Distributing Classes with Woven Concerns – An Exploration of Potential Fault Scenarios

Nathan McEachen
nathan@mceachen.us

Roger T. Alexander
rta@cs.colostate.edu

Colorado State University
Department of Computer Science
601 S. Howes Street
Fort Collins, Colorado 80523 USA
01-970-491-7026

ABSTRACT

Aspect-oriented programming (AOP) promises to benefit software engineering by providing a layer of abstraction that can modularize system-level concerns. AOP is still a very young area of research and has yet to receive mainstream acceptance in industry. As a result, the industry as a whole lacks experience and knowledge concerning long term maintenance issues with AOP in deployed commercial applications. Fault models that result from software maintenance in aspect-oriented software development (AOSD) are not nearly as well understood as they are for object-oriented software development (OOSD). This paper will explore some of the long-term maintenance issues that can occur with AspectJ, which is an implementation of AOP for the Java programming language. More specifically, the ability of AspectJ (as of version 1.2) to weave into existing bytecode that already contains woven aspects can create unexpected and potentially unsolvable problems. This will hopefully lead to further discussion in the research community that will result in a better understanding of the long-term maintenance issues inherent in AOSD.

Categories and Subject Descriptors

D.2.5 [**Testing and Debugging**]: Diagnostics .

General Terms

Reliability, Languages, Verification.

Keywords

Aspect-orientation, AspectJ, foreign aspects, aspect interference.

1. INTRODUCTION

Java classes are often distributed in the form of bytecode as either class or jar files. This allows classes to be distributed for reuse without exposing the original source code. Software developers who wish to use these classes need not recompile them, but only need to compile their own classes that are clients of the distributed classes.

© ACM, 2005. This is the author's version of the work. It is posted here by permission of ACM for your personal use. Not for redistribution.

As of version 1.1, AspectJ allows the weaving of aspects into existing bytecode. Aspects can be woven into classes that exist as bytecode in a jar file by using the `-injars` compile option. Version 1.2 replaces the `-injars` compile option with `-inpath`, which allows directories of class files and jar files to be specified [1]. Therefore, the source code of a distributed class is no longer required for aspects to be woven into it.

The ability to weave concerns into existing bytecode opens the door to a new realm of possibilities in software development. Concerns such as tracing, thread safety, and resource pooling [8] can now be extended into classes that we import into our application from another source. Undoubtedly, research in this area will continue to reveal new and beneficial applications of this ability.

Not having access to the source code of imported classes also has its drawbacks. It is not immediately clear what the impact of weaving new concerns into the imported bytecode will be. The problem is compounded by the possibility that the imported bytecode may already contain woven aspects.

We refer to an aspect woven into a class (or a set of classes) with the resultant bytecode being imported by another party not having access to the aspect source code, as *a foreign aspect*.

It is important to understand that, in AspectJ, woven aspects (foreign and our own) are still identifiable as aspects within the bytecode. This is the minimal prerequisite for enabling *aspect reweaving*. This concept is introduced in AspectJ 1.2[1] along with the `Xreweavable` option, which produces class files containing annotations that enable later reweaving. Reweaving means that the entire code base is woven again[2]. During this process, all available aspects (foreign and our own), are applied to all available aspects and base classes (foreign and our own). This allows independently developed code to be combined in possibly unanticipated ways: (1) foreign aspects are applied also to our own base classes, (2) our own aspects are applied to foreign bytecode, and (3) independently developed aspects are jointly applied to the same (our own or foreign) code.

---

[1] AspectJ 1.1 does not allow the weaving of aspects into bytecode that contains woven aspects. Attempting to do so will generate an error by the weaver.

[2] Aspect classes produced during the original weave must be present during the reweave [1].

This paper focuses on the problems resulting from the *unanticipated composition* of aspects and base classes that arises when foreign aspects are rewoven. Our discussion shows that AspectJ's reweaving capability enables additional expressiveness but the language is not yet mature enough to deal with the additional problems resulting from unanticipated composition. We conclude that proper handling of unanticipated composition requires language extensions that convey additional information to the aspect weaver and powerful interference analysis techniques that can exploit this information.

The remainder of this paper is organized as follows. Section 2 discusses these problems in detail. Section 3 then presents a discussion of the significance of these problems and how bytecode reweaving can make systems extensible. Section 4 offers guidelines to aid developers that make use of bytecode weaving using existing AspectJ implementations. Section 5 follows with a discussion of the limitations of existing technology for bytecode weaving and includes suggestions for improvements. It makes the case that metadata that captures semantic properties need to be present in foreign aspects. Section 6 presents related work. Section 7 describes contributions and future work. Finally, section 8 concludes the paper.

