Join Point Interfaces for Modular Reasoning in Aspect-Oriented Programs

Milton Inostroza Éric Tanter
PLEIAD Laboratory
Computer Science Department (DCC)
University of Chile – Santiago, Chile
{minostro,etanter}@dcc.uchile.cl

Eric Bodden
Software Technology Group
Center for Advanced Security Research Darmstadt (CASED)
Technische Universität Darmstadt - Germany
bodden@acm.org

ABSTRACT

While aspect-oriented programming supports the modular definition of crosscutting concerns, most approaches to aspect-oriented programming fail to improve, or even preserve, modular *reasoning*. The main problem is that aspects usually carry, through their pointcuts, explicit references to the base code. These dependencies make programs fragile. Changes in the base code can unwittingly break a pointcut definition, rendering the aspect ineffective or causing spurious matches. Conversely, a change in a pointcut definition may cause parts of the base code to be advised without notice. Therefore separate development of aspect-oriented programs is largely compromised, which in turns seriously hinders the adoption of aspect-oriented programming by practitioners.

We propose to separate base code and aspects using Join Point Interfaces, which are contracts between aspects and base code. Base code can define pointcuts that expose selected join points through a Join Point Interface. Conversely, an aspect can offer to advise join points that provide a given Join Point Interface. Crucially, however, aspect themselves cannot contain pointcuts, and hence cannot refer to base code elements. In addition, because a given join point can provide several Join Point Interfaces, and Join Point Interfaces can be organized in a subtype hierarchy, our approach supports join point polymorphism. We describe a novel advice dispatch mechanism that offers a flexible and type-safe approach to aspect reuse.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs

General Terms

Design, Languages

Keywords

Aspect-oriented programming, modularity

1. JOIN POINT INTERFACES

Inevitably, aspect-oriented programming [3] aids modularizing crosscutting code: it helps programmers to put together code that belongs together. So far, however, most approaches to aspect-oriented programming have failed to improve, or even preserve, modular reasoning. Such modular reasoning is easily established in traditional procedural languages. In such languages, programmers can reason about one procedure at a time and in isolation. The procedure's signature establishes a strong contract with the contexts in which the procedure may be used. Object-oriented programming already gives up modular reasoning to some extent. Object-oriented programs carry virtual method calls. For such calls, at least in statically typed languages, the signature of the call is known at the call site, and hence the usage contract for the called method is known as well. It is unknown, however, which concrete implementation the virtual method call will eventually be dispatched to in the running program.

1.1 Aspects and modular reasoning

With aspect-oriented programming, modular reasoning becomes even harder, as aspect-oriented programs adds a crucial feature: implicit invocation with implicit announcement (IIIA) [7]. Through IIIA, an aspect can become active at many different program points (called join points) without any explicit call to the aspect being present at these points. This is problematic because maintainers of the base code may be unaware of the program point being advised, and hence may refactor this point or change the point otherwise, unwittingly breaking the connection to the advising aspect. There has been several attempts to discuss and enhance the possibilities for modular reasoning in presence of pointcuts and advice, but they all eventually fall short in supporting full separate development with static and modular typechecking.

Running Example

As a running example, we will consider an e-commerce system with a set of discount rules. (Deliberately, we keep the example similar to a motivating example by Steimann et al. [7].) In the initial system, a customer can check out a product by either buying or renting the product. On the customer's birthday, the customer will be given a 5% discount when checking out a product. We will be adding further rules later.

```
1  class ShoppingSession {
2    ShoppingCart sc = new ShoppingCart();
3    Invoice inv = new Invoice();
4
5    void checkOut(Item item, float price,
6        int amount, Customer cus){
7      sc.add(item, amount);
8      inv.add(item, amount, cus);
9    }
10 }
11
12 aspect Discount {
13   pointcut checkingOut(Item item, float price,
14       int amount, Customer cus):
15     execution(* Session.checkOut(..))
16     && args(item, price, amount, cus);
17
18   void around(Item item, float price, int amt,
         Customer cus):
19     checkingOut(item, amt, cus) {
20     int factor = cus.hasBirthday()? 0.95 : 1;
21     proceed(item, price*factor, amt, cus);
22   }
23 }
```

Listing 1: Shopping session with discount aspect

Listing 1 shows an implementation of the example in plain AspectJ [5]. The around advice in lines 18–22 applies the discount by reducing the item price to 95% of the original price when proceeding on the customer birthday. Note how brittle the AspectJ implementation is with respect to changes in the base code. Most changes to the signature of the `checkOut` method, such as renaming the method or modifying its parameter declarations, will cause the `BirthdayDiscount` aspect to lose its effect. The root cause of this problem is that the aspect, through its pointcut definition in lines 13–16, makes explicit references to named entities of the base code—here to the `checkOut` method.

