

Patterns for Handling Cross-Cutting Concerns in Model-Driven Software Development

Version 2.3, Dec 26, 2005

(c) 2005 Markus Völter, Heidenheim, Germany
voelter@acm.org, www.voelter.de

NOTE:

copyright 2005 Markus Völter. Permission is hereby granted to copy and distribute this paper for the purposes of the EuroPLoP 2005 conference.

Abstract

Aspect Oriented Programming (AOP¹, see [AOP]) as well as Model-Driven Software Development (MDSD, see [MDSD]) are both becoming more and more important in modern software engineering. Both approaches attack important problems of traditional software development. AOP addresses the modularization (and thus, reuse) of cross-cutting concerns (CCC). MDSD allows developers to express structures and algorithms in a more problem-domain oriented language, and automates many of the tedious aspects of software development.

But how do the two approaches relate? And how, if at all, can they be used together? This paper looks at both of these questions. The first one – how AOP and MDSD relate – is briefly discussed in the following paragraphs. How AOP and MDSD can be used together is the subject of the main discussion, where the paper presents six patterns of how MDSD and AOP can be used in conjunction. All but one pattern focus on the code phase aspect of MDSD.

¹ I always use the term AOP instead of differentiating between AO in general and AOP as the AO approach to programming. Please see Appendix A for a differentiation between AOP and AOSD

Reader's Guide: Structure of this Paper

This paper is structured as follows. The *Patterns Prologue* section describes the pattern form used in this paper, as well as the language-wide problem statement and the forces applying to all the patterns. The *Pattern Overview* section provides a first overview over the patterns – this section is continued in *Pattern Overview Pt. 2*, the section that follows the patterns themselves. *Relationship among the Patterns* provides some more detail about how some of the patterns interrelate. Finally, the *Concluding Remarks* and *Acknowledgements* sections end the paper.

Readers who do not have a solid understanding of AOP or MDSD are recommended to read the *Appendix A*, where I introduce the two techniques and discuss a number of commonalities and differences.

For completeness, *Appendix B, Introductions and Collaborations* looks at two AOP-relevant topics –introductions and collaborations – that are not considered in the main part of the paper.

Patterns Prologue

This section contains a prologue to the patterns themselves. First, we declare the overall problem of this pattern collection, then we list the forces that influence the solutions the various patterns propose for the common problem.

General Problem statement

As we have seen in the previous sections, there are a number of commonalities between AOP and MDSD. As a consequence, developers often don't know whether, or how they should relate AOP and MDSD. Should they use either AOP or MDSD? Is AOP or MDSD a more general approach? Is MDSD a special case of AOP? Or vice versa? Can/should both approaches be used together, or would that just be “hype overkill”?

The following patterns are intended to answer some of these questions. As a consequence of the authors' experience and opinion, they are written from an MDSD perspective, i.e. they solve the following problem in the context of different forces:

How can cross-cutting concerns be handled effectively in an MDSD-based development environment?

An author with a stronger background in the AOP domain might have written the paper from the other perspective, addressing problems such as “how can domain-specific notations used in AOP”, or “how can AOP address non-programming language concerns”.

General Forces

This section introduces a number of forces that influence the solution of the patterns that follow. All the patterns described below are governed by the forces listed in this section; however, they resolve those forces differently. As a consequence, the various patterns presented below are applicably in different contexts. All patterns include an evaluation of all of these forces in their respective *consequences* section.

Applicability. We would like to be able to use the pattern’s solution to the problem above in a many situations, environments and “technology environments” as possible. The broader the applicability the better. The more we rely on particular features of languages, architectures, technologies or environments, the harder it is to use the solution in general.

Granularity. Handling CCC – as explained – is about specifying queries over the execution of a program, and then doing something at (some of) these selected points. Different approaches provide different levels of granularity, at which such a query can be specified. For example, an approach might only allow to advice calls to component operations, whereas other approaches might allow interception of any method call in the system, thrown exceptions, field access, etc.

Performance/Footprint. As usual in software development, nothing comes for free; each proposed solution has a more or less dramatic impact on system performance or footprint. In some environments, such as embedded systems, this can become a problem that deserves developers’ attention.

Complexity/Verbosity. Another well-known problem in software technology is, that while a certain approach solves a specific problem, it creates additional complexity – aka problems – in another area. For example, the requirement to use additional languages or tools can be such an issue. Another issue in this respect can be the readability of the generated code, or the verbosity of the things you have to write/specify in order to use the pattern.

Flexibility. Different approaches to CCC handling have different consequences with regards to (runtime) flexibility. Some approaches allow to turn on/off the handling of a specific aspect at runtime or allow to change the behaviour at a certain pointcut, while others don’t.

Pattern Thumbnails

The following list provides a thumbnail of each pattern. The more extensive discussions below provide a lot of additional detail.

Template-IF: Exploit the inherent cross-cutting nature of code generation templates by using simple if statements in templates to handle CCC.

AO Templates: An AO template engine allows you to “advice” code generation templates, further modularizing CCC.

AO Platforms: Exploit the architecture-based handling of CCC provided by the platform by generating configuration files that control the platform.

Pattern-Based AOP: Use generated implementations of the proxy, factory and interceptor patterns to address CCC.

Pointcut Generation: Use the code generator to generate pointcuts for pre-built (abstract) aspects based on specifications in the model.

A Model per Concern: Use several models (one for each concern) to describe the complete system, and let the generator weave the models together.

■ TEMPLATE-IF

Exploit the inherent cross-cutting nature of code generation templates by using simple *if* statements in templates to handle CCC.

Context

You are using a template-based code generator [MV03]. The templates contain code that iterates over the model as well as textual output that should be created for a certain part of the model. The CCC you need to handle can be well localized in the templates.

Example. The following template generates code from UML models. Specifically, it creates a method signature and skeleton implementation for each *Operation* in the model.

```
<<DEFINE OperationDef FOR Operation>>
    public final <<ReturnType>> <<Name>> (
        <<FOREACH Parameter AS p EXPAND USING SEPARATOR ", ">>
            <<p.Type.QualifiedJavaTypeNames>> <<p.Name>>
        <<ENDFOREACH>> ) {
        return <<Name>>Internal(
            <<FOREACH Parameter AS p EXPAND USING SEPARATOR ", ">>
                <<p.Name>>
            <<ENDFOREACH>> );
    }
<<ENDDefine>>
```

Solution

Use normal template-level *if* statements to address the CCC. Depending on the *if* expression, a particular piece of code is either added to the generated code or not.

Example. The following example code uses an *if* statement to add security checking in case security checks are enabled for the particular operation.

```
<<DEFINE OperationDef FOR Operation>>
    public final <<ReturnType>> <<Name>> ( ... as before ... ) {
        <<IF checksRequired>>
            if ( !Security.check("<<Class.Name>>", "<<Name>>") )
```

```

        throw new SecurityEx();
    <<ENDIF >>
    return «Name»Internal( ... as before ... );
}
<<ENDDEFINE>>

```

Rationale & Discussion

A template is a meta program, a program that creates programs. As such, usually a number of base-level artefacts (here: operations) are created from a single template. If you need to handle concerns that cross-cut these locations, then a template modularizes this cross cutting concern. A simple *if* on template level is therefore enough to handle the CCC.

Note that you can also implement around advice using this pattern. The generated code itself can contain an *if* statement, and call the original code only if the condition is true; it is also possible to modify the parameters.

By having several *if* statements in the template you can also advice advice, although this can become complex quickly.

AOP purists may argue that the approach isn't AOP at all, since the code calls the advice, and the advice is not "added" to the code. While this is true, the approach is nevertheless very useful to handle CCC.

Consequences

Applicability. The approach requires no special features in the target language. Also, it is not limited to programming language artefacts, the approach can be used for any generated output. With regards to the necessary features of the generator, only a simple *if* is necessary – which is available in basically all code generators.

Granularity. This approach can only be used if the pointcut is actually in a section of the code that is generated. This means that the pointcuts are limited to what is represented in the model, or to what can be derived by generation rules from the model. If, for example, you wanted to advice pointcuts in the method body (which you often don't generate) this pattern is useless.

Performance/Footprint. There is no specific performance hit or footprint issue to this approach in addition to the cycles and memory consumed by the advice itself.