2. PROBLEMS WITH FOREIGN ASPECTS

Introducing foreign aspects into a software system can change the behavior of that system in unexpected and undesired ways. Interference from foreign aspects negatively affects program correctness, comprehension and maintenance. AspectJ leaves programmers completely unaware of foreign aspects in imported bytecode, and offers no help in documenting or analyzing their impact on the importing code. Programmers might realize the existence of foreign aspects just by observing faults of their applications.

The next subsection introduces *unbounded pointcuts* and how they contribute to *unanticipated composition.* The remaining subsections provide concrete examples that demonstrate how limitations of AspectJ manifest problems that result from unanticipated composition. Each subsection includes a description of the problem, a scenario in which the problem can occur, the observable faults resulting from the problem in the analyzed scenario, and the resulting consequences for programmers.

2.1 UNBOUNDED POINTCUTS

Bounded pointcuts are pointcuts defined such that they only match join points from packages and classes present in the original environment that the aspect was specifically developed and tested for. *Unbounded pointcuts* can potentially match join points beyond the original environment. The *reweaving* of *non-orthogonal* aspects that use unbounded pointcuts raises the problem of *unanticipated composition* of aspects with foreign aspects and foreign base classes. Unanticipated composition can lead to 1) unbounded pointcuts not capturing all intended join points, 2) unbounded pointcuts capturing unintended join points, 3) multiple and mutually unknown aspects capturing overlapping join points.

In many cases, unbounded pointcuts are necessary, as they enable foreign aspects to apply to importing code during aspect reweaving. This need is illustrated in the following sections concerning partial weaving and unknown aspect assumptions (2.2 and 2.3 respectively). These sections demonstrate that faults can occur if an unbounded pointcut does not capture all intended join points. The section on unintended aspect effects (2.4) illustrates the opposite problem, where unbounded pointcuts can capture unintended join points. Section 2.5 demonstrates that problems can occur even if unbounded pointcuts capture exactly the intended join points. More specifically, interference can occur when the same join points are captured by multiple and mutually unaware aspects.

2.2 PARTIAL WEAVING

Aspects with unbounded pointcuts must always be rewoven when imported in a new environment in order to apply their effects to the new join points in that environment. Otherwise, the program will be only *partially woven*, resulting in improper behavior. The following scenario demonstrates this problem.

2.2.1 Scenario: Inheritance

Pointcuts can be defined for all subtypes of a given class. Advice that uses such a pointcut will be woven into all subtypes. Figure 1 lists an aspect that applies a before advice to method `m()` for all subtypes of class `A`. Figure 2 lists class `A`, and class `B` that extends class `A`. Figure 3 is a class that demonstrates that the aspect is woven into all subtypes of `A`.

```
public aspect SubTypeAspect {
  protected pointcut somePointCut()
    :  execution (* A+.m());

  before() : somePointCut() {
    System.out.println("Before concern");
  }
}
```

Figure 1

```
public class A {
  public int i;
  public A() {}
  public void m() {
    System.out.println("Inside A.m()");
  }
}
public class B extends A {
  public B() {}
  public void m()
  {
    System.out.println("Inside B.m()");
  }
}
```

Figure 2

```
public class Test {
  public static void main(String[] args) {
    A a = new A();
    a.m();
    B b = new B();
    b.m();
  }
}
```

Figure 3

Compiling these classes and running the main method of the `Test` class produces the following:

```
Before concern
Inside A.m()
Before concern
Inside B.m()
```

Now assume that the concern implemented in `SubTypeAspect` is required for method `m()` to be semantically correct for all classes that extend `A`. One reason would be that `SubTypeAspect` maintains an inherited invariant. Suppose further that the class files produced from the above weave were imported in another program and the importing program contains a Class `C` that extends `B`, as illustrated in Figure 4. `C`'s author did not write `B` and does not have access to `B`'s source code or knowledge of the `SubTypeAspect` aspect.

```
public class C extends B {
  public C() {}
  public void m()
  {
    System.out.println("Inside C.m()");
  }
}
```

Figure 4

2.2.2 *Faults: Fragile subclasses*

Figure 5 modifies the test class to include a call to `C.m()`.

```
public class Test {
  public static void main(String[] args) {
    A a = new A();
    a.m();
    B b = new B();
    b.m();
    C c = new C();
    c.m();
  }
}
```

Figure 5

Compiling class `Test`, `C`, and the imported class files using the normal java compiler and running the main method of the `Test` class will produce the following:

```
Before concern
Inside A.m()
Before concern
Inside B.m()
Inside C.m()
```

This shows that the subclass will not exhibit the effects of `SubTypeAspect` since the foreign aspect is not woven into `C`. Since the aspect is required in order for all subtypes of `A` to be semantically correct, subclass `C` will not behave correctly – its subclass contract will be broken. We call this fault “fragile subclasses” since it is the complement of the famous “fragile base class problem” of object-oriented development [9].