1.2 Join Point Interfaces

In this paper, we propose to establish an additional layer of abstraction between base code and aspects, through a novel mechanism called Join Point Interfaces. Our goal is to allow for complete modular reasoning on both the side of the aspect programmer and the side of the base-code programmer. In particular, our system will allow programmers to catch all possible typing errors at the time the individual code fragments (aspect, base code) are compiled. There have been other approaches to decoupling aspects from base code but those approaches are not able to capture many typing errors in a modular manner at compile time. Rather, they only detect errors at weave-time, *i.e.* at the time the aspect is actually composed with the base-code system. This is because the layers that they introduce are "transparent"; they separate aspects from the base code only to some extent. Crosscutting interfaces (XPIs) [4] are based on pure AspectJ, which provides no language-based mechanisms for the separation that we envision. Steimann et al.'s work on join point types for IIIA [7] does introduce specialized syntax but this syntax is not rich enough to be able to allow for modular reasoning and separate compilation. In Section 3 we will elaborate further on these approaches.

Listing 2 shows how Join Point Interfaces improve on the shopping cart example. Line 2 declares the join point interface (`jpi`) `checkingOut`. Syntactically, a Join Point Interface declaration is identical to a method signature: it has

```
1  jpi void checkingOut(Item item, float price,
2                       int amount, Customer cus);
3
4  class ShoppingSession {
5    exhibit void checkingOut(Item i, float price,
6        int amount, Customer c):
7      execution(* checkOut(..))
8      && args(i, price, amount, c);
9
10   ...
11 }
12
13 aspect Discount {
14   void around checkingOut(Item item, float price,
15       int amt, Customer cus){
16     int factor = cus.hasBirthday()? 0.95 : 1;
17     proceed(item, price*factor, amt, cus);
18   }
19 }
```

Listing 2: Introducing a Join Point Interface

a return type (here `void`), a name, a formal-parameter list and an optional `throws`-clause. (This choice is for a good reason. In a recent piece of work, Bodden showed that one can avoid many semantic pitfalls by regarding join points as typed closures [2].)

The base-code class `ShoppingSession` is enhanced to declare that it exhibits join points of type `checkingOut`. The `exhibit`-clause binds the Join Point Interface to concrete join points, using a regular AspectJ pointcut.

Crucially, however, through the usage of Join Point Interfaces, the aspect is completely liberated of pointcut definitions. Note that in line 14 the aspect refers directly to the Join Point Interface, obviating the need to explicitly refer to base code elements. As the example shows, Join Point Interfaces allows for the modularization of crosscutting concerns while at the same time restoring modular reasoning. The base-code programmer can maintain pointcut definitions in sync with the base code that these pointcut definitions refer to. The programmer is also immediately aware of the join points exposed to aspects. On the other hand, the aspect becomes free of any textual references to base-code elements, making the aspect reusable in other contexts.

1.3 Static and modular type checking

Join Point Interfaces support modular reasoning through a fully static and modular type system. For instance, the type system ensures that the header of the around advice at line 14 is compatible with the definition of the `checkingOut` Join Point Interface. As described in earlier work [2], this needs to involve invariant parameter and return types. Further, if an advice a advises a Join Point Interface j which declares to throw checked exceptions of type e then a must either catch exceptions of this type or declare to propagate them. Similarly, all join point shadows (i.e., code locations) matched by any pointcut tagged with j must handle or propagate these exceptions. These typing rules guarantee that the implicit invocation of a piece of advice at a given join point always comply with the Join Point Interface.

We envision Join Point Interfaces to be used such that the Join Point Interface definition in line 2 would be available to both the base-code programmer and the aspect programmer. This allows for separate compilation of aspects and base code. In this case, our type system gives the strong guarantee that there can be no weave-time errors: all typ-

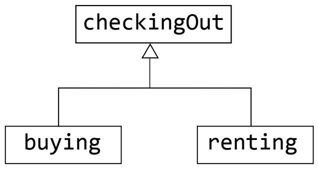

Figure 1: Inheritance between Join Point Interfaces

ing errors will be caught at the time the aspect or base code are (potentially independently!) defined, and no further errors can occur when aspects and base code are composed. To the best of our knowledge, our system is the first to provide such guarantees. For instance, while Steimann et al.'s types for IIIA [7] also offers the benefits described in Section 1.2, their approach is unable to provide modular reasoning and separate compilation because their join point types are only partial join point specifications. In their approach some conditions, such as return type compatibility, must be checked at weave time, which is undesirable in a collaborative working environment and also hinders aspect reuse.

1.4 Polymorphic join points

Join Point Interfaces give *types* to join points. In the same way that object interfaces in languages like Java support a flexible form of subtype polymorphism, Join Point Interfaces enable polymorphic join points. A join point can be seen as providing multiple Join Point Interfaces, and advice dispatch at that join point can take advantage of this polymorphism.

