RC24057 (W0609-070) September 19, 2006
Computer Science

IBM Research Report

Confirmed Join Points

Harold Ossher
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

Research Division
Almaden - Austin - Beijing - Haifa - India - T. J. Watson - Tokyo - Zurich

Confirmed Join Points

Position Paper
AOSD '06 Workshop on
Software Engineering Properties of Languages and Aspect Technologies (SPLAT)

Harold Ossher
IBM Thomas J. Watson Research Center
P.O. Box 704, Yorktown Heights, NY 10598, USA
{ossher}@us.ibm.com

1. Introduction

Adoption of AOSD is hindered by the fact that it breaks traditional notions of class ownership. An oft-repeated objection is that the developer responsible for a class (the class *owner*[1]) no longer has control over the details of the class, because weaving could result in separately-written aspect code affecting class semantics. Modern environment support, such as provided by AJDT [2], helps by showing the sites of weaving impact, and making it easy to navigate to the relevant aspect code, but this still does not give the class owner control.

Class ownership is a mixed blessing. We argued in the past, in the context of subject-oriented programming, that it can lead to bottlenecks, and hence that it is preferable for developers dealing with different concerns to be able to contribute separately to shared classes [7]. The ability to apply unanticipated aspects to existing code, commonly termed *obliviousness* [5], is an important element of AOSD, allowing extension and adaptation of existing software, even in binary form.

It is important to strike a balance between, and ideally to integrate, the sort of flexibility that is the hallmark of AOSD and the control needed for developers to fulfill their responsibilities to write correct code. Larochelle, Scheidt and Sullivan made essentially the same point in their work on *join point encapsulation* [10]. They motivated the need for some control over join point accessibility, and proposed a mechanism that allowed selective restriction of join points that can be advised. By default, no restrictions apply, allowing full obliviousness; restrictions can be added as appropriate (in an neatly aspect-oriented fashion).

This paper takes the position that the balance of control and flexibility is largely a social issue among the developers responsible for different parts of the software, such as class and aspects owners. Language and environment support are needed to facilitate the interactions among these developers, giving class owners appropriate control and aspect owners appropriate flexibility without undue bottlenecks.

Along these lines, we propose a specific approach, called *confirmed join points*, that offers a spectrum of control options. Essentially, aspect owners can write pointcuts (or queries) that refer to any join points, but class owners must confirm that those join points within their classes are acceptable sites for weaving according to those pointcuts. This goes beyond ensuring encapsulation, to ensuring that the class and aspect owners have consistent understandings of specific join points. Confirmations can be done globally, effectively relinquishing control, or with various degrees of specificity. What is more, environment support can greatly ease the task of confirming join points. One effect of this is allowing smooth movement from less-controlled to more-controlled usages. This approach has not yet been implemented, but we believe that it can readily be applied to a variety of AOSD languages, approaches or environments.

Section 2 briefly discusses ownership and its implications. Section 3 describes the confirmed join point proposal, and Section 4 environment support for it. Section 5 extends the proposal with a discussion of con-

[1] Throughout this paper, the term *owner* is used to refer to a *person* responsible for developing the owned software element. This usage is not to be confused with ownership of runtime data as used in alias control and the like.

firmation based on owner rather than pointcut, and Section 6 with a subjective notion of confirmation. Sections 7 and 8 cover related work and conclusions.

2. Ownership

The approach described here applies to contexts in which classes are considered to have *owners*, who are responsible for their semantics and integrity. *Advice* can be applied at join points within classes, as specified by *pointcuts* [7]. We assume that pointcuts and advice also have owners, with similar responsibilities relative to them.

This approach is neutral on the issue of static versus dynamic join points, so the term *join point* is used throughout; in the case of dynamic join points, some of the usages really refer to the corresponding join point shadows. The approach is equally applicable to symmetric composition situations, where classes and their members are composed with one another based on composition specifications, and where constructs like correspondence queries take the place of pointcuts [5], but pointcut-advice terminology is used here.

The owner of a class, to fulfill his/her responsibility for it, may wish to control the application of advice to it. Two forms of control are possible:

- Controlling the pointcuts, and hence where advice can be applied.
- Controlling the advice itself.