Note that the problem of partial weaving is easy to fix. The author of `C` just needs to use AspectJ (instead of the normal Java compiler) with the `-inpath` option pointing to the class file of `B`. This will force a reweave of the `SubTypeAspect`. The real problem here is that there is no way that the author of `C` can be made aware that the imported bytecode contains an aspect. The programmer of `B` might not have documented this in his code, since he might have been himself oblivious of the use of `SubTypeAspect` or might have considered this an implementation detail that he wanted to hide intentionally. The Java compiler will not complain either, since its specified behavior is to ignore any non-standard attributes in class files.

2.2.3 *Consequences*

Programmers of classes importing third-party bytecode need to be aware that foreign aspects must be rewoven in order to prevent the problems resulting from partial weaving. Lacking certainty that imported bytecode is aspect-free, cautious programmers need to compile the entire code (their own and imported) with AspectJ using the `-inpath` option pointing to all imported class files. Whether this is just an inconvenience or a serious problem depends on the personal work environment of affected programmers. Unfortunately, the major drawback of the need to force reweaving is that it prevents partial weaving at the expense of raising a series of other problems, as illustrated in the next subsections.

2.3 UNKNOWN ASPECT ASSUMPTIONS

Pointcuts encode assumptions about the base programs to which an aspect is applicable. These assumptions are problematic for the authors of foreign aspects and for authors of code that import those aspects. Authors of foreign aspects have no guarantee as to whether the assumed join points specified in the pointcuts will be present in unknown importing code. Similarly, authors of importing code might not know which join points they must provide (or must avoid) for the imported aspects to work as intended by their original authors. The observable fault is always the same: foreign aspects turn out to be fragile aspects.

2.3.1 *Scenario: Violated assumptions*

Figure 6 defines class `A` having public method `m()` that calls protected method `f()`.

```
public class A {
  public void m() {
    System.out.println(
      "Inside A.m() - make a call to this.f()");
    this.f();
  }
  protected void f() {
    System.out.println(
      "Inside A.f() - " +
      "Some functionality required by m()");
  }
}

public aspect SubTypeAspect2 {
  protected pointcut somePointCut()
  :  call (* A+.f()) && (withincode(* A+.m()));

  before() : somePointCut() {
    System.out.println(
      "Aspect woven for " +
      thisJoinPoint.getKind()+ " \"" +
      thisJoinPoint.getSignature() + "\" at " +
      thisJoinPoint.getSourceLocation());
  }
}
```

Figure 6

Figure 6 also shows SubTypeAspect2 that defines a pointcut and advice such that the advice is applied before every invocation of f() in m(). The author of SubTypeAspect2 made the assumption that a subset of the behavior of m() will be implemented through a call to method f() for all subclasses of A.

Now consider a program having a class B that extends A and overrides m() with an implementation that directly implements the effect of f() rather than invoking A's implementation of f() directly (Figure 7). As a result, the behavior for the concern represented by somePointCut in SubTypeAspect2 is excluded from B's implementation of m(), Consequently, the assumptions made by the foreign aspect author are not preserved under this new structure since the author of B is unaware of those.

```
public class B extends A {
  public void m() {
    System.out.println(
     "Inside B.m() - " +
     "Implements the effect of f()");
  }
}
```

Figure 7

2.3.2 *Faults: Fragile aspects*

Below is the output that is produced from the Test class in Figure 3 when the foreign aspect SubTypeAspect2 is rewoven into the importing code.

```
Inside A.m() - make a call to this.f()
Aspect woven for method-call "void A.f()" at
A.java:18
Inside A.f() - Some functionality required by m()
Inside B.m() - Implements the effect of f()
```

This example shows that the subclass will not exhibit the effects of SubTypeAspect2. Although AspectJ was used to reweave the foreign aspect into the imported code, the imported code did not provide the required join point. This is a type of "fragile aspect", where failing to provide the required join point will prevent the aspect from being woven where it was intended.

2.3.3 *Consequences*

Programmers must be aware of any join point requirements that foreign aspects have on importing code. Although it is easy to impose such requirements, it might be difficult for the authors of the importing code to fulfill them.

2.4 UNINTENDED ASPECT EFFECTS

Importing classes that contain aspects having unbounded pointcut definitions is problematic. Unbounded pointcut definitions that rely heavily on wildcards have the potential to capture join points in classes they were not intended to apply.

As a result, these foreign aspects can be unexpectedly woven into classes during the reweave process. Without using a development tool that could, at a minimum, provide a warning that this had occurred, the new class author would remain oblivious.

2.4.1 *Scenario: Policy enforcement*

With AspectJ, we can write aspects that aid in software development. For example, policy enforcement aspects can ensure that developers adhere to given programming practices and rules. Such aspects can be considered orthogonal to others since they do not introduce any behavior. However, even a policy enforcement aspect can create problems if it is woven into a class that is distributed as bytecode.

