DIPLOMARBEIT

in der Fachrichtung

Informatik

THEMA

Entwicklung eines Java-nach-C++-Übersetzers zur Transferierung von Java-API's

Eingereicht von: Jan-Philipp Rathje (Matrikelnummer: 963367)
Hohenrade 2
24106 Kiel
Tel.: (0431) 2391771
E-Mail: jpr@informatik.uni-kiel.de

Bearbeitungszeitraum: 05.11.2008 - 05.05.2009

Betreuender Professor: Prof. Dr. Michael Hanus

Betrieblicher Betreuer: Dipl.-Math. Oliver Schmidt
PPI AG Informationstechnologie
Wall 55
24103 Kiel
E-Mail: oliver.schmidt@ppi.de

Yes, I did say something along the lines of "C makes it easy to shoot yourself in the foot; C++ makes it harder, but when you do it blows your whole leg off." What people tend to miss, is that what I said there about C++ is to a varying extent true for all powerful languages.

Bjarne Stroustrup

Eidesstattliche Erklärung

Ich versichere, dass ich die vorliegende Arbeit selbständig angefertigt und mich fremder Hilfe nicht bedient habe. Alle Stellen, die wörtlich oder sinngemäß veröffentlichtem Schrifttum entnommen sind, habe ich als solche kenntlich gemacht.

Kiel, den 05.05.2009 _____

Inhaltsverzeichnis

1 Einleitung 1
1.1 PPI AG Informationstechnologie . . . 1
1.2 Problemstellung und Motivation . . . 1
1.3 Zieldefinition . . . 3
1.4 Zielgruppe . . . 4

2 Grundlagen 5
2.1 Definition API und Java-API . . . 5
2.2 Vergleich Java ↔ C++ . . . 5
2.2.1 Historie . . . 5
2.2.1.1 C++-Historie . . . 5
2.2.1.2 Java-Historie . . . 6
2.2.2 Einsatzgebiete . . . 6
2.2.3 Mehrdeutigkeit / Schwere der Sprache . . . 7
2.2.4 Sprachmerkmale im Vergleich . . . 8
2.2.4.1 Allgemeine Struktur . . . 8
2.2.4.2 Zeiger, Referenzen, etc. . . . 8
2.2.4.3 Speicherverwaltung . . . 9
2.2.4.4 Generische Programmierung . . . 9
2.2.4.5 Sonstiges . . . 10
2.3 Grundlagen zum Übersetzerbau . . . 12
2.3.1 Allgemeiner Aufbau . . . 12
2.3.2 Umwandlung einer Zeichenkette in eine Kurzzeichenkette . . . 13
2.3.3 Überprüfung auf syntaktische Korrektheit . . . 14
2.3.3.1 Top-Down-Parsing . . . 14
2.3.3.2 Bottom-Up-Parsing . . . 14
2.3.3.3 Zusammenfassung LR- und LL-Parsing . . . 15
2.3.4 Generierung von ASB's . . . 15
2.3.5 Semantische Korrektheitsüberprüfungen . . . 15
2.3.6 Zielcodegenerierung . . . 16

3 Evaluierung 17
3.1 Evaluierung existierender Java-nach-C(++)-Übersetzer . . . 17
3.1.1 Anforderungsprofil . . . 17
3.1.2 Werkzeuge - Java nach C / C++ . . . 19
3.1.3 Werkzeuge - Java-Bytecode nach C / C++ . . . 21

3.1.4 Sonstige . . . 24
3.1.5 Abwägung der verschiedenen Alternativen . . . 26
3.2 Auswahl eines Parsergenerators . . . 27
3.2.1 Notwendigkeit eines Parsergenerators . . . 27
3.2.2 Anforderungen an einen Parsergenerator . . . 27
3.2.3 Vorstellung der verschiedenen Parsergeneratoren . . . 28
3.2.4 Parsergenerator ANTLR . . . 30
3.2.4.1 Allgemeine Struktur . . . 30
3.2.4.2 Beispiel . . . 35

4 Übersetzerkonstruktion 37
4.1 Konkretisierung der Zielsetzung . . . 37
4.2 Zuordnung der Java-Konstrukte . . . 39
4.2.1 Klassenpräambel . . . 39
4.2.2 Klassendefinition . . . 43
4.2.2.1 „*Normale*“ Klassen, abstrakte Klassen und Schnittstellen 43
4.2.2.2 Enumerations . . . 45
4.2.3 Objekttypen der verschiedenen Klassendefinitionen . . . 46
4.2.3.1 Heap-Objekt . . . 46
4.2.3.2 Value-Objekt . . . 48
4.2.3.3 Enum-Objekt . . . 49
4.2.3.4 Exception-Objekt . . . 49
4.2.3.5 Array-Objekt . . . 49
4.2.3.6 String-Objekt . . . 51
4.2.3.7 Interface-Objekt . . . 53
4.2.4 Klassenrumpf . . . 54
4.2.4.1 Sichtbarkeitsattribute . . . 54
4.2.4.2 throws-Direktive . . . 56
4.2.4.3 Statische Klassenmitglieder . . . 57
4.2.4.4 Main-Funktion . . . 57
4.2.4.5 Nicht statische Klassenmitglieder . . . 58
4.2.4.5.1 Klassenvariablen . . . 58
4.2.4.5.2 Methoden . . . 59
4.2.4.5.3 rein virtuelle Methoden . . . 59
4.2.4.5.4 Konstruktoren . . . 59
4.2.5 Sonstiges . . . 60
4.3 Vorbereitende Maßnahmen für den Übersetzungsprozess . . . 62
4.3.1 Konfigurationsdatei . . . 62
4.3.1.1 Syntax . . . 62
4.3.1.2 Inhalt . . . 63
4.3.1.2.1 namespaces . . . 63
4.3.1.2.2 types . . . 63
4.3.1.2.3 baseClasses . . . 64
4.3.1.2.4 substitute . . . 65

4.3.1.2.5 classes 65
4.3.2 Definition und Anpassung des Ausgabeformats 66
4.3.3 Kommandozeilenargumente für den Programmstart 67

5 Beschreibung des Übersetzungsprozesses 68
5.1 Vorbereitung und Initiale Phase 69
5.1.1 Vorbereitung 70
5.1.2 Initiale Phase 73
5.2 Erste Zwischenphase 77
5.2.1 Erneutes Parsen der ASB's 78
5.2.2 Bestimmung der Klassenhierachie und der Objekttypen 79
5.2.3 Bestimmung zwingend virtueller Methoden 81
5.3 Zweite Zwischenphase 85
5.3.1 Anreicherung der ASB mit Typinformationen 85
5.3.2 Auflösung von Methodenaufrufen 88
5.3.3 Weitere Aufgaben der Zweiten Zwischenphase 91
5.4 Dritte Zwischenphase 92
5.4.1 Substitutionen 92
5.4.2 Ersetzung von Arraykonstruktoraufrufen 92
5.4.3 Anfertigung der Initialisierungslisten 93
5.4.4 Sonstiges 93
5.5 Ausgabephase 94
5.5.1 Beschreibung des Aufbau's von StringTemplate's 94
5.5.2 Integration von StringTemplate's 96
5.5.3 Definition der Anforderungen an die Ausgabe 97
5.5.4 Umsetzung der Anforderumgen 98

6 Praxisbeispiel 100
6.1 Beschreibung des FinTS-Kernel 100
6.2 Übersetzung der DTA-Toolbox 101
6.3 Übersetzung der SWIFT-Toolbox 102
6.4 Diskussion der Ergebnisse der Übersetzung 103

7 Erweiterung des Übersetzers 105
7.1 Detektion von Zyklen, Herausgabe des this-Zeiger's 105
7.2 Generische Konstrukte, Erweiterung der Unterstützung von Enumerationen und Schnittstellen 106
7.3 Annotationen 108
7.4 this-Konstruktoraufrufe 109
7.5 Sonstiges 111

8 Fazit und Ausblick 113

Anhang 118

A Beschreibung der Verwendung der beilie- genden CD mit den Quelldateien des entwickelten Übersetzers 119
A.1 Wurzelverzeichnis . . . 119
A.2 Ordner „*grammars*“ . . . 120
A.3 Ordner „*lib*“ . . . 120
A.4 Ordner „*src/j2cpp*“ . . . 120
A.5 Ordner „*cpp*“ . . . 123
A.6 Ordner „*testproject*“ . . . 123

B Messergebnisse zum Praxisbeispiel 124
B.1 LOC DTA-Toolbox . . . 124
B.2 Laufzeit DTA-Toolbox . . . 125
B.3 LOC SWIFT-Toolbox . . . 126
B.4 Laufzeit SWIFT-Toolbox . . . 127

C Code, Grammatiken und andere Beispiele 128
C.1 Vordefinierte Attribute für Regeln in ANTLR . . . 128
C.2 Anwendung der in der Konfiguration angegebenen Substitutionsregeln . 130
C.3 Code zur Java-Klasse *InternalArrayList* . . . 134
C.4 Grammatikdefinition für Konfigurationsdateien . . . 136
C.5 Beispiel einer Konfigurationsdatei . . . 140
C.6 StringTemplate’s . . . 143

Abbildungsverzeichnis

1 Allgemeine Struktur eines Übersetzers 12

2 Grundstruktur einer Grammatikdefinition in ANTLR 30

3 Spezifikation der Klassenpräambel (in Anl. an [GSB05, Seite 585 ff.] . . 39

4 Überblick des Übersetzungsprozesses 68
5 Übersetzungsprozess - Vorbereitung und Initiale Phase 69
6 Klassenstruktur für interne Speicherung von Informationen über Klassen 71
7 Kommentarerhaltung 75
8 Übersetzungsprozess - Erste Zwischenphase 77
9 Verallgemeinerung des Prozesses zur Anreicherung eines ASB's mit Typinformationen 86

Tabellenverzeichnis

1 Mitarbeiterstatistik der PPI AG 1

2 Standardwerte primitiver Datentypen 50
3 Parameter für den Programmstart 67

4 Ausführungszeiten 103

5 LOC-Statistik der DTA-Toolbox 124
6 Laufzeiten der einzelnen Phasen bei der Übersetzung der DTA-Toolbox . 125
7 LOC-Statistik der SWIFT-Toolbox 126
8 Laufzeiten der einzelnen Phasen bei der Übersetzung der SWIFT-Toolbox 127

Zusammenfassung

Diese Arbeit beschreibt die Entwicklung eines Übersetzers zur Transferierung von Java-API's in äquivalenten C++-Code. Dabei steht vor allem die Erzeugung von formatierten und leicht verständlichen Zielcode im Vordergrund. Zunächst werden die bereits verfügbaren Übersetzer vorgestellt und es wird verdeutlicht, warum ein neues Werkzeug entwickelt wurde. Anschließend werden Äquivalenzen zwischen Java- und C++-Konstrukten gebildet und der Übersetzungsprozess beschrieben. Schließlich wird anhand eines Beispiels aus der Praxis die Korrekheit des entwickelten Übersetzers exemplarisch getestet.

1 Einleitung

1.1 PPI AG Informationstechnologie

Diese Diplomarbeit ist in Kooperation mit der PPI AG Informationstechnologie (PPI AG) entstanden. Die PPI AG ist ein mittelständiges Unternehmen, dessen Tätigkeiten in der Erstellung von Softwarelösungen und der Beratung von Finanzdienstleistern liegt. Im Geschäftsjahr 2008 stieg die Anzahl der gesamten für die PPI AG tätigen Mitarbeiter um ca. 26 % von 263 (Januar 2008) auf 313 (November 2008) an (vgl. Tabelle 1).

Tabelle 1: Vergleich der Mitarbeiteranzahl zwischen Januar 2008 und November 2008

	intern	extern	Azubi	Stud.	Manag.	Verw.	Vorst.	Gesamt
Jan. 2008	167	36	8	11	24	14	3	263
Nov. 2008	201	49	9	26	27	16	3	331
Differenz	+34	+13	+1	+15	+3	+2	0	+68

Die PPI AG wurde 1984 gegründet und befindet sich seitdem in einem stetigen Wachstum. Aktuell arbeitet die PPI AG an der Erschließung des Versicherungssektors um zusätzliche Kunden außerhalb der Finanzdienstleister zu gewinnen. Des Weiteren ist die PPI AG sehr aktiv im Bereich der Ausbildung neuer Arbeitskräfte, wie in Tabelle 1 anhand der Anzahl der Auszubildenen und Studenten, die von ihr beschäftigt werden, ersichtlich ist.

1.2 Problemstellung und Motivation

Die PPI AG produziert unter anderem Softwarelösungen in den Sprachen C++ und Java. Jeder ihrer Kunden hat zumeist eine präferierte Zielsprache oder ist sogar an eine bestimmte Sprache gebunden. Beispielsweise gibt es Kunden, die ihre Applikationen auf einem Host laufen lassen, auf dem keine Java Virtual Machine verfügbar ist. So kann der Fall eintreten, dass ein Kunde A ein Produkt in Java und ein Kunde B das gleiche Produkt in C++ ausgeliefert bekommen möchte. Momentan werden in so einem Fall zwei unabhängige Lösungen entwickelt, die die gleiche Funktionalität abbilden. Das bedeutet aber, dass ein Mehraufwand durch die separate Entwicklung des

Programmcodes anfällt, da zum Beispiel die Entwicklung der Programmlogik und Algorithmen oder Tests doppelt durchgeführt werden müssen. Durch die Schaffung von geteiltem Programmcode könnte dieser Mehraufwand im Optimalfall um 50 % verringert werden. Dabei könnten unter anderem folgende Wege eingeschlagen werden:

- Entwicklung einer Zwischensprache in der geteilter Programmcode entwickelt wird. Aus diesem Programmcode könnte dann entsprechend äquivalenter Java- bzw. C++-Code generiert werden.
- Entwicklung des geteilten Programmcodes in C++ und Generierung von äquivalentem Java-Code
- Entwicklung des geteilten Programmcodes in Java und Generierung von äquivalentem C++-Code

Die Schaffung einer Zwischensprache für die Generierung des Zielcodes verspricht zwar große Vorteile, ist aber auf der anderen Seite mit einem hohen initialen Aufwand verbunden. Das liegt daran, dass die Definition einer Grammatik für eine Zwischensprache eine Evaluierung der zu vereinigenden Sprachen Java und C++ vorraussetzt, deren Resultat eine Menge von Äquivalenzklassen von Sprachbestandteilen wäre. Zudem muss die Syntax der Zwischensprache einfach gehalten werden, da sonst die Gefahr bestehen würde, dass die entwickelte Sprache nicht verwendet wird. Die Gefahr, dass am Ende kein ansprechendes Resultat herauskommt, ist bei der Entwicklung einer neuen Sprache um ein Vielfaches höher als bei den anderen beiden Vorgehensweisen.

Des Weiteren ist die Wahl von Java als Ausgangsbasis der von C++ vorzuziehen, da Java eine Sprache mit kleinerem Umfang ist als C++, was die Bestimmung von Äquivalenzen vereinfacht. Außerdem wird die Sprache Java bei der PPI AG häufiger in Rahmen von Projekten eingesetzt als die Sprache C++, so dass zum Einen eine höhere Kompetenzbreite in der Programmierung in Java vorliegt. Zum Anderen existieren somit auch mehr potentielle Java-Applikationen und -Biliotheken, für die in C++ möglicherweise Äquivalente angefertigt werden müssen. Den geteilten Programmcode zunächst in Java zu entwickeln und anschließend in äquivalenten C++-Code zu transferieren, ist die am schnellsten durchzusetzende und vielversprechenste Lösung.

Aus den oben angeführten Gründen wurde die Entscheidung getroffen einen Übersetzer zu entwickeln, der Java-Code in gleichwertigen C++-Code transferiert.

1.3 Zieldefinition

Übergeordnetes Ziel dieser Arbeit ist die Entwicklung eines Übersetzers zur Transferierung von Java-API's in äquivalenten C++-Code. Dabei steht vor allem die Erzeugung von formatiertem und leicht verständlichem Zielcode im Vordergrund. Zudem soll exemplarisch die Übersetzung eines Ausschnittes einer realen Programmbibliothek der PPI AG mit dem entwickelten Übersetzer durchgeführt und evaluiert werden, so dass eine Aufwand / Nutzen-Abschätzung für den Einsatz des Übersetzers möglich ist.

Im Folgenden werden zunächst Grundlagen, die zu einem leichteren Verständnis der Arbeit beitragen, illustriert. Anschließend werden die verschiedenen bereits existierenden Übersetzungswerkzeuge für die Portierung von Java-Quelltexten in nativen Programmcode evaluiert. Danach folgt eine Übersicht über die verschiedenen Parsergeneratoren, die mit der Vorstellung des ausgewählten Parsergeneratoren ANTLR abgeschlossen wird. In Kapitel 4 werden das zu erreichende Ziel konkretisiert und die Java-Konstrukte ihren Äquivalenten in C++ zugeordnet. Schließlich werden die vorbereitenden Maßnahmen für den Übersetzungsprozess erläutert. In Kapitel 5 werden die einzelnen Phasen des Übersetzungsprozesses beschrieben. Anschließend wird anhand der Übersetzung des FinTS-Kernel der PPI AG der entwickelte Übersetzer getestet und die dabei gewonnenen Erkenntnisse beschrieben. In Kapitel 7 werden mögliche Erweiterungen des Übersetzers diskutiert. Abschließend wird die Arbeit durch ein Fazit und einen Ausblick abgerundet.

1.4 Zielgruppe

Die Arbeit richtet sich an Studierende der Informatik und Programmierer der Sprachen Java und C++. Für die Arbeit wird ein grundlegendes Verständnis für Programmiersprachen vorausgesetzt. Des Weiteren sind Kenntnisse in den Programmiersprachen Java [1] und C++ [2] für das Verständnis der Arbeit essentiell. Zudem wird ein Basiswissen über die Konzepte der Objektorientierung und des Übersetzerbaus [3] zu Grunde gelegt.

[1] Einstiegsliteratur für Java: [Fla98]
[2] Einstiegsliteratur für C++: [Str98] und [LLM06]
[3] Einstiegsliteratur für Übersetzerbau: [VLSU08]

2 Grundlagen

Dieses Kapitel behandelt Grundlagen, die zum Verstehen des Inhalts der folgenden Kapitel nützlich sind.

2.1 Definition API und Java-API

In dieser Arbeit wird ein Übersetzer für die Transferierung von Java-API's entwickelt. API steht für „*Application Programming Interface*“ (engl. für Schnittstelle zur Anwendungsprogrammierung) und definiert „*eine Schnittstelle, die von einem Softwaresystem anderen Programmen zur Verfügung gestellt wird*“ [4]. Folglich ist eine Java-API eine in der Programmiersprache Java implementierte Schnittstelle zur Anwendungsprogrammierung.

2.2 Vergleich Java ↔ C++

Für das Verstehen dieser Arbeit ist es essentiell, dass Quell- und Zielsprache, so wie ihre Unterschiede zueinander verstanden wurden sind. Dazu werden im Folgenden die Sprachen Java und C++ vorgestellt und ggf. miteinander verglichen.

2.2.1 Historie

Die Geschichte von Java und C++ ist grundlegend, da aus ihr hervorgeht, welche Verwandschaftsverhältnisse unter ihnen bestehen.

2.2.1.1 C++-Historie

Die erste offizielle und kommerzielle Version der Sprache C++ wurde 1985 veröffentlicht. Der Erfinder von C++, Bjarne Stroustrup, begann mit der Entwicklung der Sprache im Jahre 1979 im Rahmen seiner Arbeit bei AT&T. Seine Vision war eine neue Sprache zu entwickeln, die die Eigenschaften von Simula mit denen von C vereinigt. Dazu führte er unter anderem das Klassenkonzept in C++, damals noch „C with Classes“ genannt, ein. [5] Im späteren Verlauf wurde C++ um Features, wie Mehrfachvererbung, Templates, Namensräume oder Ausnahmebehandlung erweitert. Laut

[4] siehe [Wik09b]
[5] vgl. [AS93]

TIOBE Software BV ist C++ mit ca. 10,7 % derzeit an Position drei der am meisten genutzten Programmiersprachen anzufinden und rangiert damit einen Platz hinter seiner Muttersprache C, die auf ca. 15,4 % Marktanteil kommt. [6]

2.2.1.2 Java-Historie

Die Sprache Java wurde offiziell am 23. Mai 1995 von Sun Microsystems veröffentlicht. Java entstand nicht als Hauptprodukt, sondern als Nebenprodukt der Entwicklung des „*7“, einem interaktiven Display, das zur Steuerung verschiedener Unterhaltungselektronikartikel dienen sollte. Java, damals noch „Oak“ genannt, wurde extra für dieses Produkt als neue prozessor- und plattformunabhängige Programmiersprache von James Gosling entwickelt. Gosling entschloß sich dazu Java syntaktisch an C++ anzulehnen und um Java prozessor- und plattformunabhängig zu machen, Java-Programme in einer speziell entwickelten virtuellen Maschine auszuführen. Anschließend versuchte man den *7 auf dem Markt zu platzieren, was aber scheiterte. Durch diesen Rückschlag versuchte man die Technologie auf eine andere Weise zu vermarkten. Das Entwicklerteam sah das Internet als eine Chance die Technologie doch noch erfolgreich zu vermarkten. Dazu entwickelten sie den auf Java basierenden Webbrowser „WebRunner“, der als erster Browser die Darstellung animierter, beweglicher Objekte und dynamischer Inhalten ermöglichte. Anfang 1995 wurde eine Demo von WebRunner auf der „Technology, Entertainment and Design Conference“ vorgestellt, in der die neuen Möglichkeiten der Java-Technologie für die Webentwicklung anhand eines animierten Sortieralgorithmus aufgezeigt wurden. Im Mai 1995 wurde dann schließlich die erste vollständig kostenlose Java Version zum herunterladen bereitgestellt. [7] Seitdem wurde Java stets weiterentwickelt und um einige Features, wie z. B. Generics oder Annotations, erweitert. Mittlerweile ist laut TIOBE Software BV Java mit ca. 19,3 % Marktanteil (Stand: April 2009) die am meisten genutzte Programmiersprache der Welt [8], wobei festgehalten werden muss, dass der Anteil im Juni 2001 noch bei ca. 26,5 % lag. [9]

2.2.2 Einsatzgebiete

C++ ist eine imperative, objektorientierte, strukturierte und generische Programmiersprache. [10] C++ wird hauptsächlich für die Entwicklung betriebssystemnaher Software, von Programmen mit einem hohen technischen Anteil und für Anwendungen verwendet, die zeitkritische Anforderungen besitzen. [11] Aus der Domäne der Anwendungsprogrammierung wurde C++ mit dem Aufkommen der Sprachen Java und C# zum Teil zurückgedrängt. Bei der Anwendungsprogrammierung kommt C++ heute

[6] siehe [BV09a]
[7] vgl. [Ban95] und [Byo03]
[8] siehe [BV09a]
[9] siehe [BV09b]
[10] vgl. [Hen97, Seite 8 ff.]
[11] vgl. [Lew97]

vor allem dort zum Zuge, wo maximale Forderungen an die Effizienz gestellt werden, um durch technische Rahmenbedingungen vorgegebene Leistungsgrenzen möglichst gut auszunutzen. [12] Java ist eine objektorientierte Programmiersprache, die hauptsächlich im Bereich der Internetprogrammierung und der verteilter Applikationen Anwendung findet, da sie eine *sichere* und *robuste* Sprache ist. [13] Zudem wird Java verhäuft für die Implementierung der Geschäftslogik und der Präsentationsschicht von Standalone-Anwendungen eingesetzt. Diese Beobachtung ist auf die große Anzahl der frei zu Verfügung stehenden Frameworks und API's zurückzuführen. [14]

2.2.3 Mehrdeutigkeit / Schwere der Sprache

Die Schwere der Sprache C++ ist wesentlich höher als die Java's. [15,16] Diese Tatsache lässt sich beispielsweise an dem folgenden C++-Codefragment zeigen:

```
#include <iostream>
class C {
public:
    C() : x(0)
    {}
    C(int x) : x(x)
    {}
    int x;
};
int main() {
    C c1(1);   // Initialisierung von Variable c1
    C c2();    // Deklaration einer Funktion c2, keine Initialisierung
    std::cout << c1.x << std::endl; // Ok
    std::cout << c2.x << std::endl; // Verursacht Uebersetzungsfehler
    return 0;
}
```

Einem weniger erfahrenen Programmierer könnte dieser Fehler leicht unterlaufen. Die Fehlermeldung des *g++*-Compilers [17] zu dem o.a. Beispiel lautet „error: request for member 'x' in 'c1', which is of non-class type 'C ()()'" und ist auf dem ersten Blick verwirrend und nur für erfahrene Programmierer nachvollziehbar. Des Weiteren verursachen Features, wie Mehrfachvererbung oder die Abwärtskompatibilität zu C weitere defizile Probleme, wie zum Beispiel den „*Deadly Diamond of Death*" [18].

[12] vgl. [Wik09a]
[13] vgl. [Hen97, Seite 6 und 7]
[14] vgl. [Wik09a]
[15] vgl. [Hen97, Seite 9 und 10]
[16] vgl. [Fla98, Seite 6]
[17] Der *g++* ist ein Compilerwerkzeug der GNU Compiler Collection mit dem u.ä. aus C++-Programmen ausführbare Binärdateien erzeugt werden können
[18] vgl. [Mar97]

2.2.4 Sprachmerkmale im Vergleich

Abschließend werden die verschiedenen Sprachmerkmale von Java und C++ augelistet und gegebenenfalls miteinander verglichen.

2.2.4.1 Allgemeine Struktur

In Java müssen Defintionen von Methoden, Variablen, usw. innerhalb von Klassen angegeben werden, die wiederum in sogenannten Package's abgelegt werden. Es ist im Gegensatz zu C++ nicht möglich global verfügbare Variablen zu definieren. In C++ erfüllen die Namensräume die Funktion der Package's in Java. Die Java-Mainfunktion ist eine statische Funktion, während die C++-Mainfunktion eine globale Funktion ist[19]. In C++ wird nicht explizit zwischen verschiedenen Klassenkategorien unterschieden. In Java existieren verschiedene Klassenarten, auf die im Kapitel 4 näher eingegangen wird. Zudem wird in C++ zwischen zwei verschiedenen Quelldateitypen unterschieden. Zum Einen werden Klassen zumeist innerhalb von sog. *Header-Dateien* deklariert. Zum Anderen wird die Implementierung der Methoden innerhalb von sog. *Cpp-Dateien* durchgeführt. Dies ist im Allgemeinen keine Pflicht, so dass es auch gestattet ist Methoden innerhalb von Header-Dateien zu implementieren. Die Trennung von Klassenschnittstelle und der Implementierung der einzelnen Methoden ermöglicht eine Kapselung, die insbesondere bei der Entwicklung von Bibliotheken ein gute Möglichkeit zur Strukturierung darstellt. [20]

2.2.4.2 Zeiger, Referenzen, etc.

In Java existieren keine Zeiger (engl.: Pointer) im Sinne von C oder C++. [21] Wenn in Java ein Objekt mittels **new** erzeugt wird, wird immer eine Referenz auf das erzeugte Objekt zurückgeliefert. Im Gegensatz zu Referenzen in C++ müssen Referenzen in Java nicht gleichzeitig deklariert und initialisiert werden. Des Weiteren können sie im Verlauf eines Programmes neu gebunden werden. Somit stellen Java-Referenzen eine Mischung aus C++-Referenz und -Pointer dar. Da in Java alle Argumente *by-Reference* übergeben werden, ist es nicht möglich Kopierkonstruktoren wie in C++ zu definieren. Stattdessen steht jeder Klasse die Methode *clone* zur Verfügung. Durch ihren Aufruf kann eine Instanz geklont werden. Es ist jedoch notwendig diese Methode für jede Klasse, deren Instanzen geklont werden sollen, zu überschreiben, um ein korrektes Klonen garantieren zu können. [22]

Einen weiteren essentiellen Unterschied stellen die unterschiedlichen Zugriffsoperatoren dar. In C++ werden drei verschiedenen Zugriffsoperatoren unterschieden:

- '->' (Pfeiloperator)

[19] vgl. [Eck98, Seite 817]
[20] vgl. [Lew97]
[21] vgl. [Fla98, Seite 6]
[22] vgl. [Eck98, Anhang B]

- ‘.‘ (Punktoperator)
- ‘::‘ (Bereichsoperator)

Diese werden im Zusammenhang mit dem Dereferenzierungsoperator (‘*‘) und dem Addressoperator (‘&‘) verwendet.

Der Pfeiloperator wird dafür benutzt um auf Elemente über den Zeiger eines Objektes zuzugreifen. Der Punktoperator wird hingegen für den Zugriff auf Elemente über eine Referenz benutzt. Im Gegensatz zu den beiden anderen Operatoren wird der Bereichsoperator für Zugriffe auf statische Bestandteile einer Klasse und auf global definierte Funktionen und Variablen verwendet. [23]

2.2.4.3 Speicherverwaltung

Einen signifikanten Unterschied zwischen Java und C++ stellt die Speicherverwaltung der beiden Sprachen dar. In C++ ist es prinzipiell möglich für jede Instanz eines Objektes separat zu entscheiden, ob diese auf dem Stack oder auf dem Heap angelegt werden soll. Für die Erstellung von Objekten auf dem Heap dient wie in Java, der Operator **new**. Im Gegensatz zu C++ werden in Java alle Objekte auf dem Heap angelegt. Die Destruierung der angelegten Objekte übernimmt der Garbage Collector der JVM. In C++ muss die Freigabe des reservierten Speichers von Objekten, die auf dem Heap angelegt wurden, mit Unterstützung des Operators **delete** und selbstgeschriebener Destruktoren bewerkstelligt werden. [24] Diese als „*herkömmlich*“ bezeichnete Speicherverwaltung kann zu Speicherlecks und Hängenden Zeigern führen. Ein Speicherleck tritt auf, wenn am Ende des Lebenszyklus eines Objekt's vergessen wurde dieses zu destruieren und der verwendete Speicher nicht freigegeben wird. Hängende Zeiger, sind Zeiger auf bereits freigegebenen Speicher. Diese Probleme können durch die Verwendung existierender Garbage Collectoren für C++, die jedoch nicht auf allen Plattformen verfügbar sind, minimiert werden. [25] Des Weiteren können sogenannte Smartpointer verwendet werden, die die Destruierung von Objekten auf dem Heap automatisieren. [26] In Java besteht die Möglichkeit von Speicherlecks und Hängenden Zeigern ebenfalls, aber deren Auftrittwahrscheinlichkeit wird durch die stetige Weiterentwicklung des Garbage Collection Mechanismus der JVM immer geringer.

2.2.4.4 Generische Programmierung

Beide Sprachen unterstützen generische Programmierung. In C++ werden die generischen Bestandteile *Templates*, in Java *Generics* genannt. Die Syntax beider ist verwandt jedoch nicht gleichwertig. Sowohl in Java als auch in C++ können Funktionen und Klassen generisch „*gestaltet*“ werden. In C++ müssen Klassen- und Funktionstemplates mit dem Schlüsselwort **template** gekennzeichnet werden. In Java ist hingegen kein

[23] in Anlehnung an [LLM06]
[24] vgl. [Str98]
[25] vgl. [Bau05]
[26] vgl. [Har97]

spezielles Schlüsselwort für die Kennzeichnung generischer Klassen und Funktionen notwendig. Während in C++ beliebige Typparameter verwendet werden können, ist es in Java nicht gestattet primitive Datentypen als Typparameter zu verweden. Java unterstützt sowohl Wildcards (**?**) als auch Typebounding (**extends**, **super**). C++ ermöglicht dagegen keine direkte Angabe dieser Features. Ein weiterer Unterschied liegt darin, dass bei der Definition einer C++-Templateklasse für jeden Typparameter eine separate Klasse zur Übersetzungszeit erzeugt wird. Diese Eigenschaft führt dazu, dass statische Variablen in Templateklassen nur in Instanzen, die mit gleichen Typparametern initialisiert wurden, geteilt werden. Zusätzlich wird für Templateklassen, die in einem Programm mit vielen unterschiedlichen Typparametern initialisiert werden, jeweils eine eigenständige Klasse generiert und dadurch besteht die Gefahr der Codeexplosion. In Java existiert nur eine Version jeder generischen Klasse, die von allen gemeinsam benutzt wird. Des Weiteren sind Templates im Gegensatz zu Generics turing-vollständig. [27][28]

2.2.4.5 Sonstiges

C++ unterstützt die Verwendung eines Präprozessors, der unter anderem für das Einbinden von anderen Klassen, die Verhinderung von mehrfacher Inklusion oder die „*conditional compilation*“ (engl. für fallabhängige Übersetzung) benutzt werden kann. [29]

Die primitiven Datentypen von Java und C++ unterscheiden sich dahingehend, dass die Größe der Datentypen in C++ maschinenabhängig ist [30], während in Java garantiert ist, dass diese auf allen Systemen gleich groß sind. Zudem ist es in C++ erlaubt einer Variablen vom Typ **bool** (entspricht dem Java-Typen **boolean**) einen numerischen Wert zuzuweisen. [31] In Java ist hingegen nur die Zuweisung von **true** und **false** gestattet. [32] Des Weiteren besitzen alle primitiven Typen in Java mit Ausnahme von **boolean** und **char** immer ein Vorzeichen. [33]

Java unterstützt die Verwendung von Default-Argumenten nicht. [34] In C++ ist dagegen eine Definition wie `void add(int a, int b, int c=0){...}` erlaubt. In diesem Fall ist die Angabe des dritten Parameters optional, so dass sowohl `add(1,2)` als auch `add(1,2,3)` einen korrekten Aufruf der Funktion `add` darstellen. Der Grund dafür ist, dass durch `int c=0`, dem dritten Parameter ein Defaultwert von 0 zugewiesen wird, falls dieser bei einem Methodenaufruf nicht spezifiziert wurde. [35]

In Java gibt es keine Möglichkeit Objektinhalte als konstant zu markieren. Mit dem Schlüsselwort **final** ist es lediglich möglich eine erneute Zuweisung zu verhinderen. Als

[27] vgl. [Bra04]
[28] vgl. [Str98]
[29] vgl. [Lew97, Seite 14-17]
[30] vgl. [LLM06, Seite 58]
[31] vgl. [Str98, Seite 77]
[32] vgl. [Hen97, Seite 49]
[33] vgl. [Fla98, Seite 26]
[34] vgl. [GSB05]
[35] vgl. [Str98, Seite 164]

Folge verhalten sich als **final** deklarierte Variablen primitiven Datentyps wie konstant deklarierte Variablen in C++. In C++ ist es hingegen möglich zwischen konstanten Zeigern und konstanten Objektinhalten zu unterscheiden. [36]

Inline-Ersetzung ist eine Methode zur Steigerung der Ausführungsgeschwindigkeit, indem Methdenaufrufe durch den Code der aufzurufenden Methode ersetzt werden. [37] In Java gibt es keine Möglichkeit den Compiler zu Inline-Ersetzung zu zwingen, wie es etwa in C++ möglich ist (durch Verwendung von **inline**). Es ist nur möglich durch die Deklaration einer Methode als **final**, den Compiler die Inlineersetzung nahezulegen. [38]

Insgesamt bietet Java weniger potentielle Fehlerquellen als C++. Dafür ist C++ systemnaher und ermöglicht direkten Zugriff auf den Hauptspeicher. Es bleibt festzuhalten, dass Java und C++ zwei syntaktisch ähnliche Sprachen sind, die bei genauer Betrachtung aber große Unterschiede aufzeigen. Auf weitere Unterschiede der beiden Sprachen Java und C++ wird in den Kapiteln 4, 5 und 7 eingegangen.

[36] vgl. [Str98, Seite 102-105]
[37] vgl. [Bre99, Seite 147]
[38] vgl. [Eck98, Anhang B]

2.3 Grundlagen zum Übersetzerbau

Im folgenden Abschnitt werden die Grundlagen des Übersetzerbau's, die zum allgemeinen Verständnis der verschiedenen Problemstellungen in dieser Arbeit und deren Lösungen, sowie im Speziellen für das Nachvollziehen der Evaluation im Abschnitt 3, notwendig sind, erläutert.

2.3.1 Allgemeiner Aufbau

Ein Übersetzer ist ein Programm, das „*ein Programm in einer Sprache (Quellsprache) lesen und in ein gleichwertiges Programm einer anderen Sprache (Zielsprache) übersetzen kann*“.[39] Ein Interpreter erstellt hingegen kein Zielprogramm, sondern führt die im Quellprogramm vorgegebenen Operationen (fast) direkt an den Benutzereingaben aus. In dieser Arbeit wird ein Übersetzer und kein Interpreter entwickelt. Der allgemeine Aufbau eines Übersetzers ist in Abbildung 1 dargestellt.

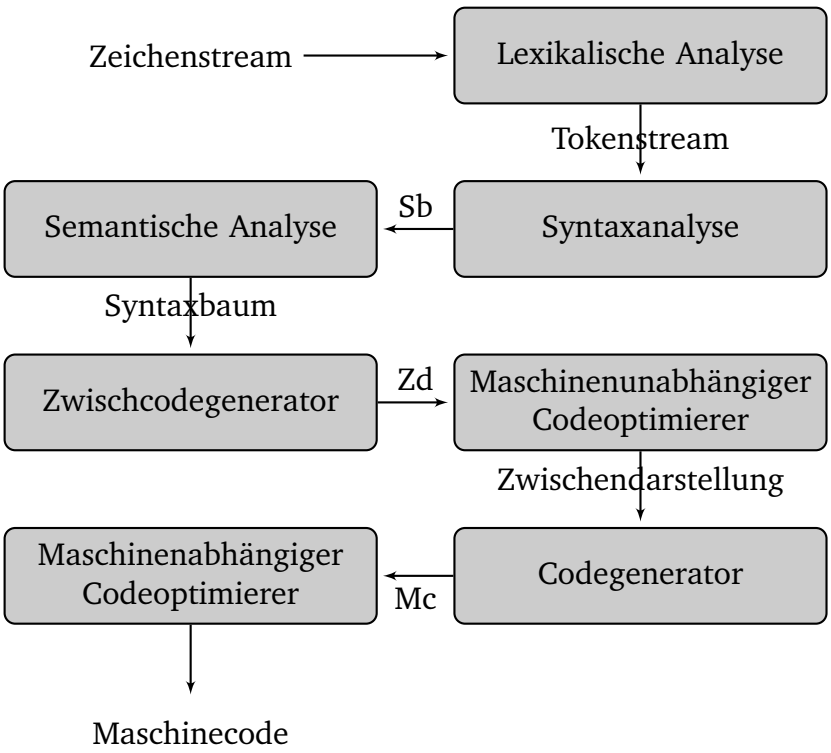


Abbildung 1: Allgemeine Struktur eines Übersetzers (Zd=Zwischendarstellung, Sb=Syntaxbaum, Mc=Maschinencode) (in Anlehnung an [VLSU08, Seite 6])

[39] siehe [VLSU08, Seite 3]

In der lexikalischen Analyse wird eine Zeichenkette (z. B. in Form einer Quelldatei) in eine Folge von Kurzzeichen (engl.: token) umgewandelt. Auf Basis dieser Folge von Kurzzeichen wird dann die Syntaxanalyse durchgeführt. Dabei wird überprüft, ob die Kurzzeichenfolge einem gültigen Ausdruck der Grammatik der Quellsprache entspricht. Falls die Kurzzeichenfolge ungültig ist, wird eine entsprechende Fehlermeldung produziert und der Vorgang entsprechend ab- oder unterbrochen. Falls sie einen gültigen Ausdruck in der Grammatik repräsentiert, kann ein Abstrakter Syntaxbaum (kurz: ASB) erstellt werden, der als Basis für semantische Analysen dient. Eine semantische Analyse stellt eine über die Möglichkeiten der syntaktischen Analyse hinausgehende Überprüfung von Bedingungen an den Quelltext dar. Nach dem Abschluss der semantischen Analyse folgt die Erzeugung von relativ maschinennahen Zwischencode, auf dessen Basis architekturunabhängige Optimierungen vorgenommen werden können. Dieser Teil des Übersetzungsvorganges ist optional. Anschließend wird entweder aus dem optimierten Zwischencode oder der aus dem aus der semantischen Analyse resultierenden ASB der Zielcode generiert. Ggf. kann im Anschluss an diese Phase eine weitere optionale Codeoptimierung auf Basis des Zielcodes stattfinden. Diese ist dann in vielen Fällen architektur- bzw. maschinenabhängig. Im Folgenden wird detaillierter auf die fünf Hauptfunktionalitäten

- Umwandlung einer Zeichenkette in Kurzzeichenkette (Lexikalische Analyse)
- Überprüfung auf syntaktische Korrekheit (Syntaxanalyse)
- Generierung von ASB's (Syntaxanalyse)
- semantische Korrektheitsüberprüfungen (Semantische Analyse)
- Zielcodegenerierung (Codegenerierung)

eines Übersetzungsvorganges eingegangen. [40]

2.3.2 Umwandlung einer Zeichenkette in eine Kurzzeichenkette

Das Resultat der lexikalischen Analyse stellt die Grundlage für die Syntaxanalyse dar. Die Aufgabe eines Lexers [41] besteht darin einzelne Zeichen des Eingabestroms zu Kurzzeichen zusammenzufassen.

Zum Beispiel wird aus der Zeichenfolge *'1','+','2','3','4','5','+','6','7','8','9'* bei einer Grammatik, die die einfache Addition von natürlichen Zahlen erlaubt, die Kurzzeichenfolge *'1','+','2345','+','6789'* gebildet. Eine weitere Aufgabe des Lexers ist es spezielle Zeichen und Zeichenfolgen, wie z. B. Kommentare oder Leerzeichen, aus dem Eingabestrom herauszufiltern und nicht mit in die resultierende Kurzzeichenfolge zu übernehmen. [42] Dieser Vorgang wird als Screening bezeichnet.

[40] vgl. [VLSU08]
[41] Bezeichnung für ein Werkzeug, dass die lexikalische Analyse durchführt
[42] vgl. [VLSU08, Seite 96]

2.3.3 Überprüfung auf syntaktische Korrektheit

Nachdem der Lexer den Eingabestrom in eine Folge von Kurzzeichen umgewandelt hat, wird diese auf ihre syntaktische Korrektheit überprüft. Informell bedeutet dies, dass versucht wird die Folge von Kurzzeichen auf grammatische Regeln abzubilden. Dieses wird zusammengefasst unter dem Begriff eines Parsers. Dabei wird zwischen zwei verschiedenen Vorgehensweisen, dem Top-Down-Parsing und dem Bottom-Up-Parsing, unterschieden.

