Byte Code Engineering with the JavaClass API

Technical Report B-17-98

Markus Dahm

Freie Universität Berlin
Institut für Informatik
dahm@inf.fu-berlin.de

July 7, 1999

Abstract

Extensions and improvements of the programming language Java and its related execution environment (Java Virtual Machine, JVM) are the subject of a large number of research projects and proposals. There are projects, for instance, to add parameterized types to Java, to implement "Aspect-Oriented Programming", and to improve the run-time performance.

Since Java classes are compiled into portable binary class files (called *byte code*), it is the most convenient and platform-independent way to implement these improvements not by writing a new compiler or changing the JVM, but by transforming the byte code. These transformations can either be performed after compile-time, or at load-time. Many programmers are doing this by implementing their own specialized byte code manipulation tools, which are, however, restricted in the range of their re-usability.

To deal with the necessary class file transformations, we introduce an API that helps developers to conveniently implement their transformations.

1 Introduction

The Java language [GJS96] has become very popular and many research projects deal with further improvements of the language or its run-time behavior. The possibility to extend a language with new concepts is surely a desirable feature, but implementation issues should be hidden from the user. Fortunately, the concepts of the Java Virtual Machine permit the user-transparent implementation of such extensions with relatively little effort.

Because the target language of Java is an interpreted language with a small and easy-to-understand set of instructions (the *byte code*), developers can implement and test their concepts

in a very elegant way. One can write a plug-in replacement for the system's class loader which is responsible for dynamically loading class files at run-time and passing the byte code to the Virtual Machine (see section 4.1). Class loaders may thus be used to intercept the loading process and transform classes before they get actually executed by the JVM [LB98]. While the original class files always remain unaltered, the behavior of the class loader may be reconfigured for every execution or instrumented dynamically.

The JAVACLASS API is a toolkit for the static analysis and dynamic creation or transformation of Java class files. It enables developers to implement the desired features on a high level of abstraction without handling all the internal details of the Java class file format and thus re-inventing the wheel every time. JAVACLASS is written entirely in Java and freely available under the terms of GNU Public License (GPL). [1]

This report is structured as follows: We give a brief description of the Java Virtual Machine and the class file format in section 2. Section 3 introduces the JAVACLASS API. Section 4 describes some typical application areas and example projects. The appendix contains code examples that are to long to be presented in the main part of this report. All examples are included in the down-loadable distribution.

1.1 Related work

There are a number of proposals and class libraries that have some similarities with JAVACLASS: The JOIE [CCK98] toolkit can be used to instrument class loaders with dynamic behavior. Similarly, "Binary Component Adaptation" [KH98] allows components to be adapted and evolved on-the-fly. Han Lee's "Byte-code Instrumenting Tool" [LZ98] allows the user to insert calls to analysis methods anywhere in the byte code. The Jasmin language [MD97] can be used to hand-write or generate pseudo-assembler code. D-Java [Sil98] and JCF [You98] are class viewing tools.

In contrast to these projects, JAVACLASS is intended to be a general purpose tool for "byte code engineering". It gives full control to the developer on a high level of abstraction and is not restricted to any particular application area.

2 The Java Virtual Machine

Readers already familiar with the Java Virtual Machine and the Java class file format may want to skip this section and proceed with section 3.

Programs written in the Java language are compiled into a portable binary format called *byte code*. Every class is represented by a single class file containing class related data and byte code instructions. These files are loaded dynamically into an interpreter (Java Virtual Machine, JVM) and executed.

Figure 1 illustrates the procedure of compiling and executing a Java class: The source file (`HelloWorld.java`) is compiled into a Java class file (`HelloWorld.class`), loaded by

[1] The distribution is available at `http://www.inf.fu-berlin.de/~dahm/JavaClass/index.html`, including several code examples and javadoc manuals.

the byte code interpreter and executed. In order to implement additional features, researchers may want to transform class files (drawn with bold lines) before they get actually executed. This application area is one of the main issues of this article.

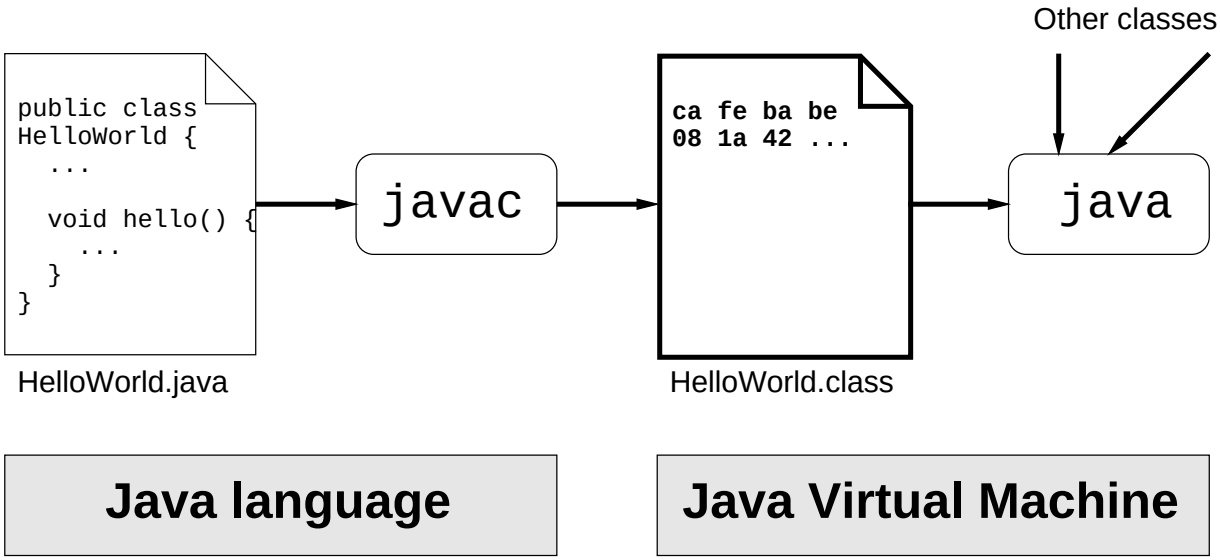

Figure 1: Compilation and execution of Java classes

Note that the use of the general term "Java" implies two meanings: on the one hand, Java as a programming language is meant, on the other hand, the Java Virtual Machine, which is not necessarily targeted by the Java language exclusively, but may be used by other languages as well (e.g. Eiffel [CCZ97], or Ada [Taf96]). We assume the reader to be familiar with the Java language and to have a general understanding of the Virtual Machine.

2.1 Java class file format

Giving a full overview of the design issues of the Java class file format and the associated byte code instructions is beyond the scope of this report. We will just give a brief introduction covering the details that are necessary for understanding the rest of this paper. The format of class files and the byte code instruction set are described in more detail in the "Java Virtual Machine Specification" [LY97] [2], and in [MD97]. Especially, we will not deal with the security constraints that the Java Virtual Machine has to check at run-time, i.e. the byte code verifier.

Figure 2 shows a simplified example of the contents of a Java class file: It starts with a header containing a "magic number" (`0xCAFEBABE`) and the version number, followed by the *constant pool*, which can be roughly thought of as the text segment of an executable, the *access rights* of the class encoded by a bit mask, a list of interfaces implemented by the class, lists containing the fields and methods of the class, and finally the *class attributes*, e.g. the `SourceFile` attribute

[2] Also available online at `http://www.javasoft.com/docs/books/vmspec/index.html`

telling the name of the source file. Attributes are a way of putting additional, e.g. user-defined, information into class file data structures. For example, a custom class loader may evaluate such attribute data in order to perform its transformations. The JVM specification declares that unknown, i.e. user-defined attributes must be ignored by any Virtual Machine implementation.

