

Prüfer: Prof. Dr. Erhard Plödereder

Betreuer: Dr. Rainer Koschke

Beginn am: 01.04.2002

Beendet am: 01.10.2002

CR-Klassifikation: D.3.4

Studiengang: Softwaretechnik

Diplomarbeit Nr. 2006

Erweiterung und Generierung der Zwischendarstellung IML für Java-Programme

Markus Knauß, Mtr. Nr. 1808971

Institut für Informatik
Universität Stuttgart
Breitwiesenstr. 20 - 22
D-70565 Stuttgart

Zusammenfassung

Diese Ausarbeitung dokumentiert die Diplomarbeit “Erweiterung und Generierung der Zwischendarstellung IML für Java-Programme”. Dokumentiert sind die einzelnen Tätigkeiten, die durchgeführt wurden, um eine IML-Darstellung von Java-Programmen erstellen zu können. Zu diesen Tätigkeiten gehören die Auswahl eines Java-Übersetzers, der für die IML-Erzeugung modifiziert wurde, die Erweiterung der Zwischendarstellung IML für Java-Programme, die Implementierung des Übersetzers für die Erzeugung der IML und dessen Test sowie die Untersuchung der Auswirkungen auf die Bauhaus-Werkzeuge durch die Erweiterung der IML für Java-Programme.



Inhaltsverzeichnis

Kapitel 1 Einleitung	1
1.1 Bauhaus Projekt	2
1.1.1 Intermediate Language	3
1.2 Einführung in die Aufgabenstellung der Diplomarbeit	4
1.2.1 Einordnung der Diplomarbeit in Bauhaus	5
1.3 Übersicht über das Dokument	5
1.4 Anforderungen an den Leser	6
Kapitel 2 Auswahl eines Java-Übersetzer-Frontends für die Erzeugung der IML	7
2.1 Anforderungen	7
2.2 Evaluation der Java-Übersetzer	8
2.2.1 Implementierung der Java-Sprachspezifikation	9
2.2.2 Lizenzen	10
2.2.3 Implementierung	11
2.2.4 Zugriff und Vollständigkeit auf erzeugten AST und Symboltabelle	12
2.2.5 Aufwand für die Anpassung der IML Erzeugung	12
2.2.6 Zugriff von Ada95	13
2.2.7 Zukunftssicherheit	13
2.3 Auswahl des Java-Übersetzers	13
Kapitel 3 Erweiterung und Spezifikation der IML für Java	15
3.1 Vorgehen für die Erweiterung der IML	15
3.1.1 Erweiterung der IML	15

3.2 Spezifikation der IML für Java	17
3.2.1 Spezifikation der Erweiterungen	17
3.2.2 Übersetzungseinheiten	18
3.2.3 Klassen, Interfaces und Vererbung	22
3.2.4 Konstruktoren, Methoden und Destruktoren	29
3.2.5 Aufruf von Methoden	35
3.2.6 Ausnahmen und Ausnahmebehandlung	39
3.2.7 Auslösen von Ausnahmen	42
3.2.8 Operatoren	44
3.2.9 Typumwandlung	47
3.2.10 Objektknoten und Typknoten	49
Kapitel 4 Entwurf und Implementierung	53
4.1 Wahl der Implementierungssprache	53
4.2 Entwurf von jafe	55
4.2.1 Entwurfsidee	55
4.2.2 Klassenmodell	56
4.2.3 Erzeugung der IML	57
4.3 Implementierung	59
4.3.1 Auflösung von Methodenaufrufen	60
4.3.2 Lokale Variablen	62
Kapitel 5 Test der Implementierung	63
5.1 Testvorgehen	63
5.1.1 Test während der Entwicklung	64
5.1.2 Regressionstest	64
5.1.3 Vergleich der Vorgehensweisen für den Test	64
5.2 Testergebnisse	65
Kapitel 6 Auswirkungen auf Bauhaus-Werkzeuge	67
6.1 Werkzeuge	67
6.2 Auswirkungen auf die Werkzeuge	69
6.3 Bewertung der Auswirkungen	71
6.3.1 Auswirkung der Modellierung von Ausnahmen	71
6.3.2 Auswirkungen von Aufrufen virtueller Funktionen	72
Kapitel 7 Ausblick und Bewertung	73
7.1 Ausblick auf zukünftige Entwicklung und Anwendung	73
7.1.1 Einsatz im Rahmen des Bauhaus Projekts	73
7.1.2 Weiterentwicklung von jafe	74
7.1.3 Unterstützung der Softwareentwicklung	74
7.1.4 Unterstützung des Software-Reengineering	75
7.2 Bewertung der Diplomarbeit	75

Anhang A Glossar verwendeter Begriffe und Abkürzungen	79
Anhang B Literatur	83
Anhang C Ressourcen im Internet	87
Anhang D Erklärung	89



Kapitel 1 Einleitung

“Although maintenance may turn out to be easier for programs written in such languages (object-oriented), it is unlikely that the maintenance burden will completely disappear.”

[Huitt, Wilde 1992]

Das obige Zitat, nach Huitt und Wilde, gibt wenig Hoffnung, dass durch die Entwicklung immer modernerer und einfacher zu verstehender Programmiersprachen, Probleme in der Wartung und Pflege von Software behoben werden. Objektorientierte Programmiersprachen, während der neunziger Jahre als Allheilmittel für alle Probleme der Softwareentwicklung und Wartung gepriesen, wurden ihrem Anspruch nicht gerecht. Es wurde klar, dass durch objektorientierte Programmiersprachen einiges verbessert wurde, aber auch neue Schwierigkeiten hinzukamen. So wurde zum Beispiel Vererbung als Abstraktionsmechanismus eingeführt. Durch Vererbung entstanden aber neue Probleme zum Beispiel im Bereich des Tests objektorientierter Software.

Das Bauhaus-Projekt der Universität Stuttgart verfolgt, unter anderen, die Ziele, Programmverstehen und Softwarewartung zu unterstützen. Bisher ist es mit Bauhaus möglich, Programme zu untersuchen, die in C implementiert sind. Die Untersuchung moderner, objektorientierter Programmiersprachen war mit Bauhaus bisher nicht möglich. Die Möglichkeit der Untersuchung der objektorientierten Sprache Java war Ziel dieser Diplomarbeit.

Die folgenden Abschnitte beschreiben nun zunächst in Kürze das Bauhaus-Projekt. Im Anschluss daran folgt die Beschreibung der Aufgabenstellung dieser Diplomarbeit.

1.1 Bauhaus Projekt

“Bauhaus’ mission is:

- *to leverage existing advanced compiler technology*
- *and to develop new analyses*

to support

- *program understanding,*
- *reverse engineering, in particular architecture recovery,*
- *software maintenance,*
- *and technical audits.”*

Das obige Zitat, aus den Einführungsunterlagen zu Bauhaus, stellt die Ziele klar heraus, die von Bauhaus verfolgt werden. Es sind der Einsatz und die Erprobung neuester Übersetzertechnologien und die Entwicklung neuer Analysewerkzeuge zur Unterstützung von Programmverstehen, Architekturerkennung und Software-Reengineering, Software-Wartung und Software-Inspektionen. Um diese Ziele erreichen zu können, werden Programme durch verschiedene Analysen untersucht.

Die Analysen, die auf den Programmen durchgeführt werden, arbeiten auf einer vereinheitlichten Zwischendarstellung. Diese Zwischendarstellung, die Intermediate Language (IML), wird im folgenden Abschnitt genauer beschrieben. Die IML-Darstellung eines Programms wird von einem Übersetzer erzeugt. Bisher gab es einen Übersetzer für die Programmiersprache C (cafe).

Bei der Untersuchung eines Softwaresystems genügt es meist nicht, die Implementierung einer einzigen Datei in IML zu übersetzen. Für Softwaresysteme müssen meist eine Vielzahl von Dateien, jeweils mit der Implementierung eines Teils des Softwaresystems, in IML übersetzt werden. Für jede übersetzte Datei wird eine eigene IML-Darstellung erzeugt. Um eine Analyse des gesamten Softwaresystems zu ermöglichen, müssen die einzelnen IML-Darstellungen zusammengefasst werden. Dieses Zusammenfassen wird von einem Linker (`imllink`) übernommen.

Die einzelnen Schritte für die Übersetzung eines Softwaresystems nach IML, um die Datengrundlage für die Bauhaus-Analysen zu schaffen, ist in Abbildung 1 dargestellt. Das beispielhafte Softwaresystem besteht aus den Dateien `a.c` und `b.c`, in denen jeweils ein Teil des Softwaresystems implementiert ist. Diese beiden Quelltextdateien werden durch das C-Frontend `cafe` in IML übersetzt. Ergebnis der Übersetzung sind die Dateien `a.iml` und `b.iml`. Die Datei `a.iml` enthält den IML-Graph für das Programm in `a.c` und die Datei `b.iml` den IML-Graph für das Programm `b.c`. Um eine Analyse des gesamten Softwaresystems zu ermöglichen, werden die beiden IML-Dateien `a.iml` und `b.iml` von dem Linker `imllink` zusammengefasst. Der Linker erzeugt die Datei `system.iml`, in der die IML-Darstellung des gesamten Softwaresystems enthalten ist. Auf der Datei `system.iml` arbeiten dann die Analysewerkzeuge von Bauhaus.

Analysen, die im Moment in Bauhaus zur Verfügung stehen, sind zum Beispiel:

- Erzeugung von Kontrollflussinformation,
- Entdecken nicht verwendeter Routinen,
- Feststellen möglicher Ziele, auf die ein Zeiger zeigen kann (Points-To-Analysis),

- Entdecken von Seiteneffekten und
- Entdecken der Verwendung uninitialisierter Variablen.

Die Informationen, die während der einzelnen Analysen gesammelt werden, werden im das Softwaresystem modellierenden IML-Graph gespeichert. Die einzelnen Analysen werden mit zunehmender Information immer komplexer, bis zum Schluss zum Beispiel versucht wird, Komponenten in Programmen zu entdecken.

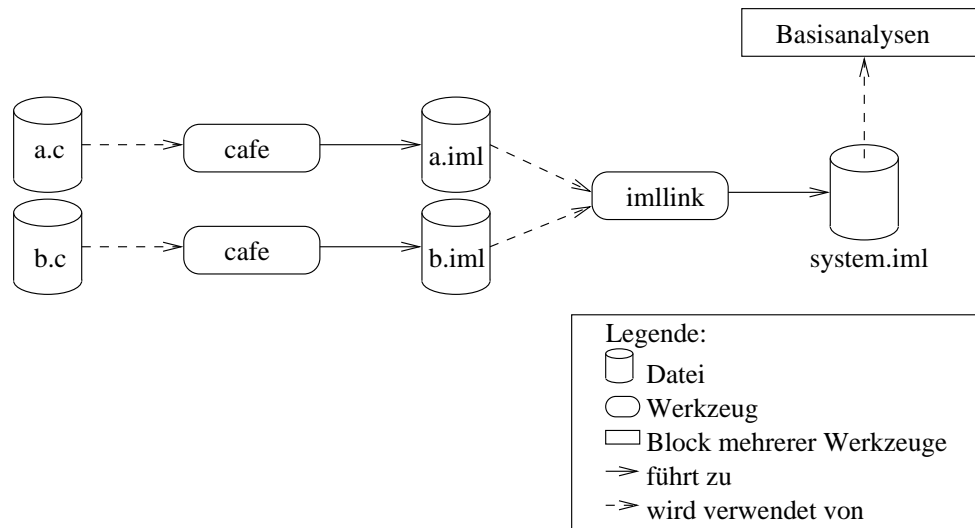


Abbildung 1: Bauhaus Werkzeuge bis zu den Basisanalysen

1.1.1 Intermediate Language

Die IML ist die grundlegende Datenstruktur für die Bauhaus-Werkzeuge. In der IML werden Programme, die in verschiedenen Programmiersprachen implementiert sind, einheitlich dargestellt. Bisher verfügte die IML über die Möglichkeit der Darstellung von C-Programmen und einer Teilmenge von Ada-Programmen.

Die Idee für die IML wird in [Eisenbarth et al. 1999] beschrieben. Eine Spezifikation der IML für die Programmiersprache C findet sich in [Rohrbach 1998].

Die IML besteht aus Klassen, die einen abstrakten Syntaxbaum (Abstract-Syntax-Tree (AST)) mit zusätzlichen, für die Programmanalyse nützlichen Informationen bilden. Die Klassen der IML sind in einer Vererbungshierarchie angeordnet. Für die Spezifikation der IML existiert eine spezielle Beschreibungssprache. In der Beschreibungssprache werden die zur Verfügung stehenden Klassen, deren Attribute und die Vererbungsbeziehungen spezifiziert.

Die IML unterscheidet drei grundsätzliche Arten von Klassen, die für den Aufbau eines IML-Graphen verwendet werden. Diese drei Arten sind:

- Klassen für den hierarchischen Programmgraph (HPG)
Mit Objekten (Knoten) dieser Klassen wird die IML-Repräsentation eines Programms erstellt. Die IML-Repräsentation entspricht einem AST mit zusätzlichen Informationen für die Programmanalyse.

- **Objekt-Klassen**
Knoten von Objekt-Klassen repräsentieren Objekte in einem Programm. Dies sind zum Beispiel Variablen in einem Programm.
- **Typ-Klassen**
Mit Typ-Klassen werden Typen, die in einem Programm definiert werden, repräsentiert. Eine Typ-Klasse repräsentiert zum Beispiel einen benutzerdefinierten Typ, eine Klasse oder ein Interface, oder einen Basisdatentyp.

Die Spezifikation der IML wird mit Hilfe des IML-Generators in eine Ada95-Implementierung übersetzt. Der Generator erzeugt für die spezifizierten Attribute einer Klasse Prozeduren und Funktionen, mit denen es möglich ist, die IML-Repräsentation eines Programms aufzubauen. Durch Aufruf von Konstruktoren für die einzelnen Klassen und Setzen der Attribute können Knoten des IML-Graphen für die Darstellung eines Programms erzeugt werden.

Die einzelnen Knoten eines IML-Graphen sind durch Kanten verbunden. Die Kanten werden unterschieden in syntaktische und semantische Kanten. Die syntaktischen Kanten spannen einen Baum auf. Der Baum repräsentiert die syntaktische Struktur des Programms und bildet den AST. Durch die semantischen Kanten wird zusätzliche Informationen in den IML-Graphen eingebracht. Zusätzliche Informationen sind zum Beispiel Verweise auf die in einem Ausdruck verwendeten Objekte und dessen Ergebnistyp.

Jedes in IML dargestellte Programm kann auf einer virtuellen Maschine ausgeführt werden, die den Befehlssatz IML versteht. Bei der Modellierung muss darauf geachtet werden, dass eine spätere Programmausführung möglich ist.

1.2 Einführung in die Aufgabenstellung der Diplomarbeit

Das Ziel der Diplomarbeit war die Entwicklung eines Übersetzers für Java-Programme in eine IML-Repräsentation. Bei der Entwicklung des Übersetzers mussten einige Nebenbedingungen eingehalten werden. Diese Nebenbedingungen werden in den folgenden Abschnitten vorgestellt.

Der Übersetzer, der Java nach IML übersetzt, basiert auf einem existierenden Java-Übersetzer-Frontend. Der Java-Übersetzer, dessen Frontend verwendet wurde, musste in einem Vorprojekt zunächst ausgewählt werden. Weiterhin musste in einem Vorprojekt der Ausgangspunkt der Übersetzung von Java-Programmen nach IML festgelegt werden. Als Ausgangspunkte waren Java-Bytecode und Java-Quelltext möglich.

Bei der Konzeption der Darstellung von Java-Programmen in IML musste darauf geachtet werden, dass die Darstellung quellennah ist. Das bedeutet, dass zwischen der Darstellung in IML und der Implementierung des Programms ein Wechsel möglich sein muss. Von einem Knoten im IML-Graph muss also die Stelle im Quelltext, die durch den Knoten modelliert wird, auffindbar sein. Weiterhin muss die Darstellung von Java-Programmen in IML erweiterbar für eine Darstellung von C++-Programmen sein.

Die Erzeugung der IML-Darstellung eines Java-Programms, muss mit den Werkzeugen des Bauhaus-Projekts erfolgen. Werkzeuge, die verwendet werden müssen, sind der IML-Generator und die Implementierung für das Laden und das Speichern von IML-Graphen in Dateien. Der IML-Generator erzeugt

aus einer IML-Spezifikation eine IML-Implementierung. Mit Hilfe der IML-Implementierung kann ein IML-Graph erstellt werden. Zusätzlich können alle weiteren Bauhaus-Werkzeuge eingesetzt werden.

Da die IML Grundlage der Bauhaus-Werkzeuge und somit einer Vielzahl existierender Analyse-Programme ist, musste die Erweiterung der IML so erfolgen, dass an den Analyse-Programmen so wenig Änderungen wie möglich nötig sind. Änderungen bedeuten hier die Anpassung an die erweiterte IML-Darstellung für Java-Programme.

1.2.1 Einordnung der Diplomarbeit in Bauhaus

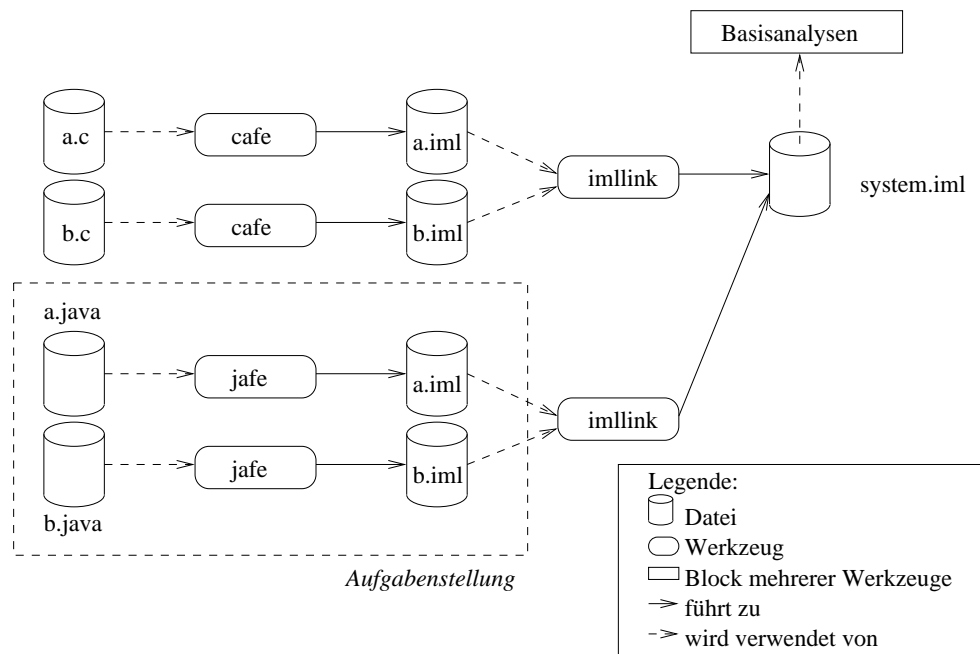


Abbildung 2: Struktur von Bauhaus mit Erweiterung durch die Diplomarbeit bis zu den Basisanalysen

In Abbildung 2 ist die Einordnung der Diplomarbeit in das Bauhaus-Projekt dargestellt. In der Diplomarbeit wurde ein Übersetzer entwickelt, der Java-Programme in eine IML-Darstellung übersetzt. Die Bestandteile der Diplomarbeit innerhalb des Bauhaus-Projekts sind in dem gestrichelt gezeichneten Kasten dargestellt. Der Übersetzer für Java-Programme nach IML trägt den Namen `jafe`. `Jafe` steht für `JAVA-FrontEnd`, in Anlehnung an den Namen `cafe` des C-Frontends.

Die einzelnen Arbeitsschritte und die Vorgehensweise für die Entwicklung von `jafe` sind in dieser Ausarbeitung beschrieben. Die Abfolge der Kapitel der Ausarbeitung orientiert sich an der Abfolge der Arbeitsschritte. Der Inhalt der folgenden Kapitel ist im folgenden Abschnitt kurz zusammengefasst.

1.3 Übersicht über das Dokument

Die schriftliche Ausarbeitung der Diplomarbeit ist wie folgt gegliedert:

Kapitel 2: beschreibt die Vorgehensweise bei der Auswahl eines geeigneten Java-Übersetzer-Frontend für die Implementierung von `jafe`.

Kapitel 3: beschreibt das Vorgehen bei der Erweiterung der IML für die Darstellung von Java-Programmen und die erweiterte IML für die Darstellung von Java-Programmen. Zusätzlich zur Beschreibung der erweiterten IML werden Beispiele für die Darstellung von Java-Programmen in der erweiterten IML vorgestellt.

Kapitel 4: beschreibt den Entwurf von `jafe` und die Besonderheiten bei der Implementierung des Java-Frontends.

Kapitel 5: beschreibt den Test, der durchgeführt wurde, um die Korrektheit der Implementierung des Java-Frontends zu überprüfen. Weiterhin werden in diesem Kapitel auch die Maßnahmen beschrieben, die unternommen wurden, um den Test zukünftiger Weiterentwicklungen zu unterstützen.

Kapitel 6: in diesem Kapitel werden die Auswirkungen der Erweiterungen der IML für Java-Programme auf die Bauhaus-Werkzeuge vorgestellt.

Kapitel 7: enthält einen Rückblick auf die Diplomarbeit, die durchgeführten Arbeiten und einen Ausblick auf die mögliche Zukunft des Java-nach-IML-Übersetzers.

Für das Verständnis der vorgestellten Kapitel werden an den Leser verschiedene Anforderungen gestellt. Die gestellten Anforderungen sind im folgenden Abschnitt beschrieben.

1.4 Anforderungen an den Leser

In den folgenden Kapiteln wird versucht, die einzelnen Grundlagen der diskutierten Fragestellungen und deren Lösung zu erläutern. Eine grundlegende Erläuterung ist jedoch nicht immer erschöpfend möglich oder im Rahmen dieses Dokuments sinnvoll. Aus diesem Grund werden die folgenden Anforderungen an den Leser gestellt.

Vom Leser dieses Dokuments wird erwartet, dass er das Bauhaus-Projekt kennt. Insbesondere sind Kenntnisse über die IML und die Darstellung von Programmen in abstrakten Syntaxbäumen nützlich. Für den Leser ist es hilfreich, wenn er sich in den Programmiersprachen C, C++ und Java auskennt. Es wird nicht erwartet, dass der Leser in den Programmiersprachen komplexe Programmieraufgaben lösen kann, jedoch sollten Syntax und Semantik der möglichen Anweisungen, Ausdrücke und Schlüsselworte bekannt sein.

Kapitel 2 Auswahl eines Java-Übersetzer-Frontends für die Erzeugung der IML

Für die Übersetzung von Java-Programmen in IML musste ein bestehendes Java-Übersetzer-Frontend verwendet werden. Das Java-Übersetzer-Frontend soll für ein Java-Programm einen abstrakten Syntaxbaum (Abstract Syntax Tree (AST)) erzeugen. Ausgehend von dem erzeugten AST wird dann die IML erzeugt.

Da der Java-Übersetzer, der für die Implementierung verwendet werden sollte, nicht vorgeschrieben war, wurde eine Auswahl unter verschiedenen Java-Übersetzern durchgeführt. Diese Auswahl wird in den folgenden Abschnitten beschrieben.

2.1 Anforderungen

Für die Auswahl eines geeigneten Java-Übersetzers wurde von mir eine Liste von Anforderungen erstellt. Jeder Java-Übersetzer, der als Ausgangspunkt für die Implementierung der IML-Erzeugung genutzt werden konnte, wurde über die Erfüllung der gestellten Anforderungen bewertet. Der Java-Übersetzer, der in den Bewertungen am besten abschnitt, wurde als Ausgangspunkt für die Implementierung der IML-Erzeugung verwendet.

Die einzelnen Anforderungen sind wie folgt:

- Es soll möglich sein, das Übersetzer-Frontend, bestehend aus Scanner, Lexer, Parser und semantischer Analyse, von den anderen Teilen des Übersetzers zu lösen. Für die Erzeugung der IML wird nur eine vollständig gefüllte Symboltabelle und der erzeugte AST benötigt.
- Auf den vom Übersetzer erzeugten AST muss ein Zugriff möglich sein. Das bedeutet, dass die im AST gespeicherten Daten von außen zugreifbar

sein müssen. Der Zugriff muss von der Programmiersprache Ada95 aus möglich sein.

- Der syntaktische Aufbau des Programms muss im AST unverändert dargestellt sein.
- Eine Rückverfolgung vom AST in den zu Grunde liegenden Quelltext muss möglich sein.
- Die Symboltabelle, die vom Übersetzer aufgebaut wird, muss von außen zugreifbar sein. Auch hier, wie beim AST, muss auf die Datenstrukturen der Symboltabelle von Ada95 aus zugegriffen werden können.
- In der erzeugten Symboltabelle sollen alle Informationen über Klassen, deren Methoden und Attribute vorhanden sein.
- Die Bedingungen, unter welchen der verwendete Java-Übersetzer lizenziert ist, müssen mit der Lizenzierung der Bauhaus-Werkzeuge vereinbar sein.
- Der Java-Übersetzer muss den aktuellen Stand der Java Sprachspezifikation implementieren¹.
- Die Anpassung der Implementierung für die Erzeugung der IML an eine neue Version des verwendeten Java-Übersetzers soll möglichst einfach sein.
- Die Zukunft des verwendeten Java-Übersetzer muss gesichert sein. Das bedeutet, dass bei Änderungen der Java-Spezifikationen eine Anpassung des Java-Übersetzers durch dessen Entwickler zumindest mittelfristig gesichert sein soll.

