Fachhochschule Aargau
Nordwestschweiz

F H A

Bytecode
Inspection & Transformation

Fachhochschule Nordwestschweiz FHA
University of Applied Science Aargau

Study	Computer Science
Field of study	Enterprise Computing
Course	Seminar Enterprise Computing
Lecturer	Prof. Dr. Dominik Gruntz

Author	Samuel Gerber s.gerber@stud.fh-aargau.ch

Date	December 11, 2003

Abstract

Manipulating Java bytecode offers some interesting possibilites to Java programmers. This report explains how Java bytecode can be analyzed and manipulated, by using two existing libraries (BCEL [2] and Javassist [1]). It also contains some suggestions where bytecode transformation can be used.

Contents

0 Introduction **3**

1 Java Bytecode **4**
1.1 Classfile Format 4
1.1.1 A Depper Look into the Constantpool 4
1.2 Bytecode Instruction Set 6
1.3 Example 6

2 Bytecode Manipulation Libraries **9**
2.1 Overview BCEL 9
2.1.1 General Approach 9
2.1.2 Utilities 9
2.2 Overview Javassist 10
2.2.1 General Apporoch 10
2.2.2 Utilities 10
2.2.3 Javassist Built-in Compiler 11
2.2.4 Javassist Classloading 11
2.3 Classfile Representation 12
2.4 Bytecode Access Classes 13
2.5 Conclusion 13

3 Transforming Bytcode **14**
3.1 Static Transformation 14
3.2 Dynamic Transformation 14
3.2.1 Transformation at Loadtime 14
3.2.2 Alter Classes during Runtime 14

4 Areas of Application **16**

5 Examples **17**
5.1 Simple BCEL Modification 17
5.2 Simple Javassist Modification 18
5.3 BCEL InstructionFactory 19
5.4 BCEL Classloader 21
5.5 Javassist Classloader 23
5.6 Interpreter 24

0 Introduction

Bytecode manipulation is a rich topic. During writing this report I realized that one could go depper and depper into specific topics.

This report gives a short introduction into the bytecode instruction and how the Java Virtual Machine works. Afterwards two bytecode manipulation librarys, that can be used to inspect and modify bytecode, are introduced. In the section Bytecode Transformation it is shown how these libraries can be used to change bytecode. It follows a short section, that gives some ideas how bytecode manipulation can be used. At the end is a large section with examples that make use of the previous introduced libraries.

To manipulate bytecode a good knowledge of the Java Virtual Machine is very helpfull, I recommend everyone to have a look at the Java Virtual Machine Specifications [3], especially the section about bytecode instructions and the one about the classfile format is very helpful. After going through some bytecode instructions with looking up the instructions in the Java Virtual Machine Specifications[1], understanding bytecode is quite easy.

[1] http://java.sun.com/docs/books/vmspec/html/VMSpecTOC.doc.html

1 Java Bytecode

To understand how to work with bytecode[2], we need to know a little bit about the java bytecode. I assume that the reader knows conceptually how the Java Virtual Machine works.

1.1 Classfile Format

The classfile contains simplified seven sections (see figure 1 on page 5):

- Header: Contains the Magic Number OxCAFEBABE and the Version Number.
- Constantpool: Contains all used constants. The first entry is the own class followed by the superclasses, then the constants, and at the end is the name of the source file. The constantpool holds the following types of constants: strings, integers, floats, longs, doubles and references to methods, fields and classes. References are entries in the constantpool which points onto another entry in the clonstantpool, these entries can be different for each instance.
- Acces rights: The acces rights of the class.
- Implemented Interfaces: All implemented Interfaces.
- Fields: A list of fields, normally an index into the constantpool.
- Methods: A list of methods, containing the instructions of each method.
- Class Attributes: Contains the source filename and other user definable attributes which are ignored by the JVM.

