Byte Code Engineering

Markus Dahm

Freie Universität Berlin
dahm@inf.fu-berlin.de

Abstract. The term "Java" is used to denote two different concepts: the language itself and the related execution environment, the Java Virtual Machine (JVM), which executes *byte code* instructions. Several research projects deal with byte code-generating compilers or the implementation of new features via byte code transformations. Examples are code optimization, the implementation of parameterized types for Java, or the adaptation of run-time behavior through load-time transformations. Many programmers are doing this by implementing their own specialized byte code manipulation tools, which are, however, restricted in the range of their reusability. Therefore, we have developed a general purpose framework for the static analysis and dynamic creation or transformation of byte code. In this paper we present its main features and possible application areas.

1 Introduction

Many research projects deal with extensions of the Java language [13] or improvements of its run-time behavior. Implementing new features in the Java execution environment (Java Virtual Machine, JVM) is relatively easy compared to other languages, because Java is an interpreted language with a small and easy-to-understand set of instructions (the *byte code*).

The JavaClass API which we present in this paper is a framework for the static analysis and dynamic creation or transformation of Java class files.[1] It enables developers to deal with byte code on a high level of abstraction without handling all the internal details of the Java class file format. There are many possible application areas ranging from class browsers, profilers, byte code-optimizers, and compilers, to sophisticated run-time analysis tools and extensions to the Java language [1,21,3]. Other possibilities include the static analysis of byte code [22], automated delegation [8], or implementing concepts of "Aspect-Oriented Programming" [16]. We think that the most interesting application area for JavaClass is meta-level programming, i.e. load-time reflection [18], which will be discussed in detail in section 3.1.

Our approach provides a truly object-oriented view upon Java byte code. For example, code is modeled as a list of instructions objects. Within such a list one may add or delete instructions, change the control flow, or search for certain patterns of code using regular expressions.

We assume the reader to have some basic knowledge about the JVM and Java class files. A more detailed introduction to the API and the Virtual Machine can be found in

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

[9]. The paper is structured as follows: We first give a brief overview of related work and present some aspects and technical details of the framework in section 2. We then discuss concepts of byte code engineering and possible application areas in section 3 and conclude with section 4.

1.1 Related work

The JOIE [7] toolkit can be used to augment class loaders with dynamic behavior. Similarly, "Binary Component Adaptation" [15] allows classes to be adapted and evolved on-the-fly. Han Lee's "Byte-code Instrumenting Tool" [17] allows the user to insert calls to analysis methods anywhere in the byte code. The Jasmin assembler [20] can be used to compile pseudo-assembler code. Kawa, a Java-based Scheme system, contains the `gnu.bytecode` package [5] to generate byte code. The metaXa Virtual Machine [12] allows to dynamically *reify* meta level events, e.g. instance field access.

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

2 The JavaClass framework

The JAVACLASS framework consists of a "static" and a "generic" part. The former is not intended for byte code modifications. It may be used, e.g., to analyze Java classes without having the source files at hand. The latter supplies an abstraction level for creating or transforming class files dynamically. It makes the static constraints of Java class files, like hard-coded byte code addresses, mutable. Using the term "generic" here may be a bit misleading, we should perhaps rather speak of a "generating" API. UML diagrams – unfortunately too large for this paper – describing the class hierarchy of the framework can be found in [9].

2.1 Static API

All of the binary components and data structures declared in the JVM specification [19] are mapped to classes, where the top-level class is called `JavaClass`, giving the whole API its name. Instances of this class basically consist of a *constant pool*, fields, methods, symbolic references to the super class and to the implemented interfaces of the class. At run-time, these objects can be used as *meta objects* describing the contents of a class. This possibility will be discussed in detail in section 3.1.

The constant pool serves as a central repository of the class and contains, e.g., entries describing the type signature of methods and fields. It also contains String, Integer, and other constants. Indexes to the constant pool may be contained in byte code instructions as well as in other components of a class file and in constant pool entries themselves.

