

Feature Interaction in AspectJ 5*

[Extended Abstract]

David H. Lorenz Sergei Kojarski[†]
Department of Computer Science, University of Virginia
Charlottesville, Virginia 22904-4740, USA
{lorenz,kojarski}@cs.virginia.edu

ABSTRACT

The goal of this paper is to draw attention to and raise awareness of a feature interaction problems in the integration of aspect mechanisms. We illustrate the problem by examining feature interactions between AspectJ 1.2 and AspectWerkz as found in AspectJ 5.

1. INTRODUCTION

Both AspectJ- and AspectWerkz-style aspects are supported in AspectJ 5. To a large degree, AspectJ 5 can be viewed as the product of integrating features of AspectJ 1.2 with features of AspectWerkz [4]. Such integration is generally susceptible to the *feature interaction problem*, where features interact in unexpected ways. In this paper, we illustrate that this problem indeed exists in AspectJ 5.

2. DEFINITION OF ADVICE

An example of a feature interaction problem in AspectJ 5 is related to the way AspectJ and AspectWerkz define advice. AspectJ differentiates syntactically between a method and an advice; whereas AspectWerkz does not. In AspectWerkz an advice is an annotated method with a dual purpose. The method may be executed implicitly when playing the role of advice. It may also be invoked explicitly as a regular Java method [1].

This deceptively small difference between AspectJ and AspectWerkz is of larger significance in their composition. Specifically, the method-advice duality poses a question:

Should executions of AspectWerkz advice methods in AspectJ 5 generate advice-execution join points?

This is not an issue in AspectJ 1.2, which possesses no such duality. It is also not an issue in AspectWerkz, which lacks advice execution join points. Rather, the question *emerges* in the composition.

*This work is supported in part by NSF's *Science of Design* program under Grants Number CCF-0438971 and CCF-0609612.

[†]Sergei Kojarski is a PhD candidate at Northeastern University and a visiting graduate student at University of Virginia.

Interestingly, in AspectJ 5 advice-execution join points are generated for *both* explicit and implicit advice method invocations. Consider the AspectJ aspect, `AJAdviceLogger`, which logs advice executions:

```
public aspect AJAdviceLogger {
  before(): adviceexecution()
    && !cflow(within(AJAdviceLogger)) {
    System.out.println("AJAdviceLogger:"
      + thisJoinPoint);
  }
}
```

and consider the AspectWerkz aspect, `AWAspect`, which logs calls to `foo` using a `beforeAdvice` method annotated as advice:

```
@Aspect
public class AWAspect {
  @Before("call(* *.foo(..)) && target(o)")
  public void beforeAdvice(Object o) {
    System.out.println("AWAspect:"
      + " before foo method call on "
      + o);
  }
}
```

`AJAdviceLogger` logs not only implicit `beforeAdvice` executions, but also executions that follow an explicit method calls. For example, running the program:

```
public class Main {

  public void foo() {
    System.out.println("Main: foo execution");
  }

  public static void main(String[] args) {
    new Main().foo();
    new AWAspect().beforeAdvice(null);
  }
}
```

would result in the trace:

```
1 AJAdviceLogger:execution(ADVICE: void aj5.
   AWAspect.beforeAdvice(Object))
2 AWAspect: before foo method call on base.Main@
   13582d
3 Main: foo execution
4 AJAdviceLogger:execution(ADVICE: void aj5.
   AWAspect.beforeAdvice(Object))
5 AWAspect: before foo method call on null
```

In AspectJ 5 the call

```
new Main().foo();
```

invokes `beforeAdvice` as advice; whereas the call

```
new AWAspect().beforeAdvice(null);
```

invokes `beforeAdvice` as a method. The trace illustrates that `AJAdviceLogger` logs *both* executions of `beforeAdvice` as advice executions.

We attribute this behavior of AspectJ 5 to a failure to detect this feature interaction problem. In our mind, AspectJ 5 generates *unexpected* advice-execution join points without any actual advice execution. A more intuitive feature integration resolution would establish advice-execution join points *only* for implicit advice method invocations.

3. ORDERING OF ADVICE

Another example of a feature interaction problem found in AspectJ 5 is related to the ordering of advice. Consider the AspectWerkz aspect, `Logger`, which advises calls to `foo` with a **before**, an **after**, and an **around** advice:

```
@Aspect
public class Logger {

    @After("call(* *.foo(..)")
    public void logAfterMethod(JoinPoint jp) {
        System.out.println("after:"
            + jp.getSignature());
    }

    @Around("call(* *.foo(..)")
    public Object logMethod(ProceedingJoinPoint jp)
    {
        System.out.println("around:"
            + jp.getSignature());
        return null;
    }

    @Before("call(* *.foo(..)")
    public void logBeforeMethod(JoinPoint jp) {
        System.out.println("before:"
            + jp.getSignature());
    }
}
```

