Using the ASM framework to implement common Java bytecode transformation patterns

Eugene Kuleshov, *eu@javatx.org*

ABSTRACT

Most AOP frameworks targeting the Java platform use a bytecode weaving approach as it is currently considered the most practical solution. It allows applying cross-cutting concerns to Java applications when source code is not available, is portable and works on existing JVMs, in comparison to VM-level AOP implementations.

Load-time bytecode weaving (LTW), which happens right before the application code is loaded into the Java VM, significantly simplifies development environment, but also raises the bar for the performance and memory requirements for these transformations. Such requirements directly apply to the toolkit that will be used to perform these transformations. In this paper, we examine how Java bytecode transformations, typical for AOP implementations can be done efficiently, using the ASM[1] bytecode manipulation framework [1].

Transformations used by general-use AOP frameworks and similar applications can be categorized, and the common patterns can be reused to implement specific transformations. These patterns can also be combined to implement more complex transformations.

General Terms

Languages, Design, Performance, Experimentation, Standardization

Keywords

Aspect-Oriented Programming, Java, bytecode, weaving, ASM

1. INTRODUCTION

The ASM bytecode framework was designed at France Telecom R&D by Eric Bruneton, Romain Lenglet and Thierry Coupaye [2]. After evaluating several existing frameworks, including BCEL [3], Serp [4] and JOIE [5], they designed a more efficient approach, providing better performance and memory foot print. Today ASM is used in many applications and has become the de-facto standard for bytecode processing.

The main idea of the ASM API [6] is not to use an object representation of the bytecode. This made it possible expressing the same transformations using only a few classes comparing to approximately 80 classes in Serp and 270 in BCEL API. Those frameworks create lots of objects during class deserialization, which takes a lot of time and memory. ASM avoids this overhead to keep transformation fast and to use very little memory. This is done by using the Visitor design pattern [7], without representing the visited tree with objects. Visitors can change call chains and therefore transform the visited code. Using the Adapter design pattern [7] visitors can be chained in order to implement complex transformation from smaller building blocks. A similar approach is also used in the SAX API for XML processing [8].

ASM hides all the complexity of the serialization and deserialization of the class bytecode, using the following techniques:

- Automatic management of the class constant pool, therefore the user does not have to manipulate indexes of these constants.
- Automatic management of the class structure, including annotations, fields, methods, method code and other standard bytecode attributes.
- Labels are used to manage instruction addresses, so it is easy to insert new code in between existing instructions
- Computation of maximum stack and local variables, as well as StackMapFrames

The event-based interaction between event producers and event consumers is defined by several interfaces: *ClassVisitor*, *FieldVisitor*, *MethodVisitor*, and *AnnotationVisitor*. Event producers, like *ClassReader* fire *visit*()* calls to those interfaces. On the other hand, event receivers like writers (*ClassWriter*, *FieldWriter*, *MethodWriter*, and *AnnotationWriter*), adapters (*ClassAdapter* and *MethodAdapter*) or classes from the *tree* package (*ClassNode*, *MethodNode*, etc) implementing those interfaces.

The following code demonstrates how this looks from the developer's point of view:

```
ClassReader cr = new ClasReader(bytecode);
ClassWriter cw = new ClassWiter(cr,
   ClassWriter.COMPUTE_MAXS |
   ClassWriter.COMPUTE_FRAMES);
FooClassAdapter cv = new FooClassAdapter(cw);
cr.accept(cv, 0);
```

Here *ClassReader* reads the *bytecode*. On *accept()* method call *ClassReader* fires all visiting events corresponding to the *bytecode* structure. *FooClassAdapter* will receive those events and can change the event flow before passing them to the *ClassWriter*. Once *ClassWriter* receive all the events it will have transformed bytecode. You may notice that the *ClassReader* instance is passed to the *ClassWriter* that allows performance optimizations based on the assumption that the transformations mostly add new code.

The following sections will show a number of practical examples that should help you to better understanding of the ASM framework.

[1] The ASM name does not mean anything: it is just a reference to the keyword in C which allows some functions to be implemented in assembly language.