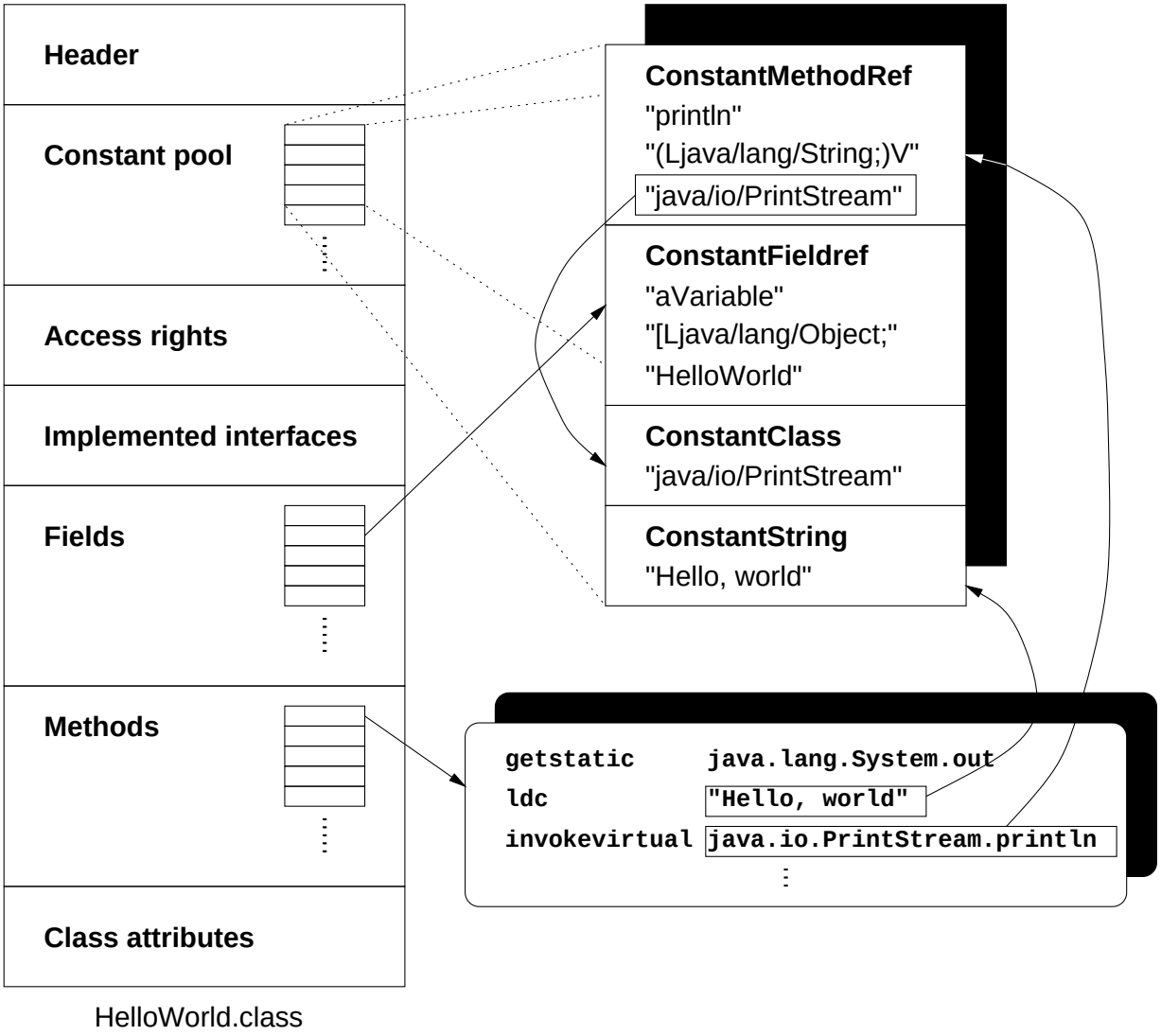

Figure 2: Java class file format

Because all of the information needed to dynamically resolve the symbolic references to classes, fields and methods at run-time is coded with string constants, the constant pool contains in fact the largest portion of an average class file, approximately 60% [AP98]. The byte code instructions themselves just make up 12%.

The right upper box shows a “zoomed” excerpt of the constant pool, while the rounded box below depicts some instructions that are contained within a method of the example class. These

instructions represent the straightforward translation of the well-known statement:

```
System.out.println("Hello, world");
```

The first instruction loads the contents of the field `out` of class `java.lang.System` onto the operand stack. This is an instance of the class `java.io.PrintStream`. The `ldc` ("Load constant") pushes a reference to the string "Hello world" on the stack. The next instruction invokes the instance method `println` which takes both values as parameters (Instance methods always implicitly take an instance reference as their first argument).

Instructions, other data structures within the class file and constants themselves may refer to constants in the constant pool. Such references are implemented via fixed indexes encoded directly into the instructions. This is illustrated for some items of the figure emphasized with a surrounding box.

For example, the `invokevirtual` instruction refers to a `MethodRef` constant that contains information about the name of the called method, the signature (i.e. the encoded argument and return types), and to which class the method belongs. In fact, as emphasized by the boxed value, the `MethodRef` constant itself just refers to other entries holding the real data, e.g. it refers to a `ConstantClass` entry containing a symbolic reference to the class `java.io.PrintStream`. To keep the class file compact, such constants are typically shared by different instructions. Similarly, a field is represented by a `Fieldref` constant that includes information about the name, the type and the containing class of the field.

The constant pool basically holds the following types of constants: References to methods, fields and classes, strings, integers, floats, longs, and doubles.

2.2 Byte code instruction set

The JVM is a stack-oriented interpreter that creates a local stack frame of fixed size for every method invocation. The size of the local stack has to be computed by the compiler. Values may also be stored intermediately in a frame area containing *local variables* which can be used like a set of registers. These local variables are numbered from 0 to 65535, i.e. you have a maximum of 65536 of local variables. The stack frames of caller and callee method are overlapping, i.e. the caller pushes arguments onto the operand stack and the called method receives them in local variables.

The byte code instruction set currently consists of 212 instructions, 44 opcodes are marked as reserved and may be used for future extensions or intermediate optimizations within the Virtual Machine. The instruction set can be roughly grouped as follows:

Stack operations: Constants can be pushed onto the stack either by loading them from the constant pool with the `ldc` instruction or with special "short-cut" instructions where the operand is encoded into the instructions, e.g. `iconst_0` or `bipush` (push byte value).

Arithmetic operations: The instruction set of the Java Virtual Machine distinguishes its operand types using different instructions to operate on values of specific type. Arithmetic operations starting with `i`, for example, denote an integer operation. E.g. `iadd` that adds two

integers and pushes the result back on the stack. The Java types `boolean`, `byte`, `short`, and `char` are handled as integers by the JVM.

Control flow: There are branch instructions like `goto` and `if_icmpeq`, which compares two integers for equality. There is also a `jsr` (jump sub-routine) and `ret` pair of instructions that is used to implement the `finally` clause of `try-catch` blocks. Exceptions may be thrown with the `athrow` instruction.

Branch targets are coded as offsets from the current byte code position, i.e. with an integer number.

Load and store operations for local variables like `iload` and `istore`. There are also array operations like `iastore` which stores an integer value into an array.

Field access: The value of an instance field may be retrieved with `getfield` and written with `putfield`. For static fields, there are `getstatic` and `putstatic` counterparts.

Method invocation: Methods may either be called via static references with `invokestatic` or be bound virtually with the `invokevirtual` instruction. Super class methods and private methods are invoked with `invokespecial`.

Object allocation: Class instances are allocated with the `new` instruction, arrays of basic type like `int[]` with `newarray`, arrays of references like `String[][]` with `anewarray` or `multianewarray`.

Conversion and type checking: For stack operands of basic type there exist casting operations like `f2i` which converts a float value into an integer. The validity of a type cast may be checked with `checkcast` and the `instanceof` operator can be directly mapped to the equally named instruction.

Most instructions have a fixed length, but there are also some variable-length instructions: In particular, the `lookupswitch` and `tableswitch` instructions, which are used to implement `switch()` statements. Since the number of `case` clauses may vary, these instructions contain a variable number of statements.

We will not list all byte code instructions here, since these are explained in detail in the JVM specification. The opcode names are mostly self-explaining, so understanding the following code examples should be fairly intuitive.

2.3 Method code

Non-abstract methods contain an attribute (`Code`) that holds the following data: The maximum size of the method's stack frame, the number of local variables and an array of byte code instructions. Optionally, it may also contain information about the names of local variables and source file line numbers that can be used by a debugger.

Whenever an exception is thrown, the JVM performs exception handling by looking into a table of exception handlers. The table marks handlers, i.e. pieces of code, to be responsible for

exceptions of certain types that are raised within a given area of the byte code. When there is no appropriate handler the exception is propagated back to the caller of the method.

2.4 Byte code offsets

Targets of branch instructions like `goto` are encoded as relative offsets in the array of byte codes. Exception handlers and local variables refer to absolute addresses within the byte code. The former contains references to the start and the and of the `try` block, and to the instruction handler code. The latter marks the range in which a local variable is valid, i.e. its scope. This makes it difficult to insert or delete code areas on this level of abstraction, since one has to recompute the offsets every time and update the referring objects. We will see in section 3.3 how JAVACLASS remedies this restriction.

2.5 Type information

Java is a type-safe language and the information about the types of fields, local variables, and methods is stored in *signatures*. These are strings stored in the constant pool and encoded in a special format. For example the argument and return types of the `main` method

```
public static void main(String[] argv)
```

are represented by the signature

```
([java/lang/String;)V
```

Classes and arrays are internally represented by strings like `"java/lang/String"`, basic types like `float` by an integer number.

