Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorarbeit

Karsten Becker

Comparison of weaving technologies in Java based Aspect Oriented Programming languages

Fakultät Technik und Informatik
Studiendepartment Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Karsten Becker

Comparison of weaving technologies in Java based Aspect Oriented Programming languages

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung

im Studiengang Technische Informatik
am Studiendepartment Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Friedrich Esser
Zweitgutachter: Prof. Dr. rer. nat. Gunter Klemke

Abgegeben am 22. Mai 2007

Thema der Bachelorarbeit
Weaving Technologien Java basierter Aspektorientierter Programmiersprachen im Vergleich
Stichworte
AOP weaving Java
Kurzzusammenfassung
Jede Aspektorientierte Programmiersprache bringt seine eigene Weaving Technologie mit. Diese werden miteinander verglichen und im Abschluss eine Empfehlung für einen JSR gegeben.
Title of the paper
Comparison of weaving technologies in Java based Aspect Oriented Programming languages
Keywords
AOP weaving Java
Abstract
Every aspect oriented programming language features its own weaving technology. In this thesis those technologies are compared with each other and a JSR is proposed.

Inhaltsverzeichnis

1 Introduction

Java has become a very famous and popular object oriented programming (OOP) language. Object orientation works very well and succeeded procedural programming in most places. But as applications grow larger and larger, the architecture became more difficult. The ultimate goals of clean modularization and reusability became harder to achieve.

One reason are crosscutting concerns. These concerns can be implemented in an object oriented way, but the invocation code is tangled or squattered across many classes. Aspect oriented programming (AOP) is meant to solve this problem by using pointcuts, advices and intertype declaration.

The question is, why is AOP still in research. Even though it is known for nearly a decade now. AspectJ is one of the most popular AOP tools and just became an Eclipse tools project. While most people have no problems with installing an extra IDE plugin, the deployment of aspect enriched code is still quite tricky. In case of offline weaving it is as easy as providing an extra lib in classpath. But when unleashing all advantages of AOP one has to deal with JVM dependant command line options, java agents (with native code parts) and bootclasspath extensions. Each of them affects the JVM behaviour and therewith performance in a certain way.

If AOP is meant to provide a solution to the modularization problem of OOP, where are the libraries that easily enable persistence, logging etc. . . There are tons of libraries written in OOP that provide help for those cases. But it is still necessary to write some AOP code to clue this OOP libraries into the code.

In this thesis we will have a look at the different weaving technologies and compare them with each other. At the end of this thesis we will outline how a solution to those problems might look like.

1.1 The problem

There are many Java based aspect oriented languages. Aosd.net[1] names a few of them. Most of them feature an offline weaver, and some provide a load time or online weaving mechanism. While offline weaving is quite easy to implement, it does not provide the flexibility the Java language inherits. In Java classloaders provide the ability to load classes during runtime. Applying the advices to those newly loaded classes is quite difficult.

For this comparison the following languages were chosen:

- HyperJ[2] - very basic weaving
- AspectJ - the most popular language

[1] http://www.aosd.net/wiki/index.php?title=Research_Projects
[2] http://www.alphaworks.ibm.com/tech/hyperj

- Spring - dynamic proxies
- SteamLoom[1] - highest integration into JVM

1.2 Introduction into Weaving

After defining advices, they need to be applied to the specified joinpoints. This process is called weaving because it changes the current control flow of a program by introducing some additional code pieces. The most common way of doing this, is modifying the generated *.class* file. As the format of a classfile is well documented[2] it is very easy to identify the correct places for insertions. The process of enriching the bytecode of a classfile is called offline weaving. Usually the code is compiled by javac, or any other compiler, and modified directly afterward.

But as Java is a dynamic language, it can load classes from any location at any time. Those classfiles are out of reach for the offline weaver. When an advice should be applied to any implementation of Comparable#compareTo(Object) only those classes with existent classfiles are modified. In order to apply the advice to *any* implementation of Comparable#compareTo(Object), it is necessary to use a different weaving approach. For load time weaving (LTW) it is necessary to have control over the classloader which is not always possible. Those few weavers that work online are mostly based on bytecode instrumentation.

1.3 Premises

To fully understand this thesis it helps to be familiar with aspect oriented programming and Java in general. The preferred language for representing aspects will be AspectJ because of its popularity. To understand the weaving results it is recommended to read an introduction into JVMs bytecode. A good introduction into the basics of JVM and its bytecode can be found on JavaWorld[3]. A quick reference of bytecode instructions can be found online[4].

[1] http://www.st.informatik.tu-darmstadt.de/static/pages/projects/AORTA/Steamloom.jsp
[2] http://java.sun.com/docs/books/vmspec/2nd-edition/html/ClassFile.doc.html
[3] http://www.javaworld.com/javaworld/jw-06-1996/jw-06-vm.html
[4] http://www.cat.nyu.edu/ meyer/jvmref/

2 Offline weaving

From an historical point of view IBM Alphaworks were among the first that researched separation of concerns. Their solution was called HyperJ. Nowadays HyperJ is outdated and just demonstrates how first solutions looked like.

2.1 HyperJ

HyperJ is not exactly aspect oriented. It calls itself subject oriented programming. But it allows to achieve the same goals as aspect oriented software development. Concerns, or as in this case, subjects are spread over different files. This way it is easier to work with many people on the same class, with each having their own subject. HyperJ has no specific language and uses regular Java classes. Those classes get compiled, and the HyperJ Compiler will mix up the bytecodes. The following example is from Lai u. a. (2000)

```
1  package offline.hyperj.kernel;
2  class A {
3   void print() {
4    System.out.println("Hello");
5   }
6   static void main(String args[]) {
7    A a = new A();
8    a.print();
9   }
10 }
```

```
1  package offline.hyperj;
2  class B{
3   void print() {
4    System.out.println(" World");
5   }
6  }
```

The missing “ World” should be added to the original class A. Both classes are written in plain Java. They can be compiled by a normal Java compiler. So compiling them is the first step. Afterwards HyperJ will mix them up, but it needs to be specified how to mix them up. Therefor 3 files must be created.