2. Accessing class data

The visitor-based approach allows capturing class data incrementally, collecting only the information required for specific use case without creating and destroying lots of short-lived objects. Incremental processing allows decisions such as whether or not part of the class needs to be transformed, skipping parsing of class parts that don't need to be transformed.

ASM provide very simple API to support this. First of all there are several bit-mask flags in the *ClassReader.accept()* method:

- SKIP_DEBUG – Used to ignore debug info, such as source file, line number and variable info.
- SKIP_FRAMES – Used to ignore StackMapTable information used for Java 6 split bytecode verifier.
- EXPAND_FRAMES – Expand the StackMapTable data, allowing visitor to have information on types of all local variables and current stack slots.
- SKIP_CODE – Exclude code of all methods from visiting, while still passing trough method's and parameter's attributes and annotations.

Additionally, the visitor can decide to skip the corresponding bytecode section it is not interested in. In order to do that, *visitField()*, *visitMethod()* and *visitAnnotation()* methods that returns nested visitor can return null. That will indicate to the bytecode producer to skip the corresponding class element.

When there is no need to read class data, but just the class hierarchy, *ClassReader* provides shortcut methods *getSuperName()* and *getInterfaces()* to read super class and implemented interfaces, respectively.

It is also possible use the *tree* package of ASM framework to read the entire class or selected method in memory and change its structures using DOM-like API. For example:

```
ClassReader cr = new ClassReader(source);
ClassWriter cw = new ClassWriter();
ClassAdapter ca = new ClassAdapter(cw) {
  public MethodVisitor visitMethod(int access,
      String name, String desc,
      String signature, String[] exceptions) {
    final MethodVisitor mv =
        super.visitMethod(access,
          name, desc, signature, exceptions);
    MethodNode mn = new MethodNode(access,
        name, desc, signature, exceptions) {
      public void visitEnd() {
        // transform/analyze this method DOM
        accept(mv);
      }
    };
    return mn;
  }
};
cr.accept(ca, 0);
```

The above code will basically suspend passing method events to the next visitor in the chain (i.e. *ClassWriter* in this case) until the *visitEnd()* event. When *visitEnd()* is passed, the *MethodNode* instance will contain the entire method data. At this point the method data can be transformed and only after that, finally passed to the next visitor using the *MethodNode.accept()* method.

It is worth mentioning that ASM provides several basic Data Flow Analysis algorithms that work on the *tree* package and can be used to analyze selected methods in isolation. The result of such analysis can be also used for inter-method analysis.

Combining those features it is possible to retrieve required class information with very controlled memory overhead by making decisions on the required transformations at load time.

In the next sections we will see concrete examples of the bytecode transformations typical for AOP.

3. COMMON TRANSFORMATIONS

AOP frameworks that are using load time transformations usually build their own high-level abstractions about how application code needs to be transformed. Those abstractions can be decomposed into smaller building blocks that can be chained together to achieve the required transformation. Here are most common use cases:

- Class Transformations
  - Introduce Interface
  - Add a New Field
  - Add a New Method
  - Replace Method Body
  - Merge Two Classes into One
- Method Transformations
  - Insert Code before Method, Constructor or Static Initializer Execution
  - Insert Code before Method Exit
  - Replace Field Access
  - Replace Method Call
  - Inline Method

3.1 Class Transformations

3.1.1 Introducing Interface

This transformation only changes the class information about the implemented interfaces. We can use simple *ClassAdapter* for this:

```
public class InterfaceAdder extends ClassAdapter {
  private Set newInterfaces;

  public InterfacesAdder(ClassVisitor cv,
      Set newInterfaces) {
    super(cv);
    this.newInterfaces = newInterfaces;
  }

  public void visit(int version, int access,
      String name, String signature,
      String superName, String[] interfaces) {
    Set ints = new HashSet(newInterfaces);
    ints.addAll(Arrays.asList(interfaces));
    cv.visit(version, access, name, signature,
        superName, (String[]) ints.toArray());
  }
}
```

Note, the actual methods required to implement the introduced interfaces must be added with a separate transformation, which will be discussed shortly.

