Statistische Analyse von Java-Classfiles

Denis N. Antonioli and Markus Pilz

Institut für Informatik, Universität Zürich
{antonioli, pilz}@ifi.unizh.ch

„It is a capital mistake to theorize before one has data."

C. Doyle, Scandal in Bohemia

Zusammenfassung Die breite Akzeptanz von Java als Netzwerkprogrammiersprache haben das Java-Classfile-Format zu der wohl am weitesten verbreiteten, portablen Zwischenrepräsentation für Programme gemacht. Dieser Beitrag fasst die wichtigsten Resultate zusammen, die bei der Analyse von 4016 unterschiedlichen Classfiles hinsichtlich Dateigrösse und Verwendung der Bytecode-Instruktionen gemacht wurden.

1 Einführung

Seit seiner Einführung Anfangs 1996 erfreut sich Java [3] einer enormen Popularität, die mindestens teilweise auf das Versprechen uneingeschränkter Programmportabilität zurückzuführen ist. Diese von Sun trefflich mit *write once, run everywhere* umschriebene Eigenschaft erreicht Java über (1) die Compilation der Programme für eine virtuelle Maschine, (2) eine standardisierte Klassenbibliothek und (3) über die Definition eines eigenen, plattformunabhängigen Distributionsformats — das sog. Classfile-Format. Classfiles sind gleichzeitig Objektcode-Dateien der Java Virtual Machine (z.B. für einen Bytecode-Interpreter) und interne Programmrepräsentationen (z.B. für einen JIT-Compiler). Ausserdem sollten sie möglichst klein sein, da sie während der Programmausführung über ein Netzwerk geladen werden können und ihre Grösse die Ladezeit wesentlich beeinflusst.

Dies alles macht Java zum grössten Experiment mit einer portablen Zwischenrepräsentation seit USCD-Pascal [8]. Zwei Jahre nach der Einführung des Classfile-Formats ist es an der Zeit, eine erste Auswertung dieses Experiments vorzunehmen. Solche Analysen sind nicht nur für Compilerbauer von Interesse, sondern helfen auch, die Java Virtual Machine besser zu verstehen.

Dieser Beitrag untersucht Eigenschaften des Java-Classfile-Formats und versucht, folgende Fragen zu beantworten:

1. Stimmt es, dass Java Programme in ein kompaktes und effizientes Zwischenformat übersetzt werden? Was sind typische Grössen von Classfiles?
2. Ein Classfile besteht aus einer Reihe von Tabellen (pools) variabler Grösse. Welchen Anteil haben die einzelnen Tabellen an der Gesamtgrösse eines Classfiles?

3. Die Java Virtual Machine definiert einen reichen Instruktionssatz mit über 200 verschiedenen Instruktionen. Werden diese alle genutzt und wie häufig?

Dieser Artikel ist folgendermassen aufgebaut: Abschnitt 2 erläutert die logische Struktur von Classfiles, Abschnitt 3 beschreibt die durchgeführten Untersuchungen, Abschnitt 4 diskutiert die drei wichtigsten Resultate ausführlich, und Abschnitt 5 fasst die gefundenen Eigenschaften zusammen.

2 Das Java-Classfile-Format

Java-Applikationen werden nicht wie in traditionellen Systemen üblich als monolithische Objektdateien (executables) verteilt, sondern als eine Menge von Classfiles. Dies erlaubt es unterschiedlichen Programmen, die gleichen Klassenbibliotheken zu benutzen, wodurch Sekundärspeicher gespart werden kann. Das Laufzeitsystem kann den Hauptspeicherbedarf senken, indem es das Laden einer Klasse hinausgezögert, bis diese das erste Mal gebraucht wird. Ein weiterer Vorteil ist die vereinfachte Wartung, da einzelne Klassen geändert werden können, ohne das die Programme neu gelinkt werden müssen.