- Hyperspace
- Hypermodule
- Concern Mapping

A Hyperspace is the grouping of Hypermodules. Those Hypermodules express concerns that are supposed to be disjoint in its dimension. Which basically means that a logging concern i.e. must not have any relation to the persistence concern. In this example the Hyperspace consists of all classes in package offline.hyperj.kernel and offline.hyperj.

```
hyperspace Test
    composable class offline.hyperj.*;
    composable class offline.hyperj.kernel.*;
```

The next step is to assign concerns to classes and to provide names for them. This is done in the concern mapping file.

```
class A : Feature.Kernel
class B : Feature.New
```

The Hypermodule file defines Hyperslices. Each Hyperslice represents a single concern. In this case the original code is called Feature.Kernel that will be extended by Feature.New. Those concerns, formerly known as features, a.k.a. subject are now tied together. This is done by declaring a relationship which in this case is mergeByName. It means that classes with the same name should be merged. But as different names have been used, a manual equate has to be performed.

```
hypermodule Test
   hyperslices:
        Feature.Kernel,
        Feature.New;
 relationships:
    mergeByName;
    equate class Feature.Kernel.A, Feature.New.B;
end hypermodule;
```

After compilation new class files were generated. The decompilation results in:

```
protected void print__from_offline_hyperj_kernel__A()
 {
  System.out.println("Hello");
 }

protected void print__from_offline_hyperj__A()
 {
  System.out.println(" World");
 }

public void print()
```

```
12 {
13   Object aobj[] = new Object[0];
14   print__from_offline_hyperj_kernel__A();
15   print__from_offline_hyperj__A();
16 }
```

The original print functions from class A and B have been copied and renamed. A new print method has been generated which calls the renamed versions of print. The order of calling can be specified in the hypermodule file.

This is the very first and easiest way of separating different concerns to different files. But the HyperJ project is in fact dead. More flexible and easier to understand approaches have been invented.

2.2 General

Most aspect compiler use some sort of library for performing the classfile transformation. The most popular is BCEL[1]. But there are some other ones like: ASM[2] or SERP[3].

A Java classfile consists of multiple parts.

- Header
- Constant pool
- Access rights
- Implemented Interfaces
- Fields
- Methods
- Class Attributes

The most interesting parts are the fields, methods and constant pool (CP). As Java is a statically typed language, it is necessary to have static references to all classes that are used. Those references are stored as special coded strings in the CP. So if a method is changed and a call to a new not yet known class is done, the constant pool needs to be updated. More detailed information about this pool is available online[4]. Field names as well as method signatures are stored as coded strings. Those strings are stored in the CP and are referenced using their index. So fields and methods both have at least:

[1] http://jakarta.apache.org/bcel/
[2] http://asm.objectweb.org/index.html
[3] http://serp.sourceforge.net/
[4] http://java.sun.com/docs/books/vmspec/2nd-edition/html/ClassFile.doc.html#20080

- access flags (public,private)
- name_index (index in cp)
- descriptor_index (coded typeinformation string)
- attributes (for fields: constant, synthetic)

Methods have a special attribute named code. In this attribute information about the method are stored:

- max_stack
- max_locals
- code[]
- exception_table
- attributes

The code field contains the java bytecode of this method. During the execution of a method, variables have to be pushed into the operand stack. Calling a method with 4 parameters for example would take at least 5 slots on the stack. The first slot on the stack will be a reference to the instance the method is called on. After calling the 5 operands are popped out and the result is pushed in. The max_stack field is the maximum number of slots used during execution of this method. Max_locals is the number of local variables that are used during execution. The exception table contains a range that is guarded by a certain handler.

When the code of a method is modified, there are several things that must be updated too. In example the max_stack, max_locals, exception_table and many more. Like branch instructions which always refer to absolute or relative positions. All tools automatically update those field. But some tools like BCEL provide a very high-level object oriented view on a class, whereas ASM can be used in a low-level way.

2.2.1 BCEL

A small example shows how easy it is to insert some code into a method using BCEL. A class that needs some after and before instructions.

```
1 public class TestClass {
2  public TestClass(){}
3  public static void main(String[] args) {
4   if (args.length==0) return;
5    System.out.print("World");
6  }
7 }
```

At first some methods for loading the class for modifications.

```
JavaClass testClass =
  Repository.lookupClass("offline.general.bcel.TestClass");
Method[] methods = testClass.getMethods();
ClassGen gen = new ClassGen(testClass);
InstructionFactory factory = new InstructionFactory(gen);
```

JavaClass is the BCEL-representation of a class file. It has methods for accessing/modifying every part of it. The ClassGen class can then be used for modifying this JavaClass. It helps keeping track of the constant pool etc. The factory is just a utility for creating some frequently used statements. As for example a println.

A call to getCode on the Method object allows the inspection of the bytecode instructions. The main method looks as following:

```
Code(max_stack = 2, max_locals = 1, code_length = 15)
0:    aload_0
1:    arraylength
2:    ifne            #6
5:    return
6:    getstatic java.lang.System.out Ljava/io/PrintStream; (22)
9:    ldc             "World" (24)
11:   invokevirtual
          java.io.PrintStream.print (Ljava/lang/String;)V (30)
14:   return
```

The easiest thing to do, is to insert a new invoke at the beginning of this method. The code in BCEL:

```
MethodGen mg = new MethodGen(method,
          testClass.getClassName(),gen.getConstantPool());
InstructionList original = mg.getInstructionList();
InvokeInstruction before = factory.createInvoke(
     Aspect.class.getName(), "before", Type.VOID,
     new Type[0],Constants.INVOKESTATIC);
original.insert(before);
```

InstructionLists are a list of multiple instructions which can easily be referred to from any part of the code. After modifying the code the numbers for max_stack and max_locals must be recomputed. This can be done by:

```
mg.setMaxLocals();
mg.setMaxStack();
Method newMethod = mg.getMethod();
```

GetMethod recreates the new method and keeps track of branch instructions and exceptions handlers that might have changed.

