b_1, \ldots, b_l)$ umgesetzt, indem

- r ein Knoten mit $r = (n, \emptyset)$ ist (keine Attribute)
- Falls m_i='t' mit $1 \leq i \leq l$ ein Terminalsymbol ist $\Rightarrow b_i = (k, \emptyset)$, wobei $k = (\epsilon, ((value, \mathrm{t})))$ ein Knoten mit Attribut $(value, \mathrm{t})$ ist.
- Falls $m_i = \mathrm{v}$ mit $1 \leq i \leq l$ eine Mustervariable ist $\Rightarrow b_i = (k, \emptyset)$, wobei $k = (\mathrm{v}, \emptyset)$ ein Knoten ist.
- Falls m_i mit $1 \leq i \leq l$ ein Muster ist $\Rightarrow$ rekursives Umsetzen von Muster m_i in Baummuster b_i

Die Abbildung 5.20 zeigt die Umsetzung des Musters von Beispiel (ii) aus Abbildung 5.19.

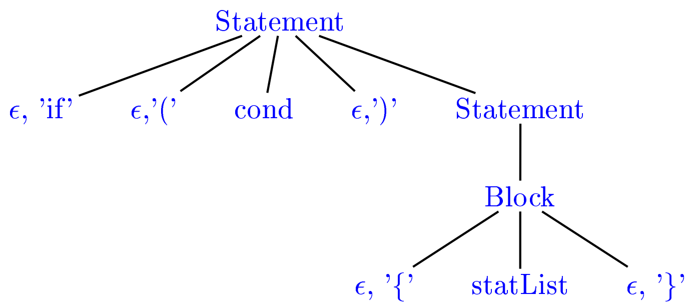

Abbildung 5.20: Baummuster umgesetzt aus Muster (ii), Abb. 5.19

5.2.2.2 Konvention (Convention)

Im Konventionsteil einer Regel werden Richtlinien für den Teilbaum spezifiziert, der über das Muster (evtl. Muster und Guard) ausgewählt wurde. Mit den Mechanismen, die in diesem Abschnitt beschrieben werden, ist zunächst nur die Formulierung von Formatierungsrichtlinien möglich. Während Namenskonventionen mithilfe von Guards formuliert werden können, wird auf eine mögliche Erweiterung des Sprachumfangs von CCML im Hinblick auf die Programmierrichtlinienklassen „Reihenfolge von Deklarationen" sowie „Verbot/Gebot von Redundanzen" in einem Ausblick in Abschnitt 7.3 eingegangen.

Definition 5.2.2 (Konvention)
Eine *Konvention* ist

- ein Terminalsymbol 't'.
- eine Mustervariable v.
- ein H-Kombinator mit optionalem Parameter hs, in dessen Wirkungsbereich (gekennzeichnet durch eckige Klammern) beliebig viele Konventionen stehen können.
- ein V-Kombinator mit optionalen Parametern vs und is, in dessen Wirkungsbereich (gekennzeichnet durch eckige Klammern) beliebig viele Konventionen stehen können.
- ein VL-Kombinator mit optionalen Parametern vs und is, in dessen Wirkungsbereich (gekennzeichnet durch eckige Klammern) beliebig viele Konventionen stehen können.
- ein ID-Kombinator, in dessen Wirkungsbereich (gekennzeichnet durch eckige Klammern) beliebig viele Konventionen stehen können.

- ein O-Kombinator, in dessen Wirkungsbereich (gekennzeichnet durch eckige Klammern) durch Kommata abgetrennte alternative Konventionen stehen können.

Die Notationen der Terminalsymbole und Mustervariablen sind wie im Muster. Den Parametern der Kombinatoren kann ein ganzzahliger Wert zugewiesen werden (z.B. hs=1). Über die Kombinatoren werden horizontale und vertikale Kompositionen der Elemente ausgedrückt. Die Syntax von Konventionen wird duch die Grammatik in Abbildung 5.21 beschrieben.

```
Convention
= terminal
 | variable
 | Combinator ;

Combinator
= 'H' ['hs=' Number] '[' {Convention} ']'
 | 'V' ['vs=' Number] ['is=' Number] '[' {Convention} ']'
 | 'VL' ['vs=' Number] ['is=' Number] '[' {Convention} ']'
 | 'ID' '[' {Convention} ']'
 | 'O' '[' [Convention {',' Convention}] ']' ;
```

Abbildung 5.21: Grammatik für Konventionen

Semantik der Kombinatoren

Die folgende Aufstellung beschreibt die Semantik der Kombinatoren H, V, VL, ID und O. Bei der horizontalen bzw. vertikalen Trennung der Elemente sind, wie in Abschnitt 4.2.1 beschrieben, das bezüglich der Ordnung kleinste horizontale bzw. vertikale Trennzeichen relevant. Die Angabe der Parameter bei der Konventionsformulierung ist optional. Jeder Parameter hat eine Voreinstellung, die bei nicht angegebenem Parameter Verwendung findet.

Generell hat ein Kombinator Wirkung auf die Zwischenräume zwischen den Elementen im Wirkungsbereich des Kombinators. Dazu wird zu einem Element (also einem Terminal, einer Variable oder wiederum einem Kombinator) der vor diesem Element liegende Leerraum spezifiziert. Handelt es sich bei dem Element um einen Kombinator, so gehört der Leerraum, der als vor diesem Kombinator liegend spezifiziert wurde, zum ersten Element im Wirkungsbereich dieses Kombinators. Handelt es sich hingegen um ein Terminal oder eine Variable, so wird der Leerraum direkt diesem Element zugeordnet.

H hs=d_h $[c_1 \ldots c_n]$ Zwischen je zwei im Wirkungsbereich liegenden Elementen c_i und c_{i+1}, $1 \leq i \leq n-1$ wird ein horizontaler Leerraum von d_h horizontalen Trennzeichen gefordert. Die Voreinstellung für hs ist 1.

V vs=d_v **is=**d_i $[c_1 \dots c_n]$ Zwischen je zwei im Wirkungsbereich liegenden Elementen c_i und c_{i+1}, $1 \leq i \leq n-1$ Elementen wird ein vertikaler Leerraum von d_v+1 vertikalen Trennzeichen und einer folgenden Einrückung von d_i horizontalen Trennzeichen gegenüber dem linken Einzug von c_1 gefordert. Die Voreinstellung für vs und is ist jeweils 0.

VL vs=d_v **is=**d_i $[c_1 \dots c_n]$ Zwischen je zwei im Wirkungsbereich liegenden Elementen c_i und c_{i+1}, $1 \leq i \leq n-1$ Elementen wird ein vertikaler Leerraum von d_v+1 vertikalen Trennzeichen und einer folgenden Einrückung von d_i horizontalen Trennzeichen gegenüber dem linken Einzug des ersten Elements der Zeile, in der c_1 steht, gefordert. Die Voreinstellung für vs und is ist jeweils 0.

ID-Kombinator $[c_1 \dots c_n]$ Es wird keine Forderung an den Leerraum zwischen den Elementen c_1 bis c_n gestellt.

O-Kombinator $[c_1, \dots, c_n]$ Je nach im globalen Teil angegebener Zeilenbreite, werden die Anforderungen nach der ersten in die Zeilenbreite passenden Konvention der angegebenen Alternativen an den Leerraum gestellt.

Eine Frage des Stils ist die Reihenfolge der Prüfung optionaler Layouts, die bei einer Verletzung der Zeilenbreite potentiell Einfluß auf die Zeilenbreite nehmen können. Als Beispiel könnte die Zeilenbreite durch ein Element im Wirkungsbereich eines O-Kombinators verletzt werden, der wiederum im Wirkungsbereich eines weiteren O-Kombinators liegt. Eine umfassende Untersuchung dieses Problematik soll an dieser Stelle (nur) als Ausblick auf fortführende Arbeiten genannt werden. Eine mögliche Strategie wird in der Beschreibung der Implementierung in Abschnitt 6.3.3.2 erläutert.

Beispiele für Konventionen

Abbildung 5.22 zeigt Beispiele für Konventionen, die mit den vorgestellten Kombinatoren formuliert wurden. Die Beispiele (i)-(iii) spezifizieren Richtlinien für ein „if-then"-Statement nach dem folgenden Muster:
Statement 'if' '(' exp ')' stat

Beispiel (iv) zeigt eine etwas komplexere Formulierung für ein „if-then"-Statement mit nachfolgenden Block, passend auf das folgende Muster:
Statement 'if' '(' exp ')' {Statement {Block '{' statList '}'}}
Das Beispiel (v) dient zur Demonstration der Verwendung eines VL-Kombinators für einen „ConstructorBody" der Form:
ConstructorBody '{' statList '}'

```
(i)   H ['if' H hs=0 ['(' exp ')'] stat]
(ii)  V is=4 [H ['if' H hs=0 ['(' exp ')']] stat]
(iii) O [H ['if' H hs=0 ['(' exp ')'] stat]
        V is=4 [H ['if' H hs=0 ['(' exp ')']] stat]]
(iv)  V [
        V is=4 [H ['if' H hs=0 ['(' exp ')'] '{']
                statList]
        '}']
(v)   VL is=0 [VL is=4 ['{' statList] '}']
```

Abbildung 5.22: Beispiele für Konventionen

Für den ID-Kombinator ist keine Richtlinie angegeben, da dieser im allgemeinen nur vom System selber verwendet wird, um Regeln für die Elemente der abstrakten Syntax zu formulieren, für die der Benutzer keine Regeln formuliert hat.

Die durch die obigen Konventionen spezifizierten Formatierungen werden in Abbildung 5.23 beispielhaft dargestellt. Dabei werden für die Variablen konkrete Instanzen gewählt (z.B. ist exp instanziiert mit a == 1). Konvention (iii) wählt zwischen den alternativen Layout (i) und (ii) in Abhängigkeit von der Zeilenbreite. Die Konvention für die „ConstructorBody" gemäß (v) wird im Kontext einer Konstruktordeklaration der Klasse „Testclass" gezeigt.

```
(i)   if (a == 0) b = 1;

(ii)  if (a == 0)
          b = 1;

(iii) wie (i) falls Zeilenbreite größer als 17 Zeichen,
      wie (ii) falls Zeilenbreite zwischen 11 und 17 Zeichen

(iv)  if (a == 0) {
          b = 1;
      }

(v)   public Testclass() {
          doSomething();
          ...
      }
```

Abbildung 5.23: Spezifizierte Formatierungen

Interpretation als Baummuster

In der Formulierung einer Konvention sind nur die Mustervariablen und Terminalsymbole des Musters enthalten, während sich die durch die abstrakten Syntaxregeln definierte Baumstruktur nicht in der Konventionsformulierung wiederfindet. Um die Konvention in ein Baummuster umzusetzen, müssen die Baumstrukturen bekannt sein. Von daher ergibt sich das Baummuster b_2 einer Regel $p = (b_1, b_2)$ zunächst als Kopie des Teils des Baummusters b_1, der durch das Muster bestimmt wurde, um die notwenige Baumstruktur zu erhalten.

Wie in der Semantikbeschreibung erklärt, wird durch eine Konvention jedem Terminal und jeder Variablen ein vor ihnen liegender Leerraum zugeordnet. Davon ausgenommen sind die Terminale und Variablen für die kein voranstehender Leerraum spezifiziert wurde, weil sie sich im Wirkungsbereich ID-Kombinators befinden (oder sie sind erstes Element eines im Wirkungsbereich eines ID-Kombinators liegenden Kombinators.)

Zu jedem Blatt im bisher nur durch das Muster bestimmten Baummuster b_2 existiert ein korrespondierendes Element in der Konvention. Die mit den Blättern korrespondierenden Elemente sind entweder die Terminale (Knoten der Form $(\epsilon, (value, t))$) oder Variablen (Knoten der Form $(n, \emptyset)$). Die im Baummuster mit den Terminalen und Variablen korrespondierenden Blätter erhalten zur Umsetzung der Konvention ein Attribut „whitespace", das als Wert die Funktion $wsMatch$ (siehe Abschnitt 4.5.1) erhält, die als Parameter den vor dem Terminal bzw. der Variablen spezifizierten Leerraum hat. Falls aufgrund von ID-Kombinatoren zu bestimmten Terminalen und Variablen kein Leerraum spezifiziert wurde, erhalten die mit diesen Terminalen und Variablen korrespondierenden Blätter kein Attribut. Abschließend werden im Baummuster die „whitespace"-Attribute wie im AST nach außen transferiert, indem das „whitespace"-Attribut (falls vorhanden) des linkesten Kindknotens eines inneren Knotens auf den inneren Knoten übertragen wird. Der Kindknoten selber erhält daraufhin das Attribut $(whitespace, (wsMatch, ""))$.

Die Abbildung 5.24 zeigt das zu der Konvention (iv) aus Abbildung 5.22 umgesetzte Baummuster. Dabei finden sich im Baum nur die Parameterwerte der Funktion $wsMatch$ des Attributes „whitespace", symbolisiert durch einen sich aus den Symbolen ␣ für ein horizontales Trennzeichen und $\hookleftarrow$ für ein vertikales Trennzeichen zusammengesetzten in Anführungszeichen stehenden Wert, wieder. Knoten ohne einen solchen Zusatz haben kein Attribut „whitespace".

Vergleicht man das neu entstandene Baummuster b_2 mit dem Baummuster b_1 aus Abbildung 5.20, stellt man fest, daß b_1 b_2 überdeckt. Dies ist damit begründet, daß bei der Konstruktion des Baummusters b_2 einer Kopie des Baummusters b_1 nur weitere Attribute hinzugefügt wurden. Es wurden weder Attribute entfernt noch die Struktur des Musters verändert.

