Pascal Costanza: Vanishing Aspects
Workshop on Advanced Separation of Concerns at OOPSLA 2000,
Minneapolis, Minnesota, USA, October 2000

Note: This paper tries to solve the issue of vanishing aspects, a particular problem that occurs in the context of Aspect-Oriented Programming. However, the solution presented is not complete - for example, the features of AspectJ that enable reasoning about the dynamic control flow of an application are much more powerful. Still, this paper gives a good and illustrative example of vanishing aspects and is therefore occasionally referenced in other work. For this reason, I still provide it on my webpage. Please contact me for further information at costanza@web.de

Vanishing Aspects

Pascal Costanza
University of Bonn, Römerstraße 164
D-53117 Bonn, Germany
costanza@cs.uni-bonn.de

August 4, 2000

1 Jumping Aspects

At the workshop "Aspects and Dimensions of Concerns" at the 14th European Conference on Object-Oriented Programming, a paper about "Jumping Aspects" has been presented [2]. Its main message is that there are cases where "join points seem to dynamically jump around", depending on the context certain code is called from. One of its examples is a list data structure with operations to add a single element (add) and to add a collection of elements (addAll). In order to signal changes of a list data structure for example to a graphical interface, each operation has to be augmented with an appropriate send of a changed message. This is obviously a typical application of aspect-oriented programming approaches. Since in the given example the operation addAll (for a collection of elements) calls add (for a single element) repeatedly, the unnecessary multiple sends of changed should be cumulated into a single send of changed for efficiency reasons. This can be reformulated as an example of a join point that has to "jump out" out of a method into a calling context.

In the following section we give an example that is complementary to those presented in that paper. Our example illustrates how join points may erroneously disappear depending on how a called method is implemented. Furthermore, we present a suggestion how this problem may be tackled in aspect-oriented programming approaches, which also solves the problem of jumping aspects.

2 Disappearing counters

Consider the class java.io.OutputStream of Java's standard API [3]. We focus our investigation on the following two methods declared in that class:

```
public class OutputStream {

  public void write(byte b) {...}

  public void write(byte[] b) {...}

}
```

In order to simplify references to these methods in the following paragraphs, we name the first writeByte and the second writeArray.

Consider that a programmer wants to write an aspect that logs the number of bytes that have actually been written to a stream, presuming that writeArray is coded in terms of writeByte as follows:

```
public class OutputStream {

  public void write(byte b) {...}

  public void write(byte[] b) {
    for (int i = 0; i < b.length; i++) write(b[i]);
  }

}
```

In this case an aspect that increases a counter inside of writeByte should suffice to solve this task.

However, this approach is too naive, since it depends heavily on this presumption. In fact, the standard implementation fulfils this presumption, but subclasses of OutputStream might override the involved methods arbitrarily. For example, one subclass might reimplement them so that writeByte is coded in terms of writeArray as follows:

```
public class OneOutputStream extends OutputStream {

  public void write(byte b) {
    byte[] a = new byte[1]; a[0] = b; // create an array holding "b"
    write(a);                         // call writeArray
  }

  public void write(byte[] b) {...}

}
```

As another example, another subclass might reimplement the write methods in terms of a totally different underlying class.

```
public class AnotherOutputStream extends OutputStream {

  private EfficientFileImplementation file;

  public void write(byte b)
  { file.putSingle(b); }

  public void write(byte[] b)
  { file.putMulti(b); }

}
```

In both cases, the presumption that writeArray is coded in terms of writeByte does not hold, and so the corresponding join point seems to vanish when writeArray is called. As a consequence the logging of the number of bytes that are actually written to a stream may not be correct depending on which subclass of OutputStream the aspect is actually applied to.

3 Inside and Outside

As a first step towards a solution for the problem outlined above, a programmer may write an aspect in AspectJ [1] to augment both write methods with code that increases a counter as follows:

```
aspect ByteCounter of eachobject(instanceof(OutputStream)) {

  private int counter = 0;

  // augment "writeByte"
  after(): instanceof(OutputStream) & receptions(void write(byte))
  { counter += 1; }

  // augment "writeArray"
  after(byte[] b): instanceof(OutputStream) & receptions(void write(b))
  { counter += b.length; }

}
```

This approach is correct for AnotherOutputStream, but neither for the original OutputStream, nor for OneOutputStream, since in both cases one write method is coded in terms of the other and as a consequence some bytes would be counted twice. The programmer could handle this case by introducing a flag that signals to a write method whether it is called by the other or not, and increasing the counter within the former only when this flag is not set. However, it should be the task of the aspect weaver to deal with such situations, as is argued in [2], so this is not an option.

The problem which we deal with here stems from the fact that one write method might be coded in terms of the other. In fact, the ByteCounter aspect is correct for AnotherOutputStream "only" because in this case the write methods do not depend on each other. This lack of dependence can be simulated in OutputStream and OneOutputStream when one differentiates between the external representation of a method and its internal implementation, as follows:

```
public class OutputStream {

// internal implementation

  private void internalWrite(byte b) {...}

  private void internalWrite(byte[] b) {
    for (int i = 0; i < b.length; i++) internalWrite(b[i]);
  }

// external representation

  public void write(byte b)
  { internalWrite(b); }

  public void write(byte[] b)
  { internalWrite(b); }

}

public class OneOutputStream extends OutputStream {

// internal implementation

  private void internalWrite(byte b) {
    byte[] a = new byte[1]; a[0] = b;
    internalWrite(a);
  }

  private void internalWrite(byte[] b) {...}

// external representation

  public void write(byte b)
  { internalWrite(b); }

  public void write(byte[] b)
  { internalWrite(b); }

}
```

In both examples, only the internal write methods depend on each other, but the external write methods are independent. So in both cases the ByteCounter aspect given above would be correct, since it only augments the write methods, but not the internalWrite methods.

The most important observation here is that the separation between the external representation and the internal implementation of methods could be automated by an aspect weaver by essentially just copying the bodies of the original methods to the internal ones, and replacing the original ones with bodies that call the internal methods. Imagine an extension of AspectJ that allows one to define pointcut methods that are explicitly applied to the external representation of a method, and that takes care of separating the representation and the implementation of the methods involved automatically. A programmer can then easily solve the problems outlined above.

Furthermore, this approach would also help in solving the problems of jumping aspects. A programmer could simply write an aspect that augments the external representations of add and addAll with a changed message. Since by definition these methods do not depend on each other, it is not necessary to cumulate multiple sends of changed at all, neither for the programmer nor for the aspect weaver.

A possible disadvantage of such an extension is that a programmer has to explicitly augment each method with appropriate code even if a single join point would suffice. For example, augmenting the external representation of the original OutputStream would result in the need to define two pointcut methods, one for writeByte and one for writeArray, where in fact only one would be absolutely necessary when a programmer would augment the "original" representation. On the other hand, our proposal works in all mentioned cases, and it makes the task of writing aspects easier, since the programmer does not have to have knowledge about the actual implementation of the classes to be augmented.

References

[1] The AspectJ Team, *aspectj.org – Crosscutting Objects for Better Modularity*, http://aspectj.org/, 1998-2000.

[2] Johan Brichau, Wolfgang De Meuter, Kris De Volder, *Jumping Aspects*, position paper at the workshop "Aspects and Dimensions of Concerns", ECOOP 2000, Sophia Antipolis and Cannes, France, June 2000.

[3] Sun Microsystems, Inc., *Java 2 SDK, Standard Edition Documentation, Version 1.3*, http://java.sun.com/products/jdk/1.3/docs/, 2000.