Complexity/Verbosity. For simple CCC there is no complexity problem. If you wanted to handle many CCCs at the same pointcuts, you would have a number of such *if* statements, but still this is not a real problem. There are situations, however, when the approach becomes complex. This happens when the same CCC cross-cuts the *templates* and not the generated code. This requires checking the *if* expression in multiple places in the template code, which could itself become a verbosity issue.

A positive complexity consequence of this approach is that the generated code shows no additional complexity, and that you do not need additional (aspect) tools.

Flexibility. The approach is completely static. Nothing can be changed (i.e. woven in/out) at runtime. If it should be possible to turn on/off the aspect at runtime, the advice itself would need to contain a suitable (runtime) *if*.

Known Uses

From a tools perspective, every template-driven code generator can be used to implement this approach. As well, all MDSD projects I know of have used some form or another of this pattern to address CCC. This pattern is so ubiquitous, that mentioning specific known uses is pointless.

Summary

While this approach seems rather trivial, it can be used to handle a surprisingly large amount of CCC that arise in practical work. And the fact that you don't need any AOP tool, is an additional benefit.

■ AO TEMPLATES

An AO template engine allows you to “advice” code generation templates, further modularizing CCC.

Context

In some cases, especially if you're building related families of code generators, using TEMPLATE-IF becomes too unwieldy, because all kinds of concerns are handled inside the templates. Typically, a few architecture-specific *hot spots* inside the templates are affected by *ifs* that handle the various different CCC. These hot spots become unmanageable rather quickly.

Also, the templates themselves contain the code for all the various concerns that might need to be handled at the specific location in the template. This leads to the typical CCC problem – now, however, on template level!

Example. Looking at the solution code of the TEMPLATE-IF pattern, you can see such a typical hot spot. The code below shows how TEMPLATE-IF handles a number of CCCs.

```
<<DEFINE OperationDef FOR Operation>>
    public final <<ReturnType>> <<Name>> ( ... as before ... ) {
        <<IF checksRequired>>
            // security code
        <<ENDIF >>
        <<IF loggingRequired>>
            // logging code
        <<ENDIF >>
        <<IF billingRequired>>
            // billing code
        <<ENDIF >>
        return <<Name>>Internal( ... as before ... );
    }
```

```

    }
«ENDDEFINE»

```

Solution

Use an AOP approach on template level. Rather than using template-level *if* statements, use an “aspect template” that advises the standard code generation templates with CCC-specific code. There are two ways how a pointcut can be defined; implicit and explicit. The examples show details of this difference.

Example. The following piece of code (again the example from above) defines two explicit join points: *MethodBegin* and *MethodEnd*.

```

«DEFINE OperationDef FOR Operation»
public final «ReturnType» «Name» ( ... as before ... ) {
    «EXPAND HookMethodBegin»
    «ReturnType» res = «Name»Internal( ... as before ... );
    «EXPAND HookMethodEnd»
    return res;
}
«ENDDEFINE»

```

After these hooks have been defined, another template can attach itself to this hook. The following piece of code shows the logging aspect as an example.

```

«DEFINE LoggingMethodBegin FOR Operation AT HookMethodBegin»
    «IF loggingRequired»
        // entering method such and such
    «ENDIF »
«ENDDEFINE»

«DEFINE LoggingMethodEnd FOR Operation AT HookMethodEnd»
    «IF loggingRequired»
        // leaving method such and such
    «ENDIF »
«ENDDEFINE»

```

Other CCCs can be handled in the same way: by providing separate templates that are “attached” to previously defined hooks.

The implicit scheme of defining join points basically means that “aspect templates” can attach to *before* or *after* already defined templates. The following piece of code shows an example.

```

«DEFINE OperationLogging BEFORE OperationDef»
    // logging stuff
«ENDDEFINE»

```

Note, however, that this means that you can only attach to template boundaries. The example above, e.g., would contribute logging code to *before* the *OperationDef* template – i.e. the logging code would be generated outside of the method body. This is clearly not what is intended. Solving this problem with implicit join point definitions would require to factor out the method body into a separate template, and attaching the aspect to this body template.

Rationale & Discussion

This pattern basically introduces AOP at the template level. TEMPLATE-IF uses normal template programming to handle CCCs in the resulting generated code by using *ifs* on template level. The pattern described in this section handles CCCs on template level and uses AOP techniques to address those.

There is an interesting challenge lurking in the details of this pattern. Some code generators use an AST implemented in a particular language (e.g Java, see *Ignore Concrete Syntax and Implement the Metamodel* in [VB04]) as the basis for code generation – the template effectively traverse the AST (aka Java objects), generating code as they go along, accessing the AST objects to retrieve information from the model to be added to the generated code. For example, the following piece of code accesses the *Name* property of an *Operation* metaclass:

```
«DEFINE OperationDef FOR Operation»  
    public void «Name» ...  
«ENDDEFINE»
```

Now, if we attach additional aspect templates to this existing template structure, we can only access those properties that are already available on the current metaclass. This is very limiting. For example, we could not access the *loggingRequired* flag that determines, whether we need to generate logging code or not, since it probably is not in the core metamodel classes. In order to solve this problem, we have to be able to contribute additional properties (operations) to existing metaclasses. Again, AOP comes to help: introductions can be used to add these additional operations to existing metaclasses.

What about join point variables? In this pattern, the advice template has the same context as the template it advises, i.e. it has access to the same information from the model. Additional information that could be made available via a join point variable could the name of the template that has been advised; however, I haven't seen this feature in practice, nor have I felt the need to use it.

We only discussed before- and after advice style aspects. However, it is also possible to extend the mechanism to also include around advice. To make this work, a special keyword *proceed* would have to be added, as shown below:

```
«DEFINE AROUND SomeTemplate FOR Operation »  
    // do something here...  
    «IF someCondition»«PROCEED»«ENDIF»  
    // do something else...  
«ENDDEFINE»
```

While this is certainly possible, I have not seen support for this in a tool. openArchitectureWare provides a related concept: template inheritance, meaning that you can override the template defined before; but a call to *super* (which would be similar in consequences as the *proceed* call above) is not possible at this time.

Consequences

Applicability. The approach requires no special features in the target language. Also, it is not limited to programming language artefacts, the approach can be used for any generated output. However, the generator tool used to implement the approach has to provide support for defining hooks and “attaching” templates to them – implicitly or explicitly.

Granularity. Same as for TEMPLATE-IF.

Performance/Footprint. There is no specific performance hit or footprint issue to this approach in addition to the cycles and memory consumed by the advice itself.

Complexity/Verbosity. The approach described in this pattern nicely factors out CCC in templates, thereby reducing the verbosity in the templates. Whether additional (accidental) complexity is created depends very much on the tool used. AOP on the template-level is not very widespread and only a few tools support it – some of them only more or less well.

Flexibility. See TEMPLATE-IF.

Known Uses

There are various tools that can be used to implement AO TEMPLATES. The openArchitectureWare code generator [OAW] provides a feature called *attached templates* that implements this pattern. It uses interceptors that can be configured by the developer to contribute additional operations to the metaclasses.

Also, the XVCL frame processor [XVCL] allows to “contribute” frames (which can be seen as a form of code generation templates) to previously defined hooks.

Summary

AOP on template level is very powerful. However, the generator and its templates effectively become an AOP language. The tools I am aware of only support this approach as an add-on, limiting the scalability of the approach. Specifically, the IDE support that we are used to (such as AspectJ’s Eclipse integration) is not available.

■ AO PLATFORMS

Exploit the architecture-based handling of CCC provided by the platform by generating configuration files that control the platform.

Context

You are generating code that is intended to run on a technical platform, usually some kind of communication or component/container middleware. Such middleware typically already supports factoring out some of the technical CCC that occur in the domain for which the middleware has been developed. The

middleware platform usually also provides some kind of configuration facility (annotations (see [VSW02]), scripts, or descriptors) to control how the middleware applies its CCC capabilities to the respective piece of application code.

Example. In EJB systems, a component encapsulates functional (or domain) concerns. Technical concerns such as transactions, security, load balancing, or pooling are taken care of by the container (which itself is embedded in the application server). Deployment descriptors accompany the components and control how the application server handles them.

