S T U D I E N A R B E I T

Berufsakademie Stuttgart - Außenstelle Horb -
Staatliche Studienakademie

Fachrichtung Informationstechnik

Migration einer Codegenerierungskomponente für einen Java-Compiler

Juni 2006

Eingereicht von:

Jürgen Schmiing
Lerchenstrasse 2
72160 Horb am Neckar

Firma:

GFT Technologies AG
Leopoldstrasse 1
78112 St. Georgen im Schwarzwald

Betreuer:
Prof. Dr. Martin Plümicke

Erklärung:

Ich habe die Studienarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt.

Ort	Datum	Unterschrift

Zusammenfassung

Das Projekt „Typinferenz in Java“ beschäftigt sich mit der Realisierung einer Programmiersprache auf Java-Syntax. Im Gegensatz zu anderen Hochsprachen wie C++, Pascal oder Java selbst ist es jedoch nicht zwingend erforderlich, den Typ einer Variablen, einer Methode oder Methodenparameter festzulegen. Durch die benutzten Methoden und Eigenschaften eines Objekts wird der Typ berechnet.

Die vorliegende Studienarbeit erweitert die bereits implementierte Lösung zur Generierung von Bytecode, die von einer Java Virtual Machine ausgeführt werden kann. Dabei wird diese Lösung nicht nur stabiler und fehlertoleranter gestaltet, sondern neue Funktionen hinzugefügt. Zudem werden Generics, eine neue Funktionalität von Java 1.5 integriert.

Abstract

The project „Typeinference in Java“ specifies and implements a programming language which is very similar to Java. In comparison to other high level programming languages like C++, Pascal or Java itself, the software-engineer is not obliged to define types of elements, including method parameters or return types. The appropriate type of an element is determined by the methods and properties used in the source-code.

This student research project enhances the former solution for creating bytecode, which can be interpreted and executed by an implementation of the java virtual machine. Mainly, this project integrates the support of generics, a new feature of Java 1.5. Additionally, the attention is drawn on making the codegeneration more fault-tolerant.

Inhaltsverzeichnis

Tabellenverzeichnis

Listings

Abkürzungen

ANT	Another Neat Tool. Tool zur Unterstützung des Build-Vorgangs.
CVS	Concurrent Versions Control. Versionsmanagementsystem.
IDE	Integrated Development Environment. Entwicklungsumgebung.
JDK	Java Development Kit. Wird zum Erstellen von Java-Awendungen benötigt.
JVM	Java Virtual Machine. Zuständig für das Ausführen von Java-Bytecode.
Log4J	Logging-Paket. Dient zur einfachen Ausgabe und Kontrolle von Logging-Ausgaben.
UTF-8	Unicode Transformation Format. Eine Kodierung für Unicode-Zeichen.
VM	Virtual Machine. Dient zur Abstraktion von der Maschine (Hardware).
XML	Extended Markup Language. Definierter Standard zum Ablegen von Informationen.

Kapitel 1

Einleitung

1.1 Motivation

Um sich ein Bild über das Ziel des Projekts Typinferenz in Java machen zu können, ist es zunächst unerlässlich, sich einen kurzen Überblick über Programmiersprachen zu machen. Aktuelle Hochsprachen, in diesem Beispiel seien einmal C++, Java, Pascal/Delphi als Vergleich herangezogen, erfordern bei der Definition einer Variablen gleichzeitig eine Typdefinition. Diese ist unumgänglich; mit ihr wird ebenfalls abgeprüft, ob bestimmte Operationen auf Datentypen überhaupt zulässig sind. Als Beispiel: Die Addition kann bei Integern oder Strings (Konkatenation) sicherlich sinnvoll sein, aber eine Addition von zwei Objekten ganz pauschal nicht. Die Programmierumgebung Visual Basic bietet darüber hinaus an, Variablen als Variant zu deklarieren. Hierbei ist der Programmierer nicht gezwungen, zum Entwicklungszeitpunkt den Typ einer Variablen festzulegen. Dies hat jedoch zur Folge, dass die Typ- und Semantikchecks erst zur Laufzeit ausgeführt werden, was unter Umständen zu Laufzeitfehlern führt. Ziel des Projekts Typinferenz in Java ist es, ebenfalls keine Definitionen des Variablentyps zu verlangen. Im Gegensatz zu Visual Basic wird die Typprüfung jedoch nach wie vor zur Compilezeit durchgeführt - anhand der durchgeführten Operationen auf der Variablen wird ermittelt, welche Typen für die Variable in Frage kommen. Diese schließt zu aktuellem Zeitpunkt eine Fest definierte Menge an Objekten ein.

1.2 Geschichte des Projekts

Das Projekt selbst entstand im Jahr 2000 im Rahmen der Vorlesungen Informatik und Software-Engineering. Aufgabe war es zunächst, mit Hilfe der Parsertools yacc und yylex eine Grammatik für die Sprache Java zu entwerfen. Im folgenden Semester sollte dann ein Klassenentwurf für den Compiler folgen, der auch implementiert wurde. Es folgten weitere Studienarbeiten, die die Erweiterung und Implementierung neuer Features zum Thema hatten. Dazu zählen unter anderen Felix Reichenbach, Jörg Bäuerle, Thomas Ott und Markus Haas (dessen Studienarbeit sich als direkte Vorgängerarbeit bezeichnen lässt). Sicherlich ist auch die Arbeit des betreuenden Dozenten Prof. Dr. Martin Plümicke an dieser Stelle zu erwähnen, der ebenfalls tatkräftig bei der Entwicklung des Projekts mitarbeitet.

1.3 Aufgabenstellung

Die Aufgabenstellung, die dieser Studienarbeit zugrunde liegt, ist die Erweiterung des Java Typinferenz-Projekts. Dabei soll vorrangig die Bytecodegenerierung fehlertoleranter und sicherer gemacht werden. Weiterhin sollen die neuen Features aus Java 1.5, dazu zählen im Wesentlichen die Generics, in den Compiler integriert werden. Dazu ist es erforderlich, den Bytecode selbst Java 1.5 Konform zu generieren.
Als weiteres Feature soll die Codegenerierung das Generieren von Bytecode für Interfaces unterstützen. Dies bedeutetet speziell für den Parsevorgang eine Einschränkung der gültigen Elemente und gleichzeitig die Erweiterung um einige neue Schlüsselworte.
In Absprache mit Timo Holzherr soll die Funktionalität erweitert werden, indem die Angabe von Packages und auch die Möglichkeit von Importen hinzugefügt wird. Dies ist vor allem auch mit neuen Anforderungen an den Typinferenzalgorithmus verbunden.
Zum Schluss soll es möglich sein, die Usecases sowohl für die Bytecodegenerierung, als auch von Timo Holzherr und Thomas Hornberger aufgestellte Funktionstests, erfolgreich in ausführbaren Bytecode zu überführen.

1.4 Projektplanung

Die verfügbare Zeit zur Durchführung des Projekts erstreckte sich von Oktober 2005 bis Juni 2006, also etwa neun Monate. Dies entspricht zwei Theoriephasen und einer Praxisphase. Im Folgenden sollen die Tätigkeiten über die Dauer des Projekts kurz dargelegt werden:

5. Semester Theoriephase

- Einarbeitung in Projekt und Aufgabenstellung
- Planung des Projektverlaufs
- Portieren des Loggings auf Log4J

5. Semester Praxisphase

- Bearbeitung des Compiler-Quellcodes, Umstellung auf Java 1.5
- Einarbeitung in die Bytecode-Spezifikation
- Beginn der Dokumentation

6. Semester Theoriephase

- Umstellung des Compilers auf Java 1.5
- Entwurf und Realisierung von JUnit-Tests
- Integration von Interfaces und Packages
- Erweiterung auf Verarbeitung und Generieren von Generics
- Fertigstellen der Dokumentation

Kapitel 2

Grundlagen

2.1 Die Struktur einer Class-Datei

Für die Arbeit an der Bytecodegenerierung des Projekts ist es unerlässlich, mit der Struktur einer Class-Datei vertraut zu sein. Das Classfile repräsentiert dabei eine Klasse oder ein Interface in einer stark verkürzten Form, dem Bytecode. Es handelt sich nicht um Maschinencode - dieser Bytecode ist plattformunabhängig und erlaubt ein Ausführen auf jeder Maschinenarchitektur, für die eine Implementation der Java Virtual Maschine (kurz: JVM) verfügbar ist.
Die kleinste Informationseinheit im Bytecode ist ein Byte. Ist zur Darstellung einer Information mehr Speicherplatz erforderlich, so können die Daten per Konvention auch als 2- oder 4- Byte Daten abgelegt werden. In Tabelle 2.1 findet sich die Strukturbeschreibung einer Class-Datei. Bei allen Feldgrößen, die mit variabel angegeben sind, handelt es sich wiederum um eigene und unter Umständen verschiedenen Strukturen, die hinterlegt sein können. Im folgenden Kapitel wird kurz auf die Bedeutung der Elemente eingegangen.

Element	**Länge**
magic	4 Byte
minor-version	2 Byte
major-version	2 Byte
constant-pool-count	2 Byte
constant-pool	variabel
access-flags	2 Byte
this-class	2 Byte
super-class	2 Byte
interfaces-count	2 Byte
interfaces	variabel
fields-count	2 Byte
fields	variabel
method-count	2 Byte
methods	variabel
attributes-count	2 Byte
attributes	variabel

Tabelle 2.1: Struktur einer Class-Datei

2.2 Elemente einer Class-Datei

magic Das Magic-Tag markiert den Anfang einer Class-Datei und besitzt immer den Wert `0xCAFEBABE`.

minor-version Gibt den Nachkommaanteil der Version an.

major-version Gibt den Vorkommaanteil der Version an. Hierdurch können Implementierungen der JVM entscheiden, ob sie den Bytecode ausführen können.

constant-pool-count Der Eintrag constant-pool-count beinhaltet die Anzahl der Konstantenpooleinträge + 1.

constant-pool Der Konstantenpool ermöglicht die Ablage von Konstanteninformationen in verschiedenen Strukturen. Sämtliche Informationen über Variablendefinitionen, Methodendefinitionen, usw. werden hier abgelegt. Im Bytecode der Klasse wird dann nur noch über Verweise auf den entsprechenden Eintrag im Konstantenpool gezeigt. Die Nummerierung der Elemente beginnt bei 1 und endet bei constant-pool-count. Eine Typidentifikation des vorliegenden Elements erfolgt über das Tag-Byte, dass den Beginn einer Konstantendefinition darstellt. Abhängig vom aufgetretenen Tag werden danach die Informationen zur jeweiligen Konstante in unterschiedlicher Weise kodiert. Die Tabelle 2.2 listet die verschiedenen Elemente des Konstantenpools auf.

