AspectWerkz – dynamic AOP for Java

Jonas Bonér
jonas@codehaus.org

ABSTRACT

This paper will introduce a new aspect-oriented framework for the Java language called *AspectWerkz*. The *AspectWerkz* framework is tailored for dynamic aspect-oriented programming. It has been designed for simplicity and ease of use in real-world applications, both J2EE and regular Java based. *AspectWerkz* utilizes load time bytecode modification to weave classes at load time. It hooks in and weaves classes loaded by any class loader except the bootstrap class loader. It is high-performant and has a rich join point model. Aspects, advices and introductions are written in plain Java and the target classes can be regular Java objects. It is dynamic in the sense that it is possible to add, remove and restructure advices as well as swapping the implementation of the introductions at runtime. Aspects can be defined using either XML or runtime attributes.

Keywords

Aspect-oriented programming, separation of concerns, dynamic AOP, runtime weaving, AspectWerkz

INTRODUCTION

Aspect-oriented software development (AOSD) [25] is gaining popularity. More and more people start to see the benefit in *separation of concerns*, start to see the shortcomings of object-oriented programming and design.

One of the obstacles for it too be fully accepted by the masses is that most of the *Aspect-Oriented Programming (AOP)* [4] implementations so far have been too complex and academic. The learning curve has been too steep. There has been a need for a simple yet powerful alternative that meets the needs of real-world application development. Another thing that has proved itself to be very useful when designing real-world applications is *dynamic AOP*, i.e. dynamic weaving of aspects as well as dynamic deployment and redeployment of aspects.

These needs have led to the implementation of the *AspectWerkz* framework for aspect-oriented programming. In this framework we have tried to design a definition model that is simple and intuitive but at the same expressive and powerful. Focus has been put on designing an aspect model that is dynamic, in which the aspects, advices and introductions are loosely coupled and are easy to add, remove and restructure at runtime. The *AspectWerkz* framework has also been designed to allow dynamic weaving in which the target classes can be weaved at load time and can therefore bring AOP to any Java [16] / J2EE [24] application in a very non-intrusive way.

THE ASPECTWERKZ AOP FRAMEWORK

Core constructs

Join points

Join points are well-defined points in the program flow. In *AspectWerkz* the join point construct is represented by the `JoinPoint` class. A JoinPoint instance is available in the advices and can be used to introspect and retrieve RTTI (Runtime Type Information) about the class, method and field at the current join point. It is also possible to change the non-static parts of the RTTI.

A construct called `JoinPointController` was also added, which allows the user to control the execution flow of the advices based on custom defined constraints and rules, rules that can be changed at runtime. This construct has showed itself very valuable when it comes to handle:

- inter-aspect compatibility
- inter-aspect dependency
- aspect redundancy

Pointcuts

One of the most important features of an AOP framework is the possibility to pick out join points. This need led to the implementation of a fine-grained pattern language, a pattern language that has borrowed a lot from the syntax of *AspectJ* [4].

A pattern normally consists of a combination of a class and a method pattern or a class and a field pattern and are build up like this:

- `<return_type> <package>.<class>.<method>(<parameter_types>)`
- `<field_type> <package>.<class>.<field>`

Examples:

- `String foo.Bar.method(int, Object)`
- `foo..*.*(..)`
- `int foo.Bar.m_field`
- `foo.Bar.*`

It also allows selection of all subtypes of a type using the '+' wildcard. So, while `foo.Bar.*(..)` picks out all methods for an instance of type `Bar` and only of type `Bar`, `foo.Bar+.*(..)` picks out all methods for an instance of any subtype of `Bar` (including `Bar` itself) . A type name pattern or subtype pattern can be followed by one or more sets of square brackets to make array type patterns. So `Object[]` is an array type pattern, and so is `foo.bar.*[][]`.

Pointcut composition is the possibility of composing new pointcuts out of existing pointcuts. So that complex pointcuts can be created out of trivial ones. *AspectWerkz* allows pointcuts to be built up using an expression of other pointcuts. The logical operators supported are, '&&', '`||`' and '`!`'. For example pointcuts could be built up like this:

