Analysis of the Java Class File Format

Denis N. Antonioli, Markus Pilz
Department of Computer Science
University of Zurich
(antonioli, pilz)@ifi.unizh.ch

"It is a capital mistake to theorize before one has data."
C. Doyle, Scandal in Bohemia

Since its release early in 1996, Java [7] enjoys a tremendous popularity [10]. Next to the original tools provided by Sun, a wealth of competing compilers [4, 12] and virtual machines [1, 2, 17] appeared in a relatively short time. Beside its interest as a programming platform, Java may qualify as the most publicized and distributed experiment in intermediate language since the days of UCSD-Pascal [15]. Two years after its inception, it is time to harvest the first results. The purpose of this paper is to gather statistics on the Java class files, which serve in Java both as an object file for the Java Virtual Machine and as an intermediate program representation for cross-platform delivery. Such statistics are of interests to both the developers of Java tools and the designers of intermediate languages.

The goal of this work is to study the properties of the Java class file format and particularly to answer these questions:

1. It is often claimed that programs written in Java are compiled to a compact and efficient format. What are the sizes of typical class files?
2. The class file is structured around several parts of variable length. How do they add to the size of the file?
3. The Java Virtual Machine defines a rich instruction set of over 200 instructions. Are all these instructions used? How often?

The paper is thus structured as follows: Section 1.0 lists the programs that were analysed. The size of the classes are examined in section 2.0. Section 3.0 analyses the components of the class file, with special attention given to the constant part and the attributes. Section 4.0 studies the instruction set of the virtual machine. Finally, section 5.0 summarizes the findings and concludes the article.

1.0 The data set

The experience was conducted over six different programs totalling 4016 unique classes. We tried to find both applications and libraries written at different organizations. The set contains:

1. The Java Developer's Kit

   The version 1.1.5 of the Java Developer's Kit (JDK) consists of 1622 different classes that make the standard Java class library, the tools (`java`, `javac`, `javap`, `javadoc`, `jar`,…) and some libraries proprietary to Sun.

2. HotJava

   HotJava is the web browser that introduced Java to the world. The version 1.1 of Sun's HotJava browser also contains the Java Runtime Environment, a subset of the JDK. We measured only the 539 classes unique to the browser.

3. Java WorkShop

   Java WorkShop (JWS) is an integrated developer's environment (IDE) for Java entirely written in Java. The JWS is a commercial product developed by Sun. The version 2.0 contains an IDE, a dedicated run-time system and a complete JDK. We measured only the 1408 classes unique to the IDE.

4. JavaCC

   JavaCC is a compiler generator written in Java. JavaCC is developed at SunTest and is freely available [14]. The version 0.7.1 of JavaCC contains 134 classes.

5. Java Generic Library

   The Java Generic Library (JGL) is a collection of containers and algorithms designed in the spirit of STL [16]. JGL is intended to supplement the JDK. The JGL is developed by ObjectSpace and is freely available [3]. JGL version 3.0 contains 262 classes.

6. classViewer

   ClassViewer is the Java program one of the author wrote to perform measurements over the class files. It is an example of the kind of utilities written in Java; classViewer contains 51 classes.

To our knowledge, it is not possible to determine the compiler used to produce the classes analysed. Although it is most probable that the first four programs have been compiled with some version of Sun's `javac`, we are only certain that the 6th program was compiled with the compiler included in the version 1.1.5 of Sun's JDK.

2.0 Size of the files

[6] observed that, with modern architectures, the time to load a file from a local hard disk is much larger than the time necessary to generate code on-the-fly. The size of the binary file determines the program start-up time because the program load time is bound by the speed of the I/O operations. The time necessary to bring the program into main memory becomes even more important when, as is intended with Java, the code is loaded over a network. This section explores the typical size of class files.

Table 1 contains the average, the standard deviation (stdev) and the median size of the classes in the data set. The four quantile columns are the maximum size for the respective percentage of all the files and the last column indicates the size of the biggest class in each program. The median is under 2'000 bytes, which reveals a large number of very small files, but the large value of the standard deviation hints at an important number of bigger files. The first three quantiles show that, although

the files get bigger, they stay within reasonable bounds until a few large and very large files tip the scale. These large files can be traced to a few `main` classes.

	average	stdev	median	80%	90%	95%	97.5%	max.
JavaCC	5’856	16’630	798	6’055	9’928	28’926	38’797	121’316
classViewer	1’790	1’884	1’147	2’498	4’286	6’285	8’006	8’973
hotjava	3’220	5’229	1’648	3’860	6’796	10’350	17’814	49’384
jdk	5’372	16’404	1’766	4’568	8’243	12’350	50’403	193’716
jgl	3’650	4’177	1’792	5’983	10’232	12’908	15’755	20’772
jws	4’230	11’445	1’921	5’614	8’908	12’946	18’819	365’470
all	4’541	13’017	1’746	4’898	8’302	12’650	23’140	365’470

Table 1 File size [byte]

Figure 1 is a graphical representation of the distribution of the sizes of the files. The cumulative frequencies show what percentage of the whole program size a given fraction of the class files occupies. The two programs plotted enclose the response domain. The progression of the maximal size between the 95 percent, 97.5 percent and 100 percent quantiles in table 1 already hinted at a small number of large files. Their presence is visible in the sharp upturn at around 95 percent. The steepness of the curves shows how much larger these files are: 50 percent of the files hardly make more than 20 percent of the total size, and 50 percent of the total size come from less than 10 percent of the files.