Tag	Bedeutung
`0x01`	**UTF-8 Constant**. Die Konstante enthält einen Unicode String mit angegebener Länge. Dabei geben die 2 folgenden Bytes die Länge an, darauf folgt direkt der Inhalt der Zeichenfolge.
`0x03`	**Integer**. Die 4 Byte der Konstante enthalten einen 64-Bit Integer-Wert.
`0x04`	**Float**. Die 4 Byte der Konstante enthalten einen 64-Bit Float-Wert.
`0x05`	**Long**. Die 8 Byte der Konstante enthalten einen 128-Bit Long-Wert.
`0x06`	**Double**. Die 8 Byte der Konstante enthalten einen 128-Bit Double-Wert.
`0x07`	**ClassInfo**. Der Konstantenpooleintrag Klasse verweist auf eine UTF-8 Konstante im Konstantenpool, die den Namen der Klasse oder des Interface beinhaltet. Ist sie Bestandteil einer Package, so steht diese ebenfalls in der angegebenen UTF-8 Konstante.
`0x08`	**String**. Der Eintrag String verweist auf eine UTF-8 Konstante, die den Wert der Stringkonstante enthält.
`0x09`	**Fieldref**. Der Typ Feldreferenz stellt den Verweis auf ein Attribut einer Klasse dar. Dazu beinhaltet die Konstante einen Verweis auf ein Element vom Typ Class und ein Element vom Typ NameAndType im Konstantenpool.
`0x0A`	**Methodref**. Der Typ Methodenreferenz stellt den Verweis auf eine Methode einer Klasse dar. Dazu wird ein Verweis auf ein Element vom Typ Class und ein Element vom Typ NameAndType verwendet.
`0x0B`	**InterfaceMethodref**. Der Typ InferfaceMethodref gleicht dem Typ Methodref mit dem Unterschied, dass es sich nicht um eine Klassen-Methode, sondern um eine Interface-Methode handelt.
`0x0C`	**NameAndType**. Die Konstante vom Typ NameAndType verweist auf zwei Elemente im Konstantenpool: Den Namen des Elements sowie des Typs bzw. der Typen bei einer Parameterliste.

Tabelle 2.2: Elemente im Konstantenpool

access-flags Der Eintrag Access-Flags gibt die Access-Modifier der Klasse an. Eine Auflistung der möglichen Access-Modifier und der zugehörigen Werte ist der Tabelle

2.6 im Kapitel Access-Modifier zu entnehmen.

this-class Zeigt auf einen Konstantenpooleintrag vom Typ ClassInfo, der auf den Namen der Klasse verweist. Sofern die Klasse eine Packagedefinition hat, ist diese dort ebenfalls zu finden.

super-class Der Eintrag Superclass zeigt auf einen ClassInfo Eintrag im Konstantenpool, der wiederrum auf die Superklasse der Klasse verweist. Für den Fall, dass die Superklasse OBJECT ist, ist der Wert 0 in diesem Feld zulässig.

interfaces-count Gibt die Anzahl der Interfaces an, die von der Klasse implementiert, oder die von dem Interface erweitert werden.

interfaces Interfaces enthält für jedes Interface einen Verweis auf einen ClassInfo Eintrag im Konstantenpool, der das Package sowie den Namen der Superinterfaces angibt.

fields-count Gibt die Anzahl der Fields (Attribute) der Klasse an.

fields Die Fields-Auflistung enthält Informationen zu jedem Attribut, dass in der Klasse definiert wurde. Dazu zählen im Wesentlichen die Access-Flags, der Name sowie der Typ des Attributs. Dabei sind die letzten beiden Eigenschaften Verweise auf einen UTF-8 Eintrag im Konstantenpool. Die Tabelle 2.3 gibt Aufschluss über die Struktur eines Fields.

Bezeichnung	**Länge**
Access-Flags	2 Byte
Name-Index	2 Byte
Descriptor-Index	2 Byte
Attributes-count	2 Byte
Attributes	*variabel*

Tabelle 2.3: Struktur eines Fields

method-count Gibt die Anzahl der definierten Methoden in der Klasse an.

methods Die Methods-Auflistung beinhaltet alle in der Klasse oder dem Interface definierten Methoden. Ähnlich wie auch die Fields-Struktur beinhaltet die Struktur zunächst einmal Access-Modifier, Namen und Parameterkonventionen. Zusätzliche Eigenschaften der Methode sind in der Attributes-Auflistung zu finden; hier wird der Bytecode der Methode abgelegt. Die Tabelle 2.4 zeigt die Struktur einer Methodendefinition.

Bezeichnung	**Länge**
Access-Flags	2 Byte
Name-Index	2 Byte
Descriptor-Index	2 Byte
Attributes-count	2 Byte
Attributes	*variabel*

Tabelle 2.4: Struktur einer Method

attribute-count Gibt die Anzahl der definierten Attribute in der Klasse an.

attribute Die Attributes-Auflistung enthält zusätzliche Informationen über die Klasse. Momentan sind nur die beiden Attribute SourceFile und Deprecated für den Typ Klasse bzw. Interface offiziell festgelegt. Hier ist es möglich, Informationen für eigene Implementationen hinzuzufügen. Die JVMs ignorieren im Regelfall diese Informationen, sofern das Attribut den in Tabelle 2.5 gezeigten Konventionen entspricht.

Grundsätzlich liegen die Attribut-Namen als UTF-8 Konstante im Konstantenpool der Klasse vor, und werden mittels des Index referenziert.

Bezeichnung	Länge
Attribute-Name-Index	2 Byte
Attribute-Length	4 Byte

Tabelle 2.5: Struktur eines Attributs

2.3 Access-Modifier

Access-Modifier dienen dazu, Zugriffeigenschaften und Zustände von Klassen und einzelnen Elementen in Klassen zu beschreiben. Die Tabelle 2.6 listet alle Access-Modifier sowie deren Anwendungsbereiche auf.

Wert	Name	Erlaubt für	Bedeutung
0x001	Public	Alle Elemente	Zugriff von anderen Packages ist möglich.
0x002	Private	Methode, Field	Zugriff nur innerhalb der Klasse möglich.
0x004	Protected	Methode, Field	Zugriff nur innerhalb der Klasse und aller Subklassen ist möglich.
0x008	Static	Methode, Field	Objekt muss nicht instanziiert werden.
0x010	Final	Methode, Klasse	Es sind keine weiteren Subklassen erlaubt.
0x020	Synchonized	Methode	Gesonderte Betrachtung für Multi-Threaded Anwendungen.
0x040	Volatile	Field	Objekt darf nicht gecached werden.
0x080	Transient	Field	Objekt kann nicht über einen Persistenz-Manager gesichert/geladen werden.
0x100	Native	Methode	Die Methode wurde nicht in Java implementiert.
0x200	Interface	Klasse	Es handelt sich um ein Interface.
0x400	Abstract	Klasse, Methode	Die Klasse kann nicht instanziiert werden.
0x800	Strict	Methode	Der Berechnungsmodus für Fließkomma-Operationen wird auf strikt gesetzt.

Tabelle 2.6: Access-Modifier

2.4 Attribute

Die Definition von Attributen dient dazu, ein Element näher zu beschreiben oder ihm zusätzliche Eigenschaften zu geben. Attribute können an Felder, Methoden und Klassen selbst angehängt werden.
Im Folgenden sollen kurz die drei Typen Code-Attribute, ConstantValue und Signature vorgestellt werden.

2.4.1 Code-Attribute

Das Code-Attribute enthält den eigentlichen Quellcode der Klasse. Dieses Attribut wird hinter jede Methode und jeden Konstruktor gehängt, der als nicht-abstrakt definiert wurde. Tabelle 2.7 zeigt den Aufbau des Attributes.

Bezeichnung	**Länge**	**Erläuterung**
Attribute-Name-Index	2 Byte	Verweist auf eine UTF-8 Konstante im Konstantenpool mit dem Inhalt „Code“. Mehrere Codeattribute können auf den gleichen Index verweisen.
Attribute-Length	4 Byte	Gibt die noch folgende Anzahl an Bytes dieser Struktur an.
Max-Stack	2 Byte	Enthält die maximale Größe des Stacks.
Max-Locals	2 Byte	Enthält die maximale Anzahl an lokal definierten Variablen.
Code-Length	4 Byte	Gibt die Länge des Bytecodes an.
Bytecode	*variabel*	Das Feld Bytecode, dass die Länge des im zuvor angegebenen Feld code-length hat, enthält den eigentlichen Bytecode der Methode.
Exception-Table-Length	2 Byte	Gibt die Anzahl der definierten Exceptions an.
Exceptions	*variabel*	Enthält für jede Exception die Angaben zum Auftrittsbereich, Handler usw.
Attributes-Count	2 Byte	Legt die Anzahl der zusätzlich definierten Attribute innerhalb dieser Struktur fest.
Attributes	*variabel*	Enthält zusätzliche Informationen zu der CodeAttribute-Struktur. Beispielsweise könnten noch Informationen über lokale Variablen (LocalVariableTable) oder Zeilenangaben (LineNumberTable) angehängt werden.

