
HOW TO WEAVE?

Noury M. N. Bouraqadi-Saâdani & Thomas Ledoux
{Noury.Bouraqadi, Thomas.Ledoux}@emn.fr
Ecole des Mines de Nantes

AOP is an emerging paradigm that allows the separation of multiple concerns in the software development process. Weaving which stands for knitting aspects together into a coherent application is one of critical issues of the AOP technology. In this paper, we focus on approaches to achieve weaving and related issues.

A. A point of view on AOP

In order to present our point of view on AOP, we start from the basics of computing and the concept of program. A given *program* P can always be viewed as a sequence of statements that are aimed to produce some *result* R . By result, we mean any kind of outcome (a calculation, a display...). This result R is obtained through the execution of the program P . This execution is done by some platform (hardware, operating system, virtual machine...) that *interprets* the program's sequence of statements. As an example, a Java program is a sequence of byte-codes interpreted by a virtual machine. A program written in C is compiled into a chain of bits that is interpreted by the CPU. Thus, any result R of a computation depends on both a program P and an interpreter I . A different result may be obtained by changing, at least, one of the elements of the couple $\langle P, I \rangle$.

Now, let's go back to AOP with this context in mind. The definition of AOP given by Kiczales et al. [1] states that a property of an application implemented using AOP can be either represented as:

- a component if it is cleanly encapsulated in a building block of the programming language, or
- an aspect if it can not be cleanly encapsulated in any construct of the programming language. Aspects are properties that cross-cut components and tend to affect component performance and semantics.

We consider the set of application components as a unique entity, which is a program P_0 written for some interpreter I_0 . This couple $\langle P_0, I_0 \rangle$ produces some result R_0 . Aspects affect application execution in such a way to produce another result R_1 . This new result can be obtained by changing, at least, one of the elements of the couple $\langle P_0, I_0 \rangle$. The change is either a transformation of the program P_0 , or a transformation of the interpreter I_0 , or both transformations. We claim that these transformations allow describing the whole set of possible approaches to achieve aspect weaving.

B. Approaches to achieve AOP

1. Weaving through program transformation

This approach consists in transforming only the program P_0 within a couple $\langle P_0, I_0 \rangle$. The interpreter I_0 is kept unchanged. A new program P_1 is built from both the application aspects and the program P_0 . Each aspect is a set of transformation rules that refer to some join-points. Aspect weaving consists in applying these transformation rules to the initial program P_0 . This

approach has been adopted in different works [3, 4] including AspectJ [2]. Note that in these works the interpreter I_0 is implicit.

As an example, consider the following simple program P_0 (written in pseudo-code):

```
class A {
void foo(){
    //do something}
void bar(){
    //do another thing}
}
```

Now, suppose we want to log in some file all invocations of the `foo()` method. This logging is a particular aspect which refers to the join point "invocation of the `foo()` method". This aspect is defined using a transformation rule that inserts a log statement at the beginning of the `foo()` method:

```
aspect Log{
    insert 'logFile.write(currentDate)' in method foo()
}
```

Aspect weaving consists in applying the log aspect transformation rule to our program P_0 . We obtain the code below that plays the role of the new program P_1 :

```
class A {
void foo(){
    logFile.write(currentDate)
    //do something}
void bar(){
    //do another thing}
}
```

2. Weaving through interpreter transformation

This approach consists in transforming only the interpreter I_0 in a couple $\langle P_0, I_0 \rangle$. The program P_0 is kept unchanged. A new interpreter I_1 is built from both the application aspects and the interpreter I_0 . Each aspect is a set of transformation rules that refer to some join-points. Aspect weaving consists in applying these transformation rules to the initial interpreter I_0 . As a result, the interpretation semantics of program P_0 is changed according to the aspects.

As an example, consider the program P_0 of the previous section and the following simple interpreter I_0 :

```
class Interpreter{
Object invoke(Method m, Object[ ] args, Object receiver){
    //invokeMethod m with arguments args on receiver}
Object read(Field f, Object receiver){
    //return value of field f in receiver}
}
```

In order to log all method invocations in our program P_0 , we define a log aspect as a transformation rule:

```

aspect Log{
    insert 'if(m.name == "foo") logFile.write(currentDate)'in method Interpreter::invoke()
}

```

This aspect refers to the same join point "invocation of the foo() method" used in the example of section B.1. But, the reference is expressed in a different way here, since the transformation rule applies to the interpreter.