2.3.3.1 Top-Down-Parsing

Das Top-Down-Parsing kann als das Problem verstanden werden, „*einen Parse-Baum für den Eingabestring ausgehend von der Wurzel zu konstruieren und die Knoten des Baumes in Präorder-Reihenfolge anzulegen*“. [43] Die größte Klasse von Grammatiken für die sich Top-Down-Parser konstruieren lassen bezeichnen die LL-Grammatiken. Ein LL-Parser liest den Eingabestrom von links nach rechts und versucht eine Linksableitung zu bestimmen. Das LL(k)-Parsing stellt eine spezielle Form des LL-Parsing dar, wobei k für die Größe des Lookaheads [44] steht, dass heißt, dass zur Bestimmung einer Linksableitung die nächsten k Zeichen mit einbezogen werden. Eine Sonderform des LL(k)-Parsing stellt das LL(*)-Parsing dar, dass beliebigen Lookahead ermöglicht, welches vereinfachende Definitionen von Grammatikregeln ermöglicht. [45]

2.3.3.2 Bottom-Up-Parsing

Im Gegensatz zum Top-Down-Parsing wird das Bottom-Up-Parsing als Versuch einen Parse-Baum von unten nach oben (engl.: bottom-up) zu erstellen verstanden. Dabei wird versucht die Blätter (die vom Lexer gelieferten Kurzzeichen) nach und nach zusammenzufügen bis schließlich der Wurzelknoten (Startregel der Grammatik) erreicht wurden ist. [46] Der verbreiteste Bottom-Up-Analysestil ist das sogenannte Shift-Reduce-Parsing. Dabei werden entweder Kurzzeichen aus dem vom Lexer erzeugten Strom auf einen Stack eingefügt oder aber auf dem Stack vorhandene Symbole mit Hilfe einer Grammatikregel zu einem neuen Symbol reduziert. Das Reduzieren kann als das Zusammenfassen zweier Äste eines Parse-Baum zu einem neuen Teilbaum verstanden werden und das Ablegen von Kurzzeichen auf dem Stack als das Einfügen von neuen Blätter in den Parse-Baum. Ziel des Shift-Reduce-Parsing ist es den gesamten Kurzzeichenstrom zu einem als Startsymbol der zugrunde liegenden Grammatik deklarierten Symbol zu reduzieren. Die größte Klasse von Grammatiken für die sich Shift-Reduce-Parser konstruieren lassen, sind die sogenannten LR-Grammatiken. Die Parser für diese Grammatiken werden als LR-Parser bezeichnet. Ein LR-Parser liest den Eingabestrom

[43] siehe [VLSU08, Seite 263]
[44] Anzahl der Zeichen, die vorrausgesehen wird
[45] vgl. [VLSU08, Seite 243]
[46] vgl. [VLSU08]

von links nach rechts und versucht eine Rechtsableitung zu bestimmen.[47] Die LR-Parser werden dabei noch in weitere Klassen unterteilt:

- reine LR-Parser
- SLR-Parser
- LALR-Parser
- GLR-Parser

2.3.3.3 Zusammenfassung LR- und LL-Parsing

Betrachtet man die beiden wichtigsten Parsertypen, LL-Parser und LR-Parser, lassen sich folgende Schlüsse ziehen:

- LR-Parser können eine größere Menge von Grammatiken parsen als LL-Parser
- Fehlerbehandlung in einem LR-Parser gestaltet sich schwieriger als in einem LL-Parser
- die Verwendung der LL(*)-Parser ist dem der LL(k)-Parser vorzuziehen

2.3.4 Generierung von ASB's

Die Generierung von ASB's stellt einen wichtigen Bestandteil des Übersetzungsvorganges dar, da ein ASB eine tiefergehende Analyse ermöglicht. Bei der Generierung eines ASB's ist es wichtig bei der Abbildung von Operationen darauf zu achten, dass die Auswertungsreihenfolge mit in die Struktur des ASB's einfließt. Das führt beispielsweise dazu, dass in einem ASB für ein Java-Programm der Ausdruck `a + b / c` implizit zu `(+ a (/ b c))` umgewandelt wird, um im ASB die Auswertungsreihenfolge des Ausdrucks abzubilden. Durch das Screening während der lexikalischen Analyse werden Kommentare und andere für die Semantik eines Programmes unwichtige Bestandteile herausgefiltert und fließen somit nicht in den ASB mit ein.

2.3.5 Semantische Korrektheitsüberprüfungen

Die Aufgabe der semantischen Analyse ist es Korrektheitsüberprüfungen durchzuführen, die während der Syntaxanalyse nicht möglich waren. Des Weiteren werden während der semantischen Analyse Typinformationen gesammelt, die im Syntaxbaum oder in einer Symboltabelle abgelegt werden.[48] Dies ist notwendig, da es z. B. möglich ist, dass ein Programm syntaktisch aber nicht semantisch korrekt ist. In vielen Sprachen ist es z. B. notwendig eine Variable vor ihrer Verwendung zu deklarieren. Dies kann nur mittels einer semantischen Analyse überprüft werden.

[47] vgl. [VLSU08, Seite 243]
[48] vgl. [VLSU08, Seite 9]

2.3.6 Zielcodegenerierung

Der Abschluss eines Übersetzungsvorganges ist entweder eine Fehlermeldung, die den Benutzer über syntaktische oder semantische Fehler im Quellcode informiert, oder aber die Erzeugung von Zielcode. Dabei wird der Zwischencode bzw. der ASB mit Hilfe von Regeln in den Zielcode transformiert.[49] Als Unterstützung für diesen Vorgang können Template-Engines, wie z. B. StringTemplate[50], verwendet werden, die eine formatierte Ausgabe vereinfachen oder eine leichtere Ersetzung der Zielsprache ermöglichen.

[49] vgl. [VLSU08, Seite 13]
[50] Webseite: `http://www.stringtemplate.org/about.html`

3 Evaluierung

Das Ziel dieser Arbeit ist es einen Java-nach-C++-Übersetzer zu entwickeln. Warum es überhaupt notwendig ist einen neuen Ansatz zu verfolgen, wird im folgenden Abschnitt näher erläutert. Dazu werden die verschiedenen bereits existierenden Übersetzer vorgestellt. Anschließend an diesem Abschnitt wird die Findung des für die Konstruktion des Übersetzers verwendeten Parsergenerators dargelegt.

3.1 Evaluierung existierender Java-nach-C(++)-Übersetzer

Im folgenden Abschnitt wird ein Überblick über Werkzeuge zum Transformieren von Java- in C++-Quellcode gegeben. Die existierenden Werkzeuge lassen sich in drei unterschiedliche Kategorien unterteilen:

- Java nach C / C++
- Java-Bytecode nach C / C++
- Sonstige

Bevor die Werkzeuge aus den verschiedenen Kategorien vorgestellt werden, erfolgt zunächst die Definition eines Profils, dass die Anforderungen an einen Übersetzer von Java-API's festhält.

3.1.1 Anforderungsprofil

Ein Werkzeug zur Übersetzung von Java-API's muss folgenden Anforderungen genügen:

- es muss quelloffen verfügbar sein, um gegebenenfalls Anpassungen durchführen zu können
- es sollte sich um ein Werkzeug handeln, dass mindestens Java 1.5 unterstützt
- falls nur eine ältere Java-Version unterstützt wird, sollte eine Portierung auf eine neuere Version erfolgen können
- das Werkzeug sollte im besten Fall keine oder wenig Einschränkungen bezüglich der Zielplattform des generierten Quellcodes aufweisen

- die Ausgabe bzw. das Aussehen des Generats sollte beeinflussbar sein (beispielsweise durch Dateien, die das Ausgabeformat spezifieren)
- das Generat muss vom Programmierer dem Originalcode zugeordnet werden können (Lesbarkeit und Debugging)
- es muss prinzipiell eine Erhaltung von Kommentaren möglich sein

3.1.2 Werkzeuge - Java nach C / C++

I3J2C ist ein kommerzielles Werkzeug der Firma CoreIntent zur Konvertierung von in Java geschriebenen Programmen für eingebettete Systeme, wie Mobiltelefone oder PDA's, in äquivalente C-Programme. Das Werkzeug basiert auf der ebenfalls von CoreIntent entwickelten Technologie der „Intentional Compilation", die garantieren soll, dass bei der Übersetzung von einem in einer Sprache *A* geschriebenen Programms in eine Sprache *B*, der Stil des Zielcodes einem handgeschriebenen Programm entspricht. Somit soll die Intention des Programmes erkenntlich bleiben und Anpassungen an dem generierten C-Code leicht durchführbar sein. Um diesem Ziel möglichst nahe zu kommen, versucht I3J2C Namen für Variablen unverändert zu lassen. Betrachtet man hingegen die Übersetzung von Methoden, wird deutlich, dass der Zielcode bei überladenen Funktionen das Lesen des Programms erschwert. Dieses Problem lässt sich darauf zurückführen, dass C keine objektorientierte Programmiersprache ist. Um das Problem hervorzuheben, betrachten wir ein Beispiel für die Übersetzung eines Java-Programmes mit Hilfe von I3J2C in ein äquivalentes C-Programm.

Beispiel 1 Übersetzung einer Java-Klasse in ein äquvalentes C-Code-Fragement mit I3J2C (in Anlehnung an [Cor08, Seite 7 ff.]

Zu übersetzende Java-Klasse:

```
public class Integer {
    private int intValue;

    public void add(double dVal) {
        ...
    }
    public void add(int dVal) {
        ...
    }
    public void add(float dVal) {
        ...
    }
    public int derivePythagoras(int a, int b, int c) {
        ...
    }
}
```

Ergebnis des Übersetzungsvorgang unter der Verwendung von I3J2C (nur Header-Datei abgebildet):

```
typedef struct Integer {int intValue;} Integer;

void Integer_add_double(double dVal);
void Integer_add_int(int dVal);
void Integer_add_float(float dVal);
int  Integer_derivePythagoras_int_int_int(int a, int b, int c);
```

Anhand von Beispiel 1 wird deutlich, dass alle Funktionen in einen gemeinsamen glo-

balen Namensraum gelegt werden und somit einzigartige Namen haben müssen. Das führt dazu, dass mehrfach überladene Java-Funktionen wie add neue Namen zugewiesen bekommen, die aus dem Klassennamen, dem Funktionsnamen und den Parametertypen der Funktion generiert sind, um die Korrektheit des generierten Programmes zu erhalten. Diese Eigenschaft führt dazu, dass der Name einer Funktion in dem generierten C-Programm unter Umständen sehr lang und somit das Programm im Gegensatz zum Java-Quelltext schwer verständlich sein kann. So wird beispielsweise aus der Java-Funktion derivePythagoras der Klasse Integer eine C-Funktion mit dem Namen Integer_derivePythagoras_int_int_int. Für spätere Versionen der Software ist eine Erhaltung von Kommentaren geplant. Die aktuelle Version unterstützt Java 1.5 mit dem Fokus auf der Java Micro Edition[51]. Laut Aussage von Navin Sinha, einem Mitarbeiter von CoreIntent, wird der Schwerpunkt der Weiterentwicklung von I3J2C zunächst weiterhin im Bereich der eingebetteten Systeme liegen. [52]

JFE ist ein kommerzielles Werkzeug der Edison Design Group, dass Java Quellcode in C, C++ oder Java-Bytecode konvertiert. Es unterstützt den vollen Funktionsumfang von Java 1.5. Es kann laut Hersteller zudem als Compiler-Frontend eingesetzt werden um den ASB-ähnlichen Zwischencode in Maschinencode oder sonstige Zielformate zu transferieren. Des Weiteren ist weder für die Benutzung von JFE noch für die Ausführung der übersetzten Java-Programme eine JVM erforderlich. Der Hersteller deklariert den generierten Code als „*nicht wartbar*“. Zudem wird vom Hersteller keine Garantie gegeben, dass der erzeugte Zielcode kompatibel zu anderem manuell erzeugten C++-Code ist. Des Weiteren sind die Anschaffungskosten, die je nach Funktionsumfang zwischen 40000 und 250000 US-Dollar liegen, relativ hoch. [53]

XML Encoded Source [54] (XES) stellt unter anderem einen Übersetzer für die Transformation von Java-Quellcode in C++-Code zur Verfügung. Es basiert auf der Idee Java-Quellcode mit Hilfe von XSLT in andere Programmiersprachen wie C++ zu übersetzen. Die veröffenlichten Versionen erzeugen nicht korrekten Quellcode, weshalb von einer weiteren Betrachtung abgesehen wird.

Java2C ist ein quelloffener Übersetzer, der Java- in C-Code konvertiert. Die Software ist unter LGPL[55] veröffentlicht und befindet sich im alpha-Zustand. Das Ziel von Java2C ist die Entwicklung eines Übersetzers mit dem Fokus auf eingebettete Systeme. Das Werkzeug ist in Java implementiert. Die aktuell unterstützten Java-Features sind auf der Webseite des Übersetzers [56] aufgelistet. Die gewählte Zielsprache C erschwert unter anderem die Übersetzung der Ausnahmebehandlung oder von Feldern. Beispielsweise übersetzt Java2c einen try-catch-Block in einen aus C-Macros bestehenden Code (vgl. Beispiel 2), die intern die Funktionen longjmp und setjmp des Standard-C-Headers „*setjmp.h*“ verwenden.

[51] Webseite JavaME: `http://java.sun.com/javame/index.jsp`, zugegriffen am 30.04.2009
[52] siehe [Cor08]
[53] Quelle: `http://www.edg.com`, zugegriffen am 03.05.2009
[54] Webseite XES: `http://sourceforge.net/projects/xes/`, zugegriffen am 03.05.2009
[55] LGPL steht für GNU Lesser General Public License
[56] Webseite: `http://www.vishia.org/Java2C/html/features.html#chapter_1`, zugegriffen am 30.04.2009

Beispiel 2 Übersetzung eines Java-Try-Catch-Blockes in C mittels Macroprogrammierung (Eine Definition der verschiedenen Macros, sowie eine detaillierte Diskussion der Problematik ist unter auf der Webseite `http://www.vishia.org/Jc/html/Exception_Jc.html#chapter_4.2` zu finden

```
...
STACKTRC_ENTRY("name");
TRY{...}_TRY
CATCH(...){...}
FINALLY{...}
END_TRY
...
```

Die Entwicklung des Werkzeuges erscheint vielversprechend für das Anwendungsgebiet der eingebetteten Systeme. Da Java2C derzeit nur von einem Programmierer entwickelt wird, ist die Frage nach dem Zeitpunkt der Veröffentlichung einer Version, die alle Java 1.5 Features unterstützt, nicht zu beantworten.

3.1.3 Werkzeuge - Java-Bytecode nach C / C++

Java2C ist ein vollständig in Java implementierter Übersetzer, der Java Bytecode als Eingabe akzeptiert und C-Code als Ausgabe produziert.[57] Der Übersetzer ist nicht mit dem bereits vorgestellten gleichnamigen Java-nach-C-Übersetzter verwechseln. Das Ziel des Übersetzungsvorganges ist es möglichst effizienten C-Code zu erzeugen, da die Anwendungsdomäne die der eingebetteten Systeme ist. Um die Anforderungen von eingebetteten Systemen zu erfüllen, benutzt Java2C zur Analyse und Optimierung das Framework Soot[58]. Mit der Unterstützung von Soot wird der Bytecode zunächst in Jimple, einer Zwischenrepräsentation in 3-Addresscode-Form, tranformiert. Der Zwischencode wird dann mit Hilfe von Soot optimiert und anschließend in C-Code überführt. Die Arbeitsweise des Übersetzers hat zur Folge, dass der C-Code aufgrund der Lowlevel-Darstellung schwer wartbar und modifizierbar ist. Auf der anderen Seite entsteht durch die Analyse ein sehr schlanker Code, da nur die zum Ablauf des Programms notwendigen Klassen übersetzt werden. So erzeugt Java2C bis zu 70 mal weniger Code als der gcj.[59] Zur Speicherbereinigung wird der Boehm-Demers-Weiser GC eingesetzt.

Toba[60] übersetzt Java-Bytecode in C-Code.[61] Der Grund für die Entwicklung von Toba war das schlechte Laufzeitverhalten von Java-Programmen, die mit Hilfe der Konvertierung nach C behoben werden sollten, um Java für Echtzeitanwendungen interessanter zu gestalten. Der mit Hilfe von Toba generierte C-Code ist äußerst schwer

[57] siehe [VB04]
[58] siehe [VRCG+99]
[59] vgl. [VB04, Seite 6, Tabelle 4]
[60] Toba wurde nach einem See auf der Insel Sumatra westlich von Java benannt
[61] siehe [PTB+97]

wartbar (vgl. Abbildung 3), da wie bei I3J2C alle Funktionen in einem globalen Namensraum abgelegt werden und der neue Name der Funktion, anders als bei I3J2C, aus dem alten Namen der Funktion und einem zufälligen Hashwert zusammengesetzt wird. Des Weiteren wird C-Code generiert, der maschinenah ist, was zur Folge hat, dass die Größe des generierten C-Code's um ein Vielfaches größer ist als der Java-Quellcode (vgl. Abbildung 3). Die letzte veröffentlichte Version ist aus dem Jahre 1999

Beispiel 3 Übersetzung einer Java-Klasse in ein äquvalentes C-Code-Fragement mit Toba (vgl. [PTB+97, Seite 6])

Zu übersetzende Java-Klasse:

```
class d {
    static int div(int i, int j) {
        i = i / j;
        return i;
    }
}
```

Ergebnis des Übersetzungsvorgang unter der Verwendung von Toba:

```
Int div_ii_3WIen(Int p1, Int p2) {
    Int i0, i1, i2;
    Int iv0, iv1;

    iv0 = p1;
    iv1 = p2;

L0: i1 = iv0;
    i2 = iv1;
    if (!i2)
        throwDivisionByZeroException();
    i1 = i1 / i2;
    iv0 = i1;
    i1 = iv0;
    return i1;
}
```

und unterstützt Java 1.1. Ferner werden Kommentare aus dem Quellcode nicht erhalten, da diese im Java-Bytecode nicht mehr vorliegen. Zur Speicherverwaltung wird der automatische Garbage-Collector (GC) von Boehm-Demers-Weiser [62] eingesetzt, der auf vielen Plattformen zur Verfügung steht, aber eine vorherige Installation erfordert.

Harissa ist ein Übersetzer zur Konvertierung von Java Bytecode in C-Code, der 1997 als Gegenstück zu javac und anderen Just-in-time-Compiler entwickelt wurde. [63] Der Übersetzer führt zum Einen aggressive Optimierungen, wie die Elimination von virtuellen Methodenaufrufen mittels der *Class Hierarchy Analysis* [64], durch. Zum Anderen

[62] Webseite: `http://www.hpl.hp.com/personal/Hans_Boehm/gc/`, zugegriffen am 30.04.2009
[63] siehe [MMBC97]
[64] siehe [DGC95b]

ist es möglich mit dem in die Laufzeitbibliotek integrierten Interpreter, Klassen zur Laufzeit dynamisch zu laden. Zur Umsetzung dieser Funktionalität wird zusätzlich eine Java Virtual Machine (JVM) benötigt. In Abbildung 4 ist ein Übersetzungsbeispiel dargestellt, das veranschaulicht, dass der Zielcode ein niedriges Sprachlevel besitzt. Zudem werden Variablennamen nicht beibehalten und die Intention des Programms ist insgesamt schwerer nachzuvollziehen als es beim Originalcode der Fall ist. Harissa erlaubt sowohl die Verwendung des Boehm-Demers-Weiser GC, als auch die Benutzung von malloc ohne Speicherbereinigung. Die letzte veröffentlichte Version von Harissa unterstützt Java 1.02.

Beispiel 4 Übersetzung einer Java-Funktion in ein äquvalentes C-Code-Fragement mit Harissa (vgl. [MMBC97, Seite 11])

Zu übersetzende Java-Funktion:

```
static int P(int a, int b) {
    int i, r;
    r = 1;
    for(i=0; i<b;i++){
        r = r * a;
    }
    return r;
}
```

Ergebnis des Übersetzungsvorgang unter der Verwendung von Harissa:

```
TINT P(TINT vi0, TINT vi1) {
    TINT si1,si0;
    TINT vi2,vi3;
    si0=1;
    vi3=si0;
    si0=0;
    vi2=si0;
    goto L14;
L7:
    si0=vi3;
    si1=vi0;
    si0*=si1;
    vi3=si0;
    vi2+=1;
L14:
    si0=vi2;
    si1=vi1;
    if(si0<si1) goto L7;
    si0=vi3;
    return si0;
}
```

TurboJ konvertiert Java-Bytecode in nativen C-Code. [65] Dabei wird der Java-Originalcode

[65] siehe [WFD+98]

durch native Methoden mit dem C-Code verknüpft, so dass der generierte Code weiterhin die Installation einer JVM voraussetzt. Somit können das Speicher- und Threadmanagement von Java weiterhin ausgenutzt werden, wodurch die Verwendung eines alternativen GC's nicht notwendig ist. Zudem können weiterhin andere Java-Klassen und -Bibliotheken verwendet werden. Der C-Code wird von TurboJ beim Übersetzungsvorgang optimiert, was zu einem effizienteren Zielcode führt. Die Optimierung und die Wahl von Bytecode als Ausgangsbasis des Übersetzungsvorganges haben zur Folge, dass der generierte C-Code schwer wartbar und modifizierbar ist. Die Entwicklung von TurboJ wurde, wie bei Harissa und Toba, mittlerweile eingestellt.

JC ist eine quelloffene JVM , die es ermöglicht Java-Bytecode in C-Code zu transferieren. Dazu benutzt es unter anderem das Soot-Framework. Der Fokus von JC liegt dabei nicht auf der Erzeugung performanten Zielcodes, sondern viel mehr in der Generierung von korrekten Programmen, wodurch auftretende Fehler leichter aufzuspüren sind. Eine vollständige Liste der Features von JC ist auf der Webseite des Werkzeuges [66] hinterlegt. Der generierte C-Code ist nicht für Manipulationen geeignet. Die letzte Version wurde im November 2005 veröffentlicht und unterstützt weder die Verwendung generischer Klassen noch die von Enumerations. Aufgrund der seit dreieinhalb Jahren ausbleibenden Updates ist davon auszugehen, dass keine weiteren Aktualisierungen von JC folgen werden.

3.1.4 Sonstige

Es existieren noch andere Wekzeuge, die nativen Quellcode erzeugen.

Eines der bekanntesten ist der gcj, ein ahead-of-time Java-Compiler der GNU Compiler Collection. Er ist unter GPL-Lizenz veröffentlicht und somit frei verfügbar. Der gcj kann Java-Quellcode in Java-Bytecode und Java-Bytecode in nativen Zielcode transformieren. Zudem können auch Java Archive für verschiedene Zielplattformen übersetzt werden. [67] Der native Zielcode besteht aus Assembleranweisungen. So führt die Übersetzung eines Hallo-Welt-Java-Programmes zu einem ca. 800 zeiligen Assembler-Programm. Somit ist der Zielcode für API's, die wesentlich umfangreicher als ein einfaches Hallo-Welt-Programm sind, alles andere als wartbar und leicht verständlich. Das Ziel bei der Verwendung des gcj ist auch nicht die Erzeugung von wartbarem äquivalenten Code, sondern vielmehr die direkte Erzeugung von ausführbarem Code. Der gcj unterstützt bisher (Stand 12. April 2009) nur ausgewählte Teile der Java SE API. Informationen über den aktuellen Abdeckungsgrad sind online verfügbar. [68]

Ein weiteres Werkzeug ist Excelsior JET, ein kommerzieller ahead-of-time Java-Compiler der Firma Excelsior LLC, der nativen Zielcode erzeugt. [69] Dieser ist im Gegensatz zum ursprünglichen Java-Code betriebssystemabhängig und deshalb nur bedingt

[66] Link: `http://jcvm.sourceforge.net/share/jc/doc/jc.html#Design%20features`, zugegriffen am 30.04.2009

[67] Quelle: `http://gcc.gnu.org/java/`, zugegriffen am 30.04.2009

[68] Link: `http://sab39.netreach.com/Software/Japitools/JDK-Results/46/`, zugegriffen am 12.04.2009

[69] Quelle: `http://www.excelsior-usa.com/jet.html`, zugegriffen am 30.04.2009

portierbar. Excelsior JET erwartet als Eingabe eine Menge von Class-Dateien, die mit dem Excelsior JET Optimizer für die gewählte Zielplattform optimiert werden. Das optimierte Programm benötigt für den Start die Excelsior JET Runtime, die die Java SE API, sowie den Garbage-Collector zur Verfügung stellt. Für das dynamische Laden von Klassen zur Laufzeit benutzt Excelsior JET einen interen just-in-time Übersetzer. Der Excelsior JET ist ein sehr nützliches Werkzeug zur Portierung von Java-Code in nativen Code, aber es erzeugt keinen C oder C++-Code, was sich negativ auf die Lesbarkeit des Zielcodes auswirkt. Des Weiteren ist die Installation einer Laufzeitumgebung für die Ausführung des Zielcodes zwingend.

Jcc ist eine nativer Java-Compiler, der Java-Quelltext direkt in ausführbaren nativen Code transferiert. Er wurde von Ronald Veldema im Rahmen seiner Masterarbeit im Jahre 1998 angefertigt.[70] Jcc ist nicht zu verwechseln mit JCC[71], dem in PyLucene von Apache integrierten C++-Code Generator, der C++-Schnittstellenklassen für die Verwendung von API's via JNI generiert. Als Eingabe erwartet Jcc Java-Quellcode, der zunächst in Bytecode und anschließend C-Code übersetzt wird. Beispielsweise wird die Funktion

```
int start() {
    int a = 1;
    int b = 2;
    int c = a * b;
    int d = c;
    return d;
}
```

in folgenden C-Zwichencode übersetzt:[72]

```
int method_test_start0(javaObject *self) {
    int ret_val = 0, sp = 0, call_ret_val = 0;
    TOperandStack operand[MAX_OPERAND_STACK];
    TOperandStack locals[MAX_LOCALS];
    javaObject *call_exception = 0;
    PUSH_CONSTANT(1);
    STORE_LOCAL_INT(1);
    PUSH_CONSTANT(2);
    ... // Es folgen im Orignalbeispiel 8 weitere Zeilen
    RET_INT;
return_no_exception:
    RETURN_PRIMITIVE(ret_val);
}
```

Der Aufbau des Zwischencodes ist stark an die Form von Java-Classfiles angelehnt. Eine Wartung des generierten Zwischencodes ist für größere Mengen Java-Quellcodes nicht vorstellbar und die Zuordnung des generierten Codes zu den jeweiligem Originalcode ist alles andere als intuitiv. Die Weiterentwicklung des Werkzeuges wurde mit Abschluss der Masterarbeit eingestellt.

[70] siehe [Vel98]
[71] Link: `http://lucene.apache.org/pylucene/jcc/index.html`, zugegriffen am 30.04.2009
[72] vgl. [Vel98, Seite 34-35]

Des Weiteren existiert ein Java-nach-C++ / CLI-Übersetzer der Firma Tangible Software Solutions. C++ / CLI ist eine von Microsoft entwickelte Variante der Programmiersprache C++, die den Zugriff auf die virtuelle Laufzeitumgbeung der .NET-Plattform mit Hilfe speziell darauf zugeschnittenen Spracherweiterungen ermöglicht. Derzeit werden die einzigen verfügbaren Compiler von der Firma Microsoft mit dem Produkt Visual Studio vertrieben.[73] Diese Abhängigkeit führt dazu, dass eine Übersetzung für die Zielsysteme Unix, Linux und Host momentan nicht möglich ist. Zudem benötigen die in C++ / CLI geschriebenen Programme die virtuelle Maschine der .NET-Plattform, was eine Portierung in ein Umfeld außerhalb von Windows geradezu unmöglich macht.

3.1.5 Abwägung der verschiedenen Alternativen

Keines der hier vorgestellten Übersetzungswerkzeuge unterstützt die Erhaltung von Kommentaren. Des Weiteren wurde die Weiterentwicklung der Werkzeuge Harissa, Toba, TurboJ, Jcc und XES eingestellt. Das vielversprechende Werkzeug JFE ist zu teuer in der Anschaffung und erzeugt laut Hersteller in der momentan verfügbaren Version keinen lesbaren und wartbaren C++-Code. Das Programm I3J2C verfolgt ebenfalls einen interessanten Ansatz, ist aber aufgrund der Zielsprache C, sowie der Einschränkung auf eingebette Systeme nicht den Anforderungen entsprechend. Werkzeuge wie gcj und Excelsior Jet überzeugen durch die Anzahl der mit Ihnen übersetzbaren Programme. Auf der anderen Seite sind sie an äußere Gegebenheiten, wie etwa einen GC oder eine JVM gebunden. Letztendlich besitzt keines der hier vorgestellten Übersetzungswerkzeuge die Eigenschaften, die eine schnelle Anpassung zur Erfüllung der gegebenen Anforderungen ermöglicht. Diese Tatsache hat den Entschluss, ein neues Werkzeug zu entwickeln, bekräftigt.

[73] Quelle: `http://de.wikipedia.org/wiki/C%2B%2B/CLI`

3.2 Auswahl eines Parsergenerators

Nachdem die Entscheidung getroffen wurde einen eigenen Übersetzer zu schreiben, wird im folgenden Abschnitt erläutert was ein Parsergenerator ist und warum dessen Verwendung sinnvoll ist. Des Weiteren wird die Entscheidung der Auswahl des Generators ANTLR begründet.

3.2.1 Notwendigkeit eines Parsergenerators

Ein Parsergenerator bezeichnet ein Programm, das Parser aus einer Beschreibungssprache generiert. Die Verwendung eines Parsergenerators für einen Übersetzungsprozess ist nicht in allen Fällen sinnvoll. So ist etwa das Erzeugen eines Parsers für einer Zeichenfolge, die lediglich aus einer durch Kommata separierten Liste von Strings besteht, mit Unterstützung eines Parsergenerators durchaus möglich, entspricht aber dem „*Schießen auf Spatzen mit Kanonen*“. Erst eine entsprechende Komplexität eines Übersetzungsproblem rechtfertigt den Einsatz eines Parsergenerators, da für einfachere Problemstellungen das Aufwand-Nutzen-Verhältnis bei einer manuellen Programmierung des Übersetzers besser ist. Das liegt zum Einen an dem relativ hohen Einarbeitungsaufwand, der notwendig ist um die zumeist komplexen Konfigurationsmöglichkeiten eines Parsergenerators aufgreifen zu können. Zum Anderen muss eine für den Parsergenerator konforme Grammatik der zu parsenden Sprache erstellt und getestet werden.

Die Verwendung eines Parsergenerators ist aufgrund der Komplexität der gegebenen Problemstellung unumgänglich. Der initiale Einarbeitungsaufwand wird durch spätere Einsparungen im Refactoring der entwickelten Software eliminiert. Des Weiteren ist Java eine komplexe Programmiersprache, so dass für tiefere semantische und sytaktische Untersuchungen des zu parsenden Quelltextes eine Zwischenrepräsentation sinnvoll erscheint. Deren Erzeugung in Form abstrakter Syntaxbäume wird von vielen Parsergeneratoren bereits unterstützt. Für die meisten Parsergeneratoren existiert zudem bereits eine formale Definition der Java-Syntax.

Im Folgenden werden zunächst die Anforderungen an einen Parsergeneratoren formuliert. Anschließend werden die verschiedenen Generatoren vorgestellt.

3.2.2 Anforderungen an einen Parsergenerator

Der Parsergenerator sollte folgenden Kriterien genügen:

Vereinigung von Lexer und Parser
: Parser und Lexer sollten die gleiche Syntax bzw. eine ähnliche Syntax benutzen. Zudem sollte die Syntax leicht verständlich, d. h. ähnlich der EBNF (Erweiterte Backus-Naur-Form), sein.

Unterstützung abstrakter Syntaxbäume
: Die Generierung und Verarbeitung abstrakter Syntaxbäume zur semantischen Analyse sollte möglich sein.

Generierung von Zielcode
: Das Ausgabeformat des Zielcodes sollte frei veränderbar sein. Falls der Parsergenerator keinen internen Mechanismus für die Angabe einer formatierten Ausgabe bereitstellt, sollte die Eingliederung eines externen Werkzeuges für die Ausgabe formatierten Zielcodes durchführbar sein.

Fehlerbehandlung
: Eine leicht zu erweiternde Fehlerverarbeitung sollte gegeben sein.

Dokumentation und Verbreitung
: Der Parsergenerator sollte über eine ausführliche Dokumentation verfügen. Des Weiteren sollte er einen hohen Verbreitungsgrad besitzen um bei etwaigen Fragestellungen schnelle und kompetente Antworten erhalten zu können.

Diese Anforderungen an einen Parsergenerator sind für jeden Anwender unterschiedlich. Die Wahl fiel auf den Parsergenerator ANTLR, der allen aufgelisteten Bedingungen genügt. Im folgenden Abschnitt werden zunächst die anderen zur Verfügung stehenden Parsergeneratoren, die nicht ausgewählt wurden, vorgestellt. Abschließend wird der verwendete Generator ANTLR vorgestellt.

3.2.3 Vorstellung der verschiedenen Parsergeneratoren

Nachdem die Anforderungen an den Parsergenerator formuliert wurden, werden nun die verschiedenen Parsergeneratoren vorgestellt.

JavaCC
: JavaCC steht für „*Java Compiler Compiler*“ und bezeichnet einen Lexer- und Parsergenerator in und für Java. JavaCC generiert LL(k)-Parser. Die Syntax von JavaCC ist nicht besonders leserlich, da die Grammatikdefinition mit eingebetteten Java-Codefragmenten versehen werden muss. JavaCC unterstützt Unicode und erlaubt einfaches Debugging zur Fehlerkorrektur. Die Fehlerbehandlung wird durch Einfügen von Java-Try-Catch-Blöcken realisiert. JavaCC unterstützt zudem die Erstellung von abstrakten Syntaxbäumen. Laut Sun ist JavaCC der am meisten verwendete Parsergenerator im Java-Umfeld. [74]

Coco/R
: Coco/R [75] ist ein Lexer- und Parsergenerator, der LL(1)-Parser generiert. Eine Generierung für Sprachen, die nicht LL(1) sind, ist durch manuelle Zeichenvorrausschau möglich. Die Syntaxdefinition ist an die EBNF angelehnt. Die Generierung und Verarbeitung von abstrakten Syntaxbäumen ist standardmäßig nicht integriert. [76]

[74] Quelle: `https://javacc.dev.java.net/`, zugegriffen am 01.05.2009
[75] Coco/R steht für „*compiler compiler generating recursive descent parsers*“
[76] vgl. [MK06]

CUP + JFlex

CUP ist eine LALR(1)-Parsergenerator für Java, der die Verwendung eines externen Lexergenerators, wie beispielweise JFlex, vorraussetzt. CUP integriert einen Mechanismus zur Fehlerbehandlung der von YACC adoptiert wurde. Dieser Fehlermechanismus muss explizit durch die Verwendung des reservierten Wortes **error** in den entsprechenden Regeln aktiviert werden. Die Fehlerbehandlung gestaltet sich aufgrund des Parsingverfahrens im Gegensatz zum LL-Verfahren schwierig. [77]

YACC/Bison+(f)lex

YACC[78] und Bison sind LALR(1)-Parsergeneratoren. Bison ist einer Weiterentwicklung von YACC, der als einer der ersten Parsergeneratoren Anfang der 70er Jahre entwickelt wurde. Die Generierung der Lexer muss durch Verwendung von Lexergeneratoren wie flex oder lex separat durchgeführt werden. Die erwähnten Generatoren erzeugen den Parser in Form von maschinenunabhängigen C- bzw. C++-Code. Bison unterstützt die Verarbeitung und Generierung von abstrakten Syntaxbäumen. [79] Die Fehlerbehandlung ist, wie bei der Verwendung von CUP mit JFLex, schwierig. YACC und Bison erfreuen sich großer Beliebheit unter Anwendern und verfügen über eine sehr gute Dokumentation, sowie eine aktive Community.

Sonstige

Es existieren noch weitere Generatoren wie die GLR-Parsergeneratoren Elkhound und Elsa [80] oder der LALR(1)-Generator SableCC, die der Vollständigkeit wegen nicht unerwähnt bleiben sollten.

[77] Quelle: `http://www2.in.tum.de/projects/cup/`, zugegriffen am 01.05.2009
[78] YACC steht für „*Yet Another Compiler-Compiler*“
[79] Quelle: `http://www.gnu.org/software/bison/`, zugegriffen am 01.05.2009
[80] vgl. [McP04]

3.2.4 Parsergenerator ANTLR

ANTLR[81] ist ein LL(*)-Parsergenerator. Er vereinigt die Definition von Lexer und Parser. Zudem ist die Generierung und Verarbeitung von abstrakten Syntaxbäumen vollständig in die Grammatik integriert. Der Fehlermechanismus von ANTLR kann leicht durch einen eigenen Mechanismus ausgetauscht werden und unterstützt in der Standardversion alle gängigen Fehlerkorrekturmechanismen.[82] Des Weiteren erfreut sich ANTLR in der Community großer Beliebtheit und verfügt über eine gute Dokumentation. Es existieren zudem verschiedene Zielsprachen für den generierten Parser, so dass es in ANTLR möglich ist diesen für Python, Java, C und weitere Sprachen zu generieren. Die Syntax von ANTLR ähnelt der EBNF und ist einfach gehalten. ANTLR erzeugt zum Parsen von Eingabeströmen endliche Automaten und ist der einzige der hier vorgestellten Generatoren, für den eine formatierte Ausgabe (in Form von String-Templates für die Zielsprache Java) direkt integriert wurde.[83] Im Folgenden wird die Syntax von ANLTR erläutert und die verschiedenen Funktionalitäten erklärt.

3.2.4.1 Allgemeine Struktur

Mit ANTLR können vier verschiedene Grammatiktypen definiert werden. Es kann entweder ein Parser, ein Lexer, ein Baumparser oder eine Kombination aus Lexer und Parser mit ihr definiert werden. Die Syntax ist für die vier Varianten konsistent, so dass alle eine gemeinsame Grundstruktur, die in Abbildung 2 dargestellt ist, verbindet.

grammarType **grammar** *name*;
«*optionsSpec*»
«*tokenSpec*»
«*attributeScopes*»
«*actions*»

rule1 : ... | ... | ...;
rule2 : ... | ... | ...;
...

Abbildung 2: Grundstruktur einer Grammatikdefinition in ANTLR (vgl. [Par07, Seite 75])

[81] ANTLR steht für „*ANother Tool for Language Recognition*“
[82] vgl. [Par07, Seite 246-250]
[83] vgl. [Par07]

In *optionsSpec* können verschiedene Konfigurationseinstellungen getätigt werden. Die in dieser Arbeit verwendeten Optionen sind: [84]

language
: Spezifizierung der Zielsprache des durch die Grammatik beschriebenen Parsers bzw. Lexers (in dieser Arbeit wurde stets Java als Zielsprache gewählt).

output
: Spezifikation der Ausgabeform. Es kann zwischen keiner Ausgabe, *ASB* und *template* gewählt werden. Falls AST gewählt wurde, werden abstrakte Synatxbäume als Ausgabe am Ende des Parsevorganges generiert. Wird *output=template* spezifiziert, wird die Ausgabe in Form von StringTemplates konstruiert. Andernfalls wird keine Ausgabe generiert.

backtrack und memoize
: Diese beiden Optionen sollten dann eingeschaltet werden, wenn die Grammatik Mehrdeutigkeiten besitzt oder aus anderen Gründen ein nicht determinierte Zeichenvorrausschau notwendig ist.

tokenVocab
: Spezifiziert die in der Grammatik zu verwendenen Tokentypen. Diese Einstellung ist besonders für mehrstufige Grammatiken wichtig, da in allen Stufen auf der gleichen Tokenmenge operiert werden muss.

ASTLabelType und TokenLabelType
: Spezifizierung der Basisklasse für Knoten des Syntaxbaumes und der Basisklasse für Token. Beispielweise kann eine Erweiterung der Basisklassen *CommonToken* (modelliert ein vom Lexer beim Einlesen des Eingabestroms erzeugtes Kurzzeichen) und *CommonTree* (modelliert einen Knoten in einem ASB) in einigen Fällen nützlich sein.

Auf die Konfiguration der Optionen folgt die Angabe der verschiedenen Token (*tokenSpec*). Diese Angabe ist nur notwendig, falls in den Optionen keine Spezifizierung der Tokentypen angegeben wurde. Dabei wird zwischen zwei verschiedenen Tokentypen unterschieden. Zum Einen können reele Token spezifiziert werden. Ein reelles Token wird durch *<tokenName>* = '*<tokenString>*'; definiert. Reelle Token werden vom Lexer zur Konvertierung des Eingabestroms in eine Folge von Kurzzeichen verwendet. Zum Anderen können imaginäre Token spezifiziert werden. Die Definition eines imaginären Tokens erfolgt durch *<tokenName>*;. Imaginäre Token werden vom Lexer nicht verwendet, sondern unterstützen die Erzeugung von strukturierten Syntaxbäumen. Zum Beispiel kann ein imaginäres Token *PARAMETER_LIST* bei der Erzeugung eines ASB's dazu verwendet werden alle Parameterdefinitionen einer Funktion unter einem Knoten zu vereinigen. In diesem Zusammenhang ist zu erwähnen, dass

[84] vgl. [Par07, Kapitel 5]

Knoten von ASB's in ANTLR immer auf Basis eines Tokens erzeugt werden und dieses Token inklusive aller an dem Token hinterlegten Informationen enthält. [85]

Anschließend können beliebig viele globale Scopes definiert werden. Scopes sind eine Art Stack, die es ermöglichen über Regeln hinweg Informationen zu teilen. Ein typischer Anwendungsfall ist z. B. die Bestimmung von Variablentypen in Java. Des Weiteren können auch regelspezifische Scopes definert werden. Diese Scopes sind nur in den Regeln in der der Scope definiert wurde und in den Unterregeln definiert. Im Gegensatz zu globalen Scopes darf für regelspezifische Scopes kein Name spezifiziert werden, da der Name des Scopes dem Namen der Regel entspricht. In einem Scope können beliebige Informationen gespeichert werden. [86] Zum Beispiel kann ein globaler Scope *S*, der eine Liste von Strings enthält, wie folgt definiert werden:

```
...
scope S {
    List<String> strList;
}
...
```

Ein neuer Scope von *S* innerhalb einer Regel *r* kann durch

```
r
scope S;
:
...
;
```

erzeugt werden. Entsprechend müsste für einen regelspezifischen Scope die Definition der Elemente des Scopes innerhalb der Regel erfolgen:

```
r
scope {
    List<String> strList;
}
:
...
;
```

Im ersten Fall wird auf das Element *strList* des Scopes durch `$S::strList` zugegriffen. Im zweiten Fall würde dies durch `$r::strList` bewerkstelligt werden. Bei jedem Betreten der Regel *r* wird ein neuer Scope erzeugt. Auf Scopes, die oberhalb des in der Regel erzeugten Scopes liegen, kann durch den '[]'-Operator zugegriffen werden. Beim Verlassen einer Regel werden die Kopfelemente der definierten Scopes entfernt. [87]

Aktionen (*actions*) können sowohl global, als auch für Regeln definiert werden. Sie müssen in der Sprache, die durch die Option *language* definiert wurde, angegeben werden. Bei der Definition von Regeln können Aktionen entweder beim Betreten der Regel (Aktion **@init**), beim Verlassen der Regel (Aktion **@after**) oder auch mittendrin

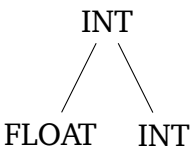
[85] vgl. [Par07, Kapitel 7]
[86] vgl. [Par07, Seite 135 ff.]
[87] vgl. [Par07, Seite 135 ff.]

ausgeführt werden (durch in **{}** eingerahmten Programmcode). Globale Aktionen werden zum Beispiel für die Definition von Funktionen verwendet, die innerhalb des Parsers in allen Regeln bekannt sein sollen. (Aktion **@members**) Des Weiteren kann auch die Fehlerbehandlung für alle Regeln durch die Defintion einer globalen Aktion angepasst werden (Aktion **@rulecatch**). [88]

In einer Grammatikdefinition müssen Lexerregeln und Namen von Token (engl. für Kurzzeichen) mit einem Großbuchstaben beginnen. Alle Regeln, die nicht zu der Definition eines Lexers gehören, beginnen mit einem Kleinbuchstaben. Insgesamt wird zwischen drei verschiedenen Regeltypen unterschieden. Zunächst gibt es Lexerregeln. Diese werden vom Lexer zur Erzeugung eines Kurzzeichenstroms verwendet. Dabei werden für alle Regeln, die nicht mit dem Schlüsselwort **fragment** markiert wurden, Kurzzeichen erstellt, die Informationen über den für die Erzeugung des Tokens verantwortlichen Teil des Quelltextes beinhalten. Diese Informationen werden als Attribute bezeichnet. Des Weiteren gibt es sogenannte Parser- und Baumparserregeln. Parserregeln werden für das Einlesen von durch den Lexer erzeugten Kurzzeichenströmen verwendet. Baumparserregeln beschreiben die Syntax eines ASB's und unterscheiden sich von den Parserregeln nur in der Verwendung von $\hat{}$ zur Markierung von Wurzelknoten. Die Parser- und Baumparserregeln verfügen ebenfalls über eine Menge vordefinierter Attribute. Für alle Regeln dürfen zusätzlich zu den vordefinierten Attributen beliebig viele weitere Attribute (auch Parameter) sowie Rückgabewerte spezifiziert werden. Eine Liste der vordefinierten Attribute ist im Anhang C.1 dargestellt. Zudem können semantische und syntaktische Prädikate zur Konfliktlösung eingesetzt werden. Für Parser und Baumparserregeln kann in Abhängigkeit des vorher eingestellten Ausgabetyps (siehe Option *output*), die Ausgabe spezifiziert werden. Dazu wird hauptsächlich der '->'-Operator verwendet. [89] Sei zum Beispiel *output = AST* und *ExampleGrammar* eine Baumgrammatik. Dann wird durch die Definition der Baumparserregel *primTypList*

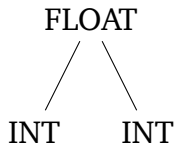
```
grammar ExampleGrammar;
...
primTypList
    :   ^(INT FLOAT INT)
        ->  ^(FLOAT INT INT)
    ;
...
```

erreicht, dass ein eingegebener Baum



in den Baum

[88] vgl. [Par07, Seite 118 ff.]
[89] vgl. [Par07, Kapitel 4, 6 und 7]



transformiert wird.