The new modified bytecode:

```
Code(max_stack = 2, max_locals = 1, code_length = 18)
0:    invokestatic      offline.general.bcel.Aspect.before ()V (39)
3:    aload_0
4:    arraylength
5:    ifne              #9
8:    return
9:    getstatic java.lang.System.out Ljava/io/PrintStream; (22)
12:   ldc               "World" (24)
14:   invokevirtual
          java.io.PrintStream.print (Ljava/lang/String;)V (30)
17:   return
```

This was quite easy as the code has only been inserted at the beginning. But for creating an after advice it is necessary to replace all return instructions with an invocation to the advice. Even this is quite easy to realize:

```
InstructionList original = mg.getInstructionList();
InstructionHandle[] handles = original.getInstructionHandles();
for (InstructionHandle handle : handles) {
 if (handle.getInstruction()
         .equals(InstructionConstants.RETURN)) {
  InvokeInstruction after = factory.createInvoke(
       Aspect.class.getName(), "after", Type.VOID,
       new Type[0],      Constants.INVOKESTATIC);
  original.insert(handle, after);
 }
}
```

2.2.2 ASM

When it comes to big classes or the requirement of speed ASM might be the best choice. Compared to XML it is the SAX of class parsing, while BCEL is more like DOM. ASM is completely event driven. Every part of a class file has its own visiting method.

The process of reading a class file is done using the visitor pattern. The ClassReader class accepts a ClassVisitor, which then visits all parts of this class. The class ClassWriter is a ClassVisitor implementation which writes every visited part to a ByteVector. If used without modification the resulting byte array will be an exact representation of the class that was

read. To increase speed it is possible to copy the constant pool from the ClassReader by passing it to the constructor of ClassWriter. This way the constant pool is meant for additions only. Old and maybe unused constants will remain unchanged.

For modifications to the bytecode of a method, it is necessary to subclass the ClassWriter and provide an own implementation of a MethodAdapter. This method adapter then performs some calls to the original MethodVisitor.

```
final String classname = "offline.general.asm.TestClass";
ClassReader cr=new ClassReader(classname);
ClassWriter writer = new ClassWriter(cr,false){
 @Override
 public MethodVisitor visitMethod(
      int access, String name, String desc,
      String signature, String[] exceptions) {
  MethodVisitor methodVisitor =
      super.visitMethod(access, name, desc, signature, exceptions);
  if ("main".equals(name)){
   MethodAdapter adapter = new MethodWeaver(methodVisitor);
   return adapter;
  }
  return methodVisitor;
 }
};
cr.accept(writer, true);
```

The class MethodWeaver performs the insertion of additional code by calling the original MethodVisitor (an implementation of ClassWriter).

```
private static final class MethodWeaver extends MethodAdapter {
 private MethodWeaver(MethodVisitor mv) {
  super(mv);
 }

 @Override public void visitCode() {
  mv.visitMethodInsn(Opcodes.INVOKESTATIC,
     "offline/general/asm/Aspect", "before", "()V");
  super.visitCode();
 }

 @Override public void visitInsn(int opcode) {
  if (opcode==Opcodes.RETURN){
   mv.visitMethodInsn(Opcodes.INVOKESTATIC,
```

```
15          "offline/general/asm/Aspect", "after", "()V");
16      }
17      super.visitInsn(opcode);
18    }
19  }
```

ASM tracks the cp,max_locals and max_stack. But once it has visited a method it is not possible to change it anymore.

3 Load time weaving

Load time weaving (LTW) is the process of weaving advices or intertype declarations into a class when it is about to be loaded. Thus this is done by providing a ClassLoader or modifying one.

At first a short overview over the classloader architecture in Java. A more in-depth overview can be found on many places in the web. For example ONJava.com[1].

In Java the classloading has several steps. If a class is needed in some way the Thread's current ContextClassLoader is consulted. The contextClassLoader is inherited from the Thread owner, but can be set to anything else. The main-thread's ClassLoader is usually the sun.misc.Launcher$AppClassLoader if not specified otherwise. The ClassLoader itself always has a parent except for the bootstrap ClassLoader. The bootstrap ClassLoader is special because it is written in native Code. Thus it can not be modified by some Java code.

When a Class needs to be loaded the ClassLoader is expected to ask its parent first. A self implemented ClassLoader may behave differently, but it would cause severe side effects. For example if the Integer Class is requested it would be very bad if the self defined ClassLoader loaded this class. The Class object from another ClassLoader would not equal. If the parent ClassLoader does not know how to load a class, it can try to load this class for itself. Therefor it tries to locate the binary representation of this class. This might be as easy as reading a class file from disk, but can also be generated on the fly. After the binary representation has been loaded a call to defineClass creates the Class object for it. The ClassLoader then returns this Class object.

3.1 Custom ClassLoader

The easiest way to perform bytecode manipulation is to subclass a regular ClassLoader. This way it is possible to inspect every class loaded through this loader. If a class that should be loaded matches the defined joinpoint pattern, it is possible to modify it before defineClass is called. In AspectJ this is done by the WeavingURLClassLoader. Because of the parent first paradigm of Classloaders the to be woven classes can not be on the regular class-path. Otherwise the Applauncher ClassLoader would find them first. This would result in unwoven classes because the regular ClassLoader does not know the self defined WeavingURLClassLoader.

```
1 public class TestClass {
2  public static void main(String[] args) throws Exception {
3   ClassLoader parent = ClassLoader.getSystemClassLoader();
4   ClassLoader cl = new WeavingURLClassLoader(parent);
5   Class<?> loadClass = cl.loadClass("net.kbsvn.ltw.HelloClass");
```

[1] http://www.onjava.com/pub/a/onjava/2005/01/26/classloading.html

```
6    ISayHello hc=(ISayHello) loadClass.newInstance();
7    hc.sayHello();
8   }
9  }
10 public interface ISayHello {
11  public void sayHello();
12 }
```

The WeavingURLClassLoader needs some configuration. At first it needs to know where the to be woven classes can be found. This is done using the java property aj.class.path.

```
1  public aspect HelloAspect {
2   after() returning:
3    execution(void net.kbsvn.ltw.HelloClass.sayHello()){
4      System.out.println(" world!");
5   }
6  }
```