2.6 Code example

The following example program prompts for a number and prints the faculty of it. The `readLine()` method reading from the standard input may raise an `IOException` and if a misspelled number is passed to `parseInt()` it throws a `NumberFormatException`. Thus, the critical area of code must be encapsulated in a `try-catch` block.

```
import java.io.*;
public class Faculty {
  private static BufferedReader in = new BufferedReader(new
                                     InputStreamReader(System.in));
  public static final int fac(int n) {
    return (n == 0)? 1 : n * fac(n - 1);
  }
  public static final int readInt() {
    int n = 4711;
```

```
    try {
      System.out.print("Please enter a number> ");
      n = Integer.parseInt(in.readLine());
    } catch(IOException e1) { System.err.println(e1); }
      catch(NumberFormatException e2) { System.err.println(e2); }
    return n;
  }
  public static void main(String[] argv) {
    int n = readInt();
    System.out.println("Faculty of " + n + " is " + fac(n));
  }}
```

This code example typically compiles to the following chunks of byte code:

2.6.1 Method fac

```
0:  iload_0
1:  ifne            #8
4:  iconst_1
5:  goto            #16
8:  iload_0
9:  iload_0
10: iconst_1
11: isub
12: invokestatic    Faculty.fac (I)I (12)
15: imul
16: ireturn

LocalVariable(start_pc = 0, length = 16, index = 0:int n)
```

The method `fac` has only one local variable, the argument `n`, stored in slot 0. This variable's scope ranges from the start of the byte code sequence to the very end. If the value of `n` (stored in local variable 0, i.e. the value fetched with `iload_0`) is not equal to 0, the `ifne` instruction branches to the byte code at offset 8, otherwise a 1 is pushed onto the operand stack and the control flow branches to the final return. For ease of reading, the offsets of the branch instructions, which are actually relative, are displayed as absolute addresses in these examples.

If recursion has to continue, the arguments for the multiplication (`n` and `fac(n - 1)`) are evaluated and the results pushed onto the operand stack. After the multiplication operation has been performed the function returns the computed value from the top of the stack.

2.6.2 Method readInt

```
0:  sipush        4711
3:  istore_0
4:  getstatic     java.lang.System.out Ljava/io/PrintStream;
7:  ldc           "Please enter a number> "
```

```
9:  invokevirtual java.io.PrintStream.print (Ljava/lang/String;)V
12: getstatic     Faculty.in Ljava/io/BufferedReader;
15: invokevirtual java.io.BufferedReader.readLine ()Ljava/lang/String;
18: invokestatic  java.lang.Integer.parseInt (Ljava/lang/String;)I
21: istore_0
22: goto          #44
25: astore_1
26: getstatic     java.lang.System.err Ljava/io/PrintStream;
29: aload_1
30: invokevirtual java.io.PrintStream.println (Ljava/lang/Object;)V
33: goto          #44
36: astore_1
37: getstatic     java.lang.System.err Ljava/io/PrintStream;
40: aload_1
41: invokevirtual java.io.PrintStream.println (Ljava/lang/Object;)V
44: iload_0
45: ireturn

Exception handler(s) =
From    To      Handler Type
4       22      25      java.io.IOException(6)
4       22      36      NumberFormatException(10)
```

First the local variable `n` (in slot 0) is initialized to the value 4711. The next instruction, `getstatic`, loads the static `System.out` field onto the stack. Then a string is loaded and printed, a number read from the standard input and assigned to `n`.

If one of the called methods (`readLine()` and `parseInt()`) throws an exception, the Java Virtual Machine calls one of the declared exception handlers, depending on the type of the exception. The `try`-clause itself does not produce any code, it merely defines the range in which the following handlers are active. In the example the specified source code area maps to a byte code area ranging from offset 4 (inclusive) to 22 (exclusive). If no exception has occurred ("normal" execution flow) the `goto` instructions branch behind the handler code. There the value of `n` is loaded and returned.

For example the handler for `java.io.IOException` starts at offset 25. It simply prints the error and branches back to the normal execution flow, i.e. as if no exception had occurred.

3 The JavaClass API

The JAVACLASS API abstracts from the concrete circumstances of the Java Virtual Machine and how to read and write binary Java class files. The API mainly consists of three parts:

1. A component that gives a "static" view upon class files, i.e. it is not intended for byte code modifications. It may be used to read and write class files from or to a file. This is useful especially for analyzing Java classes without having the source files at hand. The main data structure is called `JavaClass` which gives the whole API its name.

2. A package to dynamically generate or modify `JavaClass` objects. It may be used e.g. to insert analysis code, to strip unnecessary information from class files, or to implement the code generator back-end of a Java compiler.

3. Various code examples and utilities like a class file viewer, a tool to convert class files into HTML, and a converter from class files to the Jasmin assembly language [MD97].

Classes of the generic API may be converted into their static counterparts and vice versa.

3.1 JavaClass

The "static" component of the JAVACLASS API resides in the package `de.fub.bytecode.classfile` and represents class files. All of the binary components and data structures declared in the JVM specification [LY97] and described in section 2 are mapped to classes. Figure 3 shows an UML diagram of the hierarchy of classes of the JAVACLASS API. Figure 8 in the appendix also shows a detailed diagram of the `ConstantPool` components.

The top-level data structure is `JavaClass`, which in most cases is created by a `ClassParser` object that is capable of parsing binary class files. A `JavaClass` object basically consists of fields, methods, symbolic references to the super class and to the implemented interfaces of the represented class.

The constant pool serves as some kind of central repository and is thus of outstanding importance for all components. `ConstantPool` objects contain an array of fixed size of `Constant` entries, which may be retrieved via the `getConstant()` method taking an integer index as argument. Indexes to the constant pool may be contained in instructions as well as in other components of a class file and in constant pool entries themselves.

Methods and fields contain a signature, symbolically defining their types. Access flags like `public static final` occur in several places and are encoded by an integer bit mask, e.g. `public static final` matches to the Java expression

```
int access_flags = ACC_PUBLIC | ACC_STATIC | ACC_FINAL;
```

As mentioned in section 2.1 already, several components may contain *attribute* objects: classes, fields, methods, and `Code` objects (introduced in section 2.3). The latter is an attribute itself that contains the actual byte code array, the maximum stack size, the number of local variables, a table of handled exceptions, and some optional debugging information coded as `LineNumberTable` and `LocalVariableTable` attributes. Attributes are specific to some data structure, i.e. no two components share the same kind of attribute. In the figure the `Attribute` classes are marked with the component they belong to.

3.2 Class repository

Using the provided `Repository` class, reading class files into a `JavaClass` object is quite simple:

```
JavaClass clazz = Repository.lookupClass("java.lang.String");
```

The repository also contains methods providing the dynamic equivalent of the `instanceof` operator, and other useful routines:

```
if(Repository.instanceOf(clazz, super_class) {
  ...
}
```