If a policy is defined by a pointcut that is unbounded, then the policy could be applied to importing code during reweaving. Figure 8 shows a sample policy enforcement aspect, EnforcePublicAccessMemebres. This aspect states that setting the value of a public class attribute must occur within a class constructor, set, or get method.

```
public aspect EnforcePublicAccessMembers {
  pointcut publicAccess()
    : (set(public * *))
    && (!withincode(*.new(..)))
    && (!withincode(* *.set*(..)))
    && (!withincode(* *.get*()));

  declare error: privateAccess() :
    "Please use a setter method "
    + "when defining public member variables";
}
```

Figure 8

Figure 9 lists two classes, ServerA and ClientA. ClientA can read the public member attribute of ServerA but must use a set method to define its value. ClientA does not violate the policy.

```
public class ServerA {
  public int i;
  public ServerA() {
    this.i = 1;
  }
  public void setI(int i) {
    this.i = i;
  }
}
public class ClientA {
  public ClientA() { }
  public void m() {
    ServerA serverA = new ServerA();
    serverA.setI(2);
    System.out.println( "Value from ServerA.i = "
                        + serverA.i );
  }
}
```

Figure 9

```
public class ServerB {
  public int i;
  public ServerB() {
    this.i = 1;
  }
  public void setI(int i) {
    this.i = i;
  }
}
public class ClientB {
  public ClientB() { }
  public void m() {
    ServerB serverB = new ServerB();
    serverB.i = 2;
    System.out.println( "Value from ServerB.i = "
                        + serverB.i );
  }
}
```

Figure 10

Figure 10 lists another set of classes, `ServerB` and `ClientB`. `ServerB` is identical to `ServerA`. `ClientB` is almost identical to `ClientB`, except that it assigns a value to a public member variable of `ServerB` without using a set method.

2.4.2 Faults: Failed compilation with obscure diagnostics

Assume that `EnforcePublicAccessMembers`, `ServerA`, and `ClientA` are compiled using the `–Xreweavable` option, and that `ServerA` and `ClientA` are imported as bytecode into the same application as `ServerB` and `ClientB`. Also assume that reweaving is triggered, for instance, because some other aspect needs to be woven into that bytecode. During reweaving the policy enforcement aspect will be applied to `ServerB` and `ClientB`. As a result, the following compilation error will occur:

```
ClientB.java:9 error Please use a setter method when defining
public member variables

serverB.i = 2;

^^^^^^^^^^^^^^

        field-set(int ServerB.i)

        see also: BinaryPolicyEnforcement2\EnforcePrivateAccess
Members.java:9


1 error
```

The error message refers to a line of source code from the aspect that defines the policy. However, in this example the source code was not distributed, making it impossible for a developer to view the definition of the policy that was violated.

2.4.3 Consequences

If bytecode containing foreign policy enforcement aspects with unbounded pointcuts is rewoven, and the policy is violated in the importing code, the compilation of the importing code will be aborted. To reweave imported bytecode, all importing code would therefore need to adhere to any foreign policy. This will happen even though the policy may not be relevant to the importing code.

In a reasonably large application, even a simple and well-documented policy would make it too prohibitive a task to rewrite the importing classes to conform to the foreign policy. Moreover, due to the lack of documentation of policies hidden in imported bytecode, programmers will not know how to implement foreign policies, no matter how trivial they might be.

2.5 ARBITRARY ASPECT PRECEDENCE WHEN INDEPENDENTLY DEVELOPED ASPECTS ARE APPLIED TO THE SAME JOIN POINT

A join point may be captured by multiple foreign aspects from different authors. Authors of foreign aspects do not have knowledge of any additional aspects that are present in our code (either foreign or our own). As a result, the authors of foreign aspects cannot define explicit precedence between other foreign aspects used by importing code.

The inability to define explicit precedence for foreign aspects is compounded by the fact that default aspect weave precedence is completely arbitrary. It varies between different AspectJ compiler implementations [8], and even between different runs of the same compiler implementation on slightly changed input programs. Foreign aspects could be rewoven in a different order if a different version of the AspectJ compiler is used or if an aspect is moved to another package. We demonstrate that the latter effect occurs even if the moved aspect is completely orthogonal with respect to other aspects or classes (e.g. a tracing aspect).

2.5.1 Scenario: Swing thread safety aspect

Assume that both a client and an imported module relied on their own implementation of Swing's single-thread rule [13] that specifies which Swing operations are executed synchronously and which are executed asynchronously. In this setting we want to understand what happens if the author of the client module wished to ensure that the rule is enforced also in the imported module, while unaware that it is already enforced by an aspect within that module.