In order to know which aspects should be applied it is necessary to provide a configuration file. It is named aop.xml and expected to be in the META-INF directory.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <aspectj>
3   <aspects>
4    <aspect name="net.kbsvn.ltw.HelloAspect"/>
5   </aspects>
6   <weaver/>
7  </aspectj>
```

The configuration file points out one problem. All aspects need to be known in advance. So it is impossible to load an Aspect dynamically. This restriction is caused by the fact that after a class has been defined it is not possible to modify it. Some approaches that will be shown later pass around this restriction.

3.2 Replacing SystemClassLoder

Another way of replacing the application classloader has been introduced with Java 1.4. There it is possible to replace the system class loader (the one that loads the application class). The java property java.system.class.loader defines which class should be used. So when running a program using the AspectJ LTW one needs to specify: -Djava.system.class.loader= org.aspectj.weaver.loadtime.WeavingURLClassLoader plus the configuration for this loader.

```
1  public class TestClass {
```

```
2  public static void main(String[] args) throws Exception {
3   ClassLoader cl = ClassLoader.getSystemClassLoader();
4   System.out.println(cl.getClass());
5   Class<?> loadClass = cl.loadClass("net.kbsvn.ltw.HelloClass");
6   ISayHello hc=(ISayHello) loadClass.newInstance();
7   hc.sayHello();
8  }
9 }
```

The code now looks much clearer. The output of line 4 is ...WeavingURLClassLoader while when using a custom ClassLoader it returns the default ...Launcher$AppClassLoader.

3.2.1 HotSwap

Since java1̇3 the debug interface allowed the replacement of code sections in running JVMs. But the JVM needs to be in a suspended state in order to do that. So what aspectWerkz did was the following. The JVM was started in debug mode and once the main methods entry point was reached, the JVM suspended. In this moment the regular ClassLoader implementation was enriched with some modifications for LTW. The disadvantage is that the JVM needs to run in debug mode all the time, even though the modification was only done once. The speed drop in SUNs client JVM was quite low, unlike the server JVM. Due to the debug mode the server JVM was no longer able to use all its advanced optimisation algorithms. Benchmarks of aspectWerkz measured the drop to be around 20% for long running applications.

3.2.2 Modified bootclasspath

The transparent bootclasspath in aspectWerkz uses the -Xbootclasspath option to provide a modified version of rṫjar. This rṫjar is taken from the regular JVM directory and modified by a small program. The newly created rt.jar has an enriched ClassLoader that performs the LTW. This option is quite JVM independent and compatible to most older JVMs.

4 Online weaving

Online weaving is the modification of joinpoints during runtime. Which means that its control flow is altered after a class has been loaded. The JVM was not build to achieve this. But some very advanced techniques try to remove this limitation in order to enable modifications even at this stage of running.

4.1 Reflective

The reflective approach uses the well known reflection mechanism to manipulate the method invocations. As it is read-only and can only operate on Class objects, it is limited to method invocation and can not redirect calls to other methods. Also it can not modify the class itself nor trace any field accesses.

Despite its limitations it is very well supported in most JVMs and thus has a very high portability. And it is very flexible. Advices can be added and removed in any order but are limited to execute before/after/around advices. To some extend intertype declaration can be used.

4.1.1 Proxy

The class java.lang. reflect .Proxy exists since 1.3. This class can create proxy classes for a set of interfaces during runtime. The generated class will dispatch all method calls to a java.lang. reflect .InvocationHandler. In this InvocationHandler it is possible to invoke the original method using reflection or to do something completely different. The JDK itself uses this class for the remote stub in RMI or CORBA.

This piece of code generates a proxy class that implements the IBusinessLogic interface.

```
interface IBusinessLogic {
    public void foo();
}

public Class<?> createProxy(){
 ClassLoader classLoader = getClass().getClassLoader();
 Class[] interfaces = new Class[]{IBusinessLogic.class};
 return Proxy.getProxyClass(classLoader, interfaces);
}
```

The invocation of createProxy returns a class that decompiles to:

```
public final class $Proxy0 extends Proxy
 implements IBusinessLogic
{
```

```
4   ...
5   public final void foo()
6   {
7    try
8    {
9     super.h.invoke(this, m3, null);
10    return;
11   }
12   catch(Error _ex) { } //might be an incorrect decompilation result
13   catch(Throwable throwable)
14   {
15    throw new UndeclaredThrowableException(throwable);
16   }
17  }
18  ...
19 }
```

Spring uses this approach and calls it interceptor. The primary problem when dealing with proxies is the object-creation. Therefore interceptors can just be used for classes that were created by a factory. As in the case of Spring this is done using an ApplicationContext.

Spring works with beans and XML configurations. In order to define simple before and after advice it is necessary to define those beans and the advices.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <!DOCTYPE beans PUBLIC
3      "-//SPRING//DTD BEAN//EN"
4      "http://www.springframework.org/dtd/spring-beans.dtd">
5
6  <beans>
7  <!-- Bean configuration -->
8  <bean id="businesslogicbean"
9          class="org.springframework.aop.framework.ProxyFactoryBean">
10  <property name="proxyInterfaces">
11   <value>net.kbsvn.bachelorspring.IBusinessLogic</value>
12  </property>
13  <property name="target">
14   <ref local="beanTarget" />
15  </property>
16  <property name="interceptorNames">
17   <list>
18    <value>theTracingBeforeAdvisor</value>
```

```
19     <value>theTracingAfterAdvisor</value>
20    </list>
21   </property>
22  </bean>
23  <!-- Bean Classes -->
24  <bean id="beanTarget"
25          class="net.kbsvn.bachelorspring.BusinessLogic" />
26
27  <!-- Advisor pointcut definition for before advice -->
28  <bean id="theTracingBeforeAdvisor"
29          class="org.springframework.aop.support.RegexpMethodPointcutAdvisor">
30   <property name="advice">
31    <ref local="theTracingBeforeAdvice" />
32   </property>
33   <property name="pattern">
34    <value>.*</value>
35   </property>
36  </bean>
37
38  <!-- Advisor pointcut definition for after advice -->
39  <bean id="theTracingAfterAdvisor"
40          class="org.springframework.aop.support.RegexpMethodPointcutAdvisor">
41   <property name="advice">
42    <ref local="theTracingAfterAdvice" />
43   </property>
44   <property name="pattern">
45    <value>.*</value>
46   </property>
47  </bean>
48
49  <!-- Advice classes -->
50  <bean id="theTracingBeforeAdvice"
51          class="net.kbsvn.bachelorspring.TracingBeforeAdvice" />
52  <bean id="theTracingAfterAdvice"
53          class="net.kbsvn.bachelorspring.TracingAfterAdvice" />
54  </beans>
```

