

Visibility of Join-Points in AOP and Implementation Languages

Detlef Vollmann

vollmann engineering gmbh

P.O. Box 5106

6000 Luzern 5

Switzerland

dv@vollmann.ch

Revised version January 27, 2002

Abstract

Aspects in AOP specify and implement crosscutting concerns of a software system. So by definition they are important information at more than one place in a program. But traditional AOP approaches like AspectJ or Hyper/J try to concentrate all information about an aspect in one single place. Though this approach has the advantage to allow the control of complex crosscutting concerns in its own self-contained module, it hides vital information in the original program: the join points. Join Points are the places in the original, "classic" OO source code where the separately defined aspects are "woven" into the control flow of the final programs.

This paper argues that the implementation of crosscutting concerns should be concentrated in a single module, but the information of the application of such aspects must be at both places: the join points and the aspect implementation. And such an approach can be implemented using just one, classic OO language and does not require a separate AOP language.

This paper demonstrates this approach using C++ for two well-known examples and concludes that tool support is crucial.

1 Introduction

The main goal of AOP is to localize crosscutting concerns in software systems by defining so-called *aspects*. But this is impossible by definition: either something is crosscutting, or something is local, but not both together. What can be localized are specific facets of an aspect, the most important ones are the implementation facet and the visibility facet.

Often in an application some code that belongs logically to the same abstraction is scattered across several places in the source code. From a simple perspective, aspects in AOP try to collect such code and put it together in a single place. Additionally to the code, an aspect contains the information where this code shall be applied. Those places where the code shall be applied are called *join points*. An aspect weaver applies then the aspect's code to the join points, either before or after the compiler for the base language translates the source code; but the weaver never permanently changes the source code itself.

So in AOP there are two important places for each aspect: the definition of the aspect itself and the join points. But to the programmer, only the aspect is visible; the join points are not marked in the program text. Here the implementation and visibility is local to the aspect. Nevertheless, some AspectJ tools provide additional visibility: They show the fact that an aspect is applied at the point in the source code where this aspect is applied by a mark-up in the IDE (integrated development environment).

Traditional OOP approaches go the other way around: Some design patterns specifically allow to localize the code for a specific aspect, but the definition where this code is "applied" is part of the code where it is applied and not part of the aspect. So, here you have a localized implementation of an aspect, but a scattered visibility where the aspect is used. A programmer who looks at the aspect doesn't see where it is applied. Again, tools like IDEs or the **ctag** program help to show at the aspect code where it is used.

Kiczales et al. present in [5] a distinction between development aspects and production aspects. The former are only defined to help with the development itself.

Examples are tracing, profiling, creating test cases etc. These aspects are transient by nature, they are not meant to be part of the final code (though they might exist for the whole lifetime of the software). For such development aspects, it is not a major problem if the join points are not visible.

This paper will mainly discuss production aspects. The examples in this paper are such production aspects. Those are meant to be part of the final production code that needs to be maintained. And especially for maintenance the clear visibility of join points is a major concern.

In section 2 of this paper two examples are examined with respect to the visibility of join points. Section 3 demonstrates how these two examples can be implemented using only a single OO language. Section 4 describes some related work and finally section 5 presents some conclusions and future work.

2 Visibility of Join Points

OOP and AOP share a common modularization goal: the localization of cohesive functionality. But with crosscutting concerns, there is a natural obstacle: The information about the application of the concern is necessary at several different places.

Let's look at two examples from the literature.

Hyper/J

Ossher and Tarr present as an example of a Hyper/J application [8] a personnel system with a class hierarchy `Employee`, where each subclass provides the methods `name()`, `check()`, `print()`, `position()` and `pay()`. They then move the `position()` and `pay()` methods into a separate so-called *hyperslice*. And then they build a system that does not contain this hyperslice. But that hyperslice is only removed in the binary distribution; the source code of the `Employee` hierarchy still contains the `position()` and `pay()` methods¹. Now assume that in the field there is a problem. Then the developer looks at code that doesn't exist anymore. This is

not a very comfortable maintenance scenario.

Of course Ossher and Tarr have a very good reason to work that way. They want to use the same code base for systems with and without that specific hyperslice. They don't want to change the source code for different versions and they don't want to ship code that is not necessary. We will look at a different solution to this problem in section 3.

AspectJ

Kiczales et al. present a different example for AspectJ [5]. They add an aspect `DisplayUpdating` (a simple function call) to a number of methods in several classes. The classes are part of a figure editor, and to all modifying methods a call to `Display.needsRepaint()` is added.