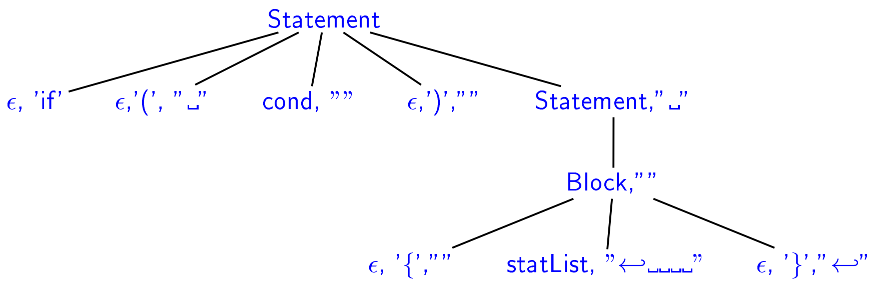

Abbildung 5.24: Baummuster umgesetzt aus Konvention (iv), Abb. 5.22

5.2.2.3 Guard

Mittels eines Guards können zusätzliche Anforderungen im Hinblick auf die Auswahl des zu prüfenden Teilbaumes oder die zu prüfende Richtlinie gestellt werden. Wozu der Guard gerechnet wird, ist abhängig davon, ob eine GuardNotMatchedAction spezifiziert wurde. Fehlt diese, ist der Guard als ein Attribut des Musters zu verstehen (bestimmt damit b_1 mit), nach dem der auf Einhaltung der Konvention zu prüfende Teilbaum ausgewählt wird. Ist eine GuardNotMatchedAction angegeben, ist der Guard Attribut der Konvention (und bestimmt damit b_2 mit).

Die Syntax eines Guards wird über den Grammatikausschnitt aus Abbildung 5.25 angegeben. In dieser Arbeit wird nur eine Funktion zur Formulierung von Guards bereitgestellt. Mögliche Erweiterungen des Umfangs von CCML werden im Abschnitt 7.3 diskutiert.

```
Guard
= GuardFunction
= '( GuardFunction '&&' Guard ')' ;

GuardFunction
= regexMatch '(' AttributName ',' '"' {Char} '"' ')' ;

AttributName
= {Char} '.' {Char} ;
```

Abbildung 5.25: Grammatik für Guard

Die Funktion *regexMatch* dient zur Prüfung von Namenskonventionen. Als Argumente bekommt die Funktion einen Attributnamen und einen regulären Ausdruck übergeben. Der Attributname, muß der Attributname eines Blattes des Baummusters b_1 sein, zusammengesetzt aus dem Namen des Knotens, einem Punkt und dem eigentlichen Namen des Attributs. Der reguläre Ausdruck wird mit der Syntax für reguläre Ausdrücke des JDK 1.4[5] formuliert. Neben den in Abschnitt 2.2 eingeführten Notationen (wie z.B. $\star$, $+$ für Iterationen, ? für optionales Autreten, usw.) werden eine Reihe von weiteren Festlegungen getroffen. Nützlich ist z.B. der Punkt, der für jedes mögliche Eingabezeichen mit Ausnahme der Zeilenumbrüche steht.

In einem Guard können beliebig viele Funktionen (Einschränkung: pro Attribut eines Knotens nur eine Funktion) mit dem logischen Operator '&&' verknüpft werden. Negation und logische Oder-Verknüpfung sind nicht ohne weiteres realisierbar, wenn der Guard als Teil eines Baummusters interpretiert werden soll. Eine logische Oder-Verknüpfung würde zu einer Menge von Baummustern (oder einer Menge von Regeln) führen (je ein Baummuster für jede Alternative). Eine Überdeckung eines Teilbaums würde dann vorliegen, falls ein Baummuster (also eine Alternative) der Menge den Teilbaum überdeckt.

```
Statement {LocalVariableDecl {Type typname} varList} ';'
  | regexMatch(typname.value, "(int|long)")
-> H [typname H hs=0 [varList ';']] ;
```

Abbildung 5.26: Beispiel für eine Regel mit Guard

Das Beispiel in Abbildung 5.26 zeigt eine Regel für eine lokale Variablendeklaration. Der Guard spezialisiert das Baummuster b_1 derart, daß es nur auf Teilbäume paßt, die für eine Variablendeklaration von Variablen des Typs 'int' oder 'long' stehen.

Interpretation als Baummuster

Abhängig davon, ob der Guard semantisch zur Auswahl des zu prüfenden Teilbaumes oder zur Konvention gehört, bestimmt der Guard das Baummuster b_1 (das außerdem vom Muster bestimmt wird) oder b_2 (das außerdem von der Konvention bestimmt wird) mit. In beiden Fällen wird dem jeweiligen Baummuster pro Funktion ein weiteres Attribut angeheftet. Konkret bedeutet dies für jede Funktion $regexMatch$ des Guards mit Argumenten $v.attrName$ und $regex$ das Anheften eines Attributes a an das Blatt des Baummusters mit Namen v, wobei $a = (attrName, regexMatch(regex))$ ist.

Die Abbildung 5.27 zeigt die Umsetzung des Guards aus Abbildung 5.26 in ein Baummuster. Der Knoten mit Namen id hat ein Attribut „value" mit

[5] Java Development Kit 1.4, für die Syntax der regulären Ausdrücke siehe JDK 1.4 API Dokumenatation unter http://java.sun.com/j2se/1.4/docs/api/ im package java.util.regex in der Dokumentation der Klasse Pattern

einer Funktion *regexMatch* als Attributwert, die ihrerseits "[a-z].*" als Parameter hat (der Attributname „value" ist wiederum nicht in der Abbildung verzeichnet).

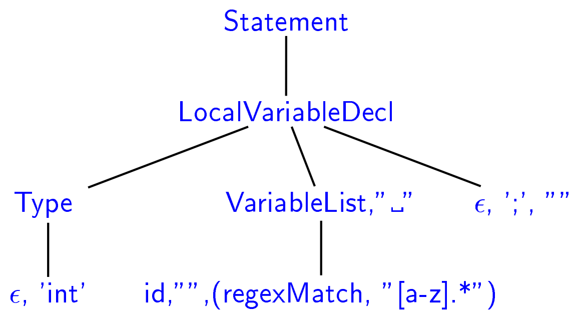

Abbildung 5.27: Baummuster umgesetzt aus Muster und Guard, Abb. 5.26

Die zweistellige boolsche Funktion *regexMatch* als Attributwert des Attributs „value" liefert *true*, falls der Attributwert des zu überdeckenden Knotens durch den regulären Ausdruck beschreibbar ist. Andernfalls hat die Funktion *false* als Rückgabewert.

5.2.2.4 GuardNotMatchedAction

Besitzt eine Regel eine GuardNotMatchedAction werden die durch die GuardNotMatchedAction spezifizierten Aktionen bei nicht erfülltem Guard ausgeführt. Ungeachtet des nicht erfüllten Guards kann bei nicht erfüllter Konvention eine Korrektur der Formatierung gemäß der Konvention vorgenommen werden.

```
GuardNotMatchedAction
= warn '(' Message ')' ;

Message
= MessagePart
| MessagePart ',' Message ;

MessagePart
= '"' {Char} '"'
| AttributName ;
```

Abbildung 5.28: Grammatik für GuardNotMatchedAction

Per Festlegung hat die GuardNotMatchedAction keinen Einfluß auf die Baummuster und kann damit keine Veränderungen am abstrakten Syntaxbaum vornehmen. Die einzige Funktionalität, die einer GuardNotMatchedAction in dieser Entwicklungsstufe zugestanden wird, ist die Ausgabe von Warnmeldungen. Diese Restriktionen werden im Abschnitt 7.3 in Form eines Ausblicks auf weitere zu beschreibende Programmierrichtlinienklassen diskutiert, und die Möglichkeit von Erweiterungen an dieser Stelle erörtet.

Der Grammatikausschnitt in Abbildung 5.28 zeigt die Grammatik in EBNF-Notation für eine GuardNotMatchedAction. Eine Warnmeldung kann dabei aus Zeichenketten und Attributwerten von Blättern der Baummuster bestehen, die durch Kommata konkateniert werden.

Insbesondere ist dabei der Zugriff auf Attribute der Knoten von Interesse, die Knoten einer Mustervariable des Baummusters b_1 überdeckt wurden. Wurde z.B. (vergleiche Abbildung 5.29) eine Namenskonvention bei einem Knoten mit Namen id nicht erfüllt, kann mit id.line und id.column die Position der Verletzung im Quelltext in die Warnmeldung aufgenommen werden.

```
Statement {LocalVariableDecl {Type 'int'} {VariableList id}} ';'
 | regexMatch(id.value, "[a-z].*")
-> H ['int' H hs=0[id ';']]
 | warn("Namenskonvention verletzt in Zeile ", id.line);
```

Abbildung 5.29: Spezifikation zur Prüfung einer Namenskonvention

Kapitel 6

Implementierung

Im Rahmen einer Fallstudie für Java entstand eine Implementierung der vorgestellten Konzepte in der Programmiersprache Java für Programme der Programmiersprache Java. Dabei ist zu bemerken, daß lediglich das Front-End der Anwendung programmiersprachenabhängig ist (also in diesem Fall speziell für Java entwickelt wurde), während das Back-End nicht von der Programmiersprache abhängt. Um eine Implementierung für weitere Programmiersprachen zu realisieren, müßte also lediglich ein neues Front-End erstellt werden. Die Schnittstelle zwischen dem Front- und Back-End ist die Programmrepräsentation, wie in Abschnitt 4.2 beschrieben. Ausgabe des Programmes sind Warnmeldungen über verletzte Programmierrichtlinien und ein gemäß der Formatierungsrichtlinien transformierter Programmquelltext.

6.1 Übersicht

Abbildung 4.1 in Abschnitt 4.1 zeigt einen Ablaufplan der Anwendung mit Ein- und Ausgaben des Programmes. In diesem Abschnitt soll ein Überblick über die, mit den in dem Ablaufplan formulierten Aufgaben, korrespondierenden Programmteilen (im Sinne der wichtigsten Klasse für die jeweilige Teilaufgabe) gegeben werden. Abbildung 6.1 zeigt in einem Klassendiagramm die wichtigsten Klassen und formuliert die jeweiligen Zuständigkeiten.

Die Klasse CodeFormatter ist die Hauptklasse der Anwendung. Eine Instanz der Klasse CodeFormatter hält Instanzen der fünf anderen Klassen aus Abbildung 6.1 zur Erledigung der jeweiligen Aufgaben. Die folgende Aufstellung erläutert die Aufgaben der jeweiligen Klasse.

XMLTreeGenerator Die Aufgabe besteht im Parsen des Java-Quellcodes und Schreiben einer XML-Datei, die den abstrakten Syntaxbaum beinhaltet.

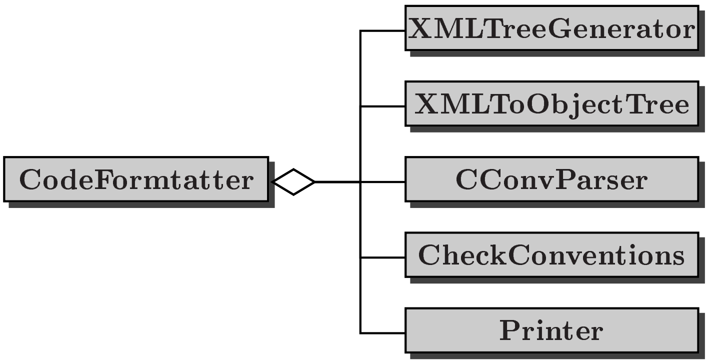

Abbildung 6.1: Übersicht über die wichtigsten Klassen

XMLToObjectTree Die Aufgabe besteht in der Umwandlung des vom XMLTreeGenerator erzeugten abstrakten Syntaxbaumes in XML-Darstellung in einen abstrakten Synatxbaum bestehend aus verknüpften Objekten.

CConvParser Die Aufgabe besteht im Parsen der Spezifikation der Programmierrichtlinien, die mittels CCML verfaßt wurden. Die geparste Spezifikation wird in einer Datenstruktur vom Typ Rules abgelegt.

CheckConventions Die Aufgabe besteht in der Prüfung des von XMLToObjectTree gelieferten abstrakten Syntaxbaumes auf Einhaltung der Programmierrichtlinien aus dem vom CConvParser gelieferten Objektes vom Typ Rules. Gegebenenfalls werden Tranformationen im abstrakten Syntaxbaum gemäß der Programmierrichtlinien vorgenommen.

Printer Die Aufgabe besteht in der Umwandlung des transformierten abstrakten Syntaxbaumes in einen Programmquelltext.

Ordnet man die Klassen nach Zugehörigkeit zum Front- bzw. Back-End, gehört lediglich die Klasse XMLTreeGenerator zum Front-End, während alle anderen Klassen zum Back-End gehören.

6.2 Front-End

6.2.1 Parser

Zur Erstellung des Parsers wurde der Parser-Generator (vgl. Abschnitt 2.5.2) SableCC[1] verwendet. Einer der Gründe, die für die Verwendung von SableCC

[1] Das Parser-Generator SableCC ist aus der Master's Thesis [Gag98] hervorgegangen (siehe auch http://www.sablecc.org).

sprachen, ist, daß SableCC Java-Quellcode erzeugt und somit den Anforderungen der Fallstudie, in Java zu programmieren, genügte.