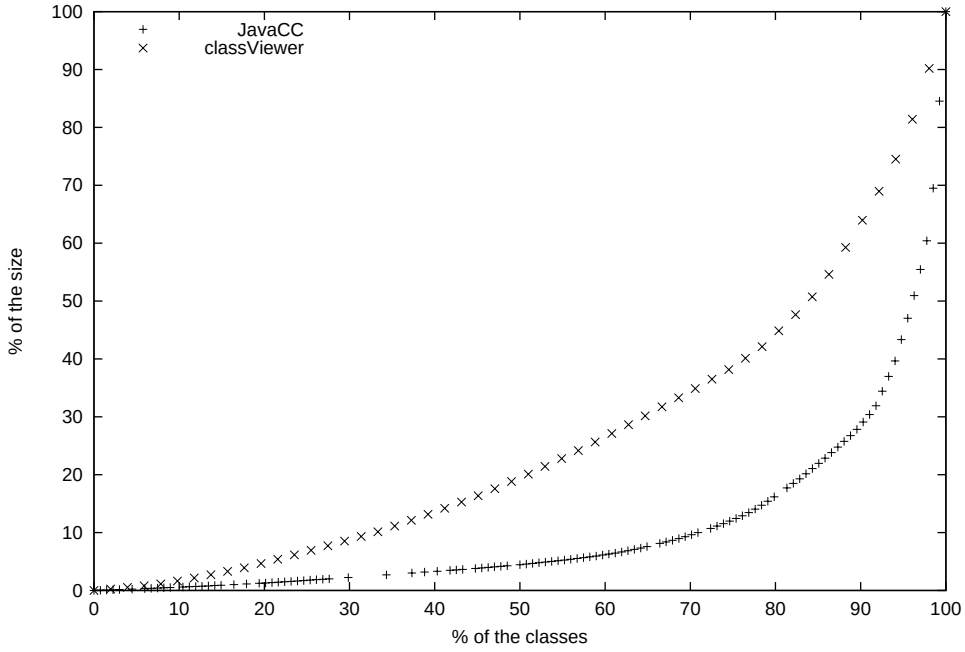

Figure 1 Cumulated frequencies of the file size

Figure 2 plots, for four different programs, the fraction of the classes that are of a given size. It is limited to the 90–95 percent of the files whose sizes are under 10K

bytes. The large value of the standard deviation is apparent in the wide distribution of those files.

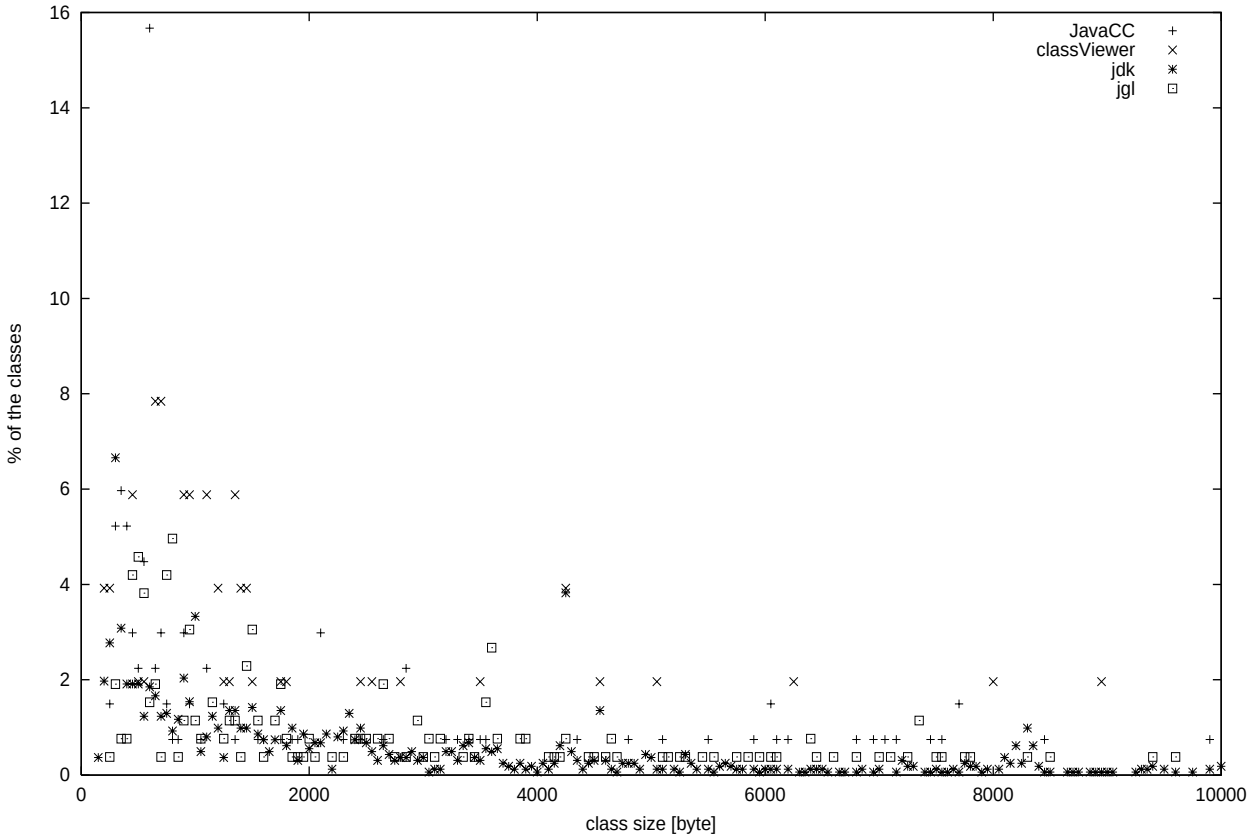

Figure 2 Distribution of the file sizes (cut at 10Kb)

3.0 Parts

The Java class file is built from the aggregation of five parts. [9] sets the succession of the parts and their contents, yet allows flexibility in some parts. We will explore in this section how theses different parts contribute to the overall file size.