To test this interference scenario, we created two packages called `swingWorker1` and `swingWorker2`. Each contains aspects that implement Laddad's solution [8] to the Swing single-thread rule. The solution uses pointcuts that specify which Swing methods are executed synchronously and asynchronously. To approximate a realistic scenario, we used two implementations that varied only by their pointcut definitions[3]. The aspect in `swingWorker1` specified synchronous execution for method calls to `JOptionPane`, while the aspect in `swingWorker2` specified asynchronous execution.

The test was run using a class written by Laddad ([8], page 290) in order to determine which aspect from the two packages would be woven first. Logging aspects from Laddad were also included in package `swingWorker1` to trace the behavior of the aspect chosen first by the weaver. The `swingWorker2` package was compiled using the `–Xreweavable` option and placed into a jar file. The `swingWorker1` package was compiled with the `–inpath` including the `swingWorker2` package jar file.

2.5.2 Faults

The results of the first test revealed that the aspect in the `swingWorker1` package was chosen first by the weaver (see Figure 11).

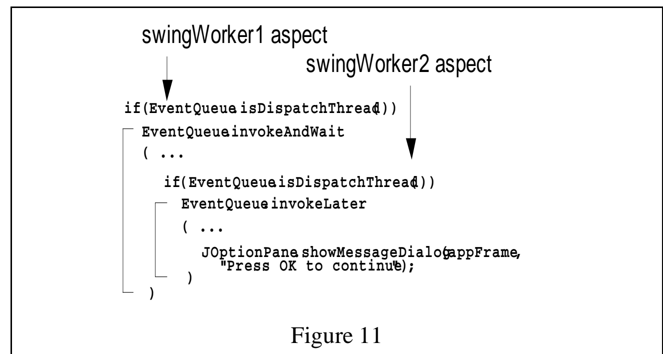

Figure 11

Control of the program halted when the message dialogue opened by the test program was rendered. Below is a fragment

[3] If really developed independently, the two aspects would have probably differed much more. However, the details of the tested example were not relevant for our purpose.

of the logging aspect output indicating that the call opening the dialogue was synchronous:

```
Executing operation synchronously
Executing:
void
javax.swing.JOptionPane.showMessageDialog(Component,Object)
Thread[AWT-EventQueue-0,6,main]
```

However, when the logging aspects were defined in the `swingWorker2` package, rather than the `swingWorker1` package, the weaver chose to weave the Swing single-thread aspect from the `swingWorker2` package first (see Figure 12).

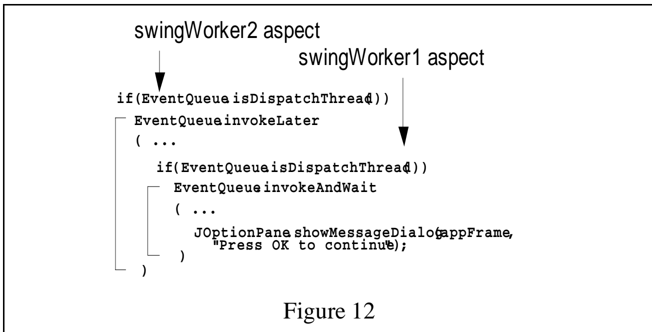

Figure 12

As a result, control of the program did not halt when the message dialogue was rendered. Below is a fragment of the logging aspect output indicating that the call to `JOptionPane` was asynchronous:

```
Executing operation asynchronously
Executing:
void javax.swing.JOptionPane.showMessageDialog(Component,Object)
Thread[AWT-EventQueue-0,6,main]
```

2.5.3 Consequences

Logging is an orthogonal concern with respect to the rest of the concerns implemented in the two packages. It should not influence the observable behavior of the other code beyond its intended purpose. However, since no explicit precedence was defined, the presence of logging influenced the weave order decision made by the AspectJ weaver. In this example, the consequences of that decision resulted in varying behavior. Such varying behavior could cause a fault in a production system. Without any knowledge of the presence or behavior of foreign aspects, such a fault could be very difficult to trace.

3. DISCUSSION

The root problem of the fault scenarios we have explored is a *lack of awareness* of the existence of foreign aspects and their semantic properties. Since foreign aspects are hidden, we need to increase our awareness of them.

There are *three basic awareness criteria* that, when satisfied, can prevent faults caused by foreign aspects.

1) We need to be aware of the presence of foreign aspects when we import bytecode that contain them.

2) We need to know the pointcut definitions for each advice in a foreign aspect.

3) We need to know the semantic impact such advice will have on our own code.

Currently, none of this information is easily available when we import bytecode and use the bytecode reweaving capabilities of AspectJ.

In addition to the awareness criteria, developers need a way to *explicitly control where foreign aspects are woven*. Such a mechanism is needed for developers who import aspects as well as developers who distribute aspects. Without support to enable awareness or explicit weave control in AspectJ concerning foreign aspects, the current best course of action is to take steps to avoid such fault scenarios as much as possible. We provide developer guidelines in section 4 for dealing with foreign aspects.

