Aspects and Class-based Security

A Survey of Interactions between Advice Weaving and the Java 2 Security Model

Andreas Sewe Christoph Bockisch Mira Mezini

Technische Universität Darmstadt
Hochschulstr. 10, 64289 Darmstadt, Germany
{sewe, bockisch, mezini}@st.informatik.tu-darmstadt.de

Abstract

Various aspect-oriented languages, e.g., AspectJ, AspectWerkz, and JAsCo, have been proposed as extensions to one particular object-oriented base language, namely Java. But these extensions do not fully take the interactions with the Java 2 security model into account. In particular, the implementation technique of advice weaving gives rise to two security issues: the erroneous assignment of aspects to protection domains and the violation of namespace separation. Therefore, a comprehensive discussion of the design choices available with respect to interactions with the dynamic class loading facilities of the Java VM is provided.

Categories and Subject Descriptors D.3.2 [*Programming Languages*]: Language Classifications—Multiparadigm languages, Object-oriented languages; D.4.6 [*Operating Systems*]: Security and Protection—Access controls

General Terms Languages, Security

Keywords Advice weaving, aspect-oriented programming, dynamic class loading, Java security model

1. Introduction

The paradigm of aspect-oriented programming (AOP) aims at the modularization of cross-cutting concerns [13], i.e., concerns which cut across modularizations as offered by other paradigms, e.g., across class hierarchies in the case of object-oriented programming (OOP). To this end, an *aspect language* typically extends a *base language* rooted in another paradigm with a new kind of module: the aspect. For various aspect-oriented languages, e.g., for AspectJ [12, 3], AspectWerkz [7], and JAsCo [18, 19], this base language was chosen to be Java [10], whose entire security model revolves around the sole kind of module known to it: the class. Thus, the question arises how to best integrate aspects with Java's class-centric model in the light of the Java VM's dynamic class loading facility. In this paper we argue for increasing the level of class loader awareness of execution environments geared towards AOP and show how failure to do so can have serious security implications. To bolster the argument we have surveyed the behavior of current implementations, characterized interactions between classes and aspects, and identified a desirable design.

The remainder of this paper is structured as follows. Section 2 provides background material on two subjects: The security model of Java is described in Section 2.1, whereas Section 2.2 describes aspect-oriented programming and the weaving technique used by its implementations. Section 3 lists the surveyed implementations, before Section 4 characterizes not only the class loading behavior their execution environments exhibit, but also the design we deem most desirable. Section 5 mentions further implementation issues, Section 6 cites related work, and Section 7 concludes our position and gives suggestions for future work.

VMIL'08, October 19, 2008, Nashville, TN.
Copyright © 2008 ACM 978-1-60558-384-6...$5.00

2. Background

The interactions between aspects and the class-based security model of Java are often subtle. It is therefore crucial to understand both the design of Java's security model and common implementation techniques of aspect-oriented languages.

2.1 The Java 2 Security Model

The Java VM's facilities for dynamic class loading [14] are an intrinsic part of the Java 2 security model, colloquially called the "Java sandbox." Said model revolves around three core components: the `SecurityManager`, the `AccessController`, and the `ClassLoader`.

2.1.1 Access Control

All operations deemed critical in the core API of Java are subject to access control; permissions, e.g., to delete a file, are ultimately granted by a `SecurityManager`, as is shown in the following.

```
class File {
  // ...
  boolean delete() {
    SecurityManager m =
      System.getSecurityManager();
    Permission p =
      new FilePermission(this.getPath(),
        FILE_DELETE_ACTION);
    if (m != null)
      m.checkPermssion(p); // May throw SecurityException
    // Perform delete
  }
}
```

With the advent of Java 2, this mechanism's flexibility has been greatly enhanced by the `AccessController` framework for user-defined policies [9]. Hereby the VM's `SecurityManager` bases its decision of whether to grant a permission both on a user-configurable policy and on the calling context of the request. To determine the latter, the `AccessController` inspects all so-called protection domains recorded on the call stack and grants only those permissions afforded by each of them.[1] Which domain a frame belongs to thereby depends on the declaring class of the frame's corresponding method; which domain the class in question belongs to is fixed during class loading, e.g., depending on the source the class's code has been obtained from.