- `(pc1 || pc2) && pc3`
- `!pc1 && pc2`

Advices

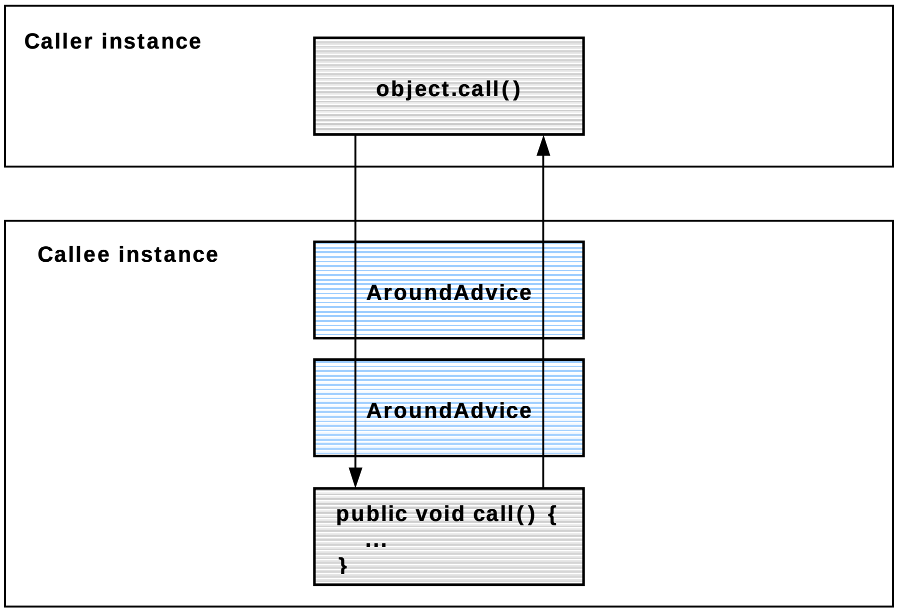
AspectWerkz supports the three main types of advices, as well as some variations on these types:

- **Around advice** – is invoked around the join point.
- **Before advice** – is invoked before the join point.
- **After advice** – is invoked after the join point.

Advices are implemented through a method that takes a `JoinPoint` instance as the only argument and that returns a `java.lang.Object` (the before and after advices returns void). The return value is the return value from the target method invocation, if the return value is a primitive it is wrapped in its object representation. Invoking the method `proceed()` on the `JoinPoint` instance passes on the invocation to the next advice in the chain or if there are no more advices; the target method. Here is an example of an around advice:

```
public Object execute(JoinPoint joinPoint) throws Throwable {
    // do some things before the invocation
    Object result = joinPoint.proceed();
    // do some things after the invocation
    return result;
}
```

The picture below illustrates the control flow of a method advised with two around advices:



AspectWerkz also allows the possibility of passing parameters to advices. This has showed itself to be very convenient in certain situations since it makes the advices much more reusable, the same advice can be reused in different contexts with different configurations. There is no limit in how many parameters can be passed to an advice. All that is needed to pass a parameter to the advice is simply to add a parameter attribute to the advice definition. The parameters can then be accessed at runtime like this:

```
String timeOut = getParameter("timeout");
```

Introductions

An introduction makes it possible to extend a class with a new interface, super class, methods or fields. In *AspectWerkz*, introductions are implemented using the concept of *Mixins* [14]. *Mixins* are a way of faking multiple inheritance, methods and fields can be ‘mixed in’, e.g. added to the class.

Both the interface and the implementation extensions are just regular Java [16] interfaces and classes. There is no need to have them implement a specific interface or extend a certain class. When defining an *interface introduction* you only have to specify the interface, but when introducing a new implementation to a class, you must specify both the implementation class and a matching interface. This is needed since if you don't specify an interface that the client can cast the target instance to, the introduced implementation will not be accessible.

Example of a simple *Mixin* and how it is used in by the client:

```
// mixin interface
public interface Hello {
    String sayHello();
}

// mixin implementation
public class HelloImpl implements Hello {
    String sayHello() {
        return “Hello!”;
    }
}

...
// if the mixin Hello has been applied to class Foo
// then we can invoke the mixin’s method on an instance
// of Foo like this:
String greeting = ((Hello)instanceOfFoo).sayHello();
...
```

Aspects

The *Aspect* is *AspectWerkz's* unit of modularity for crosscutting concerns it is defined in terms of pointcuts, advices and introductions.

AspectWerkz supports abstract aspects. Aspect inheritance works pretty much like regular class inheritance. A common idiom is to define the generic parts, e.g. the advices and introductions in the abstract aspect and define the specific parts, e.g. the pointcuts in the concrete aspect which inherits the abstract aspect. Using this idiom makes it possible to create reusable aspect components and libraries.

Dynamic aspect model

The *AspectWerkz* framework makes heavy use of loose coupling and delegation to implement a dynamic aspect model. Aspects, advices and introductions are registered in the system and are referenced using handles. This design allows the aspects, advices and introductions to live in different virtual memory spaces with different life-cycles. It also makes it easy to add, remove and restructure the aspects, advices and introductions at runtime without having to reload or reweave the target classes.