Basierend auf den angeführten Anforderungen wurde nach passenden Java-Übersetzern oder Java-Übersetzer-Frontends gesucht. Das Ergebnis der Suche waren zwei Java-Übersetzer:

- der GNU Compiler für Java (`gcj`) [Link: [gcj](#)] als Teil der GNU Compiler Collection in der Version 3.0.3 und der
- IBM Jikes Java Compiler (`jikes`) in der Version 1.15 [Link: [jikes](#)].

Sowohl `jikes` als auch `gcj` sind vollwertige Java-Übersetzer, die ein Java-Programm in Java-Bytecode übersetzen.

Für die beiden Java-Übersetzer, `jikes` und `gcj`, wurde eine Evaluation zur Überprüfung der gestellten Anforderungen durchgeführt. Die Evaluation und deren Ergebnisse sind in den folgenden Abschnitten dargestellt.

2.2 Evaluation der Java-Übersetzer

Für die Evaluation wurden verschiedene Messungen und Bewertungen der Java-Übersetzer vorgenommen. Jede Messung oder Bewertung hatte zum Ziel, die Erfüllung einer oder mehrerer der gestellten Anforderungen zu prüfen.

1. Die Sprachspezifikation ist in [Gosling et al. 2000] beschrieben. Sie basiert auf dem Stand von Java 1.2. Mit Java 1.2 wurde zuletzt das Schlüsselwort (`strictfp`) eingeführt. Mit dem Erscheinen von Java 1.4 im Februar 2002 wurde ein weiteres, neues Schlüsselwort (`assert`) eingeführt. Die Erfüllung der Sprachspezifikation für Java 1.4 wurde nicht berücksichtigt, da die Auswahl des Java-Übersetzers ebenfalls im Februar 2002 stattfand. Die Implementierung des neuen Standards wurde von den untersuchten Java-Übersetzern nicht erwartet.

Gemessen wurde die korrekte Implementierung der Java Sprachspezifikation, durch den untersuchten Übersetzer. Bewertet wurden Lizenzen, Implementierung, Zugriff und Vollständigkeit auf den erzeugten AST und Symboltabelle, Aufwand für die Anpassung der IML-Erzeugung an eine neue Version des untersuchten Übersetzers, Zugriff von Ada95 und die Zukunftssicherheit des Java-Übersetzers.

2.2.1 Implementierung der Java-Sprachspezifikation

Die korrekte Implementierung der Java-Spezifikation wurde mit Hilfe der `jacks`-Regressionstestsuite (Jacks Automated Compiler Killing Suite [Link: `jacks`]) von IBM überprüft.

Die `jacks`-Testsuite bietet für alle Abschnitte der Java-Sprachspezifikation Testfälle. Die Testfälle werden von dem zu überprüfenden Übersetzer übersetzt und auf einer virtuellen Maschine für Java überprüft. Die Regressionstestsuite enthält Konfigurationen für die geprüften Übersetzer `jikes` und `gcj`. Die Konfigurationen für die einzelnen Übersetzer bestimmen die Testfälle, die für einen speziellen Übersetzer ausgeführt werden oder nicht. Dies sind zum Beispiel Testfälle für das Schlüsselwort `assert`, das erst mit Java 1.4 eingeführt wurde, oder Testfälle, die Eigenschaften eines speziellen Übersetzers testen.

Die virtuelle Maschine, die für die Prüfung der übersetzten Programme verwendet wird, und der geprüfte Übersetzer sind unabhängig voneinander. Gemäß der Spezifikation der virtuellen Maschinen für Java [Lindholm, Yelling 1999][Link: JVM-Specification], müssen alle Übersetzer Bytecode erzeugen, der auf einer virtuellen Maschine gemäß der Spezifikation lauffähig ist.

Geprüft wurden der `gcj` und `jikes`. Als Referenz wurde der Java-Übersetzer von Sun Microsystems Inc. in der Version 1.3.3 zusätzlich geprüft [Link: `javac`]. Als virtuelle Maschine für Java-Bytecode wurde die virtuelle Maschine von Sun Microsystems Inc. in der Version 1.3.3 für alle drei geprüften Übersetzer verwendet.

Die Ergebnisse der Ausführung der `jacks`-Regressionstestsuite für Java-Übersetzer sind in Tabelle 1 dargestellt.

Tabelle 1: Ergebnis der Jacks-Regressionstestsuite für Java Übersetzer

Übersetzer	Testfälle Gesamt	Nicht Durchgeführte Tests	Bestandene Tests	Nicht Bestandene Tests
<code>javac</code>	3267	119	2908	240
<code>jikes</code>	3267	103	2932	232
<code>gcj</code>	3267	(639)	(2207)	(421)

Die Ergebnisse für den `gcj` sind in Klammern angeführt, da manche Testläufe mit dem `gcj` teilweise zu verschiedenen Ergebnissen führten. Eine Nachmessung mit der `jacks`-Regressionstestsuite in der Version vom 3. August 2002 bestätigte die wechselnden Ergebnisse.

Von der `jacks`-Regressionstestsuite wurden nicht alle Testfälle für die geprüften Java-Übersetzer ausgeführt. In der Regressionstestsuite sind beispielsweise Testfälle speziell für den `javac`-Übersetzer enthalten, die für

`jikes` und den `gcj` nicht ausgeführt werden. Ebenso sind Testfälle enthalten, die einen Test der `assert`-Anweisung durchführen. Diese Testfälle werden ebenfalls nicht ausgeführt.

Um die Ergebnisse vergleichen zu können, wurde die Prozentzahl korrekt ausgeführter Testfälle der tatsächlich ausgeführten Testfälle errechnet. Alle Testfälle, die nicht ausgeführt worden sind, wurden nicht weiter betrachtet. Die Berechnung ist in Gleichung 1 dargestellt. Die Resultate sind in Tabelle 2 zusammengefasst.

$$P_p = \frac{T_p}{(T - T_s)} \cdot 100$$

Gleichung 1: Prozentualer Anteil korrekter Testfallausführung

Die Bedeutung der Symbole in Gleichung 1 ist wie folgt:

- P_p : Prozentzahl korrekt ausgeführter Testfälle ohne nicht durchgeführte Testfälle,
- T : Gesamtanzahl aller Testfälle,
- T_p : Anzahl der korrekt ausgeführten Testfälle und
- T_s : Anzahl der nicht ausgeführten Testfälle.

Tabelle 2: Korrekt ausgeführte Testfälle ohne übersprungene Testfälle

	<code>javac</code>	<code>jikes</code>	<code>gcj</code>
P_p	92,38%	92,67%	(83,98)

2.2.2 Lizenzen

Die Auswertung der Lizenzierung der Java-Übersetzer beschränkte sich auf die Betrachtung der Verträglichkeit mit der Lizenzierung der Bauhaus-Werkzeuge. Eine ausführliche Auseinandersetzung mit den einzelnen Lizenzen wurde nicht durchgeführt, da hierfür tieferes juristisches Wissen notwendig wäre.

Die Lizenzen von `gcj` und `jikes` sind beides Open-Source-Lizenzen. Hierdurch ist gewährleistet, dass der Quelltext der Übersetzer für die IML-Erzeugung verwendet werden kann.

Die Unterschiede zwischen den beiden Lizenzen, der GNU Public License und der IBM Public License, und deren Auswirkungen auf die Bauhaus-Werkzeuge werden im Folgenden diskutiert.

GNU Compiler für Java

Der `gcj` ist unter der GNU General Public License (GPL) [FSF 1991][Link: GPL] lizenziert. Wird Quelltext aus einem GPL lizenzierten Programm verwendet, muss der verwendende Quelltext ebenfalls mit der GPL lizenziert werden. Ausschlaggebend hierfür ist Absatz 2.b) in der GPL:

“You must cause any work that you distribute or publish (Ed.: This would be `jafe`), that in whole or in part contains or is derived from the Program (Ed.: Sources used licensed under the

GPL) or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.”

[FSF 1991], Abschnitt 2.b)

Die Auswirkungen der GPL auf den IML-Generator, die IML-Implementierung, die Bauhaus-Werkzeuge und das Java-Frontend für IML werden im Folgenden einzeln betrachtet.

Auf das Java-Frontend wirkt sich die GPL insofern aus, als dass das Java-Frontend ebenfalls durch die GPL lizenziert werden muss. Dies liegt daran, dass im Java-Frontend GPL lizenzierter Quelltext verwendet wird.

Eine Alternative ist das Java-Frontend als Bibliothek die vom `gcj` genutzt wird zu entwerfen. Dann kann das Java-Frontend durch die GNU Lesser Public License (LGPL) [FSF 1999][Link: LGPL] lizenziert werden. Die LGPL verlangt, ebenso wie die GPL, dass der Quelltext der Implementierung zugänglich ist. Im Unterschied zur GPL muss ein Programm, das eine LGPL lizenzierte Bibliothek verwendet, nicht zwangsweise unter der LGPL oder der GPL lizenziert werden.

Die Implementierung der IML kann durch die GPL oder die LGPL lizenziert werden. Eine Lizenzierung durch die GPL oder die LGPL ist mit der Lizenzierung des Java-Frontends und des `gcj` verträglich. Unter beiden Lizenzen muss der Quelltext verfügbar sein. Bei der LGPL muss die IML-Implementierung als Bibliothek realisiert sein.

Eine weitere Möglichkeit ist die Lizenzierung der IML-Implementierung durch zwei Lizenzen. Diese Doppellizierung ist nur möglich, da es sich bei der IML-Implementierung um “eigenen” Quelltext handelt. Die IML-Implementierung für das Java-Frontend wird durch die GPL oder die LGPL lizenziert. Eine zweite IML-Implementierung, die von den Bauhaus-Werkzeugen benutzt wird, ist durch eine beliebige Lizenz lizenziert. Beide IML-Implementierungen, für den IML-Generator und die Bauhaus-Werkzeuge, können gleich sein.

Eine Auswirkung der GPL auf die Bauhaus-Werkzeuge findet nur statt, wenn die IML-Implementierung ausschließlich durch die GPL lizenziert wird. Dann müssen auch alle Bauhaus-Werkzeuge, welche die IML-Implementierung verwenden, durch die GPL lizenziert werden. Ist die IML-Implementierung durch die LGPL lizenziert, wirkt sich dies nicht auf die Lizenzierung der Bauhaus-Werkzeuge aus.

Jikes IBM Java-Compiler

Der `jikes` Java-Übersetzer ist unter der IBM Public License (IPL) [IBM 1999][Link: IPL] lizenziert. Wird unter der IPL lizenzierter Quelltext in eigenem Quelltext eingesetzt, so darf der eigene Quelltext unter einer “beliebigen” Lizenz lizenziert werden. Beliebig bedeutet, dass der verwendete Quelltext weiterhin unter der IPL lizenziert bleibt und der Quelltext für das gesamte Programm verfügbar ist.

Wird IPL-lizenzierter Quelltext für die Implementierung des Java-Frontend verwendet, bedeutet dies, dass die Bauhaus-Werkzeuge ihre eigene Lizenz weiter verwenden können. Der Quelltext für das Java-Frontend und die IML Implementierung muss zugänglich sein.

2.2.3 Implementierung

Die Implementierung des `gcj` ist in C ausgeführt. Die Implementierung des AST verwendet Makros. Im AST des `gcj` ist die Symboltabelle für das

übersetzte Programm mit abgelegt. Da der `gcj` ein Frontend für Java der GNU Compiler Collection darstellt, greift die Implementierung des `gcj` auf die Implementierung der GNU Compiler Collection zurück.

Der `jikes` Übersetzer ist in C++ implementiert. Die Knoten des AST sind durch C++ Klassen repräsentiert. Die Symboltabelle ist getrennt vom AST implementiert. Eine Verbindung zwischen AST und Symboltabelle ist durch Attribute der Symboltabelleneinträge und durch Attribute der AST-Klassen realisiert. So verweist zum Beispiel eine Klasse in der Symboltabelle auf den Knoten im AST, der ihre Definition repräsentiert.

Die Implementierung von `jikes` ist verständlich und ausreichend dokumentiert. Eine Einarbeitung ist in kurzer Zeit möglich. Für eine Einarbeitung in die Implementierung des AST im `gcj` ist ein höherer Aufwand erforderlich. Bei einer Einarbeitung muss nicht nur die Implementierung des Java-Übersetzer-Frontends berücksichtigt werden, sondern auch die durch den `gcj` verwendete Implementierung der GNU Compiler Collection.

2.2.4 Zugriff und Vollständigkeit auf erzeugten AST und Symboltabelle

Eine tiefgehende Untersuchung des AST und der Symboltabelle wurde nur für den `jikes` Java-Übersetzer durchgeführt. Auf eine Einarbeitung in den `gcj` wurde auf Grund der vorhergehenden Ergebnisse der Evaluation verzichtet. Es zeigte sich, dass `jikes` bei Lizenz, Korrektheit und Verständlichkeit der Implementierung gegenüber dem `gcj`, die bessere Alternative darstellt.

Die Struktur des AST ist durch eine Auswahl von Java Testprogrammen ermittelt worden. Es sind insgesamt 130 kleine Java-Programme erstellt worden, mit denen eine Überdeckung der syntaktischen Sprachmittel gemäß der Java-Sprachspezifikation [Gosling et al. 2000] erreicht wurde. Für jedes Java-Programm wurde der erzeugte AST untersucht. Ergebnis der Untersuchung war, dass im AST von `jikes` alle Programme korrekt abgebildet werden. Auf eine Auflistung der einzelnen Programme, und den für jedes Programm erzeugten AST, wird an dieser Stelle verzichtet.

Die Symboltabelle von `jikes` wurde nicht näher untersucht, da der AST als ausreichend erachtet wurde. Diese Annahme stellte sich jedoch während der Implementierung als falsch heraus. Glücklicherweise enthält die Symboltabelle von `jikes` ausreichend Informationen über alle verwendeten Klassen, deren Methoden und Attribute, wie später festgestellt wurde.

2.2.5 Aufwand für die Anpassung der IML Erzeugung

Für diese Untersuchung wurde aus Gründen, die bereits in Abschnitt 2.2.4 erläutert wurden, nur der `jikes` Java-Übersetzer untersucht.

Die Untersuchung des Programmablaufs von `jikes` ergab sehr schnell eine Stelle im Quellcode, an der die Implementierung für die Erzeugung der IML eingefügt werden kann. Ein Prototyp, der den AST von `jikes` ausgibt, wurde als Bestätigung der Einfügestelle und der Zugreifbarkeit auf den AST implementiert.

Ein Prototyp für die Erzeugung der IML konnte nicht erstellt werden. Dies liegt darin begründet, dass zu der Zeit der Auswahl des Java-Übersetzer-Frontends, ein Zugriff auf die IML-Implementierung von C++ aus nicht möglich war. Da der `jikes` Java-Übersetzer in C++ implementiert ist, wäre eine solche Schnittstelle notwendig gewesen.

Auf die Implementierung der IML-Erzeugung und der Schnittstelle für den Zugriff auf die IML-Erzeugung von C++, wird eingehender in Kapitel 4 eingegangen.

2.2.6 Zugriff von Ada95

Ein Zugriff aus Ada95 auf den AST, als auch auf die Symboltabelle beider untersuchten Übersetzer, ist möglich.

Der `gcj` erlaubt den Zugriff von Ada95 auf seine Symboltabelle und den AST über die für Ada95 definierte C-Schnittstelle.

Zugriff auf AST und Symboltabelle von `jikes`, der in C++ implementiert ist, erlaubt die Schnittstelle für C++, die für den `gnat` Ada95 Übersetzer entwickelt wurde [Comar, Dewar 1996]. Eine Schnittstelle zwischen Ada95 und C++ ist allerdings nicht Teil des Ada95 Standards [ISO/IEC 1995].

2.2.7 Zukunftssicherheit

Die Zukunftssicherheit ist bei beiden untersuchten Java-Übersetzern gewährleistet.

Der `gcj` als Teil der GNU Compiler Collection wird im Zuge der Weiterentwicklung der GNU Compiler Collection gepflegt. Seit der Untersuchung des `gcj` in der Version 3.0.3, ist der aktuelle Versionsstand der GNU Compiler Collection auf 3.1.1 vorgerückt.

`Jikes` ist Teil des IBM Open Source Programms. Er wird von einer Entwicklergemeinschaft als Open Source weiterentwickelt und ist nicht auf eine Entwicklung durch die IBM beschränkt. Seit Untersuchung der Version 1.15 im Februar 2002 ist Version 1.16 mit Unterstützung des `assert` Schlüsselwort im Juli 2002 erschienen. Anforderungen an die Implementierung der Version 1.17 werden in der Entwicklergemeinschaft diskutiert.

2.3 Auswahl des Java-Übersetzers

Das zu verwendende Java-Frontend, für die Implementierung der IML-Erzeugung für Java-Programme, wurde gemeinschaftlich, auf Basis der vorgestellten Untersuchungen von `gcj` und `jikes`, ausgewählt.

Die Wahl von `jikes` begründet sich in den besseren Ergebnissen bei der Ausführung der `jacks` Regressionstestsuite gegenüber `gcj`, einer Lizenzierung die sich nur in soweit auf die Bauhaus-Werkzeuge auswirkt, als dass der Quelltext verfügbar sein muss. Außerdem ist, meiner Meinung nach, die Implementierung von `jikes` leichter verständlich. Die Untersuchung des AST und der notwendigen Erweiterungen des Übersetzers für die Erzeugung der IML wurden nicht als Auswahlkriterium herangezogen, da diese Punkte nur für `jikes` untersucht wurden. In dem Punkt der Zukunftssicherheit unterscheiden sich beide Java-Übersetzer nicht. Ein Zugriff von Ada95 auf die Implementierung des AST und der Symboltabelle ist ebenfalls für beide Übersetzer möglich.

Nach der Auswahl des Java-Übersetzers, der für die Erzeugung der IML verwendet wird, wurde die IML für die Repräsentation von Java-Programmen erweitert. Die Erweiterung der IML ist im folgenden Kapitel beschrieben.

Kapitel 3 Erweiterung und Spezifikation der IML für Java

In den folgenden Abschnitten wird zunächst das Vorgehen für die Erweiterung der IML für Java vorgestellt. Im Anschluss daran wird die erweiterte IML für die Darstellung von Java-Programmen vorgestellt.

3.1 Vorgehen für die Erweiterung der IML

In diesem Abschnitt wird das Vorgehen für die Erweiterung der IML beschrieben. Die Erweiterung der IML für Java-Programme bestand zunächst aus der Untersuchung der Struktur der aktuellen IML. Basierend auf der Untersuchung der aktuellen Struktur, wurde die IML entsprechend den Entwurfszielen der IML und der langfristigen Planung der Erweiterung der IML für Java-Programme erweitert.

Die Erweiterungen der IML für Java und Beispiele zur Illustration der Modellierung von Java Programmen mit der erweiterten IML sind in Kapitel 3 aufgeführt.

3.1.1 Erweiterung der IML

Die IML wurde mit dem Ziel der Darstellung von Java-Programmen erweitert. Für die Erweiterung musste zunächst die aktuelle Struktur der IML ermittelt werden. Konzepte der IML und eine Spezifikation der IML für die Darstellung von C-Programmen sind in [Rohrbach 1998] beschrieben. Für die aktuelle Struktur der IML existiert nur die Datei mit der Spezifikation der IML für den IML-Generator. Der IML-Generator erzeugt aus der IML-Spezifikation Ada95-Quelltext, mit dem IML-Repräsentationen von Programmen erzeugt werden können.

Bedingt durch die Unterschiede der aktuellen IML-Version (Version 1.172) und der in [Rohrbach 1998] spezifizierten IML, wurde die Spezifikation der aktuellen IML für die Erarbeitung der Struktur der IML ausgewertet. [Rohrbach 1998] wurde für das Verständnis der Konzepte der IML genutzt.

Auswertung der aktuellen IML-Struktur

Die grundlegende Struktur der IML, Trennung der Klassen für Objekte, Typen und Hierarchischer-Programm-Graph (HPG) aus dem Entwurf von Jürgen Rohrbach blieb erhalten. Ebenfalls die Verknüpfung der Klassen durch syntaktische und semantische Kanten. Die größten Änderungen haben im Bereich der Attribute der einzelnen Knotentypen und deren Anordnung in der Vererbungshierarchie stattgefunden.

Aus der Untersuchung der IML-Spezifikation wurde die Anordnung der Klassen in der Vererbungshierarchie der IML ermittelt. Aus der Spezifikation konnten ebenfalls die Attribute der Knotentypen ermittelt werden. Für die Ermittlung der Darstellung von C-Programmen in IML wurden 38 kleine C-Programme in IML übersetzt und deren Darstellung untersucht.

Die 38 C-Programme, die für die Untersuchung der Repräsentation von C-Programmen in der IML verwendet wurden, deckten die syntaktischen Möglichkeiten der Programmiersprache C ab. Um alle Eigenschaften und syntaktischen Möglichkeiten der Programmiersprache C abzudecken, wurde [Kernighan, Ritchie 1990] als Leitfaden verwendet.

Nach Untersuchung der Vererbungshierarchie der IML-Klassen und der Darstellung von C-Programmen in der aktuellen IML konnte mit der Erweiterung der IML für Java-Programme begonnen werden.

Erweiterung der IML für Java-Programme

Bei der Erweiterung der IML für Java musste darauf geachtet werden, dass

1. C-Programme weiterhin darstellbar sind,
2. bestehende Übersetzer für die Übersetzung von Programmiersprachen nach IML geringer oder keiner Änderung bedürfen,
3. bestehende Analysen basierend auf der IML geringer oder keiner Änderung bedürfen und
4. die IML für Java-Programme soll die Modellierung von C++-Programmen unterstützen.

Ausgehend von diesen Anforderungen ergaben sich die folgenden Richtlinien für die Erweiterung der IML für Java:

- Kein Verändern oder Entfernen bestehender Attribute von Klassen.
Diese Anforderung ist notwendig, um Änderungen an bestehenden Übersetzern für Programmiersprachen nach IML zu vermeiden. Die Übersetzer greifen beim Aufbau der IML-Repräsentation eines Programms auf die Attribute der Klassen zu.
Analysewerkzeuge, die auf der IML aufbauen, greifen ebenfalls auf die Attribute der Klassen zu.
Werden Attribute verändert oder entfernt, entsteht Anpassungsaufwand in den bestehenden Übersetzer-Frontends und in den Analysewerkzeugen, da diese auf die veränderten Attribute angepasst werden müssen.
- Keine Veränderung der Vererbungsstruktur der IML, die auf ererbte Attribute von Klassen Auswirkungen hat.
Diese Anforderung ergibt sich aus dem vorigen Punkt und ist gleich begründet. Durch die Anordnung der Klassen in der Vererbungsstruktur, erben die Klassen der IML Attribute ihrer Oberklassen. Auf die ererbten

Attribute greifen wiederum Übersetzer und Analysewerkzeuge zu. Ein weiterer Punkt ist, dass durch die Vererbungsstruktur in bestehenden Übersetzern oder Analysewerkzeugen mit Oberklassen gearbeitet werden kann. Dies bedeutet, dass ein Algorithmus zum Beispiel für eine Oberklasse implementiert wird. Der Algorithmus ist dann für alle Unterklassen dieser Oberklasse gültig, sofern die Vererbung eine Spezialisierung mit Erhaltung der Semantik der Schnittstelle (Attribute) darstellt und jede Oberklasse durch ihre Unterklasse ersetzt werden kann (Liskovsches Substitutions-Prinzip [Liskov 1988]).

Durch eine Veränderung der Vererbungshierarchie können Oberklassen für Algorithmen nicht mehr geeignete Unterklassen enthalten.

- Erweiterungen der IML werden für Java und für C++ gemeinsam entworfen. Dies ist hilfreich, um frühzeitig erkennen zu können, ob eine Spezifikation der IML für C++ mit der Spezifikation für Java verträglich ist.

Die IML wurde für die Darstellung von Java-Programmen erweitert. Für Java-Programme notwendige Attribute wurden bestehenden Knotentypen hinzugefügt. Neue Knotentypen, die für die Darstellung von Java-Programmen notwendig sind, wurden in die bestehende Vererbungsstruktur integriert oder die Vererbungsstruktur wurde erweitert.

Bei der Erweiterung der Vererbungshierarchie wurden neue Knotentypen hinzugefügt oder neue, gruppierende Knotentypen in die Vererbungshierarchie eingefügt. Für den Fall, dass die Vererbungshierarchie erweitert wurde, wurde darauf geachtet, dass die ursprüngliche Schnittstelle und deren Semantik erhalten blieb.

Die Erweiterungen der IML für Java werden im folgenden Abschnitt detailliert vorgestellt.

3.2 Spezifikation der IML für Java

Dieser Abschnitt stellt die Erweiterungen der IML für Java vor. Die Vorstellung der Erweiterungen ist thematisch gegliedert und wird durch Beispiele erläutert. Als erstes werden Übersetzungseinheiten vorgestellt, dann Klassen und Interfaces. Anschließend wird die Modellierung von Methoden vorgestellt. Nach der Vorstellung der Methoden folgt dann die Vorstellung der Modellierung der Ausnahmebehandlung und einzelner Operatoren, Anweisungen und Typen, die in der IML für C nicht enthalten sind.