This paper focuses on the first, assuming that if a class owner allows advice at a particular join point, it is up to the advice owner to determine its details. It would, in fact, be reasonable and beneficial for the class owner to be able to constrain the semantics of advice by means such as contracts, including invariants to be preserved, or at least to restrict the kind of advice as suggested by Larochelle, Scheidt and Sullivan [10]. Such constraints are beyond the scope of this paper

The rest of this section briefly analyses some basic alternatives from the point of view of control over pointcuts.

2.1. Obliviousness

The pointcuts are outside the control of the class owner, probably written by an aspect owner. Advice can be applied anywhere within the class.

2.2. Exported Pointcuts

The class owner provides pointcuts as part of the exported elements of the class. Aspect owners can use only these pointcuts. Since the pointcuts are part of the class, they can refer to join points within class elements that are not visible outside.

2.3. Interface Join Points

Pointcuts are not controlled by class owners, but they may only refer to externally visible elements of a class, such as its public methods and fields. This allows aspect owners to control the pointcuts, but within limits: class owners control the elements at which advice can be applied, though not the specific join points.

2.4. Analysis

Obliviousness gives all control over pointcuts to the aspect owner, and exported pointcuts give it all to the class owner. The interface-join-point approach is a hybrid, and an appealing one, because it allows aspect authors to craft pointcuts themselves, but in terms of elements that are publicly visible. However, it falls short in two opposite ways:

- It does not give enough freedom to aspect owners in contexts where they are closely associated with class owners, in which case they might reasonably need join points in non-visible elements.
- It does not give enough control to class owners, because all join points in externally visible elements are automatically available.

3. Confirmed Join Points

The proposal in this paper is to use the time-honored programming-language-design approach of adding redundancy to achieve checking or control, in a way that is not onerous. It is described and exemplified here in general terms, without detailed reference to any specific language, though AspectJ syntax is used for pointcuts [7]; it is easy to see how the concepts could be realized in a variety of languages.

The approach makes a fundamental underlying assumption: *pointcuts serve as an interface between class and aspect owners*. A pointcut captures an abstraction, such as “all modifier methods” or “all calls to the XYZ service,” and it is important that class owners and aspect owners understand that abstraction in consistent ways. Pointcut names thus have the

status, and are assumed to have the stability, of public names in interfaces.

The essential elements of the approach are:

- Pointcuts need not be directly under the control of class owners. Class owners can include exported pointcuts in their classes, and this is certainly a good way to enhance the abstractions they are providing. But named pointcuts can also be written within aspects by aspect owners or, preferably, in separate modules that serve as interfaces between classes and aspects. Pointcut owners can thus be class owners, aspect owners, or neither.
- Class owners must confirm that the join points within their classes designated by specific, named pointcuts are acceptable. This is taken as given for pointcuts defined within the class.
- Environment support can help to manage confirmations.

3.1. Example

Consider a pointcut defined in module (e.g., class or aspect) *M* and designating calls to method *X.foo():*

```
module M {
   pointcut p(): call(* X.foo());
}
```

Now consider class *C* containing some calls to *X.foo():*

```
class C {
   void m1() { X.foo(); X.foo(); }
   void m2() { X.foo(); }
}
```

If confirmed pointcuts were in use, perhaps through a compiler option, this would result in three errors, one for each call. The owner of *C* can confirm *p* for the entire class:

```
class C confirms M.p{ … }
```

for a method:

```
… void m1() confirms M.p { … } …
```

or for a block:

```
… void m1() {
      confirm M.p {
         X.foo();
      }
      X.foo(); } …
```

3.2. Confirmation

The example illustrated the ability to confirm pointcuts at the class, method and statement level. Expression-level confirmation is also possible with suitable syntax, though likely to be somewhat clumsy.

To confirm all pointcuts, supporting obliviousness, "**confirm** *" can be used. To confirm all pointcuts within externally-visible elements, supporting interface pointcuts, "**confirm public**" can be used. To confirm all pointcuts declared in some module *M*, "**confirm** *M.**" can be used. To handle sets of pointcuts, confirmation of a compound pointcut is construed to confirm its subsidiary named pointcuts as well.