1.1.1 A Depper Look into the Constantpool

The biggest part of an average classfile is the constantpool with 61% followed by the methods with 31% [2]. Figure 2 on page 7 shows the constantpool of the class used in the Example 1.3 on page 6.
An entry in the constantpool has the general format:

- tag (1 byte)
- info (size depends on tag)

The tag specifies the type of the entry, the info may be further indicies into the constantpool or a value (double, string, integer, float, long, double, Utf8). Have a look at the entry at index 29 in figure 2, which is a method reference. This method reference contains two further indicies into the constantpool. The first

[2] A detailed report about the Java classfile format can be found on http://www.cs.fh-aargau.ch/~gruntz/courses/sem/ws03/

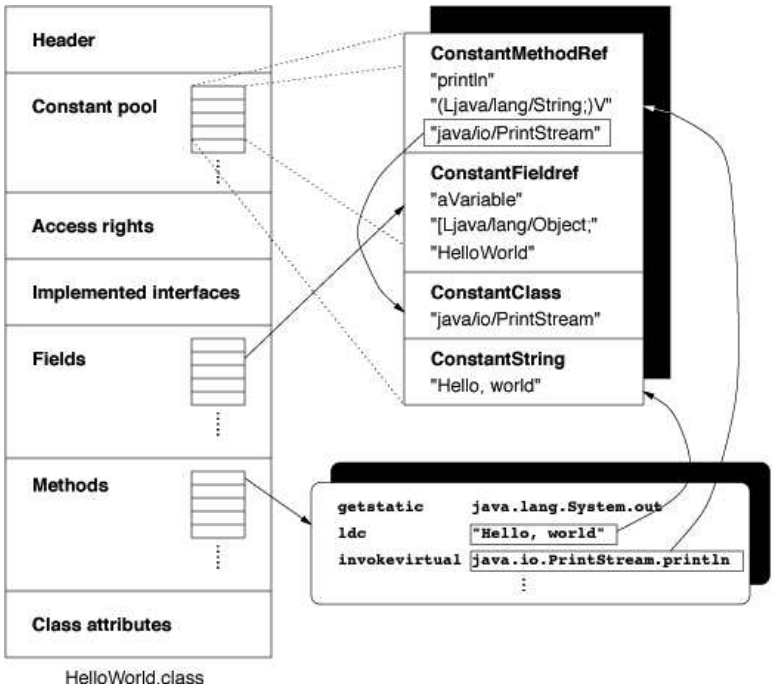

Figure 1: Classfile Structure

uses

is an entry of the type Class, the second is a NameAndType entry. Through these entries the method is exactly specified.

1.2 Bytecode Instruction Set

The Java bytecode instruction set can be grouped into eight diffrent types:

- Stack operations: Pushing constants on the stack using the (*ldc*) command or shortcut instructions (*iconst_0*) where the operand is implicitly encoded in the operation.
- Arithmetic operations: There are diffrent operations for the diffrent types in Java. For example *iadd* adds two integers and pushes the result on the stack.
- Control flow: Instructions to code loops, branches (*goto, iif*) and try-catch (*jsr, ret*).
- Load and store operations: For loading and storing local variables (*istore, iastore*) .
- Field access: Retrieving from and storing into fields of a class (*getfield, putfield, getstatic, putstatic*).
- Method invocation: Calling methods (*invokevirtual, invokestatic, invokespecial*).
- Object allocation: Allocating objects (*new, newarray, anewarray, amultinewarray*).
- Conversion and type checking: Cheking and converting basic types and instances (*i2f, checkcast, instanceof*).

A list of all instructions with detailed description can be found in the JVM-Specification [3].

1.3 Example

Let's look at the following class[3]:

```
1public class Example {
2    private int number=10;
3    private Foo myObject=new Foo();
4
5    public void callfoo(){
6        //How does this line look like in bytecode instructions?
7        int n=myObject.foo(number);
```

