Design und Implementierung eines Bytecode-zu-C-Übersetzers für das Betriebssystem JX

Diplomarbeit im Fach Informatik

vorgelegt von

Jochen Reinwand

geb. am 12. November 1977 in Nürnberg

Angefertigt am

Lehrstuhl für Informatik 4 (Verteilte Systeme und Betriebssysteme)

Friedrich-Alexander-Universität Erlangen-Nürnberg

Betreuer: ***Prof. Dr.-Ing. W. Schröder-Preikschat***
Dipl. Inf. Christian Wawersich

Beginn der Arbeit: 1. April 2004
Abgabe der Arbeit: 1. Oktober 2004

A design goal of the Java operating system JX is to provide an operating system which makes the advantages of Java (e.g. type security) usable as flexible software-based memory protection. Nevertheless the important disadvantage of the smaller speed of a software-based implementation compared to a hardware-based one has been avoided as far as possible.

JX consists of a kernel and different packages, called components. The components are written in Java. The class files containing the Java bytecode are produced out of the Java source files using a normal Java compiler. Then they are translated into machine code with a special compiler and afterwards stored in a special file format making it possible for JX to load them.

During the process of the translation an intermediate representation of the bytecode is produced, which is used among other things for the optimization of the code. This intermediate representation can be displayed in the form of a pseudo programming language. This language has a lot of similarities with the programming language C.

Since the largest parts of the JX kernel are written in C, the idea came up, to use this similarities to produce C language code, making it possible to directly translate and link it with the JX kernel.

This proceeding brings up some advantages:

- Compiled this way JX consists only of a kernel, that boots faster than the normal JX. Loading class data and code as module files is not necessary. This is very handy for embedded systems.
- Since the Java code is not directly translated into a specific native machine code, JX can be ported more easily to other architectures. Only the parts of the kernel have to be adapted, that were written in assembler or otherwise specific to a certain architecture.
- Apart from the optimizations, already applied by the JX compiler, the C compiler has abilities to further enhance the output.
- Because of the abilities of the C compiler it is possible to implement functionalities, which are not yet completely supported when producing machine code directly, like the support of floating point data types.

Inhaltsverzeichnis

1 Einleitung **5**
1.1 Anmerkungen zur Textformatierung 6
1.2 Begriffsdefinitionen 6
1.2.1 Spezielle Fachbegriffe und Namen 6
1.2.2 JX spezifische Begriffe 7
1.3 Aufbau der Ausarbeitung 7

2 Modifizieren des Build-Systems **9**
2.1 Konzept des Build-Systems von JX 9
2.2 Vom bisherigen System nicht erfüllte Anforderungen 10
2.2.1 Globale Informationen 10
2.2.2 Optimierungen durch statisches Linken 11
2.3 Aufbau des modifizierten Build-Systems 11
2.3.1 Die wichtigsten Klassen 12
2.3.2 Die verwendeten Komponenten 12
2.3.3 Weitere Anmerkungen 13
2.4 Fehlende Funktionalitäten im neuen Build-System 13
2.4.1 Einschränkungen des normalen System 13
2.4.2 Nicht implementierte Merkmale 14

3 Linken gegen den Kern **15**
3.1 Primitive und höhere Datentypen 15
3.2 Laden der Komponenten durch JX 16
3.2.1 Statische und dynamische Datenstrukturen 16
3.2.2 Ladevorgang 16
3.3 Aufbau der relevanten Datenstrukturen im Kern 17
3.3.1 Komponenten, Klassen und Domains 17
3.3.2 Der Index aller Komponenten 18
3.3.3 Methodentabellen 19
3.3.4 Strukturen für Objekte 21
3.4 Support-Funktionen im Kern 21
3.5 Statische Erzeugung der Datenstrukturen 21
3.5.1 Beschreibende Strukturen 22
3.5.2 Primitive Klassen und `java.lang.Object` 24
3.5.3 Spezielle JX-Klassen 24
3.5.4 Arrays und Strings 25
3.5.5 Methodentabellen 25
3.5.6 Symbolische Konstanten und Performance 25

3.6 Datenfelder von Klassen und Objekten . . . 26
3.7 Methoden als Funktionen . . . 27
3.7.1 Argumente und Rückgabewerte . . . 28
3.7.2 Einschränkungen von Funktionen in C . . . 28
3.7.3 Eindeutige Funktionsnamen . . . 29
3.7.4 Methodenaufruf . . . 30
3.8 Exceptions . . . 33

4 Übersetzen des Bytecodes nach C 35
4.1 Konzept der IM-Code-Objekte . . . 35
4.2 Verwendung von Support-Funktionen des Kerns . . . 35
4.3 Unterschiede zwischen den Datentypen in C und Java . . . 36
4.3.1 Bit-Operationen . . . 36
4.3.2 Fließkomma-Arithmetik . . . 36
4.3.3 Umwandlung von Fließkomma- auf Integer-Typen . . . 37
4.4 Spezielle Probleme und Einschränkungen von C . . . 37
4.4.1 Variables `goto` . . . 38
4.4.2 Anweisungen und Ausdrücke . . . 40
4.5 Probleme mit Seiteneffekten . . . 42

5 Ergebnisse und Ausblick 43
5.1 Vorteile gegenüber dem bisherigen Maschinencode-Übersetzer . . . 43
5.1.1 Zusätzliche Optimierungen durch den C-Compiler . . . 43
5.1.2 Bessere Portierbarkeit . . . 43
5.1.3 Einsparungen durch statisches Linken . . . 44
5.1.4 Volle Unterstützung von `float` und `double` . . . 45
5.2 Nachteile der Übersetzung nach C . . . 45
5.2.1 Statischeres System . . . 45
5.2.2 Kern nicht für Nutzung von C-Code optimiert . . . 46
5.2.3 Einfache Garbage Collection . . . 46
5.3 Weitere Optimierungen . . . 46
5.3.1 Effizienteres Nutzen der statischen Datenstrukturen . . . 46
5.3.2 Für C optimierte Garbage Collection . . . 47
5.4 Weiterführende Ansätze . . . 47
5.4.1 Mehr statische Strukturen . . . 47
5.4.2 Mehr dynamische Strukturen . . . 47
5.4.3 Stärkere Ausrichtung auf C . . . 48

Literaturverzeichnis 49

Abbildungsverzeichnis 50

1 Einleitung

Ziel des Java-Betriebssystems JX ist es ein auf Java basierendes Betriebssystem bereitzustellen, das die Vorteile der Programmiersprache Java (z. B. Typsicherheit) für einen flexiblen softwarebasierten Speicherschutz nutzbar macht. Dabei sollte jedoch auch der bedeutenden Nachteil der geringeren Geschwindigkeit einer software- gegenüber einer hardwarebasierten Implementierung vermieden werden.

Hierfür wurde von Christian Wawersich wie in [1] beschrieben ein Java-Bytecode-zu-Maschinencode-Übersetzer auf der Basis der Arbeit von Hans Kopp (siehe [2]) implementiert. Der erzeugte Maschinencode wurde in erster Linie auf Geschwindigkeit hin optimiert.

Die vorher mit einem Java-Compiler aus den *Quelldateien* erzeugten *Klassendateien*, die den Java-Bytecode enthalten, werden vom Compiler in Maschinencode umgewandelt und dieser dann in einem spezielles Dateiformat zum Laden in JX gespeichert.

Bei der Übersetzung wird eine Zwischenrepräsentation erzeugt, die u.a. zur Optimierung des Codes benutzt wird. Diese Zwischenrepräsentation kann in Form einer Art Pseudo-Programmiersprache ausgegeben werden. Diese Sprache hat große Ähnlichkeiten mit der Programmiersprache C.

Da der größte Teil des Kerns des Betriebssystems JX in C geschrieben ist, führte dies zu der Idee, durch ein Ausnutzen dieser Ähnlichkeit den JX-Compiler zu benutzen um C-Code zu erzeugen, der sich direkt mit dem JX-Kern übersetzen und linken lässt.

Diese Vorgehensweise bringt einige Vorteile mit sich:

- JX besteht nun nur noch aus einem Kern, der sich schnell Booten lässt. Das Nachladen von Klassendaten und -code als Moduldateien entfällt. Dies ist vor allem für den Einsatz als Embedded-System sinnvoll.
- Da nicht direkt in nativen Maschinencode übersetzt wird, kann JX leichter auf andere Architekturen portiert werden. Nur die Teile des Kerns müssen angepasst werden, die in Assembler geschrieben wurden oder anderweitig an bestimmte Architekturen angepasst wurden.
- Neben den Optimierungen, die bereits durch den JX-Compiler angewandt werden, können noch die Fähigkeiten des nachgeschalteten C-Compilers genutzt werden.
- Durch die Fähigkeiten des C-Compilers wird es möglich, Funktionalitäten zu Implementieren, die beim Erzeugen von Maschinencode noch nicht vollständig unterstützt werden, wie z. B. die Unterstützung der Datentypen `float` und `double`.

Es ist noch zu erwähnen, dass es prinzipiell zwei verschiedene Vorgehensweisen beim Übersetzen von Java nach C gibt, die jedoch strikt zu trennen sind:

1. Die erste Methode besteht darin, direkt die Java-Quelldateien in die Programmiersprache C zu übersetzen. Dies bringt auf Grund der Unterschiede der beiden Programmiersprachen einige Probleme mit sich. Weiterführende Informationen zu diesem Ansatz sind in [3] und [4] zu finden.

2. Eine andere Möglichkeit vorzugehen, stellt das Übersetzen des Java-Bytecodes nach C dar. Der eigentliche Java-Quellcode wird also zuerst mit einem normalen Java-Compiler in Bytecode und dieser dann nach C übersetzt. Ein relativ junges Projekt (*Java C Virtual Machine* [5]) will auf Grundlage dieser Idee eine komplette Java Virtual Machine bereitstellen. Ein älteres Projekt stellt *Toba* [6] dar. Es wird jedoch nicht mehr weiter entwickelt und arbeitet nur mit der JDK-Version 1.1 zusammen.

Für eine Umsetzung in JX bietet sich eigentlich nur die zweite Variante an. Der JX-Compiler überlässt das Übersetzen des Java-Quellcodes in den Bytecode vollständig einem herkömmlichen Java-Compiler und die Zwischenrepräsentation bezieht sich direkt auf den Bytecode. Außerdem stellt der Bytecode keine so starke Abstraktion von der Hardware dar, wie der Java-Quellcode. Referenzen ersetzen z. B. bereits die direkte Handhabung höherer Datentypen wie Arrays. Eine Umsetzung der Maschinenbefehlen ähnelnden Bytecode-Befehle ist somit relativ leicht möglich.

1.1 Anmerkungen zur Textformatierung

Zur Hervorhebung werden Textpassagen, die Quellcode oder vergleichbares darstellen, wie Methodennamen, Funktionsnamen und Klassennamen, in `Schreibmaschinenschrift` dargestellt. Fachbegriffe werden beim ersten Einführen oder auch bei späteren Erwähnungen zur Hervorhebung in *Kursivschrift* dargestellt.

1.2 Begriffsdefinitionen

Um Verwechslungen zu vermeiden, sollen hier einige Hinweise zu bestimmten Fachtermini gegeben werden. Diese sind entweder auf Grund unterschiedlicher Benutzung in der Literatur leicht verwechselbar oder für JX spezifisch.

1.2.1 Spezielle Fachbegriffe und Namen

Methode und _Funktion_

Der Begriff *Methode* wird meist in der Objektorientierung dazu verwendet den Unterschied zur *Funktion* (als Begriff aus der prozeduralen Programmierung)[1] zu verdeutlichen. Ich nutze diese Unterscheidung, um die Trennung zwischen C und Java zu verdeutlichen. Von *Methoden* ist daher nur in Zusammenhang mit Java und von *Funktionen* nur in Zusammenhang mit C die Rede.

Ausnahme: Im Laufe der Ausarbeitung wird ersichtlich werden, dass die Java-Methoden beim Übersetzen in C-Funktionen umgesetzt werden. Hier wird es naturgemäß schwierig mit der Begrifflichkeit. An den nötigen Stellen wird deshalb falls erforderlich auf die jeweiligen Besonderheiten hingewiesen.

Felder, _Methoden_ und _Member_

Im Folgenden werden in Übereinstimmung mit [7] die Felder und Methoden eines Objekts als *Member* bezeichnet.

[1] Im Folgenden wird wie unter C üblich *keine* Unterscheidung zwischen den Begriffen *Funktion* und *Prozedur*, wie in anderen Programmiersprachen durchaus üblich, vorgenommen. Der Begriff *Prozedur* wird vermieden.

Vektor* und *Array, String* und *Zeichenkette

Der in der deutschen Literatur gebräuchliche Begriff *Vektor* wird synonym mit dem englischen *Array* benutzt. Eine Trennung zwischen Java und C findet nicht implizit, sondern nur explizit statt. Gleiches gilt für die Begriffe *String* und *Zeichenkette*.

Der *GNU C Compiler*

Zum Übersetzen von JX wird meist der *GNU C Compiler* [8] eingesetzt, der Teil der *GNU Compiler Collection* (*GCC*) ist. Im Folgenden wird jedoch, wie es bei früheren Versionen der GCC üblich war, die Abkürzung GCC stellvertretend für den *GNU C Compiler* benutzt.

Deklaration, Definition* und *Vereinbarung

In diesem Dokument werden die Begriffe *Deklaration*, *Definition* und *Vereinbarung* besonders im Umfeld von C so genutzt, wie es in [9] im deutschen Vorwort beschrieben wird:

> „(...) im Deutschen kann man *Deklarationen*, die Eigenschaften von Namen vereinbaren, völlig von *Definitionen* unterscheiden, die dazu auch noch Speicherplatz bereitstellen, denn für beide Vorgänge zusammen gibt es im Deutschen noch den Oberbegriff der *Vereinbarung.*“ [9, ix]

1.2.2 JX spezifische Begriffe

Komponenten

In JX werden die Klassen auf so genannte *Komponenten* aufgeteilt, die man auch als Module bezeichnen könnte. Sie stellen die kleinsten Einheiten dar, die dynamisch nachgeladen werden können. Gerade im Quellcode sind jedoch oft Funktionen, Methoden und Datenfelder die mit diesen Komponenten arbeiten daran zu erkennen, dass sie die Zeichenfolge `lib` enthalten. Im Folgenden wird immer von *Komponenten* gesprochen, um einer Verwechslung vorzubeugen. Eine fertig übersetzte Komponente liegt in Form einer Datei vor, die im Folgenden als *Komponentendatei* bezeichnet werden soll. Sie hat die Endung `.jll`. Deshalb wird, wie bei anderen Dateitypen, auch die Form `JLL-Datei` benutzt.

Build-Prozess

JX besitzt einen eigenen *Build-Prozess*, der die Komponenten baut, den Kern übersetzt und beides in einem Bootimage kombiniert. Der Begriff *Build-Prozess* wird im folgenden immer hierfür verwendet.

Kern

Wie bei Betriebssystemen üblich wird im Folgenden immer vom *Kern* die Rede sein, wenn der in C und Assembler implementierte grundlegende Teil des JX-Betriebssystems gemeint ist. Der englische Begriff *Kernel* wird vermieden.

1.3 Aufbau der Ausarbeitung

Die Arbeit am Bytecode-zu-C-Übersetzer teilte sich in zwei relativ stark getrennte Teilaufgaben auf. Deshalb sind auch die folgenden drei Kapitel genau diesen Bereichen gewidmet.

In Kapitel 2 wird das Build-System von JX unter die Lupe genommen. Zuerst soll ein kurzer Überblick über die Konzepte des Systems geboten werden. Diese führen zu Problemen und Unzulänglichkeiten des Systems wenn es als Basis für die Übersetzung des Bytecodes zu C

benutzt werden soll. Nach Erörterung der Probleme, wird das neue Build-System besprochen, das entworfen wurde, um den Übersetzungsprozess zu vereinfachen. Aber auch dieses neue System wird noch einer kritischen Betrachtung unterworfen.

Kapitel 3 widmet sich dann dem Problemfeld, das entsteht, wenn der Code dem eigentlichen JX-Kern zur Verfügung gestellt werden soll. Dies bezieht sich auf den Umgang mit Datentypen, Informationen über Datenstrukturen, die Klassen- und Objektdaten, sowieso das Ausführen des eigentlichen Codes. Zuerst wird beschrieben wie dies im Moment in Zusammenhang mit dem Maschinencode-Übersetzer geschieht. Da bislang nur wenige Informationen über weite Teile dieses Themenbereichs vorliegen und gerade bei einigen dieser Strukturen im Laufe der Arbeit einige Optimierungen vorgenommen werden sollten, wird hierauf sehr ausführlich eingegangen. Anschließend wird erläutert wo die Unterschiede beim Vorgehen des Bytecode-zu-C-Compilers im Gegensatz zum Maschinencode-Compiler liegen. Einige Beispiele verdeutlichen dabei, wie die konkrete Implementierung aussieht.

Kapitel 4 beschäftigt sich dann mit der eigentlichen Umsetzung des Bytecodes zu C-Code. Zunächst werden interessante Aspekte des vorhandenen JX-Systems erläutert und dann spezielle Probleme beim Umsetzen näher beleuchtet. Darunter fallen z. B. der unterschiedliche Umgang mit Datentypen oder spezielle Sonderheiten im Programmfluss.

Zum Zusammenfassen der Ergebnisse und einem Ausblick auf weitere Möglichkeiten des Bytecode-zu-C-Übersetzer dient schließlich Kapitel 5. Es werden nochmals die Vor- und Nachteile des Implementierens von JX als statischem C-Kern aufgezeigt und mit praktischen Ergebnissen verdeutlicht. Sowohl nahe liegende, als auch weiterführende Optimierungen und Möglichkeiten des Konzepts werden besprochen.

2 Modifizieren des Build-Systems

Das zuerst erstellte einfache Makefile-System, das zum Übersetzen des JX-Kerns und der Komponenten diente, wurde vor einiger Zeit durch ein größtenteils in Java implementiertes System ersetzt. Auch wenn dieses System in relativ kurzer Zeit und mit eher geringen Anforderungen entwickelt wurde, stellt es doch ein für seinen Einsatzzweck hervorragend geeignetes System dar.

Leider gilt dies nicht mehr für die Anforderungen, die im Rahmen dieser Arbeit an das System gestellt werden mussten. Daher bestand ein nicht geringer Teil der Arbeit darin ein flexibleres System zu entwickeln, das diesen Anforderungen genügt. Da eine komplette Umarbeitung jedoch einen erheblichen Arbeitsaufwand bedeutet hätte und das bisherige System für das Übersetzen in Maschinencode vollkommen ausreichend und fehlerfrei funktioniert, habe ich nur den Teil implementiert, der für meine Arbeit direkt von Nutzen war. Eine Einarbeitung der restlichen Funktionalität sollte jedoch problemlos möglich sein, da das neue System auf größtmögliche Kompatibilität zum normalen System[1] hin entworfen wurde.

Um dies besser zu dokumentieren und die unterschiedlichen Anforderungen herauszustellen, wird im Folgenden zuerst in Kapitel 2.1 das normale Build-System kurz vorgestellt. Dann wird in Kapitel 2.2 auf der nächsten Seite auf die Anforderungen eingegangen, die das bisherige System nicht oder nur unzureichend erfüllen konnte. Kapitel 2.3 auf Seite 11 gibt dann schließlich einen Überblick über das neu implementierte System und dessen Verknüpfungen mit dem normalen System.

