Specifying Aspect-Oriented Design Constraints in AspectJ

Stefan Hanenberg and Rainer Unland
Institute for Computer Science
University of Essen, 45117 Essen, Germany
{shanenbe | unlandR}@cs.uni-essen.de

ABSTRACT

Aspect-oriented programming comes with new composition mechanisms which permit to modularize code which crosscuts other modules using traditional composition techniques. Although such mechanisms permit a better modularization they do not guarantee it: if the aspect-oriented code is better modularized depends on the design of the aspect-oriented applications. This paper describes typical design failures in AspectJ and introduces a tool implemented on top of AspectJ which permits to specify design constraints on AspectJ code.

1. INTRODUCTION

Aspect-oriented software programming [5] is about modularizing concerns which cannot be cleanly encapsulated using traditional composition techniques. To achieve this better kind of separation of concerns [2] aspect-oriented languages like AspectJ [1] offer different composition mechanisms on top of the existing ones. Nevertheless, although these mechanisms permit to modularize crosscutting code, they do not guarantee a better modularization. If the resulting aspect-oriented applications really modularize the different concerns depends on the programmer's design decision. A misuse of the composition mechanism might directly lead to bad modularization and incorrect applications.

AspectJ already offers to specify constrains on the object-oriented application to be woven by declaring errors which refer to pointcuts. The pointcuts are related to object-oriented features in Java but not to aspect-oriented features in AspectJ. Hence, it is not possible to specify constraints on the usage of the new composition mechanisms *pointcut*, *advice* and *introduction*. In that way error declarations in AspectJ do not help to check if certain design constraints are obeyed in a given application or not. This paper presents a tool which permits to specify design constraints on the usage of the aspect-oriented features of AspectJ.

In the following section we describe a number of typical design failures committed when developing applications in AspectJ. There we concentrate on the usage of *introductions*. Afterwards we present the tool AJDC (*AspectJ Design Checker*) and illustrate its basic features. Afterwards we show how the tool permits to prevents programmers to commit the previous shown design failures. Finally, we conclude the paper and give an outlook on specifying design constraints in the code.

2. TYPICAL ASPECTJ DESIGN FAILURES

2.1 Tangled introductions

A typical implementation of the visitor design-pattern [3] in AspectJ is shown in figure 1: two interface `VisitedElement` and `Visitor` are created and the double-dispatch method is introduced to the interface `VisitedElement`. Hence, every class implementing the interface gets the double dispatch method. To adapt the abstract visitor aspect it has to be connected to the target classes, that means the interface `VisitedElement` just has to be introduced. There are different ways of performing such introductions.

```
interface VisitedElement {}
interface Visitor {
  visit(A node); visit(B node); visit(C node);
}
aspect VisitedElementLoader {
  public void VisitedElement+.accept(Visitor v){
    v.visit(this);
  }
}
class ConcreteVisitor implements Visitor{...}
```

Figure 1: Abstract Visitor

For example one aspect could introduce the interface to all classes in different `declare parents` statements or one aspect could introduces the interface in to all targets by using an appropriate type pattern. Nevertheless, it is also possible to define an aspect for each visited class and introduce there `VisitedElement`.

```
aspect MakeAVisitedElement {
  declare parents: A implements VisitedElement;}
aspect MakeBVisitedElement {
  declare parents: B implements VisitedElement;}
aspect MakeCVisitedElement {
  declare parents: C implements VisitedElement;}
```

Figure 2: Tangled introductions

Figure 2 shows such a code which connects the visitor by defining one aspect for each target class. In that case the introductions, which logically belong to one single concern occur in several aspects and contradict the aimed separation of concerns, because the introductions are not separated in one single module. We call such occurances of introductions *tangled introductions*. In this concrete example the tangled introductions obviously contradict the aimed design of the visitor aspect.