When for example an advice is added to a join point, the advice itself is not weaved into the class but a `JoinPoint` field is weaved in, defining the specific join point and a method call to the method `proceed()` in the `JoinPoint` class (at the join point). When the target class is instantiated the `JoinPoint` instance registers itself in the system and is given the handles to the advices bound to the join point it defines. The advices are then invoked when the join point is reached and the method `proceed()` is invoked.

Introducing this layer of indirection gives of course a small performance penalty, but since this initialization phase only occurs once; when the class is loaded (static join points) or when an instance of the class is created (member join points), the total overhead for the initialization phase is not very high and the performance penalty for invoking an advised join point is acceptable for most systems. For example, the overhead for a call to an advised method without any advices is ~0.00013 ms/call and the overhead of invoking a method with one advice is ~0.00029 ms/call (running Java 1.4 on a Pentium4 1.6 MHz box).

The benefit of this indirection and the use of handles is that it makes it very easy to make structural changes to the aspects, advices and introductions at runtime. The operations currently supported are:

- Swapping the implementation of a *Mixin* [14]:

  ```
  SystemLoader.getSystem(id).getMixin(name).
       swapImplementation(newClassName);
  ```

- Removing advices from the join points identified by a pointcut:

  ```
  List pointcuts = SystemLoader.getSystem(id).
          getAspect(aspectName).
          getPointcuts(className, methodMetaData);
  for (Iterator it = pointcuts.iterator(); it.hasNext();) {
      Pointcut pointcut = (Pointcut)it.next();
      if (pointcut.hasAdvice(adviceName)) {
          pointcut.removeAdvice(adviceName);
      }
  }
  ```

- Loading a completely new advice or aspect and add it to the system.
- Adding an advice to a certain pointcut.
- Reordering the advices at a certain pointcut.

What the model is currently lacking is the possibility of redefining the pointcuts at runtime. It is now limited to do dynamic redefinitions at join points that have already been weaved. This is a limitation that will be addressed in the future in the JVMTI API [6].

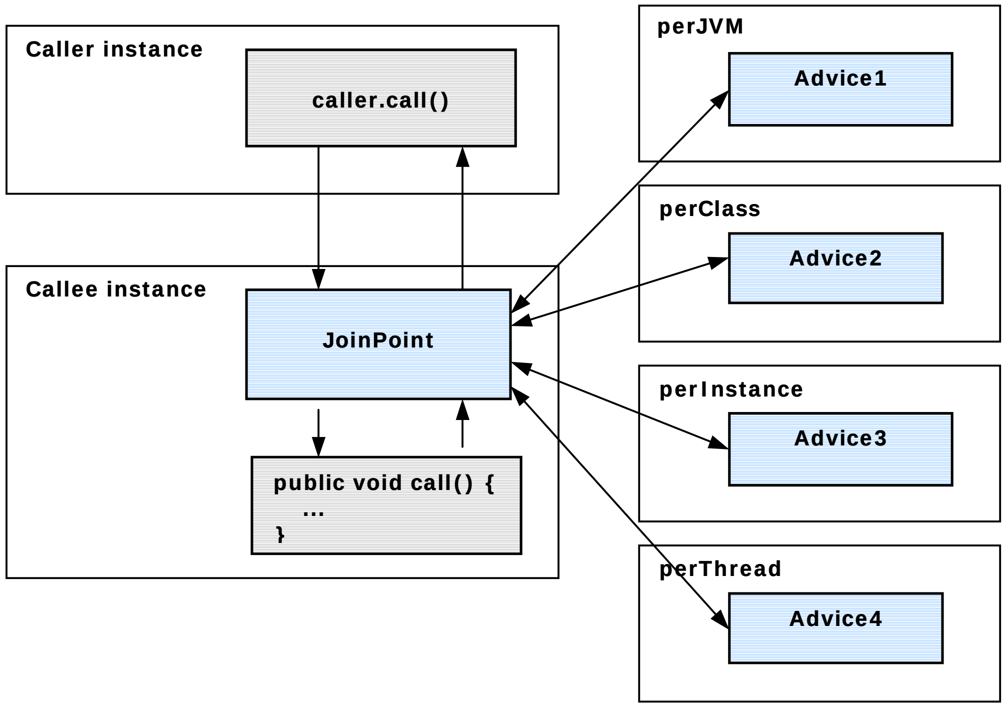
One of the early requirements was to be able to set the scope for the advices and introductions, allow them to have different life-cycles. This led to the concept of *deployment models.*

AspectWerkz supports four different *deployment models*:

- **perJVM** - one sole instance per JVM. Basically the same thing as a singleton class.
- **perClass** - one instance per class.
- **perInstance** - one instance per class instance.
- **perThread** - one instance per thread.