Aspect weaving consists in applying the log aspect transformation rule to our interpreter I_0 . We obtain the code below that plays the role of the new interpreter I_1 :

```

class Interpreter{
Object invoke(Method m, Object[ ] args, Object receiver){
    if(m.name == "foo") logFile.write(currentDate)
    //invokeMethod m with arguments args on receiver}
Object read(Field f, Object receiver){
    //return value of field f in receiver}
}

```

Interpreters are usually complex pieces of software, difficult to grasp and modify. Thus, an infrastructure that eases the definition of transformation rules is needed. *Reflective systems* play the role of such an infrastructure [5, 6, 7]. Such a system provides developers with an interface to adapt and extend its interpreter, while hiding implementation details.

C. Related Issues

1. Reuse

Building a particular application using AOP, requires to define aspects that are somehow linked to application components. In both examples of section B, log aspect refers explicitly to the join point "invocation of the foo() method". This explicit reference which is independent from the used approach, makes reuse of the log aspect difficult. More generally, strong coupling between aspects and application components restricts the opportunities of aspect reuse.

In order to reuse aspects in different applications, aspects should be generic. Then, transformation rules defined within aspects should not explicitly reference application components. However, weaving requires linking aspects to application components. To satisfy these two contradictory constraints, the explicit reference between aspects and components should be externalized in a new entity we name *configuration*. To illustrate this new entity, we rewrote the log aspect provided in the example of section B.1 as following:

```

aspect Log{
    insert 'logFile.write(currentDate)' in methods belonging to setOfMethods
}

config{
    setOfMethods = { foo() }
}

```

This principle of externalization of the reference between aspects and components was adopted in some works related to AOP [4][5].

2. Run-time adaptability

By adaptability, we mean the ability to replace, add, or suppress one or more aspects of some application. Adapting an application at run-time can be useful in critical applications that should be evolved without being stopped (e.g. network servers). When adaptation can be predicted before running the application, all the possible aspects and adaptation conditions can be defined prior to weaving. In this case, run-time adaptability results into a state change (possible use of the State design pattern [8]).

But, when adaptation is not predictable before running the application (e.g. upgrading some aspect), some infrastructure is needed to "re-weave" aspects at run-time. This infrastructure should not only be able to weave at run-time but it should also be able to replace (partially or totally) the application. According to the used approach, the program or the interpreter should be replaced. This ability of run-time replacement and more generally the ability of a system to act upon itself are characteristics of reflective systems. In other words, some reflective capabilities are required for such run-time adaptability.

3. Weaving non-orthogonal aspects

Two aspects are said to be non-orthogonal if they refer to the same join point. Transformation rules of these conflicting aspects apply to the same parts of the program (respectively the interpreter according to the approach). The precedence of conflicting transformation rules is usually important. A change of the precedence may lead to different results.

In order to obtain a precedence that solves the conflicts automatically on weaving-time, some works [4,7] suggested to extend aspect definitions with extra information. This information is application independent, and describes partially the effect of transformation rules (code insertion/replacement, used variables...). Thus, the precedence that avoids conflicts between aspects can be deduced automatically. However, some cases are non-decidable, then developers still have to define the precedence manually.

D. Conclusion

Weaving can be viewed as a transformation of a couple $\langle P, I \rangle$ where P is a program written for some interpreter I. Thus, aspects can be seen as sets of transformation rules that apply either to the program P, or to the interpreter I, or to both of them. These transformation rules must be application independent in order to make aspects reusable. As a consequence of this reusability, the weaving process is extended with an extra step where transformation rules are linked to the application.

E. Bibliography

- [1] Kiczales G. et al. "Aspect-Oriented Programming". In proceedings of ECOOP'97. LNCS 1241, Springer-Verlag, 1997.
- [2] AspectJ web: www.aspectj.org
- [3] HyperJ web: www.research.ibm.com/hyperspace/HyperJ/HyperJ.htm
- [4] Demeter and Adaptive Programming web: www.ccs.neu.edu/home/lieber/demeter.html
- [5] Proceedings of Reflection'99. Cointe P. Editor. LNCS 1616, Springer-Verlag, 1999.
- [6] Kiczales G. "Beyond the Black Box: Open Implementation". IEEE Software, vol. 13, n° 1, 1996.
- [7] MetaclassTalk web: www.emn.fr/bouraqadi/MetaclassTalk
- [8] Gamma E. et al. "Design Patterns". Addison Wesley Publisher, 1995.