Every bean is defined by a bean element. The id can later be referenced by an application context.

```
1  public static void main(String[] args) throws Exception {
```

```
  // Read the configuration file
  ApplicationContext ctx = new FileSystemXmlApplicationContext(
    "springconfig.xml");
  // Instantiate an object
  IBusinessLogic testObject =
   (IBusinessLogic) ctx.getBean("businesslogicbean");

  testObject.foo();
 }
```

The advices are regular classes that implement the AfterReturningAdvice/MethodBeforeAdvice.

```
public class TracingAfterAdvice implements AfterReturningAdvice {
 public void afterReturning(Object object, Method m, Object[] args,
   Object target) throws Throwable {
  System.out.println("Hello world!"+this.getClass().getName()+")");
 }
}

public class TracingBeforeAdvice implements MethodBeforeAdvice {
 public void before(Method m,Object[] args,Object target)
        throws Throwable{
  System.out.println("Hello world!"+this.getClass().getName()+")");
 }
}
```

Another difficulty is that proxy classes are generated from scratch and thus do not equal the old class in any way. They have different names and Class objects. Another annoyance is the missing support for annotations. Thus they can not be used as a webservice in Java6 or anywhere else where annotations are used.

Compared to most other approaches this approach lacks support for call and field access advices. They can not be accomplished because once the control flow reaches the original method it can not be changed.

The best reason to use proxies is that it is supported in every JVM. And it is possible to advice every instance that was created by a factory or at least has been wrapped at some point in time.

Another advantage is the dynamicness. Advises can be added or removed by just some simple java calls. This is very hard to achieve using other approaches.

4.2 Debugger Interface

Prose is an AOP tooling that allows dynamic AOP. But instead of modifying anything during runtime it uses the JVM Debug Interface (JVMDI). In JVMDI it is possible to set breakpoints to all positions in a class. Prose uses those breakpoints at identified join points. When such a breakpoint is hit an event is generated and interested consumers can execute code. The JVMDI extension has to be written in C code and thus may have portability issues and the JVM needs to run in debug mode all the time which causes some performance penalties when running in server mode.

4.3 ByteCode modifications

This is the most difficult method, and the most powerful at the same time. But it can not be done without support from JVM.

4.3.1 Java Agent

Since Java5 there is a new API available the JVM Tool Interface (JVMTI). It succeeds the JVM Profiler Interface (JVMPI) and the JVM Debug Interface (JVMDI). Libraries that use this API are called agents. Those agents work in the same way as JNI libraries. So they have a platform dependant part which restricts them to certain platforms. Using this API it is possible to get byte array representations of loaded classes and thereby modify them. This is called bytecode instrumentation (BCI). But the modifications are limited to non structural changes. Which means that only method bodies can be modified. Intertype declarations in aspects are thus not supported. Another limitation for java5 is that only one agent with BCI can be active. This restriction has been removed in Java6.

4.3.2 Deep JVM support

Steamloom has a very advanced JVM support. Due to its ease of development the only supported JVM at the moment is the Jikes research JVM (RVM). This RVM is written in Java, self hosted, and well documented. This makes it easy for researchers to modify it and test new extensions that require JVM support. The RVM features a Just in time (JIT) compiler or to be more exact it just has a JIT-compiler and no Interpreter. This eases the work as well because only one component needs to be maintained.

When the RVM loads a class the bytecode of a method is replaced by a lazy compilation stub. Upon first invocation this lazy compiling stubs triggers a recompilation using the baseline compiler which generates native code. The generated code of this compiler is not optimized and thus not very efficient. When a method is identified as hotspot it gets recompiled using an optimising compiler. There are two different optimising compilers. The adaptive

optimisation system (AOS) compiler is built on top of the normal optimising compiler and performs profiling to further optimise code. Steamloom extends the AOS compiler. A toolkit called bytecode augmentation toolkit (BAT)[1] is used to apply advices to bytecode like most LTWs. The biggest difference is that the internal class representation hast been modified to suite the needs of the BAT. Thus it does not need to transform it into its own representation and later back to classfile represenation. When an advice needs to be undeployed the method is marked for recompilation. This regenerates the code from the bytecode. All optimisations are reapplied. A good description of steamloom can be found in Bockisch u. a. (2004).

Steamloom uses plain java to describe aspects.

```
1  public class IopTest {
2   public static void main(String[] args) throws Exception {
3    IopTest test = new IopTest();
4    IopAspect.setupAspect().activate();
5    test.run();
6   }
7
8   public void run() {
9    System.out.println("about to call inner()...");
10   inner();
11   System.out.println("returned from inner()");
12  }
13
14  public void inner() {
15   System.out.println("now running inner()");
16  }
17 }
```

The run method should be advised by an after advice. The aspect looks like this:

```
1  public class IopAspect {
2   public void advice(IopTest diz) {
3    System.out.println("now running the advice");
4    System.out.println("this: " + diz.toString());
5   }
6
7   public static Aspect setupAspect() throws Exception {
8    EntityFactory f = SteamloomEntityFactory.instance();
9
```