In diesem Kapitel werden die Erweiterung der IML für Java vorgestellt. Alle IML-Knotentypen, die in der ursprünglichen Version der IML für C bereits enthalten sind, werden nicht vorgestellt. Die Modellierung von Operatoren, Anweisungen und Typen, die in C und Java gleich sind, werden nicht vorgestellt.

Bei der Vorstellung der Erweiterungen der IML werden IML-Knotentypen/IML-Klassen als Knotentyp bezeichnet. Instanzen von Knotentypen werden als Knoten bezeichnet. Java-Klassen werden als Klassen bezeichnet.

3.2.1 Spezifikation der Erweiterungen

Die folgenden Abschnitte beschreiben die Erweiterungen der IML und erläutern diese durch Beispiele. Die verwendete Notation ist UML und kann [OMG 2001][Link: UML-Spezifikation] entnommen werden.

Die Stereotypen der Kanten der UML wurden für die Darstellung von Kanten im IML-Graph angepasst. Syntaktische Kanten werden durch durchgezogene Linien dargestellt, semantische Kanten durch gestrichelte Linien. Der Unterschied zwischen semantischen und syntaktischen Kanten wurde bereits in Abschnitt 1.1.1 vorgestellt. Stereotypen an den Kanten geben an, welches Attribut des IML-Knotentyp, von dem die Kante ausgeht, repräsentiert wird. Die in den Zeichnungen enthaltenen UML-Kommentare mit Nummern, sind eine Referenz in den Beschreibungstext für das Beispiel.

Bei der Beschreibung der einzelnen IML-Knoten und deren Attribute wurde folgende Notation verwendet:

- Der Name des beschriebenen Knotentyp ist **fett** gedruckt.
- Attribute, die eine syntaktische Kante darstellen, sind in nicht kursiven Zeichen gedruckt.
- Attribute, die eine semantische Kante darstellen, sind in *kursiven* Zeichen gedruckt.
- Attribute und Knotentypen, die im Zuge der IML-Erweiterung für Java hinzukamen sind grau hinterlegt. Die graue Hinterlegung für neue Knotentypen wird auch in der Darstellung der Knotentyphierarchie verwendet.

3.2.2 Übersetzungseinheiten

Eine Übersetzungseinheit ist in Java eine Datei mit allen darin enthaltenen Typdefinitionen. Für jede Übersetzungseinheit kann das Paket definiert werden, in dem die definierten Typen enthalten sind. In einer Übersetzungseinheit können Typen oder Pakete importiert werden. Durch das Importieren sind die Typen oder Pakete in der Übersetzungseinheit sichtbar [Gosling et al. 2000][Link: Java Language Specification].

Bei der Übersetzung einer Übersetzungseinheit durch einen Java-Compiler wird für jeden in der Übersetzungseinheit definierten Typ eine eigene Datei mit Bytecode erzeugt. Für die Erzeugung der IML-Darstellung für Java-Programme wäre es naheliegend, für jeden in einer Übersetzungseinheit definierten Typ eine IML-Datei zu erzeugen. Dieser Ansatz wurde allerdings nicht implementiert.

Für alle Typen in einer Übersetzungseinheit, die nicht innere oder anonyme innere Typen sind, wird ein eigener IML-Graph erzeugt. Innere und anonyme innere Typen werden im IML-Graph des umschließenden Typs modelliert. Jeder IML-Graph wird in eigene IML-Datei gespeichert. Dieser Ansatz wurde gewählt, da `jikes` die einzelnen in IML zu übersetzenden Typen einzeln an `jafe` liefert. Wird ein Typ an `jafe` zur Übersetzung nach IML übergeben, liegen Informationen über alle inneren Typen vor. Information über Typen, die auf der gleichen Schachtelungstiefe wie der aktuelle bearbeitete Typ liegen, sind nicht direkt abrufbar. Aus Effizienzgründen und um die Arbeit des IML-Linkers zu vereinfachen, wird für einen Typ und alle seine eingeschachtelten Typen ein IML-Graph erzeugt.

Knotentyphierarchie

Die Knotentypen für die Abbildung von Übersetzungseinheiten sind in Abbildung 3 dargestellt. Für Übersetzungseinheiten wird nur der Knotentyp `Unit` verwendet. In den Übersichtsdarstellungen der IML-Knotentyphierarchie wird immer der Vererbungsbaum bis zum Wurzelknoten dargestellt.

Eine ausführliche Erklärung der dargestellten Knotentypen ist im folgenden Abschnitt enthalten.

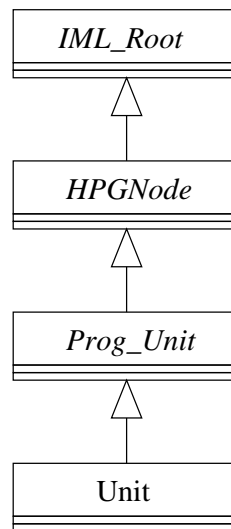


Abbildung 3: Knotentyphierarchie für Übersetzungseinheiten

Knotenbeschreibung

IML_Root	Wurzel der IML-Knotentyphierarchie. In der IML-Knotentyphierarchie sind alle Knotentypen enthalten, die in einem IML-Graph verwendet werden können.
<i>Parent: IML_Root</i>	Verweis auf den syntaktisch übergeordneten Knoten. Falls kein übergeordneter Knoten existiert, ist dieses Attribut auf 'null' gesetzt.
<i>Sloc: SLoc</i>	Position des abgebildeten, syntaktischen Programmelements im Quelltext.
HPGNode	Knotentyp für die Gruppierung aller Knotentypen, welche die syntaktische Struktur eines Programms abbilden. HPG ist ein Akronym für "Hierarchischer Programm Graph".
<i>CF_Next: HPGNode</i>	Verweis auf den nächsten Knoten im Kontrollfluss des Programms.

<i>CF_Previous: HPGNode</i>	Verweis auf den vorhergehenden Knoten im Kontrollfluss des Programms.
<i>CF_Basic_Block: HPGNode</i>	Verweis auf den “Basic-Block”, zu dem dieser Knoten gehört.
Prog_Unit	Knotentyp für die Gruppierung von Knotentypen, die Übersetzungseinheiten oder programmiersprachliche Gruppierungen modellieren. Programmiersprachliche Gruppierungen sind zum Beispiel Funktionen, Prozeduren, Klassen und Interfaces.
Subunits: list of Prog_Unit	Liste aller eingeschlossenen programmiersprachlichen Gruppierungen.
<i>Symbol_Table: set of OCNode</i>	Symboltabelle, die alle Objekte enthält, die in dieser Gruppierung definiert wurden.
Unit	Knotentyp für die Modellierung von Übersetzungseinheiten.
<i>Initialization_Code: Routine</i>	Verweis auf den Quellcode, der für die Initialisierung der modellierten Übersetzungseinheit definiert wurde.
<i>Declaration_Points: set of Declaration_Point</i>	Verweis auf alle Stellen im Quelltext, an denen ein Typ oder ein Objekt deklariert oder definiert wurde.
<i>Provided_Definitions: set of OC_Entity</i>	Verweis auf Objekte und Typen, die in der modellierten Übersetzungseinheit definiert wurden.
<i>Unresolved_Declarations: set of OC_Entity</i>	Verweis auf Objekte und Typen, die in der modellierten Übersetzungseinheit deklariert aber nicht definiert wurden.
<i>Global_Lifetime_Definitions: list of OC_Variable</i>	Verweis auf Objekte mit globaler Sichtbarkeit und Lebensdauer.
<i>Uses: list of Identifier</i>	Verweis auf die Namensräume, die von der modellierten Übersetzungseinheit importiert und sichtbar gemacht werden.

Beispiel

Der folgende Auszug aus einem Java-Programm soll die Darstellung von Übersetzungseinheiten in IML verdeutlichen. Ein Auszug aus dem IML-Graphen, der das Programm in IML darstellt, wird in Abbildung 4 dargestellt.

```

1  public class MyApp extends JFrame {
2      class WindowHandler
3          implements WindowListener {
4          ...
5      }
6      ...
7  }

```

Quelltext 1: Java Übersetzungseinheit

In Quelltext 1 wird in Zeile 1 die Klasse `MyApp` definiert. Diese Klasse definiert die Klasse `WindowHandler` als innere Klasse (Zeile 2). Die innere Klasse `WindowHandler` implementiert das Interface `WindowListener` (Zeile 3). Die Klasse `MyApp` erbt von der Klasse `JFrame` (Zeile 1).

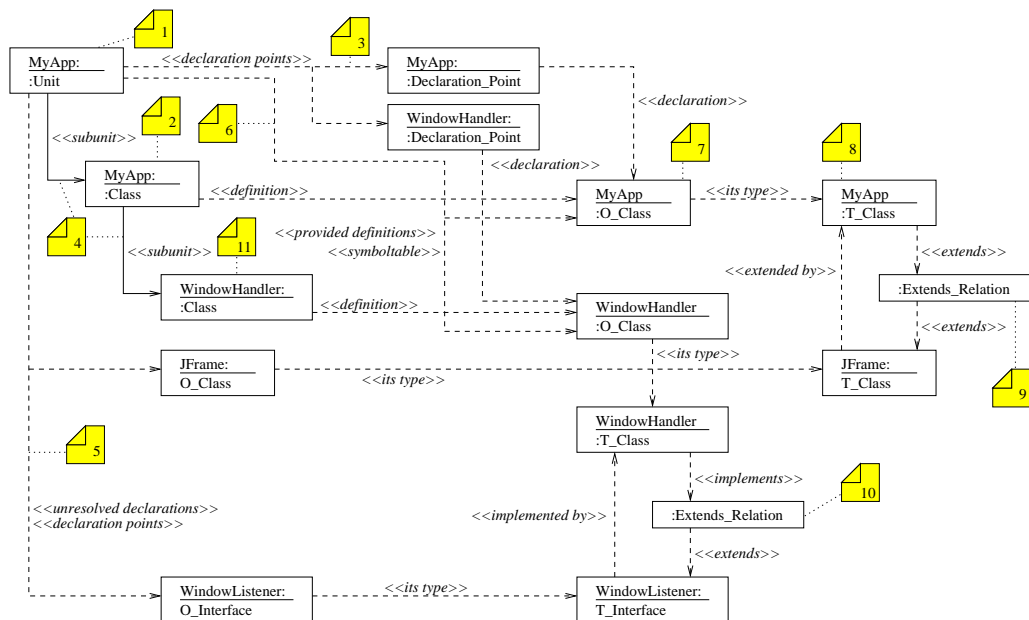


Abbildung 4: Modellierung von Übersetzungseinheiten, Klassen und Interfaces und Vererbungsbeziehungen in IML

Die folgenden Abschnitte beschreiben den in Abbildung 4 dargestellten IML-Graphen. Verweise zwischen dem dargestellten IML-Graph und der Beschreibung sind durch Nummern in Klammern angegeben. Die Nummern im Beschreibungstext verweisen auf UML-Kommentare, in denen die gleichen Nummern angegeben sind.

Der Unit-Knoten (1) ist die Wurzel des IML-Graphen und modelliert die Übersetzungseinheit. In der Übersetzungseinheit sind die beiden Klassen `MyApp` (2) und `WindowHandler` (11) als eingeschachtelte Einheiten (Subunits) (4) modelliert. `WindowHandler`, als innere Klasse von `MyApp`, ist

ein in der Klasse `MyApp` definierter Typ. Die Klasse `MyApp` ist ein in der Übersetzungseinheit definierter Typ.

Typen, Methoden oder Attribute, die in der Übersetzungseinheit definiert sind, erhalten einen Eintrag im Attribut `Symbol_Table` des `Unit`-Knoten und in dem Attribut `Provided_Definitions` (6). Typen, Methoden oder Attribute, die nicht in der Übersetzungseinheit definiert sind, erhalten einen Eintrag im Attribut `Unresolved_Declarations` (5).

Jeder Typ, der an einer beliebigen Stelle in der Übersetzungseinheit definiert oder verwendet wird, erhält einen Eintrag im Attribut `Declaration_Points` (3, 5). Das gleiche gilt für alle Attribute und Methoden eines Typs.

3.2.3 Klassen, Interfaces und Vererbung

Klassen und Interfaces werden in IML durch die Knotentypen `Class` und `Interface` modelliert. Zusätzlich zu den `Class`- und `Interface`-Knoten werden für jede Klasse `O_Class`- und `T_Class`-Knoten erzeugt. Für ein Interface wird ein `O_Interface`- und ein `T_Interface`-Knoten erzeugt.

Die `O_Class`- und `O_Interface`-Knotentypen modellieren das Objekt einer Klasse oder eines Interface. Das Objekt einer Klasse oder eines Interface enthält statische Methoden, Attribute und Konstruktoren für Objekte. Der Zustand einer Klasse oder eines Interface wird durch das Klassen- oder Interfaceobjekt repräsentiert.

Die Erzeugung einer Instanz einer Klasse wird durch Aufruf eines Konstruktors auf dem Klassenobjekt erzeugt. Eine Instanz einer Klasse wird durch einen `O_Instance`-Knoten modelliert.

Die Repräsentation des Zustands einer Klasse und die Instanzerzeugung über ein Klassen- oder Interfaceobjekt ist in Anlehnung an das Objektmodell von Smalltalk [Wallrabe, Oestereich 1997] entwickelt worden.

Die Knotentypen `T_Class` und `T_Interface` enthalten alle nicht statischen Methoden und Attribute. Über die `T_Class`- und `T_Interface`-Knotentypen werden die Vererbungs- und Implementierungsbeziehungen modelliert.

Vererbungs- und Implementierungsbeziehungen werden in zwei Richtungen modelliert: zum Einen von der implementierenden oder erbenden Klasse zur vererbenden Klasse oder dem implementierten Interface (im Vererbungsbaum von den Blättern aufwärts zur Wurzel); zum Anderen von der beerbten Klasse oder dem implementierten Interface zu der erbenden oder implementierenden Klasse (im Vererbungsbaum von der Wurzel zu den Blättern, abwärts). Die "Aufwärts"-Richtung der Vererbung, wird über `Extends_Relation`-Knoten modelliert. In einem `Extends_Relation`-Knoten ist es möglich, die Sichtbarkeit einer Vererbungsbeziehung abzubilden. In Java kann für eine Vererbungs- oder Implementierungsbeziehung keine Sichtbarkeit angegeben werden, in C++ ist dies jedoch möglich.

Arrays

Eine Spezialität in Java sind Arrays. Ein Array ist ein eigener Typ, von dem Instanzen erzeugt werden können. Der Arraytyp implementiert die beiden Interfaces `java.lang.Serializable` und `java.lang.Comparable`. Für jedes Objekt eines Arrays ist das spezielle Attribut `length` definiert. Über dieses Attribut kann die Anzahl der Elemente ermittelt werden,

die in dem Array gespeichert werden können. Um die Implementierungsbeziehungen und das spezielle Attribut `length` abbilden zu können, wurde in der IML der neue Knotentyp `TJ_Array` eingeführt. Für jeden Arraytyp wird ebenfalls ein Klassenobjekt angelegt. Wird ein Arraytyp instanziiert, wird ein `O_Instance`-Knoten für das Array erzeugt.

Die Knotentyphierarchie für die Modellierung von Klassen, Interfaces, Arrays und der Vererbung ist in Abbildung 5 dargestellt. Nach der Vorstellung der Knotentyphierarchie folgt eine detaillierte Beschreibung der einzelnen Knotentypen.

Knotentyphierarchie

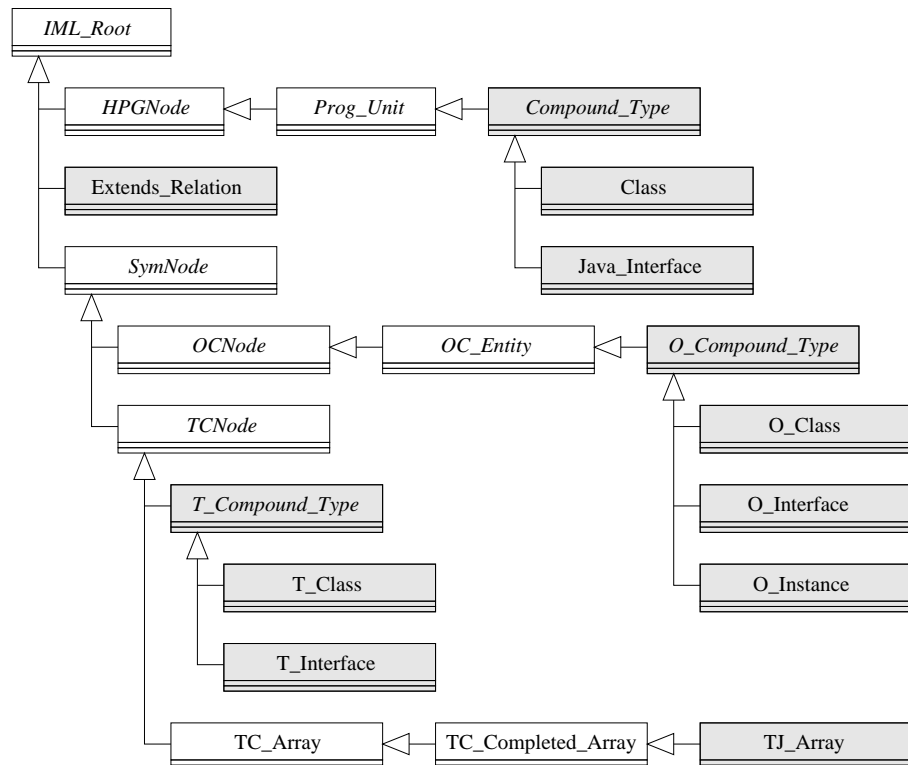


Abbildung 5: Knotentyphierarchie für die Modellierung von Klassen, Interfaces und Vererbung

Knotenbeschreibung

Extends_Relation	Knotentyp für die Modellierung einer Vererbungs- oder Implementierungsbeziehung ^a .
<i>Extends: T_Compound_Type</i>	Verweis auf den Typ, von dem der modellierte Typ erbt.
<i>Visibility_Declaration: Visibility</i>	Sichtbarkeit, die für die Vererbung definiert wurde.

a. Siehe hierzu auch Abschnitt 7.1.2.

Compound_Type	Knotentyp für die Gruppierung von Knotentypen, die Klassen und Interfaces modellieren. Klassen und Interfaces vereinen in einem Verbund Attribute und Methoden die auf den Attributen arbeiten.
Class	Knotentyp für die Modellierung einer Klasse.
<i>Definition: O_Class</i>	Verweis auf das Objekt, das die Definition der modellierten Klasse enthält.
Java_Interface	Knotentyp für die Modellierung eines Interface.
<i>Definition: O_Interface</i>	Verweis auf das Objekt, das die Definition dieses Interface enthält.
SymNode	Knotentyp für die Gruppierung von Knotentypen, deren Objekte in die Symboltabelle des IML-Graphen aufgenommen werden.
<i>Name: Identifier</i>	Name des modellierten Objekts oder Typs.
<i>Namespace_Name: Identifier</i>	Namensraum, in dem das Objekt oder der Typ definiert oder deklariert wurde.
<i>Visibility_Definition: Visibility</i>	Sichtbarkeit, die für das Objekt oder den Typ angegeben wurde.
OCNode	Knotentyp für die Gruppierung aller Knotentypen, die Objekte oder Deklarationen modellieren.
<i>Point_Of_Declaration: IML_Root</i>	Verweis in den HPG, an dem das Objekt definiert oder deklariert wurde.
<i>Its_Type: TCNode</i>	Verweis auf den Typ, aus dem dieses Objekt instanziiert wurde.

OC_Entity	Knotentyp für die Gruppierung von Knoten die Entitäten modellieren.
O_Compound_Type	Knotentyp für die Gruppierung von Objektknoten, die Klassen- oder Interfacetypen modellieren.
<i>Is_Constant: Boolean</i>	Zeigt an, ob dieses Objekt als konstant definiert wurde.
<i>Is_Anonymous: Boolean</i>	Zeigt an, ob dieses Objekt aus einem anonymen Typ erzeugt wurde.
O_Class	Knotentyp für die Modellierung von Klassenobjekten. Klassenobjekte stellen den Zustand einer Klasse dar.
<i>Class_Initializer: list of Routine</i>	Verweis auf die statischen Initialisierer, die für die modellierte Klasse definiert wurden.
<i>Block_Initializer: list of Routine</i>	Verweis auf die Block-Initialisierer, die für die modellierte Klasse definiert wurden.
<i>Class_Attributes: list of Declaration_Point</i>	Verweis auf die definierten Klassenvariablen.
<i>Class_Methods: list of Routine</i>	Verweis auf die definierten Klassenmethoden.
<i>Constructors: list of Constructor</i>	Verweis auf die definierten Konstruktoren.
O_Interface	Knotentyp für die Modellierung von Interfaceobjekten. Interfaceobjekte stellen den Zustand eines Interface dar. Ein Interface kann einen Zustand durch statische Variablen erhalten.
<i>Constants: list of Declaration_Point</i>	Verweis auf die Konstanten, die für das modellierte Interface definiert wurden.

O_Instance	Knotentyp für die Modellierung von Instanzen von Klassen. Eine Instanz einer Klasse wird durch Anwendung des <code>new</code> -Operators auf die Klasse erzeugt.
TCNode	Knotentyp für die Gruppierung von Knotentypen, die Typen abbilden.
<i>Type_Size: Natural</i>	Größe des abgebildeten Typs in Byte.
<i>Mangled_Name: Identifier</i>	“Mangled-Name” des modellierten Typs. Für Methodentypen ist dieser zum Beispiel aus dem Methodennamen, den Parametertypen und dem Rückgabotyp der Methode zusammengesetzt.
T_Compound_Type	Knotentyp für die Gruppierung von Knotentypen, die Java Klassen- und Interfacetypen modellieren.
<i>Is_Strictfp: Boolean</i>	Zeigt an, ob das Attribut ‘ <code>strictfp</code> ’ für den modellierten Typ angegeben wurde.
<i>Extends: list of Extends_Relation</i>	Verweis auf alle Typen, von denen der modellierte Typ erbt.
<i>Extended_By: set of TCNode</i>	Verweis auf alle Typen, die von dem modellierten Typ erben.
<i>Methods: list of Method</i>	Alle Methoden, die für den modellierten Typ definiert oder deklariert wurden.
<i>Nested_Types: set of T_Compound_Type</i>	Verweis auf alle Typen, die innerhalb des modellierten Typs definiert sind.
<i>Outer_Type: T_Compound_Type</i>	Verweis auf den umschließenden Typ, falls der modellierte Typ innerhalb eines anderen Typs definiert wurde.

T_Class	Knotentyp für die Modellierung von Klassen.
<i>Members: list of Declaration_Point</i>	Verweis auf alle Attribute, die für die modellierte Klasse definiert wurden.
<i>Finalizers: list of Destructor</i>	Verweis auf alle Destruktoren, die für die modellierte Klasse definiert wurden.
<i>Friends: list of Routine</i>	Verweis auf alle Routinen, die als 'friend' für diese Klasse deklariert wurden.
<i>Virtual_Call_Table: list of Routine</i>	Tabelle mit allen virtuellen Methoden, die in der modellierten Klasse deklariert, definiert oder von der Klasse geerbt wurden.
<i>Implements: list of Extends_Relation</i>	Verweis auf alle Interfaces, die von der modellierten Klasse implementiert werden.
<i>Is_Abstract: Boolean</i>	Zeigt an, ob die modellierte Klasse abstrakt ist.
<i>Is_Final: Boolean</i>	Zeigt an, ob für die modellierte Klasse das Attribut 'final' angegeben wurde.

T_Interface	Knotentyp für die Modellierung eines Interface.
<i>Implemented_By: set of TCNode</i>	Verweis auf alle Klassen, die das modellierte Interface implementieren.

TC_Array	Knotentyp für die Modellierung von C-Arrays.
<i>Elem_Type: TCNode</i>	Verweis auf den Typknoten, der den Typ der Arrayelemente modelliert.
<i>Upper_Bound: Integer_Constant</i>	Größe des Array.

TC_Completed_Array	Knotentyp für die Modellierung von C-Arrays, deren Größe nicht bekannt ist.
<i>Incomplete_Type: TC_Array</i>	Verweis auf den IML-Knoten, der das Array modelliert.
<i>Initialization: Initialize</i>	Ausdruck für die Berechnung der Größe des Arrays.

TJ_Array	Knoten für die Modellierung von Java-Arrays.
<i>Implements: list of Extends_Relation</i>	Verweis auf alle Interfaces, die von dem modellierten Arraytyp implementiert werden. Ein Array in Java implementiert immer die beiden Interfaces <code>Serializable</code> und <code>Comparable</code> .
<i>Extends: list of Extends_Relation</i>	Verweis auf alle Klassen, von denen der modellierte Arraytyp erbt. Ein Array in Java erbt immer von der Klasse <code>Object</code> .
<i>Methods: list of class Method</i>	Verweis auf alle Methoden, die auf den Arraytyp anwendbar sind. In Java sind dies alle Methoden aus der Klasse <code>Object</code> .
<i>Members: list of class Declaration_Point</i>	Verweis auf alle Attribute, die für den Arraytyp definiert sind. In Java ist dies das Attribut <code>length</code> , das die Größe des Arrays beschreibt.
<i>Virtual_Call_Table: list of class Routine</i>	Tabelle aller virtuellen Methoden, die für dieses Array definiert sind. Der Aufruf einer virtuellen Methode auf einem Objekt dieses Arraytyps wird mit Hilfe der Tabelle aufgelöst.