2.1 Konzept des Build-Systems von JX

Die Komponenten stehen im Mittelpunkt des normalen Build-Systems. Abbildung 2.1 auf der nächsten Seite gibt einen groben Überblick über das Vorgehen des Systems. Auf Grund der vorhanden Konfigurations- und Beschreibungsdateien (*META*-Dateien), ermittelt das System die für jede Komponente benötigten Java-Quelldateien (Dateiendung `.java`). Diese werden im ersten Schritt mit Hilfe eines normalen Java-Compilers (im konkreten Fall mit dem Compiler von Sun Microsystems), in Klassendateien (Dateiendung `.class`) übersetzt, die Java-Bytecode enthalten.[2]

Diese Dateien werden nun wiederum ermittelt[3] und in Form einer ZIP-Datei (Dateiendung natürlich `.zip`) zusammengefasst. Da das ganze Verfahren komponentenweise abläuft, enthält also eine ZIP-Datei genau die Klassendateien einer Komponente.

[1] Im Folgenden wird der Begriff *normales System* zur Bezeichnung des bisherigen Systems benutzt, da es durch das neue System im Moment nicht ersetzt wird und daher die Begriffe *bisherig*, oder sogar *alt* falsche Tatsachen implizieren. Das von mir erstellte System wird mit Begriffen wie *neu* oder *modifiziert* bezeichnet.

[2] Eine Ausnahme sei hier noch erwähnt: Die für *RPC* benötigten Klassendateien wurden bereits schon vom Makefile in Klassendateien übersetzt, damit die RPC-Interfaces vor dem Übersetzen bereits „aktiviert“ werden können.

[3] Es können mehr Klassendateien, als Quelldateien vorliegen, da jede Klasse in eine eigene Klassendatei übertragen wird, jedoch eine Quelldatei mehrere Klassen enthalten kann. Auch *innere Klassen* werden in eigenen Klassendateien gespeichert.

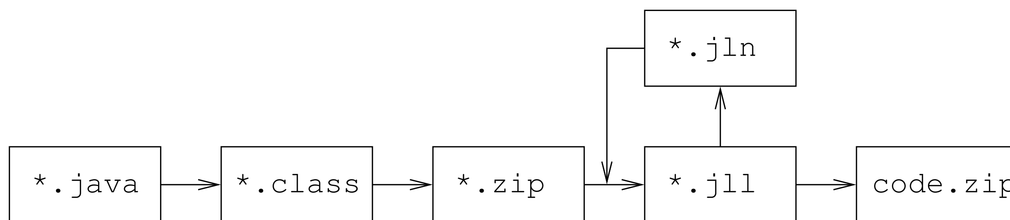

Abbildung 2.1: Schema das Build-Prozesses

Der weitere Prozess läuft nun auf Basis dieser ZIP-Dateien ab. Jeweils eine ZIP-Datei wird wieder entpackt, in Maschinencode übersetzt und dieser dann in der eigentlichen Komponentendatei (Dateiendung `.jll`) gespeichert. Hierbei entsteht wieder genau eine Komponentendatei aus der ZIP-Datei. Die Dateien haben bis auf die Endungen den gleichen Namen.

Wie dem Schema in Abbildung 2.1 zu entnehmen ist, werden beim Übersetzen noch Dateien mit der Endung `.jln` erstellt. Diese enthalten die Symbolinformationen der Komponente, die auch in der JLL-Datei zu finden sind. Die Symbolinformationen werden selbstverständlich zum Übersetzen der anderen Komponenten benötigt. Natürlich wurden die Komponenten vom Build-System vorher so sortiert, dass alle von einer Komponente benötigten Komponenten vor dieser übersetzt wurden, damit die JLN-Dateien auch zur Verfügung stehen.

Noch eine Anmerkung: In den JX-Sourcen befinden sich die Komponenten im Unterverzeichnis `libs`. Dort hat jede Komponente wiederum ihr eigenes Unterverzeichnis. Eine ähnlich Struktur hat das Verzeichnis `edomains`, das für Software „außerhalb" des JX-Systems selbst gedacht ist. Also Software, die für JX gedacht ist, aber nicht innerhalb des JX-Systems läuft oder laufen soll. Hier liegt das normale Build-System. Das neue System wurde jedoch direkt in Form von Komponenten unter `libs` implementiert. Ob die Aufteilung in `libs` und `edomains` nun sinnvoll ist oder nicht, sei dahingestellt.

2.2 Vom bisherigen System nicht erfüllte Anforderungen

Das beschriebene System funktioniert hervorragend und hilft dabei JX als modulares System zu warten und zu erweitern. Im Rahmen dieser Diplomarbeit sollte jedoch gerade die Modularität von JX eine Funktionalität darstellen, die aus Gründen der Optimierung entfallen kann. Allerdings sind weder die Weiterverwendung des bisherigen Komponenten-Systems, noch die Implementation eines optimierten Modul-Systems auf Basis des C-Übersetzers ausgeschlossen. Sie werden jedoch im Rahmen dieser Arbeit nur kurz in Kapitel 5.4 auf Seite 47 angesprochen.

2.2.1 Globale Informationen

Auch wenn das Übersetzen der JX-Komponenten meist in einer zusammenhängenden Aktion erfolgt, so werden jedoch letztendlich die einzelnen Komponenten für sich alleine übersetzt. Mit Hilfe der JLN-Datei werden nur die wirklich benötigten Informationen von anderen Komponenten zur Verfügung gestellt. Eine globale Sicht auf das System ist zu keinem Zeitpunkt wirklich vorhanden.

Diese benötigt man jedoch für die Ermittlung globaler Informationen, die beim Übersetzen der erstellten C-Dateien hilfreich oder sogar nötig sind. So bietet es sich an, eine *Definitionsdatei* für den C-Compiler zu erstellen, in der alle globalen Informationen vorhanden sind. Diese

wird von allen erstellten C-Dateien mit Hilfe der `#include`-Anweisung des Preprozessor eingebunden. Im Folgenden soll diese Datei mit dem griffigeren englischen Begriff *Header*-Datei bezeichnet werden. C sieht für diese Dateien die Dateiendung `.h` vor.

Zwar wäre es auch möglich, ein System zu entwickeln, das pro Komponente eine Header-Datei erstellt, die exakt die benötigten Informationen enthält und somit den JLN-Dateien entspräche. Jedoch trifft man hier auf das Problem, dass C selbst keine getrennten Namensräume unterstützt. Es können also Probleme mit der Eindeutigkeit insbesondere von Funktionsnamen auftreten.[4]

Die zentrale Header-Datei soll außerdem in gewisser Weise die Symbolinformationen repräsentieren. Die eigentlichen C-Quelldateien enthalten lesbare Symbolnamen für Datenfelder, Methodenaufrufe und Klassennamen, die erst durch das Anwenden des Preprozessors durch konkrete, schnellere Strukturen ersetzt werden. Dies wird durch das Erstellen von Makros mit Hilfe der `#define`-Anweisung innerhalb der zentralen Header-Datei ermöglicht, die die Symbolnamen auf die Strukturen abbildet. Diese Preprozessor-Makros werden im Folgenden kurz als *Symbolkonstanten* bezeichnet, um sich leichter auf sie beziehen zu können. Dieses Verfahren wird später noch ausführlicher erläutert.

Wichtig ist nur, dass auch hierfür eine globale Sicht über alle Informationen notwendig, bzw. hilfreich ist.

2.2.2 Optimierungen durch statisches Linken

Die Umsetzung des eigentlich relativ dynamischen Java-Bytecodes in statisch gelinkten C-Code hält selbstverständlich noch weitere Optimierungsmöglichkeiten bereit.

So sind die meisten Methodenaufrufe in Java *virtuelle Methodenaufrufe*. Das heißt, dass die Methode der zur Laufzeit aktuellen Klasse aufgerufen wird und nicht die Methode der zur Übersetzungszeit bekannten Klasse. Dies ist ein wichtiges Merkmal der Objektorientierung, führt jedoch natürlich zu einem höheren Aufwand zur Laufzeit. Es ist also von Vorteil, schon zur Zeit des Übersetzens möglichst viele Methodenaufrufe als *statisch* zu erkennen und damit den Aufruf direkt einer konkreten Methode zuzuordnen.

Wie bereits in [1, Kapitel 3.4.3.4] ausgeführt, können nicht nur die virtuellen Aufrufe von als `final` oder `private` gekennzeichneter Methoden durch statische Aufrufe ersetzt werden, die wesentlich effizienter sind. Bei einer globalen Sicht besteht zusätzlich die Möglichkeit Methoden zu bestimmen, die nicht mehr von Unterklassen überschrieben werden (*system final*) und somit durch statische Aufrufe ersetzt werden können.

Hierfür ist aber ein vollständiger Blick auf die Beziehungen der Klassen untereinander unverzichtbar. Es müssen alle Klassen bestimmt werden, die direkt oder indirekt von einer anderen Klasse abgeleitet wurden.

2.3 Aufbau des modifizierten Build-Systems

Um eine globale Sicht über alle Klassen zu ermöglichen, die statisch zusammen mit dem Kern gelinkt werden sollen, wurde ein neues Build-System implementiert, das konzeptionell anders aufgebaut ist.

[4] Erst in Kapitel 3.7 auf Seite 27 findet eine nähere Erläuterung statt, warum und wofür C-Funktionen beim Übersetzen nach C eingesetzt werden.

2.3.1 Die wichtigsten Klassen

Alle Konfigurations- und Java-Dateien werden eingelesen und ergeben gemeinsam ein komplettes Abbild aller Komponenten. Die wichtigsten Klassen, die für ein Abbild des realen Systems im Build-System verantwortlich sind, sind in Abbildung 2.2 vereinfacht dargestellt.[5]

Die Klasse `Builder` besteht dabei im Wesentlichen aus der Klassenmethode `main()`, die (wie gewohnt) die beim Start des Builders ausgeführte Methode ist. Diese Methode bedient sich nun der Klassen `ComponentsList` und `Component`, die eine Liste aller Komponenten, bzw. eine einzelne Komponente repräsentieren. Die Klassendaten selbst werden, ähnlich wie beim normalen Build-System, in einem `ClassStore`-Objekt gespeichert. Dieses enthält hierfür dann, wie vom normalen System her gewohnt Objekte, die die einzelnen Klassen repräsentieren. Diese wiederum enthalten Objekte, die Methoden- und Felder-Informationen, sowie den gesamten restlichen Bytecode darstellen auf Basis der Klassen aus der `classfile`-Komponente.

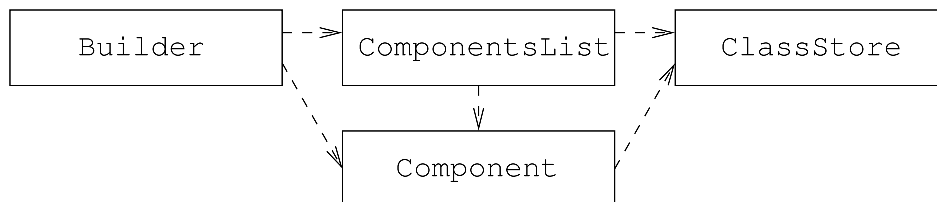

Abbildung 2.2: Vereinfachtes Klassendiagramm des modifizierten Build-Systems

Die einzelnen Klassen sind dabei vielschichtig miteinander verbunden, damit die Suche nach speziellen Informationen erleichtert wird. So besitzt jede Klasse Referenzen auf die jeweilige Basis-Klasse und auch auf die Klassen, die von ihr abgeleitet sind (falls vorhanden). Diese Referenzen reichen dabei auch über Komponentengrenzen hinaus. Man kann jedoch wiederum feststellen, welcher Komponente eine Klasse angehört. Somit ist sowohl eine Komponenten-, als auch eine Klassen-bezogene Sicht auf das System möglich.

2.3.2 Die verwendeten Komponenten

Beim Umbau des Systems sollten so wenig wie möglich Anpassungen am Quellcode vorgenommen werden. Auch bereits vorhandene Komponenten und Klassen sollten möglichst direkt verwendbar bleiben.

Die Abbildung 2.3 auf der nächsten Seite gibt einen guten Überblick über die wichtigsten JX-Komponenten, die das modifizierte Build-System enthalten (weiß), und deren Verbindungen zu bereits vorhanden JX-Komponenten (grau).[6] Es ist leicht zu erkennen, dass die Komponenten `compiler` und `classfile` in ihrer vorhandenen Form direkt benutzt werden. Die anderen Komponenten (`builder`, `components`, `compiler_c` und `compiler_env2`) sind die neu hinzugekommenen Komponenten.

Hierbei nimmt `compiler_env2` eine Sonderstellung ein. Die Komponente enthält sowohl nahezu vollständig neue Klassen, als auch direkte Kopien von Klassen aus der bereits vorhandenen Komponente `compiler_env`. Da stellenweise die APIs von Klassen in `compiler_env` stark modifiziert werden mussten, mussten sie in eine neue Komponente kopiert werden und

[5] Das Diagramm erhebt in keinster Weise Anspruch auf Vollständigkeit. Die Darstellungsform entspricht im Wesentlichen der bei *UML*-Diagrammen üblichen Form.

[6] Sowohl Beziehungen, als auch verwendete Komponenten sind nicht vollständig dargestellt.

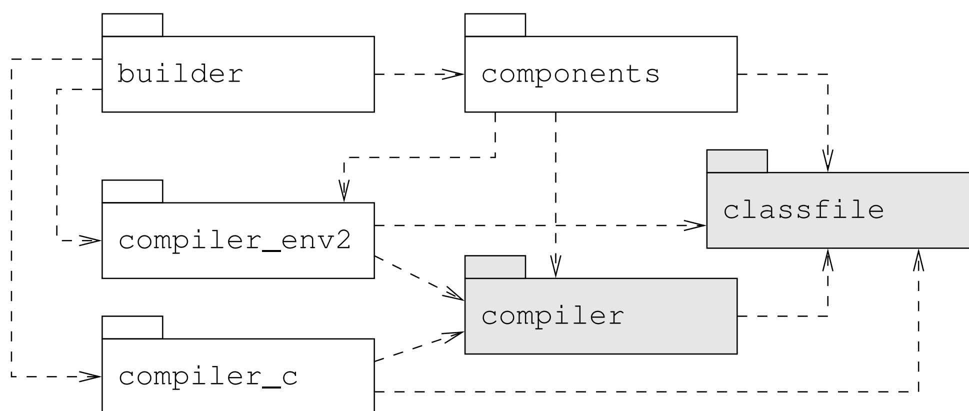

Abbildung 2.3: Überblick über die Komponenten des modifizierte Build-Systems

diese konnte dann nicht mehr zusammen mit der bisherigen Komponente `compiler_env` übersetzt und genutzt werden. Also stellt `compiler_env2` im Wesentlichen eine modifizierte Kopie von `compiler_env` dar.

2.3.3 Weitere Anmerkungen

Beim Umschreiben des Build-Systems habe ich einigen Code entfernt, der wohl nicht mehr benötigt wird. Es könnte natürlich trotzdem Code entfernt worden sein, der später einmal vermisst werden könnte. Aber im normalen Build-System sollte er ja weiterhin zu finden sein.

Allerdings ist anzunehmen, dass das normale Build-System weiter erweitert wird und das modifizierte Build-System nicht angepasst wird. Dies kann dazu führen, dass das modifizierte System neue Features nicht nutzen kann, oder sogar gar nicht mehr funktioniert!

2.4 Fehlende Funktionalitäten im neuen Build-System

Das implementierte System weist einige wesentliche Schwächen auf, die entweder vom normalen System übernommen wurden, oder darauf zurückzuführen sind, dass das System (noch) nicht komplett implementiert wurde. Dies Einschränkungen sollen im Folgenden näher erläutert werden.

2.4.1 Einschränkungen des normalen System

Das normale Build-System, wie auch die hier beschriebene Umarbeitung haben Probleme mit zirkulären Abhängigkeiten. Eine Komponente wird erst bearbeitet, wenn alle Komponenten bearbeitet wurden, von denen sie abhängt. Da dies rekursiv auf Basis der Informationen in den META-Dateien geschieht, kann es z. B. vorkommen, dass durch einen Anwenderfehler eine benötigte Komponente wiederum die Ausgangskomponente benötigt, wodurch die Bestimmung der Abhängigkeit immer im Kreis verläuft.

Die einzige Möglichkeit dies zu verhindern besteht darin, den Code dahingehend zu erweitern, dass solche Abhängigkeiten erkannt werden, indem z. B. eine Liste aller bereits betrachteten Komponenten erstellt wird und an Hand dieser Liste überprüft wird, ob die jeweils aktuell untersuchte Komponente schon einmal untersucht wurde.

Es sei nochmals betont, dass solch eine zirkuläre Abhängigkeit keinen Sinn macht und meistens wohl auf einen Benutzerfehler zurückzuführen ist.

2.4.2 Nicht implementierte Merkmale

Es wurde bereits erwähnt, dass das neue Build-System nur weit genug entwickelt wurde, um das Übersetzen in C-Code zu ermöglichen.

Soll das System auch dazu genutzt werden den Maschinencode-Übersetzer zu benutzen, müssten noch einige Erweiterungen und wohl auch Änderungen vorgenommen werden:

- Der meiste Code, der für die Erzeugung von Maschinencode zuständig wäre wurde einfach an einigen Stellen in den neuen Code übernommen, jedoch nicht an die neuen APIs angepasst und deshalb auskommentiert. Um ihn benutzen zu können, müssten noch eine Anpassung und natürlich ausführliche Tests erfolgen. Es handelt sich hierbei um Code, wie z. B. die Methoden, die die Methodentabellen serialisieren, um sie in die JLL- oder JLN-Dateien einfügen zu können.

- Die Klasse `Component` entnimmt im Moment ihre Informationen der META-Datei der Komponente und den Klassendateien. Um das bisherige System abzubilden müsste es auch möglich sein, dass `Component` seine Informationen aus einer JLN-Datei an Stelle der Klassendateien gewinnen kann. Somit wäre, wie bisher, das Übersetzen einer Komponente möglich, ohne dass alle Klassendateien der anderen Komponenten vorliegen.

  Um dies zu erreichen mag es sinnvoll sein die Klasse `Component` aufzusplitten. Während `Component` die grundlegenden Methoden bereithält, die für beide Typen gleich sind, definiert sie auch abstrakte Methoden, die von Unterklassen überschrieben werden müssen. So könnte man dann zwei unterschiedliche Klassen ableiten. Eine für die Initialisierung aus META- und Klassendateien, die andere für die Initialisierung aus META- und JLN-Dateien.

- Der Aufbau der `Builder`-Klasse ist leider etwas stark angepasst worden und ein Implementieren beider Übersetzungsverfahren wird wohl viele Änderungen mit sich bringen. Dies liegt unter anderem daran, dass die ZIP-Datei als „Archiv von Java-Quelldateien" weg fällt und somit der Prozess eigentlich anders verläuft.

  Während der Arbeit des Builders wird oft (wie bei einem Makefile üblich) überprüft, ob die Dateien überhaupt neueren Datums sind, als die zu erstellenden Dateien. Gerade hier wird das Fehlen einer ZIP-Datei etwas problematisch. Nicht zuletzt deshalb erstellt der Builder im Moment nach dem Übersetzen auch die ZIP-Dateien. Dies soll auch demonstrieren, wie der Weg über die ZIP- und JLN-Dateien wieder reintegriert werden könnte.