Um Klassen zur Laufzeit dynamisch laden und linken zu können, braucht die virtuelle Maschine Typen- und Symbolinformationen. Diese werden durch den Compiler erzeugt und neben den Bytecode-Instruktionen im Classfile gespeichert. Im folgenden beschreiben wir die Struktur des Classfiles, so wie wir sie unseren Analysen zugrundegelegt haben. Für eine ausführliche Beschreibung des Classfile-Formats verweisen wir auf [5]. Ein Classfile besteht aus einem Header und vier Tabellen: Konstanten-, Klassen-, Variablen-, und Methodentabelle. Die Grösse des Header ist fest, die der Tabellen variabel.

Header. Der Header identifiziert eine Datei als Classfile. Er enthält die Kennzahl (magic number) `0xCAFEBABE` und die Versionsnummer des Classfile-Formats. Der Header ist immer 8 Byte gross.

Klassentabelle. Die Klassentabelle enthält Informationen zu der gespeicherten Klasse. Neben Informationen zur Vererbung und der unterstützten Interfaces, welche der `class...extends...implements` Deklaration im Quellcode entsprechen, enthält die Klassentabelle die Anzahl der Variablen, die Anzahl der Methoden, die Anzahl der Konstanten[1] und eine Attributetabelle. In der Attributetabelle können zusätzliche Informationen zur Klasse gespeichert werden.

Konstantentabelle. Die Konstantentabelle (constant pool) enthält die gesamte symbolische Information, insbesondere die numerischen Konstanten, die Zeichenketten und die Informationen, die zum Linken der Klasse benötigt werden. Die Tabelle enthält verschiedene Arten von `CONSTANT` Einträge unterschiedlicher Grösse. Es sind dies:

[1] Für unsere Analyse haben wir Felder, welche die Anzahl der Einträge in den Tabellen enthalten, der Klassentabelle zugerechnet, damit leere Tabellen auch mit Grösse 0 in der Statistik erscheinen

1. `Utf8` Einträge, welche Folgen von Unicode-Zeichen speichern. Die Zeichen sind in einem leicht modifizierten UTF-8 Format codiert [9].
2. `Integer`, `Long`, `Float` und `Double` Einträge, welche numerische Konstanten des entsprechenden Java-Basistyps aufnehmen.
3. `String` Einträge, welche `java.lang.String` Objekte repräsentieren. Die Einträge verweisen auf einen `Utf8` Eintrag, der die eigentliche Zeichenkette enthält.
4. `Class` Einträge, welche Klassen oder Interfaces repräsentieren. Die Einträge verweisen auf einen `Utf8` Eintrag, welcher den Klassennamen enthält.
5. `NameAndType` Einträge, welche den Namen und den Typ einer Variablen oder einer Methode beschreiben. Die Einträge verweisen jeweils auf zwei `Utf8` Einträge; der eine enthält den Namen, der andere codiert den Typ.
6. `Methodref`, `InterfaceMethodref` und `Fieldref` Einträge, welche Variablen und Methoden mit der Klasse verbinden, zu der sie gehören, indem sie auf einen `Class` und auf einen `NameAndType` Eintrag verweisen.

Variablentabelle. Die Einträge in der Variablentabelle (field pool) beschreiben die Klassen- und Instanzvariablen. Jeder Eintrag beschreibt eine Variable und enthält ihren Namen, ihren Typ, die Zugriffsberechtigungen (access flags) und eine Attributetabelle, welche mit jener in der Klassentabelle vergleichbar ist.

Methodentabelle. Die Methodentabelle (method pool) enthält die Methode zur Initialiserung der Klasse, alle Konstruktoren und alle regulären Methoden. Jeder Eintrag beschreibt eine Methode und enthält ihren Namen, ihren Typ, die Zugriffsberechtigungen und eine Attributetabelle. Die Bytecode-Instruktionen der Methode sind in einem Attribut mit Namen `Code` in der Attributetabelle gespeichert.

3 Die statistische Analyse

Für die Untersuchung haben wir ein Programm geschrieben, welches Classfiles liest und ihren Inhalt analysiert. Das Programm wurde unabhängig einmal in C++ und einmal in Java implementiert und zur Analyse der sechs in Tabelle 1 aufgelisteten Applikationen verwendet. Wir haben versucht, Applikationen und Klassenbibliotheken von verschiedenen Herstellern zu finden. Insgesamt wurden 4016 unterschiedliche Klassen analysiert.