Tabelle 2.7: Aufbau des Attributs Codeattribute

2.4.2 ConstantValue-Attribute

Das Attribut ConstantValue wird zur Realisierung von Konstantendefinitionen im Bytecode verwendet. Das Attribut wird dazu in die Attributes-Auflistung eines Fields hinzugefügt. Tabelle 2.8 skizziert den Aufbau.

2.4.3 Signature-Attribute

Das Signature-Attribut wird zur Definition der Generics in Java 1.5 definiert. Zum Stand dieser Arbeit ist keine offizielle Dokumentation dieses Attributes verfügbar. Im Kapitel „Umsetzung der Aufgabenstellung“ wird ausführlich auf die Integration von Generics eingegangen.

Bezeichnung	Länge	Erläuterung
Attribute-Name-Index	2 Byte	Verweist auf die UTF-8 Konstante im Konstantenpool, die den Wert „ConstantValue“ hat. Mehrere Konstantendefinitionen dürfen auf den gleichen Index referenzieren.
Attribute-Length	4 Byte	Gibt die Länge des Attributes an.
ConstantValue-Index	2 Byte	Der Index verweist auf das Element im Konstantenpool, dass den Wert der Konstante beinhaltet. Die Elemente können beispielsweise vom Typ IntegerInfo, LongInfo, DoubleInfo, FloatInfo oder auch String sein.

Tabelle 2.8: Aufbau des Attributs ConstantValue

2.5 Mnemorics - Darstellung von Datentypen

Bei Java wird grundsätzlich zwischen zwei unterschiedlichen Variablentypen unterschieden: Zum einen den Basistypen, als Beispiel *int, byte, char, long,* usw. Sie stellen keine Objekte dar, und werden beim Aufruf als Wert und nicht als Referenz übergeben. Als zweiten Typ existieren die Referenztypen, als Beispiel *Object, java.util.Vector,* usw. Bei ihnen handelt es sich um Objekte, bestehend aus einer Zahl definierter Eigenschaften und Methoden. Sie werden zwangsläufig als Referenz übergeben.

Wie bereits die Beschreibung der unterschiedlichen Konstantenpooleinträge zeigt, werden die Informationen zu Feldern, Methodenparametern und Rückgabetypen im Konstantenpool abgelegt. Dazu ist die Struktur *NameAndType* verantwortlich, die den Verweis auf den Namen und den Typ bzw. die Typen als auf eine UTF-8 Konstante beinhaltet. Jedoch wird in den Konstanten nicht die Bezeichnung im Sourcecode gespeichert, sondern nur eine Verkürzung (bei Basistypen) bzw. der Name in veränderter Schreibweise (bei Referenztypen) abgelegt. Die Tabelle 2.9 zeigt eine Liste der in Java zulässigen Mnemorics.

Kurzforum	Typ	Beschreibung
`B`	byte	Wert eines Bytes.
`C`	char	Unicode-Zeichen.
`D`	double	Fließkommawert, doppelte Präzision.
`F`	float	Fließkommawert, einfache Präzision.
`I`	int	Integer-Wert.
`J`	long	Langer Integer-Wert.
`L`*`Classname`*`;`	Referenz	Instanz einer Klasse.
`S`	short	Kurzer Integer-Wert.
`T`*`Name`*`;`	Generischer Typ	Verweis auf einen generischen Typ.
`V`	void	Kein Rückgabewert (für Methoden)

Tabelle 2.9: Mnemorics

Array Typen werden dadruch gekennzeichnet, dass vor dem eigentlichen Typ eine geöffnete eckige Klammer platziert wird. Je nach Anzahl der Klammern kann es sich auch um mehrdimensionale Arrays handeln. Die Array-Konventionen seien hier nur der Vollständigkeit halber erwähnt.
Die Repräsentation der Aufrufkonvention einer Methode erfolgt ähnlich der Definition im Quellcode: In den Klammern werden die einzelnen Typen der Reihe nach aufgeführt. Der Rückgabetyp wird jedoch nicht vor die Parameterdefinition, sondern hinter diese gestellt. Drei Beispiele für die Signatur einer Methode:

- `(Ljava/lang/String;)Ljava/util/Vector;`
  Die Methode erwartet als Übergabeparameter einen String und liefert einen Vector zurück.
- `(ILjava/lang/String;)Ljava/langString;`
  Die Methode erwartet als Übergabeparameter einen int-Wert und einen String, und liefert einen String zurück.
- `()V`
  Die Methode erwartet keine Übergabeparameter und liefert nichts zurück (void).

Wie durch die Beispiele deutlich wird, werden Packages nicht wie im Quellcode mit einem Punkt (`.`), sondern mit einem Slash (`/`) getrennt. Diese Konvention gilt für alle Typanganben im Bytecode, die Packages enthalten.

2.6 Features von Java 1.5 - Generics

Eine Erläuterung von allen neuen Features der Java Release 1.5 (auch 5 oder Tiger genannt) wäre an dieser Stelle zu umfangreich. Aus diesem Grund sei hier nur auf ein wesentliches Feature eingehen, dass auch im Funktionsumfang des Compilers integriert werden soll: Generics oder generische Datentypen. Generics erlauben, einfach gesagt, eine Typangabe für Containerklassen. In früheren Java Versionen war es erforderlich, entweder für jeden unterschiedlichen Typ eine eigene Klasse anzulegen, oder Typecasts zu verwenden. Beispielweise ließen sich in einen Vektor 10 Objekte vom Typ JButton einfügen, beim Zurückgeben des Objekts aus dem Vektor musste jedoch explizit der Typecast (JButton) angegeben werden. Ein weiteres Problem dabei war, dass keine Typsicherheit bei dieser Vorgehensweise gegeben war. Wurde zum Beispiel ein anderes Objekt von einem gänzlich anderen Typ (also nicht abgeleitet von JButton oder JButton selbst) in den Vektor eingefügt, so endet die Rückgabe in einer Typecast-Exception. Mit Hilfe von generischen Datentypen kann nun der Typecast umgangen und eine Typsicherheit etabliert werden. Dazu wird bereits beim Erzeugen der Typ angegeben, den die Containerklasse aufnehmen soll. Danach können zum einen nur noch Objekte dieses Typs eingefügt werden, zum anderen ist kein Typecast mehr erforderlich. Die Listings 2.1 und 2.2 zeigen den Quellcode zum einen Java 1.4 konform, zum anderen Java 1.5 konform.

```
Vector zeichenfolgen = new Vector();
zeichenfolgen.addElement("Hallo");
zeichenfolgen.addElement("Welt!");
String text1 = (String) zeichenfolgen.elementAt(1);
String text2 = (String) zeichenfolgen.elementAt(2);
System.out.println(text1 + "␣" + text2);
```

Listing 2.1: Beispiel für Java 1.4 ohne Generics

```
Vector<String> zeichenfolgen = new Vector<String>();
zeichenfolgen.addElement("Hallo");
zeichenfolgen.addElement("Welt!");
String text1 = zeichenfolgen.elementAt(1);
String text2 = zeichenfolgen.elementAt(2);
System.out.println(text1 + "␣" + text2);
```

Listing 2.2: Beispiel für Java 1.5 mit Generics

Java 1.5 bietet über dieses einfache Beispiel hinaus noch weitere Möglichkeiten, die verwendeten Typen in einer Containerklasse zu spezifizieren. Die Artikel [LaKr04] und [Brac04] bieten dazu eine ausführlichere Dokumentation.

Kapitel 3

Vorbereitungen

3.1 Übergabe des Projekts

Das Projekt wurde in Form eines CVS-Repositories[1] mit dem Namen JavaCompilerCore übergeben. Eine Anleitung zum Einrichten des Projekts in Eclipse inklusive CVS-Verbindung und Nutzung von Cygwin findet sich in [bajo05]. Das Repository selbst enthält neben dem eigentlichen Quellcode im Wesentlichen die Studienarbeiten der Vorgänger, verschiedene UML-Diagramme in Forum von Grafikdateien und die Testcases, geordnet nach Ersteller.

3.2 Ist-Zustand der Bytecodegenerierung

Unter anderem aus der Studienarbeit [hama04] geht der Umfang des Compilers hervor. Der Compiler des Java-Typinferenz Projekts schränkt den Sprachumfang von Java folgendermaßen ein:

- Packages und Imports werden nicht unterstützt.
- Es können lediglich Klassen, keine Interfaces geparst und generiert werden.
- Das Überladen von Methoden (zwei gleiche Methoden mit unterschiedlichen Rückgabewerten) ist nicht möglich.
- Es ist nicht möglich, dass innerhalb einer Klasse eine Methode und ein Attribut den gleichen Namen besitzen.
- Es ist maximal ein Konstruktor pro Klasse gestattet.
- Arraytypen sind nicht gestattet. Damit es es nicht möglich, eine Einsprungsmehtode für den Programmaufruf zu definieren
  (`public static void main(String[] args)`).

Der Aufruf des Compilers erfolgt über die Angabe des Dateinamens und optional des gewünschten Debuglevels.

[1] CVS ist die Kurzform für Concurrent Versions Control, ein Versionsmanagementsystem

3.3 Eingesetzte Werkzeuge

In den folgenden Unterkapiteln soll zunächst ein Überblick über die benutzten Tools gegeben werden.