Solution

Use the CCC-handling capabilities of the middleware as far as possible. Use the code generator to generate the annotations (see [VSW02]) that control how the middleware handles the (manually written, or generated) application code. The information needed to generate the configuration is extracted from the model.

This pattern does not just recommends to use a platform's CCC-handling capabilities *in case it happens to provide these*. Rather, the pattern suggests to actively *build* (or select) platforms that provide hooks to handle the typical CCC in the respective domain.

Example. Many MDSD tools in the context of EJB require developers to develop POJOs that contain the business logic. The generator then creates “EJB wrappers” that make sure the POJOs conform to the constraints defined by EJB. The generator also creates a deployment descriptor to control the EJB container.

Rationale & Discussion

This pattern basically suggests to leave the handling of the CCC to the target architecture – the MDSD infrastructure does not have to deal very too much with handling CCC. A couple of comments are in order, though.

First, in order to generate the configuration for the middleware platform, the model from which the code is generated needs to contain all the information (explicitly, or in a way that allows the generator to derive it) that goes into the configuration. We need to make sure this information does not clutter the “business” model described with our DSL. A MODEL PER CONCERN is a good way to keep the core model clean.

Consequences

Applicability. The approach requires that the environment in which the generated code is intended to run provides means to handle CCC. This is not always the case, although it is usually possible to build a platform that handles the CCCs of your domain. In resource-constrained systems this is sometimes a problem, though.

Also, the approach only allows to handle those CCCs that the platform supports. In case you build the platform for your domain, this is not a problem, since you'll build it to handle the CCCs you need. In case you use an off-the-shelf platform

such as EJB, this can become a problem. To solve this problem, use PATTERN-BASED AOP or POINTCUT GENERATION.

Granularity. The granularity is limited to the granularity provided by the middleware platform. Again, if you build it yourself, this is often not a problem, since you can make it fit. In case it's not you who builds the platform, you cannot do much about it.

Performance/Footprint. Platforms that support the handling of CCCs will almost always² imply some overhead. The reason is that if a platform can generically handle (certain) CCCs, then it will always use some dynamic, generic, or reflective mechanism to achieve that. Such an approach always implies overhead in performance, and often also footprint. How big this overhead is depends on the implementation. And whether this overhead is a problem for you depends on your scenario and use case, but there *is* an overhead.

Complexity/Verbosity. The approach described in this pattern nicely factors out CCC into the platform. If the platform handles the CCC you need, this approach is unbeatable in simplicity, because you just generate configuration files and don't really need to care about generating code that handles the CCC. Specifically, your generator becomes much simpler – considering today's mainstream generator tools, this is a nice effect.

Flexibility. In case the platform handles CCC, it is usually possible to turn on/off a specific CCC during runtime, or change the way how the CCC is handled. Whether it is actually possible to do this, depends on the implementation of the platform. If you build it yourself, there is no technical reason why having that flexibility would be a problem.

Known Uses

EJB provides a platform where (some) CCC can be handled by specifying how they should be handled in the deployment descriptors [VSW02]. The CORBA component model (CCM) provides a similar feature. Plain CORBA allows developers to add interceptors to remote objects (or groups of remote objects, see [VKZ04]). However, there are no default interceptors which can be configured by generating a configuration. Many other distributed object middleware systems or web service platforms provide the same features, this is actually a pattern in distributed object middleware [VKZ04].

Summary

Non-trivial systems developed using MDSD will almost always include a rich, domain-specific platform, specific to the domain for which you build (generate)

² You can only avoid this overhead if you custom-generate the platform to the specific scenario. While this is a worthwhile approach especially in embedded systems [MV03b], we then don't handle the CCCs with a *predefined* platform anymore, moving that scenario out of the focus of this particular pattern.

applications [VB04]. From a reuse perspective, it is a good idea to move as much (generic, domain-wide) functionality into this platform because you can use it from within the generated code. CCC are primary candidates for functionality in such a platform.

■ PATTERN-BASED AOP

Use generated implementations of the proxy, factory and interceptor patterns to address CCC.

Context

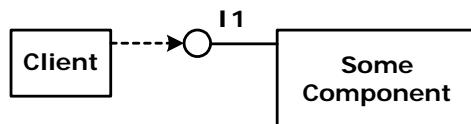
In some scenarios the platform you are required to use does not provide services that handle CCC, or it does not handle the CCC you need to address. You still need to have the flexibility to change at runtime the CCCs handled by the system. You maybe even don't know the CCCs you need to handle at generation time. However, the pointcuts are accessible to the generation process.

Example. Consider again an EJB based system. Consider also, that you need to implement so-called dynamic (or data driven) security. This means, you cannot use EJBs default (static) security model. However, you also don't want to bother application (component) developers with handling the security concerns.

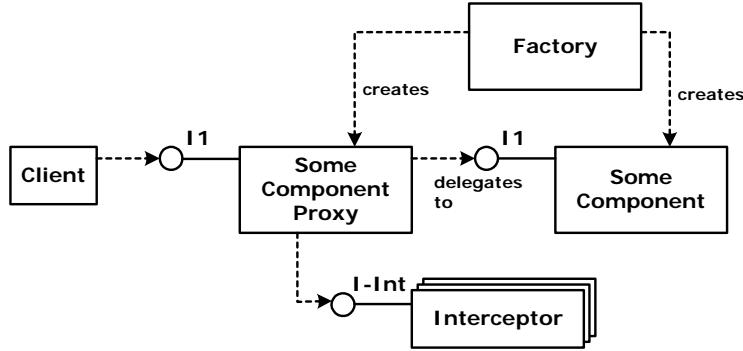
Solution

Use a selection of the well-known patterns to generate an infrastructure that allows for custom CCC-handlers to be plugged in. Typically, this consists of generating proxies [GoF] for application components that can hook-in interceptors [POSA2]. Use a factory to instantiate the proxies if necessary.

So, consider the situation, where a client accesses a component through a specific interface *I1* as in the following diagram:



Then replace this setup by a setup that includes the proxy to host interceptors as well as a factory that creates the component and the proxy, and “wires” the two accordingly:



From a client's perspective, nothing has changed, the client still uses the interface *I1*. However, the client actually talks to a proxy that handles CCC, and then forwards to the real object.

Typically, you will make sure the join points are method calls. This makes life simple for the proxy. The interceptor interface is then the typical method call interceptor interface, outlined in Java below:

```

public interface Interceptor {
    public void beforeInvoke( Object target,
                             String methodName,
                             Object[] params );
    public void afterInvoke( Object target,
                           String methodName,
                           Object[] params,
                           Object retValue );
}

```

The proxy will then be implemented along the following lines:

```

public class SomeComponentProxy implements I1 {
    private SomeComponent delegate;
    private Interceptor interceptor; // can also be a list
                                     // of interceptors
    public String someOperation( String p1, int p2 ) {
        Object target = delegate;
        String opName = "someOperation";
        Object[] params = {p1, p2};
        Interceptor.beforeInvoke( target, opName, params );
        String res = delegate.someOperation( p1, p2 );
        Interceptor.afterInvoke( target, opName, params, ret );
        return res;
    }
    // more operations of I1
}

```

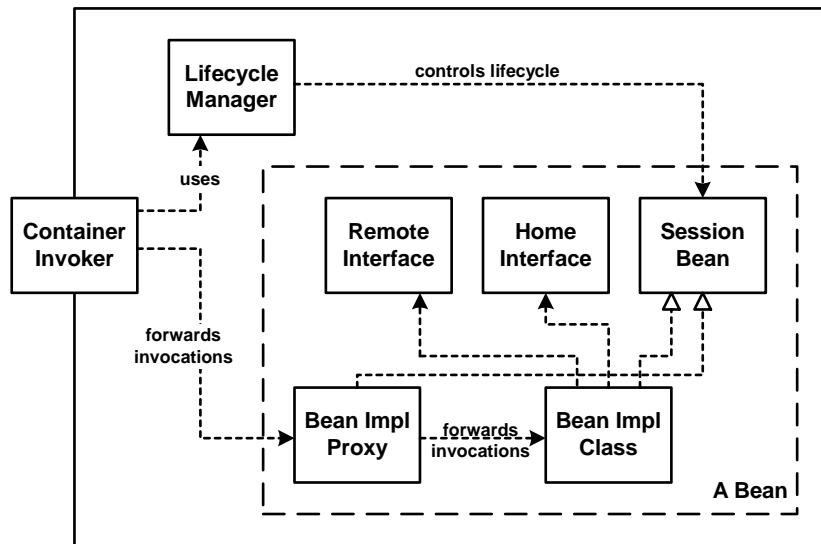
The factory uses some kind of configuration to determine, which interceptors are necessary for which component/object. In case no interceptors are necessary, the proxy can be left away, and we have an overhead-free configuration where the client directly talks to the target object.

