Handling Cross-Cutting Conerns: AOP and beyond

Markus Völter, voelter@acm.org, www.voelter.de

With the rise of aspect-oriented programming techniques and tools the issue of cross-cutting concerns and their efficient handling is becoming more and more prominent. AspectJ is the most well-known example of aspect-oriented programming. I have come across many discussions about what constitutes "real" AOP and what doesn't. While I certainly don't have the goal of settling this discussion once and for all, but I want to widen the scope of the discussion a little bit: AOP is basically a way to efficiently handle cross-cutting concerns on language-level. However, there are ways to handle cross-cutting concerns differently, on other levels, such as design or architecture. While I don't want to formally define AO-Design or AO-Architecture, I want to provide a couple of examples how cross-cutting concerns can be handled without AO language tools.

Aspect Orientation defined

The following definition is taken from aosd.net [AOSD], a great website which serves as a really good starting point to get an overview over aspect-oriented software development:

> *Aspect-oriented software development is a new technology for separation of concerns (SOC) in software development. The techniques of AOSD make it possible to modularize crosscutting aspects of a system.*

Cross-cutting concerns are all those things in a software system that cannot be localized, or modularized to one place in the system with traditional (OO, procedural) software development paradigms. Note that the definition above is not about aspect-oriented *programming* (AOP), it is about aspect-oriented *software development*. As we all know, software development is not just programming, it also includes analysis, design, architecture and modelling. Consequently, there is also AO-analysis, AO-design, AO-architecture and AO-modelling. The most important AO tool or technique, however, is an AO mindset. A developer needs to understand the problems with cross-cutting concerns and know ways how to handle (i.e. localize) them with the tools he has at hand, or with additional tools, if necessary. This article wants to provide some food for thought.

Handling Cross-Cutting Concerns with OO techniques

Let us first of all look at handling cross-cutting concerns with the means of traditional OO design and implementation techniques. Remember that AO is primarily a mindset – it can be implemented, more or less comfortably, with traditional OO programming techniques and languages. Let us look at some features that can easily be implemented with AspectJ,

and see how this can be "simulated" with OO languages. For example: do something whenever an instance of a class is created. How can we "inject" code into situations where class instances are created?

The simplest way is to use the *Factory* pattern [GoF]. Instead of simply calling a class constructor from client code, you can ask a factory to create the object for you. Consider the following example[1]:

```
public class Vehicle {
…
}

class VehicleFactory {
 public Vehicle createVehicle() {
        return …
 }
}
```

To create an instance of the *Vehicle* class, a client simply can call the *createVehicle* operation:

```
VehicleFactory f = new VehicleFactory();
Vehicle v = f.createVehicle();
```

Now, how can this help with handling cross-cutting concerns? By plugging in a *Strategy* [GoF].

```
public class BaseFactory {
 public BaseFactory parent;
 public Action beforeCreation = null;
 public void before( Class cls ) {
        if ( beforeCreation != null )
               beforeCreation.execute(cls);
        if ( parent != null ) parent.before( cls );
 }
 public Object createInstance( Class cls ) {
        before( cls );
        Object o = cls.newInstance();
        // maybe call an after( cls ) method, too
        return o;
 }
}
```

This class *BaseFactory* serves as the base for factories. Such a factory has an *Action* that is executed before the instance is created (obviously, you can easily add a method *after* that can be used to intercept the situation *after* the instance is created). For specific types, this class can be specialized, as follows:

---

[1] Note that we are not using setter/getter operations, for reason sof brevity we use public attributes. Yes, we know this is bad practice, don't do it!

```
class VehicleFactory extends BaseFactory {
 public Vehicle createVehicle() {
       return (Vehicle)createInstance( Vehicle.class );
 }
}
```

In order to print a message whenever a vehicle is created, the following code suffices:

```
f.beforeAction = new Action() {
 public void execute( Class cls ) {
       System.out.println( "Instantiating "+cls );
  }
};
```

