A Brief Tour of Join Point Interfaces

Eric Bodden
Secure Software Engineering
EC SPRIDE, TU Darmstadt, Germany
bodden@acm.org

Éric Tanter Milton Inostroza
PLEIAD Lab
Computer Science Dept (DCC)
University of Chile
{etanter,minostro}@dcc.uchile.cl

ABSTRACT

In standard AspectJ, aspects and base code are often insufficiently decoupled, as aspects hold pointcuts, which can contain explicit textual references to base code. This hinders aspect evolution and reuse, and may hinder reasoning about aspects on the base-code side.

In this demo we present join point interfaces as an extension to the aspect-oriented programming language AspectJ. Opposed to AspectJ, with join point interfaces aspects and base code communicate only through a shared interface abstraction. Aspects themselves go without pointcuts and only reference the interface. Pointcuts are typically defined on the base-code side, or not at all, as join point interfaces also support pure explicit invocation as known from publish-subscribe systems. As a result, users obtain a language which decouples aspects from base code using a modular type-checking algorithm, and which they can use to adopt aspects gradually as they desire.

One major undertaking in the design of join point interfaces was to make the language as flexible to use as standard AspectJ, while nevertheless providing interfaces supported by strong type checks that can completely avoid type errors at composition time. In this demo we will discuss this inherent trade-off, we will present JPIs as an extension to the AspectBench Compiler, and will show how the language eases the maintenance of existing AspectJ applications.

Categories and Subject Descriptors: D.3.3 [Programming Languages]: Language Constructs

General Terms: Design, Languages

Keywords: Modularity, maintainability, aspect-oriented programming, event-driven programming

1. OUTLINE

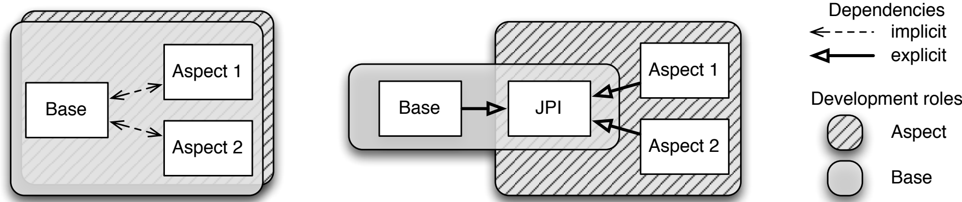
This demo paper gives a brief overview of Join Point Interfaces (JPIs), based on elements from the complete description of JPIs [2].

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Copyright 20XX ACM X-XXXXX-XX-X/XX/XX ...$10.00.

As we show in Figure 1a, aspects with pointcuts and advice as in AspectJ cause implicit dependencies between aspects and base code. This is because an aspect contains direct textual references to the base code via its pointcuts. These implicit dependencies, denoted by the dashed arrows in the figure, make programs fragile, hinder aspect evolution and reuse, and compromise separate development. Changes in the base code can unwittingly render aspects ineffective or cause spurious advice applications. Conversely, a change in a pointcut definition may cause parts of the base program to be advised without notice, breaking some implicit assumptions. This problem is known as the *fragile pointcut problem* [4, 7]. Effectively this means that both developers that maintain aspects and developers that maintain base code must usually have some global knowledge about the software system, e.g. knowing the aspects that are defined in order to determine if they affect a given module. The fact that independent development is compromised this way is particularly worrying considering that programming aspects requires a high level of expertise, and is hence likely to be done by specialized programmers, leading to different development roles. Therefore, to be widely adopted, AOP is in need of mechanisms to support separate development in a well-defined manner.

In this work, we combine important ideas of existing earlier work [3, 5, 6] with novel insights obtained through our own research, to arrive at a language design that enables safe modular type checking for aspects. Our solution, called *join point interfaces* (JPIs), consists of type-based contracts that decouple aspects from advised code (Figure 1b). JPIs support a programming methodology where aspects only specify the types of join points they advise, but do not comprise any pointcuts. It is the responsibility of the programmer maintaining the advised code to specify which join points are exposed, and of which type. Quantification is now local, restricted to a given class. In addition to implicit announcement through pointcuts, JPIs are integrated with Closure Join Points [1], a mechanism for explicit event announcement. With closure join points the dependencies become explicit, and therefore simplify reasoning.