What this means is that an introduction with the *deployment model* set to `perThread` will live and die with the thread, e.g. it will function like a thread local, but if it is deployed with the *deployment model* `perJVM` it will act like a singleton. This can lead to some very interesting new semantics in the design.

Here is an illustration of how advices with different deployment models live in different memory spaces with their own life-cycles:



Weaving

Weaving, instrumentation of classes, is when the classes are enhanced with the advices and introductions (weaved in). *AspectWerkz* allows dynamic and static weaving and weaves the classes at bytecode level currently using the bytecode modification library *BCEL* [15].

Dynamic weaving

AspectWerkz allows dynamic weaving by enhancing the core Java [16] class loading architecture to allow weaving just before a class is loaded in the JVM. It solves this by hooking in directly after the bootstrap class loader and can perform bytecode transformations on classes loaded by all the preceding class loaders, making it compatible with J2EE [24] environments and its complex class loader hierarchies with EAR, EJB-JAR and WAR deployments. This overhead is very small when there are no aspects defined and occurs only once per class per class loader hierarchy when aspects are bound.

Parts of this architecture will be simplified when the JVMTI API [6] is released with Java 1.5. JVMTI will provide a full featured Java API for class redefinition at runtime, allowing pointcut redefinition. It will also provide an in process bytecode weaving hook which will allow weaving at load time with a minimal overhead.

The hook mechanism has been tested and verified to work on Sun's HotSpot JVM [19] as well as custom JVM implementations like IBM's JVM and BEA's JRockit [20], and is available as a stand-alone library that can be used by other tools. It is for example currently used by the code generation library CGlib [18] .

AspectWerkz currently supports five different types of load time weaving. These types can be divided into two groups: *HotSwap* and *bootclasspath.*

HotSwap

The first group is based on the Java Platform Debugger Architecture (JPDA) [5] architecture. JPDA is the debugging support for the Java 2 Platform. JPDA provides the infrastructure needed to build end-user debugger applications. New in the 1.4 release was the support for class redefinition at runtime through HotSwap: A first tiny JVM launches your target application in a second JVM. The second JVM is launched with `-Xdebug` and `-Xrunjdwp` options to allow HotSwap. The classes in the target application running in the second JVM can then be redefined from the application running in the first tiny JVM.

AspectWerkz redefines the `java.lang.ClassLoader` and adds a transformation hook for the classes which allows transformation of the classes right before that they have been loaded.

In its early phases *AspectWerkz* used *JMangler* [9] as the basis for its hook and class transformation mechanism. *JMangler* is a very innovative solution and pioneered much of the usage of the JPDA API [5] to provide a JVM wide transformation hook. But having full control over the low level architecture has been a crucial point and *AspectWerkz* has gained a lot by implementing a customized solution. Since this point the concept and implementation has been extended in various ways and does now (among many other things) provide a native, in process, hook through JVMPI [10] as well as remote HotSwap.

The current implementation supports three ways of enabling class redefinition through HotSwap:

- **Regular HotSwap -** The first JVM hooks *AspectWerkz* in the second one just before the main class and all the applications dependencies gets loaded, and then connects to the *stdout, stderr* and *stdin* stream of the second JVM to make them appear as usual through the first JVM.
- **Native HotSwap -** Instead of using a launching JVM that redefines the class loader a native C JVM extension running in the target application VM handles

the replacement of the class loader by the enhanced one at VM initialization. This extension uses the JVMPI API [10].

- **Remote HotSwap -** The application VM is launched in suspended mode. The replacement of the class loader is done through a separate manual process, which can easily be scripted.

The problem with these approaches is that they only work on 1.4.x compatible JVMs.

Another problem with using HotSwap is that it requires the JVM to start up in debug mode (with the `-Xdebug` option). This is needed to allow recording of method call dependencies, in order to reorganize them when a method is replaced at runtime. With the `-client` JVM option, the overhead of the `-Xdebug` is almost zero on Java 1.4. With the `-server` JVM option, the overhead is stated to be from five to ten percent since code optimizations cannot be performed. A new JSR, JSR-163 [6], which specifies the JVMTI API, is in progress to provide an enhanced native class redefinition architecture that will not require the JVM to run with the `–Xdebug` flag. Note that it is possible to use HotSwap under the 1.4.0 release of Sun's JVM **without** setting the `-Xdebug` flag due to an omission made by Sun Microsystems.

Bootclasspath

The second group is based on the possibility to modify the bootclasspath using the `-Xbootclasspath/p:` option. This option lets you either prepend libraries to the boot class path.

