Dynamic AOP and Runtime Weaving for Java — How does AspectWerkz Address It?

Alexandre Vasseur
avasseur@bea.com

Abstract

This paper describes how AspectWerkz [1]'s dynamic AOP capabilities for Java [2] are extended to support runtime weaving. While static weaving (post compilation or class load time) allows for some dynamic AOP features with the help of a join point centric weaving model and an aspect container component, runtime weaving is richer in the sense that weaver targets can be declared at runtime. A working implementation allows to define key principles for enabling of dynamic AOP with pointcut redefinition: "in process HotSwap" and "two phases weaving."

Keywords

Aspect-Oriented Programming, AOP, AOSD, dynamic AOP, runtime weaving, HotSwap

1 Introduction

One central component in AOP is the weaver. Given a set of target programs and a set of defined aspects, the weaver alters the target program to weave in the aspects capabilities, embodied by advices, mixins and binding rules.

In the Java [2] landscape the most common approach is to alter the compiled class bytecode to produce a bytecode where aspects are weaved in. If bytecode manipulation is easy to do through a post compilation phase, it is a bit more complex to provide a solution generic enough to apply bytecode modification at class load time, but some projects like JMangler [3] and AspectWerkz have achieved it, and Java 1.5 will standardize this approach with the JSR-163 [4].

Allowing dynamic AOP on top of such a static weaving phase relies on internal AOP constructs. AspectWerkz allows many dynamic AOP operations — like adding an advice or changing an introduction implementation — with the only requirement that the pointcut is already defined. It is then possible to rearrange advices and swap mixin implementations at runtime, without any class reloading or new weaving phase. The first part of this paper aims at explaining some of the major implementation details needed to enable dynamic AOP constructs.

Even if this approach is suitable for many use-cases, one missing feature is to allow pointcut definition at runtime, so that a target component can have its normal behavior (eventually with aspects) up to a point where it is altered with new AOP constructs on totally new join points. This allows for new constructs to be applied without any prior overhead. Instead of preparing each method call, execution or field access with a join point, pointcuts are defined at runtime and corresponding targets are altered at runtime to weave the new aspect constructs only when it is required.

The second part describes the proposed solution and the key components that allow runtime definition and redefinition of pointcuts without class reloading, and with a prior runtime performance overhead reduced to zero.

2 Static weaving and dynamic AOP constructs

Dynamic AOP constructs can be built on top of a classic static weaver, where target classes are altered once, whether in a post compilation phase prior to deployment, or at class load time thus at deployment time.

A join point centric weaving model can provide enough redirection level in the weaved bytecode to be able to alter AOP constructs at the underlying framework level.

The AspectWerkz dynamic AOP framework follows such a design since its early conception.

2.1 Static weaving

The common approach for applying AOP principles in the Java landscape is bytecode manipulation. Several APIs like BCEL [5], Javassist [6] and ASM [7] provide access to the bytecode instructions and several Java AOP frameworks like AspectJ [8] and AspectWerkz [1] make use of such facilities. The bytecode manipulation is a way to support a fine-grained join point model in different context such as field access and method calls and executions.

Static weaving can be defined as the single phase operation that given a class representation and a set of aspects alters the representation to produce a new class where aspects are weaved in at precise join points.

The first weavers for Java AOP only supported static weaving through post compilation tools, and some recent steps [3][1] ahead demonstrate that the same operation can be achieved at class load time no matter the target environment. AspectWerkz's hooking architecture supports integration in J2EE application servers with complex class loading schemes, non-standard JREs like IBM and non-standard JVMs like BEA JRockit. AspectWerkz extends a concept that has been first defined by the JMangler [3] project. The class load time weaving will be standardized in Java 1.5 through the JSR-163[1] [4]. This feature can be used for AOP but is there to support better profiling as an enhancement of JPDA [9].

Once statically weaved (at compilation time or at class load time), the bytecode is loaded in the JVM and the target weaved *java.lang.Class* object is defined. The application then runs, without any further modifications at bytecode level.

Dynamic AOP constructs are then enabled through a well-designed weaving model and aspect container [10].

2.2 Weaving model and aspect container

The weaving model defines the way the target class bytecode is altered when a pointcut is matched.

The aspect container [10] is in charge of managing deployed aspects and mixin implementations, and supports fine deployment options (perThread [11] aspect shared between target instances on a per execution thread basis) as well as advice rearrangement.

Both are key elements to allow dynamic AOP constructs. If the weaving model does not have enough indirection level, it will not be possible for example to add new advices at the join point.