Figure 1b gives an overview of our design. The figure only contains solid arrows: both aspects and base code explicitly refer to JPIs, without implicit dependencies between them. This supports a form of modular reasoning: since JPIs serve as a type safe contract, at least as far as the type system is concerned, the base-code developer can reason about base code by just inspecting base code and the JPIs that it explicitly references; similarly, the aspect programmer can reason



(a) Pointcut/advice (b) Join point interfaces

Figure 1: Dependencies with traditional pointcut/advice AOP (a) and with join point interfaces (b).

locally about aspects and the referenced JPIs, without having to worry about base code.

Our language design integrates join point interfaces with a number of language features that, in combination, allow developers to introduce aspect-oriented in a gradual manner, while at the same time supporting separate development of aspects and base code through strong type checks.

2. A TOUR OF JOIN POINT INTERFACES

In the following we will explain in more detail the different constructs of our AspectJ implementation of join point interfaces.

The interface description.

Since in AspectJ, join points resemble control flows, syntactically, JPIs resemble method signatures. A typical JPI description look as follows:

```
jpi void CheckingOut(double price, Customer cus);
```

Implicit and explicit announcement.

Join point interfaces can be used both with implicit announcement through pointcuts, and with explicit announcement, using closure join points [1]. The following code shows how a class announces join points of type `CheckingOut` implicitly at all call to `checkOut` lexically contained with that class:

```
class ShoppingSession {
  exhibits void CheckingOut(double price, Customer c):
    execution(* checkOut(..)) && args(*, price, *, c);
        ...
}
```

Programmers will typically use such `exhibits` clauses in connection with pointcuts if they desire to expose a number of different join points within the same class. As an alternative, closure join points can mark explicitly regions of code to be exposed to aspects:

In Listing 1, the `exhibit` keyword marks a so-called Closure join point [1], an instantly-called closure that exhibits its encapsulated code to advising aspects. Closure join points allow programmers to exhibit pieces of code as "to be advised" even if they could not normally be captured by pointcuts. In addition, closure join points are less fragile with respect to refactorings, as code annotations will typically move together with the refactored code.

Generic JPIs and advice.

One of the main contributions of JPIs is the ability to modularize the type checking of both aspects and base code

```
jpi double CheckingOutR(double price, Customer cus);
class ShoppingSession {
   void checkOut(final Item item, double price,
                 final int amount, final Customer
                        cus) {
      totalValue =
          exhibit CheckingOutR(double alteredPrice) {
              cart.add(item, amount);
              cus.charge(alteredPrice);
              return totalValue + alteredPrice;
            }(price);
    }
}
```

Listing 1: Closure Join Point

in such a way that one can guarantee the absence of type errors at composition time. To support flexible join point matching despite strong type checks, we incorporate into our language an earlier proposal for generic advice [5].

```
<R,F extends Foo> jpi R JP(F f);
```

The use of generic types allows a JPI to binds context values such as `f` to any subtype of `Foo`, but at the same type forbids pieces of `around` advice to replace those values by values of other types, thereby guaranteeing type safety. In the following, the advice cannot replace `f` by a `Foo` object because `f` may be bound to any subtype of `F`, i.e., also to a sibling type of `Foo`:

```
<R,F extends Foo> R around JP(F f) {
    //type error: Foo is not subtype of F
    return proceed(new Foo());
}
```

Controlled Global Quantification.

The ability to abstract over concrete types using generics is particularly important in combination with another language feature: controlled global quantification.