Example. In the EJB scenario introduced above, the generated proxy would be the bean implementation class from the perspective of the application server, the real

bean implementation would be an “implementation detail” of this class. So, whenever the container wants to instantiate a bean of a specific type, it will automatically instantiate the generated proxy class. This will in turn instantiate the real bean implementation.

At runtime, the bean implementation that plays the role of the interceptor proxy will use some kind of configuration to find out which interceptors should be used, if any.

This makes it possible to introduce any kind of “CCC handler interceptor” into the EJB container.



The implementation of the interceptor would use EJB APIs to find out about the current security principal, and then use it, as well as the bean’s identity, and operation parameters to decide whether to allow the call or not. Note that using the `afterInvoke()` callback, you can also apply security checks based on the result of the call!

For details of this example, see [MV04]

Rationale & Discussion

The approach here works whenever (a) you can influence object creation so that the proxy is created instead of the real object, and (b) if method call-level pointcuts are acceptable.

While this rather coarse granularity seems like a limitation, in practice, it typically isn't. The reason is that in well-designed component based systems it is not just practical, it is even desirable to apply aspects to component boundaries to keep the system manageable and understandable. Thus, components continue to be the smallest architecturally relevant building block and cannot be "undermined" by aspects.

This approach is not really specific to MDSD, you can in principle use the same approach in the context of normal, non-generative software development. However, since you would have to manually implement all the proxies, this is impractical, and thus hardly ever done. In the context of MDSD, where you generate stuff anyway, it is trivial to generate the necessary proxies automatically during build, making sure that the proxy interface and implementations are in line with last-minute interface changes of the application component.

Consequences

Applicability. The approach can be applied quite generally, at least in object-oriented systems. The only real precondition is that you are able to “tweak in” the proxy, which means generally, that you have to be able to control object creation. No specific features are required of the generator.

Granularity. The approach only works for join points on method call level.

Performance/Footprint. There is a considerable impact on performance and on footprint. First of all, you will have an additional object (the proxy) for each domain object. Second, for each method call, the method data has to be reified and the interceptor(s) called. As a consequence, the approach does not really make sense for fine grained CCC. Using it on component level (as in the EJB example) is perfectly ok, though.

Complexity/Verbosity. The approach does add complexity, since you have the proxies, the factories and the interceptors to deal with. However, once you have the respective generators built, you just have to deal with the factory configuration, specifying, which object (or class) should have which interceptors. Thus in practice, the complexity added by the approach is tolerable.

Flexibility. Depending on whether the interceptors are configured at runtime or not, it is possible to add, remove or change “aspects” at runtime.

Known Uses

The approach to dynamic security in EJB has been used in several projects, some of which I have been directly involved with. A component infrastructure for small (mobile) devices implemented in Java uses the same approach. Java’s Dynamic Proxy API uses the same idea, but based on reflection as opposed to static code generation.

Summary

I have used this approach in various projects on component level and it works nicely. Particularly the EJB example above is useful (since it is completely portable and does not depend on any application server-specific features). Building the necessary generator (if you work with an MDSD approach anyway) is almost trivial.

■ POINTCUT GENERATION

Use the code generator to generate pointcuts for pre-built (abstract) aspects based on specifications in the model.

Context

In some scenarios all the approaches described above don't work – performance is not sufficient, the platform does not support your needs, or the granularity offered by the solution is too coarse. Is there still hope?

Example. In a component infrastructure for embedded systems [MV03b], where the component container is generated specifically for the scenario at hand, you want to be able to add a tracing mechanism, primarily for debugging and timing checking purposes. Since resource consumption and (near) real time behaviour is an important consideration, you cannot use generic solutions. Cluttering the templates with all kinds of *if* statements is also not acceptable, because you need to trace different things at different times – this would result in *if*'s all over the place. For the same reason, AO TEMPLATES don't work, since you would have to adapt the template structures continually.

Solution

Integrate an AOP language into the MDSD software development infrastructure. Specifically, define a number of prebuilt advice as part of the platform, and then generate the pointcut based on specifications in the model. Use the AOP language's standard weaver to integrate the aspects with the generated code – the code generator can stay untouched, it just has to be extended (not modified!) to generate the necessary pointcuts.

Example. The following piece of XML is a part of the model that specifies a node in the distributed, embedded system³, as well as a component container running on it [VKS05]. You can see the tracing option to be set to *app* which means that we want all application level operations (as opposed to container-internal calls) to be traced. This specification is actually a DSL-specific pointcut definition.

```
<node name="outside">
  <container name="sensorsOutside" tracing="app">
    ...
  </container>
</node>
```

Now, as part of the platform (the library of reusable artefacts used for all members of the software system family), you define the following abstract aspect (using the AspectJ language). It does not define a pointcut, it is thus “pure advice”.

```
package aspects;
```

³ Actually, a weather station – thanks to Danilo Beuche for the example ☺

```

public abstract aspect TracingAspect {
    abstract pointcut relevantOperationExecution();
    before(): relevantOperationExecution() {
        // use some more sophisticated logging,
        // in practice
        System.out.println( System.currentTimeMillis()+":::"+thisJointPoint.toString() );
    }
}

```

In addition to this abstract aspect, the code generator is supplied with a template that, for every container that has *tracing* set to *app*, generates a concrete aspect that adds the necessary pointcut. The result could be the following:

```

package aspects;
public aspect SensorsOutsideTrace extends TracingAspect {
    pointcut relevantOperationExecution() :
        execution( * manual.comp.temperatureSensor...*.*(..) )
        ||
        execution( * manual.comp.humiditySensor...*.*(..) );
}

```

This generated aspect can now be woven with the rest of the (generated, or manually written) code, and thus add tracing to the required parts. For completeness, see below for the code generation template that generates the concrete aspects for the containers.

```

«DEFINE TracingAspect FOR System»
...
«FOREACH Container AS c EXPAND»
    «IF c.Tracing == "app"»
        «FILE "aspects/"<c.Name>"Trace"»
        package aspects;
        public aspect <>c.Name>Trace extends TracingAspect {
            pointcut relevantOperationExecution() :
                «FOREACH c.UsedComponent AS comp
                    EXPAND USING SEPARATOR "||"
                    execution(* manual.comp.<>comp.Name...*.*(..) )
                »ENDFOREACH»
            ;
        }
        «ENDFILE»
    »ENDIF»
«ENDFOREACH»
...
«ENDDEFINE»

```

Annotation-based weaving is another possible – and simpler – solution for this pattern. If your base language as well as the AOP language extension support metadata annotations (for example, a combination of Java 5 and AspectWerkz) you can use the following approach: The prebuilt aspect includes an pointcut definition that tests the presence of a certain metadata attribute. If it is present, the artefact is selected by the pointcut, and the advice is added. The code generator simply has to add the metadata attribute to the artefact, if it wants the artefact to be

affected by the advice. Note that in languages that don't support annotations, you can alternatively use marker interfaces – although this only works for advising classes, and not other artefacts such as fields or operations.

Rationale & Discussion

Using an aspect language such as AspectJ to add tracing to a system is rather trivial, and, at first glance, might not deserve mentioning. However, the question remains *when* it is worth using such a (additional) language in the context of an MDSD project, where CCCs can also be handled differently (see all the other patterns above).

Also, the question is, *how* to integrate it efficiently. Providing advice as part of the platform and then generating the pointcuts is a very useful approach indeed. Of course, it requires that the domain developer decides which advice might be necessary. However, this is required anyway, since the DSL has to have a feature to control where to apply the aspect, and where not.