3.1.2 Adding a New Field

Adding new fields to an existing class is a common transformation. Usually it is used to enrich class state with additional data. Here is a *ClassAdapter* that adds new field:

```
public class FieldAdder extends ClassAdapter {
  private final FieldNode fn;

  public FieldAdder(ClassVisitor cv,
      FieldNode fn) {
    super(cv);
    this.fn = fn;
  }

  public void visitEnd() {
    fn.accept(cv);
    super.visitEnd();
  }
}
```

The above adapter introduces new events in the *visitEnd()* method, which is called at the end of visiting class data. So, the field will be added after existing fields, which is the safest way, just in case some other code relies on the field order. In the above code new events are fired by *FieldNode* instance on *accept()* call. This allows reusing field definition, including annotations or even custom attributes, in case if same field need to be added to many classes.

Note that non-static field initialization should be done separately, for example by adding code after constructor invocation, or for static fields, at the end of the static block. Those transformations will be discussed later in the paper.

3.1.3 Adding a New Method

New methods can be added for implementing newly introduced interface or for internal needs. The class adapter for this can also use *visitEnd()* method to add new method to the class.

```
public class MethodAdder extends ClassAdapter {
  private int mAccess;
  private String mName;
  private String mDesc;
  private String mSignature;
  private String[] mExceptions;

  public MethodAdder(ClassVisitor cv,
      int mthAccess, String mthName,
      String mthDesc, String mthSignature,
      String[] mthExceptions) {
    super(cv);
    this.mAccess = mthAccess;
    this.mName = mthName;
    this.mDesc = mthDesc;
    this.mSignature = mthSignature;
    this.mExceptions = mthExceptions;
  }

  public void visitEnd() {
    MethodVisitor mv = cv.visitMethod(mAccess,
        mName, mDesc, mSignature, mExceptions);
    // create method body
    mv.visitMaxs(0, 0);
    mv.visitEnd();
    super.visitEnd();
  }
}
```

3.1.4 Replace Method Body

This transformation is a variation of the transformation for adding a new method. In this case, the difference is that new method generation is triggered by the *visitMethod()* call for a method that needs to be renamed, as opposed to *visitEnd()* event:

```
public class MethodReplacer extends ClassAdapter {
  private String mname;
  private String mdesc;
  private String cname;

  public MethodReplacer(ClassVisitor cv,
      String mname, String mdesc) {
    super(cv);
    this.mname = mname;
    this.mdesc = mdesc;
  }

  public void visit(int version, int access,
      String name, String signature,
      String superName, String[] interfaces) {
    this.cname = name;
    cv.visit(version, access, name,
        signature, superName, interfaces);
  }

  public MethodVisitor visitMethod(int access,
      String name, String desc,
      String signature, String[] exceptions) {
    String newName = name;
    if(name.equals(mname) && desc.equals(mdesc)) {
      newName = "orig$" + name;
      generateNewBody(access, desc, signature,
          exceptions, name, newName);
    }
    return super.visitMethod(access, newName,
        desc, signature, exceptions);
  }

  private void generateNewBody(int access,
      String desc, String signature,
      String[] exceptions,
      String name, String newName) {
    MethodVisitor mv = cv.visitMethod(access,
        name, desc, signature, exceptions);
    // ...
    mv.visitCode();
    // call original metod
    mv.visitVarInsn(Opcodes.ALOAD, 0); // this
    mv.visitMethodInsn(access, cname, newName,
        desc);
    // ...
    mv.visitEnd();
  }
}
```

In the above *visitMethod()* checks method body should be replaced. If it should, it uses generates new method body and then replaces the method name and delegates to the next visitor in the chain. This way body of the original method will be saved in the newly created method.

