AOSD 06
Tutorial Submission

About the Presenter

Name: Markus Voelter

Contact information:

Markus Voelter

Independent Consultant

Ziegelaecker 11, 89520 Heidenheim, Germany

voelter@acm.org

www.voelter.de

mobile: +49 171 86 01 869

Brief biography:

Markus Völter works as an independent consultant and coach for software technology and engineering. He focuses on software architecture, middleware as well as model-driven software development. Markus is the author of several magazine articles, patterns and books on middleware and model-driven software development. He is a regular speaker at conferences world wide. Markus can be reached at voelter at acm dot org via or www.voelter.de

Summary of teaching experience

I have given various tutorials and presentations at conferences worldwide (with a focus in Germany); among them OOP, JAOO, JAX, OOPSLA and ECOOP. A complete listing can be found at www.voelter.de/conferences. I have also given lectures at the University of Applied Sciences, Ulm.

Typically the tutorials are quite well received and the feedback is accordingly positive.

Primary contact: Markus Voelter

About the Tutorial

Title: Models and Aspects – Handling Cross-Cutting Concerns in Model-Driven Software Development

Half-day or full-day: Half Day

Abstract

Aspect Oriented Software Development (AOSD) as well as Model-Driven Software Development (MSDD) are both becoming more and more important in modern software engineering. Both approaches attack important problems of traditional software development. AOSD 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 tutorial looks at both of these questions. The first one – how AOSD and MDSD relate – is discussed in the first part of the tutorial.

The second, and main part of the tutorial introduces six patterns of how MDSD and AOSD can be used together. These include:

- Using Code Generation Templates to handle CCC
- Using AO techniques in template languages
- Using the platform to handle CCC
- Using suitable, pattern-based design to handle CCC
- Generating pointcuts to be woven with the system
- Using aspect orientation to structure models (AO modelling)

All the patterns are illustrated with practical real-world examples taken from various model-driven software development projects.

Expected audience:

Software Developers and Architects

Level of the tutorial

intermediate

Prerequisites for participants

Basic Understanding of AO concepts as well as MDSD/MDA techniques

Synopsis

AOSD and MDSD are both becoming more and more important in the software engineering world. Both approaches attract a big crowd of followers. However, the interactions between the two groups are limited. One can often find discussions whether MDSD is a special case of AOSD, or vice versa. In real-world projects, you can find discussions about whether to use MDSD *or* AOP.

So, this tuturial has the following goals:

- Show the common aspects of AOSD and MDSD, as well as their differences in order to understand that both are complementary, not opposites.
- Show proven ways how both can be used together; specifically show the role AOSD can play in an MDSD/MDA environment

Note that the tutorial's contents are *not research*! All of the patterns are mined from actual, practical work in real world projects.

The outline and schedule of the tutorial is given in the following table. A more elaborate discussion of the tutorial's content can be found from page 5 of this submission, onward.

For even more details, see the tutorial slides at

www.voelter.de/data/presentations/ModelsAndAspects.zip

Topic	Duration [min]
What is MDSD: Brief Intro to the topic	10
What is AOP: Brief intro to the topic	10
Commonalitities and Differences: Outlines the conceptual and practical commonalities and differences of the two approaches.	30
Forces: Outlines the forces that need to be taken into account when considering how to handle CCC in MDSD	10
Patterns: This section contains the 6 core patterns; this is the "meat" of the talk. The patterns are: Templates-Based AOP AO Templates AO Platforms Pattern-Based AOP Pointcut Generation AO Modelling	80
Pattern Relationships: This topic looks at some of the relationships between the patterns, i.e. if and how some of the patterns can be combined	15
Introductions and Collaborations: The tutorial looks primarily at before/after/around-kind of things. However, AOP also addresses introductions as well as collaborations. This section explains these two aspects.	15
Overview and Summary Wraps up the tutorial	10

Previous venues at which this tutorial has been presented

JAX 2005, OOPSLA 2005, ECMDA-FA 2005

At JAX I had roughly 60 participants. The rating averaged between very good and good; I happen to still have the feedback, see below. For OOPSLA and ECMDA-FA, I don't have feedback yet since the tutorial has not yet been given.