There are two ways you can enable class redefinition through the use of the bootclasspath option:

- **Transparent bootclasspath -** At startup an enhanced class loader is transparently created and put in the target application VM's bootclasspath through another tiny JVM.
- **Prepared bootclasspath -** An enhanced class loader is build and packaged as a JAR file in a first separate manual process, which can easily be scripted. The application's JVM is launched with options to use this enhanced class loader.

This approach works both on 1.3.x and 1.4.x compatible JVMs.

Static weaving

AspectWerkz also support static weaving using a post-processor. This will add an extra compilation step to the build process but the application can be started as normal, without any special requirements and options apart from some dependency libraries. This option is useful for users that do not have full control of the startup process or do not think that the extra overhead added when transforming the classes at runtime is acceptable.

Definition models

AspectWerkz currently supports definition through *XML* [17] and *runtime attributes* [12].

One of the main objectives when designing *AspectWerkz* was to come up with a definition format that felt intuitive for the users of the framework. Since a language extension (like the approach of *AspectJ* [4] and *PROSE* [2]) would have added too much complexity, the options where: plain Java syntax, XML and JavaDoc [26] tags. Among these, XML was the natural choice. XML is widely used in the Java/J2EE [24] community, it has a good tool support etc.

Initially, *AspectWerkz* only supported the XML definition model. This model has many advantages; it is using a well-known and commonly understood format (XML), it can easily be validated, it is easily parsed at runtime etc. But after using it for a while some limitations started to become clear. One major drawback was the separation of meta-data (definition) and implementation. Another problem was that it separated the implementation parts of the aspect (advices and introductions) into different classes. For example a simple aspect that had three advices and two introductions, required six files since the three advices had to be in separate classes as well as the two introductions and on top of that, a definition file tying all the parts together was needed.

To solve this problem a new definition model was designed and implemented. This model makes use of the concept of *runtime attributes* [12] (decorating classes, methods and fields with meta-data in bytecode) to bring the definitions and the implementations together into in single unit, the aspect. See the '*Runtime attribute definition model*' section for a detailed outline.

XML definition model

The *XML definition model* went through some major changes until it reached its current state. One of the problems with the early versions where that the AOP constructs (pointcuts, advices, introductions and advices) where not orthogonal. Another problem was that it did not support *pointcut composition*. Both of these are extremely important issues that could cripple the usability of the framework and has now been solved. The underlying model now has the same essence as the *AspectJ* model [4].

Advice definition

The advices are defined by first specifying the name of the advice. This name has to be unique and will work as a handle to the advice to be used when referencing the advice in the system. Second the class representing the advice needs to be specified and last an optional attribute specifying the *deployment model* to use. If no *deployment model* is defined the default `perJVM` (singleton) will be used. Example of an advice definition:

```
<advice-def name="advices/caching"
            class="advices.CachingAdvice"
            deployment-model="perInstance">
</advice>
```

There is also a possibility of defining a stack or chain of advices called *advice stack*. These let you define a stack with advices that you can refer as a unit. The order of the advices in the stack is the same order as they will be executed in. *Advice stacks* can come in very handy when there is a bunch of advices that logically belongs together and are used in the same order at many places or when there is a need for ensuring precedence:

```
<advices-def name="advicestack">
    <advice-ref name="acl"/>
    <advice-ref name="logging"/>
    <advice-ref name="caching"/>
</advices-def>
```

Introduction definition

The introductions are defined by first specifying the name of the introduction, which will work as a handle to the introduction. Second the name of the interface class and the implementation class needs to be specified. Last an optional attribute specifying the *deployment model* to use. Examples of an introduction / mixin definition:

```
<introduction-def name="mixins/Mixin"
                  interface="mixins.Mixin"
                  implementation="mixins.MixinImpl"
                  deployment-model="perThread"/>
```

Pointcut definition

The pointcuts are defined by specifying the name, the type and the pattern. The name needs to be unique throughout the aspect definintion. The type attribute is the type of the pointcut, valid types are `method`, `setField`, `getField`, `throws`, `callerSide` and `cflow`. The pattern attribute specifies the pattern for the pointcut. This is the pattern that selects the join points that should be included in the pointcut.