Which Join Point Interfaces does a join point provide? This is the role of pointcuts. As we have seen, a class defines the pointcuts that expose certain join points in its execution, following a given Join Point Interface. For instance, in Listing 2, class `ShoppingSession` defines a pointcut that gives the type `checkingOut` to all join points that are executions of the `checkOut` method. Because a join point can be matched by several pointcuts, a join point can have multiple types. For instance, an execution of `checkOut` can be seen as a `checkingOut`, and could be seen additionally as a `loggableEvent` (a Join Point Interface whose definition is left to the imagination of the reader). In addition, as in Java, Join Point Interfaces support subtyping. We illustrate join point subtyping using two subtypes of `checkingOut`, `buying` and `renting` (Figure 1). The rest of this section illustrates their use in our running example.

Consider that we introduce the following business rule: the customer gets a 15% discount when *buying* at least 10 products of the same kind; this promotion is not compatible with the birthday discount.

Listing 3 shows how a programmer could implement this additional rule using the subtyping relationship on Join Point Interfaces. First, we declare the Join Point Interface `buying` as a subtype of `checkingOut`. The semantics of this subtyping relationship implies that `buying` represents a subset of the join points that `checkingOut` represents.

The aspect `Discount` now declares two pieces of advice. The first one is the same as in the previous example. It applies to all `checkingOut` join points, in particular also to those `checkingOut` join points that are not `buying` join points. points that *are* of type `buying` and hence *also* of type `checkingOut`? To the best of our knowledge, our approach is the first to provide a clear and natural mechanism to resolve this situation. According to the semantics of Join

```
jpi void buying(Item item, float price,
                int amount, Customer cus)
  extends checkingOut;

aspect Discount {
  void around checkingOut(Item item, float price,
      int amt, Customer cus) {
    int factor = cus.hasBirthday()? 0.95 : 1;
    proceed(item, price*factor, amt, cus);
  }
  void around buying(Item item, float price,
                     int amt, Customer cus) {
    int factor = (amt > 10) ? 0.85 : 1;
    proceed(item, price*factor, amt, cus);
  }
}
```

Listing 3: Advice overriding

Point Interfaces, the advice with the *most-specific* signature executes. Hence, in this example, a join point of type `buying` is advised *only* by the `buying` advice, since it is the most specific[1].

This semantics of advice dispatch with Join Point Interfaces is natural because it directly follows how overloading is resolved in languages with multiple dispatch, like the Common Lisp Object System (CLOS) [6]: the runtime type of the argument (here, the type of the join point) determines which method is executed (here, which piece of advice). The analogy makes sense because with implicit invocation, there is by definition no explicit receiver, just like with generic functions in CLOS. It is the runtime type of the arguments that drives the dispatch process.

Reusing behavior. In a previous version of our language design, we pushed the analogy between advice dispatch and multiple dispatch a step further, by having applicable pieces of advice ordered in terms of specificity (like applicable methods in CLOS), and providing a `nextadvice` primitive (the equivalent of `call-next-method` in CLOS) to call the next most-specific advice in the ordering). While this allows for direct reuse of advices, the interaction of `nextadvice` and `proceed` induces too much complexity. Therefore, we give up on `nextadvice`; reuse among advices has to be mediated through aspect methods, as illustrated below.

Consider another extension of our running example, with a rule stating that when renting at least 10 items of the same kind there will be a 10% discount; but in this case the birthday discount *does* apply. Listing 4 shows the code for this extended example. First, we declare the new join point subtype `renting`. Then, because we want to reuse the birthday discount logic, we extract a separate method `birthdayFactor` in the aspect. The advice for `checkingOut` join points is modified accordingly, to use this method. Now, the advice for `renting` can also reuse the birthday discount logic by invoking the `birthdayFactor` method, in order to compute the final discount rate.

2. NOVELTY

To the best of our knowledge, our approach is the first to provide true modular reasoning in the presence of pointcuts

[1]As usual with `proceed`, if there are other pieces of advice advising `buying` then those execute before the original join point, with precedence defined as in AspectJ.

```
jpi void renting(Item item, float price,
                 int amount, Customer cus)
    extends checkingOut;

aspect Discount {
  int birthdayFactor(Customer cus){
    return cus.hasBirthday()? 0.95 : 1;
  }
  void around checkingOut(...) {
    proceed(item, price*birthdayFactor(cus), amt,
        cus);
  }
  void around buying(...) { /*as before*/ }
  void around renting(Item item, float price,
                      int amt, Customer cus) {
    if(amt>10)
      int factor = (amt > 10) ? 0.9 : 1;
    proceed(item, price*factor*birthdayFactor(cus)
        , amt, cus);
  }
}
```

Listing 4: Advice overriding—reusing behavior

and advice: Join Point Interfaces are the first mechanism to provide separate compilation for aspects with the guaranteed absence of weave-time errors. Further, our approach is the first to properly support join point polymorphism, enable flexible advice reuse.

