Aspect-Oriented Programming in Java

Markus Voelter, voelter@acm.org

Object oriented programming has become mainstream over the last years, having almost completely replaced the procedural approach. One of the biggest advantages of object orientation is that a software system can be seen as being built of a collection of discrete classes. Each of these classes has a well defined task, its responsibilities are clearly defined. In an OO application, those classes collaborate to achieve the application's overall goal. However, there are parts of a system that cannot be viewed as being the responsibility of only one class, they cross-cut the complete system and affect parts of many classes. Examples might be locking in a distributed application, exception handling, or logging method calls. Of course, the code that handles these parts can be added to each class separately, but that would violate the principle that each class has well-defined responsibilities. This is where AOP comes into play: AOP defines a new program construct, called an *aspect*, which is used to capture cross-cutting aspects of a software system in separate program entities. The application classes keep their well-defined resonsibilities. Additionally, each aspect captures cross-cutting behaviour.

This article is divided into three parts: The first part explaines the concepts of AOP, the second introduces AspectJ(TM), an implementation of the AOP concepts in Java, and part three compares the AOP approach to metalevel programming.

AOP Basics

Let's introduce AOP with the help of an example. Imagine an application that works concurrently on shared data. The shared data may be encapsulated in a **`Data`** object (an instance of class **`Data`**). In this application, there are multiple objects of different classes working on a single **`Data`** object, whereas only one of these objects may have access to the shared data at a time. To achieve the desired behaviour, some kind of locking must be introduced. That means, that whenever one of these objects wants to access the data, the **`Data`** object must be locked (and unlocked after the object has finished using it). The traditional approach is to introduce an (abstract) base class, from which all "worker" classes inherit. This class defines a method **`lock()`** and a method **`unlock()`** which must be called before and after the actual work is done (semaphores, basically). This approach has the following drawbacks:

- Every method that works on the data has to take care of locking. The code is cluttered with statements related to locking.
- In a single inheritance world, it is not always possible to let all worker classes inherit from a common base class, because the one and only inheritance link may already be consumed by another concept. This is especially true if the locking features must be introduced into a class hierarchy after the hierarchy has been designed, possibly by another programmer (e.g. the developer of a class library).
- Reusability is compromised: The worker classes may be reused in another context where they don't need locking (or where they have to use another locking scheme). By putting the locking code into the worker classes, the classes are tied to the locking approach used in this specific application.

The concept of locking in our example application can be described with the following properties:

- It is not the primary job of the worker classes
- The locking scheme is independent of the worker's primary job
- locking cross-cuts the system, i.e. many classes, and probably many methods of these classes, are affected by locking.

To handle this kind of problems, aspect-oriented programming proposes an alternative approach: A new program construct should be defined that takes care of cross-cutting aspects of a system. Not surprisingly, this new program construct is called an *aspect.*

In our example application, the aspect **`Lock`** would have the following responsibilities:

- provide the neccessary features to lock and unlock objects to the classes that have to be locked/unlocked (in our example add `lock()` and `unlock()` to the `Data` class)
- ensure that all methods that modify the `Data` object call `lock()` before their work and `unlock()` when they

have finished (in our example the worker classes).

What else can aspects do? For example, if a software system needs to log certain method calls (e.g. constructors to track object creation), aspects can help. Here, too, an additional method (**`log()`**) is necessary, and this method needs to be called at certain locations in the code. Certainly nobody will waste the inheritance link of completely different classes just to introduce a **`log()`** method into the class hierarchy. Here, again, AOP can help by creating an aspect which provides a **`log()`** method to the classes that need one, and by calling this method wherever it is required. A third, and quite interesting example might be exception handling. An aspect could define **`catch()`** clauses for methods of several classes, thereby enabling consistent exception handling throughout the application.

Two questions arise when looking at aspects: The first is "Are aspects really necessary?" Of course they are not necessary in the sense that a given problem cannot be solved without aspects. But they provide a new, higher level abstraction that might make it easier to design and understand software systems. As software systems become larger and larger, this "understanding" is becoming a major problem. So aspects might prove to be a valuable tool.

A second question could be "But don't aspects break the encapsulation of objects?" Yes somehow they do. But they do so in a controlled way. Aspects have access to the private part of the objects they are associted with. But this doesn't compromise the encapsulation between objects of different classes.