Aspects definition

Aspect inheritance is defined using the `extends` attribute and an abstract aspect is defined using the `abstract-aspect` element. The advices and introductions are bound to the pointcuts using the `bind-advice` and the `bind-introduction` elements. Within these elements the advices and introductions are referenced using the `advices-ref` (advice stack reference) and the `introduction-ref` elements. Here is and example of an aspect definition using aspect abstraction and inheritance:

```
<abstract-aspect name="MyAbstractAspect">
    <bind-advice cflow="facadeCalls" pointcut="setters AND getters">
        <advices-ref name="logAndCache"/>
    </bind-advice>
</aspect>

<aspect name="MyAspect" extends="MyAbstractAspect">
    <bind-introduction class="domain.*">
        <introduction-ref name="serializable"/>
        <introduction-ref name="mixin"/>
    </bind-introduction>

    <pointcut-def name="facadeCalls"
                  type="cflow"
                  pattern="* *..facade.*.*(..)"/>
    <pointcut-def name="setters"
                  type="method"
                  pattern="* *..domain.*.set*(..)"/>
    <pointcut-def name="getters"
                  type="method"
                  pattern="* *..domain.*.get*(..)"/>
</aspect>
```

JavaDoc tag definition syntax

AspectWerkz also supports using JavaDoc [26] tags, also known as *doclet* tags, for defining both advices and introductions as well as marking the join points where they should be applied. This might seem counter-intuitive to the main goals of aspect-oriented programming, since it eliminates cross-cutting behavior by adding it again. But it has actually proved itself to be a good solution to certain problems. One of the advantages is that it supports *refactoring* [13] of the sources, something that is a big problem with the pattern-matching approach. Many users prefer it to the pattern syntax and some thinks that it is at least a good complement in certain situations. When using this approach the sources have to be processed with a special processor which parses the sources, retrieves the meta-data in the tags and produces and XML definition file.

Runtime attribute definition model

In the *runtime attribute definition model* the aspects are regular Java classes. The aspects can be abstract and inheritance is used and treated like regular Java class inheritance. This makes it easy to create reusable ‘self-defined’ aspect components and libraries. The advices and introductions are implemented as regular member methods in the aspect class. The methods representing the ‘mixin methods’, e.g. the methods that will be introduced in the target class can be any kind of member method. It does not have to have a certain name, be passed a certain parameter or return a certain object.

Currently the syntax of the runtime attribute tags is based on JavaDoc [26] tags. This is needed since the JSR-175 [7] which is defining runtime attributes for the Java language is not yet implemented (scheduled to the Java 1.5 release). Using JavaDoc tags to implement them is a work-around that works fine for now and is good to have for back-compatibility, e.g. to have it working for Java 1.4.x and below. These tags are compiled in into the class at bytecode level, see the *'Custom runtime attribute implementation'* section for details about the current implementation.

Advice definition

The model currently supports five different kinds of advices attributes:

- **`@Around`** - the advice is invoked "around" the join point.
- **`@Before`** - the advice is invoked before the join point.
- **`@After`** - the advice is invoked after the join point. This advice is always executed (regardless if the method returns normally or throws an exception).
- **`@After returning`** - the advice is invoked after a method invocation that returns normally.
- **`@After throws(<exception>)`** - the advice is invoked after a method invocation that throws an exception of a specific type

After the advice attribute the pointcut that this advice should be bound to has to be defined. An advice definition can for example look like this: `@Around methodsToCache.` Here an around advice is bound a pointcut called `methodToCache.`

Introduction definition

Mixins are defined using an inner class of the aspect. The methods of the inner class as well as the interfaces implemented by the class will be introduced on classes matching the pattern defined in the `@Introduce` attribute.

```
/**
 * @Aspect perInstance
 */
public class TestingAspect extends Aspect {

   /**
    * @Introduce foo.baz.*
    */
   class MockObject extends MyBase implements Traceable, Metered {
       ...
       public String sayHello() {
          return "Hello World!";
       }
       ...
   }
}
```

It is also possible to define single method introductions using the `@Introduce` attribute as well as plain interface introductions by using the `@Implements` attribute. Left to implement is the possibility of swapping or adding a super class using the `@Extend` attribute.

Pointcut definition

Five types of pointcut attributes are currently supported:

- **`@Execution`** **-** picks out join points defining method execution (callee side).
- **`@Call`** - picks out join points defining method call (caller side).
- **`@Set`** - picks out join points defining field modification.
- **`@Get`** - picks out join points defining field access.
- **`@CFlow`** - picks out join points defining a control flow.

The pattern for the pointcut is defined right after the pointcut attribute and the pointcuts are defined by declaring a member variable in the aspect class with the type `Pointcut`. The name of the member variable will be the name of the pointcut:

```
/**
 * @Execution * foo..Bar.*(..)
 */
Pointcut methodsToCache;
```

These pointcuts are compositional, meaning that they can form expressions based on the rules outlined in the *'Pointcuts'* section.

Aspect definition

The aspects are implemented using regular Java classes. The only rule that the aspect class needs to obey is that it has to extend the `Aspect` class. The aspects are defined using the `@Aspect` attribute. In the *runtime attribute definition model* the *deployment model* is defined on the aspect level (and not on advice and introduction level as in the *XML definition model*). The deployment type of the aspect is defined by adding the *deployment model* identifier right after the aspect attribute, e.g. `@Aspect perJVM`.