[3]The constantpool in figure 2 corresponds to this class

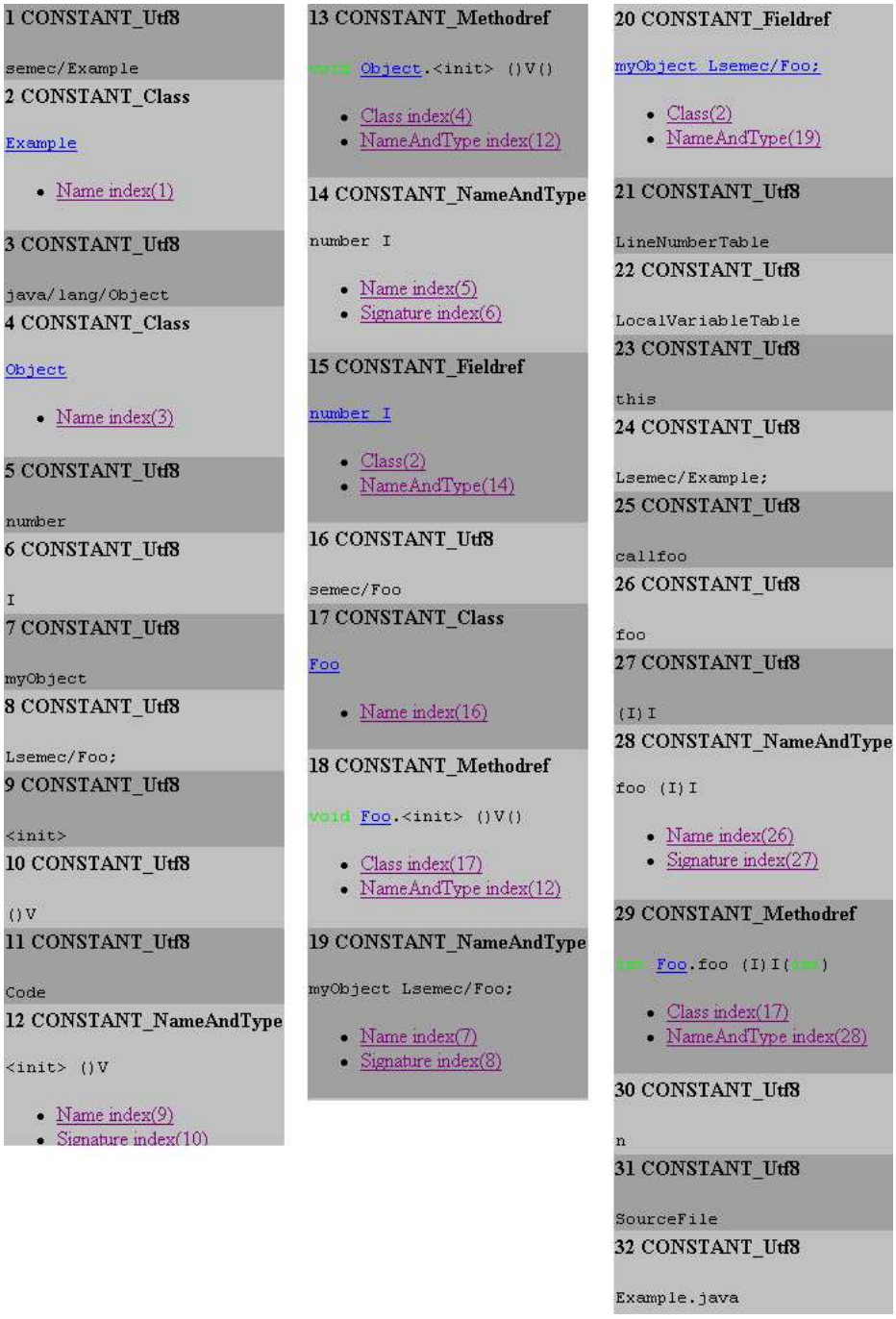

Figure 2: Constantpool

```
8    }
9}
```

We now will try to get out the instructions which are necessary for the method call on line seven. First we need to push the object reference of the method *foo* onto the stack. The instruction **getfield** loads a field from an instance onto the stack. This instruction expects that on the stack lies a object reference from which the instance to get the field can be resolved. So we need to load the own object reference (*this* pointer) onto the stack. This is done with the instruction **aload_0**. The **aload_0** instruction pushes the local variable in slot zero from the method frame (each method has it's own section in the memory, with local variables) of the current running method onto the stack [4]. And then we call **getfield myObject**, where myObject is an index into the constantpool (e.g. 20 see figure 2) where the field (in this case an object reference) can be grabbed and pushed onto the stack. Afterwards the same has to be done with the parameter of the method foo. So we use again the instructions **aload_0** and **getfield number** (e.g number = 15) to push the parameter onto the stack. Finally we call the method foo with **invokevirtual foo** (e.g. foo=25). This instruction creates an new frame on the stack for the invoked method and sets the local variable zero to the object reference (myObject) which lies on the stack and as local variable one the parameter we passed (number). Then the execution continues on the first instruction of the foo method.
At the end we get the instructions as follows:

1. aload_0
2. getfield myObject
3. aload_0
4. getfield number
5. invokevirtual foo

[4] In slot zero of a methodframe (except static methods) lies always the objet reference to itself because the first parameter passed to an instance method is this object reference (the *this* pointer)

2 Bytecode Manipulation Libraries

Because it is really not easy to manipulate bytecode with hexadecimal numbers directly, I will present two libraries which can be used to transform bytecode. There are several others, but these are the ones I used to gather information about bytecode modification.