The semantics of confirmation are straightforward. Each named pointcut is evaluated to produce a set of join points (or join point shadows in the case of dynamic join points). At each such join point, a check is made to determine whether that join point is statically nested within a confirmation of the relevant pointcut, or of a pointcut that includes the relevant pointcut. If not, an *confirmation error* is reported.

Confirmation errors are, essentially, a communication vehicle from pointcut owners to class owners. Pointcut owners introduce or modify pointcuts, which then match new join points. The errors highlight those join points so that class owners can examine and, ideally, confirm them. Lack of errors indicates agreement.

3.3. Denial

Agreement might not be so easy to attain, however. The class owner might find, upon examination, that a particular join point should *not*, in fact, match a pointcut. It might be determined that the join point was introduced in error, such as by coding a call to the wrong method, but it is more likely that the pointcut will need to be modified. One possibility would be to add a clause to the pointcut that specifically excludes the join point, or perhaps its containing method or class. For example:

```
module M {
   pointcut p():
      call(* X.foo()) &&
      ! withincode(void C.m2());
}
```

It can be argued that this type of change to a pointcut can reasonably be made by a class owner alone. When doing confirmations, the class owner is assumed to understand the abstraction the pointcut captures, and is here asserting that the specific join point in question does not fall under that abstraction. However, it is likely to be safer and more productive to negotiate with the pointcut owner, to determine whether some broader modification would be more appropriate.

Explicit pointcut *denial* facilitates this communication from class owner back to pointcut owner. For example:

```
class C {
   void m1() confirms M.p { … }
   void m2() denies M.p { … }
}
```

Here the explicit "denies" clause indicates that the class owner has determined that join points within *C.m2*() are not appropriate matches for *M.p*. This is reported as a *denial error* on the pointcut, alerting the pointcut owner to the need to modify it. This might, of course, lead to discussion between the owners.

Denial can be used to override coarse-grained confirmation, most appropriate where there are a few localized exceptions to a general rule. For example:

```
class C confirms M.p {
   void m1(){ … }
   void m2() denies M.p { … }
   ...
}
```

Denial is stronger than lack of confirmation, and has the opposite effect with respect to communication: it is an explicit statement that the join point *should not* match the pointcut, and that the pointcut owner has responsibility for fixing the pointcut; lack of confirmation indicates that the status of the join point has not been determined, and that the class owner has responsibility for doing so.

A number of embellishments or variations are possible. For example, a pointcut declared as **weak** might automatically exclude any denied join points without reporting errors; this is somewhat dangerous, however, as it makes it easy for class owners to deny lazily without really considering each case carefully. A full design based on the concepts presented in this paper would need to consider the alternatives in the context of the specific AOSD language or approach in which they are being embedded.

4. Environment Support

A software development environment supporting confirmed join points would highlight confirmation and denial errors, and allow developers to navigate to and examine the sites of such errors. Ideally, it would provide "quick fixes," as in Eclipse, to enable developers to make standard changes easily. For confirmation errors, quick fixes would be available to add confirmations or denials, offering choices of granularity (e.g., statement, method, or class). It would also be convenient to allow multiple, or all, confirmation errors to be selected and confirmed at once. For denial errors, quick fixes would allow extension of the pointcut to exclude the denied join points.

This kind of support, along with the ability to turn confirmation checking on or off through an option or preference setting, makes it particularly easy to transition from uncontrolled situations to controlled ones. Initially, pointcuts could be written and used without confirmation checking, as they are today. At any point in time, confirmation checking could be turned on, and would show where confirmations are needed. This is exactly analogous to browsing the effects of pointcuts using an environment like AJDT [1], at a stage in the project when one wants to check them manually. It has the advantage, however, that confirmations and denials can be recorded easily, using the quick fixes. Of course, it is the class owners who should do this examination and apply the confirmation error fixes. The fact that it is their responsibility is consistent with the fact that the confirmation insertions are actual modifications to their classes. Similarly, it is pointcut owners' responsibility to deal with denial errors. Once this has been done, in the absence of newly-introduced errors, class owners can be confident that advice is not being attached at unexpected places in their classes, and pointcut owners can be confident that class owners agree about where advice should be applied.

Further confirmation errors can arise either as a result of changes to a class, introducing new join points that match an existing pointcut, or due to changes in pointcuts. Both are useful, alerting class owners to the need to examine the new join points and to confirm or deny them. More sophisticated support for evolution is provided by subjective confirmations, described in Section 6.

