Improving Efficiency by Weaving at Run-time

Miklós Espák

Institute of Informatics, Department of Information Technology,
University of Debrecen,
H-4010 Debrecen, Egyetem tér 1. Hungary
espakm@inf.unideb.hu

Abstract. Since the appearance of Java several tools have been developed to allow making dynamic changes in Java programs. In general, these tools trades the flexibility for the efficiency of the program.
In this paper I present RtJAC, an aspect-oriented framework for Java based on JAC [3], which makes possible to create aspects, specify their join points and weave them into an application, all at run-time. I show that weaving at run-time not only provides more flexibility but can significantly improve the performance of a dynamic aspect-oriented system.

1 Introduction

There are numerous kinds of concerns that affect many classes at the same time, and whose code cannot be separated easily in the traditional modular approach: tracing, persistency, distribution, etc. Using aspect-oriented programming (AOP) [2] it is possible to tackle this problem. In AOP the code scattered in many classes can be specified in an aspect, separately, and beside this code, positions in the original code must be given (join-points). The mechanism for inserting the aspect code into the program at the given positions is called weaving. Based on the time when this is done we talk about compile-time, load-time or run-time weaving, respectively.

In the last few years a lot of research has been done to develop effective and flexible run-time AOP systems. In this paper I present a new approach in this field. The tool I created is called RtJAC (Run-time Java Aspect Components).

2 JAC

The architecture of this AOP system is based on Java Aspect Components (JAC) [3]. JAC is a Java framework for aspect-oriented programming. In contrast to AspectJ [4] it weaves the aspects' code into the base level program at load-time, not at compile-time. The load-time transformation of classes makes the access to the source code unnecessary. Another important advantage of JAC over AspectJ is that aspects can be enabled and disabled at runtime. JAC works as a distributed application server. It enables an aspect (in the terminology of JAC an aspect component) to be weaved into a remote, already running application. However, at the time of loading the classes of the application it cannot be determined which classes will have to be wrapped later. For this reason JAC transforms the bytecode of every class of the application (except for those mentioned in a configuration file) when they are loaded into the Java Virtual Machine (JVM). During this process it wraps all constructors and public methods of the class so that the additional behavior defined in an aspect component can be executed when needed.

JAC supports a new way of reusing aspects. Since pointcuts belong to the aspect instances, they do not need to be specified completely and can instead be parameterized at the instantiation of the aspect. The parameters can be specified in a configuration file that is processed at program startup. This way you do not need to inherit the aspect when you wish to apply it in a specific context.

3 Implementation

Beginnning with J2SE 1.4.0 the Java Platform Debugger Architecture (JPDA) [5] supports "HotSwap" replacement of classes. On one hand it aids tool (IDE) vendors to provide the ability to do fix-and-continue debugging. On the other hand it allows organizations deploying long-running servers to fix bugs without bringing down the server. Unfortunately, existing implementations of the JPDA do not allow any structural modifications to be carried out on classes. The only thing allowed is changing the body of methods.

This limitation has several points of impact on implementing a run-time weaver. In the course of transforming bytecode, JAC changes the structure of a class in three ways. At first it introduces new fields into the class for caching some values needed to perform the behavior specified by the aspects. At second it wraps all methods and public constructors of the class in two steps: it changes the name of the original method and then creates a new method with the specification of the original one, which will contain the wrapping code in its body. To enable the JVM to redefine some classes, each reference to these new fields had to be eliminated and a technique had to be worked out, by which wrapping of a method can be done within the method itself. It is implemented in RtJAC in two ways.

The first one wraps the method using a "recall" mechanism. There is a flag in class `Wrapping` indicating whether the method call is a recall by an aspect or not. This flag will be set directly before the recall and then cleared by a `Wrapping.IAmRecalled()` invocation. The `Wrapping` class is locked for this short time to avoid problems with multithreaded applications.

This technique cannot be used for constructors because their first instruction (at the bytecode level) should be either an invocation of a constructor (of the superclass or the current class) or an assignment to a field of the current class. Instead of this the constructor starts the aspect code in a new thread after the super call. The base code and the aspect code is synchronized through a state variable in the thread.