Extensible systems should have a mechanism to prevent extensions from interfering with each other. According to Szyperski, “The key requirement is that a checked and unmodified unit shall under no circumstance be invalidated merely by adding another unit to the system.” [12] However, it is acknowledged that incompatibilities between mutually unaware modules will, at some point, need to be resolved.

For AspectJ aspects to become extensible systems, standards must be developed that will help prevent modules from interfering that originate from different sources [12]. However, currently only a partial precedence of aspects within an extensible system can be defined, since authors of aspects cannot predict the future interaction of aspects from different authors. The problem is compounded by the fact that aspect precedence for one method might be completely inappropriate for another [10]. Also, according to Störzer, it is not safe to rely on naming conventions to prevent the undesired interference of aspects, since conventions are not guaranteed. Wildcards can easily and accidentally include an unwanted join point or miss a desired join point [11].

4. DEVELOPER GUIDELINES

The following subsections present guidelines and advice to assist AspectJ developers writing programs that use foreign aspects.

4.1 AVOID UNBOUNDED POINTCUT DEFINITIONS

Developers should avoid using unnecessary unbounded pointcut definitions. Such pointcut definitions rely heavily on wildcards for pattern matching, but are not explicitly bound to a set of packages or classes.

To prevent unintentional weaving of advice into additional classes in new environments, developers should explicitly restrict pointcut definitions to packages and classes present in the original environment that the aspect was specifically developed and tested for. This will help prevent unintentional weaving of advice into additional classes in new environments.

4.2 AVOID OVERLY BOUNDED POINTCUT DEFINTIONS

While bounded pointcut definitions can prevent certain faults when importing foreign aspects, under certain circumstances they can prove too restrictive. For example, if a foreign aspect implements the *observer pattern* for a particular class hierarchy, then a new child class defined in a different package from the imported hierarchy would not participate in the pattern. The

bounded pointcut definition would not match join points from the new child.

Therefore, developers must be aware of the circumstances in which foreign aspects need bounded pointcut definitions, and in which circumstances unbounded pointcut definitions are necessary.

4.3 GIVE DEVELOPERS WHO IMPORT FOREIGN ASPECTS LIMITED ABILITY TO CONTROL WEAVING

Developers not only need to be aware of pointcut definitions in foreign aspects, they also may need the ability to modify those pointcut definitions to control which join points in their own code are included or excluded during weaving. Ideally, AspectJ (and other AOP languages) would include specific language mechanisms that support this functionality. However, as that currently is not the case, this functionality can be achieved effectively (at least for AspectJ) with current language features (though not elegantly).

The author of a foreign aspect must begin by determining which pointcuts should be available for modification by the authors of importing code (which we do not consider further here). Once this is done, the next step is to define an abstract aspect that includes a single concrete pointcut definition for each concern whose weaving is to be controllable by a developer who is importing the aspect. Each such concrete pointcut defines the pattern of join points that will be selected at weave time. Finally, the foreign aspect developer implements one or more concrete aspects that are derived from the abstract aspect and that satisfy the behavioral needs of the distributed library. The source of the abstract aspect is included with the distributed library.

To control the reweaving of foreign aspects, a developer need only access the source file of the abstract aspect contained in the distributed library. Modifications are then made, as necessary, to the definitions of the concrete pointcuts that correspond to concerns of interest. Subsequent reweaving of code using the distributed library will be influenced and controlled by those modifications.

4.4 LIMIT JOIN POINT REQUIREMENTS ON IMPORTING CODE

When writing aspects, minimize join point requirements for importing code. Even if the authors of importing code are aware of such requirements, it may be difficult or impossible for them to provide the required join points.

4.5 AVOID SHARING OF POINTCUTS AMONG CONCERNS

Pointcut definitions should not be shared by more than one concern in a foreign aspect. Sharing pointcut definitions among concerns reduces the ability of a developer to selectively include or exclude individual concerns implemented by foreign aspects (section 4.3).

A concern is implemented by one or more aspect advice in a distributed library. For example, a security concern may be implemented using a set of advice that collectively provides the required behavior. In this case, you would only want to include or exclude the entire set of advice as a whole, not piecemeal.

4.6 ADEQUATELY DOCUMENT POINTCUTS

The documentation for each pointcut should include a detailed description of the advice that utilizes that pointcut. The description should minimally identify the concern that the advice addresses, the type of classes that the author believes the advice is applicable to, any join point requirements, and the semantic impact that the advice will have on those classes.

This documentation would give future developers the ability to include or exclude any foreign advice from being woven into their own code from a foreign aspect (see section 4.3). This approach would satisfy the first two criteria of awareness for foreign aspects. It will also allow developers to define explicit precedence of their own aspects with respect imported foreign aspects. However, the third criterion is only partially satisfied.