Eingabe des Parser-Generators ist eine Grammatik der Programmiersprache. Im allgemeinen werden während der lexikalischen Analyse eines Programmes Whitespace und Kommentare ignoriert. Ein Ziel der zu entwickelnden Anwendung ist jedoch die Prüfung und gegebenenfalls Transformation des Leerraums. Des weiteren sollen evtl. umformatierte Programmquelltexte die Kommentare des ursprünglichen Programmquelltextes enthalten. Abschnitt 4.2.2 beschreibt die hier verfolgten Konzepte zur Speicherung des Leerraums und der Kommentare. Auswirkungen hat das auch auf die Spezifikation, mit der der Parser-Generator den Parser erzeugt. SableCC erzeugt den Parser aus einer Grammatik, der anders als bei anderen Parser-Generatoren keine semantischen Aktionen oder andere Attribute angehängt werden können. Das bedeutet, daß Leerraum und Kommentare in die Grammatik eingearbeitet werden müssen. Eine Methode dazu ist, für jedes Terminal t eine Produktion einzuführen $\mathsf{T} = \mathsf{t}\ \{\mathsf{Whitespace} \mid \mathsf{Comment}\ \}$; und alle Vorkommen von t in den ursprünglichen Produktionen der Grammatik durch T zu ersetzen. Damit werden aller Leerraum und Kommentare mit Ausnahme des Leerraums und der Kommentare vor dem ersten Terminal eines Programmquelltextes beschrieben. Dieser Leerraum ist durch Ergänzungen in der Startproduktion beschreibbar.

SableCC erstellt für jedes Nichtterminal- und für jedes Terminalsymbol (Leerraum und Kommentare werden jetzt auch mit Nichtterminal- und Terminalsymbolen beschrieben) eine Klasse. Parst man einen Programmquelltext mit dem von SableCC generierten Parser, erstellt der Parser einen konkreten Syntaxbaum in Form von verknüpften Objekten. Durch das Überschreiben von Analyse-Klassen ist es möglich, ereignisgesteuert auf bestimmte Terminale und Nichtterminale im Syntaxbaum zu reagieren. Insbesondere ist es möglich, durch ausgewählte Behandlung von ausgewählten Knoten des konkreten Syntaxbaumes einen abstrakten Syntaxbaum zu erzeugen, der den Anforderungen aus Abschnitt 4.2.2 genügt. Im folgenden Abschnitt wird die Syntax des abstrakten Syntaxbaumes beschrieben, wie er von einer Instanz der Klasse XMLTreeGenerator erzeugt wird.

6.2.2 Programmrepräsentation

Die Klasse XMLTreeGenerator überschreibt die Methoden, der von SableCC generierten Analyse Klasse DepthFirstAdapter (siehe Abbildung 6.2) des Pakets de.ppi.fis.codeconv.javaparser[2]. Eine Instanz von XMLTreeGenerator erzeugt eine Programmrepräsentation in XML-Darstellung gemäß Abschnitt 4.2. Dabei steht ein XML-Element, das lediglich ein Attribut „whitespace" aber mindestens ein weiteres Element als Inhalt hat für ein Nichtterminal

[2]Die Angabe des Pakets bei dieser Klasse dient der Unterscheidung von der Klasse DepthFirstAdapter des Pakets de.ppi.fis.codeconv.cconvparser (siehe Abschnitt 6.3.2)

der abstrakten Syntax. Ein Element ohne Inhalt mit den Attributen „white-space", „value", „line" und „column" steht für ein Terminal der abstrakten Syntax. Ein Beispiel für einen Ausschnitt aus einem abstrakten Syntaxbaum in dieser XML-Notation wurde bereits in Abbildung 4.6 in Abschnitt 4.2.2.1 gezeigt.

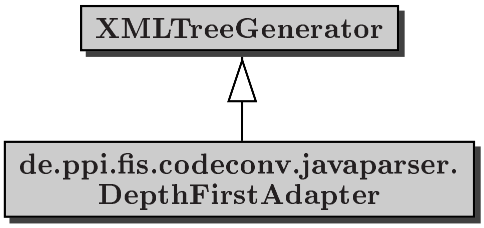

Abbildung 6.2: Vererbung beim XMLTreeGenerator

Die Darstellung des abstrakten Syntaxbaumes in XML (und damit verbunden das Schreiben dieses XML-Baumes in eine Datei) ist in einer fertigen Implementierung nicht mehr notwendig. Die später beschriebene Klasse XMLToObjectTree des Back-Ends wandelt den XML-Baum in einen äquivalenten Baum, der durch verknüpfte Objekte repräsentiert wird. Die Prüfung auf Einhaltung von Programmierrichtlinien wird dann anhand des gewandelten Baumes vorgenommen. Anstatt einen AST in XML-Darstellung zu erzeugen, könnte aus der verknüpften Objektdarstellung von SableCC direkt die Darstellung des ASTs erzeugt werden, die zur Prüfung auf Einhaltung der Programmierrichtlinien benötigt wird. Die zwischenzeitliche Darstellung in XML hat aber auch Vorteile. Zum einen ist die Schnittstelle über das zuvor beschriebene Aussehen von Terminalen und Nichtterminalen genau definiert. Zum anderen haben die Grammatiken von realen Programmiersprachen einen Umfang, der schwer zu überblicken ist. Um sicherzustellen, daß der Parser tatsächlich die abstrakte Syntax erzeugt, die der Entwickler des Parsers erzeugen will, kann die XML-Zwischendarstellung genutzt werden. Dazu formuliert der Entwickler eine DTD[3], die die abstrakte Syntax der Programmiersprache definiert und prüft die XML- Zwischendarstellung der geparsten Programme mit einem validierenden XML-Parser.

6.3 Back-End

6.3.1 Einlesen des XML Baumes

Das Einlesen des abstrakten Syntaxbaumes basiert auf der Simple API für XML (SAX), die von den Mitgliedern der Mailingliste XML-DEV entwickelt

[3]Document Type Definition (siehe [Wor00])

wurde. Diese Schnittstelle beschreibt die serielle Verarbeitung einer XML-Datei. Die Kommunikation wird über sogenannte Handler abgewickelt. Eine Implementierung der Handler-Klassen findet man im JDK 1.4 im Paket org.xml.sax.helpers. Die Handler-Klasse DefaultHandler stellt Methoden bereit, die auf Ereignisse beim Parsen eines XML-Dokumentes reagieren. Wichtige Ereignisse sind der Beginn (Anfangstag) und das Ende (Endtag) eines Elements. Durch Reagieren auf diese Ereignisse kann aus dem abstrakten Syntaxbaum in XML-Darstellung ein Baum, bestehend aus verknüften Objekten, generiert werden.

Das Klassendiagramm in Abbildung 6.3 zeigt eine Klassenhierarchie der Klassen, dessen Instanzen den abstrakten Syntaxbaum repräsentieren. Die Instanz der Klasse XMLToObjectTree liefert eine Referenz auf die Wurzel des abstrakten Syntaxbaumes. Die Wurzel des ASTs ist eine Instanz vom Typ ASTElement. Ebenso sind alle anderen Knoten des ASTs Instanzen vom Typ ASTElement in Form einer Instanz der Unterklasse Nonterminal (falls innerer Knoten) oder der Klasse Terminal (falls Blatt des ASTs).

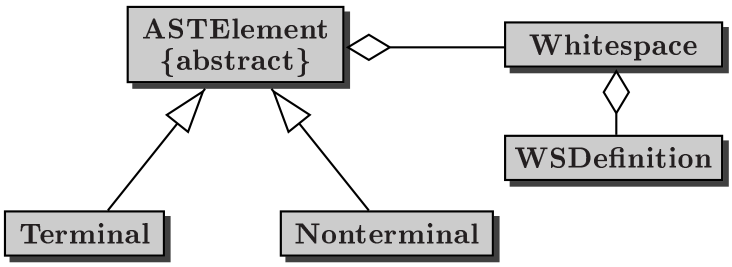

Abbildung 6.3: Klassenhierarchie für den AST

Alle Knoten des ASTs haben ein Attribut „whitespace". Eine Instanz vom Typ ASTElement hat eine Referenz auf eine Instanz der Klasse Whitespace, das den Leerraum und die Kommentare vor dem jeweiligen Terminal oder Nichtterminal repräsentiert. Die Klasse Whitespace bietet Methoden, die den formatierungsrelevanten Whitespace aus der Folge von Whitespace und Kommentaren extrahieren und als Folge von horizontalen und vertikalen Trennzeichen zurückgeben. Um die Abbildung von der Originalzeichenfolge des Leerraums und der Kommentare auf eine Folge von horizontalen und vertikalen Whitespaces realisieren zu können, hat jede Instanz der Klasse Whitespace eine Referenz auf ein Objekt der Klasse WSDefinition. Dieses Objekt wird beim Lesen des globalen Teils der Programmrepräsentation instanziiert und hält die Definitionen der verschiedenen Whitespace Typen. Beim Parsen der CCML-Spezifikation wird das Objekt vom Typ WSDefinition durch die Rollenzuweisung für die verschiedenen Kommentarstile ergänzt. Der formatierungsrelevante Whitespace wird nach dem im Abschnitt 4.5.1.2 vorgestellten Ablauf extrahiert.

6.3.2 CCML Parser

Die Klasse CConvParser hat die Aufgabe, eine Spezifikation, die mittels CCML verfaßt ist, in eine interne Darstellung umzuwandeln. Um den Parser für CCML zu generieren, wurde wieder der Parser-Generator SableCC verwendet. Wie die Klasse XMLTreeGenerator überschreibt die Klasse CConvParser Methoden der Analyse-Klasse DepthFirstAdapter (in diesem Fall des Pakets de.ppi.fis.codeconv.cconvparser), die von SableCC generiert wurde.

Das Klassendiagramm in Abbildung 6.4 zeigt einen Überblick über die interne Darstellung der Programmierrichtlinien. Alle Regeln werden zentral in einer Instanz der Klasse Rules in Form einer Hashtabelle verwaltet. Ein Objekt vom Typ Rule steht für eine Regel und hat in der Form von Instanzenvariablen Referenzen auf die vier Regelteile Muster (Pattern), Konvention (Convention), Guard (Guard) und GuardNotMatchedAction (GuardAction).

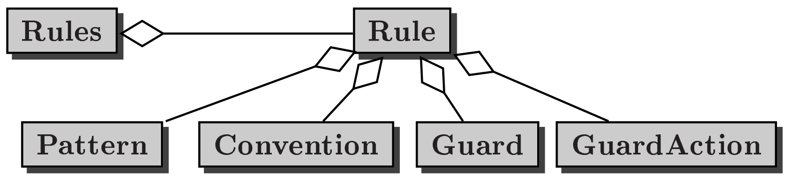

Abbildung 6.4: Klassen zur Darstellung von Regeln nach Abschnitt 5.2.2

Des weiteren besitzt die Klasse Rule Methoden, die Prüfungen erlaubt, die

- die Überdeckung eines Teilbaumes mit dem Muster prüfen.
- die Überdeckung, des durch das Muster überdeckten Teilbaumes, mit der Konvention prüfen.
- prüfen, ob der Guard, in dem vom Muster überdeckten Teilbaum, erfüllt wird.

Außerdem kann die Warnmeldung einer GuardNotMatchedAction zu dem vom Muster überdeckten Teilbaum generiert werden. Durch Kombination der Methoden, die die Prüfungen realisieren, ist das in Abschnitt 5.2.2 spezifizierte Verhalten der Prüfung auf Einhaltung der Programmierrichtlinien bei den verschiedenen Regeldefinitionen realisierbar.

Während die Implementierung von Guard und GuardAction nicht näher beschrieben werden soll, wird in den folgenden beiden Unterabschnitten näher auf die Darstellung von Muster und Konventionen eingegangen.

6.3.2.1 Darstellung des Musters

Die Abbildung 6.5 zeigt die Klassen, die zur Darstellung der Muster dienen. Ein Muster ist eine Instanz der Klasse Pattern, die eine Liste von Referenzen auf Instanzen vom Typ PatternElement besitzt. Die Klasse PatternElement ist abstrakte Oberklasse der Klassen PatternTerminal, PatternVariable und auch Pattern. Also kann eine Instanz von Pattern wiederum Referenzen auf Instanzen von Pattern haben. Damit wird der Modellierung von tieferen Mustern Rechnung getragen.

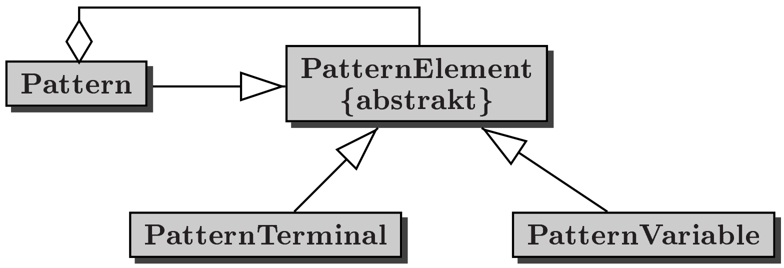

Abbildung 6.5: Klassen zur Darstellung eines Musters

6.3.2.2 Darstellung der Konvention

Wie in der Klassenhierarchie in Abschnitt 6.3.2.1, gibt es in der Klassenhierarchie zur Darstellung einer Konvention ebenfalls eine abstrakte Oberklasse für alle anderen Klassen.