2.2 Container Misusage

In [4] we designated the previous implementation as an application of an often occurring pattern called *indirect introduction*: the interface `VisitedElements` depicts a container to which members are introduced and which can be later assigned to a target class. The intention of a container is to be used only within introductions. Nevertheless, if a class directly implements the container it automatically receives all its members. This is usually a design failure: a container is a placeholder for a number of extrinsic features. By directly implementing such a container the difference between intrinsic and extrinsic features is mixed up. Figure 3 shows such a misuse of a container. Although there are no tangled introductions, the implements-relationship between `D` and `VisitedElement` comes logically from the visitor concern. Hence, this relationship should be specified in the same module like the other introductions. We call such design failures *container misusage*.

```
aspect AllVisitedClasses {
  declare parents: A implements VisitedElement;
  declare parents: B implements VisitedElement;
  declare parents: C implements VisitedElement;}
class D implements VisitedElement {...}
```

Figure 3: Container Misusage

2.3 Fragile, Aspect-Dependent Classes

Classes which directly access introduced members are quite fragile, since they directly depend on the aspect which performs the introduction but this dependency cannot be seen in the code. Furthermore such classes mix up intrinsic and extrinsic class members: an introduced member is extrinsic and concern specific while the usual class members are usually regarded as intrinsic members. If an intrinsic member depends on an extrinsic member, the difference between intrinsic and extrinsic features is mixed up.

```
aspect LogNumberOfFieldAccesses {
  private int (TargetClass).numFieldAccesses;
}
class TargetClass {
  ...
  public String toString() {
    return "#fieldAccesses=" + numFieldAccesses;
  }
}
```

Figure 4: Fragile, aspect-dependent class

In case the aspect performing the introduction is only temporarily woven, e.g. just for debugging purposes, and not part of the final product the class cannot be compiled if the aspect is unwoven. Figure 4 contains such a fragile, aspect-dependent class which uses the introduced member `numFieldAccesses` in method `toString`. Since in this concrete case the aspect is just woven for debugging purposes, `TargetClass` is fragile, since it cannot be compiled without the woven aspect. We call classes which depend on introduced members *fragile classes*.

2.4 Accidental Overridden Extrinsic Methods

Methods introduced to a class can be overridden by subclasses. Such things usually happen accidental, because the programmers of the subclass are not aware of the introduced extrinsic method.

Accidental overridden extrinsic methods often lead to unexpected behavior: the introducing aspect expects a certain behavior of the target class because of the introduction, but instead subclasses of that aspect behave in a different way. From our point of view extrinsic members should be overridden only by aspects which perform an introduction on subclasses, because this makes clear that an extrinsic feature is overridden.

```
aspect PerformIntroduction {
  public void (TargetClass).doSomething() {...}
}
class TargetClass {}
class TargetClassSubClass extends TargetClass {
  // accidental overriden method
  public void doSomething() {...}
}
```

Figure 5 : Overridden extrinsic method

2.5 Introducing Mutual Exclusive Containers

Assume two abstract aspects in the system like given in figure 6: every class to which `Subject` is introduced can play the role of a subject in the observer design pattern, i.e. whenever the state of the subject changes its observers are informed. Classes to which `FieldAccessCount` is introduced have a variable `counter` which is increased every time a public field is accessed.

```
public interface Subject {}
public interface FieldAccessCount {}
aspect SubjectLoader {
  public List Subject.obs = new ArrayList();
  pointcut stateChanges(Subject s):
    (set(* *)  && target(s) && target(Subject));
  after(Subject s): stateChanges(s) {
    for(Iterator i=s.obs.iterator();i.hasNext();)
      ((Observer)i.next()).update();
  }
}
aspect FieldAccessCountLoader {
  pointcut fieldAccess(FieldAccessCount c):
    (set(public * *) || get(public * *)) &&
    target(c) && target(FieldAccessCount);
    after(FieldAccessCount c): fieldAccess(c){
          c.counter++;
    }
    private int FieldAccessCount.counter=0;
}
aspect ErroneousMutualExclusiveAspectConnection {
 declare parents: X implements Subject;
 declare parents: X implements FieldAccessCount;
}
```

Figure 6 : Weaving Mutual Exclusive Aspects

At the design level it is clear that both interfaces should never be connected to the same target class, since both containers (and the aspects they represent) are mutual exclusive: every time the field `obs` is accessed the counter is increased which directly leads to informing the observers, and so on. Since both aspects are designed to be mutual exclusive, the developer should be prevented from connecting both to the same class at weave time.

3. AJDC: ASPECTJ DESIGN CHECKER

AspectJ provides the possibility to specify constraints within the code. Errors and warnings can be declared which consist of a referring pointcut and a message. At weave time it is checked if any errors and warning should be thrown. In such a case the specified message it shown to the developer. In AspectJ pointcuts just refer to object-oriented constructs like method calls but do not permit to refer to any aspect-oriented feature. Hence, it is not possible to declare any errors which might occur because of design failures caused by introductions like illustrated above.

