# ASM: a code manipulation tool to implement adaptable systems

**Eric Bruneton, Romain Lenglet, Thierry Coupaye**

*France Télécom R&D, DTL/ASR*
*28, chemin du Vieux Chêne*
*BP 98, 38243 MEYLAN cedex*
*{Eric.Bruneton,Romain.Lenglet,Thierry.Coupaye}@rd.francetelecom.com*

ABSTRACT. *ASM is a Java class manipulation tool designed to dynamically generate and manipulate Java classes, which are useful techniques to implement adaptable systems. ASM is based on a new approach, compared to equivalent existing tools, which consists in using the "visitor" design pattern* without *explicitly representing the visited tree with objects. This new approach gives much better performances than those of existing tools, for most of practical needs.*

RÉSUMÉ. *ASM est un outil de manipulation de classes Java conçu pour la génération et la manipulation dynamiques de code, qui sont des techniques très utiles pour la réalisation de systèmes adaptables. ASM est basé sur une approche originale, par rapport aux outils existants équivalents, qui consiste à utiliser le patron de conception « visiteur »* sans *représenter explicitement l'arborescence visitée sous forme d'objets. Cette nouvelle approche permet d'obtenir des performances bien supérieures à celles des outils existants, pour la plupart des besoins courants.*

KEYWORDS: *generation, transformation, dynamic, class, code, Java, visitor, adaptability.*

MOTS-CLÉS : *génération, transformation, dynamique, classe, code, Java, visiteur, adaptabilité.*

## 1. Introduction

Most of the techniques used to implement dynamically adaptable systems use some form of interposition, be it interposition objects, meta objects... Interposition can indeed be used not only to modify the semantics of components, for example to add them functional or non functional properties, but also to dynamically add and remove these additional properties.

Moreover, most of the interposition techniques use some tools to generate code or to modify existing code. Indeed an interposition object must have the same type as the type of the object it masks, which requires to generate interposition code specific to the type of this object. Likewise, in order to implement a pure Java meta object protocol, one must generally use either a code generation tool to generate interposition classes, or a code manipulation tool to inline the interposition code directly in the base classes.

Code generation and manipulation tools are therefore useful to implement adaptable systems. They can be used statically or dynamically, the dynamic solution (i.e., calling a compiler such as `rmic` on the fly, programmatically) being the most practical for the user. The dynamic solution is even required for open systems, where new, unforeseeable object types can be dynamically loaded: it is then necessary to generate interposition code, or to manipulate the code to be loaded, *on the fly*.

A solution to dynamically generate code would be to dynamically generate source code, and then to dynamically compile this source code. But this solution would add large overheads to applications, not only in time but also in size (since the applications would have to include a full compiler such as `rmic` or `javac`). It is therefore necessary to use a tool that is as small and as fast as possible. Unfortunately, in our opinion, the existing code manipulation tools are not enough small and fast. We have therefore implemented a new tool, called ASM[1], based on a new approach, in order to achieve this goal.

One of the main principle to optimize the performances of an application is to optimize the most frequently executed code first. In other words, one must concentrate on the most frequent use cases. The second objective of ASM was therefore to build a tool optimized for the most frequent *dynamic* class manipulation operations. That is to say, in our opinion, but also according to [CHI 00], adding, removing or renaming fields or methods, intercepting field accesses, adding code before or after the original code of a method...

Finally, the last objective of ASM was to build a general tool, that could be used to implement any class manipulation operation (unlike some tools, such as Javassit [CHI 00] or BCA [KEL 98], which limit themselves to a fixed set of predefined manipulation operations, that are supposed to be sufficient in most cases). In addition, the management of the dependencies between classes (which, for example, consists in automatically modifying all the classes that reference a class that has just been re-

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

named) or transformations (as in JMangler [KNI 01], which offers some support to compose transformations in the "right" order), was *not* a goal.

The next sections are organized as follows. After a brief presentation of the Java class file format, and of the problems that occur to manipulate Java classes, in section 2, we present the existing approaches to solve these problems (section 3), and then our new approach (section 4). We also present some performances measurements in section 5, before concluding.