3.1 Parts of the class file

The format of the class file is fully described in [9]. Shortly, the class file consists of a:

1. Header part. The header contains a magic number, which identifies Java class files, and the version number of the format of the class file. The header part has a fixed size of 8 bytes.
2. Constant part. The constant part is a variable sized structure that holds all the symbolic informations used in the rest of the file. It is from here that the loader extracts the linking information and the interpreter fetches the constants used in the bytecodes.
3. Class part. The class part contains information defining the class stored in the file. These informations are the access flags, the references to `this`, to `super`, to all the interfaces implemented by the class, the number of fields, the number of methods and an attribute pool. The attribute pool is a variable sized structure that

holds arbitrary additional properties of the class. The size of the constant part also belongs to the class part[1].

4. Field part. The field part is a table of descriptions for all the class and instance variables defined. Each entry in the table contains the access flags, the name and the type of the field and an attribute pool similar to the one in the class part.
5. Method part. The method part is a table of descriptions for the initializers, the constructors and the methods defined. Each description contains the access flags, the name and the type of the procedure and an attribute pool. The bytecodes for the procedure is stored in a `Code` attribute in the attribute pool.

For each class, we measured the sizes of these five parts and computed how many percent of the file they occupy. Table 2 presents the results computed over all the classes and table 15 on page 16 gives the results for the individual programs.

	average	stdev	median	min.	max.
header	0.74	0.81	0.45	0.00	5.00
constant	61.55	16.51	64.87	0.39	99.23
class	2.55	2.74	1.53	0.00	19.39
field	1.65	3.47	0.84	0.00	48.63
method	33.49	18.62	30.89	0.00	99.54

Table 2 Summary of the repartition of the five parts of a class [% of the file size]

The constant part and the method part make up on average 95 percent of the file size. This is not a surprise, but it is quite interesting to consider that the largest element in a class file is actually the constant part and not the method part, which contains the bytecodes to execute.

3.2 Constant part

The constant part is a table of variable length entries, the `CONSTANT` structures. They are:

1. The `CONSTANT_Utf8` structure stores a string of Unicode characters. The encoding used is a slight variation of the standard UTF-8 format.
2. The `CONSTANT_Integer`, `CONSTANT_Long`, `CONSTANT_Float` and `CONSTANT_Double` structures store a value of the corresponding Java primitive type.
3. The `CONSTANT_String` structure represents an object of the type `java.lang.String`. The structure references a `CONSTANT_Utf8` entry that contains the actual characters.
4. The `CONSTANT_Class` structure stands for classes or interfaces. The only member of the structure is the index of a `CONSTANT_Utf8` entry that contains the name of the class.
5. The `CONSTANT_NameAndType` structure brings together the name and the type of a field or a method. They are stored as references to two `CONSTANT_Utf8` entries.

1. The sizes of all the other parts are grouped in the class part to make empty parts more visible.

6. The `CONSTANT_Methodref`, `CONSTANT_InterfaceMethodref` and `CONSTANT_Fieldref` structures build the relation between a method or a field and its class or interface. The three structures have the same members: they all reference both a `CONSTANT_Class`, which names the containing class, and a `CONSTANT_NameAndType`, which identifies the method or the field.

We counted the different kind of structures in the constant part; the `CONSTANT` structures representing the Java primitive types build a single category, *Numeric*, because they appear rather infrequently. Table 3 presents these counts as a percentage of the number of entries.

	average	stdev	median	min.	max.
Utf8	59.06	10.55	56.36	0.57	96.77
Numeric	1.18	5.29	0.00	0.00	99.17
String	3.67	6.55	1.77	0.00	49.91
Class	9.30	4.50	8.45	0.03	38.57
NameAndType	12.83	5.81	14.29	0.00	29.07
FieldRef	4.11	3.86	3.50	0.00	22.95
MethodRef	9.50	5.35	10.26	0.00	28.58
InterfaceMethodRef	0.34	1.27	0.00	0.00	16.09

Table 3 Count of the different `CONSTANT` structures [%]

The importance of the `CONSTANT_Utf8` structures is apparent in the 59 percent of all the entries they represent on the average. It is interesting to consider that the sum of the average number of entries that reference `CONSTANT_Utf8` structures is less than 40 percent[1]! On the opposite, the `CONSTANT_String` and *Numeric* entries make on the average less than 5 percent and the few cases where this proportion is more important are interfaces or classes that only define constants, a common pattern in Java programs. There are no classes without `CONSTANT_Utf8` and `CONSTANT_Class` structures because these structures are used to express the `this` and the `super` references, two required members of any Java classes.

