Implementing Known Concepts in AspectJ

Arno Schmidmeier[1], Stefan Hanenberg[2] and Rainer Unland[2]

[1]AspectSoft,
Lohweg 9, 91217 Herbruck, Germany
A@schmidmeier.org

[2]Institute for Computer Science
University of Essen, 45117 Essen, Germany
{shanenbe, unlandR}@cs.uni-essen.de

Abstract. AspectJ is a so-called general-purpose aspect-language which tries to solve the problem of crosscutting code. An often uttered criticism of AspectJ is that there is no underlying concept in the implemented language features and that those features are rather arbitrary chosen instead of being based on any theoretical foundations. This paper shows beyond the the context of crosscutting and tangling code how known concepts in object-oriented programming languages can be implemented in AspectJ and argues in that way that AspectJ generalizes a large variety of features which are already known in the object-oriented world. The argumentation is based on the concepts of mixin classes, multi-dispatching and negative type information which can be implemented by using a small set of AspectJ idioms.

1 Introduction

One of the widest spread criticism of AspectJ [1] is that that it supports some mechanisms for preprocessing source code, but that there is no underlying theoretical (or even practical) concept in it. So the main problem seems to be that the language features of AspectJ are more understood as a tool for performing some source code transformations instead of a self-sufficient programming language. The typical introduction to AspectJ is done by arguing that AspectJ is about untangling tangled code or crosscutting code. Such argumentation is not very convincing, because the terms tangling or crosscutting are neither well-defined nor is there are common understanding of these terms[1]. The problem in this context is that the crosscutting phenomenon is usually reduced to a number of examples like tracing, synchronization and persistency and it is usually hard for a developer to understand how to use AspectJ in other situations. However, the new language features of AspectJ built upon Java like introduction, advice and pointcuts permit to use a lot of mechanisms which are widely known, but not part of the programming language Java.

In contrast to the usual papers on AspectJ about solving problems of code tangling, we analyze in this paper the usage of AspectJ language features to implement concepts and mechanisms well-known in the object-oriented world. That means, we show apart beyond the usual discussion on crosscutting code that desirable language features which are not available in Java can be easily implemented in AspectJ. Hence, AspectJ it is still a valuable complement for the development of Java based applications independent of the discussion on what kind of crosscutting code can be handled by AspectJ. In our argumentation we refer to the concepts of mixin classes, dynamic dispatching and negative type information. We show how corresponding implementations in AspectJ look like and discuss their parallels and differences to the known mechanisms.

2 Concepts and Mechanisms in AspectJ

2.1 Mixin Classes

Originally, the term *mixin class* was introduced in object-oriented extensions of Lisp like shown in [2] or [11]. A mixin is an abstract subclass that may be used to specialize the behavior of a variety of parent classes. Mixins don't have superclasses and are therefore not structurally bound to any specific place in the inheritance hierarchy. Just when a certain class extends a mixin, it becomes part of the inheritance structure for this particular

[1] Works like [17] propose a formal semantics for advice based on the implementation of AspectJ. However, in that way this work gives an explanation for the proposed solution offered by AspectJ (restricted to advice and neglecting introductions), but does not describe the relationship between the crosscutting phenomenon and the supposed solution.

instantiation. In that way a mixin class is inserted into the class hierarchy. In Lisp based object systems like CLOS mixins are usually used to realize multiple inheritance. The intention of mixins is, that they should not be instantiated on their own. Instead, they should depict "reusable pieces of code that can be added to other classes […]" [15]. The benefit of mixins is widely known for example for building *layered object-oriented design* [14] or *static roles objects* [16][2].

Figure 1 shows the usage of a mixin class: the mixin consists of two method definitions `m1` and `m2`. The method body of the mixin's body might already refer to other classes expected from the inheritance hierarchy the mixin is applied to. For example method `m1()` refers to a method `m3()`, which is not defined in the mixin class itself. When the mixin is used by a concrete class (in the example class `B` and class `D`), the mixins becomes part of the inheritance hierarchy. In figure 1 both applications of the mixin `M` declare a different superclass to the mixin. The developer who uses the mixin class is responsible for the correct usage. In this example it means, that the developer has to guarantee, that the superclass of the applied mixin contains a method `m3`.