2.1.2 Dynamic Class Loading

Class loaders have been introduced to the Java security model in order to facilitate multiple namespaces, a user-definable class loading policy, and type-safe linkage in the presence of lazy loading [14]. They are responsible for resolving symbolic references [15, §5.1], i.e., fully qualified class names, to `Class` instances. Furthermore, it is among the class loaders' responsibilities to assign any newly defined class to a protection domain. Since class loaders thus form one of the corner stones of Java's security model, both the creation of and access to class loaders are controlled by a `SecurityManager`.

The ability to create `ClassLoader` instances also entails the ability to define new namespaces which can cleanly separate trusted from untrusted classes. This is possible since a class's identity is not uniquely determined by its name alone; at least at run-time class loaders need also be taken into account. This is exemplified by Figure 1, which depicts a typical class loader hierarchy in which the class loader instances `AppletClassLoader@5` and `AppletClassLoader@6` are used to separate the Java applets defined by them not only from one another but also from other parts of the application; classes defined by the two instances are, e.g., unable to refer to classes defined by either `URLClassLoader` as they reside within different branches of the class loader hierarchy.

[1] Privileged actions, which exempt parts of the call stack from inspection, are beyond the scope of this paper (cf. Section 7).

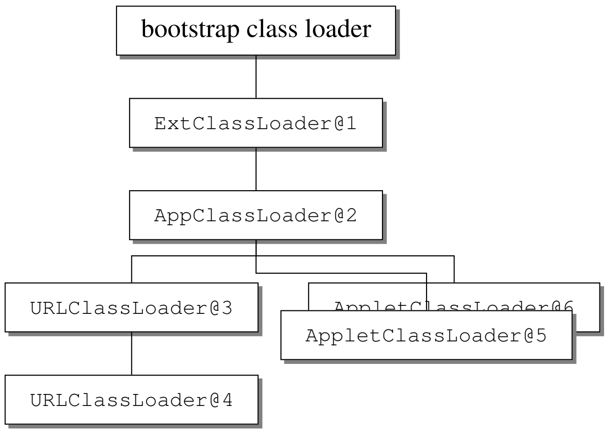

Figure 1: A class loader hierarchy, rooted at the bootstrap class loader [15, §5.3.1] and defining two separate namespaces for Java applets.

This behavior in which classes can refer only to classes defined by their own class loader or one of its ancestors in the hierarchy is only a convention—although one almost universally obeyed. What ultimately determines the `Class` returned upon a call to a class loader's `loadClass`[2] method is up to its (user-definable) implementation. Typically, this initiating class loader first delegates to its parent class loader before attempting to define the class on its own by means of its `defineClass` method. There are exceptions, though: Servlet containers follow the so-called delegation inversion model, which behaves the other way around—but can be emulated by the standard delegation model. But regardless of how delegation is handled, the Java VM places constraints upon loading which ensure type-safe linkage [15, §5.3.4]; thus, any delegation model is merely a set of guidelines on how to fulfill these constraints.

Even though these guidelines may appeal to our intuition of a class loader hierarchy, we cannot rely on any such behavior; thus, we have to adopt the following definitions [15, §5.3] and notation [14] to precisely describe class loading.

Definition 2.1. Let $c = l$`.loadClass(`"C"`)`. Then l is said to be the initiating class loader of the class c.

Definition 2.2. Let $c = l_c$`.defineClass(`"C"`)`. Then l_c is said to be the defining class loader of the class c, i.e., l_c defines c.

Note that due to delegation there may be multiple initiating class loaders of a class; however, for any class c there is always a single defining class loader l_c.

Notation 2.1. A class named "C" is denoted by $\langle\text{"C"}, l_c\rangle^l$, if l_c and l are its defining and initiating class loaders, re-

[2] All attempts to load a class dynamically are channeled through the `loadClass` method; the well-known `Class.forName`, e.g., simply defers class loading to this method.

spectively. If clear from context, this will be abbreviated to $\langle$"C", $l_c\rangle$ or $\langle$"C"$\rangle^l$.

2.2 Aspect-oriented Programming

In AOP's pointcut-and-advice flavor [16], which will be the focus of this paper, aspects affect the execution of so-called join points, e.g., calls to methods or accesses of fields. Hereby *pointcuts* select a set of join points at which—in addition to the action of the join point itself, i.e., the method call or field access—additional actions are to be performed in the form of so-called *advice*.