3.3.1 Entwicklungsumgebung - Eclipse

Bei der Entwicklungsumgebung Eclipse IDE[2] handelt es sich um ein Opensource-Projekt, dass als Nachfolger von IBM Visual Age for Java 4.0 gilt. Dazu legte IBM Ende 2001 den Quellcode frei. Eclipse zählt derzeit zu den sicherlich stärksten und leistungsfähigsten Entwicklungsumgebungen für Java, die momentan am Markt sind. Auch namhafte Hersteller, die Konkurrenzprodukt dazu anbieten (beispielsweise Borland JBuilder) gehen mehr und mehr dazu über, entsprechende PlugIns für ihre Produkte auch in Eclipse verfügbar zu machen.
Die Lizenz ist so gehalten, dass die Umgebung nicht nur für private Zwecke, sondern auch für kommerzielle Projekte kostenlos eingesetzt werden darf. Eclipse bietet über die Java Umgebung hinaus weitere Plugins und Projekte, mit denen der Leistungsumfang des Tools noch weiter ausgebaut werden kann. Einige namhafte Erweiterungen sind zum Beispiel die C/C++ Umgebung CDT, die Modellierungsplugins EMF und GEF oder auch der VisualEditor (kurz VE), der die Gestaltung von Eingabeoberflächen in Java erlaubt.
Im Folgenden sollen kurz die Vorteile, gerade im Bezug auf das Projekt, erläutert werden:

- **Plattformunabhängigkeit**. Ein großer Vorteil der Entwicklungsumgebung Eclipse ist die Plattformunabhängigkeit. Da die Umgebung selbst in Java geschrieben wurde, ist sie praktisch auf allen Hardwarearchitekturen ausführbar, die eine Implementation der JVM enthalten. Dabei ist die Gestaltung der Projektdateien so gewählt, dass auch eine Zusammenarbeit mit unterschiedlichen Betriebssystemen funktioniert.

- **Unterstützung von Versionsmanagementsystemen**. Für die Arbeit an einem Projekt, dass mehrere Mitarbeiter gleichzeitig bearbeiten, ist ein Versionsmanagementsystem unerlässlich. Für das bei dem Projekt eingesetzte System CVS bietet Eclipse standardmäßig eine Integration in die Umgebung. Dies hat den Vorteil, dass nicht nur das Aktualisieren der Dateien und das Einchecken geänderter Sourcen, sondern auch ein Zugriff auf das Repository direkt in Eclipse erfolgen kann. Beispielsweise lassen sich Vergleiche mit älteren Dateien machen, die Änderungsliste verfolgen, um nur einige wenige Funktionen zu nennen.

- **Komfortabler Quellcode-Editor**. Wie auch in anderen Entwicklungsumgebungen für Hochsprachen üblich, wird nicht nur ein einfacher Texteditor geboten, sondern ein leistungsstarkes Editierwerkzeug. Wesentliche Eigenschaften sind beispielsweise die Funktion Auto-Vervollständigen oder ein Popup-Menü mit einer Liste der Funktionen, die auf dem aktuellen Objekt gültig sind. Ebenfalls werden kontextabhängige Hilfen und Tipps gegeben, die bei Auswahl selbstständig das Problem lösen. Die Funktion der Code-Templates ist auch zu beachten, die selbstständig Schleifen über Iteratoren legt, sinnvolle Try-Catch Blöcke um eine mögliche Fehlerquelle legt, oder fehlende Methoden aus Superklassen implementiert.
  Weiterhin bietet die Umgebung umfassende Suchmöglichkeiten im gesamten Projekt. Ausgehend von einer gerade ausgewählten Methode lassen sich beispielsweise Referenzen, Aufrufe oder überschriebene Methoden mit einem einfachen Mausklick lokalisieren.
  Diese Funktionen, helfen entscheidend dabei, sich auf das Programmieren zu konzentrieren, und nicht zuviel durch Randprobleme abgelenkt zu werden.

[2] IDE ist die Kurzform von Integrated Development Environment

- **Leistungsstarker Debugger**. Für das Finden von Fehlern und das Nachvollziehen von Abbrüchen im Programm ist der integrierte Debugger eine starke Hilfe für den Entwickler. Neben den Möglichkeiten, den Quellcode schrittweise auszuführen, oder gezielt Haltepunkte zu setzen, wird eine sehr detaillierte Liste der aktuell verfügbaren Variablen angezeigt. Diese ist nicht auf die primitiven Datentypen beschränkt, sondern zeigt auch die Eigenschaften von Objekten und eigenen Klassen an. Dies hilft enorm und zwingt den Entwickler nicht, die entsprechenden Informationen über einen Logger oder gar per `System.out.println()` auszugeben.

3.3.2 Apache ANT

Apache ANT[3] ist, kurz gesagt, ein Build-Werkzeug für Java. Der eigentliche Grund der Entstehung des Tools waren im Wesentlichen nicht ausreichende Fähigkeiten für die Java-Welt. ANT bietet im Gegensatz zu make eine Plattforumunabhängigkeit, eine Definition des Buildfile in XML[4] und erlaubt eigene Erweiterungen. Im Standardumfang des Tools sind jedoch schon viele Funktionen vom Kompilieren des Quellcodes über Komprimierung (beispielsweise in JAR Archive) bis hin zum Versenden von E-Mails integriert.
Auch wenn das Projekt in Eclipse bereits den Build-Vorgang komplett integriert, wird Ant dennoch für die Parsergenerierung benötigt. Die Erstellung der Java-Klassen aus den Parseranweisungen sowie der Grammatik übernimmt das Tool *JLex* bzw. *Jay*, die durch spezielle Parameter gesteuert werden. Diese Problematik lässt sich leicht über ANT lösen. Da Eclipse die Integration von ANT erlaubt, bleibt eine komfortable Unterstützung des Build-Vorgangs in der Entwicklungsumgebung erhalten.

3.3.3 Erstellung von Class-Dateien mit javac

Obwohl die Erstellung der Java-Classen mittels der Entwicklungsumgebung Eclipse erfolgt, soll an dieser Stelle noch eine kurze Erläuterung zur Benutzung von javac erfolgen. Dabei soll der Fall betrachtet werden, den generierten Bytecode in einem Hex-Editor anzusehen. Wird der Java-Compiler ohne weitere Parameter ausgeführt, werden zusätzliche Debug-Informationen in die Class-Datei geschrieben. Zu diesen Daten zählen bspw. Informationen über Zeilen, in der ein bestimmter Befehl ausgeführt wird. Dies steigert nicht nur die Größe der Class-Dateien, sondern macht die Betrachtung der Datei deutlich unübersichtlicher.
Um keine Debug-Informationen in der Class-Datei zu generieren, kann der Compiler mit dem Parameter `-g:none` ausgeführt werden. Gerade für die Informationsbeschaffung, wie Generics im Bytecode abgelegt werden, ist diese Einstellung sehr hilfreich.

3.3.4 Java-eigener Decompiler - javap

Das Java JDK Version 1.5 liefert von Haus aus bereits ein Tool zum Decompilieren von Class-Dateien mit. Jedoch wandelt javap den Bytecode nicht zurück in eine Java-Datei, sondern listet implementierte Methoden und ihre Parameterliste auf. Auf Wunsch kann auch der Anweisungscode der Methoden decompiliert werden; dabei werden dann die Bytecodebefehle in leserlicher Form und mit Verweisen auf den Konstantenpool dargestellt.
Der Aufruf von Javac geschieht mit `javac -c <classdatei>`. Der Parameter `-c` bewirkt dabei, dass der Code zu jeder Methode angezeigt wird. Die Classdatei wird ohne Ihre Erweiterung angegeben.

[3] ANT ist die Kurzform von Another Neat Tool
[4] XML: Extended Markup Language

Um sich ein Bild der Ausgabe von javap machen zu können, zeigt Listing 3.1 eine Javaklasse, die zunächst in Bytecodeübersetzt wird. Listing 3.2 zeigt das Ergebnis des Decompilierens mit Hilfe von javap.

```
import java.util.Vector;

public class JavapTest{

        public void TestMethode() {
                Vector<String> zf = new Vector<String>();
                zf.addElement("Hallo␣Welt!");
                String text1 = zf.elementAt(0);
        }
}
```

Listing 3.1: Beispiel für javap, Java-Quellcode

```
public class JavapTest extends java.lang.Object{
public JavapTest();
  Code:
   0:   aload_0
   1:   invokespecial   #1; //Method java/lang/Object."<init>":()V
   4:   return

public void TestMethode();
  Code:
   0:   new     #2; //class java/util/Vector
   3:   dup
   4:   invokespecial   #3;
        //Method java/util/Vector."<init>":()V
   7:   astore_1
   8:   aload_1
   9:   ldc     #4; //String Hallo Welt!
   11:  invokevirtual   #5;
        //Method java/util/Vector.addElement:(Ljava/lang/Object;)V
   14:  aload_1
   15:  iconst_0
   16:  invokevirtual   #6;
        //Method java/util/Vector.elementAt:(I)Ljava/lang/Object;
   19:  checkcast       #7; //class java/lang/String
   22:  astore_2
   23:  return
}
```

Listing 3.2: Ausgabe von javap

3.3.5 JAD - DJ Java Decompiler

Ein deutlich komfortablerer Decompiler für Java ist der DJ Java Decompiler [JAD]. Die Oberfläche setzt dabei auf dem Tool JAD auf und bietet eine übersichtliche Anzeige des zurückentwickelten Codes. JAD selbst ist ein befehlszeilenorientiertes Tool, dass das Reverse-Engineering einer Class-Datei erlaubt. Da es sich bei letzterem um eine kostenlose Open-Source Entwicklung handelt, reagiert die Software bei Fehlern im Bytecode unter Umständen mit kuriosen Ergebnissen.

Folgende Einschränkungen müssen, vor allem auch bedingt durch das Fehlen der Informationen im Bytecode hingenommen werden:

- Sämtliche Kommentare und Formatierungen, die im Quellcode vorgenommen wurden, werden nicht wiederhergestellt.
- Initialisierungen von Variablen innerhalb der Klasse werden nicht direkt, sondern innerhalb des Default-Konstruktors vorgenommen.
- Die Bezeichnung von Variablen in Methoden kann nicht wiederhergestellt werden.
- Features aus Java 1.5 (speziell Generics) werden nicht angezeigt. Dies wird in der aktuellen Version von JAD noch nicht unterstützt.