The AspectWerkz's weaving model provides a good level of indirection by being *join point centric*. The altered bytecode contains information about the join point only, and the aspect container then keeps information about which advice(s) is (are) associated with the join point, or which mixin implementation(s) is (are) associated with the weaved in mixin interface(s).

[1] Java 1.3 and 1.4 supported "bytecode loaded" event callbacks at C JVMPI level, but without providing the defining ClassLoader instance, which appears to be a requirements in AspectWerkz AOP. Java 1.5 fixes this and also provides a Java level API, such the one already provided in BEA JRockit since v7 (Java 1.3).

At runtime, a join point manager allows to invoke the current advice(s) bounded at the join point. Dynamic AOP constructs around the join point are thus possible at aspect container level without any new bytecode modifications.

Some examples of the weaving model allows for a better understanding. The examples are provided as source code for the sake of simplicity as compared to bytecode excerpt.

2.2.1 Method execution pointcut

Given the initial bytecode representing:

```
public class Target {

    public Object doSomething(int param) {
        // ...
        // doSomething original method body
        return result;
    }
}
```

And given a method execution pointcut matching *Target.doSomething(int)*, the static weaving in AspectWerkz leads to the following representation:

```
public class Target {

    private static __clazz = Class.forName("Target");

    private static JoinPointManager __jpManager =
JoinPointManager.get(__clazz);

    private Object __doSomething(int param) {
        // ...
        // doSomething original method body
        return result;
    }

    public Object doSomething(int param) {
        __jpManager.proceedWithExecutionJoinPoint(
                -0x25ca1875,//method hash
                //... join point information
                );
    }
 }
```

The original method is renamed, and an indirection level is added as a replacement through an added method with the original method name and the same signature. No explicit call to the original method appears since the join point manager will handle it.

No information about the bounded advices is weaved in into the target class. Only a join point manager instance is created as a static field, providing enough information when a join point is reached to determine to which class, instance and original method execution it relates. A method hash (representing an integer) is defined, so that direct access through an array lookup can be performed at runtime. The method hash is computed given the original method signature.

The *proceedWithExecutionJoinPoint(..)* call will invoke all bounded advice(s) (if any), providing support for before, around and after advices.

For a static method, the weaving model is exactly the same.

2.2.2 Caller side pointcut

Given the initial bytecode representing:

```
public class Caller {

    public Object callTarget(Target target) {
        // ... body
        Object result = target.doSomething(...);
        return result;
    }
}
```

And given a caller side method pointcut matching "*Caller calls Target.doSomething(..),*" the static weaving in AspectWerkz leads to the following representation:

```
public class Caller {

    private static __clazz = Class.forName("Target");

    private static JoinPointManager __jpManager =
                                     JoinPointManager.get(__clazz);

    public Object callTarget(Target target) {
        // ... body
        // altered body below
        Object result = __jpManager.proceedWithCallJoinPoint(
                0xe30193c,//called method hash
                //... join point information
                );
        return result;
    }
}
```

The weaving model for the caller side pointcut construct does not express the advices bounded. This is done at the aspect container level and invoked at runtime through the join point manager. The before, around and after advices will be called subsequently.

The same approach is used for field set and get pointcuts.

2.2.3 Mixin

Mixins provide ability to add new methods to a class. A first simplistic approach would be to weave in the mixin implementation bytecode in the target class, but this would not be suitable for dynamic AOP constructs.

Again, AspectWerkz's weaver adds an indirection level. In this case there is no join point or join point manager involved, but a mixin manager, which is a subpart of the aspect container.

Given the following mixin implementation that implements the user provided interface *MixinIntf*:

```
public interface MixinIntf {

    public void sayHello();

}

public class MixinImpl implements MixinIntf {
    public void sayHello() {
        // ...
        // sayHello implementation
    }

}
```

And given that the mixin is defined to be bounded on “all class(es) whose name is Target,” the weaved bytecode will be represented as follows:

```
public class Target implements MixinIntf {

    private static __clazz = Class.forName("Target");

    private static AspectManager __aspectManager =
AspectManager.get(__clazz);

    public void sayHello() {
        __aspectManager.getMixin(1).//mixin index
                ___AW_invokeMixin(
                        1,//mixin method index
                        //...mixin invocation parameters
                        );

    }

    // ...
    //
    // might depends on other AOP construct
    // applied to Target class

}
```

2.3 Dynamic AOP constructs

The indirection level brought by the join point centric static weaving model, and a mixin / aspect container provide the low level mechanisms to apply several dynamic AOP constructs in AspectWerkz. Since the join points have been defined during the static weaving phase, the dynamic AOP constructs do not rely on a new weaving phase.