To actually handle cross-cutting concerns (such as: printing a message for *all* class instantiations, not just for vehicles) you can use a *chain of responsibility* [GoF]. You can define an instance of a *BaseFactory* that has a *beforeAction* assigned that provides default behaviour. The *VehicleFactory*, as well as the other factories in the system, has this *BaseFactory* instance assigned as a parent; it delegates to this instance:

```
BaseFactory defaultFactory = new BaseFactory();
defaultFactory.beforeAction = someAction;
VehicleFactory vf = new VehicleFactory();
vf.parent = defaultFactory;
SomeOtherFactory of = new SomeOtherFactory();
of.parent = defaultFactory;
```

In order to intercept method invocations on objects as opposed to constructors, you would typically use a *Proxy* object [GoF] that has the same interface as the target object, but internally uses the same *Action*-based approach. After executing the *before* action it would delegate execution of the method to the "real object", an instance of the implementation class.

Note that these two approaches fit together nicely since the factory can be used to insert a proxy between the client and the real object:

```
class VehicleFactory extends BaseFactory {
 public Vehicle createVehicle() {
      Vehicle v = (Vehicle)createInstance( Vehicle.class );
      VehicleAOProxy vp = new VehicleAOProxy( v );
              return vp;
 }
}
```

Obviously, writing these factories and proxies by hand is annoying and error prone. It is much more advantageous to use code generation, suitable tools are available [Voelter-CodeGen]. There is another, more serious problem: Using this approach to handling cross-cutting concerns requires invasive changes to the software system. Client code actually has to *use* the factories in order to benefit from the concerns handling. You have to design the client code with these factories or proxies in mind. It is not possible to introduce these aspects into the system *after the fact*, something you can easily do with tools such as

AspectJ. There are other reasons why you might want to use AspectJ instead of a hand-crafted solution: AspectJ provides you with information about the join point. For example, you can query the *thisJoinPoint* variable for the client who called the constructor or method. In the hand-crafted soution, you would have to pass this data to the operation manually:

```
public class SomeOtherClass {
 public void someOperation() {
       Vehicle v = f.createVehicle( this, "someOperation" );
 }
}
```

This might be acceptable for the factory, but it is certainly unacceptable for method invocations on proxies, because each operation would require these parameters in its signature. If you need this kind of data for your system on language level, use AspectJ.

AOP for handling cross-cutting concerns on language level

The most well-known realization of aspect-oriented software development is aspect-oriented programming, where new language constructs are introduced to handle cross-cutting concerns on language level. The most well-known implementations are AspectJ for Java, AspectC++ for C++ as well as several implementations for other languages (see [AOSD]). AspectJ is implemented as a so-called weaver, a kind of precompiler that merges traditional OO code (the so-called base program) with the aspect code. It produces another piece of code that contains intermingled OO and AO code. This process can happen on the basis of source code or on the basis of Java bytecode. AspectJ is a tool about which many many articles haven been written. I will not go into any more details here.

Metaprogramming

Metaprogramming and meta-object protocols [AOMP] is another way of handling cross-cutting concerns on programming language level – if the language supports these features. What is metaprogramming? Metaprogramming, in the context of this article, basically means that it is possible to modify the behaviour of a program not by modifying the program itself, but by programmatically modifying the execution engine that runs the program. Take OO programming as an example: instead of hardcoding the behavior that occurs when, for example, an operation is invoked, this behavior is also defined with the help of classes and objects (i.e. searching for the correct polymorphic implementation method of an operation, passing parameter, etc.). Such classes are called metaclasses. Each class defined by a developer is assigned a metaclass that defines the semantics behaviour related to the execution of the class. Consider the following example[2]. Let us first define an ordinary class *Test*.

[2] The example is rendered in a language called MetaJava, a hypothetical version of Java that supports metaprogramming. Real languages would be for example CLOS or, to some extent, Smalltalk.