## 2. The Java class file format

### 2.1. *General Structure*

The general structure of a Java class file [LIN 99] is a tree structure. The root of this tree contains the following nodes (* means 0 or more):

– `ConstantPool`: symbol table (cf. section 2.2);

– `FieldInfo*`: descriptions of the fields of the class;

– `MethodInfo*`: descriptions of the methods of the class;

– `Attribute*`: optional additional nodes.

A `FieldInfo` node contains the name of a field, its type, as well as some flags that indicate if the field is public, private, static... A `FieldInfo` node can also contain optional additional *attributes* (an attribute is a node whose type and size is explicitly stored). The `ConstantValue` attribute can store the value of a constant static field. The `Deprecated` and `Synthetic` can be used to indicate that a field is deprecated, or that it has been generated by the compiler (and therefore does not correspond to a field in the source code of the class).

A `MethodInfo` node contains the name of a method, the type of its parameters and of its return value, as well as flags that indicate if the method is public, private, static... A `MethodInfo` node can also contain optional additional attributes. The most important one is the `Code` attribute, that contains the code of the non abstract methods (cf. section 2.3). The `Exceptions` attribute contains the names of the exceptions that can be thrown by the method. The `Deprecated` and `Synthetic` attributes have the same meaning as above.

The optional attributes of the root node are the `SourceFile` attribute, to store the name of the source file from which the class has been compiled, the `InnerClasses` attribute, to store information about the inner classes of the class, and finally the `Deprecated` attribute. Thanks to attributes, the Java class file format is extensible. Indeed it is possible to add non-standard attributes in a class, while still being able to load this class in a virtual machine that does not recognize these attributes (a virtual machine must indeed ignore the attributes it does not recognize, which is possible thanks to the fact that each attribute contains its type and its size).

### 2.2. *The symbol table*

The symbol table of a class, or `ConstantPool`, is an array that contains a set of numerical and string constants, and of constants constructed from the previous ones, that can be referenced, through their index in this array, by other nodes in the tree structure of the class.

The goal of this table is to avoid redundancies. For example a `FieldInfo` node does not directly contain the name of a field, but the index of this name in the symbol table. Likewise, the `GETFIELD` and `PUTFIELD` instructions do not directly contain the name of the field that must be read or written, but the index of this name. The name of a field is therefore stored only once, even if many instructions refer to it.

### 2.3. *The code of the methods*

The code of a method is stored in a `Code` attribute. This code is an ordered list of basic instructions that are mainly based on a stack abstraction, although a *local variable* concept, similar to the register concept, is also available. Most of these instructions are coded into a single byte and do not have arguments (hence the name of *bytecode* instruction).

The `Code` attribute also contains an exception handler table, which corresponds to the `try catch finally` blocks in Java, and an integer that indicates the maximum size of the stack during the execution of the method. Finally the `Code` attribute can also contain optional additional sub attributes. For example the `LineNumberTable` attribute stores information that can be used to find the source code line to which each bytecode instruction corresponds. The `LocalVariableTable` stores the names of the formal parameters of the method, as well as the names of its local variables, as declared in the source code.

### 2.4. *Main problems to manipulate classes*

The main problem to manipulate a Java class is that a compiled class is just an array of bytes, which is almost impossible to modify directly. In fact it is even difficult to just *read* a compiled class, in order to analyze its tree structure (one must indeed scrupulously follow the many low level details of the serialization process used to encode the tree structure presented above into an array of bytes).

Another problem is to keep the references to the symbol table up to date. For example, if a constant is removed or added in this table, the indexes of the following constants are decremented or incremented by one, and it is therefore necessary to update all the references to these constants in the rest of the class. Likewise, after some modifications in the class a constant can be become unused, and could therefore be removed from the symbol table. However a global analysis of the class may be necessary to detect these unused constants.

Another problem is to keep up to date the instruction addresses, relative or absolute, contained in some instructions such as `GOTO` and `IFEQ`, and in some attributes such as `LineNumberTable` and `LocalVariableTable`. Indeed these addresses can change when instructions are added or removed in the code of a method (and the problem is complicated by the fact that the instructions do not all have the same size).

A last problem is to keep up to date the integer that indicate the maximum stack size of a method. Indeed the maximal stack size is a global property of a method and, in general, it is not possible to compute the maximum stack size of a transformed code by just using the maximum stack size of the original code and the characteristics of the transformation that has been made. In fact the whole transformed code must be analyzed with a control flow analysis algorithm in order to compute the new maximum stack size.