The **around** advice does not proceed; it returns `null` instead. AspectWerkz orders advice only according to their type. When this aspect is woven by a stand-alone AspectWerkz weaver, all three advice are executed at every `foo` method call:

```
1 before:public void base.Main.foo()
2 around:public void base.Main.foo()
3 after:public void base.Main.foo()
```

One would expect `Logger` to behave the same way in a composition of AspectJ and AspectWerkz. However, when we run the same aspect in AspectJ 5, only the **around** advice is executed:

```
1 around:void base.Main.foo()
```

AspectJ 5 exhibits this unexpected behavior because it orders AspectWerkz advice according to the AspectJ semantics. In AspectJ, the lexical order of advice supersedes advice type in determining the ordering of advice. An **around** advice takes precedence over **after** advice that are defined before it and over **before** advice that are defined after it.

4. DISCUSSION

This position paper draws attention to the problem of *feature interaction* [5, 3] in aspect-oriented programming (AOP) [7] languages like AspectJ 5 that comprise *several* aspect extensions. Failure to handle the interactions properly may lead to unexpected and undesired behavior of the composition. There is therefore a need to detect and resolve interactions between features of the composed extensions.

The feature interaction problem occurs because semantics for an individual aspect extension Ext_1 to the base language $Base$ is given in the context of a single-extension AOP language \mathcal{L}_1 :

$$\mathcal{L}_1 = Base \times Ext_1$$

In the integrated AOP language \mathcal{L}_n however, a program contains aspects written in the various aspect extensions:

$$\mathcal{L}_n = Base \times Ext_1 \times \dots \times Ext_n$$

Programs in \mathcal{L}_n have aspects written in Ext_1 that advise not only $Base$ and Ext_1 code but also code written in foreign extensions. \mathcal{L}_n programs may exhibit unexpected behavior because the semantics for Ext_1 is defined only over \mathcal{L}_1 programs, not over the foreign aspects. For the \mathcal{L}_n language to be well-defined it must resolve all cases of undefined semantics that emerge in the Ext_1, \dots, Ext_n composition.

Figure 1 depicts a feature interaction problem in AspectJ 5. The feature interaction problem is shown in the composition of the aspect extensions as scattered semantical holes across the composition. The holes appear in features of the individual extensions and in emergent features of the composition. Specifically, AspectJ 5 comprises AspectJ's features, AspectWerkz' features, and new emergent features (e.g., an emergent advice ordering feature that orders pieces of advice written in either AspectJ or AspectWerkz).

A solution to the problem of feature interaction in the context of aspect extensions means the detection and resolution of undefined semantics when multiple aspect mechanisms are simultaneously integrated with the same base mechanism. In a composition of AspectJ and AspectWerkz, an advice ordering interaction is resolved by specifying the semantics for the emergent advice ordering. An advice definition interaction is resolved by extending the functionality of the AspectJ's join point construction feature to handle executions of AspectWerkz advice methods properly.

Understanding the feature interaction problem is crucial for specifying the correct behavior for the integration. The feature interaction between an aspect extension and the base language is relatively well understood and well defined. In contrast, the problem of feature interaction between one aspect extension and another has not been studied.

5. FUTURE WORK

AspectJ 5 is a representative of a general trend in AOSD to combine aspect mechanisms. Future work is to identify the set of features

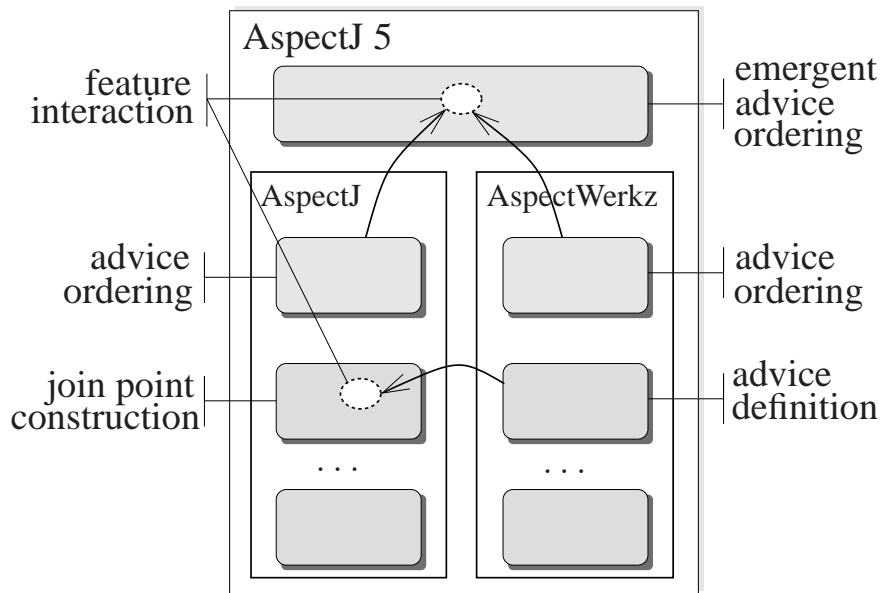


Figure 1: Feature Interactions in AspectJ5

in a typical aspect mechanism and discover the feature interaction patterns in a typical composition. This will provide a road map for resolving feature interactions similar to the ones we illustrated in AspectJ 5.