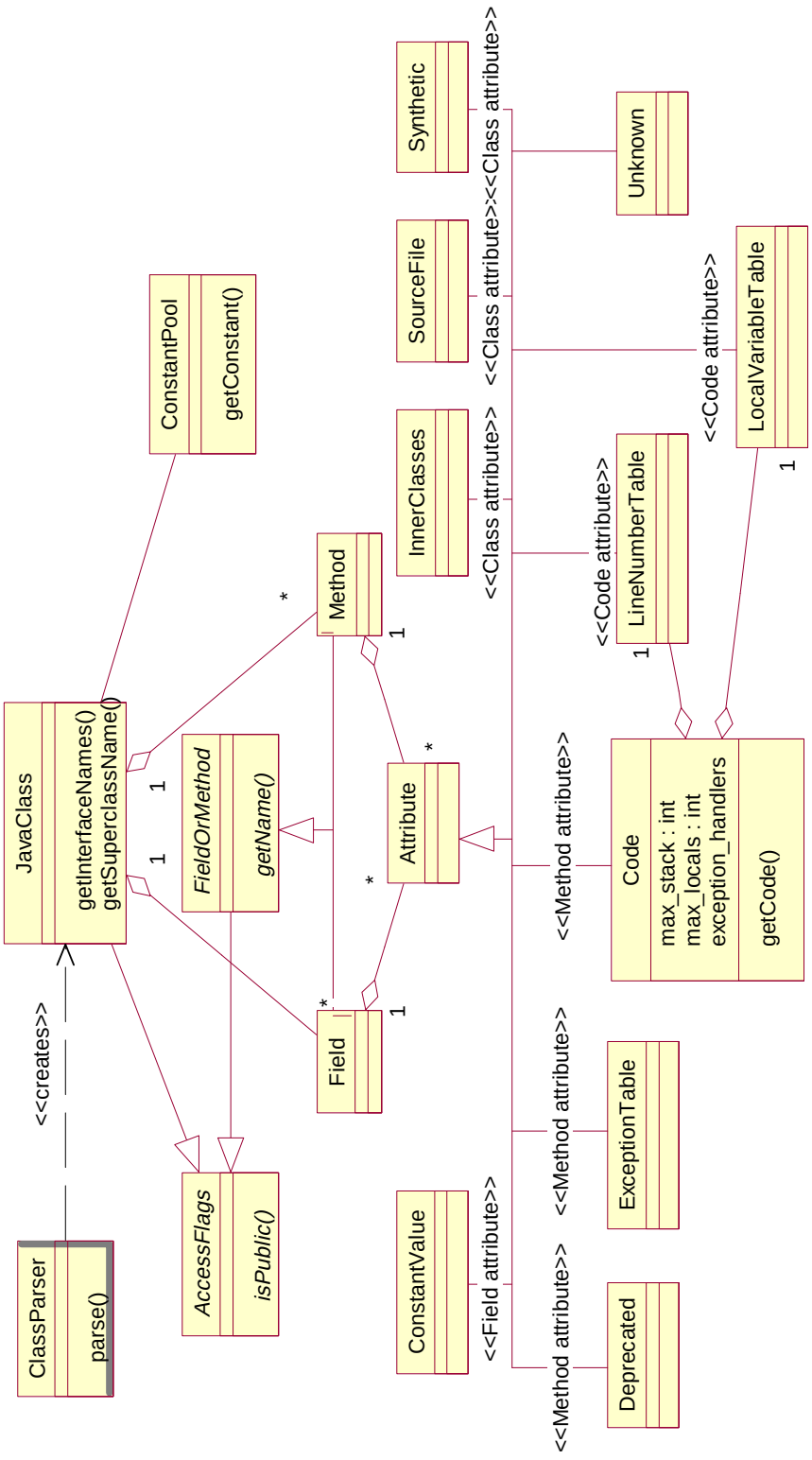

Figure 3: UML diagram for the JAVACLASS API

3.2.1 Accessing class file data

Information within the class file components may be accessed via an intuitive set/get interface. All of them also define a `toString()` method so that implementing a simple class viewer is very easy. In fact all of the examples used here have been produced this way:

```
System.out.println(clazz);
printCode(clazz.getMethods());
...
public static void printCode(Method[] methods) {
  for(int i=0; i < methods.length; i++) {
    System.out.println(methods[i]);

    Code code = methods[i].getCode();
    if(code != null) // Non-abstract method
      System.out.println(code);
  }
}
```

3.2.2 Analyzing class data

Last but not least, JAVACLASS supports the *Visitor* design pattern [GHJV95], so one can write visitor objects to traverse and analyze the contents of a class file. Included in the distribution is a class `Jasmin-Visitor` that converts class files into the Jasmin assembler language [MD97].

3.3 ClassGen

This part of the API supplies an abstraction level for creating or transforming class files dynamically. It makes the static constraints of Java class files like the hard-coded byte code addresses generic. The generic constant pool, for example, is implemented by the class `ConstantPoolGen` which offers methods for adding different types of constants. Accordingly, `ClassGen` offers an interface to add methods, fields, and attributes. Figure 4 gives an overview of the CLASSGEN API.

3.3.1 Types

We abstract from the concrete details of the type signature syntax (see 2.5) by introducing the `Type` class, which is used, for example, by methods to define their return and argument types. Concrete sub-classes are `BasicType`, `ObjectType`, and `ArrayType` which consists of the element type and the number of dimensions. For basic types the class offers some predefined constants. For example the method signature of the `main` method as shown in section 2.5 is represented by:

```
Type   return_type = Type.VOID;
Type[] arg_types   = new Type[] { new ArrayType(Type.STRING, 1) };
```

`Type` objects can be converted to textual signatures with `getSignature()`.

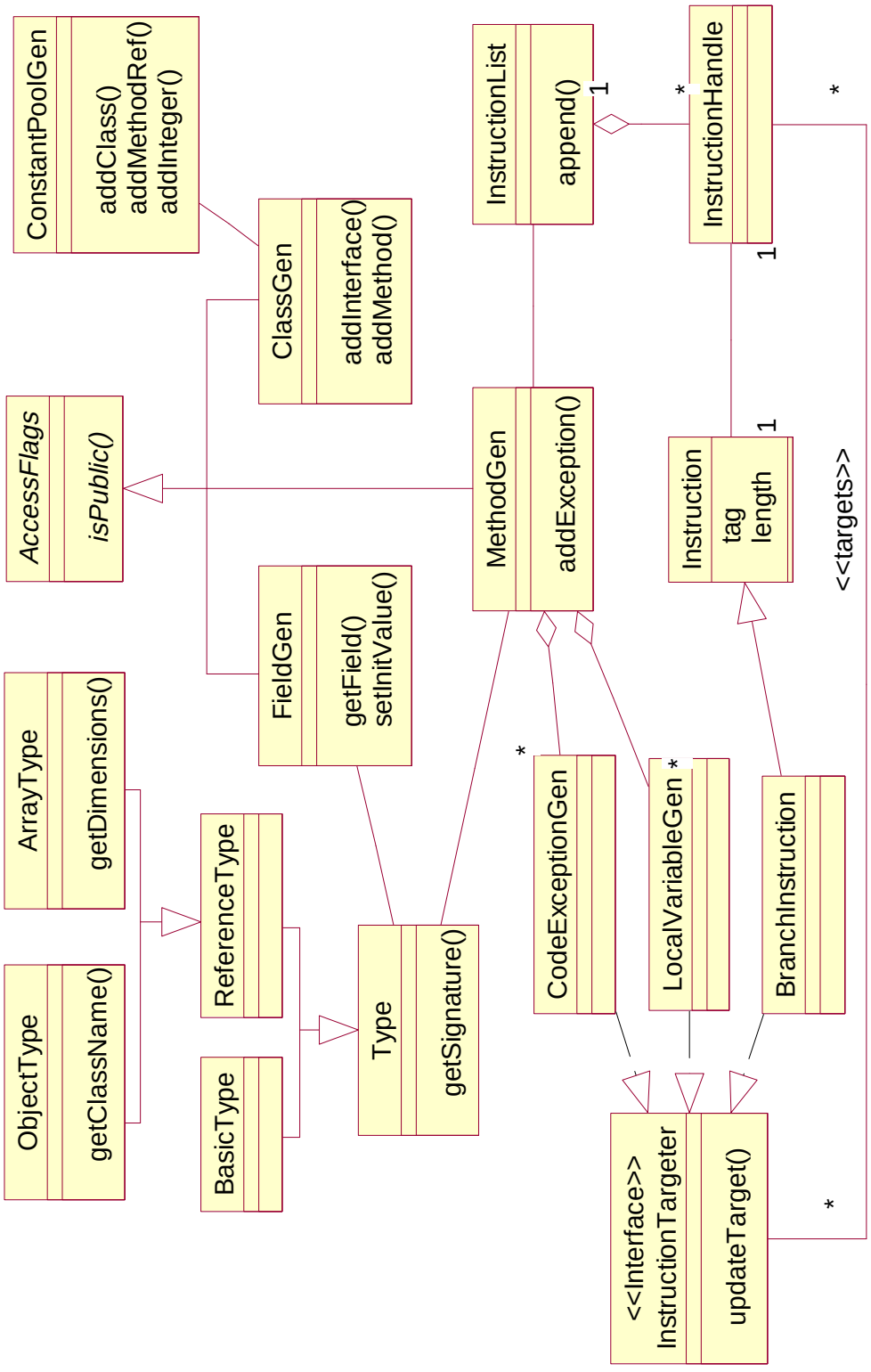

Figure 4: UML diagram of the CLASSGEN API

3.3.2 Generic fields and methods

Fields are represented by `FieldGen` objects. If they have the access rights `static final`, i.e. are constants, they may optionally have an initializing value.

Generic methods contain methods to add local variables, exceptions the method may throw, and exception handlers. Because exception handlers and local variables contain references to byte code addresses, they also take the role of an *instruction targeter* in our terminology. Instruction targeters contain a method `updateTarget()` to redirect a reference. Generic (non-abstract) methods refer to *instruction lists* that consist of instruction objects. References to byte code addresses are implemented by handles to instruction objects. This is explained in more detail in the following sections.

3.3.3 Instruction lists

Modeling instructions as objects may look somewhat odd at first sight, but in fact enables programmers to obtain a high-level view upon control flow without handling details like concrete byte code offsets. Instructions consist of a tag, i.e. an opcode, their length and an offset (or index) within the byte code.

Instructions are grouped via sub-classing, the type hierarchy of instruction classes is illustrated by figure 9 in the appendix. The most important family of instructions are the *branch instructions*, e.g. `goto`, that branch to targets somewhere within the byte code. Obviously, this makes them candidates for playing an `InstructionTargeter` role, too.

For debugging purposes it may even make sense to "invent" your own instructions. In a sophisticated code generator like the one used as a backend of the Barat framework [BS98] one often has to insert temporary `nop` (No operation) instructions. When examining the produced code it may be very difficult to track back where the `nop` was actually inserted. One could think of a derived `nop2` instruction that contains additional debugging information. When the instruction list is dumped to byte code, the extra data is simply dropped.

One could also think of new byte code instructions operating on complex numbers that are replaced by normal byte code upon load-time or are recognized by a new JVM.

An *instruction list* is implemented by a list of *instruction handles* encapsulating instruction objects. References to instructions in the list are thus not implemented by direct pointers to instructions but by pointers to instruction *handles*. This makes appending, inserting and deleting areas of code very simple. Since we use symbolic references, computation of concrete byte code offsets does not need to occur until finalization, i.e. until the user has finished the process of generating or transforming code. We will use the term instruction handle and instruction synonymously throughout the rest of the report.