4 Die Resultate

Wir diskutieren hier die drei wichtigsten Resultate und verweisen den interessierten Leser auf [1] für eine detailliertere Beschreibung aller durchgeführten Untersuchungen.

Tabelle1. Die untersuchten Programme

Programm	Hersteller	Version	Grösse [Klassen]
Java Developer's Kit (JDK)	JavaSoft	1.1.5	1622
Java Workshop (JWS)	Sun	2.0	1408
JavaCC	SunTest	0.7.1	134
Java Generic Library (JGL)	ObjectSpace	3.0	262
classViewer	ifi		51

4.1 Die Grösse von Classfiles

[2] zeigt, dass es auf modernen Rechnerarchitekturen möglich ist, schneller Maschinencode aus einer kompakten Zwischendarstellung zu erzeugen, als eine herkömmliche Objektdatei von der Festplatte zu laden. Die Zeit, die nötig ist, um ein Programm zu starten, wird wesentlichen durch die Grösse der Objektdatei bestimmt, da der Ladevorgang durch den Zugriff auf die Festplatte und nicht durch die Prozessorleistung dominiert wird (i/o bound). Wird die Datei wie im Falle von Java über ein langsames Netzwerk geladen, so wird die Grösse der Objektdatei zum dominierenden Faktor. Dieser Abschnitt untersucht die typische Grösse von Classfiles.

Die durchschnittliche Grösse $\bar{x}$ eines Classfiles haben wir mit 4'541 Byte gemessen. Dieser Wert ist aber das Resultat einiger weniger, sehr grosser Dateien, was auch die Standardabweichung σ von 13'017 Byte erklärt (Tab. 2). Der Median x_{Me} liegt unter 2000 Byte, was bedeutet, dass die Hälfte aller Classfiles sehr klein ist. Tatschlich sind 95% aller Classfiles als klein zu bezeichnen, und nur die restlichen, zum Teil sehr grossen Klassen heben das arithmetische Mittel und die Standardabweichung stark an.

Tabelle2. Dateigrösse [Byte]

$\bar{x}$	σ	Median	80%	90%	95%	97.5%	Max.
4'541	13'017	1'746	4'898	8'302	12'650	23'240	365'470

Abb. 1 stellt die Verteilung der Dateigrössen mit Hilfe einer Lorenzkurve grafisch dar. Die Lorenzkurve gibt den prozentualen Anteil der kumulierten, relativen Klassengrössen an der Programmgesamtgrösse nach anwachsender Umverteilung (increasing rearangement) an. Die zwei dargestellten Programme sind die Extrema in unserer Untersuchung. Die kleine Anzahl grosser Classfiles, welche in Tab. 2 bereits durch das überproportionale Wachstum der maximalen

Dateigrösse in der 95%, 97.5% und 100% Quantile sichtbar wurde, zeigt sich deutlich im scharfen Knick bei 95%. Die Steigung der Kurve verdeutlicht, um wie viel diese Dateien grösser sind: 50% aller Classfiles tragen nur gerade 20% zur Gesamtgrösse des Programms bei, während umgekehrt 50% der gesamten Programmgrösse nur durch 10% aller Klassen verursacht wird.