The constant part contains primarily the constants used in the bytecodes and the symbolic information necessary to perform dynamic linking and type checking. But the constant part is accessible from all the other parts of the class file and may hence hold any other constants, as the discrepancy in the number of `CONSTANT_Utf8` items suggested. We divided the entries of the constant part into three groups:

1. Constant. This group contains the `CONSTANT_Integer`, `CONSTANT_Long`, `CONSTANT_Float` and `CONSTANT_Double` as well as the `CONSTANT_String` entries, as they are intended to represent constants used by the bytecodes. The `CONSTANT_Utf8` structures referenced by `CONSTANT_String` entries belong to this group.
2. Type and link. The `CONSTANT_Class`, `CONSTANT_NameAndType`, `CONSTANT_Methodref`, `CONSTANT_InterfaceMethodref` and `CONSTANT_Fieldref` entries constitute the second group. They encode the type system and provide the linkage informations; with their help, the run-time sys-

1. String + Class + 2* NameAndType = 3.67+ 9.30 + 2*12.83 = 38.63

tem performs type checking and dynamic linking. The `CONSTANT_Utf8` entries they reference pertain here.

3. Others. Any parts of the class file may reference the constant part, so there may be `CONSTANT_Utf8`s that belong neither to the constants nor to the typing and linking data; these entries build the third group.

Sun's `javac` minimizes the size of the constant part by writing every constant once, so we had to check that the `CONSTANT_Utf8` structures were only counted once. In case a `CONSTANT_Utf8` structure should turn up in more than one group, it would be added preferentially to the type and link category. Even though a compiler can define an attribute that references any kind of `CONSTANT` structures, we did not ensure that they were actually used as expected. Of all the attributes defined in [9], the `Exceptions` attribute is the only one to reference anything other than a `CONSTANT_Utf8` structure: the `Exceptions` attribute employs `CONSTANT_Class` entries to check the type of thrown exceptions; this usage conforms to our classification. Albeit this repartition appears rather conservative, it gives good practical results.

To determine the importance of these three categories, we grouped the entries according to the rules defined above and summed up the sizes of the structures in each group. These absolute numbers, expressed as a percentage of the total size of the constant part, build the basis for the tables 4, 5 and 6.

Table 4 shows the results for the amount of constant information. The small percentage occupied by the constants confirms the results of table 3; the small sizes of the primitive types (5 to 9 bytes) and the overall small number of constants used explain these values. We linked the high maxima to classes used as enumerations, a design pattern whose usage is not equally represented in the programs studied.

	average	stdev	median	min.	max.
JavaCC	11.48	20.51	0.00	0.00	93.36
classViewer	4.05	9.78	0.65	0.00	59.63
hotjava	3.74	6.20	1.94	0.00	84.47
jdk	10.29	21.32	3.14	0.00	99.69
jgl	6.18	6.95	2.96	0.00	31.63
jws	6.46	9.10	3.93	0.00	99.86
all	7.76	15.56	3.05	0.00	99.86

Table 4 Constant information in the constant part [%]

Table 5 contains the results for the analysis of the typing and linking information. Dynamic linking and run-time resolution of the method calls require an important supply of information from the compiler. Whereas the relations between the types are efficiently encoded in small `CONSTANT` structures, the identification of the type is delivered in a textual encoding as `CONSTANT_Utf8` structures. This sparse encoding appears clearly in the hefty fraction occupied. The classes with low minima

have in common a high number of constants in the bytecodes and the use of primitive types instead of other classes.

	average	stdev	median	min.	max.
JavaCC	54.90	20.43	51.62	3.53	95.97
classViewer	62.78	14.18	60.54	17.74	85.51
hotjava	72.27	16.23	76.99	6.02	92.42
jdk	57.55	22.24	60.41	0.18	96.48
jgl	62.94	13.01	66.25	13.46	87.07
jws	59.33	16.16	63.51	0.07	86.16
all	60.48	19.45	64.80	0.07	96.48

Table 5 Type and link information in the constant part [%]

The results for the last group are in table 6. The entries of this group fill up on the average 31 percent of the constant part and 20 percent of the class file! This is a big fraction, especially when considering that this data is not required to use the class or execute its bytecodes.

	average	stdev	median	min.	max.
JavaCC	33.62	22.96	36.91	0.88	86.56
classViewer	32.87	16.17	35.13	4.05	83.29
hotjava	24.00	17.13	19.34	2.26	86.57
jdk	32.16	20.22	29.94	0.13	97.73
jgl	30.89	14.67	30.11	9.88	85.21
jws	34.22	17.89	29.50	0.07	93.90
all	31.76	19.03	28.19	0.07	97.73

Table 6 Other information in the constant part [%]

3.3 Debugging information

The field, the method and the class parts of the class file have a variable length `Attribute` table, the attribute pool. [9] defines the general structure of an `Attribute`, describes the attributes produced by the version 1.0 of the JDK and states on p. 107 “Of the predefined attributes, the `Code`, `ConstantValue`, and `Exceptions` attributes must be recognized and correctly read by a class file reader for correct interpretation of the class file by a Java Virtual Machine.” [18] proposed to reduce the size of class files by stripping them of those attributes deemed dispensable.

The debugging information is constituted by the entries in the constant part that represent neither constant nor typing information as well as by all the attributes other than `Code`, `ConstantValue` and `Exceptions`. Table 7 contains the average, the

standard deviation (stdev) and the median of the percentage of the class file it occupies. The last two columns give the minimal and the maximal percentages.

	average	stdev	median	min.	max.
JavaCC	32.64	14.38	32.88	10.23	72.21
classViewer	31.60	11.11	31.51	14.57	64.16
hotjava	26.23	11.79	22.66	3.42	72.25
jdk	31.23	15.69	28.95	0.58	82.75
jgl	30.06	10.21	28.91	16.43	66.88
jws	38.50	11.61	37.03	0.08	78.97
all	33.09	14.12	31.43	0.08	82.75

Table 7 Amount of debugging information [%]

This result expands what was found by analysing the constant part: the 33 percent of the file used by debugging information are made up of one third attributes and two thirds strings.