Beispiel

Für die Beschreibung der Darstellung von Klassen, Implementierungs- und Vererbungsbeziehungen in IML wird der in Quelltext 1 vorgestellte Programmausschnitt verwendet. Ein Ausschnitt aus dem IML-Graph ist in Abbildung 4 auf Seite 21 dargestellt.

Wie eingangs dieses Abschnitts beschrieben, werden Klassen durch einen `Class`-Knoten (2, 11), einen `O_Class`-Knoten (7) und einen `T_Class`-Knoten (8) modelliert.

Die Vererbungs- und Implementierungsbeziehungen werden durch `Extends_Relation`-Knoten modelliert. Die Vererbungsbeziehung zwischen der Klasse `MyApp` und `JFrame` ist durch (9) modelliert. Die Imple-

mentierungsbeziehung zwischen `WindowHandler` und `WindowListener` ist durch (10) modelliert. Die Knoten (9) und (10) modellieren die ‐Aufwärts‐-Richtung im Vererbungsgraph. In ‐Abwärts‐-Richtung wird eine Vererbungs- oder Implementierungsbeziehung durch die Attribute `extended_by` (Vererbung) oder `implemented_by` (Implementierung) modelliert.

3.2.4 Konstruktoren, Methoden und Destruktoren

Konstruktoren werden durch `Constructor`-Knoten modelliert. Statische Methoden werden durch einen `Routine`-Knoten modelliert, nicht statische Methoden durch einen `Method`-Knoten.

Für jeden `Routine`-, `Method`- oder `Constructor`-Knoten wird ein passender Objektknoten erzeugt. Objektknoten sind vom Typ `O_Constructor`, `O_Method` oder `OC_Routine`. Der Objektknoten verweist seinerseits auf den Typ der Methode.

Destruktoren existieren in Java nicht. Die spezielle Methode `finalize()`, die vom Garbage-Collector beim Entfernen eines Objekts aufgerufen wird, wird nicht gesondert in IML modelliert. Die Modellierung wird wie für normale Instanzmethoden durchgeführt. Für C++ ist es aber notwendig, Knotentypen für die Modellierung von Destruktoren anzubieten.

Die einzelnen verwendeten Knotentypen werden in den beiden folgenden Abschnitten detailliert vorgestellt. Nach Vorstellung der einzelnen Knotentypen, wird ein Ausschnitt aus einem IML-Graphen erläutert. Der erläuterte IML-Graph modelliert Methoden und Konstruktoren.

Knotentyphierarchie

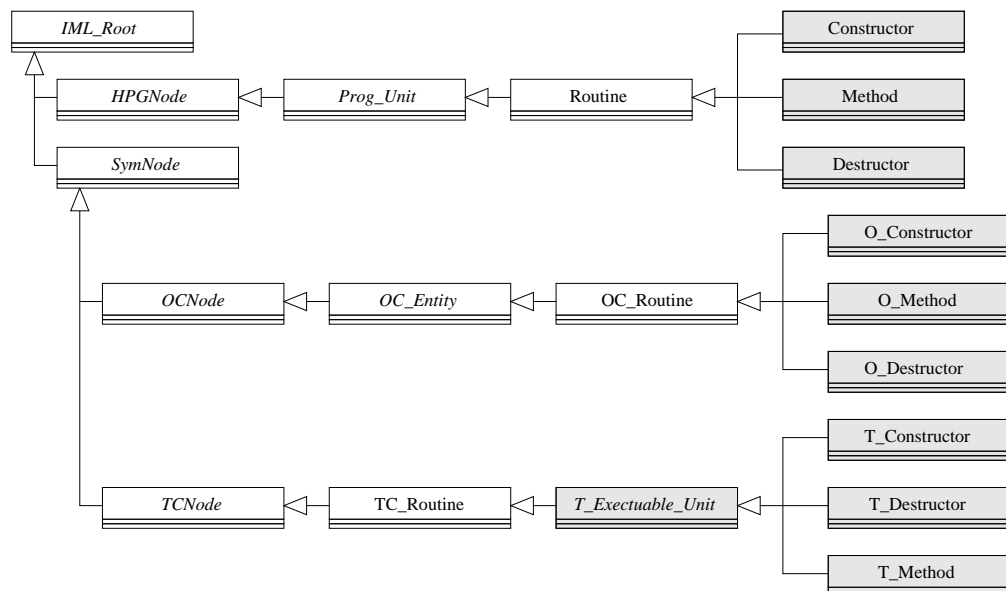


Abbildung 6: Knotentyphierarchie für Methoden, Konstruktoren und Destruktoren

Knotenbeschreibung

Routine	Knotentyp für die Modellierung von Routinen.
statements: <i>Statement_Sequence</i>	Anweisungen, die bei Aufruf der modellierten Routine ausgeführt werden.
<i>CFG_Entry: Basic_Block</i>	Eingangskante des Kontrollflussgraphen.
<i>CFG_Exit: Basic_Block</i>	Ausgangskante des Kontrollflussgraphen.
<i>Callees: list of Routine</i>	Verweis auf alle Routinen, die von der modellierten Routine aufgerufen werden.
<i>Call_Sites: set of Routine_Call</i>	Verweis auf die Stellen im Quelltext, von denen aus der modellierten Routine heraus andere Routinen aufgerufen werden.
<i>Callers: set of Routine</i>	Verweis auf alle Routinen, welche die modellierte Routine aufrufen.
<i>Local_Lifetime_Definitions: list of OC_Stack_Object</i>	Enthält Verweise auf die Objekte, die innerhalb der Routine angelegt werden, und deren Lebenszeit auf die Ausführungszeit der Routine beschränkt ist.
<i>Definition: OC_Routine</i>	Verweis auf die Definition der Routine, falls die modellierte Routine nur deklariert wurde.
Method	Knotentyp für die Modellierung einer Methode.
Constructor	Knotentyp für die Modellierung eines Konstruktors.
Destructor	Knotentyp für die Modellierung eines Destruktors.

OC_Routine	Knotentyp für die Modellierung von Objekten von Routinen.
<i>HPG_Routine: Routine</i>	Verweis auf den Knoten im HPG, an dem diese Routine definiert oder deklariert wurde.
<i>Storage_Class: C_Storage_Class</i>	Speicherklasse der modellierten Routine.
<i>Definition: OC_Routine</i>	Verweis auf die tatsächliche Definition der modellierten Routine, falls die Routine deklariert wurde.
<i>Ellipse: Boolean</i>	Gibt an, ob die Anzahl der aktuellen Parameter unbestimmt ist.
<i>Allocate_Result: Boolean</i>	Zeigt an, ob für den Rückgabewert Speicher reserviert wird.
<i>Unsure_Result: Boolean</i>	Zeigt an, ob es unsicher ist, ob für den Rückgabewert Speicher reserviert werden muss.
<i>PTFs: list of Storable</i>	Attribut, das für die Points-To-Analyse verwendet wird (PTF = Partial-Transfer-Function).
<i>Params: list of OC_Parameter</i>	Liste der aktuellen Parameter der modellierten Routine.
O_Method	Knotentyp für die Modellierung von Objekten für Methoden.
O_Constructor	Knotentyp für die Modellierung von Objekten für Konstruktoren.
O_Destructor	Knotentyp für die Modellierung von Objekten für Destruktoren.
TC_Routine	Knotentyp für die Modellierung des Typs von Routinen
<i>Params: list of OC_Parameter</i>	Liste der formalen Parameter der modellierten Routine.

<i>Return_Type: TCNode</i>	Typ des Rückgabewerts.
<i>Ellipse: Boolean</i>	Zeigt an, ob die Anzahl der aktuellen Parameter unbekannt ist.
<i>Throws: List of T_Compound_Type</i>	Liste der Ausnahmen, die von der Routine ausgelöst werden können.
<i>Is_Synchronized: Boolean</i>	Zeigt an, ob für die modellierte Routine das Attribut 'synchronized' angegeben wurde.
<i>Is_Native: Boolean</i>	Zeigt an, ob die Routine 'native' implementiert ist.
<i>Is_Strictfp: Boolean</i>	Zeigt an, ob für die modellierte Routine das Attribut 'strictfp' angegeben wurde.
<i>Friend_Of: set of T_Class</i>	Verweis auf die Klassen, in denen diese Routine als 'friend' deklariert wurde.
<i>Is_Inline: Boolean</i>	Zeigt an, ob für die modellierte Routine das Attribut 'inline' angegeben wurde.

T_Executable_Unit	Knotentyp für die Gruppierung von Knotentypen, die ausführbare Einheiten wie Methoden, Destruktoren und Konstruktoren modellieren.
<i>Is_Abstract: Boolean</i>	Zeigt an, ob die modellierte ausführbare Einheit abstrakt ist.
<i>Is_Virtual: Boolean</i>	Zeigt an, ob die modellierte ausführbare Einheit virtuell ist.
<i>Its_Class: T_Compound_Type</i>	Verweis auf den Typ, in dem die modellierte ausführbare Einheit definiert wurde.
<i>This_Offset: Natural</i>	Position in der Liste der formalen Parameter, an welcher der Parameter für das aktuelle Objekt steht.

T_Method	Knotentyp für die Abbildung einer Methode.
<i>Is_Constant: Boolean</i>	Zeigt an, ob die modellierte Methode konstant ist. Eine Methode ist konstant, wenn keine Schreibzugriffe auf das aktuelle Objekt ausgeführt werden.
<i>Is_Operator: Boolean</i>	Zeigt an, ob es sich bei der modellierten Methode um einen überladenen Operator handelt.

T_Constructor	Knotentyp für die Modellierung von Konstruktoren.
<i>Initialization: list of Assignment</i>	Liste der direkten Initialisierungen von Attributen.
<i>Is_Explicit: Boolean</i>	Zeigt an, ob für diesen Konstruktor das Attribut 'explicit' angegeben wurde.

T_Destructor	Knotentyp für die Modellierung von Destruktoren.
---------------------	--

Beispiel

In Quelltext 2 ist ein Ausschnitt aus einem Java-Programm dargestellt. In den Zeilen 2, 6 und 9 werden Methoden für die Klasse `MyApp` definiert. Die Klasse `MyApp` wird in Zeile 1 definiert. Die in Zeile 2 definierte Methode ist ein Konstruktor. Die Methode `initEventHandler` in Zeile 6 ist eine Instanzmethode. Die Methode `main` in Zeile 9 ist eine statische Methode der Klasse `MyApp`.

```
1  public class MyApp extends JFrame {
2      public MyApp()
3      {
4          ...
5      }
6      private void initEventHandling() {
7          ...
8      }
9      public static void main(String[] args) {
10         ...
11     }
12 }
```

Quelltext 2: Methoden

In Abbildung 7 ist ein Ausschnitt aus dem IML-Graph dargestellt. Der IML-Graph modelliert einen Teil des in Quelltext 2 vorgestellten Java-Programms. Auf die Darstellung der `Declaration_Point`-Knoten, die bereits in der Beschreibung zu Abbildung 4 diskutiert wurden, wurde verzichtet.

Methoden werden als eingeschachtelte Einheiten (Subunit) ihrer Klasse modelliert (1). Das Klassenobjekt der modellierten Klasse verweist auf ihren Konstruktor (9) und auf statische Methoden, die für die Klasse definiert sind (10). Auf Instanzmethoden wird vom Typknoten der modellierten Klasse verwiesen (8).

Wie bereits beschrieben, werden statische Methoden durch einen Routine-Knoten (2) modelliert, Konstruktoren durch einen Constructor-Knoten und Instanzmethoden durch einen Method-Knoten (4). Alle diese Knoten verweisen auf ihren Objektknoten. Der Objektknoten verweist auf den Typknoten. Für die Methode `initEventHandling` ist dies in (5) und (6) dargestellt.

Konstruktoren und Instanzmethoden erhalten zusätzlich zu den definierten Parametern den speziellen Parameter `this` (7). Der `this` Parameter verweist auf die Instanz, auf der ein Methodenaufruf ausgeführt wird. Bei einem Konstruktoraufruf verweist `this` auf das erzeugte, zu initialisierende Objekt. Auf die Parameter einer Methode verweisen sowohl die Objekt- als auch die Typknoten von Methoden und Konstruktoren. Statischen Methoden wird kein `this`-Parameter hinzugefügt, da sie nicht auf einer Instanz der Klasse, sondern auf dem Klassenobjekt arbeiten.

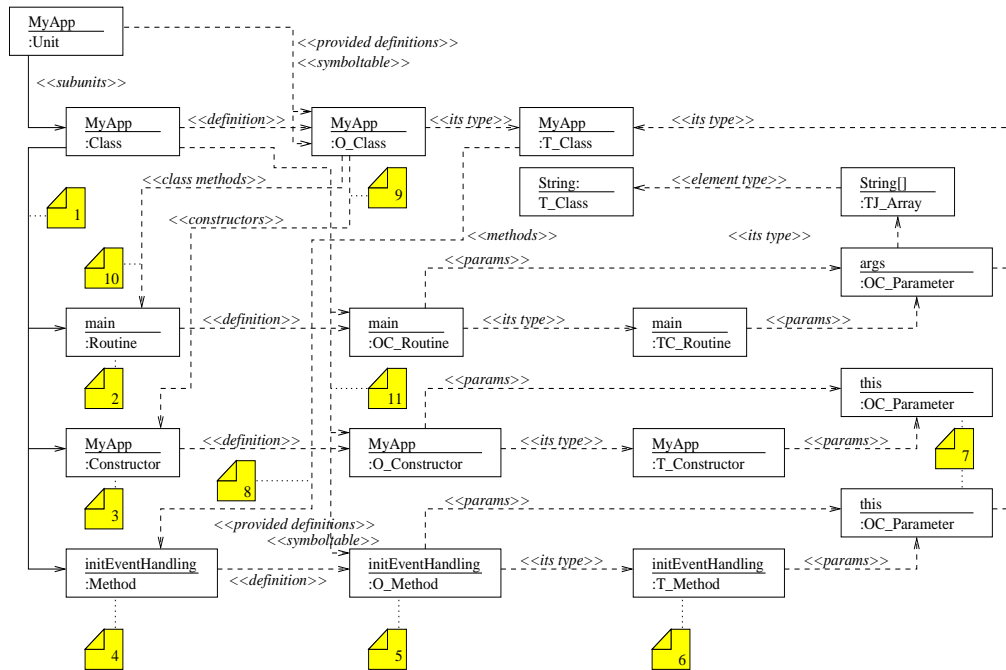


Abbildung 7: Modellierung von Methoden in IML

3.2.5 Aufruf von Methoden

Der Aufruf von Methoden auf Klassen oder Objekten wird in IML verschieden modelliert. Die Unterscheidung für die Modellierung hängt von dem Objekt ab, auf dem eine Methode aufgerufen wird.

Wird eine Methode auf dem Klassenobjekt aufgerufen, dann kann ein direkter Methodenaufruf modelliert werden. Dies ist möglich, da bei Aufruf einer Methode auf dem Klassenobjekt der Typ des Klassenobjekts angegeben werden muss und die Methode eindeutig bestimmt ist.

Erfolgt ein Methodenaufruf an einer Instanz, dann hängt die tatsächlich ausgeführte Methode vom dynamischen Typ der Instanz ab. Der Typ der Instanz kann vom definierten Typ oder einer Unterklasse des Typs sein. Mehrere Methoden können also Ziel eines Methodenaufrufs sein. Aus den möglichen Methoden, wird die passende zur Ausführungszeit ausgesucht. Um dies auszudrücken, wird ein virtueller Methodenaufruf modelliert.

Die Vererbungshierarchie der Knotentypen, die für die Modellierung von Methodenaufrufen benötigt werden, ist in Abbildung 8 dargestellt. Nach Vorstellung der Knotentyphierarchie werden die einzelnen Knotentypen detailliert beschrieben. Nach der Beschreibung der Knotentypen folgt ein Beispiel für die Modellierung von Methodenaufrufen in IML.

Knotentyphierarchie

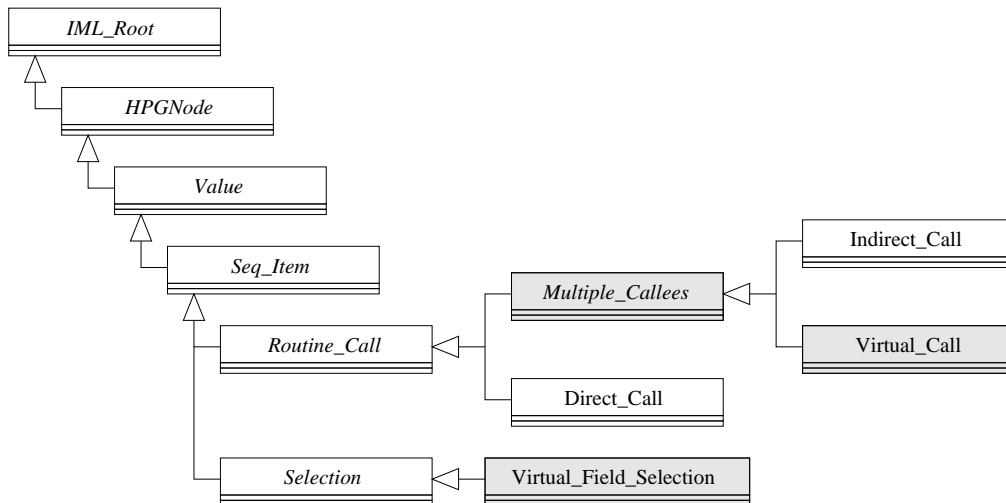


Abbildung 8: Knotentyphierarchie für die Modellierung von Methodenaufrufen

Knotenbeschreibung

Value	Knotentyp für die Gruppierung der Knotentypen des HPG. Unterknoten modellieren im HPG Anweisungen und Ausdrücke.
<i>EType: TCNode</i>	Verweis auf den Ergebnistyp der modellierten Anweisung oder des modellierten Ausdrucks.
Seq_Item	Knotentyp für die Gruppierung von Knotentypen, die Teile einer Anweisungssequenz modellieren.
Routine_Call	Knotentyp für die Gruppierung von Knotentypen, die den Aufruf von Routinen modellieren.
Expr: Value	Anweisung, die eine Routine aufruft.
Pres: list of APara	Anweisungen, die auf den aktuellen Parametern ausgeführt werden, bevor die gerufene Routine ausgeführt wird.

Posts: list of APara

Anweisungen, die auf den aktuellen Parametern ausgeführt werden, nachdem die Ausführung der Routine beendet ist.

Previous_Definition: Definition_Table

Wird benötigt für die Berechnung der SSA-Form (Static-Single-Assignment) eines Programms.

Direct_Call

Knotentyp für die Modellierung eines direkten Routinenaufrufs.

Routine_Decl: OC_Routine

Verweis auf das Objekt der gerufenen Routine.

Multiple_Callees

Knotentyp für die Gruppierung von Knotentypen, die Methodenaufrufe mit mehreren möglichen Zielen modellieren.

Value_Set: Storable

Wird benötigt für die Points-To-Analyse.

Indirect_Call

Knotentyp für die Modellierung eines Routinenaufrufs über einen Funktionszeiger.

Virtual_Call

Knotentyp für die Modellierung eines Aufrufs einer virtuellen Methode.

Selection

Knotentyp für die Gruppierung von Knotentypen, die eine Auswahl eines Werts aus einem Feld modellieren. Dies kann zum Beispiel das Lesen eines Arrayelements sein.

Virtual_Field_Selection	Knotentyp für die Modellierung der Auswahl einer gerufenen virtuellen Methode aus der virtuellen Funktionstabelle (Attribut <code>Virtual_Call_Table</code> in <code>T_Class</code>).
Object: Value	Verweis auf das Objekt auf dem die virtuelle Funktion ausgeführt werden soll. Der Typ des Objekts enthält die Tabelle virtueller Funktionen.
Field: <i>OCNode</i>	Verweis auf die Methode, die in der virtuellen Tabelle nachgeschlagen werden muss.

Beispiel

Der Ausschnitt aus einem Java-Programm in Quelltext 3 enthält zwei Methodenaufrufe. In Zeile 11 wird die Methode `initEventHandlering` auf dem aktuellen Objekt aufgerufen. Zeile 10 enthält einen indirekten Aufruf des Konstruktors der Klasse `MyApp`. Der Konstruktor wird durch den `new`-Operator aufgerufen.

```

1  public class MyApp extends JFrame {
2      public MyApp() {
3          ...
4      }
5      public void initEventHandlering() {
6          ...
7      }
8      public static void main(String[] args)
9      {
10         MyApp app = new MyApp();
11         app.initEventHandlering();
12         ...
13     }
14 }

```

Quelltext 3: Aufruf von Methoden

Abbildung 9 stellt einen Ausschnitt aus dem IML-Graphen dar, der die Modellierung der Methodenaufrufe enthält.

Der Konstruktoraufruf durch den `new`-Operator ist durch einen `Direct_Call`-Knoten modelliert (1). Der Objektknoten des Konstruktors wird als Ziel des Aufrufs durch den `Direct_Call`-Knoten referenziert. Der Ausdruck, der den Aufruf des Konstruktors auslöst, wird ebenfalls durch den `Direct_Call`-Knoten referenziert.

werden. Eine ausführliche Beschreibung der möglichen Kontrollflüsse bei Auftreten von Ausnahmen kann [Harrold, Sinha 1998] entnommen werden.

Im Folgenden wird nun die Vererbungshierarchie der für die Modellierung benötigten Knotentypen vorgestellt. Daran schließt sich eine ausführliche Vorstellung der Knotentypen an. Nach der Vorstellung der Knotentypen illustrieren zwei Ausschnitte aus einem IML-Graphen die Modellierung von Ausnahmebehandlung und -auslösung.

Knotentyphierarchie

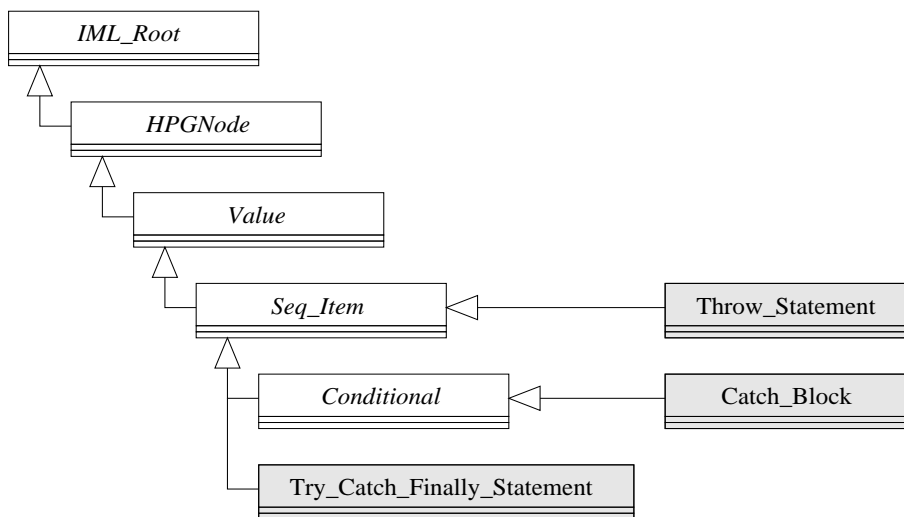


Abbildung 10: Knotentyphierarchie für die Abbildung der Ausnahmebehandlung und Ausnahmeauslösung

Knotenbeschreibung

Try_Catch_Finally	Knotentyp, der die einzelnen Anweisungsblöcke für die Behandlung von Ausnahmen zusammenfasst.
Guarded_Block: Statement_Sequence	Anweisungssequenz, die Anweisungen enthält, durch deren Ausführung möglicherweise eine Ausnahme auftreten kann.
Handling_Block: Catch_Block	Verweis auf den Beginn der Catch_Block-Kaskade.
Finalization: Statement_Sequence	Anweisungssequenz mit Anweisungen, die immer, das heißt unabhängig von einer aufgetretenen Ausnahme, ausgeführt werden.

Throw_Statement	Knotentyp für die Modellierung der <code>throw</code> -Anweisung. Durch die <code>throw</code> -Anweisung können Ausnahmen ausgelöst werden.
Exc_Type: Value	Ausdruck, der den Typ der ausgelösten Ausnahme bestimmt.
End_Lifetime	Verweis auf alle Objekte, deren Lebenszeit bei Ausführung der <code>throw</code> -Anweisung endet.

Conditional	Knotentyp für die Gruppierung von Knotentypen, die bedingte Verzweigungen modellieren.
CondV: Value	Ausdruck, der den auszuführenden Pfad der Verzweigung bestimmt.
ThenP: Value	Anweisung oder Anweisungssequenz, die ausgeführt wird, wenn der in CondV modellierte Ausdruck wahr ist.
ElseP: Value	Anweisung oder Anweisungssequenz, die ausgeführt wird, wenn der in CondV modellierte Ausdruck unwahr ist.

Catch_Block	Knotentyp für die Modellierung eines <code>catch</code> -Blocks. Mehrere <code>catch</code> -Blöcke für die Behandlung verschiedener Ausnahmetypen werden durch eine If-Kaskade modelliert. Eine solche If-Kaskade ist beispielhaft in Abbildung 12 dargestellt.
--------------------	--

Beispiel

In Quelltext 4 wird in Zeile 3 eine Ausnahme ausgelöst. In Zeile 7 bis 18 wird die Ausnahme behandelt. Im `try`-Block, Zeile 7 bis 9, werden Anweisungen ausgeführt, die eine Ausnahme auslösen können. Die beiden `catch`-Blöcke, Zeile 10 bis 15, behandeln auftretende Ausnahmen.