Die aufgeführten Änderungen und Erweiterungen des neuen Build-Systems sollten alle keine größeren Probleme bereiten, sind jedoch mit einem nicht unerheblichen Aufwand verbunden.

3 Linken gegen den Kern

Dieses Kapitel beschäftigt sich mit der Frage, wie die erstellten C-Quelldateien mit dem Kernel zusammen übersetzt und gelinkt werden können. Hierfür dient das Vorgehen des Kerns beim Laden der Maschinencode-Komponenten als Ausgangspunkt. Der Umgang mit dem C-Code soll durch möglichst wenig Änderungen am Kern möglich sein, um die Einschränkungen in der Funktionalität zu minimieren.

Für die Java-spezifischen Konstrukte wie primitive Datentypen, Objekte, Arrays usw. existieren naturgemäß bereits konkrete Datentypen und Strukturen für deren Verarbeitung im Kern. Der Kern muss mit diesen Datentypen arbeiten und sie den Java-Klassen zur Verfügung stellen. Diese Verzahnung zwischen C-Kern und Maschinencode-Komponente ist von großer Bedeutung für das Vorgehen beim Laden von Komponenten, die als C-Quellcode vorliegen, und wird deshalb in Kapitel 3.1 näher beleuchtet.

Weiterhin ist es nötig zu betrachten, wie die Datenstrukturen für die höheren Datentypen und anderen Strukturen aus den Komponenten ausgelesen werden (Kapitel 3.2 auf der nächsten Seite) und in welcher Form diese dann im Kern vorliegen (Kapitel 3.3 auf Seite 17). Auch muss betrachtet werden, wie diese Strukturen vom Maschinencode aus abgefragt und manipuliert werden können (Kapitel 3.4 auf Seite 21).

Nach diesen Betrachtungen des Ist-Zustandes, stellt sich nun zuerst in Kapitel 3.5 auf Seite 21 die Frage, wie beim Übersetzen nach C vorgegangen wird, um diese relevanten Datenstruktur dem Kern am besten statisch, also bereits zum Zeitpunkt des Linkens, bekannt zu geben.

In den Kapiteln 3.6 auf Seite 26 und 3.7 auf Seite 27 wird dann exemplarisch gezeigt, wie der C-Quellcode diese Datenstrukturen nutzt um auf die Member von Klassen und Objekten zuzugreifen.

3.1 Primitive und höhere Datentypen

Der Kern muss dafür sorgen, dass die Datentypen von Java adäquat umgesetzt werden. Während nämlich Java konkrete Vorgaben für primitive Datentypen gibt und damit eine hohe Plattformunabhängigkeit erreicht, sind Datentypen in C immer noch zu großen Teilen von der Hardware selbst und dem jeweiligen C-Compiler abhängig. Solange der JX-Kern aber Datentypen deklariert, die die Java-Datentypen repräsentieren, muss sich der Kern an einer zentralen Stelle darum kümmern, dass eben diese Datentypen korrekt verwendbar sind. Dies geschieht im Moment in der Datei `jcore/types.h`, die u.a. die Datentypen `jint`, `jlong`, `jfloat` und `jdouble` deklariert.

Höhere Datenstrukturen werden von Java, vor allem auf Bytecode-Ebene durch *Referenzen* repräsentiert. Referenzen werden von JX, wie es sich anbietet, durch den C-Datentyp `jint` repräsentiert, der dem Java-Datentyp `int` entspricht und in C als Pointer benutzt wird. Dieser Pointer zeigt dann z. B. auf eine C-Struktur (`struct`) vom Typ `ObjectDesc`. Diese C-Strukturen, die meist auf `Desc` enden, wie z. B. `ClassDesc`, `ArrayDesc` oder eben `ObjectDesc` dienen im Kern dazu die höheren Datenstrukturen von Java bereitzustellen.

3.2 Laden der Komponenten durch JX

In Kapitel 2.1 auf Seite 9 wurde erläutert, wie der Maschinencode der JX-Komponenten in JLL-Dateien gespeichert wird. Diese Komponentendateien müssen nun vom Kern geladen werden. Dies erledigen vor allem die Funktionen in der Datei `load.c`.

3.2.1 Statische und dynamische Datenstrukturen

Bevor näher auf die Vorgänge beim Laden der Komponenten eingegangen werden kann, muss noch eine wichtige Erklärung zur Architektur von JX erfolgen.

JX bietet als herausragende Funktionalität an, so genannte *Domains* zu benutzen. Das Konzept der Domains ähnelt dabei dem von Prozessen in anderen Betriebssystemen. Eine Domain stellt gewissermaßen einen Container für eine Anwendung oder einen Dienst dar. Sie kann folgende Objekte enthalten: Komponenten (also Java-Klassen), Heap, Speicherbereich für Code und Stack, Threads, Dienste, Portale und Interrupt-Handler. Die Portale dienen dabei als einzige Kommunikationsmöglichkeit zwischen den Domains. Die Domains ermöglichen so eine sehr flexible Gestaltung des JX-Systems.

Die Domain *zero* spielt eine Sonderrolle in JX. Sie ist gewissermaßen die *System-Domain*. Sie repräsentiert den JX-Kern als Domain.

Für das Übersetzen nach C ist jedoch noch eine andere Betrachtungsweise von Bedeutung: Im Speicherbereich der Domain *zero* liegen auch viele Strukturen, die gemeinsam von alle Komponenten genutzt werden. Diese werden im Folgenden auch als *shared* bezeichnet. Ein typisches Beispiel ist der eigentliche Maschinencode der Java-Klassen, der in der Domain *zero* liegt und von allen Domains gemeinsam genutzt wird.[1] Aber auch die meisten anderen Strukturen bestehen aus einem Teil, der für alle Domains gemeinsam genutzt wird (also in *zero* liegt), und einem anderen Teil, der für jede Domain lokal vorliegt.

Dies führt beim statischen Linken mit dem C-Code zu der Überlegung, die gemeinsam genutzten Strukturen, die in Domain *zero* liegen, schon beim Übersetzen mit Hilfe von statischen C-Initialisierungen zu erzeugen. Die Strukturen, die für jede Domain getrennt vorliegen, können indes *nicht* statisch erzeugt werden, da die Domains selbst und damit auch diese Strukturen von JX dynamisch erzeugt werden (müssen).

Im Folgenden wird der Einteilung in statische und dynamische Strukturen immer besondere Aufmerksamkeit geschenkt. Auch in Abbildungen werden die statischen und dynamischen Teile grafisch voneinander abgehoben.

3.2.2 Ladevorgang

Abbildung 3.1 auf der nächsten Seite gibt das prinzipielle Vorgehen beim Laden einer Komponente wieder, indem die drei hauptsächlich beteiligten C-Funktionen in einen Zusammenhang gebracht werden.

Wie sofort ersichtlich ist, übernehmen die Funktionen `loadSharedLibrary()` und `linksharedlib()` das Laden der statischen Strukturen. Die Funktion `loadSharedLibrary()` lädt die Komponente, indem sie Speicher bereitstellt und die Daten aus der Komponenten-Datei hinein kopiert. Die Funktion `linksharedlib()` löst dann die Symbole im geladenen

[1] Es besteht auch die Möglichkeit Maschinencode in eine normale Domain zu laden und zu benutzen. Der Code dafür ist vorhanden, wird jedoch nicht benutzt und es wird auch nicht empfohlen ihn zu benutzten. Deshalb wird im Rahmen dieser Arbeit auch nicht darauf eingegangen.

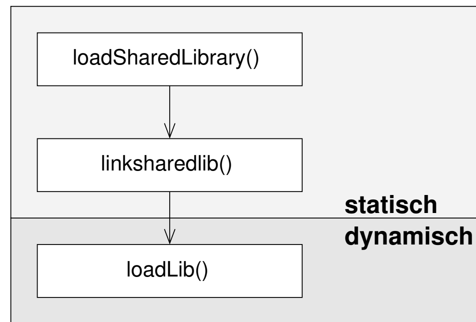

Abbildung 3.1: Schematischer Ablauf der Funktion `load()` in `load.c`

Code auf und erstellt die Methodentabellen.

Die Methode `loadLib()` nimmt dann schließlich noch die Initialisierung der domainspezifischen Strukturen vor. Sie erzeugt also die dynamischen Strukturen. Die Funktion fällt deutlich kleiner aus, als die beiden anderen.

Vereint werden die drei Funktionen in der Funktion `load()`. Sie lädt eine Komponente in eine Domain. Nur, wenn die Komponente nicht schon einmal geladen wurde, werden dabei `loadSharedLibrary()` und `linksharedlib()` ausgeführt.

3.3 Aufbau der relevanten Datenstrukturen im Kern

Die meisten der beim Ladevorgang (siehe Kapitel 3.2.2 auf der vorherigen Seite) verwendeten Strukturen sind in der Datei `code.h` als C-Strukturen (`structs`) deklariert. Durch das Laden der Komponenten werden „verzeigerte“ Instanzen dieser Strukturen erzeugt.

Dieses Kapitel soll einen Überblick über die wichtigsten Strukturen bieten, ohne jedoch zu sehr ins Detail zu gehen.

3.3.1 Komponenten, Klassen und Domains

Die Abbildung 3.2 auf der nächsten Seite gibt einen Überblick über die wichtigsten Strukturen, die zur Verwaltung von *Domains*, *Komponenten* und *Klassen* dienen.

Die Variable `sharedLibs` dient als zentrale Stelle der gemeinsam genutzten Strukturen. Von ihr aus sind alle Strukturen vom Typ `SharedLibDesc` in Form einer verketteten Liste erreichbar. Diese repräsentieren die geladenen Komponenten. In der Darstellung sind nicht alle Felder der Struktur eingetragen, sondern nur die wichtigsten. Über das Feld `allClasses` sind alle Klassen dieser Komponente erreichbar. Genauer gesagt, deren jeweilige `ClassDesc`-Strukturen. Diese Struktur hält die Daten einer Klasse bereit und stellt die größte der benutzten Strukturen dar. (In der Abbildung ist wieder nur ein Ausschnitt zu sehen.) Sie enthält wiederum einen Zeiger auf die `SharedLibDesc`-Struktur der Komponente, zu der sie gehört.

Untereinander sind die `ClassDesc`-Strukturen auch verbunden, um eine verkettete Liste zu bilden[2] und zusätzlich um das Auffinden der Basisklasse einer beliebigen Klasse zu ermöglichen.

---

[2] Diese Möglichkeit wird für die Klassen aus den geladenen Komponenten nicht genutzt. Diese werden am Stück in ein Array geschrieben und das `next`-Feld auf dem Wert `null` belassen. Anders sieht es mit den Klassen des Kerns aus. Siehe hierzu die späteren Kapitel 3.5.2 bis 3.5.4 auf Seiten 24–25.

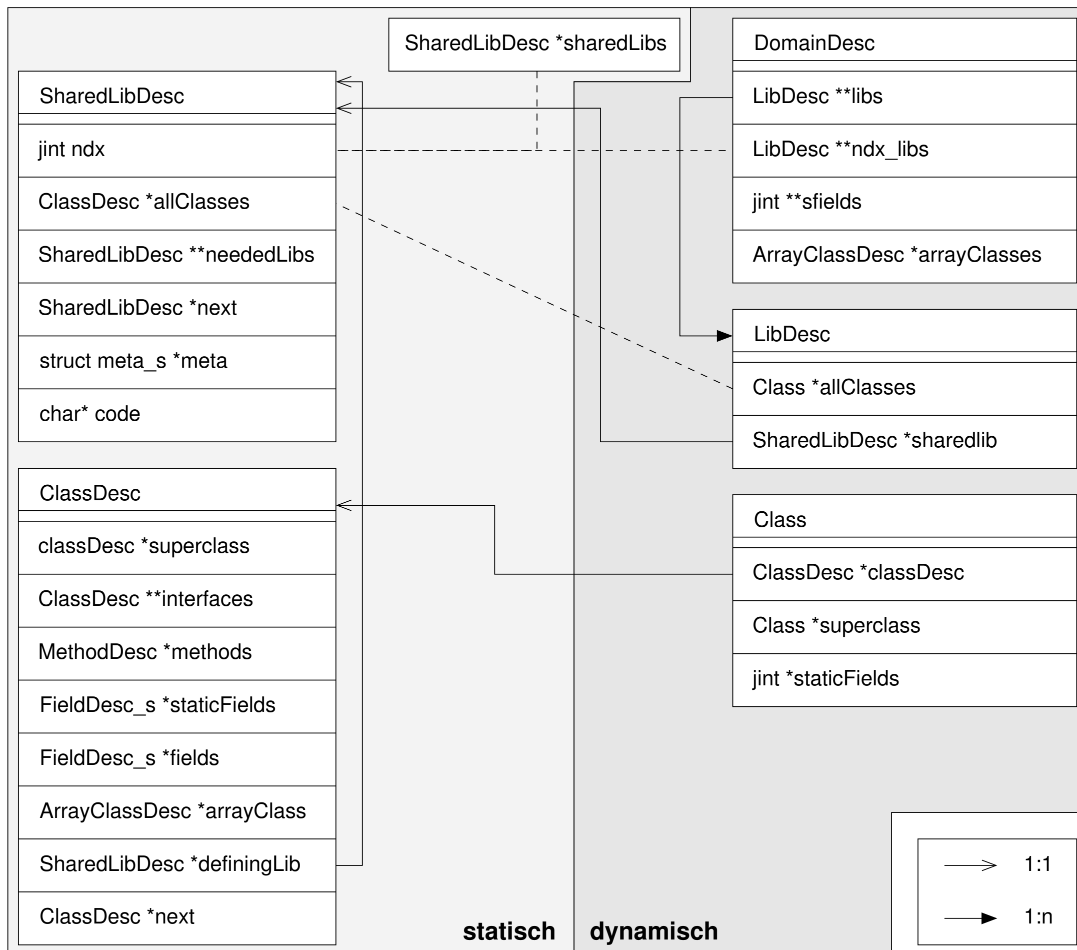

Abbildung 3.2: Die Datenstrukturen für Komponenten, Klassen und Domains

In der Abbildung rechts sind die Strukturen zu sehen, die für eine Domain lokal sind. Die Domain selbst wird durch die `DomainDesc`-Struktur repräsentiert. `LibDesc` und `Class` stellen domainspezifische Strukturen für Komponenten und Klassen dar. Sie enthalten einen Zeiger auf die jeweilige statische Struktur in Domain *zero*. Da sie nur relativ wenige Felder enthalten, ist der Zugriff auf die Informationen in den Strukturen, von denen sie abhängen, sehr wichtig.

Zu sehen ist sehr schön, dass z. B. die statischen Klassenfelder nicht in `ClassDesc`, sondern in `Class` liegen. So ist es eben möglich, dass auch von Klassen domainspezifische Versionen existieren. Wie bereits erwähnt gilt hier die Einschränkung, dass der eigentliche Maschinencode gemeinsam genutzt wird. Außerdem liegt der Speicher auf den `Class.staticFields` verweist in einem für die Domain gemeinsam benutzten Speicherbereich, der über `DomainDesc.sfields` zu finden ist. Als Fazit bleibt zu sagen, dass auch statische Klassendaten in JX in gewisser Weise „dynamisch“ sind.

3.3.2 Der Index aller Komponenten

In den Datenstrukturen finden sich etliche Konstruktionen, die das Auffinden der gesuchten Informationen beschleunigen sollen. Eine davon sei hier exemplarisch näher erläutert.

Wie bereits in Kapitel 3.3.1 auf Seite 17 erläutert, stellt `sharedLibs` den Beginn einer verketteten Liste aller `SharedLibDesc`-Strukturen dar. Zusätzlich besitzt jede dieser Komponenten jedoch auch einen eindeutigen Index. In der Abbildung 3.2 auf der vorherigen Seite ist angedeutet, dass dieser Index eine besondere Beziehung zwischen den `SharedLibDesc`-Strukturen in `sharedLibs` und dem Feld `ndx_libs` in den `DomainDesc`-Strukturen ermöglicht.

In Abbildung 3.3 wird dieser Zusammenhang verdeutlicht. Während die `SharedLibDesc`-Strukturen als verkettete Liste vorliegen, stellt `ndx_libs` einen Vektor mit Zeigern auf die für die Domain entscheidenden `libDesc`-Strukturen dar. Es ist also möglich auf Basis einer `DomainDesc`-Struktur mit Hilfe des Indexes einer Komponente direkt über einen Zeiger auf deren beschreibende Datenstruktur für die Domain zuzugreifen.

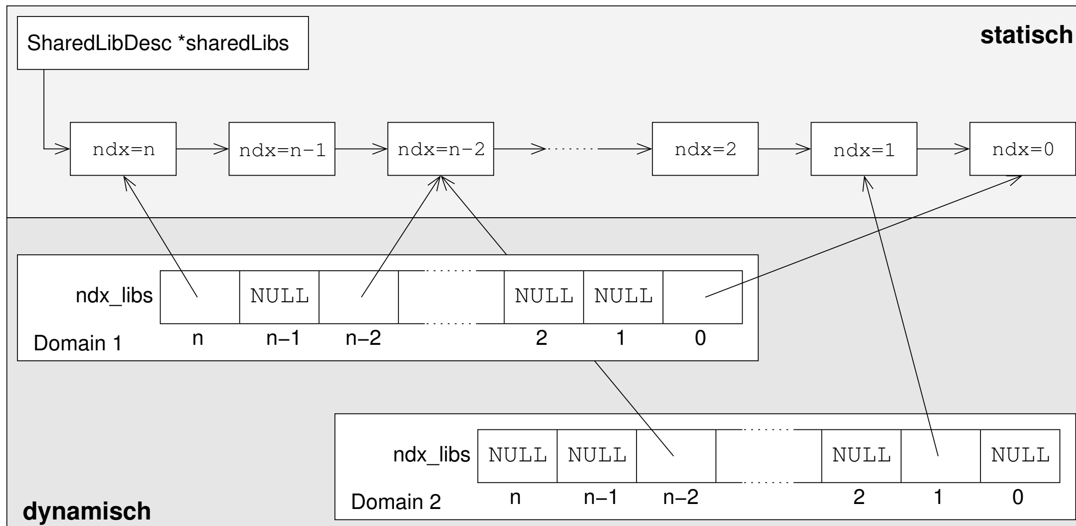

Abbildung 3.3: Statische und domainspezifisch Datenstrukturen der Komponenten

Es muss vielleicht noch erwähnt werden, dass die Zeiger in der Abbildung von den Vektoren `ndx_libs` zu den `SharedLibDesc`-Strukturen natürlich keine Zeiger im C-Sinne darstellen, sondern nur die Verbindung verdeutlichen sollen.

Mit dem Eintrag `NULL` wird in `ndx_libs` angezeigt, dass die Komponente nicht in dieser Domain geladen wurde. Außerdem wird deutlich, dass `sharedLibs` in Bezug zum Index rückwärts aufgebaut wird.

3.3.3 Methodentabellen

Abbildung 3.4 auf der nächsten Seite zeigt die Datenstrukturen, die für den Aufruf *virtueller Methoden* (siehe Kapitel 2.2.2 auf Seite 11) benötigt werden. Die dafür benötigten Methodentabellen werden dementsprechend meist als *virtuelle Methodentabellen* (kurz *vtables*) bezeichnet. Die Darstellung ist wieder vereinfacht.[3]

In der `ClassDesc`-Struktur sind die Felder `vtables` und `methodVtable` für die Verwaltung der Methoden verantwortlich. Ersteres enthält dabei einen Zeiger auf den Vektor, der die speziellen Zeiger vom Typ `code_t` enthält, die auf den eigentlichen Maschinencode verweisen. In `methodVtable` sind Zeiger auf die beschreibenden Strukturen der jeweiligen Methode zu finden, die über den Index der beiden Vektoren als zusammengehörig erkennbar sind.