2.1 Overview BCEL

BCEL [2] gives acces to the whole classfile you can modify the constantpool, methods, fields. But it has to be done at quite a low level. For example you can insert single instruction anywhere into the code, but you are responsible that afterwards the stack size is correct.

2.1.1 General Approach

If you want to insert bytecode into an existing class with BCEL you do normally the following steps[5]:

1. Create a *MethodGen* object from the method you want to modify.
2. Obtain an *InstructionList* object from the bytecode you want to alter (from the *MethodGen* object).
3. Insert instructions at a specific point into the *InstructionList*. The point is defined through an *InstructionHandle* and for each bytecode instruction there exists a class with the name of the memnonic of the instruction or a similiar instruction which is translated into bytecode instructions for you (e.g. PUSH).
4. Add the modified bytecode to the class (by setting the original method to *MethodGen.getMehod()*.

Almost the same works with the constantpool. Instead of inserting bytecode you insert constantpool entries. You can add for example a *Method*. This results in a MethodRef entry and all subentries in the constantpool are done for you by BCEL.

2.1.2 Utilities

BCEL offers a lot of useful utilities like:

- Dump a class into a html file so you can comfortably look at the class file structure (the constantpool in figure 2 was generated with BCEL).
- Generate source code, from a class, that create the class using the BCEL API.

[5] see also the example in section 5.1 on page 17

- A template classloader[6]. You only need to extend it and overwrite the method *JavaClass modifyClass(JavaClass clazz)* to obtain the loaded classes. Then you can modify the class and return it, and the modified class gets loaded.
- A factory class which generates often used bytecode sequences[7].

2.2 Overview Javassist

Javassist [1] is very simillar to BCEL. You also have acces to the whole class file structure.

2.2.1 General Apporoch

In Javassist you have two possibilties to change the bytecode.

First you can, equally to BCEL, modify bytecode at a low level. Instead of an *InstructionList* you have a *Bytecode* class in Javasssist where you can add your instructions through methods called after bytecode instructions (e.g. *bytecode.addLdc(...)* for the instruction ldc) or you can add bytecode instructions through memnonics (static integers for each bytecode instructions) with the *bytecode.add(int memnonic)* method if the instructions needs some parameters you can append them with the *bytecode.addIndex(int index)* method.

Second you can insert a strin containig javcode that gets compiled and inserted into the bytecode, more on that in section 2.2.3[8].

2.2.2 Utilities

Javassist also offers lot of utilities, that help to modify bytecode. Here is list of some of them:

- A built-in compiler which lets you insert source code instead of bytecode instructions.
- A class (*CodeConverter*) through that you can modify occurences of existing bytecode instructions (e.g. *codeconverter.replaceNew(...), codeconverter.redirectFieldAcces(...)*).
- A mechanism to modify classes at loadtime (see section 2.2.4.

[6] see also the example in section 5.4 on page 21
[7] see also the example in section 5.3 on page 19
[8] see also the example in section 5.2 on page 18

2.2.3 Javassist Built-in Compiler

A nice feature of javassist is that you can insert source code at the beginning or the end of a method. The source gets compiled by a built-in compiler into bytecode and inserted into the method. This is very comfortable because you do not have to think about stacksize, references in the constantpool and so on. For example if you insert *System.out.println("Hello ´´);* everything is done for you, you do not need to add a *System.out* object refernce onto the constantpool nor a *println* method reference. But the compiler has its limitations:

- Inner classes cannot be created.
- Labeled *break* or *continue* statements are not supported.
- The *switch* statement is not provided.
- You cannot *synchronize* anything.
- The *finally* clause is not supported.
- The compiler has problems with dispatching equal named methods.
- Array initializers are not supported.
- The *object.class* notation is not supported.
- You can insert one code statement or several statements inside a block ({...}).
- It is easier for the compiler if the separator between classes and statical field or methods is a # instead of a . .

Some expressions starting with $ in the code string have special meanings [9], see table 1 for an overview.

2.2.4 Javassist Classloading

Javassist provides a mechanisem to modify classes at loadtime. It works a little bit diffrent from BCEL. You can attach a *Translator* onto a *Classpool* object. You now have to create a Javassist classoader (Javassist class: *Loader*) with the same *Classpool* object. Afterwards your application has to be started through the created classloader. Now the *Translator* gets notified each time a class gets loaded [10].