3.3.6 Hex-Editor

Wie schon im Kapitel „Erstellung von Classdateien mit javac“ erwähnt, ist zur Kontrolle des erzeugten Bytecodes oder zum Vergleich und Informationsgewinn mit Classen, die über den Java Compiler erzeugt wurden, ein Hex-Editor erforderlich. Ein handelsüblicher Texteditor reicht dazu nicht aus, da viele Zeichen nicht direkt darstellbar sind.
Mit einem Hex-Editor lässt sich die Struktur der Class-Datei direkt betrachten und mit Hintergrundwissen über die Struktur analysieren, solange möglichst wenig Code selbst in der Klasse vorhanden ist. Für die Bearbeitung der Studienarbeit habe ich den integrierten Hex-Editor der Produktsuite Microsoft Visual Studio .NET 2005 benutzt. Diese bietet nicht nur einen übersichtlichen Editor, sondern erlaubt es auch, mehrere Dateien gleichzeitig untereinander darzustellen.

3.4 Anpassung des Logging - Log4J

Eine große Hilfe beim Testen und Debuggen von Software ist das Thema Logging. Logging beschreibt den Prozess, Meldungen über den gerade laufenden Prozess oder den Arbeitsfortschritt auszugeben. Hier ist eine schwere Gratwanderung zu vollziehen - zum einen sollten die Ausgaben nicht zu ausführlich sein, damit der Entwickler nicht vor lauter Meldungen den Überblick verliert. Zum anderen sollte es aber auch möglich sein, aus den Meldungen gegebenenfalls Fehlerquellen eingrenzen zu können. Gleichzeitig sollte aber auch der Quellcode übersichtlich bleiben, und keinesfalls mit unzähligen If-Schleifen unleserlicher werden. Um diese Ziele möglichst einfach zu erreichen, bietet sich die Umstellung des Logging auf Apache Log4J an. Log4J ist ein mächtiges Open-Source Logging-Framework, welches viel mehr als nur die Ausgabe der Meldungen auf die Konsole erlaubt. Bei Log4J wird das Logging grob in zwei Bereiche unterteilt: Den Logger selbst und die Targets. Es ist möglich, verschiedene Logger über Log4J zu betreiben - dies dient in erster Linie zur Trennung von Ausgabebereichen. Wird eine Ausgabe über einen bestimmten Logger gemacht, wird dieser Meldung gleichzeitig eine Priorität zugewiesen, von Debug, Info über Warn, Error bis Fatal oder off. Diese Prioritäten geben aufsteigend die Wertigkeit der Nachricht hat. Gleichzeitig wird in der Konfigurationsdatei eine Wertigkeit für den Logger angegeben. Alle Prioritäten, die die in der Konfigurationsdatei angegebene Priorität unterschreiten, werden nicht ausgegeben. Beispiel: Eine Nachricht, die mit dem Level Debug im Quelltext erzeugt wird, erscheint nicht, wenn der Logger auf die Priorität Info eingestellt ist. Die Targets bilden die zweite Komponente zur Ausgabe von Meldungen. Mit Targets lassen sich verschiedene Ausgabeziele definieren, zum Beispiel eine Datei, die Konsole, ein Server im Netzwerk usw. Darüber hinaus bieten Targets die Möglichkeit, Meldung zu formatieren, Informationen über die Quelle hinzuzufügen, usw. Die Einstellung der jeweiligen Logger und der Ausgabeoptionen wird in der Konfigurationsdatei log4j.xml vorgenommen. Diese

bietet die Möglichkeit, die Informationslevel der unterschiedlichen Logger zu verändern und ebenfalls eine Formatierung der Zusatzinformationen wie Ursprungsklasse der Meldung, Zeit, usw. zu setzen.

Insgesamt bietet Log4J genügend Möglichkeiten, um die Logging-Ausgaben für den Entwicklungsprozess zu verbessern. Für den Bereich der Bytecodegenerierung wurden dazu zwei Logger hinzugefügt: Der Logger codegen zeigt Informationen zur Bytecodegenerierung mit verschiedenen Prioritäten, der Logger bytecode zeigt bei der Granularität Info den generierten JVM-Statements. Da das Projekt insgesamt auf Log4J umgestellt wurde, lassen sich auch für den spezifischen Bereich des Entwicklers die interessanten Log-Mitteilungen aktivieren und Informationen aus anderen Phasen des Kompilierens ausblenden.

Für den Start des Compilers ergibt sich dadurch eine Änderung in der Aufrufkonvention. Die Angabe des Debug-Levels als zweiter Parameter ist nun nicht mehr erforderlich und wird auch nicht verarbeitet. Denkbar wäre hier sicherlich, durch eine entsprechende Maske die Ausgabepriorität der Logger über die Befehlszeile zu ändern. Unkomplizierter und intuitiver ist und bleibt aber in jedem Fall die Veränderung der Einstellungen in der Datei `log4j.xml`.

3.5 Portieren des Quellcodes auf Java 1.5

Da zum Zeitpunkt der letzten Studienarbeit mit der Thematik Bytecodegenerierung nur eine Beta-Version des JDK 1.5 verfügbar war, wurden verschiedene neue Features noch nicht genutzt. Im Wesentlichen geht es dabei um die Generics und die Vereinfachung des Quellcodes durch die Definition von Containertypen. Durch die Hinweise in Eclipse vereinfacht sich das Auffinden von untypisierten Containerklassen deutlich.
Zudem war es erforderlich, mit Hilfe der Refactoring-Funktionen der Entwicklungsumgebung den Sourcecode an unübersichtlichen Stellen zu formatieren und Kommentare einzufügen.

Kapitel 4

Umsetzung der Aufgabenstellung

In den folgenden Unterkapiteln sollen die Lösungen der Aufgabenstellung dargelegt werden. Dabei soll nicht nur Wert auf die letztendliche Umsetzung des Punktes gelegt werden, sondern auch Seiteneffekte und Konsequenzen für andere Teilbereiche betrachtet werden. Zur Unterstützung der Programmierung selbst sei hier noch auf die Literatur [Ulle04] verwiesen, die gute Hilfestellungen und Ideen lieferte.

4.1 Integration von Interfaces

Interfaces stellen in der Java-Welt eine Definition von Eigenschaften und Methodensignaturen dar. Dabei ist der wesentliche Unterscheid, dass Interfaces keinerlei Quellcode enthalten. Es handelt sich also um eine abstrakte Klasse, in der auch alle Methoden als abstrakt definiert sind. Es kann keine Instanz eines Interface gebildet werden.

4.1.1 Definierter Funktionsumfang

Die Integration von Interfaces ist prinzipiell zunächst eine Einschränkung der zulässigen Elemente in einer Class-Datei. Erlaubt sind lediglich Konstantendefinitionen und Methodendefinitionen. Weiterhin können Interfaces andere Superinterfaces erweitern.
Speziell für die Verwendung des Typinferenz-Algorithmus ergeben sich Probleme bei der Generierung von Interfaces: Dadurch, dass alle Methoden zwangsläufig abstrakt sein müssen und keinerlei Quellcode enthalten, ist die Berechnung der Typen nur unter Zuhilfenahme von anderen Klassen möglich. Aus diesem Grund ist es für Interfaces momentan vorgesehen, dass die Typangaben bereits im Quelltext gemacht werden.
Für die Realisierung der Klassen ergibt sich ebenfalls eine Neuerung. Jeder Klasse können eine Menge von Superinterfaces zugeordnet werden.

4.1.2 Lösung

Zunächst ist es erforderlich, in Parser und Grammatik Anpassungen vorzunehmen. Eine gute Vorlage, wie sie auch beim Entwurf der kompletten Grammatik verwendet wurde, liefert [BaHo98]. Das neue Schlüsselwort `interface` muss erkannt werden; hierüber wird in der Grammatik eine Unterscheidung zwischen Interfaces und Klassen realisiert. Die Grammatik schränkt ebenfalls die Anzahl der zulässigen Elemente ein: Für den Typ Interface sind lediglich Konstanten (siehe folgendes Kapitel), Instanzvariablen und abstrakte Methoden erlaubt. Sämtliche Informationen aus der geparsten Quellcodedatei werden in den Klassen

`Interface.java` bzw. `InterfaceBody.java` abgelegt. Diese Klassen enthalten wiederum Elemente, die auch von den Klassen genutzt werden.

Da die Differenzen zu einer üblichen Class-Datei nur gering sind, greift die implementierte Lösung zur Bytecodegenerierung von Interfaces auf die bereits vorhandenen Klassen zu.
Der Access-Modifier für Interfaces wird fest auf den Wert `0x601` festgelegt, was der Kombination von `PUBLIC`, `ABSTRACT` und `INTERFACE` entspricht. Zur Liste der Access-Modifier jeder Methode wird ebenfalls die Eigenschaft `ABSTRACT` hinzugefügt.
Die Integration von Superinterfaces, die sowohl Klassen als auch Interfaces betrifft, wird über die Bytecodeeigenschaften *interfaces-count* und *interfaces* realisiert. Für jedes Superinterface wird eine UTF-8 Konstante im Konstantenpool generiert, die den Full-Qualified-Name des Interfaces (Packagedefinition und Interfacebezeichnung) enthält. Zudem wird ein ClassInfo-Element im Konstantenpool abgelegt, dass den Verweis auf die UTF-8 Konstante beinhaltet. Im Bytecode wird an der Stelle *interfaces-count* die tatsächliche Anzahl der implementierten Interfaces abgelegt, und unter *interfaces* zu jeweils 2 Byte die Verweise auf die ClassInfo-Einträge im Konstantenpool.

4.2 Generieren von Konstanten

Durch die neue Möglichkeit, Interfaces über den Compiler zu erzeugen, ergibt sich auch die nötige Erweiterung der Konstantengenerierung. Andernfalls wäre die Funktionalität eines Interfaces sehr eingeschränkt, welches lediglich die Definition von abstrakten Methoden erlaubt.
Die implementierte Lösung erlaubt die Generierung von Konstanten sowohl für Interfaces als auch für Klassen.

4.2.1 Definierter Funktionsumfang

Der Funktionsumfang soll zunächst einmal auf der Stufe der zulässigen primitiven Typen in der Feldgenerierung gehalten werden. Folgende Elemente werden unterstützt:

- Unterstützung der Access-Modifier `PUBLIC`, `PRIVATE` und `PROTECTED`.
- Generieren der Konstantentypen im Format `boolean`, `char`, `int` und `String`.