3.4 Code

The bytecodes are the most visible part of the class file, they are its *raison d'être*. We already saw in section 3.1 that the method part amounted to about 30 percent of the size of the file. But the method part itself is more than raw bytecodes: a method structure can include other attributes next to the `Code` attribute. Besides, the `Code` attribute itself brings together the bytecodes with its exceptions table and may in turn include further attributes.

We first measured the percentage of the class file taken by the bytecodes. Table 8 shows that the bytecodes make on the average 12 percent of the class file. The classes `sunw.html.dtds.html32` and `java.lang.Character` are the two extremely high extrema in hotjava and jdk. They both have a very large static initializer for static arrays and they both reference only two or three other classes. This keeps the type and link part of the constant part small.

	average	stdev	median	min.	max.
JavaCC	12.63	12.21	6.14	0.00	48.05
classViewer	8.47	7.84	5.30	0.00	34.85
hotjava	10.72	7.99	9.83	0.00	84.31
jdk	15.29	13.31	12.05	0.00	94.46
jgl	9.27	6.30	8.13	0.00	37.51
jws	10.54	7.50	9.61	0.00	73.31
all	12.44	10.68	10.03	0.00	94.46

Table 8 Amount of bytecodes as a percentage of the whole class [%]

The section 3.3 introduced the notion of debugging information. Table 9 shows that in classes stripped of all this debugging information the bytecodes take on the average only 18 percent of the file size.

	average	stdev	median	min.	max.
JavaCC	17.12	15.53	9.98	0.00	87.92
classViewer	11.39	9.05	8.40	0.00	41.72
hotjava	13.97	10.31	12.71	0.00	95.37
jdk	23.21	25.02	15.26	0.00	99.31
jgl	12.63	8.21	11.54	0.00	44.89
jws	16.10	10.46	14.61	0.00	83.13
all	18.44	18.32	13.91	0.00	99.31

Table 9 Amount of bytecodes as a percentage of the stripped class [%]

4.0 Bytecode

The Java Virtual Machine defines an instruction set rich of 212 different instructions. Two features of this instruction set explain its size. First, many instructions exist in different versions according to the type of the argument(s) they use; this eases the verification of the bytecodes. Second, there are special instructions to efficiently access small constants and frequently used fields.

4.1 Instruction size

A Java Virtual Machine instruction consists of an opcode followed by zero or more bytes that provide the operands. The opcode is encoded in one byte for 200 instructions and two bytes for the 12 remaining instructions. The opcode specifies the operation to be performed and the number of operands; two instructions, `lookupswitch` and `tableswitch`, have a variable number of operands yet they have a minimum size of 9, respectively 13, bytes. Table 10 summarizes the actual sizes of the instructions.

size of the instruction [byte]	number of instructions	% of the instruction set
1	147	69%
2	14	7%
3	33	16%
4	12	6%
5	3	1%
6	1	0%
9 or more	2	1%

Table 10 Repartition of the instruction size

We measured the size of the instructions present in the programs. Table 11 shows that the average size lies just under 2 bytes at 1.92 bytes. The largest means of JavaCC and classViewer are due to the influence of the `switch` statements. In class-

Viewer, for example, the largest instruction is compiled from a single `switch` statement that contains 200 cases.

	average	stdev	median	min.	max
JavaCC	2.41	5.87	2.00	1.00	148.46
classViewer	2.18	11.42	2.00	1.00	822.00
hotjava	1.97	2.05	2.00	1.00	201.00
jdk	1.91	1.89	2.00	1.00	332.00
jgl	1.98	1.17	1.00	1.00	27.33
jws	1.96	1.75	2.00	1.00	443.00
all	1.96	2.39	2.00	1.00	822.00

Table 11 Size of the instructions [byte]

We counted the number of instructions of the different sizes and expressed the counts as a percentage of the instructions in the class file. Figure 3 plots this for the fixed-size instructions. If we compare the result to the classification made in table 10, we see that the 2-bytes and, above all, the 3-bytes instructions are over-represented, as these 7 percent and 16 percent of the instruction set represent 14 percent and 40 percent of the occurrences.