(1=very good, 2=good, 3=ok, 4=acceptable, 5=bad)

Overall
1:44.83%
2:41.38%
3:13.79%
4:0.00%
5:0.00%

Was the topic covered well?

1:51.72%
2:44.83%
3:0.00%
4:0.00%
5:3.45%

Speaker's technical competence

1:75.86%
2:20.69%
3:0.00%
4:0.00%
5:3.45%

Style of presentation

1:48.28%
2:34.48%
3:10.34%
4:3.45%
5: 3.45%

Required equipment for presentation

Overhead projector, flip charts, microphone

Required equipment for participants

None.

Material

You can download the slides for the JAX session from:

www.voelter.de/data/presentations/ModelsAndAspects.zip

The tutorial is based on a patterns paper that I wrote for EuroPLoP 2005. It can be downloaded from:

www.voelter.de/data/pub/ModelsAndAspects.pdf

SYNOPSIS CONTINUED

Introduction to AOP and MDSD

Before we actually look at the patterns of how to combine AOSD and MDSD in practice, let us first define, what we understand by AOSD 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.

What is AOSD

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 AOSD is what the mainstream considers AOSD to be. There are, however, two additional "aspects" of AOSD which I don't want to leave unmentioned: introductions and collaborations. Note that these two aspects are not ass well known in industrial practice, and several AOSD 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.

Commonalities of the two approaches

Separating Concerns. Both approaches can be used to separate concerns in a software system. AOSD 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 transformations[1] (see [FF04]). In AOSD 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 metamodel 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 AOSD, 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 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 of 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 system. 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 models. 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. AOSD, 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]).

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 developed. 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 fact.

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. AOSD, on the other side, is basically bound to the abstraction 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 runtime (or said differently: things that are not expressed in the programming language).

General Problem statement

As we have seen in the previous sections, there are a number of commonalities between AOSD and MDSD. As a consequence, developers often don't know whether, or how they should relate AOSD and MDSD. Should they use either AOSD or MDSD? Is AOSD or MDSD a more general approach? Is MDSD a special case of AOSD? 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 efficiently in an MDSD-based development environment?**

An author with a stronger background in the AOSD domain might have written the paper from the other perspective, addressing problems such as "how can domain-specific notations used in AOSD", or "how can AOSD 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. 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 complexity 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 Overview

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

Template-inherent AOP: 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.

How to handle CCC? 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.

AO Templates: In some cases, especially if you're building related families of code generators, using TEMPLATE-INHERENT AOP 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.

How to handle CCC? Use an AO approach on template level. Rather that using template-level *if* statements, use an "aspect template" that advices 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.

AO Platforms: 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.

How to handle CCC? 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.

Pattern-Based AOP: 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.

How to handle CCC? 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.

Pointcut Generation: 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?