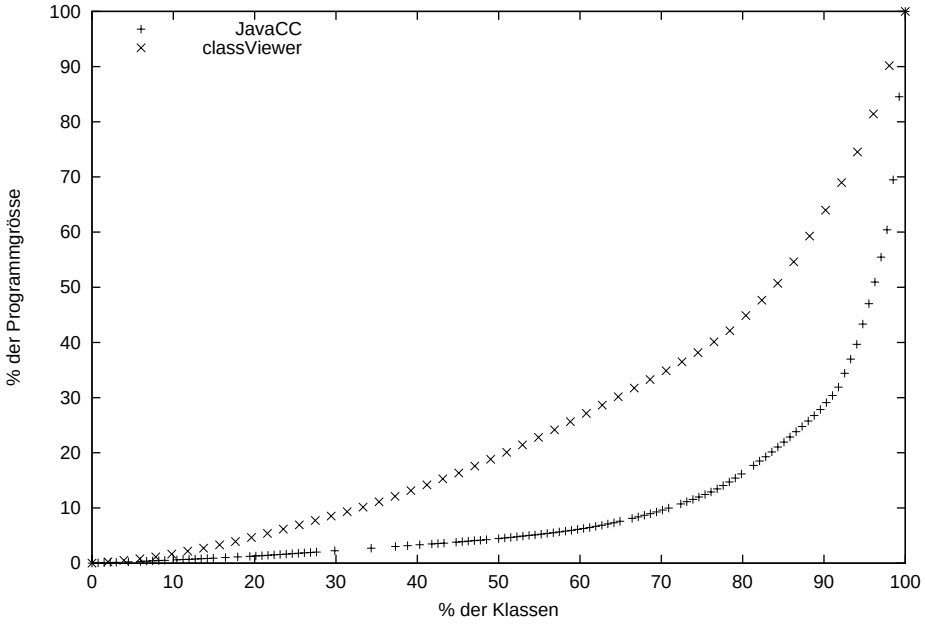

Abbildung1. Lorenzkurve der Dateigrössen

4.2 Der Inhalt der Konstantentabelle

Die Konstantentabelle enthält alle Konstanten, die in den Bytecode-Instruktionen verwendet werden, sowie die gesamte symbolische Information, die für das dynamische Linken und die Typenprüfung nötig ist. Die Konstantentabelle kann aus allen Teilen des Classfiles referenziert werden und kann weitere Einträge enthalten. Wir unterteilen die Einträge in drei Gruppen:

1. *Konstanten.* Diese Gruppe enthält alle Konstanten im engeren Sinne, die in Bytecode-Instruktionen verwendet werden. Es sind dies `Integer`, `Long`, `Float`, `Double` und `String` Einträge. Alle durch `String` Einträge referenzierten `Utf8` Einträge zählen ebenfalls zu dieser Gruppe.
2. *Typen- und Linkinformation.* Diese Gruppe enthält alle Einträge mit denen das Typsystem codiert wird und welche durch das Laufzeitsystem zum Laden, Linken und Verifizieren der Klasse benötigt werden. Es sind dies alle

`Class`, `NameAndType`, `Methodref`, `InterfaceMethodref` und `Fieldref` Einträge. Alle durch diese referenzierten `Utf8` Einträge zählen ebenfalls zu dieser Gruppe.

3. *Andere.* Die Konstantentabelle kann aus allen Teilen des Classfiles referenziert werden und Einträge enthalten, die weder den Konstanten noch den Typen- und Linkinformationen zugerechnet werden können. Diese Einträge bilden die Gruppe *Andere*.

Sun's `javac` Compiler minimiert die Grösse der Konstantentabelle, indem er gleiche Einträge (Konstanten) nur einmal schreibt und alle Referenzen entsprechend anpasst. Aus diesem Grund prüfen wir, dass `Utf8` Einträge nicht versehentlich doppelt gezählt werden. Falls eine Zeichenkette in unterschiedlichen Gruppen verwendet wird, zählen wir sie zur *Typen- und Linkinformation*. Obwohl ein Java-Compiler frei ist, eigene Attribute zu definieren, die beliebige `CONSTANT` Einträge referenzieren, haben wir die tatsächliche Verwendung der Einträge in der Konstantentabelle nicht überprüft. Von allen in [5] definierten Attributen ist das `Exceptions` Attribut das einzige, das nicht nur `Utf8` Einträge referenziert: das Attribut benutzt `Class` Einträge, um den Typ der Ausnahmebedingung zu spezifizieren, und diese Verwendung entspricht unserer Klassifikation.

Um die Bedeutung der einzelnen Gruppen zu bestimmen, haben wir alle Einträge in der Konstantentabelle klassifiziert und die Grösse aller in einer Gruppe enthalten Einträge zusammengezählt. Die absolute Grösse haben wir als prozentualen Anteil an der Gesamtgrösse der Konstantentabelle ausgedrückt.