3.1.5 Merging Two Classes into One

When adding the implementation of an introduced interface to transformed class, it is convenient to use an existing implementation of these methods from a separately compiled class. This second can be loaded into memory and used multiple times to copy class methods and fields.

```
ClassReader cr = new ClasReader(bytecode);
ClassNode cn = new ClassNode();
cr.accept(cn, 0);
```

Then we can pass loaded *ClassNode* instance to the merging adapter:

```
public class MergeAdapter extends ClassAdapter {
  private ClassNode cn;
  private String cname;

  public MergeAdapter(ClassVisitor cv,
    ClassNode cn) {
    super(cv);
    this.cn = cn;
  }

  public void visit(int version, int access,
      String name, String signature,
      String superName, String[] interfaces) {
    super.visit(version, access, name,
        signature, superName, interfaces);
    this.cname = name;
  }

  public void visitEnd() {
    for(Iterator it = cn.fields.iterator();
        it.hasNext();) {
      ((FieldNode) it.next()).accept(this);
    }
    for(Iterator it = cn.methods.iterator();
        it.hasNext();) {
      MethodNode mn = (MethodNode) it.next();
      String[] exceptions =
          new String[mn.exceptions.size()];
      mn.exceptions.toArray(exceptions);
      MethodVisitor mv =
        cv.visitMethod(
            mn.access, mn.name, mn.desc,
            mn.signature, exceptions);
      mn.instructions.resetLabels();
      mn.accept(new RemappingMethodAdapter(
          mn.access, mn.desc, mv,
          new Remapper(cname, cn.name)));
    }
    super.visitEnd();
  }
}
```

As you can see, fields and methods are copied from *ClassNode* into the visited class. Moreover, types referenced from the copied methods are remapped with the *RemappingMethodAdapter* to appropriately fit into the new target class.

3.2 Method Transformations

The major difference of the method transformations from the class-level transformations is that they require additional filtering at the class level. *ClassAdapter* can be used to make this decision and if the method happens to be interested, additional *MethodAdapter* can be inserted at the execution chain. For example:

```
public class FilterAdapter extends ClassAdapter {
  ...
  public MethodVisitor visitMethod(
      int acc, String name, String desc,
      String signature, String[] exceptions) {
    MethodVisitor mv = cv.visitMethod(acc, name,
        desc, signature, exceptions);
    if(isFooMethod(name, desc)) {
      mv = new FooMethodAdapter(mv, acc,
          name, desc);
    }
    return mv;
  }
  ...
}
```

3.2.1 Common Issues

There are several common issues method transformers have to deal with. ASM provides the *commons* package that can make these issues easier to handle for the developer.

- Complexity of generating new code. For this issue *GeneratorAdapter* provides number of more high level building blocks that assist in the creation of common code idioms like boxing and unboxing from primitive types into object wrappers, loading method parameters, and a few others.
- Inserting new local variables in the middle of a method being visited is handled by *LocalVariablesSorter*, which can transparently rename method variables after new variables are inserted in the middle of the visited method.
- Dealing with the data flow and type system. There is, of course, no single solution for all the use cases, but ASM provides number of primitives that hide this complexity from the developer. One such primitive is *AdviceAdapter* that provides a convenient way to detect the right place in the bytecode to insert new code at the beginning of method execution and before exiting of the method.

Let's look at more concrete use cases of the method transformations.

3.2.2 Insert Code before Method, Constructor or Static Initializer Execution

As already mentioned above, this transformation is greatly simplified by subclassing *AdviceAdapter*. The developer implements *onMethodEnter()* method called by the *AdviceAdapter* at the appropriate time. This implementation should not change stack and it could use methods inherited from *LocalVariablesSorter* for adding new local variables.

```
class EnteringAdapter extends AdviceAdapter {
  private String name;
  private int timeVar;
  private Label timeVarStart = new Label();
  private Label timeVarEnd = new Label();

  public PrintEnteringAdapter(MethodVisitor mv,
     int acc, String name, String desc) {
    super(mv, acc, name, desc);
    this.name = name;
  }


  protected void onMethodEnter() {
    visitLabel(timeVarStart);
    int timeVar = newLocal(Type.getType("J"));
    visitLocalVariable("timeVar", "J", null,
        timeVarStart, timeVarEnd, timeVar);
    super.visitFieldInsn(GETSTATIC,
        "java/lang/System", "err",
        "Ljava/io/PrintStream;");
    super.visitLdcInsn("Entering " +  name);
    super.visitMethodInsn(INVOKEVIRTUAL,
        "java/io/PrintStream", "println",
        "(Ljava/lang/String;)V");
  }
```

```
  public void visitMaxs(int stack, int locals) {
    visitLabel(timeVarEnd);
    super.visitMaxs(stack, locals);
  }
}
```

Note how debug information is added for the *timeVar* variable via the call to *visitLocalVariable()*, which allows us to see the value of this variable in the debugger when stepping trough transformed class in the debugger.

