

# Cross-Cutting Concerns in MDSD

Aka: AOP & MDSD



[www.mdsd-buch.de](http://www.mdsd-buch.de)



[www.mdsd-book.org](http://www.mdsd-book.org)

**Markus Völter**

[voelter@acm.org](mailto:voelter@acm.org)

[www.voelter.de](http://www.voelter.de)



Ingenieurbüro für Softwaretechnologie

[www.voelter.de](http://www.voelter.de)

- 1 -

© 2003 - 2006 Markus Völter

## TEMPLATE INHERENT AOP

- You are using a template-based code generator. 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.** This, creates a method signature and skeleton implementation for each Operation in a model.

```

«DEFINE OperationDef FOR Operation»
public final «ReturnType» «Name» (
    «FOREACH Parameter AS p EXPAND USING SEPARATOR ", "»
    «p.Type.QualifiedJavaTypeName» «p.Name»
    «ENDFOREACH» ) {
    return «Name»Internal(
        «FOREACH Parameter AS p EXPAND USING SEPARATOR ", "»
        «p.Name»
        «ENDFOREACH» );
    }
«ENDEDEFINE»
  
```



Ingenieurbüro für Softwaretechnologie

[www.voelter.de](http://www.voelter.de)

- 2 -

© 2003 - 2006 Markus Völter

## TEMPLATE INHERENT AOP II

- 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 ... );
  }
«ENDEFINE»

```

## AO TEMPLATES

- 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.
- **Example.**

```

«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 ... );
  }
«ENDEFINE»

```

## AO TEMPLATES II

- Use an AO 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.
- **Example.** The following piece of code 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;
}
«ENDEDEFINE»
```

- 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.



## AO TEMPLATES III

- **Example cont'd.**

```
«DEFINE LoggingMethodBegin FOR Operation AFTER HookMethodBegin»
    «IF loggingRequired»
        // entering method such and such
    «ENDIF »
«ENDEDEFINE»

«DEFINE LoggingMethodEnd FOR Operation AFTER HookMethodEnd»
    «IF loggingRequired»
        // leaving method such and such
    «ENDIF »
«ENDEDEFINE»
```



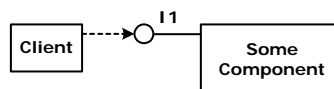
## AO PLATFORMS

- Use the CCC-handling capabilities of the middleware as far as possible.
  - Use the code generator to **generate the annotations** 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.
- **Example.** Many MDS tools in the context of EJB require developers to develop POJOs that contain the business logic.
- The generator then creates “EJB wrappers” that ensure the POJOs conform to the constraints defined by EJB.
- The generator also creates a deployment descriptor to control the EJB container.



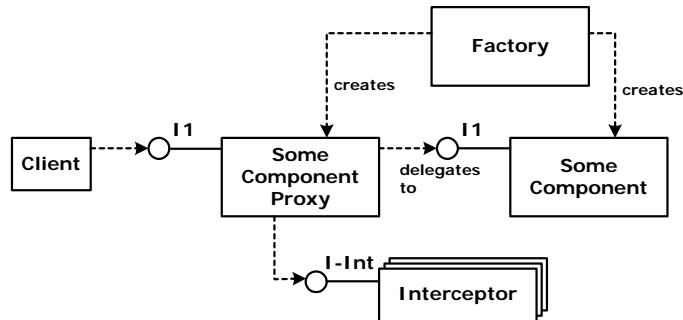
## PATTERN-BASED AO II

- 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.
- Consider you face the following situation:



## PATTERN-BASED AO III

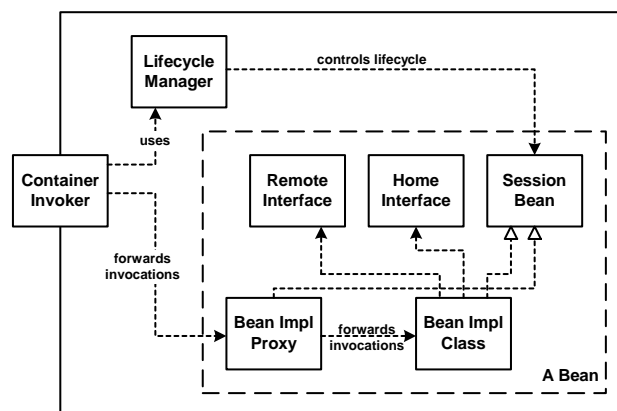
- You can replace this setup by the following:



- 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.

## PATTERN-BASED AO IV

- 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.



## POINTCUT GENERATION

- Integrate an **AOP language** into the MDS software development infrastructure.
  - Define a number of **pre-built advice**, part of the platform
  - **Generate the pointcut** based on specifications in the model.
  - **Use** the AOP language's **standard weaver** to integrate the aspects with the generated code.
- **Example.** XML below is part of the model for a node and container in the distributed, embedded system
  - Tracing option is set to *app*, i.e. we want to trace application level operations. (DSL-specific pointcut def.)

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

## POINTCUT GENERATION II

- **Example cont'd.** As part of the platform, you define the following abstract aspect (using the AspectJ language). It does not define a pointcut, it is thus "pure advice".

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

- For every container that has tracing set to *app*, code like the following (a pointcut!) is generated:

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

## AO MODELLING

- Up to now, we were mainly concerned with **handling CCC in the resulting application**, which would be built using an MDS approach.
  - App 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.



## AO MODELLING II

- 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.



## AO MODELLING III

- **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) a **Visio diagram** to describe form layout.
- Each of these **four models** 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.

## AO MODELLING IV

- **Example cont'd.** Metamodels and their connections.