Der erste `catch`-Block, Zeile 10 bis 12, behandelt alle Ausnahmen, die vom Typ `SomeException` oder einem Untertyp von `SomeException` sind.

Der zweite `catch`-Block, Zeile 13 bis 15, behandelt alle Ausnahmen, die vom Typ `Exception` oder einem Untertyp von `Exception` sind. Ausnahmen, die bereits vom ersten `catch`-Block behandelt wurden, werden im

zweiten `catch`-Block nicht mehr behandelt. Der `finally`-Block, in Zeile 16 bis 18, wird unabhängig davon ausgeführt, ob eine Ausnahme aufgetreten ist oder nicht.

```
1  public void printGreeting()  
2      throws SomeException {  
3      throw new SomeException();  
4  }  
5  
6  public void tryToPrintGreeting() {  
7      try {  
8          printGreeting();  
9      }  
10     catch (SomeException e) {  
11         ...  
12     }  
13     catch (Exception e) {  
14         ...  
15     }  
16     finally {  
17         ...  
18     }
```

Quelltext 4: Auslösen und behandeln von Ausnahmen

Ausschnitte aus dem IML-Graphen, der den Programmausschnitt modelliert, sind in Abbildung 11 und Abbildung 12 dargestellt. In Abbildung 11 wird das Auslösen der Ausnahme `SomeException` modelliert. Abbildung 12 enthält die Modellierung der Ausnahmebehandlung durch `try`-, `catch`- und `finally`-Blöcke.

3.2.7 Auslösen von Ausnahmen

Beim Auslösen einer Ausnahme (`throw`; werfen einer Ausnahme) wird ein Ausnahmeobjekt erzeugt. Das Ausnahmeobjekt definiert den Typ der aufgetretenen Ausnahme. Ein Ausnahmeobjekt kann Informationen über den aufgetretenen Fehler enthalten.

Nach Auslösen einer Ausnahme endet die Ausführung der aktuellen Anweisung. Das Ausnahmeobjekt wird an den nächsten `catch`-Block übergeben. Findet sich kein `catch`-Block in der aktuellen Methode, wird das Ausnahmeobjekt an die aufrufende Methode weitergegeben. Ein Ausnahmeobjekt wird solange an den Aufrufer weitergegeben, bis kein Aufrufer mehr vorhanden ist und das Programm verlassen wird oder die Ausnahme behandelt wird.

Wird eine Ausnahme in der aktuellen Methode oder von einer Methode im Methodenaufrufstack behandelt, dann wird die Ausführung des Programms nach der Behandlung der Ausnahme normal fortgesetzt. Das Ausnahmeobjekt existiert nach der Ausnahmebehandlung nicht mehr.

Die Modellierung der Ausnahmenauslösung ist in Abbildung 11 dargestellt. Die `throw`-Anweisung, durch die eine Ausnahme ausgelöst wird, wird durch einen `Throw_Statement`-Knoten (1) modelliert.

Im IML-Graphen gibt es genau einen Knoten, der die aktuelle Ausnahme modelliert. Das Ausnahmeobjekt wird modelliert durch einen `O_Instance`-Knoten (3). Die Initialisierung des Ausnahmeobjekts erfolgt durch den `Throw_Statement`-Knoten.

Bei der Initialisierung des Ausnahmeobjekts durch den `Throw_Statement`-Knoten wird die erzeugte Ausnahme dem IML-Ausnahmeobjekt zugewiesen (2). In Abbildung 11 wird eine Ausnahme durch den `new`-Operator erzeugt. Das Ergebnis des `new`-Operators wird dem Ausnahmeobjekt durch einen `Assignment`-Knoten zugewiesen.

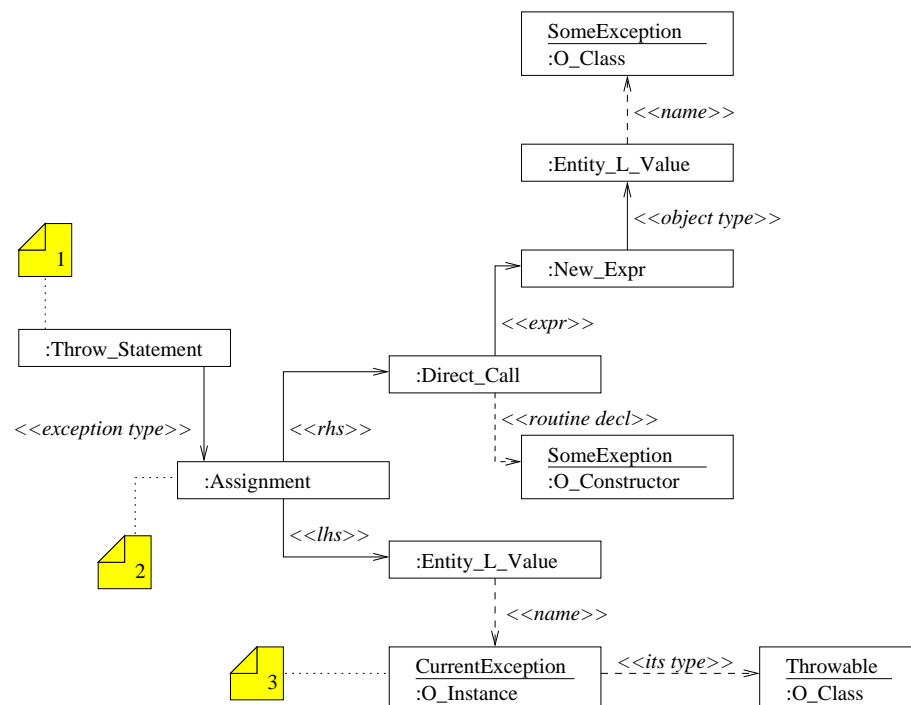


Abbildung 11: Auslösen von Ausnahmen

Ausnahmebehandlung

Ausnahmen werden in Java durch `try`-, `catch`- und `finally`-Blöcke behandelt. In der IML wird dieses Block-Tripel durch einen `Try_Catch_Finally_Statement`-Knoten (1) modelliert. In Abbildung 12 ist ein Ausschnitt aus dem die Ausnahmebehandlung modellierenden IML-Graphen dargestellt.

Der `try`-Block ist als Attribut im `Try_Catch_Finally_Statement`-Knoten vom Typ `Statement_Sequence` modelliert. Das gleiche gilt für den `finally`-Block (3).

Die Modellierung der `catch`-Blöcke ist aufwändiger. `Catch`-Blöcke werden in einer `If`-Kaskade modelliert. Die `If`-Kaskade wird mit `Catch_Block`-Knoten (4) erstellt. Der `Catch_Block`-Knoten enthält als Bedingung für den `then`-Zweig eine Überprüfung, ob der Typ der aktuellen Ausnahme mit dem Typ der behandelbaren Ausnahme übereinstimmt (5). Ist der Typ der aktuellen Ausnahme (6) mit dem Typ der behandelbaren Aus-

nahme kompatibel, wird der catch-Block ausgeführt (8). Sind die Typen nicht kompatibel, wird der nächste catch-Block geprüft (9). Nach Ausführung eines catch-Blocks werden keine weiteren catch-Blöcke ausgeführt. Die Catch_Block-Kaskade wird verlassen und der finally-Block, falls vorhanden, wird ausgeführt.

Die Modellierung des Anweisungsblocks eines catch-Blocks enthält als erste Anweisung eine Zuweisung der aktuellen Ausnahme an den Ausnahme-parameter des catch-Blocks (8).

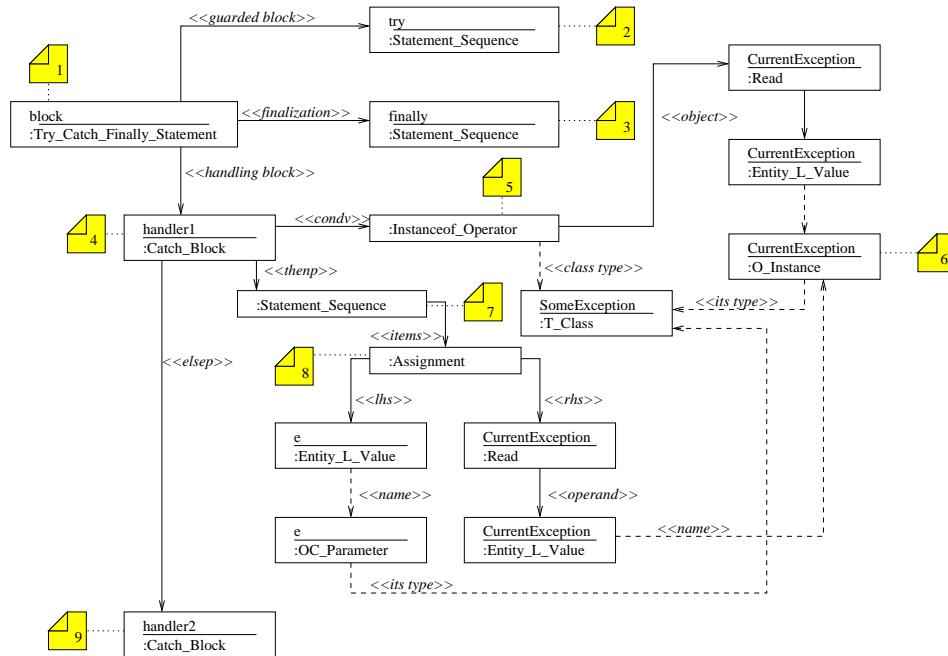


Abbildung 12: Behandlung von Ausnahmen

3.2.8 Operatoren

Viele in Java verfügbaren Operatoren stimmen mit den in C verfügbaren Operatoren überein. Für zusätzliche Operatoren, die in Java ergänzt wurden, sind IML-Knotentypen für die IML-Modellierung erstellt worden. Die Hierarchie der Knotentypen ist in Abbildung 13 dargestellt. Im Anschluss an die Knotentyphierarchie werden die einzelnen Knotentypen detailliert vorgestellt.

Knotentyphierarchie

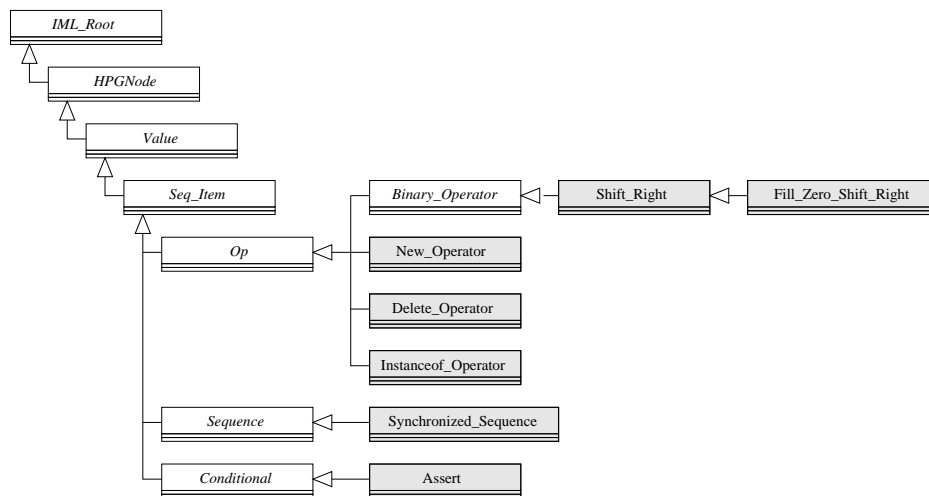


Abbildung 13: Knotentyphierarchie für Operatoren

Knotenbeschreibung

Op	Knotentyp für die Gruppierung von Knotentypen, durch die Operatoren modelliert werden.
-----------	--

New_Operator	Knotentyp für die Modellierung des new-Operators. Der new-Operator erzeugt Instanzen von Klassen.
---------------------	---

Object_Type: Value	Ausdruck, der die Klasse bestimmt, von dem eine Instanz erzeugt werden soll.
--------------------	--

Number: Natural	Anzahl der Instanzen, die von der angegebenen Klasse erzeugt werden sollen.
-----------------	---

Delete_Operator	Knotentyp für die Modellierung des <code>delete</code> -Operators. Der <code>delete</code> -Operator gibt für Objekte reservierten Speicher auf dem Heap frei. Dieser Operator ist in der Programmiersprache C++ enthalten. In Java existiert dieser Operator nicht.
Object: Value	Ausdruck, der das Objekt bestimmt, auf das der <code>delete</code> -Operator angewendet werden soll.

Instanceof_Operator	Knotentyp für die Modellierung des <code>instanceof</code> -Operators. Der <code>instanceof</code> -Operator prüft, ob der Typ eines Objekts zu einem angegebenen Typ passend ist. Passend bedeutet, dass die Typen entweder gleich, oder der Typ des Objekts ein Untertyp des angegebenen Typs ist.
Object: Value	Ausdruck, der das Objekt bestimmt, dessen Typ "verglichen" werden soll.
<i>Class_Type: T_Compound_Type</i>	Verweis auf den Typ, mit dem der Typ des angegebenen Objekts "verglichen" werden soll.

Binary_Operator	Knotentyp für die Gruppierung von Knotentypen, die Operatoren mit zwei Operanden modellieren.
LOp: Value	Ausdruck, der den linken Operanden des Operators bestimmt.
ROp: Value	Ausdruck, der den rechten Operanden des Operators bestimmt.

Shift_Right	Knotentyp für die Modellierung des Rechtsschiebe-Operators.
--------------------	---

Fill_Zero_Shift_Right	Knotentyp für die Modellierung des Rechtsschiebe-Operators, der die frei werdenden Bits mit 0 auffüllt.
Assert	Knotentyp für die Modellierung der <code>assert</code> Anweisung.
Sequence	Knotentyp für die Modellierung einer Folge von Anweisungen.
Items: list of Seq_Item	Liste der Anweisungen.
Synchronized_Sequence	Knotentyp für die Modellierung eines als <code>synchronized</code> deklarierten Anweisungsblocks.
synchronize_on: Seq_Item	Ausdruck, der den Typ bestimmt, dessen Monitor für die Synchronisation verwendet werden soll. Der ausgeführte Block wird über den Monitor synchronisiert.

3.2.9 Typumwandlung

Im Vergleich zu C werden in Java Typumwandlungsoperationen zur Laufzeit geprüft. Die Prüfung beschränkt sich darauf, ob die Typumwandlung zulässig ist [Gosling et al. 2000].

Für die Modellierung der Java-Typumwandlung in IML wurde die `Value_Conversion`-Knotentyphierarchie erweitert. Für Java wurde ein neuer Knotentyp eingefügt. Weitere Knotentypen, die eingefügt wurden, modellieren die Vielzahl von Typumwandlungsoperation in C++.

Die erweiterte Knotentyphierarchie ist in Abbildung 14 dargestellt. Die einzelnen Knotentypen werden detailliert im Anschluss an Abbildung 14 vorgestellt.

Knotentyphierarchie

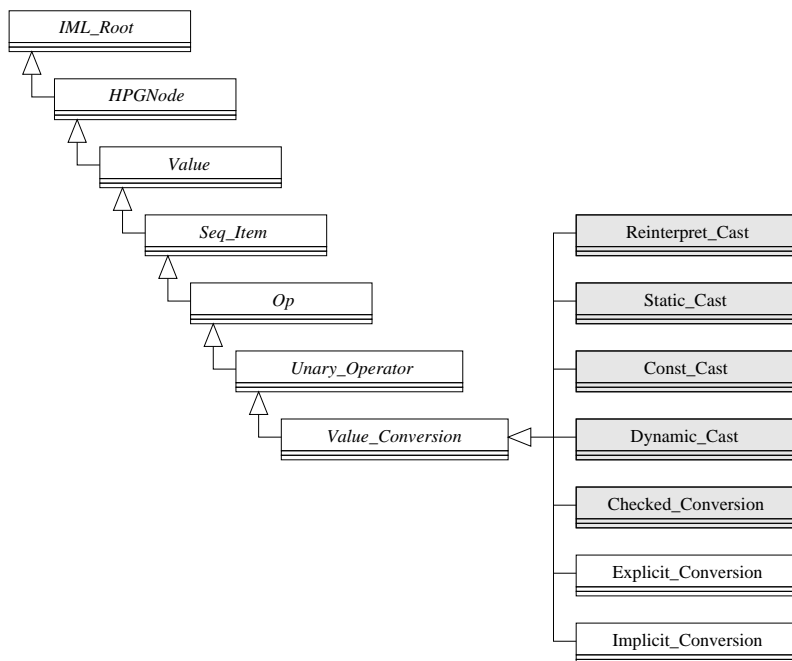


Abbildung 14: Knotentyphierarchie für Typumwandlungsoperationen

Knotenbeschreibung

Unary_Operator	Knotentyp für die Gruppierung von Knotentypen, die Operatoren mit einem Operanden modellieren.
Operand: Value	Anweisung, die den Operanden bestimmt, auf den der modellierte Ausdruck angewendet werden soll.
Value_Conversion	Knotentyp für die Gruppierung von Knotentypen, die Typumwandlungsoperatoren modellieren.
Source_Type: TCNode	<p>Typ, der gewandelt werden soll.</p> <p>Den Zieltyp der Umwandlung bestimmt das ererbte Attribut EType aus dem Knotentyp Value.</p> <p>Das Objekt, dessen Typ konvertiert werden soll, wird durch das ererbte Attribut Operand aus dem Knotentyp Unary_Operator bestimmt.</p>

Checked_Cast	Knotentyp für die Modellierung einer kontrollierten Typumwandlung. Hierbei werden die Typen auf Kompatibilität geprüft. Passen Ziel- und Quelltyp nicht zueinander, wird eine Ausnahme ausgelöst. Durch diesen Knotentyp wird der Java cast-Operator modelliert.
Reinterpret_Cast	Knotentyp für die Modellierung des C++ reinterpret_cast.
Static_Cast	Knotentyp für die Modellierung des C++ static_cast.
Const_Cast	Knotentyp für die Modellierung des C++ const_cast.
Dynamic_Cast	Knotentyp für die Modellierung des C++ dynamic_cast.
Explicit_Conversion	Knoten für die Modellierung einer expliziten Typumwandlung ohne Prüfung des Quell- und Zieltyps.
Implicit_Conversion	Modellierung einer impliziten Typumwandlung.

3.2.10 Objektknoten und Typknoten

Die größten Erweiterungen der Objekt- und Typknoten wurden bereits in den vorigen Abschnitten vorgestellt. Die hier vorgestellten Erweiterungen sind Erweiterungen der in C verfügbaren Typen. Für Strukturen in C++ ist zum Beispiel eine Erweiterung um die Möglichkeit der Vererbung nötig.

In Abbildung 15 wird nun zunächst die Hierarchie der hinzugefügten oder geänderten Typ- und Objektknoten vorgestellt. Im Anschluss daran werden die einzelnen Knotentypen detailliert vorgestellt.

Knotentyphierarchie

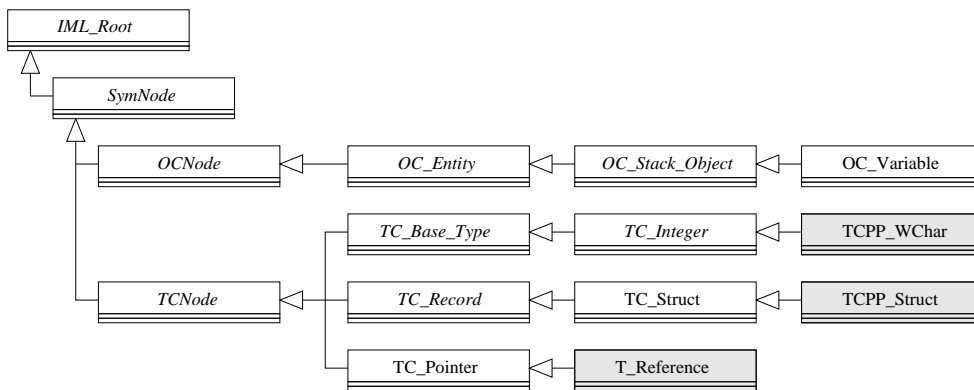


Abbildung 15: Zusätzliche Objekt- und Typklassen

Knotenbeschreibung

OC_Stack_Object	Knotentyp für die Gruppierung von Knotentypen, die Objekte modellieren, die auf dem Stack angelegt werden.
<i>Storage_Class: C_Storage_Class</i>	Speicherklasse, die für das modellierte Objekt deklariert wurde.
<i>Definition_Offset: Natural</i>	Position, an der das modellierte Objekt in der Symboltabelle in der definierenden oder deklarierenden programmiersprachlichen Gruppierung gespeichert ist.
<i>Definition_Table_Owner: HPGNode</i>	Verweis auf die programmiersprachliche Gruppierung, in der das modellierte Objekt definiert oder deklariert ist.
OC_Variable	Knotentyp für die Modellierung von Variablen.
<i>Definition: OC_Variable</i>	Verweis auf die tatsächliche Definition eines Objekts, falls das modellierte Objekt 'extern' deklariert ist.

<i>Is_Mutable: Boolean</i>	Zeigt an, ob die modellierte Variable <code>mutable</code> deklariert wurde.
<i>Is_Transient: Boolean</i>	Zeigt an, ob die modellierte Variable <code>transient</code> deklariert wurde.
TC_Base_Type	Knotentyp für die Gruppierung von Knotentypen, die Basisdatentypen modellieren.
TC_Integer	Knotentyp für die Gruppierung von Knotentypen, die ganzzahlige Typen modellieren.
<i>Sign: C_Adjective_Sign</i>	Gibt an, ob der modellierte Typ <code>signed</code> oder <code>unsigned</code> definiert wurde.
TCP_P_WChar	Knotentyp für die Modellierung des <code>wchar</code> Typ in C++.
TC_Record	Knotentyp für die Gruppierung von Strukturen in C++.
<i>Comps: list of OC_Component</i>	Liste der Komponenten, die in der Struktur enthalten sind.
TC_Struct	Knotentyp für die Modellierung von Strukturen in C.
TCP_P_Struct	Knotentyp für die Modellierung von Strukturen in C++.
<i>Extends: list of TCP_P_Struct</i>	Liste der Strukturen, von denen die modellierte Struktur ableitet.
<i>Extended_By: set of TCP_P_Struct</i>	Liste der Strukturen, die von der modellierten Struktur ableiten.

TC_Pointer	Knotentyp für die Modellierung von Zeigertypen.
<i>Pointer_Type: TCNode</i>	Verweis auf den Typ der Objekte, auf die der Zeiger zeigen kann.
T_Reference	Knotentyp für die Modellierung von Referenztypen.

Kapitel 4 Entwurf und Implementierung

Die erste Entscheidung für die Implementierung war die Wahl der Programmiersprache. Basierend auf der Wahl der Programmiersprache, wurde ein Grobentwurf für `jafe` erstellt. Der Grobentwurf, in dem das Entwurfsprinzip festgelegt ist, wurde während der Implementierung verfeinert und die endgültige Struktur von `jafe` bildete sich während der Implementierung heraus.

Die folgenden Abschnitte beschreiben nun zunächst die Auswahl der Implementierungssprache, die Entwurfsidee und den Grobentwurf von `jafe`. Im Anschluss daran folgt eine Übersicht über die einzelnen Schritte, die für die Erzeugung der IML durchgeführt werden. Am Schluss dieses Kapitels werden einige Besonderheiten der Implementierung vorgestellt.

4.1 Wahl der Implementierungssprache

Für die Implementierung von `jafe` standen zwei Programmiersprachen zur Wahl. Die möglichen Programmiersprachen waren Ada95 und C++.

Die Implementierung von `jafe` in C++ bietet sich an, da `jikes`, das verwendete Java-Frontend, in C++ implementiert ist. Dies ermöglicht einen direkten Zugriff auf die Symboltabelle und den AST des Frontends. Für den Zugriff auf die IML-Implementierung muss bei Verwendung von C++ eine Schnittstelle realisiert werden. Die Schnittstelle muss es ermöglichen, aus der C++-Implementierung auf die Ada95-Implementierung der IML zuzugreifen. Eine Übersichtsdarstellung von `jafe`, für eine Implementierung in C++, ist in Abbildung 16, Alternative A, dargestellt.

Eine Implementierung von `jafe` in Ada95 hätte den Vorteil, dass keine Schnittstelle für den Zugriff auf die Implementierung der IML realisiert werden muss. Der Nachteil dieser Alternative ist, dass eine C++-Schnittstelle für den Zugriff auf `jafe` implementiert werden muss. Über diese Schnittstelle

stößt das Java-Frontend den IML-Erzeugungsvorgang an. Ein weiterer Nachteil ist, dass auf die Symboltabelle und den AST des Java-Frontends von `jafe` aus, nicht direkt zugegriffen werden kann. Für den Zugriff auf Symboltabelle und AST muss eine Schnittstelle implementiert werden. Die Implementierung der Schnittstelle erfordert, dass jede Klasse, die in der C++-Implementierung deklariert ist, auf Ada95-Seite ebenfalls deklariert werden muss. Ebenso müssen alle Methoden der Klassen auf der Ada95-Seite deklariert werden [Comar, Dewar 1996]. Für den Zugriff auf den AST des Java-Frontends bedeutet dies, dass alle 72 Klassen, die den AST bilden, in Ada95 deklariert werden müssen. Für den Zugriff auf die Symboltabelle ist die Deklaration von 15 Klassen notwendig. Zusätzlich zu der Deklaration der Klassen müssen noch Methoden für den Zugriff auf die Daten der Klassen implementiert werden.