In the above example, the fact that developers can specify a tracing option for containers in considered so important in the context of the domain, that the necessary specification is done as part of the domain's core DSL. If that were not the case, one could also specify the tracing concern in a separate model – effectively applying the A MODEL PER CONCERN pattern shown below. Example:

```
<trace-config>
  <trace container="sensorsOutside" level="app"/>
  <trace container="sensorsInside" level="all"/>
</trace-config>
```

It is also interesting to see that this pattern suggests using an AOP language such as AspectJ as an *implementation technology* in MDSD projects. The aspectual nature of the tracing concern does *not* show up in the DSL. Rather, AOP is used to keep the implementation of the tracing feature small and fast. I think that this is the primary use case for languages like AspectJ in the context of MDSD. For more patterns on using AspectJ efficiently see [SH03].

Consequences

Applicability. The approach can be applied only if, for the respective target language, an AOP extension is available. For Java, AspectJ is a good candidate, for C++, AspectC++ [AC] can be used (although it is not as powerful as AspectJ). As of early 2005, AOP extensions are available for many languages, although their maturity and power varies significantly. Note that some aspect languages require runtime support libraries – which might be a problem in some production environments.

There are no special requirements for the generator.

Granularity. The achievable granularity depends on the join point model of the aspect language used. In most cases, the granularity offered by these language is fine enough.

Performance/Footprint. Again, this depends very much on the implementation of the aspect language, specifically, when the weaving occurs (statically before runtime, at load time, or at runtime). Static weavers such as AspectJ have, in most scenarios, a neglectable overhead in footprint and performance.

Complexity/Verbosity. Complexity can raise significantly using this approach, since it opens up a “whole new can of worms”. Domain developers have to master the underlying aspect language and integrate it suitably with the MDSD infrastructure. On the positive, application developers (those using the MDSD infrastructure to build applications) do not need to see (and thus, understand) the use of the aspect language under the hood (unless they start the debugger, that is)

Flexibility. Again, this depends on the aspect language used for the implementation.

Known Uses

The small components prototype [MV03b] uses this approach to handle CCCs that cannot be handled using PATTERN-BASED AOP. In the context of mobile phone software, the pattern has been used to generate static aspects (aspects that produce compile time errors) to check developer conformance to programming guidelines.

Summary

This approach is certainly the most powerful. However, it requires the use of an aspect language in addition to all the generator and modelling tools that are necessary for MDSD anyway. This can be a huge problem in practice. Also, in contrast to the MDSD approach, most AOP language extensions require runtime support libraries. In some production environments (such as in large companies) this can be a showstopper.

■ A MODEL PER CONCERN

Use several models (one for each concern) to describe the complete system, and let the generator weave the models together.

Context

Up to now, we were mainly concerned with handling CCC in the resulting application, which would be built using an MDSD approach. The application is described using models, and model transformations and code generation is used to create the final application. In many scenarios, however, it is necessary to separate concerns in the application *models*, too!

Example. Consider you are building a web application. Such a web application typically consists of (a) a business object model, (b) the persistence mapping of this model, (c) the web pages, forms and the workflow, and (d) the layout of these forms and pages. You have to specify all this in the model in order to be able to generate a complete application.

Solution

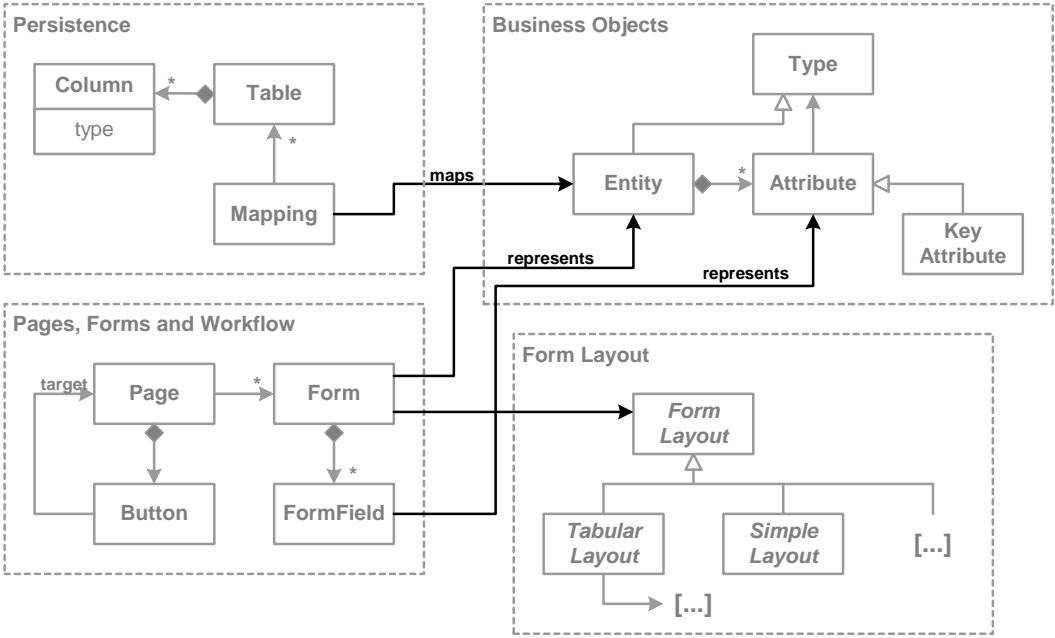
Create several models, one for each aspect. Each model uses a DSL (i.e. concrete syntax and metamodel) suitable for the expression of the particular aspect. The code generator reads all these models, weaves them, and then generates the complete application from it. Join points are defined on the metamodels, for example, by using a specific metaclass in more than one aspect's metamodel, thereby building up links between the models.

Many Models or One Model

In this context, I consider a model a self-consistent "sentence" described in a certain DSL (and based on a certain metamodel, respectively). I.e. a complete system is described by several models, each using a certain DSL to describe an aspect of the overall system. An alternative naming convention would be to call each of the models "partial models" or something like, and call the overall thing that describes the system "model".

Example. In the example above, you could use (a) a UML class diagram (of course, with suitable stereotypes) to describe the business object model, (b) an XML document to describe tables and the mapping, (c) another class diagram (with other stereotypes) to describe pages, forms and the workflow, and finally, (d) another XML document to describe form layout.

This approach would result in four models for each described application, all of which need to be connected suitably to describe a complete and consistent system. For this to work, the metamodels must be related, as shown in the next illustration. Note how associations cross the various aspect metamodels.



The code generator has to be able to read the various models – although they are rendered in different concrete syntaxes! – and perform the weaving according to the relationships shown above.

Note this pattern is a reformulation of the *Technical Subdomains* and *Gateway Metaclasses* patterns from [VB04]

Rationale & Discussion

This pattern effectively suggests to have a separate model for each aspect. The challenge of this approach lies in the fact that the generator tool must be able to:

- *read the various models*: this requires that the generator can use different forms of concrete syntax (UML, XML, ...) as input.
- *check for consistency*: the generator must make sure that the relationships among the metamodels are realized correctly, i.e. that for example each `Form` has a `FormLayout`, each `Entity` has a `PersistenceMapping`, etc. If inconsistencies are detected, the models must not be accepted for code generation, and an error has to be reported to the developer.
- *weave the models*: in order to weave the models, the models must be represented uniformly, once they are “inside” the generator. A good idea is to use an object graph based on the metamodel as the standard representation for models (see [VB04]).

Note that you cannot have separate generators for the different aspects, since the metamodels (and thus, also the models) are related. For example, the persistence generator, has to know the entity structure. This “aspectual” separation of models is therefore fundamentally different from model partitioning, where a large model

is broken down into several smaller ones. It is essential that the generator can perform the weaving process.

Note that implementing this weaving process is much easier inside a generator compared to a tool like AspectJ. The reason is, that your domain metamodels are usually vastly simpler than the metamodel (i.e. abstract syntax) of a language like Java, and that you define the join point model yourself – as part of the domain metamodel specification. In practice, weaving simply requires the association of runtime objects, based on some matching criteria such as name equivalence.

The above mentioned consistency check is also a critical ingredient, since otherwise, the generated system will be incomplete (in case specifications are missing), or your models will fill up with garbage (in the case when you still have specifications that reference model elements that already have been deleted.)