Eine Erweiterung für andere Datentypen ist offen gehalten und sollte ohne großen Aufwand zu implementieren sein.

4.2.2 Lösung

Wie auch bei der Generierung von Interfaces war zunächst eine Erweiterung des Parsers und der Grammatik erforderlich. Im Falle der Konstantenbehandlung war lediglich eine neue Memberdeklaration für das Interface nötig. Für die Klasse ergab sich das Problem, dass eine Fielddefinition eine sehr ähnliche Syntax hat. Dadurch ist der Parser nicht in der Lage, eine klare Unterteilung zwischen Konstanten und Feldern zu machen. Dies resultierte auch daraus, dass die Modifier nicht einzeln, sondern als Ganzes betrachtet werden. Der Access-Modifier `FINAL` nimmt hier keine Sonderstellung ein. Die Lösung des Problems ist eine dynamische Betrachtung des Felds bei der Bytecodegenerierung: Hier wird überprüft, ob das Field den Modifier `FINAL` trägt. Ist dies der Fall, wird auf die Generierung einer Konstante zurückgegriffen.
Zur Repräsentation, und um unnötige Komplexität zu reduzieren, war die Erstellung der

neuen Klasse `Constant.java` erforderlich. Diese Klasse ist durch die Attribute Access-Modifier, Name als Type sowie Wert vom Typ Expr gekennzeichnet.
In der Darstellung der Classfile werden Konstanten zunächst als normale Fields behandelt, aber zusätzlich mit dem Attribut `ConstantValue` gekennzeichnet. Die Tabelle 4.1 gibt Aufschluss über die genaue Ablage. Dabei wurden die Darstellungen von Strukturen Field-Info und ConstantValue-Attribute der Übersichtlichkeit halber zusammengefasst.

Bezeichnung	**Länge**	**Erläuterung**
Access-Flags	2 Byte	Das Feld beinhaltet die Bitmaske für die Access-Modifier (Public, Protected, Private). Die Modifier Final und Static sind obligatorisch.
Name-Index	2 Byte	Der Name-Index zeigt auf eine UTF-8 Konstante im Konstantenpool, die den Namen der Konstante selbst beinhaltet.
Descriptor-Index	2 Byte	Der Descriptor-Index zeigt auf eine UTF-8 Konstante im Konstantenpool, die den Typ der Konstante festlegt. Eine vollständige Liste der Kodierungen findet sich in Tabelle 2.9.
Attributes-count	2 Byte	Anzahl der Attribute. Die Anzahl ist für diesen Fall auf den Wert 1 festgelegt.
Attribute-Name-Index	2 Byte	Verweist auf die UTF-8 Konstante im Konstantenpool, die den Wert „ConstantValue“ hat. Mehrere Konstantendefinitionen dürfen auf den gleichen Index referenzieren.
Attribute-Length	4 Byte	Gibt die Länge des Attributes an. Im Fall der Konstantendefinition ist die Länge auf den Wert 2 festgelegt.
ConstantValue-Index	2 Byte	Der Index verweist auf das Element im Konstantenpool, dass den Wert der Konstante beinhaltet. Der Typ dieses Elements muss mit dem Typ übereinstimmen, der in der UTF-8 Konstante im Descriptor-Index angegeben wurde.

Tabelle 4.1: Repräsentation einer Konstante im Bytecode

4.3 Unterstützung von Packages

Durch Einteilung des Quellcodes in Packages eröffnet sich in Java eine Zahl von Vorteilen:

- Strukturierung des Quellcodes. Packages dienen der logischen bzw. funktionalen Strukturierung des Programms oder auch von Programmteilen.
- Mit Packages lassen sich hierarchische Strukturen realisieren.
- Namen können wieder verwendet werden. Innerhalb eines Packages kann ein Name nur einmalig verwendet werden.
- Packages erhöhen die Übersichtlichkeit.

4.3.1 Definierter Funktionsumfang

Die Integration einer Packagedefinition selbst stellt nur eine kleine Änderung an Parser sowie Bytecodegenerierung dar. Da zur kompletten Lösung jedoch auch die Unterstützung von Imports hinzuzählt, sind sämtliche Prozesse des Compilers von der Integration und teilweise aufwändigen Änderungen betroffen. Dies trifft speziell auf die Arbeit des Typinferenz-Algorithmus zu: Hier ist bis jetzt keine Unterstützung unterschiedlicher Packages möglich. Aus diesem Grund hatte Timo Holzherr mit diesem Teil der Aufgabenstellung die größte Arbeit.
Unterstützt wird somit nicht nur die Definition einer Package, sondern auch der gezielte Import einzelner Klassen, die im Klassenpfad verfügbar sind. Eine benutzte Klasse, die nicht innerhalb des aktuellen Package liegt, muss zwangsläufig über einen Import bekannt gemacht werden; die Nutzung eines Fully-Qualified-Name ist nicht gestattet.

4.3.2 Lösung

Für die Unterstützung von Imports sowie einer Packagedefinition müssen neue Elemente in die Grammatik aufgenommen werden. Diese werden in die Klasse `SourceFile` übergeben, um dann weiter in die Unifikation und in die Klassengenerierung einzufließen.
Für die Generierung des Bytecodes werden die Import-Anweisungen nicht beachtet. Die Imports dienen lediglich der Arbeitserleichterung des Programmierers. Im Bytecode werden stattdessen bei allen Referenztypen der komplette Name inklusive Packagedefinition angegeben.
Die Packagedefinition fließt dagegen in direkt in die Benennung der Klasse ein. Wie im Kapitel Struktur einer Class-datei erläutert, wird der Name der Klasse als UTF-8 Konstante im Konstantenpool abgelegt. Zusätzlich verweist ein ClassInfo-Eintrag auf diese Konstante. Im Aufbau der Class-Datei folgt dann die Eigenschaft *this-class*, die auf diesen ClassInfo-Eintrag verweist. Für eine Packagedefinition wird die UTF-8 Konstante mit dem Namen um die Packagenamen erweitert, die mit Hilfe der üblichen Formatierung von Slashes (/) voneinander getrennt werden. Als Beispiel würde die Klasse PackageKlasse im Package de.typinferenz in der UTF-8 Konstante als `de/typinferenz/PackageKlasse` dargstellt werden.

4.4 Integration von Generics

Die Eigenschaften und Vorteile von Generics, die mit Java 1.5 verfügbar sind, wurden bereits kurz im Kapitel Grundlagen vorgestellt. Generics erweitern nicht nur den Sprachumfang und die Möglichkeiten von Java, sondern bieten auch Typsicherheit und Vereinfachungen gerade im Bezug von Typecasts.

4.4.1 Definierter Funktionsumfang

Beim Funktionsumfang der Generics müssen vier unterschiedliche Definitionsbereiche beachtet werden:

- Definition in der Klassendefinition selbst zum Erstellen von Containerklassen: Im Kopf der Klasse ist eine Spezifikation der benutzten Typen möglich. Soll die Klasse instanziiert werden, kann zusätzlich angegeben werden, für welchen Objekttyp die Klasse als Container dienen soll.
  Der Entwickler hat ebenfalls die Möglichkeit, Einschränkungen für die verwendeten

Containertypen zu machen. Durch das Schlüsselwort `extends` kann er die Superklasse angeben, von der alle zugelassenen Objekte abgeleitet sein müssen. Sollen zusätzliche Interfaces zwangsweise implementiert werden, so können diese ebenfalls angeben werden. Zur Angabe von mehreren Interfaces, oder einer Superklasse und eines oder mehrerer Interfaces werden die jeweiligen Bezeichner durch ein kaufmännisches Und (`&`) zusammengefügt.

- Spezifikation von Superklassen und implementierten Inferfaces mit gleichzeitiger Angabe des Containertyps: Für die Angaben `extends` (Klassen und Interfaces) sowie `implements` (Klassen) kann nicht nur die Klasse selbst, sondern ebenfalls der Containertyp angegeben werden. Dadurch kann die Containertypangabe der eigentlichen Klasse weggelassen werden.

- Typisierung von Attributen, Parametern oder Rückgabewerten mit Hilfe von Generics: Ist in der Klassendefinition selbst ein Containertyp angegeben, so können diese Elemente durch den Containertyp typisiert werden. Statt einer Klassenbezeichnung wird lediglich der Platzhalter für den generischen Datentyp verwendet.
  Neben der Möglichkeit, den generischen Objekttyp zu verwenden, können auch typisierte Containerklassen als Objekttypen angegeben werden.

- Definition des Rückgabetyps von Methoden. Ähnlich der Definition eines Typs in der Klassendefinition kann auch der Rückgabetyp einer Methode beschrieben werden. Dabei wird die Definition mit einem Namen versehen und der Methodendefiniton vorangestellt. Zur genaueren Spezifikation können die Superklasse und implementierte Interfaces wiederum mit `extends` und `&` angegeben werden.

4.4.2 Lösung

Die Informationsbeschaffung für die Integration von Generics in den Bytecode stellte sich als recht aufwändig heraus, da kaum vollständige Informationen zu diesem Thema verfügbar sind. Nach aufwändiger Literatursuche blieb nur der Weg, die Repräsentation von Generics im Bytecode durch Erstellen von Beispielklassen, Kompilieren und der Betrachten des Bytecodes im Hex-Editor.
Dabei stellte sich heraus, dass die eigentlichen Definitionen der Objektypen nicht geändert werden. Erfolgt beispielsweise die Definition eines Attributs vom Typ `Vector<String>`, so wird dieser intern als `Ljava/util/Vector` spezifiziert. Dieser Repräsentationsweg wurde sicherlich auch aus Gründen der Abwärtskompatiblität und Performanz gewählt - eine generische Klasse sollte auch auf älteren Implementierungen der JVM lauffähig sein. Zudem bieten Generics in erster Linie eine Erleichterung für den Entwickler, haben aber keinerlei Effekt auf den eigentlichen Programmcode.