Definition example

Here is an example of a simple aspect with a couple of advices:

```
/**
 * @Aspect perInstance
 */
public class MyAspect extends Aspect {

    /* ===== Pointcuts ===== */
    /**
     * @Call * foo.bar.*(..)
     */
    Pointcut methodsToLog;

    /**
     * @Execution * foo.baz.*(..)
     */
    Pointcut methodsToCache;

    /* ===== Advices ===== */

    /**
     * @Around methodsToCache
     */
    public Object cacheMethod(JoinPoint joinPoint) throws Throwable {
        // check the cache for cached result, if found return it
        final Object result = joinPoint.proceed();
        // put result in cache
        return result;
    }

    /**
     * @Before methodsToLog
     */
    public void logMethod(JoinPoint joinPoint) throws Throwable {
        // log method entry
    }

    /**
     * @After returning methodsToLog
     */
    public void logMethod(JoinPoint joinPoint) throws Throwable {
        // log method entry after returning normally
    }

    /**
     * @After throws(java.lang.Exception) methodsToLog
     */
    public void logMethod(JoinPoint joinPoint) throws Throwable {
        // log method exit after throwing exception
    }
}
```

Custom runtime attribute implementation

As mentioned earlier the current implementation is based on a custom implementation of runtime attributes. This was needed since the Java language does currently not support runtime attributes. There is however a new JSR, JSR-175 [7], that will soon bring runtime attributes to the Java platform. *AspectWerkz* will support the standard implementation as soon as it is released, but will probably still support the current custom implementation for back-compatibility. The current implementation is based on

JavaDoc [26] tags. The tags are parsed by a special compiler that retrieves the meta-data and puts it into the class, method and field attributes in the compiled class. This meta-data information is then read at runtime and the aspects are configured based on this information. One of the drawbacks with this approach is that it adds an additional compilation step to the build process. This was avoided in the *XML definition model*. But fortunately this compilation step will not be needed anymore when runtime attributes has been added to the Java language (starting from Java 1.5).

RELATED WORK

AspectJ [4] is an aspect-oriented extension to the Java language. It has a very rich join point model and basically implements the whole theory of aspect-oriented programming. It currently does not have any dynamic features but provides a post-processor that weaves the classes on both source and bytecode level.

JAC [2] is a framework for aspect-oriented programming focused on separating concerns when developing distributed applications. It provides ready to go aspect components for example persistence, authentication and security. It has support for some dynamic features.

PROSE [1] (PROgrammable Service Extensions) is a framework that allows bytecode modifications at load time. It allows inserting aspects in running Java applications. The aspects are regular Java objects that can be sent to and received from computers on the network. The current implementation is based on the JPDA API [5].

Nanning [8] is a framework for aspect-oriented programming in Java based on the dynamic proxy feature added to the Java language in the 1.3 version. It has a join point model limited to method interception and supports introductions through the concept of mixins.

JMangler [9] is a framework for generic interception and transformation of Java programs at load time. JMangler hook mechanism is based on the JPDA API [5].

JBoss AOP [23] is a framework for aspect-oriented programming in Java. It is similar to *AspectWerkz* in the sense that is provides dynamic weaving on bytecode level and is defined through XML. It does not provide a JVM wide hook mechanism but is more targeted towards providing system aspects for the JBoss Application Server.

CONCLUSIONS

When implementing a framework for aspect-oriented programming one of the hardest and most important things to get right is the definition model, meaning the join point selection language as well as ways of using that language to express the needs of the design. This is crucial since the definition model is the control panel for the user. It needs to be simple and intuitive as well as expressive and powerful.

AspectWerkz provides three definition models. Initially it only supported definition through XML and doclet tags. But the shortcomings of these approaches led to the design and implementation of the third definition model, based on runtime attributes.

AspectWerkz intercepts the whole class loading hierarchy as a way of ensuring a JVM wide hook mechanism. One thing that was to be hard to get right in the beginning was to intercept the complex class loading hierarchies of application servers correctly. This lead to some refactorings [13] and redesigns until a stable implementation could be reached, compliant with the WebLogic [21], WebSphere [22] and JBoss [23] application servers.

Even though the *AspectWerkz* framework has a very dynamic aspect model it is still lacking one important feature; being able to define new pointcuts at runtime. The current limitations of the Java language [16] makes overly complex to implement runtime weaving (which is needed to allow redefinition of pointcuts at runtime without reloading the classes and still have reasonable performance). What the Java language is lacking is a standardized way of doing class redefinitions at runtime and at the same time allow the JVM do code optimizations. This is something that will be brought to Java through the JVMTI API [6] (which will be released with Java 1.5).