The environment might also highlight confirmation regions that contain no matching join points, especially if they had contained matches before the current set of changes.

5. Owner-Based Confirmation

In the approach so far presented, confirmation constructs refer to specific pointcut declarations (so do denial constructs, but denial will no longer be mentioned explicitly, for convenience). In many contexts it is actually more important to exercise control by owner, or owner's team or organization, rather than by pointcut. This supports the ability to work with trusted partners, without the overhead of enumerating the pointcuts they own. The assumption is that they will

cope with changes made within classes, and understand the essential class semantics and not violate it.

Conceptually, it is easy to extend confirmation pointcuts to support confirmation by owner, with constructs like

```
    … confirms o …
or
    … confirms * owned by o …
```

where *o* designates an owner, or a set of owners such as a team or organization. The second form allows finer control by both pointcut and owner if desired (and avoids possible confusion between pointcut names and owner designations).

The difficulty arises in certifying ownership. This is usually an extra-lingual issue, requiring support from the development environment and operating system. Such support is common, however, in environments that include software configuration management; their versioned repositories keep track of who owns and changes artifacts, and some even support team structures. Confirmation checks in the context of such an environment can use this information

6. Subjective Confirmation

When fine-grained confirmations are used, relevant changes to either classes or pointcuts will usually result in confirmation errors, leading the class owners to examine the changes as described in Section 4. When coarse-grained confirmations are used, however, such as confirming a pointcut or owner for an entire class, even radical changes might not result in confirmation errors. In the case of close collaboration and trust between the class and aspect owners, this might be acceptable, even desired. But in many cases it would be better for the changes to be brought to the notice of class owners. Subjective confirmations allow this to be done without the inconvenience of fine-grained confirmations.

A *subjective confirmation* not only names a pointcut, but includes a subjective definition of that pointcut from the point of view of the confirmer. For example:

```
class C
    confirms M.p = call(* X.foo())
{ … }
```

This states explicitly that *M.p* may refer to calls to *X.foo()* within class *C*. It records the class owner's assertion that s/he understands the abstraction that *M.p* captures, and that the definition accurately specifies the join points in this class that fall under that abstraction. If new calls to *X.foo()* are added to the class, no confirmation errors will result. However, if *M.p* is changed to

```
module M {
    pointcut p(): call(* X.foo())
                || call(* Y.bar());
}
```

then any calls to *Y.bar()* in *C* will yield confirmation errors.

The confirmation might also be broader than the referenced pointcut. For example, with *M.p* as originally defined, consider the confirmation:

```
class C
  confirms M.p = call(* X.foo(..))
{ … }
```

Now C might contain calls, such as to *X.foo(int),* that match the confirmation but not the referenced pointcut. A strong interpretation of subjectivity would require that this yield a confirmation error also: the abstraction captured by the pointcut includes more join points in the class owner's view than in the aspect owner's view, a clear inconsistency that must be resolved. However, if the objective is merely to restrict access rather than to ensure agreement about the meaning of pointcuts, then this situation can be considered acceptable.

The semantics of weak subjective confirmations are thus that any join point in the scope of the confirmation that is matched by the referenced pointcut but is not matched by the confirmation yields a confirmation error. For strong subjective confirmations, any join point in the scope of the confirmation that matches the confirmation but not the referenced pointcut also yields a confirmation error. The class and pointcut can then evolve independently, with confirmation errors arising precisely then the class and pointcut owners' views diverge.

7. Related Work

As noted in the introduction, Larochelle, Scheidt and Sullivan's work on join point encapsulation [10] was motivated by similar considerations. Their mechanism involves applying special *restriction advice* to join points that are to be restricted, using standard pointcuts to characterize the join points. The usage they describe where restriction advice is placed in inner aspects within classes provides class-owner control. The alternative usage, where the restriction advice is placed in a top-level aspect describing what restrictions apply to the system as a whole, allows overall control by an architect or designer, someone filling the *composition*

designer role mentioned in Section 8. The key difference in our approaches is that restriction advice focuses on specifying join points that are not available for advice binding, whereas ours, especially subjective confirmation, focuses on ensuring a consistent understanding of the abstractions captured by pointcuts.