AspectJ

AOP is a concept and as such it is not bound to a certain programming language or programming paradigm. It can help with the shortcomings of all languages that use single, hierarchical decomposition. This may be procedural, object oriented, or functional. AOP has been implemented in different languages, among them Smalltalk and Java. The Java implementation of AOP is called *AspectJ (TM)* and has been created at Xerox PARC.

Like any other aspect-oriented compiler, the AspectJ compiler includes a weaving phase that unambiguously resolves the cross-cutting between classes and aspects. AspectJ implements this additional phase by first weaving aspects with regular code and then calling the standard Java compiler.

Implementing the example with AspectJ

I would like to show part of the implementation of the locking example described above. The class that represents the data on which the system works is called **`Data`**. It has a couple of instance variables (some of them might be static) and some methods which work on the instance variables. Then, there is a class **`Worker`** which performs some kind of modification on the data, e.g. these modifications could be time-triggered. Listing 11 shows this **`Worker`** class. The boolean return values in the perform-methods indicate whether the modification has been carried out successfully or not.

```
LISTING 1
--------------------------------------------------------------------------------
01 aspect DataLog {
02      advise * Worker.performActionA(..), * Worker.performActionB(..) {
03          static after {
04              if ( thisResult == true )
05                System.out.println( "Executed "+thisMethodName+
06                "successfully!" );
07              else
08                System.out.println( "Error Executing "+
09                thisMethodName );
10          }
11      }
12 }
```

Listing 1 defines an aspect called **`DataLog`**. It includes one so-called advise cross-cut. The advise affects all methods that fit the signature **`* performAction(..)`**, i.e. they have to begin with performAction, they can have arbitrary parameters and any return type (line 02). After the methods have been executed, the code in the **after** section in the **DataLog** advice is executed. So after every invocation of **performActionA()** or **performActionB()** the system prints a message that says whether the method has been executed successfully or not. Within the aspect code we can use a couple of special variables, e.g. **thisResult**, that contains the result of the advised method or **thisMethodName**, which contains the name of the method that is currently executed. In addition to **after** advises, it is also possible to add **before** advises. With the help of the code weaver, AspectJ automatically introduces the necessary code into the corresponding classes.

Now lets turn to locking: With only one worker working on **sharedDataInstance**, there is no problem, because locking is not necessary if only one worker works on **sharedDataInstance**. Now imagine **AnotherWorker**, a class completely unrelated to **Worker** (i.e. no common base class), which performs other modifications on the same **sharedDataInstance**. A mechanism to lock the **Data** instance must be introduced. This is necessary to avoid both workers modifying the **sharedDataInstance** at the same time, thereby creating an inconsitent state or reading wrong data. Once again, aspects provide a very elegant way to solve this problem. All the code that is necessary for locking is encapsulated in an aspect.

```
LISTING 2
-------------------------------------------------------------------------------
01 aspect Lock {
02
03     Data sharedDataInstance;
04     Lock( Data d ) {
05         sharedDataInstance = d;
06     }
07
08     introduce Lock Data.lock;
09
10     advise Data() {
11         static after {
12             thisObject.lock = new Lock( thisObject );
13         }
14     }
15
17 }
```

Listing 2 defines a new aspect called **Lock**. In line 08 the new variable **lock** is introduced into the **Data** class with the help of a so-called **introduce** cross-cut. Then follows an **advise** cross-cut: In lines 10 through 14 the constructor of the **Data** class is modified, so that after each invocation the code within the advise is executed. This creates a new **Lock** aspect for each **Data** object created. As you can see, an aspect can have its own state (**sharedDataInstance**). The next step is to advise the classes that work on the **Data** instance (**Worker**, **AnotherWorker**). For this purpose, the **Lock** aspect has to be extended as shown in listing 3.

```
LISTING 3
-------------------------------------------------------------------------------
15     boolean locked = false;
16
17     advise Worker.perform*(..), AnotherWorker.perform*(..) {
18         before {
19             if ( thisObject.sharedDataInstace.lock.locked ) // enqueue, wait
20             thisObject.sharedDataInstace.lock.locked = true;
21         }
22         after {
23             thisObject.sharedDataInstace.lock.locked = false;
24         }
25     }
26 }
```

This introduces the variable **locked** into the **Lock** aspect. Its purpose is to remember whether the associated **Data** object is locked or not. Once again, the worker classes are modified. Before they work on the **Data** instance, they lock the **Lock** object (line 18 through 20), after they are done, they unlock it (21-23). An alternative implementation would directly "introduce" the locked variable into the associated **Data** object.