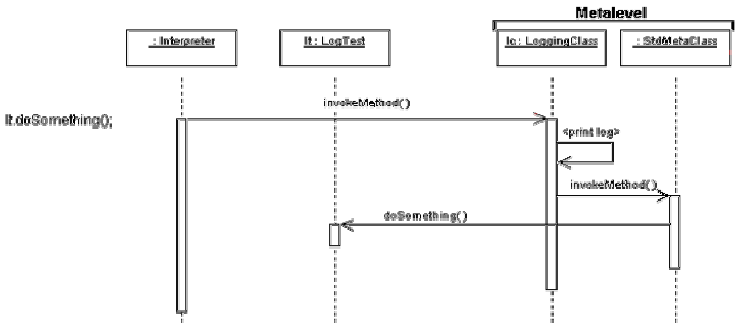
```
class Test extends Object {
     public void doSomething() {
         // do something
     }
}
```

As we did not specify anything else, the default metaclass of *Test* is *StdMetaClass*. If, for example, we want to log all invocations of methods, we can simply define a new metaclass and associate it with all classes whose method invocations should be logged:

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

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

The following sequence diagram shows what happens when a method (*doSomething()*) is invoked on class *LogTest*, which has *LoggingClass* defined as its metaclass.



Cross-cutting behavior can be defined by associating a custom-developed metaclass with all application classes for which a certain behaviour is required. The behaviour that *can* actually be influenced with the help of the metaclass depends on the level of detail that the metaobject protocol allows to control. A good introduction to meta object protocols as well as a good example example can be seen in the book by Kiczales & Co, called *The Art of the Metaobject Protocol* [AOMP].

AO and enterprise development

For many large distributed systems the level of the programming language is practically not important, the features of the language are not essential. More important is system architecture and design. It is therefore a valid approach to handle cross-cutting concerns on the level of the architecture instead of the language. As a consequence, this approach does not require an aspect-oriented language – instead, an architecture/design is required that allows for handling of cross-cutting concerns. I would like to show a couple of examples.

One way to achieve this is to design a metalevel architecture (see the *Reflection* pattern in [POSA1] and the stuff by Brian Foote [FooteReflection]). Here, the metaobject protocol is explicitly built into an application design. The factory/proxy example in the first section can be considered an (very simple) example of this approach.

Another example that is often used in enterprise application architecture are interceptors. Interceptors, as their name implies, typically intercept method invocations or other points in the execution of a program. Typically this happens in the context of remoting frameworks or component containers. In such architectures it is possible to associate the same interceptor with several remote objects/components and thus provide centralized handling of cross-cutting concerns. There are several examples:

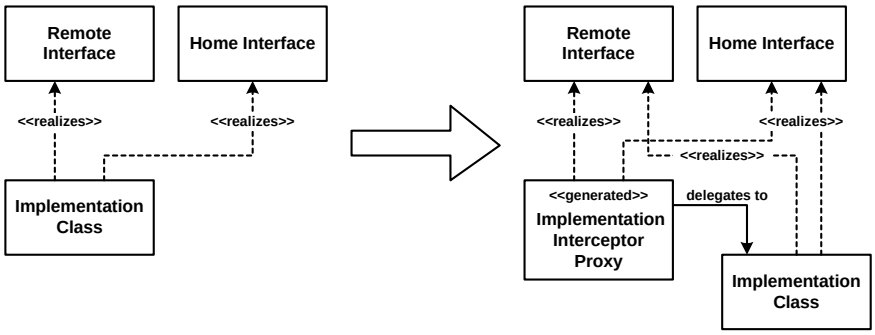
- CORBA's portable interceptors provide a way to intercept remote invocations for CORBA objects. The interceptors and their interfaces are standardized. It is also possible to include cross-cutting behaviour into the custom policy classes that can be configured into the POA [VinoskiHenningCORBA].
- In .NET it is possible to include custom-developed sinks into the message handling chain. Also, these sinks have a standardized interface and it is possible to use the same set of custom-sinks for a wide-variety of remote objects [RammerRemoting].
- In EJB, there are no standardized ways to handle cross-cutting concerns. We will show two approaches how this can be done manually below.