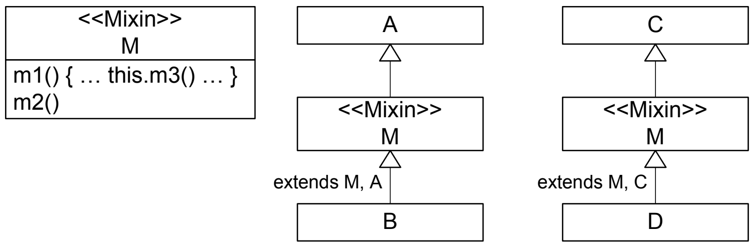

Figure 1: Usage of Mixins

Popular languages like C++, Smalltalk or Java do not provide any direct support for mixins on the language level. However, for example in C++ mixins can be implemented by a class template extending a class which is delivered by a type parameter (like proposed in [16]). Also, the meta-object facilities of Smalltalk can be used for implementing mixins as shown in [11]. Nevertheless, until now mixins are not available in Java on language level (although more recent proposals about adding generic types to Java like for example [2] will probably become part of Java in future versions and will permit a similar implementation of mixins like in C++).

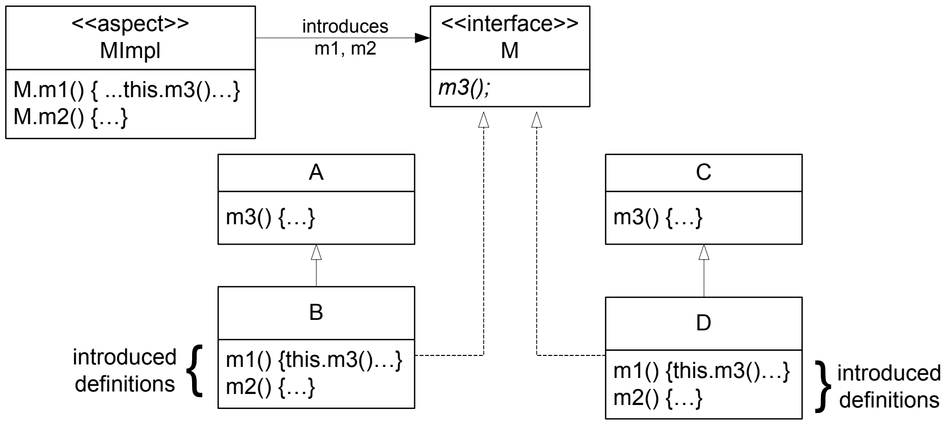

Figure 2: Mixins in AspectJ using introductions

AspectJ has a mechanism which is quite similar to mixins. *Introductions*[3] which are defined within an aspect permit to introduce members or ancestor to a target class or interface. If members are introduced to an interface (that means the *container introduction* idiom [8] is used), all classes implementing this interface will receive the new introduced members. Figure 2 illustrates the usage of mixins in AspectJ in correspondence to figure 1. An aspect `MImpl` introduces methods `m1` and `m2` to an interface `M`. The classes `B` and `D` implementing `M` receive the method definitions from the introducing aspect. Hence, in the resulting woven application the method definitions of `m1` and `m2` occur in two different classes.

In contrast to the "traditional" usage of mixins, AspectJ does not perform any linearization of the inheritance hierarchy. That means the mixin class represented by the interface `M` is not physically inserted into the inheritance hierarchy of classes `A`, `B`, `C` and `D`. Although it does not lead to any problems in the here mentioned example, the problem arises when two different mixin interfaces come with two method implementation with

[2] In [5] Filman and Friedman already argue that mixins are a kind of *oblivious quantification*, which they regard central to the idea of aspect-orientation programming.
[3] Nowadays, the term introduction is replaced with the term *intertype declarations*, but we still feel allied to the term introduction because of its frequent usage.

identical signatures like illustrated in figure 3. The compiler does not permit to implement two interfaces to which methods with the same method signatures are introduced. That means for the given example that it is not possible to introduce the containers `M1` and `M2` at the same time to a target class `A`.