Fig. 1 and 2 illustrate the two techniques I applied.

```
class Foo extends SuperFoo{
  private double d;

  public Foo(String s, double d){
    super(s);
    this.d = d;
  }
  public double m1(int i, double d){ return i + d; }
}
```

Fig. 1. Before transformation

```
class Foo extends SuperFoo{
  private double d;

  public Foo(String s, double d){
    super(s);
    AbstractMethodItem ami = ClassRepository.get().
        getClass("Foo").getAbstractMethodItem("<init>():V");
    WrappingChain wc = Wrapping.getWrappingChain(this, ami);
    Interaction interaction = new Interaction(
      wc, this, ami, new Object[]{s, new Double(d)});
    ConstructorThread ct=new ConstructorThread(interaction);
    while(!ct.beforeFinished());
    if(!ct.afterFinished(){
      this.d = d; // the original body
      ct.switchNextState();
      Thread.yield();
      while(ct.afterFinished());
    }
  }
  public int m1(int i, double d){
    if(!Wrapping.isRecall()){
      AbstractMethodItem ami = ClassRepository.get().
        getClass("Foo").getAbstractMethodItem("m1(int,double):V");
      WrappingChain wc=Wrapping.getWrappingChain(this, ami);
      Object o=Wrapping.nextWrapper(new Interaction(wc,this,
           ami,new Object[]{new Integer(i),new Double(d)}));
      return ((Integer)o).intValue();
    }
    else{
      Wrapping.IAmRecalled();
      return i + d; // original body
    }
  }
}
```

Fig. 2. After transformation

The third step of the transformation performed on classes by JAC is the introduction of the marker interface `Wrappee`. Unfortunately, this cannot be eliminated, so the references to `Wrappee`s had to be replaced to references to `Object`s in the whole framework. Although, this makes RtJAC not fully compatible with JAC, only simple modifications are required to make JAC aspect components conform to RtJAC. (It can be performed automatically.)

4 Example

Fig. 3 illustrates the usage of RtJAC. The way of managing aspects is the same as in JAC, except that now you can use the AspectManager.register() method to insert aspect components into the application. When you want the changes to take effect, you should invoke the JacVM.redefineClasses() method that redefines the classes that are affected by some of the new aspects.

It can be seen that aspect instances can be created from the application itself depending on runtime values.

```
public class Run {
  public void mm(int i){
    System.out.println("mm: " + i);
  }

  public static void main(String[] args) throws IOException{
    Run rr = new Run();
    rr.mm(62); // writes "mm: 62" to the standard output
    BufferedReader br =
        new BufferedReader(new InputStreamReader(System.in));
    MyTracingAC tracer = new MyTracingAC(br.readLine());
    tracer.addTrace(".*", ".*Run", "mm.*");
    ACManager acm = ACManager.getACM();
    acm.register("TRACER", tracerac);
    rr.mm(29); // additionally writes trace information into
               // the file given at runtime
    br.close();
  }
}
```

Fig. 3. Sample code

5 Efficiency

In spite of eliminating the fields for caching, a sample benchmark delivered by JAC (`jac.sample.bench.Bench`) shows that this way of wrapping methods can lead to faster method invocations. Unfortunately, wrapping constructors makes them reasonably slow because of the overhead of creating a thread instance and synchronization. The following table shows the result of the benchmark:

Table 1. Efficiency of methods wrapped by JAC and RtJAC

	JAC	RtJAC	Speedup
1 000 direct method calls	259.76ms	235.66ms	109.76%
1 000 direct constructor calls	1 308.86ms	4 408.87ms	29.68%

A more important difference between the two systems is that RtJAC does not need to know the set of classes to be transformed at the time the application is started. Although the hooks inserted into the bytecode are minimal, they can make the application reasonably slower because of the trade-off between speed and flexibility. Table 2 shows the running time of a test program with JAC and RtJAC.

Table 2. Speedup achieved by reducing the number of “empty loops”

JAC	RtJAC	Speedup
117 552ms	84 006ms	139.93%

The test is based on another sample program of JAC and contains a single method call nested in a loop. (`jac.samples.calcul.Calcul.bench(10000)`) The method call is traced by an aspect.

A third advantage we get is that the time spent on translating the bytecode is reduced, as fewer classes have to be transformed.

6 Related work

The first tools enabling modifications of a program (without the notion of aspect orientation) were provided by meta-object protocols (MOPs) [6]. Using MOPs some basic mechanisms of the implementation of a language can be redefined by the programmer (such as method invocation, method dispatch, access to fields, etc.) allowing him to redefine the behavior or even the structure of a program from the program itself. Two run-time MOPs have been published for Java so far: Guaraná [7] and metaXa [8] (formerly called MetaJava). Though both provide a flexible and quite effective system, they are implemented by using a modified JVM, so applications created for them cannot be run on a standard virtual machine.