[3] Siehe zu diesem Kapitel auch [1, Kapitel 3.2.3]

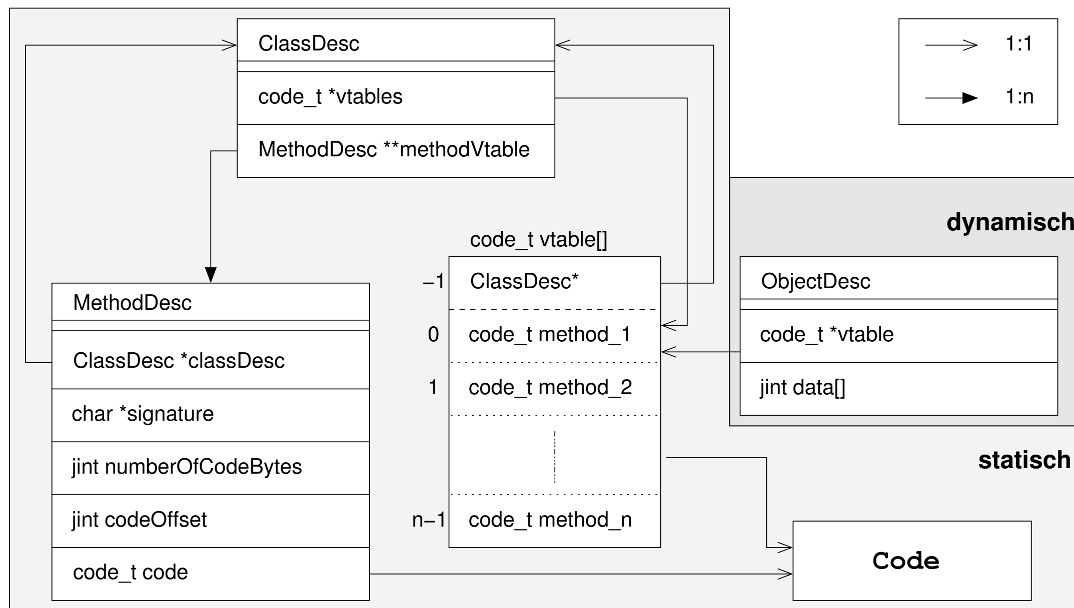

Abbildung 3.4: Klassen, Objekte, Methodentabellen und deren Beziehungen

Wie unschwer zu erkennen ist, sind all diese Datenstrukturen statisch. Dynamisch wird nur die Zuordnung einer Methodentabelle zu einem Objekt (repräsentiert als `ObjectDesc`) organisiert. Dabei wird, wie in der Abbildung ersichtlich, direkt auf den Vektor mit den `code_t`-Einträgen verzeigert. Dies ermöglicht ein schnelles Auffinden des auszuführenden Codes. Eine abgeleitete Klasse „erbt“ nämlich im Prinzip die Methodentabelle ihrer Basisklasse. Dabei werden neue Methoden an das Array angefügt, erhalten also noch verfügbare Indizes. Wird eine Methode jedoch überschrieben, so wird der `code_t`-Zeiger an der entsprechenden Stelle auf den neuen Code umgebogen. Auf diese Weise befinden sich die Methoden aller von einer Basisklasse direkt oder indirekt abgeleiteten Klassen an exakt der gleichen Position. Dies ermöglicht damit den virtuellen Methodenaufruf. An der Stelle, an der ein Objekt einer bestimmten Klasse erwartet wird, kann auch eine davon abgeleitete Klasse eingesetzt werden und dennoch direkt über den Index in der Methodentabelle die richtige Methode (ob nun vererbt oder überschrieben) ausgewählt werden.

Es sei noch erwähnt, dass an dieser Stelle das Konzept der *Interfaces* in der Programmiersprache Java besonders beachtet werden müssen. Man muss „Platz“ für alle Interface-Methoden in den Tabellen vorsehen, da Interfaces ja in gewissen Grenzen das Erben von mehreren Klassen ermöglichen. Auch wenn eine Klasse ein Interface nicht implementiert, so kann dies eine später von ihr abgeleitete Klasse dennoch tun. Dadurch können sich nun aber die Indizes der Methoden der Basisklasse mit denen der Methoden des Interfaces überschneiden. Dies macht es nötig eine „Methodentabelle mit Löchern“ zu implementieren. Es muss sichergestellt werden, dass genug „Platz“ in der Tabelle ist, auch wenn dann bei einigen Klassen nicht belegte „Löcher“ entstehen. Hier gibt es im Allgemeinen noch weitere Optimierungsmöglichkeiten, wie die Trennung von normaler virtueller Methodentabelle und virtueller Methodentabelle für Interface-Methoden. Im Moment werden solche Optimierungen unter JX jedoch nicht verwendet.

3.3.4 Strukturen für Objekte

Objekte werden, wie bereits in Kapitel 3.3.3 auf Seite 19 ersichtlich, von einer `ObjectDesc`-Struktur repräsentiert.[4] Außer dem Zeiger auf die Methodentabelle besitzt ein Objekt nur noch das `jint`-Array, in dem die Felder des Objekts gespeichert werden. Es gibt keinerlei direkten Verweis auf die Klasse von der das Objekt eine Instanz darstellt.

Allerdings befindet sich ein Zeiger auf die gesuchte `ClassDesc`-Struktur an der Index-Position `-1` in der Methodentabelle, die sonst nur `code_t`-Zeiger enthält. Auf diese Weise ist eine Typ-Bestimmung zur Laufzeit ohne Probleme möglich, während gleichzeitig die `ObjectDesc`-Struktur hinsichtlich Platz und Zugriffszeit stark optimiert ist.

Bei den Objekt-Strukturen gibt es keinerlei Optimierungsmöglichkeiten durch ein statisches System. Objekte sind prinzipbedingt ein durchweg dynamisches Konzept.

3.4 Support-Funktionen im Kern

Wie gezeigt finden sich im JX-Kern viele Datenstrukturen, die sowohl beschreibende Informationen bereithalten, als auch die höheren Datentypen von Java (Klassen, Objekte, Arrays, Strings usw.) verfügbar machen. Es stellt sich nun die Frage, wie es dem Maschinencode, der aus dem Bytecode erzeugt wird, ermöglicht wird auf diese Strukturen zuzugreifen.

In der Tat wird dies meist nicht über direkte Adressen und Maschinencode-Befehle erledigt, sondern es gibt einen Mechanismus, mit dem C-Funktionen des Kerns vom Maschinencode aus verwendet werden können. Der Kern bietet nun genau für die oben erwähnten komplexen Aufgaben spezielle Funktionen an, die im Folgenden als *Support-Funktionen* bezeichnet werden sollen. Der Maschinencode führt die benötigten C-Funktionen einfach aus, die ihm über einen speziellen Mechanismus zur Verfügung gestellt werden. Die meisten dieser Funktionen sind in der Datei `vmsupport.c` zu finden.

Es bietet sich selbstverständlich an, parallel dazu diese Funktionen auch vom erzeugten C-Code aus direkt zu benutzen. Es handelt sich immerhin schon um C-Funktionen und die Funktionalität bleibt exakt erhalten. Nur manchmal ist eine Umgehung der Support-Funktionen angeraten, wie z. B. bei der Unterstützung des Java-Datentyps `long`, die im Moment teilweise auf C-Funktionen zurückgreift.

3.5 Statische Erzeugung der Datenstrukturen

In Kapitel 2.2.1 auf Seite 10 wurde bereits erläutert, dass es sinnvoll erscheint eine zentrale Header-Datei zu generieren, die alle globalen Informationen in einer möglichst statischen Form enthält. Auf Grund der vorangegangenen Kapitel wissen wir nun, dass sich diese Informationen vor allem auf die dort beschriebenen Datenstrukturen stützen.

Auf Grund der Arbeitsweise des C-Preprozessors darf eine Header-Datei natürlich nur Deklarationen, jedoch keine Definitionen enthalten. Speicher darf für einen globalen Bezeichner nur an einer Stelle reserviert werden. Es muss also neben der Header-Datei, die Bezeichner nur über eine `extern`-Anweisung bekannt gibt, noch eine C-Datei geben, die die eigentliche Initialisierung vornimmt. Wären diese Definition in der Header-Datei, würden alle C-Quelldateien, die diese einbinden, versuchen die Initialisierung vorzunehmen.

[4] Siehe auch zu diesem Kapitel wieder [1, Kapitel 3.2.3]

Wir werden nun an Hand von vereinfachten Beispielen betrachten, wie das statische Erzeugen für die einzelnen Datenstrukturen umgesetzt wird. Es werden wieder nur die wichtigsten Strukturen exemplarisch betrachtet.

3.5.1 Beschreibende Strukturen

Eine wichtige Stellung nehmen die beschreibenden Datenstrukturen ein, die die Informationen von Komponenten und Klassen enthalten. Es ist offensichtlich, dass die in Kapitel 3.3.1 auf Seite 17 beschriebenen statischen Datenstrukturen wie `SharedLibDesc` und `ClassDesc` mit einigen Tricks in C komplett statisch initialisiert werden können.

Mit folgendem C-Programm soll nun das Vorgehen an Hand eines vereinfachten Beispiels erläutert werden.

Prinzip der statischen Definition der Struktur sharedLibs

```
typedef signed long jint;
#define NULL ((void*)0)

typedef struct ClassDesc_s {
  char *name;
  struct ClassDesc_s *superclass;
  struct SharedLibDesc_s *definingLib;
} ClassDesc;

typedef struct SharedLibDesc_s {
  char *name;
  jint ndx;
  jint numberOfClasses;
  ClassDesc *allClasses;
  jint numberOfNeededLibs;
  struct SharedLibDesc_s **neededLibs;
  struct SharedLibDesc_s *next;
} SharedLibDesc;

SharedLibDesc sharedLibsArray[];

ClassDesc allClasses_sharedLib2[] = {
  {"C1",NULL,&sharedLibsArray[2]},
  {"C2",&allClasses_sharedLib2[0],&sharedLibsArray[2]},
};

SharedLibDesc* neededLibs3[] = {
  &sharedLibsArray[0],&sharedLibsArray[2]
};

SharedLibDesc sharedLibsArray[] = {
  {"L1",0,0,NULL,                  0,NULL,       NULL},
  {"L2",1,0,NULL,                  0,NULL,       &sharedLibsArray[0]},
  {"L3",2,2,allClasses_sharedLib2, 0,NULL,       &sharedLibsArray[1]},
```

```
35    {"L4",3,0,NULL,                2,neededLibs3,&sharedLibsArray[2]}
36  };
37
38  SharedLibDesc *sharedLibs = &sharedLibsArray[3];
39
40  int main () {
41    SharedLibDesc *sharedLib = sharedLibs;
42    while (sharedLib != NULL) {
43      printf("%s ",sharedLib->name);
44      sharedLib = (SharedLibDesc *) sharedLib->next;
45    }
46    printf("  %s",sharedLibsArray[3].neededLibs[1]->name);
47    printf("   %s\n",sharedLibsArray[2].allClasses[1].name);
48  }
```

Es bietet sich in C zunächst an die Datenstrukturen von „unten nach oben“ zu definieren. Also zuerst werden die Klassen definiert und dann die Komponenten, die sie enthalten. Allerdings müssen auch hier einige *Deklarationen* von Variablen erfolgen, bevor diese *definiert* werden. So verweisen die Komponenten auf ihre Klassen und diese wieder zurück auf ihre jeweilige Komponente. In der Zeile 20 wird deshalb im Code die Array-Variable `sharedLibsArray` deklariert, beim Definieren der Klassen benutzt und erst danach definiert.

Doch nun zum Ablauf des gesamten Programms: Nachdem in den Zeilen 1 bis 2 ein zwei nötige Deklarationen aus dem JX-Kern zu finden sind, werden in den Zeilen 4 bis 18 die Strukturen `ClassDesc` und `SharedLibDesc` deklariert. Der Übersichtlichkeit wegen entsprechen sie nicht den vollständigen Deklarationen im JX-Kern, sondern enthalten nur die wesentlichen Datenfelder.

Wie bereits erwähnt wird nun in Zeile 20 die Variable `sharedLibsArray` als Array vom Datentyp `SharedLibDesc` deklariert, um in der Definition von `allClasses_sharedLib2` in den Zeilen 22 bis 25 verwendet werden zu können. Dies ist möglich, da nur Zeiger auf das Array benutzt werden und in C dabei die eigentliche Größe des Arrays keine Rolle spielt. Weiterhin ist wichtig zu erwähnen, dass gerade an dieser Stelle der C-Compiler bereits feste Adressen vergibt! Der Zeiger muss also nicht irgendwie zur Laufzeit durch ein kompliziertes Betrachten der Strukturen erstellt werden, sondern stellt direkt eine Speicheradresse dar. Es werden im Beispiel nur die zwei Klassen `C1` und `C2` erzeugt. Dabei ist ersichtlich, das `C1` als Basisklasse für `C2` fungiert. Es existiert also ein Zeiger im entsprechenden Feld der zweiten Struktur im Array auf die erste Struktur im Array.

In den Zeilen 27 bis 29 ist nun leider (zumindest in ANSI-C und ohne „böse“ Tricks) ein Zwischenschritt nötig. Die Komponente `L4` (siehe Zeile 35) soll abhängen von den Komponenten `L1` (Zeile 32) und `L3` (Zeile 34). Dazu muss in der zur Komponente gehörenden Struktur ein Zeiger auf einen Vektor enthalten sein, der wiederum Zeiger auf die benötigten `SharedLibDesc`-Strukturen im Vektor `sharedLibsArray` enthält. Dafür muss dieser Vektor aber vorher vereinbart worden sein. Dies geschieht hier mit der Vereinbarung der Variable `neededLibs3`.[5]

Diese Variable wird nun mit den anderen bis hierher definierten Variablen und Strukturen in den Zeilen 31 bis 36 im Vektor `sharedLibsArray` zusammen geführt. Die Felder der definierten

[5] Werden beim Linken die nicht benötigten Symbolinformationen verworfen oder das Ergebnis „gestript“, sollte dies jedoch zu keinerlei unnötigen Ballast im Maschinencode führen.

Komponenten sind dabei der Einfachheit halber sehr dünn besetzt! Natürlich besitzt nur eine Variable die oben definierten Klassen, die anderen gar keine. Es gibt auch nur die vorhin beschriebene Abhängigkeit zwischen den Komponenten.

Zeile 38 erzeugt nun eigentlich die einzige für den JX-Kern relevante Variable `sharedLibs`. Aus Sicht des Kerns ist diese nun der Einstieg in die Kette der benötigten Komponenten. Sie zeigt auch, wie beim dynamischen Laden, auf die „letzte" Komponente.

Es ist jedoch auch sofort ersichtlich, dass bei diesem Verfahren viel unnötiger Speicher verschwendet wird. Da die Komponenten ja tatsächlich als Array und nicht nur als verkettete Liste vorliegen, könnte sowohl effizienter darauf zugegriffen werden, als auch die Felder `ndx` und `next` eingespart werden, deren Inhalt ja schon durch die Position im Array eindeutig beschrieben ist. Dies würde jedoch bedeuten, dass alle Stellen im Kern angepasst werden müssten, die explizit oder auch implizit von der verketteten Liste ausgehen. Dies stellt einen erheblichen Aufwand dar und würde nur unnötig zu getrennt zu wartendem Code führen.

Die in den Zeilen 40 bis 48 zu findende `main()`-Funktion „spielt" nur etwas mit dem Code, indem sie die verkettete Liste abläuft und einige spezielle Informationen direkt extrahiert. Sie erzeugt die Ausgabe:

```
L4 L3 L2 L1   L3   C2
```

3.5.2 Primitive Klassen und `java.lang.Object`

Die *primitiven Klassen* (also die Klassen, die die Arbeit mit primitiven Datentypen im objektorientierten Umfeld erleichtern, wie `Integer` oder `Char`) und die Klasse `java.lang.Object` (als abstrakte Basisklasse des gesamten Java-Klassenbaums) werden vom Kern selbst dynamisch erzeugt. Da dies jedoch komplett im C-Code geschieht, kann diese Initialisierung ebenfalls statisch erfolgen. Es gibt dabei kaum Unterschiede zum Vorgehen bei den normalen Klassen.

Hinzu kommt, dass bei diesen Klassen keine *statischen Felder* existieren. Folglich kann nicht nur die Struktur `ClassDesc`, sondern auch die Struktur `Class` für diese Klassen statisch erzeugt werden. Von JX werden letztere zwar dynamisch erzeugt, sie werden aber in die Domain *zero* geladen, was ihrem „shared"-Charakter entspricht.

3.5.3 Spezielle JX-Klassen

Der Kern von JX bietet ein breites Spektrum an Funktionalitäten. Dieses muss JX bieten, da es ein vollwertiges Betriebssystem sein will. Eine Java Virtual Machine bietet in der Regel Zugriffsmöglichkeiten auf das darunter liegende Betriebssystem an und ermöglicht oft auch einen höheren Abstraktionsgrad, um Betriebssystemeigenheiten für den Programmierer transparent zu machen. So verwenden beispielsweise die Betriebssysteme Unix und Windows verschiedene Darstellungen der zur Verfügung stehenden Dateisysteme. Dennoch sind die Konzepte ähnlich und mit etwas Aufwand können Java-Programme mit Hilfe der Java-Bibliotheken entwickelt werden, die auf beiden Systemen ohne Probleme laufen.

Aber nur eine Java Virtual Machine anzubieten ist nicht ausreichend. JX bietet ein breiteres Spektrum an Möglichkeiten, wie z. B. die Verwendung von Domains. Diese Möglichkeiten müssen auch von Java-Programmen aus nutzbar sein.

JX geht sogar noch einen Schritt weiter. Große Teile des Betriebssystem, wie z. B. Treiber, liegen selbst als Komponenten und somit in Java-Code vor. Damit dies möglich ist, muss JX den Java-Klassen viele Angriffsmöglichkeiten bieten.

Der Kern von JX muss eine stark erweiterte JVM bieten, um allen diesen Anforderungen gerecht zu werden. Technisch wird dies dadurch ermöglicht, dass der Kern eigene gewissermaßen *virtuelle* Java-Klassen erstellt und diese den Java-Klassen transparent als gewöhnliche Java-Klassen bereitstellt.

Die Klassen werden dabei vom Kern ähnlich dynamisch erstellt, wie die primitiven Klassen und `java.lang.Object`. Sie jedoch statisch zu erstellen stellt einen sehr hohen Aufwand dar. Außerdem sind diese Klassen ein Bereich von JX, an dem kontinuierlich weitergearbeitet wird. Eine Übertragung auf eine statische Implementierung würde unweigerlich zu einer Aufteilung der Code-Basis führen, was somit wohl nicht wünschenswert wäre.

3.5.4 Arrays und Strings

Arrays (Vektoren) und Strings (Zeichenketten) werden direkt vom Kern unterstützt. Hierbei sind beide natürlich miteinander dahingehend verwoben, dass ein String intern durch ein Array implementiert wird. Die meisten C-Funktionen zum Umgang mit Arrays und Strings sind in `load.c` zu finden und werden vom Kern dem Maschinencode bereitgestellt (siehe hierzu auch Kapitel 3.1 auf Seite 15).