Note that all the examples mentioned above provide developers with "reflective" information about the method invocation that it intercepted. This is possible because the remoting or component infrastructure needs items anyway and can thus provide these to the developer. For example, in .NET the following is the interface of a *DynamicSink*:

```
public interface IDynamicMessageSink {
  void processMessageStart( IMessage request,
                            bool clientSide, bool isAsync );
  void processMessageFinish( IMessage reply,
                             bool clientSide, bool isAsync );
}
```

The *IMessage* interface provides information about the invoked method, such as method arguments.

Let us look at some examples of interceptors in the context of EJB component containers [VoelterEtAlComponents]. The JBoss application server [JBoss] uses the concept of the so-called generalized aspect container. The complete EJB functionality is implemented as a series of interceptors. Each handles "a little bit of EJB", such as pooling, transactions, security, passivation or entity bean persistence. A specific type of EJB container (stateless session bean, stateful session bean, entity bean or message-driven bean) is simply a specific configuration of interceptors that includes those that are required to implement the EJB standard. To be able to handle application-specific cross-cutting concerns, it is possible to develop custom interceptors and include those into the container configuration. Of course, this is non-EJB-standard and thus not portable. It is, however, relatively easy to develop such a system in a portable manner. This works by code-generating proxies for bean implementation classes. Code generators to develop such kinds of proxies are available in many facets [VoelterCodeGen]. This proxy delegates to the regular bean implementation class in order to provide the implementation functionlity. In addition, this proxy provides an interface where interceptors can be plugged in. The configuration can happen based on a configuration file or some other means. The following illustration shows this approach.



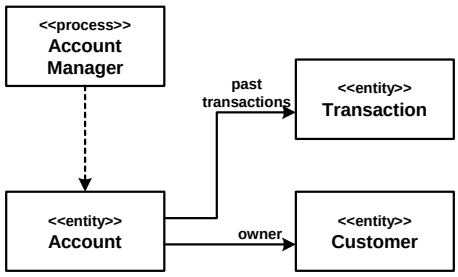
Note that information about the current "join point", i.e. the method that is invoked and its parameters, the caller, etc., can be made available to the interceptors by code-generating a suitable implementation for the proxy. As a consequence of this approach, it is possible to handle cross-cutting concerns for all or some beans in a system in a portable manner. This approach has been used by the author in several projects, mainly for implementing logging, monitoring and advanced security policies not implementable with the help of declarative J2EE security.

Another example (that actually is J2EE compliant) are Servlet filters. Servlet filters are a means to attach cross-cutting behavior to a series of servlets. They pre- or postprocess the http request and response respectively. In my view however, this approach is not particularly relevant anymore, since most larger web applications are built using the Struts framework [Struts], or similar tools. Struts provides a way to handle cross-cutting concerns by developing a custom handler servlet, or by implementing a base class for struts actions (and using the template method pattern (see [GoF]) for the concrete actions).

Cross-Cutting Concerns in the context of code generation

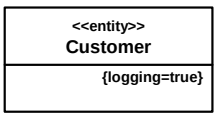
Code Generation is becoming a more and more important means for application development, mainly because of the proliferation of model driven development, and specifically OMG's MDA [OMGMDA]. Let us look at how we can handle cross-cutting concerns in the context of code generation from (UML) models. Note that we do not intend to explain a general approach to describe aspects using UML (there are already some proposals available on the web), we use some rather crude mechanisms here; the goal is to show code generation.

Let's again use the typical logging example (yes I know it is rather useless, but it is simple!). Consider a system that is modeled with the help of some UML tool. For example, the following simple model could be used:



In this example, an *Account* is what you might expect, an *Account* is owned by a *Customer*, and an *Account* also keeps track of the history of *Transactions* in which it has been involved. An *AccountManager* is responsible for running *Transactions* against *Accounts*, in a transaction (ACID) manner.

Now consider we want to log all method invocations on all the entity classes. There are two aspects (no pun intended) of this issue: How do we specify this fact in the model, and how do we implement it in the context of code generation. We don't want to look too much into the specification issue, but there are two primary ways. One is the use of *TaggedValues*. For example, a tagged value "logging" can be defined that can have *true* and *false* as its values, false being the default in case the tagged value is not specified for a model element. The following shows an example notation for such a tagged value.