Considerable work has been done exploring the implications of aspects for modularity [1][3][4][9][12][13], which is closely related to issues of ownership and control. Herrmann has explored the combination of flexibility and strictness in the context of the Object Teams paradigm [7]. Detailed analysis of our approach relative to these has not yet been done. However, we believe that the focus on facilitating communication aimed at achieving and maintaining a consistent understanding of the abstractions captured by pointcuts is novel.

8. Conclusion

This position paper proposed a simple practical approach to ameliorating the tension between control by class owners and flexibility for aspect owners. Pointcuts can be written and modified at will by pointcut owners (who might also be aspect owners or class owners). However, class owners must confirm explicitly that join points matching those pointcuts are acceptable, and those confirmations remain visible in the class code. Environment support and coarse-grained confirmation options make the confirmation process lightweight. To allow software processes involving multiple levels of trust among multiple collaborating groups, confirmation based on ownership is also supported.

This paper discussed the roles of class and aspect owners. It is also worth considering another role, *composition architect*, responsible for assembling pieces (classes, aspects, components, composition filters, hyperslices, etc.) and specifying how they should be composed, and perhaps for coordinating the efforts of the owners of the separate pieces. Maintaining pointcuts might therefore be one of his/her duties. The composition architect is assumed to have a broader view than piece owners, and probably greater or even overriding authority. Still, it would be valuable to include in the development process the ability for piece owners to confirm or deny join points used for composition. Examining the details in this context remains an area for future work.

9. Acknowledgements

Thanks to Stan Sutton for helpful comments, especially raising the issue of the composition-architect role. Thanks to Mark Wegman for helpful comments, especially raising the issue of owner-based confirmation. Thanks to the reviewers for their very helpful feedback.

10. References

[1] J. Aldrich. Open Modules: Reconciling Extensibility and Information Hiding. In AOSD workshop on Software Engineering Properties of Languages for Aspect Technologies (SPLAT '04), March 2004.

[2] AJDT: AspectJ Development Tools Eclipse Technology Project. http://www.eclipse.org/ajdt.

[3] C. Clifton and G. Leavens, "Obliviousness, Modular Reasoning, and the Behavioral Subtyping Analogy." Technical Report 03-01a, Dept. of Computer Science, Iowa State University.

[4] C. Clifton and G. Leavens, "Observers and assistants: A proposal for modular aspect-oriented reasoning." In Proc. FOAL Workshop, 2002.

[5] R. Filman and D. Friedman. "Aspect-Oriented Programming is Quantification and Obliviousness." In R. Filman et. al. (Eds.), Aspect-Oriented Software Development, Addison-Wesley, 2005.

[6] W. Harrison, H. Ossher and P. Tarr. "Concepts for Describing Composition of Software Artifacts." IBM Research Report RC23345, IBM Thomas J. Watson Research Center, Yorktown Heights, NY, September 2004.

[7] S. Herrmann. "Sustainable architectures by combining flexibility and strictness in Ojbect Teams." In *IEE Proceedings - Software Engineering* 151(2), Special Issue on Unanticipated Software Evolution, April 2004.

[8] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, Jeffrey Palm and William G. Griswold. "An Overview of AspectJ." Proc. 15th European Conference on Object-Oriented Programming, 327-353 (2001).

[9] G. Kiczales and M. Mezini, "Aspect-Oriented Programming and Modular Reasoning." In Proceedings of ICSE'05.

[10] D. Larochelle, K. Scheidt and K. Sullivan, "Join Point Encapsulation," In Proceedings SPLAT Workshop, 2003.

[11] H. Ossher, W. Harrison, F. Budinsky, and I. Simmonds, "Subject-oriented programming: Supporting decentralized development of objects." In *Proceedings of the 7th IBM Conference on Object-Oriented Technology*, IBM, July 1994.

[12] Y. Song, "Join Point Interfaces: Information Hiding Modularity for Aspect-Oriented Program Design." Masters project, University of Virginia.

[13] K. Sullivan, W. Griswold, Y. Song, Y. Cai, M. Shonle, N. Tewari, H. Rajan, "Information Hiding Interfaces for Aspect-Oriented Design." In Proceedings of ESEC/FSE '05.