Tabelle3. Anteil der Gruppen an der Konstantentabelle [%]

	$\bar{x}$	σ	Median	Min.	Max.
Konstanten	7.76	15.56	3.05	0.00	99.86
Typen- und Link-informationen	60.48	19.45	64.80	0.07	96.48
Andere	31.76	19.03	28.19	0.07	97.73

Wie aus Tab. 3 ersichtlich, belegen die *Typen- und Linkinformationen* im Mittel 60% der Konstantentabelle. Obwohl die Standardabweichung mit 19% relativ gross ist, scheint das arithmetische Mittel aussagekräftig, wird es doch durch einen Median von 65% gestützt. Der hohe Wert für die Gruppe *Typen- und Linkinformation* verdeutlicht, wie wichtig in Java Informationen sind, die durch den Compiler ermittelt werden und die zur Laufzeit für die Bytecode-Verifikation und das dynamische Linken verfügbar sein müssen. Die Klassen mit einem geringen Anteil an *Typen- und Linkinformationen* referenzieren in den Bytecode-Instruktionen viele Konstanten und Basisdatentypen statt andere Klassen.

Konstanten belegen entweder einen sehr kleinen Teil der Konstantentabelle ($x_{Me} = 3\%$) oder beinahe die gesamte Tabelle. Der kleine Anteil lässt sich mit der geringen Grösse der Basisdatentypen (5 – 9 Byte) und der geringen Anzahl effektiv verwendeter Konstanten erklären, während sehr viele Konstanten typisch sind für Klassen, die zur Modellierung von Aufzählungstypen verwendet werden (enumeration). Solche Klassen kommen in den untersuchten Programmen unterschiedlich oft vor.

Die restlichen 32% werden von der Gruppe *Andere* belegt. Diese Einträge, welche Sekundärinformation wie Kommentare, Debug-Information usw. enthalten, sind nicht nötig, um die Klasse zu benutzen oder um die enthaltenen Bytecode-Instruktionen auszuführen.

4.3 Die Grösse von Bytecode-Instruktionen

Die Java Virtual Machine definiert einen Instruktionssatz, der 212 unterschiedliche Instruktionen enthält. Seine Grösse wird durch zwei Besonderheiten erklärt: einerseits existieren viele Instruktionen in mehreren Versionen, welche sich nur durch die Typen ihrer Operanden unterscheiden (z.B. `iadd`, `ladd`, `fadd` und `dadd`). Andererseits existieren spezielle Instruktionen, bei denen kleine Konstanten implizit und nicht über eine Referenz auf einen `CONSTANT` Eintrag spezifiziert werden (z.B. `iload_1`).

Tabelle4. Aufteilung der Instruktionen im Instruktionssatz nach ihrer Grösse

Grösse [Byte]	Anzahl Instruktionen	% des Instruktionssatzes
1	147	69%
2	14	7%
3	33	16%
4	12	6%
5	3	1%
6	1	3%
9 oder mehr	2	1%

Eine Instruktion der Java Virtual Machine besteht aus einem Opcode, der von Null oder mehr Byte gefolgt wird, welche die Operanden spezifizieren. Viele Instruktionen bestehen nur aus einem 1 Byte Opcode, da die Operanden implizit durch den Stack gegeben sind (Tab. 4). Der Opcode ist für 200 Instruktionen in einem Byte und für die restlichen 12 Instruktionen in zwei Byte codiert. Der Opcode spezifiziert die auszuführende Operation, die Anzahl Operanden und die Grösse der Instruktion; `lookupswitch` und `tableswitch` sind die einzigen Instruktionen mit unterschiedlicher Anzahl Operanden und variabler Grösse: sie sind mindestens 9 bzw. 13 Byte gross.