The same approach can be used for other aspects, obviously. This approach has the disadvantage, that it requires an invasive change to the model, however. The definitions of all classes whose operations should be logged needs to be modified to include the tag. An

alternative approach could be to use an external aspect definition, for example using an XML file:

```
<aspect-definitions>
  <aspect id="logging" className=”LogAspect”>
    <config>
      <logger-target type="file" parameter="c:\\temp\\log.txt"/>
    </config>
    <apply-to>
       <class name="Customer"/>
       <class name="Account"/>
       <class name="Transaction"/>
    </apply-to>
  </aspect>
</aspect-definitions>
```

Once the application of aspects to classes (or other model elemens such as methods, attributes or associations) is defined, we need to consider how to implement this. Of course, this depends heavily on the implementation of the generator – details below. However, there is one important observation: typically, all instances of a metamodel element (such as all classes, all methods, etc.) are generated by *one* part of the generator. Cross-cutting concerns can thus be handled naturally in a centralized place: the place in the generator, where the code for the metamodel element is generated. Let us look at examples.

Using the *Jenerator* API-based code generator [VoelterGaertnerJenerator], typically, a class *CEntity* can be used for generating <<*entity*>> classes[3]. These generator classes are fed with data from the model. The following is an example:

```
public class CEntity extends CClass {
  private List definedBusinessAttributes = new ArrayList();
  public void addAttribute( BusinessAttribute bi ) {
    definedBusinessAttributes.add( bi );
  }
  public void generateCode( CodeContainer container ) {
    super.generateCode(container);
    Iterator    it    =    definedBusinessAttributes.iterator();
    while ( it.hasNext() ) {
      BusinessAttribute bi = (BusinessAttribute)it.next();
      addAttribute( generateAttribute( bi ) );
      addMethod( generateGetter( bi ) );
      addMethod( generateSetter( bi ) );
    }
  }
}
```

[3] In the example, we do not create the implementation code, this is done manually.

The following piece of code shows how to use this generator class:

```
CEntity entity = new CEntity( “Customer” );
entity.addAttribute( new BusinessAttribute( “name”, “String” );
entity.addAttribute( new BusinessAttribute( “age”, “int” );
entity.generateCode( … );
```

Although we did not explain all the details of the *CEntity* class, its use should be clear from reading the code. There is no cross-cutting concerns handling yet. Let's look at some more details. Here we show the implementation of the *generateSetter()* method. We can see that it uses the *CMethod* class internally.

```
public class CEntity extends CClass {
  // …
  public void generateSetter(BusinessAttribute bi ) {
    CMethod m = new CMethod( this, "set”+
                             bi.getName(), bi.getType() );
    return m;
  }
}
```

Once the *CMethod* object is created, it is returned to the *CEntity* class which in turn calls the *addMethod()* operation, thus adding the new method object to the list of method objects of the class. Here is where the interesting things happen with regards to aspects:

```
public class CClass {
  public void addMethod( CMethod m ) {
    methods.add( m );
    Iterator it = Aspects.getAspects();
   while ( it.hasNext() ) {
      CAspect aspect = (CAspect)it.next();
      if ( aspect.isApplicableTo( m ) ) aspect.applyTo( m );
    }
  }
}
```

Here we scan the list of aspects registered in the generator session for all the aspects that are applicable to the particular method. If it is applicable, we actually apply it to the target object:

```
public abstract class CAspect {
  public àbstract boolean isApplicableTo( CObject o );
  public abstract void applyTo( CObject o );
}

public class LogAspect {
  public void isApplicableTo( CObject o ) {
    return o instanceof CMethod;
  }
  public void applyTo( CObject o ) {
    CMethod m = (CMethod)o;
    CCode body = m.getBody();
```

```
      body.prepend( new CCode( “Calling “+m.getName()+” ) );
    }
  }
```