A third example takes care of error handling (listing 4).

```
LISTING 4
-------------------------------------------------------------------------------
25     advise Worker.perform*(..), AnotherWorker.perform*(..) {
26         static catch ( Exception ex ) { ex.printStackTrace(); }
27         static finally { thisObject.sharedDataInstace.lock.locked = false; }
28     }
29 }
```

This piece of code introduces two new advises, both for the various perform methods. The advise in line 26 has the consequence, that each exception thrown within the perform methods is printed, line 27 ensures that the lock is released under all circumstances.

The above examples have shown that AOP provides interesting new concepts for object oriented development. The next section shows the relation between AOP and metalevel programming, a concept that has had significant influence on the development of the AOP paradigm.

AOP and metalevel programming

AOP has a lot of things in common with *metalevel programming*. Both capture cross-cutting aspects of a software system in clean, controlled ways. One of the most fundamental properties of metalevel programming is that the programmer has access to the structures that represent a program, i.e. a program written in a specific language is represented at runtime in this very same language. The most popular language that implements metalevel programming concepts is CLOS, the Common Lisp Object System [CLOS]. Its implementations are based on the so-called Metaobject Protocol (MOP). The MOP can be seen as a standard interface to the CLOS interpreter [MOP]. With the help of the MOP it is possible to modify the behaviour of the interpreter in a controlled way. The following sections show some features of MOP in pseudo-Java syntax and show how they are related to AOP.

before and after methods

CLOS provides a feature called method combination. For every method it is possible to define a method that is executed immediately before the primary method is executed (called the method's *before-method*), and a method that is executed after the primary method (the *after-method*). These methods can also be defined in subclasses. An example is given in listing 5.

```
LISTING 5
-------------------------------------------------------------------------------
01 class Shape {
02     public void paint() {
03         // paint shape
04     }
05 }
06
07 class ColoredShape extends Shape {
08     public before void paint() {
09         // set brush color
10     }
11 }
12
13 class Test {
14     public void run() {
15         Shape shape = new Shape();
16         shape.paint();
17         Shape shape2 = new ColoredShape();
18         shape2.paint();
19     }
20 }
```

In line 16 the **`Test`** class calls the **`paint()`** method of a **`Shape`** object. The effect is, that **`Shape.paint`**

() is called. In line 18, when **shape2.paint()** is called, the before method of the **ColoredShape** class is invoked before the superclass's **paint()** method is called. It is interesting to see that an after- method does not influence the primary method's return value. If a class has multiple levels of parents, the invocation semantics of a method are as follows:

- execute all before-methods of the class and of all parent classes
- execute the primary method
- execute all after-methods of the class and of all parent classes

Because of the given execution order, before-methods are often used to do error checking (precondition validation) or to do preparative work, and after- methods can be used to clear up the environment (and to ensure postconditions). AOP introduces the very same features: They are called before advises and after advises.

Metaclasses

Central to CLOS' metaobject protocol is the concept of *metaclasses*. A metaclass is the class of a class. That means that a class defined by the programmer is an instance of another class: the class's metaclass. This metaclass is responsible for implementing the class's protocol, e.g. the the method call mechanism, the object creation process, etc. Each class in a system has its own metaclass. Metaclasses can be subclassed to create custom behaviour just like any other class.

```
LISTING 6
-------------------------------------------------------------------------------
class Test extends Object {
     public void doSomething() {
         // do something
     }
}
```