[9] detailed descripton is included with the javassist library
[10] see also the example in section 5.5 on page 23

$0, $1, $2, ...	Actual parameters
$args	An array of parameters. The type of $args is Object[].
$$	All actual parameters. For example, m($$) is equivalent to m($1,$2,...)
$cflow(...)	cflow variable
$r	The result type. It is used in a cast expression.
$w	The wrapper type. It is used in a cast expression.
$_	The resulting value
$sig	An array of java.lang.Class objects representing the formal parameter types.
$type	A java.lang.Class object representing the formal result type.
$class	A java.lang.Class object representing the class currently edited.

Table 1: Javassist $ expressions

2.3 Classfile Representation

The structure of the classes which BCEL and Javassist use to operate on the bytecode are very similar to java.reflect package. Table 2 shows the classes used in java.lang.reflect package and its representation in BCEL and Javassist. BCEL uses diffrent classes for describing classes (the classfile package) and modifing classes (the generic package). The classes that are used to alter and generate bytecode have usually the same name but the suffix *Gen* attached (e.g. Method - MethodGen).

java.lang.reflect	BCEL	javassist
Class	JavaClass	CtClass
Method	Method	CtMethod
Constructor	-	CtConstructor
Field	Field	CtField

Table 2: Classes

2.4 Bytecode Access Classes

The table 3 shows which classes in Javassist and BCEL are used to acces and modify parts of the classfiles structure.

classfile structure	BCEL	javassist
ConstantPool	ConstantPool	ConstPool
Code	InstructionList	ByteCode

Table 3: Bytecode Acces

2.5 Conclusion

The built-in Compiler from Javassist is easier to use than operating directly on the bytecode, but for direct access to the bytecode BCEL is the better solution.

3 Transforming Bytcode

In this section diffrent "types´´ of altering bytecode are described. It is divided into two parts, static transformation and dynamic transformation.

3.1 Static Transformation

The easiest way to modify bytecode is to read a class file from the harddisk, change it and then write it back to the harddisk. Both the Javassist and the BCEL library provide easy acces to the classfiles on disk. In section 5.1 on page 17 is an example that uses the BCEL library to modify a class statically and in section 5.2 on page 18 is one using the Javassist library.

3.2 Dynamic Transformation

Modifing classes dynamicallly is a little bit harder than statically. So why do we want to modify classes dynamically? There are several answers to this question:

- You need to know information about the runtime environment.
- A class should have diffrent behaviours on different classloaders (loaded through network, localy loaded, diffrent classloaders on a webserver).
- The user needs to pass input before the class can get modified (e.g. password for loading classes, different class behaviour dependent on user input).
- You don't want to modify the class permanently.

Dynamic transformation is achived by using classloaders.

3.2.1 Transformation at Loadtime

Altering bytecode at loadtime is achived by using classloaders[11] which modify the class. The example in section 5.4 on page 21 shows how you can add a functionality through a classloader by extending the BCEL classloader. There is also an example in section 5.5 on page 23 using the Javassist library. You could also write an absolutly new classloader and modify your classes there.

3.2.2 Alter Classes during Runtime

One could also think of modifing bytecode on execution, or generating classes during runtime. It would also be nice if you could modify a class not only at loadtime but during it is running. The problem there is, that you need to load a class which is already loaded an this is not possible (classloader throws an

[11] A detailed report about classloader can be found on http://www.cs.fh-aargau.ch/ gruntz/courses/sem/ws03/

exception). A solution would be too unload the class (if this is possible) and the reload the altered class. So you could change a programm during runtime. What are possible usages of modifing bytecode this way:

- With javassist you can easily implemented something like a interpreter which take java code strings an executes them during runtime (example in section 5.6 on page 24).
- Behaviour of an application can be modified during runtime.

4 Areas of Application

There are many usages of modifing bytecode. The list above shows some I thought of, there can be sure added a lot more.

- Adapting an existing application or library (where the sources are not provided) to your own purposes.
- Optimizing bytecode.
- Add new functionallity to the JVM.
- Bytecode manipulation could be used to implement an Aspect Orieneted Programming tool. Aspect Oriented Programming is that only aspects (logging, security, user-interaction, ...) are programmed. These aspects can afterwards be attached to a class or a method. One could do this with bytecode manipulation, or Sun could add a new language specification (probably through bytecode manipulation) which allows Aspect Oriented Programming.
- Diffrent behaviours of the same classes (e.g. someone using a slow computer getting another GUI than one using a fast computer).