Appending. One can append instructions or other instruction lists anywhere to an existing list. The instructions are appended after the given instruction handle. All append methods return a new instruction handle which may then be used as the target of a branch instruction, e.g.. For some simple instructions there also exist predefined constants which may be used to reduce memory usage.

```
InstructionList il = new InstructionList();
...
GOTO g = new GOTO(null);
il.append(g);
...
InstructionHandle ih = il.append(InstructionConstants.ACONST_NULL);
```

```
g.setTarget(ih);
```

Inserting. Instructions may be inserted anywhere into an existing list. They are inserted before the given instruction handle. All insert methods return a new instruction handle which may then be used as the start address of an exception handler, for example.

```
InstructionHandle start = il.insert(insertion_point,
                                    InstructionConstants.NOP);
...
mg.addExceptionHandler(start, end, handler, "java.io.IOException");
```

Deleting. Deletion of instructions is also very straightforward, all instruction handles and the contained instructions within a given range are removed from the instruction list and disposed. The `delete()` method may throw a `TargetLostException` when there are instruction targeters still referencing one of the deleted instructions. The user is forced to handle such exceptions in a `try-catch` block and redirect these references elsewhere. The *peep hole* optimizer described in section A.3 gives a detailed example for this.

```
try {
  il.delete(first, last);
} catch(TargetLostException e) {
  InstructionHandle[] targets = e.getTargets();
  for(int i=0; i < targets.length; i++) {
    InstructionTargeter[] targeters = targets[i].getTargeters();
    for(int j=0; j < targeters.length; j++)
       targeters[j].updateTarget(targets[i], new_target);
  }
}
```

Finalizing. When the instruction list is ready to be dumped to pure byte code, all symbolic references must be mapped to real byte code offsets. This is done by the `getByteCode()` method which is called by default by `MethodGen.getMethod()`. Afterwards you can optionally call `dispose()` so that the instruction handles can be reused internally. This helps to reduce memory usage.

```
ClassGen  cg = new ClassGen("HelloWorld", "java.lang.Object",
                            "<generated>", ACC_PUBLIC | ACC_SUPER,
                            null);
MethodGen mg = new MethodGen(ACC_STATIC | ACC_PUBLIC,
                             Type.VOID, new Type[] {
                               new ArrayType(Type.STRING, 1)
                             }, new String[] { "argv" },
                             "main", "HelloWorld", il, cp);
...
cg.addMethod(mg.getMethod());
il.dispose(); // Reuse instruction handles of list
```

3.3.4 Code example revisited

Using instruction lists gives us a generic view upon the code: In Figure 5 we again present the code chunk of the `readInt()` method of the faculty example in section 2.6: The local variables `n` and `e1` both hold two references to instructions, defining their scope. There are two `goto`s branching to the `iload` at the end of the method. One of the exception handlers is displayed, too: it references the start and the end of the `try` block and also the exception handler code.

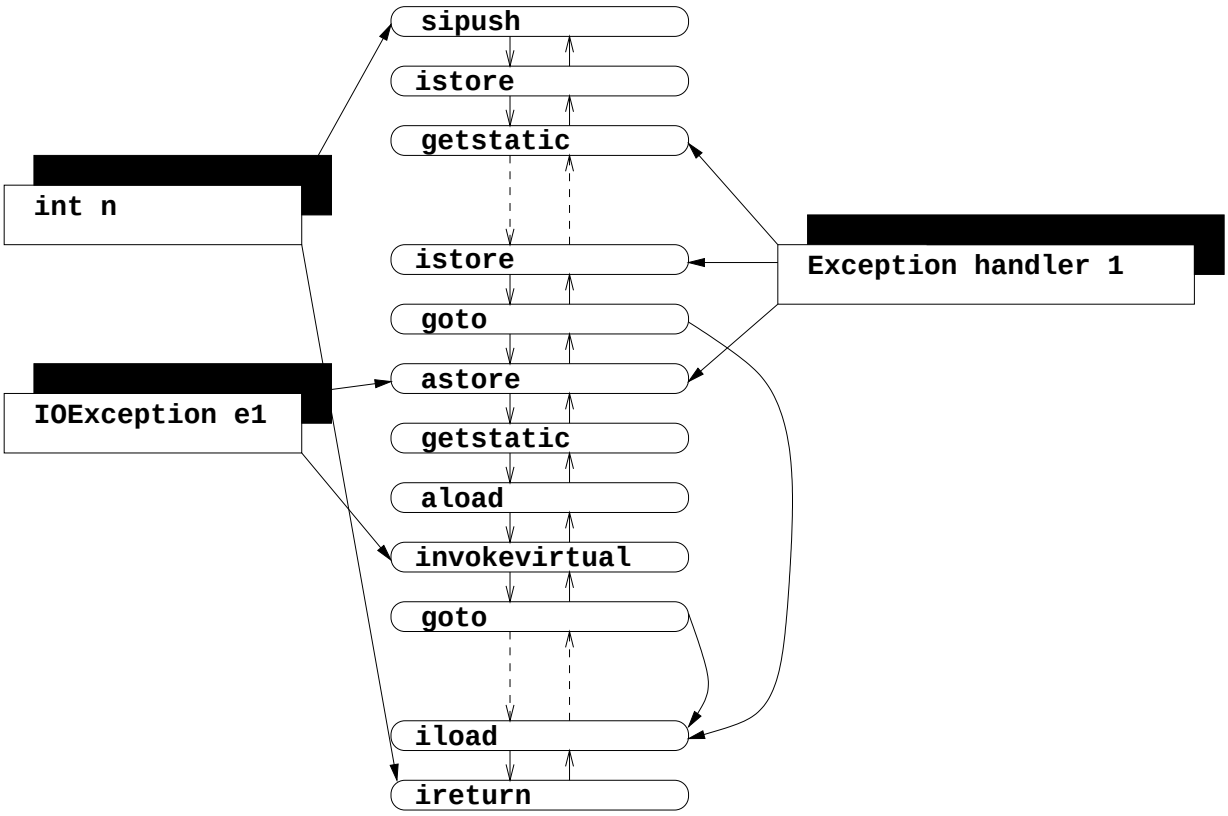

Figure 5: Instruction list for `readInt()` method

3.3.5 Compound instructions

When producing byte code, some patterns typically occur very frequently, for instance the compilation of arithmetic or comparison expressions. You certainly do not want to rewrite the code that translates such expressions into byte code in every place they may appear. Instead you want to use something like an instruction *Factory* [GHJV95]. In order to support this, the CLASSGEN API includes a *compound instruction* (an interface with a single `getInstructionList()` method). Instances of this class may be used in any place where normal instructions would occur, particularly in append operations.

Example: Pushing constants. Pushing constants onto the operand stack may be coded in different ways. As explained in section 2.2 there are some "short-cut" instructions that can be used to make the pro-

duced byte code more compact. The smallest instruction to push a single `1` onto the stack is `iconst_1`, other possibilities are `bipush` (can be used to push values between -128 and 127), `sipush` (between -32768 and 32767), or `ldc` (load constant from constant pool).

Instead of repeatedly selecting the most compact instruction in, say, a switch, one can use the compound `PUSH` instruction whenever pushing a constant number or string. It will produce the appropriate byte code instruction and insert entries into to constant pool if necessary.

```
InstructionList il = new InstructionList();
...
il.append(new PUSH(cp, "Hello, world"));
il.append(new PUSH(cp, 4711));
```

3.3.6 Code patterns using regular expressions

When transforming code, for instance during optimization or when inserting analysis method calls, one typically searches for certain patterns of code to perform the transformation at. To simplify handling such situations CLASSGEN introduces a special feature: One can search for given code patterns within an instruction list using *regular expressions*. In such expressions, instructions are represented by symbolic names, e.g. "`'IfInstruction'`". Meta characters like `+`, `*`, and `(..|..)` have their usual meanings. Thus, the expression

```
"'NOP'+('ILOAD__'|'ALOAD__')*"
```

represents a piece of code consisting of at least one `NOP` followed by a possibly empty sequence of `ILOAD` and `ALOAD` instructions.

The `search()` method of class `FindPattern` gets an instruction list and a regular expression as arguments and returns an array describing the area of matched instructions. Additional constraints to the matching area of instructions, which can not be implemented via regular expressions, may be expressed via *code constraints*.

3.3.7 Example: Optimizing boolean expressions.

In Java, boolean values are mapped to 1 and to 0, respectively. Thus, the simplest way to evaluate boolean expressions is to push a 1 or a 0 onto the operand stack depending on the truth value of the expression. But this way, the subsequent combination of boolean expressions (with &&, e.g) yields long chunks of code that push lots of 1s and 0s onto the stack.