3.2.3 Insert Code before Method Exit

Transformation for inserting code before the method exists is very similar to the transformation used to insert code before the start of the method. This time developer provides implementation of the *onMethodExit()* method. However, one difference is that *opcode* for the instruction that caused visited method to exit is passed as a parameter to *onMethodExit()* call and could be one of *RETURN*, *IRETURN*, *FRETURN*, *ARETURN*, *LRETURN*, *DRETURN* or *ATHROW*. More over, for all opcodes except *RETURN*, at the time *onMethodExit()* invoked, the top stack slot has the value that will be returned from the method, or exception that will be thrown when *opcode* is *ATHROW*:

```
class ExitingAdapter extends AdviceAdapter {
  private String name;

  public ExitingAdapter(MethodVisitor mv,
     int acc, String name, String desc) {
    super(mv, acc, name, desc);
    this.name = name;
  }

  public void onMethodExit(int opcode) {
    mv.visitFieldInsn(GETSTATIC,
        "java/lang/System", "err",
        "Ljava/io/PrintStream;");
    if(opcode==ATHROW) {
      mv.visitLdcInsn("Exiting on exception " +
          name);
    } else {
      mv.visitLdcInsn("Exiting " + name);
    }
    mv.visitMethodInsn(INVOKEVIRTUAL,
        "java/io/PrintStream", "println",
        "(Ljava/lang/String;)V");
  }
}
```

Obviously, this code won't catch the exception thrown from nested method calls and passed trough the caller method. To catch this case, we'll need to introduce *try/finally* block for the entire method body. In order to do that, we could use variant of the above adapter:

```
class FinallyAdapter extends AdviceAdapter {
  private String name;
  private Label startFinally = new Label();

  public FinallyAdapter(MethodVisitor mv,
      int acc, String name, String desc) {
    super(mv, acc, name, desc);
    this.name = name;
  }

  public void visitCode() {
    super.visitCode();
    mv.visitLabel(startFinally);
  }

  public void visitMaxs(int maxStack,
      int maxLocals) {
    Label endFinally = new Label();
    mv.visitTryCatchBlock(startFinally,
        endFinally, endFinally, null);
    mv.visitLabel(endFinally);
    onFinally(ATHROW);
    mv.visitInsn(ATHROW);

    mv.visitMaxs(maxStack, maxLocals);
  }

  protected void onMethodExit(int opcode) {
    if(opcode!=ATHROW) {
      onFinally(opcode);
    }
  }

  private void onFinally(int opcode) {
    mv.visitFieldInsn(GETSTATIC,
        "java/lang/System", "err",
        "Ljava/io/PrintStream;");
    mv.visitLdcInsn("Exiting " + name);
    mv.visitMethodInsn(INVOKEVIRTUAL,
        "java/io/PrintStream", "println",
        "(Ljava/lang/String;)V");
  }
}
```

Note that try/finally boundaries are defined by visitLabel() calls after *visitCode()* and before *visitMaxs()* calls, respective, for the start and the end of the *try/finally* block.

3.2.4 Replace Field Access

Field access can be replaced with a method call in order to provide additional logic. A static delegation method can be created to substitute static field access and delegation instance method -- for substituting access to instance fields. Transformation of the method code looks like this:

```
public class FieldAccessAdapter
    extends MethodAdapter implements Opcodes {
  private final String cname;
  private final Map adapters;

  public FieldAccessAdapter(MethodVisitor mv,
      String cname, Map adapters) {
    super(mv);
    this.cname = cname;
    this.adapters = adapters;
  }

  public void visitFieldInsn(int opcode,
      String owner, String name, String desc) {
    Info info = matchingInfo(opcode, owner,
        name, desc);
    if(info!=null) {
      super.visitMethodInsn(INVOKESTATIC,
          cname, info.adapterName,
          info.adapterDesc);
      return;
    }
    super.visitFieldInsn(opcode, owner,
      name, desc);
  }
  ...
```

Note that delegation methods should be generated separately using transformation for adding new methods to class described above.