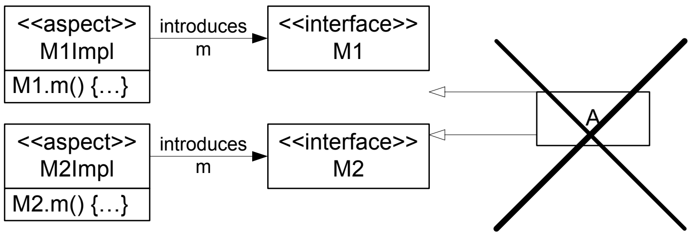

Figure 3: Conflicting Introductions in AspectJ

On the other hand the mixin implementation in AspectJ is "more safe" than for example the C++ implementation of mixins based on class templates as proposed in [16]: the usage of superclass methods is just permitted, when the corresponding method is declared within the interface (so the application of introductions to an interface "feels" more like the traditional F-bounded polymorphism [3]).

The problem with conflicting AspectJ mixins is, that this does not permit to build complex hierarchies of mixin-layers as for example used in [14]: it is not possible that an introduced method overrides the implementation of another method (it doesn't matter if introduced or not). In that way, to be able to implement mixin layers in AspectJ, the mixins have to be carefully preplanned. Hence, the benefit for such mixins much more limited than its pendant in C++. On the other hand, the usage of AspectJ mixins permits an non-invasive change of the classes to whose inheritance hierarchy the mixins should be applied to: the implements-relationship between a class and an interface does not need to be defined within the class declaration, but can be introduced by an aspect (see [8] for further discussion).

2.2 Multiple Dispatching

In contrast to programming languages like e.g. Cecil [5] which provide multi-dispatching Java provides single dispatching: the method to be invoked depends only on the runtime-type of the target object. Nevertheless, there are situations where it is desirable to dispatch not only in dependence of the target object, but also of the passed parameters.

An example for a need of double-dispatching is the visitor design pattern [7]. A typical implementation of the visitor pattern in Java is given in figure 4. All classes implementing the `VisitedElement` interface have to implement the method `accept`. This also means that within an inheritance hierarchy for all subclasses which should be visited the method has to be overridden with exactly the same implementation. On the one hand one can argue that this method depicts static crosscutting code which can be handled using introductions (following the argumentation in [9]). On the other hand one can argue that this method arises because of the single dispatch mechanism in Java[4]. Since dispatching is a mechanism which redirects messages and AspectJ permits to redirect messages via advice, AspectJ permits to redefine the underlying dispatch mechanism of Java for certain situations (that means different join points).

```
interface VisitedElement {
  public void accept(Visitor v);
}
class A implements VisitedElement {
  public void accept(Visitor v) {v.visit(this);}
  ...
}
class B extends A {
  public void accept(Visitor v) {v.visit(this);}
  ...
}
class C extends A {
  public void accept(Visitor v) {v.visit(this);}
  ...
}
```

```
interface Visitor {
....void visit(VisitedElement node);
    void visit(A node);
    void visit(B node);
    void visit(C node);
}

class ConcreteVisitor implements Visitor {
....void visit(VisitedElement node);
    void visit(A node) {.....}
    void visit(B node) {.....}
    void visit(C node) {.....}
}
```

Figure 4: Visitor implementation in Java

In the case of the visitor implementation this means, that a double dispatching is needed: the implementation to be invoked should not only depend on the target object which is of type `Visitor`, but also on the actual

[4] It should be noted that a visitor implementation can also be done using the introspection facilities of Java like discussed in [12].

parameter. In AspectJ terminology that means, that there is a number of pointcuts which are related to the same (static) call expression, but which differ in their runtime representation.