AJDC (*AspectJ Design Checker*) is a small extension of the language AspectJ. It permits to write errors and warnings which are based on AspectJ language features. Hence, it permits to specify constraints on object-oriented *and* aspect-oriented features. For specifying the errors (and warnings), AJDC provides an own (logical) pointcut language which is based on the logical programming language TyRuBa [6]. AJDC generates facts and rules out of the parse tree to be compiled and provides some rules for retrieving information like subclass relationships. For example the predicate `fieldAccess` permits to determine all field accesses, or `class` permits to determine all classes to be compiled.

```
aspect ForbidMyClassSubclass {
  ajdcError forbidSubclass
   {No MyClass subclass allowed: ?class} =
   forbiddenSubclass(?class);
  ajdcPointcut forbiddenSubclass(?class) =
    class(?classID,MyClass) &&
    superclass(MyClass,?class);
}
```

Figure 7: Forbid `MyClass` subclasses

In addition to AspectJ AJDC provides three new keywords: `ajdcError`, `ajdcWarning` and `ajdcPointcut`. The first ones are for specifying the error or warning messages which should be thrown, `ajdcPointcut` specifies the pointcuts responsible for triggering the errors and warnings. All three constructs are specified within classes or aspects. In the pointcut definition the programmer can use all language features of TyRuBa. The syntax slightly differs from TyRuBa. For example conjunction and disjunction are expressed by the boolean operators || and && from Java.

```
Error thrown by " forbidSubclass " in
"ForbidMyClassSubclass":
{No MyClass subclass allowed: A}

Error thrown by " forbidSubclass " in
"ForbidMyClassSubclass":
{No MyClass subclass allowed: B}

2 errors
```

Figure 8: Thrown Error messages

Figure 7 shows an example of an error declaration in AJDC: an error `forbidSubclass` with the error message `{No MyClass subclass allowed: ?class}` which refers to a pointcut `forbidSubclass`. The error message contains a logical variable which is replaced every time the corresponding pointcut delivers a corresponding value. The pointcut substitutes `?class` with the name of each subclass of `MyClass`. That means for every subclass of `MyClass` an error is thrown at compile time. Of special interest is the variable `?classID` in the predicate class. For each node in the parse tree a unique id is generated. This id can be used for performing more complex queries on the parse tree.

In case there are classes `A` and `B` that extend `MyClass` the compiler throws an exception as shown in figure 8.

4. ASPECT-ORIENTED DESIGN CONSTRAINTS IN ASPECTJ

In this paper we neglect to introduce the whole features of AJDC. Instead we just introduce only those which are used to specify design constraints to protect developers from the previous mentioned failures.

4.1 Tangled Introductions

The tangled introduction in 2.1 occurs, because the interface `VisitedElement` was introduced in more than one aspect to the corresponding target classes. Hence, at weave time it should be checked if such a design failure was made and the programmer should be prevented from compiling such tangled introductions.

```
aspect VisitedElementLoader {
  ajdcError forbidTangledIntroductions {..msg} =
   allIntroducingAspects(?aspect);
  ajdcPointcut allIntroducingAspects(?aspect) =
    FINDALL(
      introducedType(?aspect, VisitedElement),
      ?aspect, ?aspectList
    ) && length(?aspectList,?length) &&
    greater(?length,1) &&
    member(?aspect,?aspectList);
  ...
}
```

Figure 9: Preventing tangled VisitorElement introduction

Figure 6 contains the code which specifies that an error should be thrown when a programmer tries to compile code which contains a tangled introduction of `VisitedElement`. The pointcut `allIntroducingVariables` has one parameter (`?aspect`). In case there is more than one aspect which performs an introduction of `VisitedElement`, the pointcut substitutes `?aspect` with the aspects' names and delivers it to the error definition. `FINDALL`, `greater` and `member` are valid TyRuBa terms and will not be discussed here. The term `introducedType(?aspect, Visited)` substitutes `?aspect` with all aspects which introduce `VisitedElement`.