The following fragment exemplifies this; it defines a pointcut together with its associated advice which intercepts all calls to `delete` in order to subject this file I/O operation to access control as described in Section 2.1.1. Hereby the **`call`** atomic pointcut determines the *join point shadow*, i.e., the actual call instructions at which the advice will take effect, whereas the **`target`** atomic pointcut binds a context value, namely the callee. If permission is granted, the advice proceeds to the intercepted method, i.e., to `delete()`.

```
aspect AccessControl {
 around(File f) : call(File.delete())
    && target(f) {
   SecurityManager m =
     System.getSecurityManager();
   Permission p =
     new FilePermission(f.getPath(),
       FILE_DELETE_ACTION);
   if (m != null)
     m.checkPermssion(p); // May throw SecurityException
   proceed(f); // Proceed with f.delete()
 }
}
```

The above illustrates the usefulness of the aspect-oriented paradigm to modularize cross-cutting concerns like access control, which would otherwise be tangled with code implementing totally unrelated concerns, e.g., performing file I/O.

In order to realize these semantics on top of the Java platform, implementations of aspect-oriented languages (cf. Section 3) typically "weave" a call to a synthetic advice method into the code of the base program [11], thereby altering classes other than the aspect; this is illustrated by the listing below, which completely replaces all calls to `delete()`.[3]

```
AccessControl a = AccessControl.aspectOf();
a.around$1(f, new AroundClosure$2());
```

Code like the above is generated at different times by different implementations, e.g., at compile-time, post-compile-time, load-time, or run-time. For the purpose of this discussion, the former two options have the same implications and will therefore be subsumed under the term static weaving. While they are altogether ignorant of dynamic class loading, it is impossible to weave into classes from a code source different from the aspect's. Since consequently only the security considerations for ordinary Java apply in this case, static weavers have been excluded from the survey.

Like compile-time and post-compile-time weavers, load-time and run-time weavers share a number of characteristics and will henceforth be subsumed under the term dynamic weaving. In contrast to run-time weaving load-time weaving has one caveat [2]: "All aspects to be used for weaving must be defined to the weaver before any types to be woven are loaded." This applies not only to AspectJ, whose Development Environment Guide this quote is taken from, but to other languages supporting load-time weaving as well, for otherwise classes may be "*missed* by [weaving] aspects added later, with the result that invariants across types fail."

3. Surveyed Implementations

We have surveyed the latest incarnations of several aspect-oriented languages all of which support dynamic weaving.

3.1 AspectJ 1.6

The AspectJ programming language [12] is arguably the most prominent aspect-oriented language, whose implementations can utilize not only compile-time and post-compile time weaving but also load-time weaving. Furthermore, two alternative implementations of the latter exist: One uses a Java 5 agent and one uses a custom class loader. Both implementations take dynamic class loading into account; it is thus claimed [2] that they "compl[y] with the Java 2 security model." Of these implementations the agent-based one has been our main object of study. Not only is it the most recent, but also not subject to further issues as described in Section 5; it applies a `class` file transformation upon class definition but does not otherwise interfere with dynamic class loading.

3.2 AspectWerkz 2.0

In contrast to AspectJ, AspectWerkz [7] is not so much an aspect-oriented language but a framework. Still, just as for AspectJ, class-loader aware implementations exist. In fact, AspectWerkz comes with a number of alternative implementations of run-time weaving. Several of these are specific to a single VM, e.g., to the JRockit or HotSpot VM [17]. One implementation, however, is universally available across all VMs supporting the Java 5 platform. Consequently, this agent-based implementation has been surveyed.

3.3 JAsCo 0.8.7

The design goal of the JAsCo language [18] was to combine aspect-oriented and component-based software development. As the latter typically makes—at least in a Java environment—heavy use of dynamic class loading, it stands to reason that JAsCo should have a high level of class loader awareness. Beyond mere static weaving JAsCo also offers two implementations supporting run-time weaving by means

[3] For readability, the listing is presented as Java source code, even though advice weaving is typically performed at the level of Java bytecode.

of HotSwap or a Java 5 agent. As the former implementation is marked deprecated, we have chosen the latter.

4. Characterization of Class Loading Behavior

The presence of protection domains and class loaders [9, 14] within the Java VM gives rise to two crucial questions: Which protection domain ought aspects be assigned to? And which classes ought to be affected by which aspects? While Section 4.1 answers the former question, Section 4.2 attempts an answer to the latter.