At join point first execution, the aspect container gathers information about the join point and is then able to determine which advices are bounded. At the first mixin invocation, the mixin is instantiated. Deployment models [11] like *perThread* are implemented at the container level using the Prototype pattern [12].

At runtime, it is then possible to perform dynamic AOP operation on the system through the aspect container. An API allows to lookup aspects or pointcuts (based on their name or on specific method and class matching) to:

- add an advice at a specific pointcut,
- remove an advice at a specific pointcut,
- reorder advices at a specific pointcut,
- swap the implementation of a mixin,

- add an aspect providing the new aspect does not define new pointcuts, and defines advices (represented as regular java methods in AspectWerkz) that will be programmatically bounded to existing pointcut(s).

The following demonstrates how to use the API to access an existing pointcut and add a new advice based on the aspect, the pointcut and the advice names:

```
ExecutionPointcut executionPointcut = SystemLoader.
        getSystem("namespace").
        getAspectManager().
        getPointcutManager("aspectName").
        getExecutionPointcuts(classMeta, methodMeta).
        get("pointcutName");

executionPointcut.addAroundAdvice("Aspect.newAdvice");
```

The following demonstrates hot to use the API to replace a mixin implementation, based on the introduction and the new implementation names:

```
SystemLoader.getSystem("namespace").
        getAspectManager().
        getMixin("introductionName").
        ___AW_swapImplementation("MixinOtherImpl");
```

It is possible to call those operations from within an advice itself, which allows for adaptive behavior.

2.4 Limitations

AspectWerkz's static weaving model and aspect container are key elements to enable dynamic AOP constructs.

A shortcoming in this implementation is that the join point has to exist as a result of the static weaving phase, no matter if it is through post compilation or through class load time weaving. It means that runtime definition of new pointcuts is not possible.

One approach would be to have very generic pointcuts, and then adapt the advice's behavior according to runtime type information available in the join point. If such rules can be applied elegantly with *cflow* constructs like it is supported in AspectJ [8] and AspectWerkz [1], it might lead to a global overhead, both at weave time and at runtime.

A key requirement in dynamic AOP appears to be the capability to add new pointcuts in the running system, with a good control over the performance overhead before the pointcuts are applied, so that almost all classes in the JVM can be altered at runtime without prior preparation and without any overhead.

Wool [13] tries to address this issue using a hybrid approach between bytecode weaving at runtime and debugging breakpoints usage. Axon [14] addresses it by using a JVMDI [15] event callback registration to be notified at each method execution and do the necessary advice calls — if any.

The following part explains the implementation done in AspectWerkz to allow runtime pointcut definition while still being compliant with the classic static weaving approach.

3 Runtime weaving

Runtime weaving has actually been supported in the Java landscape since Java 1.4 and the HotSwap facilities of JVMDI [15] that allows redefinition of the class bytecode at runtime, without reloading the class.

Even if hot deployment is a common pattern to update a system, we don't think it solves the problem. Re-deploying an application in an application server is not a seamless operation and affects the application uptime. Runtime weaving allows much more fine-grained level control.

HotSwapping can indeed be used to reweave target classes during runtime, upon user request or system request. Use of this operation can allow adding pointcuts at runtime and thus address a missing core capability in AspectWerkz's static weaving model for dynamic AOP.

A first implementation has been done to validate the viability of such an approach, and to point out main issues and components, as well as to ensure that performance overhead prior pointcut runtime definition is minimal.

The proposed solution relies on a hybrid approach between static weaving and runtime weaving as well as on an in-process HotSwap custom API. This results in a null runtime overhead prior pointcut addition, at the cost of a small load time overhead.

3.1 HotSwap in Java

HotSwap, also called class redefinition, is a feature added in Java 1.4. It is part of the JVMDI [15] API available at C level and remotely at Java level through JDWP [16].

When JVMDI is activated through the *–Xdebug* flag when launching a Java 1 .4 HotSwap compliant JVM, the method call dependencies are recorded internally.

The HotSwap API allows to submit new bytecode for an already loaded class in the running JVM. Former methods that appeared to have changed are relinked in the JVM internal representation based on the method call dependencies recorded so far. When a new invocation is done, it goes through the new method, part of the HotSwapped class.

Even though the HotSwap API looks very attractive, we think there are potential problems. The current Java directions seem to aim at having HotSwap more usable:

- The HotSwap API has not been adopted by all JVMs. BEA JRockit does not support it yet. Since the JSR-163 [4] part of Java 1.5 redefines the API, we can expect changes as regards HotSwap support.
- The HotSwap API in Java 1.4 was part of JVMDI thus required the use of the *–Xdebug* flag. Such a requirement can reduce the JVM performance, especially when running in server mode. In Java 1.5, it seems it won't be required anymore, but the effective overhead will still have to be measured.
- The HotSwap API can only be called at Java level through a JDWP connector, thus from a second JVM (as done in Wool [13]), or only at C level from within the target JVM. Java 1.5 will bring the API at Java level within the target JVM, and will add the "in-process HotSwap" feature as a default.