Let's look at the example in listing 6, the **Test** class. It features a method called **doSomething()**. The metaclass for this class is **StdMetaClass**, because no other metaclass is specified in the class declaration. Among other methods, **Metaclass** (and thus its subclass **StdMetaclass**) has a method **invokeMethod()**, which is called by the interpreter whenever a method of an object is invoked by a program. The **invokeMethod()** method is responsible for implementing the semantics of method invocation, e.g. searching the superclasses if no implementation of the called method is found in the class itself. Suppose you want to log calls to all methods of certain classes. The proposed way to do this with the help of the MOP would be the following: Create a new metaclass which extends **StdMetaclass** and override the **invokeMethod()** method to display a log message. Listing 7 shows this.

```
LISTING 7
-------------------------------------------------------------------------------
public class LoggingClass extends StdMetaclass {
     public void invokeMethod( Object dest, String name, Object[] params ) {
         System.out.println( name+" called on "+dest+" with "+params );
         super.invokeMethod( dest, name, params );
     }
}
```

Every class, that wants to have its method calls logged must now use **LoggingClass** as its metaclass (listing 8). An alternative implementation would not override the **invokeMethod()** method but add a before method to **invokeMethod()** that displays the log message.

```
LISTING 8
-------------------------------------------------------------------------------
public class LogTest extends Object metaclass LoggingClass {
     public void doSomething() {
         // do something
     }
}
```

Whenever **doSomething()** is called, a log message will be displayed on stdout. This process is illustrated in illustration 1. In AOP, it is possible to achieve this behaviour by creating an aspect like the one given in listing 9.

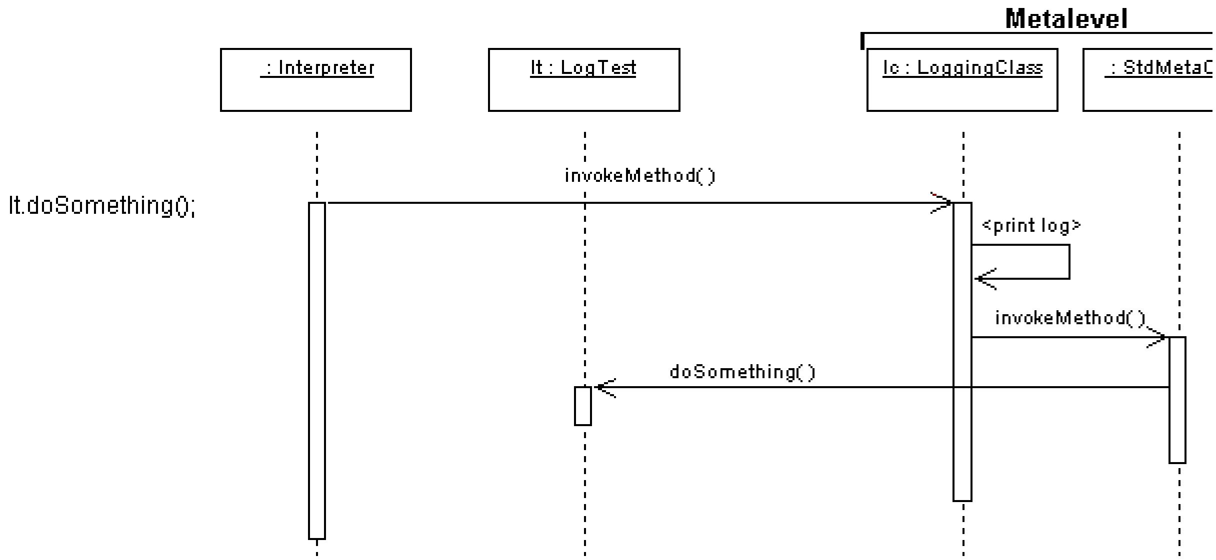

Illustration 1

```
LISTING 9
--------------------------------------------------------------------------------
aspect Log {
        // list of all classes that want their method calls logged
    advise Class1.*(..), Class2.*(..), ... {
        static before {
            // display message
        }
    }
}
```

Another example: We can use the MOP to implement classes that count how many instances have been created of them. For this purpose, we have to introduce a new metaclass, **CountingClass** (listing 10).

```
LISTING 10
--------------------------------------------------------------------------------
01 public class CountingClass extends StdMetaclass {
02     private int instanceCount = 0;
03     public after Object createInstance( Metaclass object ) {
04         instanceCount++;
05     }
06 }
07
08 public class CounterTest metaclass CountingClass {
09 }
```

The new metaclass (lines 01-06) adds one attribute, the integer **instanceCount** which is used to store the number of instances. This works the following way: Whenever a new class which specifies **CountingClass** as its metaclass is defined (e.g. CounterTest), this definition creates a new metaobject of type **CountingClass**, initializing its **instanceCount** member to 0. When a new object of type **CounterTest** is created (**CouterTest ct = new CounterTest()**) the interpreter calls the **CountingClass**'s **createInstance()** method. Because this method is not overridden in **CountingClass**, the **StdMetaclass**'s **createInstance()** method is executed and an instance of the **CounterTest** class is created.