Figure 5 shows the corresponding implementation: aspect `VisitorDispatcher` is responsible for dispatching a `visit` message to the corresponding method. The pointcuts used in each around advice are mutual exclusive that means each around advice is related to different join points (but to same static call expression). Within the pointcut `visitedCall` the pointcut `within(VisitorDispatcher)` guarantees that only calls outside the `VisitorDispatcher` are redirected. The interesting feature of this implementation is that the redirection is type-safe (in contrast to an implementation based on introspection). The objects passed to the pointcuts are still statically typed so the method calls within the advice can be checked statically. The implementation within the advice still makes use of the single dispatching in java since the arguments' types are determined by the pointcut.

```
interface VisitorElement {
  public void accept(Visitor v);
}
class A implements VisitorElement {
  public void accept(Visitor v) {v.visit(this);}
  ...
}
interface Visitor {
    void visit(A node);
    void visit(B node);
    void visit(C node);
}

class ConcreteVisitor implements Visitor {
    void visit(A node) {.....}
    void visit(B node) {.....}
    void visit(C node) {.....}
}
```

```
public aspect VisitorDispatcher {
  public pointcut visitCall(Visitor v):
    !within(VisitorDispatcher) &&
    call(void Visitor.visit(*)) && target(v);
  void around(Visitor v, A a):
  visitCall(v) && args(a) && !args(B) && !args(C)
    v.visit(a);
  }
  void around(Visitor v, B b):
  visitCall(v) && args(b) && !args(C){
    v.visit(b);
  }
  void around(Visitor v, C c):
  visitCall(v) && args(c) && !args(B){
    v.visit(c);
  }
}
```

Figure 5: Visitor in AspectJ using double-dispatching

It should be noted, that the implementation suffers from one problem: there are tangled statements inside the around advice. That is interesting, since the aim of AspectJ is to eliminate tangled code. However, AspectJ does not offer any serious features to reduce this code tangling. Although it is possible to move the implementation of the concrete visitor to the corresponding advice, its usability is quite restrictive since aspects cannot be directly instantiated by the programmer and only abstract aspects can be extended.

As shown above the language features of AspectJ can be used to simulate multi-dispatching. Pointcuts which refer to the same static call expressions, but differ in their actual parameters can be used to determine the condition under which the message should be redirected. The interesting point in implementation is, that multiple dispatching can be added later to an application instead of being hard-coded inside the programming language. Although there are some possibilities in Java to simulate some kind of multi-dispatching by using the reflection API, such an implementation has numerous disadvantages like the problem with error handling, etc.

2.3 Negative Type information

Negative type information like proposed in [13] prohibit that extensions of a given class have certain members. A practical example for such classes is a stateless class as for example known from stateless session beans in Enterprise JavaBeans. The intension of such a class is, that none of its subclasses is permitted to have a field which represent some kind of state within its objects.

```
class StatelessClass {}

class StatelessClassImpl extends StatelessClass {
    int i = 0;
}

public aspect StatelessClassRestriction {
  declare error:
    set(* StatelessClass+.*) || get(* StatelessClass+.*):
    "MyErrorDefinition: is an error";
}
```

Figure 6: Implementing Negative Type Implementations in AspectJ

AspectJ permits to declare errors in aspects which are checked at compile-time. An error declaration consists of a (static) pointcut specifying the elements whose occurrence leads to throwing a compile-time error and an error message which is printed when such an error occurs. Although the error declarations within aspects can be defined for any existing static join point, it is usual, that such errors are declared for classes along a certain inheritance hierarchy.

Figure 6 shows how a stateless class can be implemented using an error declaration in AspectJ. The aspect `StatelessClassRestriction` defines an error which is thrown whenever there is a field assignment of a member of a `StatelessClass` subclass. The AspectJ compiler prevents the classes in the example from being compiled, since there is a field assignment of field `i` in class `StatelessClassImpl`. Although AspectJ error declarations seem to be appropriate for the here mentioned problem in fact the reason why the error is thrown is different than assumed: not the member definition is responsible for throwing the error but the field assignment. This is different than the usual intention to prevent classes to define any fields.

The reason for this is that the pointcut language which is part of AspectJ is designed more to describe messages than meta information.

3 Conclusion

In this paper we argued, that it is easy to implement a number of well-known object-oriented concepts in AspectJ. We showed parallels between introductions and mixin classes, multiple pointcuts to the same static call join point (but to different dynamic join point) and dynamic dispatching and error declarations to negative type information.