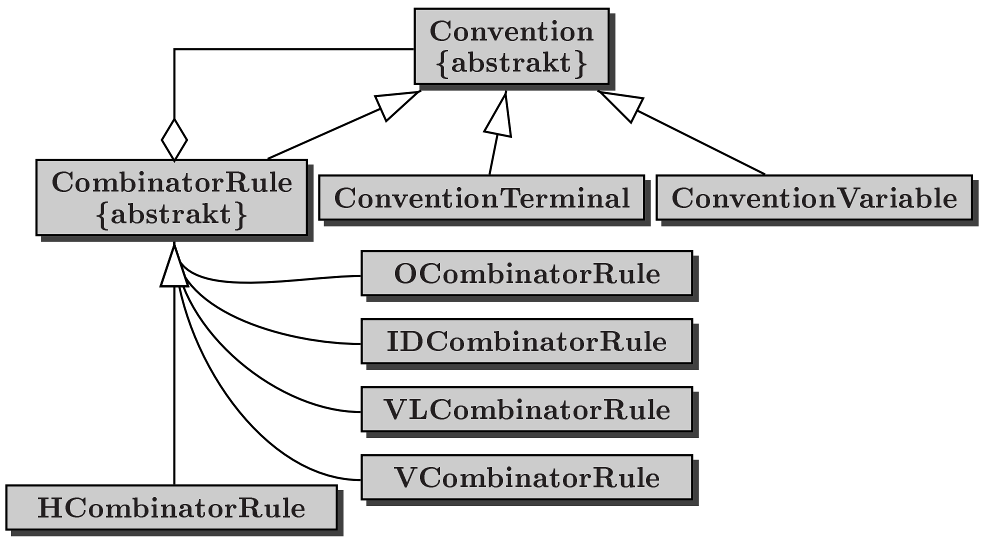

Abbildung 6.6: Klassen zur Darstellung eines Musters

Eine Konvention ist eine Instanz vom Typ Convention (der abstrakten Oberklasse). Damit kann eine Konvention eine Instanz der nicht abstrakten Subklassen ConventionTerminal, ConventionVariable oder der speziellen Kombinator Klassen sein. Die Aggregation zwischen der Klasse CombinatorRule und der Klasse Convention modelliert, daß im Wirkungsbereich eines Kombinators wieder beliebige Elemente vom Typ Convention liegen können.

6.3.3 Prüfung und Transformation

Ob bei der Verletzung von Programmierrichtlinien nur Warnmeldungen ausgegeben werden, oder ob Transformationen vorgenommen werden, hängt von der Art der Programmierrichtlinie ab, die geprüft wird. Bei der Verletzung von Namenskonventionen werden nur Warnmeldungen ausgegeben, die allerdings auch in Form einer GuardNotMatchedAction explizit angegeben werden müssen. Transformationen werden bei der Verletzung von Formatierungsrichtlinien durchgeführt, indem die ,,whitespace"-Attribute im AST überschrieben werden. Problematisch ist die Prüfung von alternativen Formatierungen bei einer Überschreitung der maximal zulässigen Zeilenbreite.

Im folgenden wird zunächst die generelle Strategie, die für die Prüfung des abstrakten Syntaxbaumes verwendet wird näher erklärt. Im Anschluß wird auf die spezielle Problematik der Prüfung von alternativen Layouts eingegangen.

6.3.3.1 Generelle Strategie

Die Instanz der Klasse CheckConventions bekommt Referenzen auf die Wurzel des abstrakten Syntaxbaumes und die Instanz der Klasse Rules übergeben. Bei der Prüfung des AST auf Einhaltung von Programmierrichtlinien wird nun nach folgendem Algorithmus vorgegangen (bezeichne cc die Instanz der Klasse Rules):

(i) Suche in cc für den aktuell zu prüfenden Teilbaum $t = (r, T)$ mit $T = ((r_1, T_1), \ldots, (r_n, T_n)), n \geq 0$ (initial ist t der gesamte abstrakte Syntaxbaum) eine Regel R mit einem auf t noch nicht geprüftem Muster p mit Wurzel r', so daß r' r überdeckt.

   (a) Es gibt ein solches Muster p in cc.
   Überdeckt das Muster p einen Teilbaum t' von t mit Wurzel r dann fahre fort mit (ii) sonst gehe zurück zu (i).

   (b) Es gibt kein solches Muster in cc.
   Dann generiere eine sogenannte Default-Regel der Form
   $r\ r_1 \ldots r_n$ –> ID $[r_1 \ldots r_n]$, wobei jedes r_i, $1 \leq i \leq n$ einen direkten Nachfolger von r überdeckt und ein Teilbaum t' von t mit Wurzel r überdeckt wird.
   Fahre fort mit (ii).

(ii) Enthält R einen Guard ?

(a) R enthält keinen Guard.
Dann prüfe auf Einhaltung der Konvention gebe ggf. Warnmeldungen aus und führe Transformationen durch. Gehe zu (iii).

(b) R enthält einen Guard.
Enthält R eine GuardNotMatchedAction ?

1. R enthält keine GuardNotMatchedAction.
Erfüllt der Teilbaum t' die Anforderungen des Guard dann prüfe auf Einhaltung der Konvention und gebe ggf. Warnmeldungen aus und führe Transformationen durch gehe zu (iii), sonst gehe zurück zu (i)
2. R enthält eine GuardNotMatchedAction.
Prüfe die Konvention und gebe ggf. Warnmeldungen aus und führe Transformationen durch. Prüfe den Guard und führe bei nicht erfülltem Guard die GuardNotMatchedAction aus. Gehe zu (iii).

(iii) Für jede Variable im Muster p, die einen inneren Knoten k des ASTs überdeckt, gehe zu (i), um den Teilbaum des ASTs mit Wurzel k zu prüfen.

6.3.3.2 Reaktion auf Verletzung von Programmierrichtlinien

Wie bereits in Abschnitt 5.2.2.2 angekündigt, wird im folgenden ein Verfahren vorgestellt, das zur Prüfung von alternativen Layouts dient. Die Implementierung geht nach diesem Verfahren vor, das jedoch nicht alle Fälle zufriedenstellend löst, wie im Anschluß diskutiert wird.

1. Wird die maximale Zeilenbreite durch das aktuell geprüfte Element verletzt ?

   (i) Die Zeilenbreite wird verletzt.
   Liegt das Element im Wirkungsbereich eines O-Kombinators ?

   (a) Das Element liegt im Wirkungsbereich eines O-Kombinators.
   Gehe zurück zum äußersten umgebenen O-Kombinator, bei dem noch nicht alle Alternativen durchlaufen wurden, wähle die nächste Alternative und gehe zurück zu 1. Existiert keine noch nicht geprüfte Alternative, gib eine Warnmeldung aus und fahre fort mit 2.

   (b) Das Element liegt nicht im Wirkungsbereich eines O-Kombinators.
   Gib eine Warnmeldung aus und fahre fort mit 2.

   (ii) Die Zeilenbreite wird nicht verletzt.
   Fahre fort mit 2.

2. Prüfe das im Sinne des Baumdurchlaufs nächste Element im AST.

Es werden also so lange alternative Layouts geprüft, bis entweder die maximale Zeilenbreite nicht mehr überschritten wird, oder keine weiteren alternativen Layouts existieren. Zur Motivation der Rücksprungs zum äußersten umgebenen O-Kombinator betrachte man das optionale Umbrechen von binären Ausdrücken. In einer Folge von mehreren verschachtelten binären Ausdrücken, wird durch obige Festlegung zunächst bei den binären Ausdrücken umgebrochen, bei denen der Operator die geringste Operatorpriorität besitzt, da sich die Prioritäten in der Struktur des AST wiederfinden.

```
Expression a binop b
-> O [H [a binop b]
     V [H [a binop] b] ;
```

Abbildung 6.7: Optionale Layouts für binäre Ausdrücke

Das Beispiel in Abbildung 6.8 zeigt das Umbrechen nach obiger Strategie und der Regel für einen binären Ausdruck nach Abbildung 6.7 unter der Annahme, daß die maximale Zeilenbreite durch das Terminal c überschritten wird. Durch die realisierten Umbrüche, an sinnvollen Stellen gesetzt wurden, wird die Zeilenbreite wieder eingehalten.

```
(i)   ergebnis = a +
                 b * c;

(ii)  ergebnis = a * b +
                 c;

(iii) ergebnis = a *
                 b * c;
```

Abbildung 6.8: Formatierung bei Überschreitung der Zeilenbreite durch c

Unzufriedenstellend ist jedoch das Verhalten, wenn die zulässige Zeilenbreite erst mit dem Semikolon überschritten wird. Dann existiert kein umgebener O-Kombinator, und obwohl durch Umbrechen bei den vorigen binären Ausdrücken ein Layout erzielt werden könnte, das die maximale Zeilenbreite nicht verletzt, wird nach dem vorgestellten Verfahren nur eine Warnmeldung ausgegeben.

6.3.4 Printer

Die Klasse Printer hat die Umwandlung des (transformierten) abstrakten Syntaxbaumes in einen Programmquelltext zur Aufgabe. Dazu wird einer Instanz der Klasse Printer die Referenz auf die Wurzel des abstrakten Syntaxbaumes übergeben.

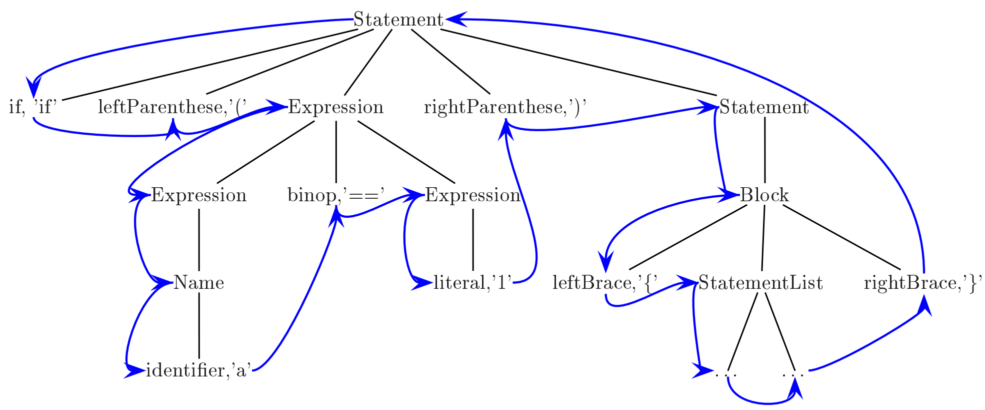

Abbildung 6.9: Beispiel für einen Printerdurchlauf

Die Instanz der Klasse Printer durchläuft den abstrakten Syntaxbaum in Präordnung. D.h., daß zuerst die Wurzel und im Anschluß die Nachfolger von links nach rechts in Präordnung durchlaufen werden. Für jeden Knoten wird der Wert des Attributes „whitespace" und für alle Blätter (Terminale) darauffolgend der Wert des Attributes „value" ausgegeben (bzw. in eine Datei geschrieben). Abbildung 6.9 zeigt das Durchlaufen des abstrakten Syntaxbaumes beginnend mit der Wurzel. Aus Gründen der Übersichtlichkeit wurde auf die Angabe der Belegung der „whitespace" Attribute verzichtet.

6.4 Demonstration der Anwendung

In diesem Abschnitt soll in einem kleinen Beispiel die Funktionsweise der Anwendung demonstriert werden. Ein sehr schlecht formatierter Programmquelltext und eine kleine CCML-Spezifkation werden als Eingabe der Anwendung dienen. Die Steuerung der Anwendung ist über eine graphische Oberfläche möglich. Neben den Eingabedateien, die den zu formatierenden Programmquelltext und die CCML-Spezifikation beinhalten, ist ein Ausgabeverzeichnis anzugeben, in dem der umformatierte Programmquelltext mit dem Dateinamen der Eingabedatei gespeichert wird. Ist das Ausgabeverzeichnis identisch zu dem Eingabeverzeichnis, wird die ursprüngliche Datei ersetzt. Abbildung 6.10 zeigt einen Screenshot der graphischen Oberfläche der Anwendung, mit der die Eingabedateien und das Ausgabeverzeichnis spezifiziert werden können.