## 3. Existing approaches

This section presents the main existing Java class manipulation tools, *among those that can do any type of transformation*, and, for each tool, presents the solutions it proposes to solve the problems presented in section 2.4.

### 3.1. *BCEL*

BCEL [DAH 99] is the Java class manipulation tool that seems to be the currently most used tool (at least 30 software use it). With this tool a class modification is done in three phases. During the first phase the byte array that represents the class is deserialized, and an object model corresponding to the structure of this class is constructed in memory. An object is created for each node in this structure, down to the level of bytecode instruction (there is therefore one object per bytecode instruction). The second phase consists in manipulating the object model previously constructed. The third phase consists in serializing this modified object graph into a new byte array. The creation of a class *ex nihilo* is similar: one just needs to construct the object model corresponding to this class, and then to serialize it.

This approach solves the serialization and deserialization problems, since these functions are provided once and for all by the tool itself. The user only manipulates the object representation of classes, and never sees all the serialization and deserialization details.

This approach could also solve the symbol table management problems but, curiously, BCEL does not provide many help in this area. Indeed the user must explicitly give the indexes of the constants in the symbol table, and must "manually" update these indexes if they become invalid (in practice this does not occur frequently because BCEL does not provide the possibility to remove a constant. Since, in addition, new constants are added at the end of the symbol table, the indexes of the existing

constants remain valid - except if the symbol table is replaced by a completely new table, which is the only way to remove constants from a symbol table!).

On the other hand, BCEL solves the instruction addresses management problem in a satisfactory way. Indeed the relative or absolute addresses, stored as number of bytes, are replaced by references to the Java objects that represent the bytecode instructions, during the deserialization phase. These references are converted back to byte offsets only during the serialization phase. Therefore the user (and the tool itself) does not have to update these addresses each time an instruction is added or removed.

Finally BCEL provides a function to compute the maximum stack size of a method. This function, based on a control flow analysis algorithm, must be called after all the desired modifications have been made on a method, just before the serialization phase.

### 3.2. *SERP*

SERP [WHI ] uses a similar approach as that of BCEL, based on an object representation of classes. However SERP uses fewer classes than BCEL to represent the 200 bytecode instructions types, which leads to a smaller tool (for example SERP uses only one `MathInstruction` class to represent all the arithmetic instructions, while BCEL defines the `IADD`, `ISUB`, `IMUL`, `DADD`, `DSUB`... classes).

SERP solves the problems presented in section 2.4 in the same way as BCEL: the serialization and deserialization functions are provided once and for all by the tool, the instructions addresses are replaced by references to the Java objects that represent the bytecode instructions, and the tool also provides a function to compute the maximum stack size of a method. The only difference is that SERP solves the symbol table management problem in a better way than BCEL. Indeed the user can specify the instruction arguments by giving their values directly, instead of by giving the indexes of theses values in the symbol table. The symbol table management is therefore completely hidden to the user[2]. However SERP, like BCEL, does not automatically removed unused constants from the symbol table.

### 3.3. *JOIE*

BCEL and SERP are very similar to JOIE [COH 98], that was one of the first tool of this kind for the Java language. Indeed, according to [COH 98], JOIE used an object representation of classes, down to the level of bytecode instructions, and also used the same solutions as in BCEL or SERP (in particular for the instruction addresses management problem). However, to our knowledge, this tool is not maintained any more. In fact we were not able to download it, and we were therefore unable to study it in more details.