3.2.4.2 Beispiel

Im diesem Abschnitt wird anhand eines Beispiels, der Aufbau einer Grammtikdefintion weiter vertieft. Zur Illustration wird eine Grammatik für das Einlesen einfacher mathematischer Ausdrücke erstellt. Die zu definierende Grammatik soll die Addition und Substraktion von Zahlen abbilden und als Resultat das Ergebnis der Operationen zurückgeben.

Zunächst müssen der Name der Grammatik, sowie die Optionen und Token angegeben werden.

```
grammar SimpleMath;
options {
    language=Java;
}

tokens {
    PLUS  = '+';
    MINUS = '-';
}
```

Der Name der Grammatik muss mit dem Namen der Datei übereinstimmen. Als Zielsprache wurde Java gewählt. Da die Grammatik Additionen und Subtraktionen von Zahlen beschreiben soll, werden die Token *PLUS* und *MINUS* entsprechend definiert. Nachdem die Grundeinstellungen gemacht wurden, müssen nun die Regeln definiert werden.

```
startRule
    :   ZAHL subRule*
    ;

subRule
    :   MINUS ZAHL
    |   PLUS  ZAHL
    ;

ZAHL:   ('0' | '1'..'9' '0'..'9'*)
    ;

WS  :   (' ' | '\n' | '\r' | '\t' )+
            {
            // JAVA-CODE-START
                skip();
            // JAVA-CODE-ENDE
            }
    ;
```

Die beiden Regeln *ZAHL* und *WS* sind Lexerregeln. *ZAHL* definiert den Aufbau einer natürlichen Zahl, während *WS* die möglichen Leerzeichen, die zwischen den Zahlen und Operatoren auftreten können, beschreibt. Alle Leerzeichen werden durch den in den geschweiften Klammer angegebenen eingebetteten Java-Code direkt übersprungen.

Die Regel *startRule* ist die Startregel der Grammatik. Sie muss nicht extra gekennzeichnet werden. Zusammen mit der Regel *subRule* beschreibt sie die Parserregeln und somit den Aufbau einer Addition / Subtraktion von natürlichen Zahlen.

Um das Ergebnis der eingegebenen Rechenvorschrift zu berechnen muss nun zunächst die Regel *subRule* angepasst werden. Sie muss der Startregel zum Einen die Operation (Addition oder Subtraktion) und zum Anderen den Wert zurückgeben. Dafür werden für *subRule* zwei Rückgabewerte *retVal* und *isAddOp* definiert, auf die die Startregel später zugreifen kann. Der Zugriff auf Rückgabewerte und Parameter, sowie Attribute lokaler Variablen und Token muss mit dem Präfix '$' geschehen.

```
subRule returns [int retVal, boolean isAddOp]
    :   ( m=MINUS z=ZAHL
        |   PLUS  z=ZAHL)
    {
        $isAddOp = (m == null);
        $retVal = Integer.parseInt($z.text);
    }
    ;
```

Nach der Anpassung von *subRule* kann die Startregel das Ergebnis der Rechenvorschrift relativ einfach bestimmen. Dazu wird eine regellokale Variable *ergebnis* definiert, die am Ende des Parsevorganges das Ergebnis der Rechenvorschrift enthalten soll. Die Variable *ergebnis* wird nach jedem Aufruf von *subRule* mit Hilfe der Rückgabewerte aktualisiert. Das Ergebnis wird nach einem erfolgreichen Einlesen auf der Konsole ausgegeben.

```
startRule
@init{
    int ergebnis = 0;
}
@after{
    System.out.println("Ergebnis: " + ergebnis);
}
    :   ZAHL { ergebnis = Integer.parseInt($ZAHL.text); }
        (subRule
    {
        if($subRule.isAddOp) {
            ergebnis += $subRule.retVal;
        } else {
            ergebnis -= $subRule.retVal;
        }
    })*
    ;
```

Das Beispiel gibt lediglich einen kleinen Einblick in die Möglichkeiten, die ANTLR zur Verfügung stellt. Für eine detaillierte Einführung, die Handhabung abstrakter Syntaxbäume und die Ausgabe mittels Templates sei auf das Buch „*The Definitive ANTLR Reference*“[90] von Terence Parr, dem Erfinder von ANTLR, verwiesen, dass eine Basis für diese Arbeit darstellt.

[90] siehe [Par07]

4 Übersetzerkonstruktion

In diesem Kapitel wird zunächst die Zielsetzung konkretisiert, indem eine Java-Teilmenge definiert wird, die mit Hilfe des konstruierten Werkzeuges übersetzbar sein soll. Anschließend werden die unterschiedlichen Java-Konstrukte der definierten Teilmenge untersucht und äquivalentem C++-Code zugeordnet.

4.1 Konkretisierung der Zielsetzung

Im diesem Abschnitt wird die Zielsetzung konkretisiert. Dafür wird eine Untermenge von Java definiert, die das zu entwickelnde Werkzeug übersetzen soll. Die Untermenge wurde so gewählt, dass deren Übersetzung im Rahmen **einer** Diplomarbeit zu bewältigen ist. Die Ausgangsbasis bildet die dritte Auflage der *„Java Language Specification“* [91]. Die Spezifikation beschreibt alle in Java 1.5 zulässigen Konstrukte und wird durch die folgenden Anforderungen eingeschränkt:

- pro Quelldatei darf genau eine Klassendefinition erfolgen, d. h. implizit, dass keine anonymen, keine statischen und keine inneren Klassen und die damit verbundenen Konstruktionen verwendet werden dürfen
- statische Imports sowie **import**-Anweisungen mit einem abschließenden ‘.*‘ sind nicht zulässig
- alle Zeichen innerhalb der zu übersetzenden Java-Programme müssen einem Zeichensatz entstammen
- für jedes Array a gilt $dim(a) <= 2$
- Enumerationen dürfen lediglich Variablen auf Klassenebene enthalten, die finalen statischen Charakter besitzen
- die Deklaration von Variablenlisten ist nicht gestattet
- die Dimensionsklammern von Feldern müssen stets am Typbezeichner stehen
- Annotationen sind von der Übersetzung ausgeschlossen
- die Verwendung der Schlüsselwörter **native**, **synchronized**, **strictfp**, **finally**, **transient** und **volatile**, sowie aller Konstrukte, die in Verbindung mit diesen Schlüsselwörtern stehen, ist untersagt

[91] siehe [GSB05]

- die Verwendung der „*foreach*“-Schleife wird nicht unterstützt
- der Aufruf anderer Konstruktoren der aktuellen Klasse durch die Verwendung von **this** ist nicht erlaubt
- Generics sind von der Übersetzung ausgeschlossen
- die Zuweisung einer Variablen mit dem Wert **this** ist nicht zulässig
- (statische) Initialisierungsblöcke werden nicht übersetzt
- Reflections und alle damit verbunden Konstruktionen dürfen nicht verwendet werden
- **break** und **continue** dürfen nur ohne die Spezifikation eines Rücksprungpunktes angegeben werden

Zudem dürfen Klassen, Variablen und Funktionen nicht einen der Namen tragen, der einem in C++ reservierten Wort entspricht [92]. Des Weiteren ist die Definition von Variablen mit dem Namen *ENUM_NULL* und von Funktionen mit den Namen *isValid*, *invalidate* oder *create* untersagt.

Sollte eine dieser Anforderungen verletzt werden, wird der Übersetzungsvorgang sofort mit einer entsprechenden Fehlermeldung abgebrochen. Des Weiteren wird vorausgesetzt, dass die zu übersetzenden Quelltexte keine Fehler beinhalten.

[92] siehe [iso98, Seite 14 und 15]

4.2 Zuordnung der Java-Konstrukte

In diesem Unterabschnitt werden die verschiedenen zu übersetzenden Java-Konstrukte untersucht und äquivalenten C++-Code zugeordnet.

4.2.1 Klassenpräambel

Die Klassenpräambel in Java besteht aus den zwei optionalen Bestandteilen. Zum Einen handelt es sich dabei um die Angabe des Package's in dem die Klasse abgelegt ist (vgl. Regel *PackageDeclaration* in Abbildung 3) und zum Anderen um import-Anweisungen (vgl. Regel *ImportDeclaration* in Abbildung 3). Die Deklaration eines

CompilationUnit:
PackageDeclaration ?
ImportDeclaration *
...

PackageDeclaration:
Annotations? `package` *QualifiedIdentifier* ';'

ImportDeclaration:
`import static` ? *Identifier* ('.' *Identifier*) * ('.' *) ? ';'

Identifier:
IDENTIFIER

QualifiedIdentifier:
Identifier ('.' *Identifier*) *

Abbildung 3: Spezifikation der Klassenpräambel (in Anl. an [GSB05, Seite 585 ff.]

Packages am Anfang einer Java-Quelldatei, ähnelt der Deklaration einer Klasse innerhalb eines Namensraumes mit Hilfe des Schlüsselwortes **namespace** in C++.[93] In Java besteht die Packagedeklaration aus dem Schlüsselwort **package** gefolgt von gültigen Java-Identifiern, die wiederum mit einem '.' separiert werden.[94] Zum Beispiel besagt die Deklaration **package** de.ppi.Kernel;, dass die alle in der Quelldatei definierten Klassen dem Package *de.ppi.Kernel* zugeordnet sind und dieses Package ein Unterpackage von *de.ppi* ist, welches wiederum ein Unterpackage des Packages *de* ist. Weiterhin bedeutet die Zuordnung einer Klasse zu einem Package, dass der Klasse fast alle anderen in diesem Package deklarierten Klassen bekannt sind (nicht sichtbar sind anonyme und als *private* deklarierte Klassen, sowie als *protected* definierte Klassen, die

[93] vgl. [Str98, Seite 179 - 181]
[94] vgl. [Fla98, Seite 19 und 20]

von Klassen bereitgestellt werden, zu denen kein Verwandschaftsverhältnis besteht). Wird die Deklaration des Package, in dem die Klasse abgelegt ist, weggelassen, wird die Klasse dem Standardpackage zugeordnet. Des Weiteren ist in Java pro Datei nur die Angabe maximal eines Package möglich.[95] In C++ ist es hingegen möglich in einer Quelldatei verschiedene Klassen in verschiedenen Namensräumen abzulegen. Des Weiteren werden in C++ keinesfalls alle in einem Namensraum deklarierten Klassen automatisch inkludiert. Zudem ist in C++ der Separator '.' als Bestandteil eines Namensraumbezeichners nicht gestattet[96].

Die Unterschiede zwischen Namensräumen und Packages werden anhand des Beispiels 5 verdeutlicht.

Beispiel 5 Verdeutlichung der Unterschiede von **package** und **namespace**

Seien die Java Klassen *A*, *B* und *C* im Package *de.ppi.alphabet* hinterlegt. Dann könnte die Klasse *A* in Java wie folgt aussehen (nur Präambel):

```
package de.ppi.alphabet;
class A {
    void fun() {
        B b;
        C c;
        ...
    }
}
```

Die äquivalente C++-Klasse sieht wie folgt aus:

```
#include "B.hpp"
#include "C.hpp"

namespace de {
    namespace ppi {
        namespace alphabet {
            class A {
            ...
            };
        }
    }
}
```

Zusätzlich wird durch dieses Beispiel ersichtlich, dass eine naive Übersetzung zu unleserlichen Code führen kann, wenn eine tiefe Verschachtelung von Packages vorliegt. Eine Möglichkeit wäre es in jeder Package-Deklation den '.' durch einen '_' zu ersetzen. Dies würde zwar die Verschachtelung der Namensraumdeklarationen aufheben, aber bei tief verschachtelten Packages die Lesbarkeit nicht verbessern. Stattdessen kann der Anwender, der das Übersetzungswerkzeug bedient, vor dem Übersetzungs-

[95] vgl. [Fla98, Seite 20]
[96] siehe [iso98, Seite 13 und 14]

vorgang festlegen durch welche Namensräume die einzelnen Packages ersetzt werden sollen (siehe Abschnitt 4.3.1.2.1). Somit wird eine flache Namensraumstruktur in C++ möglich ohne in Java auf tief verschachtelte Packages verzichten zu müssen. Auf der anderen Seite bedeutet das Zusammenführen verschiedener Namensräume, dass eine Kollisionsüberprüfung eingebaut werden muss, die verhindert, dass zwei Namensräume zusammengeführt werden, die gleichnamige Klassen beinhalten, da dies ein unerwartetes Verhalten und Fehler zu Folge haben kann. Zudem muss das Übersetzungswerkzeug feststellen, welche anderen Klassen des gleichen Namensraumes die aktuelle Klassen benutzt und diese dann importieren.

Direkt nach der Angabe des Packages in der Präambel können beliebig viele Importanweisungen folgen. Das entsprechende Gegenstück in C++ ist die Präprozessor-Direktive **#include**. Eine Importanweisung in Java unterteilt sich in drei Kategorien, von denen, wie in Abschnitt 4.1 angegeben, nur die einfachste Variante unterstützt wird. Die einfachste Variante besteht aus dem Schlüsselwort **import** gefolgt vom vollständigen[97] Klassennamen. Eine Importanweisung hat den positiven Effekt, dass der Name der importierten Klasse in der gesamten Klasse bekannt ist und somit der Klassennamen bereits als vollqualifizierter Name ausreicht.[98] Das Beispiel 6 verdeutlicht den Nutzen der Importanweisung.

Beispiel 6 Nutzen der Verwendung von Importdirektiven

Sei *A* wiederum eine Java-Klasse, die im Package *de.ppi.alphabet* liegt. Dann könnte eine Klasse *Wort*, die im Package *de.ppi.dokument* liegt, wie folgt aufgebaut sein.

```
package de.ppi.dokument;
public class Wort {
    void fun() {
        de.ppi.alphabet.A a = new de.ppi.alphabet.A();
    }
}
```

Es ist zu erkennen, dass diese Form bei tief verschachtelten Packagestrukturen unleserlich wird. Stattdessen kann folgende äquivalente Definition von *Word* unter der Zurhilfenahme von **import** angeben werden:

```
package de.ppi.dokument;
import de.ppi.alphabet.A;
public class Wort {
    void fun() {
        A a = new A();
    }
}
```

Eine **#include**-Direktive reicht in vielen Fällen nicht aus um die Funktionalität einer

[97] *vollständig* heißt hier inklusive des Namen des Packages in dem die Klasse liegt
[98] vgl. [Fla98, Seite 20]

Importanweisung nachzubilden. Betrachten wir das vorherige Beispiel 6, wird ersichtlich, dass in C++ zusätzlich noch eine Using-Direktive verwendet werden muss, damit im Fall, dass die inkludierte Klasse in einem anderen Namensraum liegt als die inkludierende Klasse, der Klassennamen bereits als vollqualifizierter Name ausreicht. [99]

Eine weitere Differenzierung muss für den Spezialfall der Verwendung zweier gleichnamiger Klassen aus unterschiedlichen Package's vorgenommen werden. Sei zum Beispiel *X* wiederum eine Java-Klasse, die im Package *de.ppi.latinAlphabet* liegt. Des Weiteren liegt eine andere Klasse *X* im Package *de.ppi.romanNumber*. Dann könnte eine dritte Klasse *Symbol*, die im Package *de.ppi.util* liegt, wie folgt aufgebaut sein.

```
package de.ppi.util;
import de.ppi.latinAlphabet.X;
public class Symbol {
    de.ppi.romanNumber.X createRomanX() {
        return new de.ppi.romanNumber.X();
    }
    ...
    X createLatinX() {
        return new X();
    }
    ...
}
```

Eine zusätzliche Importanweisung **import** de.ppi.romanNumber.X; würde in Java einen Fehler zur Übersetzungszeit erzeugen. In C++ hingegen muss die Klasse *de.ppi.romanNumber.X* explizit per **#include**-Direktive eingebunden werden. Die Using-Direktive fuer den Namensraum *de.ppi.romanNumber* darf aber nicht angegeben werden, so dass sich beispielsweise folgender nahezu äquivalenter C++-Code ergibt (unter der Prämisse, dass die Package's *de.ppi.** auf * gemappt wurden):

```
include "../latinAlphabet/X.hpp";
include "../romanNumber/X.hpp";

using namespace latinAlphabet;

namespace util {
    class Symbol {
        romanNumber::X createRomanX() {
            return romanNumber::X();
        }
        ...
        X createLatinX() {
            return X();
        }
        ...
    };
}
```

Zusammenfassend bleibt festzuhalten, dass sich die Java-Präambel mit kleinen Mo-

[99] vgl.[Str98, Seite184]

difikationen in gut lesbaren, äquivalenten C++-Code transferierbar ist. Im folgenden Abschnitt werden die verschiedenen Kategorien von Java-Klassen und ihre Definitionen dargestellt, untersucht und ihre C++-Äquivalente aufgezeigt und diskutiert.

4.2.2 Klassendefinition

Eine Klassendefinition folgt direkt auf die Klassenpräambel. Da in Abschnitt 4.1 die Einschränkung getroffen wurde, dass pro Quelldatei genau eine Klasse definiert werden darf, muss diese Klasse mit dem Sichtbarkeitsattribut **public** ausgestattet sein. In Anlehnung an die „*The Java Language Specification*“ [100] lassen sich Klassen in sieben Kategorien unterteilen:

- „*normale*“ Klassen
- Abstrakte Klassen
- Schnittstellen (engl.: Interfaces)
- Enumerations
- statische Klassen
- Annotations
- Generische Klassen

Aufgrund der vorher getroffenen Einschränkungen werden statische Klassen, Annotationen und generische Klassen für den Moment nicht näher betrachtet. In Kapitel 7 wird auf diese speziellen Klassentypen nochmals eingegangen. Im Folgenden werden die restlichen vier Kategorien genauer untersucht.

4.2.2.1 „Normale“ Klassen, abstrakte Klassen und Schnittstellen

Eine Klasse wird als „*normal*“ verstanden, wenn sie allein über die Schlüsselwortkombination **public class**, ohne Hinzunahme des Schlüsselwortes **abstract**, deklariert wurde. Eine „*normale*“ Klasse besteht aus einer Menge von Konstruktoren, Membervariablen, Methoden, sowie statischen Variablen und Funktionen. Die Definition rein virtueller Methoden ist in „*normalen*“ Klassen nicht zulässig. [101] Sie hat genau eine Vaterklasse und kann eine beliebige Anzahl von Schnittstellen implementieren. Falls die Vaterklasse nicht explizit durch Verwendung des Schlüsselwortes **extends** angegeben wurde, erbt die Klasse automatisch von der Klasse *Object*. [102] An dieser Eigenschaft wird verdeutlicht, dass Java im Gegensatz zu C++ eine „*singly-rooted hierarchy*“ besitzt, das heißt, dass alle Klassen direkt oder indirekt von einer Klasse abgeleitet sind.

[100] siehe [GSB05]
[101] siehe [GSB05]
[102] vgl. [Hen97, Seite 114]

Abstrakte Klassen sind eine Mischung aus Schnittstelle und „*normaler*“ Klasse. Sie können sowohl rein virtuelle Methoden, die, wie die Klasse selbst, mit dem Schlüsselwort **abstract** markiert werden müssen, als auch alle Bestandteile einer „*normalen*“ Klasse enthalten. Die einzige Spezialität stellen Konstruktoren dar, die in abstrakten Klassen definiert werden dürfen, jedoch nur von den Subklassen der abstrakten Klasse innerhalb ihrer Konstruktoren aufgerufen werden können. Eine explizite Erzeugung einer Instanz einer abstrakten Klasse ist somit nicht möglich. [103]

Schnittstellen sind im Prinzip eine besondere Form einer abstrakten Klasse, die nur abstrakte, rein virtuelle Methoden und statische finale Variablen enthält. Sie werden durch das Schlüsselwort **interface** deklariert. Im Gegensatz zur abstrakten Klasse kann eine Schnittstelle nur andere Schnittstellen erweiteren und es existiert keine gemeinsame „*Basisschnittstelle*“. [104] Die Angabe einer Vaterklasse ist nicht zulässig. Die Verwendung von Schnittstellen ermöglicht eine Art Mehrfachvererbung, die aber ein Auftreten des Deadly Diamond of Death verhindert. [105]

Da in C++ **interface** und **abstract** keine Schlüsselwörter sind, werden Sie bei der Deklaration von Schnittstellen durch **class** ersetzt bzw. bei abstrakten Klassen weggelassen. Somit ergeben sich für die drei verschiedenen Kategorien von Klassen gleichartige Übersetzungen, wie in Beispiel 7 dargestellt ist.

Beispiel 7 Übersetzung von abstrakten und „*normalen*“ Klassendefinitionen sowie Schnittstellen

---

Seien *J* und *K* zwei Schnittstellen. Des Weiteren seien die Klassen *A*, *B* und *I* wie folgt definiert:

```
public class A extends B implements I { ... }

public abstract class B implements J  { ... }

public interface I implements K       { ... }
```

Die äquivalenten C++-Deklarationen sind dann:

```
class A : public B, public I          { ... };

class B : public J                    { ... };

class I : public K                    { ... };
```

Es ist zu erkennen, dass die Deklaration der Klassen in allen drei Fällen dem gleichen Schema folgt. Es treten lediglich Unterschiede in der Definition von Funktionen auf, auf die später eingegangen wird

---

[103] vgl. [Hen97, Seite 145]
[104] vgl. [Hen97, Seite 149]
[105] vgl. [Mar97]

4.2.2.2 Enumerations

Seit dem Update 1.5 sind richtige Aufzählungen in Java integriert. Die Sprache nutzt seitdem das Schlüsselwort **enum** für Aufzählungen, welches vorher schon reserviert, aber nicht belegt war. In Java sind Enumerationen im Gegensatz zu C++ eine spezielle Klassenform, deren Vaterklasse immer *java.lang.Enum* ist. Eine Enumeration kann ebenfalls beliebig viele Schnittstellen implementieren. Es kann aber keine Vaterklasse angegeben werden, da diese, wie bereits erwähnt, fest vorgegeben ist. [106] In der initialen Version des Übersetzers werden die Enumerations dahingehend eingeschränkt, dass nicht statische Membervariablen und Konstruktordefinitionen nicht gestattet sind. Für die Übersetzung der Java-Enumerations, dient eine Schablone, die für jede Enumeration eine verhaltensäquivalente C++-Klasse generiert (siehe Anhang C.6).
Diese Klassen haben allerdings kleinere Einschränkungen zur Folge:

- ENUM_NULL ist ein reserviertes Wort und darf nicht anderweitig verwendet werden. ENUM_NULL wird als Konstante jeder Enumeration hinzugefuegt, um das Setzen auf **null** in C++ abbilden zu können.
- Funktionsaufrufe, die in Java direkt auf die Enumeration-Konstanten angewendet werden, müssen in C++ durch Konstruktoraufrufe ersetzt werden. Beispielsweise sei MONTAG eine Enumeration-Konstante der Enumeration-Klasse E. Dann muss der Funktionsaufruf MONTAG.toString() in C++ in den Aufruf E(MONTAG).toString() umgewandelt werden. Diese Arbeit übernimmt der Übersetzer.
- Die explizite Definition von Methoden mit den Signaturen
  (a) java.lang.String toStringDefault()
  (b) **boolean** isValid()
  (c) **void** invalidate()

  ist nicht zulässig. Sie werden für die Erzeugung einer verhaltensäquivalenten *toString()*-Methode, zur Prüfung auf Nullheit und zum Invalidieren (entspricht dem Setzen auf **null** in Java) im generierten C++-Code verwendet.
- statische Variablen in Enumerationen dürfen nicht den Namen *rc* tragen

[106] vgl. [GSB05, Seite 249-258]

4.2.3 Objekttypen der verschiedenen Klassendefinitionen

Nachdem die zulässigen Formen von Klassendefinitionen diskutiert wurden, werden im diesem Abschnitt verschiedene Objekttypen vorgestellt und erläutert wann diese verwendet werden. In der aktuellen Version des Übersetzers wird zwischen sieben verschiedenen Objekttypen unterschieden:

- Heap-Objekt
- Value-Objekt
- Enum-Objekt
- Exception-Objekt
- Array-Objekt
- String-Objekt
- Interface-Objekt

In Java wird genau genommen nur zwischen primitiven Datentypen und Heap-Objekten unterschieden. Da in Java lediglich der Punktoperator für den Zugriff auf Klassenbestandteile verwendet wird und alle Objekte *by-Reference* übergeben (einzige Ausnahme: *java.lang.String*) sowie auf dem Heap angelegt werden, ist diese Unterscheidung ausreichend. In C++ ist es möglich Objekte auf dem Stack oder auf dem Heap anzulegen. Benutzt man den Operator **new** wird eine Instanz auf dem Heap angelegt, ohne **new** wird sie auf dem Stack angelegt. Zudem ist es möglich eine Instanz einer Klasse auf dem Heap und eine andere Instanz auf dem Stack zu erzeugen. Aufgrund der einfacheren Handhabung wird für jede Klasse zur Übersetzungszeit entschieden, ob ihre Instanzen entweder auf dem Stack oder auf dem Heap erzeugt werden. Zudem führt diese eindeutige Zuordnung zu verständlicheren Code, da der Programmierer sofort erkennen kann, wo das Objekt angelegt wurde und nicht kontextabhängig prüfen muss, ob die Instanz auf dem Stack oder Heap liegt. Es wäre durchaus möglich die sieben Objekttypen auf drei zu reduzieren, doch aufgrund der einfacheren Handhabung während des Übersetzungsvorganges wurde darauf verzichtet. Im Folgenden werden die verschiedenen Objekttypen erläutert.

4.2.3.1 Heap-Objekt

In Java wird jedes Objekt auf dem Heap abgelegt. Die Destruierung dieser Objekte wird durch den Garbage Collector geregelt und bedarf keinen Eingriff durch den Anwender. In C++ werden Objekte auf dem Heap abgelegt, sobald sie unter Zuhilfenahme des Operators **new** erzeugt wurden. [107] Die Destruierung ist im Gegensatz zu Java nicht durch einen Garbage Collector geregelt. Stattdessen muss die Destruierung

[107] vgl. [Str98, Seite136]

per Hand vorgenommen werden. Eine Abhilfe schaffen sogenannte Intelligente Zeiger (engl.: Smart-Pointer). Bei ihnen handelt es sich um spezielle Objekte, die einfache Zeigervariablen einkapseln und mit zusätzlichen Funktionen ausstatten. [108] Für die Abbildung von Heap-Objekten in C++ werden in der aktuellen Version referenzzählende intelligente Zeiger verwendet. Ein referenzzählender intelligenter Zeiger zählt intern die Anzahl der gehaltenen Referenzen auf die eingekapselte Instanz. Wenn die Anzahl der Referenzen auf die eingekapselte Instanz den Wert 0 annimmt, wird die Instanz destruiert. Alle Heap-Objekte werden von der C++-Klasse *HeapObject* abgeleitet. Die Konstruktoren der Heap-Objekte werden zur Übersetzungszeit in ihrer Sichtbarkeit so reduziert, dass sie maximal von Subklassen aufrufbar sind. Für jeden Konstruktor wird eine statische Factory-Funktion mit dem Namen create generiert, die die Konstruierung von Instanzen mit Unterstützung eines Macros vornimmt.
Zur Veranschaulichung dient das folgende Beispiel, indem die Java-Klasse *A* wie folgt definiert sei:

```
public class A {
    private int x;
    public A(int x) {
        this.x = x;
    }
}
```

Diese Klasse wird mit dem Übersetzer in folgende C++-Klasse übersetzt:

```
...
#include "../utils/HeapObject.hpp"
...
using namespace utils;
...
class A : public HeapObject {
    public:
        typedef HeapObjectPtr<A>::Ptr Ptr;

        // Factory-Funktion
        static A::Ptr create(int x) {
            // Aufruf eines Makro's, das ein Heap-Objekt
            // unter Zuhilfenahme des Konstruktors kreiert
            return NEW_HEAPOBJECT(A, x);
        }
    private:
        int x;
    protected:
        A(int x) {
            this->x = x;
        }
};
```

Der Zugriff auf Klassenvariablen und sonstige Bestandteile einer von *HeapObject* abgeleiteten Klasse erfolgt über den Pfeiloperator, der auf den Smart-Pointer angewendet

[108] vgl. [Har97]

wird und die entsprechenden Variablen des eingewrappten Heap-Objektes zurückliefert. Die Nullsetzung und Überprüfung auf Nullheit erfolgt direkt am Smartpointer und muss deshalb mit dem Punktoperator durchgeführt werden. Es gelten somit für den Fall, dass *x* eine Variable vom Typ *X* ist deren Instanzen Heap-Objekte sind, folgende Ersetzungsregeln:

- `X x = null` wird ersetzt durch `X::Ptr x;` (initiale Zuweisung)
- `x = null` wird ersetzt durch `x.invalidate()` (verzögerte Zuweisung)
- `x != null` wird ersetzt durch `x.isValid()`
- `x == null` wird ersetzt durch `!x.isValid()`

Die Frage, warum nicht die in C++ integrierte Konstante NULL verwendet wird, lässt sich dadurch erklären, dass NULL lediglich eine Konstante vom Typ **int** mit dem Wert 0 ist. [109] Diese Eigenschaft könnte bei einer Veränderung des generierten Quellcodes eine potentielle Fehlerquelle bilden. Stattdessen wird im Namensraum *utils* die Konstante **null** wie folgt definiert:

```
...
namespace utils {
class NullType
{};

// diese Konstante entspricht dem Java-Symbol <code>null</code>
const NullType null = NullType();
...
}
```

Die aktuelle Implementierung vernachlässigt das Auftreten zyklischer Heap-Objekte. Diese Problematik wird in Kapitel 7 diskutiert.

4.2.3.2 Value-Objekt

Ein Value-Objekt (oder Wertobjekt) ist ein Objekt, das auf dem Stack angelegt und für die Übergabe an Methoden kopiert wird. Da diese Objekte nur auf dem Stack angelegt werden, werden sie automatisch wieder entfernt, sobald ihr Gültigkeitsbereich verlassen wurde. Zu den Value-Objekten gehören neben den primitiven Datentypen auch der String, sowie Enumerationen. Diese werden aber aufgrund ihrer Sonderstellung separat betrachtet. Die Zugriff auf Klassenbestandteile von Wertobjekten erfolgt immer mit dem Punktoperator. Des Weiteren sind Wertobjekte integrierten Typs, wie in Java, nicht mit **null** initialisierbar.

[109] vgl. [LLM06, Seite 153]

4.2.3.3 Enum-Objekt

Da in C++ keine Enumerationen im Sinne von Java existieren, werden Enumerationen gesondert betrachtet und erhalten deshalb einen eigene Objekttypen. Enumerationen verhalten sich wie Value-Objekte. Die in Abschnitt 4.2.2.2 geschilderten Spezialbehandlungen rechtfertigen jedoch die Einführung eines eigenen Objekttypens.

4.2.3.4 Exception-Objekt

Ein Exception-Objekt ist ein Objekt, dass von der Java-Klasse *Throwable* abgeleitet wurde. Ausnahmen sollten in C++ by-Value geworfen und by-Reference gefangen werden. Der Grund dafür ist, dass das Fangen by-Reference auch Instanzen abgeleiteter Ausnahmen miterfasst. Auf Grund dieser speziellen Handhabung erhalten alle Ausnahmeklassen den Objekttypen Exception-Objekt.

4.2.3.5 Array-Objekt

In Java sind Arrays ebenfalls eine Art Objekt. Sie besitzen im Gegensatz zu Feldern in C++ ein internes „*bounds checking*“, was den Zugriff auf außerhalb des Arrays liegende Speicherzellen verhindert. [110] In C++ ist es dagegen möglich auf Speicherzellen die außerhalb eines Arrays liegen zuzugreifen. [111] So führt das Java-Programmfragment in Beispiel 8 zu einem Laufzeitfehler, während das C++-Programmfragment keinen Fehler meldet. Dieses Verhalten kann in weiterem Programmverlauf unerwartete Ne-

Beispiel 8 Fehlerhafter Zugriff auf Array-Elemente in Java und C++

Java-Programmfragment

```
...
int[] intArray = new int[]{1, 2, 3};
for(int i = 0; i < 4; i++) {
    intArray[i] = intArray[i]+1;
}
...
```

C++-Programmfragment

```
...
int intArray[]={1, 2, 3}
for(int i = 0; i < 4; ++i) {
    intArray[i] = intArray[i]+1;
}
...
```

beneffekte hervorrufen. Die Fehlersuche in solchen Fällen gestaltet sich oft problematisch und langwierig, da unter Umständen der Fehler in einem komplett anderen

[110] vgl. [Hen97, Seite 64]
[111] siehe [LLM06, Seite 143 ff.]

Teil des Programms auftreten kann. Zusätzlich zu dem „*bounds checking*“ enthalten Java-Arrays die Konstante length, die als Wert die Länge des Arrays enthält. Diese Konstante ist insbesondere für Schleifen, die über alle Elemente eines Arrays iterieren, von großem Nutzen. In Java werden alle Arrays von primitiven Datentypen mit deren Standardwert initialisiert (siehe Tabelle 2). Die nicht primitiven Datentypen werden

Tabelle 2: Standardwerte primitiver Datentypen (vgl. [GSB05, S. 71-72])

Name des Typs	**double**	**float**	**long**	**int**	**short**	**byte**	**char**	**boolean**
Standardwert	0.0d	0.0f	0L	0	(**short**)0	(**byte**)0	(**char**)0	**false**

hingegen mit **null** initialisiert. In C++ werden die Elemente eines primitiven Arrays, welches außerhalb eines Funktionsrumpfes definiert wurde mit 0 initialisiert. Arrays primitiven Typs, die innerhalb eines Funktionsrumpfes definiert wurden, bleiben uninitialisiert. Ein Array nicht primitiven Datentyps wird unabhängig davon, wo das Array definiert wurde, mit dem Standardkonstruktor der entsprechenden Klasse initialisiert, sofern dieser vorhanden ist. Falls kein Standardkonstruktor definiert wurde, muss die Initialisierung des Arrays explizit erfolgen. [112] Aufgrund dieser Gegebenheiten eignen sich die „*rohe*“ C++-Arrays nicht dazu um mit ihnen die Java-Arrays in C++ abzubilden. Stattdessen wird eine selbstgeschriebene generische Container-Klasse verwendet. Arrays können in Java beliebig viele Dimensionen haben, was die Verwendung einer generischen Klasse, insbesondere die Initialisierung der Elemente, erschwert. Arrays mit höheren Dimensionen sind in vielen Fällen ein Anzeichen für ein schlechtes Programmdesign. Zudem finden sie zumeist im mathematischen Umfeld Anwendung. Diese Bibliotheken sind für eine Übersetzung ungeeignet, da sie „*per Hand*“ optimiert werden um Laufzeitanforderungen zu genügen. Zusätzlich muss festgehalten werden, dass jedes mehrdimensionale Array auf ein eindimensionales Array projiziert werden kann. Deshalb wurde für den Übersetzer die vorläufige Einschränkung getroffen, dass lediglich ein- und zweidimensionale Arrays zulässig sind.

Die beiden Template-Klassen *HeapArray* und *HeapMatrix*, die beide von *HeapObject* abgeleitet sind, bilden die Basisklassen für ein- bzw. zweidimensionale Arrays. Sie verhalten sich wie Java-Arrays, mit dem Unterschied, dass sie dynamisch vergrößert werden können und eine explizite Initialisierung nicht möglich ist. Sie besitzen statt öffentlichen Konstruktoren wie jede andere von *HeapObject* abgeleitete Klasse Factory-Funktionen zum Erzeugen neuer Arrays. Da die Array-Größe nicht konstant ist, verfügen diese Klassen nicht über die öffentliche Konstante length, sondern über eine Methode getLength, die die aktuelle Länge des Arrays zurückgibt.

Aufgrund der nicht möglichen expliziten Initialisierung, werden vom Übersetzer während des Übersetzungsvorganges Funktionen generiert, die die Initialisierung vornehmen. Eine andere Möglichkeit wäre, die explizite Initialisierung wie in Beispiel 9 durchzuführen. Diese Möglichkeit besteht aber nur, wenn die Array-Konstruierung

[112] vgl. [LLM06, Seite 145]

Beispiel 9 Möglichkeit der expliziten Initialisierung eines HeapArray's

Java-Programmfragment

```
...
int[] intArray = new int[]{1, 2, 3};
...
```

C++-Programmfragment

```
...
HeapArray<int>::Ptr intArray = HeapArray<int>::create(4);
intArray->at(1) = 1;
intArray->at(2) = 2;
intArray->at(3) = 3;
...
```

nicht in Methodenaufrufen, in anderen Array-Initialisierungen oder in statischen Variablen stattfindet. In diesen Fällen muss die Initialisierung in Funktionen ausgelagert werden. Aufgrund der einfacheren Handhabung wird in der aktuellen Version des Übersetzers für jede Array-Initialisierung eine Funktion generiert, die diese Initialisierung vornimmt. Insgesamt bleibt also festzuhalten, dass die Java-Arrays momentan nicht im vollständigen Umfang abgebildet werden. Des Weiteren rechtfertigt die spezielle Handhabung dieser Klasse die Ausgliederung der Arrays als eigenen Objekttyp, obwohl sie implizit ein Heap-Objekt darstellen.

4.2.3.6 String-Objekt

Auf den ersten Blick wirkt es absurd einer einzelnen Klasse einen eigenen Typen zu widmen, aber die Klasse *java.lang.String* hat in Java eine außerordentliche Sonderstellung. In Java ist es nicht möglich Operatoren für Klassen zu überladen, wie es etwa in C++ möglich ist. [113] Die einzige Klasse, die die Verwendung der Operatoren '+' und '+=' für die Verknüpfung zweier Objektinstanzen unterstützt, ist *java.lang.String*. So ist es möglich mit Hilfe dieser Operatoren eine Konkatenation von Strings durchzuführen. Diese Sonderstellung führt kann aber zu Verwirrungen führen, so dass bei der Konkatenation eines Strings mit einem primitiven Datentypen eine kleine Modifikation eine große Wirkung zur Folge haben kann. Seien zum Beispiel zwei Strings *s1* und *s2* wie folgt gegeben:

```
String s1 = 1 + 2 + 3 + "Test";
String s2 = "" + 1 + 2 + 3 + "Test";
```

Die beiden Variablen *s1* und *s2* enthalten nicht die gleiche Zeichenkette, denn *s1* = "*6Test*" $\neq$ "*123Test*" = *s2*. Dieses Phänomen stellt auf den ersten Blick keine Besonderheit da, kann aber zu Fehlern bei einer naiven Übersetzung führen. Zum Beispiel könnte in einem naiven Ansatz alle '+' durch '< <' ersetzt werden.

[113] vgl. [Lew97]

Das würde dazu führen, dass die Definition von *s1* in folgenden C++-Code überführt wird:

```
String s = String() << 1 << 2 << 3 << "Test"
```

Das würde bedeuten, dass *s1* = "*123Test*" = *s2* gilt $\lightning$. Deshalb muss dieser Sonderfall erkannt werden und wie folgt übersetzt werden:

```
String s = String() << (1 + 2 + 3) << "Test"
```

Des Weiteren muss bedacht werden, dass eine Konkatenation eines Strings mit einem beliebigen Objekt möglich ist, so dass beispielsweise für eine Instanz *c* einer Klasse *C* eine Konkatenation mit einem String folgende Feststellung gemacht werden kann:

```
String s = new String();
String ergebnisStr = s + c;
```

ist äquivalent zu

```
String s = new String();
String ergebnisStr = s + c.toString();
```

Das heißt, dass für jeden nicht primitiven Datentypen die *toString*-Methode aufgerufen wird und der Rückgabewert der Methode zur Konkatenation verwendet wird. Da Java diese Ersetzung „*unter der Hand*" durchführt, erschwert dies zusätzlich die Übersetzung von Stringkonkatenationen.[114] Aufgrund dieser Eigenheiten vereinfacht eine Differenzierung eines Strings von allen anderen Objekten die Handhabung von Strings während des Übersetzungsvorganges. Die Instanzen dieser Klasse werden auf dem Stack hinterlegt und *by-Value* übergeben. Somit verhalten sich Strings wie primitive Datentypen und sind eine spezielle Form eines Value-Objekt's. Die Implementierung der String-Klasse in C++ ist der in Java nachempfunden, so dass sie nahezu alle Methoden des Java-Strings enthält und Instanzen auf **null** setzbar sind. Der einzige Unterschied zu Java besteht in der Funktionalität des '=='-Operators, der in Java eine häufige Fehlerquelle darstellt. In Java prüft der '=='-Operator, ob zwei String-Instanzen sich auf den gleichen Speicherort beziehen.[115] Dieses selten nützliche Feature wurde in der C++-Klasse entfernt, so dass der '=='-Operator nun die gleiche Funktionalität vorweist wie die *equals*-Methode.

[114] vgl. [Fla98, Seite 35 und 36]
[115] vgl. [Fla98, Seite 29]

4.2.3.7 Interface-Objekt

Da die Umsetzung von Schnittstellen noch nicht vollständig ist, wird zur Vorsicht zunächst ein eigener Objekttyp eingeführt, der aber in dieser Arbeit nicht weiter betrachtet wird. Auf die Problematik, die bei der Verwendung von Schnittstellen auftreten kann, wird in Kapitel 7 eingegangen.

4.2.4 Klassenrumpf

Im folgenden Abschnitt werden die verschiedenen Bestandteile eines Klassenrumpfes untersucht und diskutiert. Zunächst werden die verschiedenen Sichtbarkeitsattribute vorgestellt.

4.2.4.1 Sichtbarkeitsattribute

Jedes Mitglied einer Klasse besitzt genau ein Sichtbarkeitsattribut. Dabei wird zwischen

- öffentlicher Sichtbarkeit (gekennzeichnet durch das Schlüsselwort **public**)
- geschützter Sichtbarkeit (gekennzeichnet durch das Schlüsselwort **protected**)
- klasseninterener Sichtbarkeit (gekennzeichnet durch das Schlüsselwort **private**)
- paketweiter Sichtbarkeit (Standardsichtbarkeit)

unterschieden. Die ersten drei Sichtbarkeitsattribute sind äquivalent zu den gleichnamigen Attributen in C++. Die paketweite Sichtbarkeit ist in Java die Standardsichtbarkeit, die einem Klassenmitglied dann zugeordnet wird, wenn keines der anderen drei Attribute explizit angegeben wurde. [116] Die ersten drei Attribute sind direkt nach C++ portierbar. Lediglich die paketweite Sichtbarkeit muss in C++ auf eine spezielle Art emuliert werden, da diese in C++ nicht integriert. Dazu wird während des Übersetzungsvorganges protokolliert, welche paketweit sichtbaren Mitglieder von welchen Klassen verwendet werden. Anschließend muss jede Klasse zu allen anderen Klassen, die ein paketweit sichtbares Mitglied ihrer Klasse referenzieren, eine Freundschaftsbeziehung deklarieren. Dies geschieht unter Zurhilfenahme des C++-Schlüsselwortes **friend**. Abschließend wird das Mitglied mit klasseninterner Sichtbarkeit ausgestattet. Seien beispielsweise die Java-Klassen *A* und *B* wie folgt definiert:

```
package alphabet;
public class A {
    ...
    int x;
    protected int y;
    ...
}

package alphabet;
public class B {
    void fun() {
        A a = new A();
        a.x = 10;
    }
}
```

[116] vgl. [Fla98, Seite 78]

Die Klasse *A* würde dann in C++ wie folgt aussehen:

```
...
namespace alphabet {
    class A {
    ...
    private:
        // Emulieren der paketweiten Sichtbarkeit
        friend class B;
        int x;
    protected:
        int y;
    ...
    };
}
```

Die Definition **friend class** B; sorgt dafür, dass die Klasse *B* auf alle Mitglieder der Klasse zugreifen kann. Diese Emulierung bildet die paketweite Sichtbarkeit nicht bijektiv ab, da die Klasse *B* auch Zugriff auf die Membervariable *y* hat, obwohl sie kein Kind der Klasse *B* ist.