Since *AspectWerkz* has been developed as an open source product with a growing user community already from the very beginning, we have had the opportunity of receiving continuous feedback on new achievements and features which has helped us forming a framework that lives up to the requirements put on a real-world product running in production.

ACKNOWLEDGEMENTS

First I would like to thank all the users of *AspectWerkz* who have given me invaluable feedback.

Special thanks to:
Alexandre Vasseur, core committer to the *AspectWerkz* project, for many very important contributions. Among other things the design and implementation of the new runtime weaving architecture. As well as for feedback on how to improve this paper.

Ron Bodkin, for the initial ideas about how to use *runtime attributes* and the upcoming JSR-175 to define ‘self-defined’ aspect components.

Gregor Kiczales, for the discussions about the shortcomings of the early XML definition model, feedback that helped me to come up with a more thought through and orthogonal model. As well as feedback on how to improve this paper.

Sara Bonér, my wife. For all the encouragement and support while designing and implementing the *AspectWerkz* framework.

REFERENCES

[1] Pawlak, R., Seinturier, L., Duchien, L. and Florin, G.
JAC: A flexible solution for aspect-oriented programming in Java.
In Proceedings of the 3rd International Conference on Reflection.
Kyoto Japan, September 2001

[2] Popovici, A., Gross, T. and Alonso, G.
Dynamic Weaving for Aspect-Oriented Programming.
In Proceedings of the 1st international conference on Aspect-oriented software development.
Enschede, Netherlands, April 2002

[3] Gamma, E., Helm, R., Johnson, R. and Vlissides, J.
Design Patterns: Elements of Reusable Object-Oriented Software.
Addison-Wesley, 1994

[4] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold.
Getting Started with AspectJ.
Communications of the ACM, 44(10):59–65, October 2001.

[5] Sun Microsystems. JPDA / HotSwap API.
At http://java.sun.com/products/jpda/, 2003.

[6] JSR-163. At http://www.jcp.org/en/jsr/detail?id=163, 2003

[7] JSR-175. At http://www.jcp.org/en/jsr/detail?id=175, 2003

[8] Nanning Aspects. At http://nanning.codehaus.org/, 2003

[9] Kniesel, G., Costanza, P. and Austermann, M.
JMangler - A Framework for Load-Time Transformation of Java Class Files.
IEEE Workshop on Source Code Analysis and Manipulation (SCAM), collocated with International Conference on Software Maintenance (ICSM), November 2001.

[10] Sun Microsystems. JVMPI API.
At http://java.sun.com/products/jdk/1.2/docs/guide/jvmpi/jvmpi.html , 2003.

[11] Sun Microsystems. JVMDI API.
At http://java.sun.com/products/jdk/1.2/docs/guide/jvmdi/jvmdi.html , 2003.

[12] Newkirk, J., Vorontsov, A.
How .NET's Custom Attributes Affect Design.
At http://www.martinfowler.com/ieeeSoftware/netAttributes.pdf, 2003.

[13] Fowler, M. Beck, K., Brant, J., Opdyke, W., Roberts, D.
Refactoring: Improving the design of existing code.
Addison-Wesley, 1999

[14] Bracha, G., Cook, W.
Mixin-based Inheritance.
In Proceedings of the Joint ACM Conf. on Object-Oriented Programming, Systems, Languages and Applications and the European Conference on Object-Oriented Programming, October 1990

[15] Dahm, M.
Byte Code Engineering
Proceedings JIT '99, Springer, 1990

[16] Gosling, J., Joy, B., Steele, G.
The Java Language Specification (2nd edition)
Addison-Wesley, 2000

[17] W3C, Extensible Markup Language (XML) 1.0 (Second Edition)
At http://www.w3.org/TR/REC-xml, 2003

[18] CGLib – Code Generation Library. At http://cglib.sourceforge.net/, 2003

[19] HotSpot JVM, Sun Microsystems. At http://java.sun.com/j2se/, 2003

[20] JRockit JVM, BEA Systems.
At http://dev2dev.bea.com/products/wljrockit81/index.jsp, 2003

[21] WebLogic Server, BEA Systems.
At http://dev2dev.bea.com/products/wlserver81/index.jsp, 2003

[22] WebSphere Application Server, IBM.
At http://www-3.ibm.com/software/info1/websphere/index.jsp, 2003

[23] JBoss Application Server. At http://www.jboss.org/index.html, 2003

[24] J2EE, Java 2 Enterprise Edition, At http://java.sun.com/j2ee/, 2003

[25] AOSD, Aspect-Oriented Software Development, At http://www.aosd.net/, 2003

[26] JavaDoc Tool, At http://java.sun.com/j2se/javadoc/, 2003