4.1 Protection Domain Assignment

As the protection domains are assigned by the class loader and all surveyed implementations compile aspects to ordinary `class` files, it is natural that an aspect gets assigned a protection domain based on the source of said file. But while this assignment seemingly integrates aspects with the Java security model, it is oftentimes jeopardized by the weaving technique used.

The reason for this is that optimizing weavers may decide to replace the call to the synthetic advice method (cf. Section 2.2) with the advice body itself. This implementation technique, known as inlining [4], severs the link of an aspect to its protection domain; the advice's code becomes part of the join point shadow's class. As such it belongs to the latter's protection domain in the eyes of the VM. This fact can be exploited by an aspect which is woven into a trusted class. Due to inlining the aspect's protection domain is no longer recorded on the call stack when the inlined advice's action is performed; consequently, all permissions granted to the trusted class are granted to the aspect as well. To prevent this escalation of privileges we therefore postulate that code contributed by an aspect should always be treated as belonging to that aspect's protection domain. This is feasible even in the presence of inling as virtual machines already cope with similar situations by maintaining a mapping from machine code addresses to methods [1, 17].

It should be noted that the above issue is independent of the issues surrounding **`privileged`** aspects [8],[4] which allow an aspect to call methods or access fields otherwise inaccessible to it. In contrast to access modifiers of method or fields, protection domains control, e.g., whether a file I/O operation may be performed; thus, they govern a different set of privileges.

4.2 Namespace Separation

In addition to the above issue, inlining may violate namespace separation: When resolving symbolic references an inlined advice would use the join point shadow's defining class loader instead of the class loader its declaring aspect is defined by. Similar to the issue of protection domain assignment all code contributed by an aspect should thus be assigned the aspect's defining class loader.

But irrespectively of the weaving technique used, there is the more general question of which aspects may affect which classes. When addressing this question, it is useful to abandon the notion of weaving (cf. Section 2.2) for the moment, and rather discuss design decisions in terms of virtual method calls alone,[5] as they ultimately give rise to the join points in question.

Consequently we focus on the three classes involved during method dispatch: the caller's dynamic type, the callee's static type, and the callee's dynamic type. Of these only the former two are decisive during resolution [15, §5.4.3], whereas the latter is not considered by the Java VM when resolving the symbolic reference. The callee's dynamic type is, however, useful when deciding whether a method call like `f.delete()` ought to be advised or not—at least, the existence of **`execution`** atomic pointcuts [5] in all surveyed languages stipulates this. Consequently, we take all three types and their class loaders into account. Hereby, b will denote the dynamic type of the caller in the base program, c will denote the static type of the callee, and d will denote the dynamic type of the callee. Their defining class loaders will be denoted by l_b, l_c, and l_d.

Utilizing the notation introduced in Section 2 we now define four cases which describe the relative position of two class loaders l_a, l_z in the class loader hierarchy—under the assumption that the standard delegation model is adhered to. If not, the definitions below approximate the class loaders' relationship as suggested by the names chosen. So, let $a = \langle \text{"A"}, l_a \rangle$ and $z = \langle \text{"Z"}, l_z \rangle$.

Same The equation $l_a = l_z$ captures the simplest case possible: Both classes are defined by the same class loader.

Ancestor-or-same This case, denoted by $l_a \succcurlyeq l_z$, is defined by the equation $a = \langle \text{"A"} \rangle^{l_z}$. Under the above assumption it reflects the intuition that the class loader l_a resides above l_z in the hierarchy. (**Ancestor**, denoted by $l_a \succ l_z$, is defined analogously.)

Descendant-or-same This case, denoted by $l_a \preccurlyeq l_z$, is defined by the equation $z = \langle \text{"Z"} \rangle^{l_a}$ and reflects the intuition of l_a residing below l_z in the hierarchy. (The analogously defined **Descendant** is denoted by $l_a \prec l_z$.)

Sibling If neither of the above three equations holds, l_a and l_z are assumed to reside in different branches of the hierarchy. This case is denoted by $l_a \not\succcurlyeq\!\!\not\preccurlyeq l_z$.

Note that neither $\succcurlyeq$ nor $\preccurlyeq$ are guaranteed to be transitive if the standard delegation model is not adhered to. If the above assumption holds, however, transitivity does hold as well. In the following we will indicate this by a distinct notation: $\geqslant$ and $\leqslant$.