4.2.4.2 throws-Direktive

In Java ist, wie in C++, die Angabe einer Liste von Ausnahmen möglich, die während der Ausführung einer Methode oder eines Konstruktors auftreten können. Der Ansatz diese Funktionalität eins-zu-eins abzubilden schlägt in vielen Fällen fehl. Der Grund dafür ist, dass in Java eine Menge bestimmter Ausnahmen existiert, die von jeder Methode und jedem Konstruktor geworfen werden können (z. B. java.lang.IndexOutOfBoundsException). In C++ dürfen, falls per **throws**-Direktive eine Liste der Ausnahmen angegeben wurde, auch nur genau diese Ausnahmen während der Ausführung des Rumpfes nach aussen weitergegeben werden. Tritt eine nicht erwähnte Ausnahme auf, führt dies zum sofortigen Programmabbruch. [117] Sei z. B. folgendes Java-Programm gegeben:

```
public class BadThrows {
    public static void fun() throws SomeException {
        int[] a = new int[4];
        ...
        a[4] = 10; // Ups -> IndexOutOfBoundsException
        if(a[0] == 0) {
            throw SomeException();

        }
    }

    public static void main(String[] args) {
        try {
            fun();
        } catch (SomeException se) {
            ...
        } catch (IndexOutOfBoundsException ioobe) {
            System.err.println("Ungueltiger Index");
        }
    }
}
```

Die Ausführung dieses Programms würde die Meldung „*Ungueltiger Index*“ ausgeben. Eine Übersetzung der Funktion *fun* in folgenden C++-Code

```
...
public:
    static void fun() throw(SomeException) {
        HeapArray<int>::Ptr a = HeapArray<int>::create(4);
        ...
        a->at(4) = 10; // Ups -> IndexOutOfBoundsException
        if(a->at(0) == 0) {
            throw SomeException();

        }
    }
```

würde beim Ausführen des Programms zu der Meldung „*terminate called after throwing an instance of 'IndexOutOfBoundsException'. Aborted*“ führen. Dass heisst, dass die

[117] vgl. [LLM06, Seite 826]

Ausnahme nicht vom **try-catch** gefangen wurde, sondern es aufgrund des Auftretens der nicht in der **throw**-Liste der Funktion angegebenen *IndexOutOfBoundsException* die Ausführung sofort abgebrochen wurde. Ein Lösung wäre für jede Methode und jeden Konstruktor die Liste der Ausnahmen zu bestimmen und diese in der **throw**-Liste anzugeben. Dies ist einerseits mit sehr viel Aufwand verbunden und führt zudem zu einem aufgeblähten Zielcode. Beispielweise müsste für jede Methode, die auf Feldern arbeitet angegeben werden, dass sie eine *IndexOutOfBoundsException* werfen kann. Stattdessen werden zur Behebung dieses Problem **throw**-Listen komplett weggelassen, da falls keine **throw**-Liste spezifiziert wurde, beliebige Ausnahmen geworfen werden können.

4.2.4.3 Statische Klassenmitglieder

Statische Mitglieder einer Klasse können zum Einen Variablen oder Konstanten und zum Anderen Funktionen sein. In Java unterscheidet sich die Deklaration eines statischen Mitglieds von einem nicht statischen Mitglied lediglich durch das Schlüsselwort **static**. In Java muss die Deklaration und Definition von statischen Bestandteilen innerhalb der Klassendefinition erfolgen. In C++ muss die Initialisierung statischer Variablen außerhalb der Klassendefinition erfolgen. [118] Wie bei „*normalen*“ Funktionen kann die Definition von statischen Funktionen entweder zusammen mit der Deklaration oder separat erfolgen, wobei in diesem Fall das Schlüsselwort **static** weggelassen werden muss. Die Referenzierung eines statischen Klassenmitglieds in Java wird über den qualifizierten Klassennamen gefolgt von einem ‘.‘ und dem Namen des Mitglieds vorgenommen. In C++ werden statische Mitglieder immer über den qualifizierten Namen gefolgt von einem ‘::‘ und dem Namen des Mitglied referenziert. So wird der Java-Ausdruck `java.lang.String.valueOf(10)` in C++ durch `utils::String::valueOf(10)` ersetzt. Die Verwendung von Usingdirektiven führt zur Einsparung des führenden Namensraum.

4.2.4.4 Main-Funktion

Die Main-Funktion ist in Java eine besondere statische Funktion. Sie dient wie die C++-Main-Funktion als Einstiegspunkt für den Programmstart. [119] Vergleicht man die Signaturen der Java- und der C++-Main-Funktion

Java `public static void main(String[] args)`

C++ `int main(int argc, *char argv[])`

miteinander wird deutlich, dass eine Eins-zu-Eins-Übersetzung nicht möglich ist. Zum Einen sind die Rückgabewerte der Funktionen unterschiedlich und zum Anderen muss die Java-Main-Funktion in eine Klassendefinition eingebettet werden, während die C++-Main-Funktion außerhalb einer Klasse definiert wird. Aufgrund dieser Eingenschaft ist es der Java-Main-Funktion möglich auf statische Klassenbestandteile der

[118] vgl. [LLM06, Seite 555 ff.]
[119] vgl. [Fla98, Seite 16]

Klasse direkt zuzugreifen. Eine C++-Main-Funktion hat keine einfache Möglichkeit um auf eine klasseninterne statische Variable zuzugreifen. Dieses Problem lässt sich lösen, indem die Java-Main-Funktion in der C++-Main-Funktion aufgerufen wird (vgl. Beispiel 10).

Beispiel 10 Übersetzung einer Main-Funktion

Sei die Java-Klasse *A* wie folgt definiert:

```
public class A {
    ...
    public static void main(String[] args) {
        ...
    }
    ...
}
```

In C++ würde die Main-Funktion wie jede andere statische Funktion übersetzt werden. Zusätzlich wird dazu folgende C++-Main-Funktion generiert, die die ursprünglich Main-Funktion des Java-Programmes aufruft.

```
int main(int argc, *char argv[]) {
    HeapArray<String>::Ptr args = HeapArray<String>::create();
    // das 0-te Argument ist in C++ der Name des Programms
    for(int i=1; i < argc; ++i) {
        args->at(i) = argv[i];
    }
    // Hier wird die eigentliche Main-Funktion aufgerufen
    A::main(args);
    return 0;
}
```

4.2.4.5 Nicht statische Klassenmitglieder

In Java ist in allen Klassendefinitionen ausser Schnittstellen die Implementierung von Methoden gestattet.[120] Zusätzlich ist in abstrakten Klassen und Schnittstellen die Deklaration rein virtueller (oder abstrakter) Methoden möglich. Die Definition nicht statischer Klassenvariablen ist in Schnittstellen und aufgrund der getroffen Einschränkungen aus Abschnitt 4.1 in Enumerationen nicht gestattet. Aus gleichem Grund ist Definition von Konstruktoren nur in „*normalen*“ Klassen erlaubt. Im Folgenden werden die einzelnen nicht statischen Bestandteile einer Klasse betrachtet.

4.2.4.5.1 Klassenvariablen

In Java ist es möglich Klassenvariablen direkt zu initialisieren, während in C++ nur

[120] vgl. [Fla98, Seite 82]

die Deklaration einer Variablen innerhalb einer Klassendefinition erlaubt ist. Des Weiteren werden in Java alle Klassenvariablen primitiven Typs, denen bei ihrer Deklaration kein expliziter Wert zugewiesen wurde, mit ihrem Defaultwert belegt (vgl. Tabelle 2). Alle unitialisierten Klassenvariablen, die einen nicht primitiven Typ besitzen, werden mit dem Wert **null** vorbelegt. Aus diesen Gründen ist es notwendig, alle Klassenvariablen in C++ in der Intialisierungsliste eines jeden Konstruktors der Klasse entsprechend zu initialisieren.

4.2.4.5.2 Methoden

Die Definition von Methoden in Java ist denen von C++ ähnlich, so dass eine Übersetzung intuitiv erfolgen kann. Es muss aber beachtet werden, dass es in C++ ein Unterschied darstellt, ob der Rumpf einer Methode im Header oder in der Implementierung der Klasse erfolgt. So behandelt ein C++-Compiler eine Methode, die innerhalb eines Headers definiert wurde, als wenn sie mit dem Schlüsselwort **inline** deklariert wurde. Er versucht dann ggf. alle Methodenaufrufe durch den gesamten Methodenrumpf zu ersetzen. [121] Ob eine Methode bzw. ein Konstruktor **inline** deklariert wird, entscheidet der Parser in der Zweiten Zwischenphase (siehe Abschnitt 5.3). Zudem muss bestimmt werden, welche Methoden als **virtuell** deklariert werden müssen (siehe Abschnitt 5.2.3). Diese müssen in C++ mit dem Schlüsselwort **virtual** markiert werden. [122]

4.2.4.5.3 rein virtuelle Methoden

In abstrakten Klassen und Schnittstellen treten zusätzlich rein virtuelle Methoden auf. Diese Methoden werden in der Klasse nur deklariert aber nicht definiert. Stattdessen müssen diese Methoden von Subklassen überschrieben werden. Um Methoden in C++ als rein virtuell zu markieren, müssen diese mit dem Schlüsselwort **virtual** und einem auf die Methodensignatur folgenden ‘=0‘ deklariert werden. [123] Explizit bedeutet dies beispielsweise, dass

```
abstract void fun();
```

in

```
vitual void fun()=0;
```

übersetzt wird.

4.2.4.5.4 Konstruktoren

In Abhängigkeit des Objekttypen der Klasse, werden für Heap-Objekte zusätzlich zu den eigentlichen Konstruktoren Fabrikmethoden zur Erzeugung neuer Instanzen generiert. Gleichzeitig wird die Sichtbarkeit der als **public** deklarierten Konstruktoren auf **protected** reduziert, so dass diese nur noch von Subklassen aufgerufen werden

[121] vgl. [LLM06, Seite 312-314]
[122] vgl. [Bre99, Seite 285 ff.]
[123] vgl. [Str98, Seite 331-333]

können. Hinsichtlich der Konstruktoren unterscheidet sich C++ von Java in zwei Dingen. Einerseits wird in C++ der Aufruf von Konstruktoren der Vaterklassen in der Initialisierungsliste und nicht am Anfang des Rumpfes angegeben. [124] In Java wird der Aufruf eines Konstruktors der direkten Vaterklasse durch das Schlüsselwort **super** realisiert. Dabei werden die entsprechenden Parameter mitübergeben. Falls kein Ausdruck für den Aufruf eines Superkonstruktors angegeben wurde, wird, falls vorhanden, der Standardkonstruktor der Vaterklasse aufgerufen. Ein weiteren Unterschied stellt die Möglichkeit dar, in Java mit dem Schlüsselwort **this** andere Konstruktoren der eigenen Klasse aufzurufen. [125] Dieses Feature wird momentan noch nicht unterstützt, da es einer ausgefeilten Kollisionsprüfung und - auflösung bedarf um äquivalenten und gleichzeitig eleganten C++ zu generieren (siehe Abschnitt 7). Wurde in einer Java-Klasse kein Konstruktor definiert, wird während des Übersetzungsvorganges ein Defaultkonstruktor generiert. Dies ist zwingend notwendig, um alle Klassenvariablen korrekt über die Initialisierungsliste initialisieren zu können. Des Weiteren werden Konstruktoren, die genau ein Argument besitzen, im Zielcode mit dem Schlüsselwort **explicit** markiert. Dies ist eine Vorsichtsmaßnahme, da in C++ Konstruktoren mit einem Parameter für implizite Typumwandlungen verwendet werden können, falls sie nicht als **explicit** markiert wurden. Diese Konstruktoren können dann nur noch für eine explizite Erzeugung von Objektinstanzen verwendet werden. [126] Für eine detaillierte Diskussion dieser potentiellen Fehlerquelle sei auf Scott Meyers Buch „*Mehr Effektiv C++ programmieren*“[Mey98, Seite 39-46] verwiesen.

4.2.5 Sonstiges

Im folgenden Abschnitt werden weitere Java-Konstrukte betrachtet, die eine besonderer Übersetzung bedürfen.

Die Angabe von expliziten Typkonvertierungen (Casts) wird im Java im Stil von C angegeben. Zum Beispiel kann ein Fließkommaliteral f auf folgende Weise in ein **int** konvertiert werden:

```
int x = (int) f;
```

In C++ wird hingegen zwischen vier verschiedenen expliziten Konvertierungen unterschieden: [127]

- dynamic_cast (zur Laufzeit geprüfte Konvertierung)
- static_cast (zur Übersetzungszeit geprüfte Konvertierung)
- const_cast (**const**-Konvertierung)
- reinterpret_cast (ungeprüfte Konvertierung)

[124] vgl. [LLM06, Seite 683]
[125] vgl. [Hen97, Seite 116]
[126] vgl. [LLM06, Seite 547 und 548]
[127] vgl. [Str98, Seite 129]

Für die in dieser Arbeit betrachteten Konvertierungen sind nur **dynamic_cast** und **static_cast** relevant. Eine explizite Konvertierung in Form eines **dynamic_cast** wird dazu benutzt eine Referenz oder einen Zeiger auf ein Objekt einer Basisklasse in eine Referenz oder einen Zeiger einer anderen Klasse, die in derselben Hierachie liegt, zu tranformieren. In der letzten Phase des Übersetzungsvorganges wird für alle expliziten Konvertierungen von Heap-Objekten und Exception-Objekten ein **dynamic_cast**-Ausdruck generiert. In einer Weiterentwicklung des Übersetzers müssen für alle nicht primitiven Datentypen (exklusive *String*) ebenfalls dynamische Casts verwendet werden. Momentan werden alle anderen expliziten Konvertierungen durch einen **static_cast** realisiert. Ein statischer Cast ist immer dann möglich, wenn ein C++-Compiler diese Konvertierung auch implizit durchführen könnte (beispielsweise für primitive numerische Datentypen) [128]

Die Übersetzung von **instanceof**-Ausdrücken wird mit Hilfe dynamischer Casts realisiert. Dafür wird die Eigenschaft ausgenutzt, dass **dynamic_cast** bei einer fehlgeschlagenen Konvertierung eines Zeigers den Wert der Konstanten NULL zurückliefert. [129] Deshalb können Ausdrücke der Form

```
if(x instanceof X) ...
```

in C++ in

```
if(dynamic_cast<X*>(x) != 0) ...
```

konvertiert werden.

Des Weiteren existiert in C++ kein primitiver Typ **byte.** [130] Dieser wird wird im Namensraum *utils* durch **typedef signed char** byte; global definiert und muss vor einer Verwendung innerhalb einer generierten C++-Klasse durch entsprechende **#include**- und Using-Direktiven bekannt gemacht werden. Der primitive Typ **char** aus Java wird in C++ als **unsigned char** interpretiert, damit eine Differenzierung der beiden Datentypen weiterhin möglich ist.

Zudem ist in Java die Verwendung von Unicodezeichen zulässig. [131] Deshalb werden diese innerhalb von String- und Characterliteralen ersetzt. Dabei wird davon ausgegangen, dass alle verwendeten Unicodezeichen aus einem Zeichensatz stammen. Dieser Zeichensatz kann durch die Angabe eines entsprechenden Paramters beim Aufruf des Programmes festgelegt werden.

[128] vgl. [LLM06, Seite 229-233]
[129] vgl. [LLM06, Seite 895]
[130] vgl. [Str98, Seite 76]
[131] vgl. [Fla98, Seite 212 ff.]

4.3 Vorbereitende Maßnahmen für den Übersetzungsprozess

In diesem Abschnitt werden die vorbereitenden Maßnahmen vor der Durchführung des Übersetzungsvorganges beschrieben. Zunächst wird die Funktionalität und der Aufbau der Konfigurationsdatei, die für den Übersetzungsvorgang benötigt wird, erläutert. Anschließend wird beschrieben, wie das Ausgabeformat des Übersetzungsprozesses definiert ist. Schließlich werden die verschiedenen Kommandozeilenargumente für den Programmstart und ihre Funktionalität erklärt.

4.3.1 Konfigurationsdatei

Die Konfigurationsdatei stellt einen wichtigen Bestandteil des Übersetzungsprozess da. Im Folgenden wird zunächst die Syntax der Konfigurationsdatei beschrieben. Anschließend werden an Beispielen die für den Übersetzungsvorgang relevanten Konfigurationseinstellungen erläutert

4.3.1.1 Syntax

Die Grammatik für die Konfigurationsdatei ist, wie die für Java und die verschieden abstrakten Syntaxbäume, für ANTLR geschrieben und somit eine LL-Grammatik. Ein Ausschnitt dieser Grammatik ist in Beispiel 11 dargestellt. Die vollständige Definition ist im Anhang C.4 abgebildet.

Beispiel 11 Ausschnitt der Mapping- und Konfigurationssyntax

```
mappingStart
    :   mappingDefinition* ;
mappingDefinition
    :   IDENT ASSIGN_EQUAL mappingDefinitionRightSide
    |   IDENT ASSIGN identList ;
mappingDefinitionRightSide
    :   mappingList
    |   IDENT ;
mappingList
    :   OPENING_MAP_BRACE mappingValueDefinition
            (COMMA mappingValueDefinition)* CLOSING_MAP_BRACE ;
mappingValueDefinition
    :   identList ASSIGN IDENT
    |   identList ASSIGN mappingList
    |   identList ASSIGN objectList
    |   IDENT ASSIGN IDENT
    |   IDENT ASSIGN objectList
    |   IDENT ASSIGN mappingList ;
```

Die Startregel ist *mappingStart*. Sie besagt, dass eine gültige Konfiguration aus beliebig vielen *mappingDefinition*'s bestehen kann. Eine *mappingDefinition* entspricht einer Zuweisung in einer Programmiersprache. Dabei wird zwischen zwei verschiedenen

Zuweisungen unterschieden. Einerseits ist es möglich einem Bezeichner eine Liste von Bezeichnern zuzuweisen. Andererseits kann einem Bezeichner ein einzelner Wert oder eine „*Map*“ von Werten zugeordnet werden, durch die die Modellierung komplexerer Konfigurationen ermöglicht wird. Die Konfigurationsdatei wird vor dem Übersetzungsvorgang eingelesen und auf Korrektheit überprüft. Im folgenden Abschnitt werden die verschiedenen verwendeten Konfigurationseinstellungen, die angegeben werden können, aufgelistet und erläutert.

4.3.1.2 Inhalt

Die Syntax der Konfigurationsdatei ist allgemein gehalten, um spätere Anpassungen zu vereinfachen. In der aktuellen Version des Übersetzers sind unter folgenden Bezeichner abgelegte Informationen relevant:

- namespaces
- types
- substitute
- baseClasses
- classes

4.3.1.2.1 namespaces

Der Bezeichner „*namespaces*“ dient dem Mappen von Java-Packages auf C++-Namensräume. Er definiert Zuweisungen, die für jedes Java-Package beschreiben, auf welchen C++-Namensraum dieser gemappt werden soll. Ein Beispiel für eine gültige Definition wäre

```
namespaces := {java.lang = utils,
               java.io   = utils,
               alphabet  = alphabet,
               %         = project,
               *         = project}
```

In dem Beispiel werden die Java-Packages *java.lang* und *java.io* auf den C++-Namensraum *utils* gemappt. Das Package *alphabet* wird auf sich selbst abgebildet. Die beiden letzten Zuweisungen bedeuten, dass alle anderen Packages (*) und das Default-Package (%) auf den Namensraum *project* gemappt werden sollen.

4.3.1.2.2 types

Der Bezeichner „*types*“ dient dem Mappen von Java-Klassen auf C++-Klassen. Beispielsweise bewirkt die Definition

```
types := {java.lang.StringBuffer
           = {className = String,
              namespace = utils}
         }
```

, dass die Java-Klasse *StringBuffer* durch die C++-Klasse *String* aus dem Namensraum *utils* ersetzt wird. Diese Art der Typüberschreibung kann unter Umständen zu fehlerhaften C++-Code führen, da dadurch beispielsweise zwei in Java mit unterschiedlichen Signaturen ausgestattete Methoden plötzlich identische Signaturen bekommen können (vgl. Beispiel 12. Deshalb sollte diese Funktionalität mir Sorgfalt verwendet werden.

Beispiel 12 Mögliche Nebeneffekte durch die Definition von „*types*“ in einer Konfigurationsdatei

Sei folgende Konfiguration gegeben:

```
namespaces := {java.lang = utils}

types      := {java.lang.StringBuffer
                 = {className = String,
                    namespace = utils}
               }
...
```

Des Weiteren sei folgendes Java-Code-Fragment zu übersetzen

```
public class X {
    ...
    public void fun(java.lang.String str) {
        ...
    }

    public void fun(java.lang.StringBuffer strBuf) {
        ...
    }
    ...
}
```

Eine Übersetzung auf Basis der angegebenen Konfiguration würde zu zwei Funktionen einer Klasse führen, die eine identische Signatur vorweisen, was weder in Java noch in C++ erlaubt ist.

4.3.1.2.3 baseClasses

Der Bezeichner „*substitute*“ dient der Definition der Basisklassen für die verschiedenen Objekttypen. So wird durch die Definition

```
baseClasses := {HEAP_OBJ      = {className    =   HeapObject,
                                 namespace    =   utils
                                },
                EXCEPTION_OBJ = {className    =   BaseException,
                                 namespace    =   utils
                                }
```

bewirkt, dass alle Heap-Objekte von der C++-Klasse *utils::HeapObject* und alle Exceptions von der Klasse *utils::BaseException* abegeleitet werden.

4.3.1.2.4 substitute

Der Bezeichner „*substitute*“ dient der Definition von Ersetzungsregeln für Mitglieder einer bestimmten Klasse. Es kann beispielsweise durch

```
substitute := {{String, utils}
               = {oldName       = toString,
                  oldMemberType = METHOD_MEMBER,
                  newName       = %,
                  newMemberType = %
                 },
              {String, utils}
               = {oldName       = length,
                  oldMemberType = METHOD_MEMBER,
                  newName       = getLength,
                  newMemberType = METHOD_MEMBER
                 }
              }
```

angegeben werden, dass der Aufruf der Methode *toString* an einem Objekt vom Typ *utils::String* wegoptimiert wird und Aufrufe der Methode *length* werden durch Aufrufe der Methode *getLength* ersetzt.

4.3.1.2.5 classes

Der Bezeichner „*classes*“ dient der Definition von Klassensignaturen. Diese Konfigurationsmöglichkeit ist für die Klassen gedacht, deren Java-Quellcode nicht vorliegt bzw nicht übersetzbar ist, aber dennoch im Programm benutzt werden. Gängige Beispiele sind Klassen aus der Standardbibliothek, wie etwa die Klasse *java.lang.Exception*:

```
classes := {BaseException,utils}
           ={objectType  = EXCEPTION_OBJ,
             classType   = CLASS,
             classMember = { { memberName  = getMessage,
                               memberType  = METHOD_MEMBER,
                               visibility  = PUBLIC,
                               returnValue = {className = String,
                                              namespace = utils,
                                              dim = 0}
                             }
                           }
            }
```

Das Paar {BaseException,utils} beschreibt die Klasse deren Signatur angegeben werden soll und fungiert als eindeutiger Schlüssel. Eine erneute Definition der Signatur der Klasse *utils::BaseException* würde die vorherige Definition überschreiben. In der Signatur muss mindestens der Objekt-Typ der Klasse angegeben werden. Zusätzlich können optional der Typ der Klasse (CLASS, ENUMERATION oder INTERFACE) und eine Liste der Mitglieder der Klasse spezifiziert werden. Die Angabe der Methodeninterfaces einer Klasse bildet eine wichtige Grundlage für das Auflösen von Methodenaufrufen (siehe 5.3.2). Eine beispielhafte vollständige Konfigurationsdatei ist im Anhang C.5 angegeben.

4.3.2 Definition und Anpassung des Ausgabeformats

Die Ausgabe des Übersetzungsvorganges wird mit Hilfe von *StringTemplate* generiert. *StringTemplate* ist eine in Java geschriebene Template Engine, die es ermöglicht formatierten Quelltext, Webseiten und andere formatierte Ausgaben zu generieren. *StringTemplate* unterstützt die Trennung eines formalen Modells von dessen visueller Aufbereitung [Par04]. Dadurch ist es beispielsweise bei der Entwicklung von Webanwendungen möglich, die Entwicklung der Oberfläche von der Entwicklung der Programmlogik zu trennen. Beim Aufruf des Übersetzers werden die zu verwendende *StringTemplate*-Definitionen mitübergeben, so dass diese einfach auszutauschen sind. Die Namen der einzelnen Templates sind fest vorgegeben, da diese explizit in der letzten Phase des Übersetzungsvorganges aufgerufen werden. Der Inhalt der einzelnen Templates kann aber beliebig modifiziert werden. Eine Veränderung der Templates ist im Allgemeinen nicht notwendig und sollte nur in speziellen Fällen, wie Fehlerkorrekturen oder Aktualisierungen des Übersetzers, durchgeführt werden. In Abschnitt 5.5 werden *StringTemplate*'s und ihre Benutzung innerhalb einer ANTLR-Grammatik beschrieben. Die derzeit verwendeten Templates sind im Anhang C.6 abgebildet.

4.3.3 Kommandozeilenargumente für den Programmstart

Beim Aufruf des Übersetzers müssen verschiedene Einstellungen durch die Übergabe spezieller Parameter getroffen werden. Die verschiedenen Parameter sind in der Tabelle 3 dargestellt.

Tabelle 3: Übergabeparameter für den Aufruf des Übersetzers

Beschreibung	**Kürzel**	**Optional**	**Def.-Wert**
Determinierung der zu übersetzenden Dateien. Zulässig ist sowohl die Angabe einzelner Java-Dateien als auch die Angabe von Verzeichnissen.	-f	nein	-
Spezifikation der Konfigurationsdatei. Zulässig sind nur Dateien mit der Endung „*map*“. Beispiel: −m Config.map	-m	nein	-
Angabe des Zielordners in dem der Zielcode abgelegt werden soll	-d	nein	-
Angabe von nicht zu übersetzenden Dateien. Zulässig ist sowohl die Angabe einzelner Java-Dateien als auch die Angabe von Verzeichnissen.	-i	ja	-
Spezifizierung der zu verwendenden Template-Datei. Zulässig sind nur Dateien mit der Endung „*stg*“.	-t	nein	-
Spezifizierung des zu verwendenden Unicode-Zeichensatz.	-enc	ja	ISO-8859-15
Flag zum Ein-/Ausschalten des Debugging-Modus. Bewirkt ein schlankeres Logging.	-edebug	ja	aus
Flag zum Anschalten von Optimierungen unter Zurhilfenahme von Annotationen (momentan ohne Auswirkung)	-eap	ja	aus

5 Beschreibung des Übersetzungsprozesses

Im folgenden Abschnitt wird der Übersetzungsprozess beschrieben. Der Übersetzungsprozess gliedert sich in verschiedene Phasen. Ein Überblick des Übersetzungsprozesses ist in Abbildung 4 dargestellt.

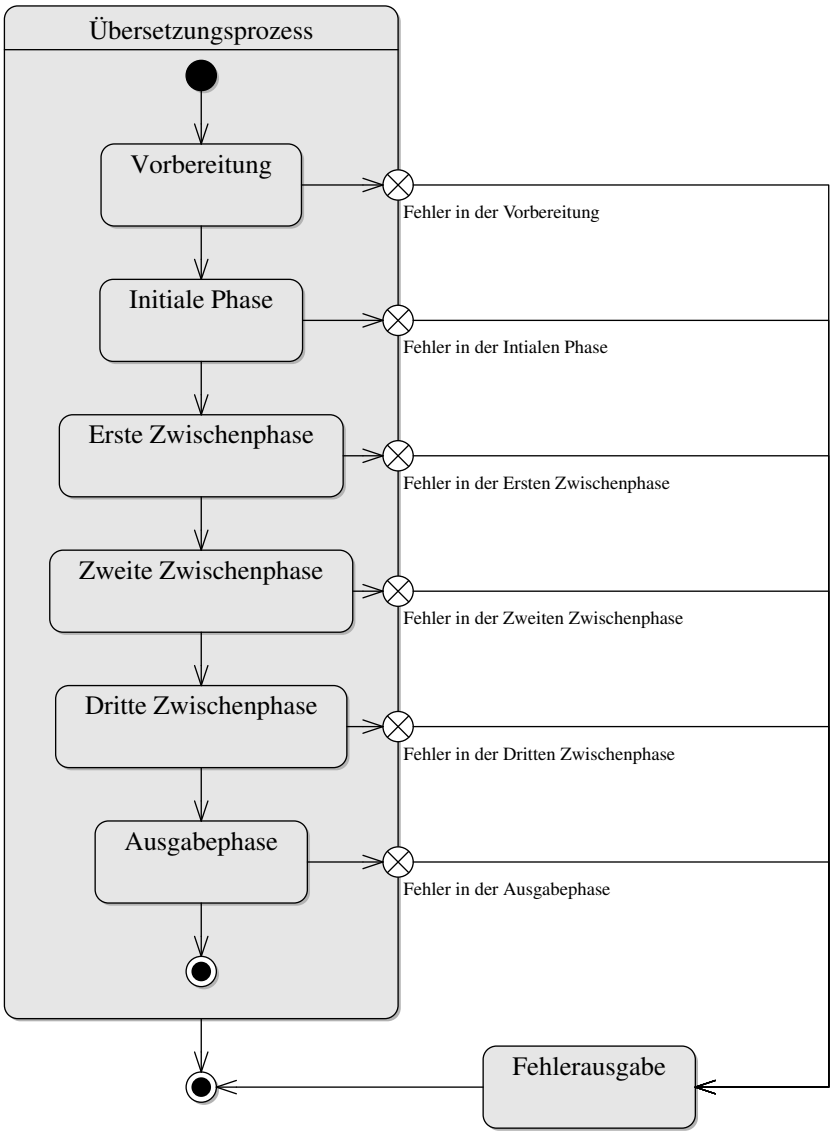


Abbildung 4: Überblick des Übersetzungsprozesses

Der Übersetzungsprozess gliedert sich in insgesamt sechs Abschnitte, die im Folgen-

den beschrieben werden.

5.1 Vorbereitung und Initiale Phase

Zunächst werden die Vorbereitung und die Initiale Phase betrachtet. Diese bilden den Ausgangspunkt eines Übersetzungsvorganges. In Abbildung 5 ist eine kompakte Zusammenfassung dieser beiden Abschnitte dargestellt.

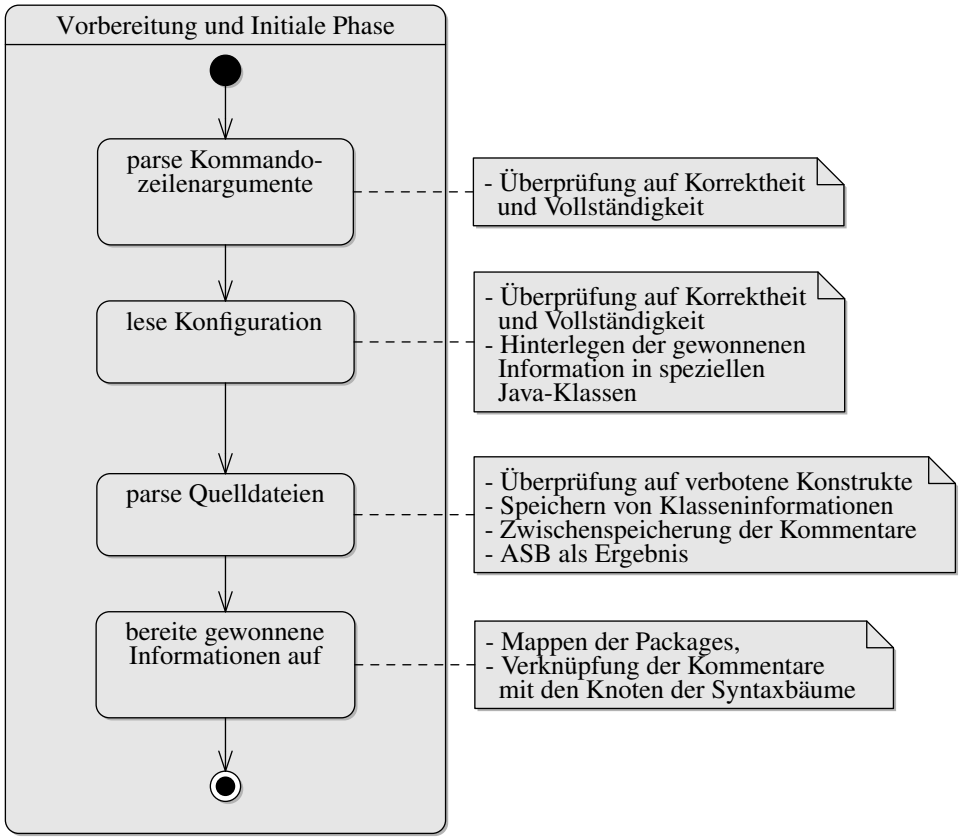


Abbildung 5: Übersetzungsprozess - Vorbereitung und Initiale Phase

Die beiden Zustände „*parse Kommandozeilenargumente*“ und „*lese Konfiguration*“ sind Bestandteil der Vorbereitung, während die Zustände „*parse Quelldateien*“ und „*bereite gewonnene Informationen auf*“ bereits der Initialien Phase zugeordnet sind.

5.1.1 Vorbereitung

Zunächst werden die Bestandteile der Vorbereitung erläutert. Die für einen Programmaufruf zulässigen Parameter und der Aufbau einer Konfigurationsdatei wurden im Abschnitt 4.3 erläutert. Die in der Konfigurationdatei hinterlegten Informationen werden durch die im Anhang C.4 definierte ANTLR-Grammatik in einer Hashmap zwischengespeichert. Diese Hashmap wird durch eine Instanz der Klasse *MappingDecoder* weiterverarbeitet. Dabei werden die entsprechenden Konfigurationseinstellungen aus der Hashmap extrahiert und in passenden Objekten abgelegt, so dass während des Übersetzungsprozesses die Informationen einfach zugänglich sind. Bevor diese Objekte spezifiert und näher erläutert werden, ist die Definition einer Objektstruktur zur Speicherung von Klasseninformationen notwendig.

Die gesammelten Informationen zu einer Klasse, die entweder durch die Angabe einer Signatur in der Konfigurationsdatei oder in einer zu übersetzenden Java-Quelldateien definiert wurde, werden jeweils in einer Instanz der Klasse *ClassDetails* abgelegt. In einer Instanz der Klasse *ClassDetails* werden folgende Informationen abgespeichert:

- Name und Namensraum / Package der Klasse
- Name der Datei in der die Klasse definiert wurde
- der Objekt-Typ von Instanzen dieser Klasse (siehe Klasse *EObjectType* in Abbildung 6)
- der Typ der Klasse (siehe Klasse *EClassType* in Abbildung 6)
- die Vaterklassen dieser Klasse
- eine Liste aller Klassenmitglieder (Instanz der Klasse *ClassMemberMap*, die Instanzen der Klasse *ClassMember* enthält), die direkt in dieser Klasse definiert wurden (vgl. Abbildung 6)

In Abbildung 6 sind die Klasse *ClassDetails*, sowie alle anderen zur Abspeicherung von Klasseninformationen verwendeten Klassen abgebildet.

Nachdem die Objektstruktur zur Abspeicherung von Klasseninformationen bekannt ist, können die Objekte spezifiziert werden, in denen die Klasse *MappingDecoder* die Informationen aus der Konfigurationsdatei zwischenspeichert. Dazu werden die im Abschnitt 4.3 vorgestellten Einträge einer Konfigurationsdatei entsprechenden Objekten zugeordnet.

- Das Namensraummapping wird in einer separaten Hashmap hinterlegt. Als Schlüssel dieser Hashmap dient der zu ersetzende Namensraum. Der zugehörige Wert ist der Name des Ersatznamensraum.

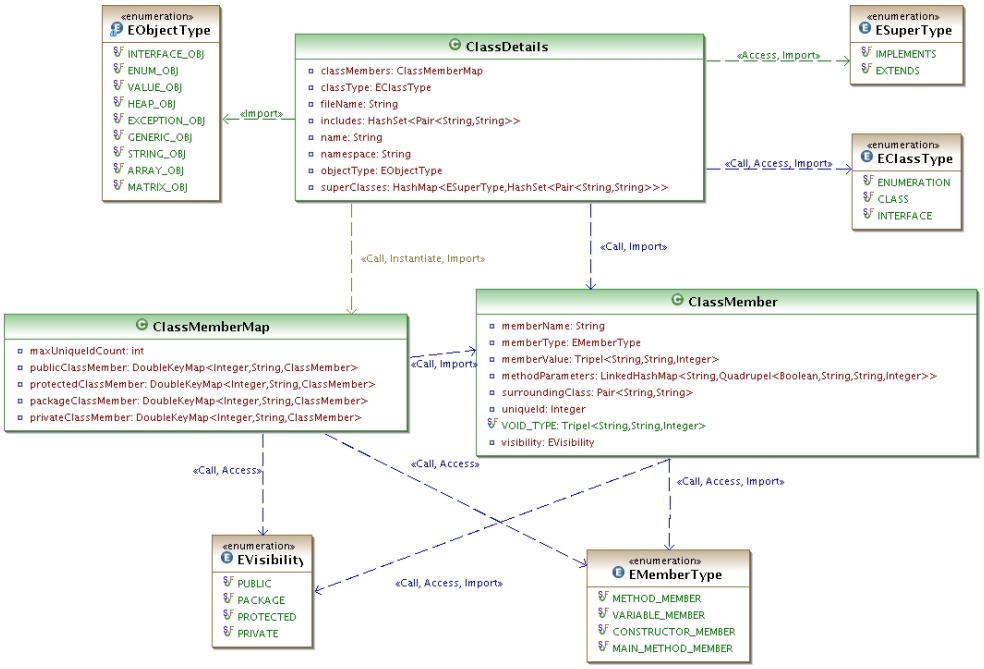


Abbildung 6: Klassenstruktur für interne Speicherung von Informationen über Klassen

- Die definierten Basisklassen werden ebenfalls in einer eigenen Hashmap abgespeichert. Der Schlüssel ist der Objekttyp (eine Enumerationskonstante vom Typ *EObjectType* (vgl. Abbildung 6)). Als Schlüssel wird ein Paar aus Klassenname und Namensraum der Basisklasse hinterlegt.

- Die Informationen bezüglich der Substitution von Klassen werden in einer Hashmap gesichert, die als Schlüssel den vollqualizierten Klassennamen der zu substituierenden Klasse erhält (z. B. *java.lang.Exception*), dem ein Paar aus Klassenname und Namensraum des Substituts zugewiesen wird.

- Die definierten Klassensignaturen werden in Instanzen der Klasse *ClassDetails* (vgl. Abbildung 6) zwischengespeichert.

- Die Ersetzungsregeln für Funktionen werden ebenfalls in einer Hashmap abgelegt. Als Schlüssel fungiert ein Paar aus dem Namen und Namensraum der Klasse, die die Funktionen implementiert. Der zugewiesene Wert enthält eine weitere Hashmap, die die für die Klasse durchzuführenden Ersetzungen enthält.

Falls während des Parsen der Konfigurationsdatei bzw. der Weiterverarbeitung der gewonnenen Informationen ein Problem auftritt (z. B. Unvollständigkeit von Informa-

tionen, falsche Syntax, etc.) wird eine Exception vom Typ *MalformedMappingFileException* geworfen und der gesamte Übersetzungsvorgang wird abgebrochen.

5.1.2 Initiale Phase

Nachdem erfolgreichen Einlesen der Konfiguration werden die übergebenen Quelldateien geparst. Dazu wird für jede zu übersetzende Datei eine Instanz der Klasse *JavaParser* erzeugt. Diese Klasse ist ein aus der ANTLR-Grammatik *InitialePhase* generierter Lexer und Parser für Java-Quelldateien. Die Grammatik basiert auf der Definition von Dieter Habelitz und die Originalversion kann von der ANTLR-Webseite [132] heruntergeladen werden. Die Originalversion beinhaltet die Konstruktion eines abstrakten Syntaxbaumes und eine rudimentäre Fehlerbehandlung. Die ursprüngliche Version verwendet die Optionen *ASTLabelType = CommonTree* und *TokenLabelType = CommonToken*. Da am Ende der Initialen Phase die Kommentare aus dem Quelltext an die Token gebunden werden müssen, wurde eine eigene Tokenklasse *CommonTokenWithCommentList* von *CommonToken* abgeleitet. Diese erlaubt die Zuordnung einer Liste von Kommentaren zu einem Token. Zusätzlich wurde die Klasse *CommonTree* durch ihr Derivat *CustomTree* ersetzt. Die Ersetzung von *CommonTree* ist technischer Natur und wird in Abschnitt 5.3 erneut aufgegriffen.

Anschließend mussten die in Abschnitt 4.1 getroffenen Einschränkungen in die Grammatik eingepflegt werden. Dazu wurde eine spezielle Ausnahmenklasse *NotYetSupportedException* implementiert. Diese Ausnahme wird dann erzeugt, wenn ein gültiges Java-Konstrukt verwendet wurde, dass momentan nicht übersetzt werden kann. Bei jedem gefundenen Fehler wird der Übersetzungsvorgang sofort abgebrochen. Die Fehlerausgabe beinhaltet unter anderem den Dateinamen in der das verbotene Konstrukt verwendet wurde. Des Weiteren werden die betroffene Zeile und eine Beschreibung des Fehlers ausgegeben.

[132] Link: `http://www.antlr.org/grammar/1207932239307/Java1_5Grammars`, zugegriffen am 01.05.2009

Zum Beispiel würde der Übersetzer beim Parsen des Ausdrucks **break** MyLabel;, der etwa in Zeile 10 der Datei *BadLabelExample.java* liegt, aufgrund der Regeldefinition

```
...
statement
    :   ...
    |   BREAK labelId=IDENT? SEMI
        {
        // JAVA-CODE-START
            if(labelId != null)
            {
                throw new NotYetSupportedException
                ("Das Schluesselwort \"break\" in Kombination mit"
                   +" einem Label wird momentan nicht unterstuetzt",
                 this.fileName,
                 $labelId.line);
            }
        // JAVA-CODE-ENDE
        }
    ->  ^(BREAK IDENT?)
    |   ...
    ;
```

die Fehlermeldung

> *„Das Schluesselwort "break" in Kombination mit einem Label wird momentan nicht unterstuetzt (Aufgetreten beim Parsen der Datei BadLabelExample.java in Zeile 10)"*

erzeugen.

Da die Erzeugung eines abstrakten Syntaxbaumes bereits in die Grammatik integriert war, mussten lediglich Informationen zu den zu übersetzenden Klassen gesammelt werden. Dazu zählen Name und Package der beschriebenen Klasse. Zudem müssen Informationen zu den Mitgliedern einer jeden Klasse gesammelt werden. Jedem Klassenmitglied wird eine eindeutige ID zugewiesen, damit nicht die gesamte Signatur eines Mitglieds für dessen Identifizierung benötigt wird. Die ID wird beim Abspeichern von Mitgliederinformationen einer Klasse mitgespeichert und gleichzeitig wird diese ID mit in den abstrakten Syntaxbaum aufgenommen, so dass in späteren Phasen ein Zugriff auf die abgespeicherten Informationen vereinfacht wird. Des Weiteren werden die Attribute der Klassenmitglieder (z. B. die Sichtbarkeit) bestimmt. Zudem wird „*normale*" Klassen, die über keinen explizit angegebenen Konstruktor verfügen, ein Standardkonstruktor definiert. Dieser wird zum Einen in der Mitgliederliste der gesammelten Klasseninformationen gesichert. Zum Anderen wird ein Knoten im ASB erzeugt der den Standardkonstruktor modelliert.

Nach dem erfolgreichen Parsen einer Quelldatei werden die gewonnenen Informationen über die Klasse in einer Map hinterlegt. Diese Map wird im Folgenden durch den Namen *ClassDetailsMap* referenziert. Zusätzlich werden in dieser Map alle in der Konfigurationsdatei angegebenen Klassen mitgespeichert. Als Schlüssel eines jeden Eintrags der *ClassDetailsMap* fungiert ein Paar aus Name und Namensraum der als Wert abgelegten Klasseninformation vom Typ *ClassDetails*.

Eine weitere wichtige Aufgabe der Initialen Phase besteht darin die Kommentarerhaltung zu ermöglichen. Dazu werden die Kommentare beim Parsen der Datei auf einen separaten Kanal abgelegt. Nachdem eine Quelldatei erfolgreich eingelesen wurde, werden die Kommentare mit den Knoten der generierten abstrakten Syntaxbäume verknüpft.

Dazu wird der vom Lexer erzeugte Kurzzeichenstrom durchlaufen. Für jedes Kurzzeichen wird dann überprüft, ob es auf dem separaten Kanal liegt, auf dem die Kommentare beim Lexen hinterlegt wurden. Falls dies der Fall ist wird, der hinterlegte Kommentar zwischengespeichert. Falls das betrachtete Kurzzeichen auf dem Standardkanal abgelegt wurde, wird überprüft, ob an diesem Kurzzeichen Kommentare abgespeichert werden dürfen. Wenn dem Kurzzeichen Kommentare zugewiesen werden dürfen, werden alle zu diesem Zeitpunkt zwischengespeicherten Kommentare an diesem Kurzzeichen abgespeichert und die temporär zwischengespeicherten Kommentare werden gelöscht. Kurzzeichen, die keine Kommentare halten dürfen sind z. B. die Token, die die Sichtbarkeit von Klassenmitgliedern abbilden. Da die Knoten der abstrakten Syntaxbäume die Kurzzeichen enthalten, wird so garantiert, dass die Kommentare über den gesamten Verlauf des Übersetzungsprozesses erhalten bleiben. In Abbildung 7 wird der beschriebene Mechanismus zur Erhaltung von Kommentaren visualisiert.