Array-Typen werden vom Kern immer erst dann erstellt, wenn ein Array diesen Typs benötigt wird. Auf Basis des Elementtyps wird dazu eine passende `ArrayDesc`-Struktur erstellt. Diese bekommt dabei einen Platz in der Domain *zero*, wie dies auch für die `ClassDesc`-Strukturen gilt. Auf Basis der `ArrayDesc`-Strukturen können nun wie gewohnt für jede Domain eigene `Class`-Strukturen erzeugt werden, die wiederum innerhalb der Domain instantiiert werden können.

Bei Strings wird ähnlich vorgegangen. Es wird immer nur der gleiche Objekttyp erzeugt, der intern ein Character-Array nutzt. Es kommt allerdings hinzu, dass in den Klassendaten bereits *konstante Zeichenketten* vorgegeben sind. Diese werden beim Laden der Komponenten bereits als String-Objekte in der Domain *zero* erzeugt.

Die Handhabung von Strings und Arrays wird natürlich einfach so beibehalten, bei der Umsetzung nach C. Man könnte hier mit einigem Aufwand dafür sorgen, dass sowohl alle `ArrayDesc`-Strukturen, als auch alle konstanten Zeichenketten, bereits statisch erstellt werden und sofort zur Verfügung stehen. Bei ersteren macht dies wohl keinen Sinn, da für *jede* existierende Klasse eine solche Struktur vorhanden sein müsste. Die höhere Geschwindigkeit steht dadurch in keinem Verhältnis zum erhöhten Platzbedarf und zu den tiefgreifenderen Änderung am JX-Kern, die nötig wären. Bei den konstanten Zeichenketten würde dies letzten Endes bedeuten, statisch Objekte zu erzeugen. Dies ist auf Grund der Struktur des JX-Kerns nicht so einfach und wird deshalb vermieden.

3.5.5 Methodentabellen

In Kapitel 3.3.3 auf Seite 19 wurde dargelegt, dass die Methodentabellen zu den statischen Strukturen gehören. Sie werden also ebenfalls – wie bereits beschrieben – statisch definiert. Wie jedoch die Handhabung der Methoden und damit auch die der Methodentabellen konkret aussieht wird später in Kapitel 3.7 auf Seite 27 erarbeitet werden.

3.5.6 Symbolische Konstanten und Performance

Wie bereits in Kapitel 2.2.1 auf Seite 10 angedeutet, kann man die C-Dateien, die den eigentlich Quellcode enthalten schöner gestalten, indem man die symbolischen Namen aus den

Klassendateien nicht schon auflöst und durch direkte Zeiger auf die Konkreten Datenstrukturen ersetzt, sondern sie zuerst in für den C-Preprozessor akzeptable Ausdrücke umsetzt und diese dann in der zentralen Header-Datei als Konstanten mit der `#define`-Anweisung definiert.

Eine Klasse wird beispielsweise, wie bereits erläutert, durch eine `ClassDesc`-Struktur repräsentiert. Wird nun in JX auf eine Klasse verwiesen, so geschieht dies meist über einen Zeiger auf eine solche Struktur. Im Bytecode findet sich an einer solchen Stelle eine Referenz auf den entsprechenden Eintrag im *Konstanten-Pool*[6], also in diesem Fall der Name der Klasse. Während nun der Bytecode in Maschinencode übersetzt wird, setzt der Maschinencode-Compiler an dieser Stelle ebenfalls eine symbolische Referenz, die beim Laden der Komponente, die den Maschinencode enthält, durch die konkrete Adresse der `ClassDesc`-Struktur ersetzt wird. Dies geschieht für alle Adressen, deren wirkliche Position zur Laufzeit des Compilers nicht bestimmt werden kann. Dieser Vorgang entspricht also dem üblichen Vorgehen, das mit dem Begriff *dynamisches Linken* beschrieben wird.

Das statische JX, das durch den C-Code erzeugt wird, braucht das dynamische Linken jedoch nicht, ja will es gerade eben vermeiden. Die verwendeten *Symbolkonstanten* stellen in gewisser Weise eine Hilfe zum statischen Linken von JX dar. In den Quelldateien werden keine direkten Adressen angegeben. Allerdings werden sie bereits in der Header-Datei den Symbolkonstanten zugewiesen.

Nehmen wir zur Verdeutlichung die Klasse `C1` im Beispiel aus Kapitel 3.5.1 auf Seite 22. In einer C-Quelldatei wollen wir nun überprüfen, ob eine Objektreferenz in der Variable `vR1` eine Instanz dieser Klasse ist. In der C-Quelldatei machen wir das mit Hilfe einer Support-Funktion:

```
vm_instanceof(vR1,JCLASS_C1)
```

Als zweites Argument wird einfach direkt der Name der Klasse verwendet. Das Präfix `JCLASS_` hilft die Namensräume von C und Java zu trennen. Java-Klassennamen könnten sonst leicht mit C-Bezeichnern kollidieren.

In der Header-Datei müssen wir nun die passende Symbolkonstante für das konkrete System (also das aus dem Beispiel) erzeugen:

```
#define JCLASS_C1 allClasses_sharedLib2[0]
```

`JCLASS_C1` wird nun in allen Quelldateien vom Preprozessor durch den Ausdruck `allClasses_sharedLib2[0]` ersetzt. Da diese Datenstruktur statisch erzeugt wurde und auch der verwendete Index eine Konstante ist, wird der Ausdruck durch Übersetzen und Linken mit dem C-Compiler in eine konkrete direkte Adresse umgewandelt.

Der Zugriff auf die Datenstrukturen bleibt somit effizient und auch die Wiederverwendbarkeit des C-Codes erhöht sich, da die Symbolkonstanten nicht konkret mit einer speziellen statischen Anordnung der Datenstrukturen verbunden sind.

3.6 Datenfelder von Klassen und Objekten

Der Zugriff auf Datenfelder von C heraus ist einfach möglich. Für ein Objektfeld sieht er konzeptionell folgendermaßen aus:

[6] Der *Konstanten-Pool* stellt in gewisser Weise die Symbolinformationen, die in Klassendatei enthalten sind, dar.

```
(jint) (OBJ_REF->data[FIELD_OFFSET]) = 1;
```

`OBJ_REF` stellt dabei die Objekt-Referenz (`ObjectDesc*`) dar, die für den Zugriff natürlich vorhanden sein muss. `FIELD_OFFSET` stellt eine Symbolkonstante für den Index im Datenfeld dar und wird vom C-Preprozessor bereits durch einen konkreten Zahlenwert ersetzt. Der Zugriff sollte damit ähnlich schnell, wie bei der Maschinencode-Implementierung laufen.

Der Zugriff auf statische Felder läuft ähnlich ab. Eine Besonderheit in JX ist jedoch die Tatsache, dass durch das Domains-Konzept auch statische Klassendaten einer gewissen „Dynamik“ unterworfen sind. Konkret gibt es mehrere „Instanzen“ einer Klasse im Speicher. In jeder Domain liegen die statischen Daten getrennt von den anderen Domains vor (siehe hierzu auch Kapitel 3.3.1 auf Seite 17).

Gleichzeitig ist der Programmcode jedoch, wie bereits mehrfach erwähnt, nur in einer Form vorhanden, also nicht spezifisch für eine Domain. Der Code muss also selbstständig abfragen im Kontext welcher Domain er gerade ausgeführt wird und kann dann erst das korrekte Datenfeld bestimmen.

In der Quelldatei `vmsupport.c` sind Funktionen definiert, die auch vom Maschinencode benutzt werden um auf statische Felder zuzugreifen (siehe auch Kapitel 3.4 auf Seite 21). Zentral ist hier die Funktion `vm_getStaticsAddr()`. An Hand der gewünschten Klasse und des Offsets des Datenfelds liefert sie die Adresse als `jint`. Zusätzlich führt sie noch einige Überprüfungen aus und wird wohl auch in Zukunft Angriffspunkt für Änderungen sein, die an JX vorgenommen werden. Deshalb habe ich auch hier versucht, diese und ähnliche Funktionen zu benutzten und keinen eigenen C-Code zu entwerfen.

Es soll jedoch hier zumindest das Prinzip vereinfacht verdeutlicht werden:

```
(curdom()->sfields[CLASS_REF->definingLib->ndx])[FIELD_OFFSET] = 1;
```

Die Funktion `curdom()` ermittelt die aktuelle Domain, in der der Code ausgeführt wird, und liefert einen Zeiger darauf zurück.

Für `FIELD_OFFSET` gilt wieder das schon oben für den Objektzugriff gesagte. `CLASS_REF` muss natürlich eine Referenz auf eine `ClassDesc`-Struktur sein, um auf eine Klasse verweisen zu können. Wie bei `FIELD_OFFSET` handelt es sich hier um eine Symbolkonstante, also um einen Namen den der C-Preprozessor in Verbindung mit einer Header-Datei in einen direkten Zugriff auf eine `ClassDesc`-Struktur umwandelt, wie dies in Kapitel 3.5.6 auf Seite 25 an Hand von `JCLASS_C1` gezeigt wurde.

Es wird übrigens ein `jint` Feld gesetzt, um das Beispiel nicht mit einem zusätzlichen *Cast* also einer Typumwandlung auf den korrekten Datentyp noch unübersichtlicher zu machen.

Es sei hier noch erwähnt, dass der Zugriff auf Felder dem Zugriff auf *Arrays* ähnelt. Diese werden hier nicht gesondert behandelt, da sie keine wirklichen Besonderheiten gegenüber einem Feld-Zugriff bereithalten.

3.7 Methoden als Funktionen

Einen weiteren wichtigen Teil der zu ladenden Daten stellt der eigentliche Programmcode in Form von Methoden dar. Der Kern des JX-Betriebssystems ist ein C-Programm und kennt so

als vergleichbares Konzept die für C typischen *Funktionen*. Der Aufruf von Methoden der Java-Klassen durch den Kern ist ebenso möglich, wie bereits der Aufruf von Funktionen des Kerns aus Java-Klassen heraus (siehe Kapitel 3.4 auf Seite 21). Der Maschinencode der Methoden verhält sich zu diesem Zweck ebenso wie Funktionen, die vom C-Compiler erzeugt wurden.

Es liegt somit nahe, die Methoden aus dem Java-Bytecode für die Verwendung im C-Quellcode in gültige C-Funktionen umzuwandeln, damit diese dann entweder direkt oder über die Methodentabellen der Objekte aufgerufen werden können.

3.7.1 Argumente und Rückgabewerte

Da, wie bereits erläutert, sowohl die primitiven als auch die höheren Datentypen (Objekte, Arrays und Strings) von Java im Kern durch C-Datentypen und Strukturen implementiert sind, können diese wie „gewöhnliche" C-Datentypen von Funktionen als Typen für Argumente und Rückgabewerte genutzt werden. Wie dann der Code der Funktion mit diesen Datentypen umgeht, wird in Kapitel 4 ausführlich erläutert.

3.7.2 Einschränkungen von Funktionen in C

Eine Umsetzung der Java-Methoden durch C-Funktionen bringt jedoch auch ein paar Probleme mit sich. Diese beruhen darauf, dass Java-Methoden einige Spezialitäten bieten, die in C-Funktionen nicht zur Verfügung stehen.

3.7.2.1 Überladen von Methoden

Wie in C++ ist es in Java möglich, mehrere Methoden mit dem gleichen Namen zu definieren, die sich jedoch in Anzahl und Datentypen der Argumente unterscheiden. Diese Methoden stellen dann auch tatsächlich unterschiedliche Methoden dar. Der Name der Methode bildet so zusammen mit Anzahl und Datentypen der Argumente eine *Signatur*, über die die Methode erst eindeutig bestimmt werden kann. Dies bezeichnet man als *Überladen von Methoden*.

In C sind Funktionen bereits durch ihren Namen eindeutig identifiziert. Dies hat jedoch zur Folge, dass es nicht möglich ist mehreren Funktionen den gleichen Namen zu geben.

3.7.2.2 Überschreiben von Methoden

Die Objektorientierung von Java bringt es mit sich, dass abgeleitete Klassen die Methoden ihrer Basisklasse überschreiben können. Damit existieren Methoden, deren Signaturen zwar übereinstimmen, die sich jedoch in der Zugehörigkeit zu einer bestimmten Klasse unterscheiden.

Allgemein sind Methoden mit gleicher Signatur aber aus unterschiedlichen Klassen für Java verschiedene Methoden. Klassen und damit auch *Pakete*, die wiederum die Klassen unterteilen, teilen also den Namensraum von Java auf.

Da C (im Gegensatz zu z. B. C++ oder anderen nicht notwendigerweise objektorientierten Sprachen wie Perl oder Pascal) jedoch keinerlei vergleichbares Konzept bietet, wird die bereits schon vorhandene Einschränkung durch das Überladen nochmals verstärkt. Nicht nur die Signatur einer Methode ist entscheidend, sondern auch ihre Zuordnung zu einer Klasse, bzw. zu einem Objekt.

3.7.2.3 Sichtbarkeit von Methoden

Zur Möglichkeit des Überladens und Überschreibens von Funktionen kommt die Tatsache hinzu, dass Java sehr genau die *Sichtbarkeit* einer Methode festlegen kann. Java kann Methoden als `public`, `protected` oder `private` definieren und damit Methoden gezielt zugreifbar machen.

In C existiert nur die Möglichkeit mit Hilfe von `static` eine Funktion als lokal zu einer Quelldatei zu definieren. Dies ist jedoch in JX unbrauchbar, da Funktionen aus verschiedenen Gründen trotzdem global sichtbar sein müssen. So müssen z. B. die Funktionen, an einer globalen Stelle in die Methodentabellen eingetragen werden.

Folglich gibt es kaum eine Möglichkeit die Einstellungen der Sichtbarkeit in einer sinnvollen Form auf den C-Code zu übertragen. Dies geschieht jedoch auch beim Übersetzen nach Maschinencode nicht, sondern muss von Übersetzer, Verifizierer und Laufzeitumgebung überprüft werden. C hätte hier höchstens zusätzliche Sicherheit bieten können.

3.7.3 Eindeutige Funktionsnamen

Das vorangegangene Kapitel zeigt, dass C genau genommen keinerlei Möglichkeit bietet den Namensraum zu trennen, wie es Java durch die Konzepte der Überladung, der Objektorientierung und auch der Sichtbarkeit von Methoden bietet.

Deshalb muss dafür gesorgt werden, dass jede Funktion einen eindeutigen Namen erhält. Am einfachsten ist dies dadurch zu erreichen, dass nicht nur der Methodenname als Funktionsname benutzt wird, sondern gleich die ganze Signatur und zusätzlich noch der Klassenname hinzugefügt wird. Durch diese Angaben ist eine Methode ja auch im Java-Bytecode eindeutig identifiziert.

Zu beachten ist allerdings, dass für C-Funktionen die üblichen Einschränkungen für Namen in C gelten (siehe [9, Seite 184]):

1. Nur Buchstaben und Ziffern sind erlaubt, wobei der Unterstrich „`_`“ als Buchstabe gilt.
2. Erstes Zeichen muss ein Buchstabe sein.
3. Es werden mindestens die ersten 31 Zeichen als signifikant angesehen. Bei externer Bindung können weitere Einschränkungen greifen.

Die Erzeugung der C-Funktionsnamen geschieht sinnvollerweise auf Basis der im Bytecode verwendeten *Deskriptoren*. Das Beispiel eines Methoden-Deskriptors aus der *Java Virtual Machine Specification* [10]:

```
Object mymethod(int i, double d, Thread t)

besitzt den Deskriptor

(IDLjava/lang/Thread;)Ljava/lang/Object;
```

Die Zeichenfolge in der Klammer stellt dabei offensichtlich die Typen der Argumente dar. `I` steht für `int`, `D` für `double` und für die Referenz `L` wird zusätzlich bis zum Strichpunkt der Typ, also die Klasse angegeben. Genau diese Angabe wird für die Signatur benötigt. Nimmt man allerdings den gesamten Methoden-Deskriptor, so bringt das zusätzliche Sicherheit. In

Java sind Methoden mit gleicher Signatur, aber unterschiedlichen Rückgabewerten nämlich verboten. Der Java-Compiler meldet einen Fehler.

Es ist unschwer zu erkennen, dass gewisse Zeichen ersetzt oder entfernt werden müssen, um gültige C-Bezeichner zu erhalten. So können Strichpunkte „;“ z. B. weggelassen und Schrägstriche „/“ durch Unterstriche „_“ ersetzt werden.

Am einfachsten ist es wohl nicht alphanumerische Zeichen prinzipiell durch einen Unterstrich zu ersetzen.[7] Weiterhin gelten für Java-Bezeichner, also auch Klassennamen, die gleichen Einschränkungen wie für C. Verwenden wir demnach den Klassennamen als ersten Teil des erzeugten Funktionsnamens, so ist auch das erste Zeichen ein Buchstabe.

Zwar haben wir jetzt immer gültige Bezeichner, aber wir haben gleichsam wieder gleich lautende Funktionsnamen ermöglicht. Durch das Ersetzen vieler verschiedener Zeichen durch ein einziges oder gar durch das Weglassen von Zeichen, können vorher verschiedene Zeichenketten wieder identisch werden. Es ist jedoch auch offensichtlich, dass dies im gegebenen Fall sehr unwahrscheinlich ist und wohl am ehesten durch „bösen“ Java-Code provoziert werden müsste. Es wurde hier deshalb vorerst auf eine Lösung des Problems verzichtet. Es gibt jedoch relativ einfache Lösungsmöglichkeiten. So könnte an Stelle der Signatur selbst ein Hash-Wert benutzt werden, der aus der Signatur gebildet wird. Durch die üblicherweise verwendeten mathematischen Verfahren wird die Wahrscheinlichkeit einer Übereinstimmung noch stärker reduziert. Eine andere Möglichkeit besteht darin, unterschiedliche Zeichen durch eine unterschiedliche Anzahl Unterstriche zu ersetzen.

Davon abgesehen, dass dies einigen Aufwand bedeutet, taucht auch schnell an ein Problem auf mit der oben genannten dritten Einschränkung von C-Bezeichnern. 31 Zeichen sind nicht viel, wenn man einen Funktionsnamen wie oben beschrieben durch Zusammensetzung erzeugt. Hier kann natürlich das Hash-Verfahren punkten: Zuerst setzt man wie oben beschrieben den Funktionsnamen zusammen und bildet einen Hash-Wert über den gesamten Namen. Ist dieser Hash-Wert z. B. 8 Zeichen lang, dann kürzt man nun den Funktionsnamen ggf. auf eine Länge von $31 - 8 = 23$ Zeichen und hängt den Hash-Wert an das Ergebnis an.

Im Rahmen dieser Arbeit wurde jedoch eine noch einfacherer „Lösung“ bevorzugt. Sowohl der *GNU C Compiler*, als auch der *GNU Linker* ermöglichen die Verwendung beliebig langer Bezeichner. In ANSI-C ist die Anzahl der maximal signifikanten Zeichen auf minimal 31 Zeichen festgelegt. Der Maximalwert ist den C-Compilern und den Linkern jedoch freigestellt. Die meisten modernen Compiler und Linker haben einfach keine Beschränkung. Deshalb wurden im Rahmen dieser Arbeit bzgl. der Länge der Funktionsnamen auch keine Einschränkungen vorgenommen.