One key issue that remains when using HotSwap is that the class representation represented by the new bytecode has to conform to some restricting rules:

- The class schema must not change: there is no way to add new methods, even private, or to change signatures of the methods. No field can be added.
- The class initialization (<clinit>) is not rerunned.

In theory the API allows to support schema changes, but up to now we are not aware of such JVMs.

Obviously, the previously described weaving model of AspectWerkz has to be adapted to be HotSwap compliant due to the current schema change restriction. The next parts express the required changes.

3.2 In-process HotSwap

In the AspectWerkz's runtime weaving implementation we decided to use the HotSwap API directly from within the target JVM, without using a JDWP connection. We think it is a way to validate what will be standardized in Java 1.5.

Since in Java 1.4 the API is only available at C level we have set up a Java API on top of it, using the JNI [17] features. It allows to have in Java 1.4 an API that is almost the same that the one defined by JSR-163, and that simplifies the use of the HotSwap feature by bringing "in process HotSwap" Java level API.

```
public class InProcessHotSwap {
     static {
        System.loadLibrary("aspectwerkz");
    }

    private static native int hotswap(
            String className,
            Class orginalClass,
            byte[] newBytes,
            int newLength
            );

   // … utility methods

}
```

The target JVM has to activate the HotSwap using the *–Xdebug* flag and having the system dependant built JNI based in process HotSwap library in its path.

3.3 Two phases weaving

One key issue to solve when using HotSwap is that we are for now restricted to a set of bytecode changes that do not lead to a "class schema change," due to current JVM limitations.

As detailed in the previous part this is almost always the default weaving model in AspectWerkz, excepted for execution pointcut that leads to addition of methods.

To address these changes while still being compatible with a class load time weaving model, we decided to use a hybrid approach where target classes are weaved at class load time as required, prepared for further HotSwap if needed, and then activated through HotSwap when new pointcuts are defined.

This hybrid approach thus makes use of both class load time weaving and runtime weaving.

3.3.1 Class load time preparation

To address the current JVMs schema change limitation, a class preparation phase is required. We decided to allow fine grained control on this preparation phase, thought it can be enable for all loaded classes if required. The classes to prepare are thus explicitly declared in the AspectWerkz's XML based descriptor.

```
<?xml version="1.0"?>
<aspectwerkz>

    <system id="demo">
        <prepare class="Target" />
        <prepare package="com.service.*" />

        <!-- ... -->
        <!-- other AspectWerkz related elements -->
    </system>

</aspectwerkz>
```

When a loaded class matches those *"<prepare …>"* declarations, the following transformations are applied:

- add a private static __*clazz* field to reference the current Class object
- add a private static __*jpManager* field to reference the current Class join point manager
- for all methods, add a new empty method whose name is the AspectWerkz's prefixed method name, if and only if the method is not already matched by a declared pointcut

The added methods are empty, simply returning the default value required for JVM bytecode compatibility.

The preparation phase is fully compatible with the static weaving phase. A prepared class will be eligible for runtime weaving while still being affected during the static weaving phase by declared AOP constructs (if any).

The following illustrates the change if the class *Target* presented so far is prepared:

```
public class Target {

    private static __clazz = Class.forName("Target");

    private static JoinPointManager __jpManager =
JoinPointManager.get(__clazz);

    private Object __doSomething(int param) {
        // empty method for further HotSwap
        return null;
    }

    public Object doSomething(int param) {
        // unchanged.
        // doSomething original method body
        return result;
    }

    // other class methods and constructors
    // advised if required during the static weaving phase

}
```

3.3.2 Activation phase

The activation phase allows the triggering of a new bytecode transformation based on the bytecode obtained after the load time preparation.

The class bytecode is weaved so that when an unaffected method match a newly defined execution pointcut, the original method body is moved into the empty method added in the preparation phase while the original method body receives the join point centric weaving model code.

The class schema is not changed and this transformation can thus be submitted to the in process HotSwap API with the current JVM limitations. The resulting bytecode is the same as the one obtained in a static weaving, where the added join points would have been initially defined.

After the activation phase the runtime overhead is the one that occurs when a method is advised, but prior to this the runtime overhead is absolutely null since the exact original bytecode gets executed with the same stack trace.