The main intention of this paper is to show, that independent of the discussion about code tangling and crosscutting code AspectJ is a valuable complement for the development of Java based applications which permits to realize a large variety of desirable features. The impact for developers is that AspectJ (including its IDE support, etc.) can be more likely used instead of occasionally using language extensions of Java which are built to handle specific sets of problems. Nevertheless, we also showed that there are still some differences between the AspectJ implementation and the underlying concepts which reduce their applicability. For example the missing ability to override methods of different introducing aspects reduces the ability to build complex mixin layers. The implementation of a multi-dispatch mechanism might lead to complex aspect definitions, because it must be guaranteed, that the dynamic join points are mutually exclusive. Furthermore, the implementation might lead to tangled code. The usage of error declarations to define negative type information might lead to the desired result.

The reason for showing the benefits of AspectJ as a programming language beyond the usual discussion of crosscutting and tangling code is that we noticed that it is really hard to communicate the problem of crosscutting code. It seems as if the phenomenon of crosscutting code is still far from being really understood and most approaches tend to provide solutions of already solved problems of crosscutting code. So convincing a developer of the benefits of aspect-oriented programming seems to be much easier (and more promising) using the argumentation of missing language features of a core language which can be easily implemented using aspect-oriented techniques.

4 References

[1] AspectJ Team: *The AspectJ Programming Guide*, http://aspectj.org/doc/dist/progguide/.

[2] Bracha, G.; Cook, W.: *Mixin-based Inheritance*. In: Norman Meyrowitz (Ed.). OOPSLA / ECOOP'90 Conference Proceedings, ACM SIGPLAN Notices 25, 10, pp. 303-311.

[3] Canning P.; Cook, W.; Hill, W.; Olthoff, W.; Mitchell, J. C.: *F-Bounded Polymorphism for Object-Oriented Programming*, In Proc. International Conference on Functional Programming Languages and Computer Architecture, 1989, pp. 273-280.

[4] Bracha, G.; Odersky, M.; Stoutamire, D.; Wadler, P.: *Making the future safe for the past: Adding Genericity to the Java Programming Language,* Proceedings of Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA), 1996. pp. 359-369, 183 – 200.

[5] Chambers, C.: *Object-Oriented Multi-Methods in Cecil*, ECOOP'92 Conference Proceedings, Utrecht, The Netherlands, July, 1992.

[6] Filman, R. E.; Friedman, D. P.: *Aspect-Oriented Programming is Quantification and Obliviousness*, In: Workshop on Advanced Separation of Concerns, OOPSLA, 2000.

[7] Gamma, E.; Helm, R.; Johnson, R.; Vlissides, J.: *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995

[8] Hanenberg, S.; Costanza, P.: *Connecting Aspects in AspectJ: Strategies vs. Patterns*, First Workshop on Aspects, Components, and Patterns for Infrastructure Software at 1st International Conference on Aspect-Oriented Software Development (AOSD), Enschede, April, 2002.

[9] Hanenberg, S.; Unland, R.: *Parametric Introductions*, To appear in: 2nd Internation Conference on Aspect-Oriented Software Development, Boston, March, 2003.

[10] Montlick, T.: *Implementing Mixins in Smalltalk*, The Smalltalk Report, July 1996.

[11] Moon, D. A.: *Object-Oriented Programming with Flavors*, Proceedings of Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA), 1986, pp. 1-8.

[12] Palsberg, J.; Jay, C.B.: *The Essence of the Visitor Pattern*, 22nd International Computer Software and Application Conference, 1998, pp. 9-15.

[13] Rémy, D.: *Typechecking records and variants in a natural extension of ML*. In Proc. 16th ACM Symp. Principles of Programming Languages, pages 242–249. ACM Press, 1989.

[14] Smaragdakis, Y.; Batory, D.: *Implementing Layered Designs with Mixin Layers*, Proceedings of 12th European Conference on Object-Oriented Programming (ECOOP), LNCS, 1445, Springer-Verlag. 1998

[15] Taivalsaari, A.: *On the Notion of Inheritance*. In: ACM Computing Surveys, Vol. 28 (1996), No. 3, pp. 439-479.

[16] VanHilst, M.; Notkin, D.: *Using Role Components to Implement Collaboration-Based Designs*, Proceedings of Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA), pp. 359-369.

[17] Wand, M.; Kiczales, G.; Dutchyn, C.: *A semantics for advice and dynamic join points in aspect-oriented programming*, Foundations of Object-Oriented Languages 9, 2002