3.7.4 Methodenaufruf

Methodenaufrufe in Java unterscheiden sich nur wenig von Funktionsaufrufen in C. Betrachtet man zusätzlich nicht die Sprache Java an sich, sondern wie hier nötig den Java-Bytecode, so bleibt eigentlich kein Unterschied übrig. Die Bindung einer virtuellen Methode an ein spezielles Objekt findet im Bytecode nicht mehr implizit sondern explizit statt, indem die Referenz auf das Objekt direkt angegeben wird. Auch der Maschinencode-Compiler erzeugt letzten Endes Code, der dem entspricht, der vom C-Compiler aus dem Kern erzeugt wird.

[7] Zusätzlich erlaubt Java in Bezeichnern auch Unicode-Buchstaben. Diese müssen natürlich ebenfalls entfernt, bzw. ersetzt werden. Allerdings unterstützt JX selbst im Moment kein Unicode. Es wurde deshalb nicht davon ausgegangen, dass Unicode-Zeichen auftreten können. Letzten Endes hängt hier auch viel vom C-Compiler ab.

Werden folglich alle Vorgaben bzgl. Funktionsnamen aus Kapitel 3.7.3 auf Seite 29 beachtet, hält der Aufruf von Methoden als Funktionen keine größeren Hindernisse mehr bereit. Wie ein Aufruf genau aussieht wird in diesem Kapitel dargelegt.

3.7.4.1 Statischer Methodenaufruf

Der Aufruf von statischen, also Klassenmethoden (Bytecode `invokestatic`) ist vom Compiler sehr einfach zu realisieren. Die aufzurufende Methode ist immer exakt bekannt. Eine Umsetzung in C gelingt einfach durch einen direkten Aufruf der entsprechenden C-Funktion.

Im Konstanten-Pool sind, wie in Kapitel 3.7.3 auf Seite 29 gezeigt, ausreichend Informationen enthalten, um einen eindeutigen Namen für eine Methode zu erstellen. Wird immer die gleiche Abbildung von Klassennamen, Methodenname und Typbezeichner auf die C-Funktion gewählt, werden die richtigen Funktionen immer korrekt gefunden.

3.7.4.2 Virtueller Methodenaufruf

Der Bytecode `invokevirtual` ermöglicht den typischen virtuellen Methodenaufruf. Über die virtuelle Methodentabelle wird die korrekte Methode ermittelt und ausgeführt.

Um diesen Vorgang zu verdeutlichen und zu zeigen, wie der Compiler diesen Aufruf in C-Code übersetzt, der vom JX-Kern benutzt werden kann, soll das folgende Beispiel einen vereinfachten Überblick bieten:

Prinzip eines virtuellen Methodenaufrufs

```
1  typedef void (*code_t) ();
2  typedef signed long jint;
3
4  typedef struct {
5    code_t *vtable;
6    jint data[1];
7  } ObjectDesc;
8
9  jint add1(jint i) {
10   return i+1;
11 }
12
13 jint add(jint a,jint b) {
14   return a+b;
15 }
16
17 code_t myvtable[] = {
18   (code_t) add1,
19   (code_t) add
20 };
21
22 main () {
23   ObjectDesc *obj = (ObjectDesc*) malloc(sizeof(ObjectDesc));
24   obj->vtable = myvtable;
25   printf("%d",((jint (*)(jint)) obj->vtable[0]) (1) );
26   printf("   %d\n",((jint (*)(jint,jint)) obj->vtable[1]) (2,2) );
```

```
27   exit(0);
28 }
```

Zeilen 1 und 2 definieren die beiden benötigten und normalerweise vom JX-Kern bereitgestellten Datentypen. Zeilen 4 bis 7 definieren die bekannte Struktur zur Repräsentation eines Objekts. Für dieses Objekt werden nun in den Zeilen 9 bis 15 Methoden definiert, die dann in den Zeilen 17 bis 20 in einer Methodentabelle eingetragen werden. Es wurde keinerlei Klasse definiert oder eine Referenz auf eine Klasse „vor“ der Methodentabelle hinterlegt, um das Beispiel nicht zu kompliziert zu gestalten.

In der `main()`-Funktion (Zeilen 22 bis 28) wird nun ein Objekt „instantiiert“ und ihm die Methodentabelle zugeordnet (Zeilen 23 und 24). Die Zeilen 25 und 26 führen schließlich die beiden Methoden aus. Interessant ist hier vor allem der *Cast*, also die Typumwandlung der unspezifischen `code_t`-Zeiger in einen an dieser Stelle passenden Zeiger auf die C-Funktion. Da Signatur und Rückgabewert bereits zur Übersetzungszeit bekannt sind, kann der Compiler den Cast bereits in dieser Form erstellen.

Die Ausgabe des Programms sieht folgendermaßen aus:

```
2   4
```

Im Beispiel fehlt ein entscheidender Punkt: Virtuelle Methoden müssen prinzipbedingt immer Zugriff auf die Objekt-Referenz besitzen, über die sie aufgerufen wurden. Während eine Klassenmethode die zugehörige Klasse kennt, kann eine virtuelle Methode das dazugehörige Objekt wegen der dynamischen Bindung nicht kennen. Der Trick mit dem dies in der Regel gelöst wird ist die Weitergabe der Objekt-Referenz als normales Argument an die Funktion. Genau diese Lösung wurde auch beim Maschinencode gewählt, der die Referenz auf den Stack „pusht“. Bei der Umsetzung nach C wird dies selbstverständlich übernommen und jede virtuelle Methode (auch wenn sie statisch aufgerufen wird; siehe nächstes Kapitel) erwartet eine Referenz auf das Objekt (also einen Zeiger auf `ObjektDesc`) als erstes Argument.

3.7.4.3 Spezialfälle

Der Java-Bytecode kennt noch zwei Spezialfälle neben den beiden Methodenaufrufen `invokestatic` und `invokevirtual`.

`invokespecial` ist eigentlich eine Art Mischform zwischen statischem und virtuellem Aufruf. Aufgerufen werden dabei eigentlich virtuelle Methoden, also Methoden, die an ein konkretes Objekt gebunden sind. Jedoch werden sie *nicht* über die virtuellen Methodentabellen bestimmt, sondern wie statische Methoden direkt ermittelt. Diese Form der Ausführung virtueller Methoden wird dazu benutzt Konstruktoren, Aufrufe an die Basisklasse und private Methoden zu implementieren. Von besonderer Bedeutung ist dies auch beim Optimieren. Als eigentlich statisch ermittelte virtuelle Methoden (siehe Kapitel 2.2.2 auf Seite 11) müssen selbstverständlich auf diese Weise aufgerufen werden.

Die Implementierung ist denkbar einfach. Es kann ein direkter Aufruf der C-Funktion erfolgen, das erste Argument beim Aufruf muss jedoch die benötigte Objektreferenz enthalten.

Um eine Interface-Methode aufzurufen wird der Bytecode-Befehl `invokeinterface` verwendet. Er unterscheidet sich kaum von `invokevirtual`, vor allem wenn – wie zur Zeit in JX – keine getrennten virtuellen Methodentabelle für Klassen und Interfaces vorliegen. In JX sind

hier jedoch Besonderheiten für den Aufruf von *Portalen* zu berücksichtigen. Portale bieten auf Basis eines Interfaces eine Art RPC (*Remote Procedure Call*) zwischen unterschiedlichen Domains. Diese könnten sonst auf Grund getrennter Speicherbereiche nur schwer kommunizieren.

3.8 Exceptions

Einen sehr mächtigen Mechanismus von Java, der dem Programmierer erstaunliche Möglichkeiten bietet, stellen die *Exceptions* dar. Sie ermöglichen eine sehr flexible Fehlerbehandlung. Es müssen nicht mehr alle Fehler, die in aufgerufenen Methoden entstehen können, abgefragt werden. Statt dessen werden sie einfach weiter „nach oben durchgereicht“. Der Fehler wird also so lange weiter den Methoden-Stack durchgereicht, bis er in einer Methode abgefangen wird, oder letztendlich zum Abbruch des Programms führt, wenn keine Methode den Fehler abfängt.

Dieses Konzept stellt jedoch das System, das den Code später ausführen muss (im Falle von Java die Java Virtual Machine), vor eine relativ große Herausforderung. Dies gilt vor allem, wenn das System in schnellen Maschinencode übersetzt werden soll. Um herauszufinden, in welcher Methode und an welcher Bytecode-Adresse eine Exception aufgetreten ist, muss es möglich sein, von einer Adresse im Maschinencodes auf diese zurückzuschließen. In JX wird dies mit Hilfe von Tabellen ermöglicht. Zuerst wird ermittelt, in welcher Methode die Exception aufgetreten ist. Für Methoden, in denen Exceptions abgefangen werden existiert eine spezielle Tabelle, die *Exception-Table*. Diese ermöglicht es direkt mit Hilfe der Maschinencode-Adresse zu ermitteln, ob für den Bereich, in dem die Exception auftrat ein auszuführender Code-Block, ein *Exception-Handler*, existiert. Dieser kann nun direkt angesprungen werden. Ist kein Handler vorhanden, wird die nächste Methode auf dem Stack gesucht und für sie der gleiche Test durchgeführt. Dies geht so lange, bis der Stack leer ist oder ein Handler gefunden wurde.

Für die Umsetzung in C-Code stellen die Exceptions jedoch eine deutlich größere Herausforderung dar. Es entsteht in erster Linie ein Problem: Die endgültigen Adressen der Bytecode-Befehle im erzeugten Maschinencode fallen in den Aufgabenbereich des C-Compilers. Weder bei der Übersetzung des Bytecodes nach C, noch beim späteren Laden und Ausführen des Codes sind die Adressen ohne weiteres ermittelbar.

Eine Lösung stellt die in *Toba* [6] verwendete dar. Mit expliziten C-Befehlen wird an den nötigen stellen eine Variable auf die aktuelle Bytecode-Position gesetzt. Die Lösung ist einfach, hat aber den Nachteil, dass diese Variable oft geändert wird. Exceptions stellen aber, wie der Name bereits sagt, Ausnahmen dar. Es wird in dieser Lösung also eine Variable verwaltet, die eher selten benötigt wird.

Vielversprechender erscheint eine andere Lösung, die bereits ansatzweise in JX vorhanden ist. Für die Funktionen im Kern wurde in gewissen Grenzen die Funktionalität von Exceptions verfügbar gemacht. In erster Linie wurde es ermöglicht, dass ermittelt werden kann, in welcher C-Funktion eine Exception ausgelöst wurde. Während der Übersetzungsphase wird der Kern zweimal übersetzt. Zuerst einmal mit Symbolen und dann nochmals ohne. Nach dem ersten Übersetzen werden mit Hilfe eines Perl-Skripts die Funktionsadressen aus den Symbolinformationen extrahiert und in C-Strukturen umgewandelt. Beim zweiten Übersetzen werden diese in den Kern integriert und stehen folglich dem Kern zur Verfügung.

Die Funktion zu bestimmen ist also bereits mit den vorhandenen Fähigkeiten von JX möglich. Es bleibt die exakte Bestimmung der Bytecode-Adressen. Die Lösung liegt hier im Erzeugen passender Symbole im Maschinencode. Dafür können ganz normale *C-Labels* benutzt werden. Leider ist deren Verwendung in den Symboltabellen nicht Standard. Für den GCC gehören

sie zu den speziellen Symbolen, die mit der Option `-g` aktiviert werden und vor allem für den GNU Debugger GDB von Interesse sind. Sie lassen sich zwar nicht mit `objdump` ohne Weiteres extrahieren, jedoch mit `readelf`. Eine Implementation von Exceptions ist folglich möglich.

4 Übersetzen des Bytecodes nach C

Eine zentrale Stelle im Verlauf dieser Arbeit nahm die eigentliche Umsetzung des Java-Bytecodes nach C-Code ein. Der vorhandene Maschinencode-Übersetzer wandelt, wie bereits erwähnt, den Bytecode aus den Klassendateien in eine interne Zwischenrepräsentation um, auf deren Basis der Maschinencode erzeugt wird. Im Wesentlichen bleiben bei der Zwischenrepräsentation jedoch die Bytecode-Befehle erhalten. Der *Virtuelle Operandenstack* des Bytecodes wird entfernt und das Ergebnis dann noch verschiedenen Optimierungen unterzogen.

Für eine Übersetzung nach C spielt jedoch der Java-Bytecode immer noch eine große Rolle. Es gilt Java-Bytecode-Befehle adäquat in C-Code umzuwandeln. Als Referenz für den Bytecode dient die *Java Virtual Machine Specification* [10]. Sie hält die nötigen Spezifikationen insbesondere für Spezialfälle bereit. Weitere, sehr wichtige Anhaltspunkte waren für mich die Quellcodes freier Projekte im Internet, die ähnliche Aufgaben zu bewältigen hatten. Ein erster Linie sei hier der *GNU Java Compiler* (GCJ) als Teil der *GNU Compiler Collection* [8] zu nennen. Im GCC-Quellcode kann in der Datei `libjava/interpret.cc` (Version 3.4.1) die Umsetzung des Java-Bytecodes in C-Befehle betrachtet werden.

Viele Bytecode-Befehle lassen sich direkt umsetzen, andere verlangen mehr Aufmerksamkeit. Einige Sonderfälle sollen nun in diesem Kapitel näher erläutert werden. Zunächst jedoch sind noch ein paar klärende Worte zum Konzept der *IM-Code-Objekte* sinnvoll.

4.1 Konzept der IM-Code-Objekte

In der Zwischenrepräsentation werden die meisten Bytecode-Befehle und auch weitere, neu eingeführte Hilfsbefehle durch spezielle Java-Objekte[1] dargestellt die im Folgenden als *IM-Code-Objekte* bezeichnet werden.

Der Übersetzer wurde parallel zu dieser Arbeit modularisiert, um das Übersetzen in verschiedene Maschinencodes und auch andere Ziele (wie z. B. C-Code) zu ermöglichen. Das Umwandeln des Bytecodes in die Zwischenrepräsentation und das Optimieren derselbigen, laufen für alle Ausgabeziele gleich ab, erst dann werden die Objekte mit der jeweils richtigen `translate`-Methode in die benötigte Ausgabe umgesetzt. Um die Ausgabe von C-Code zu ermöglichen, mussten also die meisten dieser Objekte eine spezielle `translate`-Methode erhalten, die den durch das Objekt repräsentierten Befehl in C-Code ausgibt.

4.2 Verwendung von Support-Funktionen des Kerns

Wie bereits in Kapitel 3.4 auf Seite 21 erwähnt, werden viele Bytecode-Befehle vom Maschinencode-Übersetzer nicht nur einfach in Maschinencode umgewandelt, sondern bedie-

[1] Es sollte vielleicht nochmals darauf hingewiesen werden, dass ja der gesamte Übersetzer, wie auch das restliche Build-System in Java implementiert sind und hier somit von Java-Objekten aus diesem System und nicht von den zu übersetzenden Objekten die Rede ist.

nen sich zur Erfüllung ihrer Aufgaben der Support-Funktionen des Kerns. Einige der Bytecode-Befehle werden sogar direkt durch den Aufruf einer C-Funktion umgesetzt.

Natürlich sollte, ja muss sogar, bei diesen Befehlen auch der C-Code diese Funktionen verwenden. Dies ist meistens einfach durch direktes Aufrufen der Funktion möglich, stellt jedoch manchmal auch eine größere Herausforderung dar. Dies ist z. B. bei der Umsetzung des Bytecodes `multianewarray` der Fall, für den eine C-Funktion verwendet wird, deren Aufruf aus dem Maschinencode heraus, nicht wie unter C üblich erfolgt, sondern mit einem kleinen Trick realisiert wird. Es muss nämlich ein dynamisches `int`-Array übergeben werden. Dieser Trick soll hier nicht näher besprochen werden, es sollte nur erwähnt werden, dass auch beim Aufruf von Kernfunktionen gewisse Besonderheiten beachtet werden müssen.

4.3 Unterschiede zwischen den Datentypen in C und Java

Besonderer Aufmerksamkeit bedürfen für die Umwandlung in C-Code die arithmetischen Operationen auf den primitiven Datentypen. Die primitiven Datentypen selbst werden von JX direkt auf C-Datentypen umgesetzt, die entsprechend über `typedef`-Konstrukte in C deklariert werden. Meistens werden die arithmetischen Operationen jedoch direkt in Maschinencode umgesetzt.[2]

Für die Umsetzung nach C müssen nun die Unterschiede und Gemeinsamkeiten der Operatoren näher Betrachtet werden. Die folgenden Kapitel geben einen (nicht vollständigen) Überblick über die zu beachtenden Besonderheiten.

4.3.1 Bit-Operationen

Die Bit-Operatoren unterscheiden sich bei C und Java in ein paar Punkten. So müssen in Java bei den Verschiebe-Operationen ggf. höherwertige Bits bei der Angabe der Schiebeweite maskiert werden.

Ein besonderes Problem stellt das Verschieben nach rechts bei negativen Werten dar. Während im Bytecode zwischen `shr` (*arithmetisches* Schieben, also Vorzeichen-beachtend) und `ushr` (*logisches* Schieben) unterschieden wird, macht C die Auslegung des Operators `>>` vom System abhängig. Um hier nicht unnötig komplexe, systemabhängige Überlegungen einzubauen, habe ich mich entschlossen, dem zu folgen, wie der Bytecode im GCC-Code interpretiert wird. Dies macht Sinn, da der JX-Code sowieso schon relativ GCC-lastig ist und wohl in erster Linie mit diesem Compiler übersetzt wird. Es wird also angenommen, dass der Operator `>>` *arithmetisch* schiebt.[3]

4.3.2 Fließkomma-Arithmetik

Bei der Fließkomma-Arithmetik hält die *Java Virtual Machine Specification* konkrete Vorgaben für die *Division mit Rest* bereit. Demgegenüber besitzt die Sprache C keinen direkten Operator für diesen Zweck. Der `%`-Operator ist auf die ganzzahligen Datentypen beschränkt.

[2] Eine Ausnahme stellen z. B. die Operationen auf dem Datentyp `long` dar, die als C-Funktionen implementiert sind. Aufgrund der höheren Performance werden diese jedoch nicht beim Übersetzen nach C genutzt, sondern direkt C-Code erzeugt, damit der Funktionsaufruf eingespart werden kann.

[3] Dies deckt sich auch mit der im Kern bei der Implementierung der `lshr`-Operation verwendeten C-Funktion.

In der *Java Virtual Machine Specification* findet sich bei der Beschreibung des Befehls `frem` der folgende Hinweis[4]:

> "The result of an `frem` instruction is not the same as that of the so-called remainder operation defined by IEEE 754. The IEEE 754 'remainder' operation computes the remainder from a rounding division, not a truncating division, and so its behavior is *not* analogous to that of the usual integer remainder operator. Instead, the Java virtual machine defines `frem` to behave in a manner analogous to that of the Java virtual machine integer remainder instructions (`irem` and `lrem`); this may be compared with the C library function `fmod`." [10, Chapter 6 - The Java Virtual Machine Instruction Set]

Aus diesem Grund habe ich mich entschlossen, `frem` und `drem` durch einen Aufruf der Funktion `fmod` zu implementieren.[5]

Die meisten anderen arithmetischen Operationen sind im Wesentlichen direkt in C-Code umsetzbar. Sie sind höchstens in Bereichen problematisch, in denen auch die *Java Virtual Machine Specification* Freiräume lässt.