[1] http://www.st.informatik.tu-darmstadt.de/pages/projects/BAT/

```
10    QueryLanguage ql = f.createQueryLanguage();
11
12    Aspect aspect = f.createAspect();
13
14    JoinPointSelector jps =
15     ql.evaluate("execution(void IopTest.inner())");
16
17    //Advice advice = f.createBeforeAdvice();
18    Advice advice = f.createAfterAdvice();
19
20    MethodInvocation action = (MethodInvocation) advice.getAction();
21    action.appendAction(f.createAddThisParameterAction());
22    action.appendAction((MethodInvocation) advice);
23
24    action.setMethod(IopAspect.class
25     .getDeclaredMethod("advice", new Class[]{IopTest.class}));
26    action.setTarget(new IopAspect());
27
28    advice.setAction(action);
29
30    SelectorAdviceBinding binding = f.createSelectorAdviceBinding();
31    BindingSelector bs = (BindingSelector) binding;
32    bs.addAdvice(advice);
33    binding.setJoinPointSelector(jps);
34    binding.setBindingSelector(bs);
35
36    aspect.addSelectorAdviceBinding(binding);
37
38    return aspect;
39   }
40  }
```

The biggest advantage of this approach is that some special bytecode instructions can be used. One of those is the advice instance table(AIT) Haupt und Mezini (2005). In usual bytecode weavers a static method is called in order to lookup the instance of the advice. The static method then mostly uses a hashmap of some kind to resolve perthis[1] advices. In Steamloom the AIT is reduced to a single assembler instruction. The performance benefit of this approach is significant. Other optimization techniques are used for an enhanced cflow

[1] a perthis advice is an advice that maps one instance of an advice to exactly one instance of the adviced class.

construct Bockisch u. a. (2006). Usually in a cflow all possible execution flows are searched and special code is inserted to check whether the cflow matches or not. Steamloom just uses a long to set some bits. The cflow check thus reduces to a simple bit check and the structure to store this information is a simple long.

5 Comparison

For a closed world environment where all code is known in advance the choice of weaving is quite easy. The offline weaving has the best tooling support around and is well tested. It also features the best performance in most cases. But regarding cflow there are some interesting papers that show that JVM support may yield a better result Bockisch u. a. (2006).

The downside of offline weaving is that 3rd party libraries that require weaving have to be known in advance. In an application like Eclipse it is very unlikely to know all 3rd party libraries in advance. And the application has to be bundled with that particular library which contradicts the idea of shared plugins.

Load time weaving can weave most classes. Even those loaded from 3rd party libraries. But the setup requires that those classes are loaded through a special classloader. In a container enviroment such as J2EE this is not trivial to achieve[1] as the classloader may change during the lifecycle of a container.

As classes are woven during the load time in load time weaving it may take more time to load an application. Thus it is not recommended for applications that are frequently started. Such as dekstop applications. In big enviroments the double book keeping and parsing of class files can take a considerable amount of time. This can make an application feel slow for an user. When used on a server the load time is not so important. As the server is unlikely to restart frequently.

Another restriction of load time weaving is that all aspects need to be known in advance. As for AspectJ they have to be declared in a xml file. Aspects which are loaded at a later point in time are note woven correctly as the classes they advise might be already loaded.

Online weaving is the most advanced weaving mechanism and has virtually no restrictions. Every loaded class can be woven at any point in time. This level of flexibility comes with the downside of no tooling support. Is is hard to keep track of classes that are woven. The debugging support is also very restricted.

[1] http://www.eclipse.org/aspectj/doc/released/faq.php#q:aspectjandj2ee

6 Conclusion

Deploying an aspect enriched application can become very complicated. Especially when it is used in a J2EE enviroment or alike. JBoss for example features its own weaving agent. Thus restricts the developer to use just this aop approach. In some enviroments it is not possible to control the JVM the application is running in. Therefor agents can not be used.

At the moment there are a lot of different AOP languages that all have their own weaving technology. This makes it very hard to exchange/reuse existing libraries as the used weaver might have produced something incompatible. Even just this argument would justify a JSR itself. But as steamloom shows, weaving can greatly benefit from JVM support.

6.1 Bytecode modifications

Nearly all weavers are performing bytecode modification in some way. While bytecode modifications have some advantages, they are quite expensive to perform. The classfile is meant for persistent storage of Class objects. Whenever a modification of bytecode is performed, it is the classfile representation that gets modified. This means that the classfile is read to an object representation, then transformed and afterwards written back in classfile representation. So this is like modifying an XML-File in its byte representation instead of working with the object-representation of this file and writing it back at the end of the application. This is what makes bytecode modifications so expensive.

The solution is quite simple. The JVM reads the classfile into an internal object representation. This Object representation could then fairly easy modified. The advantages of making the JVM aspect-aware can create a lot of interesting features.

At first, modifying bytecode is hard to track. If a weaver performs some wrapping of a method and another will do the same, the first weaver might no longer be able to unweave his method. So, some bytecode modifications are irreversible, while the object representation could carry the necessary information to undo the performed transformation at any later point in time.

With bytecode modification another problem arises. Debugging such woven code becomes harder because there is no source code available for the woven bytecode. Thus the debugger cannot show the call to another method correctly to the user. The debugger should be able to show that a method is advised and enable or disable this advice.

When it comes to reflection call, get and set based advices are ineffective as the calling point can not be found. Native or abstract methods can not be advised by execution advices at all. Also classes loaded by the bootstrap classloader are not weavable by loadtime weavers.

For most online weavers the JVM needs to be in debug modus which allows bytecode instrumenation. In Suns Client JVM the debug modus is said to feature a zero performance penality. But the Server JVM degrades in performance because some advanced optimization

techniques are disabled in debuging mode. Another limitation for Java 5 is that only 1 agent is allowed to perform bytecode instrumenation. This limitation has been removed in Java 6.

Bytecode manipulation ruins all efforts of optimizations that have been taken till the moment of manipulation. The bytecode needs to be recompiled and optimized. As most optimizations only occur after a certain amount of invocations of a statement it can take some time until the code is in the same optimizations level as before.

6.2 Make aspects exchangeable

Another downside of current AOP implementations is that once written aspects are only useful with their own weaving technology. This makes exchangeability and thus reusability of aspects quite low.

6.3 JSR for JVM support

It has been shown that the current weaving technologies have several disadvantages. To solve this problem a JSR would have several benefits. Developers could use the same AOP language for every supported JVM, the feature set would be the same on every JVM. Performance penalties for dynamic AOP can be significantly lowered.