The definition of this aspect contains the code to add (i.e. the function call) and a reference to a so-called *pointcut*. This pointcut references the join points for this aspect. The definition of the pointcut can explicitly name the methods which designate the join points. Alternatively, the methods can be defined by lexical match using wildcards. Still another possibility is to define the methods by structural (either static or dynamic structure) properties. The result of this rich set of possibilities is that the visibility of the join points even more looses, and it is quite difficult for a developer without proper tool support to know exactly to which methods an aspect is applied.

At the join points themselves, the situation is the same as for Hyper/J: without tool support, a developer can not see from the source of a method that an aspect is applied here. If a new method is added, e.g. `moveTo()` in addition to `moveBy()`, most developers do so by "copy and modify": they copy the source code of `moveBy()` and adjust it for the new purpose. Unfortunately, the new method will not work as expected: as the new method is not part of the respective pointcut, the aspect is not applied. But without intimate knowledge of the whole system (which maintainers often don't have) the developer will most probably forget to update the pointcut accordingly.

3 Single Language Approach

As we have seen in the previous section, join points should be visible at both places: at the definition of the aspect and at the join points themselves. But this is redundant information: the same information at two different places. Of course, this redundant information should not be provided by the developer but by a supporting tool. But with such a tool, it is not important anymore where the information is provided originally. I.e. it should be possible to use just one OO language and the features it provides to localize crosscutting concerns and let the tool do the rest.

To test the single language approach, the above examples were redesigned using C++. C++ was chosen as it is a quite popular language not only providing OO features, but also providing a rich set of additional features to separate concerns.

Hyper/J Example

The personnel system of the Hyper/J example started out with an existing `Employee` class hierarchy. From that, the hyperslice `Payroll` was extracted containing the same class hierarchy with each class containing the methods `position()` and `pay()`. Such an extraction is known as the *Extract Class* refactoring [3]. In C++, the original class could look like this:

```
class Line : public Staff
{
public:
    virtual string name();
    virtual bool check();
    virtual void print(ostream &);
    virtual PositionT position();
    virtual void pay() const;

private:
    // ...
};
```

After the extraction, we have several classes:

```
template <bool payrollSlice>
class PayrollLine;

// Line class
// composes the payroll slice by inheritance
```

```
class Line
    : public Staff,
      virtual public PayrollLine<withPayroll>
{
public:
    virtual string name();
    virtual bool check();
    virtual void print(ostream &);

private:
    // ...
};

// payroll slice implementation
// Line class specialization
template <>
class PayrollLine<true>
    : virtual public PayrollStaff<true>
{
public:
    virtual PositionT position();
    virtual void pay() const;

private:
    // ...
};

// plugin when the payroll slice is not used
// empty Line class specialization
template <>
class PayrollLine<false>
    : public PayrollStaff<false>
{};
```

Such a refactoring of a whole class hierarchy is a lot of mechanic work, tedious and error-prone. So such refactorings should be done by respective tools. Unfortunately, though refactoring tools exist (e.g. as plug-in for Eclipse [4]), they cannot do such complex refactorings yet. But quite probably, such tools will emerge in the near future.

The result is functionally the same as the Hyper/J solution: The classes `Line` and `Staff` can be used as before. If the `Payroll` slice is included, the classes provide the methods `position()` and `pay()` and if the system is shipped without the `Payroll` slice, these methods are not available. Whether the `Payroll` slice is included or not can be controlled by a boolean compile-time constant. But in this solution we have a clear marker at the `Line` class that it is a join point. On the other hand, the `PayrollLine`

template has no direct marker where it is used (apart from naming). But this information is typically provided on request by current IDEs.

AspectJ Example

The figure editor example is quite a popular example in OO. The aspect `DisplayUpdating` consists of a single function call. So, the called function can be viewed as the aspect. In a solution where the aspect application is explicit in the code, the function call is just inserted into the original code. The C++ equivalent of the original code would be something like

```
void Point::setX(int X)
{
    x_ = x;
}
```

while our approach adds the update call:

```
void Point::setX(int X)
{
    x_ = x;
    display.needsRepaint();
}
```

Actually, this looks much more natural, as the requirement to refresh the display is an inherent semantic feature of this method.

But the argument of Kiczales et al. is that the extraction of the `DisplayUpdating` aspect allows for easier evolution. But as long as future changes are local to the aspect, they are local to `needsRepaint()`, and therefore they are still local. Only in the case where the signature of `needsRepaint()` changes, all the function calls must be changed accordingly. E.g. `display.needsRepaint();` becomes `display.needsRepaint(this);` in all the methods. But such changes are actually supported by some current refactoring tools.