4.3.3 Umwandlung von Fließkomma- auf Integer-Typen

Während das Umwandeln der Integer-Typen in C und Java identisch ist, kommt es zu Problemen, sobald Fließkomma-Typen mit im Spiel sind. Während in C das Ergebnis meist „nicht definiert" oder „implementationsabhängig" ist (siehe [9, Anhang A.6]), gibt es in der *Java Virtual Machine Specification* [10] klare Definitionen. Folgender generischer C-Code setzt diese Vorgaben um:

```
if (VALUE >= (FROM_TYPE) MAX)
  return MAX;
else if (VALUE <= (FROM_TYPE) MIN)
  return MIN;
else if (VALUE != VALUE) /* VALUE == NaN */
  return 0;
else
  return (TO_TYPE) VALUE;
```

An die Stelle von `VALUE` tritt der umzuwandelnde Wert. `MAX` und `MIN` sind Minimal- und Maximal-Wert des Zieltyps (`TO_TYPE`). `FROM_TYPE` steht für den Ursprungstyp des Wertes.

4.4 Spezielle Probleme und Einschränkungen von C

Bei der Umsetzung des Java-Bytecodes auf C-Code mussten an einigen Stellen spezielle Tricks angewendet werden, da die Programmiersprache C zwar sehr flexibel ist, aber dennoch nicht alles (direkt) bietet, was benötigt wurde.

[4] Analog ist er auch beim Befehl `drem` zu finden

[5] Bemerkenswert ist nebenbei, dass im GCC-Code beide Befehle durch das Aufrufen der internen Funktion `__ieee754_fmod` umgesetzt werden. Dies widerspricht ja aber gerade der Spezifikation.

Bei Beschränkung auf den ANSI-C-Standard kommt es hier zu etwas „unschönen“ Konstruktionen. Anders sieht es mit dem GCC aus. Er besitzt viele Erweiterungen der Sprache C, die gerade in den hier beschriebenen Fällen hilfreich sind. Es wurden jedoch immer beide Möglichkeiten implementiert und sie können per Compiler-Option ausgewählt werden.

4.4.1 Variables `goto`

Im Gegensatz zur Sprache Java selbst, arbeitet der Java-Bytecode sehr viel mit *Sprungbefehlen*. Im Bytecode existieren genau genommen keine anderen Möglichkeiten den Programmfluss zu ändern. Schleifen und bedingte Verzweigungen greifen alle auf den Bytecode-Befehl `goto` zurück. Dies ist allerdings nur konsequent. Schließlich handelt es sich beim Bytecode eher um Maschinencode, als um eine Programmiersprache. Dort sind Sprünge meist ebenfalls die einzige Möglichkeit den Programmfluss zu ändern. Es handelt sich hier also nicht um ein „veraltetes“ Konzept, wie es bei Programmiersprachen gesehen wird.

C ist jedoch (zum Glück) alt genug, um ein `goto` anzubieten. Zwar wird auch in C das Verwenden von `goto` nicht empfohlen[6], für die Umsetzung des Java-Bytecodes ist es jedoch sehr hilfreich. Der Pseudo-Programmcode erzeugt bereits C-typische Sprungmarken für Bytecode-Befehle, die Ziel eines Sprungs sind. Das Bytecode `goto` funktioniert nämlich genau so. Es springt eine feste Bytecode-Adresse an. Besitzen also alle Sprungziele ein C-typisches *Label*, dann kann direkt das `goto` von C für das des Bytecodes benutzt werden.

Leider wird der Sachverhalt jedoch durch einen weiteren Bytecode-Befehl deutlich komplexer. Es handelt sich um den Befehl `ret`, der in Zusammenarbeit mit dem Befehl `jsr` für die Implementierung der `finally`-Anweisung von Java sorgt. Zwar springt auch `ret` (zurück) auf eine Bytecode-Adresse, jedoch liest es das Sprungziel aus einer lokalen Variablen aus. Dies lässt sich nicht mit dem klassischen `goto` in C nachbilden, da dieses nur auf direkt angegebene Sprungziele springt.

4.4.1.1 Lösung für ANSI-C

In ANSI-C gibt es dennoch versteckt die Möglichkeit variable Sprungziele zu implementieren. Dazu wird die `switch`-Anweisung benötigt. Ihre `case`-Sprungmarken bieten die nötige Flexibilität, da sie auf Grund des Wertes einer Variablen angesprungen werden. Es ist jedoch ein bisschen mehr Aufwand erforderlich, wie das folgende Beispiel verdeutlicht:

Variables goto in C

```
1  main() {
2    int target = 0;
3  LABELSWITCH: switch (target) {
4  B0: case 0:
5      goto B2;
6  B1: case 1:
7      printf("passed\n");
8      goto B3;
9  B2: case 2:
10     target = 1;
11     goto LABELSWITCH;
12 B3: case 3:
```

[6] Siehe dazu [9, Seiten 64 und 65].

```
13      exit(0);
14    }
15    exit(1);
16  }
```

Es ist sofort zu erkennen, dass praktisch die gesamte Funktion von einer `switch`-Anweisung umschlossen wird. In den Zeilen 4, 6, 9 und 12 sind immer noch die festen Sprungmarken der Bytecode-Befehle zu sehen. In den Zeilen 5 und 8 werden zwei davon auch ganz normal angesprungen. Die in Zeile 2 definierte Variable `target` ist von zentraler Bedeutung für den variablen Sprung. Sie gibt der großen `switch`-Anweisung das Sprungziel vor. Zu Beginn ist dies eine „0“ und somit wird zum ersten `case`-Label in Zeile 4 gesprungen, das ja mit dem Bytecode-Label `B0` identisch ist. Der entscheidende Code steckt nun in den Zeilen 10 und 11. Wir wollen auf `B1` springen, aber dies eben nicht direkt über das Label. Also setzen wir `target` auf „1“ und springen zu der Sprungmarke `LABELSWITCH`, die uns zurück zur `switch`-Anweisung führt. `target` ist nun „1“ und wir landen folglich in Zeile 6. Der variable Sprung ist vollbracht. Die Variable `target` muss ja nicht auf „1“, sondern kann auf einen beliebigen Wert gesetzt werden. Dies kann selbstverständlich auch eine lokale Variable sein. Eine Implementation von `ret` ist somit möglich.

Das Programm „springt“ also folgendermaßen durch den Code: B0 $\rightarrow$ B2 $\rightarrow$ B1 $\rightarrow$ B3

Es sei noch erwähnt, dass die Grundidee dieser Lösung auf Ideen in *Toba* [6] zurückgeht. Dort wird nahezu die gleiche Technik angewandt.

4.4.1.2 Lösung für den GCC

Die Lösung mit dem GCC sieht ein wenig einfacher aus. Sein `goto` besitzt nämlich die geforderte Fähigkeit bereits. Es ist mit dem GCC möglich einer Variable ein Sprungziel zuzuweisen. Dies sieht folgendermaßen aus:

```
void *label = &&LABEL;
```

Die Variable `label` kann fortan direkt mit `goto` angesprungen werden

```
goto *label;
```

Diese spezielle Form des `goto` (Argument vom Typ `void *`) wird in der GCC-Dokumentation als *„computed goto statement“* bezeichnet. Die Variable wird mit einem Label gleich behandelt und `goto` springt an die Stelle an der das Label `LABEL` gesetzt wurde.

Die komplette Lösung des Problems ist jedoch komplexer, da wir ja auf Basis einer ganzzahligen Variablen, nicht einer Variablen die ein Label „enthält“ springen müssen. Das verlangte ist jedoch relativ leicht zu erreichen:

Variables goto in C

```
1  main() {
2    static void *labels[] = { [1]=&&B1, [2]=&&B2, [3]=&&B3 };
3    goto B2;
4  B1:
5    printf("passed\n");
6    goto B3;
```

```
7  B2:
8    goto *labels[1];
9  B3:
10   exit(0);
11 }
```

Zeile 2 ist von zentraler Bedeutung. Es wird ein Array aus `void`-Zeigern definiert und mit Hilfe der speziellen Syntax mit Zeigern auf die Bytecode-Labels gefüllt. Dabei kommt eine weitere spezielle Syntax des GCC zum Einsatz. Es können nämlich bei der Initialisierung von Arrays direkt die Positionen angegeben werden, an denen der Wert im Array stehen soll. Nicht angegebene Werte werden dabei mit „0“ initialisiert. Dies ist im Beispiel für den Wert an Index „0“ der Fall. Da sich die Sprungziele nicht ändern, wird dieses Array als `static` deklariert, damit es nicht bei jedem Aufruf neu initialisiert werden muss.

In Zeile 8 wird dann der eigentliche Sprung ausgeführt. Die Bytecode-Adresse des Sprungziels ist dabei in der eckigen Klammer zu finden. Hier kann wieder eine lokale Variable zum Einsatz kommen und der Befehl `ret` ist somit vollständig implementiert.

Leider führt das geschilderte Vorgehen jedoch zu einer nicht unerheblichen Platzverschwendung. Erstens wird ein Array erzeugt, das in der Regel viele Lücken durch nicht vorhandene Labels aufweist. Zweitens werden bei einer typischen Verwendung von `ret` und `jsr` nur sehr wenige Labels wirklich variabel angesprungen. Letzteres macht allerdings auch der ANSI-C-Lösung Probleme auf Grund eigentlich unbenutzter `case`-Marken. Für jeden Fall wird wohl individuell festgestellt werden müssen, welche Variante kleiner, bzw. schneller ist, und auf welche der beiden Eigenschaften man besonderen Wert legt.

4.4.2 Anweisungen und Ausdrücke

Wichtig ist in C der Unterschied zwischen *Anweisungen* (*statements*) und *Ausdrücken* (*expressions*). Ein C-Programm besteht zum großen Teil aus *Anweisungen* (Befehlen), die einfach sequentiell ausgeführt werden (siehe hierfür und für Folgendes auch [9]). *Ausdrücke* sind Befehle und Konstrukte, die einen Wert zurück liefern. Dies sind z. B. vom Benutzer definierte Funktionen mit einem Rückgabewert oder arithmetische Operationen. Das Auswerten eines Ausdrucks kann als Anweisung verwendet werden. Demgegenüber sind Anweisungen wie Auswahl-Anweisungen (`if`, `select`), Wiederholungs-Anweisungen (`while`, `for`) oder auch Blöcke (`{...}`) keine Ausdrücke. Dies verhindert einige Konstrukte, wie sie in anderen Sprachen (z. B. Perl) durchaus üblich sind.

Dieser Umstand macht eine effiziente Umsetzung einiger IM-Code-Objekte schwierig. Vor allem, wenn es darum geht Überprüfungen durchzuführen, bevor der Wert eines Ausdrucks z. B. in einer Anweisung einer Variablen zugewiesen wird. Als einfaches Beispiel kann das IM-Code-Objekt `IMArrayLength` dienen.

Bevor die Länge des Arrays, dessen `ArrayDesc` über den Zeiger in der Variablen `vR1` zu finden sei, ermittelt wird, soll überprüft werden, ob `vR1` eine gültige Referenz darstellt. Dazu wird sie auf `null` und auf eine *Magic Number* hin überprüft.

Betrachten wir zunächst den einfach Fall, ohne dass irgendetwas überprüft wird. Da das IM-Code-Objekt an einer beliebigen Stelle als Ausdruck benutzt werden kann, könnte der erzeugte Code z. B. folgendermaßen aussehen:

```
vi0 = ((ArrayDesc *) vR1)->size;
```

Wünschenswert wäre es nun, die Überprüfung folgendermaßen einzubauen (Achtung! Ungültiger C-Code):

```
vi0 = {
  if (vR1==NULL) throw_NullPointerException(NULL);
  if (getObjMagic(vR1)!=MAGIC_OBJECT) exceptionHandler(THROW_MagicNumber);
  ((ArrayDesc *) vR1)->size;
};
```

Bevor also `vR1` als Zeiger auf `ArrayDesc` verwendet wird, wird die Überprüfung vorgenommen. Da diese (theoretisch effiziente) Variante jedoch nicht funktioniert, muss nach anderen Lösungswegen gesucht werden.[7]

4.4.2.1 Lösung für ANSI-C

Mit normalem ANSI-C lässt sich das Problem mit Hilfe von Funktionsaufrufen lösen:

```
jint null_check(jint ref) {
  if (ref==NULL) throw_NullPointerException(NULL);
  return ref;
}

jint magic_check(jint ref) {
  if (getObjMagic(ref)!=MAGIC_OBJECT) exceptionHandler(THROW_MagicNumber);
  return ref;
}

...

vi0 = ( (ArrayDesc *) magic_check(null_check(vR1)) )->size;
```

Die Referenz wird also von einer Funktion zur nächsten „durchgereicht“ und dabei überprüft. Ob der C-Compiler in der Lage ist dies über das Inlining von Code zu optimieren muss für jeden Fall einzeln betrachtet werden.

4.4.2.2 Lösung für den GCC

Die Entwickler des GCC haben das Problem mit Ausdrücken und Anweisungen erkannt und bieten als Lösung eine Erweiterung des C-Standards an, die unter der Überschrift *Statements and Declarations in Expressions* in der Dokumentation beschrieben wird.

Die Syntax kommt hierbei der oben gewünschten sehr nahe:

```
vi0 = ({
  if (vR1==NULL) throw_NullPointerException(NULL);
  if (getObjMagic(vR1)!=MAGIC_OBJECT) exceptionHandler(THROW_MagicNumber);
```

[7] Erwähnt werden sollte noch, dass auch der so genannte „Komma-Operator“ von C keine Abhilfe schafft. Mit ihm können nur *Ausdrücke*, aber keine *Anweisungen*, wie das `if` im Beispiel, aneinander gehängt werden.

```
  ((ArrayDesc *) vR1)->size;
});
```

Nur ein Paar runde Klammern um die geschweiften Blockklammern sind hinzugekommen.

Sollte der Operand nicht wie im Beispiel eine Variable (vR1), sondern ein komplexerer Ausdruck sein, so könnte dieses Verfahren auf Grund der Mehrfachausführung des Ausdrucks zu einer deutlichen Verlangsamung des Codes führen. Die Auswahl eines der beiden Verfahren ist also wieder situationsabhängig.

4.5 Probleme mit Seiteneffekten

Wie der Aufmerksame Leser vielleicht bereits in Kapitel 4.4.2.2 auf der vorherigen Seite bemerkt hat, kann es bei der Übersetzung nach C zu einem Problem mit *Seiteneffekten* (*side effects*) kommen.

Ein Operand, der ein Array darstellt muss selbstverständlich nicht zwingend z. B. eine Variable sein, sondern kann selbst ein Ausdruck sein. Wird also vor dem Bestimmen der Länge des Arrays, zu dem eben jener Ausdruck auswertet, der Ausdruck schon dahingehend überprüft, ob er eine gültige Referenz liefert, so wird er ja bereits ausgewertet. Dies geschieht bei jeder Überprüfung und dann natürlich noch später zum eigentlichen Bestimmen der Länge. Hat der Ausdruck nun Seiteneffekte, führt diese Mehrfachauswertung zu einem Verhalten, das nicht dem des ursprünglichen Codes entspricht.

Wie bereits erwähnt, tritt dies praktisch nur an Stellen auf, an denen ein Ausdruck überprüft wird, bevor er benutzt wird.

Um hier Probleme zu vermeiden gibt es im Wesentlichen zwei Herangehensweisen. Es liegt auf der Hand, dass die Lösung, die weiter oben für ANSI-C beschrieben wurde und mit geschachtelten Funktionen arbeitet dieses Problem nicht aufweist. Sie sollte also als portable Lösung immer funktionieren.

Bei der Lösung für den GCC sieht der Sachverhalt anders aus. Hier ist es zu empfehlen, dass, sollten Seiteneffekte auftreten, der Ausdruck nur einmal ausgewertet und das Ergebnis in einer temporären Variable gespeichert wird. Diese wird dann getestet und als Ergebnis verwendet. Der Compiler bietet eine Abfrage auf Seiteneffekte eines Operanden sogar zur Compile-Zeit an. Dies wird genutzt um zu entscheiden, ob eine temporäre Variable zu erzeugen ist oder eben nicht.

5 Ergebnisse und Ausblick

Im letzten Kapitel sollen nun die Ergebnisse präsentiert und auf weitere Möglichkeit eingegangen werden, die der Ansatz mit sich bringt, den Java-Bytecode nach C zu übersetzen. Dabei sollen sowohl Vor- als auch Nachteile des Ansatzes in Kapitel 5.1, bzw. 5.2 betrachtet werden. Als Ausblick soll das Augenmerk sowohl auf relativ leicht zu implementierende weitere Optimierungen (Kapitel 5.3), als auch auf mögliche andere Herangehensweisen (Kapitel 5.4) gerichtet werden.

5.1 Vorteile gegenüber dem bisherigen Maschinencode-Übersetzer

In der Einleitung wurde schon auf die zu erwartenden Vorteile des C-Compilers hingewiesen, um die Motivation zu verdeutlichen. In diesem Kapitel sollen nochmals konkrete Ergebnisse aufgeführt und bewertet werden, um ein abschließendes Bild zu ermöglichen. Das nächste Kapitel (5.2) wird dann zusätzlich noch die Nachteile den hier genannten Vorteilen gegenüberstellen.

5.1.1 Zusätzliche Optimierungen durch den C-Compiler

Jeder Compiler versucht bei der Umsetzung des Quellcodes in den Zielcode das erzeugte Ergebnis zu optimieren. Dies tut auch der Maschinencode-Übersetzer von JX bereits auf Basis der Zwischenrepräsentation, die also auch schon Gegenstand etlicher Optimierungen ist (siehe [1]). Bei der tatsächlichen Umsetzung der Zwischenrepräsentation in Maschinencode können zusätzlich noch weitere meist maschinenabhängige Optimierungen vorgenommen werden.

Wird die Zwischenrepräsentation jedoch in C-Code umgewandelt, so wird der Code streng genommen einer weiteren Übersetzungsstufe unterworfen. Der C-Compiler fängt eigentlich aus seiner Sicht wieder bei Null an. Er parst den (aus seiner Sicht) Quellcode, wandelt ihn in eine interne Repräsentation um, optimiert diese und gibt dann erst Maschinencode aus. Moderne C-Compiler (wie der GCC) enthalten viele Jahre Erfahrung im Optimieren von Code. Die bereits optimierte Zwischenrepräsentation lässt sich auf diese Weise nochmals verbessern.

Auch bei der endgültigen Erzeugung des Maschinencodes stehen dem C-Compiler viele Möglichkeiten zur Verfügung den Code auf die gewünschte Architektur hin zu optimieren. Diese Überlegung bringt uns auch gleich zum nächsten Punkt.

5.1.2 Bessere Portierbarkeit

Die Verwendung eines C-Compilers bringt nicht nur hervorragende Optimierungsmöglichkeiten mit sich, sondern ermöglicht, wie im Falle des GCC, auch den Code in Maschinencode für verschiedene Architekturen umzusetzen.

Zwar stellt die Sprache C aus heutiger Sicht eine sehr Hardware-nahe Programmierung dar, sie bietet aber dennoch eine relative hohe Abstraktionsebene. Ursprünglich auch zu diesem Zweck entwickelt bietet C den Vorteil einer hohen Plattform-Unabhängigkeit. Gerade mit dem

GCC ist das Erzeugen von Code für sehr viele Architekturen möglich. Zusätzlich ist es so auch noch möglich den Code auf die jeweilige Architektur hin zu optimieren.