[4] Privileged aspects are not to be confused with privileged actions.

[5] The discussion carries over to field accesses and static method calls.

	l_b	l_c	l_d	**AspectJ AspectWerkz**	**JAsCo**	**Desired Design**
	$>$	$>$	$>$	✓	✓	✗
	$<$	$<$	$<$	✗	✗	✗
	$\not\gtrless$	$<$	$<$	✗	✗	✗
	$<$	$<$	$\not\gtrless$	✗	✗	✗
	$\not\gtrless$	$<$	$\not\gtrless$	✗	✗	✗
l_a	$\not\gtrless$	$\not\gtrless$	$\not\gtrless$	✗	✗	✗
	$\geqslant$	$\leqslant$	$\geqslant$	✓	✓	✓
	$\geqslant$	$<$	$<$	✓$^{\texttt{call}}$	✗	✓
	$\geqslant$	$<$	$\not\gtrless$	✓$^{\texttt{call}}$	✗	✓
	$<$	$<$	$\geqslant$	✓$^{\texttt{execution}}$	✓	✓
	$\not\gtrless$	$<$	$\geqslant$	✓$^{\texttt{execution}}$	✓	✓

Table 1: The class loading behavior exhibited by implementations of aspect-oriented languages. (AspectJ and AspectWerkz exhibit the same behavior and are thus subsumed.)

For all relations of the class loaders l_b, l_c, l_d to an aspect's class loader l_a Table 1 shows whether AspectJ, AspectWerkz, and JAsCo allow or disallow advice weaving. The rightmost column hereby shows the class loading behavior we deem most desirable and which will be argued for in the following. Note that certain combinations are not shown, as the constraints imposed by the Java VM during class loading require that $l_b \preccurlyeq l_c$ and $l_c \succcurlyeq l_d$.

According to the AspectJ language's Development Environment Guide [2] "[a] class loader may only weave classes that it defines." This statement is, however, grossly misleading. As Table 1 shows, execution environments for AspectJ do in many case weave calls when the aspect has been defined by a class loader different from the base program's. The above quote can only be understood in the context of the following rule: "All aspects visible to the weaver are usable. A visible aspect is one defined by the weaving class loader or one of its parent class loaders." Thus, advising a call made by b is allowed whenever $l_a \succcurlyeq l_b$.

As can be seen in the table's first row, each surveyed implementation weaves advice if the aspect's class loader l_a is an ancestor of the other three class loaders l_b, l_c, l_d. However, we deem this behavior undesirable as it allows an aspect a to advise method calls even if symbolic references to methods declared by c are not resolvable since $l_a > l_c$ implies that $l_a \not\preccurlyeq l_c$. This behavior then conflicts with the semantics Java as a base language exhibits in the light of reflection: It is not possible for a class a to obtain reflective access to a method declared by c. Weaving is also undesirable in situations where all three class loaders l_b, l_c, l_d are ancestors or siblings of l_a, as this would allow an aspect loaded by l_a to affect the behavior of classes loaded by the bootstrap class loader or comprising an applet container as well as other applets (cf. Figure 1). In fact, none of the surveyed implementations exhibits this behavior.

All other situations shown are desirable and for the most part supported by AspectJ, AspectWerkz, and JAsCo. The situations where $l_a \geqslant l_b$, $l_a < l_c$, and $l_a < l_d$ or $l_a \not\gtrless l_d$, however, are only partly supported by AspectJ and AspectWerkz and unsupported by JAsCo. Situations like these occur, e.g., if an applet managed by a container calls to a system class, and said container attempts to advise these calls to introduce access control (cf. Section 2.2). In AspectJ and AspectWerkz this is only possible by means of a **`call`** atomic pointcut, but not by an **`execution`** one, as the former causes advice weaving in the caller's class, whereas the latter weaves into the dynamic callee's class [5]. As JAsCo consistently performs **`execution`**-style weaving [19], this situation cannot be coped with by JAsCo. Similar restrictions apply to situations where $l_a < l_c$, $l_a \geqslant l_d$, and $l_a < l_b$ or $l_a \not\gtrless l_b$; they occur, e.g., if an applet registers a call-back with a system class. Hereby, the call-back's interface, i.e., its static type, is defined by the system class loader l_c and the call-back's implementation, i.e., its dynamic type, is defined by the applet's class loader l_d. The situation in the last row is of special interest, as weaving may violate the loading constraints imposed by the Java VM [15, §5.3.4] if the pointcut binds context relating to the (invisible) caller.