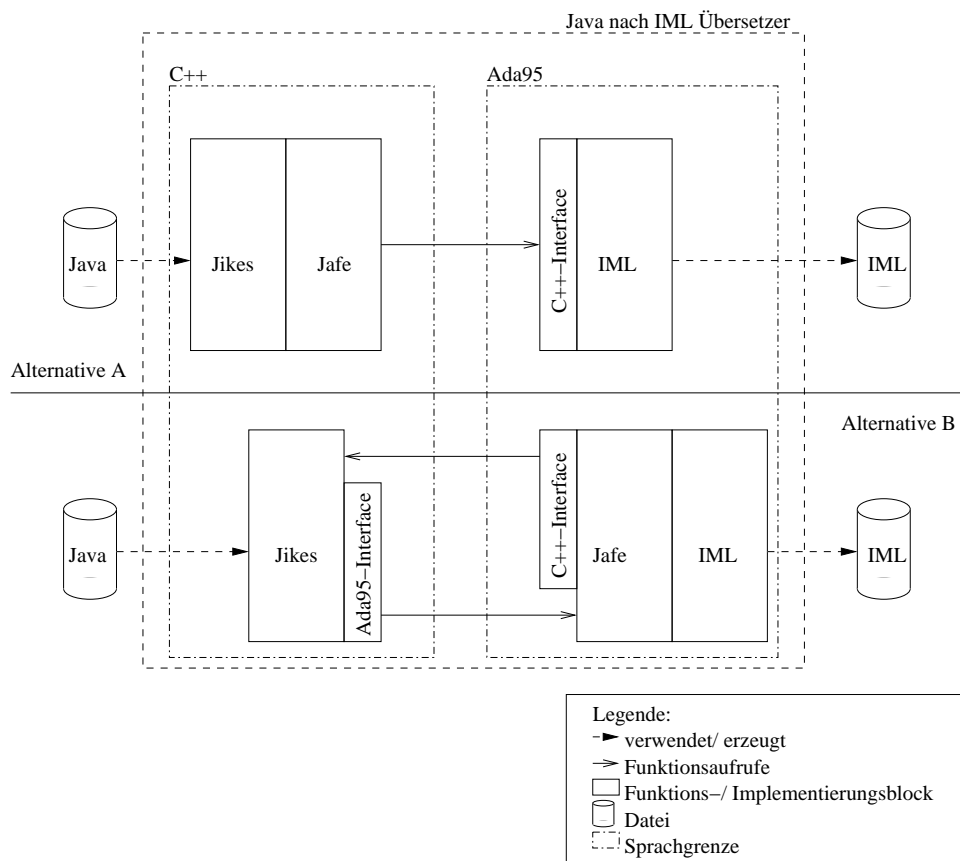


Abbildung 16: Alternativen bei der Wahl der Programmiersprache für die Entwicklung von `jafe`

Die Implementierung von `jafe` in Ada95 verlangt die Entwicklung von zwei Schnittstellen. Die beiden Schnittstellen müssen bei einer Änderung des Java-Frontends oder von `jafe` aufwändig gepflegt werden. Weiterhin ist die Schnittstelle, die den Zugriff auf die C++ Implementierungen von Ada95 aus erlaubt, nicht Teil des Ada95-Standards [ISO/IEC 1995].

Wird `jafe` in C++ realisiert, muss nur die Schnittstelle für den Zugriff auf die IML-Implementierung realisiert werden. Wie sich herausstellte, kann die Schnittstelle durch den IML-Generator automatisch aus der IML-Spezifikation erzeugt werden. Eine Anpassung der Schnittstelle bei Änderungen der IML entfällt durch die automatische Generierung. Die Schnittstelle für den

Zugriff auf Ada Funktionen, Prozeduren und Daten von C-Programmen aus ist im Ada95-Standard definiert [ISO/IEC 1995].

Aus den oben genannten Gründen fiel die Wahl der Implementierungssprache auf C++.

4.2 Entwurf von jafe

Der Entwurf von jafe wurde vor der Implementierung grob ausgeführt. Die Idee, die dem Entwurf zu Grunde liegt, wurde notiert und in der Implementierung realisiert. So bildete sich die endgültige Struktur des Systems während der Entwicklung heraus.

Das Vorgehen, die Implementierung ohne Grobentwurf zu beginnen, entspricht nicht unbedingt dem Mustervorgehen bei der Softwareentwicklung. Vielmehr ist es üblich, dass zunächst ein Grobentwurf erstellt wird. Durch den Grobentwurf wird die Struktur des Systems festgelegt. Der Grobentwurf wird während der Implementierung verfeinert. Das "Nicht-Erstellen" eines Grobentwurfs, sondern lediglich die Verfolgung einer Entwurfsidee, stellte sich im Nachhinein als Vorteil heraus, siehe hierzu Abschnitt 4.3.

Allerdings halte ich dieses Vorgehen generell nicht für sinnvoll. Für eine Einzelperson funktioniert dieses Vorgehen, da eine Person genau weiss, was hinter ihrer Entwurfsidee steckt. Wird allerdings Software im Team entwickelt, entstehen in den Köpfen der beteiligten Personen sicherlich verschiedene Vorstellungen von der Entwurfsidee. Bei einer Softwareentwicklung mit mehr als einer Person, was sicherlich der Normalfall ist, sollte also immer ein Grobentwurf erstellt werden, um den beteiligten Personen eine einheitliche Vorstellung von der Struktur des Systems geben zu können.

4.2.1 Entwurfsidee

Die Entwurfsidee von jafe ist die Realisierung in weitgehend unabhängigen Komponenten. Die einzelnen Komponenten sollen möglichst wenig voneinander abhängen und, falls nötig, einfach austauschbar sein. Für die Implementierung wurden drei Komponenten identifiziert. Diese drei Komponenten sind:

- Java-Frontend,
- Implementierung von jafe und
- Implementierung der IML.

Die Komponenten und deren Schnittstellen sind in Abbildung 17 dargestellt.

Die Komponenten IML-Implementierung und Java-Frontend sind vorgegeben. Die Komponente jafe musste implementiert werden. Sie bildet das Bindeglied zwischen der IML-Implementierung und dem Java-Frontend. Im folgenden Abschnitt wird die Struktur von jafe vorgestellt. Die Struktur der IML-Implementierung und des Java-Frontends werden nicht vorgestellt, da sie nicht verändert, beziehungsweise auf ihre Struktur Einfluss genommen wurde.

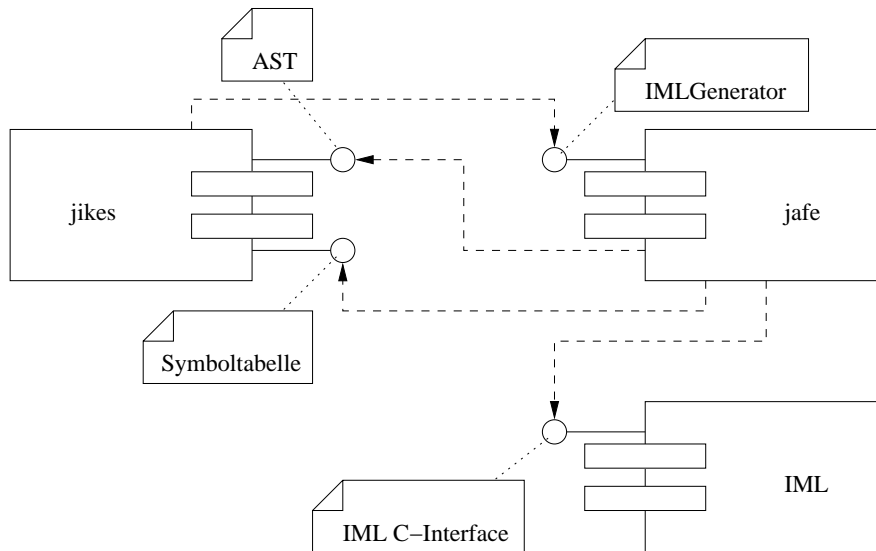


Abbildung 17: Komponentenmodell von jafe

4.2.2 Klassenmodell

Die Implementierung von `jafe` wurde in vier Pakete unterteilt. Jedes Paket implementiert eine Teilfunktionalität von `jafe`. Die Pakete und die darin implementierte Funktionalität sind:

- `Control`
Dieses Paket enthält nur die Klasse `IMLGenerator`. Die Implementierung dieser Klasse steuert den IML-Erzeugungsvorgang und dient als Schnittstelle zum Java-Frontend. Über diese Klasse kann das Java-Frontend den Übersetzungsvorgang starten.
- `Symboltable`
Im Paket `Symboltable` sind Klassen für die Symboltabelle von `jafe` enthalten. Im Wesentlichen verbindet die Symboltabelle von `jafe` die Symboltabelle des Java-Frontends mit der Symboltabelle des IML-Graphen. Die Implementierung der Klassen erzeugt die IML-Knoten für die Symboltabelle des IML-Graphen.
- `Compiler`
Im Paket `Compiler` findet sich die Implementierung für die Übersetzung des AST des Java-Frontends nach IML.
- `Tools`
Dieses Paket hält Methoden vor, die von den anderen Paketen für die Erzeugung der Symboltabelle oder des IML-Graphen benötigt werden.

Eine Übersichtsdarstellung der Pakete, deren gegenseitige Verwendung und die darin enthaltenen Klassen, findet sich in Abbildung 18. Die einzelnen Klassen und ihre Methoden werden hier nicht weiter vorgestellt. An Stelle der Vorstellung der einzelnen Klassen werden die einzelnen Schritte für die Erzeugung eines IML-Graphen durch `jafe` im folgenden Abschnitt vorgestellt. Für das Verständnis der Implementierung von `jafe` ist die Kenntnis der einzelnen Schritte bei der IML-Erzeugung und deren Abfolge, meiner Meinung nach, wichtiger als eine detaillierte Beschreibung aller Klassen und deren Methoden.

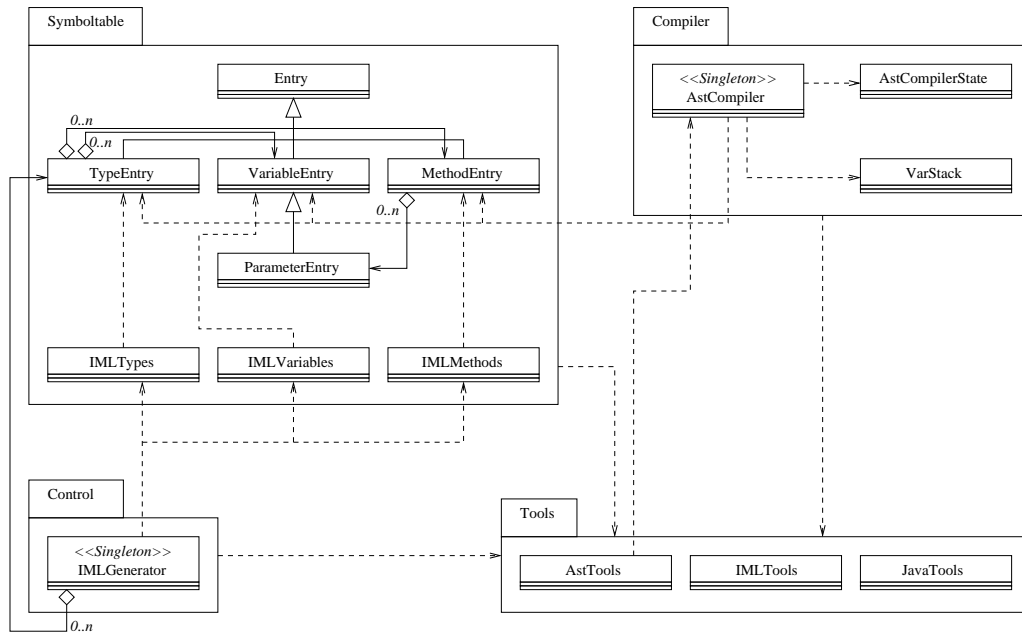


Abbildung 18: Klassenmodell von jafe

4.2.3 Erzeugung der IML

Die Erzeugung der IML verläuft in drei Schritten. Diese Schritte sind:

1. Zusammenstellen der Übersetzungseinheit,
2. Erstellen der Symboltabelle für den IML-Graphen und
3. Erzeugung der IML aus dem AST.

Diese drei Schritte werden in den folgenden Abschnitten detailliert beschrieben. In Abbildung 19 auf Seite 59 sind die einzelnen Schritte graphisch dargestellt.

Erzeugen der Übersetzungseinheit

Wie bereits in Abschnitt 3.2.1 beschrieben, besteht eine Übersetzungseinheit für `jafe` aus einer Klasse oder einem Interface. Alle Klassen und Interfaces, die innerhalb der Klasse oder des Interface definiert sind, gehören ebenfalls zu dieser Übersetzungseinheit und werden in einer IML-Datei gemeinsam gespeichert.

`Jafe` erhält bei Start der IML-Erzeugung vom Java-Frontend alle zu verarbeitenden Typen einer Java-Übersetzungseinheit. Eine Java-Übersetzungseinheit enthält, wie bereits vorgestellt, alle Typen, die in einer Datei definiert sind. Eine Java-Übersetzungseinheit ist also gleich oder größer als eine Übersetzungseinheit von `jafe`. Aus den übergebenen Typen werden die Übersetzungseinheiten für `jafe` gebildet. Die Implementierung für die Erstellung der Übersetzungseinheiten ist in der Klasse `IMLGenerator` ausgeführt.

Nach Erstellung der Übersetzungseinheiten werden alle Übersetzungseinheiten nacheinander in ihre IML-Darstellung übersetzt. Für die Übersetzung nach IML muss zunächst die Symboltabelle des IML-Graphen erzeugt werden.

Erstellen der Symboltabelle für den IML-Graphen

Für die Erzeugung der IML-Darstellung von Java-Programmen benötigt `jafe` eine Symboltabelle. In dieser Symboltabelle werden Informationen zu Typen, deren Attribute und deren Methoden gespeichert. Zusätzlich werden die Vererbungs- und Implementierungsbeziehungen in der Symboltabelle hinterlegt und eine Verbindung zwischen der Symboltabelle von `jikes` und der Symboltabelle des IML-Graphen geschaffen. Eine eigene Symboltabelle für `jafe` ist aus zwei Gründen notwendig:

- Der AST von `jikes` verweist nicht an allen notwendigen Stellen in die Symboltabelle. Dies ist zum Beispiel bei der Abbildung eines Typs im AST der Fall.
- Eine Verbindung zwischen der Symboltabelle von `jikes` und des IML-Graphen muss geschaffen werden.
Dies ist notwendig, um die zu einem Typ, einer Variablen oder einer Methode gehörenden IML-Knoten finden zu können. Eine Möglichkeit diese Verbindung zu implementieren ist die Modifikation der Symboltabelle von `jikes`. Eine zweite Möglichkeit ist, die Verbindung über eine eigene Symboltabelle zu erstellen. Die zweite Möglichkeit wurde gewählt, da erstens eine eigene Symboltabelle ohnehin implementiert werden musste und es zweitens sinnvoll ist, möglichst wenige Änderungen am Java-Frontend durchzuführen. Im Falle eines Wechsels des Java-Frontends, zum Beispiel auf eine neue Version von `jikes`, müssen alle Änderungen nachgezogen werden.

Für die Symboltabelle von `jikes` stehen vier Klassen zur Verfügung:

- `TypeEntry`
Stellt die Verbindung zwischen einem Typsymbol in `jikes` und den Typ modellierende IML-Knoten her. Diese Klasse erzeugt die IML-Knoten für die Modellierung des Typs und trägt sie in den IML-Graphen ein.
- `MethodEntry`
Verbindung zwischen Methodensymbol in `jikes` und die Methode modellierende IML-Knoten. Die IML-Knoten, welche die Methode modellieren, werden von der Klasse erstellt. Die Klasse fügt die IML-Knoten in den IML-Graph ein.
- `VariableEntry` und `ParameterEntry`
Diese beiden Klassen verbinden ein Variablensymbol aus `jikes` mit den IML-Knoten durch welche die Variable oder der Parameter modelliert wird. Die Klassen erzeugen die modellierenden IML-Knoten und tragen diese im IML-Graph ein.

Die Erzeugung der Symboltabelleneinträge werden von den drei Klassen `IMLTypes`, `IMLVariables` und `IMLMethods` gesteuert. `IMLTypes` erzeugt die Symboltabelleneinträge für Typen, `IMLVariables` für Variablen und `IMLMethods` erzeugt die Symboltabelleneinträge für Methoden und deren Parameter im IML-Graph.

Nach der Erstellung der IML-Knoten für Typen, deren Methoden und Attribute und dem Erzeugen der Symboltabelle von `jafe`, wird die IML-Darstellung der Methodenimplementierung und der Attributinitialisierung erzeugt. Das Erstellen der Symboltabelle im Voraus ist notwendig, da für die Erzeugung der IML-Darstellung alle verwendeten Typen, deren Methoden und Attribute bekannt sein müssen.

Erzeugung der IML aus dem AST

Aus dem AST wird zuerst die IML-Darstellung für die Initialisierung der Attribute erstellt. Nach der Bearbeitung der Attribute wird die Implementierung der Methoden in IML übersetzt.

Bei der Erzeugung der IML-Darstellung für die Initialisierung der Attribute wird der Teilbaum des AST traversiert, der die Initialisierung des bearbeiteten Attributs darstellt. Typ- und Objektinformation, zum Beispiel bei der Initialisierung eines Attributs mit dem Wert eines anderen Attributs oder durch einen Methodenaufruf, werden aus der von `jafe` erstellten Symboltabelle entnommen.

Die Implementierung von Methoden wird durch die Traversierung des Teils des ASTs, der die Implementierung der Methode, enthält nach IML übersetzt. Für Aufrufe von Methoden oder Zugriffe auf Attribute wird die Symboltabelle von `jafe` verwendet. Lokale Variablen werden von der Klasse `VarStack` verwaltet. Alle lokalen Variablen, die in einem Block definiert sind, werden vor Beginn der Übersetzung des Blocks in den Stack lokaler Variablen aufgenommen. Aus dem Stack lokaler Variablen können dann deren IML-Knoten, welche die lokalen Variablen modellieren, ermittelt werden. Die Repräsentation lokaler Variablen wird durch die Klasse `VariableEntry` übernommen.

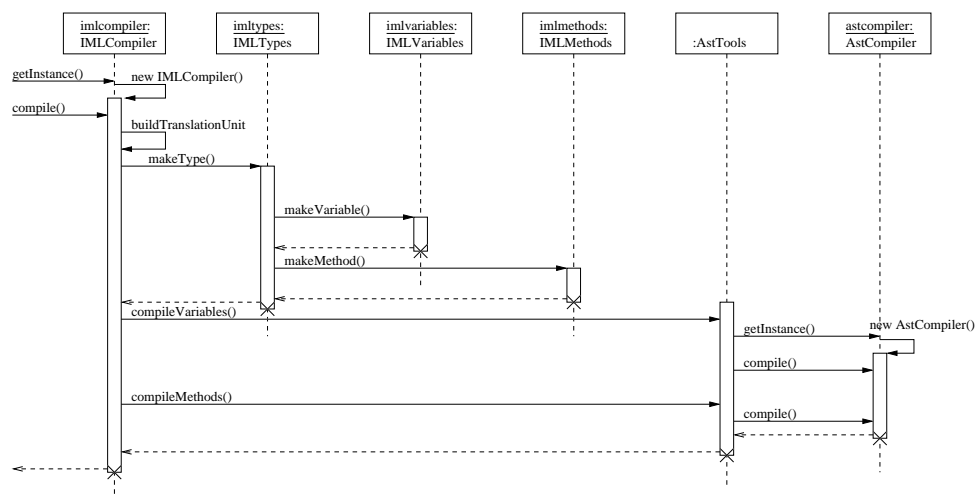


Abbildung 19: Sequenzdiagramm für die IML-Erzeugung in `jafe`

4.3 Implementierung

Das Zitat *“Plan to throw one away; you will, anyhow”* von Frederick Brooks [Brooks 1995] trifft auch auf die Implementierung von `jafe` zu. Es wäre nützlicher gewesen, vor Beginn der Implementierung einen Prototypen zu implementieren, der einen kleinen Teil eines Java-Programms nach IML übersetzen kann. Bei der Implementierung des Prototypen hätte wertvolles Wissen für die Implementierung von `jafe` gesammelt werden können. Durch das gesammelte Wissen hätte während der Implementierung sicherlich einige Zeit eingespart werden können.

Zu Beginn der Diplomarbeit ging ich davon aus, dass die IML-Darstellung von Java-Programmen durch einfaches traversieren des ASTs von `jikes` erzeugt werden kann. Diese Annahme stellte sich aber während der Implementierung als falsch heraus. Insgesamt wurden drei Ansätze für die Realisierung implementiert. Mit dem dritten Ansatz war es dann möglich, eine IML-Darstellung für Java-Programme zu erzeugen.

Der erste Ansatz für die Implementierung verfolgte die Annahme, dass eine Traversierung des ASTs genügt. Dieser Ansatz war allerdings nicht realisierbar, da in den Knoten zu wenig Information über die verwendeten Typen enthalten ist. So gibt es von AST-Knoten, die eine Typdeklaration modellieren, keinen Verweis in die Symboltabelle. Für die Erzeugung der IML-Typknoten lag zu wenig Information vor.

Bei der zweiten Implementierung wurde die fehlende Typinformation in einer eigenen Symboltabelle gespeichert. Die Symboltabelle mit den Typinformationen wurde aus den Typeinträgen der Symboltabelle von `jikes` vor Beginn der Traversierung des ASTs von `jikes` angelegt. Bei dieser Implementierung traten zwei Probleme auf: zum einen war die Typinformation nicht ausreichend; zum anderen war kein Verweis auf die den Typ modellierenden IML-Knoten hinterlegt.

In der dritten Implementierung wurde die benötigte Typinformation aus der Symboltabelle von `jikes` vollständig erzeugt. Den Typen wurden zusätzlich ihre Vererbungs- und Implementierungsbeziehungen, ihre Attribute und Methoden zugeordnet. Gleichzeitig wurden alle IML-Typ- und -Objektknoten für Typen, Methoden und Attribute erzeugt. Erst wenn alle Typinformationen in der Symboltabelle von `jafe` erzeugt sind, wird der AST traversiert. Bei der Traversierung des AST kann die benötigte Information dann schnell aus der `jafe` eigenen Symboltabelle entnommen werden.

In den folgenden Abschnitten werden nun einzelne Teile der Implementierung vorgestellt. Im Einzelnen sind dies die Implementierung der Auflösung von Methodenaufrufen und die Verwaltung lokaler Variablen bei der Übersetzung von Methodenimplementierungen. Die beiden Implementierungen wurden gewählt, da sie mir am interessantesten erschienen.

4.3.1 Auflösung von Methodenaufrufen

Bei der Modellierung von Methodenaufrufen in IML muss das Methodenobjekt der gerufenen Methode angegeben werden. Um das IML-Objekt der aufgerufenen Methode zu finden, wird die Symboltabelle von `jafe` durchsucht.

Methoden in der Symboltabelle von `jafe` sind ihrem definierenden Typ zugeordnet. Zuerst wird der Typ bestimmt, auf dem die Methode aufgerufen wird. Ist der Typ bekannt, wird in dessen Methoden nach der gerufenen Methode gesucht. Die Suche nach der Methode ist nötig, da im AST von `jikes` nicht genügend Information über die gerufene Methode vorliegt. Im AST von `jikes` ist lediglich der Name der gerufenen Method verzeichnet nicht aber deren Symboltabelleneintrag.

Die Suche nach der aufgerufenen Methode erfolgt über deren Namen und den Typen der aktuellen Parameter. Die Typen der aktuellen Parameter werden in der folgenden Beschreibung Typprofil genannt. Unterschieden wird im Folgenden auch zwischen aktuellem und formalem Typprofil. Im aktuellen Typprofil sind alle Typen der Parameter des Methodenaufrufs zusammengefasst. Das formale Typprofil entspricht den Typen, die für die Parameter einer

Methode bei deren Deklaration angegeben wurden. Methodennamen und Typprofil zusammengenommen entsprechen der Signatur der Methode.

Das Ziel der Suche ist eine formale Signatur, die am “besten” zu einer aktuellen Signatur passt. Eine “scharfe” Suche, bei der formale und aktuelle Signatur exakt übereinstimmen, ist nicht möglich, da durch Vererbung Subtypen der für die formalen Parameter definierten Typen verwendet werden können. Der Algorithmus besteht aus vier Stufen. Auf jeder Stufe des Algorithmus werden Methoden ausgewählt, die möglicherweise aufgerufen werden. Bleibt nach einer Stufe nur noch eine Methode übrig, dann wurde die aufgerufene Methode gefunden. Der Algorithmus ist in Abbildung 20 dargestellt.