3. RELATED WORK

There are have been several approaches to tackle the issue of modular reasoning in the presence of advice, including Open Modules [1], XPIs [4] and join point types for Implicit Invocation with Implicit Announcement (IIIA) [7]. The proposal of Steimann et al. is the most recent and as such includes a detailed discussion of all previous approaches to aspects and modular reasoning [7].

Similar to our proposal, Steimann et al. propose to introduce typing annotations for join points, called join point types, to facilitate modular reasoning. They also provide a subtyping hierarchy on join points, to facilitate extensibility and reuse. Our proposal builds upon IIIA, but with significant and crucial differences. IIIA falls short on several levels, most importantly, static and modular type checking, and join point polymorphism.

IIIA defines join point types using a struct-like syntax instead of a method-signature-like syntax as we do. Method signatures have the advantage that they can capture crucial elements like return types and declared exceptions, that the join point types of IIIA cannot capture. Join Point Interfaces allow us to support strong static guarantees and separate compilation. Both are impossible using IIIA, which relies on weave-time error reporting.

The IIIA approach falls short when it comes to resolving advice dispatch in presence of join point subtyping. To briefly illustrate why, consider again the join point hierarchy shown in Figure 1. Let us assume that we wish to dispatch pieces of advice for a join point that is *both* of type `buying` and of typing `renting`, and assume that there exists a unique piece of advice defined, for `checkingOut`. In Steimann et al.'s proposal to IIIA, the implementation will generate *two separate* join point instances, one of type `buying` and one of type `renting`. Because both of those instances are subtypes of `checkingOut`, they both dispatch to the `checkingOut` advice: the `checkingOut` advice executes twice!

Concretely, this means that IIIA does not support join point polymorphism; rather, it emits several join points for the same execution point. This semantics appears more than surprising and hardly useful under any circumstances. Our approach, on the other hand, supports true join point polymorphism. A single join point can have multiple types, and the well-known semantics used in multiple dispatch languages is used. In addition, this brings us the possibility of introducing `nextadvice`, to reuse and specialize pieces of advice.

4. EXPECTED FEEDBACK

First of all, at the most general level, we are interested in getting feedback on the importance of separate compilation and static modular type checking for the adoption of aspects. As language designers, we feel this is a great step forward, that even the latest development in aspects and modularity do not address properly [7]. We are really interested in discussing this with the software engineering community at large to see if this impression is shared.

At a more specific level, we are looking forward to feedback on the language design for Join Point Interfaces. Polymorphic join points and dynamic advice dispatch seem to enable a wider range of aspect designs for reuse than the existing proposals. Finally, as mentioned in Section 1.4, we gave up on the possibility to have `nextadvice` in order to trigger less-specific advice, as in multiple dispatch. The reason is that the kind of reuse offered by `nextadvice` does not seem to compose well with `proceed`, especially considering the fact that in most aspect languages, it is possible to invoke `proceed` multiple times. Discussion on these topics would be of particular interest for the further development of Join Point Interfaces.

5. REFERENCES

[1] Jonathan Aldrich. Open modules: Modular reasoning about advice. In Andrew P. Black, editor, *Proceedings of the 19th European Conference on Object-Oriented Programming (ECOOP 2005)*, number 3586 in LNCS, pages 144–168, Glasgow, UK, July 2005. Springer-Verlag.

[2] Eric Bodden. Closure joinpoints: block joinpoints without surprises. In *Proceedings of the tenth international conference on Aspect-oriented software development*, AOSD '11, pages 117–128, New York, NY, USA, 2011. ACM.

[3] Robert E. Filman, Tzilla Elrad, Siobhán Clarke, and Mehmet Akşit, editors. *Aspect-Oriented Software Development.* Addison-Wesley, Boston, 2005.

[4] William G. Griswold, Kevin Sullivan, Yuanyuan Song, Macneil Shonle, Nishit Tewari, Yuanfang Cai, and Hridesh Rajan. Modular software design with crosscutting interfaces. *IEEE Softw.*, 23:51–60, January 2006.

[5] Ramnivas Laddad. *AspectJ in Action: Practical Aspect-Oriented Programming.* Manning Press, 2003.

[6] Andreas Paepcke, editor. *Object-Oriented Programming: The CLOS Perspective.* MIT Press, 1993.

[7] Friedrich Steimann, Thomas Pawlitzki, Sven Apel, and Christian Kästner. Types and modularity for implicit invocation with implicit announcement. *TOSEM*, 20(1):1–43, 2010.