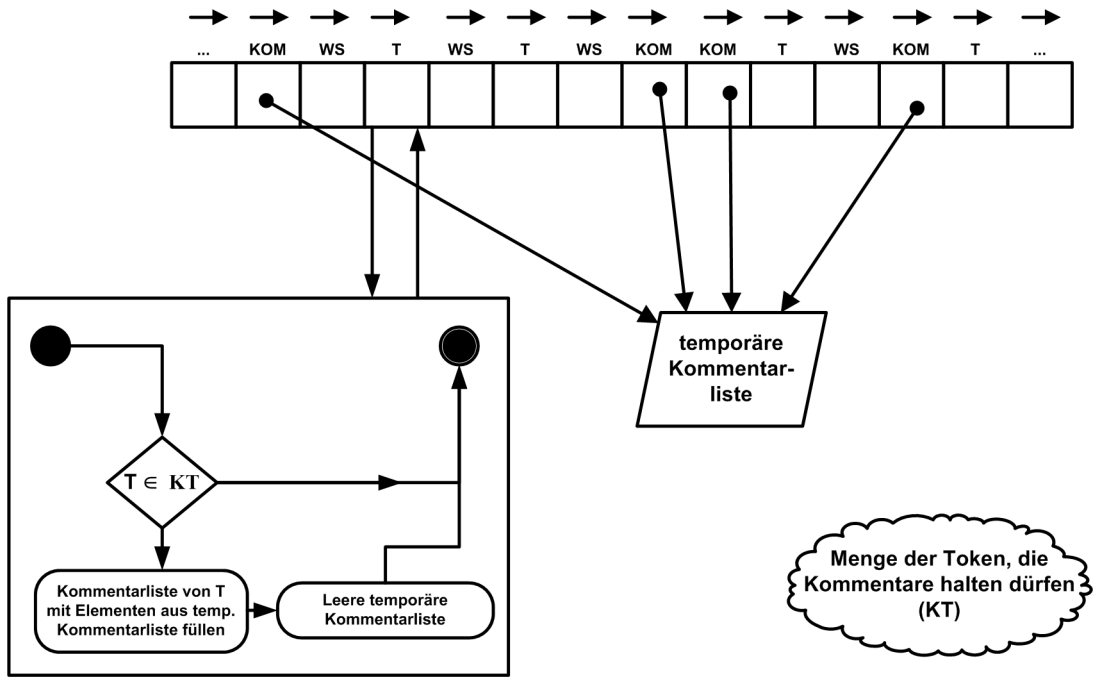


Abbildung 7: Veranschaulichung der Kommentarerhaltung (KZ=Kurzzeichen, die auf dem Standardkanal liegen, WS=Whitespace [Leerzeichen, Tabulator, etc.], KOM=Kurzzeichen für Kommentar)

Des Weiteren wird im Anschluß an die Verknüpfung der Kommentare das in der Konfigurationsdatei definierte Namensraummapping angewandt. Dazu wird die Menge aller Schlüssel der *ClassDetailsMap* hergenommen und das definierte Namensraum-

mapping angewendet. Dabei kann es zu Konflikten kommen. Beispielsweise könnte zweimal die Klasse *A* in unterschiedlichen Packages definiert worden sein. Wenn in der Konfigurationsdatei die beiden Packages auf den gleichen Namensraum gemappt wurden, würde ein Namenskonflikt entstehen. Falls ein solcher Konflikt auftritt wird eine entsprechende Fehlermeldung ausgegeben und der Übersetzungsvorgang wird sofort abgebrochen. Um einen Konflikt dieser Art zu beheben muss der Anwender das Mapping der Namensräume in der Konfigurationsdatei entsprechend anpassen. Falls keine Konflikte gefunden wurden, wird eine Hashmap zurückgeliefert. Diese Hashmap enthält die verwendeten Java-Packagenamen als Schlüssel und die gemappten Namensräume als Wert. Diese werden in der Ersten Zwischenphase zur Aktualisierung der abstrakten Syntaxbäume und der in *ClassDetailsMap* hinterlegten Klasseninformationen benutzt.

5.2 Erste Zwischenphase

Nachdem Abschluss der Initialen Phase wird die Erste Zwischenphase gestartet. Die Ziele dieser Phase sind vor allem das Einpflegen des berechneten Namensraummappings in die ASB's, die Vervollständigung der Klasseninformationen in der *ClassDetails-Map* und die Berechnung einer Klassenhierachie. Des Weiteren werden auf Basis der berechneten Klassenhierachie für jede Klasse ein Objekttyp sowie alle ihre virtuellen Methoden bestimmt. Eine grobe Skizzierung des Ablaufs dieser Phase ist in Abbildung 8 dargestellt.

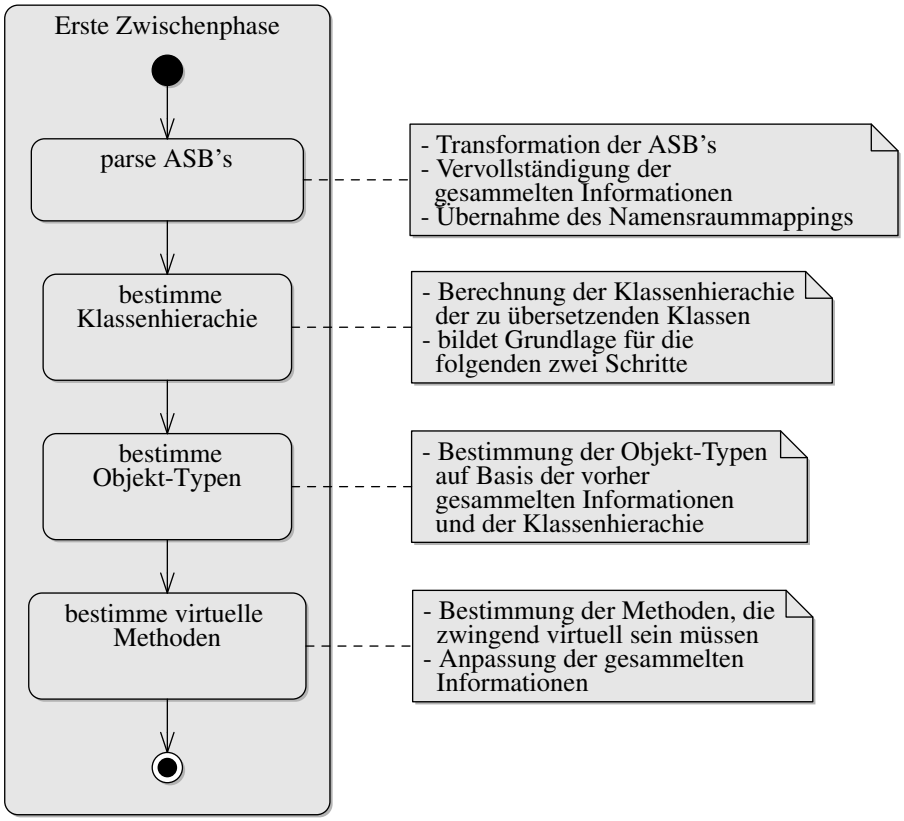


Abbildung 8: Übersetzungsprozess - Erste Zwischenphase

5.2.1 Erneutes Parsen der ASB's

In der initialen Phase wurden die Quelltexte in abstrakte Syntaxbäume transformiert. Dadurch ist es nicht notwendig die Quelldateien erneut einzulesen. Stattdessen können die in der Initialen Phase erzeugten Syntaxbäume geparst werden. Dazu musste zunächst eine Grammatik definiert werden, die einen korrekten abstrakten Syntaxbaum einer Java-Quelldatei beschreibt. Als Ausgangsbasis wurde die Baumgrammatik von Dieter Habelitz verwendet. [133] Die Fehlerbehandlung für aus der Grammatik erzeugte Baumparser wurde erweitert, so dass bei dem Auftreten eines Fehlers ein Trace erzeugt wird, der es ermöglicht im ASB die Fehlerquelle besser zu lokalisieren.

Während des Parsens der ASB's, werden die gesammelten Klasseninformationen aus der Initialen Phase vervollständigt. Zunächst werden die importierten Klassen und die Vaterklassen der Klasse bestimmt. Des Weiteren werden die Typinformationen aller Klassenmitglieder aktualisiert und komplettiert. Bei der Determinierung der Klassennamen wird das berechnete Namensraummapping und die Substitution von Klassen angewendet. Die Bestimmung der in C++ zu inkludierenden Klassen beschränkt sich nicht auf die in den Importanweisungen aufgelisteten Klassen. Es müssen zu den explizit importierten Klassen alle in der betrachteten Klasse direkt verwendeten Klassen inkludiert werden. Das heißt, dass für jeden Parametertypen, jeden Variablentypen, jeden Konstruktor- und jeden statischen Methodenaufruf eine Überprüfung notwendig ist, ob eine weitere Klasse inkludiert werden muss. In dieser Phase werden zunächst nur Vaterklassen und in **throws**-Direktiven angegebene Exceptionklassen, sowie die in den Importanweisungen angegebenen Klassen in einer Liste der zu inkludierenden Klassen hinterlegt. Die Vervollständigung dieser Liste wird in der Zweiten Zwischenphase bewerkstelligt.

Des Weiteren werden alle Knoten, die eine throws-Direktive beschreiben, aus den ASB's entfernt (siehe 4.2.4.2). Das Ergebnis eines jeden Parsevorganges ist zum Einen ein transformierter Syntaxbaum und zum Anderen neue Informationen über die modellierte Klasse, die für die Aktualisierung der *ClassDetailsMap* verwendet werden.

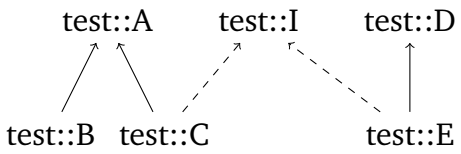
[133] Link: `http://www.antlr.org/grammar/1207932239307/Java1_5Grammars`, zugegriffen am 01.05.2009

5.2.2 Bestimmung der Klassenhierachie und der Objekttypen

Nachdem alle Syntaxbäume erfolgreich geparst und transformiert wurden, wird eine Klassenhierachie aller in der *ClassDetailsMap* abgespeicherten Klassen erzeugt. Die Klassenhierachie wird für die Bestimmung der Objekttypen der Klassen verwendet. Des Weiteren bildet sie eine Grundlage für das Auflösen von Funktionsnamen und Klassenvariablen. Die berechnete Klassenhierachie wird in einer Instanz der Klasse *ClassHierachy* abgespeichert. Diese Instanz enthält alle in der Intialen Phase eingelesenen und zusätzlich die in der Konfigurationsdatei spezifizierten Klassen in Form von Instanzen der Java-Klasse *ClassHierachieNode*. Für jeden Klassenknoten werden alle direkten Vater- und Kindsklassen bestimmt. Zum Beispiel wird für die Java-Klassen aus dem Package *test* mit den Signaturen

```
public class A                          { ... }
public class B extends A                { ... }
public class C extends A implements I   { ... }
public interface I                      { ... }
public abstract class D                 { ... }
public class E extends D implements I   { ... }
```

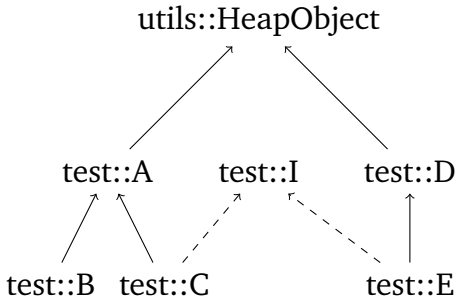
die Klassenhierachie



bestimmt. Es wird keine Klassenhierachie im Stil von Java berechnet. Ansonsten müssten alle Klassen explizit oder implizit von *Object* abgeleitet werden. [134] Auf Basis der berechneten Klassenhierachie und der Informationen aus der *ClassDetailsMap* werden allen Klassen Objekttypen zugeordnet. Dabei wird für jede Klasse, die in der Klassenhierachie keine Vaterklasse besitzt oder nur Schnittstellen implementiert, nachgeschaut, ob dieser Klasse bereits ein Objekttyp zugeordnet ist (z. B. durch die Angabe in der Konfigurationsdatei). Falls der Klasse ein Objekttyp zugeordnet ist, wird geprüft, ob eine Basisklasse für den Objekttypen in der Konfiguration angegeben wurde. Wenn dies der Fall ist, wird der Klasse, sofern sie nicht die Basisklasse selbst darstellt, die entsprechende Vaterklasse zugeordnet. Des Weiteren erhält sie den Objekttypen der Vaterklasse. Falls die durch den betrachteten Wurzelknoten modellierte Klasse noch keinen Objekttypen besitzt, wird der Klasse der Objekttyp Heap-Objekt zugeordnet. In einer Weiterentwicklung des Übersetzers wäre der Einbau einer Logik zur Unterscheidung, ob Instanzen einer Klasse Heap-Objekte oder Value-Objekte sein sollen, sinnvoll. Nachdem der Objekttyp für einen Wurzelknoten der Klassenhierachie bestimmt wurde, werden die Objekttypen der Subklassen der durch den Knoten modellierten Klasse angepasst. Das heißt, dass alle direkten und indirekten Subklassen den selben Objekttypen zugewiesen bekommen wie die durch den Wurzelknoten repräsentierte Klasse.

[134] vgl. [Fla98]

Unter der Annahme, dass für keine der Klassen *A, B, C, D, E* ein expliziter Objekttyp und als Basisklasse für Heap-Objekte die Klasse *utils::HeapObject* in der Konfiguration definiert wurden, ergibt sich für das vorherige Beispiel folgende modifizierte Klassenhierachie:



5.2.3 Bestimmung zwingend virtueller Methoden

Nachdem die Klassenhierachie der zu übersetzenden Klassen erfolgreich bestimmt wurden ist, können die zwingend virtuellen Methoden einer jeden Klasse bestimmt werden. Um das eigentliche Problem verstehen zu können, ist es sinnvoll sich zuerst einmal dem Begriff der virtuellen Methode zu widmen. Eine virtuelle Methode ist eine Methode, von der eine Basisklasse erwartet, dass eine ihrer abgeleiteten Klassen diese überschreibt. Die Definition von virtuellen Methoden ermöglicht dynamische Bindung. Das heißt, dass erst zur Laufzeit anhand des dynamischen Datentypen entschieden werden muss, welche Methode aufgerufen werden soll. [135] In Java, sowie in anderen objektorientierten Programmiersprachen wie z. B. Smalltalk-80, sind alle Methoden virtuell, was dem Programmierer die Entscheidung, ob eine Methode virtuell deklariert werden muss oder nicht, abnimmt. In C++ hingegen sind laut Standard alle Methoden zunächst nicht virtuell. [136] Erst durch das Schlüsselwort **virtual** wird eine äquivalente Funktionalität wie in Java erreicht. Ein einfacher Ansatz wäre alle Methoden, die nicht statisch sind als virtuell zu deklarieren. Der Nachteil dieses Ansatzes liegt darin, dass ein Aufruf einer virtueller Methoden zur Laufzeit mit einem Suchen dieser in der Tabelle der virtuellen Methoden der jeweiligen Klasse verbunden ist. [137] Somit würde eine Übersetzung von einem Programm mit vielen Funktionsaufrufen unter Umständen [138] zu ineffizienten C++-Code führen. [139] Deshalb ist es sinnvoll nur die Methoden als virtuell zu deklarieren, für die es tatsächlich zwingend ist.

Dazu werden nun einige Vorüberlegungen getroffen, um den Algorithmus zur Bestimmung der virtuellen Methoden so einfach wie möglich gestalten zu können. Da eine Schnittstelle eine rein virtuelle Klasse darstellt, was heisst, dass alle ihre Methoden virtuell sind, brauchen diese im Algorithmus nicht betrachten zu werden. Java-Enumerations brauchen ebenfalls nicht betrachtet werden, da sie keine anderen Klassen erweitern, sondern lediglich Interfaces implementieren dürfen. Somit müssen in dem Algorithmus nur „*nomale*“ und abstrakte Klassen betrachtet werden. Des Weitern müssen alle die Methoden einer Klasse virtuell deklariert werden, die

(i) nicht von einer direkten oder indirekten Vaterklasse bereits als virtuell deklariert wurden und

(ii) von einer direkten oder indirekten Subklasse der Klasse überschrieben werden.

Der Grund für die Bedingung (i) liegt darin, dass eine überschriebene Methode einer als virtuell markierten Methode einer Vaterklasse automatisch wieder virtuell ist. [140] Die zweite Bedingung ist zwingend notwendig um garantieren zu können, dass zur Laufzeit eines übersetzten Programmes die korrekte Methode aufgerufen wird.

[135] vgl. [LLM06, Seite 657]
[136] vgl, [LLM06, Seite 666]
[137] vgl. [Str98, Seite 40]
[138] Die meisten Compiler können virtuelle Funktion, die nicht auf einen Zeiger oder eine Referenz angewendet werden zur Übersetzungszeit auflösen.
[139] vgl. [AH96]
[140] vgl. [iso98, Seite 169]

Bevor der Algorithmus für die Bestimmung der virtuellen Methoden angegeben werden kann, müssen zunächst die in ihm verwendeten Prädikate und Definitionen vorgestellt werden.

Definition „potentiell virtuell“:

Eine Methode wird als „*potentiell virtuell*“ bezeichnet, wenn sie von Subklassen überschreibar ist. Das heißt, dass sie folgende Eigenschaften erfüllt:

(a) Sie ist nicht statisch
(b) Sie ist nicht final
(c) Sie ist nicht **private**

Prädikate isAbstractClass und isNormalClass:

Die Prädikate *isAbstractClass* und *isNormalClass* geben an, ob eine Klasse eine abstrakte bzw. eine „*normale*“ Klasse ist.

Prädikate hasParent und hasChild:

Die Prädikate *hasParent* und *hasChild* determinieren, ob eine Klasse eine Vaterklasse bzw. eine Subklasse besitzt.

Prädikat potVirtual:

Das Prädikat *potVirtual* bestimmt, ob eine explizit in der Klasse definierte Methode potentiell virtuell ist.

Funktion defMeths:

Die Funktion *defMeths* bestimmt, welche Methoden explizit in einer Klasse definiert wurden sind.

Funktion directSubClasses:

Die Funktion *directSubClasses* liefert die direkten Subklassen einer Klasse zurück.

Auf Basis dieser Definitionen, wird nun der Algorithmus zur Bestimmung der virtuellen Methoden definiert.

Algorithmus 1 (Bestimmung virtueller Methoden)**.**

determineVirtMethStart( )

02. $K := \{k \mid k \in$ *ClassDetailsMap* $\wedge\ (isAbstract(k) \vee isNormalClass(k))\}$
03. ***foreach*** $k \in K$ *mit* $hasParent(k) = false \wedge hasChild(k) = true$
04. $\quad determineVirtMethRek(k,\ \emptyset)$

determineVirtMethRek(Class currClass, MethSet meths)

06. $M_{currClass} := \{m \in defMethods(currClass) \mid potVirtual(m) = true\}$
07. $M_{currClassAll} := M_{currClass} \cup meths$
$M_{methsSubclasses} := \emptyset$
09. ***foreach*** $subclass \in directSubClasses(currClass)$
10. $\quad M_{methsSubclasses}$ *:=* $M_{methsSubclasses}$
$\quad\ \cup\ determineVirtMethRek(subclass,\ M_{currClassAll})$
11. ***end***
$ret := \emptyset$
foreach $currMeth \in M_{methsSubclasses}$
14. $\quad$ ***if*** $currMeth \notin meths \wedge currMeth \in M_{currClass}$
15. $\quad\ $ *virtualisiere Methode* $currMeth$ *in Klasse* $currClass$
16. $\quad$ ***if*** $currMeth \in meths$
17. $\quad\ ret := ret \cup \{currMeth\}$
end
return ret

Der präsentierte Algorithmus wählt zunächst alle „*normalen*“ und abstrakten Klassen aus (Zeile 2). Anschließend werden ausgehend von allen Klassen, die durch einen Wurzelknoten in der Klassenhierachie repräsentiert werden und mindestens eine Subklasse besitzen, die virtuellen Methoden rekursiv ermittelt (Zeile 3 und 4). Für die Ermittlung wird zunächst die Menge aller potentiell virtuellen Methoden der eigenen Klasse bestimmt (Zeile 6). Die bestimmte Menge wird mit allen potentiell virtuellen Methoden der direkten und indirekten Vaterklassen vereinigt (Zeile 7). Anschließend werden rekursiv alle potentiell virtuellen Methoden aller Subklassen der betrachteten Klasse bestimmt und zwischengespeichert (Zeile 9 und 10). Abschließend wird für jede potentiell virtuelle Methode einer Subklassen überprüft, ob diese Methode in der aktuell betrachteten Klasse ebenfalls definiert wurde (Zeile 14). Falls die betrachtete Methode in der aktuell betrachteten Klasse definiert wurde, wird diese in der Klasse als virtuell markiert (Zeile 15). Falls die betrachtete Methode nicht in der betrachteten Klasse, aber in einer Vaterklasse der betrachteten Klasse definiert wurde, wird diese der Rückgabemenge aller noch zu virtualisierenden Methoden hinzugefügt (Zeile 16 und 17).

Die durch den Algorithmus bestimmten virtuellen Methoden werden in der Zweiten Zwischenphase in den Syntaxbäumen entsprechend markiert. Mit der Berechnung der zwingend virtuellen Methoden endet die Erste Zwischenphase.

5.3 Zweite Zwischenphase

Das übergeordnete Ziel der Zweiten Zwischenphase ist es die abstrakten Syntaxbäume mit möglichst vielen Typinformationen anzureichern. Des Weiteren werden zusätzlich folgende Tätigkeiten durchgeführt:

- Bestimmung und Markierung der Klassenmitglieder, die **inline** zu deklarieren sind
- Markierung der virtuellen Mitglieder einer Klasse
- Bestimmung und Markierung von Stringkonkatenationen
- Behandlung des Nullpointers
- Bestimmung von Freundesklassen
- Bestimmung der Using-Direktiven

Im Folgenden wird zunächst die Realisierung der Anreicherung der abstrakten Syntaxbäume mit Typinformationen beschrieben. Anschließend werden die anderen Aufgabenstellungen und ihre Lösungen diskutiert.

5.3.1 Anreicherung der ASB mit Typinformationen

Der ursprüngliche Grund für die Anreicherung der abstrakten Syntaxbäume war, dass eine Grundlage für die Detektion von Zyklen (siehe Abschnitt 7) geschaffen werden sollte. Im Laufe der Arbeit wurde klar, dass ohne diese Informationen nicht einmal korrekte Programme generiert werden können. Der Grund dafür ist die Möglichkeit in C++, auf verschiedene Arten auf die Bestandteile eines Objekts zugreifen zu können (vgl. Abschnitt 2.2.4.2). Das heißt, dass für jeden Methodenaufruf, jeden Variablenzugriff, usw. bestimmt werden muss, von welcher Art die linke Seite des Ausdrucks ist. Es muss beispielsweise der Objekttyp einer Instanzvariablen bestimmt werden, auf der ein Methodenaufruf durchgeführt wird, um in der letzten Phase den korrekten Zugriffsoperator ausgeben zu können. Deshalb müssen für alle verwendeten Variablen, Methoden, Konstruktoren und Literale, sowie Methodenaufrufe, Memberabfragen und weitere Ausdrücke spezielle Informationen bestimmt werden. Für jeden Ausdruck werden die bestimmten Informationen in einer Instanz vom Typ *LeftSideValue* abgespeichert. Diese Instanz wird anschließend an den Knoten im ASB gebunden, der den betrachteten Ausdruck repräsentiert. Die Instanzen der Klasse *LeftSideValue* beinhalten folgende Informationen:

- den Namen der durch den Knoten repräsentierten Variablen, Methode, usw.
- den Typ der Variablen, des Rückgabewertes der Funktion, usw.
- die Dimension des Typs der Variablen, des Rückgabewertes der Funktion, usw.

- den Objekt-Typen der Variablen, des Rückgabewertes der Funktion, usw.
- den Ausdruckstypen (z.B. statische Funktion, lokale Variable, Literal, usw.)
- ggf. die eindeutige ID der Variablen, der Methode, usw.

Eine Verallgemeinerung des Prozesses zur Anreicherung der ASB's mit Typinformationen ist in Abbildung 9 dargestellt.

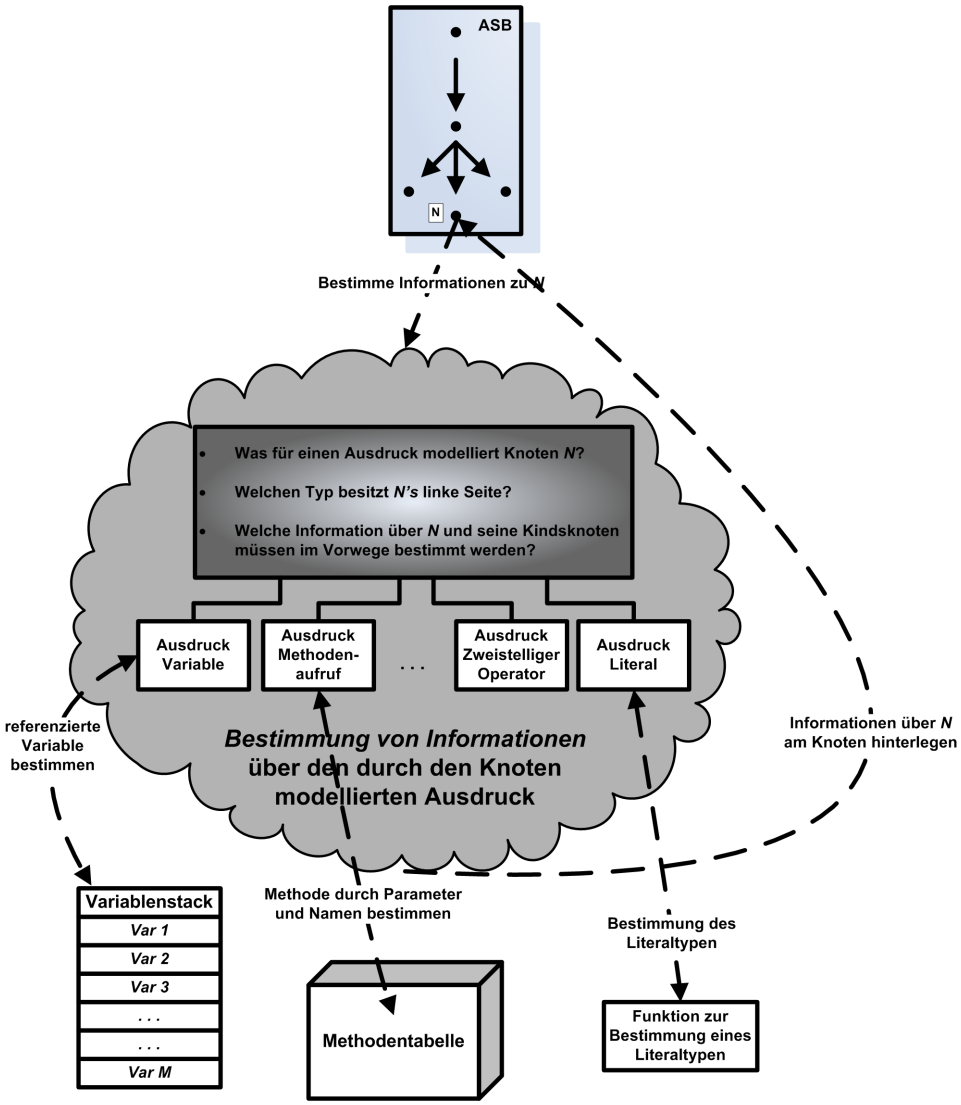


Abbildung 9: Verallgemeinerung des Prozesses zur Anreicherung eines ASB's mit Typinformationen

Der Variablenstack aus Abbildung 9 wird während des Parsens eines ASB's erstellt und aktualisiert. Dazu werden für eine deklarierte Variable zunächst Informationen in einer Instanz von *LeftSideValue* gespeichert. Diese Instanz wird dann auf den Variablenstack gepackt. Beim Verlassen des Gültigkeitsbereiches einer Variablen wird diese von dem Variablenstack entfernt. Zum Beispiel wird für die Klasse

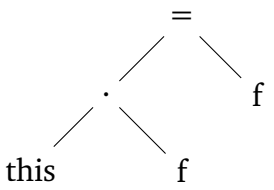
```
    public class X {
        int x;      // eindeutige ID = 1
        int y;      // eindeutige ID = 2
        float f;    // eindeutige ID = 3

        void fun(int f) {
(*)         this.f = f;
        }
    }
```

bis zur Zeile (*) folgender Variablenstack aufgebaut:

Top of Stack
{LOCAL_VARIABLE, f, (null, int), 0, -}
{MEMBER_VARIABLE, f, (null, float), 0, 3}
{MEMBER_VARIABLE, y, (null, int), 0, 2}
{MEMBER_VARIABLE, x, (null, int), 0, 1}
Bottom of Stack

Beim Verlassen des Rumpfes der Funktion *fun* wird das oberste Element *f* vom Variablestack entfernt.

Mit Hilfe des Variablenstack's können Variablennamen aufgelöst werden. Befindet sich der Übersetzer zum Beispiel in Zeile (*), muss er folgenden Teilbaum parsen:



Da ANTLR LL-Parser erzeugt, wird zunächst der Knoten *this* betrachtet. Für den *this*-Knoten werden die Typinformationen bestimmt, d. h. es wird eine Instanz von *LeftSideValue* berechnet, die eine Instanz dieser Klasse modelliert. Anschließend wird *f* auf Basis der Informationen der linken Seite (des *this*-Knoten's) gesucht. Das bedeutet, dass nach dem Klassenmitglied mit dem Namen *f* auf dem Variablenstack gesucht wird. Dazu wird der Variablenstack solange durchlaufen, bis eine entsprechende Klassenvariable gefunden wurde. Wird keine Variable mit dem entsprechenden Namen gefunden muss überprüft werden, ob die gesuchte Variable in einer Vaterklasse deklariert wurde. Wenn die entsprechende Variable schließlich bestimmt wurde, werden die gefundenen Informationen an dem Knoten hinterlegt. Die Bestimmung der Informationen für den anderen Knoten *f* läuft analog. Einziger Unterschied ist die Berücksichtigung lokaler Variablen, da *f* nicht auf **this** angewendet wurde.

Zusätzlich zu den verschiedenen Variablen müssen Typinformationen zu Operationen wie '+', '*', '≫', usw. bestimmt werden. Zum Beispiel muss für den Methodenaufruf `fun(10+10);` bestimmt werden welchen Typ der Ausdruck `10+10` hat. Dies ist notwendig um Methodennamen korrekt auflösen zu können (siehe 5.3.2). In diesem Fall würde für den Ausdruck `10+10` der Typ **int** bestimmt werden. Eine Besonderheit

stellen Ausdrücke wie `10+"A"` dar, da der Typ des Ausdrucks *String* ist. Diese Ausdrücke werden im ASB transformiert, so dass Stringkonkatenationen eindeutig identifizierbar sind.

Für weitere Ausdrücke wie Casts, Literale, usw. wird ebenfalls ein passender Typ bestimmt. Des Weiteren muss beachtet werden, dass statische Zugriffe auftreten können bei denen ein im Aufrufstring enthaltener Namensraum noch nicht gemappt wurde. Dieses Mapping muss analog zu den Ergebnissen des Mappings der Intialen Phase durchgeführt werden. Des Weiteren müssen alle Methodenaufrufe aufgelöst werden.

5.3.2 Auflösung von Methodenaufrufen

Besonders schwierig ist die Auflösung von Methodenaufrufen. Grund hierfür sind die Möglichkeit der Überladung und des Überschreiben von Methoden und Konstruktoren. Dies bedeutet, dass im Vorwege bestimmt werden muss, welche Methoden in einer Klasse zur Verfügung stehen.

Zunächst werden dazu die Begriffe Typtupel, eine Ordnung $\preceq$ auf Typtupeln und die Wohlordnung einer Menge von Typtupeln definiert. [141]

Definition Typtupel:

Sei f eine Funktion mit den Parametern $p_1, \ldots, p_n$. Des Weiteren sei $Typ(p_i)$ mit $i \in \{1, \ldots, n\}$ die Typbezeichnung des Parameters p_i. Eine Typbezeichnung ist entweder ein primitiver Datentyp oder eine Klasse. Zudem sei durch $Imp(p_i)$ die Menge der Typbezeichner definiert, in die $Typ(p_i)$ implizit umgewandelt werden kann. Dann ist für f das Typtupel t_f durch das Kreuzprodukt $Imp(p_1) \times \cdots \times Imp(p_n)$ definiert.

Definition der Ordnung $\preceq$ auf Typtupeln:

Sei T_{all} die Menge aller möglichen Typtupel und $\preceq \subseteq T_{all} \times T_{all}$. Des Weiteren seien $t_1 := X_1 \times \cdots \times X_n$ und $t_2 := Y_1 \times \cdots \times Y_m$ Typtupel.
Es gilt genau dann $t_1 \preceq t_2$, wenn

(i) $m = n$ und

(ii) $X_i \subseteq Y_i$ für alle $i \in n$

Man bezeichnet t_1 und t_2 als *unabhängig*, falls

(i) $m \neq n$ oder

(ii) $X_i \not\subseteq Y_i$ und $Y_i \not\subseteq X_i$ für mindestens ein $i \in n$

Definition der Wohlordnung einer Menge von Typtupeln:

Seien $t_1 := X_1 \times \cdots \times X_n$ und $t_2 := Y_1 \times \cdots \times Y_m$ wiederum Typtupel. Sie werden als *wohlgeordnet* bezeichnet, genau dann wenn

(i) $t_1 \preceq t_2$ oder

[141] Die Idee zu den Definitionen basiert auf [DGC95a]

(ii) t_1 und t_2 unabhängig sind.

Auf Basis der Definitionen wird im Folgenden erläutert, wie die Methodenauflösung auf Basis von Typtupeln funktioniert und welchen Einschränkungen sie unterliegt. Es müssen für alle Methoden Typtupel bestimmt werden, die alle möglichen Typkombinationen abbilden, die zu einem Aufruf der Funktion führen. Aufgrund der Komplexität dieses Problems wird für die Menge von gleichnamigen Funktionen, die über eine identische Anzahl an Parameteren verfügen und in der gleichen Klasse verfügbar sind, angenommen, dass die Typtupel aller dieser Funktionen untereinander wohlgeordnet sind.
Beispielsweise werden für die Funktionen mit der Signatur

```
void fun(boolean b, int i, float f)
void fun(boolean b, float f, double d)
```

die Typtupel

$$Tup_1 := \{\textbf{boolean}\} \times \{\textbf{char}, \textbf{byte}, \textbf{short}, \textbf{int}\} \times \{\textbf{char}, \textbf{byte}, \textbf{short}, \textbf{int}, \textbf{float}\}$$

bzw.

$$Tup_2 := \{\textbf{boolean}\} \times \{\textbf{char}, \textbf{byte}, \textbf{short}, \textbf{int}, \textbf{float}\} \times \{\textbf{char}, \textbf{byte}, \textbf{short}, \textbf{int}, \textbf{float}, \textbf{double}\}$$

bestimmt.

Nachdem die Typtupel für jede Methode und jeden Konstruktor einer Klasse bestimmt wurden, wird für jede Klasse eine Methodentabelle berechnet. Diese Methodentabelle enthält als Schlüssel Paare aus dem Namen der Methode und der Anzahl der Parameter. Jedem dieser Schlüssel wird eine geordnete Liste der Methoden der Klasse zugeordnet, die Namen und Parameteranzahl des Schlüssel entsprechen. Eine geordnete Methodenliste ist eine Liste $Ml_{Name,\ \#Parameter} := (m_1, \ldots, m_{\#Parameter})$ für die gilt, dass für die Liste der Typtupel der Methoden $T_{Ml} := (t_1^{m_1}, \ldots, t_n^{m_n})$ gilt, dass entweder $t_i^* \preceq t_j^*$ oder t_i^* und t_j^* unabhängig sind für alle $i \leq j$.

Die Methodentabellen werden für alle Klassen vor dem Parsen der abstrakten Syntaxbäume in der Zweiten Zwischenphase berechnet und in Form einer statischen Variable hinterlegt, so dass in allen Parsevorgängen der Phase die gleichen Methodentabellen zur Verfügung stehen. Im Verlauf de Phase kann durch die Angabe einer Klasse, dem Namen der aufzurufenden Methode dieser Klasse und den an die Methode übergebenen Parametern genau bestimmt werden, welche Methode aufgerufen wurde. Dazu wird die geordnete Liste für das Paar (Name der Methode, Anzahl Parameter) aus der Methodentabelle der entsprechenden Klasse solange durchlaufen, bis eine passende Methode gefunden wurde.

Durch der getroffenen Annahme wird die Menge der übersetzbaren Programme geringfügig eingeschränkt, so dass eine Übersetzung von Beispiel 13 momentan nicht möglich ist.

Während des Parsens der ASB's müssen zur Auflösung der Methodenaufrufe mit Hilfe der berechneten Methodentabellen, für alle Argumente des Aufrufs Typinformationen bestimmt werden, die für den Abgleich mit den berechneten Typtupeln verwendet werden. Des Weiteren muss der Name der Methode und die Klasse, in der die

Beispiel 13 Nicht zulässige Methodenüberladung

```
public class A {
    // Funktion 1
    public void fun(int i, float f) {
        ...
    }
    // Funktion 2
    public void fun(float f, int i) {
        ...
    }
}
```

$Tup_1 = \{$**char**, **byte**, **short**, **int**$\} \times \{$**char**, **byte**, **short**, **int**, **float**$\}$ (*Tupel für Funktion 1*)
$Tup_2 = \{$**char**, **byte**, **short**, **int**, **float**$\} \times \{$**char**, **byte**, **short**, **int**$\}$ (*Tupel für Funktion 2*)

Es ist zu erkennen, dass weder $Tup_1 \preceq Tup_2$ noch $Tup_2 \preceq Tup_1$ gilt und Tup_1 und Tup_2 nicht unabhängig sind. Somit sind Tup_1 und Tup_2 untereinander nicht wohlgeordnet.

Methode definiert sein soll, bestimmt werden. Die Anreicherung der ASB's mit Typinformationen ist derart komplex, dass eine detailliertere Darstellung den Rahmen der Diplomarbeit sprengen würde. Stattdessen sei auf die Grammatikdefinition für den Parser der Zweiten Zwischenphase und die Klassen *MethodRange*, *MethodResolver* und *MethodTable* auf der beiliegenden CD verwiesen.

5.3.3 Weitere Aufgaben der Zweiten Zwischenphase

Neben der Anreicherung der Knoten der ASB's mit Typinformationen werden die in der Ersten Zwischenphase bestimmten virtuellen Methoden im ASB markiert. Dazu wird beim Parsen eines Teilbaumes, der eine Methode repräsentiert, überprüft, ob diese in der vorherigen Phase als zwingend virtuell deklariert wurde. Wenn sie zwingend virtuell ist, wird die im ASB der Methode zugeordnete Liste von Schlüsselwörtern um einen Knoten erweitert, der die Virtualität der Funktion symbolisiert.

Zudem wird für jede Methode überprüft, ob diese **inline** zu deklarieren ist. Das vorläufige Inline-Kriterium besagt, dass genau die Methoden **inline** deklariert werden, die weniger als fünf Anweisungen enthalten. [142] Zur Überprüfung des Kriteriums werden beim Parsen eines Teilbaumes, der den Methodenrumpf der zu untersuchenden Methode modelliert, die Anzahl der Anweisungen gezählt. Die Markierung der Methode als **inline** wird analog zu der Markierung der Virtualität bewerkstelligt.

Schließlich sammelt der Parser während der Zweiten Zwischenphase Informationen darüber, welche Freundschaftsbeziehungen unter den verschiedenen Klassen bestehen müssen und welche Using-Direktiven im Zielcode verwendet werden können. Für die Using-Direktiven werden zunächst alle innerhalb der Klasse explizit und implizit verwendeten Namensräume bestimmt. Anschließend wird überprüft, ob Konflikte zwischen den verwendeten Namensräumen in Form gleichnamiger Klassen existieren. Falls dies nicht der Fall ist wird für jeden Namensraum eine Usingdirektive in Form eines Knotens im ASB angelegt. Falls Konflikte existierten, wird eine Untermenge der Using-Direktiven bestimmt, die keine Konflikte vorweist, wobei immer der Namensraum der eigenen Klasse in der berechneten Teilmenge enthalten sein muss.

Da die Bestimmung aller notwendigen Freundschaftbeziehungen mit vollkommener Sicherheit erst nach dem Parsen aller ASB's abgeschlossen ist, wird in jedem ASB ein Knoten eingeschleust, der erst nach Abschluss des Parsens aller ASB's in der Zweiten Zwischenphase modifiziert wird. Hierfür werden notwendige Freundschaftsbeziehungen während des Pasevorganges protokolliert und im Anschluß werden Knoten erzeugt, die die Beziehungen in einem ASB modellieren. Diese werden an die vorher eingeschleusten Knoten angehängt.

[142] vgl. [Bre, Seite 7]

5.4 Dritte Zwischenphase

Nachdem in der Zweiten Zwischenphase die Knoten der abstrakten Syntaxbäume mit Informationen angereichert wurden, werden diese Informationen in dieser Phase unter anderem zur Anwendung der in der Konfiguration angegebenen Substitutionen verwendet. In dieser Phase werden zusätzlich noch folgende Arbeiten erledigt:

- Ersetzung von Arraykonstruktoraufrufen durch Erzeugermethoden
- Anfertigung einer Initialisierungsliste für jeden Konstruktor
- Markierung von Konstruktoren, die nur ein Argument enthalten, als **explicit**
- Generierung von Aufrufen der *toString*-Methode von Objekten, die mit einem String konkateniert werden
- Überprüfung der Notwendigkeit der Einbettung von Enumerationkonstanten

Im Folgenden werden die einzelnen Aufgaben beschrieben.

5.4.1 Substitutionen

Die in der Konfiguration angegebenen Substitutionsregeln (siehe Abschnitt 4.3.1.2.4) bilden die Möglichkeit ab, bereits existierende C++-Klassen nicht an die Signatur einer funktional äquivalenten Java-Klasse anpassen zu müssen. Ohne diese Möglichkeit, müsste z. B. die in C++ verwendete Array-Klasse eine Variable mit dem Namen *length* beinhalten. Da die verwendete Array-Klasse dynamische Arrays abbildet, besitzen Arrays keine feste Größe. Die Folge wäre eine veränderbare öffentliche Variable *length*. Stattdessen kann mit Hilfe einer Substitutionsregel erreicht werden, dass jeder Zugriff auf das Feld *length* eines Arrays durch einen Methodenaufruf *getLength()* ersetzt wird. Die Substitution wird in der Regel *primaryExpression* unter Zuhilfenahme der Funktion *findSubstitute* durchgeführt (siehe Codeauschnitt im Anhang C.2).

5.4.2 Ersetzung von Arraykonstruktoraufrufen

In Abschnitt 4.2.3.5 wurde erläutert, warum die Generierung von Methoden zur Initialisierung von Arrays notwendig ist. Diese Methoden werden in dieser Phase generiert und in die abstrakten Syntaxbäume der jeweiligen Klassen eingebunden. Dazu werden für jede direkte Arrayinitialisierung folgende Informationen gesammelt:

- Initialisierung in einem statischen oder in einem nicht statischen Kontext
- alle Initialisierungswerte mit dem Index des zu initialisierenden Teilfeldes
- die zur Initialisierung des Feldes verwendeten Variablen

Auf Basis der gesammelten Informationen wird eine Funktion in Form eines Knotens im ASB generiert. Der Name der Funktion wird generiert. Diese Funktion wird genau dann statisch deklariert, wenn die ursprüngliche Initialisierung ebenfalls in einem statischen Kontext stattgefunden hat. Die generierte Funktion erhält alle für die Initialisierung verwendeten Variablen als Parameter, so dass keine Anpassung der Initialisierungen der einzelnen Teilfelder notwendig ist. Anschließend werden im Rumpf der Initialisierungsfunktion die einzelnen Felder des Arrays gemäß der Definition initialisiert. Das erzeugte Array ist der Rückgabewert der Funktion. Abschließend wird im abstrakten Syntaxbaum die direkte Arrayinitialisierung durch den passenden Methodenaufruf ersetzt.