4.2 Container Misuse

To prevent container misuse an exception should be thrown whenever a class directly implements a container, i.e. the implements relationship exists without any corresponding introduction. AJCD provides the predicate `implements` with two parameters for getting all interfaced implements by a class: the term `implements(Aclass,?interface)` substitutes all interface implemented by `AClass` to the variable `?interface`. The term `introducedType` with three parameters can be used

to determine what interfaces are introduced to what classes in a certain aspect. E.g. `introducedType(?aspect, MyInterface,?targetClass)` determines all aspects which introduce `MyInterface` to a target class.

```
aspect VisitedElementLoader {
  ajdcError forbidContainerMisusage {..msg} =
    misusingContainerClass(?class);
  ajdcPointcut misusingContainerClass(?class) =
    implements(?class, VisitedElement) &&
    NOT(introducedType(?introducingAspect,
                       VisitedElement,?class));
  ...
}
```

Figure 10: Preventing VisitorElement misuse

To prevent the programmer from a container misusage like given in figure 3 we specify an error every time a class implements `VisitedElement` without the existence of a corresponding introduction. The code is given in figure 10. The pointcut `misusingContainerClass` determines all classes which implement the interface `VisitedElement` whereby this implement-relationship does not come from an introduction.

4.3 Fragile Classes

To prevent fragile classes like shown in figure 4 error messages should be thrown whenever introduced members are used within ordinary classes.

```
public class WarnFragileFieldAccess {
  ajdcWarning fragileFieldAccessWarning{..msg} =
    fragileFieldAccess (?class,? fN,?tClass);
  ajdcPointcut fragileFieldAccess(?c,?fN,?tc) =
    class(?classID,?c) &&
    field(?fieldID,?tc,?type,?fN) &&
    fieldAccess(?fAID,?tc,?type,? fN) &&
    parentNode(?classID,?fAID) &&
    introduced(?fieldID) &&
    NOT(introduced(?fAID);
}
```

Figure 11: Preventing Fragile Classes

The clause `fieldAccess(?fAID,?tc,?type,?fN)` determines all accesses of fields having the identifier `?fN` of type `?type` in class `?tc` and the id `?id`. The term `introduced(?id)` determines if the node with the id `?id` was introduced. The clause `field(?fieldID,?tc,?type, ?fName)` determines fields with the identifier `?name` of type `?type` in class `?tc` (including super-class fields). `ParentNode` determines all nodes in the parse tree which are parents of second parameter. Hence, the pointcut `fragileFieldAccess` in figure 11 determines all field not introduced accesses of introduced fields.

4.4 Introducing Mutual Exclusive Containers

To prevent the introduction of mutual exclusive containers to one class the developer has to define that such containers cannot be connected to the same target classes. That means a pointcut referred by an error declaration has to be defined which determines if a there is a class which implements the mutual exclusive containers. The corresponding pointcut for the example from figure 6 is shown in figure 12.

```
public class ... {
  ...
  ajdcPointcut fragileFieldAccess(?c,?fN,?tc) =
    class(?classID,?c) &&
    implements(Subject,?c) &&
    implements(FieldAccessCount,?c);
}
```

Figure 12: Preventing mutual exclusive aspect weaving

5. CONCLUSION

In this paper we identified some often occurring design failures committed in AspectJ applications which contradict the achieved aim of separation of concerns and presented the tool AJDC which permits to specify aspect-oriented design constrains on AspectJ application. Such constraints are specified inside the code and checked at weave-time.

Since design constraints are specified inside the code they become part of the module which is logically responsible for the constrains. Nevertheless, there are constraints, which are not logically part of one single module. For example the prevention of fragile classes is more a general design guideline than an aspect-specific design constraint. Hence, for such cases new modules should be created which just contain the constraint definition. Furthermore, the developer has to be careful when defining constraints. For example it is not that clear when a container is misused: there are examples which prefer to connect to container directly declaring a corresponding implements in the class defintion.

Mainly, design constraints are interesting for developers which provide collections of abstract aspect which assume a certain design when connected to applications.

6. ACKNOWLEDGMENTS

We would like to thank Arno Schmidmeier for his help during endless discussions about aspect-oriented design.

7. REFERENCES

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

[2] Dijkstra, E.: A Discipline of Programming. Prentice-Hall, 1976.

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

[4] Hanenberg, S., Costanza, P., Connecting Aspects in AspectJ: Strategies vs. Patterns, First Workshop on Aspects, Components, and Patterns for Infrastructure Software, 2002

[5] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., Irwing, J., Aspect-Oriented Programming. Proceedings of ECOOP '97, 1997.

[6] De Volder, K., D'Hondt, T., Aspect-Oriented Logic Meta Programming, Proceedings of Reflection '99, 1999