# On Weaving Aspects

Kai Böllert
Heintzestr. 17
24143 Kiel, Germany
kaib@acm.org

## 1 Introduction

Object-oriented software systems that are developed using aspect-oriented programming techniques consist of classes and aspects. Classes implement the primary functionality of an application, for example, managing stocks or calculating insurance rates. Aspects, on the other hand, capture technical concerns like persistence, failure handling, communication, or process synchronization. They are written in special aspect description languages [5].

Common to all aspect languages is that they cannot be processed by today's compilers for object-oriented programming languages. Until that changes, aspects have to be merged with classes before a compiler can take over to produce the executable program. The process of merging is called weaving; the tool required is named Aspect Weaver.

There are two ways in which classes and aspects can be woven: static or dynamic. This paper examines both weaving techniques in more detail. Section 2 works out the advantages and disadvantages of static weaving and argues why dynamic weaving is preferable for some aspects. Section 3 presents an Aspect Weaver that weaves aspects dynamically. The paper concludes with an evaluation of this Weaver.

## 2 Static Weaving

Aspects describe behavioral additions to objects. They reference the classes of those objects and define at which join points additions should be made. Join points are, for example, methods or variable assignments.

Static weaving means to modify the source code of a class by inserting aspect-specific statements at join points. For instance, weaving a persistence aspect inserts a database update statement after every assignment to an instance variable. In other words: aspect code is inlined into classes. The result is highly optimized woven code, whose execution speed is comparable to that of code written without using aspects. Hence, static weaving prevents that the additional abstraction level introduced by aspect-oriented programming causes a negative impact on a program's performance. The Aspect Weaver delivered with AspectJ [1] is an example for a static Weaver.

However, static weaving makes it difficult to later identify aspect-specific statements in woven code. As a consequence, adapting or replacing aspects dynamically during runtime can be time consuming or not possible at all, even if the programming language used supports program manipulation through its reflection interface. Although most aspects do not need this flexibility, there are aspects that will benefit from it. For instance, a load balancing aspect could replace the load distribution strategy woven before with a better one depending on the current load of the managed servers [6]. Another example is a tracing aspect. Suited to a particular malfunction that occurred in a deployed software system, it could be woven and ran without having to restart the software [4, Chapter 4].

## 3 Building a Dynamic Aspect Weaver

An important requirement for dynamic weaving is the explicit existence of aspects both at weave-time and at runtime [6]. Therefore, aspects and woven structures need to be reified as objects and must

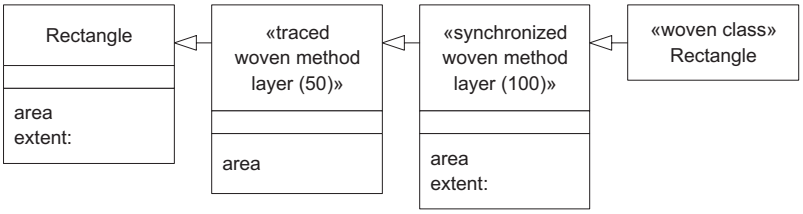


Figure 1: Woven structure after weaving two aspects into class Rectangle

be kept in the executable program. Provided a reflection interface for aspects, the Weaver is capable of adding, adapting, and removing aspects dynamically, if desired during runtime.

In the following the AOP/ST Weaver is presented. AOP/ST [2] adds aspect-oriented programming extensions to VisualWorks/Smalltalk.

### 3.1 How the Weaver Works

The AOP/ST Weaver does not modify the source code of classes while weaving aspects. Instead, inheritance is used to add aspect-specific code to classes.

Figure 1 depicts the woven structure that results from weaving two aspects, process synchronization and tracing, into the class Rectangle. Each aspect is represented by its own direct or indirect subclass of Rectangle. Methods affected by aspects are overriden in the subclasses with *woven methods*. A woven method wraps the invocation of the inherited implementation with aspect-specific statements. The empty *woven class* terminates the subclass chain.

So far the subclasses generated by the Aspect Weaver do not have any effect on program execution. If a message is sent to a Rectangle object, the method lookup will search for the corresponding method in the object's class, Rectangle, and executes the implementation found there. To bring woven code into effect, the Weaver changes the class of all current Rectangle objects to the woven class. It also installs a mechanism, so that the same will happen for Rectangle objects created in future. From now on the method lookup starts its search in the woven class rather than in the Rectangle class. Hence, woven methods placed along the subclass chain are executed, and Rectangle objects show aspect-specific behavior in addition to their functional behavior.