The main characteristic of the desired situations shown in Table 1 is that the aspect is visible from either the caller's or the callee's point of view: $l_a \succcurlyeq l_b$ or $l_a \succcurlyeq l_d$; this characterizes not all such situations, however, as the table's first row is exceptional. Furthermore, the table contains only those situations which may occur when adhering to the standard delegation model; pathological cases where, e.g., both $l_a \succ l_b$ and $l_a \prec l_b$ hold have been omitted.[6]

5. Further Issues

As mentioned in Section 3.1, AspectJ also comes with an implementation performing load-time weaving by means of a custom class loader. This implementation has one serious limitation, though: Maintaining multiple namespaces by using a hierarchy of class loaders is impossible. This is due to the fact that advice are woven by the `defineClass` method of the custom `WeavingURLClassLoader`. Thus, in order for a class to be affected, it has to be defined by this loader, which effectively supplants Java's application class loader. However, as the defining class loader, together with the fully qualified name, determines a class's identity, it is outright impossible to use two classes with the same fully qualified name (cf. Section 2.1.2); in a sense, the class loader hierarchy collapses into a single class loader.

[6] The complete data and the survey's setup are available to the public: `http://www.st.informatik.tu-darmstadt.de/static/pages/projects/ALIA/alia.html`.

Another issue is the requirement imposed by the dynamic weavers of both AspectJ and AspectWerkz to declare aspects in an `aop.xml` resource which accompanies the aspect's `class` file. Thus, the question arises whether the class loader l_r loading the resource or the class loader l_a defining the aspect's class determines the protection domain the aspect is assigned to. While it may seem reasonable to enforce the condition $l_a = l_r$, this class loading constraint prevents a valid use case: An aspect in a library defined by an ancestor class loader can selectively be enabled by declaring its use in an `aop.xml` file. Therefore, $l_a \geqslant l_r$ may be an alternative constraint well worth considering.

Finally, one noteworthy difference of JAsCo to the corresponding implementations of AspectJ and AspectWerkz is that JAsCo requires all so-called connectors, which enable JAsCo's aspects, to be loadable by the application class loader. As a direct consequence of this restriction, every aspect resides along with its connectors in the class loader hierarchy at the highest level reachable by ordinary application classes. But this severely limits the possibility to introduce aspect libraries and aspectual containers into the class loader hierarchy.

6. Related Work

While all of the surveyed aspect-oriented languages provide implementations which are aware of the Java VM's class loading facilities, little has been published on the design decisions beyond the level of developer documentation [2]. The need for controlling advice weaving, however, has been acknowledged and led to the proposal of a so-called aspect permission system [8]. The proposed system would ideally extend Java's `AccessController` framework (cf. Section 2.1.1) with permissions controlling whether a particular method call may be advised or not.

While this level of control may be useful, the use of the aforementioned framework requires not only a fine-grained policy, but also restricts the policy to one of "default deny." This generally is a sensible choice. It can be problematic when controlling advice, however, as one of the benefits of AOP is that often the base program can be oblivious of the aspects applied. Such obliviousness would be compromised if permissions needed to be granted to each aspect explicitly. Whether a policy of "default allow" is more desirable is, however, open to debate. In either case, restricting advice weaving by means of the namespace separation offered by Java's dynamic class loading facility nicely complements an aspect permission system as it offers a more coarse-grained mechanism.

7. Conclusion and Future Work

We have shown two shortcomings in the class loading behavior of several existing execution environments for aspect languages based on Java: A protection domain may be erroneously assigned when advice is inlined and namespace separation cannot always be guaranteed. Furthermore, we have identified both desirable and undesirable behavior for dynamic weaving in the presence of class loaders and characterized it analogously to virtual method calls. The desired design has been characterized by two simple conditions based on the static and dynamic type of the callee at join points. In particular, these conditions abstract away from implementation issues like the distinction between **`call`** and **`execution`** join points, which would introduce further subtleties [5] to the already subtle matter of dynamic class loading. We hope that a unified concept, e.g., virtual join points [6], will help to further clarify and formalize the class loading behavior of Java-based aspect-oriented languages. For the moment, however, we propose that the following guidelines be followed when implementing any form of dynamic weaving.