The following piece of code shows how to use this system:

```
public class GeneratorMain {
  public GeneratorMain() {
    Aspect.addAspect( new LogAspect() );
    // get all entities from the model
    // and instantiate them…
  }
}
```

This piece of code will apply the aspect to all classes in the system. If we want to integrate this with the the tagged value approach we could use the following approach:

```
public class LogAspect {
  public void isApplicableTo( CObject o ) {
    if ( o instanceof CMethod ) {
      CMethod m = (CMethod)o;
      m.getContainingClass().getTaggedValue(
              “logging” ).hasValue(“true” );
    }
  }
}
```

For an integration with the XML-config-file-driven approach, the *Aspect* class would read this file, register all the aspects configured in it, and would only ask those aspects whether they are applicable if these are configured to be applicable in the config file.

In template-driven generators, the handling of cross-cutting concerns can be implemented in the templates. For example, the following is a (part of a) template for a generator that generated EJBs from models.

```
<<FOREACH Bean b IN Model m>>
  <<DEFINE ImplementationClass FOR b>>
    public           class           <<b.Name>>Impl           {
      <<FOREACH Operation o IN b>>
        public <<o.Type>> <<o.Name>>( <<o.ParamListAsString>>) {
          <<IF b.hasAspect(“logging”)>>
            Console.WriteLine( “calling <<o.Name>>” );
          <<ENDIF>>
          <<PROTECTED-REGION MethodImplementationCode>>
        }
      <<ENDFOREACH>>
    }
  <<ENDDEFINE                                                  >>
<<ENDFOREACH>>
```

Here, the template contains an *IF* statement that asks the model whether a bean has the *logging* aspect, and if so, the log statement is included. Note that, again, we have localized a cross-cutting concern in one place, in the generator template.

You might say that this can only handle very simple, coarse-grained aspects, such as method calls. You might think it is hard to implement loggings at the location *from where* a method is called. Well, this actually depends on the amount of code we generate. If we don't generate method invocations from models, then we can't plug in aspects there using the generator. However, if we generate real behavioral code for example from state charts or activity diagrams, or with the help of UML 2's action semantics, then there is nothing that prevents us from generating whatever aspects we want.

Summary

This article wanted to provide some overview over how handling of cross-cutting concerns can be implemented without AspectJ and the like. However, I don't want to generate the impression that I consider AOP unncessary, since it can all be simulated without it. In fact, I think AOP will be one of the major improvements to current programming language practice. Today, however, in many projects it is not possible to use AOP because of the limited availability of tools and the general reluctance to use new approaches in mission-critical projects. I hope this article proposed some ideas of how some of the benefits of AOP can be realized based on clever design, architecture and implementation. Let me know what you think.

Acknowledgements

Many thanks to Alexander Schmid who reviewed earlier versions of the article.

References

AOMP	Kiczales et. al., *The Art of the Metaobject Protocol*, MIT Press 1991
AOSD	http://aosd.net
FooteReflection	Brian Foote, *Reflection Pages*, www.laputan.org
GoF	Gamma et. al., *Design Patterns*, Addison-Wesley 1994
JBoss	www.jboss.org
OMGMDA	www.omg.org/mda
POSA 1	Buschmann et. al., *Pattern-Oriented Software Architecture Volume 1*, Wiley, 1996
RammerRemoting	Ingo Rammer, *Advanced .NET Remoting*, Apress, 2002

Struts	*The Struts Framework*, http://jakarta.apache.org/struts
VinoskiHenningCORBA	Vinoski, Henning, *Advanced CORBA Programming with C++*, Addison-Wesley, 1999
VoelterCodeGen	Markus Voelter, *Code Generation, an overview over tools and techniques*, http://www.voelter.de/data/presentations/ProgramGeneration.ppt
VoelterEtAlComponents	Voelter et. al., *Server Component Patterns*, Wiley 2002
VoelterGaertnerJenerator	Voelter, Gaertner, *Jenerator – Generative Programming for Java*, http://www.voelter.de/data/pub/jeneratorPaper.pdf