The order in which the Aspect Weaver composes subclasses can be specified on a per aspect basis in the form of an ordinal number, which is called the precedence of an aspect. In the example, process synchronization aspects have precedence 100, whereas tracing aspects have 50. Subclasses are ordered with respect to this precedence, so statements from aspects with high precedence are executed first.

### 3.2 A Reflection Interface for Aspects

Every aspect needs a concrete subclass of Aspect and WovenMethodBuilder. Instances of the latter are responsible for building woven methods according to parameterized Aspect objects. Weaving such Aspect objects into a class creates a woven structure. This consists of the woven class, an instance of WovenClass, and one or more subclasses, which are instances of WovenAspectBehavior.

The interface of Smalltalk classes has been extended with a number of introspection and intercession messages like aspects, wovenClass, weave:into:, and unweave:from:. Besides the Weaver, class browsers use this interface to indicate which of the viewed classes and methods are affected by which aspects.

A more in-depth discussion of the AOP/ST Weaver's architecture and the reflection interface is given in [4, Chapter 5].

## 4 Evaluation

The following paragraphs evaluate the AOP/ST Weaver and the woven structures it produces from different perspectives.

| Join points | Aspects | Supported |
|---|---|---|
| Sending a message | communication | no |
| Receiving a message | process synchronization, assertion checking, tracing | yes |
| Returning a result | tracing | yes |
| Catching an exception | failure handling | yes |
| Accessing/assigning an instance variable | persistence, replication | no (workaround: use getter/setter methods) |
| Loop fusion | performance optimization | no |

Table 1: Join points supported by the AOP/ST Weaver

**Dynamic weaving.** Classes and woven aspects are kept separate from each other. Hence, using Smalltalk's reflection interface, the Weaver can dynamically add, adapt, or remove aspects at any time. For example, if an aspect has been modified, the Weaver updates its woven methods. If an aspect is to be removed completely, the Weaver removes the respective subclass.

**Join points.** Different aspects need to be woven at different join points. While the presented woven structure is suitable for dynamic weaving, it fails to support some join points and, thus, restricts the aspects that can be woven by the Weaver. Table 1 summarizes supported join points and aspects.

**Relationships between aspects.** Aspects woven into the same class relate to each other in three possible ways:

1. *Prerequisite.* An aspect may require others to be woven first.
2. *Composition order.* Aspects need to be woven in a specific order.
3. *Composition validation.* Some aspect combinations may not be allowed.

The aspect precedence mentioned in Section 3.1 is applicable to order aspects but not to define prerequisites or to validate compositions. In [3] Batory and Geraci describe how GenVoca generators handle such relationships. Further investigations will show if their approach could be integrated into the AOP/ST Weaver.

**Inheritance anomalies.** Subclasses do not inherit the aspects that are woven into their superclass, because the woven code is placed on a separate inheritance branch. Furthermore, no class in an application is supposed to be a subclass of one of the subclasses that the Weaver generates. Hence, the weaving does not lead to inheritance anomalies.

**Iterative development.** The Weaver is able to incrementally weave aspects as they are being developed. This shortens the turnaround time of the edit-weave-test aspect cycle. Debugging aspects is subjectively easy, because they remain clearly separate from each other in woven structures.

**Memory and performance.** Two minor disadvantages are the extra amount of memory needed for the generated subclasses and a slightly reduced performance of the method lookup.

# 5 Conclusion

Aspect Weavers should support static as well as dynamic weaving. Depending on the aspect to be woven, a Weaver switches between the two modes and, thus, combines the advantages of both techniques. While developing aspect-oriented applications, the dynamic mode is preferable because it facilitates incremental weaving and makes debugging easier. Upon deployment, aspects that do not need to be adapted at runtime should be woven statically for performance reasons.

## References


[1] AspectJ Home Page. `http://www.parc.xerox.com/aop/aspectj/`.

[2] AOP/ST Home Page. `http://www.germany.net/teilnehmer/101,199268/`.

[3] Don Batory and Bart J. Geraci. Composition validation and subjectivity in GenVoca generators. *IEEE Transactions on Software Engineering*, 23(2):67–82, February 1997.

[4] Kai Böllert. *Aspect-Oriented Programming, Case Study: System Management Application.* Graduation thesis, Fachhochschule Flensburg, 1998.

[5] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. Technical Report SPL97-008 P9710042, Xerox PARC, Palo Alto, CA, February 1997.

[6] Frank Matthijs, Wouter Joosen, Bart Vanhaute, Bert Robben, and Pierre Verbaeten. Aspects should not die. Position paper at the ECOOP '97 workshop on Aspect-Oriented Programming, 1997.