Die Informationsablage erfolgt bei allen Elementen (Klassen beziehungsweise Interfaces, Attributen oder Methodendefinitionen) in gleichen Konventionen. Dazu wird in der jeweiligen Attributstruktur ein Attribut mit der Bezeichnung Signature abgelegt. Die Tabelle 4.2 zeigt den Aufbau des Attributs.
Der Inhalt der Konstante ist abhängig vom Kontext und Definition des Datentyps. Die Tabellen 4.3 bis 4.12 veranschaulichen die Ablage der Informationen.

Der wesentliche Teil der Signaturverarbeitung findet in der Klasse `SignatureInfo.java` statt. Diese bietet zum einen eine Überprüfung, ob das Attribut, die Methode oder die Klassendefinition überhaupt eine Signaturdefinition benötigt, und liefert ein entsprechendes Signatur-Attribut zurück. Dieses kann in die Auflistung des jeweiligen Elements hinzugefügt werden.

Bezeichnung	Länge	Erläuterung
Attribute-Name-Index	2 Byte	Verweist auf die UTF-8 Konstante im Konstantenpool, die den Wert „Signature“ hat. Mehrere Konstantendefinitionen dürfen auf den gleichen Index referenzieren.
Attribute-Length	4 Byte	Gibt die Länge des Attributes an. Diese ist auf den Wert 2 festgelegt.
Signature-Index	2 Byte	Der Index verweist auf die UTF-8 Konstante im Konstantenpool, die die Signaturdefinition enthält.

Tabelle 4.2: Aufbau des Attributs Signature

Funktion:	Spezifikation des enthaltenen Typs.
Kontext:	Variablendefinition, Parameterdefinition oder Rückgabetyp.
Quelltext:	`Vector<String> variable1;`
Signatur:	`Ljava/util/Vector<Ljava/lang/String;>;`

Tabelle 4.3: Spezifikation des enthaltenen Typs

Funktion:	Spezifikation mehrerer enthaltener Typen.
Kontext:	Variablendefinition, Parameterdefinition oder Rückgabetyp.
Quelltext:	`Hashtable<String, Integer> variable1;`
Signatur:	`Ljava/util/Hashtable` `<Ljava/lang/String;Ljava/lang/Integer;>;`

Tabelle 4.4: Spezifikation mehrerer enthaltener Typen

Funktion:	Spezifikation eines Containertyps. Bei Instanziierung der Klasse wird der Typ der Variablen angegeben. Der Name muss in der Klassendefinition bereits definiert sein.
Kontext:	Variablendefinition, Parameterdefinition oder Rückgabetyp.
Quelltext:	`typ1 variable1;`
Signatur:	`Ttyp1;`

Tabelle 4.5: Spezifikation eines Containertyps.

Funktion:	Erlaubt es, beim Instanziieren der Klassen den Containertyp zu bestimmen.
Kontext:	Klassen- oder Interfacedefinition.
Quelltext:	`public class Generics<Typ1, Typ2>`
Signatur:	`<Typ1:Ljava/lang/Object;Typ2:Ljava/lang/Object;>` `Ljava/lang/Object;`

Tabelle 4.6: Typdefinition im Klassenkopf

Funktion:	Gibt dem Programmierer die Möglichkeit, beim Instanziieren der Klassen den Containertyp zu bestimmen. Dabei wird die Anzahl der möglichen Containertypen beschränkt, indem die Superklasse angegeben wird.
Kontext:	Klassen- oder Interfacedefinition.
Quelltext:	`public class Generics<Typ1 extends Vector>`
Signatur:	`<Typ1:Ljava/util/Vector;>` `Ljava/lang/Object;`

Tabelle 4.7: Typdefinition im Klassenkopf mit Superklasse

Funktion:	Gibt dem Programmierer die Möglichkeit, beim Instanziieren der Klassen den Containertyp zu bestimmen. Dabei wird die Anzahl der möglichen Containertypen beschränkt, indem die Superklasse und implementierte Inferfaces angegeben werden.
Kontext:	Klassen- oder Interfacedefinition.
Quelltext:	`public class Generics<Typ1 extends Vector & Comparable>`
Signatur:	`<Typ1:Ljava/util/Vector;:Ljava/lang/Comparable;> Ljava/lang/Object;`

Tabelle 4.8: Typdefinition im Klassenkopf mit Superklasse und Interface

Funktion:	Durch Angabe des Containertyps für die Superklasse kann dieser bereits durch die abgeleitete Klasse festgelegt werden.
Kontext:	Klassen- oder Interfacedefinition.
Quelltext:	`public class Generics<Typ1> extends Vector<String>`
Signatur:	`<Typ1:Ljava/lang/Object;> Ljava/util/Vector<Ljava/lang/String;>;`

Tabelle 4.9: Typdefinition für Superklasse

Funktion:	Durch Angabe des Containertyps für die Superklasse und implementierte Interfaces können diese bereits durch die abgeleitete Klasse festgelegt werden.
Kontext:	Klassen- oder Interfacedefinition.
Quelltext:	`public class Generics<Typ1> extends Vector<String> implements Comparable<String>`
Signatur:	`<Typ1:Ljava/lang/Object;> Ljava/util/Vector<Ljava/lang/String;>; Ljava/lang/Comparable<Ljava/lang/String;>;`

Tabelle 4.10: Typdefinition für Superklasse und implementierte Interfaces

Funktion:	Spezifikation des Rückgabetyps. Bei Instanziierung der Klasse wird der Typ der Variablen angegeben. Der Name muss in der Klassendefinition bereits definiert sein.
Kontext:	Rückgabetyp eine Methode.
Quelltext:	`typ1 Methode1(String wert1)`
Signatur:	`(Ljava/lang/String;)Ttyp1;`

Tabelle 4.11: Containertyp als Rückgabetyp einer Methode

Funktion:	Spezifikation des Rückgabetyps einer Methode, um Einschränkungen zu machen.
Kontext:	Rückgabetyp eine Methode.
Quelltext:	`<typ1 extends Vector> typ1 Methode1(String wert1)`
Signatur:	`<typ1:Ljava/util/Vector;>(Ljava/lang/String;)Ttyp1;`

Tabelle 4.12: Spezifikation des Rückgabetyps einer Methode

4.5 Boxing/Unboxing von Datentypen

4.5.1 Problemstellung

Alle Operationen und Auswertungen in der Bytecodewelt, die in erster Linie mathematischer Natur sind, zum Beispiel die Addition, Subtraktion oder Vergleiche, können im Bytecode nur auf primitiven Typen ausgeführt werden. Weiterhin ist es auch möglich, dass Methoden als Parameter nicht nur Referenztypen, sondern auch primitive Typen erwarten, oder solche zurückgeben.
Dies steht im Gegensatz zur Realisierung des Compilers, der nur Objekttypen berücksichtigt. Um dennoch den Funktionsumfang dieser elementaren Operationen zu erhalten, soll das Boxing und Unboxing von Datentypen, wie es auch in Java 1.5 integriert ist, angewendet werden. Dazu muss der Entwickler nicht weiter beachten, ob es sich um einen Referenztyp oder um einen primitiven Typ handelt; erst bei der Erstellung des Bytecodes wird überprüft, welchen Parametertyp die Methode oder die Operation verlangt.

4.5.2 Eingeschränkter Lösungsumfang

Um die Übersichtlichkeit zu wahren, wird zunächst folgende Annahme getroffen: Alle Typen liegen grundsätzlich in Objektform, also bei den primitiven Typen in Wrapper-Klassen vor. Weiterhin bekommen speziell die Klasse `RefType` und alle Literale die Eigenschaft *primitiveType*, die kennzeichnet, ob die Klasse als primitiver Typ oder als Objekttyp im Bytecode generiert werden soll.
Für die Realisierung der korrekten Methodenaufrufe wird bereits bei der Typberechnung festgelegt, ob ein Typ als Referenz oder als primitiver Typ übergeben wird. Dazu wird bei dem entsprechenden Referenztyp das primitive-Flag gesetzt, und die Signatur der Methode entsprechend übergeben. Bei der Generierung des Bytecodes wird der Code in Abhängigkeit des Attributs erstellt.
Aus der Annahme ergibt sich, dass die Generierung von Konstanten grundsätzlich als Objekt erfolgt. Damit sind einfache Wertzuweisungen der Form `a=5` möglich, auch wenn die Variable `a` den Typ Integer hat.

Nicht integriert sind zum Ende dieser Studienarbeit die mathematischen Operationen sowie die Vergleichsoperationen, die vom Compiler unterstützt werden. Hier ergibt sich die Problematik, dass die Multiplikation zum Beispiel mit einer Zahl von Literalen, einer Zahl von Objekten oder einer Kombination dieser möglich sein muss. Für die Klasse TimesOp, die die Multiplikation darstellt, wurde bereits eine exemplarische Lösung erstellt, die allerdings noch ausgiebig auf unterschiedliche Anwendungsfälle getestet werden muss.
Für Operationen, die auf unterschiedlichen primitiven Typen möglich sind, (beispielsweise der Vergleichsoperator) sind weiterführende Überlegungen erforderlich: Bevor das Wrapper-Objekt in den primitiven Typ gewandelt werden kann, muss eindeutig festgestellt werden, um welchen Typ sich es handelt. Andernfalls führt die Operation zu einem Laufzeitfehler. Gerade für komplexe Bedingungen sollte hier zunächst eine sinnvolle und robuste Lösung erarbeitet werden, bevor diese implementiert wird.