At the moment there is no easy way to express a proceed statement in Java. But chances are high that Java 7 will feature closures and function pointers. Those could easily be used to create a nice Java API.

6.3.1 What features would be possible to have in such a JSR?

Advices should be deployable/undeployable at runtime. The usecase for such a dynamic aop approach is for example debugging purposes. Imagine having a big J2EE environment which serves several thousand clients at the same time. And every once and a while a customer suffers strange unreproducible problems. What one might do would be inserting log statements into every possible point of failure. But this would mean recompiling the application and redeploy it which also would possibly mean having a downtime. The log statements lower the overall performance. The application would become less responsible. The log statements are sqattered around the code. And after some time the bug might have been found. After removing the log statements the application needs another compile/deploy cycle.

Dynamic AOP might help in those situations. Log statements can be added/removed without the need to recompile/redeploy the whole application. The performance degradation could be handled much better and the bug could be faster identified. The usual redployment also means that the optimization level of the current application is lost while JVM supported dynamic AOP could sustain the level.

Support for debugging. With JVM support it is easy to identify advices and thus show them to the user in a correct way. It would also be possible to enable/disable those advices for debugging purposes.

Exception encapsulated advices. At the moment most AOP languages can not differentiate between an exception that was caused in the advice by accident or by purpose. Let's assume that an logging facility has been applied to an application. Usually logging is not meant to change the control flow of a program. But if the logging contains a bug that throws a NullPointerException the whole control flow would become very different. Even though logging was not intended to have any impact on it. Applied advices should thus be encapsulated with only wrapped Exceptions being allowed. Here an example:

```
1  public class TestClass {
2   private static String helloWorld;
3
4   public static void main(String[] args) {
5    doSomething(helloWorld);
6   }
7
8   private static void doSomething(String arg) {
9    System.out.println(arg);
10  }
11 }
```

The doSomething method is now advised by some very simple logging mechanism:

```
1  public aspect SimpleLogger {
2   before(String arg) :
3    execution(void *.doSomething(String))
4    && args(arg) {
5      System.out.println(arg.toString());
6   }
7  }
```

Without logging the application is running fine. But the advice is a bit too lazy and does not check the arguments for null. This results in a quite unexpected NullPointerException. The logging mechanism has suddenly changed the control flow of the program.

The applied advice should thus be encapsulated in a try/catch block that reports any unwrapped Exception to the AdviceHandler that was responsible for applying this advice. In code this might look like this:

```
1   private static void doSomething(String arg) {
2    try {
3     SimpleLogger.ajc$before$SimpleLogger$1$eff035ad(arg);
```

```
4   } catch (Throwable t){
5    if (t instanceof EncapsulatedException) {
6     throw t.getCause();
7    }
8    AdviceHandler.reportException(t);
9   }
10  System.out.println(arg);
11 }
```

This would reduce unwanted side-effects of advices. And therewith make aspects easier to learn.

Support for reflection. In modern frameworks like Spring reflection plays a major role. But unfortunately it is uncovered by call, get and set pointcuts. While reflection support for fields just needs some slight modifications in the class Field, supporting call statements is more difficult. The point where invoke is called on a Method object would need to be replaced with a version that provides some extra context information. Like the current Method object. In a class that performs a reflective call a static constructor has to be generated. This static constructor would fetch the Method objects for methods were reflection is used.

```
1  public class ReflectiveTest {
2   public static void main(String[] args) throws Exception {
3    Method method =
4     ReflectiveTest.class.getMethod("log", new Class[0]);
5    method.invoke(null, new Object[0]);
6   }
7
8   public static void log(){
9    System.out.println("I'm runnning");
10  }
11 }
```

The class ReflectiveTest would be transformed to:

```
1  public class ReflectiveTest {
2   static Method m1;
3   static {
4    try {
5     m1=ReflectiveTest.class
6         .getDeclaredMethod("main", String[].class);
7    } catch (SecurityException e) {
8    //find a useful solution here
9    } catch (NoSuchMethodException e) {
```

```
10    //find a useful solution here
11    }
12   }
13
14   public static void main(String[] args) throws Exception {
15    Method method =
16     ReflectiveTest.class.getMethod("log", new Class[0]);
17    method.invoke(m1, null, new Object[0]);
18   }
19
20   public static void log(){
21    System.out.println("I'm runnning");
22   }
23  }
```

The invoke method than has a chance to lookup whether an advice has to be run for the combination of m1 and method.

Support for bodyless methods. Methods that are either abstract or native do not have bytecode that could be woven. This makes it at the moment impossible to apply execution advices to those methods. The attribute based weaving process can handle those cases.

6.3.2 Attribute based weaving

In a classfile attributes are structures that can be attached to certain elements of a classfile[1]. Annotations for example are very well known attributes. Others are line information, synthetic, deprecated etc. Attributes that are not known to a JVM are ignored. This is one of the first advantages. A JVM that does not support aspects can still run the code without advices. On the other hand a aspect aware JVM can choose how advices are used. A very simple implementation for example could use those attributes for load time weaving, while a more in depth implementation could use it for direct native code optimization. Unlike the current weaving mechanism the JVM can choose the way it is implemented by itself. This makes it very easy for JVM writers to implement such a JSR.

Writing the compiler and tooling support becomes easier as well. The attributes allow an easy distinction between advice and regular code and may contain additional information that further ease the development of such tools.