Analyzing classes. Information within the class file components can be accessed via an intuitive set/get interface. Compilers may use the framework to analyze binary class files, e.g. in order to check whether they contain certain fields or methods. Using the provided class repository, implementing a simple class viewer is quite easy:

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

System.out.println(clazz);          // Print class contents
Method[] methods = clazz.getMethods();

for(int i=0; i < methods.length; i++) {
  System.out.println(methods[i]);   // Print method signature

  Code code = methods[i].getCode(); // Print byte code of
  if(code != null)                  // non-abstract,
    System.out.println(code);       // non-native methods
}
```

JAVACLASS supports the *Visitor* design pattern [10], i.e. it allows developers to write their own visitors to traverse and analyze the contents of a class file. Included in the distribution, e.g., is a class `JasminVisitor` that converts class files into the Jasmin assembler language [20].

2.2 Generic API

This part of the API makes it possible to modify byte code components dynamically. The generic constant pool, for example, implemented by the class `ConstantPoolGen`, offers methods for adding different types of constants. Accordingly, `ClassGen` offers routines to add or delete methods, fields, and class attributes.

Representation of types. We abstract from the concrete details of type signature syntax of the JVM [19] by introducing the `Type` class, which is used, for example, to define the types of methods. Concrete sub-classes are `BasicType`, `ObjectType` and `ArrayType`. There are also some predefined constants for common types. For example, the type signature of the `main` method is represented by:

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

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.

`MethodGen` objects contain routines to add local variables, thrown exceptions, 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 such references. The code of methods is represented by *instruction lists* that contain instruction objects. References to byte code addresses are implemented by handles to instruction objects. This is explained in more detail in the following sections.

Instruction objects. 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 addresses. Instruction objects basically consist of a tag, i.e. an opcode and their length in bytes. The instruction set of the Java Virtual Machine distinguishes its operand types using different instructions to operate on values of specific type. For example `iload` loads an integer value onto the stack, while `fload` loads a float value.

Our approach enables us to group instructions via sub-classing. For example, there are *branch instructions* like `goto` and `if_icmpeq`, which compares two integers for equality. They additionally contain an address (offset) within the byte code as the branch target. Obviously, this makes them candidates for playing an instruction targeter role, too.

For debugging purposes it may even make sense to "invent" your own instructions. In a sophisticated code generator, e.g., it may be very difficult to track back in which method particular code has actually been created. One could think of a specialized `nop` (No operation) instruction (which may be inserted anywhere without effect) that contains additional debugging information. 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.

We will not list all byte code instructions here, since these are explained in detail in the JVM specification [19]. The opcode names are mostly self-explaining, so understanding the following code examples should be fairly intuitive.

2.3 Instruction lists

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. Instruction handles can be used as some kind of symbolic addresses, for example in order to define the scope of an exception handler:

```
InstructionHandle start, end, handler;
... // Define start and end of handled area, and handler code
ObjectType ot = new ObjectType("java.io.IOException");
mg.addExceptionHandler(start, end, handler, ot);
```

We will use the terms instruction and instruction handle synonymously for the rest of the paper.

Appending instructions. Instructions can be appended before or after any given handle within a list. The default is to append the instruction to the end of the list. All append methods return a new instruction handle which may then be used as the target of a branch instruction, e.g..

```
InstructionList il = new InstructionList();
GOTO g = new GOTO(null); // Target of branch not known yet
il.append(g);            // Append it to the end of the list
...
g.setTarget(il.append(new NOP())); // Set branch target
```

Deleting instructions. 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 and redirect those references where the necessary informations are stored in the exception object.

```
try {
  il.delete(start, end);
} catch(TargetLostException e) { ... }
```

2.4 Instruction factories

When producing byte code, some patterns typically occur very frequently, for instance the compilation of arithmetic or comparison expressions. One certainly does not want to copy the code that translates such expressions to every place they may appear. Instead, we supply instruction *factories* [10] and *compound instructions*. Instances of the latter may be appended just like normal instructions.

For example, pushing constants onto the operand stack may be coded in different ways. There are some "short-cut" instructions that can be used to make the produced 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 instructions when added to an instruction list and automatically insert entries into the constant pool if necessary.

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

2.5 Representing exception handlers and local variables

In Figure 1 we present how the code of the `readInt()` method is mapped to an instruction list. The local variables `n` and `e` both hold two references to instructions, defining their scope. An exception handlers is displayed, too: it references the start and the end of the `try` block and also the handling code. Altogether, there are three kinds of instruction targeters: local variables, exception handlers and branch instructions.