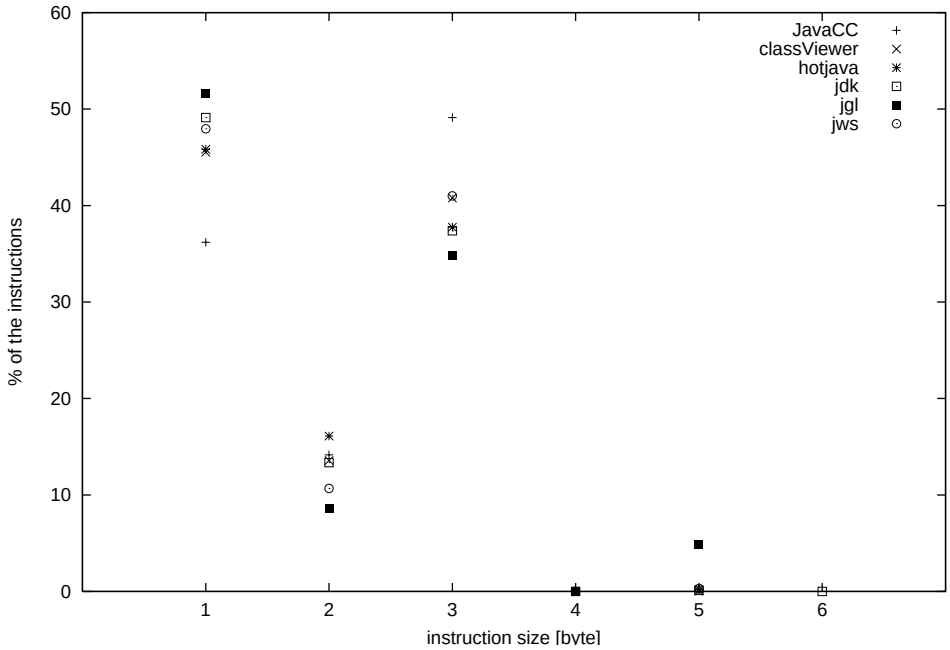

Figure 3 Repartition of the instruction by size

4.2 Usage of the different instructions

To explain the difference between the expected and the measured distributions of the instructions that was just revealed, we studied the usage of the different instruc-

tions. Table 12 shows that the programs considered use on the average 25 of the 212 instructions or 12 percent of the instruction set.

	average	stdev	median	min.	max
JavaCC	22.79	22.30	11.00	0.00	100.00
classViewer	18.06	15.22	11.00	0.00	55.00
hotjava	25.68	18.24	22.00	0.00	98.00
jdk	24.95	19.94	20.00	0.00	113.00
jgl	22.77	15.46	21.00	0.00	73.00
jws	26.86	18.64	23.00	0.00	92.00
all	25.42	19.09	21.00	0.00	113.00

Table 12 Number of different instructions

Figure 4 shows the frequency of occurrences for the different instructions. The instructions with a frequency less than 0.1 percent were omitted to enhance the readability.

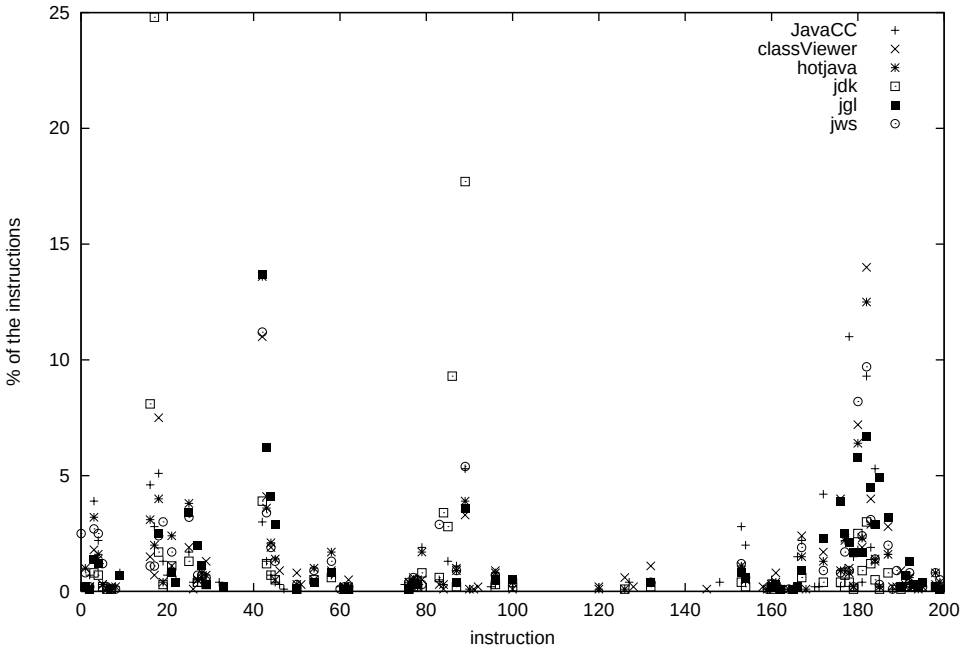

Figure 4 Frequency of occurrences of the instructions [≥ 0.1%]

The figure shows the wide difference in usage count between the instructions but also between the programs considered. Even though five clusters are visible, only a few instructions are clearly preferred by a majority of the programs. Five instructions individually account for more than 5 percent of the occurrences in at least two programs; they are `ldc` (0x12), `aload_0` (0x2a), `dup` (0x59), `getfield` (0xb4) and `invokevirtual` (0xb6). Finally, the good performance of the `nop` (0x00) instruction in jws is astounding.

4.3 Entropy & Redundancy

These variations in the usage frequencies of the instructions lead us to wonder if using a fixed size opcode was a good idea. The field of information theory provides

a measure for the actual information content of a message, the entropy *H*. To apply this metric to this case, we have to consider the bytecodes as a message whose symbols are the instructions of the Java Virtual Machine.