5 Examples

In this section are some examples how you can use the BCEL and Javassist libraries to modify bytecode.
For better understanding of the classes used from Javassist and BCEL have a look at table 2 on page 12 and consult the Java Virtual Machine Specifications [3].
Both, BCEL[12] and Javassist[13], also have a well Java documentation, and the sources are also available.

5.1 Simple BCEL Modification

In this example a class is statically modified added by using BCEL. A class gets loaded from the harddisk, a simple method that say that it is get called is added, and afterwards the class is written back to the harddisk.

```
import java.io.IOException;
import org.apache.bcel.classfile.*;
import org.apache.bcel.generic.*;
import org.apache.bcel.*;

public class SimpleBCEL {

  static public void main(String args[]) {
    //Get class to modify from programm argument
    JavaClass mod = Repository.lookupClass(args[0]);

    //Class not found
    if (mod == null) {
      System.err.println("Could not get class " + args[0]);
      return;
    }

    //Create a generic class to modify
    ClassGen modClass = new ClassGen(mod);

    //Create a generic constantpool to modify
    ConstantPoolGen cp = modClass.getConstantPool();

    //Add references to print on System.out
    int out = cp.addFieldref("java.lang.System", "out",
                             "Ljava/io/PrintStream;");
    int println = cp.addMethodref("java.io.PrintStream", "println",
                                  "(Ljava/lang/String;)V");
```

[12] http://jakarta.apache.org/bcel/
[13] http://www.javassist.org

```
    //Create a new method which prints out its name
    InstructionList il = new InstructionList();
    MethodGen verbose =  new MethodGen(Constants.ACC_PUBLIC, Type.VOID,
                         Type.NO_ARGS, new String[] {} /*argument names*/,
                         "verbose", args[0], il, cp);

    il.append(new GETSTATIC(out));
    il.append(new PUSH(cp, "Hello I am method verbose"));
    il.append(new INVOKEVIRTUAL(println));
    il.append(new RETURN());

    //Set the stack to the appropriate size.
    //Can also be down by yourself with "setMaxStack(int size)"
    verbose.setMaxStack();
    verbose.setMaxLocals();
    modClass.addMethod(verbose.getMethod());
    il.dispose();

    try {
      //Write modified class
      JavaClass newClass = modClass.getJavaClass();
      newClass.dump(args[0] + ".class");
      System.out.println("Class " + args[0] + " modified");
    }
    catch (IOException e) {
      e.printStackTrace();
    }
  }
}
```

5.2 Simple Javassist Modification

In this example a class is statically modified, by using the Javassist built-in compiler. A Class gets loaded from the harddisk, a little bit a complexer method than before is added (it prints out how often it is called, this requires a new entry in the constantpool), and afterwards the class is written back to the harddisk.

```
import java.io.IOException;
import javassist.*;

public class JavassistAddMethod {
  public static void main(String args[]){
    try {
```

```
      CtClass mod=ClassPool.getDefault().get(args[0]);

      //Create counter variable
      CtField counter=new CtField(CtClass.intType, "counter", mod);
      mod.addField(counter);

      //Create method which count how often it gets called
      CtMethod count=new CtMethod(CtClass.voidType, "count",
                                  new CtClass[]{}, mod);
      count.setBody("System.out.println(counter++);");
      mod.addMethod(count);

      //Write class to disk
      mod.writeFile();

      System.out.println("Class "+args[0]+" modified." );
    }
    catch (NotFoundException e) {
      e.printStackTrace();
    } catch (CannotCompileException e) {
      e.printStackTrace();
    } catch (IOException e) {
      e.printStackTrace();
    }
  }
}
```