Abbildung 6.10: Steuerung der Anwendung

Als Eingabe dienen in Abbildung 6.10 der Programmquelltext, der in der Datei „beispiel.java" gespeichert ist, und die CCML-Spezifikation aus der Datei „test.ccml". Zudem wurde ein Ausgabeverzeichnis spezifiziert, das sich vom Eingabeverzeichnis unterscheidet. Durch den Button „Prüfung ..." kann die Prüfung auf Einhaltung der Programmierrichtlinien, die in „test.ccml" spezifiziert sind, für den Programmquelltext aus „beispiel.java" gestartet werden.

```
/*
 * Beispielsklasse zur Demonstration der Anwendung
 */

public class beispiel {
public beispiel(int aFirstValue,
int aSecondValue, int aThirdValue) {
if (aFirstValue > 0) {
for (int p = 0;p < 2;p++)
System.out.println(aFirstValue * aSecondValue + aThirdValue);
}
} // Konstruktorende
}
```

Abbildung 6.11: Klasse beispiel der Datei beispiel.java

In Abbildung 6.11 wird zunächst der Programmquelltext gezeigt, der sich

```
Globals
  linewidth = 70

Rules
  ClassDecl mod 'class' id {ClassBody '{' cel '}' }
    | regexMatch(id.value, "[A-Z].*")
  -> V vs=1 is=0 [V vs=1 is=4 [H [mod 'class' id '{'] cel] '}']
    | warn("Namenskonvention verletzt in Zeile ", id.line,
           ": Klassennamen beginnen mit Grobuchstaben");

  ClassElement mods id '(' {ParameterList p ',' pList} ')' cbody
  -> H [mods H hs=0 [id '(' V [H hs=0 [p ','] pList] ')'] cbody];

  ConstructorBody '{' sList '}'
  -> VL [VL vs=0 is=4 ['{' sList] '}'] ;

  StatementList stat statList
  -> V [stat statList] ;

  Statement 'if' '(' exp ')' {Statement {Block '{' statList '}'}}
  -> V [V vs=0 is=4 [H ['if' H hs=0 ['(' exp ')'] '{'] statList] '}'];

  Statement 'if' '(' exp ')' stat
  -> V vs=0 is=4 [H hs=1 ['if' H hs=0 ['(' exp ')']] stat];

  Statement 'for' '(' forInit ';' exp ';' expList')' stat
  -> V vs=0 is=4 [H ['for' H hs=0 ['(' forInit ';' exp ';' expList ')']] stat] ;

  ForInit {LocalVariableDecl typ {VariableList id '=' init}}
    | regexMatch(id.value, "[i|j|k|l|m|n]")
  -> H [typ id '=' init]
    | warn("Bevorzugte Schleifenvariablen i,j,k,l,m oder n verwenden in Zeile ", id.line);

  Expression name '(' aList ')'
  -> H hs=0 [name '(' aList ')'] ;

  Expression a binop b
  -> O [H [a binop b], V [H [a binop] b]];

  ParameterList p ',' pList
  -> V [H hs=0 [p ','] pList] ;
```

Abbildung 6.12: CCML-Spezifikation der Datei „test.ccml" hinter der Datei „beispiel.java" verbirgt, während Abbildung 6.12 die CCML-Spezifikation der Datei „test.ccml" beinhaltet. Neben der sehr schlechten Formatierung des Quelltextes in Abbildung 6.11 werden auch gängige Namenskonventionen verletzt. Eine geläufige Namenskonvention in Java besagt, daß Klassennamen grundsätzlich mit einem Großbuchstaben beginnen müssen. Eine weitere Namenskonvention fordert, für Schleifenvariablen vorrangig die Variablennamen i,j,k,l,m und n zu verwenden. Diese Namenskonventionen finden sich in der CCML-Spezifikation in Form von Guard und GuardNotMatchedAction in den Regeln für die Elemente ClassElement und ForInit wieder. Im globalen Teil der Spezifikation wird lediglich eine maximale Zeilenbreite definiert. Da im Eingabequelltext keine anderen horizontalen und vertikalen Trennzeichen als Leerzeichen und Zeilenumbrüche verwendet wurden, ist die Angabe von Übersetzungsverhältnissen nicht notwenig. Den verschiedenen Kommentarstilen werden die Standardrollen zugewiesen (Blockkomentare sind vorangestellte und Zeilenendekommentare nachgestellte Erläuterungen). Ohne auf jede einzelne Regel einzugehen, soll zum einen bemerkt werden, daß für „if-then"-Statements zwei verschiedene Muster angegeben wurden, wobei das erste den spezielleren Fall eines nachfolgenden Blocks betrachtet. Zum anderen soll auf die Regel für binäre Ausdrücke hingewiesen werden, die einen O-Kombinator beinhaltet, nach dem bei Überschreitung der zulässigen Zeilenbreite anstelle einer horizontalen eine vertikale Komposition umgesetzt werden soll.

```
/*
 * Beispielsklasse zur Demonstration der Anwendung
 */

public class beispiel {

     public beispiel(int aFirstValue,
                     int aSecondValue,
                     int aThirdValue) {
          if (aFirstValue > 0) {
               for (int p = 0;p < 2;p++)
                    System.out.println(aFirstValue * aSecondValue +
                                       aThirdValue);
          }
     } // Konstruktorende

}
```

Abbildung 6.13: Umformatierter Programmquelltext

In Abbildung 6.13 ist der umformatierte Quelltext, der mit der Prüfung auf Einhaltung der Programmierrichtlinien entstanden ist, zu sehen. In dem Statusfenster der Anwendung, das Abbildung 6.14 zeigt, ist der Ablauf der

Prüfung nachvollziehbar. Neben einer Reihe von entdeckten Verletzungen der Formatierungsrichtlinien, werden zwei verletzte Namenskonventionen dokumentiert. Zum einen hat die Anwendung eine Verletzung bei der Benung der Klasse festgestellt, da der Klassenname mit einem Kleinbuchstaben beginnt. Zum anderen wurde in der „for"-Schleife die Laufvariable mit p benannt, was keiner der bevorzugten Bezeichner für Laufvariablen ist.

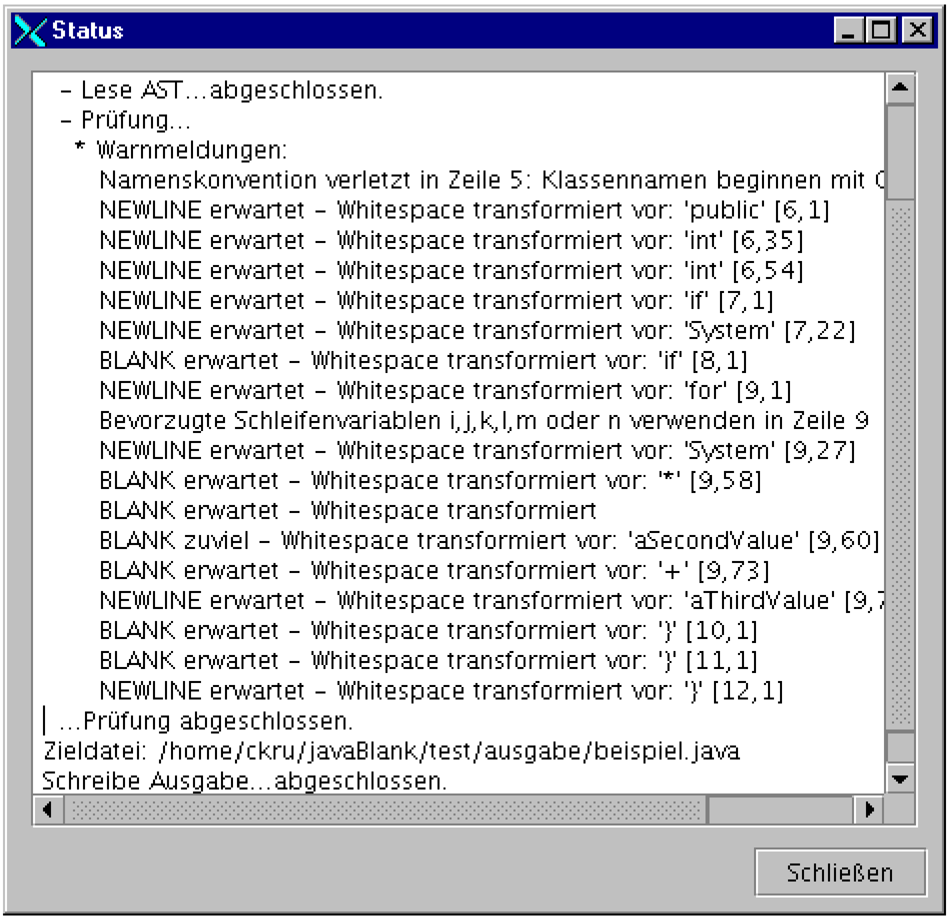

Abbildung 6.14: Statusfenster der Anwendung

Während die Fehlermeldungen bei verletzten Namenskonventionen benutzerdefiniert sind (spezifiziert in einer GuardNotMatchedAction), werden für die Verletzung von Formatierungsregeln Standardfehlermeldungen generiert. Dabei ist BLANK ein Synonym für ein (bezüglich der Ordnung kleinstes) horizontales Trennzeichen und NEWLINE ein Synonym für ein (bezüglich der Ordnung kleinstes) vertikales Trennzeichen. Die Positionsangabe bei der Meldung der Verletzung bezieht sich auf die Position im Quelltext der Eingabe.

Kapitel 7

Analyse und Ausblick

Um eine automatisierte Prüfung auf Einhaltung von Programmierrichtlinien zu ermöglichen, wurde in dieser Arbeit der Ansatz verfolgt, eine programmiersprachenungebundene Beschreibungssprache für Programmierrichtlinien zu entwickeln, die es erlaubt, Programmierrichtlinien anhand der abstrakten Syntax der Programmiersprache zu spezifizieren. Eine Gegenüberstellung zu Ansätzen, die programmiersprachengebunden agieren, und ein Vergleich mit Konzepten, die Teilbereiche der Problemstellung abdecken, folgt in Abschnitt 7.1. Anschließend werden in Kapitel 7.2 die aktuelle Ausdrucksfähigkeit, der in dieser Arbeit entwickelten Sprache CCML diskutiert, um dann im Abschnitt 7.3 mögliche Erweiterungen zu erörtern.

7.1 Neue Ansätze dieser Arbeit

In dieser Arbeit wurde der Ansatz verfolgt, unabhängig von der zugrundeliegenden Programmiersprache eine Sprache zu entwickeln, mit der es möglich ist, Programmierrichtlinien zu spezifizieren. Die bestehenden Lösungsansätze beruhen entweder auf einem programmiersprachenabhängigen Konzept, oder sie lösen nur einen Teilbereich (insbesondere das Pretty Printing) der Problemstellung.

7.1.1 Vergleich mit programmiersprachengebundenen Konzepten

Ein programmiersprachengebundenes Konzept liegt z.B. JIndent (siehe Abschnitt 3.4.1) zugrunde. JIndent ermöglicht die Spezifikation einer Vielzahl von Programmierrichtlinien für Java, durch das Setzen von ca. 160 Parametern. Vorteil von CCML gegenüber diesem Ansatz ist die leichtere Erlernbarkeit von CCML aufgrund eines kleineren Satzes von Sprachelementen. Dieser Satz der Sprachelemente ändert sich auch nicht bei der Formulierung von Programmierrichtlinien für andere Programmiersprachen. Im Gegensatz dazu müßte beim Übertragen des Konzeptes von JIndent auf andere Programmiersprachen ein neuer Satz von Parametern entwickelt werden, der auf die Syntax der Programmiersprache zugeschnitten ist. Zudem bedeutet eine Übertragung der Konzepte in der Realisierung eine komplett neue Implementierung der Anwendung, während für die in dieser Arbeit entwickelten Anwendung nur ein neues Front-End erstellt werden muß.

Ein Vorteil programmiersprachengebundener Systeme ist die leichtere Einarbeitung von Beschreibungsmöglichkeiten für exotische, programmiersprachenspezifische Programmierrichtlinien. Sind solche Richtlinien nicht mit CCML beschreibbar, ist eine Erweiterung des Sprachumfanges unter Umständen schwieriger als beim programmsprachengebundenen Ansatz.

7.1.2 Erweiterung der programmiersprachenungebundenen Ansätze

Sowohl durch Haskell's Pretty Printer Combinators (siehe Abschnitt 5.1.2) als auch mit PPML (siehe Abschnitt 5.1.3) werden Ansätze verfolgt, mit denen ein Pretty Printer durch Verknüpfen von Elementen der abstrakten Syntax durch Kombinatoren definiert werden kann. Die Sprache CCML macht sich diese Konzepte zu eigen und basiert auf PPML (siehe dazu auch Abschnitt 5.1.4). Als Erweiterung integriert CCML Konzepte zur Spezifikation von Programmierrichtlinien anderer Klassen als Formatierungsrichtlinien. Insbesondere können über Guards Namenskonventionen spezifiziert oder Muster spezialisiert werden.

Des weiteren ist es mit den bestehenden Systemen nicht möglich, nur partiell Änderungen an der Formatierung vorzunehmen, da die ursprünglichen Formatierungen beim Parsen des Quelltextes verworfen werden. In der vorgestellten Programmrepräsentation (siehe Abschnitt 4.2) werden die Formatierungen im abstrakten Syntaxbaum behalten. Daraus resultierend, können die Formatierungen des Eingabequelltextes wiederhergestellt werden, und eine partielle Umformatierung wird möglich, indem nur für ausgewählte Elemente Richtlinien spezifiziert werden.

7.2 Ausdrückbare Richtlinien

Im vorgestellten Umfang erlaubt CCML die Spezifikation von Formatierungsrichtlinien und Namenskonventionen. Die Spezifikation von Programmierrichtlinien anderer Klassen ist zur Zeit noch nicht möglich. Auf Ideen zur Einarbeitung von Beschreibungsmöglichkeiten für Programmierrichtlinien weiterer Klassen in die bestehenden Konzepte wird im folgenden Abschnitt 7.3 eingegangen.

Die Spezifikation von Formatierungsrichtlinien in CCML orientiert sich an der Definition von Pretty Printern mit PPML. Dazu werden Elemente der abstrakten Syntax über Kombinatoren, die die Formatierung spezifizieren, miteinander verknüpft (siehe Abschnitt 5.2.2).

Namenskonventionen sind nicht im Sinne von Kombinatoren prüfbar, die Elemente untereinander kombinieren. Da eine Namenskonvention semantisch als eine Anforderung an ein einzelnes Element der abstrakten Syntax zu verstehen ist, wurde zur Spezifikation von Namenskonventionen das Konzept der Guards eingeführt. Ein Guard ist als Attribut eines Musters oder einer Konvention (abhängig davon ob eine GuardNotMatchedAction spezifiziert wurde, siehe Abschnitt 5.2.2) zu verstehen.