Assume an advice has to be applied to all implementations of a certain interface. For example Comparable#compareTo(). The compiler would simply attach the advice to the method compareTo. And when any class that implements Comparable is loaded, the class Compara-

[1] http://java.sun.com/docs/books/jvms/second_edition/html/ClassFile.doc.html

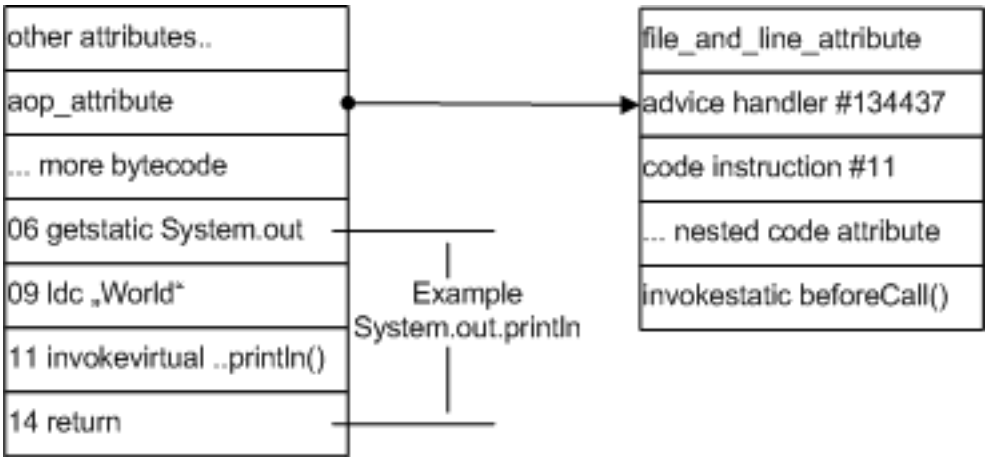

Abbildung 1: Simple call example

ble is looked up for resolution[1]. The JVM then checks if a attribute is attached to this method and “uses” it. With this way Intertype declarations could be done as well. If the method is not implemented yet, a default implementation from an attribute could be used.

A call advice would be applied by attaching the attribute to the bytecode position where the call occurs. When the class is loaded, the attribute is evaluated and the code is woven into the correct place.

6.3.3 Security

Current aspect languages, especially AspectJ, that use bytecode modification can be deployed to every part of a class. This includes private methods as well as fields. This contradicts the idea of information hiding which is considered a fundamental requirement of modern maintainable software architectures. Another problem that arises with bytecode modifications are 3rd party libraries. At the moment it is possible to weave into those libraries which are probably shipped as jar. The license usually bundles to the unmodified library. So when such a library is woven with advices it violates the license. From a realistic point of view the only way to protect intellectual property (IP) produced in Java is not to ship it to a customer and run it as a webservice or other server side tool. This of course is not always possible. So when code has to be deployed to a customer there have to be security mechanism in place that protect your IP. Beside using obfuscators this includes a mechanism in JVM that checks for signatures to garantee that the unmodified class is loaded. This mechanism is already in place for most JVMs. Once such a library is loaded it can be modified using current weaving technology. So another mechanism has to check whether a weaver is allowed to modify a class. There are 2 possible approaches. Excplicity allow a class for weaving, or explicitly

[1] http://java.sun.com/docs/books/jls/third_edition/html/execution.html

deny weaving. From an information hidingParnas (1972) point of view it would be wise to explicity allow a class for weaving. It has the same effect as raising the friendly modifier that is in place by default for method, fields etc. It restricts the access to clases that are in the same package. Even though this restriction is quite weak it is a good tradeoff of usability vs. information hiding for most programs.

Nomenclature

Advice describes a certain function, method or procedure that is to be applied at a given join point of a program.

AOP Aspect oriented programming a programming paradigm that focuses on separation of concern.

BCI Bytecode instrumentation the ability to communicate with the JVM in order to retrieve the bytecode of a class. The bytecode can also be modified to some extend.

ByteCode is the form of instructions that the JVM executes.

Concern in AOP a concern is seen as a part of a software that does have a distinct function that no other part also has. In mathematical terms one might say that the functionality of a concern needs to be disjoint to other functionalities of a software.

CP Constant Pool block in classfile that contains string constants. Special coded strings are used for references or signature.

Joinpoint a point in the flow of a program. An example joinpoint is the beginning execution of a method, constructor.

LTW Load time weaving the process of weaving a class when it is about to be loaded.

Offline weaving the weaving of a .class file.

Online weaving the ability to weave a class at any point in time.

Pointcut a set of joinpoints.

Weaving the act of applying an advice to a joinpoint. This is mostly done by manipulating the bytecode, but not limited to.

Literatur

[Bockisch u. a. 2004] BOCKISCH, Christoph ; HAUPT, Michael ; MEZINI, Mira ; OSTERMANN, Klaus: Virtual machine support for dynamic join points. In: LIEBERHERR, Karl (Hrsg.): *Proc. 3rd Int' Conf. on Aspect-Oriented Software Development (AOSD-2004)*, ACM Press, März 2004, S. 83–92

[Bockisch u. a. 2006] BOCKISCH, Christoph ; KANTHAK, Sebastian ; HAUPT, Michael ; ARNOLD, Matthew ; MEZINI, Mira: Efficient control flow quantification. In: *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*. New York, NY, USA : ACM Press, 2006, S. 125–138. – ISBN 1-59593-348-4

[Haupt und Mezini 2005] HAUPT, M. ; MEZINI, M.: Virtual Machine Support for Aspects with Advice Instance Tables. In: *L'Objet* 11 (2005), Nr. 3, S. 9–30

[Lai u. a. 2000] LAI, Albert ; MURPHY, Gail C. ; WALKER, Robert J.: Separating Concerns with HyperJ: An Experience Report. In: TARR, Peri (Hrsg.) ; FINKELSTEIN, Anthony (Hrsg.) ; HARRISON, William (Hrsg.) ; NUSEIBEH, Bashar (Hrsg.) ; OSSHER, Harold (Hrsg.) ; PERRY, Dewayne (Hrsg.): *Workshop on Multi-Dimensional Separation of Concerns in Software Engineering (ICSE 2000)*, URL `http://www.research.ibm.com/hyperspace/workshops/icse2000/Papers/lai.pdf`, Juni 2000

[Parnas 1972] PARNAS, D. L.: On the criteria to be used in decomposing systems into modules. In: *Commun. ACM* 15 (1972), Nr. 12, S. 1053–1058. – ISSN 0001-0782

Hiermit versichere ich, dass ich die vorliegende Arbeit im Sinne der Prüfungsordnung nach §22(4) bzw.§24(4) ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, 22. Mai 2007 Karsten Becker