5.4.3 Anfertigung der Initialisierungslisten

Für die Anfertigung der Initialisierunglisten im ASB wird zunächst der initiale Wert aller nicht statischen Klassenvariablen bestimmt. Falls explizit kein initialer Wert für eine Klassenvariable angegeben wurde, muss der initiale Wert, wie in Abschnitt 4.2.4.5.1 beschrieben, bestimmt werden. Anschließend wird für alle in der Klasse definierten Konstruktoren zunächst überprüft, ob sie explizit einen Konstruktor der Vaterklasse aufrufen. Falls ein Konstruktor der Vaterklasse aufgerufen wird, wird dieser zuerst in die Initialisierungsliste eingefügt. Anschließend werden alle Variableninitialisierungen an die Initialisierungsliste angehängt und die Initialisierungsliste dem betrachteten Konstruktor im ASB zugewiesen.

5.4.4 Sonstiges

Zusätzlich werden während der Dritten Zwischenphase alle Konstruktoren, die genau einen Parameter besitzen, im abstrakten Syntaxbaum als explizit markiert. Des Weiteren werden alle Stringkonkatenationen dahingehend korrigiert, dass für alle nicht primitiven Datentypen, die nicht vom Typ *String* und Teil einer Stringkonkatenation sind, explizit ihre *toString*-Methode aufgerufen wird. Schließlich werden aus den in Abschnitt 4.2.2.2 erwähnten Gründen Funktionsaufrufe, die direkt auf eine Enumeration-Konstanten angewendet werden, durch Konstruktoraufrufe ersetzt. Gleiches gilt für Stringliterale auf denen Funktionsaufrufe getätigt werden.

5.5 Ausgabephase

Die Ausgabephase umfasst die Transformation der abstrakten Syntaxbäume in C++-Code. Zunächst werden die wichtigsten Bausteine zur Anfertigung von *StringTemplate*'s vorgestellt. Danach wird die Integration von *StringTemplate* in ANTLR beschrieben. Anschließend werden die Anforderungen an die Ausgabeform und das Ausgabeformat präsentiert. Abschließend wird an einem Beispiel die Realisierung der formatierten Ausgabe erläutert.

5.5.1 Beschreibung des Aufbau's von StringTemplate's

In Abschnitt 4.3.2 wurde bereits eine kleine Übersicht zu *StringTemplate* gegeben. Im Folgenden wird der grundlegende Aufbau der Definition eines StringTemplate's gegeben. Dabei wird nur die Definition von StringTemplate's in einer separaten Datei betrachtet. Die definierten Templates besitzen alle den gleichen Aufbau:

«**RegelName**»(«**Params**») **::= <<**
    «**Rumpf**»
>>

Der *RegelName* dient der Identifizierung des Templates und muss innerhalb der Datei eindeutig sein. *Params* ist eine optionale Liste von Parametern des definierten Templates. Ihre Definition erfolgt durch die Angabe einer durch Kommata separierten Liste von Parameternamen. Der Name von anderen in der Datei definierten Templates sollte nicht für die Benennung von Parametern verwendet werden. Der *Rumpf* einer StringTemplate-Definition spezifiziert dessen Ausgabe in Abhängigkeit der übergebenen Parameter. Innerhalb des Rumpfes kann beliebiger Text stehen. Des Weiteren bietet StringTemplate spezielle Konstrukte an, von denen hier nur die Wichtigsten aufgezählt werden:

<!*Comment***!>**
: Zur Angabe von Kommentaren innerhalb des Rumpfes eines StringTemplate's

<*Param***>**
: Fügt den Wert von *Param* an der Textstelle ein. Für Objekte entspricht dies dem Aufruf der Methode *toString*. Falls das Objekt eine Liste ist, wird für jedes Element des Liste die *toString* aufgerufen.

  Beispiel:

```
identityTemplate(myParam) ::= <<
<myParam>
>>
```

<*Param***; separator=***Sep***>**
Gibt den für *Param* übergebenen Wert aus. Falls *Param* eine Liste ist wird diese durch *Sep* separiert ausgegeben.

Beispiel:

```
listTemplate(myListParam) ::= <<
<myListParam; separator=“, “>
>>
```

<*templateName***(<***Params***>)>**
Aufruf eines anderen Templates mit dem Namen *templateName*. Dabei können Parameter an das Template mitübergeben werden.

<if(<*Param***>)>** *<block>* **<endif>**
Fallunterscheidung. Dabei wird überprüft, ob *Param* nicht **null** ist bzw. bei boolschen Werten wird überprüft, ob *Param* dem Wert **true** entspricht. Im Falle der erfolgreichen Überprüfung wird der Inhalt von *<block>* in die Ausgabe eingefügt.

Es existieren noch weitere Konstrukte. Eine vollständige Auflistung ist auf der Webseite der StringTemplate-Engine[143] hinterlegt.

[143] `http://www.antlr.org/wiki/display/ST/StringTemplate+cheat+sheet`

5.5.2 Integration von StringTemplate's

Nachdem die Grundlagen für die Definition von StringTemplate's erläutert wurden, wird im Folgenden die Generierung einer formatierten Ausgabe in ANTLR unter Verwendung von StringTemplate's beschrieben. In einer ANTLR-Grammatik muss für eine Generierung von Templates die Option *output* auf den Wert *template* gesetzt werden.

Anschließend kann in der Grammatikdefinition für jede Regel ein Ausgabetemplate spezifiziert werden. Dazu wird der '->'-Operator verwendet, so dass jede Regel folgendermaßen aufgebaut ist:

```
regel :   < Alternative 1 >
          -> < Ausgabetemplate 1 >
      |   < Alternative 2 >
          -> < Ausgabetemplate 2 >
     ...
      |   < Alternative N >
          -> < Ausgabetemplate N >
      ;
```

Eine Fallunterscheidung ist ebenfalls möglich, so dass für einzelne Alternativen verschiedene Templates in Abhängigkeit von Prädikaten angeben werden können. Dabei werden die Prädikate in der Reihenfolge ihrer Definition ausgewertet. Des Weiteren muss immer eine Ausgabe ohne Angabe einer Bedingung definiert werden. Das führt zu einer Erweiterung der vorgestellten Syntax:

```
regel :   < Alternative 1 >
          -> {< Praedikat 1 >}? < Ausgabetemplate 1.1 >
          -> {< Praedikat 2 >}? < Ausgabetemplate 1.2 >
         ...
          -> {< Praedikat M >}? < Ausgabetemplate 1.M >
          -> < Ausgabetemplate 1.M+1 >
     ...
      ;
```

Die Definition der Prädikate muss in der durch die Option *language* eingestellten Sprache geschehen. Dabei kann unter anderem auf die einzelnen Teile der Regeldefinition zurückgegriffen werden. Somit kann jede Regel auf die Templates, die von Unterregeln erzeugt wurden zugreifen und diese zur Erzeugung des eigenen Templates verwenden. In dieser Arbeit beschränkt sich die Angabe der Ausgabetemplates auf anonyme und in einer externen Datei definierte Templates. Anonyme Templates können für einfache Ausgaben, wie die einer festen Zeichenkette, verwendet werden. Für komplexere Templates bietet sich die Definition in einer separaten Datei an. Zum Aufruf externer Templates müssen diese zunächst dem generierten Parser bekannt gemacht werden (wird durch die setTemplateLib(...)-Funktion bewerkstelligt). Anschließend können alle bekanntgemachten Templates direkt verwendet werden. Beispielsweise sei das externe definierte Template

```
varDeclTemplate(typName, varName) ::= <<
<typeName> <varName>;
>>
```

in einer Grammatik bekannt. Des Weiteren seien unter anderem folgende Regeln in der Grammatik definiert:

```
...
type
    :   INT | DOUBLE
    ;
variableDeclaration
    :   ^(VAR_DECL type IDENT initialValue)
    ;
...
```

Dann kann die Spezifikation der Ausgabe wie folgt aussehen:

```
...
type
    :   INT    -> {%{"INT"}}    // Anonymes Template
    |   DOUBLE -> {%{"DOUBLE"}} // Anonymes Template
    ;
variableDeclaration
    :   ^(VAR_DECL t=type IDENT initialValue?)
    // Verwendung des extern definierten Templates
    ->  varDeclTemplate(typName = {$t.st},
                        varName = {$IDENT.text})
    ;
...
```

In der Regel *type* werden zwei anonyme Templates für Generierung der Ausgabe verwendet. In der Regel *variableDeclaration* wird statt eines anonymen Templates, das extern definierte Template *varDeclTemplate* verwendet. Dabei wird dem Templateparameter *typName* das Ausgabetemplate des Knotens *t*, folglich ein anonymes Ausgabetemplate, dass durch die Regel *type* erzeugt wurde, zugewiesen. Des Weiteren wird der Parameter *varName* mit dem Wert des Attributes *text* von *IDENT* belegt. In diesem Stil kann für jede Regel ein Ausgabetemplate definiert werden. Falls eine Regel keine Ausgabe erzeugen soll, kann dies dadurch erreicht werden, dass kein Ausgabetemplate spezifiziert wird.

5.5.3 Definition der Anforderungen an die Ausgabe

Nach der Beschreibung der Definition von StringTemplate's und der Integration in ANTLR werden in diesem Abschnitt die Anforderung an die Ausgabe vorgestellt.

Eine Unterscheidung zwischen der Ausgabe für Header- und Cpp-Datei ist notwendig, da beispielsweise die Initialisierung statischer Variablen in Cpp-Dateien erfolgen soll. Des Weiteren müssen Kommentare des Quelltextes in den Zielcode eingesetzt werden. Zudem soll das Format des Zielcodes an den Formatierungstil *ansi* des Werkzeuges *Artistic Style 1.23*[144] angelehnt sein.

[144] http://astyle.sourceforge.net/

5.5.4 Umsetzung der Anforderumgen

Für die Unterscheidung der Ausgabe des Zielcodes für Header- und Cpp-Datei sind zwei Durchläufe notwendig. Der erste Durchlauf ist für alle Klassen Pflicht. Dabei wird die C++-Headerdatei für die jeweilige Klasse generiert. Für Klassen, die keine statischen Variablen und nur **inline** zu definierende bzw. rein virtuelle Methoden enthalten, ist ein weiterer Durchlauf nicht notwendig, da das Resultat eine leere Cpp-Datei wäre. Aus diesem Grund wird für jede Klasse im Vorwege anhand der in der *ClassDetailsMap* abgespeicherten Informationen überprüft, ob ein zweiter Durchlauf notwendig ist.

Beispiel 14 Fallunterscheidung für die Generierung des formatierten Zielcodes

```
variableDeclarations
    :   ^(VAR_DECLARATION
            umid=uniqueMemberId
            modList=modifierList[true]
            varType=type
            variableDeclarator)

// 1.Fall: Ausgabetemplate fuer Header-Datei
->  {outputTypeMember.equals(EOutputType.CPP_HEADER_FILE)}?
    variableDeclarationTemplate
       (modList      = {$modList.st},
        variableType = {$varType.st},
        variableName = {$umid.st})

// 2.Fall: Ausgabetemplate fuer Cpp-Datei
->  {outputTypeMember.equals(EOutputType.CPP_IMPLEMENTATION_FILE)
     && ($umid.classMemberReturn.isStatic())}?

    staticVariableMemberImplTemplate
       (variableWithInitialisation = {$variableDeclarator.st},
        variableType               = {$varType.st},
        modList                    = {$modList.st},
        className                  = {detailsOfThisClass.getName()},
        classNameSpace             = {namespaceWithUsings
                                       (detailsOfThisClass.getNamespace())})

// 3.Fall: keine Templateausgabe bzw. Ausgabe des Leertemplates
->
```

Beispielsweise werden für die Regel *variableDeclarations* Beispiel 14, die die Deklaration von Klassenvariablen beschreibt, drei Ausgaben unterschieden. Der erste Fall sorgt dafür, dass alle definierten Klassenvariablen mit Hilfe des Templates *variableDeclarationTemplate* in der Header-Datei deklariert werden. Der zweite Fall sorgt dafür, dass alle statischen Variablen in der Cpp-Datei unter Zuhilfenahme des Templates *staticVariableMemberImplTemplate* initialisiert werden. Die Deklaration nicht statischer Variablen, ist für die Erzeugung der Cpp-Datei nicht relevant und wird durch den dritten Fall ab-

gedeckt. In einem ähnlichen Stil sind die Ausgabetemplates für jede Regel definiert. Dabei kann auf die in der Zweiten Zwischenphase bestimmten und im Syntaxbaum hinterlegten Typinformationen für einzelne Knoten des ASB's zurückgegriffen werden. Dadurch ist es überhaupt möglich für einen statischen Funktionsaufruf anderen Code zu erzeugen als für einen nicht statischen Funktionsaufruf. Je nach Objekt-Typ und Ausdruckstyp der linken Seite (lhs) eines Ausdrucks der Form <lhs>.<rhs>, wird dieser in

- <lhs>.<rhs>,
- <lhs>::<rhs> oder
- <lhs>–><rhs>

umgewandelt.

Das Einfügen der Kommentare wird „*unter der Hand*“ durchgeführt, so dass für jede Ausgaberegel eine explizite Einordnung der zugehörigen Kommentare entfällt. Dazu musste der ANTLR-Mechanismus für die Erzeugung des Rückgabewertes einer Regel erweitert werden. Dazu wurde speziell das folgende Template definiert:

```
commentListTemplate(commentList, oldTemplate) ::= <<
<if(commentList)><commentList><endif><oldTemplate>
>>
```

Nach dem Durchlaufen einer Regel wird überprüft, ob ein Ausgabetemplate für die Regel angewendet wurde und das Ergebnis nicht **null** ist. Wenn dies zutrifft, wird eine neue Instanz des Templates *commentListTemplate* erzeugt und das Ausgabetemplate als Wert für den Parameter *oldTemplate* übergeben. Als Kommentarliste werden die am Token des Wurzelknotens hinterlegten Kommentare übergeben (siehe Abschnitt 5.1.2).

Die erzeugten Dateien werden in dem beim Programmstart spezifizierten Verzeichnis im Unterordner *gen* angelegt. Vor dem kompilieren der Generate müssen alle händisch erzeugten C++-Klassen, die innerhalb der übersetzten Quelltexte verwendet werden, in die entsprechenden Verzeichnisse relativ zu dem Zielverzeichnis kopiert werden.

6 Praxisbeispiel

Im diesem Kapitel wird anhand der Übersetzung von ausgewählten Teilen des FinTS-Kernel's, einem Sofwareprodukt der PPI AG, der entwickelte Prototyp getestet. Des Weiteren dient die Übersetzung zur Messung des Verhältnisses von Aufwand und Nutzen des Übersetzers, da vor einer Weiterentwicklung des Werkzeuges sichergestellt werden muss, ob ein Einsatz in „*reellen*" Software-Projekten wirtschaftlich ist. Zunächst wird eine allgemeine Beschreibung des FinTS-Kernel's gegeben und die Auswahl der zu übersetzenden Bestandteile begründet. Anschließend werden die Modifikationen an den Java-Quelldateien beschrieben, die notwendig waren, um die ausgewählte Teilmenge des Kernel's mit Hilfe des entwickelten Prototypen übersetzen zu können. Abschließend werden die gewonnenen Erkenntnisse zusammengefasst.

6.1 Beschreibung des FinTS-Kernel

Der Financial Transaction Services-Kernel (kurz: FinTS-Kernel) ist eine allgemeine Programmierbibliothek, die die clientseitige Abwicklung des FinTS 4.0 [145] Protokolls ermöglicht. Der Kernel stellt z. B. Funktionalitäten für die Bearbeitung von FinTS-Nachrichten, für die Erstellung von Geschäftsvorfällen, für Kommunikationsverfahren mit einem FinTS-Server und für die Verarbeitung von SWIFT [146]- und DTAZV [147] / DTAUS [148]-Fremdformaten zur Verfügung. Dialogklassen ermöglichen das Ausführen von Dialogen inklusive Aufsetzpunktbehandlung und kryptographischem Keymanagement. Verschiedene Krypto-Adapter binden kryptographische Medien wie z.B. hardware- oder softwarebasierte Medien (Chipkarte oder Diskette) für den Singleuserbetrieb sowie softwarebasierte Kryptographie für den Multiuserbetrieb (Datenbank) ein.

Für eine Übersetzung eignen sich vorwiegend Klassen, die keine externen Java-Bibliotheken benutzen und zudem einen nicht zu technischen Kontext vorweisen. Somit sind vor allem die Fremdformat-Toolbox für DTAZV / DTAUS sowie SWIFT geeignet für eine Übersetzung. Zudem sind sie von den restlichen Bestandteilen losgelöst, was die durchzuführenden Anpassungen auf ein Minimum reduziert.

[145] Link: `http://www.hbci-zka.de/spec/fints_v4_0.htm`, zugegriffen am 02.05.2009
[146] engl.: Society for Worldwide Interbank Financial Telecommunication
[147] Datenträgeraustausch Auslandszahlungsverkehr
[148] Datenträgeraustausch-Verfahren

6.2 Übersetzung der DTA-Toolbox

Die DTA-Toolbox besteht aus insgesamt 14 Klassen in zwei verschiedenen Packages und umfasst 1650 Zeilen reinen Java-Code (ohne Kommentare). Davon sind vier Ausnahmeklassen, die direkt oder indirekt von *java.lang.Exception* abgeleitet sind. Die restlichen Klassen sind „*normale*“ und abstrakte Klassen. Innerhalb der DTA-Toolbox werden Instanzen der generischen Klasse *java.util.ArrayList* verwendet. Da generische Klassen in der aktuellen Version nicht übersetzt werden können, müssen diese in „*normale*“ Klassen eingewrappt werden. Die angefertigten Wrapperklassen werden nicht übersetzt. Stattdessen müssen in C++ funktional äquivalente Klassen angefertigt werden. Zum Beispiel wurde die Zuweisung

```
ArrayList notificationRecords = new ArrayList();
```

durch

```
DTAZVNotificationRecordList notificationRecords
    = new DTAZVNotificationRecordList();
```

ersetzt. Die Wrapperklasse *DTAZVNotificationRecordList* ist von der generischen Klasse *InternalArrayList* abgeleitet und sieht wie folgt aus:

```
package DTAToolbox.de.ppi.fis.travic.kernel.doNotTranslate;
public class StringList extends InternalArrayList<String> {
    public String[] toArray() {
        return internalArray.toArray(new String[size()]);
    }
}
```

Die Basisklasse *InternalArrayList* wrappt eine Instanz der Klasse *ArrayList* ein und ist im Anhang C.3 abgebildet. Entsprechend wurde für alle *ArrayList*-Instanzen vorgegangen. Für den generierten Zielcode musste in C++ eine Klasse mit identischen Namen und gleicher Funktionalität angefertigt werden.

Des Weiteren wurden die Basisklassen für die Ausnahmebehandlung vereinfacht. Außerdem wurden Funktionsaufrufe von Systembibliotheken, wie zum Beispiel *System.arrayCopy(...)* durch äquivalenten Java-Code ersetzt, um den Konfigurationsaufwand und den Aufwand für die Anfertigung einer äquivalenten C++-Funktion zu eliminieren. Insgesamt mussten ca. 50 Zeilen Code in den Originalquelltexten angepasst werden.

6.3 Übersetzung der SWIFT-Toolbox

Die SWIFT-Toolbox besteht aus insgesamt 37 Klassen in zwei verschiedenen Paketen und umfasst 8845 Zeilen reinen Java-Code. In der SWIFT-Toolbox werden wie in der DTA-Toolbox Instanzen der generischen Klasse *java.util.ArrayList* verwendet. Dadurch entsteht wie bei der Übersetzung der DTA-Toolbox ein Aufwand für die Anpassung der Konfiguration. Des Weiteren müssen Wrapperklassen in Java angefertigt und äquivalente C++-Klassen implementiert werden. Aufgrund der Vielfalt mussten insgesamt 16 Wrapperklassen angefertigt werden. Zudem mussten weitere auf C++-Seite noch nicht existierende Funktionen der Klasse *String* implementiert werden. Des Weiteren verwendet die SWIFT-Toolbox zum Parsen von SWIFT-Nachrichten die Klasse *java.util.StringTokenizer*. Diese Klasse wurde rudimentär in C++ nachgebaut. Außerdem mussten aus dem Quelltext einige in der aktuellen Version des Übersetzers nicht unterstützte Konstrukte umgewandelt werden. Insgesamt mussten ca. 180 Zeilen Java-Code für Wrapperklassen manuell erzeugt und ca. 200 Zeilen Originalquelltext modifiziert werden.

6.4 Diskussion der Ergebnisse der Übersetzung

Nach den Anpassungen der Quelltexte konnten beide Toolboxen in äquivalenten C++-Code konvertiert werden. Ein anschließender Test der Funktionalität zeigte keine Fehler auf. Der generierte Zielcode ist formatiert, lesbar und verständlich. Die Zuordnung des C++-Codes zum Java-Code fällt relativ leicht. Ein Manko bezüglich der Lesbarkeit sind die generierten Funktionen zum Initialisieren von Feldern. Zudem werden an einigen Stellen im Programm zu lange Zeilen nicht umgebrochen (z. B. Bedingungen von if-Anweisungen). Der Aufwand für die Übersetzung generischer Klassen ist sehr hoch, da eine Anfertigung von Wrapperklassen sowohl in Java als auch in C++ notwendig ist. Somit wäre für eine Weiterentwicklung des Übersetzers die Behandlung von generischen Klassen essentiell wichtig um die Aufwände für deren Übersetzung minimieren zu können. Des Weiteren muss die Ausgabe für Fehler, die aufgrund unvollständiger Informationen auftreten, verbessert werden.

Die benötigte Zeit für den Übersetzungsvorgang auf einem Intel Core 2 Duo mit 2GHz und 2GB RAM betrug für die DTA-Toolbox ca. 3 Sekunden und für die SWIFT-Toolbox ca. 10 Sekunden (siehe Tabelle 4). Die Messung wurde unter Zuhilfenahme der Java-Funktion *System.currentTimeMillis()* durchgeführt. An den Werten aus Tabelle 4 wird deutlich, dass das Einlesen der Quelldateien und die Generierung des Zielcodes die meiste Zeit beanspruchen. Des Weiteren lässt sich erahnen, dass sich die Übersetzungszeit in etwa linear zur Anzahl der zu übersetzenden reinen Codezeilen verhält. Diese Beobachtungen sind jedoch rein spekulativ und es bedarf weiterer Messungen, die beispielsweise auch die Schwere eines zu übersetzenden Quelltextes berücksichtigen. Eine detaillierte Auflistung der Messergebnisse ist im Anhang B abgebildet.

Tabelle 4: Ausführungszeiten der einzelnen Phasen des Übersetzungsvorganges für die SWIFT- und DTA-Toolbox (Durchschnittswerte von jeweils 10 Messungen)

	DTA-Toolbox	**SWIFT-Toolbox**
Anzahl Java-Klassen	14	37
Anzahl Codezeilen	1650	8845
Ausführungszeit Vorbereitung	512 ms	590 ms
Ausführungszeit Initiale Phase	952 ms	3126 ms
Ausführungszeit 1. Zwischenphase	392 ms	1378 ms
Ausführungszeit 2. Zwischenphase	328 ms	1390 ms
Ausführungszeit 3. Zwischenphase	288 ms	903 ms
Ausführungszeit Ausgabephase	807 ms	2484 ms
Ausführungszeit gesamt	3279 ms	9871 ms
Ausführungszeit pro Codezeile	1,99 ms	1,12 ms

Insgesamt bleibt festzuhalten, dass der größte Aufwand für die Anfertigung und die Anpassungen der C++-Basisklassen, wie zum Beispiel die Erweiterung der C++-Stringklasse, anfielen. Da davon auszugehen ist, dass dieser Aufwand im Laufe einer Verwendung des Übersetzers stetig abnimmt, stellt dies keinen signifikanten Nachteil dar.

Abschließend ist zu erwähnen, dass die SWIFT- und DTA-Toolbox auf der beigelegten CD sowohl in Originalform als auch in der für die Übersetzung modifizierten Endform abgelegt sind. Die extra angefertigten C++-Klassen sind ebenfalls auf der CD abgelegt, so dass die Übersetzung nachvollzogen werden kann. Die für die Verwendung der CD notwendigen Informationen sind im Anhang A zu finden.

7 Erweiterung des Übersetzers

Abschließend werden die in Abschnitt 4.1 ausgeschlossenen Konstrukte erneut betrachtet und ggf. Strategien für eine Übersetzung in zukünftigen Versionen des werkzeuges präsentiert und diskutiert.

7.1 Detektion von Zyklen, Herausgabe des this-Zeiger's

Vor einem flächendeckenden Einsatz des Übersetzers muss das Problem zyklischer Datenstrukturen gelöst werden. Zyklische Datenstrukturen sind Objekte, die Referenzen auf sich selbst in Klassenvariablen speichern können. Ein Anwendungsfall ist zum Beispiel ein Knoten eines gerichteten Graphen, der alle Knoten zu denen ausgehende Kanten existieren in einer Liste speichert. Zur Vereinfachung wird angenommen, dass jeder Knoten genau eine ausgehende Kante besitzt. Diese Eigenschaft bildet sich im folgenden Java-Codeausschnitt ab:

```
public class Node {
    ...
    private Node subNode;

    public setSubNode(Node subNode) {
        this.subNode = subNode;
    }
    ...
}
```

Wird nun eine Instanz n der Klasse *Node* erzeugt, die einen Knoten eines Graphen abbilden soll, der eine ausgehende Kante $e := (n, n)$ besitzt, enthält n eine Referenz auf sich selbst in der Klassenvariable *subNode*. In Java stellt dies im Normalfall kein Problem dar, da der Garbage Collector Zyklen erkennen kann. In C++ kann eine naive Übersetzung, falls die Klasse *Node* in ein Heap-Objekt übersetzt wird, zur Folge haben, dass die Instanz dieses Knotens nie destruiert wird. Das liegt daran, dass der Smart-Pointer nur die Anzahl der Referenzen zählt, die auf das eingewrappte Objekt gehalten werden. Dabei wird nicht unterschieden, ob das Objekt sich selbst oder eine andere Instanz das Objekt referenziert. Deshalb würde unter der Annahme, dass *Node* von *HeapObject* abgeleitet wurde, das C++-Programmfragment

```
...
Node::Ptr n = Node::create();
n->setSubNode(n);
...
```

zunächst einen neue Instanz n vom Typ *Node* auf dem Heap anlegen. Anschließend wird durch die Zuweisung des Kindsknoten n zu n selbst, die Kante e realisiert. Durch diese Zuweisung wird der Referenzzähler des Smartpointers um eins erhöht. Wenn der Kindsknoten im weiteren Programmverlauf nicht auf eine andere Objektinstanz gesetzt wird, ist der Wert des Refernzzähler am Ende des Programmes immer noch größer als 0 und die Instanz wird nicht destruiert und es entsteht ein Speicherleck. Das gleiche Problem tritt ebenfalls auf, wenn sich zwei oder mehrere unterschiedliche Instanzen gegenseitig referenzieren. Im gleichen Sinne trifft dies auch für Instanzen verschiedener Objekte zu. Zur Lösung des Problems können beispielsweise schwache Referenzen an den entsprechenden Stellen verwendet werden. Das eigentliche Problem besteht aber schon in der Erkennung möglicher Zyklen, da für eine Analyse die gesamte Klassenhierachie betrachtet werden muss und eine tiefgehende semantische Analyse erforderlich ist. Des Weiteren ist die zyklische Abhängigkeit von Instanzen unterschiedlicher Klassen in C++ mit großen Schwierigkeiten verbunden, da unter Umständen Klassen vorwärts deklariert werden müssen. Das hat zur Folge, dass nur mit nackten Zeigern auf den Instanzen der vorwärts deklarierten Klasse innerhalb der Klassen gearbeitet werden kann. [149] Insgesamt bleibt festzuhalten, dass die Grundlage für die Durchführung einer Analyse bereits vorhanden ist, aber der zeitliche Aufwand für eine Übersetzung zyklischer Datenstrukturen den Rahmen dieser Diplomarbeit sprengen würde.

Ein weiteres Problem stellt die Herausgabe des Wertes des **this**-Zeigers dar. Zum Einen können dadurch zyklische Datenstrukturen aufgebaut werden und zum Anderen zeigt der **this**-Zeiger von Heap-Objekten nur auf das eingewrappte Objekt. Das heißt, dass eine Funktion, die den **this**-Zeiger zurückgibt, keine Referenz auf den Smartpointer zurückliefert. Somit werden Methoden, die den **this**-Zeiger zurückgeben, auch in Zukunft nicht zulässig sein.

7.2 Generische Konstrukte, Erweiterung der Unterstützung von Enumerationen und Schnittstellen

Die Übersetzung von generischen Klassen und die Erweiterung der Unterstützung von Schnittstellen und Enumerationen stellen neben der Erkennung zyklischer Datenstrukturen die wichtigsten noch zu implementierenden Funktionalitäten dar. Generische Klassen und Funktionen sind mittlerweile ein wichtiger Bestandteil von Java-Programmen, so dass insbesondere generische Containerklassen nicht mehr wegzudenken sind. Für die Übersetzung generischer Konstrukte müssen vor allem Anpassungen in der Methodenauflösung und der Bestimmung von Variablentypen durchgeführt werden. Da für die Übersetzung auf eine Klassenhierachie verzichtet wurde, in der alle Klassen implizit von einer Basisklasse abgeleitet sind, darf zum Besipiel eine Instanz einer generischer Containerklasse entweder nur Heap-Objekte oder nur Value-Objekte enthalten. Diese Eigenschaft müsste vom Werkzeug zur Übersetzungszeit überprüft

[149] vgl. [LLM06, Seite 521]

werden. Ansonsten muss lediglich beachtet werden, dass C++-Template's sowohl Wildcards, als auch Beschränkungen (in Java durch **extends** und **super**) nicht unterstützen. [150] Der tatsächliche Aufwand für die automatisierte Übersetzung von generischen Konstrukten ist schwer einzuschätzen.

Zusätzlich muss der Übersetzer für eine vollständige Unterstützung von Schnittstellen und Enumerationen erweitert werden. Die Definition von nicht statischen Klassenvariablen innerhalb einer Enumeration kann nur dann gestattet werden, wenn jeder einzelne Aufzählungswert durch eine einzigartige Instanz repräsentiert wird. Es wäre durchaus möglich in C++ einen Workaround dafür zu entwickeln. Stellt man die Frage nach der Notwendigkeit von Instanzvariablen in einer Enumeration, ist zu erkennen, dass die Funktionalität durch statische Felder äquivalent abgebildet werden kann. Statt der Kombination der Verwendung eines speziellen Konstruktors mit einer Klassenvariablen, kann ein statisches Feld verwendet werden, wie Beispiel 15 zeigt.

Beispiel 15 Veranschaulichung der Unnotwendigkeit von Konstruktoren und nicht statischen Klassenvariablen in Java

Die Enumeration *E* sei wie folgt definiert:

```
public enum E {
    // Jeweils Konstruktor mit entsprechenden Wert
    // fuer die Klassenvariable str aufrufen
    A("Ich bin A"), B("Ich bin B"), C("Ich bin C");

    private String str;
    private E(String s) {
        str = s;
    }
    public String getStr() {
        return str;
    }
}
```

Eine äquivalente Funktionalität kann auch durch die alleinige Verwendung einer statischen Klassenvariable erreicht werden:

```
public enum E {
    A, B, C;

    private static String[] strs
        = new String[]{"Ich bin A", "Ich bin B",  "Ich bin C"};

    public String getStr() {
        return strs[ordinal()];
    }
}
```

In der aktuellen Version des Übersetzers dürfen Schnittstellen erweitert werden,

[150] vgl. [iso98, Seite 235 ff.]

aber es ist nicht gestattet Variablen zu verwenden, deren Typbezeichnung eine Schnittstelle ist. Der Nutzen der Auflockerung dieser Beschränkung ist gegeben, da eine bessere Abstraktion durch sie verhindert wird.

7.3 Annotationen

Die Übersetzung von Annotationen wurde zunächst ausgeschlossen. In einer Weiterentwicklung des Übersetzers könnten Annotationen aber ein wichtiger Bestandteil werden. Sie könnten zur Optimierung des Übersetzungsvorganges eingesetzt werden. Zum Beispiel wäre es mit Hilfe einer Annotation *Inline* möglich, für Methoden oder Konstruktoren, die durch das intern im Übersetzer definierte Inline-Kriterium nicht inline deklariert werden, eine solche Inlinedeklaration zu erzwingen. Weiterere mögliche Anwendungsfälle sind die Ersetzung ganzer Java-Konstrukte durch nativen C++-Code, das Festlegen des Objekttypen einer Klasse oder die Markierung bestimmter Methoden als virtuell. Des Weiteren ist eine Ersetzung von Teilen der Konfiguration durch Annotationen denkbar.

7.4 this-Konstruktoraufrufe

In Java ist es möglich Konstruktoren der eigenen Klasse mit Hilfe des Schlüsselwortes **this** aufzurufen. [151] Dazu gibt es in C++ kein Pendant, so dass von einer Übersetzung in der initialen Version abgesehen wurde. Um die Problematik besser nachvollziehen zu können betrachten wir Beispiel 16.

Beispiel 16 Aufruf eines anderen Konstruktors der gleichen Klasse

```
public class Rechteck {
    private int a;
    private int b;
    private int magic;
    ...
    public Rechteck(int a, int b) {
        this.a = a;
        this.b = b;
    }

    public Rechteck(int c) {
        this(2*c, 4*c);
        int magicTemp = random();
        magic = magicTemp;
    }
    ...
}
```

Man erkennt, dass die folgende Definition äquivalent zu der des zweiten Konstruktors ist:

```
public Reckteck(int c) {
    int a = 2*c;
    int b = 4*c;
    this.a = a;
    this.b = b;
    int magicTemp = random();
    magic = magicTemp;
}
```

Die Äquivalenz ist aber nur gegeben, wenn zum Einen die Parameter der betroffenen Konstruktoren unterschiedliche Namen besitzen und zum Anderen auf den **this**-Konstruktoraufruf keine Deklaration gleichnamiger Variablen folgt. Deshalb ist eine Erweiterung notwendig, um Konflikte und Übersetzungsfehler zu verhindern. Dazu können die den this-Konstruktor ersetzenden Zeilen in einen eigenen Block geschoben werden. Zudem ist vor der Deklaration zusätzlicher Variablen eine Überprüfung auf Namenskonflikte notwendig.

[151] vgl. [Hen97, Seite 116]

Das hat zur Folge, dass die leicht modifizierte Version des zweiten Konstruktors aus Beispiel 16

```
public Rechteck(int a) {
    this(2*a, 4*a);
    int b = random();
    magic = b;
}
```

intern in folgenden Java-Code transferiert werden muss

```
public Reckteck(int a) {
    {
        int a_1 = 2*a;
        int b = 4*a;
        this.a = a_1;
        this.b = b;
    }
    int b = random();
    magic = b;
}
```

Die Übersetzung des transferierten Quellcodes kann wie in Abschnitt 4.2.4.5.4 beschrieben durchgeführt werden. Insgesamt muss aber festgehalten werden, dass der zu tätigende Aufwand nicht den Nutzen rechtfertigt, da unter Umständen Kollisionsauflösungen zu unleserlichen Zielcode führen. Des Weiteren muss bei mehreren Konstruktoren einer Klasse, die einen **this**-Konstruktor aufrufen, zunächst eine Ordnung dieser bestimmt werden. Anschließend müssten die Konstruktoren in der Reihenfolge der bestimmten Ordnung transformiert werden.

7.5 Sonstiges

Nicht zulässige Konstruktionen, wie Variablendeklarationslisten, Star-Importanweisungen, foreach-Schleifen und die Definition von mehr als einer Klasse pro Quelldatei, sind reine Fleißarbeit und relativ leicht umsetzbar.

Problematischer wird hingegen die Initialisierung finaler Variablen einer Klasse. Da es in Java zulässig ist, eine finale Variable im Konstruktorrumpf zu initialisieren, kann die Bestimmung des initialen Wertes der Variable mit Schwierigkeiten verbunden sein. Da diese Variablen in C++ in der Initialisierungsliste initialisiert werden müssten, muss dieser Wert beim Aufruf des Konstruktors bereits bekannt sein oder berechnet werden können. Ob tatsächlich für jede finale Klassenvariable deren Wert vor der Ausführung des Konstruktorrumpfes bestimmt werden kann, ist fragwürdig.

Die Definition anonymer Klassen ist in C++ nicht möglich. [152] Der Mechanismus kann lediglich emuliert werden, falls keine lokalen Variablen der Funktion in der die anonyme Klasse definiert wurde, innerhalb der anonymen Klasse verwendet werden. Dann kann statt der anonymen Klasse, eine innere Klasse definiert werden, die die Funktion der anonymen Klasse abbildet. Diese innere Klasse muss einen eindeutigen Namen haben und darf keine anderen Klassen verstecken.

Die Verwendung von speziellen Initialisierungsblöcken muss differenziert betrachtet werden. Eine Art Konstruktor für das Klassenobjekt selbst (nicht die Exemplare der Klasse) ist ein static-Block, der einmal oder mehrmals in eine Klasse definiert werden kann. Die Blöcke werden genau dann in der Reihenfolge ihrer Definition ausgeführt, wenn die Klasse in die virtuelle Maschine geladen wird. [153] Die Konvertierung von static-Blöcken in äquivalenten C++-Code ist möglich. Eine mögliche Umsetzung der Funktionalität in C++ wird von *Gops* in seinem Blog präsentiert [154]. Im Gegensatz zum static-Block ist ein Instanzinitialisierer ein Art Konstruktor ohne Namen, der lediglich aus einem Paar geschweifter Klammern besteht und einem statischen Initialisierungsblock ohne das Schlüsselwort **static** gleicht. Die Initialisierungblöcke werden am Anfang eines jeden Konstruktorrumpfes direkt nach dem Aufruf des Konstruktors der Vaterklasse ausgeführt. [155] Das heißt, dass eine ähnliche Übersetzungstrategie wie im Falle von **this**-Konstruktoraufrufen durchgeführt werden muss. Die Übersetzung von Instanzinitialisierer ist nicht geplant, da eine äquivalente Funktionalität erreicht werden kann, indem der Code aus den Instanzinitialisierern in alle Kostruktoren kopiert wird. Des Weiteren werden Instanzinitialisierer und statische Blöcke eher selten eingesetzt, so dass der Aufwand für eine Übersetzung im Vergleich zum Nutzen zu hoch scheint.

Schließlich könnten **break**- und **continue**-Anweisungen, die mit einem Rücksprunglabel versehen wurden, mit Hilfe von **goto**'s in C++ realisiert werden. Dabei muss aber beachtet werden, dass sowohl bei einem **break** als auch bei einem **continue** die Schlei-

[152] vgl. [iso98]
[153] vgl. [Fla98, Seite 62 und 63]
[154] Link: `http://gopswritings.blogspot.com/2006/03/c-static-blocks.html`, `zugegrifenam02.05.2009`
[155] vgl. [Fla98, Seite 129 und 130]

fenaktualisierung erneut ausgeführt wird. Zur Veranschaulichung ein kleines Beispiel: Würde das Java-Codefragment

```
    ...
    int i = 0;
  MYLABEL: for(; i < 10; i++)
    {
        if(i == 5) {
            continue MYLABEL;
        } else {
            System.out.println(i);
        }
    }
    ...
```

in den C++-Code

```
    ...
    int i = 0;
  MYLABEL: for(; i < 10; i++)
    {
        if(i == 5) {
            goto MYLABEL;
        } else {
            System::out.println(i);
        }
    }
    ...
```

umgewandelt werden, würde das Originalfragment terminieren und das Generat nicht. Der Grund dafür liegt darin, dass vor dem Sprung zum Label *MYLABEL* die Schleifenaktualisierung ausgeführt wird. Explizit heißt das, dass die Variable *i* um eins erhöht wird. Deshalb muss das Generat wie folgt angepasst werden:

```
    ...
    int i = 0;
    goto MYNEWLABEL;
  MYLABEL:
    i++; // Schleifenaktualisierung durchfuehren
    goto MYNEWLABEL;
  MYNEWLABEL: for(; i < 10; i++)
    {
        if(i == 5) {
            goto MYLABEL;
        } else {
            System::out.println(i);
        }
    }
    ...
```

Die Verwendung von **goto** ist im Allgemeinen kein guter Stil, aber in diesem Falle unumgänglich. Der erzeugte Code kann vor allem für tiefer verschachtelte Schleifen mit mehreren Labelmarkierungen unleserlich werden, so dass eine Übersetzung dieser Konstrukte nur in Ausnahmefällen zulässig sein sollte.

8 Fazit und Ausblick

Ziel dieser Arbeit war die Konstruktion eines Übersetzers, der Java-API's in funktional äquivalenten C++-Code transformiert. Zusätzlich sollte der generierte Zielcode gut lesbar und leicht verständlich sein und der entwickelte Übersetzer sollte anhand der Transformierung einer reellen Java-API getestet werden.

Im Vorfeld der Diplomarbeit wurden einige theoretische Überlegungen angestellt. Zunächst wurde überlegt, ob ein Java-nach-C++-Übersetzer, ein C++-nach-Java-Übersetzer oder sogar eine eigene Zwischensprache entwickelt werden soll, um den durch die zweigleisige unabhängige Entwicklung von gleichartiger Software entstehenden Mehraufwand entgegen zu wirken. Die Entscheidung fiel zu Gunsten der Entwicklung eines neuen Java-nach-C++-Übersetzers, nachdem die bereits existierenden Übersetzungswerkzeuge evaluiert und für nicht geeignet befunden wurden.

Das gesteckte Ziel wurde für Java-API's, die in einer festgelegten Untermenge von Java entwickelt wurden, erreicht. Für die Untermenge wurde mit Hilfe des Parsergenerators ANTLR ein mehrstufiger Übersetzer entwickelt. Des Weiteren ist durch die Verwendung von StringTemplate's, die bereits vollständig in ANTLR integriert sind, eine Anpassung des Ausgabeformates leicht durchführbar. Das entwickelte Werkzeug zeigt zudem, dass eine Erhaltung von Kommentaren durchaus möglich ist. Jedem ist die Verwendung von ANTLR für die Erzeugung von Parsern zu empfehlen, da es die „*komplette*“ Bandbreite an notwendigen Funktionalitäten für eine leichte und schnelle Entwicklung von Übersetzern zur Verfügung stellt.

Die Übersetzung der DTA- und SWIFT-Toolboxen zeigte zum Einen, dass der entwickelte Übersetzer funktionsfähig ist. Zum Anderen wurde deutlich, dass vor allem durch die Einschränkung, dass generische Klassen nicht direkt übersetzt werden können, ein relativ hoher Aufwand mit deren Übersetzung verbunden ist, da diese Klassen in nicht generische Java-Klassen eingekapselt werden müssen. Für diese Klassen müssen in C++ Äquivalente angefertigt werden, die die Funktionalität der Java-Klasse abbilden, da diese gekapselten Klassen nicht mitübersetzt werden können. Eine Erweiterung des Übersetzers um die Funktionalität generische Klassen übersetzen zu können, wäre somit der erste Ansatzpunkt für die Verbesserung des Werkzeuges. Zudem sind die Erweiterung des Übersetzers um einen Zyklendetektor und eine bessere Handhabung von Schnittstellen für die Vergrößerung des Anwendungsgebietes essentiell.

Durch die Anforderung einen funktional äquivalenten Zielcode zu erzeugen, der gleichzeitig gut verständlich ist, mussten an einigen Stellen Einschränkungen getroffen und Kompromisse gemacht werden. Zum Beispiel führte die Verwendung von gleichwertigen Arrays zu der Einschränkung, dass vorerst nur ein- und zweidimensionale Arrays verwendet werden können, da für die Realisierung einer expliziten Initialisie-

rung ein hoher Aufwand betrieben werden muss.

Abschließend sei erwähnt, dass die Unterschiede zwischen Java und C++ trotz einer ähnlichen Syntax zu groß sind, so dass die beiden Ziele, möglichst viele Java-Programme übersetzen zu können und gleichzeitig intuitiv verständlichen C++-Code zu erzeugen, konkurrierend sind. Somit wird es immer Einschränkungen bezüglich der zu übersetzenden Quelltexte geben, was eine Entwicklung **nur** in Java unmöglich macht.