7.3 Ausblick

Neben dem Ausblick auf die mögliche Einarbeitung von Konzepten und Formalismen in CCML, die die Spezifikation von Programmierrichtlinien weiterer Klassen zulassen, soll auf die Möglichkeit eingegangen werden, wie durch „syntaktischen Zucker" Spezifikationen vereinfacht werden könnten. Im Hinblick auf weitere Klassen von Programmierrichtlinien sollen im speziellen die Möglichkeiten erörtert werden, wie Programmierrichtlinien der in Abschnitt 3.3 vorgestellten Klassen „Reihenfolge von Deklarationen" und „Verbot/Gebot von Redundanzen" ausgedrückt werden können. Die theoretischen Grundlagen, in denen das Aussehen der Baummuster beschrieben wird, die zur Prüfung von Programmierrichtlinien der genannten Klassen dienen, wurden in den Abschnitten 4.5.2 und 4.5.4 gelegt. Es fehlen aber Formalismen in der Beschreibungssprache, die die Spezifikation der Programmierrichtlinien dieser Klassen und damit auch die Umsetzung in die Baummuster möglich machen.

7.3.1 Hinzufügen von „syntaktischem Zucker"

Durch Erweiterung der Formalismen der Sprache CCML mit sogenanntem „syntaktischen Zucker" für die Muster- sowie Konventionsformulierung könnten kompaktere Spezifikationen von Programmierrichtlinien ermöglicht werden. „Syntaktischer Zucker" stellt keine Erweiterung der Ausdrucksfähigkeit der Sprache dar, sondern ermöglicht lediglich eine kürzere Spezifikation als mit den ursprünglichen Formalismen.

Die abstrakte Syntax unterliegt der Restriktion, daß Iterationen nicht erlaubt sind. Dies stellt keine Einschränkung dar, da Iterationen durch Produktionen ohne Iteration formuliert werden können (Äquivalenz der Ausdrucksstärke von EBNF und BNF). Durch Einführung einer Schreibweise für Iterationen in Muster und Konvention könnten die abstrakte Syntax und damit auch die Spezifikation der Programmierrichtlinien kürzer und kompakter werden. Ein möglicher Formalismus zur Formulierung von Iterationen in Mustern und Konventionen findet sich in [Pro94].

Durch die Kennzeichnung von Elementen als optionale Elemente in Muster und Konvention ist ebenfalls eine kompaktere Richtlinienspezifikation möglich. Mit dem vorgestellten Sprachumfang müssen zwei Regeln beim Auftreten eines optionalen Elementes in einer abstrakten Syntaxregel formuliert werden, um alle möglichen Teilbauminstanzen dieser abstrakten Syntaxregel im AST zu überdecken. Existieren in einer abstrakten Syntaxregel beispielsweise drei optionale Elemente, muß für jede mögliche Kombination dieser optionalen Elemente eine Regel spezifiziert werden, was insgesamt die Spezifikation von acht Regeln bedeutet. Mit einer Kennzeichnungsmöglichkeit für optionale Elemente wäre nur eine Regel notwendig.

7.3.2 Verbot/Gebot von Redundanzen

Zunächst soll die Klasse ,,Verbot/Gebot von Redundanzen" näher betrachtet werden. Die einfachste Überlegung besteht darin, in einer Konvention eine vom Muster abweichende Menge von Terminalen zuzulassen. Allerdings enthält die Konvention als Elemente der abstrakten Syntax nur Terminale und Variablen aber keinerlei struktuerelle Information über die Ableitung. Bloßes Hinzufügen oder Weglassen von Terminalen führt also im allgemeinen zu einem inkonsistenten abstrakten Syntaxbaum, da dem Baummuster b_2 einer Regel $p = (b_1, b_2)$ die Terminale ohne eventuell zugehörigen Teilbaum hinzugefügt oder entfernt wüden.

Aus den vorangegangenen Überlegungen wird klar, daß in CCML zur Spezifikation von Programmierrichtlinien der Klasse ,,Verbot/Gebot von Redundanzen" Formalismen geschaffen werden müssen, die es erlauben, die Struktur der Teilbäume der redundanten Elemente zu kennzeichnen. Da die Muster die Struktur der Teilbäume bereits beinhalten, basiert die folgende Idee auf Erweiterung der Musterformulierung. Erlaubt man im Muster eine spezielle Kennzeichnung von Elementen, womit ein zu redundanten Elementen gehöriger Teilbaum gekennzeichnet wird, ist die Struktur der hinzuzufügenden und wegzulassenden Elemente definiert. Das Beispiel in Abbildung 7.1 zeigt eine mögliche Formulierung für die Richtlinie, das ein ,,if"-Statement stets einen Block im ,,then"-Teil beinhalten soll. Alle Elemente im Muster, die zum Einfügen des Blocks hinzugefügt werden müssen, sind mit einem r im Index gekennzeichnet.

Eine unterschiedliche Bedeutung haben die gekennzeichneten Elemente je nachdem, ob Redundanz hinzugefügt oder entfernt werden soll. Im Fall, daß Redundanz hinzugefügt werden soll, sind die mit dem Index r gekennzeichneten Terminale auch Elemente des Konventionsteils (wie im Beispiel in Abbildung 7.1). Soll Redundanz entfernt werden, sind die Terminale mit Index r nicht Elemente des Konventionsteils.

Die mit einem Index gekennzeichneten Elemente des Musters sind als optionale Element im Baummuster b_1 der Regel p zu verstehen. Das Baummuster b_1 überdeckt sowohl Teilbäume des ASTs, die alle Elemente mit

```
Statement 'if' '(' cond ')'
    {Statement_r {Block_r '{_r' {StatementList_r s } '}_r' }}
-> V vs=0 is=0 [
     V vs=0 is=4 [
       H ['if' H hs=0 ['(' cond ')'] '{'
       s]
     '}']
```

Abbildung 7.1: Formulierung einer Richtlinie zum Hinzufügen von Redundanz

gleichem Index beinhalten als auch Teilbäume, bei denen alle Elemente mit gleichem Index fehlen.

7.3.3 Reihenfolge von Deklarationen

Problematischer als die Umsetzung von Programmierrichtlinien der Klasse „Verbot/Gebot von Redundanzen" ist die Umsetzung von Programmierrichtlinien der Klasse „Reihenfolge von Deklarationen". Abschnitt 4.5.2 beschreibt prinzipiell das Aussehen von Baummustern, die zwei Elemente vertauschen. Abbildung 7.2 zeigt eine einfache Formulierung der Vertauschung einer inneren Klasse mit einer Methode.

```
ClassElementList {ClassElement cd}
    {ClassElementList {ClassElement mod type varList ';'}}
-> V vs=2 is=0 [
       H ['mod type H hs=0 [varList ';']]
       cd];
```

Abbildung 7.2: Vertauschung von innerer Klasse und Methode

Bei einem einmaligen Durchlauf durch den abstrakten Syntaxbaum (mehr Durchläufe sind zur Formatierung nicht notwendig) führt die in Abbildung 7.2 definierte Vertauschung im allgemeinen zu unerwünschten Ergebnissen. Als Beispiel sei dazu die Situation betrachtet, in der auf zwei innere Klassen eine Methode folgt. Bei dem einmaligen Durchlauf durch den AST kommt es zur Vertauschung von der zweiten inneren Klasse und der Methode. Damit steht die Methode in der Mitte von zwei inneren Klassen, und die gewünschte Sortierung ist nicht erfolgt.

Zur vollständigen Umsetzung der Reihenfolgenrichtlinien sind so viele Durchläufe notwendig, bis alle Vertauschungen vorgenommen wurden. Idealerweise gibt es hier einen Fixpunkt, der nach einer Anzahl von Iterationen erreicht wird. Dies ist jedoch nicht sicherzustellen, wenn die Reihenfolgen wie in Abbildung 7.2 einzeln definiert werden. Sinnvoller ist die Einführung von Funktionen, die die Einhaltung einer Reihenfolge prüfen (als Guard) und eine Sortierung vornehmen (als GuardNotMatchedAction), indem die notwendigen Baummuster anhand dieser Sortierungs-Funktion generiert werden.

Insgesamt ist ein zweistufiges Verfahren denkbar. Im ersten Schritt werden alle Elemente gemäß der spezifizierten Reihenfolgen sortiert, und im zweiten Schritt werden Redundanzen hinzugefügt oder weggelassen, die Formatierungsrichtlinien umgesetzt und Namenskonventionen geprüft.

Kapitel 8

Fazit

Angelehnt an die Sprache PPML, die zur Definition von Pretty Printern entwickelt wurde, ermöglicht CCML die Spezifikation von Programmierrichtlinien für beliebige Programmiersprachen in Verbindung mit einer abstrakten Syntax der Programmiersprache. Neben den in CCML realisierten Beschreibungsmöglichkeiten für Formatierungsrichtlinien und Namenskonventionen wurden Ansätze vorgestellt, um Programmierrichtlinien weiterer Klassen durch einfache Erweiterungen von CCML beschreibbar zu machen.

CCML zeichnet sich durch eine leicht erlernbare, kompakte Syntax aus. Im Hinblick auf die Spezifikation von Formatierungsrichtlinien bietet das vorgestellte Konzept im Gegensatz zu existierenden Formatierungswerkzeugen den Vorteil der möglichen Rekonstruktion von originalen Formatierungen. Damit wird partielle Umformatierung ermöglicht. Als Neuerung ist zudem die Integration von programmiersprachenunabhängigen Beschreibungsformalismen für Formatierungsrichtlinien und Richtlinien weiterer Klassen hervorzuheben. Dies wurde bisher nur in der Form von programmiersprachenabhängigen Anwendungen realisiert.

Eine Weiterentwicklung der Beschreibungssprache unter Verfolgung der vorgestellten Konzepte erscheint aufgrund der Ausdrucksstärke der Sprache CCML sinnvoll. Zudem könnte durch Implementierung von Front-Ends für weitere Programmiersprachen ein firmenweit einheitlicher Formalismus etabliert werden, mit dem Programmierrichtlinien spezifiziert werden können, um eine automatisierte Prüfung auf Einhaltung von Programmierrichtlinien vorzunehmen.

Tabellenverzeichnis

3.1 Formatierungsrichtlinien 17
3.2 Namenspräfixe bei ungarischer Notation 18
3.3 Beispiele für Namenskonventionen in Java 19
3.4 Kodierung der syntaktischen Klassen bei Syntaxtables 22
3.5 Flags bei Syntaxtables 23

4.1 Zeilenumbrüche bei verschiedenen Betriebsystemklassen . . . 30
4.2 Klassifizierung von Whitespace und Kommentaren für Java . 32
4.3 Klassifizierung von Kommentaren für Java 33
4.4 Kommentarinterpretationen 47

5.1 Kombinatoren in PPML 56
5.2 Kombinatoren in Box 57
5.3 Zuordnung der Regelteile zu den Baummustern b_1 und b_2 . . 63

Abbildungsverzeichnis

2.1 Beispielbaum t (ohne Attribute) 6
2.2 Zwei verschiedene Ableitungsbäume für $3 + 4 \star 5$ 11
2.3 Unterteilung eines Übersetzers in Analyse und Synthese . . . 12
2.4 Phasen der Analyse 13

4.1 Ablauf der Anwendung 27
4.2 EBNF-Notation der Programmrepräsentation 29
4.3 EBNF-Notation von Globals und Whitespace 30
4.4 Programmfragment 34
4.5 Baumdarstellung 34
4.6 XML-Darstellung des AST mit allen Attributen 35
4.7 Baummuster 37
4.8 Überdeckung mit einem Baummuster 38
4.9 Eine weitere mögliche Überdeckung 38
4.10 Eine dritte mögliche Überdeckung 39
4.11 „whitespace"-Attribut in einem Baummuster 43
4.12 Beispiel für Whitespace und einen Blockkommentar 44
4.13 Beispiel für Whitespace und Zeilenendekommentare 44
4.14 Falsche Richtlinienumsetzung 45
4.15 Generierter Code von JBuilder der Firma Borland 45
4.16 Beispiel für Whitespace, Zeilenende- und Blockkommentar . . 46
4.17 Listen zur Vertauschung von Reihenfolgen 47
4.18 Beispiel für einen Knoten mit Namenskonvention 48

5.1 Einrückung von „case" 51
5.2 Keine Einrückung von „case" 51
5.3 Kombinatoren zum Umgang mit der Datenstruktur Doc . . . 52
5.4 Layoutdefinition der Datenstruktur Tree 53
5.5 Gelayouteter Baum 53
5.6 Geänderte Funktion showTree 54
5.7 Gelayouteter Baum bei 30 Zeichen Zeilenbreite 54
5.8 Regel in PPML 55
5.9 Beispiele für Muster in PPML 55
5.10 Beispiel für eine Regel in PPML 56

5.11 Beispiel für eine Regel in Box 57
5.12 Wirkung der Kombinatoren in Box (anschaulich) 58
5.13 Ausdrücken von HV durch ALT, H und V 58
5.14 Unterteilung der CCML-Spezifikation in zwei Abschnitte . . . 60
5.15 Globale Festlegungen in CCML 61
5.16 Globale Festlegungen für Java 62
5.17 Globale Festlegungen in CCML 63
5.18 Grammatik für Muster 64
5.19 Beispiele für Muster 65
5.20 Baummuster umgesetzt aus Muster (ii), Abb. 5.19 66
5.21 Grammatik für Konventionen 67
5.22 Beispiele für Konventionen 69
5.23 Spezifizierte Formatierungen 69
5.24 Baummuster umgesetzt aus Konvention (iv), Abb. 5.22 . . . 71
5.25 Grammatik für Guard 71
5.26 Beispiel für eine Regel mit Guard 72
5.27 Baummuster umgesetzt aus Muster und Guard, Abb. 5.26 . . 73
5.28 Grammatik für GuardNotMatchedAction 73
5.29 Spezifikation zur Prüfung einer Namenskonvention 74

6.1 Übersicht über die wichtigsten Klassen 76
6.2 Vererbung beim XMLTreeGenerator 78
6.3 Klassenhierarchie für den AST 79
6.4 Klassen zur Darstellung von Regeln nach Abschnitt 5.2.2 . . 80
6.5 Klassen zur Darstellung eines Musters 81
6.6 Klassen zur Darstellung eines Musters 81
6.7 Optionale Layouts für binäre Ausdrücke 84
6.8 Formatierung bei Überschreitung der Zeilenbreite durch c . . 84
6.9 Beispiel für einen Printerdurchlauf 85
6.10 Steuerung der Anwendung 86
6.11 Klasse beispiel der Datei beispiel.java 86
6.12 CCML-Spezifikation der Datei „test.ccml” 87
6.13 Umformatierter Programmquelltext 88
6.14 Statusfenster der Anwendung 89

7.1 Formulierung einer Richtlinie zum Hinzufügen von Redundanz 94
7.2 Vertauschung von innerer Klasse und Methode 94

Literaturverzeichnis

[App98] Andrew W. Appel. *Modern compiler implementation in Java*. Cambridge University Press, Cambridge, 1st edition, 1998.