Kapitel 5

Erweiterungen

5.1 Automatische Tests

Um den implementierten Funktionsumfang des Compilers auf Korrektheit zu überprüfen, werden Use-Cases definiert und als Eingabe zum Testen verwendet. Diese sind zum großen Teil speziell zum Testen einzelner Funktionalitäten konstruiert.
Beim Debugging neuer Funktionalitäten liefern sie zudem gute Möglichkeiten, Probleme und Fehler aufzufinden. Ein deutliches Problem dabei ist jedoch, dass normalerweise nur ein Usecase in den jeweiligen Testklassen aufgerufen wird. Die Entwicklungsumgebung lässt zwar unterschiedliche Parameter zum Aufruf zu, aber selbst die Auswahl des Usecases über ein Öffnen-Dialogfeld ist zum Testen aller Usecases zu aufwändig.
Zur Automatisierung dieses Vorgangs bieten sich JUnit-Tests an. JUnit ist ein Framework, dass über die Tests selbst noch eine Vielzahl von Auswertungs- und Testmöglichkeiten erlaubt. Für den Compiler soll als Test dienen, dass der angegebene Usecase korrekt verarbeitet wird, und der Kompilierungsvorgang nicht vorzeitg mit einem Fehler abbricht. Dafür werden in der Klasse `JUnitTests.java` für jeden Usecase (jede `.jav` Datei) eine Instanz von `CompilerTest.java` erzeugt. Dazu werden alle Logger auf den Wert `ALL` gesetzt und sämtliche Ausgaben in die Datei mit Namen des Usecases und Erweiterung `.LOG` im Unterverzeichnis testResults abgelegt. Mit Hilfe des variablen Ausgabepfads werden sämtliche Class-Dateien, die korrekt erzeugt werden konnten, im Unterverzeichnis Bytecode abgelegt.
Das Ausführen der Testläufe lässt sich entweder über die Entwicklungsumgebung Eclipse, als JUnit-Test, oder auch per Befehlszeile machen. In Eclipse werden dabei der zeitliche Ablauf und der Ausgang des Tests (erfolgreich oder fehlerhaft) dargestellt. Bei einem Klick auf den jeweiligen Usecase lässt sich über die Ausgabe des Stack-Trace direkt feststellen, an welcher Stelle ein Fehler auftrat.
Insgesamt bleibt jedoch zu beachten, dass die JUnit-Tests nur auf ein korrektes Durchlaufen des Kompilierungsvorgangs überprüfen. Es wird nicht getestet, ob beispielsweise die korrekten Typen erkannt wurden oder der Bytecode korrekt ist.

5.2 Variabler Ausgabepfad

Speziell mit der Realsisierung der automatischen Tests ergab sich die Notwendigkeit, ein besseres Handling für die Ausgabe der Class-Dateien zu schaffen. Das Ausführen hatte die Konsequenz, dass alle erzeugten Dateien im Rootverzeichnis des Projekts abgelegt wurden. Die implementierte Lösung bietet nun die Möglichkeit, den Ausgabepfad variabel über die Klasse `MyCompiler` festzulegen.

Kapitel 6

Schlussbetrachtung

6.1 Erkenntnisse und Fazit

Im Rahmen dieser Studienarbeit wurden die Einzelteile der Aufgabenstellung sowie ein paar zusätzliche Erweiterungen implementiert.
Wie auch bereits in der Aufgabenstellung ausgeschrieben, brachte die Arbeit nicht nur tiefere Kenntnisse und eine größere Erfahrung mit der Programmiersprache Java mit sich, sondern auch in den Compilerbau und speziell in den Aufbau einer Class-Datei. Die Realisierung der Generics bot eine große Herausforderung, da Informationen zu diesem Thema nicht in Form einer Spezifikation vorlagen. Sicherlich wird in absehbarer Zeit eine neue Auflage von [LiYe99] erscheinen, die auch auf die Erweiterungen von Java 1.5 eingeht.
Kennzeichnend für das Projekt ist eine gewisse Komplexität, auch bei alleiniger Betrachtung des Teilbereichs Bytecodegenerierung. Eine Einarbeitung in die Thematik ist sehr zeitaufwändig, nicht nur in die offizielle Spezifikation sondern auch in die Bytecodegenerierung selbst. Dies mag mit daran liegen, dass teilweise eine recht spartanische Kommentierung den Prozess nicht vereinfacht. Aus diesem Grund habe ich mich bemüht, gerade an komplizierten Abläufen Kommentare hinzuzufügen, und auch Methoden und Klassen mit JavaDoc-konformen Kommentaren zu versehen.
Das Debugging und die Fehlersuche ist bei der Bytecodegenerierung ein sehr aufwändiger Prozess. Gerade für das Lokalisieren von Problemen, die sich nicht durch den Abbruch des Generierens, sondern durch Fehler im Bytecode ausdrücken, bleibt oftmals nur die Nutzung eines Hex-Editors.

Aus Zeitgründen war es nicht möglich, die Funktionalität des Boxings/Unboxings komplett zu integrieren. Im Kapitel 4.5 wird dazu jedoch bereits eine Möglichkeit gezeigt, und einige Denkansätze geliefert. Damit sollte es für folgende Studienprojekte leichter möglich sein, die fehlende Funktionalität zu ergänzen.

6.2 Ausblick

Um den Compiler auf eine stabile Version zu bringen, sind sicherlich noch einige Stunden Arbeit erforderlich. Neben etwas aussagekräftigeren Fehlermeldungen aus der Grammatik, sind noch einige Checks auf Korrektheit nötig. Semantische Korrektheit, aber auch die Kombination von unterschiedlichen Elementen miteinander wären zu überprüfen.

Literaturverzeichnis

[LiYe99] Tim Lindholm, Frank Yellin. The Java Virtual Machine Sepcification.
Second Edition Addison Wesley, 1999,
ISBN 0-201-43294-3

[BaHo98] Bernhard Bauer, Riitta Höllerer. Übersetzung objektorientierter Programmiersprachen.
Springer Verlag, 1998
ISBN 3-540-62256-0

[bajo05] Jörg Bäuerle. Typinferenz in Java
Berufsakademie Horb, Juni 2005.

[hama04] Markus Haas. Weiterentwicklung der Java-Codegenerierung zur Ausführung von parametrisierten Datentypen.
Berufsakademie Horb, Juni 2004.

[Ulle04] Christian Ullenboom. Java ist auch eine Insel.
4. Auflage. Galileo Computing, 2004.
ISBN 3-89842-526-6
http://www.galileocomputing.de/openbook/javainsel4/
Abruf am 11. Mai 2006.

[Brac04] Gilad Bracha. Generics in the Java Programming Language.
July 5, 2004.
http://java.sun.com/j2se/1.5/pdf/generics-tutorial.pdf
Abruf am 11. Mai 2006.

[LaKr04] Angelika Langer, Klaus Kreft. Java Generics - Parametrisierte Typen und Methoden
JavaMagazin, April 2004.
http://www.angelikalanger.com/Articles/JavaMagazin/Generics/GenericsPart1.html
Abruf am 11. Mai 2006.

[JAD] Atanas Neshkov. DJ Java Decompiler.
http://members.fortunecity.com/neshkov/dj.html
Abruf am 11. Mai 2006.

Anhang A

Usecases für die Bytecode-Generierung

A.1 InterfaceTest.jav

Dient zum Test der korrekten Generierung von Interfaces mit allen Elementen.

```
public interface InterfaceTest extends Interface1, Interface2 {
        public final static boolean testcase = true;
        public final static int variable = 32;
        public final static char testchar = 'A';
        public final static String var2 = "Hallo Welt!";

        public void TestProzedur();
        public void TestProzedur2(String wert1, int wert2);
        public int TestProzedur3(int wert2);
}
```

A.2 SimpleMethodCall.jav

Dient zum Test auf korrekte Verarbeitung von Imports.

```
import java.util.Vector;

public class SimpleMethodCall {
        public TestProzedur() {
                v;
                v = new Vector<String>();
                x;
                x = "Hallo Welt!";

                v.addElement(x);
                return v;
        }
}
```

A.3 PackageTest.jav

Testet die Bytecodegenerierung auf korrekte Verarbeitung von Packages.

```
package testPackage.second.third;

import java.util.Vector;

class PackageTest {
        public final String variable = "Hallo␣Welt!";

        TestProzedur() {
           c;
           c = new Vector();
           return c;
        }
}
```

A.4 ClassKonstanten.jav

Testet die Bytecodegenerierung auf korrekte Erkennung von Konstantendefinitionen.

```
public class ClassKonstanten {
        public final static boolean testcase = true;
        public final static int variable = 32;
        public final static char testchar = 'A';
        public final static String var2 = "Hallo␣Welt!";
}
```

A.5 Boxing.jav

Demonstriert die bisherige Funktionalität des Boxings/Unboxings.

```
import java.util.Vector;

public class Boxing {

        public BoxingTest() {
                a;
                a= new Integer(12);

                v;
                v = new Vector<Integer>();
                v.addElement(a);

                b;
                b = 0;
```

```
                c;
                c = v.elementAt(b);

                v.addElement(c);

                return v;
        }
}
```

A.6 Testen von Generics

Die folgenden Usecases überprüfen die Generierung von Generics für Klassen und Interfaces auf jeweils unterschiedliche Definitionsbereiche.

A.6.1 ClassGenerics.jav

```
import java.util.Vector;
import java.lang.Comparable;

public class ClassGenerics<typ1 extends Vector & Comparable, typ2>
                extends Vector<String>
                implements Comparable<String>{
        public typ1 variable1;
        protected typ2 variable2;

        public Integer compareTo(String s) {
                return null;
        }
}
```

A.6.2 InterfaceGenerics.jav

```
import java.util.Vector;
import java.lang.Comparable;
import java.io.Serializable;

public interface InterfaceGenerics<typ1 extends Vector & Comparable,
                typ2> extends Comparable<String>, Serializable<String>{

        public typ1 compareTo(typ2 s);
}
```

A.6.3 FieldGenerics.jav

```
import java.util.Vector;
import java.util.Hashtable;

public class FieldGenerics {
        public Vector<String> StringVec;
        public Hashtable<String, Integer> ht;
}
```

A.6.4 MethodGenerics.jav

```
import java.util.Vector;
import java.io.Serializable;
import java.lang.Comparable;

public class MethodGenerics<typ1> {

        public typ1 TestMethode(String wert1, typ1 that, String wert2) {
                return null;
        }

        <t extends Vector> t TestMethode2() {
            return null;
        }

        <u extends Vector & Comparable & Serializable> u TestMethode3() {
            return null;
        }
}
```