When the code has been finalized these chunks can be optimized with a *peep hole* algorithm: An `IfInstruction` (e.g. the comparison of two integers: `if_icmpeq`) that either produces a 1 or a 0 on the stack and is followed by an `ifne` instruction (branch if stack value $\neq 0$) may be replaced by the `IfInstruction` with its branch target replaced by the target of the `ifne` instruction:

```
InstructionList il = new InstructionList();
...
CodeConstraint constraint = new CodeConstraint() {
  public boolean checkCode(InstructionHandle[] match) {
    IfInstruction if1 = (IfInstruction)match[0].getInstruction();
```

```
    GOTO          g   = (GOTO)match[2].getInstruction();
    return (if1.getTarget() == match[3]) &&
           (g.getTarget() == match[4]);
  }
};
FindPattern f   = new FindPattern(il);
String      pat = "'IfInstruction''ICONST_0''GOTO''ICONST_1'" +
                  "'NOP'('IFEQ'|'IFNE')";
InstructionHandle[] match;
for(InstructionHandle ih = f.search(pat, constraint);
    ih != null; ih = f.search(pat, match[0], constraint)) {
  match = f.getMatch(); // Constraint already checked
  ...
  match[0].setTarget(match[5].getTarget()); // Update target
  ...
  try {
    il.delete(match[1], match[5]);
  } catch(TargetLostException e) { ... }
}
```

The applied code constraint object ensures that the matched code really corresponds to the targeted expression pattern. Subsequent application of this algorithm removes all unnecessary stack operations and branch instructions from the byte code. If any of the deleted instructions is still referenced by an `InstructionTargeter` object, the reference has to be updated in the `catch`-clause.

Code example A.1 gives a verbose example of how to create a class file, while example A.3 shows how to implement a simple peephole optimizer and how to deal with `TargetLost` exceptions.

Example application: The expression

```
if((a == null) || (i < 2))
  System.out.println("Ooops");
```

can be mapped to both of the chunks of byte code shown in figure 3.3.7. The left column represents the unoptimized code while the right column displays the same code after an aggressively optimizing peep hole algorithm has been applied:

4 Application areas

There are many possible application areas for JAVACLASS ranging from class browsers, profilers, byte code optimizers, and compilers to sophisticated run-time analysis tools and extensions to the Java language [AFM97, MBL97].

Compilers like the Barat compiler [BS98] use JAVACLASS to implement a byte code generating back end. Other possible application areas are the static analysis of byte code [TK98] or examining the run-time behavior of classes by inserting calls to profiling methods into the code. Further examples are extending Java with Eiffel-like assertions [FM98], automated delegation [Cos98], or with the concepts of "Aspect-Oriented Programming" [KLM+97].

```
5:  aload_0
6:  ifnull        #13
9:  iconst_0
10: goto          #14
13: iconst_1
14: nop
15: ifne          #36
18: iload_1
19: iconst_2
20: if_icmplt     #27
23: iconst_0
24: goto          #28
27: iconst_1
28: nop
29: ifne          #36
32: iconst_0
33: goto          #37
36: iconst_1
37: nop
38: ifeq          #52
41: getstatic     System.out
44: ldc           "Ooops"
46: invokevirtual println
52: return
```

```
10: aload_0
11: ifnull        #19
14: iload_1
15: iconst_2
16: if_icmpge     #27
19: getstatic     System.out
22: ldc           "Ooops"
24: invokevirtual println
27: return
```

Figure 6: Optimizing boolean expressions

4.1 Class loaders

Class loaders are responsible for loading class files from the file system or other resources and passing the byte code to the Virtual Machine [LB98]. A custom `ClassLoader` object may be used to intercept the standard procedure of loading a class, i.e. the system class loader, and perform some transformations before actually passing the byte code to the JVM.

A possible scenario is described in figure 7: During run-time the Virtual Machine requests a custom class loader to load a given class. But before the JVM actually sees the byte code, the class loader makes a "side-step" and performs some transformation to the class. To make sure that the modified byte code is still valid and does not violate any of the JVM's rules it is checked by the verifier before the JVM finally executes it.

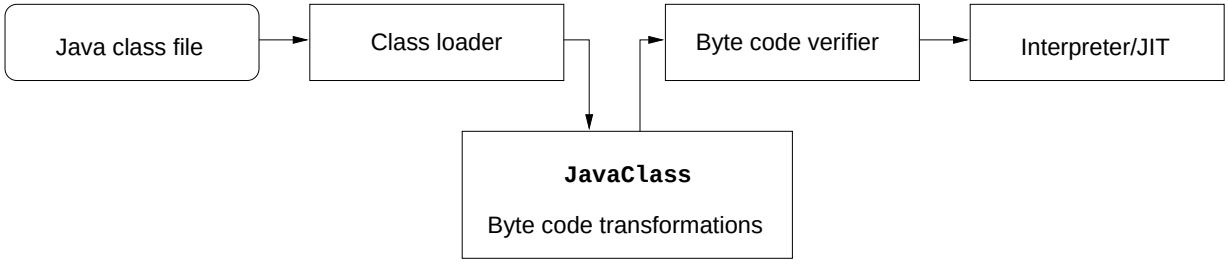

Figure 7: Class loaders

Using class loaders is an elegant way of extending the Java Virtual Machine with new features without actually modifying it. This concept enables developers to use *load-time reflection* to implement their ideas as opposed to the static reflection supported by the Java Reflection API [Jav98]. Load-time transformations supply the user with a new level of abstraction. He is not strictly tied to the static constraints of the original authors of the classes but may customize the applications with third-party code in order to benefit from new features. Such transformations may be executed on demand and neither interfere with other users, nor alter the original byte code. In fact, class loaders may even create classes *ad hoc* without loading a file at all.

4.1.1 Example: Poor Man's Generictity

The "Poor Man's Genericity" project [BD98] that extends Java with parameterized classes, for example, uses JAVACLASS in two places to generate instances of parameterized classes: During compile-time (the standard `javac` with some slightly changed classes) and at run-time using a custom class loader. The compiler puts some additional type information into class files which is evaluated at load-time by the class loader. The class loader performs some transformations on the loaded class and passes them to the VM. The following algorithm illustrates how the load method of the class loader fulfills the request for a parameterized class, e.g. `Stack<String>`

1. Search for class `Stack`, load it, and check for a certain class attribute containing additional type information. I.e. the attribute defines the "real" name of the class, i.e. `Stack<A>`.

2. Replace all occurences and references to the formal type `A` with references to the actual type `String`. For example the method

```
void push(A obj) { ... }
```

becomes

```
void push(String obj) { ... }
```

3. Return the resulting class to the Virtual Machine.

5 Conclusions and future work

In this report we presented the JAVACLASS API that is intended to be a general purpose tool for byte code engineering. It helps developers to implement analysis tools or byte code transformations conveniently. It has proved to be useful in several projects and is not restricted to a special kind of application area.

We found two issues of the API that may be considered as drawbacks: The generic constant pool is a "Write-only" data structure, i.e. constant pool entries can be added and retrieved but not be removed directly. They are referenced via integer indexes and not some kind of virtual handle. We think that the removal of entries from the constant pool is rarely an issue and that implementing the access to it via handles would cause too much overhead. One would rather write a supplementary tool to strip unnecessary entries from classes. The second issue may be not to encapsulate instructions into instruction handles anymore but to put the necessary code directly into the instructions. Yet we feel that this would not give us such a clear and elegant level of abstraction as it does now.

We are currently developing an API that makes the development of Java extensions with class loaders more comfortable and offers new possibilities: It will allow to make class loader transformations *composable*, i.e. the class loader is not restricted to perform a single transformation but may execute a dynamically configurable sequence of transformations.

References

[AFM97] O. Agesen, S. N. Freund, and J. C. Mitchell. Adding Type Parameterization to the Java Language. In *Proceedings OOPSLA'97*, Atlanta, GA, 1997.

[AP98] D. Antonioli and M. Pilz. Statistische Analyse von Java-Classfiles. In Clemens Cap, editor, *Proceedings JIT'98*. Springer, 1998.

[BD98] B. Bokowski and M. Dahm. Poor Man's Genericity for Java. In Clemens Cap, editor, *Proceedings JIT'98*. Springer, 1998.

[BS98] B. Bokowski and A. Spiegel. Barat – A Front-End for Java. Technical report, Freie Universität Berlin, 1998.

[CCK98] Geoff Cohen, Jeff Chase, and David Kaminsky. Automatic Program Transformation with JOIE. In *Proceedings USENIX Annual Technical Symposium*, 1998.

[CCZ97] Suzanne Collin, Dominique Colnet, and Olivier Zendra. Type Inference for Late Binding. The SmallEiffel Compiler. In *Proceedings JMLC'97*, 1997.

[Cos98] Pascal Costanza. *The ClassFilters package*. Universität Bonn, http://www.cs.uni-bonn.de/~costanza/ClassFilters/, 1998.

[FM98] C. Fischer and D. Meemken. JaWa: Java with Assertions. In Clemens Cap, editor, *Proceedings JIT'98*. Springer, 1998.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[GJS96] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, 1996.