Alle Erkenntnisse zusammengefasst bleibt festzuhalten, dass das entwickelte Werkzeug funktionsfähig ist. Da jedoch einige Einschränkungen obliegen, die zum aktuellen Zeitpunkt einen flächendeckenden Einsatz des Übersetzers erschweren, ist eine Weiterentwicklung unumgänglich. In Zusammenarbeit mit der PPI AG werden mein Betreuer Oliver Schmidt und ich im Anschluss an die Diplomarbeit das erarbeitete Projekt erneut evaluieren. Als Testobjekt dient dabei ein von PPI entwickelter Java-Kernel, für den bereits eine Kaufanfrage für eine äquivalente Version in C++ vorliegt. Sollte sich nach der erneuten Evaluation herausstellen, dass das Aufwand / Nutzen-Verhältnis für zukünftige Softwareentwicklungen positiv zu bewerten ist, wird die PPI AG den Übersetzer in ausgewählten Softwareprojekten einsetzen.

Literaturverzeichnis

[AH96] AIGNER, Gerald ; HÖLZLE, Urs: Eliminating virtual function calls in C++ programs . In: *LNCS* 1098 (1996)

[AS93] ATT, Bjarne S. ; STROUSTRUP, Bjarne: A History of C++: 1979-1991. In: *ACM SIGPLAN Notices* 28 (1993), S. 271–297

[Ban95] BANK, David: The Java Saga. In: *Wired* (1995), Dezember

[Bau05] BAUSCH, Daniel: Garbage-Collection in C++ / TU Darmstadt. 2005. – Forschungsbericht

[Bra04] BRACHA, Gilad: *Generics in the Java Programming Language*. 2004

[Bre] BREITBART, Jens: *Optimierungen für C++*. – Seminararbeit

[Bre99] BREYMANN, Ulrich: *C++ — Eine Einführung*. Bd. 5. Carl Hanser Verlag, 1999

[BV09a] BV, TIOBE S.: *TIOBE Programming Community Index for April 2009*. `http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html`, 2009. – [Online; Stand 30. April 2009]

[BV09b] BV, TIOBE S.: *TIOBE Programming Community Index for The Java Programming Language*. `http://www.tiobe.com/index.php/paperinfo/tpci/Java.html`, 2009. – [Online; Stand 30. April 2009]

[Byo03] BYOUS, Jon: *Java Technology: The Early Years*. `http://java.sun.com/features/1998/05/birthday.html`, 2003. – [Online; Stand 30. April 2009]

[Cor08] COREINTENT: Intentional Compilation: A Breakthrough Technology for Language Translation from Java to C-based platforms / CoreIntent. 2008. – Forschungsbericht

[DGC95a] DEAN, Jeffrey ; GROVE, David ; CHAMBERS, Craig: Optimization of Object-Oriented Programs Using Static Class Hierachy Analysis. In: *LNCS* 952 (1995)

[DGC95b] DEAN, Jeffrey ; GROVE, David ; CHAMBERS, Craig: Optimization of Object-Oriented Programs using Static Class Hierarchy Analysis. In: *Object-Oriented Programming*, Springer-Verlag, 1995, S. 77–101

[Eck98] ECKEL, Bruce: *Thinking in Java*. Prentice Hall International, 1998

[Fla98] FLANAGAN, David: *Java in a Nutshell*. Bd. 2. O'Reilly, 1998

[GSB05] GOSLING, James ; STEELE, Guy ; BRACHA, Gilad: *The Java Language Specification*. Bd. 3. Addison Wesley, 2005

[Har97] HARVEY, David: *Smart pointer templates in C++*. 1997

[Hen97] HENDRICH, Norman: *Java für Fortgeschrittene*. Springer Verlag, 1997

[iso98] ISO/IEC: *ISO/IEC 14882:1998: Programming languages: C++*. 1998

[Lew97] LEWANDOWSKI, Scott M.: *Design Issues In Java and C++*. 1997

[LLM06] LIPPMAN, Stanley B. ; LAJOIE, Josée ; MOO, Barbara E.: *C++ Primer*. Bd. 4. Addison Wesley, 2006

[Mar97] MARTIN, Robert C.: *Java and C++: A Critical Comparison*. 1997

[McP04] MCPEAK, Scott: Elkhound: A Fast, Practical GLR Parser Generator . In: *LNCS* 2985 (2004)

[Mey98] MEYERS, Scott: *Mehr Effektiv C++ programmieren*. Addison Wesley, 1998

[MK06] MÖSSENBÖCK, Hanspeter ; KEPLER, Johannes: *The Compiler Generator Coco/R - User Manual*. 2006

[MMBC97] MULLER, Gilles ; MOURA, Bárbara ; BELLARD, Fabrice ; CONSEL, Charles: Harissa: a Flexible and Efficient Java Environment Mixing Bytecode and Compiled Code. In: *COOTS'97: Proceedings of the 3rd conference on USENIX Conference on Object-Oriented Technologies (COOTS)*. Berkeley, CA, USA : USENIX Association, 1997, S. 1–20

[Par04] PARR, Terence: Enforcing Strict Model-View Separation in Template Engines. In: *WWW '04: Proceedings of the 13th international conference on World Wide Web*, 2004, S. 224–233

[Par07] PARR, Terence: *The Definitive ANTLR Reference - Building Domain-Specific Languages*. The Pragmatic Bookshelf, 2007

[PTB+97] PROEBSTING, Todd A. ; TOWNSEND, Gregg ; BRIDGES, Patrick ; HARTMAN, John H. ; NEWSHAM, Tim ; WATTERSON, Scott A.: Toba: Java For Applications: A Way Ahead of Time (WAT) Compiler. Tucson, AZ, USA : University of Arizona, 1997. – Forschungsbericht

[Str98] STROUSTRUP, Bjarne: *Die C++ Programmiersprache*. Bd. 3. Addison-Wesley, 1998

[VB04] VARMA, Ankush ; BHATTACHARYYA, Shuvra S.: Java-through-C Compilation: An Enabling Technology for Java in Embedded Systems. In: *DATE '04: Proceedings of the conference on Design, automation and test in Europe*, IEEE Computer Society, 2004, S. 161–167

[Vel98] VELDEMA, Ronald: Jcc, a native Java compiler. In: *Master's thesis, Vrije Universiteit*, 1998

[VLSU08] V.AHO, Alfred ; LAM, Monica S. ; SETHI, Ravi ; ULLMAN, Jeffrey: *Compiler - Prinzipien, Techniken und Werkzeuge*. Bd. 2. Addison-Wesley, 2008

[VRCG+99] VALLÉE-RAI, Raja ; CO, Phong ; GAGNON, Etienne ; HENDREN, Laurie ; LAM, Patrick ; SUNDARESAN, Vijay: Soot - a Java bytecode optimization framework. In: *CASCON '99: Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research*, IBM Press, 1999, S. 13

[WFD+98] WEISS, Michael ; FERRIÈRE, François de ; DELSART, Bertrand ; FABRE, Christian ; HIRSCH, Frederick ; JOHNSON, E. A. ; JOLOBOFF, Vania ; ROY, Fred ; SIEBERT, Fridtjof ; SPENGLER, Xavier: TurboJ, a Java Bytecode-to-Native Compiler. In: *LCTES '98: Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems*. London, UK : Springer-Verlag, 1998. – ISBN 3–540–65075–X, S. 119–130

[Wik09a] WIKIPEDIA: *C++ — Wikipedia, Die freie Enzyklopädie*. `http://de.wikipedia.org/w/index.php?title=C%2B%2B\&oldid=58675183`, 2009. – [Online; Stand 30. April 2009]

[Wik09b] WIKIPEDIA: *Programmierschnttstelle — Wikipedia, Die freie Enzyklopädie*. `http://de.wikipedia.org/w/index.php?title=Programmierschnittstelle\&oldid=57986015`, 2009. – [Online; Stand 2. Mai 2009]

Anhang

A Beschreibung der Verwendung der beiliegenden CD mit den Quelldateien des entwickelten Übersetzers

Auf der beiliegenden CD sind folgende Dokumente hinterlegt:

- die ANTLR-Grammatiken
- die für eine Kompilierung des Übersetzers notwendigen Bibliotheken und Ant-Skripte
- zusätzliche für den Übersetzungsprozess geschriebene Java-Klassen
- die C++-Basisklassen für generierte C++-Dateien
- die im Kapitel 6 übersetzten Java-Klassen der DTA- und SWIFT-Toolbox der PPI AG in Originalform und für den Übersetzungsvorgang modifizierter Form , sowie die dafür geschriebenen Wrapperklassen
- Shellskripte, die die Übersetzung der DTA- und SWIFT-Toolbox starten
- die verwendeten StringTemplate's

Im Folgenden wird die Verzeichnisstruktur und die in den Verzeichnissen enthalten Dateien aufgelistet und umschrieben. Es empfiehlt sich vor einer Verwendung den gesamten Inhalt der CD in ein lokales Verzeichnis auf dem verwendeten Rechner zu kopieren.

A.1 Wurzelverzeichnis

Im Wurzelverzeichnis liegt unter anderem der fertige Übersetzer im Form des Java-Archives *j2cpp.jar* bereit, so dass keine Übersetzung der Quelldateien des Parsers erforderlich ist. Stattdessen kann der Übersetzer durch den Befehl *java -jar j2Cpp.jar* und zusätzlichen Parametern gestartet werden. Die verschiedenen Parameter und Konfigurationsmöglichkeiten wurden in Abschnitt 4.3 beschrieben.

A.2 Ordner „grammars“

Im Ordner „*grammars*“ des Wurzelverzeichnisses liegen die ANTLR-Grammatiken für die einzelnen Phasen des Übersetzungsprozesses, sowie die Grammatik, die die Syntax der Konfigurationsdateien beschreibt. Diese können mit Hilfe des Ant-Skriptes, welches im Wurzelverzeichnis abgelegt ist, in entsprechenden Java-Code übersetzt werden. Die dabei generierten Parser werden in dem Ordner „*build*“ im Unterverzeichnis „*j2cpp*“ abgelegt.

A.3 Ordner „lib“

Im Ordner „*lib*“ sind die beiden verwendeten Bibliotheken *ANTLR* und *log4j* abgelegt, die für die Kompilierung des entwickelten Übersetzerwerkzeuges benötigt werden.

A.4 Ordner „src/j2cpp“

Im Unterordner „*j2cpp*“ des Verzeichnisses „*src*“ liegen die zusätzliche für den Übersetzer entwickelte Java-Klassen. Im folgenden werden die einzelnen Klassen aufgelistet und kurz beschrieben.

ClassDetails.java
: Klasse, deren Instanzen Informationen über Java-Klassen modellieren

ClassHierachieNode.java
: Klasse, die einen Knoten in einer Klassenhierachie modelliert

ClassHierachy.java
: Klasse, die eine Klassenhierachie modelliert und bestimmen kann

ClassMember.java
: Modelliert ein Klassenmitglied (Variable, Methode, Konstruktor, etc.)

ClassMemberMap.java
: Modelliert eine Liste von Klassenmitgliedern

CommonTokenWithCommentList.java
: Spezielle Token-Klasse, die die im Übersetzungsvorgang verwendeten Token modelliert. In einer Instanz dieser Klasse können zusätzlich zur ürsprünglichen Token-Klasse Kommentare des Quelltextes hinterlegt werden

Const.java
: Annotation, die in einer späteren Version des Übersetzers für Benutzerinteraktionen verwendet werden kann (Deklaration einer Variable / Funktion als konstant im Sinne von C++)

CustomTree.java
: Spezielle Klasse, die einen Knoten in einem abstrakten Syntaxbaum modelliert, an dem Informationen über den Typ des durch den Knoten modellierten Ausdrucks abgespeichert werden können

CustomTreeAdaptor.java
: Klasse, die für die Erzeugung von Knoten eine ASB's des Typs *CustomTree* benötigt wird, da sonst nur Knoten vom Typ *org.antlr.CommonTree* erzeugt werden

CustomTreeNodeStream.java
: Klasse, die einen Strom von Objekten des Typs *CustomTree* modelliert

DoubleKeyMap.java
: Klasse, die eine Hashmap modelliert, die über ein Paar von Schlüsseln verfügt

EBuildInType.java
: Enumeration für primitive Datentypen

EClassType.java
: Enumeration, die die verschiedenen Klassentypen modelliert

ELeftSideType.java
: Enumeration, die die verschiedenen Ausdruckstypen, die während der Zweiten Zwischenphase erzeugt werden, bestimmt.

EMemberType.java
: Enumeration, die den Typ einers Klassenmitglieds modelliert (Variable, Methode, etc.)

EObjectType.java
: Enumeration, die die verschiedenen Objekttypen enthält

EOutputType.java
: Enumeration für die Unterscheidung der Ausgabe

ESuperType.java
: Enumeration für die Unterscheidung des Vererbungstyps

EVisibility.java
: Enumeration, die die verschiedenen Sichtbarkeiten in Java modelliert

ForbiddenCppKeywordUseException.java
: Ausnahme, die die Verwendung eines in C++ reservierten Wortes im Java-Quelltext symbolisiert

Inline.java
: Annotation, die in einer späteren Version des Übersetzers für Benutzerinteraktionen verwendet werden kann (Deklaration einer Funktion als **inline** im Sinne von C++)

JavaTreeMain.java
: Klasse, die den Mittelpunkt des Übersetzers bildet. In ihr ist die aufzurufende Main-Funktion definiert.

LeftSideValue.java
: Klasse, die die in der Zweiten Phase bestimmten Typinformationen der verschiedenen Ausdrücke modelliert

MalformedConsoleArgumentsException.java
: Ausnahme, die ungültige oder unvollständige Parameter, die beim Aufruf des Programms übergeben wurden, symbolisiert

MalformedMappingFileException.java
: Ausnahme, die eine ungültige oder unvollständige Konfigurationsdatei symbolisiert

MappingDecoder.java
: Klasse, die die aus der Konfigurationsdatei ausgelesenen Informationen aufbereitet

MappingTracker.java
: Klasse, die die durchgeführten Namensraum- und Typenmappings protokolliert

MethodRange.java
: Klasse, die ein Typtupel einer Methode modelliert

MethodResolver.java
: Klasse, die zur Bestimmung von Methodenauflösungen verwendet wird

MethodTable.java
: Klasse, die eine Methodentabelle modelliert

Mutable.java
: Annotation, die in einer späteren Version des Übersetzers für Benutzerinteraktionen verwendet werden kann

NotYetSupportedException.java
: Ausnahme, die dem Anwender symbolisiert, dass ein zur Zeit nicht unterstütztes Java-Konstrukt innerhalb des zu übersetzenden Quelltextes verwendet wurde

Pair.java
: Modelliert ein Paar

ParseException.java
: Modelliert eine Ausnahme, die beim parsen aufgetreten ist

ParseUtil.java
: Utilityklasse, die verschiedene Funktionalitäten, wie die Umwandlung von Unicodezeichen anbietet

Quadrupel.java
: Modelliert ein Tupel, das vier Elemente besitzt

Tripel.java
: Modelliert ein Tripel

UnexpectedNullInformationException.java
: Modelliert eine Ausnahme, die symbolisiert, dass unerwartet keine Informationen bestimmt werden konnten

Virtual.java
: Annotation, die in einer späteren Version des Übersetzers für Benutzerinteraktionen verwendet werden kann

Cpp.stg
: Datei, die die derzeit verwendeten StringTemplate's enthält

A.5 Ordner „cpp“

Im Ordner „*cpp/utils*“ sind die C++-Basisklassen, die von den Generaten referenziert werden, abgelegt. Des Weitern werden im Ordner „*cpp/gen*“ die generierten C++-Klassen und das für ihre Kompilierung verwendete Makefile abgelegt.

A.6 Ordner „testproject“

Im Ordner „*testProject*“ sind die beiden übersetzten Toolboxen für SWIFT und DTA sowohl im modifizierten als auch im Originalzustand abgelegt.

Für die Übersetzung der DTA-Toolbox wurde die Konfigurationsdatei *DtaConfig.map* angefertigt. Durch den Aufruf des Shellskriptes *convertDta.sh* wird die Übersetzung der DTA-Toolbox durchgeführt. Die generierten Dateien werden im Ordner „*cpp/gen*“ abgelegt. Die manuell erzeugten Wrapperklassen, die im Unterordner „*DtaCppClasses*“ liegen, werden automatisch vom Shellskript in den Ordner „*cpp/gen*“ kopiert.

Für die Übersetzung der SWIFT-Toolbox wurden die Konfigurationsdatei *SwiftConfig.map* und das Shellskript *convertSwift.sh* angefertigt. Die generierten Dateien werden wie bei der Übersetzung des DTA-Toolbox im Ordner „*cpp/gen*“ abgelegt. Die manuell erzeugten Wrapperklassen werden automatisch vom Shellskript aus dem Ordner „*SwiftCppClasses*“ in den Zielordner kopiert.

In den Ordnern „*SwiftCppClasses*“ und „*DtaCppClasses*“ liegen jeweils Testprogramme, die zur Überprüfung der Korrektheit des übersetzten Quellcodes verwendet werden können. Diese werden ebenfalss automatisch durch die Shellskripte in den Ordner *cpp/gen/* kopiert. Für die Ausführung der Testprogramme muss lediglich der Befehl *make* im Ordner „*cpp/gen*“ aufgerufen werden und anschließend das erzeugte Programm durch *./program* gestartet werden.

B Messergebnisse zum Praxisbeispiel

B.1 LOC DTA-Toolbox

Tabelle 5: LOC-Statistik der DTA-Toolbox

Dateiname	Leerzeilen	Kommentarzeilen	Codezeilen
DTAZVTransferOrder.java	85	1367	396
DTAZV.java	77	579	281
DTAUS.java	92	780	229
DTAUSPaymentExchange.java	73	673	180
DTAUtil.java	58	243	164
DTAZVTradeNotificationRecord.java	34	548	160
DTAZVOtherNotificationRecord.java	19	250	79
DTAZVGoodsNotificationRecord.java	15	147	41
DTAUSPaymentExchangeExtension.java	13	139	29
DTAParseException.java	4	41	7
DTAZVNotificationRecord.java	4	36	5
Summe	474	4803	157

B.2 Laufzeit DTA-Toolbox

Tabelle 6: Laufzeiten der einzelnen Phasen bei der Übersetzung der DTA-Toolbox (in ms) [Vb = Vorbereitung, IP = Initiale Phase, Zp = Zwischenphase, Ap = Ausgabephase]

Versuchsnr.	Vb	IP	1. Zp	2. Zp	3. Zp	Ap	Σ
1	482	934	405	361	285	767	3234
2	458	975	395	304	289	771	3192
3	511	1019	359	296	288	763	3236
4	496	989	402	328	287	784	3286
5	572	928	363	329	291	826	3309
6	479	1033	363	335	282	842	3334
7	484	988	396	322	291	837	3318
8	562	908	409	403	288	886	3456
9	568	874	415	298	289	789	3233
10	506	868	411	299	293	802	3179

B.3 LOC SWIFT-Toolbox

Tabelle 7: LOC-Statistik der SWIFT-Toolbox

Dateiname	Leerzeilen	Kommentarzeilen	Codezeilen
MT513MT515.java	461	2717	2307
MT502.java	237	2582	1474
MT535.java	205	844	1349
MT942.java	138	1191	556
MT536.java	40	274	536
MT940.java	104	924	464
Transaction.java	123	998	412
SWIFTUtil.java	18	144	299
MT535FinancialInstrument.java	127	1538	293
SWIFTParser.java	31	63	157
MT536FinancialInstrument.java	63	282	145
PartialTrade.java	43	390	107
SubBalance.java	45	613	96
FinancialInstrument.java	27	288	61
SWIFTValidateResult.java	9	71	55
MT942List.java	18	176	52
SettlementDetail.java	23	204	47
MT940List.java	14	151	46
Price.java	21	252	44
SettlementAmount.java	15	263	33
QuantityBalance.java	17	88	33
SWIFTField.java	14	56	31
SWIFTValidateResultList.java	12	89	28
Rate.java	12	202	26
Amount.java	12	133	25
NumberIdentification.java	10	125	19
Date.java	9	157	19
AppendingStringBuffer.java	7	27	16
SWIFTMissingFieldError.java	4	51	14
CountingStringBuffer.java	5	32	13
SWIFTValidateError.java	4	28	10
SWIFTParseException.java	5	51	7
HbciVersions.java	6	31	6
AbstractStringBuffer.java	4	17	5
Summe	1883	15052	8785

B.4 Laufzeit SWIFT-Toolbox

Tabelle 8: Laufzeiten der einzelnen Phasen bei der Übersetzung der SWIFT-Toolbox (in ms) [Vb = Vorbereitung, IP = Initiale Phase, Zp = Zwischenphase, Ap = Ausgabephase]

Versuchsnr.	Vb	IP	1. Zp	2. Zp	3. Zp	Ap	Σ
1	638	3181	1258	1395	701	2570	9743
2	533	3257	1210	1507	842	2388	9737
3	575	2969	1352	1586	890	2494	9866
4	542	3054	1622	1336	1245	2277	10076
5	634	3214	1301	1240	780	2571	9740
6	624	3080	1234	1353	814	2692	9797
7	543	3164	1416	1507	1018	2327	9975
8	626	3158	1322	1224	985	2444	9759
9	582	2968	1784	1415	969	2304	10022
10	611	3223	1268	1346	795	2775	10018

C Code, Grammatiken und andere Beispiele

C.1 Vordefinierte Attribute für Regeln in ANTLR

Vordefinierte Attribute für Lexerregeln:[156]

text:
: gesamter Text für den die Regel „*gematcht*“ wurde

type:
: Typ des „*gematchten*“ Token, eindeutige ID vom typ *int*

line:
: Zeilennummer im Quelltext in der das durch die Regel beschriebene Token „*gematcht*“ wurde

pos:
: Position in der Zeile *line* des Quelltextes des ersten Zeichens der „*gematchten*“ Zeichenkette

channel:
: Spezifziert den Kanal auf dem durch die Regel „*gematchte*“ Token hinterlegt werden sollen.

Vordefinierte Attribute für Parserregeln:[157]

text:
: gesamter Text für den die Regel „*gematcht*“ wurde (inklusive aller Zeichen, die auf anderen Kanälen liegen)

start:
: Erstes Token, das durch die Regel „*gematcht*“ wurde

stop:
: Letztes Token, das durch die Regel „*gematcht*“ wurde

tree:
: Der für die Regel spezifizierte Ausgabebaum (ASB)

[156] vgl. [Par07, Seite 131]
[157] vgl. [Par07, Seite 130]

st:
Das für die Regel spezifizierte Ausgabetemplate

Vordefinierte Attribute für Baumparserregeln:[158]

text:
Text für den ersten durch die Regel „*gematchten*“ Knoten

start:
Erster Knoten, der durch die Regel „*gematcht*“ wurde

st:
Das für die Regel spezifizierte Ausgabetemplate

[158] vgl. [Par07, Seite 132]

C.2 Anwendung der in der Konfiguration angegebenen Substitutionsregeln

Grammatikausschnitt aus der Dritten Zwischenphase

```
1  primaryExpression
2  @init
3  {
4    CustomTree tempTree = null;
5    Tripel<EMemberType, String, EMemberType> replacement = null;
6    CustomTree identTree = null;
7    boolean changedToMethodCall = false;
8  }
9    :    ^(d=DOT pe=primaryExpression id=IDENT)
10       {
11         // Abhandlung des Spezialfalles, dass eine Enumeration-Konstante
12         // eine Funktion direkt aufruft
13         // Dann wird hier ein Entsprechender Konstruktorbaum angelegt
14         if($pe.tree != null
15           && $pe.tree.resultValue != null
16           && $pe.tree.resultValue.getLeftSideType()
17                 .equals(ELeftSideType.STATIC_MEMBER)
18           && $pe.tree.resultValue.getObjectType().isEnumObj())
19         {
20           CustomTree packageTree = createTree(PACKAGE, "PACKAGE");
21           packageTree.addChild(createTree(IDENT, $pe.tree.resultValue.
                 getTypeNameSpace()));
22
23           CustomTree qualifiedTypeIdentTree = createTree(
                 QUALIFIED_TYPE_IDENT, "QUALIFIED_TYPE_IDENT");
24           qualifiedTypeIdentTree.addChild(createTree(IDENT, $pe.tree.
                 resultValue.getTypeName()));
25           qualifiedTypeIdentTree.addChild(packageTree);
26
27           CustomTree argumentTree
28             = createTree(ARGUMENT_LIST, "ARGUMENT_LIST");
29           CustomTree exprTree = createTree(EXPR, "EXPR");
30           exprTree.addChild($pe.tree);
31           argumentTree.addChild(exprTree);
32
33           tempTree = createTree(CLASS_CONSTRUCTOR_CALL, "
                 CLASS_CONSTRUCTOR_CALL");
34           tempTree.addChild(qualifiedTypeIdentTree);
35           tempTree.addChild(argumentTree);
36         }
37         else if($pe.tree != null
38           && $pe.tree.resultValue != null
39           && $pe.tree.resultValue.getLeftSideType().equals(ELeftSideType.
                 LITERAL_CONSTANT)
40           && $pe.tree.resultValue.getObjectType().isStringObj())
41         {
```

```
42            CustomTree packageTree = createTree(PACKAGE, "PACKAGE");
43            packageTree.addChild(createTree(IDENT, $pe.tree.resultValue.
                  getTypeNameSpace()));
44
45            CustomTree qualifiedTypeIdentTree = createTree(
                  QUALIFIED_TYPE_IDENT, "QUALIFIED_TYPE_IDENT");
46            qualifiedTypeIdentTree.addChild(createTree(IDENT, $pe.tree.
                  resultValue.getTypeName()));
47            qualifiedTypeIdentTree.addChild(packageTree);
48
49            CustomTree argumentTree = createTree(ARGUMENT_LIST, "
                  ARGUMENT_LIST");
50            CustomTree exprTree = createTree(EXPR, "EXPR");
51            exprTree.addChild($pe.tree);
52            argumentTree.addChild(exprTree);
53
54            tempTree = createTree(CLASS_CONSTRUCTOR_CALL, "
                  CLASS_CONSTRUCTOR_CALL");
55            tempTree.addChild(qualifiedTypeIdentTree);
56            tempTree.addChild(argumentTree);
57          }
58
59          $ArrayInitializerScope::parameterList.add($id.resultValue);
60          replacement = findSubstitute($d.leftArgValue, $d.resultValue);
61
62          if(replacement != null)
63          {
64            logger.debug("REPLACED");
65            identTree = createTree(IDENT, replacement.getSnd());
66            identTree.resultValue = $id.resultValue;
67            if(replacement.getFst().equals(EMemberType.VARIABLE_MEMBER)
68              && replacement.getThd().equals(EMemberType.METHOD_MEMBER))
69            {
70              logger.debug("REPLACED!");
71              changedToMethodCall = true;
72
73              logger.debug(tempTree != null && identTree != null &&
                    changedToMethodCall);
74            }
75          }
76
77        // JAVA-CODE-ENDE
78        }
79
80      -> {tempTree != null && identTree != null && changedToMethodCall}?
81
82        ^(METHOD_CALL ^(DOT {tempTree} {identTree}) ARGUMENT_LIST)
83
84      -> {tempTree != null && identTree != null}?
85
86        ^(DOT {tempTree} {identTree})
87
```

```
88      -> {tempTree != null}?
89
90        ^(DOT {tempTree} IDENT)
91
92      -> {identTree != null && changedToMethodCall}?
93
94        ^(METHOD_CALL ^(DOT primaryExpression {identTree}) ARGUMENT_LIST)
95
96      -> {identTree != null}?
97
98        ^(DOT primaryExpression {identTree})
99
100     -> ^(DOT primaryExpression IDENT)
```

Funktion *findSubstitute*

```
1   private final Tripel<EMemberType, String, EMemberType>
2                 findSubstitute(LeftSideValue oldLsv, LeftSideValue newLsv)
3   {
4     if(oldLsv != null
5       && newLsv != null
6       && ( newLsv.getLeftSideType().equals(ELeftSideType.MEMBER)
7         || newLsv.getLeftSideType().equals(ELeftSideType.MEMBER_FUNCTION)
8         || newLsv.getLeftSideType().equals(ELeftSideType.STATIC_FUNCTION)
9         || newLsv.getLeftSideType().equals(ELeftSideType.STATIC_MEMBER)))
10    {
11
12      logger.debug("SEARCH_FOR_REPLACEMENT:␣" + newLsv + "␣##########␣" +
            oldLsv);
13      EMemberType currMemberType = null;
14      if(newLsv.getLeftSideType().equals(ELeftSideType.MEMBER)
15        || newLsv.getLeftSideType().equals(ELeftSideType.STATIC_MEMBER))
16      {
17        currMemberType = EMemberType.VARIABLE_MEMBER;
18      }
19      else
20      {
21        currMemberType = EMemberType.METHOD_MEMBER;
22      }
23
24      String typeName = null;
25      String typeNameSpace = null;
26
27      final int dim = oldLsv.getDimension();
28
29      if(dim == 0)
30      {
31        typeName = oldLsv.getTypeName();
32        typeNameSpace = oldLsv.getTypeNameSpace();
33      }
34      else if(dim == 1)
```

```
35        {
36          typeName = arrayBaseClass.getFst();
37          typeNameSpace = arrayBaseClass.getSnd();
38        }
39        else if(dim == 2)
40        {
41          typeName = matrixBaseClass.getFst();
42          typeNameSpace = matrixBaseClass.getSnd();
43        }
44
45        Pair<String, EMemberType> mappedMember
46          = mappingDecoder.getSubstitute(typeName,
47                                         typeNameSpace,
48                                         newLsv.getQualifiedName(),
49                                         currMemberType);
50
51        if(mappedMember != null)
52        {
53          return new Tripel<E   MemberType, String, EMemberType>(
                currMemberType,
54                                                                  mappedMember.
                                                                      getFst(),
55                                                                  mappedMember.
                                                                      getSnd());
56        }
57      }
58      return null;
59    }
```

C.3 Code zur Java-Klasse InternalArrayList

```
1  package DTAToolbox.de.ppi.fis.travic.kernel.doNotTranslate;
2
3  import java.util.ArrayList;
4
5  public class InternalArrayList<E>
6  {
7    protected ArrayList<E> internalArray;
8
9    public InternalArrayList()
10   {
11     internalArray = new ArrayList<E>();
12   }
13
14   public int size()
15   {
16     return internalArray.size();
17   }
18
19   public E get(int index)
20   {
21     return internalArray.get(index);
22   }
23
24   public E[] toArray()
25   {
26     return null;
27   }
28
29   public void add(E paymExch)
30   {
31     internalArray.add(paymExch);
32   }
33
34   public void set(int pos, E elem)
35   {
36     this.internalArray.set(pos, elem);
37   }
38
39   public void addAll(InternalArrayList<E> notificationRecords)
40   {
41     for(int i=0; i < notificationRecords.size(); i++)
42     {
43       internalArray.add(notificationRecords.get(i));
44     }
45   }
46
47   public boolean isEmpty() {
48     return internalArray.isEmpty();
49   }
50 }
```

Beispiel einer Subklasse von *InternalArrayList*

```
1  package DTAToolbox.de.ppi.fis.travic.kernel.doNotTranslate;
2
3  public class StringList extends InternalArrayList<String>
4  {
5    public String[] toArray()
6    {
7      return internalArray.toArray(new String[size()]);
8    }
9  }
```

C.4 Grammatikdefinition für Konfigurationsdateien

```
1  grammar MappingFile;
2
3  options{
4      backtrack=true;
5      memoize=true;
6  }
7
8  tokens {
9      //ANDERE Zeichen
10     ASSIGN               = '=';
11     ASSIGN_EQUAL         = ':=';
12     COMMA                = ',';
13     OPENING_MAP_BRACE    = '{';
14     CLOSING_MAP_BRACE    = '}';
15 }
16
17 @header{
18     package j2cpp;
19     import java.util.HashMap;
20     import java.util.ArrayList;
21 }
22
23 @members{
24     private HashMap myHashMap  = new HashMap();
25
26     public HashMap getMapping()
27     {
28         return myHashMap;
29     }
30 }
31
32 @lexer::header{
33 package j2cpp;
34 }
35
36 // Mapping-Grammar-Parser
37 // START-Regel
38 mappingStart
39     :   mappingDefinition*
40     ;
41
42 mappingDefinition
43     :   IDENT ASSIGN_EQUAL mapDefRs=mappingDefinitionRightSide
44             {
45             // JAVA-CODE-START
46                 myHashMap.put($IDENT.text, $mapDefRs.retVal);
47             // JAVA-CODE-ENDE
48             }
49     |   IDENT ASSIGN identList
50             {
```

```
51              // JAVA-CODE-START
52                  myHashMap.put($IDENT.text, $identList.retKeyList);
53              // JAVA-CODE-ENDE
54              }
55      ;
56  identList returns [ArrayList<String> retKeyList]
57  @init{
58      $retKeyList = new ArrayList<String>();
59  }
60      :   OPENING_MAP_BRACE id1=IDENT
61              {
62              // JAVA-CODE-START
63                  $retKeyList.add($id1.text);
64              // JAVA-CODE-ENDE
65              }
66          (COMMA id2=IDENT
67              {
68              // JAVA-CODE-START
69                  $retKeyList.add($id2.text);
70              // JAVA-CODE-ENDE
71              }
72          )* CLOSING_MAP_BRACE
73      ;
74  objectList returns [ArrayList<Object> retList]
75  @init{
76      $retList = new ArrayList<Object>();
77  }
78      :   OPENING_MAP_BRACE
79          (   ob1=object        { $retList.add($ob1.retObject); }
80              (COMMA ob2=object { $retList.add($ob2.retObject); }
81              )*
82          )?
83          CLOSING_MAP_BRACE
84      ;
85  object returns [Object retObject]
86  @init{
87      $retObject = null;
88  }
89      :   IDENT       { $retObject = $IDENT.text; }
90      |   objectList  { $retObject = $objectList.retList; }
91      |   mappingList { $retObject = $mappingList.retVal; }
92      ;
93
94  mappingDefinitionRightSide returns [Object retVal]
95      :   mappingList
96              {
97              // JAVA-CODE-START
98                  $retVal = $mappingList.retVal;
99              // JAVA-CODE-ENDE
100             }
101     |   IDENT
102             {
```

```
103             // JAVA-CODE-START
104                 $retVal = $IDENT.text;
105             // JAVA-CODE-ENDE
106             }
107     ;
108 mappingList returns [Object retVal]
109 scope{
110     HashMap scopeMap;
111 }
112 @init{
113     $mappingList::scopeMap = new HashMap();
114 }
115 @after{
116     $retVal = $mappingList::scopeMap;
117 }
118     :   OPENING_MAP_BRACE
119         mappingValueDefinition
120         (COMMA mappingValueDefinition)*
121         CLOSING_MAP_BRACE
122     ;
123 mappingValueDefinition
124     :   idList=identList ASSIGN id2=IDENT
125             {
126             // JAVA-CODE-START
127                 $mappingList::scopeMap.put($idList.retKeyList, $id2.text);
128             // JAVA-CODE-ENDE
129             }
130     |   idList=identList ASSIGN map=mappingList
131             {
132             // JAVA-CODE-START
133                 $mappingList::scopeMap.put($idList.retKeyList, $map.retVal
                        );
134             // JAVA-CODE-ENDE
135             }
136     |   idList=identList ASSIGN objects=objectList
137             {
138             // JAVA-CODE-START
139                 $mappingList::scopeMap.put($idList.retKeyList, $objects.
                        retList);
140             // JAVA-CODE-ENDE
141             }
142     |   id1=IDENT ASSIGN id2=IDENT
143             {
144             // JAVA-CODE-START
145                 $mappingList::scopeMap.put($id1.text, $id2.text);
146             // JAVA-CODE-ENDE
147             }
148     |   id=IDENT ASSIGN objects=objectList
149             {
150             // JAVA-CODE-START
151                 $mappingList::scopeMap.put($id.text, $objects.retList);
152             // JAVA-CODE-ENDE
```

```
153                 }
154         |   id=IDENT ASSIGN map=mappingList
155                 {
156                 // JAVA-CODE-START
157                     $mappingList::scopeMap.put($id.text, $map.retVal);
158                 // JAVA-CODE-ENDE
159                 }
160         ;
161 // Mapping-File-Grammar-Lexer
162 IDENT
163         :   (   'a'..'z'
164             |   'A'..'Z'
165             |   '.'
166             |   '-'
167             |   '_'
168             |   '<'
169             |   '>'
170             |   '\"'
171             |   '\''
172             |   '#'
173             |   '*'
174             |   '+'
175             |   '/'
176             |   '0'..'9'
177             |   '&'
178             |   '%'
179             )+
180         ;
181 WS  :  (' '|'\r'|'\t'|'\u000C'|'\n')
182             {
183             //JAVA-CODE-START
184                 skip();
185             //JAVA-CODE-ENDE
186             }
187         ;
188 COMMENT
189         :   '/*' ( options {greedy=false;} : . )* '*/'
190             {
191             //JAVA-CODE-START
192                 skip();
193             //JAVA-CODE-ENDE
194             }
195         ;
196 LINE_COMMENT
197         : '//' ~('\n'|'\r')* '\r'? '\n'
198             {
199             //JAVA-CODE-START
200                 skip();
201             //JAVA-CODE-ENDE
202             }
203         ;
```

C.5 Beispiel einer Konfigurationsdatei

```
1  // Namensraummapping
2  namespaces := {utils     = utils,
3                 java.lang = utils,
4                 java.io   = utils,
5                 java.util = utils,
6                 %         = myProject,
7                 *         = myProject}
8  // Typenmapping
9  types := {java.lang.Exception
10            = {className = BaseException,
11               namespace = utils
12              },
13           java.lang.StringBuffer
14            = {className = String,
15               namespace = utils
16              }
17          }
18 // Funktionmapping
19 substitute :={{String, utils}
20                 = {oldName       = toString,
21                    oldMemberType = METHOD_MEMBER,
22                    newName       = %,
23                    newMemberType = %
24                   },
25                {String, utils}
26                 = {oldName       = length,
27                    oldMemberType = METHOD_MEMBER,
28                    newName       = getLength,
29                    newMemberType = METHOD_MEMBER
30                   },
31                {HeapArray, utils}
32                 = {oldName       = length,
33                    oldMemberType = VARIABLE_MEMBER,
34                    newName       = getLength,
35                    newMemberType = METHOD_MEMBER
36                   },
37                {HeapMatrix, utils}
38                 = {oldName       = length,
39                    oldMemberType = VARIABLE_MEMBER,
40                    newName       = getLength,
41                    newMemberType = METHOD_MEMBER
42                   }
43               }
44 // Basisklassen fuer die verschiedenen Objekttypen
45 baseClasses := {HEAP_OBJ       = {className   =  HeapObject,
46                                   namespace   =  utils
47                                  },
48                 EXCEPTION_OBJ  = {className   =  BaseException,
49                                   namespace   =  utils
50                                  },
```

```
51                  VALUE_OBJ       = {className    =   Printable,
52                                     namespace    =   utils
53                                    },
54                  STRING_OBJ      = {className    =   String,
55                                     namespace    =   utils
56                                    },
57                  ENUM_OBJ        = {className    =   Enumeration,
58                                     namespace    =   utils
59                                    },
60                  ARRAY_OBJ       = {className    =   HeapArray,
61                                     namespace    =   utils
62                                    },
63                  MATRIX_OBJ      = {className    =   HeapMatrix,
64                                     namespace    =   utils
65                                    }
66                }
67 // Klasseninformationen ueber Klassen, die so nicht in Java existieren
68 classes:={{String,utils}
69             ={objectType  = STRING_OBJ,
70               classType   = CLASS,
71               classMember = { { memberName    = valueOf,
72                                 memberType    = METHOD_MEMBER,
73                                 visibility    = PUBLIC,
74                                 isStatic      = true,
75                                 methParameter = {0 = {type = *,
76                                                       dim  = 0}},
77                                 returnValue   = {className = String,
78                                                  namespace = utils,
79                                                  dim       = 0}
80                               },
81                               { memberName    = toCharArray,
82                                 memberType    = METHOD_MEMBER,
83                                 visibility    = PUBLIC,
84                                 returnValue   = {type = char,
85                                                  dim   = 1}
86                               },
87                               { memberName    =  equalsIgnoreCase,
88                                 memberType    =  METHOD_MEMBER,
89                                 visibility    =  PUBLIC,
90                                 methParameter =  {0 = {className = String,
91                                                        namespace = utils,
92                                                        dim = 0}},
93                                 returnValue   =  {type = boolean, dim = 0}
94                               }
95             },
96
97            {BaseException,utils}
98              ={objectType   = EXCEPTION_OBJ,
99                classType    = CLASS,
100               classMember  = { { memberName  = getMessage,
101                                  memberType  = METHOD_MEMBER,
102                                  visibility  = PUBLIC,
```

```
103                                   returnValue = {className = String,
104                                                  namespace = utils,
105                                                  dim = 0}
106                                 },
107                                 { memberName  = printStackTrace,
108                                   memberType  =  METHOD_MEMBER,
109                                   visibility  =  PUBLIC
110                                 },
111                                 { memberName  =  toString,
112                                   memberType  =  METHOD_MEMBER,
113                                   visibility  =  PUBLIC,
114                                   returnValue =  {className = String,
115                                                   namespace = utils,
116                                                   dim       = 0}
117                                 }
118                               }
119             },
120
121             {HeapObject,utils}
122               ={objectType    = HEAP_OBJ,
123                 classType     = CLASS,
124                 classMember   = { { memberName  =  toString,
125                                     memberType  =  METHOD_MEMBER,
126                                     visibility  =  PUBLIC,
127                                     returnValue =  {className = String,
128                                                     namespace = utils,
129                                                     dim = 0}
130                                   },
131                                   { memberName  =  equals,
132                                     memberType  =  METHOD_MEMBER,
133                                     visibility  =  PUBLIC,
134                                     methParameter =  {0 = {type = *,
135                                                            dim  = 0}},
136                                     returnValue =  {type = boolean,
137                                                     dim  = 0}
138                                   }
139                               }
140               }
141 }
```