```
private static BufferedReader in =
  new BufferedReader(new
    InputStreamReader(System.in));

public static final int readInt() {
  int n = 4711;

  try {
    n = Integer.parseInt(in.readLine());
  } catch(IOException e) {
    System.err.println(e);
  }
  ...

  return n;
}
```

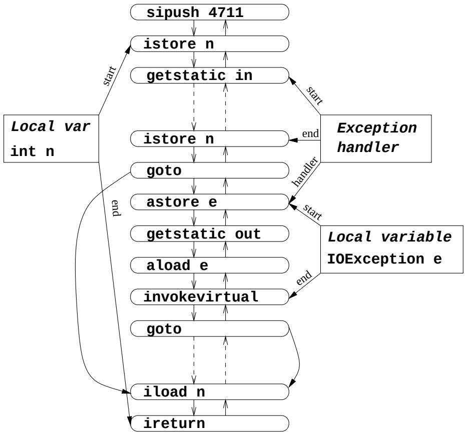

Fig. 1. Instruction list for `readInt()` method

2.6 Code patterns

When transforming code, for instance during optimization or when inserting analysis method calls, one typically searches for certain patterns of code on which to perform the transformation. In order to handle such situations JavaClass includes a novel feature: One can search for given code patterns within an instruction list using *regular expressions*. In regular expressions, instructions may be represented by symbolic names, e.g. `‘IfInstruction’` or `‘ILOAD__’`. Meta symbols like `+`, `*`, and `(..|..)` have their usual meanings. Currently we are using string literals to represent regular expressions, a future version of the API may introduce a more advanced data structure. Additional constraints to the matching area of instructions, which can not be implemented in terms of regular expressions, may be expressed via *code constraints*. Section 2.8 presents an example for the usage of this feature.

2.7 Example I: Compiling an `if` statement

The following example shows how a simple compiler would translate the Java statement

```
if(a == null)
  a = b;
```

into byte code:

```
InstructionList il = new InstructionList();
IfInstruction   i  = new IFNONNULL(null);

il.append(new ALOAD(0));  // Load local variable 0(a) on stack
il.append(i);             // Use negated condition
il.append(new ALOAD(1));  // Load local variable 0(b) on stack
il.append(new ASTORE(0)); // Store in a
// Define auxiliary target for the case a != null
i.setTarget(il.append(new NOP()));
```

2.8 Example II: Peep hole optimizer

In Java, boolean values are mapped to 1 and 0. Thus, the simplest way to evaluate boolean expressions like `a == null` during compilation is to push a 1 or a 0 onto the operand stack. 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. Additionally, one has to add a lot of `nop` operations as auxiliary branch targets.

Such code chunks can be optimized using a *peep hole* algorithm [2]: An `IfInstruction`, that either produces a 1 or a 0 on the stack and is followed by an `ifeq` instruction (branch if stack operand equals to 0) is replaced by the negated `IfInstruction` with its branch target replaced by the target of the `ifeq` instruction. Figure 2 illustrates the results of the algorithm: The left column shows the original Java expression, the middle one the naive translation into byte code, and the right one shows an optimized version.

```
if(a == null)   1: aload_0          1: aload_0
  a = b;        2: ifnull   #5      2: ifnonnull #5
                3: iconst_0         3: aload_1
                4: goto     #6      4: astore_0
                5: iconst_1         5: nop
                6: nop
                7: ifeq     #10
                8: aload_1
                9: astore_0
               10: nop
```

Fig. 2. Optimizing byte code

This algorithm can be implemented like this:

```
FindPattern f = new FindPattern(il);
f.search("‘IfInstruction’‘ICONST_0’‘GOTO’" +
         "‘ICONST_1’‘NOP’‘IFEQ’", constraint);

InstructionHandle[] match = f.getMatch();
IfInstruction       if_   = (IfInstruction)match[0].
  getInstruction().negate();           // Negate instruction

match[0].setInstruction(if_);          // Replace instruction
if_.setTarget(match[5].getTarget());   // Update branch target
try {
  il.delete(match[1], match[5]);       // Remove obsolete code
} catch(TargetLostException e) { ... } // Update targeters
```

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 instruction targeter object, the reference needs to be updated in the `catch`-clause.

Code constraints. The above applied *code constraint* object ensures that the matched code really can be transformed, i.e. it checks the targets of the branch instructions:

```
CodeConstraint constraint = new CodeConstraint() {
  public boolean checkCode(InstructionHandle[] m) {
    IfInstruction if_ = (IfInstruction)m[0].getInstruction();
    GOTO          g   = (GOTO)m[2].getInstruction();
    return (if_.getTarget() == m[3]) &&
           (g.getTarget() == m[4]);
  }
};
```

3 Concepts of Byte Code Engineering

Byte code engineering techniques can be used in many application areas. We think that the most interesting application area is the user-transparent adaptation of existing code at run-time. For example, a profiling tool can dynamically insert calls to analysis methods without affecting the semantics of the original code. There are also cases where one has to adapt classes for a certain environment when the source code is not available.

3.1 Load-time Reflection

The terms *reflection* or *meta-level programming* are generally used to denote systems that have the ability to reason about themselves, where certain aspects of the system are *reified* as *meta objects* [6]. On the one hand, separating meta-level from base-level code can separate concerns, e.g., security or persistence aspects can be addressed in the meta-program, thus enhancing base-level code reusability. On the other hand, reflection can provide more flexibility and better adaptability to changing environments. One prominent application area for reflection are programming tools like browsers, debuggers, and prototyping environments.

Java itself provides an API for structural *run-time* reflection [14], which can be used to retrieve type information about objects, to dynamically invoke methods, or to access fields of a given object. Barat [4], in contrast, is an approach for *compile-time* reflection.

Using class loaders and JAVACLASS, one can add an additional level of reflection with Smalltalk-like features [11] to Java: *load-time* reflection. Class loaders are responsible for loading class files from the file system or other resources and passing the byte code to the Virtual Machine [18]. Custom `ClassLoader` objects may be used to replace the standard procedure of loading a class. A given runtime system can then access JAVACLASS meta-level objects created at load-time and adapt them to its needs, or even create them *ad hoc* without a source file. This is an elegant way of extending the Java Virtual Machine without actually modifying it.

Similar to metaXa [12], for example, it is possible to reify events like method calls, instance creations or field accesses, but without the need to change the Virtual Machine. The reification of such an event can be implemented by simply enclosing or replacing the actual byte code instruction with invocations of runtime system methods.

A possible scenario is described in figure 3: During run-time the Virtual Machine requests a custom class loader to load a given class. Before the JVM actually sees the byte code, the class loader makes a “side-step” to create the requested class or to perform some transformation to an existing 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.

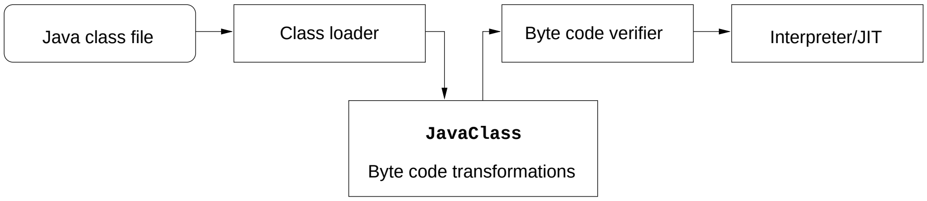

Fig. 3. Class loaders

3.2 Example III: “Reloading” classes

From the viewpoint of a run-time system it is sometimes desirable to reload a given class, i.e. to drop the old byte code and load a new implementation. This is not possible with the JVM, of course, but we can simulate the effect using byte code transformations. We illustrate this with the following simple example:

```
public class ReloadMe {
  public ReloadMe() { ... }
  public String foo(int bar) { ... }
}
```

We then implement a class loader that globally replaces all instance creations of this class with calls to an object factory. I.e., all statements like `o = new ReloadMe();` become `o = Factory.createReloadMe();` (on the byte code level). The factory is initialized with the class object of `ReloadMe` and uses it as a template for object creation (if the constructor takes arguments we have to use `Constructor` objects of the Java Reflection API as templates).

```
public class Factory {
  Class reload_me = ReloadMe.class;
  ...
  public static ReloadMe createReloadMe() {
    return (ReloadMe)reload_me.newInstance();
  }
}
```

When the run-time system decides to load a new implementation of `ReloadMe` it simply replaces the template class of the factory with a new class that extends the original one. This new class is generated using the JavaClass API and passed to the class loader:

```
JavaClass clazz = generateNew("ReloadMe");
Factory.reload_me = class_loader.load(clazz.getBytes());
```

Going even further we can reimplement the behavior of single methods, too, if we insert a *hook* at the start of every method whose behavior we want to adapt. The hook is inactive when it has a `null` value. Otherwise it redirects all incoming calls, i.e. we use delegation [10].