Tabelle5. Grösse der Instruktionen [Byte]

$\bar{x}$	σ	Median	Min.	Max.
1.96	2.39	2.00	1.00	822.00

Wir haben die Grösse der in den untersuchten Klassen gespeicherten Bytecode-Instruktionen gemessen. Tab. 5 zeigt, dass die durchschnittliche Grösse knapp unter 2 Byte liegt. Das sehr grosse Maximum von 822 Byte entspricht einer `tableswitch` Instruktion die aus einer einzigen `switch` Anweisung mit über 200 `case` Einträgen compiliert wurde.

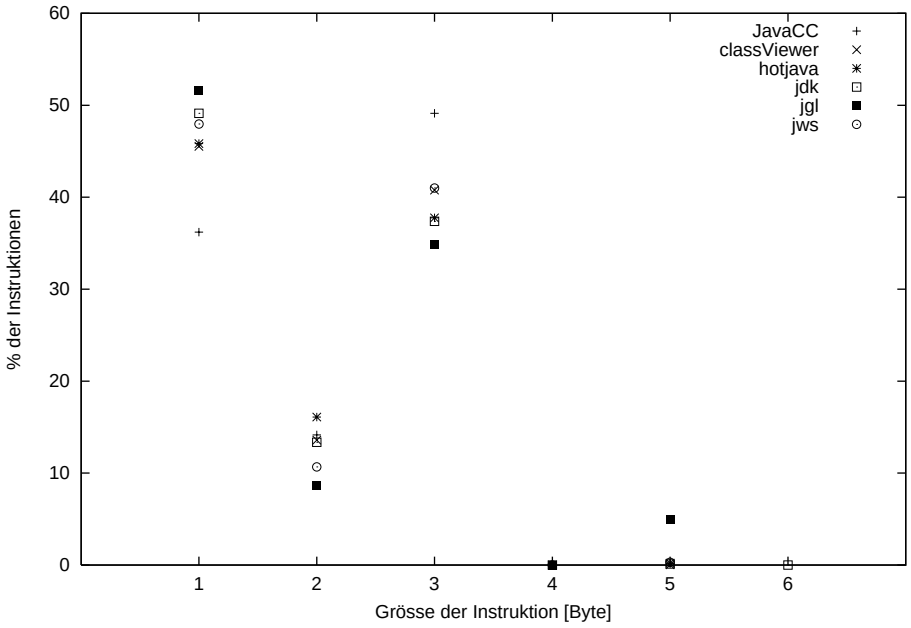

Abbildung2. Aufteilung der Instruktionen in den Programmen nach ihrer Grösse

Wir habe alle Instruktionen in den untersuchten Programmen nach ihrer Grösse klassifiziert, gezählt und ihren prozentualen Anteil am Total aller Instruktionen bestimmt. In Abb. 2 wird dies für alle Bytecode-Instruktionen mit fester Grösse graphisch dargestellt. Vergleichen wir diese Resultate mit den Werten aus Tab. 4, so fällt auf, dass 2 Byte und insbesonders 3 Byte Instruktionen übervertreten sind. Während nur 7% bzw. 16% aller Instruktionen des Instruk-

tionssatzes 2 Byte bzw. 3 Byte lang sind, stellen diese beiden Gruppen 14% bzw. 40% aller Instruktionen in den untersuchten Programmen.

5 Zusammenfassung

Wir haben versucht, drei grundsätzliche Fragen zu beantworten: Was kann man über die Grösse von Classfiles sagen? Wie tragen die einzelnen Tabellen (pools) zur Gesamtgrösse bei? Wie werden die einzelnen Bytecode-Instruktionen von gängigen Compilern verwendet?

Dazu haben wir sechs Programme untersucht, die aus 4016 unterschiedlichen Klassen bestehen. Alle Klassen sind wahrscheinlich mit Sun's `javac` Compiler übersetzt worden; aber wir möchten in Zukunft auch den Einfluss unterschiedlicher Compiler auf die Grösse von Classfiles und die Verwendung der Bytecode-Instruktionen untersuchen. Alle sechs Programme sind das, was wir Java-Applikationen der ersten Generation nennen: grosse, monolithische Programme, welche auf einer Maschine fest installiert sind und ähnlich genutzt werden wie Programme, die in konventionellen Sprachen wie C oder C++ geschrieben sind. Eine zweite Generation von Java-Applikationen ist angekündigt [7], welche aus kleinen, versatilen Komponenten besteht, den sog. Java Beans [4]. Der Einfluss des Komponentenmodells auf die Grösse der Classfiles und der einzelnen Tabellen wurde noch nicht untersucht.