Note that the above discussion focussed on specifically defined pointcuts (such as “use this layout for Entity XYZ”) and did not use quantification. However, this is merely a detail, since it is easily possible to quantify parts of the model in another part, and then let the generator still do the weaving.

Aspect-Oriented Modelling is currently a hot research topic. Behind the buzzword you can find such diverse things like adding an aspect notation to UML or describing (conceptual) frameworks for weaving models. Further examples can be found, for example, at an ECOOP workshop called “First Workshop on Models and Aspects - Handling Crosscutting Concerns in MDSD”, [MaA]

Consequences

Applicability. The approach can be applied only if the generator can handle various models with different concrete syntaxes and is able to perform the weaving.

Granularity. The achievable granularity is completely under the control of the developer, since the join point model is part of the (custom) metamodel definition.

Performance/Footprint. This pattern has no consequences for runtime performance and footprint, since it is applied at generation time.

Complexity/Verbosity. At first glimpse, the solution proposed by this pattern might seem overly complex. And in fact, if you use an unsuitable generator tool, the solution *can* be complex. However, if your generator really represents all models as objects in the implementation language once they are parsed, the implementation of this pattern becomes almost trivial. As a benefit, your application models will be well focused on a specific aspect, easier to maintain, and better usable in larger teams, since various developers can care of only selected aspects, and need not care too much about others.

Flexibility. This issue does not apply here, since the pattern has no runtime consequences.

Known Uses

All MDSD projects that I am or was involved in have used this approach, this includes a C-based component model for embedded real time systems, web applications and components for mobile devices.

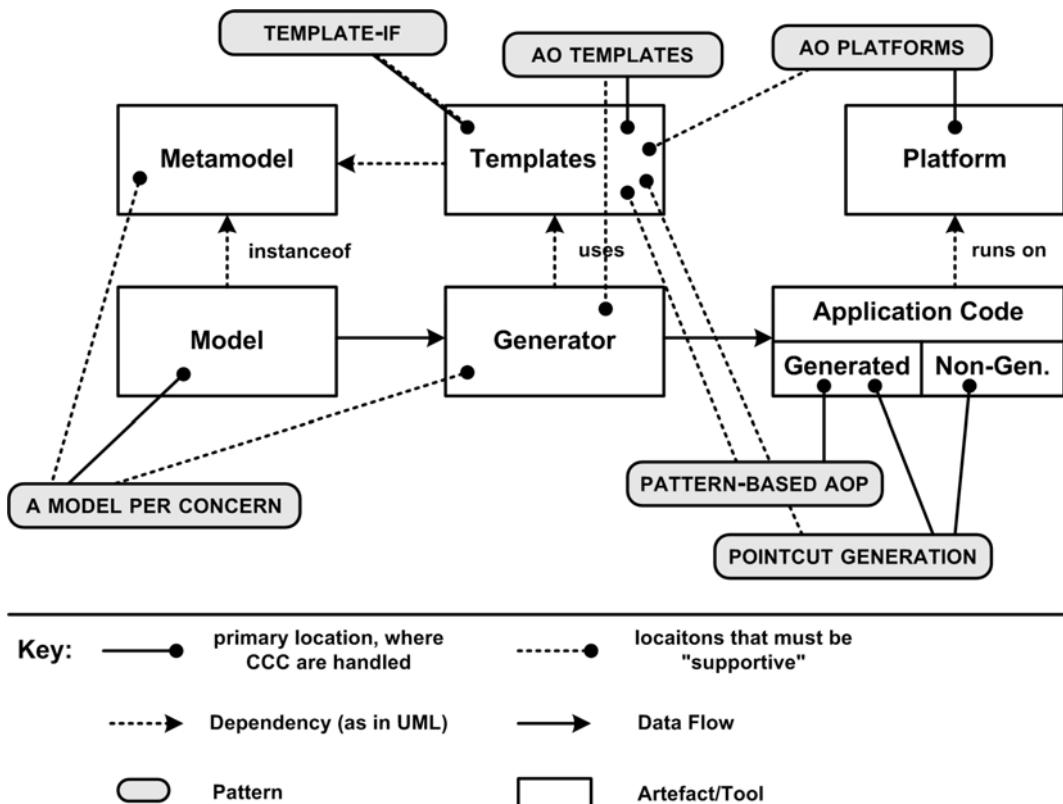
The documentation of the openArchitectureWare generator [OAW] shows an extensive practical example of using more than one model as generator input.

Summary

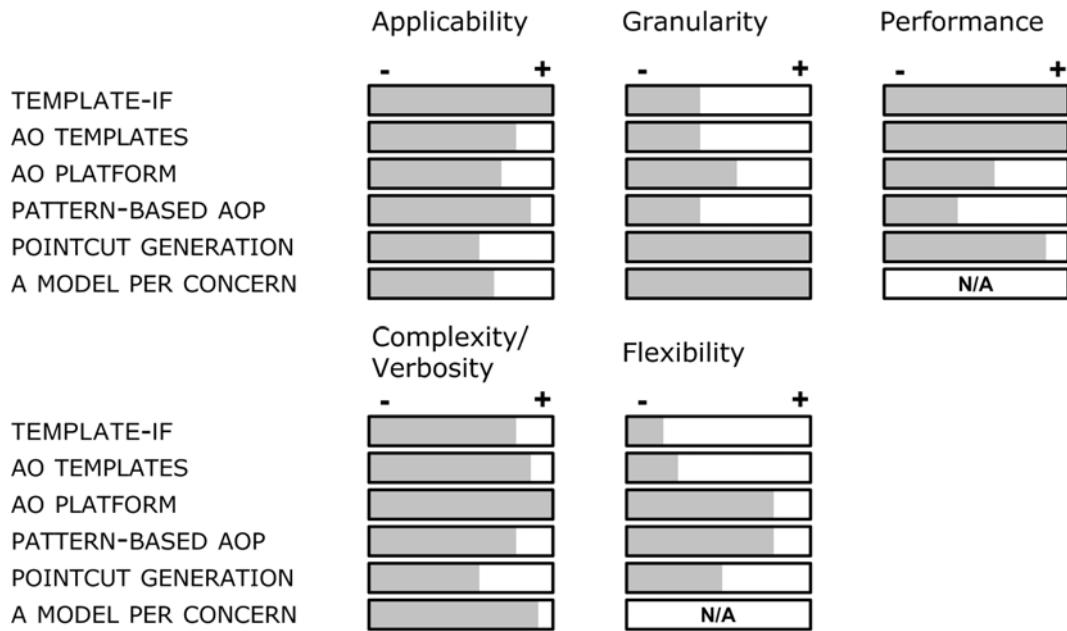
In non-trivial scenarios, A MODEL PER CONCERN is absolutely necessary to keep (large) models manageable. Make sure you use a tool where this approach can be implemented painlessly, before you use the generator tool on larger projects.

Pattern Overview – Pt.2

The following illustration shows where in an MDSD infrastructure the respective CCC-handling approach will take effect. For example, AO TEMPLATES handle the cross-cutting concerns in the templates, while the generator tool has to support it by providing the AOP support for template files.



This section provides a summary of the consequences in the form of a chart. The more grey in the box, the better. The rationale for the length of the bars is derived from the consequences sections of the respective patterns.



Where is the loom?

An important question is: where and when does the weaving happen? I have not included this question as a differentiator/consequence directly in the patterns, because in some way it is irrelevant – nobody cares where the weaving happens, as long as it happens as expected. However, from the perspective of the AO-minded reader the question is important, so I want to discuss it briefly in this section:

Pattern	Weaving Location
TEMPLATE-IF	No weaving happens – the advice are inlined into the template code.
AO TEMPLATES	The weaving is handled by the template engine, typically dynamically during template execution.
AO PLATFORMS	The platform takes care of weaving, typically at load time or runtime.
PATTERN-BASED AOP	The generator creates the proxies during system generation. Adding the interceptors (i.e. defining pointcuts) can happen during system startup or at any time during runtime.
POINTCUT GENERATION	The pointcuts are generated statically. The weaving happens in a separate weaving phase, at load time or at runtime, depending on the used AOP tool.
A MODEL PER CONCERN	The weaving is done by the code generator (acting as a model weaver) <i>before</i> code generation.