In JX steht man mit dem Maschinencode-Übersetzer vor dem Problem, dass für jede Architektur, die unterstützt werden soll, ein eigenes *Backend* für den Compiler entwickelt werden muss. Erzeugt man indessen C-Code, so wird dieser Teil des Vorgangs dem C-Compiler aufgebürdet. Es gilt dann also nur noch einen geeigneten C-Compiler zu finden.

Eine Einschränkung muss hier allerdings gemacht werden: Der JX-Kern besteht selbst zu einem Teil aus Maschinencode. Trotz architekturunabhängigen C-Code für den Java-Bytecode, muss der Kern noch auf neue Architekturen angepasst werden

Weiterhin fällt auf, dass Teile des Kerns nicht maximal kompatibel zu ANSI-C sind. So wird zum Beispiel das Schlüsselwort `__inline__` benutzt, das nur dem GCC bekannt ist. Auch für die Verwendung eines anderen C-Compilers muss also der Kern angepasst werden.

Der von mir geschriebene Bytecode-zu-C-Übersetzer sollte jedoch größtenteils ANSI-C-Code erzeugen.[1]

5.1.3 Einsparungen durch statisches Linken

JX wurde von Anfang an mit besonderen Hinblick auf den Einsatz in *Embedded Systems* entwickelt. Zwar wurde auch Java ursprünglich von Sun Microsystems für diesen Zweck entwickelt, wurde jedoch lange Zeit nicht in diesem Bereich eingesetzt. Auf Grund der hohen Dynamik von Java zur Ausführungszeit und dem damit verbundenen hohen Anspruch an die Ressourcen des Systems, wurde im Bereich der eingebetteten Systeme weiterhin in C und Assembler Programmiert.

JX zeigt gerade hier Möglichkeiten auf, die bei der Verwendung von Java zu Optimierungszwecken gerade für kleinere, schwächere Architekturen möglich sind. An Stelle des direkten Interpretierens des Bytecodes durch das System, erzeugt bereits der Compiler optimierten Maschinencode, um das laufende System zu entlasten.

Aber gerade für den Einsatz in Embedded Systems besteht eine weitere Optimierungsmöglichkeit, die JX bislang nicht nutzt. Im laufenden Betrieb können eingebettete Systeme nur selten neue Software in Form von Modulen nachladen, bzw. dürfen es gar nicht, um nicht die Stabilität des Systems unnötig zu gefährden. Selbst ausgewachsene Systeme für große Server-Systeme wie Linux kämpfen heutzutage immer noch oft mit Problemen ihres Modulkonzepts, die vor allem beim Entladen oder gar Ersetzen von Modulen auftauchen.

JX besitzt durch das Komponenten-Konzept zwar ebenfalls eine sehr intelligente Form von Modul-Konzept, es wird jedoch meist nur dazu genutzt bereits beim Starten des Systems die benötigten Komponenten zu laden. Danach werden in der Regel keine weiteren Komponenten nachgeladen.

Dieser Umstand, der, wie bereits erläutert, für eingebettete Systeme durchaus normal ist, könnte dafür genutzt werden ein Laden der Komponenten nicht erst beim Starten des Systems vorzunehmen, sondern, da die benötigten Komponenten ja bekannt sind, die Komponenten dem Kern schon beim Bau des Systems in einem direkt verwendbaren Zustand hinzuzufügen. Damit entfallen einige Arbeitsschritte und es wird Platz gespart:

- Der Code zum Laden der Komponenten muss nicht mehr ausgeführt werden. Dies verringert die Startgeschwindigkeit des Systems erheblich, da das Laden einer Komponenten

[1] In Kapitel 4.4 auf Seite 37 wurde gezeigt, dass teilweise dennoch spezielle Erweiterungen des GCCs optional verwendet werden könne.

einem kompletten Link-Prozess gleich kommt. Symbole müssen aufwendig aufgelöst und der Code angepasst werden.

- Während des Parsens und Linkens wird der gesamte Code eingelesen und beschreibende Strukturen im Speicher erstellt. Dies ist nur nötig, weil die Informationen, die in der Komponenten-Datei in einem bestimmten Modulformat vorliegen, natürlich erst in die passenden C-Strukturen umgesetzt werden müssen. Darauf zu verzichten spart Zeit und Speicher.
- Der Code selbst, der zum Laden benutzt wird, kann entfallen und damit eingespart werden.

In Kapitel 3 auf Seite 15 wurde ausführlich gezeigt, wie durch das statische Linken des C-Codes eben jener statische Aufbau des Systems ermöglicht wird. Die C-Strukturen werden direkt eingebunden und stehen beim Starten sofort zur Verfügung. Symbole werden bereits beim Übersetzen und statischen Linken vollständig aufgelöst.

Aber auch weitere Optimierungen sind möglich. Kapitel 2.2.2 auf Seite 11 beschreibt bereits die Erkennung so genannter *system final* Methoden, also Methoden, die dadurch als `final` deklariert werden können, dass sie nicht mehr innerhalb des geschlossenen Systems überschrieben werden. Bei den in der Praxis oft vorhandenen relativ flachen Klassenbäumen, führt dies zu einer erheblich geringeren Anzahl an virtuellen Methodenaufrufen.

5.1.4 Volle Unterstützung von `float` und `double`

Der Maschinencode von JX bietet bisher nur eine eingeschränkte Unterstützung des Java-Datentyps `float` und der Datentyp `double` wird gar nicht unterstützt. Durch die Verwendung von eines C-Compilers wird eine Unterstützung deutlich vereinfacht. Die meisten Operationen für diese Datentypen bringt der C-Compiler selbst mit. Nur einige spezielle Fälle waren besonders zu beachten (siehe Kapitel 4.3 auf Seite 36).

Der Compiler sollte die Datentypen `float` und `double` also vollständig unterstützen.

5.2 Nachteile der Übersetzung nach C

Das Übersetzen des Bytecodes nach C bringt jedoch auch einige Nachteile mit sich, die nun im Folgenden näher betrachtet werden sollen.

5.2.1 Statischeres System

Das in Kapitel 5.1.3 auf der vorherigen Seite als Vorteil aufgeführte statische Linken kann natürlich auch leicht als Nachteil gesehen werden: Das statisch gelinkte JX-System ist nicht mehr in der Lage dynamisch Code nachzuladen. Dies schränkt die Flexibilität stark ein, ist aber wiederum notwendige Bedingung für viele der vorgenommenen Optimierungen.

Das statische Linken als Vor- oder Nachteil zu sehen bleibt letzten Endes „Geschmackssache“ und ist auch vom gewünschten Einsatzzweck abhängig. Wer dynamisch Code nachladen will, der muss wohl ein dynamischen System benutzen. In Kapitel 5.4 auf Seite 47 werde ich mich nochmals mit diesem Problem befassen und einige Lösungen betrachten, die für JX das Dilemma lösen könnten.

5.2.2 Kern nicht für Nutzung von C-Code optimiert

Vergleicht man den JX-Kern mit einem komplett für die C-Nutzung optimierten System wie *Toba* [6] oder *Java C Virtual Machine* [5], so fällt auf, dass große Teile des Kerns eigentlich nicht für die Benutzung von Komponenten und Klassen auf C-Basis optimiert sind. Einige Datenstrukturen und Algorithmen ließen sich deutlich besser an die Anforderungen anpassen. Allerdings müssten hierfür sehr weitreichende Änderungen am Kern vorgenommen werden, um auch wirklich einen deutlichen Effekt zu erzielen. Im Kapitel 5.3 werden einige Punkte hierzu angesprochen.

5.2.3 Einfache Garbage Collection

Der JX-Kern besitzt eine sehr effiziente Implementation einer *Garbage Collection*. Diese basiert jedoch auf so genannten *Stackmaps*, die vom Übersetzer erstellt und vom Kern benutzt werden. Sie ermöglichen es auf dem Methodenstack vorhandene Referenzen zu finden. Weitere Informationen hierzu sind bei [1, Kapitel 3.2.4, Seite 24] zu finden.

Durch das Zwischenschalten des C-Compilers ist es nahezu unmöglich dieses Konzept der Stackmaps weiter zu benutzen. Deshalb muss auf eine einfachere, ältere Implementation einer Garbage Collection für JX zurückgegriffen werden.

5.3 Weitere Optimierungen

Der augenblickliche Status des hier beschriebenen Systems lässt noch einige relativ nahe liegende Optimierungen zu. Diese sind jedoch nur mit einem erheblichen Aufwand realisierbar. Im Rahmen dieser Arbeit wurde deshalb darauf verzichtet.

5.3.1 Effizienteres Nutzen der statischen Datenstrukturen

Wie in Kapitel 3.5.1 auf Seite 22 angedeutet besteht eine Optimierungsmöglichkeit durch die dort beschriebene statische Initialisierung der Daten von Klassen und Komponenten. Diese liegen eben nicht nur wie im normalen JX in einer verketteten Liste, sondern gleichzeitig auch als Array vor. Mit diesem Array lässt sich natürlich effizienter arbeiten, wie mit einer Liste, die auch noch über den Arbeitsspeicher verstreut sein kann. Zusätzlich wurde bereits erwähnt, dass auch die zur Verkettung notwendigen Felder eingespart werden könnten. Mit etwas Aufwand hätten wohl auch alle Datenstrukturen effizienter implementiert werden können, wäre eine statische Initialisierung direkt im *C-Code* von vornherein vorgesehen gewesen.

Alle diese Möglichkeiten sind jedoch mit vielen und tief greifenden Änderungen am Kern von JX verbunden. Die von JX erzeugten Klassen benötigen weiterhin dynamische Strukturen (siehe Kapitel 3.5.3 auf Seite 24). Sie müssten entweder komplett statisch erzeugt werden, oder andere Datenstrukturen verwenden. Selbiges gilt auch für Arrays und Strings (Kapitel 3.5.4 auf Seite 25): Weitere Optimierungen wären jedoch auf diese Weise auch bei Arrays und Strings möglich.

Weitere Bemühungen in diese Richtung wurden jedoch unterlassen, da das Verhältnis zwischen Aufwand und Ergebnis zu groß schien und das Ergebnis außerdem relativ starken Einschränkungen in Sachen Flexibilität unterworfen gewesen wäre.

5.3.2 Für C optimierte Garbage Collection

In Kapitel 5.2.3 auf der vorherigen Seite wurde gezeigt, dass die verwendete Garbage Collection nicht dem Optimum entspricht, das JX zu bieten hat. Es wäre also sinnvoll eine eigene Garbage Collection zu entwickeln oder die bestehende anzupassen, sodass sie für den C-Code benutzbar wird. Eine effektive und flexible Garbage Collection wurde zum Beispiel für die *Java C Virtual Machine* [5] entwickelt.

5.4 Weiterführende Ansätze

Diese Arbeit hatte zum Ziel die grundlegende Funktionalität des Kerns weder stark einzuschränken, noch viele Änderungen an ihm vorzunehmen. Weitere Möglichkeiten zur Optimierung bietet aber eben die nähere Betrachtung solcher Aspekte. Die Überlegungen und Vorschläge hier stellen vor allem Ansätze dar, die eigentlich weiter vom JX-System fortführen und sich ohne Weiteres kaum auf Basis des aktuellen Kerns implementieren lassen.

5.4.1 Mehr statische Strukturen

Das Domain-Konzept von JX sorgt dafür, dass viele dynamische Strukturen entstehen, die sich nicht in statische überführen lassen. So wurde bereits gezeigt, das alle in Java eigentlich statischen Klassendaten durch die getrennten Domains in gewisser Weise auch in verschiedenen dynamisch generierten Instanzen vorliegen. Für ein möglichst kleines und effizientes System können jedoch weitere Untersuchungen in diese Richtung unternommen werden.

Dass es von Vorteil wäre auf das Domain-System zu verzichten, will man ein kleineres System erstellen, liegt auf der Hand. Aber man könnte auch versuchen das im Moment dynamische System statischer zu gestalten. Man könnte z. B. ermöglichen, dass ein dynamisches Generieren und Laden von Domains nicht möglich ist, jedoch dafür sorgen, dass die Domains bereits beim Übersetzen und Linken vorgegeben werden können.

Spinnt man den Faden weiter, könnte man noch weitere eigentlich dynamische Strukturen bereits statisch generieren, wie z. B. Objekte, die immer benötigt werden. Aber man wird hier wohl schnell an Grenzen von Sinn und Zweckmäßigkeit stoßen.

5.4.2 Mehr dynamische Strukturen

In die andere Richtung bietet JX noch viel Spielraum für Erweiterungen. Das hier beschriebene System ist ja statisch entworfen. Man könnte nun versuchen wieder eine stärkere Dynamik einzuführen. Letzten Endes könnte es wieder möglich gemacht werden dynamisch Komponenten nachzuladen. Sehr praktisch wäre es so z. B. eine gewisse Menge an Komponenten statisch in das System einzubinden, es jedoch zu ermöglichen weitere Komponenten nachzuladen. Hier könnte man in zwei Richtungen weitere Überlegungen anstellen.

Das Laden von Maschinencode-Komponenten (JLL-Dateien) sollte eigentlich mit relativ geringem Aufwand realisierbar sein, da am Kern ja wenig Änderungen vorgenommen wurden. Die Zeit war leider zu knapp um noch mit konkreten Überlegungen und Tests zu beginnen. Natürlich ist trotzdem zu bedenken, dass einige der eingeführten Optimierungen wieder rückgängig gemacht werden müssen. Der Code zum Laden der Komponenten muss z. B. wieder verfügbar werden. Auch können Optimierungen, wie die Erkennung von Methoden als *system final*, nicht mehr angewandt werden, wenn das System nicht mehr statisch vorliegt.

Aber nicht nur die Verwendung von JLL-Dateien wäre interessant. Es wäre auch möglich, ein eigenes Modulformat zu implementieren, das es ermöglicht aus dem C-Code erzeugte Maschinencode-Komponenten zu verwenden. Also ein ähnliches Konzept, wie es beim *GNU Java Compiler* [8] oder bei der *Java C Virtual Machine* [5] verfolgt wird. Man müsste also ein Linker-Format wie *ELF* verwenden und könnte auch die C-Version von JX damit modularisieren. Auf Grund der stärkeren Verknüpfung von C und dem Linker-Format wäre diese Methode wohl um einiges effizienter, als die Verwendung von JLL-Dateien. Eine gleichzeitige Benutzbarkeit beider Verfahren wäre natürlich die optimale Lösung ...

5.4.3 Stärkere Ausrichtung auf C

Kapitel 5.2.2 auf Seite 46 legte bereits dar, dass der JX-Kern eigentlich nicht auf die Benutzung durch C-Code optimiert wurde. Gerade hier liegen bei weitreichenden Änderungen am System noch viele weitere Möglichkeiten.

Ein Punkt wäre die Implementierung der Datenfelder von Objekten und Klassen. Im Moment ist das JX-System sehr stark auf bestimmte 32-Bit-Strukturen ausgelegt. Hier könnte man für mehr Freiheiten auf die Fähigkeiten von C zurückgreifen. Die Datenfelder von Objekten und Klassen könnten direkt durch C-Strukturen und -Vektoren umgesetzt werden. Deren konkrete Speicherbelegung kann vom C-Compiler leichter modifiziert werden. So könnten spezielle Eigenschaften spezieller Architekturen besser berücksichtigt werden, aber auch leicht mit unterschiedlichen Speicherlayouts experimentiert werden, wenn der C-Compiler hier für die Architektur spezielle Parameter vorsieht.

Selbiges gilt z. B. auch für unterschiedliche Möglichkeiten den Aufruf von Funktionen zu implementieren, wie dies bei den meisten Architekturen möglich ist. Maschinencode-Komponenten und JX-Kern müssen hier eine gemeinsame Sprache sprechen. Liegen Kern und Komponenten als C-Code vor, so kann über Compiler-Optionen das Verfahren geändert werden.

Bei Architekturen mit vielen unterschiedlichen Varianten, wie beim IBM-PC (i386, i486, i586, i686, Pentium, Pentium-MMX, Pentium-Pro, K6, K6-2, K6-3, Athlon usw.), ergeben sich selbstverständlich weitere Optimierungen durch maßgeschneiderte Systeme für die Zielarchitektur, wenn der C-Compiler in der Lage ist optimierten Code zu erzeugen.

Auch noch vermisste Fähigkeiten von JX, wie die UTF8-Unterstützung, ließen sich über vorhandene C-Bibliotheken ohne größere Probleme nachrüsten.

Ob solche Überlegungen wiederum für JX überhaupt Sinn machen ist eine andere Frage. Statt dessen wäre vielleicht eher anzuraten auf ein vorhandenes System wie *Java C Virtual Machine* [5] die Konzepte von JX zu übertragen.

Literaturverzeichnis

[1] Christian Wawersich; *Design und Implementierung eines Profilers und optimierenden Compilers für das Betriebssystem JX*; Diplomarbeit; Universität Erlangen-Nürnberg, Institut für Informatik; April 2001; URL `http://www4.informatik.uni-erlangen.de/Projects/JX/publications/DA-I4-01-05-Wawersich.pdf`.

[2] Hans Kopp; *Design und Implementierung eines maschinenunabhängigen Just-in-Time-Compilers für Java*; Diplomarbeit; Universität Erlangen-Nürnberg, Institut für Informatik; Oktober 1998; URL `http://www4.informatik.uni-erlangen.de/Projects/JX/publications/DA-I4-98-04-Kopp.ps.gz`.

[3] *Translating Java to C*; Java-zu-C-Compiler-Projekt auf Basis des EDG-Compiler, soll auch mal Bytecode nach C übersetzen können; URL `http://www.dinkumware.com/jproject.html`.

[4] *Java2C Translator*; Master-Thesis von Minh The Au; URL `http://www.csse.monash.edu.au/hons/projects/1999/MinhThe.Au/`.

[5] *Java C Virtual Machine*; JVM, die auf C basiert (verwendet Soot, GCC, einen eingebauten ELF-Object-Loader und GNU Classpath); URL `http://jcvm.sourceforge.net/`.

[6] *Toba*; Bytecode-zu-C-Übersetzer für Java 1.1, wird nicht weiter entwickelt; URL `http://www.cs.arizona.edu/sumatra/toba/`.

[7] David Flanagan; *Java in a Nutshell – Deutsche Ausgabe für Java 1.2 und 1.3*; O'Reilly Verlag; 2000.

[8] *The GNU Compiler Collection*; Informationen zum GNU C-Compiler, zum Java-Compiler (GCJ) und die Software-Pakete selbst; URL `http://gcc.gnu.org/`.

[9] Dennis M. Ritchie Brian W. Kernighan; *Programmieren in C – Zweite Ausgabe – ANSI-C*; Carl Hanser Verlag; 1990.

[10] Frank Yellin Tim Lindholm; *The Java^{TM} Virtual Machine Specification – Second Edition*; Sun Microsystems, Inc.; 1999; URL `http://java.sun.com/docs/books/vmspec/index.html`.

Abbildungsverzeichnis

2.1 Schema das Build-Prozesses 10
2.2 Vereinfachtes Klassendiagramm des modifizierten Build-Systems 12
2.3 Überblick über die Komponenten des modifizierte Build-Systems 13

3.1 Schematischer Ablauf der Funktion `load()` in `load.c` 17
3.2 Die Datenstrukturen für Komponenten, Klassen und Domains 18
3.3 Statische und domainspezifisch Datenstrukturen der Komponenten 19
3.4 Klassen, Objekte, Methodentabellen und deren Beziehungen 20