[Jav98] JavaSoft. *Reflection API*. http://java.sun.com/products/jdk/1.1/docs/guide/reflectio 1998.

[KH98] Ralph Keller and Urs Hölzle. Binary Component Adaptation. In Eric Jul, editor, *Proceedings ECOOP'98*. Springer, 1998.

[KLM+97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. Technical report, Xerox Palo Alto Research Center, 1997.

[LB98] Sheng Lian and Gilad Bracha. Dynamic Class Loading in the Java Virtual Machine. In *Proceedings OOPSLA'98*, 1998.

[LY97] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1997.

[LZ98] Han Bok Lee and Benjamin G. Zorn. BIT: A Tool for Instrumenting Java Bytecodes. In *Proceedings USENIX Symposium on Internet Technologies and Systems*, 1998.

[MBL97] A.C. Myers, J. A. Bank, and B. Liskov. Parameterized Types for Java. In *Proceedings POPL'97*, Paris, France, 1997.

[MD97] J. Meyer and T. Downing. *Java Virtual Machine*. O'Reilly, 1997.

[Sil98] Shawn Silverman. *The classfile API*. University of Manitoba, http://Meurrens.ML.org/ip-Links/java/codeEngineering/viewers.html, 1998.

[Taf96] Tucker Taft. Programming the Internet in Ada95. In *Proceedings Ada-Europe International Conference on Reliable Software Technologies*, 1996.

[TK98] M. Thies and U. Kastens. Statische Analyse von Bibliotheken als Grundlage dynamischer Optimierung. In Clemens Cap, editor, *Proceedings JIT'98*. Springer, 1998.

[You98] Matt T. Yourst. *Inside Java Class Files*. Laserstars Technologies, http://www.laserstars.com/articles/ddj/insidejcf/, 1998.

A Code examples for the ClassGen API

A.1 HelloWorldBuilder.java

The following Java program reads a name from the standard input and prints a friendly “Hello”. Since the `readLine()` method may throw an `IOException` it is enclosed by a `try-catch` block.

```
import java.io.*;

public class HelloWorld {
  public static void main(String[] argv) {
    BufferedReader in = new BufferedReader(new
                          InputStreamReader(System.in));
    String name = null;

    try {
      System.out.print("Please enter your name> ");
      name = in.readLine();
    } catch(IOException e) { return; }

    System.out.println("Hello, " + name);
  }
}
```

A.2 HelloWorldBuilder.java

We will sketch here how the above Java class can be created from the scratch using the CLASSGEN API. For ease of reading we will use textual signatures and not create them dynamically. For example, the signature

```
"(Ljava/lang/String;)Ljava/lang/StringBuffer;"
```

would actually be created with

```
Type.getMethodSignature(Type.STRINGBUFFER, new Type[] { Type.STRING });
```

A.2.1 Initialization:

First we create an empty class and an instruction list:

```
ClassGen  cg = new ClassGen("HelloWorld", "java.lang.Object",
                            "<generated>", ACC_PUBLIC | ACC_SUPER,
                            null);
ConstantPoolGen cp = cg.getConstantPool(); // cg creates constant pool
InstructionList il = new InstructionList();
```

We then create the main method, supplying the method's name and the symbolic type signature encoded with `Type` objects.

```
MethodGen  mg = new MethodGen(ACC_STATIC | ACC_PUBLIC,// access flags
                              Type.VOID,              // return type
                              new Type[] {            // argument types
                                new ArrayType(Type.STRING, 1) },
                              new String[] { "argv" }, // arg names
                              "main", "HelloWorld",    // method, class
                              il, cp);
```

We add some often used constants to the constant pool:

```
int br_index   = cp.addClass("java.io.BufferedReader");
int ir_index   = cp.addClass("java.io.InputStreamReader");
int system_out = cp.addFieldref("java.lang.System", "out", // System.out
                                "Ljava/io/PrintStream;");
int system_in  = cp.addFieldref("java.lang.System", "in",  // System.in
                                "Ljava/io/InputStream;");
```

A.2.2 Create variables in and name:

We call the constructors, i.e. execute `BufferedReader(InputStreamReader(System.in))`. The reference to the `BufferedReader` object stays on top of the stack and is stored in the newly allocated `in` variable.

```
il.append(new NEW(br_index)); // create BufferedReader reference
il.append(InstructionConstants.DUP);  // duplicate reference
il.append(new NEW(ir_index)); // same for InputStreamReader
il.append(InstructionConstants.DUP); // reuse predefined constant
il.append(new GETSTATIC(system_in));
il.append(new INVOKESPECIAL(cp.addMethodref("java.io.InputStreamReader",
                                            "<init>",
                                            "(Ljava/io/InputStream;)V")));
il.append(new INVOKESPECIAL(cp.addMethodref("java.io.BufferedReader",
                                            "<init>",
                                            "(Ljava/io/Reader;)V")));
LocalVariableGen lg = mg.addLocalVariable("in",
                                          new ObjectType(
                                            "java.io.BufferedReader"),
                                           null, null);
int in = lg.getIndex();
lg.setStart(il.append(new ASTORE(in))); // 'in' valid from here
```

Create local variable `name` and initialize it to `null`.

```
lg = mg.addLocalVariable("name", Type.STRING, null, null);
int name = lg.getIndex();
il.append(InstructionConstants.ACONST_NULL); // save memory by using constant
lg.setStart(il.append(new ASTORE(name))); // ‘name’ valid from here
```

A.2.3 Create try-catch block

We remember the start of the block, read a line from the standard input and store it into the variable `name`.

```
InstructionHandle try_start = il.append(new GETSTATIC(system_out));
il.append(new PUSH(cp, "Please enter your name> "));
il.append(new INVOKEVIRTUAL(cp.addMethodref("java.io.PrintStream",
                                            "print",
                                            "(Ljava/lang/String;)V")));
il.append(new ALOAD(in));
il.append(new INVOKEVIRTUAL(cp.addMethodref("java.io.BufferedReader",
                                            "readLine",
                                            "()Ljava/lang/String;")));
il.append(new ASTORE(name));
```

Upon normal execution we jump behind exception handler, the target address is not known yet.

```
GOTO g = new GOTO(null);
InstructionHandle try_end = il.append(g);
```

We add the exception handler which simply returns from the method.

```
InstructionHandle handler = il.append(new RETURN());
mg.addExceptionHandler(try_start, try_end, handler, "java.io.IOException");
```

“Normal” code continues, now we can set the branch target of the GOTO.

```
InstructionHandle ih = il.append(new GETSTATIC(system_out));
g.setTarget(ih);
```

A.2.4 Printing ”Hello”

String concatenation compiles to `StringBuffer` operations.

```
il.append(new NEW(cp.addClass("java.lang.StringBuffer")));
il.append(InstructionConstants.DUP);
il.append(new PUSH(cp, "Hello, "));
il.append(new INVOKESPECIAL(cp.addMethodref("java.lang.StringBuffer",
                                            "<init>",
                                            "(Ljava/lang/String;)V")));
il.append(new ALOAD(name));
il.append(new INVOKEVIRTUAL(cp.addMethodref("java.lang.StringBuffer",
```

```
                                                "append",
                                                 "(Ljava/lang/String;)" +
                                                 "Ljava/lang/StringBuffer;")));

  il.append(new INVOKEVIRTUAL(cp.addMethodref("java.lang.StringBuffer",
                                                "toString",
                                                "()Ljava/lang/String;")));

  il.append(new INVOKEVIRTUAL(cp.addMethodref("java.io.PrintStream",
                                                "println",
                                                "(Ljava/lang/String;)V")));
  il.appendInstructionConstants.RETURN);
```

A.2.5 Finalization

Finally, we have to set the stack size, which normally would be computed on the fly and add a default constructor method to the class, which is empty in this case.

```
  mg.setMaxStack(5);
  cg.addMethod(mg.getMethod());
  cg.addEmptyConstructor(ACC_PUBLIC);
```

Last but not least we dump the `JavaClass` object to a file.

```
  try {
    cg.getJavaClass().dump("HelloWorld.class");
  } catch(java.io.IOException e) { System.err.println(e); }
```