Relationships among the patterns

Some of the patterns in this paper are closely related or can be used together nicely. This section provides some detail.

PATTERN-BASED AOP and AO PLATFORMS

PATTERN-BASED AOP basically combines a couple of design patterns to implement an interception framework. The necessary proxies are created using code generation. You can "morph" this pattern to become an AO PLATFORM in the following way:

- use runtime code generation (see [CGLIB] for an example byte code modification library) to add the necessary proxies (or more general, hooks) to the system, for example during class load time.
- use a configuration file to define which interceptors should be used for a certain class.

You can then use this infrastructure as the AO PLATFORM for your application code.

AO PLATFORM and POINTCUT GENERATION

The boundaries between AO PLATFORMS and POINTCUT GENERATION seems to blur. In both cases, you use information from the model and generate artefacts that control the "aspect weaving" of some other tool; POINTCUT GENERATION generates the pointcut code for an abstract aspect of the platform. AO PLATFORMS use a platform's CCC handling techniques and generate the corresponding configuration files. While both approaches are conceptually very similar, they are quite different in practice with regards to granularity and other consequences.

There are clearly the two extremes:

- AspectJ is an AOP language extension for Java. Using it is definitely an instance of the POINTCUT GENERATION pattern.
- EJB 2.x are a – quite limited – AO PLATFORM. Deployment descriptors allow you to handle certain predefined CCC.

JBoss AOP are not as readily categorized into one of these two categories. You can define arbitrary advice (basically, by implementing interceptors) and then define a pointcut definition in a separate XML file. On the one hand it is POINTCUT GENERATION: you generate a pointcut (the XML file) that determines where to weave in prebuilt advice (the interceptors). On the other hand it is an AO PLATFORM, since it also comes with a set of predefined advice that are typically used in the relevant domain (enterprise systems).

The special case of A MODEL PER CONCERN

A MODEL PER CONCERN plays a somewhat special role in that it can be used together with any of the other patterns, since it takes care of CCC on the "input side" of the MDSD process. You cannot substitute this pattern by using an AOP language extension in the generated code.

Concluding Remarks

This paper presented a collection of patterns on how to handle cross-cutting concerns in the context of model-driven software development. The paper showed that using AOP language extensions is *one* way to do this, but not the only one – and considering the implications, the other approaches are sufficient in many environments. However, I don't want to create the impression that I consider AOP useless in the context of MDSD. There are scenarios, where combinations of MDSD and AOP can be very useful. Considering the added complexity of an AOP tool, however, I think the other approaches are worth considering.

If you just remember *one* of the patterns in this paper for your day to day work, then this should be A MODEL PER CONCERN. This has probably the farthest reaching (positive!) implications for MDSD.

Acknowledgements

I have received feedback for this paper from many people. These include Arno Haase, Danilo Beuche, Alexander Schmid, Eberhard Wolff. There are two people I want to thank specifically: Arno Schmidmeier (who was my shepherd for EuroPLoP 2005) and Christa Schwanninger. Both provided really good feedback that improved the paper substantially! I also want to thank my EuroPLoP 2005 workshop.

References

- [AC] AspectC++, <http://www.aspectc.org>
- [AJ] Eclipse.org, AspectJ homepage,
<http://www.eclipse.org/aspectJ>
- [AOP] aosd.net, <http://aosd.net>
- [CGLIB] Sourceforge.net, CGLIB Code Generation Library, <http://cglib.sourceforge.net/>
- [CME] Eclipse.org, The Concern Manipulation Environment, <http://www.eclipse.org/cme>
- [FF04] Filman, Friedman, Aspect-Oriented Programming is Quantification and
Obliviousness in Filman, Elrad, Clarke, Aksit, Aspect-Oriented Software
Development, Addison-Wesley, 2004
- [GoF] Gamma, Helm, Johnson, Vlissides; Design Patterns,
elements of reusable software design, Addison-Wesley 1995
- [LV04] Martin Lippert, Markus Völter. Die 5 Leben des AspectJ, JavaSpektrum 04/2004 und
<http://www.voelter.de/publications/articles.html>

- [MaA] Groher, Schwanninger, Völter, First Workshop on Models and Aspects - Handling Crosscutting Concerns in MDSD, ECOOP 2005, <http://www.st.informatik.tu-darmstadt.de:8080/ecoop2005/maw/>
- [MDSD] mdsd.info, <http://www.mdsd.info>
- [MO03] Mira Mezini, Klaus Ostermann, Conquering Aspects with Caesar, n (M. Aksit ed.) Proceedings of the 2nd International Conference on Aspect-Oriented Software Development (AOSD), 2003, ACM Press
- [MV03b] Markus Völter, A Generative Component Infrastructure for Embedded Systems, <http://www.voelter.de/data/pub/SmallComponents.pdf>
- [MV03] Markus Völter, Patterns for Program Generation, Proceedings of EuroPLoP 2003 and <http://www.voelter.de>
- [MV04] Markus Völter, Self Made EJB AOP, <http://www.voelter.de/data/articles/SelfMadeEJBAOP.pdf>
- [OAW] The openArchitectureWare Generator Framework, <http://www.openarchitectureware.org>
- [POSA2] D. C. Schmidt, M. Stal, H. Rohnert, and F. Buschmann. Pattern-Oriented Software Architecture - Patterns for Concurrent and Distributed Objects. Wiley and Sons Ltd., 2000
- [SH03] Arno Schmidmeier, Stefan Hanenberg, *AspectJ Patterns*, proceedings of EuroPLoP 2003
- [VB04] Völter, Bettin, Patterns for Model-Driven Software Development, Proceedings of EuroPLoP 2004 and <http://www.voelter.de>
- [VKS05] Völter, Kircher, Salzmann, Model Driven Software Development in the Context of Embedded Component Infrastructures, Springer, to be published
- [VKZ04] Völter, Kircher, Zdun, Remoting Patterns, Wiley 2004
- [VSW02] Völter, Schmid, Wolff, Server Component Patterns, Wiley 2002
- [XVCL] Sourceforge, XML Variant Configuration Language fxvcl.sourceforge.net/

Appendix A: AOP and MDSD

Before we actually look at the patterns of how to combine AOP and MDSD in practice, let us first define, what we understand by AOP and MDSD, respectively, and outline the commonalities as well as the differences of the two approaches.

What is MDSD

Model-Driven Software Development is about making models first class development artifacts as opposed to “just pictures”. Various aspects of a system are not programmed manually; rather they are specified using a suitable modelling language. These models are significantly more abstract than the implementation code that would have to be developed manually otherwise – the language for expressing these models is specific to the domain for which the models are relevant. The modelling languages used to describe such models are called domain-specific languages (DSL).

Models themselves are not useful in the final application, however. Rather, models have to be translated into executable code for a specific platform. Such a translation is implemented using model transformations. A model is transformed into another, typically more detailed (and thus, less abstract) model; a series of such transformations results in executable code, since the last transformation is typically a model-to-code transformation. Because of today’s somewhat limited tool support, many MDSD infrastructures use just one generation step, directly from the models to code. Model transformation tools using the latter approach are often referred to simply as model-driven code generators.

As can be seen from this introduction, I am primarily looking the generative approach of MDSD where models are translated into more concrete artefacts. Alternatively, models could be interpreted at runtime. However, in industrial practice, the interpretative approach is a niche; I will ignore it for the rest of this paper.

Also, all but the last patterns look at handling cross-cutting concerns in the context of the last of the transformation steps, namely code generation.

What is AOP

Aspect Orientation is about modularizing cross-cutting concerns (CCC) in software systems. CCCs are features of a system, that cannot easily be localized as a single module in a software system, because the abstractions and modularization facilities provided by the respective programming language (or system) do not allow such a modularization. Aspect Orientation uses various approaches to allow the modularization (and thus, localization) of such CCC. Aspect Oriented Programming (AOP) aims at introducing programming language constructs to handle the modularization of CCC.

The above explanation of AOP is what the mainstream considers AOP to be. There are, however, two additional "aspects" of AOP which I don't want to leave unmentioned: introductions and collaborations. Note that these two aspects are not as well known in industrial practice, and several AOP tools don't even support them. This paper will not address these two aspects in detail; at the end of a paper, there is a small section that provides some detail.