C.6 StringTemplate's

```
1  group Cpp;
2
3  /**
4   * Map, die als Schluessel die Namen der primitiven Java-Datentypen
        enthaelt
5   * und deren zugeordnete Werte den passenden in C++ entspricht
6   */
7  primitiveTypeMap ::= [
8          "int":"int",
9          "long":"long",
10         "float":"float",
11         "double":"double",
12         "boolean":"bool",
13         "byte":"byte",
14         "short":"short",
15         "char":"char",
16         default:"null"
17 ]
18
19
20
21 /**
22  * Template fuer Klassendeklarationen in C++
23  *
24  * @param classInfo      Objekt vom Typ ClassInformation,
25  *                       dass Informationen ber die Klasse bereitstellt
26  * @param extendClause   Superklassen (als StringTemplate)
27  * @param classBody      Rumpf der Klassendeklaration (als StringTemplate)
28  */
29 classDeclarationTemplate(className,
30                          extendClause,
31                          classBody)
32 ::= <<
33 <! Klassendeklaration/-definition !>
34 class <className><extendClause>
35 {
36 <! Rumpf der Klasse !>
37 <classBody>
38 };
39 >>
40
41
42
43
44
45 /**
46  * Template fuer die Ausgabe der Klassen die erweitert werden
47  *
48  * @param extendClauses die erweiterten Klassen (als Liste von
        StringTemplate's)
```

```
49   */
50  extendClauseTemplate(extendClauses) ::= <<
51  <if(extendClauses)> : <extendClauses; separator=", "><endif>
52  >>
53
54  /**
55   * Template fuer die Ausgabe einer Klasse die von dieser Klasse erweitert
        wird
56   *
57   * @param visibilityType         Sichtbarkeit der Vererbung (als
        StringTemplate)
58   * @param extendClassName       Name der erweiterten Klasse
59   * @param extendClassNameSpace  Namensraum der erweiterten Klasse
60   */
61  extendTypeTemplate(visibilityType,
62                     extendClassName,
63                     extendClassNameSpace)
64  ::= <<
65  <visibilityType> <if(extendClassNameSpace)><extendClassNameSpace>::<endif
       ><extendClassName>
66  >>
67
68
69
70
71  /**
72   * Template zur Auflistung von Bestandteilen einer Klassendeklaration,
73   * die als Objekttyp HEAP_OBJ hat
74   *
75   * @param className           Name der Klasse
76   * @param memberList          Liste der aller Member/Methoden
77   * @param heapObjectNameSpace Namensraum des HeapObjects
78   */
79  heapObjClassScopeDeclsTemplate(memberList, className, heapObjectNameSpace)
        ::= <<
80  public:
81      typedef <if(heapObjectNameSpace)><heapObjectNameSpace>::<endif>
           HeapObjectPtr\<<className>\> Ptr;
82  <memberList; separator="\n">
83  >>
84
85  /**
86   * Template zur Auflistung von Bestandteilen einer Klassendeklaration,
87   * die als Objekttyp nicht HEAP_OBJ hat
88   *
89   * @param memberList Liste der aller Member/Methoden
90   */
91  classScopeDeclsTemplate(memberList) ::= <<
92  <memberList; separator="\n">
93  >>
94
95  /**
```

```
 96   * Template fuer eine Liste von Include-Direktiven einer Klasse
 97   *
 98   * @param includeList Einzubindende Header-Dateien
 99   */
100  includeListTemplate(includeList) ::= <<
101  <includeList; separator="\n">
102  >>
103
104  /**
105   * Template zur Erstellung einer einzelnen Include-Direktive einer Klasse
106   *
107   * @param includeDirective Dateiname (inkl. Pfad) zu der angegebenen
         Klasse
108   */
109  includeTemplate(includeDirective) ::= <<
110  #include "<includeDirective>"
111  >>
112
113  /**
114   * Template zur Erstellung einer Liste von Using-Direktiven
115   *
116   * @param usingList Liste der Namensraeume die per Using-Direktive
         eingebunden werden sollen
117   */
118  usingListTemplate(usingList) ::= <<
119  <usingList; separator="\n">
120  >>
121
122  /**
123   * Template zur Erstellung einer einzelnen Using-Direktive einer Klasse
124   *
125   * @param usingName Name des Namensraums
126   */
127  usingTemplate(usingName) ::= <<
128  using namespace <usingName>;
129  >>
130
131  /**
132   * Template mit Ausgabe <code>classname::nameSpace</code>
133   *
134   * @param classname der Klassenname
135   * @param nameSpace der Namensraum
136   */
137  classIdentifierTemplate(classname, nameSpace) ::= <<
138  <if(nameSpace)><nameSpace>::<endif><classname>
139  >>
140
141  /**
142   * Template fuer einen Methodenaufruf
143   *
144   * @param methodName Name der Methode
145   * @param args       Argumente der Methode
```

```
146   */
147  methodCallTemplate(methodName, args) ::= <<
148  <methodName>(<args>)
149  >>
150
151  /**
152   * Template fuer ein Cpp-File
153   *
154   * @param classImpl Liste der im Cpp-File implementierten Methoden, etc.
155   * @param headerName Name der Headerdatei
156   * @param usings Usingdirektiven
157   */
158  cppImplementationFileTemplate(classImpl, headerName, usings) ::= <<
159  <! Include-Anweisungen !>
160  <includeTemplate(headerName)>
161
162  <usings>
163
164  <classImpl>
165
166  >>
167
168  /**
169   * Template fuer eine Liste von Argumenten
170   *
171   * @param argList die Liste der Argumente
172   */
173  argumentListTemplate(argList) ::= <<
174  <argList; separator=", ">
175  >>
176
177  /**
178   * Template fuer eine Liste von Membern/Methoden, die ins Cpp-File sollen
179   *
180   * @param membersImplList die Liste der Member, die ins Cpp-File sollen
181   */
182  classScopeImplsTemplate(membersImplList) ::= <<
183  <membersImplList; separator="\n\n">
184  >>
185
186  /**
187   * Template fuer den Leer-String-Konstruktor
188   *
189   * @param classname der Klassenname der String-Klasse
190   * @param nameSpace der Namensraum der String-Klasse
191   */
192  emptyStringConstructorTemplate(classname, nameSpace) ::= <<
193  <if(nameSpace)><nameSpace>::<endif><classname>()
194  >>
195
196  /**
197   * Template fuer den Aufruf eines Konstruktors von einem HeapObject
```

```
198   *
199   * @param className Name der Klasse
200   * @param nameSpace Namensraum der Klasse
201   * @param args      Argumente des Konstruktoraufrufes
202   */
203  heapObjConstructorCall(className, nameSpace, args) ::= <<
204  <if(nameSpace)><nameSpace>::<endif><className>::create(<args>)
205  >>
206
207  /**
208   * Template fuer den Aufruf eines Konstruktors von einem ValueObject
209   *
210   * @param className Name der Klasse
211   * @param nameSpace Namensraum der Klasse
212   * @param args      Argumente des Konstruktoraufrufes
213   */
214  valueObjConstructorCall(className, nameSpace, args) ::= <<
215  <if(nameSpace)><nameSpace>::<endif><className>(<args>)
216  >>
217
218  /**
219   * Template fuer den Aufruf eines Konstruktors von einem ExceptionObject
220   *
221   * @param className Name der Klasse
222   * @param nameSpace Namensraum der Klasse
223   * @param args      Argumente des Konstruktoraufrufes
224   */
225  exceptionObjConstructorCall(className, nameSpace, args) ::= <<
226  <valueObjConstructorCall(...)>
227  >>
228
229  /**
230   * Template fuer den Aufruf eines Konstruktors von einem StringObject
231   *
232   * @param className Name der Klasse
233   * @param nameSpace Namensraum der Klasse
234   * @param args      Argumente des Konstruktoraufrufes
235   */
236  stringObjConstructorCall(className, nameSpace, args) ::= <<
237  <valueObjConstructorCall(...)>
238  >>
239
240  /**
241   * Template fuer den Aufruf eines Konstruktors von einem EnumObject
242   *
243   * @param className Name der Klasse
244   * @param nameSpace Namensraum der Klasse
245   * @param args      Argumente des Konstruktoraufrufes
246   */
247  enumObjConstructorCall(className, nameSpace, args) ::= <<
248  <valueObjConstructorCall(...)>
249  >>
```

```
250
251 /**
252  * Template fuer den aufruf des Super-Konstruktors
253  * (nur fuer Initailisierungsliste des Konstruktors)
254  *
255  * @param className      Name der Vaterklasse
256  * @param classNameSpace Namensraum der Klasse
257  * @param args           Argumente des Konstruktoraufrufs
258  */
259 superConstructorCallTemplate(className, classNameSpace, args) ::= <<
260 <if(classNameSpace)><classNameSpace>::<endif><className>(<args>)
261 >>
262
263 /**
264  * Template fuer einen Arrayzugriff
265  *
266  * @param arrayToAccess das Array auf das zugegriffen werden soll
267  * @param accessor      der Zugriffsspezifizierer
268  */
269 arrayAccessTemplate(arrayToAccess, accessor) ::= <<
270 <arrayToAccess>->at(<accessor; separator=", ">)
271 >>
272
273 localVariableDeclarationTemplate(variableNameAndInit, variableType) ::= <<
274 <variableType> <variableNameAndInit>
275 >>
276
277 thisTemplate() ::= <<
278 this
279 >>
280
281 commentListTemplate(commentList, oldTemplate) ::= <<
282 <if(commentList)>
283 <commentList; separator="\n">
284 <endif><oldTemplate>
285 >>
286
287 nullTemplate() ::= <<
288 null
289 >>
290
291 /**
292  * Template fuer den Klassenrumpf einer Enumeration
293  *
294  * @param className    Name der Klasse
295  * @param extendClause List der Superklassen
296  * @param classBody    Rumpf der Klasse
297  */
298 enumerationDeclarationTemplate(className, extendClause, classBody) ::= <<
299 <classDeclarationTemplate(...)>
300 >>
301
```

```
302 /**
303  * Template fuer eine Enumeration-Klasse im Java-Stil
304  *
305  * @param enumerationClassName Name der Enumeration
306  * @param enumConstants        Liste der Enumeration-Konstanten
307  * @param restOfClass          Rest der Klasse (Methoden, Member, etc)
308  */
309 enumScopeDeclsTemplate(enumerationClassName, enumConstants, restOfClass)
       ::= <<
310 public:
311     enum Value {
312         <enumConstants; separator=",\n">,
313         ENUM_NULL
314     };
315
316     <enumerationClassName>(Value rc)
317         : rc(rc)
318     {}
319
320     <enumerationClassName>()
321         : rc(ENUM_NULL)
322     {}
323
324     operator Value() const
325     {
326         return rc;
327     }
328
329     operator int() const
330     {
331         return rc;
332     }
333
334 private:
335     Value rc;
336
337 public:
338
339     /**
340      * Generierte valueOf()-Funktion im Java-Stil
341      */
342     <enumerationValueOfFun(enumerationClassName = enumerationClassName,
          enumConstants = enumConstants)>
343
344     /**
345      * Generierte compareTo-Funktion im Java-Stil
346      */
347     int compareTo(Value rc)
348     {
349         if((*this).rc == rc)
350         {
351             return 0;
```

```
352         }
353         else if((*this).rc \< rc)
354         {
355             return -1;
356         }
357         else
358         {
359             return 1;
360         }
361     }
362 
363     /**
364      * Generierte toString()-Funktion im Java-Stil
365      */
366     String toStringDefault()
367     {
368         switch((*this).rc)
369         {
370         <enumConstants:enumerationToStringDefaultFunCase()>
371         default:
372             return String(null);
373         }
374     }
375 
376     /**
377      * Funktion, die zurueckgibt, ob die Enumeration nicht \<code\>null\</
           code\> ist
378      */
379     bool isValid()
380     {
381         return rc == ENUM_NULL;
382     }
383 
384     /**
385      * Funktion, die Enumeration auf \<code\>null\</code\> setzt
386      */
387     void invalidate()
388     {
389         rc = ENUM_NULL;
390     }
391 
392      /**
393       * Liefert den Integerwert des Enums zurueck
394       */
395     int ordinal()
396     {
397         return rc;
398     }
399 
400 <restOfClass>
401 >>
402 
```

```
403 enumerationToStringDefaultFunCase() ::= <<
404 case <it>:
405     return String("<it>");
406
407 >>
408
409 enumerationValueOfFun(enumerationClassName, enumConstants) ::= <<
410 static <enumerationClassName> valueOf(String stringRep)
411 {
412     <enumConstants:ifClauseEnumValueFun(className = enumerationClassName)>
413     return ENUM_NULL;
414 }
415 >>
416
417 ifClauseEnumValueFun(className) ::= <<
418 if(<className>(<it>).toString() == stringRep)
419 {
420     return <it>;
421 }
422
423 >>
424 /**
425  * Leer-Template
426  *
427  */
428 epsilonTemplate() ::= <<
429 >>
430
431
432 /**
433  * Template fuer einen geklammerten Ausdruck
434  *
435  * @param expr zu klammernder Ausdruck
436  */
437 parenthesisTemplate(expr) ::= <<
438 (<expr>)
439 >>
440
441
442 /**
443  * Template fuer eine zweistellige Operation
444  *
445  * @param arg1 erstes Argument der Operation
446  * @param arg2 zweites Argument der Operation
447  * @param op   Operator, der <code>arg1</code>
448                und <code>arg2</code> miteinander verknuepft
449  */
450 binaryOperatorTemplate(arg1, arg2, op) ::= <<
451 <arg1> <op> <arg2>
452 >>
453
454
```

```
455 /**
456  * Template, das zwei Strings miteinander verknuepft
457  *
458  * @param string1 der erste String
459  * @param string2 der zweite String
460  */
461 stringConcatTemplate(string1, string2) ::= <<
462 <string1> \<\< <string2>
463 >>
464
465
466 /**
467  * Template fuer einen einstelligen Operator
468  *
469  * @param arg Argument der Operation
470  * @param op Operator, der <code>arg</code> voran/hinteran gestellt wird
471  * @param praefix Flag, das angibt ob <code>op</code> ein Praefixoperator
       ist
472  */
473 unaryOperatorTemplate(arg, op, praefix) ::= <<
474 <if(praefix)><op><arg><else><arg><op><endif>
475 >>
476
477
478 /**
479  * Template fuer eine dreistellige Operation mit zwei Operatoren
480  *
481  * @param arg1 erstes Argument der Operation
482  * @param arg2 zweites Argument der Operation
483  * @param arg3 drittes Argument der Operation
484  * @param op1 Operator, der <code>arg1</code>
485               und <code>arg2</code> miteinander verknuepft
486  * @param op2 Operator, der <code>arg2</code>
487               und <code>arg3</code> miteinander verknuepft
488  */
489 threeDigitOperatorTemplate(arg1,arg2,arg3,op1,op2) ::= <<
490 <arg1> <op1> <arg2> <op2> <arg3>
491 >>
492
493 /**
494  * Template, das eine Modifierliste erzeugt
495  *
496  * @param visibility Sichtbarkeitsqualifizierer
497  * @param virtual    Flag, ob virtuell
498  * @param stat       Flag, ob statisch
499  * @param fin        Flag, ob final
500  * @param explicit   Flag, ob explicit
501  */
502 modifierListTemplate(visibility,
503                      virtual,
504                      stat,
505                      fin,
```

```
506                         explicit)
507 ::= <<
508 <if(visibility)>
509 <visibility>:<endif>
510 <\ ><\ ><\ ><\ ><if(explicit)>explicit <endif><if(virtual)>virtual <endif
        ><if(stat)>static <endif><if(fin)><endif>
511 >>
512
513 /**
514  * Template zu Generierun des Anfangstrings fuer eine
        Namensraumdeklaration
515  *
516  * @param namespaceName Name des Namensraums
517  */
518 namespaceTemplate(namespaceName) ::= <<
519 namespace <namespaceName>
520 >>
521
522 /**
523  * Template das eine Variablendeklaration auf Klassenebene vornimmt
524  *
525  * @param modList       die Modifizierer
526  * @param varaibleType der Typ der Variable
527  * @param variableName der Name der Variablen
528  */
529 variableDeclarationTemplate(modList,
530                             variableType,
531                             variableName) ::= <<
532 <if(modList)><modList><else>    <endif><variableType> <variableName>;
533
534 >>
535
536
537 /**
538  * Template, das eine Intialisierung einer statischen variable in einem
539  * Cpp-File vornimmt
540  *
541  * @param modList                       die Modifizierer
542  * @param variableType                  der Typ der statischen Variable
543  * @param className                     der Klassenname der Klasse in der
        die statische Variable deklariert wurde
544  * @param classNameSpace                der zu <code>className zugehoerige
        Namensraum</code>
545  * @param variableWithInitialisation    der Variablenname und zusaetzlich
        der initiale Wert der Variablen
546  */
547 staticVariableMemberImplTemplate(modList,
548                                  variableType,
549                                  className,
550                                  classNameSpace,
551                                  variableWithInitialisation)
552 ::= <<
```

```
553 <!<modList>!><variableType> <if(classNameSpace)><classNameSpace>::<endif><
        className>::<variableWithInitialisation>;
554 >>
555
556 /**
557  * Template zur Erstellung von Variablen mit gleichzeitiger optionaler
558  * Initialwertzuweisung
559  *
560  * @param variableName          Name der Variablen
561  * @param variableInitialValue  der initiale Wert der Variablen
562  */
563 variableInitializerTemplate(variableName, variableInitialValue) ::= <<
564 <variableName><variableInitialValue>
565 >>
566
567 initialValueTemplate(initialValue) ::= <<
568  = <initialValue>
569 >>
570
571 /**
572  * Template zur Erstellung von Methoden im Header
573  *
574  * @param modifierList       Modifier der Methode
575  * @param methodReturnType   Rueckgabewert der Methode
576  * @param methodName         Name der Methode
577  * @param methodParams       Parameter der Methode
578  * @param isAbstract         Flag, ob Methode abstrakt ist
579  * @param methodBody         Rumpf der Methode
580  */
581 methodHeaderTemplate(modifierList,
582                      methodReturnType,
583                      methodName,
584                      methodParams,
585                      isAbstract,
586                      methodBody)
587 ::= <<
588 <if(modifierList)><modifierList><else>     <endif><if(methodReturnType)><
        methodReturnType> <else>void <endif><methodName>(<methodParams>)<if(
        isAbstract)>=0<endif><if(methodBody)>
589
590     <methodBody><else>;<endif>
591
592 >>
593
594 /**
595  * Template zur Erstellung von Methoden im Cpp-File
596  *
597  * @param modifierList       Modifier der Methode            // TODO
598  * @param methodReturnType   Rueckgabewert der Methode
599  * @param methodName         Name der Methode
600  * @param methodParams       Parameter der Methode
601  * @param methodBody         Rumpf der Methode
```

```
602   * @param className          Name der Klasse in der die Methode
          implementiert wurde
603   * @param classNameSpace     Namensraum der Klasse in der die Methode
          implementiert wurde
604   */
605  methodImplTemplate(methodName, methodReturnType, methodParams, methodBody,
          className, classNameSpace) ::= <<
606  <if(methodReturnType)><methodReturnType><else>void<endif> <if(
          classNameSpace)><classNameSpace>::<endif><className>::<methodName>(<
          methodParams>)
607  <methodBody>
608
609  >>
610
611  /**
612   * Template zur Erstellung von Main-Methoden
613   *
614   * @param className Name der Klasse
615   */
616  // TODO: Parameteruebergabe
617  mainMethodTemplate(className,mmodifierList,mmethodName, mmethodReturnType,
          mmethodParams,mmethodBody) ::= <<
618  <if(mmethodReturnType)><mmethodReturnType><else>void<endif> <className>::<
          mmethodName>(<mmethodParams>)
619  <mmethodBody>
620
621
622  int main(int argc, const char* argv[])
623  {
624      HeapArray\<String\>::Ptr args = HeapArray\<String\>::create();
625      for(int i=1; i \<argc; i++)
626      {
627
628          args->append(String(argv[i]));
629      }
630      Main::main(args);
631      return 0;
632  }
633
634  >>
635
636  /**
637   * Template zur Erstellung von Konstruktoren
638   *
639   * @param modifierList       Modifier des Kosntruktors
640   * @param className          Name der Klasse in der der Konstruktor
          implementiert wurde
641   * @param constrParams       Parameter des Konstruktors
642   * @param constrBody         Rumpf des Konstruktors
643   * @param initializerList    Initialisierungsliste des Konstruktors
644   */
645  constructorTemplate(modifierList,
```

```
646                     className,
647                     constrParams,
648                     constrBody,
649                     initializerList)
650 ::= <<
651 <if(modifierList)><modifierList><else>    <endif><className>(<constrParams
        >)<initializerList>
652     <constrBody>
653
654 >>
655
656 /**
657  * Template zur Erstellung von Konstruktoren fuer HeapObjekte
658  *
659  * @param modifierList      Modifier des Kosntruktors
660  * @param visibilityFacFun  die Sichtbarkeit fuer die Factory-Funktion (
        als String)
661  * @param className         Name der Klasse in der der Konstruktor
        implementiert wurde
662  * @param constrParams      Parameter des Konstruktors
663  * @param constrBody        Rumpf des Konstruktors
664  * @param initializerList   Initialisierungsliste des Konstruktors
665  */
666 heapConstructorTemplate(heapModifierList,
667                         visibilityFacFun,
668                         heapClassName,
669                         heapConstrParams,
670                         heapConstrBody,
671                         heapInitializerList,
672                         paramsFacFun)
673 ::= <<
674 <constructorTemplate(modifierList=heapModifierList,className=heapClassName
        ,constrParams=heapConstrParams,constrBody=heapConstrBody,
        initializerList=heapInitializerList)>
675 <factoryFuncHeaderTemplate(methodParams=heapConstrParams,className=
        heapClassName,visibility=visibilityFacFun,paramList=paramsFacFun)>
676 >>
677
678
679 factoryFuncHeaderTemplate(methodParams, className, visibility, paramList)
        ::= <<
680 <if(visibility)>
681
682 <visibility>:
683
684 <endif>    static Ptr create(<methodParams>)
685     <factoryBlockTemplate(classNameThis=className,args=paramList)>
686
687 >>
688
689 factoryBlockTemplate(classNameThis, args) ::= <<
690 {
```

```
691       return NEW_HEAPOBJECT(<className>, <if(args)><args; separator=", "><
              else>NOARGS<endif>);
692   }
693   >>
694
695   /**
696    * Template fuer die Initialsierungsliste eines Konstruktors
697    *
698    * @param explSuperConstructorCall  Aufruf eines Superkonstruktors
699    * @param memberInitializers        Variableninitialisierung
700    */
701   initializerListTemplate(explSuperConstructorCall, memberInitializers) ::=
          <<
702   <if(explSuperConstructorCall)> : <explSuperConstructorCall><endif><if(
          memberInitializers)><if(explSuperConstructorCall)>, <else> : <endif><
          memberInitializers; separator=", "><endif>
703   >>
704
705   /**
706    * Template fuer ein Intialisierer einer Membervariablen in einer
           Intialisierungsliste
707    *
708    * @param identifierName  Name der zu initialisierenden Variable
709    * @param identifierValue zuzuweisender Wert
710    */
711   initializerTemplate(identifierName, identifierValue) ::= <<
712   <identifierName>(<identifierValue>)
713   >>
714
715   /**
716    * Template fuer Parameter
717    *
718    * @param parameterName Name des Parameters
719    * @param parameterType Typ des Parameters
720    * @param typePostfix   Postfix, dass nach <code>parameterType</code>
           eingefuehrt wird
721    */
722   parameterTemplate(parameterName, parameterType, typePostfix) ::= <<
723   <parameterType><typePostfix> <parameterName>
724   >>
725
726   /**
727    * Template fuer den boolschen Wert <code>true</code>
728    */
729   trueTemplate() ::= <<
730   true
731   >>
732
733   /**
734    * Template fuer den boolschen Wert <code>false</code>
735    */
736   falseTemplate() ::= <<
```

```
737 false
738 >>
739
740 /**
741  * Template fuer eine C++-Header-Datei
742  *
743  * @param definedClassWithBody definierten Klasse (als StringTemplate)
744  * @param includeWatchString String fuer Inkludewaechteranweisung
745  * @param includes Inkludedirektiven (als StringTemplate)
746  * @param usingDirectives Usingdirektiven (als StringTemplate)
747  * @param classNameSpace Namensraum in dem die Klasse liegt (namespace <
        Namensraum>)
748  */
749 cppHeaderFileTemplate(definedClassWithBody,
750                       includeWatchString,
751                       includes,
752                       usingDirectives,
753                       classNameSpace)
754 ::= <<
755 <! Inkludewaechter !>
756 #ifndef <includeWatchString>_HPP
757 #define <includeWatchString>_HPP
758
759 <! Import-Anweisungen !>
760 <includes>
761 <if(usingDirectives)>
762
763 <usingDirectives><! Usingdirektiven !>
764
765 <endif>
766
767 <classNameSpace><! Namensraum in dem die Klasse sich befindet !>
768 {
769
770 <definedClassWithBody><! Klassenrumpf !>
771
772 } // <classNameSpace>
773
774 #endif // <includeWatchString>_HPP
775
776 >>
777
778 /**
779  * Template, das die Verwendung des friends-Schluesselwortes ermoeglicht
780  *
781  * @param className Name der Klasse mit der eine Freundschaft bestehen
        soll
782  */
783 friendTemplate(className) ::= <<
784 friend class <className>;
785 >>
786
```

```
787 /**
788  * Template, das eine Liste von Freundesdeklarationen aufbereitet
789  *
790  * @param friends die Liste der aufzubereitenden Freundesdeklarationen
791  */
792 friendListTemplate(friends) ::= <<
793 <if(friends)>
794
795 private:
796     <friends; separator="\n"><endif>
797 >>
798
799 /**
800  * Template fuer HeapObjekt-Typspezifizierer
801  *
802  * @param className Name der Klasse und somit auch des Typens
803  * @param nameSpace Namensraum der Klasse
804  */
805 heapObjQualifiedTypeTemplate(classname, nameSpace) ::= <<
806 <if(nameSpace)><nameSpace>::<endif><classname>::Ptr
807 >>
808
809 /**
810  * Template fuer StringObjekt-Typspezifizierer
811  *
812  * @param className Name der Klasse und somit auch des Typens
813  * @param nameSpace Namensraum der Klasse
814  */
815 stringObjQualifiedTypeTemplate(classname, nameSpace) ::= <<
816 <if(nameSpace)><nameSpace>::<endif><classname>
817 >>
818
819 /**
820  * Template fuer ExceptionObjekt-Typspezifizierer
821  *
822  * @param className Name der Klasse und somit auch des Typens
823  * @param nameSpace Namensraum der Klasse
824  */
825 exceptionObjQualifiedTypeTemplate(classname, nameSpace) ::= <<
826 <if(nameSpace)><nameSpace>::<endif><classname>
827 >>
828
829 /**
830  * Template fuer EnumObjekt-Typspezifizierer
831  *
832  * @param className Name der Klasse und somit auch des Typens
833  * @param nameSpace Namensraum der Klasse
834  */
835 enumObjQualifiedTypeTemplate(classname, nameSpace) ::= <<
836 <if(nameSpace)><nameSpace>::<endif><classname>
837 >>
838
```

```
839 /**
840  * Template fuer ValueObjekt-Typspezifizierer
841  *
842  * @param className Name der Klasse und somit auch des Typens
843  * @param nameSpace Namensraum der Klasse
844  */
845 valueObjQualifiedTypeTemplate(classname, nameSpace) ::= <<
846 <if(nameSpace)><nameSpace>::<endif><classname>
847 >>
848
849 /**
850  * Template fuer InterfaceObjekt-Typspezifizierer
851  *
852  * @param className Name der Klasse und somit auch des Typens
853  * @param nameSpace Namensraum der Klasse
854  */
855 interfaceObjQualifiedTypeTemplate(classname, nameSpace) ::= <<
856 <if(nameSpace)><nameSpace>::<endif><classname>*
857 >>
858
859 /**
860  * Template fuer die Deklaration eines Arrays vom Typ <code>type</code>
861  *
862  * @param type der qualifizierte Typ
863  */
864 arrayTypeTemplate(type) ::= <<
865 HeapArray\<<type>\>::Ptr
866 >>
867
868 /**
869  * Template fuer die Deklaration einer Matrix vom Typ <code>type</code>
870  *
871  * @param type der qualifizierte Typ
872  */
873 matrixTypeTemplate(type) ::= <<
874 HeapMatrix\<<type>\>::Ptr
875 >>
876
877 /**
878  * Template, das eine Ueberpruefung eines HeapObject's auf <code>null</
        code> in C++ realisiert
879  *
880  * @param caller der "Aufrufer" der is isValid-Funktion
881  */
882 heapObjIsValidFunTemplate(caller) ::= <<
883 <caller>.isValid()
884 >>
885
886 stringObjIsValidFunTemplate(caller) ::= <<
887 <caller>.isValid()
888 >>
889
```

```
890 exceptionObjIsValidFunTemplate(caller) ::= <<
891 <caller>.isValid()
892 >>
893
894 enumObjIsValidFunTemplate(caller, enumName, enumNameSpace) ::= <<
895 <caller> != <if(enumNameSpace)><enumNameSpace>::<endif><enumName>::
        ENUM_NULL
896 >>
897
898 /**
899  * Template, das eine Ueberpruefung eines HeapObject's auf "nicht" <code>
        null</code> in C++ realisiert
900  *
901  * @param templateName der Name des "isValid"-Templates, das aufgerufen
        werden soll
902  * @param caller der "Aufrufer" der is isInvalid-Funktion
903  */
904 isInvalidFunTemplate(templateName,caller) ::= <<
905 !<(templateName)(caller=caller)>
906 >>
907
908 enumObjIsInvalidFunTemplate(caller, enumName, enumNameSpace) ::= <<
909 <caller> == <if(enumNameSpace)><enumNameSpace>::<endif><enumName>::
        ENUM_NULL
910 >>
911
912 /**
913  * Template, das eine Ueberpruefung eines HeapObject'S auf <code>null</
        code> in C++ realisiert
914  *
915  * @param caller der "Aufrufer" der is invalidate-Funktion
916  */
917 heapObjInvalidateTemplate(caller) ::= <<
918 <caller>.invalidate()
919 >>
920
921 stringObjInvalidateTemplate(caller) ::= <<
922 <caller> = null;
923 >>
924
925 exceptionObjInvalidateTemplate(caller) ::= <<
926 <caller> = null;
927 >>
928
929 enumObjInvalidateTemplate(caller, enumName, enumNameSpace) ::= <<
930 <caller> = <if(enumNameSpace)><enumNameSpace>::<endif><enumName>::
        ENUM_NULL
931 >>
932
933 /**
934  * Template fuer die Ausgabe einer Liste von Strings/StringTeamplates
935  *
```

```
936   * @param list die auszugebende Liste
937   * @param sep  das Trennzeichen / die Trennzeichenfolge
938   */
939  listTemplate(list, sep) ::= <<
940  <list; separator=sep>
941  >>
942
943  // TEMPLATES FUER STATEMENTS
944
945  /**
946   * Template fuer continue-Anweisung
947   *
948   * @param labelName Name des Labels an dem weitergearbeitet werden soll
949   */
950  continueStatementTemplate(labelName) ::= <<
951  <if(labelName)>// TODO: continue mit Label als Ziel<else>continue<endif>
952  >>
953
954  /**
955   * Template fuer Statements, die mit einem Label versehen wurden
956   *
957   * @param labelName Name des Labels
958   * @param stmtAfterLabel Statement, das auf das Label folgt
959   */
960   // TODO: Spezialfall continue-Label mit Ruecksprungadresse
961  labeledStatementTemplate(labelName, stmtAfterLabel) ::= <<
962
963  <labelName>:
964  <stmtAfterLabel>
965  >>
966
967  /**
968   * Template fuer eine for-Schleife in C++
969   *
970   * @param forInit   Initialisierungsanweisung
971   * @param forCond   Abbruchbedingung
972   * @param forUpdate Aktualisierung, die nach jedem Schleifendurchlauf
        getaetigt wird
973   * @param forBody   Schleifenrumpf
974   */
975  forLoopTemplate(forInit, forCond, forUpdate, forBody) ::= <<
976  for (<forInit>; <forCond>; <forUpdate>)
977  <forBody>
978
979  >>
980
981
982  /**
983   * Template fuer eine while-Schleife in C++
984   *
985   * @param whileCond Abbruchbedingung
986   * @param whileBody Schleifenrumpf
```

```
 987   */
 988  whileLoopTemplate(whileCond, whileBody) ::= <<
 989  while <whileCond>
 990  <whileBody>
 991
 992  >>
 993
 994
 995  /**
 996   * Template fuer eine do-while-Schleife in C++
 997   *
 998   * @param doBody     Schleifenrumpf
 999   * @param whileCond Abbruchbedingung
1000   */
1001  doWhileLoopTemplate(doBody, whileCond) ::= <<
1002  do
1003  <doBody>
1004  while (<whileCond>);
1005  >>
1006
1007
1008  /**
1009   * Template fuer If-Anweisungen in C++
1010   *
1011   * @param ifCond     Bedingung, damit <code>ifBody</code> ausgefuehrt wird
1012   * @param ifBody     Block, der ausgefuehrt wird, wenn <code>ifCond</code>
          zu <code>true</code> ausgewertet wird
1013   * @param elseBody   Block, der ausgefuehrt wird, wenn <code>ifCond</code>
          zu <code>false</code> ausgewertet wird
1014   */
1015  ifConditionTemplate(ifCond, ifBody, elseBody, isNotElseif)::= <<
1016  if <ifCond>
1017  <ifBody>
1018  <if(elseBody)>else <if(isNotElseif)>
1019
1020  <endif><elseBody><endif>
1021  >>
1022
1023
1024  /**
1025   * Template fuer einzelne Block-Anweisungen in C++
1026   *
1027   * @param blockBody Anweisungen, die zum Blockrumpf gehoeren
1028   */
1029  blockTemplate(blockBody) ::= <<
1030  {
1031      <blockBody>}
1032  >>
1033
1034
1035  /**
1036   * Template fuer Return-Statements in C++
```

```
1037   *
1038   * @param returnValue Rueckgabewert der Funktion
1039   */
1040  returnStmTemplate(returnValue) ::= <<
1041  return<if(returnValue)> <returnValue><endif>
1042  >>
1043
1044
1045  /** Template fuer Throw-Statements in C++
1046   *
1047   * @param thrownException zu werfende Exception
1048   */
1049  throwStmTemplate(thrownException) ::= <<
1050  throw <thrownException>
1051  >>
1052
1053
1054  /**
1055   * Template fuer Break-Anweisungen in C++
1056   *
1057   * @param destLabel Ziel an das nach dem break gesprungen werden soll
1058   */
1059   // TODO: break mit Rueck sprung-Label
1060  breakStmTemplate(destLabel) ::= <<
1061  break<if(destLabel)>// TODO: break mit Rueck sprung-Label <endif>
1062  >>
1063
1064  /**
1065   * Uebergeordnetes Template fuer Statements
1066   *
1067   * @param stmt      das Statement
1068   * @param printSemi Flag, ob ein abschliessendes Semikolon ausgegeben
          werden soll
1069   */
1070  statementTemplate(stmt, printSemi) ::= <<
1071  <if(printSemi)><stmt>;
1072
1073  <else><stmt>
1074
1075  <endif>
1076  >>
1077
1078  /**
1079   * Template fuer einen Try-Catch-Block
1080   *
1081   * @param tryBlock     Try-Block
1082   * @param catchClauses die catch-Liste
1083   */
1084  tryCatchStatement(tryBlock, catchClauses) ::= <<
1085  try
1086  <tryBlock>
1087  <catchClauses>
```

```
1088 >>
1089
1090 /**
1091  * Template fuer einen catch-Clause eines try-catch-Blockes
1092  *
1093  * @param catchArguments die Exception die "gecatched" wird
1094  * @param catchBlock     der auf <code>catchArguments</code> folgende
        Block
1095  */
1096 catchClauseTemplate(catchArguments, catchBlock) ::= <<
1097 catch(<catchArguments>)
1098 <catchBlock>
1099
1100 >>
1101
1102 // TODO: Templates fuer switch-case ueberarbeiten
1103 /**
1104  * Template fuer das switch-Statement
1105  *
1106  * @param switchValue der Wert ueber den "geswitched" wird
1107  * @param switchBlock der Block mit den verschiedenen Faellen
1108  */
1109 switchCaseTemplate(switchValue, switchBlock) ::= <<
1110 switch<switchValue>
1111 {
1112 <switchBlock>}
1113 >>
1114
1115 /**
1116  * Template fuer einen Fall eines Switch-Case-Statements
1117  *
1118  * @param caseQualifier das Label des Falls
1119  * @param caseBlock     der anschliessende optionale Block
1120  */
1121 caseTemplate(caseQualifier, caseBlock) ::= <<
1122 case <caseQualifier>:
1123 <caseBlock; separator="\n">
1124 >>
1125
1126 /**
1127  * Template fuer den Standardfall eines Switch-Case-Statements
1128  *
1129  * @param caseBlock der anschliessende optionale Block
1130  */
1131 defaultCaseTemplate(caseBlock) ::= <<
1132 default:
1133 <caseBlock>
1134 >>
1135
1136
1137
1138
```

```
1139 /**
1140  * Template fuer eine InstanceOf-Anweisung in C++
1141  * Symbolisiert die Ueberpruefung, ob das Objekt <code>obj</code> vom Typ
       <code>type</code> ist
1142  * (dieses Template darf nur dann benutzt werden, wenn die Klassen
       virtuell sind)
1143  *
1144  * @param obj gegen den Typ <code>type</code> zu pruefendes Objekt
1145  * @param type Typ gegen den <code>obj</code> geprueft wird
1146  *
1147  * Beispiel fuer den Fall, dass isHeapObj == true
1148  *
       ------------------------------------------------------------------------

1149  * Db2StartupEvent::Ptr db2StartedEvent = Db2StartupEvent::Ptr::
       castDynamic(event);
1150  *            if (db2StartedEvent.isValid())
1151  *
1152  *################################################
1153  *
1154  * Beispiel fuer den Fall, dass isHeapObj == false
1155  * ----------------------------------------------
1156  * if(Class1* c1 = dynamic_cast<Class1*>(base))
1157  * // it's a Class1*
1158  * else if (Class2* c2 = dynamic_cast<Class2*>(base))
1159  * // it's a Class2*
1160  */
1161 heapObjInstanceOfTemplate(obj, type) ::= <<
1162 <type>::castDynamic(<obj>).isValid()
1163 >>
1164
1165 /**
1166  * Instanceof-Template fuer StringObjects
1167  *
1168  * @param obj gegen den Typ <code>type</code> zu pruefendes Objekt
1169  * @param type Typ gegen den <code>obj</code> geprueft wird
1170  */
1171 stringObjInstanceOfTemplate(obj, type) ::= <<
1172 dynamic_cast\<<type>*\>(&<obj>) != NULL
1173 >>
1174
1175 /**
1176  * Instanceof-Template fuer EnumObjects
1177  *
1178  * @param obj gegen den Typ <code>type</code> zu pruefendes Objekt
1179  * @param type Typ gegen den <code>obj</code> geprueft wird
1180  */
1181 enumObjInstanceOfTemplate(obj, type) ::= <<
1182 dynamic_cast\<<type>*\>(&<obj>) != NULL
1183 >>
1184
1185 /**
```

```
1186   * Instanceof-Template fuer ExceptionObjects
1187   *
1188   * @param obj gegen den Typ <code>type</code> zu pruefendes Objekt
1189   * @param type Typ gegen den <code>obj</code> geprueft wird
1190   */
1191  exceptionObjInstanceOfTemplate(obj, type) ::= <<
1192  dynamic_cast\<<type>*\>(&<obj>) != NULL
1193  >>
1194
1195  /**
1196   * Instanceof-Template fuer ValueObjects
1197   *
1198   * @param obj gegen den Typ <code>type</code> zu pruefendes Objekt
1199   * @param type Typ gegen den <code>obj</code> geprueft wird
1200   */
1201  valueObjInstanceOfTemplate(obj, type) ::= <<
1202  dynamic_cast\<<type>*\>(&<obj>) != NULL
1203  >>
1204
1205  /**
1206   * Cast-Template fuer HeapObjects
1207   *
1208   * @param objToCast    Objekt, das auf <code>typeToCastTo</code> gecastet
        werden soll
1209   * @param typeToCastTo Typ auf den <code>objToCast</code> gecastet werden
        soll
1210   */
1211  heapObjCastTemplate(typeToCastTo, objToCast) ::= <<
1212  <typeToCastTo>::castDynamic(<objToCast>)
1213  >>
1214
1215  /**
1216   * Cast-Template fuer StringObjects
1217   *
1218   * @param objToCast    Objekt, das auf <code>typeToCastTo</code> gecastet
        werden soll
1219   * @param typeToCastTo Typ auf den <code>objToCast</code> gecastet werden
        soll
1220   */
1221  stringObjCastTemplate(typeToCastTo, objToCast) ::= <<
1222  *(dynamic_cast\<<typeToCastTo>*\>(&<objToCast>)
1223  >>
1224
1225  /**
1226   * Cast-Template fuer EnumObjects
1227   *
1228   * @param objToCast    Objekt, das auf <code>typeToCastTo</code> gecastet
        werden soll
1229   * @param typeToCastTo Typ auf den <code>objToCast</code> gecastet werden
        soll
1230   */
1231  enumObjCastTemplate(typeToCastTo, objToCast) ::= <<
```

```
1232 static_cast\<<typeToCastTo>\>(<objToCast>)
1233 >>
1234
1235 /**
1236  * Cast-Template fuer ExceptionObjects
1237  *
1238  * @param objToCast    Objekt, das auf <code>typeToCastTo</code> gecastet
         werden soll
1239  * @param typeToCastTo Typ auf den <code>objToCast</code> gecastet werden
         soll
1240  */
1241 exceptionObjCastTemplate(typeToCastTo, objToCast) ::= <<
1242 *(dynamic_cast\<<typeToCastTo>*\>(&<objToCast>))
1243 >>
1244
1245 /**
1246  * Cast-Template fuer ValueObjects
1247  *
1248  * @param objToCast    Objekt, das auf <code>typeToCastTo</code> gecastet
         werden soll
1249  * @param typeToCastTo Typ auf den <code>objToCast</code> gecastet werden
         soll
1250  */
1251 valueObjCastTemplate(typeToCastTo, objToCast) ::= <<
1252 static_cast\<<typeToCastTo>\>(<objToCast>)
1253 >>
1254
1255 /**
1256  * Template, das auf <code>leftSide</code> <code>rightSide</code> anwendet
1257  *
1258  * @param leftSide       linke Seite
1259  * @param rightSide      rechte Seite
1260  * @param dereferSymbol wie <code>rightSide</code auf <code>leftSide</code
         > angewendet wird
1261  */
1262 dereferencingTemplate(leftSide, rightSide, dereferSymbol) ::= <<
1263 <leftSide><dereferSymbol><rightSide>
1264 >>
1265
1266 /**
1267  * Template fuer einen primitiven Datentypen
1268  *
1269  * @param primitiveType primitiver Datentyp, wird mittels <code>
         primitiveTypeMap</code> gemappt
1270  */
1271 primitiveTypeTemplate(primitiveType) ::= <<
1272 <primitiveTypeMap.(primitiveType)>
1273 >>
1274
1275 /**
1276  * Template, dass einfach nur ein Semikolon ausgibt
1277  */
```

```
1278 semiTemplate() ::= <<
1279 ;
1280 >>
1281
1282 arrayInitializerFunctionsListImplTemplate(arrayFunList) ::= <<
1283 <arrayFunList; separator="\n\n">
1284 >>
1285
1286 arrayInitializerFunctionsListHeaderTemplate(arrayFunList, privateModifier)
         ::= <<
1287 <if(privateModifier)>private:<endif>
1288     /*
1289      *
1290      * GENERIERTE FUNKTIONEN FUER INITIALISIERUNGEN VON ARRAYS
1291      *
1292      */
1293     <arrayFunList; separator="\n\n">
1294 >>
1295
1296 arrayInitializerFunHeader(static, funName, paramList, arrayType) ::= <<
1297 <if(static)><static> <endif><arrayType> <funName>(<paramList>);
1298 >>
1299
1300 arrayInitializerFunImpl(className, funName, paramList, arrayType, funBody)
         ::= <<
1301 <arrayType> <className>::<funName>(<paramList>)
1302 {
1303     <funBody>
1304 }
1305 >>
1306
1307 newArrayConstructionTemplate(arrayTypeName, arrayClassName,
       arrayClassNameSpace, arrayDimSizeList) ::= <<
1308 <if(arrayClassNameSpace)><arrayTypeName>::<endif><arrayClassName>\<<
       arrayTypeName>\>::create(<arrayDimSizeList>)
1309 >>
1310
1311 newMatrixConstructionTemplate(matrixTypeName, matrixClassname,
       matrixClassNameSpace, matrixDimSizeList) ::= <<
1312 <if(matrixClassNameSpace)><matrixClassNameSpace>::<endif><matrixClassname
       >\<<matrixTypeName>\>::create(<matrixDimSizeList>)
1313 >>
1314
1315
1316 newArrayConstructionInGenFunTemplate(arrayTypeName,
1317                                      arrayClassName,
1318                                      arrayClassNameSpace,
1319                                      arrayDimSizeList,
1320                                      tempVariableName,
1321                                      appendList)
1322 ::= <<
1323 <if(arrayClassNameSpace)><arrayTypeName>::<endif><arrayClassName>\<<
```

```
        arrayTypeName>\>::Ptr <tempVariableName> = <if(arrayClassNameSpace)><
        arrayTypeName>::<endif><arrayClassName>\<<arrayTypeName>\>::create(<
        arrayDimSizeList>);
1324
1325  <appendList:arrayAppendCallTemplate(varName=tempVariableName)>
1326  return <tempVariableName>;
1327  >>
1328
1329  newMatrixConstructionInGenFunTemplate(matrixTypeName,
1330                                        matrixClassname,
1331                                        matrixClassNameSpace,
1332                                        matrixDimSizeList,
1333                                        tempVariableName,
1334                                        appendList)
1335  ::= <<
1336  <if(matrixClassNameSpace)><matrixClassNameSpace>::<endif><matrixClassname
        >\<<matrixTypeName>\>::Ptr <tempVariableName> = <if(
        matrixClassNameSpace)><matrixClassNameSpace>::<endif><matrixClassname
        >\<<matrixTypeName>\>::create(<matrixDimSizeList>);
1337
1338  <appendList:arrayAppendCallTemplate(varName=tempVariableName)>
1339  return <tempVariableName>;
1340  >>
1341
1342  matrixAppendTemplate(dim, initialValue) ::= <<
1343  append(<dim>, <initialValue>)
1344  >>
1345
1346  arrayAppendTemplate(initialValue) ::= <<
1347  append(<initialValue>)
1348  >>
1349
1350  arrayAppendCallTemplate(varName) ::= <<
1351  <varName>-\><it>;
1352
1353  >>
1354
1355  escapedBackSlashTemplate() ::= <<
1356  '\\\\'
1357  >>
```