3.2.5 Replace Method Call

Method call replacement is a common use case. This simple transformation can be simplified even more if the method call is replaced with a static delegation method in the same class. In that case, the method size won't increase and the method transformation can be implemented like this:

```
public class MethodCallAdapter
    extends MethodAdapter implements Opcodes {
  private final String cname;
  private final Set infos;

  public MethodCallAdapter(MethodVisitor mv,
      String cname, Set infos) {
    super(mv);
    this.cname = cname;
    this.infos = infos;
  }

  public void visitMethodInsn(int opcode,
      String owner, String name, String desc) {
    Info info = matchingInfo(opcode, owner,
        name, desc);
    if(info!=null) {
      super.visitMethodInsn(INVOKESTATIC,
         cname, info.adapterName,
         info.adapterDesc);
      return
    }
    super.visitMethodInsn(opcode, owner,
        name, desc);
  }
  ...
}
```

Note that this transformation can't be used to intercept class construction (*new* in Java language). In the bytecode object construction represented by two separate instructions that can be far apart in the steam of events. First one is *NEW* opcode that creates *not-initialized* object instance of specified type. Before that instance could be used, <*init*> method of that instance has to be called using *INVOKESPECIAL* opcode. Code generated by java compilers make this even more complicated because result of *NEW* opcode is duped on the stack and duplicated reference is used as a result value. So, a transformation like this will obviously have to use some state machine or load the entire method in memory (i.e. using ASM *tree* package) and transform it directly. Such an example is beyond the scope of this paper. We are looking into the ways to generalize it and include into the ASM *commons* package.

3.2.6 Inline Method

This transformation is very similar to the one used for merging two classes. So, we can also use content of the *MethodNode* to insert inlined code into some other method. However in this case we not only need to rename types, but we also need to replace all *RETURN* opcodes with jumps to the end of the method and make sure that *try/catch* blocks are in the right order. Here is an adapter that does that:

```
public class MethodCallInliner
    extends LocalVariablesSorter {
  private final String oldClass;
  private final String newClass;
  private final MethodNode mn;

  private List blocks = new ArrayList();
  private boolean inlining;

  private MethodCallInliner(int access,
      String desc, MethodVisitor mv, MethodNode mn,
      String oldClass, String newClass) {
    super(access, desc, mv);
    this.oldClass = oldClass;
    this.newClass = newClass;
    this.mn = mn;
  }

  public void visitMethodInsn(int opcode,
      String owner, String name, String desc) {
    if(!canBeInlined(owner, name, desc)) {
      mv.visitMethodInsn(opcode,
          owner, name, desc);
      return;
    }

    Map map = Collections.singletonMap(
        oldClass, newClass);
    Remapper remapper = new Remapper(map);
    Label end = new Label();
    inlining = true;
    mn.instructions.resetLabels();
    mn.accept(new InliningAdapter(this,
        opcode==Opcodes.INVOKESTATIC ?
            Opcodes.ACC_STATIC : 0,
        desc, remapper, end));
    inlining = false;
    super.visitLabel(end);
  }

  public void visitTryCatchBlock(Label start,
      Label end, Label handler, String type) {
    if(!inlining) {
      blocks.add(new CatchBlock(start, end,
          handler, type));
    } else {
      super.visitTryCatchBlock(start, end,
          handler, type);
    }
  }

  public void visitMaxs(int stack, int locals) {
    Iterator it = blocks.iterator();
    while(it.hasNext()) {
      CatchBlock b = (CatchBlock) it.next();
      super.visitTryCatchBlock(b.start, b.end,
          b.handler, b.type);
    }
    super.visitMaxs(stack, locals);
  }
}
```