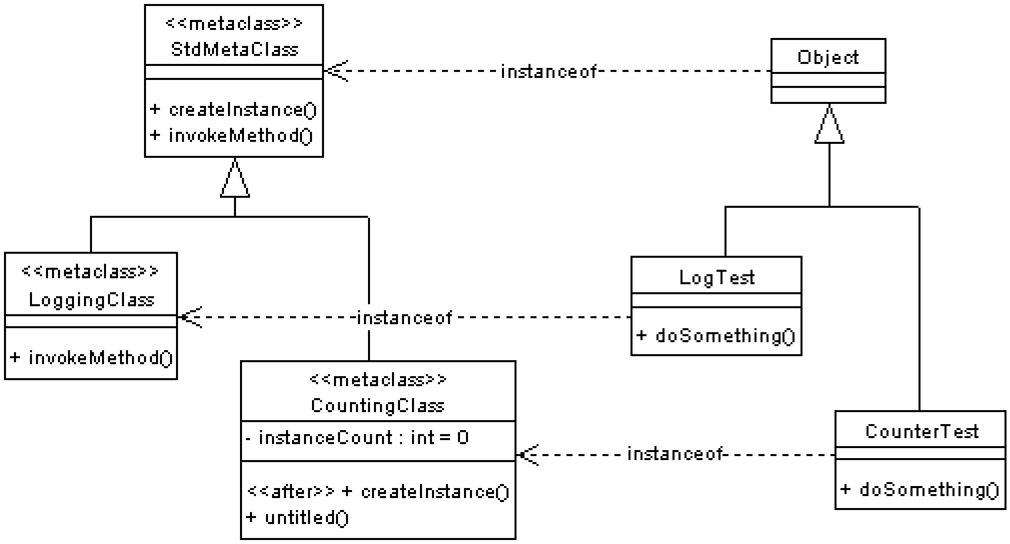

Illustration 2

However, having specified an after method for **`createInstance()`** in **`CountingClass`**, this after method is now executed in addition to the **`createInstance()`** method itself and the counter is incremented by one. Illustration 2 shows the associations between the classes and the metaclasses and illustration 3 shows the instance creation process. This behaviour can be achieved in AOP by creating an aspect, which contains after-advises for the constructors of all classes that need to implement instance counting.