- Advice must be executed within the protection domain of its declaring aspect.
- Advice must resolve symbolic references with the class loader of its declaring aspect.
- Aspects should affect only method calls when visible to the caller's or callee's dynamic type—unless the callee's static type is in turn invisible to the aspect.

However, further design choices still need to be made. We thus seek to establish answers for the following three questions: If aspects are declared and defined by different resources, should constraints on the class loaders be enforced? How to best integrate an aspect permission system with class-based security? And how to secure advice weaving at privileged actions? Answering these questions can guide implementers to aspect languages which fully comply with the security model of the Java language.

Acknowledgments

This work was supported by the AOSD-Europe Network of Excellence, European Union grant no. FP6-2003-IST-2-004349.

References

[1] B. Alpern, C. R. Attanasio, A. Cocchi, D. Lieber, S. Smith, T. Ngo, J. J. Barton, S. F. Hummel, J. C. Sheperd, and M. Mergen. Implementing Jalapeño in Java. In *Proceedings of the 14th Conference on Object-oriented Programming, Systems, Languages, and Applications*, 1999.

[2] The AspectJ Project. *The AspectJ Development Environment Guide*. `http://www.eclipse.org/aspectj/doc/released/devguide/`.

[3] The AspectJ Project. *The AspectJ Programming Guide*. `http://www.eclipse.org/aspectj/doc/released/progguide/`.

[4] P. Avgustinov, A. S. Christensen, L. J. Hendren, S. Kuzins, J. Lhoták, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam,

and J. Tibble. Optimising AspectJ. *ACM SIGPLAN Notices*, 40(6), 2005.

[5] O. Barzilay, Y. A. Feldman, S. Tyszberowicz, and A. Yehudai. Call and execution semantics of AspectJ. In *Proceedings of the 3rd Workshop on Foundations of Aspect-oriented Languages*, 2004.

[6] C. Bockisch, M. Haupt, and M. Mezini. Dynamic virtual join point dispatch. In *Proceedings of the 4th Workshop on Software Engineering Properties of Languages and Aspect Technologies*, 2006.

[7] J. Bonér. AspectWerkz. In *Proceedings of the 3rd Conference on Aspect-oriented Software Development*, 2004.

[8] B. de Win, F. Piessens, and W. Joosen. How secure is AOP and what can we do about it? In *Proceedings of the 2006 Workshop on Software Engineering for Secure Systems*, 2006.

[9] L. Gong, M. Mueller, H. Prafullchandra, and R. Schemers. Going beyond the sandbox: An overview of the new security architecture in the Java development kit 1.2. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, 1997.

[10] J. Gosling, W. N. Joy, G. L. Steele, and G. Bracha. *The Java Language Specification*. Addison-Wesley, 3rd edition, 2005.

[11] E. Hilsdale and J. Hugunin. Advice weaving in AspectJ. In *Proceedings of the 3rd Conference on Aspect-oriented Software Development (AOSD)*, 2004.

[12] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In *Proceedings of the 15th European Conference on Object-oriented Programming*, 2001.

[13] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proceedings of the 11th European Conference on Object-oriented Programming*, 1997.

[14] S. Liang and G. Bracha. Dynamic class loading in the Java virtual machine. In *Proceedings of the 13th Conference on Object-oriented Programming, Systems, Languages, and Applications*, 1998.

[15] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. The Java Series. Prentice Hall, 2nd edition, 1999.

[16] H. Masuhara and G. Kiczales. Modeling crosscutting in aspect-oriented mechanisms. In *Proceedings of the 17th European Conference on Object-oriented Programming*, 2003.

[17] Sun Microsystems. *The Java HotSpot Server VM*. `http://java.sun.com/products/hotspot/docs/general/hs2.html`.

[18] D. Suvée, W. Vanderperren, and V. Jonckers. JAsCo: an aspect-oriented approach tailored for component based software development. In *Proceedings of the 2nd Conference on Aspect-oriented Software Development*, 2003.

[19] System and Software Engineering Lab, Vrije Universiteit Brussel. *JAsCo language reference 0.8.6*. `http://ssel.vub.ac.be/jasco/lib/exe/fetch.php?media=documentation%3Ajasco.pdf`.