5.3 BCEL InstructionFactory

This example has the same result as the one before, but using BCEL instead of Javassist. This examples is provided to show the use of the BCEL *InstructionFactory*.

```
import java.io.IOException;
import org.apache.bcel.classfile.*;
import org.apache.bcel.generic.*;
import org.apache.bcel.*;

public class BCELaddMethod {

  static public void main(String args[]) {
    //Get class to modify from progtamm argument
    JavaClass mod = Repository.lookupClass(args[0]);
```

```
    //Class not found
    if (mod == null) {
      System.err.println("Could not get class " + args[0]);
      return;
    }

    //Create a generic class to modify
    ClassGen modClass = new ClassGen(mod);

    //Create a generic constantpool to modify
    ConstantPoolGen cp = modClass.getConstantPool();

    //Factory for easier creation of instructions
    InstructionFactory fac = new InstructionFactory(modClass, cp);

    //Add a field which counts how often the metod gets called
    modClass.addField(new FieldGen(Constants.ACC_PRIVATE, Type.INT,
                      "counter", cp).getField());

    //Add a references to print on System.out
    //Could also be don as above with "addField"
    int out = cp.addFieldref("java.lang.System", "out",
                             "Ljava/io/PrintStream;");
    int println = cp.addMethodref("java.io.PrintStream", "println",
                                  "(I)V");

    //Create a new Method which prints out how often it is called
    InstructionList countInstr = new InstructionList();
    MethodGen count =
new MethodGen(Constants.ACC_PUBLIC, Type.VOID,
                           Type.NO_ARGS, new String[] {}, "count",
                           args[0], countInstr, cp);

    countInstr.append(fac.createFieldAccess("java.lang.System", "out",
          new ObjectType("java.io.PrintStream"), Constants.GETSTATIC));
    countInstr.append(InstructionFactory.createLoad(Type.OBJECT, 0));
    countInstr.append(fac.createFieldAccess(args[0], "counter",
                      Type.INT, Constants.GETFIELD));
    countInstr.append(fac.createInvoke("java.io.PrintStream", "println",
                      Type.VOID, new Type[] { Type.INT },
                      Constants.INVOKEVIRTUAL));
    countInstr.append(InstructionFactory.createLoad(Type.OBJECT, 0));
    countInstr.append(InstructionConstants.DUP);
    countInstr.append(fac.createFieldAccess(args[0], "counter", Type.INT,
                      Constants.GETFIELD));
    countInstr.append(new PUSH(cp, 1));
```

```
    countInstr.append(InstructionConstants.IADD);
    countInstr.append(fac.createFieldAccess(args[0], "counter",
                      Type.INT, Constants.PUTFIELD));
    countInstr.append(InstructionFactory.createReturn(Type.VOID));

    count.setMaxStack();
    count.setMaxLocals();
    modClass.addMethod(count.getMethod());
    countInstr.dispose();

    try {
      //write modified class
      JavaClass newClass=modClass.getJavaClass();
      newClass.dump(args[0] + ".class");
      System.out.println("Class " + args[0] + " modified");
    }
    catch (IOException e) {
      e.printStackTrace();
    }
  }
}
```

5.4 BCEL Classloader

The following example shows how you can add a logging function on a class at loadtime with BCEL. The example extends the BCEL classloader. After loading a class with this classloader each method prints its name and signature onto *System.out*.

```
import org.apache.bcel.classfile.*;
import org.apache.bcel.generic.*;
import org.apache.bcel.util.ClassLoader;

public class LogMethodsClassLoaderBCEL extends ClassLoader {

  //index into the constantpool to System.out
  private static int out;
  //index into the constantpool to PrintStream.println
  private static int println;

  protected JavaClass modifyClass(JavaClass clazz) {
    //Get the constantpool of the currently loaded class
    ConstantPoolGen cp = new ConstantPoolGen(clazz.getConstantPool());
```

```
//Add the references to System.out and PrintStream.println
out = cp.addFieldref("java.lang.System", "out",
                     "Ljava/io/PrintStream;");
println = cp.addMethodref("java.io.PrintStream", "println",
                          "(Ljava/lang/String;)V");

//Go through each method of the class
Method methods[] = clazz.getMethods();
for (int i = 0; i < methods.length; i++) {
   //Create the message
   String name = methods[i].getName();
   int flags = methods[i].getAccessFlags();
   String mesg = "BCEL: Method: "
   mesg+= Utility.methodSignatureToString(
                 methods[i].getSignature(), name,
                 Utility.accessToString(flags))
   mesg+= " from Class " +clazz.getClassName() +" called";

  //Create the instructions (System.out.println(mesg);)
  InstructionList logPatch = new InstructionList();
  logPatch.append(new GETSTATIC(out));
  logPatch.append(new PUSH(cp, mesg));
  logPatch.append(new INVOKEVIRTUAL(println));

  //Get the bytecode of the current method
  MethodGen mg =
           new MethodGen(methods[i], clazz.getClassName(), cp);
  InstructionList il = mg.getInstructionList();
  if(il==null)//Method has no body
     break;

  InstructionHandle[] ihs = il.getInstructionHandles();
  //Check if method is constructor
  if (name.equals("<init>")) {
     // First let the super or other constructor be called
     for (int j = 1; j < ihs.length; j++) {
         if (ihs[j].getInstruction() instanceof INVOKESPECIAL) {
            //Append log instructions
            il.append(ihs[j], logPatch);
            break;
         }
     }
   }
   else {
      //Append log instructions
      il.insert(ihs[0], logPatch);
```

```
      }

      //Check if method has enough stack for the System.out.println call
      if (methods[i].getCode().getMaxStack() < 2){
         mg.setMaxStack(2);
      }

      //Set the method to the modified method.
      methods[i] = mg.getMethod();
      il.dispose();
    }

    //Set the constantpool to the changed constantpool
    clazz.setConstantPool(cp.getFinalConstantPool());
    //return the alterd class which now get loaded
    return clazz;
  }
}
```