The above adapter extends *LocalVariablesSorter* in order to handle local variables from the inlined code. It is also captures *visitTryCatchBlock()* calls the end of the method and replay them back there. Inlined method code, icluding try/catch blocks and local variables is inserted from the *MethodNode*, decorared by *InliningAdapter*. This adapter does all type renaming and replacing of the *RETURN* opcodes is done in a nested adapter:

```
public static class InliningAdapter
    extends RemappingMethodAdapter {
  private final LocalVariablesSorter lvs;
  private final Label end;

  public InliningAdapter(LocalVariablesSorter mv,
      Label end, int acc, String desc,
      Remapper remapper) {
    super(acc, desc, mv, remapper);
    this.lvs = mv;
    this.end = end;

    int off = (acc & Opcodes.ACC_STATIC)!=0 ?
        0 : 1;
```

```
    Type[] args = Type.getArgumentTypes(desc);
    for (int i = args.length-1; i >= 0; i--) {
      super.visitVarInsn(args[i].getOpcode(
          Opcodes.ISTORE), i + offset);
    }
    if(offset>0) {
      super.visitVarInsn(Opcodes.ASTORE, 0);
    }
  }

  public void visitInsn(int opcode) {
    if(opcode==Opcodes.RETURN) {
      super.visitJumpInsn(Opcodes.GOTO, end);
    } else {
      super.visitInsn(opcode);
    }
  }

  public void visitMaxs(int stack, int locals) {
  }

  protected int newLocalMapping(Type type) {
    return lvs.newLocal(type);
  }
}
```

Note that this adapter loads method arguments from the stack into remapped local variables and also removes the *visitMaxs()* event.

4. Performance

It is hard to provide direct performance comparison with other frameworks. We are trying to maintain performance suite for *null* transformation for ASM, BCEL, SERP and Javassist and compare results with the overhead of ASM transformations from the *commons* package. Here are the results of running those tests using Java 6 on Windows PS with Intel Duo 2.33Gz processor for reading and writing back 15840 classes from the rt.jar (not including file I/O):

ASM 3.x with copy pool	0.56 sec
ASM 3.x compute maxs	1.43 sec
ASM 3.x compute frames	2.67 sec
ASM 3.x tree package	1.77 sec
BCEL 5.2	16.19 sec
BCEL 5.2 compute maxs	18.39 sec
BCEL Aspectj 1.5.3	4.77 sec
BCEL Aspectj 1.5.3 compute maxs	5.93 sec
Javassist 3.4	2.70 sec
Serp 1.12.1	15.90 sec

The following table present results for running several ASM helper classes and transformations on the same classes:

ASM 3.x class info	0.04 sec
ASM 3.x SerialVersionUIDAdder	1.05 sec
ASM 3.x LocalVariablesSorter	1.29 sec
ASM 3.x analyze with SimpleVerifier	9.05 sec

5. CONCLUSIONS

This paper demonstrated number of Java bytecode transformations that can be easily assembled together to implement AOP solutions. The examples shown, implemented using the ASM framework, demonstrate the power and simplicity of the ASM framework. Using ASM allows developers to focus on code transformations instead of spending time on low level bytecode manipulations.

The ASM framework is the de-facto standard for high-performance bytecode transformations, and it is used in many Java-based applications and frameworks including code analyzers (SonarJ, IBM AUS), ORM mappers (including Oracle TopLink and Berkley DB, ObjectWeb EasyBeans and Speedo), and scripting languages (BeanShell, Groovy and JRuby).

6. ACKNOWLEDGEMENTS

First I would like to thank ASM team and all users of the ASM framework for being such a great community. I also want to thank Tim Eck, the lead developer of Terracotta DSO, and Jason van Zyl, from Maven project for feedback on this paper.

7. REFERENCES

[1] The ASM project web site, http://asm.objectweb.org/

[2] E. Bruneton, R. Lenglet, T. Coupaye. ASM: a code manipulation tool to implement adaptable systems.

[3] M. Dahm, Byte Code Engineering, Proceedings JIT'99, Springer, 1999.

[4] A. White, Serp, http://serp.sourceforge.net

[5] G. A. Cohen, J. S. Chase, D. L. Kaminsky, Automatic program transformation with JOIE, USENIX 1998 Annual Technical Conference, New Orleans, Louisiana, USA, 1998.

[6] E. Kuleshov. Using ASM toolkit for bytecode manipulation. http://www.onjava.com/lpt/a/5250

[7] E. Gamma, R. Helm, R. Johnson, J. Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Professional Computing Series. 1995.

[8] The SAX project. http://www.saxproject.org/