2. In fact SERP also provides low level methods to directly manipulate the indexes and the symbol table itself.

## 4. Proposed approach

### 4.1. *Overview*

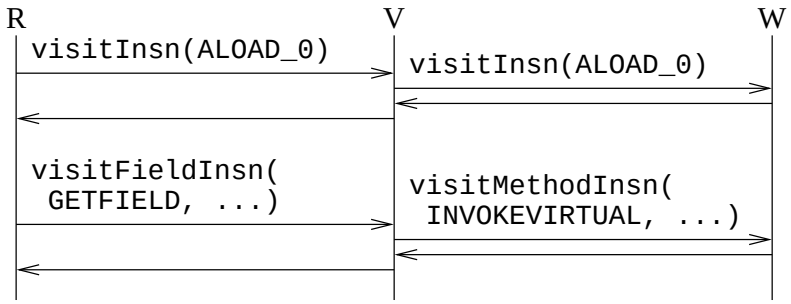
The main idea of ASM is to *not* use an object representation of the classes to be manipulated. Indeed this requires the definition of lot of classes, corresponding to the various kinds of nodes in the class tree structure, and to the various kinds of bytecode instructions (SERP contains approximately 80 classes designed for this goal, and BCEL 270), which necessarily leads to quite big tools. Moreover this approach leads to the creation of lots of objects during the deserialization of a class, which takes a lot of time and memory. Such an approach was therefore not useable in ASM, whose first goal was to be as small and as fast as possible.

But how is it possible to provide a "high-level" interface to manipulate classes, that is to say an interface that does not stay at the byte array level, without using an object representation of classes? The solution chosen in ASM is to use the *visitor* [GAM 95] design pattern, as in many transformation tools (including BCEL, SERP, OpenJava [TAT 00] . . .), but *without* explicitly representing the visited tree with objects. Indeed this pattern can be used:

– to visit a serialized object graph, without deserializing it (more precisely by deserializing only the primitive type values, i.e., numeric and string values);

– to modify an existing graph or, more precisely, to *reconstruct* a modified version of this graph (that can be constructed, in addition, directly in serialized form, without using an intermediate object representation).

In order to illustrate this idea, here is how the `ALOAD_0` `GETFIELD` *f* instruction sequence, which reads the value of the *f* field of `this`, can be transformed into `ALOAD_0` `INVOKEVIRTUAL` *getf*, which invokes the *getf* method instead, by using visitors. Three objects must be used in order to do this: a class analyzer *R*, which makes a visitor *V* visits the class, which in turn uses another visitor *W*, that reconstructs the class progressively (it is also possible to use several intermediate visitors, in order to compose several independent code transformation operations):




*R* analyzes the original instruction sequence, which is given as an array of bytes, and calls the appropriate methods on *V* for each analyzed instruction. Here this leads

to a call to the `visitInsn` method, with the `ALOAD_0` *integer* as parameter, and then to a call to the `visitFieldInsn` method, with the `GETFIELD` integer and three `String` corresponding to the owner (i.e., the name of the class to which the field belongs), name and type of the *f* field as parameters.

*V* just calls on *W* all the methods that are called on it, with the same parameters, except for the `visitFieldInsn` method. In this case, if the bytecode instruction is equal to `GETFIELD`, *V* calls the `visitMethodInsn` method on *W* (otherwise it calls `visitFieldInsn`).

Finally *W* reconstructs the class progressively, directly in serialized form. For example, when *V* calls its `visitInsn` method with `ALOAD_0` as parameter, *W* just adds, at the end of the array under construction, the byte that designates this bytecode instruction. Likewise when *V* calls its `visitMethodInsn` method, except that here the instruction requires 3 bytes.

The visitor design pattern can therefore be used to manipulate the code of methods, and more generally the complete structure of a class, without creating one object per bytecode instruction. It can therefore effectively achieve our first goal (cf. section 1), which was to build a tool as small and as fast as possible. It can also achieve our third goal, which was to be able to support any kind of manipulation operation. However the visitor design pattern is easy to use only for "simple" transformations, that are essentially local and context free. For more complex transformations, such as dead code elimination, one must use complex visitors that, in one way or another, construct an object representation of the class they visit, manipulate this representation, and make the next visitor visit it.

The approach based on the visitor design pattern is therefore more efficient than the approach used in BCEL or SERP but *only*, a priori, for simple transformations. However this is not a problem because (1) our second goal was to optimize ASM for the most frequent dynamic class transformations first and (2) because these transformations are simple in the above sense. The visitor design pattern can therefore achieve all of our goals, and this is why we choose it. The next sections show that this pattern can also solve the problems presented in section 2.4.

### 4.2. *Serialization and deserialization*

As in BCEL or SERP, all the serialization and deserialization details are managed once and for all by ASM, and are completely hidden for the user. More precisely ASM provides a `ClassReader` class that can analyze a class in binary form, and make a given visitor, implementing the `ClassVisitor` interface, visit it. ASM also provides a `ClassWriter` class, which implements the `ClassVisitor` interface, in order to construct classes directly in serialized form.