5. LIMITATIONS OF EXISTING TECHNOLOGY

Even if our suggested practices are followed, developers cannot truly ascertain the impact that foreign aspects will have on their own code. They still depend on the diligence and discipline of the authors of foreign aspects to provide awareness and make adequate documentation available. However, natural language can be ambiguous and has limited expressive power. As a result, developers still may have difficulty predicting the semantic impact of reweaving foreign aspects.

The problem is made more apparent when multiple aspects from different sources are imported. Each time developers import additional bytecode, the entire software system needs to be re-verified for correctness. Pointcut definitions modified for one foreign aspect may now not be appropriate once additional code is imported from another source.

A general solution to the presented problems can only be achieved by much more powerful aspect interference analysis techniques than we know today. Interference analysis needs to be applicable to aspects that come from different sources and contain no mutual precedence specifications.

We believe that aspects should be distributed with metadata that document properties of those aspects. At a minimum, the annotations should document the concern that is addressed by an aspect. Ideally, they would also capture semantic properties. As demonstrated in the Swing thread-safety example (section 2.5), aspects that address the same concern can interfere with each other. Laurent Bussard et al. [2] also recognize that AspectJ cannot prevent the weaving of incompatible aspects. Their example includes two aspects that address the same concern but use incompatible implementations. Resolving such conflicts is easy, yet difficult to detect. A tool could use metadata to automatically determine when advice from foreign aspects would produce semantic interference when they are rewoven in new environments.

It would also be beneficial for developers to guarantee the weave order of the aspects they distribute. Weave annotations in bytecode could instruct future weavers to preserve a given weave order when an aspect is rewoven in a new environment. This would ensure that bytecode remains semantically consistent when it is distributed and rewoven.

6. RELATED WORK

Some of the problems inherent in foreign aspects have been identified in other areas of AOSD. Previous research has identified the need to adapt and modify aspects when they are placed in diverse environments. Laurent Bussard et al. [2] recognize that aspects or their composition may need to be adjusted when aspects are reused in different contexts. They claim that specifications could be written to indicate aspect compatibility. Such specifications could be checked at composition time to detect conflicts between aspects.

Kniesel et. al [7] and Costanza et. al [3] point out the problems of unanticipated aspect composition and, in particular, the difficulties resulting from mutual interactions of aspects from different sources. They identify a class of aspects for which interferences do not require changes in the program but can be resolved automatically by a more powerful weaver.

Hanenberg and Unland [5] suggest introducing a parameter mechanism into the weave process. The parameters indicate to the weaver how to produce the actual woven code. Their work focuses primarily on static introductions, and they claim that parametric introductions will allow libraries of reusable introductions to be developed.

Tourwé, Brichau, and Gybels [14] identify software evolution problems with AOSD implementations that use a syntactic crosscut language. They point out that syntactic crosscutting languages introduce tight coupling between aspects and programs. They also recognize that naming conventions of application frameworks can conflict. The same problem exists with foreign aspects, as the naming conventions used by foreign aspects from multiple sources will most likely differ.

Gybels and Brichau [4] discuss using a logic-programming based crosscut language that allows more advanced pattern-based crosscuts. They report that their crosscut language weakens the coupling of aspects to core programs. However, more powerful pattern-based crosscut languages will not necessarily prevent the detection of semantic interference from foreign aspects.

Hanenberg and Unland [6] identify scenarios where pointcuts need to be redefined when a class is extended through inheritance. This is the same problem with foreign aspects, in that pointcuts sometimes need to be redefined whenever we import a module. To address this, Hanenberg and Unland also recommend defining pointcuts exclusively in abstract aspects and limiting pointcut definitions to only once advice.

7. CONTRIBUTIONS AND FUTURE WORK

In this paper we have demonstrated that imported modules containing woven aspects can potentially change the semantics of a client class. This paper is also unique in that we have identified how known limitations of AOSD technologies can create unexpected and difficult to detect faults when the bytecode weaving capabilities of AspectJ are utilized. To help developers overcome these difficulties, we have provided guidelines and have identified limitations in the current technology.

We also note that the limited awareness and control of foreign aspects makes the detection and resolution of these faults much more difficult, leading us to the conclusion that there is a need for a mechanism to give developers the ability to guarantee specific weave orders, scope, and precedence, regardless of how classes are integrated into other aspect-aware systems.

Many of the examples in section 2 using AspectJ are trivial and somewhat contrived. They primarily demonstrate the possibility of certain fault scenarios. It would be useful to determine the likelihood of such a fault occurring in a real software development project. Of those faults, what percentage will produce an undesired semantic change? This would depend heavily on the level of acceptance from industry of the new bytecode weaving features of AspectJ 1.2.

As AOP emerges into mainstream software development, more and more developers will use aspects to implement concerns. Of those aspects, what percentage will be distributed in a form that will cause them to be rewoven by a client (e.g. using the `-Xreweavable` option)? What new design patterns will emerge that will rely on the ability to weave aspects into an imported module? How often will this occur? For every imported foreign aspect, how do we avoid the need for late global analysis to verify correctness?