For a message in which the probability of occurrence of the *i*th symbol is P_i, the entropy *H* is defined as:

$$H = -\sum P_i \times \log P_i$$

Since we are representing information in bits, the logarithm is to the base two; *H* becomes a measure of the average number of bits of information in each bytecode instruction. Table 13 shows the entropy for the classes in the data set: with an optimal encoding, the average opcode size would be 4 bits instead of the 8 or 16 bits used now.

	average	stdev	median	min.	max.
JavaCC	3.31	1.01	3.18	0.00	5.30
classViewer	3.25	0.97	3.30	0.00	4.84
hotjava	3.64	1.11	3.93	0.00	5.48
jdk	3.22	1.47	3.48	0.00	5.81
jgl	3.64	1.09	3.95	0.00	5.12
jws	3.66	1.24	3.99	0.00	5.32
all	3.46	1.32	3.85	0.00	5.81

Table 13 Entropy *H*

An other measure of the quality of the encoding is the redundancy of the message, which ranges from 0 (no redundancy) to 1 (infinite redundancy). With $\bar{\sigma}$ the actual average symbol size, the redundancy is defined as:

$$Redundancy = 1 - \frac{H}{\bar{\sigma}}$$

The results in table 14 are remarkably homogeneous; the average redundancy is in all the cases between 54 percent and 59 percent.

	average	stdev	median	min.	max.
JavaCC	0.58	0.12	0.60	0.33	1.00
classViewer	0.59	0.12	0.58	0.39	1.00
hotjava	0.54	0.13	0.50	0.31	1.00
jdk	0.59	0.18	0.56	0.27	1.00
jgl	0.54	0.13	0.50	0.36	1.00
jws	0.54	0.15	0.50	0.33	1.00
all	0.56	0.16	0.51	0.27	1.00

Table 14 Redundancy

These two results together suggest that there is a more space efficient representation of the opcodes.

5.0 Conclusions

This work aimed at answering three basic questions: What can be said about the size of the class files? How do the different parts of the class file contribute to its total size? How are the bytecode instructions used?

We analysed six programs totalling 4016 unique classes. These programs belong to what we can call the first generation of Java applications: they are monolithic softwares intended to be installed on a machine and used like programs written in conventional languages, e.g. C, C++. A second generation of applications is announced [13] that will be build around smaller and versatile components, the JavaBeans [8]. The influence of this new model on the distribution of the file sizes has not yet been determined. The class files analysed were probably compiled with Sun's compiler and the impact of other compilers on the composition of the class files is a question left open for further investigations.

Our analyses bring forth the following facts:

1. Java class files are small in the average. We found that 50 percent take less than 2'000 bytes, 80 percent less than 6'000 and 95 percent less than 13'000 bytes. However we found that complete programs also contain a few files as large as 365'470 bytes. The size of the Java class file is important because the time to read the file is the dominant factor in the start-up time.
2. The biggest part of the Java class files is the constant part (61 percent of the file) and not the method part that accounts for only 33 percent of the file size. The other parts of the class file share the remaining 5 percent.
3. About 32 percent of the size of the file is constituted by unessential or debugging information, such as the name of the source file or tables associating offsets into the bytecodes to line numbers in the Java source code. The file can safely be reduced to 70 percent of its size and stay perfectly functional.
4. On the average, the bytecodes take only 12 percent of the class file and only 18 percent of the class file stripped of its superfluous content.
5. The average size of an instruction is slightly less than 2 bytes.
6. The programs typically use 25 different instructions and at most 113 instructions, when 212 are defined.
7. The frequencies of the instructions used vary considerably. There are five instructions that individually account in at least two programs for more than 5 percent of the occurrences.
8. The theoretical minimum average number of bits needed to encode the opcode is 4 bits instead of the 8 or 16 used today.

6.0 References

[1] *HP Offers Virtual-machine Technology to Embedded-device Market*, Palo Alto, California. March 20, 1998, <http://www.hp.com/pressrel/mar98/20mar98b.htm>

[2] *Japhar*, <http://www.hungry.com/products/japhar/>

[3] *ObjectSpace JGL: The Generic Collection Library for Java*, <http://www.objectpace.com/jgl>

[4] Barry D. Bowen, *Developer Environments*, JavaWorld, May 1996, <http://www.javaworld.com/javaworld/jw-05-1996/jw-05-devenvirons.html>

[5] Jens Ernst, William Evans, Christopher W. Fraser, Steven Lucco and Todd A. Proebsting, *Code Compression* in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'97)*, ACM SIGPLAN Notices, v32(5), p. 358-365, June 1997

[6] Michael Franz and Thomas Kistler, *Slim Binaries*, Technical Report, Department of Computer Science, University of California at Irvine, 1996

[7] James Gosling, Bill Joy and Guy Steele, *The Java Language Specification*, Addison Wesley, 1996

[8] Tom R. Halfhill, *JavaBeans: Cross-Platform Components*, Byte, v22(1), p. 74, Jan. 1997

[9] Tim Lindholm and Frank Yellin, *The Java Virtual Machine Specification*, Addison Wesley, 1996

[10] David S. Linthicum, *Java Evolves*, Byte, v23(1), p. 60, Jan. 1998

[11] Glenford J. Myers, *Advances in Computer Architectur*e, John Wiley & Sons, 1978

[12] Martin Odersky and Philip Wadler, *Pizza into Java: Translating theory into practice* in *Conference Record of POPL'97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, p. 146-159, Jan. 1997

[13] Dick Pountain, *The Component Enterprise*, Byte, v22(5), p. 93-98, May 1997

[14] Sriram Sankar, Sreenivasa Viswanadha and Rob Duncan, *JavaCC: The Java Compiler Compiler*, <http://www.suntest.com/JavaCC/>

[15] K. A. Shillington and G. M. Ackland, *UCSD Pascal Version 1.5*, Institute for Information Systems, University of California, San Diego, 1978

[16] A. A. Stepanov and M. Lee, *The Standard Template Library*, Technical Report HPL-94-34, revised July 7, 1995

[17] Tim J. Wilkinson, *KAFFE - A virtual machine to run Java code*, <http://www.kaffe.org/>