Again, the visibility of the join points is at the join points only; there is no explicit marker at the `needsRepaint()` method where it is called. But this information is easily provided by current IDEs when required.

Discussion

The above examples show that a single language approach is feasible to implement crosscutting concerns. Rich OO languages generally provide enough features to separate concerns, even if these concerns are crosscutting by nature. And the visibility of join points is even higher without AOP languages. This provides for easier maintenance.

But applying fundamental changes to crosscutting concerns is tedious and error-prone in traditional OO languages. Here AOP tools provide better means to handle such cases. For OO, refactoring tools that alter the original code base can play the role that weaving tools play in AOP.

For development aspects, AOP languages can play the role of a general tool to provide meta-access to the code, which is very useful for quite a range of development tasks.

4 Related Work

To find OOP techniques to separate concerns is a field where a lot of research is done. Prominent in this area is all the design pattern work and more recently the refactoring work as part of the Extreme Programming (XP) discussion. In XP, design changes, even fundamental ones, are quite frequent, and therefore it is important to ease such changes, even when they relate to a lot of places scattered across the whole system.

But also the importance of the visibility of join points in AOP is raised by others. Murphy et al. conducted experiments to investigate whether AOP actually makes the development and maintenance of application easier. One of their results was that the exposition of join points is important [7].

Lippert and Lopes conducted a study on exception handling using AOP. While they found AOP in general quite useful for that purpose, they found that it would sometimes be useful to reconstruct the local effects of an aspect [6]. This is exactly what this paper suggested as refactoring tools.

In a completely different field, Alexandrescu studied the design of crosscutting concerns in C++ [1]. He used so-called *policy classes* to control crosscutting functionality at a central place. To achieve this, he used templates together with multiple inheritance to overcome what Tarr et al. called the "tyranny of the dominant decomposition"[9].

Actually, Alexandrescu's policy classes are just one option to implement crosscutting concerns in C++. Czarnecki and Eisenecker [2] provide quite a comprehensive range of different techniques for C++.

5 Conclusion and Future Work

This paper discussed the weakness of AOP with respect to the visibility of join points. As an alternative, the usage of a traditional OO language was explored. It turned out that several goals of AOP can also be accomplished without an explicit AOP language. But this approach produced other problems and suggested that proper tool support can greatly increase the usability of an OO language for AOP.

This work is at a very early stage. As the approach presented in this paper heavily builds on refactoring tools, it is necessary to provide some sample tools that prove the possibility, usability and usefulness of such tools.

An interesting question is also whether the two-language approach (a base language and a weaver language) that is currently most popular is really useful for AOP. I.e. it could possibly serve the AOP goal of better understandability better to either use an existing language that is rich enough to realize aspects or to provide a new language that clearly advocates the usage of AOP (like Smalltalk did for OOP).

It might also be interesting to study whether the so-called "tyranny of the dominant decomposition" actually exists in OOP or whether this might be a misconception based on single-inheritance OO languages.

Acknowledgements

I would like to thank Kai Böllert for his valuable feed-

back on a preliminary version of this paper.

References

- [1] A. Alexandrescu: *Modern C++ Design* Addison-Wesley 2001.
- [2] K. Czarnecki and U. Eisenecker: *Generative Programming* Addison-Wesley 2000.
- [3] M. Fowler: *Refactoring*. Addison-Wesley 1999.
- [4] <http://www.eclipse.org>.
- [5] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold: Getting started with AspectJ. *CACM* 44, 10, (Oct. 2001).
- [6] M. Lippert and C. Videra Lopes: A study on exception detection and handling using aspect-oriented programming. In *Proceedings of the 22nd International Conference on Software Engineering*, 2000.
- [7] G. Murphy, R. Walker, E. Baniassad, M. Robillard, A. Lai, and M. Kersten: Does aspect-oriented programming work? *CACM* 44, 10, (Oct. 2001).
- [8] H. Ossher and P. Tarr: Using multidimensional separation of concerns to (re)shape evolving software. *CACM* 44, 10, (Oct. 2001).
- [9] P. Tarr, H. Ossher, W. Harrison, and S. Sutton: N degrees of separation: multi-dimensional separation of concerns. In *Proceedings of the 21st International Conference on Software Engineering*, 1999.

Notes

1. Hyper/J has also a mode to actually change the source code and move the functions that belong to different hyperslices to different source files. In this mode Hyper/J can actually act as a refactoring tool as proposed in this paper. But even then the system is still a two-language system, where the main source code is in Java while the weaver code is in Hyper/J, and the Java code contains no direct information about the join points.