[BCD+89] P. Borras, D. Clement, Th. Despeyrouz, J. Incerpi, G. Kahn, B. Lang, and V. Pascual. CENTAUR: The system. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments (PSDE)*, volume 24, pages 14–24, New York, NY, 1989. ACM Press. Verfügbar unter citeseer.nj.nec.com/borras88centaur.html.

[BL02] Richard Stallman Bil Lewis, Dan LaLiberte. *GNU Emacs Lisp Reference Manual*. Free Sofware Foundation, Boston, 2.8 edition, Januar 2002.

[CS91] Martin Heller Charles Simonyi. The hungarian revolution. *BYTE*, 16(8):131–38, August 1991.

[EMC86] A. Conchon E. Morcos-Chounet. PPML: A general formalism to specify prettyprinting. In *Proceedings of IFIP Congress*, pages 583–590, Dublin, North-Holland, 1986. Elsevier Science Publisher B.V.

[Gag98] Etienne Gagnon. SableCC, an object-oriented compiler framework. Master's thesis, School of Computer Science, McGill University, Montreal, 1998. Verfügbar unter http://www.sablecc.org/thesis.pdf.

[HHKR89] C. Heering, P. R. H. Hendriks, P. Klint, and J. Rekers. The syntax definition formalism SDF - reference manual -. *SIGPLAN Notices*, 24(11):43–75, 1989.

[Hug95] John Hughes. The Design of a Pretty-printing Library. In J. Jeuring and E. Meijer, editors, *Advanced Functional Programming*, pages 53–96. Springer Verlag, LNCS 925, 1995.

[ISO96] International Electrotechnical Commission International Standards Organization. Extended Backus Naur Form. ISO/IEC 14977:1996(e), ISO/IEC, 1996.

[Joh78] S.C. Johnson. Lint, a c program checker, 1978. Verfügbar unter citeseer.nj.nec.com/johnson78lint.html.

[Jon00] M. de Jonge. A pretty-printer for every occasion. In Ian Ferguson, Jonathan Gray, and Louise Scott, editors, *Proceedings of the 2nd International Symposium on Constructing Software Engineering Tools (CoSET2000)*, pages 68–77. University of Wollongong, Australia, 2000.

[Mic99] Sun Microsystems. Java code conventions, 1999. Verfügbar unter http://java.sun.com/docs/codeconv/.

[MvdB96] Eelco Visser Mark van den Brand. Generation of formatters for context-free languages. *ACM Transactions on Software Engineering and Methodology*, 5(1):1–41, 1996. Verfügbar unter citeseer.nj.nec.com/vandenbrand99generation.html.

[Opp80] Dereck C. Oppen. Prettyprinting. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 2(4):465–483, 1980.

[PRA98] S. Doaitse Swietstra Pablo R. Azero. Optimal pretty-printing combinators, 1998. Verfügbar unter http://www.cs.uu.nl/groups/ST/Software/PP/pabloicfp.ps.

[Pro94] INRIA: Centaur Project. *The PPML Manual.* Sophia-Antipolis, France, 1994. Für Centaur Version 1.2. Verfügbar via ftp von babar.inria.fr in directory pub/croap/bertot.

[TA02] A. Dangberg T. Andree. *Maual for JIndent Version 3.5.* sofware & solutions, 2002. Verfügbar unter http://www.jindent.com.

[Wad98] Philip Wadler. A prettier printer (draft), 1998. Verfügbar unter http://www.research.avayalabs.com/user/wadler/topics/language-design.html.

[Wor00] World Wide Web Consortium (W3C). Extensible markup language (XML) 1.0 (second edition), 2000. Verfügbar unter http://www.w3.org/TR/REC-xml.

Anhang A

Grammatik der Beschreibungssprache

Die Grammatik der Beschreibungssprache, die auch in den Unterabschnitten von Abschnitt 5.2 stückweise vorgestellt wird, ist in diesem Anhang nochmal zusammengefasst. Das Startsymbol der Grammatik ist CCML-Spec. Nicht aufgeführt sind die regulären Ausdrücke zur Definition der Terminalsymbole number (Zahl), terminal (Terminalsymbol notiert in der Form 't'), nonterminal (Nichterminalsymbol, beginnend mit einem Großbuchstaben) variable (Variablenname aus Buchstaben und Ziffern beginnend mit einem Buchstaben) und inputChar (alle erlaubten Eingabesymbole).

```
CCML-Spec
= Globals Rules ;

Globals
= 'Globals' HTranslation* VTranslation* LineWitdhDefiniton? ;

HTranslation
= 'htrans' 'type' '=' number 'number' '=' number;

VTranslation
= 'vtrans' 'type' '=' number 'number' '=' number;

LineWitdhDefiniton
= 'linewidth' '=' number;

Rules
= 'Rules' Rule*;

Rule
= NoGuardRule
```

```
| GuardRule;

NoGuardRule
= Pattern '->' Convention ';' ;

GuardRule
= Pattern '|' Guard '->' Convention ['|' GuardNotMatchedAction] ';' ;

Pattern
= nonterminal PatternTail ;

PatternTail
= terminal
| variable
| '{' Pattern '}' ;

Convention
= terminal
| variable
| Combinator ;

Combinator
= 'H' ['hs=' number] '[' {Convention} ']'
| 'V' ['vs=' number] ['is=' number] '[' {Convention} ']'
| 'VL' ['vs=' number] ['is=' number] '[' {Convention} ']'
| 'ID' '[' {Convention} ']'
| 'O' '[' [Convention {',' Convention}] ']' ;

Guard
= GuardFunction
= '( GuardFunction '&&' Guard ')' ;

GuardFunction
= regexMatch '(' AttributName ',' '"' {inputChar} '"' ')' ;

AttributName
= {inputChar} '.' {inputChar} ;

GuardNotMatchedAction
= warn '(' Message ')' ;

Message
= MessagePart
| MessagePart ',' Message ;
```

MessagePart
= '"' {inputChar} '"'
| AttributName ;

Anhang B

Abstrakte Syntax für Java

In diesem Abschnitt ist die abstrakte Syntax für Java, die in dieser Arbeit entstanden ist als Grammatik in EBNF Notation gemäß Kapitel 2.3 angegeben. Nichtterminalsymbole beginnen mit einem Großbuchstaben und Terminale stehen in einfachen Anführungszeichen. Symbole, die mit einem Kleinbuchstaben beginnen, stehen für eine (un-)endliche Menge von Terminalsymbolen. Das Startsymbol der Grammatik ist AST.

B.1 Globale Umgebung

```
AST
  = [PackageDecl] [ImportDeclList] [TypeDeclList] ;

ImportDeclList
  = ImportDecl
    | ImportDecl ImportDeclList ;

TypeDeclList
  = TypeDecl
    | TypeDecl TypeDeclList ;

TypeDecl
  = ClassDecl
    | InterfaceDecl
    | ';' ;
```

B.2 Deklarationen

```
PackageDecl
```

= 'package' Name ';' ;

ImportDecl
= 'import' Name ['.' '*'] ';' ;

ClassDecl
= [Modifiers] 'class' identifier [ExtendsClause] [InterfacesClause] ClassBody ;

InterfaceDecl
= [Modifiers] 'interface' identifier [InterfacesClause] InterfaceBody ;

ExtendsClause
= 'extends' Name ;

InterfacesClause
= ('extends' | 'implements') ClassNameList ;

ClassNameList
= Name
| Name ',' ClassNameList ;

ClassBody
= '{' [ClassElementList] '}' ;

ClassElementList
= ClassElement
| ClassElement ClassElementList;

InterfaceBody
= '{' [InterfaceElementList] '}' ;

InterfaceElementList
= InterfaceElement
| InterfaceElement InterfaceElementList ;

B.3 Klassen/Interface Elemente

ClassElement
= [Modifiers] identifier '(' [ParameterList] ')' [ThrowsClause] ConstructorBody
| [Modifiers] Type identifier '(' [ParameterList] ')' [ArrayBracketList]
[ThrowsClause] MethodBody
| [Modifiers] Type VariableList ';'
| ClassDecl

```
  | InterfaceDecl
  | Block
  | 'static' Block
  | ';' ;

ThrowsClause
  = 'throws' ClassNameList ;

ConstructorBody
  = '{' [ExplicitConstructorInvocation] [StatementList] '}' ;

ExplicitConstructorInvocation
  = ([Expression '.'] 'super' | 'this') '(' [ArgumentList] ')' ';' ;

ParameterList
  = Parameter
  | Parameter ',' ParameterList ;

Parameter
  = ['final'] Type identifier [ArrayBracketList] ;

ArrayBracketList
  = '[' ']'
  | '[' ']' ArrayBracketList ;

MethodBody
  = '{' [StatementList] '}' ;

InterfaceElement
  = [Modifiers] Type Name '(' [ParameterList] ')' ';'
  | [Modifiers] Type VariableList ';'
  | ClassDecl
  | InterfaceDecl
  | ';' ;
```

B.4 Anweisungen

```
StatementList
  = Statement
  | Statement StatementList ;

Statement
  = Block
```

```
  | 'break' [identifier] ';'
  | 'continue' [identifier] ';'
  | ClassDecl
  | 'do' Statement 'while' '(' Expression ')' ';'
  | ';'
  | Expression ';'
  | 'for' '(' [ForInit] ';' [Expression] ';' [ExpressionList] ')' Statement
  | 'if' '(' Expression ')' Statement ['else' Statement]
  | identifier ':' Statement
  | LocalVariableDecl ';'
  | 'return' [Expression] ';'
  | 'switch' '(' Expression ')' '{' [SwitchStatementGroupList] '}'
  | synchronized' '(' Expression ')' Block
  | 'throw' Expression ';'
  | 'try' Block [CatchClauseList]
  | 'while' '(' Expression ')' Statement ;

Block
 = '{' [StatementList] '}' ;

ForInit
 = LocalVariableDecl
  | ExpressionList ;

LocalVariableDecl
 = [Modifiers] Type VariableList ;

ExpressionList
 = Expression
  | Expression ',' ExpressionList ;

SwitchStatementGroupList
 = SwitchStatementGroup
  | SwitchStatementGroup SwitchStatementGroupList ;

SwitchStatementGroup
 = ( 'case' Expression | 'default') ':' [StatementList] ;

CatchClauseList
 = CatchClause
  | CatchClause CatchClauseList ;

CatchClause
 = 'catch' '(' Parameter ')' Block ;
```

B.5 Ausdrücke

Expression
= '(' Expression ')'
| 'this'
| Name '.' 'this'
| literal
| Type '.' 'class'
| Name
| (Name | Expression) ArrayAccessList
| [Expression '.'] 'new' (Name | identifier)'(' [ArgumentList] ')' [ClassBody]
| 'new' Type ArrayCreationList [ArrayInitializer]
| Name '(' ArgumentList ')'
| (Expression | [Name '.'] 'super') '.' identifier
| (Expression | [Name '.'] 'super') '.' identifier '(' [ArgumentList] ')'
| Expression ('++' | '–')
| ('++' | '–') Expression
| ('+' | '-' | '!' | '∼') Expression
| '(' (Type [ArrayBracketList] | Expression) ')' Expression
| Expression binOp Expression
| Expression '?' Expression ':' Expression ;

ArrayAccessList
= '[' Expression ']'
| '[' Expression ']' ArrayAccessList ;

ArgumentList
= Expression
| Expression ',' ArgumentList ;

ArrayCreationList
= '[' [Expression] ']'
| '[' [Expression] ']' ArrayCreationList ;

B.6 Hilfsproduktionen

Modifiers
= {'public' | 'protected' | 'private' | 'static' | 'abstract' | 'final' |
'native' | 'synchronized' | 'transient' | 'volatile' | 'strictfp'} ;

Type
= (primitiveType | Name | 'void') [ArrayBracketList] ;