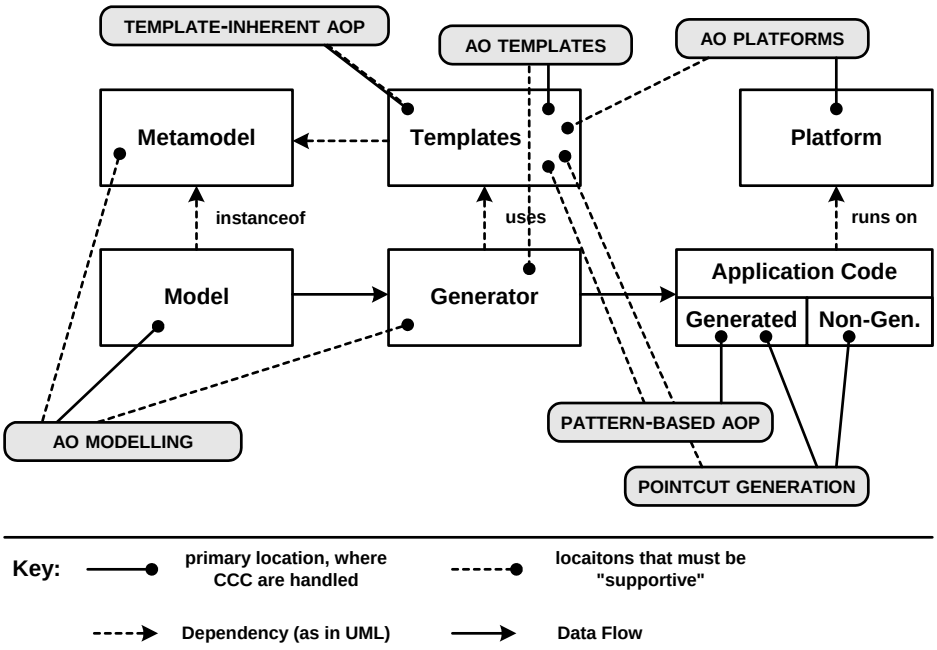
How to handle CCC? Integrate an AOP language into the MDSD software development infrastructure. Specifically, define a number of prebuilt advice as part of the platform, and then generate the pointcut based on specifications in the model. Use the AOP language's standard weaver to integrate the aspects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

AO Modelling: 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

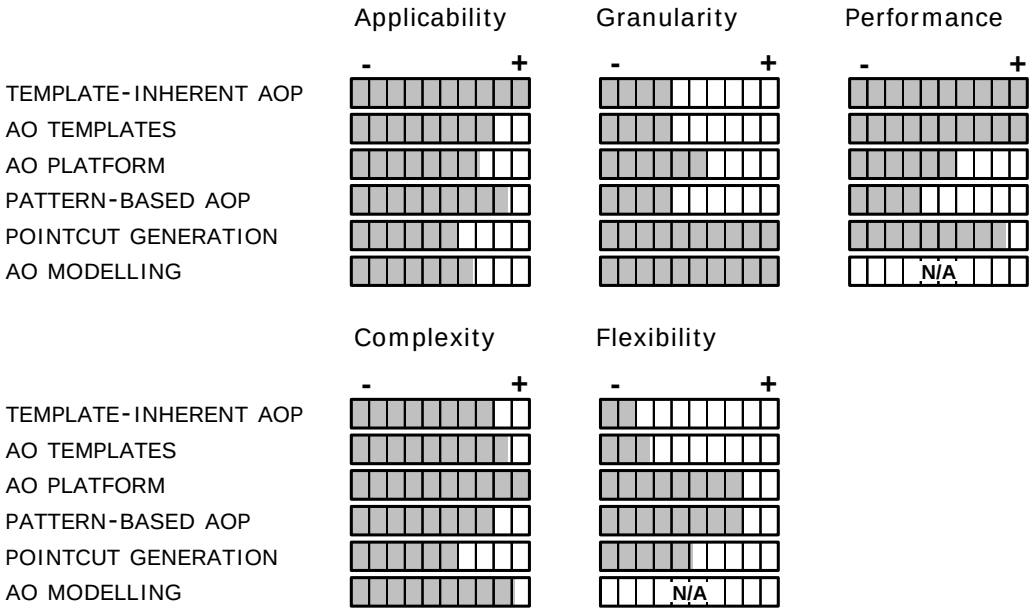
code generation is used to create the final application. In many scenarios, however, it is necessary to separate concerns in the application *models*, too!

How to handle CCC? 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.

The following illustration shows where in an MDSD infrastructure the respective CCC-handling approach will take effect.



The following diagram 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 should be obvious from the consequences sections of the respective patterns.


Applicability
Granularity
Performance
- +
TEMPLATE-INHERENT AOP
AO TEMPLATES
AO PLATFORM
PATTERN-BASED AOP
POINTCUT GENERATION
AO MODELLING
N/A
Complexity
Flexibility
- +
TEMPLATE-INHERENT AOP
AO TEMPLATES
AO PLATFORM
PATTERN-BASED AOP
POINTCUT GENERATION
AO MODELLING
N/A