The existing AspectJ precedence mechanism does not easily prevent interference from aspects in an extensible system. Developers are, to an extent, dependent on the diligence and discipline of the authors of the modules we import. Poor awareness of the presence or behavior of foreign aspects makes testing extensible systems more difficult. We believe that additional information needs to be coupled with foreign aspects to increase aspect awareness and understanding of their scope and behavior. This additional information needs to document the semantic behavior of each aspect. This could allow a developer, or possibly a development tool, to identify aspects from varying sources that are semantically equivalent, orthogonal, or interfering.

As we saw with the Swing thread-safety example (section 2.5), aspects that are almost semantically equivalent (or address the same concern but with different implementations) and have overlapping join points can produce hard to detect and subtle errors. Additional information could aid a developer in resolving such conflicts. Future research should explore other kinds of information that would aid in the integration of foreign aspects, as well as investigate the feasibility or the mechanism of embedding such semantic annotations with distributed bytecode.

8. CONCLUSIONS

AspectJ has evolved much since its inception. Initially, aspects were woven into source code, allowing developers visibility into the behavior of the woven system. They could visually trace through the source code to determine if the resultant weave order was the desired weave order.

With AspectJ 1.1, we lost the ability to weave aspects and view the result as source code since the weave process generates directly to bytecode. Although new concerns can be woven into classes that were distributed as bytecode, this new ability is at the cost of visibility. Developers no longer have source code to analyze and, as a result, an undesired aspect weaving can be difficult to detect. The problem is exacerbated with AspectJ 1.2 where we can not only weave aspects into imported bytecode, but also reweave aspects that already exist within that bytecode. Should the imported bytecode already contain aspects,

reweaving could apply those aspects to our own code in unexpected and undesirable ways. Further, AspectJ 1.2 does not give a developer the ability to control the weave precedence and scope of aspects that are distributed for reuse.

9. ACKNOWLEDGMENTS

We would like to thank the reviewers for their rime and acknowledge the high quality of the review comments. We would also like to thank and acknowledge Günther Kniesel for his outstanding contributions and comments.

10. REFERENCES

[1] *AspectJ Project*, http://eclipse.org/aspectj/. accessed May 9th, 2004

[2] Bussard, L., Carver, L., Ernst, E., Jung, M., Robillard, M., Speck, A.. *Safe Aspect Composition*. Workshop on Aspects and Dimensions of Concern at ECOOP'2000, Cannes, France, June 2000

[3] Costanza, P., Austermann, M., Kniesel, G. *Independent Extensibility for Aspect-Oriented Systems*. Workshop on Advanced Separation of Concerns at ECOOP 2001, Budapest, Hungary, 18-22 June 2001.

[4] Gybels, K., Brichau, J. *Arranging Language Features for More Robust Pattern-based Crosscuts*. Proceedings of the 2nd international conference on Aspect-oriented software development 2003, Boston, Massachusetts

[5] Hanenberg, S., Unland, R. Parametric *Introductions*. Proceedings of the 2nd international conference on Aspect-oriented software development 2003, Boston, Massachusetts

[6] Hanenberg, S., Unland, R. *Using and Reusing Aspects in AspectJ*, Workshop on Advanced Separation of Concerns, OOPSLA, 2001.

[7] Kniesel, G., Costanza, P., Austermann, M., JMangler - *A Framework for Load-Time Transformation of Java Class Files*, 1st IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2001), p. 100-110, IEEE Computer Society Press, 2001, ISBN 0-7695-1387-5

[8] Laddad, R. *AspectJ in Action*. Manning Publications, Inc., 2003.

[9] Mikhajlov, L., Sekerinski, E. *A Study of The Fragile Base Class Problem*, In European Conference on Object-Oriented Programming, 1998.

[10] Ostermann, K., Kniesel, G. *Independent Extensibility an open challenge for AspectJ and Hyper/J*. Aspects and Dimensions of Concern workshop at ECOOP 2000, Cannes, France. March 31, 2000

[11] Störzer, M. *Analysis of AspectJ Programs*. Proceedings of 3rd German Workshop on Aspect-Oriented Software Development. February 8, 2003

[12] Szyperski, C.. *Independently Extensible Systems Software Engineering Potential and Challenges*, Proceedings of the 19th Australian Computer Science Conference. 1996

[13] *Threads, How To Use,* http://java.sun.com/docs/books /tutorial/uiswing/misc/threads.html. accessed September 28th, 2004

[14] Tourwé, T., Brichau, J., Gybels, K. *On the Existence of the AOSD-Evolution Paradox.*Workshop on Software-engineering Properties of Languages for Aspect Technologies 2003, Boston, Massachusetts.