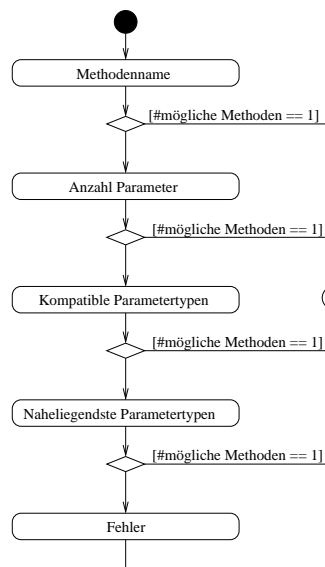


Abbildung 20: Aktivitätsdiagramm des Algorithmus für die Auflösung von Methodenaufrufen

Die einzelnen Schritte des Algorithmus sind wie folgt:

1. Im ersten Schritt werden alle Methoden ausgewählt, deren Name gleich der gerufenen Methode ist. Ist die gerufene Methode nicht überladen, endet der Algorithmus bereits nach diesem Schritt, da nur eine Methode den gleichen Namen wie die aufgerufene Methode trägt.
2. Ist die gerufene Methode überladen, dann werden zunächst alle Methoden aussortiert, bei denen die Anzahl der formalen Parameter nicht mit der Anzahl der aktuellen Parameter übereinstimmt. In Java ist dieser Schritt möglich, da es in Java nicht möglich ist, Parameter mit Standardwerten vorzubelegen. Könnten formale Parameter mit Standardwerten vorbelegt werden, wie zum Beispiel in C++, dann müsste die Anzahl der formalen Parameter nicht genau mit der Anzahl der aktuellen Parameter übereinstimmen.
3. Im dritten Schritt werden Methoden untersucht, die überladen sind und die gleiche Anzahl von Parametern haben. Für jede Methode wird nun geprüft, ob ihr formales Parameterprofil zu dem gegebenen aktuellen Parameterprofil passt. Ein Parameterprofil ist passend, wenn im formalen Parameterprofil ein Typ verzeichnet ist, von dem ein Subtyp an gleicher Stelle im Parameterprofil steht.

4. Wurden im dritten Schritt mehrere Methoden mit kompatibelem formalem Parameterprofil gefunden, wird im vierten Schritt nun das am besten passende formale Parameterprofil gesucht. Ein Parameterprofil ist am besten passend, wenn der Abstand im Vererbungsbaum zwischen aktuellem und formalem Typ eines Parameters kleiner ist als die Abstände aller übrigen formalen Parameterprofile zum aktuellen Parameterprofil.
Der Abstand zwischen formalem und aktuellem Parameter im Vererbungsbaum wird von aktuellem Parametertyp hin zum formalen Parametertyp berechnet. Der formale Parametertyp muss immer ein Supertyp des aktuellen sein.
Bei dieser Suche wird nur der erste am besten passende Parametertyp von links nach rechts beachtet. Wird ein passendes formales Parameterprofil gefunden, ist die gerufene Methode gefunden worden.
5. Kommt der Algorithmus im fünften Schritt an, dem Fehlerfall, konnte keine Methode gefunden werden oder es stehen mehrere Methoden zur Auswahl. Dieser Fall darf nicht möglich sein, da durch das Java-Frontend bereits eine semantische Analyse durchgeführt wurde.

4.3.2 Lokale Variablen

Lokale Variablen, die in einer Methode definiert werden, werden von der Klasse `VarStack` verwaltet. Die Klasse `VarStack` stellt einen Stack für lokale Variablen zur Verfügung.

Zu Beginn der Übersetzung einer Methode werden alle Attribute der Klasse, in der die Methode definiert ist, auf der untersten Ebene des Stacks abgelegt. Für jeden Anweisungsblock der Methode wird ein neuer Eintrag auf dem Stack erzeugt und alle lokalen Variablen, die in dem Block definiert sind, dem obersten Eintrag auf dem Stack hinzugefügt. Bei Verlassen eines Anweisungsblocks wird die oberste Plattform des Stacks mit allen in diesem Block enthaltenen lokalen Variablen vom Stack entfernt. Ist die Übersetzung einer Methodenimplementierung nach IML abgeschlossen, wird der Variablenstack vollständig geleert.

Beim Suchen einer lokalen Variable, zum Beispiel für die Erzeugung einer Zuweisung an die Variable in IML, wird der Stack lokaler Variablen von oben nach unten durchsucht, bis die Variable gefunden wird. Die Suche nach einer lokalen Variable wird über ihren Namen vorgenommen. Die erste Variable, die den gesuchten Namen trägt, wird als Suchergebnis zurückgeliefert. Durch Verwendung eines Stacks für die Verwaltung lokaler Variablen, ist das Überschieben lokaler Variablen einfach möglich.

Kapitel 5 Test der Implementierung

Der Test der Implementierung soll die Korrektheit von `jafe` sicherstellen. Korrektheit bedeutet hier eine korrekte Darstellung eines Java-Programms in einem IML-Graph. Ein korrekter IML-Graph, der für ein Java-Programm zu erzeugen ist, wird durch die Spezifikation der IML-Erweiterung für Java vorgegeben. Die Spezifikation der IML-Darstellung von Java-Programmen wurde in Kapitel 3 vorgestellt.

Dieses Kapitel beschreibt den Test von `jafe`. Beschrieben werden das Vorgehen, das Regressionstestwerkzeug `jafetest` und die Ergebnisse des Tests.

5.1 Testvorgehen

Der Test von `jafe` war zweigeteilt. Für die Entwicklung wurden Wegwerftests durchgeführt. Wegwerftests wurden gewählt, da sie von mir als geeignet für die Entwicklung erachtet wurden. Nach Abschluss der Entwicklung einer ersten Version wurde ein Werkzeug für den Regressionstest entwickelt. Zu dem Regressionstestwerkzeug wurde eine umfangreiche Regressionstestsuite erstellt. Eine Regressionstestsuite besteht aus einer Menge von Testfällen mit Soll-Ergebnissen. Im Falle von `jafe` sind die Testfälle kleine Java-Programme und die Soll-Ergebnisse sind IML-Graphen in HTML-Darstellung. Mit dem Regressionstestwerkzeug wurde ein reproduzierbarer Test durchgeführt.

Die beiden folgenden Abschnitte beschreiben beide Arten der durchgeführten Tests. Im Anschluss an die beiden Abschnitte folgt eine Betrachtung des Testvorgehens.

5.1.1 Test während der Entwicklung

Während der Entwicklung wurden kleine Java-Programme erstellt, die für den Test der zuletzt oder aktuell implementierten Funktionalität verwendet wurden. Die Testprogramme wurden von `jafe` in einen IML-Graph übersetzt. Für die Kontrolle des IML-Graph wurde dieser mit Hilfe des Bauhaus-Werkzeugs `iml2html` in eine HTML-Darstellung übersetzt. Die HTML-Darstellung wurde manuell überprüft. Wurde der IML-Graph korrekt erzeugt, beziehungsweise die HTML-Darstellung war korrekt, dann wurde die Fehlerkorrektur oder die Entwicklung der aktuell implementierten Funktionalität beendet und eine neue Funktionalität in `jafe` implementiert. Nach einem erfolglosen Test wurden die Testprogramme nicht weiter verwendet. Es handelte sich also um einen Wegwerftest.

5.1.2 Regressionstest

Nach Fertigstellung einer ersten Version von `jafe` wurde ein Werkzeug für Regressionstests entwickelt. Das Regressionstestwerkzeug wurde `jafetest` benannt. Dieses Werkzeug durchsucht eine Verzeichnisstruktur nach Java-Dateien. Jede gefundene Java-Datei, für die Soll-Ergebnisse existieren, wird in einen IML-Graph übersetzt. Der IML-Graph wird durch `iml2html` in eine HTML-Darstellung übersetzt. Die Erzeugung der HTML-Darstellung kann nur ausgeführt werden, wenn der IML-Graph erzeugt werden konnte. Die erzeugte HTML-Darstellung des IML-Graph wird mit einer vorgegebenen HTML-Darstellung des IML-Graph verglichen. Der Vergleich der HTML-Darstellungen erfolgt mit Hilfe des `diff`-Programms, das zwei Dateien vergleichen kann. Sind die beiden Darstellung gleich, dann war der Test erfolglos und `jafe` arbeitet korrekt. Unterscheiden sich die beiden HTML-Darstellungen, ist der Test erfolgreich und `jafe` arbeitet fehlerhaft. Die Arbeitsweise von `jafetest` ist in Abbildung 21 dargestellt.

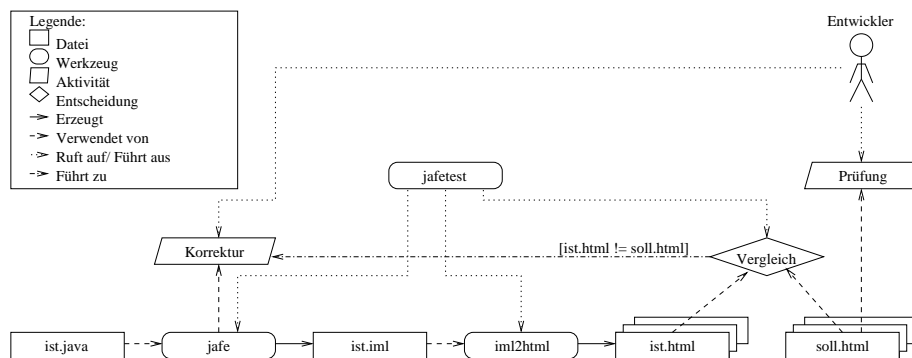


Abbildung 21: Arbeitsweise von `jafetest`

Die Ergebnisse der durchgeführten Tests werden in Abschnitt 5.2 vorgestellt. Zunächst werden die beiden Vorgehensweisen für den Test von `jafe` betrachtet.

5.1.3 Vergleich der Vorgehensweisen für den Test

Die Art des Vorgehens für den Test beruhte auf der Projektplanung. In der Projektplanung war ein ausführlicher Test nach Fertigstellung einer ersten Version von `jafe` geplant. Bei der Planung wurde nicht berücksichtigt, dass bereits während der Entwicklung Tests der aktuell implementierten Funktio-

nalität durchgeführt werden müssen. Die Zweiteilung der Tests während und nach der Entwicklung erwies sich rückblickend als nicht vorteilhaft.

Nicht vorteilhaft war, dass während der Entwicklung nur die im Moment implementierte Funktionalität getestet wurde. Eine Auswirkung der neu hinzugekommenen Implementierung zu der ursprünglichen, wurde nicht geprüft. Eine Prüfung wäre möglich gewesen, wenn das Regressionstestwerkzeug frühzeitig entwickelt worden wäre. Die Testfälle für die aktuell erweiterte Funktionalität hätten der Regressionstestsuite hinzugefügt werden können. Nach der Implementierung jeder neuen Funktionalität hätte dann durch Ausführung aller gesammelten Tests in der Regressionstestsuite die Korrektheit der neuen und der bereits bestehenden Funktionalität überprüft werden können.

Ein weiterer Nachteil der Durchführung des systematischen Tests nach der Implementierung ist die Erstellung der Testdaten. Zunächst muss eine möglichst vollständige Regressionstestsuite erstellt werden. Das bedeutet, dass kleine Java-Programme implementiert werden müssen, um möglichst alle Spracheigenschaften von Java abzudecken. Für jedes einzelne dieser Programme muss ein Soll-Ergebnis entwickelt werden. Der Soll-Ergebnis besteht aus den Ergebnissen für einen erfolglos ausgeführten Test. Jeder einzelne dieser Datensätze muss auf Korrektheit überprüft werden, um Regressionstests zu ermöglichen.

Wäre die Regressionstestsuite parallel zur Implementierung entwickelt worden, wäre der Aufwand für Testfall- und Soll-Ergebnisserstellung über die Entwicklung verteilt gewesen. Die aufwändige Implementierung der Testfälle und die Prüfung der Soll-Ergebnisse wäre mit dieser Vorgehensweise für den Entwickler angenehmer verlaufen. Der Aufwand für die Entwicklung bleibt natürlich der gleiche.

Aus den vorgestellten Erfahrungen ergibt sich, dass eine frühzeitige Entwicklung eines Regressionstestwerkzeugs und einer Entwicklung der Regressionstestsuite parallel zur Implementierung von Vorteil ist. Der Aufwand für die Erstellung der Regressionstestsuite verteilt sich und Auswirkungen neu implementierter Funktionalität auf die bestehende Implementierung können geprüft werden.

5.2 Testergebnisse

Die in diesem Abschnitt vorgestellten Testergebnisse beziehen sich ausschließlich auf den Regressionstest. Für die während der Entwicklung durchgeführten Wegwerftests wurden keine Daten gesammelt. Ebenso sind die durchgeführten Wegwerftests nicht reproduzierbar.

Die Regressionstestsuite von `jafetest` besteht aus 340 Testfällen. Die Testfälle sind kleine Java-Programme, die jeweils einen kleinen Teil der Eigenschaften der Java-Programmiersprache implementieren. Die Testfälle wurden unter Verwendung der Java-Sprachspezifikation [Gosling et al. 2000] entwickelt. Es wurde darauf geachtet, dass möglichst alle Ausprägungen korrekter Java-Programme berücksichtigt sind. Für die 340 Testfälle wurden Soll-Daten erstellt. Die Soll-Ergebnisse beschreiben einen korrekten IML-Graph, der aus dem Testfall von `jafe` erzeugt werden soll. Die erste Version von `jafe` wurde mit `jafetest` unter Verwendung der erstellten Testfälle getestet.

Die Ausführung der 340 Testfälle ergab 14 Fehler, die zu einem Abbruch der IML-Graph-Erzeugung führen. Die Fehler sind vor allem durch unvollständige Typinformation bedingt. So ist zum Beispiel der Typ ererbter Methoden oder Attribute oder von Parametern nicht immer bekannt.

Die Prüfung der Soll-Ergebnisse auf einen korrekt erzeugten IML-Graph ergab, dass für die 340 Testfälle 286 IML-Graphen korrekt erzeugt wurden. In 54 Fällen wurde der IML-Graph nicht korrekt erzeugt und eine Korrektur an `jafe` muss durchgeführt werden.

Die Behebung der aufgetretenen Fehler steht für die nächste Version von `jafe` im Vordergrund. Bei der Behebung der Fehler kann dann sofort `jafetest` mit der Regressionstestsuite für reproduzierbare Tests verwendet werden.

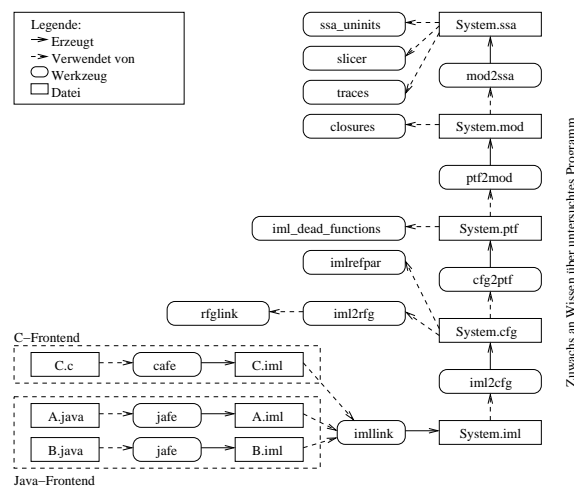
Das Auftreten der oben angeführten Fehler zeigt, dass die während der Implementierung durchgeführten Wegwerftests einen systematischen Test nicht ersetzen können. Der zum Schluss durchgeführte systematische Test deckt jedoch verbliebene Fehler auf. Mit den Ergebnissen des systematischen Tests kann eine zielgerichtete Korrektur von `jafe` durchgeführt werden.

Kapitel 6 Auswirkungen auf Bauhaus-Werkzeuge

Die Erweiterungen der IML für Java wirken sich auf die Werkzeuge in Bauhaus aus. Die Auswirkungen sind in diesem Kapitel beschrieben. Zunächst werden die einzelnen Werkzeuge kurz vorgestellt. Im Anschluss daran, werden die Auswirkungen vorgestellt.

6.1 Werkzeuge

Die einzelnen Werkzeuge, die in Bauhaus für die Programmanalyse zur Verfügung stehen, sind in Tabelle 3 kurz vorgestellt. Die Abhängigkeiten zwischen den einzelnen Werkzeugen sind in Zeichnung 22 dargestellt. Abhängigkeiten entstehen, wenn ein Werkzeug die Ergebnisse eines anderen Werkzeugs benötigt.



Zeichnung 22: Abhängigkeiten der Bauhauswerkzeuge untereinander

Tabelle 3: Werkzeuge in Bauhaus

Werkzeug	Beschreibung
cafe	Übersetzt C-Programme in einen IML-Graphen.
cfg2ptf	Erzeugt Partial-Transfer-Functions (PTF) aus dem Kontrollflussgraph. Die PTFs werden benötigt für die Points-To-Analyse.
closures	Ermittelt Parameter, die an eine als Callback registrierte Funktion durchgereicht werden.
iml_dead_functions	Ermittelt garantiert nicht ausgeführte Funktionen in einem Programmsystem.
iml_refpar	Ermittelt Referenzparameter einer Funktion.
iml2cfg	Erzeugt aus einem IML-Graphen einen Kontrollflussgraphen.
iml2rfg	Erzeugt aus einem IML-Graphen einen Ressource-Flow-Graphen (RFG).
imllink	Linker für IML-Graphen.
mod2ssa	Erzeugt die SSA-Darstellung für ein Programm, basierend auf der Seiteneffektinformation.
ptf2mod	Erzeugt aus den PTFs Informationen über Seiteneffekte.
rfglink	Linker für RFG-Graphen.
slicer	Erstellt slices für ein Programm.
ssa_lifetime	_a
ssa_uninits	Ermittelt alle Variablen, die verwendet werden, ohne zuvor initialisiert worden zu sein.
traces	Erkennung von Funktionen, die auf ein Objekt angewendet werden.

a. Der Entwickler dieses Werkzeugs konnte nicht befragt werden.

6.2 Auswirkungen auf die Werkzeuge

Die Auswirkungen auf die Bauhauswerkzeuge wurden in Gesprächen mit den Entwicklern erörtert.

In jedem Gespräch wurde dem Entwickler die erweiterte IML vorgestellt. Im Anschluss an die Vorstellung der erweiterten IML wurden notwendige Änderungen diskutiert. Basierend auf den notwendigen Änderungen wurde von dem Entwickler eine Schätzung des Aufwands erbeten. Die Aufwandschätzung soll einen groben Anhaltspunkt liefern, wie lange die Anpassung in Anspruch nehmen wird. Die Aufwandsschätzung umfasst den Zeitraum, der benötigt wird, um die Änderungen zu implementieren.

Die Ergebnisse der Gespräche sind in Tabelle 4 zusammengefasst.

Tabelle 4: Anpassungen der Bauhauswerkzeuge an die erweiterte IML

Werkzeug	Anpassungen	Aufwand (Tage)
cafe	<ul style="list-style-type: none"> Konstruktoraufrufe für die Erzeugung von IML-Knoten müssen angepasst werden. Die Anpassung ist notwendig für Knotentypen denen Attribute hinzugefügt wurden. Im Einzelnen sind dies die Knoten: SymNode, OCNode, TCNode, TC_Routine und OC_Variable. 	0,5
cfg2ptf	<ul style="list-style-type: none"> Änderungen an diesem Werkzeug sind abhängig von Änderungen am Kontrollflussgraphen. Aufrufe virtueller Methoden müssen behandelt werden. Das Konzept der "Blocks" und "Cells" für die PTF-Berechnung muss für die Darstellung von Klassen und Interfaces erweitert werden. 	15
closures	<ul style="list-style-type: none"> Die Anwendung des closures Werkzeugs macht, nach Angabe des Entwicklers, keinen Sinn für Java-Programme. Da es in Java keine Zeiger auf Funktionen gibt über die Callbacks registriert werden können. 	keine Angabe (k. A.)
iml_dead_functions	<ul style="list-style-type: none"> Aufrufe virtueller Funktionen müssen von dem Werkzeug berücksichtigt werden. 	2
iml_refpar	<ul style="list-style-type: none"> Die Anwendung des iml_refpar Werkzeugs macht, nach Angabe des Entwicklers, keinen Sinn für Java-Programme. 	k. A.
iml2cfg	<ul style="list-style-type: none"> Für jeden neuen Knoten im HPG muss eine visit()-Routine implementiert werden. Die visit()-Routine implementiert die Erzeugung des Kontrollflussgraphen (CFG). 	1

Tabelle 4: Anpassungen der Bauhauswerkzeuge an die erweiterte IML

Werkzeug	Anpassungen	Aufwand (Tage)
iml2rfg	<ul style="list-style-type: none"> Für die Darstellung von C-Programmen, in der erweiterten IML, müssen keine Änderungen vorgenommen werden. Für die Darstellung von Java-Programmen im RFG, muss zunächst die Information festgelegt werden, die im RFG repräsentiert werden soll. Da zunächst konzeptionelle Arbeit notwendig ist, kann eine Aufwandsabschätzung nicht angegeben werden. 	k. A.
imllink	<ul style="list-style-type: none"> Der Linker muss auf die Verwendung der "Mangled Names" umgestellt werden. Einbeziehung der Vererbungsrelationen in den Link-Vorgang. Erzeugung von Dummy-Knoten für Typen, die beim Linkvorgang nicht gefunden werden. 	5
mod2ssa	<ul style="list-style-type: none"> Änderungen an diesem Werkzeug sind abhängig von den Änderungen der SSA-Form. Die SSA-Form muss für die Darstellung von Ausnahmen eventuell erweitert werden. 	10
ptf2mod	<ul style="list-style-type: none"> Dieses Werkzeug betreffende Änderungen hängen von den Änderungen des Kontrollflussgraph und der PTFs ab. 	5
rfglink	<ul style="list-style-type: none"> Änderungen an diesem Werkzeug hängen von den Änderungen der RFG-Darstellung ab. Da die RFG-Darstellung für Java-Programme nicht klar ist, können für dieses Werkzeug keine notwendigen Änderungen angegeben werden. 	-
slicer	<ul style="list-style-type: none"> Dieses Werkzeug betreffende Änderungen hängen ab von den Änderungen des Kontrollflussgraph und der Datenflussanalyse. Ändert sich die Darstellungen des Kontrollflussgraph oder des Datenfluss nicht, sind keine Änderungen notwendig. 	k. A.
ssa_lifetime	_a	k. A.
ssa_uninits	<ul style="list-style-type: none"> Änderungen an diesem Werkzeug sind abhängig von den Änderungen der SSA-Form. Die SSA-Form muss eventuell für die Darstellung von Exceptions erweitert werden. 	k. A.

Tabelle 4: Anpassungen der Bauhauswerkzeuge an die erweiterte IML

Werkzeug	Anpassungen	Aufwand (Tage)
traces	<ul style="list-style-type: none"> • Änderungen an diesem Werkzeug hängen vom Kontrollflussgraphen und der Points-To-Analyse ab. • Primitive Funktionen müssen für Java definiert werden. 	0.5

- a. Der Entwickler dieses Werkzeugs konnte nicht befragt werden.

6.3 Bewertung der Auswirkungen

Wie durch die Interviews der Entwickler festgestellt werden konnte, kann die Anpassung der Werkzeuge an die IML für Java in kurzer Zeit durchgeführt werden. Schwierigkeiten bei der Anpassung der Funktionalität der Werkzeuge an die erweiterte IML werden von den Entwicklern für die Anpassung nicht erwartet.

Mögliche Schwierigkeiten werden von den Entwicklern jedoch bei der Berücksichtigung neuer Konzepte gesehen. Diese neuen Konzepte, die dann auch durch die Bauhaus-Werkzeuge behandelt werden sollen, liegen im Bereich der Ausnahmebehandlung und dem Aufruf virtueller Funktionen.

6.3.1 Auswirkung der Modellierung von Ausnahmen

Auswirkung auf alle Bauhauswerkzeuge hat die Modellierung von Ausnahmen im IML-Graph. Von allen Entwicklern wurde die Behandlung von Ausnahmen als größtes Problem genannt. Eine Anpassung aller Bauhauswerkzeuge ist für die Ausnahmebehandlung notwendig. Bei der Ausnahmebehandlung sind für die Entwickler die folgenden Punkte unklar:

- Wie werden Ausnahmen im Kontrollflussgraph behandelt?
- Welche Ausnahmen werden im Kontrollflussgraph berücksichtigt (Benutzerdefinierte Ausnahmen, Ausnahmen der virtuellen Maschine)?
- Muss die SSA-Darstellung für Ausnahmen erweitert werden?
- Wie wirken sich Ausnahmen auf die Berechnung der Partial-Transfer-Functions (PTF) aus?
- In welcher Weise müssen Ausnahmen für das Program-Slicing berücksichtigt werden?

Eine Erweiterung der Werkzeuge, um die Fähigkeit der Ausnahmebehandlung, wird von allen Entwicklern als problematisch eingestuft. Vor der Erweiterung der Werkzeuge um die Ausnahmebehandlung müssen zunächst konzeptionelle Vorarbeiten geleistet werden, da für die Behandlung von Ausnahmen im Moment kein großes Wissen vorhanden ist. Die konzeptionelle Vorarbeit besteht im Wesentlichen aus der Klärung der oben angeführten offenen Fragen.

6.3.2 Auswirkungen von Aufrufen virtueller Funktionen

Die Aufrufe virtueller Funktionen in Java stellen die Analysewerkzeuge von Bauhaus vor das Problem, dass zunächst festgestellt werden muss welche Methode tatsächlich gerufen wird. Ist es möglich, die gerufene Methode eindeutig zu ermitteln, können die Analysewerkzeuge wie gewohnt arbeiten.