```
public String foo(int bar) {
  if(hook != null)
    return hook.foo(this, bar);
  ... // Old code becomes unreachable
}
```

It would be awkward to incorporate this explicit "reloading" and delegation code into the source of the original class. But on the meta (byte code) level we can easily implement user-transparent *behavorial* reflection features like the ones described.

Another useful example for the transformation of `new` statements may be to avoid costly thread creation operations by replacing them with factory methods that reuse old thread objects. This is also a good example of how to add optimizations on the meta level that are orthogonal to the original program and thus do not affect its semantics.

4 Conclusion and Future Work

In this paper we have presented the JavaClass framework, a general purpose tool for byte code engineering. It provides the developer with a completely object-oriented view upon byte code and allows him to conveniently analyze, transform or create classes. In combination with class loaders, JavaClass is a powerful tool to implement reflectional features on a meta level not visible to the user. There are many possible application areas ranging from class browsers, profilers, byte code-optimizers, and compilers to sophisticated run-time analysis tools and extensions to the Java language.

The framework has already proved its usefulness in several projects. We have been able, e.g., to implement the byte code-generating back end of a Java compiler within ten days. We plan to further enhance JavaClass with more support for load-time reflection and control flow analysis on the byte code level.

References

1. O. Agesen, S. N. Freund, and J. C. Mitchell. Adding Type Parameterization to the Java Language. In *Proceedings OOPSLA'97*, Atlanta, GA, 1997.
2. A. Aho, R. Sethi, and J. Ullman. *COMPILERS Principles, Techniques and Tools*. Addison-Wesley, 1985.
3. B. Bokowski and M. Dahm. Poor Man's Genericity for Java. In Clemens Cap, editor, *Proceedings JIT'98*. Springer, 1998.
4. B. Bokowski and A. Spiegel. Barat – A Front-End for Java. Technical report, Freie Universität Berlin, 1998.
5. Per Bothner. *The gnu.bytecode package*. http://www.cygnus.com/~bothner/gnu.bytecode/, 1998.
6. Gerald Brose. Reflection in Java, CORBA and JacORB. In Clemens Cap, editor, *Proceedings JIT'98*. Springer, 1998.
7. Geoff Cohen, Jeff Chase, and David Kaminsky. Automatic Program Transformation with JOIE. In *Proceedings USENIX Annual Technical Symposium*, 1998.
8. Pascal Costanza. *The ClassFilters package*. Universität Bonn, http://www.cs.uni-bonn.de/~costanza/ClassFilters/, 1998.
9. M. Dahm. Byte Code Engineering with the JavaClass API. Technical report, Freie Universität Berlin, 1998.
10. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
11. A. Goldberg and D.Robson. *Smalltalk – The Language*. Addison-Wesley, 1983.
12. Michael Golm and Jürgen Kleinöder. metaXa and the Future of Reflection. In *Workshop on Reflective Programming in C++ and Java*, 1998.
13. J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, 1996.
14. JavaSoft. *Reflection API*. http://java.sun.com/products/jdk/1.1/docs/guide/reflection/, 1998.
15. Ralph Keller and Urs Hölzle. Binary Component Adaptation. In Eric Jul, editor, *Proceedings ECOOP'98*. Springer, 1998.
16. 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.
17. Han Bok Lee and Benjamin G. Zorn. BIT: A Tool for Instrumenting Java Bytecodes. In *Proceedings USENIX Symposium on Internet Technologies and Systems*, 1998.
18. Sheng Lian and Gilad Bracha. Dynamic Class Loading in the Java Virtual Machine. In *Proceedings OOPSLA'98*, 1998.
19. Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1997.
20. J. Meyer and T. Downing. *Java Virtual Machine*. O'Reilly, 1997.
21. A.C. Myers, J. A. Bank, and B. Liskov. Parameterized Types for Java. In *Proceedings POPL'97*, Paris, France, 1997.
22. M. Thies and U. Kastens. Statische Analyse von Bibliotheken als Grundlage dynamischer Optimierung. In Clemens Cap, editor, *Proceedings JIT'98*. Springer, 1998.