Note that, in order to simplify the discussion, throughout the paper I use AOP instead of differentiation when it's really about AOP and when the more general AO or AOSD terms should have been used.

Commonalities of the two approaches

Separating Concerns. Both approaches can be used to separate concerns in a software system. AOP typically modularizes CCC by separating them into aspects and later weaving them into the "normal" code (source or binary). MDSD works by specifying system functionality in a more abstract, and domain specific DSL and the transformations are used to add those concerns that can be derived from the model's content.

Technical Aspects. Both approaches are often used to factor out (and then later, reintegrate) repetitive, often technical aspects. In both cases it is also possible to factor out function (or domain-specific) aspects, although this is not widely used – usually, because technical aspects are more obvious and well-understood candidates.

Mechanics. Technically, both approaches work with queries and transformations⁴ (see [FF04]). In AOP you use pointcuts to select a number of relevant points (join points) in the execution of a program (or in its code structure) and "contribute" additional functionality called advice at these points. In MDSD, a model transformation selects a subset (or pattern) of the model, and transforms this subset into some other model.

Metamodels. Metamodels play an important role in both approaches. In MDSD, the metamodel⁵ is clearly evident, as it forms the foundation of the model that is being transformed. In model-2-model transformations, the metamodel of the transformation target is also relevant, whereas model-2-code transformations typically use textual templates to generate the target code. In AOP, the metamodel is not so readily obvious. However, the join point model of the particular AOP system is also a metamodel. A specific program (or program execution) is an

⁴ Quantification is an important concept in both approaches. Statements like "foreach model element of type X generate the following code" or "whenever the method X is called by a class in package Y" allow to quantify over a number of things, as opposed to addressing each occurrence specifically. This is what makes both approaches so powerful.

⁵ The metamodel is a model that describes a model. Instances of metamodel elements are the elements of a model. A model is related to its metamodel via an *instanceof* relationship. In the context of a DSL, the metamodel plays the role of the abstract syntax. In addition, like any other language, a DSL also has a concrete syntax and semantics.

instance of this metamodel by exhibiting the occurrence (or instantiations) of the respective join points.

Selective Use. An important concept in both approaches is the fact that the handling of CCC can be turned on or off for a specific system. In AOP, you can decide at weaving time whether you want to have a certain aspect included in the system⁶. In MDSD, you can select the transformation you want to use for a specific system – the chosen transformation may or may not address a certain concern.

Differences

Dynamic vs. Static. MDSD works by transforming static models⁷. That means, MDSD transformations work before the system is run at generation time (remember that we ignore the runtime interpretation of models in this paper), MDSD has no relevance during the execution of the system – you cannot tell that a system has been built by using MDSD by examining the finished system. AOP, on the other hand, contributes behaviour to points in the *execution* of a system. In many systems it is therefore possible, to consider dynamic aspects in the definitions of pointcuts (such as the current call stack; an example is AspectJ, [AJ])⁸.

Invasiveness. MDSD needs to be used during the development of the software system, since the (finished) system is obtained by transforming models into code. It is not possible to benefit from MDSD *after* a system has been developed⁹. With AOP, however, it is (in most systems) possible to introduce behaviour after the base system has been developed completely. This non-invasiveness is a key advantage of AOP, since aspects can be added to a system after the fact¹⁰.

Abstraction Level. A fundamental concept of MDSD is that it allows developers to express their intent with regard to the software system on a higher abstraction level, more closely aligned with the problem domain. The specifications are thus more appealing to domain specialist, compared to 3GL code. A DSL serves exactly this purpose. AOP, on the other side, is basically bound to the abstraction

⁶ ... which can be at runtime if you use an AOP system with dynamic weaving.

⁷ Of course these models can describe behaviour. Also, the generated code can include behavioural aspects. However, the model itself is always a static structure, and the transformations transform this static structure into a different static structure.

⁸ Note that this dynamic behaviour has nothing to do with when the weaving is done. Weaving can be done before runtime, at load time and at runtime. The latter is called dynamic weaving. In case weaving is done before runtime ("static weaving") the dynamic nature of aspects is achieved by statically generating code!

⁹ MDSD can be used to create wrappers for or adapters to legacy systems. In that case, however, the legacy system itself is untouched.

¹⁰ Of course, if you want to take full advantage of AOP, you have to design for it. Applying aspects after the fact is an interesting and important use case (for adapting existing systems), but the full potential of AO can only be exploited if you use aspects from the very beginning. See [LV04]

level of the system for which it handles the CCC; in AOP, this is the base programming language. While AOP can of course be used to more concisely express relationships, collaborations or other concerns of the underlying base program, there is no fundamental change to the level of abstraction of the domain specific-ness of the constructs.

Non-Programming Language Artifacts. In MDSD, it is easily possible to also generate non-programming language artefacts such as configuration files, build scripts or documentation; this is because in model-2-code transformations, any textual artefact can be created. AOP however works on the running system (remember it is dynamic in nature), and as such it cannot affect things that are not relevant at runtime¹¹ (or said differently: things that are not expressed in the programming language).

¹¹ In the context of the CME project [CME], it will be possible to work with concerns outside of the running program, for example in ant build files.

Appendix B: Introductions and Collaborations

This paper has looked at AOP primarily at a way to contribute advice to join points during program execution. As mentioned in the introduction of the paper, there are, however, two additional aspects of AOP for which I want to provide some detail in this section.

Introductions/Open Classes

Introductions are used to "inject" artefacts into an existing system statically (as opposed to dynamic advice in join points). For example, additional fields or operations can be introduced into existing classes. Typically, the pointcut query language (specifically, its static subset) is used to determine, where to inject artefacts. An example could be "for all classes that implement interface X, add the following methods:"). Some AOP languages also allow to *change* static class features, such as changing the superclass, or adding an additional implemented interface; this is called open classes. The following table provides some detail, on how this relates to the above patterns.

Pattern	Introductions	Open Classes
TEMPLATE-IF	Using a template-IF, it is easy to conditionally inject code into the generated artefacts.	
AO TEMPLATES	It is possible to add template code to existing templates. The explicitly defined hooks shown above can be considered to be an introduction.	N/A
AO PLATFORMS	Depends on the platform, not widely supported. Some allow it by using tools such as byte-code modification (Spring is an example).	
PATTERN-BASED AOP		Not possible.
POINTCUT GENERATION		Depends on the AOP tool used. If you use AspectJ, for example, both features are possible.
A MODEL PER CONCERN	Using on the fly model modifications, it is common practice to add additional features to model elements.	N/A

Collaborations

The idea of considering a collaboration between artefacts an aspect that is worth modularizing is very appealing. Using this approach, a collaboration becomes a *type*, in the same way as classes or aspects (as we know them traditionally) are types. This "kind" of AOP is very important for handling functional aspects as opposed to technical aspects but is still subject for research and not widely used in practice. The CAESAR [MO03] language is an example of a research project in this direction.

The idea of this approach is basically, that you define a collaboration (for example, the Observer pattern) as a type. The collaboration type describes the various roles that are required for the collaboration (in the example, Subject and Observer). In a concrete system, pointcuts are used to instantiate the collaboration by binding concrete artefacts to the instantiated collaboration (for example, in a drawing programm, the class Figure plays the role of the subject, while the Canvas plays the role of the Observer).

The collaboration also defines, which features artefacts must have in order to be able to play a role (for example, they must implement the ISubject interface, or have an operation registerObserver()). Traditional AOP introductions and advice can be used to enhance the concrete artefacts in order for them to play the role.

Using this approach, you can capture large portions of collaboration code in well-separated aspects and then simply bind concrete artefacts to instances of these collaborations. "Traditional" AOP mechanics are used to fill in the required "magic".

What is the relationship of the patterns introduced above to collaboration aspects? It is safe to say, that the patterns do not address this aspect, with two exceptions:

- You can extend the POINTCUT GENERATION pattern so that you generate collaboration bindings in case the underlying AOP language supports this.
- In the A MODEL PER CONCERN pattern, you can use markup in models (such as tagged values in UML models) to mark certain artefacts as playing a certain role in a collaboration. The generator can then make sure the model artefact has all the required features, or use model modifications to actually add them. Later stages can make sure the generated code can play the collaboration role.