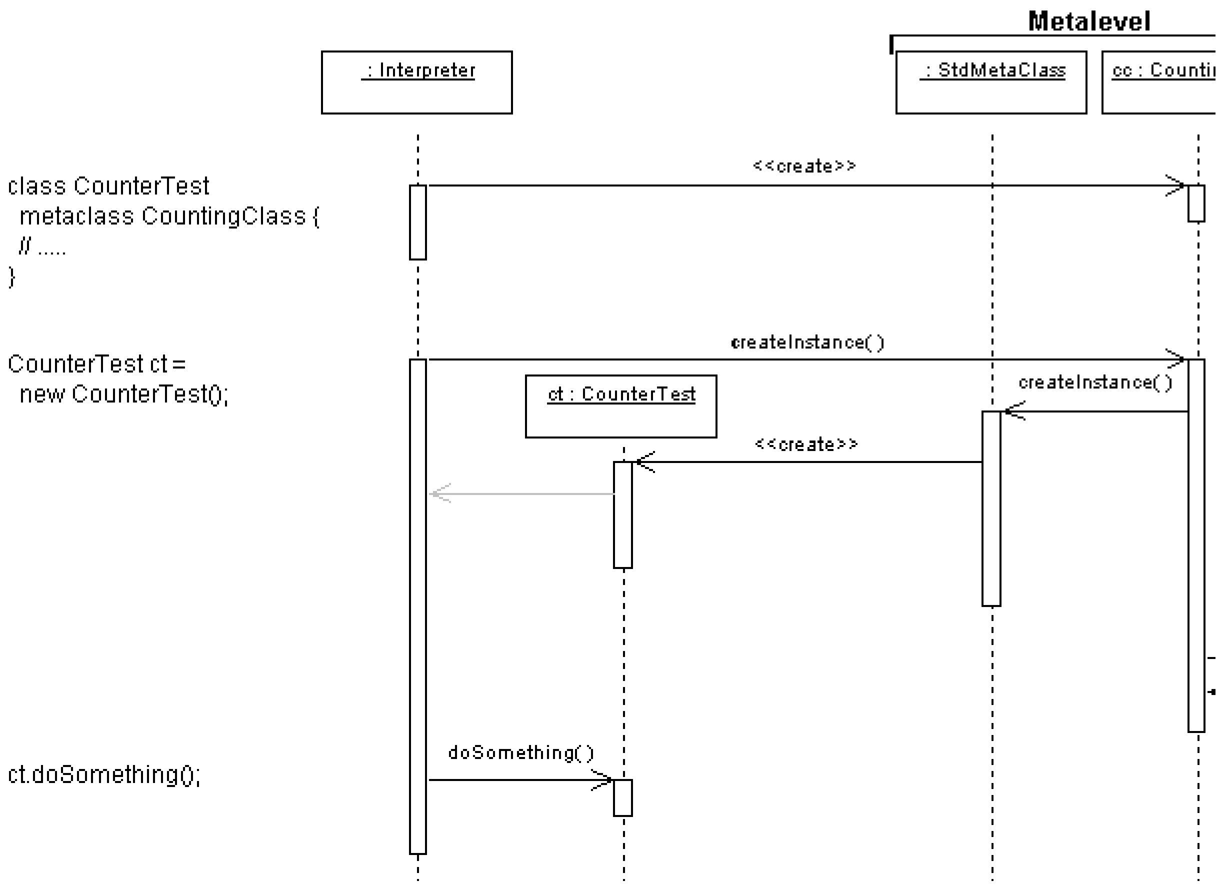

Illustration 3

Summary

AOP provides many interesting new concepts, many of them are borrowed from metalevel programming. However, AOP is easier to understand than metalevel programming and will hopefully find a broader audience. As of today, there are no industry compilers that implement AOP. But the concepts of identifying cross cutting behaviour and putting this code into separate entities can help in decomposing an application. Just being mentally aware of the distinction between a class's actual task and the additional responsiblities like locking or error handling can help in understanding the structure of an application. Besides, aspect-like concepts can be implemented by hand in conventional languages. For those wo want to experiment with "real AOP": The current beta version of AspectJ as well as information and tutorials on AOP are available from the AOP Homepage at the Xerox Palo Alto Research Center [AOP].

```
LISTING 11
-------------------------------------------------------------------------------
public class Worker extends Thread {
    Data sharedDataInstance;
    public boolean performAlgorithmA() {
        // ... do something with sharedDataInstance
        // returns true when action was executed
        // successfully, false otherwise.
    }
    public boolean performAlgorithmB() {
        // ... do something else with sharedDataInstance
    }
    public void run() {
        // schedule calls of
        // performAlgorithmA and performAlgorithmB
        // according to some external
        // cicumstances
```

```
    }
}

LISTING 12
-------------------------------------------------------------------------------
public class AnotherWorker extends Thread {
    Data sharedDataInstance;
    public boolean performA() {
        // ... do something with sharedDataInstance
    }
    public void run() {
        // schedule calls of performA
    }
}
```

Reference

[AOP] *AOP Homepage of Xerox PARC*, http://www.parc.xerox.com/aop

[AJ] *Aspect J Homepage*, http://www.parc.xerox.com/spl/projects/aop/aspectj/

[MOP] Kiczales, Rivieres, Bobrow: *The Art of the Metaobject Protocol*, MIT Press 1995, ISBN 0-262-61074-4

[CLOS] Koschmann, *The Common LISP Companion*, ISBN 0-471-50308-8