5.5 Javassist Classloader

The following code shows how to add the same logging functionallity as in the example before, using the javassist API.

```
import javassist.*;

public class LogMethodsJavassist implements Translator {

  //this method gets called when the Translator
  //is attched to a classpool
  public void start(ClassPool cp)
         throws NotFoundException, CannotCompileException {
  }

  //this method gets called when a class (called cname) gets loaded
  public void onWrite(ClassPool cp, String cname)
         throws NotFoundException, CannotCompileException {
    //Get the class
    CtClass modClass = cp.get(cname);
    modClass.defrost();
    CtMethod methods[] = modClass.getMethods();
    for (int i = 0; i < methods.length; i++) {
      try {
```

```
        //Java code to insert into method.
        String javaCode="System.out.println(\"Javassist: Method: "
        + methods[i].getName()
        + " from Class "
        + modClass.getName()
        + " called\");";

        methods[i].insertBefore(javaCode);
      }
      catch (CannotCompileException ex) {
        System.err.println(ex);
      }
    }
  }

  public static void main(String args[]){
    //Create the Translator and attach it to the Classpool
    Translator t = new LogMethodsJavassist();
    ClassPool pool = ClassPool.getDefault(t);

    //Create a javassit classloader with the pool our Translator
    //is attached to
    Loader cl = new Loader(pool);
    try {
      //Start an applivation through the the classloader so
  //that all classes get loaded via this classloader
      cl.run("semec.HelloWorld", args);
    }
    catch (Throwable e) {
      e.printStackTrace();
    }
  }

}
```

5.6 Interpreter

In this example an Interpreter is created using the Javassist library. The interpreter compiles java code into a for each input newly generated class (because a class with the same name can only be loaded once). Afterwards an instance is created and the inserted bytecode gets executed.

```
import java.io.*;
import javassist.*;
```

```
public class Interpreter {
  public static void main(String args[]){
    try {
      CtClass template=ClassPool.getDefault().get("Template");
      BufferedReader in =
         new BufferedReader(new InputStreamReader(System.in));

      String name="copy";
      for(int i=0; ;i++){
        System.out.println("Enter your code or hit enter to exit:");
        String code;
        try {
          code = in.readLine();
          if(code.length()==0) return;
          //The class needs to be defrosted to be changeable again
          template.defrost();
          CtClass copy = template;
          //For each input there is a  new class generated
          copy.setName(name+i);
          //Set code from input into the method
          CtMethod m=copy.getDeclaredMethod("execute");
          m.setBody("{"+code+"}");
          Execute command=(Execute)copy.toClass().newInstance();
          command.execute();
        }
        catch (IOException e) {
          e.printStackTrace();
        } catch (CannotCompileException e) {
          e.printStackTrace();
        } catch (InstantiationException e) {
          e.printStackTrace();
        } catch (IllegalAccessException e) {
          e.printStackTrace();
        }

      }
    }
    catch (NotFoundException e) {
      e.printStackTrace();
    }
  }
}


//Interface is needed to create instances of
```

```
//the generated class
public interface Execute {
  public void execute();
}


//Template class to add the code from input to
public class Template implements Execute {
  public void execute() {}
}
```

References

[1] S. Chiba. Javassist a reflection-based programming wizard for java. Technical report, Institute of Information Science and Electronics University of Tsukuba and Japan Science and Technology Corporation, 1998. http://www.javassist.org.

[2] M. Dahm. Bytecode engeeniering with the bcel api. Technical report, Freie Universität Berlin - Institut für Informatik, 2001. http://bcel.sourceforge.net.

[3] F. Y. Tim Lindholm. *The JavaTM Virtual Machine Specification*. Addison-Wesley, second edition, 1996. http://java.sun.com/docs/books/vmspec/html/VMSpecTOC.doc.html.