VariableList
= identifier [ArrayBracketList] ['=' Initializer]
| identifier [ArrayBracketList] ['=' Initializer] ',' VariableList ;

InitializerList
= Initializer
| Initializer ',' InitializerList ;

Initializer
= Expression
| ArrayInitializer ;

ArrayInitializer
= '{' [InitializerList] '}' ;

Name
= identifier
| identifier '.' Name ;

B.7 Terminale

binOp
= '*' | '%' | '/'
| '+' | '-'
| '<<' | '>>' | '>>>'
| '<' | '>' | '>=' | '<=' | 'instanceof'
| '==' | '! ='
| '&' | ' ˆ ' | '|' | '&&' | '||'
| '=' | '*=' | '/ =' | '%=' | '+ =' | '− ='
| '<<=' | '>>=' | '>>>=' | '&=' | '| =' | '! =' | ' ˆ ='

primitiveType
= 'boolean'
| 'byte'
| 'short'
| 'int'
| 'long'
| 'float'
| 'double'
| 'char' ;

literal
= 'true', 'false', numerische oder Character-Konstante ;

identifier
= in Java zugelassene Identifier ;

Anhang C

Beispielspezifikation für Java

Die folgende Spezifikation ist ein Beispiel für eine Spezifikation von Formatierungsrichtlinien und Namenskonventionen für Java. Kommentare sind in einer CCML-Spezifikation in Form von Java Block- oder Zeilenendekommentaren möglich. Die CCML-Schlüsselwörter Globals, Rules, htrans, vtrans, blockCommentRole, eolCommentRole, role, pre, post, type, number, linewidth, H, V, VL, O, ID, hs, vs und is sind nicht als Mustervariablen zulässig. Leerzeichen zwischen den optionalen Paramtern hs, vs, is und dem Gleichheitszeichen sind ebenfalls nicht zulässig.

```
// Globale Festlegungen
Globals
  htrans type=1 number=4
  vtrans type=3 number=2
  linewidth = 100

// Regeln
Rules

  // Deklarationen

  // Klassendeklaration
  ClassDecl mod 'class' id {ClassBody '{' cel '}' }
    | regexMatch(id.value, "[A-Z].*")
  -> V vs=1 is=0 [V vs=1 is=4 [H [mod 'class' id '{'] cel] '}']
    | warn("Namenskonvention verletzt in Zeile ", id.line,
           ": Klassennamen beginnen mit Grobuchstaben");

  // Interfacedeklaration
  InterfaceDecl mod 'interface' id {InterfaceBody '{' iel '}' }
    | regexMatch(id.value, "[A-Z].*")
  -> V vs=1 is=0 [V vs=1 is=4 [H [mod 'interface' id '{'] iel] '}']
```

```
  | warn("Namenskonvention verletzt in Zeile ", id.line,
         ": Interfacenamen beginnen mit Grobuchstaben");
```

// **Klassen- und Interfaceelemente**

```
// Konstanten
ClassElement {Modifiers 'private' 'static' 'final'} typ {VariableList id '=' init} ';'
  | regexMatch(id.value, "([A-Z]|_)*")
-> H ['private' 'static' 'final' typ id '=' init ';']
  | warn("Unzulässiger Konstantenname in Zeile", id.line );

ClassElement {Modifiers 'public' 'static' 'final'} typ {VariableList id '=' init} ';'
  | regexMatch(id.value, "([A-Z]|_)*")
-> H ['public' 'static' 'final' typ id '=' init ';']
  | warn("Unzulässiger Konstantenname in Zeile", id.line );

// Instanzen und Klassen Variablen
ClassElement mods typ varList ';'
-> H [mods typ varList ';'] ;

ClassElement typ varList ';'
-> H [typ varList ';'] ;

InterfaceElement mods typ varList ';'
-> H [mods typ varList ';'] ;

InterfaceElement typ varList ';'
-> H [typ varList ';'] ;

// Konstruktoren
ClassElement mods id '(' ')' cbody
-> H [mods H hs=0 [id '(' ')'] cbody] ;

ClassElement mods id '(' pList ')' cbody
-> H [mods H hs=0 [id '(' pList ')'] cbody];

ConstructorBody '{' sList '}'
-> VL [VL vs=0 is=4 ['{' sList] '}'] ;

// Methoden
ClassElement mods typ id '(' ')' mbody
-> H [mods typ H hs=0 [id '(' ')'] mbody];

ClassElement mods typ id '(' pList ')' mbody
```

```
-> H [mods typ H hs=0 [id '(' pList ')'] mbody];

ClassElement mods typ id '(' ')' tc mbody
-> H [V vs=0 is=8 [H [mods typ H hs=0 [id '(' ')']] tc] mbody];

ClassElement mods typ id '(' pList ')' tc mbody
-> H [V vs=0 is=8 [H [mods typ H hs=0 [id '(' pList ')']] tc] mbody];

MethodBody '{' sList '}'
-> VL [VL vs=0 is=4 ['{' sList] '}'] ;

// abstrakte Methoden
InterfaceElement mods typ id '(' ')' ';'
-> H [mods typ H hs=0 [id '(' ')'] ';'];

InterfaceElement mods typ id '(' pList ')' ';'
-> H [mods typ H hs=0 [id '(' pList ')'] ';'];
```

// **Anweisungen**

```
StatementList stat statList
-> V [stat statList] ;

// Blöcke
Block '{' sl '}'
-> VL [VL vs=0 is=4 ['{' sl] '}'] ;

Block '{' '}'
-> H hs=0 ['{' '}'] ;

// if-Statements
Statement 'if' '(' exp ')' {Statement {Block '{' statList '}'}}
-> V [V vs=0 is=4 [H ['if' H hs=0 ['(' exp ')'] '{'] statList] '}'];

Statement 'if' '(' exp ')' stat
-> V vs=0 is=4 [H hs=1 ['if' H hs=0 ['(' exp ')']] stat];

// for-Statements
Statement 'for' '(' forInit ';' exp ';' expList ')' {Statement {Block '{' sl '}'}}
-> V [V vs=0 is=4 [H ['for' H hs=0 ['(' forInit ';' exp ';' expList ')'] '{'] sl] '}'];

Statement 'for' '(' forInit ';' exp ';' expList ')' stat
-> V vs=0 is=4 [H ['for' H hs=0 ['(' forInit ';' exp ';' expList ')']] stat] ;
```

```
ForInit {LocalVariableDecl typ {VariableList id '=' init}}
  | regexMatch(id.value, "[i|j|k|l|m|n]")
-> H [typ id '=' init]
  | warn("Bevorzugte Schleifenvariablen i,j,k,l,m oder n verwenden in Zeile ", id.line);

// Expression Statement
Statement exp ';'
-> H hs=0 [exp ';'] ;

// Lokale Variablen Deklaration
Statement lvd ';'
-> H hs=0 [lvd ';'] ;

LocalVariableDecl mod typ varList
-> H [mod typ varList] ;

// Return Statement
Statement 'return' exp ';'
-> H ['return' H hs=0 [exp ';']] ;

Statement 'return' ';'
-> H hs=0 ['return' ';'] ;

SwitchStatementGroupList switchStat switchStatList
-> V [switchStat switchStatList] ;

SwitchStatementGroup 'case' exp ':' {StatementList {Statement {Block '{' sl '}'}}}
-> V [V vs=0 is=4 [H ['case' H hs=0 [exp ':'] '{'] sl] '}'];

SwitchStatementGroup 'case' exp ':' sl
-> V vs=0 is=4 [H ['case' H hs=0 [exp ':']] sl] ;

SwitchStatementGroup 'case' exp ':'
-> H ['case' H hs=0 [exp ':']] ;

SwitchStatementGroup 'default' ':' {StatementList {Statement {Block '{' sl '}'}}}
-> V [V vs=0 is=4 [H [H hs=0 ['default' ':'] ''] sl] ''];

SwitchStatementGroup 'default' ':' sl
-> V vs=0 is=4 [H hs=0 ['default' ':'] sl] ;

SwitchStatementGroup 'default' ':'
-> H ['default' H hs=0 [exp ':']] ;
```

```
// Synchronized Statement
SynchronizedStatement 'synchronized' '(' exp ')' bl
-> H ['synchronized' H hs=0 ['(' exp ')'] bl] ;

// Try Statement
Statement 'try' bl ccl
-> H ['try' bl ccl] ;

Statement 'try' bl
-> H ['try' bl] ;

CatchClauseList {CatchClause 'catch' '(' param ')' bl} ccl
-> V [H ['catch' H hs=0 ['(' param ')'] bl] ccl] ;

CatchClauseList {CatchClause 'catch' '(' param ')' bl}
-> H ['catch' H hs=0 ['(' param ')'] bl] ;

// While Statement
Statement 'while' '(' exp ')' {Block '{' sl '}'}
-> V [V vs=0 is=4 [H ['while' H hs=0['(' exp ')'] '{'] sl] '}'] ;

Statement 'while' '(' exp ')' stat
-> V vs=0 is=4 [H hs=1 ['while' H hs=0['(' exp ')']] stat] ;

// Ausdrücke

// Methodenaufrufe
Expression name '(' aList ')'
-> H hs=0 [name '(' aList ')'] ;

// Geklammerter Ausdruck
Expression '(' exp ')'
-> H hs=0 ['(' exp ')'] ;

// Methodenaufrufe
Expression name '(' ')'
-> H hs=0 [ name '(' ')'] ;

Expression name '(' aList ')'
-> H hs=0 [ name '(' aList ')'] ;

// Binäre Ausdrücke
Expression a binop b
-> O [H [a binop b], V [H [a binop] b]] ;
```

```
// Ternärer Ausdruck
Expression exp1 '?' exp2 ':' exp3
-> O [H [exp1 '?' exp2 ':' exp3],
  H [exp1 '?' V [ H [exp2 ':'] exp3]]] ;

// Instanziierung
Expression 'new' name '(' ')'
-> H ['new' H hs=0 [ name '(' ')']] ;

Expression 'new' name '(' aList ')'
-> H ['new' H hs=0 [ name '(' aList ')']] ;

Expression 'new' name '(' aList ')' {ClassBody '{' cel '}'}
-> V [V vs=0 is=4 [H ['new' H hs=0 [ name '(' aList ')'] '{'] cel] '}'] ;
```

// **Listen**

```
// Klassennamen Liste
ClassNameList name ',' clNameList
-> V [H hs=0 [name ','] clNameList] ;

// Parameter Liste
ParameterList p ',' pList
-> V [H hs=0 [p ','] pList] ;

// Argument Liste
ArgumentList a ',' aList
-> V [H hs=0 [a ','] aList] ;

// Variablen Liste
VariableList id '=' init
  | regexMatch(id.value, "[a-z].*")
-> H [id '=' init]
  | warn("Unzulässiger Variablenname in Zeile", id.line );

VariableList id
  | regexMatch(id.value, "[a-z].*")
-> H [id]
  | warn("Unzulässiger Variablenname in Zeile", id.line );

VariableList id '=' init varList
  | regexMatch(id.value, "[a-z].*")
-> V [H hs=0 [H [id '=' init] ','] varList]
```

```
  | warn("Unzulässiger Variablenname in Zeile", id.line );

 VariableList id varList
  | regexMatch(id.value, "[a-z].*")
-> V [H hs=0 [id ','] varList]
  | warn("Unzulässiger Variablenname in Zeile", id.line );
```

Anhang D

Hinweise zur Anwendung

Im folgenden werden ein paar kurze Hinweise zu Installation und Starten der Anwendung sowie den Systemvoraussetzungen gegeben. Entwickelt und getestet wurde die Anwendung auf einem PC mit Suse Linux mit einem Kernel der Version 2.2.14 als Betriebssystem. Der Entwicklungs-PC hatte einen Pentium III Prozessor mit 750 MHz und 948 MB Hauptspeicher.

Implementiert wurde in der Programmiersprache Java unter Nutzung des Java Development Kit (JDK) 1.4.0. Das JDK 1.4.0 ist notwendige Voraussetzung. Eine Übersetzung des Java-Sourcecodes unter einer früheren Version des JDK ist nicht möglich, da die Klassen des Pakets java.util.regex genutzt werden, die eine Neuerung im JDK 1.4.0 darstellen. Aufgrund der Plattformunabhängigkeit von Java sollte die Anwendung unter allen Betriebssystemen lauffähig sein, für die es eine Java Laufzeitumgebung der Version 1.4 gibt. Neben dem Entwicklungssystem wurde die Anwendung außerdem auf einem PC mit Microsoft Windows NT als Betriebsystem und einem 200 MHz Pentium I Prozessor sowie 64 MB Hauptspeicher erfolgreich getestet.

Zur komfortablen Nutzung der Anwendung wurden die notwendigen Dateien zu einer Jar-Datei mit Namen codeconv.jar gepackt. Zusätzlich wird eine Datei abstractSyntax.dtd benötigt, die in Form einer Document Type Definition die abstrakte Syntax der Programmiersprache enthält, für die die Programmierrichtlinien geprüft werden sollen (hier also die abstrakte Syntax für Java gemäß Anhang B). Die beiden Dateien codeconv.jar und abstractSyntax.dtd müssen im gleichen Verzeichnis liegen. Dann kann das Programm durch Aufruf von:
java -jar codeconv.jar
aus der Kommandozeile gestartet werden.

Eidesstattliche Erklärung

Ich versichere vorliegende Arbeit selbständig verfaßt und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet zu haben.

Kiel, den 14.11.2002