6. RELATED WORK

The Reflex framework [6] addresses the problem of unexpected behavior in multi-extension programs. The framework allows one to compose AspectJ-like extensions, and detect and resolve aspect interactions. It handles the interaction problem at the level of aspects. Interactions identified by the Reflex framework occur between same-extension aspects. In comparison, we are concerned with interaction between features of heterogeneous aspect extensions and highlighting the problem of feature interaction.

Pluggable AOP [8] presents a composition framework that supports third-part composition of arbitrary dynamic aspect mechanisms into an AOP interpreter. The Pluggable AOP framework employs a composition of aspect mechanisms, which collaborate by hiding, delegating, and exposing their expression evaluation to the other mechanisms. Pluggable AOP presents a set of collaboration design principles that guide the resolution of feature interactions between the composed mechanisms.

AspectJ 5 [2] composes AspectJ and AspectWerkz extensions. AspectJ 5 ignores individual semantical features of AspectWerkz, and applies the AspectJ mechanism to implement both extensions. In AspectJ 5 the composed extensions are simply two different syntactical interfaces to the same aspect mechanism.

As we show, AspectJ 5 does not handle all feature interactions between the two extensions. A different approach that preserves the individual features of the extensions would allow to identify feature interactions in the composition.

Lopez-Herrejon and Batory [9] study the problem of undefined aspect composition semantics in AspectJ 1.2. Although the problem

of undefined semantics in a single-extension AOP language is orthogonal to our study, the presented results can be used to specify aspect extension compositions more flexibly.

7. CONCLUSION

Different aspect extensions provide distinct concern separation capabilities. The integration of aspect extensions brings the benefit of using their capabilities together. This benefit, however, comes with the price of identifying and resolving interactions between features of the composed extensions. Failure to handle the interactions properly may lead to unexpected and undesired behavior of the composition.

This paper exposes two feature interaction problems in AspectJ 5, which integrates AspectJ and AspectWerkz. We demonstrate that for two features that AspectJ and AspectWerkz define differently and whose interaction results in unexpected behavior. The examples illustrate that feature interaction is an important and non-trivial issue that might result in unexpected behavior.

A resolution of the feature interaction problem affects the composition behavior. Each interaction can be resolved in several ways. Although some of the alternatives may seem more reasonable than others, they illustrate the complexity of the feature interaction problem. The choice of the composition semantics is the job of the composer. The problem of identifying and evaluating the desired behavior is future work.

8. REFERENCES

- [1] K. Arnold and J. Gosling. *The Java Programming Language*. The Java Series. Addison-Wesley Publishing Company, 1996.
- [2] AspectJ home page. <http://eclipse.org/aspectj/index.php>.
- [3] D. Batory. Feature models, grammars, and propositional formulas. Technical Report TR-05-14, The University of Texas at Austin, Department of Computer Sciences, Apr. 2005.

- [4] J. Bonér. Invited talk: AspectWerkz 2 and the road to AspectJ 5. In *Invited Industry Talks at*, Chicago, Illinois, USA, Mar. 14-18 2005.
- [5] W. Bouma. Feature interactions. In A. Kent and J. G. Williams, editors, *Encyclopedia of Computer Science and Technology*, volume 40, pages 63–86. Marcel Dekker, New York, 1999.
- [6] Éric Tanter and J. Noyé. A versatile kernel for multi-language aop. In *Fourth International Conference on Generative Programming and Component Engineering (GPCE'05)*, 2005.
- [7] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In M. Akşit and S. Matsuoka, editors, *Proceedings of the 11th European Conference on Object-Oriented Programming*, number 1241 in Lecture Notes in Computer Science, pages 220–242, Jyväskylä, Finland, June 9-13 1997. ECOOP'97, Springer Verlag.
- [8] S. Kojarski and D. H. Lorenz. Pluggable AOP: Designing aspect mechanisms for third-party composition. In R. Johnson and R. P. Gabriel, editors, *Proceedings of the 20th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 247–263, San Diego, CA, USA, Oct. 16–20 2005. OOPSLA'05, ACM Press.
- [9] R. E. Lopez-Herrejon and D. Batory. Taming aspect composition: A functional approach. Technical Report TR-05-27, The University of Texas at Austin, Department of Computer Sciences, June 2005.