Handi-Wrap [9] is a dynamic AOP system using both the compile-time approach and bytecode adaptation. It defines a language extension for Java to make it easier to write aspects (so called wrappers) and to insert them into the application (the wrap statement). Handi-Wrap also uses bytecode rewriting to add prologues (hooks) to method bodies. However, being also a compiler, it can probably make a good assumption on the subset of classes that needs to be transformed. Unfortunately, Handi-Wrap is not available for public access.

Prose [11] is another suite allowing aspect weaving at run-time. Like RtJAC, the initial version of Prose [10] was also based on the debugger architecture of Java. However, partly because of the inefficiency of JPDA at that time, it was also very inefficient. The implementation was based on a feature of the JVM Debugger Interface that allows the user to request for notification at certain events inside the JVM (e.g. a method call) and then control execution of that event.

In its recent implementation [11] the architecture of Prose consists of two layers: the execution monitor (EM) and the AOP engine. The execution monitor is integrated with the JVM and is responsible for managing join-points and notifying the AOP engine when the execution reaches one. The EM achieves this task by inserting minimal hooks into the classes at all possible places for join points. To improve efficiency this insertion is carried out while translating the bytecode of methods to native code. As several "empty" hooks can be inserted so, it is suggested to restrict the scope of dynamic AOP to a subset of possible classes.

7 Conclusions and future work

As based on JAC, RtJAC provides a sound, quite popular framework for aspect-oriented programming. It is largely compatible with JAC, which means you can define aspect components and introduce them into the application in the usual way. RtJAC runs on a standard JVM (at this moment only SUN's JVMs and their Blackdown ports support dynamic redefining of classes).

RtJAC improves JAC in two ways. Firstly, it provides more flexibility by letting the application control the integration of new aspects at run-time. Secondly, it avoids transformation of classes not affected by any aspect. The tests show that by minimizing the number of empty hooks a significant speedup can be achieved. Further reduction of this number could be achieved by analyzing the pointcut definitions more thoroughly (looking for affected methods not classes). However, it must be noted that in order to make redefining of classes possible some optimizations of JAC had to be eliminated.

The main disadvantage of RtJAC in contrast to JAC is that it enables only behavioral modifications of classes. By combining the load-time and run-time techniques a reasonable compromise could be made between flexibility and efficiency.

References

1. Espák, Miklós: Soul: A New Metaobject Protocol for Java (Hungarian). Master's thesis, Institute of Mathematics and Informatics, University of Debrecen, (2001)
2. G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier, J. Irwin. Aspect-Oriented Programming. In 1997, European Conf. on Object-Oriented Programming (ECOOP, '97), pages 220-242. Springer Verlag, 1997.
3. R. Pawlak, L. Duchien, G.Florin, F. Legond-Aubry, L. Seinturier, L. Martelli: Jac: An Aspect-Based Dynamic Distributed Framework (2002) http://jac.aopsys.com/doc/JAC.pdf
4. Xerox Corporation: The AspectJ Programming Guide. http://www.eclipse.org/aspectj
5. Sun Microsystems: Java Platform Debugger Architecture http://java.sun.com/j2se/1.4.1/docs/guide/jpda/
6. G. Kiczales, J. de Rivieres, D. G. Bobrow: The Art of the Metaobject Protocol. MIT Press, 1991
7. A. Oliva and L. Buzato: The design and implementation of Guaraná. In: Proc. of the 5th USENIX Conf. on Object-Oriented Technologies and Systems, 203–216 The USENIX Association (1999)
8. M. Golm: Design and Impementation of a Metalevel Architecture for Java. Diplomarbeit, Institute fr Matematische Maschinen und Datenverarbeitung der Friedrich-Alexander-Universitaet (1997)
9. J. Baker and W. Hsieh. Runtime Aspect Weaving Through Metaprogramming. In 1st Intl. Conf. on Aspect-Oriented Software Development, Enschede, The Netherlands, pages 86-95, Apr. 2002.
10. A. Popovici, T. Gross and G. Alonso: Dynamic Weaving for Aspect Oriented Programming. In: 1st Intl. Conf. on Aspect-Oriented Software Development, Enschede, The Netherlands, Apr. 2002.
11. A. Popovici, G. Alonso and T. Gross: Just-In-Time Aspects: Efficient Dynamic Weaving for Java. In: 2nd Intl. Conf. on Aspect-Oriented Software Development, 100–109, Boston, USA, March 2003.
12. M. Dahm: The Bytecode Engineering Library http://jakarta.apache.org/bcel/