### 4.3. *Management of the symbol table*

ASM completely hides the management of the symbol table. Therefore the user does not have to manipulate the indexes of the constants in this table (in the example of section 4.1, the parameters of the `visitFieldInsn` method are directly the owner, name and type of the visited field, as `String` objects, and not the indexes of these constants in the symbol table). Moreover the fact that classes are modified by "re-building" them, instead of being modified "in place", makes it easy to automatically ensure that the symbol table does not contain unused constants.

### 4.4. *Management of instruction addresses*

ASM can not solve the instruction addresses management problem as in BCEL or SERP, by using references to the Java objects that represent the bytecode instructions, since instructions are not represented by objects. ASM uses *labels* instead, represented by `Label` objects, as in assembly languages. Let us consider, for example, the instruction sequence below, on the left, that decrements the local variable 1, pushes its value on the stack, and jump to the first instruction, 5 bytes back, while this value is not nul:

```
IINC 1 -1        l: IINC 1 -1          IINC 1 -1
ILOAD 1             ILOAD 1            ILOAD_1
IFNEQ -5            IFNEQ l            IFNEQ -4
```

The `ClassReader` class makes a preliminary analysis of the code in order to detect jump instructions, and computes the labels that are necessary (here `l`). The `ClassReader` then calls the appropriate methods to visit the bytecode instructions *and* the labels: `visitLabel(l)` `visitIincInsn(1, -1)` `visitVarInsn(ILOAD, 1)` and finally `visitJumpInsn(IFNEQ, l)`. The instruction addresses are therefore hidden to the user, as in BCEL and SERP. As in these tools they are recomputed only during the serialization phase, i.e., in the `ClassWriter` class. Here, for example, the `ClassWriter` stores the absolute address corresponding to `l` during the visit of this label and, during the visit of the `IFNEQ` instruction, computes the corresponding relative address[3] (here -4 instead of -5, because the `ClassWriter` automatically replaced `ILOAD 1` by `ILOAD_1`, whose size is one byte instead of two).

### 4.5. *Computation of the maximum stack sizes*

Like BCEL and SERP, ASM provides a function to compute the maximum stack size of a method. This size is computed by using a control flow analysis algorithm.

3. For forward references the relative addresses can not be computed immediately. In this case ASM reserves space to store these addresses, memorize the position of these "holes", and fi lls them as soon as the appropriate label is visited.

This algorithm requires decomposing the method's code into *basic blocs* (an instruction sequence without ingoing nor outgoing jumps), linking these blocks into a graph, where an edge represents a possible jump from on block to the beginning of another, computing the maximal stack size for each block, and finally visiting the graph in order to compute the global maximum stack size.

The basic blocs, represented by the `Label` objects, as well as their maximum stack size, are computed during the visit of the method's code. The control flow graph is also computed during this visit. The global maximum stack size is computed by visiting this graph once it is complete, i.e., when all the instructions have been visited.

## 5. Performances

This section compares the performances of ASM to those of BCEL and SERP. We did not compare ASM to Javassist or to BCA, because they do not provide the same functionalities.

ASM contains only 13 classes, and takes only 21KB in `.jar` format, while SERP takes 150KB, and BCEL 350KB (without the bytecode verifier, which is not provided by SERP or ASM). ASM is therefore much more smaller than these tools.

In order to compare the runtime performances we measured the time needed to load a large number of classes from a `.jar` file (a) with a standard `ClassLoader`, (b) by deserializing and reserializing each class before loading it, (c) by recomputing, in addition to (b), the maximum stack size of each method and (d) by adding a counter in each class, and instructions to increment this counter at the beginning of each method, before reserializing and loading each class. The results are the following (for 1155 classes coming from the JDK's `tools.jar` archive, whose mean size is 3KB, and with the JDK1.3.1 Hotspot VM on Linux 2.4.9, on a Pentium III 1 GHz):

| | Case a | Case b | Case c | Case d |
|---|---|---|---|---|
| ASM | 1,98s | 3,22s | 3,29s | 3,26s |
| BCEL | 1,98s | 16,6s | 19,2s | 16,8s |
| SERP | 1,98s | 24,3s | 34,6s | 28,0s |

As can be seen ASM only adds a 60% overhead to the time needed to load the classes with a normal class loader, while BCEL adds a minimum 700% overhead, and SERP a minimum 1100% overhead! For the serialization/deserialization time only, given by $b - a$, ASM is 12 times faster than BCEL, and 18 times faster than SERP. ASM is also 35 (resp. 130) times faster than BCEL (resp. SERP) to compute the maximum stack sizes (time given by $c - b$), and 4 (resp. 80) times faster than BCEL (resp. SERP) to do the transformation itself (time given by $d - b$). All these results show that ASM is effectively much more efficient than BCEL and SERP.