[18] Matt T. Yourst, *Inside Java Class Files*, Dr. Dobb's Journal, n. 281, p. 46-52, Jan. 1998

7.0 Annexe: Results for the individual programs

7.1 The five parts of the class file format

		average	stdev	median	min.	max.
JavaCC	header	0.98	0.77	1.00	0.01	2.72
	constant	62.92	13.84	63.22	5.98	84.31
	class	2.96	2.31	3.01	0.02	8.16
	field	2.29	4.41	0.49	0.00	25.74
	method	30.85	16.37	28.85	5.23	93.77
classViewer	header	0.91	0.78	0.70	0.08	3.54
	constant	67.67	8.05	68.01	43.28	82.39
	class	3.44	2.45	3.24	0.24	10.62
	field	1.79	1.59	1.31	0.00	8.65
	method	26.20	9.75	26.51	3.54	47.00
hotjava	header	0.64	0.59	0.49	0.02	3.39
	constant	67.96	11.62	69.55	2.19	90.72
	class	2.33	2.10	1.54	0.06	10.17
	field	1.67	3.40	1.06	0.00	47.90
	method	27.39	13.21	25.83	0.00	97.65
jdk	header	0.84	0.91	0.45	0.00	4.94
	constant	58.79	21.55	64.77	0.39	99.23
	class	2.59	2.78	1.41	0.01	14.82
	field	1.55	3.54	0.45	0.00	48.63
	method	36.23	23.76	31.06	0.00	99.55
jgl	header	0.65	0.60	0.45	0.04	3.16
	constant	65.55	11.60	69.16	37.81	85.37
	class	2.37	2.36	1.45	0.13	12.04
	field	1.85	1.53	1.34	0.00	7.66
	method	29.58	14.12	26.11	2.27	60.78
jws	header	0.67	0.80	0.42	0.00	5.00
	constant	61.22	10.81	62.57	16.42	89.58
	class	2.55	3.02	1.56	0.01	19.39
	field	1.66	3.62	0.89	0.00	38.03
	method	33.90	13.35	33.37	0.00	81.44

Table 15 Repartition of the parts [%]

7.2 The constants

		average	stdev	median	min.	max
JavaCC	Utf8	59.40	11.25	59.46	33.69	91.67
	Numeric	1.10	4.46	0.00	0.00	24.56
	String	5.34	8.92	0.00	0.00	43.83
	Class	8.63	3.66	8.70	0.58	21.67
	NameAndType	11.28	5.15	10.16	0.00	29.07
	FieldRef	3.55	3.70	2.53	0.00	15.04
	MethodRef	10.60	5.40	10.00	0.00	28.58
	InterfaceMethodRef	0.11	0.52	0.00	0.00	4.00
classViewer	Utf8	61.84	9.29	64.15	45.71	84.62
	Numeric	0.80	1.69	0.00	0.00	9.52
	String	2.43	6.06	0.00	0.00	38.70
	Class	11.02	3.93	11.32	2.30	23.53
	NameAndType	11.52	4.41	11.77	0.00	18.82
	FieldRef	3.93	3.66	2.84	0.00	14.46
	MethodRef	8.43	5.14	7.14	0.00	20.97
	InterfaceMethodRef	0.02	0.15	0.00	0.00	1.08
hotjava	Utf8	56.34	9.44	53.54	42.41	92.31
	Numeric	0.56	2.72	0.00	0.00	31.88
	String	2.56	3.82	1.39	0.00	47.06
	Class	11.35	6.09	9.61	0.31	38.57
	NameAndType	14.18	5.60	15.82	0.00	24.59
	FieldRef	3.97	3.05	3.23	0.00	22.95
	MethodRef	10.90	5.36	11.39	0.00	24.78
	InterfaceMethodRef	0.15	0.53	0.00	0.00	5.71
jdk	Utf8	58.93	11.41	55.56	0.57	96.77
	Numeric	1.78	7.18	0.00	0.00	99.17
	String	4.21	8.27	1.75	0.00	47.81
	Class	9.38	4.22	8.64	0.09	31.03
	NameAndType	12.23	6.22	14.07	0.00	24.64
	FieldRef	4.64	4.74	3.70	0.00	18.87
	MethodRef	8.61	5.74	9.06	0.00	22.45
	InterfaceMethodRef	0.22	0.91	0.00	0.00	15.85
jgl	Utf8	57.77	9.83	55.88	41.87	90.24
	Numeric	1.16	1.42	0.74	0.00	5.56
	String	2.32	2.79	1.22	0.00	9.77

Table 16 Repartition of the constants [%]

		average	stdev	median	min.	max
	Class	9.90	4.84	9.59	1.79	25.58
	NameAndType	14.08	5.84	15.07	0.00	23.98
	FieldRef	3.11	2.90	2.67	0.00	9.77
	MethodRef	9.25	4.73	10.37	0.00	24.19
	InterfaceMethodRef	2.40	3.33	1.27	0.00	16.09
jws	Utf8	60.36	9.76	58.01	40.99	95.75
	Numeric	0.75	3.81	0.00	0.00	45.10
	String	3.61	5.12	2.29	0.00	49.91
	Class	8.31	3.74	7.69	0.03	33.33
	NameAndType	12.97	5.33	14.15	0.00	22.53
	FieldRef	3.80	3.00	3.49	0.00	17.95
	MethodRef	9.99	4.77	10.84	0.00	21.58
	InterfaceMethodRef	0.22	0.79	0.00	0.00	15.39

Table 16 Repartition of the constants [%]