Die hier und weitere in [1] beschriebenen Analysen des Java-Classfile-Formats brachten folgende Tatsachen ans Licht:

1. Im allgemeinen sind Java-Classfiles klein: 50% aller Dateien sind kleiner als 2'000 Byte, 80% sind kleiner als 6'000 Byte, und 95% sind kleiner als 13'000 Byte. Gleichzeitig enthalten Programme aber auch einige sehr grosse Classfiles: Die grösste Klasse in unserer Testmenge war 365'470 Byte. Die Grösse der Classfiles ist ein wichtiger Wert, da das Laden der Klassen durch die Ein-/Ausgabe bestimmt wird und ihre Grösse die Startup-Zeit bestimmt.
2. Der grösste Teil eines Java-Classfiles ist die Konstantentabelle, welche im Mittel 61% der Datei belegt, und nicht die Methodentabelle, welche nur gerade 33% der Datei ausfüllt. Alle anderen Tabellen teilen sich die restlichen 5%.
3. Ungefähr 32% des Classfiles werden durch die Gruppe *Andere* belegt. Diese Sekundärdaten wie z.B. der Name der Quelldatei und die Zuordnungstabellen zwischen Bytecode-Offset und Zeilennummer können gelöscht werden, wodurch sich die Grösse der Datei (und damit die Ladezeit) um 1/3 verkleinert, ohne dass die Klasse etwas von ihrer eigentlichen Funktionalität einbüsst.
4. Die Bytecode-Instruktionen machen im Mittel nur gerade 12% eines Classfiles aus. Gemessen an einer Datei, aus der die Sekundärdaten gelöscht wurden, macht der Bytecode 18% aus.
5. Die Durchschnittsgrösse einer Bytecode-Instruktion liegt etwas unter 2 Byte, was in starkem Kontrast zum Namen Bytecode steht.

6. Eine typische Klasse verwendet nur gerade 25 unterschiedliche Bytecode-Instruktionen. Der höchste Wert in den untersuchten Klassen ist 113, während die Java Virtual Machine 212 unterschiedliche Instruktionen definiert.
7. Die Häufigkeit, mit welcher die einzelnen Instruktionen verwendet werden, schwankt sehr stark. Wir haben fünf Instruktionen gefunden, welche mindestens in zwei Programmen mehr als 5% aller Bytecode-Instruktionen ausmachen.
8. Das theoretische Minimum für die im Durchschnitt nötigen Bits zur Codierung aller in einer Klasse vorkommenden Instruktionen ist 4 Bit statt der tatsächlich verwendeten 8 bzw. 16 Bit.

Literatur

1. Antonioli, D. N., Pilz, M.: Analysis of the Java Class File Format. Technical Report 98.04, Department of Computer Science, University of Zürich (1998)
2. Franz, M., Kistler, T.: Slim Binaries. Technical Report, Department of Computer Science, University of California at Irvine (1996)
3. Gosling, J., Joy, B., Steele, G.: The Java Language Specification. Addison Wesley (1996)
4. Halfhill, T. R.: JavaBeans: Cross-Platform Components. Byte, 22(1), (Jan. 1997) 74
5. Lindholm, T., Yellin, F.: The Java Virtual Machine Specification. Addison Wesley (1996)
6. Linthicum, D. S.: Java Evolves. Byte, 23(1), (Jan. 1998) 60
7. Pountain, D.: The Component Enterprise. Byte, 22(5), (May 1997) 93–98
8. Shillington, K. A., Ackland, G. M.: UCSD Pascal Version 1.5. Institute for Information Systems, University of California, San Diego (1978)
9. The Unicode Consortium: The Unicode Standard, Version 2.0, Addison-Wesley (1996)