## 6. Conclusion

Code generation and manipulation tools are useful to implement adaptable systems but, in order to be useable in a dynamic way (which is mandatory for open systems), they must be as small and as fast as possible. The new approach that we are proposing to achieve this goal, which consits in using the visitor design pattern *without* representing the visited tree with objects, and implemented in the ASM tool, can effectively achieve this goal, as shown by the measurements we made.

ASM is currently used to implement the Fractal [BRU 02] component model, in order to dynamically generate the interposition object classes, and also to dynamically merge some classes (which require code manipulation features), for optimization purposes. ASM could also be used for *contractual composition* [COU 01], in order to dynamically generate optimized code to associate contracts (given as assertions, as predicates in a temporal logic...) to components. We also plan to use ASM in order to implement what we call *operating composition* [COU 01]. In order to do this a framework to implement reusable code transformation components, called JABYCE, is under development. It will be based on the same ideas as ASM (and will probably reuse the `ClassReader` et `ClassWriter` classes), but with less emphasis on performances, and more emphasis on the interface separation [MAR 96b], common closure [MAR 96a] and open/close [MEY 97] principles (in order to provide independent and reusable class transformation *components*). ASM can be downloaded from the ObjectWeb web site, `www.objectweb.org`.

## 7. References

[BRU 02] BRUNETON E., COUPAYE T., STEFANI J. B., "Recursive and Dynamic Software Composition with Sharing", *Workshop on Component Oriented Programming*, Malaga, Spain, 2002.

[CHI 00] CHIBA S., "Load-time Structural Reflection in Java", *ECOOP 2000 – Object-Oriented Programming*, vol. LNCS 1850, Springer, 2000, p. 313-336.

[COH 98] COHEN G. A., CHASE J. S., KAMINSKY D. L., "Automatic program transformation with JOIE", *USENIX 1998 Annual Technical Conference*, New Orleans, Louisiana, USA, 1998.

[COU 01] COUPAYE T., LENGLET R., BEAUVOIS M., BRUNETON E., DÉCHAMBOUX P., "Composants et composition dans l'architecture des systèmes répartis", *Journées Composants*, Besançon, France, 2001.

[DAH 99] DAHM M., "Byte Code Engineering", *Proceedings JIT'99*, Springer, 1999.

[GAM 95] GAMMA E., HELM R., JOHNSON R., VLISSIDES J., *Design Patterns*, Addison-Wesley, 1995.

[KEL 98] KELLER R., HÖLZLE U., "Binary Component Adaptation", *ECOOP 1998 – Object-Oriented Programming*, vol. LNCS 1445, Springer, 1998, p. 307-329.

[KNI 01] KNIESEL G., COSTANZA P., AUSTERMANN M., "JMangler - A Framework for Load-Time Transformation of Java Class Files", *IEEE Workshop on Source Code Analysis*

*and Manipulation (SCAM)*, Nov. 2001.

[LIN 99] LINDHOLM T., YELLIN F., *The Java Virtual Machine Specification*, Addison-Wesley, 2nd edition, 1999.

[MAR 96a] MARTIN R. C., "Granularity", *C++ Report*, Nov. 1996.

[MAR 96b] MARTIN R. C., "The Interface Segregation Principle", *C++ Report*, Aug. 1996.

[MEY 97] MEYER B., *Object-Oriented Software Construction*, Prentice-Hall, second edition, 1997.

[TAT 00] TATSUBORI M., CHIBA S., KILLIJIAN M.-O., ITANO K., "OpenJava: A Class-based Macro System for Java", *Reflection and Software Engineering*, vol. LNCS 1826, Springer, 2000, p. 119-135.

[WHI ] WHITE A., "Serp – http://serp.sourceforge.net".