A.3 Peephole.java

This class implements a simple peephole optimizer that removes any NOP instructions from the given class.

```
import java.io.*;
import de.fub.bytecode.classfile.*;
import de.fub.bytecode.generic.*;
import de.fub.bytecode.ClassPath;

public class Peephole {
  public static void main(String[] argv) {
    try {
      /* Load the class from CLASSPATH.
       */
      ClassPath       class_path = new ClassPath();
      InputStream     is         = class_path.getInputStream(argv[0]);
      ClassParser     parser     = new ClassParser(is, argv[0]);
```

```
    JavaClass       clazz      = parser.parse();
    Method[]        methods    = clazz.getMethods();
    ConstantPoolGen cp         = new ConstantPoolGen(clazz.
                                                     getConstantPool());

    for(int i=0; i < methods.length; i++) {
      MethodGen mg       = new MethodGen(methods[i],
                                         clazz.getClassName(), cp);
      Method    stripped = removeNOPs(mg);

      if(stripped != null)     // Any NOPs stripped?
        methods[i] = stripped; // Overwrite with stripped method
    }

    /* Dump the class to <class name>_.class
     */
    clazz.setConstantPool(cp.getFinalConstantPool());
    clazz.dump(clazz.getClassName() + "_.class");
  } catch(Exception e) { e.printStackTrace(); }
}

private static final Method removeNOPs(MethodGen mg) {
  InstructionList   il    = mg.getInstructionList();
  FindPattern       f     = new FindPattern(il);
  String            pat   = "(‘NOP’)+"; // Find at least one NOP
  InstructionHandle next  = null;
  int               count = 0;

  for(InstructionHandle ih = f.search(pat);
      ih != null;
      ih = f.search(pat, next)) {

    InstructionHandle[] match = f.getMatch();
    InstructionHandle   first = match[0];
    InstructionHandle   last  = match[match.length - 1];

    /* Some nasty Java compilers may add NOP at end of method.
     */
    if((next = last.getNext()) == null)
      break;

    count += match.length;

    /* Delete NOPs and redirect any references to them to the following
     * (non-nop) instruction.
```

```
       */
      try {
        il.delete(first, last);
      } catch(TargetLostException e) {
        InstructionHandle[] targets = e.getTargets();
        for(int i=0; i < targets.length; i++) {
          InstructionTargeter[] targeters = targets[i].getTargeters();

          for(int j=0; j < targeters.length; j++)
            targeters[j].updateTarget(targets[i], next);
        }
      }
    }

    if(count > 0) {
      System.out.println("Removed " + count + " NOP instructions from method " +
                         mg.getMethodName());
      return mg.getMethod();
    }
    else
      return null;
  }
}
```

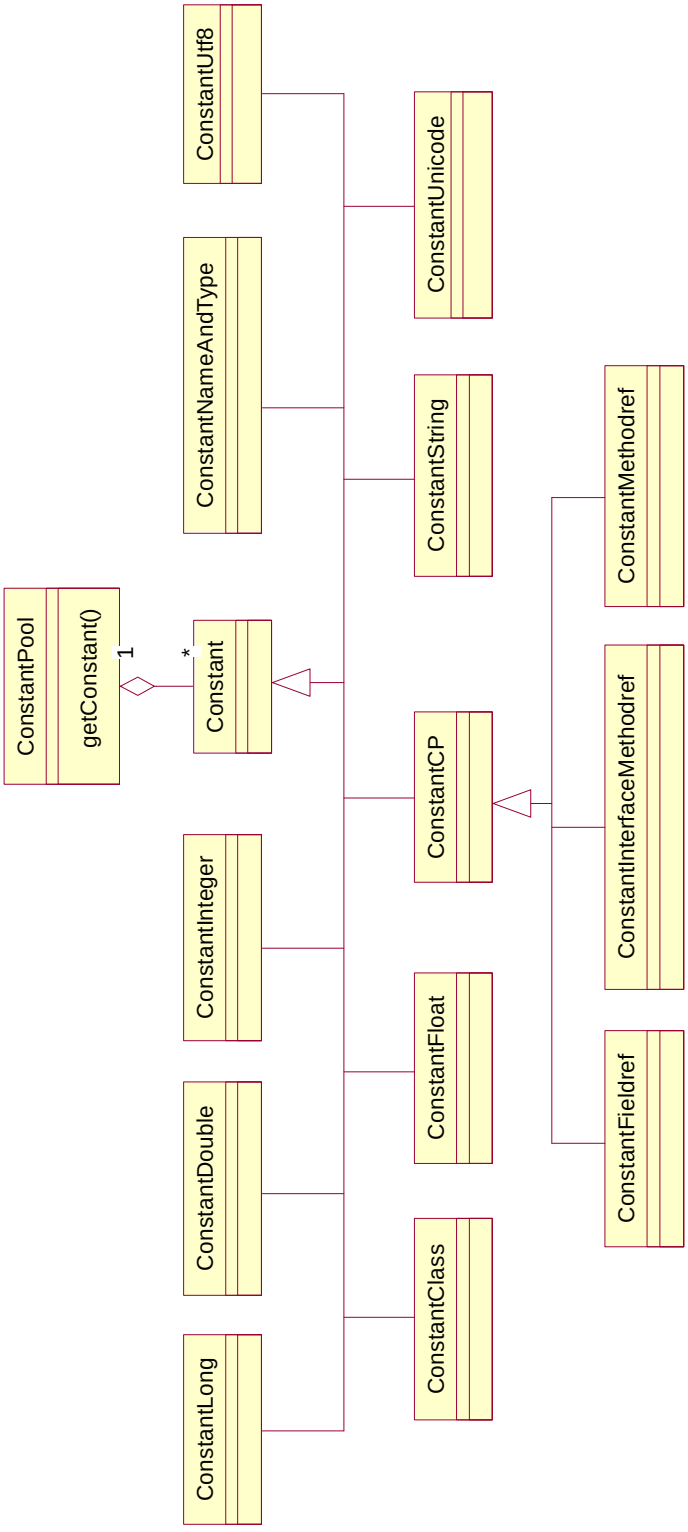

Figure 8: UML diagram for the ConstantPool API

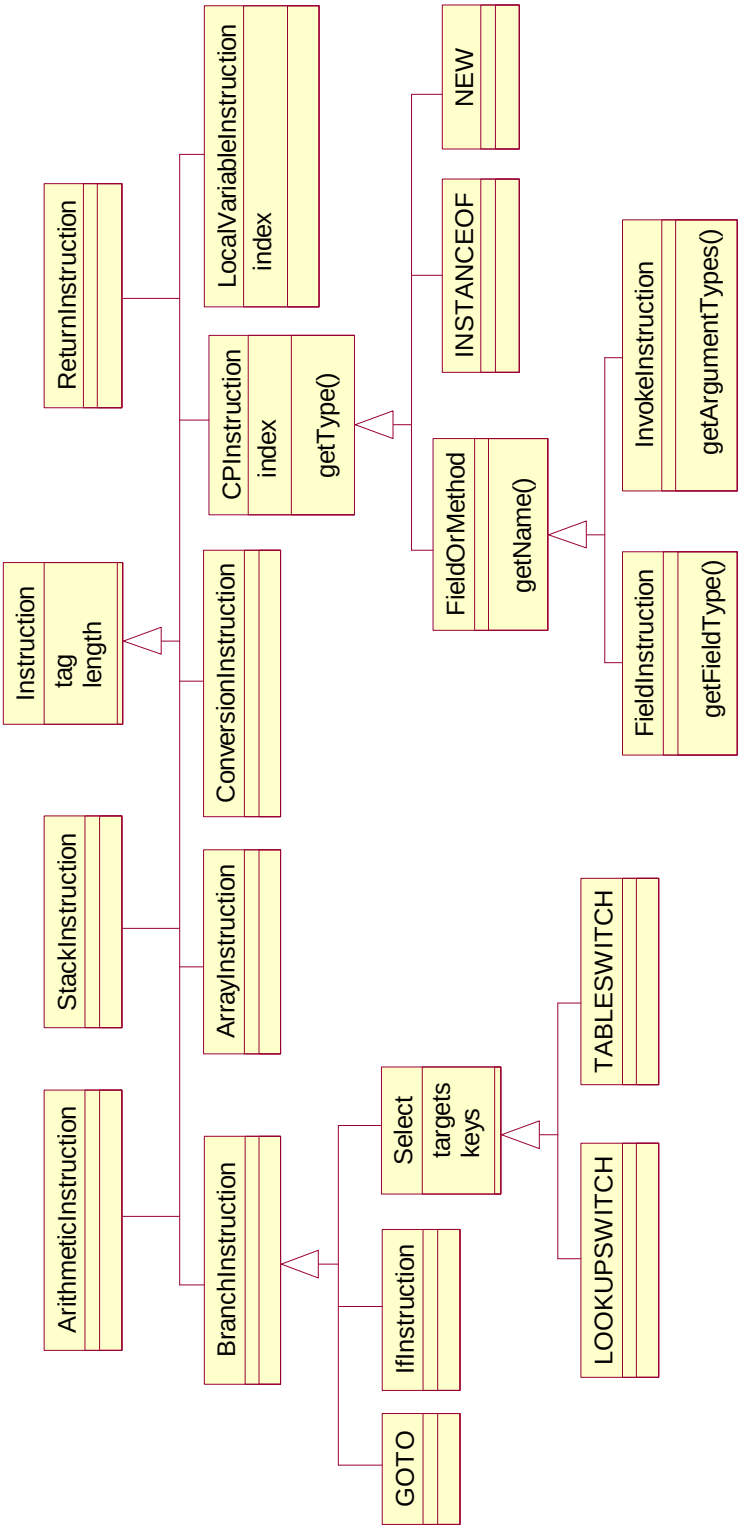

Figure 9: UML diagram for the Instruction API