3.4 Caller side constructs

Caller side constructs — like field get set and method call pointcut — are easier to be adapted for two phase weaving and HotSwap requirements since they do not require a schema change.

During the activation phase, the affected method body is changed to insert the call to the join point manager. This change is compatible with HotSwap schema changes restrictions providing that the join point manager reference has been added during the preparation phase.

4 Future work

This AspectWerkz's implementation for runtime weaving allowed us to define and validate a way to allow join point runtime redefinition without any prior runtime overhead for non advised method calls and executions and non advised field accesses. The two phase weaving approach, based on an in process HotSwap Java level API allows to address this major dynamic AOP requirement, whereas a join point centric weaving model allows to address advice and aspect rearrangement on existing join points.

One key element in this hybrid model is the prepare phase overhead. This one has to be as low as possible so that many classes — if not all — can be declared as to be prepared. The preparation phase is indeed very lightweight, and consists only in adding empty methods that won't be called prior activation and two fields to a class bytecode.

The hybrid approach allowed us to address a limitation introduced by the HotSwap API schema changes restriction. This could be simplified if JVMs would support unrestricted schema changes during HotSwap.

A more important issue that will have to be addressed in order to enhance the current implementation is that once a class is loaded in the JVM, there is currently no API to retrieve its current bytecode representation. The Java technology provides only a way to retrieve the original bytecode, as it is stored in the class path (eventually remotely), or to retrieve a single method bytecode representation (using JVMDI [15]). We think this limitation might be a major issue for large HotSwap based systems, as well as systems making use of several bytecode transformation sub-systems like AOP, or profilers. The current implementation makes use of a local cache, which is enough to have it work for small applications. This issue may come from the historical use-case that required HotSwap: runtime debugging facilities based on source code changes. We think this issue can be addressed at the JVM level, by providing an API that would allow to extract the exact bytecode representation of a class as it is actually at runtime, and not as it is when loaded.

JVM level capabilities is becoming a crucial success factor for complete dynamic AOP solutions, and the border between what should be handled by the JVM and what should be handled by the AOP framework has yet to be defined. The work described in this paper aimed at having a working solution today to allow pointcut redefinition with Java 1.4. Java 1.5 will even more ease this approach since in process HotSwap (with a java level API) and class load time weaving hook are standardized. As a future work, we will look at how aspect hot deployment can trigger runtime weaving.

No matter how dynamic AOP will be technically achieved in Java, we will have to address complex use-case in this area: how to allow for good system state control and management, how can I proceed to reproduce a problem in QA environments when several external events might have modified the running system?

Acknowledgments

Many thanks to Jonas Bonér, founder of AspectWerkz, for his invaluable feedback on how to improve the hybrid approach as well as on this paper.

References

[1] Bonér, J. AspectWerkz: dynamic AOP for Java. 2003. http://codehaus.org/~jboner/papers/aosd2004_aspectwerkz.pdf

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

[3] Kniesel G., Costanza P, Austermann M. JMangler — A Framework for Load-Time Transformation of Java Class Files. IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2001).

[4] JSR-163. At http://www.jcp.org/en/jsr/detail?id=163, 2004.

[5] BCEL, The Byte Code Engineering Library. http://jakarta.apache.org/bcel/

[6] Javassist, Java Programming Assistant, http://www.csg.is.titech.ac.jp/~chiba/javassist/

[7] ASM, http://asm.objectweb.org/

[8] 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.

[9] JPDA, Java Platform Debugger Architecture, http://java.sun.com/products/jpda/

[10] Bonér J., What are the key issues for commercial AOP use — how does AspectWerkz address them ?. Submitted for AOSD 2004.

[11] AspectWerkz homepage, http://aspectwerkz.codehaus.org, 2004.

[12] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns — Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[13] Shigeru Chiba, Yoshiki Sato, Michiaki Tatsubori. Using HotSwap for Implementing Dynamic AOP Systems. ECOOP'03 workshop on Advancing the State of the Art in Runtime Inspection (ASARTI), 2003.

[14] Aussmann S., Haupt M. Axon — Dynamic AOP through Runtime Inspection and Monitoring. ECOOP'03 workshop on Advancing the State of the Art in Runtime Inspection (ASARTI), 2003.

[15] JVMDI, The Java VM Debug Interface. http://java.sun.com/j2se/1.4.2/docs/guide/jpda/jvmdi-spec.html

[16] JDWP, Java Debug Wire Protocol. http://java.sun.com/j2se/1.4.2/docs/guide/jpda/jdwp-spec.html

[17] JNI, Java Native Interface. http://java.sun.com/j2se/1.4.2/docs/guide/jni/