Falls es nicht möglich ist die gerufene Methode zu ermitteln, müssen die Analysewerkzeuge mehrere Methoden als Ziel eines Aufrufs betrachten. Die Ergebnisse der Analysen werden bei konservativem Vorgehen umfangreicher, oder, zum Beispiel bei Verwendung möglicher Heuristiken, ungenauer.

Die Entwickler sehen bei der Behandlung dieses Problems keine größeren Hindernisse. Zunächst kann der Aufruf virtueller Funktion ähnlich dem Funktionsaufruf über Funktionszeiger behandelt werden. Für das Finden möglicherweise gerufener Funktionen über Funktionszeiger, wird die Points-To-Analyse eingesetzt. Für das Finden möglicher gerufener virtueller Funktionen muss ein Algorithmus entwickelt werden, der alle möglicherweise gerufenen virtuellen Funktionen ermittelt.

Kapitel 7 Ausblick und Bewertung

Nachdem die Diplomarbeit “Erweiterung und Generierung der Zwischensprache IML für Java Programme” ausführlich vorgestellt wurde, soll nun ein Ausblick auf die mögliche Zukunft von `jafe` gegeben werden. Anschließend wird die Diplomarbeit und die in deren Rahmen durchgeführten Arbeiten einer kritischen Bewertung unterzogen.

7.1 Ausblick auf zukünftige Entwicklung und Anwendung

In den folgenden Abschnitten wird ein Blick in die Zukunft gewagt. Zunächst wird beschrieben, welche Arbeiten für die vollständige Integration von `jafe` in Bauhaus noch nötig sind. Im Anschluss daran werden einige Ideen vorgestellt, die auf Basis der Bauhaus-Werkzeuge realisiert werden könnten.

7.1.1 Einsatz im Rahmen des Bauhaus Projekts

Um `jafe` zu einem vollwertigen Bestandteil der Bauhaus-Werkzeuge zu machen, muss `jafe` zunächst in die Bauhaus-Werkzeuge vollständig integriert werden. Die Integration umfasst die Verschmelzung der IML für Java und der IML für C und die Anpassung der Bauhaus-Werkzeuge.

Sind die IML-Varianten für C und Java verschmolzen, müssen die Bauhaus-Werkzeuge für die Verwendung der neuen IML angepasst werden. Das Vorgehen für die Anpassung und die notwendigen Arbeiten wurden bereits in Kapitel 6 beschrieben. Mögliche Erweiterungen der IML-Werkzeuge für die Untersuchung von Java-Programmen wurden ebenfalls bereits in Kapitel 6 beschrieben.

7.1.2 Weiterentwicklung von jafe

Für eine Weiterentwicklung von jafe müssen zunächst die während des Tests gefundenen Fehler behoben werden. Nach der Fehlerbehebung ist jafe uneingeschränkt für die Übersetzung von Java-Programmen nach IML bis hin zur Java-Version 1.3 geeignet.

Eine erste Weiterentwicklung von jafe ist die Implementierung der Modellierung der assert-Anweisung. Assert-Anweisungen sind seit Java 1.4 im Sprachstandard aufgenommen. Für die Modellierung der assert-Anweisung muss jikes in der Version 1.16 verwendet werden. Bei Verwendung von jikes 1.16 muss die jafe-Implementierung an die neue Version von jikes angepasst werden. Jikes in Version 1.16 wurde nicht für jafe verwendet, da diese Version erst während der Implementierung von jafe veröffentlicht wurde.

Eine zweite Weiterentwicklung ist die statische Bindung von Aufrufen final deklarerter Methoden. Durch die statische Bindung der Methodenaufrufe kann die Anzahl virtueller Methodenaufrufe verringert werden. Durch die Verringerung der virtuellen Methodenaufrufe können die Bauhaus Werkzeuge bei Analysen genauer und schneller arbeiten.

Eine dritte Weiterentwicklung ist die Aufteilung des Extends_Relation-Knotentyps. Mit diesem Knotentyp werden im Moment Vererbungs- und Implementierungsbeziehungen modelliert. Eine Aufteilung des Knotens in einen Extends_Relation- und einen Implements_Relation-Knotentyp unter einem gemeinsamen Type_Relations-Knotentyp würde im IML-Graph den Rückschluss auf eine Implementierungs- oder Vererbungsbeziehung ermöglichen.

Die Verfügbarkeit der Regressionstestsuite für jafe ermöglicht es, dass nach jeder Erweiterung oder Anpassung von jafe, die Korrektheit von jafe überprüft werden kann. Der Test zukünftiger Erweiterungen und Anpassungen wird durch die Regressionstestsuite vereinfacht.

7.1.3 Unterstützung der Softwareentwicklung

Die Bauhaus-Werkzeuge können nicht nur für Reengineering-Aufgaben eingesetzt werden. Eine Erweiterung um Werkzeuge für die Unterstützung der Entwicklung ist ebenfalls denkbar.

Eine mögliche Erweiterung ist zum Beispiel die automatische Auswahl von Regressionstests für Java-Programme. Aus einer Anzahl Testfälle, die Klassen zugeordnet sind, werden bei Änderung oder Erweiterung der Implementierung die auszuführenden Testfälle ausgewählt. In Bauhaus könnte ein solches Werkzeug unter Verwendung der Darstellung von Java-Programmen in IML implementiert werden. Eine Technik für die automatische Auswahl von Regressionstests ist in [Rothermel, Harrold 1994] beschrieben.

Ein weiteres denkbare Werkzeug kann zum Beispiel die Auswirkungen von Änderungen in einem Java-Programm anzeigen. Der Entwickler spezifiziert die Änderungen, die er an der Implementierung vornehmen möchte und das Werkzeug zeigt dem Entwickler mögliche Auswirkungen auf die bestehende Implementierung an. Ein Werkzeug, das eine solche Funktion implementiert, ist in [Kung et al. 1994] beschrieben.

7.1.4 Unterstützung des Software-Reengineering

Für die Unterstützung des Software-Reengineering von Java-Programmen ist zum Beispiel die Erkennung von Design-Patterns [Gamma et al. 1995] in Java-Programmen vorstellbar. Die Verwendung von Design-Patterns in der Softwareentwicklung hat in den letzten Jahren einen Boom erlebt. Ebenso eignen sich Design-Patterns zur Diskussion von Softwareentwürfen. Mögliche Anwendungszwecke für das Software-Reengineering sind in [Krämer, Prechelt 1996] beschrieben. [Paakki et al. 2000] beschreibt die Implementierung eines Werkzeugs für die Erkennung von Design-Patterns.

7.2 Bewertung der Diplomarbeit

In dieser Diplomarbeit wurde ein Softwarewerkzeug entwickelt, mit dem es möglich ist, Java-Programme in eine IML-Darstellung zu übersetzen. Die Diplomarbeit war unterteilt in verschiedene Tätigkeiten, die alle das Ziel der Entwicklung des Softwarewerkzeugs verfolgten. Zu Beginn der Diplomarbeit wurde ein Projektplan erstellt und die einzelnen Aufwände für die durchzuführenden Arbeiten geschätzt. Die einzelnen Teilaufgaben und deren zeitliche Planung ist in Abbildung 23 dargestellt.

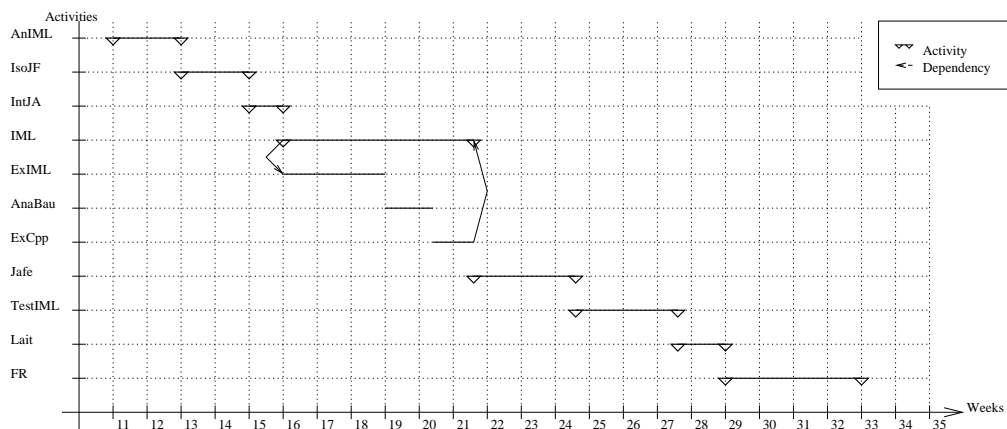


Abbildung 23: Geplante Zeiträume für die Teilaufgaben

Die einzelnen Teilaufgaben, die in Abbildung 23 dargestellt sind, waren wie folgt:

- Analyse der IML für C-Programme (AnIML)
Ermittlung der Struktur und Semantik der IML für die Darstellung von C-Programmen. Das Vorgehen und die Ergebnisse sind in Abschnitt 3.1 beschrieben.
- Isolation des Frontend aus dem Java-Compiler (IsoJF)
Abtrennung des Frontends vom Java-Compiler, da für die IML-Erzeugung nur der AST und die Symboltabelle benötigt wird. Letztendlich wurde das Frontend nicht isoliert, da eine einfache Integration der IML-Erzeugung möglich war. Wäre das Frontend abgetrennt worden, hätten umfangreichere Änderungen an `jikes` durchgeführt werden müssen. Die Änderungen, die an `jikes` durchgeführt wurden, beschränken sich auf 31 Quelltext-Zeilen. Diese Zeilen umfassen das Einführen eines neuen Aufrufparameters (`-iml`) für die Erzeugung der IML und den Aufruf des

Java-nach-IML-Übersetzers. Außerdem wurde in die Symboltabelle noch eine Funktion eingefügt. Diese Funktion liefert den “gemangelten”-Namen einer Variablen. Alle Erweiterungen sind durch Präprozessoranweisungen eingeschlossen, die es ermöglichen, `jikes` ohne IML-Erweiterungen zu übersetzen.

- Schnittstelle Ada95-C++ (IntJA)
Entwicklung einer Schnittstelle, um von der C++-Implementierung des Java-Frontends auf die Ada95-Implementierung der IML zugreifen zu können. Diese Tätigkeit ist ausführlich in Abschnitt 4.1 beschrieben.
- Erweiterung der IML für Java (ExIML)
Spezifikation der Erweiterung der IML für die Darstellung von Java-Programmen. Eine ausführliche Beschreibung findet sich in Kapitel 3.
- Analyse der Auswirkungen auf Bauhauswerkzeuge (AnaBau)
Untersuchung der Auswirkungen der IML-Erweiterungen für Java auf die bestehenden Bauhaus-Werkzeuge. Die Untersuchung der Auswirkungen ist detailliert in Kapitel 6 beschrieben.
- Erweiterung der IML für C++ (ExCpp)
Spezifikation der Erweiterungen der IML für die Darstellung von C++-Programmen. Die Erweiterung für C++-Programme wurde vorgenommen für sprachliche Bestandteile von C++, die auch in Java vorhanden sind. Erweiterungen der IML für C++ waren nötig, um die Verträglichkeit der IML-Erweiterungen für Java mit einer möglichen Erweiterung der IML für C++ gewährleisten zu können. Erweiterungen für C++ wurden gemeinsam mit den Erweiterungen für Java ausgeführt. Eine ausführliche Beschreibung der für C++ spezifizierten Erweiterung findet sich in Abschnitt 3.2.
- Implementierung von `jafe` (Jafe)
Implementierung des Übersetzers für Java-Programme nach IML.
- Test von `jafe` und Korrektur der Fehler (TestIML)
Test der Implementierung des Java nach IML Übersetzers.
- Anpassen des Linkers für IML (Lait)
Anpassen des IML-Linkers an die IML-Erweiterung für Java. Diese Tätigkeit war nicht Bestandteil der Aufgabenstellung oder notwendig für die Erreichung des Ziels der Diplomarbeit, wurde aber trotzdem eingeplant, um einen IML-Graph für Programmsysteme, die durch mehrere IML-Graphen dargestellt sind, erzeugen zu können. Aufgrund von Zeitmangel konnte die Anpassung des IML-Linkers nicht durchgeführt werden.
- Abschlussdokumentation (FR)
Erstellen dieses Abschlussberichts, in dem die Diplomarbeit dokumentiert ist.

Die ursprüngliche Zeitplanung, wie sie in Abbildung 23 dargestellt ist, erwies sich als zu optimistisch. Verzögerungen entstanden während den Tätigkeiten:

- Erstellen einer Schnittstelle zwischen Ada95 und C++.
- Implementierung des Java-nach-IML-Übersetzers.

Die aufgetretenen Verzögerungen konnten durch den eingeplanten Puffer kompensiert werden.

Gründe für die Verzögerung während der Implementierung wurden bereits in Kapitel 4 beschrieben.

Eine Verzögerung bei der Erstellung der Schnittstelle zwischen Ada95 und C++ trat auf, da die Inhalte der Aufgabe während der Projektplanerstellung nicht vollständig erfasst wurden. Bei der Schätzung wurde davon ausgegangen, dass es genügt, eine einfache Schnittstelle für die IML zu implementieren. Tatsächlich wurde aber ein vollständiges Konzept für die Integration einer Ada95-C++ Schnittstelle in den IML-Generator entwickelt. Die Entwicklung eines solchen Konzepts, das zum einen den Wartungsaufwand minimiert und zum anderen für weitere Implementierungen zur Verfügung steht, verdreifachte den geplanten Zeitraum für die Aufgabe.

Die tatsächliche Abfolge der Aufgaben und die tatsächlich benötigte Zeit für die Erfüllung einer Aufgabe sind in Abbildung 24 dargestellt. Verschiebungen von Teilaufgaben auf Grund von Verzögerungen während der Ausführung vorhergehender Aufgaben sind grau hinterlegt.

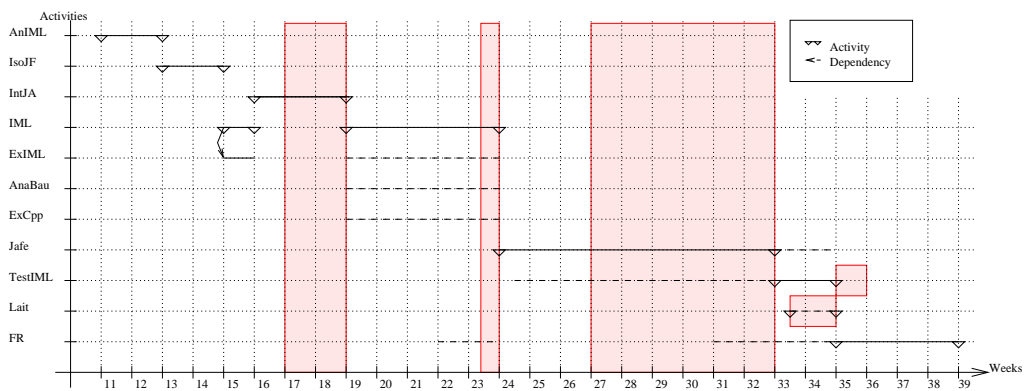


Abbildung 24: Tatsächlicher Zeitaufwand für die Realisierung der einzelnen Teilaufgaben

Eine genauere Zeitplanung im Bereich der Implementierungsaufgabe wäre hilfreich gewesen. Allerdings stellt sich die Frage, auf welcher Basis diese Zeitplanung hätte durchgeführt werden können. Wie bereits in Abschnitt 4.3 erwähnt, wäre die Realisierung eines Prototypen auch für die Zeitplanung sinnvoll gewesen.

Abschließend kann über die Diplomarbeit gesagt werden, dass durch die durchgeführten Arbeiten bei der Realisierung von `jafe` wertvolle Erfahrungen im Bereich Übersetzerbau gesammelt werden konnten, im Besonderen Erfahrungen im Bereich der Modellierung von Programmen mit Informationsstrukturen für Analyseaufgaben, wie zum Beispiel einem AST oder in der IML. Bleibt zu hoffen, dass die durchgeführte Diplomarbeit und deren Resultate als Ausgangspunkt für weitere Arbeiten in Bauhaus genutzt werden können, zum Beispiel für die Realisierung der im vorigen Abschnitt vorgestellten Erweiterungen oder einer Erweiterung der IML für die Darstellung von C++-Programmen.

Anhang A Glossar verwendeter Begriffe und Abkürzungen

Hierarchischer Programmgraph (HPG)

Teilgraph des IML-Graphen, der den abstrakten Syntaxbaum des modellierten Java-Programms darstellt. Die *Knoten* im HPG sind ausschließlich *HPG-Knoten*.

HPG-Knoten

siehe *IML-HPG-Knoten*

Intermediate Language (IML)

Typhierarchie für die Modellierung von Programmen in einem Graphen (*IML-Graph*). Der *IML-Graph* besteht aus dem *HPG* und der Symboltabelle aus *IML-Typknoten* und *IML-Objektknoten*.

IML-Darstellung

Darstellung eines Programms oder eines Programmtails in einem *IML-Graph*.

IML-Erzeugung

Erzeugen eines *IML-Graphen* für die Darstellung eines Programms.

IML-Generator

Übersetzer, der aus einer *IML-Spezifikation* eine *IML-Implementierung* erzeugt.

IML-Graph

Graph bestehend aus *IML-HPG-Knoten*, *IML-Objektknoten* und *IML-Typknoten*.

IML-HPG-Knoten

Knotentypen, die alle von dem *Knotentyp* `Value` ableiten. Mit Hilfe dieser Knoten wird der abstrakte Syntaxbaum des Programms aufgebaut.

IML-Implementierung

Ada95-Implementierung, die vom *IML-Generator* aus der *IML-Spezifikation* erzeugt wird. Die Implementierung erlaubt, aus den einzelnen Knoten, die in der *IML-Spezifikation* definiert wurden, einen *IML-Graph* zu erstellen.

IML-Klasse

siehe *IML-Knotentyp*

IML-Knoten

Einzelner Knoten im *IML-Graph*, Objekt eines *IML-Knotentyp*.

IML-Knotenhierarchie

Vererbungshierarchie der *IML-Knoten*.

IML-Knotentyp

Typ (Klasse), aus dem *IML-Knoten* für das Einfügen in den *IML-Graph* erzeugt werden können.

IML-Modellierung

Abbildung eines Programms in einen *IML-Graph*, siehe auch *IML-Erzeugung*.

IML-Objektknoten

Alle *IML-Knotentypen*, die von dem *IML-Knotentyp* `OCNode` erben. Durch diese Knoten wird eine Deklaration oder eine Entität modelliert.

IML-Repräsentation

siehe *IML-Modellierung*

IML-Spezifikation

Textuelle Beschreibung der *IML-Knotentypen*, deren Attribute und deren Vererbungsbeziehungen. Die Beschreibung ist die Spezifikation der IML und ist in einer eigenen Spezifikationsprache abgefasst.

IML-Typknoten

Alle *IML-Knotentypen* in der *IML-Knotenhierarchie*, die von dem *Knotentyp* `TCNode` erben.

Java-Übersetzer-Frontend

Teil des *Java-Übersetzers*, der die Symboltabelle, den abstrakten Syntaxbaum und die syntaktische und semantische Analyse enthält.

Java-Übersetzer

Übersetzt Java-Programme in Java-Bytecode gemäß der Java-Language-Specification [Gosling et al. 2000] und der Java-Virtual-Machine-Specification [Lindholm, Yelling 1999].

Knoten

siehe *IML-Knoten*

Knotentyp

siehe *IML-Knotentyp*

Objektknoten/ -klasse

siehe *IML-Objektknoten*

Typknoten/ -klasse

siehe *IML-Typknoten*

Übersetzer-Frontend

siehe *Java-Übersetzer-Frontend*

Anhang B Literatur

[Brooks 1995]

Frederick P. Brooks Jr.
“The Mythical Man Month: Essays on Software Engineering”
Addison Wesley Publishing Company, 1995, 2nd Ed.

[Comar, Dewar 1996]

Cyrille Comar, Robert B. K. Dewar
“Interfacing to Other Languages using GNAT and Ada95”
The Ada95 Ada-Europe Conference, April 1996

[Eisenbarth et al. 1999]

Thomas Eisenbarth, Rainer Koschke, Erhard Plödereder, Jean-Francois Girard, Martin Würthner
“Projekt Bauhaus: Interaktive und inkrementelle Wiedergewinnung von SW-Architekturen”
Workshop Software-Reengineering, Bad Honnef, Universität Koblenz-Landau, Fachberichte Informatik, Nr. 7-99, pp. 17-26, 1999

[FSF 1991]

“GNU General Public License”
Version 2, Free Software Foundation, June 1991

[FSF 1999]

“GNU Lesser General Public License”
Version 2.1, Free Software Foundation, February 1999

[Gamma et al. 1995]

Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides
“Design Patterns - Elements of Reusable Object-Oriented Software”
Addison Wesley Publishing Company, 1995

[Gosling et al. 2000]

James Gosling, Bill Joy, Guy Steele, Gilad Bracha
“The Java Language Specification”
2nd Ed., Sun Microsystems Inc., 2000

[Harrold, Sinha 1998]

Marry Jean Harrold, Saurabh Sinha
“Analysis of Programs with Exception-Handling Constructs”
Proc. Int. Conf. on Software Maintenance (ICSM), 1998

[Huitt, Wilde 1992]

Ross Huitt, Norman Wilde
“Maintenance Support for Object-Oriented Programs”
IEEE Trans. Software Eng., Vol.18, No. 12, Dec. 1992, pp. 1038-1044

[IBM 1999]

“IBM Public License”
Version 1.0, IBM Corp., 1999

[ISO/IEC 1995]

“Ada95 Reference Manual”
International Standard ISO/IEC Standard 8652:1995

[Kernighan, Ritchie 1990]

Brian W. Kernighan, Dennis M. Ritchie
“Programmieren in C”
Carl Hanser Verlag München, 1990, 2. Ausgabe

[Krämer, Prechelt 1996]

Christian Krämer, Lutz Prechelt
“Design Recovery by Automated Search for Structural Design Patterns in
Object-Oriented Software”
IEEE Proc. Working Conf. on Reverse Engineering, Nov. 1996

[Kung et al. 1994]

D. Kung, J. Gao, P. Hsia, F. Wen, Y. Toyoshima, C. Chen
“Change Impact Identification in Object Oriented Software Maintenance”
Proc. IEEE Int’l Conf. on Software Maintenance, 1994, pp. 2002-211

[Lindholm, Yelling 1999]

Tim Lindholm, Frank Yelling
“The Java Virtual Machine Specification”
2nd Ed., Sun Microsystems Inc., 1999

[Liskov 1988]

Barbara Liskov
“Data Abstraction and Hierarchy”
ACM SIGPLAN Notices 23(5), May 1988, pp. 17-34

[OMG 2001]

Object Management Group (OMG)
“Unified Modelling Language Specification”
Ver. 1.4, OMG 2001

[Paakki et al. 2000]

Jukka Paakki, Anssi Karhinen, Juha Gustafsson, Lilli Nenonen, A. Inkeri Verkamo
“Software Metrics by Architectural Pattern Mining”
Proc. International Conference on Software: Theory and Practice (16th IFIP World Computer Congress), 2002, pp. 325-332

[Rohrbach 1998]

Jürgen Rohrbach
“Erweiterung und Generierung einer Zwischendarstellung für C-Programme”
Studienarbeit Nr. 1662, Universität Stuttgart 1998

[Rothermel, Harrold 1994]

Gregg Rothermel, Mary Jean Harrold
“Selecting Regression Tests for Object-Oriented Software”
Proc. IEEE Int’l Conf. on Software Maintenance, 1994, pp. 14-25

[Wallrabe, Oestereich 1997]

Arne Wallrabe, Bernd Oestereich
“Smalltalk für Ein- und Umsteiger”
München; Wien: Oldenbourg 1997

Anhang C Ressourcen im Internet

[Link: javac]

Der Java Übersetzer von Sun Microsystems Inc.
<http://java.sun.com/j2se/1.3/>

[Link: jacks]

Jacks Automated Compiler Killing Suite
Testsuite mit Regressionstests für Java Compiler
<http://www-124.ibm.com/developerworks/oss/jacks>

[Link: gcj]

GNU Compiler for Java
Das Java-Übersetzer-Frontend für die GNU Compiler Collection
<http://gcc.gnu.org/java/>

[Link: jikes]

IBM Jikes Java Compiler
<http://www-124.ibm.com/developerworks/opensource/jikes/>

[Link: GPL]

GNU General Public License, Version 2
<http://www.gnu.org/licenses/gpl.txt>

[Link: LGPL]

GNU Lesser General Public License, Version 2.1
<http://www.gnu.org/licenses/lgpl.txt>

[Link: IPL]

IBM Public License
<http://oss.software.ibm.com/developerworks/opensource/license10.html>

[Link: Java Language Specification]

The Java Language Specification, Second Ed.

http://java.sun.com/docs/books/jls/second_edition/html/j.title.doc.html

[Link: JVM-Specification]

The Java Virtual Machine Specification, Second Ed.

<http://java.sun.com/docs/books/vmspec/2nd-edition/html/VMSpecTOC.doc.html>

[Link: UML-Specification]

Unified Modeling Language Specification 1.4

<http://www.omg.org/cgi-bin/doc?formal/01-09-67>

Anhang D Erklärung

Hiermit versichere ich, diese Arbeit selbständig verfasst und nur die angegebenen Quellen benutzt zu haben.

Markus Knauß