In some cases it may be awkward, and may hinder maintainability, if `exhibits`-clauses need to be spread out into too many base-code classes. This is particularly the case with widely-quantifying tracing and debugging aspects. JPIs therefore allow programmers to define global pointcuts, which are matched against any possible class in the system:

```
<R,F extends Foo> jpi R JP(F f):
    call(R *(Foo)) && args(f);
```

Interestingly, generics are a necessary requirement for global quantification to work properly. Assume for a moment that in the above code `R` and `F` were not variables but concrete types. To nevertheless assure type safety, this means that our type system would need to prevent the JPI to be matched against any join point signature with return types different from `R` and argument types different from `F`. Such specific pointcuts are often too fine-grained, as they would need to be extended for every other concrete type to be matched.

Opposed to AspectJ, global quantification in join point interfaces is *controlled*. The following syntax, for instance, allows the class `Secret` to explicitly prohibit calls to `secret` from being exposed:

```
sealed class Secret {
    exhibits Object JP() :
        global() && !call(* secret(..))
}
```

Join Point Polymorphism.

Join point interfaces further offer polymorphic dispatch on join points, with an advice-dispatch semantics akin to multimethods. In the following example, `Renting` is a subtype of `CheckingOut`, which typically means that it will expose a more restricted class of join point, about which it can expose additional context information (here `int amt`).

```
jpi void Renting(double price, int amount, Customer c)
     extends CheckingOut(price, c);
aspect Discount {
    void around CheckingOut(double price, Customer
         cus) { ... }
    void around Renting(double price, int amt,
                        Customer cus) {  ... }
}
```

3. CONCLUSION

Join point interfaces (JPIs) support flexible and safe decoupling of aspect-oriented programs through modular type checking. Like interfaces in statically-typed object-oriented languages, JPIs are a machine-checked type-based contract that supports separate development in a robust and safe manner. Key to this support is the specification of JPIs as method-like signatures with return types and checked exception types. JPIs can be organized in hierarchies to structure the space of join points in a flexible manner, enabling join point polymorphism and dynamic advice dispatch. The use of parametric polymorphism in JPI definitions, along with support for controlled global quantification, makes it possible for JPIs to preserve most of the flexibility that AspectJ offers, without sacrificing type safety.

Acknowledgements. This work was supported by the Deutsche Forschungsgemeinschaft within the project RUNSECURE, by the German Federal Ministry of Education and Research (BMBF) within EC SPRIDE and by the Hessian LOEWE excellence initiative within CASED, and by CONICYT through Inostroza's MSc grant, as well as FONDECYT project 1110051.

4. REFERENCES

[1] Eric Bodden. Closure joinpoints: block joinpoints without surprises. In *Proceedings of the 10th ACM International Conference on Aspect-Oriented Software Development (AOSD 2011)*, pages 117–128, Porto de Galinhas, Brazil, March 2011. ACM.

[2] Eric Bodden, Éric Tanter, and Milton Inostroza. Join point interfaces for safe and flexible decoupling of aspects. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 2013. To appear.

[3] Bruno De Fraine, Mario Südholt, and Viviane Jonckers. StrongAspectJ: flexible and safe pointcut/advice bindings. In *Proceedings of the 7th ACM International Conference on Aspect-Oriented Software Development (AOSD 2008)*, pages 60–71, Brussels, Belgium, April 2008. ACM.

[4] Kris Gybels and Johan Brichau. Arranging language features for more robust pattern-based crosscuts. In Mehmet Akşit, editor, *Proceedings of the 2nd ACM International Conference on Aspect-Oriented Software Development (AOSD 2003)*, pages 60–69, Boston, MA, USA, March 2003. ACM Press.

[5] Radha Jagadeesan, Alan Jeffrey, and James Riely. Typed parametric polymorphism for aspects. *Science of Computer Programming*, 63:267–296, 2006.

[6] Friedrich Steimann, Thomas Pawlitzki, Sven Apel, and Christian Kästner. Types and modularity for implicit invocation with implicit announcement. *ACM Transactions on Software Engineering and Methodology*, 20(1):1–43, 2010.

[7] Maximilian Stoerzer and Juergen Graf. Using pointcut delta analysis to support evolution of aspect-oriented software. In *Proceedings of the 21st IEEE International Conference on Software Maintenance*, pages 653–656, September 2005.